Knowledge-Based Control Systems Summary

1. Introduction to fuzzy sets

In this summary, we will examine various knowledge-based control systems. One type of such systems is based on fuzzy logic. We'll examine the basics of fuzzy logic in this chapter. We'll go more into depth on it in subsequent chapters.

1.1 Basic properties and representations of fuzzy sets

1.1.1 Fuzzy sets

Let's examine ordinary set theory. We have a domain X. Now examine a set A with objects $x_i \in X$. The **membership function** $\mu_A(x)$ is defined as

$$\mu_A(x) = \begin{cases} 1 & \text{iff } x \in A, \\ 0 & \text{iff } x \notin A. \end{cases} \tag{1.1.1}$$

So, an object x is either fully part of A or not at all part of A. We call such a set A a **crisp set**.

However, in **fuzzy logic**, things are different. Now an object x can also be partially in A. In other words, $\mu_A(x)$ can take values between 0 and 1 as well. We call such a set A a **fuzzy set**. Also, the value of $\mu_A(x)$ is called the **membership degree** or **membership grade**.

1.1.2 Properties of fuzzy sets

We can define various properties for fuzzy sets. The height of a fuzzy set $\text{hgt}(A)$ is the supremum (maximum) of the membership grades of A. So,

$$\text{hgt}(A) = \sup_{x \in X} \mu_A(x). \tag{1.1.2}$$

A fuzzy set A is **normal** if $\text{hgt}(A) = 1$. In other words, there is an x for which $\mu_A(x) = 1$. Any set that is not normal is called **subnormal**. Such a set A can be normalized using the normalization function $\text{norm}(A)$. It is defined such that, for all $x \in X$, we have

$$B = \text{norm}(A) \qquad \Rightarrow \qquad \mu_B(x) = \frac{\mu_A(x)}{\text{hgt}(A)}. \tag{1.1.3}$$

The **support** of a set A is the crisp subset of A with nonzero membership grades. Similarly, the **core** of a set A is the crisp subset of A with membership grade equal to one. So,

$$\text{supp}(A) = \{x | \mu_A(x) > 0\} \qquad \text{and} \qquad \text{core}(A) = \{x | \mu_A(x) = 1\}. \tag{1.1.4}$$

The **α-cut** A_α of a set A is the crisp subset of A with membership grades of at least α. So,

$$A_\alpha = \alpha\text{-cut}(A) = \{x | \mu_A(x) \geq \alpha\}. \tag{1.1.5}$$

Note that $\text{core}(A) = 1\text{-cut}(A)$. However, $\text{supp}(A) = 0\text{-cut}(A)$ is not always true.

Let's examine a set A. Its membership function $\mu_A(x)$ is called **unimodal** if it only has one global/local maximum. The corresponding set A is then called **convex**. If however $\mu_A(x)$ is **multimodel** (has several local maxima), then A is **non-convex**. Finally, the **cardinality** $\text{card}(A) = |A|$ of a finite discrete set A is the sum of the membership grades. Thus,

$$\text{card}(A) = |A| = \sum_{i=1}^{n} \mu_A(x_i). \tag{1.1.6}$$

1.1.3 Representations of fuzzy sets

There are several ways to represent fuzzy sets. We will examine a few.

- **Similarity-based representation** – We use a (dis)similarity measure $d(x, v)$ between two elements x and v. An example of a membership function is now given by

$$\mu(x) = \frac{1}{1 + d(x, v)}. \tag{1.1.7}$$

- **Trapezoidal membership function** – We choose parameters a, b, c and d ($a < b$, $c > d$) such that

$$\mu(x) = \max\left(0, \min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right)\right). \tag{1.1.8}$$

  If $b = c$, we obtain the **triangular membership function**.

- **Piece-wise exponential membership function** – We choose the position parameters c_l and c_r ($c_l < c_r$) and the width parameters w_l and w_r ($w_l, w_r > 0$) such that

$$\mu(x) = \begin{cases} \exp\left(-\left(\frac{x-c_l}{2w_l}\right)^2\right) & \text{if } x < c_l, \\ \exp\left(-\left(\frac{x-c_r}{2w_r}\right)^2\right) & \text{if } x > c_r, \\ 1 & \text{otherwise.} \end{cases} \tag{1.1.9}$$

- **Singleton set** – This is a special fuzzy set. For some chosen element x_0, we have

$$\mu(x) = \begin{cases} 1 & \text{if } x = x_0, \\ 0 & \text{otherwise.} \end{cases} \tag{1.1.10}$$

- **Universal set** – This is another special fuzzy set. We simply have $\mu(x) = 1$ for all $x \in X$.

- **Point-wise representation** – For every individual element x, we specify the value of $\mu(x)$. Two different methods of notation are

$$A = \{\mu_A(x_1)/x_1, \mu_A(x_2)/x_2, \ldots, \mu_A(x_n)/x_n\} = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + \ldots + \mu_A(x_n)/x_n. \tag{1.1.11}$$

1.2 Modifying fuzzy sets

1.2.1 Basic operations on fuzzy sets

Let's examine a fuzzy set A. In ordinary set theory, we can do several things with sets. (Think of complements, unions, intersections and such.) We can extend these ideas to fuzzy sets. First, let's examine the **complement** $\bar{A}$ of A. The common definition for $\bar{A}$ is that, for all $x \in X$, we have

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x). \tag{1.2.1}$$

To define the **intersection** $C = A \cap B$ between two sets A and B, we need a **t-norm** $T(a, b)$ such that $\mu_C(x) = T(\mu_A(x), \mu_B(x))$ for all $x \in X$. Such a t-norm must satisfy the following conditions.

$$\begin{aligned} T(a, 1) &= a, && (1.2.2)\\ b \leq c &\Rightarrow T(a, b) \leq T(a, c), && (1.2.3)\\ T(a, b) &= T(b, a), && (1.2.4)\\ T(a, T(b, c)) &= T(T(a, b), c). && (1.2.5) \end{aligned}$$

The most commonly used t-norms are the **standard intersection** (also known as the **minimum**) and the **algebraic product**, which are respectively defined as

$$T(a, b) = \min(a, b) \qquad \text{and} \qquad T(a, b) = ab. \tag{1.2.6}$$

The minimum is the largest possible t-norm.

To define the **union** $C = A \cup B$ between two sets A and B, we need a **t-conorm** $S(a, b)$ such that $\mu_C(x) = S(\mu_A(x), \mu_B(x))$ for all $x \in X$. Such a t-conorm must satisfy the following conditions.

$$\begin{aligned} S(a, 0) &= a, && (1.2.7)\\ b \leq c &\Rightarrow S(a, b) \leq S(a, c), && (1.2.8)\\ S(a, b) &= S(b, a), && (1.2.9)\\ S(a, S(b, c)) &= S(S(a, b), c). && (1.2.10) \end{aligned}$$

The most commonly used t-conorms are the **standard union** (also known as the **maximum**) and the **algebraic sum**, which are respectively defined as

$$S(a, b) = \max(a, b) \qquad \text{and} \qquad S(a, b) = 1 - (1 - a)(1 - b) = a + b - ab. \tag{1.2.11}$$

The maximum is the smallest possible t-norm.

We can also change fuzzy sets by using **hedges**. Let's suppose that the fuzzy set A indicates expensive cars. If some element x (say, x = 10,000 euros) has a low membership degree, it is not expensive. But if its membership degree is high, it is expensive. How can we find the set B that indicates very expensive cars or the set C that indicates mildly expensive cars? There are two methods. We can use **shifted hedges**: we shift the membership function along the domain. So, $\mu_B(x) = \mu_A(x - 5,000)$ and $\mu_C(x) = \mu_A(x + 3000)$. We can also use **powered hedges**: $\mu_B(x) = \mu_A(x)^2$ and $\mu_C(x) = \sqrt{\mu_A(x)}$.

1.2.2 Modifications of fuzzy sets

Let's examine some domain X and another domain Y. We can define a fuzzy set A in X or in Y, but we can also define it in $X \times Y$. We then have to define $\mu_A(x, y)$ for every combination $x \in X, y \in Y$. The same can be done for higher-dimensional spaces. Such spaces are known as **Cartesian product spaces**.

Let's examine some Cartesian product spaces U, U_1 and U_2 with $U_1 \subseteq U \subseteq U_1 \times U_2$. (In other words, U_1 and U_2 together encompass U, which in turn encompasses U_1.) We also have some set A defined in U. We can now find the **projection** of A onto U_1 using

$$\text{proj}_{U_1}(A) = \left\{ \sup_{U_2} \mu_A(u)/u_1 | u_1 \in U_1 \right\}. \tag{1.2.12}$$

In other words, for each set of parameters of U_1, we browse through all combinations of the parameters of U_2 and look for the one with the highest value of $\mu_A(u)/u_1$.

Again, examine Cartesian product spaces U, U_1 and U_2 with $U_1 \subseteq U \subseteq U_1 \times U_2$. But now, we have a set A defined on U. We can find the **cylindrical extension** to U_1 using

$$\text{ext}_U(A) = \{\mu_A(u_1)/u | u \in U\}. \tag{1.2.13}$$

It is important to note that, with a projection, you go to a lower-dimensional space. This generally results in a loss of data. However, with a cylindrical extension, you go to a higher-dimensional space. You now do not lose data.

Let's examine two fuzzy sets A_1 and A_2, defined on domains X_1 and X_2, respectively. We would like to take the intersection between the two sets. But, because the two sets are defined on different domains, we can't do this using the normal definition. As a solution, we use cylindrical extensions. So,

$$A_1 \times A_2 = \text{ext}_{X_2}(A_1) \cap \text{ext}_{X_1}(A_2) \qquad \Rightarrow \qquad \mu_{A_1 \times A_2}(x_1, x_2) = T(\mu_{A_1}(x_1), \mu_{A_2}(x_2)). \tag{1.2.14}$$

1.2.3 Fuzzy relations

A **fuzzy relation** R is a fuzzy set in the Cartesian product space $X_1 \times X_2 \times \ldots \times X_n$. This fuzzy set has a membership function $\mu_R(x_1, x_2, \ldots, x_n)$ which gives a value between 0 and 1 (inclusive) for all combinations of parameters $x_1, x_2, \ldots, x_n$.

Now let's examine a fuzzy relation R in $X \times Y$ and a fuzzy set A in X. We can find a fuzzy set B in Y through the **composition** of A and R:

$$B = A \circ R = \text{proj}_Y \left(R \cap \text{ext}_{X \times Y}(A) \right). \tag{1.2.15}$$

We thus extend A to $X \times Y$, intersect it with R, and then project the result on Y. It can be shown that the membership function of B now satisfies

$$\mu_B(y) = \max_x \min(\mu_A(x), \mu_R(x, y)). \tag{1.2.16}$$

2. Fuzzy models

We can use fuzzy logic to build fuzzy models. In this chapter, we examine how this works.

2.1 Types of fuzzy models

A static/dynamic systems which makes use of fuzzy sets is called a **fuzzy system**. Most common are fuzzy systems defined by if-then rules. These are called **rule-based systems**, also known as **fuzzy models**. An if-then rule generally takes the form of

$$\textbf{If } \textit{antecedent proposition} \textbf{ then } \textit{consequent proposition}. \tag{2.1.1}$$

The antecedent proposition is always a fuzzy proposition of the type '$\mathbf{x}$ is A', where $\mathbf{x}$ is a **linguistic variable** and A is a **linguistic constant**. (For example, it can be 'if Temperature is high then ....) The structure of the consequent proposition, however, depends on the model we use.

- In a **linguistic fuzzy model**, both the antecedent and the consequent are fuzzy propositions.
- The **fuzzy relational model** is an extension of the linguistic fuzzy model. Now, a fuzzy antecedent can be coupled to multiple fuzzy propositions at the same time.
- In the **Takagi-Sugeno (TS) fuzzy model**, the consequent is a crisp function of the antecedent variables.

2.2 The linguistic fuzzy model

2.2.1 Properties of the linguistic model

As we just saw, in a linguistic fuzzy model, relations take the form of

$$\mathcal{R}_i : \textbf{if } \mathbf{x} \text{ is } A_i \textbf{ then } \mathbf{y} \text{ is } B_i. \tag{2.2.1}$$

A **linguistic variable** L (for example 'Temperature') is defined as a set $L = (\mathbf{x}, \mathcal{A}, X, g, m)$. Here, $\mathbf{x}$ is the **base variable**, having the same name as the linguistic variable. $\mathcal{A}$ is the **set of linguistic terms** (for example 'cold', 'normal' and 'warm'). X is the **domain** of x (for example, $[-273, \infty)$). Finally, g is a **syntactic rule** for generating linguistic terms and m is a **semantic rule** that assigns to each linguistic term its meaning. The latter two are in a way just formalities: we won't consider them here.

It is often required that a linguistic term satisfies properties of coverage and semantic soundness. **Coverage** means that each domain element $\mathbf{x} \in X$ is assigned to at least one fuzzy set A_i. (For example, there isn't a single temperature which is not either 'cold', 'normal' or 'warm'.) A stronger requirement is ϵ-coverage. This demands that each domain element $\mathbf{x} \in X$ is at least assigned to one fuzzy set A_i with $\mu_{A_i}(\mathbf{x}) > \epsilon$. Next to this, **semantic soundness** relates to how well a system can distinguish between different variables $\mathbf{x}$. (For example, if a system can't find the difference between a low temperature of 0° C and a low temperature of 5° C, then it is doesn't have a lot of semantic soundness.)

2.2.2 Inference in the linguistic model

Inference in fuzzy rule-based systems is the process of deriving a fuzzy output set given the rules and the inputs. Each rule $\mathcal{R}_i$ can be seen as a fuzzy relation $R : (X \times Y) \to [0, 1]$ such that

$$\mu_r(\mathbf{x}, \mathbf{y}) = I(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})). \tag{2.2.2}$$

The I operator can be either a fuzzy implication or a conjuction operator (t-norm). Fuzzy implication is used when the rule has the form 'A implies B'. Examples of fuzzy implications are the **Lukasiewicz implication** and the **Kleene-Diene implication**, respectively defined as

$$I(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})) = \min(1, 1 - \mu_A(\mathbf{x}) + \mu_B(\mathbf{y})) \qquad \text{and} \qquad I(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})) = \max(1 - \mu_A(\mathbf{x}), \mu_B(\mathbf{y})). \tag{2.2.3}$$

Alternatively, conjunction is used is when $A \wedge B$. That is, when A and B simultaneously hold. Examples of t-norms are the minimum (also often referred to as the **Mamdani 'implication'** and the **Larsen 'implication'**, respectively defined as

$$I(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})) = \min(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})) \qquad \text{and} \qquad I(\mu_A(\mathbf{x}), \mu_B(\mathbf{y})) = \mu_A(\mathbf{x}) \cdot \mu_B(\mathbf{y}). \tag{2.2.4}$$

So how do we use this? Well, let's suppose we have a rule **if** $\mathbf{x}$ is A_i **then** $\mathbf{y}$ is B_i and we also know that $\mathbf{x}$ is A', then we can find the set B' satisfying $\mathbf{y}$ is B' using

$$B' = A' \circ R. \tag{2.2.5}$$

The question remains, what do we do if we have multiple rules/relations R_i? In that case we have to join them somehow to some joined relation R. When dealing with implications, we do this using an intersection, like

$$R = \bigcap_{i=1}^{K} R_i \qquad \text{meaning that} \qquad \mu_R(\mathbf{x}, \mathbf{y}) = \min_{1 \leq i \leq K} \mu_{R_i}(\mathbf{x}, \mathbf{y}). \tag{2.2.6}$$

If, however, we are dealing with conjunction, then the aggregated relation R is the union of the individual relations R_i. So,

$$R = \bigcup_{i=1}^{K} R_i \qquad \text{meaning that} \qquad \mu_R(\mathbf{x}, \mathbf{y}) = \max_{1 \leq i \leq K} \mu_{R_i}(\mathbf{x}, \mathbf{y}). \tag{2.2.7}$$

Again, the output set B' is found in the same way, by using $B' = A' \circ R$.

2.2.3 Max-min Mamdani inference

In the previous method of inference, we had to use relations R. When the domains X and Y get very big, this will become rather complicated. But luckily, the relational calculus can be bypassed by using **max-min (Mamdani) inference**. In Mamdany inference, we can find the output using

$$\mu_{B'}(\mathbf{y}) = \max_{1 \leq i \leq K} \left(\max_X \left(\mu_{A'}(\mathbf{x}) \wedge \mu_{A_i}(\mathbf{x}) \right) \wedge \mu_{B_i}(\mathbf{y}) \right) = \max_{1 \leq i \leq K} \left(\beta_i \wedge \mu_{B_i}(\mathbf{y}) \right). \tag{2.2.8}$$

In this equation, we have defined the **degree of fulfillment** β_i as

$$\beta_i = \max_X \left(\mu_{A'}(\mathbf{x}) \wedge \mu_{A_i}(\mathbf{x}) \right). \tag{2.2.9}$$

Basically, this number is an indication of how much A' and A_i are alike. A big advantage of the Mamdani method is that it does not require discretization of the domain. It can thus work with analytically defined membership functions.

2.2.4 Defuzzification

We now know how to find an output fuzzy set B', based on fuzzy rules. But usually, we don't want to know that some parameter $\mathbf{y}$ belongs to a fuzzy set B'. Instead, we want to know a value $\mathbf{y}'$. The process of finding a value $\mathbf{y}'$ from the knowledge that $\mathbf{y}$ is B' is called **defuzzification**.

There are two commonly used defuzzification methods: the center of gravity method and the mean of maxima method. In the **center of gravity (COG) method**, we calculate the **y**-coordinate of the center of gravity of the fuzzy set B'. This is done according to

$$\mathbf{y}' = \text{cog}(B') = \frac{\sum_{j=1}^{F} \mu_{B'}(y_j) y_j}{\sum_{j=1}^{F} \mu_{B'}(y_j)} = \frac{\int_Y \mu_{B'}(y) y \, dy}{\int_Y \mu_{B'}(y)}. \tag{2.2.10}$$

The first part of the above equation is used for discretized domains Y, whereas the second part is used for continuous domains Y.

In the **mean of maxima (MOM) method**, we find all points where $\mu_{B'}(\mathbf{y})$ is at its maximum. We then take the mean of all these points. In mathematical notation, we then have

$$\mathbf{y}' = \text{mom}(B') = \text{cog}\left(\left\{\mathbf{y} | \mu_{B'}(\mathbf{y}) = \max_{\mathbf{y} \in Y} \mu_{B'}(\mathbf{y})\right\}\right). \tag{2.2.11}$$

In a way, the MOM method selects the 'most probable' output. It is often used with inference based on fuzzy implications. On the other hand, the COG method is usually used together with Mamdani inference.

Finally, there is also a third defuzzification, called **fuzzy-mean defuzzification**. It is often used after Mamdani inference, to avoid the integration step from the COG method. When applying this method, first the **consequent fuzzy sets** B_j are found, using

$$\mu_{B_j}(\mathbf{y}) = (\beta_i \wedge \mu_{B_i}(\mathbf{y})). \tag{2.2.12}$$

Instead of first using $\mu_{B'}(\mathbf{y}) = \max \mu_{B_j}(\mathbf{y})$ and then applying defuzzification, we now first apply defuzzification using $b_j = \text{mom}(B_j)$. Now, a crisp output $\mathbf{y}'$ is obtained by taking the weighted average of b_j. So,

$$\mathbf{y}' = \frac{\sum_{j=1}^{M} \omega_j b_j}{\sum_{j=1}^{M} \omega_j}, \qquad \text{where} \qquad \omega_j = \mu_{B'}(b_j). \tag{2.2.13}$$

The weight ω_j is thus the maximum of the degrees of fulfilment β_i over all rules $\mathcal{R}_i$ with consequent B_j. In this way, an integration over the domain is avoided.

2.2.5 Rules with several inputs

Previously, we have considered multivariate membership functions $\mu_A(\mathbf{x})$. Sometimes, it may be convenient to use univariante membership functions $\mu_A(x)$. But what do we do then if we still have multiple variables $x_1, \ldots, x_n$? In this case, we simply add them together in the antecedent. This gives us the **conjunctive form** of the antecedant:

$$\mathcal{R}_i : \textbf{if } x_1 \text{ is } A_{i1} \textbf{ and } \ldots \textbf{ and } x_{in} \text{ is } A_{in} \textbf{ then } \mathbf{y} \text{ is } B_i. \tag{2.2.14}$$

This is, in fact, a special case of our previous multivariate rules. In fact, if we use $A_i = A_{i1} \times \ldots \times A_{in}$, and insert this into the normal multivariate rule, then we get exactly the same result. As such, the possibilities of the conjunctive form are limited. Also, it is often necessary to define a lot of rules. For every combination of $x_1, \ldots, x_n$, a rule is necessary.

One way in which the number of rules can be reduced, is by using additional **logical connectives**, like 'or' and 'not'. In this way, a rule can be defined like

$$\mathcal{R}_i : \textbf{if } x_1 \text{ is } \textbf{not } A_{i1} \textbf{ or } x_{i2} \text{ is } A_{i2} \textbf{ then } \mathbf{y} \text{ is } B_i. \tag{2.2.15}$$

In this way, less rules are required than in the conjunctive form. Yet still, this method allows for fewer possibilities than the multivariate rule form. So, the multivariate rule form is the most general form you can use.

2.3 Other kinds of fuzzy models

2.3.1 The singleton model

A special case of the linguistic fuzzy model is the **singleton model**. It is obtained when the consequence fuzzy sets B_i are singleton sets. In this case, we can write the rules as

$$\mathcal{R}_i : \textbf{if } \mathbf{x} \text{ is } A_i \textbf{ then } y \text{ is } b_i. \tag{2.3.1}$$

For the singleton method, defuzzification simply means applying the fuzzy-mean method. So we have

$$y = \frac{\sum_{i=1}^{K} \beta_i b_i}{\sum_{i=1}^{K} \beta_i}. \tag{2.3.2}$$

We can also generalize the singleton model to a class of functions called the **basis functions expansion**. We now have

$$y = \sum_{i=1}^{K} \phi_i(\mathbf{x}) b_i. \tag{2.3.3}$$

So, for the singleton model, $\phi_i(\mathbf{x})$ is simply the normalized degree of fulfillment of the rule antecedents.

2.3.2 The fuzzy relational model

The **fuzzy relational model** is an expansion of the linguistic model. In the linguistic model, y always belonged to a certain linguistic term. (e.g. we had y is Fast.) In the relational model, y can also partly belong to multiple linguistic terms. So, an example of a rule might be

$$\textbf{if } x_1 \text{ is } \textit{Low} \textbf{ and } x_2 \text{ is } \textit{High} \textbf{ then } y \text{ is } \textit{Cold}\ (0.9), y \text{ is } \textit{Normal}\ (0.2), y \text{ is } \textit{Warm}\ (0.0). \tag{2.3.4}$$

Let's denote the set of linguistic terms of antecedent variable x_j by $\mathcal{A}_j$. The set of all combinations of linguistic variables $x_1, \ldots, x_n$ is now denoted by $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$. Similarly, we can denote the set of linguistic terms of consequent variable y by $\mathcal{B}$. The fuzzy relational model can now be seen as a fuzzy relation

$$R : \mathcal{A} \times \mathcal{B} \to [0, 1]. \tag{2.3.5}$$

The relation R can be represented by a matrix. The elements r_{ij} of this matrix now equal the numbers denoted in parantheses in the rules.

It is important to note the difference between the matrix R for the linguistic model and the matrix R for the relational model. In the linguistic model, R denoted the degree of association between elements from X and Y (that is, from the input and the output space). However, in the relational model R denotes the association between the linguistic terms of the input and the output.

So how does inference work in the relational model? Well, we first compute the degree of fulfillment of the rules. This still goes according to

$$\beta_i = \mu_{A_{i1}}(x_1) \wedge \ldots \wedge \mu_{A_{in}}(x_n). \tag{2.3.6}$$

Now, we can find the degree ω_j to which y belongs to class B_j. This is done using $\omega = \beta \circ R$ or, equivalently,

$$\omega_j = \max_{1 \leq i \leq K} \left(\beta_i \wedge r_{ij} \right). \tag{2.3.7}$$

Finally, we need to find the defuzzified output y. For that, we simply take the weighted mean of the classes B_j. So,

$$y = \frac{\sum_{j=1}^{M} \omega_j b_j}{\sum_{j=1}^{M} \omega_j}. \tag{2.3.8}$$

Here, $b_j = \text{cog}(B_j)$ is the centroid of B_j.

The main advantage of the relational model is that the input-output model can be fine-tuned without changing the consequent fuzzy sets. Instead, you can simply adjust the values of r_{ij} in the rules of the fuzzy system.

2.3.3 The Takagi-Sugeno model

The **Takagi-Sugeno (TS) model** uses crisp functions as consequents. Basically, a rule has the form

$$\mathcal{R}_i : \textbf{if } \mathbf{x} \text{ is } A_i \textbf{ then } \mathbf{y} = \mathbf{f_i}(\mathbf{x}). \tag{2.3.9}$$

If the function $\mathbf{f_i}(\mathbf{x})$ has an affine form (so $\mathbf{f_i}(\mathbf{x}) = \mathbf{a_i}^T\mathbf{x} + \mathbf{b_i}$), then the model is called an **affine TS model**. To apply inference with the Takagi-Sugeno model, we simply use the fulfillment degrees. So,

$$\mathbf{y} = \frac{\sum_{i=1}^K \beta_i \mathbf{y_i}}{\sum_{i=1}^K \beta_i} = \frac{\sum_{i=1}^K \beta_i \mathbf{f_i}(\mathbf{x})}{\sum_{i=1}^K \beta_i} = \frac{\sum_{i=1}^K \beta_i \left(\mathbf{a_i}^T\mathbf{x} + \mathbf{b_i}\right)}{\sum_{i=1}^K \beta_i}. \tag{2.3.10}$$

2.3.4 Dynamic fuzzy systems

Let's examine a time-invariant system. We can model such a system using

$$\mathbf{x}(k+1) = \mathbf{f}(\mathbf{x}(k), \mathbf{u}(k)), \tag{2.3.11}$$

where $\mathbf{x}(k)$ is the **state**, $\mathbf{u}(k)$ is the **input** and $\mathbf{f}$ is the **state transition function**. We can use a fuzzy model to approximate $\mathbf{f}$. However, it is usually hard to do this, since we can't always measure the state $\mathbf{x}$. So instead, we usually use a fuzzy model to approximate the **output y** of the system. In the **dynamic TS model** this is done according to rules of the form

$$\textbf{if } \mathbf{y}(k) \text{ is } A_{i1} \textbf{ and } \mathbf{y}(k-1) \text{ is } A_{i2} \textbf{ and } \ldots \textbf{ and } \mathbf{y}(k-n_y+1) \text{ is } A_{in_y} \tag{2.3.12}$$

$$\textbf{and } \mathbf{u}(k) \text{ is } B_{i1} \textbf{ and } \mathbf{u}(k-1) \text{ is } B_{i2} \textbf{ and } \ldots \textbf{ and } \mathbf{u}(k-n_u+1) \text{ is } B_{in_u} \tag{2.3.13}$$

$$\textbf{then } \mathbf{y}(k+1) = \sum_{j=1}^{n_y} a_{ij} y(k-j+1) + \sum_{j=1}^{n_u} b_{ij} u(k-j+1) + c_i. \tag{2.3.14}$$

The values of n_u and n_y (i.e. how far we look back in time for the input/output) depend on the order of the dynamic system.

3. Fuzzy clustering

Let's suppose that we have a lot of object, and we've made some measurements of these objects. Can we now divide these objects into groups called **clusters**? And if so, how do we do this using fuzzy logic? That is what this chapter is about.

3.1 Types of clustering

3.1.1 The data set

Let's suppose that we have N objects (e.g. pieces of fruit). Of each of these objects, we make n **measurements** (e.g. size, weight, etcetera). These measurements are also called **features** or **attributes**. The set of measurements of one object, $\mathbf{z_k} = [z_{1k}, \ldots, z_{nk}]^T$, is called a **sample**, a **pattern** or simply an **object**. We can also put all measurements in a matrix. We then get the **data matrix** $\mathbf{Z} = [\mathbf{z_1} \ \ldots \ \mathbf{z_N}]$.

To divide objects into clusters, we often make use of **(dis)similarity measures**. One well-known example of a dissimilarity measure is the **Euclidian distance** $||\mathbf{z_j} - \mathbf{z_i}||$, but we'll consider more later. Based on the similarity measures, objects are divided into clusters. How exactly this can be done will be discussed later in this chapter.

3.1.2 Hard clustering

There is an important distinction between hard clustering and fuzzy clustering. In **hard clustering** we make a **hard partition** of the data set $\mathbf{Z}$. In other words, we divide them into $c \geq 2$ clusters (with c assumed known). With a partition, we mean that

$$\bigcup_{i=1}^{c} A_i = \mathbf{Z} \qquad \text{and} \qquad A_i \cap A_j = \emptyset \quad \text{for all } i \neq j. \tag{3.1.1}$$

Also, none of the sets A_i may be empty.

To indicate a partitioning, we make use of **membership functions** μ_{ik}. If $\mu_{ik} = 1$, then object i is in cluster k. Alternatively, if $\mu_{ik} = 0$, then object i is not in cluster k. Based on the membership functions, we can assemble the **partition matrix** $\mathbf{U}$, of which μ_{ik} are the elements. Finally, there is the rule that

$$\sum_{i=1}^{c} \mu_{ik} = 1. \tag{3.1.2}$$

In other words, every object is only part of one cluster. Thus, every column of $\mathbf{U}$ has only a single 1. The set of all hard clusterings $\mathbf{U}$ that can be obtained with hard clustering is now denoted as M_{hc}.

3.1.3 Fuzzy clustering

Hard clustering has a downside. When an object roughly falls between two clusters A_i and A_j, it has to be put into one of these clusters. Also, outliers have to be put in some cluster. This is undesirable. But it can be fixed by fuzzy clustering.

In **fuzzy clustering**, we make a **fuzzy partition** of the data. Now, the membership function μ_{ik} can be any value between 0 and 1. This means that an object $\mathbf{z_k}$ can be for 0.2 part in A_i and for 0.8 part in A_j. However, requirement (3.1.2) still applies. So, the sum of the membership functions still has to be 1. The set of all fuzzy partitions that can be formed in this way is denoted by M_{fc}.

Fuzzy partitioning again has a downside. When we have an **outlier** in the data (being an object that doesn't really belong to any cluster), we still have to assign it to clusters. That is, the sum of its membership functions still must equal one. In **possibilistic partitioning**, this requirement (3.1.2) is relaxed. Instead, it is only required that for every object we have $\mu_{ik} > 0$ for some cluster A_k. The set of all possibilistic partitions that can be formed in this way is denoted by M_{pc}.

3.2 The fuzzy c-means clustering method

3.2.1 The goal of the fuzzy c-means clustering method

Given a data set $\mathbf{Z}$, how do we find a good fuzzy clustering $\mathbf{U} \in M_{fc}$? For that, we have to use a clustering algorithm. One of the most-used algorithms is **fuzzy c-means clustering** which we will examine in this part. In the fuzzy c-means clustering method, we try to minimize the cost function called the **fuzzy c-means functional**, being

$$J(\mathbf{U}, \mathbf{V}|\mathbf{Z}) = \sum_{i=1}^{c} \sum_{k=1}^{N} (\mu_{ik})^m ||\mathbf{z_k} - \mathbf{v_i}||_{\mathbf{A}}^2. \tag{3.2.1}$$

In this equation, $\mathbf{V} = [\mathbf{v_1}, \ldots, \mathbf{v_c}]$ is a vector of **cluster prototypes** (centers) and m is a constant. Also, we have

$$D_{ik\mathbf{A}}^2 = ||\mathbf{z_k} - \mathbf{v_i}||_{\mathbf{A}}^2 = (\mathbf{z_k} - \mathbf{v_i})^T \mathbf{A} (\mathbf{z_k} - \mathbf{v_i}). \tag{3.2.2}$$

3.2.2 The fuzzy c-means clustering algorithm

So how do we minimize the cost function? Well, we start by taking a random partition matrix $\mathbf{U^{(0)}} \in M_{fc}$. We then continue doing the following steps.

1. We compute the weighted means of the clusters using

$$\mathbf{v_i}^{(l)} = \frac{\sum_{k=1}^{N} \left(\mu_{ik}^{(l-1)}\right)^m \mathbf{z_k}}{\sum_{k=1}^{N} \left(\mu_{ik}^{(l-1)}\right)^m}. \tag{3.2.3}$$

2. We compute the distances $D_{ik\mathbf{A}}^2$ using $D_{ik\mathbf{A}}^2 = ||\mathbf{z_k} - \mathbf{v_i}^{(l)}||_{\mathbf{A}}^2$.

3. We update the partition matrix $\mathbf{U}$. For all objects k, we define the new measurement functions $\mu_{ik}^{(l)}$ as

$$\mu_{ik}^{(l)} = \frac{1}{\sum_{j=1}^{c} \left(\frac{D_{ik\mathbf{A}}}{D_{jk\mathbf{A}}}\right)^{\frac{2}{m-1}}}. \tag{3.2.4}$$

   However, a problem occurs if $D_{ik\mathbf{A}}^2 = 0$. (This can occur if $\mathbf{z_k} = \mathbf{v_i^{(l)}}$ for some k, i or if $\mathbf{A}$ is a singular matrix.) Let's suppose that there are q clusters A_i for which $D_{ik\mathbf{A}}^2 = 0$. We then simply give all these clusters a membership degree of $\mu_{ik}^{(l)} = 1/q$. All the other clusters (with $D_{ik\mathbf{A}}^2 > 0$) get a membership function of $\mu_{ik}^{(l)} = 0$.

We repeat the above iteration until the partition matrix $\mathbf{U}$ doesn't really change anymore. That is, until $||\mathbf{U}^{(l)} - \mathbf{U}^{(l-1)}|| < \epsilon$ for some norm $||.||$ and for some defined ϵ. (Often $\epsilon = 0.01$ or $\epsilon = 0.001$ works well enough, depending on the trade-off between run-time and accuracy.)

3.2.3 Properties of the fuzzy c-means clustering method

There are several important things to know about fuzzy c-means clustering. First of all, it converges to a local minimum. (This depends on the initialization of $\mathbf{U}$.) So, to make sure that a good clustering is obtained, the algorithm needs to be run several times for different initializations $\mathbf{U}$.

It is also important to set the parameters of the algorithm right. The most important one is the number of clusters c. Sometimes, this is obvious. But often it is not. To test whether a clustering has the right number of clusters, you can look at a **validity measure** like the **Xie-Beni index**

$$\xi(\mathbf{U}, \mathbf{V}|\mathbf{Z}) = \frac{\sum_{i=1}^{c} \sum_{k=1}^{N} (\mu_{ik})^m ||\mathbf{z_k} - \mathbf{v_i}||^2}{c \cdot \min_{i \neq j} (||\mathbf{v_i} - \mathbf{v_j}||)}. \tag{3.2.5}$$

The upper side of the fraction can be seen as the 'average distance within the cluster', while the bottom side is an indication of the 'distance between clusters'. A small index is positive. So, if we simply run the algorithm for different numbers of clusters c, then we can select the solution with the smallest index.

Another important parameter is the **fuzziness parameter** m. If $m = 1$, then we wind up with a hard clustering. However, if $m \to \infty$, we wind up with a very fuzzy clustering $\mu_{ik} = \frac{1}{c}$ for all i, k. Usually, $m = 2$ offers a good compromise, though this number can be varied during subsequent runs of the algorithm.

The **norm-inducing matrix** mainly determines the shapes of the clusters. If $\mathbf{A} = \mathbf{I}$, then we are using a **Euclidian norm**. The shape of the clusters will be circular. Alternatives are the **diagonal norm** and the **Mahalanobis norm**, which respectively use

$$\mathbf{A} = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2^2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_n^2} \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \mathbf{R}^{-1} = \left(\frac{1}{N} \sum_{k=1}^{N} (\mathbf{z_k} - \bar{\mathbf{z}})(\mathbf{z_k} - \bar{\mathbf{z}})^T \right)^{-1}. \tag{3.2.6}$$

Here, the parameters σ_i^2 are the variances of the matrix $\mathbf{Z}$ in direction i. Both the diagonal norm and the Mahalanobis norm will result in clusters with ellipsoidal shapes. However, the fundamental difference is that, with the diagonal norm, the axes of the ellipses are aligned with the main axes. When using the Mahalanobis norm, the axes are arbitrary.

3.2.4 An extension of the fuzzy c-means method

The downside with using a single matrix $\mathbf{A}$ is that all clusters will have the same shape and orientation. When there are clusters with different shapes, this will be undesirable. The **Gustafson-Kessel algorithm** is an extension of the fuzzy c-means method which gets rid of this downside.

The main idea is that, instead of using the same matrix $\mathbf{A}$ all the time, we use a different matrix $\mathbf{A_i}$ to calculate the norm $D^2_{ik\mathbf{A_i}}$. Now, also an optimum for the matrix $\mathbf{A_i}$ needs to be found. This does give rise to a problem though. If we minimize the cost function J right away, then the matrices $\mathbf{A_i}$ simply go to zero. We thus need to constrain them in some way. Usually, $\det(A_i) = |A_i| = \rho_i$ is used as a constraint. Here the choice of values for ρ_i depends on earlier experience. If this experience is lacking, $\rho_i = 1$ is simply chosen.

So how can this be implemented in the algorithm? Well, we simply replace step 2 by the following. We first calculate the **fuzzy covariance matrix** $\mathbf{F_i}$ for all clusters i using

$$\mathbf{F_i} = \frac{\sum_{k=1}^{N} \left(\mu_{ik}^{(l-1)}\right)^m (\mathbf{z_k} - \mathbf{v_i})(\mathbf{z_k} - \mathbf{v_i})^T}{\sum_{k=1}^{N} \left(\mu_{ik}^{(l-1)}\right)^m}. \tag{3.2.7}$$

Now, the improved value of $\mathbf{A_i}$ can be found using

$$\mathbf{A_i} = (\rho_i \det(\mathbf{F_i}))^{\frac{1}{n}} \mathbf{F_i}^{-1}. \tag{3.2.8}$$

This matrix $\mathbf{A_i}$ is then used to compute the distance norm $D^2_{ik\mathbf{A_i}}$ after which the algorithm proceeds as normal.

The matrices $\mathbf{F_i}$ are quite important. In fact, the shapes of the clusters depend on them. The main axes of the ellipses are denoted by the eigenvectors $\phi_{\mathbf{ij}}$ of $\mathbf{F_i}$. The sizes of the ellipses in these directions are proportional to $\sqrt{\lambda_{ij}}$, with λ_{ij} the eigenvalue corresponding to the eigenvector $\phi_{\mathbf{ij}}$.

4. Building fuzzy systems and controllers

A **fuzzy system** is a system that makes use of fuzzy logic. In this chapter, we're going to examine how to build such a system. We also look at the working principles of fuzzy controllers.

4.1 Construction of fuzzy systems

4.1.1 Basic fuzzy system construction

When building a fuzzy system, the first thing that needs to be defined is the **structure** of the system. This structure consists of the following parts.

- The number and type of **input and output variables**.
- The **structure of the rules**. That is, the kind of fuzzy model that is used.
- The number of **linguistic variables** and the number and type of **membership functions** for each variable. (E.g. whether triangular or trapezoidal membership functions are used.)
- The type of **inference mechanism** that is used.
- The **defuzzification method**.

Once the structure has been set up, the available knowledge should be formulated as a set of 'if-then' rules. This then results in a fuzzy system. But it's not done yet. Based on available test data, the parameters of the system can be fine-tuned. (With parameters, we mainly mean the parameters of the membership functions, like the position of the top at the triangular membership function and such. But in some models, the if-then rules also have parameters.) Finally, the system needs to be evaluated. If it does not meet the expectations, then a new system should be created with a somewhat different structure.

4.1.2 The least-squares estimation of consequents

Let's suppose that we are using an affine Takani-Sugeno model. So we have a set of K rules with consequents of the form $y = \mathbf{a_i}^T\mathbf{x} + \mathbf{b_i}$. We need to set these parameters $\mathbf{a_i}$ and b_i. This can be done by using a set of N input-output data pairs $(\mathbf{x_i}, y_i)$. We can put these pairs into an input matrix $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x_N}]^T$ and an output vector $\mathbf{y} = [y_1, \ldots, y_N]^T$. Now, because the models are linear, we can use the least-squares method to find optimal values for $\mathbf{a_i}$ and b_i.

First, let's define the matrix $\mathbf{\Gamma_i}$ as the $N \times N$ diagonal matrix having the normalized membership degrees $\gamma_i(\mathbf{x_j})$ on its diagonal. We also append the matrix $\mathbf{X}$ with a column of ones to get $\mathbf{X_e} = [\mathbf{X}\ \mathbf{1}]$. Now, define the matrix $\mathbf{X'}$ as

$$\mathbf{X'} = \begin{bmatrix} \mathbf{\Gamma_1 X_e} & \mathbf{\Gamma_2 X_e} & \ldots & \mathbf{\Gamma_K X_e} \end{bmatrix}. \tag{4.1.1}$$

The consequent parameters, which need to be found, are put into one big vector, being

$$\theta = \begin{bmatrix} \mathbf{a_1}^T & b_1 & \mathbf{a_2}^T & b_2 & \ldots & \mathbf{a_K}^T & b_K \end{bmatrix}. \tag{4.1.2}$$

The least-squares equation which we want to solve is $\mathbf{y} \approx \mathbf{X'}\theta$. The solution for θ is thus given by

$$\theta = \left((\mathbf{X'})^T\mathbf{X'}\right)^{-1}(\mathbf{X'})^T\mathbf{y}. \tag{4.1.3}$$

This trick can also be used for the singleton model. But now we simply omit $\mathbf{a_i}$ and set $\mathbf{X_e} = \mathbf{1}$.

4.1.3 Defining linguistic terms and membership functions

It is often difficult to choose which linguistic terms to use and which membership functions to give them. In **template-based modeling**, we use a simple technique. We simply define a set of K linguistic terms A_i. The membership functions of A_i are now distributed such that the whole interval of possible inputs x is covered. Every membership function A_i then gets its own if-then rule.

The question remains how to distribute the membership functions. If no knowledge on the system is present, the functions are distributed evenly over the interval. The big downside of the template-based modeling now is that the number K of linguistic terms may grow very fast.

Another way to define rules is by using **fuzzy clustering**. We simply take N samples of data and divide them over K fuzzy sets A_i using a fuzzy clustering technique. Now we can define K rules; one for each fuzzy cluster A_i. When also the Takagi-Sugeno model is applied, we will get rules like

$$\textbf{if } x \text{ is } A_i \textbf{ then } y = a_i x + b_i. \tag{4.1.4}$$

4.1.4 Applying fuzzy system construction to systems

Let's suppose that we have some system in which the output $y(t+1)$ can be modeled as

$$y(t+1) = f(y(t), y(t-1), u(t), u(t-1)). \tag{4.1.5}$$

Such a model is called a **second-order NARX model**. In the above equation, $y(t+1)$ is called the **regressand**, while $y(t)$, $y(t-1)$, $u(t)$ and $u(t-1)$ are the **regressors**. We can define the **regressor matrix X** and the **regressand vector y** up to time N_d as

$$\mathbf{X} = \begin{bmatrix} y(2) & y(1) & u(2) & u(1) \\ y(3) & y(2) & u(3) & u(2) \\ \vdots & \vdots & \vdots & \vdots \\ y(N_d-1) & y(N_d-2) & u(N_d-1) & u(N_d-2) \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} y(3) \\ y(4) \\ \vdots \\ y(N_d) \end{bmatrix}. \tag{4.1.6}$$

Now we can construct our fuzzy system in one of the normal ways. So we see that fuzzy systems can also be used to approximate system dynamics.

Sometimes we can help the fuzzy system a bit. Let's suppose that we have some system $y = f(x)$. We also know, thanks to physical insights, that the output y is roughly proportional to x^2. In this case, it might be better to use x^2 as input instead of x. (Fuzzy systems are better at approximating linear functions than they are at approximating quadratic functions.) So, by simply changing the input of the system, the performance will most likely increase. This trick of using physical insights to give the system an easier job is called **semi-mechanistic modeling**.

4.2 Construction of fuzzy controllers

4.2.1 Why use fuzzy controllers?

In convential control theory, we use mathematical models to design a controller. However, if it is hard to obtain a model, or if the system is highly nonlinear, this approach doesn't work so well. Luckily, fuzzy models can be used to approximate nonlinear functions. So, we can make a fuzzy controller! A **fuzzy controller** is a controller that contains a mapping that has been defined using if-then rules.

In the world of control systems, a lot of systems are nonlinear. So, the demand for nonlinear control methods is big. Such methods need to have several properties. First of all, it would be nice if they don't

need too many input variables to obtain good results. Also, the method needs to be able to deal with nonlinearities well, it should have a good learning/training rate, it should not be too computationally intensive, and several more criteria need to be met. Fuzzy logic is a method that performs quite well on these criteria.

4.2.2 The Mamdani controller

Several types of fuzzy controllers exist. One important example is the **Mamdani controller**. It is usually used as a feedback controller.

As input, the Mamdani controller generally receives the error signal e. This error signal is then processed by **dynamic pre-filters**. These pre-filters often **scale** the data, for example to put it on the normalized interval $[-1, 1]$. Also, **dynamic filtering** is often applied, where quantities like $\dot{e}$ and $\int e$ are derived. Other pre-filtering methods can be applied as well. The resulting parameters are then fed into the part of the fuzzy controller known as the **static map**. The output of the static map is then fed to **dynamic post-filters**. These filters can again scale the data and/or apply dynamic filtering.

Let's take a closer look at the static map. It is important to define it properly. So, we're going to look at the individual steps needed to design the static map.

- First, the necessary **inputs and outputs** need to be selected. For conventional linear controllers, it can be advantageous to use integral or derivative gains. Similarly, for fuzzy logic, it can sometimes be advantageous to also use parameters like $\dot{e}$, $\int e$ and/or others as inputs. However, things can get complicated if too many input variables are used. If this is the case, then it may be worthwhile to split up the system into several subsystems and let the output from one subsystem be the input to the other.
- Second, the **number of linguistic terms** for every input variable needs to be set. If too few terms are used, then the fuzzy system doesn't have a lot of flexibility – it can't approximate all kinds of functions. However, if too many linguistic terms are used, the rule base will become rather big.
- For every linguistic term, a **membership function** needs to be selected. For computational reasons, triangular and trapezoidal functions are usually preferred to bell-shaped functions.
- A very important step is to design the **rule-base**. This is generally done based on the knowledge of an expert. Also, a model of the system may be used.
- Finally, the fuzzy controller needs to be **tuned**. This step is just as important as the tuning of the gains in a conventional PID controller. The tuning of a fuzzy controller can be difficult because sometimes, by changing only one variable, the whole system changes. But luckily, the effects of changing parameters in a fuzzy controller are usually quite localized.

4.2.3 The Takagi-Sugeno controller and supervisory control

The Takagi-Sugeno fuzzy controller is somewhat similar to gain scheduling. Rules now take the form

$$\textbf{if } e \text{ is } \mathit{Low} \textbf{ then } u = a_i e + b_i \tag{4.2.1}$$

or something similar. Each rule is valid for only a small region of the controller's input space. So, every region of the controller's input space more or less has its own control law. The general controller then simply interpolates between these control laws.

Something entirely different is the **supervisory controller**: it is a secondary controller. It augments an existing (conventional) controller. To do this, the supervisory controller usually defines certain parameters of the existing controller. For example, it may define the proportional and derivative gains K_p and K_d of a conventional PID controller, based on the current state of the system. Rules can thus take the form of

$$\textbf{if} \text{ process output is } \mathit{High} \textbf{ then} \text{ reduce } K_p \ \mathit{Slightly} \textbf{ and} \text{ increase } K_d \ \mathit{Moderately}. \tag{4.2.2}$$

5. Artificial neural networks

Artificial neural networks (ANNs) are imitations of biological neural networks, like our brains. They can be very adept at approximating nonlinear functions, even when only few training data is available. How they do this, and how they can be trained, will be examined in this chapter.

5.1 The structure of a neural network

5.1.1 The neuron

The basic building block of an ANN is a **neuron**. A neuron has several inputs x_i. Each of these inputs is multiplied by a weight w_i and then added up. Often, a bias b is added as well. The result is the neuron's **activation** z. So,

$$z = \sum_{i=1}^{p} w_i x_i = \mathbf{w}^T \mathbf{x} \qquad \text{or} \qquad z = \sum_{i=1}^{p} w_i x_i + b = \begin{bmatrix} \mathbf{w}^T & b \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}. \tag{5.1.1}$$

From the above equation, it can be seen that adding a bias b works the same as adding an additional input with weight b and value 1. So, we will simply use $z = \mathbf{w}^T \mathbf{x}$ in the remainer of this chapter.

Once the neuron's activation z has been obtained, it is fed into the **activation function** $\sigma(z)$. This function returns a value on the interval $[-1, 1]$ (or alternatively sometimes on the interval $[0, 1]$). Which activation function is used depends on the ANN designer's choice. Common activation functions are the **threshold function** and the **sigmoidal function**, respectively defined as

$$\sigma(z) = \begin{cases} 0 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases} \qquad \text{and} \qquad \sigma(z) = \frac{1}{1 + \exp(-sz)}. \tag{5.1.2}$$

The parameter s determines how 'steep' the sigmoid function is. If $s \to \infty$, then the threshold function is again obtained. But often $s = 1$ is simply chosen. The output of the activation is then the output of the neuron.

5.1.2 Neural network architecture

An artificial neural network consists of interconnected neurons. (That is, the output of one neuron is fed as input to the next neuron.) The neurons are usually assembled in layers. In a **feedforward network**, the neurons of every layer are connected to the next layers. On the other hand, in **recurrent networks**, neurons are also connected to previous layers as some sort of 'feedback mechanism'. We will mainly consider feedforward networks though, because they are relatively simple.

ANNs always have an **input layer** (at the start) and an **output layer** (at the end). Often, there are also **hidden layers** in between. Most of the times, only one hidden layer is used. The reason is that multiple hidden layers make the neural network computationally quite complex. Also, one hidden layer is already capable of approximating any continuous function. That is, as long as there is a sufficient number of hidden neurons in it.

Choosing the right number of hidden neurons in the hidden layer is very important, but also very difficult. If you use too few neurons, then the neural network can't approximate the desired output function well enough. If, however, you use too many neurons, then overtraining can occur: the system only works on the few test samples that have been provided, but is useless for any other input. Generally, the number of hidden neurons primarily depends on the number of training samples (more training samples implies that more neurons can be used) and the complexity of the output function (more complex output functions often require more neurons).

5.1.3 Finding the output of a neural network

Let's suppose that we have an ANN with 1 hidden layer. Given an input $\mathbf{x_i}$, how do we find the output $\mathbf{y_i}$ of this network?.

Well, we start with the input $\mathbf{x_i}$. The input layer doesn't really do anything with this input. It only passes it on to every neuron of the hidden layer. The activation of one hidden neuron j can then be found using $z_j = \mathbf{w_j^h}^T\mathbf{x}$. But we usually have multiple (N) input samples $\mathbf{x_i}$ and multiple (p) hidden neurons j. So, we can define

$$\mathbf{X} = \begin{bmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \vdots \\ \mathbf{x_N}^T \end{bmatrix}, \qquad \mathbf{W^h} = \begin{bmatrix} \mathbf{w_1^h}^T \\ \mathbf{w_2^h}^T \\ \vdots \\ \mathbf{w_p^h}^T \end{bmatrix} \quad \text{and} \quad \mathbf{Z} = \begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1p} \\ z_{21} & z_{22} & \cdots & z_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ z_{N1} & z_{N2} & \cdots & z_{Np} \end{bmatrix}. \tag{5.1.3}$$

The element z_{ij} of $\mathbf{Z}$ thus denotes the activation of hidden neuron j to input sample $\mathbf{x_i}$. Now we can simply find the hidden layer activation $\mathbf{Z}$, the hidden layer output $\mathbf{V}$ and the system output $\mathbf{Y}$ using

$$\mathbf{Z} = \mathbf{X}\mathbf{W^h}, \qquad \mathbf{V} = \sigma(\mathbf{Z}) \quad \text{and} \quad \mathbf{Y} = \sigma(\mathbf{V}\mathbf{W^o}). \tag{5.1.4}$$

Here, we have $\mathbf{Y} = \begin{bmatrix} \mathbf{y_1} & \mathbf{y_2} & \cdots & \mathbf{y_N} \end{bmatrix}^T$. Also, the output layer weights $\mathbf{W^o}$ are defined similarly as the hidden layer weights $\mathbf{W^h}$.

5.2 Training neural networks

Before ANNs work, they need to be trained. That is, their weights (and biases) need to be set such that certain inputs give certain outputs. So, let's suppose that we have a set of inputs $\mathbf{X}$ with **desired outputs D**. How do we find the right weights? Several techniques exist. One of the simplest is the **backpropagation technique**. Let's examine it.

5.2.1 Backpropagation – the output layer

First, we randomly initialize the neural network. We then take an input $\mathbf{x}$ and find the resulting output $\mathbf{y}$. We compare this with the desired output $\mathbf{d}$ and calculate the error $\mathbf{e} = \mathbf{d} - \mathbf{y}$. Our goal now is to minimize the cost function

$$J = \frac{1}{2}\sum_l e_l^2. \tag{5.2.1}$$

First, let's focus on minimizing the contribution of the output layer. For simplicity, we assume that the output layer has no activation function. So, we simply have $y_l = \sum_j w_{jl}^o v_j$. We now adjust the weights of the output layer using the update rule

$$w_{jl}^o(n+1) = w_{jl}^o(n) - \alpha(n)\frac{\partial J}{\partial w_{jl}^o}. \tag{5.2.2}$$

Here, $\alpha(n)$ is the **learning rate**. To find the **Jacobian** $\partial J/\partial w_{jl}^o$, we can use the chain rule. So,

$$\frac{\partial J}{\partial w_{jl}^o} = \frac{\partial J}{\partial e_l}\frac{\partial e_l}{\partial y_l}\frac{\partial y_l}{\partial w_{jl}^o}. \tag{5.2.3}$$

These three partial derivatives are all relatively easy to find. We have

$$\frac{\partial J}{\partial e_l} = e_l, \qquad \frac{\partial e_l}{\partial y_l} = -1 \qquad \text{and} \qquad \frac{\partial y_l}{\partial w_{jl}^o} = v_j. \tag{5.2.4}$$

Thus, the update rule for the output layer weights becomes

$$w_{jl}^o(n+1) = w_{jl}^o(n) + \alpha(n) v_j e_l. \tag{5.2.5}$$

5.2.2 Backpropagation – the hidden layer

A similar principle is applied when adjusting the weights for the hidden layer. But now the Jacobian is given by

$$\frac{\partial J}{\partial w_{ij}^h} = \frac{\partial J}{\partial v_j} \frac{\partial v_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}^h}. \tag{5.2.6}$$

Finding the partial derivatives now is a bit more difficult. But, after some computation, we can find that

$$\frac{\partial J}{\partial v_j} = \sum_l \left(\frac{\partial J}{\partial e_l} \frac{\partial e_l}{\partial y_l} \frac{\partial y_l}{\partial v_j} \right) = \sum_l -e_l w_{jl}^o, \qquad \frac{\partial v_j}{\partial z_j} = \sigma_j'(z_j) \qquad \text{and} \qquad \frac{\partial z_j}{\partial w_{ij}^h} = x_i. \tag{5.2.7}$$

These data result in the update law for hidden neuron weights, being

$$w_{ij}^h(n+1) = w_{ij}^h(n) + \alpha(n) x_i \sigma_j'(z_j) \sum_l e_l w_{jl}^o. \tag{5.2.8}$$

By using this equation, the weights of the hidden layer are adjusted.

When using the backpropagation technique, you usually use a set of N test samples $(\mathbf{x_i}, \mathbf{d_i})$ to adjust the weights. The presentation of the whole training set to the system is called an **epoch**. Multiple epochs are necessary before the backpropagation algorithm converges to a minimum. However, since backpropagation is a gradient descent method, it is quite likely that the resulting minimum is a local minimum. This is a significant downside of the backpropagation method.

5.2.3 The radial basis function network

An other type of neural network is the **radial basis function network** (RBFN). This network has a hidden layer. However, the neurons in this hidden layer don't have weights. Also, there is no real activation function. Instead, a **radial basis function** (RBF) $\phi_i(r)$ is used. A common RBF is the **Gaussian function**

$$\phi_i(r) = \exp\left(-\frac{r^2}{\rho_i^2} \right), \qquad \text{where } r = ||\mathbf{x} - \mathbf{c_i}||. \tag{5.2.9}$$

The output layer does have weights, but it does not have an activation function. The output of an output node is thus given by

$$y_j = \sum_{i=1}^{n} w_{ij} \phi_i \left(||\mathbf{x} - \mathbf{c_i}|| \right). \tag{5.2.10}$$

If we put the outputs of the RBFs in a row vector $\mathbf{V} = [\phi_1(r) \ \ldots \ \phi_n(r)]$, then the output equation reduces to $\mathbf{y} = \mathbf{V}\mathbf{w}$. This is a linear equation. So, if we have a set of known inputs $\mathbf{x}$ with desired outputs $\mathbf{d}$, then we can use the least-squares theorem to find $\mathbf{w}$. To do this, we simply have to find $\mathbf{V}$ and apply

$$\mathbf{w} = \left(\mathbf{V}^T \mathbf{V} \right)^{-1} \mathbf{V}^T \mathbf{d}. \tag{5.2.11}$$

In this way, the weights of the network can be trained. However, training the values of $\mathbf{c_i}$ and ρ_i is not possible in this way, since the output $\mathbf{y}$ doesn't linearly depend on these parameters. Instead, more complicated nonlinear optimization methods need to be applied.

6. Control using knowledge-based models

Previously, we have examined a lot of knowledge-based models. Now it is time to look at how we can control systems with them. First, we investigate the working principle of inverse control. After that, we also look at other types of knowledge-based control.

6.1 Inverse control

6.1.1 Basics of inverse control

Let's examine a system. For simplicity, we will examine a **single-input single-output** (SISO) system. This system can be described by a **model** of the system

$$y(k+1) = f(\mathbf{x}(k), u(k)). \tag{6.1.1}$$

Let's suppose that we know the function f and that we can find an inverse function f^{-1} such that

$$u(k) = f^{-1}(\mathbf{x}(k), y(k+1)). \tag{6.1.2}$$

Now let's say that we want to reach a **desired state** $r(k+1)$. Then we simply replace $y(k+1)$ by $r(k+1)$ in the inverse function. The resulting value of $u(k)$ will be the input that makes sure that $r(k+1)$ is reached. This method of controlling a system is called **inverse control**.

There are various ways in which we can implement inverse control. Which one we use depends on whether we know the state $\mathbf{x}$. In **open-loop feedback control** we use the output of the system to determine $\mathbf{x}$. This state is then inserted into the inverse function f^{-1} to find the required input $u(k)$.

However, sometimes we can't use the output $y(k)$ of the system to find the state $\mathbf{x}$. In this case, **open-loop feedforward control** is an alternative. Now we use the model f of the system to keep track of $\mathbf{x}(k)$. This method has as a downside that the error between the model and the actual system can grow over time. So, an accurate model needs to be available.

The question remains how you can find the inverse f^{-1}. This is mostly done using numerical methods. In these methods, you try to minimize an objective function like

$$J(u(k)) = (r(k+1) - f(\mathbf{x}(k), u(k)))^2. \tag{6.1.3}$$

For some models, the inverse f^{-1} can be computed analytically. Let's examine a few of such cases.

6.1.2 The inverse of an affine TS fuzzy model

Let's examine an affine TS fuzzy model in which the rules do not contain the input $u(k)$ in the antecedent. Instead, $u(k)$ only occurs in the consequent. So, the K rules will have the form

$$\mathcal{R}_i: \textbf{ if } \mathbf{x}(k) \text{ is } X_i \textbf{ then } y(k+1) = \mathbf{a_i}^T\mathbf{x}(k) + b_i u(k) + c_i. \tag{6.1.4}$$

To find the actual value of $y(k+1)$ we have to know the degree of fulfillment $\beta_i(\mathbf{x}(k))$ or the corresponding normalized degree of fulfillment $\lambda_i(\mathbf{x}(k))$. If we do, then we can find $y(k+1)$ using

$$y(k+1) = \frac{\sum_{i=1}^{K} \beta_i(\mathbf{x}(k)) \left(\mathbf{a_i}^T\mathbf{x}(k) + b_i u(k) + c_i\right)}{\sum_{i=1}^{K} \beta_i(\mathbf{x}(k))} = \sum_{i=1}^{K} \lambda_i(\mathbf{x}(k)) \left(\mathbf{a_i}^T\mathbf{x}(k) + b_i u(k) + c_i\right). \tag{6.1.5}$$

It can be noted that this equation is linear in $u(k)$. So, it can easily be solved for $u(k)$. If we also replace the output $y(k+1)$ by the desired output $r(k+1)$, we will find that

$$u(k) = \frac{r(k+1) - \sum_{i=1}^{K} \lambda_i(\mathbf{x}(k)) \left(\mathbf{a_i}^T\mathbf{x}(k) + c_i\right)}{\sum_{i=1}^{K} \lambda_i(\mathbf{x}(k)) b_i}. \tag{6.1.6}$$

This is the inverse function f^{-1} of the system model f.

6.1.3 The inverse of a singleton model

We now examine a similar case: the singleton model. However, there is an important difference. The input $u(k)$ now isn't in the consequent anymore, but in the antecedent. So, the rules have the form

$$\mathcal{R}_{ij}: \textbf{ if } \mathbf{x}(k) \text{ is } X_i \textbf{ and } u(k) \text{ is } U_j \textbf{ then } y(k+1) = c_{ij}. \tag{6.1.7}$$

The problem now is that the degree of fulfillment $\beta_{ij}(k)$ of rule ij at time k not only depends on the (known) state $\mathbf{x}(k)$, but on the (to-be-determined) input $u(k)$ as well. Luckily, this can be solved, if we use the product t-norm operator. We then have

$$\beta_{ij}(k) = \mu_{X_i}(\mathbf{x}(k)) \cdot \mu_{B_j}(u(k)). \tag{6.1.8}$$

We also assume that the antecedent membership functions $\mu_{B_j}(u(k))$ form a partition. That is,

$$\sum_{j=1}^{N} \mu_{B_j}(u(k)) = 1. \tag{6.1.9}$$

If this is the case, and if the state $\mathbf{x}(k)$ is known, then we can simplify the rule base. To do this, we first define

$$c_j(k) = \sum_{i=1}^{M} \lambda_i(\mathbf{x}(k)) \cdot c_{ij}. \tag{6.1.10}$$

The rules can now be simplified to

$$\mathcal{R}_j: \textbf{ if } u(k) \text{ is } U_j \textbf{ then } y(k+1) = c_j. \tag{6.1.11}$$

The main trick to invert the singleton model is to invert the rule base. There is just one problem: $c_j(k)$ is not a fuzzy set. As a solution, we use the fuzzy sets $C_j(k)$. All rules are thus rewritten as

$$\mathcal{R}_j: \textbf{ if } r(k+1) \text{ is } C_j(k) \textbf{ then } u(k) = U_j. \tag{6.1.12}$$

The fuzzy sets $C_j(k)$ are defined as to have triangular membership functions. Also, all membership functions add up to one. So,

$$\mu_{C_j(k)}(r) = \begin{cases} \max\left(0, \min\left(1, \frac{c_2 - r}{c_2 - c_1}\right)\right) & \text{if } j = 1, \\ \max\left(0, \min\left(\frac{r - c_{j-1}}{c_j - c_{j-1}}, \frac{c_{j+1} - r}{c_{j+1} - c_j}\right)\right) & \text{if } 1 < j < N, \\ \max\left(0, \min\left(\frac{r - c_{N-1}}{c_N - c_{N-1}}, 1\right)\right) & \text{if } j = N. \end{cases} \tag{6.1.13}$$

In the above equation, N is the number of fuzzy sets U_j corresponding to the input $u(k)$. By the way, sometimes it may occur that a rule base is not invertible. In this case, you first need to split up the rule base into invertible parts, and afterwards connect them again. However, we won't go any further into detail on that here.

6.1.4 Other types of inverse control

Some models have an **input delay** n_d. The system model is then given by $y(k+1) = f(\mathbf{x}(k), u(k-n_d))$. We cannot invert this function directly, since $u(k)$ can't affect $y(k+1)$. The first output which is affected by $u(k)$ is $y(k+n_d+1)$. We thus use as inverse function

$$u(k) = f^{-1}(r(k+n_d+1), \mathbf{x}(k+n_d)). \tag{6.1.14}$$

The only problem is finding $\mathbf{x}(k+n_d)$. Luckily, it does not depend on $u(k)$ or any of the later inputs. And all the previous inputs $u(m)$ with $m < k$ are already known. So, $\mathbf{x}(k+n_d)$ can simply be predicted using our known model.

Another problem occurs when the system is subject to (output) noise. This causes the system output $y(k)$ to be a bit unreliable. In this case, **internal model control** (IMC) offers a solution. When IMC is applied, we use a model of the system without noise. This system predicts the output $y_m(k)$ of the system without noise. The difference $y(k) - y_m(k)$ is then used to change the desired input $r(k+1)$, such that it is more accurately reached. In this way, the effects of the noise are significantly reduced.

6.2 Other types of knowledge-based control

6.2.1 Model-based predictive control

Let's suppose that we have a system, of which we have a model f. We now need to decide on a series of H_c inputs $\mathbf{u}$. Based on these inputs, the following H_p predicted outputs $\hat{\mathbf{y}}$ are determined. (To find all these outputs, we usually assume that, after a time $k + H_c$, the input u stays constant.) How do we decide on these inputs?

Usually, the inputs $\mathbf{u}$ are chosen such that a cost function is minimized. This cost function usually looks like

$$J = \sum_{i=1}^{H_p} ||\mathbf{r}(k+i) - \hat{\mathbf{y}}(k+i)||^2_{\mathbf{P_i}} + \sum_{i=1}^{H_c} ||\Delta\mathbf{u}(k+i-1)||^2_{\mathbf{Q_i}}. \tag{6.2.1}$$

The first parts adds a 'penalty' if the output deviates from the desired output. The second part adds a penalty if the input changes a lot. (That is, if the control effort is high.) The matrices $\mathbf{P_i}$ and $\mathbf{Q_i}$ should be chosen such that the right parts of the input and the output are prioritized/penalized.

When a set of H_c inputs has been decided in this way, only the first of these inputs $\mathbf{u}(k)$ is executed. The resulting output $y(k+1)$ is then examined. Based on this data, a new series of inputs $\mathbf{u}$ is determined, after which again only the first one is executed. This is called the **receding horizon** principle.

You may think that the receding horizon principle is silly: why determine a whole set of future input values, when only the first one of them is executed? The reason behind this is that possibilities for future inputs are also taken into account. If, on the other hand, you only look at the input $\mathbf{u}(k+1)$ at time $k+1$, it might occur that you select an input $\mathbf{u}$ with very good short-term effects, but which does put the system into trouble after that.

6.2.2 Adaptive control

Sometimes, we encounter a process of which the behavior changes over time. A controller with fixed parameters won't work anymore. Instead, **adaptive control** is required.

We can make a distinction between indirect and direct adaptive control. In **indirect adaptive control**, we use a model of the system which is continuously adapted. (For example, by comparing the predicted output $\mathbf{y_m}$ of the model by the actual output $\mathbf{y}$ of the system.) This model is then used to determine the controller parameters. (For example, the model can be inverted at every time step to find the controller.) On the other hand, in **direct adaptive control**, we don't use a model. We then simply directly adapt the controller parameters.

7. Reinforcement learning

In this chapter, we discuss the technique called reinforcement learning. First, we examine the basic principles. Then we look at how it is applied when a model of the environment is present. Next, we examine what to do when a model of the environment is missing. Finally, we look at how we can apply reinforcement learning to control problems.

7.1 Basics of reinforcement learning

7.1.1 Definitions in reinforcement learning

In **reinforcement learning** (RL), there is an **agent** and an **environment**. The agent has a certain **state** $s_k \in S$. During every step, the agent needs to choose one of the possible **actions** $a_k \in A$. He then reaches a new state s_{k+1}. By doing this, he gets an **immediate reward** $r_k \in \mathbb{R}$ from the environment.

The goal of the agent now is to maximize the **total reward** R_k. This total reward is a function of all future rewards. Often, the sum is used. So, $R_k = r_{k+1} + r_{k+2} + \ldots$. Another often-used function is

$$R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n r_{k+n+1}. \tag{7.1.1}$$

The parameter γ, which satisfies $0 \leq \gamma \leq 1$, is called the **discount rate**. We will use the latter total reward function in the remainder of this chapter.

The whole point of reinforcement learning is to find the optimal **policy**. A policy is a mapping: for every state s, it maps which action a is chosen by the agent in that state. If we can write $a = \pi(s)$, then we deal with a **deterministic policy**: for every state s, always the same action a is chosen. However, we can also deal with a **stochastic policy**. In this case, $\Pi(s, a)$ denotes the probability that in state s action a is chosen by the agent.

7.1.2 The environment

Let's suppose that the agent is in some state s_k and chooses action a_k. Also, all the previous states and actions $s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots$ are known. In a **stochastic environment**, it is uncertain in which state s_{k+1} the agent winds up in. The probability that the agent reaches state s_{k+1} with reward r_{k+1} is denoted by

$$P\left(s_{k+1}, r_{k+1} | s_k, a_k, s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots\right). \tag{7.1.2}$$

However, usually we assume that the system has the **Markov property**. This means that the state and reward at time $k+1$ only depends on the state and action at time k. Thus, the above probability is simply written as $P\left(s_{k+1}, r_{k+1} | s_k, a_k\right)$. An RL task which satisfies this property is called a **Markov decision process** (MDP).

Let's discuss some more notations. We denote the chance that the agent winds up in state s', given that he now is in state s and chooses action a, by

$$\mathcal{P}^a_{ss'} = P\left(s_{k+1} = s' | s_k = s, a_k = a\right). \tag{7.1.3}$$

This function is called the **state transition probability function**. Similarly, we can define the **expected reward** as

$$\mathcal{R}^a_{ss'} = E\left\{r_{k+1} | s_k = s, a_k = a, s_{k+1} = s'\right\}. \tag{7.1.4}$$

Here, we do have assumed that the agent always knows in which state he is. If the agent can't always observe in which state he is in, then we are dealing with a **partially observable MDP** (POMDP). We won't deal with POMDP problems though.

7.1.3 The value function

Let's suppose that we have an agent that is in some state s. This agent also has a policy π. The **value function** $V^\pi(s)$ now is the expected total reward R_k when the policy π is used. So,

$$V^\pi(s) = E^\pi \{R_k | s_k = s\} = E^\pi \left\{ \sum_{n=0}^{\infty} \gamma^n r_{k+n+1} | s_k = s \right\}. \tag{7.1.5}$$

By the way, E^π is the expectation operator, given that the agent follows the policy π. In a similar way, we can define the **action-value function** $Q^\pi(s, a)$ as the expected total reward R_k when an agent chooses action a in state s and follows policy π afterwards. So,

$$Q^\pi(s, a) = E^\pi \{R_k | s_k = s, a_k = a\} = E^\pi \left\{ \sum_{n=0}^{\infty} \gamma^n r_{k+n+1} | s_k = s, a_k = a \right\}. \tag{7.1.6}$$

When applying RL, we always use either V or Q, never both. However, sometimes V is convenient to use and sometimes Q. So, in this summary, we will treat them both.

The goal of reinforcement learning is to find an **optimal policy** π^*. This optimal policy π^* is the policy π which maximizes the value function V^π or, alternatively, Q^π. How this policy can be found depends on the type of problem.

7.2 Model based RL

7.2.1 The Bellman optimality equation

Sometimes we have an exact model of the environment. Solution techniques to find the optimal policy are now known as **dynamic programming**.

Let's suppose that we are in a state s and choose an action a. If we do this, then there is a chance $\mathcal{P}^a_{ss'}$ that we wind up in state s'. In this state, our expected reward will be the sum of our immediate reward $\mathcal{R}^a_{ss'}$, and of the expected total reward of future states $\gamma V^*(s')$. (Note that a discount rate has to be added.) Based on this, we can find the expected total reward of choosing action a. Of course, we want to choose the action a which maximizes the expected total reward. This logic results in the recursively defined **Bellman optimality equation**

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^*(s') \right). \tag{7.2.1}$$

A similar equation can be derived for Q. We then get

$$Q^*(s, a) = \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right). \tag{7.2.2}$$

Solving for the value function can be quite difficult though. So we'll treat that in the next paragraph separately.

You may wonder, when we have the value function V^* (or Q^*), how do we find the optimal policy? Well, in this case the optimal policy is the so-called **greedy policy**. We simply take the action a which maximizes the value function. So,

$$\pi(s) = \arg\max_{a \in A} \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s) \right) \qquad \text{or} \qquad \pi(s) = \arg\max_{a \in A} Q^*(s, a). \tag{7.2.3}$$

7.2.2 Finding the optimal value function

There are two often-used methods to find the optimal value function. One of them is **policy iteration**. We start with a certain initialization $V_0(s)$ of the value function and with a certain policy π. We then simply iterate.

During every step, there is a **policy evaluation** and a **policy improvement** step. In the policy evaluation step, we use the policy to update the value function. This is done according to

$$V_{n+1}(s) = E^{\pi}\left\{r_{k+1} + \gamma V_n(s_{k+1})\right\} = \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V_n(s')\right), \qquad \text{with} \qquad a = \pi(s). \tag{7.2.4}$$

In the policy improvement step, we improve our policy. In fact, as policy the greedy policy π is used, corresponding to the value function $V_{n+1}(s)$. These steps are then iterated until a stopping criterion is met. For example, the policy π hasn't changed for several consecutive iterations, or the difference in the value function $V(s)$ is below a certain threshold ϵ.

A similar method is the **value iteration** method. In this method, no policy is computed anymore. Instead, the value function is updated directly using

$$V_{n+1}(s) = \max_a E\left\{r_{k+1} + \gamma V_n(s_{k+1}) | s_k = s, a_k = a\right\} = \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V_n(s')\right). \tag{7.2.5}$$

7.3 Model free RL

7.3.1 Temporal difference methods

It may occur that we don't have any model of our environment. In this case, the agent simply needs to explore it. There are several ways to do this. But most of the methods do use a value function. Among these methods are the **temporal difference** (TD) methods.

Let's suppose that we are in some state s_k. We then go to a state s_{k+1} in which we receive a reward r_{k+1}. We use this reward to update $V(s_k)$. This kind of makes sense: if r_{k+1} is big, then $V(s_k)$ should have been big as well, while if r_{k+1} is small, then $V(s_k)$ should have been small as well. The equation that is used is

$$V(s_k) \leftarrow (1-\alpha_k)V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1})\right) = V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)\right) = V(s_k) + \alpha_k \delta_k. \tag{7.3.1}$$

In the above equation, α_k is the **learning rate** at time k. Also, $\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k)$ is the **TD-error**.

You might be wondering, why do we use r_{k+1} to only update s_k. Can't we use r_{k+1} to update $s_{k-1}, s_{k-2}, \ldots$ as well? Well, we can. The question just is: how much should we update them? For this, we define the **eligibility trace** $e_k(s)$. This eligibility trace can be seen as the 'strength' of the relation between the reward r_{k+1} and the state s. If, for example, $s = s_{k-1}$, then there is a relatively strong relation between s and r_{k+1}. So, $e_k(s)$ should be big. On the other hand, if $s = s_{k-20}$, then $e_k(s)$ should be small. So, we can define $e_k(s)$ as

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k, \\ 1 & \text{if } s = s_k. \end{cases} \tag{7.3.2}$$

The parameter λ is called the **trace-decay parameter**. (γ is still the discount rate.) Based on this eligibility trace, we can update $V(s)$. The change in $V(s)$ (denoted as $\Delta V(s)$) is now given by

$$\Delta V(s) = \alpha \delta_k e_k(s). \tag{7.3.3}$$

7.3.2 Q-learning and SARSA

Another model free RL method is Q-learning. It is a so-called **off-policy** method: it doesn't use a policy while learning. Instead, it simply uses the action-value function $Q(s, a)$ to learn. To start, we give the function $Q(s, a)$ initial values. We then update it using

$$Q(s_k, a_k) \to Q(s_k, a_k) + \alpha \left(r_{k+1} + \gamma \max_a Q(s_{k+1}, a) - Q(s_k, a_k) \right). \tag{7.3.4}$$

How does this work? Well, let's suppose that we want to update $Q(s_k, a_k)$. We then start in state s_k, choose action a_k, which brings us in state s_{k+1} with immediate reward r_{k+1}. The new value $Q(s_k, a_k)$ then depends on the old value, the immediate reward r_{k+1} which we received and the maximum expected future reward $Q(s_{k+1}, a)$ which we expect to be able to get. However, to make sure that the algorithm converges, we do have to visit all state-action pairs (s_k, a_k) continually.

If also eligibility traces are used, then the above equation turns into

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta_k e_k(s, a), \qquad \text{where} \quad \delta_k = r_{k+1} + \gamma \max_{a'} Q(s_{k+1}, a') - Q(s_k, a_k). \tag{7.3.5}$$

Another method, which is somewhat similar to Q-learning, is the **SARSA** method. But contrary to Q-learning, SARSA is an **on-policy** method. That is, it does require a policy π or Π. This time, we update $Q(s_k, a_k)$ using

$$Q(s_k, a_k) \to Q(s_k, a_k) + \alpha \left(r_{k+1} + \gamma Q(s_{k+1}, a_{k+1}) - Q(s_k, a_k) \right). \tag{7.3.6}$$

The action a_{k+1} follows from the policy. So, $a_{k+1} = \pi(s_{k+1})$ or, alternatively, the chance that an action a is chosen to be a_{k+1} is $\Pi(s, a)$.

7.3.3 Exploration

Previously, we saw that, to apply Q-learning, we need to examine all possible state-action combinations (s_k, a_k). But what do we do if the agent can't choose which state he is in? (That is, if he can only just 'walk' around?) In this case, it would be bad to stick to our policy. Instead, we need to **explore**. And although most of the times an **explorative action** gives a lower reward than the action we would otherwise choose, sometimes it may give a higher reward. And this will result in a better eventual outcome.

There are several ways to explore. However, we will only consider one group of methods, called **undirected exploration**. It simply means that there is a chance that you select a random action. For example, when following an **ϵ-greedy policy**, there is a chance ϵ that you select a random action. In the other cases, you simply follow a normal greedy policy and thus choose the action with the highest $Q(s, a)$ value.

Another type of undirected exploration is **Max-Boltzmann exploration** (also called **soft-max exploration**). Now, the chance that we choose an action a is given by

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}. \tag{7.3.7}$$

The parameter τ is a variable that determines how much you explore. If $\tau = \infty$, you select actions fully randomly (as if $\epsilon = 1$). But if $\tau = 0$, you are back to the greedy policy.

We could also use **optimistic initial values**. What this means is that we initialize $Q(s, a)$ (or alternatively, $V(s)$) with very high values. We then follow a greedy policy with a normal updating method for Q. So, when you try an action a, the $Q(s, a)$ value will very likely decrease. So the next time you arrive at state s, you will choose a different action. Only when a $Q(s, a)$ value stops to decrease, will you

continue to follow the same action. And of course, the first action a for which $Q(s, a)$ stops to decrease is quite likely the best action.

A very interesting question to ask is: how much should you explore? This is called the **exploration vs. exploitation dilemma**. Initially, you should explore quite a bit. But as time progresses, and you are bound to have found some good sets of actions, you should exploit. Thus, when applying an ϵ-greedy policy or a Max-Boltzmann policy, the value of ϵ or τ should decrease over time.

7.4 Application of RL to control systems

Let's suppose that we have some system which we want to control. How can we use RL for this? The first problem which we run into is that in RL, all states and actions are discrete. But in most control problems, the states $\mathbf{x}$ are continuous. The first step in applying RL is thus the **quantization of state variables**.

The second step which you need to do is define the **action set** A. (That is, the set of all possible actions.) An example of an action might be 'a_1 = apply maximum negative input' and 'a_2 = apply maximum positive input'. But more difficult control laws can also be used, like 'a_1 = use fuzzy controller number 1' and 'a_2 = use fuzzy controller number 2' or something similar.

The third step is to **define the reward function**. What states do we want to reach? (Give these a high reward.) And what states do we definitely want to avoid? (Give those a low reward.) Important when defining the reward function is the rule: 'you should only tell the agent what it should do, and not how.' If you do this, then the RL algorithm might just come up with a very surprising but very effective solution.

Finally, the **results** of the algorithm should be examined. Does the resulted policy control the system sufficiently? If not, what went wrong? Can you fix it by doing the previous steps in a different way?

8. Swarm intelligence

In **swarm intelligence**, we deal with **swarms**: large groups of N individuals. (Think of flocks of birds or schools of fish.) Each individual has its own behavior and goals. And although the behavior of each individual might be simple, the whole swarm often behaves itself in a complicated yet effective way. This phenomenon is called **emergence**.

Using swarm intelligence has several advantages. All the individuals, called agents, can be produced in series. This saves costs. Also, the swarm is robust: if one agent fails, the swarm still functions. Finally, the swarm is easily scalable: you simply add more agents.

In this chapter, we'll examine three methods that use swarm intelligence. Let's start off by examining particle swarm optimization.

8.1 Particle swarm optimization

A particular application of swarm intelligence is **particle swarm optimization** (PSO). In PSO, we want to find the value $\mathbf{x}$ which minimizes a function $f(\mathbf{x})$. To do this, we create an n-dimensional search space, with n the size of the vector $\mathbf{x}$. In it, we put N particles. Every particle i has a (randomly initialized) position $\theta_i(k)$ and a velocity $\mathbf{v}_i(k)$ at time k. At every time step, the position of each particle is updated using

$$\theta_i(k+1) = \theta_i(k) + \mathbf{v}_i(k). \tag{8.1.1}$$

Also, the velocity is updated. This is done using

$$\mathbf{v}_i(k+1) = w(k)\mathbf{v}_i(k) + c_1 r_1(k)\left(\theta_{i,pbest}(k) - \theta_i(k)\right) + c_2 r_2(k)\left(\theta_{i,lbest}(k) - \theta(k)\right). \tag{8.1.2}$$

Let's walk through the terms in this equation. The first term is the **momentum term**. It causes particles to keep on going in the same direction as they currently are moving. The goal of this is to prevent particles from converging to a local minimum too quickly. By giving them momentum, they search the entire search space. Thus, the constant $w(k)$ is initially relatively big. (That is, almost equal to 1.) But as the algorithm proceeds, the constant is reduced.

The second term in the above equation is the **cognitive component**. The parameter $\theta_{i,pbest}(k)$ is the **personal best position**: the best position (with lowest $f(\theta)$) which particle i has found so far. This term thus causes the particle to be pulled back to its personal best. c_1 is a constant and $r_1(k)$ is a random variable, often uniformly distributed in the interval $[0, 1]$.

The third term, called the **social component**, is similar to the cognitive component. However, this time the particle compares its position to the **global best position** $\theta_{i,lbest}(k)$: the best position found by all particles together so far. The rest of the term works similarly.

When applying PSO, the particles start at random positions. Initially, they all move across the whole search space. But as time progresses, they should converge to minima of the function. When the algorithm is stopped, the actual solution is simply equal to $\theta_{i,lbest}(k)$: the best position found by all particles so far.

8.2 Artificial potential fields

Another way to search a space is by using an approach with **artificial potential fields**. The basic idea is that we have several particles with position $\mathbf{x}_i$. We now want to find the minimum for the function $\sigma(\mathbf{x})$. We then simply let each particle 'flow down'. This is done by applying a force on every particle of

$$\mathbf{u_i} = -\nabla_{\mathbf{x}}\sigma(\mathbf{x}_i). \tag{8.2.1}$$

Next to this, we also don't want particles to come closer together. It's no use if multiple particles search exactly the same space. To ensure that they don't, we use artificial potential fields. Every particle has a potential field around it, which repels other particles. This gives an additional force

$$\mathbf{u_{i,apf}} = \sum_{j=1, j\neq i}^{M} g_j(\mathbf{x_i} - \mathbf{x_j}). \tag{8.2.2}$$

To shorten notation, we usually write $\mathbf{y}_{ij} = \mathbf{x_i} - \mathbf{x_j}$. Examples of functions $g_j(\mathbf{y}_{ij})$ are

$$g(\mathbf{y}_{ij}) = -\mathbf{y}_{ij}\left(a - b\exp\left(-\frac{||\mathbf{y}_{ij}||^2}{c}\right)\right). \tag{8.2.3}$$

Next to this, obstacles might also be added. Just like particles, these obstacles do not move. They only repel other particles. Obstacles are useful if we want to restrict the search parameters to certain values.

8.3 Ant colony optimization

Ant colony optimization is a method to find the shortest path to a certain destination. Let's suppose that we have a graph. On a node i in this graph is an ant. This ant needs to select which arc he is going to walk on. The chance that he select an arc j at time k is given by

$$p_{ij}(k) = \frac{(\tau_{ij}(k))^{\alpha}(\eta_{ij})^{\beta}}{\sum_l (\tau_{il}(k))^{\alpha}(\eta_{il})^{\beta}}. \tag{8.3.1}$$

In this equation, $\tau_{ij}(k)$ denotes the **pheromone level** of the arc at time k. η_{ij} is a (constant) heuristic; for example the inverse of the length of the arc.

But we don't have one ant. We have N_a ants. Every ant chooses its arc at time step k in this way. After every ant has walked along its arc, the pheromone levels are updated. This is done using

$$\tau_{ij}(k+1) = (1-\rho)\tau_{ij}(k) + \sum_{a=1}^{N_a} \Delta\tau_{ij,a}(k). \tag{8.3.2}$$

Here, ρ is the **pheromone decay rate**. Also, we have

$$\Delta\tau_{ij,a}(k) = \begin{cases} F(s_a) & \text{if arc } (i,j) \text{ is used by ant } a, \\ 0 & \text{otherwise.} \end{cases} \tag{8.3.3}$$

Here, $F(s_a)$ is the **fitness function** of the node s at which ant a is. It can be seen as the amount of food at this node. What happens now is that arcs that lead to food sources (i.e. high fitness functions) get relatively high pheromone levels. So, they will be selected relatively often in the future as well. If, however, a path to a food source is found that is faster, then the ants will start to travel along that path more. And because it takes less time to walk along this path, more pheromone can be dropped along it. This route will thus become more preferable. In this way, the ants will find the fastest routes between food sources.

Extensions of the ant colony optimization method are also possible. For example, it can be combined with fuzzy logic. Now, an ant can be for a part in one node, and for another part in another node. And, although this can be a very interesting method, we won't go into depth on it here.