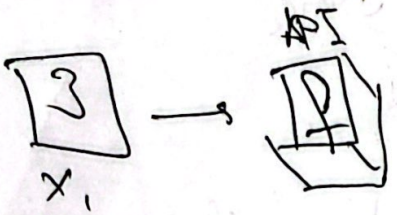
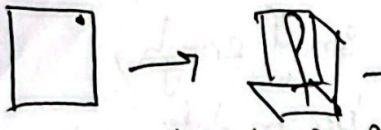
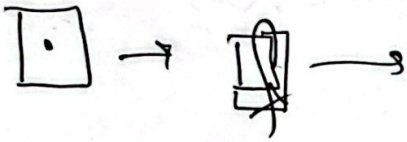
lec 2 (1)





↳ handwritten digit 2 with black pixels at LT corner.



$$\left[\nabla_{x_3} p(x_3) \right] \quad \because \nabla \text{: del means lots of values.}$$

Not ↳ $\frac{dp(x_3)}{dx_3}$ because we calculate it for all different pixels

$$\text{loss} = f(w, x) = MSE(y, \hat{y})$$

y → actual, $\hat{y}$ → predicted

$\frac{\partial \text{loss}}{\partial x_{1,1}}$, $\frac{\partial \text{loss}}{\partial x_{1,2}}$, same for every pixel.

↳ change in loss at pixel (1,1)

↳ change in loss w.r.t Δ in value of pixel (1,1)

$\Rightarrow \nabla_x \text{loss}$

↳ del or nabla

Hower Δ - delta mean small change

$\frac{\Delta \text{loss}}{\Delta x_{(i,j)}}$; if we make small change small enough it will become derivative, So,

bcoz in nets

$\frac{\nabla \text{loss}}{\nabla x}$ is actually $\nabla_x \text{loss}$ for too many values

So, instead of change weights of in a model, we are change inputs to the model. So, we are trying to make X_3 more like digit. So, we can change pixels accordy to this gradient $\nabla_{x_3} p(x_3)$. So we are gonna take every pixel and we are going to subtract its gradient a little bit.

$$X_3 = X_3 - C \times \left[\nabla_p (X_3) \right]$$

Diffusion

- CFG.
- Textual inversion
- KLDiv
  - JSDiv
  -

U-Net - predict noise

Encoder : → latent

Decoder : latent → img.

} Computationally expensive

$H(x) = -\sum p(x) \log p(x)$

$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$

$\mathbb{E}[f(x)] = \sum p(x) f(x)$

weighted sum of $f(x)$

sample

2)

As I change input pixels, how does it change probability that this is a digit. what tells which pixels to make darker which pixels to make lighter.

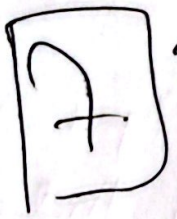
So, when you do this by changing each pixel one at a time to calculate derivatio, this is called a finite difference method of calculating derivatives and its slow.. because we have to call the func 784 time for a 28×28 img. but its slow so we won't have to use finite difference.

we can do this by

→ f.backward()

→ X.grad

So we don't particularly need the thing that calculates probabilities. we only need the thing that tells us what pixels to change to calculate probabilities.

but where is  ?

7 is a NN that tells us which pixels to change to make an image look more like handwritten digits. So, we could create some training data & use that to get info we want.

we can pass sounds like

- digit
- doesn't look like digit
- more corrupted digit

the check which is more like a handwritten digit which exists less but that would be complicated. what if we could predict how much noise we added?

4)

So, we take input & pass through NN.
We try to predict what noise was ($\hat{n}$) added
and actual noise is n
& loss fn would be

$$\text{loss} = \sqrt{\underline{(\hat{n} - n)^2}} = MSE(n, \hat{n})$$

$\#$

→ So input noisy digit whose n we are controlling

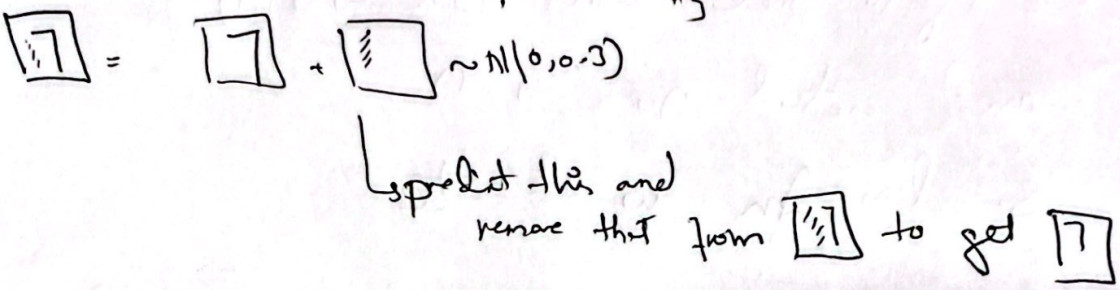
→ we predict noise for it.

→ calculate $MSE(n, \hat{n})$

↳ we have n because we are manually adding it

So, if we can predict the noise, we get exactly what we want which is

$$x_3 = x_3 - \nabla p(x_3) \cdot c \cdot \nabla p(x_3)$$



So, if we just input pure noise into the model (NN), it's gonna split up info about which part of it, it thinks is noise and it's going to leave behind the bits that look most like digits.

this NN is called U-Net.

So, for unet

- noisy img (input)
- output : noise;

5)

Encoder:

- Down sampling
- Up sampling

512×512×3 ⟹ (Down sampling) 256 ... ⇒ 64×64×24

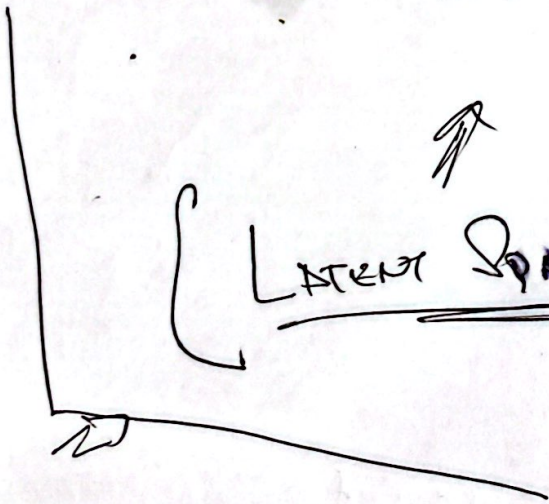
↓ Resnet

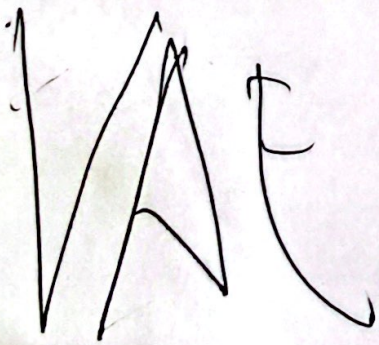
64×64×4 <= ... <= 512×512×3

(Up sampling)

- Initial decoder output would be random
- NN will learn

loss = MSE (decoder output, encoder input)



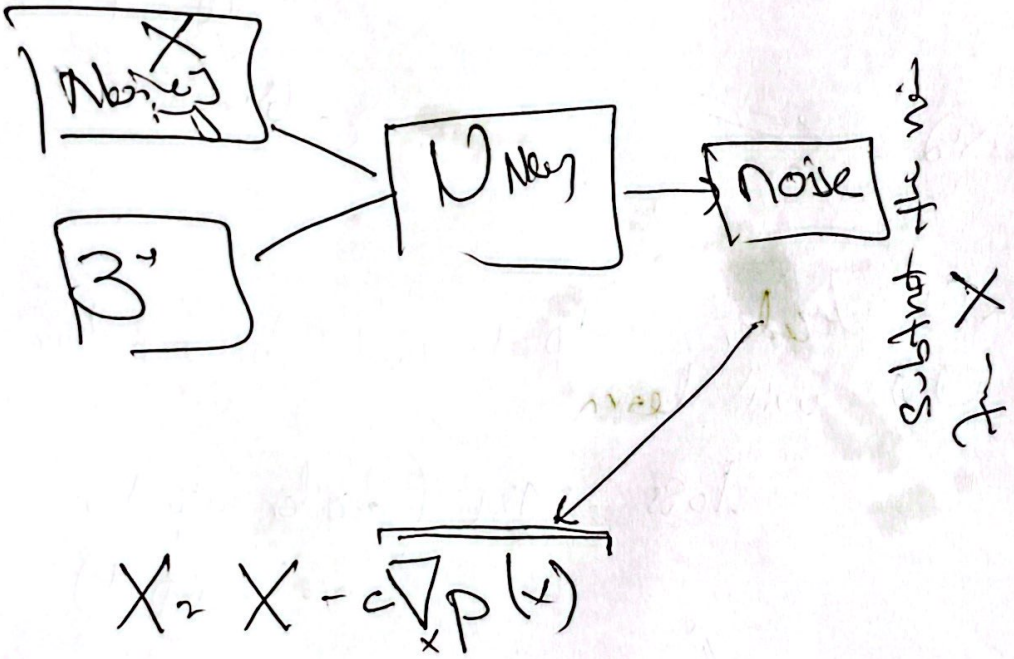


Dimensionality Reduction

Naive version

but how to tell "produce me an image of teddy riding a horse"?

↳ In addition of passing in noisy input, we can pass in one hot encoding of what digit it is [HMM]



$$X = X - c\nabla_x p(x)$$

This is called Guidance

but for "a cute teddy" we can not use one hot encoding.

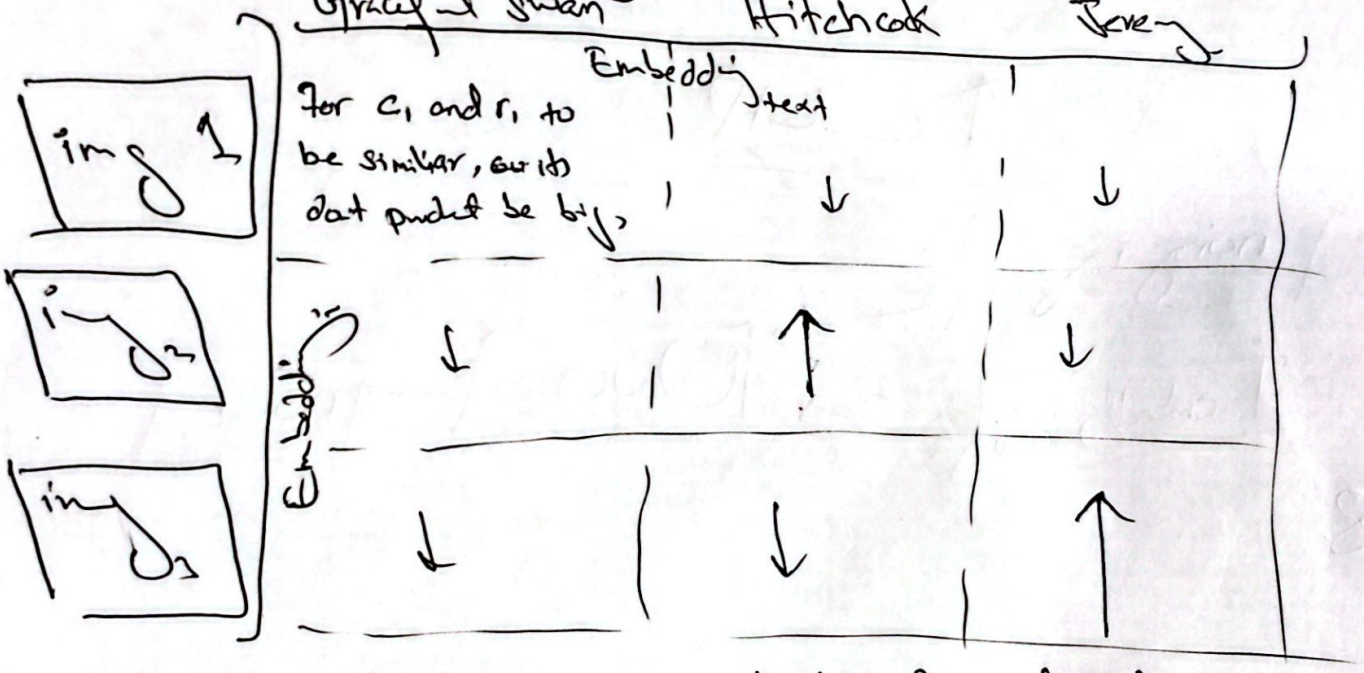
7)

Embedding "a CLIP story"

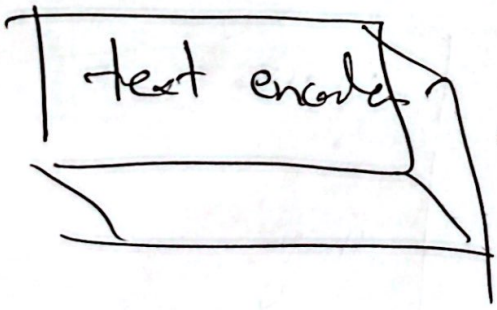
↳ text create two models

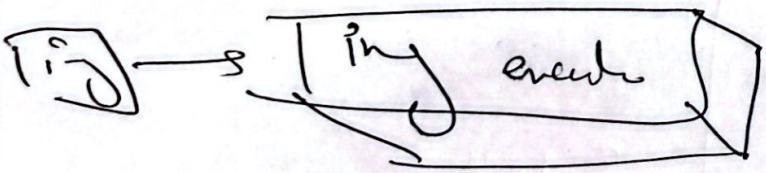
↳ Text encoder

↳ img encoder



"a graceful swan"



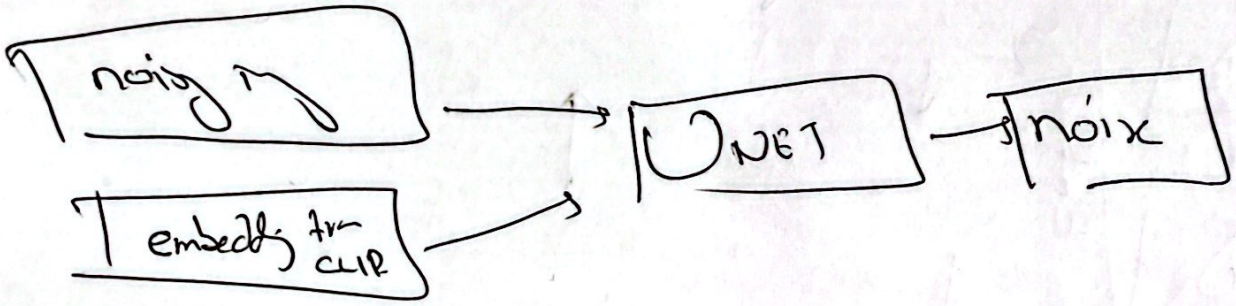


Dot prod-t = $\sum_i$ (elemt wise multiplic^n)

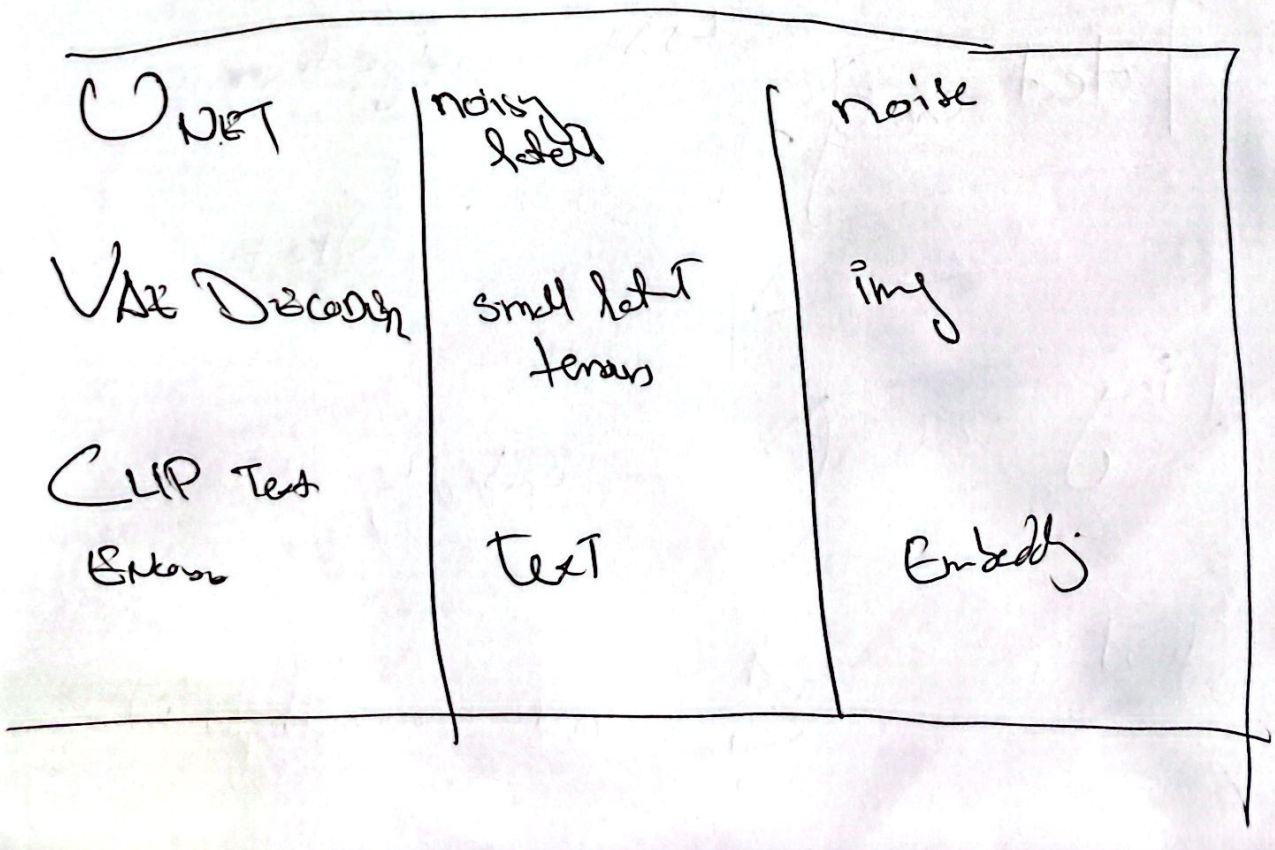
Σ (digital entities) − Σ (non-digital entities)

= "CONTRADICTIVE log"

↑ "CLIP"



&,



Q)

$$\text{latents} = \text{latents} - c \cdot \left[\nabla_{x_j} P(x_j) \right]$$

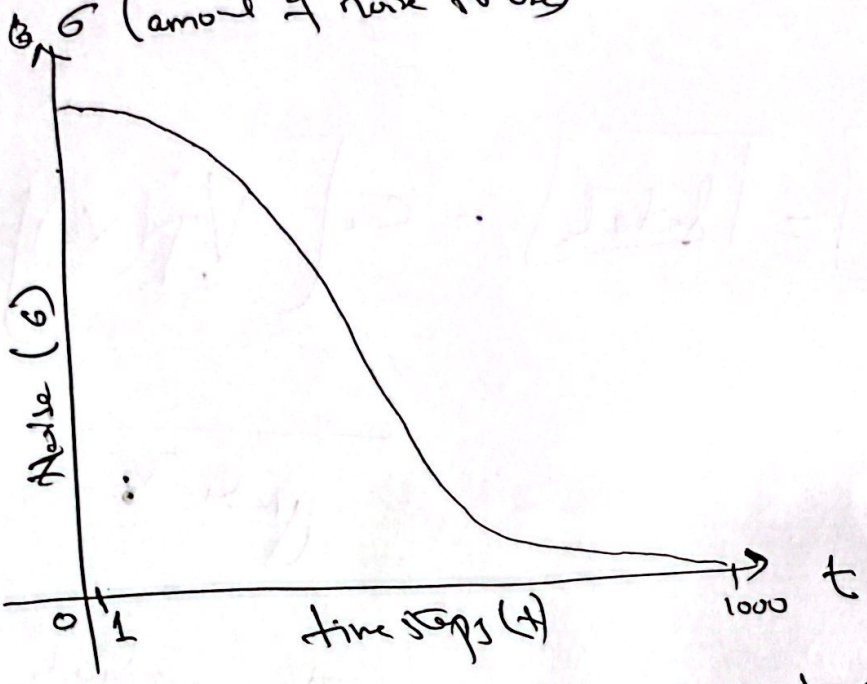
gradients

↳ These gradients are often called score fn.

- ID (CNIC)
- Degrees
- Pics

Time Steps

↳ We use varying amount of noises.



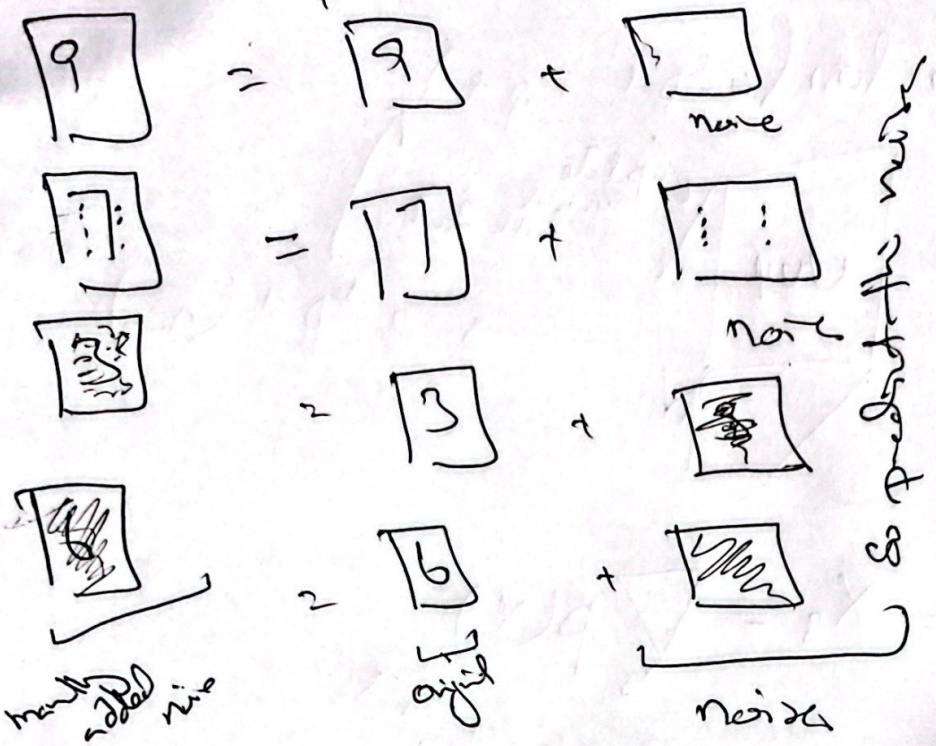
Pick a random # from 0 to 1000 and that's how will add random noise. 1 mean max noise, 1000 mean minimum, mean

noise σ can also be denoted as (β), more commonly.

11

Each time you create a mini-batch to pass into the model, you randomly pick an image, you ~~not~~ randomly pick amount of noise or timestamp (t) and then you use that amount of noise to each one. Then you pass that mini batch to model to train it. That trains weight in model so it can learn to predict noise

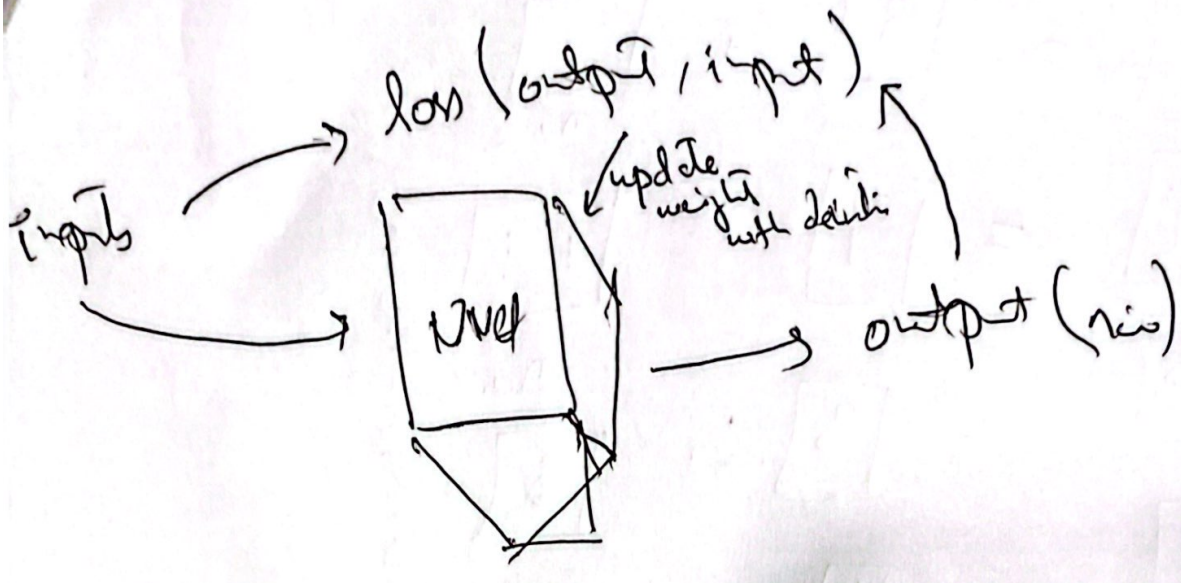
3. Let's predict how much noise



So how amount of noise tells how much of a digit it is.

Something with no noise is a very much like a digit and something with more noise is less like digit.

So, amount of noise tells how much of like a digit it is.



$$X_3 = X_3 - \nabla_{X_3} p(X_3)$$

noise / output

$$9 = 8 + \square \rightarrow N(0,0)$$

$$17 = 7 + 1 \rightarrow N(0,0,1)$$

$$\text{[noisy]} = 3 + \text{[noise]}$$

inputs to model

$\rightarrow N(0,1)$

So, why don't we as the output, use variance to predict amount of noise or maybe, even, actual noise.

b)

So, when you come to iteration.
i.e Generating image from pure noise,
Mean now model is starting at
S_2 (max noise) and you want it
to learn to remove noise.

Caveat ~~noise = x~~

We can not do it all in single step
because, we don't have noisy images
in dataset.

$$noise_2 = noise - c \nabla p(noise)$$

noise

c is hyperparameter

This do look like "Optimizers".

⇒ Momentum: If you change the same parameter by similar amount by multiple times a multiple steps, Maybe you should increase the amount by which you change them.

→ momentum
→ adam
↳ what if variance changes.

] optimizer

→ Yes we can use these tricks.

→ It all come from differential Calc.
which is more about taking these little steps and trying to figure out how to take bigger steps.

→ Differential Eq always tend to take 't' as input. So state DNN models take (x_i, d, t)

- x_i - pixels
- caption (d)
- t : timestep.

't' because model would remember better if you know what it is.

8)

Q1

Stable Diff (Maths):

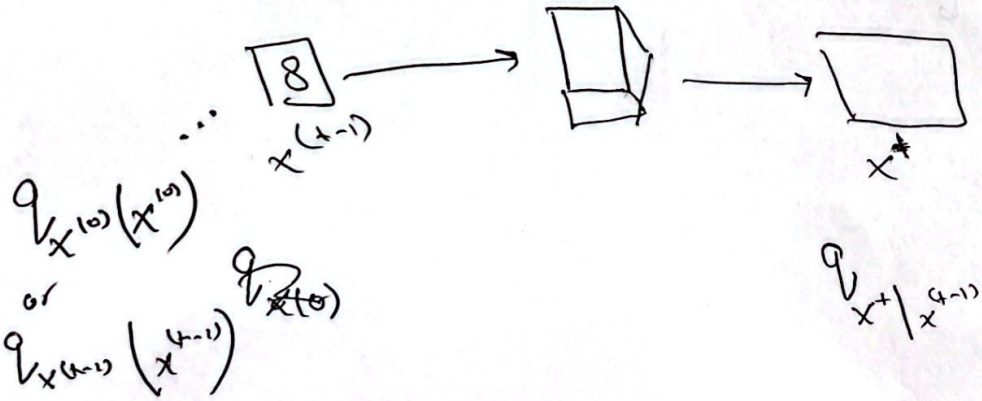
$q_{x^{(0)}}(x^{(0)})$ - data distribution

- x - input variable.
- $x^{(0)}$ - instance 0 or sequences of xs.
- q - prob density function: outputs probability.

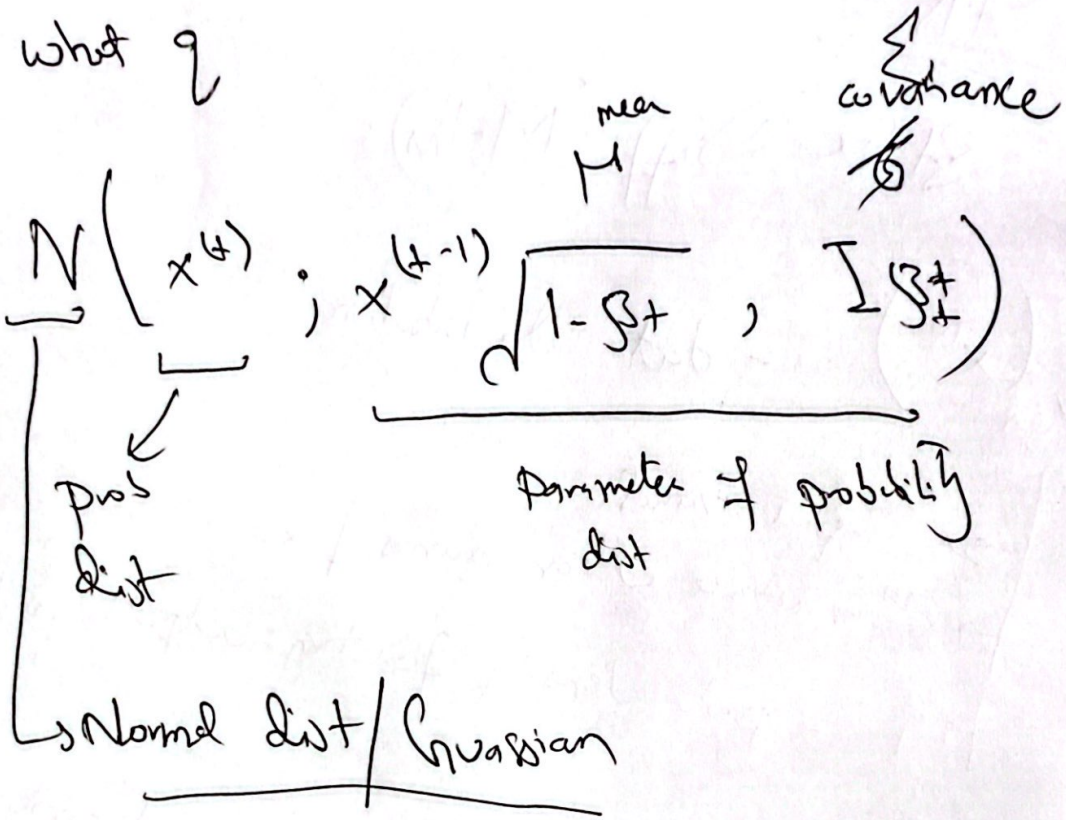
$q(x^{(t)} | x^{(t-1)})$ - conditional probability.

more formal would be,

$q_{x^{(t)}|x^{(t-1)}}(x^{(t)} | x^{(t-1)})$



So, what ?



↳ this tails
↳ mean you only need to describe their behavior in small region of space.

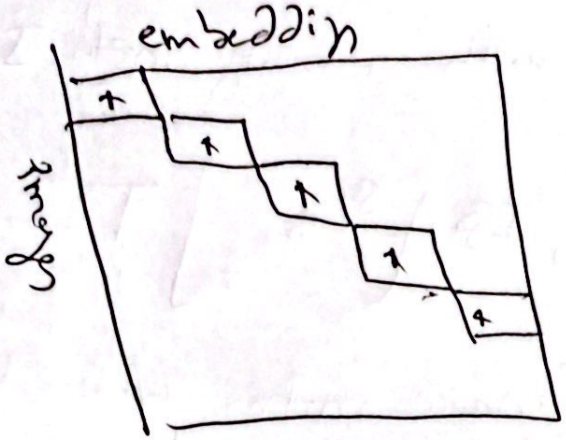
σ - for style variable

Σ - for covariance (multiple variables)

σ^2 - variance ; σ - std deviation

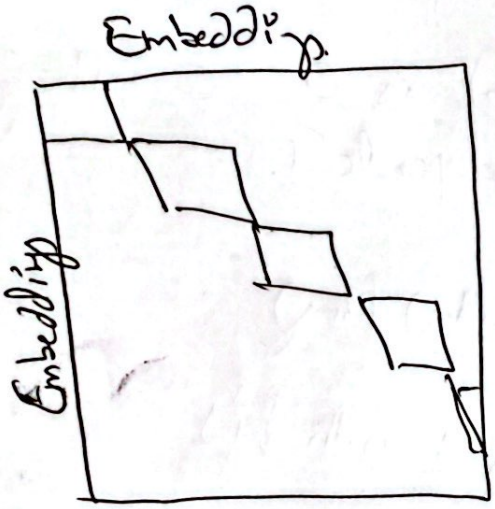
In CLIP

~~(12)~~ ~~12~~ (13)



Assume for embedding a both side

if we subtract mean from diagonal entries



then diagonals are $(X - E(x))^2$

↓ that is variance for each vector.

There dist $q_{(i)}(i)$ is of each pixel diffn.

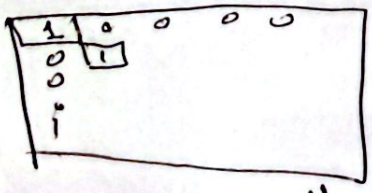
So we are assuming $C_i r_i = 1$, so all other places ~~$C_i r_i$ is that so~~ across that pixels

i.e $C_i r_1$ and $C_i r_i$ are 0.

var is spread from mean

$(x - E(x))^2$

for all 1s at diagonal only var = 0



So that mean all 0 result that pixels are independent.

This the assumption we are making,
&, if we start with identity matrix I.

$$I = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}, \quad I\beta = \begin{bmatrix} \beta_1 & & \\ & \beta_2 & \\ & & \beta_3 \end{bmatrix}$$

So, covariance b/w pixels is 0
but variance b/w pixels (itself) is 1.

so $I\beta$ is a covariance matrix where for each individual pixel β_i is a variance of that pixel and covariance (relation b/w pixels) is 0 & they're expected to be independent.

$\beta_1 = 1$ for (embedding)·(img)

β_1 is variance of pixel 1 with itself

$(X - \underbrace{E(X)}_{\text{avg}})^2$ - would be Variance.

↳ for each pixel there would be different distribution

14)

mean : $x^{(t-1)}\sqrt{1-\beta_t}$

Covar = $I\beta_t$

Lets see how it behaves at edges (extreme cases)

$\rightarrow \beta_t = 0$:

So, for $t = 1$

mean $(t) = x^{(t-1)}\sqrt{1-\beta_t}$

mean $(1) = x^{(0)}\sqrt{1-\beta_1}$

mean $(t=0) = x^{(0)}\sqrt{1-0}$

mean $= x^{(0)}\sqrt{1-\beta_1}$

$\rightarrow x^{(1)}$ example ka pixel

mean $= x^{(t-1)}\sqrt{1-\beta_t}$

$= x^{(0)}\sqrt{1-0}$

mean $= x^{(0)}$

$=$

$\text{mean}(t) = x^{(t-1)}\sqrt{1-\beta_t}$

$\text{mean}(1) = x^{(t-1)}\sqrt{1-\beta_1}$

$= x^{(0)}\sqrt{1-\beta_1}$

for $\beta_1 = 0 \therefore \beta_t = 0$

$\therefore x^{(t-1)}$ is previous.

there will be other items for $x^{(0)}$

$\boxed{\text{mean}(t=1) = x^{(0)}}$ — so we have mean of our previous img.

var = $I\beta_t$

var(1) = $I(0)$

$\boxed{\text{var} = 0}$

So, we have a normal dist with mean of previous img and variance 0 which mean we have same image & var = 0 mean no noise.

At last mean same image

when $\beta_t = 1$

$= x^{(0)} \sqrt{1-1}$

mean $\mu_t = 0$

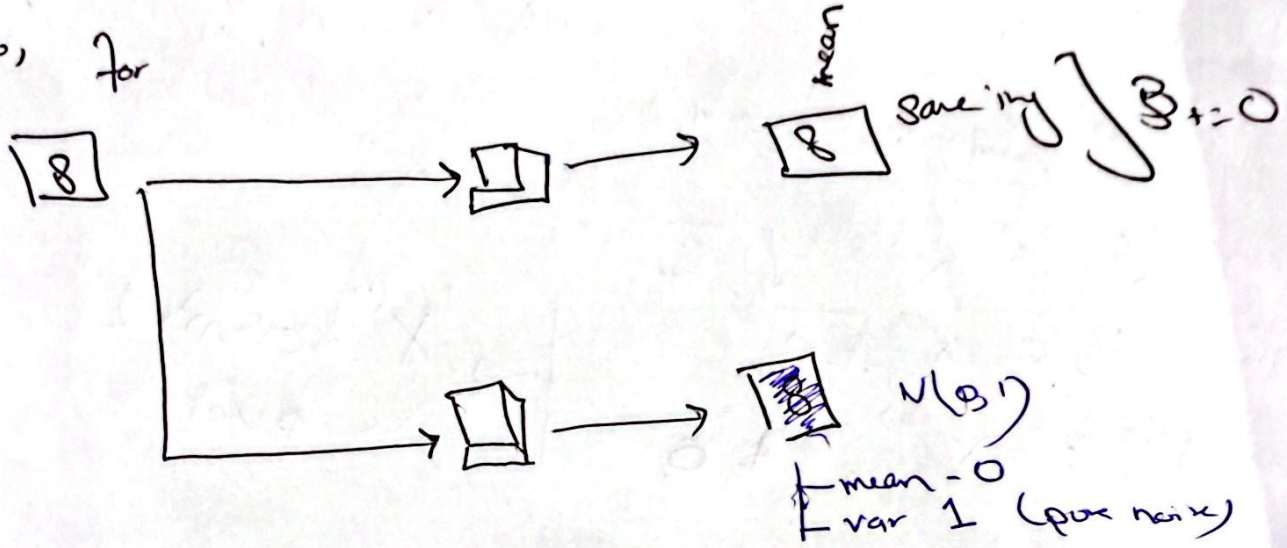
var $= I\beta_t$

var $= I$

↳ I - identity matrix

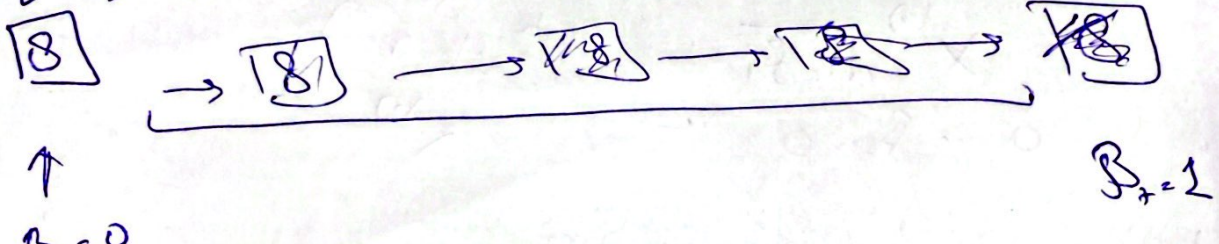
mean variance of 1.

So, for



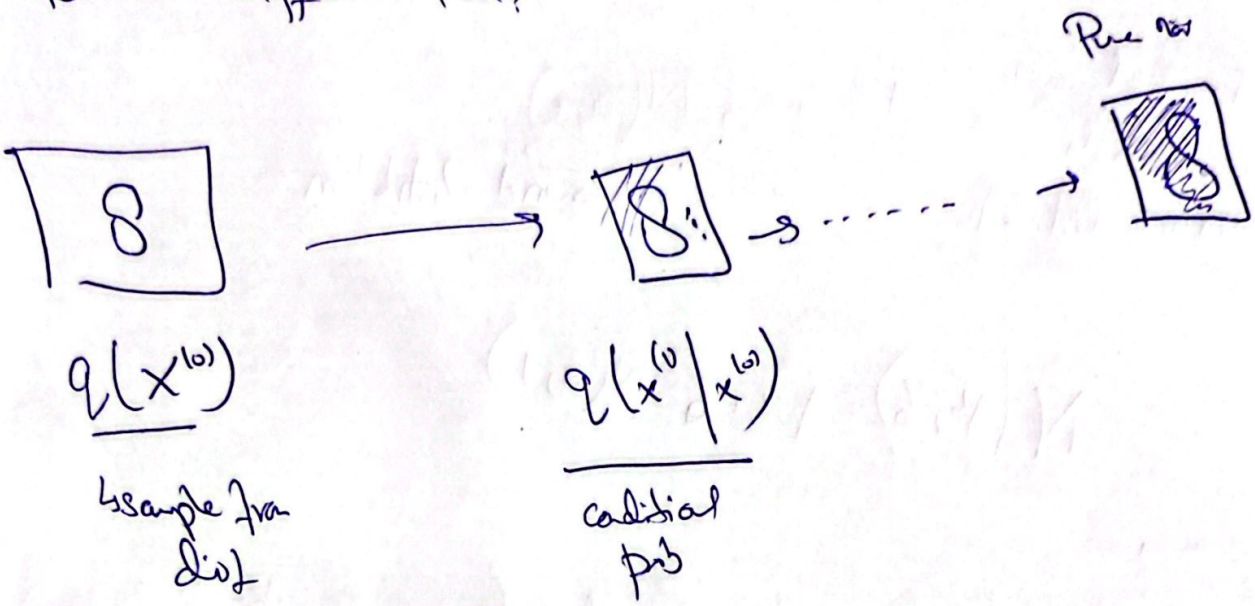
So, b/w these two cases there would be some mixtures (b/w pure noise & img)

$q(x^{(0)})$ - sample from distribution



this is called Forward Diffusion process.

Forward Diffusion Process



$q(x^{(0)})$

↳ sample from dist

$q(x^{(1)} | x^{(0)})$

conditional prob

this is called

Markov ~~chai~~ Process

↳ slightly depends on initial state

↳ Markov Process with Gaussian transition

↳ noise

$q(x^t | x^{t-1})$

↳ transition is the fn.

→ Sampling from distribution

↳ trying to find some random data samples that maximizes that likelihood (or has high likelihood)

$$\prod P(x_i | \theta)$$

To modify normal distribution to any value

To get any particular noise/distribution $= \mu + \sigma \underbrace{\mathcal{N}(0,1)}_{\text{standard distribution}}$

$$\mathcal{N}(\mu, \sigma) = \mu + \sigma \mathcal{N}(0,1)$$

$\hookrightarrow$

Process of going backward (same but different params)

$$p\left(x^{(t-1)} \,\middle|\, x^{(t)}\right) = \mathcal{N}\left(x^{(t-1)}; \text{mean}=?, \text{variance}=?\right)$$

previous | current

↳,

q - forward process	P - backward/reverse process.

P is what we're figuring out,

~~q is input~~

To figure out μ & σ of $p(x)$,

we'll use MLE

16)

Find: μ & σ of $P(\sigma)$ using MLE.

Likelihood is not optimized

But we can do this exactly b/c will need to do some integrals that'll (high dimensional (continuous)) So, you can't check this.

There are thousand of steps & there would be many possible values, in the reverse proj process, so it will be tough to evaluate it for over all these steps for all possible values.

So, log likelihood:

↳ log is for scaling.
- ↳ monotone fn
- ↳ products → sum
- ↳ Normal dist have exp functions that disappear with log.

We'll still can not optimize log likelihood directly because it is not differentiable w.r.t parameter of models, so we can not use gradient descent

⇒ Reason for ELBO is used with log Likelihood in stable diffusion: (Evidence lower Bound)

↳ Two main purpose
  ↳ Regularization
  ↳ Tractability.

1- ELBO in regularization: without ELBO, maximizing the log likelihood alone can lead to overfitting.

So, ELBO introduces a regularization term that penalizes complex models, encouraging them to learn simpler and more generalize representation.

This is achieved by adding the KL div b/w the prior and posterior dist to the objective function.

15000 AED.

2- Tractability.

↳ Directly optimizing the log likelihood can be computationally expensive, ~~especially~~ especially for large models and complex datasets.

ELBO provides a tractable way to approximate log likelihood by introducing a latent variable (the noise variable in stable diffusion and performing variational inference. This allows the model to learn a lower bound on the log likelihood

11)

Imagine you have a huge pile of dirty dishes to wash. Washing each dish individually would take a long time & would be inefficient. Instead, you might use a dish washer to clean them all at once. This is similar how ELBO works in stable diffusion.

Directly optimizing the log likelihood is like washing each dish by hand. It is accurate but slow and computationally expensive. ELBO is like using a dish washer. It introduces a shortcut by using a latent variable (think of it as a special container for the dirty dishes). This container holds all the info about the img in a compressed and manageable ~~form~~ way. Then, ELBO uses a technique called variational inference to estimate the log likelihood based on this container. This is much faster and easier than calculating log likelihood for each individual img. True log likelihood is the ideal result, I want to achieve. ELBO might not give you the exact result but it provides a 'lower bound' which means it guarantees that the final output would be at least as good as [illegible] In most cases lower bound is good enough

So, instead of directly calculating the log likelihood for every possible img, ELBO introduces a noise variable that represents the image in a simplified way. This allows the model to learn about the rules of good imgs, without having to generate & analyze every single possible img.

Think of it learning how to draw a portrait. Instead of trying to draw every single detail of a face at once, you might start by ~~sketching~~ sketching a basic outline and then add in more details.

ELBO is like that first outline, allowing the model to learn basic structure of img before focusing on fine details

10)

9B (continue)

we also learn β_t

for backward process, we are using same fn but parameter are different.

$$= N\left(x^{(t-1)} ; \text{mean}?, \text{var}.?\right)$$

but we can not ~~use~~ to compute log likelihood directly to find these parameter, we need to use ELBO (Evidence lower bound).

ELBO can be calculated easily so, we can use it as loss function to predict mean and variance of this reverse process.

⇒ important to clarify: You can recover whole distribution but you can not a single image, convert it to pure noise and recover it. So, it is an distribution level.

So, approach ELBO is like:

↳ we have forward process (intermediate distn) going all the way to pure noise. With ELBO we are trying to match the our distribution that we are trying to optimize to those distributions that we see in forward process.

KL Div is to compare two distributions

So, we are trying to minimize the difference b/w forward process distn and reverse process distn.

DDPM (2020), (maths like Jensen)

'Denoising Diffusion Probabilistic Models'

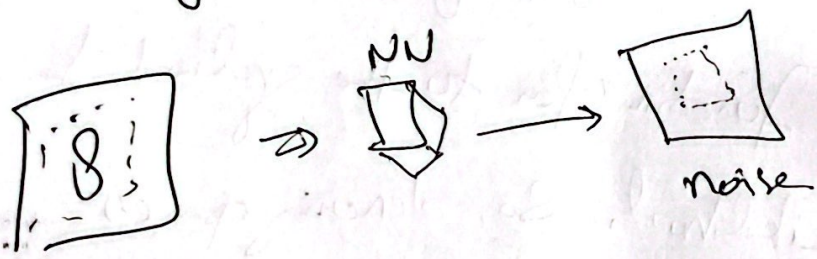
↳ Let's assume that var(in reverse process) is just a constant so, we do not learn it.

And we also assume that step size from earlier (β_t) - the variance of the noise we added at each step is also constant. So, we are just predicting the mean. there are set conventions.

18) then the loss turns out to be, that you predict the noise.

loss ⇒ noise

So, You need to train NN that takes in images & it tells you what I this my is noise.

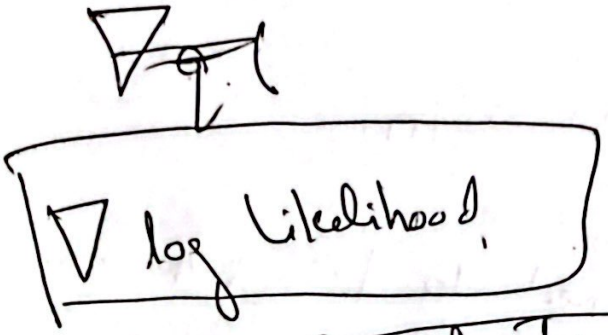


⇒ The idea of adding & removing noise from from distribution is that adding noise to the images we have, that takes us away from
↳(max likelihood)

our real images to noisy images (lowers low likelihood)
So, we want to learn how to get back to high likelihood images.

This Denoising process is equivalent to learning score function. Score function is gradient of log of the likelihood

Score fn is gradient of log likelihood.



~~Score function particularly refers to gradient of log likelihood~~

Score function refer to the gradient of log likelihood. So, denoising process allows us to learn the score function. So, when we are doing the noise prediction, we know we have a probabilistic framework using the likelihood and it comes all down to predicting noise. Its what DDPM proposed & its equivalent to calculating the score fn

---

With DDPM - you just predict NOISE

20)

(Distillation)

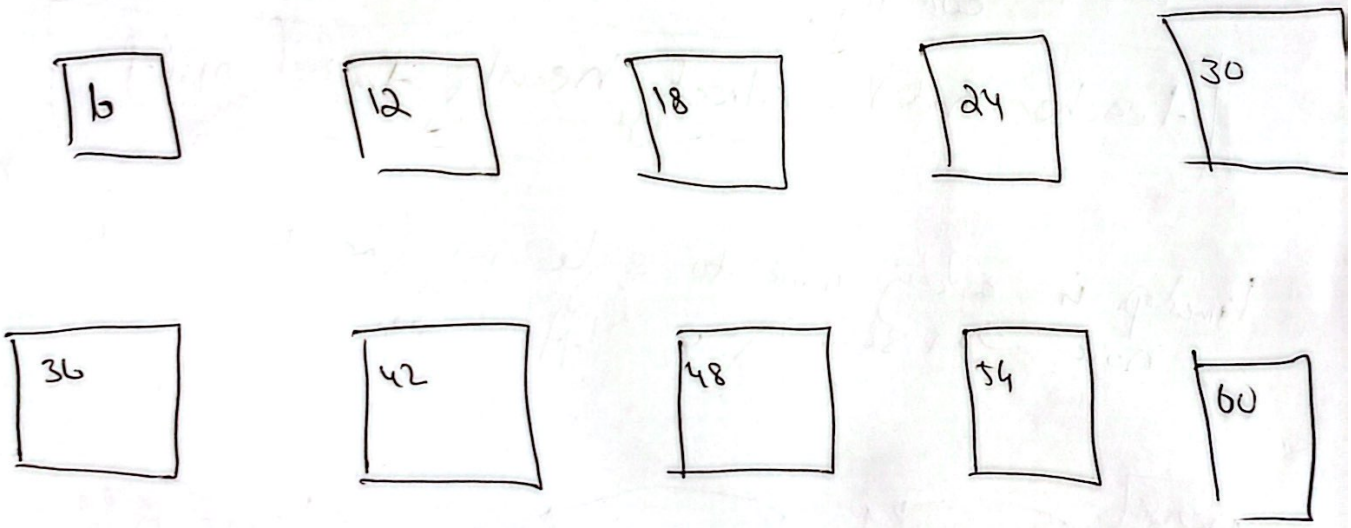
Lecture #10

↳ reducing steps from 60 → 4.

Distillation:

↳ You take a teacher network that knows how to do something but is slow and big and then teacher is used by a student network which learns to do same thing but faster or ~~and~~ with less memory; Here we want our to be faster

60 steps:



It is taking almost 18 steps to go from [42] to [60]. It seems like something that can be done in single step.

So, think of a model that takes input img at 42 and outputs img at 60 exactly all intermediate steps

42 → UNet → 60 (predicted)

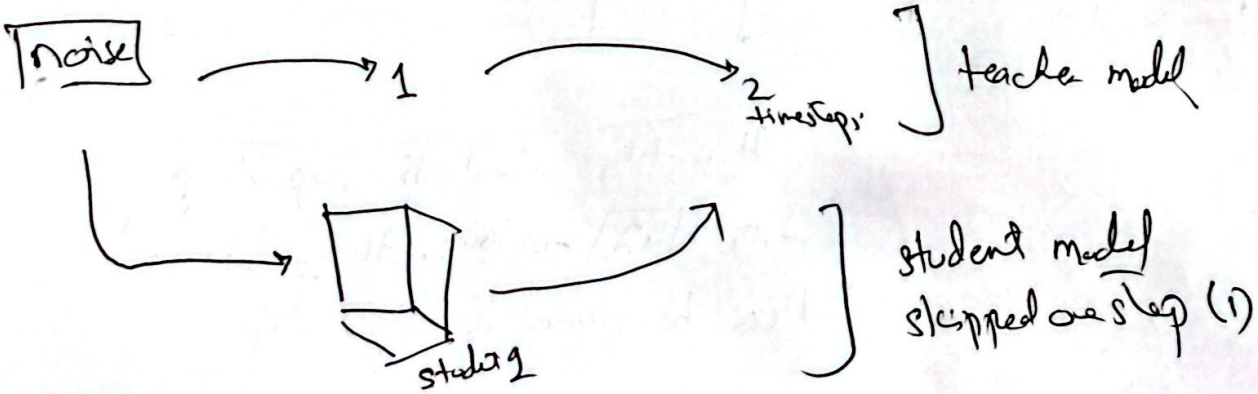
compare $\hat{60}$ predicted output with original output from teacher model 60.

loss = MSE(60, $\hat{60}$)

So, this UNET will learn how to do it correctly.

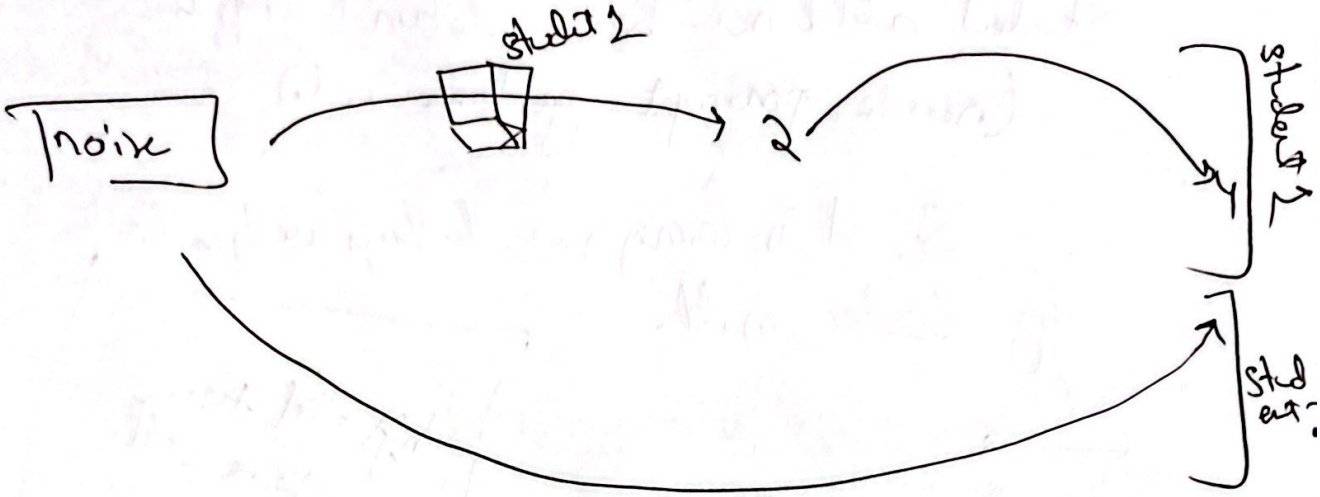
teacher model - already heavily trained model

Timestep is adding noise to original image (here we are adding noise into original image in different steps



TIMESTEPS ARE JUST GIVING MODEL SAME INPUT AGAIN.

2) Now treat student 1 as a teacher.

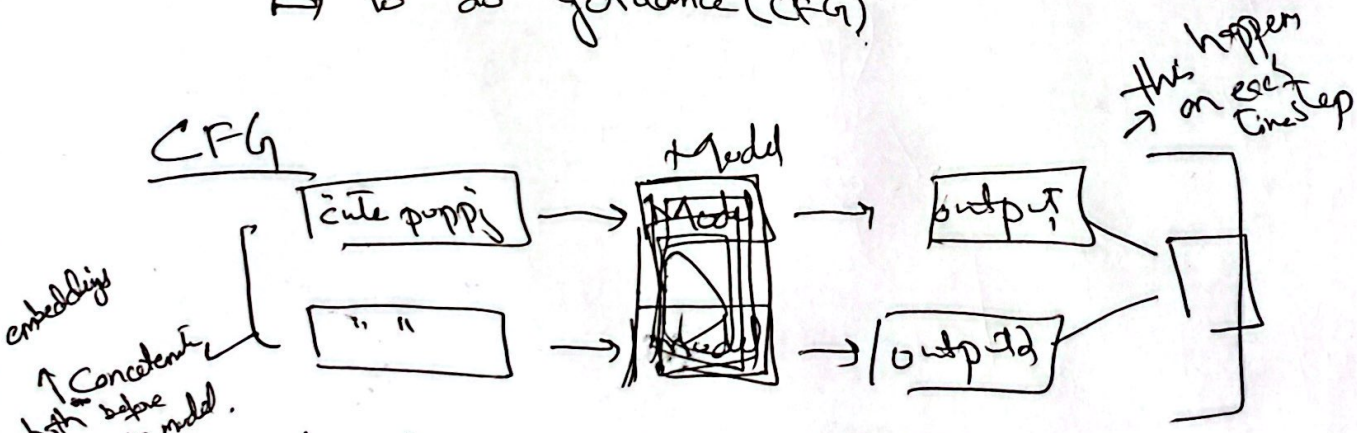


we treated student 1 as a teacher.
and student 2 as a student & go on.

⟶

On Distillation of Guided Diffusion Models.

↳ To do guidance (CFG)



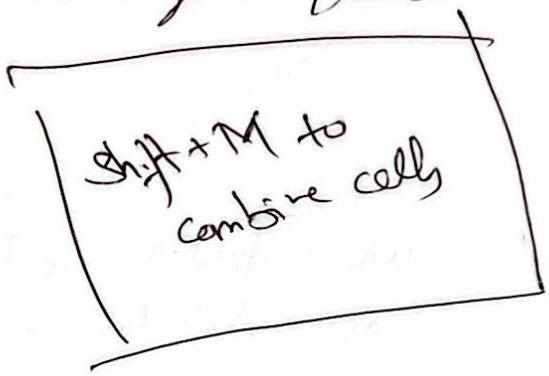
In cfg we take weighted avg of output1 and output2.

[Just like previous paper but add embeddings to. here

So,

Student model now has additional inputs (noise, prompt, guidance scale).

So, it is learning how all things are handled by teacher model.



IMAGIC

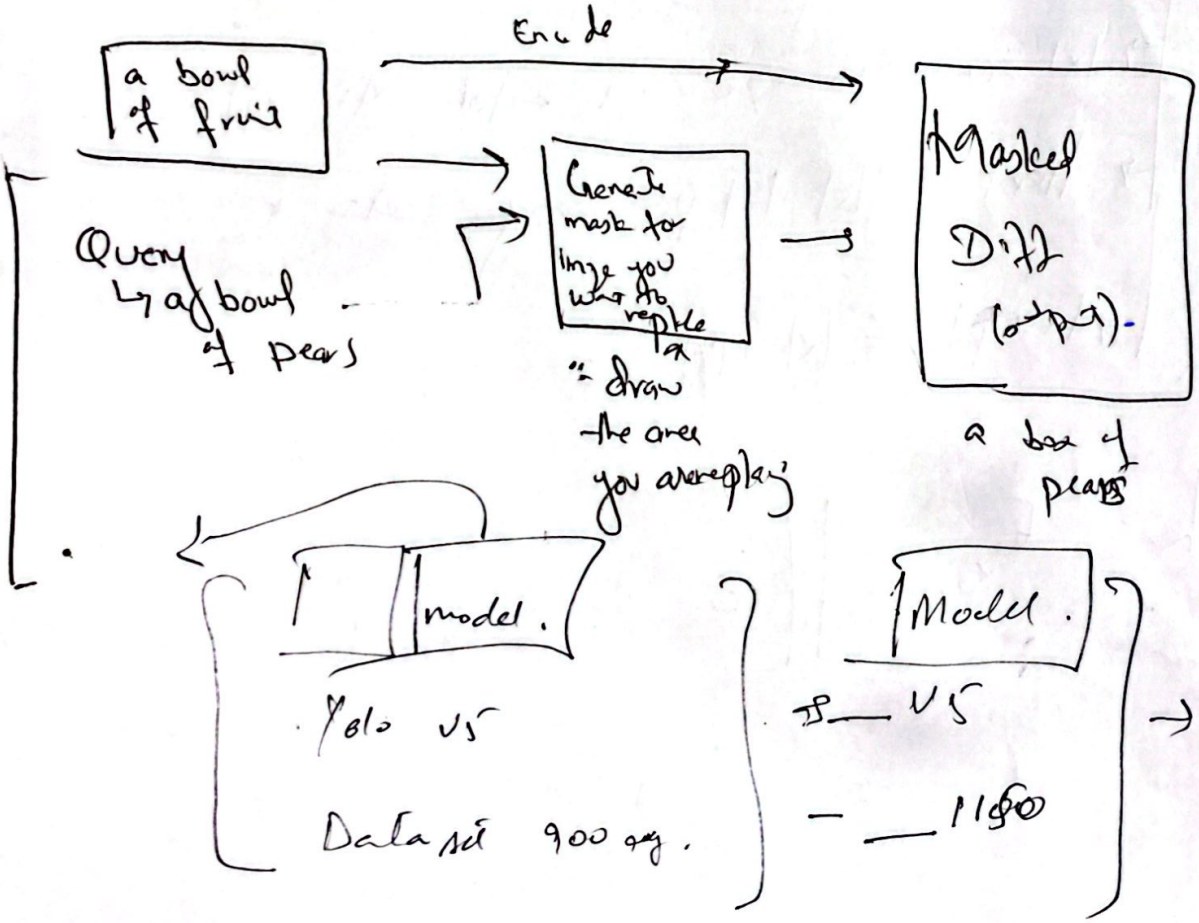
22)

Build from foundations:

DIFFEDIT

Semantic Image Editing

↳ extension of img Generation. We'll edit image based on text query. It is Semantic Image Edit with additional constraint that generated image should be as similar as possible to given ~~output~~ input.



Norm – to get non-ve values

$|$ - vector norm

$\|$ - matrix norm

$$|x|_p = \left(\sum_i |x_i|^p \right)^{1/p}$$

$\epsilon \sim N(0, I)$

I : identity matrix

mean 1

So far

$$|x|_2 = |x| = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

$\|x\|$ is a non-negative norm defined such that

- $|x| > 0$ when $x \neq 0$ and $|x| = 0$ iff $x = 0$
- $\|cx\| = |c| \cdot |x|$ for any scalar k.
- $|x + y| \leq |x| + |y|$ ∵ x or y can be -ve so $\leq$ sign

$$L = E_{x_0, t, \epsilon} \left\| \epsilon - \epsilon_\theta (x_t, t) \right\|_2^2$$

It just mean sum of squares of $\epsilon - \epsilon_\theta (x_t, t)$

Expected value $= x \cdot p(x)$ $\Rightarrow \sum (\text{value})(\text{prob of that value})$

1

23)

&, i

$$L = E_{x_0,t,\epsilon} \left\| \epsilon - E_\theta(x_t, t) \right\|_2^2$$

(ϵ: noise; E_θ: noise estimator (prediction of noise))

E_θ is the noise estimator which aims to find the noise $\epsilon \sim N(0,1)$ that is mixed with an input x_0 to yield $x_t = \sqrt{\alpha_t}\, x_0 + \sqrt{1-\alpha_t}\, \epsilon$,

$\alpha_t \in [0,1]$ (works same as mask).

coeff α_t defines the level of noise and is a decreasing fn of time-step t, with $\alpha_0 = 1$ (no noise) and $\alpha_t \approx 0$ (almost pure noise).

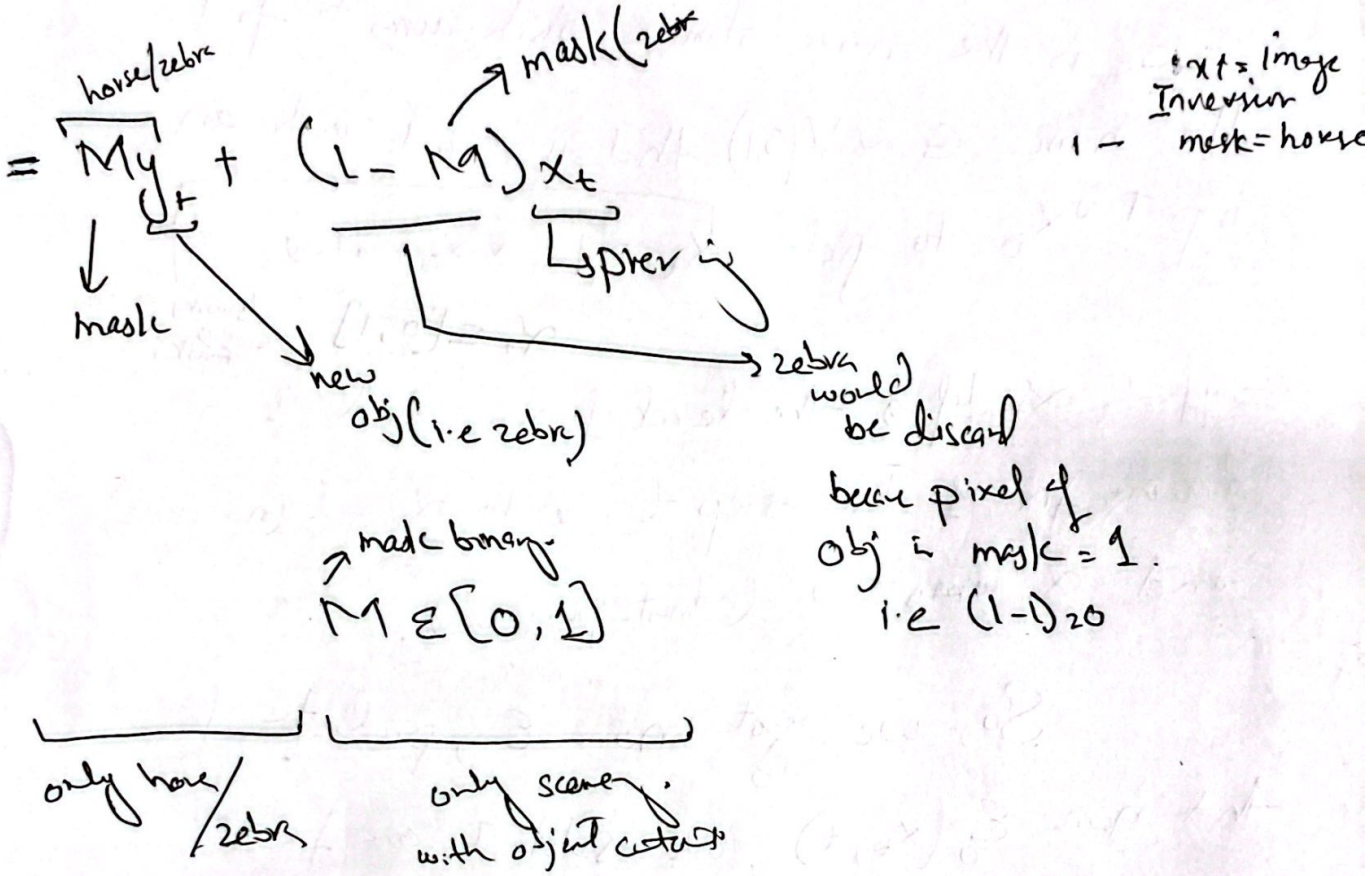
So, we get noise ϵ, predict that noise $E_\theta(x_t, t)$, we subtract one from other and square it to find the expected value.

In other words its MSE.

here $x_t = \sqrt{\alpha_t}\, x_0 + \sqrt{1-\alpha_t}\, \epsilon$

(reduce value of each pixel and noise to it.

The effect of mask is that original pixels in black area wont be changed i.e ignored. So, the scenery would be same.



mask is binaried & it is actually a bit bigger than the object i.e it loosely captures object. That is fine

24)

Automatically Generate a Mask]:

$\longleftrightarrow$

Matrix Multiplication → x_0, i

$$L = \underset{x_0,t,\epsilon}{E} \left\| \epsilon - \epsilon_\theta(x_t, t) \right\|_2^2$$

$$\underset{x_0,t,\epsilon}{E} \left(\epsilon - \epsilon_\theta(x_t, t) \right)^2$$

$$= \frac{\sum \left(\epsilon - \epsilon_\theta(x_t, t) \right)^2}{n}$$

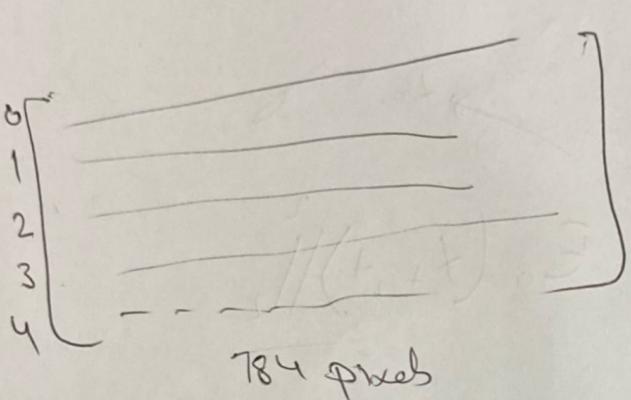
$$\epsilon \sim N(0, I)$$

↳ identity Matrix
↳ = 1

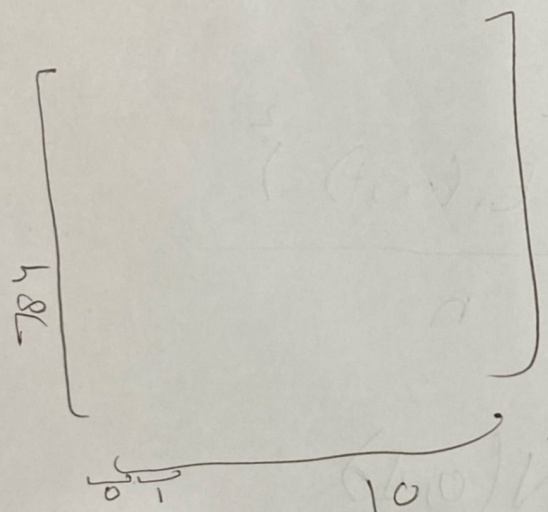
Matrix Multiplication →

M_1 : matrix (first five MNIST digit) : 5×784

M_1



M_2 : matrix (weights) : 784 × 10

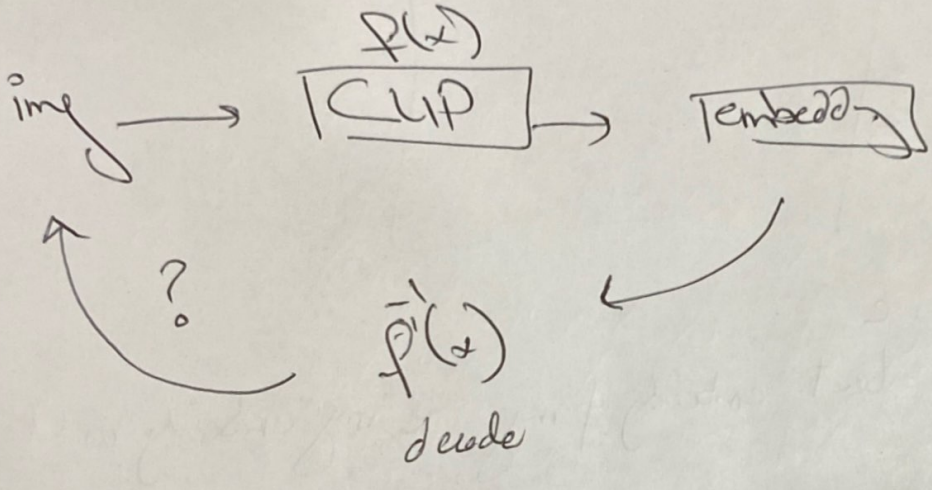


i.e first col to figure 0
second col : : out 1

2) $M_1 \times M_2$

Decoding CLIP | Can we get text prompt from generated img? No

→ You got CLIP embeddg of an image
can you decode it back to img.



Think of `def f(x): return 0`

could you get x back? No

but what if we could put it thru diffusion process?

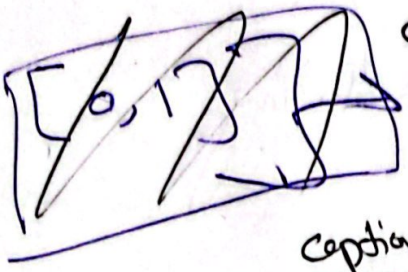
So replace

$f^{-1}(x)$ with noise + embeddg

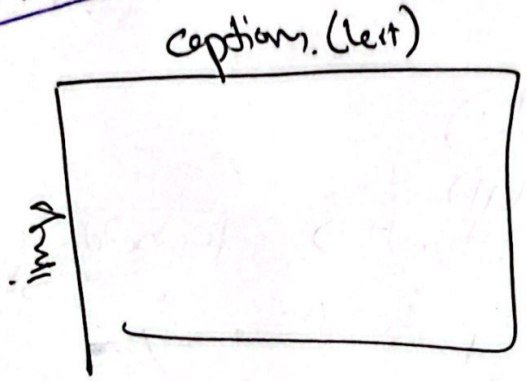
and we can pass it n times.

It might be a bit similar but not exact

But in prompts we don't get image embeddings instead we get text embeddings. but that does not matter.



Remember how we trained CLIP



i.e

text embedding $\approx$ img embedding

CLIP embedding: Text embedding of this picture should be similar to actual embedding of img.

So $f^{-1}(x)$ is basically inverting embedding fn.

These kind of problems are generally referred to as inverse problems.

So, Stable diffusion attempts to approximate the solution to an inverse problem.

27) So, CLIP inverter is not actually inverting the picture to give us back the

"INVERSE PROBLEM"

If we get an image embedding, try to undo that to get back to picture & try to undo that to get back to a suitable ~~prompt~~ prompt is equally infeasible. Both of them require inverting an encoder that doesn't exist.

All we can do is approximate that via diffusion process.

[BLIP is not inverse of CLIP encoder
↳ captioning

Einstein Summation

Here we are using indices not only as subscript but also superscript.

a_i , a^i $\quad \because i$ = index.

$a^i \times a^i = (a^i)^2$] ∴ 2 power.

a^i] ∴ a superscript i

In tensor calculus, a lot of expressions include summation over particular indices.

$$\sum_{i=1}^{3} a_i x_i = a_1x_1 + a_2x_2 + a_3x_3$$

In einstein summation we don't write summation (Σ)

$$\sum_{i=1}^{3} a_i x_i \xrightarrow{\text{in einstein summation}} a_i x_i$$

2p) index refers to indices of a tensor.

Einstein Notation Rules:

↳ Rule 1: ~~If you~~ Any twice-repeated index in a single term is summed over.

Index: 1, 2, 3, ..., n

typically n=3 ∵ because of three dimensions

$$a_{ij} b_j = a_{i1} b_1 + a_{i2} b_2 + a_{i3} b_3$$

↳ where j is a dummy index because it is repeated twice in a term.

↙ $j = 1, 2, 3$

i = free index (can take any value that j takes on, eg i=1,2,3 but only one value. It is not summed over).

So, i can be 1, 2 or 3, etc but it can only be one of these values.

↳ occurs only once in expression but it can not be replaced by another free index. Dumy index can be replaced.

$a_{ij} b_j \neq a_{kj} b_j$

Dummy index is summed over
- appears twice
- can be replaced by another dummy index.

Free index is not summed over
- occurs once
- can not be replaced by another free index

Rule 3
- No index can occur 3 or more times in given ~~index~~ term

$a_{ij}b_{ij}$ ✓ $a_{ii}b_{ij}$ ✗

$a_{ij}b_{jj}$ ✗

we can have only ~~three~~ twice indices that occur once or dummy indices that occur twice in a single term. So, no index occurs more than twice.

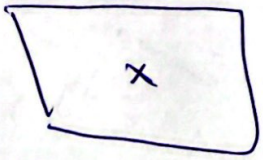
28) "PRACTIC TENSOR MANIPULATION"

MEAN SHIFT

↳ CLUSTRING:

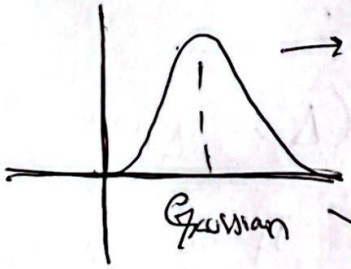
↳ alternative to k-means clustering theorem

For each datapoint, take weighted avg from all other data points in ~~sample~~ X.

Remember avg of all points would be centre of 

So, we are taking weighted avg.

- ↳ closer data points have higher weightage & further away has lowest.
- ↳ So, we create weights for every point compared to one we're interested in using Gaussian kernel.

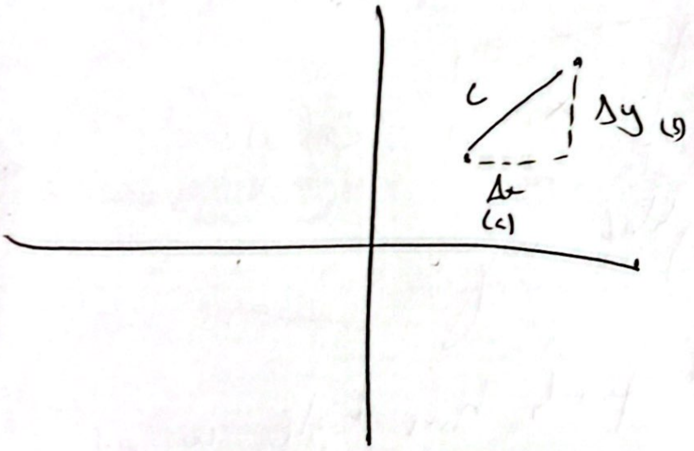


→ centre has highest weightage

→ far from (point of interest) mean has lowest.

If you have ~~sorter~~ shorter two shapes of diff length, we can use shorter length & it will add unit axis to the front to make it long as necessary.

NORMS :



using pythagoras

$a^2+b^2=c^2$

$c^2=(\Delta x)^2+(\Delta y)^2$

$c=\sqrt{\Delta x^2+\Delta y^2}$

but if without using hype

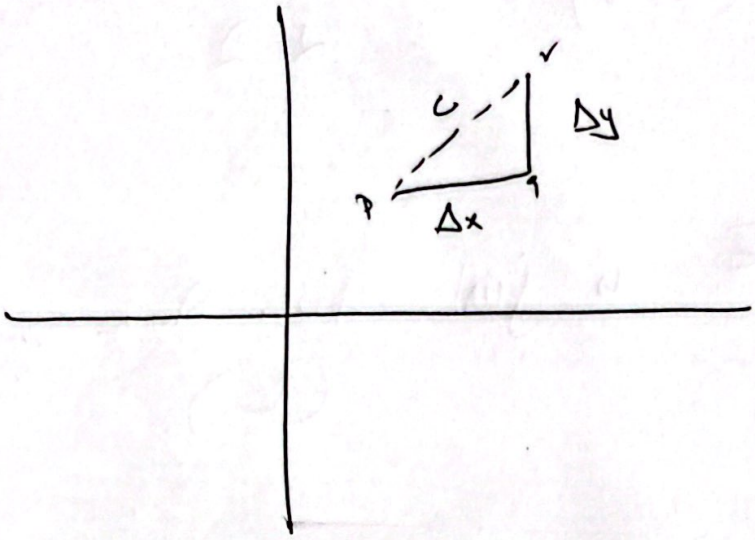
then we first cover Δx and then Δy

$(\Delta x^1+\Delta y^1)^{1/1} = \Delta x+\Delta y$

So, ⇙ If we got list of numbers v we can add them up for each # power α & take that sum $1/\alpha$, this is Norm.

it is $\left(\sum v^\alpha\right)^{1/\alpha}$

Norms



to find c, using pythagoras,

$$a^2 + b^2 = c^2$$

$$c = \sqrt{\Delta x^2 + \Delta y^2}$$

but what if we just cover Δx first then Δy, then its sum would be -

$$c = (\Delta x + \Delta y)^{1/2} \times \Delta x + \Delta y$$

So general eq of norm is

$$\left(\sum |v|^{\alpha} \right)^{1/\alpha}$$

where v is vector / L^2 of number.

↳ because dist can't be -ve

$$\|v\| = \sum v \quad \because \alpha = 1$$

$$\|v\|_2 = \left(\sum v^2\right)^{1/2} \quad \because \alpha = 2$$

So,

RMSE is just two-Norm (L_2)

↙

Root Mean Squared Error. } → Euclidian distance

whereas

MAE is just 1-norm (L_1)

↳ Mean Abs error.

So, $\|v\|_2^2 = \Delta x^2 + \Delta y^2$] as seen in per paper

51)

Generally ~~weighted~~ eq :

$$\frac{\sum x_i}{n}$$

& weighted eq :

$$\frac{\sum w_i x_i}{\sum_i w_i} \quad \text{" divide by sum of weights}$$

X : [1500, 2]

weight = 1500

In Matrix Multiplication, they are compatible if

- One of them is 1
- they are equal

[1500, 1]
[1500, 2]

→ One & Two will match 1500 are equal

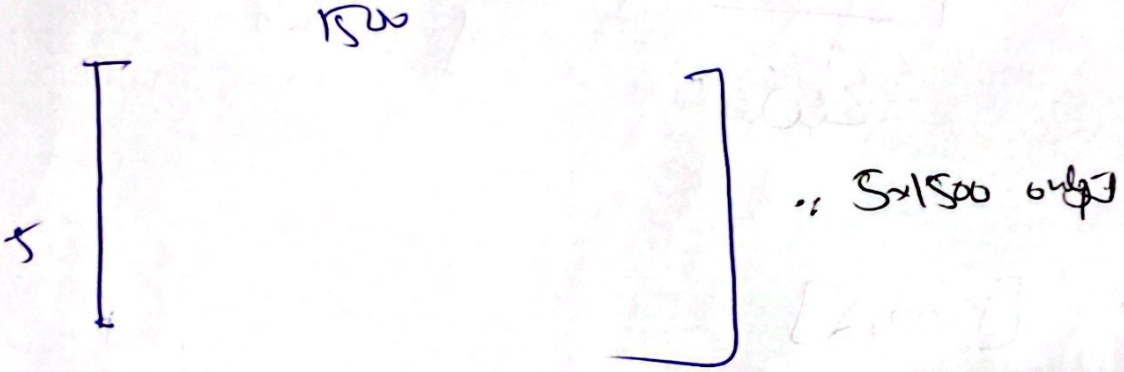
In case of mini batch

$x[bs] = [5 \times 2]$

$\{ X : [1500 \times 2]$

the distance will be b/w every $x[bs]$ with every instance in X and vice versa so

result matrix distance matrix will be



$A[Node] = [1, 1500, 2]$

↳ one dimension $\rightarrow [1500, 2]$

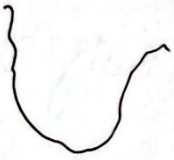
CALCULUS

$f(t)$ = distance

$f(t) =$

$$\frac{f(t_2) - f(t_1)}{t_2 - t_1} = \frac{\Delta d}{\Delta t}$$

but what if slope is curvy.



here comes limit.

$$\frac{f(t_1 + d) - f(t_1)}{(t_1 + d) - t_1}$$: d is a small value

$$\Rightarrow \frac{f(t_1 + d) - f(t_1)}{d}$$ ← difference

if d is min ≈ 0

$\frac{dy}{dx}$ → y on x is result of this small change

dx → x is very small number

< is just < very small number

LIMITS BECOME CALCULUS

FORWARD & BACKWARD PASS :

lets consider a linear model of

pick a single pixel from MNIST digits and that would be our x

y - value is how likely it is #3 based on value of this one pixel.

- pixel value is x
- prob of it being 3 is $y = P(x)$

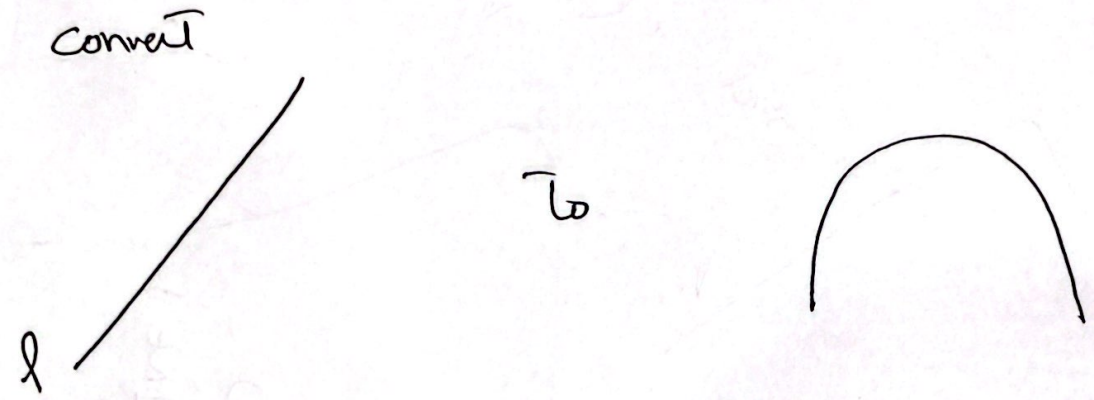
For linear model it says, the brighter the pixel is, the more likely it is that it's number 3.
We are trying to see this like the ∩ but to get this from linear model

Linear model.

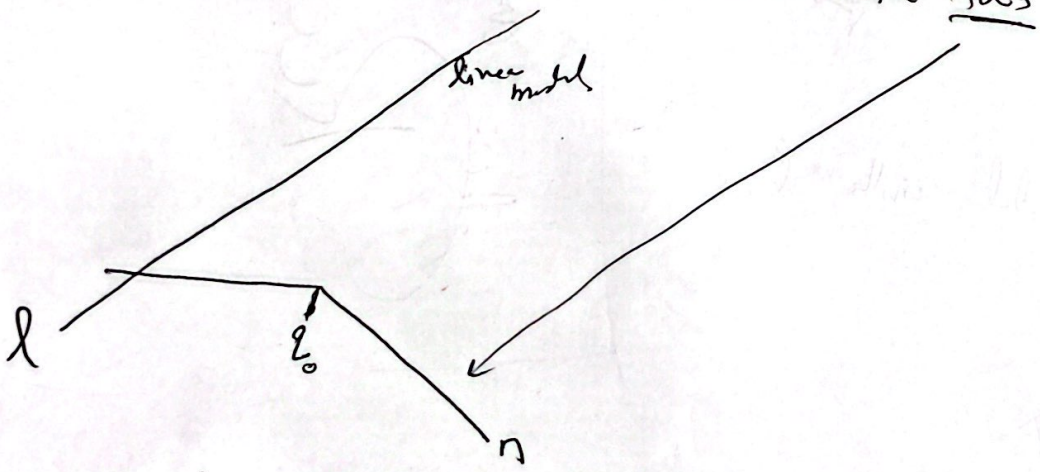
convert straight line to curve.

33)

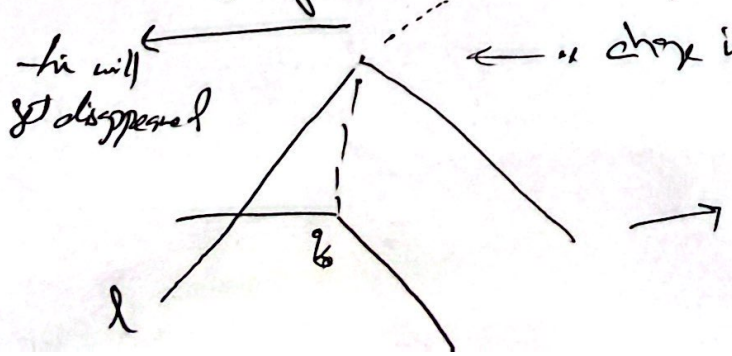
we can use activation fns for that!



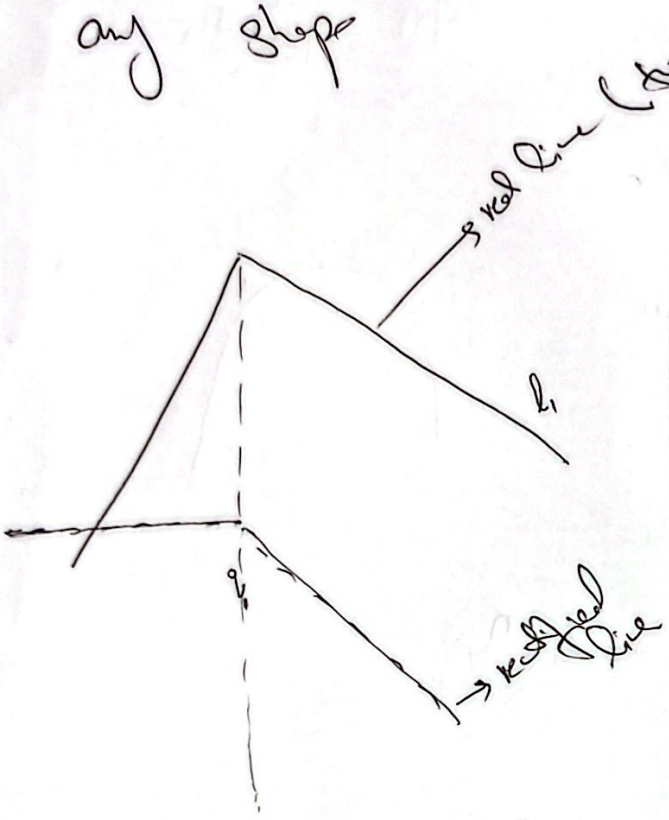
Instead we can create a line 'n' like this,



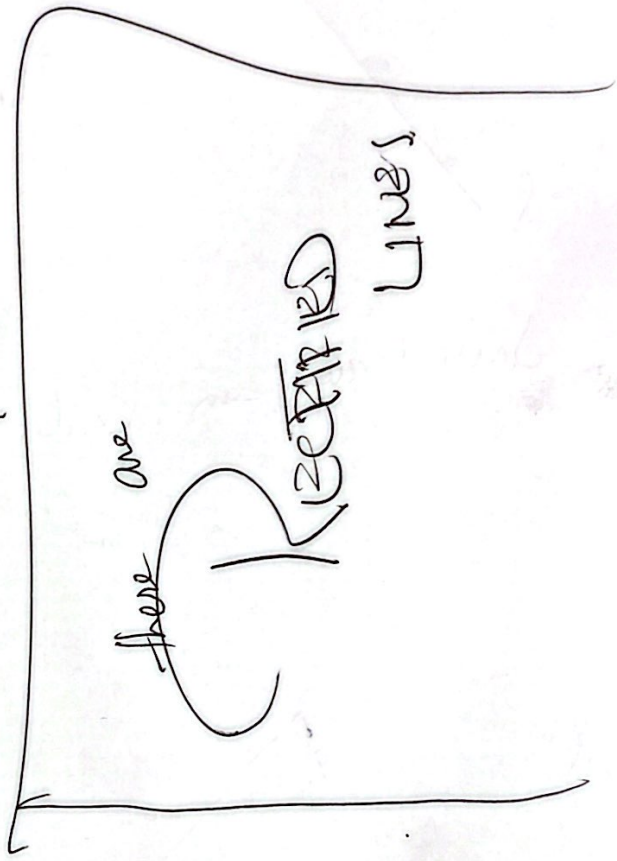
Everything to the left of point z_0 is not going to change if I add these two lines together because this is zero. but anything on right of it is going to be reduced.

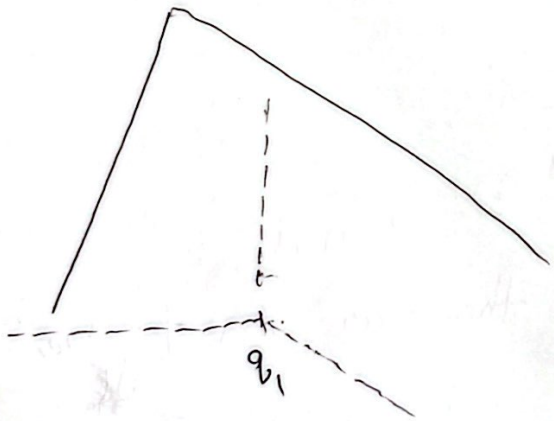


we can keep doing this to get any slope



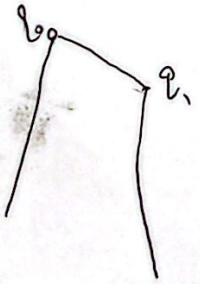
Now if add angle line





we can add angles from this

adding both



So

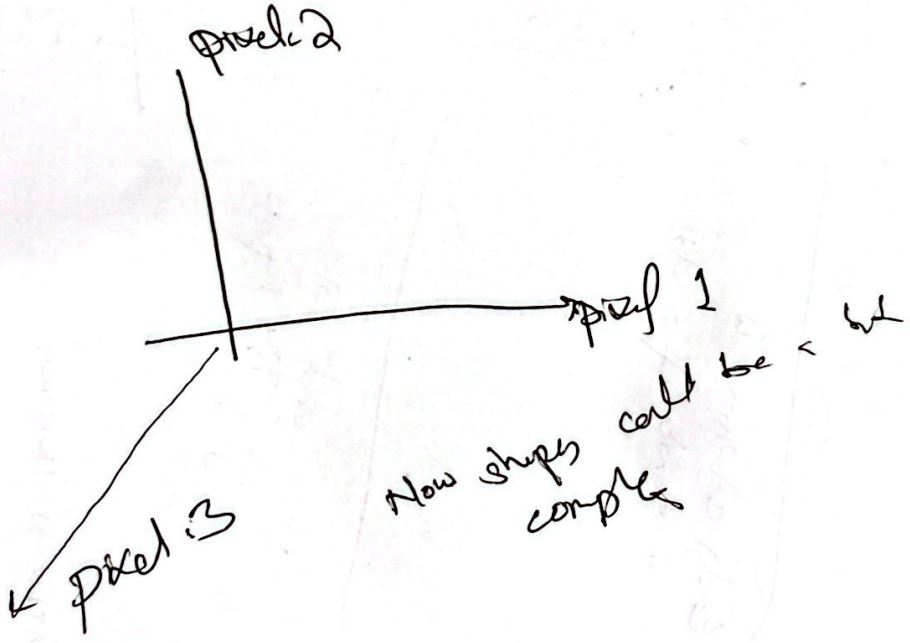
34
a

we are going to have lots of pixels,

Adding a rectified line would make it rectified plane.

↳ If its smaller/less than zero, truncate it to zero.

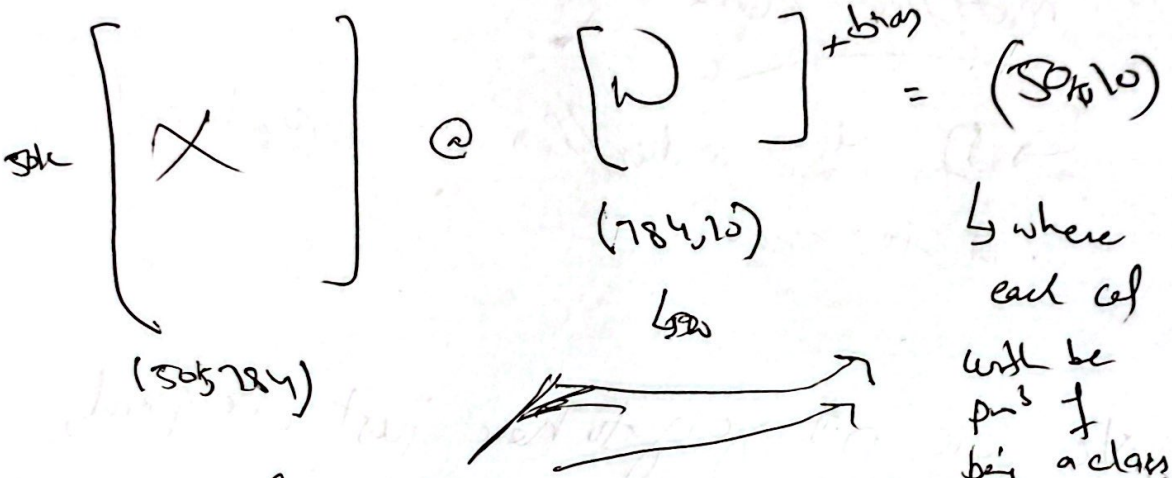
We are not going to have just one pixel but many lots of

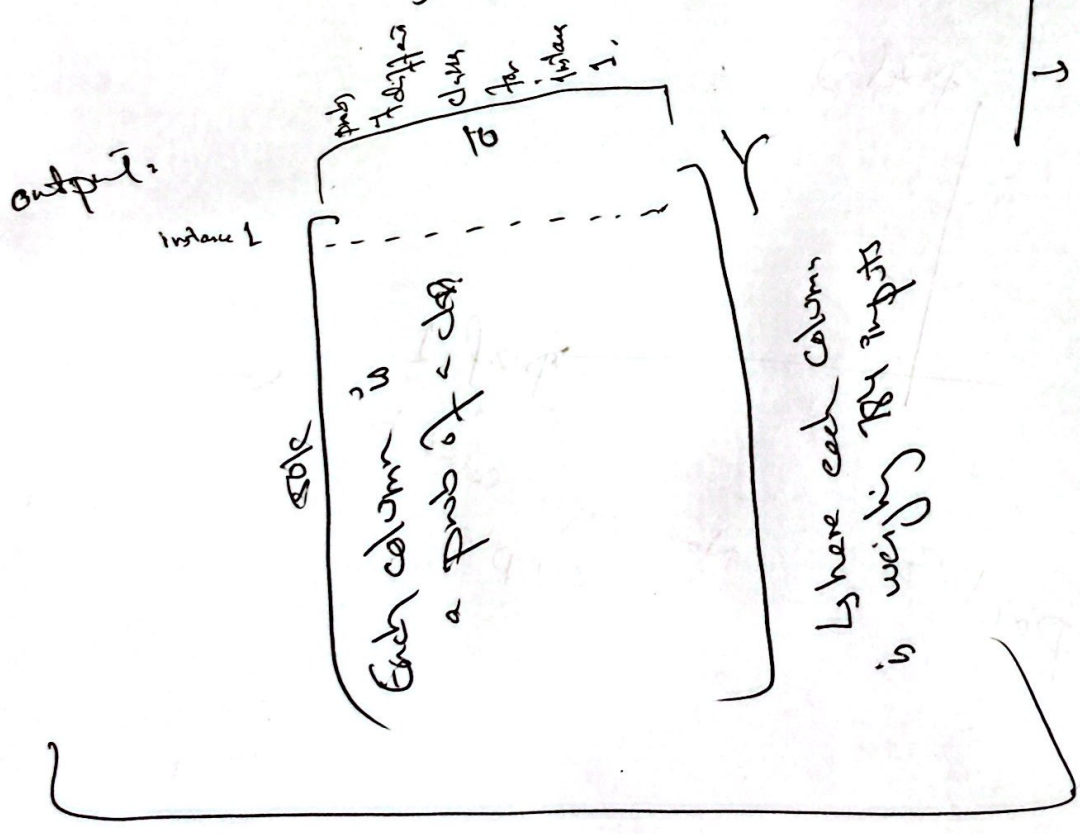


activations

hidden layer = # of Rectified lines to add up

without hidden layer





This was linear model

5y)

b linear:

res = x@w + b

(res - y) =

Gradients (slopes)

f(x, w) - model

loss = L (f(x,w) , y) - loss fn

If we could get derivative of loss w.r.t one particular weight

$\frac{\partial \text{loss}}{\partial w_0}$: If I increase weight a little bit what happens to loss

- If it increase loss, I want to decrease it
- If it decreases, I would increase it

SGD

$SGD = \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$

for

if

multiple inputs x, y (2 outputs)

& derivative $\begin{bmatrix} \frac{\partial z}{\partial x} & \frac{\partial z}{\partial y} \\ \frac{\partial z}{\partial x} & \end{bmatrix}$

↳ for 784 inputs, we'd have 784 of these

Shady with w

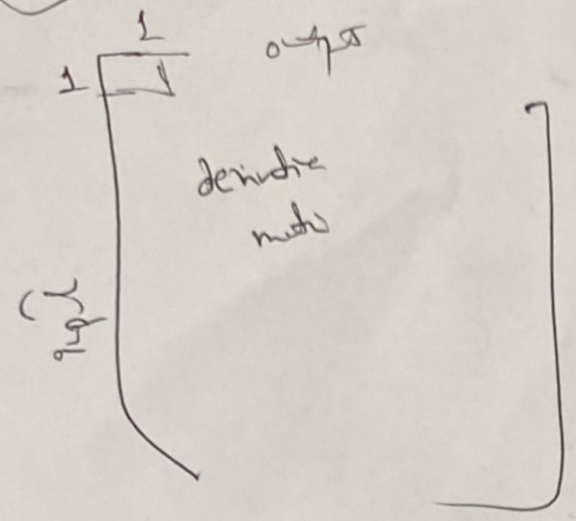
$\frac{\partial L}{\partial w}$ → for all weights

inputs: 784 pixels

output: 50 weights

So, $\frac{\partial L}{\partial w}$ will be long matrix of derivatives, that say for every input that you change, by a little bit, how much does it change every output.

33)



& (1,1) in this matrix will tell how changes output input 1 will result change output 1. by

the we'll get end result a scalar (Mean)

» in organized way

$$\nabla f(x,y) = \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right] \quad \because f(x,y) = 3x^2y$$

$$= [6yx, 3x^2]$$

Jacobian is Metrix of Derivation

Chain Rule: In chain rule, we multiply derivatives

1) $l_1(x, w_1)$: input layer
2) $r(l_1)$: relu of layer 1
3) $l_2(r, w_2)$: layer 2
4) $loss(l_2, y)$: loss computation

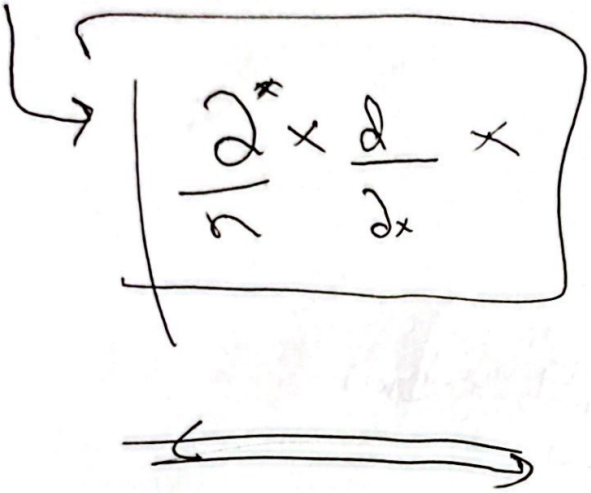
So to compute derivative of 'loss (l_2, y)' but (4) is a fn so, then would need to multiply its derivative with derivative of (3) but that's also a fn, so we'd need to multiply (3)'s derivative with (2) and so on.

This is back propagation

$$\frac{d\,loss}{dl_2} \times \frac{\partial l_2}{\partial l_1} \times \frac{\partial l_1}{\partial w_1}$$

l_2 [lin-grad (l_2, out, w_2, b_2) # out.g are gradients of loss

$$\frac{\sum(x)^2}{n}$$



inp: holds forward pass value
inp.g: holds gradient value calculated during backward pass.

⇒ lin-grad (inp, out, w, b).

↳ inp.g: gradient of the input w.r.t the loss.

This line calculates the gradient of loss w.r.t input 'inp'. Here, the gradient is computed by taking the dot product of the gradient of the output ('out.g') and transpose of weight matrix. This process/operation essentially backpropagates the error from the output to input through the weight matrix

$$\frac{\partial L}{\partial inp} = \frac{\partial L}{\partial out} \cdot W^T$$

⇒ 'w.g' is lin_grad(inp, out, w, b)

↳ Gradient of the weight matrix w.r.t loss

It calculates how the loss changes w.r.t 'w'.

w.r.t weight

$$\frac{\partial M}{\partial w_2} = \frac{\partial (w_2 x_2)}{\partial w_2}$$

$$= (x_2)$$

↳ 0

Multiply inputs, weights & bias with output gradient

p - output
c - continue.

w - whereamI
n - next line

r - return

[Derivative of sum of fn ≈ Sum of derivatives of fn.

*: ©

37)

backward - f_forward(inp, tag)

→ # forward pass

L1 = lin(

forward_pass = call()

↳ to process forward pass

↳ backward()

↓

to get gradients

einsum

'ij,jk→ik'

Cross entropy loss

$$\underset{(50000,784)}{[x]} @ \underset{(784,10)}{[w]} + b; aq = \underset{(50000,10)}{\hat{y}}$$

here we are

$\hat{y}$

50k [10 numbers for each image
0.99 0.01 0.
]

y

[0 0 1 0 0 0 0 0 0 0]

targets would be one hot encoded

or we can just store actual target variable

[6 — 3]

38)

	model output	exp	Softmax	Actual	x_entropy
cat	-4.89	0.01	0.00	0	0
dog	2.60	13.43	0.87	1	-0.05891
plane	0.59	1.81	0.12	0	0
fish	-207	0.13	0.01	0	0
building	-4.57	0.01	0.00	0	0
		15.38	1		+0.05891

Softmax & actual will be compared

$$-\sum_{j=1}^{M} y_i \log p(\hat{y}_i)$$

we can just look at actual label and only look at that value i.e value of it by dog -0.05 matters only, else is discarded

So, For CE loss;

so, $\hat{y}$ (after softmax would

$$[50{,}000, 10]$$

So for each of 50000 k examples we have an actual value (label), so we only need to consider that index value from each prediction row.

$$\{ \text{Softmax}[\text{actual index}]$$