Revision notes - CS3244

Ma Hongqiang

March 16, 2019

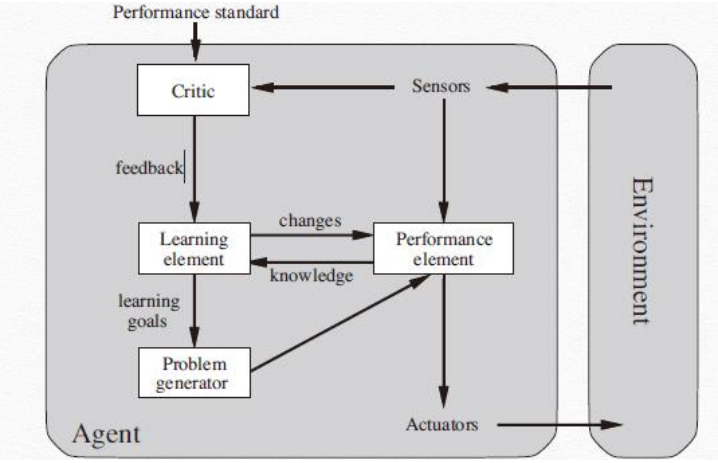
Contents

1 Introduction

Definition 1.1 (Learning)**.**
An agent is said to be **learning** if it improves its performance P on task T based on experience E.
Here T must be fixed, P be measurable and E must exist.

It is useful for the agent to learn since it could be hard to preprogram the agent's strategy, hard to encode all human knowledge, and less to program.

Definition 1.2 (Design of Learning Agent)**.**
Here, the performance element selects the external actions and sends to actuators.



The learning element takes in critic's output and improves agent to perform better, by updating the performance element.
The critic gives feedback on how well the agent is doing.
The problem generator suggests explorative actions that will lead to new, informative, but not necessarily better experience.

The design of the learning agent is affected by

- Which components of the performance element are to be learned
- What representation is used for data and the components
- What feedback is available to learn these components

The types of feedback include:

- **Supervised learning**: correct answer given for each example
- **Unsupervised learning**: correct answers not given
- **Reinforcement learning**: occasional rewards given

2 Concept Learning

Definition 2.1 (Concept, concept learning)**.**
A concept is a boolean-valued function over a set of input instances, each comprising input attributes.
Concept learning is a form of **supervised learning**. It is to infer an unknown **boolean valued function** from *training examples*.

Definition 2.2 (Hypothesis)**.**
Hypothesis h is a conjunction of constraints on input attributes, where each **constraint** can be:

- A specific value, e.g. `water = warm`
- Don't care, e.g. `water = ?`
- No value allowed, e.g. `water = ∅`

Since conjunction is commutative, we can represent a hypothesis in an unordered list like To learn using concept learning, we are given input instances X. Each instance $x \in X$ is represented by the a list of input attributes describing the state, in the form of `key, value`, where `value` is an element from teh set of values corresponding to the `key`.

Sky	AirTemp	Humidity	Wind
Sunny	?	?	Strong

We are also given hypothesis space H. Each hypothesis $h \in H$ with the form $h : X \to \{0, 1\}$ is represented by a conjunction of constraints on input attributes.

Definition 2.3 (Satisfying Hypothesis)**.**
An input instance $x \in X$ **satisfies** all constraints of a hypothesis $h \in H$ iff $h(x) = 1$.

In other words, h classifies x as a positive example.

Definition 2.4 (Aim of Training)**.**
Given unknown **target concept** $c : X \to \{0, 1\}$, and *noise-free* training examples $D = \{\langle x_1, c(x_1)\rangle, \ldots, \langle x_n, c(x_n)\rangle$ determine a hypothesis $h \in H$ that is **consistent** with D.
Here, a hypothesis is **consistent** with training example D *if and only if* $h(x) = c(x)$ for all $\langle x, c(x)\rangle \in D$.

Here, we have this **inductive learning assumption**: any hypothesis found to approximate the target function well over a *sufficiently large set of training examples* will also approximate the target function well over other *unobserved examples*.
One can view concept learning as search for a hypothesis $h \in H$ consistent with D.
Every hypothesis containing 1 or more $\varnothing$ symbols represents an **empty set** of input instances, hence classifying

every instance as a negative example.
Usually the hypothesis space H is quite large or even infinite, so we need to exploit structure for searching efficiently.
Before we can do this, we defines a relation $geq_g : H \times H \to \{0, 1\}$.

Definition 2.5 (More General than or Equal To)**.**
h_j is **more general than or equal to** h_k, denoted by $h_j \geq_g h_k$ if and only if any input instance x that satisfies h_k also satisfies h_j:

$$\forall x \in X, h_k(x) = 1 \Rightarrow h_j(x) = 1$$

$\geq_g$ relation defines a *partial order* over H and not total order.

Definition 2.6 (More General than)**.**
h_j is **more general than** h_k, i.e. $h_j >_g h_k$ if and only if $h_j \geq_g h_k$ and $h_k \not\geq_g h_j$.
h_j is **more specific than** h_k if and only if h_k is more general than h_j.

Theorem 2.1 (Find-S algorithm)**.**

- Initialize h to most specific hypothesis in H
- For each positive training instance x
  - For each attribute constraint a_i in h,
    - if x satisfies constraint a_i in h, do nothing
    - else, replace a_i in h by the next more general constraint that is satisfied by x
- Output hypothesis h

Essentially, we just run the current most specific hypothesis possible against the remaining training data. After one iteration, we can obtain the hypothesis we want for Find-S algorithm.

Theorem 2.2.
h is consistent with D if and only if every positive training instance satisfies h and every negative training instance does not satisfy h.

Theorem 2.3.
Suppose that $c \in H$. Then h_n is consistent with $D = \{\langle x_k, c(x_k)\rangle\}_{k=1,\ldots,n}$.

However, although find-S guarantees to find consistent hypothsis if it exists in hypothesis space, it has the following limitations:

- It cannot tell whether find-WS has learned target concept
- It cannot tell when training examples are inconsistent
- It picks a maximally specific h
- Depending on H, there might be several

Definition 2.7 (Version Space)**.**
The **version space** $VS_{H,D}$ with respect to hypothesis space H and training examples D, is the subset of hypothesis from H that are consistent with D:

$$VS_{H,D} = \{h \in H \mid h \text{ is consistent with } D\}$$

If $c \in H$, then a large enough D can reduce $VS_{H,D}$ to $\{c\}$. However, if D is sufficient, then the cardinality of $VS_{H,D}$ will be more than 1, which measn that $VS_{H,D}$ represents the **uncertainty** of what the target concept is.
$VS_{H,D}$ contains all consistent hypothesis, which includes the maximally specific hypothesis.

Theorem 2.4 (List-Then-Eliminate Algorithm)**.**

- *VersionSpace*$\leftarrow$ a list containing every hypothesis in H
- For each training example $\langle x, c(x)\rangle$
  - Remove from *VersionSpace* any hypothesis h for which $h(x) \neq c(x)$.
- Output the list of hypothesis in *VersionSpace*

This generates the $VS_{H,D}$. However, it is prohibitively exprensive to exhaustively enumerate all hypothesis in finite H.

Definition 2.8 (General Boundary, Specific Boundary)**.**
The **general boundary** G of $VS_{H,D}$ is the set of **maximaly general members** of H consistent with D:

$$\begin{aligned} G = &\{g \in H \mid g \text{ consistent with } D \\ &\wedge (\neg\exists g' \in H, \text{ s.t. } g' >_g g \wedge g' \text{ consistent with } D\} \end{aligned}$$

The **specific boundary** S of $VS_{H,D}$ is the set of maximally specific members of H consistent with D.

$$\begin{aligned} S = &\{s \in H \mid s \text{ consistent with } D \\ &\wedge (\neg\exists s' in H, \text{ s.t. } s >_g s' \wedge s' \text{ consistent with } D\} \end{aligned}$$

From the definition, we can see that every member of version space lies *between* these boundaries:

Theorem 2.5 (Version Space Representation Theorem)**.**

$$VS_{H,D} = \{h \in H \mid \exists s \in S, \exists g \in G g \geq_g h \geq_g s\}$$

Theorem 2.6 (Candidate Elimination Algorithm)**.**

- $G \leftarrow$maximally general hypothesis in H
- $S \leftarrow$maximally specific hypothesis in H
- For each training sample d
  - If d is a positive example
    - Remove from G any hypothesis inconsistent with d
    - For each $s \in S$ not consistent with d
      - Remove s from S
      - Add to S all minimal generalizations h of s such that h is consistent with d, *and* some member of G is more general than h
      - Remove from S any hypothesis that is more general than another hypothesis in S
  - If d is a negative example
    - Remove from S any hypothesis inconsistent with d
    - For each $g \in G$ not consistent with d
      - Remove g from G
      - Add to G all minimal specifications h of g such that h is consistent with d, *and* some member of S is more specific than h
      - Remove from G any hypothesis that is more specific than another hypothesis in G

The candidate elimination algorithm has the following properties:

- Error in training data will remove hypothesis inconsistent with the erroneous example, which include target concept c
  S and G will reduce to $\varnothing$ with sufficiently large data
- Insufficiently expressive hypothesis representation, for example a biased hypothesis space which does not contain c, will cause S and G to be reduced to $\varnothing$ with sufficiently large data.

To make this algorithm efficient, an active learner should query input instance that satisfies *exactly half* of hypotheses in version space, which reduces the version space by half with each training example.
This implies we need at least $\lceil log_2(|VS_{H,D}|)\rceil$ examples to find target concept c.

Theorem 2.7.
An input instance x satisfies every hypothesis in $VS_{H,D}$ if and only if x satisfies every member of S.

The above result is a direct implication from definition of specific boundary S.
We have a counterpart theorem regarding general boundary G:

Theorem 2.8.
An input instance x satisfies none of the hypothesis in $VS_{H,D}$ if and only if x satisfies none of the members of G.

So we can use these theorems to completely classify new unobserved input instance.
Usually when training, an unbiased learner will have a hypothesis space H that can express every teachable concept, which is the power set of X. In such setting, we need training examples for every input instance in X to converge to that target concept. The **limitation** is that it cannot classify new unobserved input isntances.
To overcome such limitations, we introduce **inductive bias**.

Definition 2.9 (Inductive Bias)**.**
Given

- Concept learning algorithm L
- Input isntances X, unknown target concept c
- Noise free training examples $D_c = \{\langle x_k \in X, c(x_k) \in \{0,1\}\}_{k=1,\dots,n}$

We use $L(x, D_c) \rightarrow \{0,1\}$ to denote the classification of input instance x by L after learning from training example D_c.
The **inductive bias** of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c,

$$\forall x \in X, (B \wedge D_c \wedge x) \models (c(x) = L(x, D_c))$$

Theorem 2.9 (Inductive Bias of Candidate Elimination)**.**

The Inductive bias of candidate elimination is the piece of infomration: $B = \{c \in H\}$.
Here, we assume candidate elimination outputs a classification $L(x, D_c)$ of input instance x if this vote among hypothesis in VS_{H,D_c} is unanimously positive or negative, and do not output otherwise.

Here, we have three types of learner:

- **Rote-learner**: Store examples and classify input instance x if and only if it matches that of previously observed example. There is *no* inductive bias, since we do not make any induction.

- **Candidate-Elimination**: Inductive bias: $c \in H$.
- **Find-S**: Inductive bias is $c \in H$ and all instance are negative unless the opposite is entailed by its other knowledge.
  This is because it only has a specific boundary without any general boundary to begin with.

3 Decision Tree Learning

The advantage of decision tree(DT) learning over concept learning is listed in the table below: (See the last page) For decision tree learning, the input instance $X_i = A_1 \times \cdots \times A_n$ is described by input attribute values, which can be `boolean`, discrete or continuous. The classification is still `positive` or `negative`.
We can think of decision tree as a nother possible representation of hypothesis.
Decision tree has the expressive power, which can express any function of input attributes, i.e., $f : A_1 \times \cdots \times A_n \to \{0, 1\}$. Here, we put the realisation of A_p on the edges, which leads to another attribute A_q or the classification.
We would most likely want to find **compact** decision trees.
Note, a `boolean` decision tree can be expressed in disjunctive normal form. For example, $f(A, B) := A\texttt{XOR}B$ can be expressed as $f(A, B) = (\neg A \wedge B) \vee (A \wedge \neq B)$.
The more important idea is that we can view each conjunction as a **path**! The above statement suggests

$$\text{Goal} \Leftrightarrow (\text{Path}_1 \vee \cdots \vee \text{Path}_n)$$

where each path is a **conjunction** of attribute-value tests, of the form $\text{test}_1 \wedge \cdots \wedge \text{test}_m$.
The search space for decision tree grows hyper-exponentially. Specifically, the number of distinct binary decision trees with m `boolean` attributes $= 2^{2^m}$, as we have $|\prod_{i=1}^{m} X_i| = 2^m$, and the distinct decision trees is the power set of this set, as each value can go either 0 or 1.

3.1 Decision Tree Learning Algorithm

The aim of decision tree learning is to find a **small** tree **consistent** with training examples.
We need to *greedily*(which may not be optimal) choose the most "important" attribute as root of subtree, which essentially removes duplication in the rest of the tree.

Theorem 3.1 (Decision Tree Learning Algorithm)**.**
function DECISION-TREE-LEARNING(*examples, attributes, parent_examples*) **returns** tree
  if *examples* is empty **then return** PLURALITY-VALUE(*parent_examples*)
  else if all *examples* have the same classification **then return** the classification
  else if *attributes* is empty **then return** PLURALITY-VALUE(*examples*)
  else
    $A \leftarrow \arg\max_{a \in attributes}$ IMPORTANCE(a, *examples*)
    tree $\leftarrow$ a new decision tree with root test A
    for each value v_k of A **do**
      exs$\leftarrow \{e : e \in examples$ **and** $e.A = v_k\}$
      subtree$\leftarrow$ DECISION-TREE-LEARNING(*exs*, *attribu*
A, *examples*)
      add a branch to `tree` with label $(A = v_k)$ and subtree *subtree*. **return** *tree*

Here, the function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.
The function IMPORTANCE is related to the notion of information gain, which is defined in terms of **entropy**, which measures **uncertainty of classification**.

Definition 3.1 (Entropy)**.**
Entropy measures uncertainty of random variable $C \in \{c_1, \ldots, c_k\}$:

$$H(C) = -\sum_{i=1}^{k} P(c_i) \log_2 P(c_i)$$

We can define $B(q)$ as entropy of Boolean variable with probability q to be `true`, i.e.,

$$B(q) = -q \log_2 q - (1 - q) \log_2 (1 - q)$$

For a training set containing p positive examples and n negative examples, entropy of target concept C on this set is

$$H(C) = B(\frac{p}{p+n}) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}$$

Here, some important special case of entropy are:

- If $p = n(\neq 0)$, then $H(C) = 1$ attains its **maximum**.
- If $p = 0$ or $n = 0$, then $H(C) = 0$, which suggests no uncertainty.
- Any other non-zero combinations will attain an entropy in $(0, 1)$.

Specifically, we note the 2-class entropy $B(\frac{p}{p+n})$ to increase monotonically between $(0, \frac{1}{2})$ and monotonically decreasing between $(\frac{1}{2}, 1)$.
Suppose a chosen attribute A divides the training set E into subset $E_1, \ldots, E_d$ corresponding to the d distinct values of A. Each subset E_i has p_i positive and n_i negative

examples. Then the expected entropy remaining after testing attribute A is

$$H(C \mid A) := \sum_{i=1}^{d} \frac{p_i + n_i}{p+n} B(\frac{p_i}{p_i + n_i})$$

Definition 3.2 (Information Gain)**.**
Information Gain of target concept C from the attribute test on A is the expected reduction in entropy:

$$\text{Gain}(C, A) = B(\frac{p}{p+n}) - H(C \mid A)$$

where the first term on the RHS is the entropy $H(C)$.

In the decision tree learning, we choose the attribute A with the largest Gain. This gives the implementation of IMPORTANCE.
In other words, DECISION-TREE-LEARNING uses information gain *heuristic* to search through the space of decision trees, from simplest to increasingly complex.

Theorem 3.2 (Inductive Bias of DT Learning)**.**
There are 2 inductive bias:

1. **Shorter trees** are preferred.
2. Trees that place **high information gain attributes close to the root** are preferred.

If we only include (a) in the bias, it is the **exact** inductive bias of BFS for shortest consistent DT, which is prohibitively expensive.
Do note, bias is a *preference* for some hypothesis over the others. Bias does not restrict hypothesis space.

3.2 Overfitting

Definition 3.3 (Overfit)**.**
Hypothesis $h \in H$ **overfits** the set D of training examples if and only if

$$\exists h' \in H \backslash \{h\} (\text{error}_D(h) < \text{error}_D(h') \wedge \text{error}_{D_X}(h) > \text{error}_{D_X}(h'))$$

where $\text{error}_D(h)$ denotes error of h over D; $\text{error}_{D_X}(h)$ denotes erros of h over D_X, exmamples corresponding to instance space X.

Here, it is clear that training set D is a subset of instance space X.
Overfitting is more likely as hypothesis and number of input attribute grows, and less likely if we increase number of training examples.
We can avoid overfitting by

- Stop growing DT when expanding a node is not *statistically* significant.
- Allow DT to grow and overfit the data, and **then post-prune** it.

In order to have a metric measuring the quality of DT, we can measure against

- Training data
- or a separate **validation** dataset
- Minimum description length, which minimizes the size of tree and size of misclassifications of the tree

Theorem 3.3 (Reduced-error Pruning)**.**
We here explore ideas of partition data into *training* and *validation* sets. The algorithm is below:
Do, until further pruning is harmful:

1. Evaluate impact on *validation* set of pruning each possible node (i.e., removing subtree rooted at it)
2. Greedily remove the one that **most** improves validation set accuracy

Then, produce the smallest version of the most accurate subtree.

Theorem 3.4 (Rule Post-Pruning)**.**
Convert learned DT to an equivalent set of rules by creating one rule for each path from root to a leaf:

```
IF   Rules(conjunction)
THEN Classification
```

Then prune each rule by removing any precondition that improves its estimated accuracy.
Sort pruned rules by estimated accuracy into desired sequence for use, when classifying unobserved input instances.

3.3 Dealing with Attributes

For continuous-valued attributes, we consider only the discrete set of intervals derived from the continuous values.
There is one problem with the guiding function for IMPORTANCE, the Gain function. In training, Gain will select attribute with **many values**.
To resolve the problem, we define another function called `GainRatio`:

Definition 3.4 (Gain Ratio)**.**

$$\texttt{GainRatio}(C, A) = \frac{\text{Gain}(C, A)}{\text{SplitInformation}(C, A)}$$

where $\text{SplitInformation}(C, A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} \log_2 \frac{|E_i|}{|E|}$.

Another problem arises we want to learn consistent DT with low expected cost. In such case we can replace Gain by things like

$$\frac{\text{Gain}^2(C,A)}{\text{Cost}(A)}, \frac{2^{\text{Gain}(C,A)}}{(\text{Cost}(A)+1)^\omega}$$

where $\omega \in [0,1]$ determines importance of cost. If there is some examples with missing values in A, we can still use training examples, and sort through decision tree. The below are three different permissible approaches:

- If node n tests A, then assign most common value of A among other examples sorted to node n
- Assign most common value of A among other examples sorted to node n with same value of output/target concept
- Assign probability p_i to each possible value of A, and assign fraction p_i of example to each descendant in DT

Then we can classify new unobserved input instances with missing attribute values in same manner.

4 Neural Networks

The neural net has the characteristics listed below:

- Neuron-like threshold switching units
- Weighted interconnections among units
- Highly parallel, distributed process
- Tuning weights automatically

Below table is a comparison between decision tree(DT) learning and neural nets. (See last page) Neural net is based on **perceptron unit**, where it has input $x_1, \ldots, x_n$ and output a binary value 1 or -1. Specifically, the perceptron can be viewed as the function

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

where $\mathbf{w} = (w_0, \ldots, w_n)^T$ and $\mathbf{x} = (1, x_1, \ldots, x_n)^T$.
Essentially, the decision surface of a single perceptron is a hyperplane. However, not all functions are linearly separable and therefore not all functions can be represented by 1 perceptron.
In the case where the function is linearly separable, we can apply the **perceptron training rule**.

Definition 4.1 (Perceptron Training Rule)**.**
Initialize $\mathbf{w}$ randomly. Iterate through all training examples till $\mathbf{w}$ is consistent, and do

$$w_1 \leftarrow w_1 + \delta w_i, \quad \text{where } \delta w_i = \eta(t-o)x_i$$

for $i = 0, 1 \ldots, n$, where

- $t = c(\mathbf{x})$ is the target output for training example $\langle \mathbf{x}, c(\mathbf{x})\rangle$
- $o = o(\mathbf{x})$ is the perceptron's current output
- η is small positive constant called learning rate.

Perceptron training rule is guaranteed to converge if training examples are linearly separable and η is sufficiently small.
However, in the case where the training examples are not linearly separable, due to a linearly non-separable function, or due to error, we can use **gradient descent**, whose aim is to minimize the **squared error/loss** $L_D(\mathbf{w})$:

$$L_D(\mathbf{w}) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

where D is the set of training examples, t_d, o_d are target output and output of linear unit $o(\mathbf{x}) := \mathbf{w}\cdot\mathbf{x}$, for training sample d.

Definition 4.2 (Gradient Descent)**.**
GRADIENT-DESCENT(D, η)

- Initialize each w_i to some small random value
- Until termination condition is met, do
  - Initialize each $\delta w_i \leftarrow 0$.
  - For each $d \in D$, do
    - Input instance $\mathbf{x}_d$ to linera unit, and compute output $o := o(\mathbf{x}_d)$
    - For each linear unit weight w_i, do

$$\delta w_i \leftarrow \delta w_i + \eta(t-o)x_{id}$$

  - After we traverse all $d \in D$, do

$$w_i \leftarrow w_i + \delta w_i$$

Here, the gradient $\delta w_i := -\delta\frac{\partial L_D}{\partial w_i} = \eta\sum_{d\in D}(t_d - o_d)x_{id}$.

It is guaranteed that gradient descent will converge to the hypothesis vector $\mathbf{w}$ with minimum squared error if learning rate η is sufficiently small.
However, batch gradient descent will be problematic when the training set is large, as the update is done only every data in the dataset is traversed. Therefore, we introduce **stochastic gradient descent**.

Definition 4.3 (Stochastic Gradient Descent)**.**
In stochastic gradient descnet, for each training example $d \in D$, do

- Compute Gradient $\nabla L_d(\mathbf{w})$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L_d(\mathbf{w})$ where $L_d(w) := \frac{1}{2}(t_d - o_d)^2$ is the loss of a single data.

It can be shown that SGD can approximate batch GD arbitrarily closely if η is sufficiently small.

Here, the perceptron unit has the characteristic of output binary answers based on a linear threshold rule, which is not differentiable and difficult to apply gradient descent on. Also, linear units will still produce linear functions even if they stack together, so we cannot use it to represent *highly nonlinear functions*. As a result, we choose **sigmoid** unit, which is like perceptron but will a smoothed, differentiable threshold function.

Definition 4.4 (Sigmoid Unit)**.**
Like a linera unit, the sigmoid unit compute its output o as

$$o = \sigma(\mathbf{w} \cdot \mathbf{x})$$

where $\sigma(y) = \frac{1}{1+e^{-y}}$, is called the sigmoid function, or the logistic function. It has the property of $\frac{\mathrm{d}\,\sigma(y)}{\mathrm{d}\,y} = \sigma(y)(1 - \sigma(y))$.
By this property, we have

$$\frac{\partial L_D}{\partial w_i} = -\sum_{d \in D}(t_d - o_d)o_d(1 - o_d)x_{id}$$

Gradient Descent can be applied to train either 1 sigmoid unit, or multilayer network of sigmoid units via **backpropagation**.
The Backpropagation algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It aims to minimize the squared error between the network output value and the target values for these outputs.
Since the network can have *multiple output*, we redefine squared loss $L_D(\mathbf{w})$ as

$$L_D(\mathbf{w}) = \frac{1}{2}\sum_{d \in D}\sum_{k \in K}(t_{kd} - o_{kd})^2$$

where K is the set of output units in the network.
Backpropagation here assumes 2 layers of sigmoid units and is based on SGD, which simplifies alway the set D in the loss function: $L_d(\mathbf{w}) := \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$.

Definition 4.5 (Backpropagation Algorithm)**.**
Initialize $\mathbf{w}$ randomly to some small random numbers. Until satisfied, do

- For each training example $\langle \mathbf{x}, (t_k)_{k \in K}^T \rangle$ do
  1. Inpute instance $\mathbf{x}$ to the network and compute output of every sigmoid unit in the hidden and output layer
  2. For each output unit k, compute error $\Delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
  3. For each hidden unit h, compute error $\Delta_h \leftarrow o_h(1-o_h)\sum_{k \in K} w_{hk}\Delta_k$ (Backpropagation of loss)
  4. Update each weight $w_{hk} \leftarrow w_{hk} + \delta w_{hk}$ where $\delta w_{hk} = \eta \delta_k o_h$
  5. Update each weight $w_{ih} \leftarrow w_{ih} + \delta w_{ih}$ where $\delta w_{ih} = \eta \delta_h x_i$.

Note, L_D will have multiple *local minima*. GD is guaranteed to converge to some local minima but not necessarily global minima. However, we can use *multiple* random initialization of $\mathbf{w}$ to explore different local minima.

One variation of backpropagation adds momentum during update:

$$\delta w_{hk} \leftarrow \eta \Delta_k o_h + \alpha \delta w_{hk}, \quad \delta w_{ih} \leftarrow \eta \Delta_h x_i + \alpha \delta w_{ih}$$

where $\alpha \in [0, 1)$ is the **weight momentum**.
This algorithm can be further generalized to feedforward network of arbitrary depth by

- In step 3, let K denote all units in the next deeper layer whose inputs include output of h
- Let x_i denote output of unit i in the previous layer that is input to h

The advantage of backpropagation is that

- **Expressive hypothesis space**: Every boolean functi onc an be represetned by a network with 1 hidden layer, but may require exponential hidden units in number of inputs; every bounded continuous function can be approximated with arbitrarily small error by a network with 1 hidden layer; any function can be approximated to arbitrarily accuracy by a network with 2 hidden layers
- **Approximate inductive bias**: smmoth interpolation between data points

We can have variation on loss functions too. Some examples include

- Penalize large weights:

$$L_D(\mathbf{w}) = \frac{1}{2}\sum_{d \in D}\sum_{k \in K}(t_{kd} - o_{kd})^2 + \gamma \sum_{j,l} w_{jl}^2$$

- Train on target values as well as slopes

$$L_D(\mathbf{w}) = \frac{1}{2}\left[\sum_{d \in D}\sum_{k \in K}(t_{kd} - o_{kd})^2 + \mu \sum_{i=1}^{n}(\frac{\partial t_{kd}}{\partial x_{id}} - \frac{\partial o_{kd}}{\partial x_{id}}\right.$$

- Tie together weights

	Concept Learning	DT Learning
Target function/concept	Binary outputs	Discrete Outputs
Training Data	Noise-free	Robust to noise
Hypothesis space	Restricted(hard bias)	Complete, expressive
Search strategy	Complete: version space Refine search per example	Incomplete: prefer shorter tree(soft bias) Refine search using all examples No backtracking
Exploit structure	General to specific ordering	Simple to complex ordering

Table 1: Concept vs DT

	DT Learning	Neural Nets
Target function/output	Discrete Outputs	Discrete/Real vector
Input instance	Discrete	Discrete/real, high-dimensional
Training Data	Robust to noise	Robust to noise
Hypothesis space	Complete, expressive	Restricted: #hidden units(hard bias), expressive
Search strategy	Incomplete: prefer shorter tree (soft bias) Refine searfh using all examples No backtracking	Incomplete: prefer smaller weights (soft bias) Gradient ascent batch mode: all examples; stochastic: mini-batches
Training time	Short	Long
Prediction Time	Fast	Fast
Interpretability	White-box	Black-box

Table 2: DT vs Neural