Neural Networks

⇒ Advantages: (1) even if reduced dim coords are complex func^{ns} of our original coords, they are still explicit func^{ns}.

$Y = f(X)$

↓

→ 'f' is invertible so explanation can be computed

→ 'f' is differentiable, in principle ∞-no. of times depending on our choice of NNs.

(in principle with some caveats)

(2) We introduced kernel methods to treat highly nonlinear & complex data manifolds. If we introduce a kernel, we implicitly decide to treat the neighborhood of our data in a given manner (determined by the kernel),

For eg if we say the kernel is Gaussian in distance

$$k(x^i, x^j) = \exp\left(\frac{-\|x^i - x^j\|^2}{2\sigma^2}\right)$$

→ only variational parameter we have is σ

→ so, we have few parameters to play with, & in kernel methods we sort of assume we know the structure/relationship b/w our data / distances.

⇒ In NNs :- there are many variational parameters.

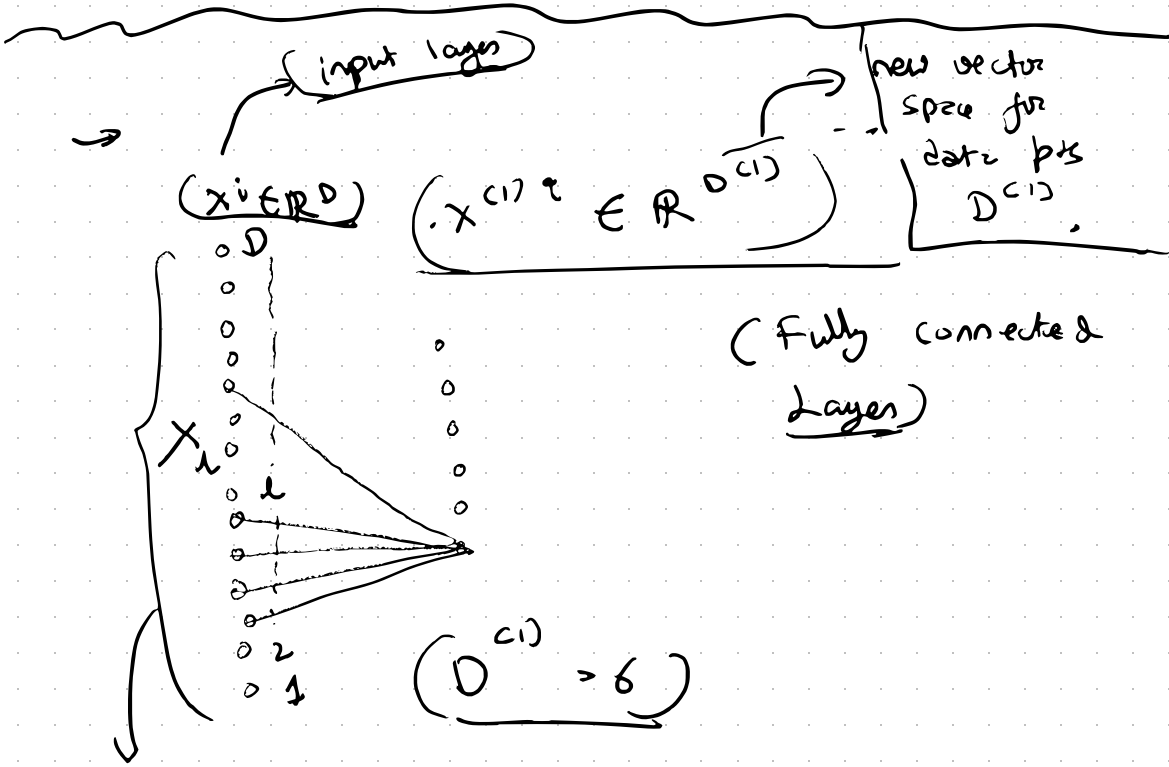
$$Y = f(X|\pi)$$

↳ Y depends explicitly on a very large # of params (π). Ordinary NNs are tuned to reproduce ground truth predictions / represent our data manifold.

In many applications it is really hard to choose / find a suitable kernel. This is where NNs are powerful.

⇒ Disadvantages of NNs :-

(1) Really complicated process & training a NN is like 'alchemy' & is poorly understood.



no. of input nodes in our network = D

⇒ The input node will take as input (x_1), i.e. the 1^{st} feature of our data point.

⇒ We are mapping our data from $x^{(0)} i \in \mathbb{R}^{D^{(0)}}$ in input layer to $x^{(1)} i \in \mathbb{R}^{D^{(1)}}$ → which is the next layer of our network ←

$D^{(1)}$ is the new vector space of our data

→ $x^{(0)}$ → input data

→ $D^{(0)} = D$ → initial dimensional space of our data.

⇒ So this transformation from $x^{(0)} \to x^{(1)}$ is in general performed with an explicit function & there are many possibilities.

→ architecture in NN : function definition that maps data across the layers

→ The example given so far has "Fully Connected layers" architecture.

→ In a "linear" Fully Connected layer :-

① We do something identical to PCA

② $x^{(1)} = W \cdot x^{(0)}$

(W matrix)

↓

(= A in PCA)

→ same operation as PCA

→ transformation done independently for each data point (i)

→ $x^{(1)}$ has dim = $D^{(1)}$

→ $x^{(0)}$ " " = $D^{(0)}$

→ ∴ (W) will be a rectangular matrix with dimension = $(D^{(1)} \times D^{(0)})$

→ if our NN is a Linear func°
is a fully connected layer :-

4 $D^{(1)} = 5$ (as in PCA)

then the functional form of this single layer NN will be identical to PCA.

→ if we then optimize the NN to reproduce the distances, we EXACTLY get PCA.

② if we have a fully connected nonlinear layer :-

↳ we apply "pointwise nonlinearity"

↳ i.e., to each element of $x^{(1)}$ i, we apply independently a nonlinear function of our choice.

→ mathematically

$$x_i^{(1)} = g\left(\sum_{m=1}^{D^{(0)}} w_{im}\, x_m^{(0)}\right)$$

{ component (i) of $x^{(1)}$ }

(nonlinear func^n)

representing $w x^{(0)}$ in terms of components.

$$\left(w x^{(0)} = \sum_{m=1}^{D^{(0)}} w_{im}\, x_m^{(0)} \right)$$

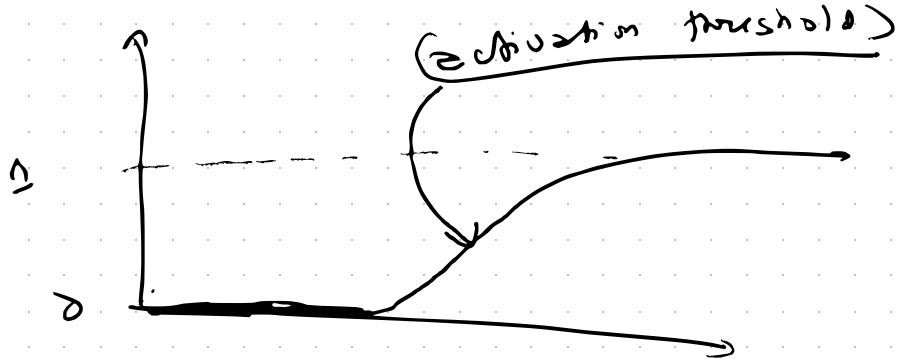
This nonlinear func^n (g) is applied to each component of the generated vector.

→ In principle (g) can be diff. for each (i), but in practice, (g) is same for each layer.

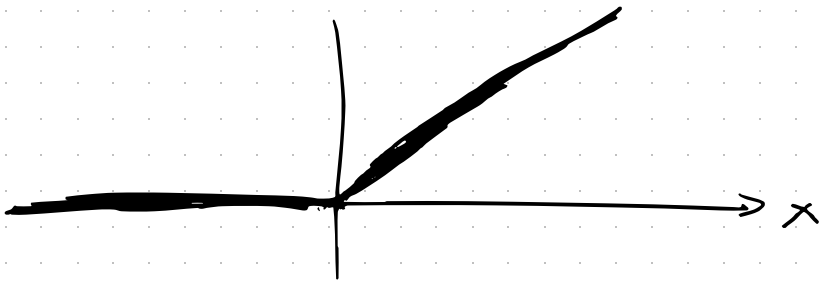
→ Functional forms of nonlinear func^{n} (g):

① $g(x) = \tanh(\,)$

(switches from $0 \to 1$)



② RELU (Rectified linear Unit)



$$\text{RELU}(x) = \begin{cases} x, & x > 0 \\ 0, & \text{else} \end{cases}$$

→ In some cases one introduces an extra additional parameter called a "shift"

→ So we typically optimize a function of the form :-

$$g(x - \lambda_i)$$

↓

"shift" parameter

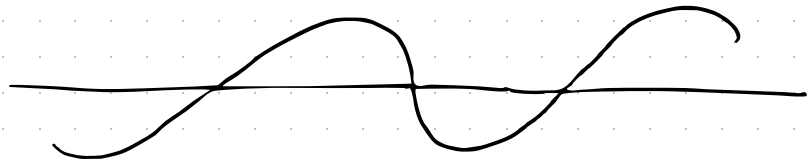
↓

shifts $\mathbf{w} \cdot \mathbf{x}^{(i)}$ by a quantity which we try to optimize

↓

NOTE: λ_i → so can vary from component to component.

③ $g(x) = \cos(x)$



⑤ ... — there can be many different forms of $g(x)$ → any nonlinear form is mostly used if it suits our data.

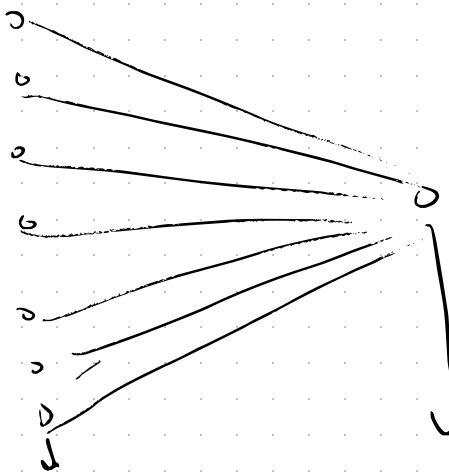
⇒ Why does RELU / tanh get used so much?

① ∵ NN's try to mimic how neurons in the brain work. neuron fire only if total activity is above a certain threshold (activation threshold) & RELU / tanh mimic this property very well.

↳ "Weights"

→ applying these transformations successively in multiple layers made NNs very popular

⇒ Single "Perceptron" model from the '70s.



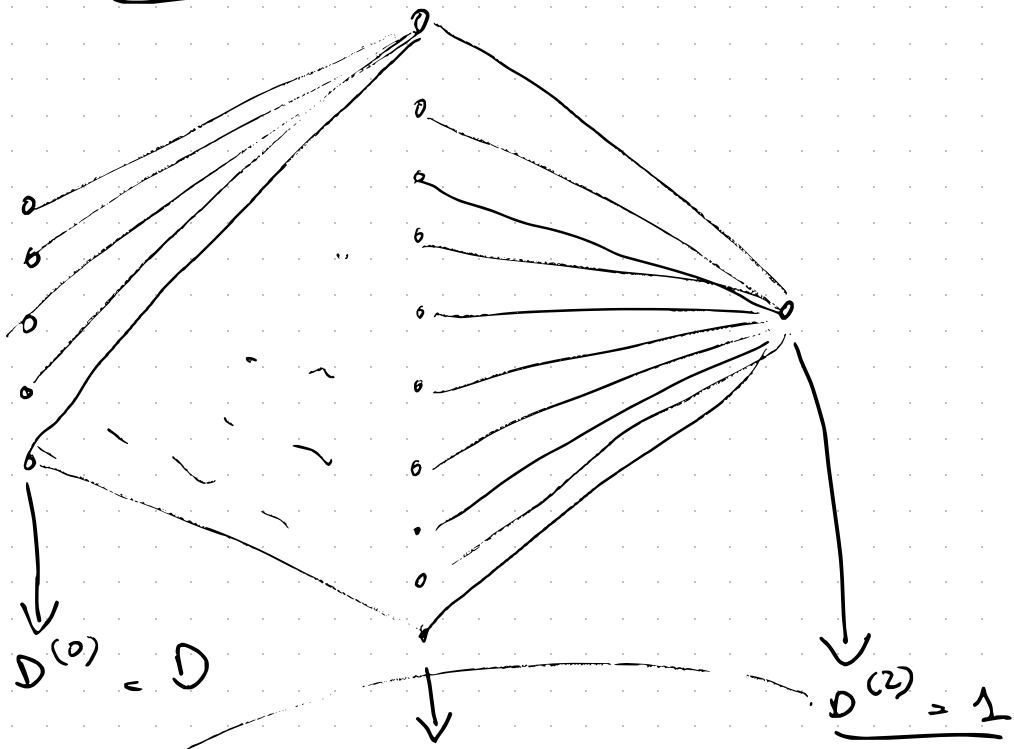
$D^{(0)} = D$

$D^{(1)} = 1$

↓

we want a single real valued result

→ "Perceptron" model in practice :-



$D^{(0)} = D$

$D^{(2)} = 1$

Intermediate layers

$D^{(1)} = \infty$ (∞ # of nodes)

→ We want an output that is only a single real number

"We can represent any nonlinear func^n of the input with RELU / tanh etc. activation func^{ns}"

"Theorem" relevant to Perceptrons

→ $Y = f(x)$

↓

complicated func^n of the input (x)

IFF, it has a single value

the perception architecture allows us to represent this func^n with a given activation func^2 of our choice.

→ Caveat → even if this representation is known & it exists (as the theorem states) algorithmically it is very difficult almost impossible to find it.

(multi-layer Perceptrons)

☺ Breakthrough in MLPs :-

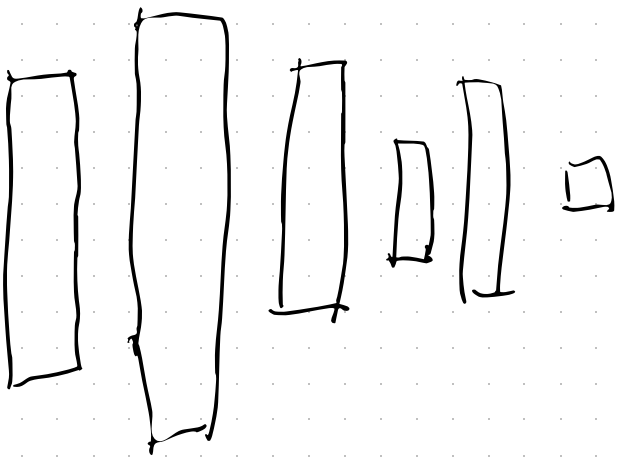
↳ (~2010s or so)

↓

(Hinton, le Cunn etc.)

↓

they realized that concatenating many layers of perceptrons finding the soln becomes easy.



$x^{(0)}$ | $x^{(1)}$ | $x^{(2)}$ | ... | $x^{(L)}$

$D^{(0)} = D$ | $D^{(1)}$ | $D^{(2)}$ | ... | $D^{(L)} = 1$ → simple case where output is a single no. $\in \mathbb{R}$

→ $L \equiv$ no. of layers in our deep NN.

→ each representation is a func of preceding representation.

⇒ i.e. $x^{(l+1)} = f_l(x^{(l)} \mid W^{(l)})$

set of weights / parameters specific to a layer.

→ In principle $f()$ can be diff. for each layer & usually is

→ But for simplicity, let's look at the case where $f()$ is same for all layers.

→ The nature of this approach is that it implements a Markovian Process.

$x^{(l+1)}$ depends ONLY on $x^{(l)}$.

→ There can be skipped links / even bad links but the reason it is useful to implement our NN as a Markov Process is that there is a simple way to optimize it

→ ($l \to l+2$, $l+2 \to l+4$ etc.) "Backpropagation"

In USL, $D^{(L)} = D$ is taken; no dim reduc° is performed in their output

⊡ Training a NN :-

→ For the case of binary classification.

① x^i

(images of cats / dogs)

↓

$Y = f(X | W)$

input data

parameters

$f()$, given an image (x^i) will return a real no., Y which is the "doggishness" of an image.

↓

if ($Y > 0$: dog
$Y < 0$: cat)

We want our NN to be using only differentiable func^{n}.

⇒ For each image :- we have a ground truth label → binary label → $b^i = 1$, if $i \equiv$ dog
$= -1$ $i \equiv$ cat

→ What we want is to find :

$$\tanh(y^i) = \tanh(f(x^i|w)) \simeq b^i$$

by training the NN

we find w, s.t. this is true for all (i)

→ $\tanh()$ is chosen as $f()$

=> How do we find these desired parameters?

→ By minimizing w.r.t. the parameters a "loss func" → which is a func of our parameters & data.

$$L(W|X) = \frac{1}{N}\sum_{i=1}^{N}\left(b^i - \tanh\left(f(x^i|W)\right)\right)^2$$

minimize this loss func^n w.r.t. W

(L2 - loss func^n)

(loss func^n)

measures # errors we make. if all predictions are correct for binary classific^n, $\lambda = 0$. if $\lambda = 0.01$, 1% errors are made.

another possible loss func^n can be the Cross-entropy loss func^n.

$Y = f(x^i|W) \Rightarrow$ chain of func^{ns} $\therefore$ we describe the NN as a Markov Process

$= x^{(L)}$

$$\Rightarrow \therefore \quad Y = x^{(L)} = f(x^i|W) = f(x^{(L-1)}|W^{(L-1)})$$

$$= f\left(f(x^{L-2}|W^{(L-2)})\,|\,W^{(L-1)}\right)$$

$= \ldots\ldots$ (& so on till we get the input)

∴ Basically x^L is written as a

func² ∫ = func² ∫ = func²

⇒ So basically, we take a derivative of the loss func² w.r.t. the parameters.

→ If the NN has a Markovian str., this is easy & such a NN is called a "feed-forward NN"

$\frac{\partial L}{\partial W} \equiv$ compute the derivatives explicitly.

↳ train the NN by (plain gradient descent.)

start :- ① W ~ random gaussian no. (initial guess)

Then :- ② $W = W - \epsilon \frac{\partial L}{\partial W}$

→ $w = w - \epsilon \frac{\partial L}{\partial w}$

⇓

plain gradient descent

$\epsilon \rightarrow$ learning Rate

⇒ This process is called "Backpropagation"

↓

diff. manner of computing derivative of a func^2 of a func^2 of a func^2 ... by chain rule

→ "Deep" NN → it's called deep because many layers are concatenated one after another

↓

⊙ Consequence of this :- In a typical SL/USL setting, we try to learn a model that depends on data (x) & set of parameters (W)

∴ $Y = f(x|W)$ → typical setting of ML training protocol

→ N data points on which we train our model.

Say, N = 10,000

→ Another thing that defines our model is # of free params. It can be diff.

eg. (# of params = dim. of data)

∴ → "n" # of parameters

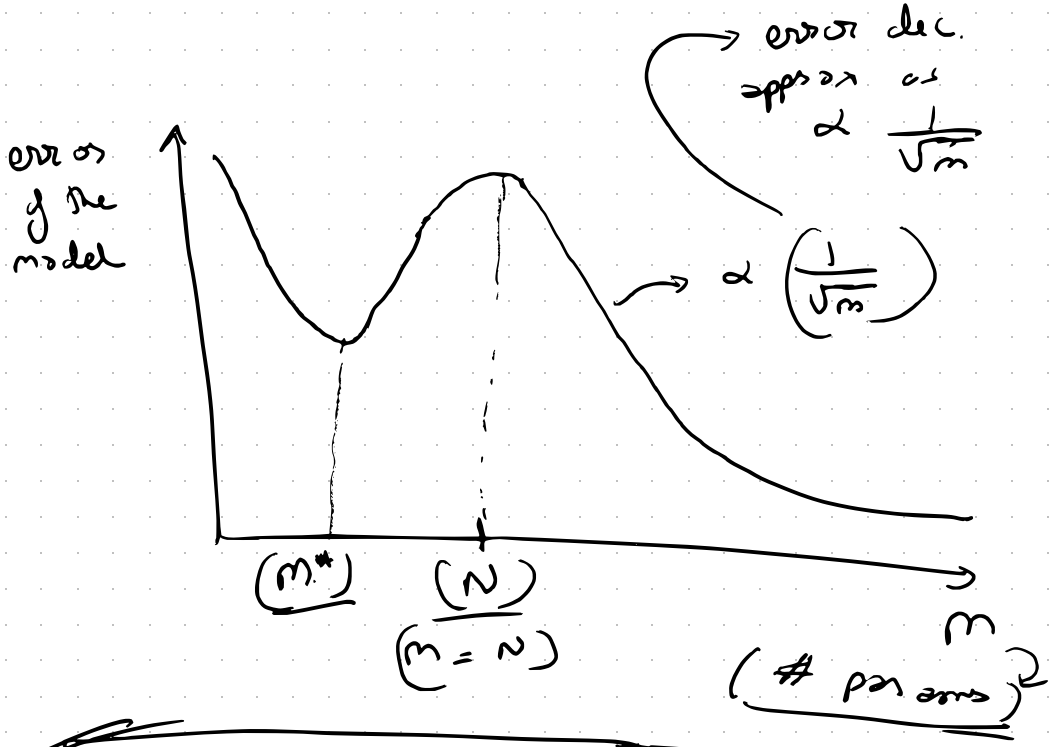
→ "D" → dimension of data

⊙ What if for $N = 10K$, we want $M = 10^6$?

→ normally, we would expect (M) to be 1 order of magnitude less than (N) → data

→ instead in NNs we go at odds with this paradigm & choose $M = 10^6$ for $N = 10^4$.

→ i.e. we choose an insanely large number of parameters to describe our data, infact (# params) much greater than (# data pts).



Double - Descent curve

→ few params → high error

→ increase n → error reduces

→ even more increase → overfit the data → reproduce training data → prediction is garbage ← error increases.

⊙ m^* ≡ best no of parameters in classical statistical learning.

→ Beyond (n^*), error ↑ due to overfitting.

→ error peaks at $(n \sim N)$

params ~ # data

→ then error monotonically ↓.

(double descent phenomena)

↓

observed first in NNs, but also in other models

(·) Laio's interpretation of Double Descent:

→ classical statistical regime

→ N data, n params

→ if training instead of being deterministic is stochastic

↓

stochasticity can be trivial. For eg loss landscape has many local minima we initialize params in a stochastically indep manner & then do gradient descent

→ we do optimization once & obtain the model :-

$$f(x|w)$$

→ if loss landscape is complex, we can initialize 1000 times & so instead of having 1 model we have 2000 models.

Finally, our best guess for (y) is chosen as statistical avg. of the output of 1000 models

$$\therefore y = \sum_{m \in \text{models}} f(x|w^m)$$

So our optimization procedure having a degree of stochasticity improves the quality of our prediction.

→ ; m is initialized randomly

$$y = \sum_{\text{models}} f(x|W^m)$$

has $(1000 \times m)$ # of params.

→ This can also be looked at as a single model with 1000 × more params.

↕

simple case of where we can improve reliability of our prediction by inc. the # of params.

↳ ; error dec $\propto \frac{1}{\sqrt{m}}$, it implies that a deep NN is implicitly defining a large # of statistically indep. models. ; A deep NN can be considered to be averaging over statistically indep. representations of our data.

10 Autoencoders (AE)

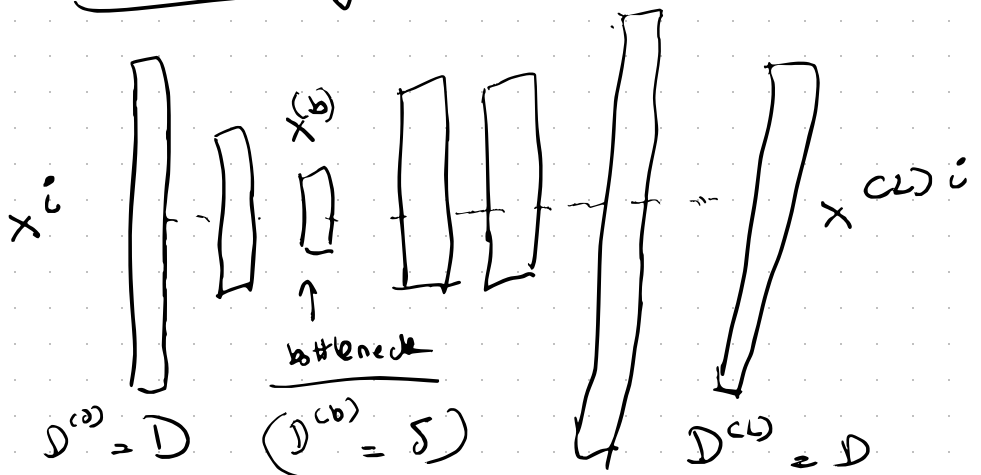
→ all real world autoencoders work in the over-parameterized regime.

→ having an insanely large # params allows us to find a sol^n ∵ having so many params allows us many diff. statistically indep. methods of fitting our data.

→ That's why NNs & AEs work well in the over-parameterized regime.

"Distillation" → train in over-parameterized regime, then kill unused params.

(1) Structure of AE :-



→ input layer takes as input our original data

→ output layer also ALWAYS has (D nodes)

→ L = # of layers

→ x & $x^{(L)}$ belong to same space,

→ in btw 2 we have a chain of other layers.

→ no reason for layers to be symmetric.

→ In AEs there's a layer called a bottleneck which is the smallest intermediate layer with the smallest dimension (least # of nodes).

→ **Dimension of bottleneck = δ** (like in PCA)

→ $Y = X^{(b)}$

low dim. representation of our data that we try to learn $\equiv Y^{(l)} = X^{(b)}$

→ Basically our AE is typically a FF NN (feed fod NN) that performs a Markov Process on our data, tries to reduce the dim. of our data representation upto a minimum value chosen by the user.

→ Train this FFNN / AE :-

(1) define a loss func :-

$$L(W) = \sum_i \left(x^{(L)i} - x^i \right)^2$$

$$= \sum_i \left(f(x^i \mid W) - x^i \right)^2$$

Concatenation of several complex funcs.

output is an explicit func of the input, that is produced by concatenating many layers.

→ each layer's transformation is some

(1) $f(x^{(l)i} \mid W^{(l-1)})$

Layer specific params.

→ ∴ L depends on ALL the params (W)

→ finally $L(W)$ depends on ALL these params.

→ implication of loss func^2 :- We want the A, E, to learn the identity of our original data points.

→ $x^{(L)i} = f(x^i | W)$

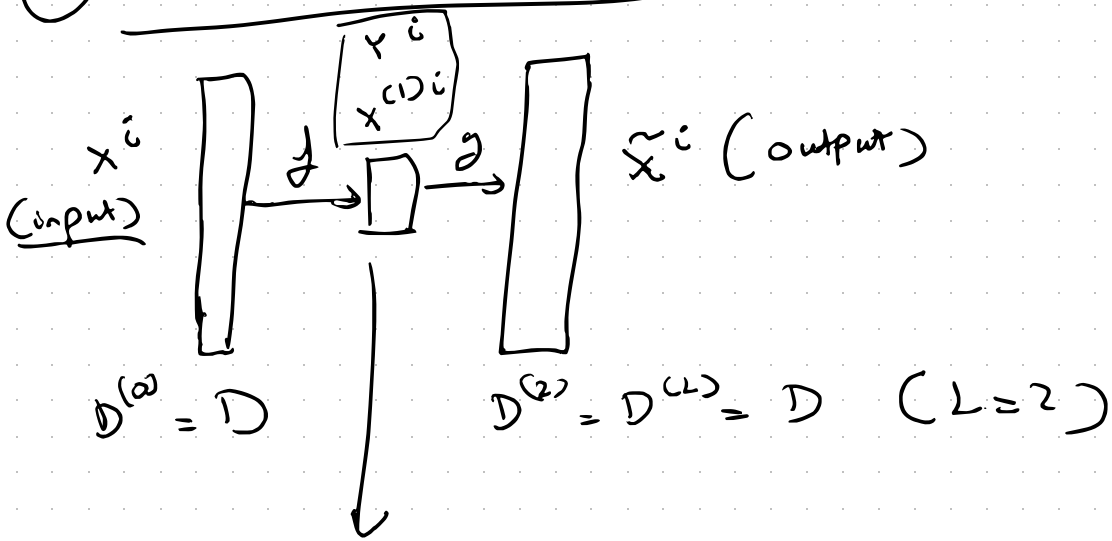
↓

$x^{(L)}$ is a func$^=$ of original data, given the params (W)

→ Goal of the A, E$_0$:- Find W that minimizes $L(W)$

▷ Simple eg. of A.E. :-

① Linear Auto encoder :-



→ only 1 hidden layer

→ 2 func^{n} → f & g

↓

both are linear func^{n}

→ y^i (bottleneck representation) = $x^{(1)\,i}$

$$y_l^i = w_l \cdot x^i \quad ; \; l = 1, \dots, \delta$$

(l^{th} component of vector y^i)

→ Then we can variationally define

w_l

↓

vector of dim = D

⇒ Determine these (δ) vectors to minimize loss func^n.

→ From our representation of data in the bottleneck we have to reconstruct original data. This is linear projection → first project on w, & then we reconstruct original x. we have to represent x on the basis of w.

$$\therefore \tilde{x}^i = \sum_{l=1}^{\delta} Y_l^i \, w_l$$

$$= \sum_{l=1}^{\delta} (w_l \cdot x^i) \, w_l$$

$$[\because Y_l^i = w_l \cdot x^i]$$

→ Rewriting the generalized loss func^n for this specific case :-

$$L = \sum_i (x^i - \tilde{x}^i)^2$$

$$= \sum_i \left(\|x^i\|\right)^2 + \sum_i \left(\|\tilde{x}^i\|\right)^2 - 2 \sum_i x^i, \tilde{x}^i$$

→ Let's choose 'w', that preserves the "norm" of the data, i.e., we choose w, s.t. $\|x^i\|^2 = \|\tilde{x}^i\|^2$

same concept as for deriving PCA can be done with Lagrange multipliers

norm of x^i = norm of $\tilde{x}^i$

→ To minimize L, we need to maximize the scalar pdt. $\sum_i x^c \cdot \tilde{x}^i$

$\because \|\tilde{x}^i\|^2 = \|x^i\|^2$ is already fixed & constant

→ Let's rewrite the scalar pdt:

$$\sum_i x^i \cdot \tilde{x}^i$$

$$= \sum_i x^i \cdot \sum_{l=1}^{\delta} (w_l \cdot x^i) w_l$$

$$= \sum_{l=1}^{\delta} \sum_i (w_l \cdot x^i)^2 \qquad (x^i \to \text{scalar})$$

↓ (rewrite in components)

$$= \sum_l \left(\sum_i \sum_{m\,m'} w_{lm} x^i_m w_{lm'} x^i_{m'} \right)$$

↓ summing over i, we get the covariance matrix (ref: P(A))

$$= \sum_{d} \sum_{mm'} C_{mm'} W_{dm} W_{dm'}$$

$$\therefore L = Tr\left(WCW^T\right)$$

trace → $\sum_d$

multiply by W → $\sum_m$

$\cdot W^T$ → $\sum_{m'}$

this is basically the same as loss func^n in PCA

the only difference is the normalization cond^n which in PCA is implemented by adding the lagrange multipliers.

(☺) TL; DR :- A linear autoencoder in which the dimension of the bottleneck is $< d$ & the encoder & decoder weights (fwd & bwd) are the same, is equivalent to PCA.

→ ∴ PCA can be seen as a very special (simple case) of A.E.

→ In PCA we project our data on a low dim. space & we assume this is done without significant information loss, & so we assume we can reverse our transformation & go back to the original data.

→ if we are NOT able to project back, then PCA is NOT apt.

→ if rank of C is low, then

$\delta = \text{rank}(C)$, $\alpha = \underline{1}$ $C_{ii} x^i$

& $\tilde{x}^i$ are colinear & when subtracting them norm, we get 0)

here we ignore the norm terms in final representation of (loss func)^2 for convenience.

→ if rank(C) > 5, the $\lambda \neq 1$,

& there's higher loss ∴ some info is lost.

→ this shows why in PCA we try to maximize the trace of the covariance matrix, which is the same as minimizing the loss func^n.

→ paradoxical manner of defining an AE.

→ dataset of images → 64 × 64 pixels

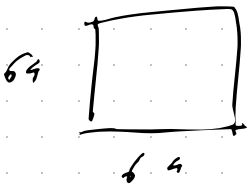
↓

black & white

↓

16 levels of grey

2^4



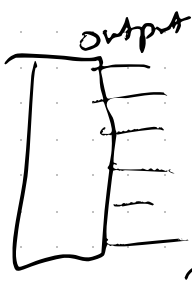
→ pics of 100 diff animals doing diff. stuff. We D:

→ architecture :-

(1) input → pixel of image

(2) output → " " "

(3) bottleneck → single node



like in AE,
input data
= output data

→ Is this possible ?

① It seems NO ∵ we think (so far) that dim. of bottleneck should be ≈ ID of dataset.

② However, this is not only possible it's also quite easy.

(HOW?)

→ We made some assumptions to simplify some things.

→ 16 levels of gray → 2^4 possible values of each pixel

→ 64 × 64 pixels → $(2^6 \times 2^6)$

∴ any possible such image with this level of accuracy & this gray level can be uniquely encoded by assigning $(2^4 . 2^6 . 2^6 = 2^{16})$ bits.

→ So, if this is represented by a single double precision no. (contains 2^{64} bits)

the info we store in each bit of the double precision # allows us to reconstruct the image.

→ However this doesn't mean we have learnt any info of our dataset so far we can "compress" the image, which is a different task.

→ i.e. To obtain an A.E. that can learn meaningful features we need some stochasticity in the training process.

↓

Why? In the architecture we defined now, an image of donkey 4 cat can be mapped quite close to each other in (Y) space.

;; To train an A.E. on just information compression, we violate the rule of USL that data which is semantically close should be close in reduced dim. representation too,

→ ;; Dim reduc ≈ is totally useless ;;
we understand nothing

How to fix this problem?

→ instead of asking the A.E. to reproduce exact input image, we ask the A.E. to reproduce an ensemble of images that are very similar to input image with some tiny added noise.

eg, if input image = cat,

output = an ensemble of cats with slightly longer/shorter whiskers etc.

→ General idea :- train A.E. to reproduce NOT exact input image but something which is almost the same even if we perturb it.

→ So, our previous idea of using a single bottleneck node can't work anymore.

↓

"Denoising Auto encoders"

↓

$$\mathcal{L} = \left\langle \sum_i \left(f(x^i + \eta \mid W) - x^i \right)^2 \right\rangle_\eta$$

η ↓ noise ↓ (can be GDN etc.)

↳ empirical avg. over all realizations of the noise.

⇒ This forces the bottleneck size to be ATLEAST > ID of the dataset ∴ here we force the A.E. to learn the neighbourhood of each pixel ∴ similar images can be categorized effectively by the A.E. now

---

↓

Problems of Denoising A.E. :-

↳ choosing appropriate noise is hard.

→ choosing indep. G.D.N for each pixel is quite arbitrary.

→ ∵ we could prefer noise that is semantically connected, & the images after adding noise on transformed s.t. it makes sense to us physically, & is not just something arbitrary.

5 Variational AutoEncoders (VAE) :-

→ helps solve this problem with "denoising A.E."

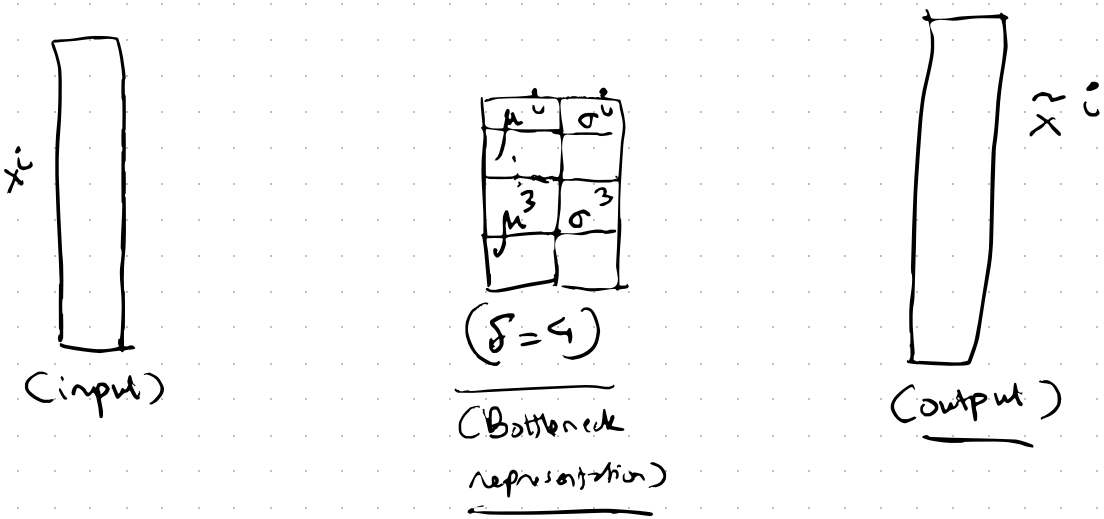
→ **Key idea :-** Adding the noise in the bottleneck.

→ in denoising A.E., we add (η) to x^i, i.e. we add noise to the input, where we have something arbitrary.

→ We assume for a meaningful A.E., the intermediate representation in the bottleneck will have some semantic meaning. So, images of cats should be similar somehow in δ-space. So we add noise in the bottleneck step ; if after this it reproduces something close to original image we can say that the VAE learnt the relevant semantic features.

→ How to do this in practice ?

(1) Architecture of VAE :-



→ assume :- what we learn is not a single representation of our input $x^{(b)i} = y^i$ (bottleneck representation) but rather it is a probability density in $\{y^i\}$ space,

→ Say, $\delta = 4$ $(D^{(B)} = 4)$

→ We learn :- μ^i & σ^i $(i = 1, \ldots, \delta)$ (4)

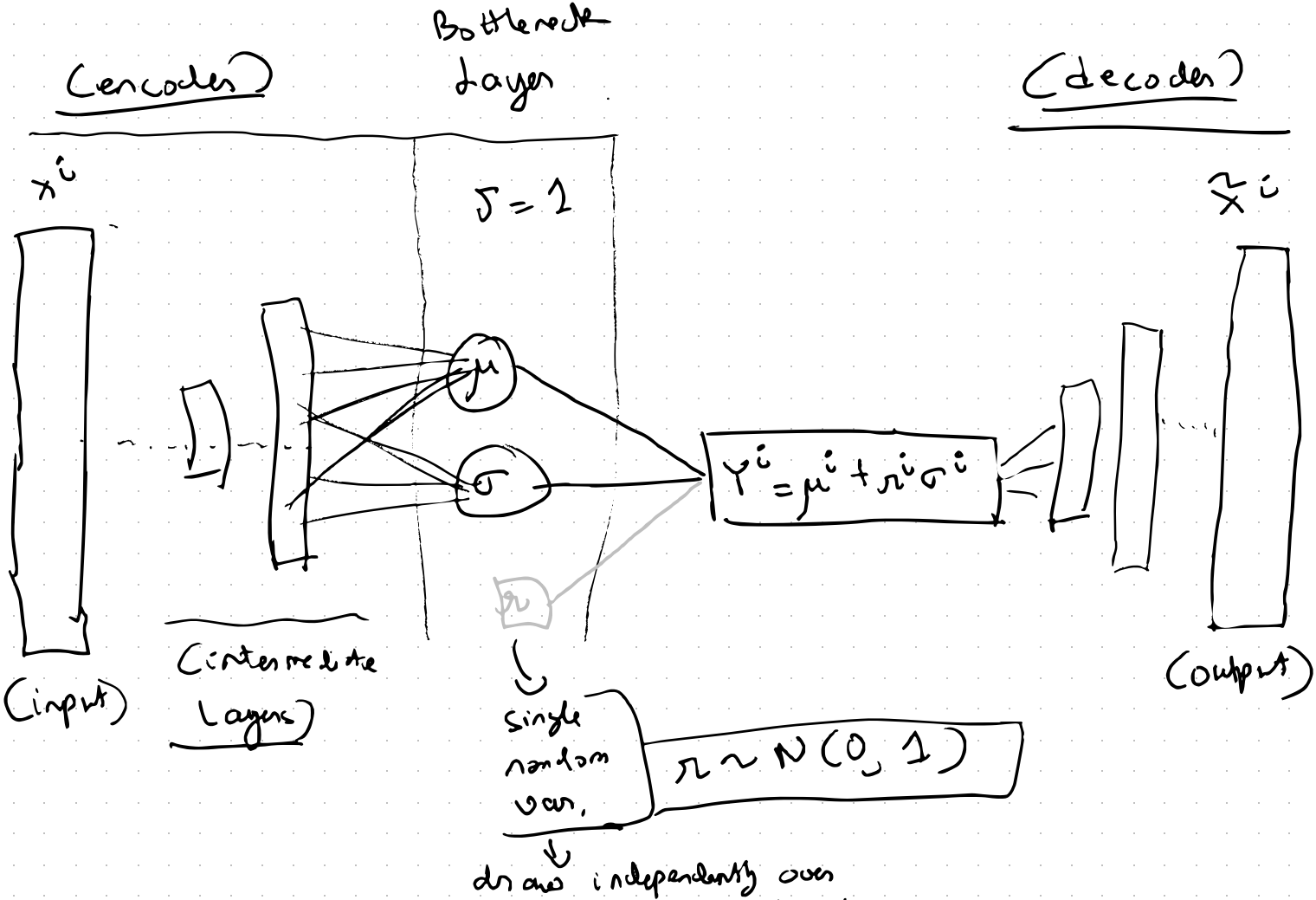
→ μ^i & σ^i plays role of $\langle y^i \rangle$ & std dev. of y^i

$\Rightarrow$ ∴ We can sort of see component (i) of $\varphi^i \rightarrow \varphi^i_x$ as harvested from a normal distribution of average (μ^i_x) & variance (σ^i_x).

$$\therefore \boxed{\varphi^i_x \sim N(\mu^i_x, \sigma^i_x)}$$

So, instead of learning directly the representation of φ^i, we learn the parameters of a Gaussian distrib° from which we can then harvest (μ^i) and generate a representative representation.

$\Rightarrow$ Let's try to implement this architecture in practice. For simplicity, we take $\sigma = 1$ & so we have 2 nodes in the bottleneck layer (1 for μ, 1 for σ).


Bottleneck layer
(encoder)
(decoder)
x^i
$\nu = 1$
μ
σ
$y^i = \mu^i + r^i \sigma^i$
r
$\tilde{x}^i$
(input)
(intermediate layers)
(output)
single random var.
$r \sim N(0, 1)$
draws independently over several cycles. Can be diff. for each index.

→ (z) is harvested from a Gaussian / Normal distrib^n with mean = 0, var = 1

s.t. that we transform y^i to a Normal Distrib^n with mean = μ^i & var = σ^i.

⇒ $$\mathcal{L}(w)_2 \equiv \sum_{i,r} \left(f(x^i, r \mid w) - x^i\right)^2$$

↓

added in the bottleneck layer

So, minimizing this loss func^n, we learn not only a low dim. representation for each data pt. (or image), we also learn how reliable the representation is

→ In denoising A.E., noise is chosen by us. So, for large (2), we get crap results; we blur all the image.

→ In VAE, the level of noise is learnt by the network itself, ∵ the network must produce output (after decoding) at least similar to input.

→ We must add some constraints on the optimizer, ∵ if $\sigma^i = 0$ there's no noise term, & we have same problem as before.

→ How to fix this?

① We add to loss a prior on the values of μ & σ.

↓

expected prob. distrib we'd like to see on the data.

↳ We have many different choices.

→ Simplest possible choice of Priors :-

$\mu^i = 0 \quad \forall \; i$

$\sigma^i = 1 \quad \forall \; i$

} arbitrary choice for all priors

↳ We can use additional info. to drive our choice of priors.

→ Key criteria is to ensure that $\sigma^i \neq 0$ regardless of other choice of prior.

→ Another choice of priors :-

→ μ^i → uniform $\forall \; i$

→ σ^i → forced to be $\neq 0 \; \forall \; i$

∴ Loss func$^{=}$ is re-written as :-

$$L(W) + \frac{1}{2} \sum_{i=1, l=1}^{i=N, l=S} \left((\mu_l^i)^2 + (\sigma_l^i)^2 - \log \sigma_l^{i\,2} \right)$$

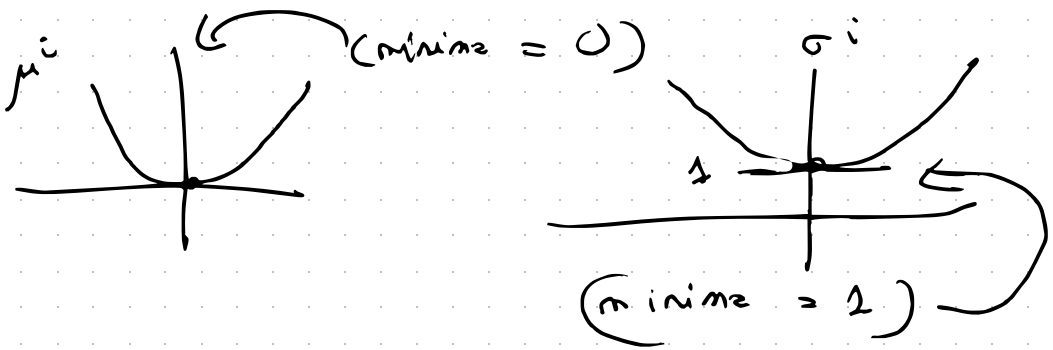
↓ (as before)

↑ favours μ_l that are close to 0

↳ the func$^{=}$ has a minimum for $\sigma^i = 1$

⇒ KL divergence :- (look it up)

$$KL\left(N(\mu, \sigma) \,||\, N(0, 1)\right)$$



→ This breakthrough of VAEs made them a competitive technique for dim. reduc^n.

---

① Take - Home Message :-

(1) Relation btw^n AE & PCA

(2) manner of introducing noise is a very relevant principle

---

② VAE v/s SNE :-

(i) Y^i's are explicit func^n of X^i in VAE

(ii) VAE's dim reduc^n can be inverted.

∴ So, VAE is better for interpretability than SNE & other methods (except for PCA) ∵ (Y^i) is explicit func^n of (X^i)