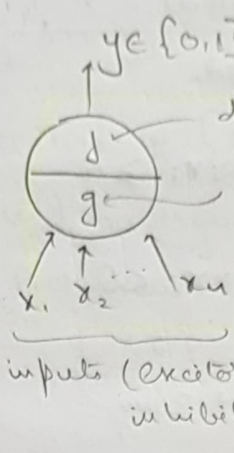
DL - NPTEL

Lec 2.2 : Mc Culloch Pitts Neuron (Thresholding Logic)

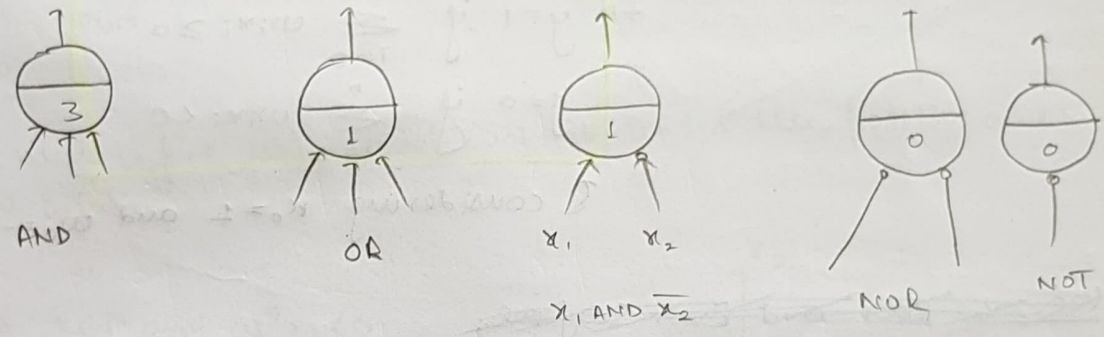


$$g(x_1, x_2, x_3, \ldots, x_n) = \sum_{i=1}^{n} x_i$$

$$y = f(g(x)) = 1 \text{ if } g(x) \geq \theta$$

θ ? threshold

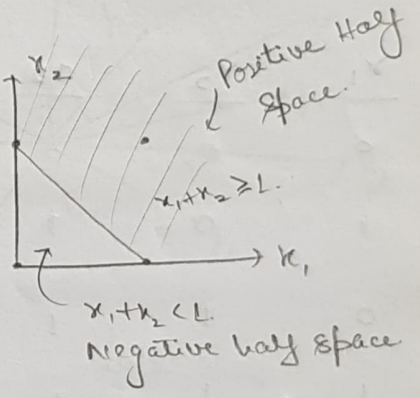
$$= 0 \text{ for } g(x) < \theta$$



Geometric Interpretation:

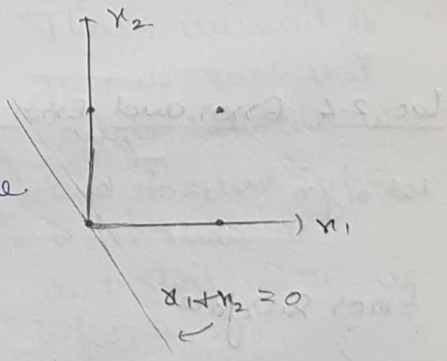
OR: $x_1 + x_2 \geq 1 \Rightarrow y = 1$

$< 1 \Rightarrow y = 0$



* MP Neurons learn a linear decision boundary.

Tautology (always ON) $x_1 + x_2 \geq 0$ for $g \geq 1$



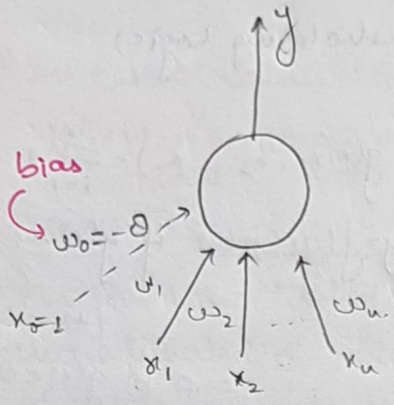
* MP Neurons can represent boolean functions which are linearly separable

Lec 2.3 : Perceptrons

What about non-boolean (real) inputs?

How to assign more weights to some inputs?

How to deal with non-linearly separable?



Here ~~x_0 is~~ w_0 is called the bias.

* weighted inputs and possible to learn these weights (not hard coded)
* Real values can be used.

$$y = 1 \quad \text{if} \quad \sum_{i=1}^{n} w_i x_i \geq \theta$$

$$y = 0 \quad \text{if} \quad \sum_{i=1}^{n} w_i x_i < \theta$$

weighted sum.

$$y = 1 \quad \text{if} \quad \sum_{i=1}^{n} w_i x_i - \theta \geq 0$$

$$\Rightarrow \quad y = 1 \quad \text{if} \quad \sum_{i=0}^{n} w_i x_i \geq 0$$

$$y = 0 \quad \text{if} \quad \sum_{i=0}^{n} w_i x_i < 0$$

considering $x_0 = 1$ and $w_0 = -\theta$

~~Lec 2.4: Error and Error Surface:~~

x_1	x_2	OR
0	0	0
1	0	1
0	1	1
1	1	1

$\sum_{i=0}^{2} w_i x_i < 0 \Rightarrow w_0 < 0$

" $\geq 0 \Rightarrow w_0 + w_1 \geq 0$

" $\geq 0 \Rightarrow w_0 + w_2 \geq 0$

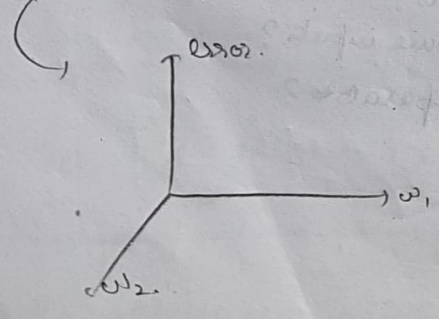
" $\geq 0 \Rightarrow w_0 + w_1 + w_2 \geq 0$

Solve these to get the linear threshold.

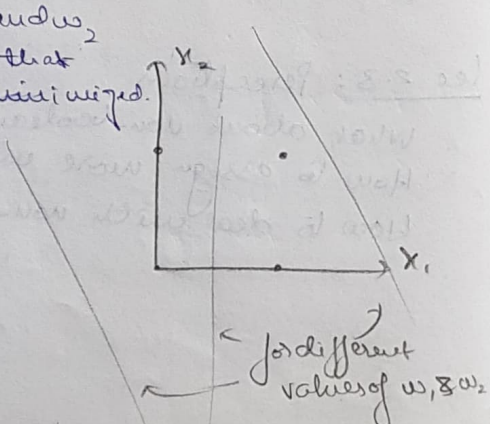
Lec 2.4 Error and Error Surface:

let's fix $w_0 = -1$

Error Surface



Choose w_1 and w_2 such that error is minimized.



Lecture 2.5 : Perceptron learning Algorithm

* Adjusting the weights of the input of a perceptron.

P : inputs with label 1. (positive inputs)
N : " " " 0 (-ve inputs)

Initialize $w = \{w_0, w_1, \dots w_n\}$ randomly

while ! convergent do

  Pick random $x \in P \cup N$:

    if $x \in P$ and $w^T x < 0$, then

      $w = w + x$;

    end

    if $x \in N$ and $w^T x > 0$, then

      $w = w - x$;

    end

end.

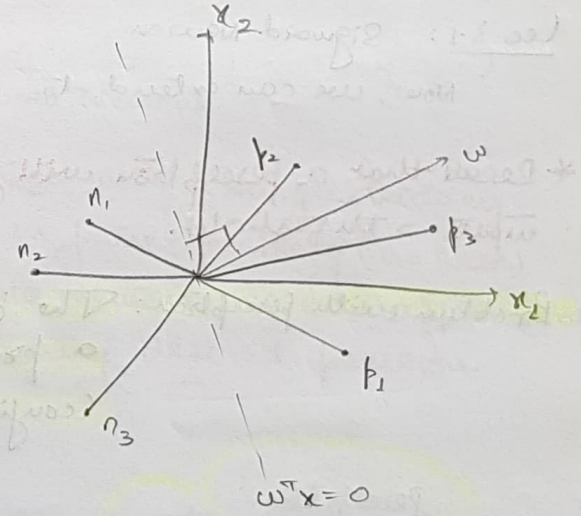
$w = [w_0, w_1, w_2, \dots, w_n]$

$x = [1, x_1, x_2, \dots, x_n]$

vectors

$w^T x = \sum_{i=0}^{n} w_i x_i = w \cdot x$

We need to find $w^T x$ such that it divides the positive and -ve inputs.



For $x \in P$ and $w^T x < 0$

$\Rightarrow \cos\alpha < 0$

$\Rightarrow \alpha > 90^\circ$

and we want $\alpha < 90^\circ$

So modify w

$w_{new} = w + x$.

$\cos(\alpha_{new}) \propto w_{new}^T x$.

$\propto (w+x)^T x$

$\propto w^T x + x^T x$

$\propto \cos\alpha +$ +ve number

$\Rightarrow \cos(\alpha_{new}) > \cos\alpha$.

$\Rightarrow \alpha_{new} < \alpha$

Thus we decrease α repeatedly.

Similarly, for $x \in N$ and $w^T x > 0$

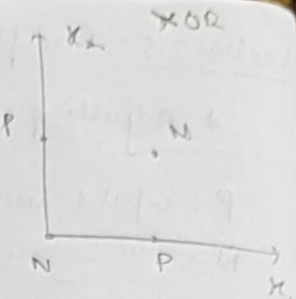
$\alpha_{new} > \alpha$.

we increase α repeatedly.

Thus, we need to find w such that angle between w and positive inputs is less than 90° and that with -ve inputs is $> 90^\circ$.

$\cos\alpha = \dfrac{w^T x}{\|w\| \|x\|}$

Lec 2.7 : Linearly Separable Boolean Func:

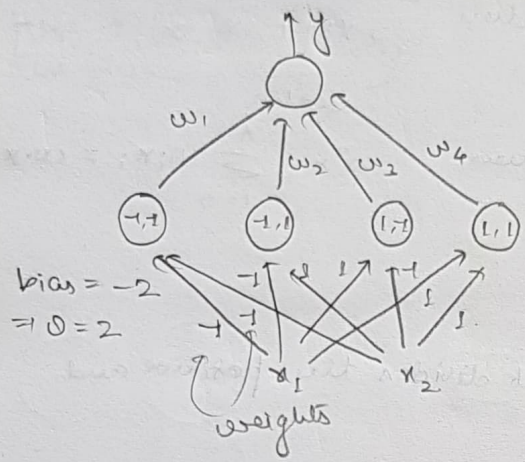


XOR and XNOR are non linearly separable.

Multiple perceptrons are needed to implement non-linearly separable.

Lec 2.8: Network of Perceptrons:

Implementing XOR:



For $w_2, w_3 = 1$
and $w_1, w_4 = 0$
XOR

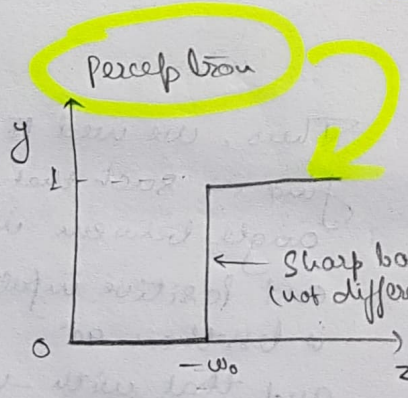
Lec 3.1: Sigmoid Neuron

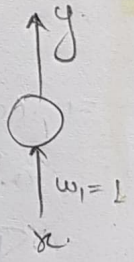
Now, we can extend to $y = f(x)$ $x \in R^n$ and $y \in R$ (non-binary)

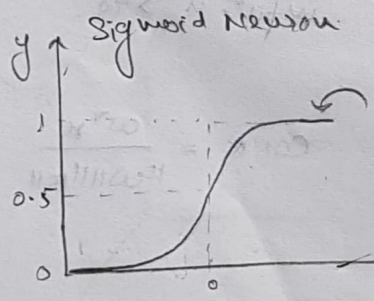
* Recall that a perceptron will fire if the weighted sum of inputs > threshold.

Problem with perceptron: The output is 1 or 0. We don't have a probability for each output (confidence is not known)

Eg:-







$y = 1$ if $x > 0.5$

$y = 0$ if $x < 0.5$

$x = 0.49, 0.51$ are very close but still give very different outputs.

logistic func. output lies between 0 and 1 (analogous to prob).

lecture 3.2 Supervised Learning:

$$\hat{y} = \frac{1}{1+e^{-w^T x}}$$ logistic regression

weight vector

$\hat{y} = w^T x$ linear reg.

weight matrix

$\hat{y} = x^T W x$ quad. reg.

Given x and y, we need to estimate w.

Loss func: $L(w) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$.

lecture 3.4: Gradient Descent

$w = w - \eta \nabla w$

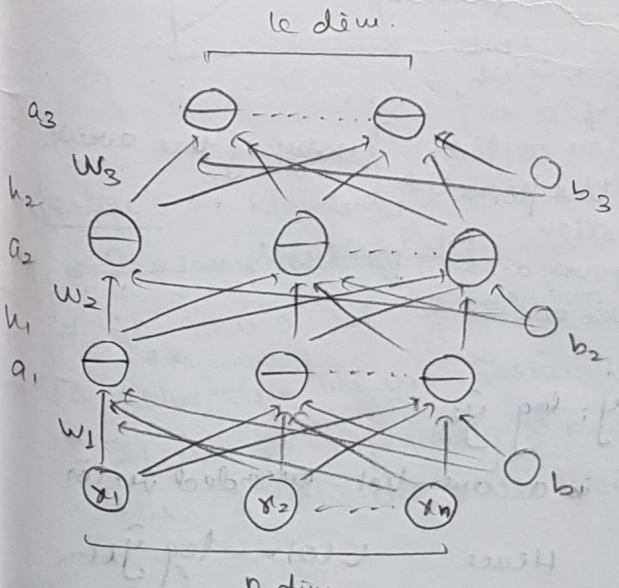
$b = b - \eta \nabla b$ ← gradient of bias

↑ learning rate

lec 3.5: Multilayer Networks of Sigmoid Neurons.

* A multilayer network of perceptrons with a single hidden layer can represent any boolean func. precisely (no error) whereas for the case of sigmoid neurons, we can approximate any func. (continuous) to any desired precision.

lecture 4.1: Feed forward NN:



Preactivation:

$a_i(x) = W_i h_{i-1}(x) + b_i$

Activation:

$h_i(x) = g(a_i(x))$

element wise (can be operation sigmoid, tanh etc)

Activation of output layer:

$h_L(x) = O(a_L(x))$

can be softmax, linear etc

$h_1(x) = g(\underset{R^{n\times n}}{W_1} x + b_1) \to R^n$

$h_2(x) = g(W_2 h_1(x) + b_2)$

$h_3(x) = o(W_3 h_2(x) + b_3) \to R^k$

Lecture 4.3: output func. and loss func.

For regression problems, we want the outputs to be Real numbers and unbounded. Then, o can be a linear func.

$$\hat{y} = h_L = o(a_L) = W_0 a_L + b_0$$

For classification problems, the output is a probability distribution over all the classes. Then, o should be softmax. We can't use softmax because the sum of the individual prob. will not equal 1.

$$\hat{y} = [\hat{y}_1 \quad \hat{y}_2 \quad \cdots \quad \hat{y}_k] \qquad \hat{y}_j = \frac{e^{a_j}}{e^{a_1} + e^{a_2} + \cdots + e^{a_n}}$$

$$a_L = [a_{L_1} \quad a_{L_2} \quad \cdots \quad a_{L_3}]$$

Loss function for classification (cross entropy):

Information: $I(S) = -\log_2 P(S)$

Entropy: $H(S) = \sum_i P(S_i) I(S_i)$

$= -\sum_i P(S_i) \log P(S_i)$

Cross entropy: (tells how close predicted dist. is to the true dist.)

$P \to$ true distribution.

$q \to$ predicted distribution.

$= -\sum_i P_i \log(q_i)$

- $\log(q_i)$: information content based on the predicted distribution } value of the event
- summing over actual prob. dist (prob. of the event)

$$L(\theta) = -\sum_{i=1}^{n} y_i \log \hat{y}_i$$

But for classification, y is a one-hot-encoded vector

thus $y_i = 1$; for $i = k$ Hence $L(\theta) = -\log \hat{y}_k$

$y_i = 0$; otherwise. predicted prob. of the correct class

Thus we minimize $-\log \hat{y}_k$ or maximize $-L(\theta) = \log \hat{y}_i$

($\hat{y}_i$ → probability that x belongs to class-1 (log likelihood of data))

Minimizing the cross entropy:

minimize $-\sum p_i \log q_i$ under the constraint $\sum q_i = 1$
q_i

using lagrangian multiplier:

$$\min_{q_i} \left[-\sum p_i \log q_i + \lambda \left(\sum q_i - 1 \right) \right]$$

Diff, $-\frac{p_i}{q_i} + \lambda \times 1 = 0$

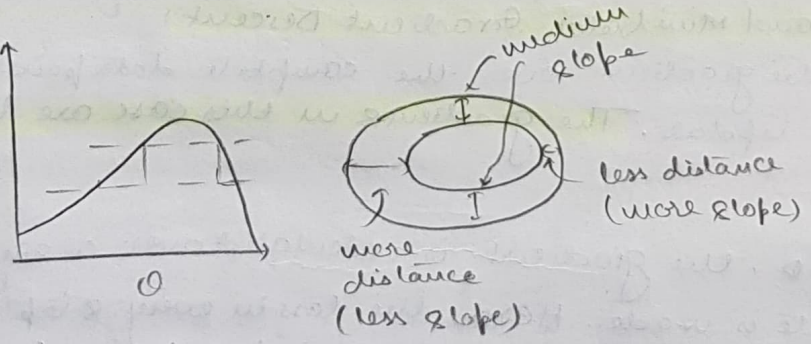
$\Rightarrow p_i = \lambda q_i$

Here, $\lambda = 1 \Rightarrow p_i = q_i$

Lecture 5.1: Gradient Descent.

$\theta = \theta - \eta \nabla \theta$ ⇒ When gradient (slope) is large update is large and vice versa

Lecture 5.3: Contour Maps



Lecture 5.4: Momentum Based Gradient Descent:

~~It takes a lot of steps~~

* GD takes a lot of steps to navigate regions of gentle slope.

To solve this we use momentum:

If we are repeatedly taking steps in a particular direction, then I should start taking bigger steps in that direction.

$\theta_{t+1} = \theta_t - \eta \nabla\theta - \gamma\, update_{t-1}$.

$\theta_{t+1} = \theta_t - update_t$

$update_t = \eta \nabla\theta_t + \gamma . update_{t-1}$ ← current update depends upon prev update

$update_0 = 0$

$update_1 = \eta \nabla\theta_1$

$update_2 = \eta \nabla\theta_2 + \gamma \cdot \eta \nabla\theta_1$

$update_3 = \eta \nabla\theta_3 + \gamma \eta \nabla\theta_2 + \gamma^2 \eta \nabla\theta_1$

↑ exponentially weighted avg. (more weight → less weight)

$0 < \gamma < 1$.

↑ Momentum term

* Due to build up of momentum, the loss overshoots (problem) but still converges faster than GD. (overshoots → oscillation)

lec 5.5: Nesterov Accelerated Gradient (NAG):

* Solves the oscillating loss problem.
* Look before you leap.
* First move in the intended direction according to the history and check the derivative at this look ahead point and move accordingly. Thus oscillations are reduced.

$\theta_{lookahead} = \theta_t - \gamma . update_{t-1}$

$update_t = \gamma . update_{t-1} + \eta . \nabla\theta_{lookahead}$.

$\theta_{t+1} = \theta_t - update_t$

lec 5.6: Stochastic and Mini batch Gradient Descent:

GD calculates the gradient over the complete data points and then makes the update. The gradients in this case are true gradient.

In Stochastic GD, the gradient is calculated over a single data point and update is made. Here, the loss in every step may not decrease. We will have oscillations in loss during training.

In batch GD, we take a random batch of data points and calculate gradients (close to true gradients) and update is done. It has faster convergence than GD. It has lower oscillations than SGD.

Lec 5.7: Tips for adjusting learning rate and momentum:

Learning Rate annealing:

- Step decay
- Half the learning rate after an epoch if the valid. loss is more than that at the prev. epoch.

Suboptimal Soln:

- To tackle local minima, start training again with different initialization and find the optimal loss. Repeat it several times and choose the one with min. loss.

Momentum:

- We start with a small value of γ (discounting more on history). After every step update, γ is increased (having more faith in history)

Lec 5.9: Gradient Descent with adaptive learning Rate

When any of the features is sparse (zero for most data points), the weights corresponding to that input feature will not get sufficient updates compared to dense features. This sparse feature could be very important and must be learnt properly. We need to increase the learning rate for sparse features.

Decay the learning rate according to the parameters update history

Adagrad:

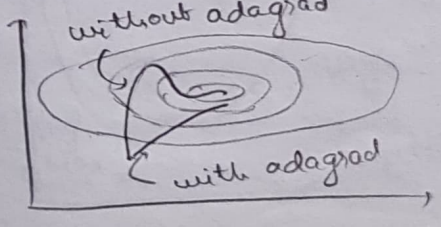
$$v_t = v_{t-1} + \overbrace{(\nabla \theta_t)^2}^{\text{accumulate the magnitude of gradient}} \quad \leftarrow \text{history}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla \theta_t$$

$v_t \uparrow \Rightarrow$ learning rate $\downarrow$

$\epsilon \rightarrow$ small number

Eg:- Consider 1000 data point (x, y) pairs with x being 0 80% of the times. $\Rightarrow$ $x \rightarrow$ sparse feature
$y \rightarrow$ dense feature



without adagrad, initially updates in x-direction were little (sparse) updates occur along y and then slowly along x.

with adagrad, the learning rate for x-direction is large due to small history. Thus, the updates occur towards the optima but due to large no. of updates along y-dir. the history becomes large and hence learning rate becomes very small. Thus we are not able to reach the minima in y-dir.

* Adagrad decays the learning rate very aggressively.

RMS Prop:

Here, we update the history slowly. Thus, the learning rate decay is slow.

exponentially weighted avg. $\rightarrow v_t = \beta v_{t-1} + (1-\beta)(\nabla \theta_t)^2$ $\quad 0.95 \le \beta \le 1$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} (\nabla \theta_t)$$

with RMS Prop, we will reach the minima.

Adam: Adaptive Moments Estimation

Here, we add momentum to the RMS prop.

$m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla \theta_t$

↑ running avg of the gradients (mean) (moment)

$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla \theta_t)^2$

↑ update history

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

↑ Bias correction ↑

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

* we take the mean of the gradient avg it is better to update according to avg. behaviour of gradients.

* we want $E[m_t]$ to be equal (near) to $E[\nabla \theta_t]$. For that we do the bias correction.

~~Lec 8.1: Eigen values and Eigen vectors:~~

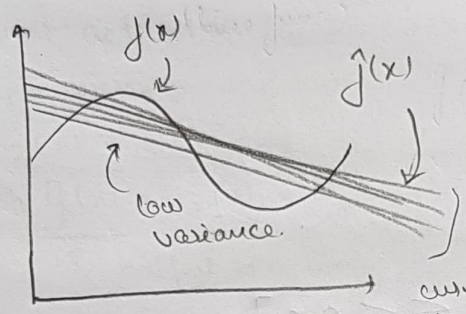
Lecture 8.1: Bias and Variance

Consider 25 points randomly sampled from a sinusoidal curve is used to train a model.

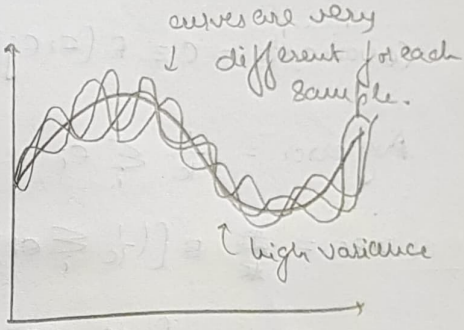
We have two models: Degree 1 polynomial

Degree 25 Polynomial.

If we sample k times and train we will obtain different curves each time.



Simple model (underfit)



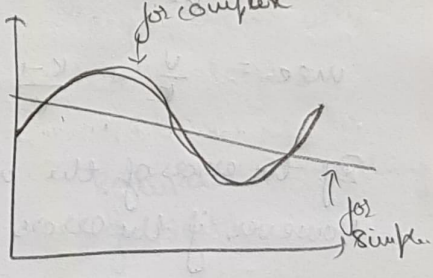
Complex model (overfit)

$$\text{Bias} = E[\hat{f}(x)] - f(x)$$

expected value from approximated func.

* High for simple model.
(low for complex "

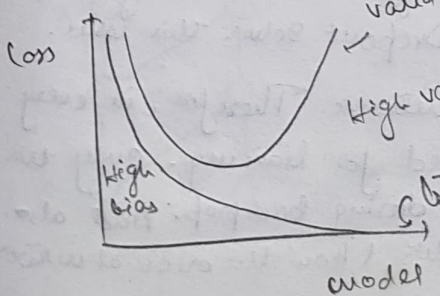
$$\text{Variance} = E\left[\hat{f}(x) - E(\hat{f}(x))\right]$$



Lecture 8.2: Train loss vs Valid. Loss

$$\text{Loss} = E[(y - \hat{f}(x))^2] = \text{Bias}^2 + \text{Variance} + \text{some error}$$

Therefore, for minimum loss both bias and variance should be low.



Lec 8.6: Parameter Sharing & Tying:

* It is used in CNNs

Lecture 8.10:

Ensemble Methods:

Combine output of different models to reduce generalization error.

Bagging: ensemble of different instances of same classifier trained on different subsets of training data.

Why ensemble works:

Let's have k logistic regression models.
Each model makes error ϵ_i on the test example.

Variance $= V = E[\epsilon_i^2]$

Covariance $= C = E[\epsilon_i \epsilon_j]$

Avg error $= \frac{1}{k} \sum_i \epsilon_i$

$$mse = E\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right]$$

$$= \frac{1}{k^2} E\left[\sum_i \sum_{i=j} \epsilon_i \epsilon_j + \sum_i \sum_{i \neq j} \epsilon_i \epsilon_j\right]$$

$$= \frac{1}{k^2} E\left[\sum_i \epsilon_i^2 + \sum_i \sum_{i \neq j} \epsilon_i \epsilon_j\right]$$

$$= \frac{1}{k^2} \sum_i E[\epsilon_i^2] + \sum_i \sum_{i \neq j} E[\epsilon_i \epsilon_j]$$

$$mse = \frac{1}{k^2}(kV + k(k-1)C)$$

$$mse = \frac{V}{k} + \frac{k-1}{k} C$$

If the errors of the model are perfectly correlated $C = V \Rightarrow mse = V$
However, if the errors ϵ_i are uncorrelated $C = 0 \Rightarrow mse = \frac{V}{k}$

Thus error decreases with ensemble as different models trained on different subsets of data will have uncorrelated errors.

Lec 8.11:

Dropout:

*Training several instances of network on different subsets of data is expensive. Even testing is expensive. Dropout solves this issue.

Dropout drops nodes randomly in the network. Therefore, in every iteration, different networks will be used for training. Only the weights for the active nodes are updated during backprop. And also, different networks share the same weights (from the original network)

So, even if a particular network gets sampled only a few times, its weights get updated by the other networks (weight sharing).

Since, each node was present with a prob. p, we scale the weights by p during testing (we only have p confidence)

Dropout also prevents the neurons from depending on other neurons. The neighbouring neurons in a layer can be dropped, so the active neurons must learn to improvise. The neurons are prevented from co-adapt.

Lec 9.3: Better Activation Func:

Without activation func:

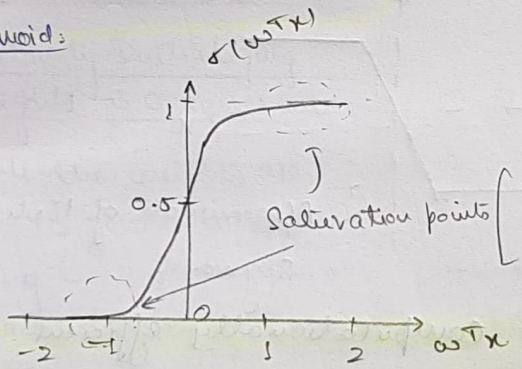
$h_1 = w_1 x + b_1$

$h_2(x) = w_2 h_1 + b_2$

$\hat{y}(x) = h_3(x) = w_3 h_2 + b_3$

↑ output is a linear func of x.

Sigmoid:



When $\nabla \to 0$

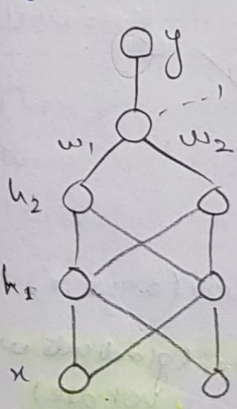
$\theta_{t+1} = \theta_t - \eta \nabla$

$\Rightarrow$ no weight update.

Part 2

* At the saturation points, the gradient of $\sigma(x)$ will be close to zero (vanishing gradients)
So, if in any layer if the weights become large then $w^T x$ becomes large.
$\sigma(w^T x)$ can be saturated (vanishing gradient)

* Sigmoid is also not zero centered.



$a_3 = w_1 h_{21} + w_2 h_{22}$ (common)

h_{21}, h_{22} → both +ve as output of $\sigma(x)$ is between $[0,1]$

$$\nabla w_1 = \frac{\partial L(w)}{\partial y} \frac{\partial y}{\partial h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_1}$$

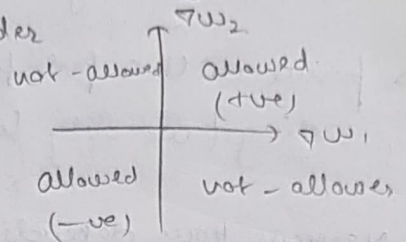
$$\nabla w_2 = \frac{\partial L(w)}{\partial y} \frac{\partial y}{\partial h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_2}$$

So, If this is +ve
$\Rightarrow \nabla w_1, \nabla w_2 \to$ +ve

If −ve $\Rightarrow \nabla w_1, \nabla w_2 \to$ −ve.

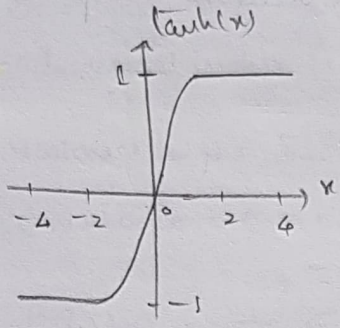
Thus all the gradients of a layer are either +ve or -ve. Restricts update direction (slow convergence).

Eg:- For 2nd order



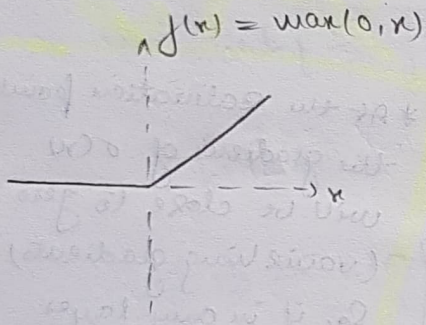
* Sigmoid is computationally expensive due to exp(x).

Tanh:

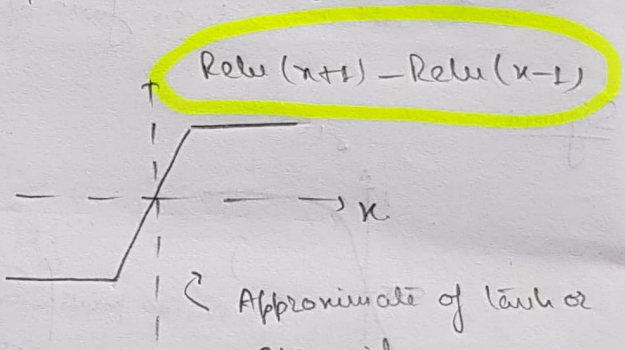


* Zero centred [-1, 1]
* Vanishing gradient at saturation.
* Computationally expensive: exp(x)

ReLU:



How, is it non-linear?



Approximate of tanh or sigmoid.

* If a large gradient (due to large learning rate) causes the bias to become a large -ve value.

Relu($w_1x_1 + w_2x_2 + b$)

Σ normalized

Thus input can be -ve.

This is called dead neuron. (forever)

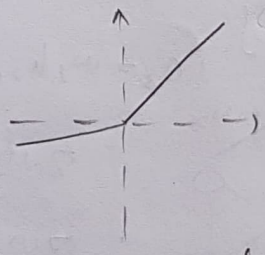
After this, w_1 and w_2 (weights) will not get updated (0 gradient), thus the neuron will remain dead.

* If the learning rate is high.

$$\frac{\partial\, Relu(x)}{\partial x} = \begin{cases} 0 ; & x<0 \\ 1 ; & x>0 \end{cases}$$

* Computationally efficient
* Does not saturate in +ve region.
* Faster convergence
* Can lead to dead neurons.
  - derivative is 1 for +ve

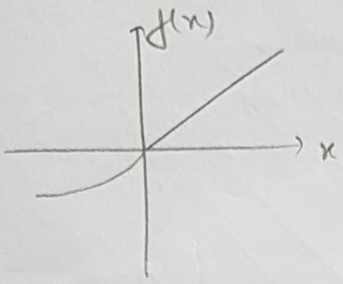
Leaky ReLU:



$f(x) = max(0.01x, x)$

* Not saturate (gradients will not die)
* close to zero centred.

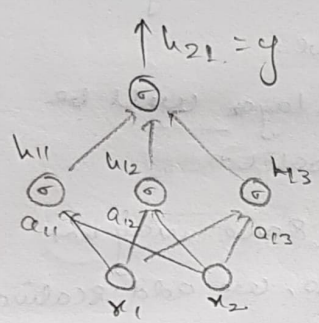
* ELU: Exponential Linear Unit:



* Benefits of ReLU
* Small gradient in the -ve side.
* Expensive : exp (x)
* close to zero mean output

Lec 9.4 : Better Initialization Strategies.

Initializing weights to zero:



Here; $a_{11} = w_{11}x_1 + w_{12}x_2$

$a_{12} = w_{21}x_1 + w_{22}x_2$

If, all weights are zero.

$a_{11} = a_{12} = 0 = a_{13}$

$\Rightarrow h_{11} = h_{12} = h_{13}$

Now; $\nabla w_{11} = \frac{\partial L(w)}{\partial y} \times \frac{\partial y}{\partial h_{11}} \times \frac{\partial h_{11}}{\partial a_{11}} \cdot x_1$

$\nabla w_{12} = \frac{\partial L(w)}{\partial y} \times \frac{\partial y}{\partial h_{12}} \times \frac{\partial h_{12}}{\partial a_{12}} \cdot x_1$

Hence, $\nabla w_{11} = \nabla w_{12} = \nabla w_{13}$

~~In the next iteration~~, Thus the updates to all the neurons in a layer will be same because the gradients will be same.

Thus the weight update is restricted.

* In general initializing weights to even non-zero the same value, we will have symmetry breaking problem.

Randomly Initializing weights to a small value

If all the weights are close to zero $w^T x \approx 0$

During back prop. $\nabla \propto w^T x$

gradient $\propto$ input. Gradients vanish.

Randomly initializing weights to a large value.

$w^T x \rightarrow$ large

If each $w_i \approx 1$, $\sum w_i x_i$

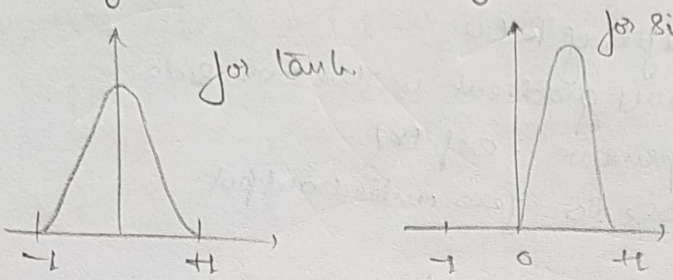
Neurons will saturate (Vanishing gradients)

* Therefore draw the weights from a unit Gaussian and scale them by $1/\sqrt{n}$. This way the variance in the output of a layer does not blow up or shrink.

Here, $n \rightarrow$ no. of neurons in the given layer.

* weights depend on the no. of neurons in the layer. Since, the weights are divided by n, the can't ~~blow~~ blow up.

* weights should have a good distribution. (with sufficient variance)



Lec 9.5: Batch Normalization:

In mini-batch GD, the distribution of the output of each layer keeps changing across mini batches. Training becomes difficult. Even if the ~~original~~ inputs to the first layer is unit gaussian, the input to subsequent layers may not be.

So, we normalize the outputs of each layer will be made unit gaussian over the current mini batch.

But, forcing the distribution to be same may not always be good (same variance & bias). So, we add scaling and shifting operation (learnable parameters) for the network to decide what is best.

$$y_k = \gamma_k \hat{x}_k + \beta_k$$

(learnable)

Deep Learning NPTEL

Lec 11.1: The conv. operation:

$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ → Blur (averaging)

$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ → Sharpen (Subtract the neighbours and emphasise the centre pixel)

$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ → edge detector (evaluates to zero when neighbour pixels are same as centre pixel)

* For 3d image, kernel will also be 3D and usually has the same depth. Then, the output of conv. will be a 2D image (feature map).
* we can have k-filters for k different feature maps.

Lec 11.2: Input and output size relation:

* For a filter size F, and stride = 1, output dim. decreases by (F−1)

$$\Rightarrow W_{out} = W_{in} - (F-1)$$

* If we want the same output dim, we pad input,

$$W_{out} = W_{in} - F + 1 + 2P$$

* For stride S,

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1$$

Lec 11.3: CNN:

SIFT, HOG were dominant at that time.

* Earlier people used convolution to find edges etc. in an image, flatten them manually and feed them to a DNN for image classifications. The filters were hand-crafted.

Instead, we can learn these filters and have multiple layers of these filters. This is CNN.

* CNN have sparse connection, weight sharing (as the same kernel is used for the entire image)
* Since the kernel size is small, under fitting is a problem. So, we have multiple kernels.
* Weight sharing results in much less no. of parameters.
* Maxpooling has no parameter. It is just a max operation.
* In CNNs, Dense connections (FC layers) have large no. of parameters.

Eg:- Significance of weight sharing:

Input = 32×32

output = 28×28

FC-layer: $32^2 \times 28^2$

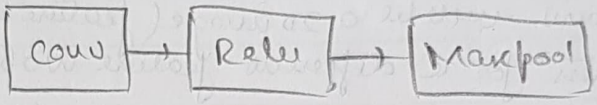
CNN: $5 \times 5 \times 6 = 150$

$\begin{bmatrix} F=5 \\ K=6 \\ S=1 \end{bmatrix}$

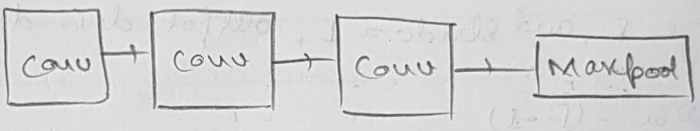
Imp: when we count the no. of layers in a Network, we only count those that have parameters.

Thus, Maxpool is not counted.

Imp: Non-linearity should be done after convolution and before Maxpool



* we can have back to back conv layer and then a maxpool (VGG style)



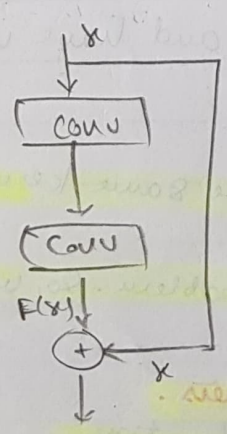
Googlenet:

* uses inception modules.

Inception modules first use 1×1 conv to reduce the no. of channels (depth), then they compute different convolutions with different filter size in parallel and concat the outputs. This lets the network to utilize the good of different filter sizes.

Resnet:

A shallow network works better than a deep network because it is very difficult to learn identity mapping by conv. nets. Soln. is to use skip connections. This way a superset of a good working network will perform at least as good as the shallow network.



$H(x) = F(x) + x$

Eg:- Significance of weight sharing:

Input = 32×32

Output = 28×28

FC-layer: $32^2 \times 28^2$

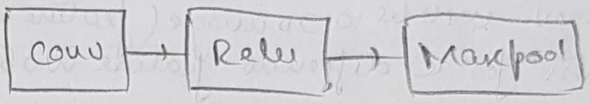
CNN: $5 \times 5 \times 6 = 150$

$\begin{bmatrix} F=5 \\ K=6 \\ S=1 \end{bmatrix}$

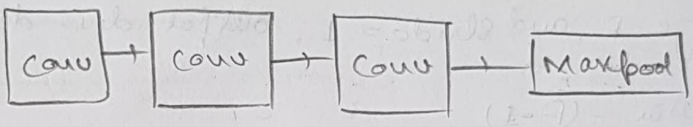
Imp: When we count the no. of layers in a N/W, we only count those that have parameters.

Thus, Maxpool is not counted.

Imp: Non-linearity should be done after convolution and before Maxpool.



* we can have back to back conv layer and then a maxpool (VGG style)



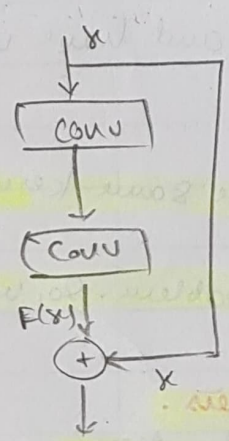
Googlenet:

* uses inception modules.

Inception modules first use 1×1 conv to reduce the no. of channels (depth), then they compute different convolutions with different filter size in parallel and concat the outputs. This lets the network to utilize the good of different filter sizes.

Resnet:

A shallow network works better than a deep network because it is very difficult to learn identity mapping by conv. nets. Soln. is to use skip connections. This way a superset of a good working network will perform at least as good as the shallow network.



$H(x) = F(x) + x$

Lec 12.3: Occlusion Expt:

It can be used to check the robustness of the network. The idea is to randomly occlude (gray out) a random portions of the image and check for prediction.

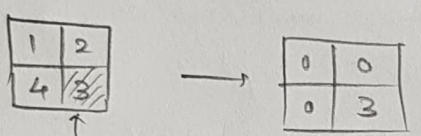
For eg:- a good face detector should work even if we cover the nose. (Should not depend too much on nose)

* The network should not rely heavily on a single part of image.

Lec 12.5: Guided Backprop:

This is used to find the influence of input image on any neuron in the network. (basically finding out the specific portions like eyes, ears etc that the neuron responds to).

We first do a forward pass through a network. We then take a neuron



Selecting this neuron.

↑ other neurons in this layer are set to zero.

Now, backpropagate to get the reconstructed image.

While backpropagating, use ReLU just like we did during forward pass. This will stop the -ve gradients.

This method is called guided backprop. This gives a more clear influence of input image on the neuron.

Lec 12.6: Optimization over images

Suppose we have a trained network and we want to generate an image that when passed through the network maximizes the output to a certain class.

We can solve this as an optimization problem. Assume the input image to be a trainable parameter that we will update to minimize loss.

* Start with zero image
* Set the target (label) to [0 0 .... 1 ... 0] (one hot encoded)
* Back propagate and update the pixel values. $\theta = \theta - \eta \nabla \theta$
* Do a forward pass and calculate cross entropy loss.
* Repeat till convergence.

* These generated images are nowhere near to the actual image of the particular class but can be used to fool the network.

Lec 12.7: Create images from embeddings

Take an input image and do a forward pass through the network. Now, the output of each of the layers in this network can be used as embeddings to generate the original image (reconstruction)

To regenerate, put a blank image at the input and the loss will be MSE between the embeddings and output of the chosen layer. Now, use GD to update pixel values of the input image. As we go deeper into the network, the reconstructed image starts to look blurred and weird.

Lec 12.8: Deep Dream:

Basically, we start with some input image (eg:- sky) and we want to maximize the firing of a certain neuron in some layer. So,

objective func ~~loss~~ = $(h_{ij})^2$

to be maximized — output of the neuron.

Now, if we update the image to maximize the objective func, the network will draw ~~cats~~ certain objects in the input image in correlation to the input image.

Eg:- for sky, birds and castles may be formed and for grasses, certain ~~animals~~ animals may be formed.

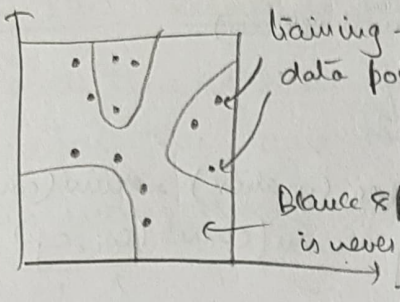
Basically, when a network sees an input image (eg sky), it expects to see birds or castles. If they aren't present, it draws them.

Lec 12.10: Fooling Deep convnets:

To fool a network, we either take a blank image or an image of one of the classes eg. bus and then update this image slightly using back prop to fit some other class eg. cat.

This image will still look the bus, but the network will predict it as cat. We can start from blank image and add the updated img. to bus image to obtain the same effect.

Why this happens is explained by Karpathy. Images have very high dimensionality eg (224 x 224) and in any dataset we only see a very small subset of this space. Using the training data, we fit some decision boundary but we leave out the other possible images. We generalize the output (decision) for a large no. of points that we haven't seen.



but we still decided which class it belongs to. Since only 1-bit is hot

Lecture 10.1: One hot representation of words (Sparse representation)

Given a corpus (collection of sentences), the set of all the possible words in it is called the vocabulary 'V'. Each word in the vocabulary can be represented by one-hot encoded vector of length V.

Disadvantage:
- One-hot encoding of words does not capture similarities between words.

  Eg:- euclid_dist (cat, dog) = $\sqrt{2}$.
  euclid_dist (dog, breed) = $\sqrt{2}$ } same for all pairs.

  cosine_sim (cat, dog) = 0
  cosine_sim (cat, breed) = 0 } orthogonal pairs.
- As V increases, large space is required.

Lec 10.2: Distributed Representations of words:

Co-occurrence Matrix:

	human	machine	system	...	user
human	0	1	0	...	0
machine	1	0	0	...	0
system	0	0	0	...	2
:	:	:	:	:	:
user	0	0	2	...	0

Context

Words

rows = length of vocab.

dimn. [words x context]

Columns = length of vocab.

Cooccurrence matrix captures the no. of times a word appears in context of another word. Context here is defined as a window of k words around that word.

* A few columns can be dropped as words like {a, for, the, etc} don't carry much info. (stop words → their count is very high.)

Now, each row can be taken as a vector representation of the word. This representation is still sparse (very few non-zero columns)

Some fixable Problems

Stop words are very frequent ⇒ count is large (it will skews)

Soln 1: Ignore high count words.

Soln 2: use a threshold t, every entry in the table (matrix) = min(count, t)
⇒ $x_{ij} = \min(\text{count}(w_i, c_j), t)$
(word, context)

Soln 3: Instead of count, use PMI.

$$PMI(w,c) = \log\left[\frac{count(w,c) * N}{count(w) * count(c)}\right]$$ N → total no. of words.

If two words occur frequently independently, but rarely together
⇒ PMI is low.

PMI will be high for very frequently co-occuring words.

Another variant,

Positive PMI (PPMI) = PMI(w,c) if PMI(w,c) > 0
= 0 ; otherwise

* we are facing bigger problems:
- very high dimensionality (V)
- very sparse
- grows with the size of vocab.

Soln use dimensionality reduction (SVD) Singular value Decomposition:

* PCA only works for Sq. matrix
* SVD is a generalization of PCA

Lec 13.1 Sequence Learning Problems:

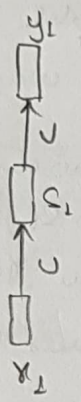
In feed forward NN and CNN, we had inputs of fixed size and independency on the previous inputs.

When dealing with sequences eg text completion, the ~~above~~ inputs may be correlated and of variable size. Each input occurs at a time step.

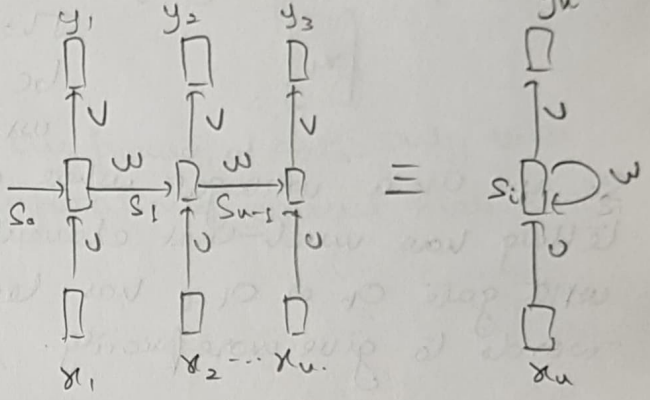
Sometimes, we produce output only at the final stage. Eg:- In video analysis, we can only predict after watching the complete video.

Lecture 13.2: Recurrent Neural Networks:

At each time step we feed an input to the NN and it gives an output. Therefore, the same network can be passed each element of the sequence. This way, the length of sequence doesn't affect the network



$$s_i = \sigma(Ux_i + b)$$
$$y_i = O(Vs_i + c)$$



Add recurrent connection to enable dependency between inputs.

(y_n through s_{n-1}, y_n depends on all the prev. inputs)

s_i: State of the network at time 'i'

(encodes all the inputs information so far)

$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$
$$y_i = O(Vs_i + c)$$

Lecture 13.3: Back prop through time:

Lec 13.4: Exploding and Vanishing gradients:

When back propagating through a ~~long~~ long sequence, a small change in the gradient at each time step (> or less than 1) can cause the gradient to explode or vanish.

To avoid this, we backpropagate only till k time steps (truncated backprop)

To avoid exploding gradients, we can clip the magnitude.

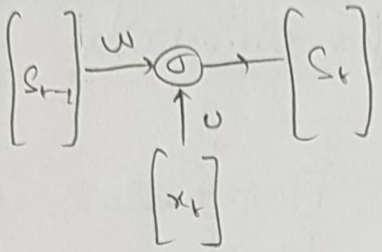
Lec 14.1: Finite state size

The state s_i of an RNN records information from all prev. time steps. Due to fixed ~~size~~ size of s_i, old information gets morphed. So, old information becomes impossible to extract. Thus the morphing should be done in a controlled way.

Lec 14.2: LSTM and GRU:

Consider the task of sentiment analysis of a movie review. The network (RNN) will read the review from left to right. By the time we reach the end of the review, information about the first few words will be lost (finite state size). So, ideally, our goal is to save important words in the state and forget stop words.

At every time step 't', we want to compute s_t given s_{t-1} and the input at current time step 'x_t'.



$s_t = \sigma(W s_{t-1} + U x_t)$ (ignoring bias)

The problem here is that the complete s_{t-1} will be used to compute s_t. But we want selective write.

So, we use a write gate where each element is a value between 0 and 1, telling how much that element of s_{t-1} should be sent to s_t. The write gate o_t or o_{t-1} has learnable parameters. So, it can learn which words to give more priority. [Sigmoid is used to get values between 0 & 1]

$$\begin{bmatrix} s_{t-1} \end{bmatrix} \odot \begin{bmatrix} o_{t-1} \end{bmatrix} = \begin{bmatrix} h_{t-1} \end{bmatrix}$$

Selective write

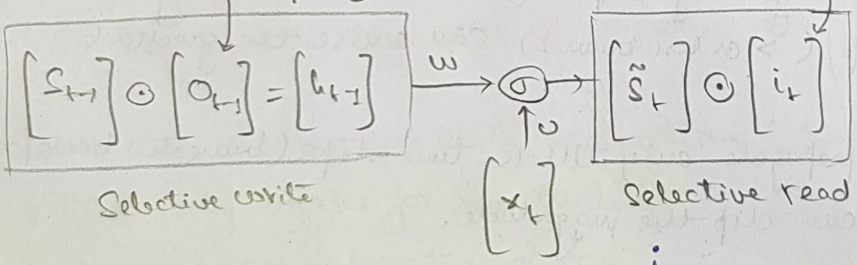
$$o_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$$

learnable parameters

$$h_{t-1} = o_{t-1} \odot s_{t-1}$$

↑ element wise multiplication

* Using output gate, we have created a model which can forget un important words.



Now, we calculate $\tilde{s}_t$ which captures complete information from h_{t-1} and x_t. So, we selectively read using

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

Now, we have s_{t-1} and $\tilde{s}_t \odot i_t$, we need to combine them to get s_t. Instead of using s_{t-1} completely, we forget some parts of it (forget gate)

$$\underbrace{\begin{bmatrix} s_{t-1} \end{bmatrix} \odot \begin{bmatrix} o_{t-1} \end{bmatrix} = \begin{bmatrix} h_{t-1} \end{bmatrix}}_{\text{Selective write}} \xrightarrow{W} \sigma \xrightarrow{} \underbrace{\begin{bmatrix} \tilde{s}_t \end{bmatrix} \odot \begin{bmatrix} i_t \end{bmatrix}}_{\text{Selective read}} + \underbrace{\begin{bmatrix} s_{t-1} \end{bmatrix} \odot \begin{bmatrix} f_t \end{bmatrix}}_{\text{Selective forget}} = \begin{bmatrix} s_t \end{bmatrix}$$

(U from x_t)

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

$$s_t = f_t \odot s_{t-1} + \tilde{s}_t \odot i_t$$

Therefore, in a typical LSTM, we have 3 gates (learnable)

* Sometimes s_t is notated as c_t (cell state)

Gated Recurrent units (GRU):

One variant of LSTM is GRU where the input and forget gates are tied together. The idea here is that, we take it from $\tilde{S}_t$, then just take $(1 - i_t)$ from S_{t-1} as output.

$$[S_{t-1}] \odot [O_{t-1}] \xrightarrow{W} (\sigma) \xrightarrow{U [x_t]} [\tilde{S}_t] \odot [i_t] + [S_{t-1}] \odot [1 - i_t] \longrightarrow [S_t]$$

*LSTM prevents vanishing gradients due to the presence of gates. Only those prev. states which contributed to the computation of current state will be used for backprop.

Lec 15.1: Intro to Encoder Decoder Models:

Language modelling:

Given 't-1' words, predict 't' word (Text completion)

y_{t-1} contains information till (using RNN)

$$y^* = \text{argmax}\, P(y_t | y_1, y_2 \ldots, y_{t-1}) = \text{argmax}\, P(y_t | s_t)$$

we basically want to find a prob. dist. over the vocab and find the max.

Compact way of writing:

$$s_t = RNN(s_{t-1}, x_t)$$
$$s_t = GRU(s_{t-1}, x_t)$$
$$u_t, s_t = LSTM(u_{t-1}, s_{t-1}, x_t)$$

Image captioning:

Here, we are interested to find,

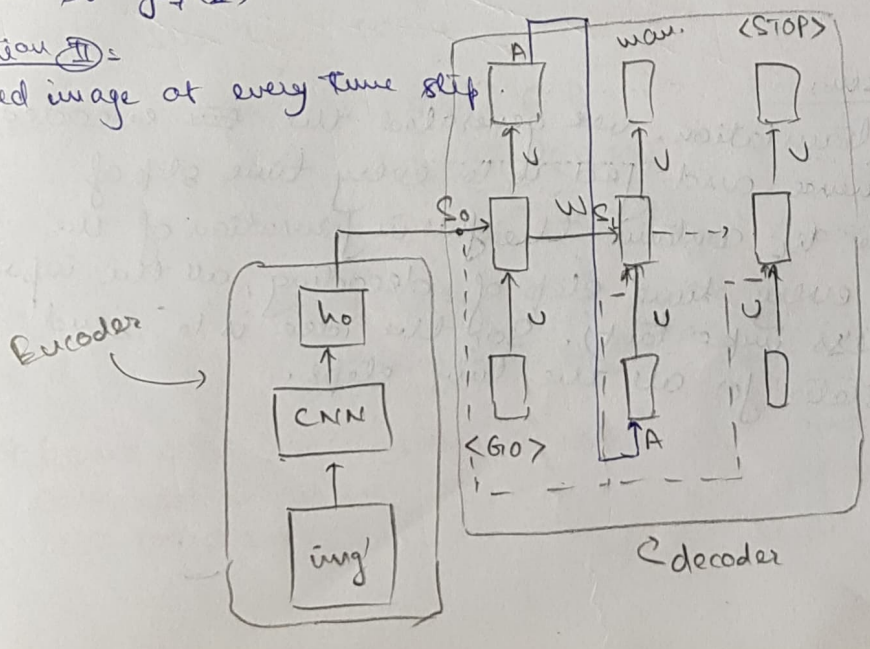
$$P(y_t | y^{t-1}, I) = P(y_t | s_t, \underbrace{f_{c_7}(I)}_{\text{abstract repr. of image (7th layer)}})$$

Option (I):

Set $s_0 = f_{c_7}(I)$

Option (II):

feed image at every time step



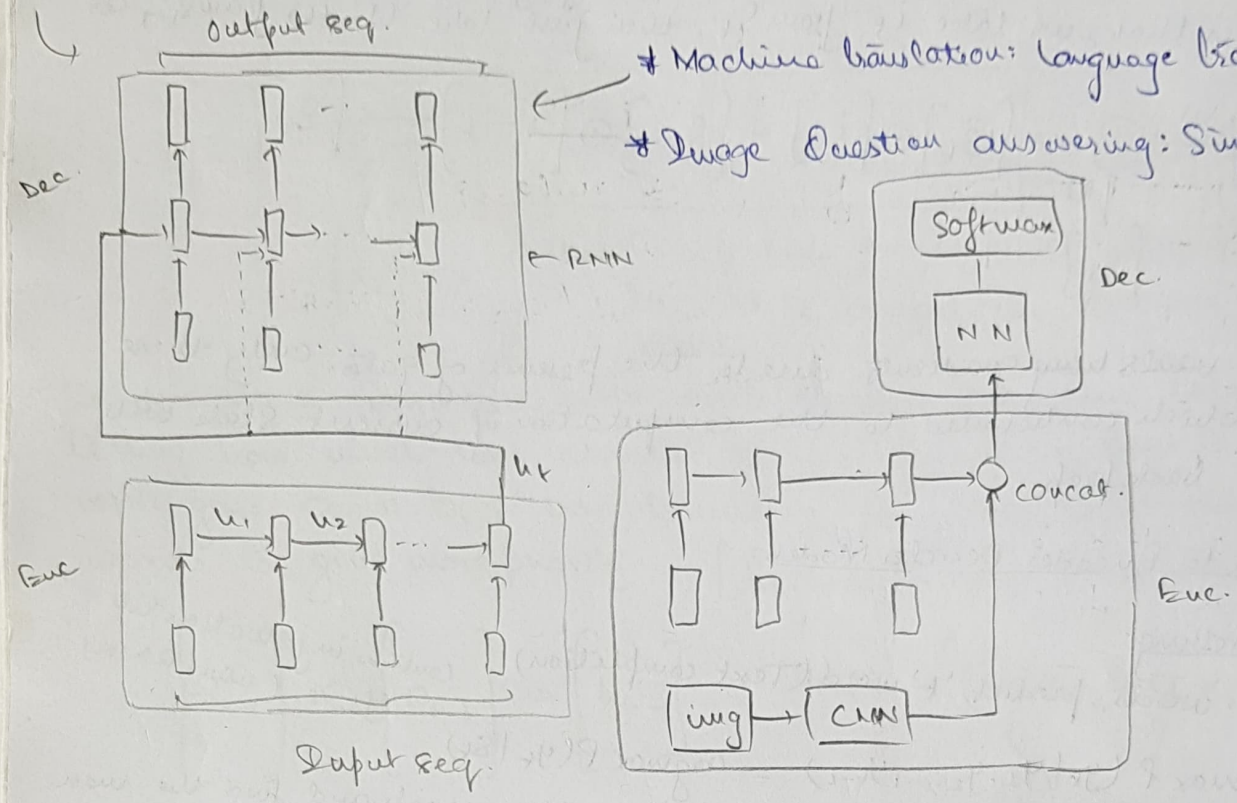
* To train the RNN, the algorithm used is backpropagation through time (BPTT).
* The loss used is sum of cross entropy.

Lec 15-2: Applications of Encoder-Decoder Models:

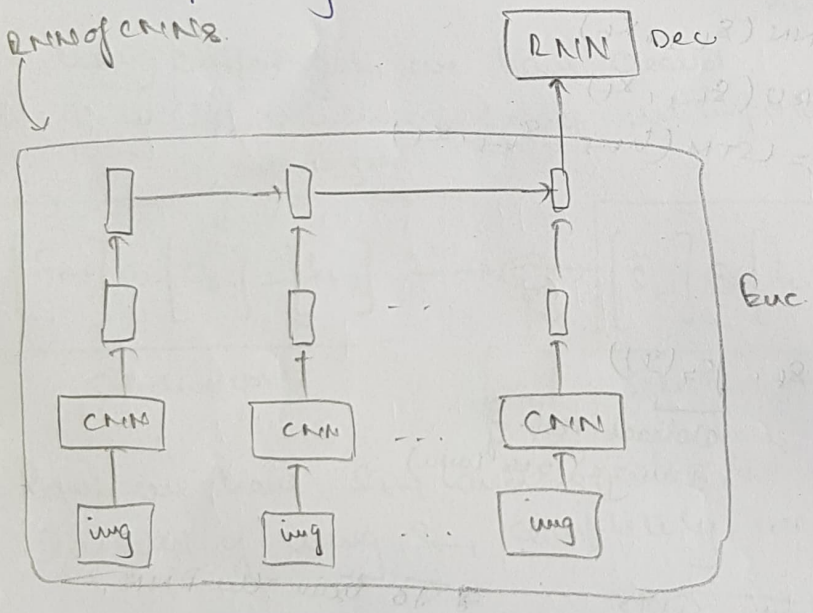
* Textual Entailment: Input: sentence output: Implying sentence

* Machine translation: Language translation

* Image Question answering: Single word output.

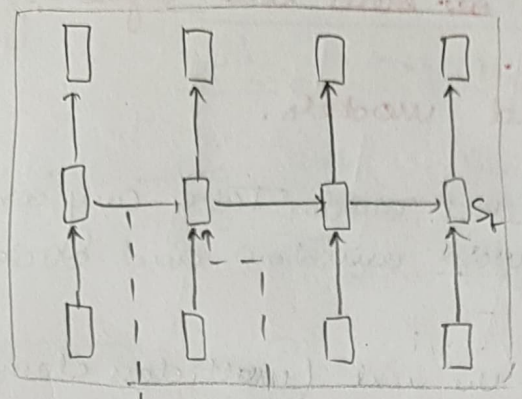


* video captioning:



Lec 15-3: Attention Mechanism:

In the example of machine translation. we generated the encoded state h_t of the input sequence and fed it to every time step of decoder. This encoded state h_t contains the full information of the input sequence. However, at every time step of decoding, all the input words are not required (less important). So the idea is to send a weighted sum of input states for all the time steps.



learnable func

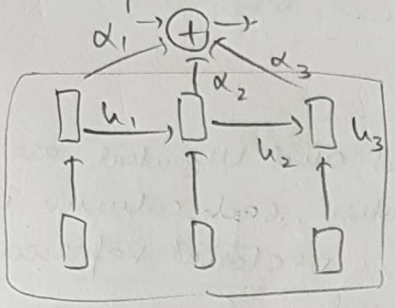
$$e_{jt} = f_{ATT}(s_{t-1}, h_t)$$

$$\alpha_{jt} = \frac{\exp(e_{jt})}{\sum_j \exp(e_{jt})} \leftarrow \text{Softmax}$$

α_{jt} ← j^{th} word

$$e_{jt} = v_{att}^T \tanh(W_{att} s_{t-1} + W_{att} h_j)$$

e_{jt} ← scalar.



* When back propagating, these parameters will learn and hence, the attention vectors will learn too.

Continuation:

10.3: SVD for learning word representation:

SVD gives the best rank-k approximation of the original rank-n matrix. 'k' here is variable, we can decide what value we want. closer 'k' will be to 'n', more the accuracy.

SVD also brings out latent co-occurrence between the words.

Eg:- user - human has value = 0 in co-occurrence matrix(X), but since these two words occur in context with words like "install" etc, in the low rank matrix, their value is non-zero. This hidden co-occurrence becomes visible.

* After doing SVD, the matrix is still of the same size, it is just an approximation (best).

* If we do XX^T of the original matrix, each value is a cosine similarity, after SVD this cosine similarity increases (learning better co-occurrence).

From SVD, $\hat{X} = (U \Sigma V^T) \in R^{m \times n}$

Now; $XX^T = (U \Sigma V^T)(U \Sigma V^T)^T$

$= U \Sigma \Sigma^T U^T \quad (\because V^T V = I)$

$= (U\Sigma)(U\Sigma)^T = W_{word} W_{word}^T \Rightarrow \boxed{XX^T = W W^T}$

$W_{word} = U\Sigma \in R^{m \times k}$

Thus, cosine similarity is maintained and W_{word} is the representation of m-words in the vocab.

* Remember, k is very small compared to m & n.
k ≈ 100 or 200.

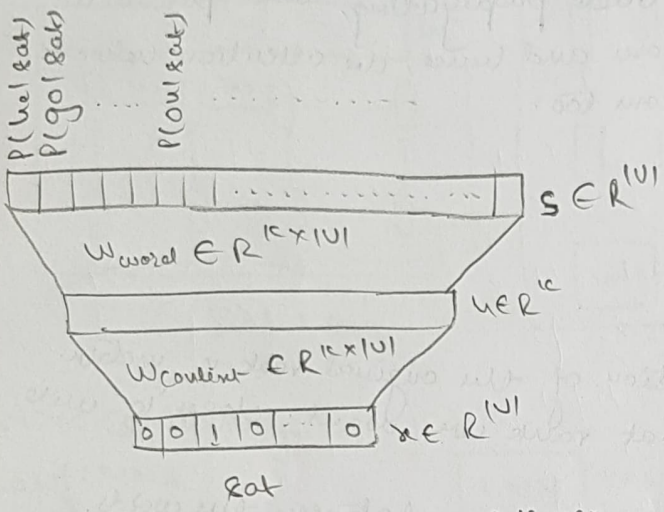
Also, $W_{context} = V \in R^{n \times k}$ ← representation of context words.

Lec 10.4: Continuous Bag of words model:

*Till now, whatever word representations we ~~~~ BOW was before deep learning approaches. (count based models).
Now, we will learn prediction based models.

Consider the task of predicting n^{th} word, given $n-1$ words. Take any written material eg:- stories, ~~~~ blogs etc, make a n-word window and slide over it for training.

Let $n=2$. $\Rightarrow$ given one word predict the next (multiclass classification)



Here, W_{word} and $W_{context}$ are $k \times |V|$ matrix where, each column is a k-dimn. vectorial representation of the word. These will be learnt

$$h = W_{context} \cdot x = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}$$

Thus, if input is j^{th} word, then j^{th} column of the $W_{context}$ is selected.

Now, to get $P(word=i \mid sat)$ ~~~~ Softmax

$$= \frac{e^{(W_{word} \cdot h)[i]}}{\sum_j e^{(W_{word} \cdot h)[j]}}$$

Thus, $P(word=i \mid sat)$ depends on j^{th} column of $W_{context}$ and i^{th} column of W_{word}. Thus, i^{th} column of W_{word} is the vectorial representation of word 'i'.

Let's see how this will learn good word representations:

Let's denote context word (sat) by index x. and the correct output word (on) by index w.

Output func: Softmax
Loss func: Cross entropy

$$L(\theta) = -\log \hat{y}_w$$

↑ maximize w.r.t entry in $\hat{y}$

$$u_c = h = W_{context} \cdot x_c$$

↑ vectorial rep. of context word.

$\hat{y}_w = \dfrac{\exp(u_c \cdot v_w)}{\sum_{w' \in V} \exp(u_c \cdot v_{w'})}$ ← vectorial rep. of w'th word. (output)

$\Rightarrow L(\theta) = -\log \dfrac{\exp(u_c \cdot v_w)}{\sum_{w' \in V} \exp(u_c \cdot v_{w'})} = -(u_c \cdot v_w - \log \sum_{w' \in V} \exp(u_c \cdot v_{w'}))$

Now; $\nabla_{v_w} = \dfrac{\partial}{\partial v_w} L(\theta) = -u_c \left(1 - \dfrac{\exp(u_c \cdot v_w)}{\sum_{w' \in V} \exp(u_c \cdot v_{w'})}\right)$

$\nabla_{v_w} = -u_c (1 - \hat{y}_w)$

Update rule, $v_w = v_w - \eta \nabla_{v_w}$

$v_w = v_w + \eta u_c (1 - \hat{y}_w)$

Now; $\hat{y}_w = 1 \Rightarrow$ prediction is correct

$\hat{y}_w = 0 \Rightarrow$ wrong

$\Rightarrow v_w = v_w + \eta u_c$

* We are basically adding a fraction of context vector to word vector. Thus, the word representations will become closer to context rep. Thus cos-sim between v_w and u_c is maximized.

Thus, if we take two words like dog and cat, both of these words will be similar in context word like eats, sleeps etc but a bicycle will not. This however does not mean that dog and sleep, eat are similar. We only care about the word representations.

Similarly $n = d$,

for this, just concatenate the word 1-hot vectors, so, the input will be a vector of dim $|V|$ with d-elements 1

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{12}+a_{13} \\ a_{22}+a_{23} \\ a_{32}+a_{33} \end{bmatrix}$$

Sum of columns.

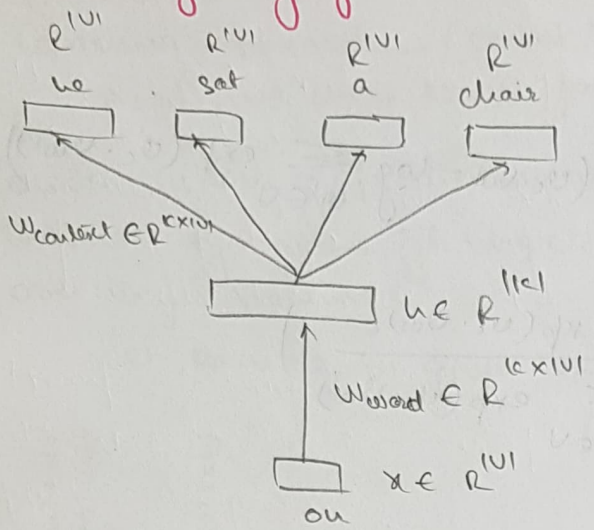
Here, the order of words in input does not matter. (bag of words)

Problem: Softmax over $|V|$ is computationally expensive.

* Bag of words model predicts the output given the context.

Lec 10.5: Skip gram model:

Reverse of bag of words: Given a word, predict all the context words.

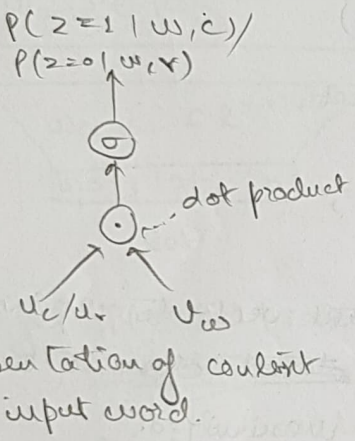


~~The~~ The sum of cross entropy will be the loss.

It has the same problem as bag of words (computationally expensive softmax)

Solns: (1) Negative Sampling
(2) Contrastive estimation
(3) Heirarchical Softmax

Negative Sampling:



representation of context & input word.

Make a set D of correct (w,c) pair and incorrect (w,r) pairs D'. r → random word.

The task now is to maximize the sigmoid operation of the dot product between input and context words. At the same time we need to make the output ≈0 for incorrect pairs.

The objective therefore is to maximize.

$$\text{maximize} \prod_{(w,c) \in D} P(z=1|w,c) \prod_{(w,r) \in D'} P(z=0|w,r)$$

* The idea here is that we are trying to bring context words close to input words. Thus, words in the same context will come closer. Additionally, we are also making the dot product (cosine similarity) between unrelated words very small. Thus unrelated words are also pushed far apart. (improvement from bag of words)

* We have '$\prod$' because we want to maximize this for all (w,c) and (w,r) pairs.

* Ingenual size of D' is k times that of D because in natural language incorrect pairs are more common that true pairs.

* Now, instead of sampling r using uniform dist, we can do it based on the freq of occurence.

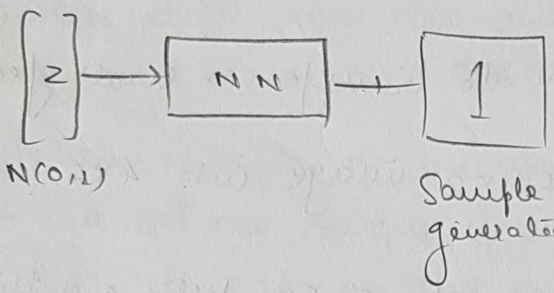
$$P(r) \sim \frac{count(r)^{(3/4)}}{N}$$

↙ hyper parameter

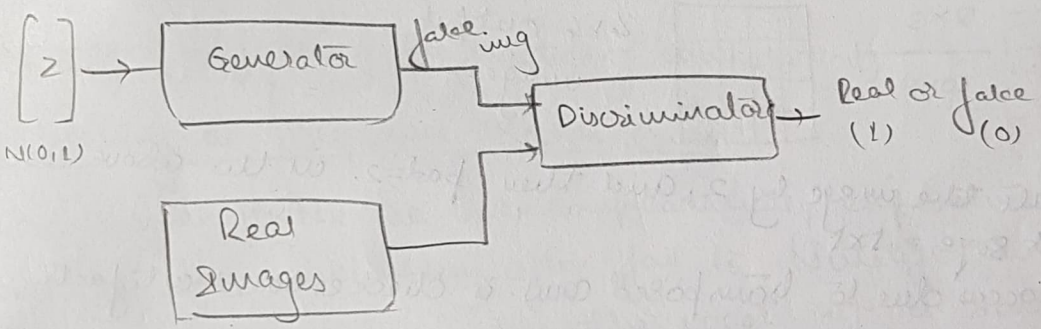
↑ total no. of words in the corpus.

Lec 22.1: Generative Adversarial Networks (GANs):

Given some dataset, the goal is to generate more data that looks like they have come from the distribution. Datasets belong to a very complex high dimensional dist. It is very complex (difficult) to sample from this very high dimensional dist. So, instead we can sample from a simple normal dist. and learn a transformation from this to the training data dist.



Architecture:



There are two networks, generator and discriminator. Generator's job is to generate better and better images that look like they have come from the same dist (training dist). The discriminator's job is to identify whether the images are real or fake (generated). The discriminator outputs 1 for real images and 0 for fake. The generator keeps trying to fool the discriminator.

$D(x) = 1$
$D(G(z)) = 0$ } for ideal disc.

Discriminator's job is easier, so it will learn quicker and disc. loss will initially be lesser than generator's loss. Then, as generator starts to learn its loss will decrease. and $D(G(z))$ will start to increase and $D(x)$ will decrease. This indicates that discriminator is having a hard time identifying (generated images are really good).

Special types of convolution:

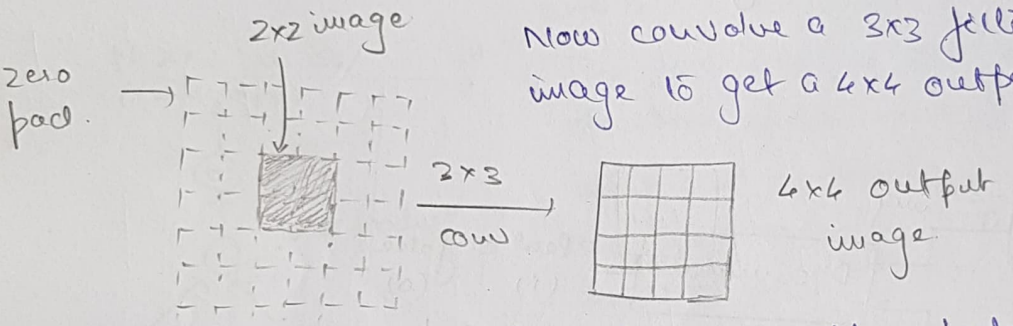
1x1 conv:

- It is used to reduce the depth of feature maps without the expensive 3x3 and 5x5 conv.
- Dimensionality reduction results in a small but dense feature maps (feature pooling)
- 1x1 conv along with ReLu can be used to learn complex functions.

Transposed Convolution:

The CNN learns the interpolation to transform a small feature map (image) into a larger one.

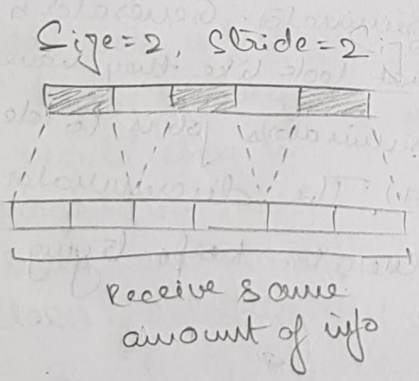
Eg:- We want to convert a 2x2 image into 4x4.

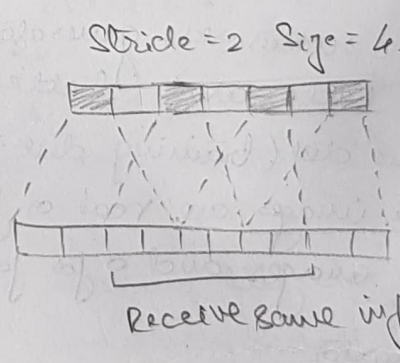


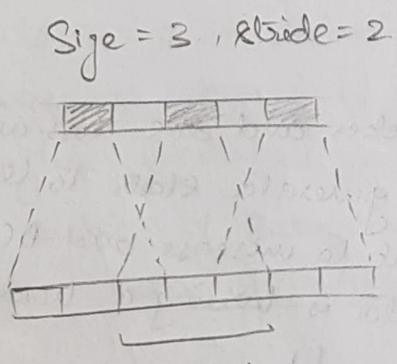
Now convolve a 3x3 filter over this padded image to get a 4x4 output image.

*If we dilate the image by 2, and then pad=2. in the above case. output size = 5x5.

A problem that can occur due to transposed conv. is checkerboard artifacts.







Every pixel in the output receives different amount of info. compared to its adjacent pixel.

To prevent filter size must be divisible by stride.

Dilated Convolution (Atrous Conv):

They are used to increase the receptive field of the kernel without increasing its size. When $d = 1$, it behaves like a normal conv.

A no. of dilated conv. can be stacked with increasing receptive field. The output of these can be used to aggregate multiscale contextual information without losing resolution.

Depthwise Separable Conv:

It consists of 2 steps depth wise conv. and 1×1 conv. They are commonly used in deep learning (MobileNet & Xception)

To a volume of depth D, we apply depth wise conv. (filter of depth 1). So, we get an output consisting of depth D. Note that D different kernels convolve on the respective channel of the input.

The output vol. of depth D is converted to depth 1 using a 1×1 conv.

Advantage: Efficiency (a fraction of computation)

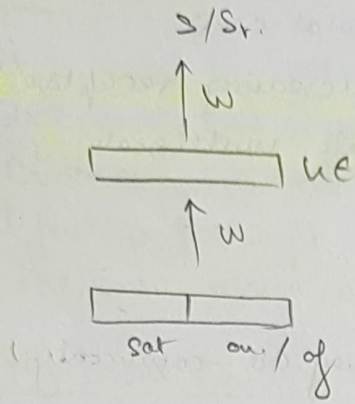
Deeplearning.ai Sequence Models Course:

* character level RNNs are more computationally expensive because of the long sequences and they often fail to utilize the correlation due to longer sequences. However, the vocab. size is small and there are no <unk> tokens needed.

* Language modelling is basically learning the probability of a sequence of words. To determine which sentence is more proper.

* RNNs suffer from vanishing and exploding gradients. Exploding gradients can simply be solved using gradient clipping. Vanishing gradients are harder to solve.

Skip-gram model (contrastive estimation):



Here, we use a NN where we pass input and context word pair of or incorrect pair. The network will output a score (single value)

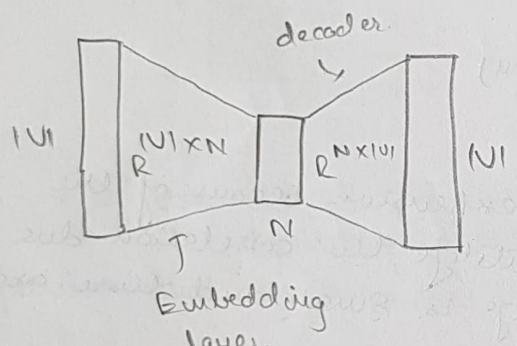
We want the score for correct pair should be at least 'm' (margin) greater than Sr.

Thus we want to maximize the...

$$\text{maximize} \left[\max(0, S - (S_r + m))\right]$$

Word 2 vec. (Skip gram model):

In skip gram model we, we use central words in the corpus to find the context word and this way we learn the embeddings. we use correct pairs only to train the network.



We input the words as 1 hot vector into the embedding layer. The output to the hidden layer is just the ith row of embedding weights if ith index is hot.

The task of the decoder is to map the embedded vector to context word. Thus words that are similar in context come close to each other.

Beam Search:

When an RNN generates a sequence of words or characters, the output of every time step may be optimal considering the part of the seq that it has already written but may not be optimal considering the full sentences.

So, instead of using greedy search at each step, we use a beam width eg:-3. (let's say we have a vocabulary of 1000 words. To choose first word, we select top 3 words.

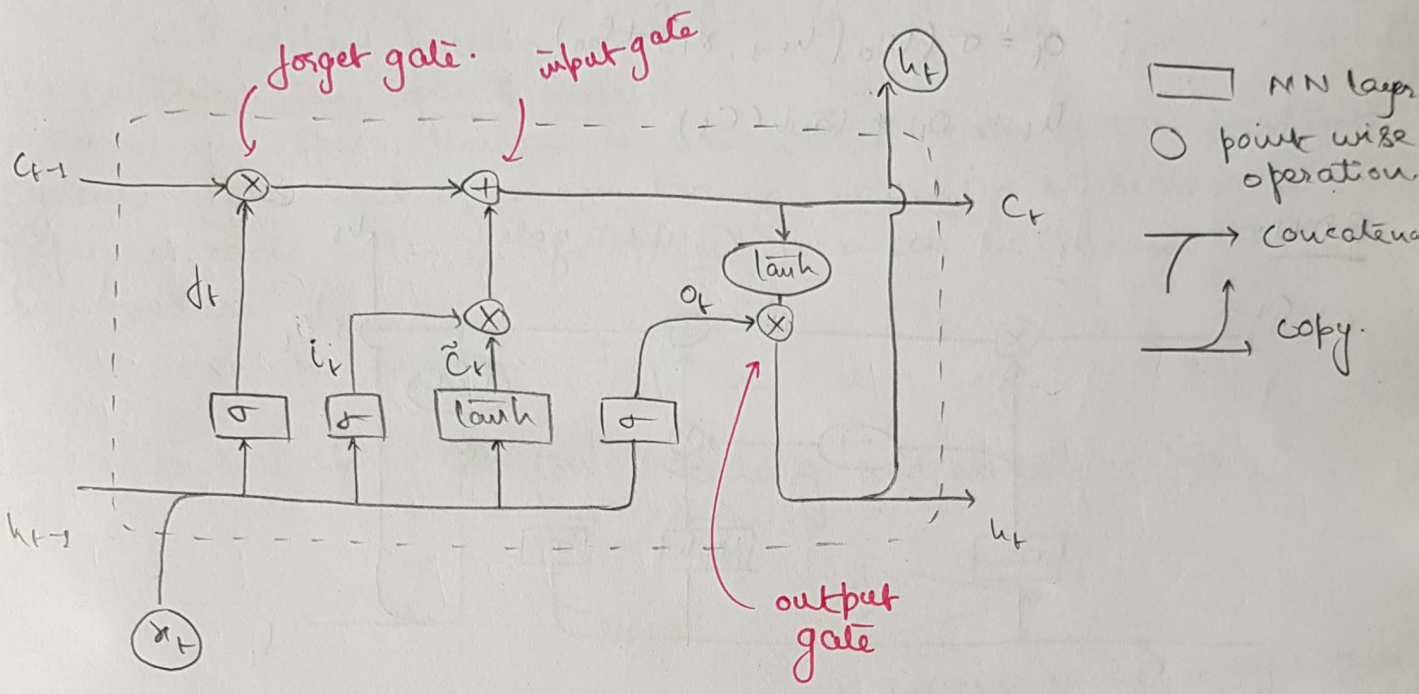
Now, input each of these to the next step, for each we get 1000 output (softmax), so total 3000.

Now, choose top 3. We now have top 3 most probable pairs (1st and 2nd word).

Now, input 2nd word into the network, again choose top 3 from 3000 outputs.

This is the top 3 combination of 1st, 2nd and 3rd word.
Keep repeating.

Christopher Olah's Blog (LSTM):



The key to LSTM is the cell state. It runs through the entire chain (time step). Each LSTM cell along the time step has the option to selectively add or remove information from this cell state. This selective operation is done through gates.

First, it decides what information to throw from the previous cell state (forget gate).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Next, we decide what new information we want to add. For this sigmoid layer (NN) generates attention map. Then, a tanh layer (NN) makes some useful transformations before adding. This new representation is then weighted by the input gate and added to the cell state.

$$i_t = \sigma(W_i [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c [h_{t-1}, x_t] + b_c)$$

Thus, the total update to C_t (cell state):

$$C_t = \underbrace{f_t * C_{t-1}}_{\text{forget information}} + \underbrace{i_t * \tilde{C}_t}_{\text{add info.}}$$

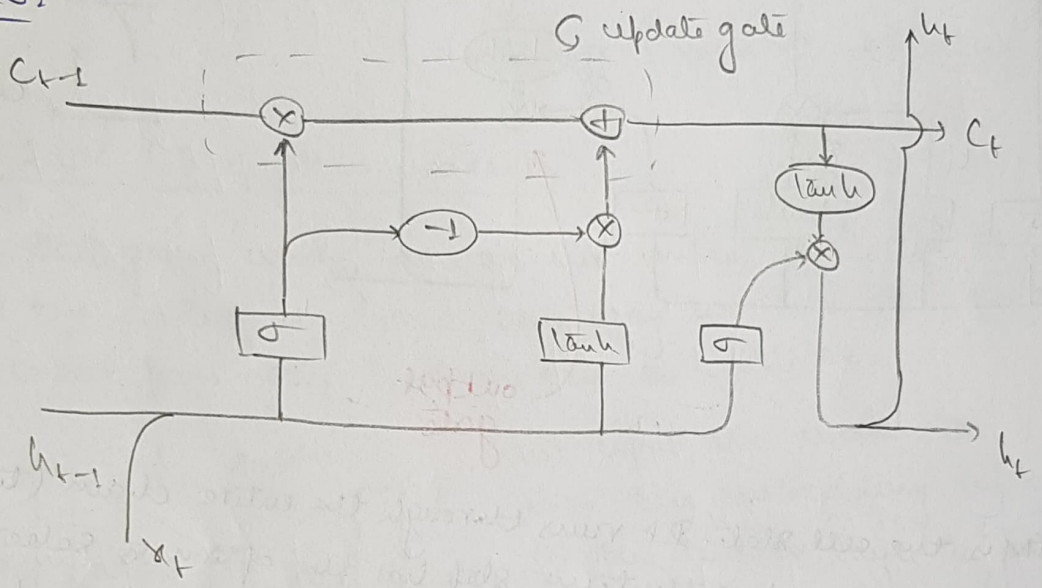
Now, the cell state captures all the essential information about the seq. over all the time steps, but we don't need all the information at each

time step to predict the output. Thus, we filter some information from the cell state using x_t and h_{t-1} to get the hidden state h_t. This h_t is then passed to a NN to output the element for that time step. This filtering is done using output gate.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

GRU:



In GRUs, the input and forget gates are combined into one update gate. Basically what we allowed to pass in the forget gate, the opposite of that must be added (info. that is forgotten from prev. step should be added in the current step)

$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

Thus GRU's have only 2 gates (update and output)

```
if __name__ == '__main__':
    main()
```

This statement is used to check if the current .py file is imported or being run directly.

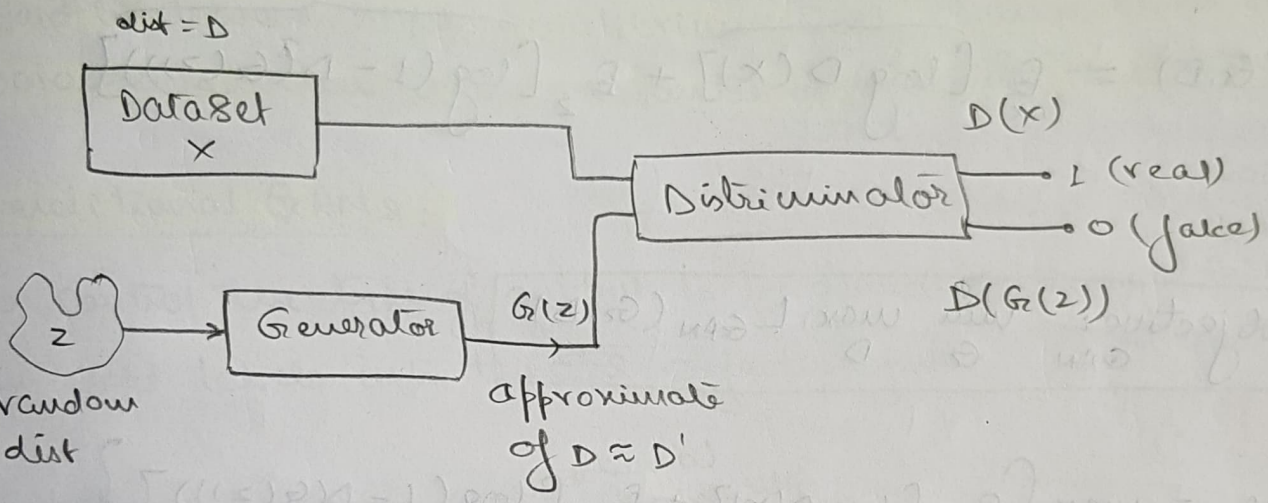
Eg:- If the file is func.py and we import it as
import func.
Now, running Print(__name__) will output 'func'.
However, if we are running func.py directly, the output will be '__main__'.

Generative Adversarial Networks:

Objective:

Given, some dataset having distribution D. We don't know this distribution, so we cannot sample new data from this distribution.

However, we can sample data from a random distribution and using this try to train a generator to approximate D.



Loss Function:

$$BCE(\hat{y}, y) = [y \log \hat{y} + (1-y) \log (1-\hat{y})]$$

Now,

$$BCE(D(x), 1) = \log D(x) \quad \text{---(1)}$$

$$BCE(D(G(z)), 0) = \log (1 - D(G(z))) \quad \text{---(2)}$$

Now, the objective of Discriminator is to classify correctly

$\Rightarrow D(x) = 1$ and $D(G(z)) = 0$

Since $D(x) \in [0, 1]$, when $D(x) = 1$, $BCE(D(x), 1) \Rightarrow$ maximum.

Similarly,

when $D(G(z)) = 0 \Rightarrow BCE(D(G(z)), 0) \Rightarrow$ maximum.

Therefore, the goal of the discriminator is to maximize ① and ②

Now, let's consider the generator: It's objective is to fool the discriminator.

* we don't want the discriminator to misclassify the fake examples.

$\Rightarrow$ ~~$D(x) = 0$~~ and $D(G(z)) = 1$

~~when $D(x) = 0$, $BCE(D(x), 1) \Rightarrow$ minimum~~

when $D(G(z)) = 1$, $BCE(D(G(z)), 0) \Rightarrow$ minimum.

Therefore, the goal of the generator is to minimize ~~① and~~ ②.

Now,

$$L_{GAN}(G, D) = \underbrace{E_x[\log D(x)]}_{BCE(D(x), 1)} + \underbrace{E_z[\log(1 - D(G(z)))]}_{BCE(D(G(z)), 0)}$$

↳ loss function for the GAN

Objective: $\min_{G} \max_{D} L_{GAN}(G, D)$

$$\text{Objective}_D = \max_D \{E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]\}$$

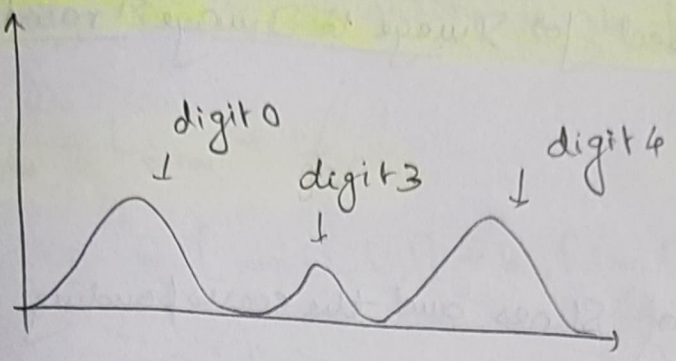
$$\text{Objective}_G = \min_G \{E_z[\log(1 - D(G(z)))]\}$$

Limitations of GANs:

* **Vanishing Gradients:** When starting to train the GAN, the discriminator is trained first. It becomes reasonably good and easily identifies the fake images as fake $\Rightarrow D(G(z)) \to 0$. The derivative of the loss func becomes small.

* **Mode Collapse:** The GAN may start to produce identical images after some epochs. Most of the real world data has multimodal dist.

  Eg:- MNIST: digits belonging to each class may be drawn from a normal dist and each digit group may have a different dist.



* The generator is supposed to learn this multimodal dist, but sometimes it learns only a single mode (mode collapse).

* This happens because the only requirement for the generator is to fool the discriminator. It is much easy for the generator to learn one mode and produce really good samples from that mode.

* Hard to achieve Nash Equilibrium: Both change their own parameters. There is no guarantee of reaching the optima.

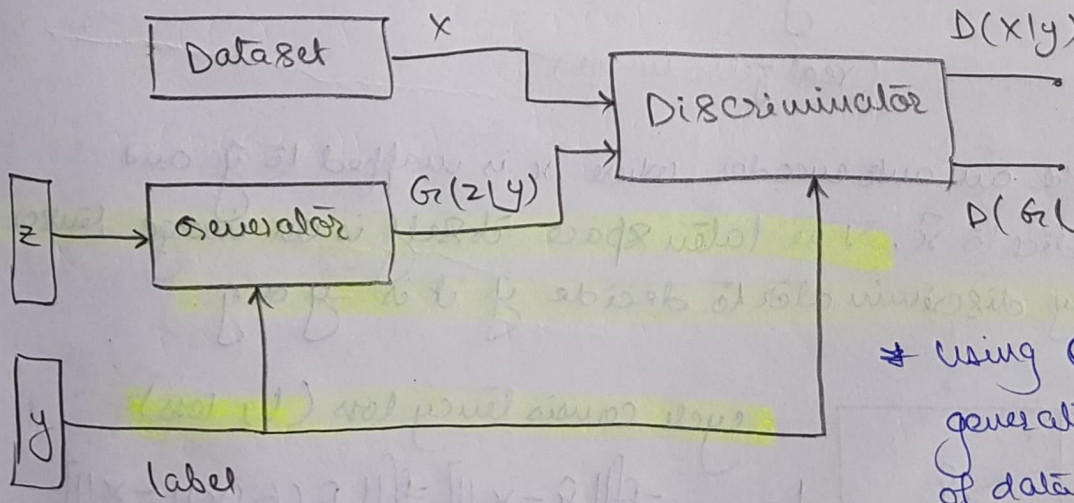
Conditional GANs:

To control the type of output to be generated through GANs, we can add labels into the objective func.

$$L_{CGAN} = E_x[\log D(x|y)] + E_z[\log(1 - D(G(z|y)|y)]$$

(D(x|y): conditional probability)

$$\text{Objective}_{CGAN} = \min_G \max_D L_{CGAN}(G, D)$$



* Using CGAN, we can generate the exact type of data we want to generate.

Pix2Pix vs Cycle GAN (used for Image to Image Translation):

used for paired data

for unpaired data

Paired Data: Silhouette of Shoes and the corresponding Shoes.

unpaired Data: Collection of Scenery and collection of paintings.

Cycle GAN:

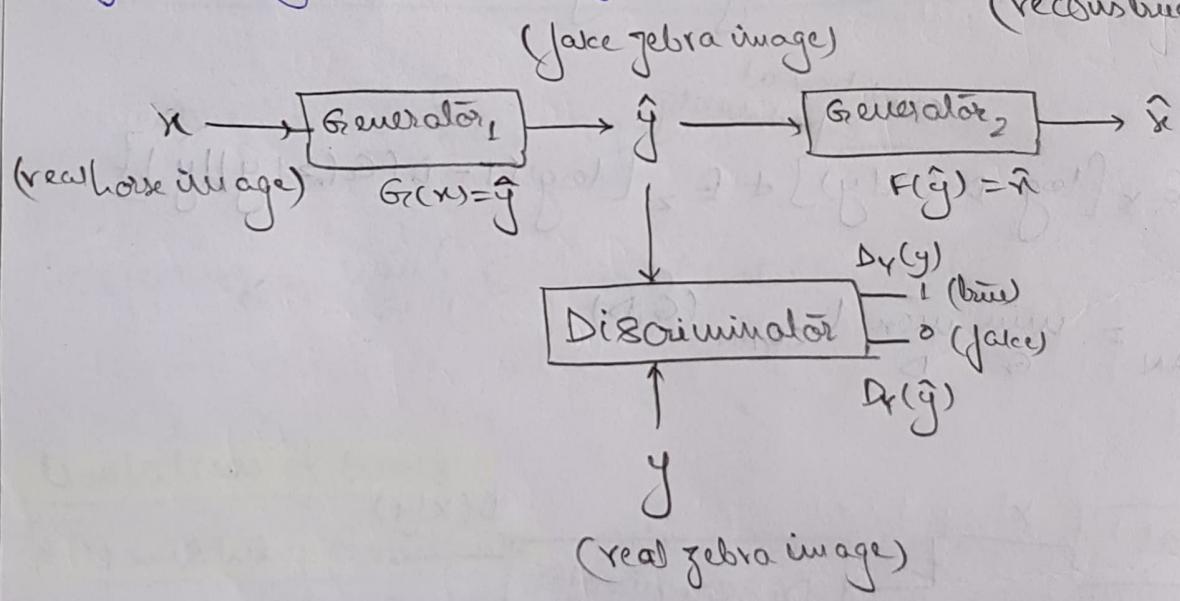
Let's convert horse images into zebra images.

X: Collection of Horse images.

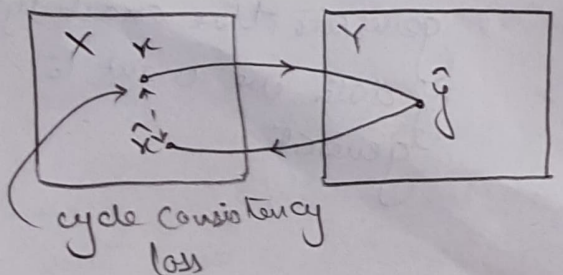
x: a single horse image taken from X.

Y: Collection of zebra images

y: a single zebra image taken from Y.



* The setup is like an autoencoder where x is mapped to $\hat{y}$ and then reconstructed to $\hat{x}$. The latent space itself is an image which is used by discriminator to decide y it is $\hat{y}$ or y.



cycle consistency loss (L_1 loss)

$$L_{cyc} = \mathbb{E}_x[\|\hat{x}-x\|] = \mathbb{E}_x[\|F(G_1(x))-x\|]$$

$$L_{cyc} = \mathbb{E}_x[\|F(G_1(x))-x\|]$$

Similarly, $Y \rightarrow X$ generation is also possible.

$$L_{cycleGAN} = L_{GAN} + L_{cyc}$$

$$= E_y[\log D_Y(y)] + E_x[\log(1 - D_Y(G(x)))]$$

$$+ E_x[\| F(G(x)) - x \|]$$

DiscoGANs

* In DiscoGANs, the cycle consistency loss has L_2 Norm.