Example: A 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 \( i \).

Learning = Training in Neural Networks

- 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.

Artificial Intelligence

Learning and Neural Networks

Readings: Chapter 19 & 20.5 of Russell & Norvig

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.
**The Perceptron Learning Method**

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

- 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 $O$.

**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**

- 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
  \[ Err = T - O \]

**Normalizing Unit Thresholds.**

- Notice that, if $t$ is the threshold value of the output unit, then
  \[ \text{step}(\sum_{j=1}^{n} W_j I_j) = \text{step}_0(\sum_{j=0}^{n} W_j I_j) \]
  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

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:

\[
W_j \leftarrow W_j + \alpha \times I_j \times (T - O)
\]

where \( \alpha \) is a positive constant, the learning rate.

Theoretic Background

Learn by adjusting weights to reduce error on training set.

The squared error for an example with input \( x \) and true output \( y \) is

\[
E = \frac{1}{2} Err^2 = \frac{1}{2} (y - h(x))^2 ,
\]

Perform optimization search by gradient descent:

\[
\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j} \left( y - g(\sum_{j=0}^{n} W_j x_j) \right)
= -Err \times g'(in) \times x_j
\]

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., \( \text{step}_{-3} \)). Finally, set up parameters (i.e., \( W_i = 0 \)) and start to learn:

Assuming \( \alpha = 1 \), we have \( \text{Sum} = \sum_{i=1}^{5} W_i I_i, \text{Out} = \text{step}_{-3}(\text{Sum}), Err = T - \text{Out} \), and \( W_j \leftarrow W_j + I_j \times Err \).

| \( I_1 \) | \( I_2 \) | \( I_3 \) | \( I_4 \) | \( I_5 \) | \( T \) | \( W_1 \) | \( W_2 \) | \( W_3 \) | \( W_4 \) | \( W_5 \) | \( \text{Sum} \) | \( \text{Out} \) | \( \text{Err} \) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Learning a 5-place Minority Function

The same as the last example, except that $\alpha = 0.5$ instead of $\alpha = 1$. Sum = $\sum_{i=1}^{5} W_i I_i$, Out = step-3(Sum),

$$Err = T - Out,$$

$W_j \leftarrow W_j + I_j \times Err.$

<table>
<thead>
<tr>
<th></th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>$I_4$</th>
<th>$I_5$</th>
<th>$T$</th>
<th>$W_1$</th>
<th>$W_2$</th>
<th>$W_3$</th>
<th>$W_4$</th>
<th>$W_5$</th>
<th>Sum</th>
<th>Out</th>
<th>Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_2$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_6$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_7$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$e_8$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Multilayer Perceptrons

Layers are usually fully connected; numbers of hidden units typically chosen by hand.

<table>
<thead>
<tr>
<th></th>
<th>$a_i$</th>
<th>$W_{i,j}$</th>
<th>$a_j$</th>
<th>$W_{k,j}$</th>
<th>$a_k$</th>
</tr>
</thead>
</table>

Back-propagation learning

Output layer: same as for single-layer perceptron,

$$W_{j,i}' \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Hidden layer: back-propagate the error from the output layer:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i .$$

Update rule for weights in hidden layer:

$$W_{k,j}' \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j .$$

(Many neuroscientists deny that back-propagation occurs in the brain)

Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers
Back-propagation derivation

The squared error on a single example is defined as

$$ E = \frac{1}{2} \sum_i (y_i - a_i)^2, $$

where the sum is over the nodes in the output layer.

$$ \frac{\partial E}{\partial W_{k,j}} = -\sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left( \sum_j W_{j,i} a_j \right) $$

$$ = -\sum_i \Delta_i W_{j,i} \frac{\partial a_i}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} g'(in_i) \frac{\partial}{\partial W_{k,j}} \left( \sum_j W_{j,i} a_j \right) $$

$$ = -\sum_i \Delta_i W_{j,i} g'(in_i) \frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} g'(in_i) a_k = -a_k \Delta_j $$

Decision trees

One possible representation for hypotheses. E.g., here is the “true” tree for deciding whether to wait:

Classification of examples is positive (T) or negative (F)
Expressiveness

Decision trees can express any function of the input attributes. E.g., for Boolean functions, each truth table row is a path from root to leaf:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A xor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Trivially, there exists a consistent decision tree for any training set with one path to leaf for each example (unless \( f \) nondeterministic in \( x \)) but it probably won’t generalize to new examples.

Prefer to find more compact decision trees.

Hypothesis spaces

- How many distinct decision trees with \( n \) Boolean attributes
  - \( \# \) = number of Boolean functions
  - \( \# \) = number of distinct truth tables with \( 2^n \) rows = \( 2^{2^n} \)
Hypothesis spaces

- How many distinct decision trees with \( n \) Boolean attributes
  - = number of Boolean functions
  - = number of distinct truth tables with \( 2^n \) rows = \( 2^{2^n} \)
  - E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees
- How many purely conjunctive hypotheses (e.g., \( \text{Hungry} \land \neg \text{Rain} \))
  - Each attribute can be in (positive), in (negative), or out.

Decision tree learning

- Aim: find a small tree consistent with the training examples
- Idea: (recursively) choose “most significant” attribute as root of (sub)tree
- Reason: A good attribute splits the examples into subsets that are (ideally) “all positive” or “all negative”

```
None  Some  Full
Patrons?  French  Italian  Thai  Burger

Type?
```

Patrons? is a better choice—gives information about the classification

Hypothesis spaces

- How many distinct decision trees with \( n \) Boolean attributes
  - = number of Boolean functions
  - = number of distinct truth tables with \( 2^n \) rows = \( 2^{2^n} \)
  - E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees
- How many purely conjunctive hypotheses (e.g., \( \text{Hungry} \land \neg \text{Rain} \))
  - Each attribute can be in (positive), in (negative), or out.
  - \( 3^n \) distinct conjunctive hypotheses
Information

- Suppose we have $p$ positive and $n$ negative examples at the root. Hence $I((p/(p+n), n/(p+n)))$ bits needed to classify a new example. E.g., for 12 restaurant examples and $p = n = 6$, we need 1 bit.
- Suppose an attribute $A$ splits the examples $E$ into subsets $E_i$, each of which (we hope) needs less information to complete the classification.
- Let $E_i$ have $p_i$ positive and $n_i$ negative examples. Then $I((p_i/(p_i+n_i), n_i/(p_i+n_i)))$ bits needed to classify $E_i$.
- The remaining bits after choosing $A$ will be
  \[
  \text{Remainder}(A) = \sum_i \frac{p_i + n_i}{p + n} I((p_i/(p_i+n_i), n_i/(p_i+n_i)))
  \]
  Idea: Choose $A$ such that $\text{Remainder}(A)$ is minimal.

Example contd.

Decision tree learned from the 12 examples:

Substantially simpler than “true” tree—a more complex hypothesis isn’t justified by small amount of data
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.