09s1: COMP9417 Machine Learning and Data Mining

Machine Learning for Numeric Prediction

April 1, 2009

Acknowledgement: Material derived from slides for the book Machine Learning, Tom M. Mitchell, McGraw-Hill, 1997
`http://www-2.cs.cmu.edu/~tom/mlbook.html`

and slides by Andrew W. Moore available at
`http://www.cs.cmu.edu/~awm/tutorials`

and the book Data Mining, Ian H. Witten and Eibe Frank, Morgan Kauffman, 2000. `http://www.cs.waikato.ac.nz/ml/weka`

and the book Pattern Classification, Richard O. Duda, Peter E. Hart and David G. Stork. Copyright (c) 2001 by John Wiley & Sons, Inc.

Aims

This lecture will enable you to describe and reproduce machine learning approaches to the problem of numerical prediction. Following it you should be able to:

- define linear regression
- reproduce the basic method of implementing linear regression
- describe least mean squares
- define the problem of non-linear regression
- define neural network learning in terms of non-linear regression
- reproduce the method of back-propagation with sigmoid function and hidden layers
- reproduce the regression and model tree approaches for non-linear regression

Recommended reading: Mitchell, Chapter 4 (6.4);
Witten & Frank, pp. 119–126, 223-233.

Suggested exercises: Mitchell, 4.1-4.3, 4.5, 4.7 (4.11)

Relevant WEKA programs:
Linear Regression, Logistic, Multi-Layer Perceptron, M5P

Introduction

So far we have considered mainly *discrete* representations for data and hypotheses in Machine Learning . . .

. . . however, often find tasks where the most natural representation is that of *prediction of numeric values*

Introduction

For this class of representations, machine learning is viewed as:

searching a space of **functions** . . .

Introduction

Some methods:

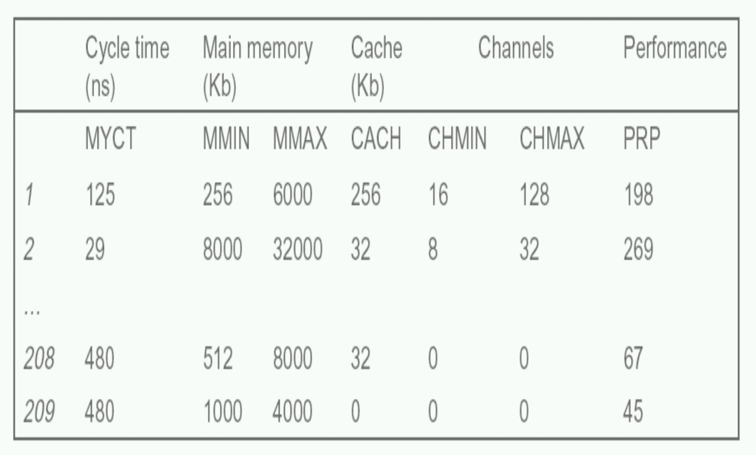
- linear regression (statistics) the process of computing an expression that predicts a numeric quantity
- perceptron (machine learning) a biologically-inspired linear prediction method

Introduction

- multi-layer neural networks (machine learning) learning non-linear predictors via hidden nodes between input and output
- regression trees (statistics / machine learning) tree where each leaf predicts a numeric quantity
  - the average value of training instances that reach the leaf
  - internal nodes test discrete **or** continuous attributes
- model trees (statistics / machine learning) regression tree with linear regression models at the leaf nodes
  - can fit with non-axis-orthogonal slopes
  - smoothing at internal nodes to approximate continuous function

Dataset: predicting CPU performance

Examples: 209 different computer configurations



	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
…							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

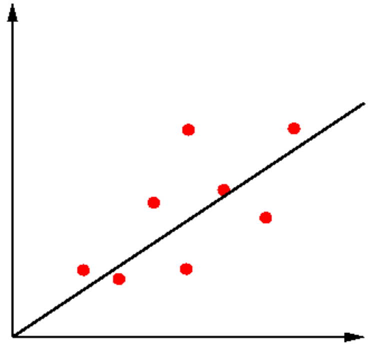
Linear Regression for CPU dataset

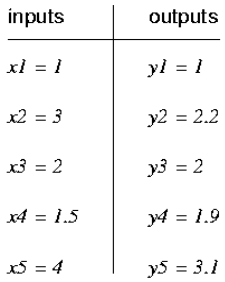
```
PRP =
  - 56.1
    + 0.049 MYCT
    + 0.015 MMIN
    + 0.006 MMAX
    + 0.630 CACH
    - 0.270 CHMIN
    + 1.46 CHMAX
```

Regression equation
Outcome: linear sum of attribute values with appropriate weights.

Regression
The process of determining the weights for the regression equation.

Linear Regression





inputs	outputs
$x1 = 1$	$y1 = 1$
$x2 = 3$	$y2 = 2.2$
$x3 = 2$	$y3 = 2$
$x4 = 1.5$	$y4 = 1.9$
$x5 = 4$	$y5 = 3.1$

Linear regression assumes that the expected value of the output given an input, $E[y|x]$, is linear.
Simplest case: Out$(x) = wx$ for some unknown w.
Given the data, we can estimate w.

Linear Regression

- Numeric attributes and numeric prediction
- Linear models, i.e. outcome is *linear* combination of attributes

$$y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m$$

- Weights are calculated from the training data
- **Predicted** value for first training instance $\vec{x}^{(1)}$ is:

$$w_0 x_0^{(1)} + w_1 x_1^{(1)} + w_2 x_2^{(1)} + \ldots + w_m x_m^{(1)} = \sum_{j=0}^{m} w_j x_j^{(1)}$$

Minimizing Squared Error

- Difference between *predicted* and *actual* values is the error !

$m+1$ coefficients are chosen so that sum of squared error on all instances in training data is minimized

Squared error:

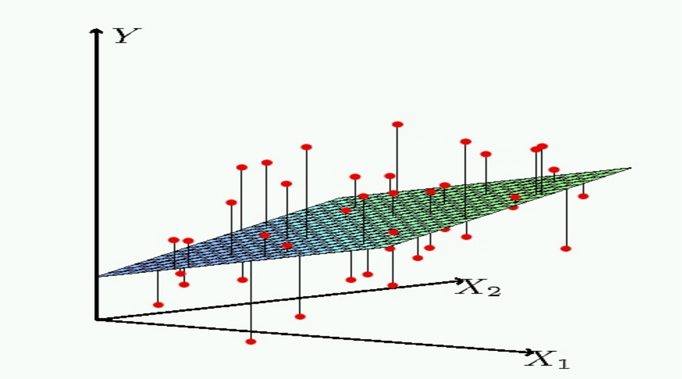
$$\sum_{i=1}^{n}\left(y^{(i)} - \sum_{j=0}^{m} w_j x_j^{(i)}\right)^2$$

Coefficients can be derived using standard matrix operations

Can be done if there are more instances than attributes (roughly speaking).

“Ordinary Least Squares” (OLS) regression – minimizing the sum of squared distances of data points to the estimated regression line.

Multiple Regression



Linear least squares fitting with 2 input variables.

Regression for Classification

- *Any* regression technique can be used for classification
  - Training: perform a regression for each class, setting the output to 1 for training instances that belong to class, and 0 for those that don’t
  - Prediction: predict class corresponding to model with largest output value (*membership value*)
- For linear regression this is known as *multiresponse linear regression*

Pairwise regression

- Another way of using regression for classification:
  - A regression function for every *pair* of classes, using only instances from these two classes
  - An output of $+1$ is assigned to one member of the pair, an output of -1 to the other
- Prediction is done by voting
  - Class that receives most votes is predicted
  - Alternative: “don’t know” if there is no agreement
- More likely to be accurate but more expensive

How many regressions for k classes ?

Logistic regression

- Problem: some assumptions violated when linear regression is applied to classification problems (not really fitting a line)
- Alternative: *logistic* regression
  - Designed for classification problems
  - Tries to estimate class probabilities directly using *maximum likelihood* method
  - Uses the linear model to predict the log of the odds of the class probability P

$$\log(P/(1-P)) = w_0x_0 + w_1x_1 + \ldots + w_kx_k$$

Discussion of linear models

- Not appropriate if data exhibits non-linear dependencies
- But: can serve as building blocks for more complex schemes (i.e. model trees)
- Example: multi-response linear regression defines a separating *hyperplane* for any two given classes:

$$w_0^{(1)}+w_1^{(1)}x_1+w_2^{(1)}x_2+\ldots+w_m^{(1)}x_m > w_0^{(2)}+w_1^{(2)}x_1+w_2^{(2)}x_2+\ldots+w_m^{(2)}x_m$$

which can be rewritten as

$$(w_0^{(1)}-w_0^{(2)}) + (w_1^{(1)}-w_1^{(2)})x_1 + \ldots + (w_m^{(1)}-w_m^{(2)})x_m > 0$$

where $\vec{w}^{(1)}$ and $\vec{w}^{(2)}$ are the weight vectors for the two classes.

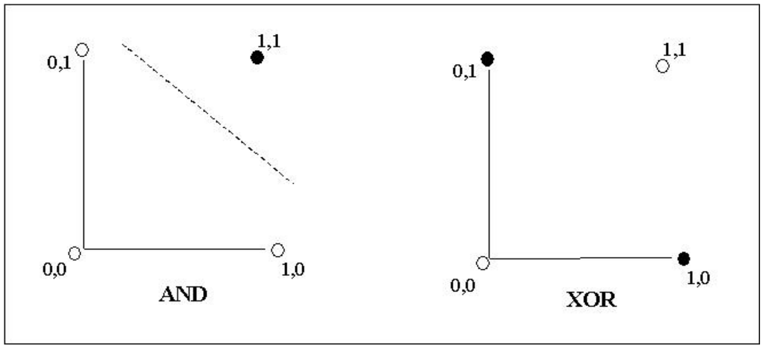
Discussion of linear models

Think of the hyperplane as separating all the positives on one side from the negatives on the other.

For pairwise linear regression this also applies – the only difference is that only the instances in each of the two classes is considered

However, sometimes it is not possible to define a separating hyperplane, even for some very simple functions . . .

Discussion of linear models



Filled circle – output one; hollow circle – output zero.

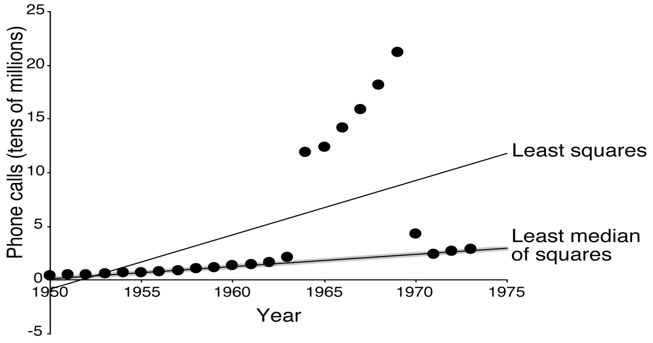
AND – divide 1's from 0's with single line; XOR – not possible.

Discussion of linear models

Dealing with noise – **robust regression**

- Statistical methods that address problem of outliers are called robust
- Possible way of making regression more robust:
  - Minimize absolute error instead of squared error
  - Remove outliers (i.e. 10% of points farthest from the regression plane)
  - Minimize *median* of squares instead of mean of squares (copes with outliers in x and y direction)
    * Geometric interpretation: finds narrowest strip covering half the observations (see figure)
    * Thickness measured in vertical direction
    * Least median of squares lies in centre of fuzzy grey band in figure

Discussion of linear models



Artificial Neural Networks

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

Connectionist Models

Consider humans:

- 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

→ much parallel computation

Connectionist Models

Properties of artificial neural nets (ANN's):

- 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

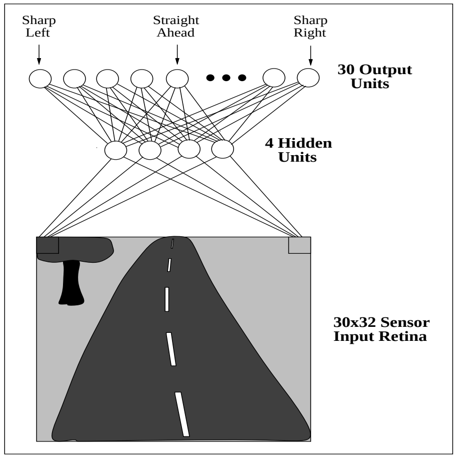
- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

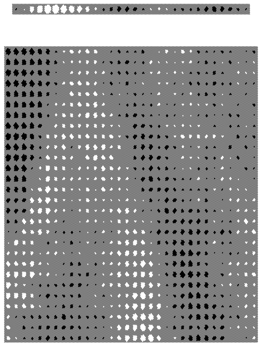
Examples:

- Speech phoneme recognition (NetTalk)
- Image classification (see face recognition data)
- many others . . .

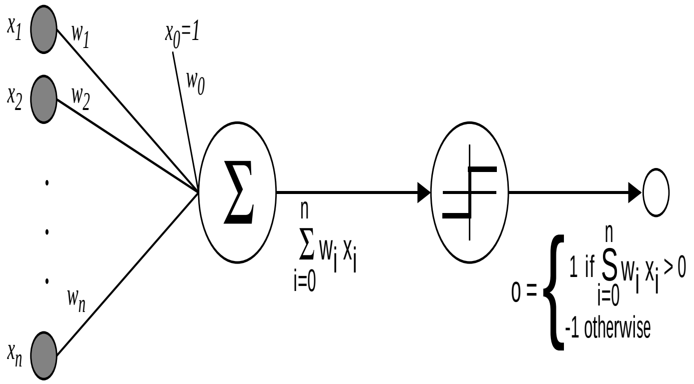
ALVINN drives 70 mph on highways

ALVINN





Perceptron



x_1
w_1
$x_0=1$
w_0
x_2
w_2
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

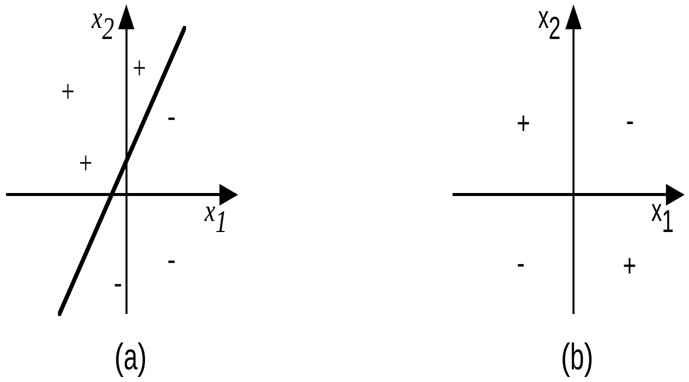
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Perceptron

Sometimes simpler vector notation used:

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

Decision Surface of a Perceptron



Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

Decision Surface of a Perceptron

But some functions not representable

- e.g., not linearly separable
- Therefore, we'll want networks of these...

Perceptron training 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
- η is small constant (e.g., .1) called *learning rate*

Perceptron training rule

Can prove it will converge

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

Gradient Descent

To understand, consider simpler *linear unit*, where

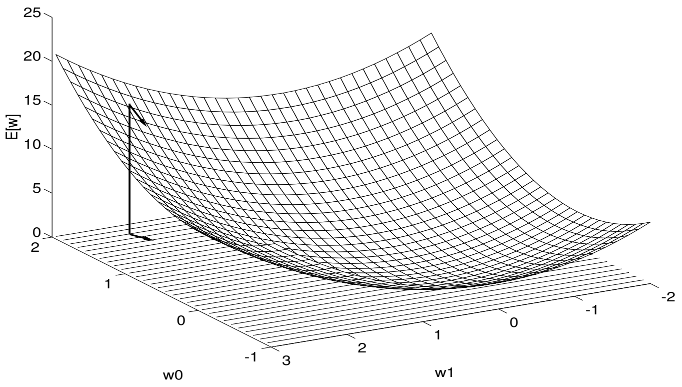
$$o = w_0 + w_1x_1 + \cdots + w_nx_n$$

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

Gradient Descent



Gradient Descent

Gradient

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

Training rule:

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

i.e.,

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

Gradient Descent

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

Gradient Descent

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

Each training example is a pair $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input
values, and $t$ is the target output value. $\eta$ 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 $\Delta w_i$ 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 $w_i$
            $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
    For each linear unit weight $w_i$
            $w_i \leftarrow w_i + \Delta w_i$
```

Training Perceptron vs. Linear unit

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

```
Do until satisfied

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

Incremental mode Gradient Descent:

```
Do until satisfied

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

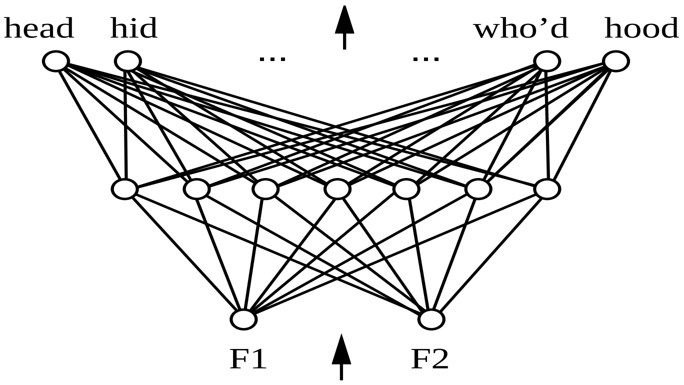
Incremental (Stochastic) Gradient Descent

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

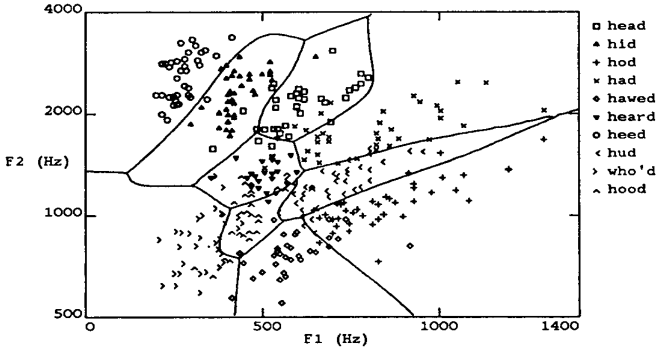
$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

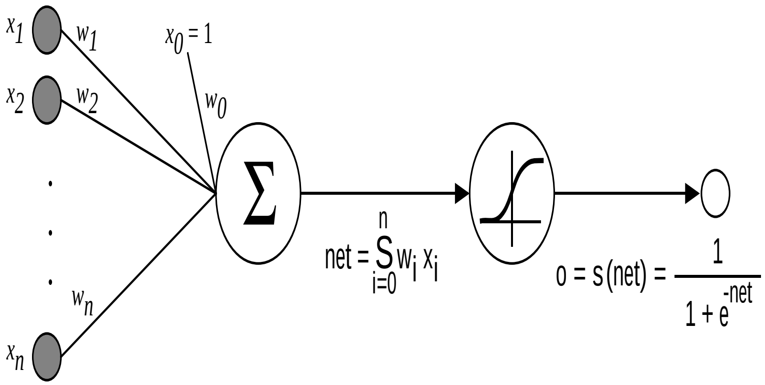
Multilayer Networks of Sigmoid Units



Multilayer Networks of Sigmoid Units



Sigmoid Unit


x_1
w_1
$x_0 = 1$
w_0
x_2
w_2
w_n
x_n
Σ
net = $\sum_{i=0}^{n} w_i x_i$
o = s(net) = $\frac{1}{1+e^{-net}}$

$\sigma(x)$ is the sigmoid function

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

Sigmoid Unit

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 → Backpropagation

Error Gradient for a Sigmoid Unit

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

Error Gradient for a Sigmoid Unit

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

```
Initialize all weights to small random numbers.
Until satisfied, Do
  For each training example, Do
    Input the training example to the network and
        compute the network outputs
    For each output unit $k$
      $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
    For each hidden unit $h$
      $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$
    Update each network weight $w_{i,j}$
      $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$
    where
      $\Delta w_{i,j} = \eta \delta_j x_{i,j}$
```

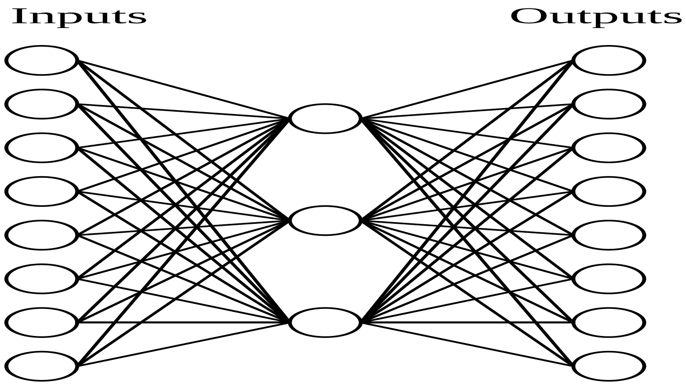
More on Backpropagation

- 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 → slow!
- Using network after training is very fast

Learning Hidden Layer Representations



Learning Hidden Layer Representations

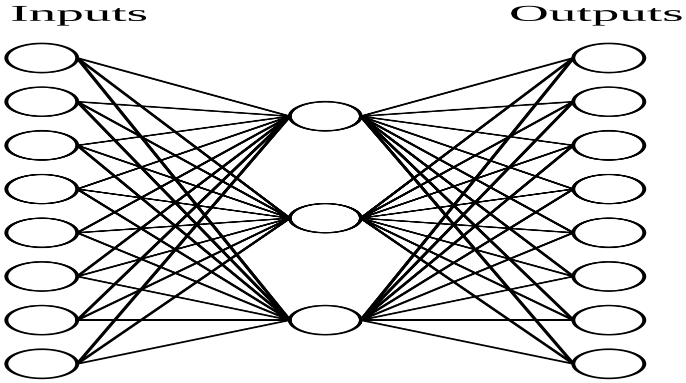
A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Learning Hidden Layer Representations

An *autoassociator* network:

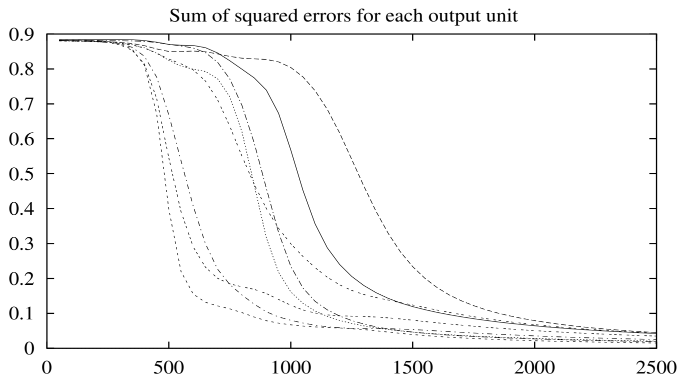


Learning Hidden Layer Representations

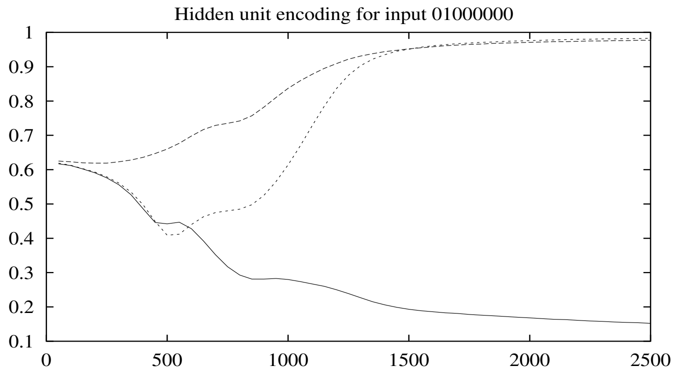
Learned hidden layer representation:

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

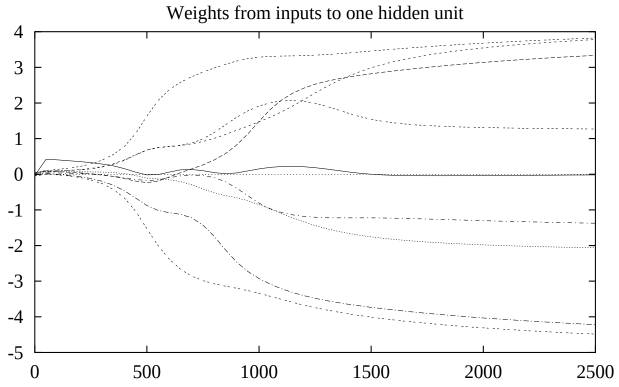
Training



Training the network



Training the network



Convergence of Backpropagation

Gradient descent to some local minimum

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

Nature of convergence

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

Expressive Capabilities of ANNs

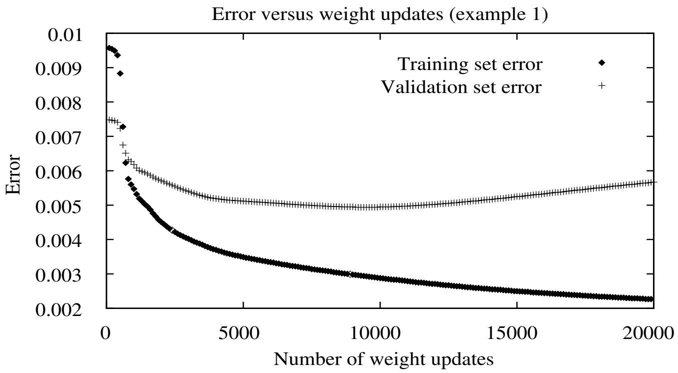
Boolean functions:

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

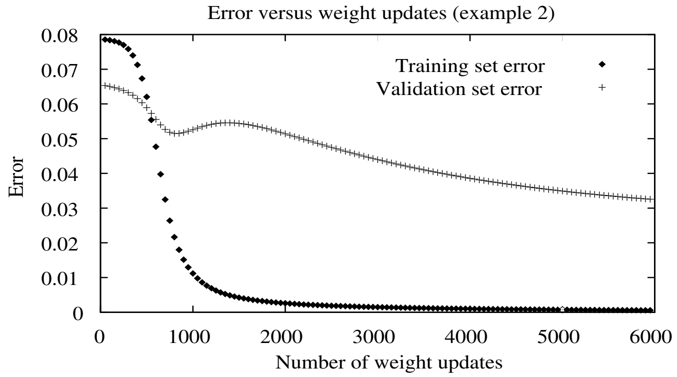
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [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



Overfitting in ANNs



Alternative Error Functions

Penalize large weights:

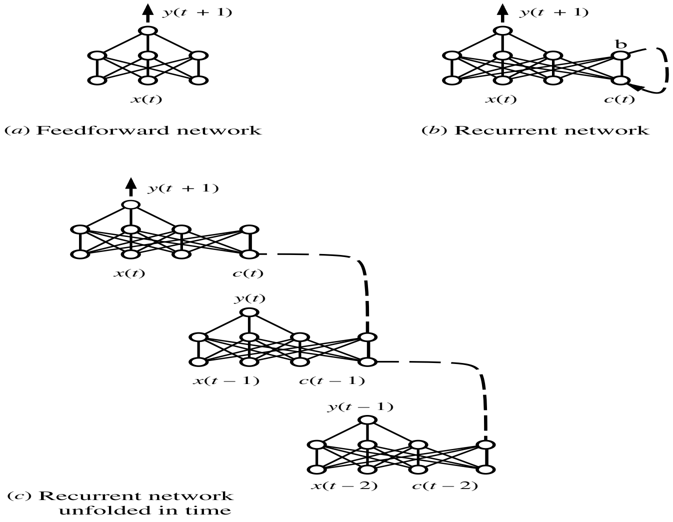
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

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

Tie together weights, e.g., in phoneme recognition network

Recurrent Networks



(a) Feedforward network

(b) Recurrent network

(c) Recurrent network unfolded in time

Evaluating numeric prediction

- Same strategies: independent test set, crossvalidation, significance tests, etc.
- Difference: error measures
- Actual target values: $a_1, a_2, \ldots, a_n$
- Predicted target values: $p_1, p_2, \ldots, p_n$
- Most popular measure: mean-squared error

$$\frac{(p_1 - a_1)^2 + \ldots + (p_n - a_n)^2}{n}$$

Easy to manipulate mathematically.

Other measures

The root mean-squared error:

$$\sqrt{\frac{(p_1 - a_1)^2 + \ldots + (p_n - a_n)^2}{n}}$$

The average (mean) absolute error is less sensitive to outliers than the mean-squared error:

$$\frac{|p_1 - a_1| + \ldots + |p_n - a_n|}{n}$$

Sometimes relative error values are more appropriate (e.g. 10% for an error of 50 when predicting 500)

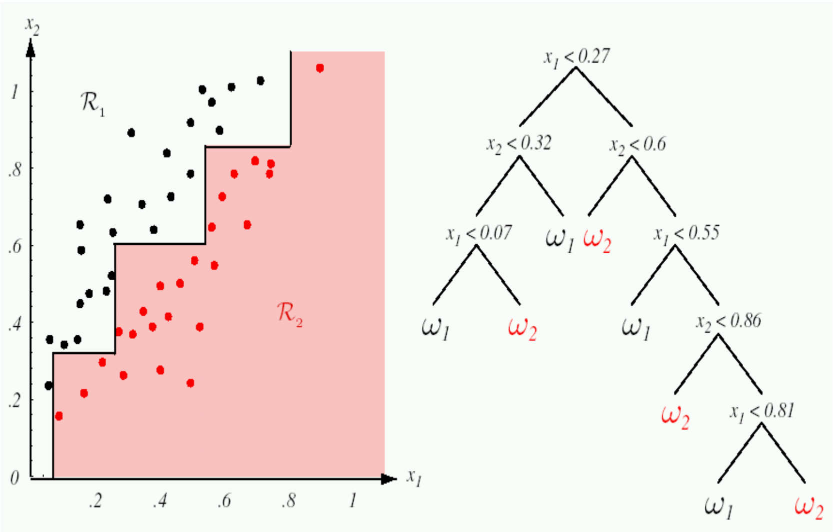
Non-linear Regression with Trees

Despite some nice properties of ANNs, such as generalization to deal sensibly with unseen input patterns and robustness to losing neurons (prediction performance can degrade gracefully), they still have some problems:

- Back-propagation does not appear to scale well – large nets may have to be partitioned into separate modules that can be trained independently, e.g. NetTalk
- ANNs are not very *transparent* – hard to understand the representation of what has been learned

Possible solution: exploit success of tree-structured approaches in ML

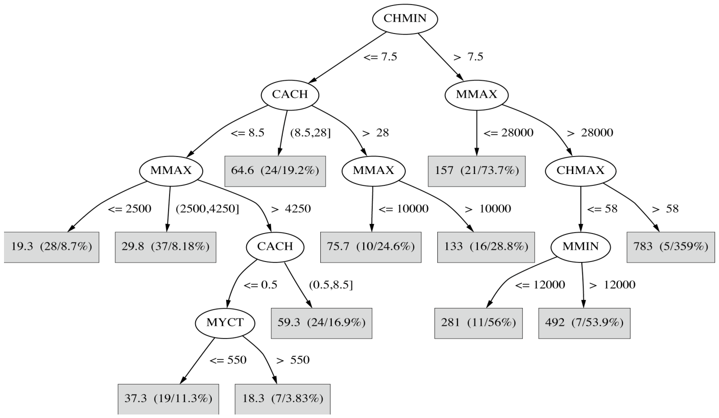
Example: Regression Tree



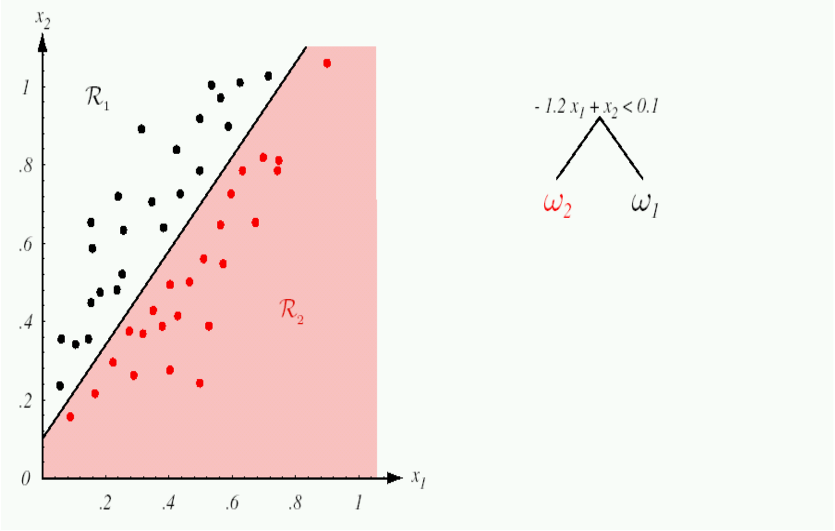
Regression trees

- Differences to decision trees:
  - Splitting criterion: minimizing intra-subset variation
  - Pruning criterion: based on numeric error measure
  - Leaf node predicts average class values of training instances reaching that node
- Can approximate piecewise constant functions
- Easy to interpret
- More sophisticated version: model trees

Regression Tree on CPU dataset



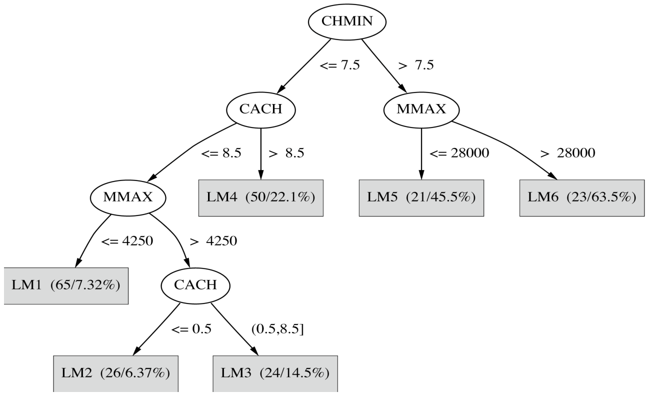
Model Tree



Model trees

- Like regression trees but with linear regression functions at each node
- Linear regression applied to instances that reach a node after full tree has been built
- Only a subset of the attributes is used for LR
  - Attributes occurring in subtree (+maybe attributes occurring in path to the root)
- Fast: overhead for Linear Regression (LR) not large because usually only a small subset of attributes is used in tree

Model Tree on CPU dataset



Smoothing

- Naïve prediction method – output value of LR model at corresponding leaf node
- Improve performance by *smoothing* predictions with *internal* LR models
  - Predicted value is weighted average of LR models along path from root to leaf
- Smoothing formula: $p' = \frac{np+kq}{n+k}$ where
  - p' prediction passed up to next higher node
  - p prediction passed to this node from below
  - q value predicted by model at this node
  - n number of instances that reach node below
  - k smoothing constant
- Same effect can be achieved by incorporating the internal models into the leaf nodes

Building the tree

- Splitting criterion: *standard deviation reduction*

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i)$$

  where $T_1, T_2, \ldots$ are the sets from splits of data at node.

- Termination criteria (important when building trees for numeric prediction):
  - Standard deviation becomes smaller than certain fraction of sd for full training set (e.g. 5%)
  - Too few instances remain (e.g. less than four)

Pruning the tree

- Pruning is based on estimated absolute error of LR models
- Heuristic estimate:

$$\frac{n+v}{n-v} \times \text{average\_absolute\_error}$$

  where n is number of training instances that reach the node, and v is the number of parameters in the linear model

- LR models are pruned by greedily removing terms to minimize the estimated error
- Model trees allow for heavy pruning: often a single LR model can replace a whole subtree
- Pruning proceeds bottom up: error for LR model at internal node is compared to error for subtree

Discrete (nominal) attributes

- Nominal attributes converted to binary attributes and treated as numeric
  - Nominal values sorted using average class value for each one
  - For k-values, $k-1$ binary attributes are generated
    - $*$ the ith binary attribute is 0 if an instance's value is one of the first i in the ordering, 1 otherwise
- Best binary split with original attribute provably equivalent to a split on one of the new attributes

Pseudo-code for M5prime

- Four methods:
  - Main method: MakeModelTree()
  - Method for splitting: split()
  - Method for pruning: prune()
  - Method that computes error: subtreeError()
- Note: linear regression method is assumed to perform attribute subset selection based on error

MakeModelTree()

```
MakeModelTree(instances)
{
   SD = sd(instances)
   for each k-valued nominal attribute
      convert into k-1 synthetic binary attributes
   root = newNode
   root.instances = instances
   split(root)
   prune(root)
   printTree(root)
}
```

split()

```
split(node)
{
   if sizeof(node.instances) < 4 or
         sd(node.instances) < 0.05*SD
      node.type = LEAF
   else
      node.type = INTERIOR
      for each attribute
         for all possible split positions of the attribute
            calculate the attribute's SDR
      node.attribute = attribute with maximum SDR
      split(node.left)
      split(node.right)
}
```

prune()

```
prune(node)
{
   if node = INTERIOR then
      prune(node.leftChild)
      prune(node.rightChild)
      node.model = linearRegression(node)
      if subtreeError(node) > error(node) then
         node.type = LEAF
}
```

subtreeError()

```
subtreeError(node)
{
   l = node.left; r = node.right
   if node = INTERIOR then
      return (sizeof(l.instances)*subtreeError(l)
            + sizeof(r.instances)*subtreeError(r))
            / sizeof(node.instances)
   else return error(node)
}
```

Summary

- ANNs since 1940s; popular in 1980s, 1990s
- Regression trees were introduced in CART
- Quinlan proposed the M5 model tree inducer
- M5′: slightly improved version that is publicly available
- Quinlan also investigated combining instance-based learning with M5
- CUBIST: Quinlan's commercial rule learner for numeric prediction `www.rulequest.com`
- Interesting comparison: Neural nets vs. M5 – both do *non-linear regression*