AI
- think like humans / rationally
- act like humans /rationally
- . weak ai - act smart,
- . strong ai - conscious

[T: Theorem]

Search > Uninformed

[Fringe: , Expanded: ,]
[N.=Node I=Implementation]

BFS
- Expands **Shallowest** unexpanded N.
- (I: put children of the Expanded N. at the end of the fringe)
- *Complete: Yes (if b is finite)*
- *Optimal: No in General (Yes if **Step Cost** is the same)*
- *Time: 1+b+b^2+...b^d = O(b^d), expontnl*
- *Space: O(b^d), (keeps every node in memory)*

Uniform Cost (UCS)
- Expands **Least-Cost** Unexpanded N, g(n)
- (I: insert nodes in the fringe in order of increasing path cost From the root)
- *Complete: Yes, if step cost > 0*
- *Optimal: Yes*
- *Time & Space: #N. with g≤cost of Optimal Soln O(b^d), (depends on path costs, not depths, difficult to caterize in terms of b, d)*
- UCS = BFS, when g(n)=depth(n)

DFS
- Expands **Deepest** Unexpanded N
- (I: insert successors at front of fringe)
- *Complete: No, fails in infinite-depth spaces (i.e. m= ∞)*
- *Optimal: No*
- *Time: 1+b+b^2+...b^m = O(b^m), (higher than BFS, as M>>d (m=max depth, d=least cost soln path))*
- *Space: O(bm), linear, excellent.*

Depth Limited Search
- DFS w/ depth limit l
- *Complete: No in general, Yes in finite spc.*
- *Optimal: No*
- *Time: 1+b^2+...b^l= O(b^l) (as BFS)*
- *Space: O(bl) (as DFS)*

Iterative Deepening (IDS)
- Expands deepest unexpanded node within level l.
- *Complete: Yes (as BFS)*
- *Optimal: Yes, if step cost=1 (as BFS)*
- *Time: O(b^d) (as BFS)*
- *Space: O(bd), linear (as DFS)*
- (can be modified to explore uniform-cost tree)

Search > Informed (heuristic)

(+ve over uninformed, Knows if a non-goal node > another, typically more efficient)
[Best First Search Algorithms : Expands the most desirable unexpanded node]
[N=Node, g(n)=pathCost, h(n)=heuristic]

Greedy
- Expands N. with smallest h(n).
- *Complete: Yes, in finite space. Fails in ∞ space (+ can get stuck in loops)*
- *Optimal: No.*
- *Time: O(b^m), but good heurstic can Improve lots.*
- *Space: O(b^m), keeps every node in memory*
- Greedy=BFS, h(n) = depth(n), ties+ L}R
- Greedy=DFS, h(n)=-depth(n), ties+ deepest 1st.
- Greedy=UCS, h(n)=g(n)

A* (Tree Search)
- Expands N. with smallest f(n)=g(n)+h(n)
- T: If h is an admissible heuristic, than A* is Complete And Optimal. (only w/ Tree S.)
- T: If h is consistent, than A* is Optimally Efficient, among all optimal search algorithms. (always true ^v) (It will not revisit states (as in graph search)).
- *Complete: Yes, unless there are ∞'ly many nodes with f≤f(G), G–Optimal Goal State*
- *Optimal: Yes, with admissible heuristic*
- *Time: O(b*^d),exponential, b*=effective branching factor*
- *Space: Exponential, keeps all nodes in memory.*
- *[Both Time&Space are probs for A*, typically running out of SPACE b4 time] May be better to settle for a non-admissible h that works well even though completeness and optimality are no longer guaranteed. Simpler, faster h may be better even though more N.s expan+.*
- A*=UCS, if h(n)=0 for all N.

A* (Graph Search)
- If h is **admissible**, Complete & Optimal, if revisiting repeated (:||) states allowed (reopening closed N.s), else not optimal.
- If h is **Consistent**, we avoid :|| states.
- **Enforced Consistency**, using 'PathMax': set child's f value to parent's f value. (If done as we search, may not solve problem of reopening N.s from Closed. Better to ensure before search starts that h(n) is consistent).

Heuristics (h(n))

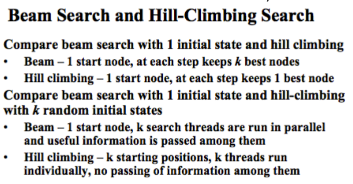
- **Admissible** h(n) are *optimistic* (smaller than true cost). e.g. SLD.
- **Dominance**, when an Admissible Heuristic is better than another admissible one. A better estimate of true $ to G, and expanding fewer nodes in A*.
- **Inventing**, h(n)'s by creating for a relaxed version of prolem (1 w/ less restrictions on the actions)
  [T: $ of an optimal soln to a Relaxed problem is an Admisible h(n) for the Original problem.]
- (composite heuristics h(n)=max{h1(n),h2(n)...} are admissible, good to use when there's a bunch of h(n), none dominating one another, composite will Dominate.)
- **Consistent (monotonic) Heurisic**, if all such pairs in the search graph satisfy the triangle inequality:
  [h(ni),par. ≤ cost(ni,nj)+h(nj),chi. for all n]
  • f(nj),child ≥ f(ni),parent : f is non-decreasing along any path
- (Admissible (Consistent))

Local Search Algorithms

(Optimisation Problems)

Hill-Climbing
- Finds closest local min.imum / max.imum. (may not be global)
- Soln found depends on Initial State: Can run several times starting from ∂ rand. points.
- Plateus: (random walk - no change in v, wander endlessly, revisiting prev. N.s): Can keep track of # of times v is the same and don't allow revisitng of nodes w/ same v.
- Ridges: (cur. local max. not good 'nuff): Can combine 2/+ moves in a macro, or allow limited # of look-ahead search

Beam Search
- Keeps track of **k** BEST states (not 1)
- (• 2 vrsns: 1. start w/ 1 given state OR 2. k randomly generated states
  • At each iteration (lvl): gen.erate all successors of all k states
  • If any one is a goal state, stop; else select k best successors and continue.)



Beam Search and Hill-Climbing Search

Compare beam search with 1 initial state and hill climbing
- **Beam – 1 start node, at each step keeps *k* best nodes**
- **Hill climbing – 1 start node, at each step keeps 1 best node**

Compare beam search with 1 initial state and hill-climbing with *k* random initial states
- **Beam – 1 start node, k search threads are run in parallel and useful information is passed among them**
- **Hill climbing – k starting positions, k threads run individually, no passing of information among them**

- Can be used w/ A*, +ve: memory efficieny, -ve: !complete, !optimal
  - Variations: Keep only nodes that are at most €(beam width) worse than best N.

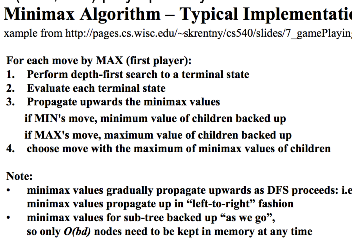
Simulated Annealing
- (similar to hill climbing, but selects random successor)
- (• select initial state s. set cur. N. to s
  • Randomly select m, one of N.'s succssrs
  • if v(m) > v(n), n=m //accept m
    else n=m w/ small probability
    //accept m w/ small prob.
  • Anneal T, • Repeat xtimes/goodnuff)
- Probability: P = e^((v(m)-v(n))/T)
  i.e. bad move v(n)>v(m) asuming looking for min., P decreases expo. w/ badness of move.
- T decreases, anneals, w/ time, e.g. T*=.8
- Thrm: If schedule lowers T slowly enough, algorithm will find global optimum. *Complete & Optimal, given a long enough cooling schedule.*
- Difficult to set "slowly enough" (T).

Genetic Algorithms
- (• Select best individuals, from fitness f()
  • CrossOver, about an init random point
  • Mutate, random change of bits)
- Success depends on representation **(encoding)**
- *!complete, !optimal*

Games

[Deterministic vs Chance]
[Perfect vs Imperfect]
[Zero-Sum vs Non-0-∑ : (1 P's gain is another's loss)]
(Processes forward, !backward from goal, cause often too many goal states. + if goal state too far, will not provide any useful info on termination otherwise)

MinMax Algorithm
- Perfect, Deterministic, Assumes both P's (Max, Min) play optimally



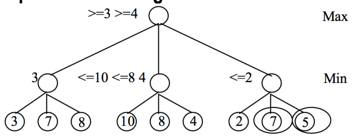
Minimax Algorithm – Typical Implementati
xample from http://pages.cs.wisc.edu/~skrentny/cs540/slides/7_gamePlayin

For each move by MAX (first player):
1. **Perform depth-first search to a terminal state**
2. **Evaluate each terminal state**
3. **Propagate upwards the minimax values**
   if MIN's move, minimum value of children backed up
   if MAX's move, maximum value of children backed up
4. **choose move with the maximum of minimax values of children**

Note:
- **minimax values gradually propagate upwards as DFS proceeds: i.e minimax values propagate up in "left-to-right" fashion**
- **minimax values for sub-tree backed up "as we go", so only *O(bd)* nodes need to be kept in memory at any time**

- Only O(bd) nodes need be kept in memory at a time
- If Min doesn't play optimally, Max will do even better.
- Implemented as DFS
- Assumptions: branching factor b, all terminal Nodes ad depth d.
- *Optimal: Yes.*
- *Time: O(b^m) as in DFS – main Problem.*
- *Space: O(bm) as in DFS*

Alpha Beta Pruning

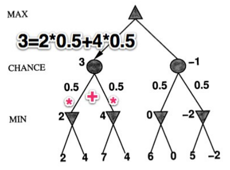


- **Traverse the search tree in depth-first order**
- **Apply utility function at each leaf node generated**
- **Compute values backwards**
- **At each non-leaf node store a value indicating the best backed up value so far**
  - **MAX nodes: alpha value = best (max) value found so far**
  - **MIN nodes: beta value = best (min) value found so far**
- **Given a node *n*, cutoff the search below *n* if:**
  - ***n* is a MAX node and *alpha(n) ≥ beta (i)* for some MIN node *i* that is ancestor of *n* (*beta cutoff*)**
  - ***n* is a MIN node and *beta(n) ≤ alpha(i)* for some MAX node *i* that is ancestor of *n* (*alpha cut*)**

- Pruning doesn't effect final result
- Worst Case: No Pruning: O(b^d)
- Best Case: Perfect Ordering O(b^(d/2))

Imperfect
[Both MinMax & AlphaBeta require too much time]
i.e. Heuristic Evaluation at leaf nodes.
- Probs: Horizon Effect (hidden pitfalls)
- - Soln: Evaluation f() should be applied onlty to positions that are **quiescent**, unlikely to change extremely in near future. + **Secondary** Search, extending search to make sure there's no hidden pitfall.

ExpectMinMiMax
(Non-deterministic Games)
- Time: O(b^m . n^m) n=# of distinct dice rolls.



Machine Learning [Supervised, Unsupervised, Reinforced, Associations Learning]

Supervised Learning
(Classification - categorical, Regression - Numeric)

NEAREST NEIGHBOUR
- (or distance/instance-based learning)
- An eg. of Lazy Learning, stores all training eg.s + doesn't build a classifier until new eg. needs to be classified. - Opposite to Eager learning (constructing classifier before recieving new eg.s, like1R,DT,NB..)
- **Lazy** classifiers are Faster at training(=memorizing), but Slower at classification.
- Nearest determined by **distance**

Eucledian distance – most frequently used

$$D(A,B)=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+...+(a_n-b_n)^2} \qquad \mathbf{q = 2}$$

Manhattan (or city block) distance

$$D(A,B)=|a_1-b_1|+|a_2-b_2|+...+|a_n-b_n| \qquad \mathbf{q = 1}$$

Minkowski distance – generalization of Eucledian & Manhattan

$$D(A,B)=(|a_1-b_1|^q+|a_2-b_2|^q+...+|a_n-b_n|^q)^{1/q} \quad q \text{ – positive int}$$

- Need for **Normalization**, as when calculating distances between 2 examples, the effect of the attributes with smaller scale will be less significan than those larger. i.e. Normalize (between 0 and 1)
- *Training=Fast, no model built, just storing.*
- *Classification>Lookup: O(mn), m training examples w/ dimensionality n.*
- *Memory: O(mn), need remember each eg*
- *BAD for large datasets, **slow**.*

K-Nearest Neighbour
- k majority voting
- Very Sensitive to value of k, general rule: [k ≤ sqrt(#training_egs)]
- Also usable w/ numeric prediction (regression) by averaging values.
- **Distance** for Nominal Attributes: 0 - ∂ the same, 1 otherwise
- for **Missing Values**: 0 - both same & NON-MISSING, else 1 (if numeric, d=max(v, 1.0-v))
- d((red, new, ?, ?, ?),(blue, new, ?, 0.3, 0.8)) = 1 + 0 + 1 + 0.7 + 0.8
- **Variation: Weighted Nearest Neighbor**
- Closer neighbors count more than distant neighbours. Instead of k, all training egs.
- Weight Contribution based off distance to new example:

$$w_i=\frac{1}{D(X_{new},X_i)}$$ -ve: slower algorithm

- **Curse of Dimensionality:** NN great in low dimensions (up to 6), but become ineffective as dimensionality increases. As more examples are far from one another, and close to the boundaries. (Notion of nearness becomes ineffective in high-dim space)
  Soln: Feature selection (attrs) to reduce dimensionality.
- Produces arbitarially shaped decision boundary defined by a subset of the Voronoi edges.
- **Sensitive to Noise**
- Standard algorithm makes predictions based on LOCAL info. 1R, DT, NNs, try to find a GLOBAL model that fits training set.

1-RULE
- For each attribute value makes rule by majority class. Calculates error rate of rules. Chooses rule w/ smallest error rate.
- **Missing** Values: Treated as another attribute Value
- **Nominal** Attributes are discretized to nominal. - May lead to overfitting due to noise in data - Soln: impose min num of egs of majority class in each partition, merge.
- Simple, Computationally Cheap.

NAIVE BAYES
(Statistical-Based Classification)
- P(H|E) = P(E|H).P(H) / P(E)
- P(yes|E)= P(E1|yes).P(E2|yes).P(yes)/P(E)
  • (all P(Ex|yes) have same denominator)
- Assumes attributes are *equally important* and *independent* of one another.
- **Laplace Correction** to handle Zero-Numerators. (add 1/Num_Attrs to all attrs.)

$$P(sunny\,|\,yes)=\frac{\boxed{0+1}}{9+3}=\frac{1}{12}$$

$$P(overcast\,|\,yes)=\frac{4+1}{9+3}=\frac{5}{12}$$

$$P(rainy\,|\,yes)=\frac{3+1}{9+3}=\frac{4}{12}$$

- (note, in tut examples, only the value is ∂d, not cousins)
- **Missing** Nominal Values: Ommit Value from P(yes|E) and P(no|E) counts.
- If Numeric: Calc. Probability Distribution using the Probability Densitiy Function (assuming normal distribution). μ=mean, Ó=sd.

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad f(temperature=66\,|\,yes)=\frac{1}{\boxed{6.2}\sqrt{2\pi}}e^{-\frac{(66-\boxed{73})^2}{2*\boxed{6.2}^2}}$$

- +ves: • simple,
  • Excellent Computational Complexity: Requires 1 scan of the training data to calculate statisics (for both nominal & continuous attributes assuming normal distribution). O(pk), p=#training_egs, k=valued_attributes
  • Robust to isolate **noise** points (avgd out)
- -ves: • Correlated attributes reduce power (violation of independence assumption) - Soln: feature selection b4hand.
  • many numeric features not normally distrubted - Soln: other types of distributions /transform attribute to normally distributed one /discretize data first.

Evaluating Classifier
- **Holdout Procedure** - split data into 2 independent sets;Training & Test (~2/3,1/3)
- Accuracy (Success Rate) = 1.0 - Error Rate
- Validation Set (for DTs, NNs) - Classifier built from Training Set, - Tuned w/ Validation Set, Evaluated w/ Test Set.
  • DTs - training set used to build tree, validation set used to prune, test to eval
  • NNs - validation set used to stop training, prevent overtaining.
- Prob: egs in training set may not be representative of all classes.
  Soln: **Stratification**, ensures each class is represented w/ ~= proportions in both sets.
- Holdout more reliable by repeating (Repeated Holdout Method) - which can be improved by ensuring Test sets don't overlap - **Cross Validation**.
- **Leave-One-Out Cross Validation** - n-fold cross-validation, where n=#Egs in data set.
  • +ve: greates possible amount of data used, deterministic procedure.
  • -ve: high computation cost
  i.e. more useful for small data sets

Comparing Classifiers

	C1	C2	
Fold 1	95%	91%	d1=|95-91|=4
…			
Fold 2	82%	85%	d2=|82-85|=3
==================			
mean	91.3%	89.4%	d_mean=3.5

$$\hat{\sigma}^2=\frac{\sum_{j=1}^{k}(di-d\_mean)^2}{k(k-1)}$$

1. **Calculate the differences di**
2. **Calculate the variance of the difference (an estimate of the true variance)**
   If k is sufficiently large, di is normally distributed
3. **Calculate the confidence interval Z**

$$Z=d\_mean\pm \boxed{t_{(1-\alpha)(k-1)}}\sigma$$

• Obtained from a probability table
1-α – confidence level
k-1 – degree of freedom

4. **Interval contains 0 –difference not significant, else significant**

Suppose that
- We use 10 fold CV => k=10
- d_mean=3.5 and the standard deviation is 0.5
- We are interested in significance at 95% confidence level

$$Z=3.5\pm 2.26\times 0.5=3.5\pm 1.13$$

Interval does not contain 0 => difference is statistically significant

Confusion Matrix
(remember accuracy=(tp+tn)/(tp+tn+fp+fn))

Information retrieval (IR) uses *recall (R), precision (P)* and their combination *F1 measure (F1)* as performance measures

$$P=\frac{tp}{tp+fp} \qquad R=\frac{tp}{tp+fn} \qquad F1=\frac{2PR}{P+R}$$

examples	# assigned to class yes	# assigned to class no
# from class yes	true positives (tp)	false negatives (fn)
# from class no	false positives (fp)	true negatives (tn)

Where retrieved=#retrievedDocs, relevant=#relDocs
Precision = (R+R)/Retrieved
Recall = (R+R)/Relevant

Inductive Learning
- Supervised Learning is Inductive Learning.
- Induction: inducing the universal from the particular.
- We can generate many hypothese h, the set of all possible h form the hypothesis space H, a good h will generalize well (i.e. predict new egs correctly).
- Choice of H important (eg sin.f() vs polyn).
- Empirical Evaluation: - best thing we can do. (Test data).

DECISION TREES
- (top-down recursive divide-and-conquer)
- Growing the tree: *hill climbing search* guided by information gain.
- Can represent any boolean function.
- Compact Decision tree with most pure (high entropy) attributes. Entropy is measured in bits. For binary classification; value:0.5=1bit, value:0or1=0bit.
- **Entropy** H(Y) measures: • the disorder of a set of training egs w/ respect to class Y. • shows the amount of surprise of the receiver by the answer Y based on probability of answers • the smallest # of bits per symbol (on avg) needed to transmit a stream of symbols drawn from Y's distribution.
- **Information Gain** is the expected reduction in entropy caused by the partitioning of the set of examples using that attribute.

$$Gain(S\,|\,A)=H(S)-\sum_{j\in values(A)}P(A=v_j)H(S\,|\,A=v_j)=$$
$$=H(S)-\sum_{j\in values(A)}\frac{|S_v|}{|S|}H(S\,|\,A=v_j)$$

- (DT: The best attribute has **Highest** Gain)
- **Overfitting**, due to DTs growing each branch deeply to perfectly classify, coupled with noise in training data, +/or small training set. Soln: Pre/Post-Pruning.
- **Stop** Pruning: Estimate accuracy w/ validation set (acts as safety net), however tree is built on less data.
- Post-**Pruning** > Tree Pruning:
  • Sub-Tree Replacement:
  - start from leaves+work to root.

For each candidate node
- **remove the sub-tree rooted at it**
- **make it a leaf and assign the most common label of the training examples affiliated with that node**
- **Calculate accuracy of the new tree on validation set and compare with the old tree**

Choose the node whose removal results in the highest increase of accuracy and prune it
- **no nodes are pruned if the new tree is worse than the old**

- continue iteratively till further pruning=harmful
• Sub-Tree Raising
- potentially time consuming operation, restricted to raising the sub-tree of most popular branch.
- Post-Pruning > **Rule** Pruning:
  • Grow Tree • Convert tree to equivalent set of rules by creating 1 rule per path (if statement) • Prune each rule by removing any pre-conditions that result in improving estimated accuracy • Sort pruned rules by estimated accuracy, consider in this sequence when classifying subsequent instances.
  • Converting DT to Rules Be4 Pruning: Provides bigger **flexibility**, when trees are pruned; can only remove node completely or retain. when rules are pruned there are less restrictions: pre-conditions, not nodes, are removed. each branch in tree (i.e. each rule) treated separately. removes distinction between attribute tests that occur near the root of the tree and those near the leeves.
  • +ve > trees, easier to read.
- **Numerical** attributes need discretization, in DTs we're restricted to binary split.
- Problem: if an attribute is **highly-branchng** then likely selected by Information Gain, can lead to **Overfitting**.
  Soln: **Gain Ration**, a modification of the Gain that reduces its bias towards highly branching attributes.

it penalizes highly-branching attributes by incorporating *SplitInformation*

$$splitInformation(S\,|\,A)=-\sum_{i=1}^{c}\frac{|S_i|}{|S|}\log_2\frac{|S_i|}{|S|}$$

SplitInformation is the entropy of S w.r.t. the values of A

Gain ratio:
$$GainRatio(S\,|\,A)=\frac{Gain(S\,|\,A)}{SplitInformation(S\,|\,A)}$$

- **Missing** values: handled by • unique value • A(x)=most common value among training egs at n w/ class(x) • sample fractioning strategy - assign probability to each possible value of A, calc prob.s, use frequencies of values of A among examples at n
- Attributes w/ Different Costs, Incorporating cost in Gain (penalizing attrs w/ high cost)

nmer (90)
$$\frac{Gain^2(S\,|\,A)}{Cost(A)}$$

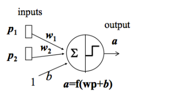
- $$\frac{2^{Gain(S|A)}-1}{(Cost(A)+1)^w}$$ **where w ∈ [0,1] determines cost importance**

- Efficient:
  • Cost of Building tree O(mnlogn), n-instances and m attributes.
  • Cost of pruning tree w/ sub-tree replacement O(n)
  • Cost of pruning by subtree lifting O(n(logn)^2)
  • ∑ (build+prune): O(mnlogn)+O(n(logn)^2)
- Resulting Hypthoese Easy to interpret by humans if DT not too big.

NEURAL NETWORKS

(Perceptron - StepTransferF(), Forms linear Decision Boundary)

Single-Neuron Perceptron - Investigation of the Decision Boundaries



• 4. The decision boundary is:

$n = \mathbf{wp}+b = w_1 p_1 + w_2 p_2 + b =$
$= p_1 + p_2 - 1 = 0$

i.e. a line in the input space

• 5. Draw the decision boundary

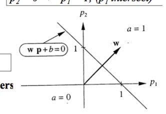
$p_1 = 0 \Rightarrow p_2 = 1;$ *(p_2 intersect)*
$p_2 = 0 \Rightarrow p_1 = 1;$ *(p_1 intersect)*

• 1. Output of the net:

$a = hardlim(\mathbf{wp}+b) =$
$= hardlim(w_1 p_1 + w_2 p_2 + b)$

• 2. Decision boundary:

$n = \mathbf{wp}+b = w_1 p_1 + w_1 p_2 + b = 0$



• 3. Let's assign values to the parameters of the perceptron (w and b):

$w_1 = 1;\ w_2 = 1;\ b = -1;$

Perceptron Learning Law – Summary

1. Initialize weights (including biases) to small random values, set *epoch=1.*
2. Choose a random example (input-output pair {p,t})from the training set
3. Calculate the response of the network for this example a (also called network activation) a= hardlim(wp+b) = 0 || 1 eg. p1= $\begin{bmatrix}1\\0\\0\end{bmatrix}$, t1=1
4. Compute the output error e=t-a
5. Update weights:
   - Add a matrix ΔW to the weight matrix W, which is proportional to the product ep^T between the error vector and the input:
     $\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{ep}^T$ from example
     evolves
   - Add a vector Δb to the bias vector b, which is proportional to the error
     $\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$
     evolves
6. Repeat steps 2-5 (by choosing another example from the training data)
7. At the end of each *epoch* check if the stopping criteria is satisfied: all examples are correctly classified or a maximum number of epochs is reached; if yes-stop, else *epoch*++ and repeat from 2.

- Limitations: • Binary Output • if training examples are linearly separable, guarantees a soln in finite # of steps. • Doesn't try to find 'optimal' line.
- (The weight vector is orgthoganal to the decision boundary.)

BACKPROPAGATION ALGORITHM

- (Powerful, Can learn Non-Linear Decision Boundaries, but difficult to Tune)
- (Uses Steepest descent algorithm for minimizing the mean square error)
- Multi-Layer NNs trained w/ backpropagation popular.
- To learn input-output (I/O) mapping - Error F() formulated (eg. ∑ of squared errors between target & actual output) & use a learning rule that Minimizes this error.

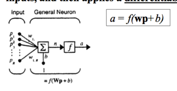
Sum of Squared Errors (E) is a classical measure of error

- E for a single training example over all output neurons
- d_i - desired, a_i - actual network output for <u>output neuron</u> *i*

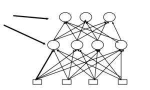
$$E = \frac{1}{2}\sum_i e_i^2 = \frac{1}{2}\sum_i (d_i - a_i)^2$$

- a Feedforward netowrk, a Fully conected network (typically), weights initialized to small rand. values.

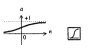
5) Each neuton (except the input neurons) computes the weighed sum of its inputs, and then applies a <u>differentiable</u> transfer function



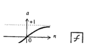
$a = f(\mathbf{wp}+b)$



- any *differentiable* transfer function f can be used, i.e. the derivative should exist for every point;
- most frequently used: sigmoid and tan-sigmoid (hyperbolic tangent sigmoid)



$a = \frac{1}{1+e^{-n}}$



$a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$

- Numerical data requires no **Encoding**, unlike nominal - typically binary encoded.
- Output Encoding:

Local encoding
- 1 output neuron
- different output values represent different classes, e.g. <0.2 – class 1, >0.8 – class 2, in between – ambiguous class (class 3)

Distributed (binary, 1-of-n) encoding - typically used in multi class problems
- Number of outputs = number of classes
- Example: 3 classes, 3 output neurons; class 1 is represented as 1 0 0, class 2 - as 0 1 0 and class 3 - as 0 0 1
- Another representation of the targets: use 0.1 instead of 0 and 0.9 instead of 1

Motivation for choosing binary over local encoding
- Provides more degree of freedom to represent the target function (n times as many weights available)
- The difference between the the output with highest value and the second highest can be used as a measure how confident the prediction is (close values => ambiguous classification)

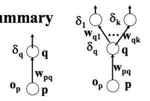
- Heuristic to start w/ : 1 **hidden layer** w/ n hidden neurons, n=(inputs+ouput_nurons)/2
- **BackPropagation (BP)** adjusts weights *backwards* by propagatong the weight ∂.
- An optimization search (hill climbing) in the weight space. Using Steepest Gradient Descent, ∏ learning rate, definies step (quickness moving downhill). May not find global min. (local instead).
- Can **adjust weights** by: 1. Incremental - after each training example is applied - liked.requires less space. 2.Batch - weights adjusted once all training examples are applied and a total error calculated.

Backpropagation Rule - Summary

$w_{pq}(t)$ - weight from node *p* to node *q* at time *t*

$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}$

$\Delta w_{pq} = \eta \cdot \delta_q \cdot o_p$ - weight change



- The weight change is proportional to the output activation of neuron p and the error δ of neuron q
- δ is calculated in 2 different ways:
  - q is an output neuron $\delta_q = (d_q - o_q)\cdot f'(net_q)$
  - q is a hidden neuron $\delta_q = f'(net_q)\sum_i w_{qi}\delta_i$ (*i* is over the nodes in the layer above *q*)

Derivative of the activation function used in neuron q with respect to the input of q (netq)

Backpropagation – Example (cont. 1)

Forward pass for ex. 1 - calculate the outputs o_6 and o_7

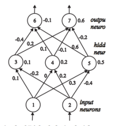
o_1=0.6, o_2=0.1, target output 1 0, i.e. class 1

• Activations of the hidden neurons:

$net_3 = o_1 * w_{13} + o_2 * w_{23} + b_3 = 0.6*0.1+0.1*(-0.2)+0.1=0.14$
$o_3 = 1/(1+e^{-net3}) = 0.53$

$net_4 = o_1 * w_{14} + o_2 * w_{24} + b_4 = 0.6*0+0.1*0.2+0.2=0.22$
$o_4 = 1/(1+e^{-net4}) = 0.55$

$net_5 = o_1 * w_{15} + o_2 * w_{25} + b_5 = 0.6*0.3+0.1*(-0.4)+0.5=0.64$
$o_5 = 1/(1+e^{-net5}) = 0.65$



• Activations of the output neurons:

•$net_6 = o_3 * w_{36} + o_4 * w_{46} + o_5 * w_{56} + b_6 = 0.53*(-0.4)+0.55*0.1+0.65*0.6-0.1=0.13$
$o_6 = 1/(1+e^{-net6}) = 0.53$

$net_7 = o_3 * w_{37} + o_4 * w_{47} + o_5 * w_{57} + b_7 = 0.53*0.2+0.55*(-0.1)+0.65*(-0.2)+0.6=0.52$
$o_7 = 1/(1+e^{-net7}) = 0.63$

Backpropagation – Example (cont. 2)

Backward pass for ex. 1

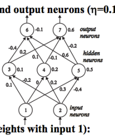
- Calculate the output errors δ_6 and δ_7 (note that d_6=1, d_7=0 for class 1)
- $\delta_6 = (d_6-o_6) * o_6 * (1-o_6)=(1-0.53)*0.53*(1-0.53)=0.12$
- $\delta_7 = (d_7-o_7) * o_7 * (1-o_7)=(0-0.63)*0.63*(1-0.63)=-0.15$

- Calculate the new weights between the hidden and output neurons (η=0.1)

$\Delta w_{36} = \eta * \delta_6 * o_3 = 0.1*0.12*0.53=0.006$
$w_{36}^{new} = w_{36}^{old} + \Delta w_{36} = -0.4+0.006=-0.394$

$\Delta w_{37} = \eta * \delta_7 * o_3 = 0.1*-0.15*0.53=-0.008$
$w_{37}^{new} = w_{37}^{old} + \Delta w_{37} = 0.2-0.008=-0.19$

Similarly for w_{46}^{new} , w_{47}^{new} , w_{56}^{new} and w_{57}^{new}



For the biases b_6 and b_7 (remember: biases are weights with input 1):

$\Delta b_6 = \eta * \delta_6 * 1 = 0.1*0.12=0.012$
$b_6^{new} = b_6^{old} + \Delta b_6 = -0.1+0.012=-0.012$

Similarly for b_7

- Every **Boolean** f() of incputs can be represented by network with a single hidden layer.
- Any continuous f() can be approximated w/ arbitrary small error by a network w/ 1 hidden layer. Any f() (inc. discontinuous) ~'able to arbitrary small error by a network w/ 2 hidden layers.
- **Overfitting**, occurs w/ **Noise**, OR when # of free (trainable) params is bigger than # of training examples. OR network been trained too long.
  Soln: • Use network that's "just large enuff", • network shouldn't have more free params than there are training egs.
  • **Validation Set** can be used to STOP traning if error increases for a pre-specified # of iterations, the weights,bias' at the min. are returned.
  • Prob w/ Validation Sets: **Small** data sets. Soln 2x: K-Fold Cross Validation, get mean number of optimum epochs. Final Run: train network on ep_mean.
- With *Gradient Descent*:
  Small Learning Rate = slow **Convergence.**
  Large Learning Rate = oscillation, overshooting of minimum. **Momentum** used to stabilize the algorithm.

For Advanced stude

First Order Linear Filter

Oscillation in the filter output *y(k)* is less than the oscillation in the filter input *w(k)*

As the momentum coefficient increases, the oscillation in the output is reduced

The average filter output is the same as the average filter input

- **Although as the momentum increases the filter output is slow to respond**

The filter tends to reduce the amount of oscillation, while still tracking the average value

- (All layers have bias' !input)
- (BP iterates till it minimizes the sum of the squared errors of the output values over all training examples).

Support Vector Machines

- 1. Maximize margin.
- 2. Transform data into a higher dimensional space where it's more likely to be linearly separable (not to high tho, overfitting danger)
- 3. Kernel Trick - Do calculations in the original, not the new higher di. space.

$$K(\mathbf{u},\mathbf{v}) = \Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (\mathbf{u}\cdot\mathbf{v})^2$$

("Kernel function of the new vector and the support vectors", instead of dot product)
(Mercer's T: restricts the class of usable f()s K)

- Scales well in high dimensions.
- Multi-Class problems need to be Transformed to 2 class problems. (Slows them down).
- Compared to Perceptrons ('linear only'), and Backpopragation NNs ('tuning,localMinima') (which SVM can reduce too), SVM's: are relatively efficient training algorithms that can learn non-linear decision boundaries.
- SVM: an optimization problem using Lagrange multipliers (/\) to maximize the margin of the decision boundary.
- Margin Width: d=2/||w||
- Linear Constraint: $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1, \forall i$ (1)

The optimization problem can be transformed into its dual

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i,j=1}^{N}\lambda_i\lambda_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j$$

Dot product of pairs of training vectors

Training vectors class

$$f = \mathbf{w}\cdot\mathbf{z} + b = \sum_{i=1}^{N}\lambda_i y_i \mathbf{x}_i\cdot\mathbf{z} + b$$

Dot product of the new vector and the support vectors

$sign(f)$ i.e. the new example belongs to class 1, if f>0 or class -1 if f<0

w: the optimal decision boundary:

$$\mathbf{w} = \sum_{i=1}^{N}\lambda_i y_i \mathbf{x}_i$$

- **Soft Margin**: allows some misclassifications. Tradeoff between margin width & #misclassifications. Soln is same as w/ hard margins but there is an upper bound C on values of /\s.

Ensemble of Classifiers

[Works when individual classifiers are highly *accurate* and *diverse* (uncorrelated, don't make same mistake. Generated by manipulating ...), & when base classifiers are good 'nuff i.e. better than random guessing] [Enlarges Hypothesis Space]

|

(Manipulating Training Data)

Bagging "Bootstrap Aggregation"

- Majority Vote
- • Creates M Bootstrap samples • each sample used to build a classifier • classify a new eg by getting majority vote
- Effective for Unstable classifiers, (small ∂s in the training set results in large ∂s in predictions, e.g. DTs, Neural Networks). May slightly decay performance of stable classifiers (e.g. k-nn).
- Applicable to regression (votes avg'd)

Boosting

- Combo of weighted votes
- • each training weight has associated weight • higher the weight, more imporant the eg during training.
- • training eg weights init = 1 • generate classifier • correctly classified egs decrease in weight + vice versa • repeat • finally combine the M hypothese, each weighed according to performance on training set
- **Adaboost** - typically performs better than individual classes.
  - If base learning algorithm is a weak learning algorithm, then AdaBoost will return a hypothesis that classifies the Training data Perfectly for Large enough M.
- Boosting fails if indi. classifiers 2 'complex'
- Boosting allows building a powerful combined classifier from Very simple ones, eg. Simple DTs generated by 1R.

Bagging vs Boosting

- Similarities: • Use voting (for classification) and averaging (for prediction) to combine the outputs of the individual learners. • Combimes models of the Same Type
- Differences: • Bagging builds individual models separately, Boosting builds them iteratively. • Bagging weighs opinions equally, Boosting weights by performance.
- Boosting, typically more Accurate.
- but Boosting more sensitive to Noise.

|

(Manipulating Attributes)

Random Forest

- Bagging & Random selection of features (this generates diversity, reducing correlation)
- Proven that RF does not overfit.
- RF Faster than Adaboost, gives comparable accuracy results.

|

(Manipulating Learning Algorithm)

- Same learning algorithms applied to same dataset but w/ ∂ params (e.g. NNs w/ ∂ architecture / params). Train them on same training data to create M classifiers, which can output a Majority Vote.

|

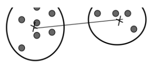
(Using a Meta-Learner)

Stacking

- Instead of voting, uses a (level-1) metalearner to learn which (level-0) base classifiers are reliable.
- Seperates training data into training, validation sets, trains level-0 classifiers, applies validation set to classifiers and use the predictions to build training data for level-1 model above
- - could use cross validation instead of training & validation sets. Slow, but allows level-0 full-use of data.
- level-0 base learners do most work, level-1 can be just a simple classifier.
- Can be applied to numeric predictions, instead of a class-value a numeric target value is attached to level-1 training eg

Unsupervised > Clustering
(don't know class labels, may not know #classes, want to group similar egs)
[Single link (MIN), Complete link (MAX), Average link (avg distance between each element in one cluster to each ele. in other) Centroid, Medoid]
{ Good Clustering produces: High Cohesion. High Separation. as measured by a distance function such as Davies-Boulding index (prefferably small) }

$$DB = \frac{1}{c}\sum_{i=1}^{c} \max_{j,j\neq i} = \left[\frac{D(x_i)+D(x_j)}{D(x_i,x_j)}\right]$$



c **– number of clusters**

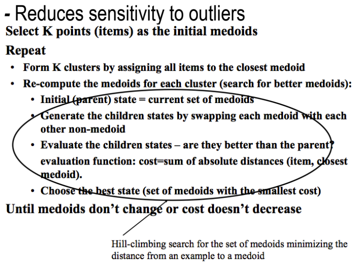
D(xi) **–mean-squared distance from the points in the cluster** ***i*** **to the center**

D(xi,xj) **– distance between the centers of cluster** ***i*** **and** ***j***

[Types: • Partitional - creates one set of clusters • Hierarchical • Density-based • Model-Based (generative) • Fuzzy Clustring]

K-Means Clustering Algorithm
- Requires k, #clusters, to be specified.
- • select K points as initial centroids • (Repeat: • form K clusters based on closest centroid • recompute centroid of each cluster) Until: Centroids don't ∂
- Issues: • data should be normalized • nominal data needs to ∂ to numeric • #epochs for convergence typically much smaller than #points, converges quickly :) • Stopping Criterion usually e.g. <1%, not ==. • SENSITIVE to choice of Initial Seeds (could run several times w/ ∂ initial centroids)
- Not sensitive to order of input egs :)
- Doesn't work well for clusters w/ non-spherical / non-convex shapes.
- Doesn't work well w/ data containing Outliners (Soln: preprocess, cut outliers)
- *Space: Modest; O((m+k)n), m=#egs, n=#attributes, k=#clusters*
- *Time: Expensive; O(tkmn), t=#iterations*
- *Not Practical for Large datasets.*
- Can be viewed as an optimization problem; find k clusters that minimize SSE.
- May find local minimum instead of global.
- Variations: • improving chances of finding global min. ; ∂ ways to initialize., allow splitting&merging of clutsers. • can be used for hierarchical clustering, w/ k=2 & recursively :|| w/ each cluster.

K-medoids
- Use medoid instead of cluster means.
- Reduces sensitivity to outliers



Select K points (items) as the initial medoids

Repeat

- **Form K clusters by assigning all items to the closest medoid**
- **Re-compute the medoids for each cluster (search for better medoids):**
  - **Initial (parent) state = current set of medoids**
  - **Generate the children states by swapping each medoid with each other non-medoid**
  - **Evaluate the children states – are they better than the parent? evaluation function: cost=sum of absolute distances (item, closest medoid).**
  - **Choose the best state (set of medoids with the smallest cost)**

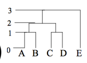
Until medoids don't change or cost doesn't decrease

Hill-climbing search for the set of medoids minimizing the distance from an example to a medoid

- Computationally expensive, not suitable for large databases: *Time: O(n(n-k)), Space: (O(n^2)-needs proximity matrix).*
- Doesn't depend on order of examples.
- Can be used when oly Distances are given and not raw data.

Nearest Neighbor Clustering Algorithm
- 1 pass algorithm, Partitional algorithm.
- Sensitive to Input Order.
- Puts ite,s in cluster of itself, single-link distance between item and new cluster, threshold t determines merging/creation.
- *Space & Time: O(n^2), n=#items*

(Hierarchical Clustering)



- Suitable for domains w/ natural nesting relationships between clusters
- Computationally Expensive, limiting applicability to high dimensional data :(*Space: O(n^2), n=#items, to store proximity distance matrix & dendrogram. Time: O(n^3), n levels, at each of them, n^2 proximity matrix must be searched & updated (reducable to O(n^2 .logn) if distances store in sorted list.*
- Not Incremental, assumes all data static.

Aglomerative Clustering
- bottom up, merges clusters iteratively (w/ single-link (min) clustering)
- • d=0 • compute proximity matrix • let each data pt be a cluster • (Increment d, merge clusters w/ dist ≤ d, update Proximity Matrix [DRAW] :||)
- using **Complete Link**, is less sensitive to Noise and Outliers than Single Link. Generating more compact clusters.

Divisive Clustering
- top-down, splits clutster iteratively. reverse of agglomerative, less popular.