



# C6: ANN and Deep Learning



**BITS** Pilani  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)



**Session 1  
Date – 29<sup>th</sup> June 2024  
Time – 10 am to 12.15pm**

These slides are prepared by the instructor, with grateful acknowledgement of and many others who made their course materials freely available online.

# Agenda

- Background of ANN
- Introduction to Perceptron
- Perceptron Training
- Multilayer Network
- Forward Algorithm
- Backpropagation Algorithm

# Course Content

M1	<b>Artificial Neural Network:</b> Introduction and Background, Discrimination power of single neuron, Training a single perceptron (delta rule) , Multilayer Neural Networks, Activation functions and Loss functions, Backpropagation
M2	<b>Deep Learning:</b> Introduction to end to end learning, Abstractions of features using deep layers, Hyper parameter tuning, Regularization for Deep Learning, Dropout
M3	<b>Convolution Networks with Deep Learning:</b> CNN, Pooling, Variants of pooling functions, CNN with Fully connected Networks, RCNN, Faster RCNN
M4	<b>Sequence Modeling in Neural Network:</b> Architecture of RNN , Unfolding of RNN, Training RNN, LSTM and its applications
M5	<b>Autoencoders with Deep Learning:</b> Undercomplete Autoencoders, Regularized Autoencoders, Variational autoencoders, Manifold learning with Autoencoders, Applications of Autoencoders
M6	<b>Generative deep learning models:</b> Boltzmann Machine, Restricted Boltzmann Machine, Deep Belief Machines, GAN, Applications of GAN

# Evaluation components

Evaluation Component	Marks	Type
Comprehensive Examination	40%	Open
Quizzes (2)	20%	Open
Minor Project (1)	24%	Open
Assignment (1)	16%	Open

# Artificial Neural Network

- Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.
- Successfully applied to problems
  - Interpreting visual scenes
  - Speech recognition
  - Recognize handwritten character
  - Face recognition

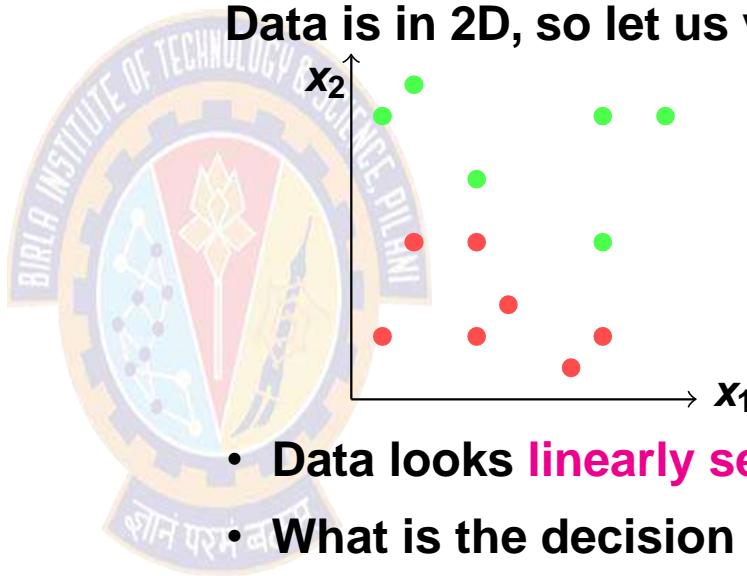
Rumelhart, D., Widrow, B., & Lehr, M. (1994). The basic ideas in neural networks. Communications of the ACM, 37(3), 87-92.

# Linear Classification

Consider Following data

$x_1$	$x_2$	y
1	9	Green
10	9	Green
4	7	Green
4	5	Red
5	3	Red
8	9	Green
4	2	Red
2	5	Red
7	1	Red
2	10	Green
8	5	Green
1	2	Red
8	2	Red

Data is in 2D, so let us visualize



- Data looks **linearly separable**
- What is the decision boundary?

Many Possibilities, such

if  $(2x_1 + 3x_2 - 25 > 0)$  it is **green**  
otherwise  
**red**

# What about this arrangement?

With chosen *decision boundary*

$$2x_1 + 3x_2 - 25 = 0$$



- This illustration is called as **perceptron**
- Provides a graphical way to represent the linear boundary
- Given a data “How to find appropriate parameters?” is an important issue

# Perceptron Training Rule

Different algorithms may converge to different acceptable hypotheses

## Algorithm 2: Perceptron training rule

- 
- 1 Begin with **random** weights  $w$
  - 2 repeat
  - 3 for each **misclassified** example do
  - 4            $w_i = w_i + \eta(t - o)x_i$
  - 5 until **all training examples are correctly classified**;
  - 6 return  $w$
- 

- Why would this strategy converge?

- Weight does not change when classification is correct
- If perceptron outputs  $-1$  when target is  $+1$ : weight increases ↑
- If perceptron outputs  $+1$  when target is  $-1$ : weight decreases ↓

Conversion with perceptron training rule is subject to linear separability of training example and appropriate  $\eta$

# Example

Consider the same data

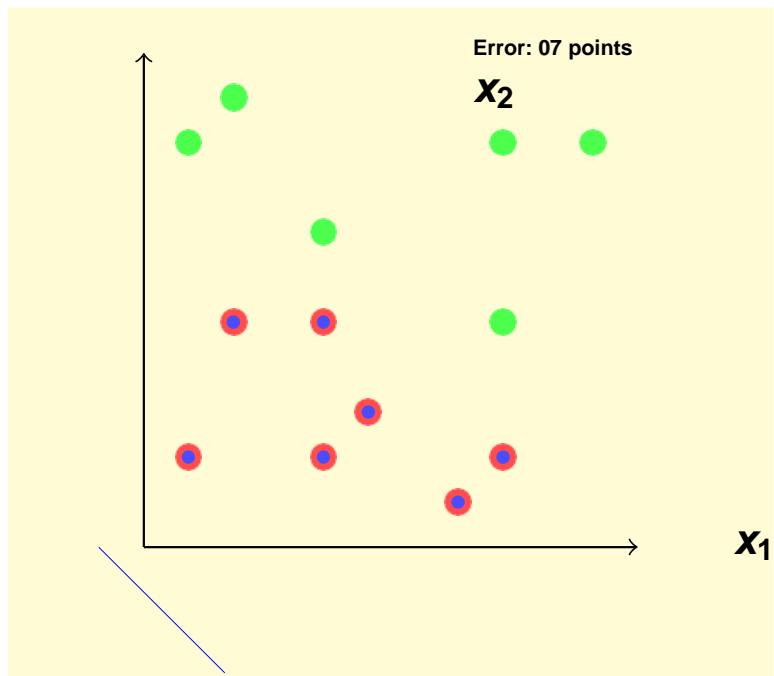
X <sub>1</sub>	X <sub>2</sub>	Y
1	9	Green
10	9	Green
4	7	Green
4	5	Red
5	3	Red
8	9	Green
4	2	Red
2	5	Red
7	1	Red
2	10	Green
8	5	Green
1	2	Red
8	2	Red



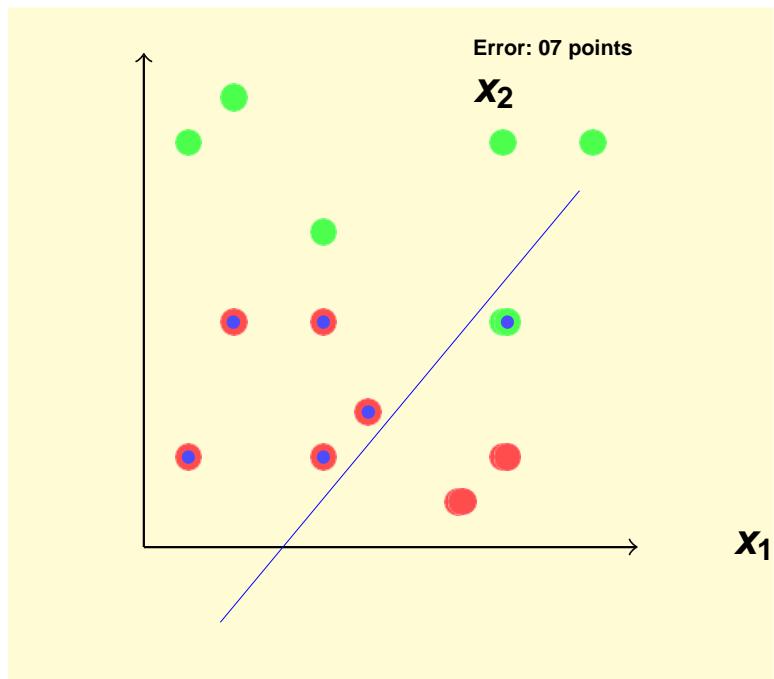
$$\eta = 0.01$$

w0=0.500, w1=0.500, w2=0.500	err=7
w0=0.360, w1=-0.120, w2=0.100	err=6
w0=0.300, w1=-0.180, w2=0.060	err=5
w0=0.240, w1=-0.140, w2=0.140	err=4
w0=0.180, w1=-0.200, w2=0.100	err=5
w0=0.120, w1=-0.160, w2=0.180	err=4
w0=0.080, w1=-0.060, w2=0.180	err=5
w0=0.020, w1=-0.120, w2=0.140	err=4
w0=-0.040, w1=-0.180, w2=0.100	err=5
w0=-0.100, w1=-0.140, w2=0.180	err=4
w0=-0.140, w1=-0.040, w2=0.180	err=5
w0=-0.200, w1=-0.100, w2=0.140	err=3
w0=-0.260, w1=-0.160, w2=0.100	err=4
w0=-0.320, w1=-0.120, w2=0.180	err=3
w0=-0.360, w1=-0.020, w2=0.180	err=3
w0=-0.420, w1=-0.080, w2=0.140	err=2
w0=-0.420, w1=-0.080, w2=0.240	err=2
<b>Fourteen more iterations</b>	
w0=-0.900, w1=-0.020, w2=0.180	err=1
w0=-0.900, w1=-0.020, w2=0.240	err=2
w0=-0.920, w1=0.020, w2=0.220	err=2
w0=-0.960, w1=-0.020, w2=0.220	err=3
w0=-0.980, w1=0.020, w2=0.200	err=2
w0=-1.000, w1=0.060, w2=0.180	err=2
w0=-1.040, w1=0.020, w2=0.180	err=0

# Visual Interpretation



# Visual Interpretation



# Visual Interpretation

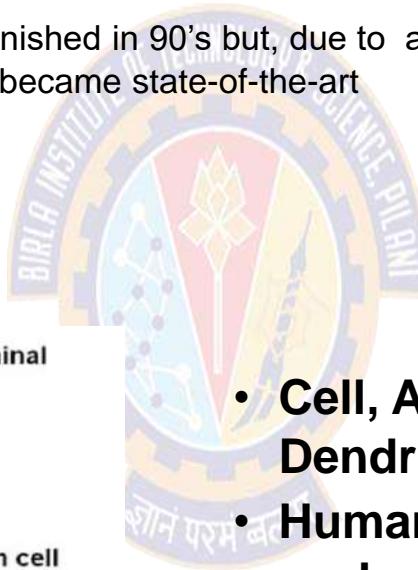
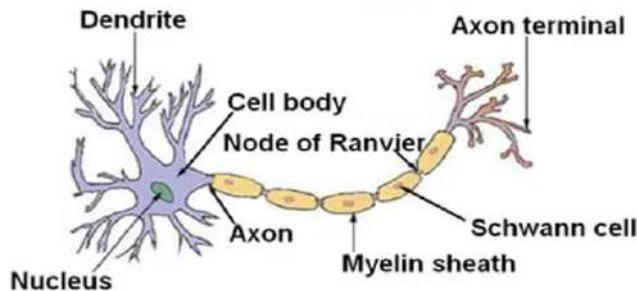


- Conversion is not gradual. (Error is NOT reducing monotonically)
- It is difficult to decide when to stop if data is not linearly separable

# Neural Network (NN)

NN is biologically motivated learning model that mimic human brain

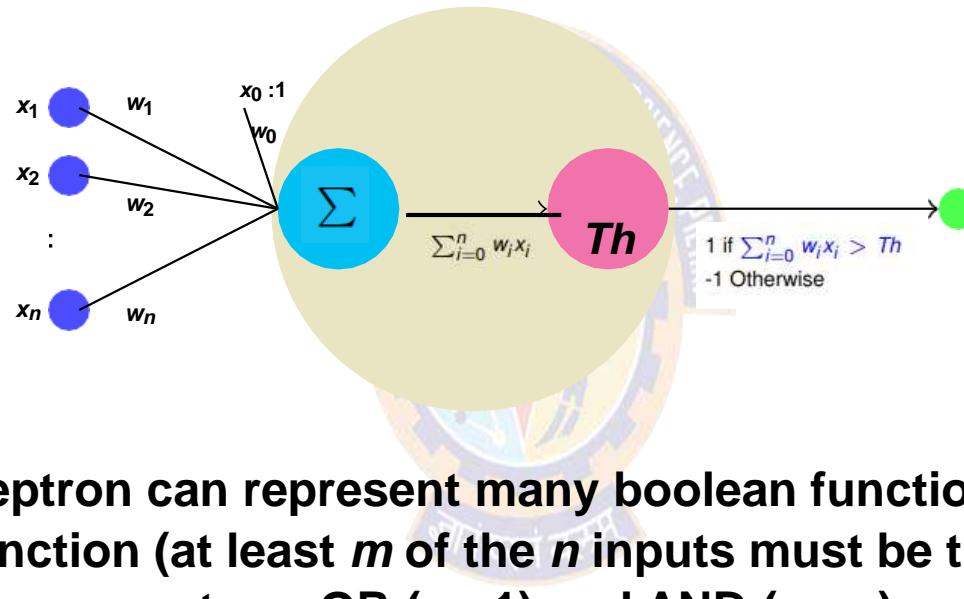
- Started by *W. McCulloch* study on working of neurons in 1943
- MADALINE (1959), an adaptive filter that eliminates echoes on phone lines was the first neural network
- Popularity of Neural Network diminished in 90's but, due to advances in **processing power** and availability of **large data** it again became state-of-the-art



- Cell, Axon, Synapses, and Dendrites**
- Humans have  $10^{11}$  neurons, each connected to  $10^4$  others, switches in  $10^{-3}$  sec**

# A Single Perceptron

## Perceptron representation

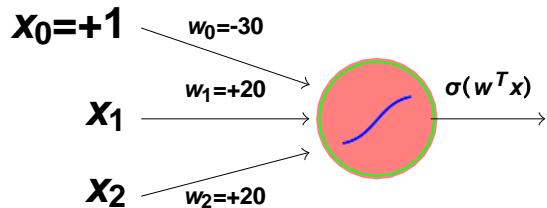


- A single perceptron can represent many boolean functions
- Any  $m$ -of- $n$  function (at least  $m$  of the  $n$  inputs must be true) can be represented by perceptron. OR ( $m=1$ ) and AND ( $m=n$ )

Two layer NN can represent any boolean function (Consider SOP)

# An Example

Consider a perceptron with output 0/1 as below



$x_1$	$x_2$	Output
0	0	0
0	1	0
1	0	0
1	1	1

This perceptron computes **logical AND**

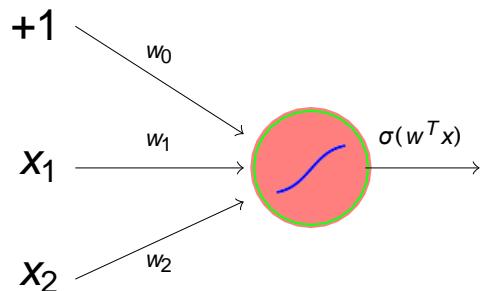
- $w_0 = -10$  gives **logical OR**
- $w_0 = 10$ ,  $w_1 = -20$  with single input gives **logical NOT**
- XOR is not possible

# An Example

Design a perceptron for

$x_1$	$x_2$	Classification
0	0	0
0	1	0
1	0	1
1	1	0

Let us assume following



We have following four equations

$$w_0 + w_1 \times (0) + w_2 \times (0) < 0 \quad (1)$$

$$w_0 + w_1 \times (0) + w_2 \times (1) < 0 \quad (2)$$

$$w_0 + w_1 \times (1) + w_2 \times (0) \geq 0 \quad (3)$$

$$w_0 + w_1 \times (1) + w_2 \times (1) < 0 \quad (4)$$

By (1)  $w_0 < 0$  so let  $w_0 = -1$

By (2)  $w_0 + w_2 < 0$  so let  $w_2 = -1$

By (3)  $w_0 + w_1 \geq 0$  so let  $w_1 = 1.5$

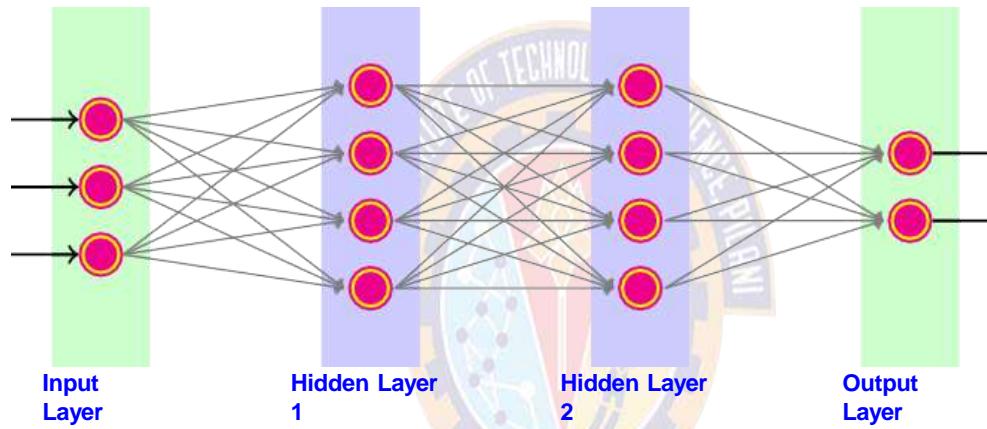
By (4)  $w_0 + w_1 + w_2 < 0$  that is valid

So  $(w_0, w_1, w_2) = (-1, -1, 1.5)$

Other possibilities are also there

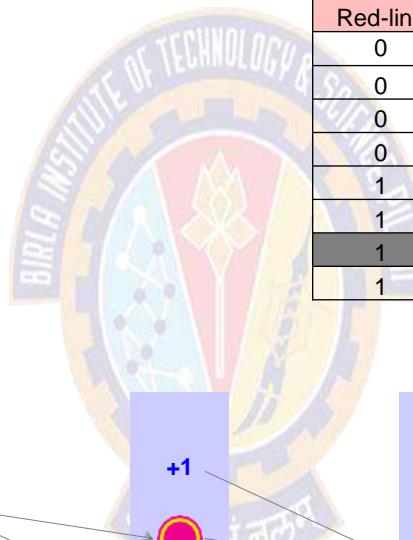
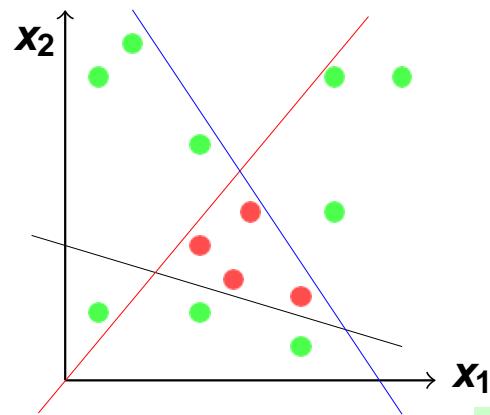
# Neural Network

When neurons are interconnected in layers

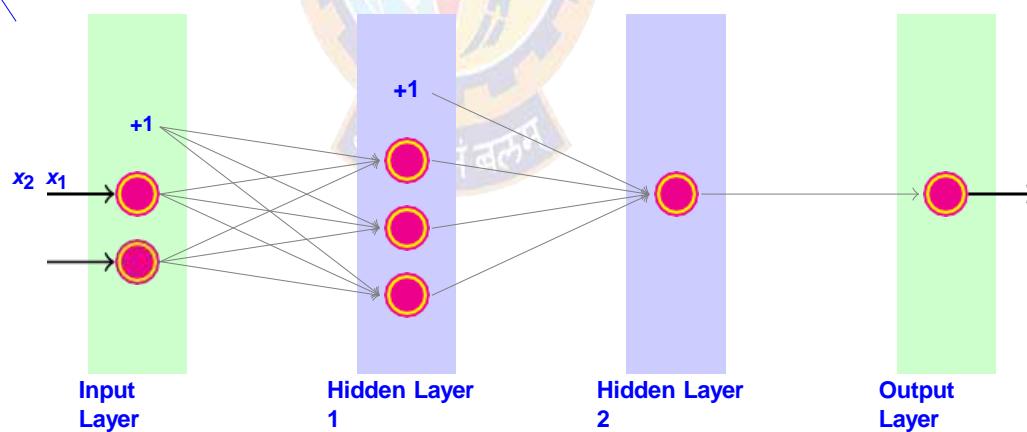


- Number of layers may differ
- Nodes in each intermediate layers may also differ
- Multiple output neurons are used for different class
- Two levels deep NN can represent any boolean function

## Whether it is green?

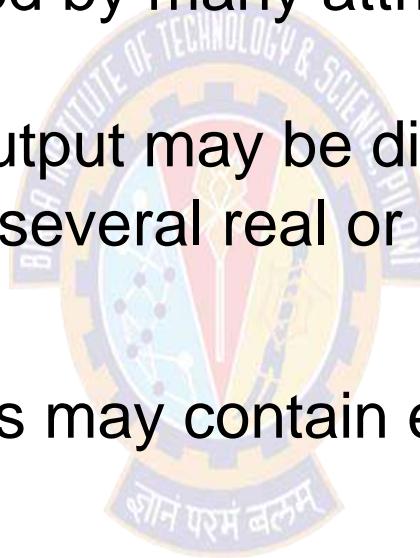


Red-line	Blue-line	Black-line	Color
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

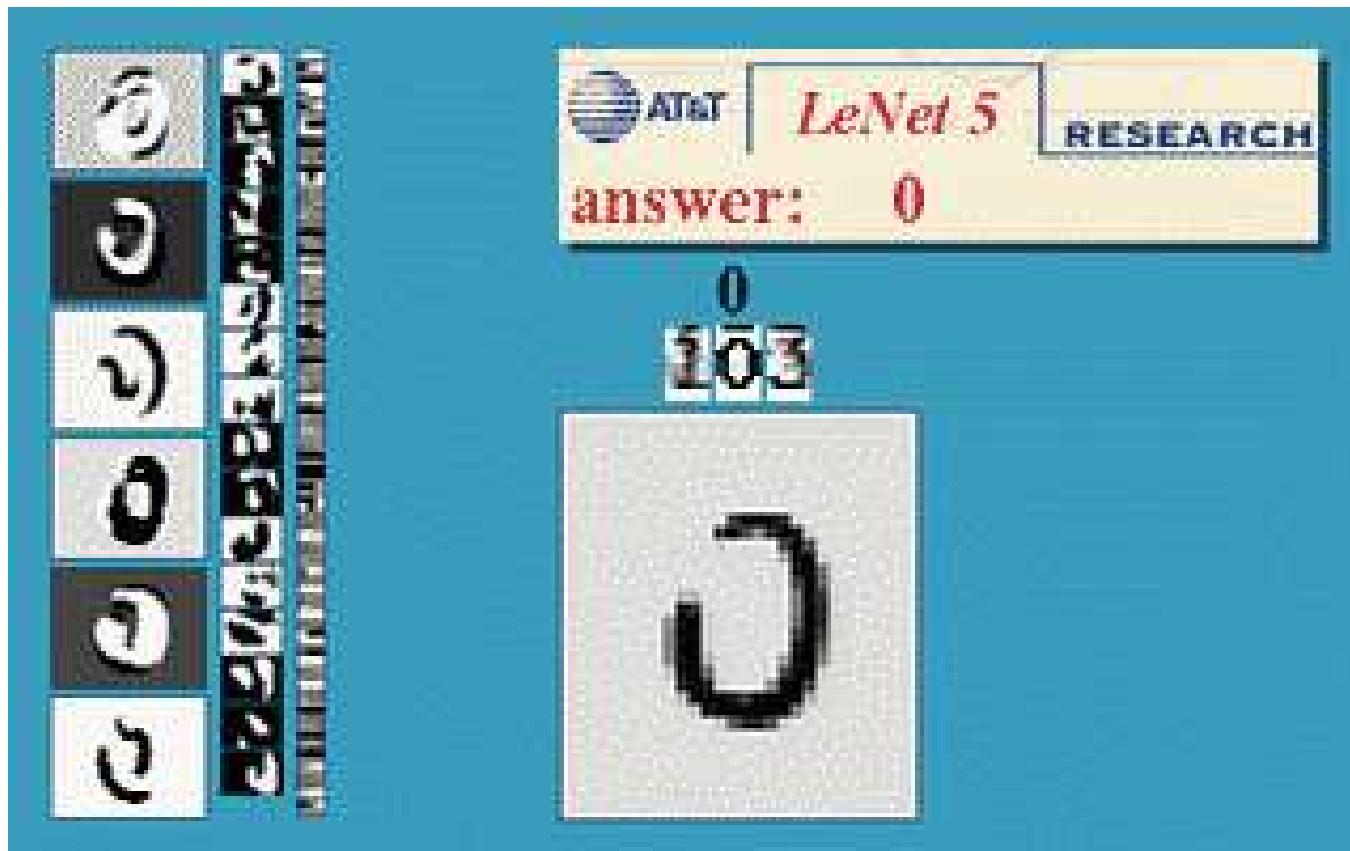


# Neural Network Applications

- NN is appropriate for problems with the following characteristics:
- Instances are provided by many attribute-value pairs (more data)
- The target function output may be discrete-valued, real-valued, or a vector of several real or discrete valued attributes
- The training examples may contain errors Long training times are acceptable
- Fast evaluation of the target function may be required
- The ability of humans to understand the learned target function is not important



# Digit Recognition



# Handwriting Recognition

**LeNet 5 Demonstration:**

<http://yann.lecun.com/exdb/lenet/>

<http://yann.lecun.com/exdb/lenet/weirdos.html>

# Perceptron Training (delta rule)

- Delta rule converges to a best-fit approximation of the target
- Uses gradient descent
- Consider unthresholded perceptron,
- Training error is defined as

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- Gradient would specify direction of steepest increase

$$\nabla E(\vec{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

- Weights can be learned as  $w_i = w_i - \eta \frac{\partial E}{\partial w_i}$

- It can be seen that  $\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$

# Perceptron Training (delta rule)

---

## Algorithm : Gradient Descent ( $D, \eta$ )

---

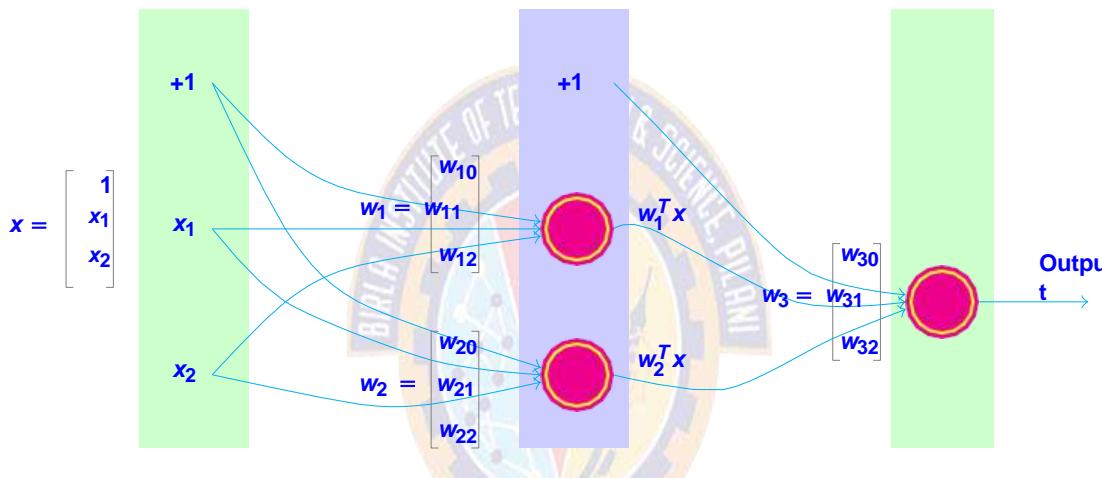
```
1 Initialize  $w_i$  with random weights
2 repeat
3   For each  $w_i$ , initialize  $\Delta w_i = 0$ 
4   for each training example  $d \in D$  do
5     Compute output  $o$  using model for  $d$  whose target
6     For each  $w_i$ , update  $\Delta w_i = \Delta w_i + \eta(t-o)x_i$ 
7   For each  $w_i$ , set  $w_i = w_i + \Delta w_i$ 
8 until termination condition is met;
9 return  $w$ 
```

---

- A date item  $d \in D$ , is supposed to be multidimensional  $d = (x_1, x_2, \dots, x_n, t)$
- Algorithm converges toward the minimum error hypothesis.

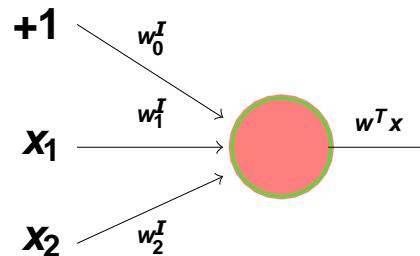
# Linear Activation is Not Much Interesting

NN with perceptrons have limited capability, even with many layers



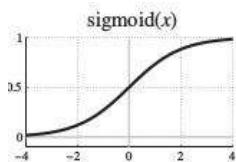
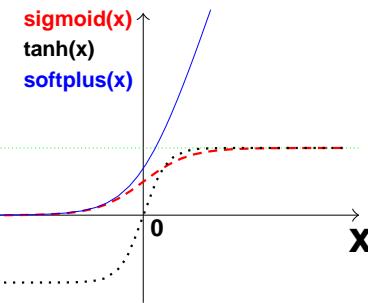
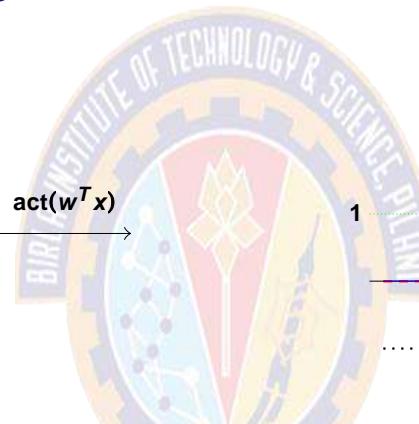
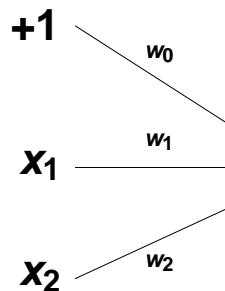
$$\begin{aligned} \text{Output} &= w_{30} \times 1 + w_{31} \times (w_1^T x) + w_{32} \times (w_2^T x) \\ &= w_{30} \times 1 + w_{31} \times [w_{10} \times 1 + w_{11} \times x_1 + w_{12} \times x_2] \\ &\quad + w_{32} \times [w_{20} \times 1 + w_{21} \times x_1 + w_{22} \times x_2] \\ &= (w_{30} + w_{31} w_{10} + w_{32} w_{20}) + (w_{31} w_{11} + w_{32} w_{21}) \times x_1 \\ &\quad + (w_{31} w_{12} + w_{32} w_{22}) \times x_2 \\ &= w'_0 + w'_1 \times x_1 + w'_2 \times x_2 \end{aligned}$$

Expression of single  
perceptron

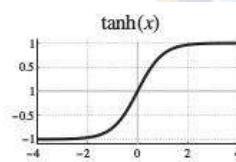


# Non-linear activation functions

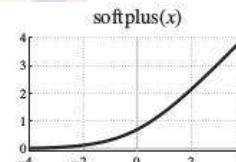
Neuron uses nonlinear **activation functions** (**sigmoid**, **tanh**, **ReLU**, **softplus** etc.) at the place of thresholding



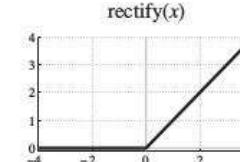
$$h(x) = \frac{1}{1 + \exp(-x)}$$



$$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$



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



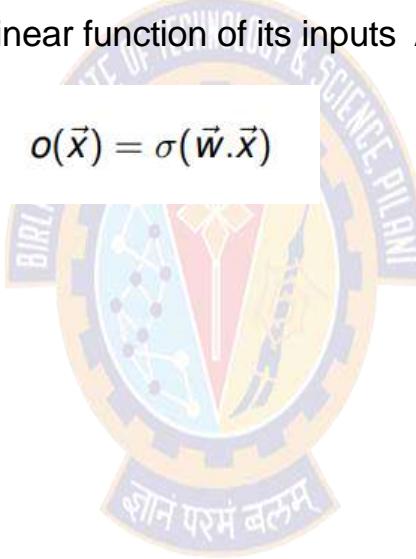
$$h(x) = \max(0, x)$$

# Multilayer Networks and Backpropagation

- Single perceptron can only express linear decision surface
- We need units whose output is a nonlinear function of its inputs AND is also differentiable (Use Neuron not Perceptron)

$$o(\vec{x}) = \sigma(\vec{w} \cdot \vec{x})$$

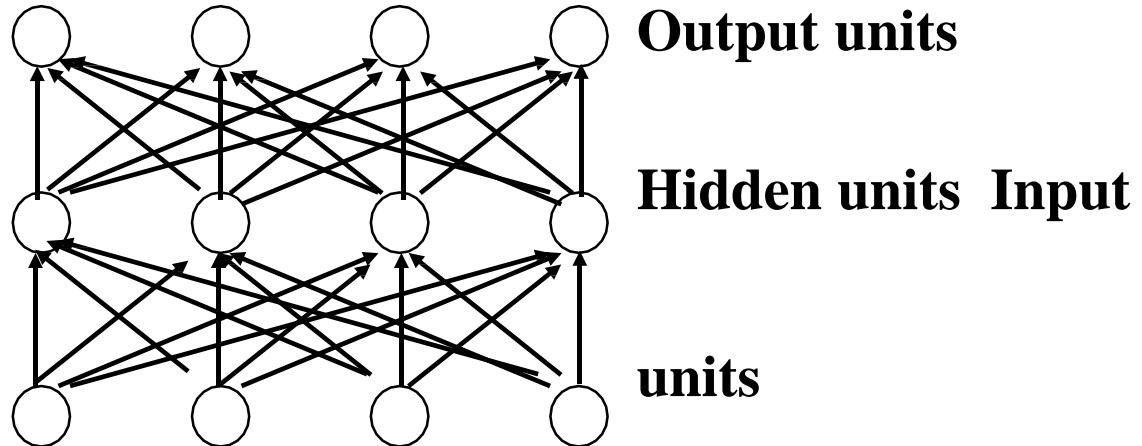
where  $\sigma(y) = \frac{1}{1+e^{-y}}$



# Multilayer network

- Single perceptrons can only express linear decision surfaces.
- In contrast, the kind of multilayer networks learned by the FORWARD and BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces

# Neural networks



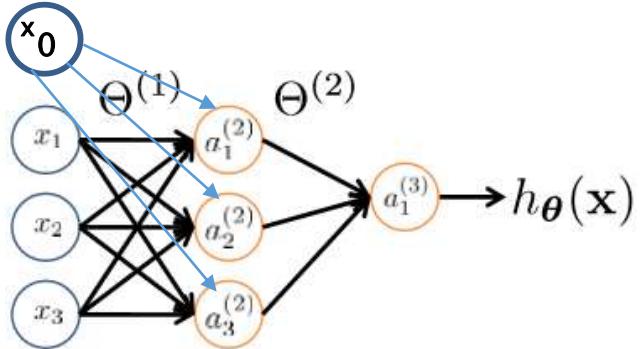
**Layered feed-forward network**

- Neural networks are made up of nodes or units, connected by links
- Each link has an associated weight and activation level
- Each node has an input function (typically summing over weighted inputs), an activation function, and an output

# Feed-Forward Process

- Input layer units are set by some exterior function (think of these as sensors), which causes their output links to be activated at the specified level
- Working forward through the network, the input function of each unit is applied to compute the input value
  - Usually this is just the weighted sum of the activation on the links feeding into this node
- The activation function transforms this input function into a final value
  - Typically this is a nonlinear function, often a sigmoid function corresponding to the “threshold” of that node

# Neural Network



$a_i^{(j)}$  = “activation” of unit  $i$  in layer  $j$   
 $\Theta^{(j)}$  = weight matrix controlling function mapping from layer  $j$  to layer  $j + 1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

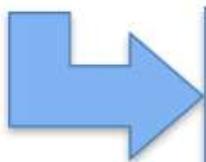
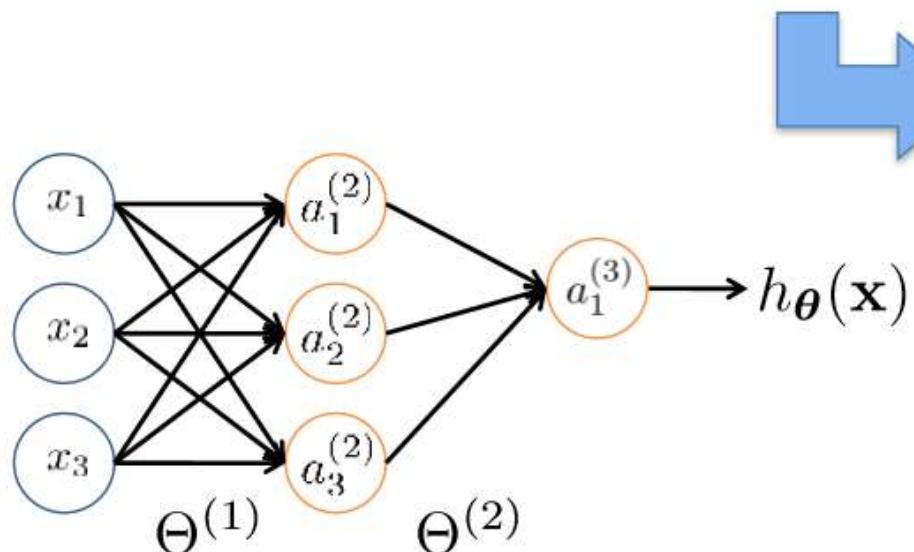
# Vectorization

$$a_1^{(2)} = g \left( \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \right) = g \left( z_1^{(2)} \right)$$

$$a_2^{(2)} = g \left( \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \right) = g \left( z_2^{(2)} \right)$$

$$a_3^{(2)} = g \left( \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \right) = g \left( z_3^{(2)} \right)$$

$$h_{\Theta}(\mathbf{x}) = g \left( \Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)} \right) = g \left( z_1^{(3)} \right)$$



Feed-Forward Steps:

$$\mathbf{z}^{(2)} = \Theta^{(1)} \mathbf{x}$$

$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

$$\text{Add } a_0^{(2)} = 1$$

$$\mathbf{z}^{(3)} = \Theta^{(2)} \mathbf{a}^{(2)}$$

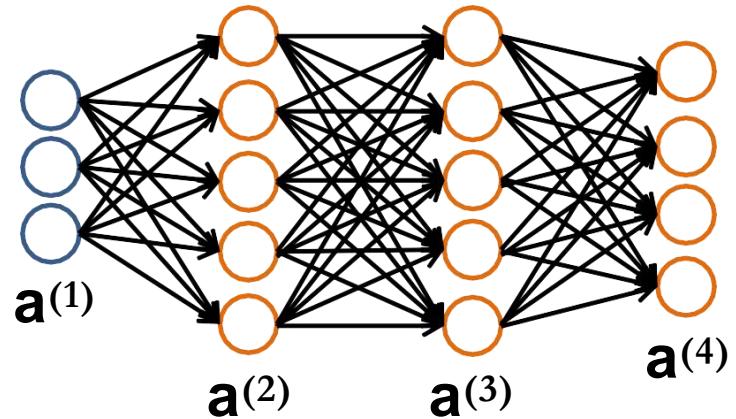
$$h_{\Theta}(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$$

# Forward Propagation

- Given one labeled training instance  $(x, y)$ :

## Forward Propagation

- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$  [add  $a_0^{(2)}$ ]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$  [add  $a_0^{(3)}$ ]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$
- $\mathbf{a}^{(4)} = h_{\Theta}(\mathbf{x}) = g(\mathbf{z}^{(4)})$



34

# Learning in NN: Backpropagation

- **Similar to the perceptron learning algorithm, we cycle through our examples**
  - If the output of the network is correct, no changes are made
  - If there is an error, weights are adjusted to reduce the error
- **The trick is to assess the blame for the error and divide it among the contributing weights**

# Backpropagation Intuition

- Each hidden node  $j$  is “responsible” for some fraction of the error  $\delta_j^{(l)}$  in each of the output nodes to which it connects
- $\delta_j^{(l)}$  is divided according to the strength of the connection between hidden node and the output node
- Then, the “blame” is propagated back to provide the error values for the hidden layer

# Chain rule

## Chain Rule of Differentiation (reminder)

- The rate of change of a function of a function is the multiple of the derivatives of those functions.
- You have probably learned this in school (nothing new here)

$$\frac{\partial}{\partial x} f(g(x)) = g'(x) \cdot f'(g)$$

$$\frac{\partial}{\partial x} f(g(h(i(j(k(x)))))) = \frac{\partial k}{\partial x} \frac{\partial j}{\partial k} \frac{\partial i}{\partial j} \frac{\partial h}{\partial i} \frac{\partial g}{\partial h} \frac{\partial f}{\partial g}$$

# Backpropogation

## Backpropagation Generalized to several layers

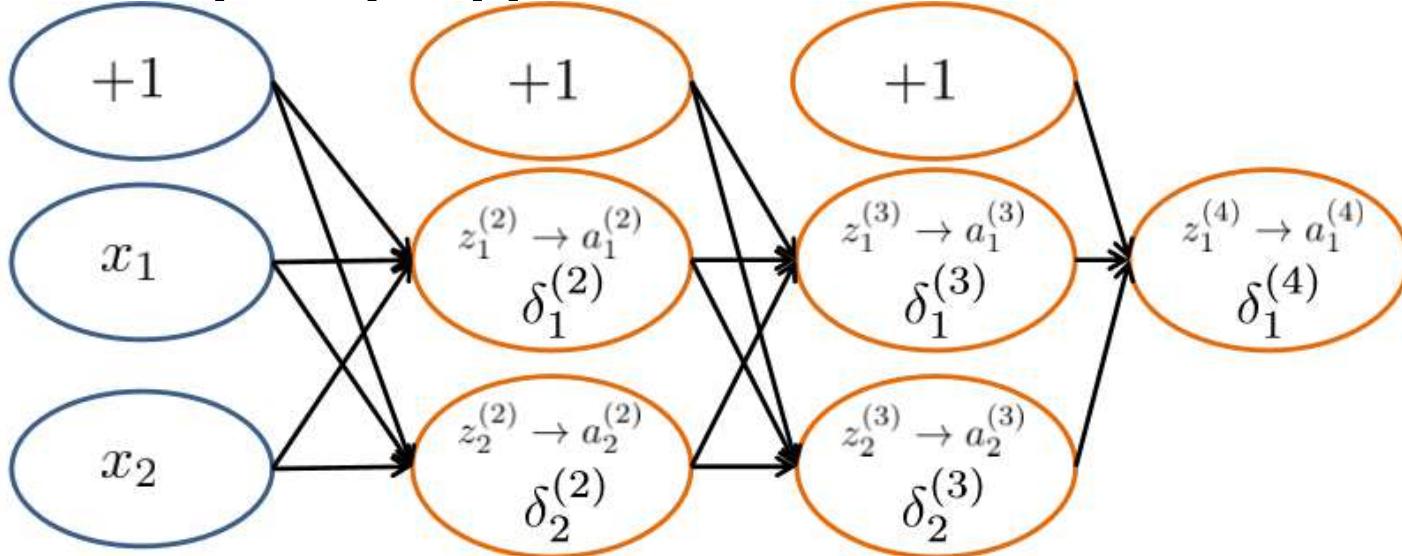
$$a_4$$
$$a_3 = w_3 \cdot a_4$$
$$a_2 = w_2 \cdot a_3$$
$$a_1 = w_1 \cdot a_2$$
$$a_0 = w_0 \cdot a_1$$
$$w_3 \quad w_2 \quad w_1 \quad w_0$$
$$\begin{aligned} & -r \frac{\partial a_1}{\partial w_1} \frac{\partial a_0}{\partial a_1} \frac{\partial C}{\partial a_0} & -r \frac{\partial a_0}{\partial w_0} \frac{\partial C}{\partial a_0} \\ & -r \frac{\partial a_2}{\partial w_2} \frac{\partial a_1}{\partial a_2} \frac{\partial a_0}{\partial a_1} \frac{\partial C}{\partial a_0} & C = (a_0 - y)^2 \\ & -r \frac{\partial a_3}{\partial w_3} \frac{\partial a_2}{\partial a_3} \frac{\partial a_1}{\partial a_2} \frac{\partial a_0}{\partial a_1} \frac{\partial C}{\partial a_0} \end{aligned}$$

For example, we adjust  $w_3$  as follows:

$$w'_3 = w_3 - r \cdot a_4 \cdot w_2 \cdot w_1 \cdot w_0 \cdot 2(a_0 - y)$$

$$-r \frac{\partial a_3}{\partial w_3} \frac{\partial a_2}{\partial a_3} \frac{\partial a_1}{\partial a_2} \frac{\partial a_0}{\partial a_1} \frac{\partial C}{\partial a_0}$$

# Backpropagation Intuition

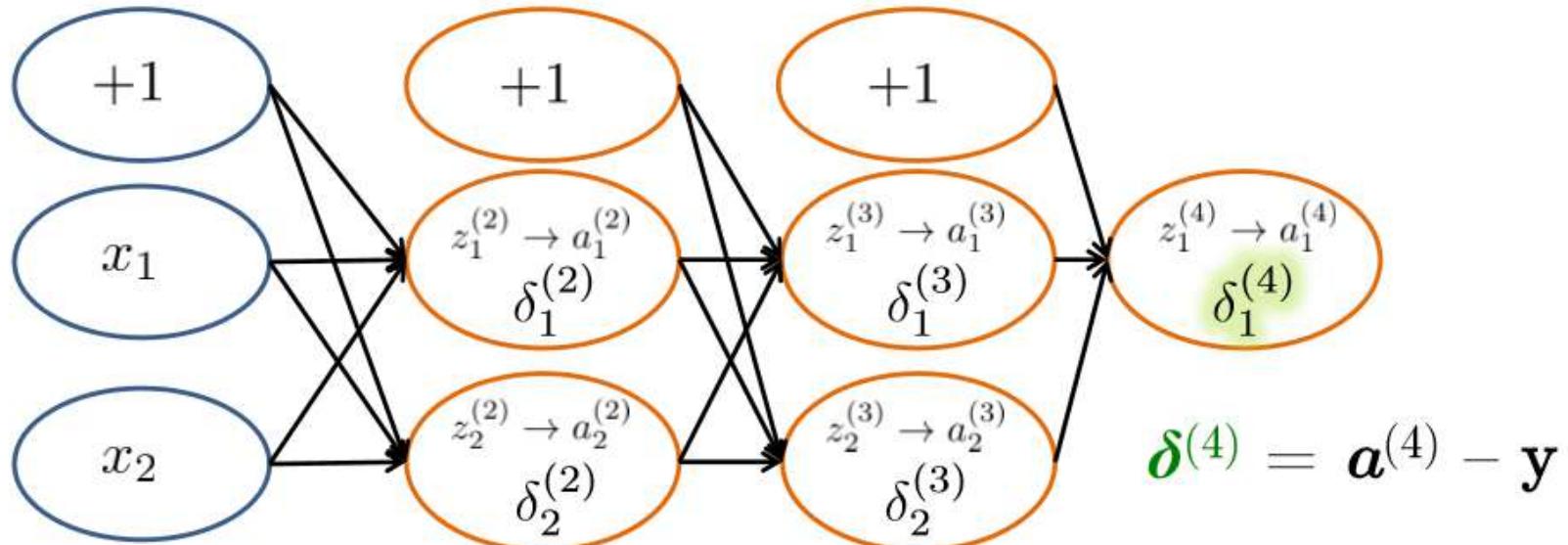


$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

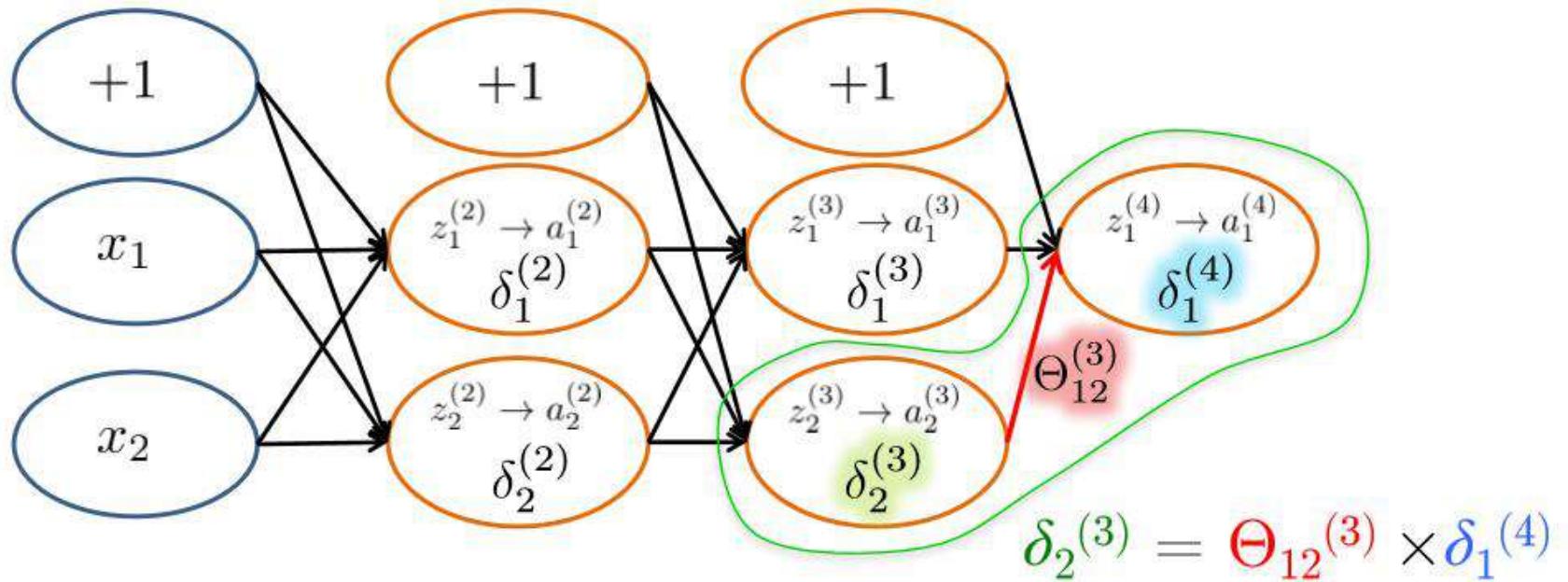
# Backpropagation Intuition



$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

# Backpropagation Intuition

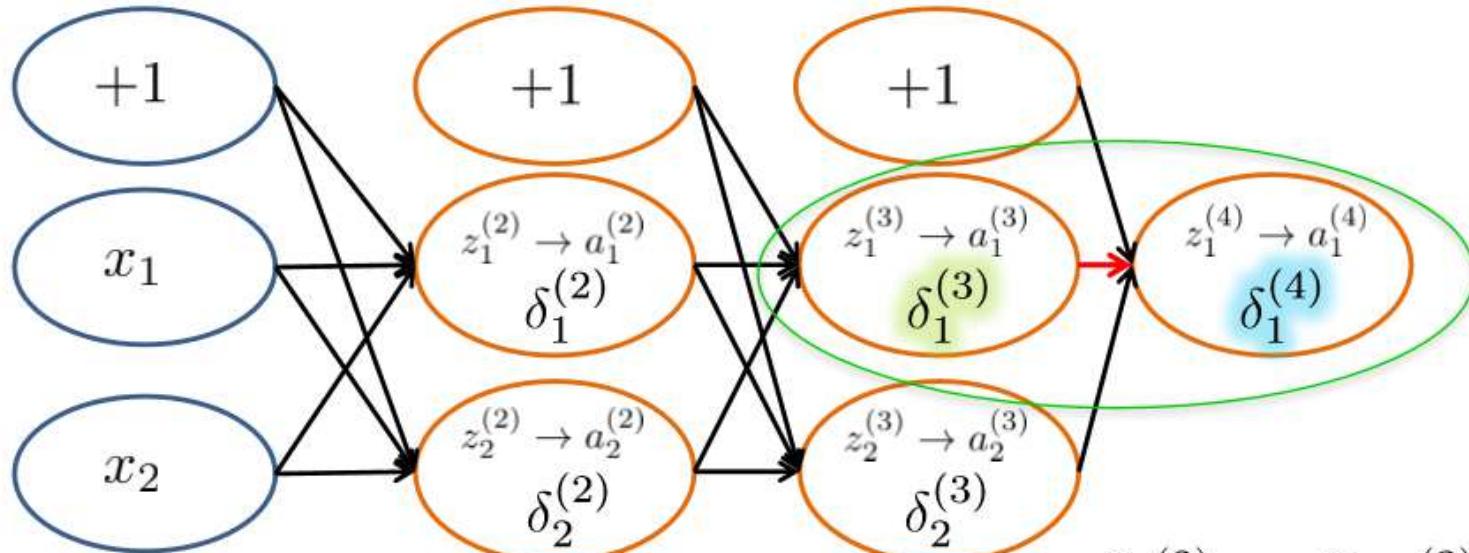


$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

# Backpropagation Intuition



$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)}$$

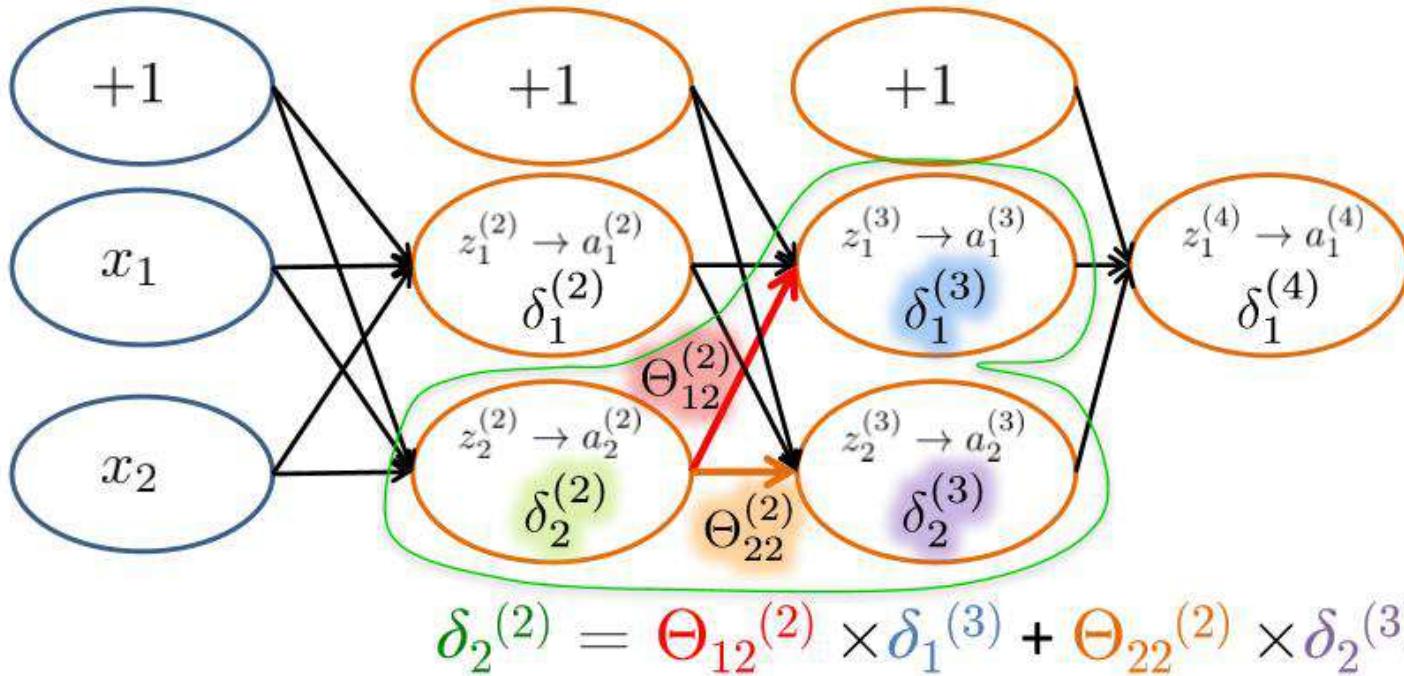
$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)}$$

$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

# Backpropagation Intuition



$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

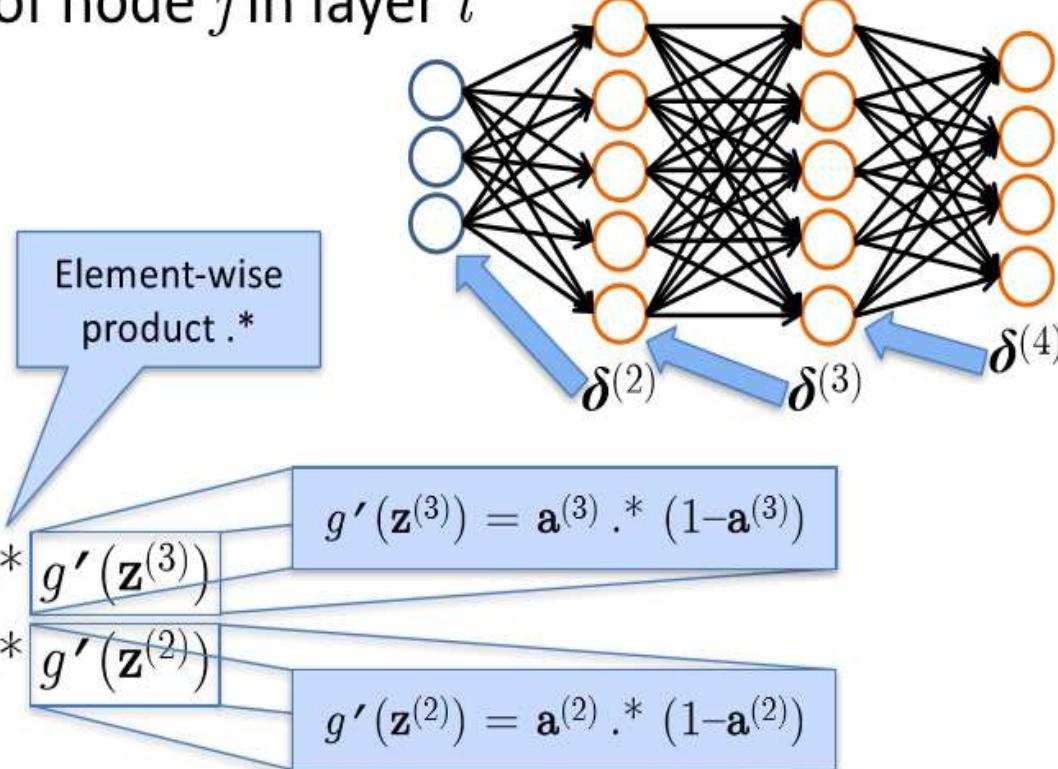
# Backpropagation: Gradient Computation

Let  $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

(#layers  $L = 4$ )

## Backpropagation

- $\delta^{(4)} = a^{(4)} - y$
- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .*$   $g'(\mathbf{z}^{(3)})$
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .*$   $g'(\mathbf{z}^{(2)})$
- (No  $\delta^{(1)}$ )



$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \quad (\text{ignoring } \lambda; \text{ if } \lambda = 0)$$

# Backpropagation

## Backpropagation

Given: training set  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$

Initialize all  $\Theta^{(l)}$  randomly (NOT to 0!)

Loop // each iteration is called an epoch

Set  $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$  (Used to accumulate gradient)

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  via forward propagation

Compute  $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors  $\{\boldsymbol{\delta}^{(L-1)}, \dots, \boldsymbol{\delta}^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

Update weights via gradient step  $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

Until weights converge or max #epochs is reached

# Backpropagation

---

## Algorithm: Backpropagation( $D, \eta, n_{in}, n_{out}, n_{hidden}$ )

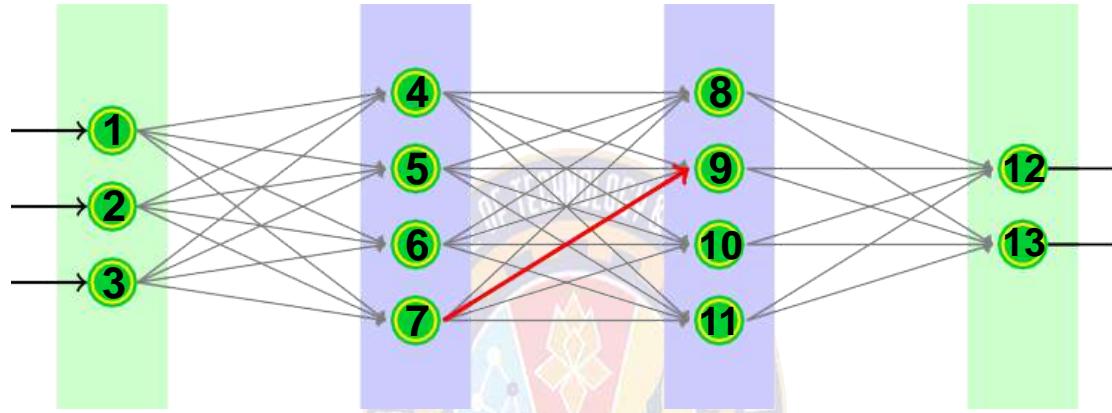
---

1 Create the feedforward network with  $n_{in}$ ,  $n_{out}$ ,  $n_{hidden}$  layers  
2 Randomly initialize weights to small values  $\epsilon[-0.05, +0.05]$   
3 **repeat**  
4     **for each**  $\langle \vec{x}, \vec{t} \rangle \in D$  **do**  
5          $o_u = \text{get output from network } \forall \text{unit } u$   
6          $\delta_k = o_k(1 - o_k)(t_k - o_k)$  for all **output unit**  $k$   
7          $\delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} (w_{kh} \delta_k)$  for all **hidden unit**  $h$   
8          $w_{ji} = w_{ji} + \Delta w_{ji}$      where  $\Delta w_{ji} = \eta \delta_j x_{ji}$   
9 **until** converge;

---

- Recall error function is  $E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$
  - For a single training example  $E_d(\vec{w}) = \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$
  - Weight  $w_{ji}$  is updated by adding  $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$
-

# Conventions Over The Network



$x_j$   $i$ th input to unit  $j$  ( $x_{94}$  is highlighted)

$w_{ji}$  weight associated with  $i$ th input to unit  $j$

$net_j$  be  $\sum_i w_{ji}x_{ji}$  the weighted sum of input for unit  $j$

$O_j$  output computed by unit  $j$ . Let it be  $\sigma(net_j)$

target output for unit  $j$

set of units in final layer ( $12, 13$ ) in our case)

$Downstream(j)$  units whose immediate input is the output of unit  $j$

We are interested in  $\frac{\partial E_d}{\partial w_{ji}}$  it is  $\frac{\partial E_d}{\partial net_j} \times \frac{\partial net_j}{\partial w_{ji}}$  and therefore,  $\frac{\partial E_d}{\partial net_j} \times x_{ji}$

## Value of $\frac{\partial E_d}{\partial \text{net}_j}$ for output units

$$\frac{\partial E_d}{\partial \text{net}_j} = \frac{\partial E_d}{\partial o_j} \times \frac{\partial o_j}{\partial \text{net}_j}$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2 = \frac{\partial}{\partial o_j} \frac{1}{2} (t_j - o_j)^2 = -(t_j - o_j)$$

Note that  $o_j = \sigma(\text{net}_j)$  therefore  $\frac{\partial o_j}{\partial \text{net}_j}$  is derivative of sigmoid

$$\begin{aligned}\frac{d}{dx} \sigma(x) &= \frac{d}{dx} \frac{1}{1 + e^{-x}} = (-1)(1 + e^{-x})^{-2} \frac{d}{dx} (1 + e^{-x}) \\ &= (-1)(1 + e^{-x})^{-2} (0 - e^{-x}) \\ &= \frac{1}{1 + e^{-x}} \times \frac{e^{-x} + 1 - 1}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x))\end{aligned}$$

As a result  $\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial \sigma(\text{net}_j)}{\partial \text{net}_j} = \sigma(\text{net}_j)(1 - \sigma(\text{net}_j)) = o_j(1 - o_j)$

$$\frac{\partial E_d}{\partial \text{net}_j} = -(t_j - o_j)o_j(1 - o_j)$$

Term  $(t_j - o_j)o_j(1 - o_j)$  is treated as  $\delta_j$

Therefore,  $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial \text{net}_j} \times x_{ji} = \boxed{\eta(t_j - o_j)o_j(1 - o_j)x_{ji}}$

## Value of $\frac{\partial E_d}{\partial net_j}$ for hidden units

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial net_k} \times \frac{\partial net_k}{\partial net_j} \\&= \sum_{k \in \text{Downstream}(j)} -\delta_k \times \frac{\partial net_k}{\partial net_j} \\&= \sum_{k \in \text{Downstream}(j)} -\delta_k \times \frac{\partial net_k}{\partial o_j} \times \frac{\partial o_j}{\partial net_j} \\&= \sum_{k \in \text{Downstream}(j)} -\delta_k \times w_{kj} \times o_j(1 - o_j)\end{aligned}$$

$\delta_j$  being  $-\frac{\partial E_d}{\partial net_j} = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k \times w_{kj}$

Therefore,  $\Delta w_{ji} = \eta \delta_j x_{ji} = \boxed{\eta(o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k \times w_{kj}) x_{ji}}$

# Backpropagation

---

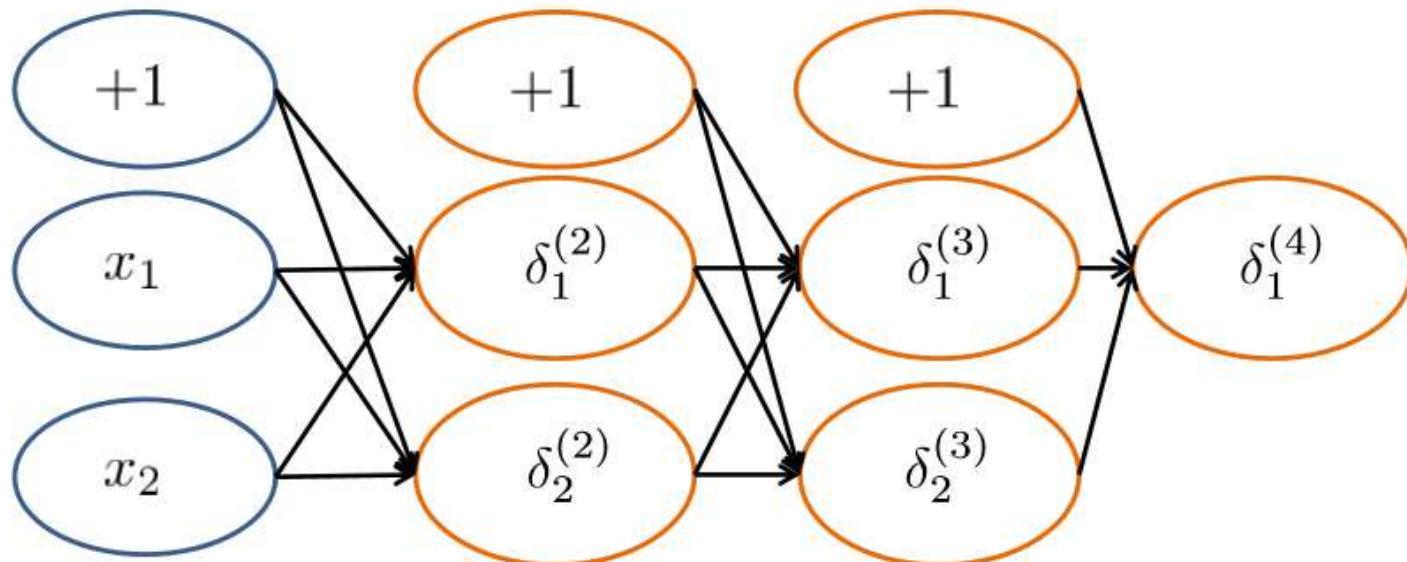
## Algorithm: Backpropagation( $D, \eta, n_{in}, n_{out}, n_{hidden}$ )

---

- 1 Create the feedforward network with  $n_{in}$ ,  $n_{out}$ ,  $n_{hidden}$  layers
  - 2 Randomly initialize weights to small values  $\in [-0.05, +0.05]$
  - 3 **repeat**
  - 4     **for each**  $\langle \vec{x}, \vec{t} \rangle \in D$  **do**
  - 5          $o_u = \text{get output from network } \forall \text{unit } u$
  - 6          $\delta_k = o_k(1 - o_k)(t_k - o_k)$  for all **output unit**  $k$
  - 7          $\delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} (w_{kh}\delta_k)$  for all **hidden unit**  $h$
  - 8          $w_{ji} = w_{ji} + \Delta w_{ji}$      where  $\Delta w_{ji} = \eta \delta_j x_{ji}$
  - 9 **until** converge;
-

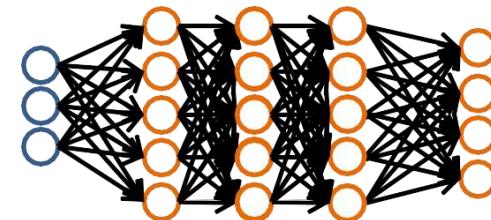
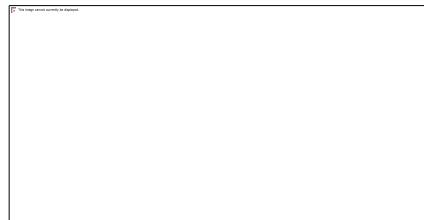
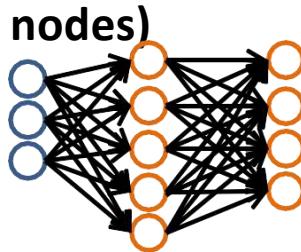
# Random Initialization

- Important to randomize initial weight matrices
- Can't have uniform initial weights, as in logistic regression
  - Otherwise, all updates will be identical & the net won't learn



# Training a Neural Network

Pick a network architecture (connectivity pattern between



- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer

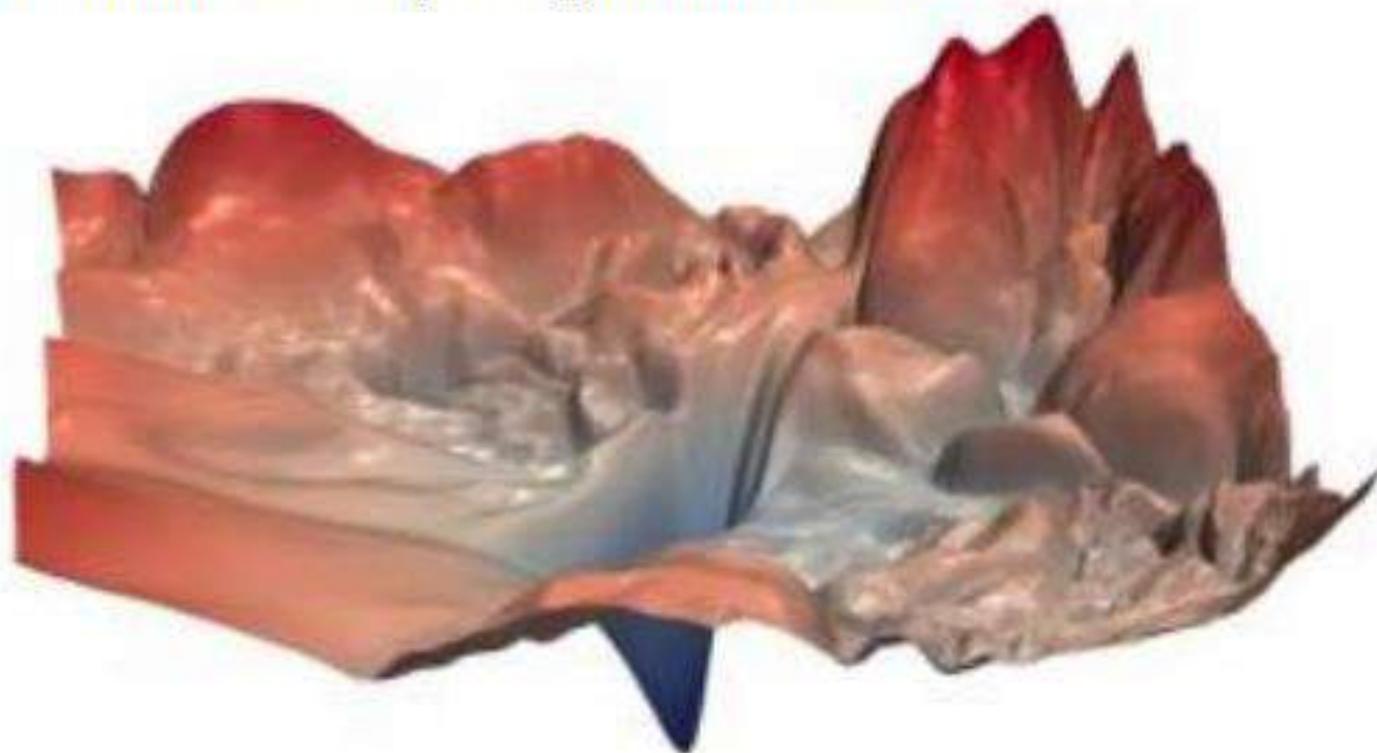
- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)

# Training a Neural Network

1. Randomly initialize weights
2. Implement forward propagation to get  $h_{\Theta}(x_i)$  for any instance  $x_i$
3. Implement code to compute cost function  $J(\Theta)$
4. Implement backprop to compute partial derivatives
$$\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$$
5. Use gradient checking to compare  $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$  computed using backpropagation vs. the numerical gradient estimate.
  - Then, disable gradient checking code
6. Use gradient descent with backprop to fit the network

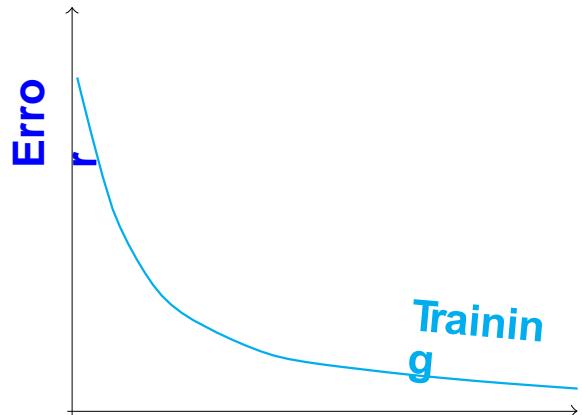
# Non-Convex

- Result of Backpropagation over multilayer networks is only guaranteed to converge toward some local minimum and not necessarily to the global minimum error



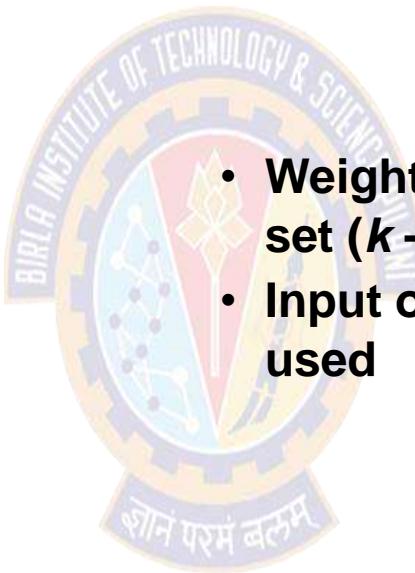
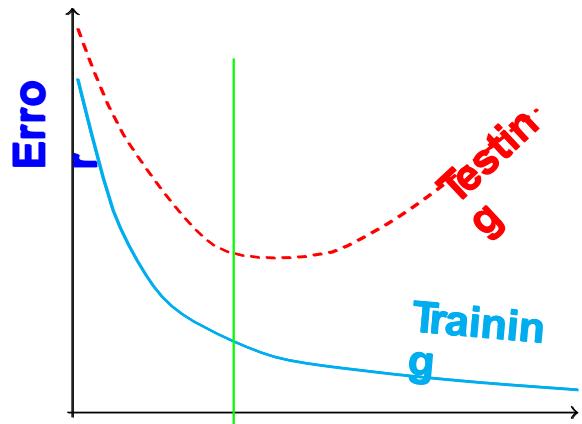
# Generalization, Overfitting, and Stopping Criterion

Continue training until the error on the training examples falls below some predetermined threshold could be a poor strategy



# Generalization, Overfitting, and Stopping Criterion

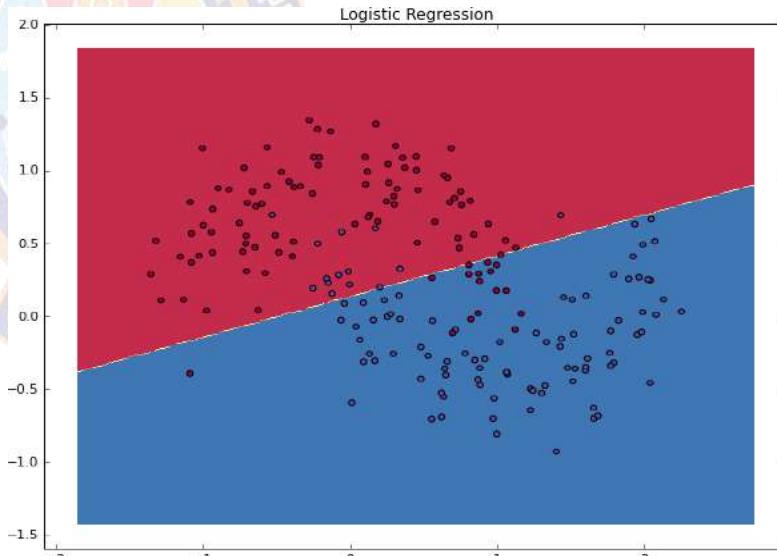
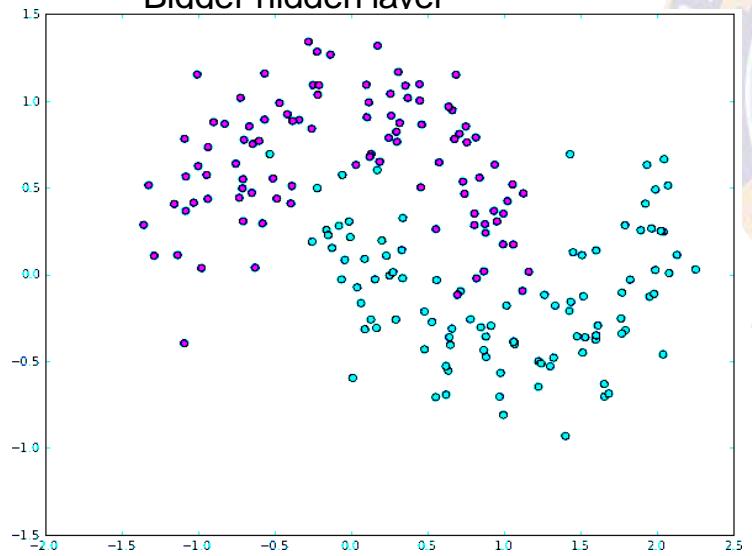
Continue training until the error on the training examples falls below some predetermined threshold could be a poor strategy



- Weight decay or use of validation set ( $k$ -fold ?) is suggested
- Input or output encoding can be used

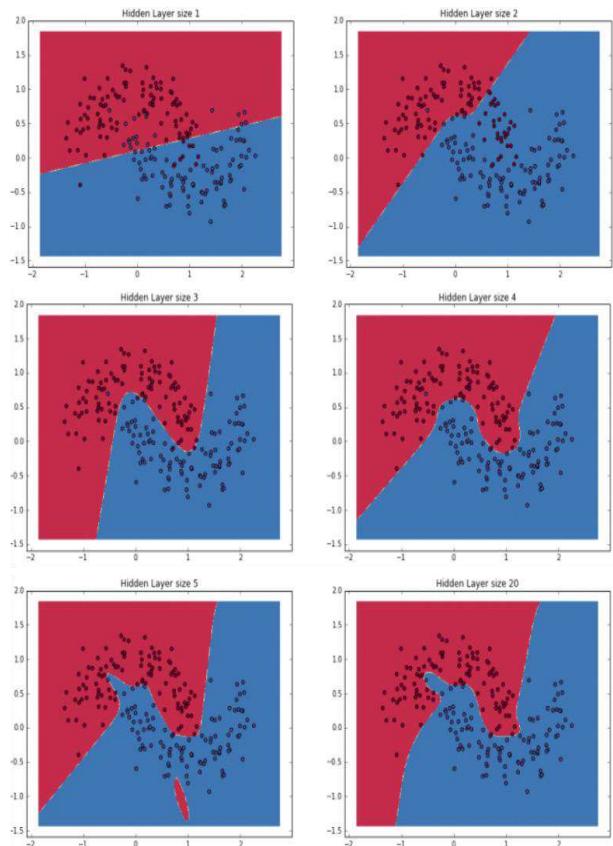
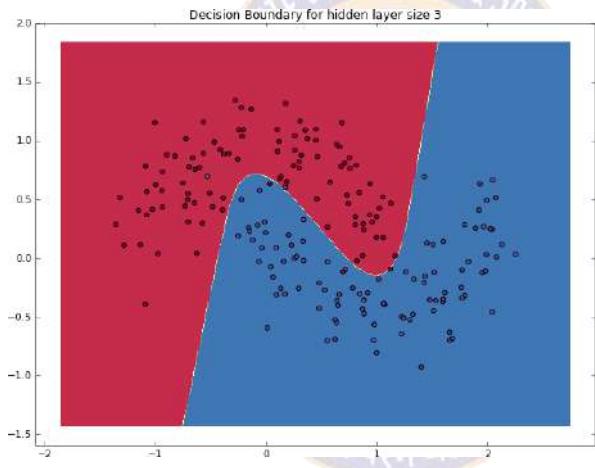
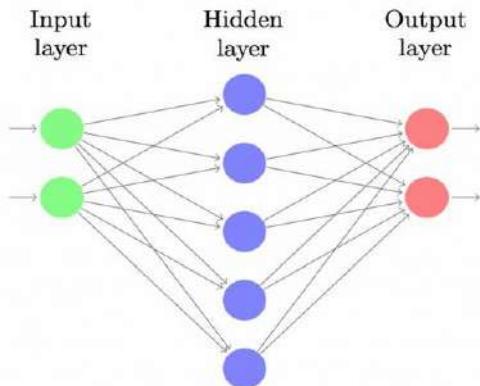
# Coding Example 1

Consider two class data  
Logistic Regression  
Neural Network  
Decision Boundary  
Bigger hidden layer



<sup>1</sup><https://github.com/dennybritz/numpy-from-scratch/blob/master/numpy-from-scratch.ipynb>

# Coding Example 1



<sup>1</sup><https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb>

# Good References for understanding Neural Network

Andrew Ng videos on neural network

[https://www.youtube.com/watch?v=EVegrPGf uCY&list=PLssT5z\\_DsK-h9vYZkQkYNWcItqhlRJLN&index=45](https://www.youtube.com/watch?v=EVegrPGf uCY&list=PLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=45)

[https://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture4.pdf](https://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf)

Autonomous driving using neural network

[https://www.youtube.com/watch?v=ppFyPUx9RIU&list=PLssT5z\\_DsK-h9vYZkQkYNWcItqhlRJLN&index=57](https://www.youtube.com/watch?v=ppFyPUx9RIU&list=PLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=57)

**Thank you**



# C6: ANN and Deep Learning



**BITS** Pilani  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)



**Session 2  
Date – 7<sup>th</sup> July 2024  
Time – 10 am to 12.15pm**

These slides are prepared by the instructor, with grateful acknowledgement of and many others who made their course materials freely available online.

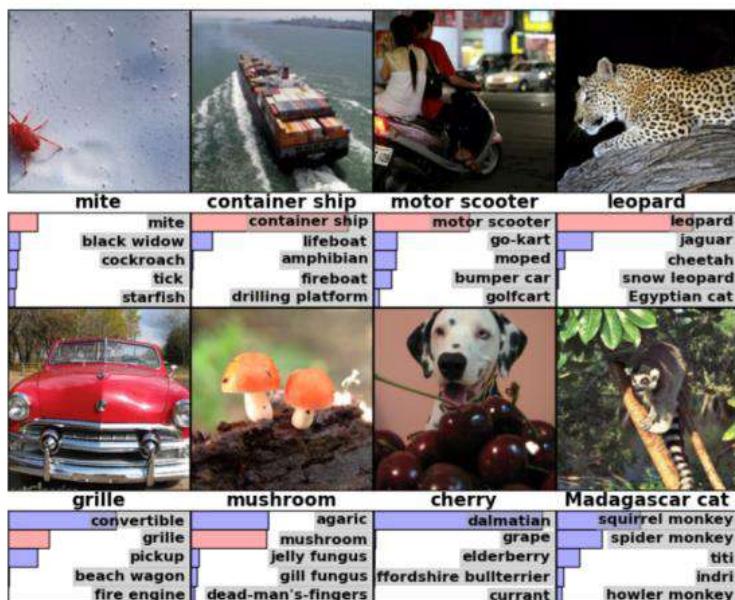
# Agenda

- Background of ANN
- Introduction to Perceptron
- Perceptron Training
- Multilayer Network
- Forward Algorithm
- Backpropagation Algorithm

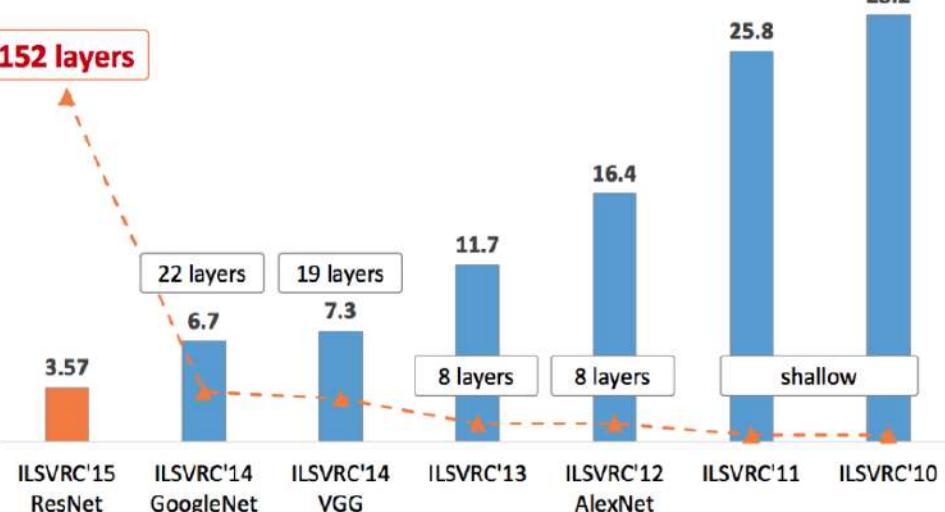
# Deep Learning (DL)

Deep neural networks are capable of solving very complex problems

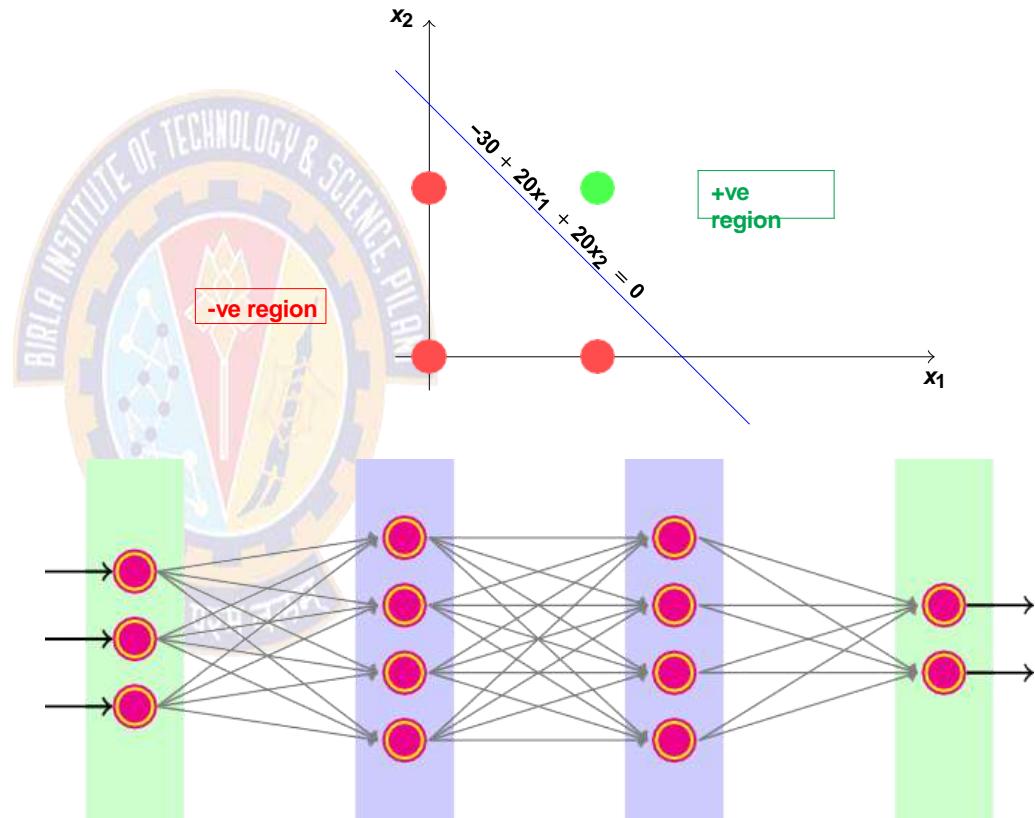
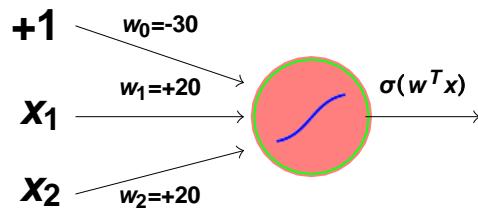
- ImageNet Challenge (2011): identify objects in images  
(4 million images in 21841 categories [Challenge 1000 categories])



**Classification: ImageNet Challenge top-5 error**

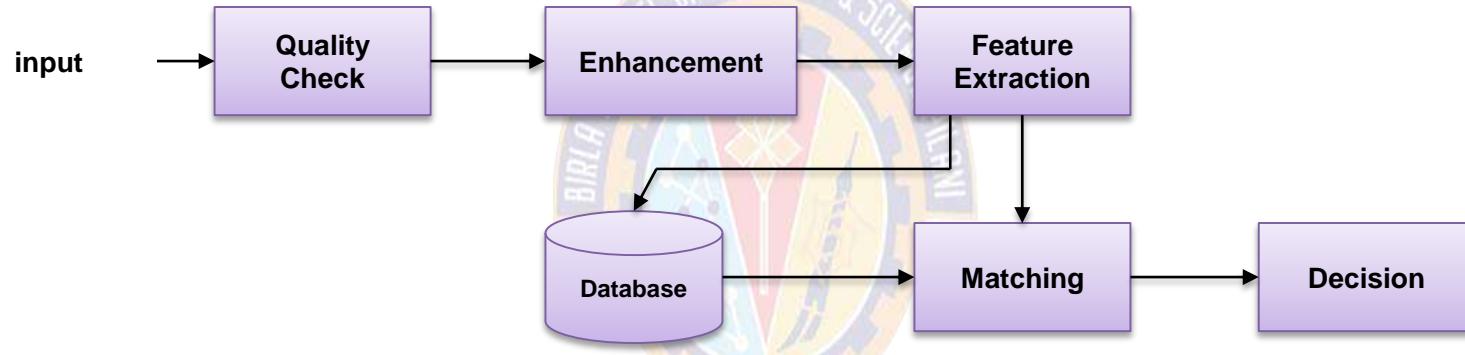


# Essentially it Represents A Decision Boundary



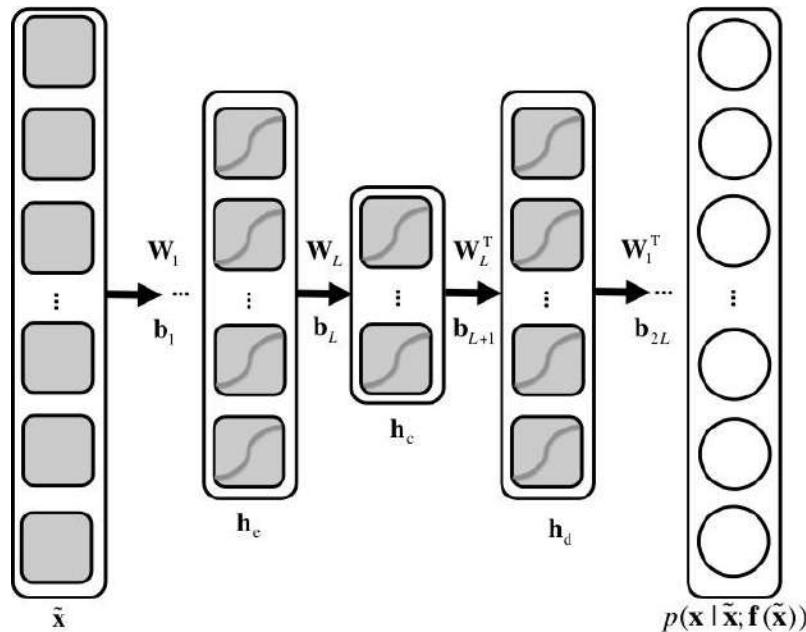
However, complex boundaries could be made using multiple neurons and stacking them in a layer.

# General ML Pipeline



Deep Learning can provided end-to-end learning

# Autoencoders



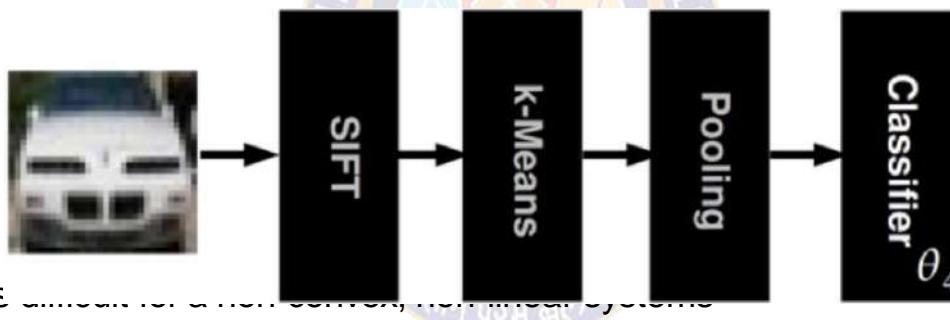
- An unsupervised learning algorithm, setting the target values to be equal to the inputs
- Learns an approximation

# Deep Learning

- **Shallow to Deep:** is linear combination to composition

$$\sum_i g_i \rightarrow g_1(g_2(g_3(\dots)))$$

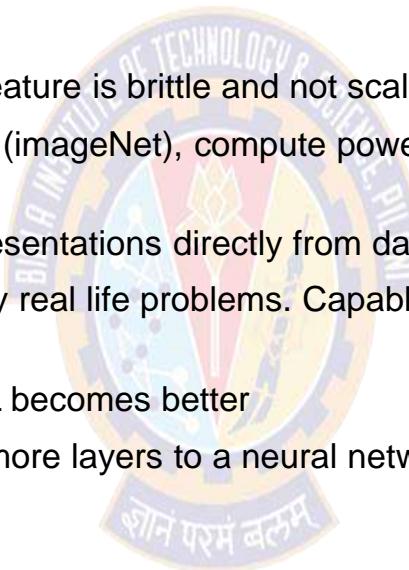
- Why?



- Optimization is ~~done in a bottom-up manner, layer by layer~~
- Freezing some layers makes it shallow.

# Key Points

- Traditional way to hand engineer feature is brittle and not scalable
- **Why now?** because we have data (imageNet), compute power (GPUs) and software (tensorFlow)
- DL learns underlying features/representations directly from data.
- DL have been found useful to many real life problems. Capable of solving very complex problems.
- As the data increases algorithm DL becomes better
- Deep learning is **NOT** just adding more layers to a neural network.



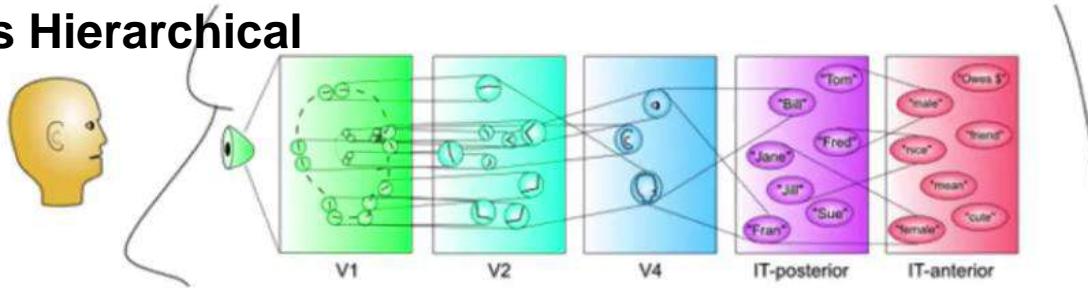
# Deep Learning

DL is constructing network of parameterized functional modules and training them from examples using gradient based optimization

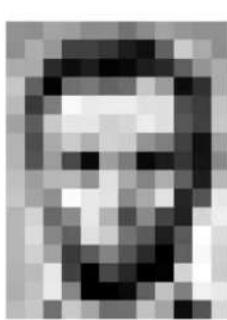
- **Yun LeCun**

# Visual Cortex is Hierarchical

# Visual Cortex is Hierarchical



**Images are numbers (so determine feature)**

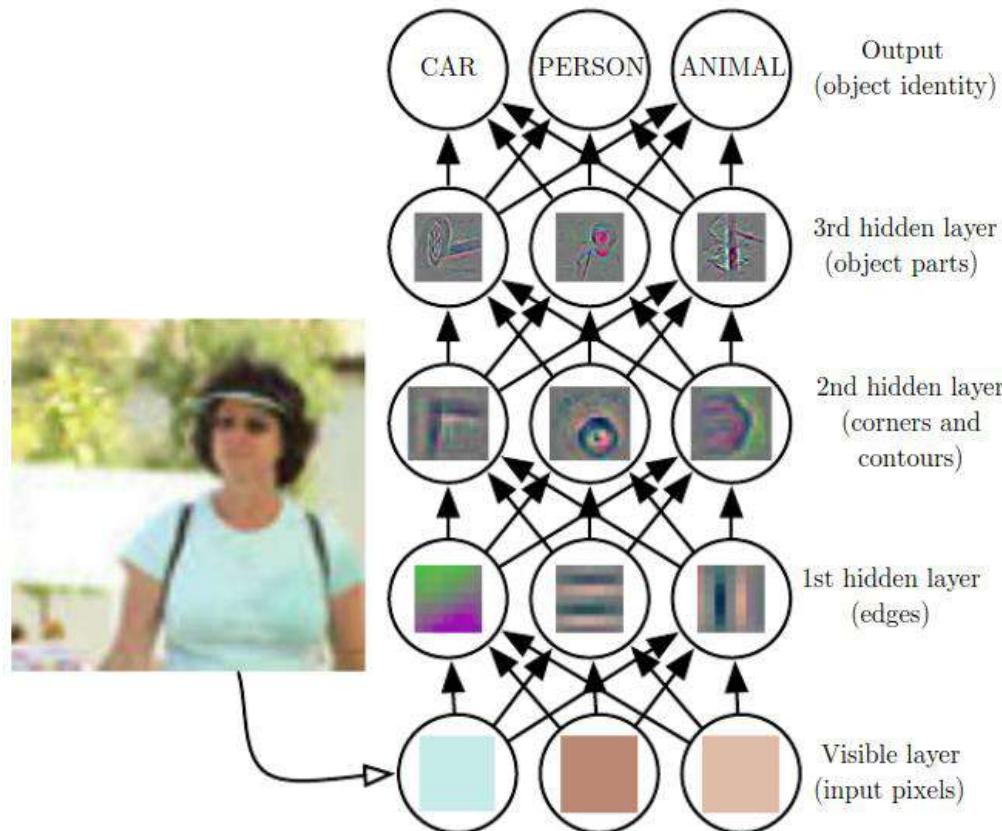


### What the computer sees

There could be occlusion, viewpoint variation, scale variation, illumination variation, deformation, background clutter, noise, intra-class variation and much more.

**DL helps to get hierarchy of features**

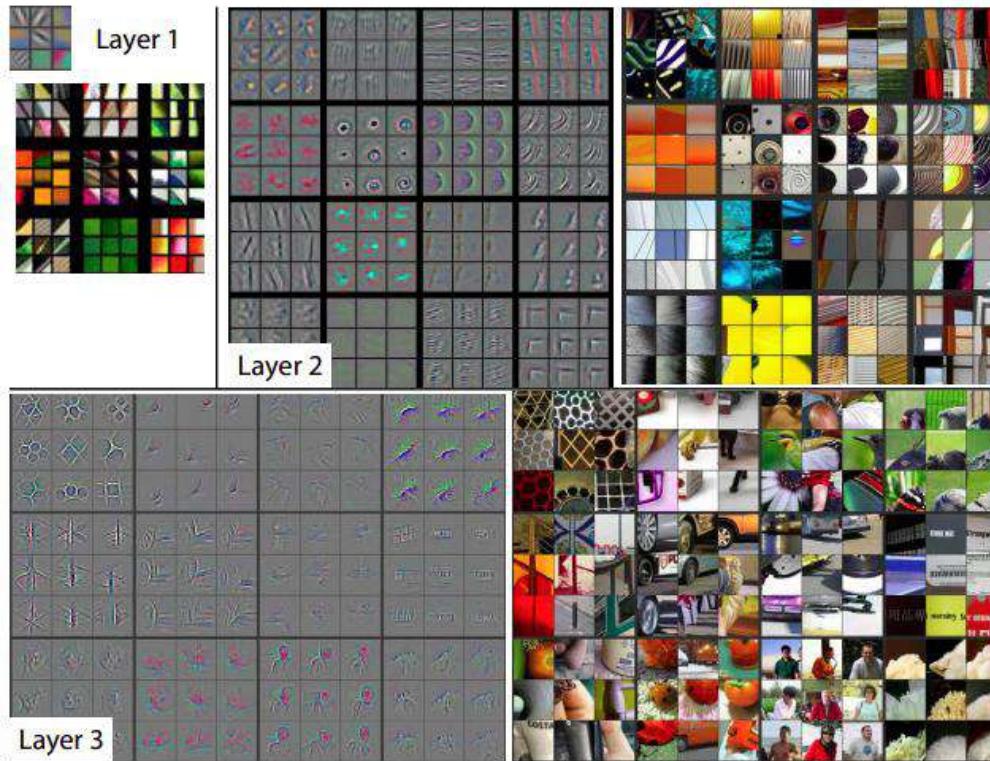
# DL Builds Hierarchy of Features



**Higher (or deeper) layers represents abstraction of the the features**

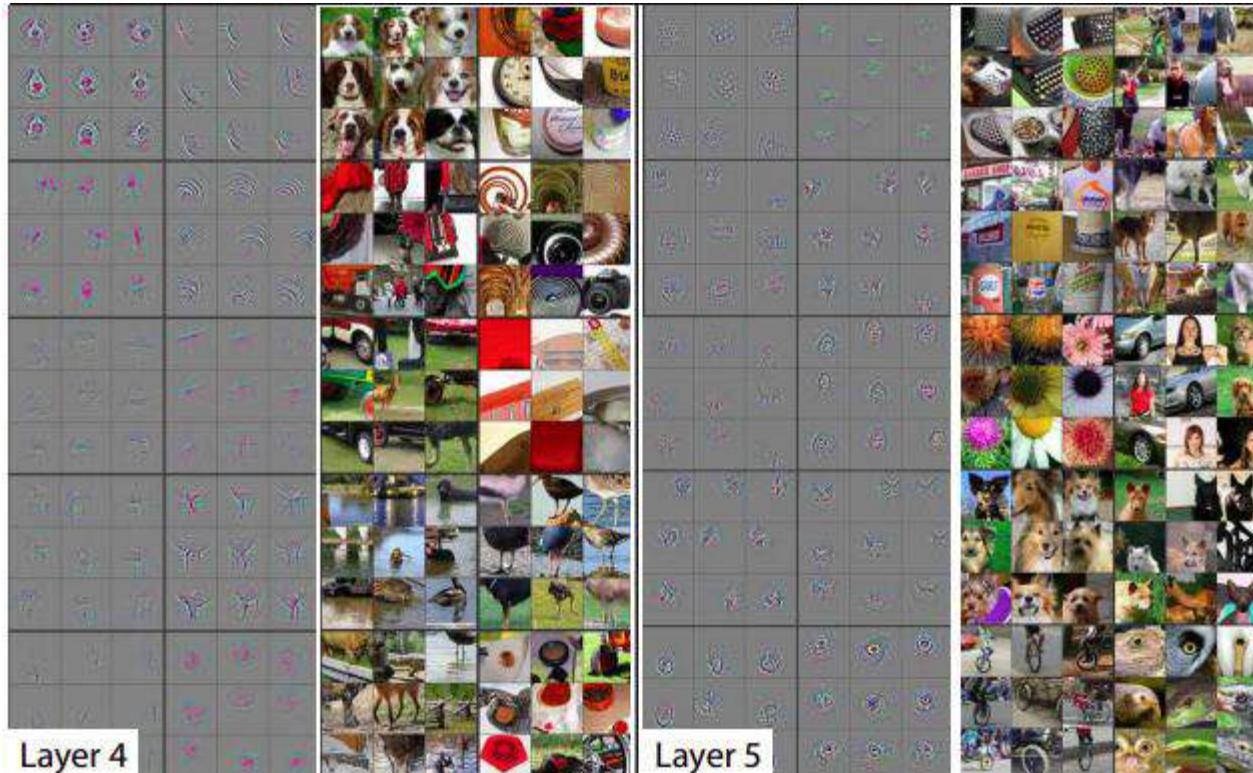
# ECCV2014 - Visualizing and Understanding CNN

A case study on 5 layer trained network



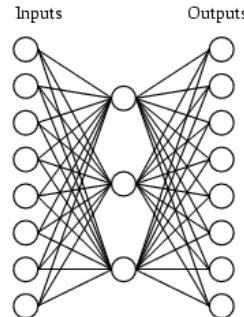
Visualizing and Understanding Convolutional Networks, ECCV2014

# ECCV2014 - Visualizing and Understanding CNN



# Learning Hidden Layer Representations

---



A target function:

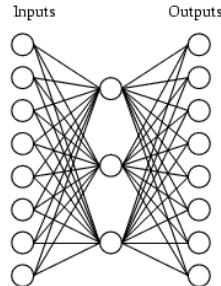
Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

Can this be learned??

# Learning Hidden Layer Representations

---

A network:

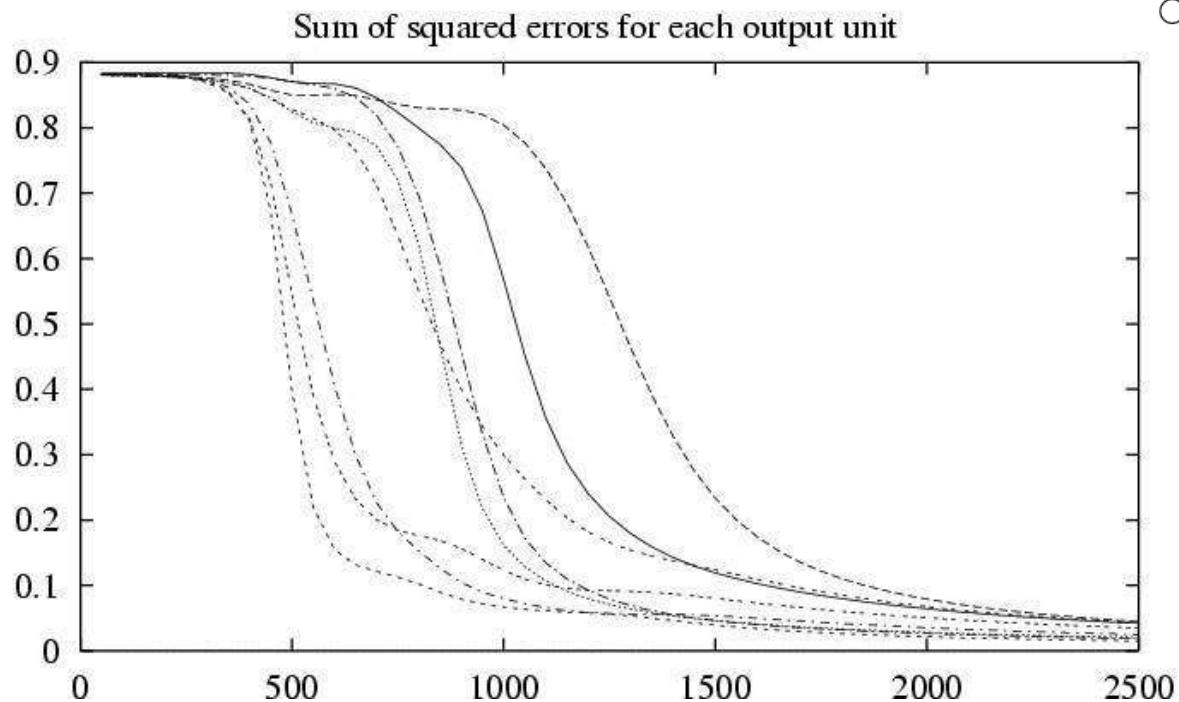
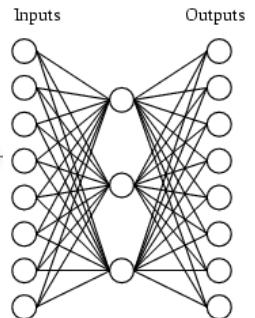


Learned hidden layer representation:

Input	Hidden Values	Output
10000000	→ .89 .04 .08	→ 10000000
01000000	→ .01 .11 .88	→ 01000000
00100000	→ .01 .97 .27	→ 00100000
00010000	→ .99 .97 .71	→ 00010000
00001000	→ .03 .05 .02	→ 00001000
00000100	→ .22 .99 .99	→ 00000100
00000010	→ .80 .01 .98	→ 00000010
00000001	→ .60 .94 .01	→ 00000001

# Training

---



# Hyperparameter Tuning

**There are many interesting questions for the network**

- How many layers?
- How many units in each layer?
- What should be the learning rate?
- What is right activation function? etc.

**Difficult to answer in the beginning**

**It is an iterative process**



# Loss Landscape



Low training error comes with a risk of overfitting (high variance)

# Weight Decay

- Optimization minimize loss  $\min_w J(w)$  by adjusting  $w$  where

$$J(w) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

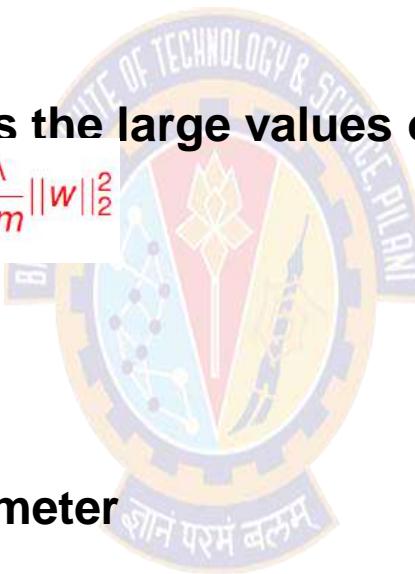
- Regularization penalizes the large values of  $w$  by**

$$J(w) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

**where**

$$\|w\|_2^2 = \sum_j w_j^2 = w^T w$$

**$\lambda$  being regularization parameter**



# Regularization for logistic regression

- As there could be L layers each with their parameters so

$$J(w^{[1]}, w^{[2]}, \dots, w^{[L]}) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|_F^2$$

where frobenius norm is

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} w_{ij}^2$$

- How you update the parameter earlier?

Get

$$dw^{[l]} = (\text{from backpropagation}) \text{ that is } \frac{\partial J}{\partial w^{[l]}}$$



then

$$w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]}$$

# Regularization for NN

- With regularization term  $d w^{[l]} = (\text{from backpropagation}) \frac{\lambda}{m} w^{[l]}$

- Therefore the update is modified to

$$w^{[l]} = w^{[l]} - \alpha.(\text{(from backpropagation)} + \frac{\lambda}{m} w^{[l]})$$

Which is

$$w^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]} - \alpha. (\text{from backpropagation})$$

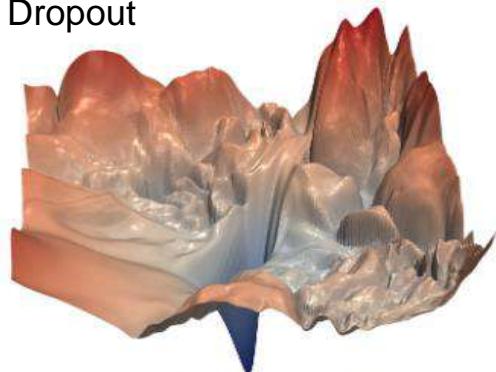
- Due to the  $\left(1 - \frac{\alpha\lambda}{m}\right)$  factor, this update method is also called **weight decay**
- Our objective here is to penalize the weight matrices being too large

# Regularization

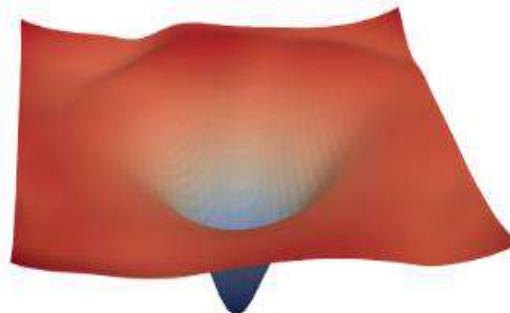
- With low weight, the connection get weakened so network has effectively less connections and become simpler.
- Another intuition is that, when weights are lower, the output of the units is also lower. Assume activation function be tanh small input values tends to produce linear output.
- So overall n/w tends to becomes linear (more biased)

# Regularization

- Dropout



(a) without skip connections



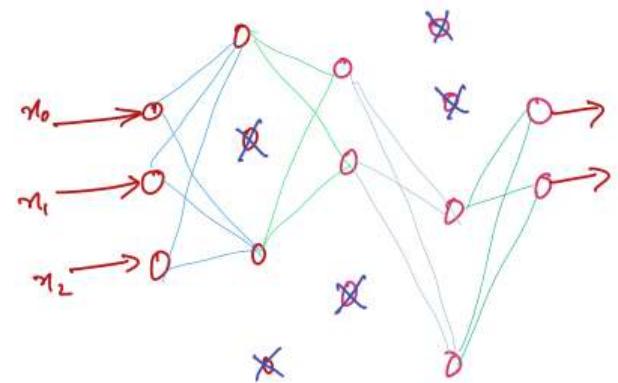
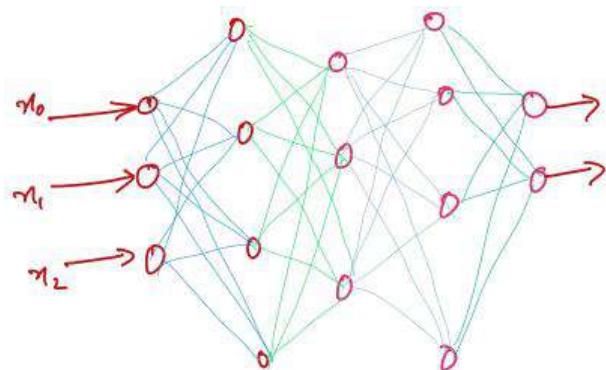
(b) with skip connections

- Early stopping



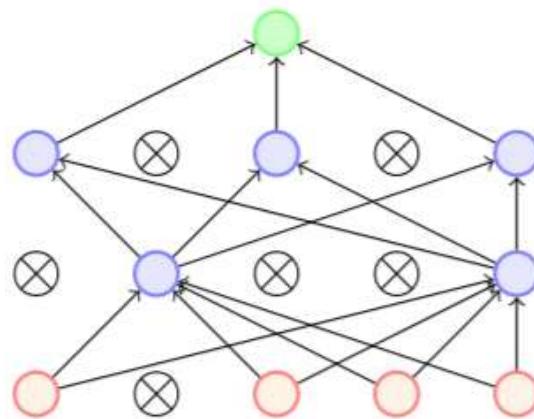
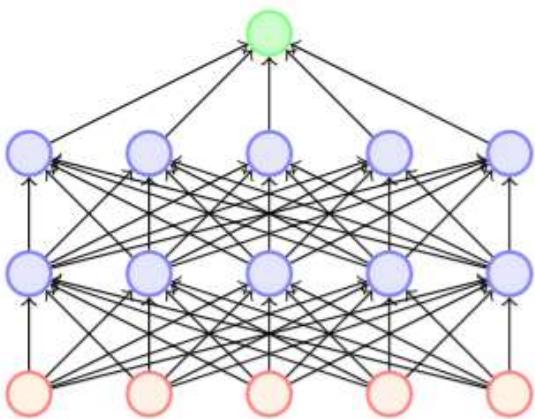
# Dropout Regularization

Randomly shutdown some of the units



- Drop probabilities for different layers may be different
- Networks can not rely on single connection. So have to give importance to others also
- Issue is that cost function is now not well defined

# Dropout



- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained with a fixed probability (typically  $p = 0.5$ ) for hidden nodes and  $p = 0.8$  for visible nodes

# Dropouts

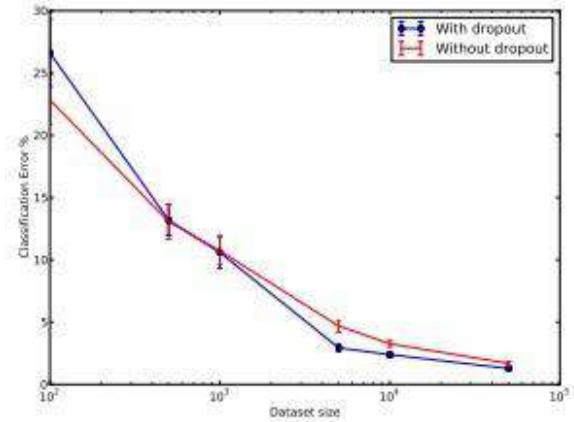
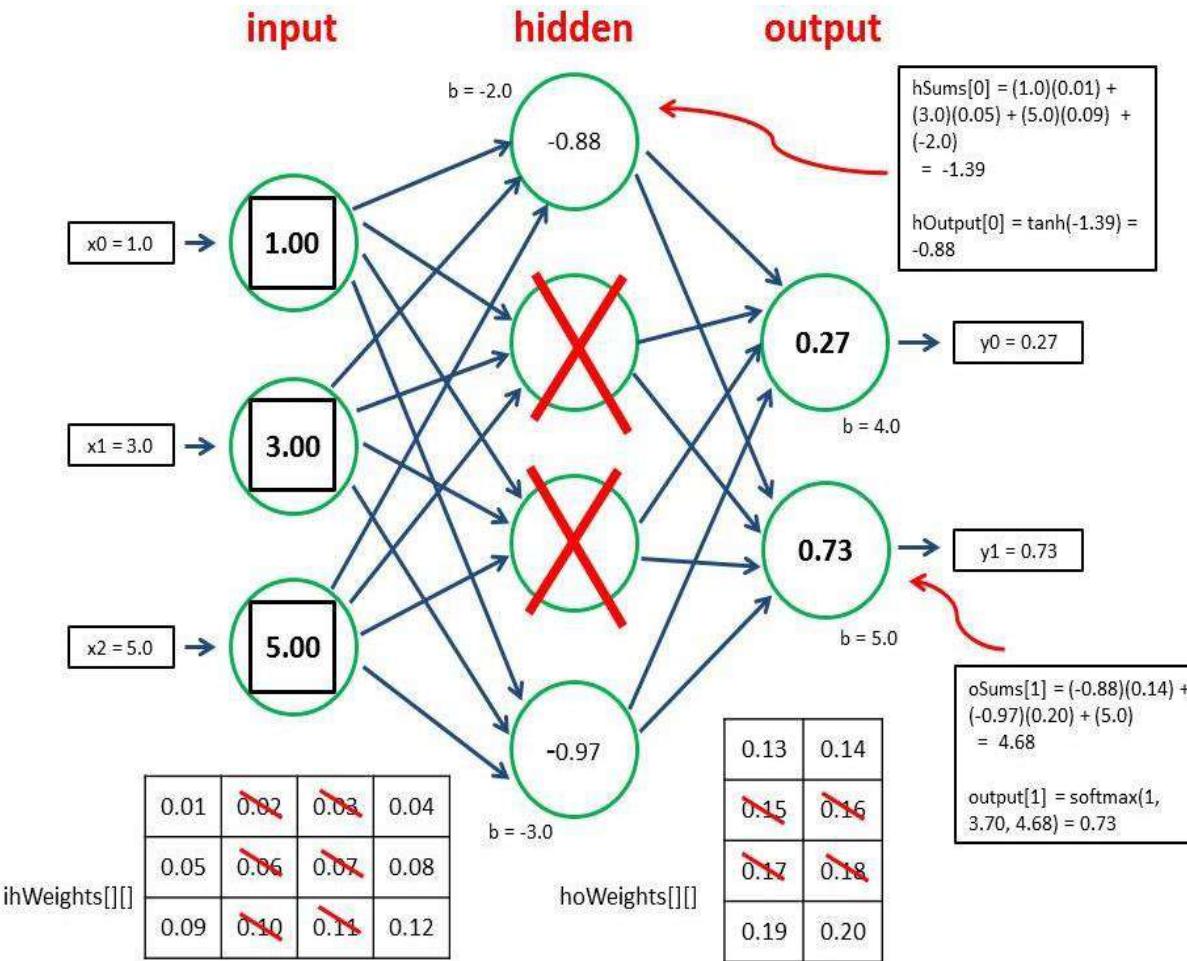


Figure 10: Effect of varying data set size.

# Early Stopping

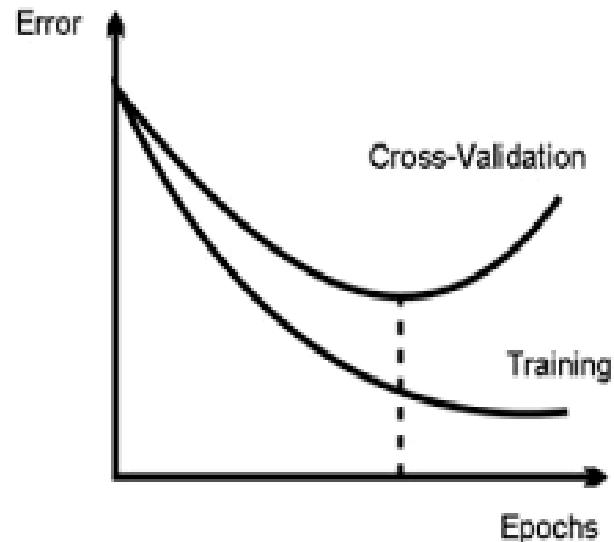


Figure 2: Profiles for training and cross-validation errors.

If we let a complex model train long enough on a given data set it can eventually learn the data exactly.

Given data that isn't represented in the training set, the model will perform poorly when analyzing the data (overfitting).

How is the sweet spot for training located?

When the error on the training set begins to deviate from the error on the validation set, a threshold can be set to determine the early stopping condition and the ideal number of epochs to train.

# Other Regularization Methods

- Increase the data by **data augmentation**
- Flip, rotate, scale, translate, deform ....



- Normalize training examples to **speed up your training**





# Weight initialization and training

- Exploding and vanishing gradient is an issue
- Set the variance of weights to be  $1/n$
- For ReLu it is better to use variance as  $2/n$
- For tanh use variance  $\sqrt{1/n}$  called **xavier** initialization
- **Adam** optimization, uses momentum and RMSProp simultaneously





# Demo

[A Neural Network Playground  
\(tensorflow.org\)](#)

# References

---

Mitesh Khapra

<https://www.youtube.com/watch?v=yw8xwS15Pf4>

Back propagation

[https://www.youtube.com/watch?v=G5b4jRBKNxw&list=PLZbbT5o\\_s2xq7Lwl2y8\\_QtvuXZedL6tQU&index=25](https://www.youtube.com/watch?v=G5b4jRBKNxw&list=PLZbbT5o_s2xq7Lwl2y8_QtvuXZedL6tQU&index=25)



# Thank You!

Next Session: CNN



# C6: ANN and Deep Learning



**BITS** Pilani  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)



**Session 3  
Date – 14<sup>th</sup> July 2024**

**Time – 10 am to 12.15pm**

These slides are prepared by the instructor, with grateful acknowledgement of and many others who made their course materials freely available online.

# Agenda

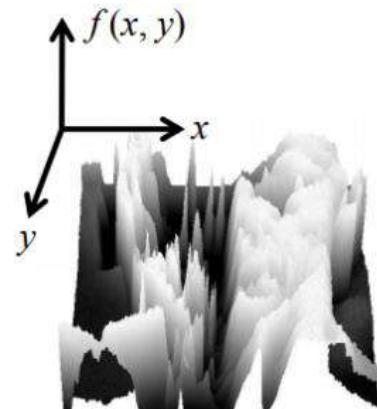
- Computer Vision
- CNN
- Pooling
- Variants of pooling functions
- CNN with Fully connected Networks
- RCNN
- Faster RCNN

# Image

Grayscale image can be considered as a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$

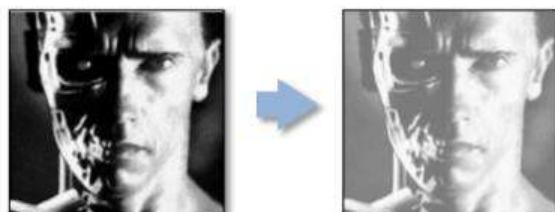


[snoop](#)

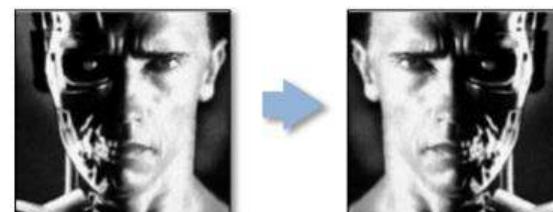


[3D view](#)

- Being digital adds quantization and sampling. We may apply operators on this function



$$g(x, y) = f(x, y) + 20$$



$$g(x, y) = f(-x, y)$$

# Computer Vision - history

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
PROJECT MAC

Artificial Intelligence Group  
Vision Memo. No. 100.

July 7, 1966

THE SUMMER VISION PROJECT  
Seymour Papert

The summer vision project is an attempt to use our summer workers effectively in the construction of a significant part of a visual system. The particular task was chosen partly because it can be segmented into sub-problems which will allow individuals to work independently and yet participate in the construction of a system complex enough to be a real landmark in the development of "pattern recognition".

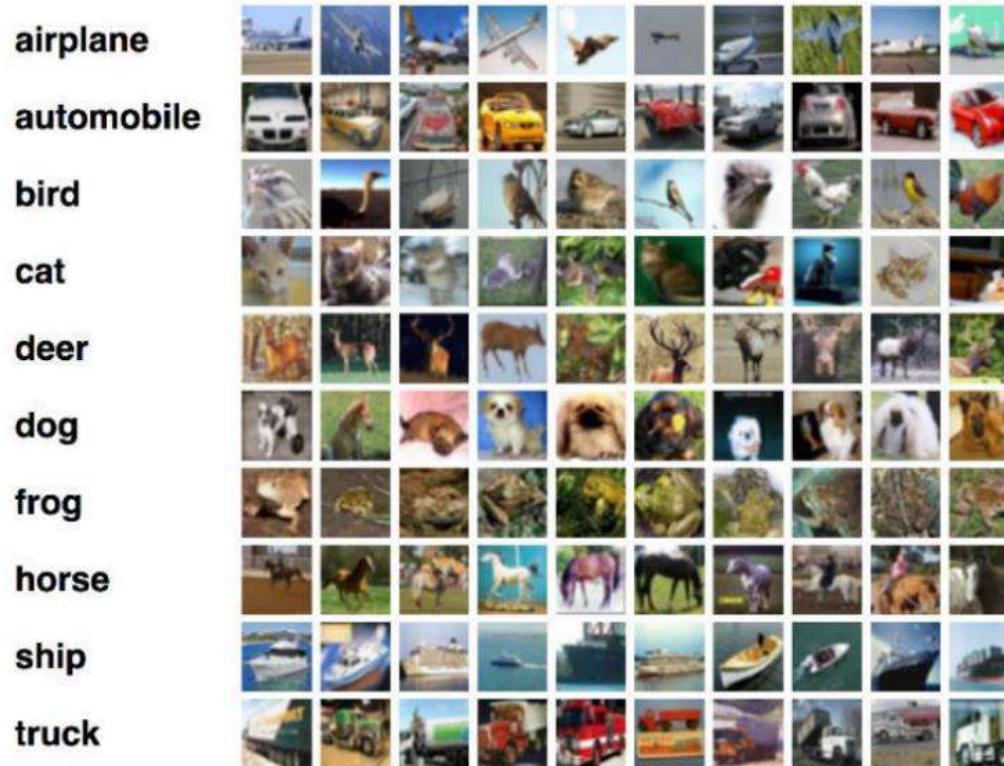
## Some Tasks

- Classification
- Annotation
- Detection
- Segmentation

How the Afghan Girl was Identified by her Iris Patterns 1984-2002



# Classification



# Classification Challenges

Viewpoint variation



Scale variation



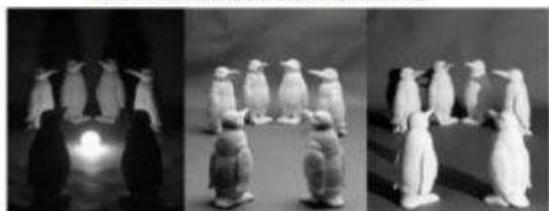
Deformation



Occlusion



Illumination conditions



Background clutter



Intra-class variation



# Annotation

					
Predicted keywords	sky, jet, plane, smoke, formation	grass, rocks, sand, valley, canyon	sun, water, sea, waves, birds	water, tree, grass, deer, white-tailed	bear, snow, wood, deer, white-tailed
Human annotation	sky, jet, plane, smoke	rocks, sand, valley, canyon	sun, water, clouds, birds	tree, forest, deer, white-tailed	tree, snow, wood, fox

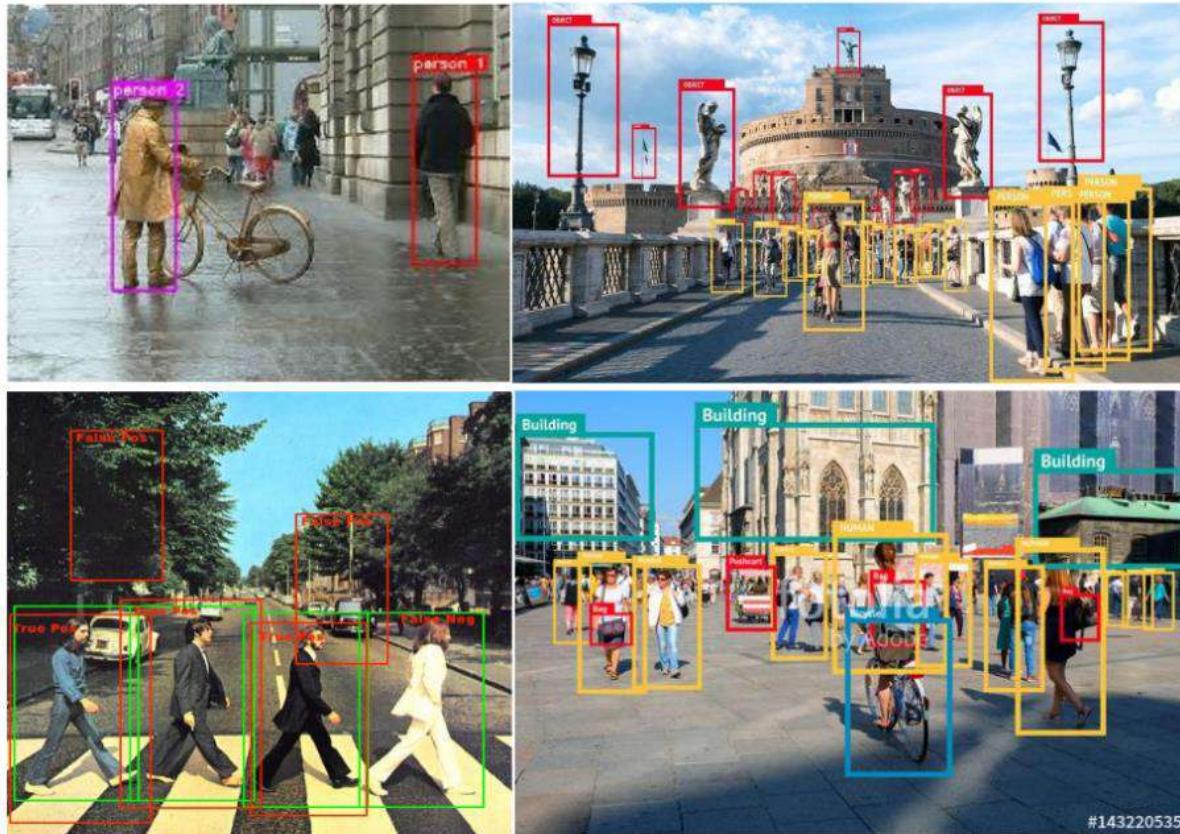


Prediction for queries: Sky (Row1), Street (Row2), Mare (Row3) and Train (Row4)

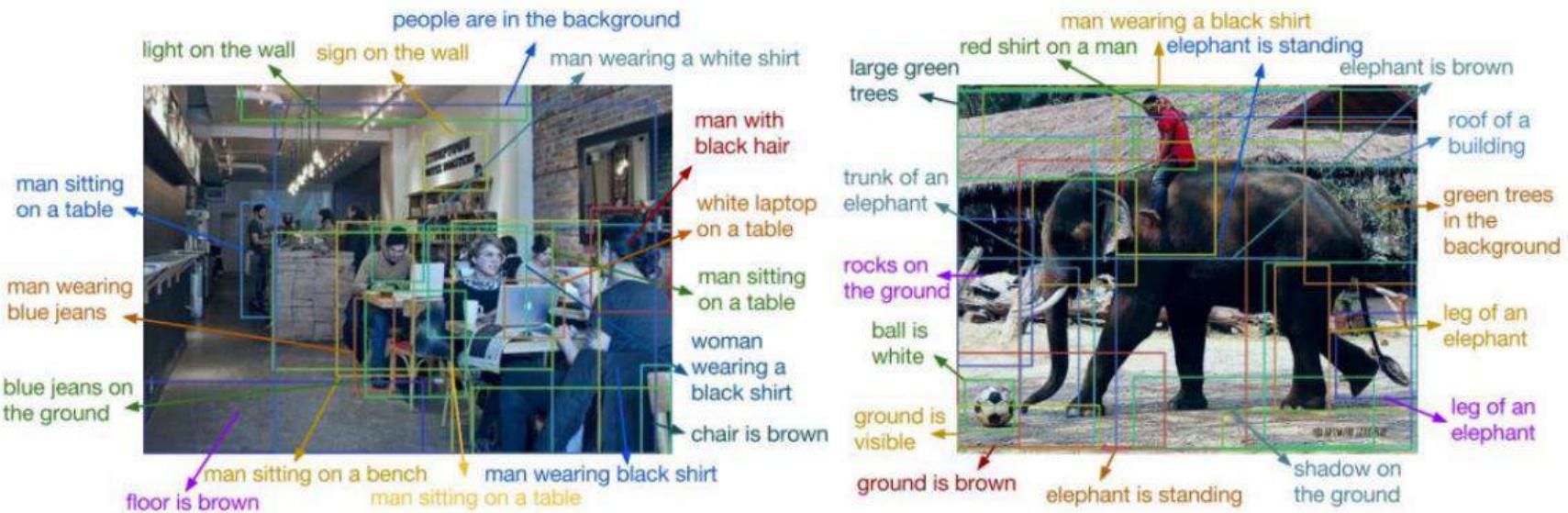
Ref: A. Makadia, V. Pavlovic, and S. Kumar.

A New Baseline for Image Annotation. In Proceedings of ECCV 08.

# Detection



# Description

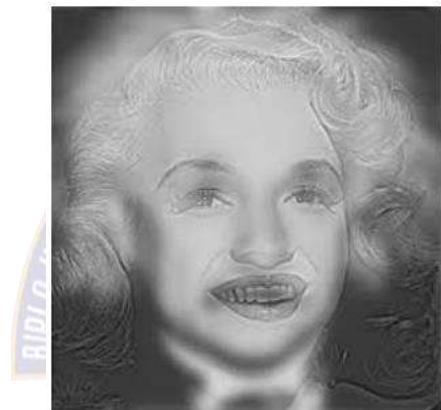


# Segmentation

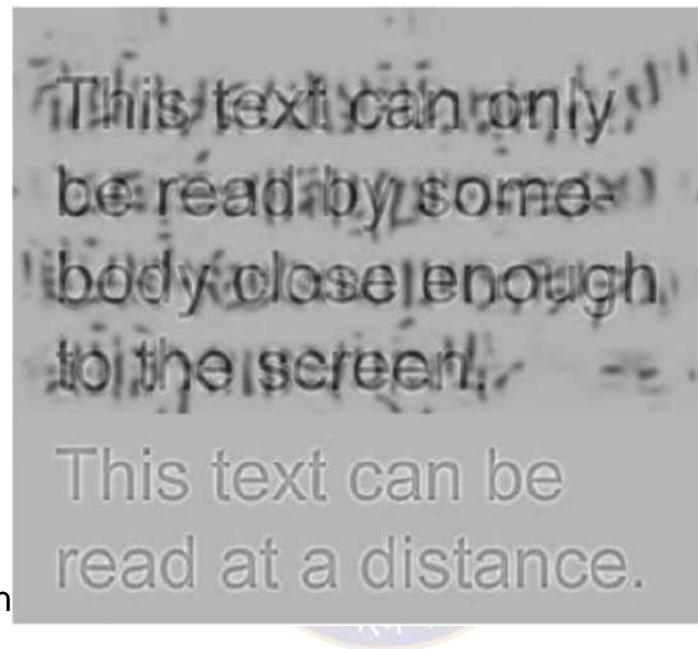


Vijay Badrinarayanan, Alex Kendall and Roberto Cipolla "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." PAMI, 2017.

# Hybrid Images



# Example Hybrid Image



The hybrid font becomes in  
relatively long distances.

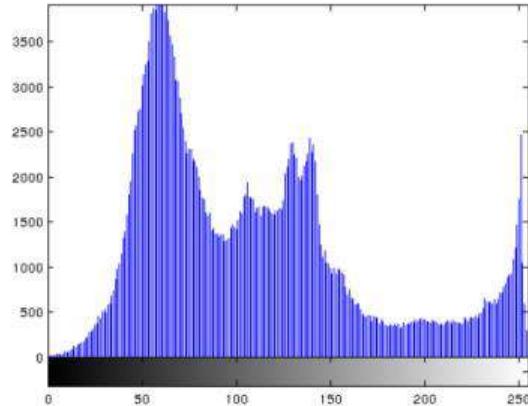
ns easy to read at

# Histogram

- Discrete probability distribution of the image intensity values



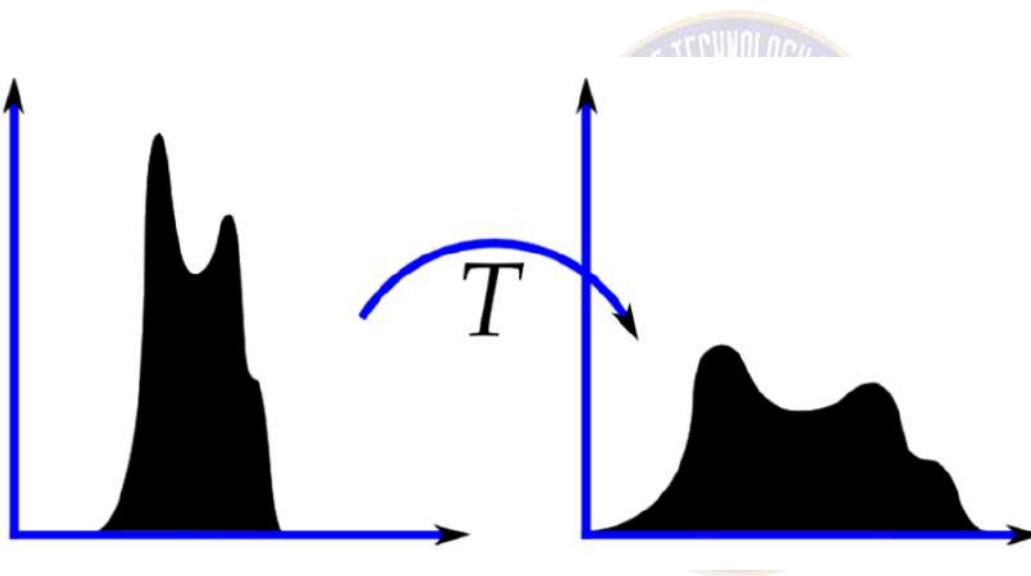
(a) Image



(b) Histogram

# Histogram Equalization

One method to enhance images is to equalize the histogram



Exercise: Prove that an image transformed by its cumulative distribution function results in an image with uniform histogram. More generally, histogram can be modified by histogram specification

# Shuffling the pixels

- What happens if we shuffle all the pixels in an image randomly?

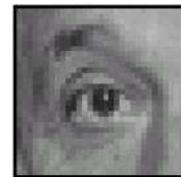


- Different images may have same histogram
- Need more local operators

# Linear filtering

 $*$ 

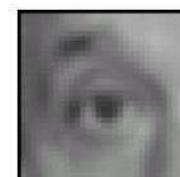
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

 $=$ 

Original

 $*$ 

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

 $=$ 

Blur (with a mean filter)

 $*$ 

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

 $=$ 

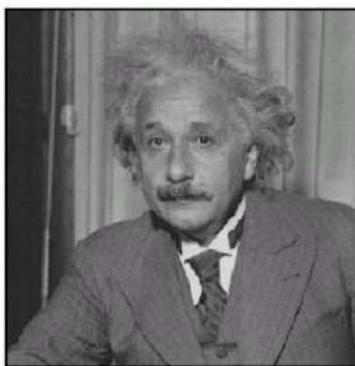
Original

Shifted left By 1 pixel

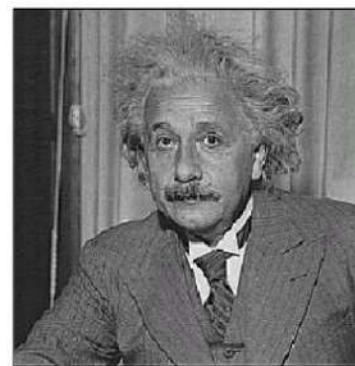


$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$- \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Sharpening filter  
(accentuates edges)

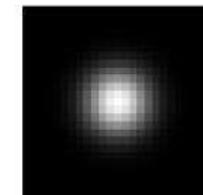
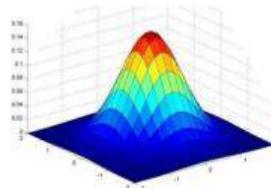
before



after

# Gaussian filter

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



Act as low-pass filter as it removes high-frequency components. Convolution with self is another Gaussian.



$\sigma = 1$  pixel



$\sigma = 5$  pixels



$\sigma = 10$  pixels

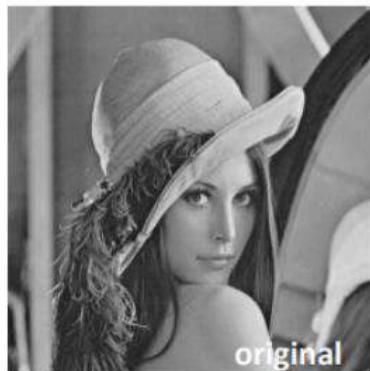


$\sigma = 30$  pixels

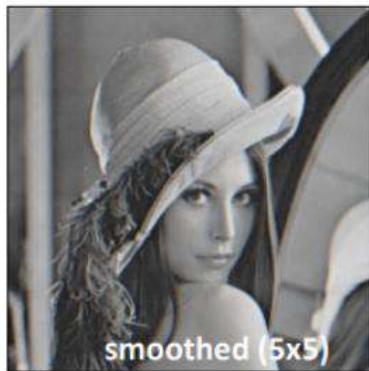
Convolving twice with Gaussian kernel of width  $\sigma$  is equivalent to convolving once with Gaussian kernel of width  $\sqrt{2}\sigma$

# Sharpening

What does blurring take away?



-



=



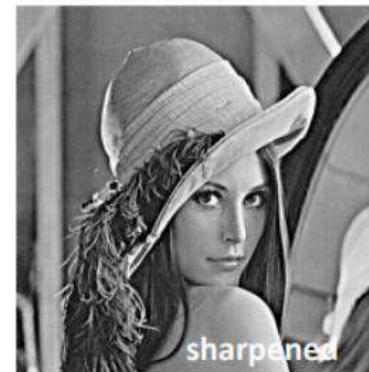
Let's add it back:



$+\alpha$



=



# Edge detection

Common approximation of derivative of Gaussian

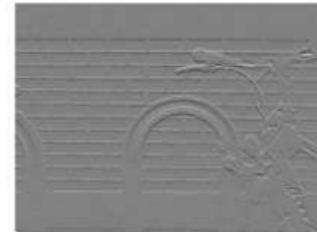
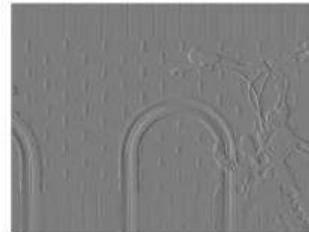
$$\frac{1}{8} \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$s_x$

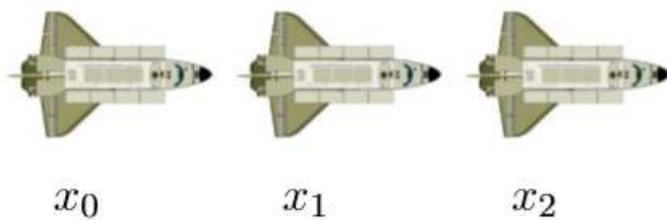
$$\frac{1}{8} \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

$s_y$

Edge magnitude is  $\sqrt{s_x^2 + s_y^2}$  and the direction is  
Edge is detected by threshold  $\tan^{-1}(s_y/s_x)$



# Convolution



$$s_t = \sum_{a=0}^{\infty} x_{t-a} w_{-a} = (x * w)_t$$

Diagram illustrating the convolution operation:

- input**: The sequence of measurements  $x_0, x_1, x_2$ .
- filter**: The filter  $w$  applied to the input sequence.
- convolution**: The result of the convolution operation, which is the weighted sum of the input measurements.

- Suppose we are tracking the position of an aeroplane using a laser sensor at discrete time intervals
- Now suppose our sensor is noisy
- To obtain a less noisy estimate we would like to average several measurements
- More recent measurements are more important so we would like to take a weighted average

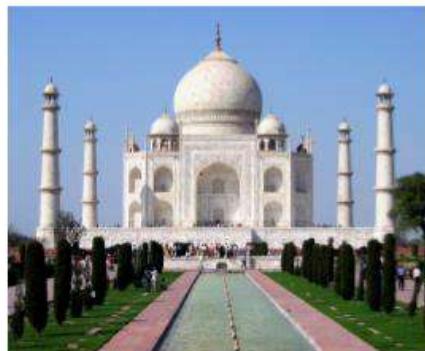
# Convolution



- We can think of images as 2D inputs
- We would now like to use a 2D filter ( $m \times n$ )
- First let us see what the 2D formula looks like
- This formula looks at all the preceding neighbours ( $i - a, j - b$ )

$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i-a,j-b} K_{a,b}$$

# Convolution



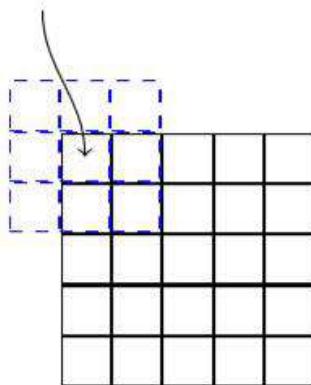
$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a, j+b} K_{a,b}$$

- We can think of images as 2D inputs
- We would now like to use a 2D filter ( $m \times n$ )
- First let us see what the 2D formula looks like
- This formula looks at all the preceding neighbours ( $i - a, j - b$ )
- In practice, we use the following formula which looks at the succeeding neighbours

# Convolution

$$S_{ij} = (I * K)_{ij} = \sum_{a=\lfloor -\frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{b=\lfloor -\frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} I_{i-a, j-b} K_{\frac{m}{2}+a, \frac{n}{2}+b}$$

pixel of interest



- For the rest of the discussion we will use the following formula for convolution
- In other words we will assume that the kernel is centered on the pixel of interest
- So we will be looking at both preceding and succeeding neighbors

Let us apply this idea to a toy example and see the results

**Input**

a	b	c	d
e	f	g	h
i	j	k	A

**Kernel**

w	x
y	z

**Outpu**

aw+bx+ey+fz	t	

Let us apply this idea to a toy example and see the results

Input			
a	t	b	c
e	f	g	h
i	j	k	A

Kernel			
w	x		
y	z		

Output		
$aw + bx + ey + fz$	$t$ $bw + cx + fy + gz$	

Let us apply this idea to a toy example and see the results

Input

a	t	c	d
e	f	g	h
i	j	k	A

Kernel

w	x
y	z

Output

aw+bx+ey+fz	t bw+cx+fy+gz	cw+dx+gy+hz

Let us apply this idea to a toy example and see the results

Input

a	t b	c	d
e	f	g	h
i	j	k	A

Kernel

w	x
y	z

Output

$aw+bx+ey+fz$	$bw+cx+fy+gz$	$cw+dx+gy+hz$
$ew+fx+iy+jz$		

Let us apply this idea to a toy example and see the results

Input

a	t b	c	d
e	f	g	h
i	j	k	A

Kernel

w	x
y	z

Output

$aw+bx+ey+fz$	$bw+cx+fy+gz$	$cw+dx+gy+hz$
$ew+fx+iy+jz$	$fw+gx+jy+kz$	

Let us apply this idea to a toy example and see the results

Input

a	t	c	d
e	f	g	h
i	j	k	A

Kernel

w	x
y	z

Output

$aw+bx+ey+fz$	$bw+cx+fy+gz$	$cw+dx+gy+hz$
$ew+fx+iy+jz$	$fw+gx+jy+kz$	$gw+hx+ky+Az$

# Example of kernel: Blur



$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} =$$

**blurs the image**



# Example of kernel: Edge detection

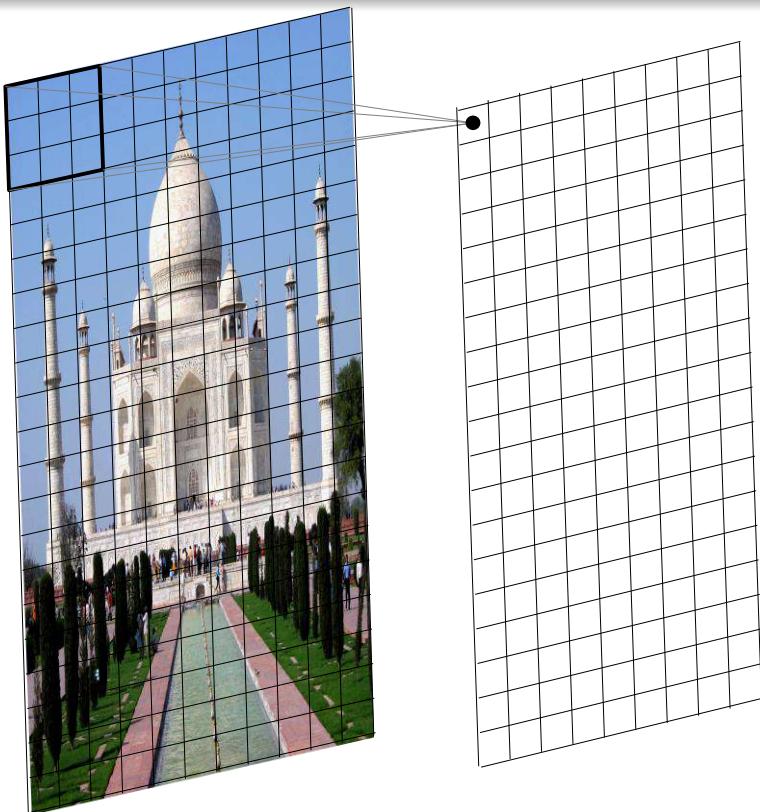


$$\begin{array}{ccc} 1 & 1 & 1 \\ 1 & -8 & 1 = \\ 1 & 1 & 1 \end{array}$$

**detects the edges**

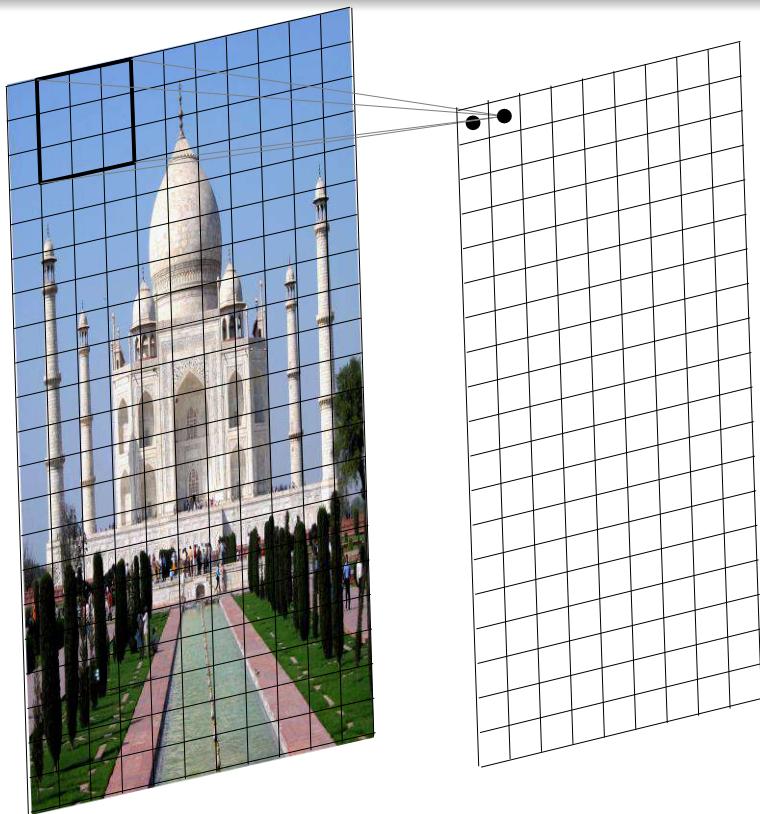


# Convolution



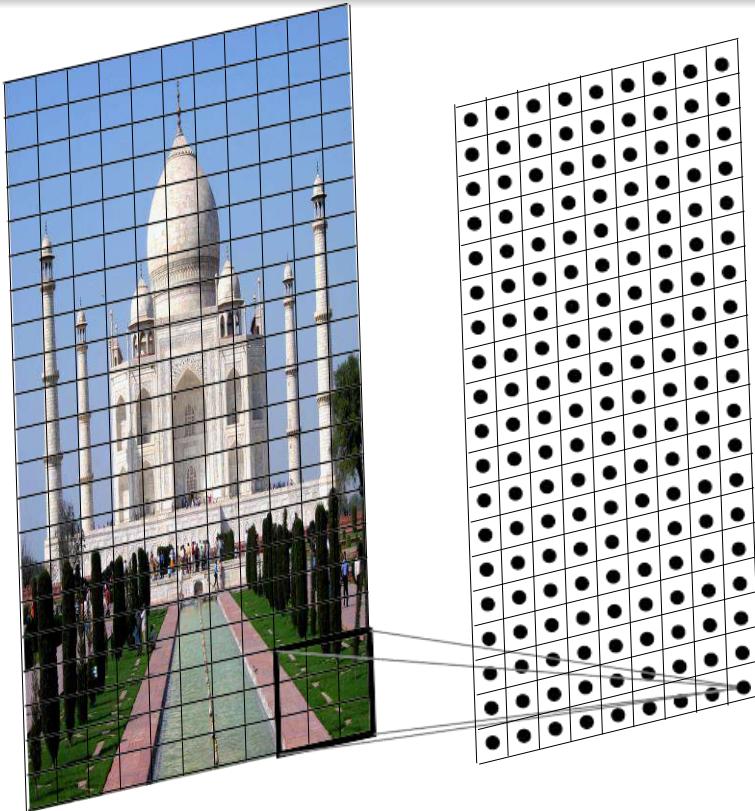
- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

# Convolution

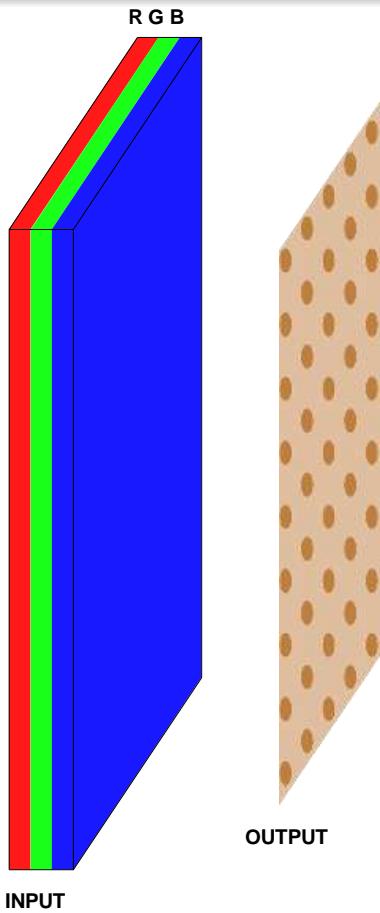


- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output

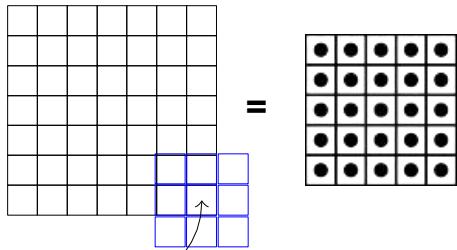
# 2D convolutions applied to images



- We just slide the kernel over the input image
- Each time we slide the kernel we get one value in the output
- The resulting output is called a feature map.
- We can use multiple filters to get multiple feature maps.

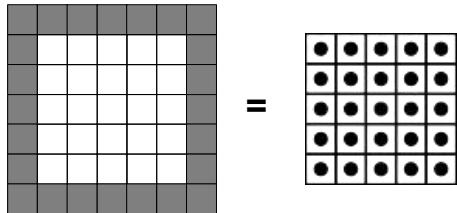


- What would a 3D filter look like?
- It will be 3D and we will refer to it as a volume
- Once again we will slide the volume over the 3D input and compute the convolution operation
- We will assume that the filter always extends to the depth of the image
- In effect, we are doing a 2D convolution operation on a 3D input (because the filter moves along the height and the width but not along the depth)
- As a result the output will be 2D (only width and height, no depth)
- Once again we can apply multiple filters to get multiple feature maps

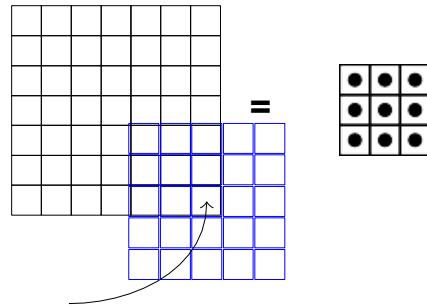


**pixel of  
interest**

- Let us compute the dimension  $(W_2, H_2)$  of the output
- Notice that we can't place the kernel at the corners as it will cross the input boundary

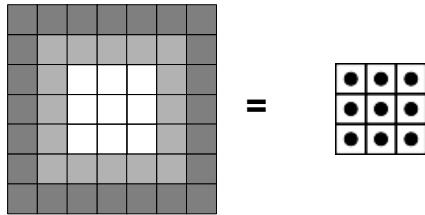


- Let us compute the dimension  $(W_2, H_2)$  of the output
- Notice that we can't place the kernel at the corners as it will cross the input boundary
- This is true for all the shaded points (the kernel crosses the input boundary)
- This results in an output which is of smaller dimensions than the input
- As the size of the kernel increases, this becomes true for even more pixels



**pixel of  
interest**

- Let us compute the dimension  $(W_2, H_2)$  of the output
- Notice that we can't place the kernel at the corners as it will cross the input boundary
- This is true for all the shaded points (the kernel crosses the input boundary)
- This results in an output which is of smaller dimensions than the input
- As the size of the kernel increases, this becomes true for even more pixels
- For example, let's consider a  $5 \times 5$  kernel
- We have an even smaller output now



***In general,***  $W_2 = W_1 - F + 1$   
 $H_2 = H_1 - F + 1$   
***We will refine this formula further***

- Let us compute the dimension  $(W_2 \ H_2)$  of the output
- Notice that we can't place the kernel at the corners as it will cross the input boundary
- This is true for all the shaded points (the kernel crosses the input boundary)
- This results in an output which is of smaller dimensions than the input
- As the size of the kernel increases, this becomes true for even more pixels
- For example, let's consider a  $5 \times 5$  kernel
- We have an even smaller output now

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

=

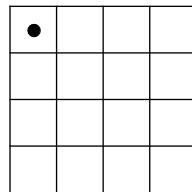
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•

- What if we want the output to be of same size as the input?
- We can use something known as padding
- Pad the inputs with appropriate number of 0 inputs so that you can now apply the kernel at the corners
- Let us use pad  $P = 1$  with a  $3 \times 3$  kernel
- This means we will add one row and one column of 0 inputs at the top, bottom, left and right

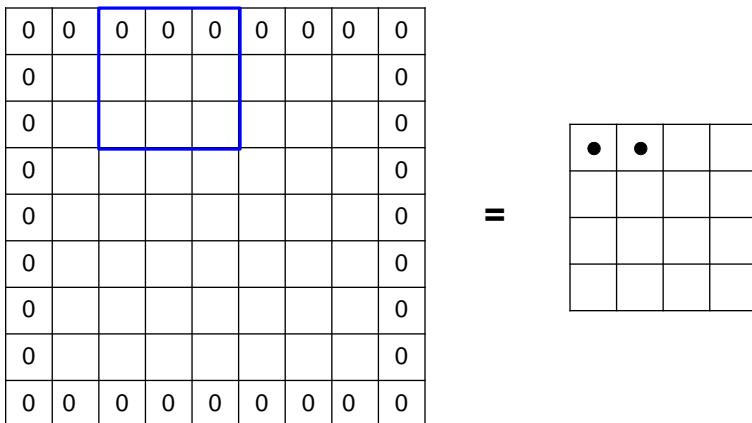
We now have,  $W_2 = W_1 - F + 2P + 1$   $H_2 = H_1 - F + 2P + 1$

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

=



- What does the stride S do?
- It defines the intervals at which the filter is applied (here  $S = 2$ )
- Here, we are essentially skipping every 2nd pixel which will again result in an output which is of smaller dimensions

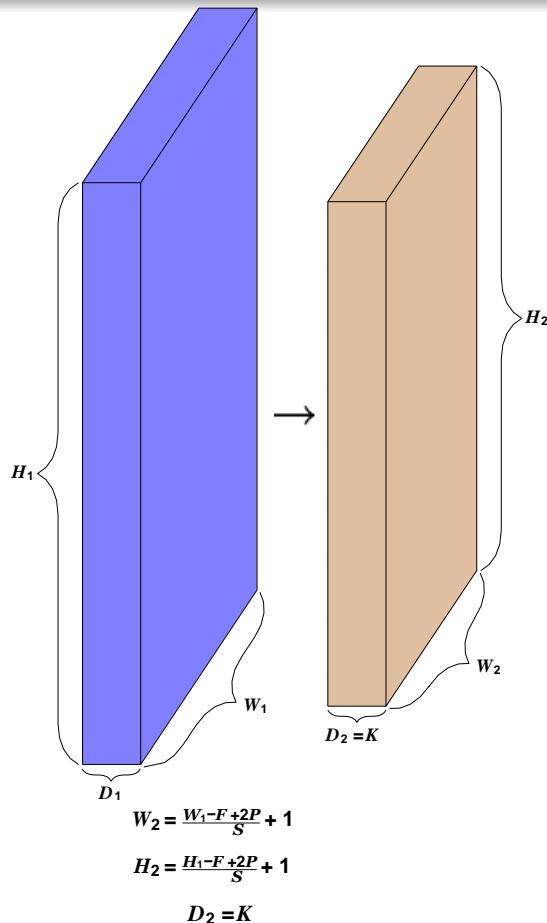


- What does the stride S do?
- It defines the intervals at which the filter is applied (here  $S = 2$ )
- Here, we are essentially skipping every 2nd pixel which will again result in an output which is of smaller dimensions

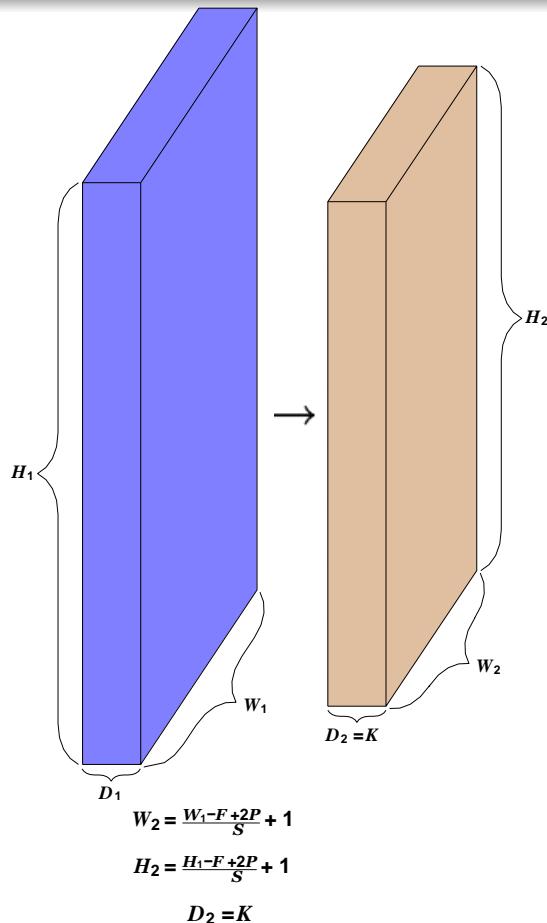
**So what should our final formula look like,**

$$\frac{W_1 - F + 2P}{S} + 1$$

$$\frac{H_1 - F + 2P}{S} + 1$$

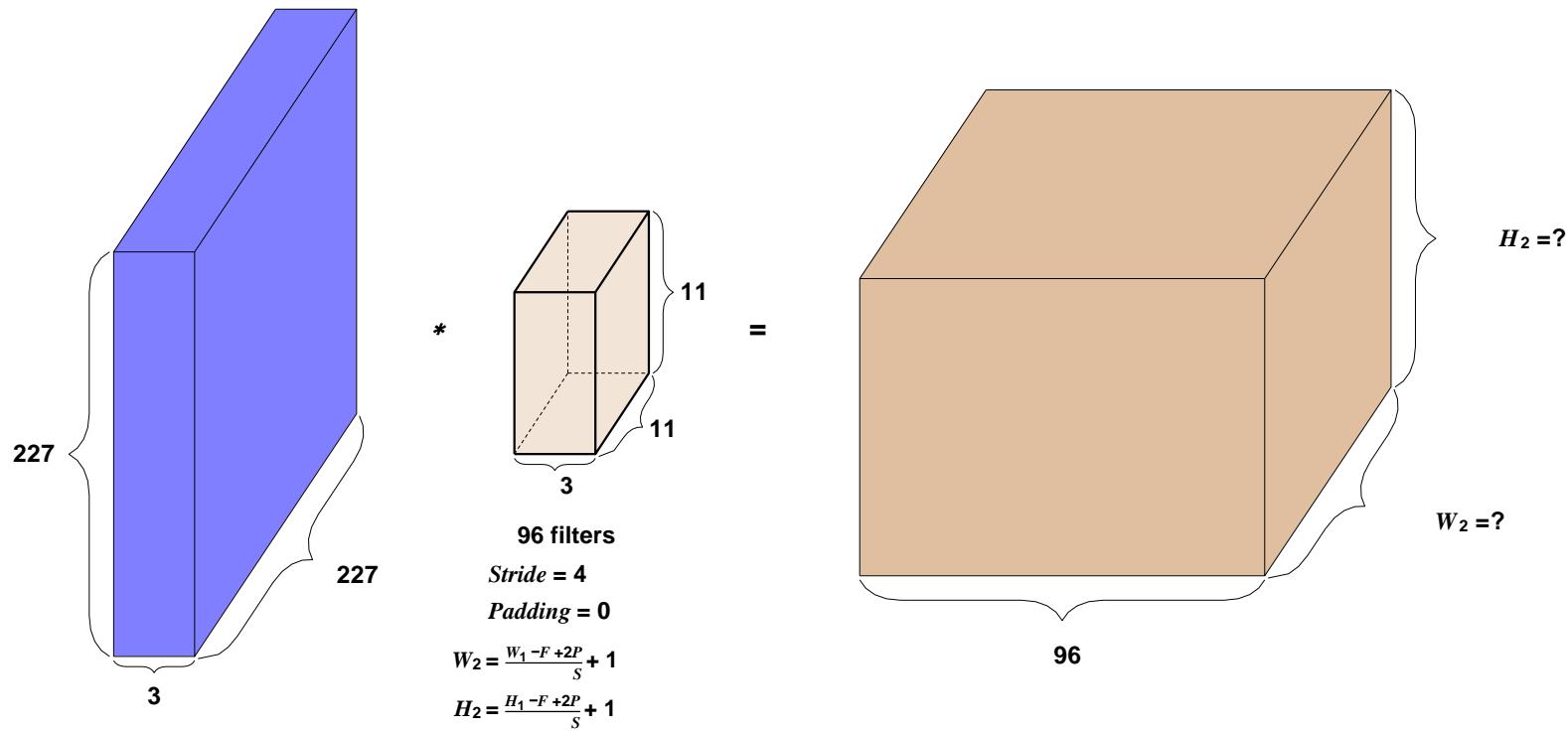


- Finally, coming to the depth of the output.
- **Each filter gives us one 2D output.**
- **$K$  filters will give us  $K$  such 2D outputs**

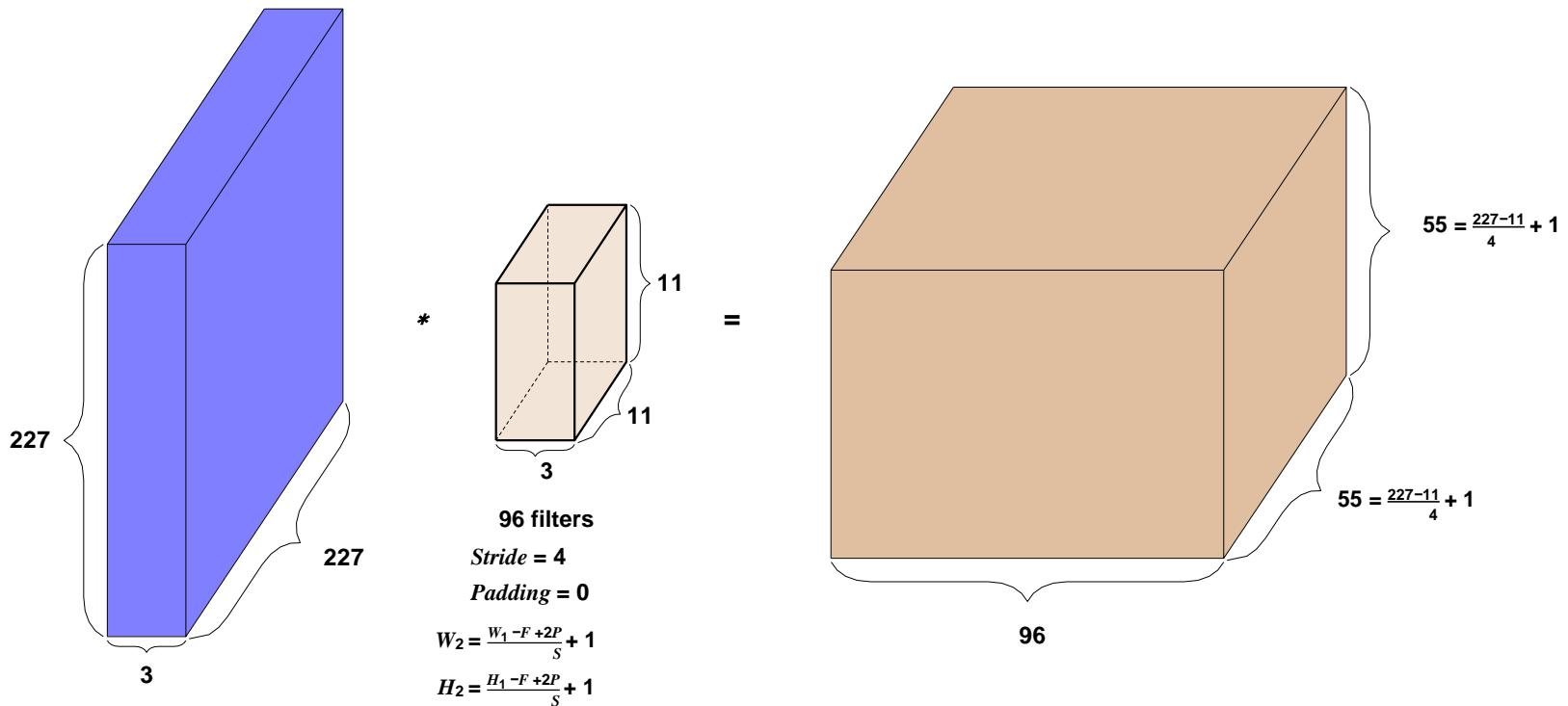


- Finally, coming to the depth of the output.
- **Each filter gives us one 2D output.**
- **$K$  filters will give us  $K$  such 2D outputs**
- **We can think of the resulting output as  $K \times W_2 \times H_2$  volume**
- **Thus  $D_2 = K$**

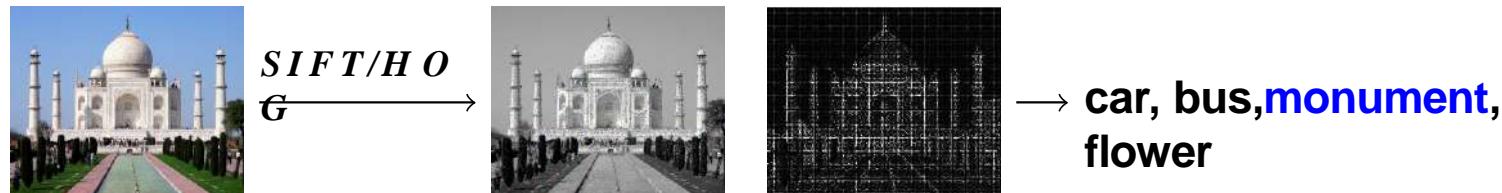
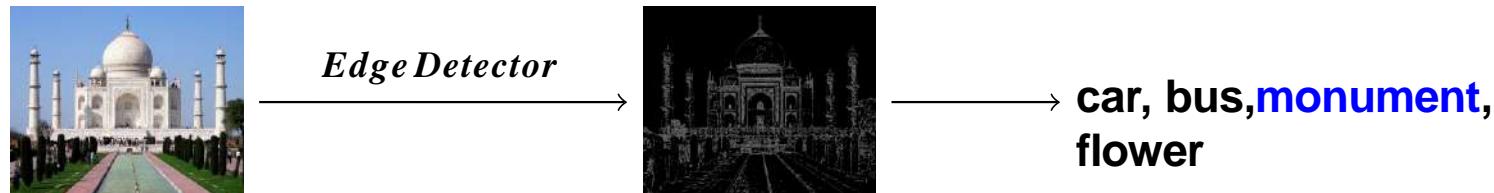
# Example



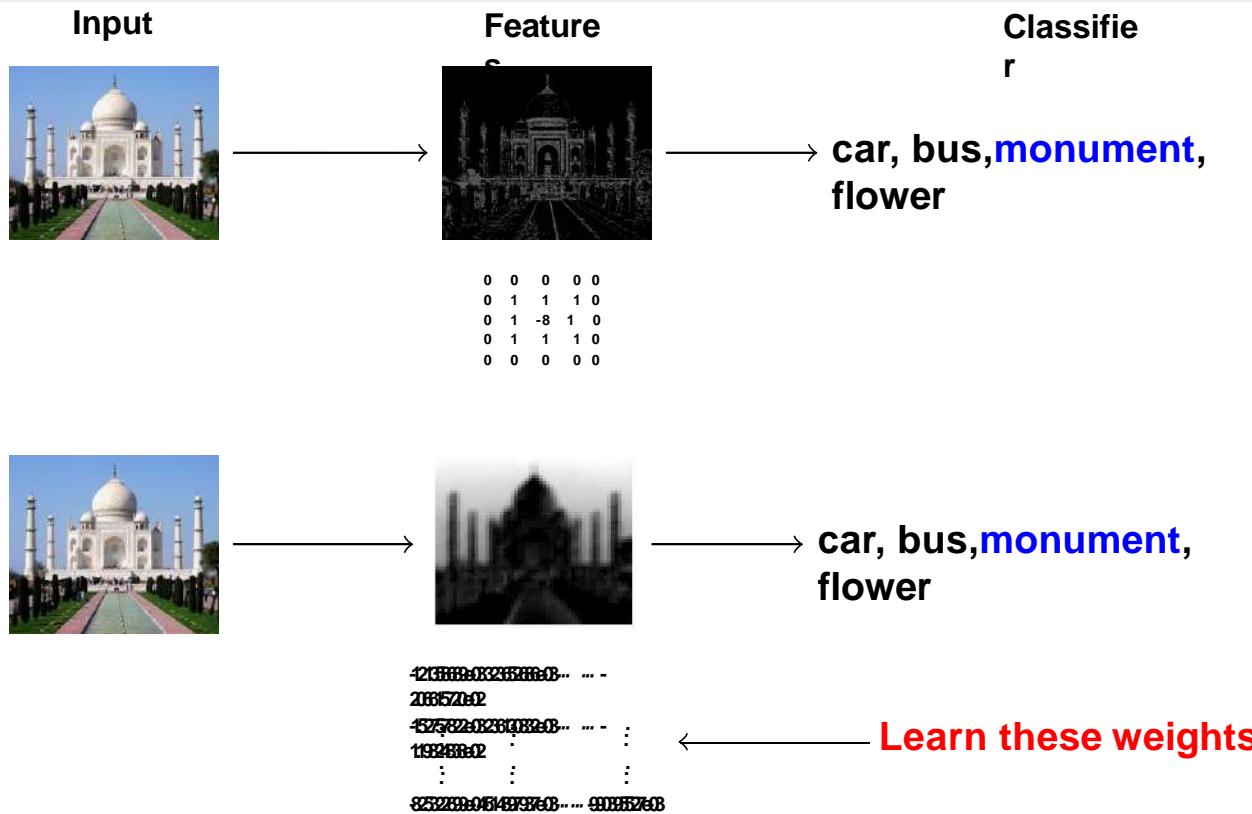
# Example



# Features

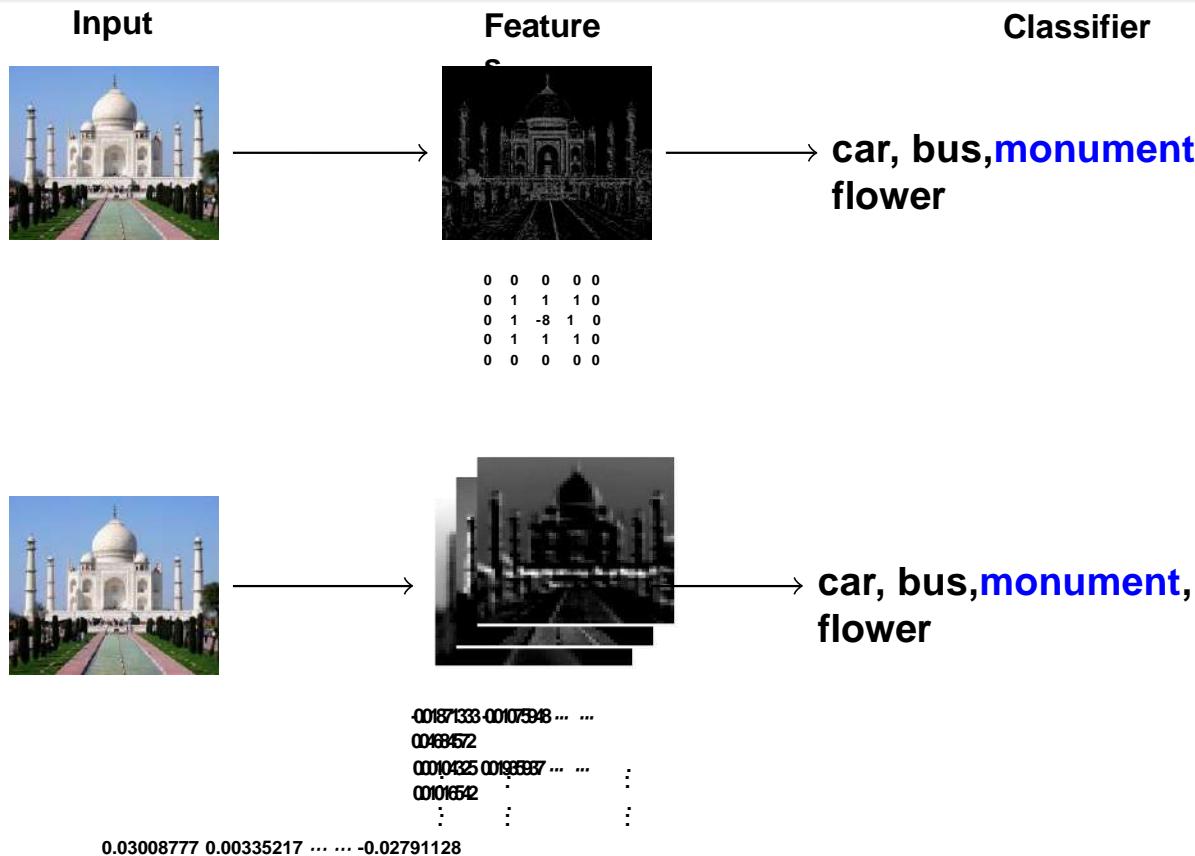


# Example



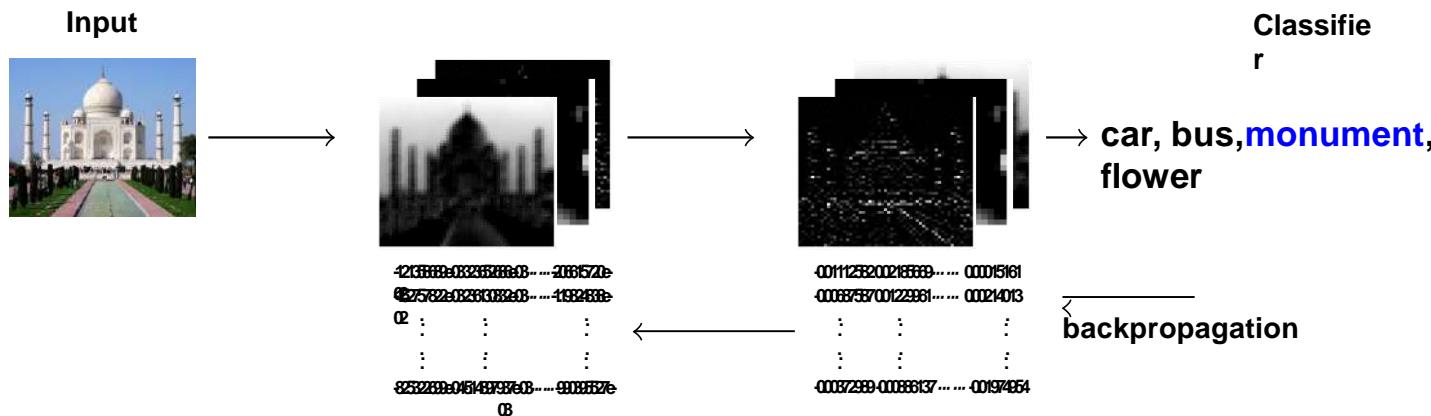
- Instead of using handcrafted kernels such as edge detectors can we learn meaningful kernels/filters in addition to learning the weights of the classifier?

# Example

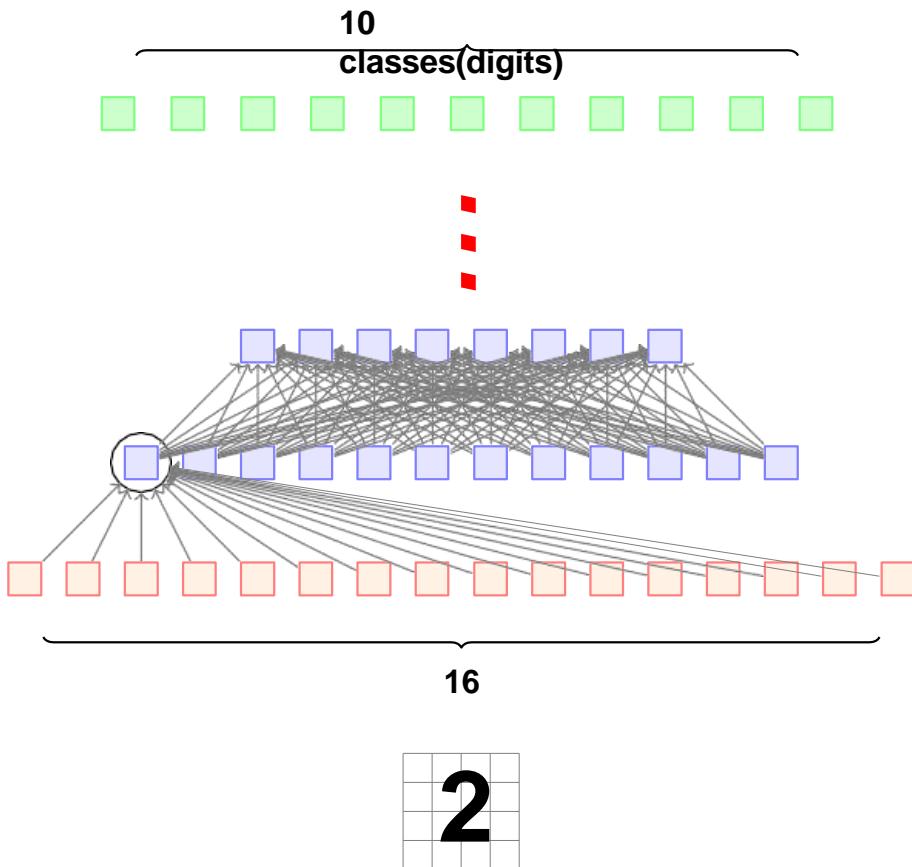


- Even better: Instead of using handcrafted kernels (such as edge detectors) can we learn **multiple** meaningful kernels/filters in addition to learning the weights of the classifier?

# Convolutional Neural Network

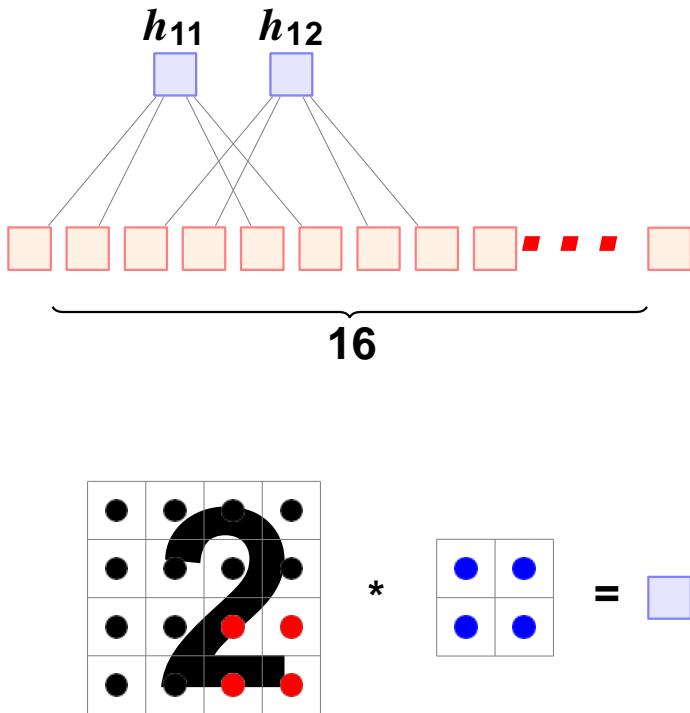


- Can we learn multiple **layers** of meaningful kernels/filters in addition to learning the weights of the classifier?
- Yes, we can !
- Simply by treating these kernels as parameters and learning them in addition to the weights of the classifier (using back propagation)
- Such a network is called a **Convolutional Neural Network**.



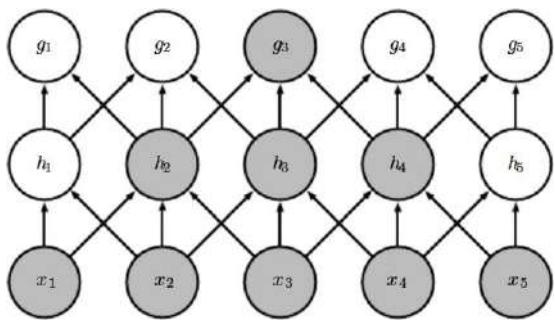
- This is what a regular feed-forward neural network will look like
- **There are many dense connections here**
- **For example all the 16 input neurons are contributing to the computation of  $h_{11}$**
- **Contrast this to what happens in the case of convolution**

# Sparse Connectivity



- Only a few local neurons participate in the computation of  $h_{11}$
- For example, only pixels 1, 2, 5, 6 contribute to  $h_{11}$
- The connections are much sparser
- We are taking advantage of the structure of the image (interactions)
- This sparse connectivity reduces the number of parameters in the model

# Sparse Connectivity

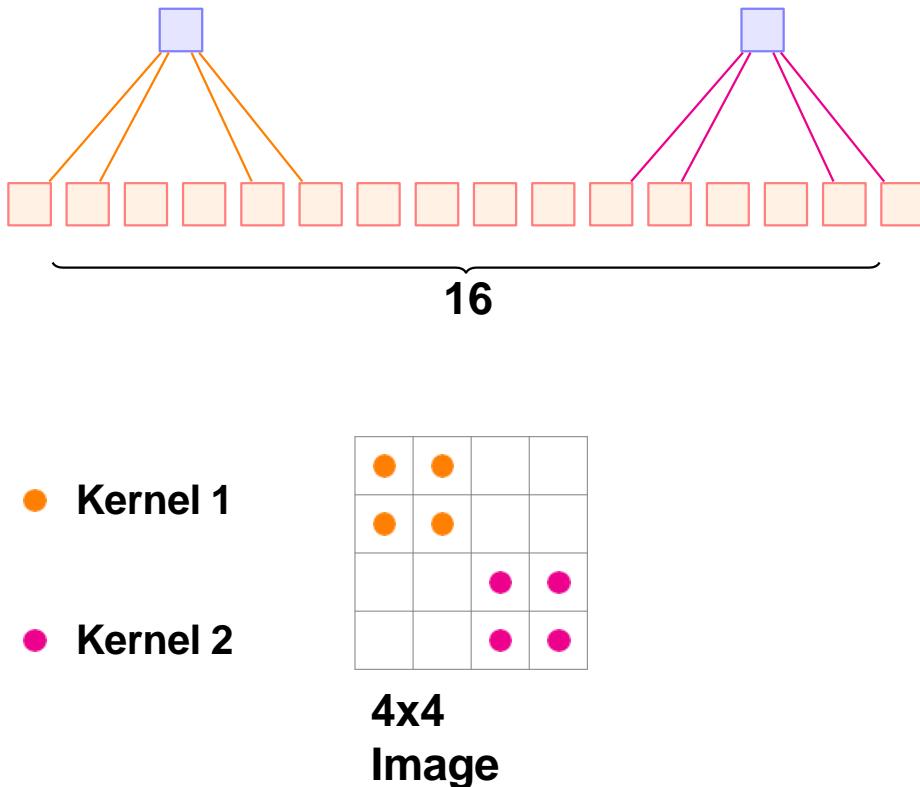


- But is sparse connectivity really good thing ?
- Aren't we losing information (by losing interactions between some input pixels)
- Well, not really
- The two highlighted neurons ( $x_1$  &  $x_5$ ) do not interact in *layer 1*
- But they indirectly contribute to the computation of  $g_3$  and hence interact indirectly

---

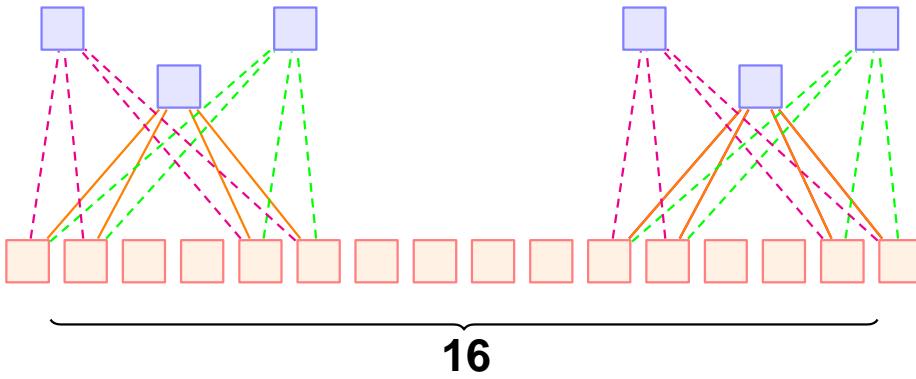
\*Goodfellow-et-al-2016

# Weight-sharing



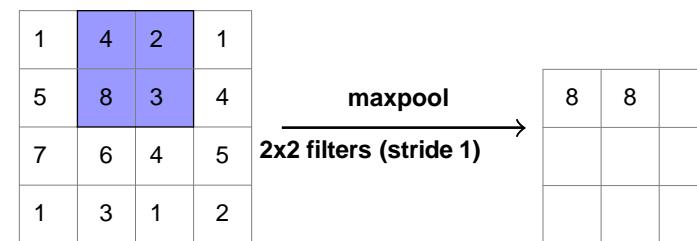
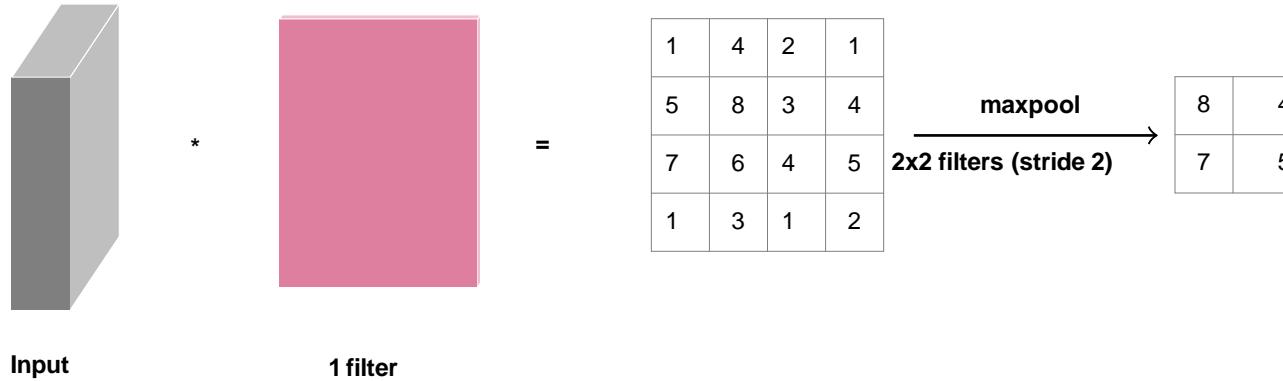
- Another characteristic of CNNs is **weight sharing**
- Consider the following network
- Do we want the kernel weights to be different for different portions of the image?
- Imagine that we are trying to learn a kernel that detects edges
- Shouldn't we be applying the same kernel at all the portions of the image?

# Weight-sharing

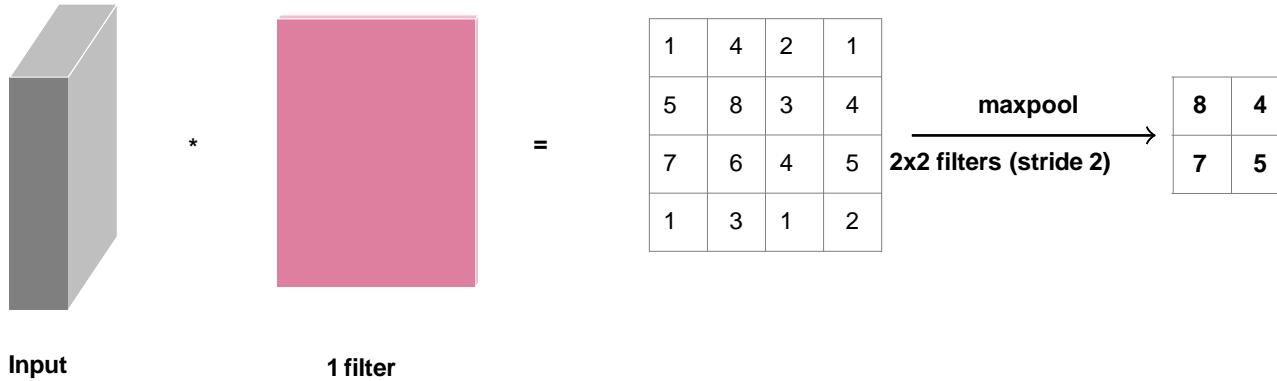


- In other words shouldn't the *orange* and *pink* kernels be the same
- Yes, indeed
- This would make the job of learning easier(instead of trying to learn the same weights/kernels at different locations again and again)
- But does that mean we can have only one kernel?
- No, we can have many such kernels but the kernels will be shared by all locations in the image
- This is called “weight sharing”

# Pooling



# Pooling



The diagram illustrates max pooling with stride 1. It shows the same 4x4 input matrix as above. A horizontal arrow labeled "maxpool" points to a 3x3 output matrix:

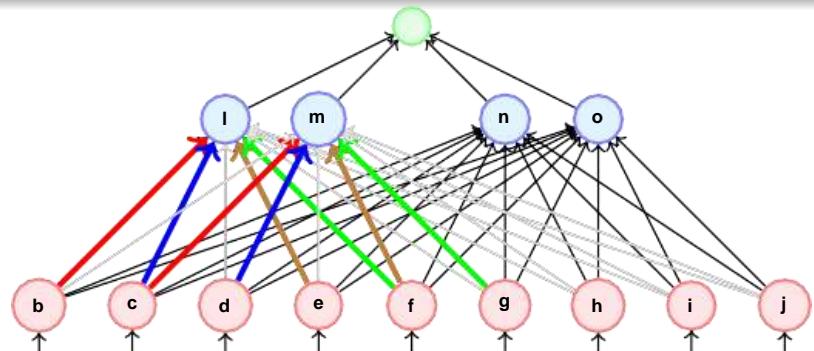
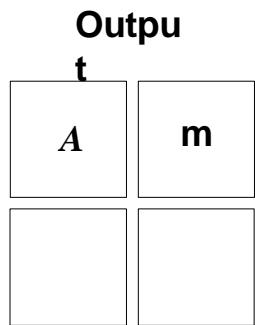
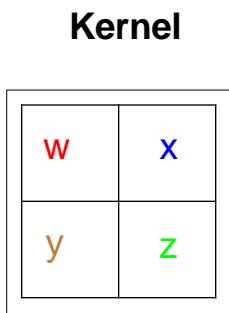
8	8	4
8	8	5
7	6	5

Below the input matrix, the text "2x2 filters (stride 1)" indicates the kernel size and stride.

- Instead of max pooling we can also do average pooling

# Training CNN

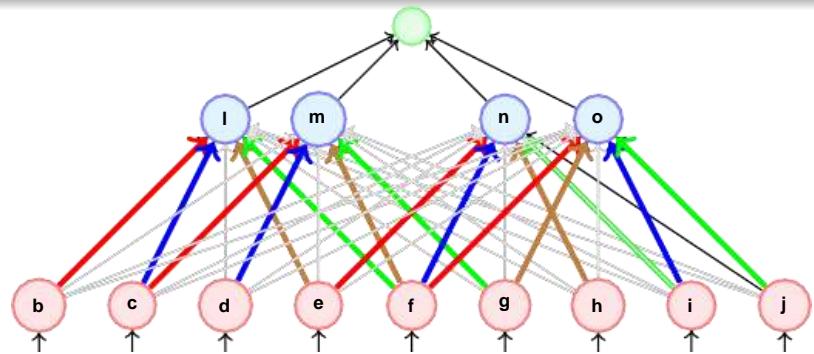
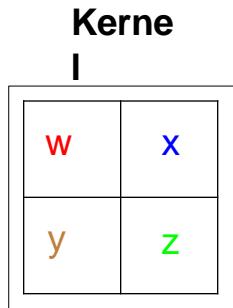
Input		
b	c	d
e	f	g
h	i	j



- A CNN can be implemented as a feedforward neural network
- wherein only a few weights(in color) are active
- the rest of the weights (in gray) are zero

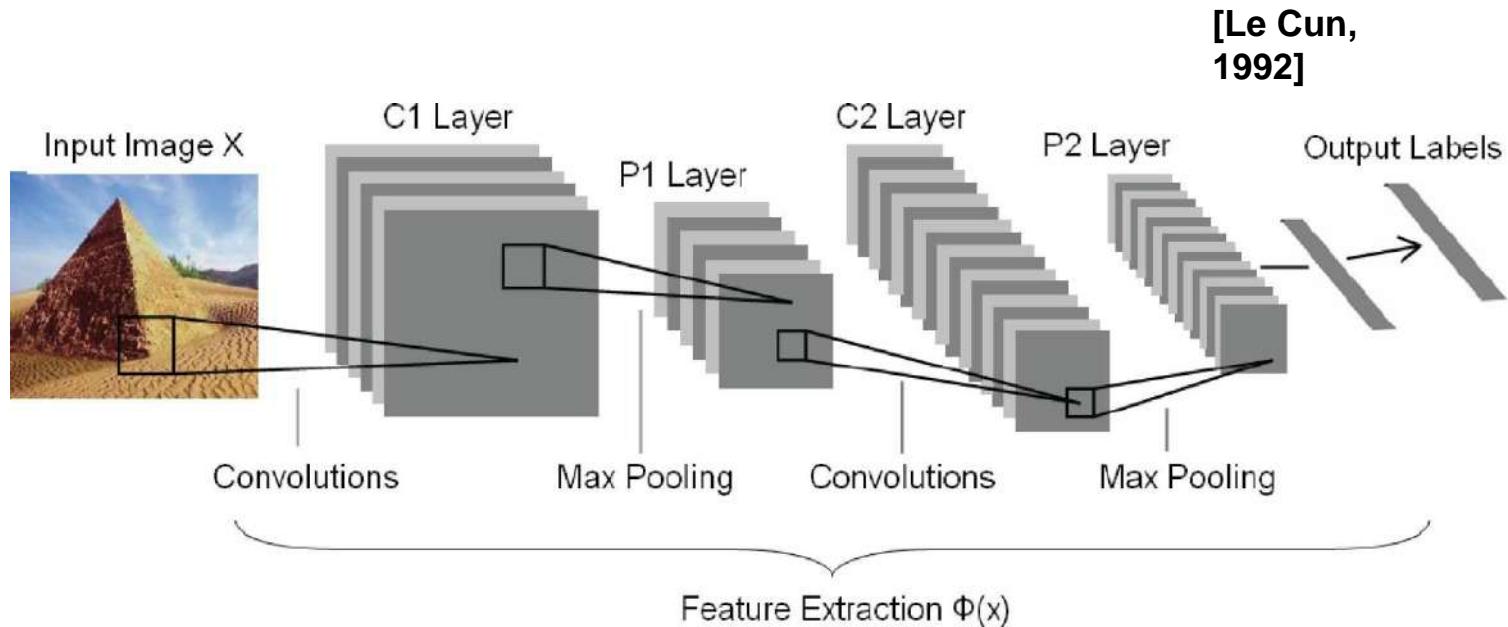
# Training CNN

Input		
b	c	d
e	f	g
h	i	j



- We can thus train a convolution neural network using backpropagation by thinking of it as a feedforward neural network with sparse connections
- A CNN can be implemented as a feedforward neural network
- wherein only a few weights(in color) are active
- the rest of the weights (in gray) are zero

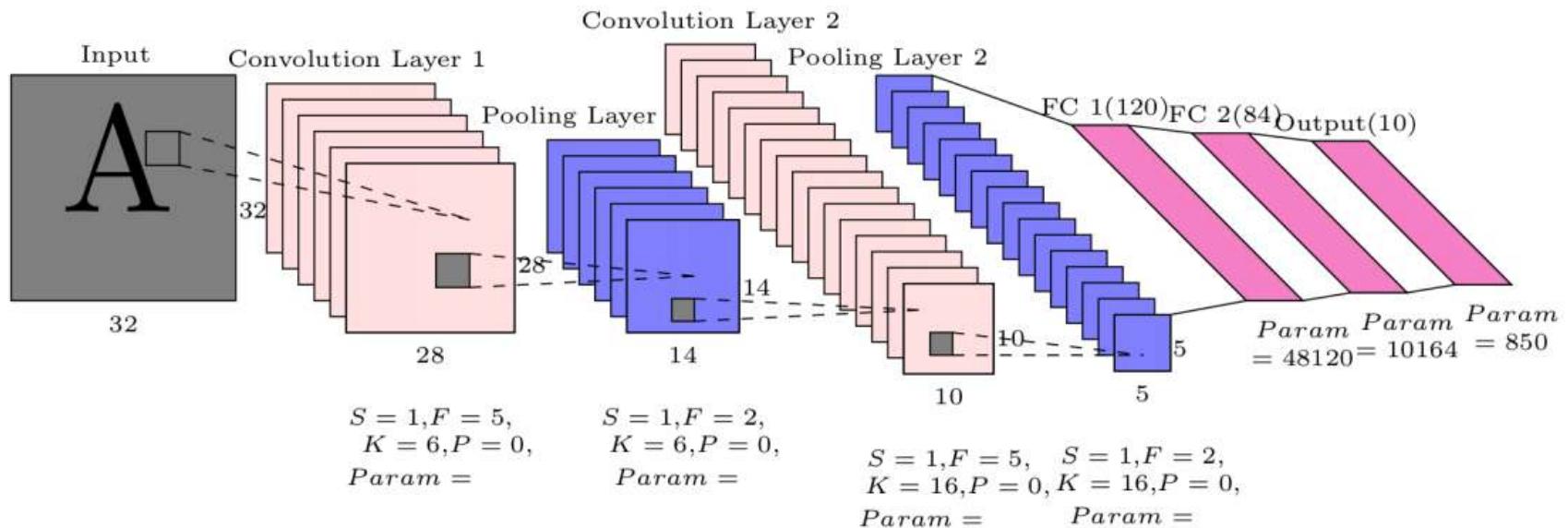
# Convolutional Neural Nets for Image Recognition



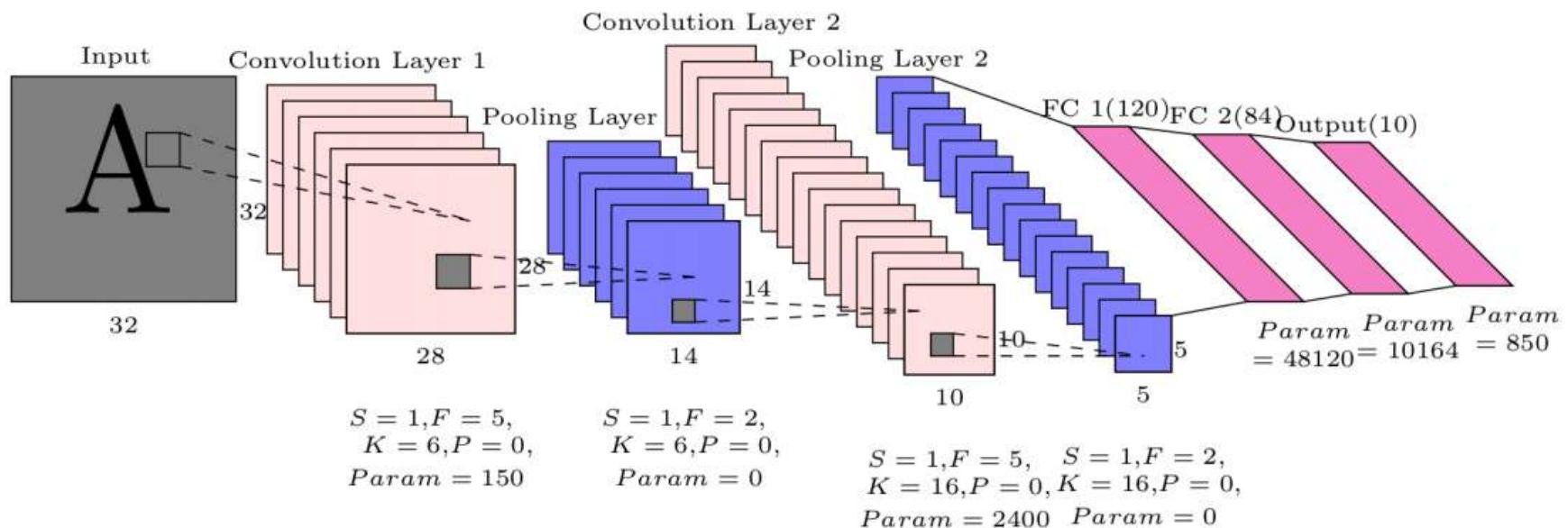
- **specialized architecture: mix different types of units, not completely connected, motivated by primate visual cortex**
- **many shared parameters, stochastic gradient training**
- **very successful! now many specialized architectures for vision, speech, translation, ...**

# Case-Study: LeNet-5

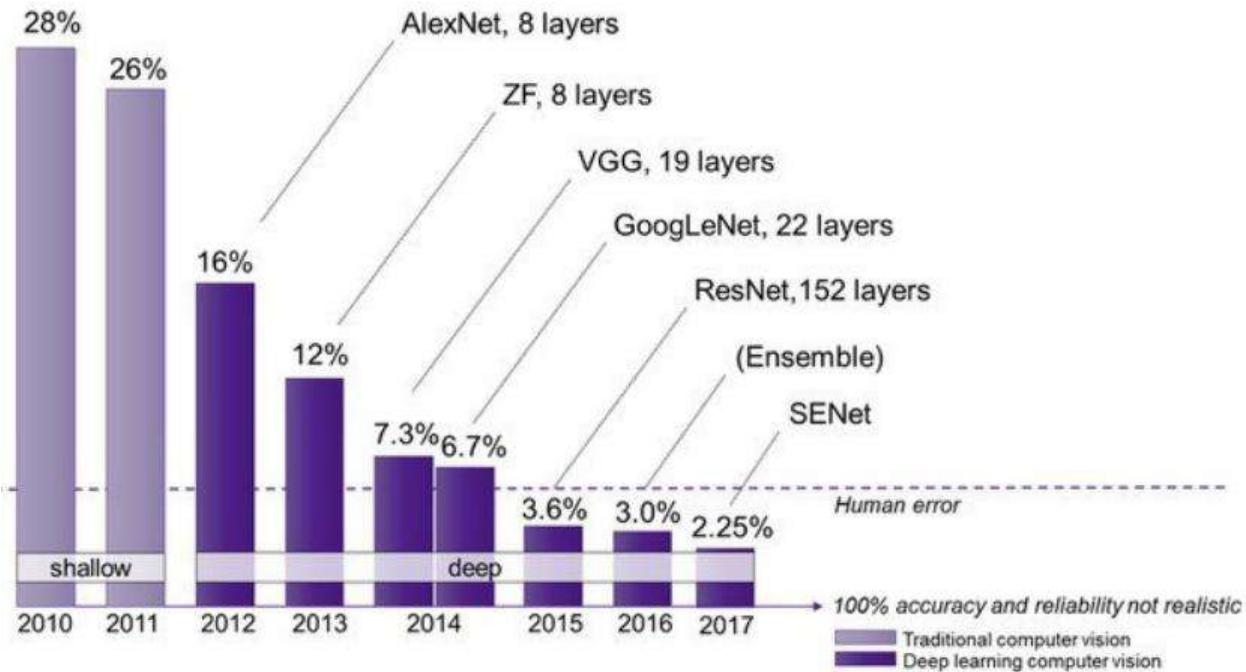
## Handwritten character recognition



# LeNet-5 for handwritten character recognition



# ImageNet ILSVRC



- (2009) 22K category, 14M images
- Challenge 1000 class, 1431167 images
- HoG, LBP, SVM ...



# AlexNet

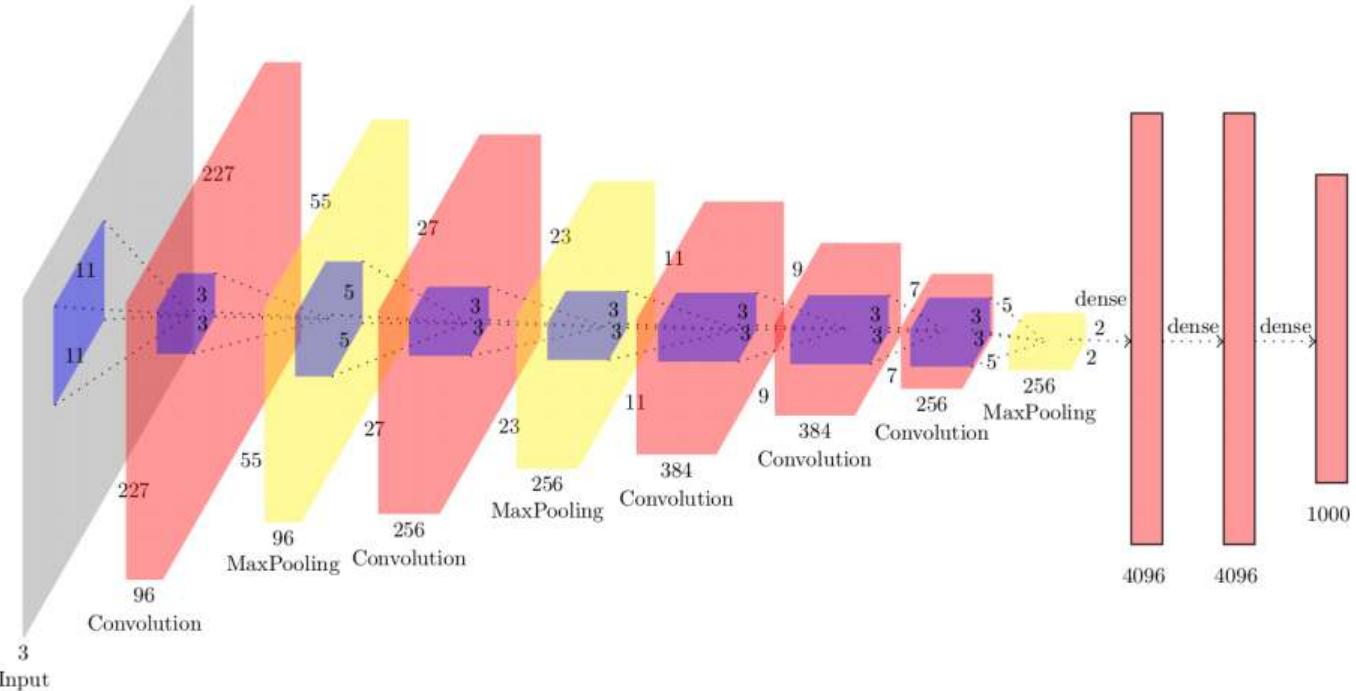


Image size  $227 \times 227 \times 3 \rightarrow [96 F=11, S=4, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow [256 F=5, S=1, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow [384 F=3, S=1, P=0] \rightarrow [384 F=3, S=1, P=0] \rightarrow [256 F=3, S=1, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow \text{FC } 4096 \rightarrow \text{FC } 1000$

# ZFNet

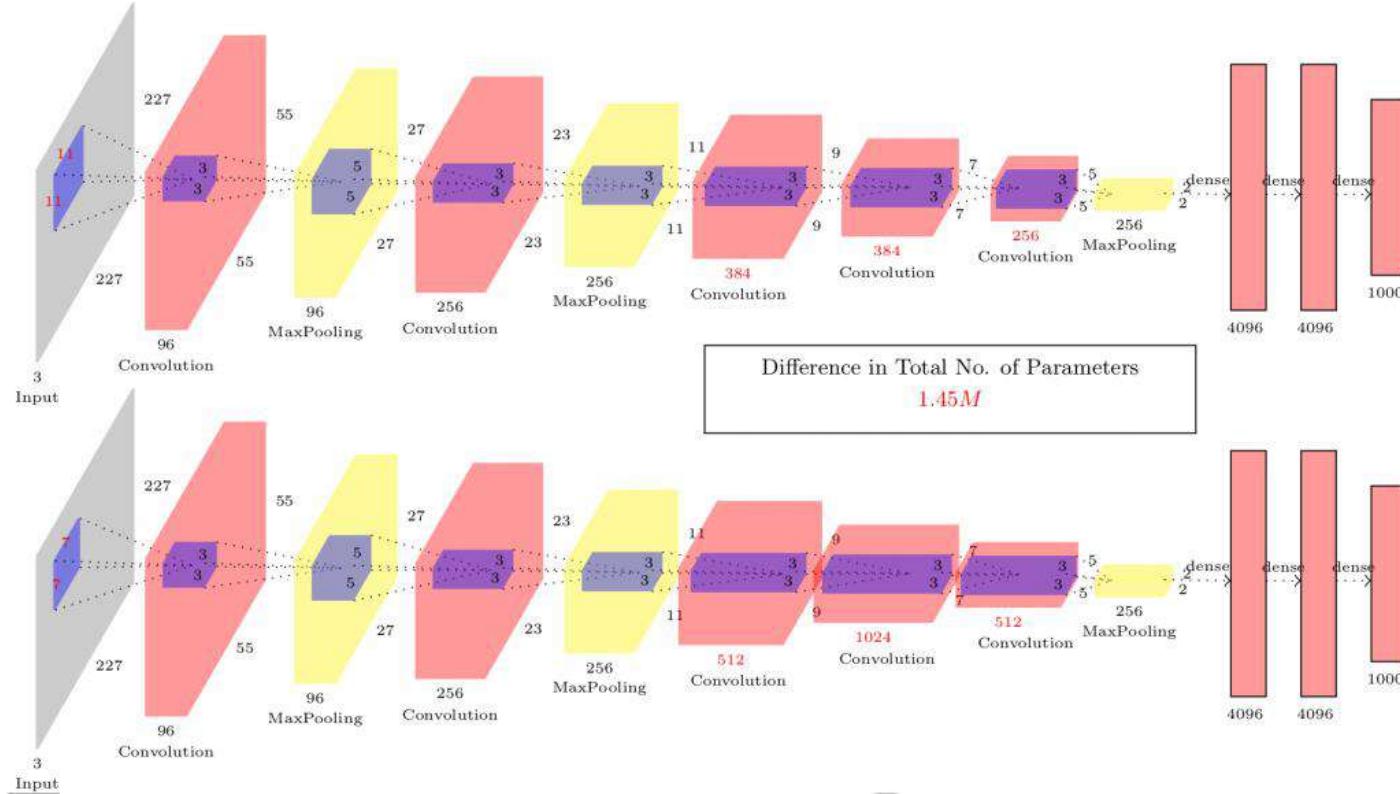
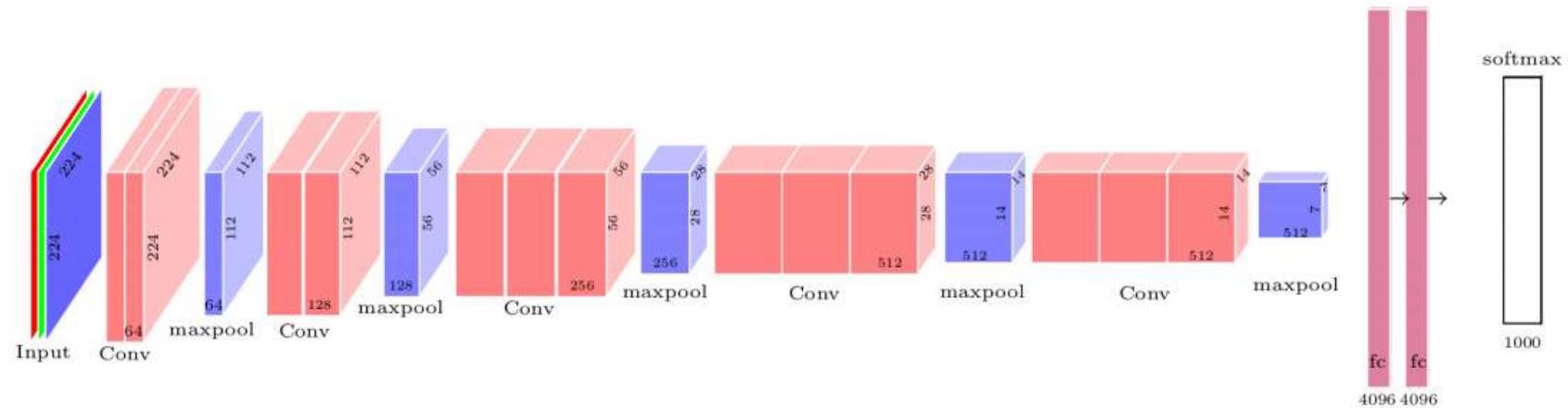


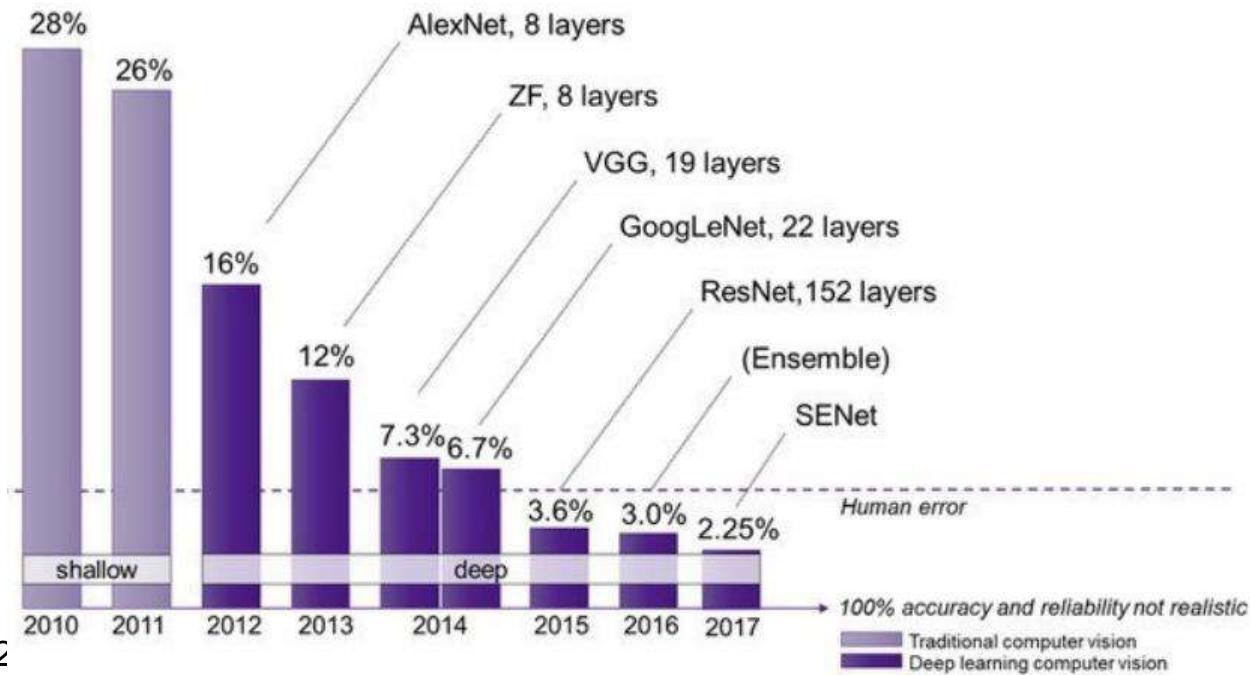
Image size  $227 \times 227 \times 3 \rightarrow [69 F=7, S=4, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow [256 F=5, S=1, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow [512 F=3, S=1, P=0] \rightarrow [1024 F=3, S=1, P=0] \rightarrow [512 F=3, S=1, P=0] \rightarrow [\text{MaxPool } F=3, S=2] \rightarrow \text{FC } 4096 \rightarrow \text{FC } 1000$

# VGG16



- Kernel size is always  $3 \times 3$
- 16M parameters in pre-FC and 122 in FC. First FC layer is huge
- Layers represents abstract representation and can be reused (FC or Conv)

# ImageNet ILSVRC



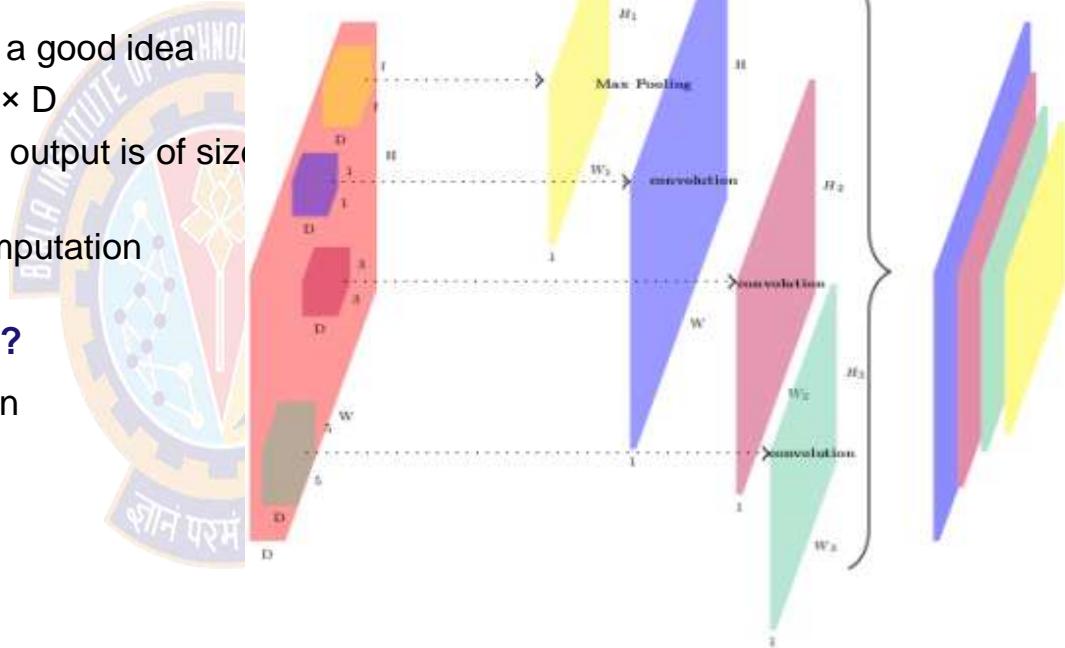
- (2009) 2
- Challenge 1000 class, 1431167 images
- HoG, LBP, SVM ...

# GoogLeNet

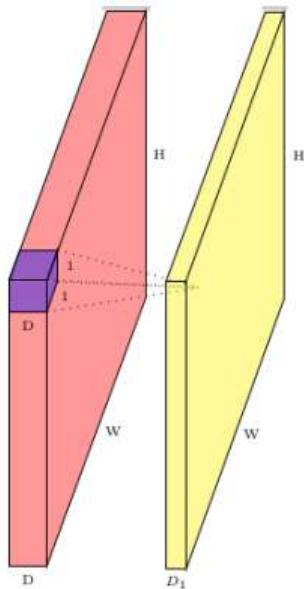
- Recall scale invariance in SIFT
- Multiple filters of different size is a good idea
- With  $W \times H \times D$  input and  $F \times F \times D$
- Filter and  $S = 1$  and no padding, output is of size  $(W - F + 1) \times (H - F + 1)$
- Each value needs  $F \times F \times D$  computation

**Can we reduce this computation a bit?**

- Idea is to have  $1 \times 1$  computation

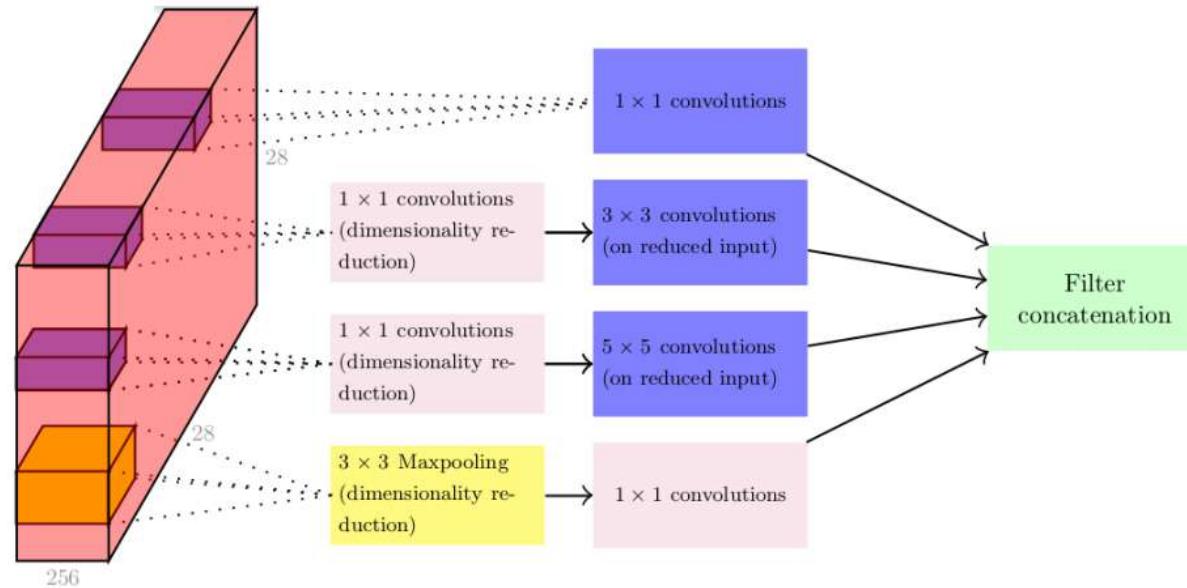


# $1 \times 1$ convolution



- $1 \times 1$  is  $1 \times 1 \times D$
- They produce one output place
- By using D<sub>1</sub> such  $1 \times 1$  convolution output becomes  $F \times F \times D_1$
- We have  $D_1 < D$

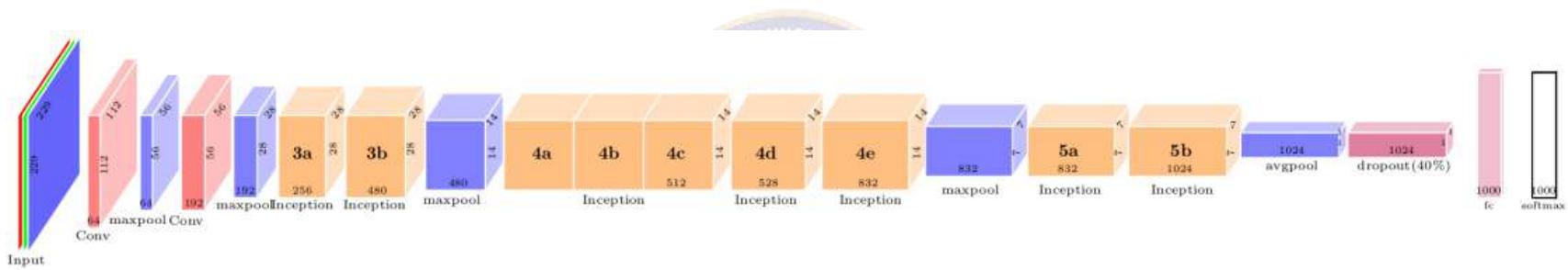
# Inception Block: Multiple convolutions



- $1 \times 1$  convolution
- $1 \times 1$  convolution followed by  $3 \times 3$
- $1 \times 1$  convolution followed by  $5 \times 5$
- $3 \times 3$  maxpool followed by  $1 \times 1$
- Appropriate padding is done to make things of same size

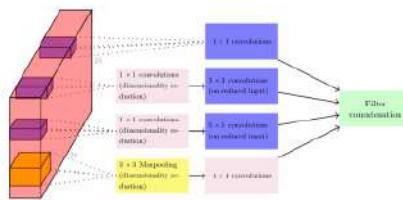


# GoogLeNet

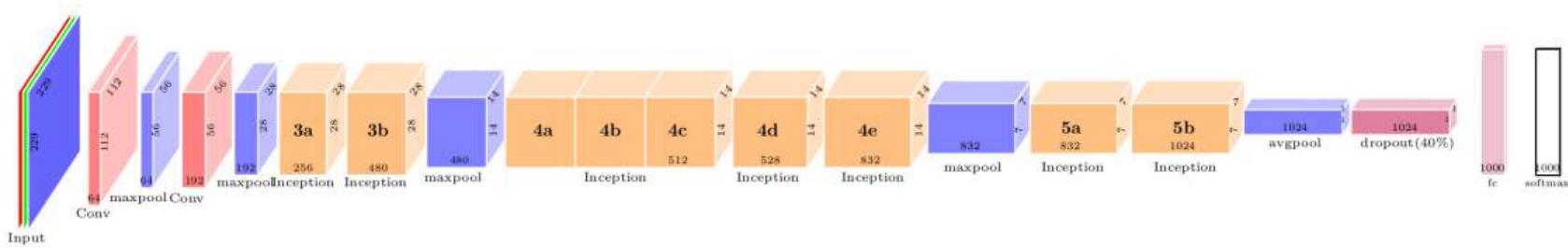


- Input is RGB  $229 \times 229$
- Each inception module have very specific configuration.

(3a)	$192 \times 28 \times 28$	64 96 128 16 32 32
(3b)	$256 \times 28 \times 28$	28 128 192 32 96 61
(4a)	$48 \times 14 \times 14$	192 96 208 16 48 96
(4b)	$512 \times 14 \times 14$	160 112 224 24 64 64
(4c)	$512 \times 14 \times 14$	128 128 256 24 64 64
(4d)	$512 \times 14 \times 14$	112 144 228 32 64 64
(4e)	$528 \times 14 \times 14$	256 160 320 32 128 128
(5a)	$832 \times 7 \times 7$	256 160 320 32 128 128
(5b)	$832 \times 7 \times 7$	384 192 384 48 124 128

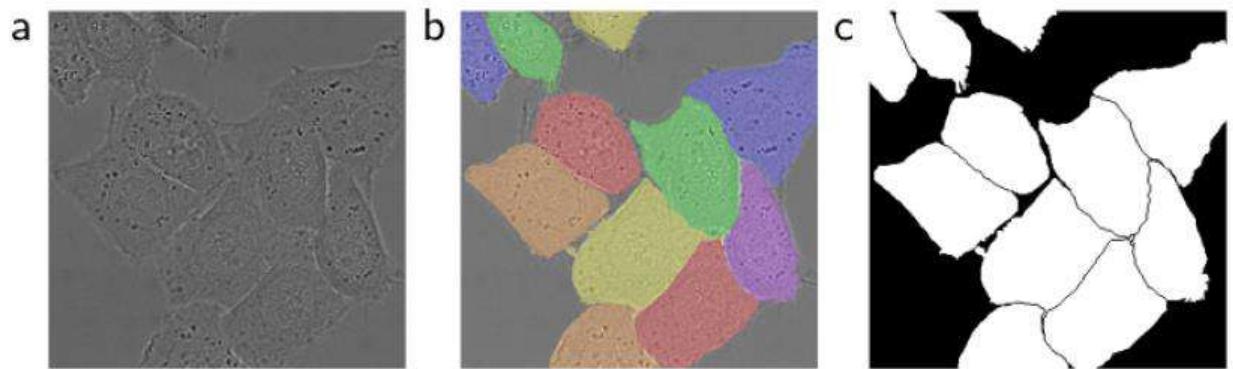


# GoogLeNet



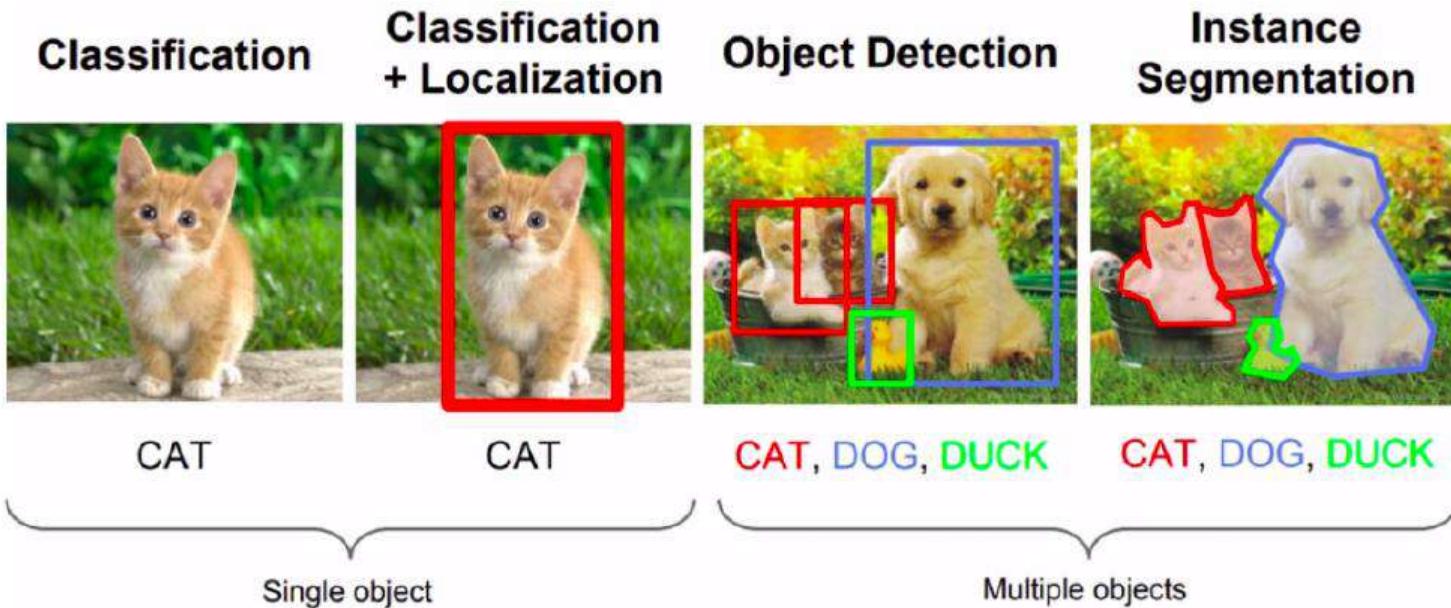
- VGGNET has  $512 \times 7 \times 7$  size at pre-FC this was an issue to connect with 4096
  - GoogLeNet applies a average pool. Gives 49 time reduction. has 1024 values only
  - Dropout and connect to 1000
  - 12 times less connections as compared to AlexNet
  - 2 times more computation as compared to AlexNet
  - Very high accuracy. Error reduced from 16% -to- 6.7%

# Segmentation



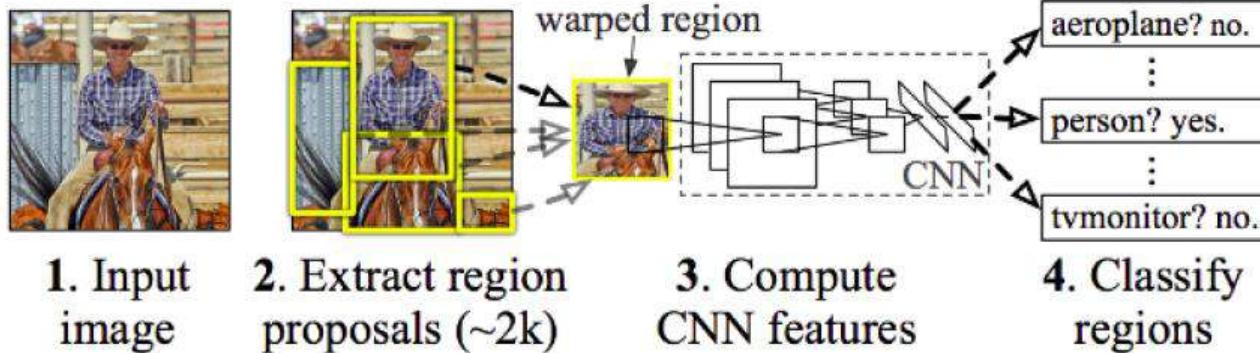
- ISBI challenge for segmentation of neuronal structures in electron microscopic stacks
- Works with very few training images (30/application) and touching boundary. Yield more precise segmentation
- Data augmentation is essential (mainly shift, rotation and elastic deformation)

# Object Detection and Localization

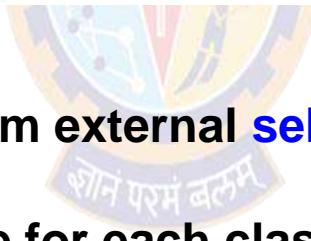


# R-CNN

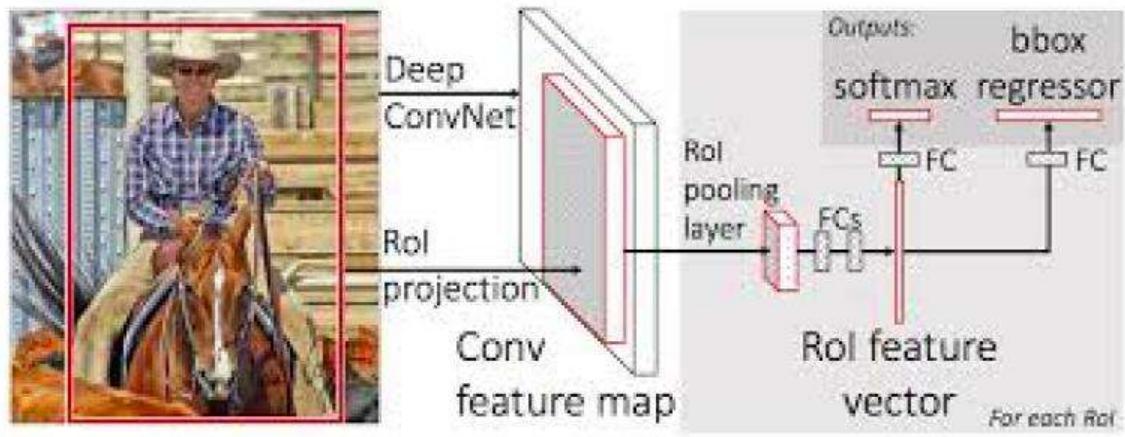
## R-CNN: *Regions with CNN features*



- Region proposals 20K (from external **selective search**)
- Warped image (resizing)
- SVM for classification (one for each class)



# Fast R-CNN

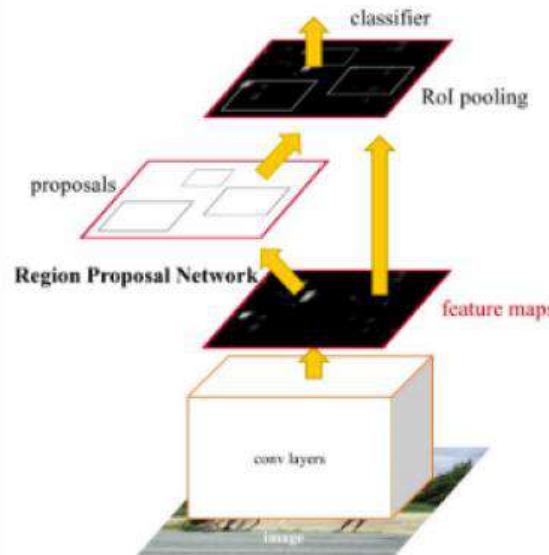


- **ROI pooling**
- **Multi-task loss**



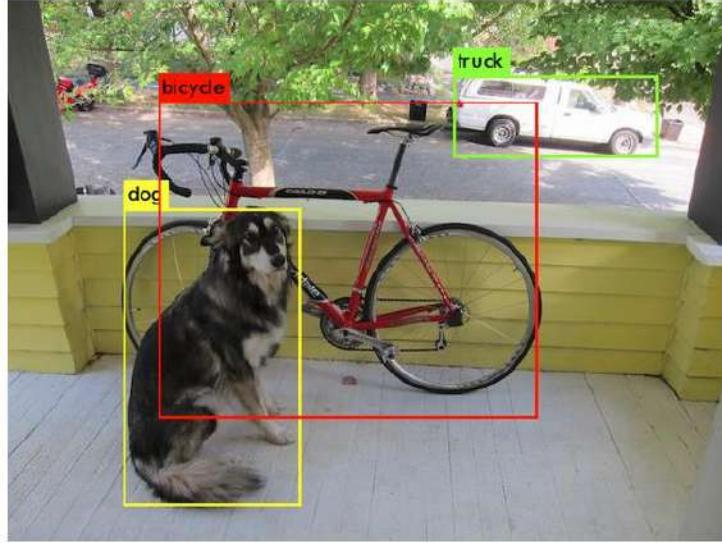
# Faster R-CNN

- Region Proposal Network (RPN)
- Four loss: RPN classification loss, RPN regress loss, Final classification loss, Final box coordinate loss
- 250 time faster than R-CNN. (Fast R-CNN is 25 times fast)



# Yolo

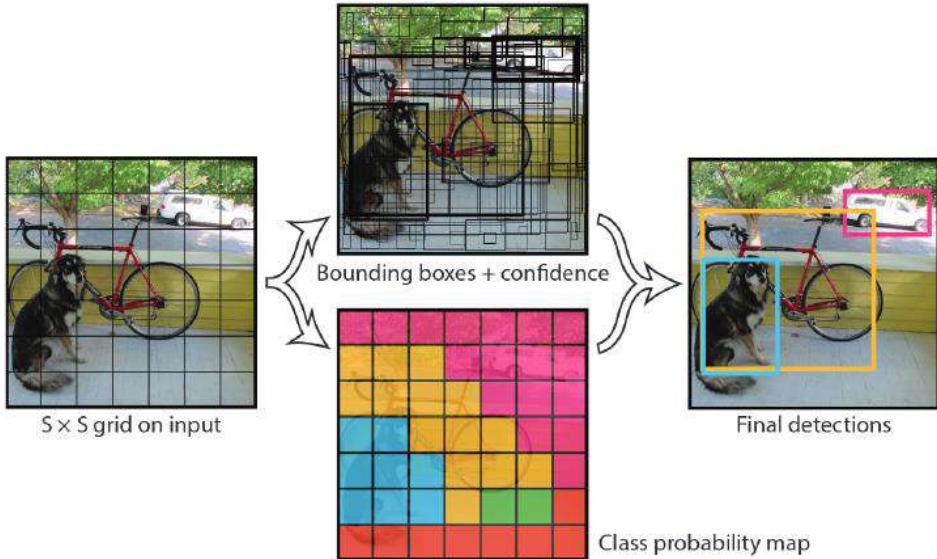
- What is there and where?
- Deformative Part Model, and F-RCNN



- Apply the model to an image at multiple locations and scales.
- High scoring regions are considered detections.

**Yolo:** apply a single neural network to the full image that divides it into regions and predicts bounding boxes and probabilities for each region

# Yolo



- **Conditional probability map**
- See <https://pjreddie.com/darknet/yolo/>

# Yolo

	Pascal 2007 mAP	Speed	
DPM v5	33.7	.07 FPS	14 s/img
R-CNN	66.0	.05 FPS	20 s/img
Fast R-CNN	70.0	.5 FPS	2 s/img
Faster R-CNN	73.2	7 FPS	140 ms/img
YOLO	63.4	45 FPS	22 ms/img



- **It is fast**
- **Speed comes at the price of accuracy. Improved to 69%**
- **Generalizes well**
- **Latest version YOLOv3 2018**

# References

---

Mitesh Khapra

<https://www.youtube.com/watch?v=yw8xwS15Pf4>

Back propagation

[https://www.youtube.com/watch?v=G5b4jRBKNxw&list=PLZbbT5o\\_s2xq7Lwl2y8\\_QtvuXZedL6tQU&index=25](https://www.youtube.com/watch?v=G5b4jRBKNxw&list=PLZbbT5o_s2xq7Lwl2y8_QtvuXZedL6tQU&index=25)



# Thank You!

Next Session: CNN



# C6: ANN and Deep Learning



**BITS Pilani**  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)

# Agenda

1

- SEQUENCE LEARNING

2

- RECURRENT NEURAL NETWORK (RNN)

3

- TYPES OF RNN

4

- LEARNING IN RNN

5

- ISSUES IN RNN

6

- LONG SHORT TERM MEMORY UNIT (LSTM)

7

- GATED RECURRENT UNIT (GRU)

8

# Feedforward NN and CNN

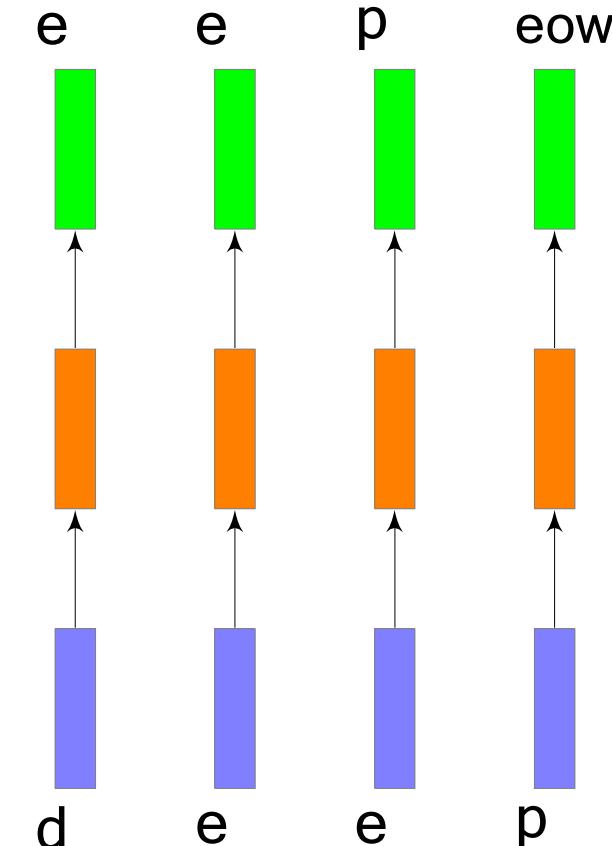
In feedforward and convolutional neural networks

- The size of the input is always fixed.
- Each input to the network is independent of the previous or future inputs.
- The computations, outputs and decisions for two successive inputs / images are completely independent of each other.

# SEQUENCE LEARNING PROBLEMS

- This is not true in many applications.
  - The size of the input is not always fixed.
  - Successive inputs may not be independent of each other.
  - Each network (blue - orange - green structure) is performing the same task
    - input : character output : character.

Example: Auto-completion.



# SEQUENCE LEARNING PROBLEMS



To model a sequence we need

- Process an input or sequence of inputs.
- The inputs may have be dependent.
- We may have to maintain the sequence order.
- Each input corresponds to one time step.
- Keep track of long term dependencies.
- Produce an output or sequence of outputs.
- Supervised Learning.
- Share parameters across the sequences.

# Applications

Speech recognition



"The quick brown fox jumped over the lazy dog."

Music generation

$\emptyset$



Sentiment classification

"There is nothing to like  
in this movie."



DNA sequence analysis

AGCCCCCTGTGAGGAACTAG



AG~~CCCCCTGTGAGGAACTAG~~

Machine translation

Voulez-vous chanter avec  
moi?



Do you want to sing with  
me?

Video activity recognition



Running

Name entity recognition

Yesterday, Harry Potter  
met Hermione Granger.

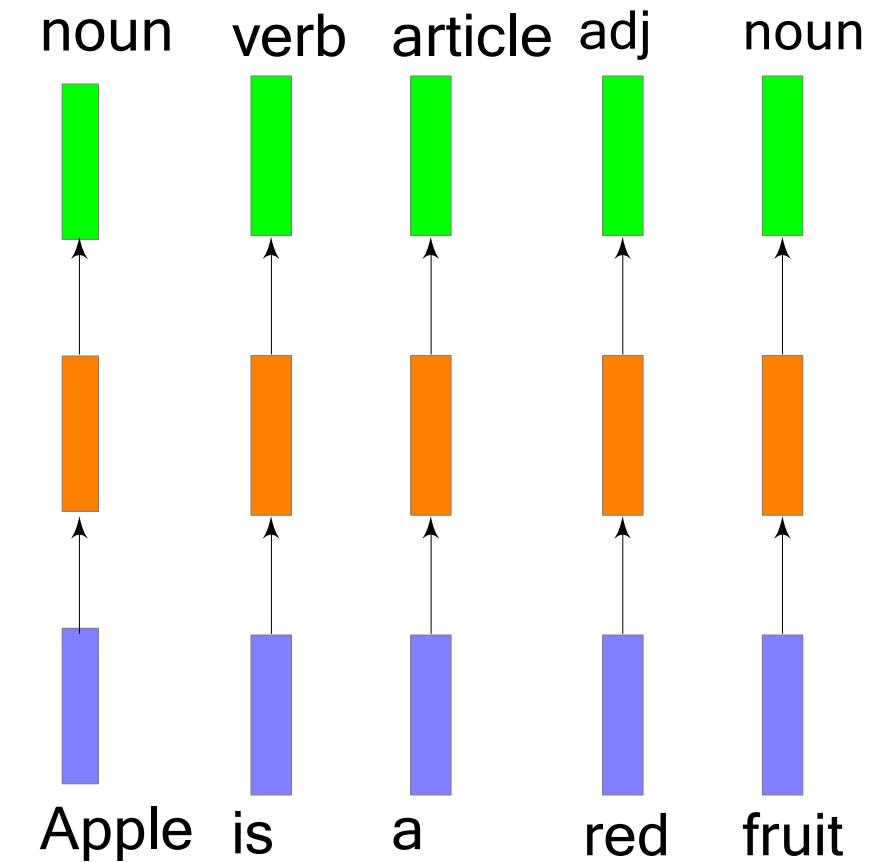


Yesterday, Harry Potter  
met **Hermione Granger**.

# PART OF SPEECH TAGGING

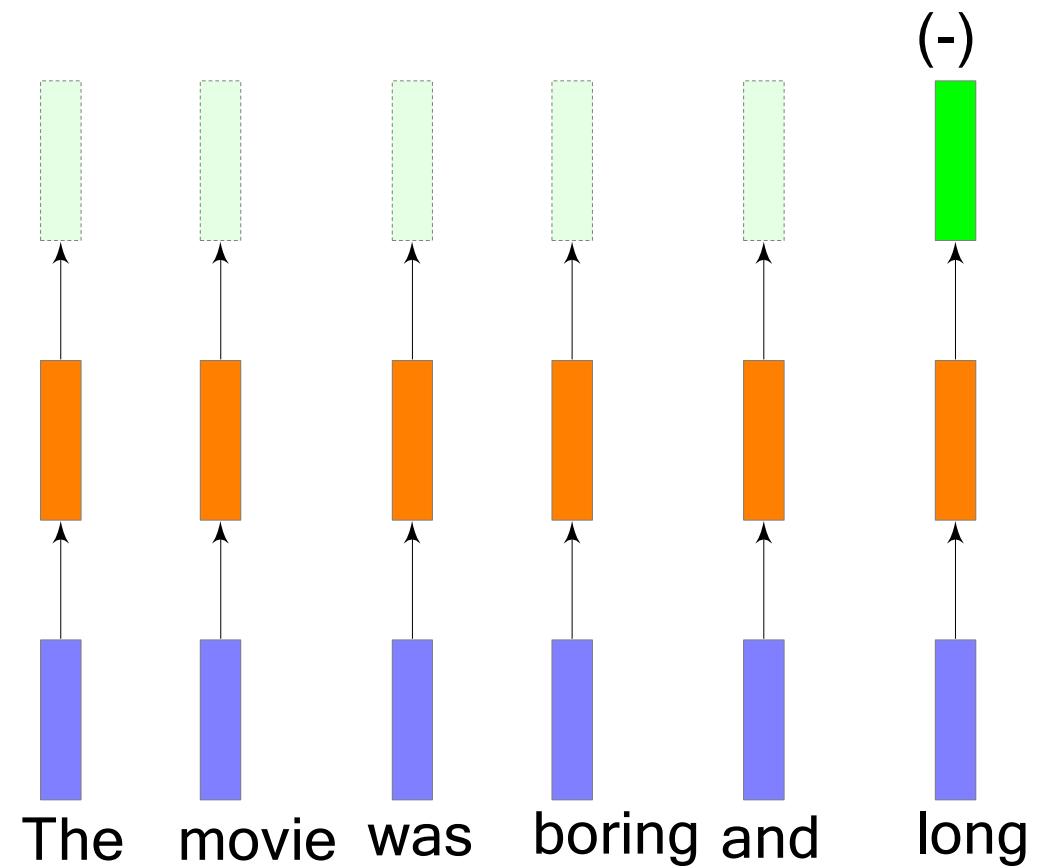


- Task is predicting the part of speech tag (noun, adverb, adjective, verb) of each word in a sentence.
- When we see an adjective we are almost sure, the next word should be a noun.
- The current output depends on the current input as well as the previous input.
- The size of the input is not fixed. Sentences have any number of words.
- An output is produced at end of each time step.
- Each network is performing the same task - input : word, output : tag.

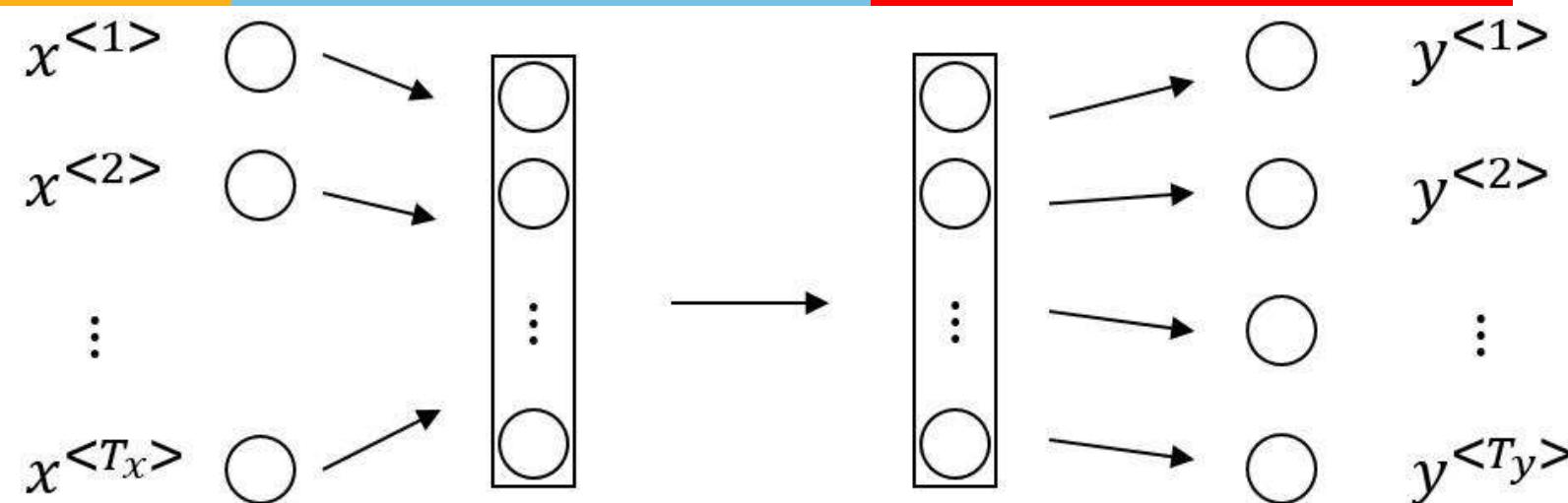


# SENTIMENT ANALYSIS

- Task is predicting the sentiment of a whole sentence.
- Input is the entire sequence of inputs.
- An output is **not** produced at end of each time step.
- Each network is performing the same task - input : word, output : polarity +/−.



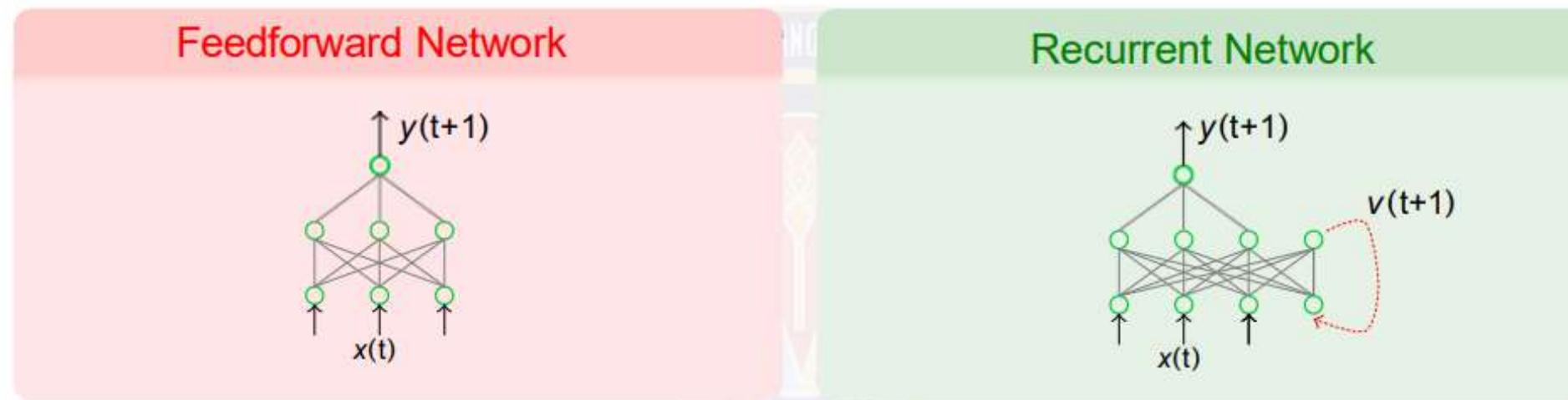
# RECURRENT NEURAL NETWORK (RNN)



Andrew Ng

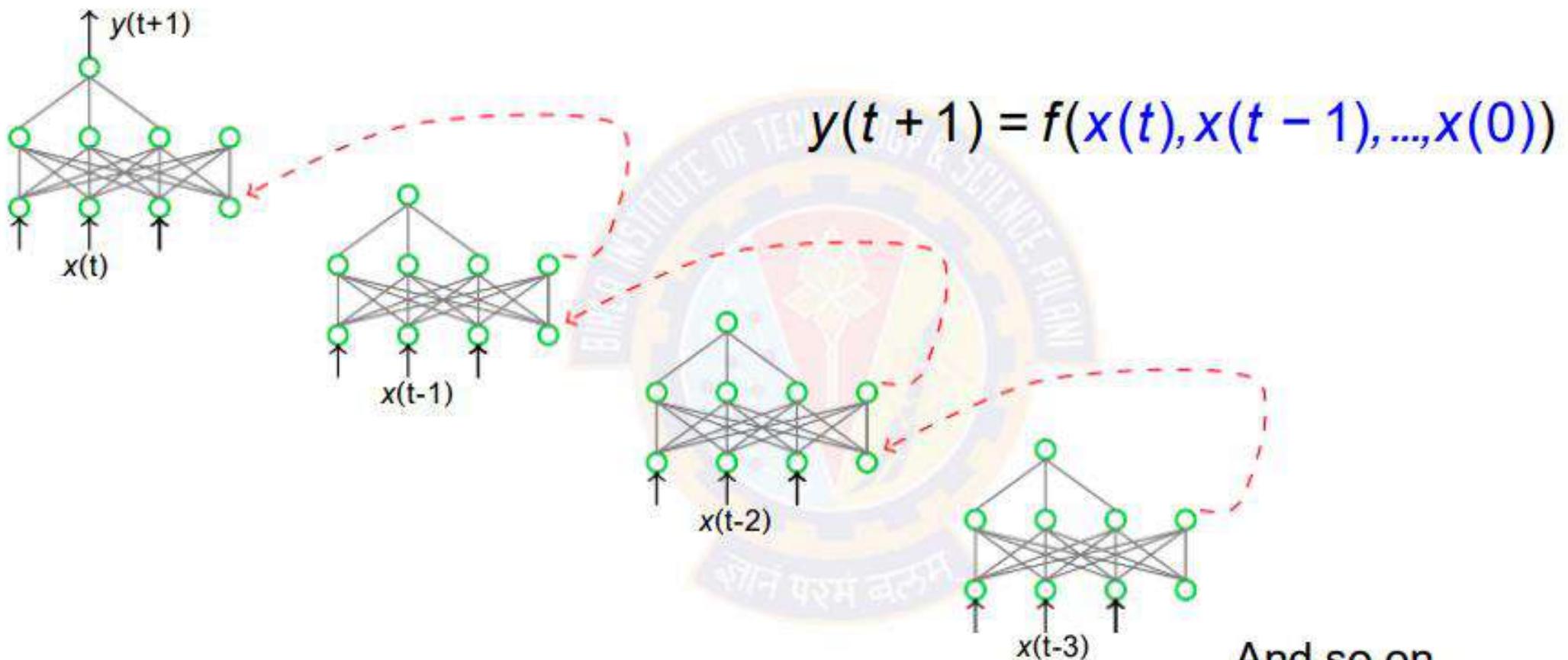
- Accounts for variable number of inputs.
- Accounts for dependencies between inputs.
- Accounts for variable number of outputs.
- Ensures that the same function executed at each time step.
- The features learned across the inputs at different time step has to be shared.

Feedforward network cannot capture the dependence of  $y(t + 1)$  on earlier values of  $x$  such as  $x(t - 1)$



- RNN uses **states** (called self-state) of the **network units** available at **time  $t$**  as an input to the other units at **time  $t + 1$**
- RNN is suitable for temporal data (like time series)
- Training may involve unfolding and averaging.

When input at the time  $t$  is provided, what is output at time  $(t + 1)$  ?



- The function learned at each time step.

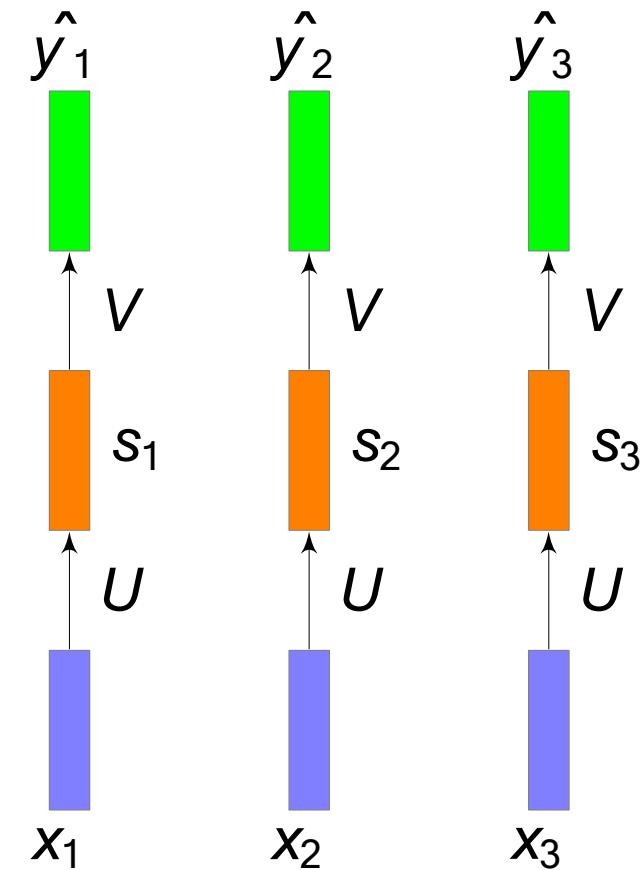
$t = \text{time step}$

$x_t = \text{input at time step } t$

$s_t = \sigma(Ux_t + b)$

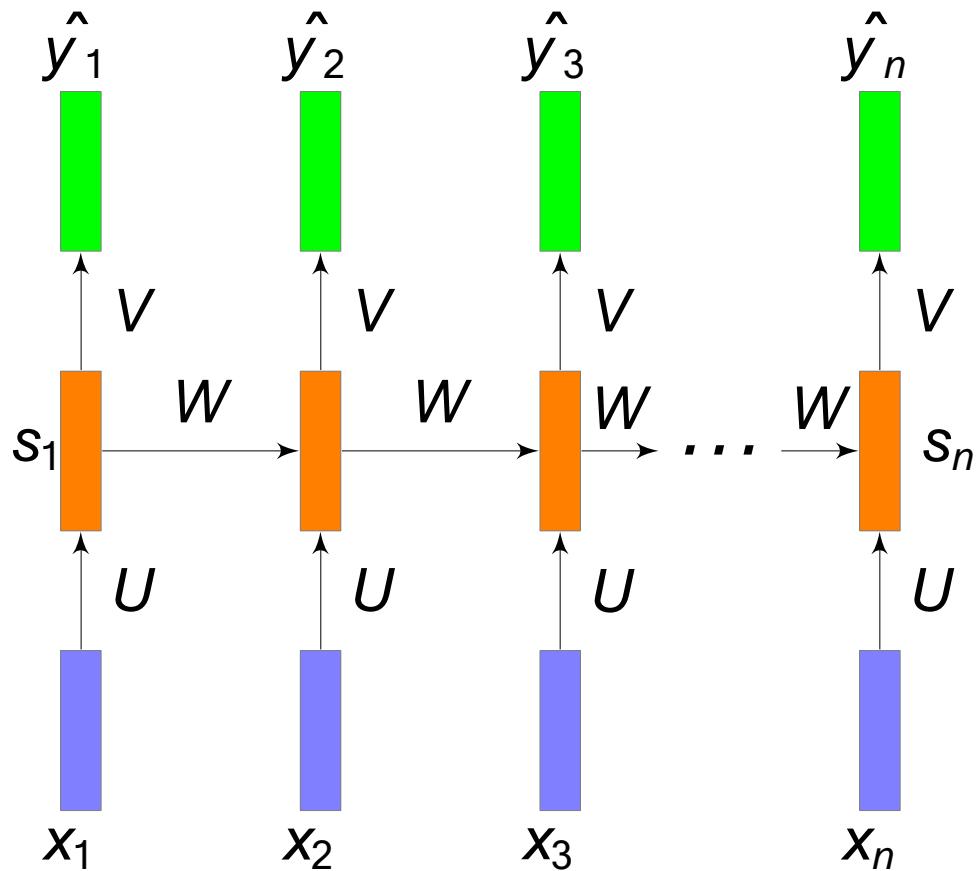
$y_t = g(Vs_t + c)$

- Since the same function has to be executed at each time step we should share the same network i.e., same parameters at each time step.



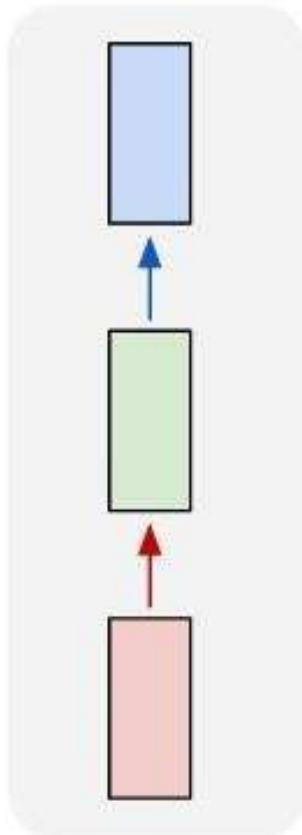
# RNN

- The parameter sharing ensures that
  - ) the network becomes invariant to the length of the input.
  - ) the number of time steps doesn't matter.
- Create multiple copies of the network and execute them at each timestep.
  - ) i.e. create a loop effect.
  - ) i.e. add recurrent connection in the network.

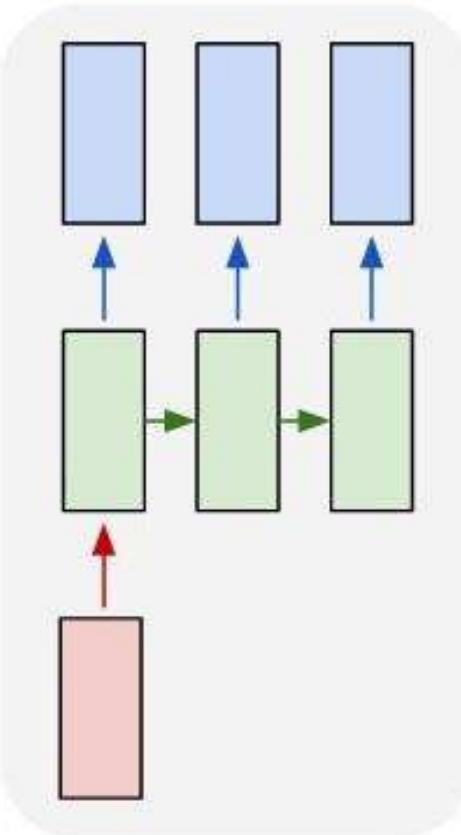


# Types of RNN

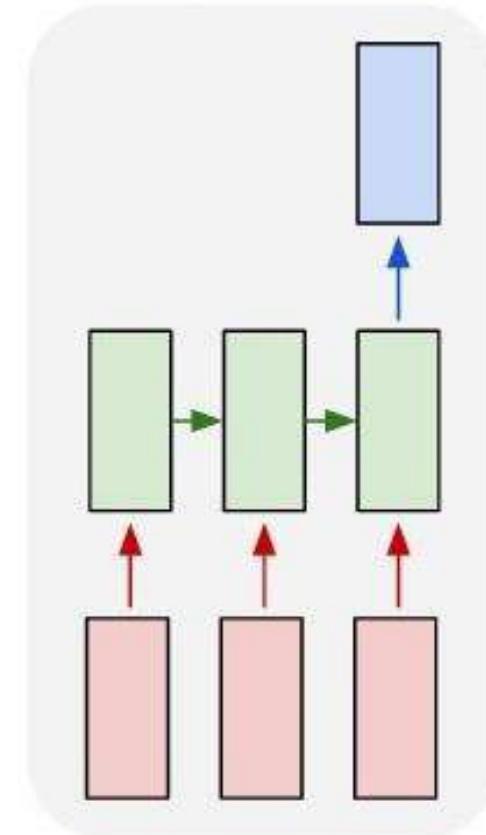
one to one



one to many

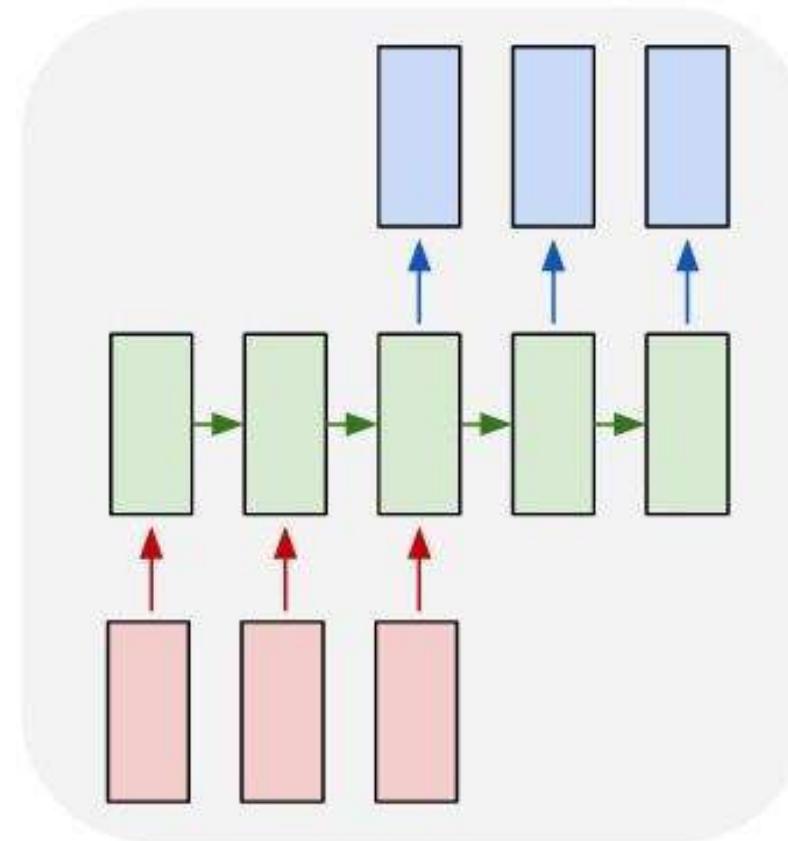


many to one

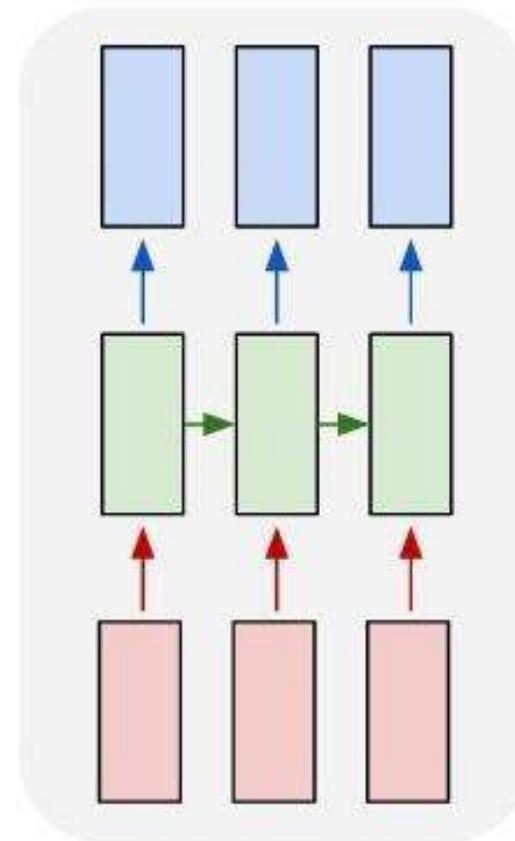


# Types of RNN

many to many



many to many

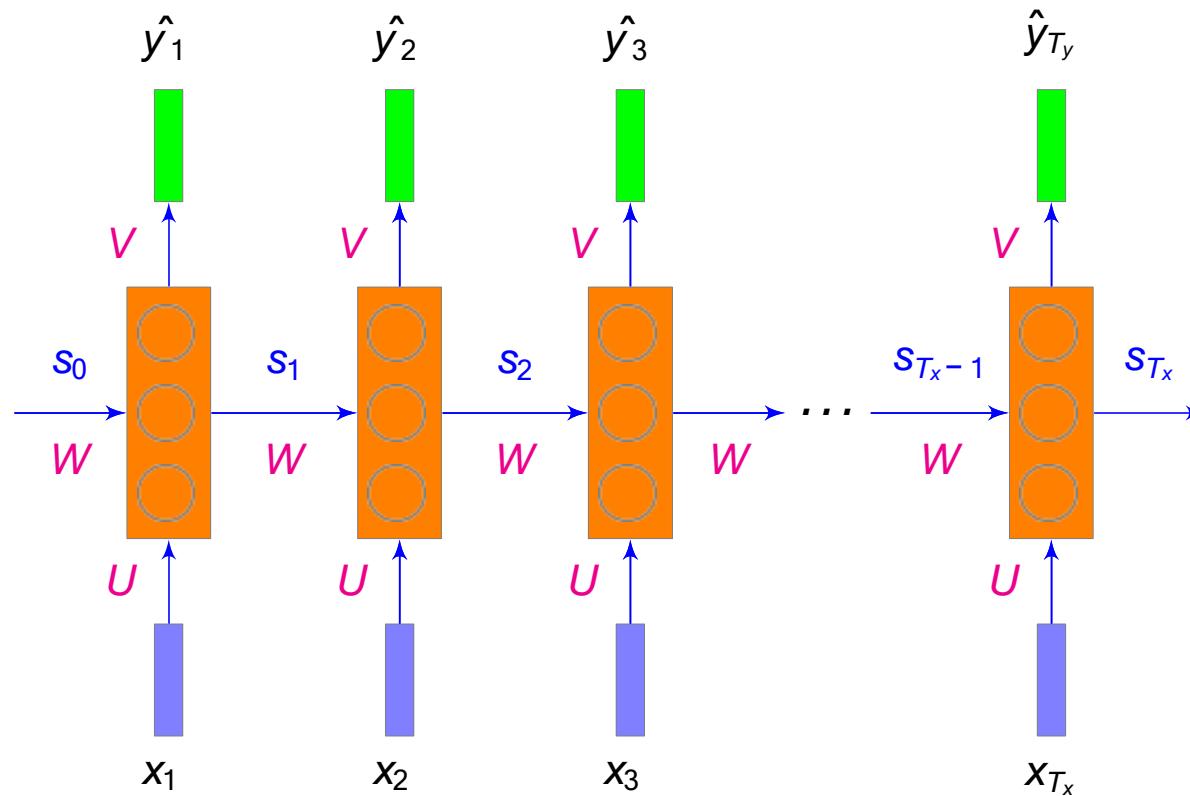


# TYPES OF RNN AND APPLICATIONS

---

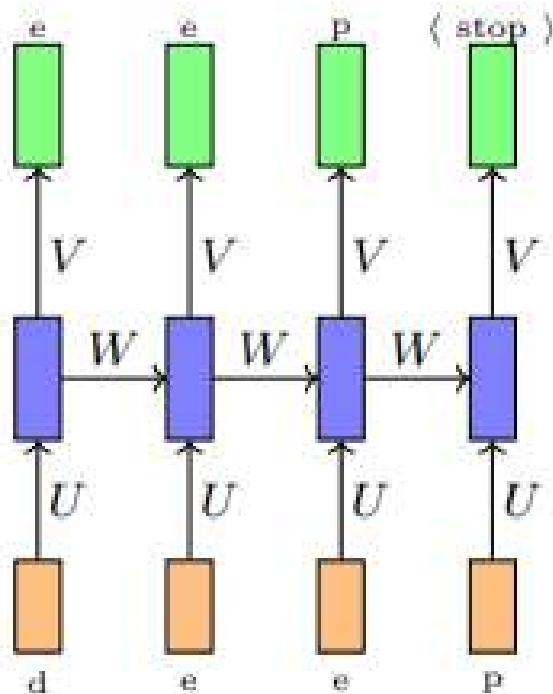
- One to one - Generic neural network, Image classification
- One to many - Music generation, Image Captioning
- Many to one - Movie review or Sentiment Analysis
- Many to many - Machine translation
- Synced Many to many - Video classification

# FORWARD PROPAGATION IN RNN



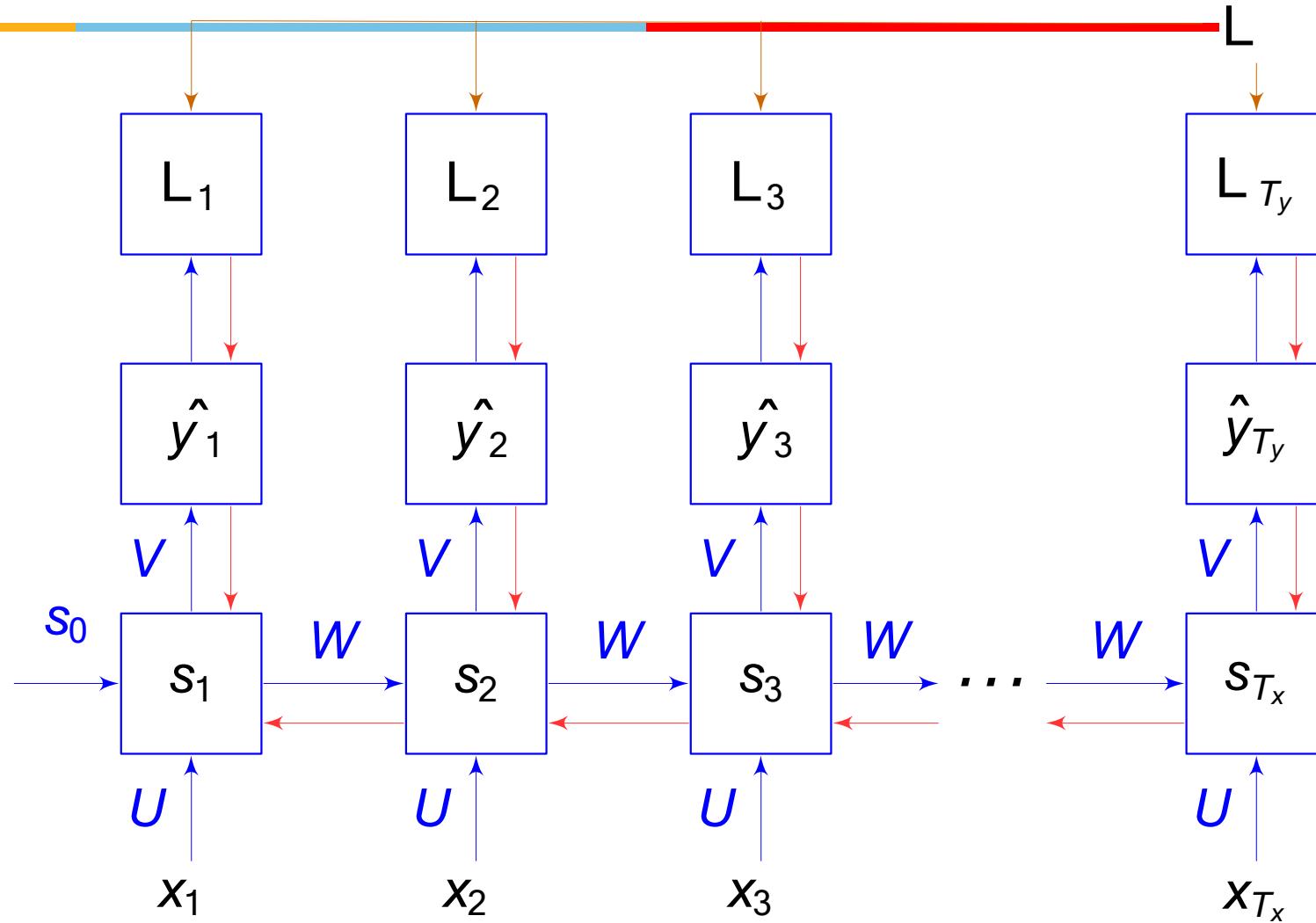
- $s_t$  is the **state** of the network at time step  $t$ .
- $s_0 = 0$
- $s_t = \sigma(Ux_t + Ws_{t-1} + b)$
- $\hat{y}_t = g(Vs_t + c)$
- or
- $\hat{y}_t = f(x_t, s_{t-1}, W, U, V, b, c)$
- The parameters  $W, U, V, b, c$  are shared across time steps.

# FORWARD PROPAGATION IN RNN



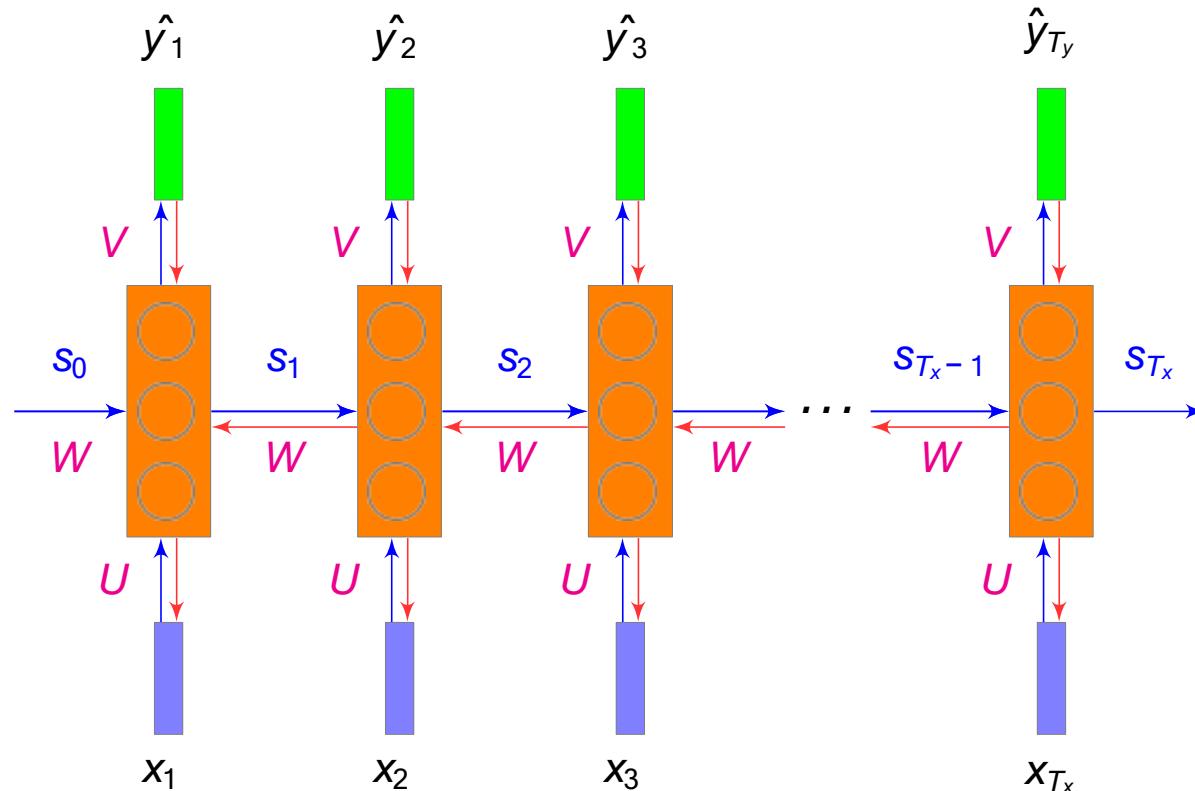
- Suppose we consider our task of auto-completion (predicting the next character)
- For simplicity we assume that there are only 4 characters in our vocabulary (d,e,p, <stop>)
- At each timestep we want to predict one of these 4 characters
- What is a suitable output function for this task ? (**softmax**)
- What is a suitable loss function for this task ? (**cross entropy**)

# BACK PROPAGATION IN RNN



Back-propagation through time.

# BACK PROPAGATION IN RNN



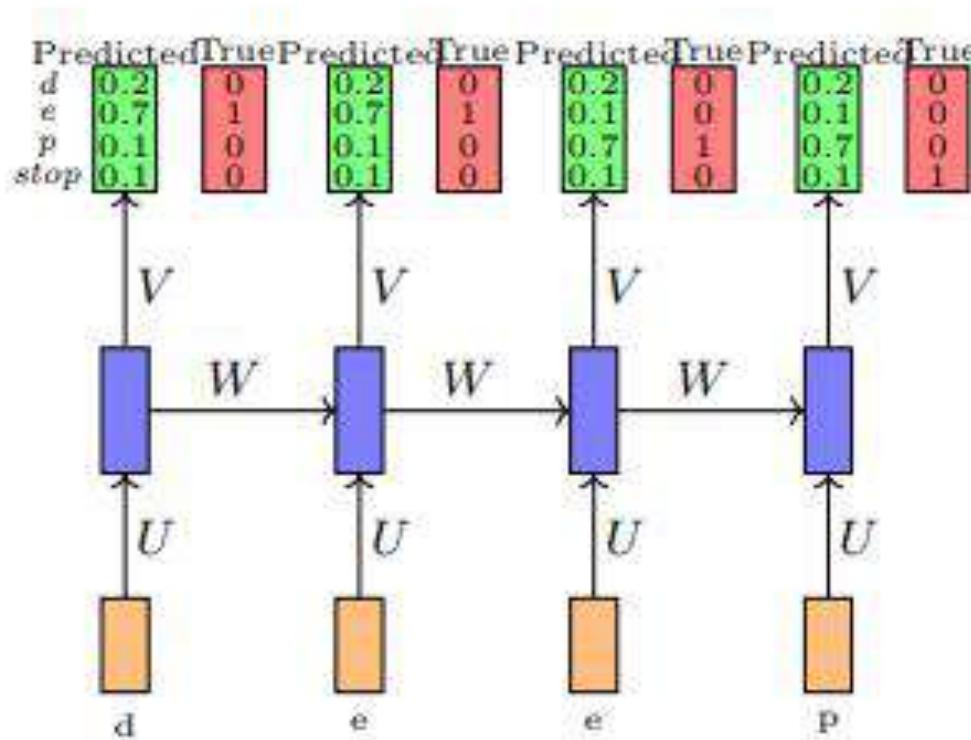
Loss function

$$L_t(\hat{y}_t, y_t) = \sum_{t=1}^{T_y} P(\hat{y}_t | \hat{y}_{t-1}, \dots, \hat{y}_1)$$

Overall Loss

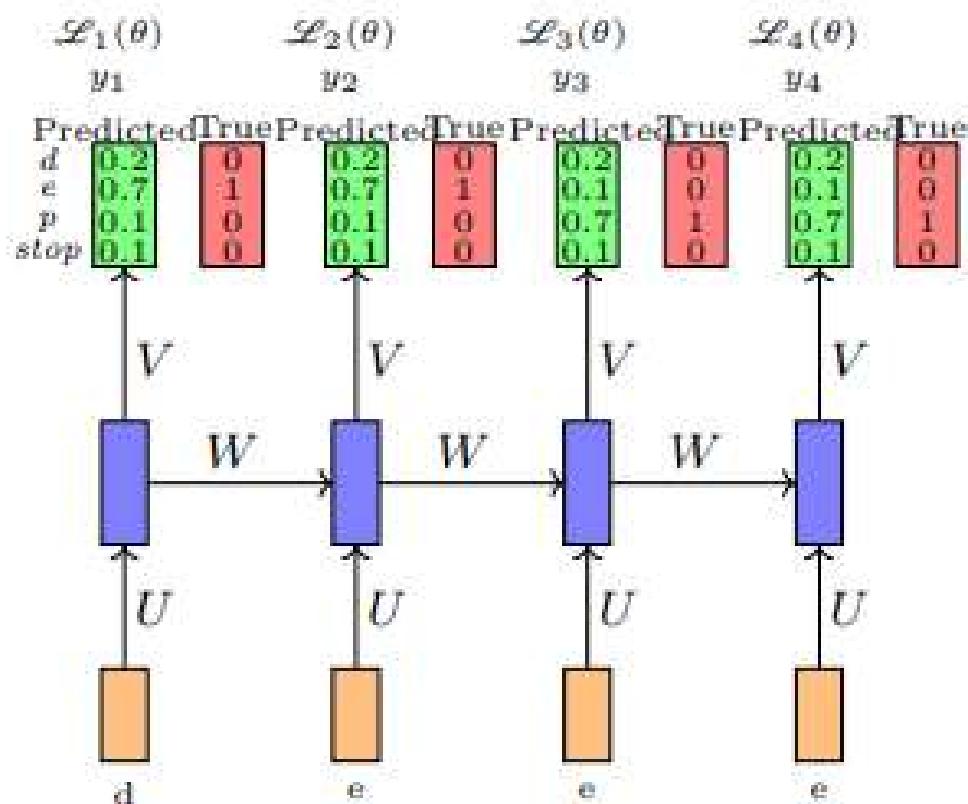
$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L_t(\hat{y}_t, y_t)$$

# Back PROPAGATION IN RNN



- Suppose we initialize  $U, V, W$  randomly and the network predicts the probabilities as shown
- And the true probabilities are as shown
- We need to answer two questions
- What is the total loss made by the model ?
- How do we backpropagate this loss and update the parameters ( $\theta = \{U, V, W, b, c\}$ ) of the network ?

# Back PROPAGATION IN RNN



- The total loss is simply the sum of the loss over all time-steps

$$\mathcal{L}(\theta) = \sum_{t=1}^T \mathcal{L}_t(\theta)$$

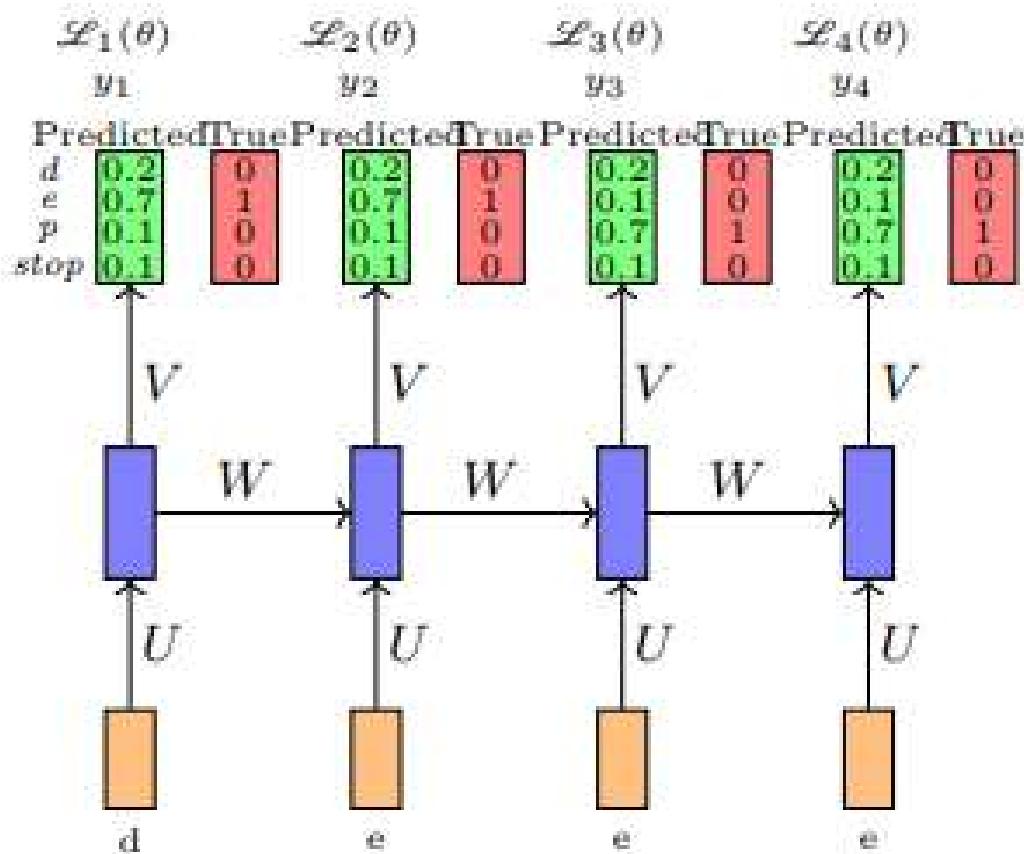
$$\mathcal{L}_t(\theta) = -\log(y_{tc})$$

$y_{tc}$  = predicted probability of true character at time-step *t*

*T* = number of timesteps

- For backpropagation we need to compute the gradients w.r.t. *W*, *U*, *V*, *b*, *c*

# Back PROPAGATION IN RNN

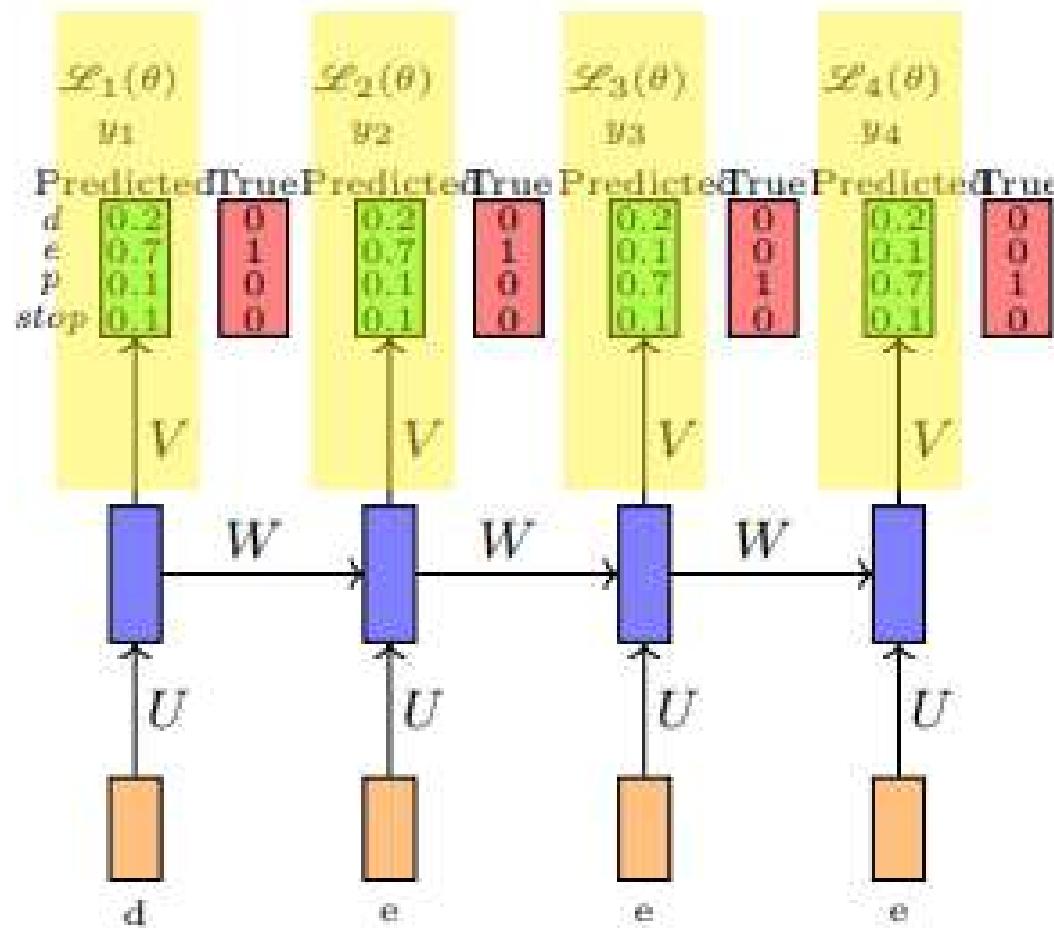


- Let us consider  $\frac{\partial \mathcal{L}(\theta)}{\partial V}$  ( $V$  is a matrix so ideally we should write  $\nabla_V \mathcal{L}(\theta)$ )

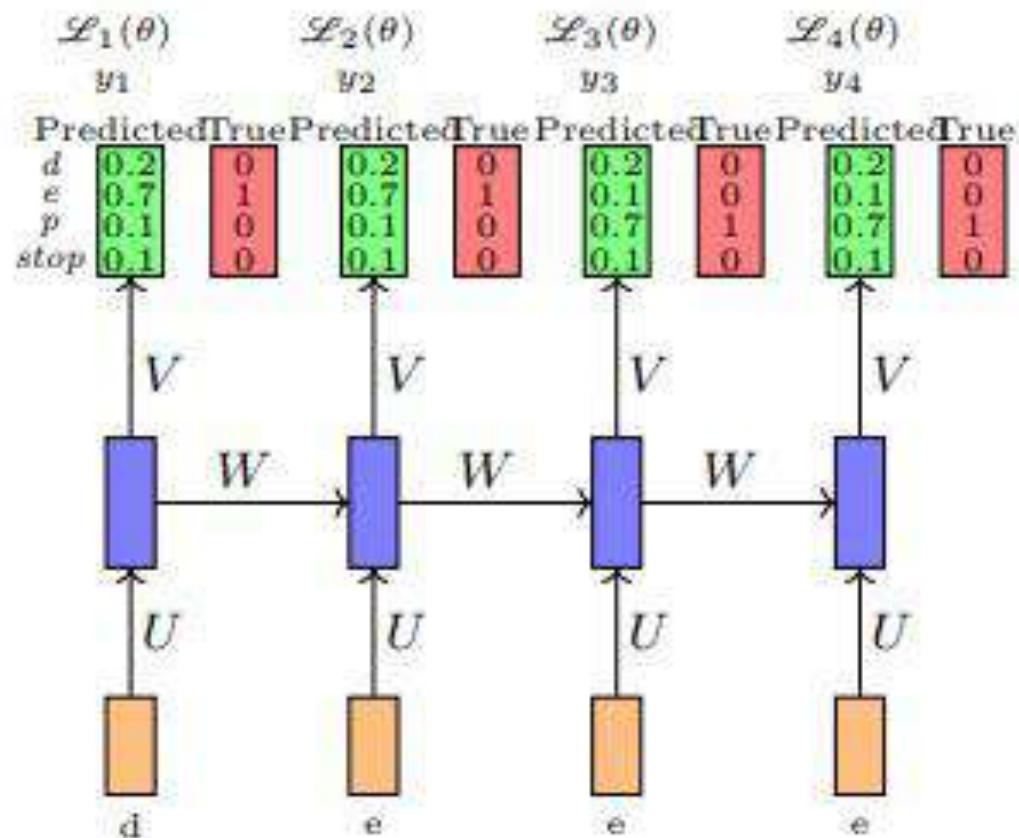
$$\frac{\partial \mathcal{L}(\theta)}{\partial V} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t(\theta)}{\partial V}$$

- Each term is the summation is simply the derivative of the loss w.r.t. the weights in the output layer

# Back PROPAGATION IN RNN



# Back PROPAGATION IN RNN

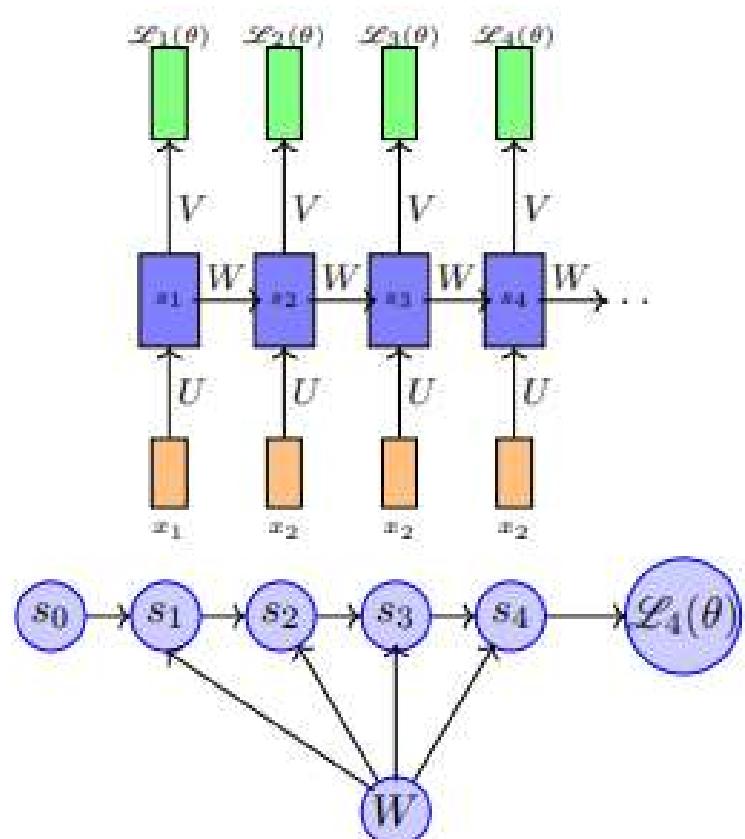


- Let us consider the derivative  $\frac{\partial \mathcal{L}(\theta)}{\partial W}$

$$\frac{\partial \mathcal{L}(\theta)}{\partial W} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t(\theta)}{\partial W}$$

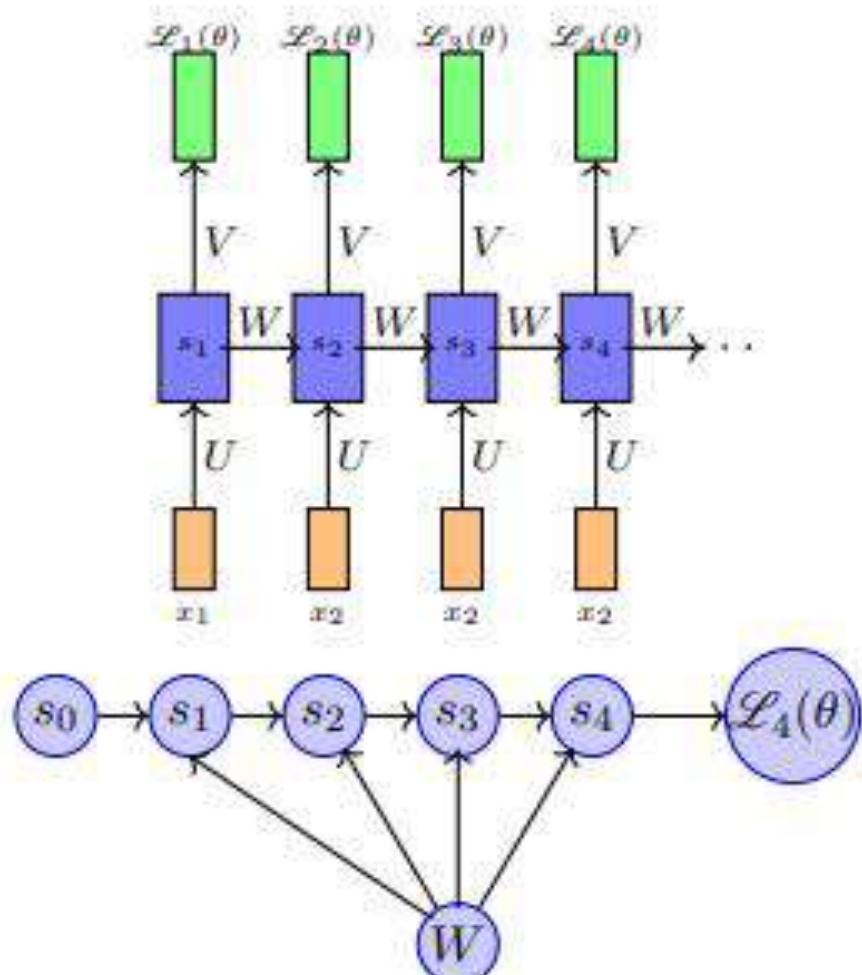
- By the chain rule of derivatives we know that  $\frac{\partial \mathcal{L}_t(\theta)}{\partial W}$  is obtained by summing gradients along all the paths from  $\mathcal{L}_t(\theta)$  to  $W$
- What are the paths connecting  $\mathcal{L}_t(\theta)$  to  $W$  ?
- Let us see this by considering  $\mathcal{L}_4(\theta)$

# Back PROPAGATION IN RNN



- $\mathcal{L}_4(\theta)$  depends on  $s_4$
- $s_4$  in turn depends on  $s_3$  and  $W$
- $s_3$  in turn depends on  $s_2$  and  $W$
- $s_2$  in turn depends on  $s_1$  and  $W$
- $s_1$  in turn depends on  $s_0$  and  $W$   
where  $s_0$  is a constant starting state.

# Back PROPAGATION IN RNN

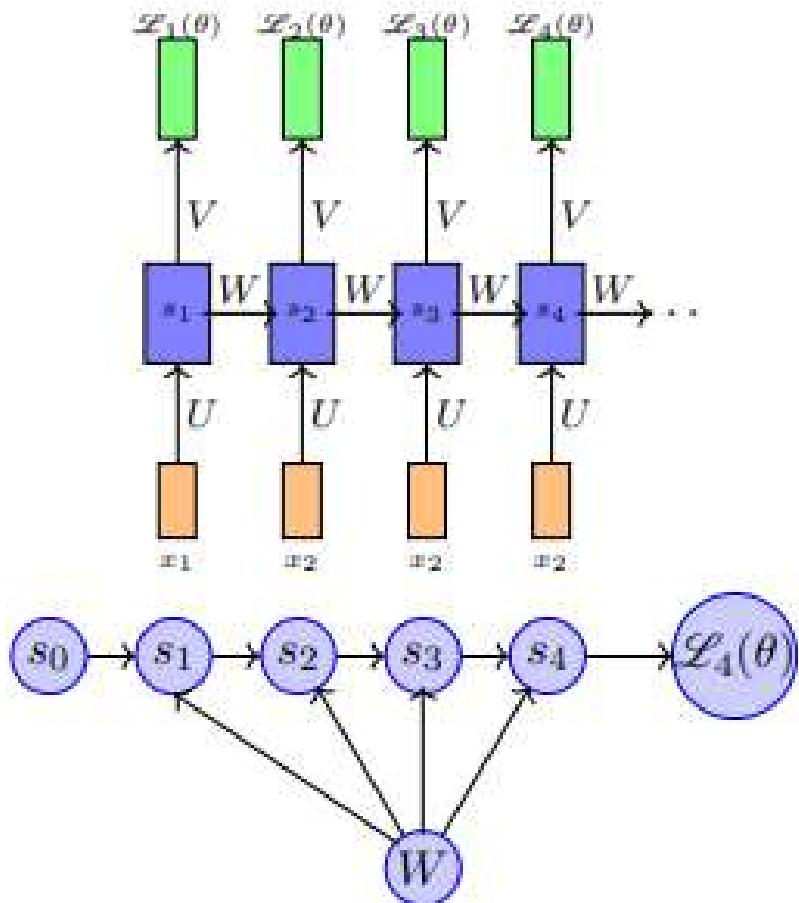


- What we have here is an ordered network
- In an ordered network each state variable is computed one at a time in a specified order (first  $s_1$ , then  $s_2$  and so on)
- Now we have

$$\frac{\partial \mathcal{L}_4(\theta)}{\partial W} = \frac{\partial \mathcal{L}_4(\theta)}{\partial s_4} \frac{\partial s_4}{\partial W}$$

- We have already seen how to compute  $\frac{\partial \mathcal{L}_4(\theta)}{\partial s_4}$  when we studied backprop
- But how do we compute  $\frac{\partial s_4}{\partial W}$

# Back PROPAGATION IN RNN



- Recall that

$$s_4 = \sigma(Ws_3 + b)$$

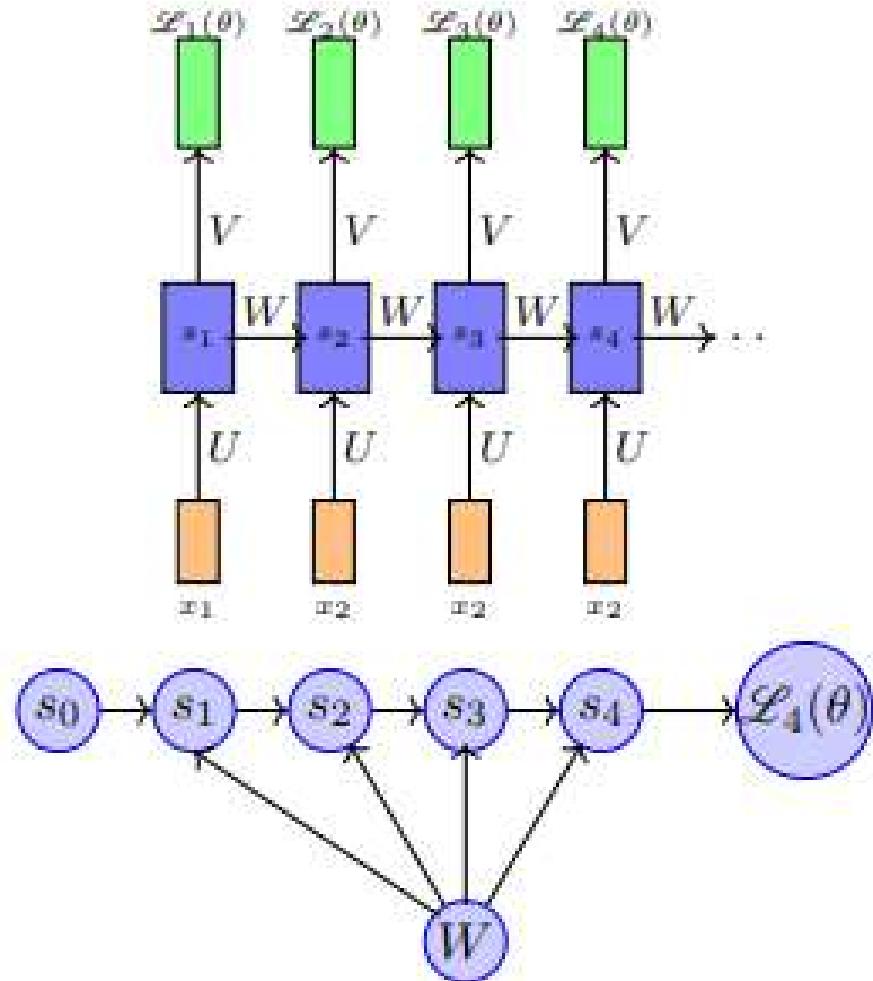
- In such an ordered network, we can't compute  $\frac{\partial s_4}{\partial W}$  by simply treating  $s_3$  as a constant (because it also depends on  $W$ )
- In such networks the total derivative  $\frac{\partial s_4}{\partial W}$  has two parts
- Explicit :**  $\frac{\partial^+ s_4}{\partial W}$ , treating all other inputs as constant
- Implicit :** Summing over all indirect paths from  $s_4$  to  $W$

# Back PROPAGATION IN RNN

innovate

achieve

lead



- Finally we have

$$\frac{\partial \mathcal{L}_4(\theta)}{\partial W} = \frac{\partial \mathcal{L}_4(\theta)}{\partial s_4} \frac{\partial s_4}{\partial W}$$

$$\frac{\partial s_4}{\partial W} = \sum_{k=1}^4 \frac{\partial s_4}{\partial s_k} \frac{\partial^+ s_k}{\partial W}$$

$$\therefore \frac{\partial \mathcal{L}_t(\theta)}{\partial W} = \frac{\partial \mathcal{L}_t(\theta)}{\partial s_t} \sum_{k=1}^t \frac{\partial s_t}{\partial s_k} \frac{\partial^+ s_k}{\partial W}$$

- This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps

# Back PROPAGATION IN RNN



$$\begin{aligned}\frac{\partial s_4}{\partial W} &= \underbrace{\frac{\partial^+ s_4}{\partial W}}_{\text{explicit}} + \underbrace{\frac{\partial s_4}{\partial s_3} \frac{\partial s_3}{\partial W}}_{\text{implicit}} \\ &= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3} \left[ \underbrace{\frac{\partial^+ s_3}{\partial W}}_{\text{explicit}} + \underbrace{\frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial W}}_{\text{implicit}} \right] \\ &= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial s_3}{\partial s_2} \left[ \frac{\partial^+ s_2}{\partial W} + \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial W} \right] \\ &= \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial^+ s_2}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \left[ \frac{\partial^+ s_1}{\partial W} \right]\end{aligned}$$

For simplicity we will short-circuit some of the paths

$$\frac{\partial s_4}{\partial W} = \frac{\partial s_4}{\partial s_4} \frac{\partial^+ s_4}{\partial W} + \frac{\partial s_4}{\partial s_3} \frac{\partial^+ s_3}{\partial W} + \frac{\partial s_4}{\partial s_2} \frac{\partial^+ s_2}{\partial W} + \frac{\partial s_4}{\partial s_1} \frac{\partial^+ s_1}{\partial W} = \sum_{k=1}^4 \frac{\partial s_4}{\partial s_k} \frac{\partial^+ s_k}{\partial W}$$

## Backpropogation is used to train a RNN

- Total loss is the summation of  $J_t$  at every time step  $J = \sum_t J_t$
- To train a parameter we need to find its gradient with respect to loss and shift the parameter in its opposite direction

$$W = W - \alpha \frac{\partial J}{\partial W} = W - \alpha \frac{\partial}{\partial W} \sum_t J_t = W - \alpha \sum_t \frac{\partial J_t}{\partial W}$$

- Consider a single summation term

$$\frac{\partial J_i}{\partial W} = \frac{\partial J_i}{\partial s_i} \times \frac{\partial s_i}{\partial W}$$

- Term  $\frac{\partial s_i}{\partial W}$  is complicated. It depends on  $s_{i-1}, s_{i-2}, \dots, s_1$

# Exploding and Vanishing Gradients

Consider  $\frac{\partial s_i}{\partial s_k}$  that is  $\frac{\partial s_i}{\partial s_{i-1}} \frac{\partial s_{i-1}}{\partial s_{i-2}} \frac{\partial s_{i-2}}{\partial s_{i-3}} \dots \frac{\partial s_{k+2}}{\partial s_{k+1}} \frac{\partial s_{k+1}}{\partial s_k} = \prod_{j=i}^{k+1} \frac{\partial s_j}{\partial s_{j-1}}$

Focus on a single term  $\frac{\partial s_j}{\partial s_{j-1}}$

it can be shown that it is upper-bounded by some constant (which depends on  $W$ )  
let's call that value  $c$

Since  $\frac{\partial s_i}{\partial s_k}$  is a multiplication of  $i - k$  such terms (and we expect  $i - k$  to be large to address long term dependency) therefore the value

$$\frac{\partial s_i}{\partial s_k} \rightarrow \begin{cases} \text{Vanish} & \text{if } c < 1 \\ \text{Explode} & \text{if } c > 1 \end{cases}$$

(0.90 → 0.81 → 0.73 → 0.66 → 0.59 → 0.53 → 0.48 → 0.43 → 0.39)

(1.50 → 2.25 → 3.38 → 5.06 → 7.59 → 11.39 → 17.09 → 25.63)

# Exploding and Vanishing Gradients

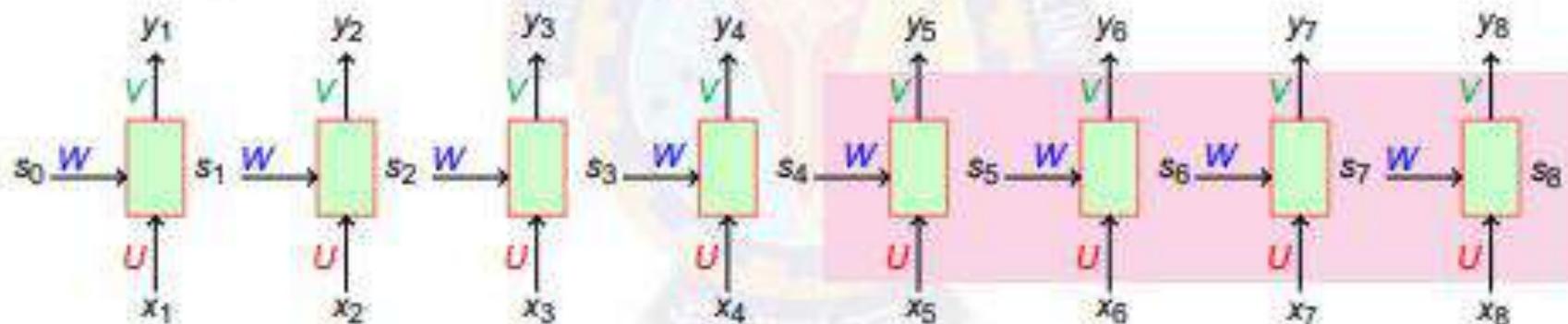


- **Clipping**

Try normalizing value of  $c$  keeping it some range  $T_l \leq c \leq Th$

- **Truncated Backpropagation**

At any step look for only for last  $k$  timestamps



- **LSTM**

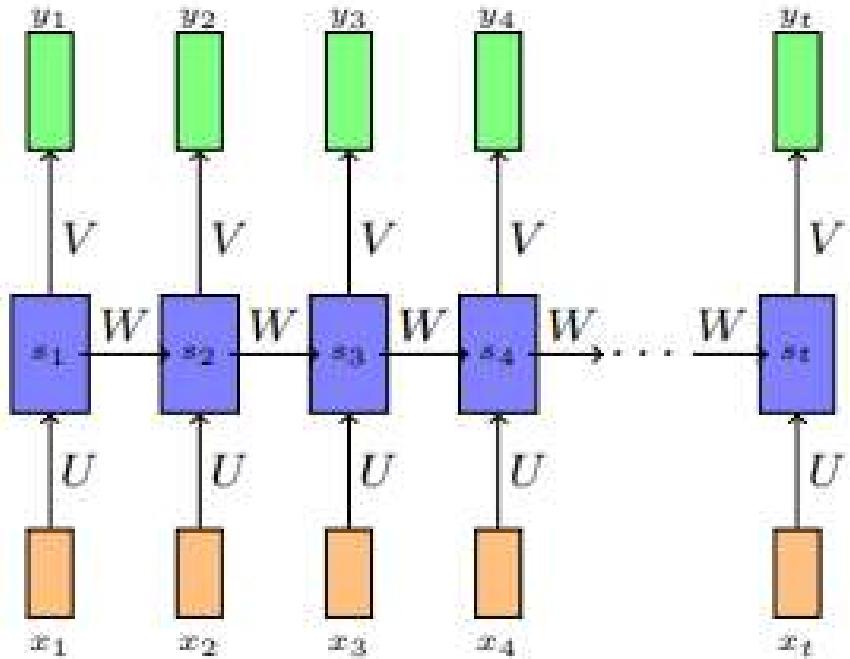
Special circuits for handling long term dependencies

# ISSUE OF MAINTAINING STATES

---

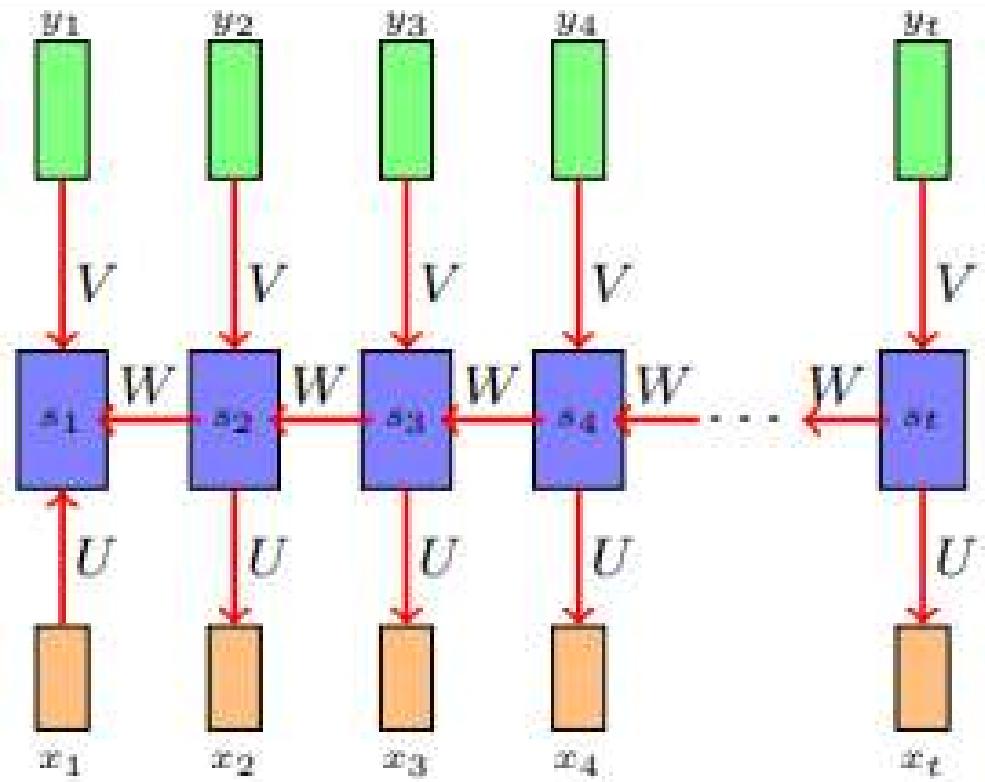
- The old information gets morphed by the current input at each new time step. After  $t$  steps the information stored at time step  $t - k$  (for some  $k < t$ ) gets completely morphed so much that it would be impossible to extract the original information stored at time step  $t - k$ .
- It is very hard to assign the responsibility of the error caused at time step  $t$  to the events that occurred at time step  $t - k$ .
- Basically depends on the size of memory that is available.

# ISSUE OF MAINTAINING STATES



- The state ( $s_i$ ) of an RNN records information from all previous time steps
- At each new timestep the old information gets morphed by the current input
- One could imagine that after  $t$  steps the information stored at time step  $t - k$  (for some  $k < t$ ) gets completely morphed so much that it would be impossible to extract the original information stored at time step  $t - k$

# ISSUE OF MAINTAINING STATES



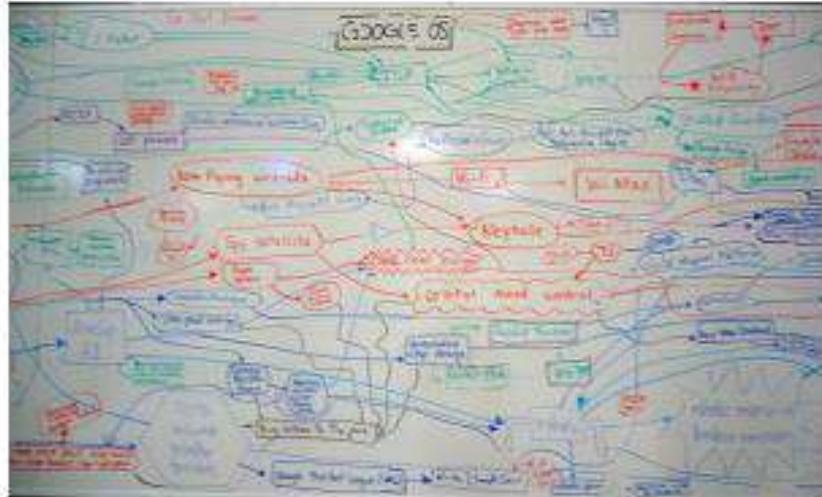
- A similar problem occurs when the information flows backwards (backpropagation)
- It is very hard to assign the responsibility of the error caused at time step  $t$  to the events that occurred at time step  $t - k$
- This responsibility is of course in the form of gradients and we studied the problem in backward flow of gradients

# ISSUE OF MAINTAINING STATES

---

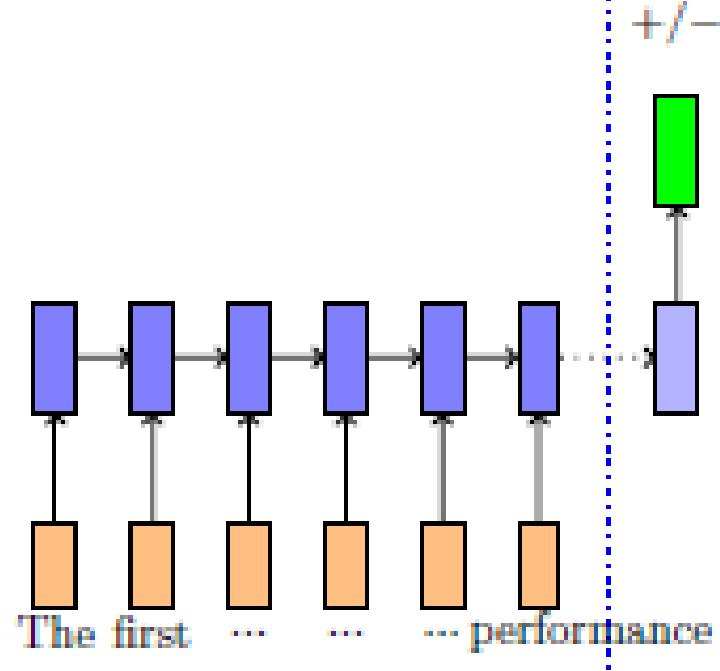
- Selectively write on the states.
  - Selectively read the already written content.
  - Selectively forget (erase) some content.
-

# Whiteboard Analogy



- Let us see an analogy for this
- We can think of the state as a fixed size memory
- Compare this to a fixed size white board that you use to record information
- At each time step (periodic intervals) we keep writing something to the board
- Effectively at each time step we morph the information recorded till that time point
- After many timesteps it would be impossible to see how the information at time step  $t - k$  contributed to the state at timestep  $t$ .

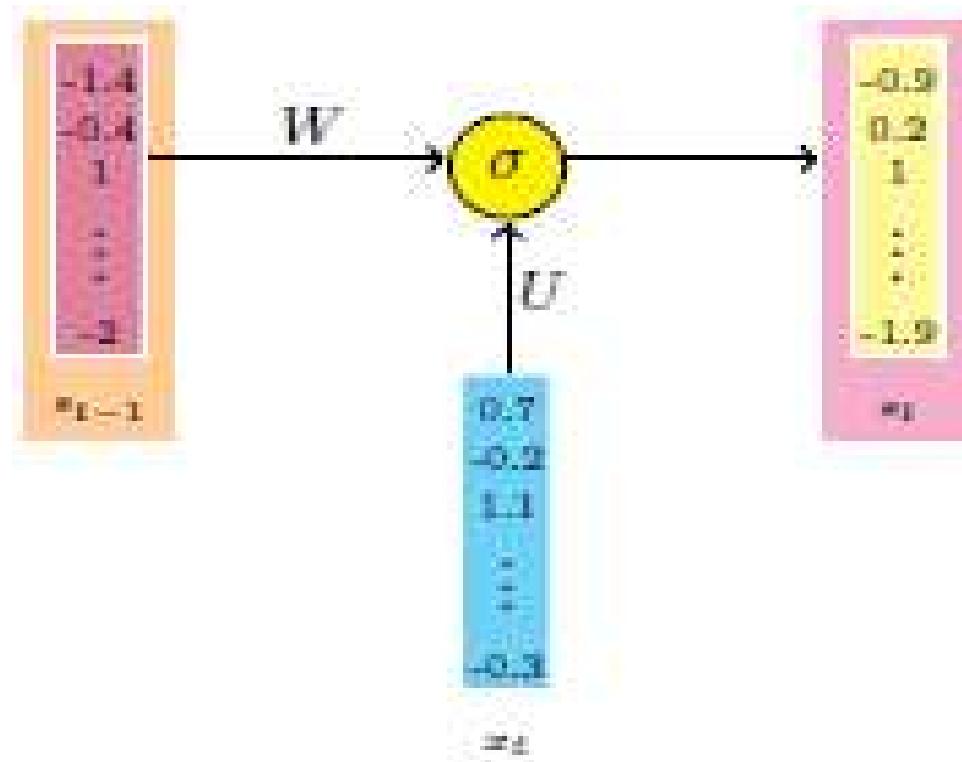
# SENTIMENT ANALYSIS



**Review:** The first half of the movie was dry but the second half really picked up pace. The lead actor delivered an amazing performance

- RNN reads the document from left to right and after every word updates the state.
- By the time we reach the end of the document the information obtained from the first few words is completely lost.
- Ideally we want to
  - ) forget the information added by stop words (a, the, etc.).
  - ) selectively read the information added by previous sentiment bearing words (awesome, amazing, etc.)
  - ) selectively write new information from the current word to the state.

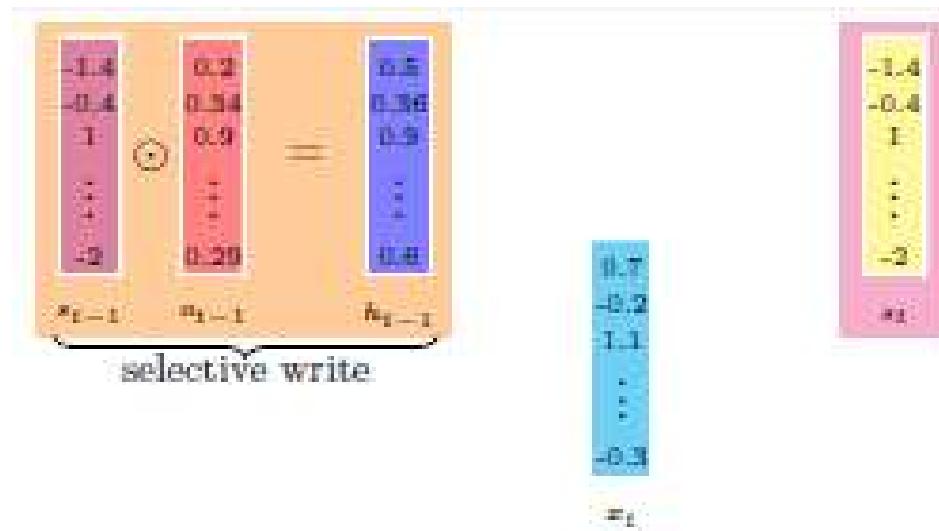
# SELECTIVE WRITE



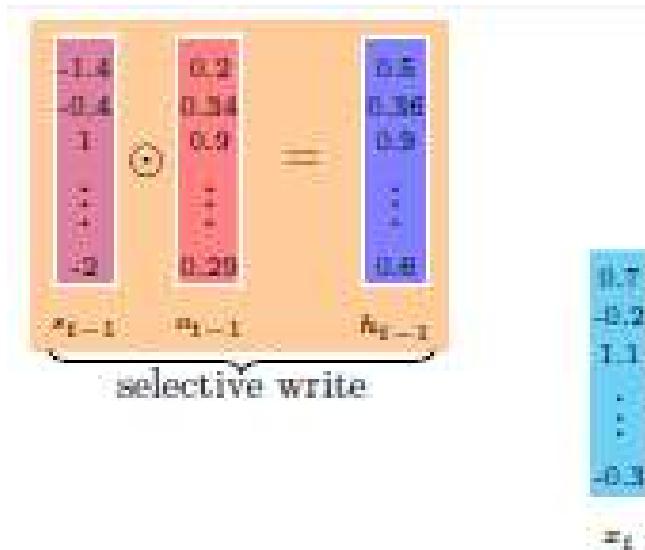
- Recall that in RNNs we use  $s_{t-1}$  to compute  $s_t$ .

$$s_t = \sigma(Ws_{t-1} + Ux_t + b)$$

# SELECTIVE WRITE



- Introduce a vector  $o_{t-1}$  which decides what fraction of each element of  $s_{t-1}$  should be passed to the next state.
- Each element of  $o_{t-1}$  gets multiplied with the corresponding element of  $s_{t-1}$ .
- Each element of  $o_{t-1}$  is restricted to be between 0 and 1.
- The RNN has to learn  $o_{t-1}$  along with the other parameters ( $W, U, V$ ).



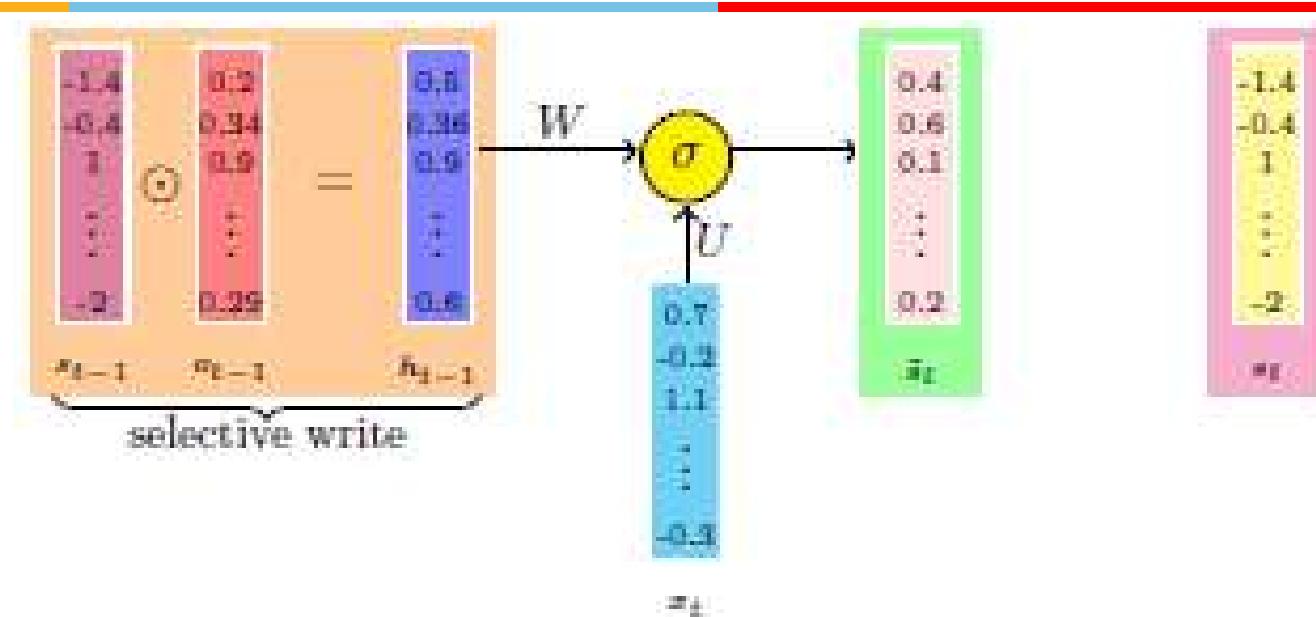
- Compute  $o_{t-1}$  and  $h_{t-1}$  as

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

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

- The parameters ( $W_o, U_o, b_o$ ) are learned along with the existing parameters ( $W, U, V$ ).
- The sigmoid function ensures that the values are between 0 and 1.
- $o_t$  is called the **output gate** as it decides how much to pass (write) to the next time step.

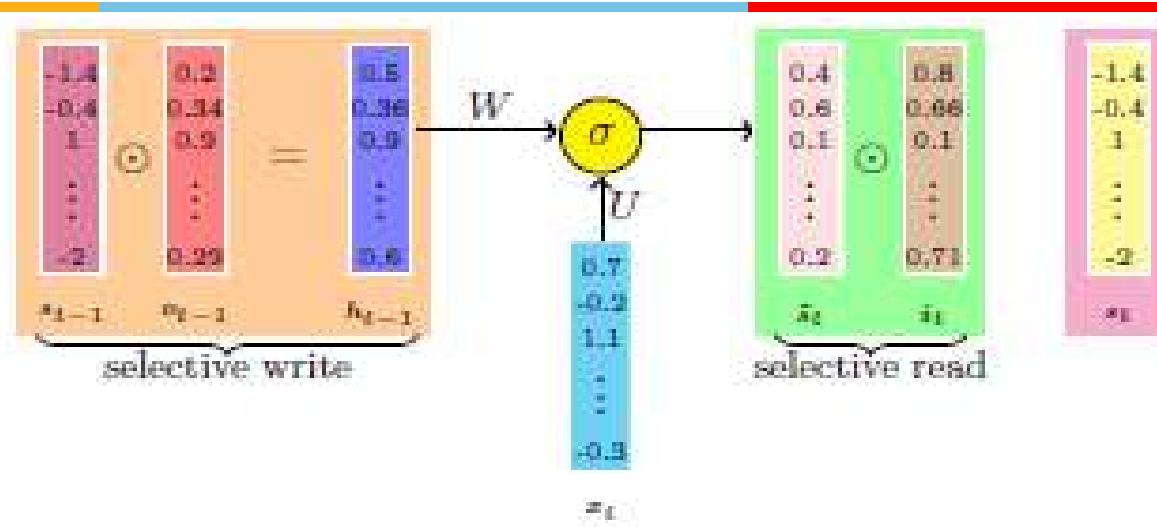
# COMPUTE STATE



- $h_{t-1}$  and  $x_t$  are used to compute the new state at the next time step.

$$\tilde{s}_t = \sigma(Wh_{t-1} + Ux_t + b)$$

# SELECTIVE READ

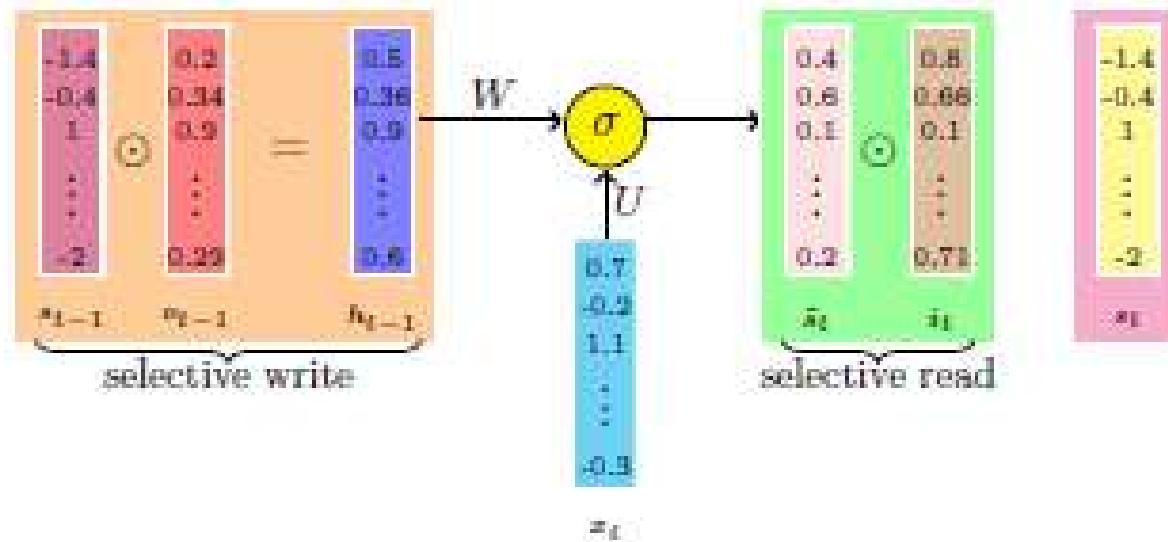


- $\tilde{s}_t$  captures all the information from the previous state  $h_{t-1}$  and the current input  $x_t$ .
- To do selective read, introduce another gate called the **input gate**.

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

$$\text{Selectively Read } = i_t \circ \tilde{s}_t$$

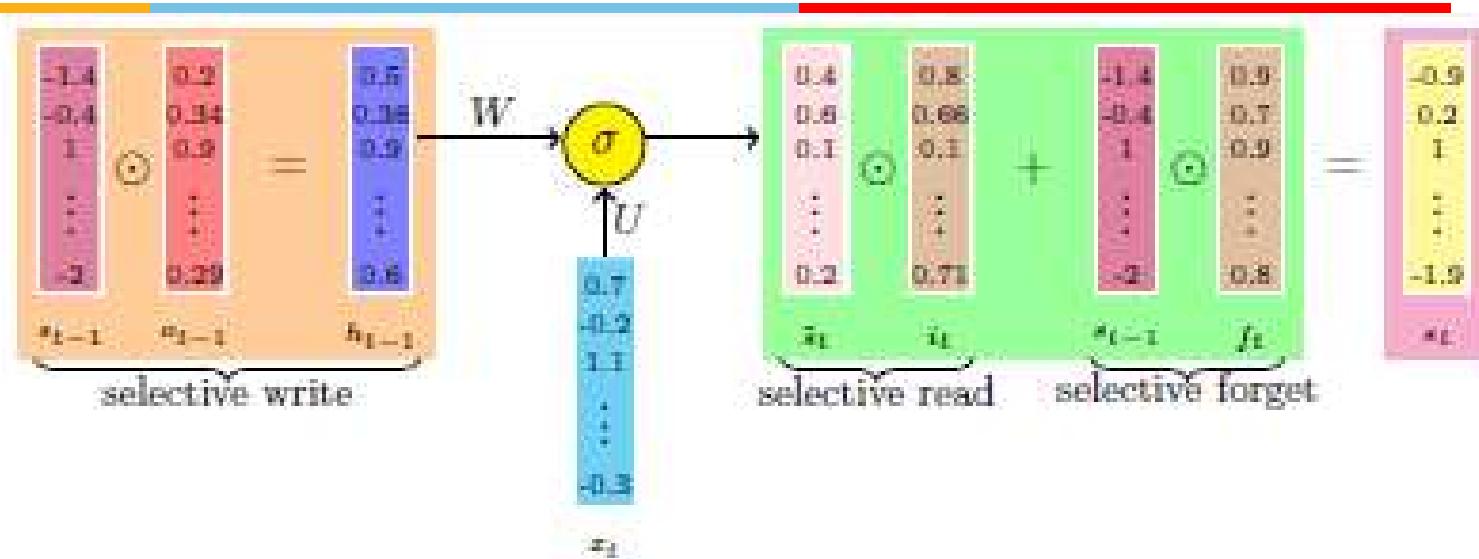
# SELECTIVE READ



- $\tilde{s}_t$  captures all the information from the previous state  $h_{t-1}$  and the current input  $x_t$ .
- To do selective read, introduce another gate called the **input gate**.

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

# SELECTIVE FORGET

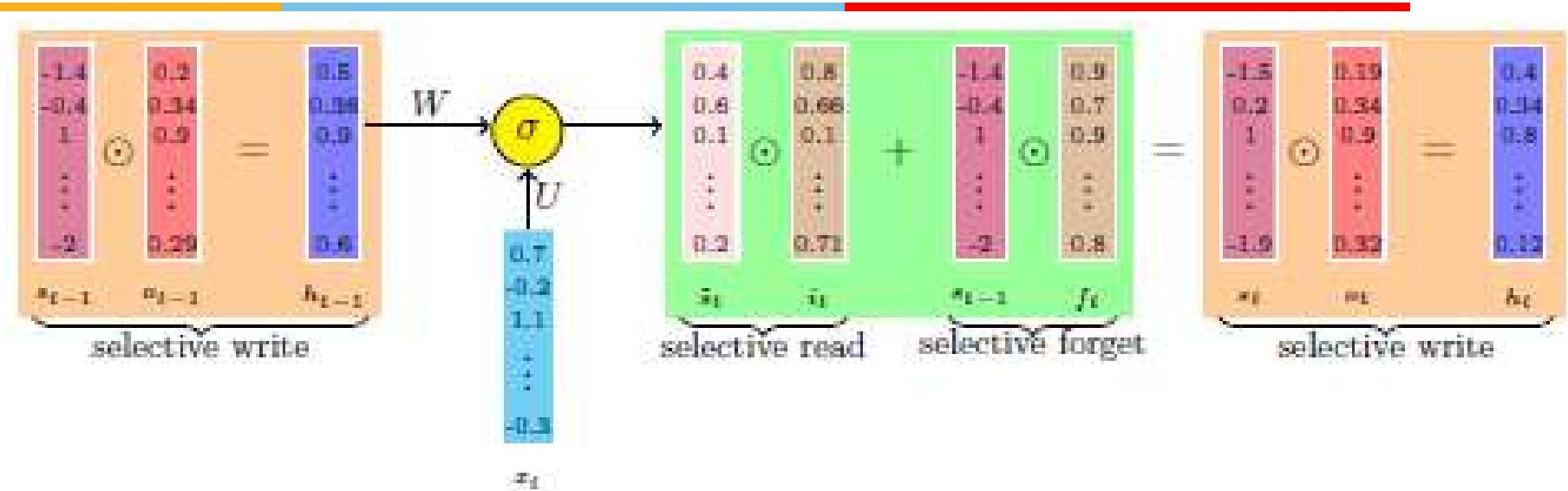


- To do selective forget, introduce another gate called the **forget gate**.

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

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

# FULL LSTM



- 3 gates

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

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

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

- 3 states

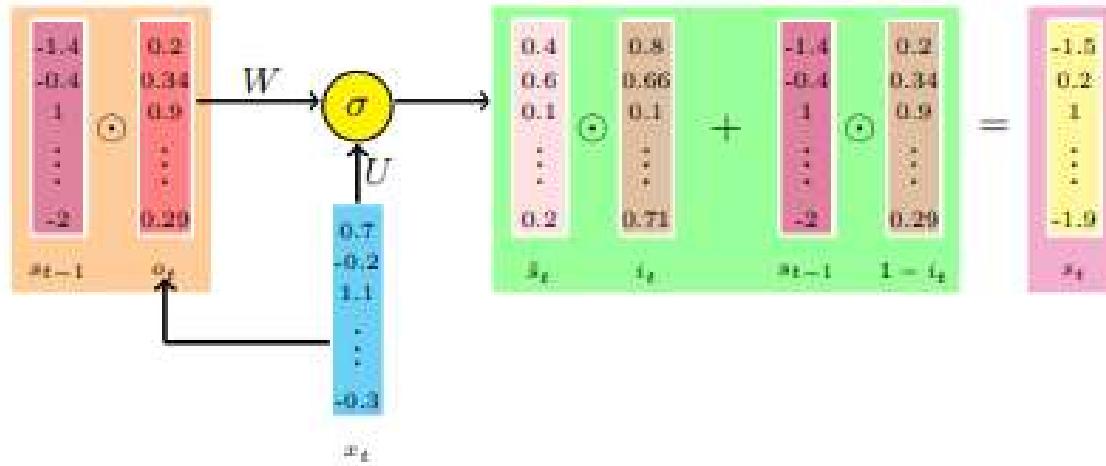
$$\tilde{s}_t = \sigma(Wh_{t-1} + Ux_t + b)$$

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

$$h_t = o_t \circ \sigma(s_t)$$

$$\hat{y}_t = g(Vs_t + c)$$

# Gated Recurrent Network(GRU)



The full set of equations for GRUs

### Gates:

$$o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$$

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

### States:

$$\tilde{s}_t = \sigma(W(\tilde{s}_t) + Ux_t + b)$$

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

- No explicit forget gate (the forget gate and input gates are tied)
- The gates depend directly on  $s_{t-1}$  and not the intermediate  $h_{t-1}$  as in the case of LSTMs

# Applications



- Use GRU, when dependency is short. Eg: Weather forecasting Use LSTM, when dependency is long. Eg: NLP Translation

# References

- 
- 1 Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron Courville  
<https://www.deeplearningbook.org/>
  - 2 Deep Learning with Python by Francois Chollet.  
<https://livebook.manning.com/book/deep-learning-with-python/>

Mitesh Khapra RNN

<https://www.youtube.com/watch?v=BBTTS9gx7Q&t=153s>

Mitesh Khapra LSTM

<https://www.youtube.com/watch?v=9TFnjJkfqmA>

[https://www.youtube.com/watch?v=9TFnjJkfqmA&list=PLEAYkSg4uSQ1r-2XrJ\\_GBzzS6I-f8yfRU&index=110](https://www.youtube.com/watch?v=9TFnjJkfqmA&list=PLEAYkSg4uSQ1r-2XrJ_GBzzS6I-f8yfRU&index=110)

---



Dr. Chetana Gavankar has over 25 years of Teaching, Research and Industry experience. She has published papers in peer reviewed international conferences and journals. She is also reviewer for multiple conferences and journals. She has worked on different projects with multiple industries and received awards for her research work . Her areas of research interests include Natural Language Processing, Information Retrieval, Web Mining and Semantic Web, Ontology, Big Data Analytics, Machine learning, Deep learning and Artificial Intelligence.



# C6: ANN and Deep Learning



**BITS Pilani**  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)



**Session 5  
Date – 30<sup>th</sup> Dec 2023**

**Time – 10 am to 12.15pm**

These slides are prepared by the instructor, with grateful acknowledgement of and many others who made their course materials freely available online.

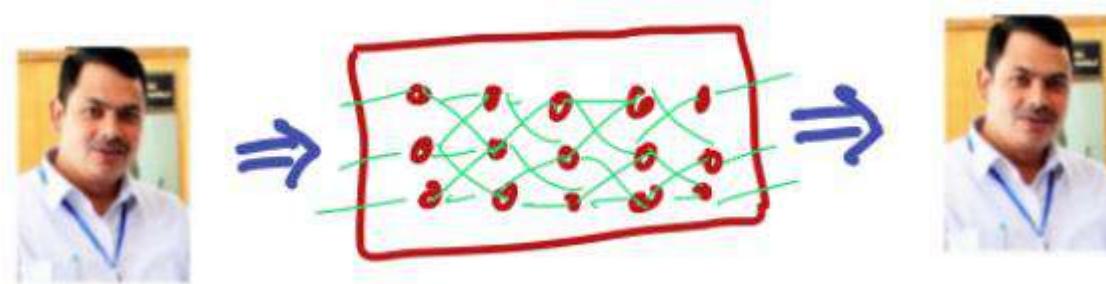
# Autoencoders with Deep Learning

---

- Undercomplete Autoencoders
- Regularized Autoencoders
- Variational autoencoders
- Applications of Autoencoders

# Autoencoder

A neural network that is trained to copy its input to its output (not sure how successful it would be)



Internally, it has a hidden layer  $h$  that describes a code used to represent the input  $x$

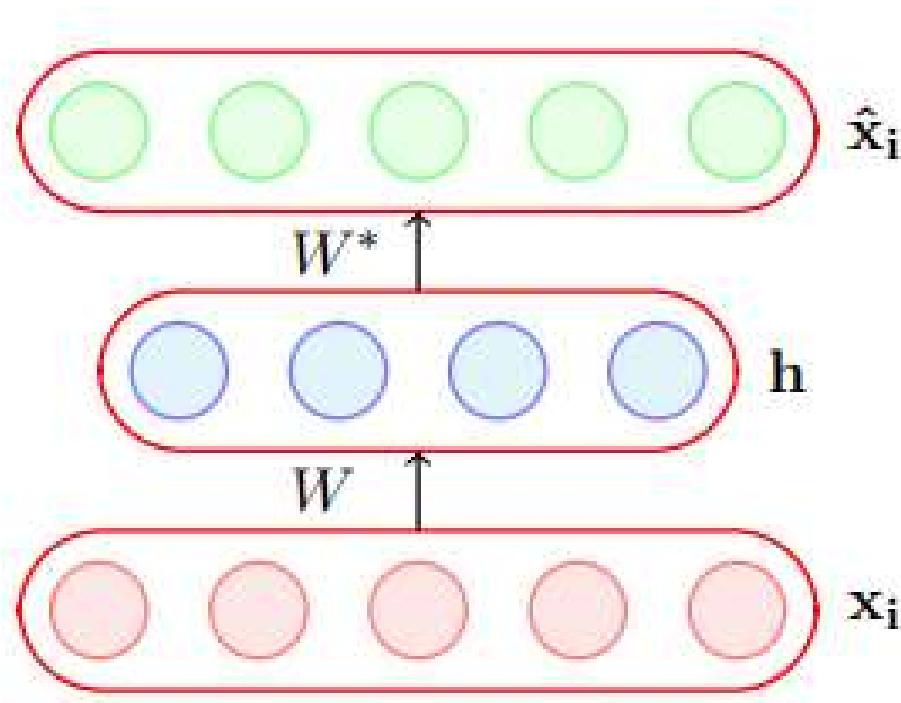
The network has two parts

1. An encoder function  $h = f(x)$
2. And a decoder that produces a reconstruction  $r = g(h)$

Network is designed such that it is not easy to learn a perfect copy

**Essentially, we need  $g(f(x)) = x$**

# Autoencoder

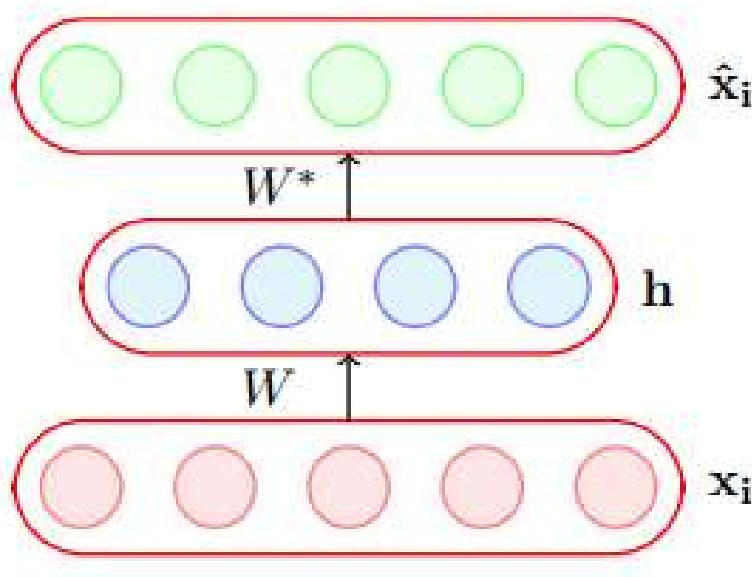


$$h = g(Wx_i + b)$$

$$\hat{x}_i = f(W^*h + c)$$

- An autoencoder is a special type of feed forward neural network which does the following
- Encodes its input  $x_i$  into a hidden representation  $h$
- Decodes the input again from this hidden representation
- The model is trained to minimize a certain loss function which will ensure that  $\hat{x}_i$  is close to  $x_i$  (we will see some such loss functions soon)

# Undercomplete autoencoders



$$\mathbf{h} = g(W \mathbf{x}_i + \mathbf{b})$$

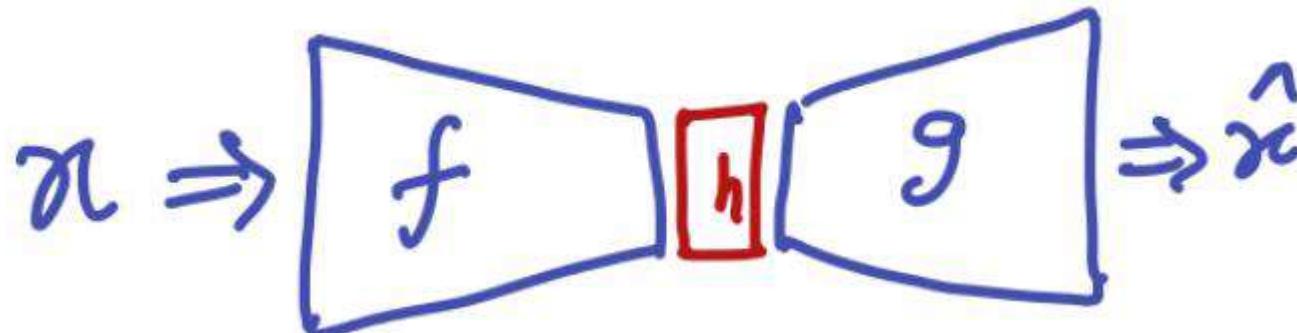
$$\hat{\mathbf{x}}_i = f(W^* \mathbf{h} + \mathbf{c})$$

- Let us consider the case where  $\dim(\mathbf{h}) < \dim(\mathbf{x}_i)$
- If we are still able to reconstruct  $\hat{\mathbf{x}}_i$  perfectly from  $\mathbf{h}$ , then what does it say about  $\mathbf{h}$ ?
- $\mathbf{h}$  is a loss-free encoding of  $\mathbf{x}_i$ . It captures all the important characteristics of  $\mathbf{x}_i$
- Do you see an analogy with PCA?

An autoencoder where  $\dim(\mathbf{h}) < \dim(\mathbf{x}_i)$  is called an under complete autoencoder

# Under complete autoencoders

Interesting things happen when we force  $h$  to be of smaller then  $x$



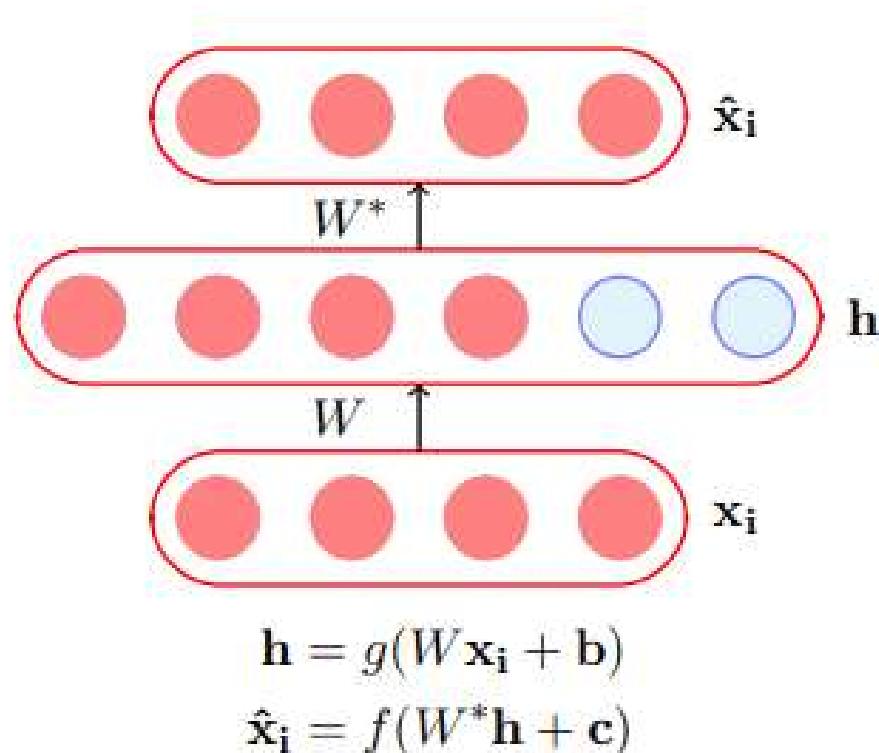
Network is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data

Learning process minimizes the loss  $L(x, g(f(x)))$

When decoder is linear and  $L$  is MSE, it learns subspace as PCA

$f$  and  $g$  with too much capacity may fail to learn

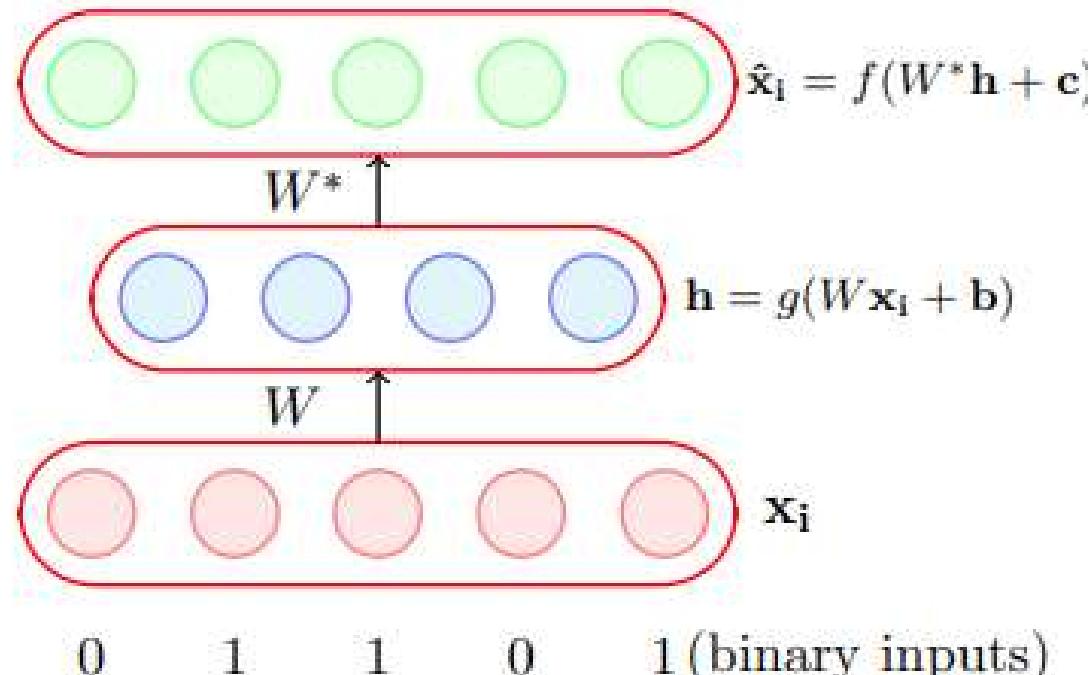
# Over complete Encoder



- Let us consider the case when  $\dim(h) \geq \dim(x_i)$
- In such a case the autoencoder could learn a trivial encoding by simply copying  $x_i$  into  $h$  and then copying  $h$  into  $\hat{x}_i$
- Such an identity encoding is useless in practice as it does not really tell us anything about the important characteristics of the data

An autoencoder where  $\dim(h) \geq \dim(x_i)$  is called an over complete autoencoder

# Choice of $f(x_i)$ and $g(x_i)$



$g$  is typically chosen as the sigmoid function

- Suppose all our inputs are binary ( $\text{each } x_{ij} \in \{0, 1\}$ )
- Which of the following functions would be most apt for the decoder?

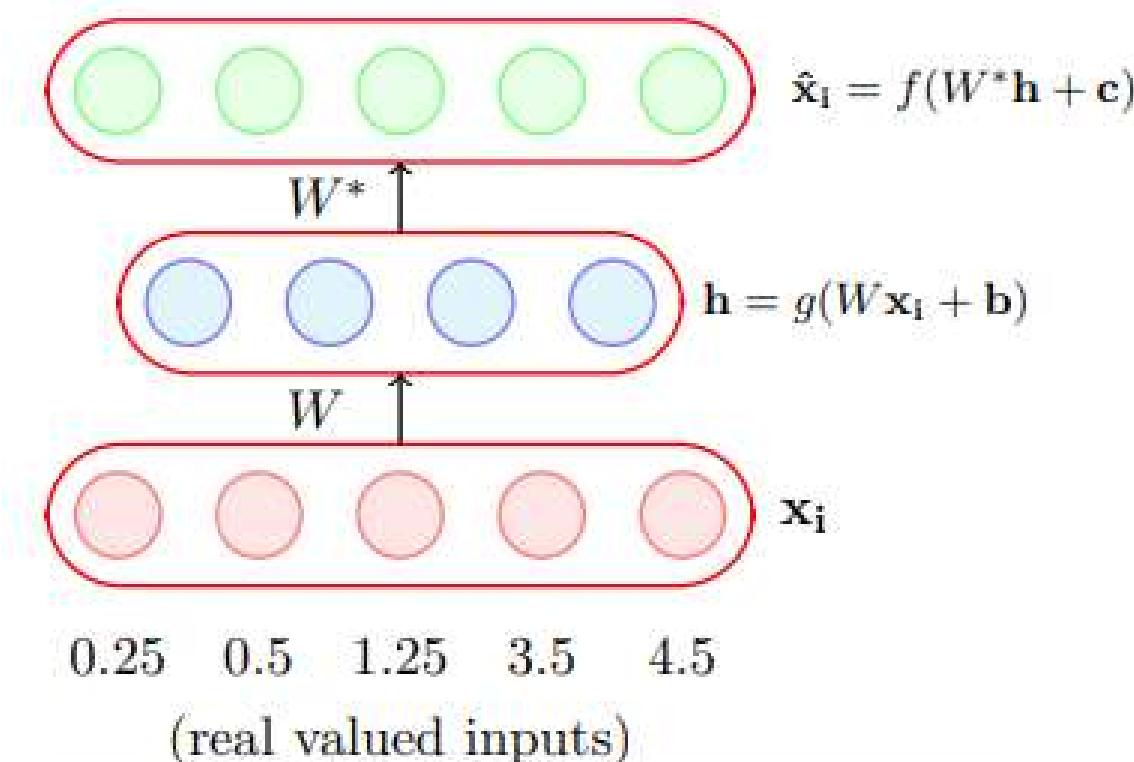
$$\hat{\mathbf{x}}_i = \tanh(W^*\mathbf{h} + \mathbf{c})$$

$$\hat{\mathbf{x}}_i = W^*\mathbf{h} + \mathbf{c}$$

$$\hat{\mathbf{x}}_i = \text{logistic}(W^*\mathbf{h} + \mathbf{c})$$

- Logistic as it naturally restricts all outputs to be between 0 and 1

# Choice of $f(x_i)$ and $g(x_i)$



Again,  $g$  is typically chosen as the sigmoid function

- Suppose all our inputs are real (each  $x_{ij} \in \mathbb{R}$ )
- Which of the following functions would be most apt for the decoder?

$$\hat{x}_i = \tanh(W^* \mathbf{h} + \mathbf{c})$$

$$\hat{x}_i = W^* \mathbf{h} + \mathbf{c}$$

$$\hat{x}_i = \text{logistic}(W^* \mathbf{h} + \mathbf{c})$$

- What will logistic and tanh do?
- They will restrict the reconstructed  $\hat{x}_i$  to lie between [0,1] or [-1,1] whereas we want  $\hat{x}_i \in \mathbb{R}^n$

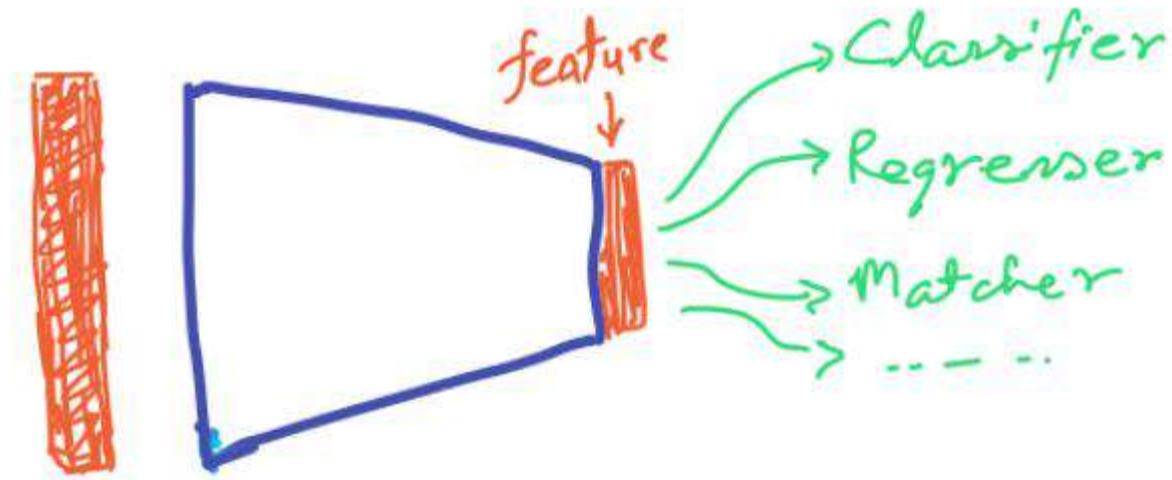
# Regularized Autoencoders

- As we know the learning process minimizes the loss  $L(x, g(f(x)))$
- In overcomplete cases, encoder and decoder could learn to copy input to output without learning anything useful
- Encoder/decoder architecture should be chosen based on the complexity of distribution to be modeled
- Loss function of regularized autoencoders encourages the model to have other properties like sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs
- **Sparse autoencoder** has loss function as  $L(x, g(f(x))) + \Omega(h)$  where  $\Omega(h)$  correspond to some other task such as classification
- **Denoising autoencoder** minimizes  $L(x, g(\tilde{f}(x)))$  where  $\tilde{x}$  is a copy corrupted by some noise
- **Contractive autoencoder** minimizes  $L(x, g(f(x))) + \lambda \sum_i \|\nabla_x h_i\|^2$  that forces  $h$  not to change much when  $x$  changes

# Autoencoders As Feature Extractor



Can serve as feature extractor for many applications



End-to-end training could be done on extracted feature

Train classifier, regressor, matching module, segmentation module, or ...

Feature could also be taken from inside the encoder

With classifier, you could sift the focus of the feature

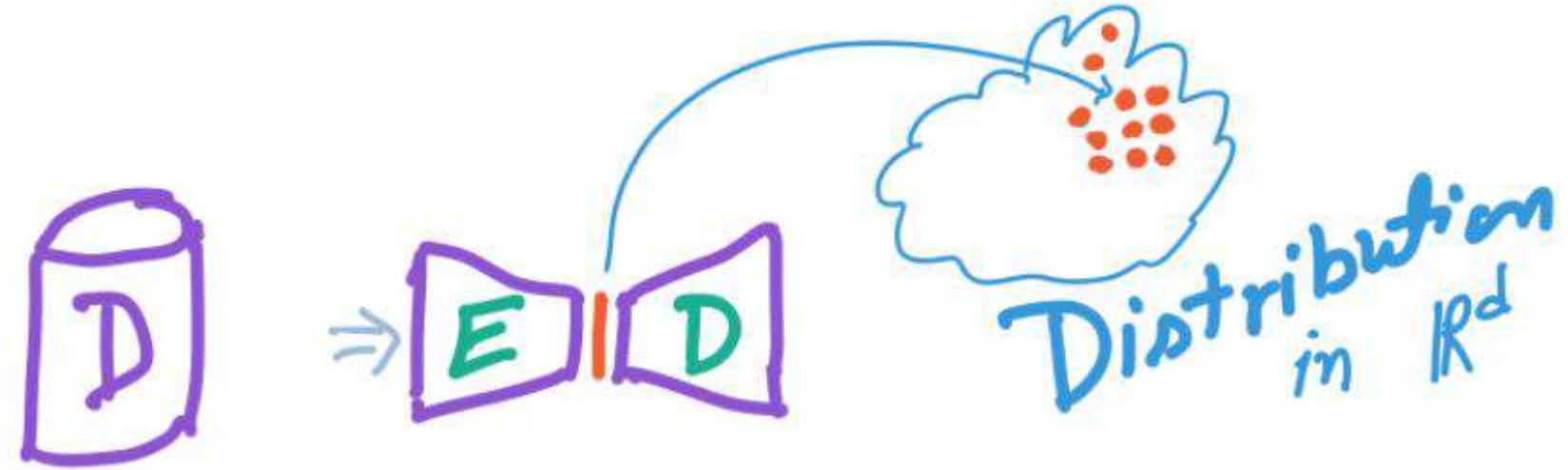
# Autoencoders As Feature Extractor

---

- Autoencoder could be an unsupervised feature extractor
- Classifier on bottleneck converges very fast
- Decoder had never seen the original image it has only seen the code  $h$ . But, distribution is captured
- What would happen if you randomly perturb  $h$  and show to decoder
- How to get new cat images?

# Variational Autoencoders

A generative model



$h$  corresponding to a dataset would not cover whole space

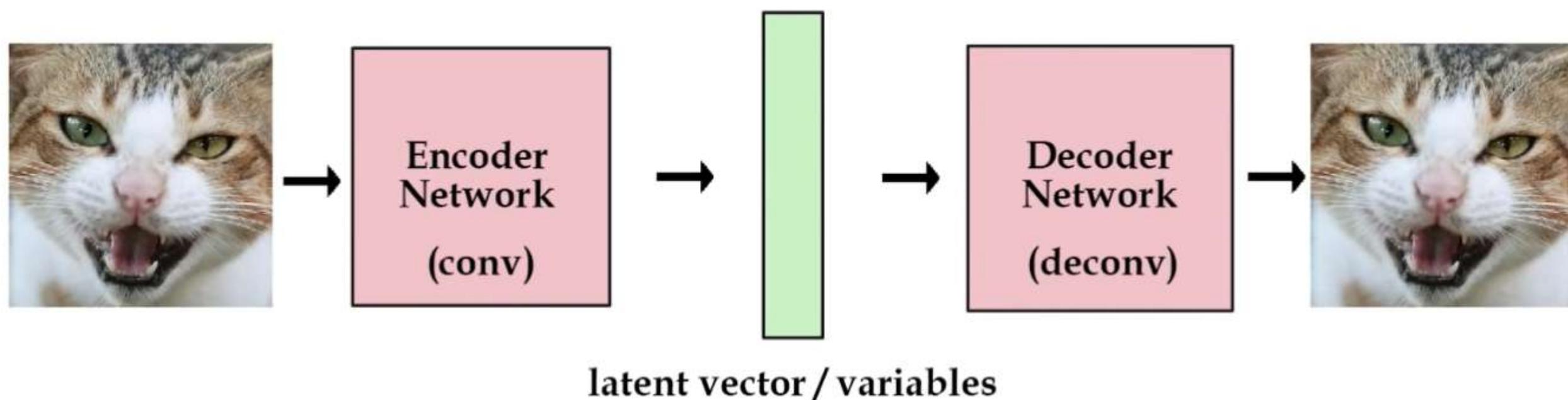
Consider randomly perturb  $h$ , it would be very similar

How to sample a new latent vector point in feature space

**It is going to give me something that is not in my database**

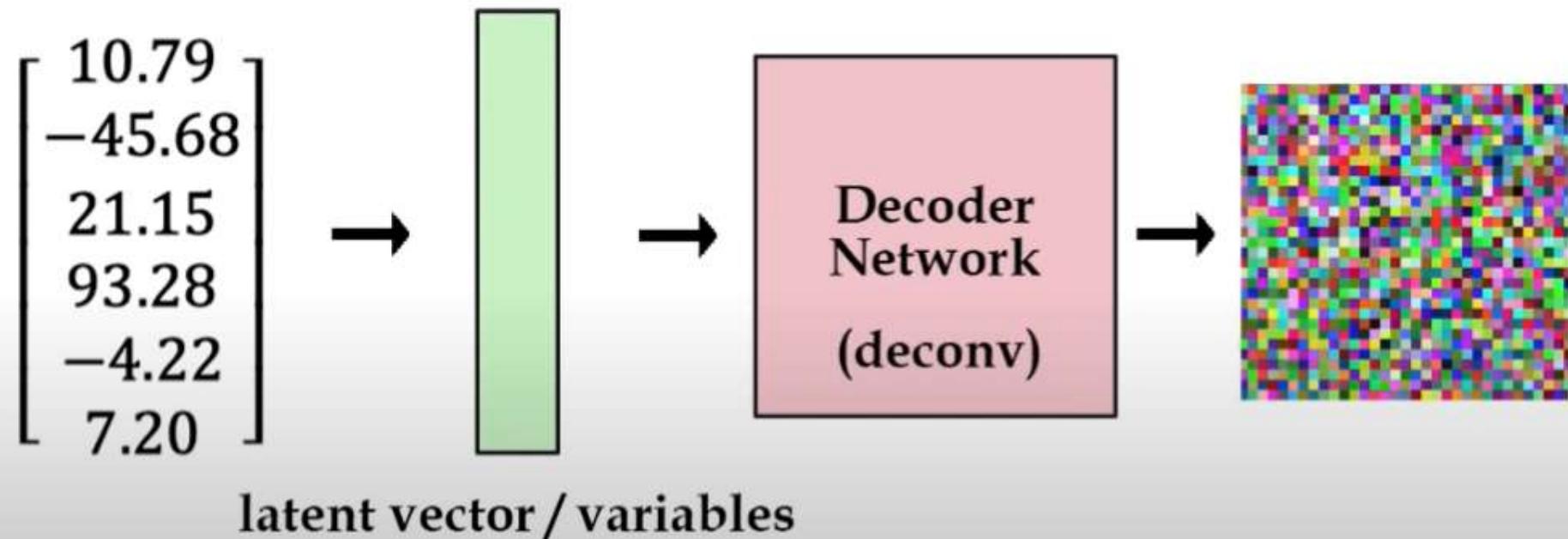
# AutoEncoder

Training Phase



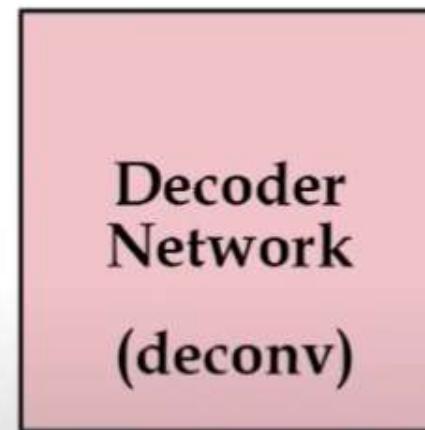
# Autoencoders

Testing Phase



## Testing Phase

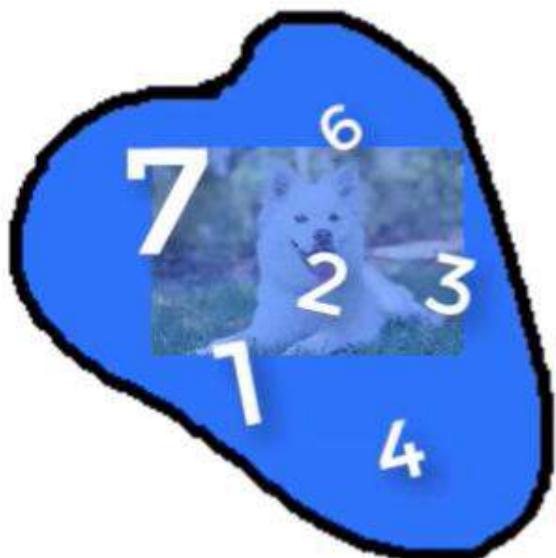
“Sample” from  
a “Distribution” →



latent vector / variables

# Distribution - Concept

Dog  
Distribution



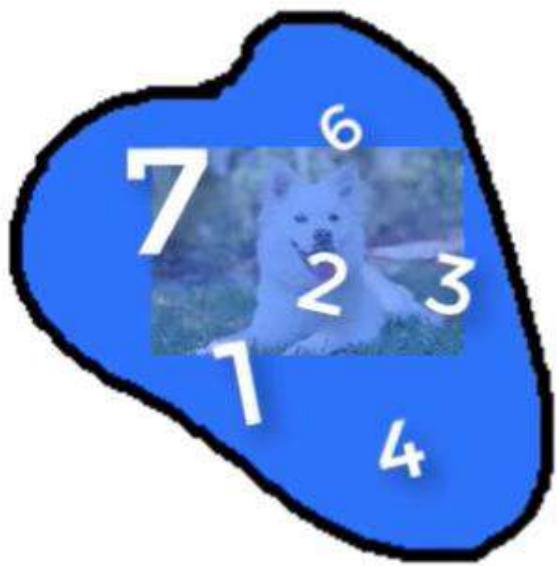
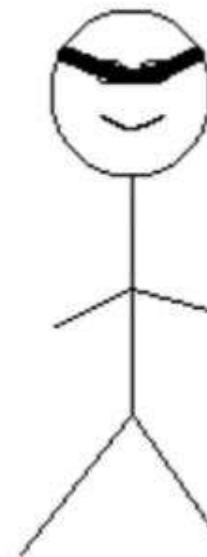
Giraffe  
Distribution



Cat  
Distribution



## Sampling - Concept

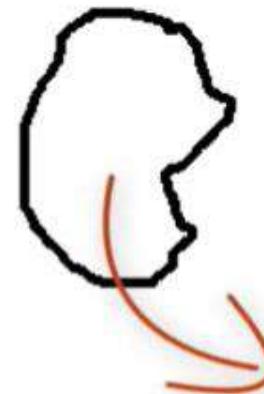

$$\begin{bmatrix} 0.31 \\ 5.62 \\ 9.51 \\ 14.21 \\ -3.01 \end{bmatrix}$$


# Sampling - Concept

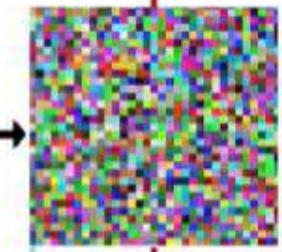
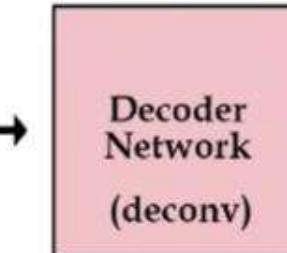
we don't know how  
to access this pool!



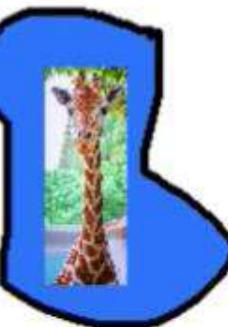
model finds these  
pools during training



$$\begin{bmatrix} 0.69 \\ -4.88 \\ 113.15 \\ 93.81 \\ 14.22 \\ -67.20 \end{bmatrix} \rightarrow$$

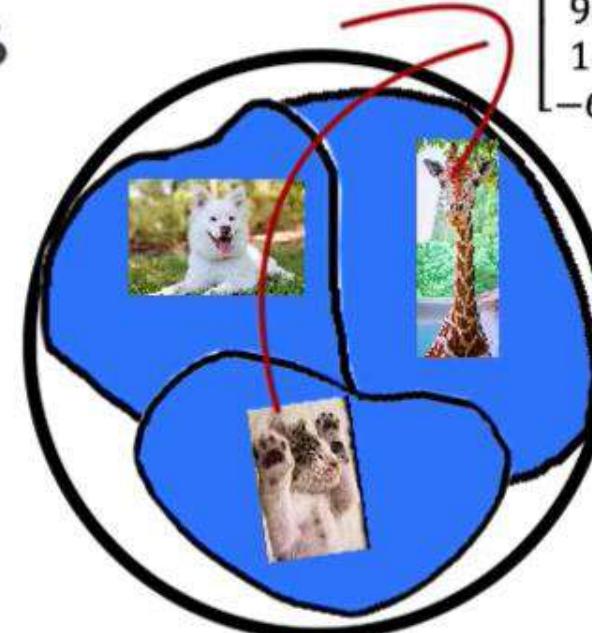


“garbage” all  
over here

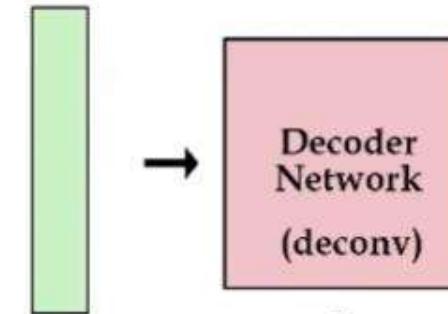


# Sampling - Concept

continuous  
region

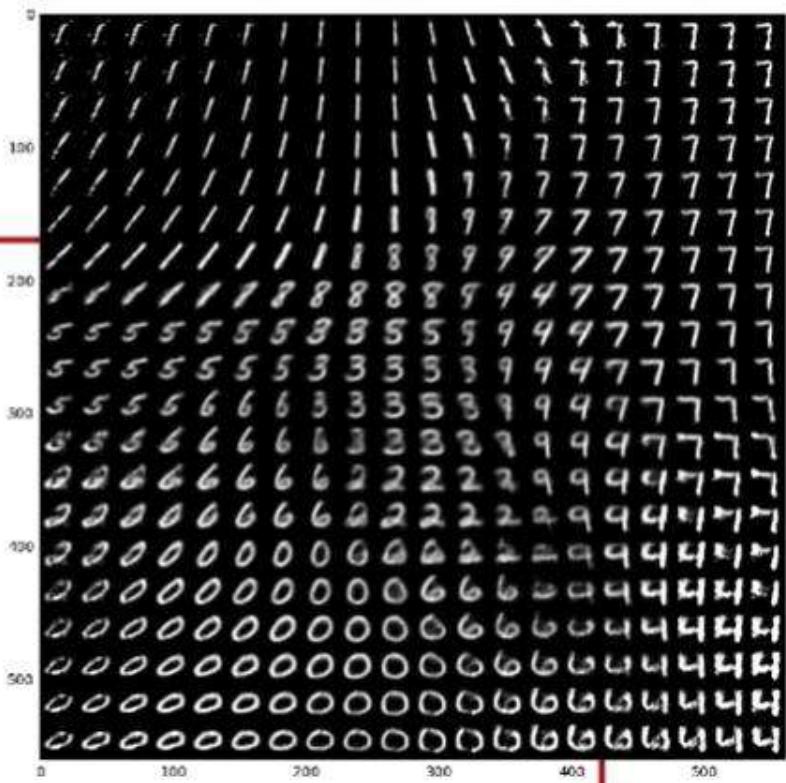
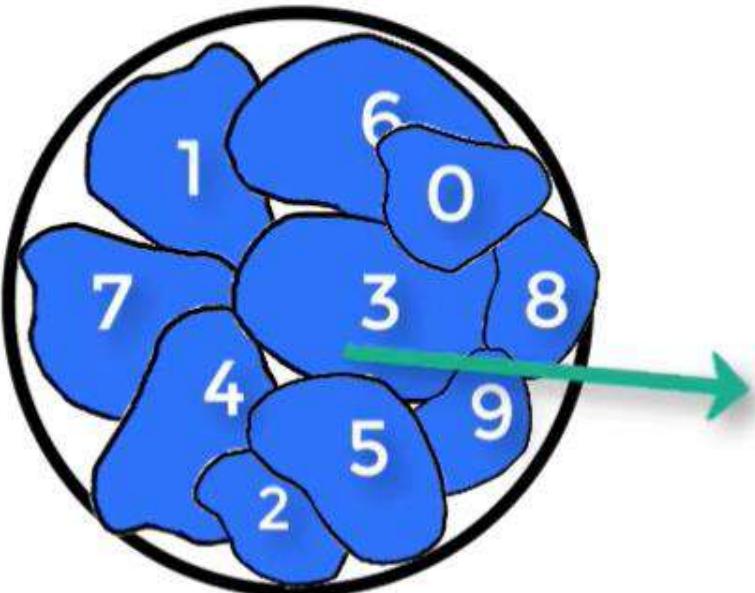


$$\begin{bmatrix} 0.69 \\ -4.88 \\ 113.15 \\ 93.81 \\ 14.22 \\ -67.20 \end{bmatrix} \rightarrow$$



# Sampling - Concept

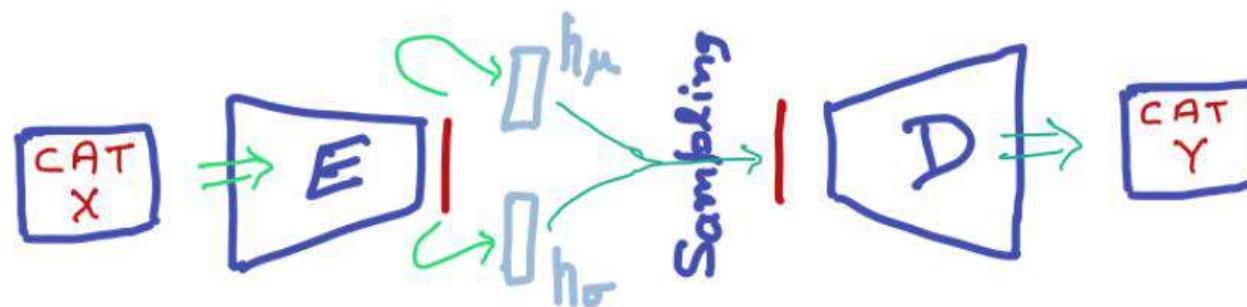
continuous  
region



# Variational Autoencoders

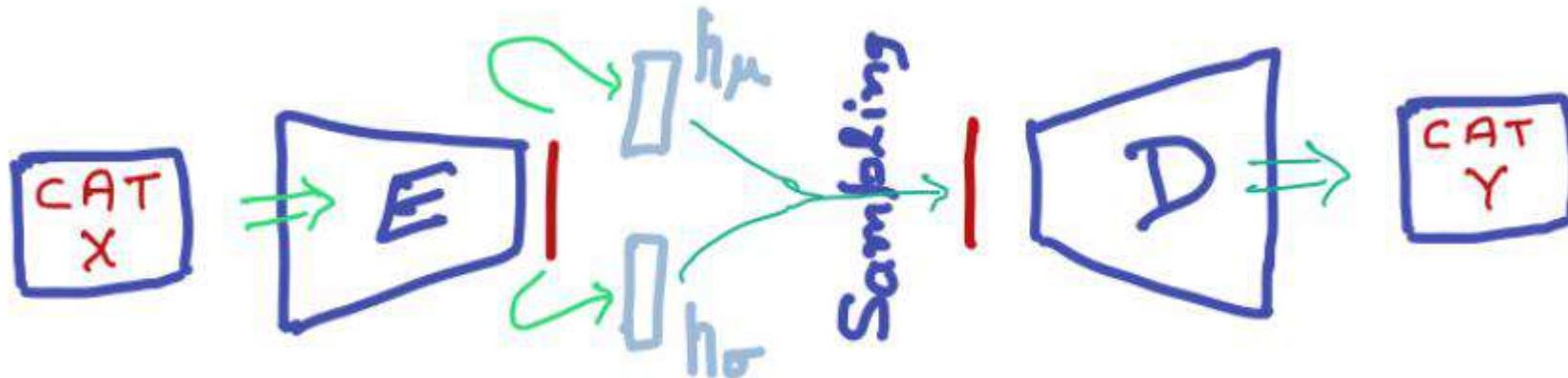
## Two loss

1. Distribution of latent vector is unit normal
  2. Reconstruction loss is minimum  $\|x - x'\|$
- Knowing distribution of latent vector, one can sample a new point
  - If I don't know the distribution of latent variable, can I force it to be one of my known one?



- Why Gaussian? simple, it has only two parameters  $\mu$  and  $\sigma$
- Distance between two distributions can be measured using KL-Divergence

# Variational Autoencoders



## Two loss

Reconstruction loss:  $MSE(X, Y)$

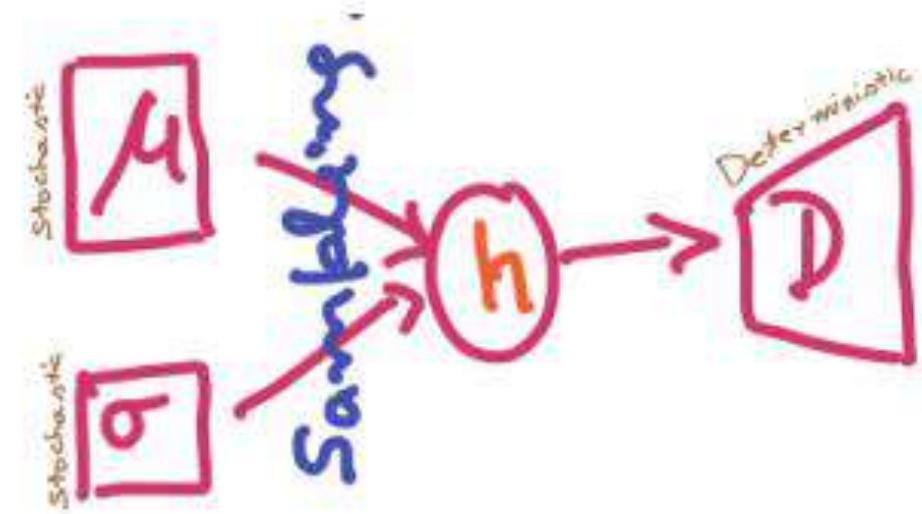
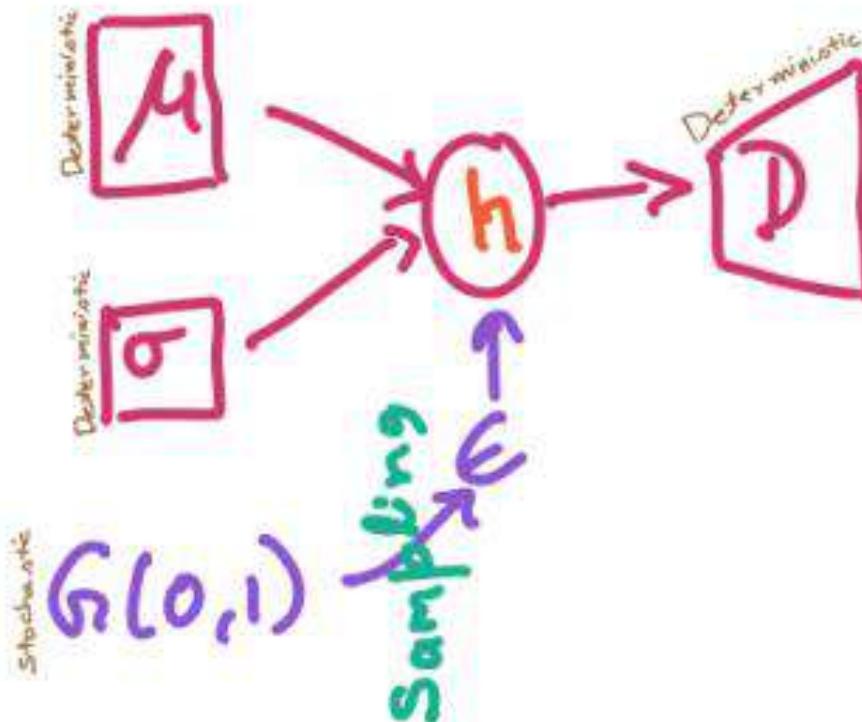
KL-D:  $KLD(G(h_\mu, h_\sigma), N(0, 1))$

Issue is that now you are unable to do backpropagation

Solution is re-parametrization

# Re-parametrization

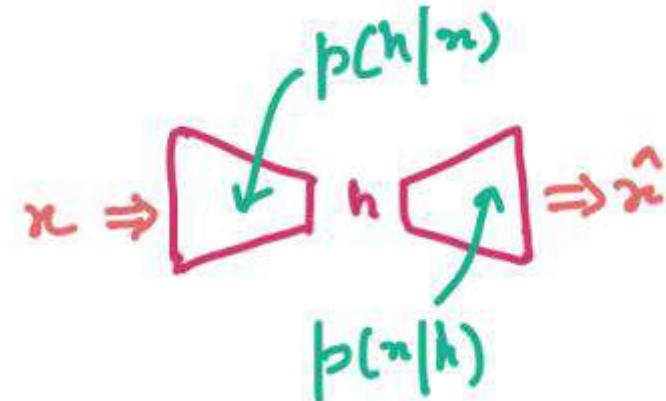
Sampling is Stochastic  
Therefore, it blocks backpropagation



- How would you sample given the  $h_\mu$  and  $h_\sigma$ 
  - Get  $\epsilon$  from unit normal  $N(0, 1)$
  - Then  $\hat{h} = h_\mu + \epsilon \times h_\sigma$

**Important thing is that you do not want to learn anything for  $N(0, 1)$**

# Probabilistic view



Bayes Theorem  $p(A|B) = \frac{p(A \cap B)}{p(B)} = \frac{p(B|A)p(A)}{p(B)}$  if an event

Event and

Statement $x$	Probability $p(x)$	Information – $\log(p(x))$
Sun rise in east	1 (high)	0 (low)
Today is solar eclipse	1/100 (low)	(high)

# Entropy

Entropy is expected value of information with respect to an event. Average in short

$$-\sum_x p(x) \log(p(x))$$

It measures disorder and is zero for pure systems

If you have two distributions  $p$  and  $q$  then you can find the difference between the information content between them as

$$-\sum_x q(x) \log(q(x)) + \sum_x p(x) \log(p(x))$$

KL-Divergence is almost same except that the expectation is always computed with respect to  $p(x)$  when taking KLD of  $q$  with respect to  $p$

$$-\sum_x p(x) \log(q(x)) + \sum_x p(x) \log(p(x))$$

# KL-Divergence

KL-Divergence is

$$\begin{aligned} KL - Div(p(x)|q(x)) &= - \sum_x p(x) \log(q(x)) + \sum_x p(x) \log(p(x)) \\ &= - \sum_x p(x) \log \frac{q(x)}{p(x)} \end{aligned}$$

It is not symmetric

It is always positive

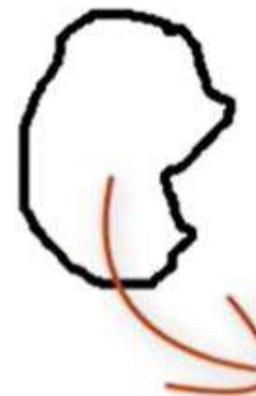
$$KL - Div(q(h)|p(h|x)) = - \sum q(h) \log \frac{p(h|x)}{q(h)}$$

# Sampling - Concept

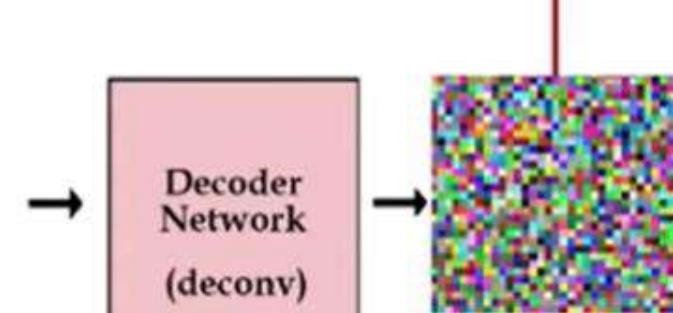
we don't know how  
to access this pool!



model finds these  
pools during training



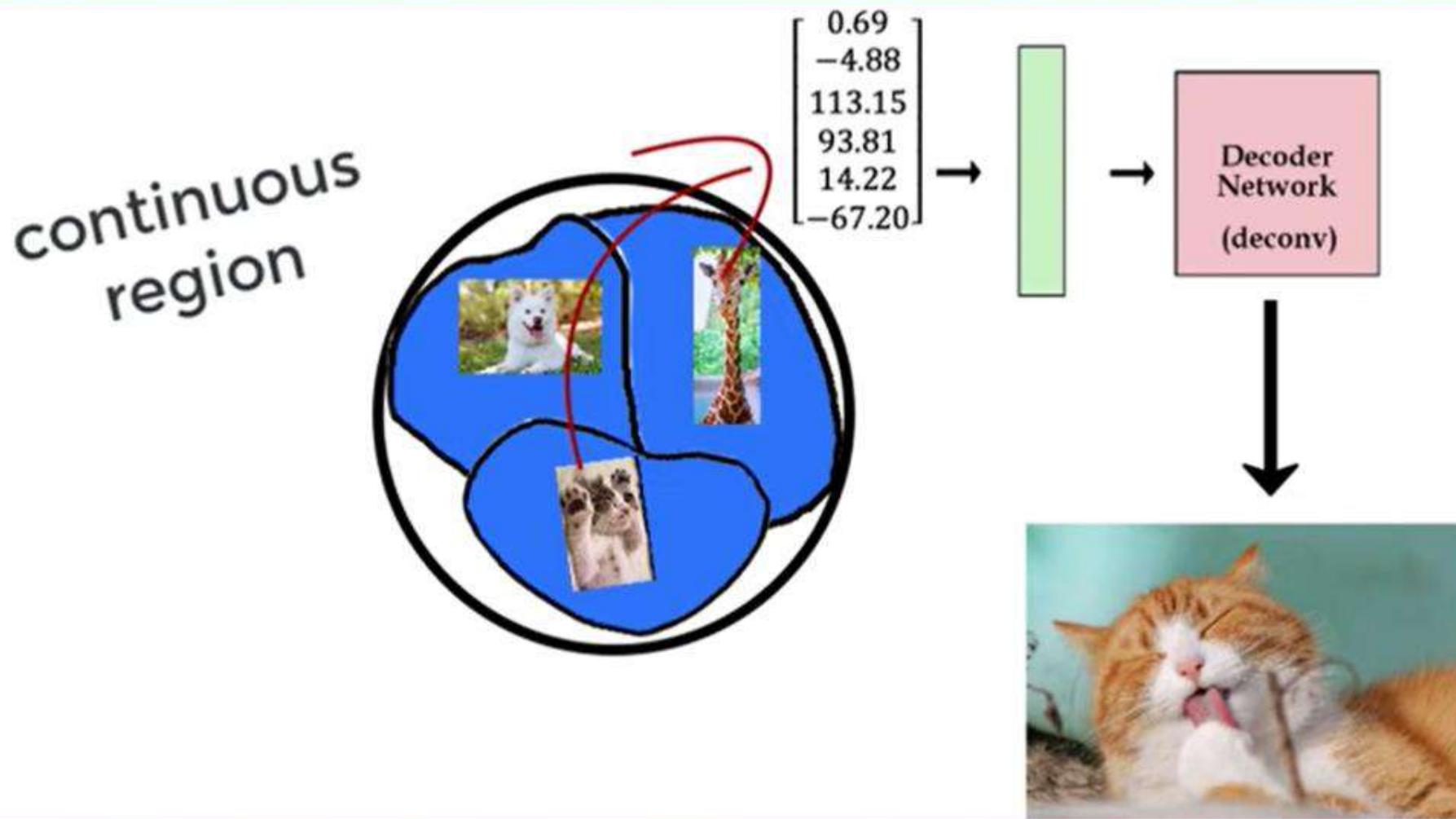
$$\begin{bmatrix} 0.69 \\ -4.88 \\ 113.15 \\ 93.81 \\ 14.22 \\ -67.20 \end{bmatrix} \rightarrow$$



“garbage” all  
over here



# Sampling - Concept



SUBSCRIBE

# An example

---

- Assume you have a database of 1-Million images (1000,1000)
- Each of size  $100 \times 100$  As a pixel have 0-255 i.e. 256 value, we have  $(256)^{10000}$  logically possible images (however, most of them don't make any sense)
- Joint probability distribution  $p(x) = p(x_1, x_2, \dots, x_{10000})$
- Can we get a distribution parameter  $\theta$  such that if we sample from this distribution then the probability of getting an image from our training set is very high.

$$\theta^* = \operatorname{argmax}_{\theta} p(x \in D)$$

- If you know this distribution parameter you could do Generative Modeling

# Issue is

Issue is that the pixel values are not independent

$$p(x) = p(x_1, x_2, \dots, x_{10000}) = p(x_1) \times p(x_2|x_1) \times p(x_3|x_1, x_2) \dots$$

Due to this the functions became intractable

However, as there is huge amount of redundancy in the data

There could be some  $z_1, z_2, \dots, z_{100}$  hidden variables that directly influence  $p(x)$

Distribution of the  $x$  is directly influenced by marginalization of  $z$  so

$$p(x) = \sum_z p(x, z) = \sum_z p(x|z).p(z)$$

Finding this is also difficult as number of parameters are still large.

# Issue is

As  $p(z|x) = p(z, x)/p(x)$  so we cannot even estimate  $p(z|x)$

Let us approximate  $p(z|x)$  via  $q(z) \in L\{\text{tractable function family}\}$

$$\begin{aligned}
 q^*(z) &= \arg \min_{q(z) \in L} KL-Div(q(z)|p(z|x)) = - \sum_z q(z) \log \frac{p(z|x)}{q(z)} \\
 &= - \sum_z q(z) \log \frac{p(z, x)}{p(x)q(z)} = \sum_z q(z) \left[ \log \frac{p(z, x)}{q(z)} - \log(p(x)) \right] \\
 &= - \sum_z q(z) \log \frac{p(z, x)}{q(z)} + \log(p(x)) \sum_z q(z) \\
 &= - \sum_z q(z) \log \frac{p(z, x)}{q(z)} + \log(p(x))
 \end{aligned}$$

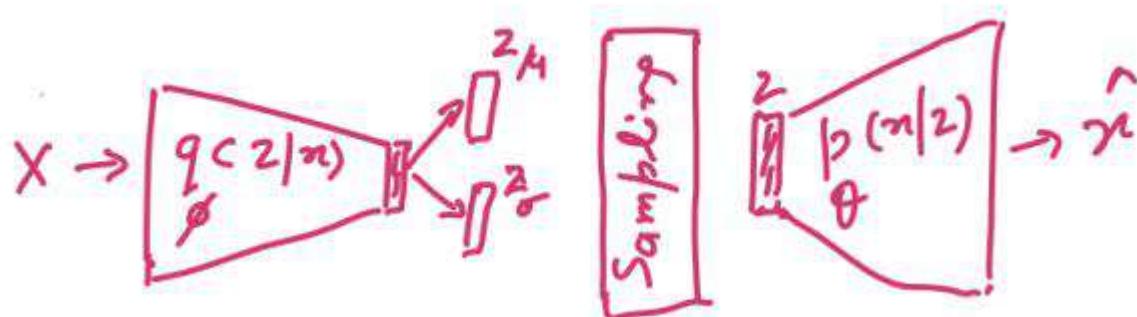
$$log(p(x)) = KL-Div(q(z)|p(z|x)) + \sum_z q(z) \log \frac{p(z, x)}{q(z)}$$

Maximize ELB

$$A = B + C$$

# VAE minimizes MSE and KLD

## ELBO Maximization



ELBO is equivalent

$$\begin{aligned}
 \sum_z q(z) \log \frac{p(z, x)}{q(z)} &= \sum_z q(z) \log \frac{p(x|z) \cdot p(z)}{q(z)} \\
 &= \sum_z q(z) \log(p(x|z)) + \sum_z q(z) \log \frac{p(z)}{q(z)} \\
 &= E_{q_\phi(z|x)} \log(p_\theta(x|z)) - KLD(q(z) || p(z)) \\
 &= -MSE - KLD
 \end{aligned}$$

## Normal AutoEncoder

### Why does this exist?

- Learn a hidden representation of input (that “vector”)
- Can NOT generate new data

### What does this optimize?

- Minimize reconstruction loss

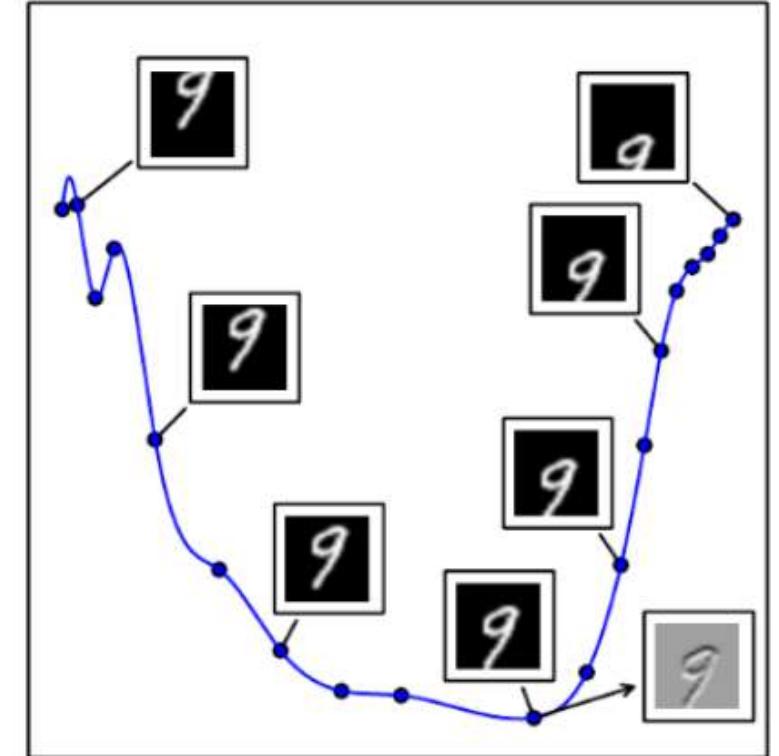
## Variational AutoEncoder

- Learns to generate new data

- Minimize reconstruction loss + latent loss
- Latent vectors are sampled from Gaussian Mixture

# Learning Manifolds with Autoencoders

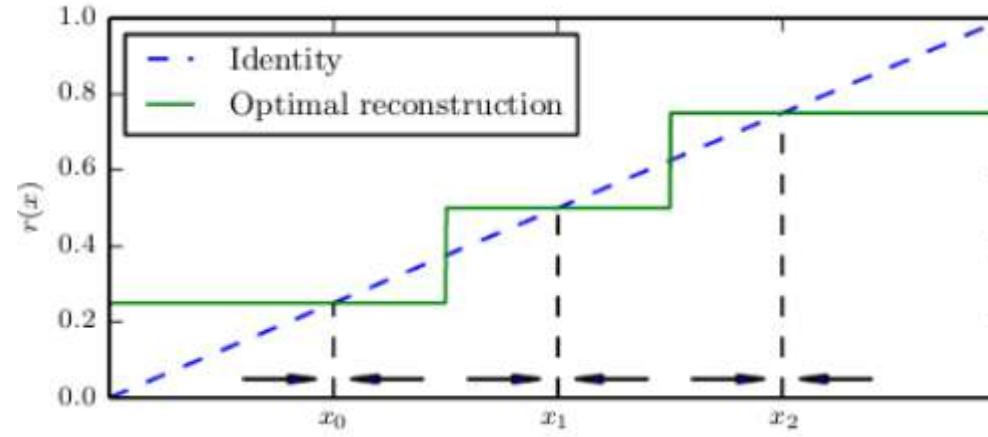
- The process of modeling the manifold on which training instances lie is called Manifold Learning
- Manifold learning is an approach that assumes that data lies on a manifold of a much lower dimension.
- These manifolds can be linear or non-linear.
- Thus, the area tries to project the data from high-dimension space to a low dimension
- Manifolds are hidden structures in the data
- Its tangent planes specify how one can change infinitesimally while staying on the manifold
- Regularization penalty and reconstruction loss with respect to small  $h$  both play role in manifold learning
- Variations tangent to the manifold around correspond to changes. Hence the encoder learns a mapping (from the input to representation space) that is only sensitive to changes along the manifold directions, but that is insensitive to changes orthogonal to the manifold.



# Learning Manifolds with Autoencoders

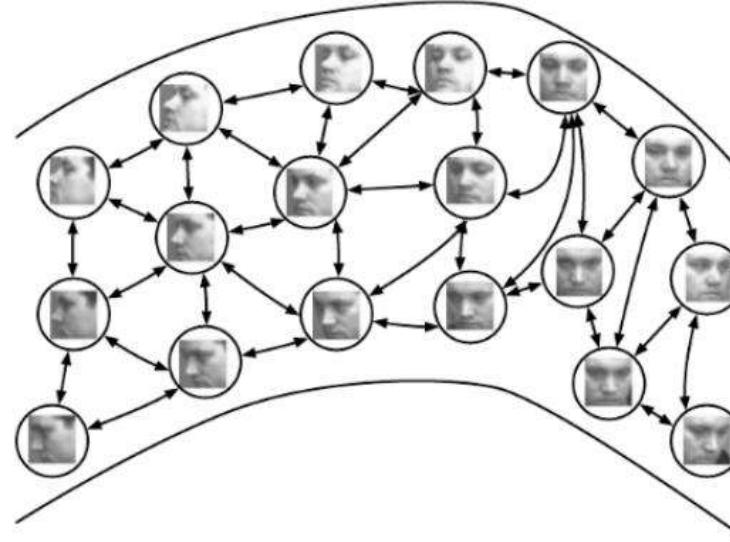


By making the reconstruction function insensitive to perturbations of the input around the data points, we cause the autoencoder to recover the manifold structure.



Difficulty arises if the manifolds are not very smooth, one may need a very large number of training examples to cover each one of these variations

# Learning Manifolds with Autoencoders

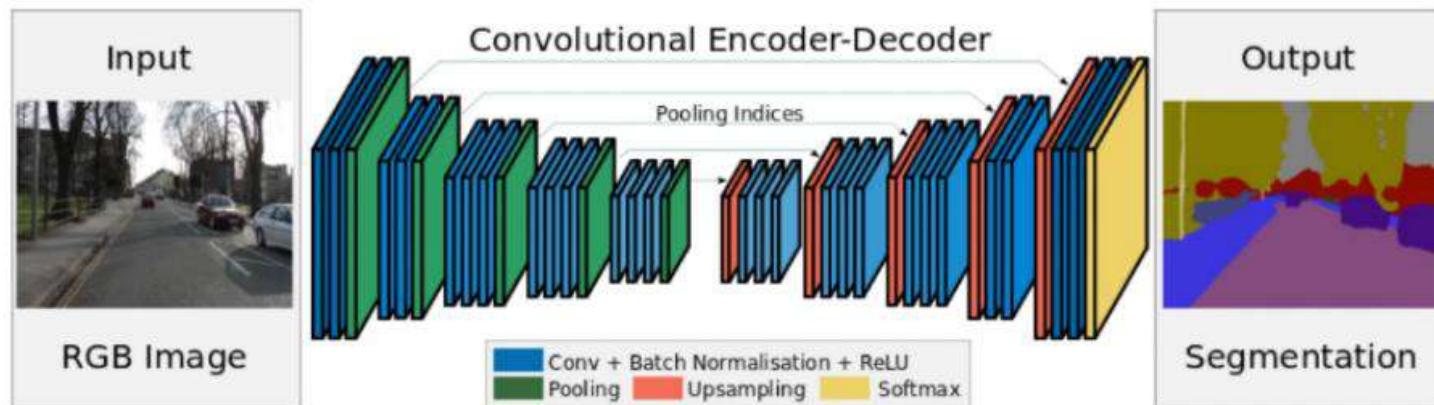


Nonparametric manifold learning could build a nearest neighbor graph. Various procedures can thus obtain the tangent plane associated with a neighborhood of the graph as well as a coordinate system that associates each training example.

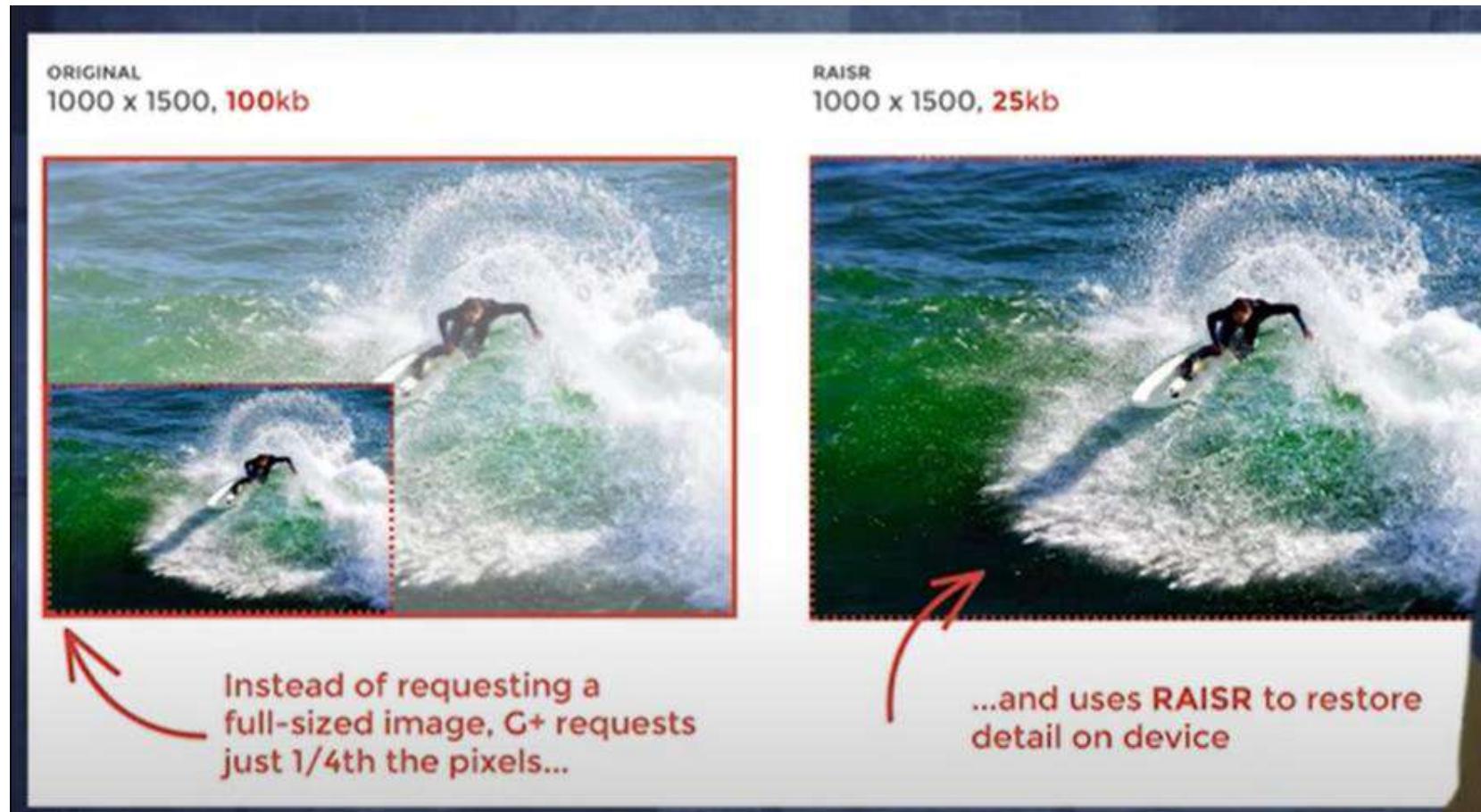
# Applications of Autoencoders



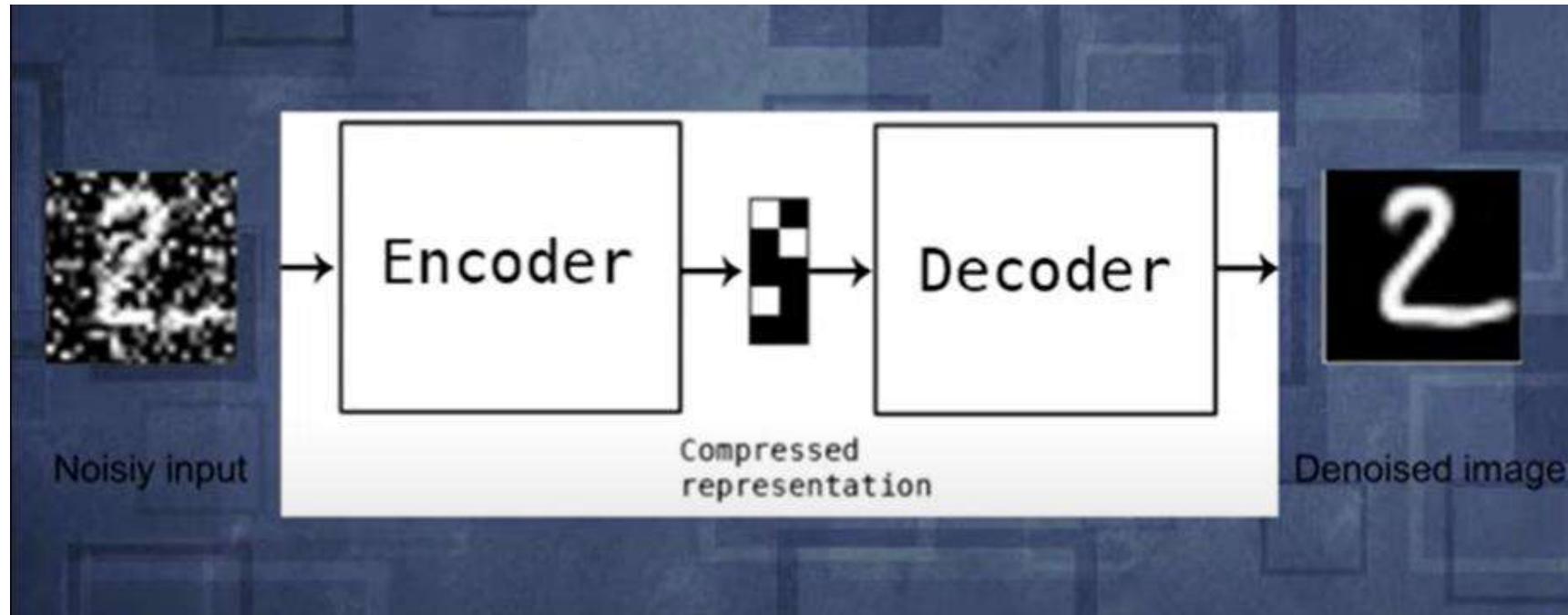
- Dimensionality reduction (representation learning)
- Information retrieval/semantic hashing tasks - we can store all database entries in a hash table that maps binary code vectors to entries
- Classification
- Useful for segmentation and deep-feature



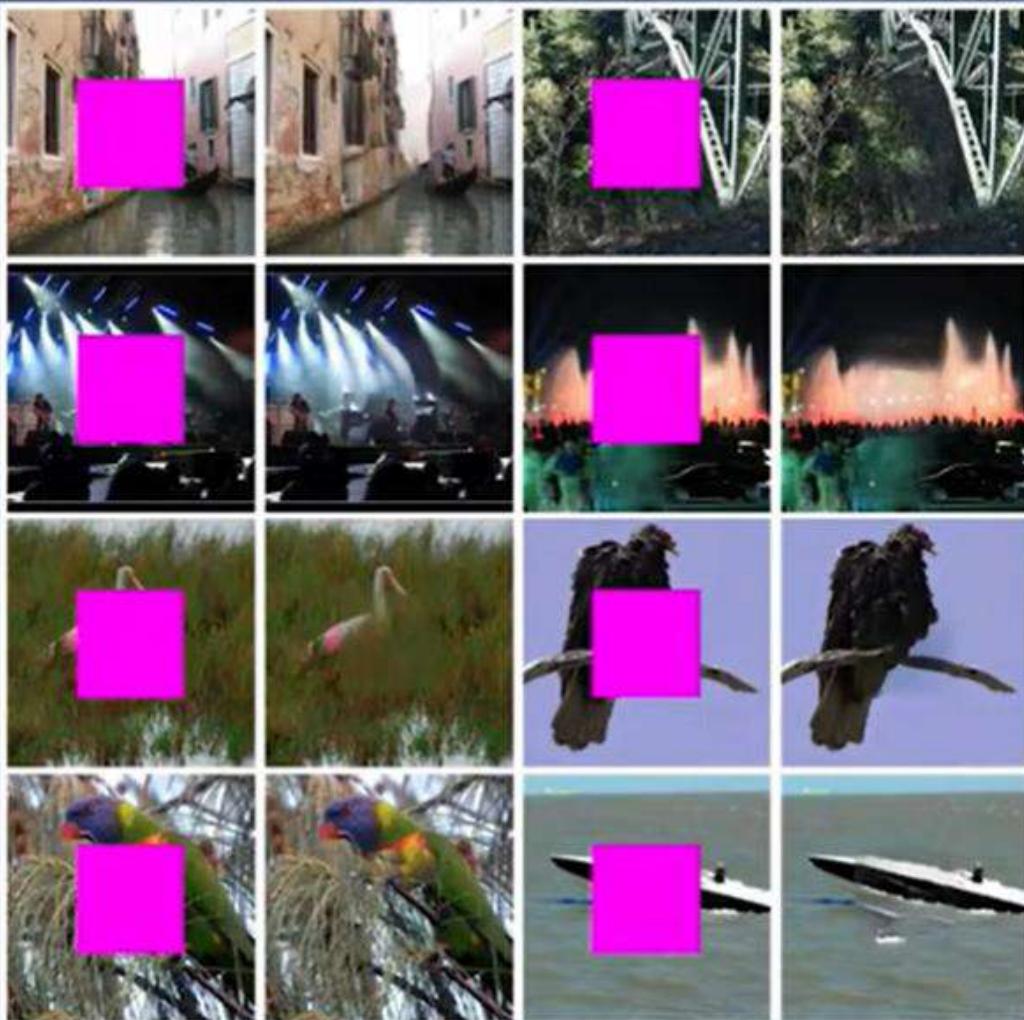
# Dimensionality reduction (representation learning)



# Denoising Encoders



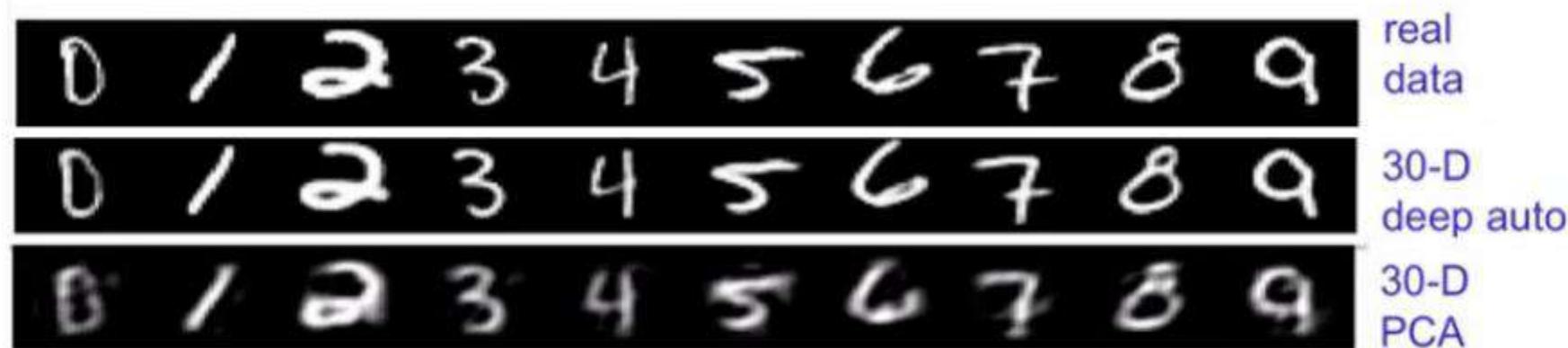
# Neural Inpainting



(source: Github -- Faster-High-Res-Neural-Inpainting)

# Autoencoder example-1

Compressing digit images to 30 numbers.



MNIST1 digit images  $28 \times 28$  three hidden layers (weights were transpose)

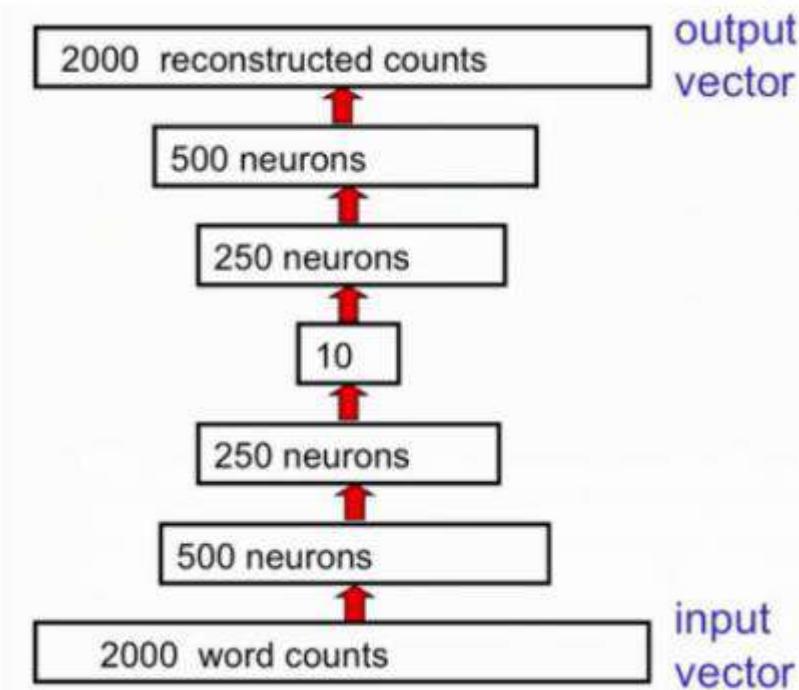
$784 \rightarrow 1000 \rightarrow 500 \rightarrow 250 \rightarrow 30$

It is a supervised learning method to do unsupervised learning

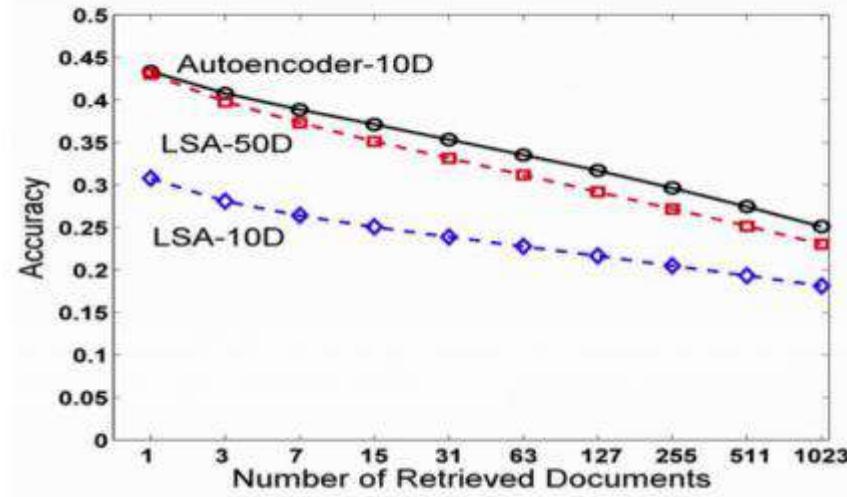
# Autoencoder example-2

## Compare documents for similarity

- Bag of words
- Word count is normalized for probability
- Compressed to 10 denominational
- Softmax is used at output
- 400K business documents
- Hand labeled for ground truth categories
- cosine similarity



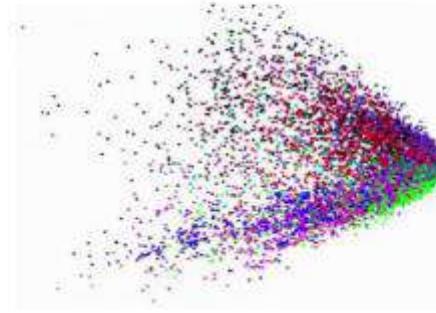
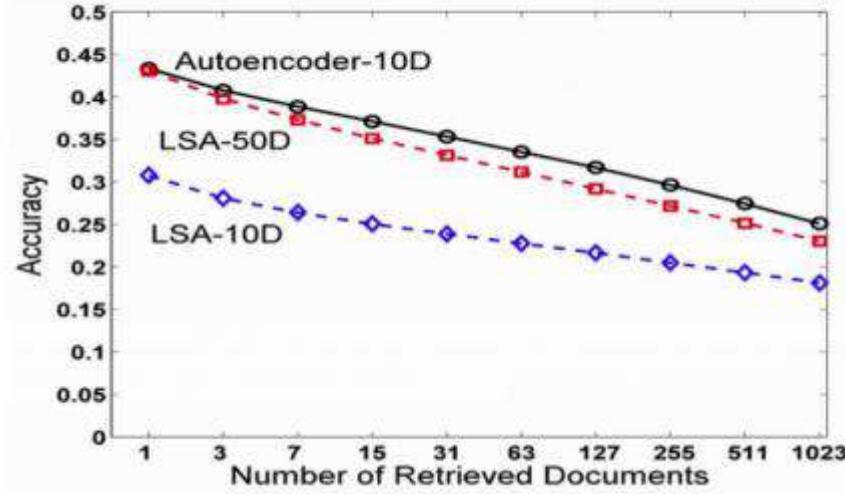
# Autoencoder example-2



Linear semantic analysis is much worse

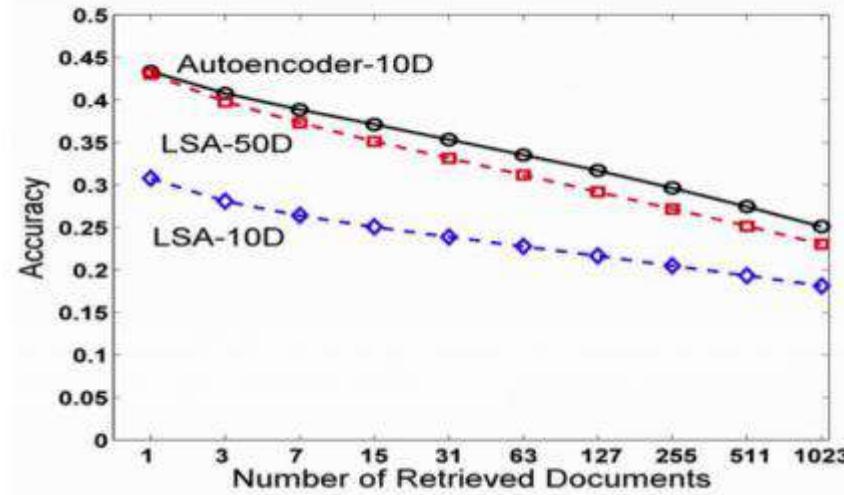
# Autoencoder example-2

Reduce to 2 real numbers using PCA with  $\log(1 + \text{count})$



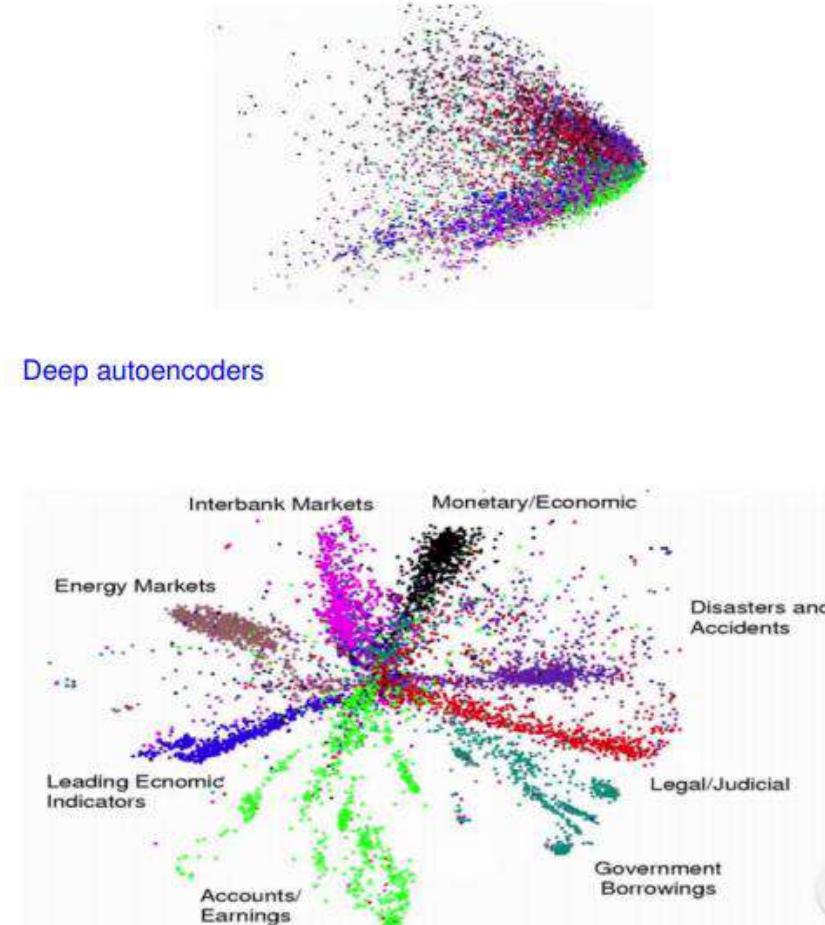
Linear semantic analysis is much worse

# Autoencoder example-2



Linear semantic analysis is much worse

Reduce to 2 real numbers using PCA with  $\log(1 + \text{count})$



# References

<https://www.youtube.com/watch?v=fcvYpzHmhvA>

NPTEL

<https://www.youtube.com/watch?v=wPz3MPI5jvY>

<https://www.youtube.com/watch?v=vy8q-WnHa9A&t=108s>

<https://www.youtube.com/watch?v=5WoltGTWV54>

<https://www.youtube.com/watch?v=9zKuYvjFFS8>

<http://web.stanford.edu/class/cs294a/>

[https://onlinecourses.nptel.ac.in/noc19\\_cs85/preview](https://onlinecourses.nptel.ac.in/noc19_cs85/preview)

[https://compneuro.neuromatch.io/tutorials/Bonus\\_Autoencoders/student/Bonus\\_Tutorial1.html](https://compneuro.neuromatch.io/tutorials/Bonus_Autoencoders/student/Bonus_Tutorial1.html)



# C6: ANN and Deep Learning



**BITS Pilani**  
Hyderabad Campus

Dr. Chetana Gavankar, Ph.D,  
IIT Bombay-Monash University Australia  
[Chetana.gavankar@pilani.bits-pilani.ac.in](mailto:Chetana.gavankar@pilani.bits-pilani.ac.in)



**Session 6  
Date – 4<sup>th</sup> August 2024**

**Time – 10 am to 12.15pm**

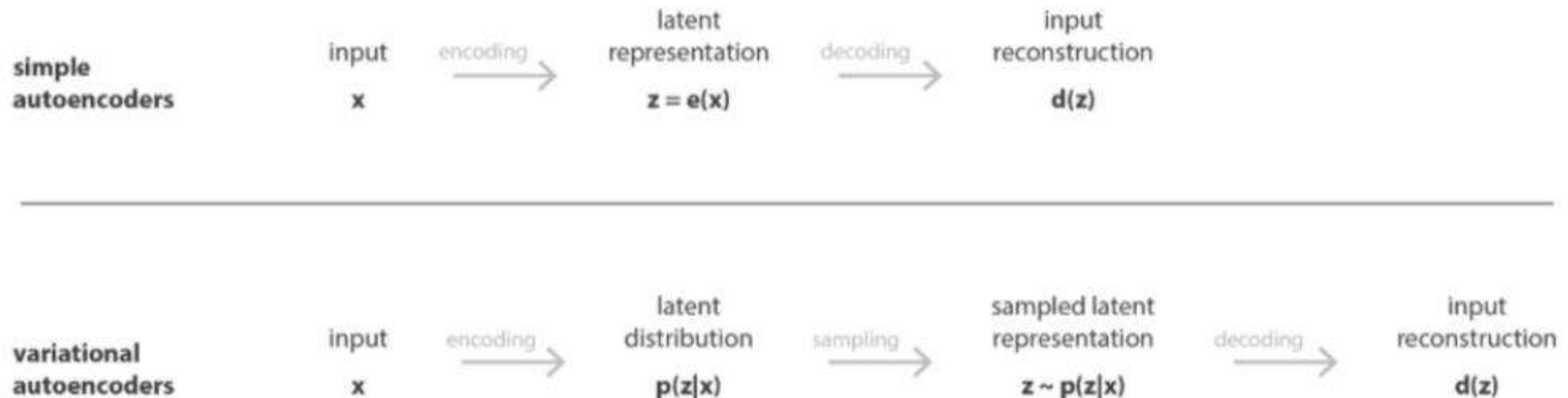
These slides are prepared by the instructor, with grateful acknowledgement of and many others who made their course materials freely available online.

# Overview

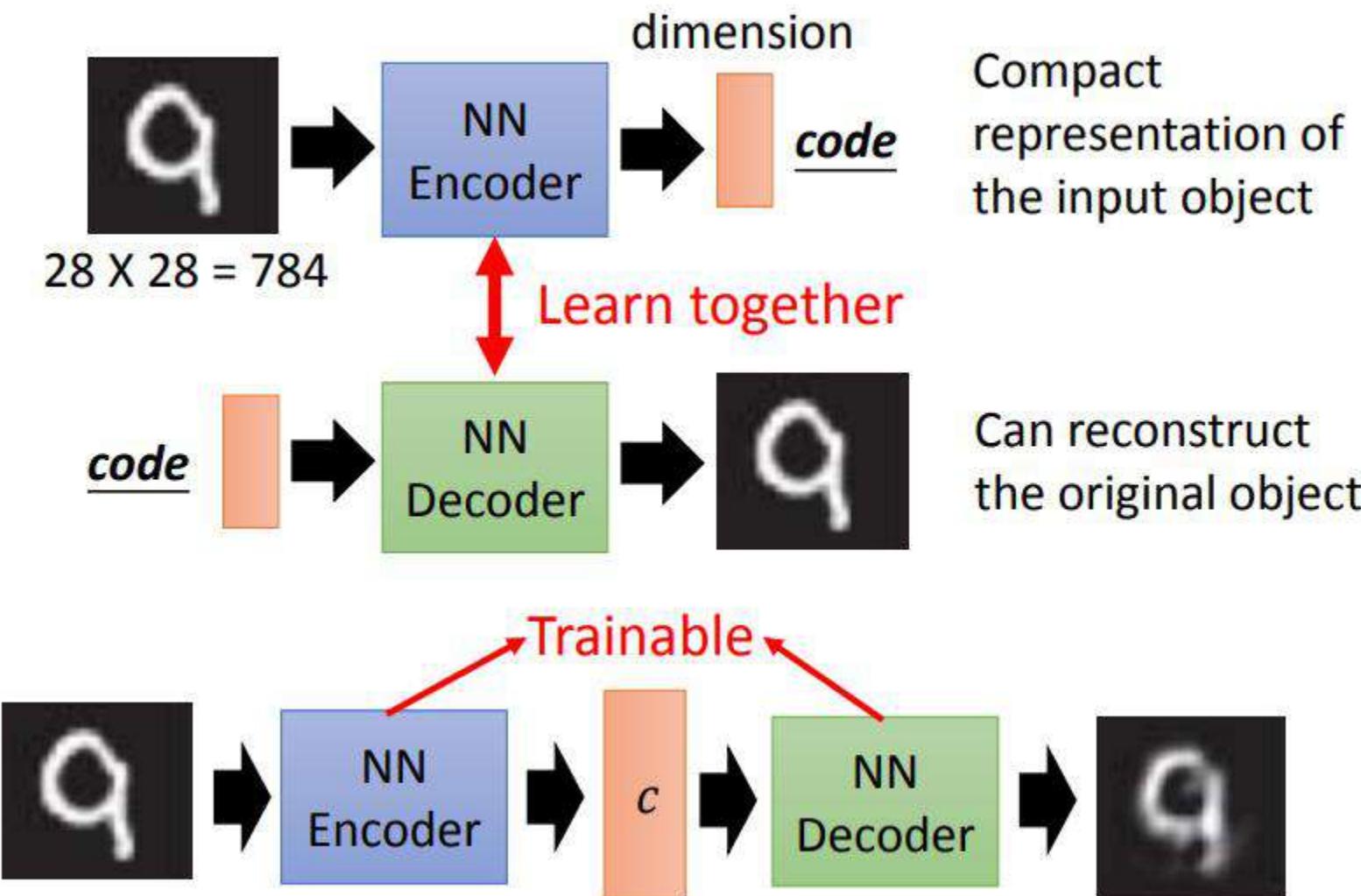
## Generative Models

- Quick recap of VAE
- GAN
- Applications of GAN

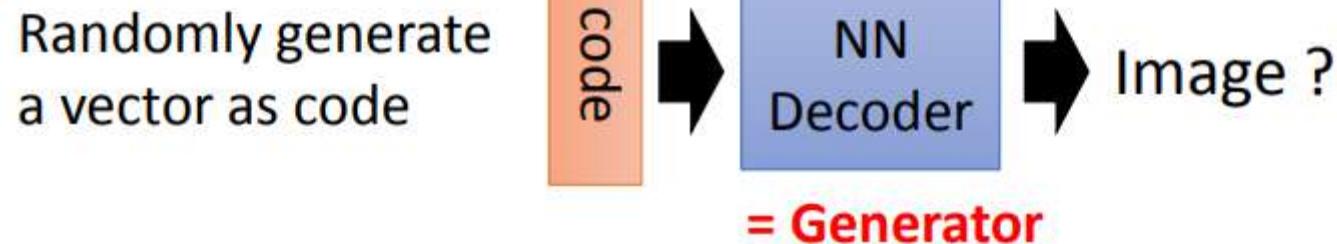
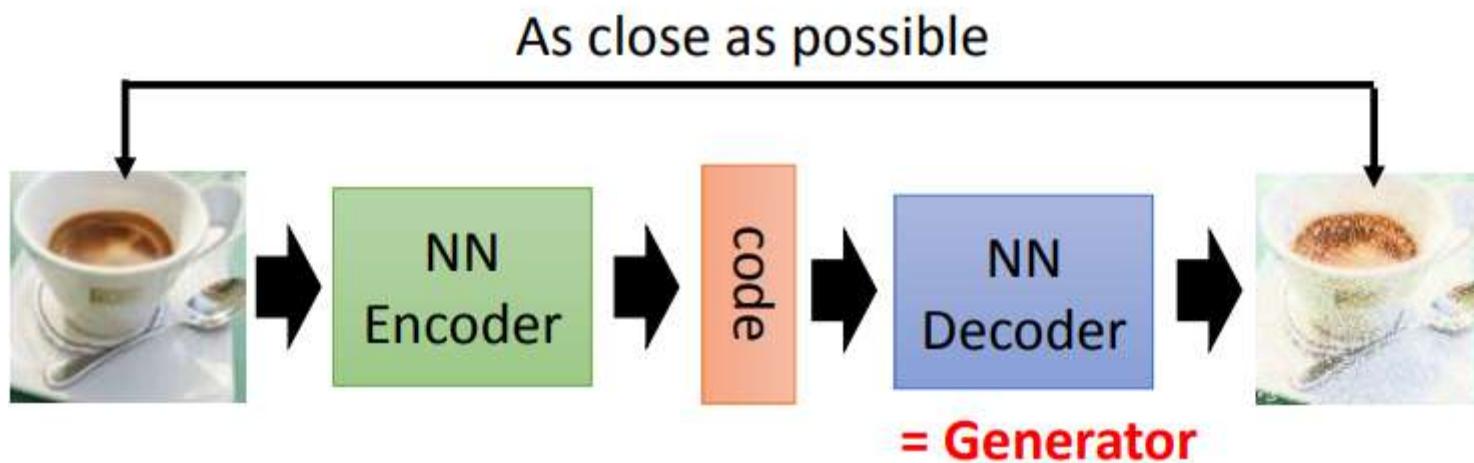
# Usual versus Variational Autoencoders



# Auto Encoders

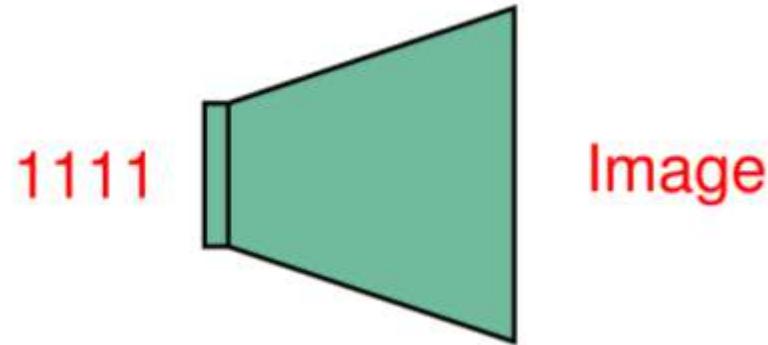


# Auto Encoders



# Generative Modeling

- Consider an upsampling decoder network



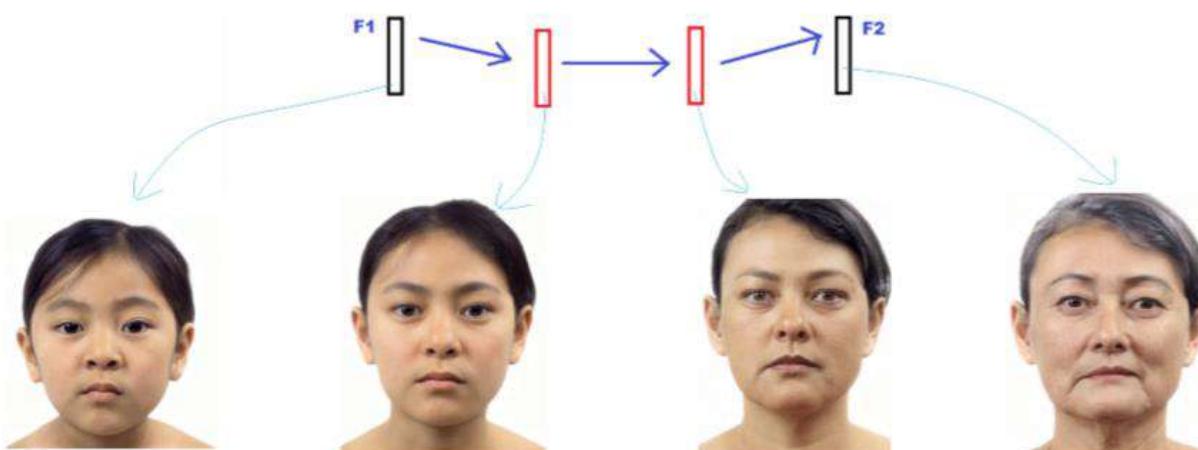
- Let us feed a vector (1111) and try to construct a cat image. Backpropagation would help reduce error and do the same.
- With (0000) try to construct a dog image. Again with the help of backpropagation training, it can be done
- With many classes; encoder can provide what vector to feed
- Autoencoder is concatenation of encoder and decoder that is a unsupervised model (as input is produced as output)

# How can we generate New Images?

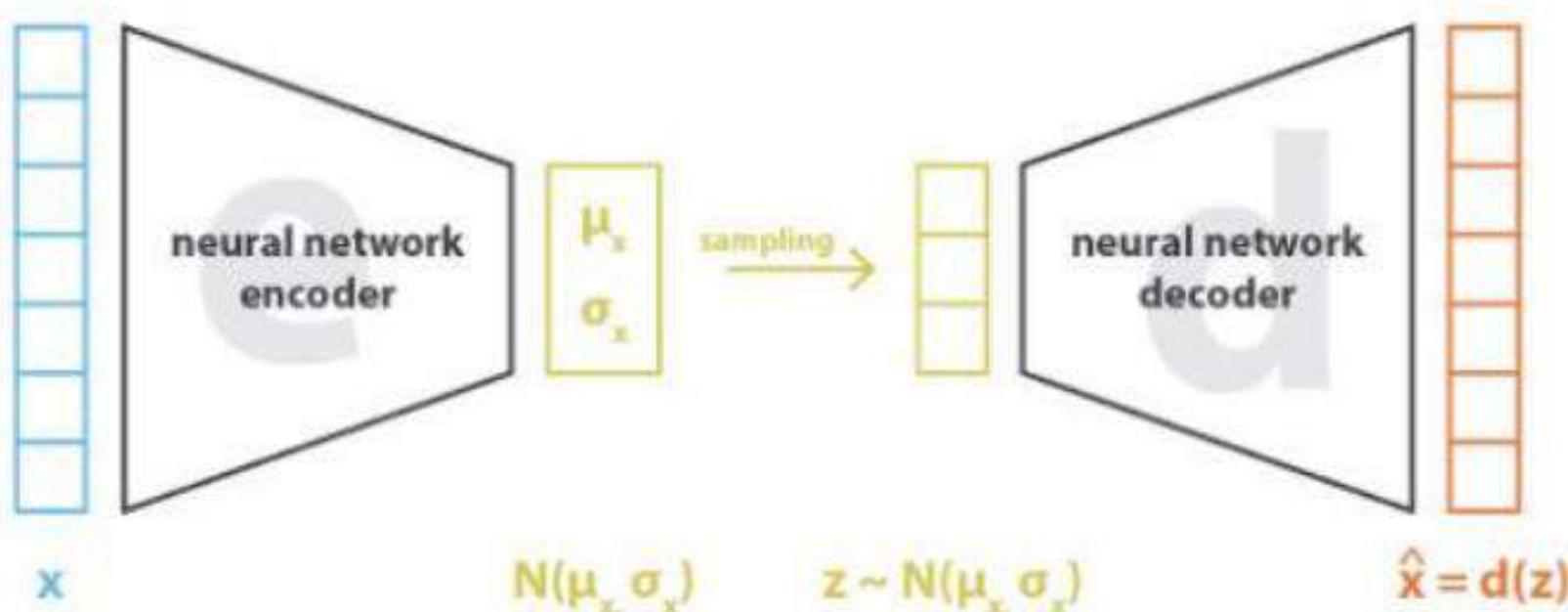
- Train an autoencoder for cat images
- 1. Show a cat image -> 2. get feature from encoder -> 3. add small noise -> 4. feed to decoder -> →5. get output image (of cat) from the decoder 3
- This new cat image may not be in your database

# How can we generate New Images?

- Train an autoencoder for cat images
- 1. Show a cat image -> 2. get feature from encoder -> 3. add small noise -> 4. feed to decoder -> →5. get output image (of cat) from the decoder 3
- This new cat image may not be in your database
- Consider video made by decoding intermediate states of the transformation from one feature to other



# Variational Autoencoder



---

$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

# Variational Autoencoder :After training

Visualizing Reconstructions:



Input



Reconstructions

Using as Generative Model

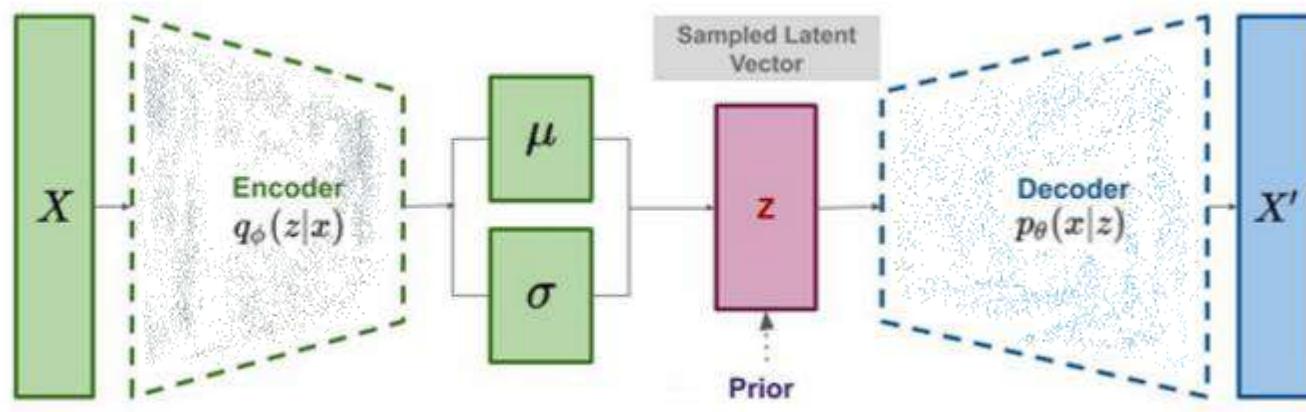
- Sample a random vector from  $N(0, I)$
- Feed forward the vector through the pre-trained Decoder



Generated Samples

# VAE

- Variational Autoencoder is a type of generative model



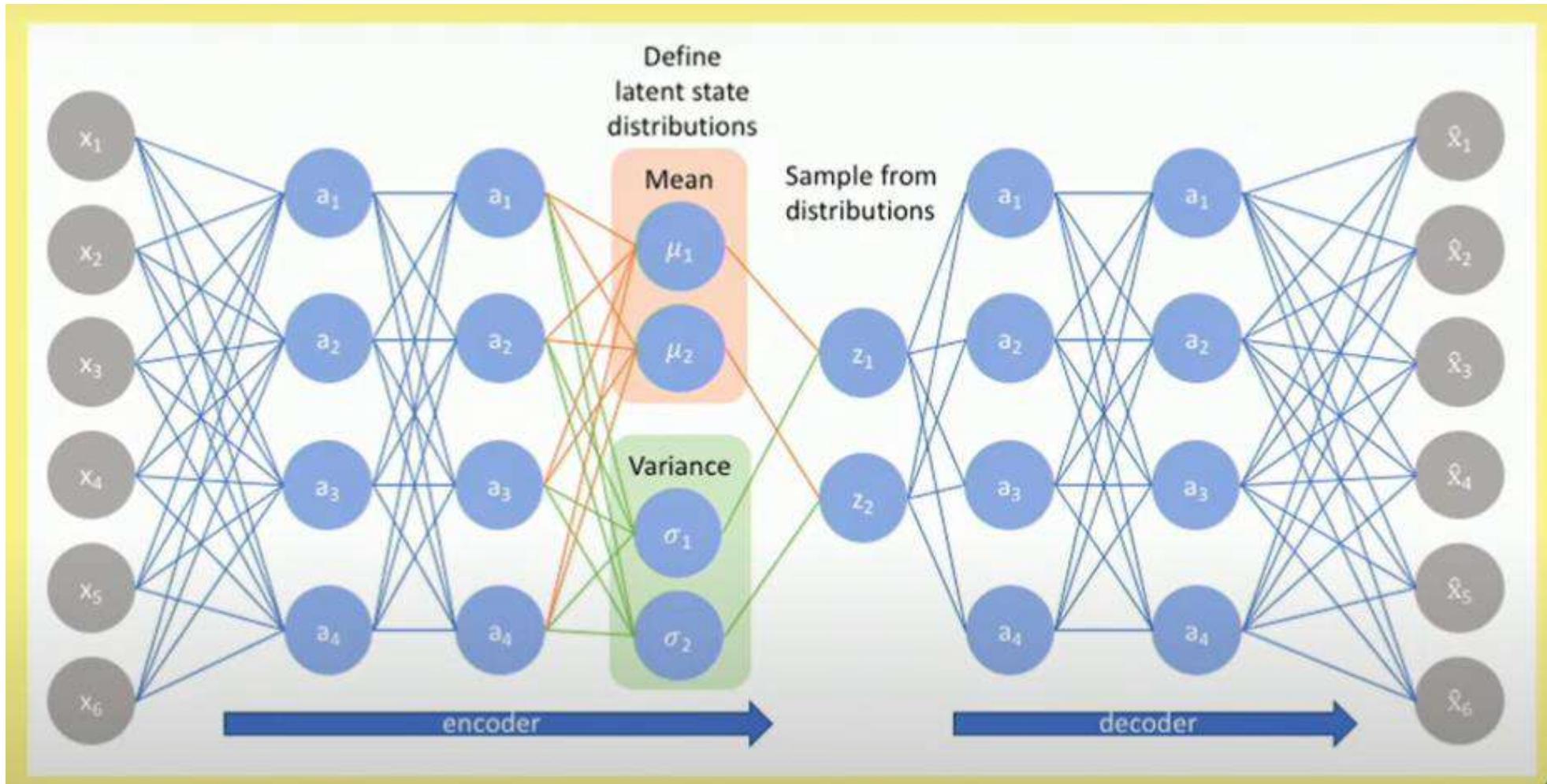
- Force parameters to be of a known type
- Minimize two quantities: reconstruction and latent loss

Reconstruction loss  $\|x - \hat{x}\|$   
LK-Divergence( $G(z_\mu, z_\sigma)$ ,  $N(0, 1)$ )

Kullback-Leibler Divergence measures the difference between two probability distributions

$$D_{KL}(p(x)||q(x)) = \sum_{x \in X} p(x) \cdot \ln \frac{p(x)}{q(x)}$$

# VAE



# Variational Autoencoders

## Variational Autoencoders

Probabilistic spin to traditional autoencoders => allows generating data

Defines an intractable density => derive and optimize a (variational) lower bound

### Pros:

- Principled approach to generative models
- Allows inference of  $q(z|x)$ , can be useful feature representation for other tasks

### Cons:

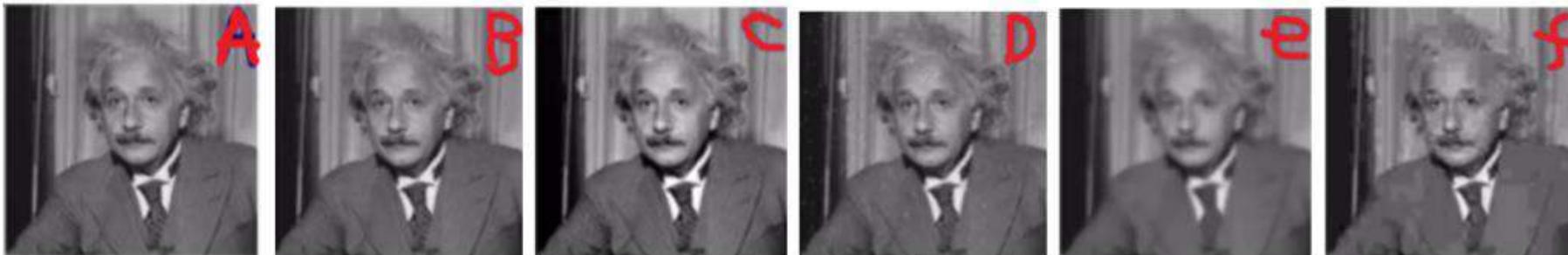
- Maximizes lower bound of likelihood: okay, but
- Samples blurrier and lower quality compared to state-of-the-art (GANs)

### Active areas of research:

- More flexible approximations, e.g. richer approximate posterior instead of diagonal Gaussian
- Incorporating structure in latent variables

# Issues with the Mean Square Error (MSE)/L2

$$L = \frac{1}{2.n.m} \sum_{i=1}^m \sum_{j=1}^n (\hat{x}_{ij} - x_{ij})^2$$



- Hope for A, if gets B or C it is fine. But not D, E, F as they have noise, blur and pixelization. B..F all have same L2 distance with A
- MSE introduces blur because of averaging
- L2 minimization is equivalent to maximization of log likelihood

# Guess the celebrity?



Training Data  
(CelebA)

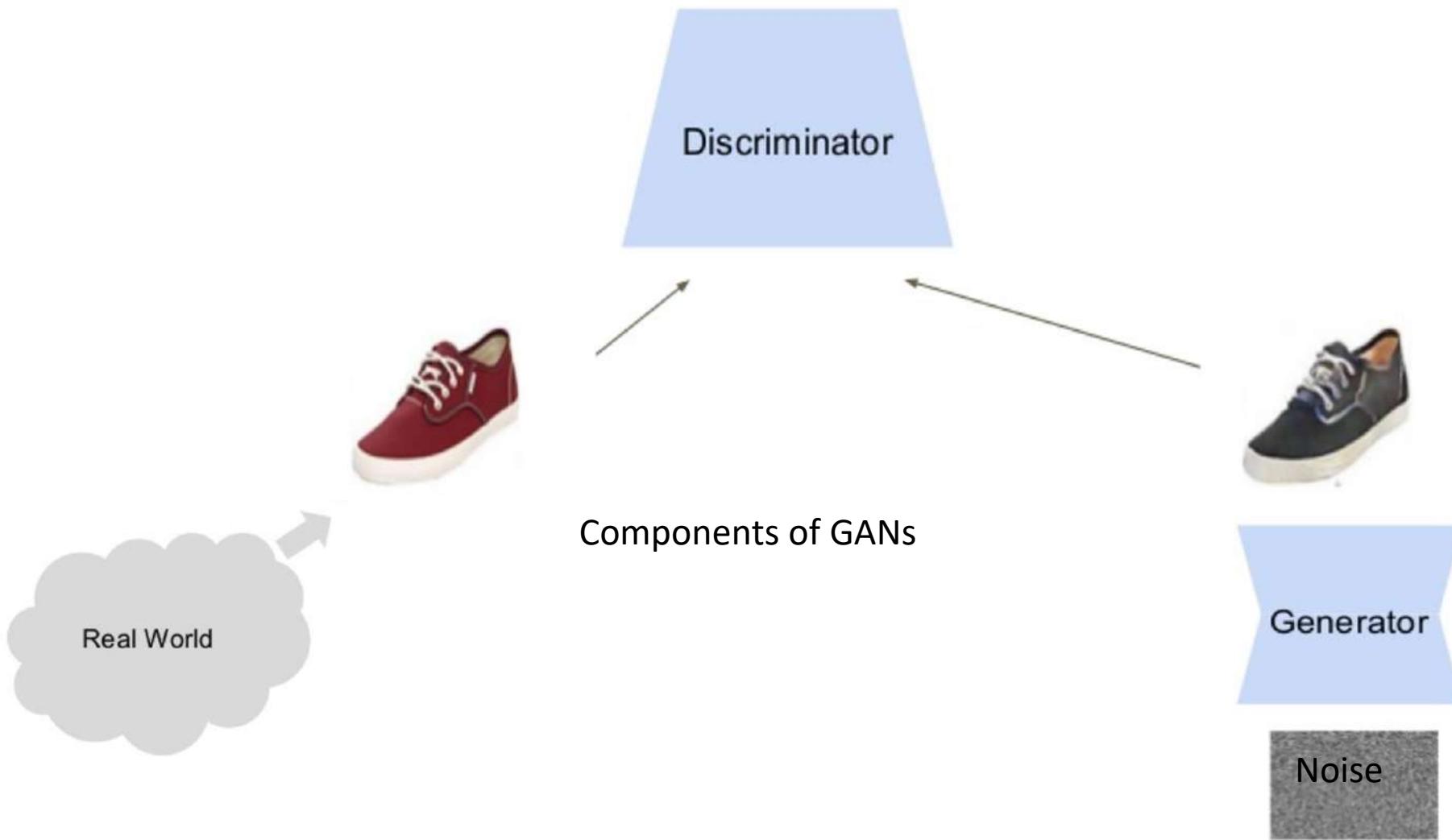


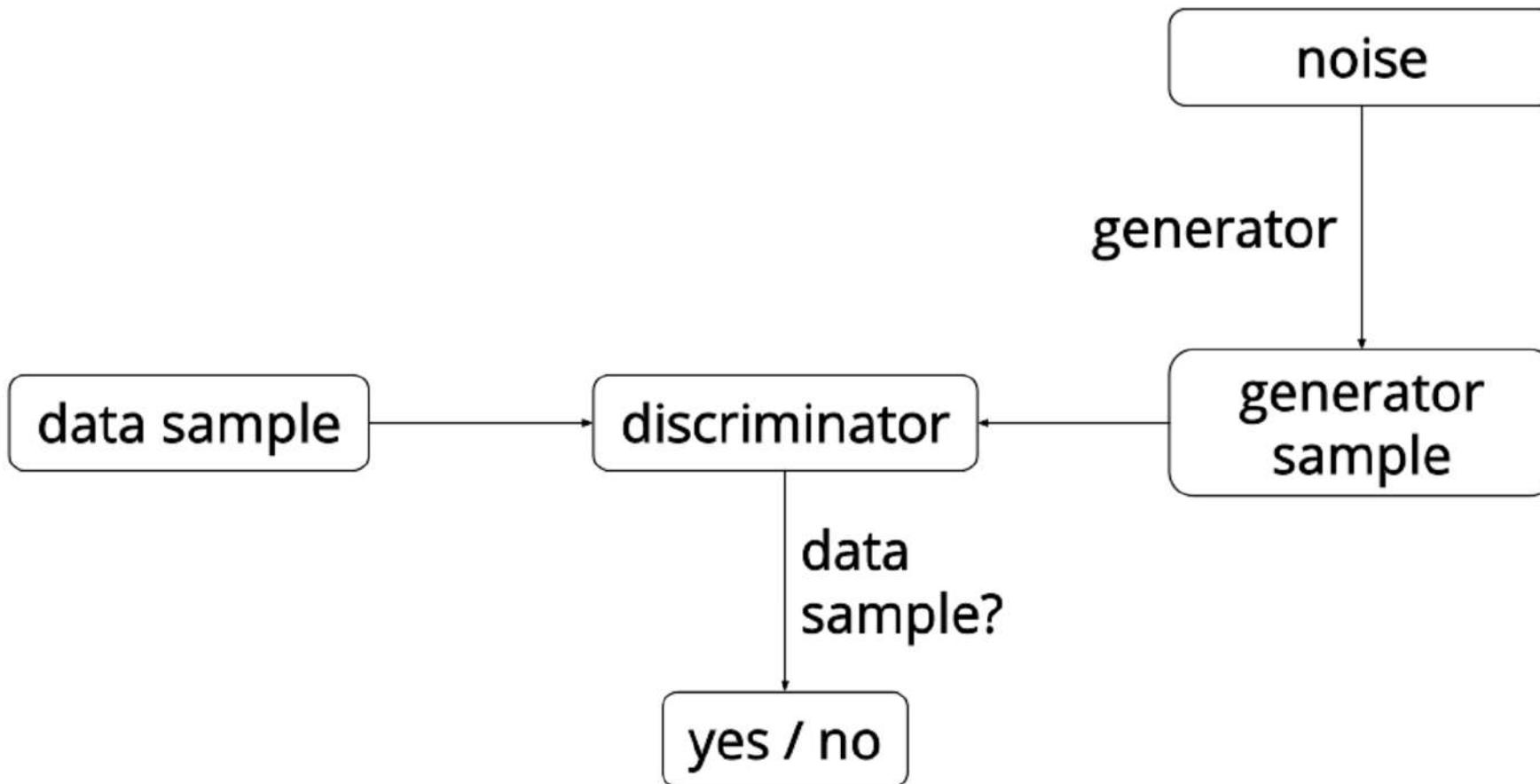
Sample Generator  
(Karras et al, 2017)

# What are GANs?

- System of two neural networks competing against each other in a zero-sum game framework.
- They were first introduced by [Ian Goodfellow](#) *et al.* in 2014.
- Can learn to draw samples from a model that is similar to data that we give them.

## True/False





*Overview of GANs*

Source: <https://ishmaelbelghazi.github.io/ALI>

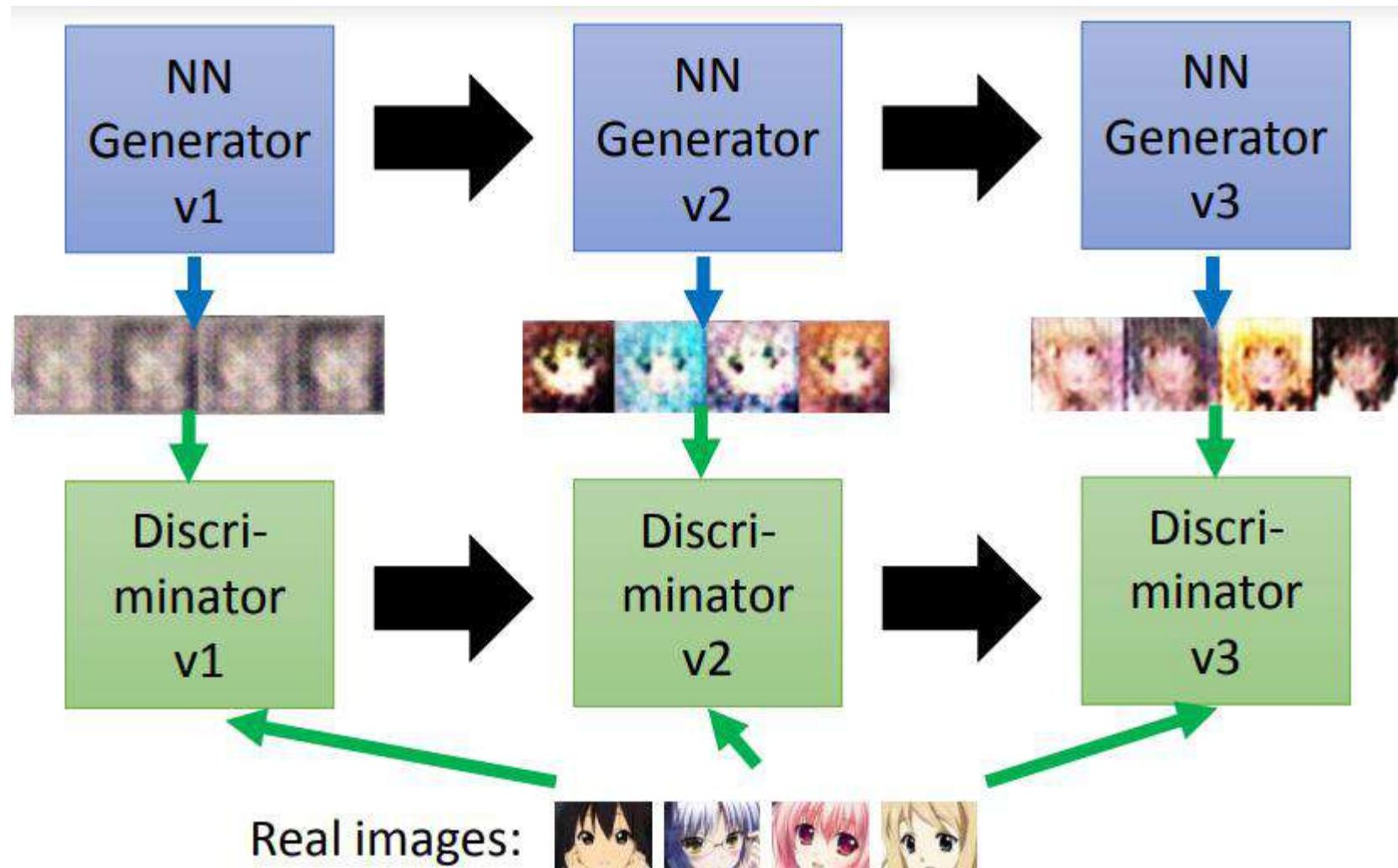
# Discriminative Models

- A **discriminative** model learns a function that maps the input data ( $x$ ) to some desired output class label ( $y$ ).
- In probabilistic terms, they directly learn the conditional distribution  $P(y/x)$ .

# Generative Models

- A **generative** model tries to learn the joint probability of the input data and labels simultaneously i.e.  $P(x,y)$ .
- Potential to understand and explain the underlying structure of the input data even when there are no labels.

# Face Generation using GAN



# Face Generation using GAN



# Face Generation using GAN

1000 updates



# Face Generation using GAN

50,000 updates



# How GANs are being used?

- Applied for modelling natural images.
- Performance is fairly good in comparison to other generative models.
- Useful for unsupervised learning tasks.

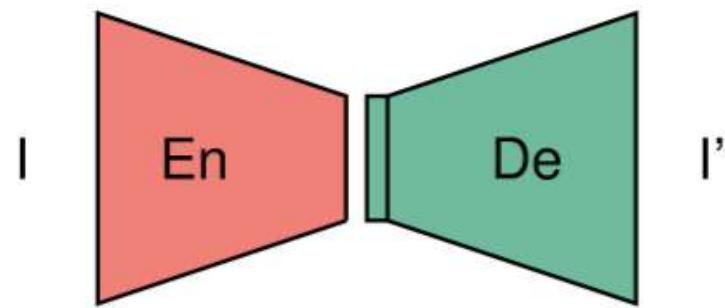
# Why GANs?

- Use a latent code.
- Asymptotically consistent (unlike variational methods) .
- Often regarded as producing the best samples.

# How to train GANs?

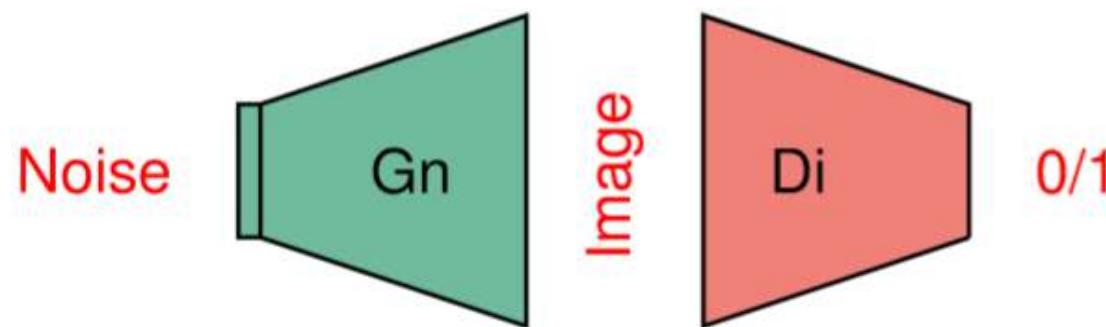
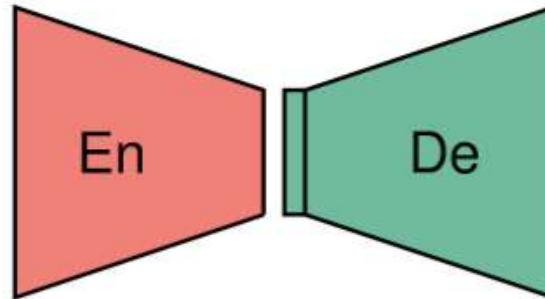
- Objective of generative network - increase the error rate of the discriminative network.
- Objective of discriminative network – decrease binary classification loss.
- Discriminator training - backprop from a binary classification loss.
- Generator training - backprop the negation of the binary classification loss of the discriminator.

# Exchanging encoder and decoder



# Exchanging encoder and decoder

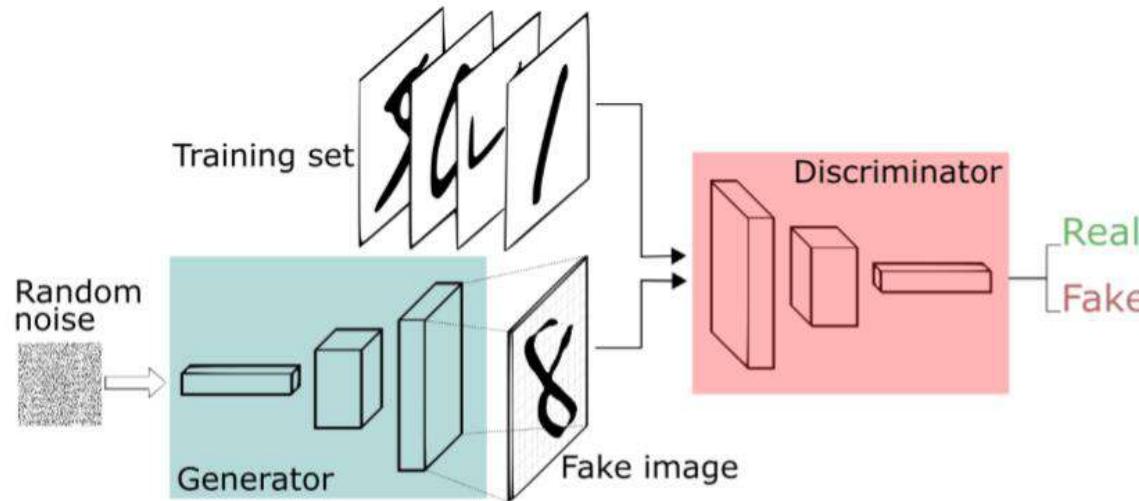
- Feeding something as input to get an image, and then to classify if generated or not



Decoder → Generator and Encoder → Discriminator

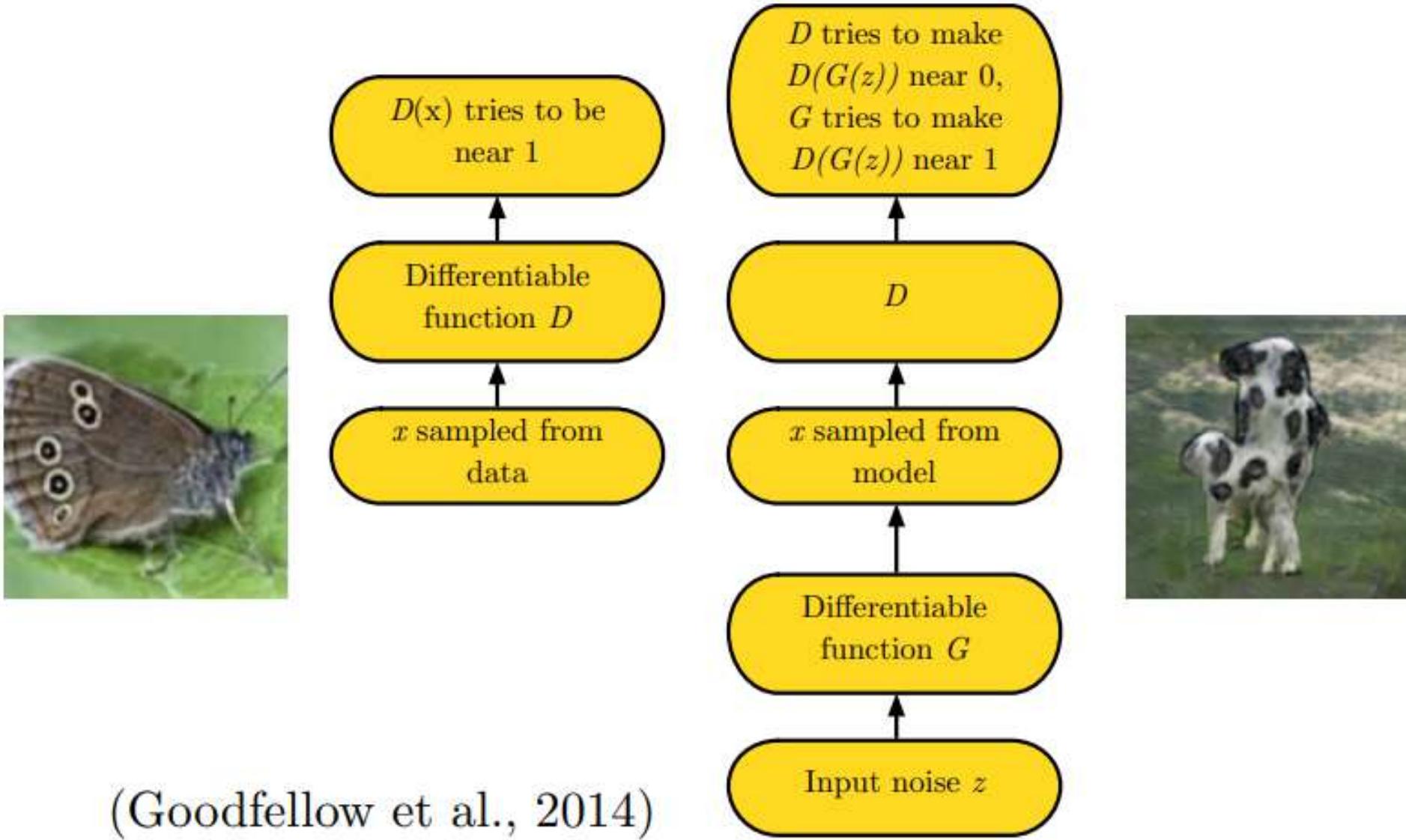
# Generative Adversarial Networks (GAN)

- GAN has two NN, generator and discriminator (wish to fool each other)

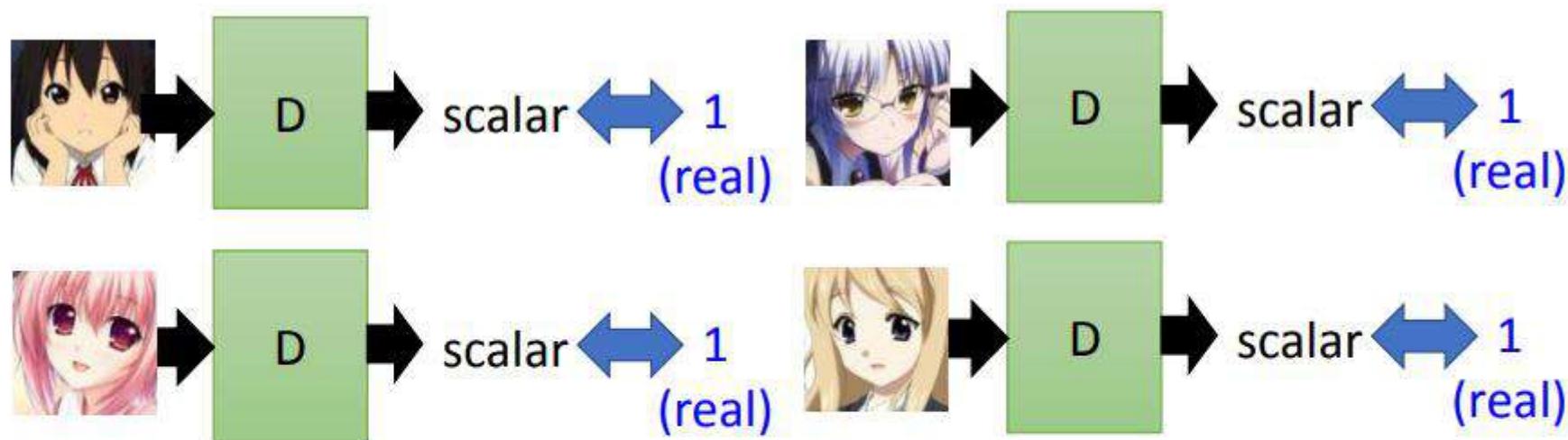


- Do not explicitly model the data distribution but provide samples
- NN uses random noise to map in model distribution
- Worst possible training by min-max game
- Underline distribution plays a role

# Adversarial Nets Framework



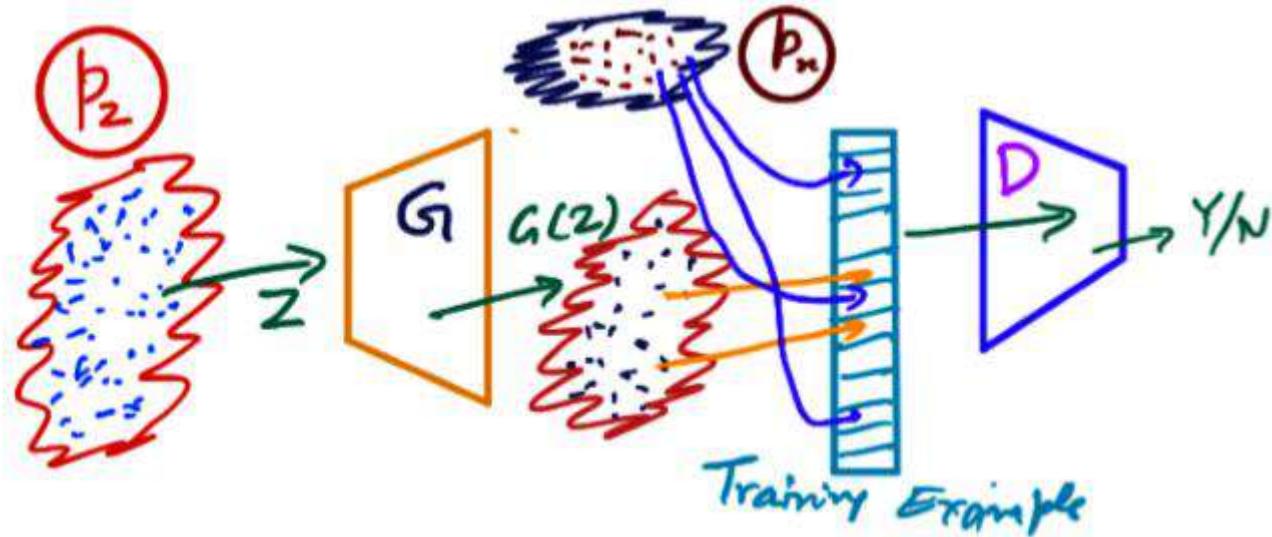
# Discriminator Training



Discriminator only learns to output “1” (real).

Discriminator training needs some negative examples.

# Loss in a GAN



- Cross entropy loss is defined as

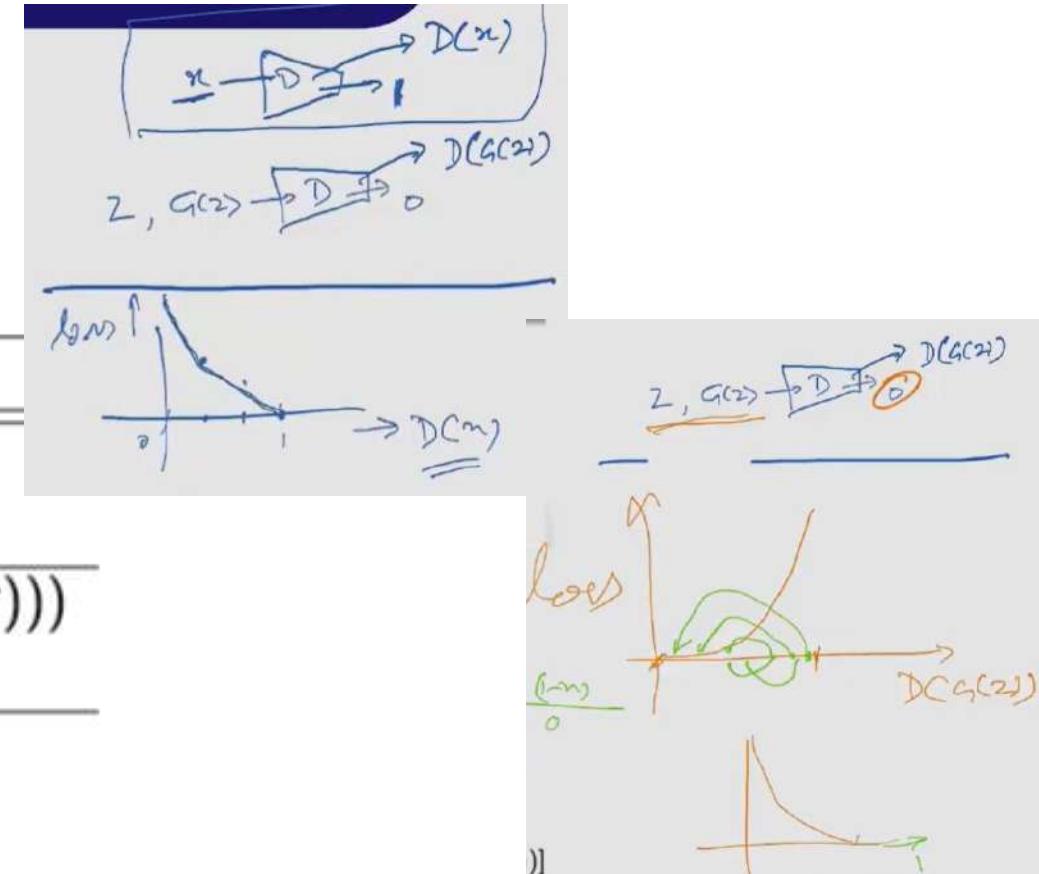
$$L(\hat{y}, y) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

where  $\hat{y}$  is the predicted label and  $y$  is the ground truth

- There is no ground truth for Generator
- Discriminator is pass  $\rightarrow$  Generator is in loss
- Generator is pass  $\rightarrow$  Discriminator is in loss

# Discriminator Loss

Input	Ground-Truth	Predicted	Loss
x	1	$D(x) = 1$	-
	1	$D(x) = 0$	$\log(D(x))$
$G(z)$	0	$D(G(z)) = 1$	$\log(1 - D(G(z)))$
	0	$D(G(z)) = 0$	-



- Objective is to maximize loss of discriminator

$$\text{Max}_D [E_{x \sim p_{\text{data}}(x)} \log(D(x)) + E_{z \sim p_z(z)} \log(1 - D(G(z)))]$$

- And minimize loss of generator

$$\text{Min}_G [E_{x \sim p_{\text{data}}(x)} \log(D(x)) + E_{z \sim p_z(z)} \log(1 - D(G(z)))]$$

- Combined loss function

$$\text{Min}_G \text{Max}_D [E_{x \sim p_{\text{data}}(x)} \log(D(x)) + E_{z \sim p_z(z)} \log(1 - D(G(z)))]$$

$\rightarrow \log D(x) - \log(1 - D(G(z)))$   
 (minimize)  
 maximize  
 $\log D(x) + \log(1 - D(G(z)))$

# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

**Generator network:** try to fool the discriminator by generating real-looking images

**Discriminator network:** try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data } x} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data } G(z)}) \right]$$

- Discriminator ( $\theta_d$ ) wants to **maximize objective** such that  $D(x)$  is close to 1 (real) and  $D(G(z))$  is close to 0 (fake)
- Generator ( $\theta_g$ ) wants to **minimize objective** such that  $D(G(z))$  is close to 1 (discriminator is fooled into thinking generated  $G(z)$  is real)

# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

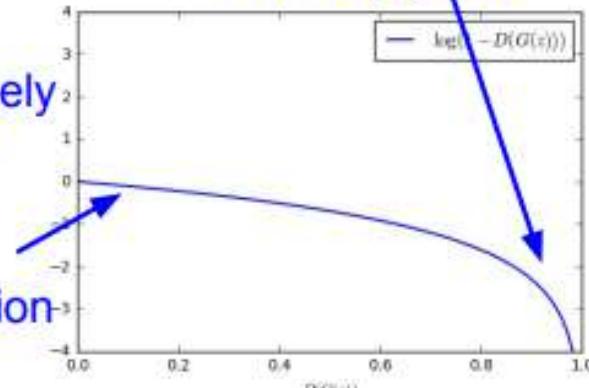
Gradient signal dominated by region where sample is already good

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, optimizing this generator objective does not work well!

When sample is likely fake, want to learn from it to improve generator. But gradient in this region is relatively flat!



# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

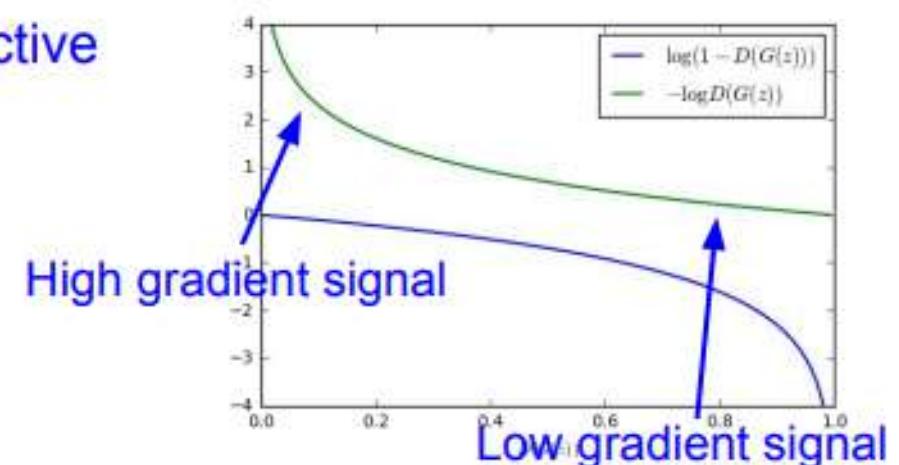
$$\max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. Instead: **Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.

Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.



# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

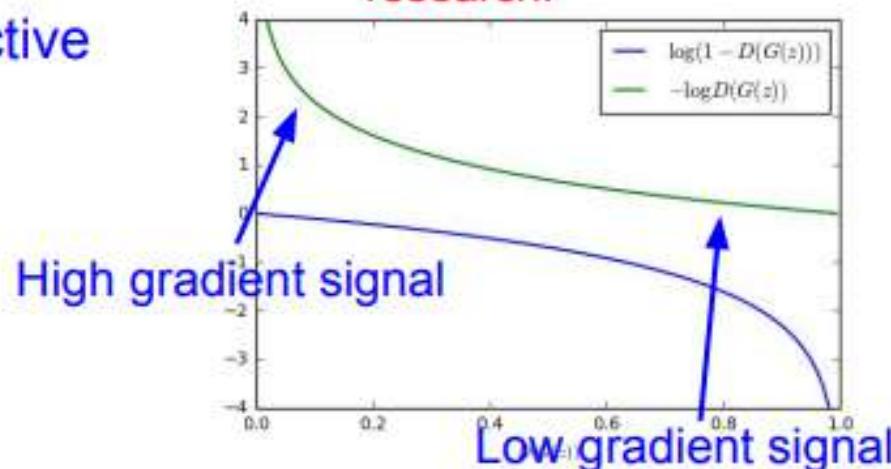
2. Instead: **Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.

Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.

Aside: Jointly training two networks is challenging, can be unstable. Choosing objectives with better loss landscapes helps training, is an active area of research.



# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

## Putting it together: GAN training algorithm

```
for number of training iterations do
    for k steps do
        • Sample minibatch of m noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
        • Sample minibatch of m examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
        • Update the discriminator by ascending its stochastic gradient:
            
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

    end for
    • Sample minibatch of m noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
    • Update the generator by ascending its stochastic gradient (improved objective):
            
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

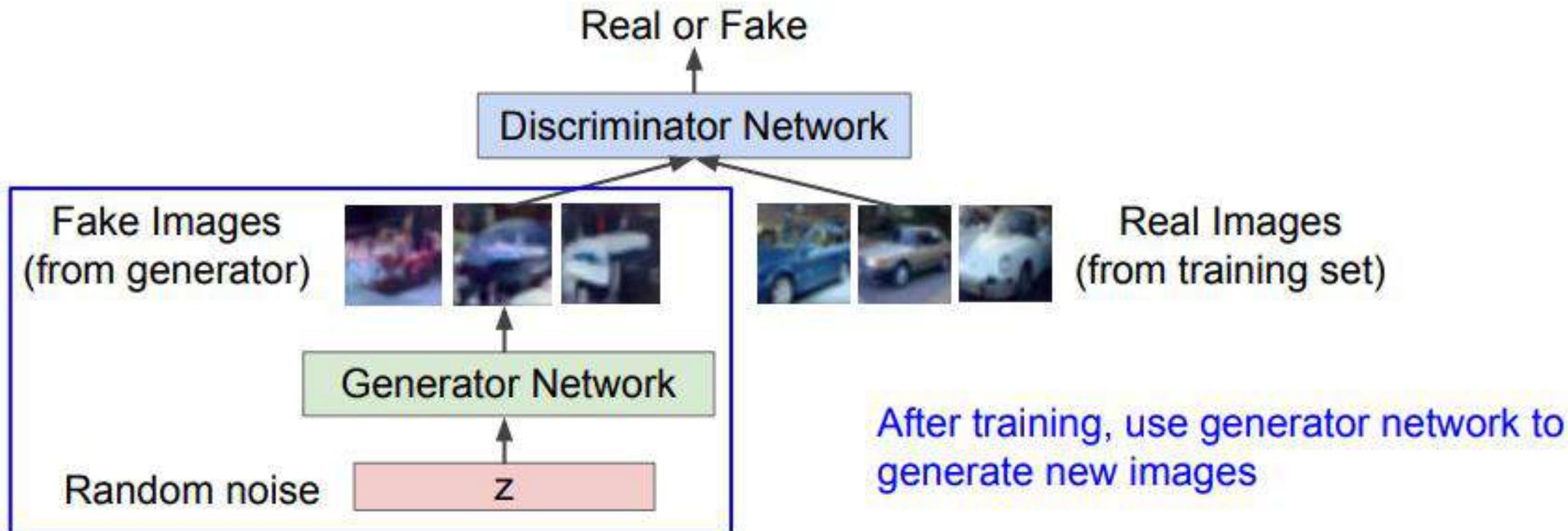
end for
```

# Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

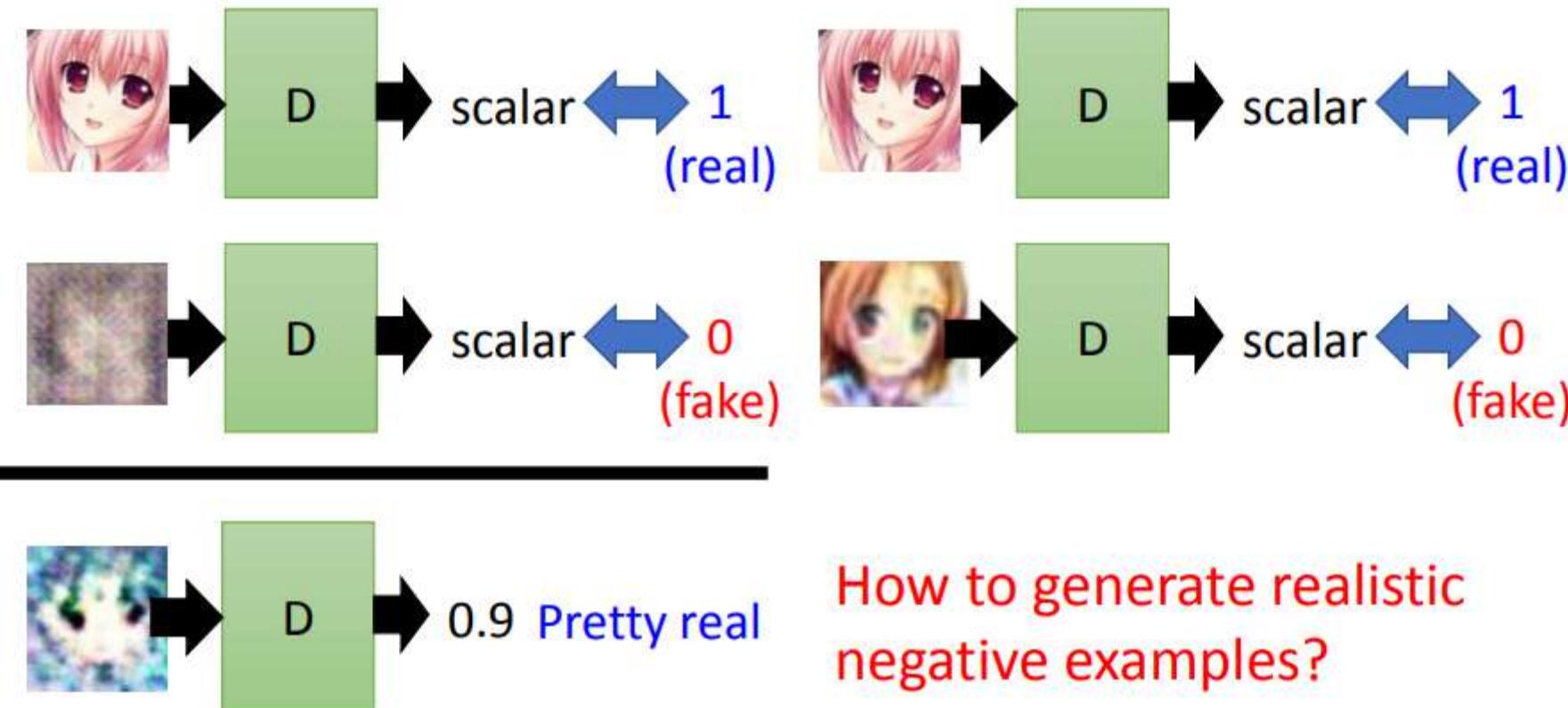
**Generator network:** try to fool the discriminator by generating real-looking images

**Discriminator network:** try to distinguish between real and fake images



# Discriminator Training

- Negative examples are critical.



How to generate realistic  
negative examples?

# Discriminator Training

- **General Algorithm**

- Given a set of **positive examples**, randomly generate a set of **negative examples**.



- In each iteration



- Learn a discriminator D that can discriminate positive and negative examples.



v.s.



- Generate negative examples by discriminator D



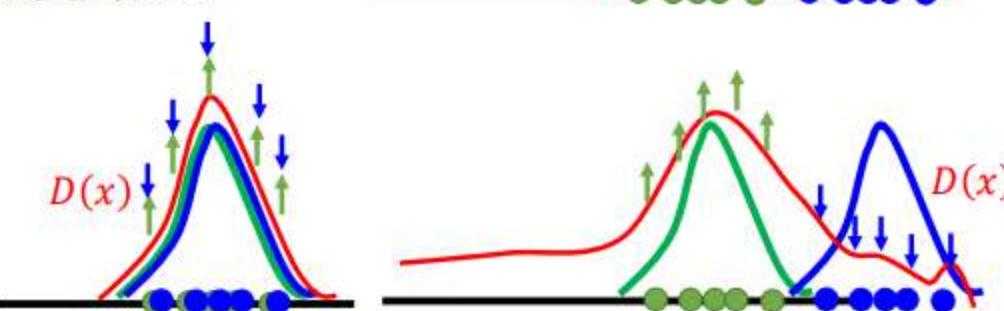
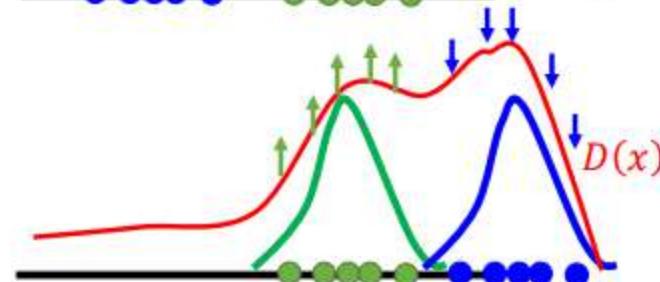
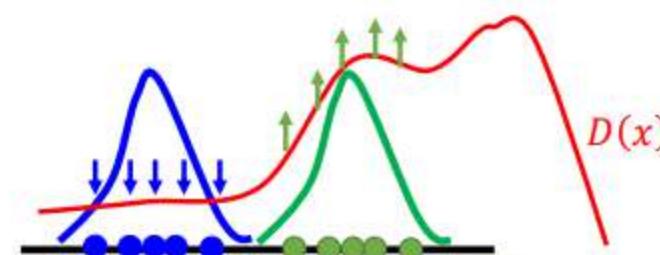
$$\tilde{x} = \arg \max_{x \in X} D(x)$$

# Discriminator Training

Discriminator  
- Training



In the end .....



- General Algorithm



- Given a set of **positive examples**, randomly generate a set of **negative examples**.



- In each iteration

- Learn a discriminator D that can discriminate positive and negative examples.



v.s.



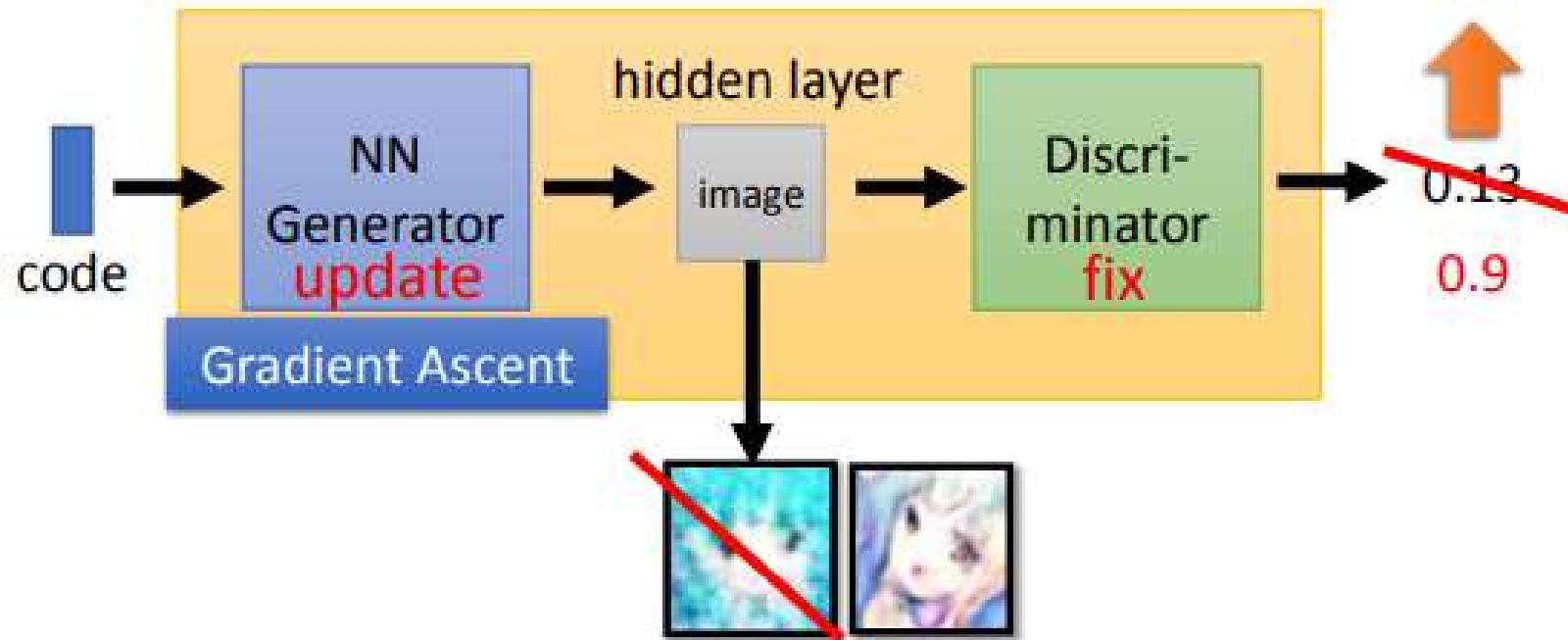
- Generate negative examples by discriminator D

$$G \rightarrow \tilde{x}$$

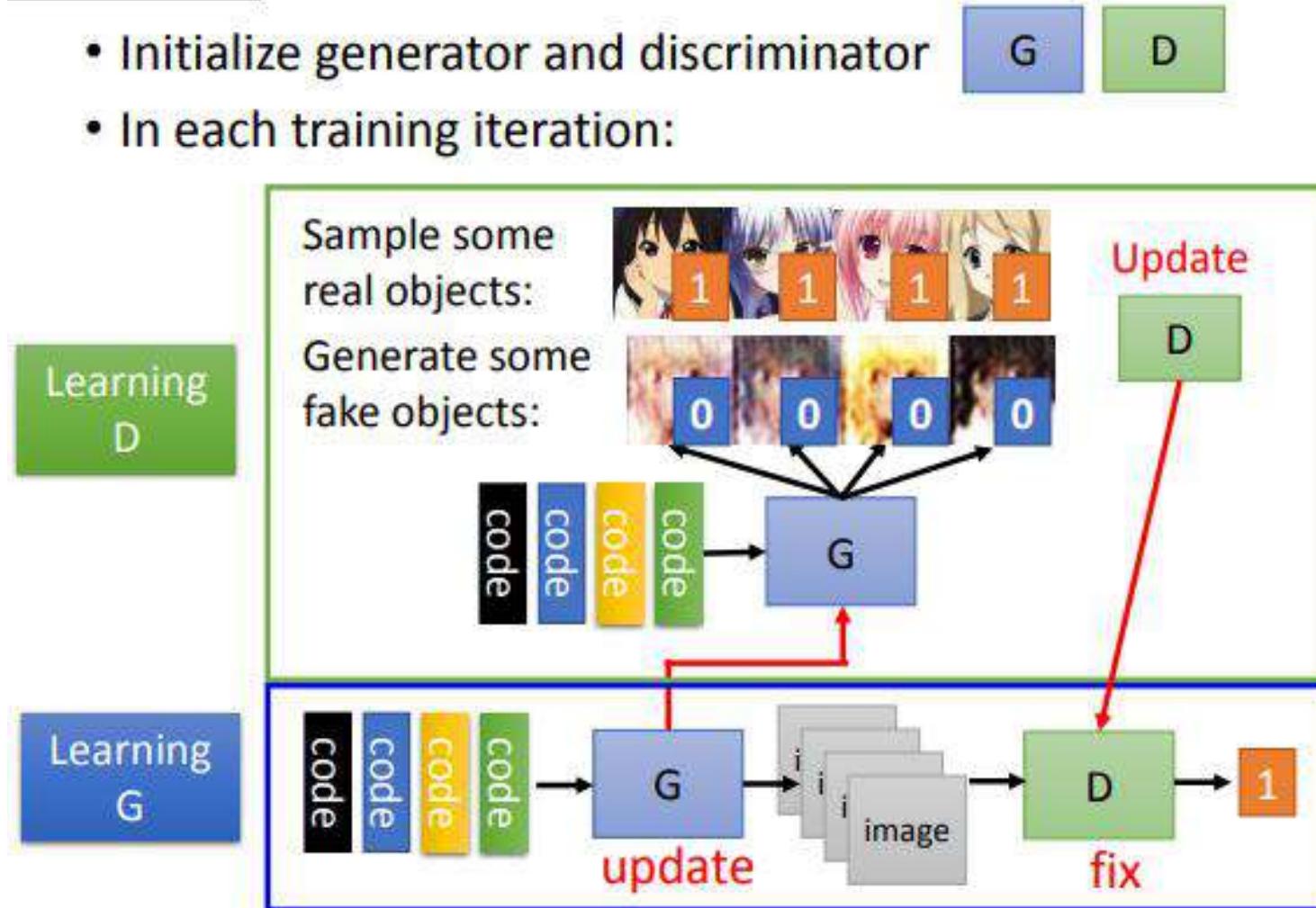
$$\tilde{x} = \arg \max_{x \in X} D(x)$$

# Generating Negative Examples

$$G \rightarrow \tilde{x} = \arg \max_{x \in X} D(x)$$



- Initialize generator and discriminator
- In each training iteration:



# Variations to GANs

- Several new concepts built on top of GANs have been introduced –
  - InfoGAN – Approximate the data distribution and learn interpretable, useful vector representations of data.
  - Conditional GANs - Able to generate samples taking into account external information (class label, text, another image). Force  $G$  to generate a particular type of output.

# Generative Adversarial Nets: Convolutional Architectures

Generator is an upsampling network with fractionally-strided convolutions

Discriminator is a convolutional network

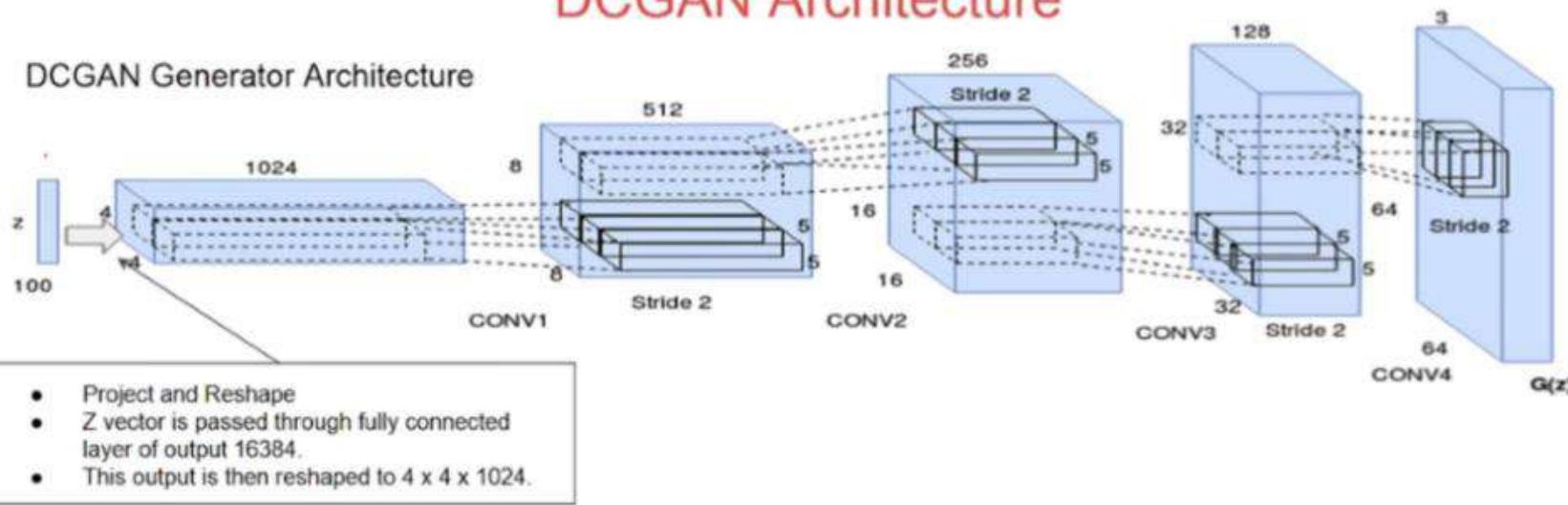
Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

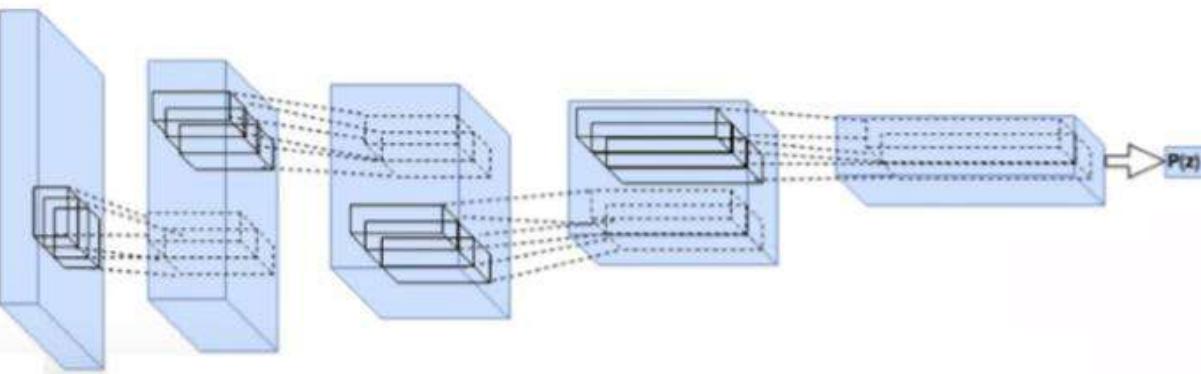
# DCGAN

## DCGAN Architecture

DCGAN Generator Architecture



DCGAN Discriminator  
Architecture





Generated bedrooms. Source: “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” <https://arxiv.org/abs/1511.06434v2>



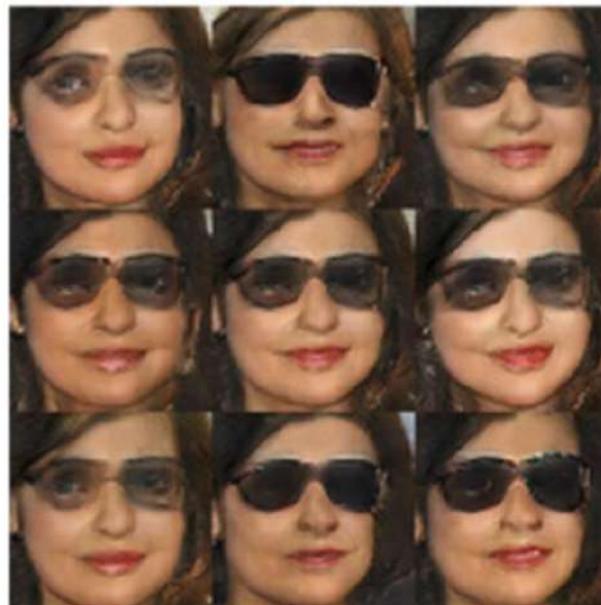
man  
with glasses



man  
without glasses



woman  
without glasses



woman with glasses

# GAN Applications

- DC GAN: Deep Conv. GAN



# GAN Applications



# Text to Image, by conditional GAN

Caption	Image
a pitcher is about to throw the ball to the batter	
a group of people on skis stand in the snow	
a man in a wet suit riding a surfboard on a wave	

# Text to Image - Results

"red flower with  
black center"

From CY Lee lecture

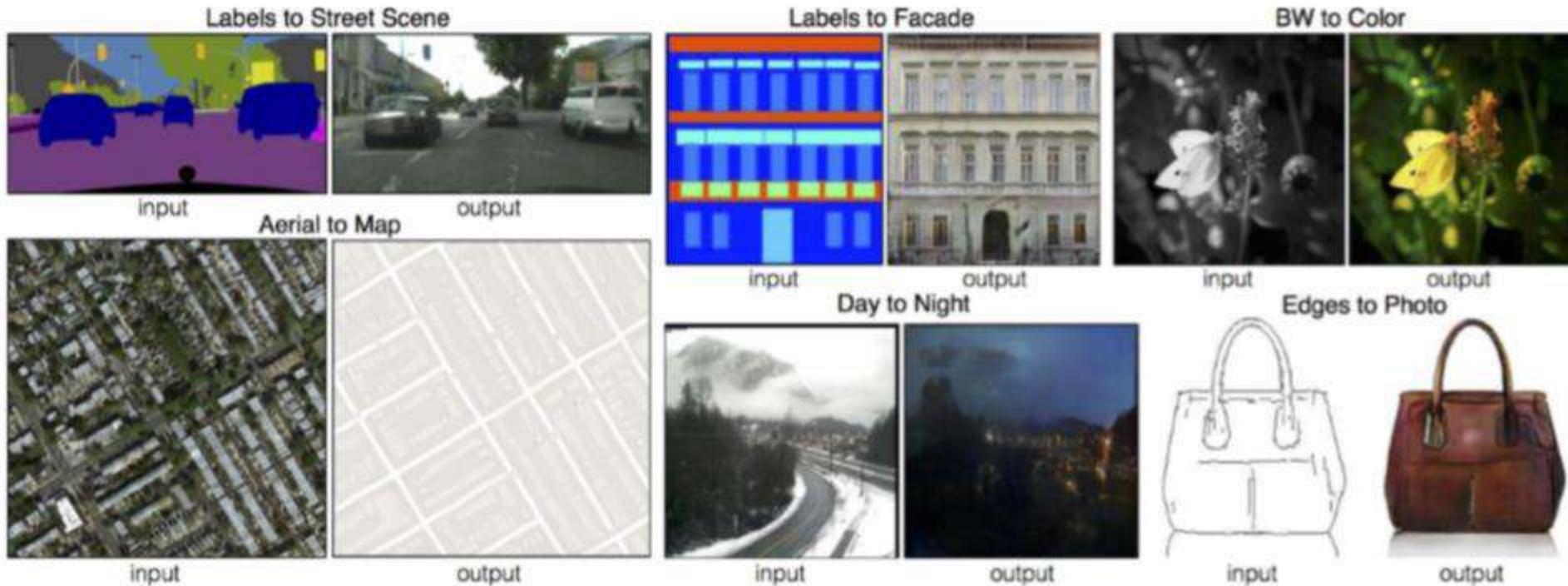


Caption	Image
this flower has white petals and a yellow stamen	
the center is yellow surrounded by wavy dark purple petals	
this flower has lots of small round pink petals	 <small>One image in the bottom row contains the word "rabbit".</small>

Project topic: Code and data are all on web, many possibilities!

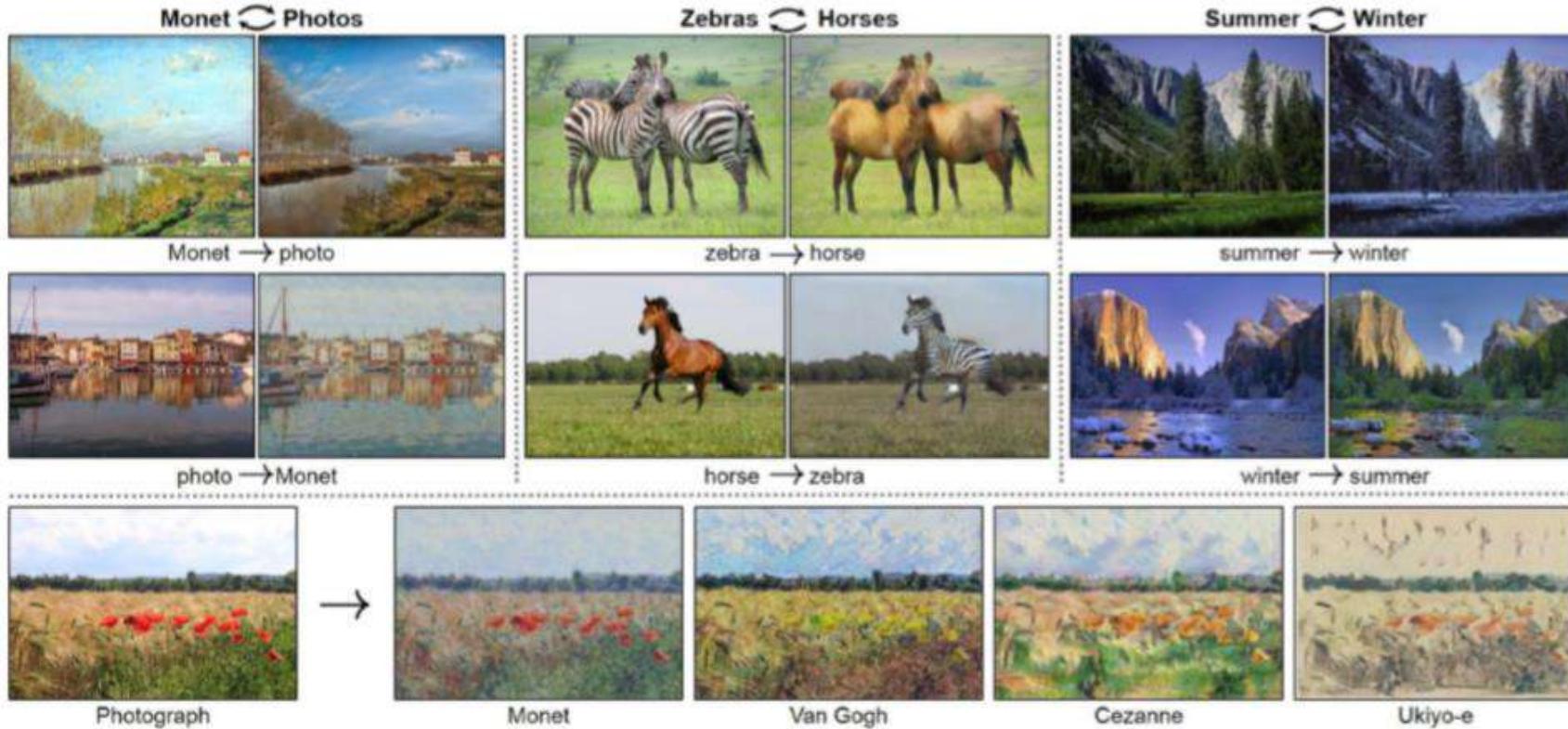
# GAN Applications

## Pix2Pix



# GAN Applications

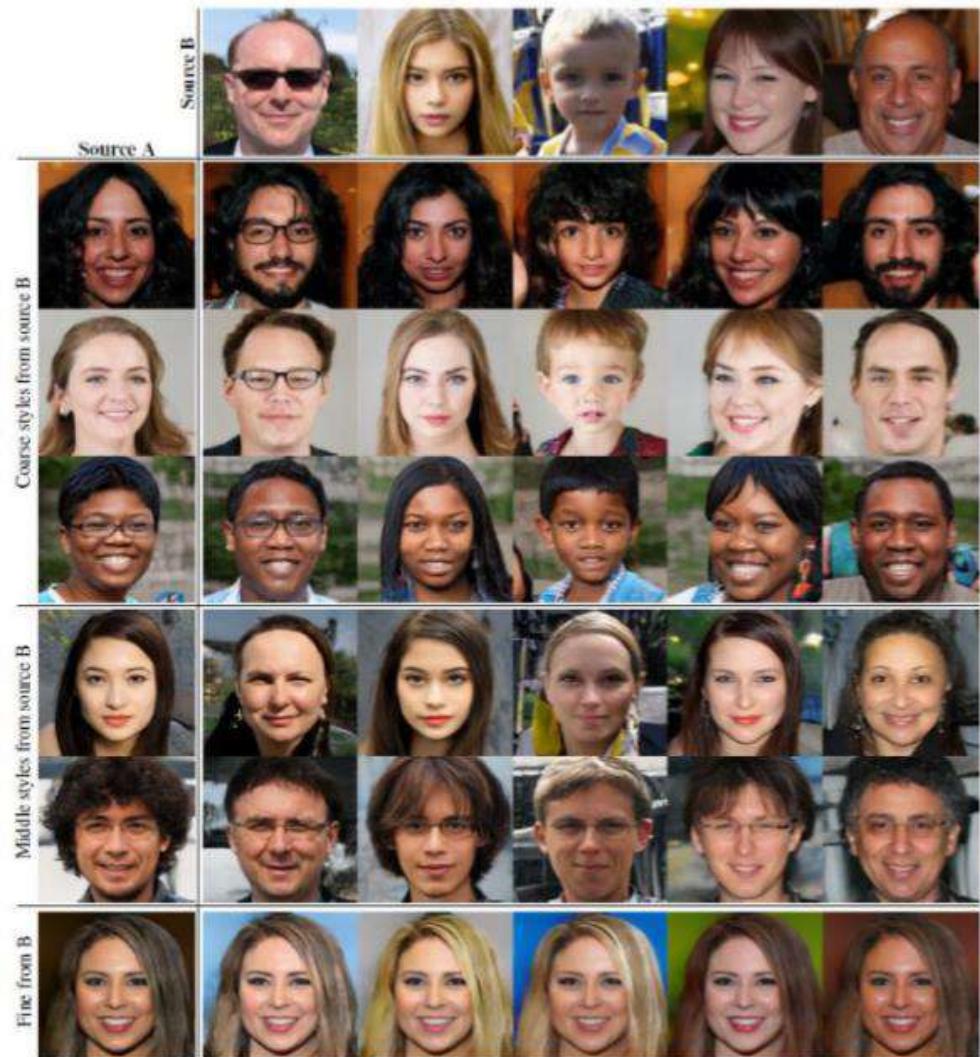
## CycleGAN



# GAN Applications

A style based generator architecture for GANs

arXiv, March 2019



# Major Difficulties

- Networks are difficult to converge.
- Ideal goal – Generator and discriminator to reach some desired equilibrium but this is rare.
- GANs are yet to converge on large problems (E.g. Imagenet).

# Common Failure Cases

- The discriminator becomes too strong too quickly and the generator ends up not learning anything.
- The generator only learns very specific weaknesses of the discriminator.
- The generator learns only a very small subset of the true data distribution.

# So what can we do?

- **Normalize the inputs**
- **A modified loss function**
- **Use a spherical Z**
- **BatchNorm**
- **Avoid Sparse Gradients: ReLU, MaxPool**
- **Use Soft and Noisy Labels**
- **DCGAN / Hybrid Models**
- **Track failures early (D loss goes to 0: failure mode)**
- **If you have labels, use them**
- **Add noise to inputs, decay over time**

# Conclusions

- Train GAN – Use discriminator as base model for transfer learning and the fine-tuning of a production model.
- A well-trained generator has learned the true data distribution well - Use generator as a source of data that is used to train a production model.

# Generative Adversarial Network

- Have been used in generating images, videos, poems, some simple conversation.
  - Note, image processing is easy (all animals can do it), NLP is hard (only human can do it).
  - This co-evolution approach might have far-reaching implications. Bengio: **this may hold the key to making computers a lot more intelligent.**
- 
- Ian Goodfellow:  
[https://www.youtube.com/watch?v=YpdP\\_0-IEOw](https://www.youtube.com/watch?v=YpdP_0-IEOw)
  - Radford, (generate voices also here)  
<https://www.youtube.com/watch?v=KeJINHjyzOU>
  - Tips for training GAN: <https://github.com/soumith/ganhacks>

# References

## Ian Goodfellow

- <https://www.youtube.com/watch?v=9JpdAg6uMXs&t=226s>
- <https://www.youtube.com/watch?v=HGYYEUSm-0Q>
- <https://www.youtube.com/watch?v=1WaRDrfMhmw&t=19s>
- <https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/>
- <https://www.youtube.com/watch?v=1WaRDrfMhmw&t=19s>
- <https://www.youtube.com/watch?v=5WoltGTWV54>
- GAN ZOO
- <https://happy-jihye.github.io/gan/>

# References

- <https://tryolabs.com/blog/2016/12/06/major-advancements-deep-learning-2016/>
- <https://blog.waya.ai/introduction-to-gans-a-boxing-match-b-w-neural-nets-b4e5319cc935#.6l7zh8u50>
- [https://en.wikipedia.org/wiki/Generative\\_adversarial\\_networks](https://en.wikipedia.org/wiki/Generative_adversarial_networks)
- <http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow/>
- <https://github.com/soumith/ganhacks>



**BITS** Pilani  
Pilani | Dubai | Goa | Hyderabad

# Machine Learning Diagrams

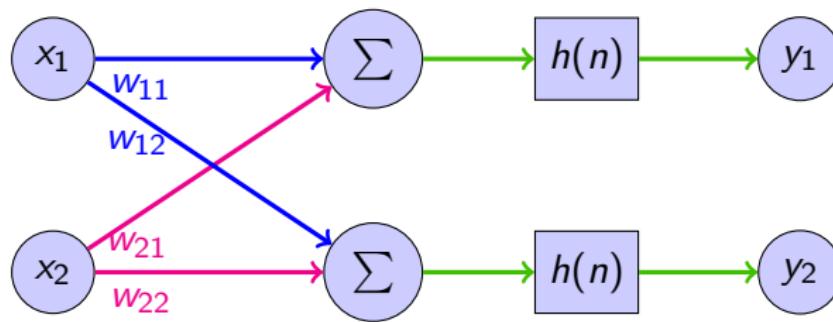
Seetha Parameswaran

[seetha.p@pilani.bits-pilani.ac.in](mailto:seetha.p@pilani.bits-pilani.ac.in)

The instructor is gratefully acknowledging the authors who made their course materials freely available online.

## Question 1

Consider the two perceptron model of the given neural network to classify the test sample while the training samples are given in the adjoining table.



$x_1$	$x_2$	class
1	1	bus
1	4	car
2	5	car
1	2	bus
4	2	plane
4	5	bicycle
5	4	bicycle
5	2	plane

1. Model the output in terms of  $y_1$  and  $y_2$ .
2. Obtain the weights appropriately to classify the test patterns correctly.
3. Design the activation functions with threshold appropriately.
4. Classify the input  $< 2.1, 4.5 >$ . Show the relevant steps.

# Answer 1

1. Model the output in terms of  $y_1$  and  $y_2$ .

class	$y_1$	$y_2$
bus	0	0
car	0	1
plane	1	0
bicycle	1	1

# Answer 1

Compute the weights and activation threshold for the perceptron.

Let  $w_{11} = 1$

$w_{12} = 0$

*threshold* = 3

$$sum_1 = w_{11}x_1 + w_{21}x_2$$

$$y_1 = \begin{cases} 1 & \text{if } sum_1 \geq 3 \\ 0 & \text{if } sum_1 < 3 \end{cases}$$

$x_1$	$x_2$	$sum_1$	$y_1$
1	1	$1 + 0 = 1$	0
1	4	$1 + 0 = 1$	0
2	5	$2 + 0 = 2$	0
1	2	$1 + 0 = 1$	0
4	2	$4 + 0 = 4$	1
4	5	$4 + 0 = 4$	1
5	4	$5 + 0 = 5$	1
5	2	$5 + 0 = 5$	1

# Answer 1

Compute the weights and activation threshold for the perceptron.

Let  $w_{12} = 0$

$w_{22} = 1$

threshold = 3

$$sum_2 = w_{12}x_1 + w_{22}x_2$$

$$y_2 = \begin{cases} 1 & \text{if } sum_2 \geq 3 \\ 0 & \text{if } sum_2 < 3 \end{cases}$$

$x_1$	$x_2$	$sum_2$	$y_2$
1	1	$0 + 1 = 1$	0
1	4	$0 + 4 = 4$	1
2	5	$0 + 5 = 5$	1
1	2	$0 + 2 = 2$	0
4	2	$0 + 2 = 2$	0
4	5	$0 + 5 = 5$	1
5	4	$0 + 4 = 4$	1
5	2	$0 + 2 = 2$	0

- 
2. Obtain the weights appropriately to classify the test patterns correctly.

$$w_{11} = 1$$

$$w_{12} = 0$$

$$w_{21} = 0$$

$$w_{22} = 1$$

3. Design the activation functions with threshold appropriately.

$$\text{threshold} = 3$$

# Answer 1



4. Classify the input (2.1, 4.5).

$$\begin{aligned}sum_1 &= w_{11}x_1 + w_{21}x_2 \\&= 1 * 2.1 + 0 * 4.5 = 2.1 \\&< 3 \leftarrow threshold\end{aligned}$$

$$y_1 = 0$$

$$\begin{aligned}sum_2 &= w_{12}x_1 + w_{22}x_2 \\&= 0 * 2.1 + 1 * 4.5 = 4.5 \\&> 3 \leftarrow threshold\end{aligned}$$

$$y_2 = 1$$

$$\begin{aligned}y &= (0, 1) \\&= Car\end{aligned}$$

# Answer 1

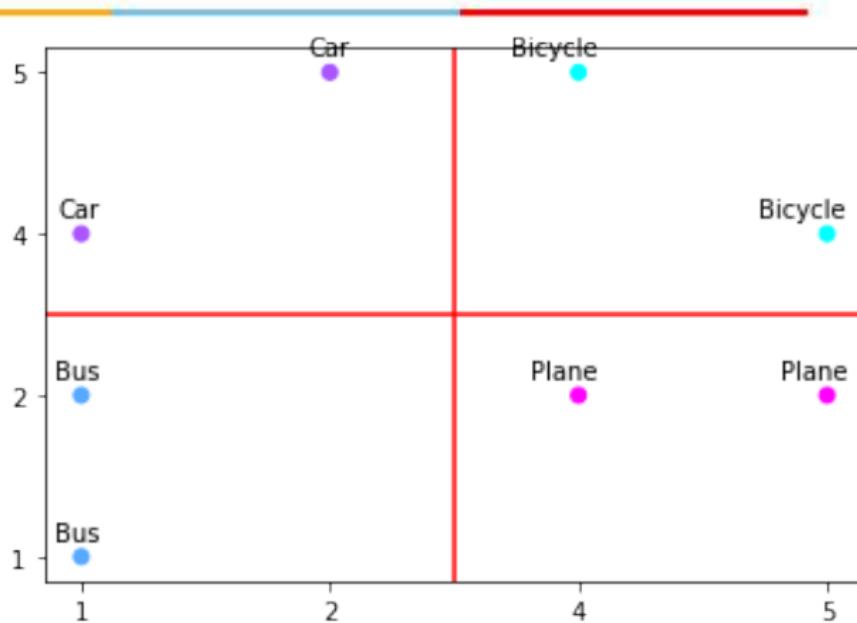


Figure: Graphical representation of the dataset

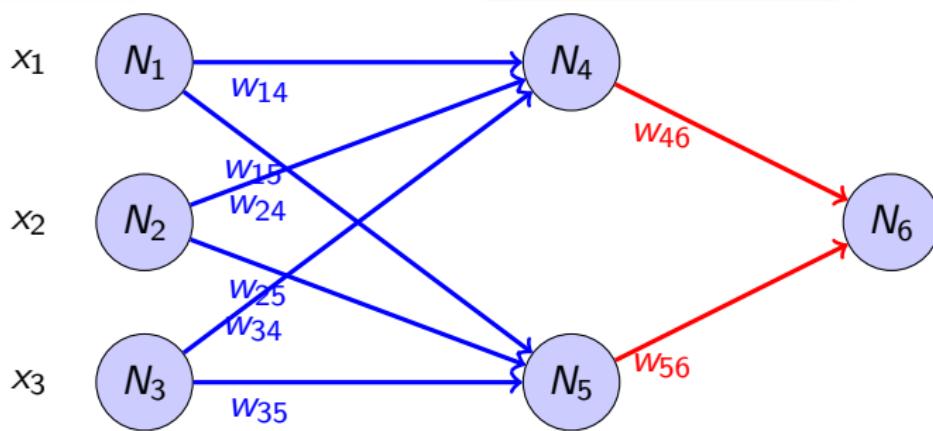
## Question 2

Consider the given feed forward neural network. Let learning rate be 0.9. The initial weights and the biases are given in the table. Consider the following sample as first training example  $X = (1, 0, 1)$  with class label as 1.

1. Show

1. computation of net output at each node.
2. calculation of error at each node.
3. calculation of updation of weights and biases after one iteration.

## Question 2



$$b_4 = (-0.4)$$

$$w_{14} = 0.2$$

$$w_{24} = 0.4$$

$$w_{34} = (-0.5)$$

$$b_5 = 0.1$$

$$w_{15} = (-0.3)$$

$$w_{25} = 0.1$$

$$w_{35} = 0.2$$

$$b_6 = 0.1$$

$$w_{45} = (-0.3)$$

$$w_{56} = (-0.2)$$

## Answer 2

A] Computation of net output at each node.

$$z_j = \sum_{ij} w_{ij}x_i + b_i$$

$$a_j = \frac{1}{1 + e^{-z_j}}$$

$$\begin{aligned} N_4 \quad z_4 &= w_{14}x_1 + w_{24}x_2 + w_{34}x_3 + b_4 & a_4 &= \frac{1}{1+e^{-(0.7)}} = 0.331 \\ z_4 &= 0.2 * 1 + 0 - 0.5 * 1 - 0.4 = (-0.7) \end{aligned}$$

$$\begin{aligned} N_5 \quad z_5 &= w_{15}x_1 + w_{25}x_2 + w_{35}x_3 + b_5 & a_5 &= \frac{1}{1+e^{-(0.1)}} = 0.525 \\ z_5 &= (-0.3) * 1 + 0 + 0.2 * 1 + 0.2 = 0.1 \end{aligned}$$

$$\begin{aligned} N_6 \quad z_6 &= w_{46}a_4 + w_{56}a_5 + b_6 & a_6 &= \frac{1}{1+e^{(-0.014)}} = 0.474 \\ z_6 &= (-0.3) * 0.331 - 0.2 * 0.525 + 0.1 \\ z_6 &= (-0.104) \end{aligned}$$

### B] Calculation of error at each node.

At output layer  $E_j = a_j(1 - a_j)(y - a_j)$

At hidden layer  $E_j = a_j(1 - a_j) \sum_k E_k w_{jk}$

$$N_6 \quad E_6 = a_6(1 - a_6)(y - a_6)$$

$$E_6 = 0.474(1 - 0.474)(1 - 0.474) = 0.131$$

$$N_5 \quad E_5 = a_5(1 - a_5)(E_6 * w_{56})$$

$$E_5 = 0.525(1 - 0.525)(0.131 * (-0.2)) = (-0.0065)$$

$$N_4 \quad E_4 = a_4(1 - a_4)(E_6 * w_{46})$$

$$E_4 = 0.331(1 - 0.331)(0.131 * (-0.3)) = (-0.0087)$$

## Answer 2

C] Calculation of updation of weights and biases after one iteration.

$$w_{ij} = w_{ij} + \alpha E_j a_i$$

$$b_j = b_j + \alpha E_j$$

$$w_{46} = (-0.3) + 0.9 * 0.131 * 0.331 = 0.268$$

$$w_{56} = (-0.2) + 0.9 * 0.131 * 0.525 = (-0.138)$$

$$w_{14} = 0.2 + 0.9 * (-0.0087) * 1 = 0.192$$

$$w_{15} = (-0.3) + 0.9 * (-0.0065) * 1 = (-0.306)$$

$$w_{24} = 0.4 + 0 = 0.4$$

$$w_{25} = 0.1 + 0 = 0.1$$

$$w_{34} = (-0.5) + 0.9 * (-0.0087) * 1 = (-0.508)$$

$$w_{35} = 0.2 + 0.9 * (-0.0065) * 1 = 0.194$$

$$b_4 = (-0.4) + 0.9 * (-0.0087) = (-0.408)$$

$$b_5 = 0.2 + 0.9 * (-0.0065) = 0.194$$

$$b_6 = 0.1 + 0.9 * 0.131 = 0.218$$