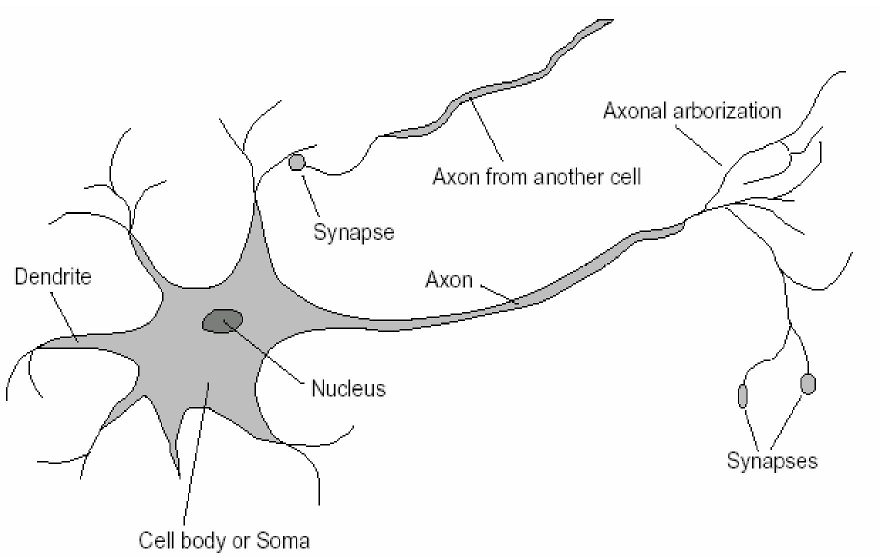
Artificial Neural Networks

Ronald J. Williams
CSG220, Spring 2007

Brains

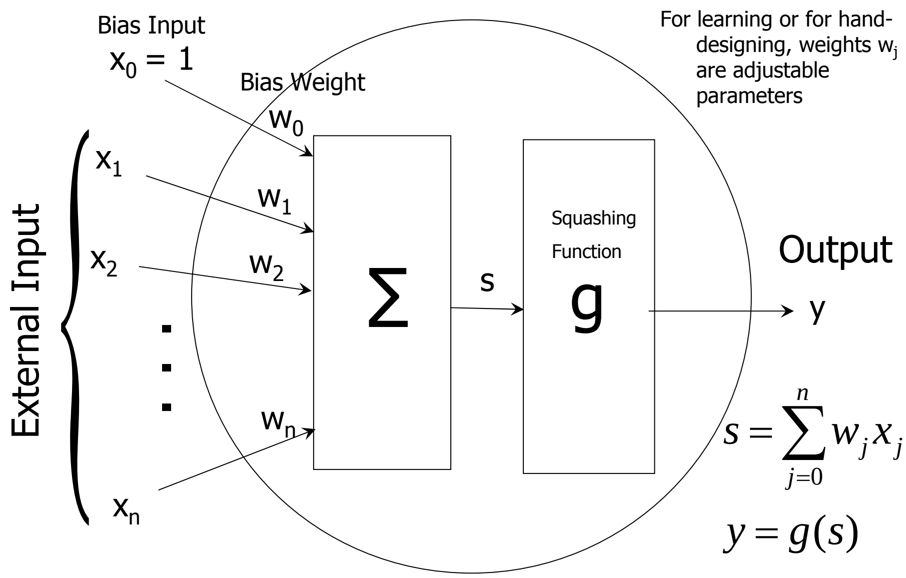
- ~10^{11} neurons of > 20 types, ~10^{14} synapses, 1-10ms cycle time
- Signals are noisy spike trains of electrical potential
- Synaptic strength believed to increase or decrease with use (⇔learning?)

A Neuron



Standard ANN "Neuron" or Unit

For learning or for hand-designing, weights w_j are adjustable parameters

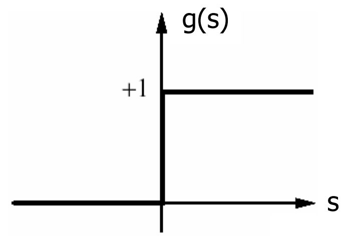


$$s = \sum_{j=0}^{n} w_j x_j$$

$$y = g(s)$$

Linear Threshold Unit
Simple Perceptron Unit
Threshold Logic Unit

Use "hard-limiting" squashing function



$$g(s) = \begin{cases} 1 & \text{if} \quad s > 0 \\ 0 & \text{if} \quad s \le 0 \end{cases}$$

Boolean interpretation: 0 ⇔ false, 1 ⇔ true

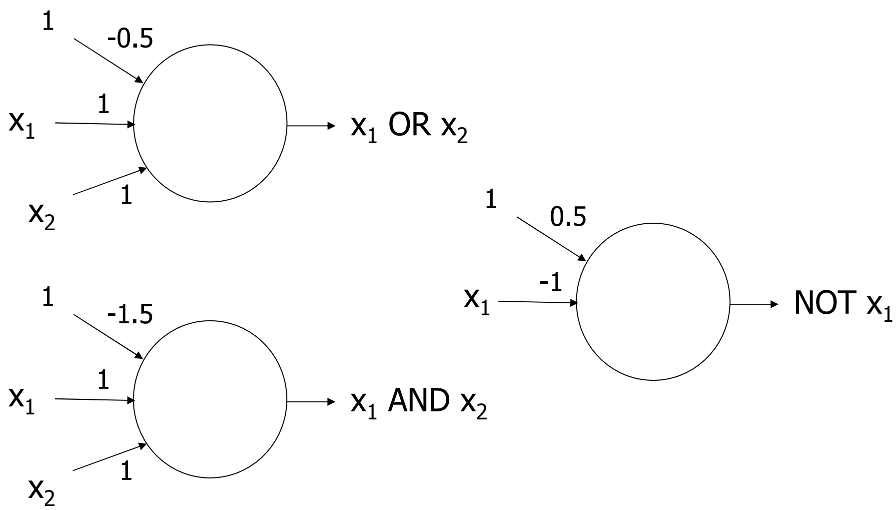
Note that

$$\sum_{j=0}^{n} w_j x_j > 0 \Leftrightarrow \sum_{j=1}^{n} w_j x_j > -w_0 x_0 = -w_0$$

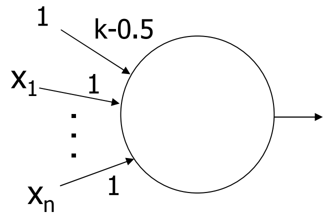
Thus an equivalent formulation is to take the appropriate weighted sum involving only the true (external) inputs and compare it against the threshold $-w_0$

The use of a bias input of 1 and a corresponding bias weight is a mathematical device to allow us to treat the threshold as just another weight

Implementing Boolean Functions



Implementing Boolean Functions (cont.)



At-least-k-out-of-n gate

Generalizes AND, OR

Challenge: Write a Boolean expression for this

Another challenge: Construct a decision tree for this

Geometric Interpretation

Define $\mathbf{x} = (x_1, x_{2,} \ldots, x_n)$

and $\mathbf{w} = (w_1, w_{2,} \ldots, w_n)$

I.e., here the bias input and bias weight are *not* included

Then the output of the unit is determined by the sign of

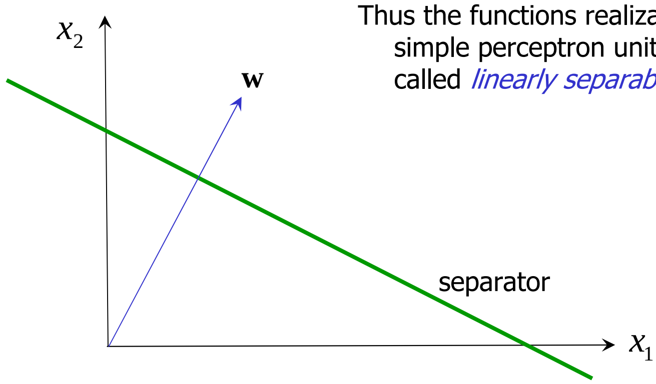
$$\sum_{j=0}^{n} w_j x_j = \mathbf{w} \cdot \mathbf{x} + w_0$$

so the separator between the y=0 and y=1 regions of the input space consists of all points **x** for which

$$\mathbf{w} \cdot \mathbf{x} + w_0 = 0$$

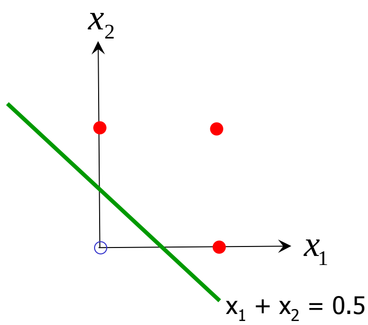
Geometric Interpretation (cont.)

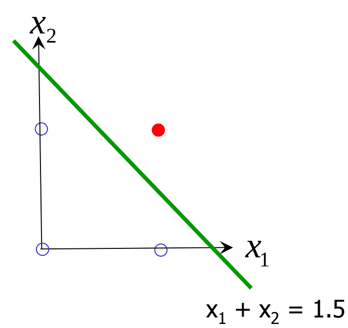
This separator is a hyperplane in n-dimensional space with normal vector $\mathbf{w}$ and whose distance to the origin is $|w_0|/\|\mathbf{w}\|$



Thus the functions realizable by a simple perceptron unit are called *linearly separable*

Boolean examples





x_1 OR x_2

$w_1 = 1$

$w_2 = 1$

$w_0 = -0.5$

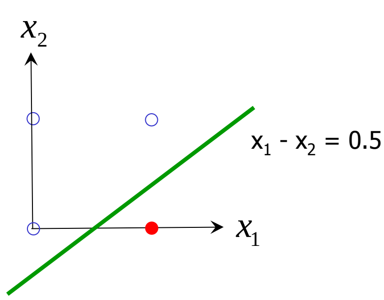
x_1 AND x_2

$w_1 = 1$

$w_2 = 1$

$w_0 = -1.5$

Boolean examples (cont.)



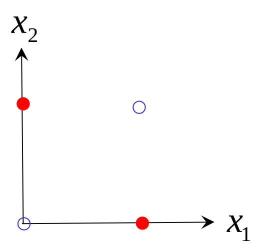
x_1 AND NOT x_2

$w_1 = 1$

$w_2 = -1$

$w_0 = -0.5$

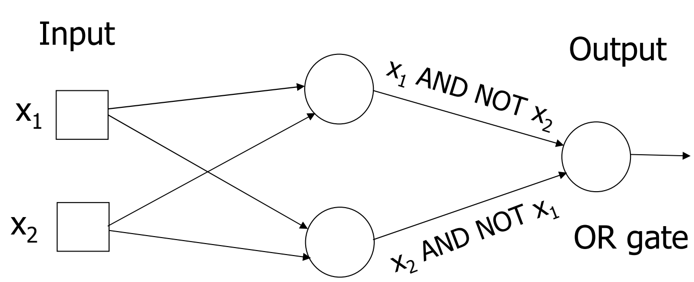
But ...



x_1 XOR x_2

- Not linearly separable
- XOR and its negation are the only Boolean functions of two arguments that are not linearly separable
- However, for larger and larger n, the number of linearly separable Boolean functions grows much more slowly than the number of possible Boolean functions

Implementing XOR with simple perceptron units



- Suffices to use one intermediate stage of simple perceptron units
- Approach generalizes to any Boolean function: write it in DNF, use one intermediate unit for each disjunct, then use an OR gate for output
- Proves that any Boolean function is realizable by a network of simple perceptron units

What about learning?

- Start with training data $\{(\mathbf{x}^r, d^r)\}$, where each input/desired output pair is indexed by r = 1, ..., R and $\mathbf{x}^r = (1, x_1^r, x_2^r, \ldots, x_n^r)$ represents the input (this time augmented by the bias input $x_0^r = 1$)
- Each d^r is of course either 0 or 1
- The objective is to find a weight vector $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$ such that $y^r = g(\mathbf{w} \cdot \mathbf{x}^r)$ agrees with d^r for each r, where g is the hard-limiting threshold function

Perceptron algorithm

$\mathbf{w} \leftarrow \mathbf{0}$ (any initial values ok)
repeat
  for r=1 to R
    $\mathbf{w} \leftarrow \mathbf{w} + \eta(d^r - y^r)\mathbf{x}^r$
until no errors

$\eta > 0$ is the learning rate
It can be taken to be 1 when inputs are 0 and 1
In that case, body of inner loop is:

- if actual output too small, add input vector to weight vector
- if actual output too large, subtract input vector from weight vector
- else don't change weights

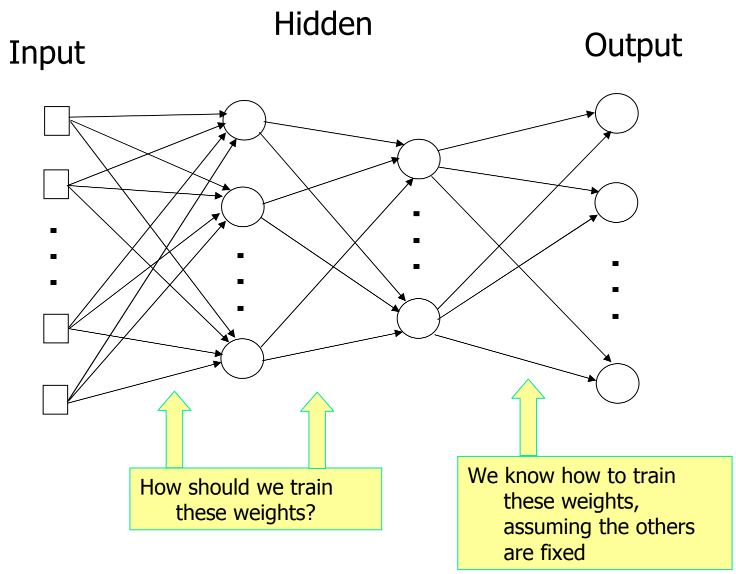
Perceptron algorithm (cont.)

- Easy to check that this moves weights greedily in correct direction for the current training example
- Convergence theorem: For any linearly separable training data, the algorithm converges to a solution (as long as the learning rate is suitably small). But if the data is not linearly separable, the weights loop indefinitely.

Multilayer Networks

- This algorithm has been known since ~1960 (Rosenblatt)
- But the most interesting functions we might want to learn are not necessarily linearly separable
- Dilemma faced by ANN researchers between ~1960 and ~1985:
  - for greater expressiveness, need multilayer networks of these linear threshold units
  - only known reasonable algorithm was for single-layer networks (i.e., one layer of weights)

Multilayer Networks (cont.)



Learning in multilayer nets – basic idea

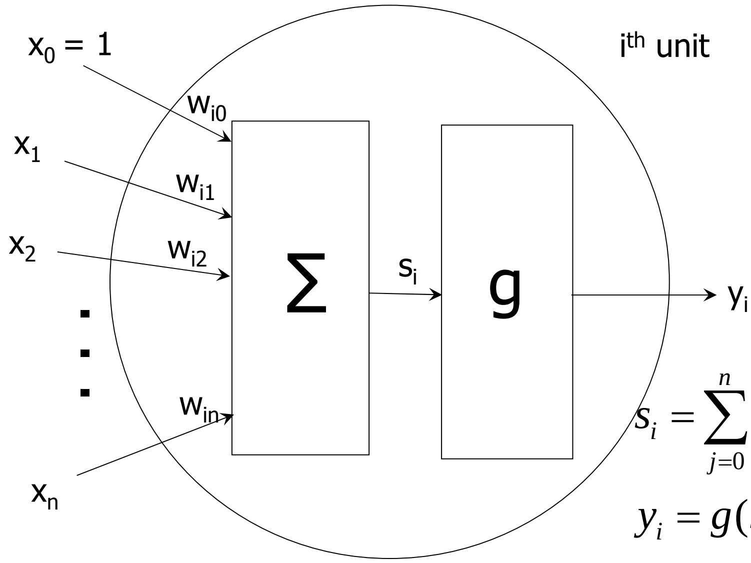
One general way to approach any learning problem:

- express the learning objective in terms of a function to optimize
- search the hypothesis space for a hypothesis giving the optimal value

Applied to a supervised learning task:

- for each possible hypothesis, define a measure of its overall error on the training data
- simplest way: define this error measure for each training example and then define the overall error measure as the sum of these

Expanded notation: necessary since using multiple units



$$s_i = \sum_{j=0}^{n} w_{ij} x_j$$

$$y_i = g(s_i)$$

Learning in multilayer nets

Define the error on the r^{th} training example to be

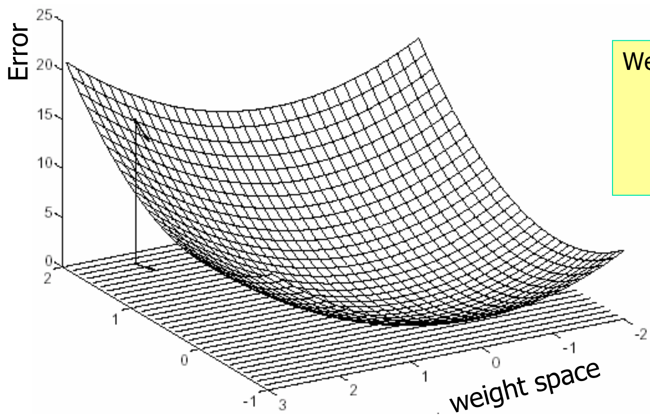
$$E^r = \frac{1}{2} \sum_{i \in \text{OutputUnits}} (d_i^r - y_i^r)^2$$

where d_i^r and y_i^r are the desired and actual outputs, respectively, of the i^{th} unit for training example r.
This is a function of the network weights since y_i^r is.

Then define the overall error to be

$$E = \sum_r E^r$$

Gradient Descent



Weight space is N-dimensional, where N is the total number of weights in the network

Gradient $\nabla_{\mathbf{w}} E$ is a vector whose α^{th} component is $\frac{\partial E}{\partial w_\alpha}$, where w_α is a weight in the network.

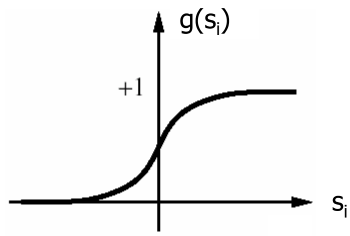
Gradient descent: increment each w_α by $\Delta w_\alpha = -\eta \frac{\partial E}{\partial w_\alpha}$

Oh, oh ..., a problem

- For a network of linear threshold units, the gradient is zero everywhere it exists (which is almost everywhere)
- The error function has a "terrace" shape – flat everywhere with occasional "cliffs"
- So gradient descent useless in this case
- Now introduce a trick ...

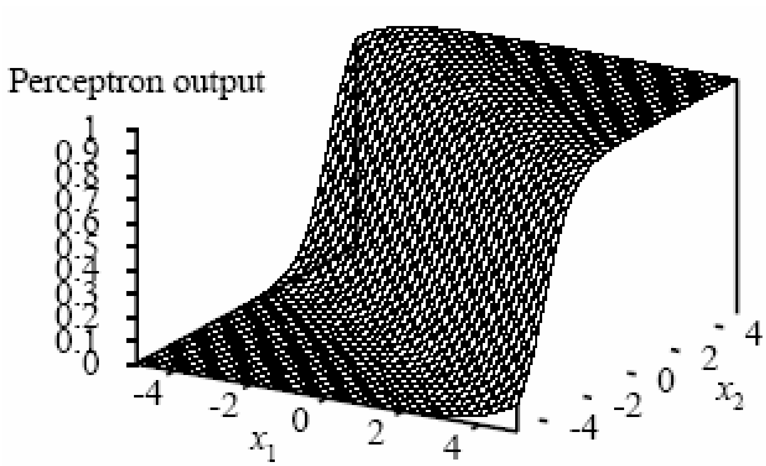
Sigmoid squashing function

Instead of the hard-limiting threshold function of the simple perceptron unit, use a smooth approximation to it



Commonly used: $$g(s) = \frac{1}{1+e^{-s}}$$ Logistic function

"Soft" linear separation



For any network of such sigmoid units, the network output is a smooth function of its input.

Thus so is the error function.

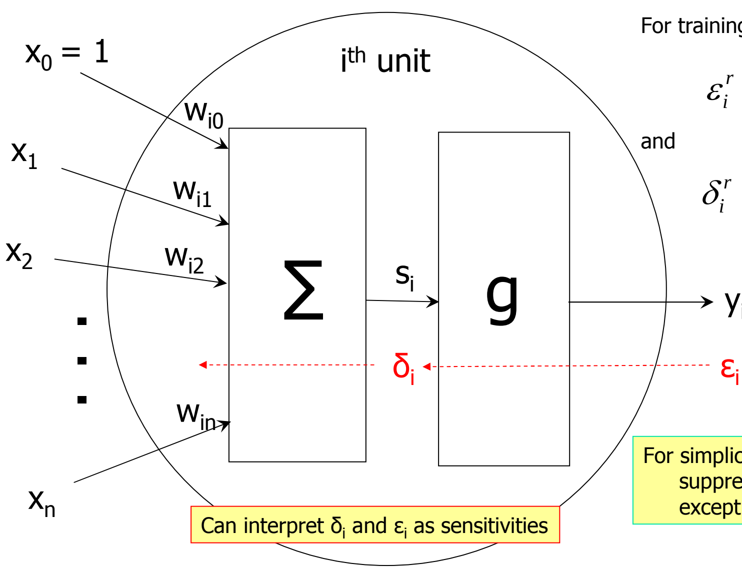
But how do we compute the necessary gradient?

It would be painful to write down an explicit expression for the network output (or the error) as a function of the network input and the weights.

Then imagine trying to differentiate it.

To the rescue: the chain rule

The error backpropagation algorithm



For training example r, define

$$\varepsilon_i^r = -\frac{\partial E^r}{\partial y_i}$$

and

$$\delta_i^r = -\frac{\partial E^r}{\partial s_i}$$

For simplicity, we henceforth suppress the superscript r except on E^r

Derivation of backprop

Since

$$E = \sum_r E^r$$

it follows that

$$\frac{\partial E}{\partial w_{ij}} = \sum_r \frac{\partial E^r}{\partial w_{ij}}$$

for any weight w_{ij}.

Now we focus on how to compute $-\frac{\partial E^r}{\partial w_{ij}}$.

Derivation of backprop (cont.)

Since $s_i = \sum_j w_{ij} x_j$

we see that

$$-\frac{\partial E^r}{\partial w_{ij}} = -\frac{\partial E^r}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}} = \delta_i x_j$$

Furthermore,

$$\delta_i = -\frac{\partial E^r}{\partial s_i} = -\frac{\partial E^r}{\partial y_i}\frac{dy_i}{ds_i} = \varepsilon_i g'(s_i)$$

so all that remains is to compute ε_i for any unit i.

Derivation of backprop (cont.)

For each output unit i,

$$\varepsilon_i = -\frac{\partial E^r}{\partial y_i} = -\frac{\partial}{\partial y_i}\left[\frac{1}{2}\sum_{k \in \text{OutputUnits}} (d_k - y_k)^2\right] = d_i - y_i$$

What about hidden units?

For each hidden unit i, let Downstream(i) = all units to which that unit directly sends its output.

Note that from the point of view of each unit k in Downstream(i), the output y_i of unit i is the input x_i of unit k (i.e, the signal on the input with weight w_{ki}).

Derivation of backprop (cont.)

Thus for hidden unit i,

$$\varepsilon_i = -\frac{\partial E^r}{\partial y_i} = -\sum_{k \in \text{Downstream}(i)} \frac{\partial E^r}{\partial s_k}\frac{\partial s_k}{\partial y_i} = \sum_{k \in \text{Downstream}(i)} \delta_k \; w_{ki}$$

using the fact that $s_k = \sum_j w_{kj} x_j$

so

$$\frac{\partial s_k}{\partial y_i} = \frac{\partial s_k}{\partial x_i} = w_{ki}$$

Backprop summary

- This gives a recursive formulation of how all the relevant intermediate quantities are computed.
- To do the computation iteratively, start at the output units, computing the appropriate ε and δ values there, then proceed through the network backwards until all units have the necessary δ values.
- It is more common to formulate this without explicitly identifying ε, although doing it our way more clearly demonstrates the general stage-wise organization of this computation.
- Here is the more common δ-only formulation of backprop:

Backprop algorithm – single step

Basic single forward/backward computation for a given input/desired output pair:

1. Place the input vector at the input nodes and propagate forward
2. At each output node i, compute $\delta_i = g'(s_i)(d_i - y_i)$
3. At each hidden node i, compute

$$\delta_i = g'(s_i) \sum_{k \in \text{Downstream}(i)} w_{ki}\delta_k$$

4. For each weight w_{ij} compute $\delta_i x_j$

Derivative of squashing function

- If the squashing function is the logistic function

$$g(s_i) = \frac{1}{1+e^{-s_i}}$$

the derivative has the convenient form

$$g'(s_i) = g(s_i)(1-g(s_i)) = y_i(1-y_i)$$

Exercise:
Prove this

- Another popular choice of squashing function is tanh, which takes values in the range (-1,1) rather than (0,1)
  - requires plugging a different g' into the algorithm

The full backprop algorithm

Initialize weights to small random values

Repeat until satisfied

For each training example r

Do one forward and backward pass to compute $\delta_i^r x_j^r$ for each adjustable weight w_{ij}

Batch version: accumulate these values over the training set, then do

$$w_{ij} \leftarrow w_{ij} + \eta \sum_r \delta_i^r x_j^r$$

Incremental version: inside inner loop do

$$w_{ij} \leftarrow w_{ij} + \eta \delta_i^r x_j^r$$

Remarks

- Batch version represents true gradient descent
- Incremental version only an approximation, but often converges faster in practice
- Many variations:
  - Momentum – essentially smooths successive weight changes
  - Different values of η for different units, or as function of time, or adapted based on still other considerations
  - Use of second-order techniques or approximations to them
- Drawbacks
  - May take many iterations to converge
  - May converge to suboptimal local minima
  - Learned network may be hard to interpret in human-understandable terms

Remarks (cont.)

- Gradient-based "credit assignment"
  - make changes to all parameters where such changes would contribute some beneficial effect
  - size of change proportional to sensitivity – make larger changes to parameters to which beneficial outcome most sensitive

Linear units

- Sometimes useful to take g = identity function, i.e., no squashing
- Appropriate for output units if the range of the function to be learned not bounded
- But if all units are linear, multilayer networks are no more expressive than single-layer networks

Can you see why?

- In a single-layer network of linear units, backprop also known as LMS or Widrow-Hoff rule
  - widely used in adaptive control, signal-processing, etc.

Practical considerations

- Useful squashing functions only approach their extreme values asymptotically
- E.g., logistic function can never actually attain values of 0 or 1
- With such output units, training to unattainable output values would never terminate
- Instead, in practice use either
  - a dead zone: e.g., train to targets of 0 and 1 but consider any output within a tolerance of, say, 0.1 to be correct
  - targets of, say, 0.1 and 0.9 in place of 0 and 1, respectively

Neural net representations

- Have to encode all possible input and output as Euclidean vectors
- What if input or output is discrete (e.g., symbolic)?
- If exactly two possible values, one natural encoding would be to use 0 for one of these and 1 for the other
- Alternative encoding that works for any finite number of values: use a separate node for each value and set exactly one node to 1 and all others to 0
  - called 1-out-of-n or radio button encoding
- But if the values have a natural topology (e.g., fall on an ordinal scale), might make sense to use an encoding that captures this

Representation example

- Consider Outlook = Sunny, Overcast, or Rain
- 1-out-of-3 encoding:
  - Sunny ⇔ 1 0 0
  - Overcast ⇔ 0 1 0
  - Rain ⇔ 0 0 1

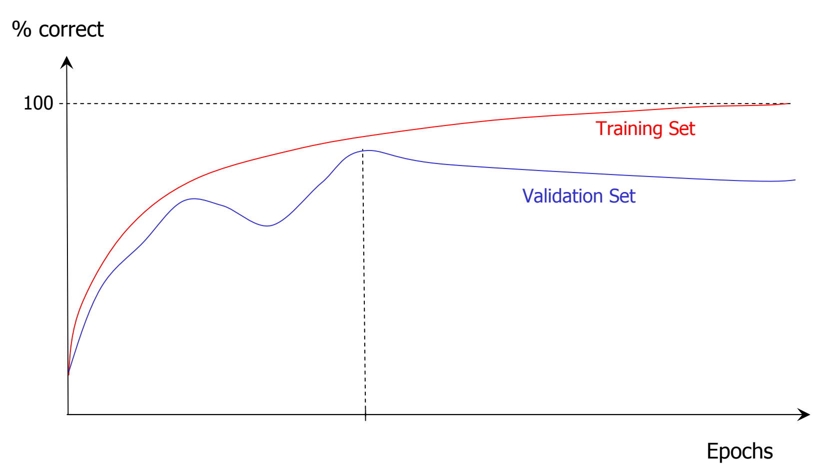
  Uses 3 input nodes
- Treating Overcast as halfway between Sunny and Rain:
  - Sunny ⇔ 0.0
  - Overcast ⇔ 0.5
  - Rain ⇔ 1.0

  Uses 1 input node
- Such choices help determine the underlying inductive bias

Other considerations

- Avoiding overfitting
  - early stopping
  - explicit penalty terms
  - weight decay
- Incorporating prior knowledge
  - enforcing invariances through "weight sharing"
  - limiting connectivity
  - letting some of the input represent more complex precomputed features
  - initializing the network according to a best guess, then letting backprop fine-tune the weights
  - setting some weights by hand and keeping them fixed

Avoiding overfitting by early stopping



Expressiveness

- Any continuous function can be approximated arbitrary closely over a bounded region by a two-layer network with sigmoid squashing functions in the hidden layer and linear units in the output layer (given enough hidden units)

Inductive bias

- When weights are close to zero, behavior is approximately linear
- Keeping weights near zero gives a preference bias toward linear functions

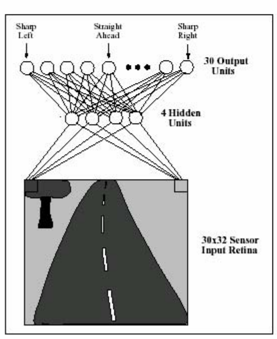
Wide variety of applications

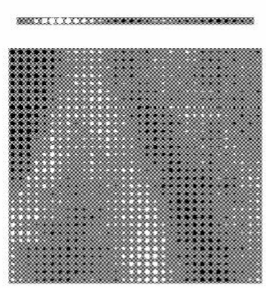
- Speech recognition
- Autonomous driving
- Handwritten digit recognition
- Credit approval
  - But may be hard to translate network behavior into more explicit, easily-understood rules
- Backgammon
- Etc.

Generally appropriate for problems where the final answer depends heavily on combinations of many input features

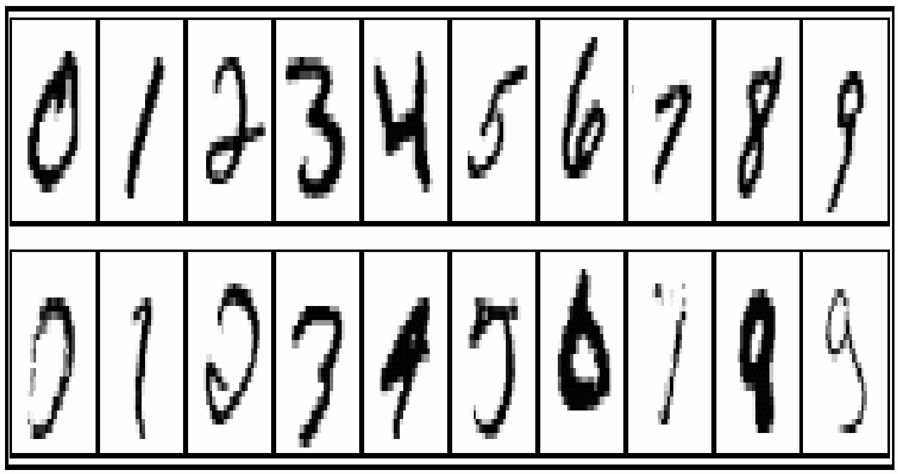
Decision trees might be better when decisions depend on only a small subset of the input features

ALVINN: autonomous vehicle





Handwritten digit recognition



Accuracy on test digits

- 3-nearest-neighbor → 2.4% error
- 400-300-10 unit MLP → 1.6% error
- LeNet: 768-192-30-10 unit MLP → 0.9%
  - limited connectivity to enforce locality constraints
  - weight sharing to create translation-invariant features (learned)

Extension: Recurrent networks

- With feedback connections, artificial neural networks can exhibit interesting temporal behaviors
  - oscillations
  - convergence to fixed points
  - approximate finite-state machine behavior
- An extension of backprop (*backprop-through-time*) can be used to train these behaviors

Learning to be a FSM

Consider the following input/output behavior

time →

Input	C	B	D	A	D	B	C	B	B	A	C	A	D	B	C	D	. . .
Output	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	. . .

Learning to be a FSM

Consider the following input/output behavior

time →

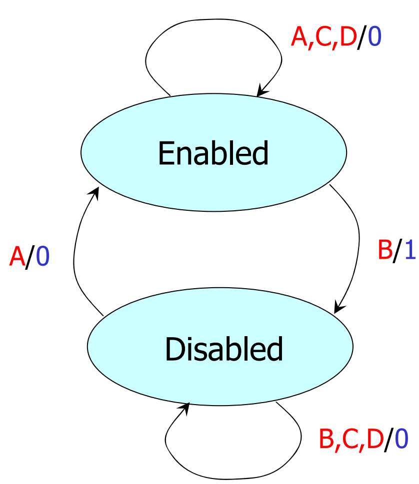
Input	C	B	D	A	D	B	C	B	B	A	C	A	D	B	C	D	. . .
Output	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	. . .

```
If input == A
    enabled ← true
Else If input == B
    If enabled
        output ← 1
        enabled ← false
    Else output ← 0
Else output ← 0
```

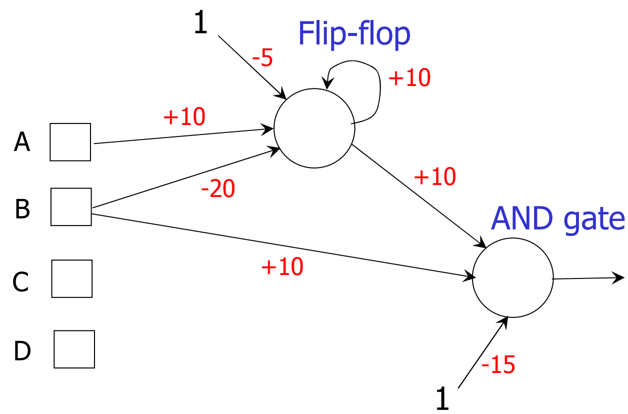
"Bus driver problem"

Finite state machine

Mealy Machine



Neural net implementation



- Weight values appropriate for standard logistic squashing function
- 1-step delays not explicitly shown
- Gradient descent can learn this from a stream of I/O examples

Summary

- Most brains have lots of neurons, so maybe the kinds of computing that brains are good at are best accomplished by large networks of simple computing units (linear threshold units?)
- One-layer networks insufficiently expressive
- Multilayer networks are sufficiently expressive and can be trained by gradient descent, i.e., error backpropagation
- Some general-purpose ways to look at learning
  - Formulation as an optimization problem
  - Gradient search when appropriate
- Various techniques for incorporating prior knowledge and for avoiding overfitting
- Many applications
- Even some temporal behaviors can be trained by backpropagation-like gradient descent algorithms