Welcome to the Jungle: A Conceptual Comparison of Reinforcement Learning Algorithms

Kenneth Schröder, Alexander Kastius and Rainer Schlosser
Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

Keywords: Reinforcement Learning, Markov Decision Problem, Conceptual Comparison, Recommendations.

Abstract: Reinforcement Learning (RL) has continuously risen in popularity in recent years. Consequently, multiple RL algorithms and extensions have been developed for various use cases. This makes RL applicable to a wide range of problems today. When searching for suitable RL algorithms to specific problems, the options are overwhelming. Identifying the advantages and disadvantages of methods is difficult, as sources use conflicting terminology, imply improvements to alternative algorithms without mathematical or empirical proof, or provide incomplete information. As a result, there is the chance for engineers and researchers to miss alternatives or perfect-fit algorithms for their specific problems. In this paper, we identify and explain essential RL properties. Our discussion of different RL concepts allows to select, optimize, and compare RL algorithms and their extensions, as well as reason about their performance.

1 INTRODUCTION

Many recent RL algorithms to solve Markov decision problems (MDP) include at least one unique feature designed to improve performance under specific challenges. These challenges usually originate from different properties of the decision process or the statistical drawbacks of related optimization algorithms. The corresponding publications often include performance comparisons to the related algorithms but sometimes lack clear distinctions of the advantages and disadvantages of solutions within this field of research. In (Schulman et al., 2017) the authors, for example, motivate in their introduction that robustness of RL algorithms with respect to hyperparameters is desirable. The evaluation section of their publication does not include any comparisons to other algorithms in this aspect. In addition, research articles often fail to include categorizations of their solutions concerning other RL properties and capabilities.

Introductory material often only categorizes algorithms sparsely or within the main RL families instead of consistently outlining and explaining the differences for various properties. In this paper, we discuss properties of modern RL approaches and outline respective implications for capabilities of RL algorithms, their variants, extensions, and families.

2 PROPERTIES DICTATED BY THE PROCESS

The properties discussed in this section can be mostly seen as algorithm properties which cannot be chosen freely, but are predefined by the Markov process which has to be solved. Further, value estimation and policy optimization methods are discussed.

2.1 Model-Free & Model-Based RL

A popular distinction in RL is between model-free and model-based algorithms. In contrast to standard machine learning terminology, the term "model" in model-free or model-based does not refer to the trainable algorithm but to a representation of the environment with knowledge beyond the observations. A model like this can include state transition probabilities, reward functions, or even optimal expected future return values for the environment at *training time*. Notably, this information does not necessarily have to be available at evaluation time for model-based RL algorithms. In self-play for example, the environment model during training is defined by a clone of the best version of the algorithm. At evaluation time, the algorithm is expected to have learned a policy that generalizes from its training experience to be able to handle other opponents. Model-based algo-

Schröder, K., Kastius, A. and Schlosser, R.
Welcome to the Jungle: A Conceptual Comparison of Reinforcement Learning Algorithms.
DOI: 10.5220/0011626700003396
In *Proceedings of the 12th International Conference on Operations Research and Enterprise Systems (ICORES 2023)*, pages 143-150
ISBN: 978-989-758-627-9; ISSN: 2184-4372

rithms use such information to simulate the environment without executing actual steps or taking actions. Some model-based algorithms do not use an environment at all. Dynamic Programming, for example, uses its knowledge about state transition probabilities and rewards to build a table of exact expected future rewards instead of stepping through an environment and making observations. Environment models can be learned or known by an RL algorithm. Another popular model-based algorithm with a given model is AlphaZero by DeepMind (Silver et al., 2017). According to (Achiam, 2018) model-based algorithms can improve sample efficiency but tend to be harder to implement and tune.

2.2 Policy Optimization & Value Learning

The following subsections discuss several algorithms, which consist of two categories, policy learning and value learning methods. The later mentioned algorithms DDPG, SAC, and REINFORCE are policy learning methods. In those, a parametric representation of the mapping from state to action is available and those parameters are adjusted to maximize the expected discounted reward of the policy (Sutton and Barto, 2018). Many of those methods also incorporate value learning, which can also be used on its on to derive policies. For value learning, the goal is to develop an estimation of the expected discounted reward given a certain action is performed in a certain state. This value is called the Q-value, which leads to Q-learning as large group of algorithms.

2.3 Finite & Infinite Horizon Problems

Whether a problem has a finite or an infinite horizon has many implications on which RL algorithms are applicable, on the objective functions of the usable RL algorithms, and on which learning strategies can be applied. The decisive factor behind all those implications is whether an algorithm follows a Monte Carlo (MC) or Temporal Difference (TD) approach (Sutton and Barto, 2018). This section introduces MC and TD methods. Additionally, the core ideas behind TD(n), TD(λ), and eligibility traces are examined.

The main features of MC methods are that they are *on-policy* by design and only work on finite horizon problems. The *current policy* plays the environment until a terminating state is reached, after which the realized discounted returns for the episode are calculated for each visited state. The realized discounted returns for a state can vary significantly in-between such trajectories, even under the same policy, because of stochastic environments or stochastic policies. Such MC returns are used by some Policy Gradient variants, like REINFORCE.

TD approaches do not need complete trajectories or terminating states to be applied. Therefore, they can be used on infinite horizon problems as well. The core idea of TD methods is to define target values for a value estimator $\hat{V}^{\phi}$ using some section of a trajectory between the states s_t and s_{t+n} by summing the realized rewards r (discounted by factor γ) and adding an approximation of the expected future return of the last state s_{t+n} using the current version of the value-learner. The target value is calculated by $G^{\phi}_{t:t+n} := \sum_{t'=t}^{t+n-1} \gamma^{t'-t} \mathrm{r}(s_{t'+1}|s_{t'}, a_{t'}) + \gamma^{n} \hat{V}_{\phi}(s_{t+n})$ and the *temporal difference*, also called approximation error is given by $G^{\phi}_{t:t+n} - \hat{V}_{\phi}(s_t)$ and is used to update $\hat{V}_{\phi}(s_t)$. Using the current version of a value-estimator for an update of the same estimator is called *bootstrapping* in RL. Algorithms that use the temporal difference over n steps like this are called n-step TD methods or TD(n) algorithms. TD(n) for all $n > 1$ is automatically on-policy learning, as the sequence of rewards used for the network updates depends on the policy. Therefore, the values learned by the value-learner reflect the values of states under the current policy (see Section 3.1). TD(1) algorithms can be trained in an off-policy or on-policy fashion, as single actions and rewards are policy-independent. The off-policy updates would be unbiased towards the current policy. DQN is an example of an off-policy TD(1) algorithm, while SARSA is an on-policy TD(1) method. Some RL libraries like Tianshou (Weng et al., 2021) include off-policy DQN algorithms with an n-step parameter. RAINBOW uses similar updates with different tricks, e.g. omitting off-policy exploration (see Section 3.2). While MC methods have high variance and no bias in their network updates, TD approaches have a low variance. They only consider a few realized rewards but add a significant bias towards their initial estimations by the bootstrapping term. Increasing the number of steps in TD(n) makes the updates more similar to MC updates, and bias decreases while variance increases.

Because of the bias-variance tradeoff and differences in episode lengths, for example, the best n in TD(n) is highly problem-specific. An approach called TD(λ) tries to combine the advantages of all TD(n) versions by creating a target value $G^{\lambda,\phi}_{t}$ that is combination of all possible TD(n) targets. TD(λ) can be explained and calculated from a forward or a backward view, but both variants are mathematically equivalent (Sutton and Barto, 2018).

The forward view combines all TD(n) targets

$G^{\phi}_{t:t+n}$ using an exponentially weighted moving average over all possible TD(n)'s and is defined by: $G^{\lambda,\phi}_{t} = (1-\lambda)\cdot\sum_{n=1}^{T-t-1}\lambda^{n-1}G^{\phi}_{t:t+n} + \lambda^{T-t-1}G_{t:T}$ for the finite horizon case. For infinite horizon problems, this definition includes an infinite sum, which is approximated in practice using the truncated λ-return (Sutton and Barto, 2018), which truncates the sum after many steps. Notably, for $\lambda = 1$ this formula reduces to the MC update $G_{t:T} = G^{1,\phi}_{t}$ and for $\lambda = 0$ it reduces to the TD(1) update $G_{t:t+1} = G^{0,\phi}_{t}$ for $0^0 = 1$. This forward version of TD(λ) can be described as an MC method because it needs to wait until the end of an episode before starting to compute the updates.

The backward view was introduced to overcome this limitation. Instead of looking *forward* on the trajectory to compute the network updates, it calculates errors locally using TD(1). It passes this information *back* to the previously visited states of the trajectory. This means that if a high error is discovered at step t of a trajectory, the state-values $\hat{V}_{\phi}(s_{t'})$ of the previous states $s_{t'}$ with $t' < t$ are adjusted in the same direction as the local error. Not all of the states visited previously are similarly responsible for the change in the value of state s_t. The backward version of TD(λ) uses *eligibility traces* to quantify the eligibility of past states for discovered state-value errors. In practice, such an eligibility trace is a vector e containing decaying factors for each previously visited state of the trajectory. Different decay strategies are possible, in which eligibility values of revisited states are increased differently.

Similar to TD(n) with $n > 1$, TD(λ) is also an on-policy approach and is theoretically incompatible with off-policy techniques like replay buffers.

2.4 Countable & Uncountable State Sets

Uncountable state sets are found in many real-world applications, i.e., at least one state parameter lives in a continuous space. All of the table-based predecessors of RL, like Dynamic Programming or tabular Q-Learning, fail under uncountable state sets, as an infinite amount of memory and time would be necessary to compute all state-values (Sutton and Barto, 2018). Those algorithms require a discretization of the state space, which can limit the solution quality. RL algorithms using neural networks as regression algorithms can take continuous values of states as inputs and have the ability to generalize over uncountable state sets after learning from a finite number of training samples.

2.5 Countable & Uncountable Action Sets

Like uncountable state sets, uncountable action sets are standard in continuous control tasks, where actions can be chosen from a continuous interval. Table-based algorithms like Q-Learning that calculate state-action values cannot be applied to problems with uncountable action sets, as an infinite amount of memory and time would be required for calculations across the whole table (Sutton and Barto, 2018).

On-policy (see Section 3.1) RL algorithms can be trained under uncountable action sets by learning the parameters to a parameterized distribution over a range of actions. Training *off-policy* RL algorithms under uncountable action sets incorporates new challenges. In regular off-policy RL, the policy network is trained to reflect a discrete probability distribution over the *countable* Q-values of each state. In the case of uncountable action sets, the Q-value distribution over the uncountable action set is neither fully accessible nor differentiable if the learner has trained to output single values to state-action input pairs. Generally, constructing the value-learners under uncountable action sets to output parameters to parameterized distributions is impossible, as the action values follow an unknown, possibly non-differentiable distribution. Training with arbitrary distributions would lead to bad state-value estimates[1]. Because of these limitations, deterministic policies must be trained in off-policy RL under uncountable action sets. DDPG for example, takes advantage of the differentiability of continuous actions output by deterministic policies (Lillicrap et al., 2015). The objective function of DDPG is defined as: $J_{\pi}(\theta) = E_{s\sim D}\left[\hat{Q}^{\phi}\left(\mu_{\theta}(s)|s\right)\right]$. The policy parameters θ are trained to choose the Q-value-maximizing actions for the state $s \sim D$ of the state distribution D of the off-policy data. Gradients of the policy network parameters are backpropagated through the Q-learner $\hat{Q}^{\phi}$, through the continuous action policy output $\mu_{\theta}(s)$ and into the policy network. Gradient calculations like this would not be possible for discrete actions.

Some algorithms take one further step further and make it possible to learn *stochastic* policies under uncountable action sets. One example of those algorithms is SAC (Haarnoja et al., 2018b), which introduces some beneficial additions which improve learning performance when compared to DDPG.

[1] In policy optimization for uncountable action sets, the learned distribution does not have to accurately reflect the distribution of the true Q-values, as it is not used to estimate state-values.

The objective of the SAC policy is given by:

$$J_\pi(\theta) = E_{s\sim D}\left[D_{KL}\left(\pi_\theta(\cdot|s)\middle|\middle| e^{\hat{Q}^{\phi}_{\pi_\theta}(\cdot|s)}/Z_\phi(s)\right)\right]$$

It contains a KL-divergence term, initially proposed by (Haarnoja et al., 2018b), that aims at minimizing the difference between the distribution of the policy π_θ in a state s and the distribution implied by the Q-value estimations $\hat{Q}^{\phi}_{\pi_\theta}(\cdot|s)$. $Z_\phi(s)$ is a normalization term that does not contribute to the gradients and can be ignored. Negating the remaining terms results in the equivalent maximization objective:

$$\mathrm{J}'_\pi(\theta) = E_{s\sim D, a\sim\pi_\theta}\left[\hat{\mathrm{Q}}^{\phi}_{\pi_\theta}(a|s) - \log\pi_\theta(a|s)\right] \qquad (1)$$

This equation demonstrates, that SAC's policy is trained to maximize both the expected Q-values, as well as its entropy $\mathcal{H}(\pi_\theta(\cdot|s)) = -E_{a\sim\pi_\theta}[\log\pi_\theta(a|s)]$. Although the Q-values in SAC already contain an entropy coefficient, optimizing the policy to maximize the Q-values would not guarantee that the policy itself has high entropy. The policy might choose the Q-maximizing actions with 99% probability density, i. e., a very low entropy, which is why this entropy term is also necessary for the policy objective.

A problem with the formulation in (1) is the expectation over the actions $E_{s\sim D,a\sim\pi_\theta}$, which depends on the policy parameters θ (Achiam, 2018). Notably, with countable action sets, this is no problem. The expected value of the Q-values in a state s can be calculated precisely using the finite sum $\sum_{a\in A}\pi(a|s)\hat{\mathrm{Q}}^{\phi}_{\pi_\theta}(a|s)$. For uncountable action sets there is usually no way to calculate the expectation precisely. Instead, it is approximated using samples. This leads to a high variance in the resulting gradients, which can be avoided by a reformulation of the problem using the reparametrization trick, in which the source of randomness is externalized. Using the reparametrization trick, the objective can be further rewritten, which exchanges the expectation over the actions with an expectation over a random sample from the standard normal distribution $\varepsilon \sim \mathbb{N}$:

$$\mathrm{J}'_\pi(\theta) = E_{s\sim D,\varepsilon\sim\mathbb{N}}\left[\hat{\mathrm{Q}}^{\phi}_{\pi_\theta}(\tilde{a}|s) - \log\pi_\theta(\tilde{a}|s)\right],$$

where $\tilde{a} = f(\varepsilon, \pi_\theta(\cdot|s))$. The reparametrization function f uses the policy network outputs to transform this random sample from a standard normal distribution to the distribution defined by the policy network. This is done by adding the mean and multiplying with the variance produced by the policy network. This separates sampling from the policy distribution and reduces the variance of the computed gradients.

3 ALGORITHM PROPERTIES

Next, we categorize different aspects of RL algorithms and discuss their learning dynamics.

3.1 On-Policy & Off-Policy RL

This subsection explores the differences between on-policy and off-policy for value-learning RL algorithms like DQN. Afterward, importance sampling and its applications for off-policy learning in policy optimization methods are discussed.

In value-learning, the difference between on-policy and off-policy is best explained by the type of state-action values learned. There are two main options for learning state-action values.

One possibility is learning the expected state-action values *of the current policy*. For this case, in each learning step, the Q-values reflect expected returns under the current policy. This kind of policy is optimized by repeatedly adjusting the parameters slightly in a direction that minimizes the difference to a target value and reevaluating the expected returns. This is the main idea behind the Q-Learning variant SARSA. Algorithms like SARSA are called on-policy, as the current policy needs to be used to collect new experiences, and no outdated or unrelated data can be utilized. This dependency on policy-related actions is also reflected in the Q-value update function which uses the *realized* action in the Bellman expectation equation instead of an aggregation over all the possible actions. Notably, not all algorithms that learn values under the current policy are on-policy methods. The critics in SAC learn Q-values *of the current policy* in an off-policy way by inputting the succeeding states (s_{t+1}) of the off-policy experience into the current *policy network*. The result is a probability map over the actions that can be taken in s_{t+1}. The value of state s_{t+1} under the current policy can be estimated as the sum of the Q-values in s_{t+1} weighted by the calculated distribution.

The second possibility to approach value-learning is to learn the *optimal* state-action values achievable *in the environment* and independent of the current policy. This is done by Deep Q-Learning (Mnih et al., 2013). A deterministic policy is implicitly defined by maximizing actions, and stochastic policies can be formulated by applying temperature-regulated softmax over the state-action values. This is called off-policy learning, as state-action values like this can be learned using policy-unrelated experience from the environment. While in some cases, the data is generated using a greedy version of the current policy, completely policy-unrelated experience can be used, for example, from a human player. All data points can even be trained multiple times, as they are never outdated w. r. t. the state-action-value definition.

The optimization objectives of pure policy optimization methods are usually defined as maximizing the expected future return when following the *current policy*. Because of this, most pure policy optimization algorithms are naturally on-policy methods. During training, states and actions are sampled from the current policy, and probabilities or actions are adjusted according to their realized or expected future return. Without adjustments, using state samples from the stationary state distribution of the current policy is crucial. While it is technically possible to collect policy-independent experience from the environment and use the policy log probabilities of the chosen actions for policy updates, this optimization objective would maximize the expected value under the wrong state distribution. With this setting, the policy might visit different states when performing independently and achieve far-from-optimal expected future returns under its state distribution.

One way to overcome this limitation and train a policy optimization algorithm using data that was collected from a different state distribution is *importance sampling* (Sutton and Barto, 2018). The importance sampling theorem states that the expected value of any deterministic function f when drawing from a distribution π_θ is equal to the expected value of f multiplied by the ratio of the two probabilities when drawing from a distribution D.

In the RL setting, the expectations of the policy optimization objectives can not be calculated precisely and are approximated using a batch of samples. By applying the importance sampling equation, the expectation can be approximated with samples from an *arbitrary*, known state distribution. As per the central limit theorem, the approximations with both methods follow a normal distribution around the actual expected value. Still, they can have significant differences in variance if the two distributions are very different (Sutton and Barto, 2018). Because of this unreliability, to our knowledge, there are no pure off-policy optimization algorithms Instead, some Monte Carlo RL variants use importance sampling to reduce simulation and calculation overhead for environments with long trajectories. For those cases, simulating the realized rewards for just a single network update is inefficient. Using importance sampling, the latest trajectories can be reused for multiple network updates if the policy does not change drastically within a few updates.

Actor-critic algorithms can be designed to operate on-policy or off-policy. The on-policy actor-critic variants use a policy optimization objective similar to pure policy optimization algorithms (Sutton and Barto, 2018). In this case, the policy is improved by altering the log probabilities of actions according to their learned Q-values or advantage-values. Off-policy actor-critic variants are possible by training the policy to reflect the action distributions implicitly defined by the learned Q-values. This is a valid objective, as the policy is trained to choose the Q-value-maximizing action in *every* state.

The choice between on-policy and off-policy algorithms primarily affects its sample efficiency and the bias-variance tradeoff of the network updates (Fakoor et al., 2020). On-policy algorithms need to consistently generate new data using their current policy, making them sample inefficient. Accordingly, they are less suited for problems with high time complexity environments. Off-policy algorithms can operate on problems with arbitrarily generated experience, even without direct access to the environment.

Off-policy algorithms tend to be harder to tune than on-policy alternatives because of the significant bias from old data and value-learner initializations (Fujimoto et al., 2018).

3.2 Stochastic & Deterministic Policies

RL algorithms are designed to learn stochastic or deterministic policies. While most stochastic methods can be evaluated deterministically by selecting the actions with the highest probabilities, there is no sophisticated way of converting deterministic solutions to stochastic variants other than applying ε-greedy actions or temperature-controlled softmax.

It could be argued, that deterministically choosing the optimal action could outperform stochastic policies, as they tend to diverge from the optimal policy. In practice this does not hold, e.g., SAC outperforms DDPG in its peak performance (Haarnoja et al., 2018a). Also, there are problems with imperfect information that can only be reliably solved by stochastic policies, as demonstrated by (Silver, 2015).

Stochastic policies have advantages over their deterministic alternatives in multi-player games, as opponents can quickly adapt to deterministic playstyles. A simple example of such a game is rock-paper-scissors (Silver, 2015).

3.3 Exploration-Exploitation Tradeoff

The exploration-exploitation tradeoff is a dilemma that not only occurs in RL but also in many decisions in real life. When selecting a restaurant for dinner, one can revisit a favorite restaurant or try a new one.

By choosing the favorite restaurant, there is a high likelihood of achieving the known satisfaction, but without *exploring* other options, one will never know whether there are even better alternatives. By only exploring new restaurants, the pleasure will generally be lower in expectation. The same concept applies to RL as well. The agent needs to balance exploration (for finding better solutions) with exploitation (to direct exploration and to obtain optimal solutions).

Many different exploration-exploitation strategies exist, including ε-greedy exploration, upper confidence bounds exploration, Boltzmann exploration, maximum entropy exploration, and noise-based exploration (Weng, 2020).

Upper confidence bounds exploration introduces a notion of confidence to Q-value estimations. Actions are chosen based on the sum of each Q-value and an individual uncertainty value. The uncertainty value is inversely proportional to the number of times an action was taken. The sum of a Q-value and its uncertainty value represents a confidence bound of that Q-value, i. e., Q-values of actions that have rarely been trained have high uncertainty and could be much larger in reality. Upper confidence bounds encourage exploration of rarely visited actions while considering their current Q-value estimation. In maximum entropy exploration, the agents' objective functions are extended by an entropy term that penalizes the certainty of the learned policy. Algorithms like SAC use this type of exploration. Some settings are especially challenging for exploration to find better solutions consistently. For example, very sparse or deceptive rewards can be problematic, which is called the hard exploration problem. (Weng, 2020) and (McFarlane, 2018) provide further information on exploration challenges and possible solutions.

3.4 Hyperparameter & Robustness

In RL, hyperparameter sensitivity characterizes how much an algorithm's performance depends on carefully tuned hyperparameters and how much the hyperparameters need to be adjusted between different problems. In other words, it describes the size of the hyperparameter space that generally produces good results. Many publications of new RL algorithms mention hyperparameter sensitivity and claim or imply improvements compared to previous work; e.g., (Haarnoja et al., 2018b; **?**) note that their algorithm SAC is less sensitive than DDPG. (Schulman et al., 2017) also suggest in their publication of PPO that it is less sensitive than other algorithms. These papers fail to include concrete sensitivity analysis to support their claims.

Other researchers have published work on hyperparameter tuning and sensitivity comparisons on specific RL tasks (Henderson et al., 2018; **?**). Most emphasize the high sensitivity of all compared RL algorithms and the lack of generally well-performing configurations. These articles suggest a need for less sensitive algorithms to be developed and for more research on best-performing hyperparameters in different settings.

Robustness is used as an antonym to hyperparameter sensitivity in some articles (Schulman et al., 2017) to describe how successful an algorithm is on various problems without hyperparameter tuning. Most of the literature uses robustness as a measure of how well a learned algorithm can handle differences between its training and test environment (**?**). The second definition of robustness is specifically important for real-world applications that incorporate a shift between training and testing environments.

Robustness of algorithms in RL can generally be achieved in multiple ways. One option is to design a distribution of environments and optimize the average performance of an agent on multiple environment samples from this distribution. A second option is to create an adversarial setup, where an adversary is trained to adjust the environment such that the agent's performance drops. Hence, the agent constantly trains on different environments. Such setups are promising but not easy to implement. (Eysenbach and Levine, 2021) show that maximum entropy RL algorithms like SAC are robust to environment changes, as the respective agents learn to recover from disturbances introduced by "surprising" actions.

3.5 Learning Stability

The learning stability of an algorithm characterizes its tendency to forget intermediate best-performing policies throughout training. This is sometimes referred to as *catastrophic forgetting* in the literature (Géron, 2019), although catastrophic forgetting is also used in the context of sequentially training a model on different tasks (Kirkpatrick et al., 2017).

Learning instability in policy optimization algorithms is mainly caused by noise in the gradient estimations, producing destructive network updates. Such noise is usually created by a high variance in the gradient estimates. Therefore, learning stability can be increased by choosing RL algorithms with low-variance gradient estimators. Another option to prevent destructive updates to the policy during training is to limit the change of the policy in-between updates, as done by PPO. A less sophisticated alternative with similar effects is to clip the gradient norms before using them in network updates. This is called gradient norm clipping. (Nikishin et al., 2018) propose to transfer stochastic weight averaging (SWA) to the RL setting to increase learning stability. SWA has improved generalization in supervised and unsupervised learning and is based on averaging the weights of the models collected during training.

4 SUMMARY & RECOMMENDATIONS

This section summarizes the key takeaways for each RL property discussed in the Section 2 and 3.

(i) Model-Free & Model-Based RL. The categorization of model-free and model-based RL algorithms indicates whether an algorithm learns or is given additional knowledge of the environment beyond the observations. This allows simulations of the environment and wider updates and exploration compared to following one trajectory at a time. It can be especially powerful if acting in the real environment is expensive or an environment model is available anyway. Model-free algorithms are more popular, and model-based implementations are not supported by as many frameworks because of the problem-specifics.

(ii) Policy Optimization & Value Learning Methods. For many problems, there is only one alternative. For all others, it depends on the complexity of a problem's value function, if it is easier to learn a value function or directly search for the optimal policy.

(iii) Finite & Infinite Horizon Problems. Whether a problem has a finite or infinite horizon has implications on which options for value estimations are applicable. MC methods only work with finite horizons, as whole trajectories are necessary. MC targets for value estimators have no bias but high variance, especially with extended episodes. TD methods are applicable for finite & infinite horizons and allow tuning the problem-specific tradeoff between bias and variance. TD(λ) combines all TD(n) updates and ideally their problem-specific benefits and is technically an on-policy method, as it includes MC estimates. Truncating the infinite sum is possible for applying TD(λ) to infinite horizon problems. Forward and backward views can be used to calculate TD(λ) estimates. Usually, the backward view is applied with eligibility traces, as it allows online updates of the estimator.

(iv) Countable & Uncountable State Sets. Traditional table-based methods can only operate on countable state sets. Neural networks can generalize to continuous inputs with limited memory, making them suitable for problems with uncountable state sets.

(v) Countable & Uncountable Action Sets. Table-based algorithms are usually inapplicable for uncountable action sets, but RL with neural networks can be used. Most on-policy RL algorithms can handle both countable and uncountable action sets. Off-policy learning of uncountable action sets includes additional challenges because the distribution type of the action values of a state is unknown. TD methods' state values cannot be calculated easily, and adding another expected value over these distributions is required in the objectives. Using action samples and the reparameterization trick, these expectations can be approximated and gradients can be backpropagated through the continuous actions into the policy network. Learning stochastic policies under uncountable action sets is also possible, as done by SAC.

(vi) On-Policy & Off-Policy RL. On-policy algorithms need to be trained with experience of the latest version of the policy. Off-policy alternatives can use and re-use any experience acquired in an environment. A significant advantage of off-policy algorithms is sample efficiency, but they tend to be harder to tune because of the bias-variance implications between the respective learning strategies. Learning *values* of states or state-action pairs *under the current policy* is usually done *on-policy*, in which case, all MC, TD(n), and TD(λ) methods can be used. SARSA is an example that uses TD(1) calculations. Only the TD(1) targets can be used if values under the current policy are learned *off-policy*. The bootstrapped value of the succeeding state can then be estimated by the current policy's probability distribution of that state and the respective Q-values, as done by SAC. Actual values within the environment are learned *off-policy*, and the maximizing action is chosen for updates. One example of such an algorithm is Deep Q-Learning.

Most policy optimization objectives are defined over the stationary state distribution of the current policy and therefore require on-policy training. Importance sampling still allows for off-policy gradient calculations in policy optimization but can significantly increase gradient approximation variance. Consequently, it is only used in special cases, for example, to execute a few consecutive network updates with just one batch of data. PER is an extension that can only be applied to off-policy algorithms.

(vii) Stochastic & Deterministic Policies. Some problems with imperfect information can not be solved by deterministic policies, in which case stochastic alternatives are more powerful. Deterministic policies or deterministic evaluations of stochastic policies tend to have higher total expected returns as only the best known actions are chosen. Stochastic policies have benefits in exploration and in multiplayer games, where unpredictability is beneficial.

(viii) Exploration-Exploitation Tradeoff. The exploration-exploitation trade-off is a natural dilemma when not knowing whether an optimum is reached in a decision process. Exploration is necessary to discover better solutions, but exploitation is needed for directed learning instead of random wandering. Many different exploration strategies exist, such as ε-greedy, stochastic exploration, maximum entropy exploration, upper confidence bounds exploration, etc. Some problems are especially challenging for exploration, for example, occurs in environments with very sparse rewards.

(ix) Hyperparameter Sensitivity & Robustness. The sensitivity of algorithms indicates how narrow the usable ranges of hyperparameters are and how well they can be trained on multiple problems with the same configuration. Sensitivity is often mentioned in the literature but rarely analyzed in detail. It is highly problem-specific, which makes studying sensitivity on multiple problems valuable. Algorithms like PPO can make algorithms less sensitive by limiting the incentives of drastic adjustments to the policy. The ability of an algorithm to cope with changes between training and test environment is referred to as robustness. It can be improved by training in multiple environments, creating adversarial setups, or applying maximum entropy RL.

(x) Learning Stability. The learning stability of an algorithm is an indicator for its tendency to forget intermediate best-performing policies throughout training. Low variance gradient estimators help improve algorithms' stability, and methods like PPO's objective function clipping can help prevent destructive updates. Additional options include gradient norm clipping and stochastic weight averaging.

5 CONCLUSION

To solve real-world problems with incomplete information, RL is a promising approach as it only requires a suitable reward function and no optimal data. Over the training process, the model incrementally builds better solutions by itself. Downsides of RL are the opaque selection of algorithms and their extensions as well as the more complicated tuning compared to supervised learning. In this context, we sorted out different RL algorithms' properties and inferred helpful guidelines to decide under which circumstances to apply which algorithm.

REFERENCES

Achiam, J. (2018). Spinning up in deep reinforcement learning. URL: https://spinningup.openai.com/.

Eysenbach, B. and Levine, S. (2021). Maximum entropy rl (provably) solves some robust rl problems. *arXiv preprint arXiv:2103.06257*.

Fakoor, R., Chaudhari, P., and Smola, A. J. (2020). P3o: Policy-on policy-off policy optimization. In *Uncertainty in Artificial Intelligence*, pages 1017–1027. PMLR.

Fujimoto, S. et al. (2018). Addressing function approximation error in actor-critic methods. In *ICML*, pages 1587–1596. PMLR.

Géron, A. (2019). *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* " O'Reilly Media, Inc.".

Haarnoja, T. et al. (2018a). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.

Haarnoja, T. et al. (2018b). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, pages 1861–1870. PMLR.

Henderson, P. et al. (2018). Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Kirkpatrick, J. et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Lillicrap, T. P. et al. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

McFarlane, R. (2018). A survey of exploration strategies in reinforcement learning. *McGill University*.

Mnih, V. et al. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Nikishin, E. et al. (2018). Improving stability in deep reinforcement learning with weight averaging. In *Uncertainty in artificial intelligence workshop on uncertainty in Deep learning*.

Schulman, J. et al. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D. (2015). Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/.

Silver, D. et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT press.

Weng, J. et al. (2021). Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*.

Weng, L. (2020). Exploration strategies in deep reinforcement learning. URL: https://lilianweng.github.io/.