Foundations of Machine Learning

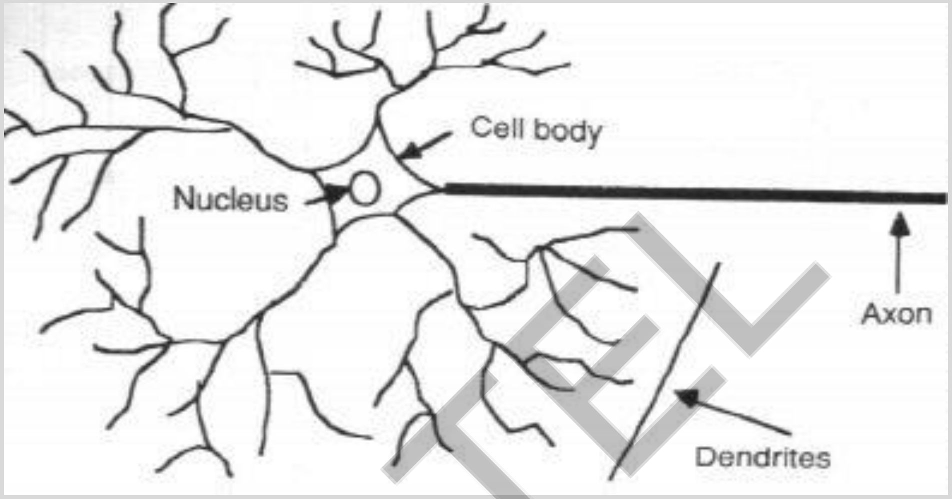
Module 6: Neural Network
Part A: Introduction

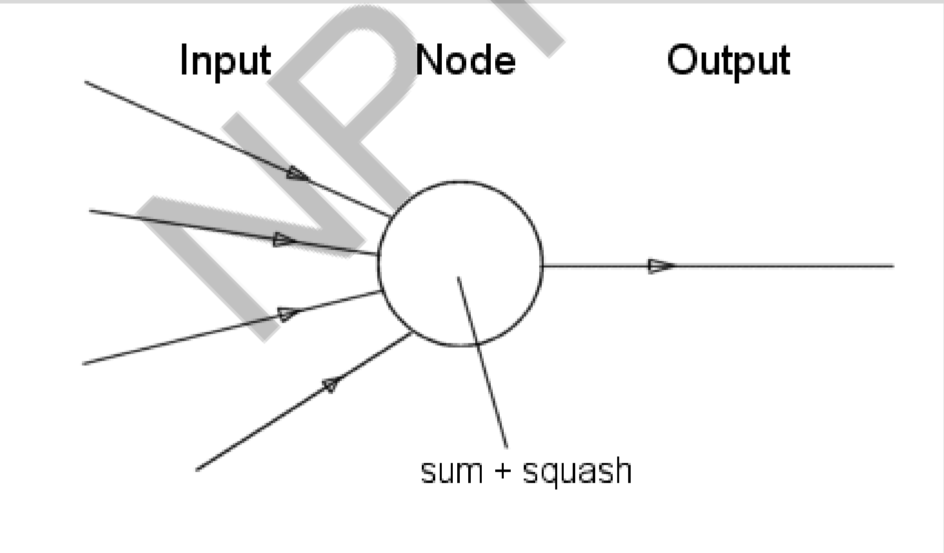
Sudeshna Sarkar
IIT Kharagpur

Introduction

- Inspired by the human brain.
- Some NNs are models of biological neural networks
- Human brain contains a massively interconnected net of 10^{10}-10^{11} (10 billion) neurons (cortical cells)
  - Massive parallelism – large number of simple processing units
  - Connectionism – highly interconnected
  - Associative distributed memory
    - Pattern and strength of synaptic connections

Neuron

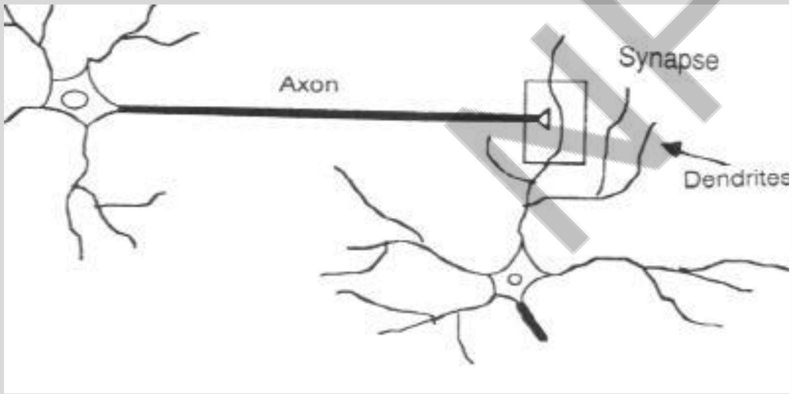


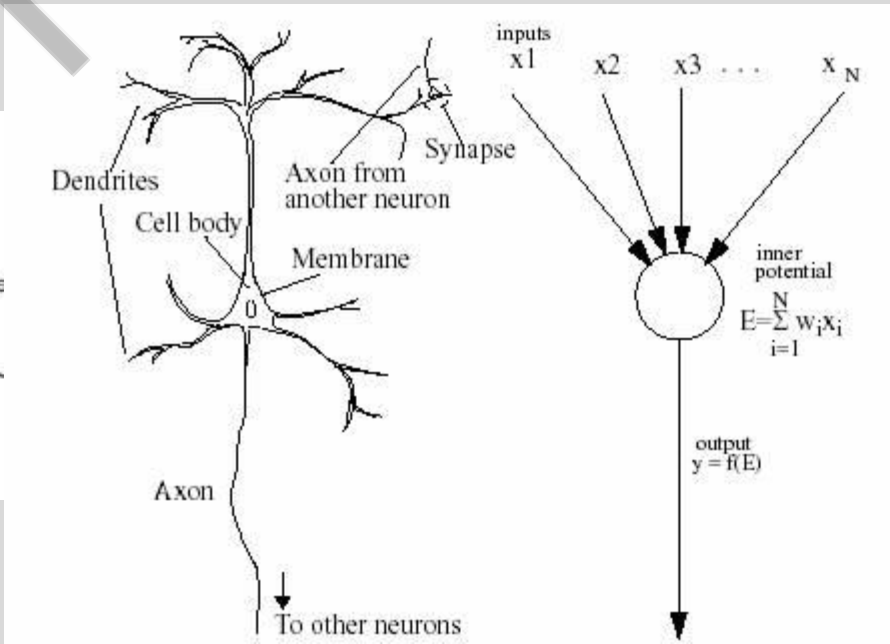


Neural Unit

ANNs

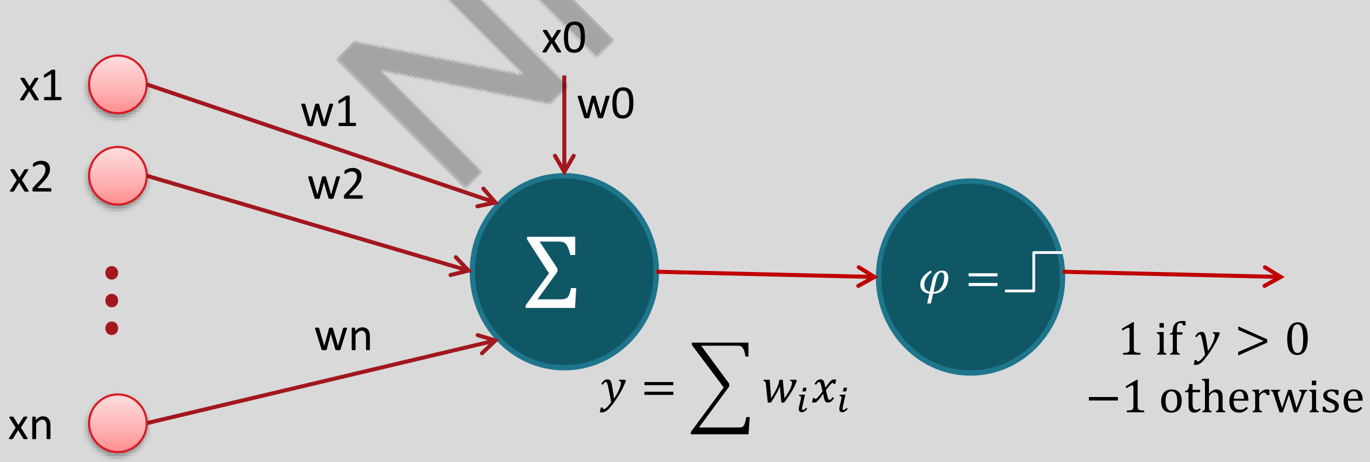
- ANNs incorporate the two fundamental components of biological neural nets:
  1. Nodes - Neurones
  2. Weights - Synapses





Perceptrons

- Basic unit in a neural network: Linear separator
  - N inputs, x1 ... xn
  - Weights for each input, w1 ... wn
  - A bias input x0 (constant) and associated weight w0
  - Weighted sum of inputs, $y = \sum_{i=0}^{n} w_i x_i$
  - A threshold function, i.e., 1 if y > 0, −1 if y <= 0



Perceptron training rule

Updates perceptron weights for a training ex as follows:

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

- If the data is linearly separable and η is sufficiently small, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations

Gradient Descent

- Perceptron training rule may not converge if points are not linearly separable
- Gradient descent by changing the weights by the total error for all training points.
  - If the data is not linearly separable, then it will converge to the best fit

Linear neurons

- The neuron has a real-valued output which is a weighted sum of its inputs

$$\hat{y} = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

- Define the error as the squared residuals summed over all training cases:

$$E = \frac{1}{2} \sum_j (y - \hat{y})^2$$

- Differentiate to get error derivatives for weights

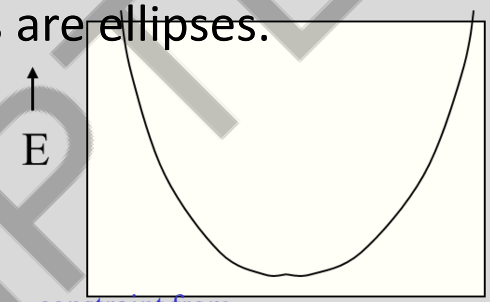
$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1..m} \frac{\partial \widehat{y_j}}{\partial w_i} \frac{\partial E_j}{\partial \widehat{y_j}} = - \sum_{j=1..m} x_{i,j} (y_j - \widehat{y_j})$$

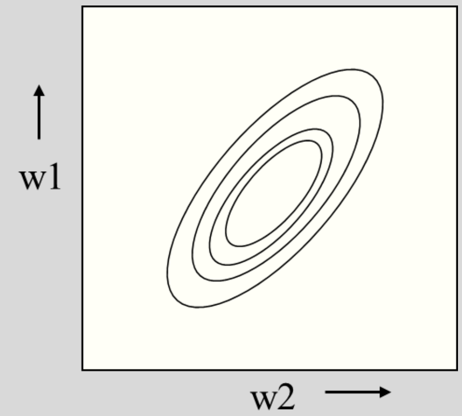
- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

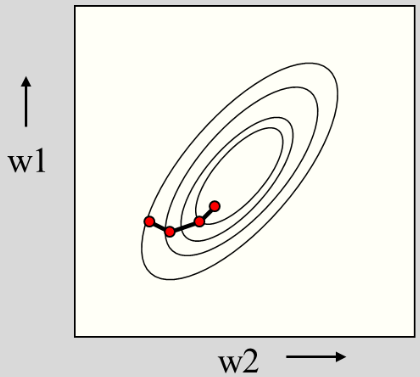
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

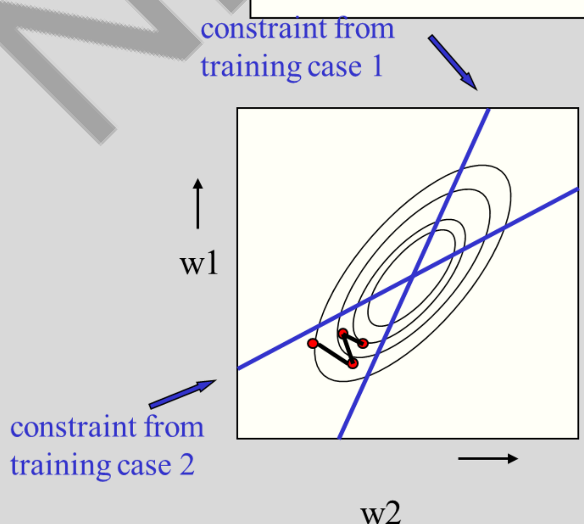
Error Surface

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
  - For a linear neuron, it is a quadratic bowl.
  - Vertical cross-sections are parabolas.
  - Horizontal cross-sections are ellipses.









Batch Line and Stochastic Learning

Batch Learning

- Steepest descent on the error surface

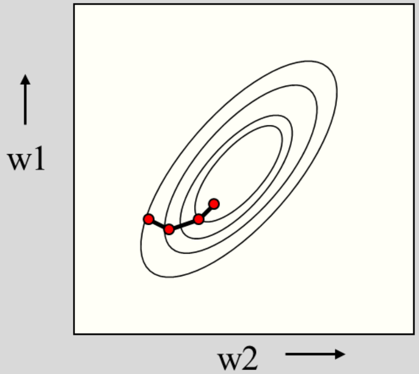
Stochastic/ Online Learning

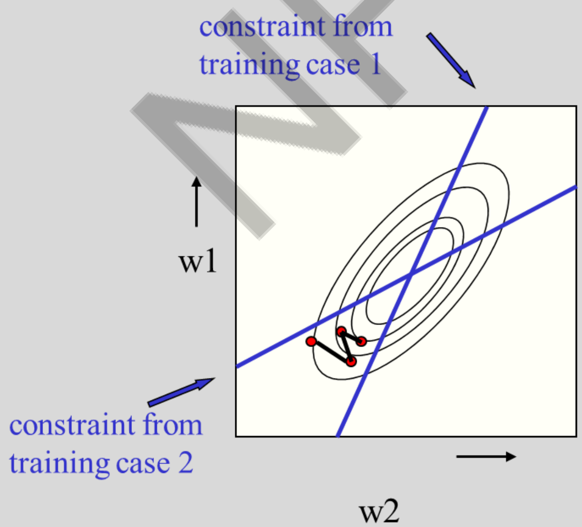
For each example compute the gradient.

$$E = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\frac{\partial \hat{y}}{\partial w_i}\frac{\partial E_j}{\partial \hat{y}}$$

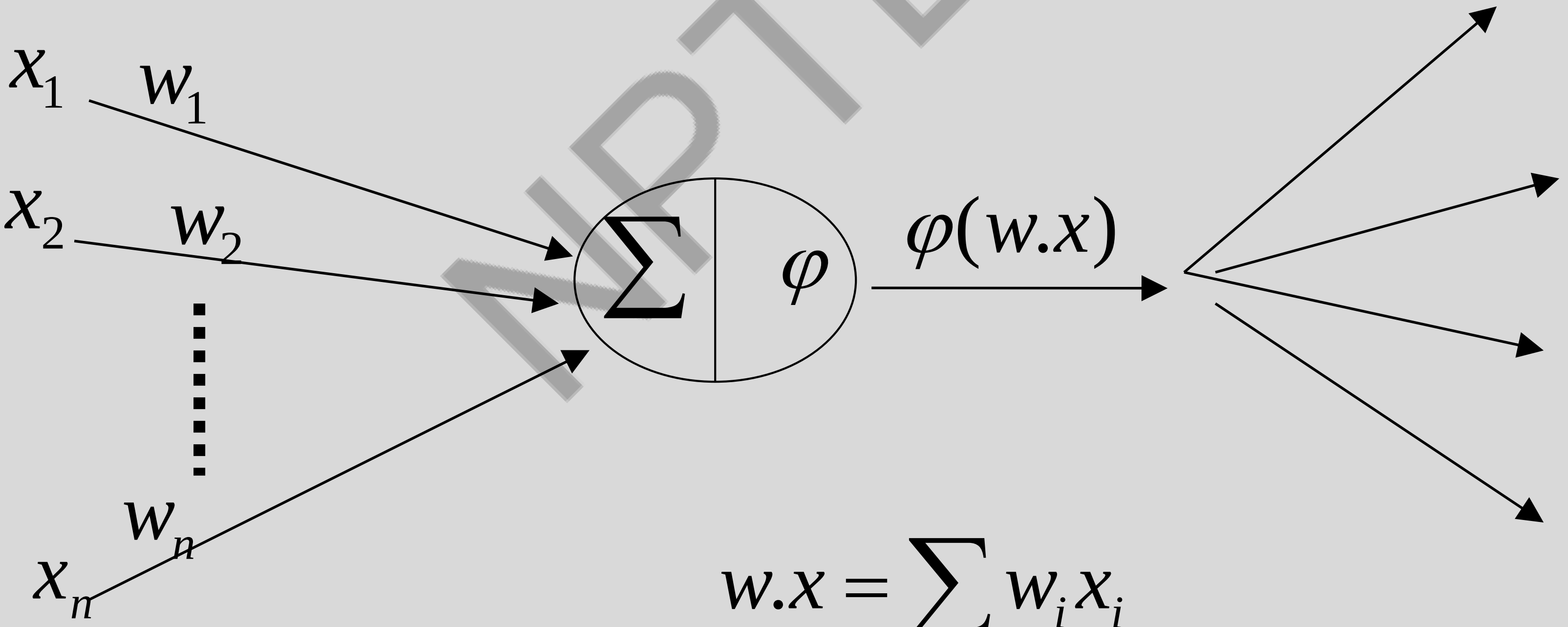
$$= -x_i(y - \hat{y})$$





Computation at Units

- Compute a 0-1 or a *graded* function of the weighted sum of the inputs
- $\varphi()$ is the *activation* function



$$w.x = \sum w_i x_i$$

Neuron Model: Logistic Unit

$$\varphi(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w.x}}$$

$$\phi'(z) = \varphi(z)\big(1-\varphi(z)\big)$$

$$E = \frac{1}{2}\sum_d (y-\hat{y})^2 = \frac{1}{2}\sum_d (y-\varphi(w.x_d))^2$$

$$\frac{\partial E}{\partial w_i} = \sum_d \frac{1}{2}\frac{\partial E_d}{\partial \widehat{y_d}}\frac{\partial \widehat{y_d}}{\partial w_i}$$

$$= \sum_d (y_d - \widehat{y_d})\frac{\partial y}{\partial w_i}(y_d - \varphi(w.x_d))$$

$$= -\sum_d (y_d - \widehat{y_d})\,\varphi'(w.x_d)\,x_{i,d}$$

$$= -\sum_d (y_d - \widehat{y_d})\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$$

Training Rule: $\Delta w_i = \eta \sum_d \big(y_d - \widehat{y_d}\big)\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$

Thank You

Foundations of Machine Learning

Module 6: Neural Network
Part B: Multi-layer Neural Network

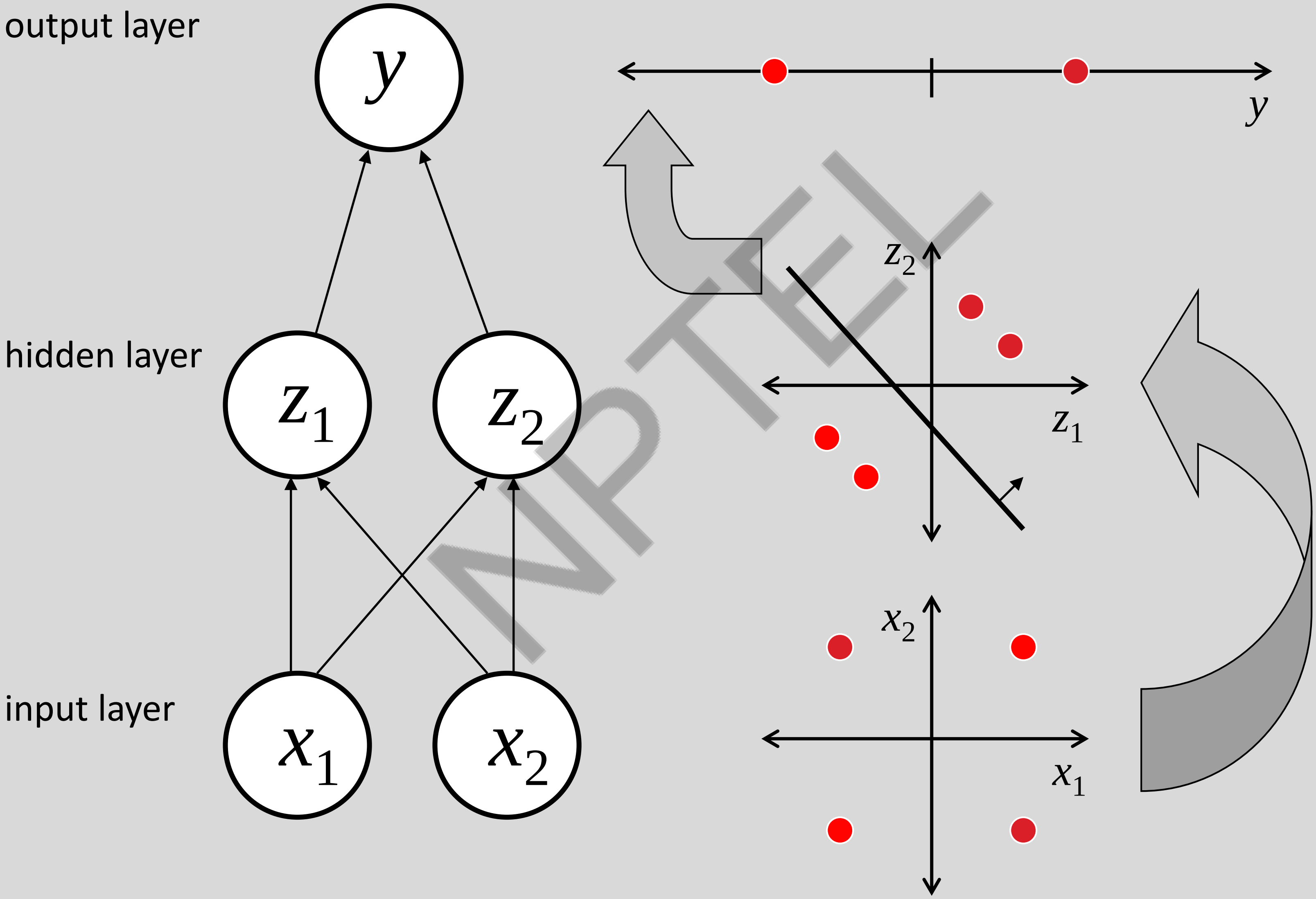
Sudeshna Sarkar
IIT Kharagpur

Limitations of Perceptrons

- Perceptrons have a *monotinicity* property:

  If a link has positive weight, activation can only increase as the corresponding input value increases (*irrespective* of other input values)
- Can't represent functions where input *interactions* can cancel one another's effect (e.g. XOR)
- Can represent only linearly separable functions

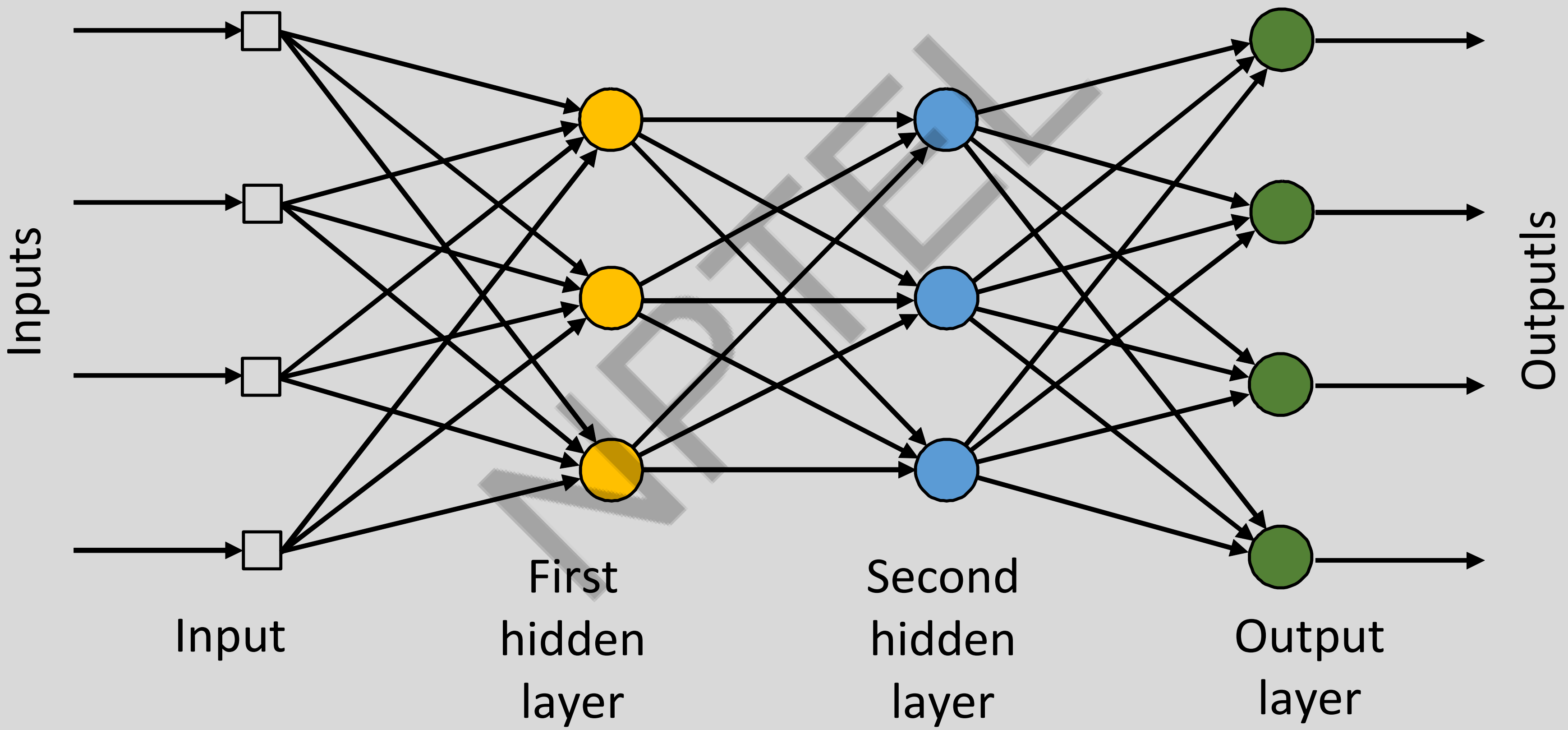
A solution: multiple layers



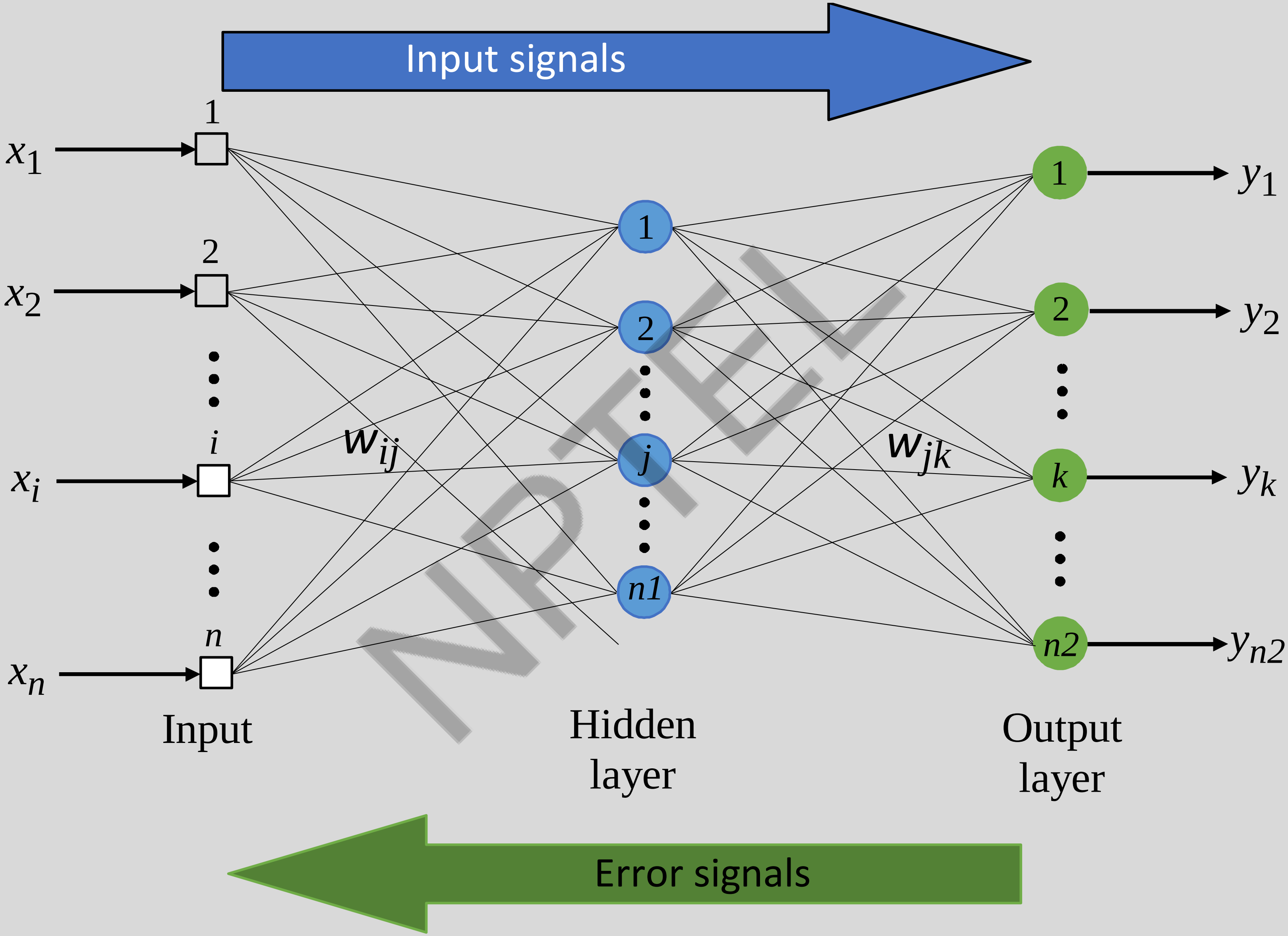
Power/Expressiveness of Multilayer Networks

- Can represent interactions among inputs
- Two layer networks can represent any Boolean function, and continuous functions (within a tolerance) as long as the number of hidden units is sufficient and appropriate activation functions used
- Learning algorithms exist, but weaker guarantees than perceptron learning algorithms

Multilayer Network

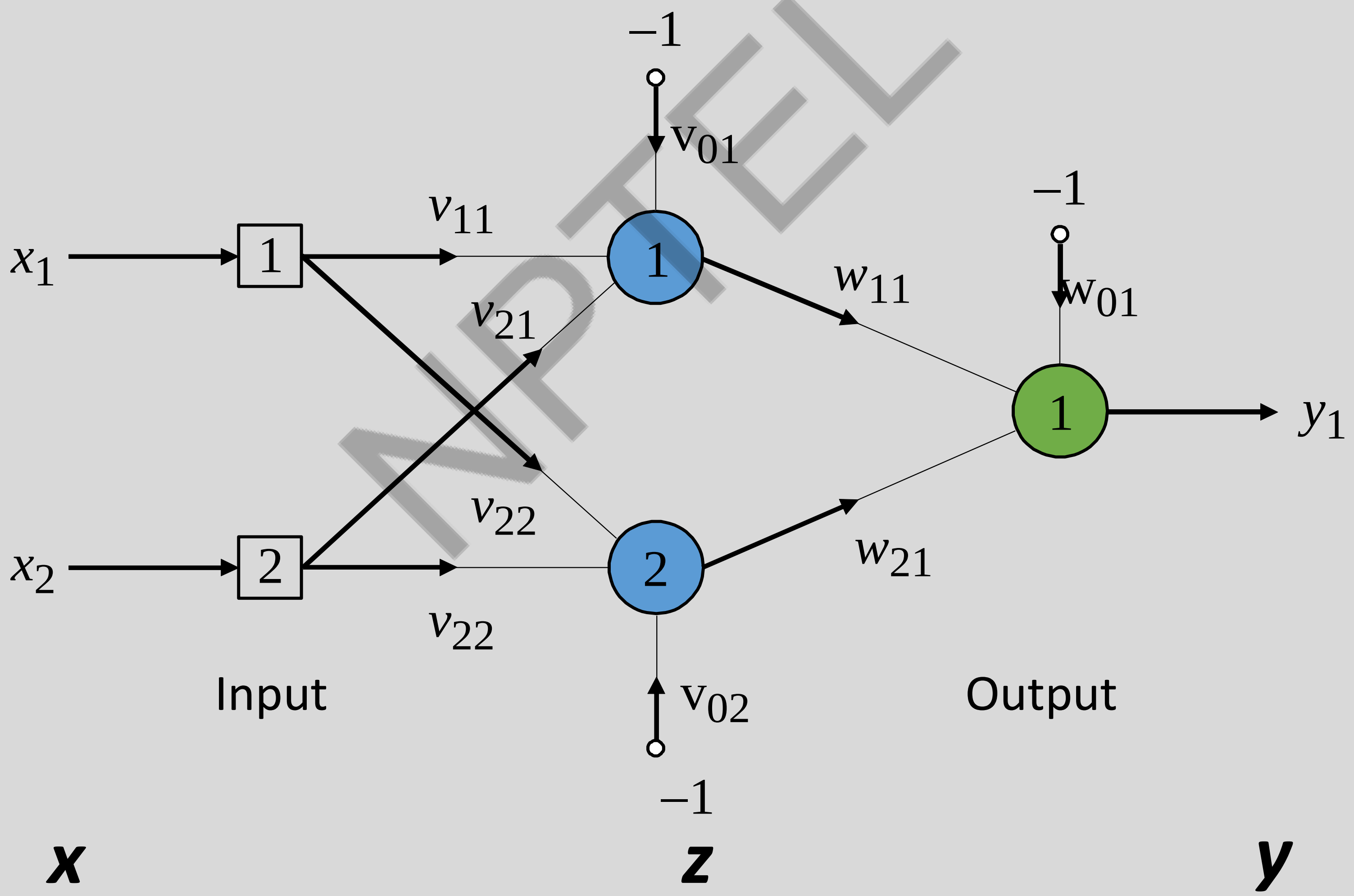


Two-layer back-propagation neural network



The back-propagation training algorithm

- Step 1: Initialisation
  Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range



Backprop

- Initialization
  - Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range
- Forward computing:
  - Apply an input vector ***x*** to input units
  - Compute activation/output vector ***z*** on hidden layer

  $z_j = \varphi(\sum_i v_{ij} x_i)$

  - Compute the output vector ***y*** on output layer

  $y_k = \varphi(\sum_j w_{jk} z_j)$

  y is the result of the computation.

Learning for BP Nets

- Update of weights in W (between output and hidden layers):
  - delta rule
- Not applicable to updating V (between input and hidden)
  - don't know the target values for hidden units z1, Z2, … ,ZP
- Solution: Propagate errors at output units to hidden units to drive the update of weights in V (again by delta rule) (error BACKPROPAGATION learning)
- Error backpropagation can be continued downward if the net has more than one hidden layer.
- How to compute errors on hidden units?

Derivation

- For one output neuron, the error function is

$$E = \frac{1}{2}(y - \hat{y})^2$$

- For each unit j, the output o_j is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right)$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous n neurons.

- Finding the derivative of the error:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Derivation

- Finding the derivative of the error:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = o_i$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j}\varphi(net_j) = \varphi(net_j)\left(1 - \varphi(net_j)\right)$$

Consider E as as a function of the inputs of all neurons $Z = \{z_1, z_2, \dots\}$ receiving input from neuron j,

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(net_{z_1}, net_{z_2}, \dots)}{\partial o_j}$$

taking the total derivative with respect to o_j, a recursive expression for the derivative is obtained:

$$\frac{\partial E}{\partial o_j} = \sum_l \left(\frac{\partial E}{\partial net_{z_l}}\frac{\partial net_{z_l}}{\partial o_j}\right) = \sum_l \left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial net_{z_l}} w_{jz_l}\right)$$

$$\frac{\partial E}{\partial o_j} = \sum_l \left(\frac{\partial E}{\partial net_{z_l}} \frac{\partial net_{z_l}}{\partial o_j} \right) = \sum_l \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial net_{z_l}} w_{jz_l} \right)$$

- Therefore, the derivative with respect to o_j can be calculated if all the derivatives with respect to the outputs o_{z_l} of the next layer – the one closer to the output neuron – are known.
- Putting it all together:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

With

$$\delta_j = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j) o_j (1 - o_j) \text{ if } j \text{ is an output neuron} \\ \left(\sum_Z \delta_{z_l} w_{jl} \right) o_j (1 - o_j) \text{ if } j \text{ is an inner neuron} \end{cases}$$

To update the weight w_{ij} using gradient descent, one must choose a learning r

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

Backpropagation Algorithm

Thank You

Foundations of Machine Learning

Module 6: Neural Network

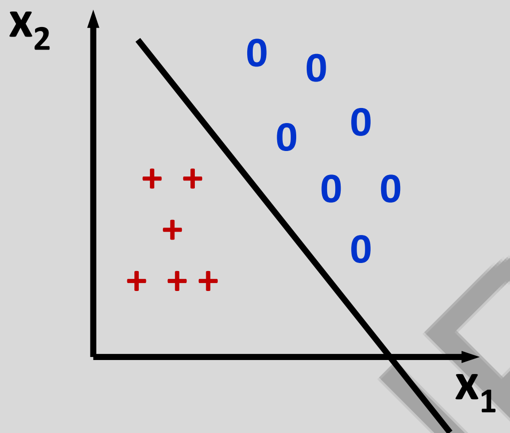
Part C: Neural Network and Backpropagation Algorithm

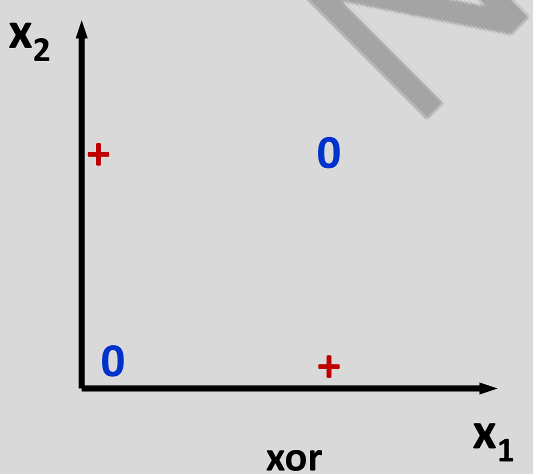
Sudeshna Sarkar

IIT Kharagpur

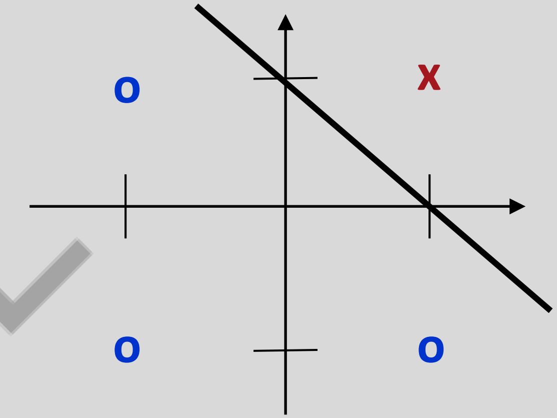
Single layer Perceptron

- Single layer perceptrons learn linear decision boundaries

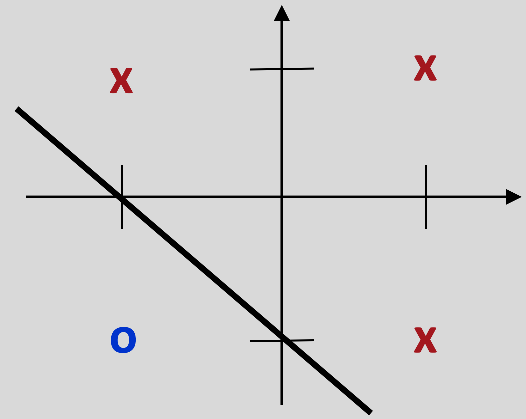




xor



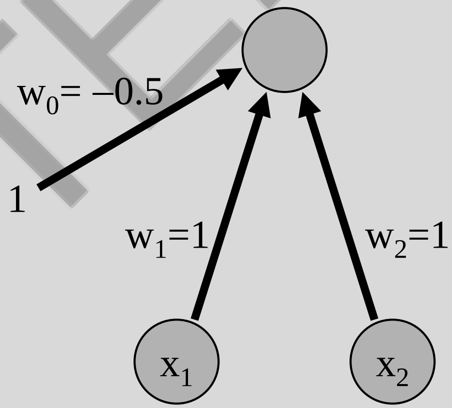
x: class I (y = 1)
o: class II (y = -1)

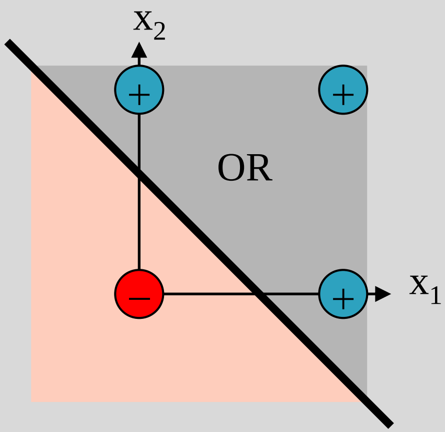


x: class I (y = 1)
o: class II (y = -1)

Boolean OR

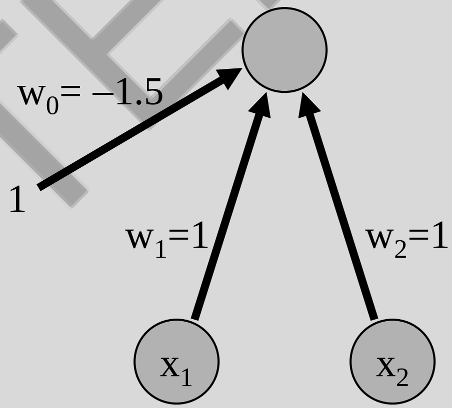
input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	1

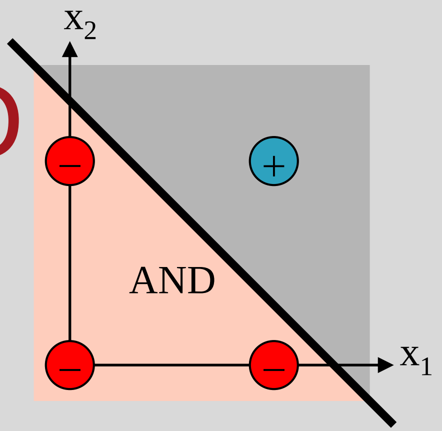




Boolean AND

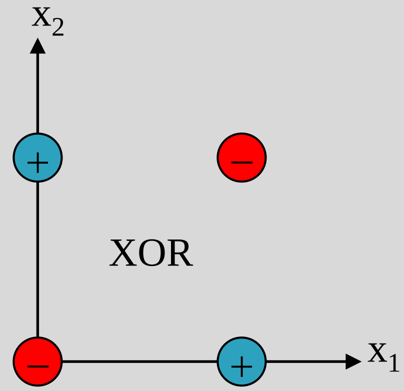
input x1	input x2	ouput
0	0	0
0	1	0
1	0	0
1	1	1





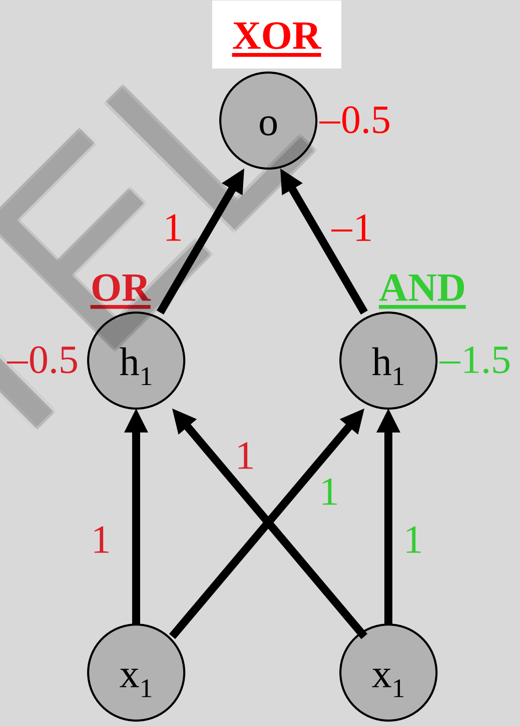
Boolean XOR

input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0



Boolean XOR

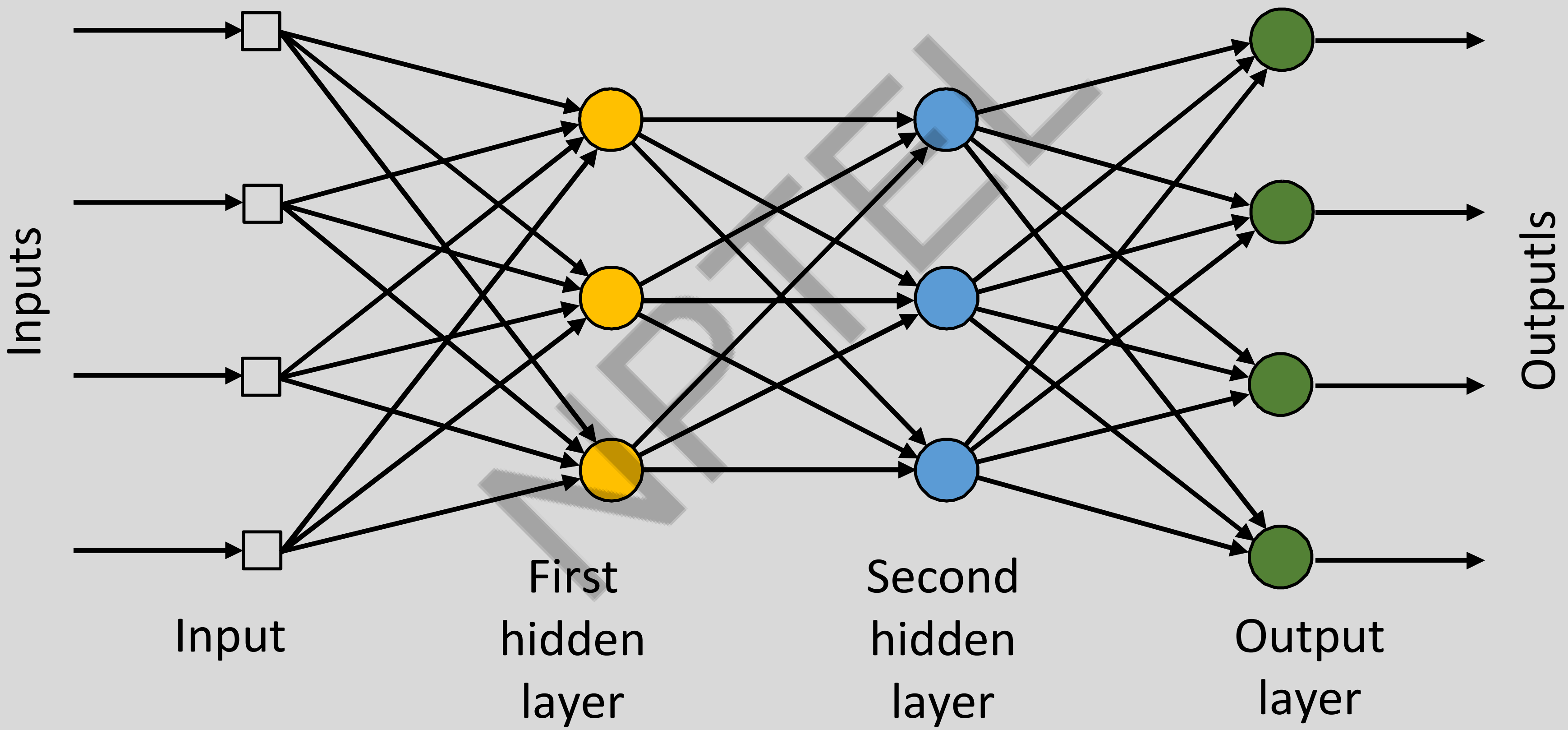
input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0



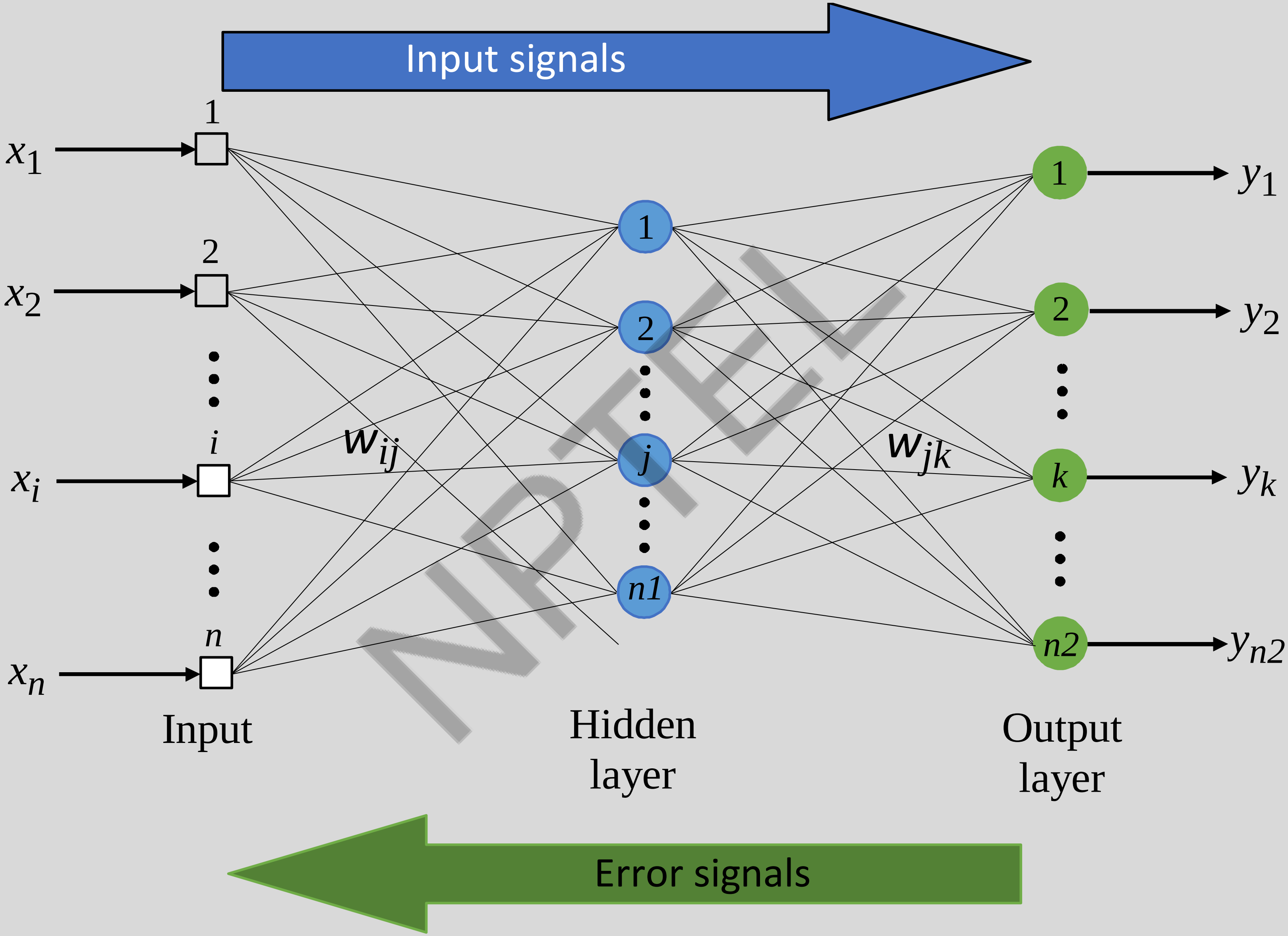
Representation Capability of NNs

- Single layer nets have limited representation power (linear separability problem). Multi-layer nets (or nets with non-linear hidden units) may overcome linear inseparability problem.
- Every Boolean function can be represented by a network with a single hidden layer.
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.

Multilayer Network



Two-layer back-propagation neural network



Derivation

- For one output neuron, the error function is

$$E = \frac{1}{2}(y - o)^2$$

- For each unit j, the output o_j is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right)$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous n neurons.

- Finding the derivative of the error:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

For one output neuron, the error function is $E = \frac{1}{2}(y - o)^2$

For each unit j, the output o_j is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right)$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$= \sum_{l} \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial net_{z_l}} w_{jz_l} \right) \varphi(net_j)\left(1 - \varphi(net_j)\right) o_i$$

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

with

$$\delta_j = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - y_j) o_j (1 - o_j) \text{ if } j \text{ is an output neuron} \\ \left(\sum_{Z} \delta_{z_l} w_{jl} \right) o_j (1 - o_j) \text{ if } j \text{ is an inner neuron} \end{cases}$$

To update the weight w_{ij} using gradient descent, one must choose a learning rate η.

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

Backpropagation Algorithm

Initialize all weights to small random numbers.
Until satisfied, do

- For each training example, do
  - Input the training example to the network and compute the network outputs
  - For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(y_k - o_k)$$

  - For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}, \delta_k,$$

  - Update each network weight w_i, j

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

x_d = input

y_d = target output

o_d = observed unit output

w_{ij} = wt from i to j

Backpropagation

- Gradient descent over entire network weight vector
- Can be generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
- May include weight momentum α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Training may be slow.
- Using network after training is very fast

Training practices: batch vs. stochastic vs. mini-batch gradient descent

- **Batch gradient descent**:
  1. Calculate outputs for the entire dataset
  2. Accumulate the errors, back-propagate and update

Too slow to converge
Gets stuck in local minima

- **Stochastic/online gradient descent**:
  1. Feed forward a training example
  2. Back-propagate the error and update the parameters

Converges to the solution faster
Often helps get the system out of local minima

- **Mini-batch gradient descent**:

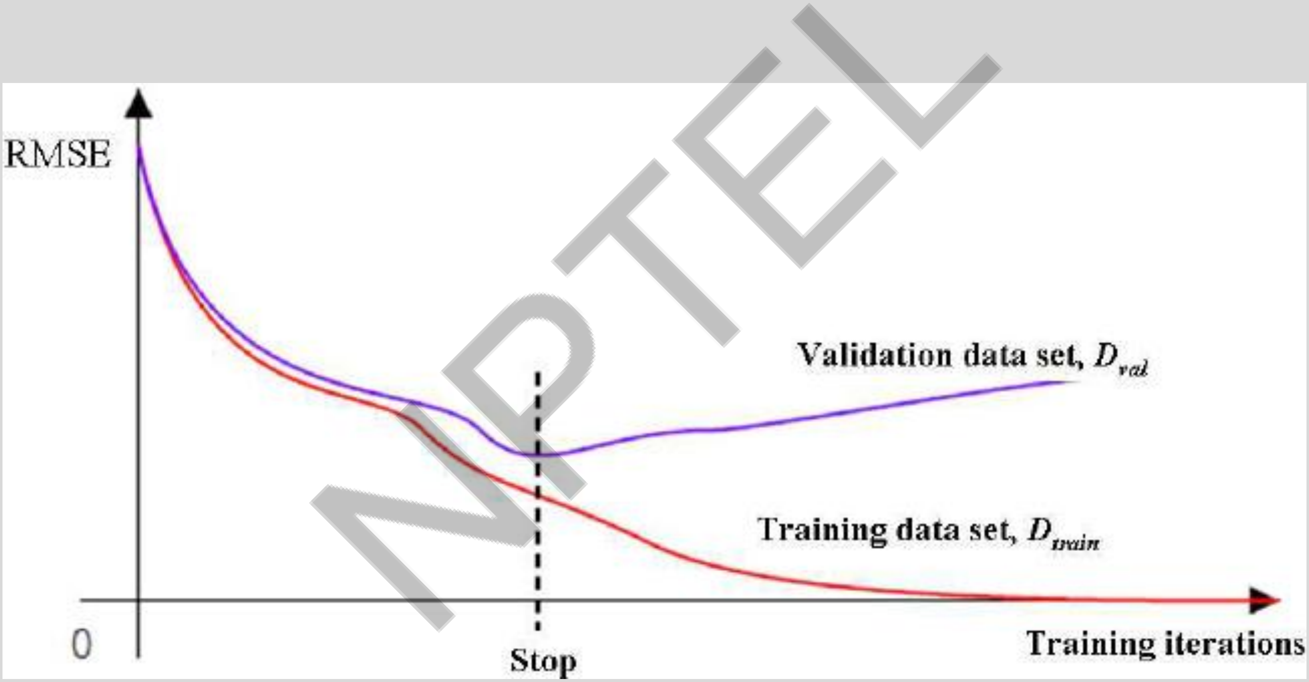
Learning in *epochs*
Stopping

- Train the NN on the entire training set over and over again
- Each such episode of training is called an “epoch”

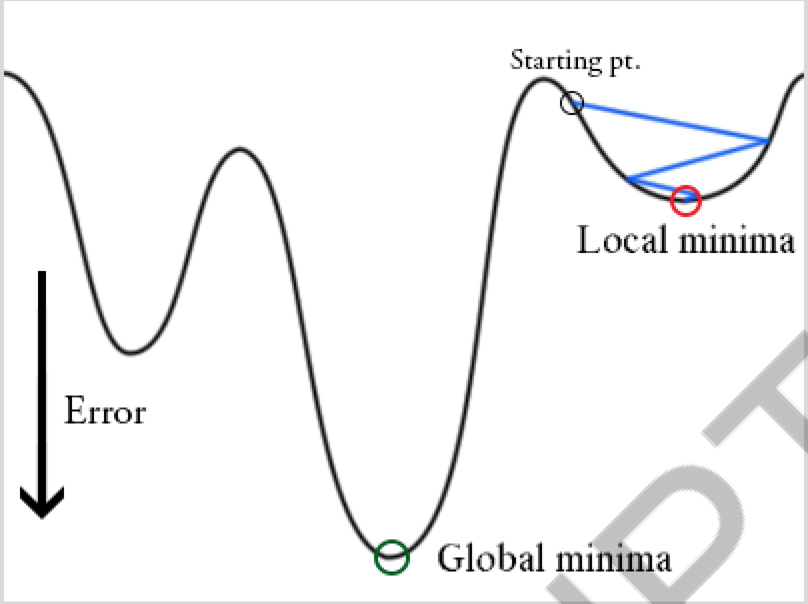
Stopping

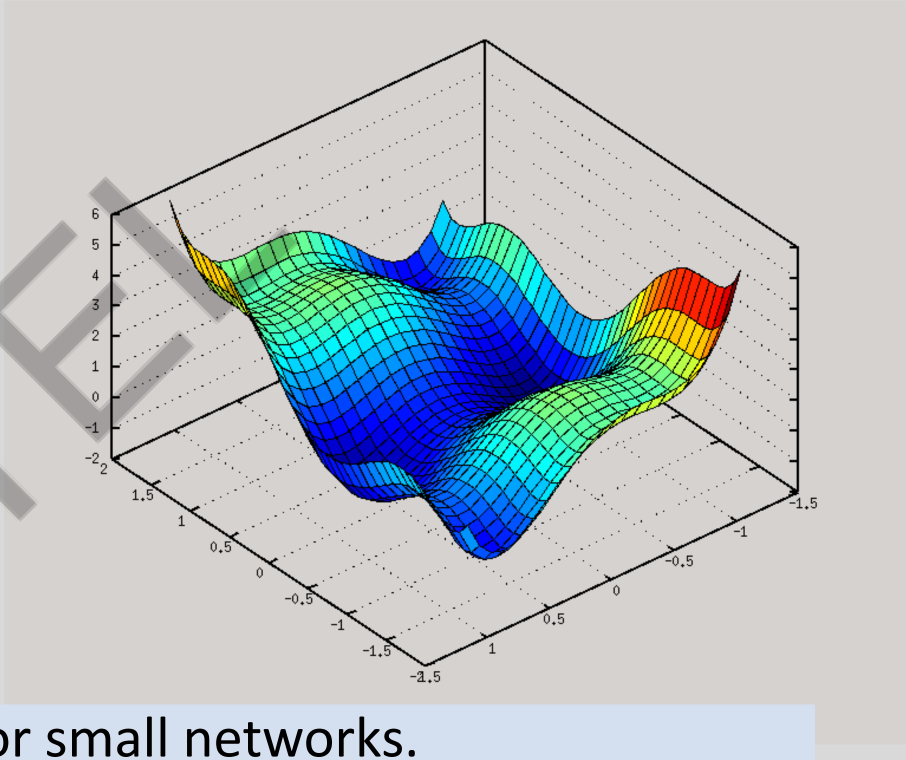
1. Fixed maximum number of epochs: most naïve
2. Keep track of the training and validation error curves.

Overfitting in ANNs



Local Minima





- NN can get stuck in local minima for small networks.
- For most large networks (many weights) local minima rarely occurs.
- It is unlikely that you are in a minima in every dimension simultaneously.

ANN

- Highly expressive non-linear functions
- Highly parallel network of logistic function units
- Minimizes sum of squared training errors
- Can add a regularization term (weight squared)
- Local minima
- Overfitting

Thank You

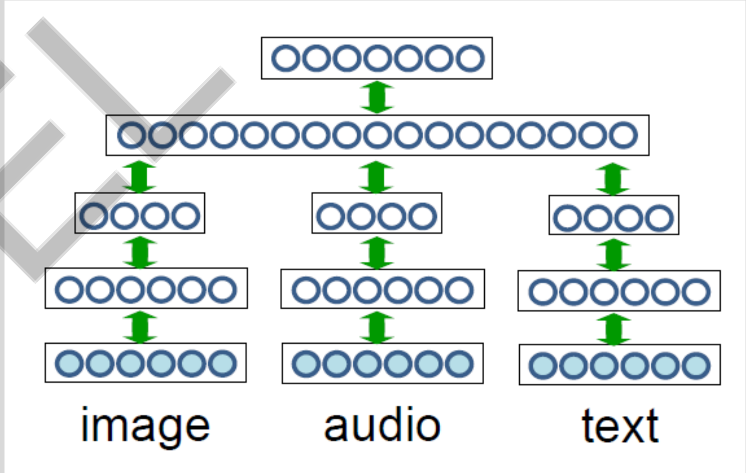
Foundations of Machine Learning

Module 6: Neural Network
Part D: Deep Neural Network

Sudeshna Sarkar
IIT Kharagpur

Deep Learning

- Breakthrough results in
  - Image classification
  - Speech Recognition
  - Machine Translation
  - Multi-modal learning



Deep Neural Network

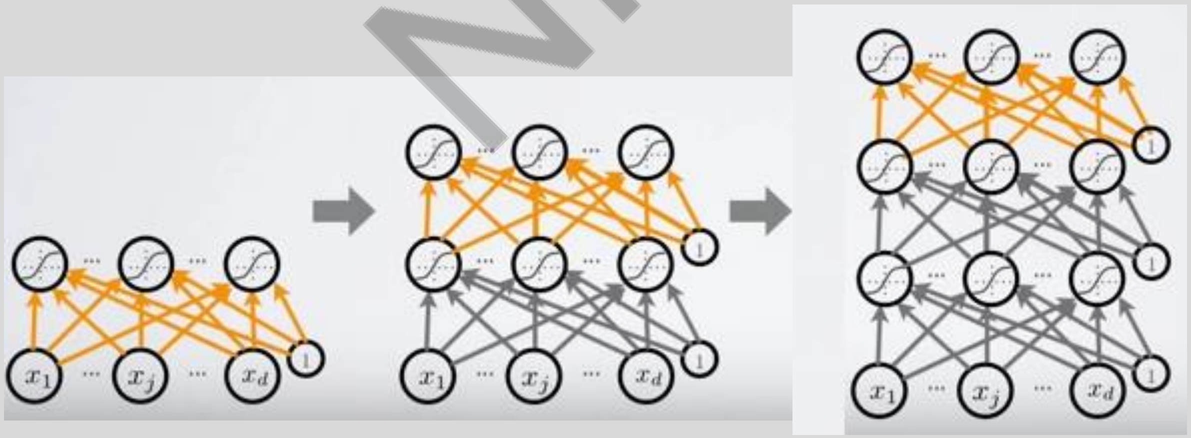
- Problem: training networks with many hidden layers doesn’t work very well
- Local minima, very slow training if initialize with zero weights.
- Diffusion of gradient.

Hierarchical Representation

- Hierarchical Representation help represent complex functions.
- NLP: character ->word -> Chunk -> Clause -> Sentence
- Image: pixel > edge -> texton -> motif -> part -> object
- Deep Learning: learning a hierarchy of internal representations
- Learned internal representation at the hidden layers (trainable feature extractor)
- Feature learning

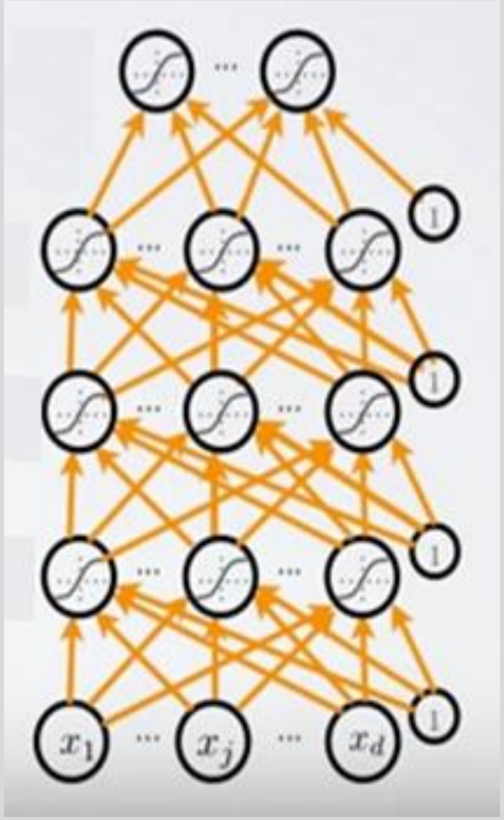
Unsupervised Pre-training

- We will use greedy, layer wise pre-training
  - Train one layer at a time
  - Fix the parameters of previous hidden layers
  - Previous layers viewed as feature extraction
- find hidden unit features that are more common in training input than in random inputs



Tuning the Classifier

- After pre-training of the layers
  - Add output layer
  - Train the whole network using supervised learning (Back propagation)



Deep neural network

- Feed forward NN
- Stacked Autoencoders (multilayer neural net with target output = input)
- Stacked restricted Boltzmann machine
- Convolutional Neural Network

A Deep Architecture: Multi-Layer Perceptron

Output Layer

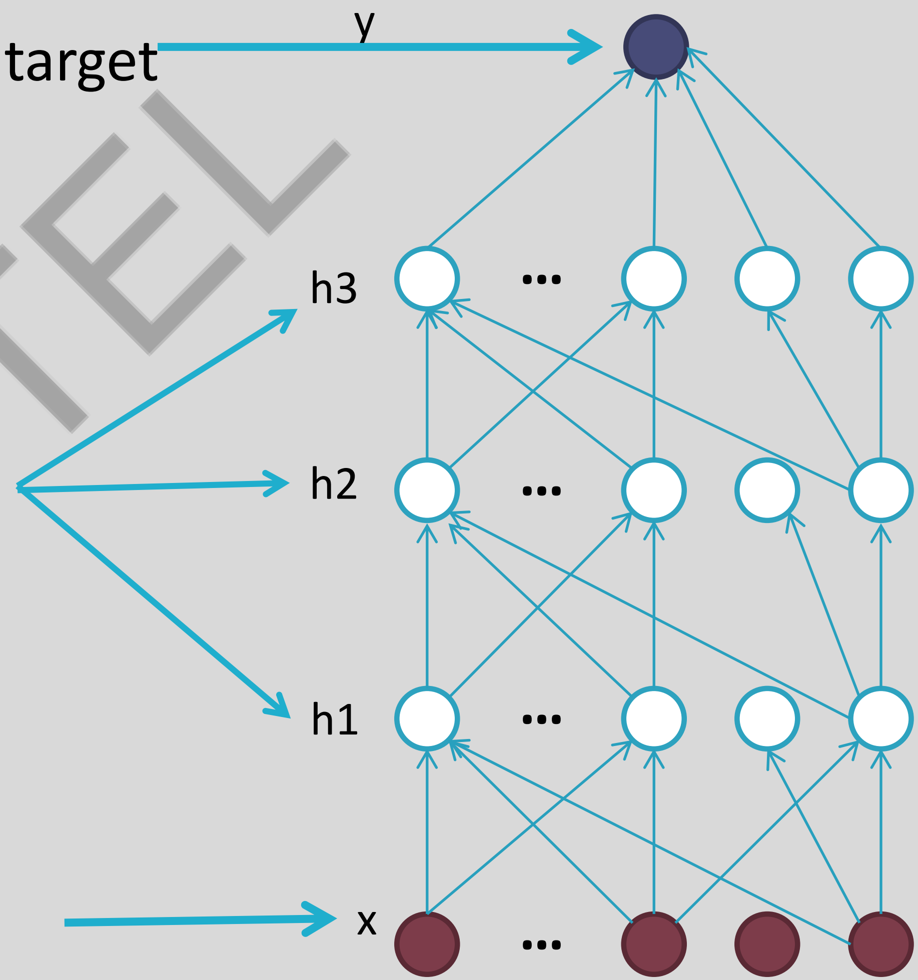
Here predicting a supervised target

Hidden layers

These learn more abstract representations as you head up

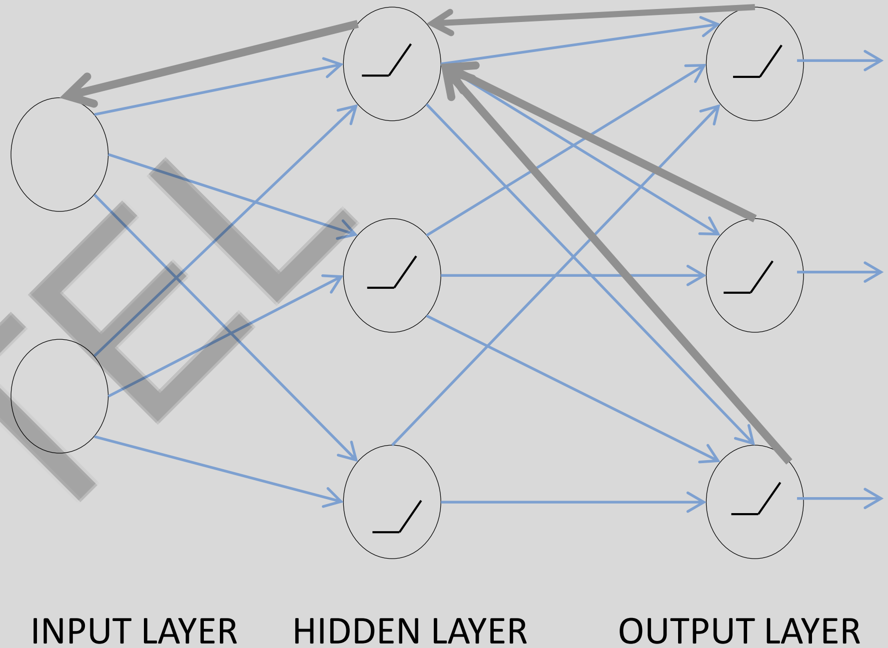
Input layer

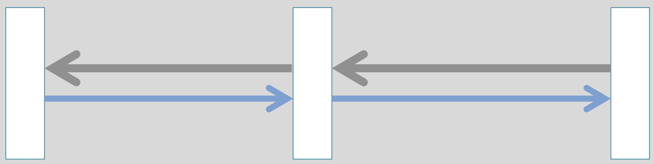
Raw sensory inputs



A Neural Network

- Training : Back Propagation of Error
  - Calculate total error at the top
  - Calculate contributions to error at each step going backwards
  - The weights are modified as the error is propagated





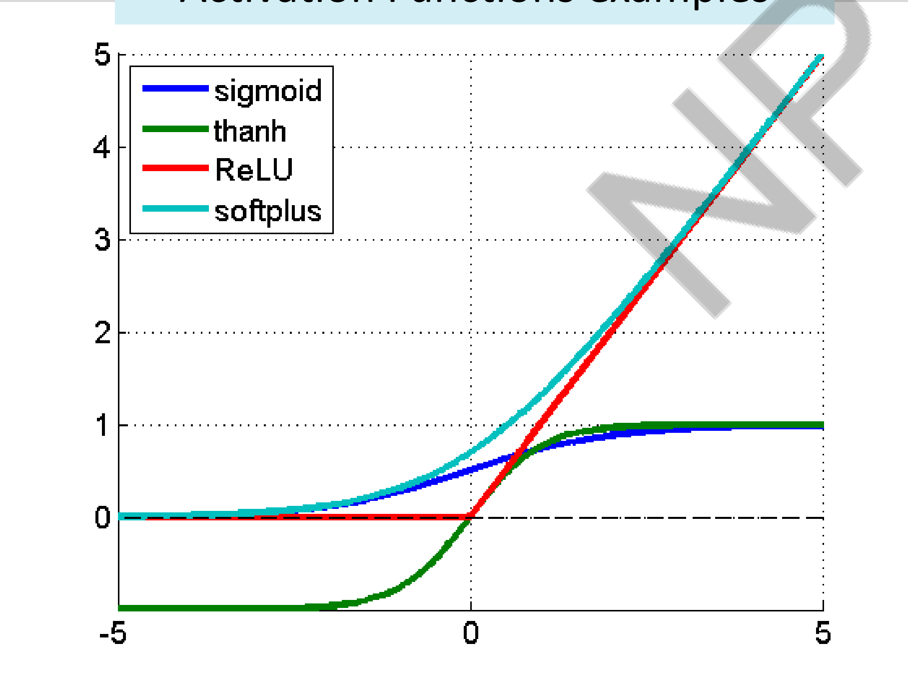
Training Deep Networks

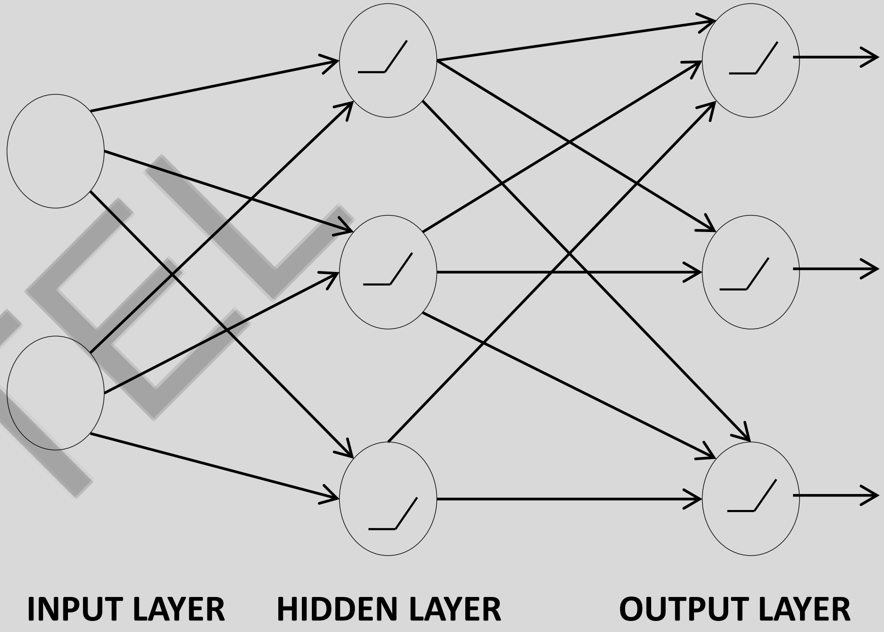
- Difficulties of supervised training of deep networks
  1. Early layers of MLP do not get trained well
     - Diffusion of Gradient – error attenuates as it propagates to earlier layers
     - Leads to very slow training
     - the error to earlier layers drops quickly as the top layers "mostly" solve the task
  2. Often not enough labeled data available while there may be lots of unlabeled data
  3. Deep networks tend to have more local minima problems than shallow networks during supervised training

Training of neural networks

- Forward Propagation :
  - Sum inputs, produce activation
  - feed-forward

Activation Functions examples





Activation Functions

Non-linearity

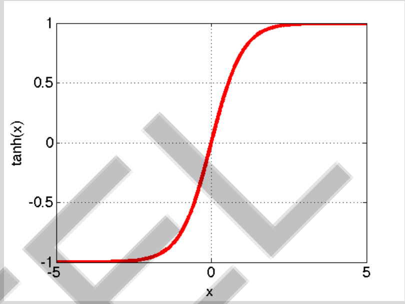
- $\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

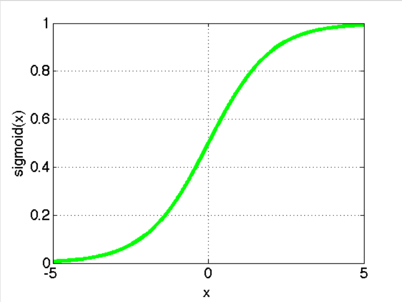
- $\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$

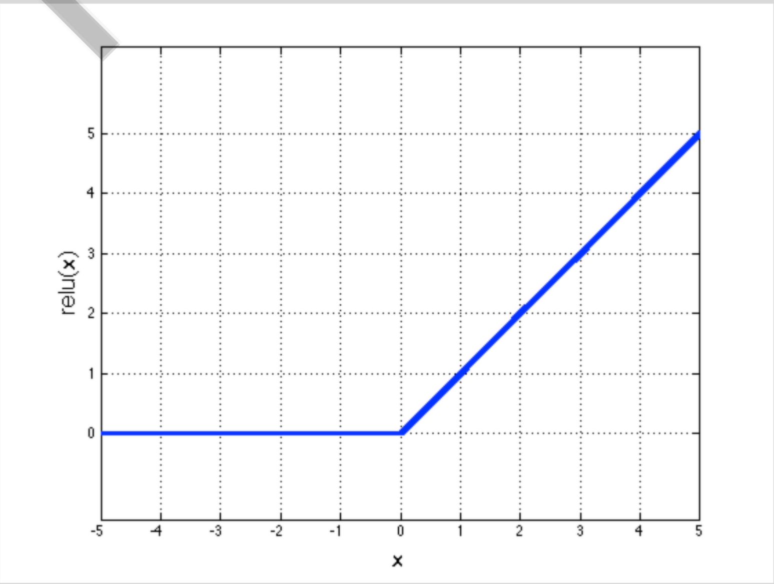
- Rectified linear
  relu(x) = max(0,x)

- Simplifies backprop
- Makes learning faster
- Make feature sparse

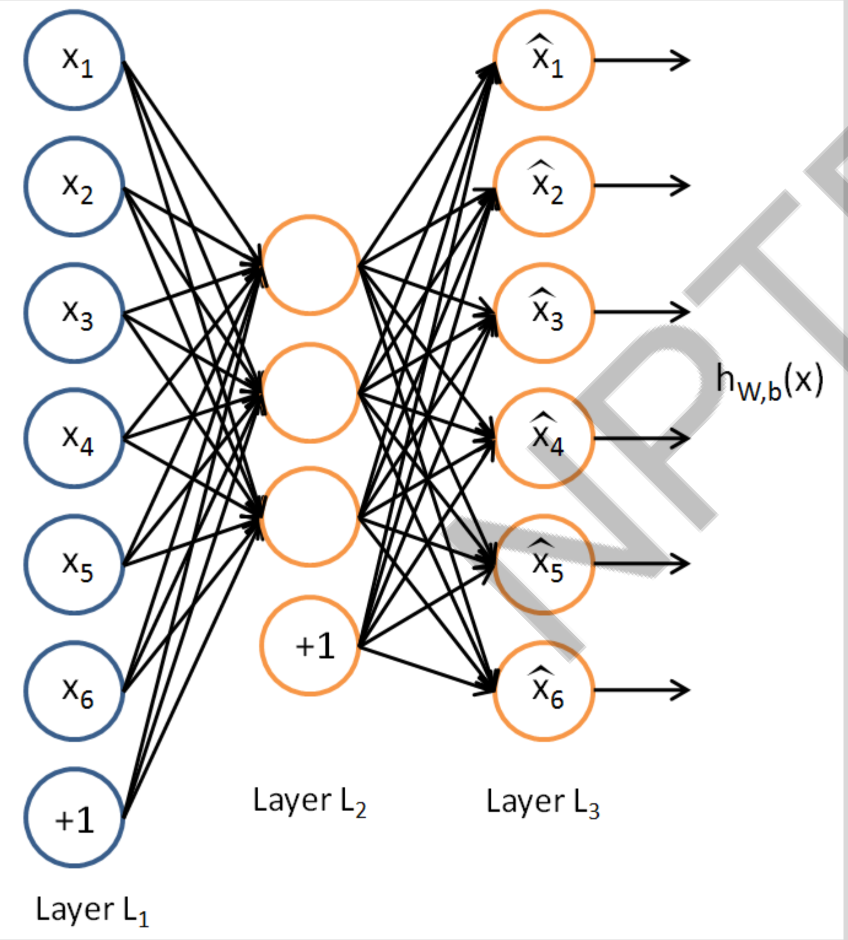
→ Preferred option







Autoencoder



Unlabeled training examples set $\{x^{(1)}, x^{(2)}, x^{(3)} \ ...\}, \ x^{(i)} \in \mathbb{R}^n$

Set the target values to be equal to the inputs. $y^{(i)} = x^{(i)}$

Network is trained to output the input (learn identify function).

$$h_{w,b}(x) \approx x$$

Solution may be trivial!

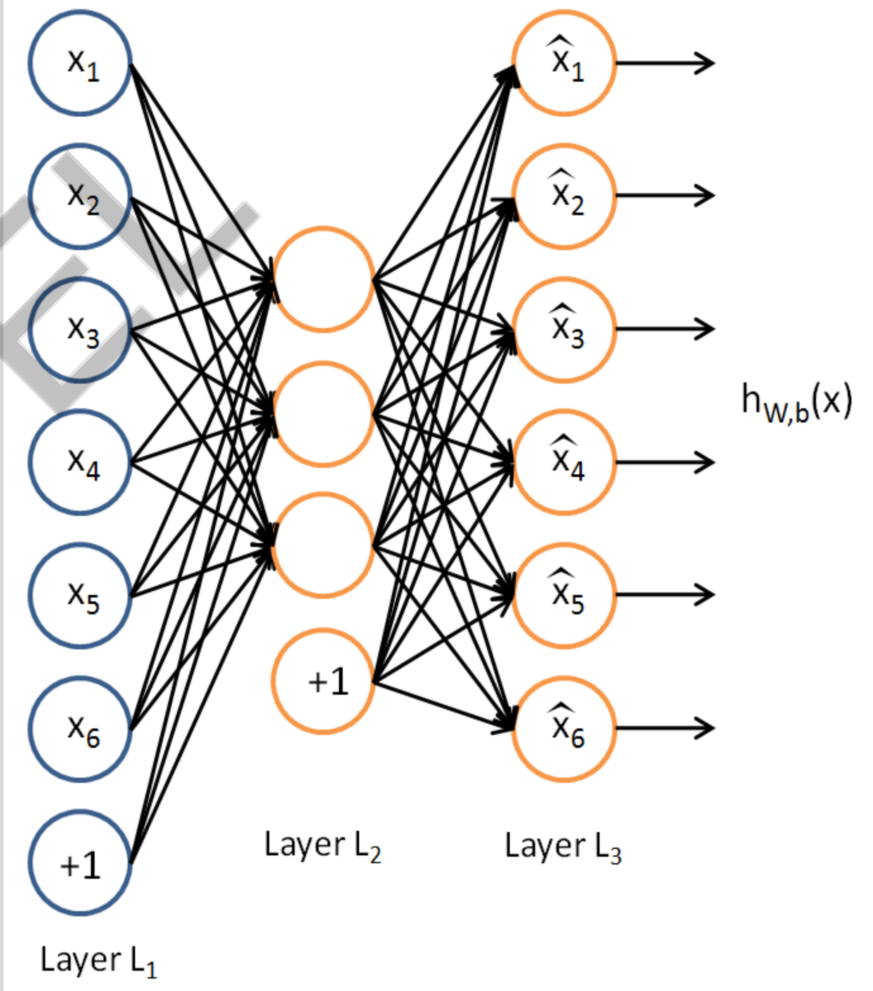
Autoencoders and sparsity

1. Place constraints on the network, like limiting the number of hidden units, to discover interesting structure about the data.
2. Impose sparsity constraint.

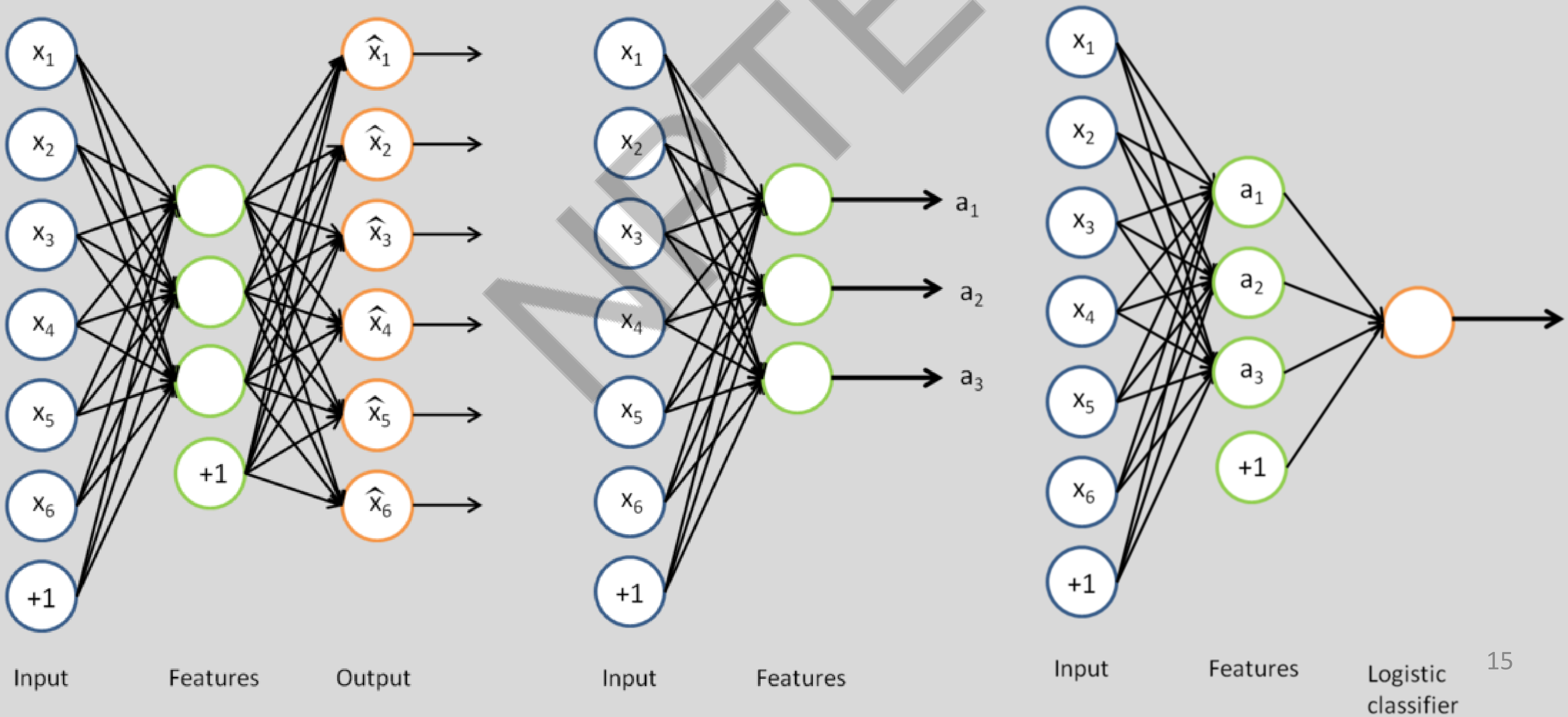
a neuron is “active” if its output value is close to 1

It is “inactive” if its output value is close to 0.

constrain the neurons to be inactive most of the time.

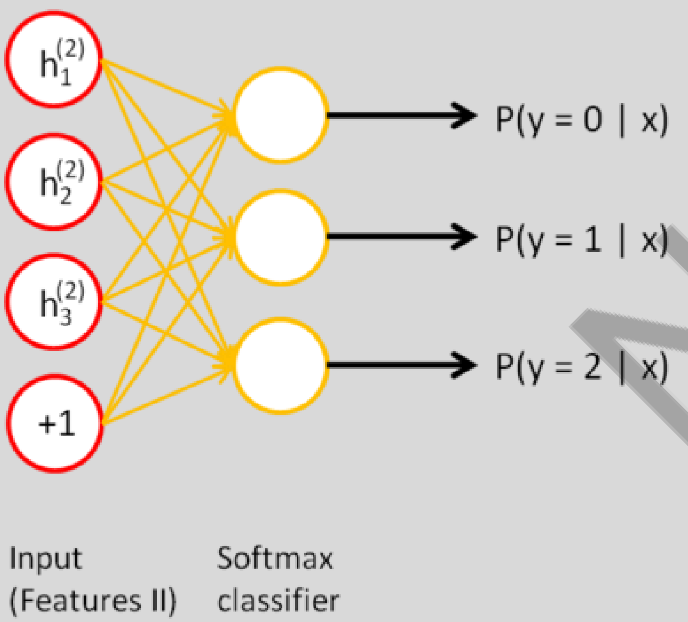


Auto-Encoders

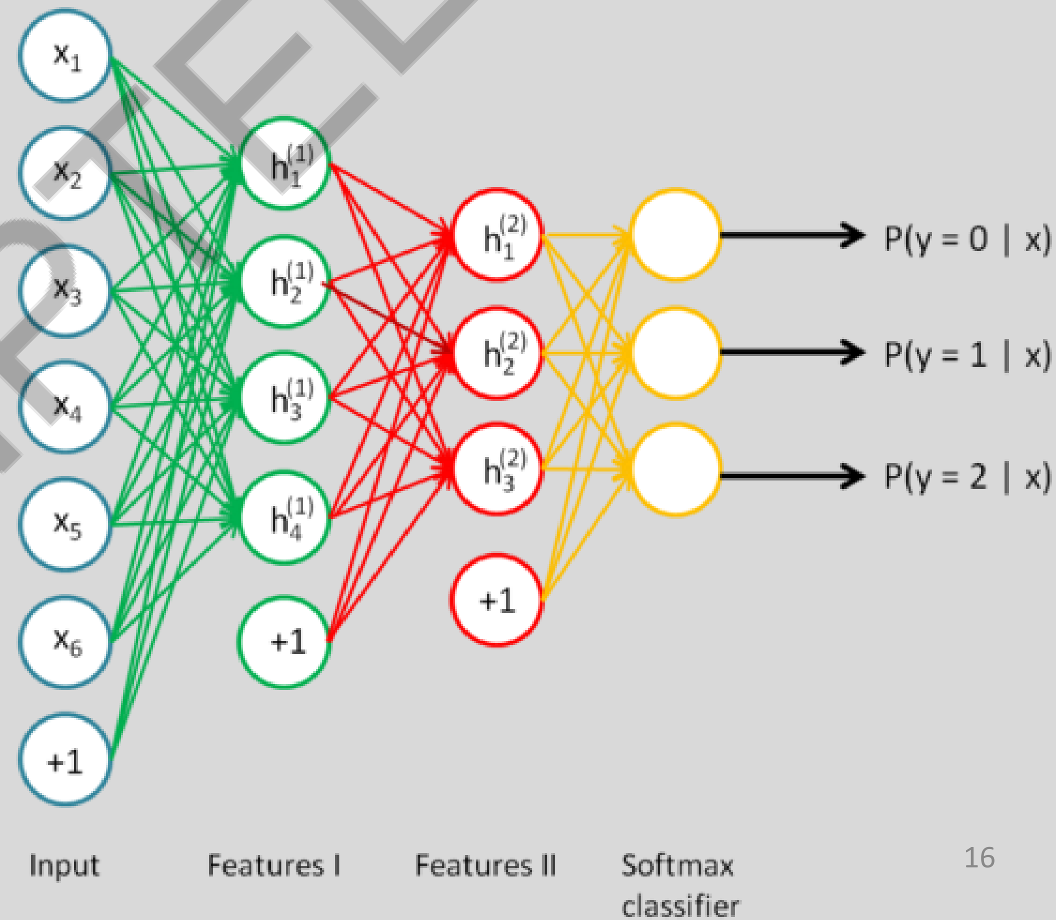


Stacked Auto-Encoders

- Do supervised training on the last layer using final features
- Then do supervised training on the entire network to fine- tune all weights



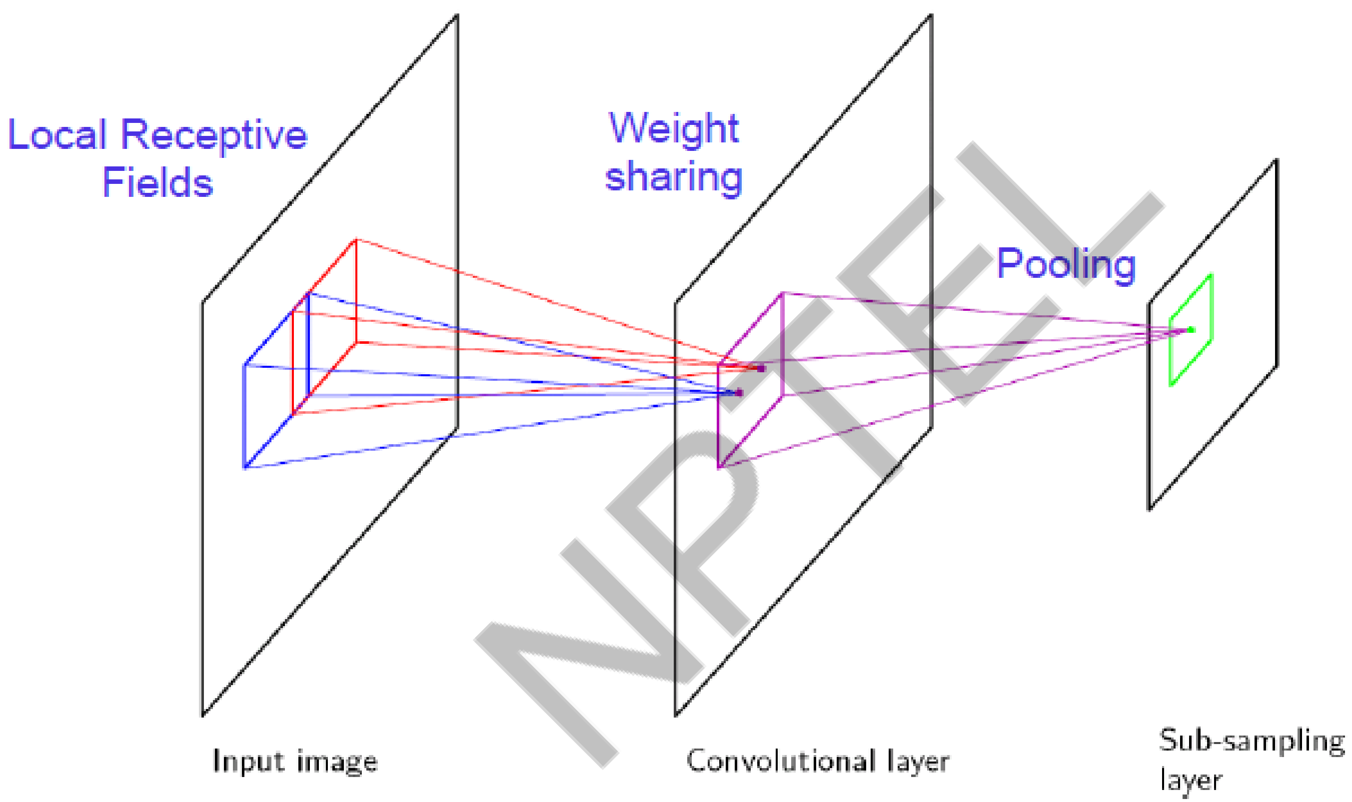
$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$



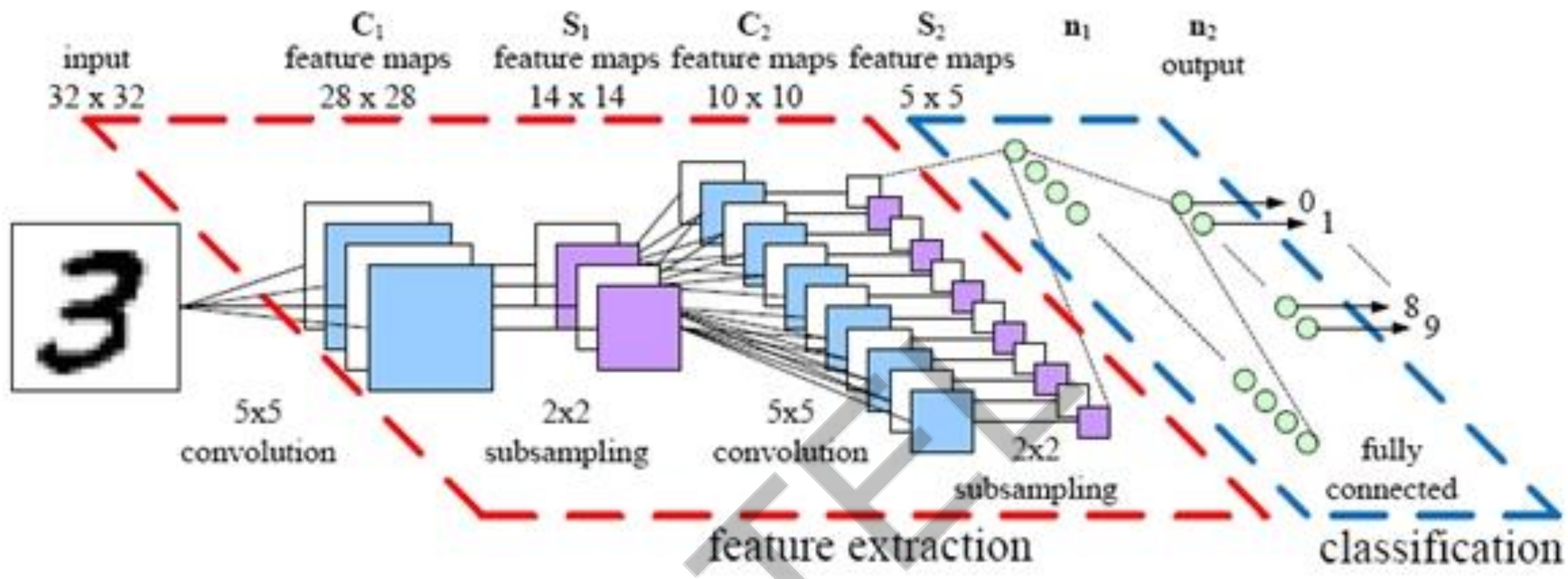
Convolutional Neural netwoks

- A CNN consists of a number of convolutional and subsampling layers.
- Input to a convolutional layer is a m x m x r image where m x m is the height and width of the image and r is the number of channels, e.g. an RGB image has r=3
- Convolutional layer will have k filters (or kernels)
- size n x n x q
- n is smaller than the dimension of the image and,
- q can either be the same as the number of channels r or smaller and may vary for each kernel

Convolutional Neural Networks

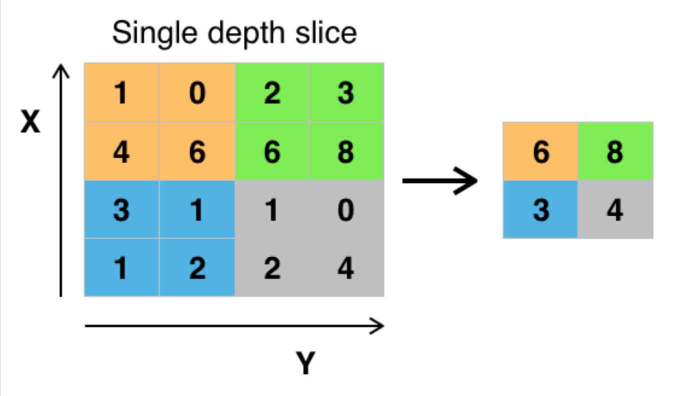


Convolutional layers consist of a rectangular grid of neurons
Each neuron takes inputs from a rectangular section of the previous layer
the weights for this rectangular section are the same for each neuron in the convolutional layer.



Pooling: Using features obtained after Convolution for Classification

The pooling layer takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block : max, average, etc.

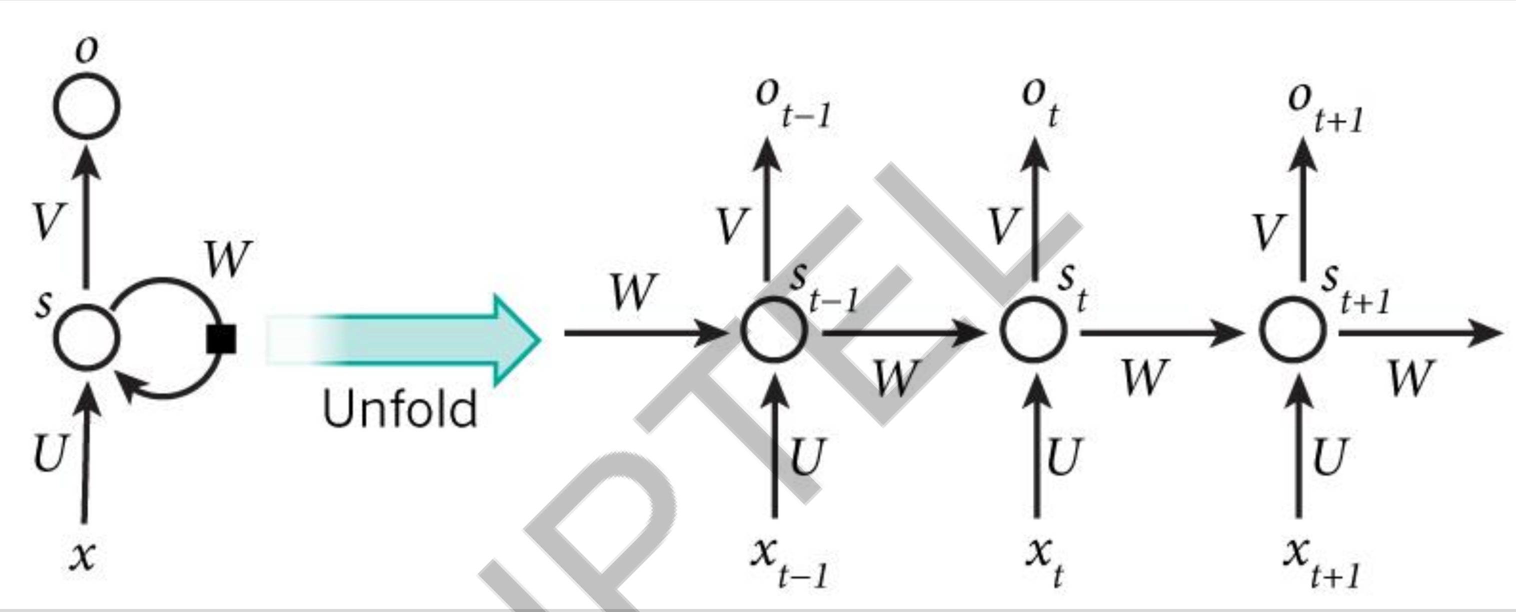


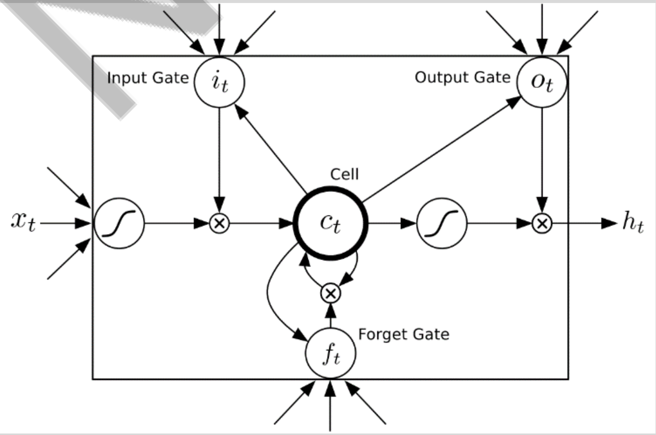
CNN properties

- CNN takes advantage of the sub-structure of the input
- Achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features.

- CNN are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

Recurrent Neural Network (RNN)





Thank You