Overview

- Learning in general
- Inductive learning (chapter 18)
- Statistical learning: neural networks (chapter 20; old 19)

Learning

- Adapt through interaction with the world: rote memory to developing a complex strategy
- Types of learning:
  1. Supervised learning (dense feedback)
  2. Unsupervised learning (no feedback)
  3. Reinforcement learning (sparse feedback, environment altering), etc.
- Advantages (two, among many):
  1. Fault tolerance
  2. No need for a complete specification to begin with
- Becoming a central focus of AI.

Inductive Learning

- Given example pairs \((x, f(x))\), return a function \(h\) that approximates the function \(f\):
  - pure inductive inference, or induction.
- The function \(h\) is called a hypothesis.

Inductive Learning and Bias

Given (a) as the training data, we can come up with several different hypotheses: (b) to (d)

- selection of one hypothesis over another is called a bias.
  - exact match to training data
  - prefer imprecise but smooth approximation
  - etc.
Decision Trees

- learn to approximate **discrete-valued** target functions.
- step-by-step decision making (disjunction of conjunctions)
- applications: medical diagnosis, assess credit risk of loan applicants, etc.

**Decision Trees: What They Represent**

- In other words, for each instance (or example), there are attributes (Patrons, Hungry, etc.) and each instance have a full attribute value assignment.
- For a given instance, it is classified into different discrete classes by the decision tree.
- For training, many *(instance, class)* pairs are used.

**Decision Trees: What They Represent (cont’d)**

<table>
<thead>
<tr>
<th>Example</th>
<th>Alt</th>
<th>Bar</th>
<th>Fri</th>
<th>Hun</th>
<th>Pat</th>
<th>Price</th>
<th>Rain</th>
<th>Res</th>
<th>Type</th>
<th>Est</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>SSS</td>
<td>No</td>
<td>Yes</td>
<td>French</td>
<td>6-10</td>
<td>Yes</td>
</tr>
<tr>
<td>X₂</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Full</td>
<td>S</td>
<td>No</td>
<td>No</td>
<td>Thai</td>
<td>30-60</td>
<td>No</td>
</tr>
<tr>
<td>X₃</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Some</td>
<td>S</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>10-30</td>
<td>Yes</td>
</tr>
<tr>
<td>X₄</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>S</td>
<td>No</td>
<td>No</td>
<td>Thai</td>
<td>30-60</td>
<td>Yes</td>
</tr>
<tr>
<td>X₅</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Full</td>
<td>SSS</td>
<td>No</td>
<td>Yes</td>
<td>French</td>
<td>&gt;60</td>
<td>No</td>
</tr>
<tr>
<td>X₆</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>SSS</td>
<td>Yes</td>
<td>Yes</td>
<td>Italian</td>
<td>0-10</td>
<td>Yes</td>
</tr>
<tr>
<td>X₇</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>S</td>
<td>Yes</td>
<td>No</td>
<td>Thai</td>
<td>0-10</td>
<td>No</td>
</tr>
<tr>
<td>X₈</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>SSS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>&gt;60</td>
<td>No</td>
</tr>
<tr>
<td>X₉</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Full</td>
<td>S</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>&gt;60</td>
<td>No</td>
</tr>
<tr>
<td>X₁₀</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>SSS</td>
<td>No</td>
<td>Yes</td>
<td>Italian</td>
<td>10-30</td>
<td>No</td>
</tr>
<tr>
<td>X₁₁</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>S</td>
<td>No</td>
<td>No</td>
<td>Thai</td>
<td>0-10</td>
<td>No</td>
</tr>
<tr>
<td>X₁₂</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>S</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>30-60</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Constructing Decision Trees from Examples**

- Given a set of examples (training set), both positive and negative, the task is to construct a decision tree that describes a concise decision path.
- Using the resulting decision tree, we want to **classify** new instances of examples (either as yes or no).
Constructing Decision Trees: Trivial Solution

- A trivial solution is to explicitly construct paths for each given example.
- The problem with this approach is that it is not able to deal with situations where, some attribute values are missing or new kinds of situations arise.
- Consider that some attributes may not count much toward the final classification.

Finding a Concise Decision Tree

- Memorizing all cases may not be the best way.
- We want to extract a decision pattern that can describe a large number of cases in a concise way.
- Such an inductive bias is called Ockham's razor: The most likely hypothesis is the simplest one that is consistent with all observations.
- In terms of a decision tree, we want to make as few tests before reaching a decision, i.e. the depth of the tree should be shallow.

Finding a Concise Decision Tree (cont’d)

- Basic idea: pick up attributes that can clearly separate positive and negative cases.
- These attributes are more important than others: the final classification heavily depend on the value of these attributes.
**Decision Tree Learning Algorithm**

```plaintext
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 vi of best do
        examplesi ← {elements of examples with best = vi}
        subtree ← DECISION-TREE-LEARNING(examplesi, attributes − best, MAJORITY-VALUE(examplesi))
        add a branch to tree with label vi and subtree subtree
    end
    return tree
```

**Resulting Decision Tree**

- Some attributes are not tested at all.
- Odd paths can be generated (Thai food branch).
- Sometimes the tree can be incorrect for new examples (exceptional cases).

**Accuracy of Decision Trees**

![Accuracy Graph](image)

- Divide examples into training and test sets.
- Train using the training set.
- Measure accuracy of resulting decision tree on the test set.

**Choosing the Best Attribute to Test First**

Use Shannon’s information theory to choose the attribute that gives the maximum information gain.

- Pick an attribute such that the information gain (or entropy reduction) is maximized.
- Entropy measures the average surprisal of events. Less probable events are more surprising.
**Entropy and Information Gain**

\[
Entropy(E) = \sum_{i \in C} -P_i \log_2(P_i)
\]

\[
Gain(E, A) = Entropy(E) - \sum_{v \in Values(A)} \frac{|E_v|}{|E|} Entropy(E_v)
\]

- **E**: set of examples
- **A**: a single attribute
- **E_v**: set of examples where attribute \(A = v\).
- **|S|**: cardinality of set S.

**Issues in Decision Tree Learning**

- Noise and overfitting
- Missing attribute values from examples
- Multi-valued attributes with large number of possible values
- Continuous-valued attributes.

**Key Points**

Decision tree learning:

- What is the embodied principle (or bias)?
- How to choose the best attribute? Given a set of examples, choose the best attribute to test first.
- What are the issues? noise, overfitting, etc.

**Neural Networks**

Neural networks is one particular form of learning from data.

- simple processing elements: named units, or neurons
- connective structure and associated connection weights
- learning: adaptation of connection weights

Neural networks mimic the human (or animal) nervous system.
Many Faces of Neural Networks

- Abstract mathematical/statistical model
- Optimization algorithm
- Pattern recognition algorithm
- Tools for understanding the function of the brain
- Robust engineering application

The Central Nervous System

- Cortex: thin outer sheet where most of the neurons are.
- Sub-cortical nuclei: thalamus, hippocampus, basal ganglia, etc.
- Midbrain, pons, and medulla, connects to the spinal cord.
- Cerebellum (hind brain, or small brain)

Function of the Nervous System

Function of the nervous system:

- Perception
- Cognition
- Motor control
- Regulation of essential bodily functions

The Central Nervous System: Facts

Facts: human neocortex

- Thickness: 1.6mm
- Area: 36cm × 36cm (about 1.4 ft²)
- Neurons: 10 billion ($10^{10}$)
- Connections: 60 trillion ($6 \times 10^{13}$) to 100 trillion
- Connections per neuron: $10^4$
- Energy usage per operation: $10^{-16}$ J (compare to $10^{-6}$ J in modern computers)

How the Brain Differs from Computers

- Densely connected.
- Massively parallel.
- Highly nonlinear.
- Asynchronous: no central clock.
- Fault tolerant.
- Highly adaptable.
- Creative.

Why are these crucial?

Neurons: Basic Functional Unit of the Brain

- Dendrites receive input from upstream neurons.
- Ions flow in to make the cell positively charged.
- Once a firing threshold is reached, a spike is generated and transmitted along the axon.
- Axon terminals release neurotransmitters to relay the signal to the downstream neurons.

Propagation of Activation Across the Synapse

1. Action potential reaches axon terminal.
2. Neurotransmitters are released into synaptic cleft and bind to postsynaptic cell’s receptors.
3. Binding allows ion channels to open (Na\(^+\)), and Na\(^+\) ions flows in and makes the postsynaptic cell depolarize.
4. Once the membrane voltage reaches the threshold, an action potential is generated.

Lesson: neural activity propagation has a very complex cellular/molecular mechanism.

Abstraction of the Neuron in Neural Networks

- Input
- Connection weight
- Transfer function: \( f(\cdot) \)

Typical transfer functions: step-function or sigmoid.
**Typical Activation Functions**

- **Step** \( (x) \) = 1 if \( x \geq t \), 0 if \( x < t \)
- **Sign** \( (x) \) = +1 if \( x \geq 0 \), −1 if \( x < 0 \)
- **Sigmoid** \( (x) = \frac{1}{1+e^{-x}} \)

Note that **Step** \( t \) \( (x) = \) Step \( 0 \) \( (x - t) \), which we will simply call **Step** \( (x - t) \).

**More Activation Functions:** \( tanh(\frac{x}{2}) \)

- **Sigmoid** \( (x) = \frac{1}{1+e^{-x}} \)
- \( tanh(\frac{x}{2}) = \frac{1-e^{-x}}{1+e^{-x}} \)

---

**Classification of Neural Networks**

Teacher exists?

- Supervised (with teacher): perceptrons, backpropagation network, etc.
- Unsupervised (no teacher): self-organizing maps, etc.

Recurrent connections?

- Feed-forward: perceptrons, backpropagation network, etc.
- Recurrent: Hopfield network, Boltzmann machines, SRN (simple recurrent network), etc.

**Feedforward Networks**

- Perceptrons: single layer, threshold-gated.
- Backpropagation networks: multiple layers, sigmoid (or tanh) activation function.
Perceptrons

\[ f(\Sigma w_i x_i) \]

\[ a_i = \text{step}_t \left( \sum_{j=1}^{n} W_{j,i} a_j \right) \]

\[ = \text{step}_0 \left( \sum_{j=1}^{n} W_{j,i} a_j - t \right) \]

\[ = \text{step}_0 (t \times (-1) + \sum_{j=1}^{n} W_{j,i} a_j) \]

\[ = \text{step}_0 \left( \sum_{j=0}^{n} W_{j,i} a_j \right), \text{ where } W_{0,i} = t \text{ and } a_0 = -1 \]

Boolean Logic Gates with Perceptron Units

- Note that the activation function is the \( \text{step}_0(\cdot) \) function.
- Perceptrons can represent basic boolean functions.
- Thus, a network of perceptron units can compute any Boolean function.

What about XOR or EQUIV?

• A geometrical interpretation of this is:

\[ t = \frac{W_0}{W_1} x_0 + \frac{t}{W_1}, \]

where points on or above the line, the output is 1, and 0 for those below the line (when \( W_1 \) is positive). Compare with

\[ y = \frac{-W_0}{W_1} x + \frac{t}{W_1}. \]

Note: When dividing both sides with \( W_1 \), depending on the sign, the inequality can flip its direction (see previous page).

Limitation of Perceptrons

Perceptrons can only represent linearly-separable functions.
- Output of the perceptron:

\[ W_0 \times I_0 + W_1 \times I_1 - t \geq 0, \text{ then output is 1} \]

\[ W_0 \times I_0 + W_1 \times I_1 - t < 0, \text{ then output is 0} \]
Limitation of Perceptrons (cont’d)

- Thus, only functions where the points that result in 0 and 1 as output can be separated by a line can be represented by perceptrons.

- **Note:** the previous result is generalizable to functions of \( n \) arguments, i.e. perceptron with \( n \) inputs plus one threshold (or bias) unit.

---

Linear Separability (cont’d)

- For functions that take integer or real values as arguments and output either 0 or 1.
- Left: linearly-separable (i.e., can draw a straight line between the classes).
- Right: not linearly-separable (i.e., perceptrons cannot represent such a function)

---

Learning in Perceptrons

- The weights do not have to be calculated manually.
- We can train the network with (input,output) pair according to the following weight update rule:

\[
W_j \leftarrow W_j + \alpha \times I_j \times Err,
\]

where \( \alpha \) is the learning rate parameter, \( I_j \) is the input, and

\[
Err = DesiredOutput - NetworkOutput.
\]
Exercise: Implementing the Perceptron

- It is fairly easy to implement a perceptron.
- You can implement it in any programming language: C/C++, etc.
- Look for examples on the web, and JAVA applet demos.

Key Points

- The central nervous system: how it differs from conventional computers.
- Basic mechanism of synaptic information transfer
- Types of neural networks
- Perceptrons: basic idea, and the geometric interpretation. What is the limitation? How to train?

Overview

- Multilayer feed-forward networks
- Gradient descent search
- Backpropagation learning rule
- Evaluation of backpropagation
- Applications of backpropagation

Multilayer Feed-Forward Networks

- Proposed in the 1950s
- Proper procedure for training the network came later (1969) and became popular in the 1980s: back-propagation
Back-Propagation Learning Rule

- Back-prop is basically a **gradient descent** algorithm.
- The tough problem: output layer has explicit error measure, so finding the error surface is trivial. However, for the hidden layers, how much error each connection eventually cause at the output nodes is hard to determine.
- Backpropagation determines how to distribute the **blame** to each connection.

Gradient Descent

- We want to minimize the total error $E$ by tweaking the network weights.
- $E$ depends on $W_i$, thus by adjusting $W_i$, you can reduce $E$.
- Figuring out how to simultaneously adjust weights $W_i$ for all $i$ at once is practically impossible, so use an iterative approach.
- A sensible way is to reduce $E$ with respect to one weight $W_i$ at a time, proportional to the gradient (or slope) at that point.

\[
\Delta W_i = \alpha \times \left(-\frac{\partial E}{\partial W_i}\right),
\]

where $\alpha$ is the learning rate parameter.

- $E$ can be a function of many weights, thus the partial derivative is used in the above:

\[
E(W_1, W_2, \ldots, W_i, \ldots W_n, \ldots)
\]

Error function:

\[
E = \frac{1}{2} \sum_i (E_i)^2
\]

\[
E_i = T_i - O_i
\]

\[
= T_i - g \left( \sum_j W_{j,i} a_j \right),
\]

where $g(\cdot)$ is the sigmoid activation function, and $T_i$ the target.
Hidden to Output Weights (cont’d)

\[
\frac{\partial E}{\partial W_{j,i}} = \frac{\partial}{\partial W_{j,i}} \left( \frac{1}{2} \sum_i \left( T_i - g \left( \sum_j W_{j,i} a_j \right) \right)^2 \right)
\]

\[
= (T_i - O_i) \times \left( - \frac{\partial g(\sum_j W_{j,i} a_j)}{\partial W_{j,i}} \right)
\]

\[
= -(T_i - O_i) \times g'(\sum_j W_{j,i} a_j) \times a_j
\]

For easier calculation later on, we can rewrite:

\[
\frac{\partial E}{\partial W_{j,i}} = -(T_i - O_i) \times g'(\sum_j W_{j,i} a_j) \times a_j
\]

\[
= -a_j \times (T_i - O_i) \times g'(\sum_j W_{j,i} a_j)
\]

\[
= -a_j \times \Delta_i
\]

It is easy to verify \( g'(x) = g(x)(1 - g(x)) \) from \( g(x) = \frac{1}{1+e^{-x}} \), so we can reuse the \( g(x) \) value from the feed-forward phase in the feedback weight update.

Hidden to Output Weight Update

\[
\frac{\partial E}{\partial W_{j,i}} = -a_j \times \Delta_i
\]

where \( \Delta_i = (T_i - O_i) \times g'(\sum_j W_{j,i} a_j) \).

Input to Hidden Weights

\[
\frac{\partial E}{\partial W_{k,j}} = \frac{\partial}{\partial W_{k,j}} \left( \frac{1}{2} \sum_i \left( T_i - g \left( \sum_j W_{j,i} a_j \right) \right)^2 \right)
\]

\[
= \frac{\partial}{\partial W_{k,j}} \left( \frac{1}{2} \sum_i \left( T_i - g \left( \sum_j W_{j,i} (g(\sum_k W_{k,j} I_k)) \right) \right)^2 \right)
\]

However, this is too complex.
Input to Hidden Weights (cont’d)

Use the chain rule for easier calculation of the partial derivative:

\[
\frac{\partial E}{\partial W_{k,j}} = \frac{\partial E}{\partial a_j} \times \frac{\partial a_j}{\partial W_{k,j}}
\]

\[
= -\sum_i \left( (T_i - O_i) g'(\sum_j W_{j,i} a_j) W_{j,i} \right) \times g'(\sum_k W_{k,j} I_k) I_k
\]

\[
= -\sum_i (\Delta_i W_{j,i}) g'(\sum_k W_{k,j} I_k) I_k
\]

Back-Propagation: Summary

• Weight update:

\[
\Delta W_{x,y} \propto \Delta_y \times Input_x
\]

• The \(\Delta\)s:

\[
\Delta_x = Error_x \times g'(Weighted\_Sum_x)
\]

Thus, each node has its own \(\Delta\) and that is used to update the weights. These \(\Delta\)s are backpropagated for weight updates further below.

Input to Hidden Weight Update Rule

From \(\frac{\partial E}{\partial W_{k,j}}\), we can rename

\[
\sum_i (\Delta_i W_{j,i}) g'(\sum_k W_{k,j} I_k)\]

to be \(\Delta_j\), then the whole equation becomes:

\[
\frac{\partial E}{\partial W_{k,j}} = -\Delta_j I_k
\]

Thus the update rule becomes:

\[
W_{k,j} \leftarrow W_{k,j} + \alpha \times \Delta_j \times I_k
\]

General Case: More Than 2 Layers

• In general, the same rule for back-propagating \(\Delta\)s apply for multiple layer networks with more than two layers.

• That is, \(\Delta\) for a deep hidden unit can be determined by the product of the weighted sum of feedback \(\Delta\)s and the first derivative of feedforward weighted sum at the current unit.
Backpropagation Algorithm

1. Pick \((Input, Target)\) pair.

2. Using input, activate hidden and output layers through feed-forward activation.

3. At the output node, calculate the error \((T_i - O_i)\), and from that calculate the \(\Delta s\).

4. Update weights to the output layer, and backpropagate the \(\Delta s\).

5. Successively update hidden layer weights until Input layer has been reached.

6. Repeat step 1–5 until the total error goes below a set threshold.

Problems With Backprop

- Learning can be extremely slow: introduce momentum, etc.
- Network can be stuck in local minima: this is a common problem for any gradient-based method.

Other issues are: how to introduce new batches of data after the training has been completed.

Technical Issues in Training

- Batch vs. online training
  - Batch: accumulate weight updates for one epoch, and then update
  - Online: immediately apply weight updates after one input-output pair.

- When to stop training
  - Training set: use for training
  - Validation set: determine when to stop
  - Test set: use for testing performance

Backprop Application

- Speech generation: NetTALK (Sejnowski and Rosenberg, 1987)
- Character recognition: LeCun (1989)
- Driving a car: ALVINN, etc.

and many other Engineering applications – control, etc.
Demo: NetTALK

- I want to go to my grandmother's ...
- friend, sent, around, not, red, soon, doubt, key, attention, lost

Key Points

- Basic concept of a multi-layer feed-forward network.
- How hidden units know how much error they caused.
- Backprop is a gradient descent algorithm.
- Drawbacks of backprop.

Overview

- More on backprop
- Self-organizing maps

Another Application of Backpropagation: Image Compression

- Image compression
  1. target output is the same as the input.
  2. hidden layer units are fewer than the output (and input) layer units.
  3. the hidden layer forms the compressed representation.
To overcome the local minima problem:

- Adding momentum
\[
\Delta W_{j,i}(t) = \alpha \times \Delta_i \times I_j + \eta \times \Delta W_{j,i}(t-1)
\]

- Incremental update (as opposed to batch update) with random input-target order.

- Add a little bit of noise to the input.

- Allow increasing \( E \) with a small probability, as in Simulated Annealing.


---

**Backpropagation Exercise**

- **URL:** http://www.cs.tamu.edu/faculty/choe/src/backprop-1.6.tar.gz

- Untar and read the README file:

  ```bash
gzip -dc backprop-1.6.tar.gz | tar xvf -
  ```

- Run `make` to build (on departmental unix machines).

- Run `./bp conf/xor.conf` etc.

---

**Backpropagation: Example Results**

- Epoch: one full cycle of training through all training input patterns.

- OR was easiest, AND the next, and XOR was the most difficult to learn.

- Network had 2 input, 2 hidden and 1 output unit. Learning rate was 0.001.

---

**Backpropagation: Example Results (cont’d)**

Output to (0,0), (0,1), (1,0), and (1,1) form each row.
**Backpropagation: Things to Try**

- How does increasing the number of hidden layer units affect the (1) time and the (2) number of epochs of training?
- How does increasing or decreasing the learning rate affect the rate of convergence?
- How does changing the slope of the sigmoid affect the rate of convergence?
- Different problem domains: handwriting recognition, etc.

**Unsupervised Learning**

- No teacher signal (i.e. no feedback from the environment).
- The network must discover patterns, features, regularities, correlations, or categories in the input data and code them in the output.
- The units and connections must display some degree of **self-organization**.
- Unsupervised learning can be useful when there is **redundancy** in the input data.
- A data channel where the input data content is less than the channel capacity, there is redundancy.

**What Can Unsupervised Learning Do?**

- **Familiarity**: how similar is the current input to past inputs?
- **Principal Component Analysis**: find orthogonal basis vectors (or axes) against which to project high dimensional data.
- **Clustering**: \( n \) output class, each representing a distinct category. Each cluster of similar or nearby patterns will be classified as a single class.
- **Prototyping**: For a given input, the most similar output class (or exemplar) is determined.
- **Encoding**: application of clustering/prototyping.
- **Feature Mapping**: topographic mapping of input space onto output network configuration.

**Self-Organizing Map (SOM)**

- 1-D or 2-D layout of units.
- One reference vector for each unit.
- Unsupervised learning (no target output).
**SOM Algorithm**

1. Randomly initialize reference vectors $w_i$.
2. Randomly sample input vector $x$.
3. Find Best Matching Unit (BMU): 
   $$i(x) = \text{argmin}_j \| x - w_j \|$$
4. Update reference vectors:
   $$w_j \leftarrow w_j + \alpha \Lambda(j, i(x))(x - w_j)$$
   $\alpha$: learning rate 
   $\Lambda(j, i(x))$: neighborhood function of BMU.
5. Repeat steps 2 – 4.

**Typical Neighborhood Functions**

- **Gaussian**: 
  $$\Lambda(j, i(x)) = \exp\left(-\frac{(r_j - r_{i(x)})^2}{2\sigma^2}\right)$$
- **Flat**: 
  $$\Lambda(j, i(x)) = 1 \text{ if } |r_j - r_{i(x)}| \leq \sigma, \text{ and 0 otherwise.}$$
- $\sigma$ is called the neighborhood radius.

**Training Tips**

- Start with large neighborhood radius. 
  Gradually decrease radius to a small value.
- Start with high learning rate $\alpha$. 
  Gradually decrease $\alpha$ to a small value.

**Properties of SOM**

- **Approximation of input space.** 
  Maps continuous input space to discrete output space.
- **Topology preservation.** 
  Nearby units represent nearby points in input space.
- **Density mapping.** 
  More units represent input space that are more frequently sampled.
Performance Measures

• Quantization Error
  Average distance between each data vector and its BMU.
  \[ \epsilon_Q = \frac{1}{N} \sum_{j=1}^{N} \| x_j - w_{i(x_j)} \| \]

• Topographic Error
  The proportion of all data vectors for which first and second BMUs are not adjacent units.
  \[ \epsilon_T = \frac{1}{N} \sum_{j=1}^{N} u(x_j), \]
  \[ u(x) = 1 \text{ if the 1st and 2nd BMUs are not adjacent} \]
  \[ u(x) = 0 \text{ otherwise.} \]

Example: 2D Input / 2D Output

• Train with uniformly random 2D inputs.
  Each input is a point in Cartesian plane.
• Nodes: reference vectors (x and y coordinate).
• Edges: connect immediate neighbors on the map.

Different 2D Input Distributions

• What would the resulting SOM map look like?
• Why would it look like that?

High-Dimensional Inputs

SOM can be trained with inputs of arbitrary dimension.

• Dimensionality reduction: N-D to 2-D.
• Extracts topological features.
• Used for visualization of data.
Applications

- Data clustering and visualization.
- Optimization problems:
  - Traveling salesman problem.
- Semantic maps:
  - Natural language processing.
- Preprocessing for signal and image-processing.
  2. Phonetic map for speech recognition.

Exercise

1. What happens when $N_i(x)$ and $\alpha$ was reduced quickly vs. slowly?
2. How would the map organize if different input distributions are given?
3. For a fixed number of input vectors from real-world data, a different visualization scheme is required. How would you use the number of input vectors that best match each unit to visualize the property of the map?

Key Points

- How can backprop be improved?
- What are the various ways to apply backprop?
- SOM basic algorithm
- What kind of tasks is SOM good for?

Overview

- SOM demo
- Recurrent networks
- Genetic Algorithms
**SOM Example: Handwritten Digit Recognition**

- Preprocessing for feedforward networks (supervised learning).
- Better representation for training.
- Better generalization.

**SOM Demo**

Jochen Fröhlich's *Neural Networks with JAVA* page:
http://rfhs8012.fh-regensburg.de/saj39122/jfroehl/diplom/e-index.html

Check out the Sample Applet link.

**SOM Demo: Traveling Salesman Problem**

Using Fröhlich’s SOM applet:

- 1D SOM map \((1 \times n\), where \(n\) is the number of nodes).
- 2D input space.
- Initial neighborhood radius of 8.
- Stop when radius \(< 0.001\).
- Try 50 nodes, 20 input points.

Click on [Parameters] to bring up the config panel. After the parameters are set, click on [Reset] in the main applet, and then [Start learning].

**SOM Demo: Space Filling in 2D**

Using Fröhlich’s SOM applet:

- 1D SOM map \((1 \times n\), where \(n\) is the number of nodes).
- 2D input space.
- Initial neighborhood radius of 100.
- Stop when radius \(< 0.001\).
- Try 1000 nodes, and 1000 input points.
**SOM Demo: Space Filling in 3D**

Using Fröhlich’s SOM applet:

- 2D SOM map \((n \times n)\), where \(n\) is the number of nodes.
- 2D input space.
- Initial neighborhood radius of 10.
- Stop when radius < 0.001.
- Try \(30 \times 30\) nodes, and 500 input points. Limit the \(y\) range to 15.

Also try \(50 \times 50\), 1000 input points, and 16 initial radius.

---

**Recurrent Networks**

Connection graph can contain cycles, e.g. reciprocal connections: i.e. not strictly feed-forward.

- Statistical mechanics based models (associative or content-addressable memory): Hopfield network, Boltzmann machines, etc.
- Sequence encoding: Simple Recurrent Network, etc.
- Other biologically motivated networks: laterally connected self-organizing maps, etc.

---

**Simple Recurrent Network (Elman Network)**

- Sequence encoding.
- Hidden layer activation from previous time step used as input.
- Use standard back-propagation learning.

---

**SRN Example**

Example input and target sequence: output 1 when two 1s appear in a row.

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---

\[
\text{Input: } 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 1 0 1
\]
\[
\text{Target: } 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0
\]
Genetic Algorithms

Population of Genes Selection

\[ f = 100 \]
\[ f = 10 \]
\[ f = 5 \]
\[ f = 20 \]

Mating (Crossover)

New Generation

Random Mutation

Evolution as a problem solving strategy:

- population of solutions, where each chromosome represent an individual
- selection based on fitness function: survival of the fittest
- mating (cross-over) and reproduction
- random mutation

Properties of Genetic Algorithms

- each chromosome encodes a solution
- similar to hill-climbing search
- parallel search
- works for both immediate or delayed reward

Designing a GA Solution to a Problem

There are many different issues:

- What is the fitness function?
- How is an individual chromosome represented (how to encode) and what does it represent?
- How are individuals selected?
- How do individuals reproduce?

Issues in GA: Diversity

How to maintain diversity:

- Letting only successful ones to reproduce can seriously reduce the gene pool and an epidemic can wipe out the whole population: solution can not generalize in new and unexpected conditions.
- Converged population can often times be found at local minima, not at the global optimum.
More Issues in GA

• Cross-over strategy: success depends on how genes are encoded (or represented).
• Not too much theoretical understanding about why it works so well.
• Crevices in fitness landscape: similar to spikes in hill climbing.
• How to combine learning with evolution.
• How to use cultural leverage.

GA as a Learning Algorithm

• An individual chromosome may not seem to learn, but when we look at the evolution of individuals over time, they can be seen as adapting, and thus learning to cope with the environment.
• If each individual encodes a function rather than a simple solution, the above point becomes clearer. At each generation, the parameters in the function can be seen as being adapted.
• Fitness can then be measured by using the function with the given parameters in specific tasks.

GA as a Learning Algorithm: Neuroevolution

Evolving neural networks:
• Genes encode neural networks (connection topology and connection weights).
• Evaluate, select, and reproduce new population of neural networks.
Problem: individual neurons performing good work may get lost.

Neuroevolution: Evolving Individual Neurons

SANE: Moriarty and Miikkulainen
• Genes encode individual neurons.
• Neurons solve sub-problems and the ones that solve the problem well gets a chance to participate in a network in the next generation.
• Better diversity is maintained.
GA Demo

Neuroevolution:
http://www.cs.utexas.edu/users/nn/
- Generation of melodies (Chen and Miikkulainen)
- Gaming AI; harvesters and predators (Stanley and Miikkulainen)
- Non-markovian control (Gomez and Miikkulainen)

Key Points
- SOM: Try out the effects of different parameters, network size, 1D or 2D map, neighborhood radius, etc.
- Simple recurrent networks: how can it encode sequences, how is it different from standard backprop and who similar is it?
- Genetic algorithms basics.
- What are the issues to be solved in genetic algorithms.