AI2 - Module 3

Task 5: Learning from Data
Task 6: Coping with Incomplete Information

Lecturer: Mark Steedman
e-mail: `steedman@inf.ed.ac.uk`
Office: Room 2R.14, 2 Buccleuch Place
Notes: Copies of the lecture slides.
Activities: 18 lectures. Two practicals covering both tasks.
Required Text: Russell & Norvig, 2nd Ed., Chaps. 13-16, 18-20.
Further Reading: Tom Mitchell, *Machine Learning*, 1997.

Task 5: Learning from Data Overview

1. Introduction
2. Learning with Decision Trees.
3. Learning as Search.
4. Neural Networks: the Perceptron, multi-layer networks, back-propagation.

Text: Chapters 18-20 of Russell & Norvig

Learning from Data

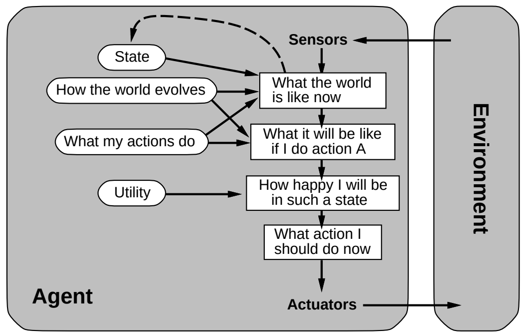
Two related tasks in applying AI techniques to any task:

1. *Represent* relevant knowledge in a computationally tractable form.

2. *Design and implement algorithms* which effectively employ that knowledge so represented to achieve the desired processing.

Learning systems actually *change* and/or *augment* the represented knowledge on the basis of experience.

A Performance Element (PE)


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

- PE selects external action
- New rules can be installed to modify PE
- What aspects of the PE can be changed by learning?

What components of PE can change?

Examples are:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves.
4. Information about the results of possible actions the agent can take.
5. Utility information indicating the desirability of world states.

6. Action-value information indicating the desirability of particular actions in particular states.
7. Goals that describe classes of states whose achievement maximises the agent's utility.

Each of these components can be learned

- Have notion of performance standard
  - can be hardwired: e.g. hunger, pain etc.
  - provides feedback on quality of agent's behaviour

Learning from Data

We will consider a slightly constrained setup known as

Supervised Learning

- The learning agent sees a set of *examples*.
- Each example is *labelled*, i.e. associated with some relevant information.
- The learning agent generalises by producing some *representation* that can be used *to predict* the associated value on unseen examples.
- The predictor is used either directly or in a larger system.

A Range of Applications

Scientific research, control problems, engineering devices, game playing, natural language applications, Internet tools, commerce.

- Diagnosing diseases in plants.
- Identifying useful drugs (pharmaceutical research).
- Predicting protein folds (Genome project).
- Cataloguing space images.
- Steering a car on highway (by mapping state to action).
- An automated pilot in a restricted environment (by mapping state to action).

- Playing Backgammon (by mapping state to its "value").
- Performing symbolic integration (by mapping expressions to integration operators).
- Controlling oil-gas separation.
- Generating the past tense of a verb.
- Context sensitive spelling correction ("I had a cake for *desert*").
- Filtering interesting articles from newsgroups.
- Identifying interesting web-sites.
- Fraud detection (on credit card activities).
- Identifying market properties for commerce.
- Stock market prediction.

Major issues for learning problems

- What *components* of the performance element are to be improved.
- What *representation* is used for the knowledge in those components.
- What *feedback* is available.
- What *representation* is used for the examples.
- What *prior knowledge* is available.

Sources and Types of Learning Systems

Is the system *passive* or *active*?

Is the system *taught* by someone else, or must it *learn for itself*?

Do we approach the system as a whole, or component by component?

Our setup of Supervised Learning implies a passive system that is taught through the selection of examples (though the "teacher" may not be helpful).

Other Types of Learning Problems

***unsupervised* learning** The system receives no external feedback, but has some internal utility function to maximise.
For example, a robot exploring a distant planet might be set to classify the forms of life it encounters there.

***reinforcement* learning** The system is trained with post-hoc evaluation of every output.
For example, a system to play backgammon might be trained by letting it play some games, and at the end of each game telling it whether it won or lost. (There is no direct feedback on every action.)

Supervised Learning

- Some part of the performance element is modelled as a function f - a mapping from possible descriptions into possible values.
- A *labelled example* is a pair (x, v) where x is the description and the intention is that the $v = f(x)$.
- The learner is given some labelled examples.
- The learner is required to find a mapping h (for hypothesis) that can be used to compute the value of f on unseen descriptions: "h approximates f"

Supervised Learning: Example

The first major practical application of machine learning techniques was indicated by Michalski and Chilausky's (1980) soybean experiment. They were presented with information about sick soya bean plants whose diseases had been diagnosed.

- Each plant was described by values of 35 attributes, such as `leafspots halos` and `leafspot size`, as well as by its actual disease.
- The plants had 1 of 4 types of diseases.

- The learning system automatically inferred a set of rules which would predict the disease of a plant on the basis of its attribute values.
- When tested with new plants, the system predicted the correct disease 100% of the time. Rules constructed by human experts were only able to achieve 96.2%.
- However, the learned rules were much more complex than those extracted from the human experts.

Supervised Learning with Decision Trees Overview

1. Attribute-Value representation of examples
2. Decision tree representation
3. Supervised learning methodology
4. Decision tree learning algorithm
5. Was learning successful ?
6. Some applications

Text: Sections 18.2-18.3 of Russell & Norvig

Supervised Learning with Decision Trees

- Some part of the performance element is modelled as a function f - a mapping from possible descriptions into possible values.
- A *labelled example* is a pair (x, v) where x is the description and the intention is that $v = f(x)$.
- The learner is given some labelled examples.
- The learner is required to find a mapping h (for hypothesis) that can be used to compute the value of f on unseen descriptions.
- h is represented by a decision tree.

Attribute-Value Representation for Examples

The mapping f decides whether a credit card be granted to an applicant. What are the important *attributes* or *properties* ?

Credit History What is the applicant's credit history like?
(values: *good*, *bad*, *unknown*)

Debt How much debt does the applicant have?
(values: *low*, *high*)

Collateral Can the applicant put up any collateral?
(values: *adequate*, *none*)

Income How much does the applicant earn?
(values: numerical)

Dependents Does the applicant have any Dependents ?
(values: numerical)

The last two attributes are numerical.
We may need to discretize them e.g. using
(values: $> 10000, < 10000$) for Income and
(values: *yes*, *no*) for Dependents.

Example (cont.)

A set of examples (descriptions and labels) can be described as in the following table.

Example	Attributes					Goal
	Credit History	Debt	Collateral	Income	Dependents	Yes/no
X1	Good	Low	Adequate	20K	3	yes
X2	Good	High	None	15K	2	yes
X3	Good	High	None	7K	0	no
X4	Bad	Low	Adequate	15K	0	yes
X5	Bad	High	None	10k	4	no
X6	Unknown	High	None	11K	1	no
X7	Unknown	Low	None	9k	2	no
X8	Unknown	Low	Adequate	9K	2	yes
X9	Unknown	Low	None	19k	0	yes

The learner is required to find a mapping h that can be used to compute the value of f on unseen descriptions.

Example (cont.)

Here the only possible values for h are $\{yes, no\}$.

Such mappings are called *Boolean functions*.

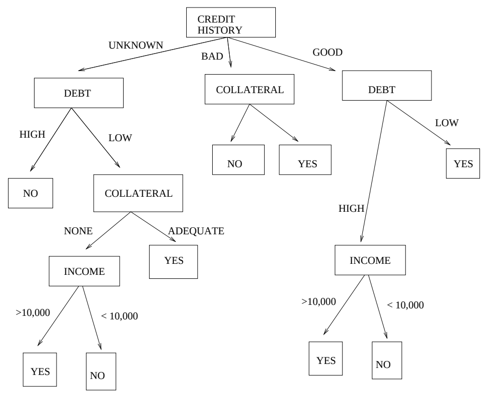
They can be used to model *concepts*, where yes means that the description refers to an object that belongs to the concept.

In our example the concept is "person who should be given credit".

Supervised Learning: Terminology

- An **example** describes the value of each of the **attributes** and the value of the **goal predicate**.
- The value of the goal predicate is called the **classification** or the **label**.
- For boolean goal predicates we identify *positive examples* and *negative examples*:
  If the classification is "yes" (or TRUE) the example is **positive**.
  If the classification is "no" (or FALSE), the example is **negative**.
- The complete set of examples is the **training set**.

Decision Trees: Example



Knowledge Representation

A **Decision Tree** takes as input an object or situation described by a set of properties (attributes) and outputs a *yes/no* decision.

- Decision trees represent Boolean Functions.
- Each internal node in the tree corresponds to a test of a value of one of the properties.
- Each **branch** of the tree is labelled with the possible value of the test.
- Each **leaf-node** specifies the Boolean value to be returned if the leaf is reached.

Knowledge Representation (cont.)

- Any path through a decision tree can be represented by a conjunction of logical tests.
- Can write an equivalent logical description of Yes leaves (or No leaves).
- Logically, a decision tree is a collection of individual implications corresponding to paths in tree ending in Yes nodes.
- Attributes correspond to propositions

E.g. $\forall m.\, credit\_story(m, good) \wedge debt(m, low) \Rightarrow given\_credit(m)$

Expressive Power of Decision Trees

- Despite the quantifier $\forall$, Decision Tree language is essentially propositional, limited to defining a new property over a single variable in terms of a logical combination of attributes of that variable—hence a decision tree cannot represent a test such as

IF credit-history(*man*, bad)
  AND income(*man*, $< 10K$)
  AND married-to(*man*, *wife*)
  AND income(*wife*, $< 10K$)
  THEN no credit

- Any Boolean Function can be written as a tree but some such functions would require a very large tree.

  The obvious translation:
  (1) at top level split on first attribute
  (2) at 2nd level split on 2nd attribute
  . . .
  may produce large trees (due to exponential growth).

- NB The ideas here can be generalised for situations where there are more than two outcomes.

Supervised Learning: How?

- The learner is required to find a mapping h that can be used to compute the value of f on unseen descriptions.

- In order to do that, machine learning programs normally *try to find a hypothesis h that gives correct classification to the training set*.

Is this reasonable?

Decision Tree Induction: Quality

(How to induce decision trees from examples)

- A *trivial* solution has one path to a leaf for each example.
- However, this just *memorises* the examples, and does not extract a pattern.
- A decision tree should be able to extrapolate from the given examples to examples it has not seen.
- A good decision tree should not only agree with the examples. It should also be *concise*.

Supervised Learning: How? (cont.)

Principle of Ockham's Razor The most likely hypothesis is the simplest one consistent with all the observations.

This general argument has been given a rigorous quantitative treatment in *computational learning theory*.

Can We Find the Smallest Decision Tree ? There is no known efficient solution to this problem! What we can do is devise heuristics that will often give us fairly small trees.

Decision Tree Learning Algorithm

- Choose a "good attribute" to put at the top level.
- Take this attribute and split up the examples into subsets, one for each value of the chosen attribute.
- For each subset that has only positive or only negative examples, attach a leaf with the corresponding value.
- Each subset that has *both* positive and negative examples, needs a new decision tree.
  $\Rightarrow$ Apply the *decision-tree-learning-algorithm* recursively.
  NB recursion has *fewer examples* and *one fewer attribute*

Applying the Algorithm: Example

Choose `Collateral`:
Value `Adequate` induces subset: {X1,X4,X8}
Value `None` induces subset: {X2,X3,X5,X6,X7,X9}

{X1,X4,X8} all labelled `yes` - attach a leaf to this branch.

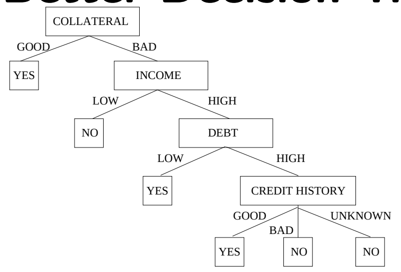
{X2,X3,X5,X6,X7,X9} has both positive and negative examples.
Choose `Income`:
Value < 10K induces subset: {X3,X7}
Value ≥ 10K induces subset: {X2,X5,X6,X9}

{X3,X7} all labelled `no` - attach a leaf to this branch.

{X2,X5,X6,X9} has both positive and negative examples.
Choose `Debt`:
Value `Low` induces subset: {X9} - attach a leaf to it
Value `High` induces subset: {X2,X5,X6}

{X2,X5,X6} has both positive and negative examples.
Choose `Credit History`:
Value `Good` induces subset: {X2} - attach a leaf to it
Value `Bad` induces subset: {X5} - attach a leaf to it
Value `Unknown` induces subset: {X6} - attach a leaf to it

A Better Decision Tree



- This is a better tree than Slide 2-8, with fewer nodes including leaf nodes
- Neither tree needs to use the Dependents property

What if . . .

- If a value induces an empty subset {}, no such example has been observed: ⇒ return a default value using the majority classification of the parent set.
- If there are no attributes left, but still positive and negative examples, then there is a problem! The algorithm returns the majority classification of the remaining examples.

Noisy Training Set

If there are no attributes left, but still positive and negative examples, then there is no decision tree that gives a correct classification to all the examples in the training set.
What can be the reason?

- Some of the data may be incorrect—the data is said to be **noisy**.
- The attributes may not give enough information to fully describe the situation.
- The domain may be truly non-deterministic.

Algorithm returns the majority classification of remaining examples.

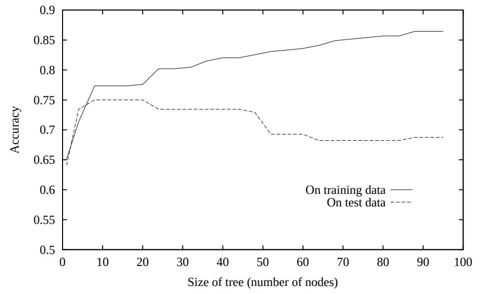
```
function DECISION-TREE-LEARNING(examples, attributes, default) returns a decision tree
  inputs: examples, set of examples
          attributes, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MAJORITY-VALUE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attributes, examples)
     tree ← a new decision tree with root test best
     for each value $v_i$ of best do
        $examples_i$ ← {elements of examples with best = $v_i$}
        subtree ← DECISION-TREE-LEARNING($examples_i$, attributes − best,
                                          MAJORITY-VALUE(examples))
        add a branch to tree with label $v_i$ and subtree subtree
     end
     return tree
```

Noise and Overfitting

Noisy examples can also lead to growing the tree too much (so as to classify noisy examples correctly).
Often, using a smaller part of the tree is better.
How can we avoid overfitting ?

- Stop growing when data split not statistically significant (small number of examples). or
- Grow full tree, then post-prune.

Effect of Overfitting



Choosing attributes

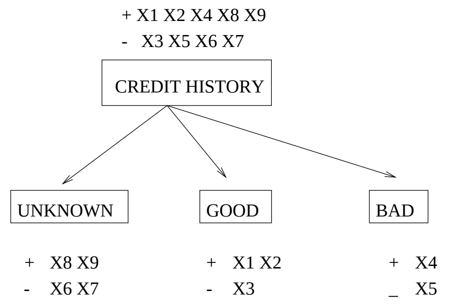
Imagine we have 100 positive and 100 negative examples.
Use notation $[100P, 100N]$ to describe this.
Imagine we have 3 attributes generating the following splits:

- A1 generates $[100P, 0N]$, $[0P, 100N]$.
- A2 generates $[70P, 30N]$, $[30P, 70N]$.
- A3 generates $[50P, 50N]$, $[50P, 50N]$.

Which attribute is the best one? and the worst?

Choosing attributes

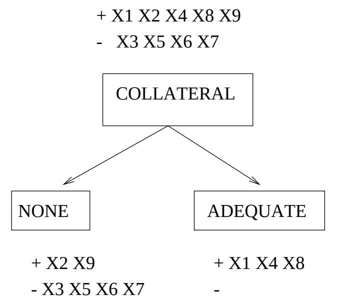
Credit History is NOT a good attribute to start with.



Has three possible outcomes, each of which has both +ve and -ve examples

Choosing attributes

Collateral gives a definite response (Yes) for 3 cases.



Revision: Information Content

The *information content* (IC) of an event is the amount of new information communicated when we learn about the event. The information content of the event $X = i$, where X is a random variable and i is the outcome, is defined as

$$IC(X = i) = \log_2 \frac{1}{p(X = i)}.$$

The definition agrees with a number of common-sense ideas regarding 'information':

Revision: Information Content

1. More surprising events provide more information.

   For example, if X is a random variable representing the current weather in Edinburgh, the information content of the event 'sunny', $p(sunny = 0.001)$, $IC = 11$ bits, is higher than that of the event 'cloudy', $p(cloudy = 0.8)$, $IC = 0.322$ bits.

2. Learning that an event that was bound to happen, did happen, provides no information.

   Such an event has a probability 1. Since $\log_2 1 = 0$, IC $= 0$ bits.

Revision: Information Content

1. Learning the outcome of related random variables reduces the information content.

   For example, the information provided by learning that a randomly selected English character is 'u' is lower when we already know that the previous letter was'q'.

Revision: Entropy

The *entropy* of a random variable, $I(X)$, is an average of the information content over the outcomes of the random variable. If a variable X has N possible outcomes,

$$I(X) = \sum_{i=1}^{N} p(X=i) \log_2 \frac{1}{p(X=i)}.$$

Entropy can be thought of as the average *uncertainty* of the random variable. Prediction is easier when the entropy is lower since we are less uncertain (on average).

Revision: Entropy

For example, the entropy of a coin is maximized when it is fair i.e. when $p(heads = 0.5)$ and $p(tail = 0.5)$, $I(X) = 0.5 \cdot \log_2 \frac{1}{0.5} + 0.5 \cdot \log_2 \frac{1}{0.5} = \log_2 2 = 1$ bit. A fair coin is also most difficult to predict.

When the coin is biased, say, $p(heads = 0.8)$ and $p(tails = 0.2)$, the entropy will be lower i.e. $0.8 \cdot \log_2 \frac{1}{0.8} + 0.2 \cdot \log_2 \frac{1}{0.2} = 0.722$ bits, and we can win money if we predict heads...

Revision: Entropy

Decision trees predict a class variable by asking questions about (hopefully correlated) attributes of an input example. The answers to these questions reduce the entropy (uncertainty) of the class assignment making prediction gradually easier at each node in the tree.

Choosing attributes - the Details

The current example set has p positive and n negative examples.
A split on attribute A with v values generates v subsets with p_i, n_i examples respectively ($i = 1 \ldots v$).

$$I(p,n) = \frac{p}{p+n}\log(\frac{p+n}{p}) + \frac{n}{p+n}\log(\frac{p+n}{n})$$

measures the entropy of the current set.
Trying to measure how far we are from having a single label.
$I(X,X) = 1$ and $I(0,X) = I(X,0) = 0$
for any value of X

$$Remainder(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(p_i, n_i)$$

where A is an attribute, measures average entropy after split.
Again, trying to measure how far we are from having a single label.

$$Gain(A) = I(p,n) - Remainder(A)$$

tries to measure the improvement obtained by the split.

$Gain(\texttt{Collateral}) = I(5,4) - (\frac{6}{9}I(2,4) + \frac{3}{9}I(3,0)) =$
$\frac{5}{9}\log\frac{9}{5} + \frac{4}{9}\log\frac{9}{4} - \frac{6}{9}\cdot\frac{2}{6}\log\frac{6}{2} - \frac{6}{9}\cdot\frac{4}{6}\log\frac{6}{4} - \frac{3}{9}\cdot\frac{3}{3}\log\frac{3}{3} - \frac{3}{9}\cdot\frac{0}{3}\log\frac{3}{0} = 0.38$

Choosing attributes - the Details

1. For each attribute A compute $Gain(A)$
2. Choose the attribute that has the maximum $Gain(A)$

This is only a heuristic.
But it works well in practice.
Other criteria for choosing attributes were developed
(including dealing with numerical attributes directly).

Decision Tree Quality

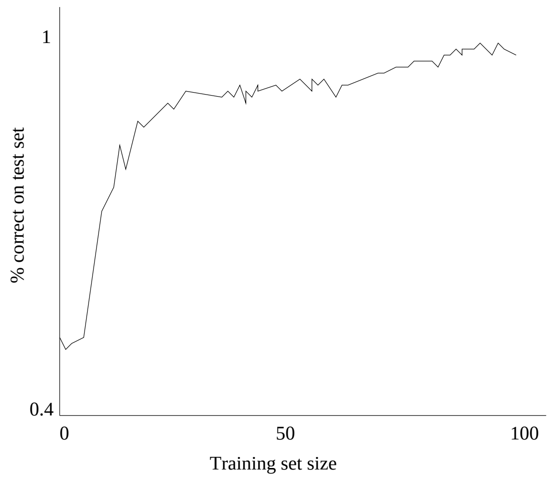
Having used a heuristic for finding a small decision tree, our hope was that this tree can be used to (correctly) compute the value of f on unseen instances.

- What if this tree is consistent with the training set but far from correct otherwise?
- Can we assess the quality of the tree?

Some answers are provided by statistical analysis.
One approach, plotting learning curves, also gives some qualitative impression.

Assessing decision trees

1. Collect a large set of examples.
2. Divide into two disjoint sets: the **training** set and the **test** set.
3. Use the learning algorithm with the training set to generate a hypothesis H.
4. Measure the percentage of examples in the test set, correctly classified by H.
5. Repeat steps 1-4 with different sizes of training sets, and different randomly selected training sets of each size.
6. Plot the training set size against the average % correct on test sets —this is called the **learning curve**.



Designing Oil Platform Equipment

- In 1986, BP used an expert system called GASOIL for designing oil-gas separation systems for offshore platforms.
- Separations are done by a system whose design depends on a large number of attributes—*relative proportions of oil, gas, water, flow-rate, pressure, density, viscosity, temperature....*
- GASOIL system contained 2500 rules!
- Building such a system by hand would have taken 10 person-years.
- Using decision-tree learning methods, the system was developed in 100 person days.

Learning To Fly

- An automated controller can be constructed by learning the correct mapping from a state of the system to the correct action.
- *Sammut et al. 1992* used this method for learning to fly a Cessna on a flight simulator, in a restricted environment.
- Data generated by watching human pilots perform a flight plan 30 times, each action taken resulted in a training examples being created.

- 90,000 examples were obtained, each described by 20 state values, and a resulting action.
- Decision tree created and converted into C code for use by the flight simulator (in a controlled manner).
- The program learns to fly, and at times flies *better* than its teachers.

Sky Image Cataloguing and Analysis

System called SkyCat by (Fayad, Djorgovski, Weir, 1996).

- Task: catalogue entries for objects in images.
- Large amounts of data collected by astronomers; objects too "faint" need special methods.
- Images split to smaller parts; various features measured on each to generate examples.
- Relatively small number of examples classified by astronomers.
- Decision Tree methods used to learn classifiers.
- Performance: 94.1% correct on test data; results used by astronomers.

Learning as Search Overview

1. Learning can be done by searching for a good hypothesis
2. Learning Logical Descriptions
3. Current-Best-Search Learning
4. Version Space Learning

Text: Section 19.1 of Russell & Norvig

Supervised Learning

The Task: The learner is required to find a mapping h that can be used to compute the value of f on unseen descriptions.

The Approach: By Ockham's Razor, the learner tries to find a *concise* representation for h that is *consistent with all the examples*.

Hypothesis Space: A *representation language* for the possible hypotheses must be fixed.

Bias: Criteria for choosing between different hypotheses (such as conciseness) are employed; this shows an a-priori *bias* of the learner to prefer some hypotheses to others.

Learning as Search

Once the hypothesis space and preference criteria are fixed this approach to learning can be viewed as a kind of **search among a set of candidate concepts**, the hypothesis space.

This is useful since we can apply our general knowledge on search strategies to learning problems!

Learning Logical Descriptions: Example

Task: For instance, we could seek a definition of when it is worth waiting at a restaurant.

Language: Logical expressions in the form of disjunctions of conjunctions, with negation only applying to individual predicates and with quantification only universal and over one variable.

Vocabulary: Unary predicates, corresponding to Boolean properties of a situation, e.g. $Hungry(r)$, $Fri/Sat(r)$.
Binary predicates, corresponding to other attributes, e.g. $Patrons(r, Full)$, $Type(r, French)$.
The **goal predicate** $WillWait(r)$.

Some possible hypotheses

$h_1 =$
$\forall r. WillWait(r) \Leftrightarrow$
$Patrons(r, Some)$
$\vee\ Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, French)$
$\vee\ Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, Burger)$

$h_2 =$
$\forall r. WillWait(r) \Leftrightarrow$
$Patrons(r, Some) \wedge Hungry(r)$
$\vee\ Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, French)$

Note that h_2 is more restrictive than h_1.

Examples

- *Examples* are again descriptions of objects.
  The label indicates whether the goal predicate holds for the object.
- An object is described using a logical expression that has one "free" argument, referring to the object (like decision tree examples).
- Normally, a subset of the language used for hypotheses is used for examples. Here we do not allow disjunctions.

The example X_1 may be described using:
$Patrons(X_1, Some) \wedge Hungry(X_1) \wedge Type(X_1, Thai) \wedge \ldots$
and the classification $WillWait(X_1)$.

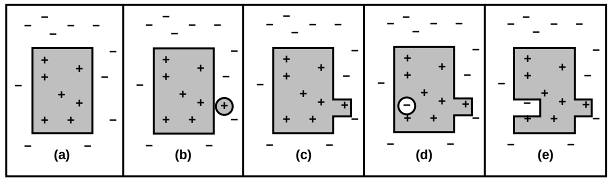
Consistency

When is an example consistent with a hypothesis?
Except in the two cases:

False positive: If $WillWait(X_1)$ follows from the hypothesis but X_1 in fact is a negative example. Eg:
Example: $Patrons(X_1, Some) \wedge Hungry(X_1) \wedge \ldots$
Classification: $\neg WillWait(X_1)$
Hypothesis: $\forall r.WillWait(r) \Leftrightarrow Hungry(r)$

False negative: If $\neg WillWait(X_2)$ follows from the hypothesis but X_2 in fact is a positive example. Eg:
Example: $Patrons(X_2, Full) \wedge \neg Hungry(X_2) \wedge \ldots$
Classification: $WillWait(X_2)$
Hypothesis: $\forall r.WillWait(r) \Leftrightarrow Hungry(r)$

Dealing with inconsistent examples

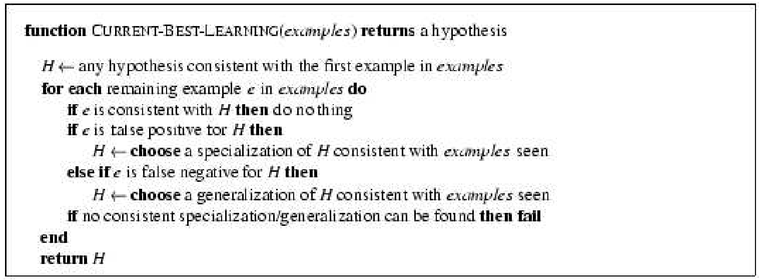


Coping with a false negative (b) requires *generalisation* (c).

Coping with a false positive (d) requires *specialisation* (e).

A learning algorithm results by performing a search over the hypothesis space using generalisation and specialisation operators.

Current Best Learning Algorithm



```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

  H ← any hypothesis consistent with the first example in examples
  for each remaining example e in examples do
      if e is consistent with H then do nothing
      if e is false positive for H then
          H ← choose a specialization of H consistent with examples seen
      else if e is false negative for H then
          H ← choose a generalization of H consistent with examples seen
      if no consistent specialization/generalization can be found then fail
  end
  return H
```

The algorithm needs to choose generalisations and specialisations (there may be several). If it gets into trouble, it has to backtrack to an earlier decision or otherwise it fails.

Generalising a Description

- Remove a conjunct, e.g.
  change $[Hungry(r) \wedge Patrons(r, Full)]$ to $[Hungry(r)]$
- Add a disjunct, e.g.
  change $[Hungry(r)]$ to $[Hungry(r) \vee Patrons(r, Full)]$
- Replace a predicate/value by a more general one, e.g.
  change $[Type(r, French)]$ to $[Type(r, European)]$

  Here we assumed that the learner knows the semantics of $French$ and $European$.

Specialising a Description

- Add a conjunct. e.g.
  change $[Hungry(r)]$ to $[Hungry(r) \wedge Patrons(r, Full)]$
- Remove a disjunct, e.g.
  change $[Hungry(r) \vee Patrons(r, Full)]$ to $[Hungry(r)]$
- Replace a predicate/value by a more specific one, e.g. change $[Hungry(r)]$ to $[VeryHungry(r)]$

  Here we assumed that the learner knows the semantics of $Hungry()$ and $VeryHungry()$.

CBL Example

Current Hypothesis	Example	Action
$\forall r.WillWait(r) \Leftrightarrow false$	X_1 false negative	Add disjunct
$\forall r.WillWait(r) \Leftrightarrow Alternate(r)$	X_2 false positive	Add conjunct
$\forall r.WillWait(r) \Leftrightarrow [Alternate(r) \wedge Patrons(r, Some)]$	X_3 false negative	Remove conjunct
$\forall r.WillWait(r) \Leftrightarrow Patrons(r, Some)$	X_4 false negative	Add disjunct
$\forall r.WillWait(r) \Leftrightarrow [Patrons(r, Some) \vee (Patrons(r, Full) \wedge Fri/Sat(r))]$		

There were, of course, many other possibilities.

Problems with CBL

Although CURRENT-BEST-LEARNING has been popular, it has a number of problems:

- It needs to store all encountered examples (to check that changes are consistent with all of them).
- Checking over all encountered examples when a change is made is expensive.
- It is difficult to find good heuristics.
  Real hypothesis spaces are large or infinite.
  Backtracking may not quickly reconsider the right decision and so may take a long time.

Least Commitment Search

- LCS Avoids making arbitrary decisions that might end up wrong.
- The set of hypotheses consistent with the examples seen so far is called the **version space**.
- The algorithm works by successively eliminating inconsistent hypotheses from the version space. Hence called the **candidate elimination algorithm**.

Candidate Elimination Algorithm

(Version Space Learning)

```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: V, the version space: the set of all hypotheses

  V ← the set of all hypotheses
  for each example e in examples do
      if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  end
  return V
```

```
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

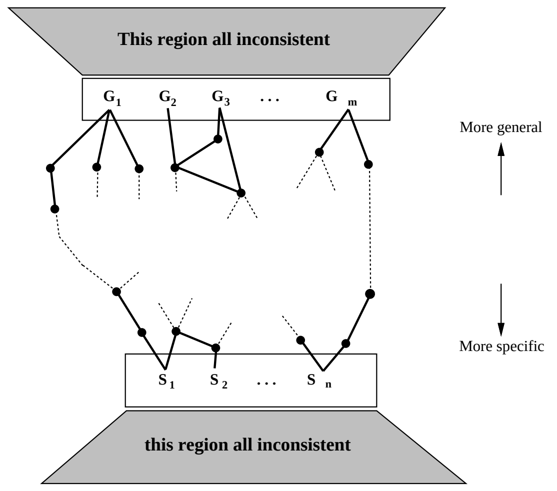
Compact Representation of Version Spaces

Imagine you had to represent all the numbers between 1 and 2. You could represent this set just by specifying the boundaries [1,2]. This works because numbers are *ordered*.

Hypotheses are also ordered, in terms of *specificity*.

For instance, for hypotheses given on Slide 3-5, h_1 is more general (less specific) than h_2.

Representation of a version space in terms of its boundaries uses two sets: S (most specific) and G (most general).


This region all inconsistent
G_1 G_2 G_3 … G_m
More general
More specific
S_1 S_2 … S_n
this region all inconsistent

Updating the Version Space

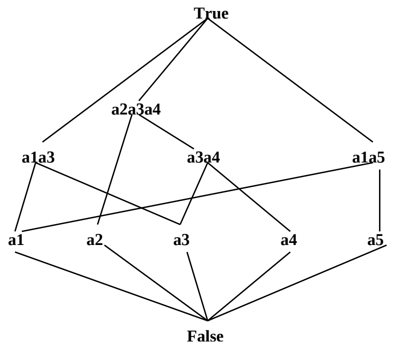
Given the current version space (constructed from examples seen so far) and a new example, the sets S and G are updated to construct the new version space. Four cases arise:

1. False negative for S_i: $\Rightarrow$ Further generalise S_i.
2. False positive for S_i: $\Rightarrow$ Remove S_i from the S set.
3. False positive for G_j: $\Rightarrow$ Further specialise G_j.
4. False negative for G_j: $\Rightarrow$ Remove G_j from the G set.

We initialize the version space as $S_0 = \{\mathbf{false}\}$, $G_0 = \{\mathbf{true}\}$

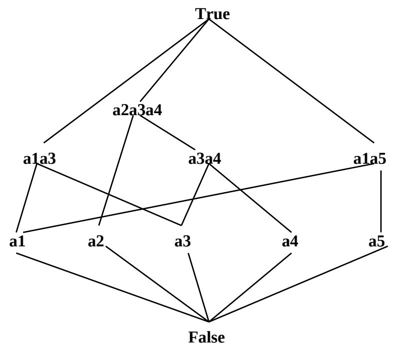
VS Example

- Possible examples: a1, a2, a3, a4, a5. NB in order to simplify the description we are not giving the representation of examples here but just their names.
- All "good" hypotheses are marked as nodes on the graph.
- Each hypothesis description shows the examples that are positive for it.
- Edges represent generalisations and specialisations.



VS Example

S	G	Example
false	true	a5 negative
false	a1a3,a2a3a4	a1 negative
false	a2a3a4	a4 positive
a4	a2a3a4	a2 negative
a4	a3a4	a3 positive
a3a4	a3a4	CONVERGENCE



Termination

Learning with version spaces can terminate in three ways:

1. We get to a single concept in the version space.
   $\Rightarrow$ Return that as the answer.

2. The version space becomes empty, indicating that no hypotheses (in the given space) are consistent with all the examples.
   $\Rightarrow$ version space "collapses"

3. We run out of examples, still with multiple elements in the version space. Not clear how to classify new examples. Several possibilities:
   (a) Select an element of the version space at random and use it.
   (b) Take the majority vote of all elements in the version space.
   (c) When classifying a new object, allow "maybe" as well as "yes" or "no" (in case not all hypotheses agree).

   We must be able to perform this efficiently.

Version Space Learning

+ Is a *complete* search method (unlike heuristic methods for DTs).
+ Is an *incremental* method (no need to see all examples at once).
+ Has been shown by Gunter et al. (1997 AIJ) to be closely related to Assumption-Based Truth Maintenance (ATM)
– Is *not* tolerant to noise (version space will collapse).
– Will not work if arbitrary disjunctions are allowed in the concept language (generalisation then simply remembers the positive examples by rote).
– May not be efficient if the sets S and G are large.

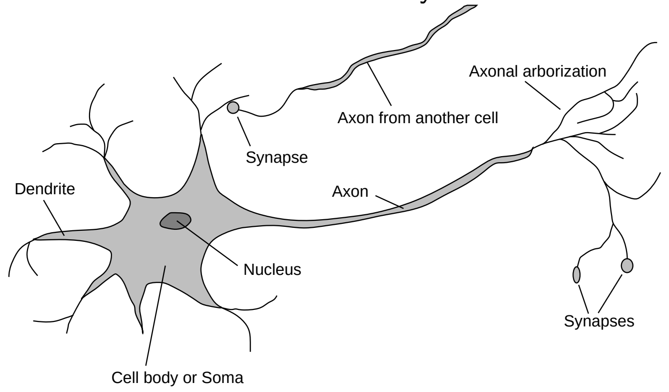
Neural Networks — Overview

1. Overview, real neurons, basic neural computing units
2. Neural Networks as a Representation
3. Perceptrons and their learning algorithm
4. Useful mathematical concepts (gradient descent)
5. Multi-layer Perceptrons, back-propagation
6. Some Applications

Text: sections 20.5 of Russell & Norvig

Real neurons

The fundamental unit of all nervous system tissue is the *neuron*



A neuron consists of

- a **soma**, the cell body, which contains the cell nucleus
- **dendrites**: input fibres which branch out from the cell body
- an **axon**: a single long (output) fibre which branches out over a distance that can vary between 1cm and 1m
- **synapse**: a connecting junction between the axon and other cells

Real Neurons - Properties

- Each neuron can form synapses with anywhere between 10 and 10^5 other neurons
- Signals are propagated at the synapse through the release of chemical transmitters which raise or lower the electrical potential of the cell
- When the potential reaches a **threshold value**, an **action potential** is sent down the axon
- This eventually reaches the synapses and causes potentiation of the subsequent neurons

- Synapses can be **inhibitory** (lower the post-synaptic potential) or **excitatory** (raise the post-synaptic potential)
- Synapses can also exhibit long term changes of strength in response to the pattern of stimulation

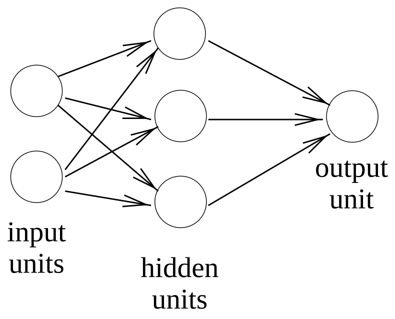
Artificial Neural Networks

- A neural network is composed of a number of units connected together by links. Each link has an associated numeric weight
- Input and output units are connected to the environment
- Weights are the long term storage, learning involves changing the weights
- Learning modifies the weights so as to try to make the output value(s) correct given the input values.

Recurrent networks allow cycles on directed graph i.e. links can form arbitrary topologies.

We will concentrate on acyclic networks, known as **feed-forward** networks.

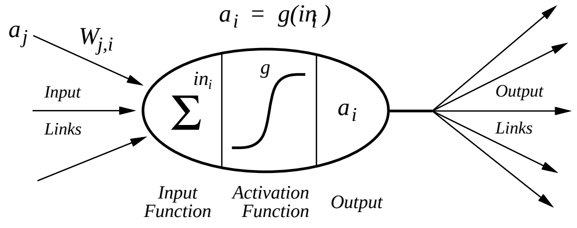
- links are unidirectional
- network computes a function of the input values that depends on the weight settings
- no internal state other than weights themselves



The formal neuron

- Each unit has input links from other neurons, and a current **activation** level.
- Each unit updates its activation (output) using a local computation based on its inputs without any need for global control over the set of units as a whole.
- This formal model is a gross simplification of the detailed function of a neuron.

The formal neuron



- The neuron computes the total weighted input to the neuron

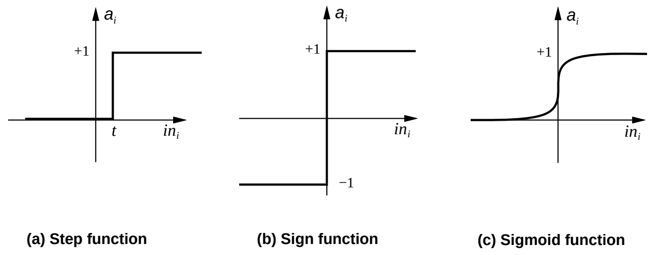
$$in_i = \sum_j W_{j,i} a_j = \mathbf{W}_i \cdot \mathbf{a}$$

- $W_{j,i}$ is weight on link from neuron j to neuron i, $\mathbf{W_i}$ is the vector of weights leading *into* unit i, and $\mathbf{a}$ is the vector of input values.
- computing in_i is a *linear* operation
- A *non-linear* component called the *activation function* g transforms the sum of weighted inputs into the final output value a_i

$$a_i \leftarrow g(in_i) = g(\mathbf{W}_i \cdot \mathbf{a})$$

Note: operation in $\mathbf{W}_i \cdot \mathbf{a}$ is vector **dot product**
- We can use different mathematical functions for g
- Usually, all units (neurons) in network have the same activation function

Activation Functions



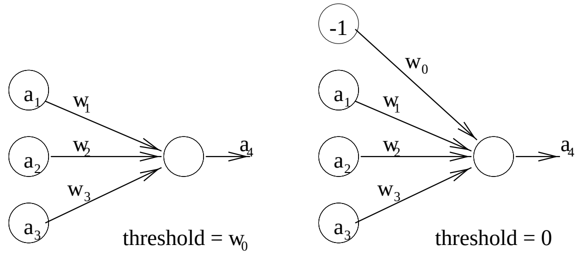
$$\mathrm{step}_t(z) = \begin{cases} 0 & \text{if } z < t \\ 1 & \text{if } z \geq t \end{cases}$$

The step and sign functions are examples of the **threshold activation** function.

Thresholds

- For a step or sign function, with *threshold value* t: if $in_i \geq t$ the unit fires. The threshold thus corresponds to the minimum total weighted input necessary to cause the neuron to fire.
- Convenient to replace the threshold with an extra weight W_0, the **bias weight**, from unit 0 which always has activation $a_0 = -1$. Since $in_i \geq t \Leftrightarrow in_i - t \geq 0$

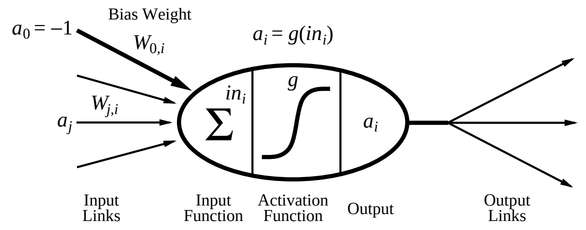
$$a_i = \mathrm{step}_t\left(\sum_{j=1}^{n} W_{j,i} a_j\right) = \mathrm{step}_0\left(\sum_{j=0}^{n} W_{j,i} a_j\right)$$



- Trick also useful for other activation functions, e.g. the sigmoid
- This makes the learning algorithm simpler as only weights need to be adjusted rather than weights *and* threshold

Alternative Representation of Formal Neuron

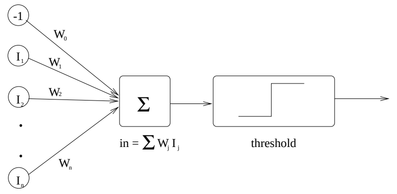
This leads to a modified mathematical model of the formal neuron where the bias weight $W_{0,i}$ is connected to a fixed input $a_0 = -1$:



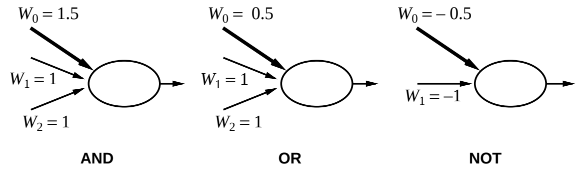
The Perceptron

A perceptron is a **single-layer feed-forward neural network**. Consider an example in which the activation function is a **step** function:

- Set $I_0 = -1$
- Unit fires when $\sum_{j=0}^{n} W_j I_j = \sum_{j=1}^{n} W_j I_j - W_0 \geq 0$
- W_0 is the *threshold*: the unit fires when $\sum_{j=1}^{n} W_j I_j \geq W_0$



Computing Boolean Functions with Perceptrons



Units with a (step) threshold activation function can act as logic gates, given appropriate input and bias weights.

AND-gate truth table

Bias	input		output
a0	a1	a2	
-1	0	0	0
-1	0	1	0
-1	1	0	0
-1	1	1	1

AND $= \text{step}_{1.5}(1 \cdot a_1 + 1 \cdot a_2) = \text{step}_0(1.5 \cdot -1 + 1 \cdot a_1 + 1 \cdot a_2)$

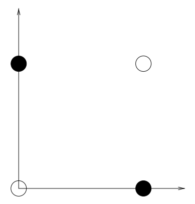
However, single-layer feed-forward nets (i.e. perceptrons) cannot represent *all* Boolean functions

Some Geometry

- In 2 dimensions $w_1 x_1 + w_2 x_2 - w_0 = 0$ defines a line in the plane.
- In higher dimensions $\sum_{i=1}^{n} w_i x_i - w_0 = 0$ defines a hyperplane.
- The decision boundary of a perceptron is a **hyperplane**.
- If a hyperplane can separate all outputs of one type from outputs of the other type, the problem is said to be **linearly separable**.

XOR is not linearly separable

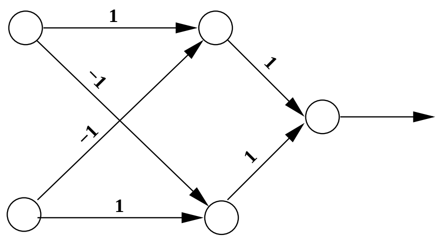
	I_1	I_2	$XOR(I_1, I_2)$
(a)	0	0	0
(b)	0	1	1
(c)	1	0	1
(d)	1	1	0



- Function as 2-dimensional plot based on values of 2 inputs
- black dot: $XOR(I_1, I_2) = 1$ and white dot: $XOR(I_1, I_2) = 0$
- Cannot draw a line that separates black dots from white ones

Multilayer Neural Network

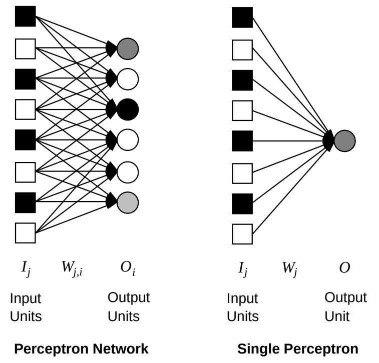
Can represent XOR using a network with two inputs, a hidden layer of two units, and one output. A step (threshold) activation function is used at each unit (threshold weights (not shown) are all zero). Many architectures possible, this is an AND-NOT OR AND-NOT network.



In fact, any Boolean function can be represented, and any bounded continuous function can be approximated.

Single Layer — Multiple Outputs

- Each output unit is independent of the others; each weight only affects one output unit.
- We can limit our study to single-output Perceptrons.
- Use several of them to make a multi-output perceptron.



Supervised Learning: How?

- The learner sees labelled examples $e = (I_e, T_e)$ such that $f(I_e) = T_e$.
- The learner is required to find a mapping h that can be used to compute the value of f on unseen descriptions.
- In order to do that, machine learning programs normally try to find a hypothesis h that gives correct classification to the training set (or otherwise minimises the number of errors).

Learning Perceptrons — Basic Idea

- **Important Note:** We assume a threshold activation function, namely a step function, in the next few slides.
- Start by assigning arbitrary weights to $\mathbf{W}$.
- On each example $e = (\mathbf{I}, T)$:
  classify e with current network:
  $O \leftarrow \mathrm{step}_0(\mathbf{W} \cdot \mathbf{I}) = \mathrm{step}_0(\sum W_i I_i)$
  if $O = T$ (correct prediction) do nothing.
  if $O \neq T$ change $\mathbf{W}$ "in the right direction".

But what is "the right direction" ?

- if $T = 1$ and $O = 0$ we want to increase $\mathbf{W} \cdot \mathbf{I} = \sum W_i I_i$
  Can do this by assigning $\mathbf{W}^{new} = \mathbf{W} + \eta \mathbf{I}$
  since $\mathbf{W}^{new} \cdot \mathbf{I} = \sum W_i^{new} I_i = \sum W_i I_i + \eta \sum I_i I_i > \sum W_i I_i$
- Amount of increase controlled by parameter $0 < \eta < 1$
- if $T = 0$ and $O = 1$ we want to decrease $\mathbf{W} \cdot \mathbf{I} = \sum W_i I_i$
  Can do this by assigning $\mathbf{W}^{new} = \mathbf{W} - \eta \mathbf{I}$
  since $\mathbf{W}^{new} \cdot \mathbf{I} = \sum W_i^{new} I_i = \sum W_i I_i - \eta \sum I_i I_i < \sum W_i I_i$
- In both cases we can assign $\mathbf{W}^{new} = \mathbf{W} + \eta \mathbf{I}(T - O)$

```
**function** perceptron-learning(examples) **returns** a perceptron hyp.
    network ← a network with randomly assigned weights
    **repeat**
        **for each** $e$ **in examples do**
            $O \leftarrow$ perceptron-output(network,$\mathbf{I}_e$)
            $T \leftarrow$ required output for $\mathbf{I}_e$
            update weights in network based on $\mathbf{I}_e$, $O$ and $T$
            $\mathbf{W} \leftarrow \mathbf{W} + \eta\ \mathbf{I}_e\ (T - O)$
        **end**
    **until** all examples correctly predicted or other stopping criterion
**return** NEURAL-NET-HYPOTHESIS(network)
```

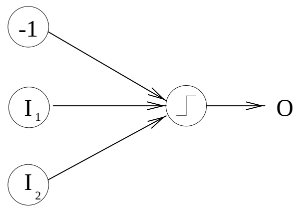
Perceptron algorithm with step (threshold) activation function.

Perceptron Learning Algorithm

- $0 < \eta < 1$ is known as the **learning rate**, other symbols e.g. α, ϵ used by different authors
- Rosenblatt (1960) showed that the PLA converges to $\mathbf{W}$ that classifies the examples correctly (if this is possible).
- PLA behaves well with noisy examples.
- Note that PLA given above is an *incremental* algorithm; *batch* version also possible.

PLA:Example

- Assume that output O=1, and target is T=0, $\Rightarrow$ T-O=-1
- $W_0 \leftarrow W_0 + \eta * (-1) * (-1)$
- $W_1 \leftarrow W_1 + \eta * I_1 * (-1)$
- $W_2 \leftarrow W_2 + \eta * I_2 * (-1)$

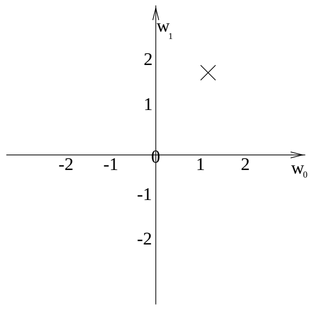


Understanding the PLA

- Will look at a simpler problem: *learning a linear model* $y = \mathbf{W} \cdot \mathbf{I} = \sum W_i I_i$.
- For example: $y = w_0 + w_1 x$ the model has unknown parameters w_0, w_1.
- We want to determine these on the basis of some training data.
- We can then use the model to *generalize*, i.e. to predict outputs for new inputs.
- Trivial without noise but not so with noise.

Weight space

- Any linear model has a particular number of weights (parameters).
- Particular values for these parameters can be thought of as a point in weight space.
- Can measure the cumulative error as function of parameters.



Error Function

- Use an error function, $E = \frac{1}{2}\sum_e (T_e - O_e)^2$
- Has desirable property that $E = 0$ if $T_e = O_e$ for all e.
- So, $E = 0$ can be obtained if there is no noise.
- Otherwise $E > 0$ but we can try to minimise it.
- Each value of $\mathbf{W}$ generates a value of E, call it $E(\mathbf{W})$. We are looking for $\mathbf{W}$ that minimises $E(\mathbf{W})$.

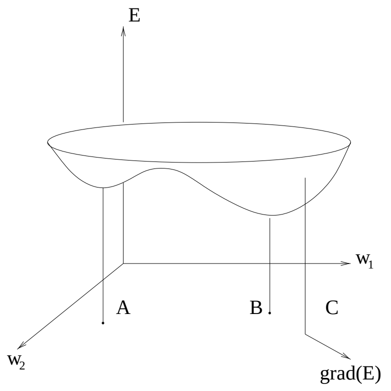
Error Function — Example

- Fit $y = w_0 + w_1 x$ with 3 examples (the form is (x, y)): $(1, 9)$, $(2, 7)$, $(3, 4)$.
- Error is

$$
\begin{aligned}
E(\mathbf{W}) =& \frac{1}{2}\left\{(9 - w_1 - w_0)^2 + (7 - 2w_1 - w_0)^2 + (4 - 3w_1 - w_0)^2\right\} \\
=& 1.5w_0^2 + 7w_1^2 - 20w_0 - 35w_1 + 6w_0w_1 + 73
\end{aligned}
$$

Error Surface

- "Best" model corresponds to the the lowest point on the error surface.
- Our problem has a single minimum (can be obtained analytically).
- General, non-linear problems, have a difficulty with local minima.
- $\Rightarrow$ Search using methods that "go downhill" on the error surface.



Gradient descent

- If $E(\mathbf{W})$ is the error function, then the derivative $\frac{\partial E}{\partial W_i}$ measures the slope of the error surface in the W_i direction.
- This is summarised in vector notation as: $\mathbf{g} = \frac{\partial E}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial E}{\partial W_0} \\ \vdots \\ \frac{\partial E}{\partial W_n} \end{pmatrix}$
- Locally, if we are at point $\mathbf{W}$, the quickest way to decrease E is to take a step in the direction $-\mathbf{g}$.
- Given a formula for E, $\frac{\partial E}{\partial W_i}$ can often be derived with straightforward algebra.

Gradient Descent Algorithm

```
Initialize $\mathbf{W}$
while $E(\mathbf{W})$ is unacceptably high
    calculate $\mathbf{g} = \frac{\partial E}{\partial \mathbf{W}}$
    $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{g}$
end while
return $\mathbf{W}$
```

As in the PLA η is the **learning rate**.

Here updates are in batch (error computed on all examples); incremental version as in PLA also possible.

Example (continued)

- Fit $y = w_0 + w_1 x$ with 3 examples (the form is (x, y)): $(1, 9)$, $(2, 7)$, $(3, 4)$.
- Error is $E(\mathbf{W}) = 1.5w_0^2 + 7w_1^2 - 20w_0 - 35w_1 + 6w_0w_1 + 73$
- Partial derivatives:
  $\frac{\partial E}{\partial w_0} = 3w_0 - 20 + 6w_1$
  $\frac{\partial E}{\partial w_1} = 14w_1 - 35 + 6w_0$
- Initial guess: $w_0 = 0, w_1 = 0$; $\eta = 0.1$

- Iteration:
  $w_0 \leftarrow w_0 - 0.1(3w_0 - 20 + 6w_1)$
  $w_1 \leftarrow w_1 - 0.1(14w_1 - 35 + 6w_0)$
- $w_0 \leftarrow -0.1 \cdot -20 = 2$; $w_1 \leftarrow -0.1 \cdot -35 = 3.5$
- $w_0 \leftarrow 2 - 0.1(6 - 20 + 21) = 1.3$
  $w_1 \leftarrow 3.5 - 0.1(49 - 35 + 12) = 0.9$
- $\ldots$
- **Convergence** $w_0 = 11.66$; $w_1 = -2.50$
  changes to weights $< 10^{-4}$ after 226 iterations

Linear Models Again

- $O = \mathbf{W} \cdot \mathbf{I} = \sum W_i I_i$
- Use an error function, $E = \frac{1}{2}\sum_e (T_e - O_e)^2$
- $\frac{\partial E}{\partial W_i} = -(T_e - O_e)\frac{\partial O_e}{\partial W_i} = -(T_e - O_e)I_i$
- So the update is: $W_i \leftarrow W_i + \eta(T_e - O_e)I_i$.
- PLA uses the same update!
  The formula however does not correspond to its O function
  (the derivative of step is not very useful).
  NB This just says that it does not correspond to our analysis not that it is a bad algorithm.

Problems with Gradient Descent

- Need to choose η (Too small $\Rightarrow$ too slow; Too big, unstable).
- Local minima.
- Plateaux.
- Gradient descent is a "hill-climbing" (actually "hill-descending") search: An iterative improvement approach that always moves in the direction that is most promising *locally*.
  No memory, no global perspective, no ability to backtrack.
- **But** it works well in many cases.

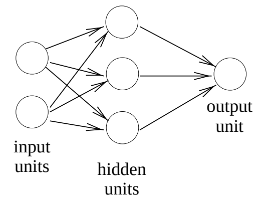
Perceptron Learning Algorithm (Version 2)

```
function perceptron-learning(examples) returns a perceptron hypothesis
inputs: examples, a set of examples, each with input $\mathbf{x} = x_1, ..., x_n$ and output $y$.
        network, a perceptron with weights $W_j$, $j = 0...n$, and activation func. $g$
    repeat
        for each $e$ in examples do
            $in \leftarrow \sum_{j=0}^{n} W_j x_j[e]$
            $Err \leftarrow y[e] - g(in)$
            $W_j \leftarrow W_j + \eta \times Err \times g'(in) \times x_j[e]$
        end
    until all examples correctly predicted or other stopping criterion
return NEURAL-NET-HYPOTHESIS(network)
```

Gradient descent learning algorithm for perceptrons, with differentiable activation function g. For threshold perceptrons, $g'(in)$ is omitted from the weight update leading to previously seen algorithm (version 1).

Multi-Layer Neural Networks

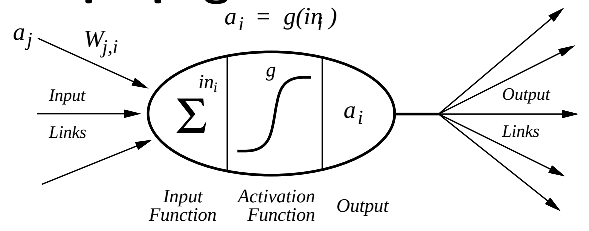
- More expressive than Perceptrons.
- Two issues for learning:
  (1) what size and structure to use
  (2) how to update the weights
- For (2) we can use the same scheme as before.
- But we need a differentiable activation function, in order to use the mathematical tools.



Training a MLP using Gradient Descent

- Assume network structure chosen.
- Weights are adjusted to reduce the error $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial E}{\partial \mathbf{W}}$
  or for individual weights $W_{ji} \leftarrow W_{ji} - \eta \frac{\partial E}{\partial W_{ji}}$
  NB change in notation: W_{ji} connects unit j to unit i
- Need differentiable activation function (so step will not do).
- Calculation like before for the output layer
  but we do not have the output value for hidden layers.

Backpropagation - Basic Idea



- Calculate $\Delta_i = -\frac{\partial E}{\partial in_i}$ for each unit in the network
- **Then** $\frac{\partial E}{\partial W_{ji}} = \frac{\partial E}{\partial in_i} \frac{\partial in_i}{\partial W_{ji}} = -\Delta_i a_j$
  because $\frac{\partial in_i}{\partial W_{ji}} = \frac{\partial [\sum_k W_{ki} a_k]}{\partial W_{ji}} = a_j$

The Update Rule

- Threshold Perceptron $W_{ji} \leftarrow W_{ji} + \eta I_j (T_i - O_i)$
- Backpropagation $W_{ji} \leftarrow W_{ji} + \eta a_j \Delta_i$
- Q: How do we compute the Δs ?
- A: Compute them first for the output units, and then propagate them backwards through the net, from outputs to inputs
- Hence the name **backpropagation**

Computing Δ_i for Output Unit

- Here error refers to a single example.
- For an output unit indexed by i

$$\Delta_i = -\frac{\partial E}{\partial in_i} = -\frac{\partial[\frac{1}{2}(T_e - O_e)^2]}{\partial in_i} = g'(in_i)(T_i - O_i)$$

where g' is the derivative of g and $O_i = g(in_i)$

Computing Δ_i for Hidden Unit

- in_i contributes to E only through the outputs
  $\Rightarrow$ only through nodes connected to the output of node i.
  Let k range over these nodes.

$$\Delta_i = -\frac{\partial E}{\partial in_i} = -\sum_k \frac{\partial E}{\partial in_k} \frac{\partial in_k}{\partial a_i} \frac{\partial a_i}{\partial in_i} = \sum_k \Delta_k W_{ik}\, g'(in_i)$$

- Reorganising: $\Delta_i = g'(in_i) \sum_k W_{ik} \Delta_k$
- Calculating a Δ value only requires Δs from further forward in the network.

Backpropagation: summary

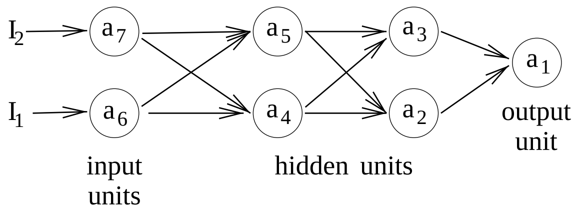
- Compute Δ values for the output units: $\Delta_i = g'(in_i)(T_i - O_i)$
- Starting with the output layer, keep propagating the Δ values back to the previous layer, until the input layer is reached.
- This is computed by $\Delta_i = g'(in_i) \sum_k W_{ik} \Delta_k$
- Update each weight using gradient descent $W_{ji} \leftarrow W_{ji} + \eta a_j \Delta_i$

```
function backpropagation-learning(examples) returns network
    network ← a network with randomly assigned weights
    repeat
        for each e in examples do
            O ← neural-network-output(network,e)
            T ← required output for e
            compute error and Δs for unit in output layer
            for each subsequent layer in the network
                compute the Δs for units in the layer
            end
            update all weights
        end
    until network has converged
return network
```

Activation functions

- The activation function should be differentiable everywhere.
- This rules out the *sign* or *step* functions.
- The sigmoid function $g(z) = \frac{1}{1+e^{-z}}$ is a common choice for multilayer networks
- It “approximates” a step function.
- Also has nice property that $g'(z) = g(z)(1 - g(z))$ so the computations are simple.

Back Propagation - Example



- Assume
  $\eta = 0.1$
  $w_{21} = w_{31} = 1$, $w_{42} = w_{43} = w_{52} = w_{53} = 0.6$
  $w_{64} = w_{65} = w_{74} = w_{75} = 1$

- Example (I_1,I_2) = (2,3) and $T = 0$
- First Step: compute in_i, a_i, $g'(in_i)$
  - $in_4 = 1 \cdot 2 + 1 \cdot 3 = 5$; $a_4 = \frac{1}{1+e^{-5}} = 0.993$; $g'(in_4) = 0.993 \cdot 0.007 = 0.007$
  - $in_5 = 5$; $a_5 = 0.993$; $g'(in_5) = 0.007$
  - $in_2 = 0.6 \cdot 0.993 + 0.6 \cdot 0.993 = 1.192$; $a_2 = 0.767$; $g'(in_2) = 0.179$
  - $in_3 = in_2 = 1.192$; $a_3 = a_2 = 0.767$; $g'(in_3) = 0.179$
  - $in_1 = 1 \cdot 0.767 + 1 \cdot 0.767 = 1.534$; $a_1 = \frac{1}{1+e^{-1.534}} = 0.823$; $g'(in_1) = 0.823 \cdot 0.177 = 0.146$

  The output of the network, $a_1 = 0.823$, is far from the true output $T = 0$.

- Second step: compute Δ
  - $\Delta_1 = g'(in_1)(T - a_1) = 0.146(0 - 0.823) = -0.120$
  - $\Delta_2 = g'(in_2)w_{21}\Delta_1 = 0.179 \cdot 1 \cdot -0.120 = -0.021$
  - $\Delta_3 = \Delta_2$
  - $\Delta_4 = g'(in_4) \cdot [w_{42}\Delta_2 + w_{43}\Delta_3] = -0.000176$
  - $\Delta_5 = \Delta_4$
- Third Step: update weights
  - $w_{64} \leftarrow w_{64} + \eta a_6 \Delta_4 = 1 + 0.1 \cdot 2 \cdot (-.000176) = 0.9999648$
  - $\ldots$
  - $w_{42} \leftarrow w_{42} + \eta a_4 \Delta_2 = 0.6 + 0.1 \cdot 0.993 \cdot (-0.021) = 0.5579$
  - $\ldots$

- $w_{21} \leftarrow w_{21} + \eta a_2 \Delta_1 = 1 + 0.1 \cdot 0.767 \cdot (-0.120) = 0.9908$
- $\ldots$

- We are now ready to handle the next example

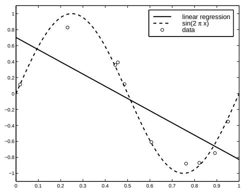
Using backprop to train a MLP

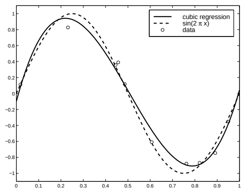
- Each pass over all examples is called an **epoch**.
- With m examples and $|\mathbf{W}|$ weights, each epoch takes $O(m|\mathbf{W}|)$ time.
- How many epochs are needed? Perhaps lots! Have to choose a good η
- Local optima can be a problem; use different starting points in weight space and (?) find the best performance
- How do you choose a network architecture (i.e. number of layers, number of units in each layer)?

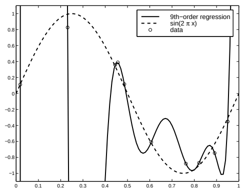
Generalization

- We care about the **generalization** performance of a network, i.e. its predictions on *new* inputs.
- Use strategy similar to decision trees.
- Divide into two disjoint sets: the **training** set and the **test** set.
- Use the learning algorithm with the training set.
- Estimate generalization error using the test set.

Choosing the Right Model







Choice is crucial for getting good generalisation.
Model too complex: "overfitting"
Model too simple: "underfitting"

Choosing a model

- Pick a number of different models and find out which one has the best test error.
- Different models may include:
  (1) different architectures,
  (2) networks with the same architecture but different starting weight vectors.
- The above method is the standard one, but this is an active research area ...

ALVINN

Autonomous Land Vehicle in a Neural Network (Pomerleau, 1993)

- Task: steer a vehicle along a single lane on a highway by observing the performance of a human driver
- The vehicle (Chevy van) is fitted with computer controlled steering, acceleration and braking
- On-board sensors include colour stereo video, laser rangefinder, radar, inertial navigation system
- Researchers ride along inside the vehicle to monitor the progress of the vehicle and network.

The ALVINN Architecture

- Input: 30×32 pixel array
- Output: 30 units, each corresponding to a steering direction.
- 5 hidden units, fully connected to inputs and outputs
- Method: Map single video frames to steering direction (pure reactive agent)
  – collect data from human drivers (5 minutes)
  – Train network, then ready to drive
  – problem: human drivers are too good!
  – solution: synthesize data from slightly off course

ALVINN: Results

+ Has driven at up to 70 mph for 90 miles on public highways near Pittsburgh
+ Has driven at normal speeds on single lane dirt roads, paved bike paths and two lane suburban streets
- Not able to drive on a road type for which it has not been trained
- Not very robust with respect to changes in lighting conditions.

Playing Backgammon

Tesauro (1990, 1995)

- A conventional program generates possible moves and presents them to the network for assessment
- Network evaluates moves by outputting a score for any (board position, dice values, possible move) combination
- Training set of 3000 instances; there are $\sim 10^{20}$ legal board positions in backgammon
- Each possible move is rate on a scale of -100 to 100
- “higher level” features such as “degree of trapping” turned out to be a necessary part of the input.

Playing Backgammon: Results

- Neurogammon convincingly won the computer backgammon championship at the 1989 International Computer Olympiad

- Can train with supervised learning, but better with reinforcement learning

- RL trained program is at, or near to, playing strength of best humans

Optical Character Recognition

(LeCun et al, 1989)

- Task: read ZIP codes
- Input: segmented, normalized, 16×16 images
- Output: Decision $0, 1, 2, \ldots, 9$
- Architecture: complex, layers of trainable feature detectors (weight sharing)
- Performance: very good.

Context Sensitive Spelling

System called WinSpell, by Golding and Roth (1995).

- Mistakes like "I had a cake for *desert*" cannot be corrected by conventional spell checkers.
- Learn a classifier that for each occurrence decides which word of a "confusion set" it should be.
- Extract Boolean features from a sentence using given patterns. E.g. "in the *", "arid within ± 10 words".
- Features extracted automatically from patterns.

WinSpell Architecture and Performance

- Several layers, but each layer is trained directly and separately.
- Use one layer of Perceptrons to learn many predictors for each word (vary parameters).

  Learning algorithm is *Winnow* (Littlestone 1989), a variant of the PLA which is suitable for handling many irrelevant features (as is the case here).

- Use one layer of "selection nodes" to combine output of predictors for each word.

  Use algorithm *Weighted Majority* (Littlestone and Warmuth, 1994) suitable for combining predictions.

- Use one node to make final decision.
- Performance: gets 96.4% correct on test set (currently best).

Learning from Data

- We have looked at techniques for supervised learning:
  given a set of examples with associated labels
  find a function h that can be used to predict label values for unseen examples.
- Decision Trees, Logical Descriptions, Neural Networks.
- Representation matters.
- Large number and various kinds of applications.
- Much more exists: both theory and practical aspects.
  See modules LFD1, LFD2, CLT, GA, PMR