Reinforcement Learning

(0) Summary

This document summarizes the learnings of the Reinforcement Learning class of the MBD at IE University, Madrid during the 3rd term 2021. It gives a broad overview over key concepts learned in the form of a Q&A document and applies learnings of the studied concepts to practical problems.

(1) What is RL and how is it different from other methods like Supervised or Unsupervised Learning (2 pts.)

While Reinforcement Learning is one of the oldest disciplines in Machine Learning, it didn’t fully take off until the publication of DeepMind’s Atari Game-playing system in 2013. Since then, a range of exciting developments have been brought forward by the RL community – among the most famous one was the victory of DeepMind’s AlphaGo System against the professional Go-player Lee Sedol in 2016 and against the world champion Ke Jie in 2017.

Reinforcement Learning is characterized by a software agent making observations and taking actions within an environment as a result of being incentivized by rewards for its actions. The objective of the agent is to learn over time how to maximize its reward.

Contrary to Supervised or Unsupervised Learning, in Reinforcement Learning the agent learns by itself to react to the environment and it does not have predefined or pre-collected data (labelled or un-labelled). In RL, the input are states and actions and data is collected on the go through a trial-and-error learning process. As such, in RL there is also no concept of first training and then testing as we know it from Supervised and Unsupervised ML. A challenge unique to RL (vs. supervised/unsupervised learning) is the exploration vs exploitation trade-off: In order to get a high reward, the agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. Contrary to many other methods, RL also considers the whole problem of the agent in an uncertain world – many other approaches address only subproblems in their application but do not take into account the entire big picture. In RL, the agent’s goal is an overall goal that takes into account the entire environment with all its possible actions and states and the impact of each. Finally, RL uses training information that evaluates the possible actions and its implications whereas other methods rather instruct by giving correct actions – correct behavior is already known in supervised learning whereas it needs to be explored in RL through trial-and-error.

Below table gives an overview of the three different methods:

	RL	Supervised	Unsupervised
Description	Uses live data to predict what strategy is best in order to achieve a goal while dealing with a exploration vs exploitation tradeoff	Uses a dataset to estimate a relationship between features and a label. Estimates are then used to unlabeled data to predict new labels	Uses a dataset to detect unknown patterns
Objective	Maximizing reward	Extrapolate, or generalize, its responses so it can predict correctly in situations not present in the dataset	Find a hidden structure and get clear separation
Input	States & Actions (data is collected on the go)	Data with Labels (complete dataset)	Data w/o Labels
Output	State/Action	Mapping	Classes
Limitations	RL is data hungry	Situations in which it is impossible to obtain representations of all possible situations	No precise information on data sorting

Possible applications for RL are:

- Robots: the agent controls a robot which takes in its environment through sensors, e.g. in rescue missions
- Gaming: the agent is incentivized to win the game through the collection of gaming points
- Thermostat: the agent controls the temperature on a smart thermostat and gets a negative reward whenever humans need to manually adjust the temperature
- Stock trading: the agent decides when to buy or sell stocks and receives a monetary reward (win/loss) for its actions

(2) What are Markov Decision Process (MDP) and Bellman's equation? (2 pts.)

Markow Decision Processes (MDP) are an idealized form of an RL problem which provides us precise theoretical results. It is based on the Markov Property, which states that the effects of an action taken in a state depend only on that state and not on the prior history (it has no memory). For example, my decision to take an umbrella with me before leaving the house depends only on the current weather and not on the weather the day before or before that.

We can use MDP to calculate where a process will be after n steps for problems where the number of possible states and actions is finite and very small (such that its value function can be represented as arrays and tables) and where the states and its transition probabilities do not change over time. To calculate MDPs, we have a finite set of possible world states and a finite set of possible actions as well as a reward function and a description of each action's effect in each state. With this information we can construct a Markov Chain that allows us to calculate the probability of reaching each final state by moving along the different states and considering at each state the transition probability to move into another state. At each state, only the current state (no previous states) determines the likelihood for the next state. The transition probability is an outcome of all combinations for the current state, the next state, the possible actions and the reward for each action.

Richard Bellman was the first to describe MDP in the 1950s and proposed a solution on how to estimate the optimal state value of any state, the one that gives the highest expected reward. The Bellman Optimal Equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to. It breaks down the value function into the immediate reward plus the discounted future values (discounted future rewards). The discount factor, generally between 0 and 1, determines the present value of the future rewards and defines how far to "look ahead".

(3) What is a Dynamic Programming (DP) method and how is it different from MDP methods? (2 pts.)

Dynamic Programming (DP) helps us to break down a complex problem into several subproblems which can be tackled subsequently. It refers to a collection of algorithms used to compute optimal policies in situations where we have a complete and accurate model of the environment (probability parameters and reward functions) in the form of a Markov decision process. The key idea is to use value functions to organize and structure the search for good policies.

A downside of DP is that it requires a complete and accurate model of the environment, which is hardly ever the case and also involve operations over the entire state set of the MDP, which can become computationally intensive.

DP algorithms can be classified into tow main subclasses: Value-iteration algorithms and Policy-iteration algorithms. Sometimes, Policy-search algorithms are mentioned as a third category.

- **Value-iteration algorithms** uses at its core the bellman equation to solve an MDP (objective: finding the utilities (state-value) for each state). Initially, arbitrary state-values are assigned (generally zero). Using the Bellman equation, utility is calculated and assigned for each state. This process is repeated indefinitely until an equilibrium is reached (reached often, to reach a stop-criteria has to be implemented). Once we have reached an equilibrium, we have obtained the utility values and can use them to estimate the best move for each state.
- **The Policy Iteration algorithm** focuses on finding an optimal policy that maximizes the expected reward by defining a policy which assigns an action to each state. Using random actions and Bellman equation, the expected state-value of the policy can be computed. The optimal policy is the one that has the highest reward. Policy iteration is guaranteed to converge with the optimal policy and state-value function being the current one at convergence.

So, Dynamic Programming is a way of solving a problem by defining the solution to a large problem recursively, in terms of sub problems - which themselves are defined in terms of their own sub problems, and so on. Markov Decision Process is a discrete stochastic process, it describes the problem in a more structured manner which consists of states, actions, rewards, epoch and transition probability from state i to j.

(4) What is a Multi-armed Bandit problem, and what are example applications of it in real-life? (2 pts.)

In multi-armed bandit problems, we have an agent that is repeatedly presented with multiple similar solutions to a problem and needs to solve which one will give the highest reward without knowing the probability distribution corresponding to each reward. After each choice, a reward is received based on the probability distribution of this action and the objective is to maximize this total (stochastic) reward over time by concentrating on those actions that yield the highest reward.

Multi-armed bandit algorithms are commonly used in practice today, among others for the following applications:

- **Internet display advertising:** deciding among several online display ads to display to site visitors with the objective of maximizing sales, trade-off between exploration (collect information on ad-performance) and exploitation (playing ad that has performed best so far)
- **Finance**: choosing among the stocks that yield the highest returns
- **Clinical trials:** exploring several treatments to identify the best out of several treatments (exploration) and treating patients as effectively as possible (exploitation), thus minimizing losses

- **Network Routing:** allocating network channels to the right users (i.e. telephone, internet) such that the overall throughput of the net is maximized

(5) Describe the following bandits methods, and how these work in high-level: (2 pts.) a. Epsilon Greedy, b. Upper Confidence Bound (UCB), c. Thompson Sampling

The challenge in a multi-armed bandit problems lies in finding a strategy that allows to maximize the reward. Since the true probability distribution is unknown, learning is carried out via trial and error and value estimation. Several algorithms have been developed to solve this challenge. All deal with the issue of exploration vs exploitation and the concept of maximizing reward and minimizing regret (the loss that we incur due to spending time on suboptimal “arms”)

- **Epsilon Greedy:** this algorithm performs very little exploration, time is mostly spend on exploitation. At each step, the algorithm chooses between exploitation (with probability epsilon) or exploration (with probability 1-epsilon). In most cases, it chooses exploitation.
- **Upper Confidence Bound (UCB):** This method is the most widely used solution for multi-armed bandit methods. It keeps optimism to explore uncertain options – either justified and it receives a positive reward, or unjustified and the algorithm will eventually learn. Ultimately, it favors actions that are very likely to provide the optimal value. Examples of this algorithm class are Hoeffding’s Inequality, UCB1, Bayesian UCB.
- **Thompson Sampling:** also called Posterior Sampling or Probability Matching. This method is the most complex of all solutions to the multi-armed bandit problems. Contrary to the previous methods, Thompson Sampling does not select its actions based on current averages of the rewards received from those actions. Instead, it builds up a probability distribution from the obtained rewards for each bandit iteratively and samples from this to choose an action. This way, we obtain an increasingly accurate estimate of the possible award retained but also increasing confidence over time (also known as: Bayesian Inference).

Most multi-armed bandit problems have the assumption of being stationary: at each timestep, the bandits and distributions of the rewards stay the same. Non-stationary bandit problems require more complex solutions. Additionally, the problem can become even more complex when there are multiple situations (contexts), these are called contextual bandits.

(6) What is the difference between Model-based and Model-free RL methods?, describe a method for each (2 pts.)

Model-free vs model-based is one of the key distinctions between algorithms in RL and is determined by the question of whether the agent has access to or learns from a model of the environment.

Model-based RL methods construct a model of the environment which the agent then uses to predict how the environment will respond to its actions in the form of one (sample model) or several (distribution model) probabilities. Model-based RL methods mainly rely on planning as their primary component, the agent can see what would happen if choosing a particular action and can explicitly decide between these options and can put these learnings into a learned policy.

On the other hand, there are also reinforcement methods that do not need an environment model at all. Those models do not provide any guidance on how certain actions will impact the environment in which they act; they do not contain a transition-matrix that guides transitioning from one state to another (depending on an action taken and the underlying value function) nor a reward function. Instead, model-free methods mainly rely on learning. Model-free methods can be advantageous in situations in which not sufficient information is available to build a complete environment. Given that this is most often the case in practical applications, those are also considered as “the real world” whereas model-based methods are referred to as simulations.

Despite the differences, both methods compute a value function and are based on looking into what lies ahead to approximate this value function.

Within model-free and model-based RL, a further distinction can be made between on-policy and off-policy. Examples for model-free methods are: Temporal Difference-Learning (on-policy) as well as Q-learning (off-policy, s. Question 7+9).

- **Temporal Difference-Learning** is very similar to the Value Iteration algorithm (s. Question 3) but adapted such that the agent knows only part of the Markov Decision Process (mostly: only possible states and actions, nothing else). The agent then uses an exploration policy to explore the MDP and updates the estimates of the state value over time with the observed transitions and rewards, it is learning. For each state, the agent keeps a running average of the immediate rewards it received by leaving this state plus the expected future reward (assuming it’s optimal). This makes it advantageous in that it can learn from incomplete episodes and we don’t need to track the episode until it is terminated.

Examples for model-based methods are: Adaptive Dynamic Programming (on-policy) as well as Adaptive Dynamic Programming with proper exploration function (off-policy)

- **Adaptive Dynamic Programming (ADP):** ADP is considered as an effective intelligent control method and has played an important role in seeking solutions for the optimal control problem. An ADP agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using dynamic programming. It has two main forms: Policy Iteration and Value Iteration. Both require an initial control policy which is then iteratively updated.

(7) What is the difference between On-policy and Off-policy RL methods?, describe a method for each (2 pts.)

On-policy and off-policy are both dealing with the issue of exploring starts: How can we ensure that the agent selects all actions infinitely often? There is two approaches to this: on-policy and off-policy methods.

On-policy RL methods focus on improving or evaluating the policy that is used for making decisions, it "learns on the job". Off-policy methods, on the other hand, improve or evaluate another, a second policy than the one that is used to generate data. They "learn over someone's shoulder".

Generally, on-policy methods are generally simpler and more stable than off-policy methods, however also less sample efficient, meaning that on-policy methods need more data for learning. Off-policy methods on the other hand require additional concepts and are more complex. Also, since the data comes from a different policy, off-policy methods often have greater variance and are slower to converge but are nevertheless more powerful and general.

Examples for on-policy methods are Policy Iteration, Value Iteration (both described in Question 3), Monte-Carlo for On-Policy (see Question 8), Sarsa (see Question 9) etc.

Examples for off-policy methods are Q-learning, expected Sarsa or Adaptive Dynamic Programming with proper exploration function.

- **Q-Learning:** Q-learning is an adaptation of the Q-Value-Iteration such that transition probabilities and rewards are initially not known. The agent plays and gradually improves its estimates for the Quality Value (the optimal state-action value). Once it has more or less accurate Q-Value estimates, the optimal policy is to choose the action with the highest Q-Value (e.g. the greedy policy). For each state-action pair the algorithm keeps a running average of the reward that it received by leaving this state with a specific action plus the expected future reward. We can estimate this sum by taking the maximum of the Q-Value estimates for the next state, assuming that the target policy would act optimally from this on. In Q-learning, the policy being trained is not necessarily the policy being executed, which is why it is being called an off-policy algorithm.

(8) What are the advantages and disadvantages of Monte Carlo (MC) methods vs Temporal Difference (TD) methods? (2 pts.)

Monte Carlo is a method for solving RL problems based on averaging sample returns (we consider here only episodic tasks in the sense that we have several episodes and all episodes end at some point to ensure that well-defined returns are available). In Monte Carlo Problems, it is not necessary to have complete knowledge of the environment – instead only experience is required which is obtained in the form of sample sequences of states, actions, and rewards from interacting with the environment. In this sense, Monte Carlo is similar to bandit-methods where we also sample and average rewards for each action. However, in Monte Carlo we consider multiple states with each basically acting as a different bandit problem and the bandit problems are interrelated in the sense that the return after taking an action in one state depends on the actions taken in later states in the same episode. As such, we are in a nonstationary problem where the action selections are undergoing learning.

The primary goal in Monte Carlo Methods for control (when approximating optimal policies) is to estimate the function Q of a state under a policy. The function Q returns the utility of a state-value pair and it helps us in directly estimating the value of a given action.

Monte Carlo methods differ from Temporal Difference Methods (s. Question 6) in several aspects:

- **TD can learn before knowing or even in the absence of a final outcome:** TD learns online after every step, regardless of whether the sequence has been finished or not. As such, it works in continuing (non-terminating) environments. Monte Carlo on the other hand can only estimate the return once an episode has ended, as such it can only learn from finished sequences and only works in episodic (terminating) environments
- **Monte Carlo has high variance and zero bias:** Monte Carlo shows very good convergence properties (even with function approximation). It is almost insensitive to the initial value and very user-friendly (simple to understand and use). T
- **TD has low variance and some bias:** TD on the other hand is usually more efficient than MC. Simple TD (TD(0)) has been shown to converge to the optimal policy, although not always with function approximation. However, contrary to MC it is more sensitive to the initial value.

(9) What is SARSA and how is it different from Q-learning? (2 pts.)

SARSA is a model-free, on-policy Temporal Differencing control method – we can use it to estimate the optimal policy from starting with a random policy. SARSA learns an action-value function, considering transitions from state-action pair to state-action pair, in a Markov chain manner with a reward process. Updates are performed after each transition from a non-terminal state and it uses each of the following input factors: state, action, reward, next state and next action (which are responsible for its name). To perform the optimal policy, the Q-function is used. SARSA deals with the trade-off between exploration and exploitation by avoiding the mistakes due to exploration. How does it work? In SARSA, the agent selects one action from the policy and makes a move. It then observes the reward, the new state and the associated action while the algorithm then updates the action-value function through the update rule. At each step, the policy is updated and following a greedy policy the action with the highest action-value is chosen (same as MC for Control), applying the exploring starts condition.

An alternative to on-policy SARSA in high-action space is to use Expected SARSA (off-policy), which takes the average of the Q-values of the possible actions with the probability estimated by the policy used.

The key difference between regular SARSA and Q-learning is that SARSA is on-policy while Q-learning is off-policy.

The underlying algorithms are quite similar however SARSA considers one more step to get to the next action than Q-learning. It replaces the maximization of Q-Learning by expectation over the probability distribution of actions by any exploration policy.

In a typical application of a robot getting from its origin A to a destination B, Q-learning computes the shortest path, it doesn't check if this action is safe or not. SARSA on the other hand discounts the value of these actions and thus finds a safer path – it is exploring different options.

(10) Describe the following RL algorithms and how these work in highlevel:(4pts.) - a. Proximal Policy Optimization (PPO), b. Deep Q-Learning, c. Asynchronous Advantage Actor Critic (A3C), d. Random Network Distillation (RND)

---

a. Proximal Policy Optimization (PPO)

PPO (Proximal Policy Optimization) is an algorithm that is based on AC2 (a variant of the below described AC3 algorithm that removes the asynchronicity, all model updates are synchronous). As its name suggests, it is a policy optimization algorithm which learns a parameterized policy without consulting a value function and mainly deals with the question: "how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse?". It has been celebrated as a major-break through in the search for finding the optimal step size, given that Policy Gradient methods are very sensitive to the choice of the step size and often have low sample efficiency. PPO balances ease of implementation, sample complexity, and ease of tuning, by computing an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip with the latter being most commonly used. Its key feature is the clipping of the loss function (regularization) to avoid excessively large weight updates (which often leads to training instability). PPO trains a stochastic policy on-policy, meaning that it explores by sampling actions according to the most recent version of its stochastic policy. The initial conditions and training procedure determine how randomly it choses its actions with a decreasing amount of randomness over time given that the update rule encourages the policy to exploit already found rewards. This, however, may cause the policy to get trapped in a local optima.

b. Deep Q-Learning

The main problem with Q-learning is that it does not scale well to large or even medium MDP (Markov Decision Process) with many states and actions. For example, let's image we use Q-Learning to train an agent to play PacMan. With about 150 pellets that can be eaten and each one being either still there or already gone (eaten), there is ca. $2^{150} \approx 10^{45}$ possible states. If we now want to add all possible combinations of positions for all the ghosts and Pac-Man, the number of possible states becomes beyond comprehensible.

Deep Q-Learning provides a solution to this problem: it finds a function that approximates the Q-Value of any state-action pair using a manageable number of parameters ("Approximate Q-Learning") and a Deep Q-Network (DQN). To approximate the Q-Value we use the fact that every Q-function for some policy obeys the Bellman equation, which breaks down the Q-value into the reward plus future discounted value estimate. DQN is most often used together with Experience Replay, for storing the episode steps in memory for off-policy learning, where samples are drawn from the replay memory at random.

c. Asynchronous Advantage Actor Critic (A3C): A3C is a variant of an actor-critics algorithm. Actor-critic algorithms combine Policy Gradients with Deep Q-Networks. An actor-critic agent contains two neural networks: a policy net and a DQN. The DQN is trained by learning from the actor's experience while the policy net learns differently: instead of estimating the value of each action over multiple episodes, summing the future discounted rewards for each action and then normalizing them, the agent (actor) gets action value estimates from the DQN (critic). It is often compared to a child (actor) learning and the mother (critic) giving feedback.

A3C is a variant of actor-critics algorithms and was introduced in 2016. It is a simple scalable method to parallelize the training of RL algorithms by using asynchronous gradient ascent and combines a few key ideas: (1) An update strategy that uses segments of experience (of fixed length) to compute estimators of the return and advantage function. (2) Asynchronous policy updates, and (3) An architecture that shares layers between the policy and value function. The A3C name stands for

Asynchronous Advantage Actor-Critic and describes its key attributes: in an A3C multiple agents are learning in parallel, each agent is exploring different copies of the environment. The agents update a target network periodically, but asynchronously (hence the name), with their learned parameters from exploration by fetching the current weights from the global network, performing some independent exploration and training in the environment for n steps and then sending back the learned weights to the global network, at a unique “asynchronous” time. When an agent sends back an update in A3C model, the global network updates itself and the agent then syncs itself with the newly updated global network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Training is additionally stabilized by the DQN estimating the advantage of each action instead of estimating the Q-values, thus not only determining how good the agent's actions were but also how much better the agent’s actions were than expected. This provides the ability to focus on improving predictions.

d. Random Network Distillation (RND)

Random Network Distillation (RND) was introduced by OpenAI in 2018 as a new approach to develop curiosity in RL agents using two neural networks (fixed and predictor) that learn previously-visited state and give smaller rewards for visiting them again. It is a prediction-based method and exceeded upon its publication average human performance on Montezuma’s Revenge.

Curiosity has been approached from different angles in RL and the idea behind is to build a reward function that is intrinsic to the agent (generated by the agent itself) such that the agent is both the learner and giving itself feedback. It will motivate the agent to explore the environment by going to states that are novel or unfamiliar.

One approach to curiosity is RND, which incentivizes visiting unfamiliar states by measuring how hard it is to predict the output of a fixed random neural network on visited states. In unfamiliar states it’s hard to guess the output, and hence the reward is high. RND differs from previous methods in that it is not attracted by the stochastic elements of an environment (e.g. a TV playing random pictures and thus distracting our agent with a continuous, stochastic flow of novel images). An RND has two networks:

- A target network with fixed, randomized weights which is never trained and randomly initialized. This network generates a feature representation for every state which will always stay the same
- A prediction network that tries to predict the target networks output

The target network outputs a fixed feature representation for the next state and the predictor network tries to predict the target’s network output for the next state (instead of predicting the next state given our current state and action). The prediction network is trained to minimize the difference mean square error between the actual output and its prediction. By removing the dependency on the previous state, the agent will not get stuck on a random noise, such as a TV with an unpredictable next frame on a screen.

RND can be applied to any reinforcement learning algorithm, is simple to implement and efficient to scale.