Deep Multi Layered Perceptrons : 1980's to 2010s (2.1) :-

→ Upto 2010 people were using NN with 2-3 layers. Because of Vanishing Gradient & lack of data. Also no computing power.

→ From 2010, we started having lots of labelled data. We also started having GPUs. New ideas & algos as well.

→ Approach was also changed. Instead of going theory first, experiments were performed more which took lot less time than coming up with new theory.

Dropout Layers & Regularization (2.2):-

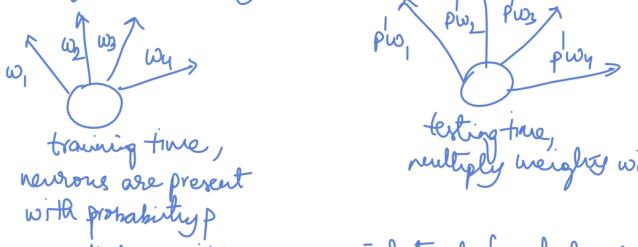
→ one of the major issues of Deep NN is overfitting.
We use regularization to overcome this.

→ In random forest, we get many trees. But we only select subset of trees & subset of features.
We use randomization of features to create regularization.
Core idea of dropout layer is to use this philosophy for MLP's

→ In each iteration randomly remove some neurons. As a result, some incoming & outgoing connections will be lost.
dropout rate $0 \le p \le 1$. At any iteration, in each layer $p*100$ % of neurons shall be inactive.

→ Dropout is very similar to having random subset of features in Random Forest.

→ During testing we multiply w with p



→ When we are overfitting with many weights & few data points, dropout rate should be large.

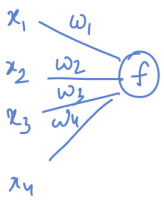
→ Dropouts can also be thought of as a layer that takes inputs from one layer but does not send all of them to the next layer.

Rectified Linear Unit (ReLU) (2.3):-

→ One of the main problems with classical NN is vanishing gradient problem.

→ ReLU is the most common activation function for MLP these days.

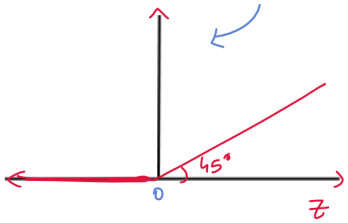
→



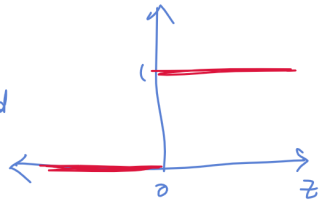
input of $f = \vec{w}^T x$

$f(z) = (z^+)$ → denotes of ReLU.

$f(z) = \max(0, z) = \begin{cases} 0 & \text{if } z \le 0 \\ z & \text{otherwise} \end{cases}$



when ⇒ differentiated



↳ Not differentiable at 0
Since it's not a smooth functions.

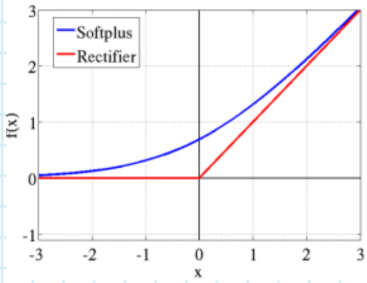
→ $\frac{\partial f_{ReLU}}{\partial z}$ is $\begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$ ⇒ As a result no vanishing gradient, no exploding gradient.

↳ But this 0 could cause dead activations

→ ReLU converges faster than tanh/sigmoid because there's no vanishing gradient/exploding gradient.

→ ReLU is sometimes also called simply as rectifier.

→ Softplus function is like an approximation of ReLU



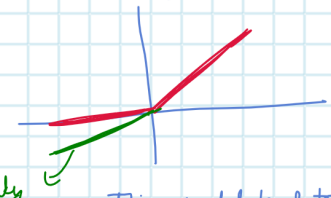
$f(x) = \log(1+\exp(x))$

$\frac{\partial f}{\partial x} = \frac{\exp(x)}{(1+\exp(x))}$ → logistic function.

→ Variations of ReLU :-

Noisy ReLUs :- $f(x) = \max(0, x+Y)$, $Y \sim N(0, \sigma(x))$

Leaky ReLUs :- $z = W^T x$, if weights are -ve,
then z = -ve
if z = -ve, $\partial f(z) = 0$
→ the product of this derivative with others in chain rule = 0
→ $w_{new} = w_{old} - \eta \left(\frac{\partial f}{\partial w}\right)$
$\Rightarrow w_{new} = w_{old}$. weights are not updated.
This state is called dead activation state.
We must always keep track of this when using ReLU
Leaky ReLU fixes this



$f(x) = \begin{cases} x & \text{if } x \le 0 \\ 0.01x & \text{otherwise} \end{cases}$ $\quad \frac{\partial f}{\partial z} = \begin{cases} 1 & ; z \text{ +ve} \\ a & ; z \text{ -ve} \end{cases}$

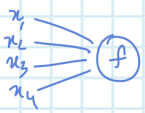
↳ it's the hyperparameter typical $0 \le a \le 1$

This could lead to vanishing gradient sometimes.

Weight Initialization (2.4) :-

→ In SGD, we initialize randomly (uniform (or) Gaussian) ie, $w_{ij}^k \sim N(0, \sigma)$

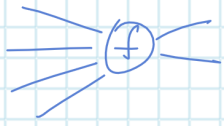
→ Problem of Symmetry :- when all weights are initialized to 0/1/... this is bad.
if f is also same, then
(i) All neurons compute the same thing.
(ii) Same gradient updates
This is called problem of symmetry.
We want stuff to be asymmetric because we want each neuron of layer to learn something different



$z = (W^T)X \Rightarrow z = 0$
if ReLU, whole thing becomes zero.

→ if we initialize to large -ve values, then $f(z) = 0$ if ReLU is used. if Sigmoid or tanh is used, leads to vanishing gradient problem.

idea ① :- → weights should be small
→ not all zero
→ good variance
We can do $w_{ij}^k \sim N(0, \sigma)$ σ is small.
This is called Gaussian/Normal initialization
As a result we get good σ and there will be both true & -ve weights



for f, fan_in = 4
fan_out = 2

→ After lots of experiments, people concluded, weights should be based on fan_in, fan_out.

idea ② :- Uniform initialization :-

$$W_{ij}^{k} \sim \text{Uniform dist}\left[\frac{1}{\sqrt{\text{fan\_in}}}, \frac{1}{\sqrt{\text{fan\_out}}}\right] \quad \text{for any neuron}$$

This works well for sigmoid functions.

idea ③ :- Xavier / Glorot :-

two variations.

ⓐ Xavier / Glorot Normal :-

$$W_{ij}^{k} \sim N(0, \sigma_{ij}) \qquad \sigma_{ij} = \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}$$

} we use fan_in & fan_out in this. In prior one we only used fan_in.

ⓑ Xavier / Glorot Uniform :-

$$W_{ij}^{k} \sim U\left[\frac{-\sqrt{6}}{\sqrt{\text{fan\_in}+\text{fan\_out}}}, \frac{+\sqrt{6}}{\sqrt{\text{fan\_in}+\text{fan\_out}}}\right]$$

Works well for sigmoid too.

idea ④ :- He-init :-

ⓐ $W_{ij}^{k} \sim N(0, \sigma) \qquad \sigma = \sqrt{\frac{2}{\text{fan\_in}}}$

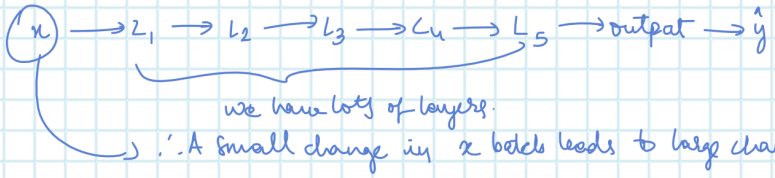
ⓑ $W_{ij}^{k} \sim U\left[-\sqrt{\frac{6}{\text{fan\_in}}}, +\sqrt{\frac{6}{\text{fan\_in}}}\right]$

W well for ReLU.

Batch Normalization (2-5) :

$$\tilde{x}_i = \left[\frac{x - \mu}{\sigma}\right] \qquad \sigma = \text{std-dev}, \quad \mu = \text{mean}$$

→ In minibatch SGD we only send batch of points



we have lots of layers.

∴ A small change in x back leads to large change in w.

batch 1
batch 2
⋮

→ L_1 wont see much difference.
But L_6 will because of chain rule before it.
This is called internal covariance shift.

→ In Batch Normalization, we add a new layer that normalizes the data of that batch only.

→ γ & β are to be learned in backpropagation

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \tilde{x}_i = \gamma \hat{x}_i + \beta$$

ε → small value.

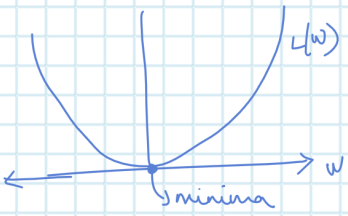
→ Batch Normalization is differentiable.

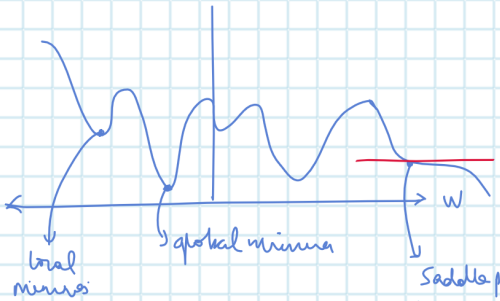
→ During testing, we take the mean & variance of the whole dataset. □

→ Generally, if we use higher learning rate with no normalization, it tends to overshoot minima & convergence is much slower.

Optimizers: Hill Descent analogy in 2D :- (2.6)

→ Simple SGD in deep learning cannot solve the optimization functions properly.



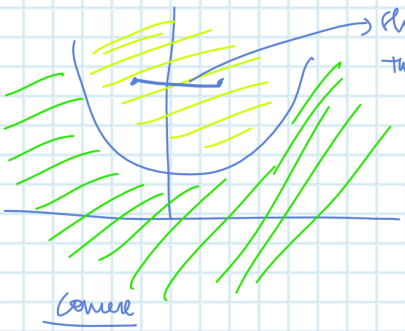


at minima & maxima and saddle point, $\frac{dL}{dw} = 0$

$$W_{new} = W_{old} - \eta \frac{dL}{dw}$$

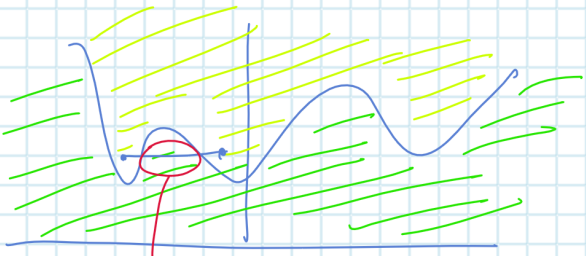
Simple SGD could get stuck at minima maxima & saddle point.

→ Convex functions :-



→ Shortest dist b/w any two points of the region. If all points on this line lie's in the same region it's a convex functions.

Convex



don't lie in the same region - ∴ Not convex.

→ Convex funcs only have one minima & 1 maxima.
local minima = global minima.
non convex have multiple local minima & 1 global minima.

→ In log reg, lin reg & SVM : all are convex funcs.

→ In deep learning : non convex functions. ie, we have local minima, saddle points etc.,
Depending on W initialization we could end up with local minima.

rajat_chaudhary — 47 Votes

Hi,the way I see this, it is upto us to choose which ever loss function we can choose for our algorithm. So is that one of the reason we use squared loss in linear regression, as the squared loss is a convex function, hence no issue of getting stuck at local minima or local maxima or a saddle point. just wanted to confirm my understanding here.

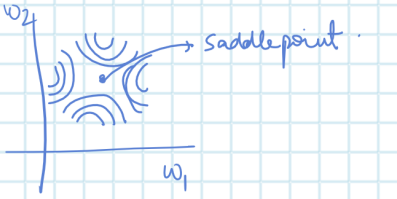
Reply — Mar 03, 2019 08:20 AM

Applied_AI

Good question especially for the interviews. You are asked why to use the squared loss and power 4 or modulus. One of the major reason is squared loss upon derivative produces a single value of parameter set and hence gives one unique solution while others don't. However it is not true for one value of the parameter, the loss would be minimal. Loss can be minimal for multiple values also. Hence you can't tell that squared loss would only give you the least solution. Just because you have multiple minima doesn't mean that they need to be different in values. Honestly in Square and power 4 both generates minimal solution but the squared loss gives one solution while power 4 gives 3 solutions.

Reply — Mar 03, 2019 10:36 AM

contour plot of a saddle point :-



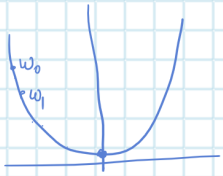
Simple mini batch SGD will get stuck at saddle point.

SGD Recap (2.8) :-

$$(w_{ij}^k)_{new} = (w_{ij}^k)_{old} - \eta \left[\frac{\partial L}{\partial w_{ij}^k}\right]_{(w_{ij}^k)_{old}}$$

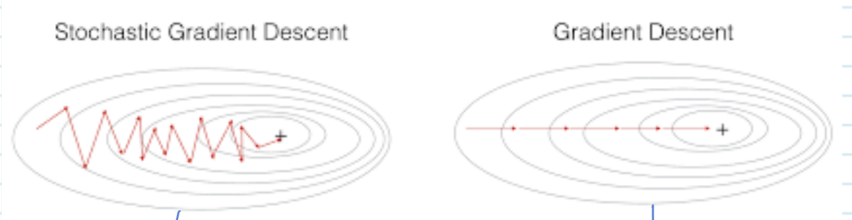
$$\Downarrow$$

$$w_t = w_{t-1} - \eta \left[\frac{\partial L}{\partial w}\right]_{w_{t-1}}$$



→ $\frac{\partial L}{\partial w}$ when calculated using all points → Gradient Descent (computationally expensive)
using one random point in each iterations → SGD
using a random subset of points each iter → MiniBatch SGD (most common in DL)

MiniBatch SGD is erroneous, using it an approximation of GD
ie) SGD with lots of iterations ≈ GD
We are estimating the gradient in SGD. So the updates in each iterations are more noisy in SGD than in GD.



Noisy SGD path.
More epochs but faster.

GD path.
Less epochs. More time.

Batch SGD with Momentum (2.9) :-

→ Assume we are calculating values

at $t=$	1	2	3
	a_1	a_2	a_3
	v_1	v_2	v_3

Let γ be a value b/w 0 & 1

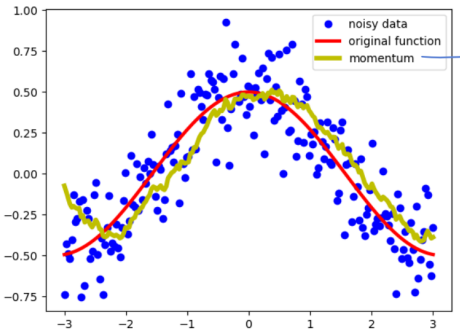
$v_1 = a_1$
$v_2 = \gamma v_1 + a_2$
$v_3 = \gamma v_2 + a_3 = \gamma(\gamma v_1 + a_2) + a_3$

if $\gamma = 1$	$\gamma = 0$	$\gamma = 0.5$
$v_1 = a_1$	$v_1 = a_1$	$v_1 = a_1$
$v_2 = a_1 + a_2$	$v_2 = a_2$	$v_2 = 0.5a_1 + a_2$
$v_3 = a_1 + a_2 + a_3$	$v_3 = a_3$	$v_3 = 0.25a_1 + 0.5a_2 + a_3$

most weightage to a_3
less to a_2
lesser to a_1 ...

$\Rightarrow v_t = \gamma v_{t-1} + a_t$

v_t at any time is called a denoised value. These are called exponentially weighted averages.



→ obtained after exponentially weighted average. Nice approximation to the original function

→ taking this concept & applying it to mini batch SGD :-

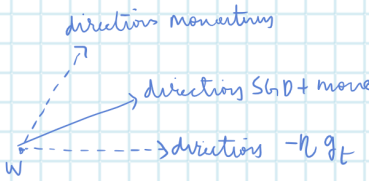
$$w_t = w_{t-1} - \eta \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$$

Let $g_t = \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$ $\Rightarrow w_t = w_{t-1} - \eta g_t$

Applying exponential weighing.

$$v_t = \gamma v_{t-1} - \eta g_t \quad (\gamma v_{t-1} : \text{momentum})$$

$$w_t = w_{t-1} - v_t$$

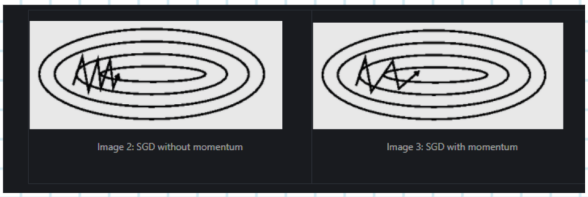


In practice typical value of $\gamma = 0.9$

denoising with exponential weighing = SGD + momentum

With momentum, SGD starts taking longer steps towards minima.

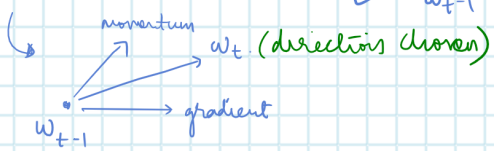
∴ SGD + momentum speeds up convergence.



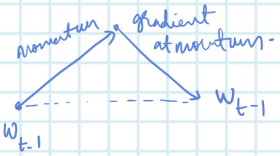
→ As we are moving closer to minima SGD points in the best direction. Therefore we use exponential average & not simple average.

Nesterov Accelerated Gradient (2.10) :-

in SGD + momentum $g_t = \left[\frac{\partial L}{\partial w}\right]_{w_{t-1}}$



But in NAG :-



$$w_t = w_{t-1} - (\gamma v_{t-1} + \eta g')$$

$$g' = \left(\frac{\partial L}{\partial w}\right)_{w_{t-1} - \gamma v_{t-1}}$$ = gradient calculated at momentum.

→ if we ever skip minima in an iteration, NAG will take lesser steps till convergence than SGD with momentum.

Adagrad (2.11) :-

→ In SGD & SGD + Momentum, learning rate (η) is constant.

features → sparse, dense } same learning rate for both is bad.

→ Adagrad = Adaptive learning rate.

SGD :- $w_t = w_{t-1} - \eta g_t$ (η → same for all weights)

Adagrad :- $w_t = w_{t-1} - \eta'_t g_t$ η' is different for each weight at each iteration

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

ϵ is a small value to prevent 0 division errors.

η is a normal learning rate value.

$\alpha_{t+1} = \sum_{i=1}^{t+1} (g_i)^2$ → always +ve ⇒ α is always +ve.

$\alpha_t > \alpha_{t-1} > \alpha_{t-2} \ldots$

as $t \uparrow$, $\alpha_t \uparrow$, $\eta'_t \downarrow$

as iterations increase, learning rate decreases adaptively.

Advantages :-

→ No need of manually tuning η

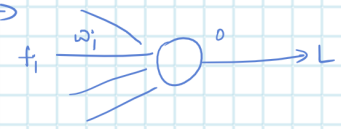
→ if there are some sparse & some dense features, adagrad automatically chooses the best learning rate.

Disadvantage :-

→ As $t \uparrow$, α_t could become very large & η' becomes very small ⇒ $w_t \approx w_{t-1}$ leads to slower convergence.

→ to avoid getting stuck at saddle points, add some value ϵ to optimization problem.

→



f_i = input to neuron with weight i

if f_i is sparse ⇒ mostly NAs

in mini batch SGD, lot of batches contain mostly NAs & only few valid values.

ex:- let loss be squared loss

$$\Rightarrow \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}} = \sum_{i=1}^{n} \underbrace{(-2x_i)}_{\text{will be 0 for sparse features}} (y_i - w_{t-1} x_i)$$

$\Rightarrow \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$ will be much smaller for sparse/random features.

∴ Adagrad will help with this.

Optimizers : Adadelta and RMSProp (2.12) :-

→ In adagrad :- if α_{t+1} becomes very large, η becomes low, convergence becomes slower.

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \epsilon}} \qquad \alpha = \sum_{i=1}^{t-1} g_i^2$$

→ In Adadelta, we replace α with exponentially decaying average so as to avoid small η values & avoid slow convergence.

$$\Rightarrow \eta'_t = \frac{\eta}{\sqrt{eda_t + \epsilon}}$$

$$eda_t = \gamma eda_{t-1} + (1-\gamma) g_t^2$$

if $\gamma = 0.95$, then $eda_t = 0.95\, eda_{t-1} + 0.05\, g_t^2 = 0.05\, g_t^2 + \left[0.95\{0.05\, g_{t-1}^2 + 0.95\, eda_{t-2}\}\right]$

$$= 0.05\, g_t^2 + (0.95 * 0.05) g_{t-1}^2 + (0.95)^2 eda_{t-2}^2$$

We are controlling the growth of α

Adam (2.13) :-

→ In AdaDelta, we are taking the eda of g_t^2.

In Adam, we take eda of g_t.

Adam = Adaptive Moment Estimation.

In Statistics : mean is aka 1st order moment (Adam more like 1st order)

variance is aka 2nd order moment (AdaDelta more like 2nd order)

mean at time t, $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ $\quad 0 \le \beta_1 \le 1$ $\longrightarrow$ eda g_t

$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ $\quad 0 \le \beta_2 \le 1$ $\longrightarrow$ eda g_t^2

β_1 is typically 0.9

β_2 is typically 0.99

α is typically 0.001

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)_t} \qquad \hat{v}_t = \frac{v_t}{1-(\beta_2)_t}$$

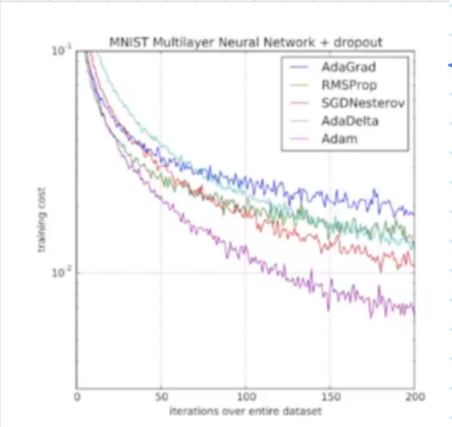
} very similar to ada delta

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{v_t} + \epsilon}$$

if $\beta_1 = 0 \Rightarrow$ Adadelta

$\beta_1 = \beta_2 = 0 \Rightarrow$ not exactly adadelta but similar.

→ Adam has all the advantages as that of AdaDelta.



→ Adam was the fastest.

→ Adam has the problem of overshooting the minima.

NADAM = NAG + Adam + RMSProp.

→ As of 2021 AdaBound an Adam variant that employs dynamic bounds on learning rates to achieve gradual & smooth transitions to SGD, is faster.

When to choose which algorithm (2.14):-

Mini Batch SGD – Momentum & NAG – AdaGrad – AdaDelta & RMSProp – Adam.

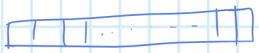
- Mini Batch SGD: good for small neural networks. Can't get out of saddle points.
- Momentum & NAG: work well in most cases. But slow.
- AdaGrad, AdaDelta & RMSProp: good for sparse data.
- Adam: Best and fastest.

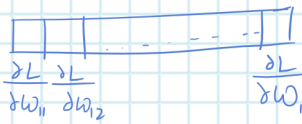
Gradient checking & Clipping (2.15):-

→ good habit to monitor weights & gradients for each epoch, weight & layer.

→ helps detect vanishing gradient / exploding gradient.

→ Solution to exploding gradient = clipping:-

$w =$ 

$G =$ 

L_2 Norm clipping: $G_{new} = \frac{G}{\|G\|_2} * z$ $\quad z$ = threshold

$\|G\|_2$ → L_2 Norm

All values of G_{new} will be $< z$

Softmax and Cross Entropy for Multi Class Classifications (2.16) :-

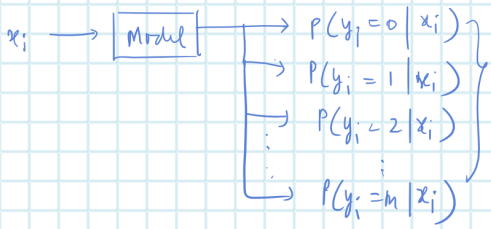
→ Logistic Regression on its own works only for binary classifications.
for multiclass → one v/s rest.
Instead of one v/s rest : log reg for multi class = softmax.

→ Logistic Regression from neural net perspective

$$z = x_i^T w$$

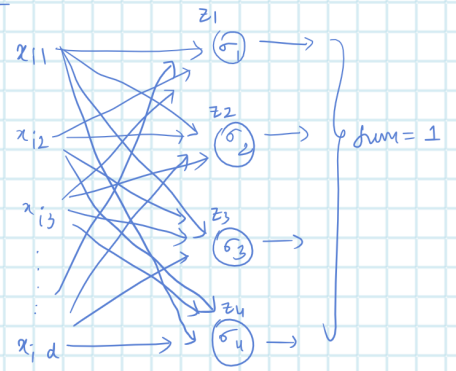
$$P(y_i = 1 | x_i) = \hat{y}_i = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z + 1}$$

→ $D = \{x_i, y_i\}$ $\quad y_i \in \{0, 1, 2, \cdots m\}$



Soft-max :-

let m = 4



$z_1, z_2, \cdots z_m$ — inputs to neurons.

$$z_1 = \sum_{j=1}^{d} x_{ij} w_{j1} \qquad z_2 = \sum_{j=1}^{d} x_{ij} w_{j2} \quad \cdots$$

$$\sigma_1(z_1) = \frac{e^{z_1}}{\sum_{i=1}^{m} e^{z_i}} \qquad \sigma_2(z_2) = \frac{e^{z_2}}{\sum_{i=1}^{m} e^{z_i}}$$

$$\sigma_1(z_1) + \sigma_2(z_2) + \cdots = \frac{e^{z_1}}{\sum_{i=1}^{m} e^{z_i}} + \frac{e^{z_2}}{\sum_{i=1}^{m} e^{z_i}} + \cdots + \frac{e^{z_m}}{\sum_{i=1}^{m} e^{z_i}} = \frac{\sum_{i=1}^{m} e^{z_i}}{\sum_{i=1}^{m} e^{z_i}} = 1 \quad \text{Sum} = 1$$

→ softmax minimizes multi class log loss aka cross-entropy

$$-\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{K} y_{ij} \log p_{ij}$$

$y_{ij} = 1$ if $x_{ij} \in y_{ij}$ else otherwise $\quad p_{ij}$ = prob of belonging to y_{ij}.

How to train MLP (2.17) :-

① Preprocess : Data Normalization.

② weight initialization : (i) Xavier / Glorot → Sigmoid / tanh
(ii) He → ReLU
(iii) Gaussian → Small σ

③ Choose activation function :- ReLU as of now

④ Perform Batch Normalization (especially for deep MLP (later layers))
Dropout is compulsory.

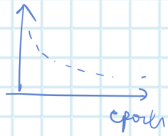
⑤ Optimizer : Adam (best one rn)

⑥ hyperparameters : lots of hyperparams in DL (#hyperas if you have GPU)

⑦ Loss fn → 2 class → log loss
K class → multiclass log loss
regression → square loss

⑧ Monitor Gradients - Clipping

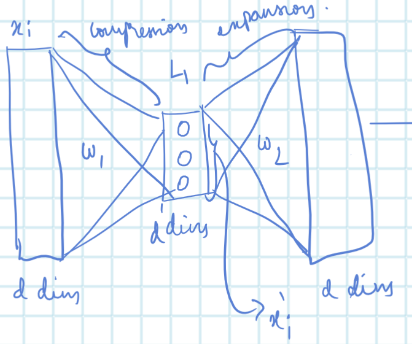
⑨ plots -



⑩ Avoid overfitting - Easy to overfit in deep learning because lots of weights.

Autoencoder :- (2.18)

→ Neural network which can perform dim-red.

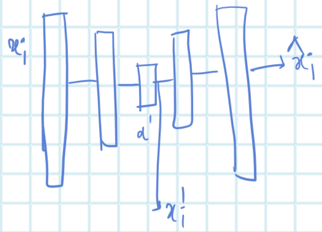


if the d' outputs from L_1 i.e, $\hat{x}_i$ accurately represent x_i
then x_i encodes all of the info of x_i

$$L(x_i, \hat{x}_i) = \|x_i - \hat{x}_i\|^2$$

if $x_i == \hat{x}_i$, $L = 0$.

→ just like MLP, we can have deep autoencoders i.e, multiple hidden layers



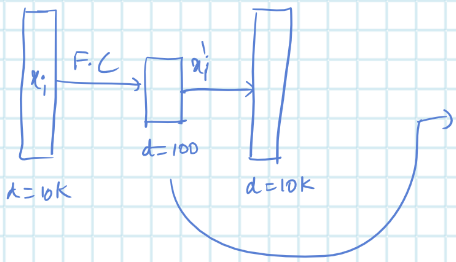
Denoising Autoencoder :-

$x_i = \{\tilde{x}_1, \ldots \tilde{x}_n\}$ → noisy dataset

AutoEncoder $(x_i) = \tilde{x}_i$ → noise free, robust. Because it reduces the number of dimensions
$\quad R^d \quad R^{d'}$

→ if original dataset is already noise free, we add noise to it ourselves and denoise it. Because test data points may not be noise free. Model can learn this.

Sparse Auto Encoder :-



→ if we want this to be sparse, we can use L_1 reg to the x_i'.

→ If there's only one hidden layer that only uses sigmoid act. functions, the AE is strongly related to PCA.

→ AE can also be seen as unsupervised feature extraction process.

ex :- MNIST 784 → 50 dimensions features.

Word2Vec : CBOW (2.19) :-

→ Not a neural network.

→ The cat sat on the wall.

if 'sat' is focus word
these are context words.

→ word2vec uses 2 algos

(i) CBOW :- Continuous bag of words.

(ii) Skipgram.

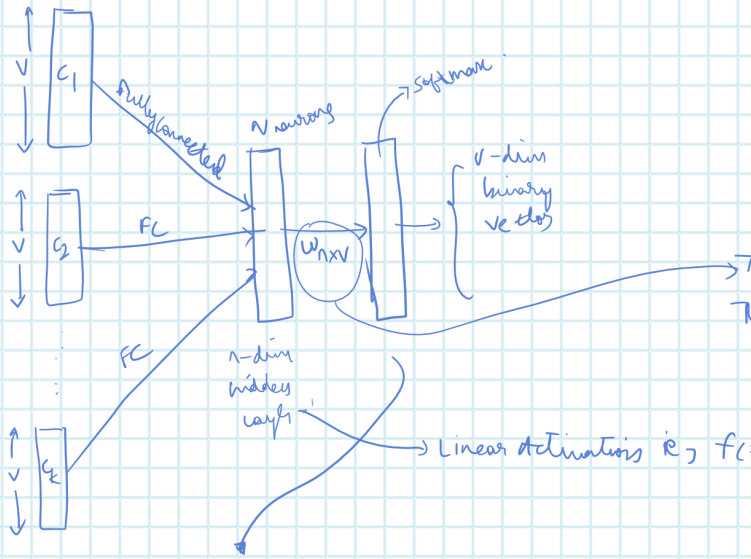
CBOW :-

① dictionary / vocab of words = V

v = length / size of vocab

② one hot encoding each word

$w_i \in \mathbb{R}^V$



This vector is the vector representation that contains all words info

This is the vector returned by word2vec.

Linear activation i.e, $f(z) = z$.

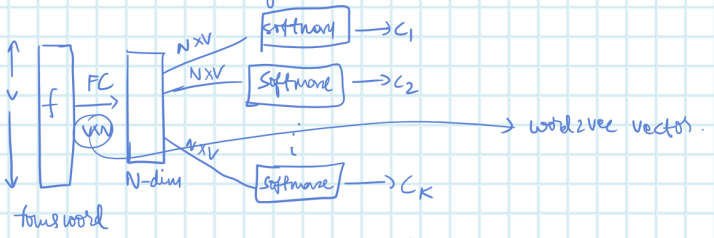
core idea : predict focus word using context words -

⇒ Multi class classification problem.

→ training is done by taking all possible combinations of focus words & context words -

word2vec (skipgram) (2.20) :-

→ predict context words given focus words.



K - Multi class classifiers.

→ CBOW : 1 softmax

Skipgram : K softmax

∴ skipgram is computationally more expensive.

→ CBOW → faster than skipgram

→ better for frequently occuring words

skipgram → can work with smaller amounts of data.

→ works well for infrequent words.

K = # of context words.

K ↑ ⇒ more context ⇒ N-dim vector for each point is better.

N is typically 200/300.

→ # of weights to be trained is large in both $\approx (K+1)(N \times V)$

So we need to find algorithmic optimizations.

word2vec (Algorithmic Optimizations) (2.21) :-

→ lots of weights to be trained.

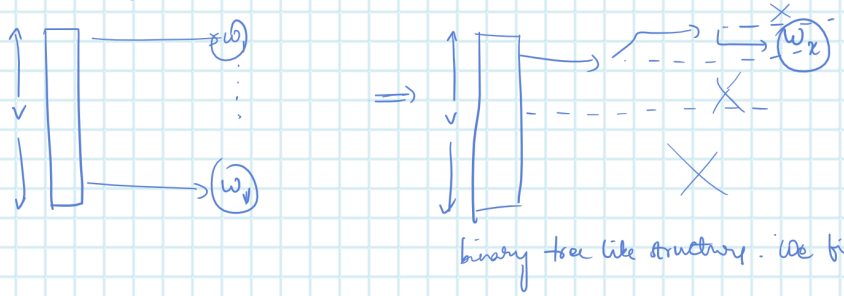
→ 2 optimizations

(i) Hierarchical softmax (algo)

(ii) Negative sampling (statistical)

Hierarchical :-

→ idea is to modify V-softmaxes to make it optimal (skipgram)

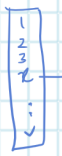


binary tree like structure. We find correct halt by backpropagation as correct half results in less loss.

$= \log(V)$ softmax classifiers required.

Negative Sampling :-

→ update only a sample of words per iterations



→ target word (keep this, sample rest).

↳ always keep the target

↳ sample non target words with $P(w_i) = \sqrt{\frac{z}{freq(w_i)}}$, $z = 10^{-5}$