

Backpropagation Networks

Introduction to Backpropagation

- In 1969 a method for learning in multi-layer network, **Backpropagation**, was invented by **Bryson and Ho**.
- The Backpropagation algorithm is a sensible approach for **dividing the contribution of each weight**.
- Works **basically** the same as perceptrons

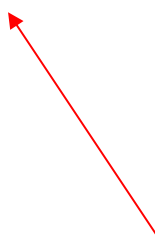
Backpropagation Learning Principles: **Hidden Layers** and **Gradients**

There are two **differences for the updating rule** :

- 1)** The **activation of the hidden unit** is used instead of activation of the input value.
- 2)** The rule contains **a term** for the **gradient** of the activation function.

Backpropagation Network training

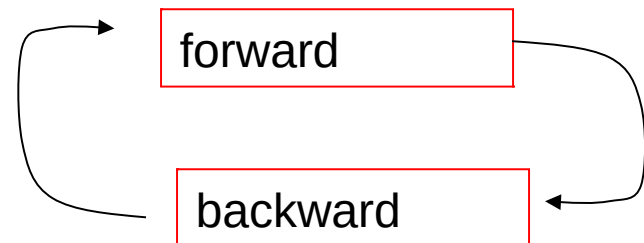
- 1. **Initialize** network with **random** weights
- 2. **For all** training cases (**called examples**):
 - **a.** Present training inputs to network and calculate output
 - **b.** For all layers (starting with output layer, back to input layer):
 - i. Compare **network output** with **correct output** (error function)
 - ii. **Adapt weights** in current layer



This is
what
you
want

Backpropagation Learning Details

- Method for **learning weights** in feed-forward (FF) nets
- Can't use Perceptron Learning Rule
 - no **teacher values** are possible for **hidden units**
- Use **gradient descent** to minimize the error
 - **propagate deltas** to **adjust for errors**
backward from outputs
to hidden layers
to inputs



Backpropagation Algorithm – Main Idea – error in hidden layers

The ideas of the algorithm can be summarized as follows :

1. Computes the **error term for the output units** using the observed error.
2. From output layer, repeat
 - propagating the error term back to the previous layer and
 - **updating the weights between the two layers** until the earliest hidden layer is reached.

Backpropagation Algorithm

- Initialize weights (typically random!)
- Keep doing epochs
 - For each example e in training set do
 - **forward pass** to compute
 - $O = \text{neural-net-output}(\text{network}, e)$
 - $\text{miss} = (T - O)$ at each output unit
 - **backward pass** to calculate deltas to weights
 - update all weights
 - end
- until **tuning set error stops improving**

Forward pass explained earlier

Backward pass explained in next slide

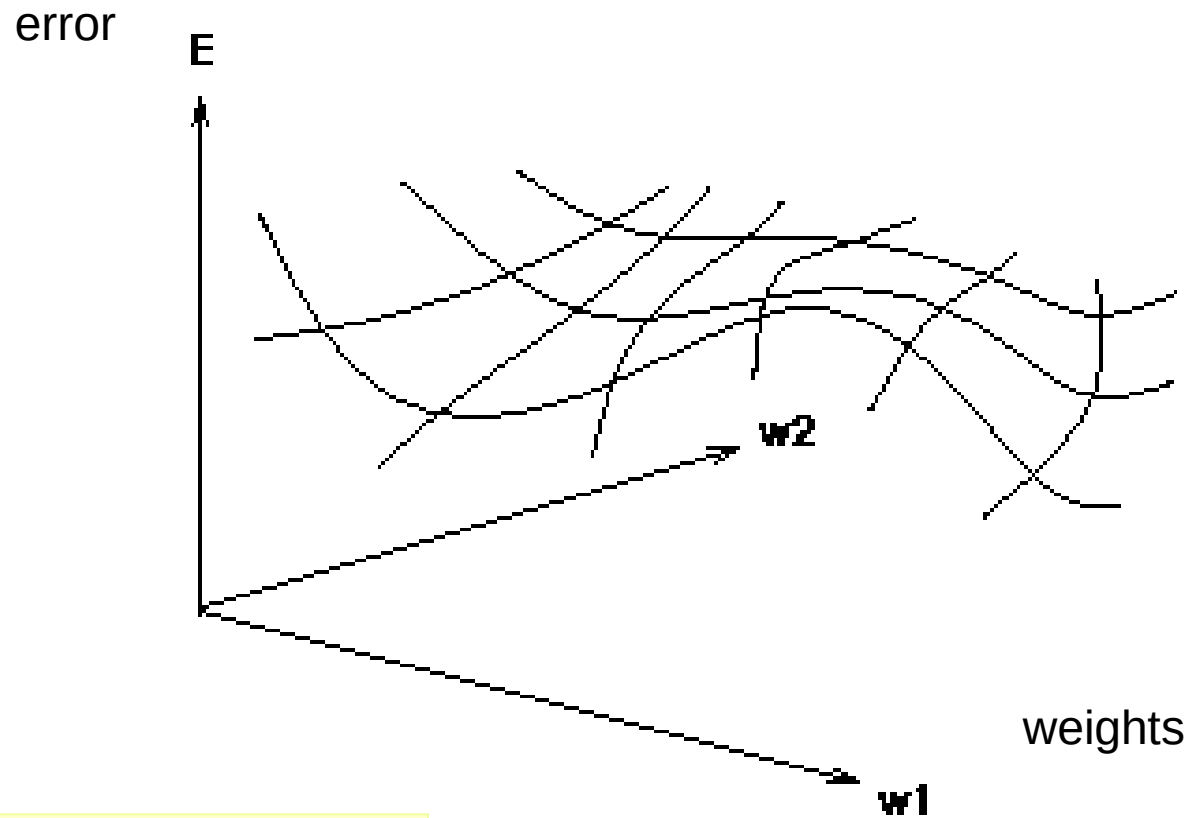
Backward Pass

- Compute **deltas** to weights
 - from **hidden** layer
 - to **output** layer
- Without changing any weights (yet), compute the **actual contributions**
 - within the hidden layer(s)
 - and **compute deltas**

Gradient Descent

- Think of the N weights as a point in an N -dimensional space
- Add a dimension for the observed error
- Try to minimize your position on the “error surface”

Error Surface



Error as function of
weights in
multidimensional space

Gradient

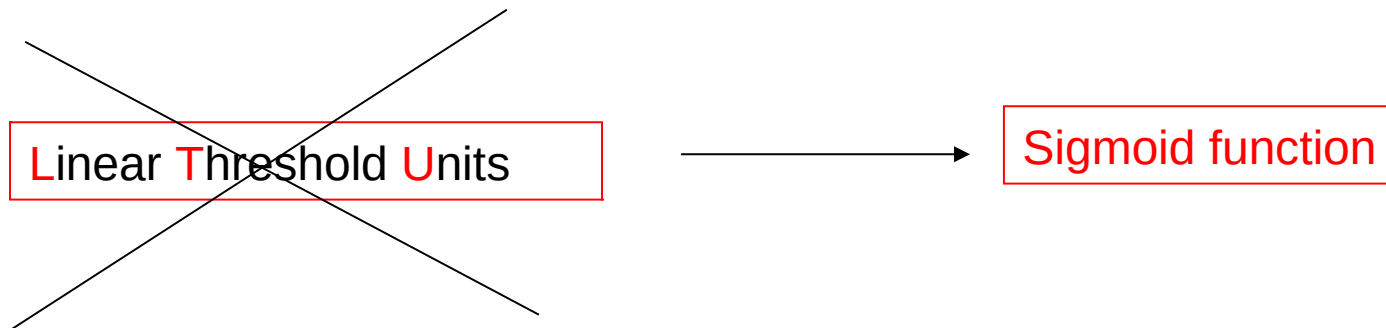
Compute
deltas

- Trying to make error decrease the fastest
- **Compute:**
 - $\text{Grad}_E = [dE/dw_1, dE/dw_2, \dots, dE/dw_n]$
- **Change** i -th weight by
 - $\text{delta}_{w_i} = -\text{alpha} * dE/dw_i$
- We need a **derivative**!
- Activation function must be **continuous**, differentiable, non-decreasing, and easy to compute

Derivatives of error for weights

Can't use LTU

- To effectively assign credit / blame to units in hidden layers, **we want to look at the first derivative** of the activation function
- **Sigmoid function** is easy to **differentiate** and easy to compute forward



Updating hidden-to-output

- We have **teacher supplied** desired values

- $$\text{delta}_{wji} = \alpha * a_j * (T_i - O_i) * g'(in_i)$$
$$= \alpha * a_j * (T_i - O_i) * O_i * (1 - O_i)$$

– for sigmoid the derivative is, $g'(x) = g(x) * (1 - g(x))$

alpha

Here we have
general formula with
derivative, next we
use for sigmoid

miss

derivative

Updating interior weights

- Layer k units provide values to all layer k+1 units
 - “miss” is **sum of misses** from all units on k+1
 - **miss_j** = $\sum [a_i(1 - a_i) (T_i - a_i) w_{ji}]$
 - weights coming into this unit are **adjusted based on their contribution**

$$\text{delta}_{kj} = \alpha * I_k * a_j * (1 - a_j) * \text{miss}_j$$

For layer k+1

Compute deltas

How do we pick α ?

1. Tuning set, or
2. Cross validation, or
3. Small for slow, conservative learning

How many hidden layers?

- Usually just **one** (i.e., a 2-layer net)
- How many **hidden units** in the layer?
 - **Too few** ==> can't learn
 - Too many ==> poor generalization

How big a training set?

- Determine your **target error rate**, e
- **Success rate** is $1 - e$
- Typical training set approx. n/e , where n is the number of weights in the net
- Example:
 - $e = 0.1$, $n = 80$ weights
 - training set **size 800**
trained until **95% correct training** set classification
should produce 90% correct classification
on **testing set** (typical)

Examples of Backpropagation Learning

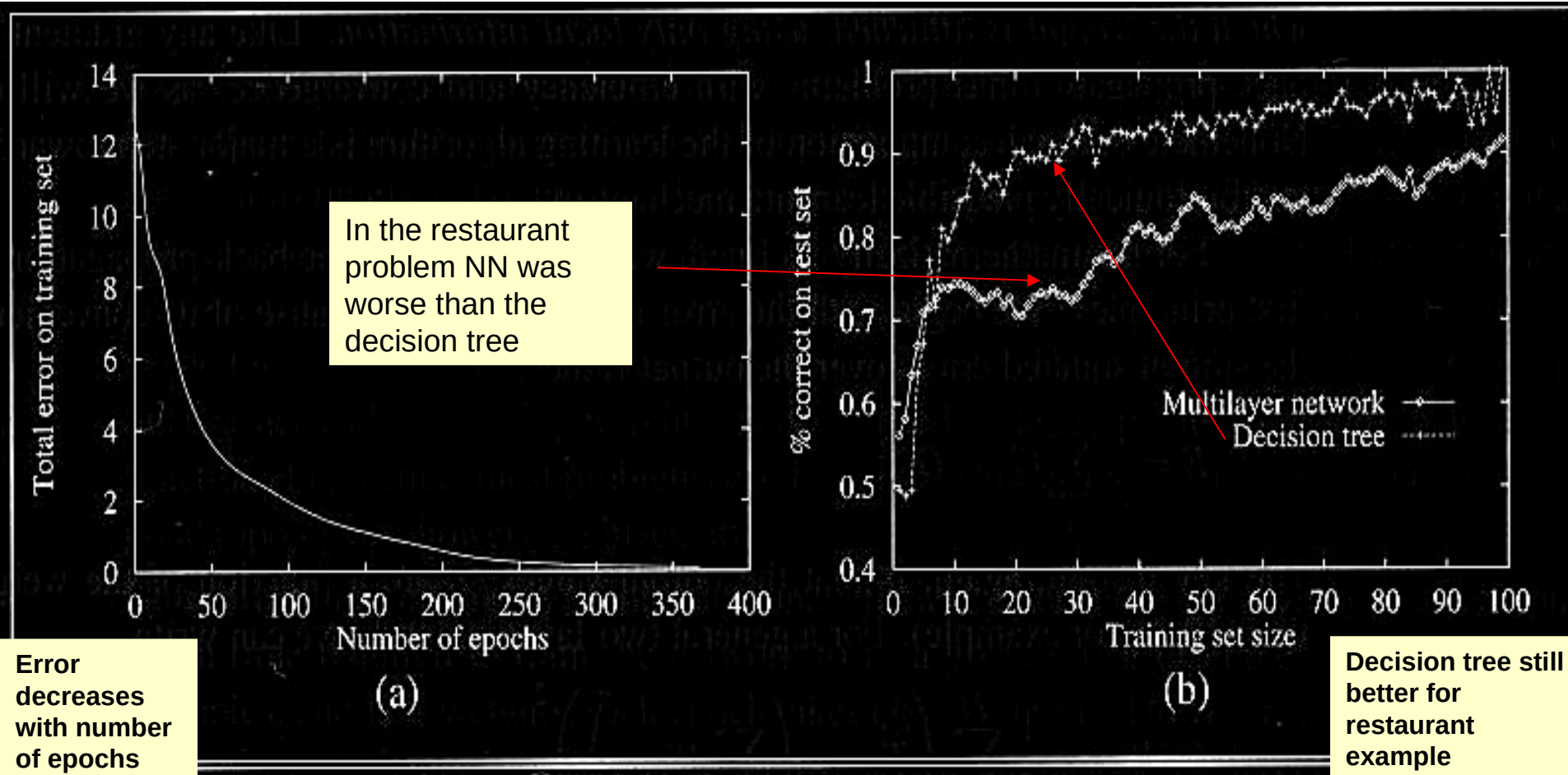


Figure 19.15 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves for a back-propagation and decision-tree learning.

Examples of Backpropagation Learning

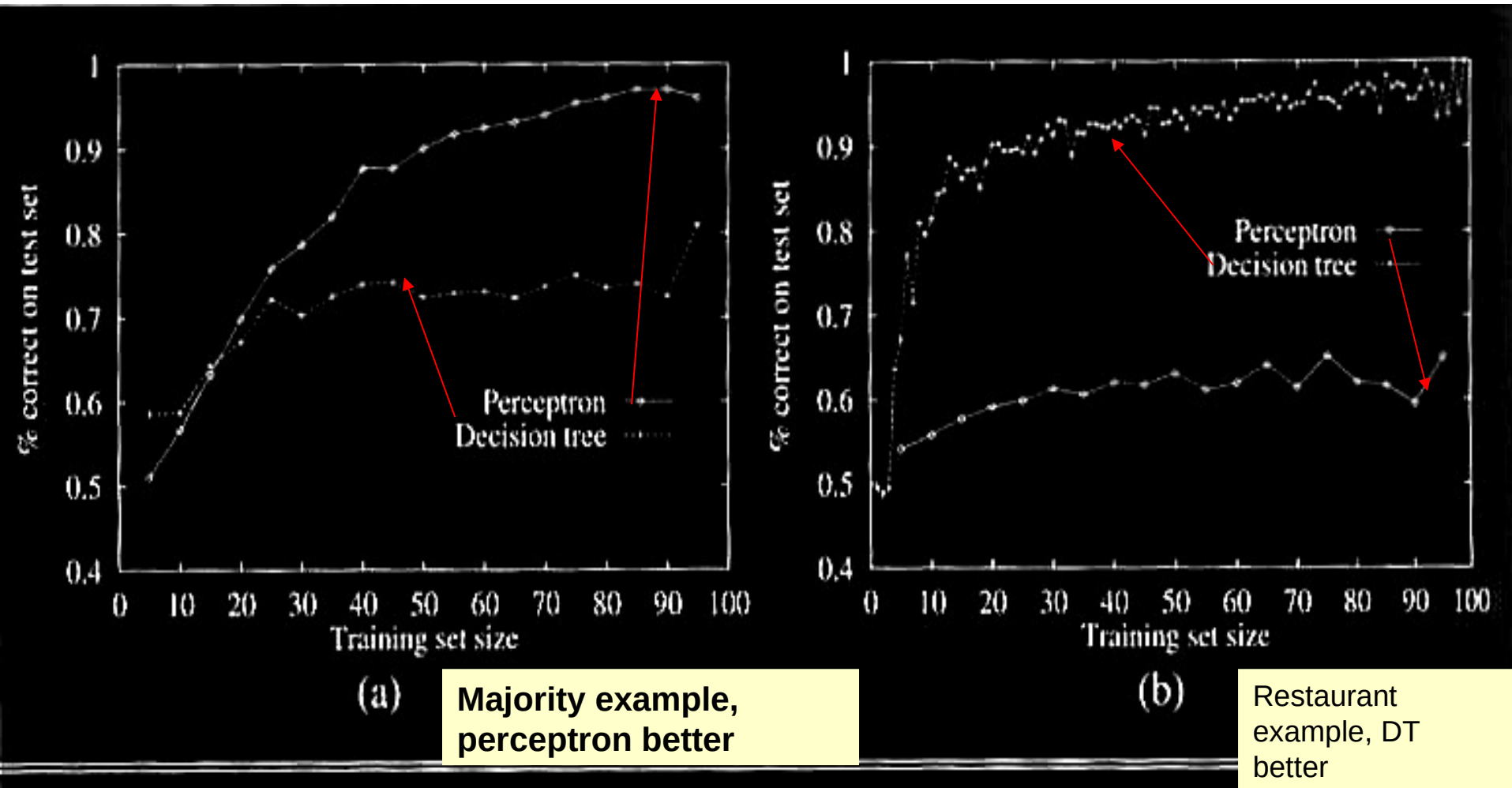


Figure 19.12 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate for the restaurant example.


Backpropagation Learning Math

$$E_{\text{out } i} = d_{\text{out } i} - \text{out}_i$$

$$E_{\text{total}} = \sum_{i=0}^{\text{num}(n_{\text{out}})} E_{\text{out } i}^2$$

$$E_{\text{hid } i} = \sum_{k=1}^{\text{num}(n_{\text{out}})} E_{\text{out } k} \cdot w_{\text{out } i, k}$$

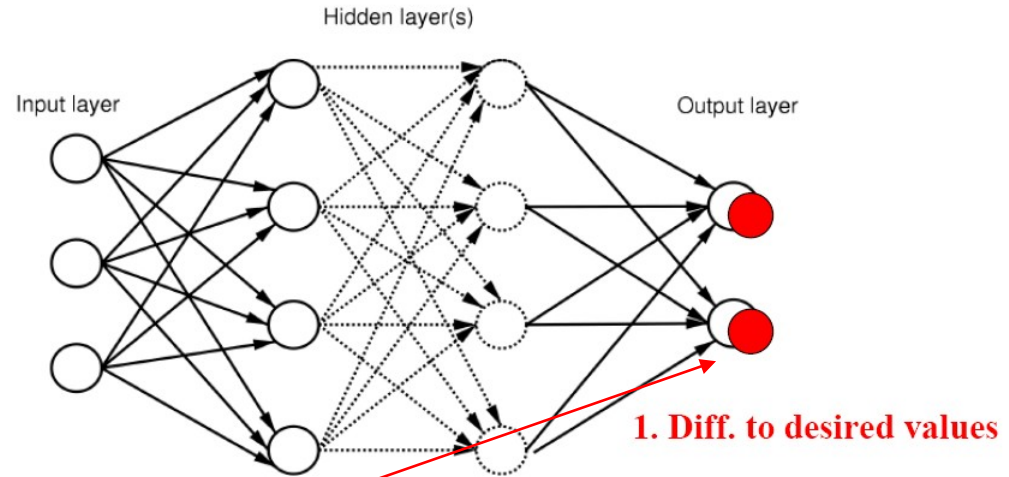
$$\text{diff}_{\text{hid } i} = E_{\text{hid } i} \cdot (1 - o(n_{\text{hid } i})) \cdot o(n_{\text{hid } i})$$



See next
slide for
explanation

Visualization of Backpropagation learning

Backpropagation Learning



Brauni 2003

8

Backpropagation Learning

$$E_{out\ i} = d_{out\ i} - out_i$$

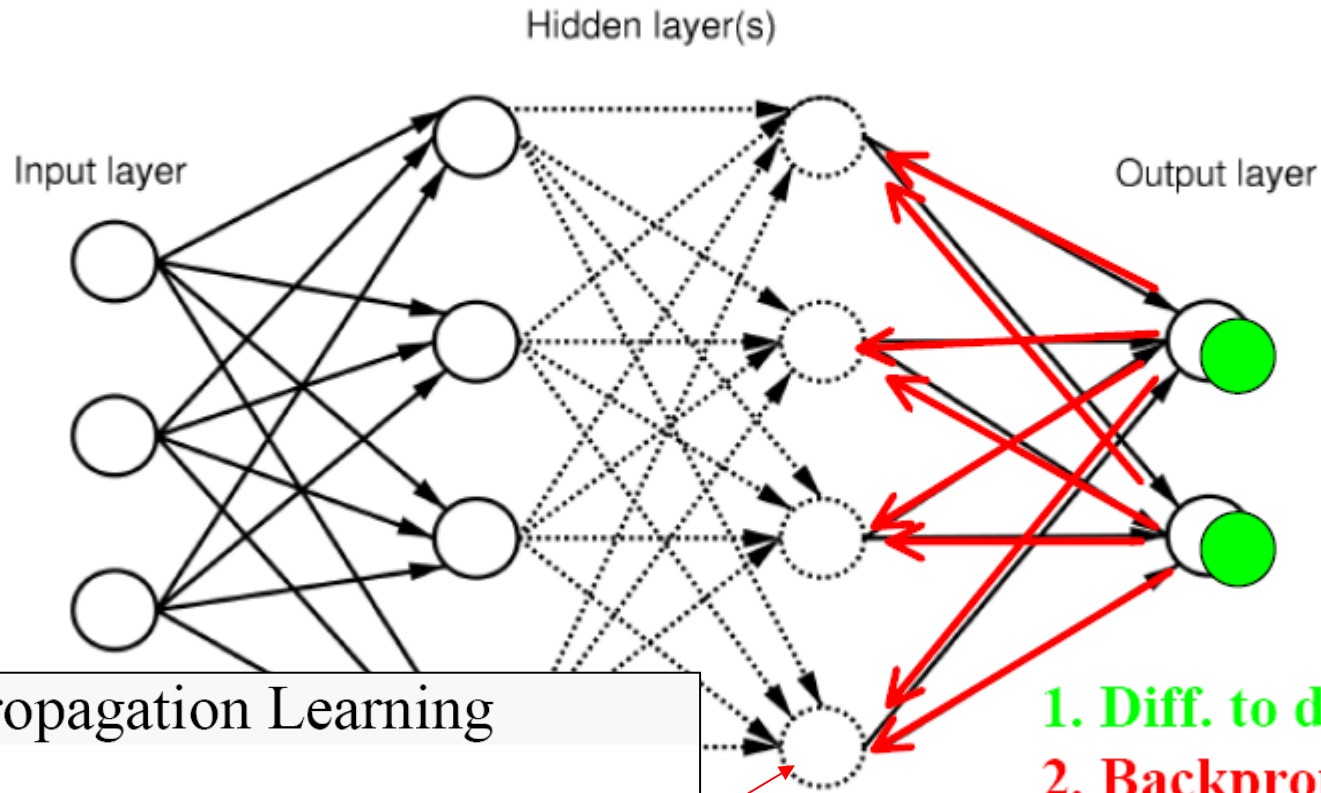
$$E_{total} = \sum_{i=0}^{num(n_{out})} E_{out\ i}^2$$

$$E_{hid\ i} = \sum_{k=1}^{num(n_{out})} E_{out\ k} \cdot w_{out\ i,k}$$

$$diff_{hid\ i} = E_{hid\ i} \cdot (1 - o(n_{hid\ i})) \cdot o(n_{hid\ i})$$

Backprop output layer

Backpropagation Learning



Backpropagation Learning

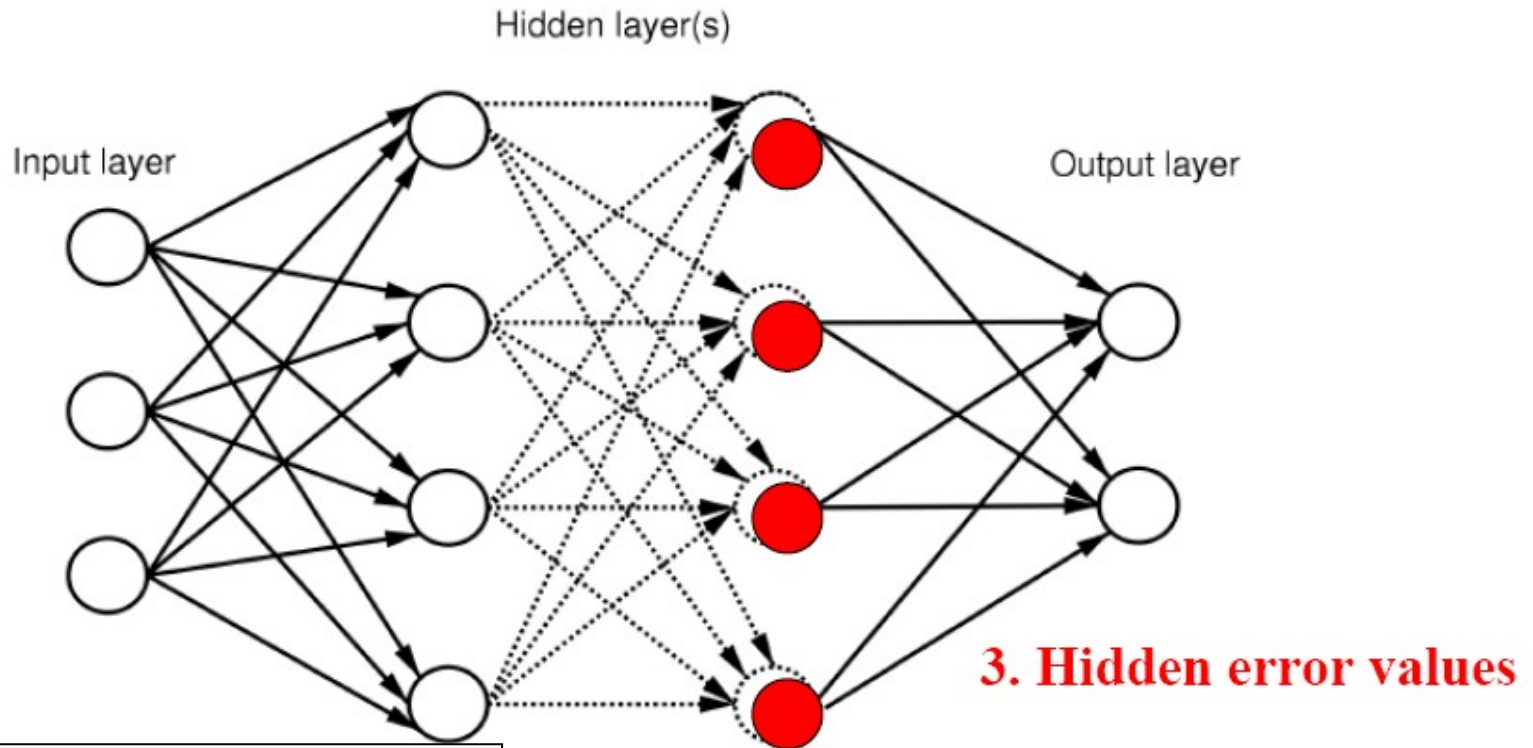
$$E_{out\ i} = d_{out\ i} - out_i$$

$$E_{total} = \sum_{i=0}^{num(n_{out})} E_{out\ i}^2$$

$$E_{hid\ i} = \sum_{k=1}^{num(n_{out})} E_{out\ k} \cdot w_{out\ i, k}$$

$$diff_{hid\ i} = E_{hid\ i} \cdot (1 - o(n_{hid\ i})) \cdot o(n_{hid\ i})$$

Backpropagation Learning



Backpropagation Learning

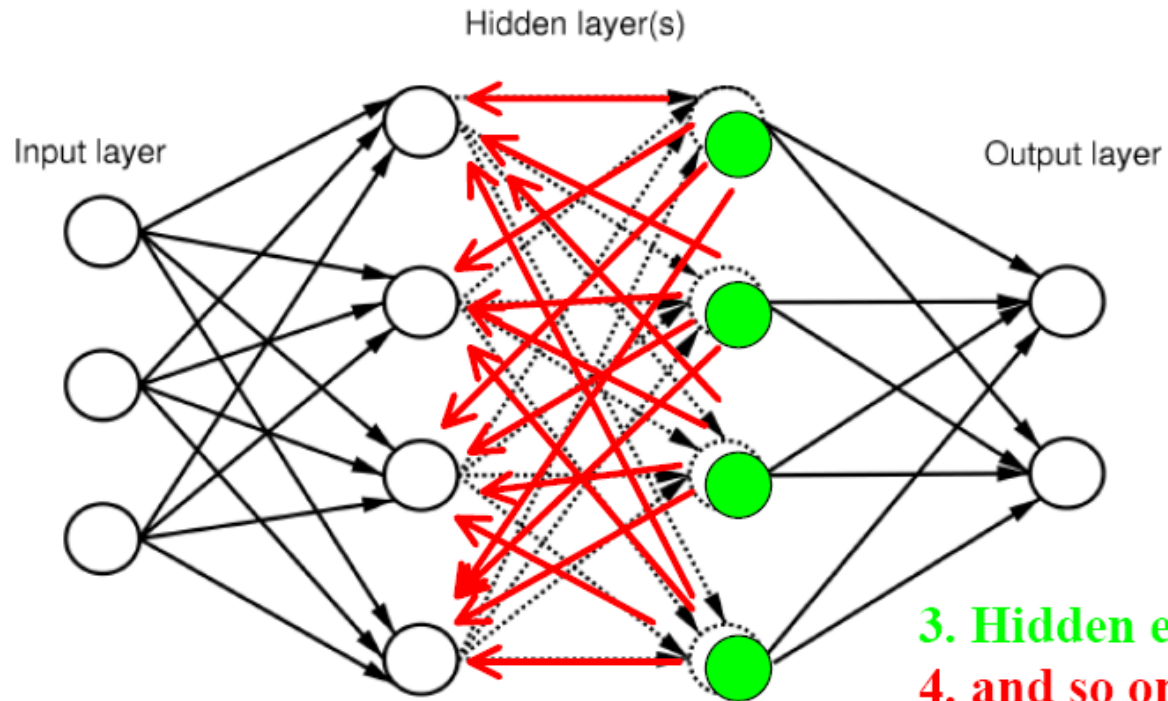
$$E_{out\ i} = d_{out\ i} - out_i$$

$$E_{total} = \sum_{i=0}^{num(n_{out})} E_{out\ i}^2$$

$$E_{hid\ i} = \sum_{k=1}^{num(n_{out})} E_{out\ k} \cdot w_{out\ i, k}$$

$$diff_{hid\ i} = E_{hid\ i} \cdot (1 - o(n_{hid\ i})) \cdot o(n_{hid\ i})$$

Backpropagation Learning



Backpropagation Learning

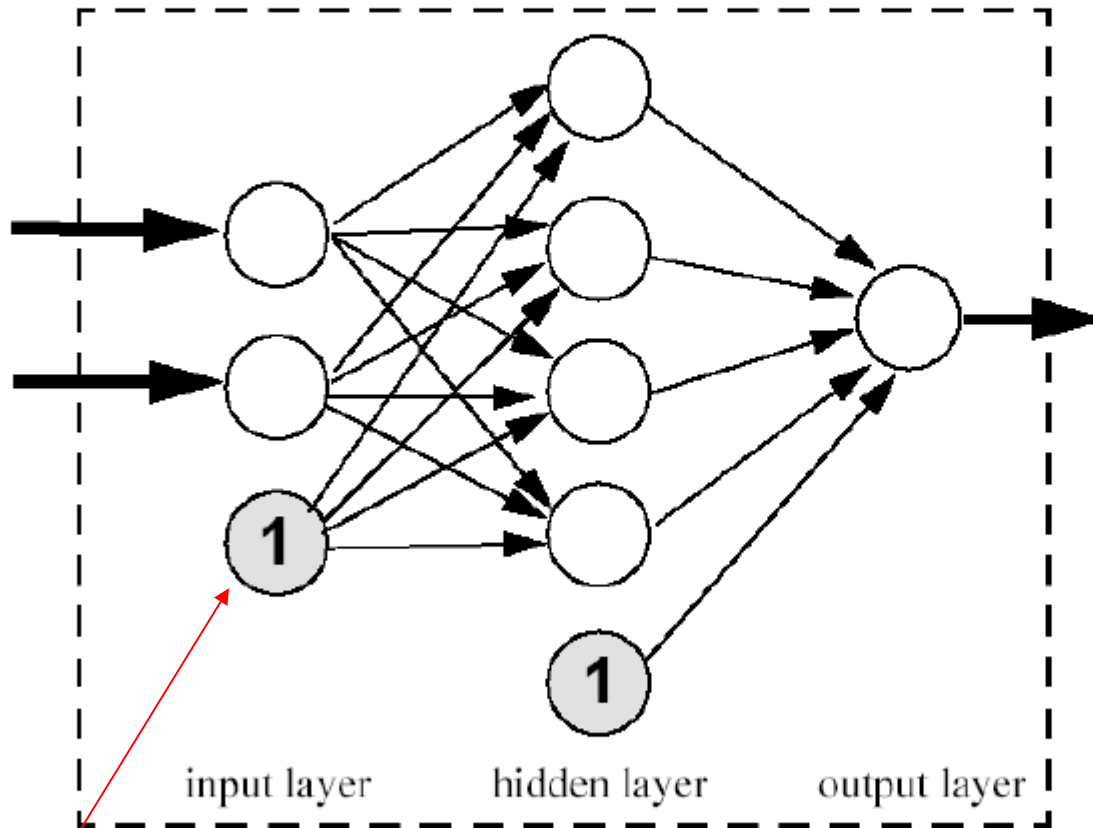
$$E_{out\ i} = d_{out\ i} - out_i$$

$$E_{total} = \sum_{i=0}^{num(n_{out})} E_{out\ i}^2$$

$$E_{hid\ i} = \sum_{k=1}^{num(n_{out})} E_{out\ k} \cdot w_{out\ i, k}$$

$$diff_{hid\ i} = E_{hid\ i} \cdot (1 - o(n_{hid\ i})) \cdot o(n_{hid\ i})$$

Bias Neurons in **Backpropagation** Learning



Bias neurons

bias neuron in input layer

Software for Backpropagation Learning

Training pairs

This routine
calculate error for
backpropagation

```
float backprop(float train_in[NIN], float train_out[NOUT])  
/* returns current square error value */  
{ int i,j;  
  float err_total;  
  float N_out[NOUT],err_out[NOUT];  
  float diff_out[NOUT];  
  float N_hid[NHID], err_hid[NHID], diff_hid[NHID];  
  
  //run network, calculate difference to desired output  
  feedforward(train_in, N_hid, N_out);  
  err_total = 0.0;  
  for (i=0; i<NOUT; i++)  
  { err_out[i] = train_out[i]-N_out[i];  
    diff_out[i]= err_out[i] * (1.0-N_out[i]) * N_out[i];  
    err_total += err_out[i]*err_out[i];  
  }  
  ...  
}
```

Run network forward.
Was explained earlier

Calculate difference to
desired output

Calculate total error

Software for Backpropagation Learning continuation

Here we do not use alpha, the learning rate

Update output weights

Calculate hidden difference values

Update input weights

Return total error

```
...
// update w_out and calculate hidden difference values
for (i=0; i<NHID; i++)
{ err_hid[i] = 0.0;
  for (j=0; j<NOUT; j++)
  { err_hid[i] += err_out[j] * w_out[i][j];
    w_out[i][j] += diff_out[j] * N_hid[i];
  }
  diff_hid[i] = err_hid[i] * (1.0-N_hid[i]) * N_hid[i];
}
// update w_in
for (i=0; i<NIN; i++)
  for (j=0; j<NHID; j++)
    w_in[i][j] += diff_hid[j] * train_in[i];

return err_total;
}
```

The general Backpropagation Algorithm for updating weights in a multilayer network

Repeat until
convergent

Here we use alpha, the
learning rate

function BACK-PROP-UPDATE(*network*, *examples*, α) **returns** a network with modified weights

inputs: *network*, a multilayer network
examples, a set of input/output pairs
 α , the learning rate

Go through all
examples

Run network to
calculate its
output for this
example

repeat

for each *e* **in** *examples* **do**

/ Compute the output for this example */*

$\mathbf{O} \leftarrow \text{RUN-NETWORK}(\text{network}, \mathbf{I}^e)$

/ Compute the error and Δ for units in the output layer */*

$\text{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}$

Compute the
error in output

/ Update the weights leading to the output layer */*

$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \text{Err}_i^e \times g'(in_i)$

Update weights
to output layer

for each subsequent layer **in** *network* **do**

/ Compute the error at each node */*

$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$

Compute error in
each hidden layer

/ Update the weights leading into the layer */*

$W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j$

Update weights in
each hidden layer

end

end

until *network* has converged

return *network*

Return learned network

Examples and Applications of ANN

Neural Network in Practice

NNs are used for **classification** and **function approximation** or **mapping problems** which are:

- **Tolerant** of some imprecision.
- Have **lots of training data** available.
- **Hard and fast** rules **cannot** easily **be applied**.

NETalk (1987)

- Mapping **character strings** into **phonemes** so they can be pronounced by a computer
- Neural network trained **how to pronounce** each letter in a word in a sentence, **given the three letters before and three letters after it in a window**
- Output was the **correct phoneme**
- Results
 - 95% accuracy on the **training data**
 - 78% accuracy on the **test set**

Other Examples

- **Neurogammon** (Tesauro & Sejnowski, 1989)
 - Backgammon learning program
- **Speech Recognition** (Waibel, 1989)
- **Character Recognition** (LeCun et al., 1989)
- **Face Recognition** (Mitchell)

ALVINN

- Steer a van down the road
 - 2-layer feedforward
 - using backpropagation for learning
 - Raw input is 480 x 512 pixel image 15x per sec
 - Color image **preprocessed** into 960 input units
 - 4 hidden units
 - 30 output units, each is a steering direction

Neural Network Approaches



Learning on-the-fly

- ALVINN learned as the vehicle traveled
 - initially by **observing a human** driving
 - learns from its own driving by watching for **future corrections**
 - **never saw bad driving**
 - didn't know what was dangerous, NOT correct
 - computes alternate views of the road (rotations, shifts, and fill-ins) **to use as "bad" examples**
 - keeps a buffer pool of **200 pretty old examples to avoid overfitting** to only the most recent images



Feed-forward vs. Interactive Nets

- Feed-forward
 - activation propagates in one direction
 - We usually focus on this
- Interactive
 - activation propagates forward & backwards
 - propagation continues until equilibrium is reached in the network
 - We do not discuss these networks here, complex training. May be unstable.

Ways of learning with an ANN

- **Add** nodes & connections
- **Subtract** nodes & connections
- **Modify connection** weights
 - current focus
 - can simulate first two
- **I/O pairs:**
 - given the inputs, what should the output be?
[“typical” learning problem]

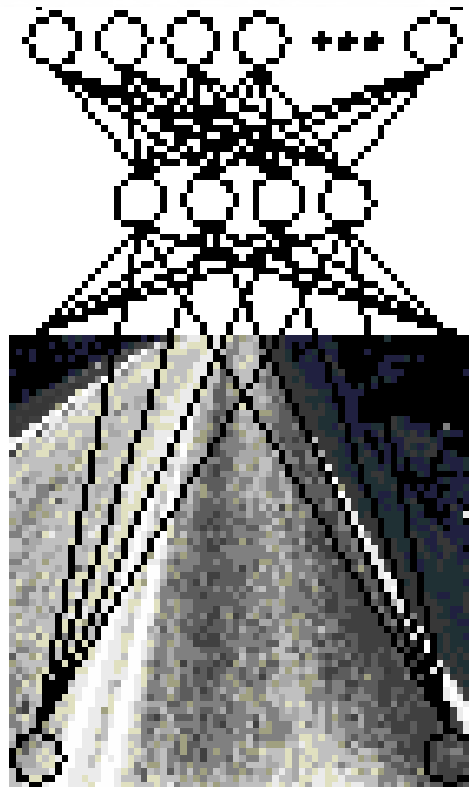
More Neural Network Applications

- May provide a model for massive parallel computation.
- More successful approach of “parallelizing” traditional serial algorithms.
- Can compute any computable function.
- Can do everything a normal digital computer can do.
- Can do even more under some impractical assumptions.

Neural Network Approaches to driving

- Use special hardware

- ASIC
- FPGA
- analog



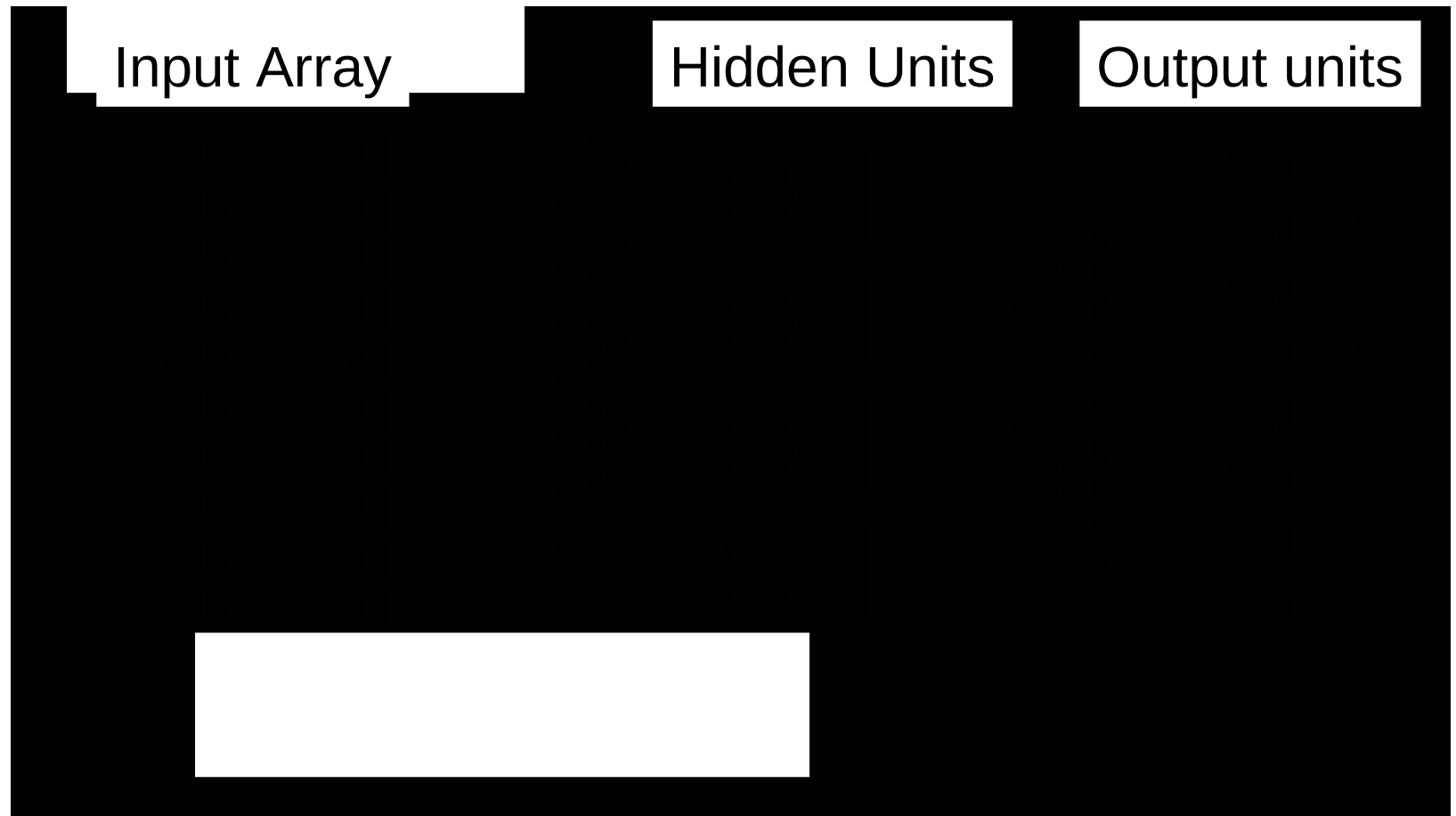
Output units

Hidden layer

Input units

- Developed in 1993.
- Performs driving with Neural Networks.
- An intelligent VLSI image sensor for road following.
- Learns to filter out **image details not relevant** to driving.

Neural Network Approaches



Actual **Products** Available

ex1. Enterprise Miner:

- **Single multi-layered feed-forward** neural networks.
- Provides business solutions for **data mining**.

ex2. Nestor:

- Uses Nestor Learning System (NLS).
- Several multi-layered feed-forward neural networks.
- **Intel has made such a chip** - NE1000 in VLSI technology.

Ex1. Software tool - Enterprise Miner

- Based on SEMMA (Sample, Explore, Modify, Model, Access) methodology.
- **Statistical tools** include :
Clustering, decision trees, linear and logistic regression and neural networks.
- **Data preparation tools** include :
Outlier detection, variable transformation, random sampling, and partition of data sets (into training, testing and validation data sets).

Ex 2. Hardware Tool - Nestor

- With low connectivity within each layer.
- Minimized connectivity within each layer results in rapid training and efficient memory utilization, **ideal for VLSI**.
- Composed of multiple neural networks, each specializing in a subset of information about the input patterns.
- Real time operation without the need of special computers or custom hardware DSP platforms
 - Software exists.

Problems with using ANNs

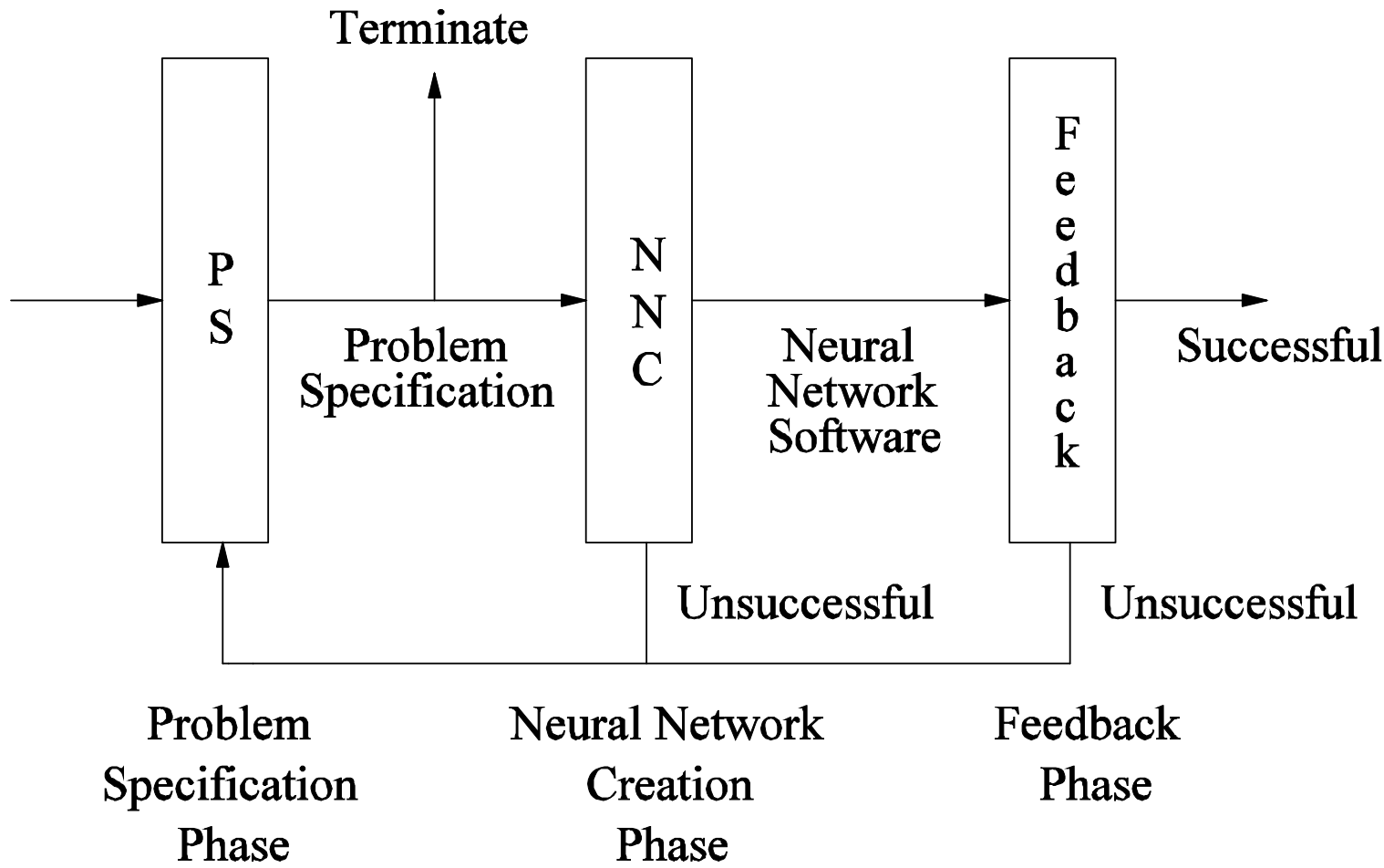
1. Insufficiently characterized development process compared with conventional software
 - What are the steps to create a neural network?
2. How do we create neural networks in a repeatable and predictable manner?
3. Absence of quality assurance methods for neural network models and implementations
 - How do I verify my implementation?

Solving Problem 1 – The Steps to create a ANN

Define the process of developing neural networks:

1. Formally capture the specifics of the problem in a document based on a template
2. Define the factors/parameters for creation
 - Neural network creation parameters
 - Performance requirements
3. Create the neural network
4. Get feedback on performance

Neural Network **Development Process**



Problem Specification Phase

- Some factors to define in problem specification:
 - **Type** of neural networks (based on experience or published results)
 - How to collect and transform **problem data**
 - Potential **input/output** representations
 - **Training & testing** method and data selection
 - **Performance** targets (accuracy and precision)
- Most important output is the **ranked collection** of factors/parameters

Problem 2 – How to create a Neural Network

- **Predictability** (with regard to resources)
 - Depending on creation approach used, record **time for one iteration**
 - Use timing to predict **maximum and minimum times** for all of the combinations specified
- **Repeatability**
 - **Relevant information must be captured** in problem specification and combinations of parameters

Problem 3 - Quality Assurance

- Specification of **generic neural network software** (models and learning)
- **Prototype** of specification
- **Comparison** of a given implementation with specification prototype
- Allows practitioners to create arbitrary neural networks **verified against models**

Two Methods for Comparison

- Direct comparison of outputs:

	20-10-5 (with particular connections and input)
Prototype	<0.123892, 0.567442, 0.981194, 0.321438, 0.699115>
Implementation	<0.123892, 0.567442, 0.981194, 0.321438, 0.699115>

- Verification of weights generated by learning algorithm:

20-10-5	Iteration 100	Iteration 200	Iteration n
Prototype	Weight state 1	Weight state 2	Weight state n
Implementation	Weight state 1	Weight state 2	Weight state n

Further Work on improvements

- Practitioners to use the development process or at least document in problem specification
- Feedback from neural network development community on the content of the problem specification template
- Collect problem specifications and analyse to look for commonalities in problem domains and improve predictability (eg. control)
- More verification of specification prototype

Further Work (2)

- Translation methods for formal specification
- Extend formal specification to new types
- Fully prove aspects of the specification
- Cross discipline data analysis methods (eg. ICA, statistical analysis)
- Implementation of learning on distributed systems
 - Peer-to-peer network systems (farm each combination of parameters to a peer)
- Remain unfashionable

Summary

- Neural network is a computational model that simulate some properties of the human brain.
- The connections and nature of units determine the behavior of a neural network.
- Perceptrons are feed-forward networks that can only represent linearly separable functions.

Summary

- Given enough units, any function can be represented by Multi-layer feed-forward networks.
- Backpropagation learning works on multi-layer feed-forward networks.
- Neural Networks are widely used in developing artificial learning systems.

References

- Russel, S. and P. Norvig (1995). Artificial Intelligence - A Modern Approach. Upper Saddle River, NJ, Prentice Hall.
- Sarle, W.S., ed. (1997), *Neural Network FAQ, part 1 of 7: Introduction*, periodic posting to the Usenet newsgroup comp.ai.neural-nets,
URL: <ftp://ftp.sas.com/pub/neural/FAQ.html>

Sources

Eric Wong

Eddy Li

Martin Ho

Kitty Wong