Artificial Neural Networks *

October 13, 2015

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations

Connectionist Models

Artificial Neural Networks (ANN/NN) : general framework for learning a function (real-valued, discrete-valued, vector-valued) from examples.

Initially, the field developed from the analogy with biological neural networsk

Biological motivation

In humans, biological neural networks:

- Neuron switching time ˜ .001 second
- Number of neurons ˜ 10^{10}
- Connections per neuron ˜ 10^{4-5}
- Scene recognition time ˜ .1 second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Although analogy with biological NN provided the original impetus for ANN, the field developed independently of these, just as abstract models of (parallel) computation.

Efforts along the biological significance of NN models remain and are now a very strong component of computational neuro-science.

Properties of artificial neural nets (ANN's):

*based on Mitchel's book

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

The last point is **very important**: the weights in the ANN capture the knowledge captured by the network. However, this type of knowledge is NOT for "human consumption". In other words it is difficult for people to understand exactly what how/what the network has learned, which are the important attributes, express the concept in some linguistic description.

In recent years significant effort has been going on in extracting rules from the NN, but such steps are **in addition** to the NN and not part of it! NN tend to be **black boxes**: in goes the example, out comes its corresponding output.

NN have been used successfully in

- scene recognition and understanding;
- speech recognition;
- various classification tasks in high dimensional domains (e.g. financial, stock market prediction).

Examples include:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

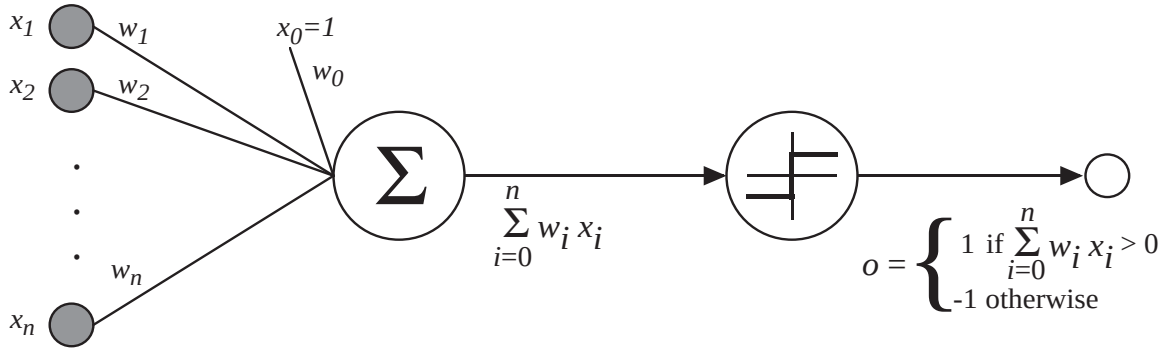


Figure 1: A single perceptron

The perceptron

The first model of ANN was the perceptron (Rosenblatt 1959, The perceptron: A probabilistic model for information storage and organization in the brain, *Psychological Review*, 65, 386-408.) The history of the perceptron is very interesting! Its first model is actually displayed at the Smithoninan!

The perceptron takes a vector-valued input, calculates a linear combination of these and outputs a 1 if the result is greater than some threshold or -1 otherwise (see figure 1).

Given the inputs $x_1, \ldots, x_n$,

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases} \quad (1)$$

In vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases} \quad (2)$$

In terms of our previously used vocabulary: the hypothesis space for the perceptron is

$$H = \{\vec{w} | \vec{w} \in \Re^{(n+1)}\}$$

Decision Surface of a Perceptron

The perceptron can represent many useful functions: logical **AND, OR, NAND(¬ AND), NOR (¬ OR)**. This means that, in fact, the perceptron (alone, or with several levels) can represent **every** boolean function, because any such function can be represented in terms of the boolean primitives **AND, OR, NOT, NAND, NOR**.

For example,

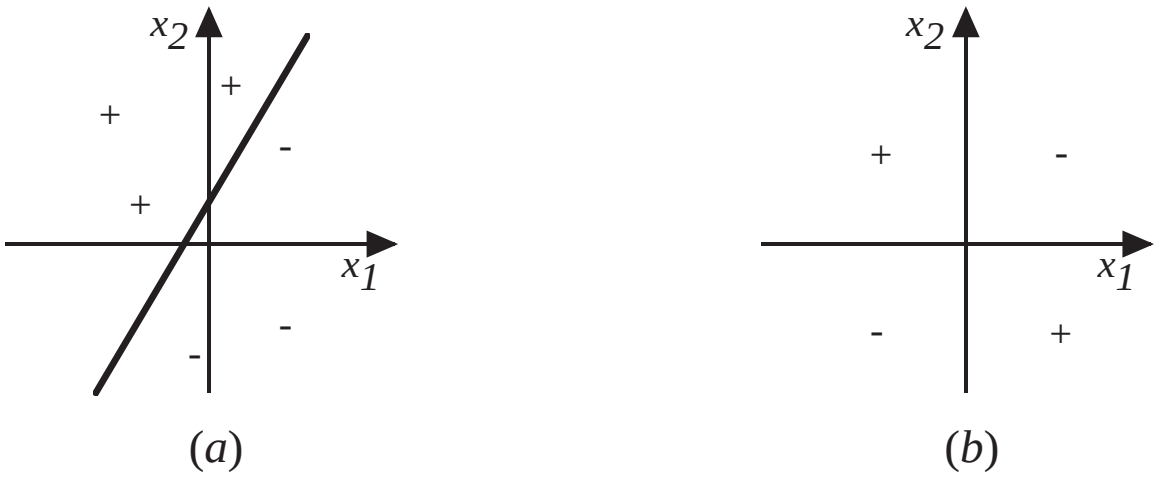


Figure 2: The decision surface for the perceptron

- a two-input, one-output perceptron, with the weights $w_0 = -0.8, w_1 = w_2 = 0.5$ represent $g(x_1, x_2) = AND(x_1, x_2)$
- the same perceptron can represent $OR(x_1, x_2)$ if the weight $w_0 = -0.3$

However, some functions are not representable: for example the logical **XOR**. In fact, we have the following result:

Fact 1 *If the training set for the perceptron is linearly separable, then a learning procedure based on the perceptron training rule will converge in a finite number of steps, in the sense that the perceptron will correctly classify all the examples. When the training set is not linearly separable, a network of perceptrons is necessary.*

Perceptron training rule

The idea behind the perceptron training rule is to devise a procedure such that when a positive example (true output is +1) is miss-classified, the weights must be altered in order to increase the value of $\vec{w} \cdot \vec{x}$:

- if the component x_i of $\vec{x} > 0$ then w_i, the corresponding component of w must be increased as well;
- if $x_i < 0$, w_i must be decreased.

This behavior is captured by the following rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output: $o = sgn(\vec{w} \cdot \vec{x})$
- η is small constant (e.g., 0.1) called *learning rate*

One can prove that the perceptron rule will converge

- If training data is linearly separable
- and η sufficiently small

The files `per-rule.m, toydatax.mat, toydatay.mat` on Blackboard contain Matlab code for the perceptron, and some small training sets. Also the file `heart.txt` contains a more realistic data set.

Gradient Descent

To overcome the limitation of the perceptron rule for data which are not linearly separable other rules can be designed:

The gradient descent rule is derived from the minimization a measure of the training error. More precisely, to understand, consider simpler *linear unit*, where

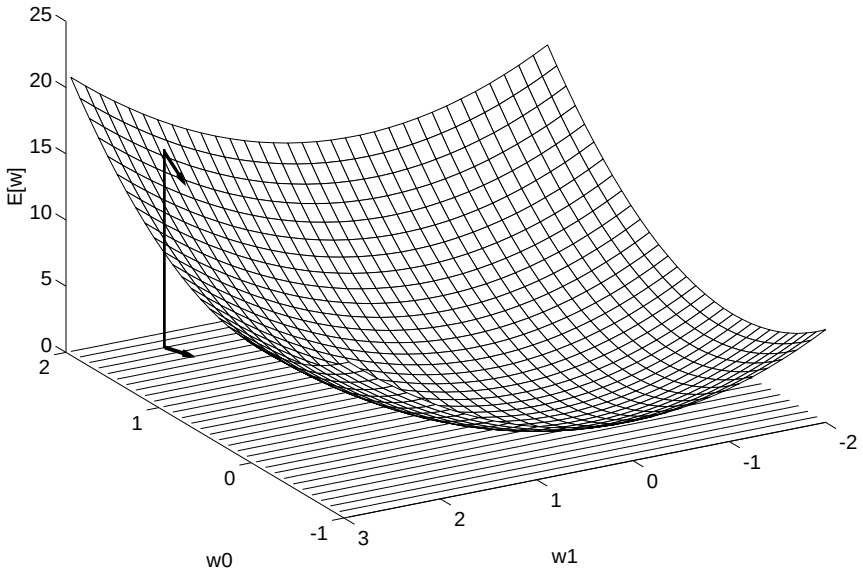
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is set of training examples.

Note that since o is a function of $\vec{w}$ the error, E is a function of $\vec{w}$.



To derive the gradient descent rule we need to consider the partial derivatives of E with respect to the components of $\vec{w}$, or the gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

Question: Why is the gradient important?

Answer: Because *when considered as a vector in the weight space, the gradient specifies the direction that produces the steepest (fastest) increase in E.*

This means that its negative gives the direction of the steepest decrease!

Therefore, for the steepest decrease, the training rule must be

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}]$$

or, by components,

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{3}$$

Then, in principle, the weight update rule is

$$\vec{w} \leftarrow \vec{w} + \nabla E(\vec{w})$$

To construct a practical algorithm we must carry out the computation of $\frac{\partial E}{\partial w_i}$ as follows:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 take\ \partial\ under\ sum \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d})
\end{aligned} \tag{4}$$

Therefore,

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d}) \tag{5}$$

where $x_{i,d}$ is the ith component of the training example $\vec{x_d}$.

Substituting (5) into (3) we obtain **the weight update rule for the gradient descent**:

$$\nabla w_i = \eta \sum_{d\in D} (t_d - o_d)x_{i,d} \tag{6}$$

Gradient Descent Algorithm

GRADIENT-DESCENT($training\_examples, \eta$)

> *Each training example is a pair of the form $\langle \vec{x}, t\rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t\rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Summary

The perceptron training rule guaranteed to succeed if

- The training examples are linearly separable;
- The learning rate η is sufficiently small.

By contrast, the linear unit training rule that uses gradient descent is:

- guaranteed to converge to hypothesis with minimum squared error
- given sufficiently small learning rate η
- even when training data contains noise, and
- even when training data not separable by H

Incremental (Stochastic) Gradient Descent

Idea:

Approximate the gradient descent by updating the weights **incrementally**, after calculating the error for **each** individual example.

Therefore, instead of a **batch mode** Gradient Descent in which the steps are:

`Do until satisfied`

1. `Compute the gradient` $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

---

use an **incremental mode** Gradient Descent with steps:

`Do until satisfied`

- `For each training example` d in D
  1. `Compute the gradient` $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_{\mathbf{D}}[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \text{ is replaced by } E_{\mathbf{d}}[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

We have athe following result:

Fact 2 Incremental Gradient Descent *can approximate* Batch Gradient Descent *arbitrarily closely if* η *(the learning rate) is made small enough.*

The Incremental Gradient Descent algorithm is then similar to the gradient descent except that we take $\nabla E_d(\vec{w})$ instead of $\nabla E_D(\vec{w})$ to obtain:

Incremental Gradient Descent Algorithm

INCREMENTAL-GRADIENT-DESCENT($training\_examples, \eta$)

> *Each training example is a pair of the form* $\langle \vec{x}, t \rangle$*, where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- `Initialize each` w_i `to some small random value`
- `Until the termination condition is met, Do`
  1. `Initialize each` Δw_i `to zero.`
  2. `For each` $\langle \vec{x}, t \rangle$ `in` *training_examples*, `Do`
     (a) `Input the instance` $\vec{x}$ `to the unit and compute the output` $o = \vec{w} \cdot \vec{x}$
     (b) `For each linear unit weight` w_i, `Do`

$$w_i \leftarrow w_i + \eta(t - o)x_i \tag{7}$$

Gradient Descent vs. Incremental Stochastic Gradient Descent

- Error Summation
  - In Gradient Descent the error is summed over all examples before updating the weights;
  - In stochastic Incremental Gradient Descent weights are updated by **each** example;
- **Computational aspects.** The computations required depend on the learning rate η and the computation required by each update step (basically, η determines the number of steps)
  - In the Gradient Descent summing over all examples at each step requires more computation per step, but because it actually uses the true gradient, η needs not be very small;
- **Local minimum.**

– Stochastic Gradient Descent can avoid the local minimum for $\nabla E(\vec{w})$ by using $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$.

The equation

$$\Delta w_i = \eta(t - o)x_i$$

is known as the *delta rule*, or LMS (least mean squares), Adaline, Widrow-Hoff.

It looks similar to the perceptron rule, and indeed it is, with the difference that for the perceptron,

$$o = sgn(\vec{w} \cdot \vec{x})$$

while for the gradient descent is

$$o = \vec{w} \cdot \vec{x}$$

The *delta* rule can also be used to train a thresholded perceptron:
Let

$$o = \vec{w} \cdot \vec{x}$$

and

$$o' = sgn(\vec{w} \cdot \vec{x})$$

To train a perceptron to fit target values ± 1 for o' the same target values can be used for o and the *delta* rule. Since $o' = \pm 1$ then of course $o = sgn(o') = \pm 1$ and $o = 1$ *or* -1 exactly for the values for which $o' = 1$, *or* -1.

However, the weights which minimize the error in o will not necessarily minimize the error in o'.

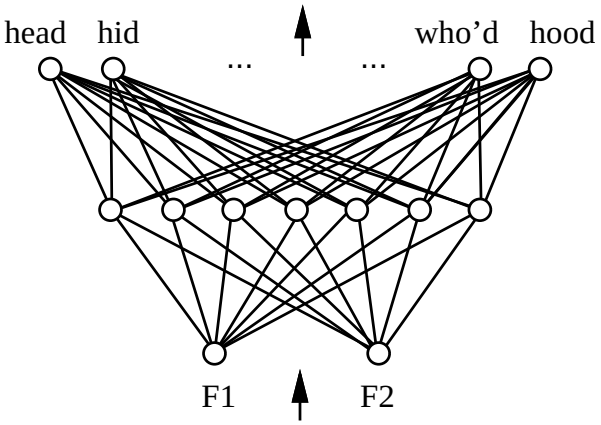
The Perceptron Rule vs. The Delta Rule

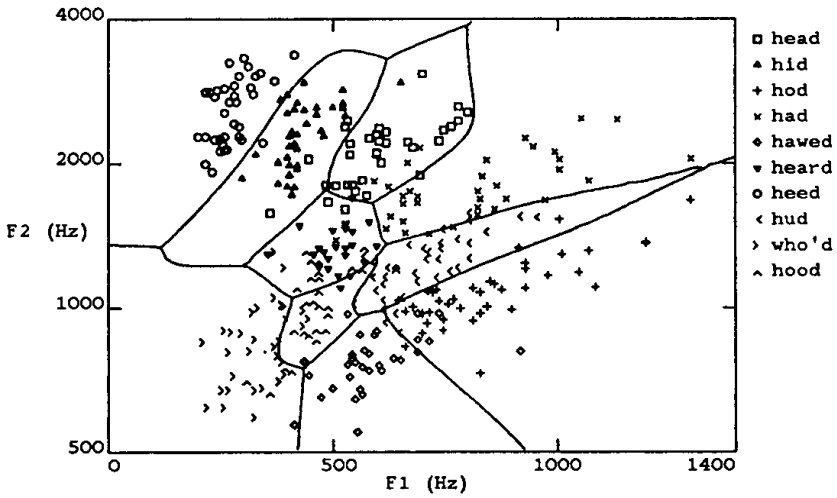
1. **Updating**
   (a) The perceptron rule updates weights based on the error in the thresholded perceptron output;
   (b) The *delta rule* updates weights based on the true magnitude of the perceptron output (unthresholded).
2. **Convergence properties**
   (a) For a set of linearly separable data, and small learning rate, the **perceptron rule converges after a finite number of steps** to a hypothesis that perfectly classifes the training data.
   (b) The delta rule converges **asymptotically**, that is it can require an unbounded number of steps, to the hypothesis with minimum error, andit does this **regardless of whether the training data is linearly separable or not**.

Other algorithms for solving this type of problems

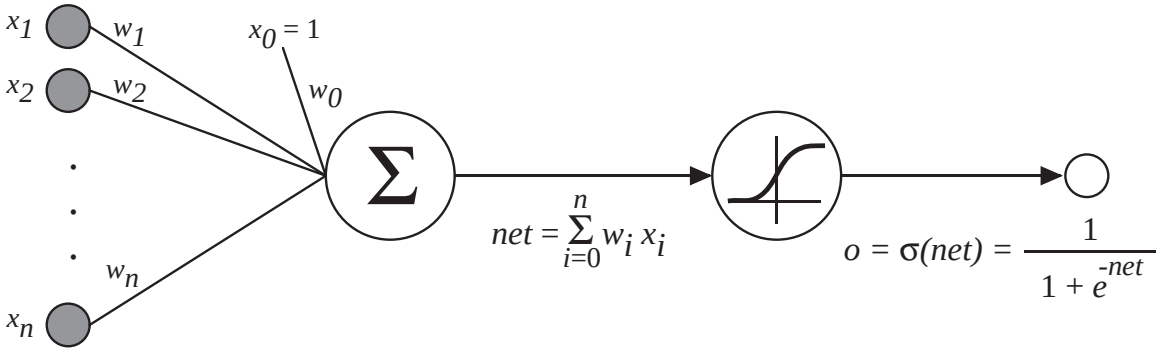
- **Linear programming.** Simultaneous inequalities ($\vec{w} \cdot \vec{x} > 0$ for positive examples, $\vec{w} \cdot \vec{x} \leq 0$ for negative examples.
- **Support vector machines (SVM).** Simultaneous inequalities with max. generalization (for linearly separable data).

Multilayer Networks of Sigmoid Units





Sigmoid Unit



$\sigma(x)$ is the sigmoid function

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

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$
We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units to obtain **Backpropagation**

Error Gradient for a Sigmoid Unit

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\ \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\ \frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_d (t_d - o_d)\ \frac{\partial o_d}{\partial net_d}\ \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1-o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

Backpropagation Algorithm

1. `Initialize all weights to small random numbers`

   `Until satisfied, Do`

   (a) `For each training example, Do`

      i. `Input the training example to the network and compute the network outputs`

      ii. `For each output unit` k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

      iii. `For each hidden unit` h

$$\delta_h \leftarrow o_h(1 - o_h)\sum_{k \in outputs} w_{h,k}\delta_k$$

      iv. `Update each network weight` $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

   `where`

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

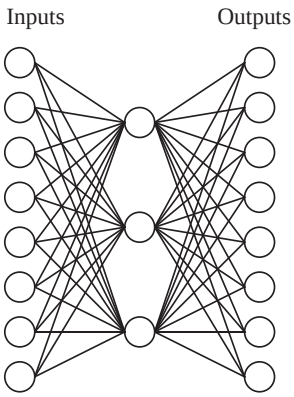
More on Backpropagation

- 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!
- HOWEVER, using network after training is very fast

Learning Hidden Layer Representations



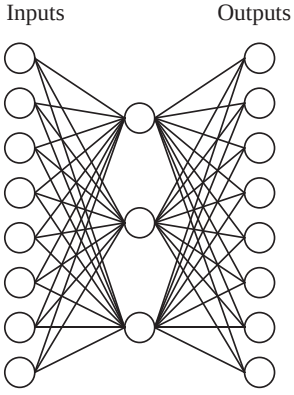
A target function:

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??

Learning Hidden Layer Representations

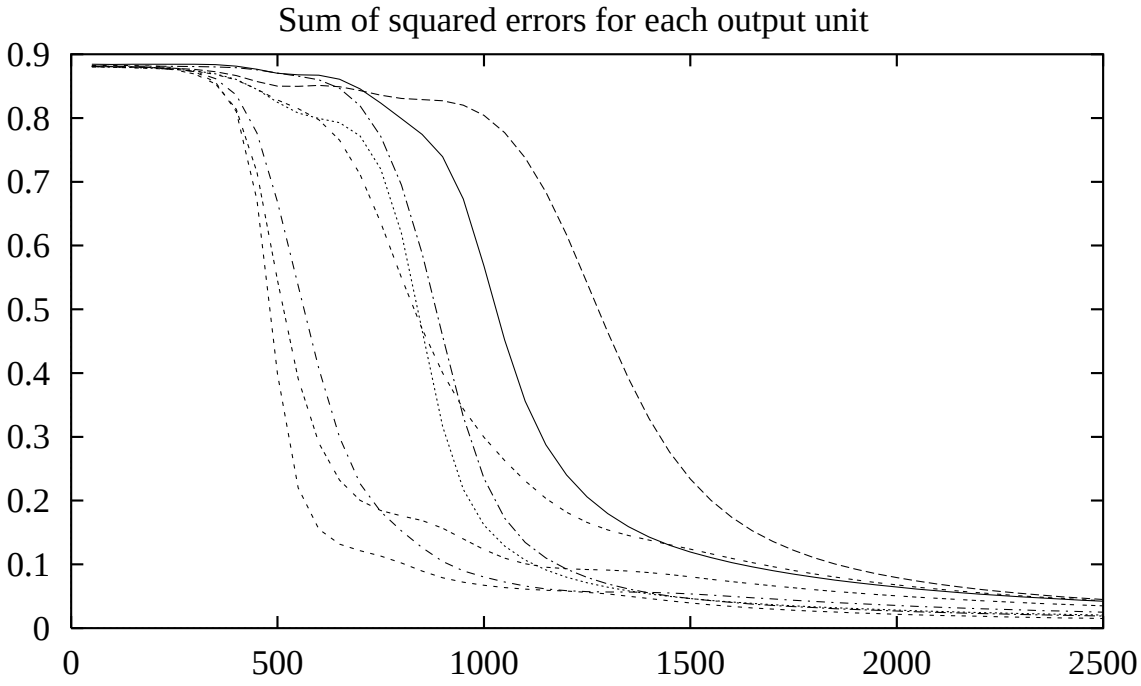
A network:



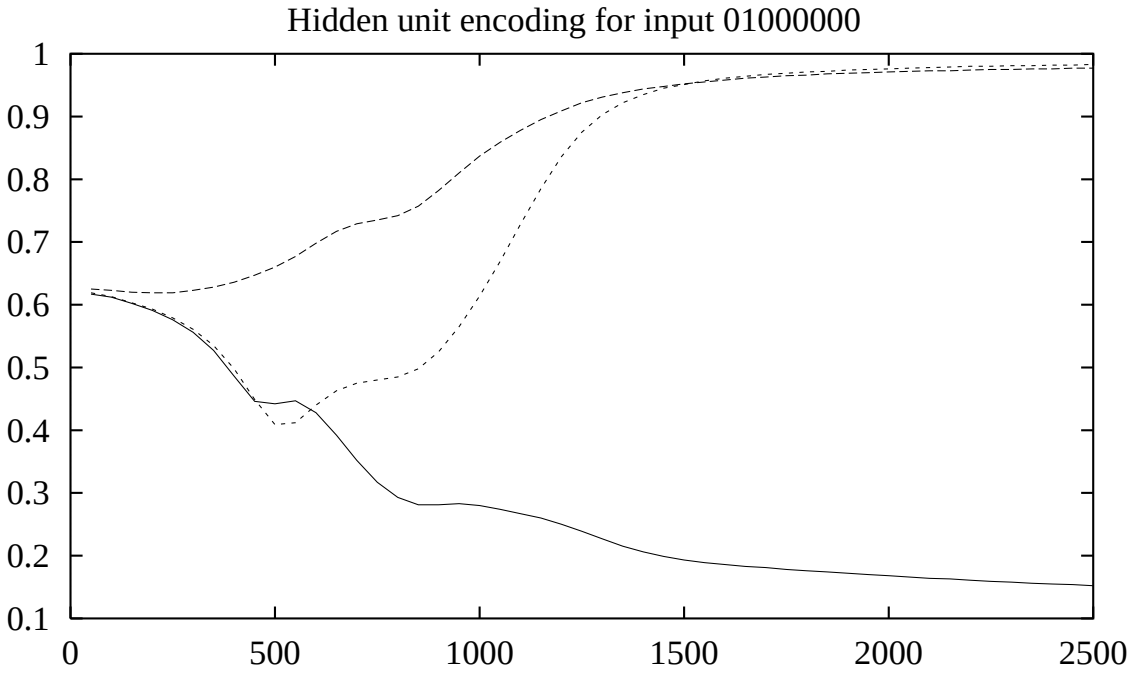
Learned hidden layer representation:

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

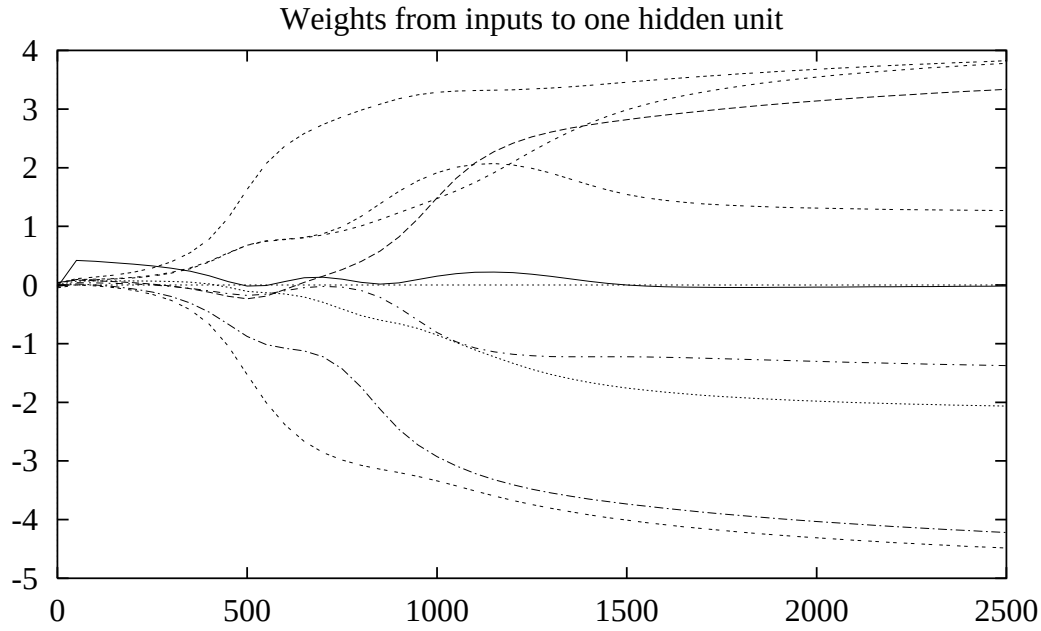
Training: ann - output - errors



Training - ann - encoding



Training ann - HiddenUnits - weights



Convergence of Backpropagation

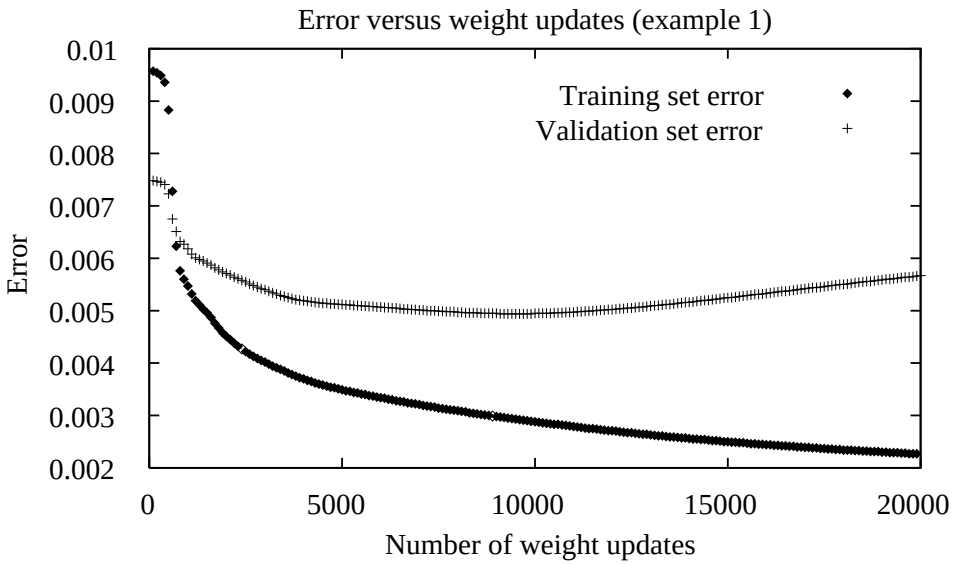
1. **Gradient descent to some local minimum**
   - Perhaps not global minimum...
   - Add momentum
   - Stochastic gradient descent
   - Train multiple nets with different initial weights
2. **Nature of convergence**
   - Initialize weights near zero
   - Therefore, initial networks near-linear
   - Increasingly non-linear functions possible as training progresses

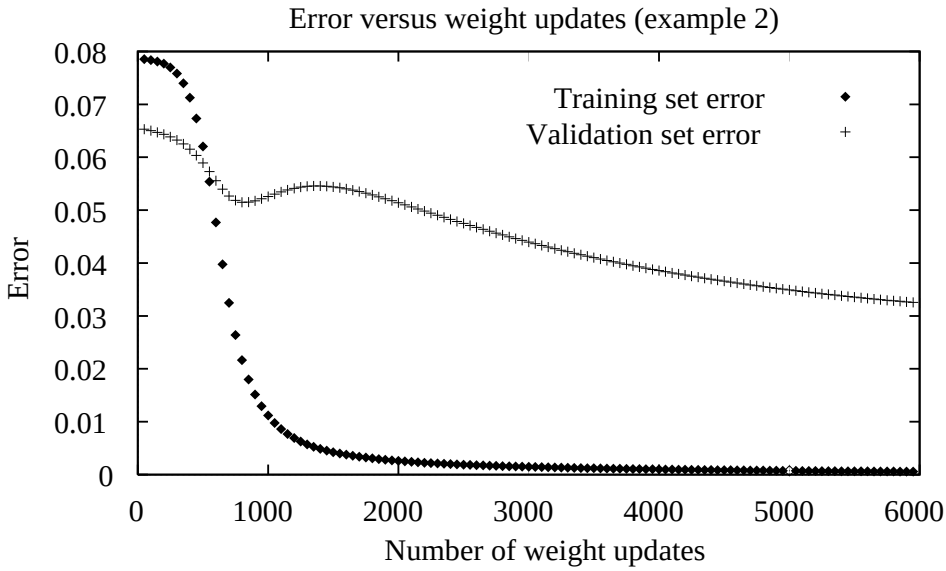
Expressive Capabilities of ANNs

1. **Boolean functions**
   - Every boolean function can be represented by network with single hidden layer BUT it might require exponential (in number of inputs) hidden units
2. **Continuous functions**

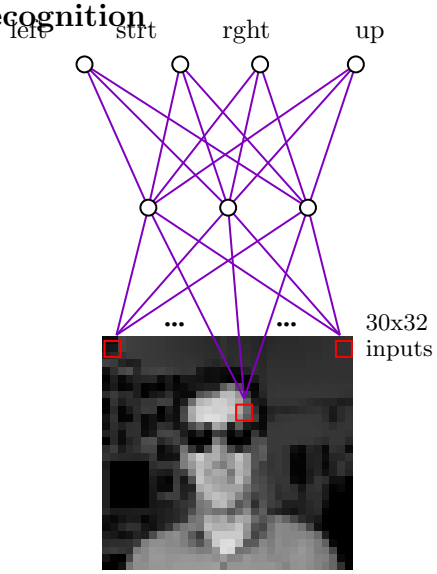
- **Universal Approximators.** Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Overfitting in ANNs



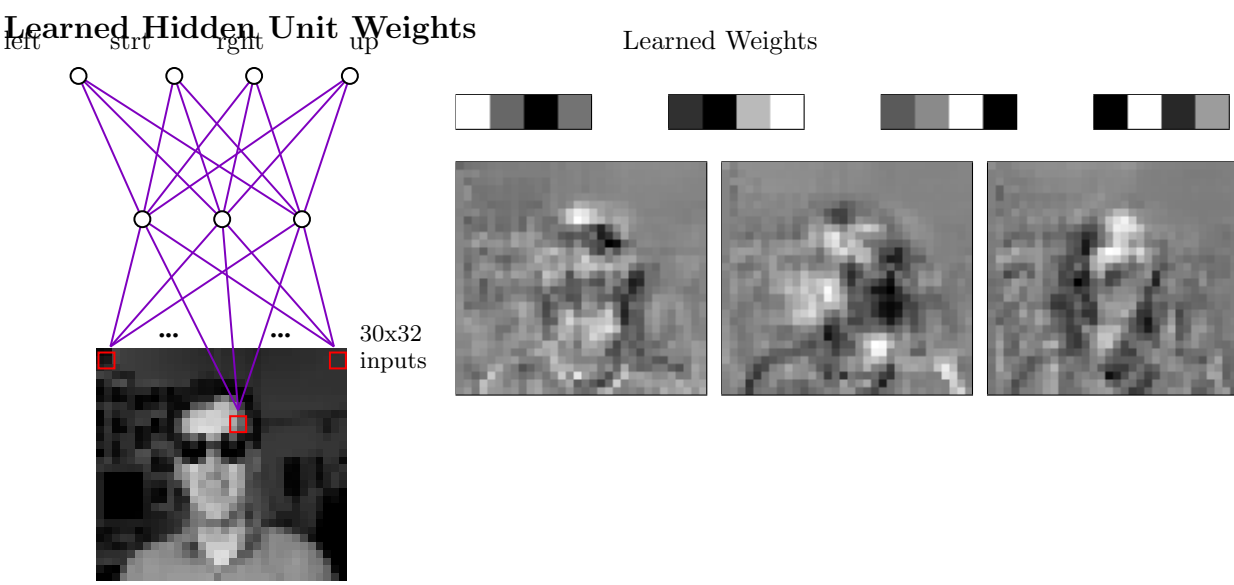


Neural Nets for Face Recognition



Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces



Typical input images

http://www.cs.cmu.edu/∼tom/faces.html

Alternative Error Functions

1. Penalize large weights:

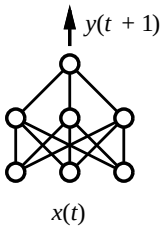
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

2. Train on target slopes as well as values:

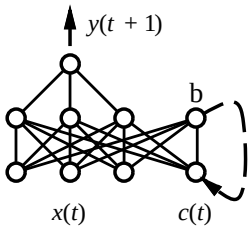
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

3. Tie together weights:
   - e.g., in phoneme recognition network

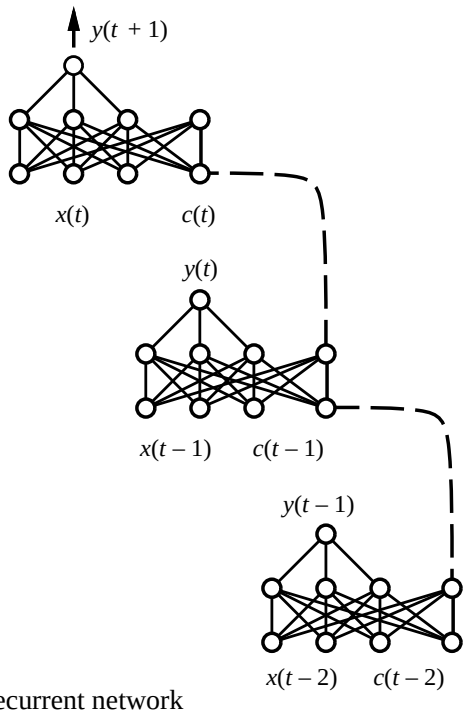
Recurrent Networks



(*a*) Feedforward network



(*b*) Recurrent network



(*c*) Recurrent network unfolded in time