Volunteers for next!!!

Discussion about Tasks I and II?

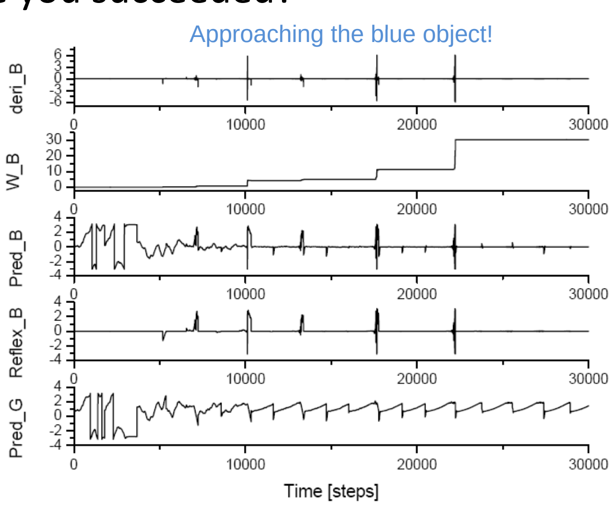
Goal-directed behavior & Dynamical control problem with ICO learning

Goal directed behavior learning

- Have you succeeded?

Goal directed behavior learning

- Have you succeeded?



Goal directed behavior learning

```
//2) goal 2 GREEN

//ICO
xt_reflex_angle_old2 = xt_reflex_angle2;

if(input_distance_s2 < range_reflex/*1.2 ~0.0120 very close to target*/)
{
  xt_reflex_angle2 = input_angle_s.at(1);
}
else
{
  xt_reflex_angle2 = 0.0;
}
reflexive_signal_green = xt_reflex_angle2;
//ICO
deri_xt_reflex_angle2 = xt_reflex_angle2-xt_reflex_angle_old2;

//3) goal 3 BULE
xt_reflex_angle_old3 = xt_reflex_angle3;

if(input_distance_s3 <range_reflex/*1.2 ~0.0120 very close to target*/)
{
  //xt_reflex_angle3 = xt_ico_lowpass3;//input_angle_s.at(2);
  xt_reflex_angle3 = input_angle_s.at(2);
}
else
{
  xt_reflex_angle3 = 0.0;
}
reflexive_signal_blue = xt_reflex_angle3;

deri_xt_reflex_angle3 = xt_reflex_angle3-xt_reflex_angle_old3;
```

Store previous reflex! (Green obj.)

Cal. reflex! (Green obj.)

Cal. Derivative reflex

Store previous reflex! (Blue obj.)

Cal. reflex! (Blue obj.)

Cal. Derivative reflex

Goal directed behavior learning

```
// ico learning --------------------------------------------------------------------//
double rate_ico = 0.1;

u_ico_in[0] = k_ico[0]* predictive_signal_green+reflexive_signal_green; // Green
u_ico_in[1] = k_ico[1]* predictive_signal_blue+reflexive_signal_blue;// Blue

k_ico[0] += rate_ico*deri_xt_reflex_angle2* predictive_signal_green ; //Green
k_ico[1] += rate_ico*deri_xt_reflex_angle3* predictive_signal_blue;  //Blue

u_ico_out = 1.0*u_ico_in[0]+1.0*u_ico_in[1]+exp_output;

//----Students--------Adding your ICO learning here--------------------------------//
```

Cal. The outputs of the learner neurons

Cal. the weights

Cal. The final output for steering!

Goal directed behavior learning

- Have you succeeded?
- Can we control the robot such that it learns to approach a desired object, e.g., Green?

Goal directed behavior learning

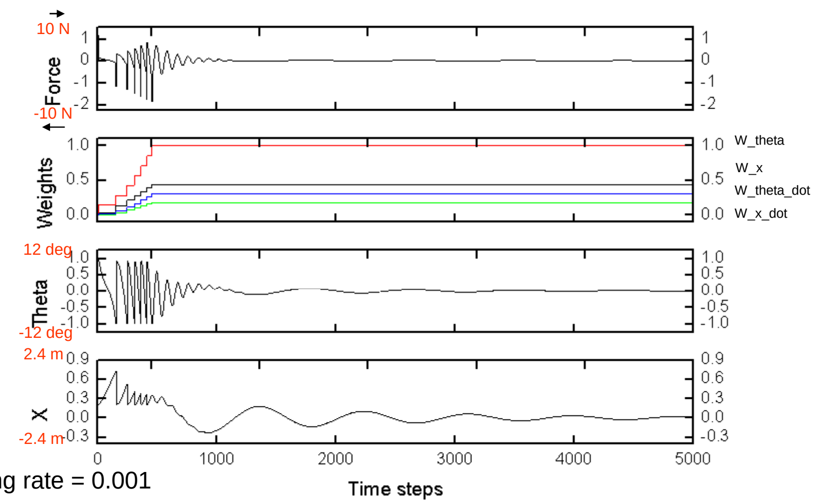
- Have you succeeded?
- Can we control the robot such that it learns to approach a desired object, e.g., Green?

- We will come to this point when we talk about RL!

Pole balancing

Experimental result: Initial condition at X = 0.5 m, Theta = 11 deg

ICO



Learning rate = 0.001

Pole balancing

```
//------------END-For students: Create your own controller and learning here------------//


// Remember the old state
Penalty_pre =  Penalty;
```

Store previous reflex!

```
//#### (1)
// Setting reflex
// Reflex action is trigger
// when theta > +-0.2007 rad (+-11.5 deg)
// or when x > +- 2.35 m

if(x_ico[TH] > /*0.196*/ 0.2007  || x_ico[_X] < -2.35)
```

Cal. reflex!

```
{
  Reflex = 1.0;    // for reflex action
  Penalty = -1.0; // for learning
}
else if(x_ico[TH] < /*-0.196*/ -0.2007  ||  x_ico[_X] > 2.35 )
{
  Reflex =  -1.0; // for reflex action
  Penalty = -1.0; // for learning
}
else
{
  Reflex = 0.0; // for reflex action
  Penalty = 0.0; // for learning
}
```

Pole balancing

```
//Find derivative of Reflex signal for learning/////////////////
deri_Penalty = Penalty-Penalty_pre;
```

Cal. Derivative reflex!

```
//Only positive derivative for update weights
deri_Penalty_actual = abs((deri_Penalty>0)? 0:deri_Penalty);
```

Cal. The output of the learner neuron

```
u_ico[0] = Reflex*1.0+kico[_X]*x_com+kico[_V]*x_dot_com+kico[TH]*theta_com+kico[OM]*theta_dot_com;
```

```
kico[_X] += learningRate_ico*deri_Penalty_actual*fabs(x_com);//(x_ico[_X]);
kico[_V] += learningRate_ico*deri_Penalty_actual*fabs(x_dot_com);//(x_ico[_V]);
kico[TH] += learningRate_ico*deri_Penalty_actual*fabs(theta_com);//(x_ico[TH]);
kico[OM] += learningRate_ico*deri_Penalty_actual*fabs(theta_dot_com);//(x_ico[OM]);
```

Cal. Weights

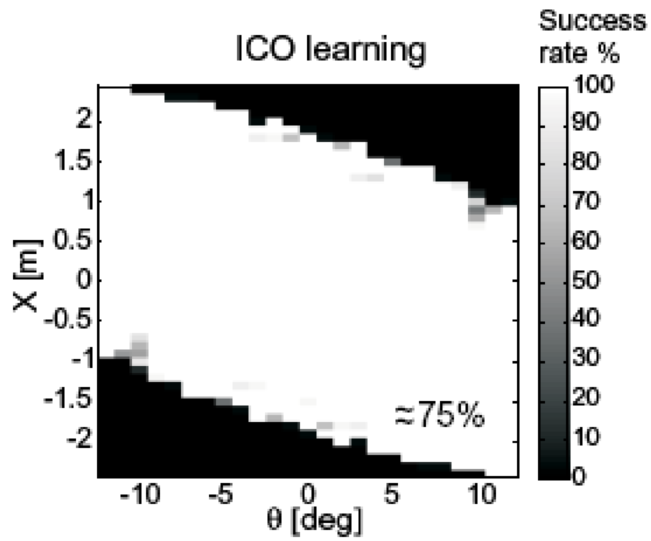
Pole balancing

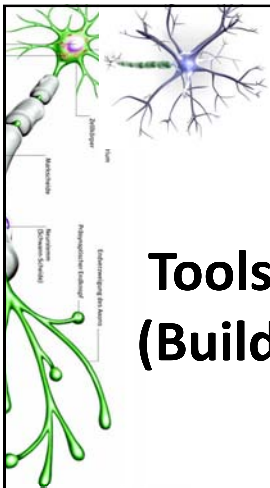
- Have you tried with other initial conditions?
- E.g., x = -1.6 m, theta = -10 deg?
- Did it work?

Pole balancing

- Have you tried with other initial conditions?
- E.g., x = -1.6 m, theta = -10 deg?
- Did it work? OF course NOT!!!!
- ICO cannot find proper combination → it requires an additional mechanism to explore other parameter space
- We will come to this point when we talk about RL

Pole balancing

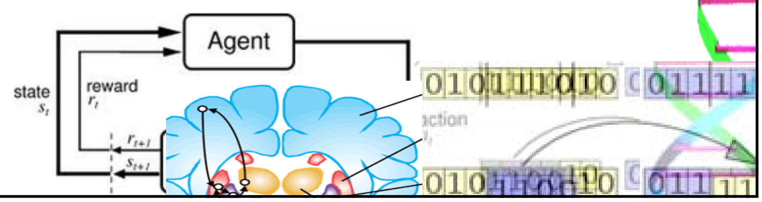




Tools of Artificial Intelligence (Building brains for machines)

RMAI2-U1 (ANNs_L2)

Poramate Manoonpong



About the course

- **Poramate Manoonpong** (KOH, poma@mmmi.sdu.dk)
- **My room:** Mærsk Mc-Kinney Møller Instituttet, ∅13-610b-1

- **5 ECTS credits**
- **Spring 2014** (3.5 hours / Block)→ One Block/ week, 12 Blocks
- **Lectures & exercises**

Lecture (**Theory**: up to 2.5 hours of each block):
From 7th Feb – 28th March 2014 (8 Blocks)
:→ 12:15-15:45 am. **(15-30 min presentation & discussion about tasks)**

Exercises (**Practice**: Robot simulation & Ludo game):
Leon Bodenhagen (room: D214A, lebo@mmmi.sdu.dk)
Another 1 hour of each block &
From 4th April - 9th May 2014 (4 Blocks)
:→ 12:15-15:45 am.

About the course

- **Evaluation:** Individual written report based on project and evaluated according to the Danish 7-point grading scale with external co-examiner
- Assessment: individual max 11 page report; implement one AI technique, compare to a second for Ludo Game (deadline by May 31st, 2014)

Guideline for the report & template:
http://manoonpong.com/AI2Lecture/
in the folder: /report

Slides: http://manoonpong.com/AI2Lecture/

User: student
Pass: ai2lecture

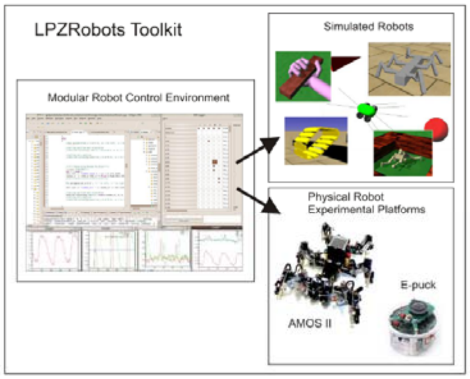
About the course

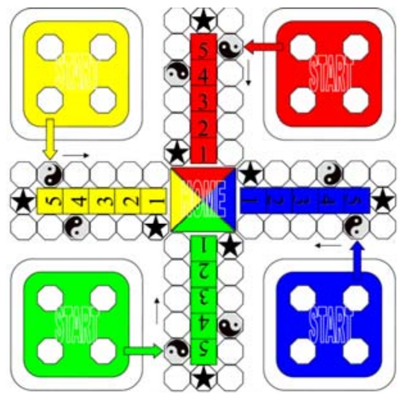
Recommended books:

1) Tom Mitchell: Machine Learning, McGraw-Hill, 1997, ISBN 0-07-042807-7
2) Neural Networks: A Systematic Introduction, Raúl Rojas
3) Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning), Richard S. Sutton and Andrew G. Barto
4) An Introduction to Genetic Algorithms, Melanie Mitchell

Exercises

1) Robot simulation (C++, gorobots_edu)
2) Ludo game (Java)





Contents

Artificial Intelligence (Week 1: 7 Feb)

Artificial neural networks (Weeks 2-4: 14, 21, ~~28 Feb~~ NO lecture, 7 March)

Reinforcement learning (Weeks 5-6: 14 March, 21 March)

Evolutionary computation (Weeks 7-8: 28 March, 4 April)

Weeks 9 onward just for practical work (LUDO)! (11, 18, 25 April, 2, 9 May)

Future & all remaining presentations Break Break

What did we learn last time?

Contents

Embodied AI

Artificial Intelligence
Artificial neural networks
Reinforcement learning
Evolutionary computation

"Intelligence requires a body (actuators/muscles, sensors, structure, materials)!"→ Interactions between body, brain, environment.

What did we learn last time?

Contents

Embodied AI

Artificial Intelligence
Artificial neural networks
Reinforcement learning
Evolutionary computation

What did we learn last time?

Contents

Embodied AI

Artificial Intelligence

Artificial neural networks

Reinforcement learning

Evolutionary computation

Single layer feedforward net.

What did we learn last time?

Contents

Embodied AI

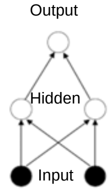
Artificial Intelligence

Artificial neural networks

Reinforcement learning

Evolutionary computation

Single layer feedforward net.



Multilayer feedforward net.

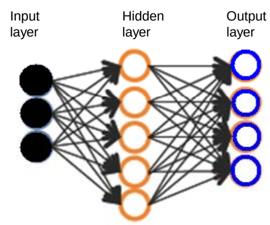
Today's Outline

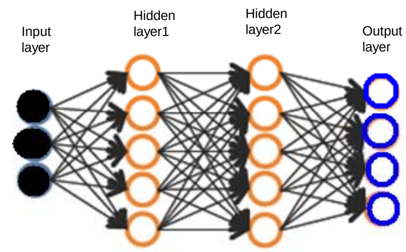
- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Today's Outline

- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

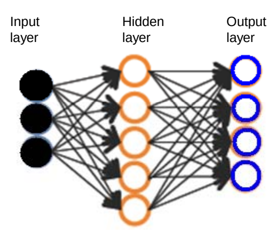
Multilayer Feedforward Neural Networks

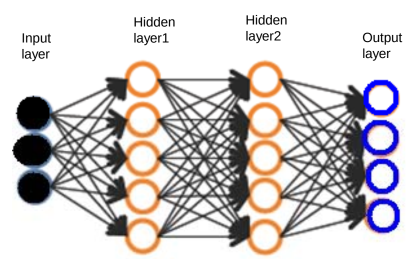




•Feedforward network topology (No connections with in a layer)

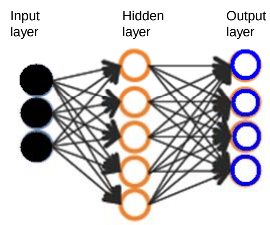
Multilayer Feedforward Neural Networks

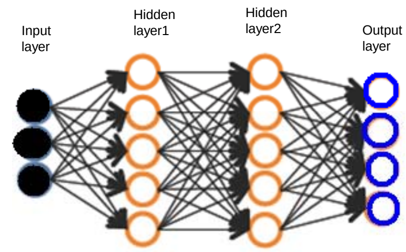




•Feedforward network topology (No connections with in a layer)
•Single or multiple linear input neurons

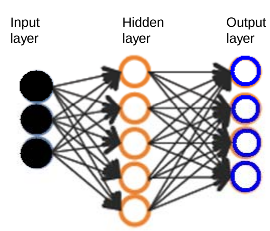
Multilayer Feedforward Neural Networks

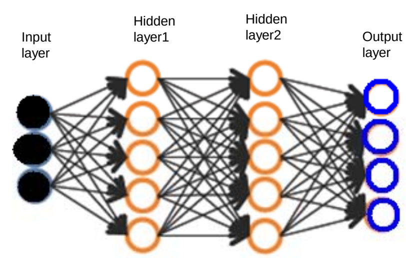




- Feedforward network topology (No connections with in a layer)
- Single or multiple linear input neurons
- Single or multiple hidden neurons

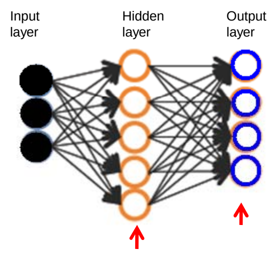
Multilayer Feedforward Neural Networks

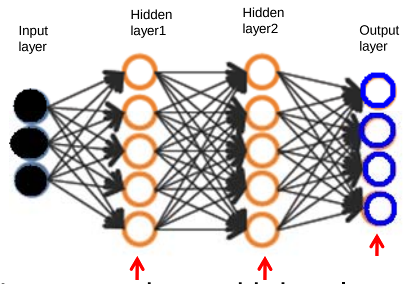




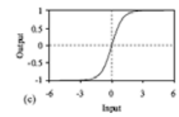
- Feedforward network topology (No connections with in a layer)
- Single or multiple linear input neurons
- Single or multiple hidden neurons
- Single or multiple output neurons

Multilayer Feedforward Neural Networks





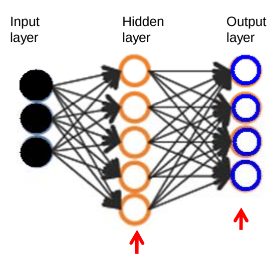
- •Feedforward network topology (No connections with in a layer)
- •Single or multiple linear input neurons
- •Single or multiple hidden neurons
- •Single or multiple output neurons
- •Perceptron-like units with smooth f(·) → Multilayer perceptrons

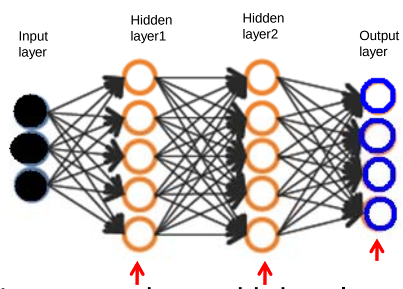


$$f(u) = \frac{1}{1+e^{-u}}$$

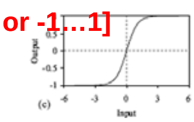
$$f(u) = \tanh(u) = \frac{2}{1+e^{-2u}} - 1$$

Multilayer Feedforward Neural Networks





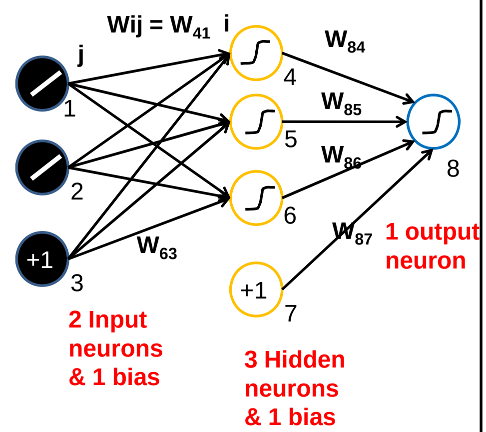
- •Feedforward network topology (No connections with in a layer)
- •Single or multiple linear input neurons **[0..1 or -1…1]**
- •Single or multiple hidden neurons
- •Single or multiple output neurons
- •Perceptron-like units with smooth f(·) → Multilayer perceptrons
- •Trainable using ‘Backpropagation’



$$f(u) = \frac{1}{1+e^{-u}}$$

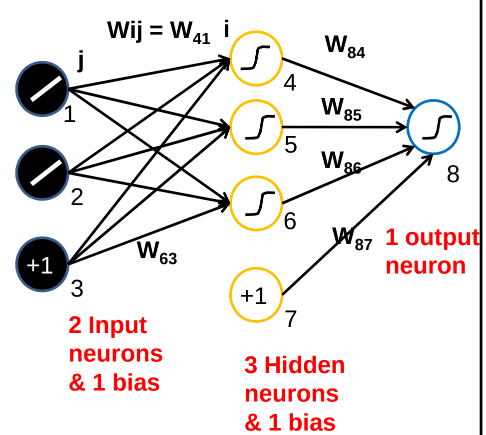
$$f(u) = \tanh(u) = \frac{2}{1+e^{-2u}} - 1$$

Multilayer Feedforward Neural Networks (Forward Operation or Propagation)



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

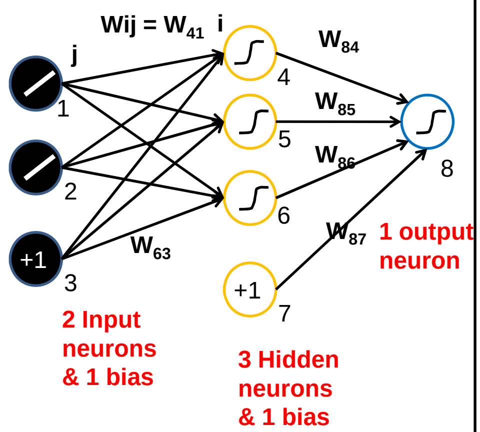
•*Wij* is weight **from** *j* **to** *i*



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

•*Wij* is weight **from** *j* **to** *i*

• Order units (topological sort)
❑ Label from input, hidden, to output unit(s): 1,…, n

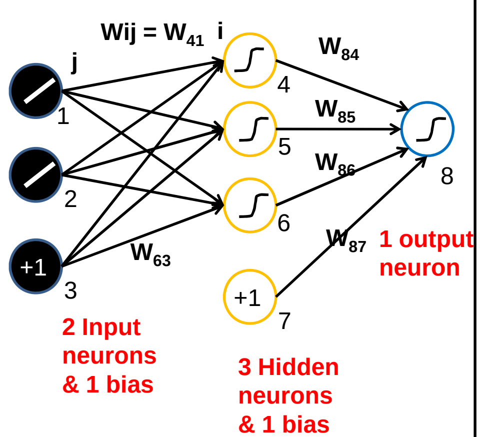


Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

•*Wij* is weight **from** *j* **to** *i*

• Order units (topological sort)
❑ Label from input, hidden, to output unit(s): 1,…, n

• Apply an input pattern ***X****(p)*
[0,..,1] or [-1,…,1]



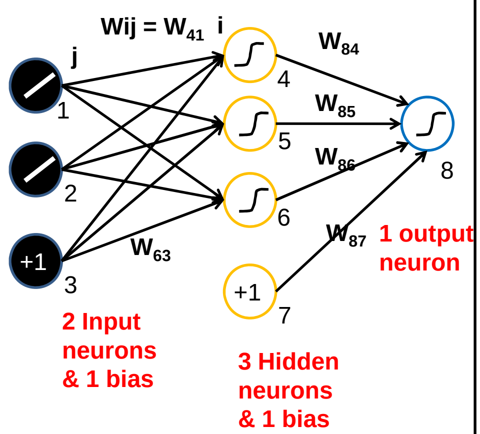
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

•*Wij* is weight **from** *j* **to** *i*

• Order units (topological sort)
❑ Label from input, hidden, to output unit(s): 1,…, n

• Apply an input pattern ***X(p)***
[0,..,1] or [-1,…,1]

• For each unit *i*
❑ Compute $a_i = \sum w_{ij} y_j$



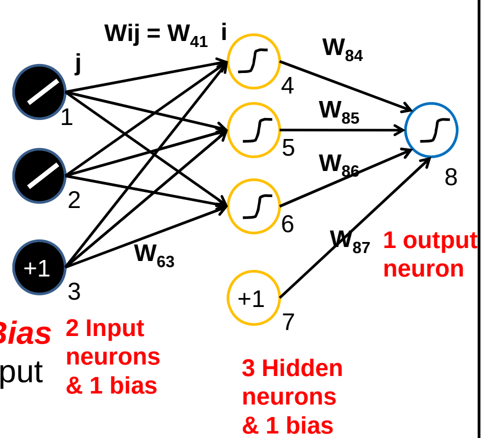
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

•*Wij* is weight **from** *j* **to** *i*

• Order units (topological sort)
❑ Label from input, hidden, to output unit(s): 1,…, n

• Apply an input pattern ***X(p)***
[0,..,1] or [-1,…,1]

• For each unit *i*
❑ Compute $a_i = \sum w_{ij} y_j + W_{bias}\, Bias$
❑ Bias included as a constant input with its weight



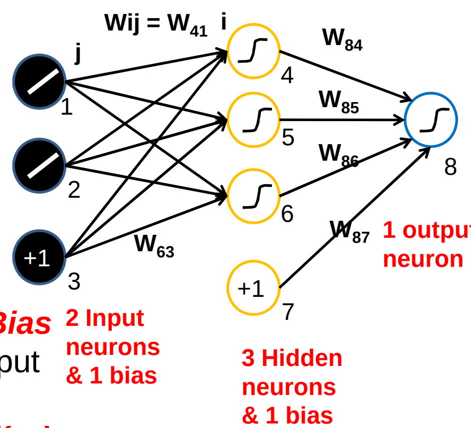
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

•*Wij* is weight **from** *j* **to** *i*

• Order units (topological sort)
❑ Label from input, hidden, to output unit(s): 1,…, n

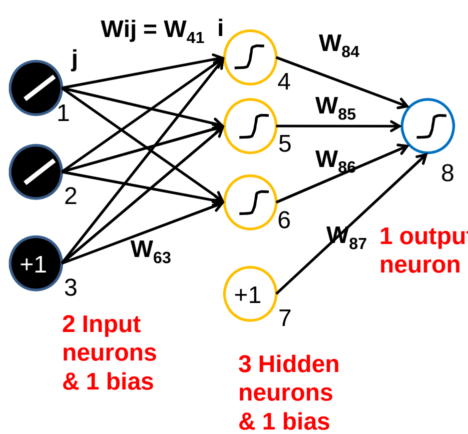
• Apply an input pattern ***X****(p)*
[0,..,1] or [-1,…,1]

• For each unit *i*
❑ Compute $a_i = \sum w_{ij}\, y_j + W_{bias}\, Bias$
❑ Bias included as a constant input with its weight
❑ Apply activation function $y_i = f(a_i)$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:
1) Calculating activations:

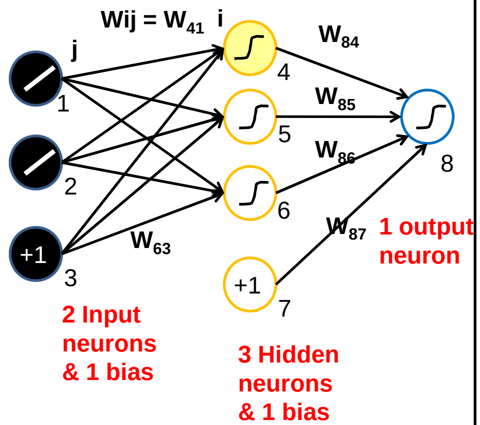


Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

a_4 = *????*

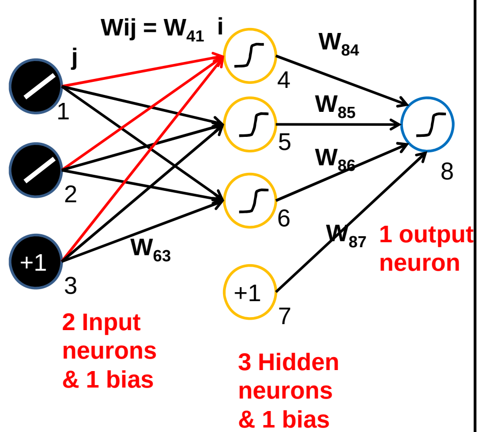


Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41}\, y_1 + w_{42}\, y_2 + w_{43} * 1;$



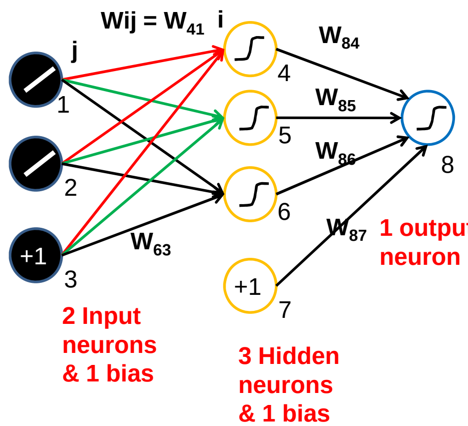
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$

$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

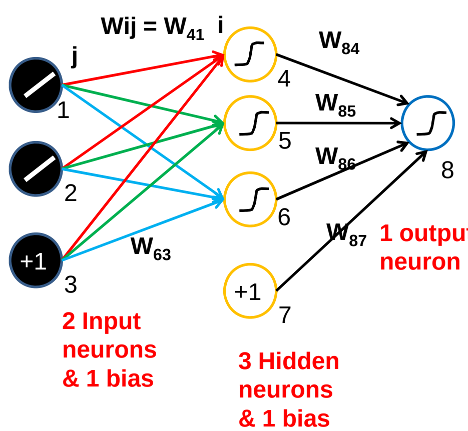
Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$

$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$

$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

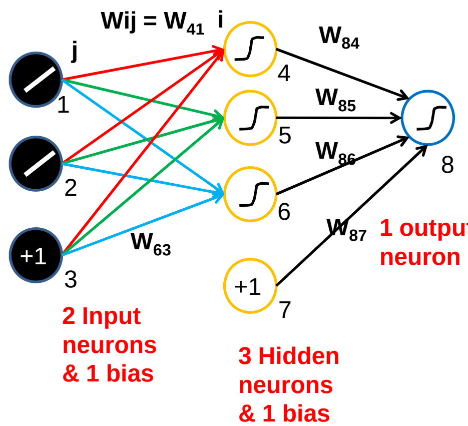
1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$

$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$

$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

2) Calculating activities:



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$

$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$

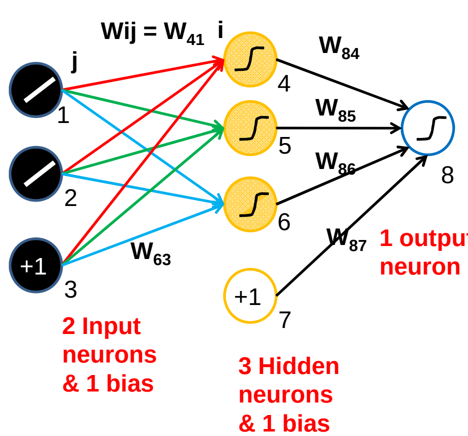
$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

2) Calculating activities:

$y_4 = f(a_4);$

$y_5 = f(a_5);$

$y_6 = f(a_6);$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

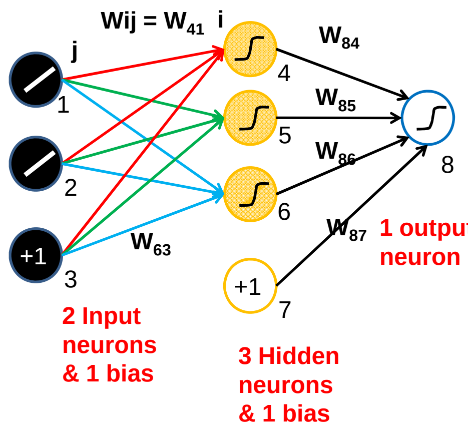
$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$
$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$
$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

2) Calculating activities:

$y_4 = f(a_4);$
$y_5 = f(a_5);$
$y_6 = f(a_6);$

Output layer:

3) Calculating activation:



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$
$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$
$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

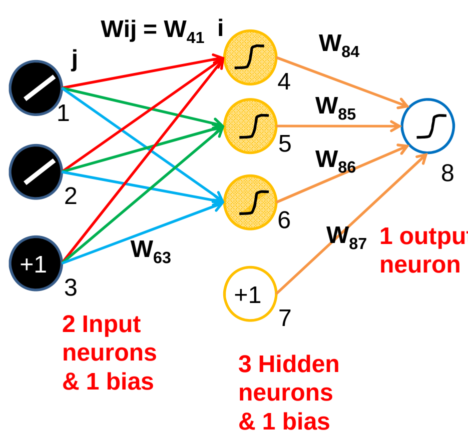
2) Calculating activities:

$y_4 = f(a_4);$
$y_5 = f(a_5);$
$y_6 = f(a_6);$

Output layer:

3) Calculating activation:

$a_8 = w_{84} y_4 + w_{85} y_5 + w_{86} y_6 + w_{87} * 1;$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$
$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$
$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

2) Calculating activities:

$y_4 = f(a_4);$
$y_5 = f(a_5);$
$y_6 = f(a_6);$

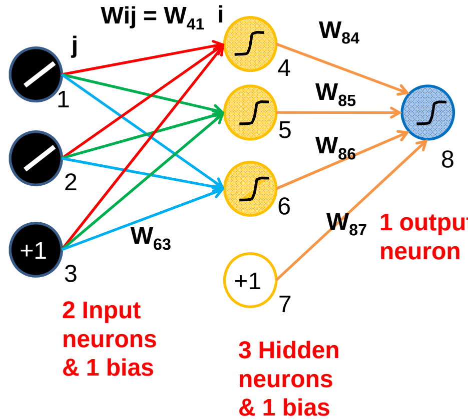
Output layer:

3) Calculating activation:

$a_8 = w_{84} y_4 + w_{85} y_5 + w_{86} y_6 + w_{87} * 1;$

4) Calculating activity:

$y_8 = f(a_8);$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Hidden layer:

1) Calculating activations:

$a_4 = w_{41} y_1 + w_{42} y_2 + w_{43} * 1;$
$a_5 = w_{51} y_1 + w_{52} y_2 + w_{53} * 1;$
$a_6 = w_{61} y_1 + w_{62} y_2 + w_{63} * 1;$

2) Calculating activities:

$y_4 = f(a_4);$
$y_5 = f(a_5);$
$y_6 = f(a_6);$

Output layer:

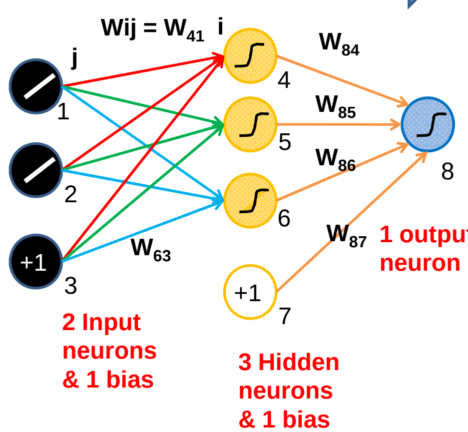
3) Calculating activation:

$a_8 = w_{84} y_4 + w_{85} y_5 + w_{86} y_6 + w_{87} * 1;$

4) Calculating activity:

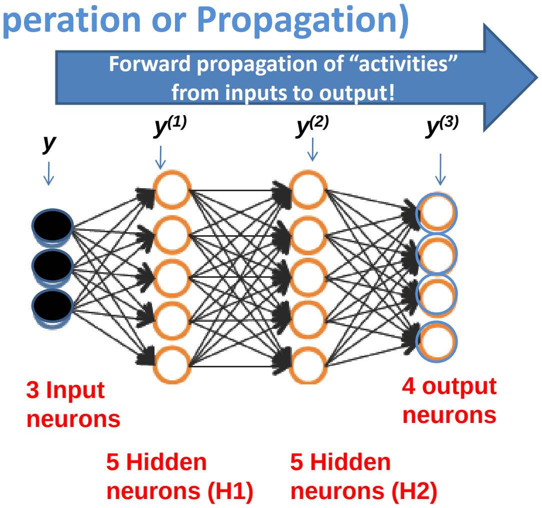
$y_8 = f(a_8);$

Forward propagation of "activities" from inputs to output!



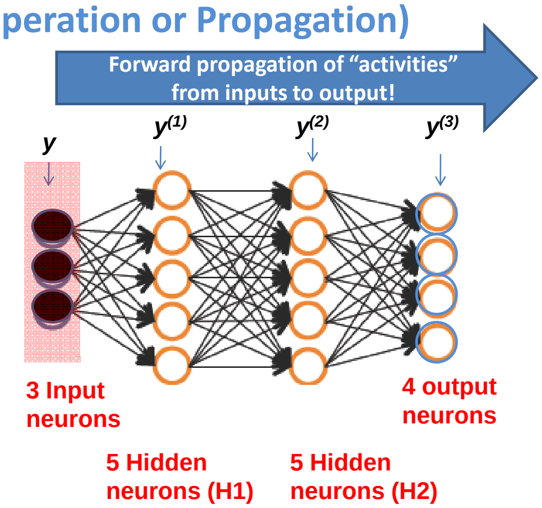
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Two hidden layers



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Two hidden layers

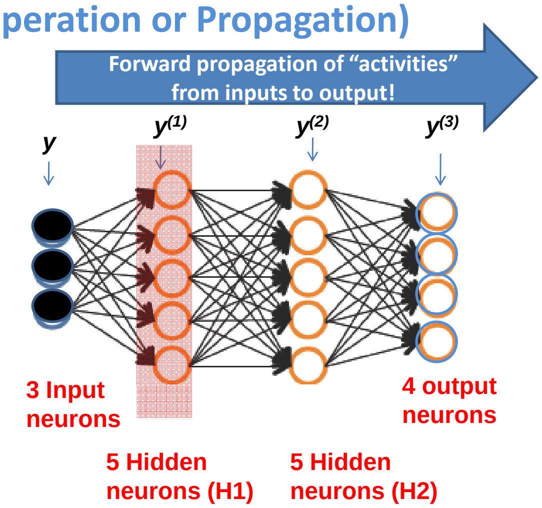


Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Two hidden layers

Hidden layer 1:

$a^{(1)} = w\,y;$
$y^{(1)} = f(a^{(1)});$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

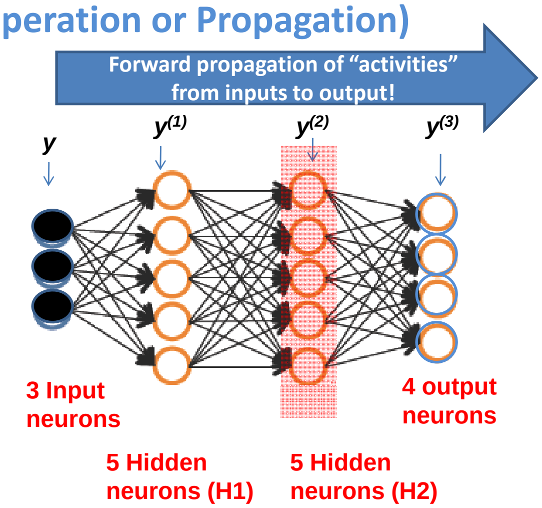
Two hidden layers

Hidden layer 1:

$a^{(1)} = w\,y;$
$y^{(1)} = f(a^{(1)});$

Hidden layer 2:

$a^{(2)} = w^{(1)}\,y^{(1)};$
$y^{(2)} = f(a^{(2)});$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Two hidden layers

Hidden layer 1:

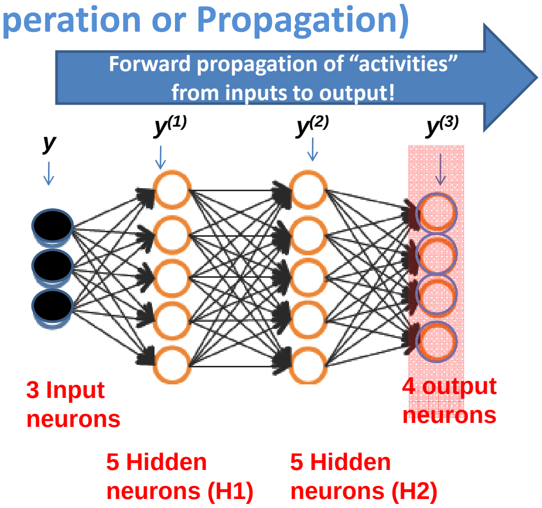
$a^{(1)} = w\, y;$
$y^{(1)} = f(a^{(1)});$

Hidden layer 2:

$a^{(2)} = w^{(1)}\, y^{(1)}\,;$
$y^{(2)} = f(a^{(2)});$

Output layer:

$a^{(3)} = w^{(2)}\, y^{(2)}\,;$
$y^{(3)} = f(a^{(3)});$



Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

Two hidden layers

Hidden layer 1:

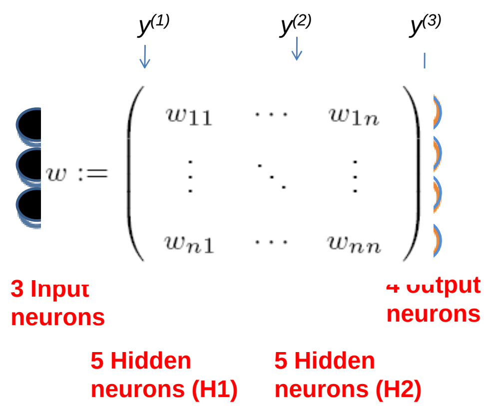
$a^{(1)} = w\, y;$
$y^{(1)} = f(a^{(1)});$

Hidden layer 2:

$a^{(2)} = w^{(1)}\, y^{(1)}\,;$
$y^{(2)} = f(a^{(2)});$

Output layer:

$a^{(3)} = w^{(2)}\, y^{(2)}\,;$
$y^{(3)} = f(a^{(3)});$



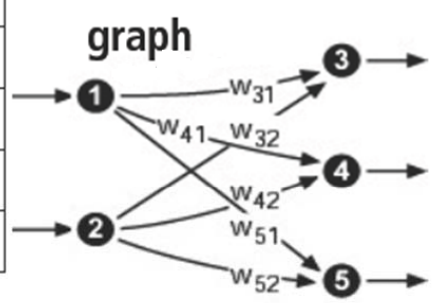
$$w := \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nn} \end{pmatrix}$$

Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

	node 1	node 2	node 3	node 4	node 5
node 1	0	0	0	0	0
node 2	0	0	0	0	0
node 3	0.7	0.4	0	0	0
node 4	1.0	-0.5	0	0	0
node 5	0.6	0.9	0	0	0

matrix

feedforward network

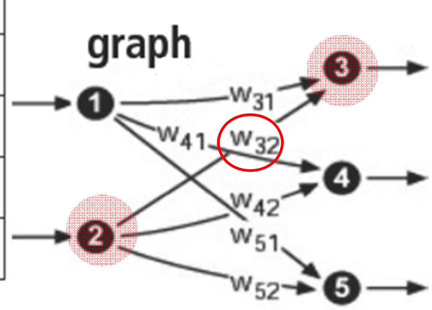


Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

	node 1	node 2	node 3	node 4	node 5
node 1	0	0	0	0	0
node 2	0	0	0	0	0
node 3	0.7	0.4 w_{32}	0	0	0
node 4	1.0	-0.5	0	0	0
node 5	0.6	0.9	0	0	0

matrix

feedforward network



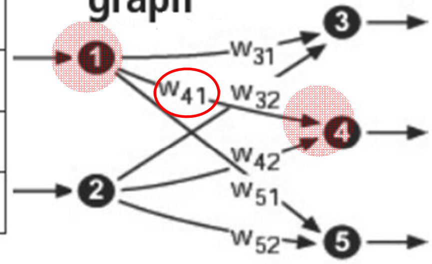
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

	node 1	node 2	node 3	node 4	node 5
node 1	0	0	0	0	0
node 2	0	0	0	0	0
node 3	0.7	0.4	0	0	0
node 4	1.0 w_{41}	-0.5	0	0	0
node 5	0.6	0.9	0	0	0

matrix

feedforward network

graph



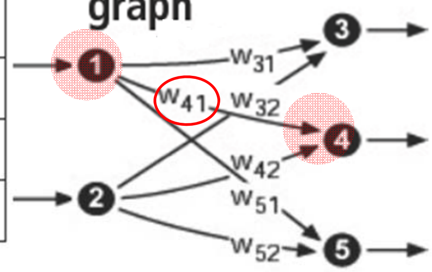
Multilayer Feedforward Neural Networks (Forward Operation or Propagation)

	node 1	node 2	node 3	node 4	node 5
node 1	0	0	0	0	0
node 2	0	0	0	0	0
node 3	0.7	0.4	0	0	0
node 4	1.0 w_{41}	-0.5	0	0	0
node 5	0.6	0.9	0	0	0

matrix

feedforward network

graph



How to learn the weights?

Today's Outline

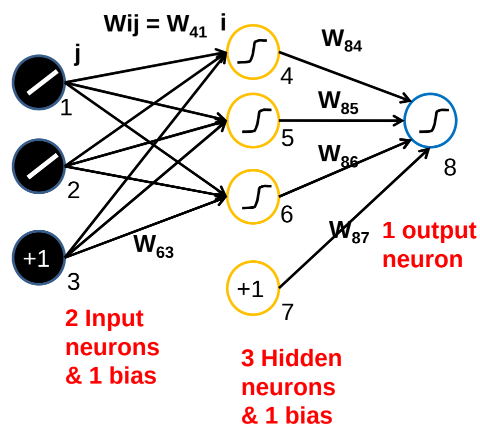
- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Today's Outline

- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

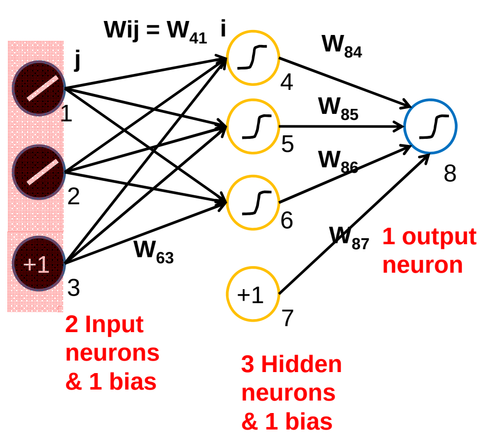
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating activities forward to obtain final output!



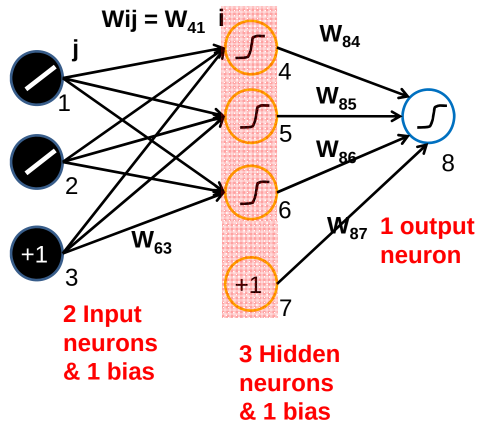
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating activities forward to obtain final output!



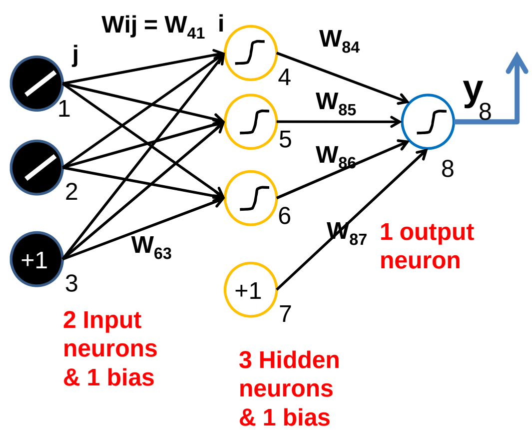
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating activities forward to obtain final output!



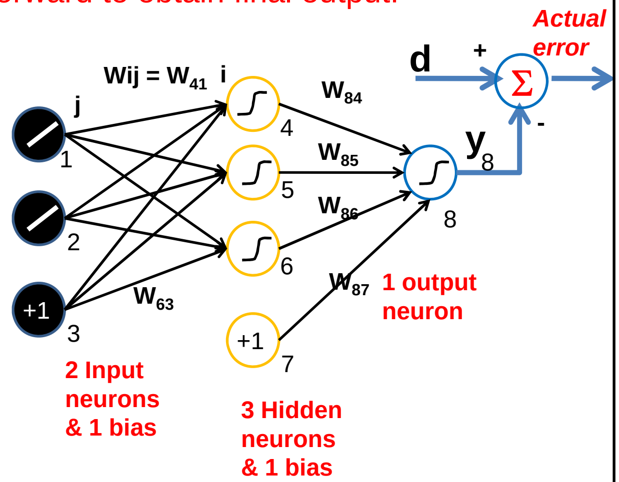
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating activities forward to obtain final output!



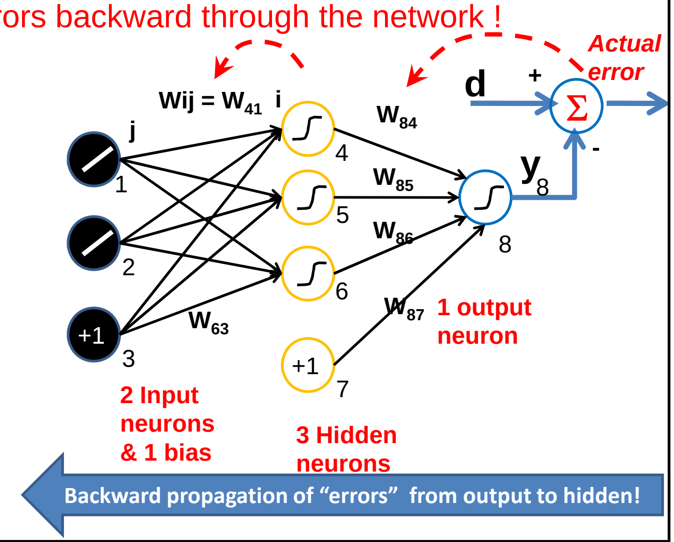
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating activities forward to obtain final output!



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

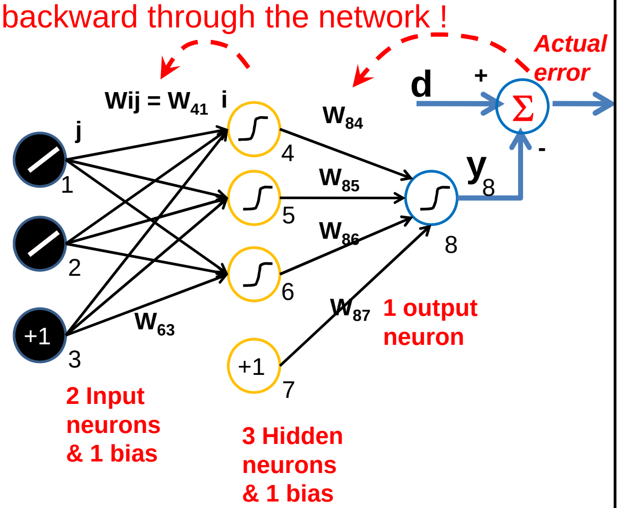


Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

- Define an Error

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



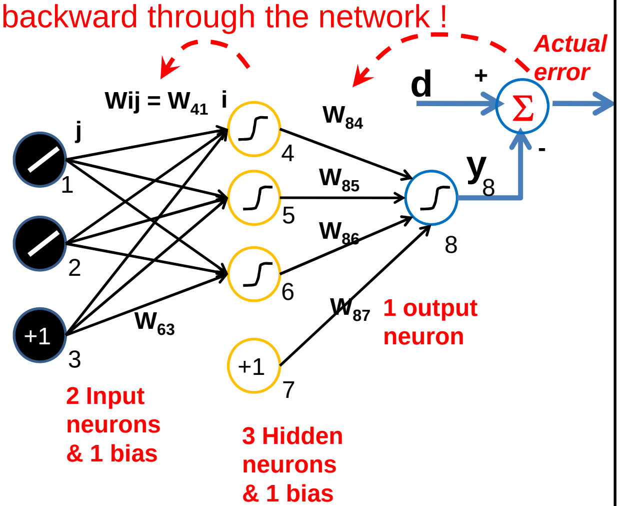
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

- Define an Error

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

- Depends on Desired Output *d* and Actual Output *y*



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

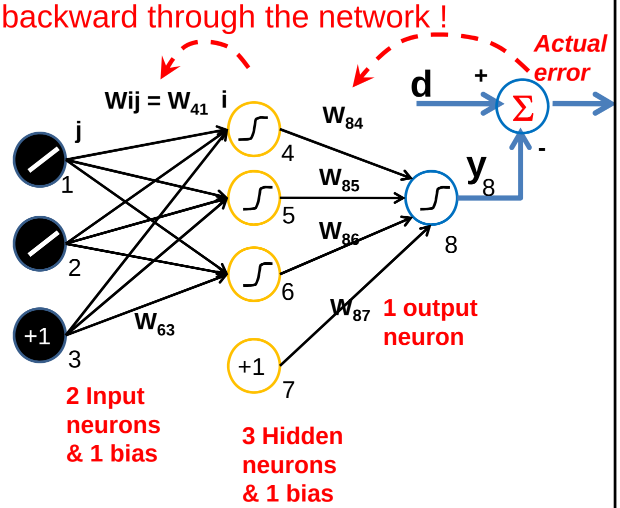
Propagating the errors backward through the network !

- Define an Error

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

- Depends on Desired Output *d* and Actual Output *y*
- Minimize it by using Gradient descent rule

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



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

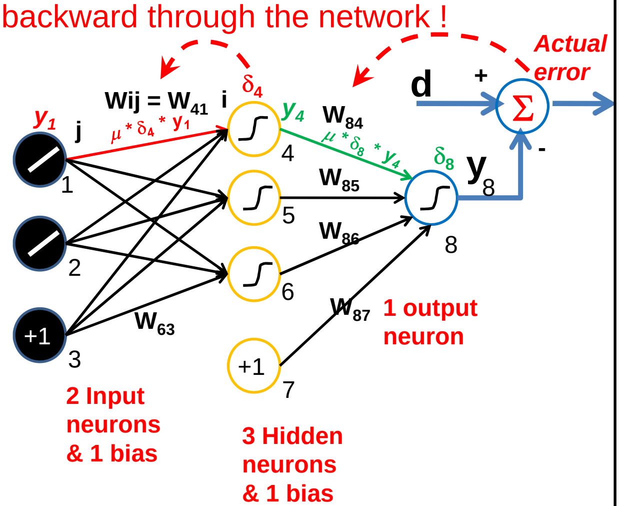
- Define an Error

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

- Depends on Desired Output *d* and Actual Output *y*
- Minimize it by using Gradient descent rule

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

$$\Delta W_{ij} = \mu * \delta_i * y_j$$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

- Define an Error

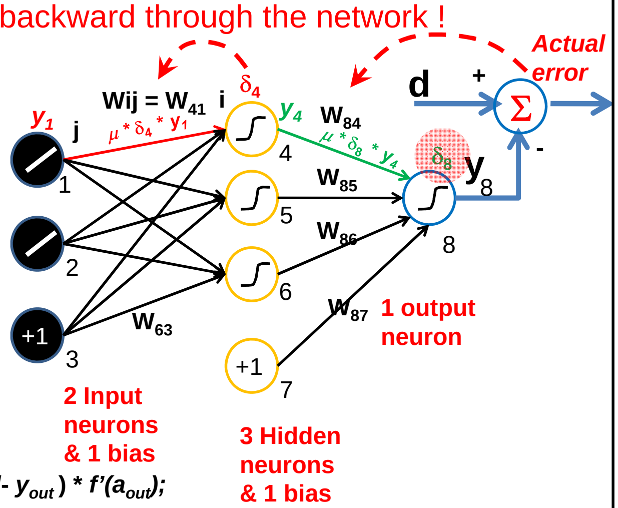
$$E = \frac{1}{2}(d - y)^2$$

- Depends on Desired Output *d* and Actual Output *y*
- Minimize it by using Gradient descent rule

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

$\Delta W_{ij} = \mu * \delta_i * y_j$

Output unit weights: $\delta_{out} = (d - y_{out}) * f'(a_{out});$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

- Define an Error

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

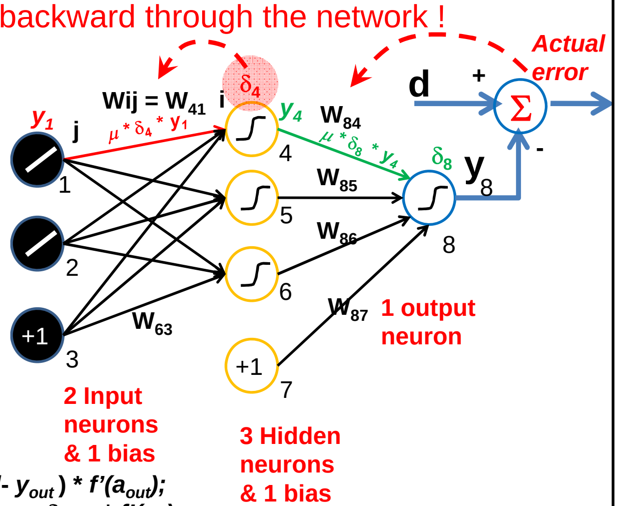
- Depends on Desired Output *d* and Actual Output *y*
- Minimize it by using Gradient descent rule

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

$\Delta W_{ij} = \mu * \delta_i * y_j$

Output unit weights: $\delta_{out} = (d - y_{out}) * f'(a_{out});$

Hidden unit weights: $\delta_h = \sum w_{out\_h}\delta_{out} * f'(a_h);$

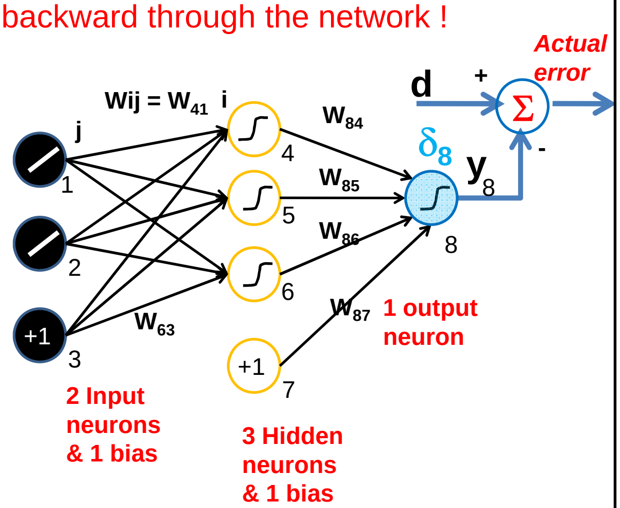


Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

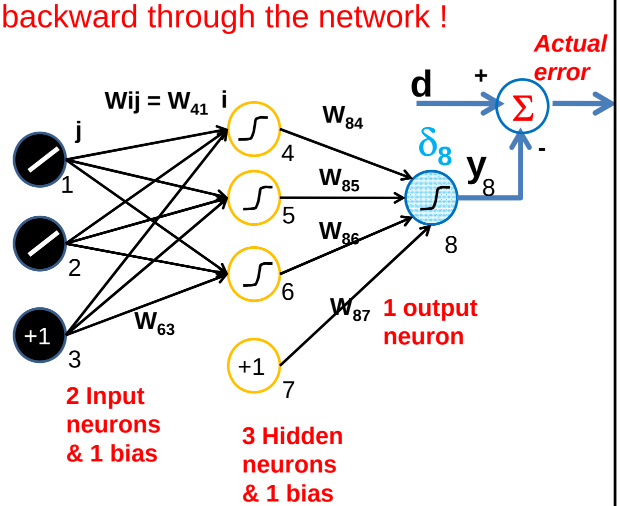


Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = \overbrace{(d - y_8)}^{\textit{Actual error}} * f'(a_8);$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

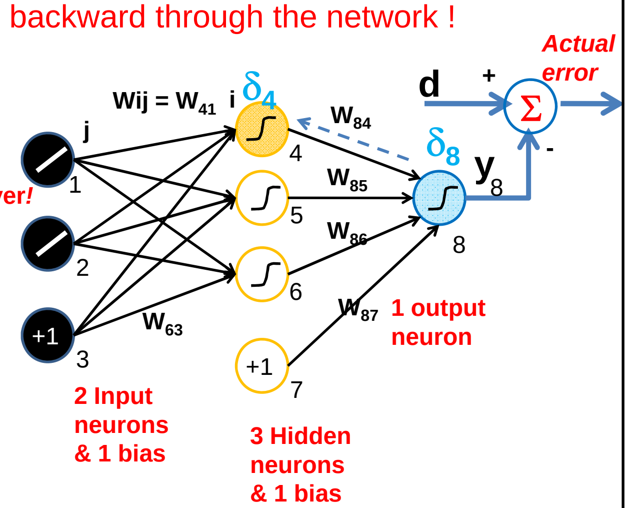
Propagating the errors backward through the network !

For each unit:

$$\delta_8 = \overbrace{(d - y_8)}^{\textbf{Actual error}} * f'(a_8);$$

All incoming δ of output layer!

$$\delta_4 = (W_{84} * \delta_8) * f'(a_4);$$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

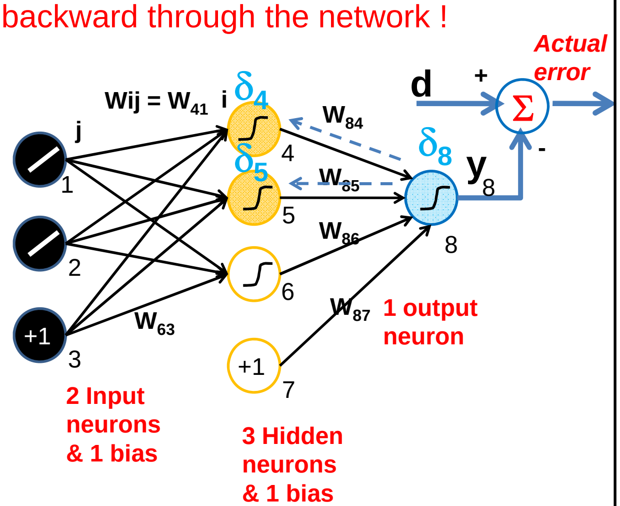
Propagating the errors backward through the network !

For each unit:

$$\delta_8 = (d - y_8) * f'(a_8);$$

$$\delta_4 = W_{84} * \delta_8 * f'(a_4);$$

$$\delta_5 = W_{85} * \delta_8 * f'(a_5);$$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

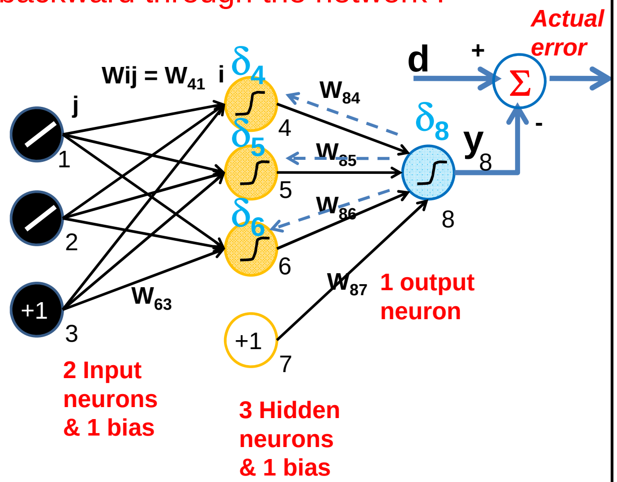
For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

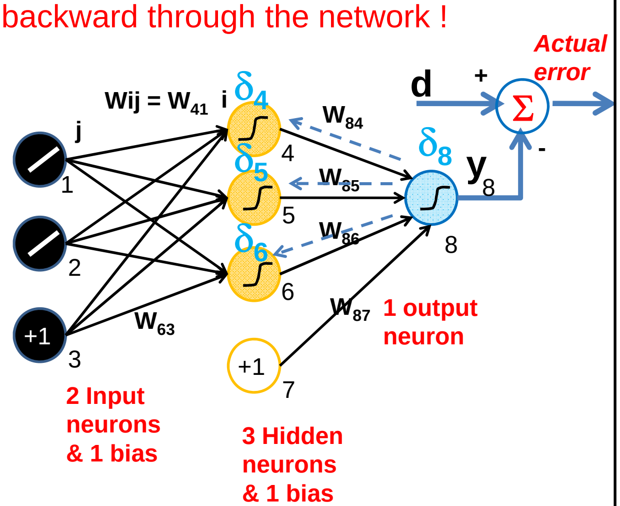
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

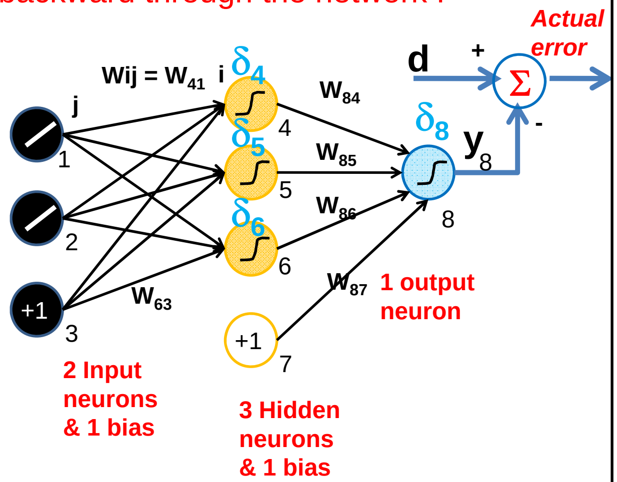
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

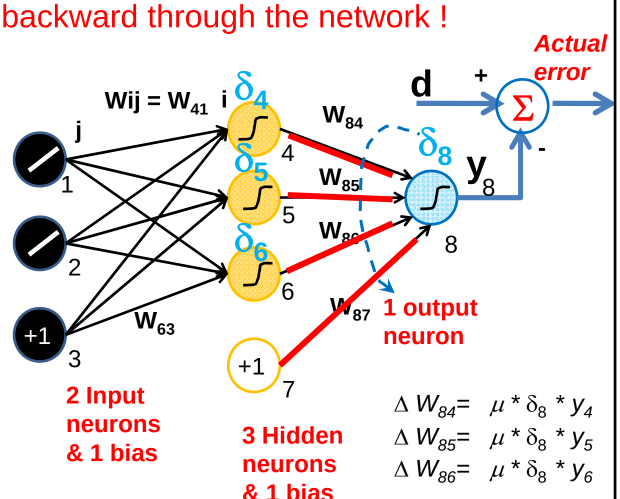
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



$\Delta W_{84} = \mu * \delta_8 * y_4$

$\Delta W_{85} = \mu * \delta_8 * y_5$

$\Delta W_{86} = \mu * \delta_8 * y_6$

$\Delta W_{87} = \mu * \delta_8 * y_7$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

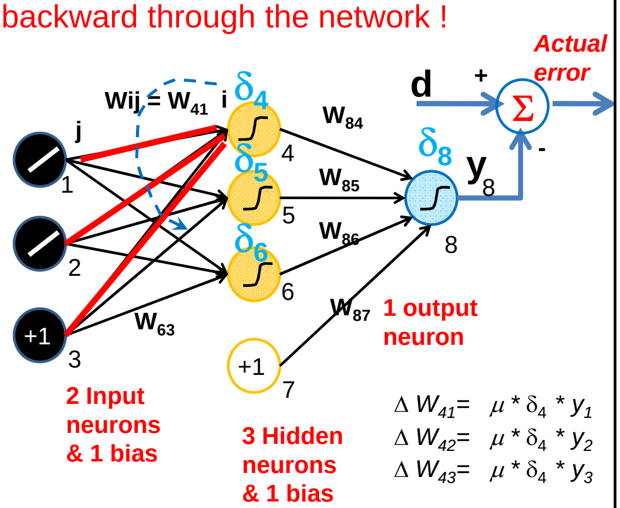
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



$\Delta W_{41} = \mu * \delta_4 * y_1$

$\Delta W_{42} = \mu * \delta_4 * y_2$

$\Delta W_{43} = \mu * \delta_4 * y_3$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

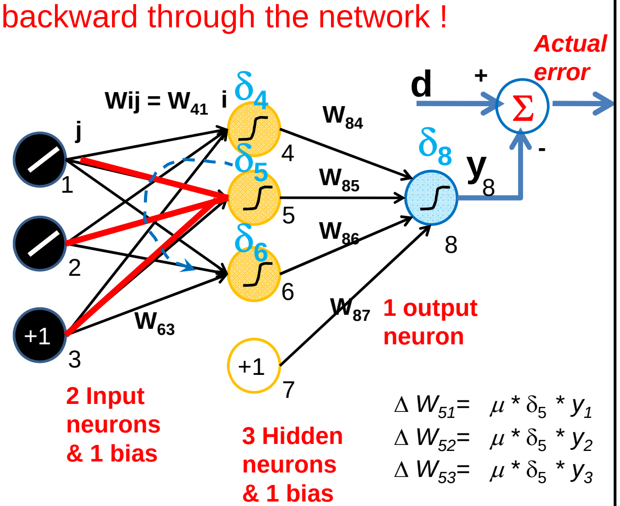
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



$\Delta W_{51} = \mu * \delta_5 * y_1$

$\Delta W_{52} = \mu * \delta_5 * y_2$

$\Delta W_{53} = \mu * \delta_5 * y_3$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

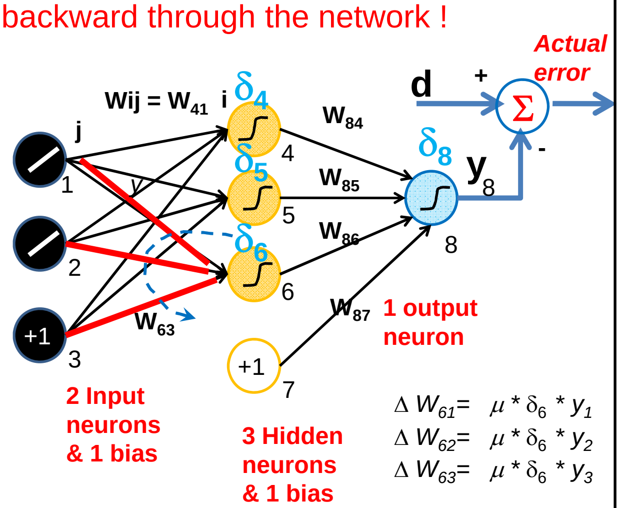
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

μ = learning rate; $0 \leq \mu < 1$



$\Delta W_{61} = \mu * \delta_6 * y_1$

$\Delta W_{62} = \mu * \delta_6 * y_2$

$\Delta W_{63} = \mu * \delta_6 * y_3$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

Linear function $f'(a) = 1$

$\delta_8 = (d - y_8) * f'(a_8);$	$\delta_8 = (d - y_8);$
$\delta_4 = W_{84} * \delta_8 * f'(a_4);$	$\delta_4 = W_{84} * \delta_8;$
$\delta_5 = W_{85} * \delta_8 * f'(a_5);$	$\delta_5 = W_{85} * \delta_8;$
$\delta_6 = W_{86} * \delta_8 * f'(a_6);$	$\delta_6 = W_{86} * \delta_8;$

= Delta rule!!

$\Delta W_{ij} = \mu * \delta_i * y_j$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

Sigmoid (Logistic transfer function ($y \in (0, 1)$).

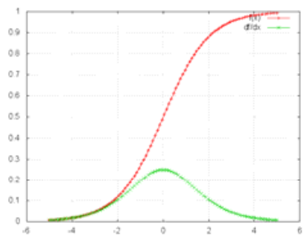
$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$



$$y = f(a) = \frac{1}{1+e^{-a}}$$

$f'(a) = f(a) * (1 - f(a)) = \boldsymbol{y*(1-y)}$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

Sigmoid (Logistic transfer function ($y \in (0, 1)$).

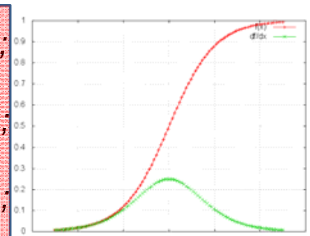
$\delta_8 = (d - y_8) * f'(a_8);$ → $\delta_8 = (d - y_8) * y_8 * (1 - y_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$ → $\delta_4 = W_{84} * \delta_8 * y_4 * (1 - y_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$ → $\delta_5 = W_{85} * \delta_8 * y_5 * (1 - y_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$ → $\delta_6 = W_{86} * \delta_8 * y_6 * (1 - y_6);$

$\Delta W_{ij} = \mu * \delta_i * y_j$



$$y = f(a) = \frac{1}{1+e^{-a}}$$

$f'(a) = f(a) * (1 - f(a)) = \boldsymbol{y*(1-y)}$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

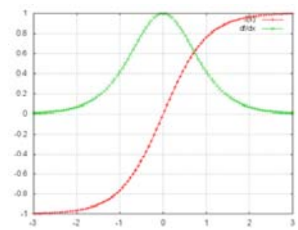
Sigmoid (tanh transfer function ($y \in (-1, 1)$)).

$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$



$\Delta W_{ij} = \mu * \delta_i * y_j$

$$y = f(a) = \tanh(a) = \frac{2}{1+e^{-2a}} - 1$$

$$f'(a) = 1 - f^2(a) = \mathbf{1 - y^2}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

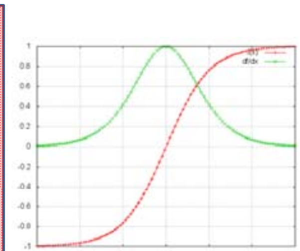
Sigmoid (tanh transfer function ($y \in (-1, 1)$)).

$\delta_8 = (d - y_8) * f'(a_8);$ → $\delta_8 = (d - y_8) * (1-(y_8)^2);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$ → $\delta_4 = W_{84} * \delta_8 * (1-(y_4)^2);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$ → $\delta_5 = W_{85} * \delta_8 * (1-(y_5)^2);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$ → $\delta_6 = W_{86} * \delta_8 * (1-(y_6)^2);$



$\Delta W_{ij} = \mu * \delta_i * y_j$

$$y = f(a) = \tanh(a) = \frac{2}{1+e^{-2a}} - 1$$

$$f'(a) = 1 - f^2(a) = \mathbf{1 - y^2}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

Heaviside step or unit step transfer function.

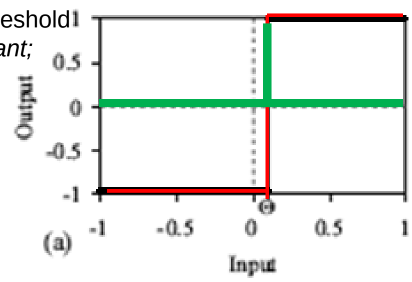
$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

if ***a*** crosses threshold
f'(a) *= a constant;*
Otherwise
f'(a) *= 0;*



$\Delta W_{ij} = \mu * \delta_i * y_j$

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ -1 & \text{if } u < \theta \end{cases}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Propagating the errors backward through the network !

For each unit:

Heaviside step or unit step transfer function.

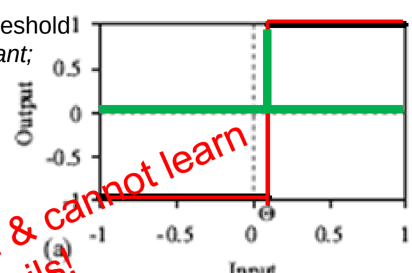
$\delta_8 = (d - y_8) * f'(a_8);$

$\delta_4 = W_{84} * \delta_8 * f'(a_4);$

$\delta_5 = W_{85} * \delta_8 * f'(a_5);$

$\delta_6 = W_{86} * \delta_8 * f'(a_6);$

if ***a*** crosses threshold
f'(a) *= a constant;*
Otherwise
f'(a) *= 0;*



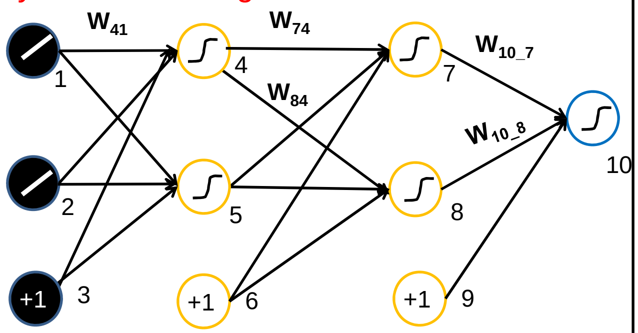
No gradient!! & cannot learn
Perceptron fails!

$\Delta W_{ij} = \mu * \delta_i * y_j$

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ -1 & \text{if } u < \theta \end{cases}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

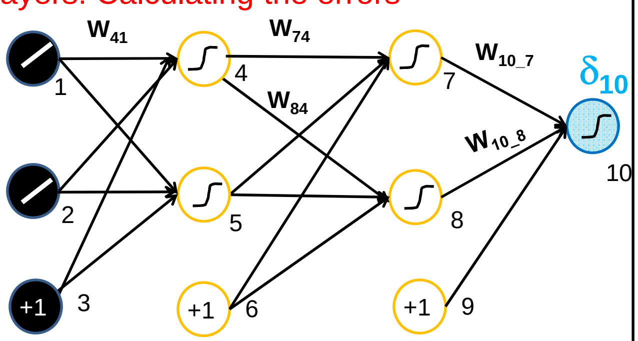


Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:

$\delta_{10} = (d - y_{10}) * f'(a_{10});$



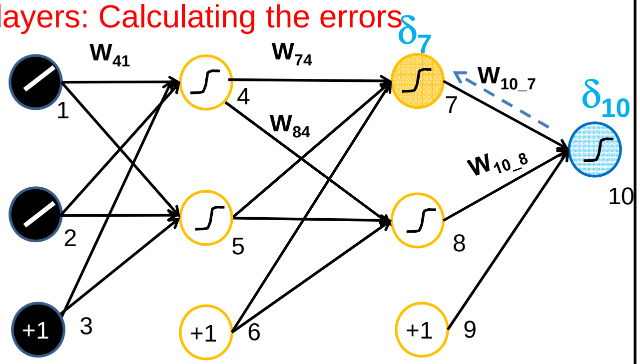
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:

$\delta_{10} = (d - y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

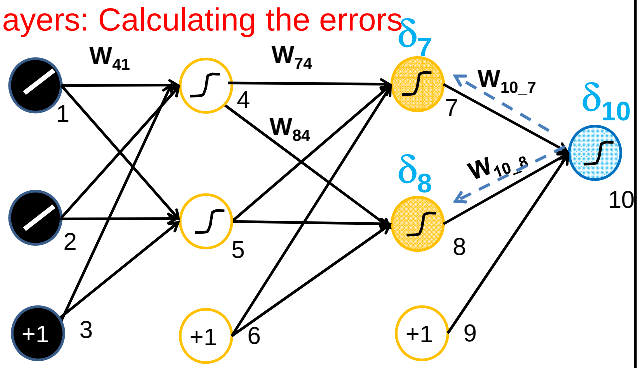
Example of 2 hidden layers: Calculating the errors

For each unit:

$\delta_{10} = (d - y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$

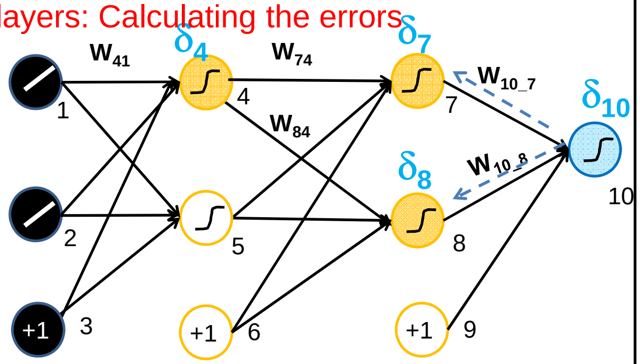
$\delta_8 = W_{10\_8} * \delta_{10} * f'(a_8);$



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:



$\delta_{10} = (d- y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$

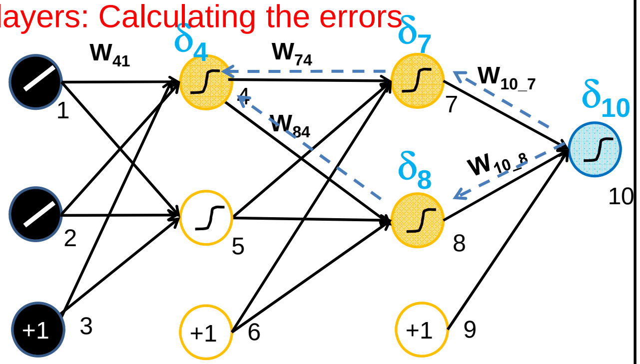
$\delta_8 = W_{10\_8} * \delta_{10} * f'(a_8);$

$\delta_4 =$????

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:



$\delta_{10} = (d- y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$

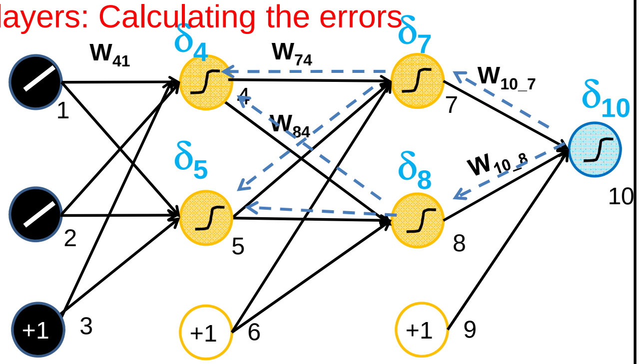
$\delta_8 = W_{10\_8} * \delta_{10} * f'(a_8);$

$\delta_4 = (W_{74} * \delta_7 + W_{84} * \delta_8) * f'(a_4);$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:



$\delta_{10} = (d - y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$

$\delta_8 = W_{10\_8} * \delta_{10} * f'(a_8);$

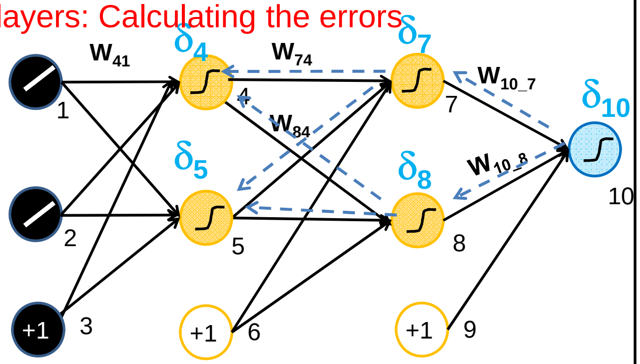
$\delta_4 = (W_{74} * \delta_7 + W_{84} * \delta_8) * f'(a_4);$

$\delta_5 = (W_{75} * \delta_7 + W_{85} * \delta_8) * f'(a_5);$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Example of 2 hidden layers: Calculating the errors

For each unit:



$\delta_{10} = (d - y_{10}) * f'(a_{10});$

$\delta_7 = W_{10\_7} * \delta_{10} * f'(a_7);$

$\delta_8 = W_{10\_8} * \delta_{10} * f'(a_8);$

$\delta_4 = (W_{74} * \delta_7 + W_{84} * \delta_8) * f'(a_4);$

$\delta_5 = (W_{75} * \delta_7 + W_{85} * \delta_8) * f'(a_5);$

$\Delta W_{ij} = \mu * \delta_i * y_j$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:

For each ($\vec{x}$, $\vec{d}$) in training examples, Do

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:
For each $(\vec{x}, \vec{d})$ in training examples, Do

Propagate the input forward through the network
1) Input (the instance) $\vec{x}$ to the network and compute output y_u of every unit u in the network.

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:
For each $(\vec{x}, \vec{d})$ in training examples, Do

Propagate the input forward through the network
1) Input (the instance) $\vec{x}$ to the network and compute output y_u of every unit u in the network.

Propagate the errors backward through the network
2) For each network output unit k, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:
For each $(\vec{x}, \vec{d})$ in training examples, Do

Propagate the input forward through the network
1) Input (the instance) $\vec{x}$ to the network and compute output y_u of every unit u in the network.

Propagate the errors backward through the network
2) For each network output unit k, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

3) For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow f'(a_h) * \sum_{k \in \text{ layer } m+1} (w_{kh} * \delta_k)$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

Summary of the algorithm:
For each $(\vec{x}, \vec{d})$ in training examples, Do

Propagate the input forward through the network
1) Input (the instance) $\vec{x}$ to the network and compute output y_u of every unit u in the network.

Propagate the errors backward through the network
2) For each network output unit k, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

3) For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow f'(a_h) * \sum_{k \in \text{ layer } m+1} (w_{kh} * \delta_k)$$

4) Update each network weight w_{ij}

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad ; \Delta w_{ij} = \mu * \delta_i * y_j \; ; \quad \mu = \text{learning rate; } 0 \le \mu < 1$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How often should we update weights?

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How often should we update weights?

❑ Online learning = update weights at each step or each training data (each step).

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How often should we update weights?

❑ Online learning = update weights at each step or each training data (each step).

❑ Full-batch learning = update weights after a full presentation of all training data (each epoch) where all weight changes are summed over the presentation.

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How often should we update weights?

❑ Online learning = update weights at each step or each training data (each step).

❑ Full-batch learning = update weights after a full presentation of all training data (each epoch) where all weight changes are summed over the presentation.

❑ Mini-batch learning = update weights after a certain set of all training data where all weight changes are summed over the set. It is good for large neural networks with very large and highly redundant training sets

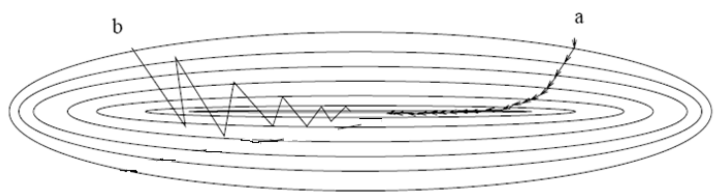
Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Large learning rate (oscillation→ unstable) ***Small learning rate (slow)***

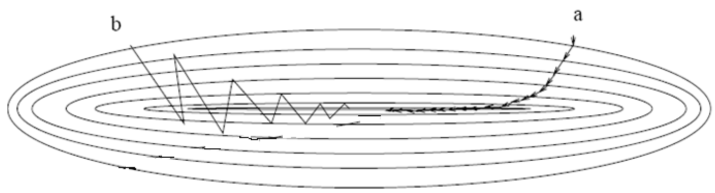


$\Delta w_{ij}(n) = \mu * \delta_i * y_j$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Large learning rate (oscillation → unstable) *Small learning rate (slow)*



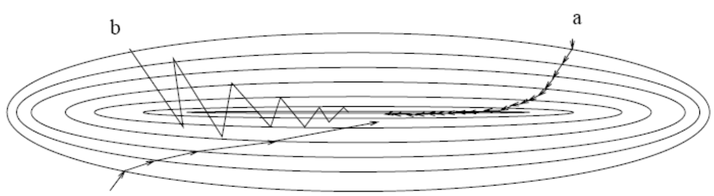
$$\Delta w_{ij}(n) = \mu * \delta_i * y_j$$

How to make the learning fast and without oscillation?

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Large learning rate (oscillation → unstable) *Small learning rate (slow)*



Large learning rate with momentum term (faster convergence)

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j + \alpha \Delta w_j(n-1)$$

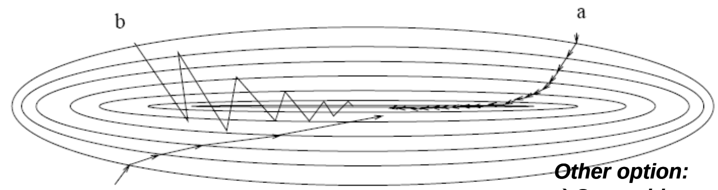
$0 \le \alpha < 1$ is a constant called the momentum term (e.g., 0.9)
$\Delta w_{ij}(n)$ is the weight update performed during the n^{th} iteration

Adding momentum→ Avoid sudden change of directions of weight update, smoothing the learning process!

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Large learning rate (oscillation → unstable) ***Small learning rate (slow)***



Large learning rate with momentum term (faster convergence)

Other option:
→Start with small momentum (α = 0.5)
→ Then increase it to 0.9

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j + \alpha \Delta w_j(n-1)$$

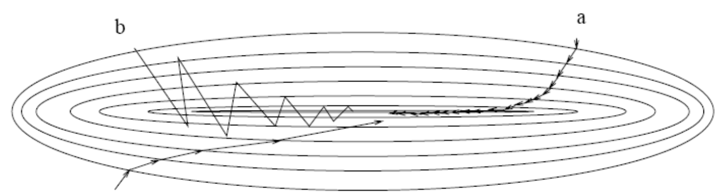
$0 \leq \alpha < 1$is a constant called the momentum term (e.g., 0.9)
$\Delta w_{ij}(n)$ is the weight update performed during the n^{th} iteration

<u>Adding momentum→</u> Avoid sudden change of directions of weight update, smoothing the learning process!

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- How much to update? → Setting learning rate!

Large learning rate (oscillation → unstable) ***Small learning rate (slow)***



Large learning rate with momentum term (faster convergence)

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j - \frac{\frac{\partial E}{\partial w_i(t)}}{\left[\frac{\frac{\partial E}{\partial w_i(t)} - \frac{\partial E}{\partial w_i(t-1)}}{\Delta w_i(t-1)}\right]}$$

The Fahlman Variation (Quickprop, Fahlman, S. 1988)→Dynamic momentum

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- Can we make adaptive learning rate?

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- Can we make adaptive learning rate?

- ❑ "Adaptive learning rate for each connection"

$$\Delta w_{ij} = -\varepsilon\, g_{ij} \frac{\partial E}{\partial w_{ij}}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- Can we make adaptive learning rate?

❑ "Adaptive learning rate for each connection"

→Start with a local gain ***g*** of 1 for every weight.

$$\Delta w_{ij} = -\varepsilon \, g_{ij} \, \frac{\partial E}{\partial w_{ij}}$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- Can we make adaptive learning rate?

❑ "Adaptive learning rate for each connection"

→Start with a local gain ***g*** of 1 for every weight.

$$\Delta w_{ij} = -\varepsilon \, g_{ij} \, \frac{\partial E}{\partial w_{ij}}$$

→If the gradient for that weight does not change sign, then increase the local gain by a small additive value.

$$if \quad \left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$$

$$then \quad g_{ij}(t) = g_{ij}(t-1) + .05$$

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))

- Can we make adaptive learning rate?

❑ "Adaptive learning rate for each connection"

→Start with a local gain ***g*** of 1 for every weight.

$$\Delta w_{ij} = -\varepsilon\, g_{ij} \frac{\partial E}{\partial w_{ij}}$$

→If the gradient for that weight does not change sign, then increase the local gain by a small additive value.

→ But if the gradient changes sign, then decrease the gain by a multiplicative value.

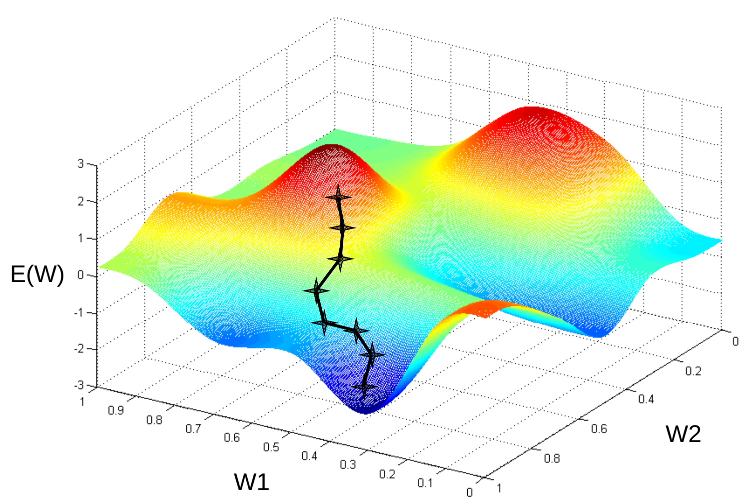
$$if \quad \left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$$

$$then \quad g_{ij}(t) = g_{ij}(t-1) + .05$$

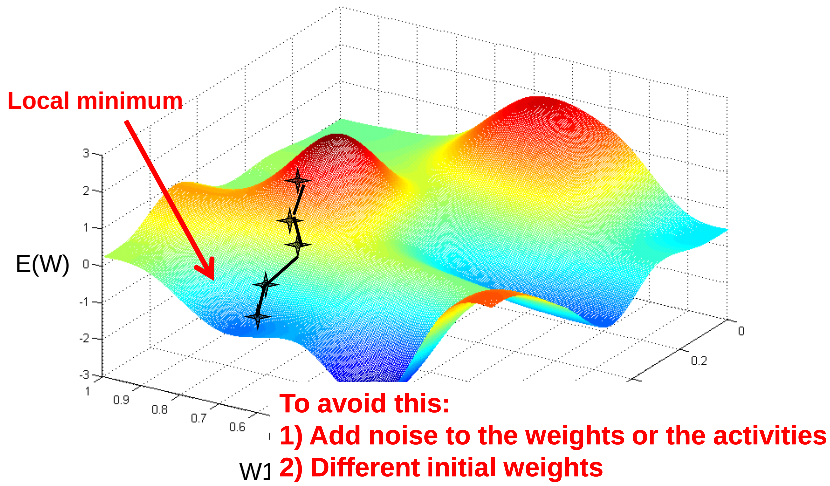
$$else \quad g_{ij}(t) = g_{ij}(t-1) * .95$$

The gain needs to be limited in a reasonable range e.g., [0.1, 10]

Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))



Multilayer Feedforward Neural Networks (Backpropagation Algorithm (Supervised Learning))



Today's Outline

- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Today's Outline

- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

- Present input pattern $\vec{x}^{(p)}$

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit *k*, calculate its error δ_k

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}(p)$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit *k*, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}^{(p)}$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit *k*, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

- ❑ For each hidden unit *h*, calculate its error term δ_h (backward mode)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}^{(p)}$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit *k*, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

- ❑ For each hidden unit *h*, calculate its error term δ_h (backward mode)

$$\delta_h \leftarrow f'(a_h) * \sum_{k \in \text{ layer } m+1} (w_{kh} * \delta_k)$$

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}^{(p)}$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit k, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

- ❑ For each hidden unit h, calculate its error term δ_h (backward mode)

$$\delta_h \leftarrow f'(a_h) * \sum_{k \in \text{layer } m+1} (w_{kh} * \delta_k)$$

- ❑ Update weights: $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$; $\Delta w_{ij} = \mu * \delta_i * y_j$; (Batch VS Online)

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

1) Initialize random weights
2) Initialize neural activities (e.g., 0.0)
3) For each training pattern $\vec{x}^{(p)}$:

- ❑ Present input pattern $\vec{x}^{(p)}$
- ❑ Compute activations and then activities of hidden units (forward mode)
- ❑ Compute activations and then activities of output units (forward mode)
- ❑ For each network output unit k, calculate its error δ_k

$$\delta_k \leftarrow f'(a_k) * (d_k - y_k)$$

- ❑ For each hidden unit h, calculate its error term δ_h (backward mode)

$$\delta_h \leftarrow f'(a_h) * \sum_{k \in \text{layer } m+1} (w_{kh} * \delta_k)$$

- ❑ Update weights: $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$; $\Delta w_{ij} = \mu * \delta_i * y_j$; (Batch VS Online)

4) If Average Error is SMALL, then STOP else loop to step 3

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

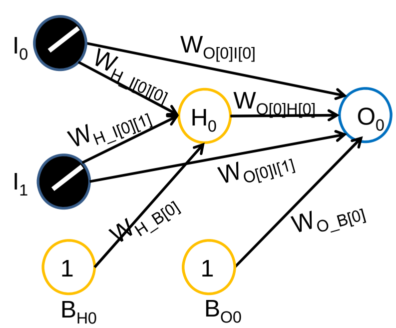
X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

What kind of transfer function of hidden and output neurons should we use?



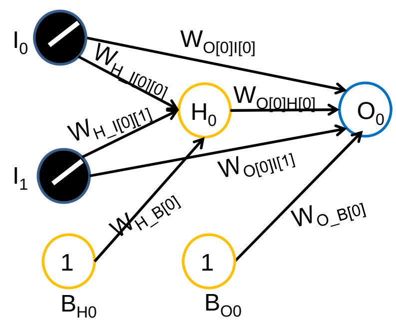
Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

//Initialization

W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

//Initialization

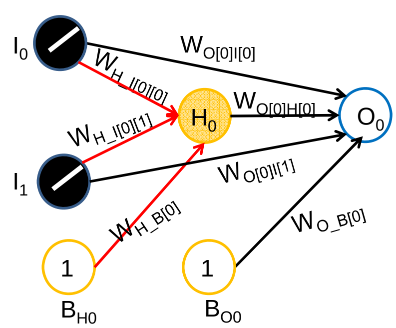
W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities

//Forward propagation

a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] * WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0] = 1./(1.+exp(- a_Hidden[0])); // Logistic function

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

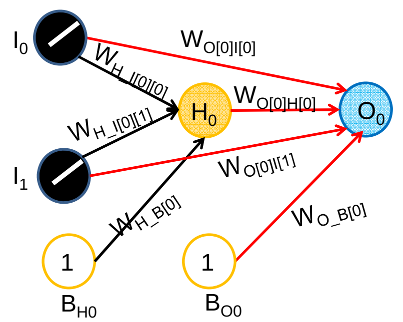
X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

//Initialization

```
W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities
```

//Forward propagation

```
a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] *
WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0]  = 1./(1.+exp(- a_Hidden[0] )); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] *
WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] *
WeightO_I[0][1];

o_Output[0] = 1./(1.+exp(- a_Output[0] )); // Logistic function
```



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

//Initialization

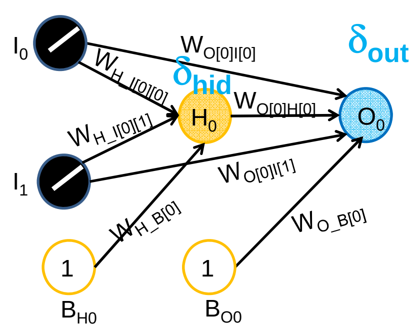
```
W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities
```

//Forward propagation

```
a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] *
WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0]  = 1./(1.+exp(- a_Hidden[0] )); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] *
WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] *
WeightO_I[0][1];

o_Output[0] = 1./(1.+exp(- a_Output[0] )); // Logistic function
```

//Backward propagation

```
deltaOutput = o_Output[0]*(1-o_Output[0])*(d-o_Output[0]);

deltaHidden[0] = o_Hidden[0] *(1-o_Hidden[0])
*(WeightO_H[0][0]*deltaOutput);
```



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

//Initialization

W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities

//Forward propagation

a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] * WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0] = 1./(1.+exp(- a_Hidden[0])); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] * WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] * WeightO_I[0][1];

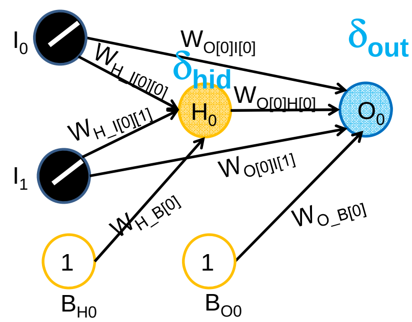
o_Output[0] = 1./(1.+exp(- a_Output[0])); // Logistic function

//Backward propagation

deltaOutput = o_Output[0]*(1-o_Output[0])*(d-o_Output[0]);

deltaHidden[0] = o_Hidden[0] *(1-o_Hidden[0]) *(WeightO_H[0][0]*deltaOutput);

→ *f'(a)*



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

//Initialization

W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities

//Forward propagation

a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] * WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0] = 1./(1.+exp(- a_Hidden[0])); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] * WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] * WeightO_I[0][1];

o_Output[0] = 1./(1.+exp(- a_Output[0])); // Logistic function

//Backward propagation

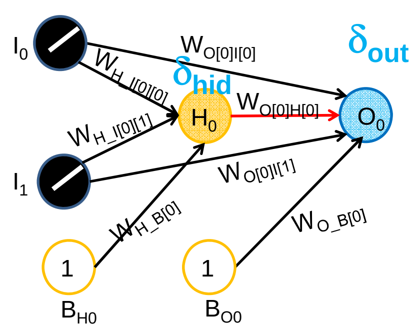
deltaOutput = o_Output[0]*(1-o_Output[0])*(d-o_Output[0]);

deltaHidden[0] = o_Hidden[0] *(1-o_Hidden[0]) *(WeightO_H[0][0]*deltaOutput);

//Weight update

DeltaWeightO_H[0][0] = BP_LEARNING*deltaOutput*o_Hidden[0];

WeightO_H[0][0] = WeightO_H[0][0]+DeltaWeightO_H[0][0]; …….

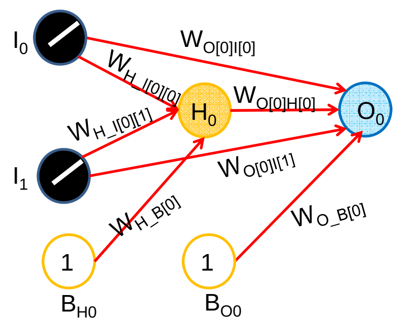


Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

```
//Initialization
W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities
//Forward propagation
a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] *
WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0]  = 1./(1.+exp(- a_Hidden[0] )); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] *
WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] *
WeightO_I[0][1];

o_Output[0] = 1./(1.+exp(- a_Output[0] )); // Logistic function
//Backward propagation
deltaOutput = o_Output[0]*(1-o_Output[0])*(d-o_Output[0]);

deltaHidden[0] = o_Hidden[0] *(1-o_Hidden[0])
*(WeightO_H[0][0]*deltaOutput);
//Weight update
DeltaWeightO_H[0][0] = BP_LEARNING*deltaOutput*o_Hidden[0];

WeightO_H[0][0] = WeightO_H[0][0]+DeltaWeightO_H[0][0]; .......
```

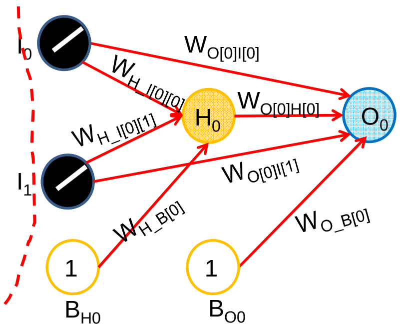


Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

Example: XNOR (Classification)

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	1

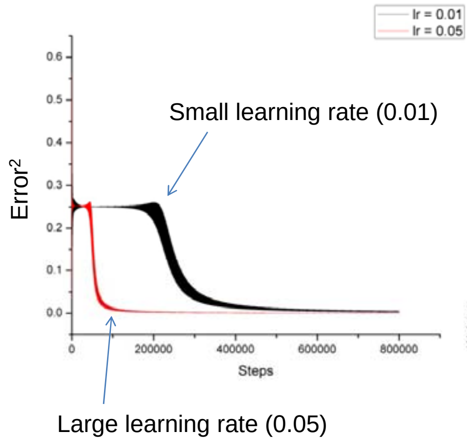
```
//Initialization
W = (double)(rand()%1000)/1000.0; // All weights
a = 0.0; // Hidden & Output neural activations
o = 0.0; // Hidden & Output neural activities
//Forward propagation
a_Hidden[0] = BiasH[0] * WeightH_B[0] + Input[0] *
WeightH_I[0][0] + Input[1] * WeightH_I[0][1];

o_Hidden[0]  = 1./(1.+exp(- a_Hidden[0] )); // Logistic function

a_Output[0] = BiasO[0] * WeightO_B[0] + o_Hidden[0] *
WeightO_H[0][0] + Input[0] * WeightO_I[0][0] + Input[1] *
WeightO_I[0][1];

o_Output[0] = 1./(1.+exp(- a_Output[0] )); // Logistic function
//Backward propagation
deltaOutput = o_Output[0]*(1-o_Output[0])*(d-o_Output[0]);

deltaHidden[0] = o_Hidden[0] *(1-o_Hidden[0])
*(WeightO_H[0][0]*deltaOutput);
//Weight update
DeltaWeightO_H[0][0] = BP_LEARNING*deltaOutput*o_Hidden[0];

WeightO_H[0][0] = WeightO_H[0][0]+DeltaWeightO_H[0][0]; .......
```

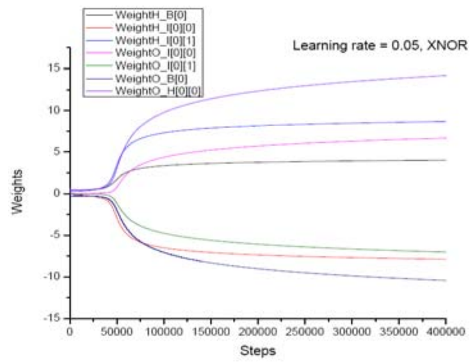


Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)

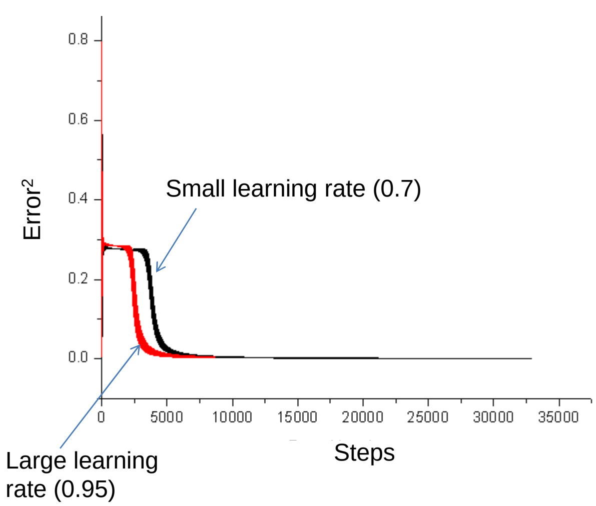
Example: XNOR (Classification)

X	Y	O	O_0
0	0	1	0.98
0	1	0	0.025
1	0	0	0.020
1	1	1	0.976

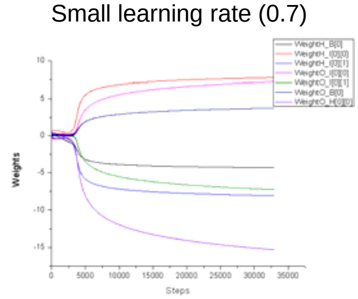




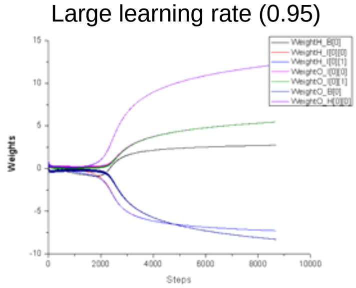
Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)



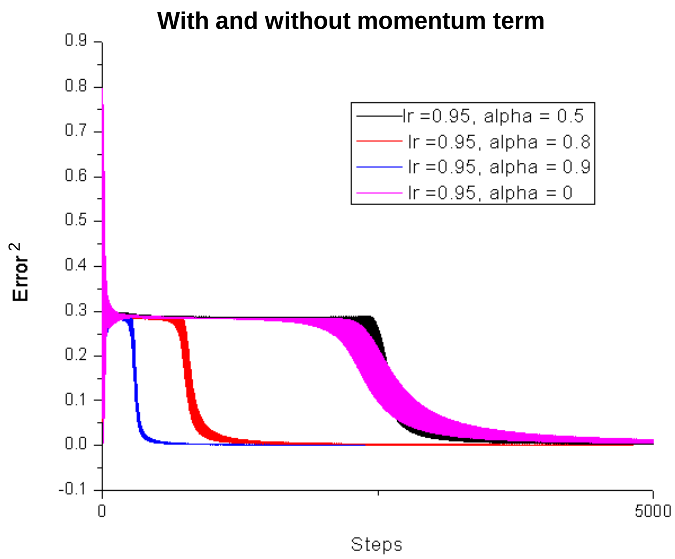
Small learning rate (0.7)



Large learning rate (0.95)



Multilayer Feedforward Neural Networks (Implementation→ Putting all things together)



Practicalities

- Representing input/output data
- Experiments
- Remarks

Representing Input Data

Different Kinds of Data require Different Techniques

Representing Input Data

Different Kinds of Data require Different Techniques

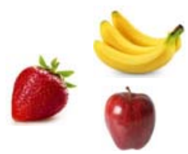
- Categorical Data

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding



			X_1	X_2	X_3	X_n
Fruits:	Apple	→	1	0	0	... 0
	Banana	→	0	1	0	... 0
	Strawberry	→	0	0	1	0
	n	→	0	0	0	1

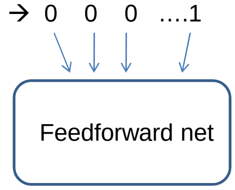
Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

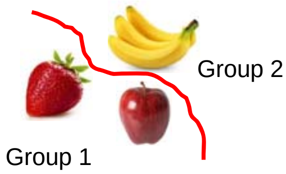
			X_1	X_2	X_3	X_n
Fruits:	Apple	→	1	0	0	... 0
	Banana	→	0	1	0	... 0
	Strawberry	→	0	0	1	0
	n	→	0	0	0	1



Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data



Use 1-of-N encoding

Fruits:		X_1	X_2	X_3	X_n
	Apple →	1	0	0	… 0
	Banana →	0	1	0	… 0
	Strawberry →	0	0	1	….0
	n →	0	0	0	….1

Feedforward net

Classification

Group1: 0

Group2: 1

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

Fruits:		X_1	X_2	X_3	X_n
	Apple →	1	0	0	… 0
	Banana →	0	1	0	… 0
	Strawberry →	0	0	1	….0
	n →	0	0	0	….1

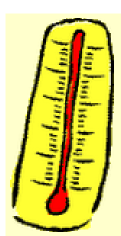
- Ordinal Data

Very hot

Hot

Warm

Cold



Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

		X_1	X_2	X_3	...... X_n
Fruits:	Apple →	1	0	0	... 0
	Banana →	0	1	0	... 0
	Strawberry →	0	0	1	0
	n →	0	0	0	1

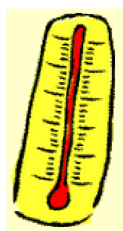
- Ordinal Data

Very hot

Hot

Warm

Cold



Use thermometer code

	X_1	X_2	X_3
Very hot →	0	0	0
Hot →	0	0	1
Warm →	0	1	1
Cold →	1	1	1

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

		X_1	X_2	X_3	...... X_n
Fruits:	Apple →	1	0	0	... 0
	Banana →	0	1	0	... 0
	Strawberry →	0	0	1	0
	n →	0	0	0	1

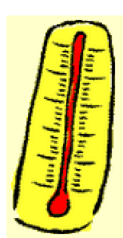
- Ordinal Data

Very hot

Hot

Warm

Cold



Use thermometer code

4 levels

	X_1	X_2	X_3	
Very hot →	0	0	0	
Hot →	0	0	1	Group 1
Warm →	0	1	1	
Cold →	1	1	1	Group 2

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

		X_1	X_2	X_3	...... X_n
Fruits: Apple	→	1	0	0	... 0
Banana	→	0	1	0	... 0
Strawberry	→	0	0	1	0
n	→	0	0	0	1

- Ordinal Data

Grade	Evaluation
A	Excellent
A-	Excellent
B+	Very Good
B	Good
B-	Good
C+	Above Average
C	Average
C-	Below Average
D+	Less than Acceptable
D	Less than Acceptable
F	Failure

Use thermometer code

5 levels

		X_1	X_2	X_3	X_4
A	→	1	1	1	1
A-	→	0	1	1	1
B+	→	0	0	1	1
B	→	0	0	0	1
B-	→	0	0	0	0

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data

Use 1-of-N encoding

		X_1	X_2	X_3	...... X_n
Fruits: Apple	→	1	0	0	... 0
Banana	→	0	1	0	... 0
Strawberry	→	0	0	1	0
n	→	0	0	0	1

- Ordinal Data

Grade	Evaluation
A	Excellent
A-	Excellent
B+	Very Good
B	Good
B-	Good
C+	Above Average
C	Average
C-	Below Average
D+	Less than Acceptable
D	Less than Acceptable
F	Failure

Use thermometer code

5 levels

		X_1	X_2	X_3	X_4
A	→	1	1	1	1
A-	→	0	1	1	1
B+	→	0	0	1	1
B	→	0	0	0	1
B-	→	0	0	0	0

Group 1
Group 2
Group 3

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding

Fruits:

		X_1	X_2	X_3	X_n
Apple	→	1	0	0	... 0
Banana	→	0	1	0	... 0
Strawberry	→	0	0	1	0
n	→	0	0	0	1

- Ordinal Data — Use thermometer code

Grade	Evaluation
A	Excellent
A-	Excellent
B+	Very Good
B	Good
B-	Good
C+	Above Average
C	Average
C-	Below Average
D+	Less than Acceptable
D	Less than Acceptable
F	Failure

5 levels

		X_1	X_2	X_3	X_4	
A	→	1	1	1	1	Group 1
A-	→	0	1	1	1	
B+	→	0	0	1	1	Group 2
B	→	0	0	0	1	Group 3
B-	→	0	0	0	0	

- Numerical Data — Input Scaling [-1,…,1], [0,…,1]

Representing Input Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding
- Ordinal Data — Use thermometer code
- Numerical Data — Input Scaling

Representing Input/Output Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding
- Ordinal Data — Use thermometer code
- Numerical Data — Input Scaling

Similar considerations for Outputs:

Representing Input/Output Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding
- Ordinal Data — Use thermometer code
- Numerical Data — Input Scaling

Similar considerations for Outputs:

- Choice of Items — One unit per item

Representing Input/Output Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding
- Ordinal Data — Use thermometer code
- Numerical Data — Input Scaling

Similar considerations for Outputs:

- Choice of Items — One unit per item
- Winner-take-all coding — Only the highest activity

Representing Input/Output Data

Different Kinds of Data require Different Techniques

- Categorical Data — Use 1-of-N encoding
- Ordinal Data — Use thermometer code
- Numerical Data — Input Scaling

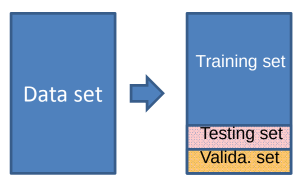
Similar considerations for Outputs:

- Choice of Items — One unit per item
- Winner-take-all coding — Only the highest activity
- Output probabilities — Probability function

Experiments

Experiments

- How to run an ANN experiment:



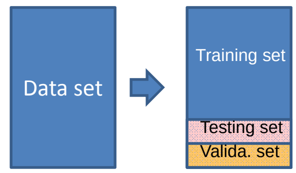
Divide patterns into 3 sets:

→Training Set (e.g., 80% of data, input-target pairs)

→Testing Set (e.g., 10% of data, unseen inputs)

→Validation Set (e.g., 10% of data, unseen inputs)

Experiments



- How to run an ANN experiment:

  Divide patterns into 3 sets:

  →Training Set (e.g., 80% of data, input-target pairs)

  →Testing Set (e.g., 10% of data, unseen inputs)

  →Validation Set (e.g., 10% of data, unseen inputs)
- **Train** using Back-Prop with **Training Set**
- **Check** the generalization of the trained net using **Testing Set**
- If error increases when Testing Set then stop!! Using different initialization, parameter setup, etc.
- **Evaluate** your network using **Validation Set**. This Validation Set is also use for comparing the performances of different methods!!

Remarks!

Reliable ML feedforward net performance depends on:

Remarks!

Reliable ML feedforward net performance depends on:

- ❑ Careful input format selection (mapping)
- ❑ Careful experimental procedure (training, testing, validation)
- ❑ Careful choice of parameters
- Learning rate large → instability
- Learning rate small → slow convergence

Remarks!

Reliable ML feedforward net performance depends on:

- ❑ Careful input format selection (mapping)
- ❑ Careful experimental procedure (training, testing, validation)
- ❑ Careful choice of parameters
- Learning rate large → instability
- Learning rate small → slow convergence

- ❑ Static momentum
- ❑ Dynamic momentum

$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} + \boxed{\alpha\, \Delta w_i(t-1)}$$

$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} - \boxed{\frac{\dfrac{\partial E}{\partial w_i(t)}}{\left[\dfrac{\dfrac{\partial E}{\partial w_i(t)} - \dfrac{\partial E}{\partial w_i(t-1)}}{\Delta w_i(t-1)}\right]}}$$

Remarks!

Reliable ML feedforward net performance depends on:

- ❑ Careful input format selection (mapping)
- ❑ Careful experimental procedure (training, testing, validation)
- ❑ Careful choice of parameters
- Learning rate large → instability
- Learning rate small → slow convergence
  - ❑ Static momentum
  - ❑ Dynamic momentum
  - ❑ Use adaptive learning rate OR a smarter algorithm, e.g. RLS

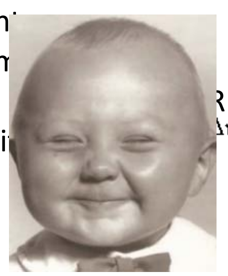
$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} + \boxed{\alpha\, \Delta w_i(t-1)}$$

$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} - \boxed{\frac{\frac{\partial E}{\partial w_i(t)}}{\left[\frac{\frac{\partial E}{\partial w_i(t)} - \frac{\partial E}{\partial w_i(t-1)}}{\Delta w_i(t-1)}\right]}}$$

Remarks!

Reliable ML feedforward net performance depends on:

- ❑ Careful input format selection (mapping)
- ❑ Careful experimental procedure (training, testing, validation)
- ❑ Careful choice of parameters
- Learning rate large → instability
- Learning rate small → slow convergence
  - ❑ Static momen
  - ❑ Dynamic mom
  - ❑ Use adaptive
  a smarter algori



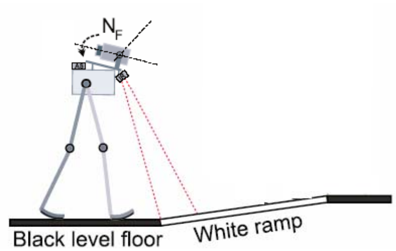
$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} + \boxed{\alpha\, \Delta w_i(t-1)}$$

$$\Delta w_i(t) = -\rho \frac{\partial E}{\partial w_i(t)} - \boxed{\frac{\frac{\partial E}{\partial w_i(t)}}{\left[\frac{\frac{\partial E}{\partial w_i(t)} - \frac{\partial E}{\partial w_i(t-1)}}{\Delta w_i(t-1)}\right]}}$$

❖ **Some LUCK!**

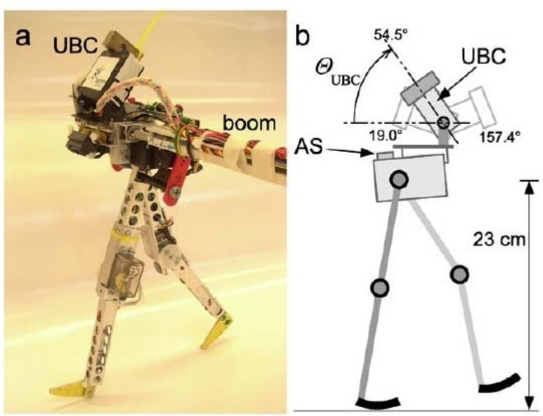
Example: Robot Control (Prediction)

Walking up slope without seeing a slope!



Example: Robot Control (Prediction)

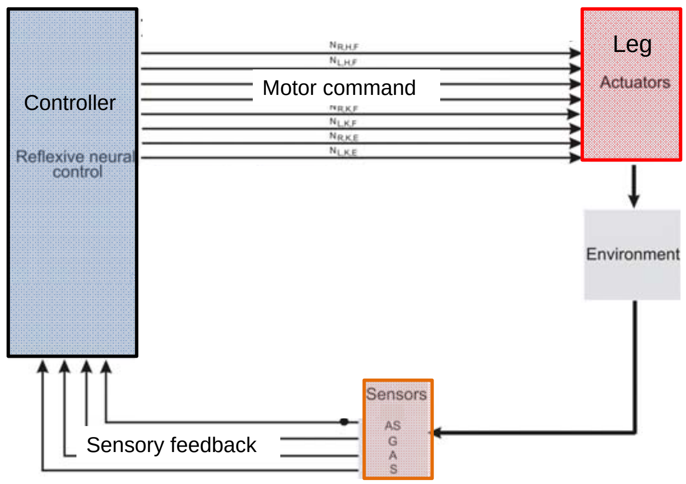
Dynamical biped robot RuBot



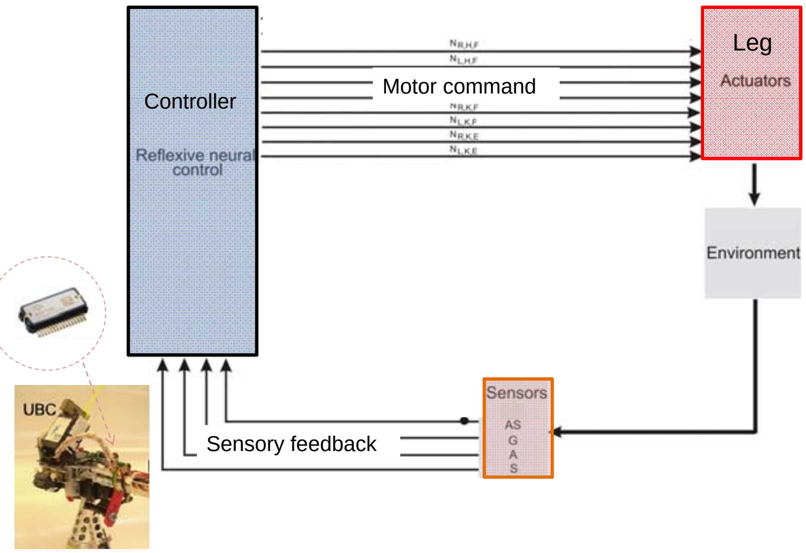
Schroeder-Schetelig, J.; Manoonpong, P. and Woergoetter, F.(2010) Using Efference Copy and a Forward Internal Model for Adaptive Biped Walking. Autonomous Robots, doi:10.1007/s10514-010-9199-7

Example: Robot Control (Prediction)

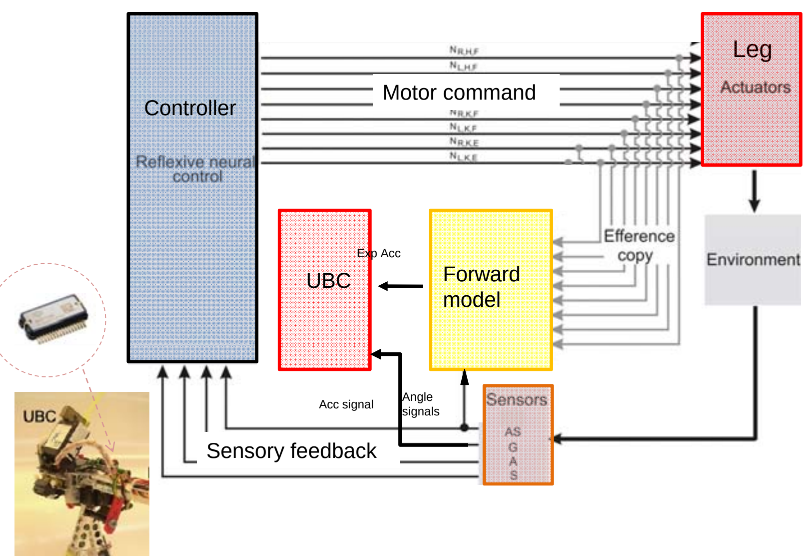
Neural locomotion control



Neural locomotion control and a forward model for walking on different terrains

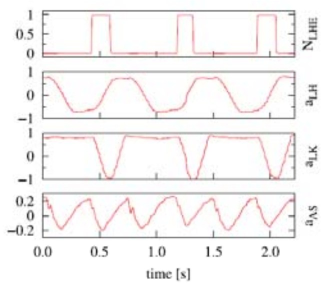


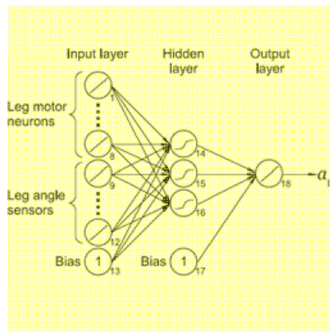
Neural locomotion control and a forward model for walking on different terrains

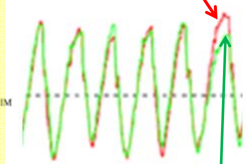


Forward model & UBC controller for walking on different terrains

Target







Training data

Forward model

Network output

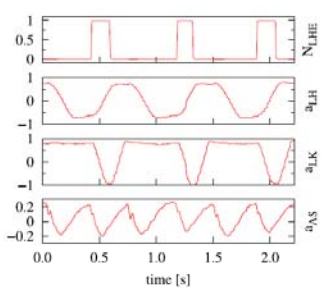
Schroeder-Schetelig, J.; Manoonpong, P. and Woergoetter, F.(2010) Using Efference Copy and a Forward Internal Model for Adaptive Biped Walking. Autonomous Robots, doi:10.1007/s10514-010-9199-7

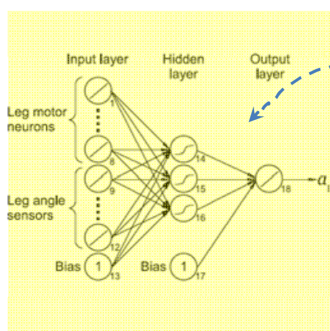
Forward model & UBC controller for walking on different terrains

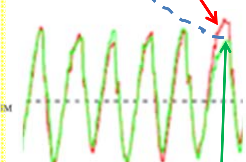
Target

Input Scaling

Error







Training data

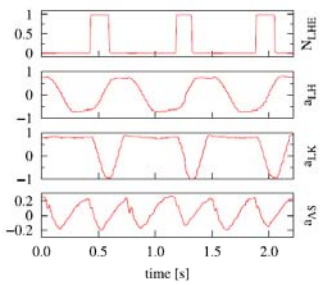
Forward model

Network output

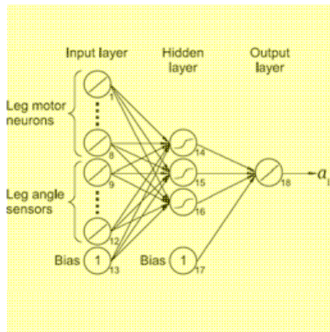
Schroeder-Schetelig, J.; Manoonpong, P. and Woergoetter, F.(2010) Using Efference Copy and a Forward Internal Model for Adaptive Biped Walking. Autonomous Robots, doi:10.1007/s10514-010-9199-7

Forward model & UBC controller for walking on different terrains

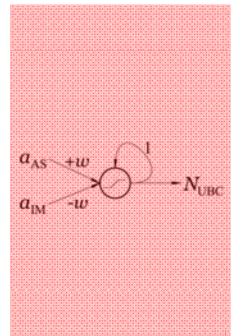
Input Scaling



Training data

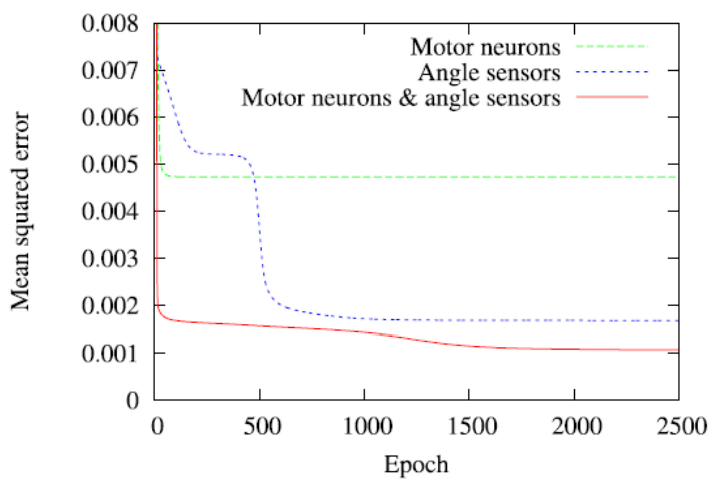


Forward model

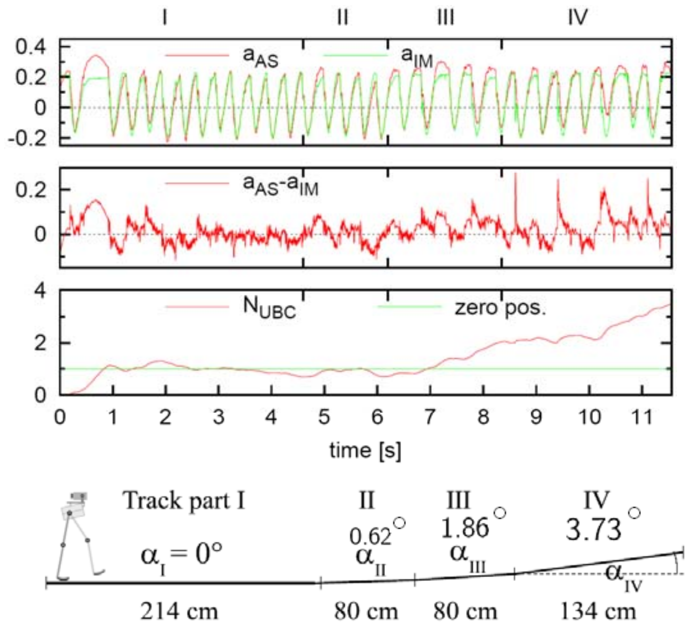


UBC controller

Schroeder-Schetelig, J.; Manoonpong, P. and Woergoetter, F.(2010) Using Efference Copy and a Forward Internal Model for Adaptive Biped Walking. Autonomous Robots, doi:10.1007/s10514-010-9199-7



Robot walking experiments



Example: Computer vision (Classification)


Pedestrian


Car


Motorcycle


Truck

Multiple Classes (Multiple output units)


Pedestrian


Car


Motorcycle


Truck

Multiple Classes (Multiple output units)


Pedestrian


Car


Motorcycle


Truck

4 classes (Outputs)!

Pedestrian

Car

Motorcycle

Truck

Multiple Classes (Multiple output units)


Pedestrian


Car


Motorcycle


Truck

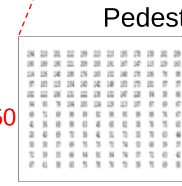
4 classes (Outputs)!

Pedestrian

Car

Motorcycle

Truck


50

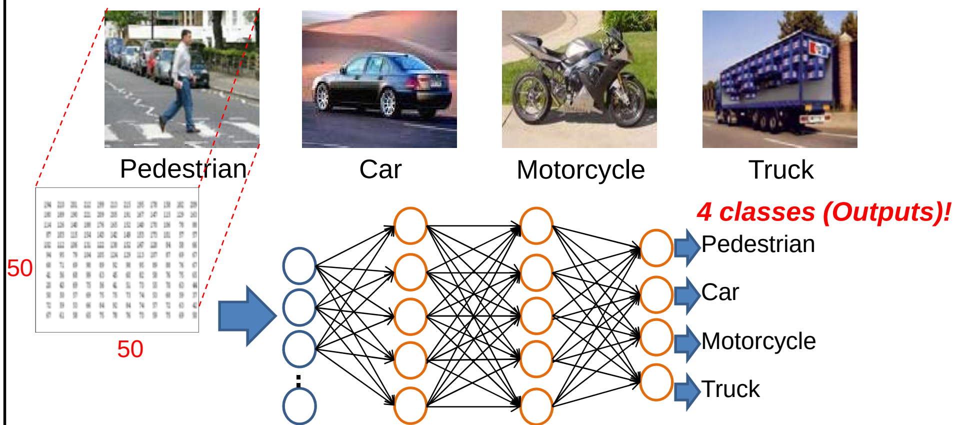
50

50 x 50 pixel images→ 2500 pixels (7500 if RGB)

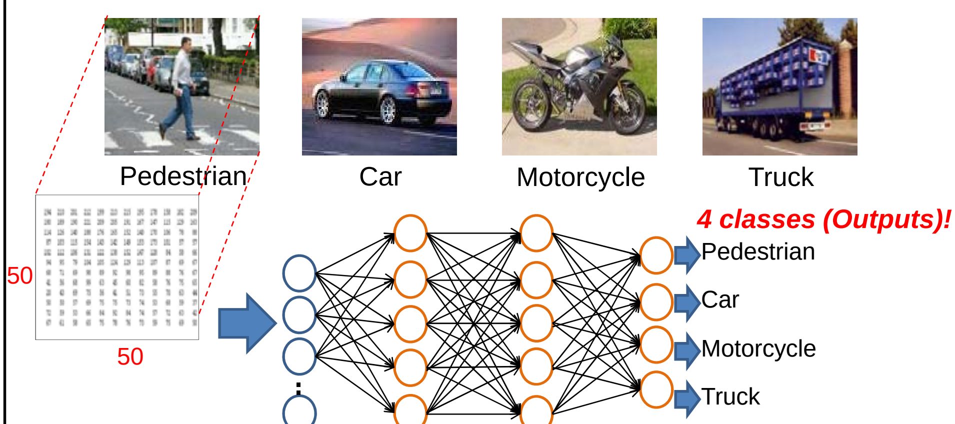
0-255

Inputs $x = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$

Multiple Classes (Multiple output units)



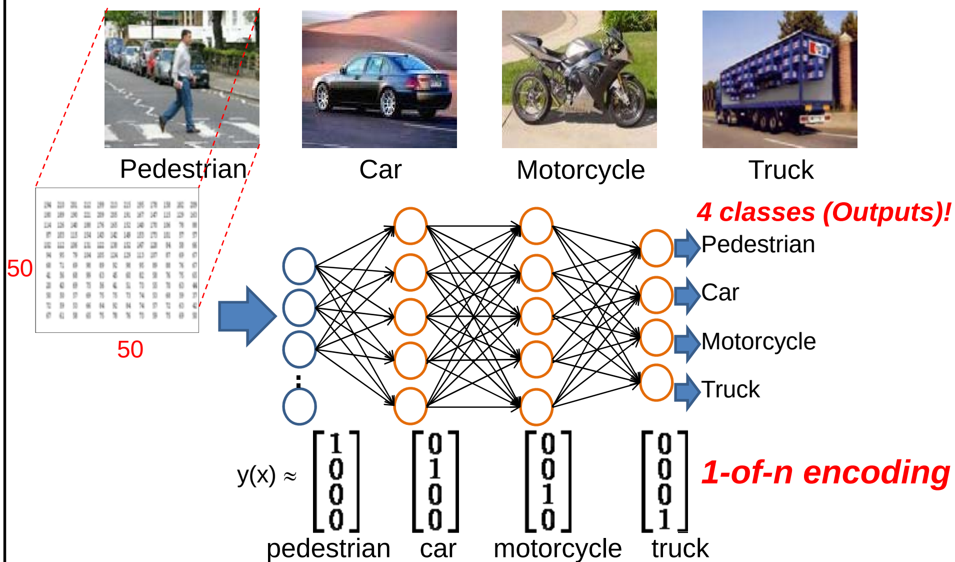
Multiple Classes (Multiple output units)



$$y(x) \approx \underset{\text{pedestrian}}{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}} \quad \underset{\text{car}}{\begin{bmatrix}0\\1\\0\\0\end{bmatrix}} \quad \underset{\text{motorcycle}}{\begin{bmatrix}0\\0\\1\\0\end{bmatrix}} \quad \underset{\text{truck}}{\begin{bmatrix}0\\0\\0\\1\end{bmatrix}}$$

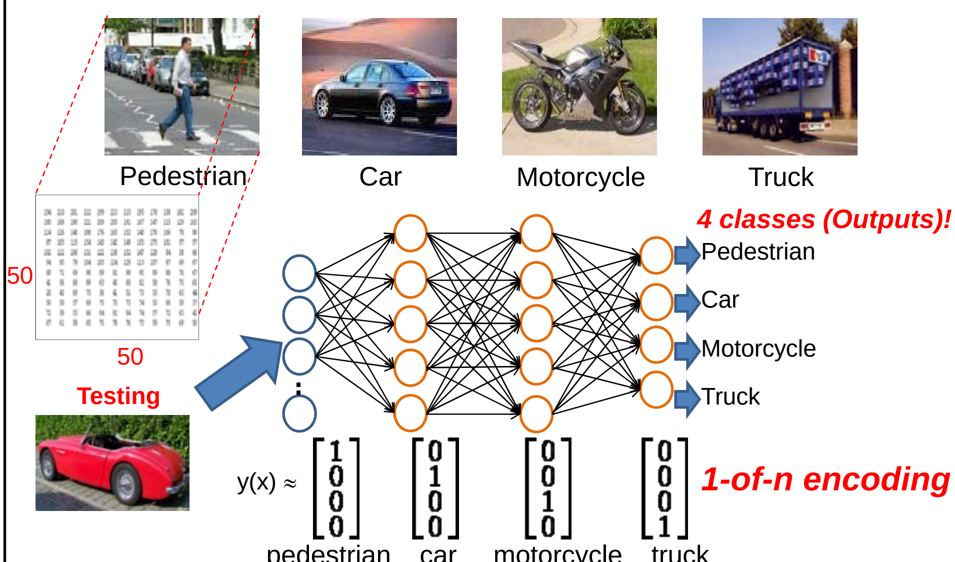
1-of-n encoding

Multiple Classes (Multiple output units)



Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

Multiple Classes (Multiple output units)



Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

Today's Outline

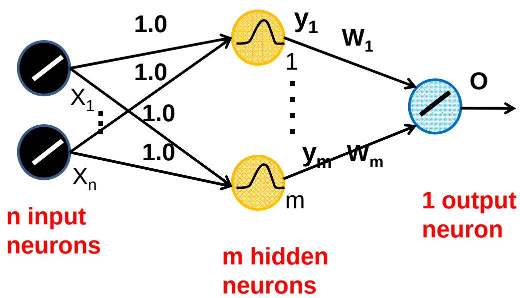
- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Today's Outline

- Multilayer Feedforward Neural Networks
  - Forward propagation
  - Backpropagation algorithm (supervised learning)
  - Implementation (Examples)
- Radial Basis Function Neural Networks

Radial Basis Function Neural Networks (RBF)

- Universal function approximator
- Two layers: 1 x hidden layer and 1x output layer
- Using a bell shaped *radial basis* transfer function as the activation function of each hidden neuron.
- Input and output neurons are linear neurons

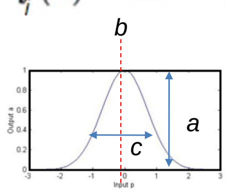


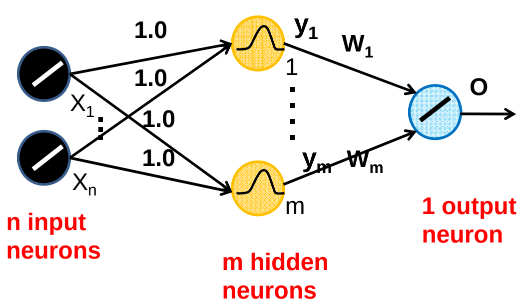
Radial Basis Function Neural Networks (RBF)

The activation function of each hidden neuron:

Gaussian function

$$y_i = f_i(\mathbf{x}) = ae^{-\frac{(\mathbf{x}-b_i)^2}{2c_i^2}} ;i=1,..,m$$





(x-b)^2 = the square of the distance between the input feature vector ***x*** and the center vector ***b*** for that radial basis function

c or variable sigma (σ) = the width or radius of the bell-shape and is something that has to be determined empirically

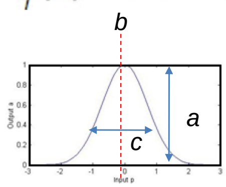
a = amplitude, normally it is set to 1.0

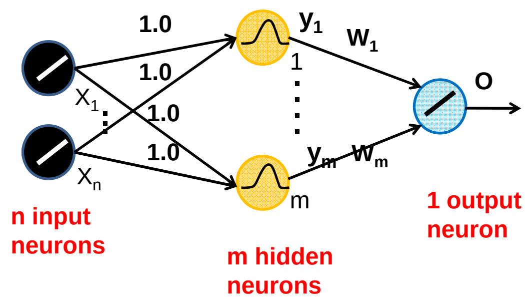
Radial Basis Function Neural Networks (RBF)

The activation function of each hidden neuron:

Gaussian function

$$y_i = f_i(\mathbf{x}) = a e^{-\frac{(\mathbf{x}-b_i)^2}{2c_i^2}};\ i=1,..,m$$





The activation function of output neuron:

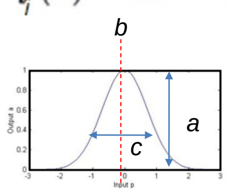
$$o = f(y) = \sum_{i=1}^{m} w_i \cdot y_i$$

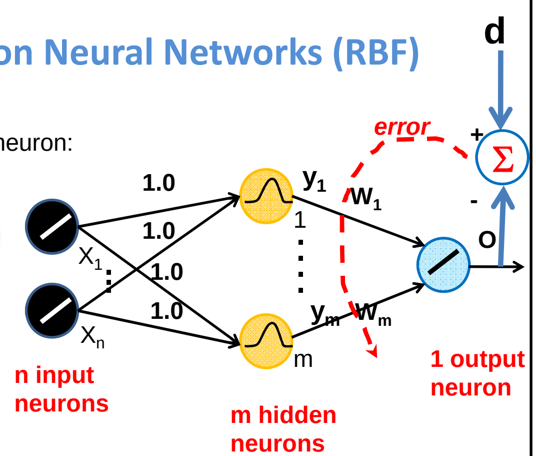
Radial Basis Function Neural Networks (RBF)

The activation function of each hidden neuron:

Gaussian function

$$y_i = f_i(\mathbf{x}) = a e^{-\frac{(\mathbf{x}-b_i)^2}{2c_i^2}};\ i=1,..,m$$





The activation function of output neuron:

$$o = f(y) = \sum_{i=1}^{m} w_i \cdot y_i$$

Delta learning rule:

$$\Delta w_i = \eta(d - o) y_i$$

Radial Basis Function Neural Networks (RBF)

Training the network:

Radial Basis Function Neural Networks (RBF)

Training the network:

The training is performed by deciding on

– How many **hidden nodes** there should be.

Radial Basis Function Neural Networks (RBF)

Training the network:

The training is performed by deciding on

– How many **hidden nodes** there should be.

– The **centers *(b)* and the widths *(c)*** of the Gaussians where the input data set is used to determine the parameters.

Radial Basis Function Neural Networks (RBF)

Training the network:

The training is performed by deciding on

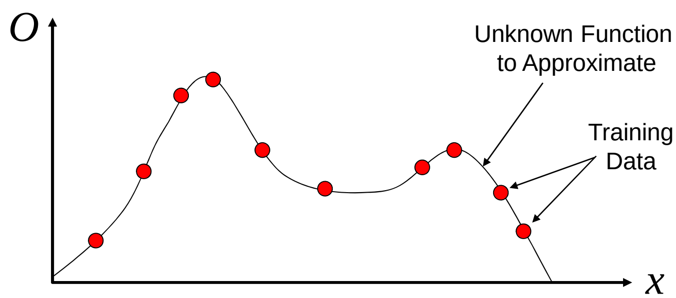
– How many **hidden nodes** there should be

– The **centers *(b)* and the widths *(c)*** of the Gaussians where the input data set is used to determine the parameters

– Functions are kept fixed while **the second layer weights are trained** (Simple delta learning rule).

Radial Basis Function Neural Networks (RBF)

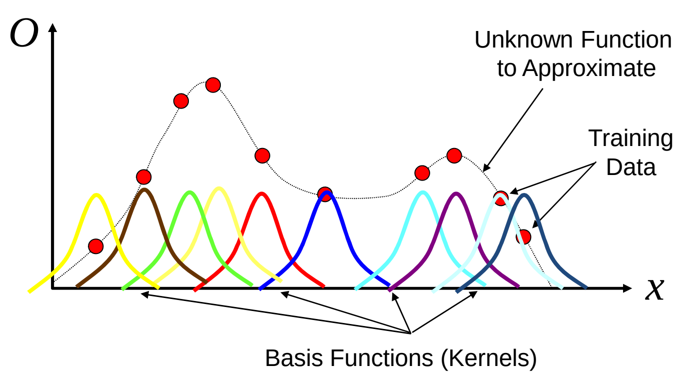
The idea



Function approximation

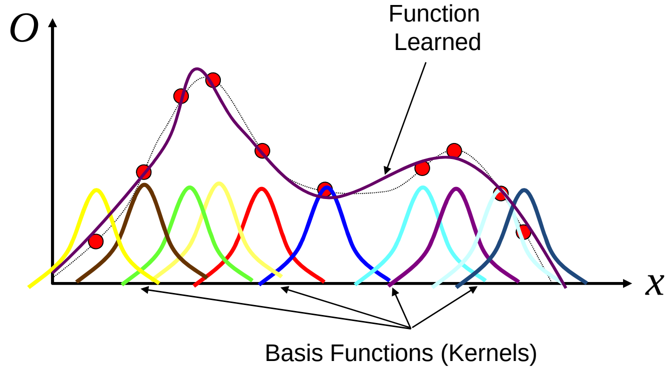
Radial Basis Function Neural Networks (RBF)

The idea



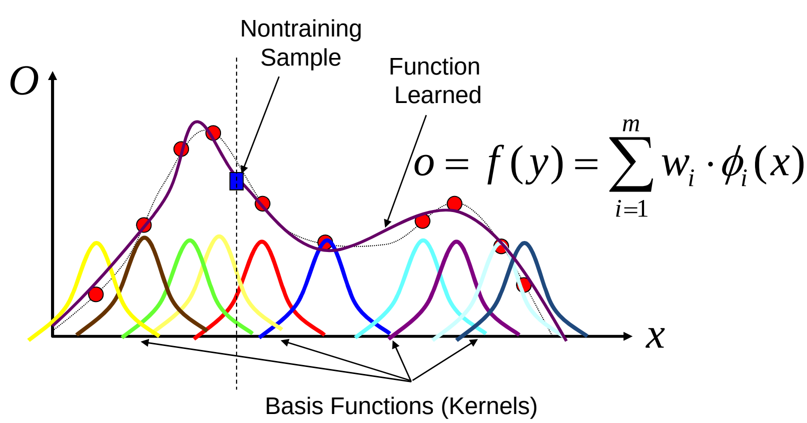
Radial Basis Function Neural Networks (RBF)

The idea



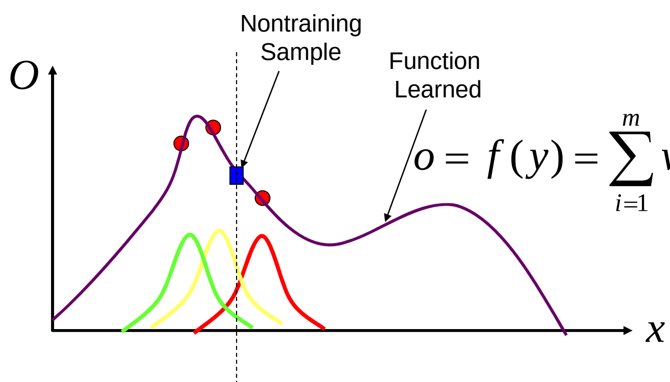
Radial Basis Function Neural Networks (RBF)

The idea



Radial Basis Function Neural Networks (RBF)

The idea



$$o = f(y) = \sum_{i=1}^{m} w_i \cdot \phi_i(x)$$

"Only inputs near a receptive field produce an activation"

ML Feedforward net vs RBFN

Global hyperplane	Local receptive field
EBP	Delta rule
Local minima	Serious local minima
Smaller number of hidden neurons	Larger number of hidden neurons
Longer learning time (all weights)	Shorter learning time (only output weights)

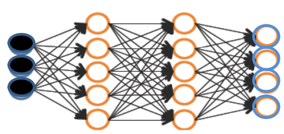
See also Xie, et al., 2011, Comparison between traditional neural networks and radial basis function networks, 2011

Summary

Summary

- **Multilayer Feedforward Neural Networks**

  - No connections with in a layer → sigmoid or tanh as activation function

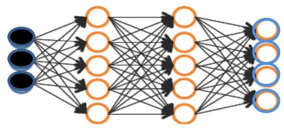


  - Forward propagation → Propagating “activities” from inputs to outputs!

Summary

- **Multilayer Feedforward Neural Networks**

  - No connections with in a layer → sigmoid or tanh as activation function



  - Forward propagation → Propagating “activities” from inputs to outputs!

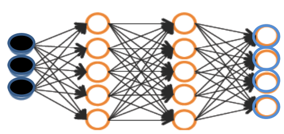
  - Backpropagation (supervised learning)→ Propagating the errors backward through the network for weight adaptation

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j$$

Summary

- **Multilayer Feedforward Neural Networks**

  - No connections with in a layer → sigmoid or tanh as activation function



  - Forward propagation → Propagating “activities” from inputs to outputs!

  - Backpropagation (supervised learning)→ Propagating the errors backward through the network for weight adaptation

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j$$

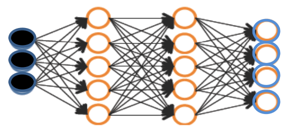
  - Adding momentum to speed up learning

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j + \alpha \Delta w_{ij}(n-1)$$

Summary

- **Multilayer Feedforward Neural Networks**

  - No connections with in a layer → sigmoid or tanh as activation function



  - Forward propagation → Propagating "activities" from inputs to outputs!

  - Backpropagation (supervised learning)→ Propagating the errors backward through the network for weight adaptation

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j$$

  - Adding momentum to speed up learning

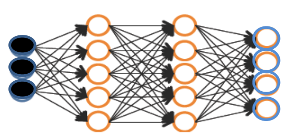
$$\Delta w_{ij}(n) = \mu * \delta_i * y_j + \alpha \Delta w_{ij}(n-1)$$

  - Implementation, Representing input/output data (mapping), Experiments (Training, Testing, Validation)

Summary

- **Multilayer Feedforward Neural Networks**

  - No connections with in a layer → sigmoid or tanh as activation function



  - Forward propagation → Propagating "activities" from inputs to outputs!

  - Backpropagation (supervised learning)→ Propagating the errors backward through the network for weight adaptation

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j$$

  - Adding momentum to speed up learning

$$\Delta w_{ij}(n) = \mu * \delta_i * y_j + \alpha \Delta w_{ij}(n-1)$$

  - Implementation, Representing input/output data (mapping), Experiments (Training, Testing, Validation)

- **Radial Basis Function Neural Networks** → using Gaussian function with delta learning rule! (learning only output weights)

Task III of the course

- Task: Implement a feedforward network for XOR function (2 input neurons, 2 hidden neurons, 1 output neuron)
- Try to test with and without momentum term!
- Try to plot Weigh and Error values to see how the system work!

Input1	Input2	Output
+1.0	+1.0	−1.0
+1.0	−1.0	+1.0
−1.0	+1.0	+1.0
−1.0	−1.0	−1.0

Reading Materials of Today!

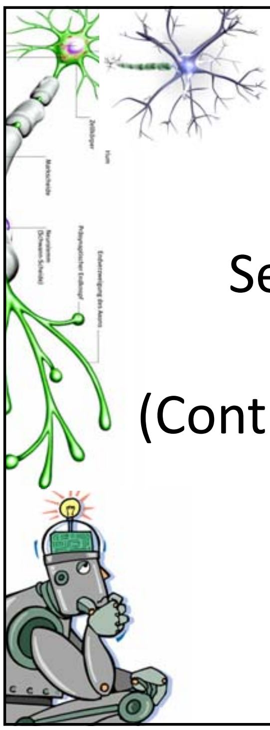
http://manoonpong.com/AI2Lecture:
In the folder: /week3/ReadingMaterialsCH2

Quickprop:
Scott E. Fahlman: An Empirical Study of Learning Speed in Back-Propagation Networks, September 1988

ANN for RunBot Locomotion Control:
Schroeder-Schetelig, J.; Manoonpong, P. and Woergoetter, F. (2010) Using Efference Copy and a Forward Internal Model for Adaptive Biped Walking. Autonomous Robots, DOI:10.1007/s10514-010-9199-7.

Software

FANN: http://leenissen.dk/fann/wp/download/



See you on March 7th!
With ANNs
(Continue!→ Recurrent neural networks)

Supplementary information

- **Derivation of the Backpropagation rule**
- **Structural plasticity methods**

Derivation of the Backpropagation rule

For each training example ***d*** every weight ***wji*** *is updated by adding to it* Δw_{ji}

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

where ***Ed*** is the error on training example ***d***, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit ***k*** for training example ***d***, and o_k is the output of unit ***k*** given training example ***d.***

Derivation of the Backpropagation rule

For each training example ***d*** every weight ***wji*** *is updated by adding to it* Δw_{ji}

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

where ***Ed*** is the error on training example ***d***, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit ***k*** for training example ***d***, and o_k is the output of unit ***k*** given training example ***d.***

Chain rule:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \quad \leftarrow \text{Activation}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji}$$

$net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)

Derivation of the Backpropagation rule

For each training example ***d*** every weight ***wji*** *is updated by adding to it* Δw_{ji}

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

where ***Ed*** is the error on training example ***d***, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit ***k*** for training example ***d***, and o_k is the output of unit ***k*** given training example ***d.***

Chain rule:

2 Cases:
"j" = an output unit
"j" = a hidden unit

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \quad \leftarrow \text{Activation}$$

$$= \boxed{\frac{\partial E_d}{\partial net_j}} x_{ji}$$

$net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights
"j = output unit".

$$\frac{\partial E_d}{\partial net_j}$$

Chain rule: $$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$o_j =$ the output computed by unit j

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights
"j = output unit".

$$\frac{\partial E_d}{\partial net_j}$$

Chain rule: $$\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}}\frac{\partial o_j}{\partial net_j}$$

$o_j =$ the output computed by unit j

$t_j =$ the target output for unit j

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

Derivation of the Backpropagation rule

***Case 1: Training Rule for Output Unit Weights* $\frac{\partial E_d}{\partial net_j}$**
"j = output unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}} \frac{\partial o_j}{\partial net_j}$

$o_j =$ the output computed by unit j

$t_j =$ the target output for unit j

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

$$\begin{aligned} \frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j} \frac{1}{2} (t_j - o_j)^2 \\ &= \frac{1}{2} 2 (t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \end{aligned}$$

The derivatives will be zero for all output units ***k*** except when ***k = j***.
→ We therefore drop the summation over output units and simply set ***k = j***.

Derivation of the Backpropagation rule

***Case 1: Training Rule for Output Unit Weights* $\frac{\partial E_d}{\partial net_j}$**
"j = output unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}} \frac{\partial o_j}{\partial net_j}$

$o_j =$ the output computed by unit j

$t_j =$ the target output for unit j

$\boxed{-(t_j - o_j)}$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}} \boxed{\frac{\partial o_j}{\partial net_j}}$

$\boxed{-(t_j - o_j)}$

o_j = the output computed by unit j

t_j = the target output for unit j

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

In case of sigmoid transfer function

$o_j = \sigma(net_j)$

The derivative of the sigmoid function

$\sigma(net_j)(1 - \sigma(net_j))$.

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}} \boxed{\frac{\partial o_j}{\partial net_j}}$

$\boxed{-(t_j - o_j)}$

o_j = the output computed by unit j

t_j = the target output for unit j

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1 - o_j)$$

In case of sigmoid transfer function

$o_j = \sigma(net_j)$

The derivative of the sigmoid function

$\sigma(net_j)(1 - \sigma(net_j))$.

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \boxed{\frac{\partial E_d}{\partial o_j}} \boxed{\frac{\partial o_j}{\partial net_j}}$

$\boxed{-(t_j - o_j)}$

$o_j =$ the output computed by unit j

$t_j =$ the target output for unit j

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$
$$= o_j(1 - o_j)$$

In case of sigmoid transfer function

$o_j = \sigma(net_j)$

The derivative of the sigmoid function

$\sigma(net_j)(1 - \sigma(net_j)).$

Final: $\boxed{\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j)}$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Weight adaptation: $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Weight adaptation: $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Weight adaptation: $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$\boxed{\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta \ \underbrace{(t_j - o_j)\ o_j(1 - o_j)}\, x_{ji}}$$

$$\delta_k = -\frac{\partial E_d}{\partial net_k}$$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = output unit".

Weight adaptation: $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

F'(x) = sigmoid

$$\boxed{\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji}}$$

$$\delta_k = -\frac{\partial E_d}{\partial net_k}$$

This term will be changed according to used function

Derivation of the Backpropagation rule

For each training example ***d*** every weight ***wji*** *is updated by adding to it* Δw_{ji}

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

where ***Ed*** is the error on training example ***d***, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit ***k*** for training example ***d***, and o_k is the output of unit ***k*** given training example ***d.***

Chain rule:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

Activation

2 Cases:
"j" = an output unit
"j" = a hidden unit

$$= \boxed{\frac{\partial E_d}{\partial net_j}} x_{ji}$$

$net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

Downstream(j) = the set of units whose immediate inputs include the output of unit **j**

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule:
$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$
$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$
$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$
$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j}$$

Downstream(j) = the set of units whose immediate inputs include the output of unit **j**

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \boxed{\frac{\partial o_j}{\partial net_j}} \rightarrow o_j(1 - o_j)$$

Downstream(j) = the set of units whose immediate inputs include the output of unit **j**

In case of sigmoid transfer function

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \boxed{\frac{\partial E_d}{\partial net_j}} \frac{\partial net_j}{\partial w_{ji}}$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \boxed{\frac{\partial E_d}{\partial net_j}} \boxed{\frac{\partial net_j}{\partial w_{ji}}} \rightarrow x_{ji}$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights "j = hidden unit". $\frac{\partial E_d}{\partial net_j}$

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \boxed{\frac{\partial E_d}{\partial net_j}} \boxed{\frac{\partial net_j}{\partial w_{ji}}} \rightarrow x_{ji}$

$$\Delta w_{ji} = \eta \underbrace{o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k \; w_{kj}}_{\delta_j} \; x_{ji}$$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights "j = hidden unit". $\frac{\partial E_d}{\partial net_j}$

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \boxed{\frac{\partial E_d}{\partial net_j}} \boxed{\frac{\partial net_j}{\partial w_{ji}}} \rightarrow x_{ji}$

$$\Delta w_{ji} = \eta \underbrace{o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k \; w_{kj}}_{\delta_j} \; x_{ji}$$

$$\Delta w_{ji} = \eta \; \delta_j \; x_{ji}$$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights $\frac{\partial E_d}{\partial net_j}$
"j = hidden unit".

Chain rule: $\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1-o_j)$

Weight adaptation: $\Delta w_{ji} = -\eta \boxed{\frac{\partial E_d}{\partial w_{ji}}} \Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \boxed{\frac{\partial E_d}{\partial net_j}} \boxed{\frac{\partial net_j}{\partial w_{ji}}} \rightarrow x_{ji}$

F'(x) = sigmoid

$$\Delta w_{ji} = \eta \, \underbrace{o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}}_{\delta_j} \; x_{ji}$$

This term will be changed according to used function

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Self adapting neural architectures (Structural plasticity)

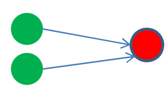
- **Growing method:** Small net and then add hidden neurons
- **Pruning method:** Large net. and then cut connections
- **Decomposition method:** Several net. and then each net corresponds to each task

Self adapting neural architectures (Structural plasticity)

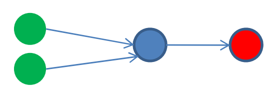
- **Growing method:** Start with networks that are too small to solve a problem and then add neurons and connections during training process.

  For example, Fahlman & Lebiere (1990) proposed a mechanism (*cascade correlation learning algorithm*) where the initial structure includes only direct connections from input to the output units. During learning (Back-prop**), if the error does not decrease below a defined threshold after a certain number of training cycles, a new initial neuron connected** to all the input and output units as well as to all previously created hidden neurons **is added**.

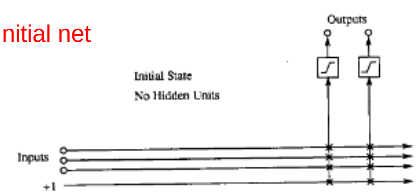
Initial net.



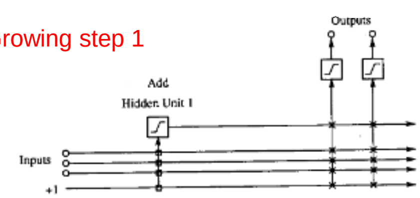
Growing net.



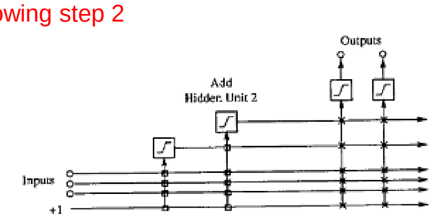
Initial net



Growing step 1



Growing step 2



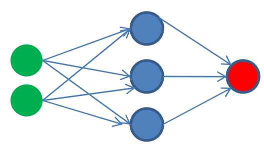
Fahlman, S. E. and C. Lebiere (1990) "The Cascade-Correlation Learning Architecture" in Advances in Neural Information Processing Systems 2, D. S. Touretzky (ed.), Morgan-Kaufmann, Los Altos CA, 1990.

Self adapting neural architectures (Structural plasticity)

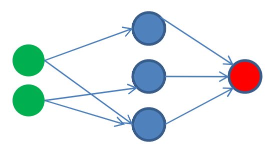
- **Pruning method:** Start with a large networks and then progressively reduce the network size by eliminating connections until the error becomes unacceptable.

For example, Rumelheart & Huberman (1991) proposed the *weight decay mechanism*. The mechanism tries to minimize the size of the connection weights in addition to the learning error. (Back-prop). That is **weights can be eliminated, if they get close to zero.**

Initial net.



Pruning net.



Rissanen [Ris89] and Cheeseman [Che90] formalized the old but vague intuition of Occam's razor as the information theoretic *minimum description length (MDL) criterion:* Given some data, the most probable model is the model that minimizes

$$\underbrace{\text{description length}}_{\text{cost}} = \underbrace{\text{description length(data|model)}}_{\text{error}} + \underbrace{\text{description length(model)}}_{\text{complexity}} .$$

Error

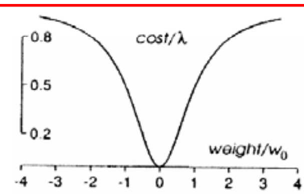
Network size

$$\textit{Cost} = \sum_{k \in T} (\text{target}_k - \text{output}_k)^2 + \lambda \sum_{i \in C} \frac{w_i^2 / w_0^2}{1 + w_i^2 / w_0^2}$$

Weights change by the gradient of the cost function

Wi = wieghts, W0 = scaling factor

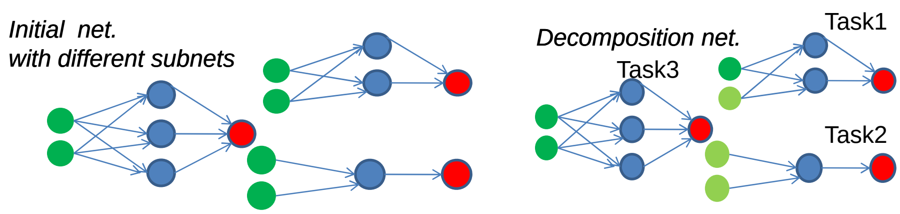
- Start with λ = 0, then network size is ignored first
- Then λ → is gradually increased while learning is progressed!
- But it can be also decreased depending only on the Error!

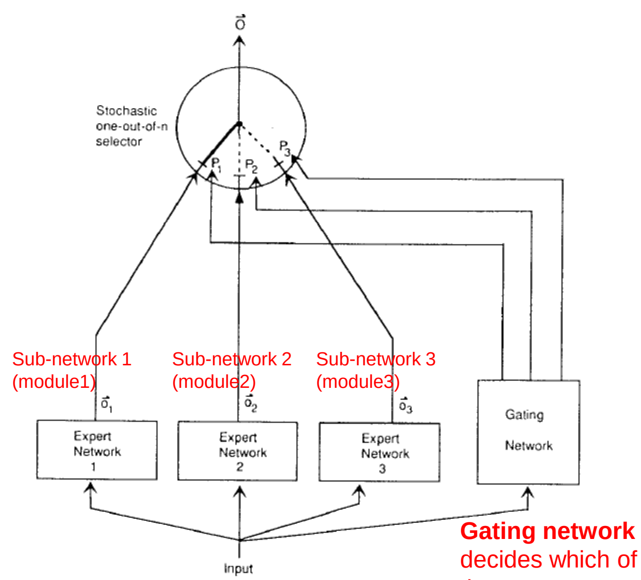


Andreas S. Weigend, David E. Rumelhart, Bernardo A. Huberman: Generalization by Weight-Elimination with Application to Forecasting. NIPS 1990: 875-882 (1990)

Self adapting neural architectures (Structural plasticity)

- **Decomposition method:** Start with a **network composed of a certain number of hand designed sub-networks** that have the same input and output units, but they might differ in their internal structure. They may compete to learn the training patterns or to control different subsets of the output units. As a consequence, at the end of the learning process, **different sub-networks may be responsible for different sets of patterns or for producing different parts of the output; thereby computing different functions** (Jacobs&Jordan (1991)).





Gating network decides which of the expert nets should be used for each training case (predefined!)

Jacobs, R. A.; Jordan, M. I.; Nowlan, S. J.; Hinton, G. E. (1991). "Adaptive Mixtures of Local Experts". *Neural Computation* **3**: 79. doi:10.1162/neco.1991.3.1.79

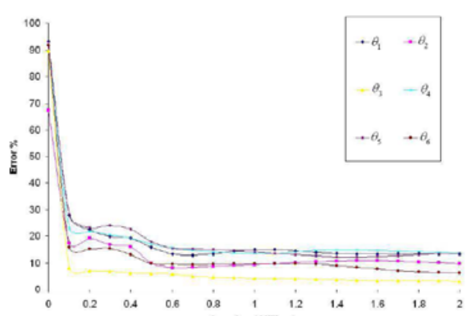
6 DOFs Manipulator

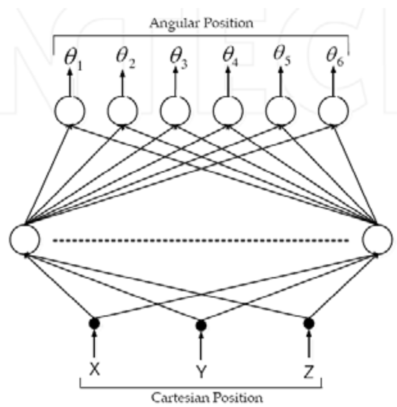
Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities

6 DOFs Manipulator

- 43 Hidden neurons
- 600 data (400 training, 200 testing)
- After 1.5 million steps

Setup 1



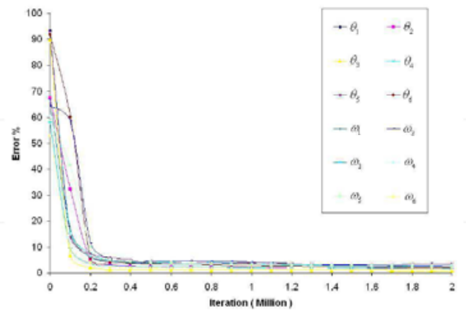


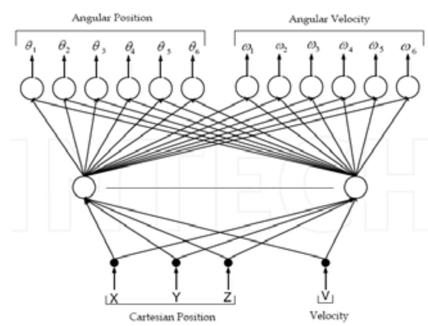
Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities

6 DOFs Manipulator

- 77 hidden neurons

Setup 2



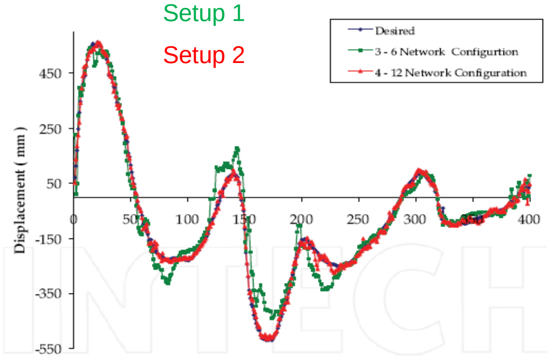


Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities

6 DOFs Manipulator

Setup 1

Setup 2



X axis

Fig. 11. Trajectory tracking for both configurations compared to each other after the training was finished for the X coordinate

Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities

6 DOFs Manipulator

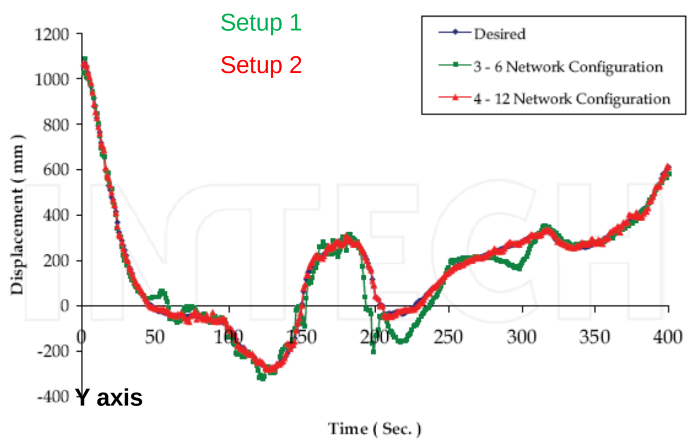


Fig. 12. Trajectory tracking for both configurations compared to each other after the training was finished for the Y coordinate

Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities

6 DOFs Manipulator

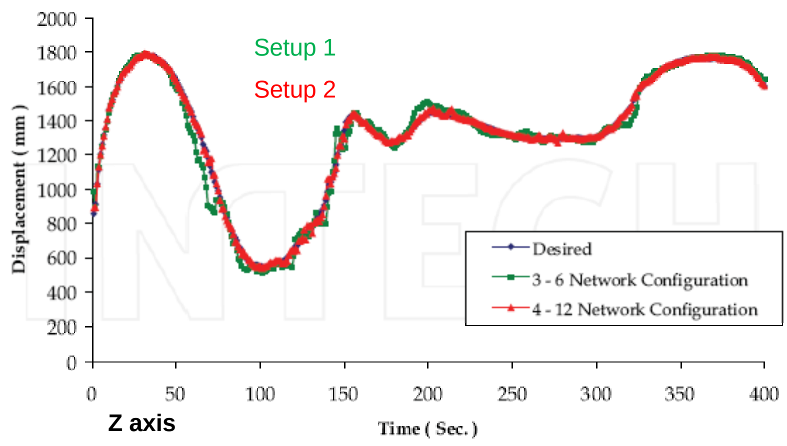


Fig. 12. Trajectory tracking for both configurations compared to each other after the training was finished for the Z coordinate

Ali T. Hasan, Hayder M.A.A. Al-Assadi and Ahmad Azlan Mat Isa, Neural Networks' Based Inverse Kinematics Solution for Serial Robot Manipulators Passing Through Singularities