UCL&Deepmind COMPM050: Reinforcement Learning - Notes

Tomasz Bartkowiak
https://github.com/bartkowiaktomasz

June 2019

Contents

1 Introduction to Reinforcement Learning

1.1 About Reinforcement Learning

1.1.1 Reinforcement Learning Paradigm

1. No supervisor, only reward.
2. Feedback is delayed, reward might be given after some time.
3. The agent influences the data it receives.

1.1.2 The Reinforcement Learning Problem

The reward is a scalar signal, and the goal of an agent is to maximize total future cumulative award. The history H is a sequence of observations, actions, rewards. The algorithm is a mapping from the history to the next action A. The history might be long and complex, so state S is used to represent that history and is used to determine what action should be performed next: $S = f(H)$.

There is an agent state and an environment state. The agent does not usually have access to the latter so it makes actions based solely on the agent state. An information state a.k.a. Markov state contains all useful information from the history. A state S is Markov iff:

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, ...S_t)$$

meaning that the future state is only dependent on the current state, not on the history.

The future is independent of the past given the present.

The environment state, by the definition is Markov.

Full Observability means that the agent sees the environment state. Here, agent state = environment state = information state, which is called a Markov Decision Process (MDP).

Partial observability means that the agent does not see a complete environment state. This is called a Partially Observable MDP (POMDP).

The agent needs to construct its own state representation. This can be done in different ways, e.g. based on the whole history, beliefs of environment state, i.e. a probability distribution among different possible environment states, a recurrent neural network, i.e. weighted sum of previous state and current observation.

1.1.3 Inside an Reinforcement Learning Agent

An RL Agent might include some of these components:

1. Policy: agent's behaviour function (map from a state to action) - it can be deterministic (given some state make some action) or stochastic (probability distribution over actions given state).
2. Value function: how good is each state and/or action (prediction of expected future reward).
3. Model: agent's representation of the environment (it predicts what the environment will do next). There are usually two models: Transitions model (predict the next state) and Rewards model (predict the next, immediate reward).

The agent can be:

1. Value-based (value function)
2. Policy-based
3. Actor-critic (both value-based and policy-based)

or:

1. Model-free (based on policy and/or value function), without trying to figure out how the environment works.
2. Model-based - first we build a model of an environment.

1.1.4 Problems within Reinforcement Learning

There are two problem settings with sequential decision making in RL:
Reinforcement Learning: the environment is initially unknown and the agent improves its policy via the interaction with it.
Planing: the model of the environment is known and the agent performs computations with its model. There is an exploration vs. exploitation trade-off.

2 Markov Decision Process

2.1 Markov Processes/Markov Chain

Markov Decision Process formally describes a fully observable environment in RL. However, partially observable problems can be converted into MDPs. Markov Chain is a tuple $< S, P >$ where S is a finite space of possible states and P is a probability (transition) matrix of switching from one state to another (future) state.

2.2 Markov Reward Process

MRP is a tuple $< S, P, R, \gamma >$, where R is a reward function and γ is a discount factor. The return G_t is the total discounted reward from time-step t. The value function $v(S)$ gives the long term value of for the state S. It is the expected value of the total reward given you find yourself in the state S . The Bellman Equation describes the value function vector as a sum of immediate rewards R and the product of transition matrix P and the value function vector $v : v = R + \gamma P v$. It is a linear equation and solving for v is $O(n^3)$ for n states (requires the inversion of a matrix). We deal with it using iterative methods:

1. Dynamic Programming
2. Monte Carlo evaluation
3. Temporal-difference learning
4. Markov Decision Process

MDP is a tuple $< S, A, P, R, \gamma >$, where A is a finite set of actions. For each action a there is a separate transition matrix P and the reward function R. A policy π is a probability distribution over actions given states. They fully define the behaviour of an agent and are only dependent on the current state. An action-value function $q_\pi(s, a)$ is the expected return of starting at state s, performing an action a and then following a policy π.
Bellman Optimality Equation is nonlinear and does not have a closed form solution. The iterative solutions exist and include:

1. Value iteration

2. Policy iteration

3. Q-learning

4. SARSA

2.3 Extensions to MDPs

1. Infinite and continuous MDPs

2. Partially observable MDPs

3. Undiscounted, average reward MDPs

3 Planning by Dynamic Programming

3.1 Introduction

Policy Evaluation - Given policy, how good it is?
Policy Iteration - Takes an idea of evaluating policy to make it better.
Value Iteration - Works on improving the value function by applying a Bellman Equation iteratively.
Markov Decision Process satisfies the properties of dynamic programming: optimal substructure and overlapping subproblems through the Bellman Equation (recursive decompositions). The value function, on the other hand, acts as a cache of solutions.

3.2 Policy Evaluation

Problem: Evaluate given policy π. **Solution:** Iterative application of Bellman expectation backup. Given a policy (e.g random walk on a grid) we can iteratively apply Bellman Equation to find a value function (i.e. expected long-term reward for each state) which, itself might start pointing towards better policy.

3.3 Policy Iteration

Given a policy π we want to first evaluate it: $v_{\pi(s)}$ and then, greedily improve it with respect to v_π to get π' .

We can improve the policy after a specific number of iteration k or introduce a stopping condition, i.e. stop if the value function does not change more than ϵ . In the extreme case, if $k = 1$, this is called a value iteration.

3.4 Value Iteration

Value iteration is another mechanism for solving MDP.
Problem: Find optimal policy π.
Solution: Iterative application of Bellman optimality backup.
In value iteration there is no explicit policy (as in Policy Iteration), for the intermediate value functions there might not necessary exist a policy that leads to this value function.

3.5 Synchronous Dynamic Programming Algorithms

Problem	**Bellman Equation**	**Algorithm**
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

Prediction is about how much reward are we going to get given policy v_π. Control is about finding v_* and hence $\pi*$.

3.5.1 The Complexity of the algorithms, given m actions and n states

In each sweep we consider n states and for each state n possible successor states, for which we can take m actions, which gives $O(mn^2)$ per iteration. You can apply Bellman Equations for q (action-value function) just like for v, but this is $O(m^2n^2)$ per iteration (since you need to consider all action pairs).

3.6 Extensions to Dynamic Programming

Asynchronous DP: In-place DP, Prioritized sweeping, Real-time DP

DP uses full-width backups (we consider every action and all possible states that we might find ourselves after making that action). This is effective for medium-sized problems (millions of states), but for large problems it suffers from Bellman's curse of dimensionality, since the number of states grows exponentially with the number of state variables. We use sampling instead of considering every possible combination.

3.7 Contraction Mapping

For any metric space ν that is complete (i.e. closed) under an operator $T(v)$, where T is a $\gamma - contraction$ (i.e. makes function values closer by at least γ): T converges to a (unique) fixed point at a linear convergence rate of γ.

4 Model-Free Prediction

4.1 Introduction

Model-free prediction is about estimating the vale function (given policy) for an unknown MDP. Here we use value function v.
Model-free control is about optimizing value function (and thence finding an optimal policy) for an unknown MDP. Here we use action-value function q , as we have an access to the actions in control.

4.2 Monte-Carlo Learning

Goal: Learn a value function v_π (expected future return for any state) from episodes of experience under a policy π .
MCRL is about sampling (S, A, R , i.e state, action, reward) and assigning a mean return as a value for given initial state. In another words, you run some episodes and collect all the rewards and update the estimate of the return for each state you visit.

4.3 Temporal-Difference Learning

The difference between TD and MC is that TD learns from incomplete episodes (by bootstrapping), so it does not need to complete the whole trajectory (until the termination state is achieved), but the partial trajectory and then make an estimate of the remaining reward.

In TD learning we use an estimated return $R_{t+1} + \gamma v_{s_{t+1}}$ (also called a TD target) instead of the actual return G_t .

4.4 The properties of TD:

1. TD can learn before knowing the actual outcome and can learn online after each step.
2. TD can learn from incomplete sequences.
3. TD works in continuing (non-terminating) environments.
4. MC has high variance, zero bias, whereas TD has high bias, low variance.

TD exploits Markov property (is more efficient in Markov environments), whereas MC does not (and is more efficient in non-Markov environments). The contrast between TD/MC and Dynamic Programming is that MC/TD are sampling from the possible actions/rewards and update the value function, whereas DP does a full look-ahead at all possible actions and what the environment can do to us and compute a full expectation.

Bootstrapping: update involves an estimate - DP, TD (MC does not bootstrap, it uses real return values on the way). In DP, TD we bootstrap from our estimated value at the next step.

Sampling: update samples an expectation - MC/TD (DP does full-width exhaustive search).

4.5 TD(λ)

TD(λ) algorithm is a generalization of TD/MC, where we are somewhere in between the shallow backup (TD) and deep backup (MC). We can do 1, 2, ..., n MC steps: that would be called TD(1), TD(2), ... TD(n) respectively. In TD(λ), λ-return combines all n-step returns G and is a geometrically weighted average of all of them.
Backward-view TD(λ) will keep an eligibility trace (heuristic that combines both frequency and recency heuristics) and update the value function in proportion to the TD error and the eligibility trace.

5 Model-Free Control

5.1 Introduction

On-policy learning - Learn about the policy π by sampling from the same policy.
Off-policy learning - Learn about the policy π from experience sampled from μ (you do not have to learn about the behaviour that you sample).

5.2 On-policy Monte Carlo Control

Greedy Policy Improvement over $V(s)$ requires model of MDP, while greedy policy improvement over $Q(s, a)$ is model-free. In order to ensure exploration, we might use ϵ-Greedy exploration, which chooses a greedy action with probability of $1 - \epsilon$ and a random action with probability of ϵ. We are not sure, however, that $\epsilon - Greedy$ will lead to an optimal policy $\pi*$. To ensure it does, we must prove it is GLIE (Greedy in the Limit with Infinite Exploration). GLIE has two properties:

1. all state-action pairs are explored infinitely many times,

2. the policy converges on a greedy policy.

$\epsilon - Greedy$ has those two properties if we assume that ϵ decreases at each iteration, e.g. $\epsilon = 1/k$.

5.3 On-policy Temporal Difference Learning

5.3.1 MC vs. TD

Advantages of TD: Lower variance, online, incomplete sequences. Online means: ”you see one step of data, you bootstrap, you update your value function immediately”. The idea is to use TD instead of MC in the control loop: apply TD to $Q(S, A)$, use ϵ-Greedy policy improvement and update every step. This algorithm is called SARSA. Here we start with state S and given action A, we sample from the environment and get an immediate reward R, we end up in other state S' and sample a new action A' from our policy.
SARSA(λ) allows for extending SARSA in future by computing all n-step returns and weighting them accordingly. The problem is that this algorithm

is not online anymore as we need to go through the whole trajectory until terminal state is achieved (is such exists). We deal with that using Eligibility Traces. Eligibility Traces are the maps E(s,a) that store the value indicating how given pair was responsible for the reward we got. Whenever we experience a pair s,a , its eligibility trace increases and its value decays in time if we do not experience it. In SARSA(0) you only propagate the information back by one step per episode, but in SARSA(λ) you do it for many states weighted by their eligibility traces. In MC every state would be updated with the same amount.

5.3.2 Off-policy Learning

Off-policy learning is about evaluating target policy $\pi(a|s)$ to compute $v(s)$ or $q(s,a)$ while following behaviour policy $\mu(a|s)$. In off-policy, MC is extremely high variance, so we have to use TD (bootstrapping), but it requires Importance Sampling and has some variance as well. The idea that works best with off-policy learning is Q-learning. The idea of Q-learning is to update the q values in the direction of the best possible next q value you can have after one step.

6 Value Function Approximation

6.1 Introduction

For large MDPs, instead of finding the true value function $v(s)$ we will try to fit an approximate function $\hat{v}(s,w)$, which is a parametric function with some weight matrix **w**. Similarly, for the action value function $q(s,a)$ we find $\hat{q}(s,a,w)$.

6.2 Incremental Methods

Here we are going to train the approximate function using incremental method, e.g. SGD. It is useful to represent the state space as a feature vector, which is responsible for representing our state, e.g. the entries in the vector specify our x,y,z coordinate. In linear case (linear combination of features) this feature vector is used to compute the update of weights. In nonlinear case (neural networks) we need to compute the actual gradient. In order to train $\hat{v}(s,w)$, we need to know true $v(s,w)$ to compute error and back-propagate it using SGD. However, we do not know the true value function so we substitute it by some target.

1. For MC the target is the return G_t (we need to get to the end of our episode)
2. For TD(0) the target is $R_{t+1} + \gamma v(s_{t+1}, w)$.
3. For TD(λ) the target is the λ-return.

In MC we are essentially finding a mapping $S \to G$ given pairs S, G making it a supervised problem. G_t is an unbiased estimator of a true function value. In TD the target is a biased estimator of a true value function because when calculating an error of estimation for given state, we need to query the same network for the value for the next state.
For approximating action-value function the process is similar. The feature vector here represents both states and actions. In MC the target is again G_t and in TD(0) the target is $R_{t+1} + \gamma q(s_{t+1}, A, w)$.
In both cases the forward and backward view (with Eligibility traces) are equivalent!
Prediction
The problem with TD methods is that they might diverge for on-policy in nonlinear case (nonlinear function approximator, e.g. neural network) and for off-policy in linear and nonlinear case (it only converges for table lookup). However, Gradient TD converges in all of those cases.
Control
In Control, both MC Control, Sarsa, Q-learning and Gradient Q-learning can diverge in nonlinear case. Q-learning does not guarantee to converge even in linear case.

6.3 Batch Methods

In Batch Methods we collect a dataset of experience D (s, v pairs) and after that, we randomly select samples from that dataset and perform SGD using them (this is called experience replay). Important thing about the experience network is that it stabilizes neural network methods since it decorrelates the trajectories (highly correlated parts of the trajectory are one after the other). This is used in Deep Q-Networks (DQN), where experience replay and fixed Q-targets ensure that the function approximator does not diverge. Fixed

Q-targets means a second neural network used for storing parameters for some time (frozen), so that we do not "bootstrap directly towards the thing that we're updating at that moment, because that might be unstable". After some time we equalize those two networks and again, freeze one of them and use it to update the other.
Experience replay might need many iterations to find a solution. For linear value function approximation we can solve the Least Squares directly. We know that the expected change to the weights given an optimum is zero (we do not want to update anything). Given that knowledge we can find a closed-form solution. This requires inverting a matrix, which, for n features is $O(n^3)$ (it does not depend on the number of states anymore), or Sherman-Morrison for incremental solution in $O(n^2)$ if we want to invert $(A + uv')^{-1}$ and the inverse of A is known.
We end up with new algorithms: LSMC (Least Squares Monte Carlo), LSTD, LSTD(λ). They all always converge to the right policy both in on/off policy. In Control we have an LSPI (Least Squares Policy Iteration)

7 Policy Gradient Methods

7.1 Introduction

We need to parametrize the policy to give us a distribution of possible actions that we can undertake given some state:

$$\pi(s, a) = P(a|s, \theta)$$

and we need to learn the parameter θ , e.g. using neural network. We have a Value-based and Policy-based RL:

1. Value-based: Learnt value function and implicit policy (e.g. ϵ-Greedy),

2. Policy-based: No value function and learnt policy,

3. Actor-Critic: Learnt value function and learnt policy. Sometimes using the policy might be more compact, e.g. in Atari games instead of computing the value function (i.e. given move will give me a total cumulative future reward of R), you learn a policy to do a certain move when something happens.

7.2 Advantages and Disadvantages of Policy-based method

Advantages: Better convergence, Effective in high dimensional and continuous space, can learn stochastic policies.
Disadvantages: Typically converges to local (not global) optimum, Evaluating policy is inefficient and has high variance.
State aliasing means that the world is partially observed. In such cases stochastic policy might be better.

7.2.1 Policy Optimisation

Optimisation methods can be divided into:
Gradient-free methods, e.g.

1. Hill-climbing
2. Simplex/amoeba/Nelder Mead
3. Genetic algorithms

Gradient-based methods, e.g.

1. Gradient descent
2. Conjugate gradient
3. Quasi-newton

7.3 Finite Difference Policy Gradient

In Gradient ascent if we do not have an access to the closed form gradient we can compute it using finite difference method. For each parameter we perturb its value by a tiny bit ϵ and see what is the response in the function. However, this naive method is ineffective in high dimensional spaces. The solution would be to use techniques for random directions, e.g. Simultaneous perturbation stochastic approximation (SPSA) (they are also quite noisy though).

7.4 Monte Carlo Policy Gradient

Score function describes how sensitive is the Likelihood function to a parameter θ.
Softmax Policy is the alternative for e.g. ϵ-Greedy (Softmax, in general, is a policy that is proportional to some exponentiated value).
Gaussian Policy, where the action is sampled from a Gaussian distribution with mean of $\mu(s)$ (linear combination of state features) and variance σ^2.
Policy Gradient Theorem states that the policy gradient is the expectation of the score function multiplied by the long-term action-value function $Q(s,a)$ - *"you want to adjust the policy that does more of the good things"*.
Monte Carlo Policy Gradient (a.k.a REINFORCE) is an algorithm for finding parameters θ by updating parameters using Stochastic Gradient Ascent (SGA), using Policy Gradient Theorem and using v_t as an unbiased sample of $Q(s_t, a_t)$ (Q for policy π).

7.5 Actor-Critic Policy Gradient

The problem with MCPG is that it has high variance. In Actor-Critic methods, instead of using the return to estimate the action-value function we are going to explicitly estimate the action-value function using a critic.
The name Actor-Critic comes from the fact that the algorithm maintains two sets of parameters: Critic - updates action-value function parameters w, and Actor - updates policy parameters θ in the direction suggested by critic. Those algorithms follow an approximate policy gradient. The critic is solving the problem of policy evaluation, so we can use methods such as MC, TD(0), TD(λ), Least Squares Policy Evaluation etc. If you use Softmax function approximator you will achieve a global optimum. This is not guaranteed for a neural network.
Baseline Method is a method to decrease a variance of expectation without changing it. This is achieved by subtracting a baseline function $B(s)$ from the policy gradient.
Advantage function $A(s,a)$ tells us how much better than usual is it to take action a. The advantage function can significantly reduce the variance of policy gradient.
Natural Policy Gradient *"Adjust the policy in the direction that gets you more Q"*. It scales much better in high dimensions and is better than stochastic approach.

8 Integrating Learning and Planning

8.1 Introduction

Model, in RL is the agent's understanding of the environment. Recap: RL can be:

1. Model-free: no model, learn value function (and/or policy) from experience (the agent does not try to explicitly represent the transition dynamics of the reward function).
2. Model-based: learn a model from experience and plan a value function (and/or policy) from model. This allows us to think, plan our actions, make some search trees for possible actions without actually making any action. Here, model learning is like building an MDP and planning is solving this MDP.

8.2 Model-Based Reinforcement Learning

The goal is to estimate the model M from experience $S, A, R...S$, which is a supervised learning problem of form:

1. $s, a \to r$ is a Regression problem,
2. $s, a \to s'$ is a density estimation problem (we want to find the best distribution of possible states after performing action a from state s.

Here the problem is to find an appropriate loss function: in case of Regression that might be MSE, for density estimation: KL divergence. Example Models:

1. Table Lookup Model
2. Linear Expectation Model
3. Linear Gaussian Model
4. Gaussian Process Model
5. Deep Belief Network Model

Table Lookup Model - count the number of times we end up in a particular state and compute an average reward for given state and action.

8.3 Integrated Architectures

Integrating Learning and Planning
We can integrate Model-free and Model-based learning into Dyna - where we learn a model from the real experience and we learn and plan a value function (and/or policy) from both real and simulated experience. The simplest version of Dyna is a Dyna-Q Algorithm. Another variation, Dyna-Q+ encourages exploration more.

8.4 Simulation-Based Search

Forward Search algorithm selects best action based on look-ahead (search tree), which does not require solving whole MDP but the sub-MDP starting from now (we do not care about the states that are not reachable from now). Simulation-Based Search is a paradigm using Forward Search.
Monte Carlo Search simulates K episodes from current (real) state s_t and evaluates all the possible actions that we might take from this state by a mean return (Monte Carlo Evaluation). Monte Carlo Tree Search is the state of the art search method. Here, the policy π improves with time. It essentially is a Monte Carlo control applied to simulated experience (from now on).
Temporal Difference Search is applying SARSA to simulated experience (from now on).
Dyna-2 stores two sets of feature weights: Long-term memory (updated from real experience) and Short-term memory (updated from simulated experience).

9 Exploration and Exploitation

9.1 Introduction

There are roughly tree families of approaches to Exploration vs. Exploitation trade-off. These are:

1. Random exploration, e.g. ϵ-Greedy , Softmax, Gaussian noise.

2. Optimism in the face of uncertainty: estimate the uncertainty of the value and prefer to explore states/actions with highest uncertainty, e.g. Optimistic initialization, UCB, Thompson sampling.

3. Information state search (most computationally difficult): consider agent's information as a part of its state (i.e. ”my state of being here and knowing what is behind that door is different than me being here and not knowing that”), e.g. Gittins indices, Bayes-adaptive MDPs.

9.2 Multi-Armed Bandits

Multi-Armed Bandit is a tuple $< A, R >$, where the goal is to maximize a total cumulative reward. Regret is the difference between the optimal value $V*$ and our action value $Q(a)$. The possible approaches are:

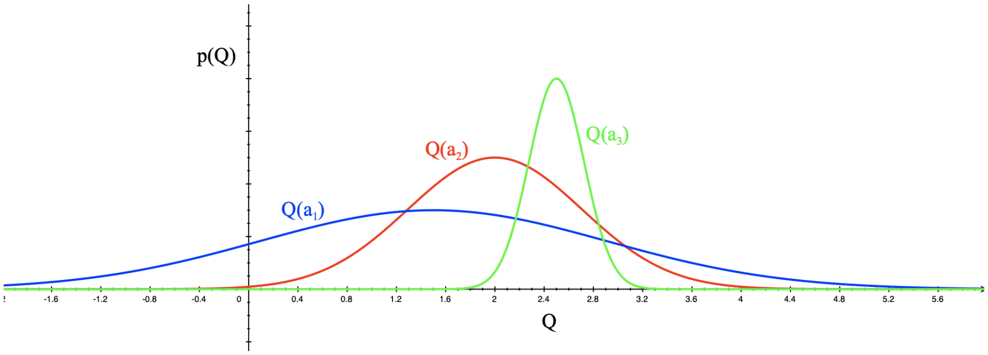
1. Greedy algorithm (might be stuck in a suboptimal action forever and has a linear total regret - remember regret is an expectation).

2. Optimistic Initialization - initialize values to the maximum reward (if known) and then act greedily.

3. ϵ-Greedy

4. Decaying ϵ-Greedy - the value of ϵ decays in time - this gives a sublinear (logarithmic) asymptotic total regret.

5. Upper Confidence Bound (UCB) - select the action which has the highest UCB value of return (not expected return!), e.g. UCB1 Algorithm



The Theorem (Lai and Robbins):
Total Regret is at least logarithmic in the number of steps.

9.3 Bayesian Bandits

Bayesian Bandits exploit prior knowledge about the distribution $p(Q|w)$ over the action-value function. Then we do the same thing as in UCB, selecting an action for which the $\mu + n\sigma$ of the reward distribution is the biggest (we choose n).

9.4 Probability Matching

Instead of using UCB, we can select an action according to the probability that this action is an optimal action. This again encourages selection of actions that have heavy tails, cause they have the highest possibility of being the best actions. For this purpose we use **Thompson sampling** - we sample (just one sample) from each of the distributions $Q(a_1), Q(a_2)$... and select an action for which the sample has the highest Q value. Thompson sampling is asymptotically optimal. Also it does not have a free parameter n (number of standard deviations) to tune.

Information state search is the third family of algorithms dealing with Exploration vs. Exploitation trade-off (apart from Random exploration and Optimism in the face of uncertainty). The previous two approaches were just heuristics. Information state space takes different approach by building an augmented MDP out of the bandit problem. It now keeps an information about the state (i.e. how many times did I choose which action). Here we can build a tree search of all possible actions and choose the best one - solving for this MDP is an optimal way for the Exploration vs. Exploitation trade-off (not a heuristic anymore).

Bayes-adaptive RL is when we characterize the information via the posterior distribution.

9.5 Contextual Bandits

Contextual Bandit is a Multi-Armed bandit with a space, i.e. a tuple $< A, S, R >$. The example are advertisements on the web - which ads should be displayed and which should not? The state here provides a contextual information, i.e. information about the user visiting website.

9.6 MDPs

All of the ideas (e.g. UCB) extend to MDPs as well, not only Bandits.

One successful approach to Exploration vs. Exploitation in Model-based RL is R-max algorithm: *R-max – A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning, R. Braffman, M. Tennenholz, 2012*

10 Classic Games

10.1 Game Theory

Nash equilibrium is a joint policy for all players such that every player's policy is a best response (Best response is the optimal policy against every other player's policies if they fixed their policy, e.g. keep choosing paper in rock-paper-scissors), i.e. no player would choose to deviate from Nash. Nash Equilibrium is a fixed-point of self-play RL, i.e. when we control two agents playing against each other and their policies converge towards a fixed point, that must be a Nash equilibrium.

10.2 Minimax Search

Minimax value function $v*$ maximises white's expected return while minimising black's expected return. Minimax policy is a joint policy $< \pi^1, \pi^2 >$ that achieves a minimax value. Assuming two-player game, zero-sum game, perfect information: there is a unique solution and a minimax policy is a Nash Equilibrium. Deep Blue was an algorithm that defeated Garri Kasparov in 1997. It used minimax search (alpha-beta search a.k.a. $\alpha\beta$).

10.3 Self-play Reinforcement Learning

Apply value-based RL algorithms to games of self-play:
MC: update the value function towards the return G_t.
TD(0): update the value function towards successor value $v(s_t + 1)$.
TD(λ): update the value function towards the λ-return G_{t_λ}.

10.4 Combining Reinforcement Learning and Minimax Search

Simple TD: update value towards successor value.
TD Root: update value towards successor search value.
TD Leaf: update search value towards successor search value.

Note: The plus ”+” notation (in slides) means we run a search from this particular leaf onwards.

TreeStrap: update search values towards deeper search values.

One of the most effective variants of Self-play RL is a UTC algorithm, which is a Monte Carlo Tree Search with UCB for exploration/exploitation balance.

Reinforcement Learning in Imperfect-Information Games Smooth UTC Search is a variant of UTC, inspired by game-theoretic Fictitious Play, where agents learn against and respond to opponent's average behaviour (not his current behaviour). Here we pick our action according to the UTC with probability η or the average strategy π_{avg} with probability $1 - \eta$. This converges to the Nash equilibrium.

Program	Input features	Value Fn	RL	Training	Search
Chess *Meep*	Binary *Pieces, pawns, ...*	Linear	TreeStrap	Self-Play / Expert	$\alpha\beta$
Checkers *Chinook*	Binary *Pieces, ...*	Linear	TD leaf	Self-Play	$\alpha\beta$
Othello *Logistello*	Binary *Disc configs*	Linear	MC	Self-Play	$\alpha\beta$
Backgammon *TD Gammon*	Binary *Num checkers*	Neural network	TD(λ)	Self-Play	$\alpha\beta$ / MC
Go *MoGo*	Binary *Stone patterns*	Linear	TD	Self-Play	MCTS
Scrabble *Maven*	Binary *Letters on rack*	Linear	MC	Self-Play	MC search
Limit Hold'em *SmooCT*	Binary *Card abstraction*	Linear	MCTS	Self-Play	-

11 Side Notes, Terminology

11.1 Policy

Policy π is the probability distribution over actions given states. E.g. Random policy, greedy policy (action with the highest Q-value has $\pi(a) = 1$), ϵ-greedy.

11.2 On-policy vs. Off-policy: SARSA vs. Q-learning

The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s_{t+1} and the **greedy action** a. In other words, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma max_a(Q(s_{t+1}, a)) - Q(s_t, a_t))$$

The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state s_{t+1} and the **current policy**'s action a_{t+1}. It estimates the return for state-action pairs assuming the current policy continues to be followed:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

The distinction disappears if the current policy is a greedy policy. However, such an agent would not be good since it never explores.

Link to discussion

11.3 Prediction vs. Control

Prediction - estimate the value of a particular state or state/action pair, given an environment and a policy.

Control - find the best policy given an environment.

11.4 Online vs. Offline

Online learning algorithms work with data as it is made available. Strictly online algorithms improve incrementally from each piece of new data as it arrives, then discard that data and do not use it again. It is not a requirement, but it is commonly desirable for an online algorithm to forget older examples over time, so that it can adapt to non-stationary populations. Stochastic gradient descent with back-propagation - as used in neural networks - is an example.

Offline learning algorithms work with data in bulk, from a dataset. Strictly offline learning algorithms need to be re-run from scratch in order to learn from changed data. Support vector machines and random forests are strictly offline algorithms (although researchers have constructed online variants of them).

11.5 Policy iteration vs. Value iteration

In **policy iteration** algorithms, you start with a random policy, then find the value function of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on. In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Given a policy, its value function can be obtained using the Bellman operator.

In **value iteration**, you start with a random value function and then find a new (improved) value function in an iterative process, until reaching the optimal value function. Notice that you can derive easily the optimal policy from the optimal value function. This process is based on the optimality Bellman operator.

Link to discussion

11.6 Monte Carlo vs. Temporal Difference Learning

MC is about sampling (S, A, R) and assigning a mean return (over many episodes) as a value for given initial state. It uses G_t.

TD learns from incomplete episodes (by bootstrapping) so it does not need to complete the whole trajectory (until the termination state is achieved), but the partial trajectory and then makes an estimate of the remaining reward. It uses an estimated return $R_{t+1} + \gamma v_{s_{t+1}}$

11.7 Action-value function vs. Value function vs. Reward

R_t is an immediate reward
G_t is a discounted reward, i.e.

$$G_t = R_{t+1} + \gamma R_{t+2} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$V_\pi(s)$ is a state-value function in an MDP. It's an expected return starting from state s and following policy π:

$$V_\pi(s) = E_\pi[G_t|s_t = s]$$

$Q_\pi(s, a)$ is an action-value function. It's an expected return starting from state s, following policy π, taking action a

$$Q_\pi(s, a) = E_\pi[G_t|s_t = s, a_t = a]$$

If you sum/integrate all the action-values by their probabilities (policy) you'll get a state-value function:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) * Q_\pi(s, a)$$