- How to optimize dep learning algorithms for hardware : focus
- Presentation – Motivation
  Problem Statement
  Method
  Results
  } for research paper
- Coding – part : present some part of the paper (1 or 2 experiments)
- Autocrat (grad) – What is it? : something differentiation etc

29/07/25

Topic : ML Basics

a) Focus on : (i) Gradient Descent
(ii) Backprop (BP)

b) Covering high level motivation in ML

C1 • MNIST – handwritten digits (eg. 0 written in various ways) ← dataset

⇐ Q1 How can we identify it without ML?
Q2 using DSA? – (hard)

C2 • Supervised ML : training set of images with known labels

↓

Neural ML algorithm

↑

features : (i) Hypothesis class – why? necessary?
(ii) loss function – for accessing how well model works
(iii) optimization method – helps in determining set of params which help in reducing the sum of losses

↑ (Starts with random init) h( ) ← (Model)

(Later) Foundational Models

C3 Logistic Regression – Theory

- What's the setting? – k classes to learn

ex. for MNIST digit identifying

$n = 28 \times 28 = 784$ – dimension of input data

$k = 10$ – no. of different classes/labels

$m = 60,000$ – # points in the training set

- Is random splitting of data into always good? (No)

train test

for time series: as it is future thing ⇒ 60% data for training etc

Step 1:

- Hypothesis function selected: Linear H.F

$h_\theta(x) = \theta^T x$

θ: random → don't start with 0 directly – for complicated optimisation

properties of h_θ

x is input

$$h_\theta(x) = \underset{k \times 1}{\theta^T} \; \underset{n \times 1}{x} \quad \text{——} (1)$$

x is projected from n dimensional to k dimensional

→ it takes time to compute

→ we need to do parallelize it

→ matrix notation used

- Matrix based notation: (Parallel computation)

$x \in \mathbb{R}^{m \times n}$, $X = \begin{bmatrix} - & x^{(1)T} & - \\ & \vdots & \\ - & x^{(m)T} & - \end{bmatrix}$ $\quad y \in \{1, \ldots, k\}^m$ $= \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$

$$h_\theta(x)_{m\times k} = X_{m\times n}\ \Theta_{n\times k} \quad \text{——} \quad (2)$$

What's the advantage of (2) over (1)?

→ faster

→ GPU — parallel hardware (else no benefit in CPU)

Step 2: (This is done to evaluate)

• Loss function selection #1 : classification error

$$\ell_{err}(h(x), y) = \begin{cases} 0 & \text{correct} \\ 1 & \text{incorrect} \end{cases}$$ → i.e if $\arg\max_i h_i(x) \neq y$

What's wrong with this?

⇒ ex. $n = 1000$

if correct = 950

incorrect = 750

(random if we scale up)

⇓

loss function is not optimiser friendly : as not differentiable

$\arg\max_i h_i(x)$ ⇓ output of h at x_i

⇓

What should be do now?

Loss #2: → to make all positive & sum to 1

• Can we do with probabilities?

$$Z_i = P(\text{label} = i) = \frac{\exp(h_i(x))}{\sum_{j=1}^{K} \exp(h_i(x))}$$

This is Right!! → ~~Softmax function~~ ≡ ~~exp~~ normalise $(\exp(h(x)))$

$h_i(x)$: -0.5 -0.2 0.1 0.5

↓

To handle with this, we go for

⇓

(as minimising) ↓ ↙ for numerical consistency

$$L_{CE}(h(x), y) = -\log P(\text{label} = y)$$

$$= -h_y(x) + \log \sum_{j=1}^{k} \exp(h_j(x))$$

~~Softmax~~ → Cross entropy

~~Softmax function!!~~ (*) (Our focus)

Optimiser friendly loss → $(\overline{w})$ ↑ ↓ $(\overline{w})$

⇒ exponential function in $(\overline{w})$ as any → (pos) out

~~NOTE: Softmax & cross entropy same!!~~

Step 3: (Optimiser) (This is done to optimise the optimal)

What to do with loss function?

⇒ minimize:

$$\frac{1}{m} \sum_{i=1}^{m} L\left(h_\theta(x^{(i)}), y(i)\right)$$

for cross entropy: L_{ce}

MIDSEM

Backprop
Gradient Descent
(By hand)

C-4 Gradient Descent Discussed

- We get optimal Θ_{nxk}

(We start with loss first)

- for optimization: we use gradient descent

$L(w)$: When can we apply GD? $w \approx \Theta$

(Cost loss)

- Convex
- Continuous
- differentiable

How to do it? (optimisation part)

Algo: → how do we find $(\Theta)_{opt}$? Start with random → w_0

Some # iterations: until we converge (repeat),

at each iteration: i) $\vec{w}^{t+1} = \vec{w}^t + \vec{s}$

ii) how do we know we are improving? → if $\|\vec{w}^{t+1} - \vec{w}^t\|_2 < \epsilon \Rightarrow$ converging

- If we don't know Loss: L

if we want to do: minimize $L = \sum_{i=1}^{n} L(\vec{w})$

How would we do it? ~~(We can't if we don't know, we need to know it)~~

$L(\vec{w} + \vec{s}) < L(\vec{w})$ ← How to ensure? during optimisation?

⇓

Taylor series expansion

classmate
Date
Page

Taylor series

gradient

$$l(\vec{w}+\vec{s}) \approx l(\vec{w}) + \underbrace{\vec{g}(\vec{w})^T \vec{s} + \frac{1}{2}\vec{s}^T H(\vec{w})\vec{s}}$$

High order terms : req different methods to Optimize (not GD)

$$g(\vec{w}) = \nabla(L(\vec{w}))$$

04/08/25

C1 (Continuing Gradient Descent)

- If I don't know $\log(\cdot)$, how do I ensure the loss always reduce? $\Rightarrow$ minimize L = but Not much aware about L (as in dunno if convex)

$L(w)$

$\Downarrow$

Step 1: Construct simpler tractable functions

Approach: Taylor Approximation

- Take small step(s) in $L(w)$

(s not defined yet) $\|\vec{s}\|_2$ is small (norm) (*) — imp condition 1

- Then what will be new loss? (Break it down)

$\Rightarrow$ Initial loss = $L(w)$

New = $L(w+\vec{s})$ ← s is vector (or scalar product)

V1: $L(\vec{w}+\vec{s}) \approx L(\vec{w}) + \vec{g}(\vec{w})^T \vec{s}$

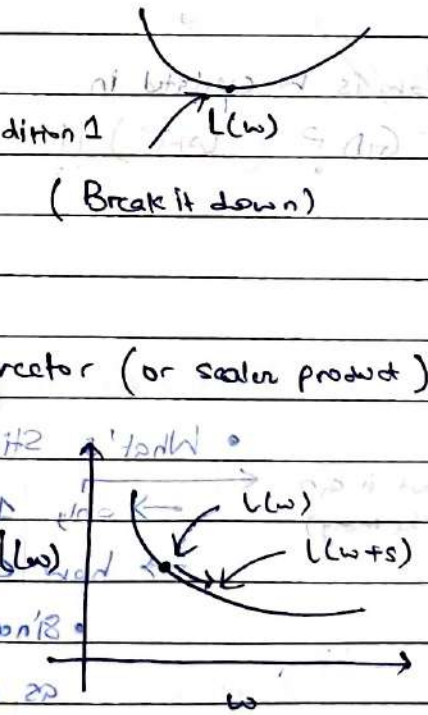
(GD) Where: $\vec{g}(\vec{w}) = \nabla L(\vec{w})$

V2: $L(\vec{w}+\vec{s}) \approx L(\vec{w}) + \vec{g}(\vec{w})^T \vec{s} + \frac{1}{2}\vec{s}^T H(\vec{w})\vec{s}$

Newton's Method

Condition 2: L twice differentiable (b)

Where: $H(\vec{w}) = \nabla^2(L(\vec{w}))$ ← Hessian



- Which one is more appropriate?

→ (2) is precise but there is a problem of H which requires function to be double differentiable (expensive)

⇓

we go with V1 (so henceforth gradient descent)

- What is exactly gradient descent in V1?

⇒ Our goal is min(L)

⇒ We need to ensure $L(\vec{w}+\vec{s}) < L(\vec{w})$

⇒ for this we need to do some trick in V1

NOTE: $\vec{g}(\vec{w})$ is fixed we can't do anything here

⇒ $\vec{s}$ needs to be ~~minimized~~ changed

⇒ $\boxed{\vec{s} = -\alpha \vec{g}(\vec{w})}$ (force it)

⇓

$$L(\vec{w}+\vec{s}) = L(\vec{w} - \alpha \vec{g}(\vec{w})) \quad \text{— (B)}$$

$$\approx L(\vec{w}) - \alpha \vec{g}(\vec{w})^T \cdot \vec{g}(\vec{w})$$

$\vec{g}(\vec{w})^T \cdot \vec{g}(\vec{w}) > 0$ (as sq of gradient)

α is in our hand

& we can force it to be > 0

⇒ RHS is always greater than LHS //

Q How is w updated in GD? $(w+s)$!!

Discuss the problem how to choose α (best = $1/t$)

- What's still the problem here?

→ only 1 local minima : used in classical model (but it can be many)

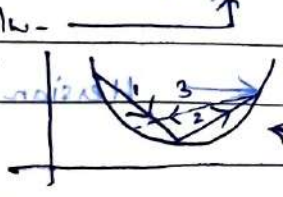
→ how can we determine α?

- Since (B) is an iterative process

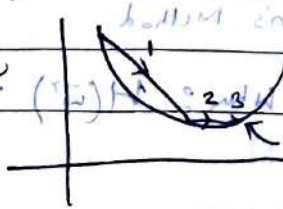
as we slowly $\vec{s}$ $\vec{g}(\vec{w})$

⇒ $\alpha = 1/t$ (first time) ← (errors initially high & α then lower)

What if α is always large?



← not good



← finally converge

classmate
Date
Page

- Note: previous section works for only 1 dimension

What if $x : n$ dimension? ← each dimension will have a curve associated with it
(input)

⇓

it should be independent ⇐ • (α should be associated with a dimension)

⇓

(fix coordinate learning rate) ← adaptation # of multi dimension, how to descent ← instead of gradient descent, we go for AdaGrad (adaptive gradient)

C2 # AdaGrad do we update? → multi dimension != multiple minima

w_0 and $\vec{z}$ $\forall$ d = $w_d^0 := 0$, $z_d = 0$ (running avg)

Assuming initially //

- Repeat until convergence

S1: $\vec{g} = \frac{\partial f(\vec{w})}{\partial \vec{w}}$ → where $g_{i,t} = \frac{\partial L}{\partial w_{i,t}}$

$\sum_{t=1}^{t} g_{i,t}^2 = z_{i,t}$ ← accumulated squared gradient for weight w_i if prev g_d was high

$\forall d$: $z_{d,t} \leftarrow z_{d,t-1} + g_{d,t}^2$

⇓

we need to take small steps

⇓

which this expression ensures //

- note: w_d → $\forall d$: $w_d^{t+1} \leftarrow w_d^t - \frac{\alpha g_d}{\sqrt{z_d + \epsilon}}$

i.e weights of a particular dimension depends on its gradient only!!

if $\|\vec{w}^{t+1} - \vec{w}^t\|_2 < \delta$ → converge

(better criteria)

C3 Newton's Method

⇓

- Since V1 not so powerful ⇒ to be precise we need to consider more terms

$$L(\vec{w} + \vec{s}) \approx L(\vec{w}) + \vec{g}(\vec{w})^T \vec{s} + \frac{1}{2} \vec{s}^T H(\vec{w}) \vec{s}$$

⇒ faster convergence

$$H(\vec{w}) = \begin{pmatrix} \frac{\partial^2 L}{\partial w_1^2} & \frac{\partial^2 L}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 L}{\partial w_1 \partial w_n} \\ \cdots & \cdots & \cdots & \cdots \\ & & & \frac{\partial^2 L}{\partial w_n^2} \end{pmatrix}$$

↳ always symmetric square matrix & positive semi definite

- H needs to be twice differentiable

↓ ↳ generally we don't for finding minima

but if we still want to use it

How to minimise it?

$$\frac{d\left(L(\vec{w}+\vec{s})\right)}{ds} = \arg\min_{s} \; L(\vec{w}) + \vec{g}(\vec{w})^T \vec{s} + \frac{1}{2}\vec{s}^T H(\vec{w})(\vec{s})$$

(That's why we don't use it)

S1 : LHS = 0

S2 : What should be $\vec{s}$? (NOTE: differential of RHS wrt. s)

$$\Rightarrow 0 = 0 + g(\vec{w}) + H(\vec{w})\vec{s}$$

$$\Rightarrow \vec{s} = -\left[H(\vec{w})\right]^{-1} g(\vec{w})$$

(i) Converges very fast if Hessian approximation accurate

(ii) but if fn is flat in some dimensions (second derivative ≈ 0), the inverse Hessian blows up → huge unstable steps → divergence risk

* Limitation of Hessian: here it's inverse which is not appealing ⇒ we don't use Hessian ∵ (as cubic for inversion) ⇒ therefore we go for G.D

Alternative: Hybrid version

(i) if we want to use Newton's method, we need to modify Hessian ← make it only diagonal matrix (approximation solution/method) ⇓ (here inverse becomes easier)

(ii) use SGD first to get close to the optimum then switch to Newton's method since it's quadratic approximation works better near the minimum ⇐ ⇒ rest all entries 0

(3) Limitations of GD ⇒ • if function not convex (multiple minima), it sticks

• quadratic approximation ⇒ using gradients & hessian combinely !! (i.e. Newton's method!)

classmate
Date 05/08/25
Page

- Gradient Descent ensure loss minimisation (not lets: it can go anywhere)
  ↳ can be applied in classification as well

Topic Discussed: Stochastic G.D

$$J(\theta) = \min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \underbrace{l_{ce}}_{\text{loss function}} \left(\underbrace{\theta^T x^{(i)}}_{h(x^{(i)})}, y^{(i)} \right) \quad \text{(i)}$$

(what does this expression represent?)

↓ apply gradient Descent

$$\Rightarrow \nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_{11}} & \cdots & \frac{\partial f(\theta)}{\partial \theta_{1K}} \\ & \ddots & \\ \frac{\partial f(\theta)}{\partial \theta_{n1}} & & \frac{\partial f(\theta)}{\partial \theta_{nK}} \end{bmatrix}_{n \times K}$$

n → no. of dimension, K → class

What's the problem with this method? (i.e apply grad on (i)?)

$$\sum_{i=1}^{m} \nabla_\theta l \left(h(x^{(i)}), y^{(i)} \right)$$

m → (we take all in gradient descent)

m is often large ⇒ time consuming process

⇒ parallelise it ← memory consumption

Multiple minima ⇒ get stuck

- So vanilla gradient descent not good

⇓

Since we want to generalise on test data

⇓

We move to Stochastic gradient descent

(B → data points with n dimension)

$B \ll m$

$x \leftarrow \mathbb{R}^{B \times n}$

$y \leftarrow \{1, \ldots, K\}^B$ (B×1 dimension, K → classes)

Repeat,

Sample a mini batch

Sample mini batches from m

then do gradient on it

(note: B changes on every iteration)

We don't get stuck

Update parameters, $\theta: \theta - \alpha \left(\frac{1}{B} \sum_{i=0}^{B} \nabla_\theta (l(h, y)) \right)$

(note: minima for every batch can be different) ? (thus the local minima is avoided)

How to compute the gradients manually

∇_θ (gradient symbol) $l_{ce}(\theta^T x, y) = z$ (calling it) ← Cross entropy Loss

$h(\,)$: hypothesis function

θ^T : $k \times n$ ($\equiv k \times m$) → for m data points

x : $n \times 1$ → Only one class or one dimension, 1 data point

$\Rightarrow \theta^T x \to y_{k\times 1}$

Called logits.

$$\Rightarrow \nabla_\theta \frac{l_{ce}(h, y)}{\partial h_i} \leftarrow *$$

$\Rightarrow$ Where

$$l_{ce}(h(*), y) = -h_y(x) + \log \sum_{j=1}^{k} \exp(h_j(x))$$

Method 1: Using h_i

$$\frac{\partial L_{ce}(h,y)}{\partial h_i} = \frac{\partial}{\partial h_i}\left(-h_y + \log \sum_{j=1}^{k} \exp(h_j)\right)$$

$$= -1\{i = y\} + \frac{\left(\frac{\partial}{\partial h_i} \sum_{j=1}^{k} \exp h_j\right)}{\sum_{j=1}^{k} \exp h_j}$$

When; $\{i = y\}$ it's an indicator

Are we supposed to use this?

$$= -1\{i = y\} + \frac{\exp h_j}{\sum_{j=1}^{k} \exp h_j}$$

($\to$ 0 when will be constant for $j = i$)

This is softmax ← This gives the probability

$$= -1\{i = y\} + \text{softmax}()$$

→ NOTE: e_y usually denotes the one-hot vector with 1 at the true class y

is it e_y? How do we get it?

$$= -e_y + z \quad (= \text{normalised}(\exp(h)))$$

$$\nabla_\theta \, l_{ce}(h, y) = z - e_y$$

$*$ → is the simplification one

Method 2: for generalising,

wrt wts $\nabla_\theta \, l_{ce}(\theta^T x, y)$

$$= \frac{\partial}{\partial \theta} l_{ce}(\theta^T x, y) \quad \{\text{use chain rule}\}$$

instead of $\theta_{(*)} \leftarrow \theta_i$ (something)

⟹ { Here the trouble was differentiating directly with θ }

$$= \frac{\partial}{\partial \theta^T x} l_{ce}(\theta^T x, y) \cdot \frac{\partial \theta^T x}{\partial \theta} \to (x^T \theta)$$

$$= \frac{\partial \, l_{ce}(h, y)}{\partial h} \cdot x^T$$

← note: we already computed this; h is the hypothesis function

$$= (z - e_y)_{k \times 1} \cdot x^T \quad \text{~~}n \times 1\text{~~} \; 1 \times n?$$

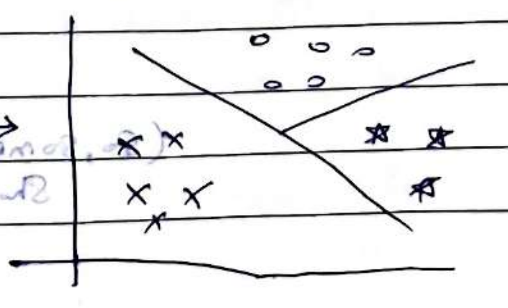
(1m) → $\theta \leftarrow \theta - \eta \nabla_\theta$

$$= x_{n \times 1} (z - e_y)^T_{1 \times k}$$

input, prediction, trans

- Why do we hate: $h(x) = \theta^T x$? ← linear hypothesis

(classical ML → works fine) classifier



but for non linear classification,

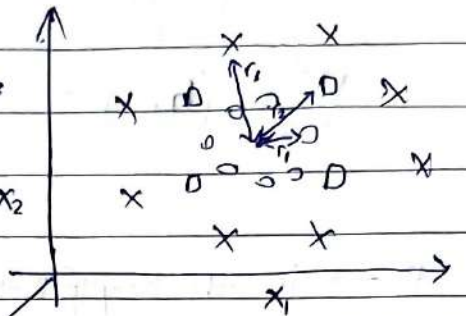
linear classifier doesn't work

⇒ we can do feature expansion

Given

x_1 & x_2 are dimensions

known



kernel trick (feature expansion) ⟸ $x_1^2 + x_2^2$ (introduced) $x_1^2 + x_2^2$

feature expansion can be done

$h(x) = \theta^T x$ (= $wx+b$) initially

$= \theta^T \phi(x)$

(Not so good)

That's why now we go for neural networks

- no need for feature expansion : self understood

Neural Networks for non-linear classification

08/08/25

- How to come up with good $\phi(x)$?
  - → kernel trick (m1) — i.e feature expansion
  - → learn by itself (m2)

- let's assume, $\phi(x) = W^T x$

$h_\theta(x) = \theta^T(\phi(x))$

$= \theta^T W^T x$ (where $\phi(x) = W^T x$)

$n_1 \times n_2$ $n_2 \times n_3$

$\Rightarrow \underset{n_1 \times n_3}{\tilde{\theta}} x$ ⟸ similar to previous hypothesis

(So, somehow it should be non-invertible) ⟸ (no benefit) {more specifically non-linear}

Topic: Neural Networks

C1: Why do we need ? → for non linear classification

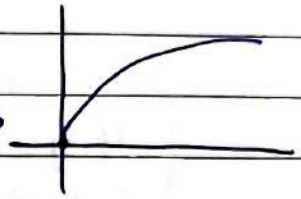
C2: (continuation of prev page)

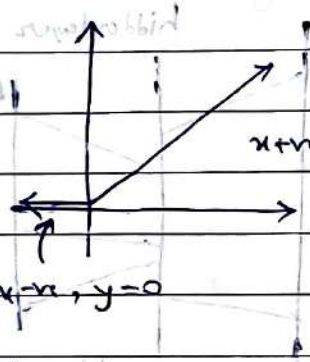
$$h_\theta(x) = \theta^T(\phi(x))$$

$$= \theta^T \sigma(W^T x) \neq \tilde{\theta}.x$$

(This can't be broken into first hypothesis)

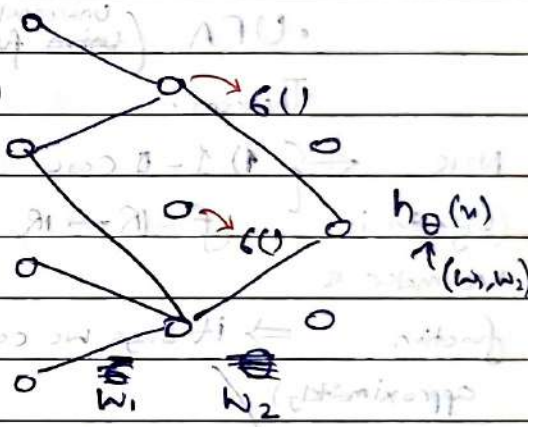
non linear function (like sigmoid) →





(Same as sigmoid)

sticking to ReLU for non-linear function



finding σ : example modelling

- How to optimise: $h_\theta(x) = \theta^T(\phi(x))$ — non-linear

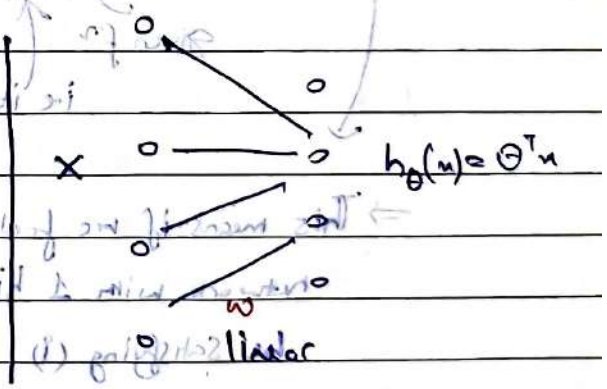
$= \theta^T \sigma(W^T x)$

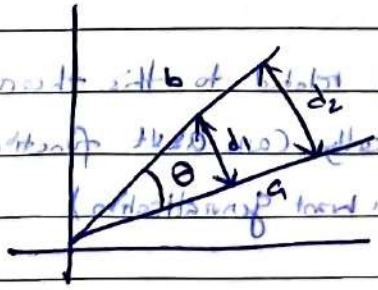
let $W \rightarrow$ be random

$\sigma \rightarrow$ cosine( )

$\theta \rightarrow$ learned

- first layer — cosine is random
- second/last layer — learned





← Note: how θ are going to be same for both d_1 & d_2

⇒ Cosine won't work for all scenario

• are the weight associated with layers?
(note the straight lines are ~~just a boundary depiction~~)

θ : W_1 W_2 ← both are parametrics (learned)

$h(x) = W_2^T\, \sigma(W_1^T x)$ ← 2 layer N.N (as 2 wts)

$\sigma(XW_1)W_2$; XW_1

⇒ How to find optimal value of W_1 & W_2?

i.e

$$W_1^{t+1} = W_1^t + S_1 \longrightarrow S_1 = -\alpha \vec{g}(W_1)$$

$$W_2^{t+1} = W_2^t + S_2 \longrightarrow S_2 = -\alpha \vec{g}(W_2)$$

} How to determine this?

↳ if you give me enough neurons & the right activation function setup, I can represent done through backprop the ~~true~~ $f(x)$ − even if I don't know it explicitly!

• UFA (~~theorem~~ universal function Approximation)

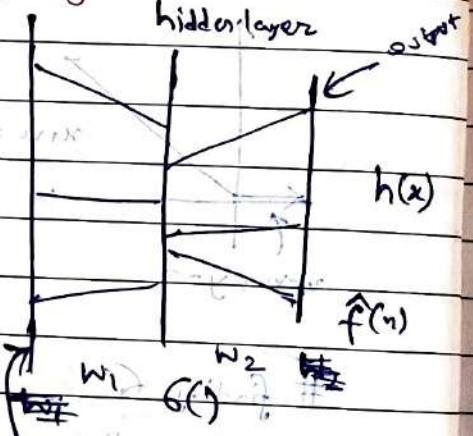
Theorem:

NOTE: (Objective is we make a function approximately)

{ 1) 1 − D Case

$f : \mathbb{R} \to \mathbb{R}$ | $\forall \epsilon > 0$ $\exists$ ~~$\epsilon \in \mathbb{R}$~~ X_1

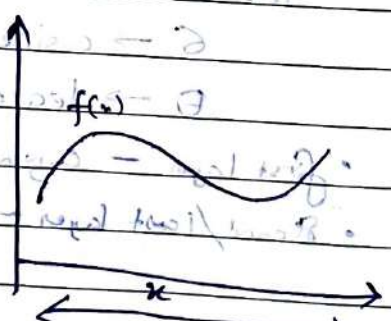
⇒ it says we can create 1 hidden layer



$$\max_{x \in D} |f(x) - \hat{f}(x)| < \epsilon \quad \text{---(1)}$$

given f^n ; output of N.N ; i.e it is bounded

⇒ This means if we feed x to the ~~linear~~ neural network with 1 hidden layer, will satisfying (1)

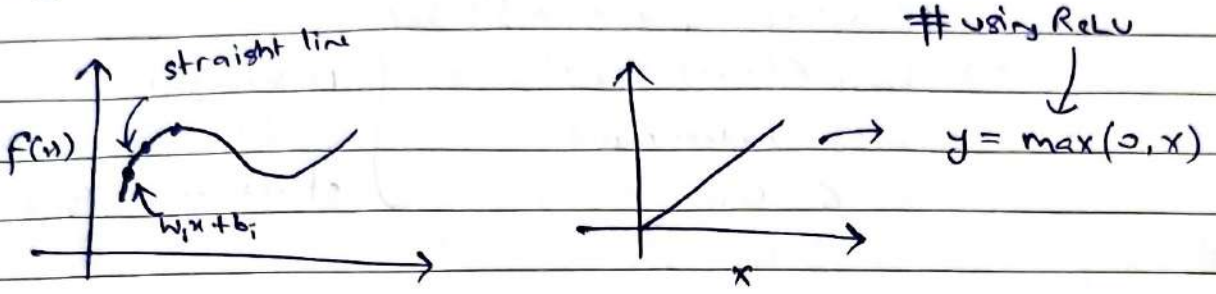


• There is some problem related to this theorem:

⇒ do we really care about function approximating accurately?

↳ (we want generalisation)

- Why is 1 layer hidden very powerful?



$$\hat{f}(x) = \sum_{i=1}^{d} \pm \max\{0, w_i x + b_i\}$$

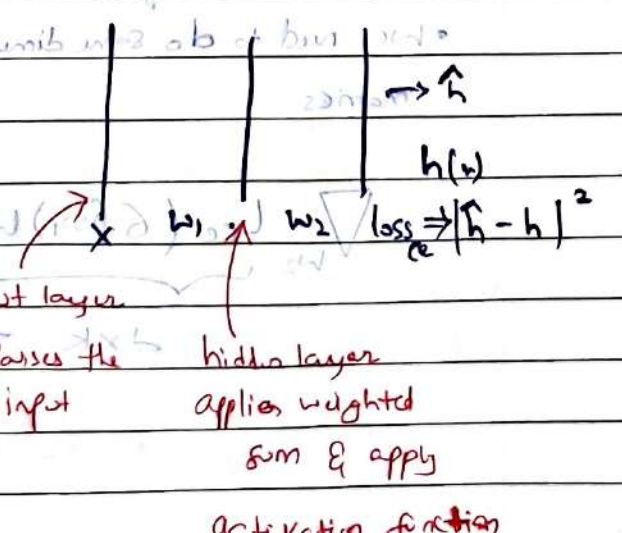
(This can map $f(x)$ very closely)

$y = mx + c$ → here we have to find optimal w & b

→ universal function approximator: Says give me the function, then I will add small² linear functions & approximate the function provided w_i & b_i are already given (computed using back prop)

- ~~do~~ we know $f(x)$ then? — not exact mathematical formulation
- ~~Why~~ would we compute if we know? — Sometimes expensive so approximation helps & here we make it learned from the examples we have //
- Defining loss ← computed in final layer ← (no layer specific loss)

$$h(x) = w_2\, \sigma(x w_1)$$



→ is the hypothesis function

Major focus: how to do weight updates → back propagation

#Neural Network

$w^{t+1} = w^t + s \quad ; \quad s = -\alpha g'(w)$

$\Rightarrow l_{ce}\left(\sigma(Xw_1)w_2, y\right)$

w_1 : random normal

σ : Cosine

Not very good ⇓

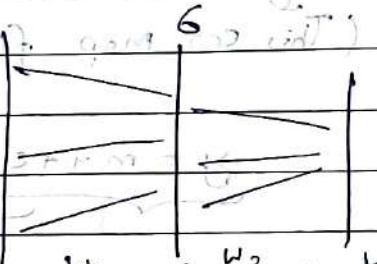
- how to get $g(w_1)$ & $g(w_2)$

#finding the right W_1 & W_2

- how do we train a neural network? (a multi layer NN)

→ What are the principles?

⇒ use backpropagation



need to compute gradients wrt W_1 & W_2

$\nabla_{\{W_1, W_2\}} l_{ce}\left(\sigma(Xw_1)w_2, y\right)$ X

$g(w_2) = \frac{\partial l_{ce}(\sigma(Xw_1)w_2, y)}{\partial w_2}$ ← for layer L_2

$= \frac{\partial l_{ce}(\sigma(Xw_1)w_2, y)}{\partial \sigma(Xw_1)w_2} \cdot \frac{\partial \sigma(Xw_1)w_2}{\partial w_2}$

$\Rightarrow w_2 = d \times k$

output of hidden layer : $n \times d$

$= \frac{\partial l_{ce}(h(), y)}{\partial (h)} \cdot \sigma(Xw_1)$ (from previous class)

$= (S - I_y) \cdot \sigma(Xw_1)$

S: Softmax, I_y: identity, $n \times k$, $\sigma(Xw_1)$: $n \times d$

Batch size of n (n samples)

- We need to do some dimension matching to represent it in form of matrices

$\nabla_{w_2} l_{ce}\left(\sigma(Xw_1)w_2, y\right) = \sigma(Xw_1)^T (S - I_y)$

$d \times k$: To compute gradient, wrt W_2 we req

$d \times n$: Activation o/p of previous layer

$n \times k$: softmax output from L_2 layer

classmate
Date
Page

- We don't have the loss for L_1 layer but for L_2 (overall at the end)

⇓

for layer 1 →

$$\frac{\partial \ell_{ce}(\sigma(xw_1)w_2, y)}{\partial w_1} = \frac{\partial \ell_{ce}(\sigma(xw_1)w_2, y)}{\partial(\sigma(xw_1)w_2)} \cdot \frac{\partial(\sigma(xw_1)\cdot w_2)}{\partial(\sigma(xw_1))} \cdot \frac{\partial(\sigma(xw_1))}{\partial(xw_1)} \cdot \frac{\partial(xw_1)}{\partial w_1}$$

$$\nabla_{w_1} \ell_{ce}(\sigma(xw_1)w_2, y) = \underset{m\times k}{(S - I_y)} \cdot \underset{d\times k}{w_2} \cdot \underset{m\times d}{\sigma'(xw_1)} \cdot \underset{m\times n}{x}$$

Matching the dimension:

Weights of next layer

$$= \underset{n\times m}{X^T} \cdot \Big(\underset{m\times k}{(S-I_y)} \underset{k\times d}{w_2^T} \odot \underset{m\times d}{\sigma'(xw_1)} \Big) \quad \text{— (A)}$$

$m \times d$

gradient depends on (input)

output of final layer

(output activation)

backward pass much more computationally expensive than forward pass : as memory hungry — as seen in (A)

more multiplication

- Scaling upto more layers

$i = 1 \dots L$ (L layers) (for forward pass) (*)

$z_{i+1} = \sigma_i(z_i w_i)$ ⟶ let $z_1 = x$

output for each layer

activation output

last layer

$$\frac{\partial \ell(z_{L+1}, y)}{\partial w_i} = \frac{\partial \ell}{\partial z_{L+1}} \cdot \frac{\partial z_{L+1}}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdots \frac{\partial z_{i+2}}{\partial z_{i+1}} \cdot \frac{\partial z_{i+1}}{\partial w_i}$$

$$G_{i+1} = \frac{\partial \ell(z_{L+1}, y)}{\partial z_{i+1}}$$

(why last ⟹ the loss taken as z_{L+1} w.r.t $L+1$?)

here we need the gradient flow from right to left

isn't this the (second output?)

↳ NO (we take activation applied one)

activation

(weight of this layer previously!!)

What if we want to do gradient wrt z_i?

$$\frac{\partial L(z_{L+1}, y)}{\partial z_i} = G_i = G_{i+1} \cdot \frac{\partial z_{i+1}}{\partial z_i}$$

wrt activation

gradient of ith layer is dependent on (i+1) i.e the next layer

$$= G_{i+1} \cdot \frac{\partial \sigma_i(z_i w_i)}{\partial z_i w_i} \cdot \frac{\partial z_i w_i}{\partial w_i}$$

$$= \underset{m \times n_{i+1}}{G_{i+1}} \cdot \underset{m \times n_{i+1}}{\sigma'(z_i w_i)} \cdot \underset{n_i \times n_{i+1}}{w_i} = \underset{m \times n_i}{G_i}$$

Dimension matching

gradient from prev layer (right to left)

$$\Rightarrow \underset{m \times n_i}{\nabla_{z_i} L(z_{L+1}, y)} = \left(\underset{m \times n_{i+1}}{G_{i+1}} \odot \underset{m \times n_{i+1}}{\sigma'(z_i w_i)} \right) \underset{n_{i+1} \times n_i}{w_i^T}$$

called incoming gradient

$$\Rightarrow \underset{n_i \times n_{i+1}}{\nabla_{w_i} L(z_{L+1}, y)} = G_{i+1} \cdot \frac{\partial z_{i+1}}{\partial w_i}$$

$$= G_{i+1} \odot \frac{\partial \sigma_i(z_i w_i)}{\partial z_i w_i} \cdot \frac{\partial z_i w_i}{\partial w_i}$$

$$= G_{i+1} \odot \sigma'(z_i w_i) \cdot \underset{\text{activation output of current layer}}{z_i}$$

$$= z_i^T \left(G_{i+1} \odot \sigma'(z_i w_i) \right)$$

(aka ~~backtracks~~ ~~jacobian~~ product)
Jacobian

Vector

$n_i \times m$

(needs to be stored)

$m \times n_{i+1}$

incoming backward gradient (difficult - annoying)

NOTE: quadratic error mathus how because heavy penalty for large errors!

(*) requirements for F.P

requirements for B.P : i.e what all req from F.P ?

→ store activation output (memory intense)

Multiplication req ? — how many ? (lower bound)

⇒ 2× f.p

14/08/25

Type of differentiation

• NOTE:

BP

$$G_{L+1} = \nabla_{Z_{L+1}} l(Z_{L+1}, y) = S - I_4$$

G_i ← same as last class

• Gradient Descent & BackProp differs → still gradient descent but includes both backpass & forward pass

↓ ↓

→ Update only for last layer → Single minima

• for all layers, we can update {multiple minima}

• we can apply vanilla gradient descent as well but stochastic to optimise the memory

Next Topic: Automatic Differentiation (AD)

• Why do we need ? — apply backprop in much{more principle way} (??)

• how to compute gradients in software ?

i) Numerical differentiation

$$\Rightarrow \frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon\theta_i) - f(\theta)}{\epsilon}$$ (aka first order (linear) approximation) : error is $O(\epsilon)$

⇓ more exact way to do it

$$\frac{\partial f(\theta)}{\partial \theta_i} = \frac{f(\theta+\epsilon\theta_i) - f(\theta-\epsilon\theta_i)}{2\epsilon}$$ (this is second order (quadratic) approximation) : more efficient as error is $O(\epsilon^2)$

• Problems: sensitive to ϵ (we can choose any ϵ)

classmate
Date
Page

→ here what we are trying to do is checking if the computed gradient is correct or not by calculating the slope!!

ii) Even though it has problem, we use it for gradient checking {as in if correctly computed}

$$\delta^T \nabla_\theta f(\theta) = \frac{f(\theta+\epsilon\delta) - f(\theta-\epsilon\delta) + O(\epsilon^2)}{2\epsilon}$$

(What is this? → δ^T) ⇒ it's a direction vector

$\nabla_\theta f(\theta)$: we get from software

it's a dot product

$\frac{2\epsilon}{}$: computed by hand

iii) Symbolic Differentiation

Naive method

$$f(\theta) = \prod_{i=0}^{n} \theta_i \qquad \frac{f(\theta)}{\partial \theta_k} = \prod_{j \neq k}^{n} \theta_j$$

→ Redundancy

→ form graph method

for optimisation cost : $\underbrace{n(n-1)}_{\approx n^2}$ multiplies to compute all partial gradients

(iii) Computational graph : (we need to go back so to understand how computationally heavy the expression is)

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$

⇒ node : intermediate value in the computation

edge : input-output relationship

$v_1 = x_1$

$v_2 = x_2$

$v_3 = \ln v_1$

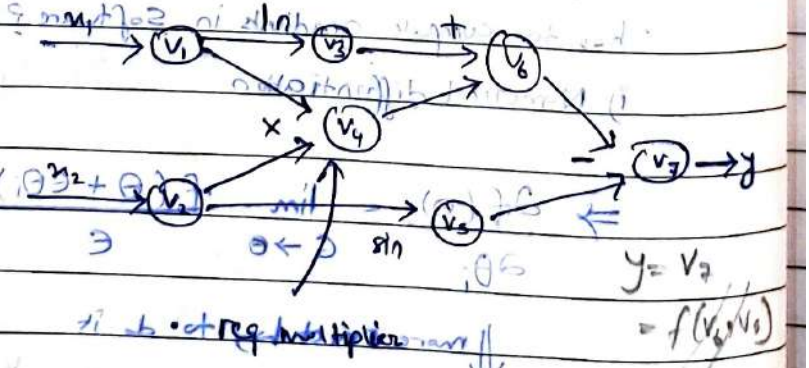
$v_4 = v_1 v_2$

$v_5 = \sin v_2$

$v_7 = v_6 - v_5$

$v_6 = v_3 + v_4$

(# forward pass)



• Computationally hungry

- now we got y
- we need to apply backward pass: (on loss)
- how to compute the gradient?

forward mode AD
↳ cost is $O(n)$ ← input

→ M1: from left (we get the outputs of next layer easily)

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \qquad \dot{v}_i' = \frac{\partial v_i}{\partial x_2}$$

$$\Rightarrow \dot{v}_1 = 1 \quad \left(\frac{\partial x_1 = 1}{\partial x_1}\right)$$

$$\dot{v}_2 = 0 \quad \left(\frac{\partial x_2}{\partial x_1}' = 0\right)$$

$$\dot{v}_3 = \frac{1}{v_1} = \frac{\dot{v}_1}{v_1} = 0.5 \quad \left(\frac{\partial v_3}{\partial x_1} = \frac{\partial \ln v_1}{\partial x_1} = \frac{\partial \ln v_1}{\partial v_1} \cdot \frac{\partial v_i}{\partial x_1}\right)$$

$$\dot{v}_4 = \dot{v}_1 v_2 + \dot{v}_2 v_1 = 5$$

$$\dot{v}_5 = \cos v_2 \cdot \dot{v}_2 = 0$$

$$\dot{v}_6 = \frac{\dot{v}_1}{v_1} + \dot{v}_1 v_2 + \dot{v}_2 v_1 = \dot{v}_3 + \dot{v}_4 = 5.5$$

$$\dot{v}_7 = \dot{v}_6 - \dot{v}_5 = 5.5$$

$$\Rightarrow \frac{\partial y}{\partial x_1} = \dot{v}_7 = 5.5$$

- What is the problem with this method?

⇒ If lots of input ≡ huge storage. → soln is reverse mode AD

↳ as (n) forward AD passes {for ~~each~~ input — 1 pass}

- What is the cool thing about it? ↓ to compute all gradients

⇒ simplicity & low memory usage

⇒ if $n < K$ for example $n = 3$, $K = 1000$

but for $n > K$ ⇒ reverse mode AD (good)

→ backprop

→ M2 : Reverse mode AD ← cost ∝ O(K) (output)

(Autograd)

- forward evaluation remains same
  - ↳ additionally : Compute adjoint i.e $\frac{\partial y}{\partial v_i} = \bar{v}_i$

(for gradient computation)

going from right → left

$$\bar{v}_7 = \frac{\partial y}{\partial v_7} = 1$$

$$\bar{v}_6 = \frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} \quad \{\text{use chain rule}\}$$

$$= \bar{v}_7 \cdot \frac{\partial v_7}{\partial v_6}$$

$$= 1 \cdot (1) = 1 \quad (\text{NOTE: } v_7 = v_6 - v_5)$$

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_5} = \bar{v}_7 \cdot \frac{\partial v_7}{\partial v_5}$$

$$= 1 \cdot (-1) = -1$$

(Since partial differentiation so independent) i.e v_5 Constant //

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_4}$$

$$= 1 \cdot (\bar{v}_6 \cdot 1)$$

$$= 1 \cdot 1 = 1 //$$

$$\bar{v}_3 = \frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_3}$$

$$= \bar{v}_6 \cdot 1 = 1$$

(i.e it is coming in for 2 different nodes) → $v_2 \to v_4$, $v_2 \to v_5$ (*)

2 paths exist for v_2 ←

$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_2}$$

Similary for v_1 & both contributions we need to consider

* $\bar{v}_2 = \bar{v}_5 \frac{\partial v_5}{\partial v_2} + \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2}$ (we say v_5 & v_4 both contribute equally to v_2)

$\Rightarrow \bar{v}_2 = \bar{v}_5 \cos v_2 + \bar{v}_4 \cdot v_1$

$= 1.716$

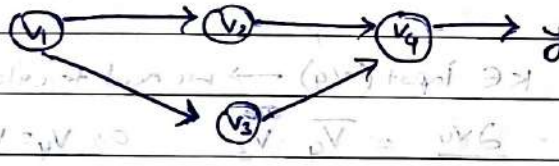
* $\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} + \bar{v}_2 \frac{\partial v_2}{\partial v_1}$

$= \bar{v}_4 v_2 + \bar{v}_3 \frac{1}{v_1}$

$= 5.5$

Derivation for the multiple pathway case!

How to arrive at * ?



$y = f(v_2, v_3)$

$\bar{v}_1 = \frac{\partial \bar{y}}{\partial v_1} = \underbrace{\frac{\partial f(v_2, v_3)}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_1}}_{\text{Partial adjoint}} + \frac{\partial f(v_2, v_3)}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1}$

Partial adjoint ← can be computed for every edge

Partial adjoint: $\bar{v}_{i \to j} = \bar{v}_j \frac{\partial v_j}{\partial v_i}$ Where $i \to j$ denotes an outgoing edge from i

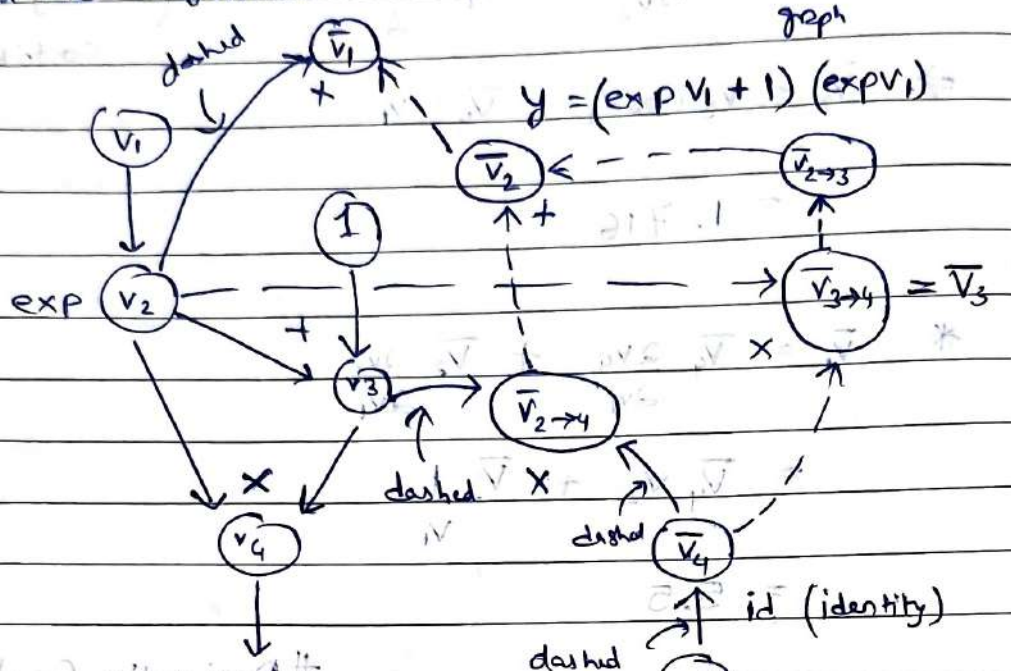
$\bar{v}_i = \sum_{j \in next(i)} \bar{v}_{i \to j}$ ← Sum all outgoing edge partial adjoint

Running of algorithm

→ combining all precomputed adjoints (partial)

computational graph

- Reverse AD algorithm



$i = 4$

Step 1

- If I am at v_4, sum of all partial adjoints but there are (none) → (adjoint of $y = 1$)

- $v_2 \& v_3$ (for all $k \in$ input (v_4) → we need to calc partial adjoint

$$\bar{v}_{2\to 4} = \bar{v}_4 \cdot \frac{\partial v_4}{\partial_2} = \bar{v}_4 \cdot v_3 \quad \text{as } v_4 = v_2 \cdot v_3$$

$$\bar{v}_{3\to 4} = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_3} = \bar{v}_4 \cdot v_2 = \bar{v}_3$$

(as partial adjoint is the read adjoint)

$i = 3$

Step 2

$$\bar{v}_{2\to 3} = \bar{v}_3 = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = \bar{v}_3 \cdot \frac{\partial(\exp v_1) + 1}{\partial \exp v_1} = \bar{v}_3$$

$i = 2$

Step 3

$v_2 \to v_3, v_4$ (input to)

$$\bar{v}_{1\to 2} = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} \quad \Rightarrow \text{as } v_2 = \exp(v_1)$$

$$= \bar{v}_2 \cdot \frac{\partial \exp(v_1)}{\partial v_1} = \bar{v}_2 \cdot v_2 = \bar{v}_1$$

NOTE: Here we are calculating partial adjoints beforehand ⇒ time complexity reduced !

• No need to arrive at v_1 as it's the starting node!

NOTE: if the values change for $V_1, V_2, V_3 \ldots V_n$ then gradients are gonna change (so we may need to recompute)

But in here if we see the computational graph, we have the exact relation ⟶ reducing the no. of unnecessary computation (of grads)

hence it is beneficial than

conventional backprop {~~which one using on?~~}

• ↳ Reverse mode AD by extending computational graph

⟶ req both forward & backward pass

↳ no need of backward pass as already pre-computed

(??)

→ Since backprop is implementation of reverse mod AD, what's difference?

↳ here I think it was, this? once that graph is plotted it's fixed (i.e the graph is fixed once the function is written) ⟶ on changing the input : only the numerical values flowing through those formulas change, not the chain rule itself

Here prob is: when we do GD, W update happens by looking at steepest path/dirn !!

classmate
Date 19/08/25
Page

- Optimization (# Fully connected Network) : FCN

memory hungry → so we use broadcasting ie instead of storing $1_{m\times 1}$

$$Z_{i+1} = \sigma_i(Z_i W_i + 1 b_i^T)$$

$Z_{i+1} \in \mathbb{R}^{m\times n}$, $1_{m\times 1}$, $b_i^T{}_{1\times n}$

(note: it might be last layer is non activation only)

⇒ we just repeat b matrix (m) times without storing it

↳ key Qs on (FCN)

Q How do we choose no. of layers & no. of neurons per layer?

Optimization

(i) considering a convex function, (gradient descent again discussed)

↳ next, Adagrad

Nesterov's method

Loss fn: $f(\theta) = \frac{1}{2}\theta^T P\theta + q^T\theta$ (quadratic)

→ keep track of past actions / directions) to avoid fig 1 kind of movement

(# level set diagram first fig)

(ii) Momentum

$$v_{t+1} = \beta v_t + (1-\beta)\nabla_\theta f(\theta_t)$$

$\beta \in [0,1]$ → gives better well informed dirn

(dec with time)

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

previously it was gradient

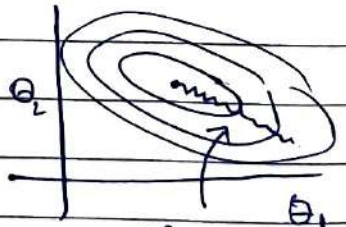
→ minima θ_1

→ if we go out, loss inc

let $v_0 = 0$

$t=1$, $v_1 = (1-\beta)\nabla_0 + \theta_t$ now it's momentum

$t=2$,



after momentum

If $\beta = 0$ → no momentum; same like previous case

classmate
Date
Page

crude momentum

Pb 1 → Slow start when using momentum as $v_0 = 0$ (smaller in initial iterations than in later ones)

↓ for larger Network it is not good

⇓

So modifying the wt update

unbiased momentum

$$\text{Soln 1 } \theta_{t+1} = \theta_t - \alpha v_{t+1}$$

v_{t+1} ← unbiasing factor

$\left(1 - \beta^{t+1}\right)$

(gives a big jump at start)

for $t = 0$, $\left(\frac{(1-\beta)}{(1-\beta)} \nabla \ldots\right)$

Nesterov Momentum

benefit: {converges faster & oscillates less}

updated wts

$$\Rightarrow v_{t+1} = \beta v_t + (1-\beta) \nabla_\theta f\left(\theta_t - \alpha v_t\right)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

previously only θ_t

NOTE: Momentum used for wt update

↳ till now 2-D wts

Pb 2 ↳ but about high dimensional wts?

⇒ multiple minimas for diff wts

⇓

That's why we go for stochastic update (i.e batch update in epochs)

Same α for θ_1 & θ_2 {in vanilla method}

(both tells different α for each dimensions we req)!

⇓

Adagrad in momentum update (Adam)

Adam (most popular optimising method)

$$v_{t+1} = \beta_1 v_t + (1-\beta_1) \nabla_\theta f(\theta_t)$$

moving avg → $$V_{t+1} = \beta_2 V_t + (1-\beta_2) \left(\nabla_\theta f(\theta_t)\right)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha v_{t+1}}{\left(\sqrt{V_{t+1}} + \epsilon\right)}$$

if $V_t \uparrow$, slow update else fast update

$\left(\sqrt{V_{t+1}} + \epsilon\right)$ ⇒ To avoid 0 case

optimisation: On top of this, stochastic gradient descent (instead of doing on small moments, I do multiple small moments)

Initialisation of wts

- can we initialise to 0? → no
  - ↳ wt's the pb? — backpropagation will become impossible
  - If we initialise to (all gradients will be 0) — as wts are 0
- ↑ random wts → distribution of wts don't change much
  - same as original

⇒ it's very important step: what's a good initialisation?

Start with $N(0, \sigma^2 I) \ni W_i$ ← normal wts

depends on 2 factors

- What should be the choice of σ^2? ⇒ $2/n$ (optimal)

Forward: → How $2/n$ good is good?
→ Why normalisation req?

classmate Date 25/08/25 Page

→ for stabilizing & speeding up training
Normalisation, Regularisation → for preventing overfitting
(initialisation ctd...)

W | $W^T x$ ← no activation (linear)
x

Why is $2/n$ optimal? → (no ReLU activation considered) → in linear activation — $1/n$ good but not in ReLU

(Check it out) Reasoning: Why Wt updates should be initialised properly — as bad initialisation may lead to bad result/final value (proof)

- Independent R.V: $x \sim N(0,1)$, $w \sim N(0, 1/n)$

$$E[w_i x_i] = E(x_i)\,E[w_i] = 0$$

$$Var[w_i x_i] = Var[x_i]\,Var[w_i] = 1/n$$

$$\Rightarrow E[w^T x] = 0 \quad (n \times 0)$$
$$Var[w^T x] = 1 \quad (n \times 1/n)$$

- $w^T x = \sum_{i}^{n} w_i x_i$ ($w^T x \to N(0,1)$ by central limit theorem)

Since, no activation non linear ones,

$\Rightarrow$ if $z_i \sim N(0, \frac{1}{n} I)$ & $W_i \sim N(0, \frac{1}{n} I)$

$$z_{i+1} = W_i^T z_i \sim N\left(0, \frac{1}{n} I\right)$$

→ as it sets the variance to half!!

i.e every layer will have normal distribution with mean 0 & var 1

NOTE: if we use ReLU, then half component ⟸ set to 0 → to compensate for loss,

$$W \sim N\left(0, 2\left(\frac{1}{n}\right)\right)$$

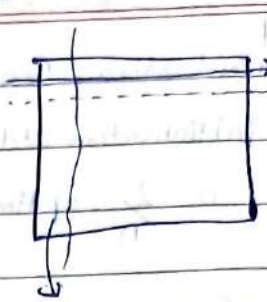
NOTE: initialisation — messes up the network training
It can persist throughout layers

(need for normalisation)

(From presentation) Focus: Types of Normalisation
↳ Their benefits & pbs
Why regularisation & types!!

Batch norm completed (?)

a) Layer Norm (mean (taken))



$$\hat{z}_{i+1} = \sigma_i(W_i^T z_i + b_i)$$

$$\hat{z}_{i+1} = \frac{\hat{z}_{i+1} - E(\hat{z}_{i+1})}{(Var[\hat{z}_{i+1}] + \epsilon)^{1/2}}$$

for batch norm, we take along a batch

(gradients?)

- Outputs & activation not dependent on normalisation
- What's the pb here? (if we go std fully connected network)

⇒ used in transformers

⇒

b) Batch Norm

- Where is it useful?

→ original data
↓
noise added
↓ to change distribution
↓ distinguishing difficult
⇒ but model trained on out of ordered data during using batch norm ⇒ good result!

CNN as well
→ use batch norm
→

So if we apply it in batches since using a dimension (i.e applied) on norm between 1

for layer norm on batches

Problem: ⇒

~~a)~~ Mini Batch Dependence is core characteristic of Batch norm where the normalization of a single data point is calculated using the statistics (mean & variance) of the specific, random mini batch it happens to be in. *

- Regularisation
  - → more parameters in network
  - → solution space ~~large~~ less ⇒ memorise the data, but not generalisation

Regularisation limits the complexity of the network → Implicit / explicit

$$\underset{W_{1:D}}{\text{minimise}} \ \frac{1}{m}\sum_{i=1}^{m} \ell\left(h_{W_{1:D}}(x^{(i)}), y^{(i)}\right) + \frac{\lambda}{2}\sum_{i=1}^{D} \|W_i\|_2^2$$

Type 1

→ L_2 regularisation aka weight decay (why?) ← check it out

(models with smaller weights are simpler & less prone to overfitting)

$$W_i = W_i - \alpha \nabla_{W_i} L(h(x), y) - \alpha \lambda W_i$$

$$= (1 - \alpha\lambda) W_i - \alpha \nabla_{W_i} L(h(x), y)$$

→ prevents exploding wts

⇒ because of this term $(1-\alpha\lambda)$ we say it's weight decay! (as slightly less than 1)

⇒ as it shrinks the value at each step

Type 2

⇒ Dropout → used during training only!! (similar to ensemble method)

- We randomly drop some neurons from layer (??)
- randomly set some fraction of the activations at each layer to zero

$$\hat{Z}_{i+1} = \sigma_i (W_i^T Z_i + b_i)$$

~~$(Z_{i+1})_j =$~~

→ DL Basics Completed ←

$$(Z_{i+1})_j = \begin{cases} (\hat{Z}_{i+1})_j / (1-p) & \text{with probability } 1-p \\ 0 & \text{with } p \end{cases}$$

How Batch Norm works

→ this method normalises the activations across the examples in a minibatch

* This makes the model's training sensitive to both the batch size & the other examples it is grouped with at each step!

classmate
Date 26/08/25
Page

#Here focus was CPU!!

A bit about Hardware

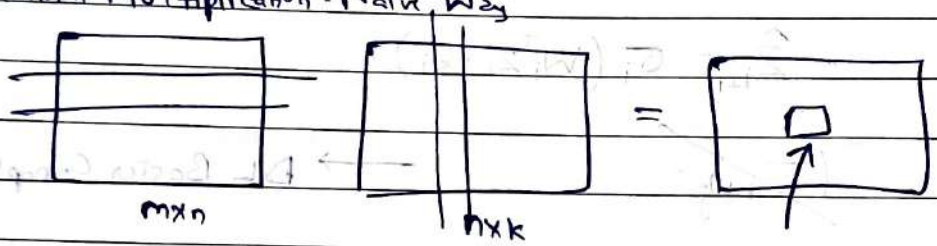
→ How to link DL to Hardware : focus

- MLP
- Application — CNN in hardware
  - Transformer
  - DL for inference
  - Optimal Training Methods
  - (.....)

→ Gain : in terms of energy
↑ Speed
(If we use hardware thing) no. of computations

- Matrix Multiplication : Naive Way

⇒



- To get this, n computations req

- What are the things ppl do while applying DL in hardware?

a) vectorization → (group computation better than single data computation)

how to add 2 arrays of length 256?

A : 256 × 256

B : 256 × 256

To get C : 256 × 256

Naive: Load all 256 one by one

Optimisation: Load 4 at a time ($256/4 = 64$)

b) How to store data/matrices in memory

→ to store 256×256 directly : long/heavy

T1 → Row Major – store row by row A[i,j] = Adata[i × A.shape[1] + j] (no. of columns)

T2 → Column " – " col by col (1 col after another)

T3 → Strides format – random position defined by strides (same offset)

A[i,j] = Adata[j × A.shape[0] + i]

↳ A[i,j] = A[data[i * A.strides[0] + j * A.strides[1]]

↑ jump size

↳ how does this help?

- a lot of slicing in array : there helps
  ex A[2:, :] ← skipping first row
- transpose
- broadcast } ← Stride = 0 i.e I can repeat reading it multiple times

↳ Downside : as memory access not continous
- vectorization harder
- (- - - - - - - -) many linear algebra operations may require compact in the array first

↑ i.e making it contiguous by forming a new array

c) What other optimisations exists?

- Parallelizing the operations (open Mp).
  (ex. loading data from 2 or different matrices at a time
  ↳ it depends on code if possible
  Lot in ML, since lot of matrix operation is possible

d) Matrix vector Multiplication (# CPU for DL)

→ Memory heavy

→ Naive way : $O(n^3)$: this is seen when same type of memory is used for storing/loading

⇓

improve the placement of data in different memory layers : called garnishing : CPU Thread

NOTE:

```
for i=0...n
  for j=0...n
    for k=0...n
      (⟶)
```

⇒ instead what we can do is, we can bring A both the inputs of B one at a time (tile by V → i.e bring set of data)

Registers

L1 cache – 0.5 ns

L2 " – 7 ns

DRAM – 200 ns

NOTE: naive matrix multiplication

for A & B → Load cost : 2 * dram speed * n^3

Register cost : 3 //

Register

d1) ^Tiled Matrix Multiplication

→ Blocks wise multiplication

→ Cost : n^3/V_2 for A's DRAM

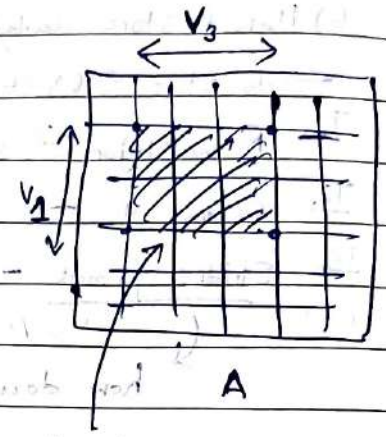
NOTE:

- func A[n_1/V_1][n_1/V_3][V_1][V_3]

  B[n_2/V_2][n_2/V_3][V_2][V_3] n^3/V_1 for B's DRAM

- dram → reg time cost = No. of iterations * cnt of data brought

- Load cost = time cost * speed!!



- loaded at a time

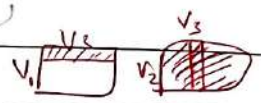
register cost increase fine!!

- $V_1 \times V_3$ for A's register cost
- $V_2 \times V_3$ " B's " "
- $V_1 \times V_2$ " C's " "

for B : $V_2 \times V_3$

for C : $V_1 \times V_2$ stored

d2) Cache line aware tiling



→ Use of caches previously not used

→ loads from DRAM can be reduced.

→ by using caches

Idea is:

from A → for n/b_1 times bring $b_1 \times n$

" B → for n/b_2 times bring $b_2 \times n$

constraint : $b_1 \times n + b_2 \times n \leq$ L1 cache size

(as the register part for doing the third for loop ←

- $b_1 \% V_1 == 0$
- $b_2 \% V_2 == 0$

reg) ↳ like $V_1 \times V_2$ at a time computed!!

L1 load cost

- L1 speed (instead of DRAM now)

$$\left(\frac{n^3}{V_1} + \frac{n^3}{V_2}\right) * L_1 \text{ speed}$$

DRAM load cost

- dRAM speed → $\left(n^2 + \frac{n^3}{b_1}\right)$ * DRAM speed

01/09/25

GPU for DL

- CPU & GPU together coexist
- Why GPU?

→ on CPU — we optimise using memory

but more? — parallelise multiplication 4 times
at most {if only 4 cores}

⇓

but ~~GPU~~ (like give) 4 controllers

→ which will req as single operation

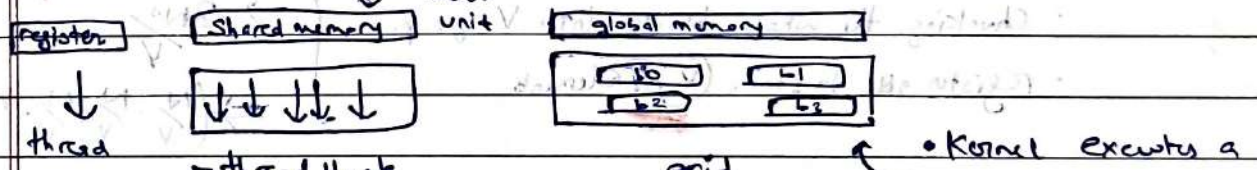
→ 1 controller enough

hence GPU { → more cores req!!
parallel

- massive computing units
- bare min instruction

→ Garnishing GPU hardware with caches

→ GPU programming model: SIMT — single instruction multiple threads

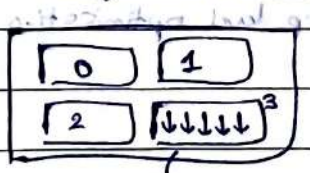


• have shared memory (within a block)

• executes same code, but can take different path

- When we launch a kernel what exactly happens?

→ Thread identified by blockid, gridid



(for global id) ↑

no. of blocks

• {blockDim.x * blockIdx + threadId}

↳ in the reference block

(5x3+4)

↑ threadId in the blocks

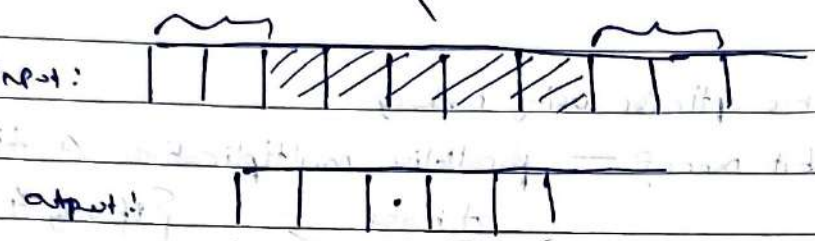
Total time

Learning thru record

use of hierarchy's memory to optimise task in GPU

→ like we did in CPU

example window sum ≅ (1D CNN) ← NOTE: the idea is multiple threads executing the same



⇒ each block will be computed at the same time by looking at radius left & right!!

if I want to compute only these 5

⇒ 2 from left & 2 from right only req

⇒ not whole input array

⇒ threads per block + 2 * radius

matrix multiplication on GPU

Case 1: register tiling

- Chunking the whole block into V groups
- register able to store (V) elements

here if no tiling ⇒ N^3

but if tiling of V ⇒ N^3/V

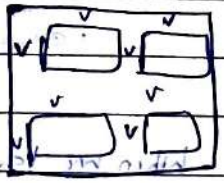
→ $2N^3/V$

⇒ $\left(\frac{N}{V} \cdot \frac{N}{V} \cdot \frac{N}{V} \cdot V \cdot V\right)$

Case 2: shared memory tiling

- bring L×S elements (> V)
- NOTE: L % V == 0 (req)

What about S ? ← (not req)



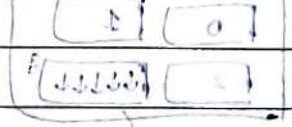
global → shared

copy = $2 * N^2/L$

shared → register = $2N^3/V$

- for each core → warp level optimisation also exist

↳ (skip slide 18)



Multiprocessing Stream → ability to use thread & register

↓ → Tradeoff ↳ thread register → more data brought in register but less threads

Thus we have autotune to find via L & V

block & shared memory ↳ global memory

↑ more data brought in memory than available blocks

• Mathematical constraints on N.N with the given hardware constraints

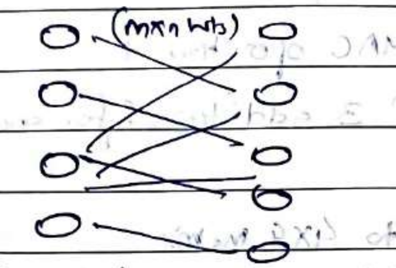
⇓

Seeing these constraints in mathematical form ⇐ { Whatever we discussed before
→ memory hierarchy

Specific to CNN

• Multilayer Perceptron : Vanilla n.n ← fully connected

• MXN weights — huge storage & computation req (fully connected)



⇒ if we don't want to load all MXN
↓
• then we do (eg) 1D-CNN
loading (RxS) wts at a time & doing partial sum (something)
• here we instead of processing inputs in 1D we process in 2D

• we can take multiple wts (filter or kernel) at a time also

• input processed in 2D
∴ wts = RxS at a time

CNN basic

input → filter → output

3x32x32 (↑ no. of channels; ↑ N)

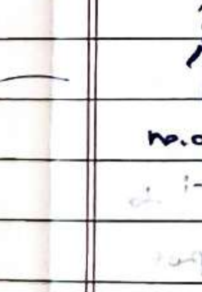
filter: 5x5x3 ← f

Output size: $\left(\left(\frac{N-F}{S}\right)+1\right)n$

- N: input size
- F: filter size
- S: stride (how much sliding we are doing to the filter after a computation)
- n: no. of filters

f brings non-linearity

• what kind of activation function?
• how large is the output?



→ After padding with zeroes, the input size becomes $\left(\frac{N-F+2P}{S}+1\right)$

P → how many lines of zeros added extra on each side

why should we do it?

S=1
F=2
9? 2x2 → 4

NOTE: in this method, if we look closely, we are reducing the size wts but no. of computations, it increases

$\Rightarrow N \times M \times R \times S \times C \times (E \times F)$ — $O(n^7)$ for computation!!

N: batches, (E×F): output

→ how to optimize MACS computation?

- this is known as convolution

- To reduce • 4 MAC operation + 3 addition ? for one output

Not efficient (same no. of multi)

Ideal: • Toeplitz Matrix

↳ convert 3×3 to 4×4 matrix

a	b	c
d	e	f
g	h	i

→

a	b	d	e
b	c	e	f
d	e	g	h
e	f	h	i

← each row now can be easily multiplied

does it solve anything ? → another form of representation?

↳ converted matrix becomes huge size

#. How to reduce multiplication

(reduction in multiplication) a) Fast Fourier Transform

filter * input = output

↓ ↓

apply FFT * apply FFT = apply FFT^{-1} to get output

- Multiplication complexity:

$O(N_o^2 N_f^2) \rightarrow O(N_o^2 \log_2 N_f)$

filter

- what's the problem?

→ FT represents in sinusoids of diff freq & amplitudes of a signal

↓

due to which memory req huge (bigger than RAM)

↳ though computation reduced → energy hungry too!

'Benefits:

(i) Reduced complexity

(ii) larger convolution kernels, better the performance

Improv → Can we have few more addition than multiplication? (Yes)
count on FFT

ex. $(a+bi)(c+di) = (ac+bd)+(bc+ad)i$

4 mult
3 add

$k_1 = c(a+b)$
$k_2 = a(d-c)$
$k_3 = b(c+d)$

Real part $= k_1 - k_3$
Imaginary part $= k_1 + k_2$

3 mult
3 add

2 add → more add but fine!!

→ $O(N^3) \rightarrow O(N^{2.807})$

Strassen's Algorithm — not much red but yes!
- Reduces multiplication than matrix mult
- Numerical stability less
- requires more memory

Winograd - 1D - F(2,3) — targets convolution instead of matrix multiply
output size, filter size
can be extended to 2D

⇒ (do more add & footprint & errors accordingly)

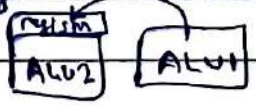
09/09/25

- What is a convolution?

Accelerator Architectures

- Why MACC memory hungry?
  - 4 memory access req. per MAC op
  - MAC not liked in ML mostly because of this
    - → Add nearby memory
    - → Also parallelisation
  - helps in reducing the pb (wt?)

- ASIC — has spatial architecture
  - → give all ALUs each individual controller & register
  - → Nearby ALU can access others register



Can access

Network on Chip

• SIMD / SIMT Architecture in GPU mostly,
→ less communication bet^n ALUs
→ Single Controller (global one)
→ 2 tradeoffs in GPU
(i) memory & blocks } discussed previously
(ii) register & threads }

• Spatial Architecture
→ data flow processing : ALU passes data bet^n each other

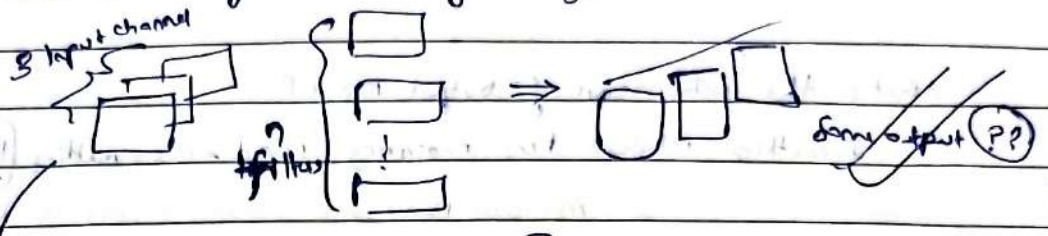
← processing element (PE)

Why do this ?
⇒ Why bad ? : not space efficient, but energy efficient
→ (bring global memory close & remove registers) ← don't store anything there!
→ no local reuse dataflow

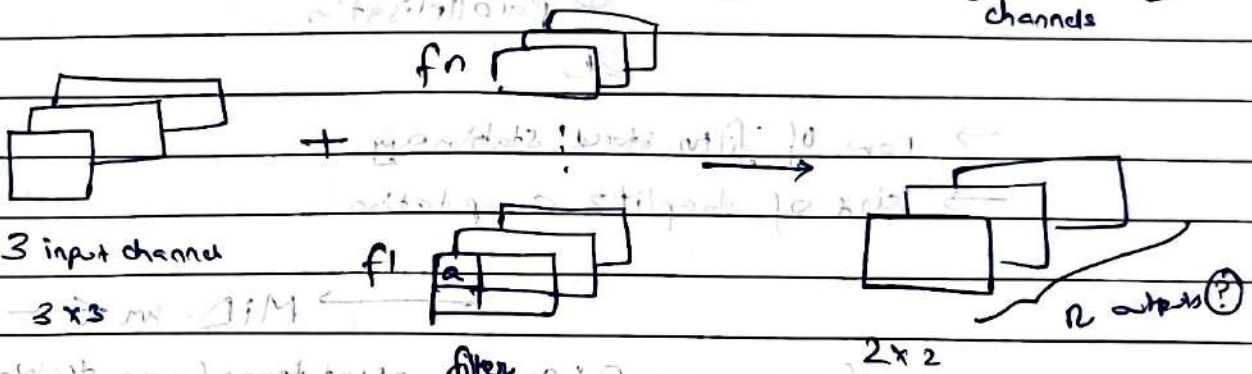
→ Creates a hierarchy of memory !!
• own memory
• many PE memory
- register file
- DRAM
⇐ small

So how to avoid accesses from DRAM ?
fmaps
- kind of how to utilise this small 'own' memory & many PE memory
⇒ generally first filter then activation
but we can reuse : (i) filter + activation both
(ii) only filter
(iii) only activation
⇒ 3 kinds of data stored for a single convolution op :
(i) weights filter
(ii) activation input
(iii) activation output

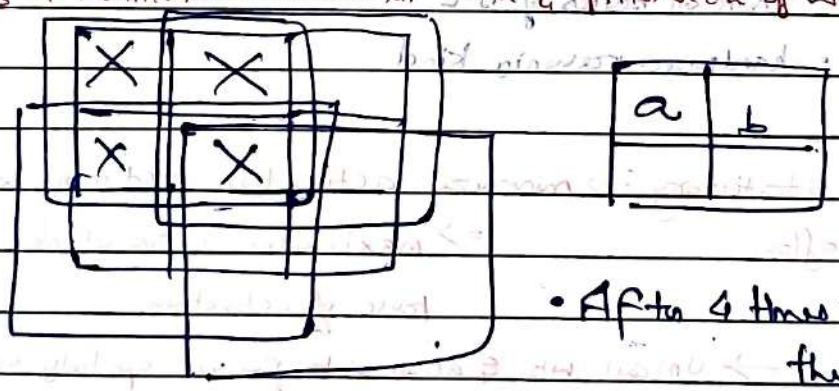
• Why are there multiple filters?
→ Shouldn't filter be constant with no. of channels?
→ how is no. of channel & filter affecting?



inputs
{ depth of filter = no. of channels
no. of filter = no. of output channels } *



• In Weight Stationary Design
→ first compute for small (a) with all the possible matrics in the first input image
• minimise energy of reading weights ⇒ maximise convolutional reuse & filter reuse of wts



• After 4 times → change/update the registrar value to b

• i.e r times we update : r is filter dimension (n x m) part of
• for each of these (r). we need to bring activation

↳ minimize the energy of reading & writing partial sums

• In output stationary design ⇒ maximise local accumulation of maps partial sums
→ partial sum stored in register
↳ not weights / activation

What's the advantage of output row ?
→ omitting related less memory reader's omitting (?)
maximise reuse & accumulation at the RF level for all type of data (weights, activation, partial sums)

• Row Stationary
→ can be done using 1D
↓ parallelisation
2D

→ row of filter stored stationary
→ kind of toeplitz computation

→ MIT-IBM ←

• few numerical Q : CNN • next transformer, training large models
• justification
• how to train NN
• by hand
• optimisation : (——)
• More on basics of DL
• hardware research kind

• input stationary :→ minimize activation read energy consumption
⇒ maximises convolutional & fmap reuse of activation
dataflow
used for sparse CNNs → Unicast wts & accumulate psums spatially across the PE array

MIDSEM revision

(i) NOTE: $y \in \{1, \dots K\}^m = \begin{bmatrix} y(1) \\ \vdots \\ y(m) \end{bmatrix}_{m \times 1}$ where $y(1)$ is $[1, K]$

(ii) ML Basics > L1_V2.pdf > Pg 24 – optimisation
{check back}

(iii) Backpropagation: Forward & Backward Pass

(i) Forward Pass

initialise: $Z_1 = X$

iterate: $Z_{i+1} = \sigma_i(Z_i W_i) \quad i = 1 \dots L$

(ii) Backward Pass

initialise: $G_{L+1} = \nabla_{Z_{L+1}} l(Z_{L+1}, y) = S - I_y$

Iterate: $G_i = \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right) W_i^T, \quad i = L \dots 1$

$$\nabla_{W_i} l(Z_{K+1}, y) = Z_i^T \left(G_{i+1} \circ \sigma'(Z_i W_i)\right)$$

(iv) Practice Questions of Automatic Differentiation (AD)
↳ Repeat the slide

(v) In the initialisation case,

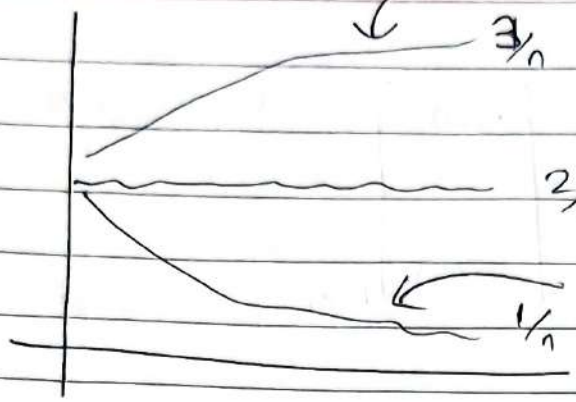
$W_i \sim N(0, \sigma^2 I)$ {randomly initialising}

The choice of variance σ^2 affect 2 quantities:

a) The norm of the forward activations, Z_i

b) The norm of the gradients $\nabla_{W_i} L(h_\theta(X), y)$

exploding gradient problem



↑ signal becomes unstable

vanishing gradient problem

↑ signal dies out

$\sigma^2 = 1/n \;/\; 2/n \;/\; 3/n$

Reasoning for RELUs (why 2/n)

$$Var(y) = Var\left(\sum_{i=1}^{n} w_i x_i\right)$$

as ID

$$= \sum_{i=1}^{n} Var(w_i x_i)$$

$$= n \cdot Var(wx)$$

$$= n\left(E[(wx)^2] - [E(wx)]^2\right)$$

$$= n\left(E[w^2]E[x^2] - (E[w]E[x])^2\right)$$

$\because E[w] = 0$

$$= n\left(E[w^2]E[x^2]\right) \longrightarrow Var(w) = E(w^2) - (E(w))^2 \quad (*)$$

$$= n\, Var(w)\, E(x^2) \longrightarrow E(x^2) = E(\max(0,p)^2)$$

$$= n\, Var(w)\, E(p^2) \qquad = \int_{-\infty}^{\infty} \max(0,p)^2 f(p)\,dp$$

$$Var(y) = \frac{n\, Var(w)\, Var(p)}{2} \leftarrow (*) \qquad = \int_{-\infty}^{0} 0^2 f(p)\,dp + \int_{0}^{\infty} p^2 f(p)\,dp$$

$$= \frac{1}{2} E[p^2] \quad \because E[y] = \int_{-\infty}^{\infty} (\cdot)$$

Final output $var(y) = var(p)$ which is input

$$\Rightarrow \quad 1 = n \frac{Var(w)}{2}$$

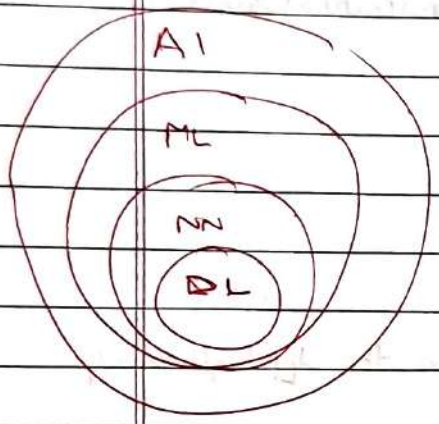
$$\boxed{\frac{2}{n} = Var(w)}$$

(vi) Questions of Normalisation
(i) Layer norm
(ii) Batch norm
(iii) Minibatch dependence
} Which one is efficient & why? any numerical question?

(vii) NOTE:
- depth of filter == depth of input map
- Many filters can be present
- for each filter we get a map ⇒ which determines the depth of the o/p

(viii) ANN/NN
- → Feedforward Network (info. flows in one direction)
  - → Fully connected network (FCN) / Multilayer perceptron (MLP)
  - → CNN (convolutional neural network)
  - → AutoEncoders (AEs)
- → Recurrent Networks (information flows in loops/cycles)
  - → Recurrent neural network (RNN)
    - → variants: LSTM & GRU
    - → Specialized for sequential data
- → Other Advanced Architectures
  - → Transformers — LLM (application)
  - → Generative Adversarial Networks (GANs)
  - → Graph Neural Networks (GNNs)

AI
→ ML
→ Supervised Learning
→ Unsupervised "
→ Reinforcement "
→ Deep " — utilizes ANN
→ ANN
→ RNN
→ Advanced Architectures



(ix) Activation Functions

- Sigmoid — $y = \frac{1}{1+e^{-x}}$
- Hyperbolic Tangent — $y = \frac{e^{x} - e^{-x}}{e^{x} - e^{-x}}$
- Rectified Linear Unit (ReLU)

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

- Leaky ReLU

$$y = \max(\alpha x, x)$$

α = small const (eg. 0.1)

- Exponential LU

$$y = \begin{cases} x & x \geq 0 \\ \alpha(e^{x} - 1) & x < 0 \end{cases}$$

(x) What kind of theory question on CNN?

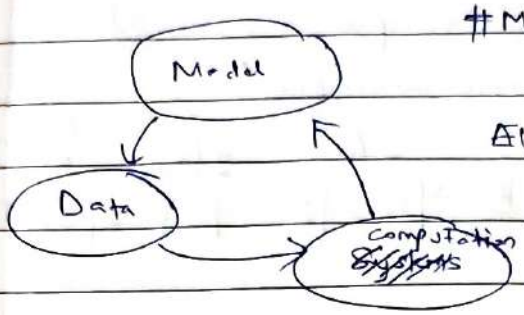
→ Matrix multiplication?

$h(x) = Wx$

classmate
Date 22/09/25
Page

- Training Large Models — req for transformers
  - → 2 things included: Memory Saving + parallel operations
  - # Sticking to GPU {optimisation}

Memory Saving Discussed

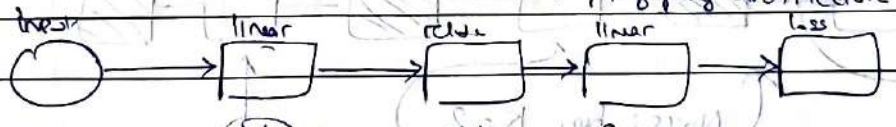


* Elements of machine learning systems

- In computational graph, we store the loss-grad, linear grad etc which {I think for backpropagation}
  - but in inference not req
- What consumes memory? — i) weights {model} + momentum for Adam etc high order optimisers
  - ii) optimiser stats {more in CNN} ⇒ huge memory req
  - iii) intermediate activation values — required during backprop
- What is used during inference? — → weights, activations little but not much
  - → model weights is already optimiser stats {no}
  - handled by converting nn to CNN {small filter size considered at a time}
- How many memory buffers required for inference?
  - nn stacked up in layer
  - only needed for ideal case {escape connection skipped}
  - (might req more buffers)

2 buffers

Ping Pong architecture



M_1 M_2 for this we can use M_1

- Storing h_1, (W) ← already there in model (no explicit storing)
  - hidden layer → Output
  - Model saved in memory

NOTE: To find bet updates, we need to ~~require~~ store the outputs of each layer : $O(N)$ memory req

6 memory buffers?

⇓

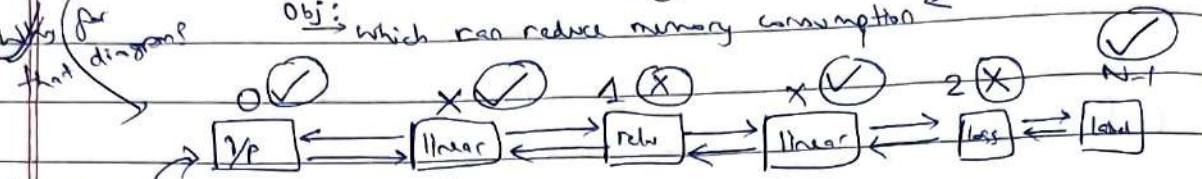
Instead we can do : Checkpointing some

Sublinear Memory Cost Theorem

• do we really need to do 1 forward pass at a time?

Obj: → which can reduce memory consumption

• buffers (for that diagram)



anyways need to be stored

• bit more forward pass to save memory

⇒ Don't store this but we can do forward pass after reaching there & get the values as prev layer info. stored

• reduction in 2 memory buffers i.e $6-2=4$

• Not just 1 method — there can be many Checkpointing techniques

it can be represented in form of mathematical exp

↳ so recn happens!!

↳ using ping pong architecture

$$\text{Memory cost} = O\left(\frac{N}{K}\right) + O(K)$$

Checkpoint cost

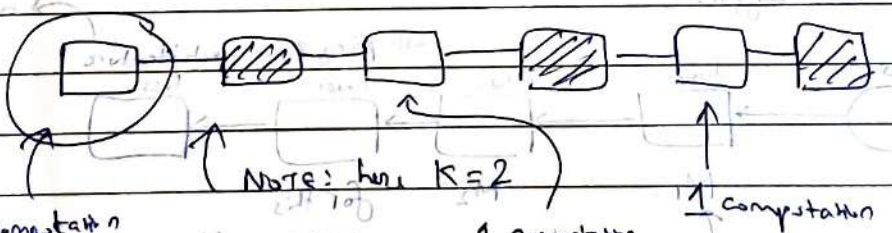
↑ recomputation cost!!

⇒ how many we are sliding through

(Note: $K != N$; $K = \sqrt{N}$ generally (for recomputing how much memory we are using))

NOTE: fn should be reversible

In transformer, we represent layers in form of blocks



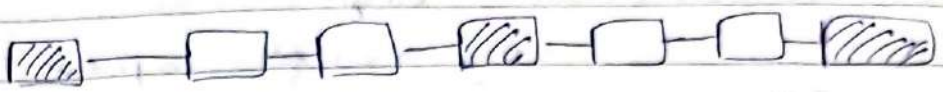
(No computation req)

NOTE: here, $K=2$

1 computation

1 computation

#No. of recomputation can't be more than k but we are sliding by $K \Rightarrow$ $K-1$ of them aren't computed when doing final pass again!! ↳ So we need all of them in memory!!



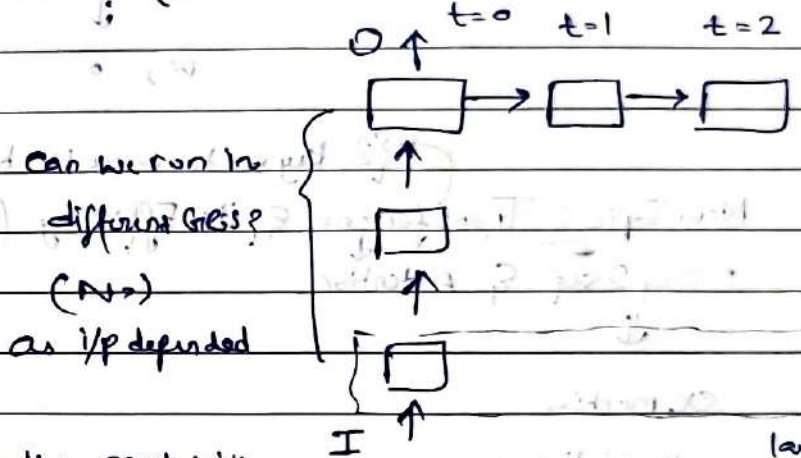
is it possible in nn?
⇒ more memory req

What is K here? (3)

06/10/25

Topic: Training Large Models

How to exploit parallelism? (in GPU)



(i) Breaking the computation graph with
- Send/Receive commands into 2 halves

How to train across GPUs

M1 { • In 1st cycle we cannot but somehow what we can do is when first pass happening → we can kind of do backward pass in parallel : Computation Graph

M2 → Data parallel training : Split the data among different GPUs & then sum it up (the partials)

→ How to do sum up?
→ How to replaces the replica of models?

→ All reduce abstraction
⇓
- Collect all g_i
- Sum up all $g_i = G$
- pass G to all replicas so they can update by itself

What's the fall back?
- Each GPU will run nicely!
- all models will remain same but

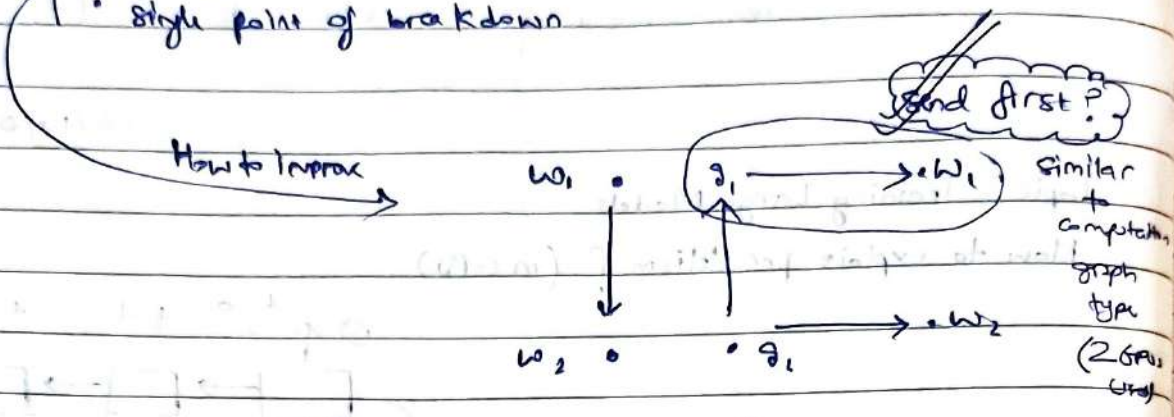
Alt.? ← ⇒ do global wt updates & then pass : Parameter server abstraction no!

Parameter Server Model — benefit ?

• one server failure no problem : as in one replica failure

fallback ?

• privacy problem — as g_i shared
• single point of breakdown



{2 key words come in here}

New Topic : Transformers & its Efficiency (Efficient Transformers)

• Seq 2 seq & Attention

⇓

ex. machine translation

⇒ encoder decoder framework

↓ uses source representation } How to achieve translation ?

• gets input : generate target seq
• builds a sort of representation
• uses conditional language model

↓

embedding

• process source + previous prediction

↓

to generate probability distribution for diff next o/p

• The simplest model : 2 RNN

→ using 1 vector to represent whole input : that's given to decoder

→ may not be perfect representation of input

↳ if seq is long → suffer from vanishing gradient problem

#Attention

- Importance of vector : like out of whole – some given proportion

no single vector only passed

- for current computation, we have attention computation which kind of determine off of all input vectors which is important !!
  or state

(a_k values 'bet^n 0 to 1 & $\sum a_k = 1$) : Applying softmax

To make everything differential

(like kind of give wts to all states)

How to compute score?

→ Dot product (s_i, h_k) for the decoder state h_k !!

–

07/10/25

Transformers : {has ~~Encoder~~ attention in all encoder, decoder & ~~or~~ more}

↑ Why?

Req 1 : parameters should not increase with input length

↳ {Sharing wts} !!

Req 2 : (- - - - - -) Should handle text of different length (i.e independent of text length)

- Query, Key, Value : 3 important terms in transformers

Req 3 : Connection bet^n words & strength of these connections depends on words themselves

Req for design of transformer : like basic terms

→ using only 3 wts : W_q, W_k, W_v } – independent of input {i.e shared} length

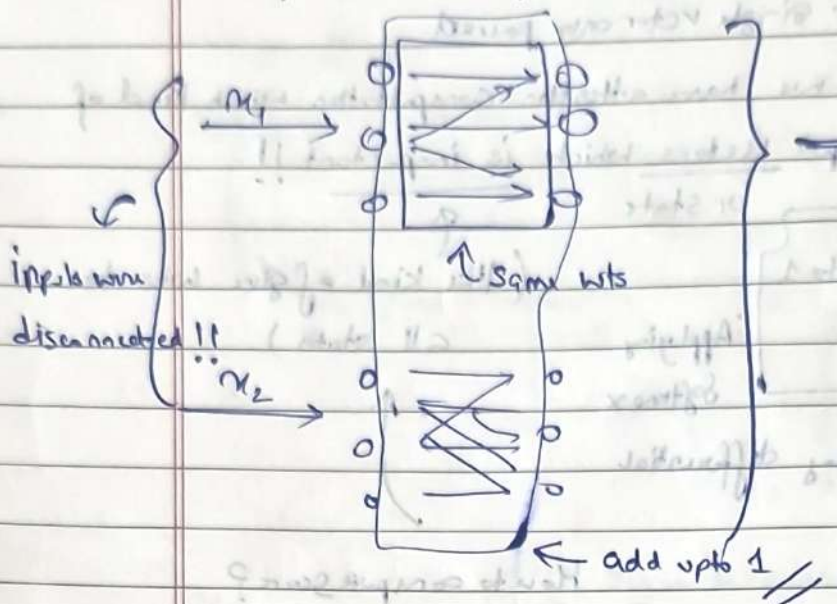
$W_q \times$ [ ] = Query (??)

i/p

this is done through dot product → take attention then

- Attention as routing

a) inputs → x_1, x_2, x_3



b) Sparsity in wt matrix

c) hyper network – attention wts computed in a separate clss

d) How to compute attention wts – compute similarity betn queries & keys

↓

Takes up $O(n^2)$ space

↓

Can explode so scale it up!!

→ next class: optimisation of space

Addon ~~#Encoder Discussion~~ e) Position Encoding (P.E.) – Why, when, how, what

↳ uses sinusoids

(f) Multi Head Attention: sort of during training we split the data & train ⇒ output is combined but while taking a avg with 1 data to reduce multiplications

↳ Why do we need this?

↳ to process this input we require a ~~Feed Forward~~ Neural Networks

Resi ~~dual~~ connection: helps in training large data

↳ Sort of add the previous data!!

Complete Transformer (Pg 90) – covered!!

classmate
Date
Page

↳ like what kind of customisations

- How to use transformers for NLP — we use Tokenisation
  → converting text to numerical value (91 - 111)

Next Lecture : Efficient Transformers — How to improve the architecture

13/10/25

Efficient Transformers

- Self Attention Wt Computation → takes $O(n^2)$
  - How to optimize : transform algorithm

→ MHA : Multi Head Attention : most inefficient : in Transformer steps

- Do we need to pay attention to all words ? — (No)

⇒ fix the viewpoint ↳ learnt ?

⇓

To do this : apply Gaussian scale : left centred

M1 • attention are hardcoded

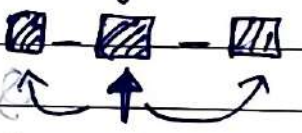
• What's the problem ?

M2 • local attention : keep a window : K previous tokens

longformer — (i) Sliding window : 3 on right & 3 on left {handcrafted attention}

↳ receptive view field is limited

↳ similar to 1D conv ps

(ii) do dilation : i.e instead of 3 consecutive do it like 

- leads to inc in field of view
- but performance is issue
- No. of computation skips !!

(iii) Global + Sliding window

→ Not all global attentions but some of them + Sliding window //

Linformer

↳ reduce Matrix size : low rank ~~of~~ matrix

d — original embedding dimension

Note: Keys & values of size: N x d

⇓

reduce to K x d //

↳ What plb with it? ⇒ the sequence length is actually small for inference on this models !!

if dom on long ⇐ (so not real)

- it fails

What to do? (previously only this)

- Can we care about data & architecture both?

⇒ based on diff token pos ☐ change the sliding window 'k'

⇒ # Routing attention : tokens learn / lead to attention of tokens that belong to same cluster //

how to form it?

⇒ k means clustering : same centroids for both keys & queries (??)

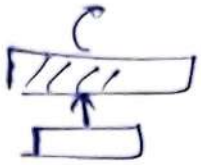
Limitation for this design:

(i) Slow — build on k

Q Where do we need large context window?

⇒ Question Answering

(Slide - 45/50)



classmate
Date 14/10/25
Page

#MLOps

Model Deployment

- not backprop exactly - but automatic differentiation
- Model trained — but how to use in different applications?

⇒ as in different environment

data preprocessing
post "

- AI guided compilation — can it help the model optimisation for diff devices?

Learned Models

↓ → simpler than AD one as it's inference

Represented in computational graphs & wts

↓

learnt by inference engines — like coreML, TFLite

↑ By Apple ↑ by Google

- How to represent it?

→ inference done using pipeline architecture
2 buffer

They use libraries for it
→ but need a lot of customisation & burden

Machine Learning Compilation

ML model

↓ IR

High level optimisation

↓

Tensor level

↓

Direct Code generation (general; not device specific)

↑ So for this one, we try to use the ML to search for all possible config

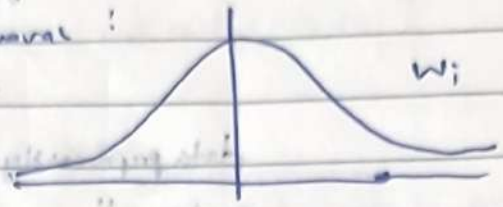
then also can be done using ML (??=)

Here the other variant is hardware optimisation but here we code level optimisation using ML here

(#ML Perf)

Pruning → due to less memory storage

- Instead of keeping all n.n layers, remove some of them
- how to figure out which one to remove?
  - → as in weight removal :



S1 blind removal ← {either remove layers or remove activations}

doesn't work — {i.e removing neurons with W=0}

NOTE: only deal with ~~addition~~ multiplication

~~Q is activations & Wts same?~~

~~Q is pruning done after training?~~

optimisation req!!

- Hashing (why?) repeated multiplication of same type
  - ↳ to do multiplication as it's expensive but addition not!!
  - 7.5 × 2.7 → 11 → 34 = ( )

→ extreme pruning + fine tuning helps!!

S2

S3 Iterative pruning & finetuning

- W → W_p
  - ↳ No. non-zero values //

15/10/25

- Why Prune? → reduction in parameters, MACs, energy
- How should we prune?

Pruning Strategy : like from which layer, how much

Pruning Granularity

→ ~~fine~~ GPU friendly & ~~coarse~~ not

M1 → weight pruning — fit some to zero : irregular part so hard to optimize in GPU

(fine grained)

M2 → Coarse grained — fit some rows to zero

classmate
Date
Page

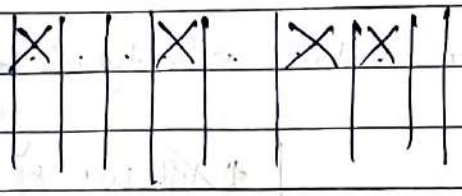
- What can we prune?
  - input & op channel
  - kernel + (.....)

~~What is kernel?~~
~~channel?~~
~~is pruning & pooling same?~~

Fine grained Pruning : (Benefit) flexible pruning indices
larger compression ratio
speed on custom hardware (not on GPU)

Pattern based Pruning : {(N:M) i.e. in every M blocks, N of them will be pruned}

ex. 2:4



← Instead of storing all cells
simply store the non-zero cells
with corresponding indices
⇓
Compressed matrix

Channel based Pruning : (reduce channel numbers in every layer

↓
latency is preserved either uniformly or non-uniformly

(??)

synapses → we know there is a method to do it, but how do we select the cells?

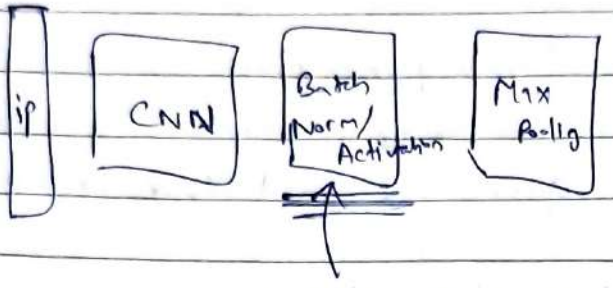
- How do we determine which cells to prune?

M1 → remove less important ones
↳ how to we determine?

M2 → heuristic pruning criteria : L1, L2 norm & consider importance for now!!
- Can the algorithm learn it & do?
- Scaling factor associated with filters

classmate
Date
Page

- In batch normalisation → we kind of learn scaling factor
  → that is only restricted for the filter on



Second Order based Pruning

↓

minimising the loss after pruning is done: using taylor's series

(sort of wts)

~~Diff~~ btn synapses & neurons ? or connection

Percentage of Zero Based Pruning

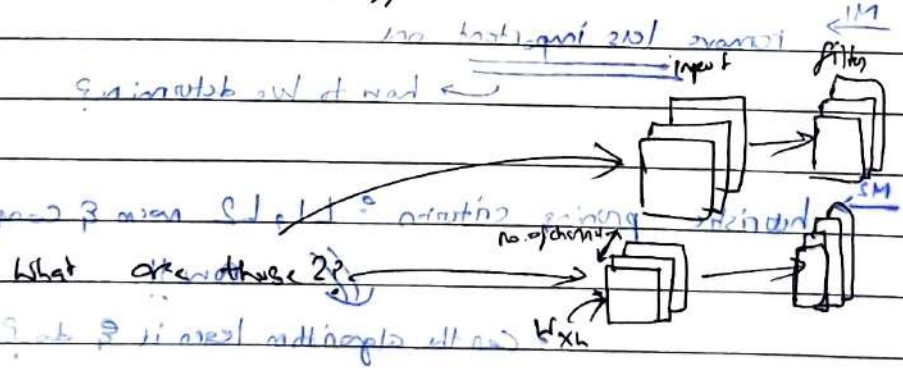
↓

(here neuron some context involved !!)

- Heuristic method: does calculation (forward)
- Neural numerical method: can we use back prop to prune the weights?
  → something: local loss & MSE

Regression based pruning

→ some reduction in loss for the ~~the~~ non pruned pruned layers



classmate
Date 25/10/25
Page

Principle way to Prune
↑ easy for specific hardware

- Pruning Ratio for each layer

M1 - uniform prune : latency high

M2 - AMC : latency low
↑ how fast we determine the result

- uses RL
- as in come up with different sets (L1, L2, L3, L4, L5)
- use RL to find the best set

M2-a if we do grid search ⇒ sensitivity per layer
(Sensitivity analysis)

(?) Q is RL & grid search same? (NO)

→ Drawback: Analysing Accuracy & pruning for only a specific layer at a time & trying to make inference
↑
Threshold T may not be achieved

M2-b So we go for RL method : overall goal (known)

(?) per layer we need to determine

AMC: AutoML for Model Compression

- What is error, here? ⇒

NOTE: AutoML gives smaller model inspite of giving same accuracy!

1X1 CNN - hard to prune as only 1 wt

3X3 CNN - representation

M2-c : NetAdapt : doesn't use RL (non RL based)

- try few combinations pruning
- fine tune ⇒ accuracy high
- Select it
- is the accuracy determined for that layer or the whole model?

How to improve performance of pruned model

→ on pruning → sparsity high

⇒ reduce learning rate : $\frac{1}{100}$ or $\frac{1}{10}$ of original

⇒

Regularisation

↳ penalise non zero params

Encourage smaller ||

- When L1 & L2 better ?

⇓

apply both of them in case of non training

How model is pruned

↳ how to optimize for hardware

(3 methods)

(i) sparse weight } use need for multiplication

(ii) sparse activation }

(iii) Quantization

Storing sparse matrices

↳ CSR

↳ store vals & rows & cols (for non zero)

⇓

reduces no. of computations // (Slide 86)

EIE : hardware accelerator

classmate

Date
Page

28/10/25

Quantization

a) 32 bit —(Cannot be done directly)→ 4 bit {Point 1}

so how to?

b) 32 floating point no

0 → Zero / Subnormal

1 – 254 – normal

255 — infinity / NAN

(?) {c) Why learning rate when kept high, wts turned to be NAN?
→ gradients

⇓

how can we exceed a 32 bit FP range?

d) importance of w, b, X, Δ, l : differ } ⇒ exponent & fraction size differs

exponent — controls range

fraction — precision

in neural networks wb ⟶ exponent less req as range is

-1 to 1
-2 to 2 } ⇒ precision req

for input data → more range req

e) Input data continuous ⟶ transformed to discrete set
↓
through quantization

K-Means Quantization ensures!! ← • reduce storage space
• reduce computation energy } ~~Linear Quantization~~ Rounding off congruently ⇓ But affects accuracy

→ ~~if~~ Point1 holds so codebooks & old no be get only!!

→ if centroids go very low after finetuning ⟶ do lower the reassignment!

- Pruning first ⟶ Quantization next

- NOTE: Huffman Coding
  ↓
  → One kind of bits representation
  → motivation: all weights do not come with same frequency

- How do we know linear quantized multiplication is efficient?

$$Y = \underbrace{WX}_{FP}$$

{as all params in complex process already quantized} → a no multiplication in quantized process as all elements already quantized & no more FP but integers!!

- NOTE: $Z_W = 0$ {emphirical observation} {conventionally}

31/10/25

- Quantization {linear} < tensor
  ↓
(So) Different scaling Types
  a) have scaling factor$^{(i)}$ per {...} ← for whole channel: 1 (not good)
     (i) per channel
     (ii) per group
       ↳ type: VS Quant → γ: per tensor; S_q: per vector

$$S = \frac{r_{max}}{2^{n-1} - 1}$$

  b) Quantization error: {measures how good quantization is}
     → it should be less

  c) E8M0 – Exponential 8 Mantissa 0
     MX4 → S1M2: Signed 1 bit, Mantissa 2 bit

(S1) WHAT SHOULD BE THE SCALING FACTOR?

→ x_{min} & r_{max} : unknown

So using statistical observation on training data, we figure it out for S_{min} & S_{max}

- need'nt be extreme vals
- How to determine? — Minimise mn sq error
  Minimise KL divergence
- then follow with clipping : it helps

(S2) Rounding Schemes

(i) Stochastic Rounding & Ada Round

↳ W_x : Activation output

↳ here we try to learn on what range, we should round ~~Quantize~~ by learning from activation outputs

- Mobile Net V 1/2 — efficient models (like no redundancy)
  ↳ Post Quant — Training doesn't work well
  ⇒ & hardware specific we need to determine!!
  ⇒ During training we learn to quantize
  → we have both Wt & quantized wts during training
  → we use quantized wts in testing

- Why do we care about binary & ternary quantization?
  ↑
  - because we want to avoid Multiplications!! ↑ we use pop count operations!!

{last topic}

04/11/25

#Offline KD → memory hungry trained big model: costly

- Logit - output before softmax applied
- Knowledge Distillation : using bigger model to train tiny models → to softmax output similar

(to make smaller network as good as large model)

Student Teacher model

Temperature : helps in controlling softmax output

Matching of Student & Teacher network : logits (already trained)

(i) weights should match when Student Network projected to same dimensions

(ii) distillation loss : some pb !!

↑ match logits : output dimensions same, to match channels

How to find similar projections? → 1×1 convolution

MLP for matching & learning

{dimensions not same} on projection → another loss introduced as we are learning to match the wts

(iii) ~~for~~ match intermediate features / activation output

(iv) Matching gradient wrt input : $\frac{\partial L}{\partial x}$

helps in generating attention maps of DNN

(v) Matching sparsity patterns : (Os)

(vi) Matching relational information

→ 2 ways : relation within the model (s_i, s_j)

" across samples (s_i, s_{i+1})

Self
~~Online~~ KD
(i) iterative classification & distillation!!
(iii) Pb ? – req some step
may not work in all situations : Not better accuracy

Online KD
(i) mutual learning
(ii) no pretrained Teacher
← will not size of tiny same as big?
→ then where is tiny coming into picture?

Combined Self + Online
• Early exit networks : previous stage layers improved after the future layers trained ⇒ future layers can be dumped
• Limitations : (i)

Previously Distillation for classification
↓
how to do for object detection (done) !!
(3 things that need to be taken care of)
(i) object classification – foreground & background
(ii) regression – bounding boxes
(iii) final output
how to match this output for Student & teacher?
↓
$L_b(R_s, R_t, y)$ interpretation!! (Pg 44)
• If teacher bad ⇒ Student not follow teacher
• If teacher doing good ⇒ upper bound for Student is teacher's

(Pg 55) – KD for LLMs (transformers)
o can we do KD of attentions?
(Pg 61) – network augmentation ? – for small network (not good)
↳ like dropout, cut etc ↳ so they do not Aug !!

NetAug : During training increase the network!!

↓

only works for small network (not big ones)!!

→ENDSEM←

a) How to modify Transformer architecture to handle images instead of text?

Step 1: Input Representation

- Transformer for text

Text → Tokens → Embedding layer

for Images - Patch Based Approach as in Vision Transformer

Image → Non overlapping patches → Flatten patch to vector = patch vector

→ Project it to Transformer's embedding dimension

↓

consider them as input tokens

- Pixel Based Approach

Pixel as tokens & RGB values as features → pass it to embedding layer

- Hybrid Approach
  Use CNN to extract initial feature maps
  ↓
  flatten
  ↓
  Embed these features as tokens

Step 2: Attention Mechanism
- Transformer for text
  Self attention computes (i) pair wise interactions between tokens
  (ii) capturing dependencies regardless of their position in the sequence
  (iii) modulated by positional encoding

- for images
  - Global Self Attention as in Vision Transformer
  (i) Apply self attention on seq of patch embeddings
  (ii) Each patch attends to all other patches capturing global spatial relationships

  - Local or Sparse Attention as in Swin Transformer
  (i) Restrict attention to nearby patches or pixels to model local spatial dependencies similar to convolutions
  (ii) Shifted windows allow gradual cross region mixing

Step 3: Positional Encoding

- in case of Texts

(i) It is added to tokens embeddings to encode the order of tokens in a 1D sequence

- in case of Image

(i) 1D Positional Encoding
assign a positional embedding to each patch based on its index in the flattened sequence

(ii) 2D positional encoding
encode (x,y) of each patch or pixel in the 2D grid using sinusoidal functions or learned embeddings

(iii) Relative Positional encoding
encode relative distances between patches (eg Manhatten distance) in the attention mechanism

b) Computing Self Attention output for first token
~~what~~ about second or third ?
(Similar manner)

Given:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad V = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 1 & 1 \end{bmatrix}$$

first token query = $q_1 = [1, 0]$

1. Dot products $q_1 \cdot k_j$ $\left[(q_i \cdot K^T)\right]$

Score vector = $q_1 \cdot k_j$

$$= \left[(1,0) \cdot [1,1] \;,\; [1,0] \cdot [0,1] \;,\; [1,0] \cdot [1,0] \right]$$

$$= [1, 0, 1]$$

2. Scale by $\sqrt{d_k} = 8$ (64)

$$[1, 0, 1] = [0.125, 0, 0.125]$$

3. Softmax

$e^{0.125}$, $e^0 = 1$, $e^{0.125}$

sum = $e^{0.125} + e^0 + e^{0.125} = S$

$$w_1 = \frac{e^{0.125}}{S}$$

$$w_2 = \frac{e^{0}}{S}$$

$$w_3 = \frac{e^{0.125}}{S}$$

4. Weighted sum of V

$V_1 = [2, 0]$ $V_2 = [0, 2]$ $V_3 = [1, 1]$

Output: $w_1 V_1 + w_2 V_2 + w_3 V_3$

x-comp: $2w_1 + w_2 \cdot 0 + w_3 = 2w_1 + w_3$

y-" : $2w_2 + w_3$.

⇒ Attention output: $[x, y]$

has keys/values

has queries

Cross attention

c) Suppose in one encoder-decoder attention head, all query vectors for a given output position in the decoder are zero, while encoder keys and values are arbitrary.

a) What is the resulting attention weight vector?

(i) Attention Scores : $q \cdot k_i = 0$

for every key as query is a zero vector

(ii) Scaled Scores are all 0

(iii) after softmax,

$$\text{weights} = \left[\frac{1}{N}, \frac{1}{N}, \cdots, \frac{1}{N}\right]$$

$\Rightarrow$ Uniform distribution over all encoder tokens

b) What is the Cross attention output for that position?

Cross attention output :

$$= \sum_{i=1}^{N} w_i v_i$$

Since every weight $w_i = \frac{1}{N}$

$$\text{output} = \frac{1}{N} \sum_{i=1}^{N} v_i$$

$\Rightarrow$ The output is the mean of all encoder value vectors

d) # Light Weight Transformer

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

(i) first token: $q_1 = [1, 0]$

$\sqrt{d_k} = \sqrt{8}$

(ii) Score : $q_1 \cdot k_j$

With k_1 : $(1,0) \cdot (1,1) = 1$

" k_2 : $(0,1) \cdot (1,0) = 0$

" k_3 : $(1,0) \cdot (1,0) = 1$

" k_4 : $(1,0) \cdot (0,0) = 0$

$$[1, 0, 1, 0]$$

(iii) Scale it : $\left[\frac{1}{\sqrt{8}}, 0, \frac{1}{\sqrt{8}}, 0 \right]$

(iv) apply softmax:

$q_1 = e^{1/\sqrt{8}}$

$q_2 = e^0 = 1$

$q_3 = e^{1/\sqrt{8}} = q_1$

$q_4 = e^0 = 1$

$S = q_1 + q_2 + q_3 + q_4$

(v) weights => $w_1 = q_1/S$, $w_2 = q_2/S$, $w_3 = q_3/S$,

$w_4 = q_4/S$

(vi) Weighted sum of V

x-component: $\frac{2a_1}{5} + \frac{a_2}{5} = \frac{3a_1}{5}$

y component: $\frac{2a_2}{5} + \frac{a_2}{5}$

e) Difference b/w ~~what~~ Attention with more ~~smaller~~ heads, smaller d_k V/s ~~less~~ fewer heads. larger d_k

on factors:

(i) expressivity (ability to capture distinct relationships)

(ii) memory usage

(iii) training stability

Case 1:

expressivity

a) More heads (→ 16), smaller d_k (→ 32)

Pros: → high diversity of attention patterns

→ each head can focus on a different relationship (syntax, negation, long range etc)

→ better for tasks needing many linguistic cues (translation, QA)

Cons: → smaller d_k ⇒ each head has less capacity

→ per head representation is weaker

(# Breadth over depth) (many simple specialists)

→ 4 → 128

b) Fewer heads, larger d_k

Pros: → each head is more expressive ∴ can model complex patterns
→ Stronger per head feature extraction

Cons: → fewer heads → less diversity
→ might miss some specialized patterns

(# Depth over breadth) few powerful specialists

Case 2:

memory usage:

(a) More heads
→ more attention score matrices → slightly higher memory
→ must store $16 \times n \times n$ attention matrices
→ more intermediate tensors per head
→ more softmax + projection buffers

(b) Less heads
→ fewer attention score matrices → lower memory
→ only $4 \times n \times n$ score matrices
→ fewer intermediate activations

NOTE: Difference is small because total dimension and n^2 dominate memory.

Case 3: Training Stability

a) More heads

Pros: → more heads : more diverse gradient signals
→ each head learns different patterns → reduces overfitting
→ diversity smooths the optimisation landscape
→ can improve Stability

Cons: → very small d_k can cause unstable attention scores
• Scores become very noisy
• Softmax may fluctuate noisy
→ risk of head collapse (weak heads) due to low capacity

Key Idea: Stability from diversity, instability from tiny d_k

b) Fewer Heads

Pros: → Larger d_k → lower variance in $\frac{QK^T}{\sqrt{d_k}}$

→ Softmax becomes smoother → more stable gradients
→ more expressive heads handle deeper patterns clearly

Cons: → Less diversity: fewer gradient directions
→ May reduce robustness since fewer heads share the learning load

Key Idea: Stability from stronger heads, but less diversity

f) Layer Normalization → what?
→ applied after self attention and feed forward networks in each transformer layer to stabilize training ← why?

Why Layer Norm preferred over Batch Norm in Transformers?
→ due to its robustness in handling variable sequence lengths
→ ability to stabilize training in deep networks
→ computationally efficient for parallel processing

When Batch Norm can be used?
→ in specific scenarios such as an NLP Task plus
→ with fixed length sequences, larger batch sizes (trained on a homogeneous dataset with consistent statistical properties across samples) and shallow models

Case 1: Sequence Length Variability
Layer Norm
(i) normalizes within a single token's features (across embedding dimension)
(ii) independent of sequence length
(iii) padding doesn't affect normalization
(iv) Stable for NLP because each token gets normalized separately

Idea: Token wise normalisation; length invariant

Batch Norm

(i) normalizes across the batch (across tokens and examples)

(ii) requires consistent batch structure → assumes homogenous input size

(iii) variable length sequences cause issues: padding skews mean/variance

(iv) statistics become unreliable for small batches or lots of padding

(v) can cause unstable training for NLP

Idea: sensitive to padding, & sequence length

Case 2: Training Dynamics

Layer Norm

(i) normalizes each token independently → stable for variable length NLP seq

(ii) keeps activations and gradients consistent across layers → crucial for deep transformers

(iii) works well with residual connections, preventing gradient explosions/vanishing

Batch Norm

(i) depends on batch statistics, which are noisy for small or variable batches in NLP

(ii) padding affects mean/variance → leads to unstable normalization

(iii) inference uses running averages → can mismatch training statistics

Case 3: Computational Efficiency

(i) Layer Norm

(i) computes mean/variance per token over feature dimension → cost $O(d)$

(ii) for a sequence of n tokens → cost $O(n \cdot d)$ (independent of batch size)

(iii) no reliance on batch size → simpler, faster and no cross-device sync needed

(iv) very efficient for large scale NLP with varying batch sizes

Batch Norm

(i) computes statistics across the batch → cost $(N \cdot d)$ per layer

(ii) less efficient for small batches and memory heavy for large batches

(iii) requires synchronization across devices in distributed training → extra overhead

(iv) computationally heavier and less suited for NLP transformers

g)