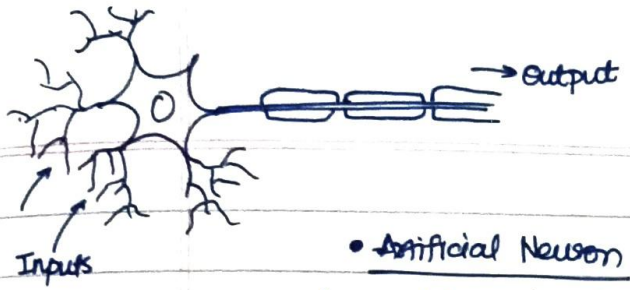
#DEEP LEARNING



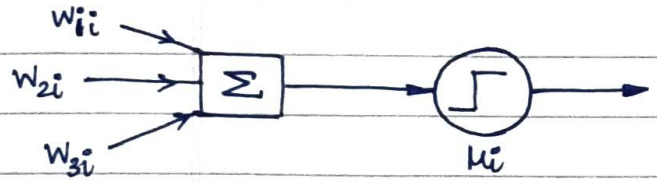
Vast Amt of parellelism in brain.

Output spike is roughly dependent on total strength of inputs that enter a neuron. Whether I give output / not depends on a threshold.

- Artificial Neuron –

1. m binary inputs (-1 or 1), 1 output (-1 or 1)
2. Synaptic Weights (w_{ji})
3. Threshold μ_i

Schematic Representation of Neuron.



$$v_i = \Theta\left(\sum_j w_{ji} u_j - \mu_i\right)$$

$$\Theta(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases}$$

A neuron would be similar to function from all the inputs to single output which could be 1 or -1.

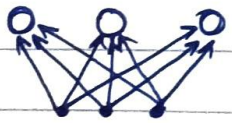
Generally before learning parameters we learn the structure. In case of single artificial neuron; the parameters are the weights & the threshold.

The sum function here is a linear function but as soon as threshold enters the picture (we say that o/p 1 if sum ≥ threshold 'else -1'); linearity is lost & non-linearity comes up.

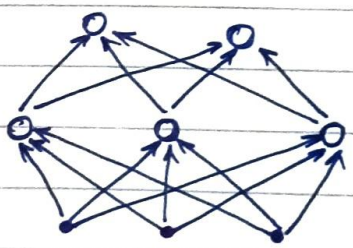
⟶ This single unit (artificial neuron) is termed as a "Perceptron".

Interconnection of multiple perceptrons builds a neural network.

- Fancy name for a type of layered 'feed forward' networks (with no loops) Uses artificial neurons with binary inputs & outputs.



Single Layer



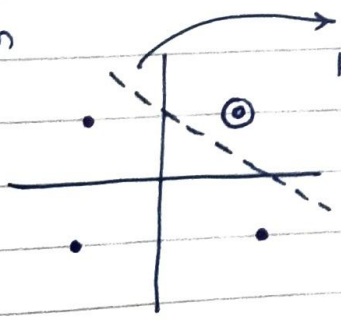
Multi layer

LINEAR SEPARABILITY –

If data points in the problem are binary, we say they are separable if there's a classifier whose decision boundary separates the positive objects from the negative ones. If such a boundary is linear function / line / flat hyperplane – it is said to be linearly separable.

(Eg) AND function

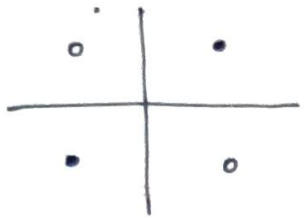
u_1	u_2	AND
-1	-1	-1
-1	1	-1
+1	-1	-1
1	1	1



⇒ Similarly for OR, NOT

(Eg) XOR function

x_1	x_2	XOR
-1	-1	-1
1	1	-1
-1	1	1
1	-1	1



We cannot separate +1 outputs from -1 because they aren't linearly separable.

↳ It means there does not exist a linear surface – a line / hyperplane (in higher dimensional space) which can separate the +ve class from the -ve class.

Minsky & Papert's : Perceptron with a threshold unit fails if classification task is not linearly separable.

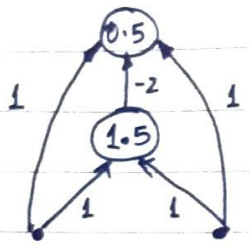
HOW TO DEAL WITH LINEAR INSEPARABILITY?

→ Multilayer Perceptron.

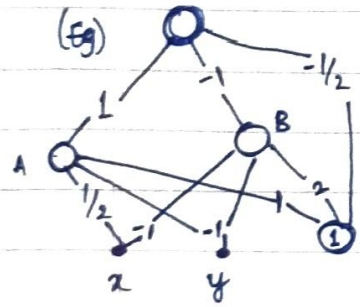
- Removes the limitations of single layer networks. Can solve XOR.

(Eg) Two layer perceptron that computes XOR.

Output is +1 iff $x+y-2\,\Theta(x+y-1.5)-0.5>0$

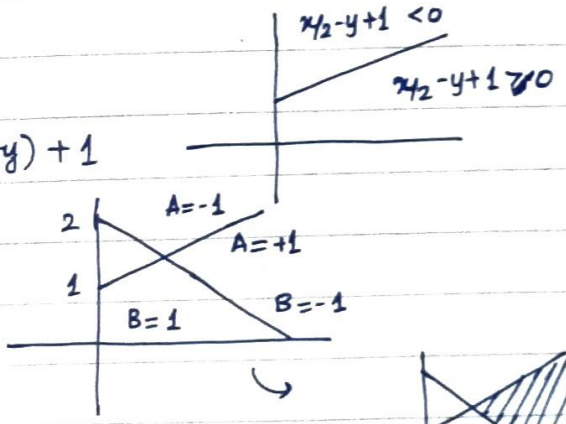


(Eg)



Neuron A : $1/2\,x+(-y)+1$

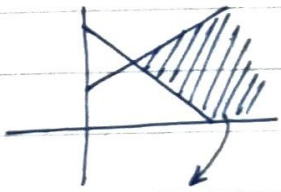
Neuron B : $2-x-y$



Output Neuron : $A-B-1/2$

$A-B-1/2>0 \Rightarrow A-B>1/2$ $\quad A=1, B=(-1)$

$A-B-1/2<0 \Rightarrow A-B<1/2$ ~~$A=(-1), B=$~~



Universal Approximation Theorem –

Given enough number of neurons any 2 layer neural network can approximate a given function with arbitrary closeness.

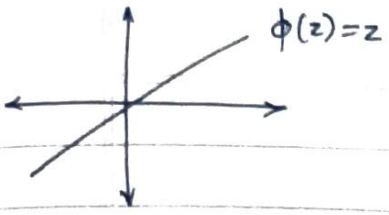
→ Activation Functions.

Outputs of +1 & -1 are very discrete — this will not allow computation of a continuous function. Output of perceptron is highly non linear & non differentiable — we cannot use gradient based methods to train perceptrons.

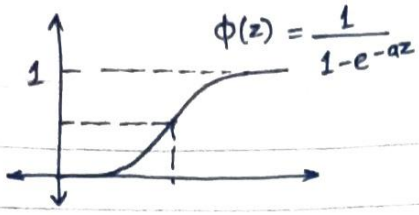
Instead of having output containing all 1, a sudden drop & Then all -1 — let us have smoother & continuous — done using Activation function.

Types of Activation Functions –

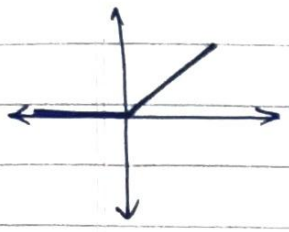
1. Linear Activation.



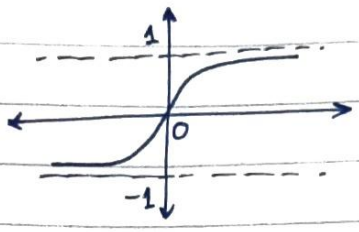
2. Logistic Activation



3. ReLU



4. Hyperbolic Tangent Activation.

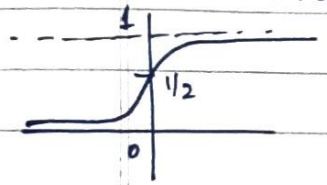


$\phi(u) = \tanh(\gamma u) = \frac{1-e^{-2\gamma u}}{1+e^{-2\gamma u}}$

• **Sigmoid Activation**

Takes a real valued number & squashes it into range between 0 & 1.

$f(x)$ (Output) $= \frac{1}{1+e^{-x}}$ where x is sum of all inputs to the neuron..



Advantage – Nice interpretation as the firing rate of a neuron. → 0 (not firing at all) ; 1 → (fully firing).

Disadvantage –

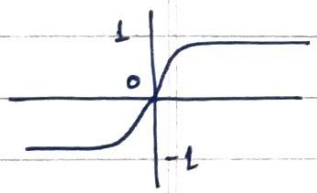
Sigmoid Neurons saturate & kill gradients, thus the Neural Network will barely learn when neuron's activation are 0 or 1 (saturate).

I can have +ve or no influence on you.

- Gradient at these regions almost zero.
- Almost no signal flows to its weights.
- If initial weights are too large then most neurons would saturate.

• **Hyperbolic Tangent Activation**

- Takes a real valued number & squashes it to range between −1 & 1.
- Just like sigmoid tanh neurons saturate ; unlike sigmoid output is zero centered.



- Tanh is a scaled sigmoid : $\tanh(x) = 2\text{sigm}(2x) - 1$.

$$\tanh(x) = \frac{2}{1+e^{-2x}} - 1.$$

↳ Smooth approximation of perceptron.

I can +ve/0/-ve influence on you.

• **Rectified Linear Unit Activation (ReLU)**

– Takes a real valued number & thresholds it at zero $f(x) = \max(0, x)$

Non differentiability at zero (Below 0 pt I will not send any signal out ; At ≥0 pt, I will send a linear signal out).

$$f(x) = \begin{cases} 0 & \text{for } x<0 \\ x & \text{for } x \geq 0 \end{cases}$$

- Most deep Networks use ReLU nowadays
- Trains much faster
  - Accelerates the convergence of SGD
  - Due to linear, non saturating form.
- Less Expensive form Operations
  - Compared to sigmoid / tanh exponentials.
  - Implemented by simply thresholding a matrix at zero.
- More expressive
- Reduces gradient vanishing problem. → Model doesn't train because gradient vanishes to 0.

Advantage – Gradients in +ve region don't saturate / become 0 ; your gradient is 1 or -1. & that means until you're in +ve region, you can keep training this weight much better. Better results than sigmoid.

EXAMPLE APPLICATION : HANDWRITING RECOGNITION.

Input → Machine → Output

(16×16 bit vector with each pixel either black/white)

(Vector of inputs $x_1, x_2, x_3 .. x_{256}$)

↳ where each x_i represents i^{th} pixel.

10 outputs $y_1, y_2, y_3, y_4 .... y_{10}$ where each y_i represents the probability / confidence that the input is the i^{th} digit. (i is 0)

(Eg : {0.1, 0.7 ..... , 0.2} ⇒ My digit is 2)

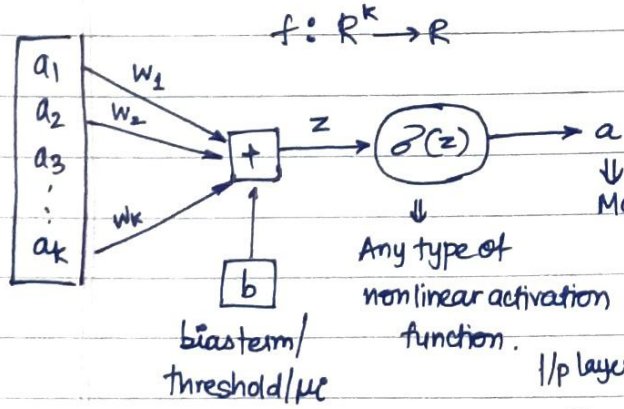
P(1) P(2) P(0)

How do we build such a machine?

⇒ Input dimensionality – 256 Output dimensionality – 10

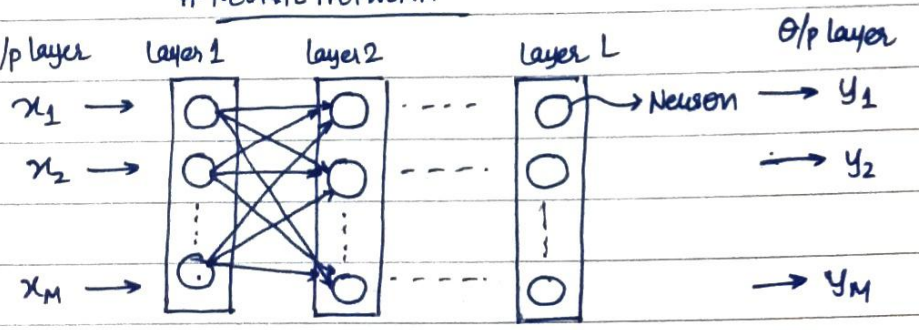
We wish to learn a function that goes from R^{256} (256 dimensional input) to a R^{10} (10 dimensional output).

This function $f : R^{256} \rightarrow R^{10}$ is represented by our neural network.



$z = a_1 w_1 + a_2 w_2 + a_3 w_3 - - - - + a_k w_k + b$.

NEURAL NETWORK



The layers in between I/p & O/p are called hidden layers because human does not understands what happens in them.

Fully connected feed forward network

Every o/p of previous layer goes to every input of next layer.

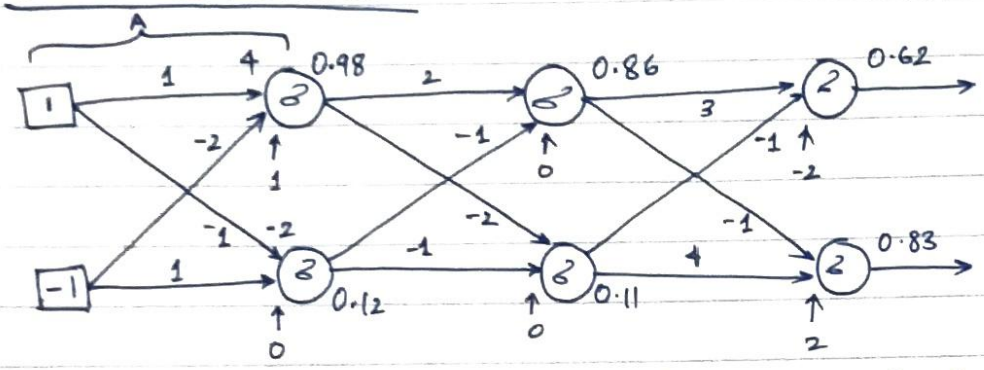
How will you go about identifying this?

1st you will identify the smallest shapes, then slightly bigger & so on ... such that they can arranged in sequential sort of fashion → Layer 1 → Layer 2 → Subsequent layers.

Technically if we knew all the weights we can find out what each neuron represents However, it would still be difficult to do this for middle neuron because the i/p for this neuron would function of all neurons in previous layer.

"Deep" means too many layers.

EXAMPLE NEURAL NETWORK



$\sigma(x) = \frac{1}{1+e^{-x}}$ Similarly for different inputs pairs I will get diff. outputs. $f: R^2 \to R^2$

So, $f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right) = \begin{bmatrix}0.62\\0.83\end{bmatrix}$ $f\left(\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}0.51\\0.85\end{bmatrix}$

Different parameters will define a different function.

The computation in Part A can be represented as — matrix multiplication

$$\begin{bmatrix}1 & -2\\-1 & 1\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}4\\-2\end{bmatrix}$$

W Matrix where each row represents the weights of one neuron ← ↓ This product gives weighted sum of inputs ↓ bias term

Each layer is nothing but matrix multiplication & bias addition.

Adding non-linearity - $\sigma\left(\begin{bmatrix}1 & -2\\-1 & 1\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}0.98\\0.12\end{bmatrix}$

Each layer of the neural network is of the form $WX+B$

$\therefore$ i^{th} layer can be represented as (w_i, b_i) & function computed is $\sigma(w_i x + b_i)$

$$a_i = \sigma(w_i a_{i-1} + b_i)$$

a_i: o/p of current layer ← a_{i-1}: o/p of previous layer.

$\therefore y = f(x) = \delta(W_L \ldots \delta(W_2 \delta(W_1 x + b_1) + b_2) \ldots + b_L)$

We may use parallel computing techniques to speed up these operations. (using GPUs)

In the general version of Neural Network any number from $-\infty$ to $+\infty$ could be output of neural network. At the end, we will put in a specialised layer to convert that representation into output representation.

SOFTMAX

Softmax layer as output layer

Suppose my general representation is $z_1, z_2, z_3 \ldots z_{10}$ I wish to convert z_1 to y_1, z_2 to y_2 & so on such that these y_i's are probability distribution. Clearly I wish to find a δ that helps me achieve this.

$z_1 \longrightarrow (\delta) \longrightarrow y_1 = \delta(z_1)$

$z_2 \longrightarrow (\delta) \longrightarrow y_2 = \delta(z_2)$

$z_3 \longrightarrow (\delta) \longrightarrow y_3 = \delta(z_3)$

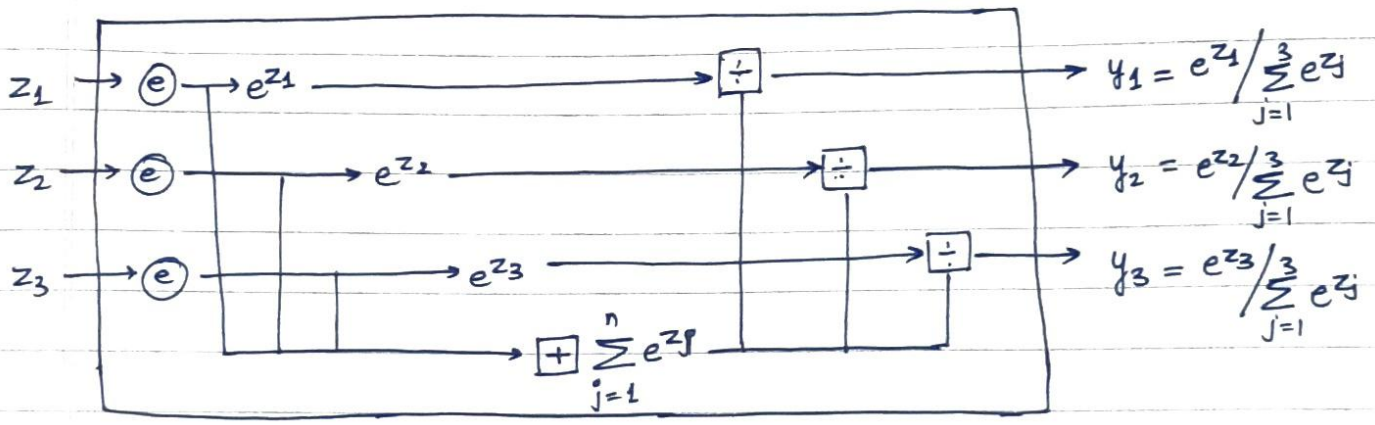
Let's say if we put in a sigmoid as δ. Then we will have $0 \le y_1, y_2, y_3 \le 1$ but they all may not necessarily sum to zero.

Moreover z_i's may not be easy to interpret because they may be coming from complex combination of Neural Network.

Now, we wish to enforce that these prob. sum to 1.

We do this using a softmax layer. In softmax layer, we recognise that z_i's may be +ve/-ve. However a higher value must lead to higher prob.

So, we want a function that converts -ve z to +ve & is higher for higher $z \rightarrow f(x) = e^x$.



SOFTMAX FUNCTION.

Takes a lots of inputs between $-\infty$ & $+\infty$ & converts them into probability. dist. of same dimension.

You may use other functions than softmax — coz the ultimate goal is to output the higher value. We may also use the normal max function — nothing wrong — however max function may not be differentiable — in that case we may have to use subgradients — softmax is a softer version of max which is differentiable.

Architechture of the problem –

Take input → Convert into Bit Vector → Pass through Numerous layers → Get a probability distribution

Output Max.

What is unknown in the above Architechture ? ⇒ Weights (W_i & b_i)

Let θ be the set of all the parameters ⇒ $\theta = \{W_1, b_1, W_2, b_2 \ldots . W_L, b_L\}$

Goal : find parameters in θ such that it makes correct predictions

TRAINING DATA –

- Preparing Training data : Images & their Labels
- Cost function – Given a set of network parameters θ, each example has a cost value e.g.

  Say if I given an example from training data as input, my output would be a probability distribution. This prob. distribution could then be compared to true value from which we can calculate how costly / bad the prediction is.

  (Eg) –

  Target distribution → $\{1, 0, 0, 0, \ldots 0\}$

  Actual / Predicted distribution → $\{0.2, 0.3, \ldots . 0.5\}$

Clearly the predictions are really bad. & you could like to have high penalty for such parameters.

So, you can have cost function for this.

Cost function can be Euclidean distance (Squared Loss) or cross entropy of the network output & target.

Let this cost function be denoted by '$L(\theta)$'.

Finally I want those parameters that minimise the loss.

Now, I have many training data points & using these I can compute many loss functions.

Total Cost ⇒ $C(\theta) = \sum_{r=1}^{R} L^r(\theta)$

We want to find set of parameters that minimize this $C(\theta)$.

We can do this using Gradient Descent –

We have a loss function, some set of parameters (θ) → we can compute the derivative

Randomly pick a starting point (θ_0) & compute negative gradient at θ_0 (because we're minimizing), move in this direction until we get θ^*. ↳ $-\nabla C(\theta_0)$

↳ $-\eta \nabla C(\theta_0)$

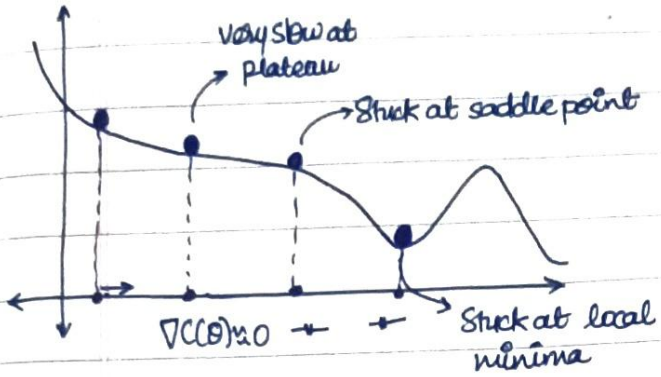
Are we always guranteed to reach a global minima ?

→ No, never in case of NNs. coz there are layers of neurons & each layer has activation which is not necessarily convex & combination of all these

makes a highly non-linear function. ∴ θ^* you finally reach is some local optimum.

∴ Different Initial Point (θ_0) $\xrightarrow{\text{leads to}}$ Different Local Minima (θ^*) ⟶ & Different Results.

What else apart from Local Minima?



Generally we have huge training data & therefore going over this entire training data to compute (θ) & then computing gradient & then taking one step towards desired direction takes a lot of time. ⌊—(*)

To resolve this, you compute (depending on size of GPU you have) the number of data points, you can fit ~~in 1 matrix~~ so that each matrix computation takes 1 unit of time — Say k.

⟶ Then you divide full training data into ~~piec~~ pieces of k data points each & you call each of this piece — "Mini-Batch".

So, I now take the gradient step only based result of one mini batch. In the next iteration I will give next mini batch as input. When I finish going over all the batches I can:

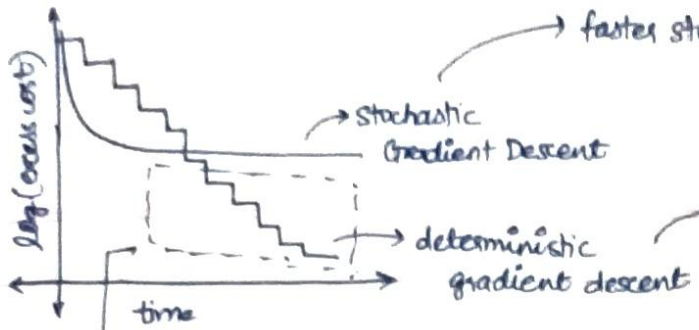
- Modify the mini-batch (size/content) & restart
- Do not make any changes; just start from the 1st mini batch again.

Going over the training data once is termed as "One Epoch". One epoch goes over multiple batches & takes many gradient steps. Summarizing, the steps are as follows —

1. Randomly initialise θ^0
2. Pick the 1st Batch; $C = L_1 + L_{31} + L$ - - - - - -
   $\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$
3. Pick the 2nd Batch: $C = L_2 + L_{16} + L$ - - - -
   $\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$

Extreme Case of Mini Batch - Stochastic Gradient Descent (SGD) — we take only 1 data pt at a time & other extreme is Gradient Descent (given by (*))

In SGD, there are greater chances of going in wrong direction.



→ faster steps but slower decrease in cost. → Superior in low accuracy / time situations.

→ Very slow – because you go over entire training data before taking 1 move.

Which is better? — It depends!

Early on in the training SGD is better & later down the line GD is better

→ Since we have large training data points we are almost never in this region of the curve.

Generally people use GD in the minibatch & Stochastic GD across batches.

BACKPROPOGATION –

We have chosen a differentiable loss (i.e. the whole loss is differentiable wrt each parameter in the neural network).

- Computing Gradients –

1. If we choose a differentiable loss, then whole function will be differentiable wrt all parameters.
2. Because of non-linear activations whose combination is not convex, the overall learning problem is not convex.
3. What does (Stochastic) (sub) gradient descent do with non-convex functions? It finds a local minimum
4. To calculate gradients, we need to use chain rule from calculus.
5. Special name for (S) GD with chain rule invocations : Backpropogation.

We will convert the NN into a computational graph.

for every node in the computation graph, we wish to calculate the 1st derivative of L_n with respect to that node. For any node a, let :

$$\bar{a} = \frac{\partial L_n}{\partial a} \qquad \text{Base Case : } \bar{L_n} = \frac{\partial L_n}{\delta L_n} = 1.$$

After working forwards through the computation graph to obtain the loss L_n, we work backwards through the computation graph, using the chain rule to calculate $\bar{a}$ $\forall$ node a, making use of the work already done for the nodes that depend on a.

$$\frac{\delta L_n}{\delta a} = \sum_{b:a\to b} \frac{\delta L_n}{\delta b} \cdot \frac{\delta b}{\delta a} = \sum_{b:a\to b} \bar{b} \cdot \frac{\delta b}{\delta a}$$

$$= \sum_{b:a\to b} \bar{b} \cdot \begin{cases} 1 & \text{if } b = a + c \text{ for some } c \\ c & \text{if } b = ac \text{ for some } c \\ (1-b^2) & \text{if } b = \tanh(a) \end{cases}$$

(Eg) $b = \tanh(a) = \frac{2}{1+e^{-2a}} - 1 = \frac{1-e^{-2a}}{1+e^{-2a}}$

$1-b^2 = (1+b)(1-b)$

$= \left(\frac{2}{1+e^{-2a}}\right)\left(\frac{+2e^{-2a}}{1+e^{-2a}}\right)$

$= \left(4e^{-2a}/(1+e^{-2a})^2\right)$

$\frac{\delta b}{\delta a} = \frac{-2}{(1+e^{-2a})^2} \cdot (-2)(e^{-2a}) = \frac{4e^{-2a}}{(1+e^{-2a})^2}$

$\therefore \frac{\delta b}{\delta a} = (1-b)^2$

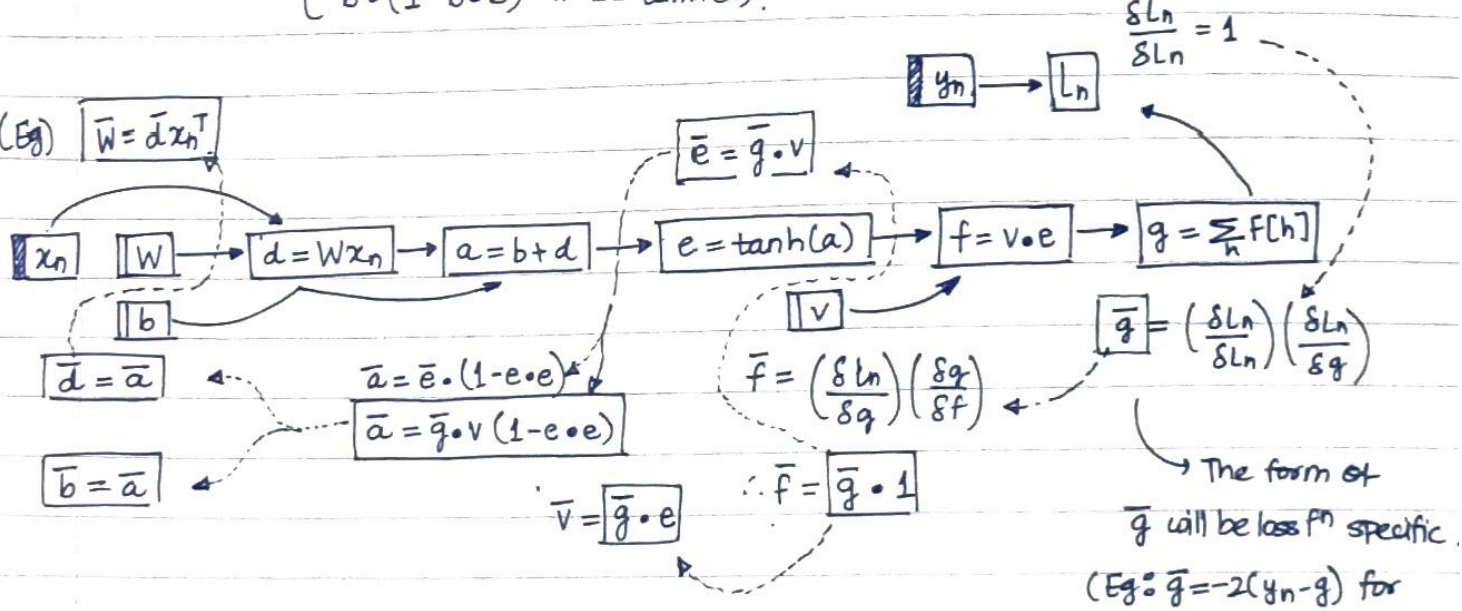
- In terms of Vectors / Matrix Notation –

→ Hadamard Product (Dot Product / Inner Product) : Pointwise product for vectors in $\mathbb{R}^n$.

$$\therefore a \cdot b = \begin{bmatrix} a[1].b[1] \\ a[2].b[2] \\ \vdots \\ a[n].b[n] \end{bmatrix}$$

$$\bar{a} = \sum_{b:a\to b} \left(\sum_{i=1}^{|b|} \bar{b}[i] \cdot \frac{\delta b[i]}{\delta a} \right)$$

$$= \sum_{b:a\to b} \begin{cases} \bar{b} & \text{if } b = a + c \text{ for some } c \\ \bar{b} \cdot c & \text{if } b = a \cdot c \text{ for some } c \\ \bar{b} \cdot (1 - b \cdot b) & \text{if } b = \tanh(a). \end{cases}$$



Now we wish to evaluate derivative of loss wrt W;

$\bar{d}$ is a vector, x_n is vector but W is matrix. Now I wish to differentiate d wrt w_{ij} & then do it in a matrix form –

- Derivative Wrt Matrix Multiplication –

$$\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + W_{13}x_3 \\ W_{21}x_1 + W_{22}x_2 + W_{23}x_3 \\ W_{31}x_1 + W_{32}x_2 + W_{33}x_3 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

The main point to realise — is w_{ij} only influences d_i

Moreover $\frac{\delta d_i}{\delta w_{ij}} = x_j$

If we are given $\bar{d}$. Then;

$$\frac{\delta L_n}{\delta W} = \begin{bmatrix} \bar{d}_1 x_1 & \bar{d}_1 x_2 & \bar{d}_1 x_3 \\ \bar{d}_2 x_1 & \bar{d}_2 x_2 & \bar{d}_2 x_3 \\ \bar{d}_3 x_1 & \bar{d}_3 x_2 & \bar{d}_3 x_3 \end{bmatrix} = \begin{bmatrix} \bar{d}_1 \\ \bar{d}_2 \\ \bar{d}_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \bar{d} x^T$$

↳ output is going to be a 3×3 matrix.

Summarising,

- Each computation that NN is doing can be converted into a computational graph where at one node only one computation got done (either + / dot product / non-linearity)
- Similarly each NN can be converted into larger computational graphs & then perform back propogation. — at each step multiply the gradient that is flowing back times the gradient of the edge & adding for all outgoing edges gives gradient of current node.
- This forms the basis of training NNs.
- Note that here we did not compute derivative wrt x_n because x_n was the primary input; if x_n had been output of previous layer of NN; we would have computed $\bar{x}_n$.

WHY SHOULD WE TRAIN A DEEP NN? :-

- Deeper is Better? — As Number of layers ↑, Accuracy Improves

  Not suprising as ↑ in #layers → more parameters

  You can figure out the total # parameters in an NN (if it is fully connected)

UNIVERSALITY THEOREM :-

Any continuous function $f : R^N \rightarrow R^M$ can be realised by a network with 1 hidden layer, given hidden neurons in enough number / quantity.

Question that now arises is :

Why should we train a Thin & Tall NN instead of Fat & Short NN?

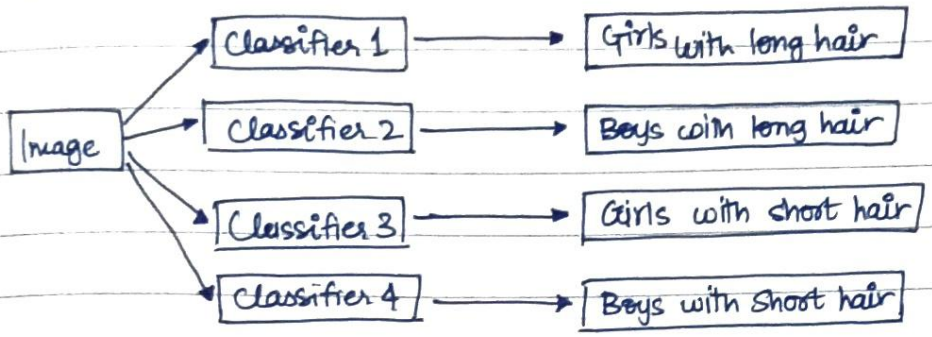
⇒ Note that the above theorem is representability theorem & not learnability theorem. It only says that any continuous fn can only be realised or represented using this 1 layer NN. — there exists some set of parameters such that this function can be realised by those set of parameters. It doesn't say that those parameters can be learned by backpropogation / training data.

When we start to learn a shallow-fat NN to realise a function, that we are looking for, we don't usually succeed.

Intuitively, tall & thin NN requires lesser # neurons (However this isn't proved).

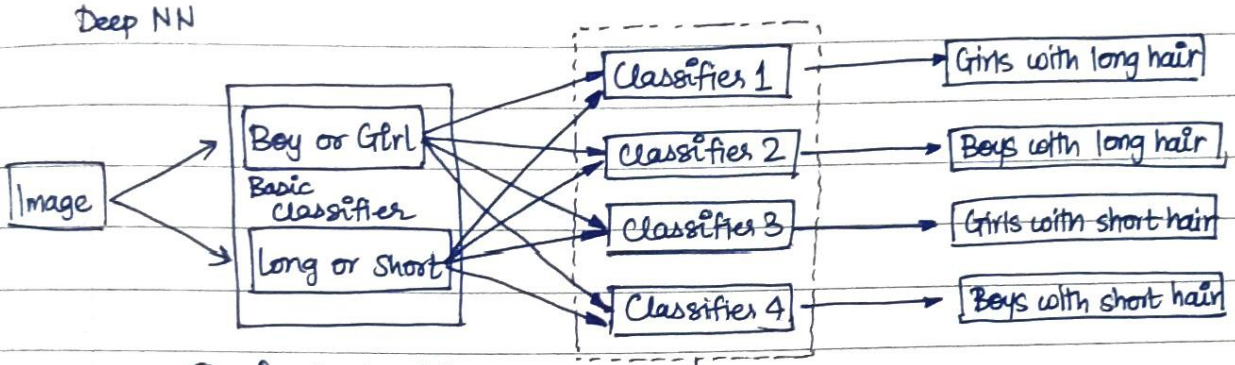
Studies show that 2 layer NNs with lesser neurons give extremely better results than 1 layer NNs with very large number of neurons. (for same # parameters).

(Eg) Shallow NN



(Weak because of little examples)

Deep NN



(Fine with little data)

Sharing by the following classifiers as module

↓

Enough data for both

Can be trained with little data

Deep → Modularization

First layer is learning something which is used by 2nd; 2nd learns something which is used by 3rd

This depth brings out the compositionality of the world. (A thing is composed of various parts which are further composed of parts and so on). It leads to deconstruction at every depth — with the lowest layer having simplest description of required object.

TRADITIONAL ML Vs DEEP LEARNING -

Most ML methods worked well because of human designed representations & input features.

ML becomes just optimizing weights to best make a final prediction.

Traditional ML was all about our ability to come up with good features so that the machine can learn good weights for it.

∴ ML in Practice → Describing your data with features a computer can understand. } Domain Specific Requires Ph-D. level talent

→ Learning Algo } Optimizing the weights on features.

- What did DL do?

DL allowed learning both features & weights by machine together.

- A ML subfield of learning representations of data. Exceptional effective at learning patterns.
- DL algo attempt to learn (multiple levels of) representation by using hierarchy of multiple layers.

If you provide the system tons of info, it begins to understand it & respond in useful ways.

→ I may not learn the features that aren't necessary for current task.

Feature Extraction & Classification happens together.

What is the role of human here? — Architechture Design. — decide which neuron should be connected to what; what should be non-linearities; how many weights should be these etc.

~~Not~~ Meta learning / Auto ML — M/c does Architechture learning!. No human req!.

Human in loop ML / Human in loop AI — Both human & m/c work together.

CONVOLUTIONAL NEURAL NETWORKS (CONNs) :-

Until now we were given — that we are giving input as bit vector representation however one must note that for correct predictions, our m/c must predict 1 for all the images like : 

So, ~~see~~ The computer scientists came up with translational variances — if rotate 1, shift 1 left/right, shrink in size / blow up — all o will give the same prediction 1.

Only bit vector representation wouldn't allow such type predictions.

CONVOLUTION

Image →

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter →

1	0	1
0	1	0
1	0	1

Convoluted feature →

4	3	4
2	4	3
2	3	4

→ Think this 1 feature.

Applying a filter ⟺ finding a feature

I can have many such convoluted features

1st layer – low level information about img

2nd Layer – Slightly better info & so on.

I put a particular filter on the input, create another image – learn some useful information, then create more filters on that image as input – which helps me create next level features.

Is this translational invariant? → Yes, this can be ensured using max pooling

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pooling with 2×2 filters & stride 2 →

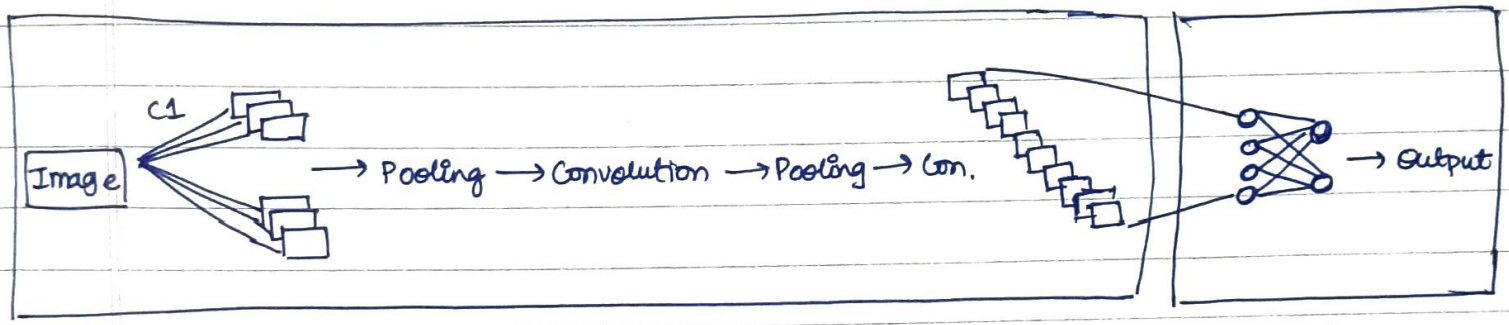
6	8
3	4

I will keep the strongest influence irrespective of where it was in this small sub image

I will do this at every such sub image

This would give me local translational invariances.

#CNN ARCHITECHTURE



Problem – How do we figure out what should the filters be?

Let the algorithm learn it. – You will just give the structure (3×3 filter / & 4×4 filter etc) & then model will independently figure out which ~~filter~~ filters are necessary for task at hand.
(using backpropogation etc)

Zero Short learning – Ability to complete a task without having received any training examples.

→ M/c don't have this.

#DEEP REINFORCEMENT LEARNING –

- Function Approximation –
  - Lookup Table (Eg Q(S,a) table) : doesn't scale (Humongous for large problems). Also known as curse of dimensionality; not feasible if state space is continuous.

- The key idea is to use function Approximation:
  - Approximate $Q(s,a)$ as a parametric function.
    Automatically learn the parameters (w) $Q(s,a) \sim \hat{Q}(s,a;w)$

- The key idea of Deep Q learning:
  Train a deep network to represent Q function.
  w are parameters of deep network.

Regular Q learning: nudge $Q(s,a)$ towards target

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\underbrace{r + \gamma \max_{a'} Q(s',a')}_{\text{target}} - Q(s,a))$$

Deep Q learning: Optimise for a given loss function (like nudge approximated Q values towards target by minimizing squared error)

$$\text{loss}(w) = (\underbrace{r + \gamma \max_{a'} Q(s',a';w)}_{\text{New target value computed by the network}} - \underbrace{Q(s,a;w)}_{\text{Old value computed by the network}})^2$$

↳ Nudge the parameters w such that loss is minimised.

Difference: Target Q-value is also moving!
↳ Between Standard Deep learning & Deep RL.
Suppose I change my weights such that $r + \gamma \max_{a'} Q(s',a';w)$ also moves closes to $Q(s,a;w)$ there is a problem — Weights have changed so the target itself has changed!
In typical deep learning the training data is given to you — fixed targets.
Here, training data is generated by w & then is used to update w

- Online Deep Q learning Algorithm—

(1) Estimate $Q^*(s,a)$ values using deep network (w)
(2) Receive a sample (s,a,s',r)
(3) Compute target: $y = r + \gamma \max_{a'} Q(s',a';w)$
(4) Update w to minimize: $L(w) = (y - Q(s,a;w))^2$
(5) Nudge the estimates:

$$w \leftarrow w - \alpha \, \partial L / \partial w \quad \text{or} \quad w \leftarrow w - \alpha \nabla_w L$$

— opposite to direction of gradient because we're minimizing.

where gradient $\nabla_w L = (Q(s,a;w) - y) \nabla_w Q(s,a;w)$

CHALLENGES IN TRAINING DEEP Q LEARNING -

- Target values are moving
- Successive states are not independent & identically distributed (i.i.d) they are correlated.
- Successive states depend on w (policy)
- Small changes in w might lead to large changes in policy
- These make training deep Q networks highly unstable

— solutions :

- freeze target Q ~~values~~ network weights ; update sporadically
- Experience Replay

↳ I will learn based on the experience in my memory — like a person who never loses memory

• EXPERIENCE REPLAY -

- Step 1 : Compute Experience Buffer
  - At each time step :
    - Take action a_t according to ϵ-greedy policy
    - Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in Replay memory buffer
    - Replay Memory Buffer contains many $(s_t, a_t, r_{t+1}, s_{t+1})$ tuples for $t = 0, 1, 2 \ldots T$.
- Step 2 : Weight updates using Replay Buffer
  - Repeat K steps :
    - Randomly sample a minibatch B of (s, a, r, s') experiences from replay memory buffer
    - Perform update to minimize loss function on this minibatch.

$$\sum_{(s,a,r,s') \in B} E\left[\left(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a, w)\right)^2\right]$$

$\max_{a'} Q(s', a'; w^-)$: Parameters of target network are frozen.

$Q(s, a, w)$ → I am saying find me those parameters w such that one step look ahead Q function as computed by old parameters, computes the same value as I compute with my current parameters

    - Once in a while $w^- \leftarrow w$
- Go to Step I (Recompute Replay Buffer).

This basically says that previous set of deep networks one step look ahead I have already encapsulated in my current w parameters.

(Q) Do I need to enumerate all states possible in my experience?

No, - the whole pt was that we will be able to generalise about state features. We hope that function approximation will be able to capture salient parts of state.

So, during testing if I get a state I have never seen before, it may share some properties with other states I have experienced in the past & based on that

function approximation would be able to give could predicted value.

(Q) How do you initialize these things?

Random initialization of both w, w^- → Then I get trained w which I assign to w^- → Now I may again randomly initialize w or I can choose to make small perturbations in previously obtained trained values.

A3C – Asynchronous Advantage Actor Critic Algorithms.

SUMMARY –

- Deep Learning Strengths:
  1. Universal Approximations: learn non trivial functions.
  2. Compositional Models: Similar to human brain
  3. Universal Representation: Across Modalities
  4. Discover features Automagically in a task specific manner; features not limited by human creativity.
- Deep Learning Weaknesses:
  1. Resource Hungry (data/compute)
  2. Uninterpretable
- Deep RL: Replace value/policy tables by deep networks

  Great success in Go, Atari.