Multi Layer Networks

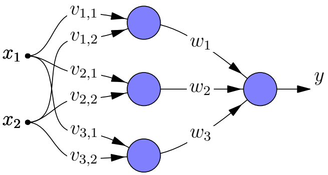
1. What can be Computed?
   - Convex Areas
   - Arbitrary Areas
2. Learning
   - Learning Methods
   - Generalized Delta Rule
   - Error Back-Propagation
   - Problems
3. Usage
   - System Identification
   - Data Compression

Multi Layer Feed-Forward Networks

What can a thresholded two layer network compute?

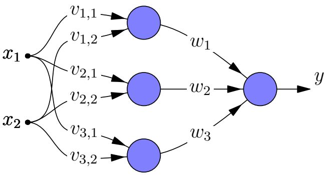
Multi Layer Feed-Forward Networks

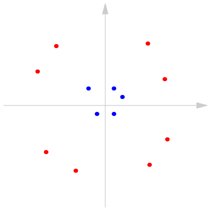
What can a thresholded two layer network compute?



Multi Layer Feed-Forward Networks

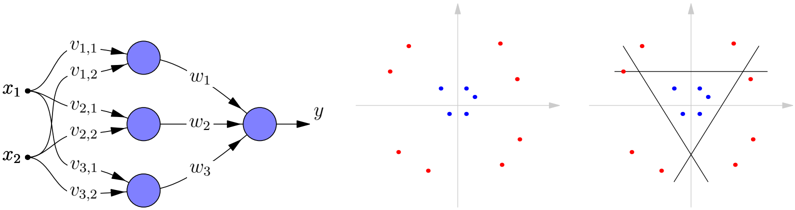
What can a thresholded two layer network compute?





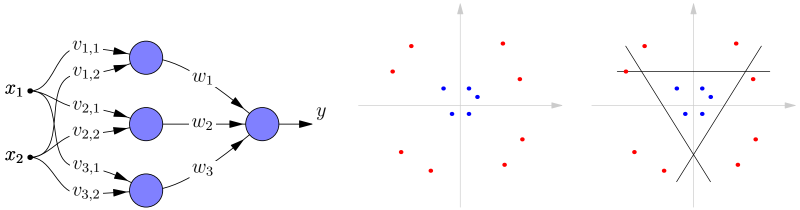
Multi Layer Feed-Forward Networks

What can a thresholded two layer network compute?



Multi Layer Feed-Forward Networks

What can a thresholded two layer network compute?



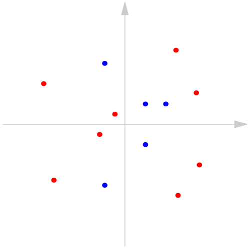
With $w_1 = w_2 = w_3 = 1$ and $\theta = 2.5$ the second layer operates as an AND-gate.

Multi Layer Feed-Forward Networks

What happens if the area is not convex?

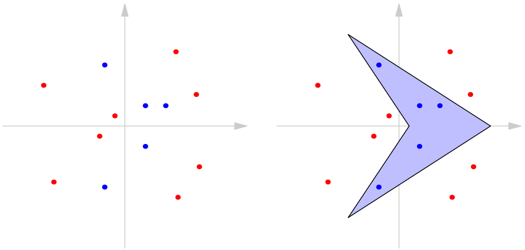
Multi Layer Feed-Forward Networks

What happens if the area is not convex?



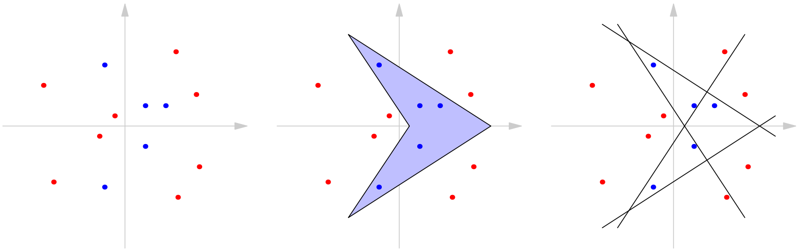
Multi Layer Feed-Forward Networks

What happens if the area is not convex?



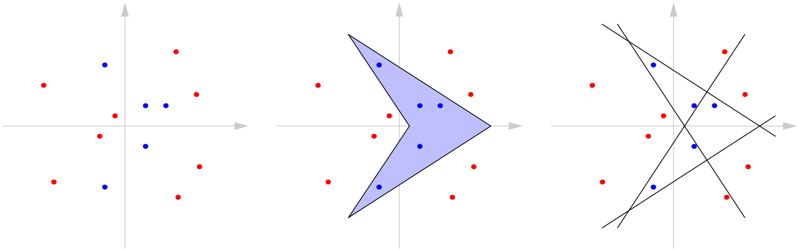
Multi Layer Feed-Forward Networks

What happens if the area is not convex?



Multi Layer Feed-Forward Networks

What happens if the area is not convex?



Arbitrarily complex areas can be extracted
provided there are enough hidden units

Learning

How can we train a multi layer network?

- Perceptron Learning
- Delta Rule

Learning

How can we train a multi layer network?

- Perceptron Learning
  Requires that we know in which direction the weights should be changed to come nearer the solution.

- Delta Rule

Learning

How can we train a multi layer network?

- Perceptron Learning
  Requires that we know in which direction the weights should be changed to come nearer the solution.
  Does not work!
- Delta Rule

Learning

How can we train a multi layer network?

- Perceptron Learning
  Requires that we know in which direction the weights should be changed to come nearer the solution.
  Does not work!
- Delta Rule
  Requires that we can measure the error before thresholding, but this only works for the last layer.

Learning

How can we train a multi layer network?

- Perceptron Learning
  Requires that we know in which direction the weights should be changed to come nearer the solution.
  Does not work!
- Delta Rule
  Requires that we can measure the error before thresholding, but this only works for the last layer.
  Does not work!

Learning

Dilemma:

- Thresholding destroys information needed for learning
- Without thresholding we loose the advantage of multiple layers

Learning

Dilemma:

- Thresholding destroys information needed for learning
- Without thresholding we loose the advantage of multiple layers

Solution

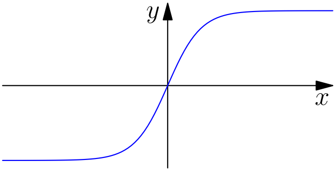
Use threshold-like but differentiable transfer functions

Learning

Two commonly used transfer functions $\varphi(\sum)$

Learning

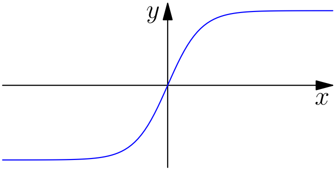
Two commonly used transfer functions $\varphi(\sum)$

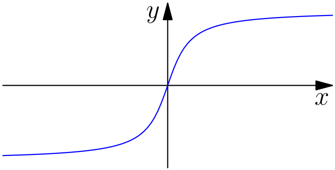


$$\varphi(x) = \frac{1 - e^{-x}}{1 + e^{-x}}$$

Learning

Two commonly used transfer functions $\varphi(\sum)$





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

$$\varphi(x) = \arctan(x)$$

Generalization of the Delta Rule:

Generalization of the Delta Rule:

1. Choose a cost function ε

2. Minimize it using Steepest Decent

Generalization of the Delta Rule:

1. Choose a cost function ε

$$\varepsilon = \frac{1}{2}||\vec{t} - \vec{y}||^2 = \frac{1}{2}\sum_k (t_k - y_k)^2$$

2. Minimize it using Steepest Decent

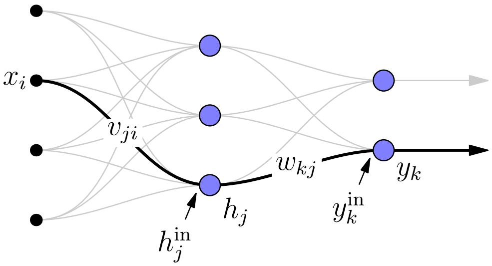
Generalization of the Delta Rule:

1. Choose a cost function ε

$$\varepsilon = \frac{1}{2}||\vec{t} - \vec{y}||^2 = \frac{1}{2}\sum_k (t_k - y_k)^2$$

2. Minimize it using Steepest Decent
Compute the gradient, i.e.

$$\frac{\partial \varepsilon}{\partial v_{ji}} \quad \text{and} \quad \frac{\partial \varepsilon}{\partial w_{kj}}$$


x_i
v_{ji}
h_j
h_j^{in}
w_{kj}
y_k
y_k^{in}

First case: derivative w.r.t. a weight w_{kj} in the second layer

$$
\begin{aligned}
\frac{\partial \varepsilon}{\partial w_{kj}} &= \frac{\partial \varepsilon}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_{kj}} \\
&= -(t_k - y_k) \cdot \frac{\partial \varphi(y_k^{\mathrm{in}})}{\partial w_{kj}} \\
&= -(t_k - y_k) \cdot \varphi'(y_k^{\mathrm{in}}) \cdot \frac{\partial y_k^{\mathrm{in}}}{\partial w_{kj}} \\
&= -(t_k - y_k) \cdot \varphi'(y_k^{\mathrm{in}}) \cdot h_j \\
&= -\delta_k h_j
\end{aligned}
$$

Here we have introduced $\delta_k = (t_k - y_k) \cdot \varphi'(y_k^{\mathrm{in}})$

Second case: derivative w.r.t. a weight v_{kj} in the first layer

$$
\begin{aligned}
\frac{\partial \varepsilon}{\partial v_{ji}} &= \sum_k \frac{\partial \varepsilon}{\partial y_k} \cdot \frac{\partial y_k}{\partial v_{ji}} \\
&= -\sum_k (t_k - y_k) \cdot \frac{\partial y_k}{\partial v_{ji}} \\
&= -\sum_k (t_k - y_k) \cdot \varphi'(y_k^{\text{in}}) \cdot \frac{\partial y_k^{\text{in}}}{\partial v_{ji}} \\
&= -\sum_k \delta_k \cdot \frac{\partial y_k^{\text{in}}}{\partial v_{ji}} \\
&= -\sum_k \delta_k \cdot w_{kj} \cdot \frac{\partial h_j}{\partial v_{ji}}
\end{aligned}
$$

We continue...

$$
\begin{aligned}
\frac{\partial \varepsilon}{\partial v_{ji}} &= -\sum_k \delta_k \cdot w_{kj} \cdot \frac{\partial h_j}{\partial v_{ji}} \\
&= -\sum_k \delta_k \cdot w_{kj} \cdot \varphi'(h_j^{\mathrm{in}}) \cdot \frac{\partial h_j^{\mathrm{in}}}{\partial v_{ji}} \\
&= -\sum_k \delta_k \cdot w_{kj} \cdot \varphi'(h_j^{\mathrm{in}}) \cdot x_i \\
&= -\delta_j x_i
\end{aligned}
$$

Here we have introduced $\delta_j = \sum_k \delta_k \cdot w_{kj} \cdot \varphi'(h_j^{\mathrm{in}})$

Summary

$$\frac{\partial \varepsilon}{\partial w_{kj}} = -\delta_k h_j \qquad \text{where } \delta_k = (t_k - y_k) \cdot \varphi'(y_k^{\text{in}})$$

$$\frac{\partial \varepsilon}{\partial v_{ji}} = -\delta_j x_i \qquad \text{where } \delta_j = \sum_k \delta_k \cdot w_{kj} \cdot \varphi'(h_j^{\text{in}})$$

Gradient Decent

$$\Delta w_{kj} = \eta \delta_k h_j$$

$$\Delta v_{ji} = \eta \delta_j x_i$$

Error Back-Propagation

1. Forward Pass: Compute all h_j and y_k

$$h_j = \varphi\Big(\sum_i v_{ji} x_i\Big) \qquad y_k = \varphi\Big(\sum_j w_{kj} h_j\Big)$$

2. Backward Pass: Compute all δ_k and δ_j

$$\delta_k = (t_k - y_k) \cdot \varphi'(y_k^{\mathrm{in}}) \qquad \delta_j = \sum_k \delta_k \cdot w_{kj} \cdot \varphi'(h_j^{\mathrm{in}})$$

3. Weight Updating:

$$\Delta w_{kj} = \eta \delta_k h_j \qquad \Delta v_{ji} = \eta \delta_j x_i$$

Problems with BackProp

Problems with BackProp

- Does not always converge (gets stuck in local minima)

Problems with BackProp

- Does not always converge (gets stuck in local minima)
- Slow convergence

Problems with BackProp

- Does not always converge (gets stuck in local minima)
- Slow convergence
- Many parameters need to be tuned

Problems with BackProp

- Does not always converge (gets stuck in local minima)
- Slow convergence
- Many parameters need to be tuned
- Bad scaling behavior for large problems

Problems with BackProp

- Does not always converge (gets stuck in local minima)
- Slow convergence
- Many parameters need to be tuned
- Bad scaling behavior for large problems
- Biologically unrealistic
  - Backward propagating signal
  - Requires known target values

Tips when using BackProp

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ
- Order or weigh the training examples so the hard examples dominate

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ
- Order or weigh the training examples so the hard examples dominate
- Choose smart initial weights

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ
- Order or weigh the training examples so the hard examples dominate
- Choose smart initial weights
- Introduce momentum in the weight updating

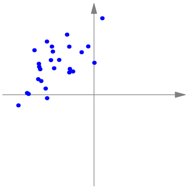
Tips when using BackProp

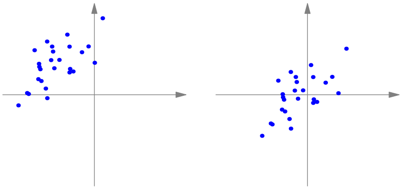
- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ
- Order or weigh the training examples so the hard examples dominate
- Choose smart initial weights
- Introduce momentum in the weight updating
- Add random noise to the weights during training

Tips when using BackProp

- Use an antisymmetric $\varphi(x)$
- Put the target values $\vec{t}$ inside the domain interval of φ
- Order or weigh the training examples so the hard examples dominate
- Choose smart initial weights
- Introduce momentum in the weight updating
- Add random noise to the weights during training
- Remove the squashing-function $(\varphi(x))$ for the output units

Preprocessing of input patterns



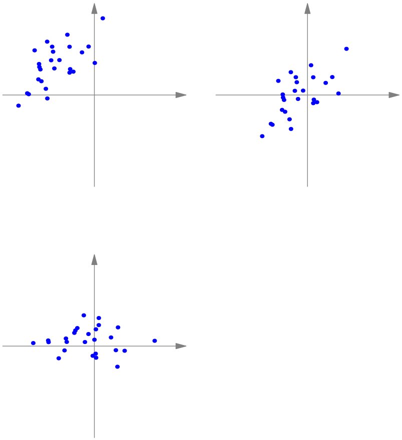


Preprocessing of input patterns

1. Subtract the average

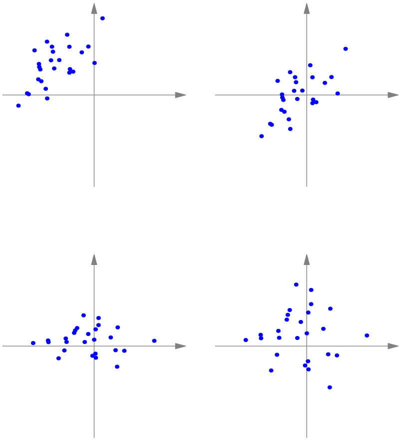
Preprocessing of input patterns

1. Subtract the average
2. Decorrelate



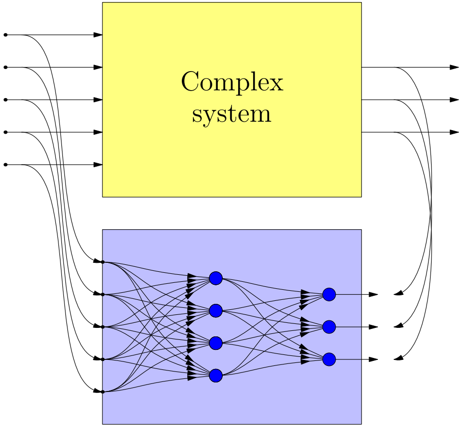
Preprocessing of input patterns

1. Subtract the average
2. Decorrelate
3. Normalize the variance

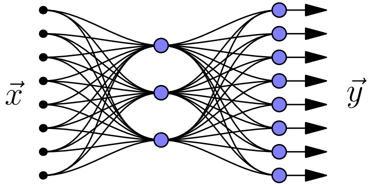


System Identification

"Mimic" an existing system



Data Compression



Hour-Glass form

Train with $\vec{x} = \vec{y}$ (auto-association)
Forces the network to use a compact encoding of the patterns.