Classification / Regression

Neural Networks 2

Neural networks

- Topics
  - Perceptrons
    - structure
    - training
    - expressiveness
  - Multilayer networks
    - possible structures
      - activation functions
    - training with gradient descent and backpropagation
    - expressiveness

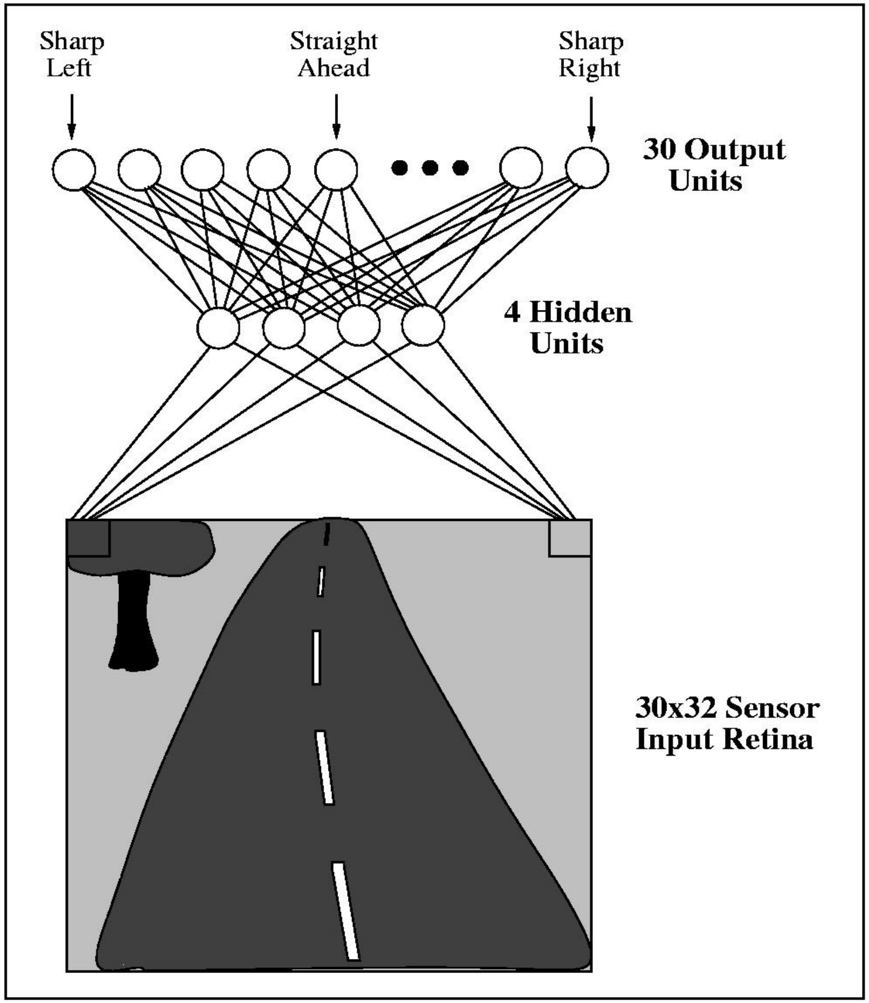
Neural network application

ALVINN: An Autonomous Land Vehicle In a Neural Network

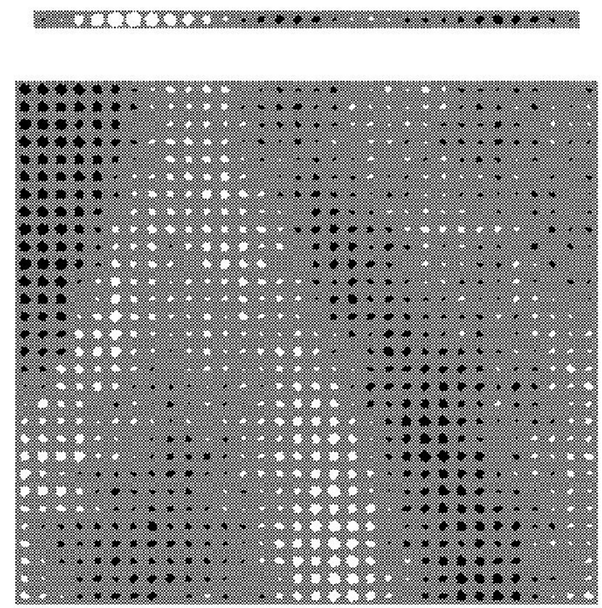
(Carnegie Mellon University Robotics Institute, 1989-1997)

ALVINN is a perception system which learns to control the NAVLAB vehicles by watching a person drive. ALVINN's architecture consists of a single hidden layer back-propagation network. The input layer of the network is a 30x32 unit two dimensional "retina" which receives input from the vehicles video camera. Each input unit is fully connected to a layer of five hidden units which are in turn fully connected to a layer of 30 output units. The output layer is a linear representation of the direction the vehicle should travel in order to keep the vehicle on the road.

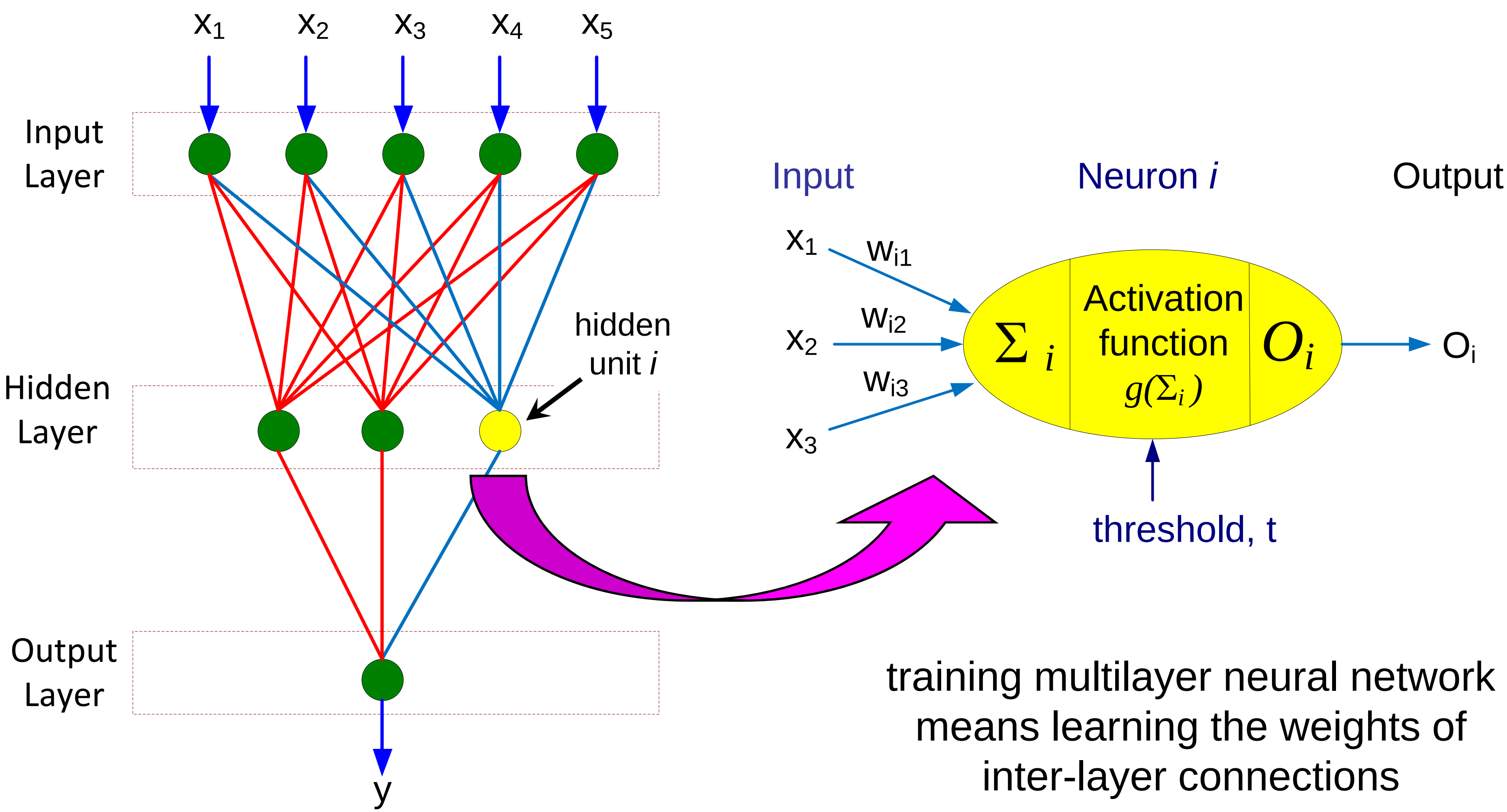
Neural network application



ALVINN drives 70 mph on highways!



General structure of multilayer neural network



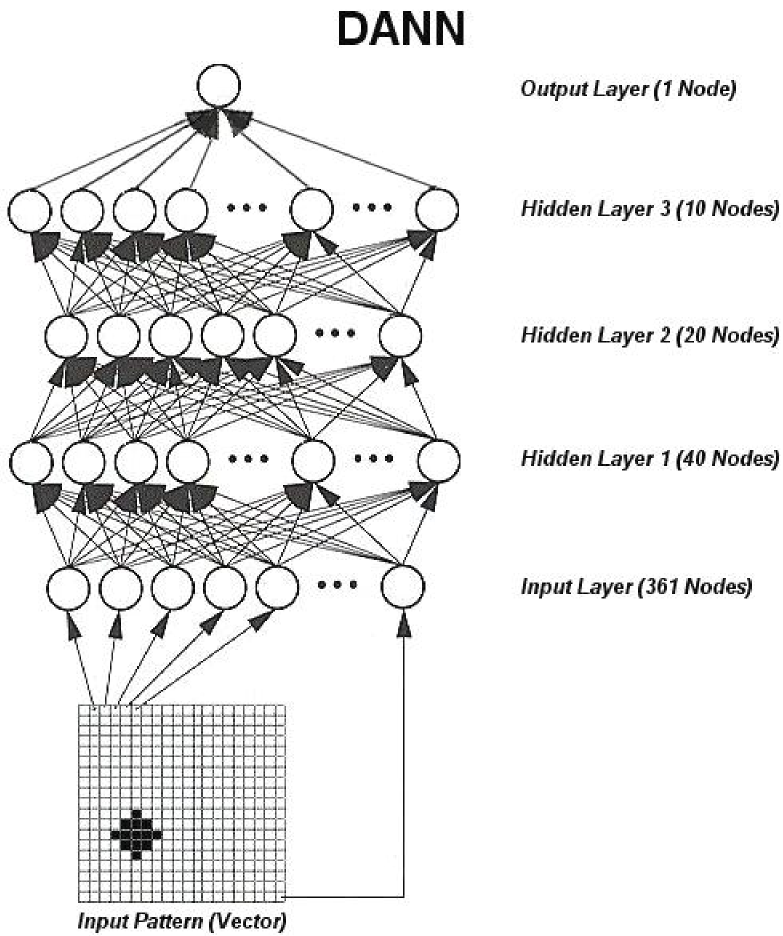
training multilayer neural network means learning the weights of inter-layer connections

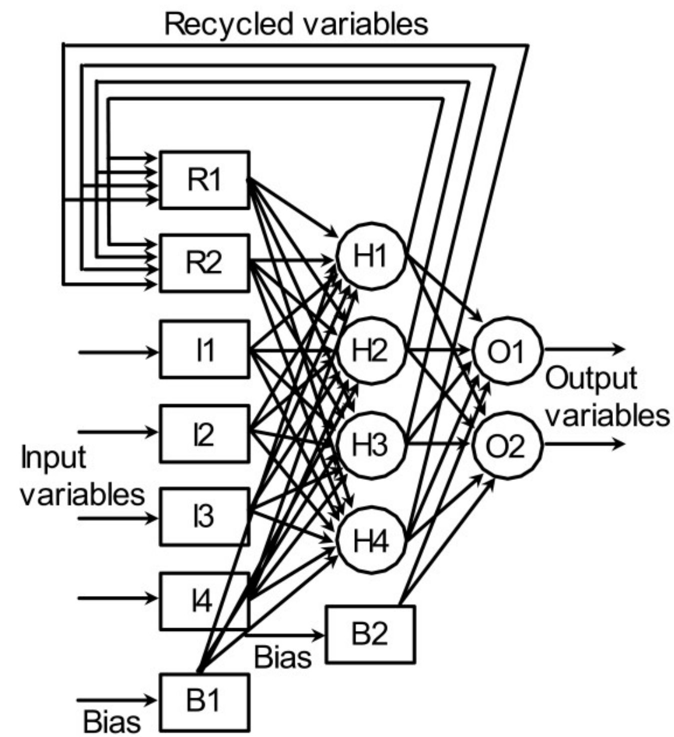
Neural network architectures

- All multilayer neural network architectures have:
  - At least one hidden layer
  - Feedforward connections from inputs to hidden layer(s) to outputs

  but more general architectures also allow for:
  - Multiple hidden layers
  - Recurrent connections
    - from a node to itself
    - between nodes in the same layer
    - between nodes in one layer and nodes in another layer above it

Neural network architectures



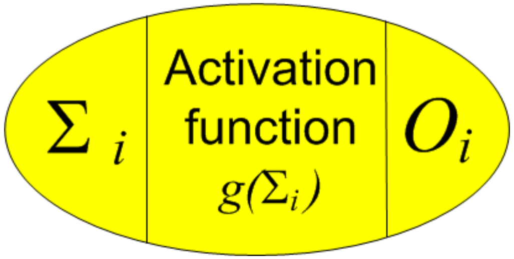


More than one hidden layer

Recurrent connections

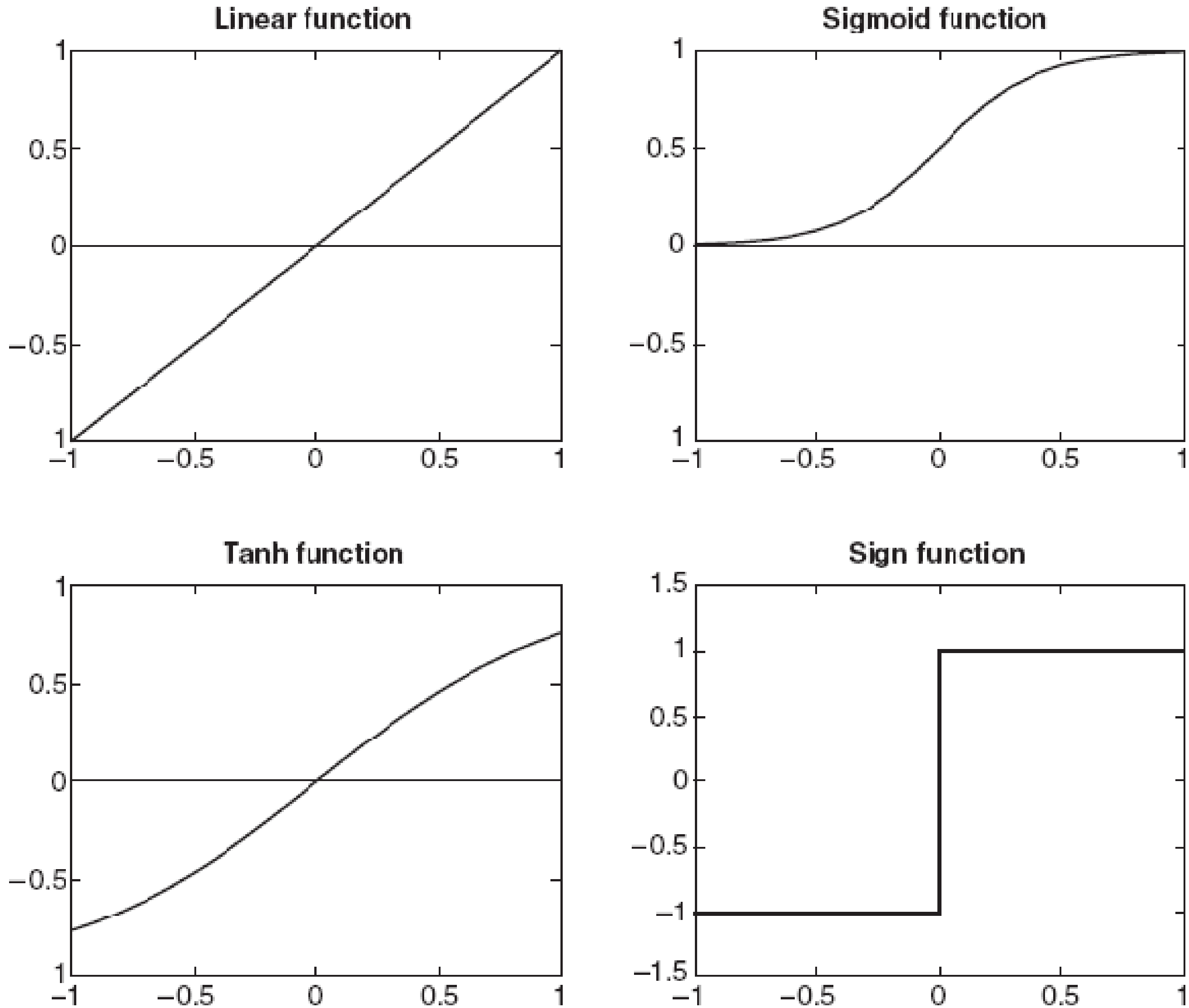
Neural networks: roles of nodes

- A node in the input layer:
  - distributes value of some component of input vector to the nodes in the first hidden layer, without modification
- A node in a hidden layer(s):
  - forms weighted sum of its inputs
  - transforms this sum according to some activation function (also known as transfer function)
  - distributes the transformed sum to the nodes in the next layer



- A node in the output layer:
  - forms weighted sum of its inputs
  - (optionally) transforms this sum according to some activation function

Neural network activation functions

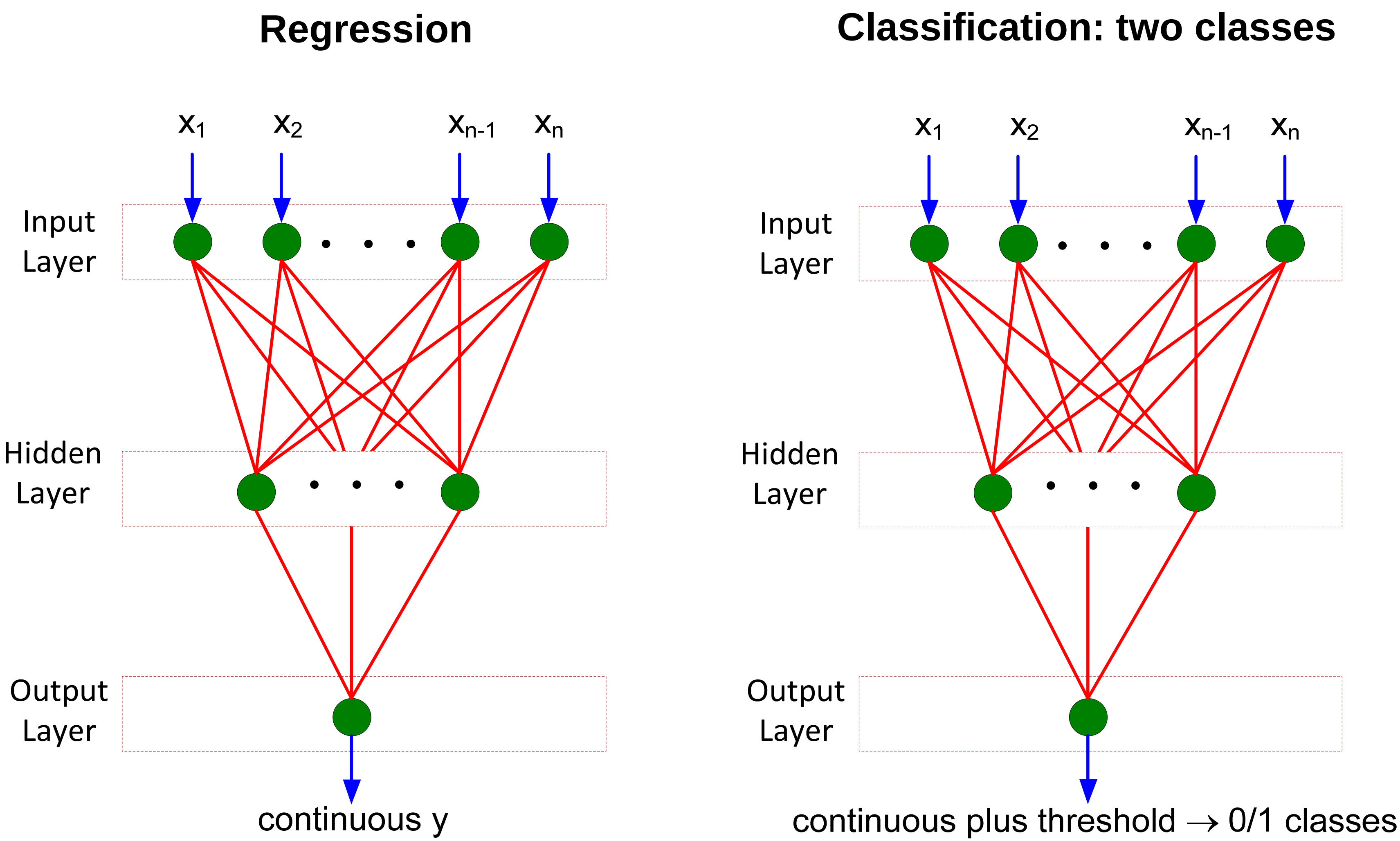


Neural network architectures

- The architecture most widely used in practice is fairly simple:
  - One hidden layer
  - No recurrent connections (feedforward only)
  - Non-linear activation function in hidden layer (usually sigmoid or tanh)
  - No activation function in output layer (summation only)

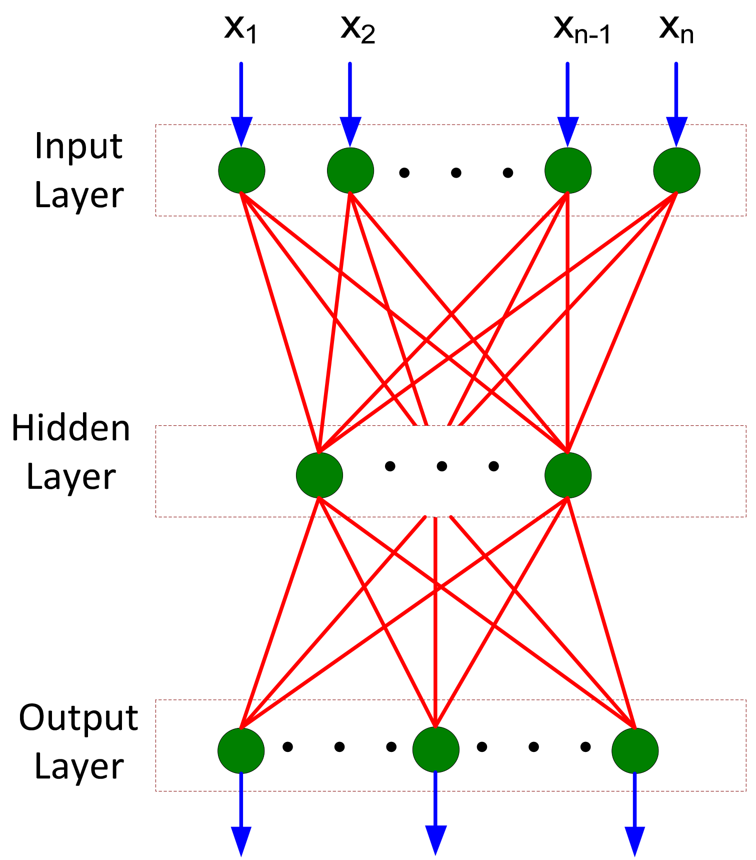
- This architecture can model *any* bounded continuous function.

Neural network architectures



Neural network architectures

Classification: multiple classes



continuous plus max → 1 of k classes

Classification: multiple classes

- When outcomes are one of *k* possible classes, they can be encoded using *k* dummy variables.
  - If an outcome is class *j*, then *j* th dummy variable = 1, all other dummy variables = 0.
- Example with four class labels:

$$y_i = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 2 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

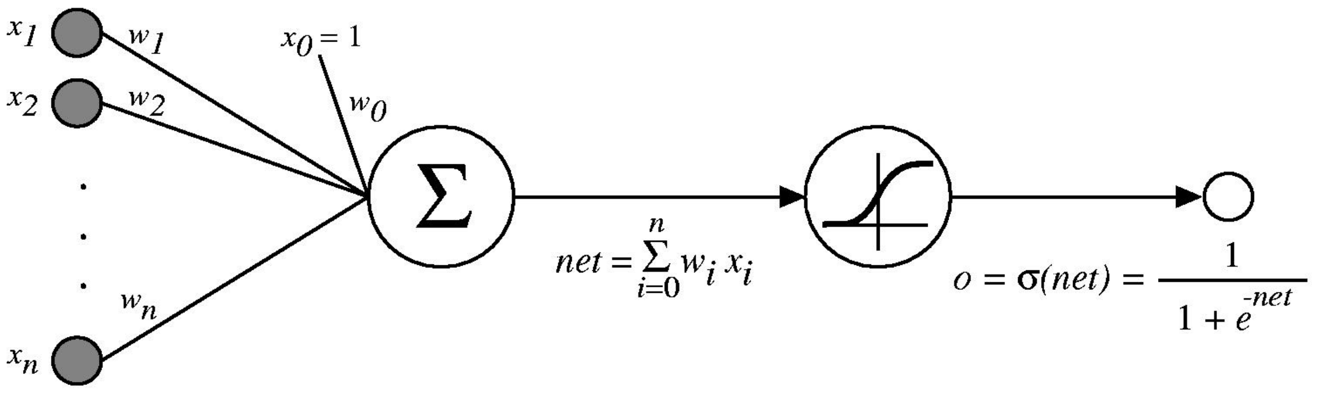
Algorithm for learning neural network

- Initialize the connection weights $\mathbf{w} = (w_0, w_1, \ldots, w_m)$
  - $\mathbf{w}$ includes all connections between all layers
  - Usually small random values
- Adjust weights such that output of neural network is consistent with class label / dependent variable of training samples
  - Typical loss function is squared error:

$$E(\mathbf{w}) = \sum_i \left[\mathbf{y}_i - \hat{\mathbf{y}}_i\right]^2 = \sum_i \left[\mathbf{y}_i - f(\mathbf{w}, \mathbf{x}_i)\right]^2$$

  - Find weights w_j that minimize above loss function

Sigmoid unit



$\sigma(x)$ is the sigmoid function

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

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

Sigmoid unit: training

- We can derive gradient descent rules to train:
  - A single sigmoid unit
  - Multilayer networks of sigmoid units
    - referred to as backpropagation

Backpropagation

Example: stochastic gradient descent, feedforward network with two layers of sigmoid units

Do until convergence

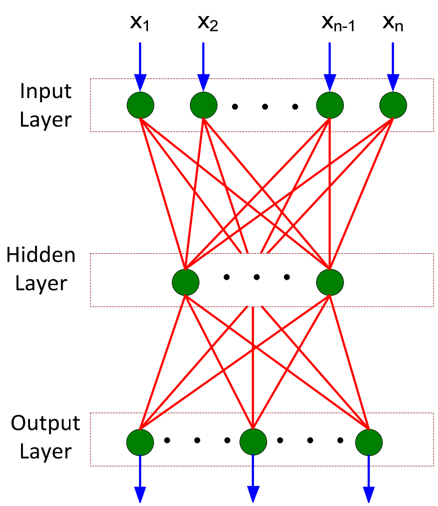
For each training sample $i = \langle \mathbf{x}_i, \mathbf{y}_i \rangle$

Propagate the input forward through the network

Calculate the output o_h of every hidden unit

Calculate the output o_k of every network output unit

Propagate the errors backward through the network



For each network output unit k, calculate its error term δ_k

$$\delta_k = o_k(1 - o_k)(\mathbf{y}_{ik} - o_k)$$

For each hidden unit h, calculate its error term δ_h

$$\delta_h = o_h(1 - o_h) \sum_k(w_{hk}\delta_k)$$

Update each network weight w_{ba}

$$w_{ba} = w_{ba} + \eta\delta_b z_{ba}$$

where z_{ba} is the a^{th} input to unit b

More on backpropagation

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

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

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

MATLAB interlude

matlab_demo_15.m

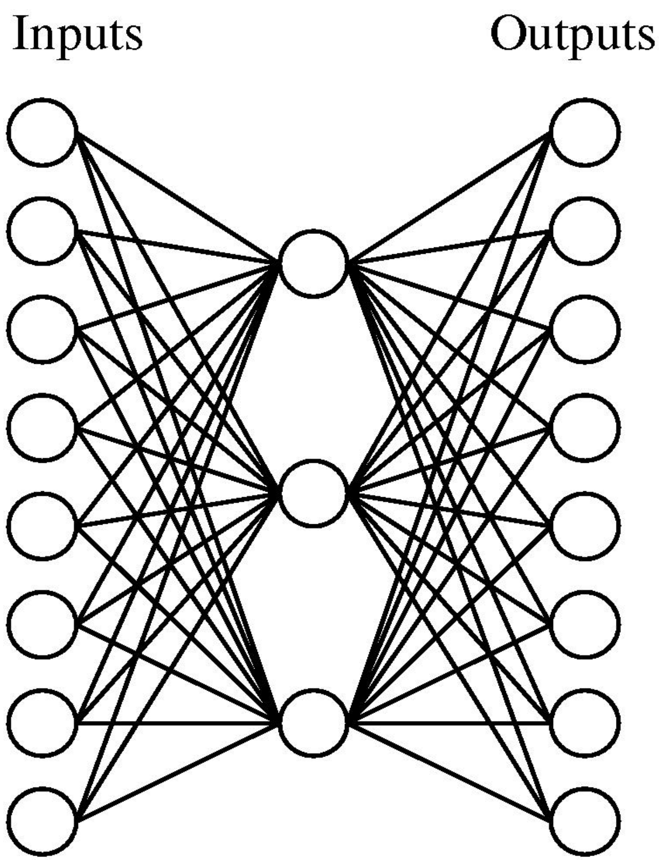
neural network classification of crab gender

200 samples

6 features

2 classes

Neural networks for data compression



- Goal: learn compressed representation of data
  - Number input nodes = number of output nodes
  - Number of hidden nodes < number of input/output nodes
- Train by applying each sample as both input and output
  - Otherwise like standard neural network
- Learned representation is weights of network

Neural networks for data compression

A target function:

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

Can this be learned?

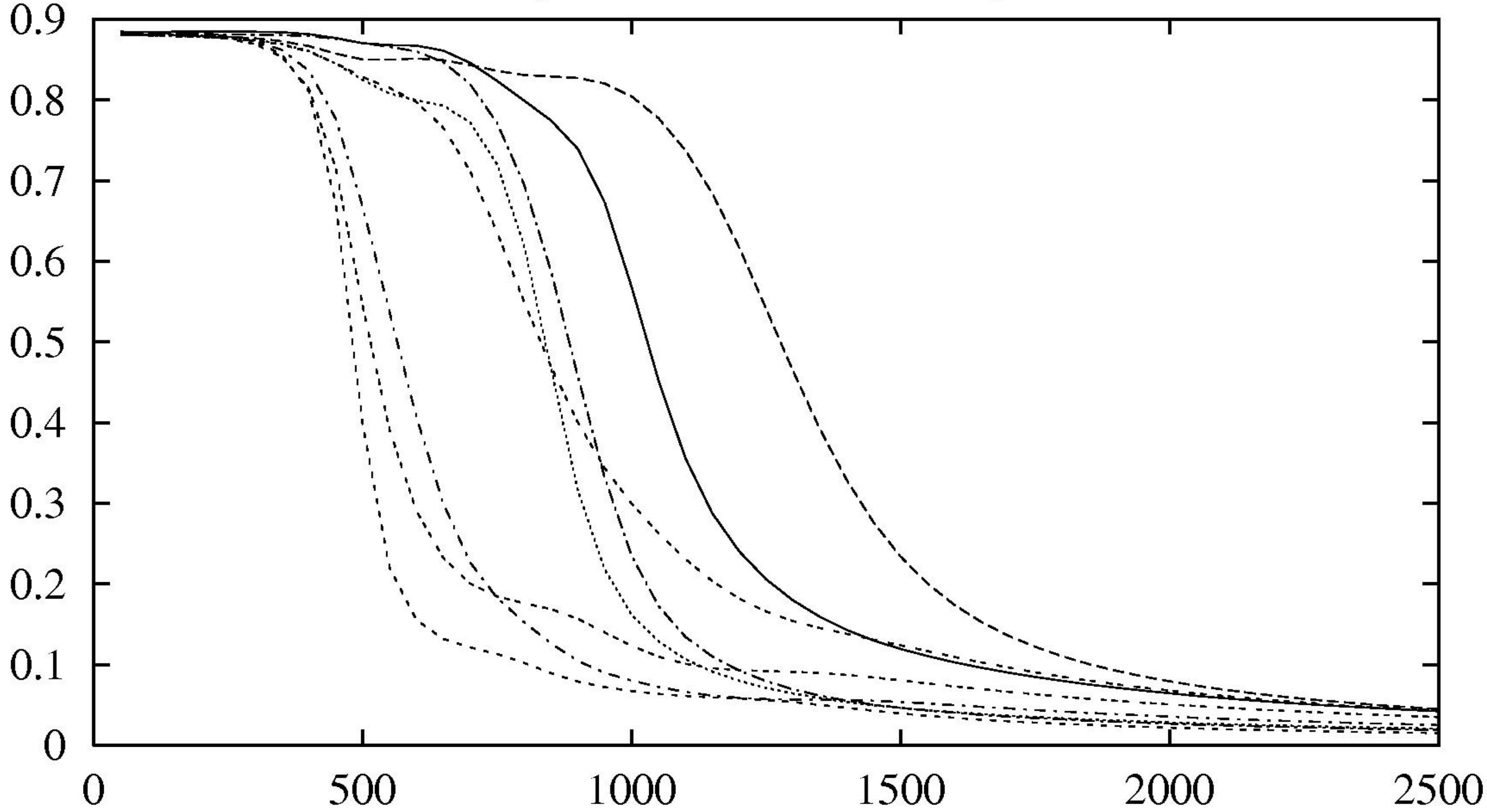
Neural networks for data compression

Learned hidden layer representation:

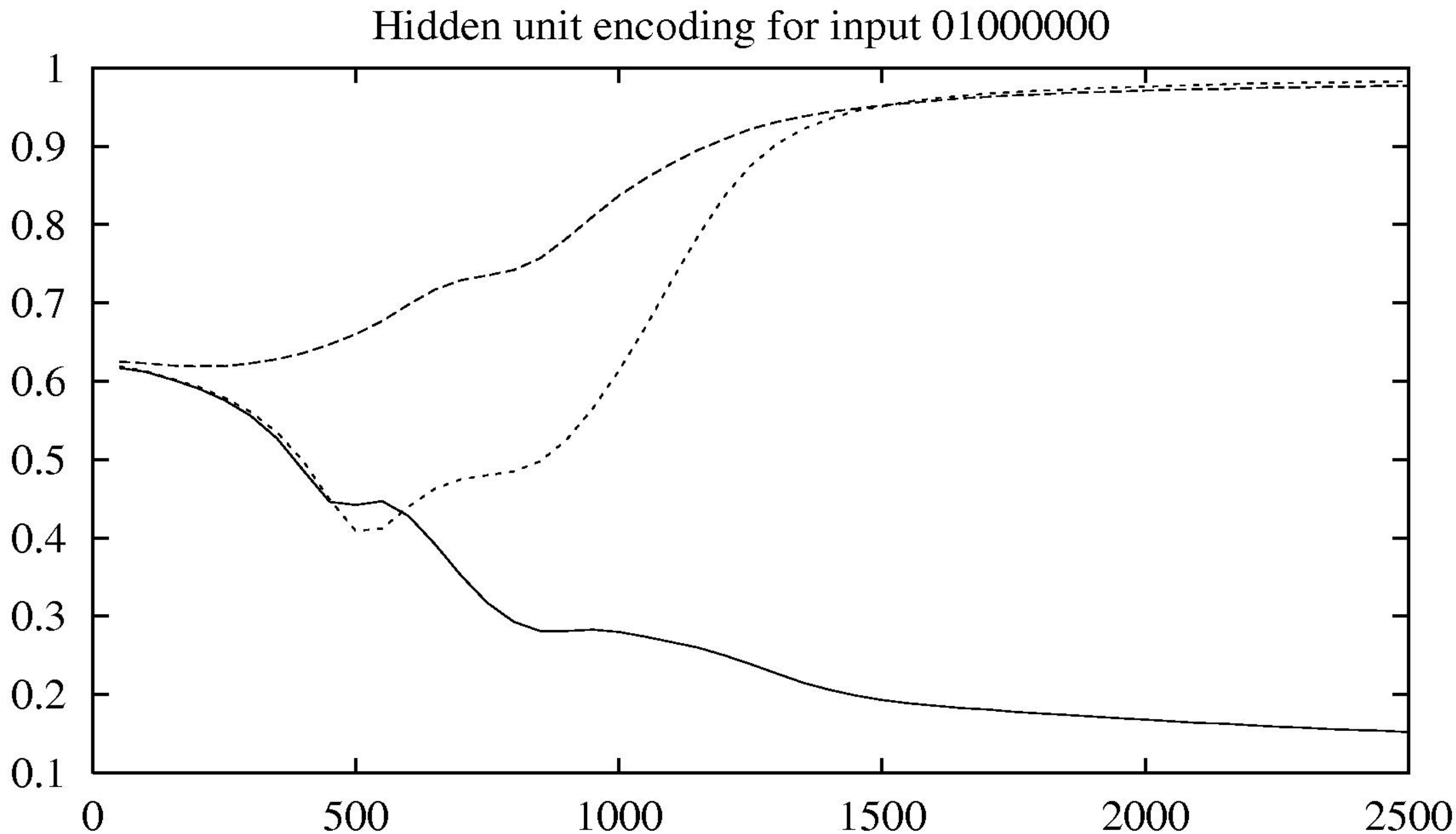
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Training

Sum of squared errors for each output unit

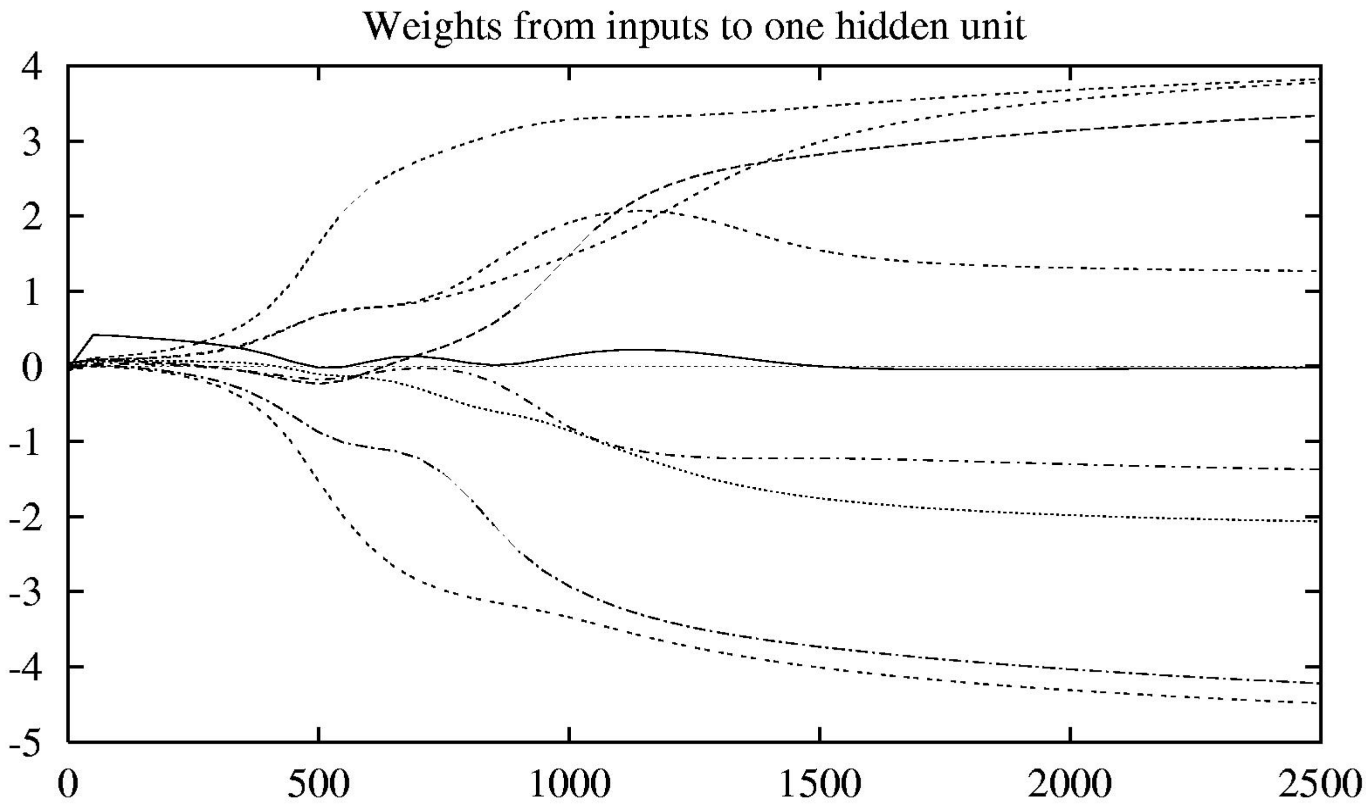


Training



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

Training


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

Neural networks for data compression

- Once weights are trained:
  - Use input > hidden layer weights to encode data
  - Store or transmit encoded, compressed form of data
  - Use hidden > output layer weights to decode

Convergence of backpropagation

Gradient descent to some local minimum

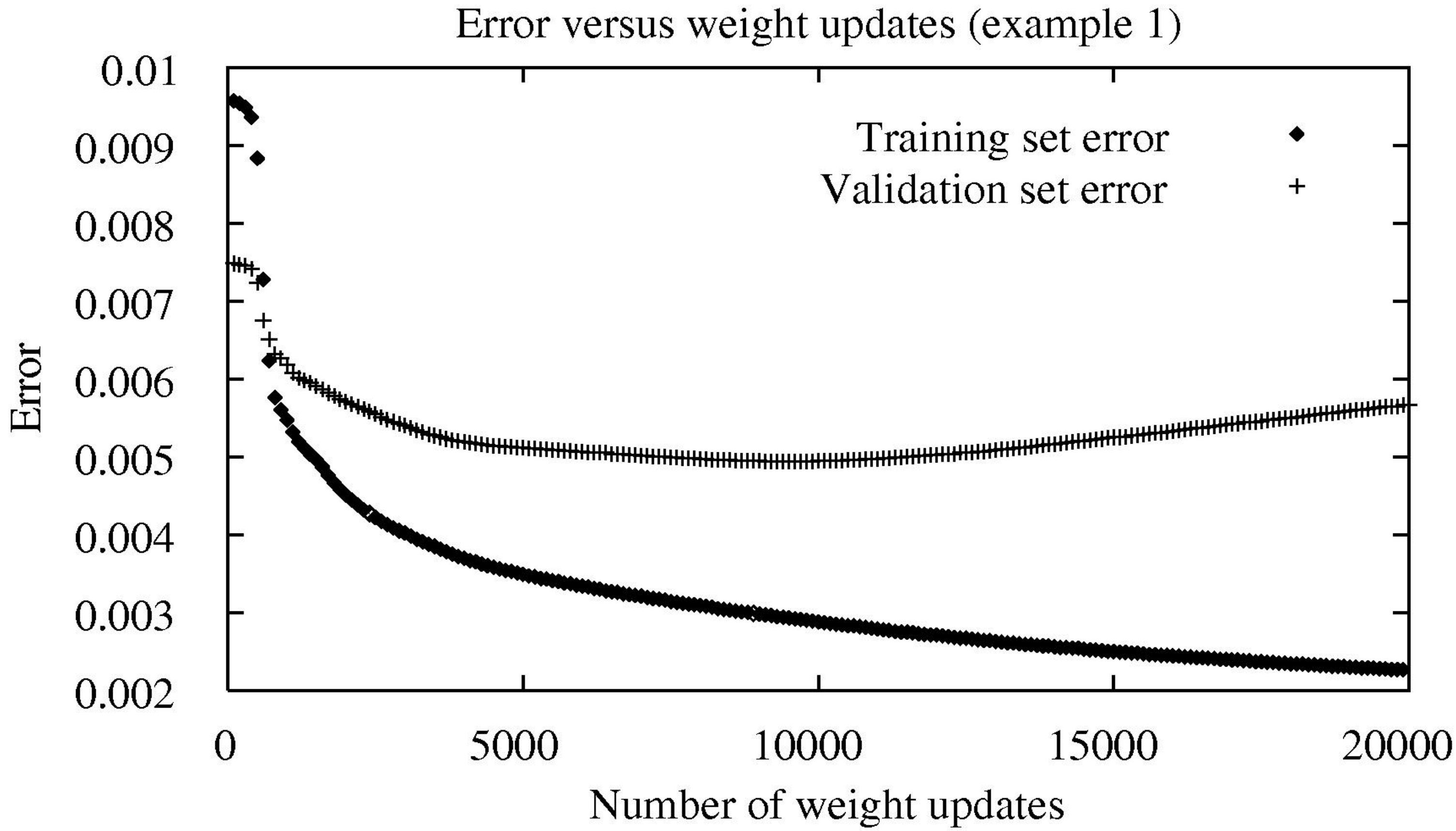
- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

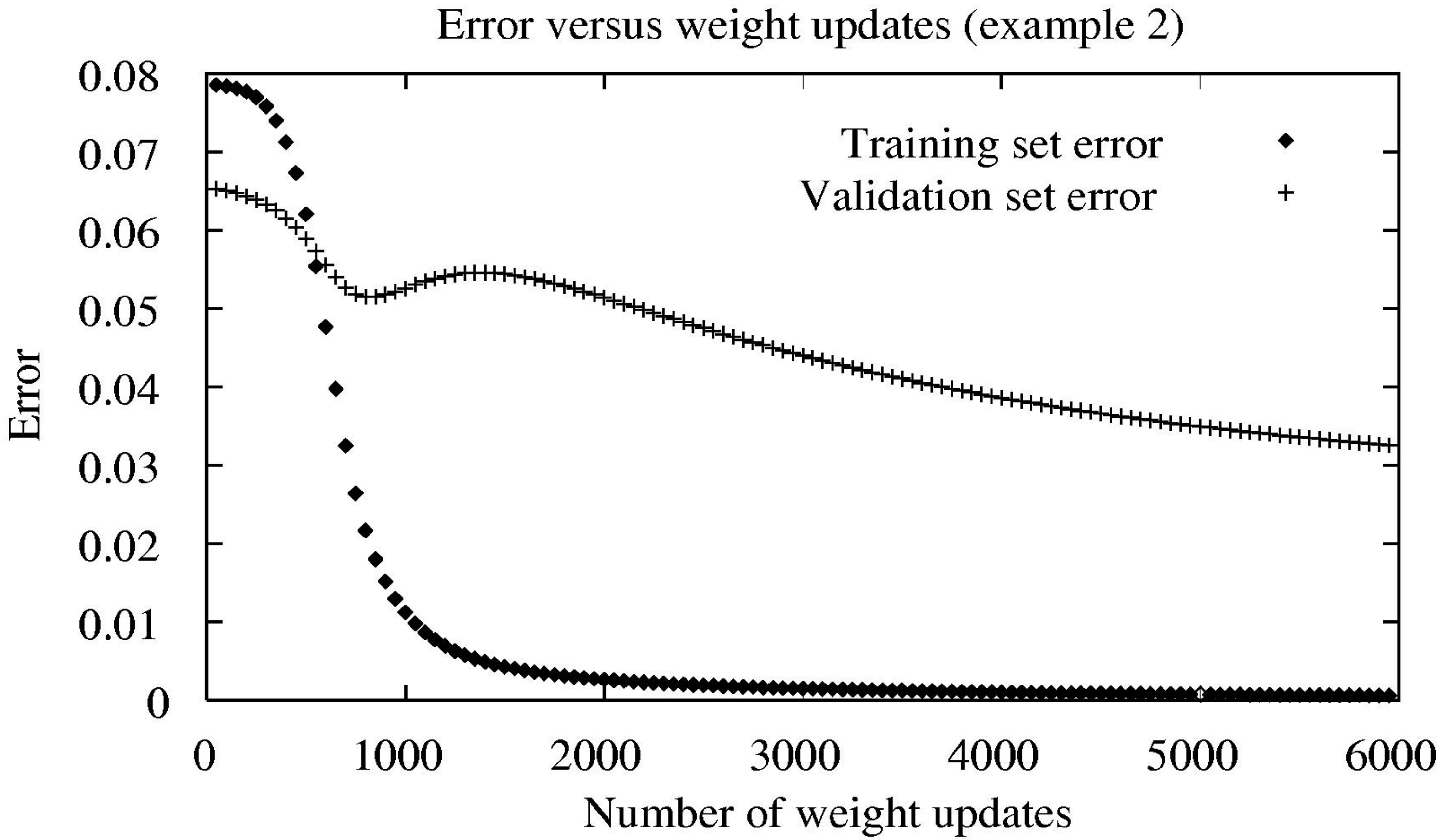
Overfitting in neural networks

- Robot perception task (example 1)



Overfitting in neural networks

- Robot perception task (example 2)



Avoiding overfitting in neural networks

Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Weight sharing

Early stopping

Expressiveness of multilayer neural networks

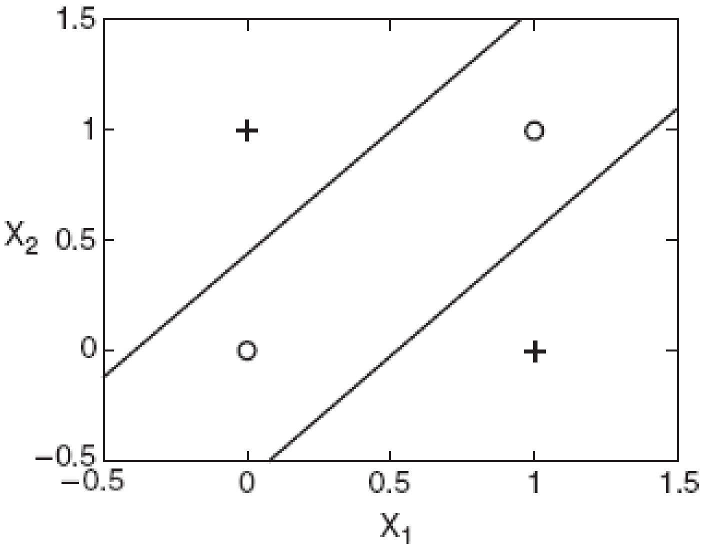
Boolean functions:

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

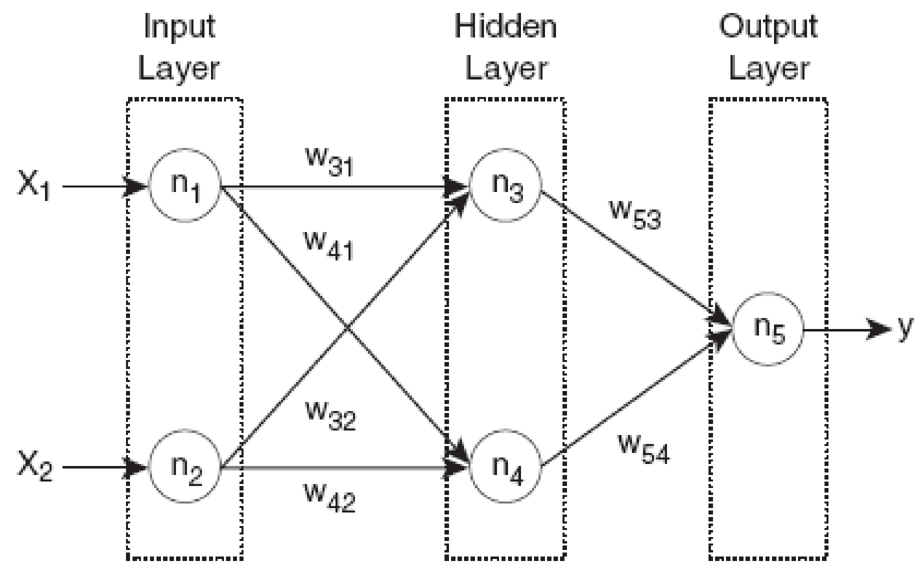
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Expressiveness of multilayer neural networks



(a) Decision boundary.

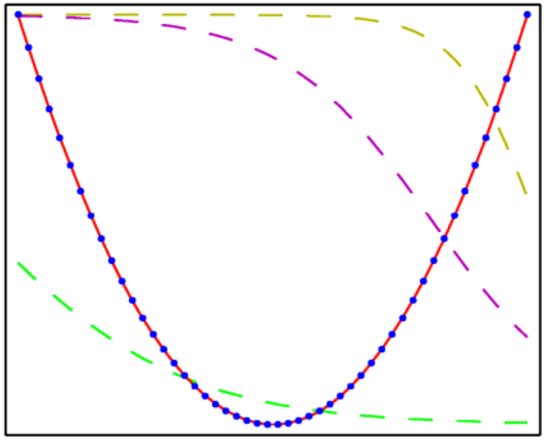


(b) Neural network topology.

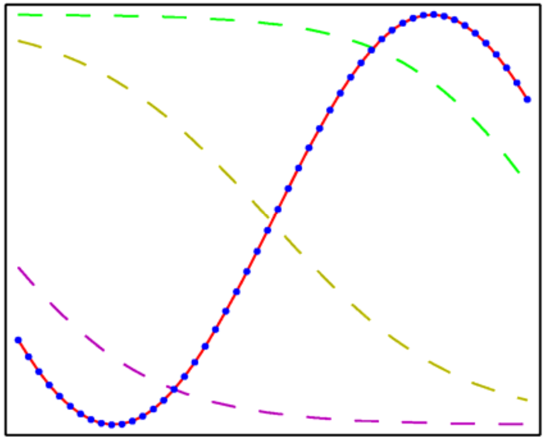
Figure 5.19. A two-layer, feed-forward neural network for the XOR problem.

Expressiveness of multilayer neural networks

- Trained two-layer network with three hidden units (`tanh` activation function) and one linear output unit.
  - Blue dots: 50 data points from *f*(*x*), where *x* uniformly sampled over range (-1, 1).
  - Grey dashed curves: outputs of the three hidden units.
  - Red curve: overall network function.



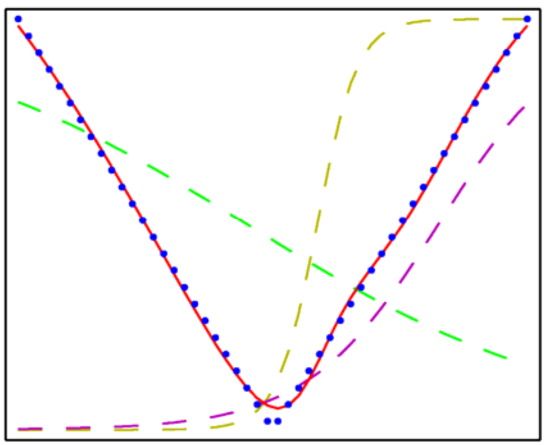
$f(x) = x^2$



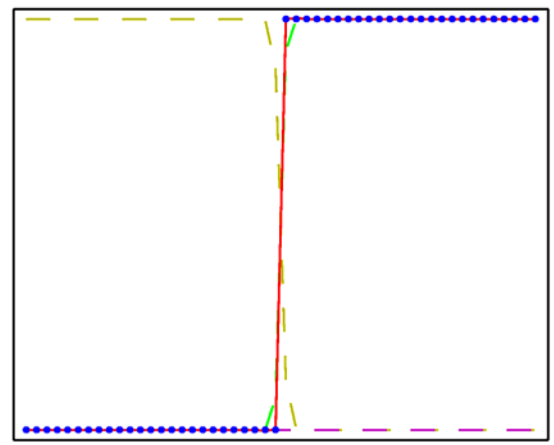
$f(x) = \sin(x)$

Expressiveness of multilayer neural networks

- Trained two-layer network with three hidden units (`tanh` activation function) and one linear output unit.
  - Blue dots: 50 data points from *f*(*x*), where *x* uniformly sampled over range (-1, 1).
  - Grey dashed curves: outputs of the three hidden units.
  - Red curve: overall network function.



f(*x*) = abs(*x*)



f(*x*) = *H*(*x*)
Heaviside step function

Expressiveness of multilayer neural networks

- Two-class classification problem with synthetic data.
- Trained two-layer network with two inputs, two hidden units (`tanh` activation function) and one logistic sigmoid output unit.

  Blue lines:

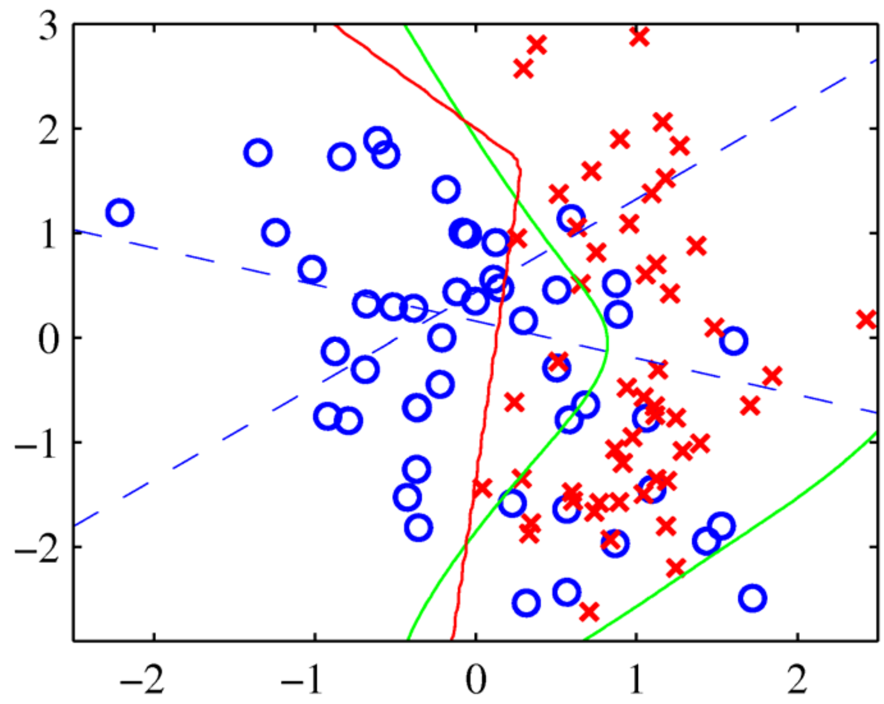
  $z = 0.5$ contours for hidden units

  Red line:

  $y = 0.5$ decision surface for overall network

  Green line:

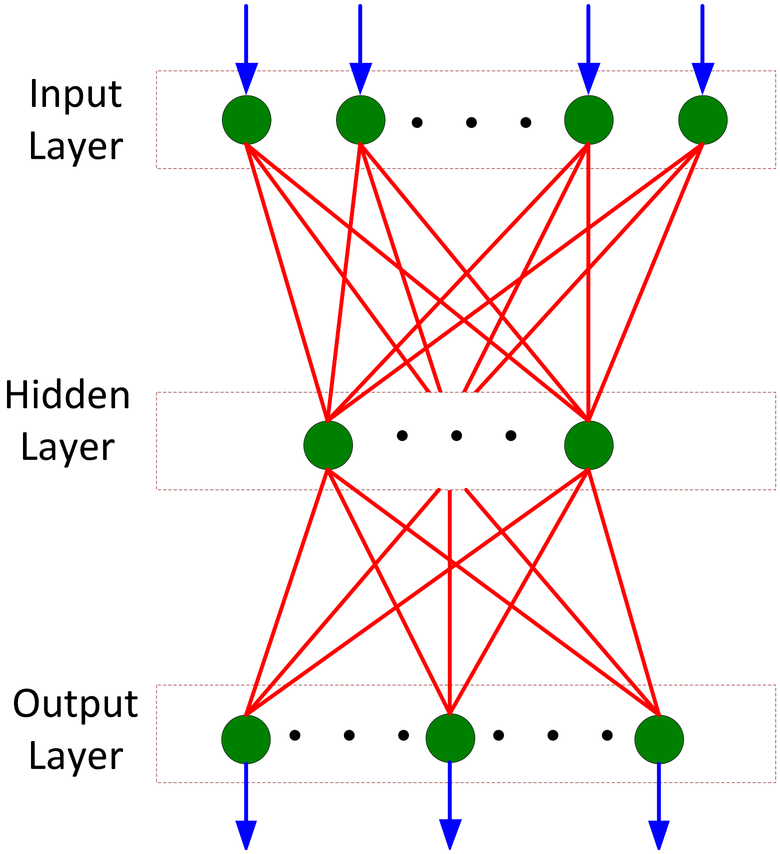
  optimal decision boundary computed from distributions used to generate data



Equivalence of neural networks with other learning algorithms

logistic transfer function

linear transfer function

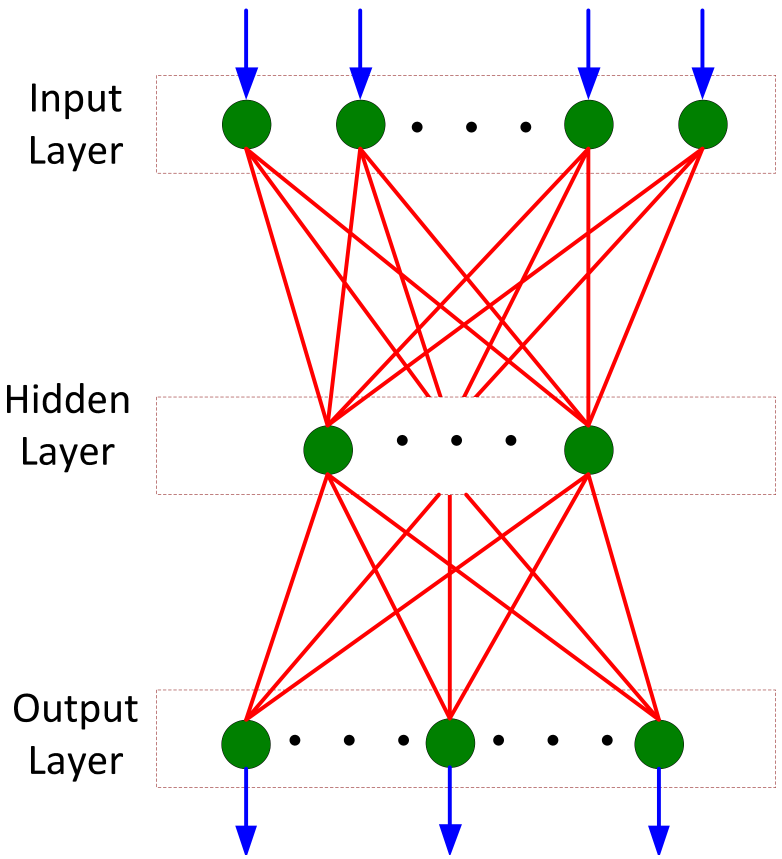


Each hidden unit is a logistic regression model, whose **w** vector is being trained while trying to match multiple, competing outputs.

Equivalence of neural networks with other learning algorithms

linear transfer function

linear transfer function



This entire network is equivalent to:

Matrix factorization!