Contents

1 Graph neural networks

1.1 Graph definition

A **graph** G is defined as an ordered couple (V, E), where V is the set of **vertices** or **nodes** and E is the set of **edges**. Each edge represents the existence of a relationship betweeen two vertices. An edge $e \in E$ is therefore defined as an ordered couple $(u, v) \in V \times V$.

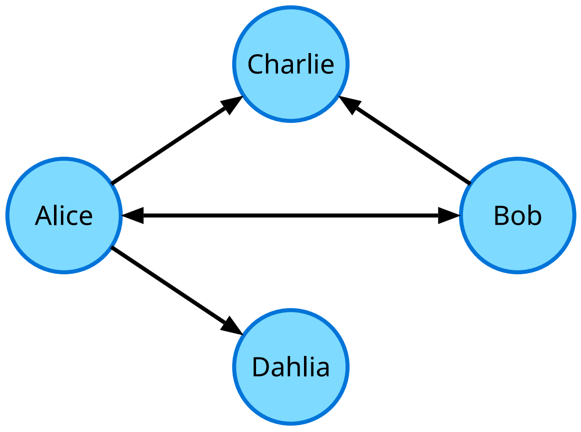
A graph is graphically represented using circles as nodes and using arrows as edges, whose tip is oriented in the direction of the edge. If the relationship holds both ways for a certain pair of nodes, the arrow is double-tipped.

A graph is matematically encoded in an **adjacency matrix**, a matrix that contains information concerning the existence of its edges. Formally, for a graph $G = (V, E)$ it is possible to construct an adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$ such that each entry (i, j) has the value 1 if $(i, j) \in E$ and 0 otherwise. Of course, the adjacency matrix of a graph can be constructed only if its edges can be enumerated.

Exercise 1.1

Alice knows Bob, Charlie and Dahlia, whereas Bob knows Alice and Charlie. Represent the relationship with a graph.

Proof:



$$\begin{bmatrix} & \text{Alice} & \text{Bob} & \text{Charlie} & \text{Dahlia} \\ \text{Alice} & 0 & 1 & 1 & 1 \\ \text{Bob} & 1 & 0 & 1 & 0 \\ \text{Charlie} & 0 & 0 & 0 & 0 \\ \text{Dahlia} & 0 & 0 & 0 & 0 \end{bmatrix}$$

□

The **neighborhood** of a node is the set of all nodes that connected to that node. Given a graph $G = (V, E)$, for any node $v \in V$ the neighborhood $N(v) \subseteq V$ is defined as the set $\{u \mid (v, u) \in E\}$ or, equivalently with respect to its adjacency matrix A, $\{u \mid A[v, u] \neq 0\}$.

A graph is said to be **connected** if every node appears in at least one edge, except for **loops** (an edge connecting a node to itself). Formally, a graph $G = (V, E)$ is connected if, for any $v \in V$, exists $u \in V - \{v\}$ such that $(u, v) \in E$ or $(v, u) \in E$.

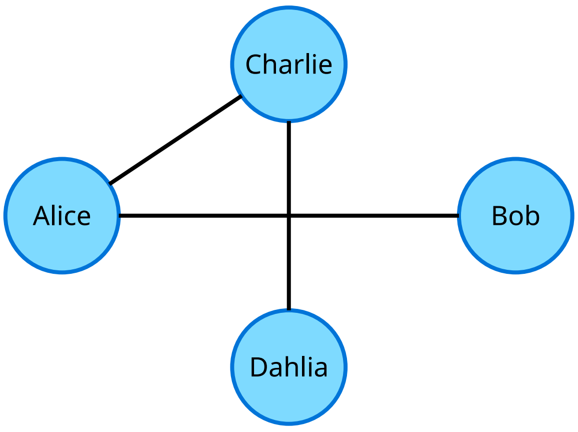
A graph is said to be **undirected** if the relationship between the nodes is symmetric, and holds both ways for every node. Formally, a graph $G = (V, E)$ is undirected if, for any $(u, v) \in E$, it is also true that $(v, u) \in E$. For clarity, the edges of an undirected graph are often drawn tipless. The adjacency matrix of an undirected graph will clearly be symmetric. If a graph is not undirected, it is said to be **directed**.

If a graph is connected, undirected and has no loops, it is called **simple**. It is easy to see that the adjacency matrix of a simple graph has 0 as each element of the diagonal.

Exercise 1.2

Alice is a friend of Bob and Charlie, whereas Dahlia is a friend of Charlie. Represent the relationship with a graph; is the graph simple?

Proof: Yes, the graph would be simple. This is because the “is a friend of” relationship is (assumed to be) symmetric, non reflexive and every person appears at least once as friend of someone else.



$$\begin{bmatrix} & \text{Alice} & \text{Bob} & \text{Charlie} & \text{Dahlia} \\ \text{Alice} & 0 & 1 & 1 & 0 \\ \text{Bob} & 1 & 0 & 0 & 0 \\ \text{Charlie} & 1 & 0 & 0 & 1 \\ \text{Dahlia} & 0 & 0 & 1 & 0 \end{bmatrix}$$

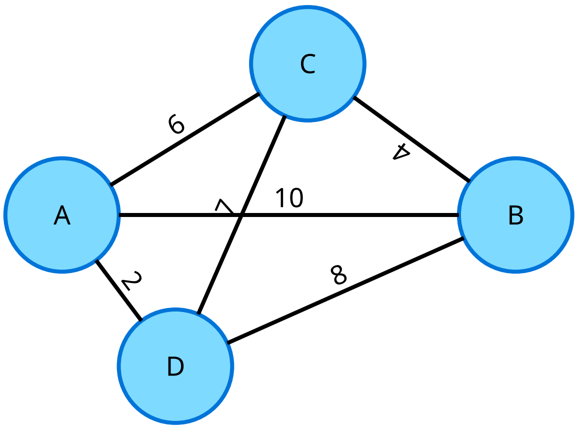
□

A graph $G = (V, E, W)$, that has a function $W : E \to \mathbb{R} - \{0\}$ that associates a real non-zero number to any edge of the graph is called **weighted graph**. The adjacency matrix of a weighted graph has $W(i, j)$ instead of 1 as the value of the (i, j)-th cell. The graphical representation of a weighted graph has the weight of each edge nenoted on the size of the corresponding arrow and, in general, the lenght of the arrow is scaled with respect to the weight.

Exercise 1.3

The town of A dists 10 from B, 6 from C and 2 from D, B dists 4 from C and 8 from D and D dists 7 from C. Represent the relationship with a graph.

Proof:



$$\begin{bmatrix} & A & B & C & D \\ A & 0 & 10 & 6 & 2 \\ B & 10 & 0 & 4 & 8 \\ C & 6 & 4 & 0 & 7 \\ D & 2 & 8 & 7 & 0 \end{bmatrix}$$

□

A graph can represent more than one relationship with the same entities at once; a graph with this characteristic is called a **multirelational graph**. Formally, a graph $G = (V, E, T)$ is a multirelational graph if each edge e is defined as a tuple $e = (u, v, \tau)$, where $u, v \in V$ and $\tau \in T$ is the type of the relationship. A multirelational graph is encoded as $|T|$ adjacency matrices, each representing one type of relation. Said matrices can be combined into a single tensor, an **adjacency tensor** $A \in \mathbb{R}^{|V|\times|V|\times|T|}$.

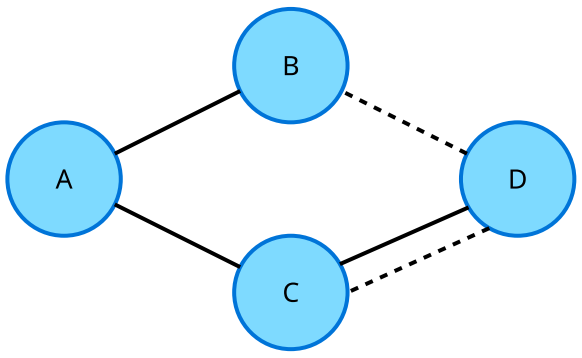
Exercise 1.4

Drugs can interact with each other, inducing certain side effects when taken together. Suppose that:

- Drug A induces headache when taken with drug B or with drug C, and vice versa;
- Drug B induces tachychardia when taken with drug D, and vice versa;
- Drug C induces both tachychardia and headache when taken with drug D and vice versa.

Represent this relationship in a multirelational graph.

Proof:



$$\begin{bmatrix} & A & B & C & D \\ A & 0 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 0 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} & A & B & C & D \\ A & 0 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 0 & 1 & 1 & 0 \end{bmatrix}$$

□

An **eterogeneous graph** is a multirelational graph where nodes have a type, partitioning the set of nodes. Formally, a (multirelational) graph $G = (V, E, T)$ is an eterogeneous graph where the set V is constructed as the union of n sets V_i, with $i \in \{1, ..., n\}$, such that:

$$V = \bigcup_i V_i \qquad\qquad V_i \cap V_j = \emptyset, \forall i, j \text{ when } i \neq j$$

An eterogeneous graph whose edges can only connect nodes of different types is called a **multipartite graph**. Formally, an eterogeneous graph $G = (V, E, T)$ is a multipartite graph if, for any edge $e = (u, v, \tau) \in E$, it follows that $u \in V_i, v \in V_j, i \neq j$.

Exercise 1.5

Certain illnesses are treated using certain drugs. Drugs can be incompatible with each other and influence the behaviour of certain proteins. Suppose that:

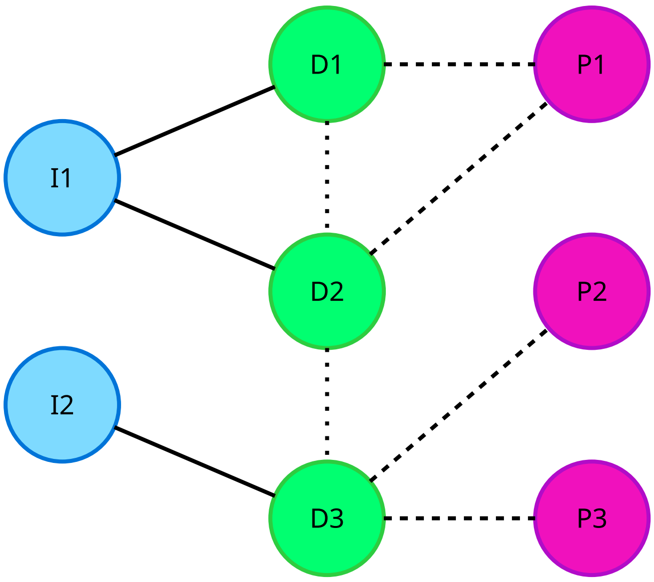
- Illness I_1 is treated using either drug D_1 or drug D_2;
- Illness I_2 is treated using drug D_3;
- Drug D_1 is incompatible with drug D_2 (and vice versa);
- Drug D_2 is incompatible with drug D_3 (and vice versa);
- Protein P_1 is influenced by drug D_1 and drug D_2;
- Protein P_2 is influenced by drug D_2;
- Protein P_3 is influenced by drug D_3.

Represent this relationship in a multirelational graph. Is it a multipartite graph?

Proof: No, because the relationship "being incompatible with" relates entities of the same type (drugs).

$$\begin{bmatrix} & I1 & I2 & D1 & D2 & D3 \\ I1 & 0 & 0 & 1 & 1 & 0 \\ I2 & 0 & 0 & 0 & 0 & 1 \\ D1 & 1 & 0 & 0 & 0 & 0 \\ D2 & 1 & 0 & 0 & 0 & 0 \\ D3 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} & D1 & D2 & D3 \\ D1 & 0 & 1 & 0 \\ D2 & 1 & 0 & 1 \\ D3 & 0 & 1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} & \text{D1} & \text{D2} & \text{D3} & \text{P1} & \text{P2} & \text{P3} \\ \text{D1} & 0 & 0 & 0 & 1 & 0 & 0 \\ \text{D2} & 0 & 0 & 0 & 1 & 0 & 0 \\ \text{D3} & 0 & 0 & 0 & 0 & 1 & 1 \\ \text{P1} & 1 & 1 & 0 & 0 & 0 & 0 \\ \text{P2} & 0 & 0 & 1 & 0 & 0 & 0 \\ \text{P3} & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$\square$

1.2 Graph embedding

When manipulating and comparing graphs, there is often interest in knowing whether two graphs are **similar**. The notion of graph similarity is muddy, since there is no set definition of what it means for two graphs to be similar.

In general, it is impractical to directly compute the similarity (however defined) between two graphs, because real graphs tend to have a very high amount of nodes. A better approach consists in retrieving a reasonable approximation of similarity by extracting features from the graph nodes.

The goal of these methods is to *encode* nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood. The idea is to **embed** nodes into a **latent space** where geometric relations between its elements correspond to relationships (that is, edges) in the original graph.

This model requires a pair of operation, an **encoder** and a **decoder**. An encoder is a function that maps each node in the graph into a low-dimensional vector, whereas a decoder is a function that takes the low-dimensional node embeddings and uses them to reconstruct information about each node's neighborhood in the original graph.

For this mapping to work it is necessary to assign a unique identifier to each node, so that the mapping is one-to-one. This can be done, given an arbitrary order of the nodes, by assigning an **indicator vector** to each node. The indicator vector $\boldsymbol{v}_i$ for a node $v \in V$ is a binary vector that has 0 in each component except a single component which is 1. In this way, each node has unique and unambiguous ID to be referred to.

Formally, the encoder is a function $\text{Enc} : V \to \mathbb{R}^d$ that maps nodes $v \in V$ to their respective vector embeddings $\boldsymbol{z}_v \in \mathbb{R}^d$, where d is the dimension of the latent space[1]. The easiest way to define this function is to adopt the **shallow embedding** approach, where the encoding function simply performs a "lookup" on a matrix that contains all embedding vectors. More formally:

$$\text{Enc}(v) = \boldsymbol{Z}[v] = \boldsymbol{Z}\boldsymbol{v}_i = \boldsymbol{Z}_{\boldsymbol{v}_i} \qquad \boldsymbol{Z} = \left(\left(\boldsymbol{z}_{v_1}\right) \; \left(\boldsymbol{z}_{v_2}\right) \; \left(\ldots\right) \; \left(\boldsymbol{z}_{v_{|v|}}\right) \right)$$

Where $\boldsymbol{Z} \in \mathbb{R}^{|V| \times d}$ is a matrix containing the embedding vectors for all nodes, $\boldsymbol{Z}[v]$ denotes the row of $\boldsymbol{Z}$ corresponding to node v and $\boldsymbol{v}_i$ is the indicator vector for v. Note how shallow embedding is simply a matrix multiplication.

The decoder's purpose is to reconstruct certain graph statistics from the node embeddings that are generated by the encoder. For example, given a node embedding $\boldsymbol{z}_v$ of a node v, the decoder might attempt to predict the neighborhood of v or its row $A[v]$ in the graph adjacency matrix.

[1]Realistic applications of encoders have latent spaces whose dimension ranges from roughly 16 to 1000.

The most common way to define a decoder is to define a **pairwise decoder**, a function $\mathrm{Dec} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ that has two encodings as input and a (predicted) non-zero value as output:

$$\mathrm{Dec}(\mathrm{Enc}(u), \mathrm{Enc}(v)) = \mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v])$$

Many modern graph embedding approaches such as `GraRep` and `HOPE` define decoding as the dot product between the encodings:

$$\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v]) = \boldsymbol{Z}[u]^T \boldsymbol{Z}[v]$$

Of course, it is not possible to simply define the decoder as the inverse of the encoder, because encoding entails a partial loss of information. Therefore, the idea is to define the decoder so that the difference between the "real" value of a similarity measure and its estimated value is as close as possible:

$$\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v]) \approx \boldsymbol{S}[u, v]$$

Where $\boldsymbol{S}[u, v]$ is a generic graph-based similarity measure between the nodes u and v and $\boldsymbol{S} \in \mathbb{R}^{|V| \times |V|}$ is the similarity matrix summarizing all pairwise node similarities.

To achieve this "closeness" to the real similarity measure, the standard practice is to minimize an empirical reconstruction loss L over a set of training node pairs D:

$$L = \sum_{(u,v) \in D} l(\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v]), \boldsymbol{S}[u, v])$$

Where l is any loss function measuring the discrepancy between the decoded (that is, estimated) similarity values $\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v])$ and the true values $\boldsymbol{S}[u, v]$.

The overall objective is to train the encoder and the decoder so that pairwise node relationships can be effectively reconstructed on the training set D. The choice of l depends on the definition of the decoder and the similarity function, and different graph embedding algorithms use different loss functions.

Algorithms such as `GraRep` and `HOPE` use *mean square error* as loss function, which would give:

$$L = \sum_{(u,v) \in D} ||\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v]) - \boldsymbol{S}[u, v]||_2^2$$

Other algorithms, such as `DeepWalk`, use cross entropy:

$$L = \sum_{(u,v) \in D} -\boldsymbol{S}[u, v] \log(\mathrm{Dec}(\boldsymbol{Z}[u], \boldsymbol{Z}[v]))$$

It is therefore necessary to give a definition of $\boldsymbol{S}[u, v]$. As stated before, there is no set definition of similarity, but many were proposed. The easiest one is given by counting the number of neighbors that are shared between the two nodes:

$$\boldsymbol{S}[u, v] := |N(u) \cap N(v)|$$

A more refined approach is given by considering general neighborhood overlap measures. One such example is the **Katz index**, obtained by counting the number of paths of any length between a pair of nodes and summing them together:

$$\boldsymbol{S}[u, v] := S_{\mathrm{Katz}}[u, v] = \sum_{i=1}^{\infty} \beta^i (\boldsymbol{A}[u, v])^i$$

Where β is a user-defined parameter that tunes how much weight is given to short versus long paths. A small value of β would give short paths more weights and long paths less weight, for example. The value of β must be strictly positive and must be less than the greatest eigenvalue of $\boldsymbol{A}$.

General neighborhood overlap measures are employed as the notion of similarity by graph embedding algorithms such as `HOPE`, whereas algorithms such as `GraRep` define $\boldsymbol{S}$ based on powers of the adjacency matrix.

Note that, even though easy to perform, shallow encoding suffers from some problems. Notably:

- Does not share parameters between nodes on the encoding;
- Does not preserve node features, only indexing;
- Is **transductive**: applies only to the nodes present during the training phase.

2 Reinforcement learning

2.1 Agents and environments

An **intelligent agent**, or simply **agent**, is any entity that is capable of **perceiving** the **environment** in which it finds itself through **sensors** and **influencing** said environment through **actions**. The environment is the section of the universe available for the agent to be perceived and acted upon.

The sequence of perceptions of an agent is the complete history of anything it has perceived. In general, what action an agent performs depends from its prior knowledge and/or from its previous perceptions. Formally, an agent's behaviour is described by a function $\mathrm{Pow}(P) \mapsto A$ that maps sequences of perceptions to actions.

An agent is said to be **rational** if it picks the best choice of actions. The notion of "best choice" commonly adopted in the field artificial intelligence is **consequentialism**: the agent's behavior is evaluated on the basis of the consequences of its actions. If an agent, in relation to a certain perception, performs an action that is deemed desirable for the user, then that agent has made the "best choice", and is then rational.

The notion of desirability is described by a **performance measure** that evaluates each sequence of states in which the environment is found. This measure allows an operational definition of a rational agent: for each possible sequence of perceptions, a rational agent will choose to perform the action that, based on previous perceptions and on its prior knowledge, maximises the performance measure.

Note how "rational" does not necessarely mean "omniscient", that is, able to know in advance the consequences of its actions. Indeed, an agent might not even know the entirety of the environment, or might have unreliable perceptions that might not match the environment analyzed. For these reasons, a rational agent should limit himself to performing actions that maximize *expected* performance.

The hereby given definition of rational agent, especially since the environment might be partially or totally unknown to it, does not exclude the possibility for it to **learn**: its initial configuration might be set, but it can be modified and improved with experience. In the case where the environment is entirely known a priori, the agent is said to be **autonomous**, since it needs no learning and merely performs the preset actions.

It is useful to classify the environments with respect to five informal metrics, in order to better approach the construction of agents:

- **Accessible** or **inaccessible**. The accessibility of an environment is a measure of how much information an agent is able to extract from it. An environment can be inaccessible due to limitations of the agent or be inherently so. Environments in the real world are always, have necessarily to some degree, inaccessible.
- **Deterministic** or **non-deterministic**. An environment is deterministic if the actions of the agent influence it in a predictable manner, that is if the result of the agent's actions always match the expected outcome. The physical world to be model always has some degree of nondeterminism.
- **Episodic** or **sequential**. In an episodic environment, an agent's experience can be divided into atomic steps where the choice of an action depends solely on current perception. In a sequential environment, the actions an agent takes may depend wholly or in part on which actions have been taken previously.
- **Static** or **dynamic**. An environment is static if it is affected only and exclusively by the agent's actions. If it can also change beyond the capabilities of the agent, then it is dynamic.
- **Discrete** or **continuous**. An environment is discrete if the number of states in which it can be is finite, that is, if it is possible (at least theoretically) to enumerate all its possible states, otherwise it is continuous. Since computers are discrete by definition, modeling a continuous environment through an automatic system will always require some degree of approximation.

Exercise 2.1

Suppose that the environment is the game of chess, the agent is the player (a human or a computer) and the actions are the possible moves. How would the environment be?

Proof:

- Accessible, because the agent has complete knowledge on the game state (the pieces on the board, their positions, ecc...)
- Deterministic, because a piece moved will always land on the predetermined square
- Sequential, because the allowed set of moves depends on the previous moves

- Static, because the state of the board can only change in response to a player's move
- Discrete, because the possible number of states is finite (albeit very large)

□

It is also possible to informally classify agents themselves, in four categories in order of complexity.

2.1.1 Simple-reflex agents

The simplest agents to construct are the **simple-reflex agents**. These agents have no model of the environment: the chosen action depends solely on the current perception and have no cognizance of previous perceptions.

Agents of this kind pick their action following **condition-action rules**: *if* a certain condition is met, *then* the action associated to that condition is performed.

A schematic representation of a simple-reflex agent is presented below. The function `INTERPRET-INPUT` generates an abstract description of the perception received by the agent, while the function `RULE-MATCH` returns the first action associated to that perception representation in the rule set `rules`.

Algorithm 2.2

```
rules <= set of condition-action rules

function SIMPLE-REFLEX-AGENT(percept)
  state <= INTERPRETER-INPUT(percept)
  rule <= RULE-MATCH(state, rules)
  action <= rule.action
  return action
```

Simple-reflex agents have limited intelligence. Indeed, agents of this kind operate on the predicament that the optimal action to be taken can be deduced entirely from its perceptions, i.e. if the environment is fully accessible. If this isn't the case or if the agent's a priori knowledge is faulty, the agent is bound to operate in a irrational manner.

An even more problematic situation that can arise with simple-reflex agents is infinite loops, since these agents are unable to detect them. The issue can only be ameliorated by introducing a degree of randomness on the chosen actions, so that the chance of applying the same action more than once in a row is reduced.

2.1.2 Reflex-based, model-based agents

The most efficient way to tackle the problem of having to deal with a partially accessible environment is to keep track of the section of the environment of which the agent has no knowledge. That is, the agent should be equipped with an **internal state** that depends on the perceptions it has previously picked up, so that it stores knowledge both about the current state and about other states. Agents of this kind are called **reflex-based, model-based agents**.

Periodically updating such an internal state requires the agent to know how the environment evolves over time, both in terms of how the agent's actions affect the environment and in terms of how the environment evolves on its own. This body of information goes by the name of **transition model**. In addition, it is necessary to have information regarding how the evolution of the environment is reflected on the agent's perceptions, collectively called **sensory model**.

A schematic representation of an agent with reflexes but based on a model is presented below, where the function `UPDATE-STATE` updates the agent's internal state before returning the action to be taken.

It should be noted how a reflex-based, model-based agent can hardly determine with certainty the state of the environment, and most likely has to rely on approximate descriptions.

Algorithm 2.3

```
state            <= the agent's current conception of the environment state
transition_model <= a description on how the next state depends on
                    the current state and action
sensor_model     <= a description on how the current world state is
                    reflected in the agent's percepts
rules            <= set of condition-action rules
action           <= the most recent action (starts NULL)

function MODEL-BASED-REFLEX-AGENT(percept)
  state <= UPDATE-STATE(state, action, percept, transition_model, sensor_model)
  rule <= RULE-MATCH(state, rules)
  action <= rule.action
  return action
```

2.1.3 Model-based, objective-based agents

There are situations in which the best action to be chosen also depends on some kind of long-term goal. This goal might not be reachable in the span of a single action, but could instead require many intermediate steps. In agents of this kind, the same action and the same internal state may result in different actions if the goal is different. Such agents are called **model-based, objective-based agents**.

2.1.4 Model-based, utility-guided agents

It is not always possible to construct a rational agent simply by pushing it to achieve a goal. Indeed, if that goal can be achieved through different sequences of actions, one might be preferable to another. Moreover, an agent might have to pursue several simultaneously incompatible goals, i.e., perform actions that “bring it closer” to one goal but at the same time “push it away” from another.

Instead of thinking about states exclusively in terms of “favorable” and “unfavorable”, a more nuanced approach involves introducing a measure of **utility**, which influences the agent’s choice in choosing which action to take (together with the performance measure, the goal to be followed, and from one’s own internal state).

The utility measure allows the agent to, in having to pursue several mutually incompatible goals, choose the action that gives the best tradeoff in advancing all of them. In addition, the structure of the environment does not always guarantee that a goal can be achieved with absolute certainty simply by performing the appropriate actions; even in this case, the utility measure allows one to assess how “convenient” it is for the agent to perform a certain action weighting the probability of actually achieving a goal if said action is undertaken.

2.1.5 Learning agents

The most interesting agents are undoubtedly those capable of **learning**; all agents presented so far can be constructed as learning agents. The considerable advantage is that they can operate in a completely unknown environment with little to no prior knowledge and learn from it, so that they can perform the best actions even in situations where even the designer cannot predict what this best action would be.

An agent capable of learning can be conceptually broken down into four components:

- The **learning component**, which is concerned with improving the agent’s performance;
- The **performance component**, which chooses which action to take based on perceptions and of the internal knowledge state (this component constituted the entire agent in previous models);
- The **critic**, which informs the learning component of what the agent is behaving optimally (rationally) based on a predetermined performance standard. This component is necessary because perceptions, by themselves, are not able to inform the agent about the optimality of its behavior;
- The **problem generator**, which suggests actions to the agent that may result in new and informative experiences. This component is necessary because if the agent relied exclusively on the performance component, it would always choose the best actions on the basis of his *current* knowledge, which is not necessarily complete. The problem generator can induce the agent to take actions that can be suboptimal in the short run but that can lead to even better outcomes in the long run.