Machine Learning 10-601

Tom M. Mitchell
Machine Learning Department
Carnegie Mellon University

April 15, 2015

Today:
- Artificial neural networks
- Backpropagation
- Recurrent networks
- Convolutional networks

Reading:
- Mitchell: Chapter 4
- Bishop: Chapter 5
- Quoc Le tutorial:
- Ruslan Salakhutdinov tutorial:

Artificial Neural Networks to learn f: X → Y

- f might be non-linear function
- X (vector of) continuous and/or discrete vars
- Y (vector of) continuous and/or discrete vars

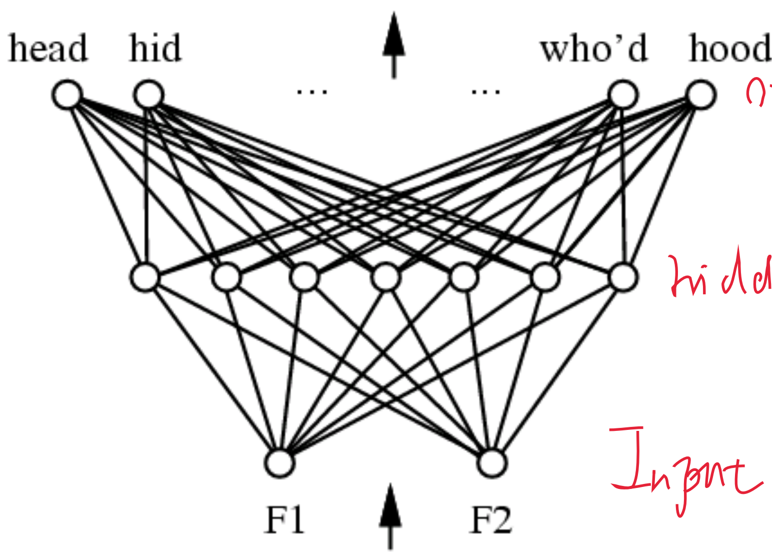
- Represent f by ***network*** of logistic units
- Each unit is a logistic function

$$unit\ output = \frac{1}{1 + exp(w_0 + \sum_i w_i x_i)}$$

$y = f(x) + \varepsilon, \ \varepsilon \sim N(0, \sigma^2)$

- MLE: train weights of all units to minimize sum of squared errors of predicted network outputs
- MAP: train to minimize sum of squared errors plus weight magnitudes of w.

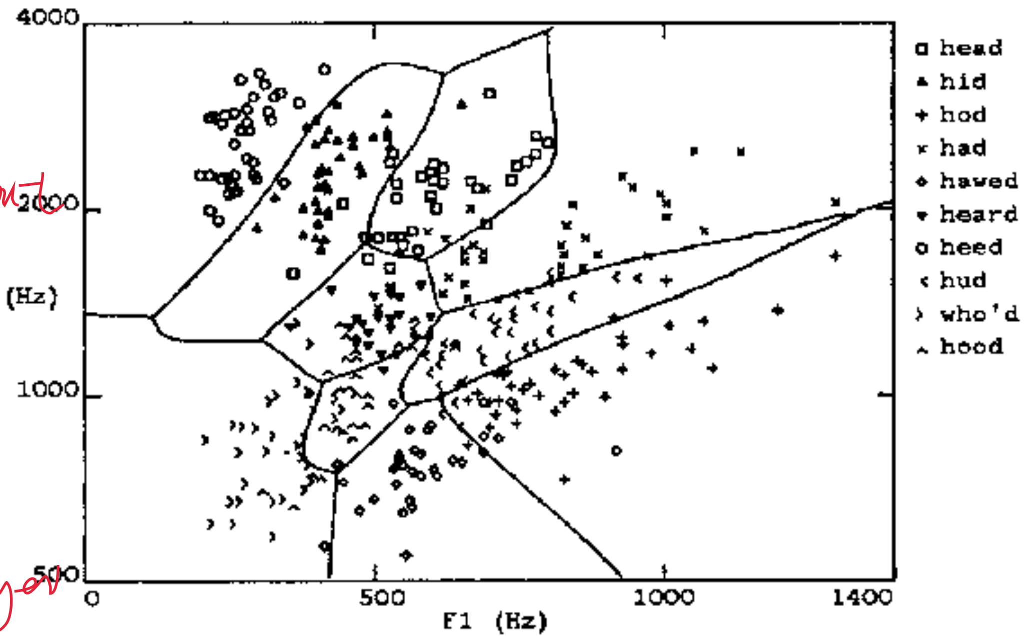
Multilayer Networks of Sigmoid Units



output

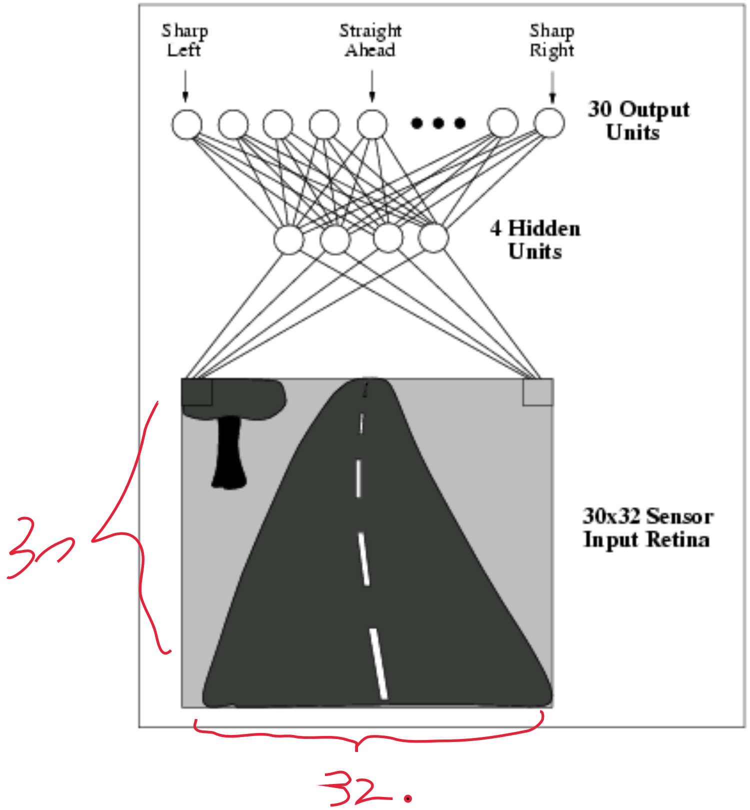
hidden

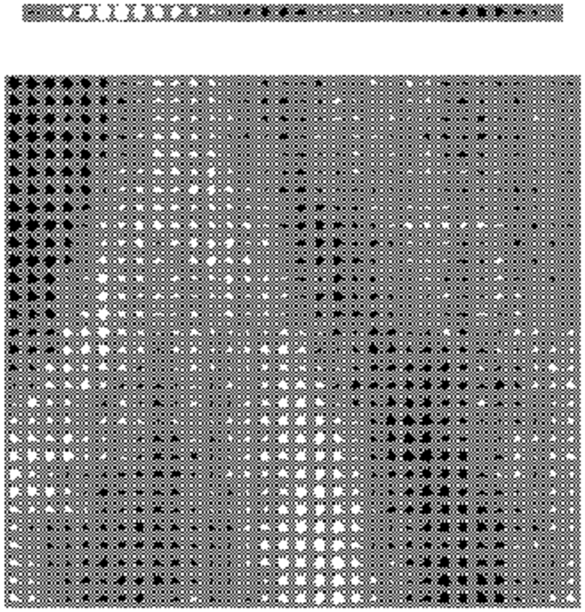
Input layer



ALVINN

[Pomerleau 1993]





Connectionist Models

Consider humans:

- Neuron switching time ˜ .001 second
- Number of neurons ˜ 10^{10}
- Connections per neuron ˜ 10^{4-5}
- Scene recognition time ˜ .1 second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

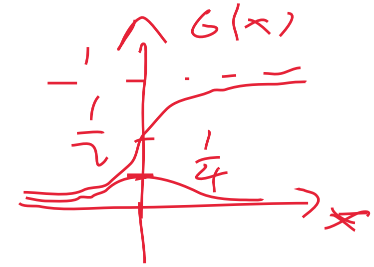
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

ReLU: Rectified Linear Unit.

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

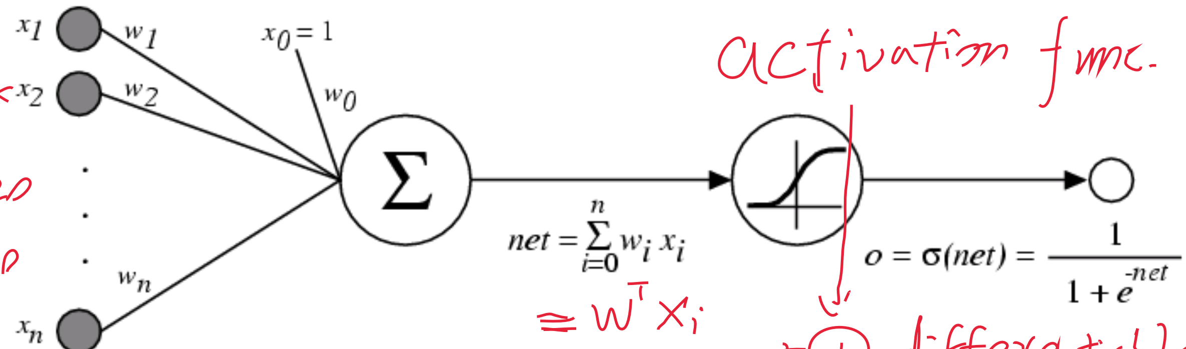
$f'(x) = \begin{cases} 1, & x \ge 0 \\ 0, & x < 0 \end{cases}$

Sigmoid Unit



$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

(Gradient vanishing)



$= w^T x_i$

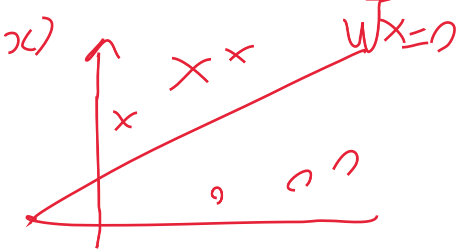
activation func.

1. differentiable (simple)
2. non-linear.

$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

$\log \frac{P(y=1 \mid X=x)}{P(y=0 \mid X=x)} = w^T x = 0$



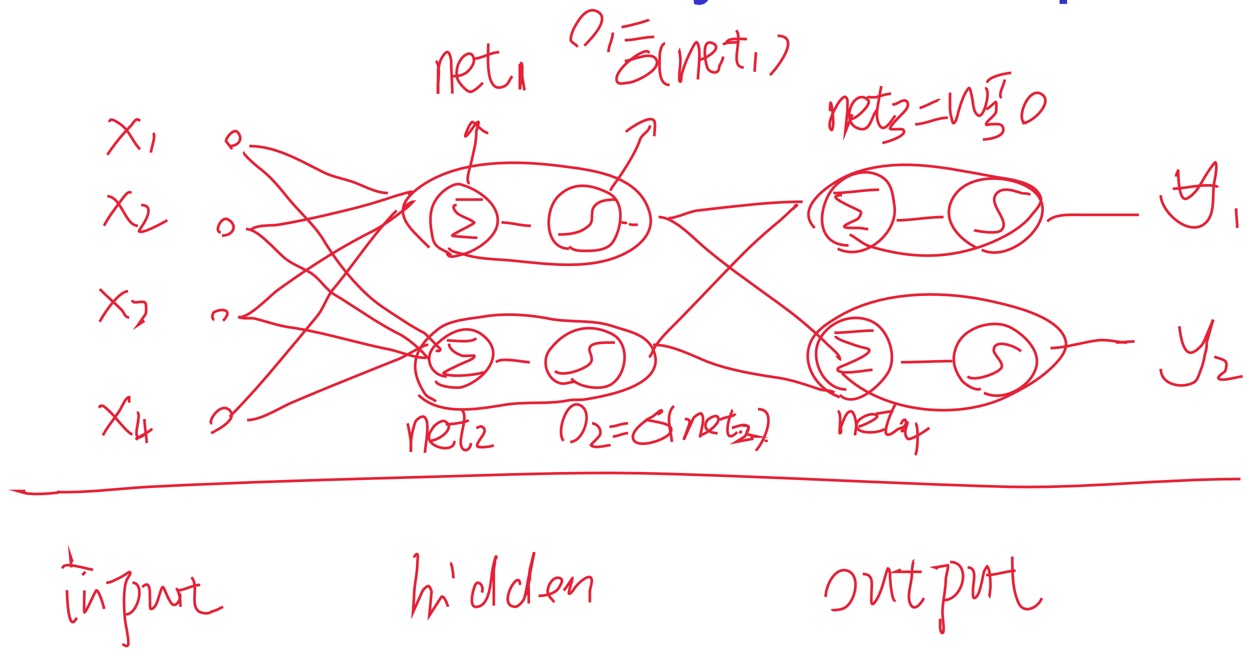
Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron (MLP)



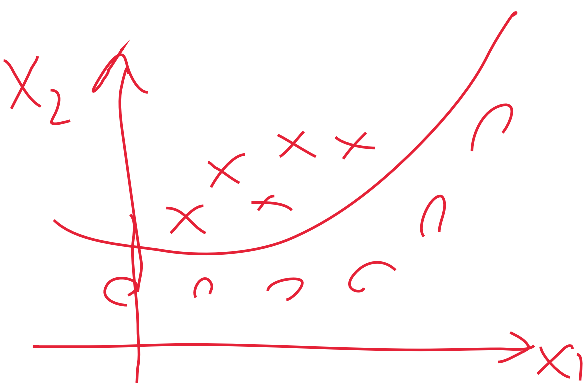
$y_1 = \sigma(net_3) = \sigma(w_3^T o)$

$y_2 = \sigma(net_4) = \sigma(w_4^T o)$

$$\boxed{\sigma(net) = \frac{1}{1+e^{-net}}}$$

$$\log \frac{P(y=1|x=x)}{P(y=0|x=x)} = net_3 = w_3^T o = w_{31} \cdot o_1 + w_{32} o_2$$

$$= w_{31} \cdot \sigma(w_1^T x) + w_{32} \cdot \sigma(w_2^T x) = 0$$



- single-class: $p(y=1|x) = \dfrac{e^{w^T x}}{1+e^{w^T x}}$
- multi-class: $p(y=j|x) = \dfrac{e^{w_j^T x}}{\sum_{j=1}^{K} e^{w_j^T x}}$ (softmax)

M(C)LE Training for Neural Networks

- Consider regression problem f:X→Y , for scalar Y

$y = f(x) + \varepsilon$

assume noise $N(0,\sigma_\varepsilon)$, iid

deterministic

- Let's maximize the conditional data likelihood

$$W \leftarrow \arg\max_W \ \ln \prod_l P(Y^l|X^l, W)$$

$$W \leftarrow \arg\min_W \ \sum_l (y^l - \hat{f}(x^l))^2$$

Learned neural network

MAP Training for Neural Networks

- Consider regression problem f:X→Y , for scalar Y

$$y = f(x) + \varepsilon$$

noise $N(0,\sigma_\varepsilon)$

deterministic

Gaussian $P(W) = N(0,\sigma I)$

$$W \leftarrow \arg\max_W \ \ln \ P(W) \prod_l P(Y^l | X^l, W)$$

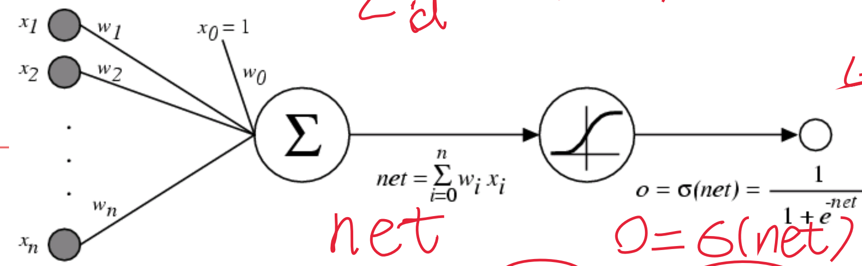
$$W \leftarrow \arg\min_W \left[c \sum_i w_i^2 \right] + \left[\sum_l (y^l - \hat{f}(x^l))^2 \right]$$

$\ln P(W) \leftrightarrow c \sum_i w_i^2$

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

Error Gradient for a Sigmoid Unit

$E=\frac{1}{2}\sum_d (t_d - o_d)^2$



net $\qquad O = \sigma(net)$

$$\frac{\partial E}{\partial w_i} = \sum_d \frac{\partial E_d}{\partial O_d}\cdot\frac{\partial O_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}$$

$-(t_d - O_d)$ $\qquad O_d(1-O_d)$ $\qquad x_{i,d}$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \in R^{(n+1)\times 1}$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\ \frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_d (t_d - o_d)\ \frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}
\end{aligned}$$

x_d = input

t_d = target output

o_d = observed unit output

w_i = weight i

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

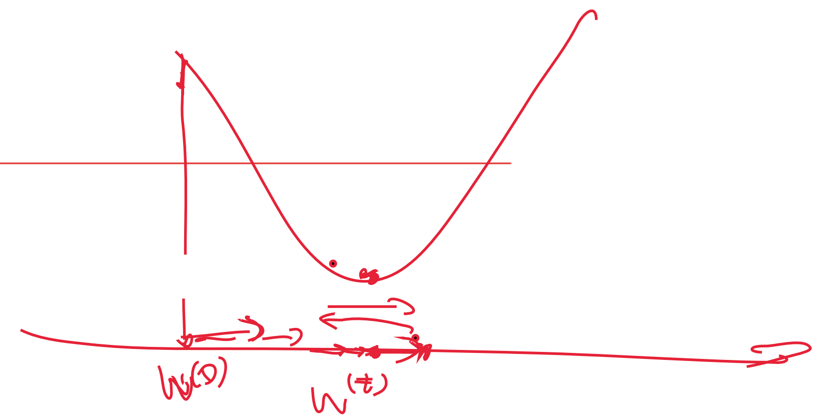
$$W_i \leftarrow W_i - \eta\frac{\partial E}{\partial W_i}$$

Gradient Descent (GD)

$$\frac{\partial E}{\partial W} = -(t-o)^T \mathrm{Diag}(o_d(1-o_d))$$

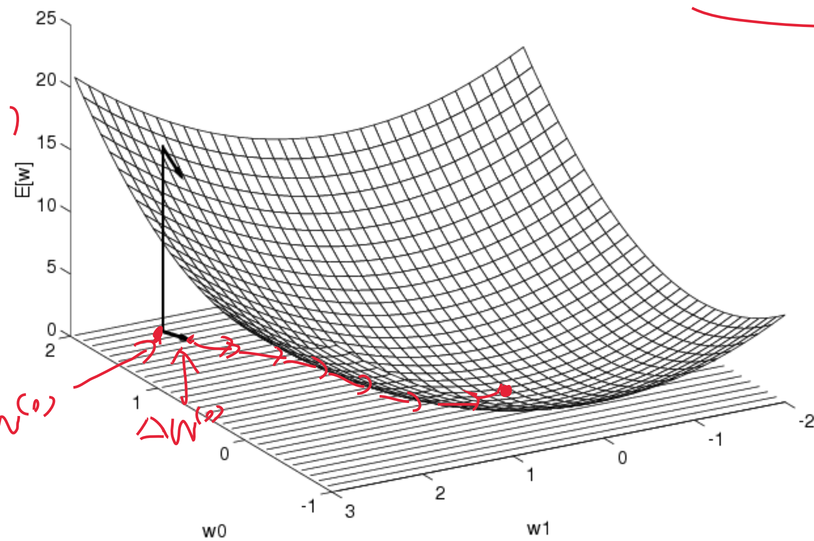
Gradient Descent

$$E=\frac{1}{2}\sum_d (t_d-o_d)^2$$



$$w=\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \in R^{(n+1)\times 1}$$

$$\frac{\partial\,\cdot}{\partial\,[\;]}=\begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{bmatrix}$$



Goal: $\min_w E[w]$

Solution: $w_i \leftarrow w_i + \Delta w_i$

$(i=0,1,\ldots,n)$

Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

learning rate

① fixed $\ll 10^{-3}$

② line search

i.e.,

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

Incremental (Stochastic) Gradient Descent

$E_D = \frac{1}{2} \sum_d (t_d - o_d)^2$

$E_d = \frac{1}{2} (t_d - o_d)^2$

Batch mode Gradient Descent: (GD)
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent: Stochastic GD (SGD)
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

(Adam)

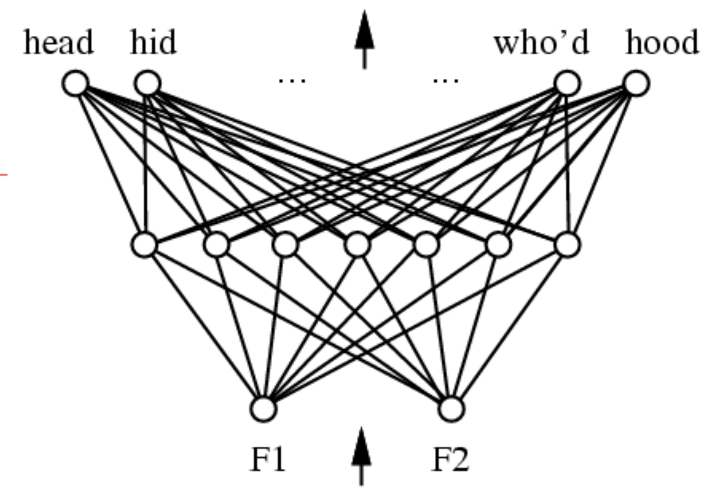
$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Backpropagation Algorithm (MLE)

(MLP)



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

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit k
  $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$
  3. For each hidden unit h
  $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$
  4. Update each network weight $w_{i,j}$
  $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
  where
  $$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{ij}}$$
  $$\Delta w_{i,j} = \eta \delta_j x_i$$

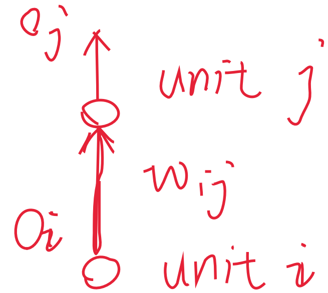
x_d = input

t_d = target output

o_d = observed unit output

w_{ij} = wt from i to j

Backpropagation



$net_j = x_{ij} w_{ij}$
$= o_i w_{ij}$

x_{ij}: input from unit i to unit j
($o_i = x_{ij}$)

① $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

$$= -\eta \frac{\partial E_d}{\partial w_{ij}}$$

② $E_d = \frac{1}{2} \sum_k (t_k - o_k)^2$

(k: #outputs)

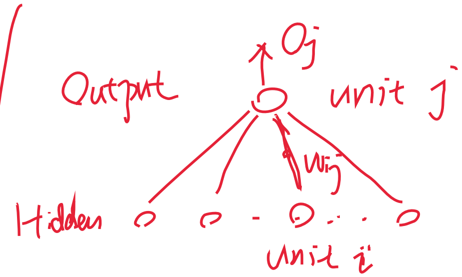
③ $\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}}$

$x_{ij} = o_i$

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

$$= \eta \delta_j x_{ij}, \quad \left(\delta_j = -\frac{\partial E_d}{\partial net_j}\right)$$

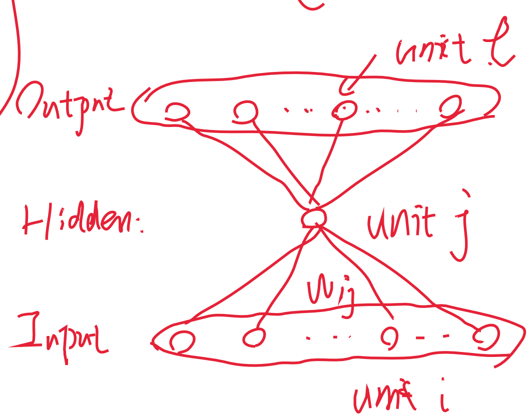
- Case 1: unit $j \in$ output layer.



$$\delta_j = -\frac{\partial E_d}{\partial net_j} = -\sum_{j=1}^{K} \frac{\partial E_d}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{j=1}^{K} (t_j - o_j)\, o_j (1 - o_j)$$

- Case 2. unit $j \in$ Hidden Layer



$$\delta_j = -\frac{\partial E_d}{\partial net_j}$$

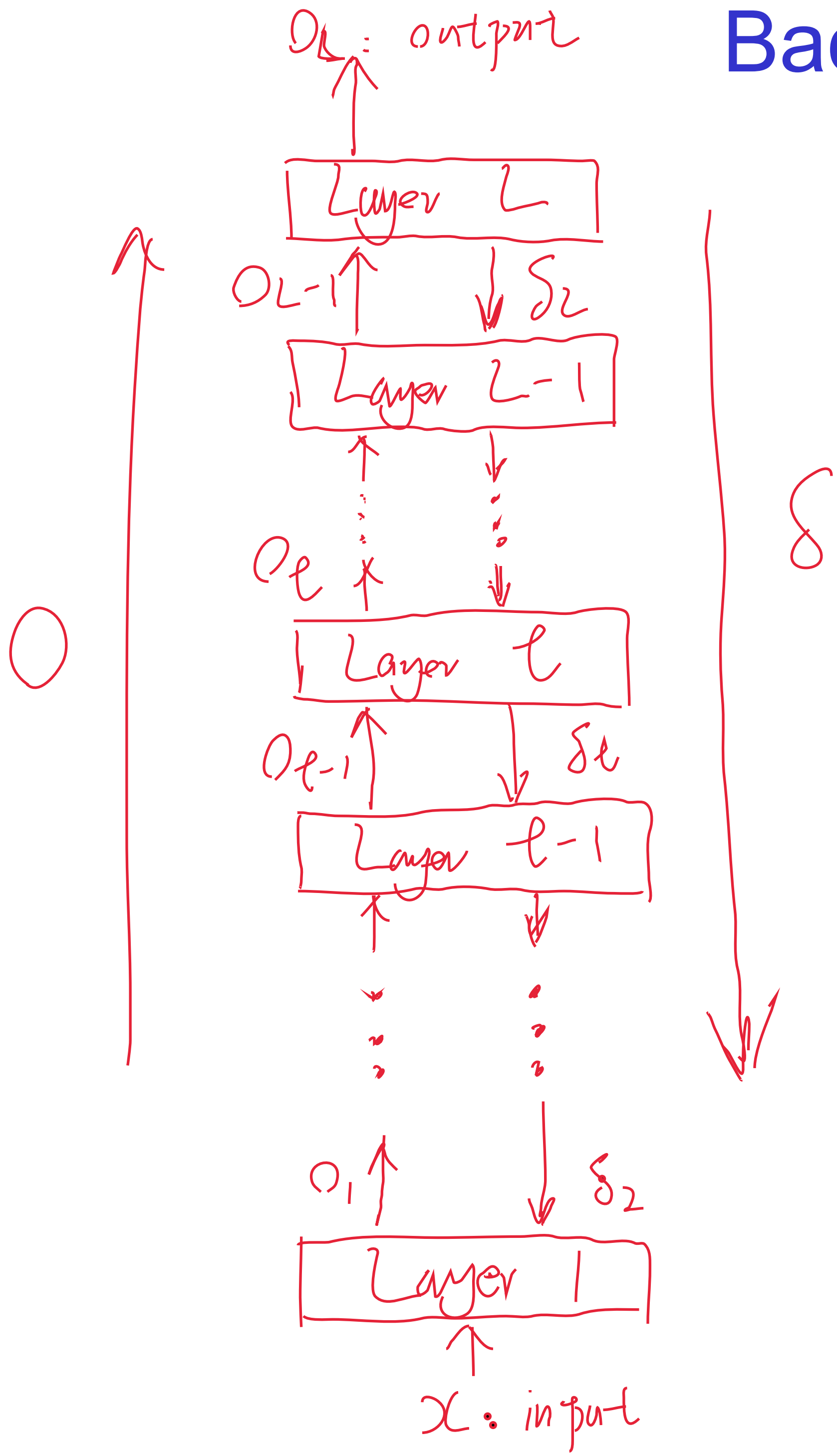
$$= -\sum_{l=1}^{K} \frac{\partial E_d}{\partial net_l} \cdot \frac{\partial net_l}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{l=1}^{K} \delta_l w_{jl}\, o_j (1 - o_j)$$

$$= o_j (1 - o_j) \sum_{l=1}^{K} \delta_l w_{jl}$$

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

Backpropagation



1. Initialization $w^{(0)}$

2. Repeat

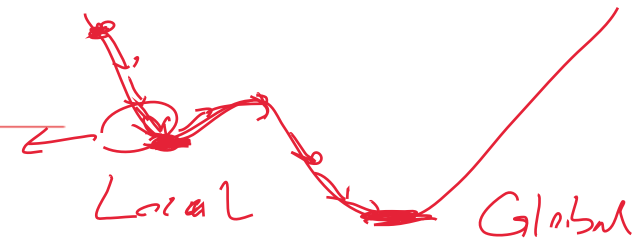
3. $$w_{ij} \leftarrow w_{ij} - \eta \cdot \delta_j \cdot O_i$$

   where $\delta_j = \begin{cases} (t_j - O_j) O_j (1 - O_j), & j \in \text{output} \\ O_j (1 - O_j) \sum_{\ell=1}^{K} \delta_\ell w_{j\ell}, & j \in \text{hidden} \end{cases}$

4. Until convergence

$$\frac{|Q^{(t+1)} - Q^{(t)}|}{|Q^{(t)}|} < \varepsilon = 10^{-5}$$

More on Backpropagation



- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α $\in (0,1)$

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

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

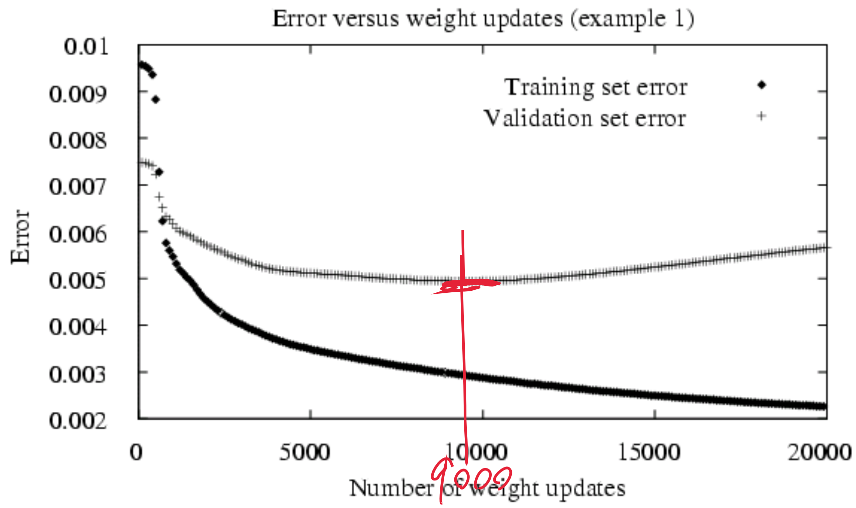
Local minimum.

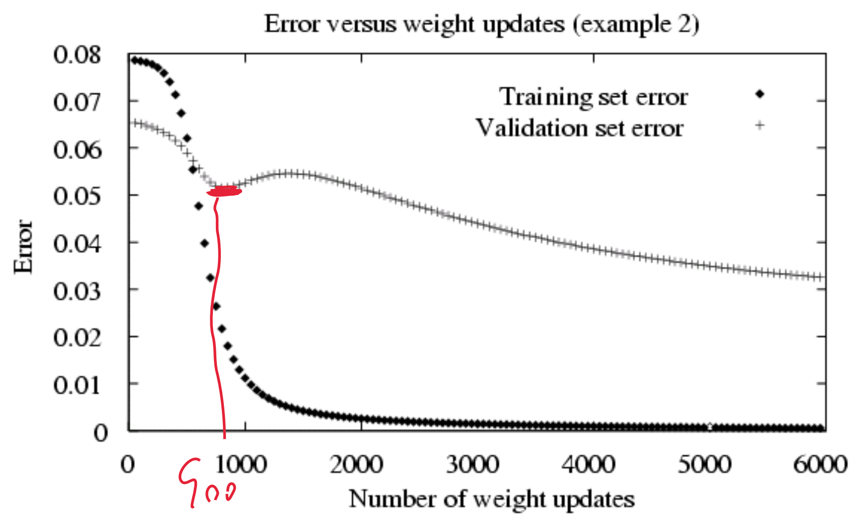
(1) momentum

(2) SGD

(3) different initializations

Overfitting in ANNs





overfitting

① weight decaying

② validation
(K-fold CV)

Expressive Capabilities of ANNs

$y \leftarrow f(x)$

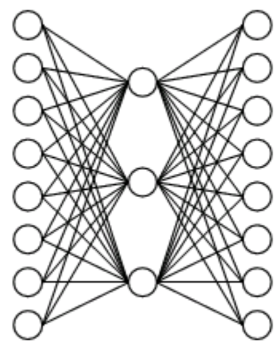
Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Learning Hidden Layer Representations



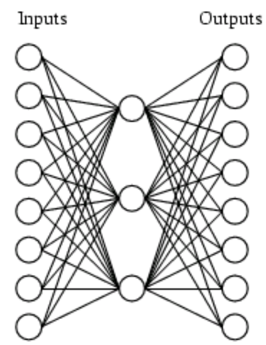
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learning Hidden Layer Representations

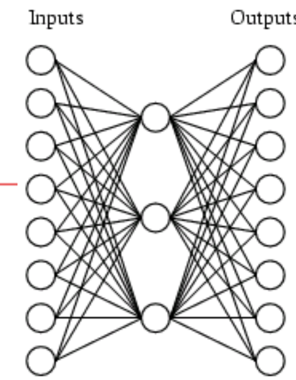
A network:



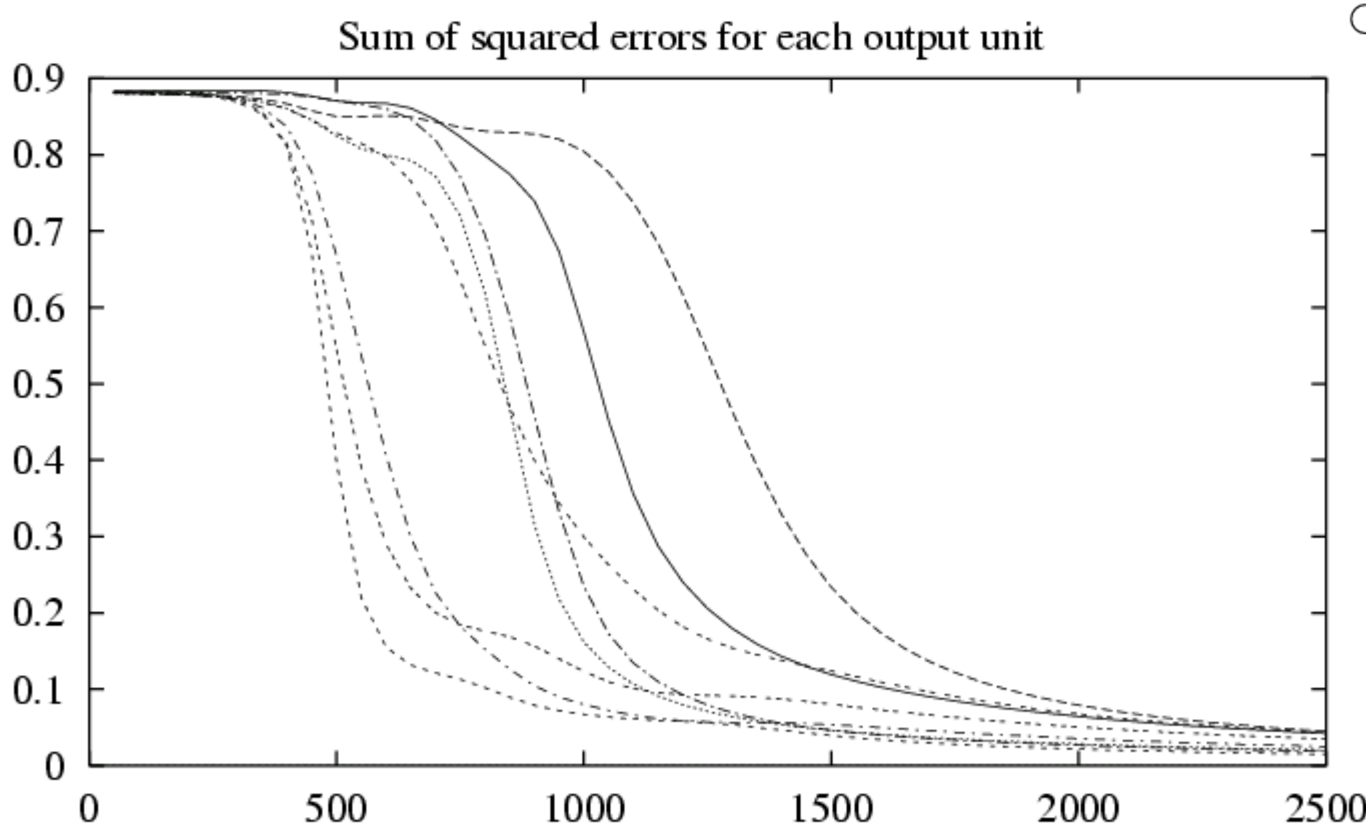
Learned hidden layer representation:

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

Training

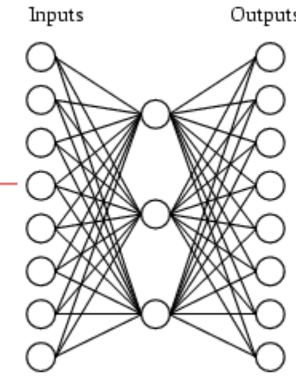


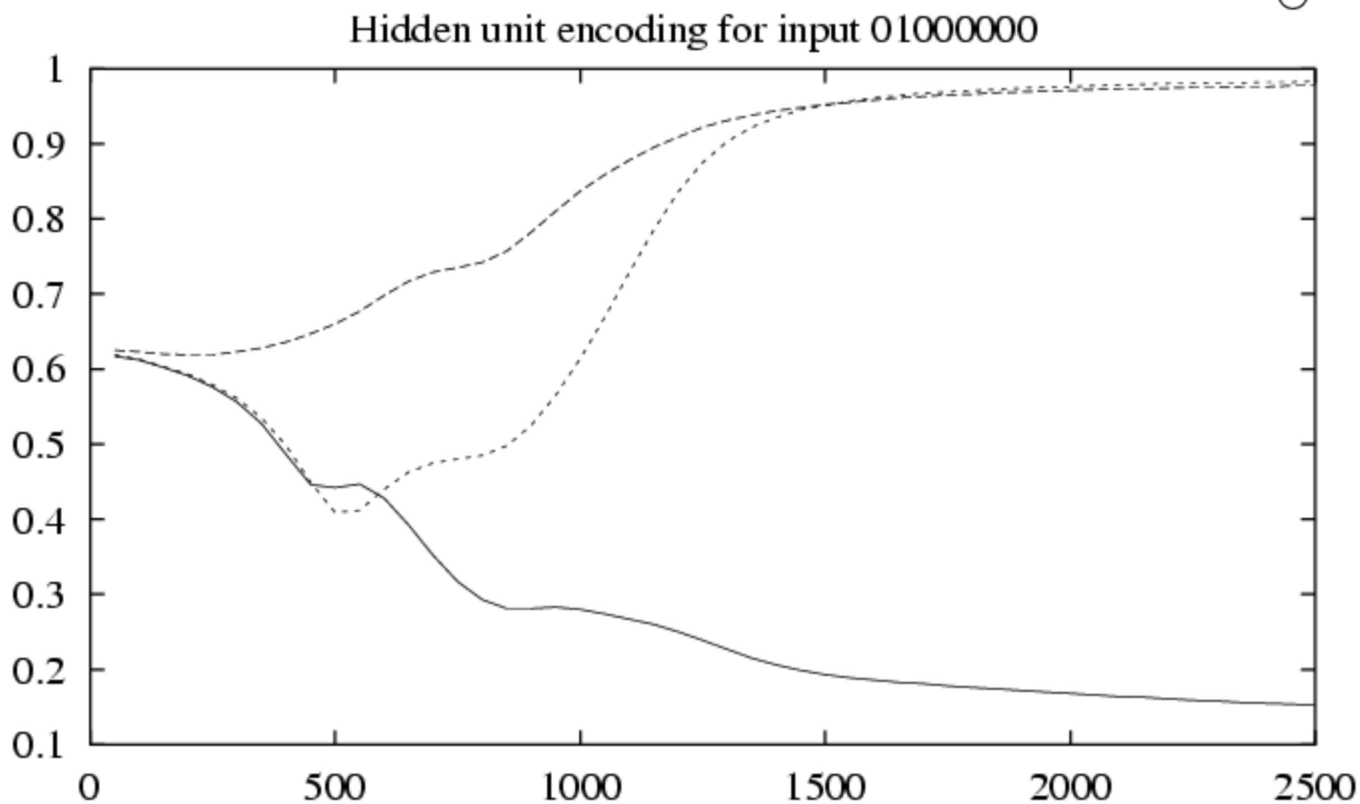
Inputs
Outputs



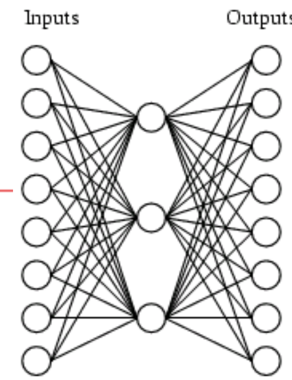
Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500

Training

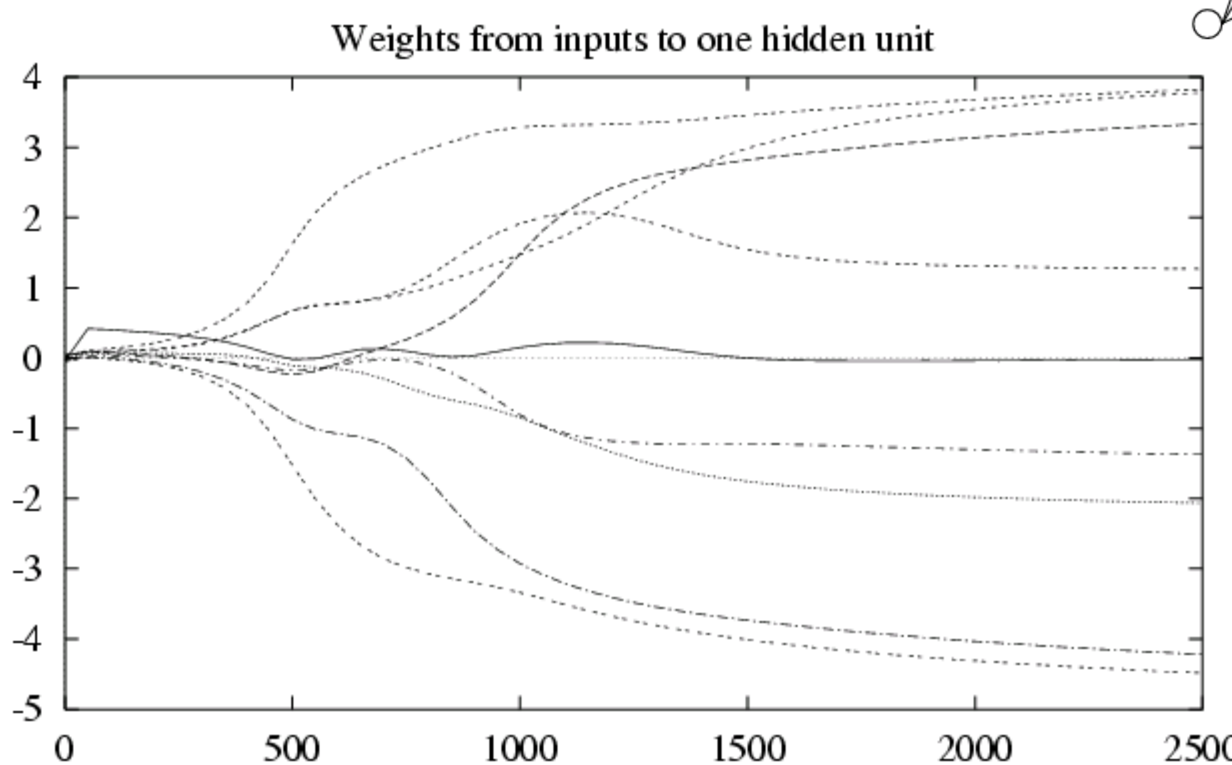

Inputs
Outputs


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Training



Inputs
Outputs



Weights from inputs to one hidden unit
4
3
2
1
0
-1
-2
-3
-4
-5
0
500
1000
1500
2000
2500

RNN: Recurrent Neural Networks

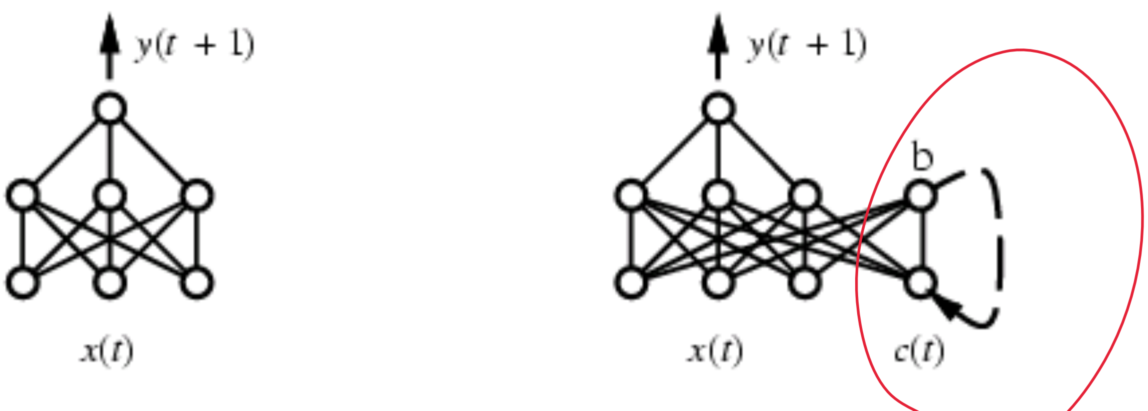
Training Networks on Time Series

- Suppose we want to predict next state of world
  - and it depends on history of unknown length
  - e.g., robot with forward-facing sensors trying to predict next sensor reading as it moves and turns

Recurrent Networks: Time Series

- Suppose we want to predict next state of world
  - and it depends on history of unknown length
  - e.g., robot with forward-facing sensors trying to predict next sensor reading as it moves and turns

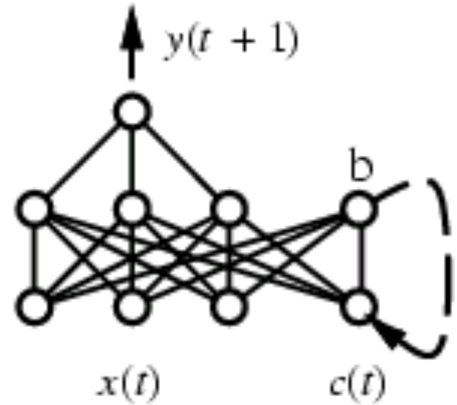
- Idea: use hidden layer in network to capture state history



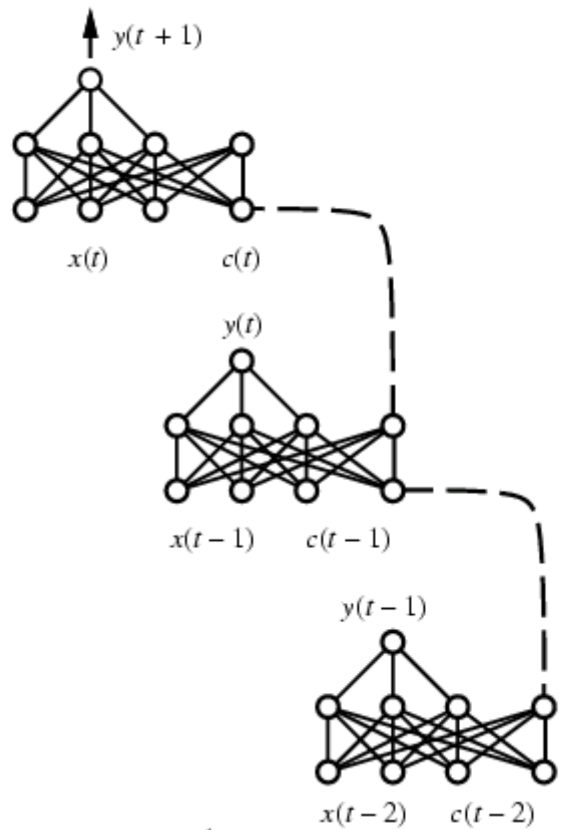
(a) Feedforward network (b) Recurrent network

Recurrent Networks on Time Series

How can we train recurrent net??



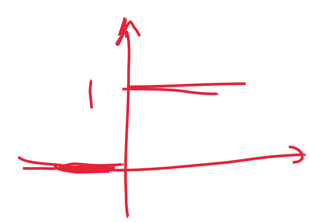
(parameter sharing)



(c) Recurrent network unfolded in time

$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

ReLU

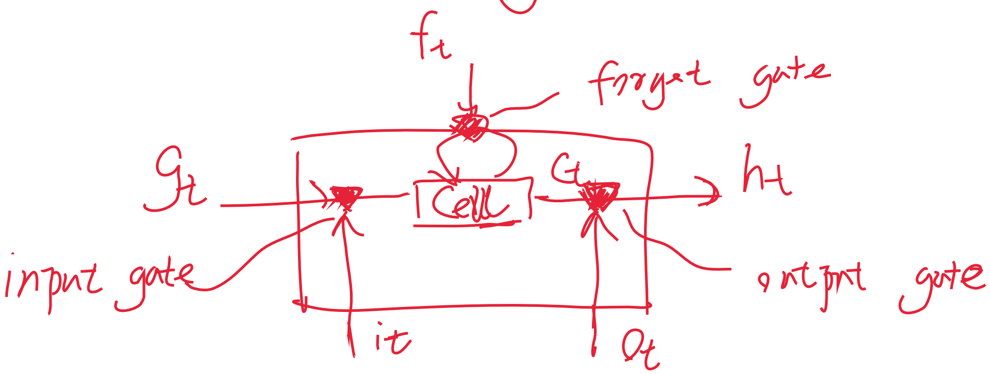


Recurrent Networks on Time Series

Gradient vanishing / exploding

$$\frac{\partial C_{t'}}{\partial C_t} = \prod_{k=1}^{t'-t} f_{t+k}$$

- LSTM (Long-Short Term Memory).



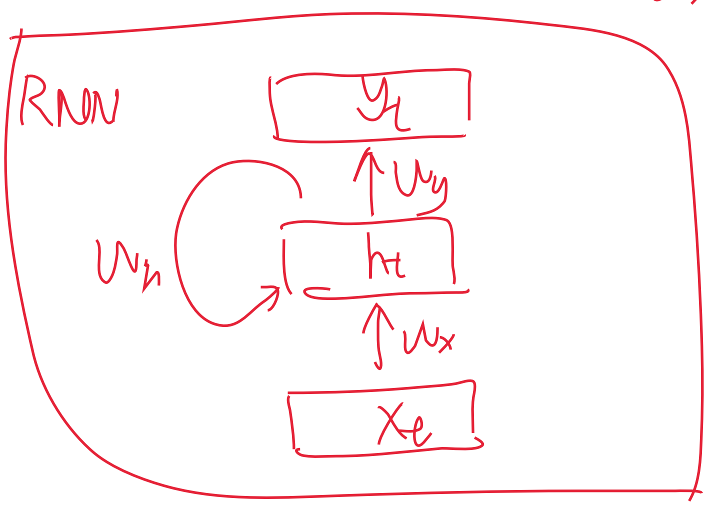
$$\begin{cases} C_t = f_t \odot C_{t-1} + i_t \odot g_t \\ h_t = o_t \odot \phi(C_t) \end{cases}$$

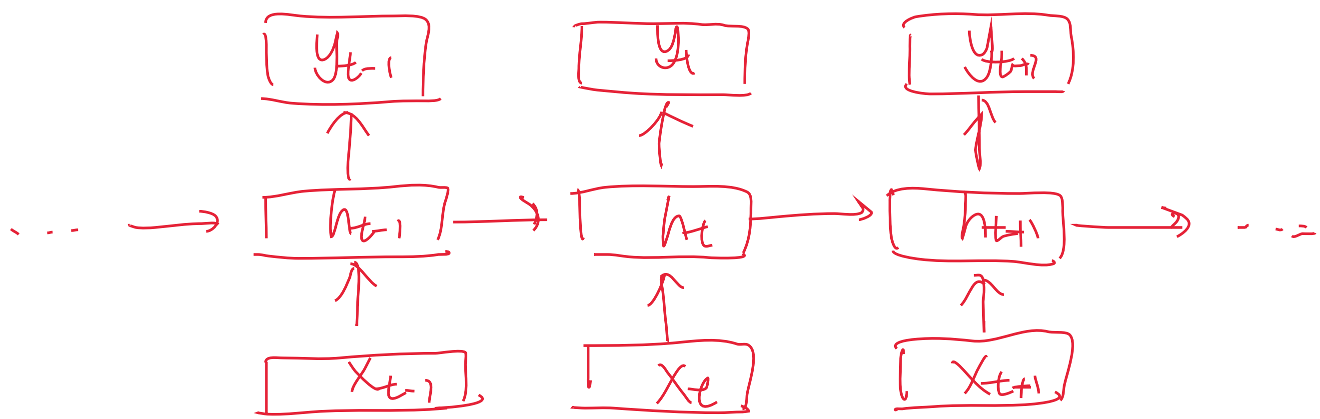
$\odot$: element-wise product.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \odot \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} ad \\ be \\ cf \end{bmatrix}$$

- GRU + Attention

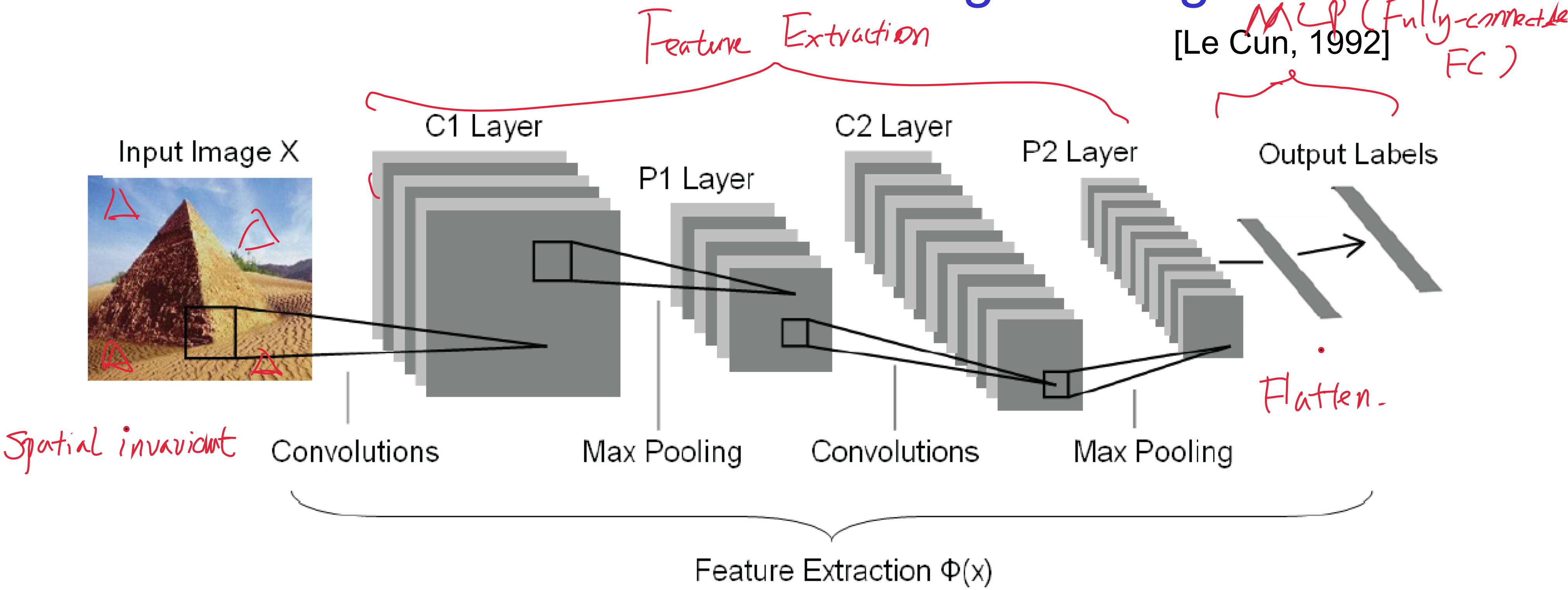
(Gate Recurrent Unit)





Convolutional Neural Nets for Image Recognition

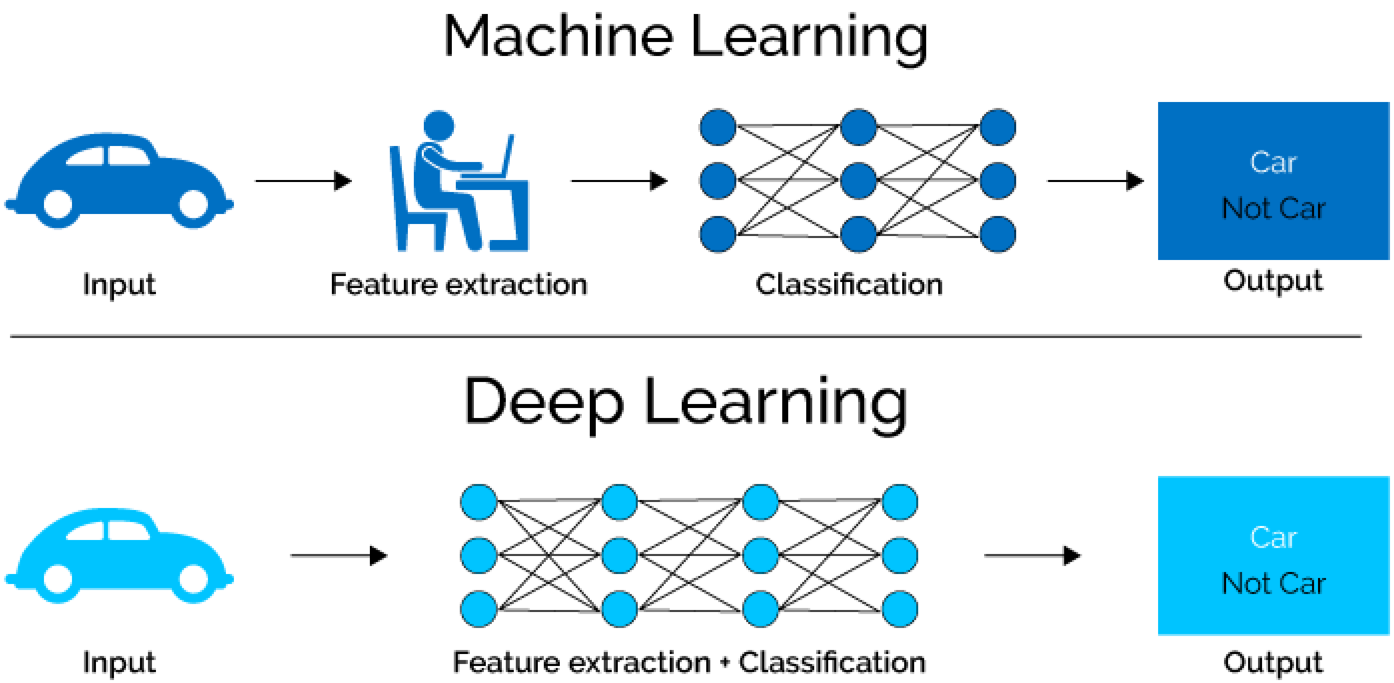
[Le Cun, 1992]



- 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, …

CNNs - Convolution Layer(s)

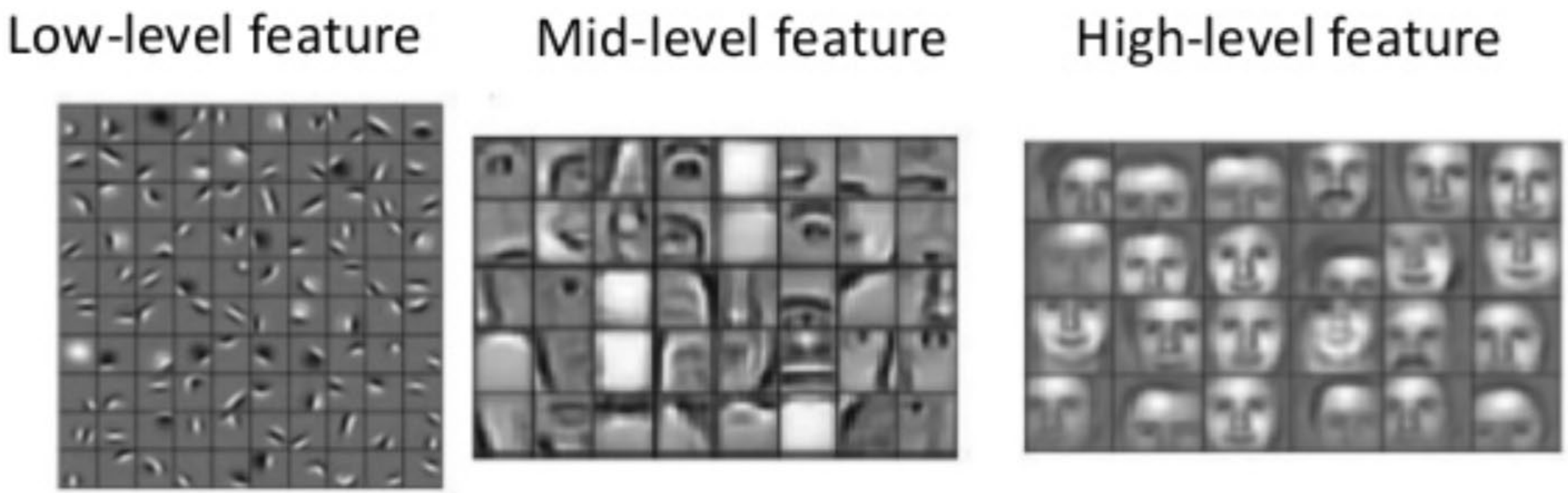
In CNNs and deep learning in general, the **features are learned** rather than manually selected during the data preprocessing phase.



CNNs - Convolution Layer(s)

There can be multiple convolution layers where earlier layers captures low-level features such as edges or colours while added layers capture high-level features such as facial parts.

Below are examples of **feature maps** created from a convolution layer.



Convolutional Neural Nets for Image Recognition

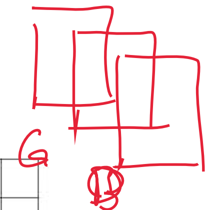
$$\underbrace{f(x,y)}_{\text{image}} * \underbrace{g(x,y)}_{\text{filer}} = \sum_{m,n} f(m,n) \cdot g(x-m, y-n)$$

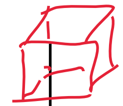
3x3 filter

Convolution

padding

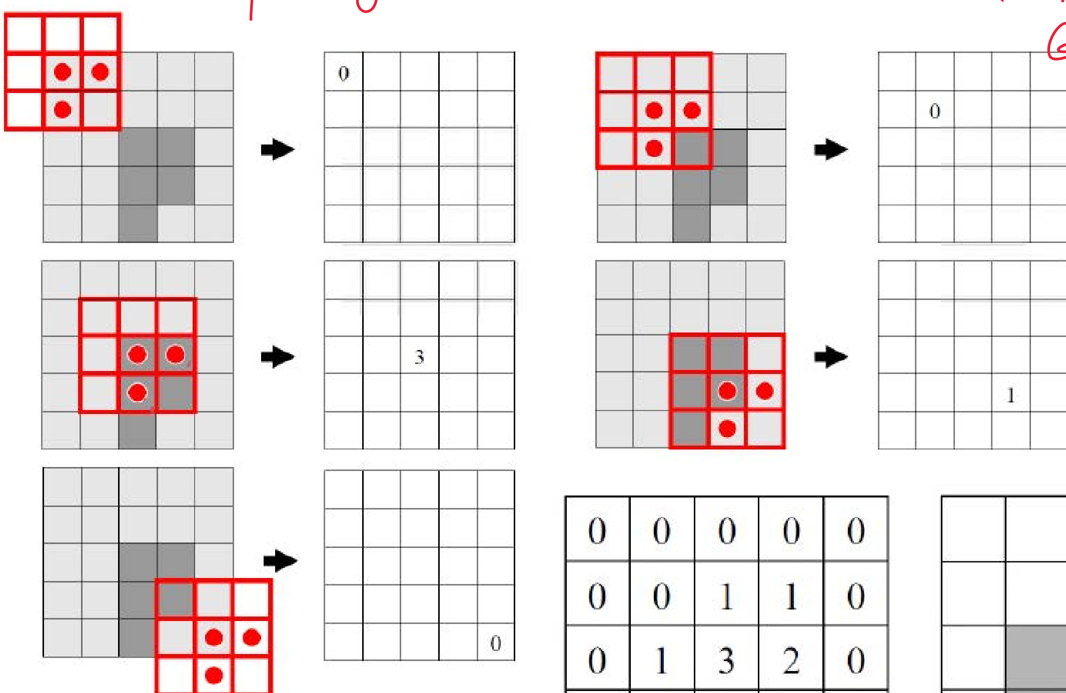
stride = 1



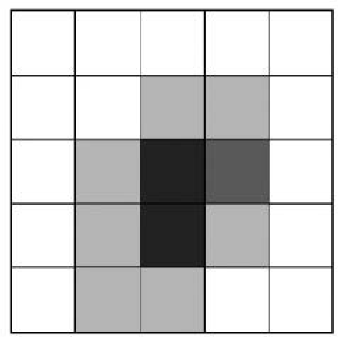


3x3x3

Max-Pooling



0	0	0	0	0
0	0	1	1	0
0	1	3	2	0
0	1	3	1	0
0	1	1	0	0



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

→

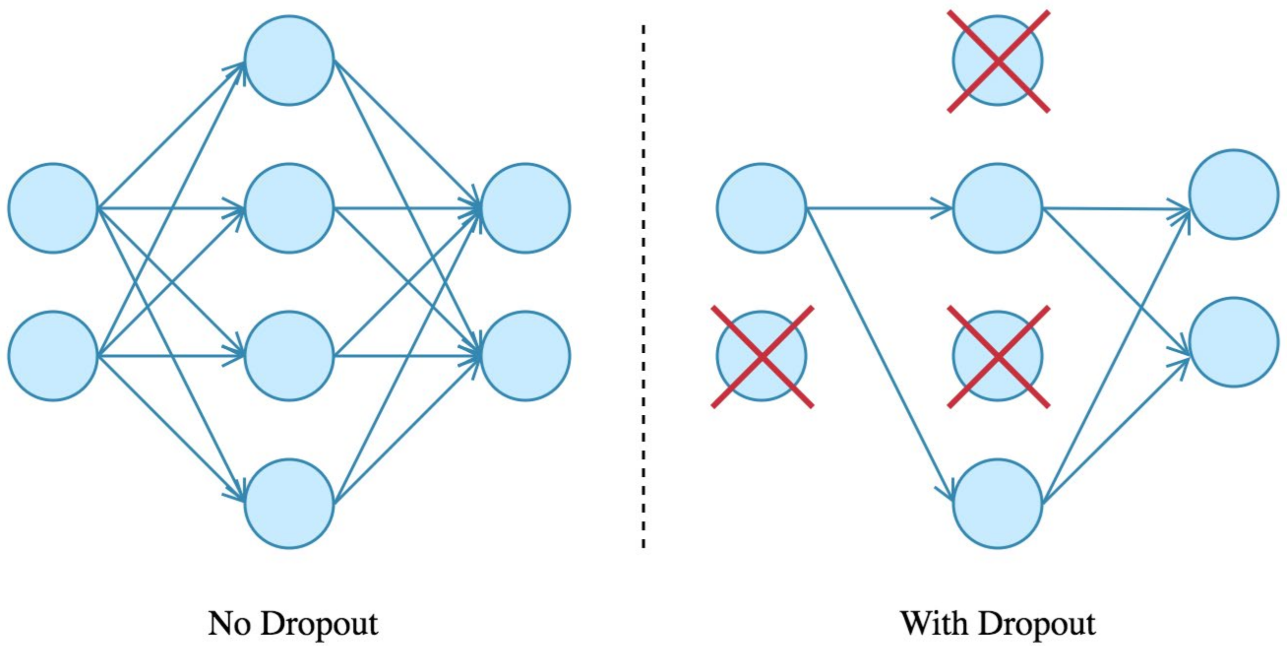
6	8
3	4

http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Dropout - Regularization Technique

This is basically the idea of dropout - **disabling neurons with probability p** so that the network isn't dependent on one node.

Dropout can be applied to input or hidden layers, but not output.



Artificial Neural Networks: Summary

- Highly non-linear regression/classification
- Hidden layers learn intermediate representations
- Potentially millions of parameters to estimate
- Stochastic gradient descent, local minima problems

- Deep networks have produced real progress in many fields
  - computer vision
  - speech recognition
  - mapping images to text
  - recommender systems
  - …
- They learn very useful non-linear representations