Reinforcement Learning using Matlab

C. R. Koch / M. Shahbakhti

2022-9-18

4 L05-Reinforcement Learning

4.1 Concepts of RL and Deep RL

4.1.1 RL Background

Reinforcement Learning (RL) is

- one of the most exciting fields of Machine Learning today
- also one of the oldest since it has been around since the 1950s
- some interesting applications over the years
  - particularly in games (e.g., TD-Gammon, a Backgammon-playing program)
  - in machine control, but not as publicized in the common press

4.1.2 RL milestones in games

- in 2013, when researchers from a British startup called DeepMind demonstrated a system
  - that could learn to play just about any Atari game from scratch
  - eventually outperforming humans in most of them
  - using only raw pixels as inputs and without any prior knowledge of the rules of the games.
- This was the first of a series of successes, culminating in
  - March 2016 with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go
  - in May 2017 against Ke Jie, the world champion.
  - No program had ever come close to beating a master of this game, let alone the world champion.
- Today the whole field of RL is very active with new ideas for a wide range of applications.
- DeepMind was bought by Google for over $500 million in 2014.

4.1.3 RL Concept

In Reinforcement Learning

- a software agent makes observations
- takes actions within an environment

- in return it receives rewards.

Its objective is to learn to act in a way that will maximize its expected rewards over time.

- the algorithm, a software agent used to determine its actions, is called its policy.
- The policy could be a neural network taking observations as inputs
- with the action to take as the output
- This concept is schematically shown in Figure

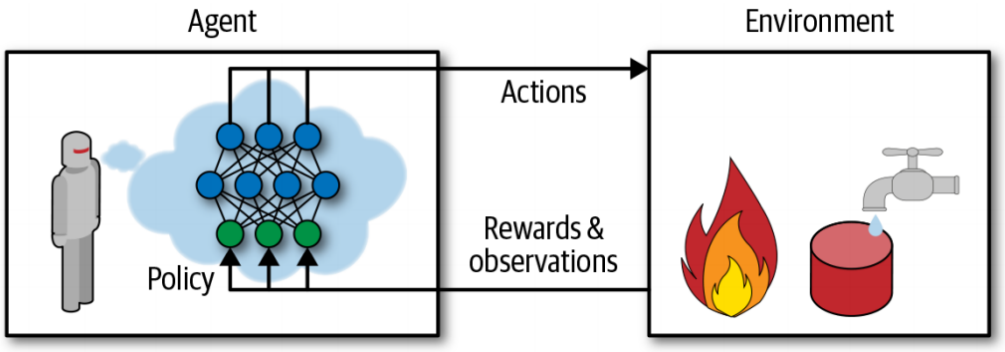


Figure 1: Reinforcement Learning using a neural network policy

4.1.4 RL Policy

The policy can be

- any algorithm you can think of
- it does not have to be deterministic
- in fact, in some cases it does not even have to observe the environment!

1. Example, a robotic vacuum cleaner
   - the reward is the amount of dust it picks up in 30 minutes
   - its policy could be
     - to move forward with some probability p every second
     - or randomly rotate left or right with probability $1 - p$.
     - The rotation angle would be a random angle between $-r$ and $+r$
   - since this policy involves some randomness, it is called a stochastic policy
   - the robot will have an erratic trajectory
     - which guarantees that it will eventually cover the entire floor to collect the dust.

4.1.5 Definition of RL component in context of control system (Based on Mathwork)

Many control problems encountered in areas such as robotics and automated driving require complex, nonlinear control architectures. Techniques such as gain scheduling, robust control, and nonlinear model predictive control (MPC) can be used for these problems, but often require significant domain expertise from the control engineer. For example, gains and parameters are difficult to tune. The resulting controllers can pose implementation challenges, such as the computational intensity of nonlinear MPC.

You can use deep neural networks, trained using reinforcement learning, to implement such complex controllers. These systems can be self-taught without intervention from an expert control engineer. Also, once the system is trained, you can deploy the reinforcement learning policy in a computationally efficient way. You can also use reinforcement learning to create an end-to-end controller that generates actions directly from raw data, such as images. This approach is attractive for video-intensive applications, such as automated driving, since you do not have to manually define and select image features.

- **Environment**: Everything that is not the controller — In the preceding diagram, the environment includes the plant, the reference signal, and the calculation of the error. In general, the environment can also include additional elements, such as:
  - Measurement noise
  - Disturbance signals
  - Filters
  - Analog-to-digital and digital-to-analog converters
- **Observation**: Any measurable value from the environment that is visible to the agent – In the preceding diagram, the controller can see the error signal from the environment. You can also create agents that observe, for example, the reference signal, measurement signal, and measurement signal rate of change.
- **Action**: Manipulated variables or control actions
- **Reward**: Function of the measurement, error signal, or some other performance metric — For example, you can implement reward functions that minimize the steady-state error while minimizing control effort. When control specifications such as cost and constraint functions are available, you can use generateRewardFunction to generate a reward function from an MPC object or model verification blocks. You can then use the generated reward function as a starting point for reward design, for example by changing the weights or penalty functions.
- **Learning Algorithm**: Adaptation mechanism of an adaptive controller

4.1.6 Environment and observations

- A formal definition of the environment is everything that is not the controller.
- The environment can be
  - a dynamic model of the system
  - or a real-time experimental system
- an RL agent interfaces with the environment by sending action based on the environment observations.

4.1.7 Reward function

Defining the reward function plays

- a central role in the RL learning process,
- this is the metric of the agent's success in doing the task.

1. During the learning process,
    - the agent is updated based on the rewards received for different state-action combinations.
    - the reward can be defined as a continuous function,
        - such as Quadratic Regulator (QR) cost function, where the highest reward is zero,
        - or as a discrete function, where reward varies discontinuously with changes in the environment.
    - typically a combination of reward functions are used
        - a continuous function is used to minimize the error,
        - a discontinuous function is used to penalize violation of defined constraints (negative reward)
        - or to encourage RL to do specific side tasks such as removing steady-state error (positive reward).

4.1.8 Reinforcement Learning Workflow

1. **Formulate problem**: Define the task for the agent to learn, including how the agent interacts with the environment and any primary and secondary goals the agent must achieve.
2. **Create environment**: Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model.
3. **Specify reward**: Specify the reward signal that the agent uses to measure its performance against the task goals and how to calculate this signal from the environment.
4. **Create agent**: Create the agent, which includes defining a policy approximator (actor) an value function approximator (critic) and configuring the agent learning algorithm.
5. **Train agent**: Train the agent approximators using the defined environment, reward, and agent learning algorithm.
6. **Validate agent**: Evaluate the performance of the trained agent by simulating the agent and environment together
7. **Deploy policy**: Deploy the trained policy approximator using, for example, generated GPU code.

4.1.9 RL Agent structure

A policy and a learning algorithm are the two primary components of an RL agent.

1. The policy
    - correlates the received observation to the selected actions by using a tunable function approximator.
    - table or deep neural network can represent this function approximator.

- The policy learning algorithm is used to
  - continuously update the policy approximator parameters
  - by interfacing with the environment and collected reward.

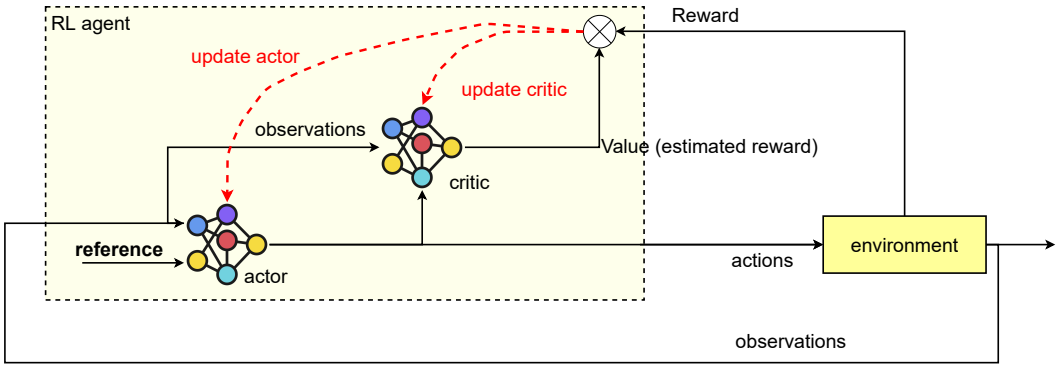
2. The algorithm
   - aims to find an optimal policy that makes the RL agent collect maximum cumulative rewards during a specific task.

4.2 Reinforcement Learning Agents

4.2.1 Actor Critic RL

Based on the type of algorithm,

- an agent maintains one or two function approximators.
- In general, there are two kinds of function approximators: critic and actor.
- Schematics of Actor critic RL is shown in the Figure



4.2.2 The critic

which represents the value function,

- can estimate the expected long-term reward either based on observation only $V(s|\theta_V)$

4.2.3 Actors

which represent the policy function,

- are able to select an action based on an observation,
- which is represented by the function $\mu(S|\theta_\mu)$.
- Each function approximator has parameters (θ),
- which are computed throughout the learning process.

4.2.4 Function Approximators

Different function approximators such as

- using tabular or linear function approximation methods are often used in RL applications.
- However, RL can obtain impressive performance in the same application by implementation

- a nonlinear function approximator using multi-layer Artificial Neural Networks (ANNs).

Deep Neural Network (DNN) can be used to represent both critic and actors

- where DNN is constructed by
  - connecting a series of layers for each input path (observation or actions) and
  - each output path (estimated rewards or actions) [**sutton2018reinforcement**].
- The actor also
  - updates itself with the response from the critic so that
  - it can adjust its probabilities of taking that action again in the future.
- In this way, the policy now ascends the reward slope in the direction

that the critic recommends rather than using the rewards directly.

4.3 RL algorithms

4.3.1 Different RL algorithms

- **Indirect policy representation (off-policy)**: Agents that use only critics to select their actions rely on an indirect policy representation. These agents are also referred to as value-based, and they use an approximator to represent a value function or Q-value function. In general, these agents work better with discrete action spaces but can become computationally expensive for continuous action spaces.
- **Direct policy representation (on-policy)**: Agents that use only actors to select their actions rely on a direct policy representation. These agents are also referred to as policy-based. The policy can be either deterministic or stochastic. In general, these agents are simpler and can handle continuous action spaces, though the training algorithm can be sensitive to noisy measurement and can converge on local minima.
- **Actor-critic agents (off-policy)**: Agents that use both an actor and a critic are referred to as actor-critic agents. In these agents, during training, the actor learns the best action to take using feedback from the critic (instead of using the reward directly). At the same time, the critic learns the value function from the rewards so that it can properly criticize the actor. In general, these agents can handle both discrete and continuous action spaces.

Well-known agents:

- Q-learning Agents– value-based – Discrete action space
- Deep Q-network Agents (DQN)– value-based – Discrete action space
- SARSA agents– value-based – Discrete action space
- Policy Gradient (PG)– policy-based – Discrete or continuous action space
- Deep Deterministic Policy Gradient (DDPG)– actor-critic – Continuous

Among these methods, DDPG and DQN are usually used in engineering application and we will go through it here in this course.

4.4 Deep Q-network Agents (DQN) algorithm

4.4.1 Structure of DQN algorithm

The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards.

DQN agents can be trained in environments with the continuous or discrete observation with only discrete action spaces.

DQN has a Q-value function critic $Q(S, A)$, but it has no actor. During training, the agent:

- Updates the critic properties at each time step during learning.
- Explores the action space using epsilon-greedy exploration. During each control interval, the agent either selects a random action with probability ϵ or selects an action greedily with respect to the value function with probability $1 - \epsilon$. This greedy action is the action for which the value function is greatest.
- Stores past experiences using a circular experience buffer. The agent updates the critic based on a mini-batch of experiences randomly sampled from the buffer.

To estimate the value function, a DQN agent maintains two function approximators:

- Critic $Q(S, A; \phi)$ — The critic, with parameters ϕ, takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q_t(S, A; \phi_t)$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters ϕ_t using the latest critic parameter values.

both $Q(S, A; \phi)$ and $Q_t(S, A; \phi_t)$ have the same structure and parameterization.

4.4.2 DQN training algorithm

DQN agents use the following training algorithm, in which they update their critic model at each time step.

- Initialize the critic $Q(S, A; \phi)$ with random parameter values ϕ, and initialize the target critic parameters ϕ_t with the same values.$\phi_t = \phi$.

For each training time step:

1. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest:

$$\begin{cases} A = \text{random action} & \text{if } n < \epsilon \\ A = \underset{A}{\text{argmax}} \ Q(S, A; \phi) & \text{Otherwise} \end{cases} \tag{4.1}$$

   where n is uniform random number.

2. Execute action A. Observe the reward R and next observation S'.
3. Store the experience (S, A, R, S') in the experience buffer.
4. Sample a random mini-batch of M experiences (S_i, A_i, R_i, S'_i) from the experience buffer.
5. If S'_i is a terminal state, set the value function target y_i to R_i (y_i value function target or reference and R_i is collected reward). Otherwise, set it to

$$y_i = R_i + \gamma \underset{A'}{\text{max}} \ Q_t(S'_i, A'; \phi_t) \tag{4.2}$$

   where γ is the discount factor

6. Update the critic parameters by one-step minimization of the loss L across all sampled experiences

$$L = \frac{1}{M}\sum_{i=1}^{M}(y_i - Q(S_i, A; \phi))^2 \tag{4.3}$$

where M is mini-batch of experiences size

7. Update the target critic parameters as

$$\phi_t = \tau\phi + (1-\tau)\phi_t \tag{4.4}$$

where τ is smoothing factor.

8. Update the probability threshold ϵ for selecting a random action based on the decay

$$\epsilon = \epsilon \;\; r \tag{4.5}$$

where r is decay rate. With this more flexible choice to end at the very small exploration probability, after the training process will focus more on exploitation (i.e., greedy by reducing ϵ) while it still can explore with a very small probability when the policy is approximately converged. r usually set as very small value such as 0.01.

4.5 Deep Deterministic Policy Gradient Agents (DDPG) algorithms

- The DDPG is a model-free RL algorithm
  - where an actor-critic RL agent calculates an optimal policy by maximizing the long-term reward.
- The main difference between Deterministic Policy Gradient (DPG) and DDPG is
  - using neural network as an approximator in DDPG to learn for huge state and action pairs.
- due to use of DNN, DDPG algorithms are referred to Deep DPG, as shown in Algorithm 1 [**lillicrap2015continuous**] (see below).

4.5.1 DDPG training algorithm

During training

- the actor and critic are updated by the DDPG agent at each time sample
- the agent stores past experiment using an experience buffer
- the actor and critic are then updated
  - using a mini-batch of those experiences randomly sampled from the buffer.
- the policy's selected action
  - is perturbed using a stochastic noise model at each training step [**MalabRL**].

4.5.2 Actor and Critic Functions

To estimate the policy and value function, a DDPG agent maintains four function approximators:

- Actor $\pi(S, \theta)$— The actor, with parameters θ, takes observation S and returns the corresponding action that maximizes the long-term reward.
- Target actor $\pi(S, \theta_t)$ — To improve the stability of the optimization, the agent periodically updates the target actor parameters θ_t using the latest actor parameter values.

- Critic $Q(S, A; \phi)$ — The critic, with parameters ϕ, takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.
- Target critic $Q_t(S, A; \phi_t)$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters ϕ_t using the latest critic parameter values.

Both $\pi(S, \theta)$ and $\pi_t(S, \theta_t)$ have the same structure and parameterization, and both $Q(S, A; \phi)$ and $Q_t(S, A; \phi_t)$ have the same structure and parameterization.

4.5.3 DDPG Training Algorithm

- Initialize the critic $Q(S, A; \phi)$ with random parameter values ϕ, and initialize the target critic parameters ϕ_t with the same values: $\phi = \phi_t$

- Initialize the actor $\pi(S, \theta)$ with random parameter values θ, and initialize the target actor parameters θ_t with the same values: $\theta = \theta_t$

For each training time step:

1. For the current observation S, select action $A = \pi(S; \theta) + N$, where N is stochastic noise from the noise model.
2. Execute action A. Observe the reward R and next observation S'.
3. Store the experience (S, A, R, S') in the experience buffer.
4. Sample a random mini-batch of M experiences (S_i, A_i, R_i, S'_i) from the experience buffer.
5. If S'_i is a terminal state, set the value function target y_i to R_i (y_i value function target or reference and R_i is collected reward). Otherwise, set it to

$$y_i = R_i + \gamma \, Q_t(S'_i, \pi_t(S'_i; \theta_t); \phi_t) \tag{4.6}$$

   where γ is the discount factor. The value function target is the sum of the experience reward R_i and the discounted future reward. To compute the cumulative reward, the agent first computes a next action by passing the next observation S'_i from the sampled experience to the target actor. The agent finds the cumulative reward by passing the next action to the target critic.
6. Update the critic parameters by one-step minimization of the loss L across all sampled experiences

$$L = \frac{1}{M} \sum_{i=1}^{M} (y_i - Q(S_i, A_i; \phi))^2 \tag{4.7}$$

   where M is total experience sampled.
7. Update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward.

$$\begin{aligned} \nabla_\theta &= \frac{1}{M} \sum_{i=1}^{M} G_{ai} G_{\pi i} \\ G_{ai} &= \nabla_A Q(S_i, A; \phi) \text{ where } A = \pi(S_i, A; \theta) \\ G_{\pi i} &= \nabla_\theta \pi(S_i, A; \theta) \end{aligned} \tag{4.8}$$

   Here, G_{ai} is the gradient of the critic output with respect to the action computed by the actor network, and $G_{\pi i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

8. Update the target actor and critic parameters as

$$\phi_t = \tau\phi + (1-\tau)\phi_t \tag{4.9}$$

$$\theta_t = \tau\theta + (1-\tau)\theta_t \tag{4.10}$$

where τ is smoothing factor.

4.6 Hands-on example of DDPG and DQN agent in Matlab: Swing Up and Balance Pendulum

This example shows how to train a deep Q-learning network (DQN) and Deep Deterministic Policy Gradient (DDPG) agent to swing up and balance a pendulum modeled in Simulink. The dynamics equation of pendulum is

$$(l + ml^2)\ddot{\theta} + c\dot{\theta} - mgl\sin(\theta) = \tau \tag{4.11}$$

where l is length of pendulum, m is mass, c is damper, and τ is actuator torque.

4.6.1 Swing Up and Balance Pendulum control problem

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

- The upward balanced pendulum position is 0 radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the sine of the pendulum angle, the cosine of the pendulum angle, and the pendulum angle derivative.
- The reward r_t, provided at every timestep, is

$$r_t = -\left({\theta_t}^2 + 0.1{\dot{\theta}_t}^2 + 0.001u_{t-1}^2\right) \tag{4.12}$$

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

4.6.2 DQN and DDPG training

main program for DQN: `SimulinkPendulumSwingupDQNExample.m`
main program for DDPG: `SimulinkPendulumSwingupDDPGExample.m`

4.7 Hands-on example of DDPG agent in Matlab: Adaptive Cruise Control

4.7.1 Matlab Hands-on example

- main program `TrainDDPGAgentForACCExample.m`

4.7.2 Details of Adaptive Cruise Control example:

- a vehicle (ego car) is equipped with Adaptive Cruise Control (ACC)

1. Sensor
    - has a sensor, such as radar
        - measures the distance to the preceding vehicle in the same lane (lead car), D_{rel}.
        - also measures the relative velocity of the lead car, V_{rel}.

4.7.3 The ACC system operates in the following two modes:

Speed control ego car travels at a driver-set speed.

Spacing control ego car maintains a safe distance from the lead car.

The ACC system decides which mode to use based on real-time radar measurements, as

- if the lead car is too close, the ACC system switches from speed control to spacing control.
- if the lead car is further away, the ACC system switches from spacing control to speed control.

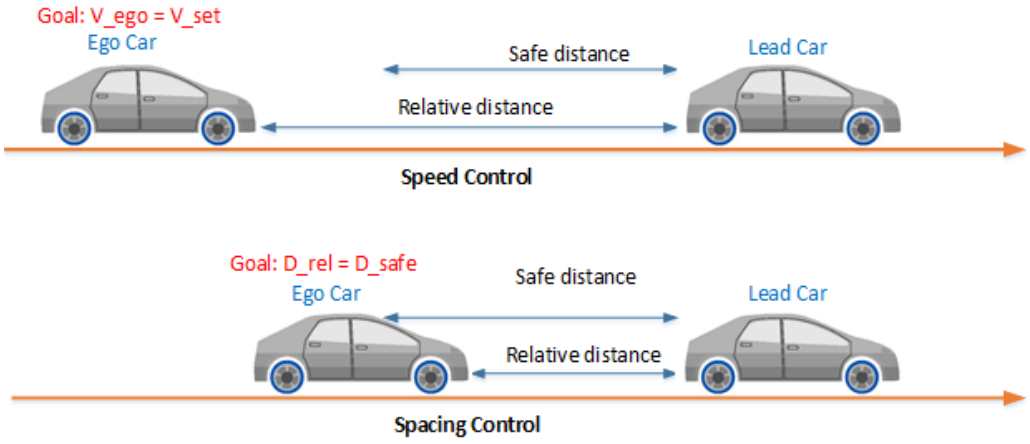
the ACC system then travels at a driver-set speed if it can maintains a safe distance.

Rules - pseudo code

```
if D_rel >= D_safe
  then speed control mode ACTIVE
% control goal is to track the driver-set velocity, V_set

if D_rel < D_safe
  then spacing control mode is ACTIVE
% The control goal is to maintain the safe distance, D_safe
```

The schematic of ACC problem is shown in the Figure below



4.7.4 Simple car model for both the ego vehicle and the lead vehicle

- the dynamics between acceleration command (force) and velocity are modeled as:

$$G = \frac{1}{s(0.5s+1)}$$

- which approximates the dynamics of the throttle body and vehicle inertia
- the acceleration action signal from the agent to the environment is from -3 to 2 m/s^2.

1. The reference velocity for the ego car V_{ref} is defined as follows:
   - if the relative distance d_{rd} is less than the safe distance d_{sd}
     - the ego car tracks the minimum of the lead car velocity and driver-set velocity
     - thus in this case the ego car maintains safe distance from the lead car $d = d_{sd}$
   - If the relative distance is greater than the safe distance
     - the ego car tracks the driver-set velocity.

4.7.5 Safe Distance as a linear function of speed

- the safe distance $d_{sd}$is defined as
  - a linear function of the ego car longitudinal velocity V plus an offset D_{default}
  - so, $d_{sd} = t_{gap} \times V + d_{\text{default}}$.
  - The safe distance determines the reference tracking velocity for the ego car.
- the observations from the environment are:
  - the velocity error $e = V_{ref} - V_{ego}$,
  - its integral $\int_\tau e d\tau$,
  - and the ego car longitudinal velocity V.
- The simulation is terminated when
  - longitudinal velocity of the ego car is less than $V < 0$,
  - or the relative distance between the lead car and ego car becomes less than $d_{sd} < 0$.

4.7.6 The reward function

- r_t, is provided at every time step t and for this example is defined as

$$r_t = -(0.1e_t^2 + u_{t-1}^2) + M_t$$

- where u_{t-1} is the control input from the previous time step
- e_t is the error at time t
- M_t is a logical value (0,1) with
  - $M_t = 1$ if velocity error $e_t^2 \geq 0.25$
  - otherwise, $M_t = 0$.

4.8 References

[1] Reinforcement Learning Toolbox: User's Guide: available at `https://www.mathworks.com/help/pdf_doc/reinforcement-learning/rl_ug.pdf`