# Chapter 13
# Neural Networks

Chapter 3 described reactive behaviors inspired by the work of Valentino Braitenberg. The control of simple Braitenberg vehicles is very similar to the control of a living organism by its biological *neural network*. This term refers to the nervous system of a living organism, including its brain and the nerves that transmit signals through the body. Computerized models of neural networks are an active topic of research in artificial intelligence. *Artificial neural networks (ANNs)* enable complex behavior to be implemented using a large number of relatively simple abstract components that are modeled on neurons, the components of biological neural networks. This chapter presents the use of ANNs to control the behavior of robots.

Following a brief overview of the biological nervous system in Sect. 13.1, Sect. 13.2 defines the ANN model and Sect. 13.3 shows how it can be used to implement the behavior of a Braitenberg vehicle. Section 13.4 presents different network topologies. The most important characteristic of ANNs is their capability for learning which enables them to adapt their behavior. Section 13.5 presents an overview of learning in ANNs using the Hebbian rule.

## 13.1 The Biological Neural System

The nervous system of living organisms consists of cells called *neurons* that process and transmit information within the body. Each neuron performs a simple operation, but the combination of these operations leads to complex behavior. Most neurons are concentrated in the brain, but others form the nerves that transmit signals to and from the brain. In *vertebrates* like ourselves, many neurons are concentrated in the spinal cord which efficiently transmit signals throughout the body. There are an immense number of neurons in a living being: the human brain has about 100 billion neurons while even a mouse brain has about 71 million neurons [2].

© The Author(s) 2018

M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_13

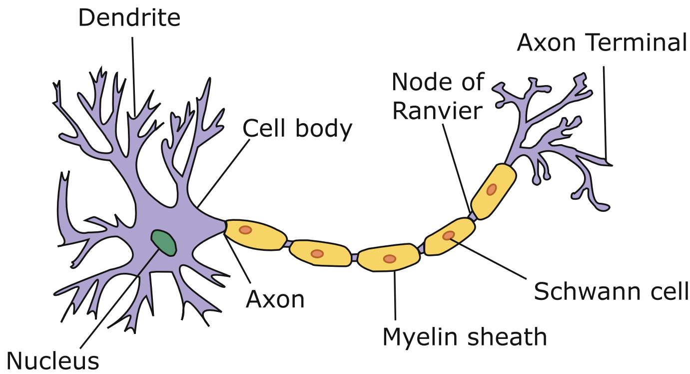


**Fig. 13.1** Structure of a neuron. *Source* https://commons.wikimedia.org/wiki/File:Neuron.svg by Dhp1080 [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL https://en.wikipedia.org/wiki/en:GNU_Free_Documentation_License)], via Wikimedia Commons

Figure 13.1 shows the structure of a neuron. It has a main body with a *nucleus* and a long fiber called an *axon* that allows one neuron to connect with another. The body of a neuron has projections called *dendrites*. Axons from other neurons connect to the dendrites through *synapses*. Neurons function through biochemical processes that are well understood, but we can abstract these processes into *pulses* that travel from one neuron to another. Input pulses are received through the synapses into the dendrites and from them to the body of the neuron, which processes the pulses and in turn transmits an output pulse through the axon. The processing in the body of a neuron can be abstracted as a function from the input pulses to an output pulse, and the synapses regulate the transmission of the signals. Synapses are adaptive and are the primary element that makes memory and learning possible.

## 13.2 The Artificial Neural Network Model

An artificial neuron is a mathematical model of a biological neuron (Fig. 13.2a, b; see Table 13.1 for a list of the symbols appearing in ANN diagrams). The body of the neuron is a node that performs two functions: it computes the sum of the weighted input signals and it applies an output function to the sum. The input signals are multiplied by weights before the sum and output functions are applied; this models the synapse. The output function is usually nonlinear; examples are: (1) converting the neuron's output to a set of discrete values (turn a light on or off); (2) limiting the range of the output values (the motor power can be between −100 and 100; (3) normalizing the range of output values (the volume of a sound is between 0 (mute) and 1 (maximum).

Artificial neurons are analog models, that is, the inputs, outputs, weights and functions can be floating point numbers. Here we start with an unrealistic activity

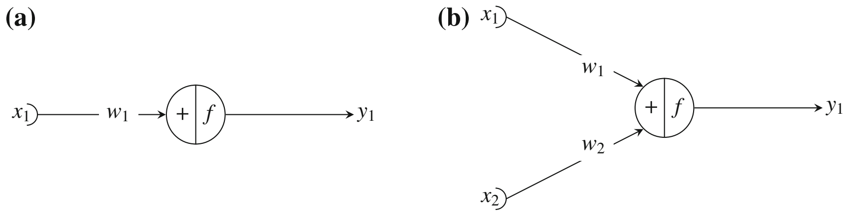


**Fig. 13.2** **a** ANN: one neuron with one input. **b** ANN: one neuron with two inputs

**Table 13.1** Symbols used in ANN diagrams

| Symbol | Meaning |
|---|---|
| $f$ | Neuron output function |
| $+$ | Sum of the inputs |
| $x_i$ | Inputs |
| $y_i$ | Outputs |
| $w_i$ | Weights for the inputs |
| 1 | Constant input of value 1 |

that demonstrates how artificial neurons work within the familiar context of digital logic gates.

Figure 13.3a shows an artificial neuron with two inputs, $x_1$ and 1, and one output $y$. The meaning of the input 1 is that the input is not connected to an external sensor, but instead returns a constant value of 1. The input value of $x_1$ is assumed to be 0 or 1. The function $f$ is:

$$\begin{aligned} f(x) &= 0 \quad \text{if} \;\; x < 0 \\ f(x) &= 1 \quad \text{if} \;\; x \geq 0\,. \end{aligned}$$

Show that with the given weights the neuron implements the logic gate for not.

**Activity 13.1: Artificial neurons for logic gates**

- The artificial neuron in Fig. 13.3b has an additional input $x_2$. Assign weights $w_0$, $w_1$, $w_2$ so that $y$ is 1 only if the values of $x_1$ or $x_2$ (or both) are 1. This implements the logic gate for or.
- Assign weights $w_0$, $w_1$, $w_2$ so that $y$ is 1 only if the values of $x_1$ and $x_2$ are both 1. This implements the logic gate for and.
- Implement the artificial neurons for logic gates on your robot. Use two sensors, one for $x_1$ and one for $x_2$. Use the output $y$ (mapped by $f$, if necessary) so that an output of 0 gives one behavior and an output of 1 another behavior, such as turning a light on or off, or starting and stopping the robot.

The following activity explores analog processing in an artificial neuron.

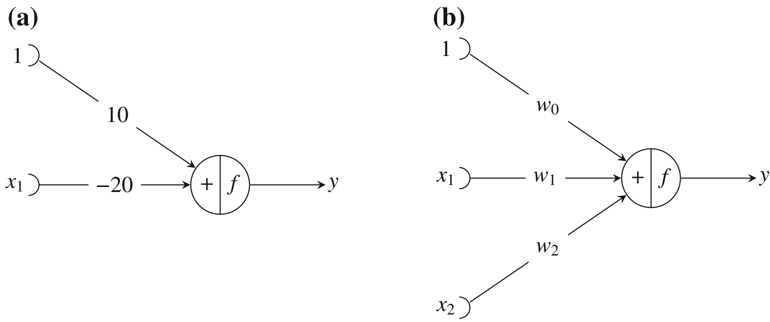


**Fig. 13.3** **a** Artificial neuron for the not gate. **b** Artificial neuron for the and and or gates

**Activity 13.2: Analog artificial neurons**

- Implement the artificial neuron shown in Fig. 13.2a so that it demonstrates the following behavior. The input to the neuron will be the reading of a proximity sensor at the front of the robot. The output will be one or both of the following: (1) the intensity of a light on the robot or the volume of the sound from a speaker on the robot; (2) the motor power applied to both the left and right motors so that the robot retreats from an object detected by the sensor.
- The output value will be proportional to the input value: the closer the object, the greater the intensity (or volume); the closer the object, the faster the robot retreats from the object.
- Modify the implementation so that there are two inputs from two proximity sensors (Fig. 13.2b). Give different values to the two weights $w_1, w_2$ and show that the sensor connected to the input with the larger weight has more effect on the output.

## 13.3 Implementing a Braitenberg Vehicle with an ANN

Figure 13.4 shows a robot inspired by a Braitenberg vehicle whose behavior is implemented using a simple neural network. We describe the ANN in detail and then give several activities that ask you to design and implement the algorithm.

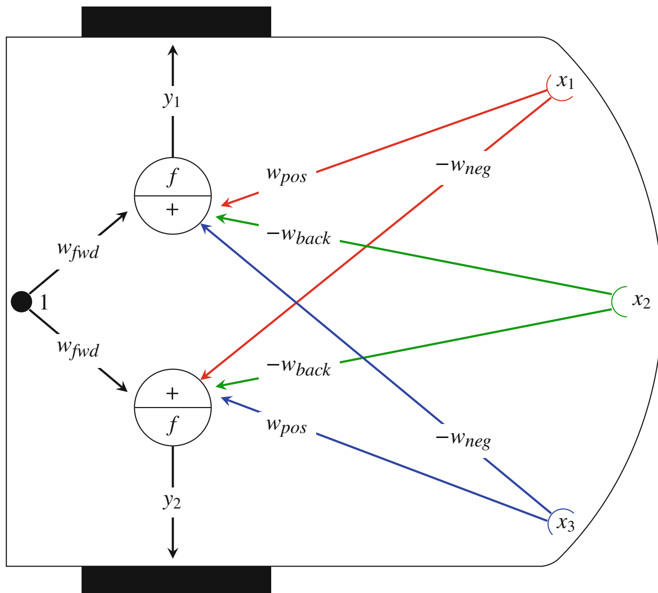


**Fig. 13.4** Neural network for obstacle avoidance

**Specification (obstacle avoidance)**:
The robot has three forward-facing sensors.

- The robot moves forwards unless it detects an obstacle.
- If the obstacle is detected by the center sensor, the robot moves slowly backwards.
- If the obstacle is detected by the left sensor, the robot turns right.
- If the obstacle is detected by the right sensor, the robot turns left.

Figure 13.4 shows the two neurons whose outputs control the power sent to the motors of the wheels of the robot. Table 13.2 lists the symbols used in the figure.

Each neuron has four inputs. The function $f$ must be nonlinear in order to limit the maximum forward and backward speeds. The large dot at the back of the robot

**Table 13.2** Symbols in Fig. 13.4 in addition to those in Table 13.1

| Symbol | Meaning |
|---|---|
| $w_{fwd}$ | Weight for forward movement |
| $w_{back}$ | Weight for backward movement |
| $w_{pos}$ | Weight for positive wheel rotation |
| $w_{neg}$ | Weight for negative wheel rotation |

denotes a constant input of 1 that is weighted by $w_{fwd}$. This ensures that in the absence of signals from the sensors, the robot will move forwards. When implementing the ANN you need to find a weight so that the output motor powers are reasonable in the absence input from the sensors. The weight should also ensure that the constant input is similar to inputs from the sensors.

The $x_1, x_2, x_3$ values come from the sensors that return zero when there is no object and an increasing positive value when approaching an object. The center sensor is connected to both neurons with a negative weight $-w_{back}$ so that if an obstacle is detected the robot will move backwards. This weight should be set to a value that causes the robot to move backwards slowly.

The left and right sensors are connected to the neurons with a positive weight for the neuron controlling the near wheel and a negative weight for the neuron controlling the far wheel. This ensures that robot turns away from the obstacle.

The following activity asks you to think about the relative values of the weights.

**Activity 13.3: ANN for obstacle avoidance: design**

- What relation must hold between $w_{fwd}$ and $w_{back}$?
- What relation must hold between $w_{fwd}$ and $w_{pos}$ and between $w_{fwd}$ and $w_{neg}$?
- What relation must hold between $w_{back}$ and $w_{pos}$ and between $w_{back}$ and $w_{neg}$?
- What relation must hold between $w_{pos}$ and $w_{neg}$?
- What happens if the obstacle is detected by both the left and center sensors?

In the following activities, you will have to experiment with the weights and the functions to achieve the desired behavior. Your program should use a data structure like an array so that it is easy to change the values of the weights.

**Activity 13.4: ANN for obstacle avoidance: implementation**

- Write a program for obstacle avoidance using the ANN in Fig. 13.4.

**Activity 13.5: ANN for obstacle attraction**

- Write a program to implement obstacle attraction using an ANN:
  - The robot moves forwards.
  - If the center sensor detects that the robot is *very close* to the obstacle, it stops.
  - If an obstacle is detected by the left sensor, the robot turns left.
  - If an obstacle is detected by the right sensor, the robot turns right.

## 13.4 Artificial Neural Networks: Topologies

The example in the previous section is based on an artificial neural network composed of a single layer of two neurons, each with several inputs and a single output. This is a very simple topology for an artificial neural network; many other topologies can implement more complex algorithms (Fig. 13.5). Currently, ANNs with thousands or even millions of neurons arranged in many layers are used to implement *deep learning*. In this section we present an overview of some ANN topologies.

### *13.4.1 Multilayer Topology*

Figure 13.6a shows an ANN with several layers of neurons. The additional layers can implement more complex computations than a single layer. For example, with a single layer it is not possible to have the robot move forward when only one sensor detects an obstacle and move backwards when several sensors detect an obstacle. The reason

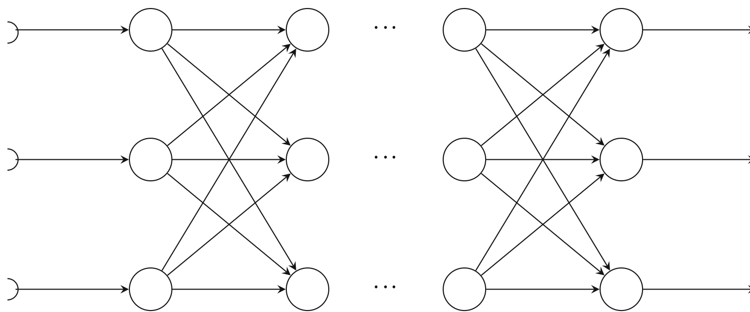

**Fig. 13.5** Neural network for deep learning

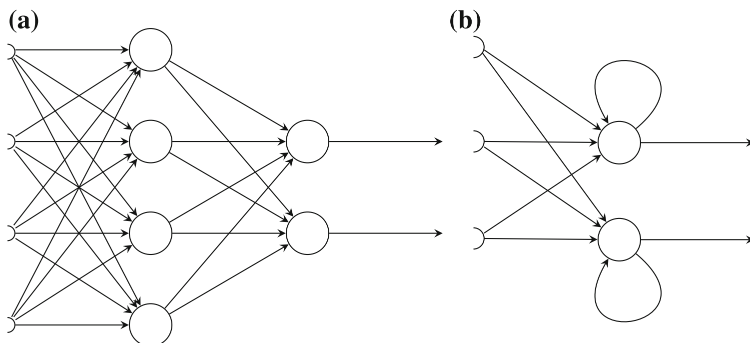


**Fig. 13.6** **a** Multilayer ANN. **b** ANN with memory

is that the function in a single layer linking the sensors and the motors is monotonic, that is, it can cause the motor to go faster when the sensor input increases or slower when the sensor input increases, but not both. The layer of neurons connected to the output is called the *output layer* while the internal layers are called the *hidden layers*.

**Activity 13.6: Multilayer ANNs**

- The goal of this activity is to understand how multilayer ANNs can perform computations that a single-layer ANN cannot. For the activity, assume that the inputs $x_i$ are in the range $-2.0$ to $2.0$, the weights $w_i$ are in the range $-1.0$ to $1.0$, and the functions $f$ limit the output values to the range $-1.0$ to $1.0$.
- For the ANN consisting of a single neuron (Fig. 13.2a) with $w_1 = -0.5$, compute $y_1$ for inputs in increments of 0.2: $x_1 = -2.0, -1.8, \ldots, 0.0, \ldots, 1.8, 2.0$. Plot the results in a graph.
- Repeat the computation for several values of $w_1$. What can you say about the relationship between the output and the input?
- Consider the two-layer ANN shown in Fig. 13.7 with weights:

$$w_{11} = 1,\ w_{12} = 0.5,\ w_{21} = 1,\ w_{22} = -1\,.$$

Compute the values and draw graphs of the outputs of the neurons of the hidden layer (the left neurons) and the output layer (the right neuron). Can you obtain the same output from an ANN with only one layer?

**Activity 13.7: Multilayer ANN for obstacle avoidance**

- Design an ANN that implements the following behavior of a robot: There are two front sensors. When an object is detected in front of one of the sensors, the robot turns to avoid the object, but when an object is detected by both sensors, the robot moves backwards.

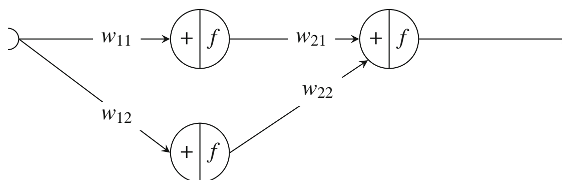


**Fig. 13.7** Two-layer ANN

### 13.4.2 Memory

An artificial neural network can have *recurrent connections* from the output of a neuron to the input of a neuron in the same layer (including itself). Recurrent connections can be used to implement memory. Consider the Braitenberg vehicle for obstacle avoidance (Fig. 13.4). It only turns when obstacles are detected by the sensors. When they are no longer detected, the robot does not continue to turn. By adding recurrent connections we can introduce a memory effect that causes the robot to continue turning. Suppose that each of the sensors causes an input of 0.75 to the neurons and that causes the output to be saturated to 1.0 by the non-linear output function. If the sensors no longer detect the obstacle, the inputs become 0, but the recurrent connection adds an input of 1.0 so the output remains 1.0.

**Activity 13.8: ANN with memory**

- Consider the network in Fig. 13.6b with an output function that saturates to 0 and 1. The inputs and most weights are also between 0 and 1. What happens if the weight of the recurrent connections in the figure is higher than 1? What happens if it is between 0 and 1?
- Modify the implementation of the network in Fig. 13.4 to add recurrent connections on the two output neurons. What is their effect in the obstacle-avoidance behavior of the robot?

### 13.4.3 Spatial Filter

A camera is a sensing device constructed from a large number of adjacent sensors (one for each pixel). The values of the sensors can be the inputs to an ANN with a large number of neurons in the first layer (Fig. 13.8). Nearby pixels will be input to adjacent neurons. The network can be used to extract local features such as differences of intensity between adjacent pixels in an image, and this local property can be used for tasks like identifying edges in the image. The number of layers may be one or more. This topology of neurons is called a *spatial filter* because it can be used as a filter before a layer that implements an algorithm for obstacle avoidance.

**Example** The ANN in Fig. 13.8 can be used to distinguish between narrow and wide objects. For example, both a leg of a chair and a wall are detected as objects, but the former is an obstacle that can be avoided by a sequence of turns whereas a wall cannot be avoided so the robot must turn around or follow the wall.

Suppose that the leg of the chair is detected by the middle sensor with a value of 60, but since the leg is narrow the other sensors return the value 0. The output values of the ANN (from top to bottom) are:

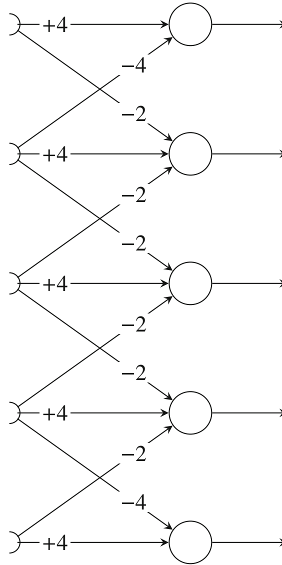


**Fig. 13.8** ANN for spatial filtering

$$(0 \times 4) + (0 \times -4) = 0$$
$$(0 \times -2) + (0 \times 4) + (60 \times -2) = -120$$
$$(0 \times -2) + (60 \times 4) + (0 \times -2) = +400$$
$$(60 \times -2) + (0 \times 4) + (0 \times -2) = -120$$
$$(0 \times 4) + (0 \times -4) = 0\,.$$

When the robot approaches a wall all the sensors will return more or less the same values, say, 45, 50, 40, 55, 50. The output values of the ANN are:

$$(45 \times 4) + (50 \times -4) = -20$$
$$(45 \times -2) + (50 \times 4) + (40 \times -2) = +30$$
$$(50 \times -2) + (40 \times 4) + (55 \times -2) = -50$$
$$(40 \times -2) + (55 \times 4) + (50 \times -2) = +40$$
$$(55 \times -4) + (50 \times 4) = -20\,.$$

Even though 48, the average value returned by the sensors detecting the wall, is about the same as the value 60 returned when detecting the leg of the chair, the outputs of the ANNs are clearly distinguishable. The first set of values has a high peak value surrounded by neighbors with large negative values, while second set is a relatively flat set of values in the range −50 to 40. The layer of neurons can confidently identify whether the object is narrow or wide.

**Activity 13.9: ANN for spatial filtering**

- Implement the ANN for spatial filtering in Fig. 13.8.
- The inputs to the ANN are the readings of five proximity sensors facing forwards. If only one sensor detects an object, the robot turns to face the object. If the center sensor is the one that detects the object, the robot moves forwards.
- Implement three behaviors of the robot when it detects a wall defined as all five sensors detecting an object:
  - The robot stops.
  - The robot moves forwards.
  - The robot moves backwards.

  Remember that there are no if-statements in an artificial neural network; you can only add additional neurons or change the weights associated with the inputs of the neurons. Look again at Activity 13.7 which used two levels of neurons to implement a similar behavior.
- The implementation will involve adding two additional neurons whose inputs are the outputs of the first layer. The output of the first neuron will set the power setting of the left motor and the output of the second neuron will set the power setting of the right motor.
- What happens if an object is detected by two adjacent sensors?

## 13.5 Learning

Setting the weights manually is difficult even for very small networks such as those presented in the previous sections. In biological organisms synapses have a plasticity that enables *learning*. The power of artificial neural networks comes from their ability to learn, unlike ordinary algorithms that have to be specified to the last detail. There are many techniques for learning in ANNs; we describe one of the simpler techniques in this section and show how it can be used in the neural network for obstacle avoidance.

### *13.5.1 Categories of Learning Algorithms*

There are three main categories of learning algorithms:

- **Supervised learning** is applicable when we know what output is expected for a set of inputs. The error between the desired and the actual outputs is used to correct the weights in order to reduce the error. Why is it necessary to train a network if we already know how it should behave? One reason is that the network is required to provide outputs in situations for which it was not trained. If the weights are adjusted so that the network behaves correctly on known inputs, it is reasonable to assume that its behavior will be more or less correct on other inputs. A second reason for training a network is to simplify the learning process: rather than directly relate the outputs $\{y_i\}$ to specific values of the inputs $\{x_i\}$, it is easier to place the network in several different situations and to tell it which outputs are expected in each situation.
- In **reinforcement learning** we do not specify the exact output value in each situation; instead, we simply tell the network if the output it computes is good or not. Reinforcement is appropriate when we can easily distinguish correct behavior from incorrect behavior, but we don't really care what the exact output is for each situation. In the next section, we present reinforcement learning for obstacle avoidance by a robot; for this task it is sufficient that the robot avoid the obstacle and we don't care what motor settings are output by the network as long as the behavior is correct.
- **Unsupervised learning** is learning without external feedback, where the network adapts to a large number of inputs. Unsupervised learning is not appropriate for achieving specified goals; instead, it is used in classification problems where the network is presented with raw data and attempts to find trends within the data. This approach to learning is the topic of Chap. 14.

### *13.5.2 The Hebbian Rule for Learning in ANNs*

The *Hebbian rule* is a simple learning technique for ANNs. It is a form of reinforcement learning that modifies the weights of the connections between the neurons. When the network is doing something good we reinforce this good answer: if the output value of two connected neurons is similar, we increase the weight of the connection linking them, while if they are different, we decrease the weight. If the robot is doing something wrong, we can either decrease the weights of similar connected neurons or do nothing.

The change in the weight of the connection between neuron $k$ and neuron $j$ is described by the equation:

$$\Delta w_{kj} = \alpha\, y_k\, x_j\ ,$$

where $w_{kj}$ is the weight linking the neurons $k$ and $j$, $\Delta w_{kj}$ is the change of $w_{kj}$, $y_k$ is the output of neuron $k$ and $x_j$ the input of neuron $j$, and $\alpha$ is a constant that defines the speed of learning.

The Hebbian rule is applicable under two conditions:

- The robot is exploring its environment, encountering various situations, each with its own inputs for which the network computes a set of outputs.
- The robot receives information on which behaviors are good and which are not.

The evaluation of the quality of the robot's behavior can come from a human observer manually giving feedback; alternatively, an automatic system can be used to evaluate the behavior. For example, in order to teach the robot to avoid obstacles, an external camera can be used to observe the robot and to evaluate its behavior. The behavior is classified as bad when the robot is approaching an obstacle and as good when the robot is moving away from all the obstacles. It is important to understand that what is evaluated is not the *state* of the robot (close to or far from an obstacle), but rather the *behavior* of the robot (approaching or avoiding an obstacle). The reason is that the connections of the neural network generate behavior based on the state as measured by the sensors.

**Learning to avoid an obstacle**

Suppose that we want to teach a robot to avoid an obstacle. One way would be to let the robot move randomly in the environment, and then touch one key when it successfully avoids the obstacle and another when it crashes into the obstacle. The problem with this approach is that it will probably take a very long time for the robot to exhibit behavior that can be definitely characterized as positive (avoiding the obstacle) or negative (crashing it into the obstacle).

Alternatively, we can present the robot with several known situations and the required behavior: (1) detecting obstacle on the left and turning right is good; (2) detecting an obstacle on the right and turning left is good; (3) detecting an obstacle in front and moving backwards is good; (4) detecting an obstacle in front and moving forwards is bad.

This looks like supervised learning but it is not, because the feedback to the robot is used only to reinforce the weights linked to good behavior. Supervised learning would consist in quantifying the error between the desired and actual outputs (to the motors in this case), and using this error to adjust the weights to compute exact outputs. Feedback in reinforcement learning is binary: the behavior is good or not.

**The algorithm for obstacle avoidance**

Let us now demonstrate the Hebbian rule for learning on the problem of obstacle avoidance. Figure 13.9 is similar to Fig. 13.4 except that proximity sensors have been added to the rear of the robot. We have also changed the notation for the weights to make them more appropriate for expressing the Hebbian rule; in particular, the negative signs have been absorbed into the weights.

The obstacle-avoidance algorithm is implemented using several concurrent processes and will be displayed as a set of three algorithms. Algorithm 13.1 implements an ANN which reads the inputs from the sensors and computes the outputs to the motors. The numbers of inputs and outputs are taken from Fig. 13.9. Algorithm 13.2 receives evaluations of the robot's behavior from a human. Algorithm 13.3 performs the computations of the Hebbian rule for learning.

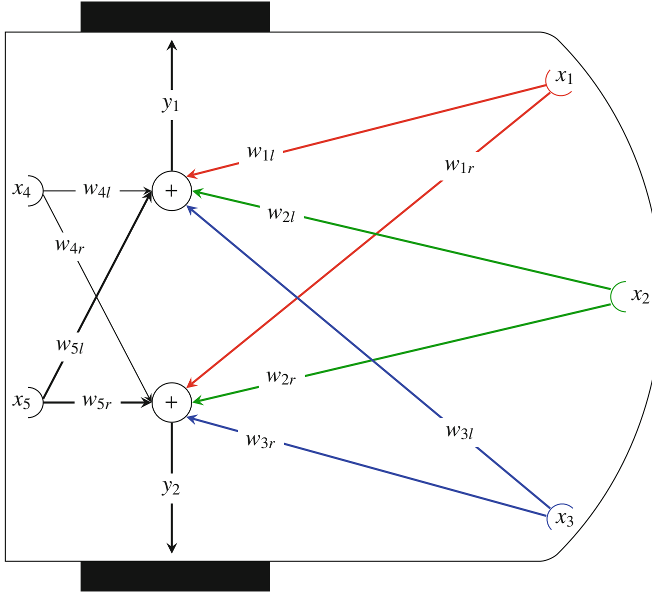


**Fig. 13.9** Neural network for demonstrating Hebbian learning

In Algorithm 13.1 a timer is set to a period such as 100 ms. The timer is decremented by the operating system (not shown) and when it expires, the outputs $y_1$ and $y_2$ are computed by Eq. 13.3 below. These outputs are then used to set the power of the left and right motors, and, finally, the timer is reset.

**Algorithm 13.1: ANN for obstacle avoidance**

```
integer period ← ···        // Timer period (ms)
integer timer ← period
float array[5]      x
float array[2]      y
float array[2,5]    W

1: when timer expires
2:    x ← sensor values
3:    y ← W x
4:    left-motor-power ← y[1]
5:    right-motor-power ← y[2]
6:    timer ← period
```

There are five sensors that are read into the five input variables:

$$
\begin{aligned}
x_1 &\leftarrow \textsf{front left sensor}\\
x_2 &\leftarrow \textsf{front center sensor}\\
x_3 &\leftarrow \textsf{front right sensor}\\
x_4 &\leftarrow \textsf{rear left sensor}\\
x_5 &\leftarrow \textsf{rear right sensor}
\end{aligned}
$$

We assume that the values of the sensors are between 0 (obstacle not detected) and 100 (obstacle very close), and that the values of the motor powers are between $-100$ (full backwards power) and 100 (full forwards power). If the computation results in saturation, the values are truncated to the end points of the range, that is, a value less than $-100$ becomes $-100$ and a value greater than 100 becomes 100.[1] Recall that a robot with differential drive turns right by setting $y_1$ (the power of the left motor) to 100 and $y_2$ (the power of the right motor) to $-100$, and similarly for a left turn.

To simplify the presentation of Algorithm 13.1 we used vector notation where the inputs are given as a single column vector:

$$
\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}.
$$

Referring again to Fig. 13.9, the computation of the outputs is given by:

$$
y_1 \leftarrow w_{1l}x_1 + w_{2l}x_2 + w_{3l}x_3 + w_{4l}x_4 + w_{5l}x_5 \tag{13.1}
$$

$$
y_2 \leftarrow w_{1r}x_1 + w_{2r}x_2 + w_{3r}x_3 + w_{4r}x_4 + w_{5r}x_5 \,. \tag{13.2}
$$

Expressed in vector notation this is:

$$
\mathbf{y} = \begin{bmatrix} y_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{1l} & w_{2l} & w_{3l} & w_{4l} & w_{5l} \\ w_{1r} & w_{2r} & w_{3r} & w_{4r} & w_{5r} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \mathbf{W}\,\mathbf{x}\,. \tag{13.3}
$$

To cause the algorithm to learn, feedback is used to modify the weights $\mathbf{W}$ (Algorithms 13.2, 13.3). Let us assume that there are four buttons on the robot or on a remote control, one for each direction forwards, backwards, left and right. Whenever we note that the robot is in a situation that requires a certain behavior, we touch the corresponding button. For example, if the left sensor detects an obstacle,

[1] Algorithm 13.1 declared all the variables as `float` because the weights are floating point numbers. If the sensor inputs and motor outputs are integers, type conversion will be necessary.

the robot should turn right. To implement this, there is a process for each button. These processes are shown together in Algorithm 13.2, where the forwards slashes / separate corresponding events and actions.

**Algorithm 13.2: Feedback on the robot's behavior**

1: when button {forward / backward / left / right} touched
2: $\quad y_1 \leftarrow \{100 \,/\, -100 \,/\, -100 \,/\, 100\}$
3: $\quad y_2 \leftarrow \{100 \,/\, -100 \,/\, 100 \,/\, -100\}$

The next phase of the algorithm is to update the connection weights according to the Hebbian rule (Algorithm 13.3).

**Algorithm 13.3: Applying the Hebbian rule**

1: $\quad \mathbf{x} \leftarrow$ sensor values
2: $\quad$ for $j$ in $\{1, 2, 3, 4, 5\}$
3: $\quad\quad w_{jl} \leftarrow w_{jl} + \alpha\, y_1\, x_j$
4: $\quad\quad w_{jr} \leftarrow w_{jr} + \alpha\, y_2\, x_j$

**Example** Assume that initially the weights are all zero. By Eqs. 13.1 and 13.2 the outputs are zero and the robot will not move.

Suppose now that an obstacle is placed in front of the left sensor so that $x_1 = 100$ while $x_2 = x_3 = x_4 = x_5 = 0$. Without feedback nothing will happen since the weights are still zero. If we touch the right button (informing the robot that the correct behavior is to turn right), the outputs are set to $y_1 = 100$, $y_2 = -100$ to turn right, which in turn leads to the following changes in the weights (assuming a learning factor $\alpha = 0.0001$):

$$
\begin{aligned}
w_{1l} &\leftarrow 0 + (0.0001 \times 100 \times 100) = 10 \\
w_{1r} &\leftarrow 0 + (0.0001 \times -100 \times 100) = -10\,.
\end{aligned}
$$

The next time that an obstacle is detected by the left sensor, the outputs will be non-zero:

$$
\begin{aligned}
y_1 &\leftarrow (10 \times 100) + 0 + 0 + 0 + 0 = 1000 \\
y_2 &\leftarrow (-10 \times 100) + 0 + 0 + 0 + 0 = -1000\,.
\end{aligned}
$$

After truncating to 100 and $-100$ these outputs will cause the robot to turn right.

The learning factor $\alpha$ determines the magnitude of the effect of $y_k\, x_j$ on the values of $w_{kj}$. Higher values of the factor cause larger effects and hence faster learning. Although one could think that a faster learning is always better, if the learning is too fast it can cause unwanted changes such as forgetting previous good situations or strongly emphasizing mistakes. The learning factor must be adjusted to achieve optimal learning.

**Activity 13.10: Hebbian learning for obstacle avoidance**

- Implement Algorithms 13.1–13.3 and teach your robot to avoid obstacles.
- Modify the program so that it *learns* to move forwards when it does not detect an obstacle.

## 13.6 Summary

Autonomous robots must function in environments characterized by a high degree of uncertainty. For that reason it is difficult to specify precise algorithms for robot behavior. Artificial neural networks can implement the required behavior in an uncertain environment by learning, that is, by modifying and improving the algorithm as the robot encounters additional situations. The structure of ANNs makes learning technically simple: An ANN is composed of a large number of small, simple components called neurons and learning is achieved by modifying the weights assigned to the connections between the neurons.

Learning can be supervised, reinforcement or unsupervised. Reinforcement learning is appropriate for learning robotic behavior because it requires the designer to specify only if an observed behavior is good or bad without quantifying the behavior. The Hebbian rule modifies the weights connecting neurons by multiplying the output of one neuron by the input of the neuron it is connected to. The result is multiplied by a learning factor that determines the size of the change in the weight and thus the rate of learning.

## 13.7 Further Reading

Haykin [1] and Rojas [4] are comprehensive textbooks on neural networks. David Kriesel wrote an online tutorial [3] that can be freely downloaded.

## References

1. Haykin, S.O.: Neural Networks and Learning Machines, 3rd edn. Pearson, Boston (2008)
2. Herculano-Houzel, S.: The human brain in numbers: a linearly scaled-up primate brain. Front. Hum. Neurosci. **3**, 31 (2009)
3. Kriesel, D.: A Brief Introduction to Neural Networks. http://www.dkriesel.com/en/science/neural_networks (2007)
4. Rojas, R.: Neural Networks: A Systematic Introduction. Springer, Berlin (1996)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.