Deep Learning

History of Neural Networks

- Perceptron (1957) [Roosenblat]. {loosely inspired by biological neuron}.
- Then later realized brain is a network of such neurons.

  so, came up with

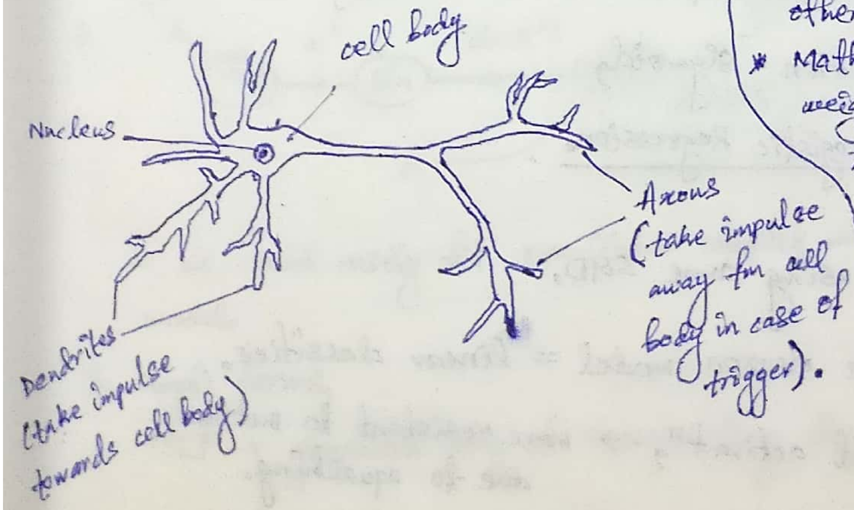
- Neural Networks (1986) [Hinton]. { Backpropagation to train }.

  $\Downarrow$

  chain rule only

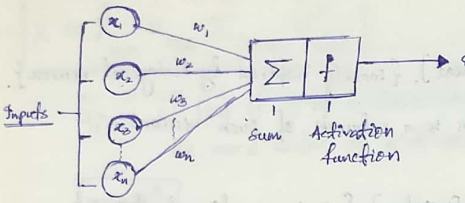
- Due to lack of computation power, data & algorithmic advancements, (had problems training deep N/w). NNs couldn't take off in 90s.
- Breakthrough - 2012 - ImageNet dataset. - Deep NN beat other models by a huge margin.
- First, we'll learn classical NN. & their limitations.
- Then, Deep NN, CNN, RNN & LSTMs.

How biological neuron works


cell body
Nucleus
Axons (take impulse away fm cell body in case of trigger).
Dendrites (take impulse towards cell body)

- Some dendrites are thicker than others.
- Mathematically, we assign weights to inputs to represen thickness.
- A neuron is said to be fired/ activated if it gets enough i/p.

corresponding AN is designed as —



$$\text{output} = f(w_1x_1 + w_2x_2 + \cdots \quad w_nx_n)$$

$$\boxed{o = f\left(\sum_{i=1}^{n} w_ix_i\right)}$$

for perceptron, (the oldest design of Artificial Neuron), which was modelled around biological neuron had activation function 'f'

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

If we recall Logistic Regression, the model ~~way~~ was

$$\boxed{\hat{y} = \sigma(w^Tx + b)}$$

$$\boxed{\hat{y} = \sigma\left(\sum_{i=1}^{d} w_ix_i + b\right)}$$

* So, if we add 1 more constant i/p '1' & weight 'b', and activation function = sigmoid, then Neuron = Logistic Regression.

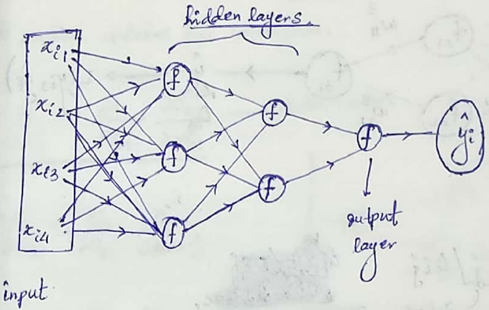
* Training can be done using same SGD.

* So, basically a single neuron model = linear classifier.
  - ↳ If sigmoid activatn, → more resistant to outlier due to squashing.
  - → perceptron → prone to outliers.

Multi-layered perceptrons

Perceptron & logistic regression → single neuron model.

Later people came up with network of neurons.

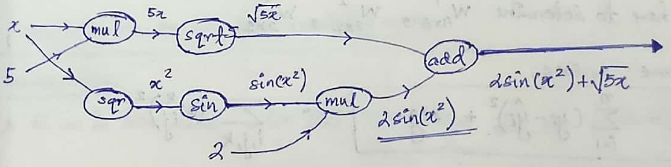


why should we came about MLP?

→ ⓐ biological inspiration →
from Neuroscience study, on humans, rats, monkeys, they found network of neurons.

ⓑ mathematical inspiration →
more the depth, more complex functions we can fit.

Ex → say $f(x) = 2\sin(x^2) + \sqrt{(x \times 5)}$



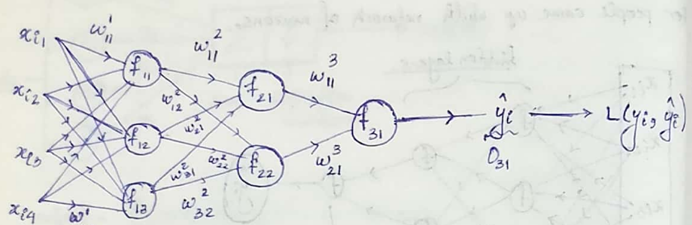
So, we could easily fit non-linear complex model due to depth of model.

In simple terms,
MLP → graphical way of representing fog, gof, etc.

* But overfits very easily.

Notations

$\mathcal{D} = \{x_i, y_i\}$; $x_i \in R^n$; $y_i \in R$ (regression).



$x_{\underbrace{i}_{\text{data pt}}\underbrace{j}_{\text{feature}}}$

$f_{\underbrace{i}_{\text{layer}}\underbrace{j}_{\text{index}}} / O_{ij}$

$w^{\overset{\text{layer}}{k}}_{\underbrace{i}_{\text{from index}}\underbrace{j}_{\text{to index.}}}$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}_{4\times3}$$

$w^1 - \underbrace{4}_{\text{prev layer neurons.}} \times \underbrace{3}_{\text{next layer neurons.}}$

Training an MLP

- Refer to the diagram above.
- We have to determine $W^1_{4\times3}$, $W^2_{3\times2}$, $W^3_{2\times1}$
- Assume sq. loss.

$$\boxed{L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{reg.}} \longrightarrow \sum_{i,j,k} (w^k_{ij})^2$$

optimizatⁿ – $\boxed{\min_{w^k_{ij}} L}$

SGD

ⓐ initialize w^k_{ij} randomly {later will see details of diff initializatⁿ}.

ⓑ $(w^k_{ij})_{new} = (w^k_{ij})_{old} - \eta \left(\dfrac{\partial L}{\partial w^k_{ij}} \right)$ — calculating this is the crux.

ⓒ perform updates till convergence.

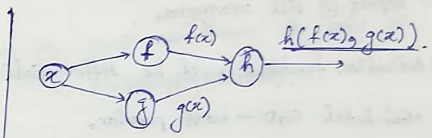
$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w^3_{11}}$$ — chain-rule.

$$\frac{\partial L}{\partial w^3_{21}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w^3_{21}}$$

$$\frac{\partial L}{\partial w^2_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w^2_{11}}$$

$$\frac{\partial L}{\partial w^2_{21}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w^2_{21}}$$

$$\frac{\partial L}{\partial w^2_{12}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial w^2_{12}}$$



$$\boxed{\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}}$$

special case of chain rule.

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial O_{31}} \left(\frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w^1_{11}} + \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w^1_{11}} \right)$$

similarly we can calculate remaining ones.

Memoization

- As can be seen above, calcⁿ like $\frac{\partial L}{\partial O_{31}}$, $\frac{\partial O_{31}}{\partial O_{21}}$ etc are repeatedly required.
- So, its good idea to compute once, store & re-use. — Memoization.
- This consumes some memory, but speeds up drastically.

So, the full algo is—

① initialize w_{ij}^{k}

② for each x_i in D

ⓐ pass x_i forward through the network

ⓑ compute $L(\hat{y}_i, y_i)$.

ⓒ compute all the derivative using chain rule and memoization.

ⓓ update weights.

③ Repeat ② till convergence.

* Activation functions must be differentiable. & should be easy & fast to diff.

* Mini-batch SGD — most popular.

Activation functions

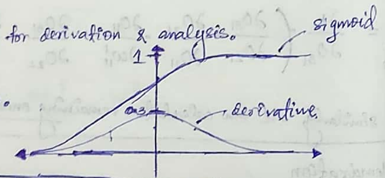
Pre-deep learning era → sigmoid, tanh.

$$\text{sigmoid } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\boxed{\frac{\partial\sigma}{\partial z} = \sigma(z)(1-\sigma(z))}$$ → The derivative of $\sigma(z)$ can be represented in terms of $\sigma(z)$ itself.

Refer → "ronny.rest/blog/" — for derivation & analysis.

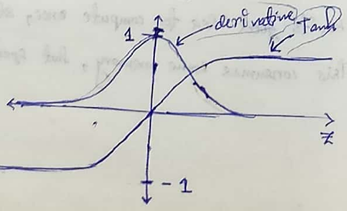
* $$\boxed{0 \leq \frac{\partial\sigma}{\partial z} < 1}$$ Imp.



$$\text{Tanh}(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

$$\boxed{\frac{\partial(\tanh(z))}{\partial z} = 1-\tanh^2(z)}$$

$$\boxed{0 \leq \frac{\partial \tanh}{\partial z} \leq 1}$$



* Both have similar shapes.

* More popular now — ReLU

Vanishing gradients

Refering to the earlier NN,

$$\frac{\partial L}{\partial w_{11}^{1}} = \frac{\partial L}{\partial O_{31}}\left[\underbrace{\frac{\partial O_{31}}{\partial O_{21}}}\cdot\frac{\partial O_{21}}{\partial O_{11}}\cdot\frac{\partial O_{11}}{\partial w_{11}^{1}} + \frac{\partial O_{31}}{\partial O_{22}}\cdot\frac{\partial O_{22}}{\partial O_{11}}\cdot\frac{\partial O_{11}}{\partial w_{11}^{1}}\right]$$

$$\downarrow$$

$$\left(\frac{\partial f_{31}}{\partial O_{21}}\right) \text{ which is } \underbrace{\left(\frac{\partial\sigma}{\partial z}\right)_{z=z_0}} \text{ for some } z_0.$$

So, all such derivative terms $\in \underline{(0,1)}$

* Repetitive multiplication makes $\frac{\partial L}{\partial w_{11}^{1}}$ small and hence the rate of convergence decreases.
  ↓
  (vanishing gradients).

* Similarly if we have $\left(\frac{\partial f}{\partial z}\right) > 1$, then we can have situation of exploding gradiants.

Bias-variance trade-off

① #layers ↑ ⇒ more weights ⇒ higher chance of overfitting ⇒ high variance.

Regularization → L₁ or L₂.

$$L = \sum_{i=1}^{n} loss_i + \lambda L_2 reg.$$

λ ↑ ⇒ underfit

λ ↓ ⇒ overfit.

2 hyper-parameters — #layers ✓
λ ✓

Play @ playground.tensorflow.org

Deep MLP

1980s – 2006 ⟶ 2-3 layered network because.

① vanishing gradient

② too little data. → easily overfit

③ too little computation power.

2010 → lots of data → internet.

powerful computations → GPUs

New ideas → algos.

Classical ML (SVM, RF) → Build great mathematical theory and expt. done to justify theory.
(Mathematician way).

* Building new theory – v. hard.
* Performing expts given advanced computers – easier.

Modern DL → conduct a lot of expt & see what works.
(Engineers way) once successful, build theory for them.

* In the end what matters is 'Results" in industry.

Dropout and Regularization

Deep NN → overfitting major issue → Regularizat" used to handle.

Drawing parallels with Random forest

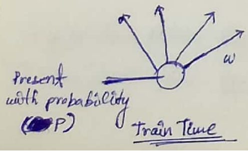
Random forest → samples a random subset of features.
uses randomization for regularization. (RFR).

Dropout makes use of RFR for NN.

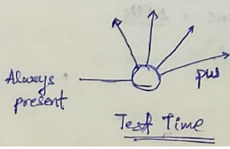
* In each layer, sample a bunch of neurons randomly and remove all incoming & outgoing edges.
* The dropout rate (P) determines the %age of neurons to be removed in each layer.

$P = 0.2 \Rightarrow$ 20% of neurons in each layer ~~removed~~ present.

* This removal is valid only for that iteration.
* For each neuron, neurons in prev. layer act as i/p features.
So, dropout ≃ random subset of features in RF (which reduces variance).



Train Time



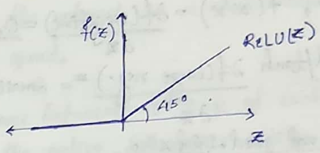
Test Time

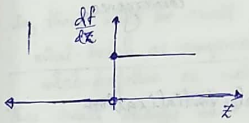
ReLU

Classical NN – vanishing gradient (sigmoid or tanh)
└ convergence slows down.

- As a result, can't train deep NN.
- ReLU → best activation function as of 2018

$$\text{ReLU}(z) = \begin{cases} z, & z > 0 \\ 0, & z \le 0 \end{cases}$$



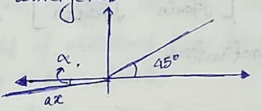


Not differentiable at 0.

$$\frac{\partial f_{ReLU}}{\partial z} \in \{0, 1\}$$ So, no problem of vanishing gradient.
Hence, has faster convergence.

But, with ReLU, we can have dead activations.
i.e. if any one derivative becomes 0, the whole derivate chain product = 0
Hence, no further convergence.

Sol" → leaky ReLU.



$\alpha \to 0$

$$\frac{\partial f}{\partial z} = \begin{cases} 1, & z > 0 \\ a, & z < 0 \end{cases}$$

$a = 0.01$ (v. small).

*. But again vanishing gradient can occur for large negative weights.
* weights must be initialized carefully.

Weight Initialization

For logistic regression → uniform random
→ $w_{ij}^k \in N(0, \sigma)$ $\sigma \to 0$

for deep MLP

① $w_{ij}^k = 0 \; \forall i, j, k.$ → v.v. bad.
Since started equal, all will have same gradient & will remain same after updates as well.
- All neurons compute the same thing basically.

* We can think of deep MLP as stacking ensemble of neurons. so, more different each one is, the better. will be output.

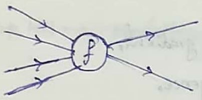
② w_{ij}^{k} = large -ve number. { x is normalized always }.

* for ReLU, $f(w^T x) = \frac{\partial f(\text{negative})}{\partial z} = \underline{0}.$ So, no convergence.

* for sigmoid/tanh $\frac{\partial f(\text{large neg.})}{\partial z}$ = small gradient. hence slow convergence.

Hence, not good. initz^n.

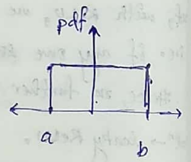
③ $W_{ij}^{k} \sim N(0, \sigma)$ σ: small. Gaussian/Normal Initialization.



fan_in = 4
fan_out = 2.

④ uniform init -

$$w_{ij}^{k} \sim \text{uniform}\left[\overset{a}{\frac{-1}{\sqrt{\text{fanin}}}}, \overset{b}{\frac{1}{\sqrt{\text{fanin}}}}\right]$$



works well for sigmoid activation.

⑤ Xavier/Glorot Init (2010)

ⓐ Normal - $w_{ij}^{k} \sim N(0, \sigma_{ij})$ $\sigma_{ij} = \frac{2}{\text{fan\_in} + \text{fan\_out}}$

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

works well for sigmoid.

⑥ He Initialization (2015)

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

ⓑ uniform $W_{ij}^{k} \sim U\left[-\sqrt{\frac{6}{\text{fan\_in}}}, \sqrt{\frac{6}{\text{fan\_in}}}\right]$ works well with ReLU.

Batch Normalization

$D = \{x_i, y_i\}$. As a part of pre-processing, we perform data -normalization. It includes
① mean-centering.
② variance scaling.

$$\left\{ \tilde{x}_i = \frac{x_i - \mu}{\sigma} \right\}$$

μ = mean(x_i)
σ = stddev(x_i)

* Our i/p to the network is a mini-batch for optimizing speed.

So, there will be small difference between σ & μ of each mini-batch as compared to the entire dataset's σ & μ. & other mini-batches as well.

This small change will become huge & significant deep down the network.

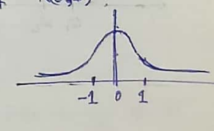
So, for a neuron deep in the network, 2 datapoints of different mini-batche will appear to be coming from entirely different distributions, hampering the performance.

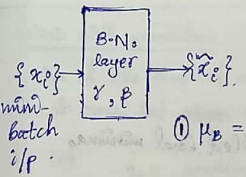
* This problem is called internal co-variance shift.

Sol^n:- After output from a layer, normalize those batch of points.

It can be thought of additional layer being added, which must be added atleast deep in the network.

So, all pts appear to be coming from same distb^n. of N(0,1).





γ & β are earnt via Back-propagation.

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

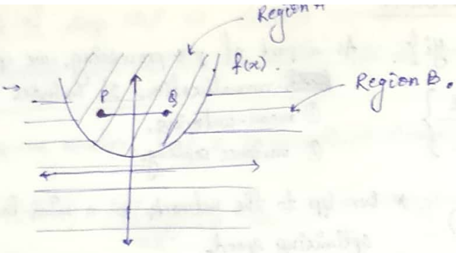
② $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ // mini-batch variance.

③ $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ // normalization

④ $\tilde{x}_i = \gamma \hat{x}_i + \beta$ // scale & shift.

Optimization

convex function →



The curve f(x) divides the plane into Region A & Region B.

* If we select any 2 pts P & Q in the inner region, line joining them will lie entirely in that region. - property of convex function.

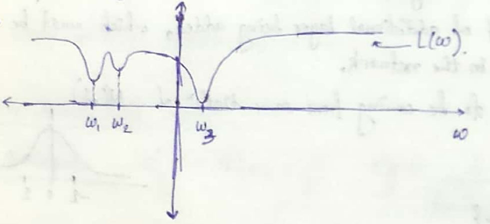
Property of convex fun → They have one minima/maxima.

Logistic Regression, Linear Regression, SVM — convex loss function.

But Deep MLP - non-convex functions.

They can have more than 1 maxima/minima.

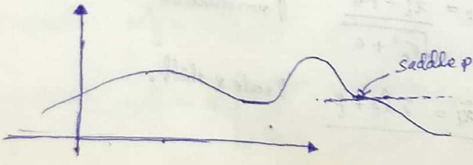
ex →



3 minimas - at ω_1, ω_2 & ω_3.

ω_3 is called global minima & all 3 are called local minima.

At minima/maxima, $\frac{\partial f}{\partial x} = 0$. But $\frac{\partial f}{\partial x} = 0$ doesn't imply maxima/minima.



saddle point. $\frac{\partial f}{\partial x} = 0$, but neither maxima/minima

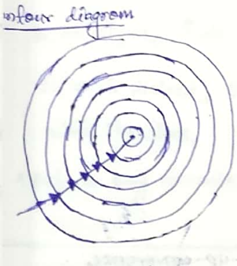
So, SGD can get stuck at saddle point.

Hence, simple SGD/ mini-batch SGD isn't good enough to train MLP.

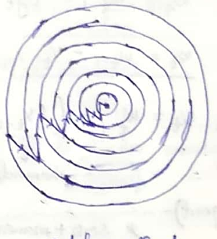
* $\left(\frac{\partial L}{\partial w}\right)_{\substack{\text{mini-batch}\\ \text{SGD}}} \approx \left(\frac{\partial L}{\partial w}\right)_{GD}$.

↓ noisy

contour diagram



Gradient Descent



Batch gradient descent

* Since $\left(\frac{\partial L}{\partial w}\right)$ is noisy, updates are erroneous a little bit.

Gradient descent will converge in fewer iterations.

If we run a lot of iterations,

w^*_{SGD} mini-batch	$\approx$	w^*_{GD}
20		5

So, we need to come up with techniques to denoise $\left(\frac{\partial L}{\partial w}\right)$.

SGD with momentum

One of the simplest method to denoise would be to take average so as to cancel out noise.

Let $g_t =$ gradient at t^{th} step.

We define
$$V_0 = g_0$$
$$V_1 = \delta V_0 + g_1$$
$$V_2 = \delta V_1 + g_2$$
$$\boxed{V_t = \delta V_{t-1} + g_t}$$

$0 \le \delta \le 1$.

If we expand v_t,

$$v_t = g_t + \delta\, g_{t-1} + \delta^2 g_{t-2} + \delta^3 g_{t-3} + \cdots \delta^t g_0$$

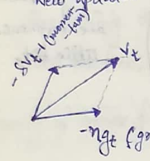
since $0 \le \delta \le 1$, $\underline{1 \ge \delta \ge \delta^2 \ge \delta^3 \cdots \delta^t}$.

* So, v_t does a weighted sum of previous gradient values, giving more weight to recent ones.

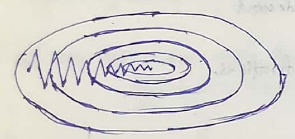
Old update: $w_t = w_{t-1} - \eta g_t$

New update: $\underline{w_t = w_{t-1} - v_t}$

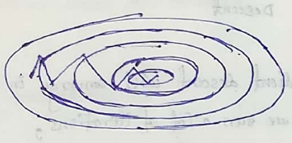
$$w_t = w_{t-1} - (\underbrace{\delta v_{t-1}}_{\text{momentum}} + \underbrace{\eta g_t}_{\text{gradient}})$$



* SGD + momentum = speeds-up convergence.



SGD

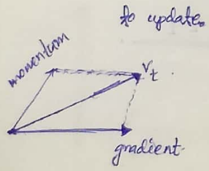


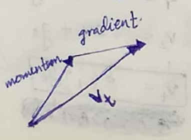
SGD + Momentum.

Nestrov Accelerated Gradient (NAG).

(SGD + Momentum) → updates weight by calculating resultant of momentum and gradient at the current position & using the resultant to update.

(NAG) → goes along momentum at current position, calculate the gradient there. Now, the resultant of momentum & gradient used to update.





$\underline{v_t = (\delta v_{t-1} + \eta g')}$ $\quad g' = \left(\frac{\partial L}{\partial w}\right)_{(\text{at } w_{t-1} - \delta v_{t-1})}$

Adagrad

It varies the learning rate with time.

$\boxed{w_t = w_{t-1} - \eta_t g_t}$ $\quad \eta_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \epsilon}}$ — small (+)ve.

So, as $t\uparrow$, $\alpha_{t-1}\uparrow \Rightarrow \eta_t \downarrow$ $\quad \underline{\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2}$ — $\left(\frac{\partial L}{\partial w}\right)$

So, as is desired, learning rate $(\eta)\downarrow$ decreases with iterations.

* But it may decrease too much so as to $\underline{\text{hinder}}$ convergence. (slow).

Adadelta and RMSProp

* The problem with Adagrad was α_{t-1} become large due to $\sum g_i^2$. causing η to decrease very much.

* Instead of taking summation of prev. grads, we take exponential avg of grad. — Adadelta.

$$w_t = w_{t-1} - \eta'_t g_t \qquad \eta'_t = \frac{\eta}{\sqrt{eda_{t-1} + \epsilon}}$$

* Taking exponential decaying average controls growth of the denominator which was happening in Adagrad. ← $\boxed{eda_{t-1} = \delta\, eda_{t-2} + (1-\delta)\, g_{t-1}^2}$

Adam (Adaptive Moment Estimation)

· Fastest convergence algo till date.

In statistics, mean → 1st order moment.
var → 2nd order moment.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad \underline{0 \le \beta_1, \beta_2 \le 1}$$

$$v_t = \beta_2 m_{t-2} + (1-\beta_2) g_t^2$$

$$\boxed{\hat{m}_t = \frac{m_t}{1-(\beta_1)^t}} \; ; \; \boxed{\hat{v}_t = \frac{v_t}{1-(\beta_2)^t}}$$

Scanned by CamScanner

$$\boxed{W_t = W_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}}$$

α, β_1 & β_2 – hyperparameters.

Gradient clipping

Monitor gradients of each weight.

If they become too large,

$$\boxed{G_{new} = \frac{G}{\|G\|} * \tau}$$ ← threshold

L2-norm clipping.

Softmax classifier

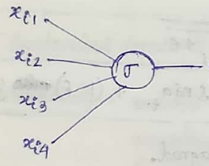
Logistic Regression → binary classification.

For making multi-class classification, – One Vs Rest.

But to naturally perform multi-class classification, we use softmax. (extension of L.R.).

Linear Regression

$y_i \in \{0, 1\}$.

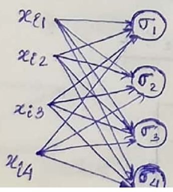


$$\hat{y}_i = P(y_i = 1 | x_i) = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$$

where $z = w^T x$

Softmax Layer

$y_i \in \{0, 1, \dots k\}$. say $k=4$.



$$z_1 = \sum_{j=1}^{d} x_{ij} w_{j1}.$$

$$z_2 = \sum_{j=1}^{d} x_{ij} w_{j2}$$

* Similarly others

$$\boxed{\sigma_1 = \frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}} \quad \boxed{\sigma_2 = \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}}$$ & so on.

* Softmax minimizes multi-class log-loss. (cross-entropy).

$$\boxed{L = \frac{-1}{N} \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log(p_{ij})}$$

$$y_{ij} = \begin{cases} 1 & \text{if } y_i \in \text{class } j. \\ 0 & \text{otherwise} \end{cases}$$

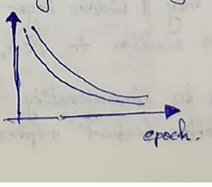
$$p_{ij} = P(y_i \in c_j | x_i)$$

* So if our problem is a 2-class classification, o/p layer is sigmoid unit.

If k-class classification, o/p layer = softmax layer.

How to train an MLP.

① pre-process → data normalization.
② weight init → Xavier/Glorot → sigmoid/tanh.
He → ReLU.
③ choose activation function. – ReLU best.
④ Batch Normalization → esp. for deep MLP. (later layers).
⑤ dropout ✓
⑥ optimizer – Adam (2018 – fastest).
⑦ hyperparams – Architecture – #layers.
#neurons in each layer.
- dropout rate (p)
Adam – α, β_1, β_2.
⑧ loss function – 2-class classification → log-loss.
k-class classification → multiclass log-loss.
regression → sq. loss.
⑨ monitor gradients & perform clipping if necessary.
⑩ plot train-loss & test-loss.
⑪ Avoid overfitting.



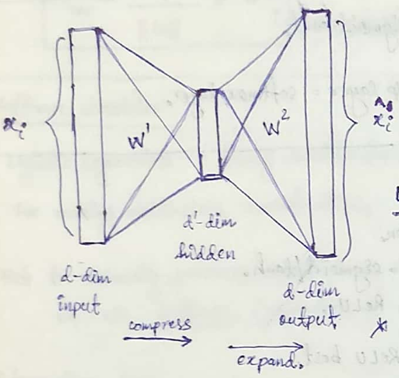
Auto-encoders

* NN which performs dim-red^n

* Given: $\mathcal{D} = \{x_i\}_{i=1}^{n}$ $x_i \in R^d$

we have to get $\mathcal{D}' = \{x_i'\}_{i=1}^{n}$ $x_i' \in R^{d'}$ $d' < d$

* The goal is to map to low dimensional space in such a way that it preserves information.



Autoencoder with 1 hidden layer.

o/p of hidden layer = $\bar{x}_i'$

$\text{LOSS} = \|x_i - \hat{x}_i\|^2$

Ideal case $x_i - \hat{x}_i = 0$

* x_i' is the compressed representation in d' dimens^n.

wiki - Auto-encoders. (diagram).
stanford.edu - Autoencoders.

* Just like MLP, we can have deep autoencoders.

* Suppose the $\hat{x}_i$ we get from the autoencoder is noisy, Then we try to build an encoder on top of this noisy data. So as to make the encoder learn to remove noise.

* Hence, usually

$$x_i \longrightarrow \tilde{x}_i = x_i + N(0, \sigma). \text{ create noisy datapts.}$$

Then encode $\tilde{x}_i$ using NN. This way, they'll be robust to noise.

* If we have only 1 hidden layer with sigmoid units, then AutoEncoder is similar to PCA.

* In addition to dimensionality red^n, AE are also used to find better feature representation of data in unsupervised way.

word2vec

word2vec(word) ⟶ vector. (These vectors are ~~symantically~~ similar for similar words). symantically

$\underbrace{\text{The cat}}_{\text{context}}$ (sat) $\underbrace{\text{on the wall.}}_{\text{context}}$
↑
focus word.

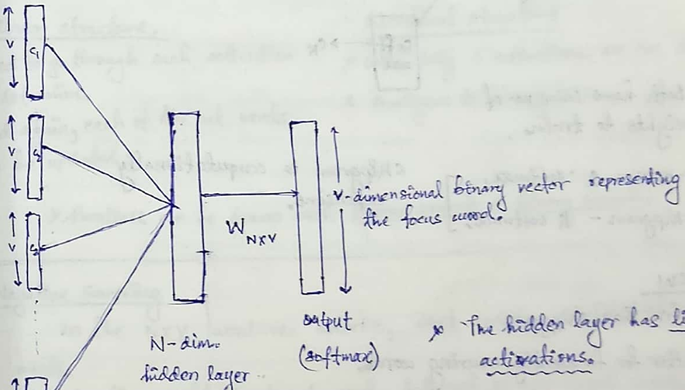
core-idea: context words are very useful in understanding focus word and vice versa.

2 algorithms for constructing word2vec
- CBOW (continous bag of words).
- skipgram.

CBOW

V = dictionary / vocabulary of words.
v = length / size of vocals.
We use 1-hot encoding to represent each word.

core-idea - given context words, predict the focus word. (multi-class classification).



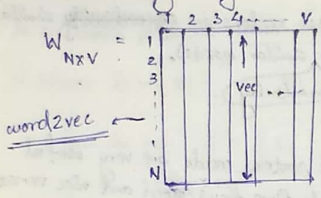
* The hidden layer has linear activations.

training dataset

focus	context
sat	{ the, cat, on, the, wall }
cat	{ the, sat, on, the wall }

similarly, create the dataset for training.

* After training using the created dataset, we'll have weights.

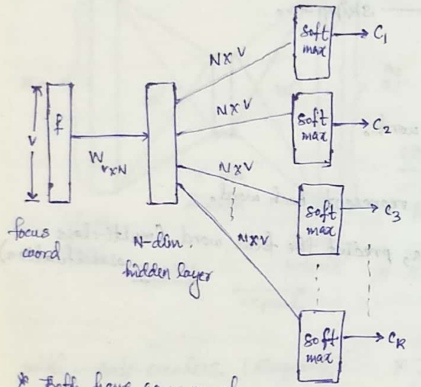


So, corresponding to each word we have a column vector of N-dimension.

$\boxed{w_i \rightarrow vec_i}$

Skip-gram

core-idea → predict context words given focus words.



Output layer :- k-softmax units.

for predicting k-context words.

Total weights :

$= k(N\times V) + V\times N$

$= \underline{(k+1)\times(N\times V)}$.

* Both have same no. of weights to train.

* C-BOW - 1 softmax.
skipgram - k softmax.

skipgram is computationally expensive.

C-BOW

* faster than skipgram.
* better for frequently occuring words.

Skipgram

* can work with smaller amount of data.
* well for infrequently occuring words.

$k\uparrow$ (#context words) ⟹ N-dim representatⁿ for each word better.

$W_{V\times N}$ - rows are the word vectors of N-dimension.

$\#weights = (N\times V)(k+1)$ N=200

$= (200\times 10k)(5+1)$ k=5

$= (2000k)(6)$ V=10k

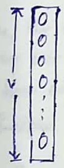
$=$ 12M weights. — huge number.

So, we need to come up with good algorithmic optimization.

) Hierarchical softmax

* say $V = 8$. (total no of words).

Earlier

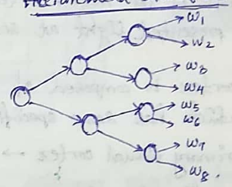


linear structure.

Need to go through each activation to determine.

And again, each of the wt needs to be updated.,

Hierarchical structure



hierchical structure

* with only 3 activations, we can determin

* Analogous to binary tree.

* Parallels can be drawn with linear search & binary search.

) Negative Sampling

In the N×V word2vec matrix, don't update weights of all words.

* update the weight of target word, but out of non-target words, sample a few

$$\boxed{P(w_i) = 1 - \sqrt{\frac{z}{freq(w_i)}}}$$

probability of sampling w_i for updating.

blog.acolyer.org - "The Amazing power of word vectors".