Artificial Intelligence

Learning and Neural Networks

Readings: Chapter 18 & 20.5 of Russell & Norvig
Learning Agents

A distinct feature of intelligent agents in nature is their ability to learn from experience.

Using his experience and his internal knowledge, a learning agent is able to produce new knowledge.

That is, given his internal knowledge and a percept sequence, the agent is able to learn facts that are consistent with both the percepts and the previous knowledge,

*do not just follow* from the percepts and the previous knowledge.
Learning element

Design of learning element is dictated by:
- what type of performance element is used
- which functional component is to be learned
- how that functional component is represented
- what kind of feedback is available

Example scenarios:

<table>
<thead>
<tr>
<th>Performance element</th>
<th>Component</th>
<th>Representation</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha–beta search</td>
<td>Eval. fn.</td>
<td>Weighted linear function</td>
<td>Win/loss</td>
</tr>
<tr>
<td>Logical agent</td>
<td>Transition model</td>
<td>Successor–state axioms</td>
<td>Outcome</td>
</tr>
<tr>
<td>Utility–based agent</td>
<td>Transition model</td>
<td>Dynamic Bayes net</td>
<td>Outcome</td>
</tr>
<tr>
<td>Simple reflex agent</td>
<td>Percept–action fn</td>
<td>Neural net</td>
<td>Correct action</td>
</tr>
</tbody>
</table>

Supervised learning: correct answers for each instance
Inductive Learning

- Simplest form: learn a function from examples
- $f$ is the target function

An example is a pair $\langle x, f(x) \rangle$, e.g., $\langle X, X \rangle$, $+1$.

Problem: find hypothesis $h$ such that $h \approx f$ given a training set of examples.

This is a highly simplified model of real learning:
- Ignores prior knowledge
- Assumes a deterministic, observable “environment”
- Assumes examples are given
- Assumes that the agent wants to learn $f$
Inductive learning method

- Construct/adjust $h$ to agree with $f$ on training set
  ($h$ is consistent if it agrees with $f$ on all examples)

- E.g., curve fitting:
Inductive learning method

- Construct/adjust $h$ to agree with $f$ on training set ($h$ is consistent if it agrees with $f$ on all examples)
- E.g., curve fitting:

![Diagram showing curve fitting]
Inductive learning method

- Construct/adjust $h$ to agree with $f$ on training set ($h$ is consistent if it agrees with $f$ on all examples)
- E.g., curve fitting:

![Graph showing curve fitting]
Inductive learning method

- Construct/adjust $h$ to agree with $f$ on training set
  ($h$ is consistent if it agrees with $f$ on all examples)
- E.g., curve fitting:

![Graph showing curve fitting]
Inductive learning method

- Construct/adjust \( h \) to agree with \( f \) on training set (\( h \) is consistent if it agrees with \( f \) on all examples)
- E.g., curve fitting:
Inductive learning method

- Construct/adjust $h$ to agree with $f$ on training set
  ($h$ is consistent if it agrees with $f$ on all examples)
- E.g., curve fitting:

Ockham’s razor: maximize a combination of consistency and simplicity
Performance Measurement

How do we know that $h \approx f$? (Hume's Problem of Induction)

1. Use theorems of computational/statistical learning theory
2. Try $h$ on a new test set of examples (use same distribution over example space as training set)

Learning curve = % correct on test set as a function of training set size
Performance Measurement

Learning curve depends on

- realizable (can express target function) vs. non-realizable
  non-realizability can be due to missing attributes or restricted hypothesis class (e.g., thresholded linear function)

- redundant expressiveness (e.g., loads of irrelevant attributes)
Why Learning Works?

- There is a theoretic foundation: Computational Learning Theory
- The underlying principle: Any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong – it must be Probably Approximately Correct (PAC).
- The Stationarity Assumption: The training and test sets are drawn randomly from the same population of examples using the same probability distribution.
- Suppose \( H \) is the set of all hypotheses, to make sure that the probability error of the test set is less than \( \epsilon \) and the probability error that a good hypothesis escapes the learning algorithm is less than \( \delta \), we need to have a training set of size \( m \), where

\[
m \geq \frac{1}{\epsilon} \left( ln \frac{1}{\delta} + ln |H| \right)
\]
Brains as Computational Devices

**Motivation:** Algorithms developed over centuries do not fit the complexity of real-world problems. The human brain is the most sophisticated computer suitable for solving extremely complex problems.

- **Reasonable Size:** $10^{11}$ neurons (neural cells) and only a small portion of these cells are used.

- **Simple Building Blocks:** No cell contains too much information.

- **Massively parallel:** Each region of the brain controls specialized tasks.

- **Fault-tolerant:** Information is saved mainly in the connections among neurons.

- **Reliable**

- **Graceful degradation**
## Comparing Brains with Computers

<table>
<thead>
<tr>
<th></th>
<th>Computer</th>
<th>Human Brain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational units</td>
<td>1 CPU, $10^5$ gates</td>
<td>$10^{11}$ neurons</td>
</tr>
<tr>
<td>Storage units</td>
<td>$10^9$ bits RAM, $10^{10}$ bits disk</td>
<td>$10^{11}$ neurons, $10^{14}$ synapses</td>
</tr>
<tr>
<td>Cycle time</td>
<td>$10^{-8}$ sec</td>
<td>$10^{-3}$ sec</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>$10^9$ bits/sec</td>
<td>$10^{14}$ bits/sec</td>
</tr>
<tr>
<td>Neuron updates/sec</td>
<td>$10^5$</td>
<td>$10^{14}$</td>
</tr>
</tbody>
</table>

Even if a computer is one million times faster than a brain in raw speed, a brain ends up being one billion times faster than a computer at what it does.

**Example: Recognizing a face**

- **Brain:** $< 1s$ (a few hundred computer cycles)
- **Computer:** billions of cycles
A neuron does nothing until the collective influence of all its inputs reaches a threshold level.

At that point, the neuron produces a full-strength output in the form of a narrow pulse that proceeds from the cell body, down the axon, and into the axon’s branches.

“It fi res!”: Since it fi res or does nothing it is considered an all or nothing device.

Increases or decreases the strength of connection and causes excitation or inhibition of a subsequent neuron.
Artificial neurons are viewed as a node connected to other nodes via links that correspond to neural connections.

Each link is associated with a weight.

The weight determines the nature (+/-) and strength of the node’s influence on another.

If the influence of all the links is strong enough the node is activate (similar to the firing of a neuron).
A Neural Network Unit

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

\[ o(\vec{x}) = \begin{cases} 
1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\
-1 & \text{otherwise.} 
\end{cases} \]
Artificial Neural Network

- A neural network is a graph of nodes (or units), connected by links.
- Each link has an associated weight, a real number.
- Typically, each node $i$ has several incoming links and several outgoing links. Each incoming link provides a real number as input to the node and the node sends one real number through every outgoing link.
- The output of a node is a function of the weighted sum of the node’s inputs.
The Input Function

Each incoming link of a unit $i$ feeds an input value, or activation value, $a_j$ coming from another unit.

The input function $in_i$ of a unit is simply the weighted sum of the unit’s input:

$$in_i(a_1, \ldots, a_{n_i}) = \sum_{j=1}^{n_i} W_{j,i} a_j$$

The unit applies the activation function $g_i$ to the result of $in_i$ to produce an output:

$$out_i = g_i(in_i) = g_i\left(\sum_{j=1}^{n_i} W_{j,i} a_j\right)$$
Typical Activation Functions

(a) Step function

\[ \text{step}_t(x) = \begin{cases} 
1, & \text{if } x \geq t \\
0, & \text{if } x < t
\end{cases} \]

(b) Sign function

\[ \text{sign}(x) = \begin{cases} 
+1, & \text{if } x \geq 0 \\
-1, & \text{if } x < 0
\end{cases} \]

(c) Sigmoid function

\[ \text{sig}(x) = \frac{1}{1+e^{-x}} \]
Typical Activation Functions 2

- Hard limiter (binary step)
  
  \[
  f_\theta(x) = \begin{cases} 
  1, & \text{if } x > \theta \\
  0, & \text{if } -\theta \leq x \leq \theta \\
  -1, & \text{if } x < -\theta
  \end{cases}
  \]

- Binary sigmoid (exponential sigmoid)
  
  \[
  \text{sig}(x) = \frac{1}{1 + e^{-\sigma x}}
  \]
  
  where \(\sigma\) controls the saturation of the curve. When \(\sigma \leftarrow \infty\), hard limiter is achieved.

- Bipolar sigmoid (atan)
  
  \[
  f(x) = \tan^{-1}(\sigma x)
  \]
Units as Logic Gates

Since units can implement the AND, OR, NOT boolean operators, neural nets are Turing-complete: they can implement any computable function.
Structures of Neural Networks

- Directed:
  - Acyclic:
    - Feed-Forward:
      - Multi Layers: Nodes are grouped into layers and all links are from one layer to the next layer.
      - Single Layer: Each node sends its output out of the network.
    - Tree: ...
    - Arbitrary Feed: ...
  - Cyclic: ...
- Undirected: ...
Multilayer, Feed-forward Networks

A kind of neural network in which

- links are directional and form no cycles (the net is a \textit{directed acyclic graph});

- the root nodes of the graph are \textit{input units}, their activation value is determined by the environment;

- the leaf nodes are \textit{output units};

- the remaining nodes are \textit{hidden units};

- units can be divided into \textit{layers}: a unit in a layer is connected only to units in the next layer.
A Two-layer, Feed-forward Network

Output units $O_i$

Hidden units $a_j$

Input units $I_k$

Notes:
- The roots of the graph are at the bottom and the (only) leaf at the top.
- The layer of input units is generally not counted (which is why this is a two-layer net).
Another Two-layer, Feed-forward Network

\[ a_5 = g_5(W_{3,5}a_3 + W_{4,5}a_4) \]
\[ = g_5(W_{3,5}g_3(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g_4(W_{1,4}a_1 + W_{2,4}a_2)) \]

where \( a_i \) is the output and \( g_i \) is the activation function of node \( i \).
Multilayer, Feed-forward Networks

Are a powerful computational device:

- with just one hidden layer, they can approximate any continuous function;
- with just two hidden layers, they can approximate any computable function.

However, the number of needed units per layer may grow exponentially with the number of the input units.
Perceptrons

Single-layer, feed-forward networks whose units use a step function as activation function.
Perceptrons caused a great stir when they were invented because it was shown that

If a function is representable by a perceptron, then it is learnable with 100% accuracy, given enough training examples.

The problem is that perceptrons can only represent linearly-separable functions.
Linearly Separable Functions

On a 2-dimensional space:

(a) \( I_1 \) and \( I_2 \)

(b) \( I_1 \) or \( I_2 \)

(c) \( I_1 \) xor \( I_2 \)

A black dot corresponds to an output value of 1. An empty dot corresponds to an output value of 0.
A Linearly Separable Function

On a 3-dimensional space:
The *minority* function: Return 1 if the input vector contains less ones than zeros. Return 0 otherwise.

(a) Separating plane

(b) Weights and threshold

\[
W = -1 \\
W = -1 \\
W = -1 \\
t = -1.5
\]
Some Applications of Neural Networks

- **Signal and Image Processing**
  - Signal prediction (e.g., weather prediction)
  - Adaptive noise cancellation
  - Satellite image analysis
  - Multimedia processing

- **Bioinformatics**
  - Functional classification of protein and genes
  - Clustering of genes based on DNA microarray data
Some Applications of Neural Networks

- Astronomy
  - Classification of objects (stars and galaxies)
  - Compression of astronomical data

- Finance and Marketing
  - Stock market prediction
  - Fraud detection
  - Loan approval
  - Product bundling
  - Strategic planning
Computing with NNs

- Different functions are implemented by different network topologies and unit weights.
- The lure of NNs is that a network need not be explicitly programmed to compute a certain function $f$.
- Given enough nodes and links, a NN can learn the function by itself.
- It does so by looking at a training set of input/output pairs for $f$ and modifying its topology and weights so that its own input/output behavior agrees with the training pairs.
- In other words, NNs learn by induction, too.
Neural networks are trained using data referred to as a training set.

The process is one of computing outputs, compare outputs with desired answers, adjust weights and repeat.

The information of a Neural Network is in its structure, activation functions, weights, and

Learning to use different structures and activation functions is very difficult.

These weights are used to express the relative strength of an input value or from a connecting unit (i.e., in another layer). It is by adjusting these weights that a neural network learns.
Process for Developing Neural Networks

1. **Collect data** Ensure that application is amenable to a NN approach and pick data randomly.

2. **Separate Data into Training Set and Test Set**

3. **Define a Network Structure** Are perceptrons sufficient?

4. **Select a Learning Algorithm** Decided by available tools

5. **Set Parameter Values** They will affect the length of the training period.

6. **Training** Determine and revise weights

7. **Test** If not acceptable, go back to steps 1, 2, ..., or 5.

8. **Delivery of the product**
The Perceptron Learning Method

Weight updating in perceptrons is very simple because each output node is independent of the other output nodes.

With no loss of generality then, we can consider a perceptron with a single output node.
Normalizing Unit Thresholds.

Notice that, if \( t \) is the threshold value of the output unit, then

\[
\text{step}_t\left( \sum_{j=1}^{n} W_j I_j \right) = \text{step}_0\left( \sum_{j=0}^{n} W_j I_j \right)
\]

where \( W_0 = t \) and \( I_0 = -1 \).

Therefore, we can always assume that the unit’s threshold is 0 if we include the actual threshold as the weight of an extra link with a fixed input value.

This allows thresholds to be learned like any other weight.

Then, we can even allow output values in \([0, 1]\) by replacing \( \text{step}_0 \) by the sigmoid function.
The Perceptron Learning Method

If \( O \) is the value returned by the output unit for a given example and \( T \) is the expected output, then the unit’s error is

\[
Err = T - O
\]

If the error \( Err \) is positive we need to increase \( O \); otherwise, we need to decrease \( O \).
The Perceptron Learning Method

Since \( O = g(\sum_{j=0}^{n} W_j I_j) \), we can change \( O \) by changing each \( W_j \).

Assuming \( g \) is monotonic, to increase \( O \) we should increase \( W_j \) if \( I_j \) is positive, decrease \( W_j \) if \( I_j \) is negative.

Similarly, to decrease \( O \) we should decrease \( W_j \) if \( I_j \) is positive, increase \( W_j \) if \( I_j \) is negative.

This is done by updating each \( W_j \) as follows

\[
W_j \leftarrow W_j + \text{pha} \times I_j \times (T - O)
\]

where \( \text{pha} \) is a positive constant, the learning rate.
Learning a 5-place Minority Function

At first, collect the data (see below), then choose a structure (a perceptron with five inputs and one output) and the activation function (i.e., \( step_3 \)). Finally, set up parameters (i.e., \( W_i = 0 \)) and start to learn:

Assuming \( \alpha = 1 \), we have

\[
\text{Sum} = \sum_{i=1}^{5} W_i I_i, \quad \text{Out} = step_3(\text{Sum}),
\]

\[
Err = T - \text{Out}, \quad \text{and} \quad W'_j = W_j + I_j \ast Err.
\]
Summary

- Learning needed for unknown environments, lazy designers
- Learning agent = performance element + learning element
- Learning method depends on type of performance element, available feedback, type of component to be improved, and its representation
- For supervised learning, the aim is to find a simple hypothesis approximately consistent with training examples
- Learning performance = prediction accuracy measured on test set
- Most brains have lots of neurons; each neuron $\approx$ linear–threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, credit cards, etc.