Computers and Chemical Engineering 139 (2020) 106886

Contents lists available at ScienceDirect

Computers and Chemical Engineering

journal homepage: www.elsevier.com/locate/compchemeng

A review On reinforcement learning: Introduction and applications in industrial process control

Rui Nian, Jinfeng Liu*, Biao Huang

Department of Chemical and Materials Engineering, University of Alberta, Edmonton, Alberta, Canada

ARTICLE INFO

Article history:
Received 16 January 2020
Revised 10 April 2020
Accepted 22 April 2020
Available online 28 April 2020

Keywords:
Reinforcement learning
Model predictive control
Optimal control
Machine learning
Process industry
Process control

ABSTRACT

In recent years, reinforcement learning (RL) has attracted significant attention from both industry and academia due to its success in solving some complex problems. This paper provides an overview of RL along with tutorials for practitioners who are interested in implementing RL solutions into process control applications. The paper starts by providing an introduction to different reinforcement learning algorithms. Then, recent successes of RL applications across different industries will be explored, with more emphasis on process control applications. A detailed RL implementation example will also be shown. Afterwards, RL will be compared with traditional optimal control methods, in terms of stability and computational complexity among other factors, and the current shortcomings of RL will be introduced. This paper is concluded with a summary of RL's potential advantages and disadvantages.

1. Introduction

Artificial intelligence (AI) has recently triggered a paradigm shift in numerous industries around the world, ranging from technology to health care. The previously arcane topic is now an insuppressable wildfire igniting countless industrial and academic minds alike. Rapid advancements in computer hardware and ever-cheapening data storage combined with AI's ability to 'self-learn' has pushed AI to become the forefront algorithm for many applications such as computer vision and natural language processing. According to PwC (2019), AI is projected to generate over 15 trillion USD to the world economy while providing a 26% boost in GDP by 2030. Overall, AI is a massive field encompassing many goals.

The major goals/topics of AI are shown in Fig. 1. Currently, the most influential topic in AI is machine learning (ML). ML can be described as the scientific field that studies and develops algorithms and statistical models to give machines the explicit ability to learn tasks without being programmed to do so (Russel and Norvig, 2009). The ML field can be further decomposed into supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

The breakdown of the ML field is shown in Fig. 2. In supervised learning, the agent (ML algorithm) learns the input-output mapping (model) from a training data set labeled by subject matter expert(s) (Ng, 2018). A supervised learning algorithm generally attempts to generalize across training examples and uses this knowledge to predict labels for unseen data. Note that not all labels are guaranteed to be correct. In the process industry, the subject matter expert is often a sensor measuring the current state (temperature, pressure, etc.) of a process and could be unreliable and noisy. Ultimately, the performance of the supervised learning agent cannot outperform the subject matter expert or supervisor because the agent essentially only mimics the labeling behavior of the expert. In literature, this performance limit of the agent is known as the Bayes error rate (Ng, 2018). Unsupervised learning is typically used for identifying hidden structures within unlabeled data sets. Examples include segregating data based on their similarity to identify different operating regimes, or identifying the principal components within a data set (Hinton and Sejnowski, 1999; Sutton and Barto, 2018). The three main goals of unsupervised learning are for dimensional reduction, feature extraction, and clustering. Semi-supervised learning is obtained by combining ideas from supervised and unsupervised learning. In the process industry, manually labeling data is a costly endeavor; however, many applications such as fault detection require labeled data sets to materialize useful applications. Here, semi-supervised learning can be applied to learn from the small amount of labeled data and extract additional useful insights from remaining unlabeled data (Ge et al., 2017). Nevertheless, semi-supervised learning still suffers from the inability to surpass the supervisor. Therefore, the previous methods can only create value through cost reductions while failing to

* Corresponding author.
E-mail addresses: rnian@ualberta.ca (R. Nian), jinfeng@ualberta.ca (J. Liu), bhuang@ualberta.ca (B. Huang).

https://doi.org/10.1016/j.compchemeng.2020.106886
0098-1354/

Nomenclature

a	Anomalous state
$\mathcal{A}$	The set of anomalous states
b	Behaviour policy
$\mathbb{E}$	Expectation
H	Control horizon
k	Update step
K_d	Derivative gain
K_i	Integral gain
K_p	Proportional gain
N	Arbitrary number of time steps
$\mathcal{N}$	Ornstein-Uhlenbeck exploratory noise
o	Observation
$\mathcal{O}$	The set of possible observations, observation space
p	Probability transition function
$\mathbf{p}$	Action probability vector
Pr	Probability
q^*	Action-value
Q	Estimated action-value
Q_{mpc}	Tuning matrix for the states MPC
r	Expected reward
R	Sampled reward
R_{mpc}	Tuning matrix for the inputs in MPC
t	Time step
u	Control action
u'	Noise corrupted input signal
u^*	Optimal control action
$\mathcal{U}$	The set of all possible control actions, action space
v	Expected value
V	Estimated value
w	Weight vector for function approximation
W_t	Wiener process
x	State
y	Predicted variable
$\mathcal{X}$	The set of possible states, state space
γ	Discount factor
π	Policy
α	Learning rate or step size
β	Fixed discount factor in Semi-MDPs
η	Time spent in a particular state
μ	State distribution
μ_a	Threshold for system to be identified as anomalous in RL anomaly detection
ϵ	Percent chance of performing a random action in TD methods
θ	Ornstein-Uhlenbeck hyper parameter for the time parameter
σ	Ornstein-Uhlenbeck hyper parameter for the Wiener process

Abbreviations

ADP	Approximate dynamic programming
AI	Artificial intelligence
CMDP	Constrained Markov decision process
CSTR	Continuously stirred tank reactor
DCS	Distributed control system
DPG	Deterministic policy gradient
DDPG	Deep deterministic policy gradient
DMC	Dynamic matrix control
DP	Dynamic programming
DQN	Deep q-learning network
EMPC	Economic model predictive control
FOMDP	Fully observable Markov decision process
FTC	Fault tolerant control
HARL	Heuristically accelerated reinforcement learning
LQG	Linear quadratic Gaussian
LQR	Linear quadratic regulator
LSTM	Long short term memory
MC	Monte Carlo
MDP	Markov decision process
MIMO	Multiple-input multiple-output
ML	Machine learning
MP	Mathematical programming
MPC	Model predictive control
MRP	Markov reward process
MSE	Mean squared (tracking) error
NP	Non-deterministic polynomial time
OPC	Open platform communication
P	Pressure
PID	Proportional-Integral-Derivative
POMDP	Partially observable Markov decision process
PPO	Proximal policy optimization
PUE	Power usage effectiveness (used by Google to quantify energy efficiency)
RL	Reinforcement learning
RNN	Recurrent neural network
RTO	Real time optimization
SGD	Stochastic gradient descent
SISO	Single-input single-output
SMDP	Semi Markov decision process
SP	Set-point
TD	Temporal difference
TPU	Tensor processing unit
TRPO	Trust region policy optimization

expand the current capabilities of modern methods. An intuitive example is as follows:

> World renowned chemists may have the capability to achieve 95% purity in chemical A using state-of-the-art methods. With the aid of supervised learning agents to replicate trivial tasks, the synthesization process may be faster and/or cheaper; however, the purity will not increase beyond 95% because the agent is simply replicating the supervisor. In other words, the current knowledge base of chemistry for synthesization of a higher purity chemical A has not changed.

Reinforcement learning (RL) attempts to overcome the above mentioned performance limit by combining previous ML fields into a unifying algorithm with modifications on the learning process. Ultimately, the goal of RL is to give machines the ability to surpass all known methods. Specifically, the goal of the RL agent is to push the boundaries of what is currently possible through learning the optimal mapping of situations to actions (called policy) through a trial-and-error search guided by a scalar reward signal. In many challenging scenarios, actions affect not only the immediate reward, but also all subsequent rewards. These two features – guided trial-and-error search and delayed feedback – distinguish RL from all other topics of ML and ultimately enables the ability to push the current boundaries of knowledge (Sutton and Barto, 2018).

Nevertheless, this characteristic introduces unique challenges to the RL agent, one being the trade-off between **exploration** and **exploitation**. The goal of the agent is to maximize the reward signal; however, the agent is initialized *tabula rasa* (i.e., a clean state). Thus, the agent must first explore the state space to identify the optimal actions. Furthermore, for stochastic systems or systems with delayed reward signals, each state must be visited many times to obtain reliable information. Ultimately, the agent must ex-

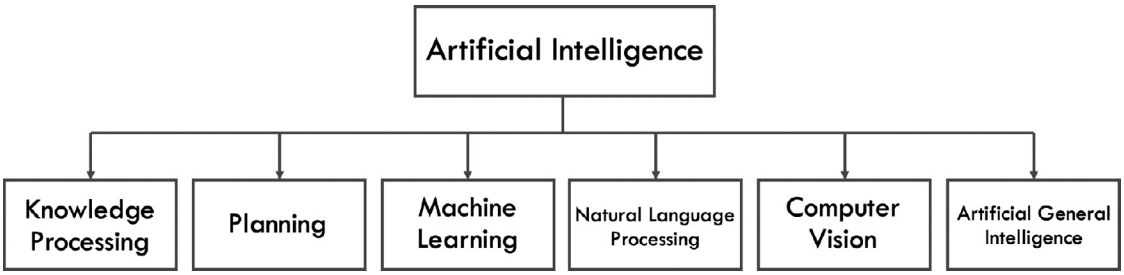


Fig. 1. The major goals of artificial intelligence.

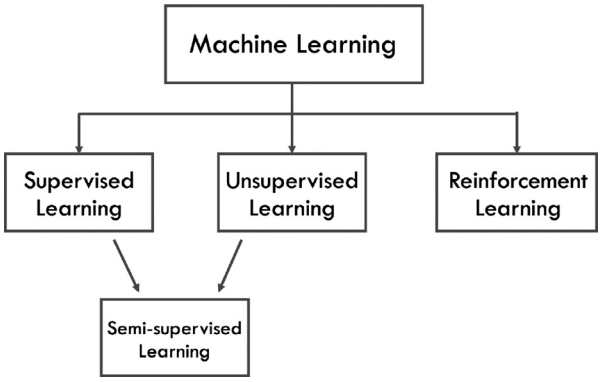


Fig. 2. The sub-components of machine learning.

ploit the current information to maximize rewards. However, exploiting too soon leads to locally optimal or sub-optimal actions. Likewise, exploiting too late results in forgone rewards. The problem is further complicated by non-stationary or time-varying systems, where exploration is a requirement for the continued optimality of the agent. In control, this dilemma is known as identification (or estimation) versus control.

Mathematically, RL is formulated as an optimal sequential decision making algorithm with the ability to account for stochasticity within the system. Furthermore, RL was partly built upon stochastic optimal control which may provide advantages in certain systems compared to modern control techniques (Rawlik et al., 2013). Traditional optimal control methods (e.g., model predictive control (MPC)) typically employ mathematical programming (MP) based trajectory optimization methods. The successes of such methods in addressing multi-stage optimal control problems are widely demonstrated; however, industrial applications of such methods in large-scale stochastic multiple-input multiple-out (MIMO) problems are still limited due to their online computational requirements (Maravelias and Sung, 2009). Furthermore, the solutions to systems with uncertainty typically use stochastic programming with only a finite number of uncertainty scenarios and assume that the information regarding the uncertainty is known. Unfortunately in practice, uncertainty information is typically unknown, non-stationary and contain uncertainty themselves (Shin et al., 2019). Moreover, the control horizon of MP methods for large MIMO systems is generally short to ensure computational feasibility, although the identified optimal solution for short horizons might be highly sub-optimal in the long term (Mayne and Rawlings, 2017).

Compared to MP control methods, RL overcomes the issue of long online computation times by pre-computing the optimal solutions offline, a concept similar to parametric programming in explicit model predictive control (Bemporad et al., 2002). This characteristic of RL is advantageous for systems where online computation time is of importance. However, for complex systems, offline training of the RL agent may require hundreds of thousands of steps to achieve even a near-optimal policy; thus, making the training step infeasible in live operations. To overcome this issue, the agent may be first trained in a process simulator to obtain general knowledge of the process. The performance of the agent after this phase will be strongly correlated with the fidelity of the simulator. It is expected that the RL agent's ultimate performance during such a training process will not outperform the performance of a corresponding MPC designed based on the same simulator model if global optimality is ensured in obtaining the MPC's solution. When compared to advanced control, RL is similar to the widely used combination of real-time optimization (RTO) and MPC. In traditional optimal control, RTO provides the optimal steady state set-point(s) and MPC identifies the optimal input trajectory to achieve the desired set-points. For RL, the learned optimal policy implicitly carries information regarding the optimal set-point(s) and the optimal input(s) to reach the set-points. This is indeed very similar to the concept of economic MPC which aims to combine the RTO and MPC layers (Ellis et al., 2014). Due to these unique features, it is a natural curiosity to explore the areas where RL may have potential in the process control industry.

In the literature, there exists some review papers focusing on ML and process control such as Shin et al. (2019) and Lee et al. (2018). In these existing review papers, the target audience is catered towards academic researchers and typically involve more rigorous mathematics. The objectives of this paper are to introduce the motivations and concepts of RL in tutorial way to potential practitioners. A general overview of all branches and state-of-the-art RL algorithms will be briefly explored. The focus of this study is on the potential implementation and value creation of RL in the process control industry, not on rigorous mathematical proofs and numerical studies of RL methods compared to traditional process control. This review starts with an introduction to RL, the Markov decision process (MDP) and different families of RL methods. Concepts explored will also be intuitively correlated to process control ideas to enhance understanding. Section 3 compares RL qualitatively to traditional control methods and also introduces some of successful RL applications. A quantitative example where RL was applied onto an industrial pumping system will also be shown here to provide additional intuition. The section is concluded with RL implementation techniques catered towards the process industry. Section 4 presents the current weaknesses that may be preventing RL from being adopted in process industry. Finally, the review is concluded in Section 5 with the advantages and disadvantages of RL summarized.

2. Reinforcement learning

2.1. A brief history

RL originated from two main fields of research: *optimal control* using value functions and dynamic programming and *animal psychology* inspiring trial-and-error search. The optimal control problem was originally proposed to design a controller to minimize the loss function of a dynamical system over time (Mayne and Rawlings, 2017). In the mid 1950s, Richard Bellman extended the works of Hamilton and Jacobi and developed an approach to solve the

Table 1
From left to right, the evolution of reinforcement learning.

k-armed bandits	Contextual bandits	Reinforcement learning
Optimal action	Optimal action	Optimal action
One situation	Many situations	Many situations
Immediate conseq.	Immediate conseq.	Long-term conseq.

optimal control problem. This approach, now called dynamic programming, optimizes the input trajectory by using the functional equation (a function where the unknowns are also functions) generated from the system's state information in conjunction with a value function (Bellman, 1957a). The functional equation, known as the Bellman equation, is given as:

$$V(x) = r(x) + \gamma \sum P(x'|x, u) \cdot V(x') \tag{1}$$

where $V(x)$ is the value function in state x, γ is a discount factor to incorporate uncertainty into future rewards, $r(x)$ is the reward obtained as a function of the system's behaviour with respect to a desired performance, $P(x'|x, u)$ is the transitional probability of arriving at the next state x' given current state and control input x and u, respectively, $V(x')$ is the value function at the next state x'. Intuitively, the value function can be understood as the goodness of being in a particular state, assuming optimal behaviour thereafter; high values correspond to good states and low values for bad states. Unfortunately, dynamic programming suffers from the *curse of dimensionality* (i.e., the computational cost grows exponentially with the number of states). A significant step in RL literature was made when approximate dynamic programming (ADP) methods were developed to overcome this obstacle (Mes and Rivera, 2017). Reinforcement learning leverages one such ADP method to solve for the optimal policy offline. The design of the ADP may take many forms dependent on the objective of the agent. For example, Lee and Wong (2010) found that post-decision-state formulation of ADPs offers the most benefit to process control problems where the main objectives are safety and economics. More details can be found in Lee and Wong (2010). The concept of a reward/punishment trial-and-error learning system in RL originated from *animal psychology*. More specifically, the original concept was proposed in Thorndike and was named the *Law of Effect*, stating that actions resulting in good outcomes are likely to be repeated, while actions with bad outcomes are muted. Initially, the agent undergoes a trial-and-error search to identify the outcomes corresponding to each action, then only repeating the good outcome actions thereafter. Through the combination of dynamic programming from optimal control and trial-and-error search from animal psychology, the modern field of RL was developed. For additional details regarding the history of reinforcement learning, please refer to Sutton and Barto (2018).

2.2. The bandit problem

The evolution of RL is shown in Table 1. Reinforcement learning takes its roots from the *k*-armed bandit problem where the agent is only concerned with making the optimal decisions in one situation. Additionally, only the immediate consequences were considered. Eventually, the concept was extended in the early 1980s by Barto et al. (1981) to solve multi-situation systems. This new problem was named contextual bandits (also named associative search). Here, the agent was still only concerned with the immediate consequences. However, real world problems can rarely be solved by considering only the immediate consequences. In most scenarios, near term sacrifices are absolutely necessary to reach long term success. To overcome this dilemma, RL was developed to find the optimal decisions in multi-situation systems that optimized not only the immediate rewards, but also the trajectory thereforth.

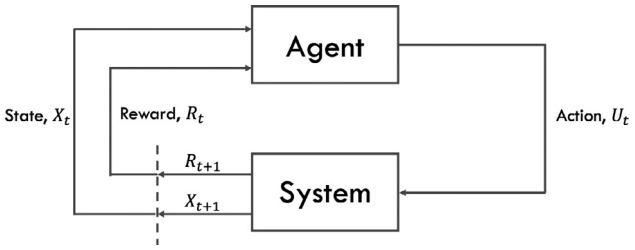


Fig. 3. The general Markov decision process framework. Original image from Sutton and Barto (2018).

The *k*-armed bandit problem establishes the fundamental knowledge for understanding modern RL. In this problem, the agent assumes the system has only one constant state with many possible control actions. A classic example of the bandit problem would be choosing which slot machines to play in a casino. Here, the agent has one state (being inside the casino), and must identify which slot machine yields the highest reward. Objectively, the agent attempts to maximize reward over N steps by identifying the optimal control action, $u^* \in \mathcal{U}$, where $\mathcal{U}$ is a set of k possible actions. For each action in $\mathcal{U}_{k\times 1}$, there is an expected reward called *value*, given by:

$$q^*(u) = \mathbb{E}[R_t|U_t = u] \tag{2}$$

where u is the control action taken at time t, R_t is a scalar reward signal obtained by the agent after performing action u. For stationary (non-stationary) stochastic processes, R_t is drawn from a stationary (non-stationary) probability distribution, $R_t \sim N(q^*(u), \sigma^2)$. Finally, $q^*(u)$ is the expected reward of taking action u. In a real process, $q^*(u)$ is unknown, but can be estimated through exploration of each u. The estimated value is denoted as $Q(u)$. As all $u \in \mathcal{U}$ are picked infinitely many times, by the law of large numbers, $Q(u) \to q^*(u)$ (Borel, 1909).

Given any time t, one $Q_t(u)$ will be greater than all others, signifying its corresponding action is optimal at time t and should be picked. Methods where action selection is based on the estimated values are called action-value methods (Sutton and Barto, 2018). Algorithms to solve the *k*-armed bandit problem are easily applied to situations where the concept of state is inert and only the actions are of concern; a near impossibility in the real world.

Naturally, Barto et al. (1981) extended the original problem to incorporate many states and named it contextual bandits. In contextual bandits, different optimal policies are associated with different states (contexts). Mathematically, Eq. (2) was extended to:

$$q^*(x, u) = \mathbb{E}[R_t|X_t = x, U_t = u] \tag{3}$$

where x is the current state of the system. Here, the value function is a function of both the state and action; therefore, allowing the agent to behave differently for different situations. Nevertheless, the agent is still concerned with only the immediate reward, rather than the long term optimal trajectory. To alleviate this, RL was developed, introducing the concept of sequential decision making.

2.3. Markov decision process

The reinforcement learning paradigm is shown in Fig. 3 and consists of two components: the *agent* and the *system*. The agent is the continuously learning decision maker (i.e., RL algorithm). The agent attempts to learn and conquer the system through meaningful interactionswith the system. The system is comprised of everything the agent cannot arbitrarily change. Relating to process control, the agent would be the controller logic and everything else would make up the system. Reinforcement learning's decision making process is formalized in the Markov decision process (MDP).

Table 2
A comparison of different Markov decision processes.

FOMDPs	SMDPs	POMDPs
All states observable Discrete time	All states observable Continuous time	Some states observable Discrete time

The MDP is a discrete representation of the stochastic optimal control problem and a classical formulation of sequential decision making where both immediate and future rewards are considered (Bellman, 1957b; Sutton and Barto, 2018). MDPs provide formalism to agents when rationalizing about planning and acting in the face of uncertainty. Many different definitions of MDPs exist and are equivalent up to small alterations of the problem. One such definition is that a MDP, $\mathcal{M}$, is a tuple $(\mathcal{X}, \mathcal{U}, p(x', r|x, u), \gamma, R)$ comprised of (Ng, 2003):

- $x \in \mathcal{X}$: *state* space that describes the industrial system. Typical states in the process industry include temperatures, pressures, flow rates, etc.
- $u \in \mathcal{U}$: *action* space of the reinforcement learning agent. In control, this is the bounded control inputs.
- $R \in \mathbb{R}$: expected *reward* from the system after the agent performs u at x. Rewards are generated based on a desired performance metric and is called the *objective function* in control literature. Typically, $|R| \leq \mathcal{R}$ for stability and convergence purposes, where $\mathcal{R}$ is some upper bound of the reward.
- $p(x', r|x, u)$: *dynamics function* of the environment. It denotes the probability of transitioning to x' and receiving r given $x \in \mathcal{X}$, $u \in \mathcal{U}$ as described below:

$$p(x', r|x, u) \doteq Pr\{X_t = x', R_t = r|X_{t-1} = x, U_{t-1} = u\} \tag{4}$$

  where p describes the dynamics of the system and Pr denotes probability (Sutton and Barto, 2018). Additionally, p is a probability distribution satisfying:

$$\sum_{x' \in \mathcal{X}} \sum_{r \in \mathcal{R}} p(x', r|x, u) = 1, \forall x \in \mathcal{X}, u \in \mathcal{U} \tag{5}$$

  Notice that p depends only on the immediate past, thus assumes that x_{t-1} and u_{t-1} contain information about the history. This property is known as the Markov property and is critical for successful RL applications in process control. Note also that RL formulations with past state information augmented as: $x_{t-1} = [s_{t-1}, s_{t-2}, \ldots s_{t-N}], u_{t-1} = [a_{t-1}, a_{t-2}, \ldots, a_{t-N}]$, where s_{t-N} and a_{t-N} denote the past states and actions, still exhibit the Markov property because decisions can be made exclusively using x_{t-1} and u_{t-1}.
- γ: *discount factor* associated with future uncertainty, $(0 \leq \gamma \leq 1)$.

Three different versions of MDPs exist and are used to describe problems containing different characteristics as shown in Table 2 and are discussed below.

2.3.1. Fully observable MDPs

Fully observable MDPs (FOMDP) are used by agents to reason about decision making in discrete systems where all states are observable (measurable in control terminology). It will serve as the foundation of the MDP framework. The framework is initialized by the agent starting in some initial states x_0. At each time t, the agent picks some action u_t given x_t corresponding to its policy π. Given x_t and u_t, the system will then transition to the new state x_{t+1} following Eq. (4) and output reward R_{t+1} based on the desired performance metric. In control, the performance metric is typically the squared tracking error of the controller. By repeating this procedure many times, the agent is able to traverse through some sequence, $x_t, u_t, R_{t+1}, x_{t+1}, u_{t+1}, R_{t+2}, x_{t+2}, \ldots$ and accumulate:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ \ldots \tag{6}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{7}$$

where G_t is the cumulative discounted return at time t. Furthermore, γ is the discount factor and captures the uncertainty of future rewards. MDPs can be finite or infinite; the former describes episodic systems with explicit terminal states while the latter may continue forever. For example, a game of chess can be described as a finite MDP where the game is terminated after one player is defeated. Contrarily, an infinite MDP system, such as the control system in a refinery, could continue on indefinitely. For infinite MDP systems such as those in process control, $\gamma < 1$ is required to keep G_t bounded. The RL agent tries to find the optimal policy, π^*, that maximize G_t (instead of R_t in the bandit case) over N steps. The value function for each state in the system is given as Sutton and Barto (2018):

$$\begin{aligned} v_\pi(x) &= \mathbb{E}_\pi[G_t|X_t = x] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|X_t = x\right] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|X_t = x], \forall x \in \mathcal{X} \end{aligned} \tag{8}$$

where $v_\pi(x)$ is the value function of state x under policy π. Additionally, v_π is guaranteed to exist and be unique for continuous systems where $\gamma < 1$ or in systems with guaranteed termination. Compared to Eq. (2), Eq. (8) takes the expectation of G_t (defined in Eq. (7)) rather than R_t; therefore, optimizing the long term trajectory compared to only immediate rewards. The action-value version of Eq. (8) is given as:

$$\begin{aligned} q_\pi(x, u) &= \mathbb{E}_\pi[G_t|X_t = x, U_t = u] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|X_t = x, U_t = u\right], \forall x, u \in \mathcal{X}, \mathcal{U} \end{aligned} \tag{9}$$

FOMDPs can accurately represent discrete systems where all states are observable. Unfortunately, states in industrial processes are often times not fully measurable due to hardware limitations or other factors. During such situations, the system no longer exhibits the Markov property ultimately resulting in sub-optimal decision making.

2.3.2. Partially observable MDPs

Partially observable Markov decision processes (POMDPs) extend the concepts of FOMDPs and are used to solve systems with states that are no longer fully observable. Observability in RL terminology is equivalent to measurability in control as mentioned earlier; thus, the two terminologies will be used interchangeably here-forth. Previously in FOMDPs, the agent at each time t can observe its current state x_t. In the more general setting of POMDPs, the agent has a set of possible observations $\mathcal{O}$ rather than some states $\mathcal{X}$. At each time t, the agent instead sees observation o_t which corresponds to probability distributions over states giving the agent information regarding the state it might currently be in Ng (2003). In a process control setting, existing sensors usually only measure a subset of the current states directly. Using available measurements and the input information, one is able to infer the remaining unmeasured states using probabilistic inference approaches such as Kalman filter. Such systems are partially observable in RL terminology and are represented by POMDPs.

In general, finding the true optimal policy, π^*, in a POMDP system is significantly harder compared to FOMDPs. Even finding a near-optimal policy is NP-hard (non-deterministic polynomial

time) (Lusena et al., 2001). Additionally, agents knowing all the true value functions of the system are still unable to behave optimally in POMDP systems because the current states are unknown (Ng, 2003).

The use of belief states is one possible approach for agents to behave optimally in POMDPs. Belief states, b, are probability distributions over states representing what the agent thinks its current state is, given previous observations and actions. Using these probabilities, one can compute the value functions of each state-action pair and use it to act optimally. Note that the behaviour is not optimal with respect to the system, rather, it is optimal with respect to the available information. Ultimately, belief states transform the POMDP problem into a FOMDP since all belief states are fully available. An quantitative example is as follows:

> Assume an agent exists in a simple POMDP system with two unmeasurable states (x_1 and x_2) and two actions (u_1 and u_2) and suppose the problem has a horizon of one (for longer horizons, the agent must identify the trade-off between immediate and long term rewards, making the example less intuitive). Such a system has four value functions corresponding to the immediate reward obtained for each state-action pair. Suppose u_1 yields a reward of 2 in x_1 and 0 in x_2. Similarly, u_2 yields a reward of 0 in x_1 and 1 in x_2. If the current belief state b is [0.2, 0.8] (probabilities of being in x_1 and x_2, respectively), then $Q(b_t, u_1) = 0.2 \cdot 2 + 0.8 \cdot 0 = 0.4$ and $Q(b_t, u_2) = 0.2 \cdot 0 + 0.8 \cdot 1 = 0.8$, making u_2 the optimal action. For more relevant examples, please refer to Kaelbling et al. (1999).

In control theory, observers are leveraged to estimate unknown states. Observers are typically based on first principles models, but could also be based on data driven or probabilistic models. The concept of belief states is very similar to observer design in control theory. Kalman filter is a popular observer design method. Likewise in RL, recurrent neural networks (RNNs) are typically used to estimate the belief states. In Chenna et al. (2004), the performance of RNN is compared with Kalman filter, showing both methods' similarities in performance, objective, and theory.

Optimal solutions from FOMDPs and POMDPs work well in discrete tasks where transition time is consistent and transition dynamics are unimportant; however, such topics are critical to successful optimal control in the process industry.

2.3.3. Semi-MDPs

Typical MDPs are discrete representations of the optimal control problem and are sub-optimal in continuous tasks. Semi-Markov decision processes (SMDP) are an extension of MDPs to continuing tasks with unknown transition time and system dynamics. In SMDPs, the transition dynamics of the system are explicitly captured using reward function (Bradtke and Duff, 1994):

$$R(x_t, x_{t+1}, u_t) = \int_0^\infty \int_0^t e^{-\beta s} \rho(x_t, \pi(x_t)) ds dF_{x_t, x_{t+1}}(t|\pi(x_t)) \quad (10)$$

where $R(x_t, x_{t+1}, u_t)$ is the expected reward to be received when transitioning from x_t to x_{t+1} after action u_t. The rewards, R, are calculated at each time step in the transition period to explicitly capture transition information. Then, the average reward of the transition is used to update the agent. Here, $\rho(x_t, \pi(x_t))$ represents the average reward during the transition following policy, π. $F_{x,x_{t+1}}(t, u)$ denotes the probability distribution of the time required to transit from x_t to x_{t+1}. Finally, $\beta > 0$ denotes the constant discount factor in SMDPs, where higher β results in short-sighted agents. In SMDPs, the discount factor is corrected for transition time during each update step. The corrected discount factor is given by:

$$\gamma(x_t, x_{t+1}, u) = \int_0^\infty e^{-\beta t} dF_{x_t, x_{t+1}}(t|\pi_t) \quad (11)$$

where $\gamma(x_t, x_{t+1}, u_t)$ is the expected discount factor that will be applied to the value of state x_{t+1} during the update step shown in Eq. (1). The value function for SMDPs is obtained from combining Eqs. (10) and (8):

$$v_\pi(x_t) = \frac{1 - e^{-\beta\tau}}{\beta} R(x_t, x_{t+1}, \pi(x_t)) + e^{-\beta\tau} v_\pi(x_{t+1}) \quad (12)$$

where τ is the unknown transition time. Similarily, the action-value form is given by:

$$q_\pi(x_t, u_t) = \frac{1 - e^{-\beta\tau}}{\beta} R(x_t, x_{t+1}, \pi(x_t)) + e^{-\beta\tau} q_\pi(x_{t+1}, u_{t+1}) \quad (13)$$

By representing control problems as SMDPs, control strategies resulting in large overshoot, inverse response, or any other undesirable dynamics behaviour can be minimized. Additionally, this representation can handle systems with unknown transition time. An intuitive example illustrating the advantages of SMDPs in process control is as follows:

> Suppose a refinery company is operating a continuously stirred tank reactor (CSTR). Objectively, the CSTR must maintain 200° C for optimal performance. The temperature is controlled through a heat exchanger using cold water. A RL agent was built to optimally control the flow of cold water to maintain the temperature at the set point. Suppose the CSTR starts at 220° C. Agents using MDP representations may be overly aggressive and send large input signals because the reward is only calculated right before the next evaluation step. Therefore, input signals resulting in large overshoot or inverse response may not be reflected in the reward. Contrarily, SMDP representation uses the average reward accumulated along the trajectory to provide feedback to the agent, allowing the transition dynamics to be explicitly captured. This way, input signals resulting in undesirable behaviour can be captured and mitigated. Furthermore, SMDP representations can have flexible evaluation time (traditional representations evaluate after a set time period), enabling re-evaluation during the transitional period and adjusts the discount factor in accordance to the elapsed time from the last evaluation.

2.4. Solving the Markov decision process

The optimal solution to a reinforcement learning problem refers to the policy that generates the highest reward over a trajectory. Formally, an optimal policy must satisfy the principle of optimality: the optimal policy π^* is optimal if and only if $v_{\pi^*}(x) \geq v_{\pi \neq \pi^*}(x)$ for all $x \in \mathcal{X}$ (Poznyak, 2008). Note that there may exist many optimal policies, where $v_{\pi_1^*} = v_{\pi_2^*} = \ldots = v_{\pi_N^*}$. The optimal value function is denoted mathematically as:

$$v^*(x) \doteq \max_\pi v_\pi(x), \forall x \in \mathcal{X} \quad (14)$$

Similarily, the optimal action-value function is denoted as:

$$q^*(x, u) \doteq \max_\pi q_\pi(x, u), \forall x, u \in \mathcal{X}, \mathcal{U} \quad (15)$$

In a more explicit form, the optimal value function and action-value function written in terms of Eqs. (8) and (9) are given, respectively, by Sutton and Barto (2018):

$$v^*(x) = \max_u \mathbb{E}[R_{t+1} + \gamma v^*(X_{t+1}) | X_t = x, U_t = u] \quad (16)$$

$$q^*(x, u) = \mathbb{E}\left[R_{t+1} + \gamma \max_{u_{t+1}} q^*(X_{t+1}, u_{t+1}) | X_t = x, U_t = u\right] \quad (17)$$

Here, the *max* denotes that the optimal action will be taken in the next step and thereafter for the remaining of the trajectory. In theory, the optimal value functions can be explicitly solved for all systems using the above equation; however, such a task would require tremendous amounts of computation power even in trivial tasks.

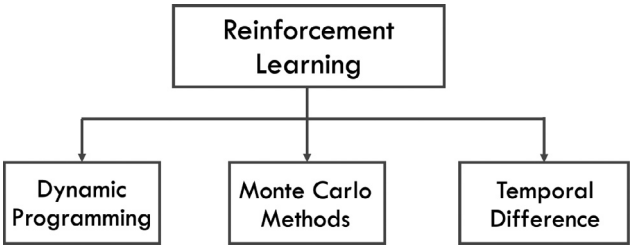


Fig. 4. The sub-components of machine learning.

In the following section, three popular methods will be introduced to estimate the value and action-value functions in reinforcement learning.

2.5. Methods of reinforcement learning

The three families of algorithms used to solve optimal policies in RL are shown in Fig. 4. Dynamic programming (DP) methods can provide exact solutions to the optimal policy, but requires a perfect system model and has infeasible computational requirements for non-trivial tasks. Comparatively, both Monte Carlo (MC) and temporal difference (TD) methods approximate DP solutions using less computational power and does not assume the presence of a perfect system model. MC methods find the optimal policy through averaging the value function over many sampled trajectories of states, actions, and rewards; however, the variance in the sampled trajectories results in high variance in final results. TD methods combine the ideas of DP and MC methods into one unifying algorithm. TD methods learn from sampled data like in MC methods, while also being able to perform *mid-trajectory* learning, like in DP methods; however, TD methods experience high bias due to estimating values through previously estimated values (known as bootstrapping). The general details of each method will be shown throughout this section. For a comprehensive introduction to each algorithm, see Sutton and Barto (2018).

2.5.1. Dynamic programming

Dynamic programming refers to a set of algorithms with the ability to find optimal policies assuming a perfect model is available. DP algorithms are in general not widely used due to their very high computational cost for non-trivial problems. The two most popular methods in DP are policy iteration and value iteration.

On a high level, policy iteration searches for the optimal policy by iterating through many policies, $\pi \in \Pi$, keeping only the policy with the highest cumulative returns. The optimal policy is found when the cumulative returns of π can no longer be improved (convergence). Policy iteration includes two iterative steps: policy evaluation and policy improvement. *Policy evaluation* predicts the value functions for policy π through an iterative approach. Value functions for all states are initialized as 0, and are updated using:

$$\begin{aligned} v_{k+1,\pi}(x) &= \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{k,\pi}(x_{k+1})] \\ v_0(x) &= 0, \ \forall x \in \mathcal{X} \end{aligned} \tag{18}$$

where k is the k^{th} update step, and $v_{k+1,\pi}$ is the predicted value function following policy π after $k+1$ update steps. As $k \to \infty$, $v_k(x) \to v_{\pi}(x)$ for all $x \in \mathcal{X}$ (i.e., the true value functions for π). However, there may exist a π' where $v_{\pi'}(x) \geq v_{\pi}$, deeming π suboptimal. The goal of *policy improvement* is to identify situations where $v_{\pi'}(x) \geq v_{\pi}$ for any state. Once identified, π will violate the principle of optimality, hence disqualifying it from being optimal and π' will be deemed the new optimal policy. This procedure will continue iteratively and infinitely until a policy where $v_{\pi^*}(x) \geq v_{\pi \neq \pi^*}(x)$ for all $x \in \mathcal{X}$ is found.

A visualization of the policy iteration algorithm is shown in Fig. 5 and is as follows (Sutton and Barto, 2018):

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi^* \xrightarrow{E} v^* \tag{19}$$

where $\xrightarrow{E}$ and $\xrightarrow{I}$ represent the policy evaluation and policy improvement steps, respectively. From Fig. 5, the agent starts with some arbitrary policy. Initially, there exists a large gap between V_{π} and π, illustrating much room for improvement. As policy iteration continues, the gap is reduced until $V(x)$, $\pi \to V^*(x)$, π^*. Presently, policy iteration is rarely used due to its high computational expense for the required iterative steps for each policy evaluation.

Value iteration finds the optimal policy through identifying the optimal value functions rather than evaluating many policies. Intuitively, it is a special case of policy iteration where the policy evaluation is terminated after one step. After identifying the value functions, the optimal policy can be trivially extracted by simply traversing through the states and identifying the actions corresponding to the highest values. Note here that extraction of the optimal policy using $V(x)$ is only possible if a perfect model of the system is provided. Using the model, one can find the state transition probabilities, and subsequently identify the actions with the highest probabilities to traverse to the high value states. Without a model, $Q(x, u)$ must be identified instead of extracting the optimal policy. The one-step policy evaluation for the value function and action-value function, respectively, is:

$$v_{k+1}(x) = \max_{u} \mathbb{E}[R_{t+1} + \gamma v_k(x_{t+1})] \tag{20}$$

$$q_{k+1}(x, u) = \mathbb{E}[R_{t+1} + \gamma \max_{u_{t+1}} q_k(x_{t+1}, u_{t+1})] \tag{21}$$

where the max operation ensures that each $v_k(x)$ is updated using only the maximizing action; thus, ultimately finding $v^*(x)$. After convergence of all $v^*(x)$, an agent can be initialized in any arbitrary state and still behave optimally as long as the agent takes the maximizing action in each successive state. Note that both policy and value iteration *bootstraps* its current estimate using previously calculated values, $v_k(x_{t+1})$ and $q_k(x_{t+1}, u_{t+1})$. This concept is used by RL to increase data efficiency and allow updates to explicitly capture long-term trajectory information; however, the method also introduces unintended biased updates.

In industrial problems, both policy and value iterations have limited utility because their updates are applied to all $x \in \mathcal{X}$ simultaneously (i.e., all value functions for all states are found simultaneously). In large multi-dimensional problems, performing even one iterative step may be infeasible. Asynchronous dynamic programming methods try to avoid this problem by only updating frequently visited states, while avoiding the update of states that are never visited. Doing so can reduce computation time considerably, but render the agents less useful in states that are rarely encountered.

2.5.2. Monte Carlo methods

Unlike dynamic programming, Monte Carlo methods technically do not require a model of the system (a characteristic known as *model-free*). MC methods find the optimal policy by first estimating the average returns for different policies by sampling many sequences of states, actions, and rewards under said policy. As enough samples are generated, $v_k(x) \to v_{\pi}(x)$ for all $x \in \mathcal{X}$. The average returns are updated after each trajectory. Due to the nature of MC updates, the most suitable systems are finite tasks with explicit terminal states, called *episodic* tasks. Discrete manufacturing is an example of an episodic task in process control. After the assembly of each object (cars, toys, etc.), the system terminates and starts anew. In episodic tasks, the value functions can be updated naturally after each terminal state. Typically, episodic tasks are rare

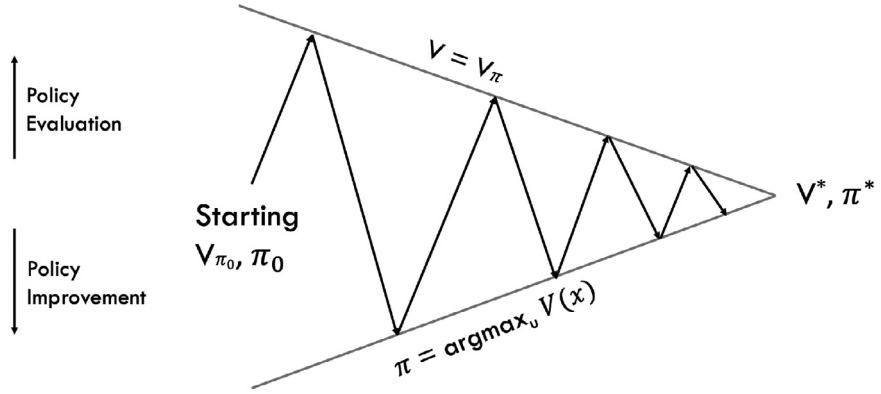


Fig. 5. A visualization of the policy iteration algorithm. Original image from Silver (2018).

in process control because most control systems are required to operate indefinitely. Processes with no terminal states are known as *continuous* tasks. To train a continuous task agent using the MC method, a maximum episode length should be initially specified so the agent can stop and update its value functions at certain intervals. In doing so, the agent can exploit its new learnings before continuing onward. Note here that all estimated value functions are independent and unbiased since no bootstrapping was used (opposite of dynamic programming); however, MC methods may suffer from large variances for systems highly corrupted by noise (Sutton and Barto, 2018). Moreover, exploration is mandatory in MC methods since the system model is not available to the agent. Only through exploration can the agent discover the value functions (state transition probabilities and rewards) for each action in each state, and subsequently, the optimal policy. Typically, exploration is conducted by starting in a random state at the beginning of each episode. After a sufficiently large amount of episodes are explored, all states will be visited sufficiently many times.

Policy search in MC methods is similar to policy iteration. There are three main differences: 1) not all states are updated simultaneously; 2) value functions are updated using sampled data from the agent interacting with the environment; 3) $q_\pi(x, u)$ is identified instead of $v_\pi(x)$. In MC methods, the action-value functions are identified instead because a model is not provided to the agent. The value functions alone are not useful to the agent because the actions required to transition to the high value states are not known. Instead, the action-values provide the agent with explicit information on the expected returns for each action in each state. The iterative procedure to find the cumulative returns is given by:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \ldots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} q_{\pi_*} \tag{22}$$

Intuitively, training starts by the agent being initiated in an unknown system. Here, the agent traverses through the state space $x_1, x_2, \ldots, x_n$ by performing actions $u_1, u_2, \ldots, u_n$ under a policy, π, and collect rewards $R_1, R_2, \ldots, R_n$. Eventually, the agent will reach a terminal state, concluding the first episode. Upon termination, a sequence of returns $G_1, G_2, \ldots, G_{n-1}$ can be calculated via the collected rewards throughout the trajectory using:

$$G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots + \gamma^{n-1} R_n$$

$$G_2 = R_2 + \gamma R_3 + \gamma^2 R_4 + \ldots + \gamma^{n-2} R_n$$

$$G_3 = R_3 + \gamma R_4 + \gamma^2 R_5 + \ldots + \gamma^{n-3} R_n$$

$$\vdots$$

$$G_{n-1} = R_n$$

or:

$$G_m = \sum_{i=0}^{n} \gamma^i R_{m+i} \tag{23}$$

where G_m is the discounted cumulative return received on the m^{th} step. Next, the action-values, $Q(x, u)$, are estimated for each step by using the states, actions, and returns:

$$Q_{k+1}(x, u) = Q_k(x, u) + \frac{1}{k}[G - Q_k(x, u)] \tag{24}$$

where $Q_k(x, u)$ represents the k^{th} action-value update and G corresponds to the sampled returns of performing u in x. Note that Eq. (24) is unsuitable for non-stationary processes because as $k \to \infty$, $\frac{1}{k} \to 0$. For non-stationary processes, Eq. (24) becomes:

$$Q_{k+1}(x, u) = Q_k(x, u) + \alpha[G - Q_k(x, u)] \tag{25}$$

where $\alpha \in (0, 1]$ is the learning rate (also called step size). Here, α is lower bounded to prevent the update from approaching 0; therefore, continually adapting in non-stationary problems. Additionally, α should not be too high or the updates will be significantly affected by short-term noise. After each update, a new episode starts and the above procedure is repeated. As the number of episodes approaches infinity ($k \to \infty$), $Q(x, u) \to q(x, u)$ (i.e., the estimated action-values, $Q(x, u)$ approaches the true action-values, $q(x, u)$). Once the action-value functions reach convergence, the optimal policy can be extracted by:

$$\pi^*(x) = arg\max_u q(x, u) \tag{26}$$

That is, the optimal policy is simply performing the action corresponding to the highest expected returns in each state. MC methods allow the agent to learn directly from trial-and-error experiences without a system model; however, whole episodes must be completed before the agent can update its knowledge base. Additionally, such a procedure is unnatural in continuous systems (most common in process control), severely disadvantageous in systems with long episodes, and is not intuitive to human behaviour. Humans typically learn immediately after feedback, not in pre-set increments. Temporal difference methods combine the ideas of both dynamic programming and Monte Carlo methods to become more human-like.

2.5.3. Temporal difference learning

Temporal difference (TD) methods are the most widely used RL algorithm today because of its simplicity and relatively cheap computational cost. TD methods combine the ability to learn from experiences (like MC methods) with bootstrapping (like DP methods).

TD methods do not require a model of the system, and will instead learn the dynamics from interactions. Moreover, TD methods do not need to wait until the termination of an episode before updating its value functions. Instead, TD methods can update immediately after x_{t+1} and R_{t+1} are received. The TD update for value and action-value functions are given by Eqs. (27) and (28), respectively (Sutton, 1988):

$$V(x_t) \leftarrow V(x_t) + \alpha[R_{t+1} + \gamma V(x_{t+1}) - V(x_t)] \tag{27}$$

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha[R_{t+1} + \gamma Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)] \tag{28}$$

where $\leftarrow$ denotes the update operator. Intuitively, the old value functions are corrected using the *TD errors* at each update by a fixed amount (dictated by α). The TD errors are given as:

$$\delta_t = R_{t+1} + \gamma V(x_{t+1}) - V(x_t)$$

$$\delta_t = R_{t+1} + \gamma Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)$$

where δ_t denotes the TD error at time t, the two terms, $R_{t+1} + \gamma V(x_{t+1})$, denote what the agent *thinks* the real value function is according to the last interaction, and $V(x_t)$ is the old value function. After the agent traverse through each state-action pair many times, $V(x_t) \rightarrow v(x_t)$ (i.e., the estimated values converge to the true values). The action-values follow a similar paradigm. After convergence, the optimal policy can be extracted via Eq. (26).

The algorithms presented in Eqs. (27) and (28) are called $TD(0)$ because the agent updates its knowledge after just one action. $TD(0)$ is a special case of the more general TD(λ) algorithm. Like DP, $TD(0)$ also experiences high bias due to bootstrapping. However, as $TD(0) \rightarrow TD(1)$, bootstrapping is reduced and at $TD(1)$, the algorithm becomes the MC method. Details regarding this algorithm can be abstract and will be omitted. A detailed description of $TD(\lambda)$ and supplementary code can be found in Reis (2017).

Like MC methods, TD methods are also *model-free*; therefore, action-values are typically learned and exploration is mandatory. TD methods typically explore using ϵ-greedy policies where the agent performs a random action with $\epsilon \in [0, 1]$ probability and performs the returns-maximizing action otherwise. Furthermore, ϵ is typically decayed throughout training and starts at a high value during the initial phase when the agent knows nothing. Eventually, ϵ decays to a low value when training is almost complete.

There are two popular TD methods with slightly different update steps: *SARSA* and *Q-learning*. SARSA is an on-policy algorithm meaning that its behaviour policy is identical to its target policy. Target policy refers to the policy the agent wants to eventually find. Typically, this will be the optimal policy. On the other hand, the behaviour policy, $b(u|s)$, is how the agent actually behaves. If both the target and behaviour policy are identical, the agent is on-policy. An on-policy agent (assuming the target policy is the optimal policy) during training may quickly converge to a local optimum and never explore (since exploratory policies are typically not the optimal), resulting in a sub-optimal solution. Contrarily, off-policy agents, such as *Q*-learning, may follow an equiprobable random policy (equal probability of selecting all actions in all states) during training to conduct deep exploration and switch to the optimal policy online. Moreover, off-policy agents are guaranteed to find the optimal policy under the assumption that each state-action pair is visited infinite times and $b(u^*|s) > 0$ (i.e., the probability of picking the optimal action under behaviour policy is not 0) (Sutton, 1988). Since SARSA is an on-policy approach, the action-value functions are updated through Eq. (28) with the quintuple $(x_t, u_t, R_{t+1}, x_{t+1}, u_{t+1})$. Compared to SARSA, *Q*-learning updates by using only four parameters $(x_t, u_t, R_{t+1}, x_{t+1})$ through Eq. (29) while assuming u_{t+1} is a decision variable to maximize the action-value function:

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha\left[R_{t+1} + \gamma \max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)\right] \tag{29}$$

In *Q*-learning, u_{t+1} is not used because the actual u taken at time $t+1$ might differ from the target policy since the algorithm is off-policy. To ensure *Q*-values are still updated towards the optimal policy, Eq. (29) uses the max operation to force the update into using the optimal u_{t+1}. Overall, TD methods unify the best parts of DP and MC methods, allowing the agent to learn purely based on experiences while still being able to perform inter-episode updates to exploit the most recent learnings.

2.5.4. Summary of different RL methods

A summary of the characteristics for the DP, MC and TD methods is shown in Table 3. On a high level, dynamic programming requires a model to learn the value functions while both MC and TD methods can learn from sampling state, action and rewards alone. DP and TD methods use bootstrapping to estimate value functions; that is, they estimate the current value function based on previously estimated value functions. This method is data efficient, but introduce unwanted bias to the estimates. On the other hand, MC methods estimate each value function independently through sampling many trajectories to avoid estimation biases. However, this method instead introduces high variance for noisy systems. In terms of computational cost, DP methods require the most because all value functions are simultaneously updated. Comparatively, MC methods only update the value functions that were visited in the sampled trajectories, and updates are conducted at the end of each episode. TD methods update the value function immediately after an experience, and are also very efficient since it only updates the value functions in the visited states. For exploration, DP methods do not require any because the model of the system (both transition probabilities and expected reward) is known to the agent at all times. For MC methods, exploration is conducted by starting at random states when episodes are reset. In TD methods, the agent will occasionally perform a random action. A detailed introductory example is provided in Section 3 to enhance additional intuition and understanding of the implementation procedure.

2.6. Function approximation methods

Function approximation is widely used in applications of RL in industry. Function approximation methods are briefly discussed in this section to provide a more comprehensive overview of RL methods.

Originally, RL was proposed to solve MDPs, which are discrete representations of the optimal control problem. For discrete tasks, the value functions are stored in a table, known as the *Q*-table, with the x- and y-axes being the states and actions. A visual representation of the *Q*-table for a system with n states and m actions is shown in Fig. 6. However, the states and actions of a typical process control application are both multi-dimensional and continuous. The storage of value functions in such complex tasks could be indefinitely large and intractable. Extending RL solutions to such systems require the value functions to be generalized using a parameterized functional form. Function approximation can be used to address this issue. The approximate value function is given as:

$$\hat{v}(x, \mathbf{w}) \approx v_\pi(x) \tag{30}$$

where $\hat{v}$ is a continuously differentiable approximation of the value function and $\mathbf{w} \in \mathbb{R}^d$ is the weight vector, where d is much smaller than the number of states. Supervised learning models can be used to approximate $\hat{v}$ and can be linear or nonlinear. In the tabular case, learning is decoupled – updates of one value function do not impact any other value functions – allowing the optimal value

Table 3
A comparison of DP, MC, and TD methods.

	Dynamic Programming	Monte Carlo	Temporal Difference
Requires model	Yes	No	No
Estimate bias	High	Low	High
Estimate variance	Low	High	Low
Computational cost	High	Medium	Low
$v(x)$ update	All states simultaneously	After a trajectory	After an experience
Exploration	Not needed, all states update	Random initialization	Performing a random action

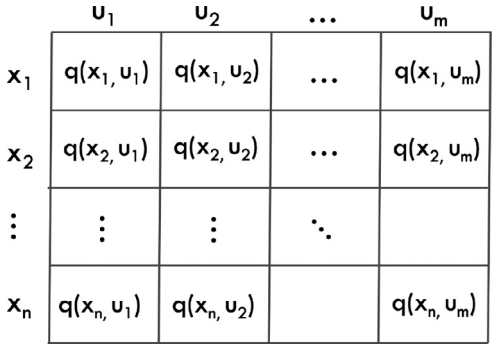


Fig. 6. An example of the Q-table for a system with n states and m actions.

functions to be found for all states. This no longer holds in the approximation case. Instead, as an element of $\mathbf{w}$, w_i, is updated, all approximate value functions utilizing w_i will change accordingly, making it impossible for all value functions to be exactly correct (Sutton and Barto, 2018). As such, RL methods using function approximation can approach, but never achieve, perfect optimal control. To focus accuracy of the function approximation on the most important states, the state distribution $\mu(x) \geq 0, \sum \mu(x) = 1$ must be defined. Often times, $\mu(x)$ can simply be the fraction of time spent in x and can be computed trivially as:

$$\mu(x) = \frac{\eta(x)}{\sum_{x'} \eta(x')}, \forall x \in \mathcal{X} \tag{31}$$

where $\eta(x)$ is the amount of time spent in state x and $\Sigma\eta(x')$ is the total time spent in all states. Combining the state distribution with mean squared error, the *mean squared value error* objective function can be define as:

$$\overline{VE}(w) = \sum_{x \in \mathcal{X}} \mu(x) \left[v_\pi(x) - \hat{v}(x, \mathbf{w}) \right]^2 \tag{32}$$

Ultimately, the goal for approximate value function methods is to find the optimal weight vector w^* for which $\overline{VE}(w^*) \leq \overline{VE}(w)$ for all w. The weights are typically identified using gradient methods such as stochastic gradient descent (SGD). SGD is a popular method in machine learning due to its ease of use and advantages in big data applications. SGD is a special case of the more general gradient descent algorithm. In gradient descent, all available data are used to compute the gradient; however, such a method is computationally infeasible in big data applications. Instead, SGD is used to randomly sample a subset of data for the gradient computation. More detailed explanations about SGD can be found in Goodfellow et al. (2015). On each step, SGD adjusts the weights w slightly (adjustment size controlled by α) with accordance to the gradient of the loss function (Bottou, 2010):

$$w_{k+1} = w_k - \frac{1}{2} \alpha \nabla \left[v_\pi(x_t) - \hat{v}(x_t, w_k) \right]^2 \tag{33}$$

$$w_{k+1} = w_k + \alpha \left[v_\pi(x_k) - \hat{v}(x_k, w_k) \right] \nabla \hat{v}(x_k, w_k) \tag{34}$$

where w_{k+1} is the weight vector after the $(k+1)^{th}$ update. Furthermore, α and ∇ are the learning rate and gradient operator, respectively. Larger α results in larger update steps and are typically used at the beginning of training. As $k \to \infty$, $\alpha \to 0$ to ensure that the weights do not oscillate in noisy systems. In Eq. (33), it was assumed that the true value $v_\pi(x_t)$ was known; this is possible in theory but not in real world applications. Instead, $V_k(x)$ is used in practical applications, where $V_k(x)$ is an unbiased estimate of $v_\pi(x)$ satisfying $\mathbb{E}[V_k | X_t = x] = v_\pi(x)$.

Now that the objective function and model optimization algorithms are defined, the simplest function approximation approach will be introduced. Suppose that each state x can be represented using a feature vector:

$$\mathbf{s}(x) = [s_1(x), s_2(x), \ldots, s_d(x)]$$

that has the same length as w. Additionally, each component s_i of $\mathbf{s}(x)$ is a function known as a feature of x. In linear methods, the features form a linear basis for the set of approximation functions and are known as the basis functions (Sutton and Barto, 2018). Using the basis functions, the value function can be approximated as a linear basis function using:

$$\hat{v}(x, w) = w_1 s_1(x) + w_2 s_2(x) + \ldots + w_d s_d(x)$$

Or

$$\hat{v}(x, w) = \mathbf{w}^T \mathbf{s}(x) = \sum_{i=1}^{d} w_i s_i(x) \tag{35}$$

Note that Eq. (35) is linear-in-parameter. Although such a model does not explore interaction effects between features, it is still effective for many value functions. The gradient of $\hat{v}(x, w)$ from Eq. (35) is nothing more than:

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{s}(x) \tag{36}$$

By combining Eqs. (36) and (34) yields:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \left[V_k - \hat{v}(x_t, \mathbf{w}_t) \right] \mathbf{s}(x_t) \tag{37}$$

This is a very simple and elegant solution that is easy to implement and interpret. Additionally, the solution to $\mathbf{w}^*$ is unique due to its linear structure. Lastly, the design of the basis functions will be briefly explained. Here, the simplest basis function will be introduced: the *polynomial basis function*. Polynomial functions offer high flexibility in its model structure and are intuitively to implement. The general structure of a two-regressor polynomial model is given as:

$$\begin{aligned} y = {} & w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + \ldots + w_n x_1^{\frac{n}{2}} \\ & + w_{n+1} x_1^{\frac{n}{2}+1} + w_{n+2} x_1 x_2 + w_{n+3} x_1^2 x_2 + \ldots + w_0 \end{aligned} \tag{38}$$

where w_i are the model weights and w_0 is the model bias. Here, x_1 and x_2 are states from an arbitrary system. A more illustrative example for designing the features for an agent in a process control problem is shown below:

> Suppose we are tasked with optimally controlling the temperature of a CSTR with two states: mass flow rate x_1, and temperature x_2. Due to limited hardware and continuous states, the system requires function approximation. The task is to construct a

Fig. 7. Growth in search results of reinforcement learning on Google from 2007 - 2019. Figure from Trends (2017).

feature vector that captures both states. A trivial design would be:

$$\mathbf{s}(x) = [s_1(x) = x_1, s_2(x) = x_2],$$

where Eq. (35) becomes:

$$\hat{v}(x, \mathbf{w}) = w_1 x_1 + w_2 x_2.$$

Such a design is simple and intuitive, but does not explore the interaction effects between x_1 and x_2. In chemical reactors, the interaction effects between mass flow rate (x_1) and temperature (x_2) can be critically important for control and is used to estimate the enthalpy of the reaction (Borgnakke and Sontag, 2008). Moreover, the system may be affine and does not depend on either x_1 or x_2. In such a case, the relationship is lost when both x_1 and x_2 are 0 because $\hat{v}(x, \mathbf{w})$ must equal 0. To accommodate for these factors, the feature vector could instead be:

$$\mathbf{s}(x) = [s_1(x) = 1, s_2(x) = x_1, s_3(x) = x_2, s_4(x) = x_1 x_2]$$

where $s_1(x) = 1$ captures affine relationships and $s_4(x) = x_1 x_2$ captures interaction effects.

There exists many more advanced basis functions and function approximation methods such as Fourier basis functions, coarse coding, tile coding, radial basis functions and nonlinear neural network function approximation. For detailed information regarding these methods, please refer to Chapter 9 in Sutton and Barto (2018).

2.6.1. Emergence of deep reinforcement learning

This subsection introduces major contributions in the deep RL field, where deep neural networks are used for function approximation in *Q*-learning.

Fig. 7 shows the interest in RL worldwide from 2007 - 2019. It can be seen that RL had rather stagnant interest until 2015, when significant growth began. The first contributing factor to this growth was the publication of Mnih et al. (2013) in 2013, where the authors introduced a deep RL algorithm, called deep *Q*-learning network (DQN), that was able to play many Atari games from image inputs alone using the same algorithm (though the agent had to be re-trained for each game). Performance of the agent was compared to human players after training for 10 million frames (approximately 56 hours of continuous play time[1]). To ensure a fair comparison, the skill of the agent was handicapped by human features such as delays in response time and was only able to see information presented on the screen. DQN surpassed human level in half of the games. DQN used *Q*-learning with deep neural networks for function approximation; the complete details can be found in Mnih et al. (2013). In DQN, continuous states can be used, but the actions were discrete because Atari games typically had discrete inputs. Furthermore, like other TD methods, DQN experienced high bias. The algorithm was a massive contribution in RL literature but lacked continuous actions required for process control.

[1] Assuming Atari 2600 games ran at 50 frames per second (Monfort and Bogost, 2018).

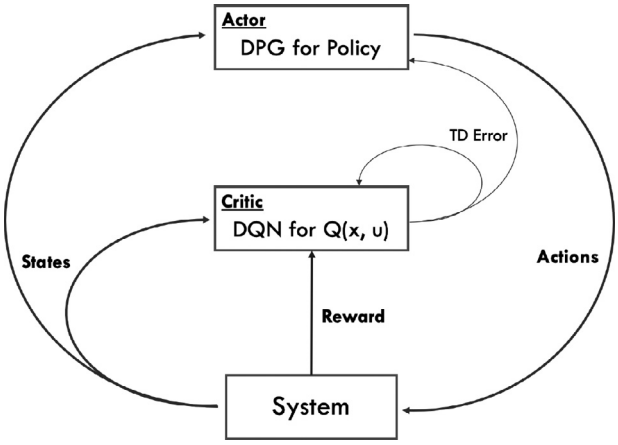


Fig. 8. The architecture of the deep deterministic policy gradient algorithm (Lillicrap et al., 2015).

The first RL algorithm that can handle both continuous states and actions naturally was developed in Silver et al. (2014). The algorithm, called deterministic policy gradient (DPG), was a policy iteration method trained using Monte Carlo methods. DPG deterministically maps continuous states to continuous actions through a neural network that approximates the optimal policy. Although DPG can handle both continuous states and actions and is another major contribution, it is very computationally expensive, experiences high variance, and is unnatural for continuous tasks due to the Monte Carlo training method. More details regarding the DPG algorithm can be found in Silver et al. (2014).

In 2015, RL had evolved again into an algorithm that may be suitable for online optimal control. Lillicrap et al. (2015) pushed the boundary of RL literature through unifying the ideas of Mnih et al. (2013) and Silver et al. (2014) into one actor-critic RL algorithm known as the deep deterministic policy gradient (DDPG). The general architecture is shown in Fig. 8. Here, the DPG was used to map states into actions and was known as the *actor*. Moreover, the DQN was the *critic* and was used to identify the action-values to update the DPG without using MC methods. Originally, DQN and DPG suffered from high bias and high variance, respectively. However, by unifying the two algorithms, both the bias and variance can be significantly reduced (Lillicrap et al., 2015). Additionally, policy gradient methods are typically trained using Monte Carlo methods at the end of episodes (Sutton et al., 1999); however in DDPG, this limitation is overcome through training the DPG by the gradient of the critic (i.e., the derivative of the *Q* values with respect to the model weights) using gradient ascent. Intuitively, the DPG's weights are updated to maximize the action-value, $Q(x, u)$. Traditionally, continuous action RL algorithms struggle in exploration because techniques such as ϵ-greedy assumes a discrete action space. DDPG explores through the Ornstein-Ulhenback ex-

ploratory noise given by:

$$u'(x_t) = u(x_t|w_t) + \mathcal{N} \tag{39}$$

where $u'(x_t)$ is the input signal corrupted by the Ornstein-Ulhenback exploratory noise, $\mathcal{N}$, given by the following stochastic differential equation (Uhlenbeck and Ornstein, 1930):

$$dx_t = \theta x_t dt + \sigma dW_t, \tag{40}$$

where $\theta > 0$, $\sigma > 0$, and W_t denotes a special case of a continuous time stochastic process, known as the Wiener process. Detailed information regarding the Wiener process and its properties can be found in Karatzas and Shreve (1991). Sometimes, Gaussian white noise is used for exploration; however, de-correlated random signals are ineffective for deep exploration since the signal has zero average effect resulting in no displacement in any particular direction and simply introduces oscillation in the process. Intuitively, $\mathcal{N}$ is a temporally correlated process that promotes deep exploration and needs to be tuned to suit different environments. More detailed theoretical overview of the DDPG algorithm can be found in Lillicrap et al. (2015).

In DDPG, four neural networks are used collectively when identifying the optimal policy. Furthermore, the neural networks are implicitly trained upon each others weights, which potentially causes function approximation errors and leads to sub-optimal policies. In Fujimoto et al. (2018), the authors introduced a novel way to minimize the errors by delaying policy updates. The new algorithm was also tested on different benchmark environments and showed better performance in each case.

DDPG was the first RL algorithm that effectively solved many high-dimensional continuous control tasks. Indeed, most previous actor-critic or policy optimization RL algorithms were never demonstrated to work on high dimensional state and action spaces due to instability caused by catastrophic interference (a symptom where newly learned information would undesirably overwrite previously learned knowledge) or were simply learning too slow for industrial applications (Deisenroth and Neumann, 2013). Instead, the abilities of DDPG were demonstrated on many high dimensional control problems in the MuJoCo physics environment, some being as large as 102 observations and 9 actions (Lillicrap et al., 2015).

At around the same time, Schulman et al. (2015) proposed another deep RL method utilizing policy optimization. This algorithm was also shown to be effective on high-dimensional continuous control tasks. The algorithm, known as trust region policy optimization (TRPO), guarantees monotonic improvements after each update step through careful parameter updates governed by applying a KL divergence threshold on the new policy compared to the existing policy. More specifically, the update constraint guarantees that the new policy lies within the *trust region*: a subspace in which the local function approximations are reliable. Unlike DDPG where there exist an actor and a critic, TRPO identifies the policies directly. In doing so, Lillicrap et al. (2015) believe that TRPO is much less data efficient. Moreover, the parameter updates must be solved using a conjugate gradient method, due to the update constraint, which may be difficult to implement.

To improve upon the previous flaws of TRPO, Schulman et al. (2017) published a new RL algorithm in 2017 called proximal policy optimization (PPO). Compared to TRPO, PPO was much simpler to implement, more general, and has improved data efficiency. Specifically, PPO implements the update constraint directly in the objective function as a penalty. In doing so, SGD can be used instead. Test cases showed that PPO was able to achieve much better performance compared to TRPO after training for the same time on various continuous control tasks. Unfortunately, Schulman et al. (2017) did not compare its performance against DDPG, making it difficult to conclude if it is a better algorithm.

In 2018, Haarnoja et al. (2018) introduced a new actor-critic algorithm to improve the sample efficiency and convergence factors of previously introduced methods. In the new actor-critic algorithm, both expected reward and action randomness are maximized together during optimal policy search. Ultimately, this new algorithm was shown to surpass state-of-the-art performance on various continuous control benchmarks while being stable during its learning process.

A complete time line of popular deep reinforcement learning methods is shown in Fig. 9. Deep RL methods began with continuous states and discrete actions but quickly evolved to accommodate for continuous actions as well. Unfortunately, the MC methods were required to train the method, resulting in high variance. DDPG was introduced to resolve this shortcoming; deep algorithms from 2015 there-forth can handle both continuous states and actions. For a comprehensive performance comparison between state-of-the-art continuous control deep RL algorithms, please refer to Henderson et al. (2018).

2.6.2. Deep RL and its implications on industrial control

The implementation potential of RL before the emergence of deep function approximation was quite limited due to its application being confined to discrete state and action systems. Using deep function approximation, RL succeeded in solving various complex tasks such as in Lillicrap et al. (2015) and Mnih et al. (2013). However, most deep RL applications thus far have been in simulated environments and were not implemented in safety-sensitive industrial settings. Furthermore, using deep learning for function approximation require proper tuning of the neural networks and results are not repeatable because neural networks are typically initiated with random weights and the exploration phase is also stochastic. Moreover, training agents that leverage large neural networks can take up to several days depending on the computation power available. Assuming that the deep function approximations are tuned properly, two major hurdles for deep RL still exist: i) the black-box nature of the control policy; ii) the connectivity of RL to the industrial distributed control system (DCS).

Firstly, the black-box nature of RL introduces risks to process operations because predicting the behaviour of RL may be difficult. In MPC, the control actions could be understood through analyzing the system model. However, in RL, it is nearly impossible to understand why or how the agent learned the control policy. One could test the behaviour of RL by providing different operating conditions and observing the response of the agent; however, control policies using deep function approximation are likely to be highly nonlinear. This implies that operating conditions close to the tested points may still result in very different behaviours. Secondly, reliable RL communication with industrial DCS systems might pose challenges because there exists no industrially accepted RL software as of 2020. Moreover, deep learning control policies cannot be directly imported in modern industrial DCS because of a lack of support for large, interacting, neural networks. One possible solution is to build the RL agent in an external software and leverage Open Platform Communication (OPC) to communicate to modern control systems used by AspenTech's Advanced Process Control Suite (AspenTech, 2019).

Due to the stochastic and black-box nature of deep learning and the difficulty of implementing such a system into modern control systems, deep RL may still require further research before it is industrially ready.

3. Applications of reinforcement learning

This section starts with a literature review of the most influential RL papers. Then, RL will be compared to traditional control

Fig. 9. Date of significant breakthroughs in deep reinforcement learning.

frameworks. Afterwards, a detailed tutorial where RL was applied onto an industrial pumping system will be shown. Finally, a literature review of other potential applications of RL in the process control industry will be introduced.

3.1. Renowned triumphs

RL first gained massive publicity after the publication of Mnih et al. (2013) and Mnih et al. (2015), where a general agent was able to successfully conquer many ATARI video games using image inputs alone. Unfortunately, such games are near-deterministic and the state space was sufficiently small allowing even rules-based methods to be feasible in such systems (though no previous algorithms can *learn* the games through a general algorithm). Although the algorithm was impressive, previous methods were also able to approach the ultimate performance achieved. To conquer a task never done before, Silver et al. (2016) and DeepMind (2016a) were published in early 2016. The two studies introduced a RL algorithm to conquer Go, a board game invented more than 3000 years ago in China. Go is known as the most challenging game for AI due to its massive state and action space (more than 10^{170} possible states), and the requirement to defeat random opponents with different play styles. Modern AI solutions to Go struggle against even amateur players; however, AlphaGo was able to convincingly defeat the world's best Go player, Ke Jie. At the beginning, the algorithm used supervised learning to obtain fundamental knowledge from amateur level players. Then, expert level players were used to learn advanced strategies. After confidently surpassing the experts, the agent continued to perfect itself through self-play, ultimately becoming the world's best Go player (Silver et al., 2016; DeepMind, 2016a). Intuitively, these experiments demonstrate the potential of RL to identify hidden patterns and provide valuable contributions to modern engineering beyond what is already known.

The original AlphaGo contained human engineered features that were believed to aid the agent in learning. Interestingly, DeepMind believed the opposite. That is, DeepMind believed that the agent's skill was handicapped by said features; thus, leading to the publication of AlphaGo Zero (zero referring to zero human knowledge), a different version of AlphaGo without human bias (Silver et al., 2017b). In AlphaGo Zero, all human engineered features were removed, leaving only the locations of the black and white stones as states. Within 40 days of training, AlphaGo Zero, starting *tabula rasa*, was able to surpass the best performance ever achieved by AlphaGo through pure self-play, a feat only achievable through RL. Moreover, only 3 days of training was needed for AlphaGo Zero to achieve world championship level. The changes also made AlphaGo Zero more efficient, consuming less than 10% of power and using only 4 tensor processing units (TPUs) compared to the 48 used previously.

By late 2017, AlphaZero was released following the ideas of AlphaGo Zero, where a general agent taught itself how to play Chess, Shogi, and Go and was able to defeat the world champion program in each respective case (DeepMind, 2016b; Silver et al., 2017a; 2018). The world's best Chess player in history, Magnus Carlsen, had a peak FIDE ELO (measurement of skill assigned by FIDE, a world renowned Chess organization) of 2882. Using traditional methods such as supervised learning, the ideal AI would be capped at 2882, representing zero replication error. Within just 200,000 training steps, AlphaZero was able to achieve an ELO above 3300 from pure self-play. Within 300,000 steps (4 hours physical time), AlphaZero surpassed the world's best Chess engine, Stockfish (Chabris, 2015). Comparing the two Chess engines, Stockfish required decades to refine by expert engineers. AlphaZero was simply initiated *tabula rasa*, and after 4 hours, it became the best. Most impressively, AlphaZero demonstrated RL's ability for long-term decision making, that is, it sacrificed many pieces in the early game to obtain a significant advantage in the end game some thirty steps in the future. Furthermore, AlphaZero only searches 10^4's moves per turn compared to traditional Chess engines that searches up to 10^7's moves. Moreover, the same algorithm was used to learn Shogi and Go and was able to defeat the respective best game engines, Elmo and AlphaGo Zero.

The achievements of RL up until this point are nothing short of amazing; however, all previous applications assumed a perfect information system where all system states are perfectly observable. For example, you cannot hide the location of your pieces in Chess from your opponent. Additionally, large amounts of time were allowed for the computer to find the optimal action. Unfortunately, systems in the real world may contain fast dynamics and are littered with incorrect, unmeasurable, and/or unreliable information. To demonstrate RL's ability to perform in a real time partially measurable settings, AlphaStar was released in late 2018 (DeepMind, 2019; Vinyals et al., 2019). Here, the agent played a game called StarCraft II, a real-time strategy game where the player is the general of an army and is tasked with building various structures for military, resource, or energy needs in order to ultimately defeat the opponent. Compared to previous games, StarCraft poses a highly challenging (most humans would find it difficult to play) and real time environment where the opponent's moves are hidden and unknown. Additionally, the number of states and actions are near infinite. Here, the agent must evaluate fast enough to win real time battles while managing the required resources of its army. After training, AlphaStar was able to decisively defeat two of the best StarCraft II players using pixel inputs alone. Moreover, AlphaStar was not given any information unavailable to players and was handicapped to be at human level (e.g., the agent cannot perform thousands of actions per second, etc.). AlphaStar showcased RL's ability to react to unexpected situations, real-time high dimensional action selection, and hierarchical long-term planning. Such characteristics could be very applicable in industrial process control for fault-tolerant control or high dimensional multi-variate optimal control.

All of the above applications assumed a single agent environment. In industrial process control, the agent must also understand the consequences of its actions on the overall system. RL's abilities in multi-agent partially measurable systems was first demonstrated on DotA 2 by OpenAI. DotA, like StarCraft, is a real-time

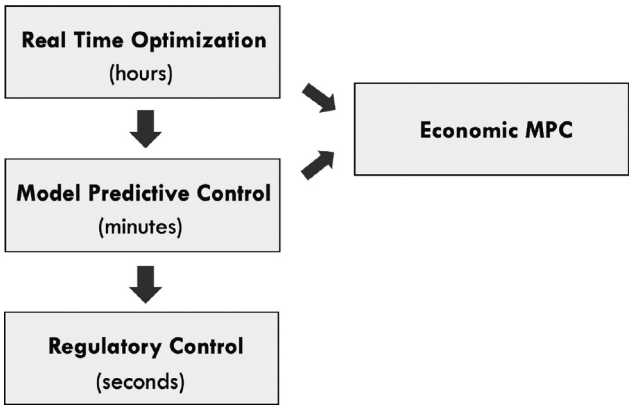


Fig. 10. The traditional control architecture.

high dimensional strategy game where each team tries to defeat the opponent. Unlike StarCraft, there are five players per team; therefore, the agent's interaction effects with other agents must also be considered. Additionally, the time horizon per game can be up to 80,000, a dramatic increase compared to Chess or Go, which typically ends within 150 turns (OpenAI, 2018b). At each time t, the agent observes 20,000 continuous observations and has access to 1000 different actions. The reward function of the agent contains two components: individual performance and team performance. To enhance cooperation among the independent agents, a separate hyper parameter called *team spirit*, denoted here as ϕ, was used to specify the importance of the individual reward function compared to the team reward function. Throughout the game, team spirit will be annealed from 0 to 1 to communicate that in the end game, only the team reward function matters. The reward function is given as:

$$r(x,u) = \phi \cdot \text{team reward function} + (1-\phi) \cdot \text{individual reward function} \quad (41)$$

As of April 2019, OpenAI Five was able to defeat the best DotA 2 teams in the world (OpenAI, 2018a).

State-of-the-art RL research was first applied to near-deterministic low dimensional systems, eventually transitioning to complex video games that reflect the uncertain and stochastic nature of the real world. RL was shown to effectively handle partially measurable, long horizon, and high dimensional systems. Additionally, RL can quickly react to unexpected situations, learn to behave optimally in a team environment and is feasible for real-time applications with fast dynamics. Most importantly, RL was shown to be a general algorithm that can be used for different applications. Such characteristics hold huge implications for useful applications in industrial process control.

3.2. Comparison with common advanced control frameworks

A typical control framework is shown in Fig. 10. Real-time optimization (RTO) situates in the top layer and solves complex steady-state optimization problems to find the optimal steady states with respect to a desired performance metric (Huesman et al., 2008). Typically, this layer evaluates on the hourly time scale. The optimal steady states are then communicated to the MPC layer, where dynamic optimization is performed on a minutes scale to identify the optimal state and input trajectories to arrive at the optimal steady states. Typically, the objective function in this layer is given by:

$$J = \sum_{i=1}^{H} x_i^T Q_{mpc} x_i + \Delta u_i^T R_{mpc} \Delta u_i \quad (42)$$

due to its convex nature (Mayne and Rawlings, 2017). Here, H denotes the prediction and control horizon. Q_{mpc} and R_{mpc} are the tuning matrices for the state and input costs, respectively. Often times, plant managers do not allow for direct manipulation of the process actuators by MPC due to safety concerns. In these scenarios, the state trajectories are computed from the process model using the ideal input trajectory determined by the MPC and are communicated to the regulatory control layer, where PIDs are typically used to track the set points. On a simplified level, RTO identifies the optimal set points for economic objectives to be met while MPC finds the optimal trajectory to achieve the desired set points. More recently, researchers began to intertwine ideas from RTO with MPC where the economic objective of RTO is placed directly into the MPC objective function. Such a strategy is now known as economic model predictive control (EMPC) (Ellis et al., 2014).

The gain in optimality after the addition of each control layer is shown in Fig. 11. The objective of each additional layer is to increase the control optimality with respect to performance metrics. In typical processes, an optimal operating condition exists at a boundary. However, the optimal operating point cannot be achieved because of uncertainty and other factors.

For each control layer, there exists a set of distinct algorithms such as PIDs for regulatory control and a variant of MPC for the upper layers. Theoretically, the flexible nature of RL *may* enable it to potentially replace any of the above layers due to its general nature. For example, if RL were to replace MPC, the agent's reward function simply has to be the negative of Eq. (42), with the actions being the recommended set points. For the regulatory layer, the reward function can remain the same, but the action would result in direct control of the system's actuators. Finally, RL as an EMPC would have the economic objective added onto Eq. (42) as the reward function. However, proper design of the state space, action space, and reward function must all be considered during the design of the agent and will most likely be more challenging compared to implementing the simpler algorithms.

Perhaps one of the biggest advantages of RL is its rapid *online* computation time. Using most solvers, MPCs have a computational complexity of $O(H^3(n+m)^3)$, where n and m are the dimensions of the states and actions, respectively (Richter et al., 2012). Even after exploitation of problem heuristics, MPCs' computational complexity improves only to $O(H(n+m)^3)$ (Dunn and Bertsekas, 1989; Wright, 1997; Wang and Boyd, 2008). Therefore, the online computation time may be infeasible for large systems and/or for systems with long control horizons. On the other hand, RL's optimal policy is pre-computed *offline*, making online evaluation extremely fast. Such a method is very similar to the concept of parametric programming from explicit MPCs (Bemporad et al., 2002). Although, RL still has to be trained offline to identify the optimal policies and may require hundreds of thousands of interactions before reaching a near-optimal policy. For applications where offline computation time is not of concern, RL might be the preferred method.

Another significant difference is that RL is *model-free*. That is, a model is only required for initial training of the agent simply to reduce training time, and is not used during real-time implementation. In contrast, the identified model will be used exclusively online in MPC and can potentially lead to sub-optimal control. Offset free control is used in MPCs to overcomes offset errors in MPCs through online model modification (Pannocchia et al., 2015); however, it may not work well for very noisy processes. Furthermore, some complex process dynamics might be difficult to explicitly model and could increase online computation time. Adaptation in RL is conducted by changing the control policy directly and occurs after each control input. Model identification is not required and could be advantageous for systems where accurate process models are not mathematically identifiable. However, the continually adaptive control policy of RL is black box by nature and could pose

Table 4
A comparison between RL, MPC in literature, and industrial MPC software.

	Reinforcement Learning	Model Predictive Control	Typical Industrial DMC
Performance	Close to optimal	Optimal with perfect model	Close to optimal
Online comp. cost	Low	High	High
Offline comp. cost	Policy & model identification	Model identification	Model identification
Reliance on model	Only for training	At all times	At all times
Online calibration	Exploratory moves	Various methods	Exploratory moves
Sensitivity to tuning	Low	High	High

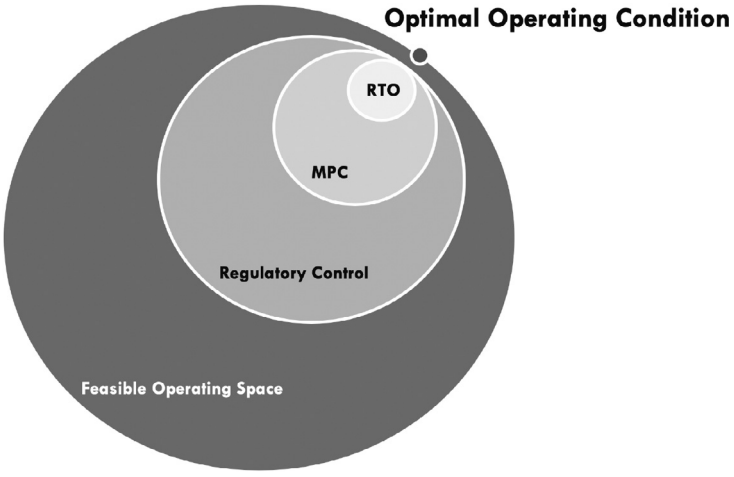


Fig. 11. Optimality of each control layer.

Fig. 12. The FLUIDMechatronix experiment from Turbine Technologies (Technologies, 2019).

safety concerns. Moreover, the speed of adaptation is a function of the learning rate and must be tuned. A low learning rate would render the adaptive feature meaningless. Contrarily, a high learning rate may result in unstable control of the process, especially in processes with poor signal-to-noise ratios because the agent will learn inaccurate system dynamics.

In terms of optimization window, MPC considers each future stage to be equally important when computing for the optimal input trajectory. On the other hand, traditional RL considers each consecutive stage to be of less value (due to the discount factor). Recently, Asis et al. (2020) published an RL algorithm formulated in a fixed-horizon fashion identical to MPC. In doing so, the authors also demonstrated the increased stability and effectiveness of the new algorithm.

Lastly, RL has very few hyper parameters in the tabular case; however, initial design of RL typically require process experts to configure. First of all, the state and action spaces must be properly configured to the regions of operation. Additionally, the reward function require careful design so unintentional behaviour does not occur. For example, if the reward function does not contain a cost for manipulating inputs, the agent may continuously change the input for noisy systems. This may lead to unwanted oscillation and equipment wear in the process. As for convergence of the control policy, as long as the learning rate is annealed to near zero, RL should converge sufficiently well for most cases.

A high level comparison between RL and MPC is shown in Table 4. In addition to the above comparisons, RL was also compared to industrial MPCs currently available on the market because they exist as proven technology, and not just academic studies. Most industrial MPCs are in the form of dynamic matrix control (DMC), an early variant of MPC. In any real applications, the process models are never perfect, resulting in a near-optimal solution. Additionally, the online computation time for DMC is high, especially for nonlinear systems. RL can perform at the same speed regardless of the system linearity. For online implementations, RL must explore random actions to adapt (an idea that sounds risky for live processes). Interestingly, industrial DMCs indeed performs random actions online for model adaption. Typically, DMCs are initialized in the *smart step* mode, where the system will perform random step tests to calibrate the identified model to the real process. Afterwards, the operators will switch the system to the *calibrate* mode. In calibrate mode, the system continues to perform step tests; however, they are much more infrequent and are lower in magnitude. Such a model adaptation technique is identical to RL, where exploration is plentiful initially, but is eventually annealed to near zero. Currently, one major flaw that could be preventing RL from industrial wide adoption is the non-interpretable nature of its control policy. For a more detailed, explicit comparison between

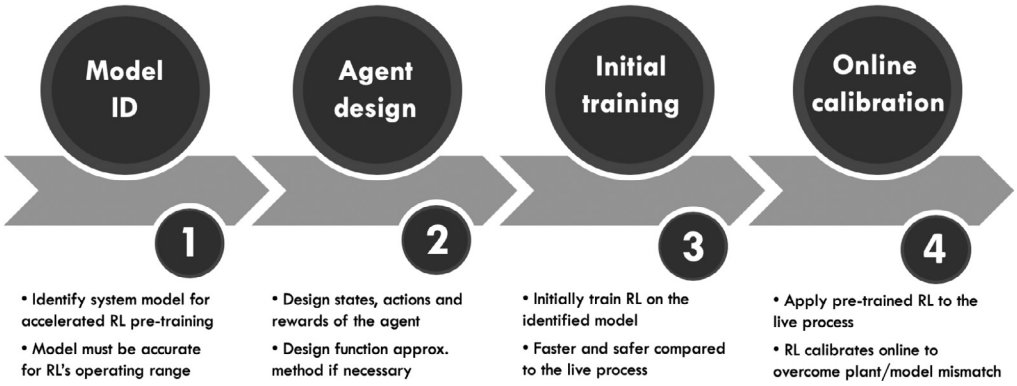


Fig. 13. General procedure for implementing industrial reinforcement learning.

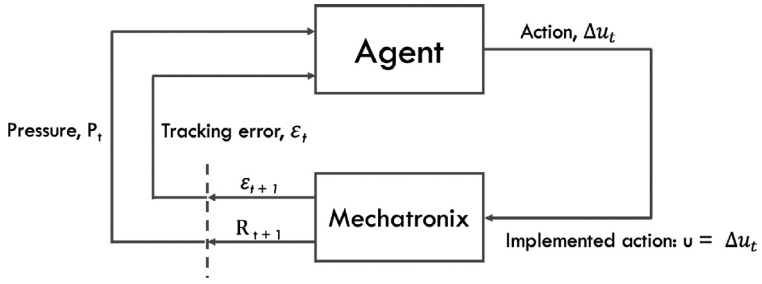


Fig. 14. The RL set-up for the FLUIDMechatronix experiment.

MPC and RL, see Gorges (2017). Although RL may appear to have many advantages compared to traditional optimal controllers, RL literature is still embryonic compared to MPC and lacks solutions to many fundamental problems. A list of shortcomings currently barring RL from industry wide adoption is provided in Section 4. A summary of RL compared to literature MPC and industrial MPC is shown in Table 4.

3.3. A RL control experiment: Optimal control of a pump system

This section introduces an illustration on how to implement RL onto a pilot-scale industrial process control system[2]. A tabular *Q*-learning RL agent will be implemented onto the FLUIDMechatronix system from Turbine Technologies (machine shown in Fig. 12) for set-point tracking. The system is equipped with a variable frequency drive to regulate the output pressure. The output pressure, *P*, and pump RPM (manipulated input) will be used for this example. The operating ranges of the pressure and pump RPM are:

$$0\ kPa \leq P \leq 45\ kPa$$

$$0\ Hz \leq RPM \leq 60\ Hz$$

A FOMDP will be used to describe this system because the system measurements are available and system dynamics are fast. The initial conditions of the system are given by:

$$P_0 = 41\ kPa \tag{43}$$

$$RPM_0 = 60\ Hz \tag{44}$$

The steps required for implementation of RL algorithms are shown in Fig. 13. The RL set-up for this example is shown in Fig. 14. The RL agent will track the pressure set-point by changing the pump RPM on the single-input single-output (SISO) system. Specifically, the state and action of the agent are the current set-point tracking error:

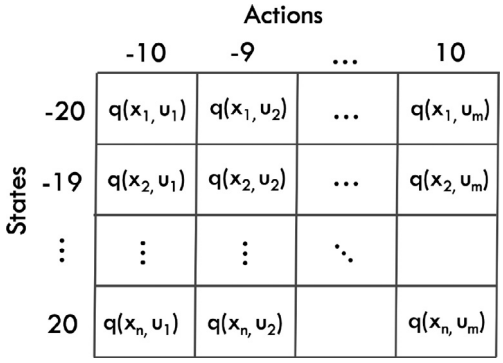


Fig. 15. *Q*-matrix of the Mechatronix system.

$$\varepsilon = P_t - P_{t,sp} \tag{45}$$

and the change in pump RPM Δu, respectively. In Eq. (45), P_t denotes the pressure at time t and $P_{t,sp}$ denotes the corresponding pressure set-point. This use of the change of RMP as the action allows the agent to track multiple set-points. If the action was the pump RPM instead, then the agent can only track one set-point since it simply maps different tracking errors to the steady state pump RPMs. The reward function of the agent is given by:

$$r(x, u) = \max(-\varepsilon^2 - |\Delta u|, -200) \tag{46}$$

where Δu is the change in input (i.e., changing the input has a cost). Additionally, the reward is capped to -200 to avoid numerical issues and unexpectedly large bootstrapping errors. The agent will evaluate every five seconds to ensure that the system has reached steady state before the consecutive action is made. Five-second was selected because it was the longest observed transition time

[2] Supplementary code for results generated in this section can be found here: https://github.com/RuiNian7319/Research/tree/master/2.RL_Codes/Mechatronix

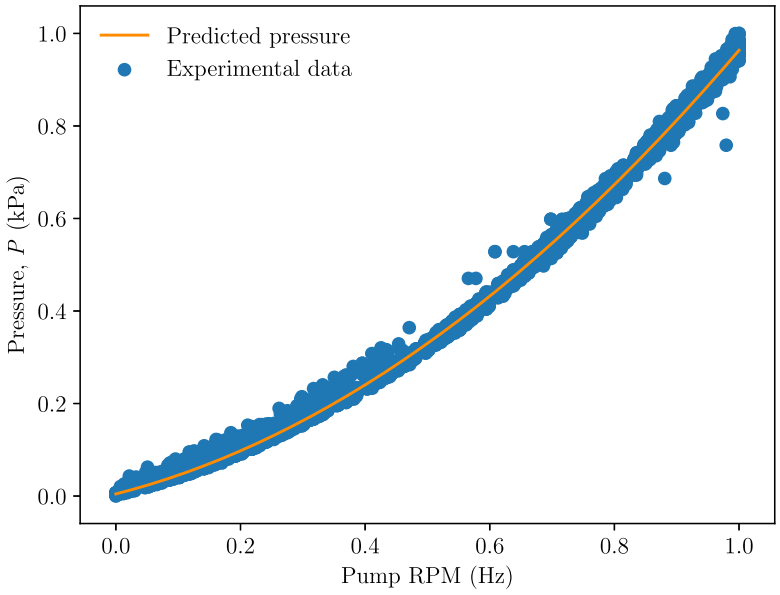


Fig. 16. Performance of the identified system model on a test data set.

Table 5
Summary of the agent's hyper parameters in the Mechatronix experiment.

Hyper Parameter	Value
States, x	$\varepsilon = [-20, -19, \ldots, 20]_{1\times 41}$
Actions, u	$\Delta u = [-10, -9, \ldots, 10]_{1\times 21}$
Reward, r	$\max(-(\varepsilon^2 + \lvert\Delta u\rvert)$, -200)
Learning rate, α	[0.001, 0.7]
Discount factor, γ	0.9
Exploratory factor, ϵ	[0.1, 1]
Evaluation interval	5 seconds
System representation	FOMDP

required for the system to reach steady state. The very short dynamic transition is not to be considered in this experiment.

The hyper parameters of the agent are summarized in Table 5. In this example, the states and actions are discretized as:

$$x = [-20, -19, \ldots, 20]_{1\times 41} \tag{47}$$

$$u = [-10, -9, \ldots, 10]_{1\times 21} \tag{48}$$

and the Q-matrix corresponding to the action-value functions is given in Fig. 15. Initially, all action-values are initiated as 0. The states and actions, x and u, correspond to ε and Δu, respectively. The discount factor, γ, of the agent was 0.9. Overall, the agent was trained for 2,000,000 time steps corresponding to approximately 23 days of continuous operating experience. After every 400^{th} time step, the agent was reset back to the initial states given in Eqs. (43) and (44) to prevent controller saturation during initial episodes. Such a long continuous training time on a live process is unreasonable; therefore, a crude model of the system was first identified and was used to initially train the agent in simulation. The identified model had a mean squared error of 0.056 on the experimental data and is given as:

$$P_t = 0.012 \cdot RPM^2 + 0.024 \cdot RPM - 2.073 \tag{49}$$

The model fit on a normalized test data set is shown in Fig. 16.

Exploration-wise, the agent starts with an equiprobable random policy ($\epsilon = 1$), which decays linearly until $\epsilon = 0.1$ by the $500{,}000^{th}$ time step. Likewise, the learning rate, α, is initiated at 0.7 and decays linearly until 0.001.

The agent will behave as follows: initially, the agent observes ε_t and performs action Δu_t, that is, the agent observes the current tracking error and then changes the previous input by some amount. Next, the input signal corresponding to $u_t = u_{t-1} + \Delta u_t$ will be sent to the Mechatronix experiment. After waiting for five seconds to ensure the system reached steady state, the agent will receive reward R_{t+1} and observe the new tracking error, ε_{t+1}. Then, the agent uses Eq. (29) to update its current knowledge, and the cycle starts anew. A numerical example is provided below:

Suppose the agent discretized the system into five states and three actions corresponding to:

$$x = [-21, -10, 0, 10, 21]_{1\times 5}$$

$$u = [-10, 0, 10]_{1\times 3}$$

The Q-matrix was initialized as:

$$Q(x,u) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

where rows 1–5 correspond to the five states and columns 1–3 correspond to the three actions. After the initial set-up, the agent has to be trained. RL agents are typically trained through a series of episodes, where each episode consists of multiple update steps. In this example, the agent will be trained through 1000 episodes, where each episode consists of 2000 seconds (2,000,000 seconds of total training time). Furthermore, the Q-matrix is updated after every 5 seconds, resulting in 400 update steps per episode. At the end of each episode, the agent will be reset to its initial states. Note that simulations were used for initial training. Total simulation time was only a few minutes for the entire 1000 episodes.

The initial set-point of the system was set to 30 kPa. At $t = 0$, the system was at steady state with 15 kPa and 37 RPM, resulting in $\varepsilon = -15$. The agent receives the error and rounds it to the nearest state, $x = -10$. Given this state, the agent then picks the action that coincides with the highest Q value from the Q-matrix:

$$Q(-10, u) = [0, 0, 0]$$

where the first, second, and third value correspond to the current predicted Q values for selecting actions $\Delta u = -10, 0, 10$, respectively. Because the agent was not provided with any prior information about the system, the agent must first pick an action arbitrarily to identify more information. During scenarios where the highest Q values are identical, ties must be broken arbitrarily to avoid biasing one action over others.

If $u = -10$ was picked: the system will transition to a steady state of 7.4 kPa after five seconds and the new error would be -23. Clearly, this is a sub-optimal action but the agent was not equipped with prior knowledge of this. Additionally, the agent would receive reward:

$$r_1 = max(-23^2 - |-10|, -200) = -200$$

and $x_1 = -21$. From this newly learned knowledge, the agent would update the Q-matrix using Eq. (29):

$$Q(-10,-1) \leftarrow Q(-10,-1) + 0.7[-200 + \gamma Q(-21,0) - Q(-10,-1)]$$

$$Q(-10,-1) \leftarrow 0 + 0.7[-200 + 0.9 \cdot 0 - 0]$$

$$Q(-10,-1) \leftarrow -140$$

and the updated Q-matrix would be:

$$Q(x,u) = \begin{pmatrix} 0 & 0 & 0 \\ -140 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Notice here that all three u's are the maximizing action for $Q(x_{t+1}, u_{t+1})$; therefore, the ties are broken randomly here as well to avoid unnecessary bias. Suppose the first episode was terminated early and the system was reset back to $x_0 = -10$. This time:

$$Q(-10,u) = [-140, 0, 0]$$

meaning that picking $\Delta u = -10$ is a sub-optimal action compared to $\Delta u = 0$ or 10. Here, the agent would pick either $\Delta u = 0$ or 10, instead.

After traversing through the state space many times, the Q-matrix is now given by:

$$Q(x,u) = \begin{pmatrix} -278 & -231 & -202 \\ -66 & -59 & -48 \\ -31 & -22 & -33 \\ -52 & -62 & -68 \\ -209 & -244 & -291 \end{pmatrix}$$

This time, the agent has much more information about the system and can begin acting optimally. After, once again, resetting the agent back to 15 kPa and 37 RPM, the decision making of the agent this time around is deterministic. Given action values:

$$Q(-10,u) = [-66, -59, -48]$$

the agent would pick $\Delta u = 10$ corresponding to $Q(-10,10) = -48$ and transition the system to $P_1 = 25.6\,kPa$ and yield reward $r_1 = -(4.4^2) - |10| = -29.4$. Now, the system's state is closest to $x = 0$. In $x = 0$, the optimal action-value is $Q(0,0) = -22$. The new update step is given as:

$$Q(-10,1) \leftarrow -48 + 0.001[-29.4 + 0.9 \cdot -22 + 48]$$

$$Q(-10,1) \leftarrow -48 + 0.001[-1.2]$$

$$Q(-10,1) \leftarrow -48$$

Here, the α is 0.001 due to decay throughout the training process. Furthermore, the TD error is at -1.2, signifying that the agent's value functions are very close to optimal and the agent is well trained. All TD errors will eventually converge to near-zero and the agent's policy will become optimal.

After simulating the system for many steps, the reward obtained by the agent during the training phase is shown in Fig. 17. The reward never converged to zero because the lower bound of ϵ was set to 0.1, meaning that the agent continued to explore sub-optimal actions during training. As for the desired set-point during training, the agent's set-point is drawn from a Gaussian distribution $N(30, 5)$.

After 2,000,000 update steps, the agent was applied onto the real process and was tasked to track pressure set-points of 35 and 5. The pressure trajectory of the Mechatronix experiment is shown in Fig. 18a and b. The mean squared tracking error (MSE) for set-points of 35 and 5 are 14.2 and 15.5, respectively. In both cases, the initial pressure of the system was about 5.5 kPa above the desired set-point to ensure fair MSE calculations. The agent behaves much like a linear self-tuning PID where the RL maps tracking errors to changes in input. Such a set-up only works well locally for nonlinear systems because the system is locally linear. As the agent moves away from the local linear region, the performance deteriorates significantly, as shown in Fig. 18b. The agent's performance is significantly better when tracking $P = 35$ because the set-points during training were biased towards the upper range. Additionally, it can be seen in Fig. 16 that the controller gain changes significantly at lower pressures, making the optimal policy for the higher pressure range sub-optimal in the lower pressure range. There also exists a large off-set between the set-point and pressure trajectory. This is caused by the discretization error; there is no action Δu that can get the system to exactly $P = 35$ *or* 5 *kPa*. To overcome this, the action space can be more finely discretized, but will increase the training time and space complexity required by the agent. Another simpler way will be introduced later on in this example.

One simple way to extend the current agent to nonlinear systems is to approximate the system using piecewise linear functions as shown in Fig. 19. Communication wise, the agent will receive this information through a *second state*:

$$x^{(2)} = \begin{cases} 1, & \text{if } P \le 10 \\ 2, & \text{if } 10 < P \le 20 \\ 3, & \text{if } 20 < P \le 30 \\ 4, & \text{if } 30 < P \le 50 \\ 5, & P > 50 \end{cases}$$

and the new state space for the agent is given by:

$$x = [(-20,1), (-20,2), (-20,3), \ldots, (-19,1), \ldots, (20,5)]_{1\times 205} \tag{50}$$

where 1, 2, 3, 4 and 5 for the second state correspond to the region the agent is currently in. Here, the Q-matrix will instead be initialized as $0_{(41\,\cdot\,5)\,\times\,3}$ to accommodate for the second state. Intuitively, the agent now observes the current tracking error and the region in which this tracking error has occurred. Since each region is locally linear, the control law is also linear allowing the agent's policy to be applied along each region (Seborg et al., 2013). Such an idea is similar to linear parameter-varying models or multimode models. Here, the agent's policy changes depending on the region it is currently in.

After training the new agent for 2,000,000 steps, the agent was again implemented onto the Mechatronix experiment. The new pressure trajectories for tracking $P = 35$ and 5 are shown in Fig. 20a and b with MSEs of 14.2 and 12.5, respectively. It can be

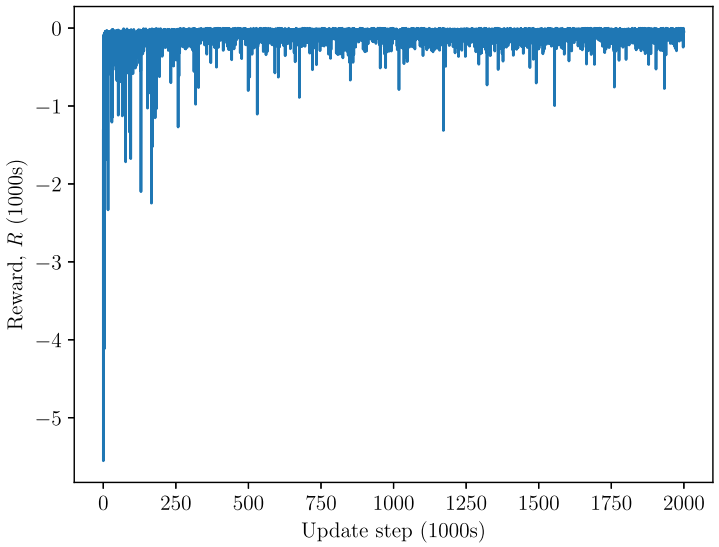
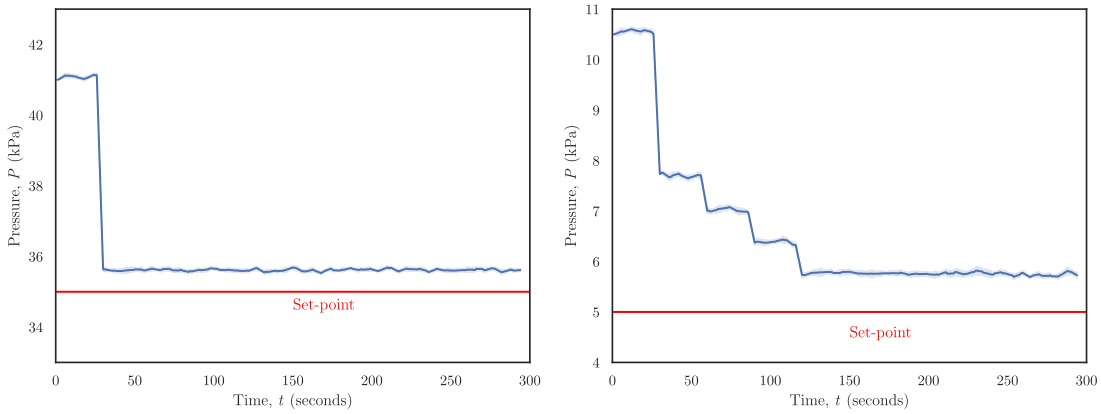


Fig. 17. Loss curve of the agent during training.



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Fig. 18. Pressure trajectory of the Mechatronix experiment. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

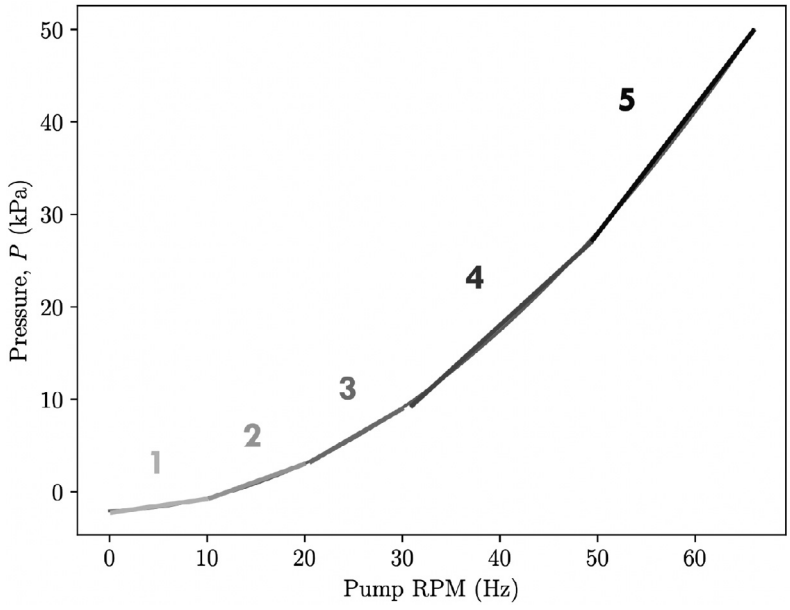
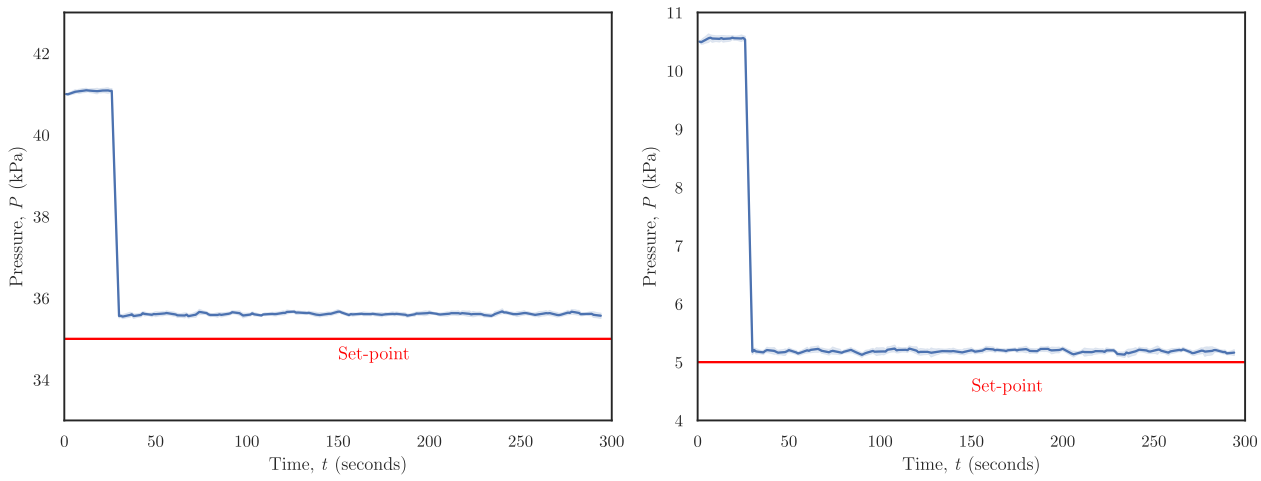


Fig. 19. Approximating the nonlinear Mechatronix system.



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Fig. 20. Pressure trajectory using the nonlinear agent. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

Table 6
A comparison between RL, MPC in literature, and industrial MPC software.

	Normal Q-learning	Two-state Q-learning	Two-state interpolated Q-learning
MSE (High/Low SP)	14.2 & 15.5	14.2 & 12.5	13.6 & 11.7
Offset	Yes	Yes	No
Nonlinear	No	Yes	Yes

seen that the performance for tracking the lower set-point has dramatically improved; however, the offset still exists.

Because the nonlinear system was approximated using linear components, both the system and control law are locally linear. Using this characteristic, the optimal control action can be interpolated using the linear interpolation equation (Meijering, 2002):

$$u = u_{low} + (x - x_{low})\frac{u_{high} - u_{low}}{x_{high} - x_{low}} \tag{51}$$

where x is the actual state from the system; typically, x is not included in the discretized states $\mathcal{X}$ and is instead between x_{high} and x_{low}. For example, suppose the discretized states are $x = [0, -5, -10]$. If the current state is -3, x_{high} and x_{low} would correspond to 0 and -5, respectively. Likewise, u_{high} and u_{low} correspond to the greedy action (i.e., return maximizing action) for x_{high} and x_{low}, respectively. For example, given the action space $u = [-5, 0, 5]$ and $Q-$matrix:

$$Q(x, u) = \begin{pmatrix} -5 & 2 & 1 \\ 4 & 1 & -2 \\ -2 & 0 & 3 \end{pmatrix},$$

and u_{high} and u_{low} would be 0 and -5 (actions corresponding to the index of the highest Q-value), respectively. The optimal action for $x = -3$ would be:

$$u = -5 + (-3 + 5)\frac{0 + 5}{0 + 5}$$

$$u = -3$$

By adding the interpolation technique onto the two-state RL agent (without re-training the agent), the new pressure trajectories are shown in Fig. 21a and b with new MSEs of 13.6 and 11.7, respectively. Now, the off-set is eliminated and the system can operate optimally.

A summary of the simple RL solutions and their respective characteristics are shown in Table 6. In this illustration, implementation of a simple RL agent onto a pilot-scale industrial experiment was introduced. Techniques to extend the agent's ability to nonlinear systems and for off-set free control were also shown. The agent's performance on the live systems were replicated 10 times for each algorithm to ensure reproducability; the resultant standard deviation in the pressure trajectories was very narrow, representing high reproducability.

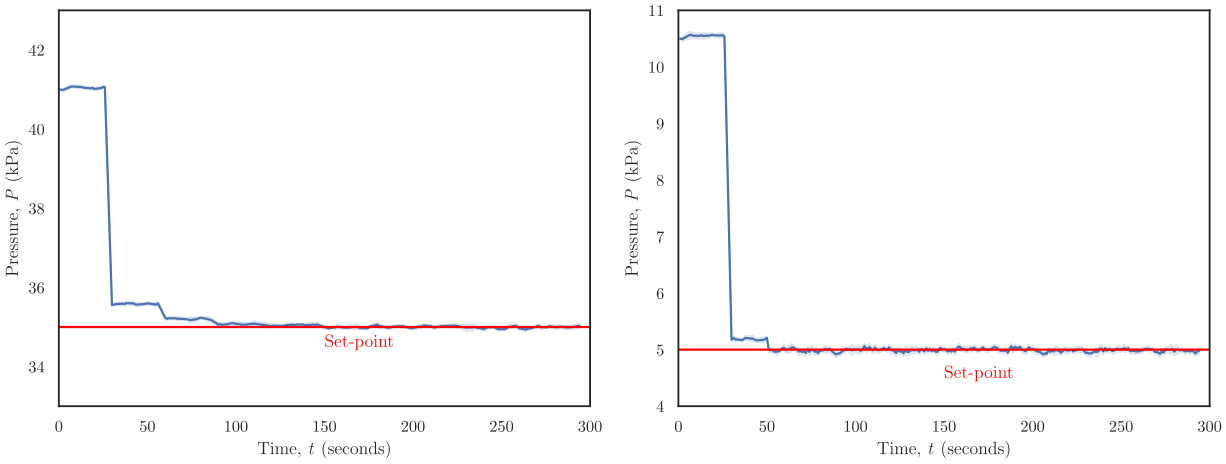
RL agents typically only provide the control action for the immediate future. This is similar to traditional methods like MPC where only the first input is used during each step; however, MPC is implemented in a receding horizon fashion and also calculates an input trajectory for future steps making open-loop control possible for short horizons. RL can also be implemented in such a way. Such RL methods are called planning methods or model-based RL and require a model of the system. In receding horizon RL, the agent outputs the immediate control action, uses the model to identify the next state, identifies the optimal control action for the next state, and continues the cycle thereforth.

Because the example shown here is for illustration purposes, the example is simple, evaluates at a preset intervals and does not consider system dynamics or unmeasured states. For systems with variable transition times and system dynamics consideration, SMDPs should be used. The SMDP Q-learning algorithm is given by Bradtke and Duff (1994):

$$Q(x, u) \leftarrow Q(x, u) + \alpha\left[\frac{1 - e^{-\beta\tau}}{\beta} r(x, x_{t+1}, u) + e^{-\beta\tau} \max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x, u)\right] \tag{52}$$

where $r(x_t, x_{t+1}, u)$ is the reward rate and is given in Eq. (10). For systems with unmeasured states, concepts of POMDPs provided in Section 2 are required.

If a deep RL algorithm, such as DDPG, was used in this particular example, the major difference would be the design of the state and action spaces. In DDPG, the two states and control action



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Fig. 21. Pressure trajectory of the nonlinear agent using interpolation action selection. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

would instead be continuous and be given by:

$$x_1 \in [-20, 20]$$

$$x_2 \in [0, 60]$$

$$u \in [-10, 10]$$

In this set-up, there would naturally no discretization error because both the states and actions are continuous. However, note here that the search space for the optimal policy is much larger compared to the tabular case and will require significantly more time before a near-optimal policy is found.

3.4. Automated PID tuning

Proportional-Integral-Derivative (PID) controllers are widespread throughout industry due to their effectiveness and ease of implementation. The general PID formulation is given by Seborg et al. (2013):

$$u(t) = K_p \varepsilon(t) + K_i \int_0^T \varepsilon(t)dt + K_d \dot{\varepsilon}(t) \tag{53}$$

where $\dot{\varepsilon}(t)$ is the derivative of the error at time t, K_p, K_i, and K_d are hyper parameters corresponding to the proportional gain, integral gain, and derivative gain and should be well tuned for acceptable controller performance. However, depending on the process to be controlled, this tuning process may be difficult and time-consuming, especially in MIMO systems where control loops are intertwined (i.e., tuning of one control loop results in de-tuning of another). Many methods exist for initial tuning, such as the Ziegler-Nichols method. But in most cases, the final controller performs well below optimal, especially in industry where engineers are time constrained (Howell and Best, 2000). Instead, an RL agent can be used to automatically and optimally tune the PID parameters.

One of the earliest studies on automated PID tuning using concepts from RL was published by Howell and Best (2000) in 2000 (architecture shown in Fig. 22) where the authors automated the tuning of a Ford Motors Zetec engine. The algorithm here, named Continuous Action Reinforcement Learning Automata (CARLA), was used to fine tune PIDs after initial parameters were set using methods like Ziegler-Nichols. Results showed a 60% reduction in the cost function after RL tuning. CARLA works as follows: Initially, each system parameter is associated with one CARLA. The action of each CARLA is the recommended system parameter and is drawn from

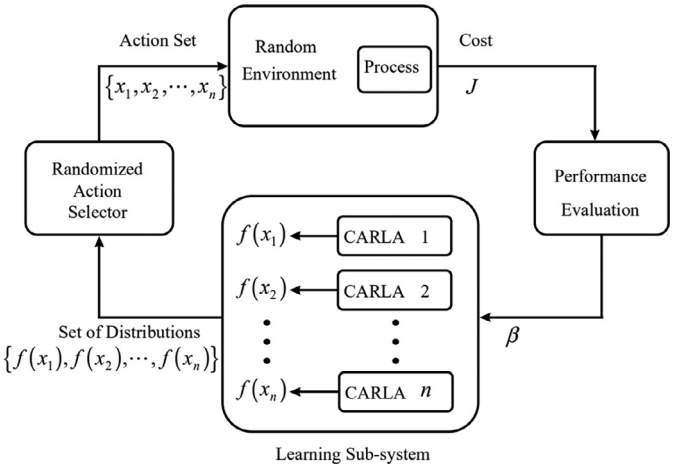


Fig. 22. CARLA: An RL-powered automatic PID tuning algorithm.

a corresponding probability density function, $f(x)$. For example, in a system with one PID, there would be three CARLAs corresponding to the three PID parameters. The action of each CARLA is the recommended parameter and is implemented into the process. Afterwards, the performance of the system with respect to the desired performance metric is observed. Performances better than the mean performance will cause the distribution means to be shifted towards the recommended parameters, vice versa for lower performances. Exploration in CARLA is conducted in a similar way as DDPG, except Gaussian white noise is injected into the action rather than the Ornstein-Uhlenbeck noise. More detailed information regarding the CARLA algorithm can be found in Howell and Best (2000). Such a method is simple, but may not be scalable to large MIMO systems due to the vast numbers of CARLAs required.

By 2006, Wang et al. (2006) developed a more advanced PID tuning method through utilizing an actor-critic RL algorithm. In this algorithm, the agent's states are given as:

$$x = [\varepsilon_t, \Delta\varepsilon_t, \Delta^2\varepsilon_t]$$

$$\Delta\varepsilon_t = \varepsilon_t - \varepsilon_{t-1}$$

$$\Delta^2\varepsilon_t = \varepsilon_t - 2\varepsilon_{t-1} + \varepsilon_{t-2}$$

and the actions are:

$$u = [K_i, K_p, K_d]$$

Notice that the three states correspond to the integral, proportional, and derivative errors of the discrete PID formulation. The RL formulation here maps the current error, and the first- and second-order difference of errors to some optimal PID parameters at each time t (i.e., the PID parameters change at every time t). Intuitively, the agent observes the current errors and outputs the optimal PID parameters at current time t. The PID is then re-parameterized using these new parameters and is used to calculate Δu_t:

$$\Delta u_t = K_i \varepsilon_t + K_p(\varepsilon_t - \varepsilon_{t-1}) + K_d(\varepsilon_t - 2\varepsilon_{t-1} + \varepsilon_{t-2}) \quad (54)$$

where K_i, K_p, and K_d are outputted by the agent and may change for each time t. From Δu_t, u_t can be calculated by:

$$u_t = u_{t-1} + \Delta u_t$$

From the original paper, simulation results showcased the algorithm's adaptability, robustness and ability to perfectly track complex nonlinear systems. By 2008, Sedighizadeh et al. (2008) applied this adaptive PID to a wind turbine experiment, yielding near perfect tracking performance in the industrial application. Complete applicational details can be found in Sedighizadeh et al. (2008). In 2015, the algorithm was implemented again, this time on an under-actuated robotic arm (Akbarimajd, 2015). The robotic arm has fast dynamics and lacks adequate actuators for ideal control. Such a study allows for the exploration of the RL tuned PID's fault tolerant characteristics. In the experiments, the robotic arm's goal was to maintain a formation, but was exposed to many disturbances. Traditional control methods typically lead to overshooting and other undesired behaviours; however, the RL tuned PID did not exhibit any such behaviour and showed significantly better performance in terms of disturbance rejection and response time compared to traditional approaches. By 2017, a Q-learning variant of this proposed algorithm was used in Gurel (2017) to tune a race track robot. Compared to manually tuned PIDs, the RL tuned PID robots achieved up to 59% faster lap times.

In 2010, Brujeni et al. (2010) leveraged the SARSA RL algorithm to dynamically tune a PI-controller used to control a continuous stirred tank heater. The agent was first pre-trained on an estimated model for the process and was then implemented to continuously tune the tank heater online. The agent aimed to reject disturbances and track the set-point. In the end, the authors compared the performance of the RL tuned controller against internal model control tuning methods. It was found that RL was the superior tuning method due to its continuous adaptive nature.

In 2013, Hakim et al. (2013) implemented an automated tuning strategy similar to the one proposed above for a multi-PID soccer robot. The agent's states were altered. Instead of receiving the error signals, the agent received the state it currently resides in for the soccer game. Intuitively, this allowed the agent to understand its current situation, and tune its characteristics accordingly. For example, the robot may require faster speed while running down the field compared to when it is ready to take a shot. In an industrial setting, such ideas may be useful for an event triggered control system. For example, if the weather conditions are poor, the control system should be more conservative and have less gain. Ultimately, the paper demonstrated the superior performance of the RL tuned robots compared to robots tuned using the Ziegler-Nichols method.

On the more advanced side, RL was also shown to be effective in a model-based PID tuning strategy where the controller was tuned based on a finite horizon cost. Ultimately, this method was found to work on nonlinear MIMO systems with arbitrary couplings. The method was tested on Apollo, a real-life robot with imperfect low-level tracking controllers and unobserved dynamics. More details regarding this method can be found in Doerr et al. (2017).

Over the past 3 years, many more automated PID tuning methods using RL were published and are not all presented here. The ideas are very similar to the ones presented above with only slight alterations of the agent set-up.

3.5. Various simulated process control applications

One key advantage of RL in optimal control probably is its direct adaptive characteristic. Optimal control methods aim to extremize the functional equation of the controlled system and have shown to be less tractable both computationally and analytically compared to tracking or regulations problems. Consequently, adaptive optimal control has received relatively less attention, with existing studies mostly focusing on indirect methods (Sutton et al., 1991). Sutton et al. (1991) showed that RL can overcome this dilemma by serving as a direct optimal control method. Here, indirect methods refer to process re-identification methods whereas direct methods alter the control policy directly. Direct adaptive optimal control could be especially useful for systems where accurate models are not available. In such scenarios, RL can update the control policy directly through interactions with the system, eventually adapting to the optimal policy. This was shown in Moriyama et al. (2018), where the authors applied an RL algorithm to a data-center cooling application where accurate system models are very difficult to identify even with sufficient data. However, the RL agent was able to find an optimal policy to control the system after sufficient online interactions. The agent resulted in 22% reduced power consumption compared to previous model-based methods. In another work, Raju et al. (2015) showed that RL was able to adapt to changing load fluctuations in power systems, eventually resulting in optimal control after sufficient online interactions. Wireless networking is another system with non-identifiable dynamics. In Fan et al. (2019), the authors achieved increased energy efficiency using deep RL by allowing the agent to learn the optimal policy online instead of mathematically modelling the system.

For general applications of RL in simulated control environments, Hoskins and Himmelblau (1992) was perhaps the first instance where reinforcement learning was used for process control (in a set-point tracking sense). The authors trained a neural network based agent to control a CSTR. More recently, Spielberg et al. (2017) showed that DDPG can be used to successfully control arbitrary SISO and MIMO systems so long as the reward functions are properly formulated. In Spielberg et al. (2017), the agents mapped states, $x = [y_t, y_{sp}]$, to actions $u = [u_t]$. In Wang et al. (2017), an actor-critic RL method was applied to control the temperature of a building heating, ventilation, and air conditioning (HVAC) system, resulting in 2.5% reduction in energy usage and 15% increase in thermal comfort. The same HVAC system was also optimized using a proximal actor-critic RL agent in Wang et al. (2018). All previous applications formulated the agent to perform set-point tracking; however, RL is very flexible and can be used for optimal control (i.e., optimize an economic objective) by changing the reward function to be in terms of an economic objective. The viability of RL has also been shown in fault-tolerant control (FTC) (Nian et al., 2019). It was illustrated that RL agents were able to mediate faults and were able to adapt to changing operating conditions.

3.6. RL and chemical engineering

In as early as 2005, Lee and Lee (2005) investigated two approximate dynamic programming (ADP) algorithms, J-learning and Q-learning, for optimal control of a CSTR. In both cases, the learning algorithm attempted to identify the optimal policy offline.

Compared to MPC, these methods were shown to have lower online computational burden and the ability to control model extrapolation when computing for the optimal control actions. The performance of these algorithms were then compared to proportional-integral controllers, a successive linearization MPC, and nonlinear MPC on a CSTR. First, an nonlinear auto-regressive with exogenous input (ARX) model was identified for the CSTR. Then, each controller was applied in simulation to control the process. In the end, the performance of the two ADP methods were superior compared to its counterparts. In this particular example, MPC struggled to achieve closed-loop stability due to large model extrapolations during online optimization.

Joy and Kaisare (2011) built upon previous findings and applied the *J*-learning and *Q*-learning techniques to an adiabatic plug flow reactor. In this example, the plug flow reactor was modelled using three partial differential equations. Two different PIDs and three different linear MPCs were applied onto the system along with the ADP control algorithms. In the simulation cases presented, the ADP control algorithms were able to achieve superior control performance compared to the PIDs and linear MPCs.

By 2018, Sidhu et al. (2018) demonstrated the capabilities of an ADP method to optimally control the proppant concentration for hydraulic fracturing. It was shown that traditional optimal controllers were not ideal due to the high sampling time and large-scale nonlinear system. The proposed ADP controller was first trained on a simulation model of the reservoir to gain preliminary process insight. After training, the ADP optimal controller was used to generate an online pumping schedule while being able to handle plant-model mismatch within the rock formation.

3.7. Electricity optimization at Google data centers

One of the only, and most impressive in terms of achievement and value creation, live implementation of reinforcement learning[3] was achieved by Google DeepMind where the agent succeeded in reducing electricity usage of Google data centers by up to 40%. Indirectly, this also reduced the carbon footprint of all companies using Google's services. Services such as Google Search, Gmail, and YouTube, are all ran on servers powered by Google's data centers and generate enormous amounts of heat. Consequently, the data centers' primary source of energy usage is cooling. Cooling is accomplished using industrial equipment such as heat exchangers, pumps, and cooling towers. The difficulty comes from the problematic dynamics of the environment caused by DeepMind (2016c):

1. Highly complex, multi-dimensional environment with numerous nonlinear interactions. Such an environment renders traditional system identification approaches ineffectively. Additionally, human operators simply cannot intuitively understand all the complex interactions.
2. Highly variable internal and external conditions (such as weather, server load, etc.) rendering rules- and heuristics-based approaches fruitless.
3. All data centers have unique set-ups, requiring custom-tuned models for each individual environment. Such a dilemma requires an artificial general intelligence set-up where one algorithm can learn many different scenarios.

To overcome this, DeepMind research scientists first used historical operating data from the data centers to train neural network models for different operating conditions. The inputs to the neural networks were sensor information such as temperature, pump speeds, etc., and the output was the power usage effectiveness (PUE) defined as:

$$PUE = \frac{\text{Total building energy usage}}{\text{IT energy usage}} \tag{55}$$

The neural networks were used as simulators for the physical data centers. Different agents were trained on different data centers and during different operating conditions to minimize the PUE over a long hoirzon. Initially, only recommendations were provided by the algorithm. The PUE of a data center with and without implementing the agent's recommendations is shown in Fig. 23. By 2018, the agents were allowed to fully control the data centers after safety modifications were added. Specifically, a RL agent obtains measurements from sensors throughout the data center once every five minutes and replies with the optimal inputs that satisfy a robust set of safety constraints (DeepMind, 2018). The inputs are then verified with the local control operators to ensure that the system remains within constraint boundaries. Over the first few months of live implementation, the agents were able to successfully reduce electricity consumption by an average of 30% and are expected to get better as they learn online.

Table 7
Time required for RL agents to learn tasks.

RL algorithm	Approx. time required
OpenAI Five	Hundreds of years
AlphaZero	10000/s of games
DQN	Days of continuous play
DDPG	Millions of steps

3.8. Sequential anomaly detection

Anomaly detection is another field where reinforcement learning has gained traction. In industrial processes, anomaly detection is a proactive risk management strategy to identify and localize potential hazards before loss incidents occur. RL anomaly detection's sequential (time-series) nature and ability to self-learn provide an attractive edge. One of the earliest papers regarding RL-based anomaly detection was presented in Cannadey (2000) (architecture shown in Fig. 24), where the author used concepts of reinforcement learning to build an adaptive neural network to learn and identify new cyber attacks. The architecture was similar to time-series anomaly detection. The system was represented as a POMDP where the agent optimally mapped the observations $o = [x_{t-n}, x_{t-n+1}, \ldots, x_t]$ to actions $u = [\text{Normal}, \text{Anomalous}]$, guided by a scalar reward. Observations were an augmentation of past states to give the agent access to time-series information. Correctly identifying anomalies yielded +1 reward while any misclassification yielded -1 reward. More recently, Zighra (a online security company) deployed the proposed algorithm from Cannadey (2000) into a product called SensifyID. Although the idea was originally applied to network systems, a very similar concept was proposed in Nian et al. (2019) where the algorithm was instead used as a general fault detection tool for process control systems.

By 2010, Xu (2010) extended the original ideas by first representing the system as a partially measurable Markov reward process (MDP with no actions). The states remained $o = [x_{t-n}, x_{t-n+1}, \ldots, x_t]$ and there were no actions in this system. Instead, the authors proposed the agent to learn the probabilities of each state transitioning into an anomalous state given as:

$$P_a(x) = P\{o_{t+1} \in \mathcal{A} | o_t\} \tag{56}$$

where $P_a(x)$ denotes the probability of transitioning into an anomalous state a given observation o_t. $\mathcal{A}$ is a set of anomalous states.

[3] Google DeepMind did not explicitly state the technology used to achieve the savings, only machine learning. However, DeepMind is a company that focuses on reinforcement learning approaches and there were many mentions of creating a general algorithm for all the data centers in the article; therefore, it was assumed reinforcement learning was used. More specifically, meta-RL was most likely used due to the construction of many simulators and the agent's adaptation speed.

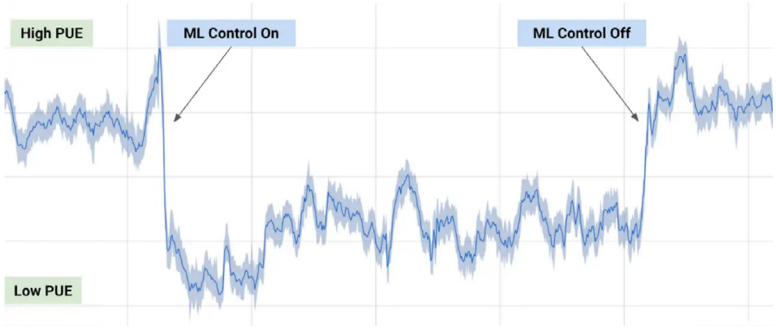


Fig. 23. Power usage effectiveness with and without ML control. Original figure from DeepMind (2016c).

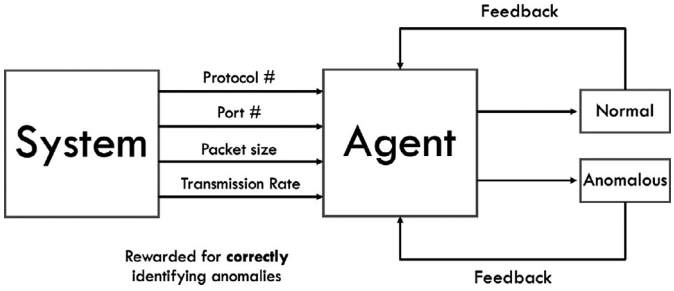


Fig. 24. A sample anomaly detection architecture.

If $P_a(x) > \mu_a$, an anomaly was deemed imminent, where μ_a is a threshold hyper parameter. High values reduce false positives, but may miss anomalies. On the other hand, low values increase true positives, but also increase false alarms. The value function of this approach is represented as:

$$V(x) = \sum_{i}^{n} P(o_{t+1} \in \mathcal{A} | o_t) \cdot r(o_t) \tag{57}$$

where $r(o_t)$ is the reward obtained in o_t. If $o_t \in \mathcal{A}$, $r(o_t) = 1$, otherwise 0. In this setup, states with high values have higher chance of being anomalous. In the end, Xu (2010) also compared the RL anomaly detection algorithm to other classification algorithms such as support vector machines. Results showed that RL anomaly detection resulted in a higher detection accuracy compared to all other methods, although all algorithms scored an accuracy above 99.8% on the selected data sets including linear methods like logistic regression.

More recently in 2018, Huang et al. (2018) proposed a recurrent neural network (RNN) RL anomaly detection algorithm without the need to tune μ_a. Overall, the algorithm was very similar to Cannadey (2000) where a policy, $\pi(u, x)$, was identified to map states to actions $u = [\text{Normal}, \text{Anomalous}]$. In this representation, the classification is binary so no threshold is required. Compared to Cannadey (2000), the system from Huang et al. (2018) was still a POMDP; however, a long short term memory (LSTM) recurrent neural network (RNN) was used to memorize previous states instead of using the state augmentation strategy. The differences in performance between the two strategies have yet to be explored in literature. Ultimately, the algorithm was implemented onto the Yahoo anomaly detection benchmark data set (Laptev et al., 2015) and was able to identify all anomalies with no false alarms.

3.9. Temporal credit assignment

Another advantageous difference exhibited by reinforcement learning is its credit assignment capabilities. There are three forms of credit assignments:

1. *Temporal credit assignment:* Given a sequence of $x_0, u_0, x_1, u_1, \ldots,$ properly assign values based on a desired performance metric to different state-action pairs. For example, the reason a company went bankrupt is rarely caused by the actions of the CEO during the day of the bankruptcy. Most likely, a chain of poor decisions ultimately resulted in this outcome.
2. *Transfer credit assignment:* Ability to generalize one action across many tasks, e.g., driving a car in Canada should be very similar to driving a car in the United States.
3. *Structural credit assignment:* Identifying the effects of individual parameters on the ultimate outcome. For example, increasing the temperature would increase the output flow rate by some amount.

Transfer credit assignment and structural credit assignment are mainly used in transfer learning and supervised learning, respectively. Reinforcement learning conducts *temporal credit assignment* through assigning values to different state-action pairs (Minsky, 1973). RL conducts temporal credit assignment through interactions and understanding the system dynamics. After training (assuming an accurate simulator), the agent's value functions for each state might be useful to aid process operators in determining the current state of the plant. States with high value functions denote good plant operation while states with low value functions may denote sub-optimal operation.

For example in alarm management, RL can assign values to alarms when they go off based on the system's state or state trajectory. In doing so, each alarm corresponds to a respective value and alarms can be sorted based on priority. Additionally, alarms failing to meet a certain value threshold can be filtered out altogether to mitigate alarm flooding from nuisance alarms. Other applications requiring temporal credit assignment, such as root cause analysis, should also be feasible using RL although no studies have been conducted so far on this topic.

4. Shortcomings of reinforcement learning

Although the potential of RL seems promising, there are still several problems hindering it from industry wide adoption. Some shortcomings include: data inefficiency, un-established stability theory, model-free state constraint handling, and the requirement of a representative simulator.

4.1. Data inefficiency

Table 7 shows the time required for popular RL algorithms to learn specific tasks. Perhaps the most controversial topic of reinforcement learning is its poor learning efficiency. For example, OpenAI Five accumulated over 180 years of experience per day, but still required training for many days before becoming competent

in DotA (OpenAI, 2018b). Likewise, AlphaZero was trained for tens of thousands of games to master Chess, Shogi, and Go (DeepMind, 2016b; Silver et al., 2017a; 2018). In process control, plant managers cannot wait for such an lengthy period of time for the initial agent training; therefore, the training of RL agents is infeasible if simulators cannot be used or are too inaccurate. Compared to a human, the learning speed of reinforcement learning seems unreasonably slow. For example, DQN required days of continuous play to become skilled at the game. A task that took most humans a few minutes. One main difference between RL and humans is that RL starts *tabula rasa* while humans are equipped with almost all of the knowledge required before learning a new task. For example, when a human learns to drive, they already understand the functionality of the car's pedals, the street signs, and the ultimate goal. On the other hand, RL starts by knowing nothing – not even the purpose of driving. Intuitively, this scenario is comparable to tasking a newborn baby to drive a car.

One method to inject prior knowledge into the agent is called *transfer learning*. The agent's weights are initiated from some previously trained agent whose task was similar. For example, the knowledge of an agent trained for set-point tracking of a pump could be transferred to an agent doing a similar task on a control valve. The topic of transfer learning is very popular in traditional deep learning, especially in image tasks where training is extremely computationally expensive. A survey on transfer learning in deep learning can be found in Tan et al. (2018). For a survey of transfer learning catered to reinforcement learning tasks, refer to Taylor and Stone (2009).

Another popular way to increase data efficiency is the concept of a replay buffer (also called experience replay) first introduced in Lin (1992). DQN was the first deep RL method to leverage a replay buffer (Mnih et al., 2013; 2015). One of the advantages of RL is its ability to update the agent using tuples of experiences. For example, Q-learning can be updated by the following tuple of experiences:

$$(x_t, u_t, r_{t+1}, x_{t+1})$$

The replay buffer is a large archive (often times 1,000,000 records) of previous experiences and is used to train the agent on decorrelated random experiences from the past. The buffer ensures that the agent's policy does not overfit the current operating regime during time correlated tasks and enhances data efficiency through learning the same experiences many times, a similar concept to running through many epochs in deep learning. During updates, random mini-batches of previous experiences are sampled from the replay buffer to update the agent. Relating to humans, a replay buffer is similar to hippocampal replay, where the memories of experiences are replayed over and over sub-consciously. Indeed, that is one theory of how humans learn so efficiently (Schuck and Niv, 2019). One flaw is that the experiences are sampled randomly. To overcome this, Schaul et al. (2015) proposed the prioritized experience replay algorithm to extend the original concept and biases sampling to experiences exhibiting large TD error. Such experiences can be intuitively understood as *shocking* because the outcome was significantly different than what was expected. For humans, shocking experiences are naturally remembered and replayed more often. The agent's learning speed using prioritized experience replay was compared to the original concept on playing ATARI games. Results show that the agent learned faster in 41 out of 49 games.

Eligibility traces is a third way to increase learning efficiency. The use of eligibility traces is equivalent to combining TD and MC methods into a unifying algorithm. On a high level, eligibility traces allow agents to update multiple value functions per step, like in MC methods, without the termination of an episode. A detailed overview regarding eligibility traces can be found in Sutton and Barto (2018).

It is also possible to speed up training through exploiting the heuristics of the environment. An heuristically accelerated RL (HARL) algorithm uses an external heuristics function $\mathcal{H} : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$ to guide the agent's action selection (Reinaldo et al., 2008). Intuitively, during action selection, HARL agents observe the following values instead:

$$Q(x_t, u) = [Q(x_t, u_1) + H(x_t, u_1), Q(x_t, u_2) + H(x_t, u_2), \ldots] \quad (58)$$

where H is the heuristics function that adds some value to each action-value to promote or discourage its corresponding action. Heuristics functions are very flexible. For example, an upper confidence bound heuristics function promotes exploration of potentially optimal states, instead of random exploration such as in ϵ-greedy, and is given by (Garivier and Moulines, 2008):

$$U_t = arg\max_u \left[Q_t(x, u) + c\sqrt{\frac{ln\, t}{N_t(x, u)}} \right] \quad (59)$$

where c is the degree of exploration and N_t is the number of times that u was selected prior to time t. As $N_t(x, u) \rightarrow \infty$, the corresponding $Q(x, u)$ value becomes very accurate and $\mathcal{H} \rightarrow 0$. Heuristically accelerated Q-learning was first introduced in Reinaldo et al. (2004), showing significantly better results when applied onto mobile robots. For the design of a heuristic function compatible with eligibility traces, see Reinaldo et al. (2012). Ferreira et al. (2014) introduced a heuristics function to accelerate training in multi-agent multi-objective environments. In Martins and Bianchi (2014), the performance of popular HARL algorithms is compared with their non-heuristic counterpart. It was found that the heuristics variant significantly improved the performance of the original algorithm.

Lastly, a newer topic called meta RL was introduced recently to improve the calibration time of the agent. Specifically, meta RL targets industrial applications where accurate simulators are difficult to identify. In such situations, the agent requires a long online calibration time even after being pre-trained in simulation because the simulator-process mismatch is large. By applying meta-RL, this calibration time can be significantly reduced. Meta-learning was first introduced in Hochreiter et al. (2001), but the ideas were first applied to RL in Wang et al. (2016); Duan et al. (2017). In meta RL, the agent learns a general policy through interacting with many different simulation models (different models capture model uncertainty). During testing, the general policy should have the ability to adapt to new similar tasks quickly. The framework of meta-RL is nearly identical to normal RL; the state and action spaces of the agent do not change across the different simulators. However, the identified policy in meta-RL is given by:

$$\pi(u_{t-1}, r_{t-1}, x_t) \rightarrow u_t \quad (60)$$

That is, the agent uses the previous action and reward in conjunction with the current states to obtain the current action. Intuitively, u_{t-1} and r_{t-1} provide the agent with intuition on what the objective of the current task may be, creating a near-Markovian setting. Additionally, RNNs are used to provide the agent with a memory of past states; hence, not needing it as an explicit input. For more information regarding meta-RL, see Wang et al. (2016) and Duan et al. (2017).

Normally, RL algorithms are initiated *tabula rasa*, and are very data inefficient. With the addition of a replay buffer, eligibility traces, heuristics, and more recently meta-RL, applications of RL onto industrial process control systems are now more feasible for small scale applications.

4.2. Scalability

Historically, the exact solution to MDPs required dynamic programming and was cursed by dimensionality (i.e., the computational complexity increased exponentially as the states and/or actions increased) (Bellman, 1957a). However, rapid advancements in nonlinear function approximation for the value function, as demonstrated in Mnih et al. (2013), Mnih et al. (2015), Silver et al. (2014), Lillicrap et al. (2015), Schulman et al. (2015), and Schulman et al. (2017), has largely solved the scalability issue for systems with a reasonable number of states (i.e., systems with less than 100 states). For larger systems, multi-agent RL architectures can be used, as demonstrated in OpenAI (2018b), to achieve optimality in an overall system. One flaw with function approximation approaches, especially using deep learning, is their lack of explainability. If an incident were to occur, identifying the root cause is nearly impossible due to the black-box nature of the policy function. Conversely, explainable algorithms, such as tabular *Q*-learning, becomes infinitely large as the states and actions increase and are infeasible for most problems. For deep RL methods to truly be explainable, more fundamental problems must be solved in the literature of deep learning; therefore in the near future, industrial applications of RL may be limited to small scale or safety-insensitive systems if explainability is mandatory.

4.3. Stability

Stability guarantees are also typically required for optimal control projects to be commissioned. The literature of RL in the process control space is quite embryonic with most papers explicitly studying stability stemming from 2017 onwards. Berkenkamp et al. (2017) was the first influential paper regarding RL stability where the authors developed a model-based RL algorithm with stability guarantees. More specifically, the proposed algorithm was proved to be stable given a Lipschitz continuous settings with reliable system models. Additionally, it was assumed that the system is initiated in the region of attraction: a forward invariant subspace where all state trajectories stay within it and eventually converges to a goal state. The region of attraction is asymptotically stable and is initially found using the system model. As the agent explores, the zone of attraction will increase in size. However in control theory, it has been shown that the region of attraction is very difficult to find for large nonlinear systems (Zecevic and Siljak, 2009). In Jin and Lavaei (2018), a stability-certified RL method was introduced. The input-output gradients of the policy during updates were regulated to obtain strong guarantees of robust stability. Furthermore, the authors applied the proposed algorithm to a multi-flight formation and a power system frequency regulation task to demonstrate its effectiveness in terms of performance, learning speed, and stability. For a more theoretical overview, Busoniu et al. (2018) provides a great overview of RL's stability theory compared to traditional optimal control methods such as H_∞, linear quadratic regulator (LQR), or linear quadratic Gaussian (LQG).

4.4. Convergence

Convergence of tabular reinforcement learning algorithms for MDPs can be guaranteed given learning rate $0 \leq \alpha < 1$, bounded reward function $|R_n| \leq R^{\max}$ and satisfying (Sutton and Barto, 2018):

$$\sum_{i=1}^{\infty} \alpha_i(x,u) = \infty, \forall x,u \tag{61}$$

$$\sum_{i=1}^{\infty} \alpha_i^2(x,u) < \infty, \forall x,u \tag{62}$$

Eq. (61) is a condition to ensure that short term noise can be overcome. Subsequently, Eq. (62) imposes a condition to ensure α eventually becomes sufficiently small for convergence to occur. Such conditions are strict but are required given the stochastic conditions (Marti, 2008). For POMDPs, convergence guarantees are difficult because the agent does not know its current true states.

Linear function approximation methods were also proven to converge and can be found in Sutton and Barto (1998). No proofs currently exist for the convergence of nonlinear function approximation cases where value functions are updated using direct methods. Direct methods refer to algorithms that perform gradient descent updates assuming model weights affect only $Q(x_t, u_t)$, and not $Q(x_{t+1}, u_{t+1})$. The lack of proofs stem from the flawed assumption. In function approximation cases, parameter updates would most definitely affect both $Q(x_t, u_t)$ and $Q(x_{t+1}, u_{t+1})$ because they are often calculated using the same weights. Tabular methods do not suffer from such an assumption because each action-value was explicitly stored in the *Q*-table, and each update is independent. On the other hand, convergence of RL algorithms using residual gradient descent, where both $Q(x_t, u_t)$ and $Q(x_{t+1}, u_{t+1})$ are considered during weight updates, has been shown to converge, even in the POMDP case (Baird, 2013; 1995).

4.5. Constraints

For nearly all industrial control applications, both state and input constraints are required to address safety concerns (Arendt and Lorenzo, 2000). In RL, input constraints are trivial. Conversely, state constraints (i.e., guaranteeing the avoidance of certain states) can be very challenging. One could implement soft state constraints and design a reward function to discourage policies resulting in arrival of such states. Indeed, humans are trained in such a way where guardians discourage undesirable behaviour; however, such a weak condition does not satisfy the strict safety requirements of industrial process control.

The earliest study of constrained RL was conducted in Altman (1999) where the author introduced the constrained Markov decision process (CMDP). In CMDPs, there exists an additional value function called the constrained value function, *C*, and is identified concurrently with V_π. During action selection, the selected action must satisfy:

$$max\, V_\pi$$

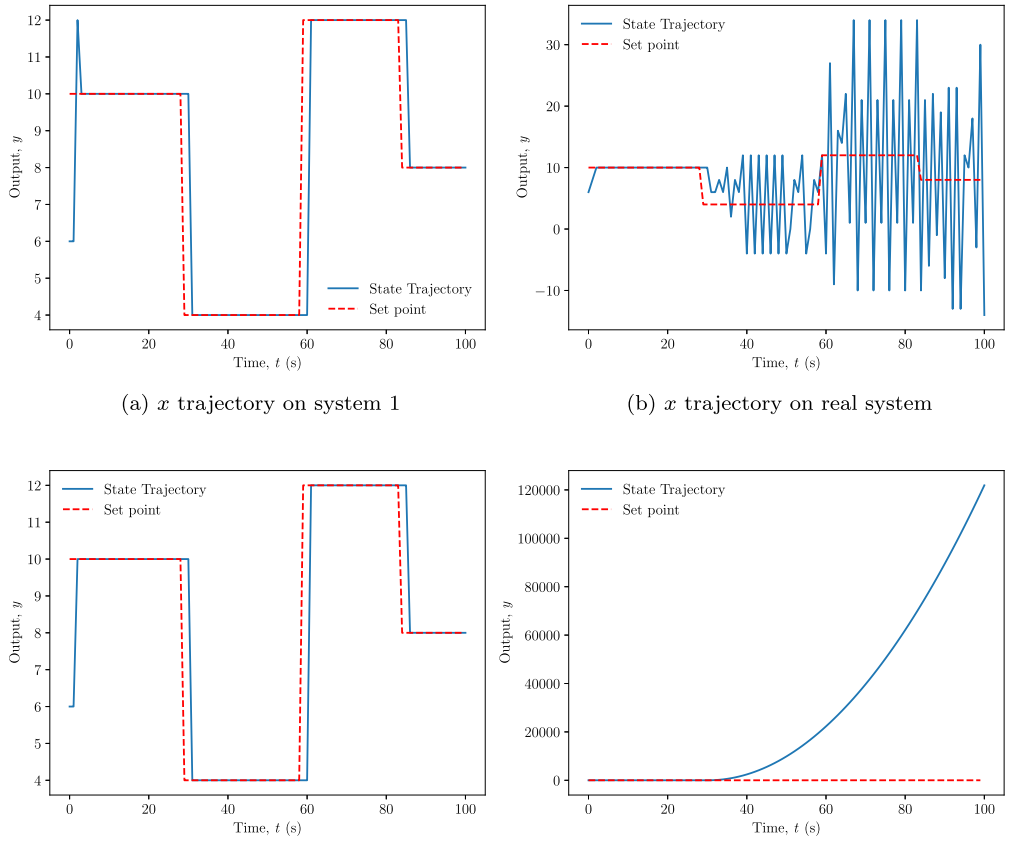
$$s.t\, C_\pi \geq c$$

where c is a real value threshold. Typically, C is represented as a probability of exceeding some constraints and is given as:

$$Pr(C \leq c) \leq pr_0$$

where pr_0 is the maximum allowable probability of violating the given constraints (Geibel, 2006). CMDP systems are solved using linear programming due to multiple reward functions. Furthermore, CMDP RL is usually model-based methods and many solutions are intractable for high-dimensional problems. Moreover, nearly all recent progress of constrained RL uses concepts from constrained optimization to perform a policy search (Achiam et al., 2017; Andersson et al., 2015; Bhatia et al., 2018). For example, Achiam et al. (2017) introduced solution to continuous CMDPs using a constrained policy optimization method. In Bhatia et al. (2018), authors developed three different optimization approaches on top of DDPG to handle resource allocation constraints.

More recently, system constraints are handled through a new field of study in RL named *safe reinforcement learning* where authors attempt to design agents to solve tasks without violating safety characteristics of the system. Typically, there are two methods: i) The first way modifies the optimality criterion of the agent



(a) x trajectory on system 1

(b) x trajectory on real system

(c) x trajectory on system 2

(d) x trajectory on real system

Fig. 25. State trajectories of the two RL agents applied to *system 1* and *system 2* and the *real system*.

with a safety factor; ii) the second way is to provide the agent with external heuristics of the system or to guide the agent using a risk metric. For example, Berkenkamp et al. (2017) trained an agent to solve the inverted pendulum problem without the pendulum ever falling down. A survey of safe RL methods can be found in Garcia (2015).

Table 8
Two data sets collected from an arbitrary process.

$\mathbf{y}^{(1)}$	$\mathbf{u}_1^{(1)}$	$\mathbf{u}_2^{(1)}$	$\mathbf{y}^{(2)}$	$\mathbf{u}_1^{(2)}$	$\mathbf{u}_2^{(2)}$
10	1	3	6	2	−4
18	3	3	8	3	−7
12	2	2	10	4	−12

4.6. Accurate simulator

Overall, it seems that the most impressive applications of RL (i.e., agents that were easily and decisively able to triumph human level performance) were all applied onto video games with very limited applications elsewhere. The main reason is twofold: games have explicit rules that are well understood and the lack of accurate simulators for industrial settings.

Firstly, even the most complex games are known completely by the designers of the games; therefore, it is much simpler to design reward functions because the ultimate goal is known. Indeed, an agent pursing the true ultimate goal without unnecessary bias is demonstrated to be vastly superior as shown by AlphaGo Zero and AlphaZero (Silver et al., 2016; 2017b). However, when consider even the simplest SISO control tasks, it is not explicitly known whether a controller with oscillations but superior tracking performance is better compared to more conservative counterparts. As such, designing a good reward function is very challengin because the true objective is often unclear. Notice that even in Google DeepMind's case, the ultimate objective was not just to minimize electricity, but rather to minimize PUE. The implications of this were not provided, but one can assume minimizing electricity costs may lead to unintended consequences.

Secondly, most major triumphs of reinforcement learning were applied to systems with a perfect simulator. That is, the performance achieved by following specific policies in the training phase can be exactly achieved during online application. Additionally, the mapping of arbitrary simulated states to actions is exactly representative of what would occur in the real world. Without a doubt, such a condition is somewhat achievable in the real world for some tasks. For example, astronauts and pilots are first trained in simulation before deployment; however, the production of such a simulator in industrial process control is time consuming, costly, and often times, impossible. In fact, this might only be economically feasible for small scale systems or if the agent was given the opportunity to directly manipulate the industrial distributed control system. Moreover, if such models were identifiable and can be represented using mathematics, model based optimal control methods (known as planning methods in RL literature) such as MPC may be the superior choice assuming feasible computation times. In previous sections, it has been shown that RL can be trained on sub-optimal models and eventually adapt its policy to the environment (a significant advantage over traditional model-based approaches); however, this topic has yet to be thoroughly explored outside of simulations.

Table 9
Hyper parameters for the agents controlling an arbitrary system.

Hyper Parameter	Value
States, x	$\varepsilon = [-15, -14, \ldots, 15]_{1\times 31}$
Action 1, u_1	$\Delta u_1 = [-5, -4, \ldots, 5]_{1\times 11}$
Action 2, u_2	$\Delta u_2 = [-5, -4, \ldots, 5]_{1\times 11}$
Reward, r	Eq. (66)
Learning rate, α	[0.001, 0.7]
Discount factor, γ	0.95
Exploratory factor, ϵ	[0.1, 1]
Evaluation time	1 seconds
System representation	FOMDP

Most importantly, the identified system model must be representative of the real process during process identification. If not, the optimal policy identified during training will perform poorly and could potentially become a safety concern on the real process. Consider the following quantitative example:

Suppose there exists a simple arbitrary system given by:

$$y = u_1^2 + u_2 + 6 \tag{63}$$

Table 8 shows two data sets collected for the system described by Eq. (63). The first and second data sets were then used to identify Eqs. (64) and (65), respectively. In both cases, the MSE of the model was zero given their respective data sets. Despite zero modelling errors given their respective data sets, it can be seen that both models do not represent the real system, shown in Eq. (63), in the slightest. In fact, Eq. (65) does not even consider u_2 in the system model. To avoid confusion, the system described by Eqs. (64) and (65) will be referred to as *system 1* and *system 2*, respectively. Furthermore, the actual system shown in Eq. (63) will be denoted as the *real system*. Moreover, the RL agent trained on systems 1 and 2 are denoted as *agent 1* and *agent 2*, respectively.

$$y = 4u_1 + 2u_2 \tag{64}$$

$$y = 2u_1 + 2 \tag{65}$$

Suppose that this arbitrary system requires a controller for set-point tracking, with the objective described by the following reward function:

$$r(x, u) = -(y - y_{sp}) - \Delta u_1 - \Delta u_2 \tag{66}$$

where Δu_1 and Δu_2 denotes the change in controller input between $t-1$ and t. For this task, two different RL agents were trained on the system models provided by Eqs. (64) and (65). Both agents were trained for 50,000 update steps and shared the same hyper parameters shown in Table 9.

Fig. 25 a and c show the state trajectories of the agents trained using Eqs. (64) and (65) applied to their respective training systems. Fig. 25b and d show the trajectories of the actual system when the two agents are implemented. The cumulative reward of the different agents is shown in Table 10. It can be seen that both agents performed poorly on the real system due to the non-representative training model.

Ultimately, the poor performance by the RL agents were caused by the non-representativeness of the data and/or poor selection of the model structures. The optimal policy identified by agents trained on either model is not reliable because the models do not resemble the real system; therefore, it is critical to ensure the training model is, at least somewhat, representative of the real system.

Table 10
Cumulative reward of the agents.

Agent 1 on System 1	−150
Agent 2 on System 2	−156
Agent 1 on Real System	−6255
Agent 2 on Real System	−10698

Table 11
Most influential advantages and disadvantages of reinforcement learning.

Advantages	Disadvantages
Online computation time	Accurate simulator required
Can learn many tasks	Reward design can be difficult
Direct adaptive optimal control	Stability theory lacking
Engineered features not needed	State constraints are difficult

5. Conclusions

Reinforcement learning has demonstrated to have great potential in surpassing human level performance in many complex tasks assuming an sufficiently accurate simulator exists or can be constructed. It was shown to possess the ability to learn many different tasks using the same algorithm. This may imply great potential in engineering where custom design of similar projects result in significant cost and time expenditure. Moreover, reinforcement learning also exhibits the ability to self-learn, significantly reducing development time.

With a proper problem formulation, RL has also been successfully simulated in process control problems with regulation or set point tracking objectives. Optimal control problems have also been shown to be feasible for RL and ADP methods in literature. A gentle first step towards industrial scale RL implementation could be to use RL for PID tuning. Indeed, many methods have been proposed to re-configure PID parameters as a function of proportion, integral, and derivative errors, achieving superior control performance compared to other tuning methods. For more ambitious projects, a closely monitored RL agent might be feasible for continuous adaptive optimal control. Google was one of the first movers to do so, resulting in up to 40% electricity savings in their industrial data centers. Another potential advantage of RL is its direct adaptive characteristic. Compared to traditional adaptive optimal control frameworks where model re-identification is required, RL is model-free and adapts the policy directly. Lastly, RL can solve the temporal credit assignment problem where values are assigned to each state to denote its desirability. This information could have potential implications in alarm management, root cause analysis, and other related applications. The most influential advantages and disadvantages of reinforcement learning are summarized in Table 11.

As a closing note, the truly greatest characteristic of RL is its general nature, allowing for learning of nearly anything through a general algorithm. Although modern RL still faces many shortcomings, it is expected that RL will play an important role in industrial process control in the near future.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported in part by the NSERC Industrial Research Chair in Control of Oil Sands Processes and NSERC discovery grants.

References

Achiam, J., Held, D., Tamar, A., Abbeel, P., 2017. Constrained policy optimization. ML arXiv:1705.10528.

Akbarimajd, A., 2015. Reinforcement learning adaptive PID controller for an under--actuated robot arm. Int. J. Integrat. Eng. 7, 20–27.

Altman, E. (1999). Constrained markov decision processes. ISSN 01676377. 10.1016/0167-6377(96)00003-X

Andersson, O., Heintz, F., Doherty, P., 2015. Model-based reinforcement learning in continuous environments using real-time constrained optimization. AIII.

Arendt, J.S., Lorenzo, D.K., 2000. Evaluating process safety in the chemical industry: a user's guide to quantitative risk analysis. American institute of chemical engineers: New york, new york, USA. ISBN 978-0-81-690746-5

Asis, D.K., Chan, A., Pitis, S., Sutton, R.S., Graves, D., 2020. Fixed-horizon temporal difference methods for stable reinforcement learning. AAAI arXiv:1909.03906.

AspenTech (2019). AspenONE advanced process control. [Online]. Available: https://www.aspentech.com/uploadedfiles/v7/1732_15_aspen_apc_web.pdf.

Baird, L.C., 1995. Residual Algorithms. In: Proceedings of the Workshop on Value Function Approximation, Machine Learning.

Baird, L.C., 2013. Reinforcement learning with function approximation. Machine Learning. San Francisco, CA

Barto, A., Sutton, R.S., Brouwer, P.S., 1981. Associative search network: a reinforcement learning associative memory. Biol. Cybern. 40, 201–211.

Bellman, R.E., 1957a. Dynamic Programming. Princeton University Press, New Jersey, USA.

Bellman, R.E., 1957b. A markovian decision process. J. Math. Mech. 6, 679–684.

Bemporad, A., Morari, M., Dua, V., Pistikopoulos, E.N., 2002. The explicit linear quadratic regulator for constrained systems. Automatica 38, 3–20. 10.101610.1016/s0005-1098(01)00174-1

Berkenkamp, F., Turchetta, M., Schoellig, A.P., Krause, A., 2017. Safe model-based reinforcement learning with stability guarantees. NIPS 1–11.

Bhatia, A., Varakantham, P., Kumar, A., 2018. Resource constrained deep reinforcement learning. ML arXiv:1812.00600.

Borel, 1909. Les probabilities denombrables et leurs applications aritbmttieuqs. Rendiconti del Circolo Matematico di Palermo 27, 247–271.

Borgnakke, C., Sonntag, R.E., 2008. Fundamentals of Thermodynamics. Wiley, Hoboken, New Jersey, United States. ISBN 978-0-470-04192-5

Bottou, L., 2010. Large-scale machine learning with stochastic gradient descent. COMPSTAT 177–186. doi:10.1007/978-3-7908-2604-3-16.

Bradtke, S.J., Duff, M.O., 1994. Reinforcement learning methods for continuous-time markov decision problems. NIPS 393–400.

Brujeni, L.A., Lee, J.M., Shah, S.L., 2010. Dynamic tuning of PI-controllers based on model-free reinforcement learning methods. ICCAS 453–458. Gyeonggi-do

Busoniu, L., Bruin, T.D., Tolic, D., Kober, J., Palunko, I., 2018. Reinforcement learning for control: performance, stability, and deep approximators. Annu. Rev. Control (46) 8–28.

Cannadey, J., 2000. Next generation intrusion detection: autonomous reinforcement learning of network attacks. NISSC 1–12.

Chabris, C., 2015. The real kings of chess are computers. Wall Street J..

Chenna, S.K., Jain, Y.K., Kapoor, H., Bapi, R.S., Yadaiah, N., Negi, A., Rao, V.S., Deekshatulu, B.L., 2004. State estimation and tracking problems: a comparison between kalman filter and recurrent neural networks. ICONIP 275–281.

DeepMind, G. (2016a). Alphago. [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far.

DeepMind, G. (2016b). Alphazero: shedding new light on chess, shogi, and go. [Online]. Available: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go.

DeepMind, G. (2016c). Deepmind AI reduces google data centre cooling bill by 40%. [Online]. Available: https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40.

DeepMind, G. (2018). Safety-first AI for autonomous data centre cooling and industrial control. [Online]. Available: https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control.

DeepMind, G. (2019). Alphastar: mastering the real-time strategy game starcraft II. [Online]. Available: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.

Deisenroth, M.P., Neumann, G.P., 2013. A survey on policy search for robotics. Found. Trend. Robot. 1–142.

Doerr, A., Nguyen-Tuong, D., Marco, A., Schaal, S., Trimpe, S., 2017. Model-based policy search for automatic tuning of multivariate PID controllers. ICRA arXiv:1703.02899.

Duan, Y., Schulman, J., Chen, X., Bartlett, P., Sutskever, I., Abbeel, P., 2017. RL^2: fast reinforcement learning via slow reinforcement learning. ICRL arXiv:1611.02779.

Dunn, J.C., Bertsekas, D.P., 1989. Efficient dynamic programming implementations of newtons method for unconstrained optimal control problems. J. Optim. Theory Appl. 63 (1), 23–38.

Ellis, M., Durand, H., Christofides, P.D., 2014. A tutorial review of economic model predictive control methods. J. Process Control 24, 1156–1178. doi:10.1016/j.compchemeng.2010.07.001.

Fan, H., Yao, C., Guo, J., Lu, X., 2019. Deep reinforcement learning for energy efficiency optimization in wireless networks. ICCCBDA.

Ferreira, L., Riberiro, C., Bianchi, R., 2014. Heuristically accelerated reinforcement learning modularization for multi-agent multi-objective problems. Appl. Intell. 41, 551–562.

Fujimoto, S., Hoof, H., Meger, D., 2018. Addressing function approximation error in actor-critic methods. ICML arXiv:1802.09477.

Garcia, J., 2015. A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. 16, 1437–1480.

Garivier, A., & Moulines, E. (2008). On upper-confidence bound policies for non-stationary bandit problems. arXiv:0805.3415.

Ge, Z., Song, Z., Ding, S.X., Huang, B., 2017. Data mining and analytics in the process industry: the role of machine learning. IEEE 20590–20616.

Geibel, P., 2006. Reinforcement learning for MDPs with constraints. ECML 646–653.

Goodfellow, I., Bengio, Y., Courville, A., 2015. Deep Learning. The MIT Press, Cambridge, Massachusetts, USA.

Gorges, D., 2017. Relations between model predictive control and reinforcement learning. IFAC 50, 4920–4928.

Gurel, C.S., 2017. Technical Report: Q-Learning for Adaptive PID Control of a line Follower Mobile Robot. University of Maryland 1–19.

Haarnoja, T., Zhou, A., Abbeel, P., Levine, S., 2018. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. ICML arXiv:1801.01290.

Hakim, A., Hindersah, H., Rijanto, E., 2013. Application of reinforcement learning on self-tuning PID controller for soccer robot multi-agent system. rICT & ICeV-T.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D., 2018. Deep reinforcement learning that matters. AAAI arXiv:1709.06560.

Hinton, J., Sejnowski, T., 1999. Unsupervised Learning: Foundations of Neural Computation. The MIT Press: Cambridge, Massachusetts, USA. ISBN 978-0-26-258168-4

Hochreiter, A., Younger, S., Conwell, P., 2001. Learning to learn using gradient descent. ICANN.

Hoskins, J.C., Himmelblau, D.M., 1992. Process control via neural networks and reinforcement learning. Comput. Chem. Eng. 16, 241–251.

Howell, M.N., Best, M.C., 2000. On-line PID tuning for engine idle-speed control using continuous action reinforcement learning automata. Control Eng. Pract. 8, 147–154.

Huang, C., Wu, Y., Zuo, Y., Pei, K., Min, K., 2018. Towards experienced anomaly detector through reinforcement learning. AAAI 10, 8087–8088.

Huesman, A.E.M., Bosgra, O.H., Van Hof, P.M.J., 2008. Integrating MPC and RTO in the process industry by economic dynamic lexicographic optimization; an open-loop exploration. AIChE.

Jin, M., Lavaei, J., 2018. Stability-certified reinforcement learning: acontrol-theoretic prespective. ML arXiv:1810.11505.

Joy, M., Kaisare, N., 2011. Approximate dynamic programming-based control of distributed parameter systems. J. Chem. Eng. (6) 452–459.

Kaelbling, L., Littman, M., & Cassandra, A. (1999). DPOMDPs and their algorithms, sans formula![Online]. Available: http://cs.brown.edu/research/ai/pomdp/tutorial/index.html.

Karatzas, I., Shreve, S.E., 1991. Brownian Motion and Stochastic Calculus, 2nd Springer-Verlag, Berlin, Germany. ISBN 978-0-387-97655-6

Laptev, N., Amizadeh, S., Flint, I., 2015. Generic and scalable framework for automated time-series anomaly detection. ACM SIGKDD 1939–1947.

Lee, H.L., Shin, J., Realff, M.J., 2018. Machine learning: overview of the recent progresses and implications for the process systems engineering field. Comput. Chem. Eng. 114 (9), 111–121.

Lee, J.H., Wong, W., 2010. Approximate dynamic programming approach for process control. J. Process Control 20 (9), 1038–1048.

Lee, J.M., Lee, J.H., 2005. Approximate dynamic programming-based approaches for input-output data-driven control of nonlinear processes. Automatica 41 (7), 1281–1288.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv:1509.02971.

Lin, L., 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. Mach. Learn. 8, 293–321.

Lusena, C., Mundhenk, M., Goldsmith, J., 2001. Nonapproximability results for partially observable markov decision processes. J. AI Res. 14, 83–103.

Maravelias, C.T., Sung, C., 2009. Integration of production planning and scheduling: overview, challenges and opportunities. CCE 33, 1919–1930.

Marti, K., 2008. Stochastic Optimization Methods, 2nd Springer, New York, USA. ISBN 978-3-540-79458-5

Martins, M., Bianchi, R., 2014. Heuristically-accelerated reinforcement learning: acomparative analysis of performance. TAROS 15–27.

Mayne, D., Rawlings, J.B., 2017. Model Predictive Control: Theory and Design, 2nd Nob Hill Publishing, Wisconson, USA.

Meijering, E., 2002. A chronology of interpolation: from ancient astronomy to modern signal and image processing. IEEE 90, 319–342.

Mes, M.R.K., Rivera, A.P., 2017. Approximate Dynamic Programming by Practical Examples. Springer, New York, USA. ISBN 978-3-319-47764-0

Minsky, M., 1973. Steps towards artificial intelligence. IRE 8–31.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M., 2013. Playing atari with deep reinforcement learning. NIPS.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D., 2015. Human-level control through deep reinforcement learning. Nature 518, 529–533.

Monfort, N., Bogost, I., 2018. Racing the Beam: The Atari Video Computer System. The MIT Press, Cambridge, Massachusetts, USA. ISBN 0-262-01257-X

Moriyama, T., Magistris, G.D., Tatsubori, M., Pham, T., Munawar, A., Tachibana, R., 2018. Reinforcement learning testbed for power-consumption optimization. AsiaSim.

Ng, A., 2003. Shaping and Policy Search in Reinforcement Learning. University of California, Berkeley Phd dissertation.
Ng, A. (2018). Machine Learning Yearing. First Edition
Nian, R., Liu, J., Huang, B., Tawanda, M., 2019. Fault tolerant control system: a reinforcement learning approach. SICE 1010–1015.
OpenAI (2018a). How to train your openAI five. [Online]. Available: https://openai.com/blog/how-to-train-your-openai-five/.
OpenAI (2018b). OpenAI five. [Online]. Available: https://openai.com/blog/openai-five/.
Pannocchia, G., Gabiccini, M., Artoni, A., 2015. Offset-free MPC explained: novelties, subtleties, and applications. IFAC 48, 342–351.
Poznyak, A.S., 2008. Advanced Mathematical Tools for Automatic Control Engineers: Deterministic Techniques, Volume 1. Elsevier, Oxford, United Kingdom. ISBN 978-0-08-044674-5
PwC (2019). Sizing the prize: What's the real value of AI for your business and how can you capitalise?[Online]. Available: www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf.
Raju, L., Milton, R.S., Suresh, S., Sankar, S., 2015. Reinforcement learning in adaptive control of power system generation. Procedia Comput. Sci. 46, 202–209.
Rawlik, K., Toussaint, M., Vijayakumar, S., 2013. On stochastic optimal control and reinforcement learning by approximate inference. IJCAI 3052–3060.
Reinaldo, B., Riberiro, C., Costa, A., 2004. Heuristically accelerated q-learning: a new approach to speed up reinforcement learning. SBIA 245–254.
Reinaldo, B., Riberiro, C., Costa, A., 2008. Accelerating autonomous learning by using heuristic selection of actions. J. Heurist. 14, 135–168.
Reinaldo, B., Riberiro, C., Costa, A., 2012. Heuristically accelerated reinforcement learning: theoretical and experimental results. ECAI 169–175.
Reis, A. (2017). Reinforcement learning: Eligibility traces and TD(λ). [Online]. Available: https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html.
Richter, S., Jones, C.N., Morari, M., 2012. Computational complexity certification for real-time MPC with input constraints based on the fast gradient method. Automat. Control 57, 1391–1403.
Russel, S.J., Norvig, P., 2009. Artificial Intelligence: A Modern Approach, 3rd. Prentice Hall, Upper Saddle River, New Jersey, USA, pp. 1–22. ISBN 978-0-13-604259-4
Schaul, T., Quan, J., Antonoglou, I., Silver, D., 2015. Prioritized experience replay. ML arXiv:1511.05952.
Schuck, N.W., Niv, Y., 2019. Sequential replay of nonspatial task states in the human hippocampus. Science 364, 1–11. doi:10.1126/science.aaw5181.
Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P., 2015. Trust region policy optimization. ICML arXiv:1502.05477.
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. Mach. Learn. arXiv:1707.06347.
Seborg, D.E., Mellichamp, D.A., Edgar, T.F., Doyle, F.J., 2013. Process Dynamics and Control, 2nd Wiley, Hoboken, New Jersey, USA. ISBN 978-0-470-12867-1
Sedighizadeh, M., Rezazadeh, A., Sun, W., 2008. Adaptive PID controller based on reinforcement learning for wind turbine control. Int. J. Electric. Inf. Eng. 2, 124–130.
Shin, J., Badgwell, T.A., Liu, K., Lee, J.H., 2019. Reinforcement learning - overview of recent progress and implications for process control. CCE 127, 282–294. doi:10.1016/j.compchemeng.2019.05.029.
Sidhu, H.S., Siddhamshetty, P., Kwon, J.S., 2018. Approximate dynamic programming based control of proppant concentration in hydraulic fracturing. Mathematics 6 (8), 132.
Silver, D. (2018). Class lecture, topic: "planning by dynamic programming." COMPGI13. Imperial College London, London, Mar. 12.
Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529, 484–503.
Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ML arXiv:1712.01815.
Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science 362. doi:10.1126/science.aar6404.
Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M., 2014. Deterministic policy gradient algorithms. ICML.
Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G., Graepel, T., Hassabis, D., 2017b. Mastering the game of go without human knowledge. Nature 550, 354–359.
Spielberg, S.P.K., Gopaluni, R.B., Loewen, P.D., 2017. Deep reinforcement learning approaches for process control. AdCONIP (10) 201–207. doi:10.1109/ADCONIP.2017.7983780.
Sutton, R., Barto, A., 1998. Reinforcement Learning: An Introduction, 1st. The MIT Press, Cambridge, Massachusetts, USA, pp. 206–207.
Sutton, R., Barto, A., 2018. Reinforcement Learning: An Introduction, 2nd The MIT Press, Cambridge, Massachusetts, USA.
Sutton, R.S., 1988. Learning to predict by the methods of temporal differences. Mach. Learn. 3, 9–44.
Sutton, R.S., Barto, A.G., Williams, R.J., 1991. Reinforcement learning is direct adaptive optimal control. ACC.
Sutton, R.S., McAllester, D., Singh, S., Mansour, Y., 1999. Policy gradient methods for reinforcement learning with function approximation. NIPS 1057–1063.
Tan, C.T., Sun, F., Kong, T., Zhang, W., Yang, C., Liu, C., 2018. A survey on deep transfer learning. ICANN.
Taylor, M.E., Stone, P., 2009. Transfer learning for reinforcement learning domains: a survey. ML 10, 1633–1685.
Technologies, T. (2019). Pumps & process automation lab. [Online]. Available: http://www.turbinetechnologies.com/educational-lab-products/pump-lab-with-automation.
Thorndike, E. L.. Animal intelligence. Hafner, Darien, CT.
Trends, G. (2017). The growth in search of reinforcement learning. [Online]. Available: https://trends.google.com/trends/explore?date=2007-01-01%202019-08-21&q=reinforcement%20learning.
Uhlenbeck, G., Ornstein, L.S., 1930. On the theory of the brownian motion. Phys. Rev. 36.
Vinyals, O., Babuschkin, I., Czarnecki, W., Mathieu, M., Dudzik, A., Chung, J., Choi, D., Powell, R., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., Silver, D., 2019. Grandmaster level in starcraft II using multi-agent reinforcement learning. Nature (362) doi:10.1038/s41586-019-1724-z.
Wang, J., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J., Munos, R., Blundell, C., Kumaran, D., Botvinick, M., 2016. Learn. Reinforcement Learn arXiv:1611.05763.
Wang, X., Cheng, Y., Sun, W., 2006. A proposal of adaptive PID controller based on reinforcement learning. China Univ Min. Technol. 17, 40–44.
Wang, Y., Boyd, S., 2008. Fast model predictive control using online optimization. In: Proc. 17th world congress.
Wang, Y., Velswamy, K., Huang, B., 2017. A long-short term memory recurrent neural network based reinforcement learning controller for office heating ventilation and air conditioning systems. Processes 5, 1–18. doi:10.3390/pr5030046.
Wang, Y., Velswamy, K., Huang, B., 2018. A novel approach to feedback control with deep reinforcement learning. IFAC 31–37.
Wright, S.J., 1997. Applying new optimization algorithms to model predictive control. Chem. Process Control 91 (316), 147–155.
Xu, X., 2010. Sequential anomaly detection based on temporal-difference learning: principles, models and case studies. Appl. Soft Comput. 10, 859–867.
Zecevic, A.I., Siljak, D.D., 2009. Regions of attraction. Control Complex Syst. 111–141.