SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER APPLICATIONS

DATA SCIENCE

II YEAR - III SEM

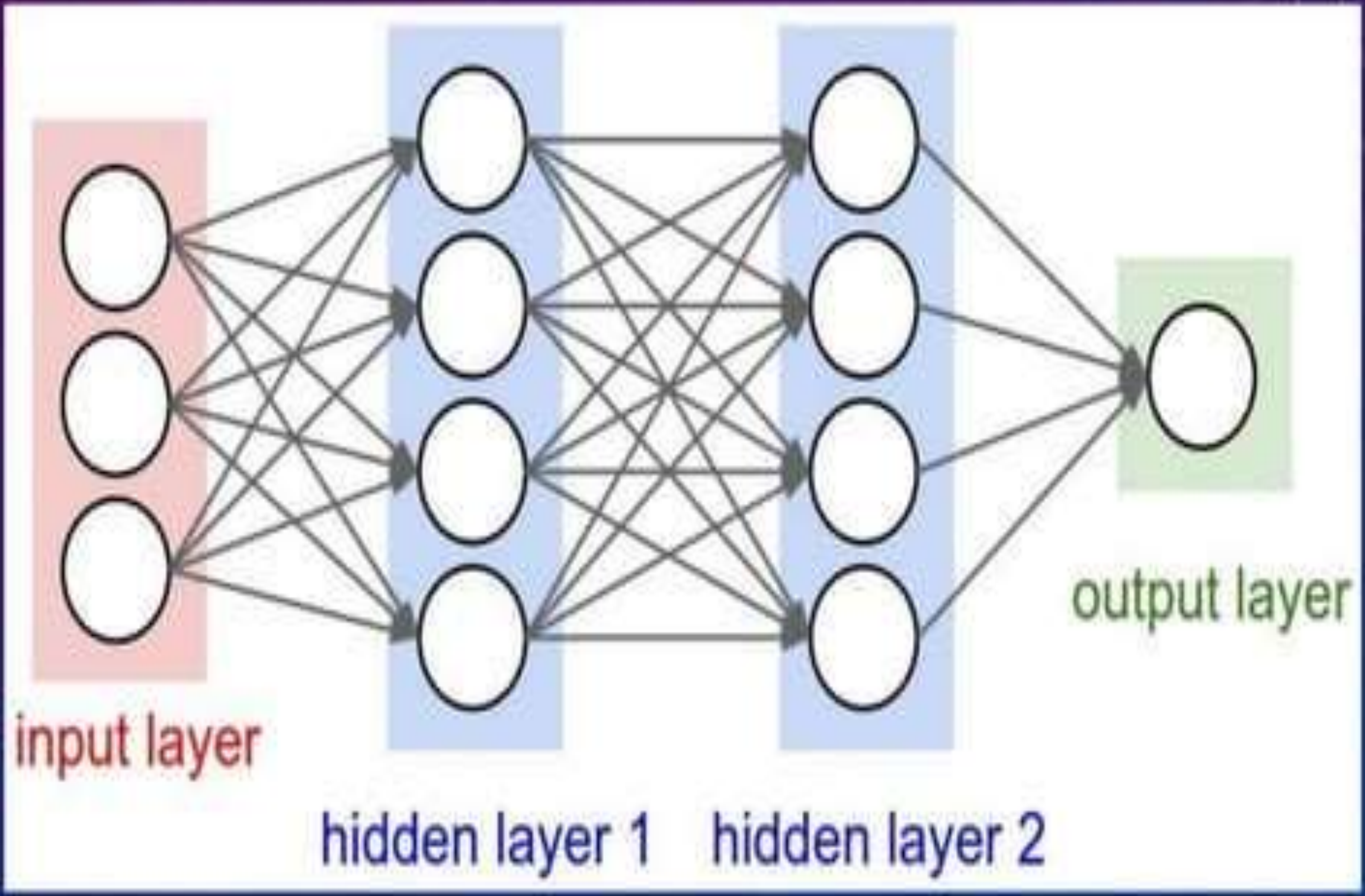
UNIT – IV DEEP LEARNING

TOPIC : DEEP FEEDFORWARD NETWORKS AND REGULARIZATION

OVERVIEW

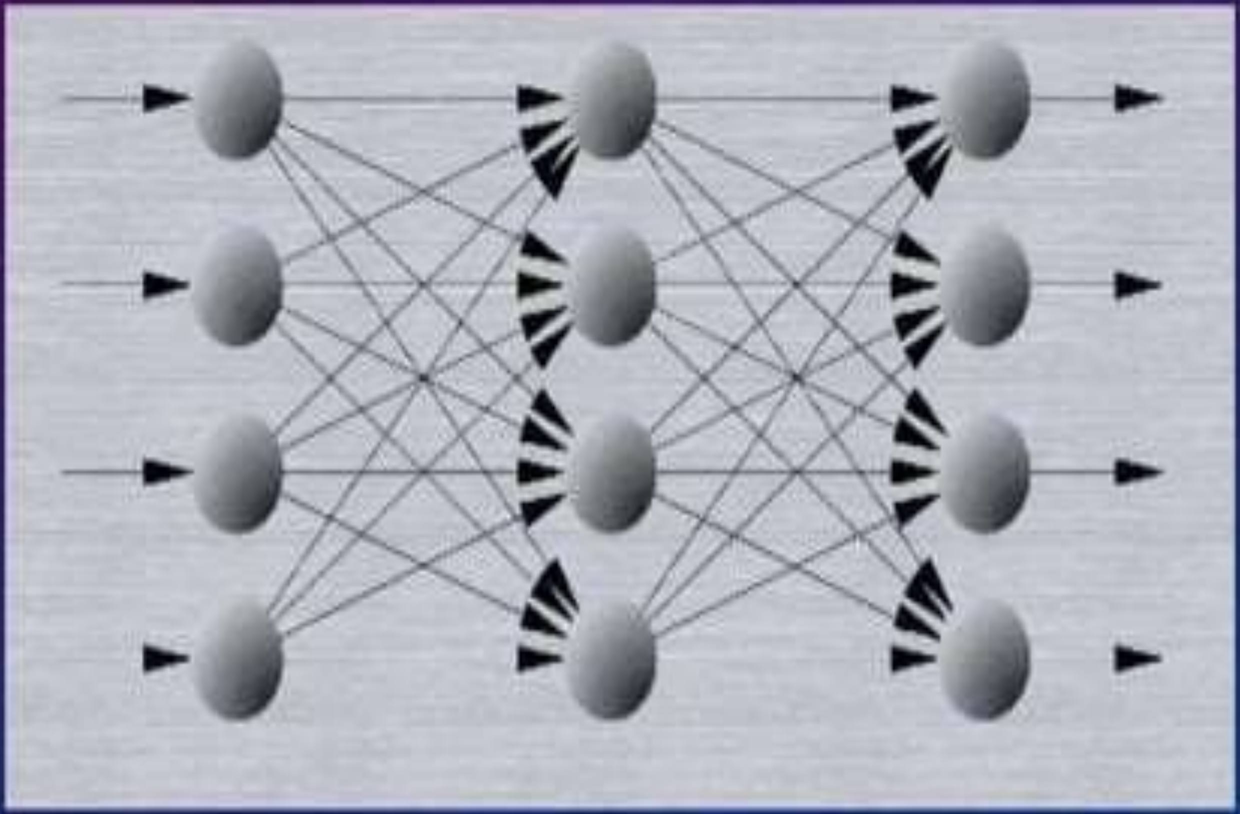
- Neural network
  - Perceptron
  - Activation functions
  - Back-propagation
- Regularization
  - L2/L1/elastic
  - Dropout
  - Batch normalization
  - Data augmentation
  - Early stopping

FEEDFORWARD NETWORK

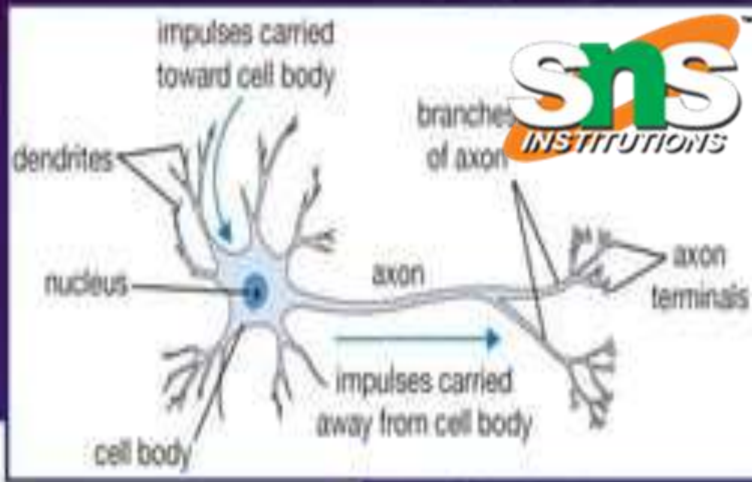


"3-layer neural net" or "2-hidden-layers neural net"

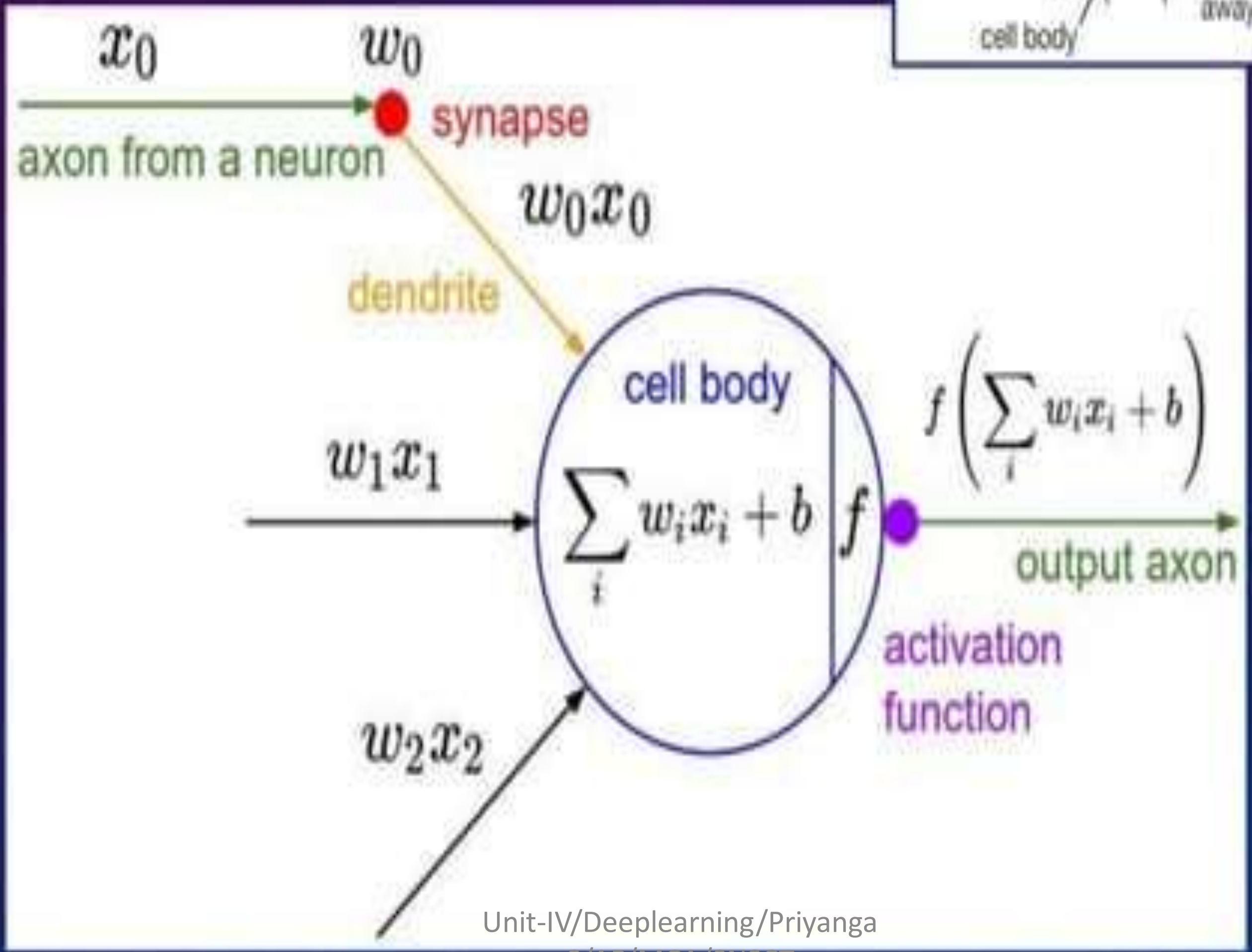
FEEDFORWARD NETWORK (ANIMATION)



NEURON (UNIT)



impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body



x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$f\left(\sum_i w_i x_i + b\right)$
$w_1 x_1$
$\sum_i w_i x_i + b$
f
output axon
activation function
$w_2 x_2$

PERCEPTRON FORWARD PASS

Output = f(

(2*0.1)+

(3*0.5)+

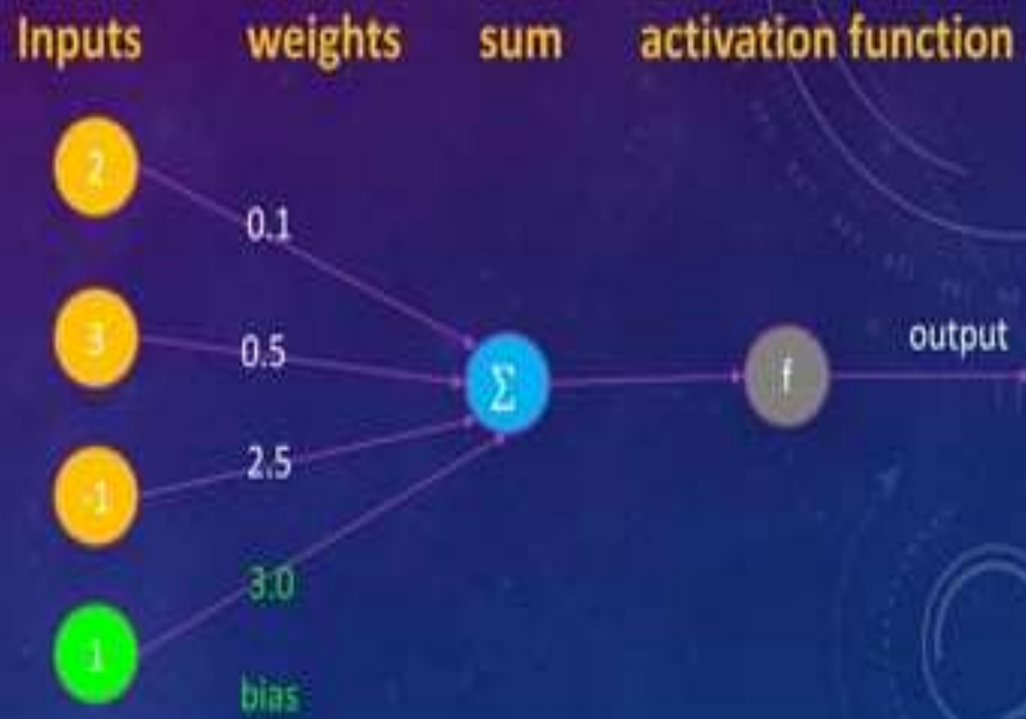
(-1*2.5)+

(1*3.0)

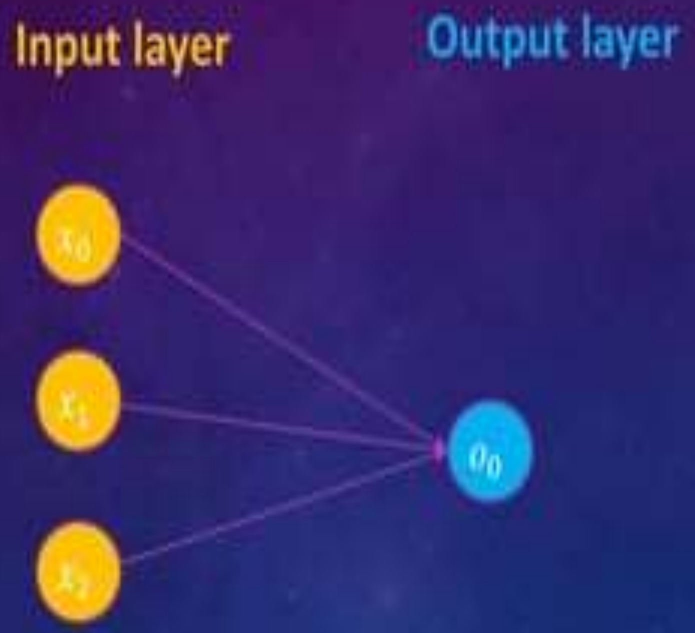
)

PERCEPTRON FORWARD PASS

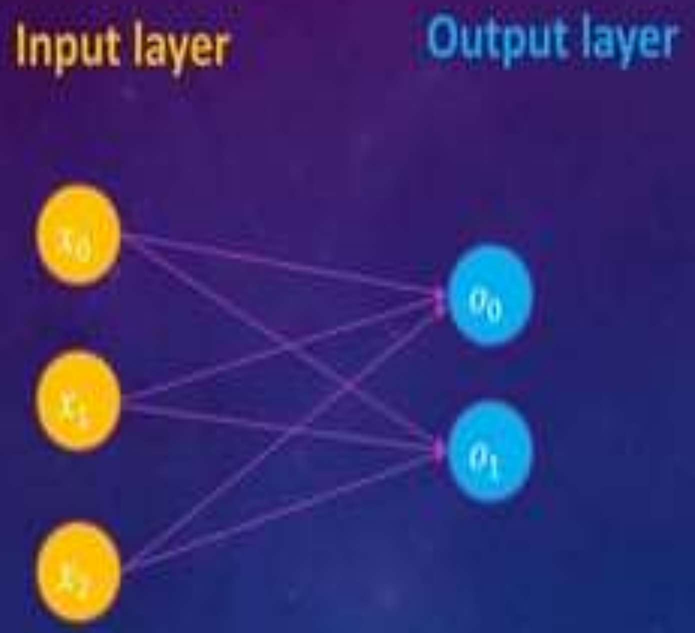
$$\text{Output} = f(2.2)$$
$$= \sigma(2.2)$$
$$= \frac{1}{1+e^{-2.2}} = 0.90$$



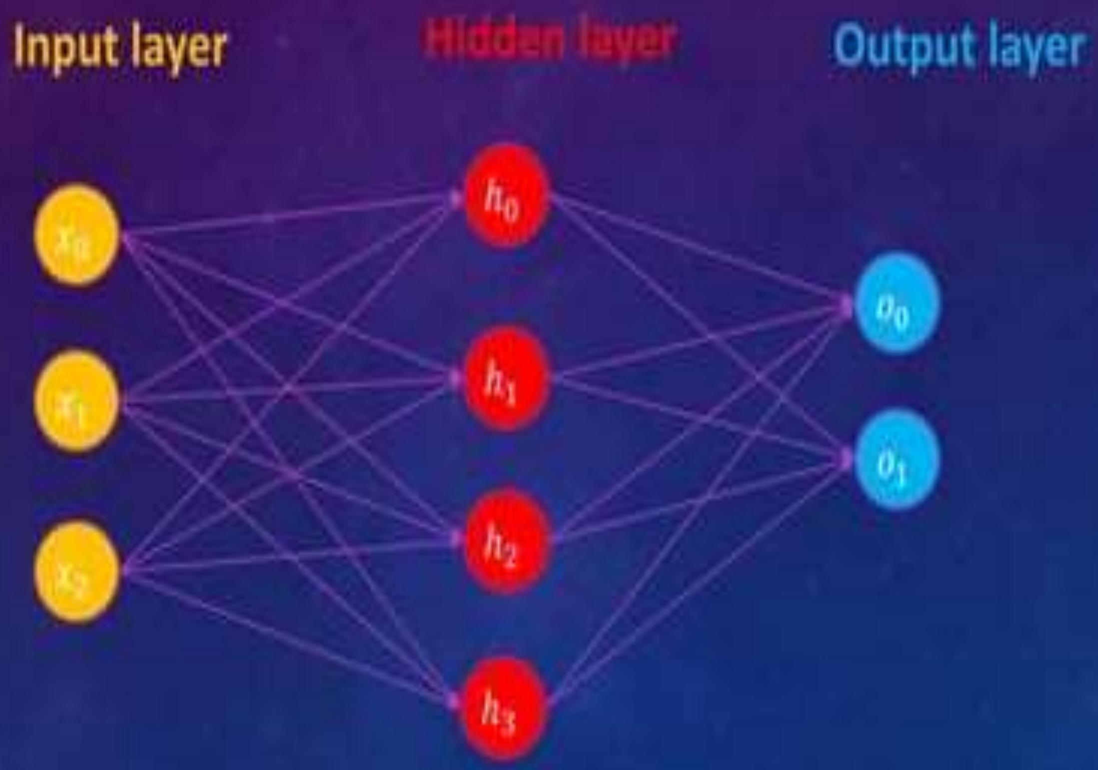
MULTI-OUTPUT PERCEPTRON



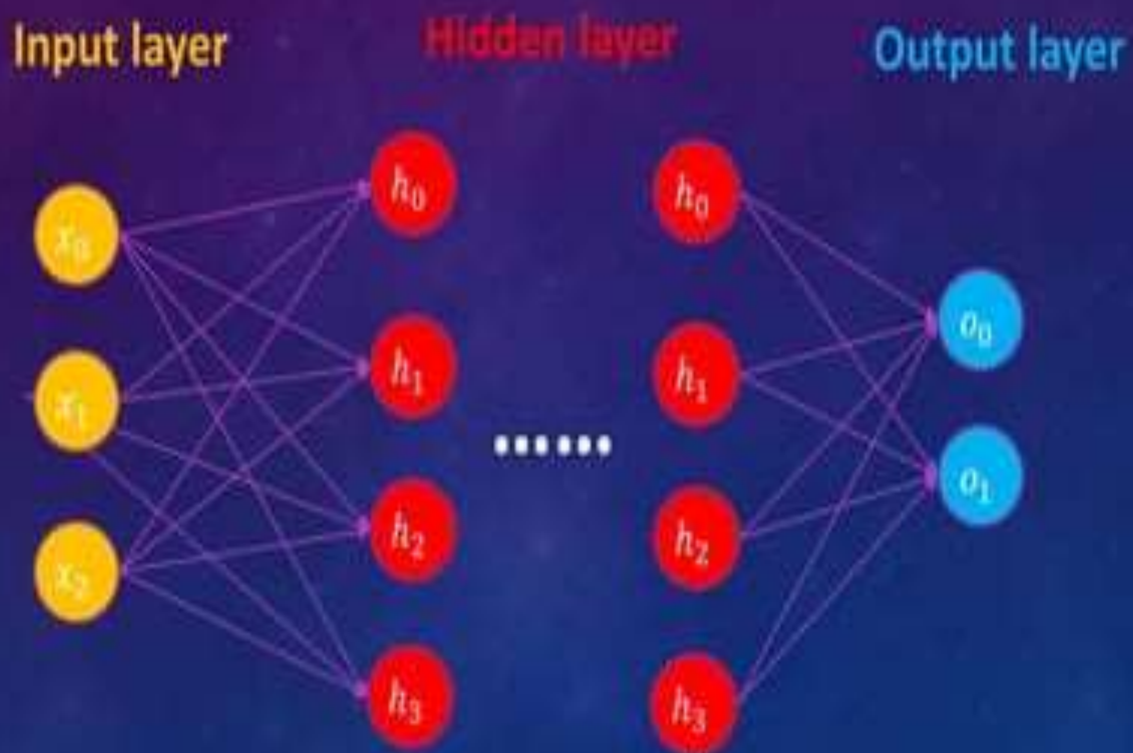
MULTI-OUTPUT PERCEPTRON

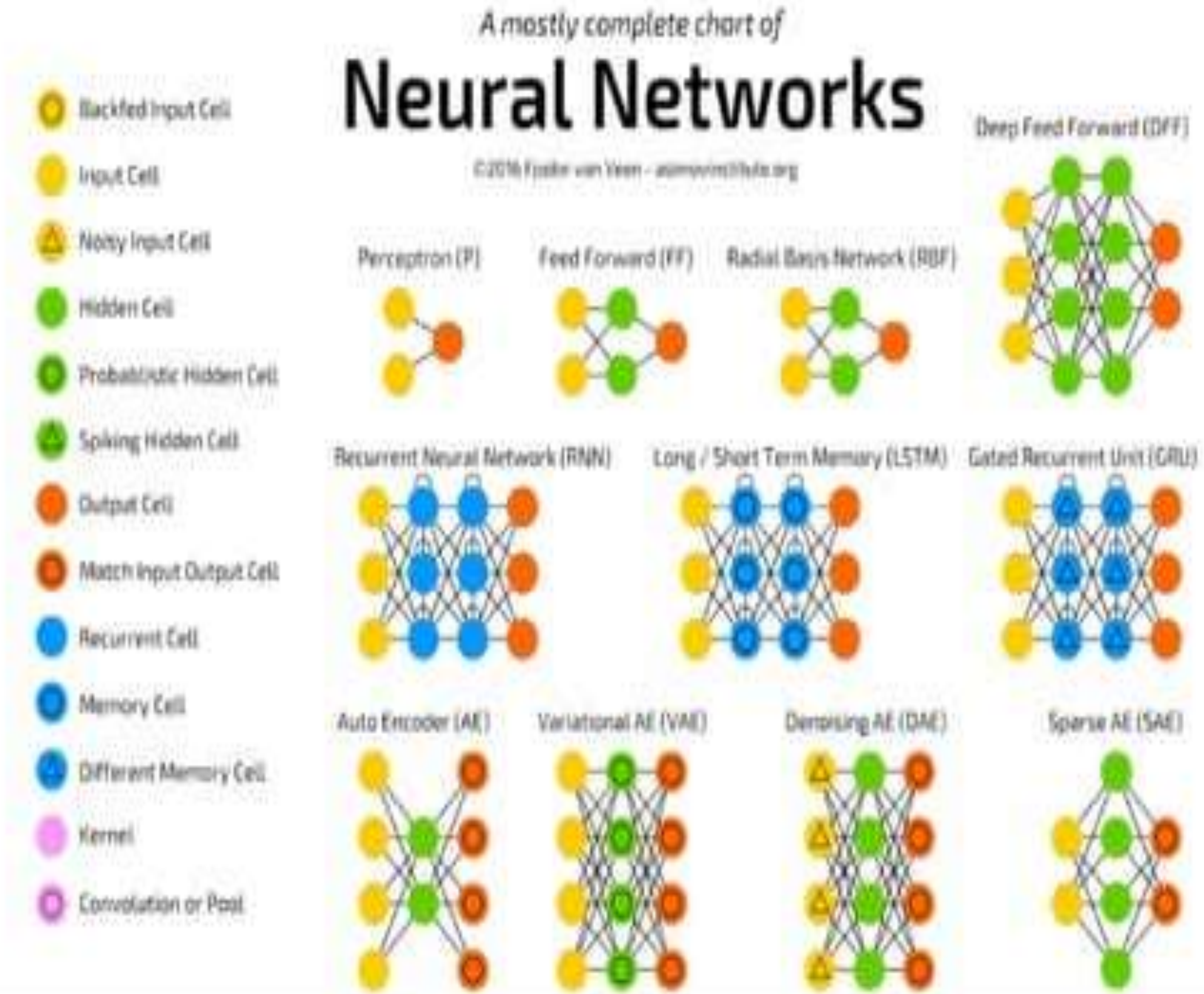


MULTI-LAYER PERCEPTRON (MLP)



DEEP NEURAL NETWORK





http://www.asimovinstitute.org/neural-network-zoo/

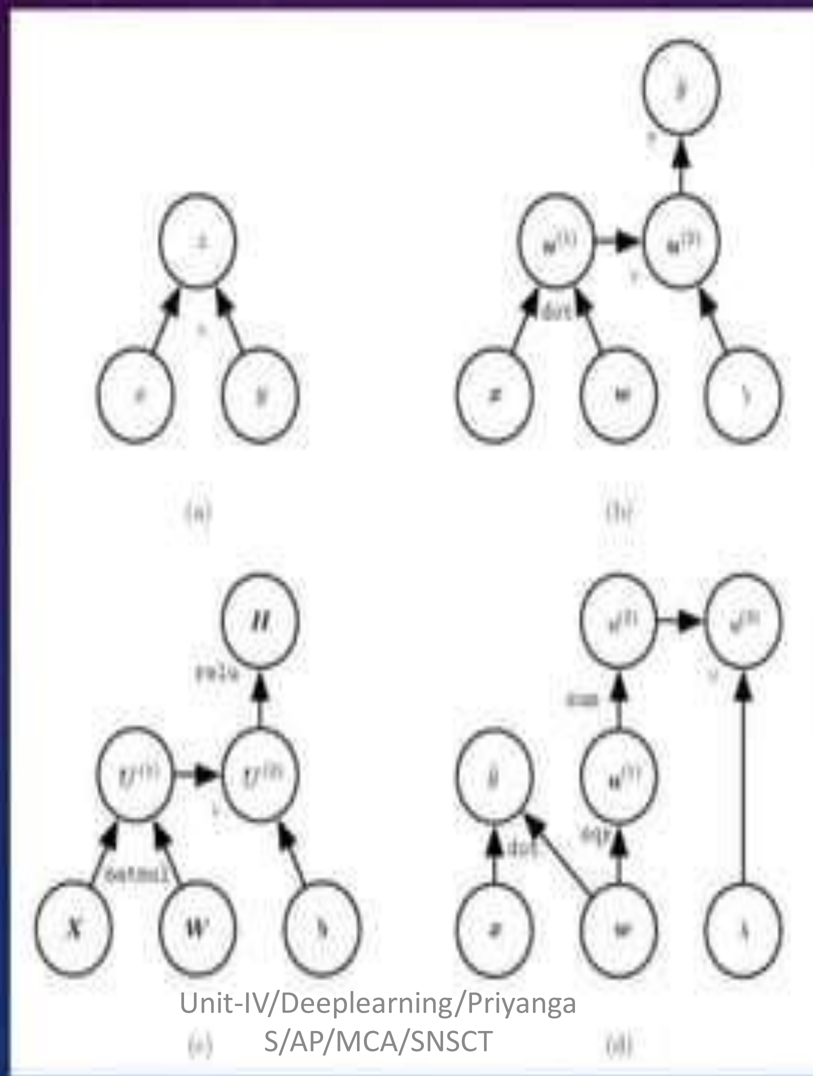
UNIVERSAL APPROXIMATION THEOREM

"A feedforward network with a linear output layer and at least one hidden layer with any 'squashing' activation function (such as the logistic sigmoid) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units."

- ----- Hornik et al., Cybenko, 1989

COMPUTATIONAL GRAPHS

Z=x*y



$$y = \sigma(wx + b)$$

$$\mathrm{H} = relu(WX + b) = \max(0, WX + b)$$

$$y = wx$$

$$u^{(3)} = \lambda \sum \sqrt{w}$$

LOSS FUNCTION

- A loss function (cost function) tells us how good our current model is, or how far away our model to the real answer.

$$L(w) = \underbrace{\frac{1}{N}\sum_{i}^{N}}_{\text{N = \# examples}} loss\,(\underbrace{f(x^{(i)}; w)}_{\text{predicted}}, \underbrace{y^{(i)}}_{\text{actual}})$$

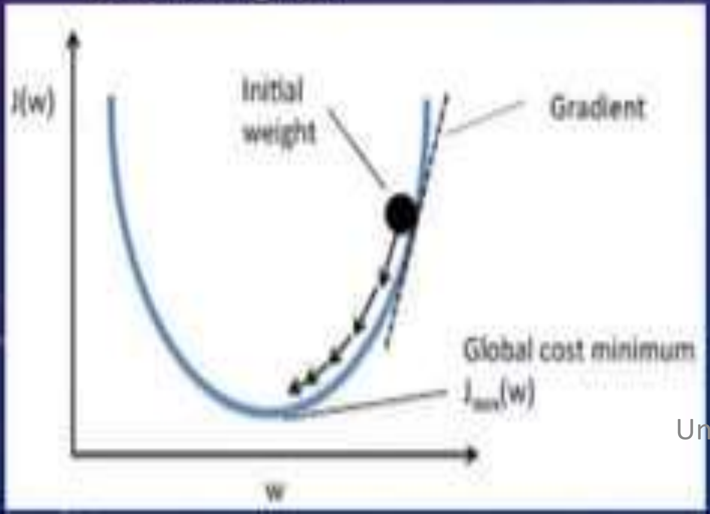
- Hinge loss
- Softmax loss
- Mean Squared Error (L2 loss) → Regression $L(w) = \frac{1}{N}\sum_{i}^{N}\left(f(x^{(i)}; w) - y^{(i)}\right)^2$
- Cross entropy Loss → Classification $L(w) = \frac{1}{N}\sum_{i}^{N}[\, y^{(i)} \log f(x^{(i)}; w) + (1 - y^{(i)}) \log\left(1 - f(x^{(i)}; w)\right)]$
- ...

GRADIENT DESCENT

- Designing and training a neural network is not much different from training any other machine learning model with gradient descent: use Calculus to get derivatives of the loss function respect to each parameter.

$$w_j = w_j - \alpha \frac{\partial L(w)}{\partial w_j}$$

α is learning rate



GRADIENT DESCENT

- In practice, instead of using all data points, we do
  - Stochastic gradient descent (using 1 sample at each iteration)
  - Mini-Batch gradient descent (using n samples at each iteration)

Problems with SGD:

- If loss changes quickly in one direction and slowly in another → jitter along steep direction
- If loss function has a local minima or saddle point → zero gradient, SGD gets stuck

Solutions:

- SGD + momentum, etc

BACK-PROPAGATION

- It allows the information from the loss to flow backward through the network in order to compute the gradient.

$$x_0 \xrightarrow{W_1} h_0 \xrightarrow{W_2} O_0 \longrightarrow L(w)$$

$$\frac{\partial L(w)}{\partial w_2} =$$

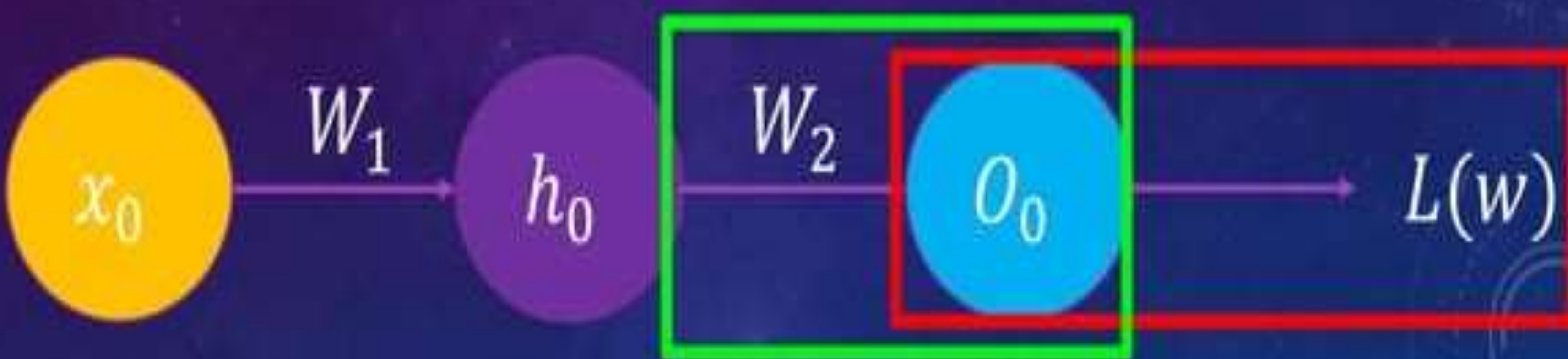
BACK-PROPAGATION

- It allows the information from the loss to flow backward through the network in order to compute the gradient.

$$\frac{\partial L(w)}{\partial w_2} = \frac{\partial L(w)}{\partial O_0} *$$

BACK-PROPAGATION

- It allows the information from the loss to flow backward through the network in order to compute the gradient.



$$\frac{\partial L(w)}{\partial w_2} = \frac{\partial L(w)}{\partial O_0} * \frac{\partial O_0}{\partial w_2}$$

Chain rule

BACK-PROPAGATION

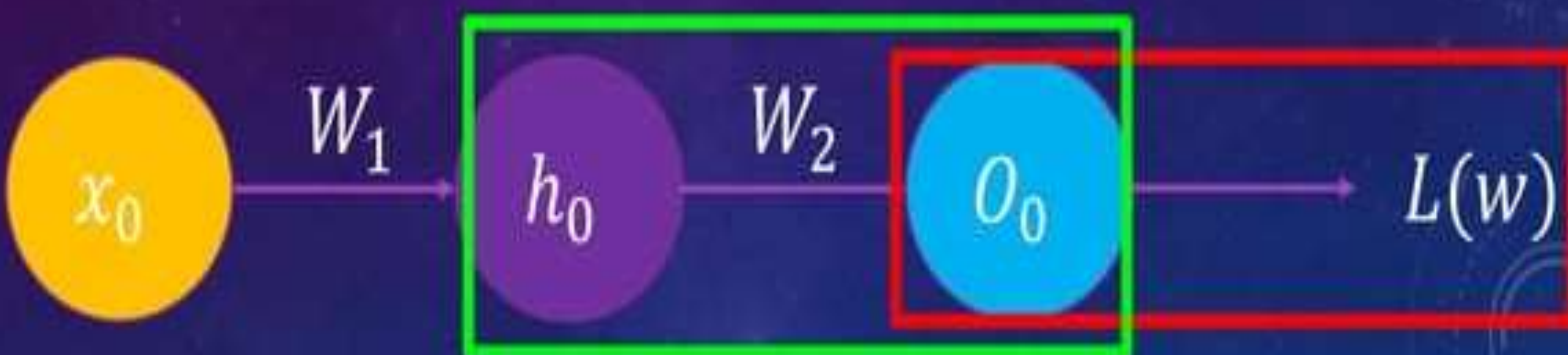
- It allows the information from the loss to flow backward through the network in order to compute the gradient.

$$x_0 \xrightarrow{W_1} h_0 \xrightarrow{W_2} O_0 \longrightarrow L(w)$$

$$\frac{\partial L(w)}{\partial w_1} =$$

BACK-PROPAGATION

- It allows the information from the loss to flow backward through the network in order to compute the gradient.

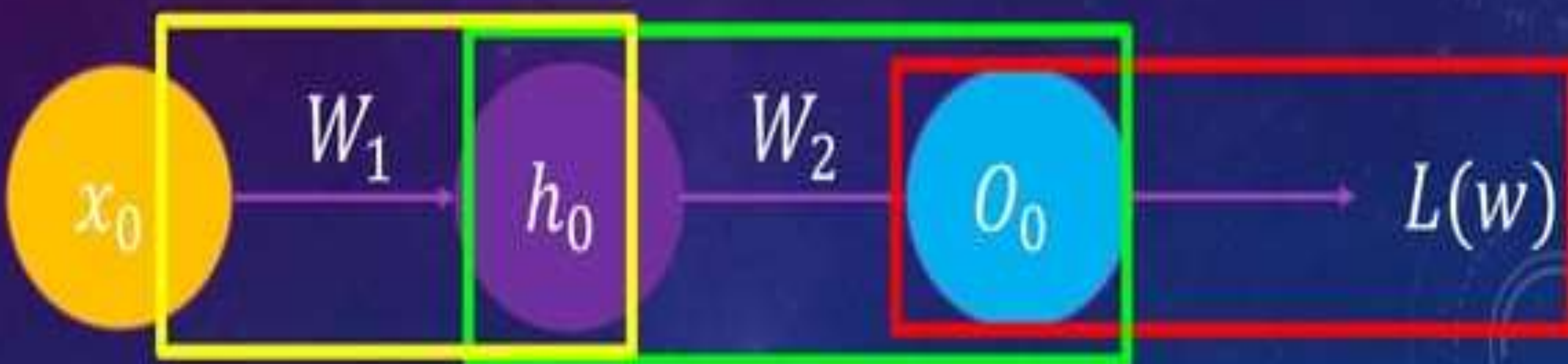


Chain rule

$$\frac{\partial L(w)}{\partial w_1} = \frac{\partial L(w)}{\partial O_0} * \frac{\partial O_0}{\partial h_0} *$$

BACK-PROPAGATION

- It allows the information from the loss to flow backward through the network in order to compute the gradient.



$$\frac{\partial L(w)}{\partial w_1} = \frac{\partial L(w)}{\partial O_0} * \frac{\partial O_0}{\partial h_0} * \frac{\partial h_0}{\partial w_1}$$

Chain rule

BACK-PROPAGATION: SIMPLE EXAMPLE

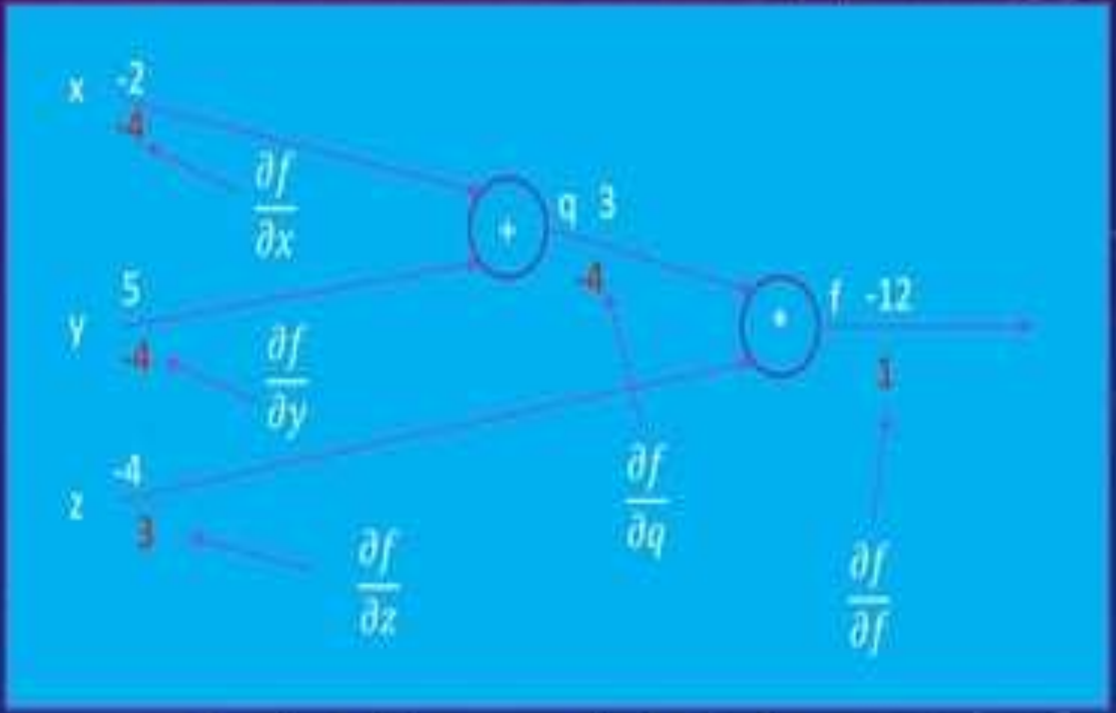
$f(x,y,z) = (x+y)z$

e.g. x = -2, y = 5, z=-4

$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$

f = qz $\quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

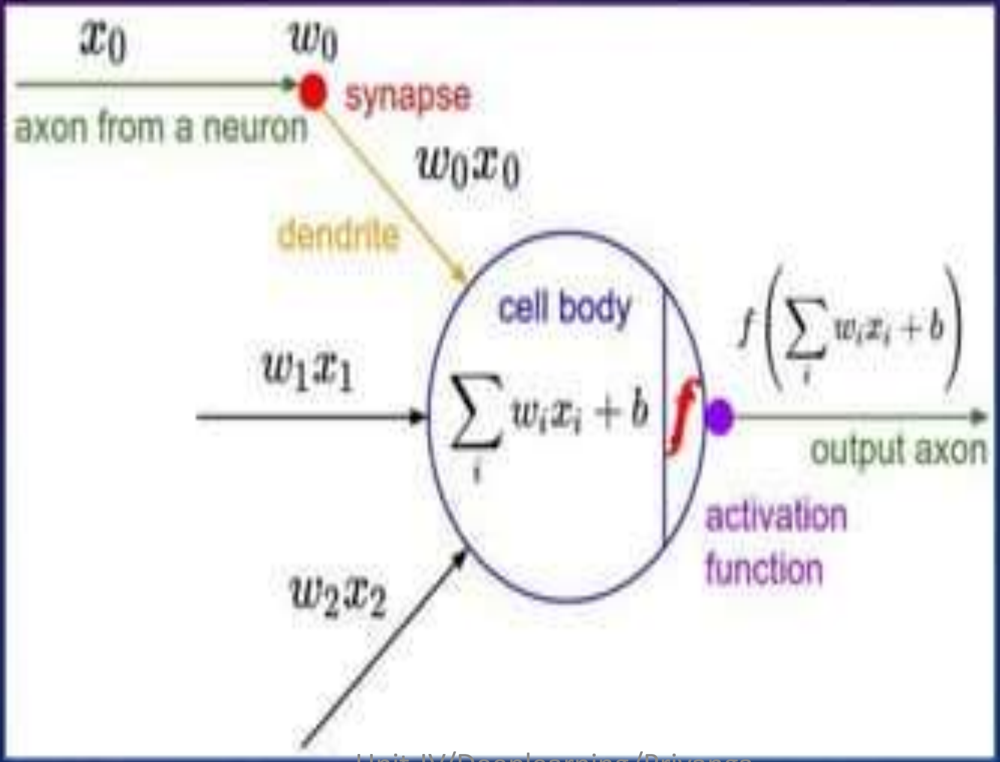


Chain Rule:
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

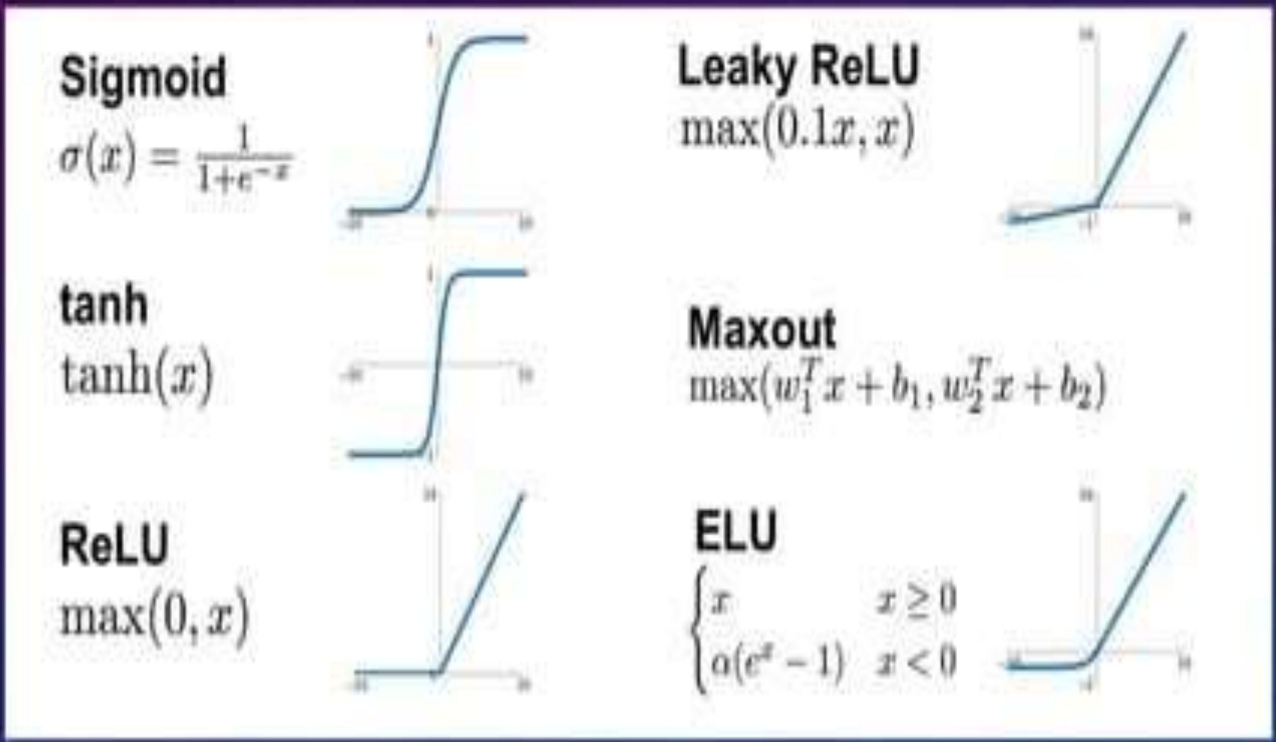
Chain Rule:
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

ACTIVATION FUNCTIONS

Importance of activation functions is to introduce non-linearity into the network.



ACTIVATION FUNCTIONS



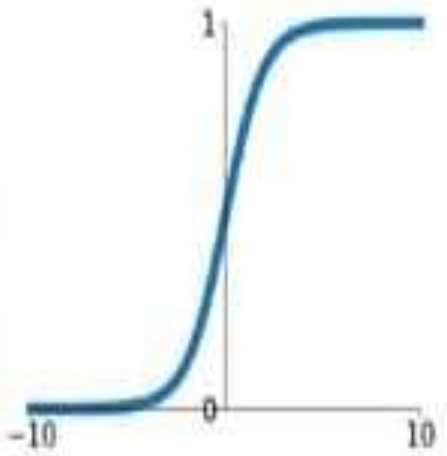
For output layer:

- Sigmoid
- Softmax
- Tanh

For hidden layer:

- ReLU
- LeakyReLU
- ELU

ACTIVATION FUNCTIONS

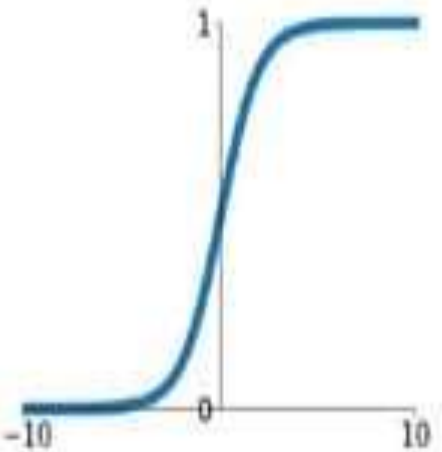


Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

ACTIVATION FUNCTIONS



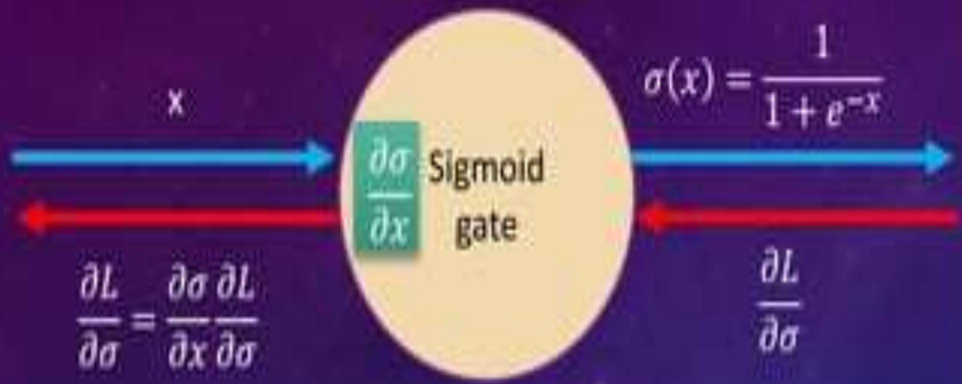
Sigmoid

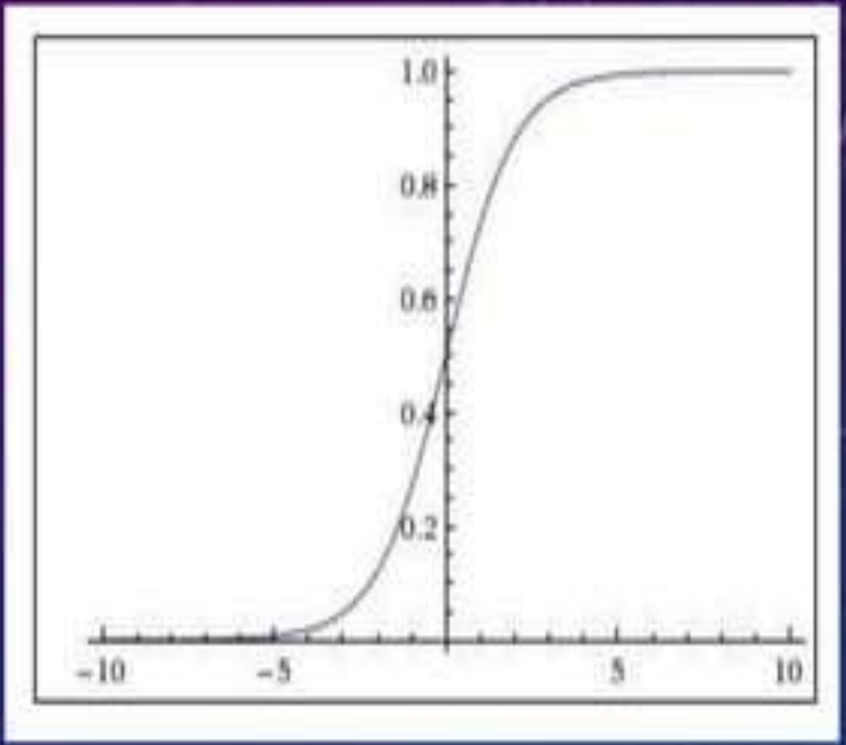
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

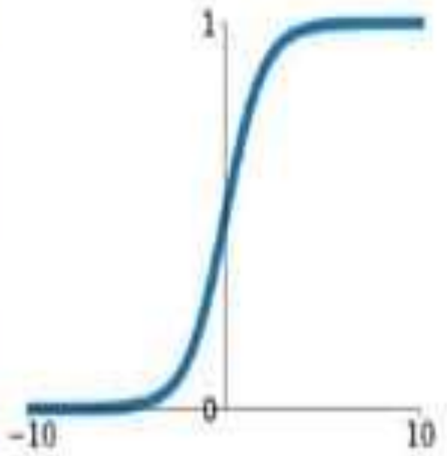
1. Saturated neurons "kill" the gradients





- What happens when x= -10?
- What happens when x = 0?
- What happens when x = 10

ACTIVATION FUNCTIONS



Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i}$$

$$= \frac{\partial L}{\partial f} * x_i$$

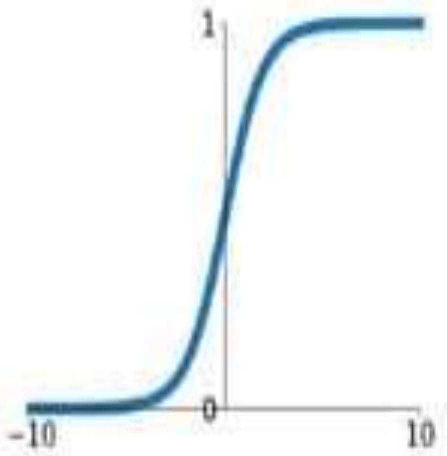
What can we say about the gradients on w?

Always all positive or all negative ☹

(this is also why you want zero-mean data!)

Inefficient!

ACTIVATION FUNCTIONS



Sigmoid

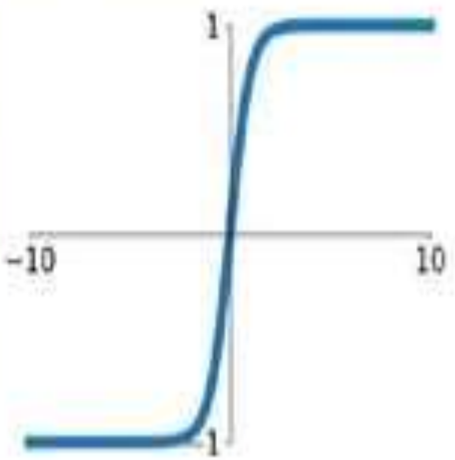
$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. [illegible] compute expensive

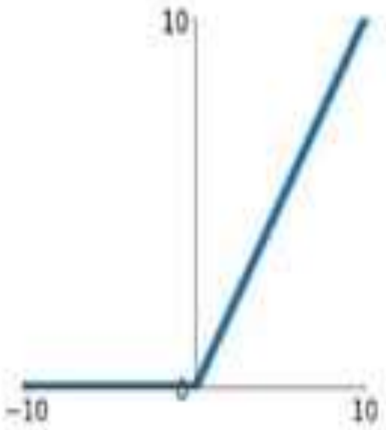
ACTIVATION FUNCTIONS



tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

ACTIVATION FUNCTIONS

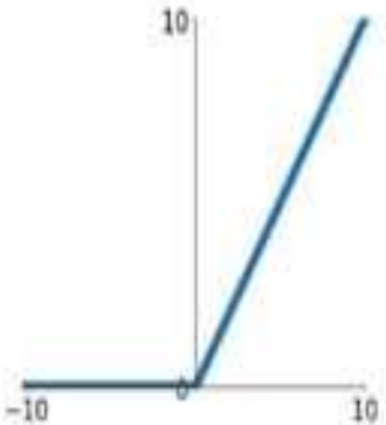


ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

ACTIVATION FUNCTIONS



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance when $x < 0$

People like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

ACTIVATION FUNCTIONS

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

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

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

ACTIVATION FUNCTIONS

Exponential Linear Units (ELU)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

Clevert et al., 2015

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- Computation requires exp()

MAXOUT "NEURON"

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

IN PRACTICE (GOOD RULE OF THUMB)

- For hidden layers:
  - Use ReLU. Be careful with your learning rates
  - Try out Leaky ReLU / Maxout / ELU
  - Try out tanh but don't expect too much
  - Don't use Sigmoid

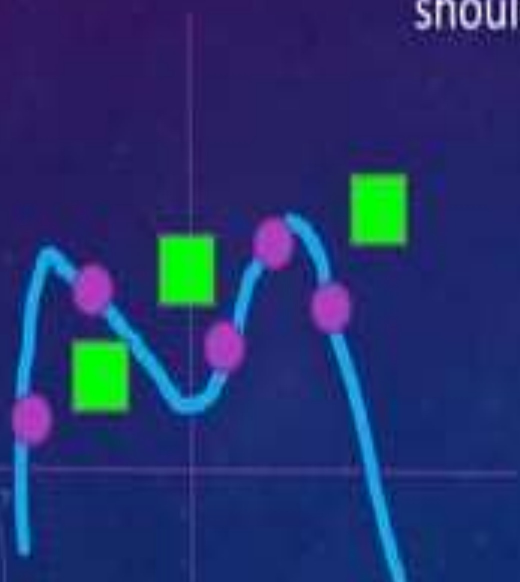
REGULARIZATION

- Regularization is *"any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error"*.

REGULARIZATION

$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$$

Data loss: model predictions should match training data



REGULARIZATION

$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i\left(f\left(x^{(i)}; W\right), y^{(i)}\right) + \lambda R(W)$$

Data loss: model predictions should match training data

Regularization: Model Should be "simple", so it works on test data

Occam's Razor:
"Among competing hypotheses, The simplest is the best"
William of Ockham, 1285-1347

REGULARIZATION

$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i\left(f(x^{(i)}; W), y^{(i)}\right) + \lambda R(W)$$

- In common use:
  - L2 regularization $R(w) = \sum w_j^2$
  - L1 regularization $R(w) = \sum |w_j|$
  - Elastic net (L1 + L2) $R(w) = \sum(\beta w_j^2 + |w_j|)$
  - Dropout
  - Batch normalization
  - Data Augmentation
  - Early Stopping

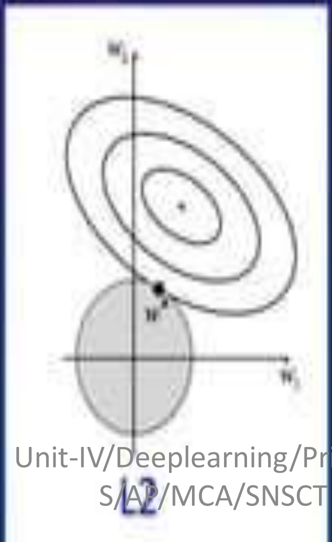
Regularization is a technique designed to counter neural network over-fitting.

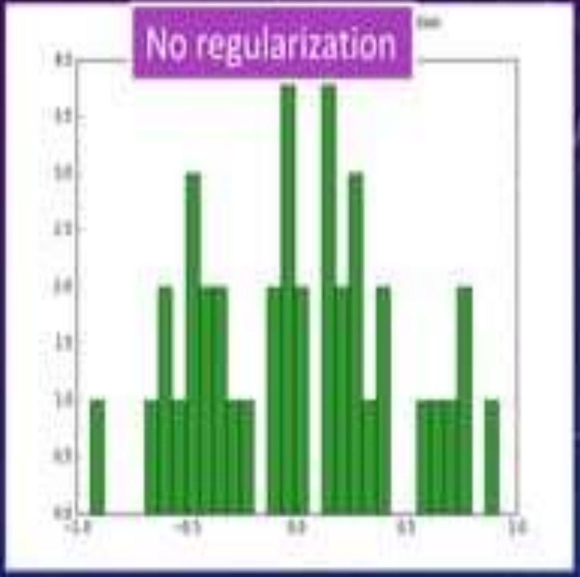
L2 REGULARIZATION

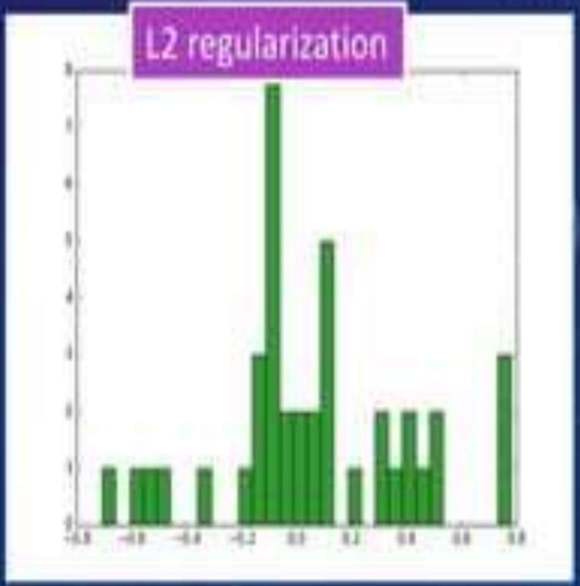
Weights distr

$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i\left(f\left(x^{(i)};W\right), y^{(i)}\right) + \lambda\sum w_j^2$$

- penalizes the square value of the weight (which explains also the "2" from the name).
- tends to drive all the weights to smaller values.



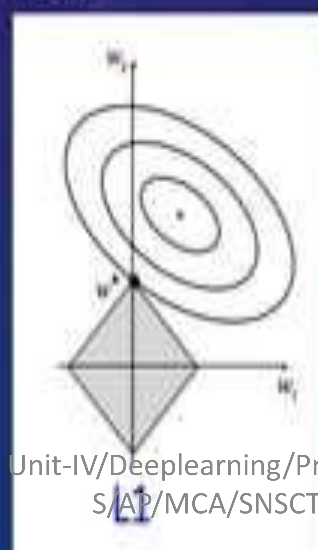




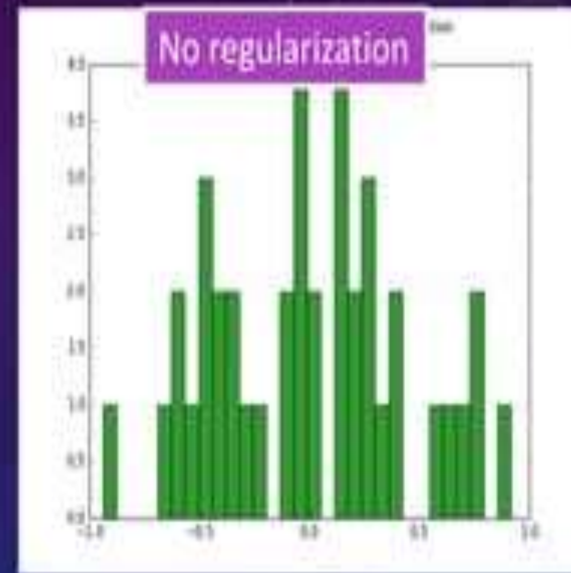
L1 REGULARIZATION

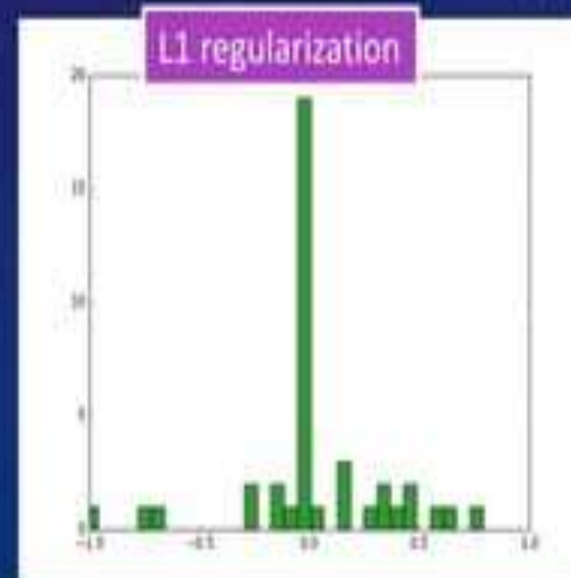
$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i\left(f\left(x^{(i)};W\right), y^{(i)}\right) + \lambda |w_j|$$

- penalizes the absolute value of the weight (v- shape function)
- tends to drive some weights to exactly zero (introducing sparsity in the model), while allowing some weights to be big



Weights distr

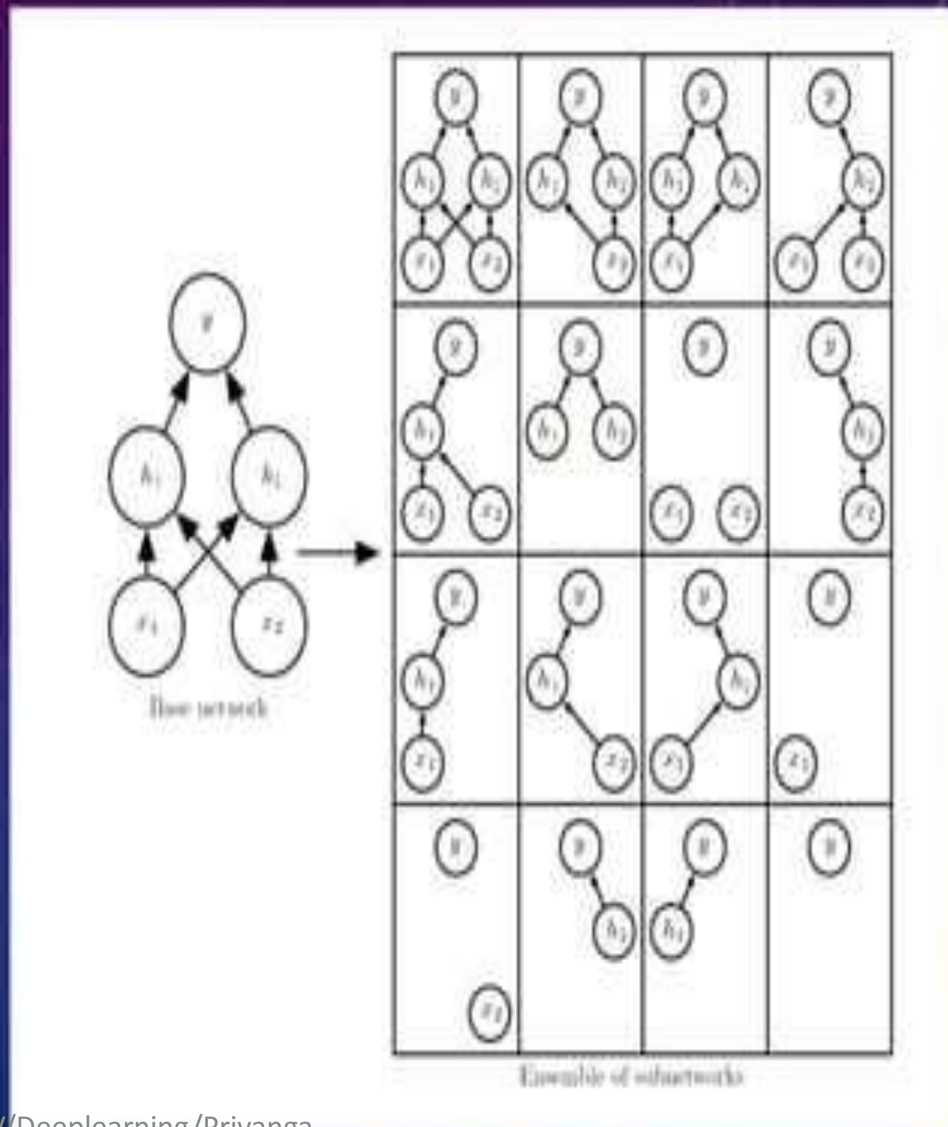




DROPOUT

In each forward pass, randomly set some neurons to zero. Probability of dropping is a hyperparameter; 0.5 is common.

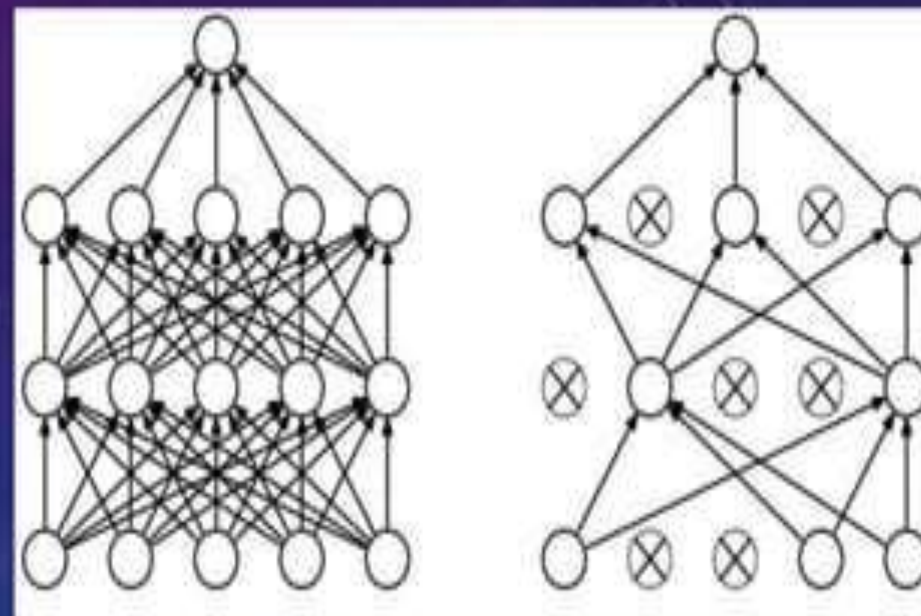
You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.



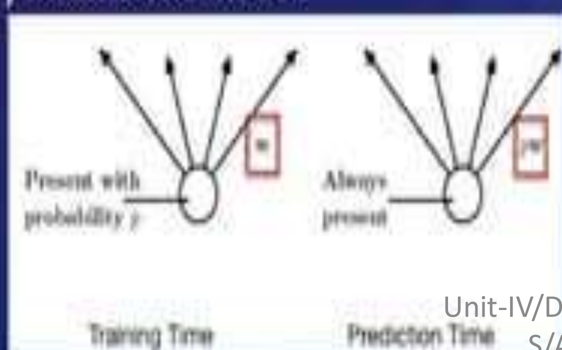
DROPOUT

Another interpretation:

- Dropout is training a large ensemble of models (that share parameters)
- Each binary mask is one model

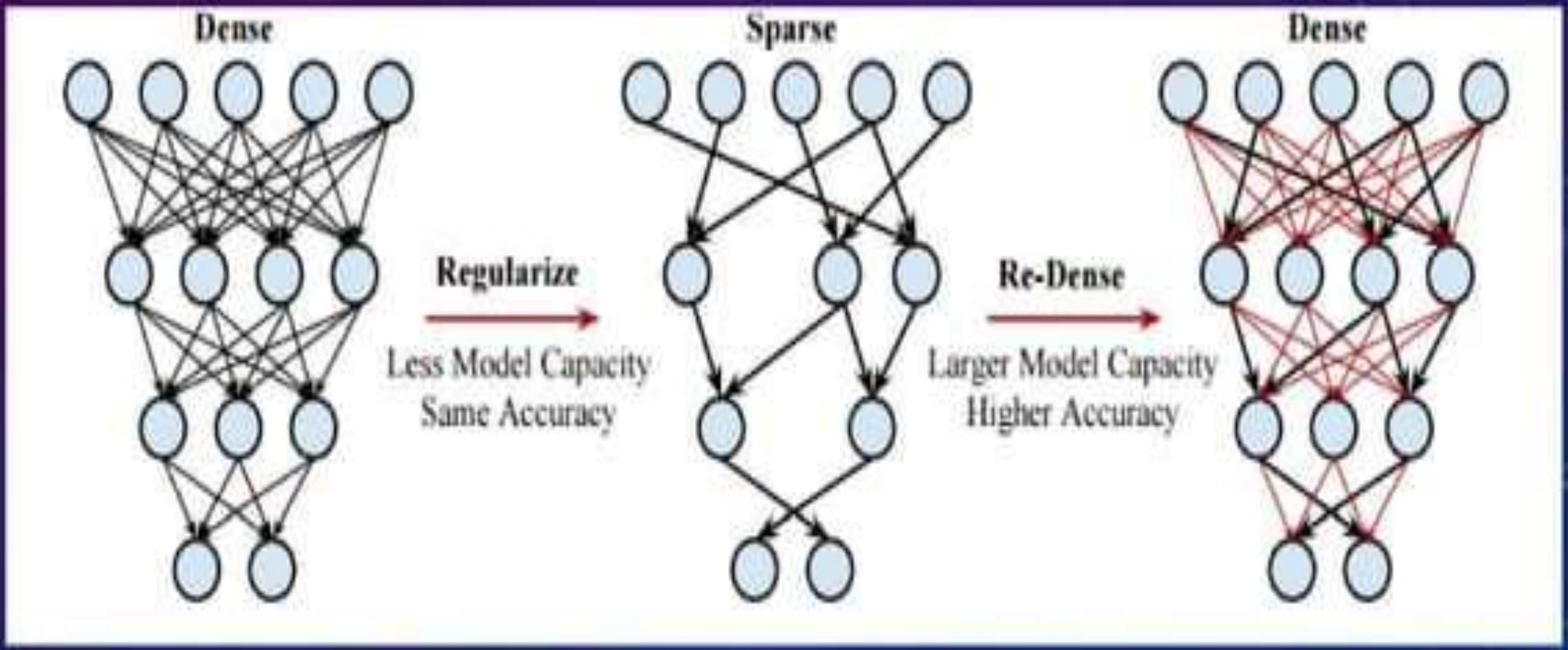


(a) Standard Neural Net (b) After applying dropout.



An fully connected layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

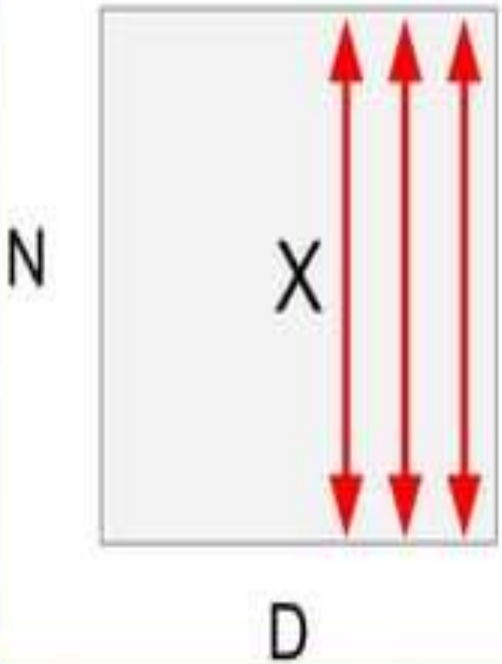
DENSE-SPARSE-DENSE TRAINING



https://arxiv.org/pdf/1607.04381v1.pdf

BATCH NORMALIZATION

"you want unit Gaussian activations? Just make them so."



1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

BATCH NORMALIZATION

Usually inserted after fully connected or convolutional layers, and before nonlinearity.

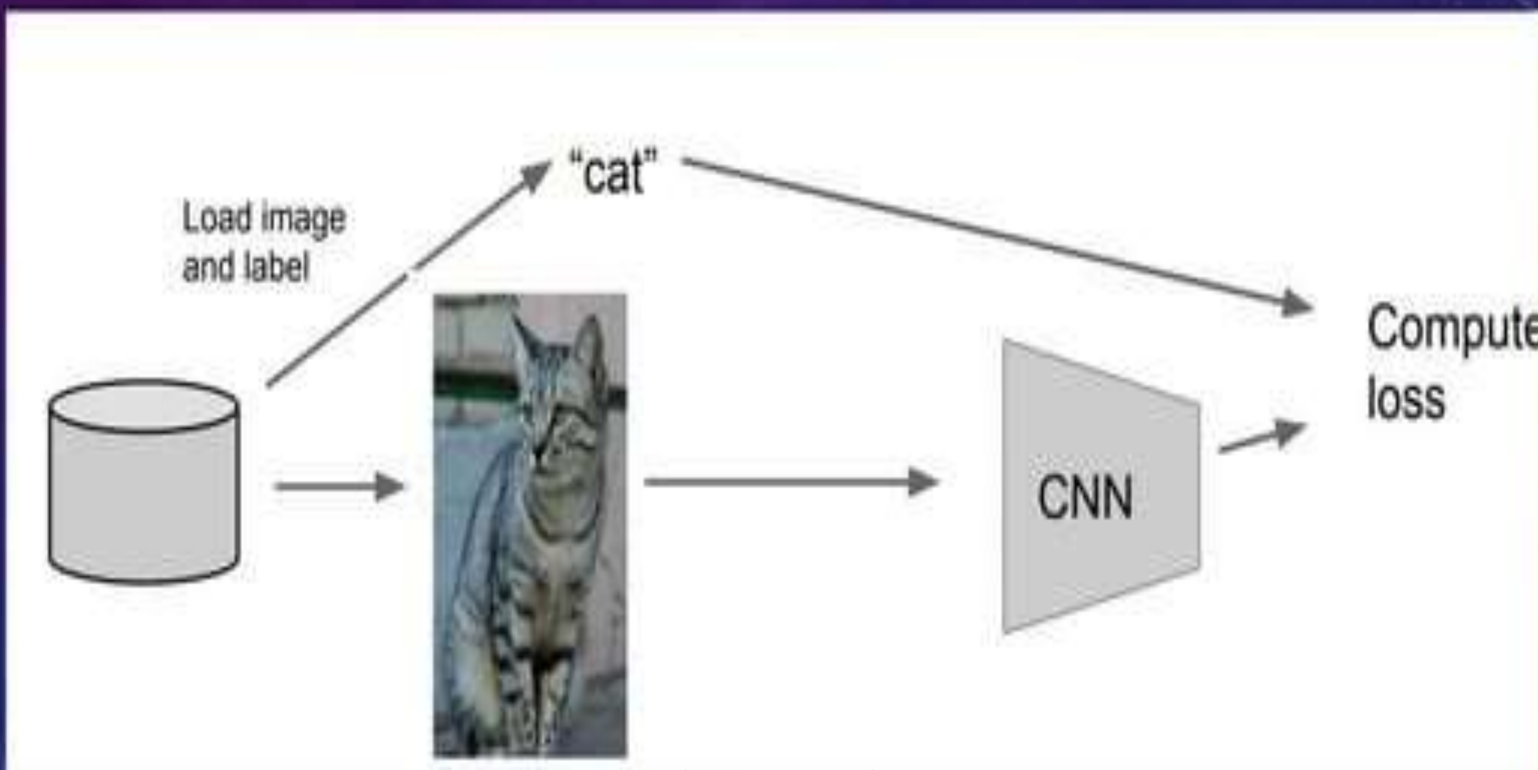
- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used. (e.g. can be estimated during training with running averages)

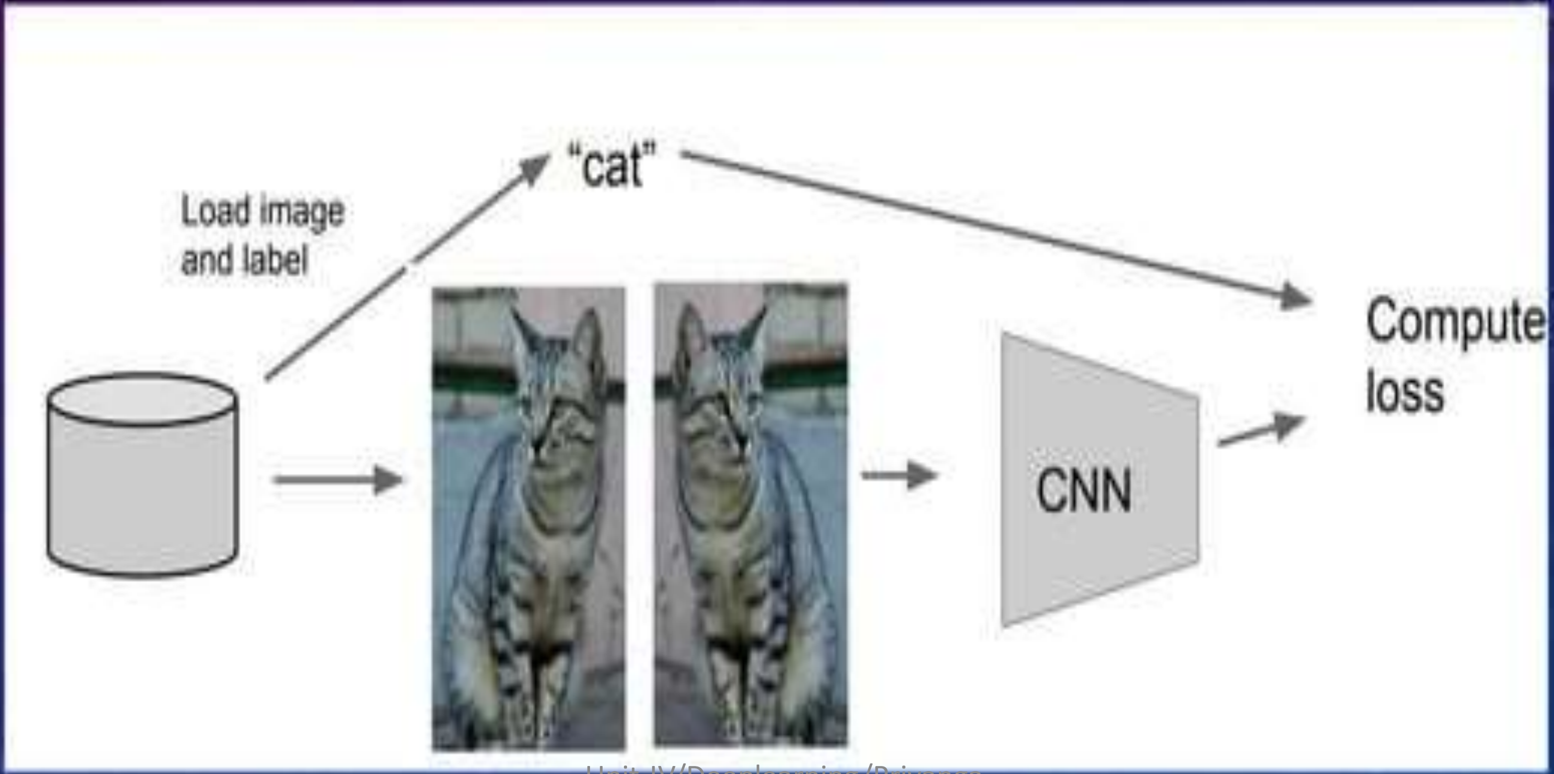
DATA AUGMENTATION

The best way to make a machine learning model generalize better is to train it on more data.



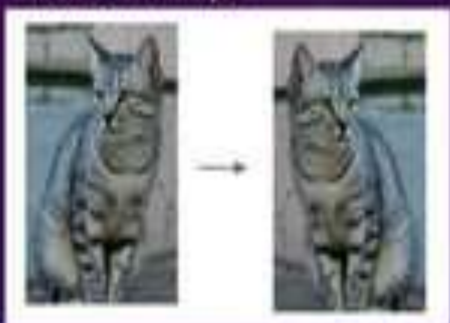
DATA AUGMENTATION

The best way to make a machine learning model generalize better is to train it on more data.



DATA AUGMENTATION

Horizontal flips



Random crops and scales

Color Jitter

- Simple: Randomize contrast and brightness

Get creative for your problem!

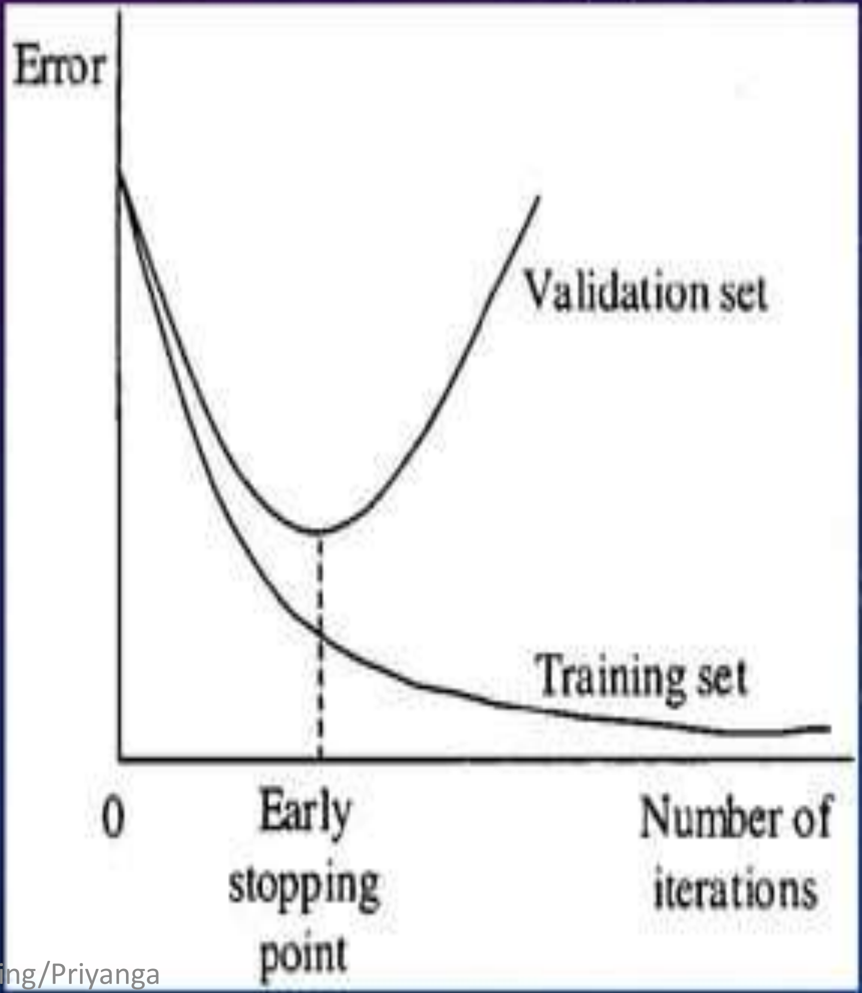
- Translation
- Rotation
- Stretching
- Shearing
- Lens distortions
- (go crazy)

EARLY STOPPING

It is probably the most commonly used form of regularization in deep learning to prevent overfitting:

- Effective
- Simple

Think of this as a hyperparameter selection algorithm. The number of training steps is another hyperparameter.



REFERENCE

- Deep Learning book ------ http://www.deeplearningbook.org/
- Stanford CNN course ----- http://cs231n.stanford.edu/index.html
- Regularization in deep learning ----- https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0

- So much more to learn, go explore!

- THANK YOU