OPTIMIZATION METHODS

→ minimize or maximize a fn
→ minimization = − maximization

FIRST ORDER OPTIMIZATION

Uses first derivatives (of loss fn w.r.t parameters

Examples: Gradient Descent methods
Conjugate gradient

SECOND ORDER OPTIMIZATION

Uses second derivative (of loss fn wrt. parameters

Examples: Newton's Method
Gauss Newton
Quasi Newton
(L)BFGS

CONSTRAINED OPTIMIZATION: In optimization, we want to maximize or minimize a function $f(x)$ over all possible values of x. When we put a constraint on x i.e. if it can only assume values that satisfy a condition / lie in a subset of x values → constrained optimization

E.g. When regularization term is added to Loss fn → limits the search space for W's
= Constrained optimization

JACOBIAN MATRIX: Matrix of derivatives of loss fn w.r.t. params

$f: R^n \to R^m$ (f: loss function)

I/P: vector $x \in R^n$
O/P: vector $f(x) \in R^m$

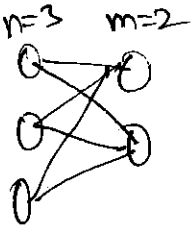
If m=1 !

$$J = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}$$

If m=2

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

dim same as wt. matrix



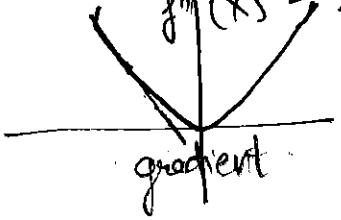
SECOND ORDER OPTIMIZATION

Second derivative —provides info→ curvature of fn

E.g. $f(x) = x^2$

$f'(x) = 2x$

$f''(x) = 2 \Rightarrow$ +ve curvature

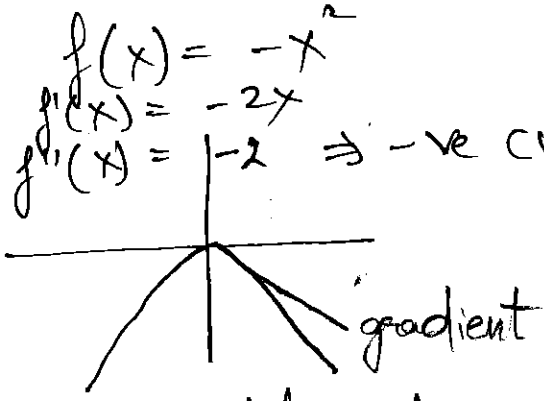


Actual fn decreases slower than what gradient predicts

$f(x) = -x^2$

$f'(x) = -2x$

$f''(x) = -2 \Rightarrow$ -ve curvature



Actual fn decreases faster than what gradient predicts

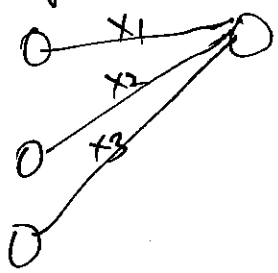
$\Rightarrow$ Second derivative tells whether a gradient step will cause as much of an improvement as we would expect based on gradient alone

→ Second order optimization requires computation of ~~Hessian derivative~~ matrix

$f: R^n \to R$ I/P is a vector $x \in R^n$

O/P is scalar $f(x) \in R$

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$



WHY SECOND ORDER METHODS NOT PREFERRED:

1) Computation of second derivative of all i/p wt. pairs → expensive operation

2) Newton method can get stuck in saddle points (whereas GD flavors can get out of it) though can be mitigated with regularization

3) Soln requires inverting Hessian → computationally expensive & Memory-Intensive

SMOOTH VS NON-SMOOTH FNS:

SMOOTH FN: → A fn is said to be smooth of order n if it has "unique" n^{th} derivative" at every point in the domain of fn. Order Denoted by C^n.

→ They are continuous by definition but vice-versa is not true (i.e. continuous fns may not be smooth

→ In short, smooth fns are differentiable and continuous

NON-SMOOTH FN: Non-differentiable at ~~any~~ point in domain of fn

~~Dis~~continuous at any point in domain of fn

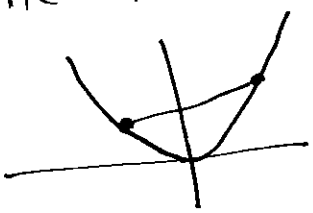
e.g. ~~xxx~~ gaps in fn or sharp bends

← |x| : Non smooth fn (not differentiable at 0 as left limit ≠ right limit)

∴ GD can be applied to smooth fns (or differentiable fns)

CONVEX FNS VS NON CONVEX FNS:

CONVEX FN: → A fn is said to be convex if at any two points on the fn a line is drawn, the line is above or on the graph of fn



→ $f''(x) \geq 0$ for all x in the domain of $f(x)$

→ global minima = local minima and no saddle points

→ A fn is concave if $-f$ is convex

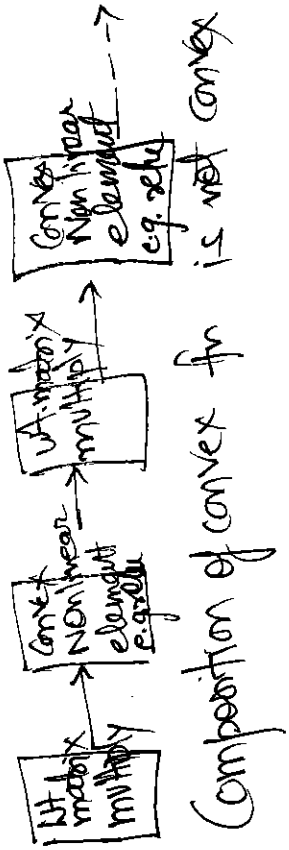
→ Traditional ML loss fns ~~are~~ convex loss fns to simplify optimization

MSE
Hinge loss
Cross-Entropy
⇕
logistic loss
(logistic loss is not sigmoid)

} → all convex

but sigmoid fn → non-convex
(∵ it is activation fn and not loss fn)

→ The above convex fns when used in DL with two or more layers become non-convex. This is because parameters of the later layers (wts & activation) can be highly recursive fn of parameters in previous layers. Recursive structure like this tends to destroy convexity.



NON-CONVEX FN = ! CONVEX FN + global minima != local minima + saddle points
+ ! CONCAVE FN

GRADIENT DESCENT METHODS APPLIED TO CONVEX VS. NON-CONVEX LOSS FNS

- GD methods applied for loss fn optimization will eventually converge to a stationary point of fn regardless of convexity.
  If fn is convex → global/local minima. If non-convex → local minima

Convex fns: global minima = local minima
× saddle points

- Thus GD methods (with a suitable step size/learning rate) is guaranteed to find a global minimizer

SGD or mini-batch GD might keep over shooting the global minima with a constant learning rate.
However, with a decaying learning rate
= same convergence as batch gradient descent

Curves "up & down" – neither concave nor convex. e.g. sine fn

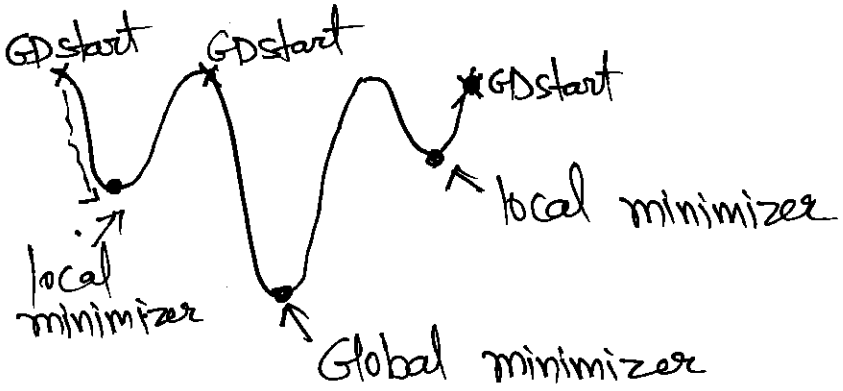
Convex → π to 2π
Concave → 0 to π

Non-convex fns: global minima ≠ local minima
✓ saddle points

- Finding global minimizer for non-convex loss fn is NP-hard (non deterministic polynomial time) and one settles with local minimizer (i.e. GD methods) → for practical purposes this is good enough

- Thus GD methods (with suitable step size/learning rate) is guaranteed to find a local minimizer

- Mini-batch GD might keep over-shooting local minima with const. step size.
  Soln: Decay learning rate

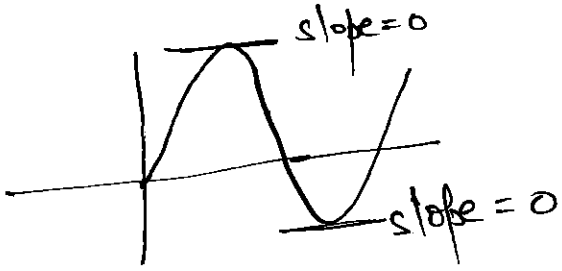


← Non-convex loss fn

SADDLE POINTS IN NON-CONVEX LOSS SURFACE :

Saddle Pt / Minimax pt : 1) slopes in orthogonal dir = 0 at the point
2) Pt ≠ local minima / local maxima of fn

Recall in 1-dim graphs when $f'(x) = 0$
⇒ local minima or local maxima



e.g for $f(x) = x_1^2 - x_2^2$ (3 dimensional surface hyperbolic paraboloid)

(0,0) is saddle point

1) $\frac{d}{dx_1} f(x) = 2x_1$
at (0,0) $= 0$

2) $\frac{d}{dx_2} f(x) = -2x_2$
$= 0$

∴ slopes = 0

2) (0,0) is not local minima as clearly (0,ϵ) for $f(x)$ has lower value than (0,0)

Ways to escape saddle points: It has been shown that

(1) Noisy gradient : Add gaussian noise to gradient

$$w_t = w_t - \alpha\left(\frac{\partial L}{\partial w_t} + N(\mu, \sigma^2)\right)$$

(2) Random choice of initial values of parameters makes GD methods not stuck in saddle points

∴ When GD methods are used to optimize non convex loss fn:

1) In low dimensions — Local minima more common
2) In high dimensions — Saddle points more common

SPECIALIZED METHODS FOR SOLVING NON-CONVEX PROBLEMS:

1) Alternating minimization methods
2) Branch-and-bound methods

But these are not very popular for machine learning problems

GRADIENT DESCENT

Loss FN

Convex (traditional ML loss fns)
- global minima
- no saddle pts

Non-convex (DL)
(due to recursive nature of NN + non-linear act. fns (even traditional convex fns become non-convex))
- local minima
(finding global minima is NP-hard)
- saddle pts
  - To avoid saddle pt
    - i) Add noise to gradient (e.g. gaussian noise)
    - ii) Any reasonable Random initialization of params
- Adjusting learning rate
  - Soln: optimizers such as RMSProp, Adam

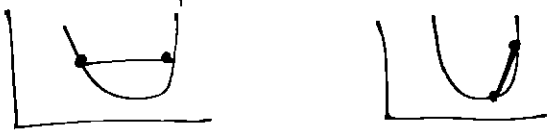
GRADIENT DESCENT

- Used to solve problems which are convex in nature
- If the loss fn is convex, local minima = global minima (no saddle points)

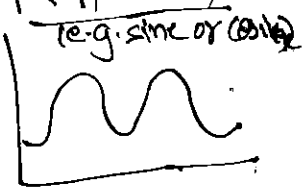
CONVEX FNS: 1) Draw a line using any two points on curve — above or on the curve

2) $f''(x) \geq 0$

E.g. quadratic and exponential fn are convex fns



Non-Convex (e.g. sine or cosine)



Advantage of SGD

- SGD is a computationally efficient way to solve problems involving convex loss functions e.g. in Linear Regression, Logistic Regression & SVM.

Computationally Efficient (Time complexity) — linear in the number of training examples

If X is a matrix of size (n, p) (n: no. of obs, p: no. of features)

then training cost $O(knp)$ (k: no. of epochs)

Hence, scales well

Disadvantages of SGD

1) Can get stuck at local minima, instead of global one (if the cost fn is non-convex)
   - soln → run it with different initial wts & hyperparameters (learning schedule)
2) Sensitive to feature scaling (if the features are on different scales, takes long time to converge)
3) A number of hyperparameters to tune such as regularization parameter & no. of iterations

EXAMPLE OF GRADIENT DESCENT (Batch GD)

(for Regression)

a	b	X	Y	YP = a+bX	SSE	$\frac{\partial SSE}{\partial a}$ = -(Y-YP)	$\frac{\partial SSE}{\partial b}$ = -(Y-YP)X
0.45	0.75	0.00	0.00	0.45	0.101	0.45	0.00

① Start with random values of a, b

Total SSE = 0.667 Sum = 3.3 Sum = 1.545

$$SSE = \frac{1}{2} \sum (Y - YP)^2$$

② ½ for mathematical convenience as it helps in calculating gradient easily

Update Rules: New a = a - learning rate $\times \frac{\partial SSE}{\partial a}$

$= 0.45 - 0.01 \times 3.30 = 0.42$

New b = b - learning rate $\times \frac{\partial SSE}{\partial b}$

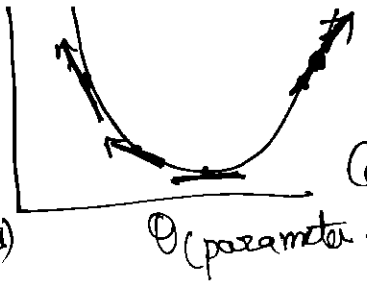
$= 0.75 - 0.01 \times 1.545 = 0.73$

③ Use new a & b to calculate new Total SSE

If total SSE goes down ⇒ pred. accuracy is improved

CONCEPTUAL WAY OF UNDERSTANDING GRADIENT DESCENT

If we are at any pt. of this cost fn, in order to reach min, we need to know (cost)
i) dir in which to go (gradient)
ii) steps to take (learning rate)



Gradient descent helps in both

Move in opposite direction to gradient (slope) to minimize cost fn

Coeff. value step (new) = old value − learning rate × gradient

TYPES OF GRADIENT DESCENT:

1) BATCH GD: a) Wts are updated after going thru all training examples (multiple passes on full training set are required)
↓ guaranteed to converge to global minimum → convex error surfaces
b) Deterministic: Every run results in same minima, irrespective of initial wts given same training examples
local minimum → non-convex surfaces
c) Takes long time to converge, if training examples are many
d) Final param values are optimum (for convex loss fns)
e) Cannot be used if whole dataset does not fit in memory

2) STOCHASTIC GD: a) Wts are updated after every training example (multiple passes required but less than batch GD)
↓ It has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch GD ie. almost certainly global → convex min, local → non-convex min
In reality SGD is implemented as mini batch
[random sample from training data are chosen in each run to update params]
b) Different runs may yield different values depending on initial wt. assignment
c) Takes faster to converge → error fn is not as well minimized as GD.
d) Final parameter values are close to optimum, but not optimum, since it keeps overshooting (+path to optimum is much noisier) (decay in learning rate alleviates this issue)

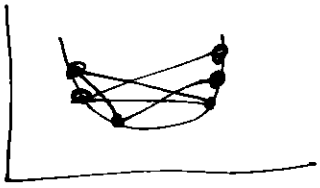
3) MINI BATCH GD: a) Wts are updated after each mini batch of training examples

Adv. 1) Averages out noise (noisy sample in SGD can cause fluctuation in wts) compared to SGD
2) Final parameter values are closer to minimum than SGD

LEARNING RATE IN GRADIENT DESCENT

1) If learning rate is small → can take a lot of time to converge

2) If learning rate is high → parameter values & cost fn can fluctuate heavily

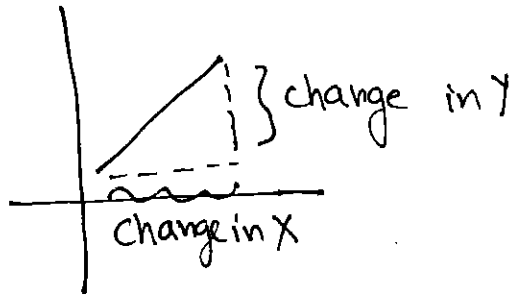


One solution to this problem is gradually reduce the learning rate → simulated annealing: The steps start out large (to make quick progress & escape local minima) then get smaller & smaller, allowing the algorithm to settle at the global minimum.

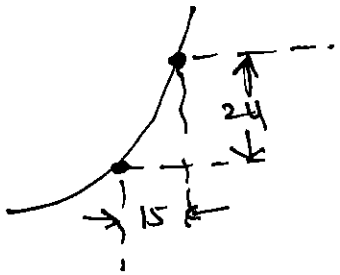
↑
learning schedule

DERIVATIVES

Slope = $\frac{\text{Change in } Y}{\text{Change in } X}$



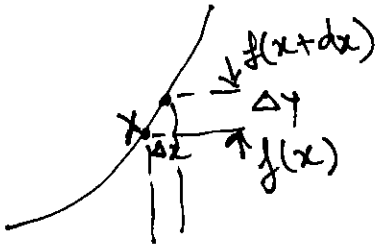
Avg. slope b/w 2 pts



$= \frac{24}{15}$

Slope at a point (since no Δy or Δx) → Derivatives

We use a small difference & then have it shrink towards zero



$$\frac{dy}{dx} = y'(x) = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

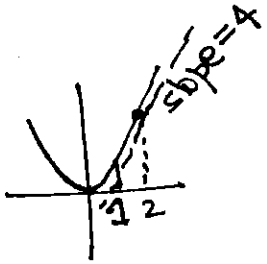
E.g. $$\frac{d}{dx}(x^2) = \frac{(x+\Delta x)^2 - (x)^2}{dx}$$

$$= \frac{x^2 + 2x\,dx + dx^2 - x^2}{dx}$$

$$= 2x + dx$$

Now $dx \to 0$ $\therefore$ $= 2x$

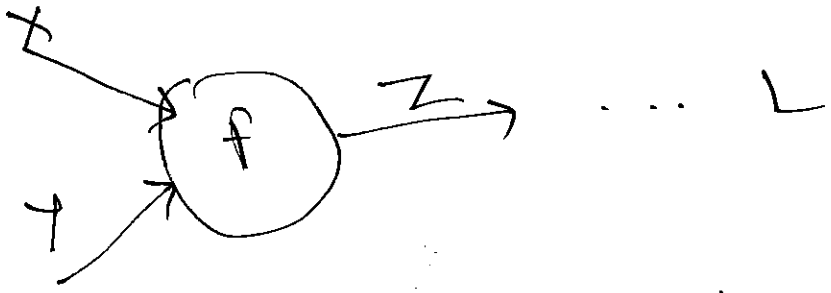
⇒ Rate of change at any pt in function x^2 is $2x$



BACKPROPAGATION

→ Backpropagation is the way of calculating gradients of Loss w.r.t. weights of network using chain rule

→ A neural network can be thought of as recursive computation graph



In forward Pass : compute local gradient i.e. $\frac{\partial Z}{\partial x} = \frac{\partial f(x,y)}{\partial x}$

In backward Pass: Get Gradient flowing backwards $\frac{\partial L}{\partial Z}$

We want to know: $\frac{\partial L}{\partial x}$ & $\frac{\partial L}{\partial y}$

Using Chain Rule: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial x}$ (Gradient flowing back (backward pass) → local gradient (forward pass))

In case $f(x,y) = x \cdot y$ then $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial Z} \cdot \frac{\partial (x \cdot y)}{\partial x} = \frac{\partial L}{\partial Z} \cdot y$ (gradient flowing back (backward pass) ← Input (activation previous layer))

→ -ve gradient e.g. $\frac{\partial L}{\partial w_1}$ = -ve means

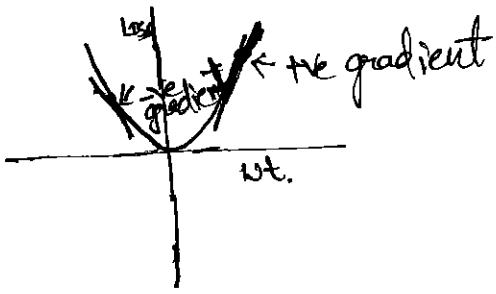
inc. in weight w_1 leads to decrease in loss

✓ ($w_1 = w_1 - \alpha \frac{\partial L}{\partial w_1}$ ⇒ decrease in loss

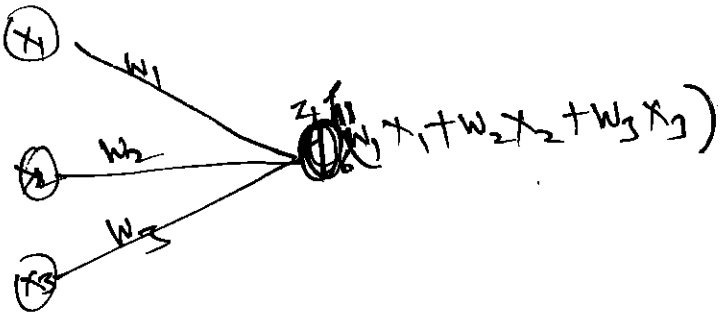
↓

-ve

$\Rightarrow w_1 = w_1 + (\text{something})$



→



$$\frac{dL}{dw_1} = \frac{dL}{dh_1} \cdot \frac{dh_1}{dz_1} \cdot \frac{dz_1}{dw_1}$$

$$= \frac{d}{dh_1} \frac{(y-h_1)^2}{2} \cdot \frac{d}{dz_1} \phi(z_1) \cdot \frac{d}{dw_1} (w_1x_1 + w_2x_2 + w_3x_3)$$

$$= \frac{1}{2} \times 2 (y-h_1) \cdot (-1) \cdot \phi(z_1)(1-\phi(z_1)) \cdot x_1$$

$$= -(y-h_1) \cdot \phi(z_1)(1-\phi(z_1)) \cdot x_1$$

= depends on input/activation from previous layer (x_1, x_2, x_3)
weights computed during forward pass (multilayer NNs) (w_1, w_2, w_3)
Choice of loss fn
Choice of activation fn ($h_1 = \phi(z_1)$)

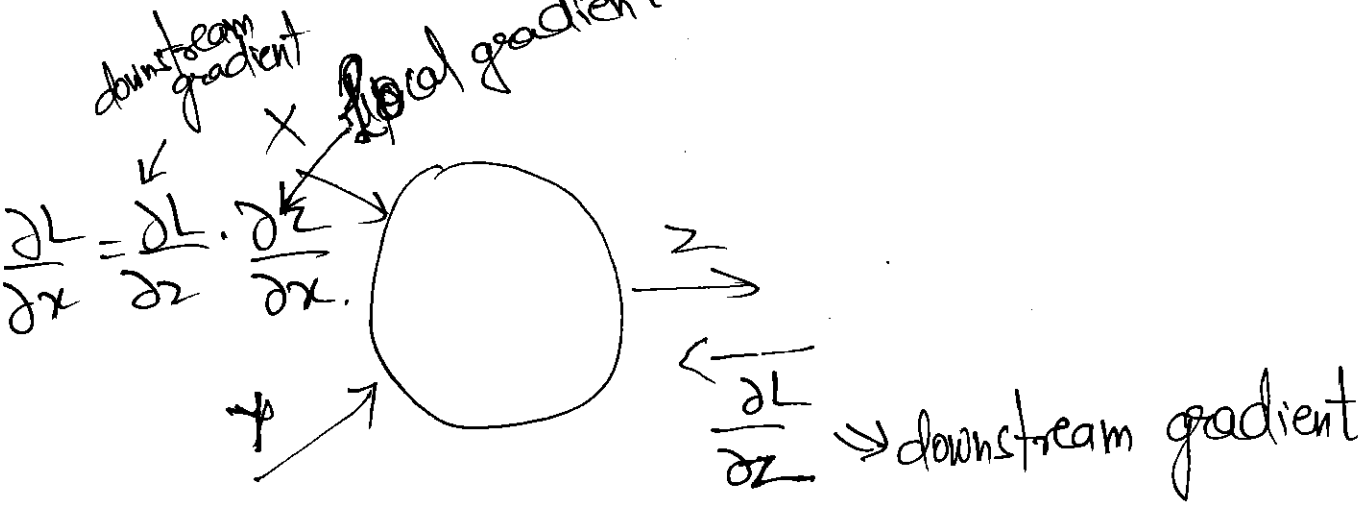
MINI-BATCH SGD

Loop:

1) SAMPLE a batch of data
2) FORWARD pass through the n/w, get Loss
3) BACKPROP to calculate the gradients
4) UPDATE parameters using the gradient

$$w_i = w_i - \alpha \frac{dL}{dw_i}$$

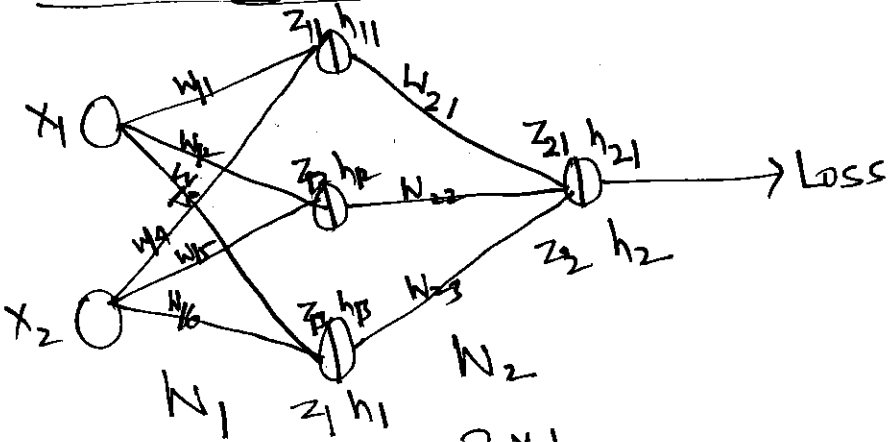
α ↓ learning rate



At each node, keep track of "local gradient" in forward pass

At the time of backprop, get "downstream gradient" & multiply them together and proceed back

NEURAL NETWORK FROM SCRATCH



$[X_1 \; X_2]$ dot $\begin{bmatrix} N_{11} & N_{12} & N_{13} \\ N_{21} & N_{22} & N_{23} \end{bmatrix} \Rightarrow [z_{11} \; z_{12} \; z_{13}]$ 6 $[h_{11} \; h_{12} \; h_{13}]$ dot $\begin{bmatrix} W_{21} \\ W_{22} \\ W_{23} \end{bmatrix}$

1X2 dot 2X3 $\Rightarrow$ 1X3 6 1X3 dot 3X1

$\Rightarrow [z_{21}]$

6

$[h_{21}]$

$$\text{Loss} = \frac{1}{2}(y - h_{21})^2$$

$$\frac{\partial L}{\partial h_{21}} = \frac{1}{2} \cdot 2 (y - h_{21}) \cdot (-1) = -(y - h_{21})$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial W_2}$$

$$\frac{\partial L}{\partial h_2} = -(y - h_2)$$

$$\frac{\partial h_2}{\partial z_2} = \frac{\partial (\sigma(z_2))}{\partial z_2}$$

$$= \frac{\partial}{\partial z_2}(\sigma(z_2))$$

$$= \sigma(z_2)(1 - \sigma(z_2))$$

$$= h_2 (1 - h_2)$$

$$\frac{\partial}{\partial W_2}(h_1 \cdot W_2) = h_1$$

Now $\frac{\partial L}{\partial W_2}$ needs to be same size of W_2 i.e. 3×1 matrix

$$\frac{\partial L}{\partial W_2} = \left(\frac{\partial z_2}{\partial W_2}\right)^T \text{dot} \left(\frac{\partial L}{\partial h_2} * \frac{\partial h_2}{\partial z_2}\right)$$

1. Bring last term to front
2. Transpose
3. dot with other operands

$$= (h_1)^T \text{dot} \left(-(y - h_2) * h_2(1 - h_2)\right)$$

3×1 ; 1×1 ; 1×1

$= 3 \times 1$ matrix

$$W_2 = W_2 - \alpha \frac{\partial L}{\partial W_2}$$ (All element-wise operations)

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

$$\frac{\partial z_1}{\partial W_1} = \frac{\partial}{\partial W_1}(X \cdot W_1) = X$$

$$1 \times 2$$

$$\frac{\partial h_1}{\partial z_1} = \frac{\partial}{\partial z_1}(\sigma(z_1))$$

$$= \sigma(z_1)(1 - \sigma(z_1))$$

$$= h_1(1 - h_1)$$

$\frac{\partial L}{\partial h_1}$ can be written as $= \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial h_1}$

$$\frac{\partial}{\partial h_1}(h_1 W_2) = W_2$$

Also we know $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$

$$-(Y - h_2) \quad h_2(1 - h_2)$$

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_2} * \frac{\partial h_2}{\partial z_2} \, dot \left(\frac{\partial z_2}{\partial h_1}\right)^T$$

$$1 \times 1 \quad 1 \times 1 \quad 1 \times 3$$

$$1 \times 1 \quad 1 \times 3 = 1 \times 3$$

1. Dot
2. Last term transpose

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

1. Bring last term to front
2. Transpose
3. dot with other operands

$\frac{\partial L}{\partial W_1}$ needs to be of same size as W_1 i.e: 2X3

$$= \left(\frac{\partial z_1}{\partial W_1}\right)^T \text{dot} \left(\frac{\partial L}{\partial h_1} * \frac{\partial h_1}{\partial z_1}\right)$$

$$= 2\times1 \qquad 1\times3 \quad 1\times3$$

$$= 1\times3$$

$$= 2\times3 \text{ matrix}$$

LEARNING RATE

→ GD ⇒ $W_{new} = W_{old} - \alpha \frac{\partial L}{\partial \theta}$ (mini-batch)

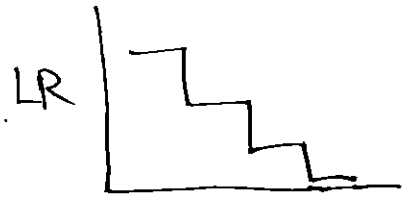
α ↓ Learning Rate

→ In order for (mini-batch) GD to work effectively, learning rate needs to be tuned

→ Learning Rate Schedule:

1) Constant LR
   i) High: large fluctuations, will end up dancing around optimum
   ii) Low: long time to converge

2) Decayed LR
   i) Step decay:



Drop learning rate by a factor every few epochs e.g. drop LR by half every 2 epochs

   ii) Exponential decay

Drop learning rate by a factor of e or 10 every S epochs. $lr_{new} = lr_{old} \cdot e^{(-k \cdot t)}$ (t: epoch, k: hyperparam)

Faster decay than step decay

iii) Performance Scheduling:

Measure validation error every N epochs and reduce LR by a factor of λ when error stops dropping.

E.g. In Keras, ReduceLROnPlateau (factor, epochs) fn can be used

→ With the advent of optimizers with adaptive learning rate such as Adam, RMSProp, Adagrad → learning rate need not be tuned specifically (initializing learning rate with default values generally works)

However, Learning Rate needs to be tuned with vanilla SGD, momentum & NAG

OPTIMIZERS: OPTIMIZING GD ALGOS:

→ Primarily aimed at resolving the drawbacks of learning rate
  → Learning rates that are constant or decayed on a schedule fail to adapt to the model, dataset or contours of loss fn
  → Learning rates applied uniformly to all wts don't account for sparsity

MOMENTUM: – Accumulates the gradient of past steps, adds a fraction to current time step gradient in order to update current wts

– $V_t = \gamma V_{t-1} + \alpha \frac{\partial L}{\partial \theta}$

↳ coeff of momentum ≈ 0.9

Coeff. of momentum γ can be interpreted as friction (inverse)
0.9 : high friction
1 : no friction

$$\theta_{new} = \theta_{old} - V_t$$

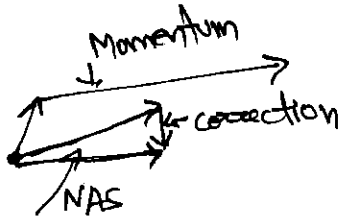
– Can be viewed as adapting to the slope of loss fn
– In effect what happens is gradients in zig-zag dimension cancel out and are accumulated slower over time. whereas gradient in preferred dir is accumulated gradually hence fluctuations reduce
– Encourages faster convergence but can also wander in wrong direction

NESTEROV MOMENTUM OR NESTEROV ACCELERATED GRADIENT (NAS)

- NAS updates the wts using previously accumulated gradients then measures the gradient (with the updated wts) then makes a correction (using the measured gradient)

  whereas
  Momentum adds a fraction of previously accumulated gradients to current time step gradient and then updates the wt.

- Leads to faster convergence & less oscillations than plain momentum



ADAGRAD: - Adapts wt. updates according to feature sparsity
- Features that activate freq perform smaller wt updates than features that are infrequent
- This is achieved by accumulating squared gradients per feature in the denominator of learning rate so learning rate decreases more often features activate
- Cons: The learning rate may decay so much that the algo stops before convergence

$$\rightarrow W_{new} = W_{old} - \frac{\alpha \frac{\partial L}{\partial \theta}}{\sqrt{(\text{accumulated grad.})^2 + \epsilon}}$$

- Maintains per parameter learning rate

RMSProp / Adadelta:

- Addresses the decay in learning rate to the extent that algo stops before convergence by allowing learning rates to speed up again
- Instead of keeping entire history of gradients from past time steps (accumulated gradient), keeps an "exponentially decaying moving average" of squared gradients (per feature)
- More memory efficient than Adagrad
- Maintains per parameter learning rate

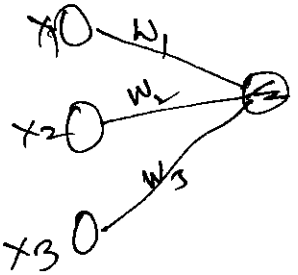
Adam (Adaptive Moment):

- RMS Prop + Momentum
- In addition to accumulating an exponentially decaying average of squared gradients in the denominator, Adam also stores the accumulated exponentially decaying average of gradients (not squared, otherwise Numerator & denominator will cancel) in the numerator (that acts as momentum term)
- Maintains per parameter learning rate

ACTIVATION FNS

Desirable properties of activation fns:

1. Differentiable (else no gradient)
2. Non-linear (else NN will be linear)
3. Derivative ≠ constant (else no dependence on input)
4. Should not saturate at some values of x (else vanishing gradient)
5. Zero-centered (else weight updates will be in one direction either all positive or all negative)



Let $f = \Sigma w_i x_i + b$

$$\frac{df}{dw_i} = x_i$$

$$\frac{dL}{dw_i} = \frac{dL}{df} \cdot \frac{df}{dw_i} = \frac{dL}{df} \cdot x_i$$

x_i is the I/P, now if the I/P is all +ve (output of activation of previous layer in case of multiple layers) $\frac{dL}{dw_i}$ will have the same sign as $\frac{dL}{df}$ (all +ve or all -ve) and weight update which is $w_{new} = w_{old} - \alpha \frac{dL}{dw_{old}}$ will be in one direction

Also, Zero-centered activations means that mean activation value is around zero and it has been shown empirically that models operating on normalized data - whether it be inputs or intermediate activations - enjoy faster convergence. Also explains the fact that NN with batch normalization layers converge faster.

Although ReLu is not zero-centered but since gradient can be calculated easily & no saturation happens at $x > 0$ → converges faster

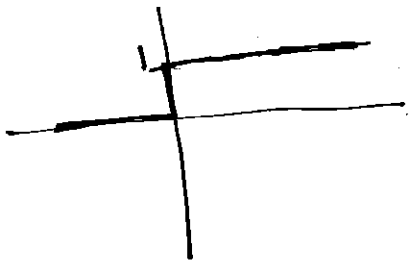
Combined with batch norm + dropout = works for deeper neural nets.

POPULAR ACTIVATION FNS:

1. Binary step fn

$f(x) = 1 \quad x >= 0$

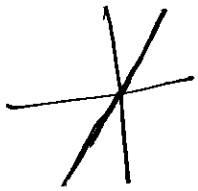
$= 0 \quad x < 0$



However $f'(x) = 0$ for all x → backprop not possible

2. Linear fn

$f(x) = ax$



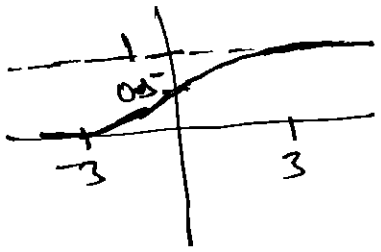
However, $f'(x) = a$ for all x → gradient does not depend on input value and is same always ⇒ gradient update step will not improve error (as gradient update will not depend on I/P)

Also if activation fn is linear, NN will not be able to model non-linear relationship between input and output.

3. SIGMOID [0,1]

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

$$f'(x) = f(x)(1-f(x))$$

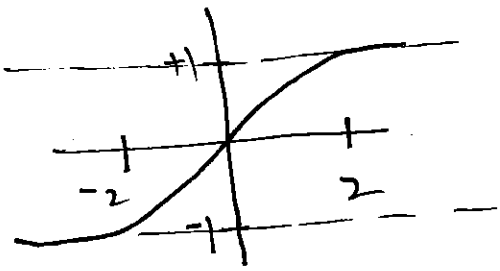


Adv: 1) Non-linear
2) Differentiable

Disadv: 1) $f'(x) \to 0$ when $x > 3$ or $x < -3$ (vanishing gradient)
2) Not zero-centered (All wt. updates will be in one dir)

4. TANH: Rescaled sigmoid ($\sigma(x)$) fn [-1,1]

$$f(x) = 2\,\sigma(2x) - 1$$



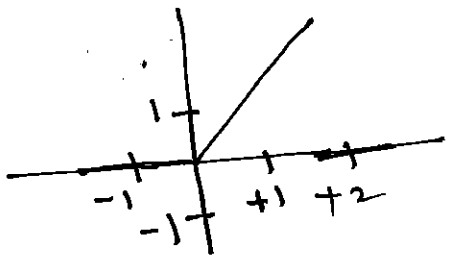
Adv: 1) Non-linear
2) Differentiable
3) Zero-centered

Disadv: 1) $f'(x) \to 0$ when $x > 2$ or $x < -2$ (vanishing gradient)

5. RELU

$$f(x) = \max(0, x)$$

$$f'(x) = \begin{cases} 0 & x < 0 \\ \text{undefined} & x = 0 \\ 1 & x > 0 \end{cases}$$, but in practice it is taken as 0



Adv: 1) Non-linear
2) Differentiable
3) Does not saturate at higher values of x
i.e. $f'(x) \neq 0$ when x is ~~large~~ positive
4) (Computationally) Fast to compute

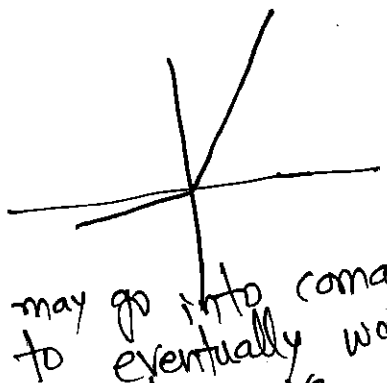
Disadv: 1) Dying ReLU

for $x < 0$, $f'(x) = 0$ meaning if the i/p is -ve then no activation. This also means that some neurons which go dead once are never activated.

6. LEAKY RELU :

$$f(x) = \max(ax, x)$$

↓ some constant usually 0.01



Adv: No dying ReLU, neurons may go into coma but have a chance to eventually wake up.
Outputs can be -ve

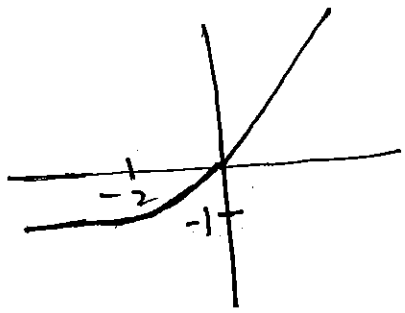
7. Parameterized ReLU

$$f(x) = \max(ax, x)$$

↑ learnable parameter

8. ELU

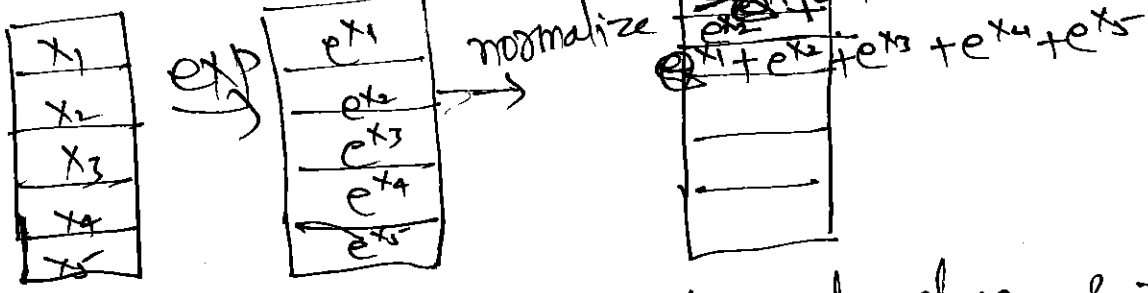
$$f(x) = \max(\alpha(e^x - 1), x)$$



For all the RELU variants, at high values of x, y can blow up meaning activation values can be very high (exploding gradient)

9. SOFTMAX:

I/P : Unnormalized log probabilities of each class



O/P : normalized prob. of each class & they sum to 1.

Maps input values to a prob. dist. over multiple classes

INTUITION AROUND Loss FN, OPTIMIZATION & REGULARIZATION

General way of training (neural networks/any other model)

1) Define a loss fn that quantifies our unhappiness with the scores across the training data
2) Come up with a way of efficiently finding the parameters that minimize the loss fn (optimization)

In One way:

Loss fn optimization ≈ searching for appropriate W's in the search space of W's defined by loss fn

When Regularization term is added to loss fn ≈ some kind of restriction is added to the search space of W's (penalizing complexity)

* It is important to note that regularization term only depends on W's (e.g. L1 norm or L2 norm) and not on input data.

WEIGHT INITIALIZATION IN NEURAL NETWORKS

WHY NOT RANDOM INITIALIZATION?

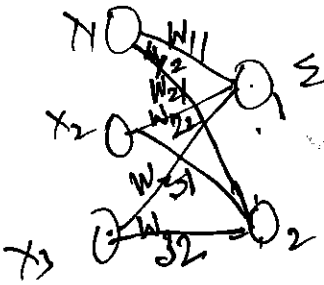
1) VANISHING GRADIENT PROBLEM:

for sigmoid & tanh act; random initialization can push some or all of the nodes into saturated region where gradient is near zero or zero $\Rightarrow$ ~~Boundary~~ propagation will stall

2) EXPLODING GRADIENT PROBLEM:

for relu activation, random initialization can push wt. of some nodes very high $\Sigma w_i x_i$ becomes large & since relu is linear for the inputs, the output goes big. Now at the time of BP, gradient flowing back is multiplied by this big value $\rightarrow$ gradients become large $\rightarrow$ changes in w will be in huge steps $\Rightarrow$ training will not converge

WHY NOT ZERO OR CONSTANT INITIALIZATION?



$\Sigma w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3$

If all $w_i = 0$

$\Sigma w_i x_i = 0$

$\Downarrow$

next layer O/P = same for all neurons

If all $w_i = C$

$\Sigma w_i x_i$ = same for O_1 & O_2

next layer O/P = same for all neurons

At the time of BP, when gradient will flow back.
⇒ gradient update will be same for all neurons
⇓ symmetry
Diff. neurons will not be learning different things/features

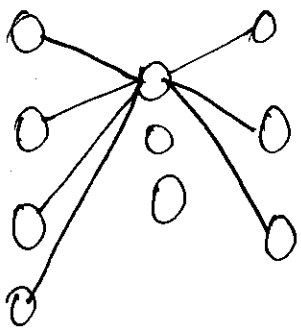
GLOROT/XAVIER AND HE INITIALIZATION:

— For the signal to flow properly, the variance of O/Ps of a layer = variance of its I/P
— Gradients too should have equal variance before and after flowing through a layer in reverse direction

GLOROT/XAVIER INIT: ① (LOGISTIC, TANH, SIGMOID, SOFTMAX ACT. FN)

② Wt. of each layer must be initialized by taking samples from following dist ⇒
Normal dist with mean 0 and var $\sigma^2 = \frac{1}{fan_{avg}} = \frac{2}{fan_{in} + fan_{out}}$

Where:
$$fan_{avg} = \frac{fan_{in} + fan_{out}}{2}$$



$fan_{in} = 4$

$fan_{out} = 3$

HE INIT : ① (Relu & its variants)

② Normal dist. with mean 0 and var $\sigma^2 = \frac{2}{fan_{in}}$

ADVANTAGE : 1) Speeds up training considerably (one of the tricks that led to current success of deep learning)

MINI-BATCH SIZE

→ The size of mini-batch should be such that it fits in GPU memory

→ Mini-batch size can be understood as no. of samples needed to make one wt. update

If epoch is constant : Bigger batch size → less no. of updates
Smaller batch size → more no. of updates

→ Mini-batch size : Same as difference b/w Batch GD & SGD

→ The jury is still out on : size of mini-batch & learning rate & mini-batch size interacting and optimum value

→ Higher mini-batch size →

1) Allows for parallelization (every mini-batch can be processed parallelly) & reduced training time
2) Reduces variance in gradient calculation .ie. less noisy gradient calculations Now since gradients are better, it
3) enables to take bigger step sizes.

Lower mini-batch size → 1) better generalization is known to have

→ Practical Advice : Somewhere in middle

ADDRESSING OVERFITTING IN DNNs

1. DROPOUT
2. L1/L2 Reg. (or Max-Norm Regularization)
3. Early stopping (when val. loss stops decreasing)
4. Data Augmentation
5. Reducing N/W Capacity (# of layers ↓, # of hidden units ↓)

MAX NORM REGULARIZATION:

→ For each neuron, it constraints the weights w of incoming connections such that $\|w\|_2 \leq r$ (r ← max-norm hyperparam, $\|w\|_2$ ← L2 norm)

→ Max-norm reg. does not add reg. loss term to overall loss. Instead, it is implemented by computing $\|w\|_2$ after each training step & rescaling w if needed ($w \leftarrow w \frac{r}{\|w\|_2}$).

→ Reducing r increases amt. of regularization & helps reduce overfitting

SOME ISSUES IN TRAINING DEEP NEURAL NETWORKS

* If it takes too much time to train
  → may be inc. size of mini-batch to reduce the variance in observations and help in convergence

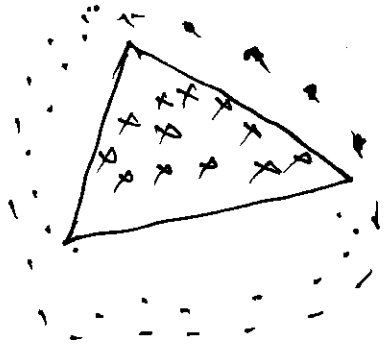
* If NaN predictions:
  First check input data should not have NaNs or missing values
  else it's an exploding gradient issue
  Remember SGD involves a bunch of matrix multiplication

1) Decrease learning rate so gradient updates will be smaller
2) Decrease # of layers (which will decrease # of multiplications)
3) Gradient Clipping

NOTE ABOUT NO. OF HIDDEN NEURONS & LAYERS

of hidden neurons = the extent of non-linearity in decision boundary

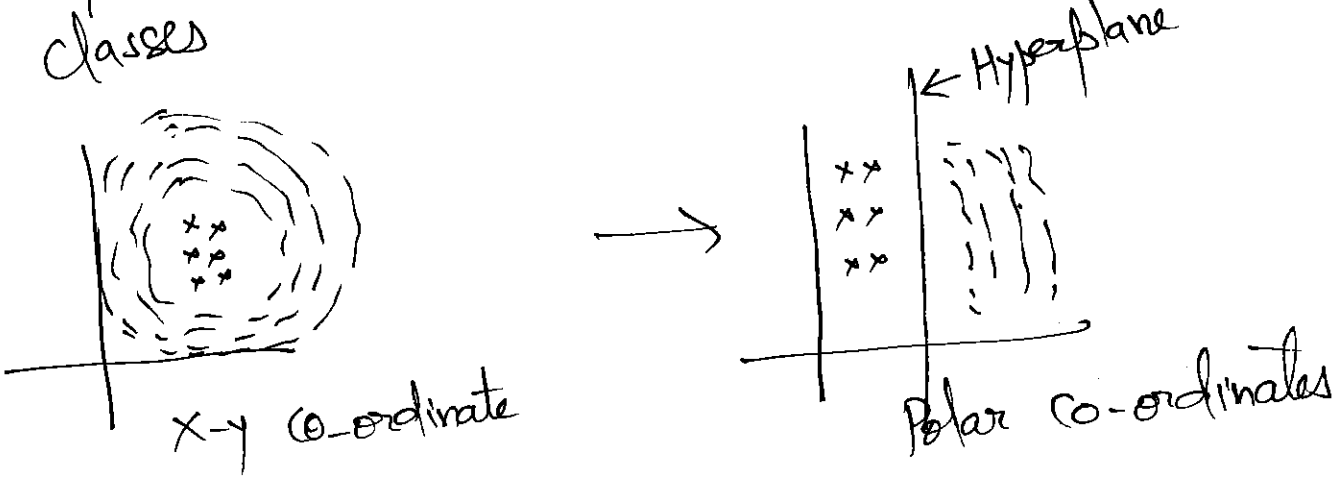
E.g. in 2D data:



Minimum # of hidden units required to separate the two classes perfectly = 3

= 3 piece-wise linear boundary

Layers can be understood as geometrical transformation of input (manifold transform) → so that later layers can cut a hyperplane to distinguish the classes



EXPLODING GRADIENT

What it is? If the gradients keep becoming larger and larger as training progresses
→ exploding gradient

(remember: DNN training involves a bunch of matrix multiplication)
e.g. relu is linear for +ve Inputs)

Why it happens? → Mostly common in Recurrent N/Ws
- Can be solved using LSTM (instead of stack of RNNs)

(Exploding gradient makes it harder for stack of simple RNNs to learn long-term temporal relations)

→ Can be caused by bigger wts initialization or larger learning rate

How to detect it? 1) Keep track of gradients of each layer
2) Model loss or model wts goes to NaN during training

How to Resolve it?

1. Gradient Clipping
2. Add weight Regularization (L2)
3. Reduce Learning Rate

$$w_t = w_t - \alpha \frac{\partial L}{\partial w_t}$$

if large → large updates to wts

and if wts become large then in subsequent BP where gradient is multiplied with wt vectors → swell up

4. Batch Normalization
5. Reduce N/W size
6. Proper wt. initializer
e.g. Glorot initialization with normal distribution

GRADIENT CLIPPING:

Clip by Value : Clip every component of gradient vector between some interval
e.g. $[-1, 1]$

: Can change the orientation of gradient vector

Original grad. vector $[0.9 \quad 100]$
↓clip
$[0.9 \quad 1.0]$
⇊ points in dir of second axis
⇊ points roughly in the diagonal of two axes

∴ In practice above works

Clip by Norm : Clip the whole gradient vector if its L2 norm is greater than the threshold chosen

e.g. if clipnorm = 1 then $[0.9 \quad 100]$
⇓ clip

$$1 \times \left[\frac{0.9}{\sqrt{(0.9)^2 + (100)^2}} \quad \frac{100}{\sqrt{(0.9)^2 + (100)^2}} \right]$$

: Preserves the orientation of original gradient vector

VANISHING GRADIENT

What it is?

The back propagation algorithm works by going from O/P layer to I/P layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of cost fn with regards to each parameter in the n/w, it uses these gradients to update each parameter with a GD step.

If the gradients get smaller and smaller as the algorithm progresses down to the lower layers
→ vanishing gradient

Why it happens?

1. Sigmoid & tanh Activation fns
When inputs become large(+ve) or become small (-ve) to sigmoid or tanh activation fn, the function saturates at 0 or 1 (sigmoid), 1 and -1 (tanh)
→ derivative becomes extremely close to 0

2. When wts are initialized with very high or very ~~low~~ values (possible in random initialization) $\Sigma w_i x_i$ can ~~become~~ a value where the activation fn saturates → derivative becomes extremely close to zero

↑
generalized problem case
($\Sigma w_i x_i$ = value where act. fn saturates)

How to Resolve it?

1. Avoid sigmoid & tanh activation fn (instead use RELU-like activation fns)
2. Use Glorot ~~of~~ Xavier initialization (weight initializer with variance)
3. Use Batch Normalization
4. Reduce Network Size

How to detect it?

- keep a watch on gradients (of lower layers especially)

BATCH NORMALIZATION

Why does it work ?

1. The original paper postulates that BN works since it reduces Internal Covariate Shift (ICS)

ICS: Training a deep NN can be viewed as collection of separate optimization problem - each one corresponding to training a different layer. Now during training, each step involves updating each of the layers simultaneously. As a result, updates to earlier layers cause changes in input distributions of later layers. This implies that optimization problem solved by subsequent layers change at each step.

Constant changes in layer's input distribution force the corresponding optimization processes to continually adapt, thereby hampering convergence.

2. A recent paper postulates/demonstrates that BN makes the landscape of corresponding optimization problem/loss surface significantly more smooth and does not reduce ICS.

Loss fn in DNN are not only non-convex but also tend to have large number of 'kinks' i.e. flat regions, sharp minima etc. This makes Gradient descent based training algorithms unstable, e.g. exploding or vanishing gradients, thus highly sensitive to the choice of learning rate & initialization.

BN thus enables any gradient-based training algorithm to take larger steps without the danger of running into such change of loss landscape such as (exploding gradient) flat (vanishing gradient) region, or sharp local minimum. This in turn enables us to use a larger/bigger value of learning rates & speed up training. Also becomes less sensitive to the choice of hyperparameters. Further, since VG or EG problem is reduced significantly, deeper networks can be trained

How BN works?

Let $x_1, x_2, \ldots x_m$ be activations in a mini-batch of size m

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \text{// normalize (centering at 0 \& var} \approx 1)$$

$$y_i = \gamma \hat{x}_i + \beta \qquad \text{// scale + shift (learnt params)}$$

(γ: scale, β: shift)

→ Simply normalizing & not doing scaling + shifting may change what the layer can represent i.e. reduced expressiveness of a network

→ parameters γ and β are introduced which are learned along with model parameters and restores the representation power of the network

→ γ & β can take any values "best" for the n/w including

identity such that $y_i = \hat{x}_i$

BN at Test time: For individual predictions

We require 4 params : μ, σ, γ and β

γ → learnt during training time

β → learnt during training time

For μ and σ, at every layer for every mini batch exponential moving avg of μ & σ is maintained and those values are used at test/prediction time to compute

$$y_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Advantages of BN:

1) Reduces VG/EG problem
2) Higher learning rates can be used leading to faster convergence & less training time
3) Reduces strong dependence on weight initialization
4) Acts as a regularizer to a smaller extent

Disadvantages of BN:

1) Prediction time increases (due to extra computation at test time)

Where to apply BN:

Orginal paper → between $\Sigma W_i x_i$ & Activation.
However, in practice it works better when applied after activation of a layer.

BEST PRACTICE:

Normalize inputs before feeding to NN.
And after Hidden layers apply BN layer (but before dropout)
BN Not after O/P layer

Why BN acts as Regularizer?

Mini-batch can be considered as random samples from training data.

For every mini-batch, μ & σ will fluctuate (meaning being noisy) depending on training samples in that mini batch

$\Rightarrow \hat{x}_i$ (normalized x_i) $= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$

$\hat{x}_i$ fluctuates

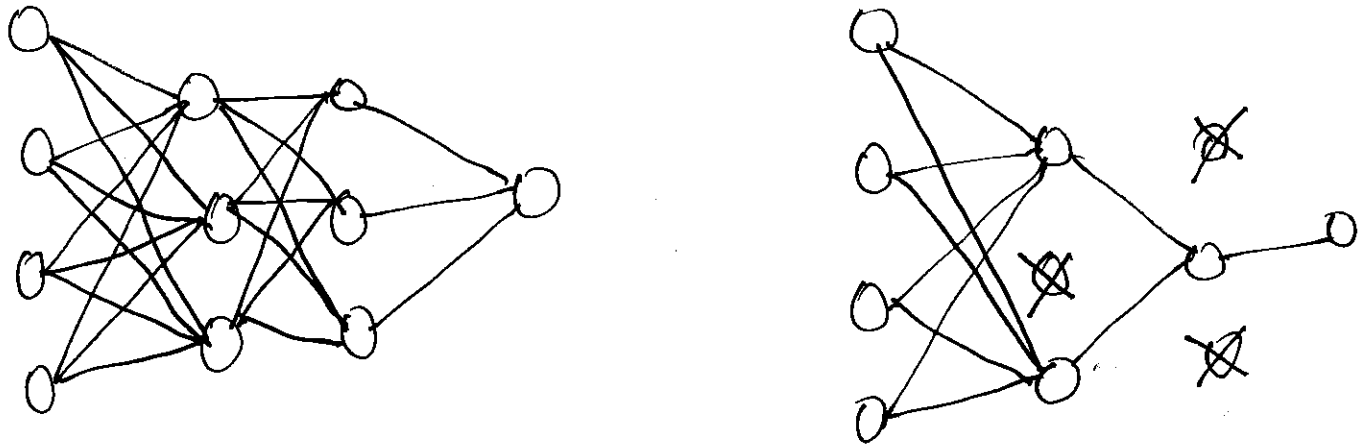
i.e. for a given x_i, $\hat{x}_i$ will fluctuate depending on other examples in that mini-batch,

This in turn means, layers have to learn to be robust to a lot of variation in its input, just like dropout.

Another way to look at it — When training with BN, a training example is seen in conjunction with other examples in the mini-batch and the training network no longer produces deterministic values for a given training example.

This also means that large mini-batch sizes have less noise in μ & σ (better estimate of full training set's μ & σ) $\Rightarrow$ lesser regularization

DROPOUT



Dropout Concept

→ Dropout means temporarily removing a unit
✓ from the network along with its incoming and outgoing connections. at training time with a prob p.
At test time, all units are present but with w scaled by p (ie w becomes pw)

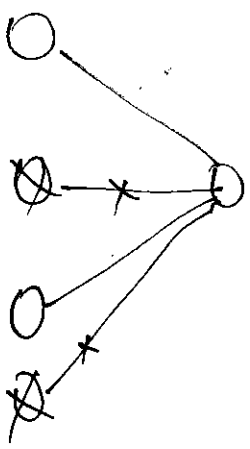
→ Let p be the probability that each unit will be retained at a given layer then $1-p$ will be prob that the unit will dropout

→ For each training example, at every hidden
✓ layer some units are cut off = "thinned" version of original n/w. Now forward & backprop is applied on this "thinned" network for a training example. For the next training example, some other units may be switched off = "another thinned" network and forward & backprop is applied.

→ Dropout can be interpreted as a way of regularizing a neural network by adding noise to its hidden units. = reduces over-fitting

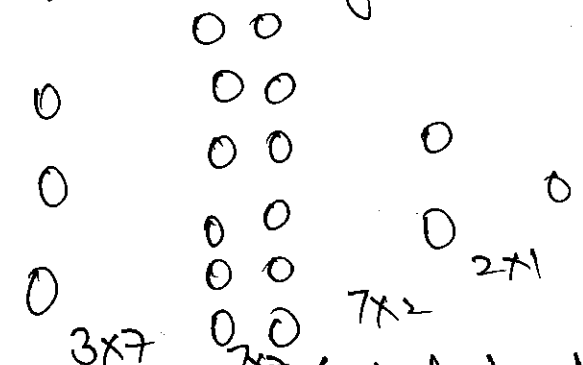
→ Dropout increases the number of iterations required to converge but training time for each epoch (one iteration) is reduced.

→ At input layer, if dropout needs to be applied it should be small else we will loose training data.

→ 

Can't rely on any one feature since any one feature/combi of features can be turned off
≈ L2 regularization
(coeff/weights are shrunk)

→ Generally dropout should be higher at a layer whose weight matrix is bigger



highest since dropout should be applied here chances of overfitting are highest here

→ In Applied case : Dropout makes cost fn not well-defined as units randomly keep dropping (n/w architecture changes). So in practical cases, first check that loss is decreasing smoothly then add dropout.

Why does Dropout Work?

Weakens/breaks CO-ADAPTATION: In a standard neural network, the derivative received by each parameter tells how it should change so that final loss fn is reduced, "given what all other units are doing". Therefore, units may change in a way that fixes the mistakes of other units. This may lead to complex co-adaptations. However, these co-adaptations do not generalise well to unseen data → leads to over-fitting. By randomly shutting off units, such co-adaptions are broken making a unit not rely on other units to correct its mistakes.

Dropout can be thought of as creating an implicit ensemble of neural networks

DROPOUT IMPLEMENTATION in FORWARD PASS:

```
Z1 = np.add (np.matmul(W1, X), b1)
A1 = np.tanh(Z1)
D1 = np.random.rand( A1.shape[0], A1.shape[1])
D1 = D1 < keep-prob
        # Create a matrix of same dimension as
          A1 & randomly initialize its elements
        # Now create a binary matrix
          where element = 1 if its value is
          less than keep-probability
A1 = np.multiply (A1, D1)   # shut down corresponding
                              elements of A1 where element
                              in D1 == 0
A1 = A1 / keep-prob         # scale the value of
                              non-zero elements so that
                              expected value of the
                              layer remains same as
                              without dropout
```

DROPOUT IMPLEMENTATION IN BACKPROP:

- Gradients for units that were dropped are zeroed out
- Other gradients are scaled by $\frac{1}{\text{keep\_prob}}$

AT TESTTIME: We do not need to dropout units at test time since the activations are scaled back up to expected value of the layer during ~~training~~ test time. (can be done at test time as well)

On another note, if dropout is applied at test time it usually is done to get some confidence range around predictions.

ENTROPY

→ measure of uncertainty of a system
or
→ amt. of info needed to remove uncertainty
or
→ expected value of surprise

probability $\propto \frac{1}{\text{Surprise}}$

Prob. of picking red dot is high ⇒ surprise of picking red dot is low

However, when prob = 1 then surprise = 1 (if prob = $\frac{1}{\text{surprise}}$)
But, if prob is 1 then surprise should be zero
hence taking log

⇒ Surprise = $\log\left(\frac{1}{\text{prob}}\right)$ $\left(= \log 1 = 0\right)$

E.g. Let's take a biased coin such that

	H	T
Prob	0.9	0.1
Surprise $\log_2\left(\frac{1}{p}\right)$	0.15	3.32

Total surprise after flipping the coin 100 times
= Number of times we will get heads (0.9 × 100) × Surprise from one head
+ Num. of times we will get Tails (0.1 × 100) × surprise from one tail
= $0.9 \times 100 \times 0.15 + 0.1 \times 100 \times 3.32$

Avg. Surprise per coin toss = Entropy

$= \frac{0.9 \times 100 \times 0.15 + 0.1 \times 100 \times 3.32}{100}$

$= 0.47$

→ $H(P) = \sum p_i \log \frac{1}{p_i} = -\sum p_i \log p_i$

$H(P)$: Entropy of discrete prob. dist. with N outcomes

p_i: prob. of i^{th} outcome

$\forall\, p_i < 1$ & log (less than 1) = -ve
$= - \times -ve = +ve$ or 0

→ The base of log is usually the no. of outcomes possible
e.g. for coin toss = 2

→ Range of entropy $[0, \log N]$

↓ when all outcomes have equal prob. of $\frac{1}{N}$

↓

if base of log = No. of outcomes

↓

$[0, 1]$

→ E.g. for a fair coin

Entropy $= -\left[\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right]$

$= -\left[\log_2\frac{1}{2}\right]$

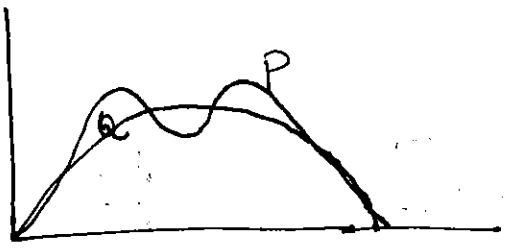
$= -\left[\log_2 2^{-1}\right]$

$= +1\left[\log_2 2\right]$

$= 1$ → Highly uncertain system

KL DIVERGENCE

→ $D_{KL}(P||q)$: Information loss when distribution P is represented by dist. Q
or
how "dissimilar" prob. dist. P is to candidate dist. q



P distribution (True coin) Q distribution (Another coin)

E-g. $\begin{cases} p_1 & \text{heads} \\ p_2 & \text{tails} \end{cases}$ $\begin{cases} q_1 & \text{heads} \\ q_2 & \text{tails} \end{cases}$

Flip coin n times

H H T H H T H H H T H T

Prob. of True coin generating this

$p_1\ p_1\ p_2\ p_1\ p_1\ p_2\ p_1\ p_1\ p_1\ p_2\ p_1\ p_2 = p_1^{N_H} p_2^{N_T}$ $\begin{cases} N_H = \text{No. of heads} \\ N_T = \text{No. of tails} \end{cases}$

Prob of Coin 2 generating this

$q_1\ q_1\ q_2\ q_1\ q_1\ q_2\ q_1\ q_1\ q_1\ q_2\ q_1\ q_2 = q_1^{N_H} q_2^{N_T}$

Ratio of prob. to get likelihood: $\dfrac{p_1^{N_H} p_2^{N_T}}{q_1^{N_H} q_2^{N_T}}$

Normalizing it $\left(\frac{p_1^{N_H} p_2^{N_T}}{q_1^{N_H} q_2^{N_T}}\right)^{1/N}$

Taking log $\frac{1}{N} \log \left(\frac{p_1^{N_H} p_2^{N_T}}{q_1^{N_H} q_2^{N_T}}\right)$

$$= \frac{1}{N} \log p_1^{N_H} + \frac{1}{N} \log p_2^{N_T} - \frac{1}{N} \log q_1^{N_H} - \frac{1}{N} \log q_2^{N_T}$$

$$= \frac{N_H}{N} \log p_1 + \frac{N_T}{N} \log p_2 - \frac{N_H}{N} \log q_1 - \frac{N_T}{N} \log q_2$$

If the coin is flipped infinite no. of times then

$\frac{N_H}{N} = p_1$ ~ True coin prob of getting heads

$\frac{N_T}{N} = p_2$ ~ True coin prob of getting tails

$$= p_1 \log p_1 + p_2 \log p_2 - p_1 \log q_1 - p_2 \log q_2$$

$$= p_1 \log p_1 - p_1 \log q_1 + p_2 \log p_2 - p_2 \log q_2$$

$$= p_1 \log \frac{p_1}{q_1} + p_2 \log \frac{p_2}{q_2}$$

∴ Normalized log likelihood

→ $D_{KL}(p||q) = \sum_{i=1}^{N} p_i \log \frac{p_i}{q_i}$ where p_i = prob of i^{th} outcome from dist. p

q_i = prob of i^{th} outcome from dist. Q

N = Number of outcomes

For continuous prob. dist.

$$= \int p(x) \log \frac{p(x)}{q(x)}$$

E.g. If $P = [1 \; 0]$ True Dist.

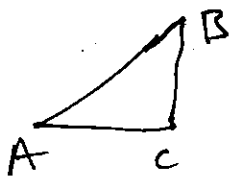
$Q = [0.7 \; 0.3]$ Pred. Prob. by Model

$$D_{KL}(p||q) = 1 \log \frac{1}{0.7} + 0 \log \frac{0}{0.3}$$

$$= \log \frac{1}{0.7}$$

→ $D_{KL}(p||q) \neq D_{KL}(q||p)$: asymmetric

→ Not a distance measure since it is asymmetric and asym. since it does not satisfy triangle inequality



$D_{KL}(A||B) \not\leq D_{KL}(A||C) + D_{KL}(C||B)$

i.e. any side of Δ must be shorter than the other 2 sides added together (Δ inequality theorem)

→ If the other distribution (Q) perfectly matches the true distribution (P)

$$= \log \frac{p_i}{q_i} = \log 1 = 0 \quad = D_{KL}(p||q)$$

✓ i.e. Lower the KL divergence the better we have matched the true distribution with our approximation

→ ∴ Range of KL divergence $[0, \infty]$

→ Relation to Cross Entropy:

✓ $$D_{KL}(p||q) = \sum_{i=1}^{N} p_i \log \frac{p_i}{q_i}$$

$$= \sum_{i=1}^{N} p_i (\log p_i - \log q_i)$$

$$= \sum_{i=1}^{N} (p_i \log p_i - p_i \log q_i)$$

$$= \sum_{i=1}^{N} p_i \log p_i - \sum_{i=1}^{N} p_i \log q_i$$

↓ ↓

= −Entropy of True Dist P + Cross Entropy b/w dist P&Q

∴ Cross-Entropy b/w dist P&Q − Entropy of true dist P

→ In ML systems, P ⇒ label/target dist. which does not depend on parameters of model, hence

✓ optimizing for KL div. is equivalent to optimizing for Cross Entropy

CROSS-ENTROPY / LOG LOSS / LOGISTIC LOSS / NEGATIVE LOG LIKELIHOOD

→ same interpretation as KL divergence + not symmetrical + not a dist. measure

→ $H(p,q) = \sum_{i=1}^{N} p_i \log \frac{1}{q_i} = -\sum_{i=1}^{N} p_i \log q_i$

Where p_i = prob. of outcome i (true dist.)
q_i = prob of outcome i (another dist.)
N = No. of outcomes

In ML systems, p_i = true class distribution

e.g. $P = [1 \quad 0]$ (Labels/target) $\quad Q = [0.7 \quad 0.3]$ for one instance

↑ True Prob. of Class 1 ↑ True prob. of class 0 ↑ Pred. prob of class 1 ↑ pred. prob of class 0

$$H(p,q) = -(1 \log 0.7 + 0 \log 0.3)$$
$$= -\log 0.7$$

✓ = −log predicted prob from softmax fn for class 1 (true class)

(pred. prob from softmax)

x_1	x_2	label	p	Cross-Entropy
0.04	0.42	Class 1	0.57	0.56
1	0.54	Class 2	0.58	0.54
0.5	0.37	Class 3	0.52	0.65
			Total = 1.75	
			Avg = 1.75/3 → used to calc. avg. loss & back-propagation	

Given — pred. from model — Computed

$CE_{Class1} = -\log(0.57) = 0.56$

$CE_{Class2} = -\log(0.58) = 0.54$

$CE_{Class3} = -\log(0.52) = 0.65$

→ Reason for -ve sign in Cross-entropy formula

$= -\sum_{i=1}^{N} p_i \log q_i$, q_i is less than 1 & log <1 = -ve so overall +ve

$= \sum_{i=1}^{N} p_i \log \frac{1}{q_i}$. $\log \frac{1}{q_i}$ term is "surprise" (refer entropy notes)

Another way of interpreting cross-entropy is how "surprised" we are, on average, when we learn the true value of y.
Low surprise → if the o/p is what we predict
High surprise → if the o/p is unexpected

→ Why don't we use residual(SSR) of ~~pred.~~ prob. as loss fn instead of cross-entropy?

Residual = True Prob − Pred. Prob (from softmax)

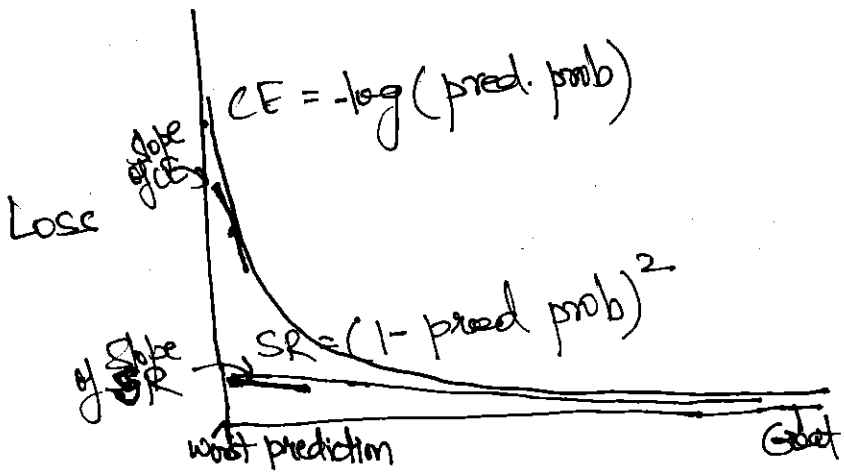
$\text{Residual}^2 = (\text{True Prob} - \text{Pred prob (from softmax)})^2$

$SSR = \sum_{i=1}^{N} (\text{True Prob} - \text{Pred prob (from softmax)})^2$

(True Prob = 1)

↓ Sum of squared residuals

Reason: i) For worst predictions, the loss kind of explodes in CE
ii) The derivative, or slope of tangent line, for CE, for a bad prediction will be relatively large



− CE
− ~~SSR~~

So when neural n/w makes a really bad prediction, CE will help take a relatively large step towards a better pred.

→ Relationship b/w CE, KL divergence & Entropy

$$H(p,q) = D_{KL}(p\|q) + H(p)$$

where p = true dist
q = pred. dist

- $H(p,q)$: Cross-Entropy
- $D_{KL}(p\|q)$: K-L divergence (info loss when P is rep. by Q)
- $H(p)$: Entropy (Uncertainty in dist. P)

In ML systems, P → label/target and is not dependent on parameters of model hence optimizing for KL divergence is equivalent to optimizing for cross entropy

→ Why CE is used and not KL divergence in loss fn?

i) CE has simpler form than KL divergence i.e. no need to calc. $\log \frac{p_i}{q_i}$ and implicitly entropy of true dist. (which is not dependent on params of model)

ii) Softmax prob. outputs can be straightaway used to calc. CE i.e. $-\log$ (pred. prob. from softmax)

→ Range of CE = $[0, \infty]$

DERIVATIVE OF BINARY CROSS-ENTROPY:

PRODUCT RULE: For two differentiable fns $u(x)$ and $v(x)$, the derivative of uv $= \frac{d}{dx}uv = u\frac{d}{dx}(v) + v\frac{d}{dx}(u)$

BINARY CROSS ENTROPY $= -\sum_{i=1}^{2} p_i \log q_i$ (p_i: true prob, q_i: pred. prob)

since $p_1+p_2=1$, $q_1+q_2=1$

$$= -p_1 \log q_1 - (1-p_1)\log(1-q_1)$$

$$= -[p_1 \log q_1 + (1-p_1)\log(1-q_1)]$$

Let $p_1 = t$ (true prob), $q_1 = \hat{y}$ (pred. prob) } for sake of simplicity/clarity

$\therefore$ Binary CE $= -[t\log\hat{y} + (1-t)\log(1-\hat{y})]$ (i)

$$\frac{\partial}{\partial\hat{y}}(t\log\hat{y}) = t\frac{\partial}{\partial\hat{y}}(\log\hat{y}) + \log\hat{y}\frac{\partial}{\partial\hat{y}}(t) \qquad \text{Since } \frac{d}{dx}(\log x) = \frac{1}{x}$$

$$= t\cdot\frac{1}{\hat{y}} + \log\hat{y}\cdot 0$$

$$= \frac{t}{\hat{y}} \quad \text{(ii)}$$

$$\frac{\partial}{\partial\hat{y}}(1-t)\log(1-\hat{y}) = (1-t)\frac{\partial}{\partial\hat{y}}(\log(1-\hat{y})) + \log(1-\hat{y})\frac{\partial}{\partial\hat{y}}(1-t)$$

$$= -\frac{1-t}{1-\hat{y}} + \log(1-\hat{y})\cdot 0$$

$$= -\frac{1-t}{1-\hat{y}} \quad \text{(iii)}$$

Subs. (ii) & (iii) in (i)

$$\frac{\partial CE_{\text{binary}}}{\partial\hat{y}} = -\left[\frac{t}{\hat{y}} - \frac{1-t}{1-\hat{y}}\right] = \frac{1-t}{1-\hat{y}} - \frac{t}{\hat{y}}$$

Minimizing Cross Entropy = Minimizing KL = Maximizing likelihood

i) Minimizing CE = Minimizing KL

$$H(p,q) = D_{KL}(p\|q) + H(p)$$

↓

uncertainty in true/label dist.
& since dataset with labels is given (constant)

ii) Minimizing KL = Maximizing (log) likelihood

$$D_{KL}(p\|q) = \sum p_i \log \frac{p_i}{q_i}$$

$$= \sum \underbrace{p_i \log p_i}_{\text{True dist = constant}} - p_i \log q_i$$

Minimize $= \sum -p_i \log q_i$

Maximize $= \sum +p_i \log q_i$

↓

Law of Large Numbers i.e. : from a large number of repeated trials, the expected value of random var = avg value

↓ constant

$$= \sum \log q_i \quad (i)$$

Likelihood Ratio: prob. of observing data under two diff. hypothesis.

$$LR = P(\text{Data}|H_1) \,/\, P(\text{Data}|H_2)$$

↓ provides quantitative measure of how much more likely the data is under one hypothesis compared to the other

$LR > 1 \Rightarrow$ data is more likely under H_1,

$LR < 1 \Rightarrow$ data is more likely under H_2

$$LR = \frac{q_i}{p_i}$$ → Maximizing this likelihood means maximizing the likelihood that the data came from q_i (i.e. the model)

For a set of data with independent samples, we can compute the likelihood ratio for entire set by taking the product of likelihood ratio for each sample

Note: if A & B are independent
$P(A \text{ and } B) = P(A) \cdot P(B)$

$$LR = \prod \frac{q_i}{p_i}.$$

Taking log ($\prod \to \Sigma$)

$$\log LR = \Sigma \log \frac{q_i}{p_i}$$

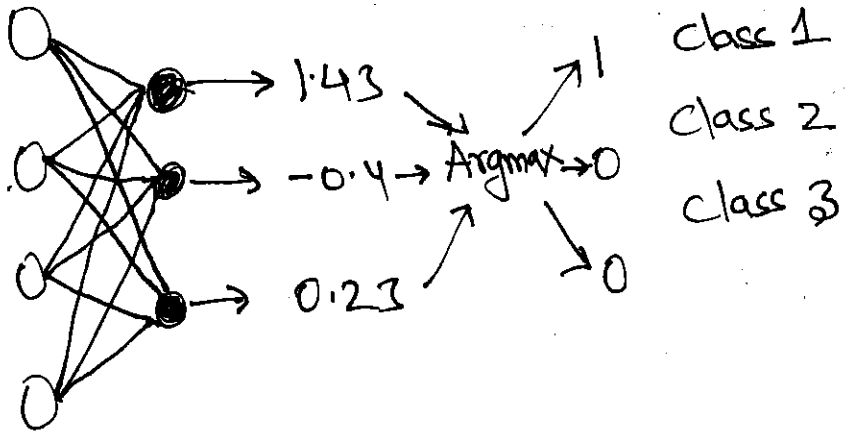
$$= \Sigma \log q_i - \underbrace{\log p_i}_{\text{const}}$$

$$= \Sigma \log q_i \qquad \text{(ii)}$$

Since (i) = (ii)

ARGMAX

→ Neural N/W o/p scalar values in their basic form ($\sum x_i w_i$) [in absence of any activation fn]



→ One way to determine the class of this instance is to take "argmax" of these values and output the class.

→ However, argmax is not differentiable so ~~backpropagation~~ not possible

argmax (x_1, x_2) takes a pair of numbers (in this case) and o/ps 0 if $x_1 > x_2$

1 if $x_2 > x_1$

(value at $x_1 = x_2$ is arbitrary / undefined).

So wherever you are on (x_1, x_2) plane, as long as you are not on the $x_1 = x_2$, if you move infinitesimal tiny bit in any dir, you won't change the value (0 or 1) that argmax outputs i.e. gradient of argmax (x_1, x_2) w.r.t. x_1 and x_2 is (0,0) almost everywhere except when $x_1 = x_2$. At $x_1 = x_2$ (and argmax value changes abruptly from 0 to 1 or vice-versa), its gradient w.r.t. x_1, x_2 is undefined.

→ Another way to say argmax is not differentiable
✓ is due to the fact that it outputs a constant (0 or 1)
and constants gradients are 0 w.r.t. any variable

→ However, argmax can be used at the time of
✓ prediction.

→ At the time of training, instead of argmax,
✓ Softmax can be used.

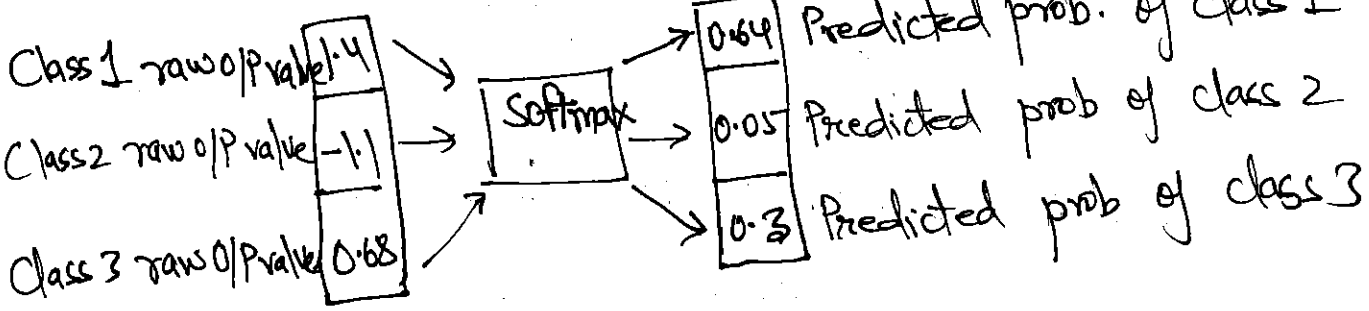
SOFTMAX

→ Softmax fn takes raw output values from basic neural networks (ΣWX) and converts them into predicted probabilities of each class with the following properties

i) prob. of each class $[0,1]$

ii) sum of prob. of all classes $= 1$

iii) relative ordering of scalar/raw output values is maintained



$$\text{Softmax}(S_i) = \frac{e^{S_i}}{\sum_{n=1}^{N} e^{S_n}}$$

where N = no. of classes

S_i = raw o/p value from class i

e.g. for above

$$P(1) = \text{Softmax}(\text{raw o/p value from class 1}) = \frac{e^{1.4}}{e^{1.4} + e^{-1.1} + e^{0.68}} = 0.64$$

$P(1)$ → pred. prob. of class 1

→ Softmax is generalization of sigmoid over multiple classes

→ In contrast to argmax, softmax has valid gradients
✓ that can be used in back propagation

→ derivative of "predicted" prob of class 1 w.r.t. raw output value for class 1

$$\frac{d\, p_{class1}}{d\, raw_{class1}} = p_{class1} \times (1 - p_{class1})$$

↑ predicted prob. of class 1

→ derivative of "predicted" prob. of class 1 w.r.t. raw output value for class 2

$$\frac{d\, p_{class1}}{d\, raw_{class2}} = -\, p_{class1} * p_{class2}$$

↑ predicted prob of class 1

↑ predicted prob. of class 2

INTUITION ABOUT ABOVE DERIVATIVE

Let marks of a student

100	→ Maths
5	→ Biology
4	→ Arts

} raw scalar values

Probability of choosing major?

Prob. of choosing maths as a major is very high compared to other subjects

i.e. $\frac{e^{100}}{e^{100} + e^5 + e^4} \approx 1$

Now $\frac{d\, p_{Math}}{d\, raw_{Math}} = p_{math} \times (1 - p_{math})$ is telling us that

$= 1 \times (1-1) \approx 0$

i.e small changes in math marks will not change the prob. much

Similarly, in other extreme case, if math marks were very low compared to other subjects i.e. $p_{math} \approx 0$ then again small changes in math marks will not change the prob. much as $0 \times (1-0) = 0$

Now $\frac{d\,p_{math}}{d\,p_{biology}}$: Prob. of choosing math if there are changes to marks in other subject

Intuitively, if marks in biology increase then the prob. of choosing math decreases (since sum of prob = 1)
hence the -ve sign in below formula

$$= - p_{math} \times p_{biology}$$

Different Blocks Present in A Typical NER Model

A typically named entity recognition NLP model consists of several components, including:

1. Tokenization: Tokenization breaks text into individual tokens (usually words or punctuation marks).
2. Part-of-speech tagging: Labelling each token with its corresponding part of speech (e.g., noun, verb, adjective, etc.).
3. Chunking: Group tokens into "chunks" based on their part-of-speech tags.
4. Name entity recognition: Identifying named entities and classifying them into predefined categories.
5. Entity disambiguation: The process of determining the correct meaning of a named entity, especially when multiple entities with the same name are present in the text.

Deep Understanding of Named Entity Recognition with An Example

To get a better understanding of how named entity recognition NLP works, let's walk through an example using the following sentence:

"Mark Zuckerberg founded Facebook in 2004 in Menlo Park, California."

Tokenization: The first step in our NER process is to tokenize the text, which means breaking it into individual tokens. In this case, our tokens would be: ['Mark', 'Zuckerberg', 'founded', 'Facebook', 'in', '2004', 'in', 'Menlo', 'Park', ',', 'California', '.']

Part-of-speech tagging: We would label each token with its corresponding part of speech. This step might produce the following tags: ['NNP', 'NNP', 'VBD', 'NNP', 'IN', 'CD', 'IN', 'NNP', 'NNP', ',', 'NNP', '.']

Chunking: Using the part-of-speech tags, we can now group the tokens into "chunks" based on their tags. In this case, we might have the following chunks: [('Mark', 'NNP'), ('Zuckerberg', 'NNP'), ('founded', 'VBD'), ('Facebook', 'NNP'), ('in', 'IN'), ('2004', 'CD'), ('in', 'IN'), ('Menlo', 'NNP'), ('Park', 'NNP'), (',', ','), ('California', 'NNP'), ('.', '.')]

Named entity recognition: Using the information from the chunking step, we can now identify and classify the named entities in the text. In this case, we have two named entities: "Mark Zuckerberg" and "Facebook", both of which are people, and "Menlo Park, California", which is a location.

Entity disambiguation: In this step, we would determine each named entity's meaning. For example, if multiple people have the name "Mark

Zuckerberg", we must determine which one is being referred to in the text.

How does Named Entity Recognition Work?

Several approaches can be used to perform named entity recognition NLP models. The most common methods include the following:

Rule-based methods use a set of predefined rules and patterns to identify named entities in text.

Statistical methods use a probabilistic framework to identify named entities in a text by training a model on a large annotated text corpus.

Machine learning methods also use probabilistic frameworks but rely on Machine Learning algorithms to learn the patterns in the data.
Once the model is trained, we can identify named entities in a new text by applying the learned patterns and features.
Machine learning-based methods tend to be more accurate and scalable than rule-based methods, but they require more labeled training data.