Brief reference notes for the Deep RL workshop

Harshad Khadilkar, Spring 2024

February 6, 2024

Abstract

These notes accompany a workshop series of lectures on Deep RL, hosted in the CSE department, IIT Bombay. Even though most of the papers come from the last decade, they are still not the absolute cutting edge of work today. Instead, the goal of this workshop is to enable students to connect the theory of RL to the practice of Deep RL (including understanding of code implementations), so that they can follow the latest literature in this fast-growing area.

1 Connection to tabular RL

Consider a Markov Decision Process (MDP) with a state space S, action space A, and a transition probability distribution $P : S \times A \to S$. For every transition $(s_t, a_t) \to s_{t+1}$, there is an associated reward r_t. Then the goal of Reinforcement Learning (RL) is to maximise the expected return,

$$G_t = \mathbb{E}_\pi \left[r_t + \gamma \, r_{t+1} + \ldots + \gamma^{T-t-1}(r_T + R_\tau) \right], \tag{1}$$

where $\gamma \in [0, 1]$ is a discount factor and π is a policy mapping $S \to A$. The final term R_τ is called the terminal reward, and is applicable in the case of episodic MDPs.

The standard approach for defining π is to first compute an approximation $Q(s, a)$ for (1) conditional on the current state $s_t = s$ and action $a_t = a$, following which the optimal policy is simply given by,

$$\pi^* = \max_a Q(s, a). \tag{2}$$

In tabular RL, the function Q is implemented by a look-up table containing states as rows and actions as columns, so that any particular $Q(s, a)$ is simply the entry at index (s, a) in the table. All convergence results in RL (including for Q-learning) are based on this approach, assuming infinite number of visits to every pair (s, a) in the problem [WD92].

As the reader will recall, various special types of problems can be derived from the general form of (1) and (2). If there is no concept of the state of the system and the return only depends on actions, we have the standard bandit problem. When the return does depend on the state but there are no transition dynamics, we have the contextual bandit problem. The most general case is the sequential decision-making problem, where Q depends on states and actions, and the actions have a causal effect on the next states. We will mostly be interested in the last (most general) version of the problem.

The obvious practical difficulties of using tabular look-up for Q are,

1. The size of the table in memory as state and action spaces grow,
2. Inability to handle continuous states or actions, and
3. The intractability of visiting each (s, a) pair an infinite number of times.

To handle these challenges, the usual solution is to use a function approximation for $Q(s, a)$. This approach is valid whenever the behaviour of Q is sufficiently smooth (informally speaking). Many versions of function approximation are available in literature, from linear models to specialised models based on the problem type. The concept itself has been used for a long time under the name of approximate dynamic programming [Pow07]. In recent times, the expressivity of neural networks has made this approach very popular, and the use of ANNs for function approximation in RL is called Deep RL. Until we come to Section 7, we will focus exclusively on model-free RL.

2 Simple implementations of value-based methods in Deep RL

Before we get into the implementation discussion, we need to revise the simplification of the approximation problem for (1) as given by the Bellman equation [Bel54]. First derived for dynamic programming, the simplification is due simply to the observation that there is a recursive relationship between G_t and future returns. Specifically, note that,

$$G_t = \mathbb{E}[r_t] + \gamma\, G_{t+1}.$$

This gives us a clue for simplifying the estimation of $Q(s_t, a_t)$ in a similar way. If we wait to observe the reward r_t for the current time step, we can immediately update the estimate using the relationship,

$$Q(s_t, a_t) \approx r_t + \gamma Q_{a_{t+1}\sim\pi}(s_{t+1}, a_{t+1}). \tag{3}$$

Note that (3) does not contain expectations over r_t and s_{t+1} because we are observing these values after one time step. The equation is called one-step Q-learning [Ros14] and is covered in any RL theory course. We will not be covering this material, or its extensions to TD methods, in this manuscript.

For our purposes, it is sufficient to note that if we choose π in (3) as per (2), then the "Deep Q Network" [MKS$^+$15] update relationship becomes,

$$Q(s_t, a_t) \approx r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}). \tag{4}$$

When using Deep RL, the function Q is implemented by a neural network on both sides of (4). The simplest architectural option is to use a concatenation of s and a as inputs, and have a scalar regression output from the network corresponding to $Q(s, a)$. In the *experience collection* phase, we initialise the environment to some initial state s_0 and implement some policy π on the left-hand side to choose the action a_t, and wait for one time step to observe r_t and s_{t+1}. We then store the tuple (s_t, a_t, s_{t+1}, r_t) into a *memory buffer*. After a sufficient number of tuples have been collected (sometimes over multiple episodes of the MDP), we create the training data set for the neural network in the following way.

We first choose a subset of tuples from the memory buffer to generate the mini-batch for our training pass. For each of these tuples, we generate predictions $Q(s_t, a_t)$ which are the equivalent of *predictions* $\hat{y}$ in the supervised setting. Using the corresponding r_t and s_{t+1} in each of the tuples, we form the *targets* $[r_t + Q(s_{t+1}, a_{t+1})]$. These are the equivalent of the labels y in the supervised setting. Then we define a loss function over the error between predictions and targets (for example the mean squared error) and perform an optimisation step for the neural network parameters. The whole procedure is repeated as many times as required.

A common improvement for computational efficiency in this procedure is based on the input-output structure of the neural network. As it stands in (4), the maximisation over a_{t+1} requires the scalar output $Q(s_{t+1}, a_{t+1})$ to be generated $|A|$ number of times for each training sample. To avoid this, practical implementations of DQN use only the state s as input to the neural network, and have $|A|$ outputs corresponding to each possible action. In the forward pass, the maximisation step becomes just the $\arg\min$ over all outputs of the network. During training, the regression error for all actions apart from the implemented one is given as 0. The regression error for the implemented action is the error between the predicted and target values as described earlier.

The procedure described in this section is the default go-to option for starting with Deep RL, and is extremely powerful. However for complex problems, it faces the following challenges:

1. Bootstrapping: Since Q appears on both sides of the equation and is produced by the same network, it is difficult to ensure good convergence of the approximation.

2. Stabilisation: Closely related to bootstrapping is the frequent instability of the Q estimate, especially in the case of noisy rewards r_t. This can cause the estimates to grow unbounded as training progresses.

3. In the implementation with $|A|$ outputs from the network, a typically fully-connected architecture can only discriminate between the actions in the final layer of the network. All latent representations until the penultimate layer are common to all actions.

4. It is not easy to decide how to pick tuples from the memory, and to estimate the validity of samples derived from previous versions of π (several steps earlier in training).

All these challenges are addressed in subsequent sections.

3 Stabilisation of value-based Deep RL

From this section onwards, the manuscript will only link the logical ideas tersely, and refer to the relevant papers for additional details. The cited papers should be referred to for a complete understanding of the discussion. The description in Section 2 points to one basic challenge in RL as opposed to supervised learning: the fact that training data points (the memory buffer) are collected in a correlated manner by interacting with the MDP or environment. As a result, we do not have the luxury of drawing i.i.d. minibatches for training. The use of a memory buffer allows us to partially de-correlate the samples by drawing them in random order from multiple episodes; however this is not a complete fix. A number of approaches have been developed to solve this issue, and some of them are described below. Note that the literature on RL frequently refers to the memory buffer as *experience replay*.

One of the earliest fixes for managing the draws from memory was the idea of prioritized experience replay (PER) [SQAS15]. The intuition behind PER is to preferentially draw those samples that have higher expected learning potential, and in this case the potential is measured by the TD-error in prediction[1]. The TD-error is a proxy for the level of unexpectedness in the transition, either in the next state or in the generated step reward. If the priority of a sample is defined to be p_i^α, the sample is drawn with a probability,

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}.$$

In the original PER, the priority is simply equal to the TD-error, but with a small difference: instead of the target values (see Section 2) being drawn from the same network as the action values, we use a separate target network (which is a slower-update copy of the online network).

Apart from the issue of finding the right samples from the memory buffer, a fundamental issue with RL is *generating* useful samples in the first place. Unlike with supervised learning, we do not have an externally provided data set. The training data are generated by the algorithm itself, and in hard tasks we may never find samples which actually complete the task in its entirety. Hindsight experience replay (HER) [AWR$^+$17] is one approach to fixing this issue. In essence, it defines whatever state the agent reaches at the end of an episode as a 'task', and asks the agent to at least learn to get there consistently if asked. In addition to the state s given as input, we must also include the goal state g in each forward run. The agent gets an auxiliary reward for reaching g, in addition to the actual environment reward.

Finally, a number of such improvements over two years were combined together to form an algorithm called Rainbow [HMVH$^+$18]. The paper does not propose any new modifications to RL; it engineers several previous methods into a single coherent algorithm. It implements PER, dueling networks (which separately compute state value and action advantage), multi-step learning (in contrast to single-step Q-learning), distributional RL (which predicts a distribution of values rather than a single expected one), and noisy nets (where exploration is injected into the value predictions directly) into a single algorithm. Rainbow was a landmark paper because it was the first to handsomely beat median human performance on all 57 Atari games.

4 The exploration-exploitation dilemma

While sampling methods as described in Section 3 help with the stability of learning and approaches such as hindsight replay help with collecting good samples, there is a need to examine the exploration-exploitation tradeoff more systematically. The particular difficulty that arises is in the case of long sequential tasks, where typical $\epsilon-$ greedy strategies fail to find any useful trajectories with random action sampling. This leads to uneven exploration of the state space and a lack of diversity in the memory buffer, especially in tasks that begin from the same initial state (e.g. chess). The challenge is even tougher when the rewards are sparse, for example a terminal reward of 1 for completing the task and 0 for everything else.

There are a very large number of papers on improving exploration efficiency, and we cannot cover everything here. If the reader is interested, the following keywords are good starting points for a search: *directed exploration, frontier states, curiosity, novelty, option critics*. The common threads among these approaches take one or both of two paths: (i) reward shaping to provide 'intrinsic motivation'

[1]Unlike the implementation of DQN where predictions of Q are made after samples are drawn, these TD-errors must be stored at the time of experience collection.

for exploring fresh parts of the state space, and (ii) following predefined sequences of actions rather than drawing actions randomly in each time step.

Among the first set of papers, one of the easiest (and earliest) methods is Model-Based Interval Estimation with Exploratory Bonus (MBIE-EB) [SL08]. Although this is a mouthful to remember, the concept itself is very simple. The authors propose to solve an augmented version of the Bellman equation that increases the reward for new (unseen) states. Adapted to our notation, it can be written as,

$$Q^*(s_t, \cdot) = \max_{a_t} \left[\mathbb{E}[r_t(s_t, a_t)] + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) + \beta N(s_t, a_t)^{-1/2} \right]. \tag{5}$$

Only the last term is added to the regular Bellman update (combined with the greedy policy), with β being a user-defined weighting parameter and N being the number of times a given state-action pair has been observed. This additional reward (effectively an increase to the step reward) encourages the agent to find new *frontiers* of the state space. Several generalisations of this concept to continuous state-action spaces exist, such as [BSO+16].

One of the drawbacks of only emphasising new state exploration is the fact that one may discount previous visited promising states just because they have been seen before. A pair of papers called Never Give Up [BSV+20] and Agent 57 [BPK+20] address this problem by defining two forms of novelty: *episodic* (visited within the same episode) and *lifelong* (visited during the training lifetime). Never Give Up reshapes the step reward using the relation,

$$r_t = r_t^e + \beta r_t^i, \tag{6}$$

where the first term is the original environment reward and the second term is an additional intrinsic reward. The authors also propose a Universal Value Function Approximator (UVFA) which is effectively a network that predicts $Q(s_t, a_t, \beta)$ for any given value of β, thus simultaneously learning a range of exploration policies. We will connect the UVFA to the concept of General Value Functions in the next paragraph. Meanwhile, Agent 57 performs a set of minor tweaks to Never Give Up, resulting in an agent that beats the human benchmark (not the median performance like Rainbow) on all 57 Atari games. The key improvement is really the separation of prediction of r_t^e and r_t^i in (6), giving more stable estimates of r_t (since the policy need not be parameterised by β any more).

Among the second set of literature, exploration efficiency is improved by following a predefined sequence of actions. The concept of temporally extended actions was introduced by [SPS99], who called these primitives as *options*. Examples of options are a predefined set of actions corresponding to picking up an object, travelling to a distant city before engaging in low-level street search, etc. The idea is to retain the learning regime at the higher level of primitives, while the lower-level actions are already known to the agent. Among the early extensions of this concept was the option-critic architecture [BHP17] (an extension of the actor-critic architecture, which we will cover in Section 5). However at this stage, the low-level action primitives must be defined by hand.

EZ-Greedy [DOB20] provided a simple workaround to this issue, by holding a randomly chosen action for a certain number of times (known as the persistence Z). This is usually too simplistic for complex environments. A more sophisticated alternative is to use General Value Functions (GVFs) [SMD+11], which employ Q-learning to predict not the reward but some other useful properties of the environment. We adapted GVFs to provide actions that maximise a particular GVF for a persistence value of time steps, effectively exploring based on a chosen rather than the primary Q-network during the initial stages of training [KSN+22]. Later on, we were able to show the usefulness of this concept for the practical problem of inventory management [KSK23].

The outstanding challenge that is still open at this point, is the handling of very large (or continuous) action spaces. These problems are not easily handled by value-based methods, because the number of action outputs is necessarily finite and fixed. We can use policy gradients as a solution to this problem, which we will now cover.

5 Policy gradient and actor-critic methods

Up to this point, we have focused exclusively on algorithms that approximate $Q(s, a)$ explicitly. The optimal policy π^* is defined subsequently, according to (2). Since this is intractable as the action space grows in size (or becomes continuous), we turn to policy gradient based algorithms. These algorithms have the same optimisation objective (that of maximising G_t), but they parameterise the policy directly as $\pi(a|s;\theta)$. The concept was first introduced as REINFORCE [Wil92], but was not easily scalable until the arrival of deep learning. The idea is to update policy parameters θ (the neural network weights) in the positive direction of $\nabla_\theta \log \pi(a_t|s_t;\theta)G_t$. See the policy gradients chapter of [SB18], but the intuition is given by the following expansion of the gradient:

$$\nabla_\theta \log \pi(a_t|s_t;\theta)G_t = \frac{\nabla_\theta \pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta)}\, G_t$$

The probability of an action a_t is increased if the gradient has the same sign as the return, and is further normalised by the prior probability of the action. Practical implementations of the vanilla version of REINFORCE have a high variance in cases where the G_t for different actions are difficult to distinguish from each other. The standard fix for this issue is to subtract a baseline, resulting in an update in the direction of $\nabla_\theta \log \pi(a_t|s_t;\theta)\,(G_t - b_t(s_t))$. Furthermore, the baseline value b_t is usually given by the state-value function $V^\pi(s_t)$, resulting in the famous advantage actor critic algorithm (A2C) [MBM+16]. The advantage in this case is the excess value of a particular action a_t compared to the average value $V(s_t)$ of the state. Hence,

$$A(a_t, s_t) = Q(a_t, s_t) - V(s_t).$$

The practical implementation of this method will thus involve two neural networks: one to parameterise the policy (let us call this θ^π) and another to parameterise the value function. The Deep Deterministic Policy Gradient (DDPG) [LHP+15] algorithm provides a variant where the value function is the action-value function (not the state-value function), which we shall denote by θ^Q. Based on first principles, the direction of update of θ^π should be given by the gradient of G_t, which is approximated by $Q(s,a)$. Hence we have,

$$\nabla_{\theta^\pi} J \approx \mathbb{E}\left[\nabla_{\theta^\pi} Q(s,a|\theta^Q)|_{s=s_t,a=\pi(s_t|\theta^\pi)}\right].$$

In cases where the action is deterministic (hence "deterministic policy"), we can apply the chain rule to get a simultaneous gradient for the critic and the actor:

$$\nabla_{\theta^\pi} J \approx \mathbb{E}\left[\underbrace{\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\pi(s_t)}}_{\text{critic}}\underbrace{\nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{s=s_t}}_{\text{actor}}\right]. \tag{7}$$

In cases where the actions are continuous, we obviously cannot apply an $\epsilon-$greedy policy which depends on categorical actions. The usual approach is to introduce a Gaussian noise $\mathcal{N}$ to the actions $\pi(s_t|\theta^\pi)$.

While A2C and DDPG offer a scalable way to implement RL for continuous actions, they do not completely overcome the stability challenges faced by policy gradients. For this, we need to limit the rate at which the actor policy can deviate from its current value. Such methods are covered in the next section.

6 Trust region based methods

The drawback of the actor-critic methods described in Section 5 is that while they work in expectation, they do not provide any guarantees on the improvement of the policy. Addressing this issue was the primary goal of Trust Region Policy Optimization (TRPO) [SLA+15]. A corollary of this guarantee would be the fact that hyperparameter tuning would no longer be necessary[2].

Let us examine where the concept of a trust region comes from. Our usual definition of the return (optimization objective) is,

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$$

[2]As we shall see, the algorithm with guarantees is impractical to implement, and so we return by default to something similar to the usual A2C approach.

Then the improvement in objective function due to $\tilde{\pi}$ over π is given by,

$$\begin{aligned}
J(\tilde{\pi}) - J(\pi) &= \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] - \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] \\
&= \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] - \mathbb{E}_{s_0}[V_\pi(s_0)] \\
&= \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t) - V_\pi(s_0)\right] \text{ since independent} \\
&= \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t \left(r(s_t,a_t) + \gamma\, V_\pi(s_{t+1}) - V_\pi(s_t)\right)\right] \text{ telescoping series} \\
&= \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t,a_t)\right] \\
&= \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s)\, A_\pi(s,a), \qquad (8)
\end{aligned}$$

where $\rho_{\tilde{\pi}}(s)$ is the frequency of observation of state s under the policy $\tilde{\pi}$. We have done away with the summation over time by collecting all the observations of state s into ρ.

Intuition: At first glance, it appears surprising that the difference in the two returns should be a function of the advantage of Q over V. But recall that when the value function V is correctly learnt over the final policy π, the advantage reduces to 0 (since the expected value is the value obtained through the optimal action). As a result, the RHS of (8) over the policy π is 0. Whatever improvement is obtained by $\tilde{\pi}$ is a result of the residual advantage of $\tilde{\pi}$ over the value function of π.

Returning to the discussion of TRPO, the observation based on (8) is that the RHS is intractable because we cannot estimate a-priori the visitation frequency $\rho_{\tilde{\pi}}(s)$ for a policy we have not computed yet. Therefore we approximate $\rho_{\tilde{\pi}}(s)$ by the known frequency $\rho_\pi(s)$ (which is why TRPO is an on-policy method). For this approximation to hold good, we need $\tilde{\pi} \approx \pi$, and hence we impose a KL-divergence constraint on the policy update. Cutting out the complex notation, the TRPO formulation for policy update is,

$$\max_\theta \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} Q_{\theta_{\text{old}}}(s,a)\right] \text{ subject to } \mathbb{E}\left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}||\pi_\theta)\right] \leq \delta. \qquad (9)$$

In the form given above, TRPO requires the computation of Hessians (second order optimisation). This is tricky for large dimensional problems, which is nearly always the case with Deep RL. Proximal Policy Optimisation (PPO) [SWD+17] provides an approximate first-order alternative to the more restrictive TRPO. Instead of the constrained optimisation problem (9), we want to solve the unconstrained version obtained by including the constraint into the objective (as a penalty). However, evaluating the KL divergence is computationally hard, so PPO has the following simplification.

PPO proposes to clip the maximum deviation from the old policy, and assumes that this is roughly like constraining the KL divergence. It also replaces the Q value by the advantage A, which has the same optimum since we only subtract a constant term. Tje clipped objective is then given by,

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min\left(\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}}A_t, \text{clip}(\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}}, 1-\epsilon, 1+\epsilon)A_t\right)\right] \qquad (10)$$

Implementations of PPO usually add a value function loss and an entropy regulariser to the clip loss. On the other hand, a subsequent method called the Soft Actor Critic [HZAL18] incorporates entropy into the policy training itself. The simple concept underlying SAC is that instead of adding noise to the action post-facto (as we do with DDPG) or the entropy in the loss term, we should incorporate randomness in the reward itself. The goal is to append an entropy term to the step reward, resulting in the following optimisation objective:

$$J(\pi) = \sum_0^T \mathbb{E}_{s_t,a_t}\left[r_t(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right]. \qquad (11)$$

After some fairly involved derivations, they end up with an algorithm that looks very similar to the usual PPO approach. Instead of two networks (actor and critic), they implement three neural networks (value, action-value, and policy). However the losses for the two value networks are effectively the typical MSE values, and the policy network is updated by a modified policy gradient. The results in the SAC paper are impressive, but the method somehow did not become as popular as PPO.

7 Model-based methods and Neural MCTS

Every method that we have studied up to this point assumes no prior knowledge of the environment's behaviour. This is a powerful assumption, but in some cases it is possible to partially or completely relax it. The sorts of environments where this is possible are,

- Games with deterministic rules but stochastic opponent behaviour, such as chess or go, where it is possible to *roll out* the game to one or more steps in the future [SHM+16, SSS+17],
- Systems where dynamics can be modelled – at least to some extent – using physical principles[3], such as robotics [PN17], and
- Environments where enough data points have been collected to build an approximate and empirical model of the behaviour [MGFL20, HLBN19].

Since model-based RL (abbr. to MBRL from now on) is very close to Model Predictive Control (MPC) [ML99], let us begin the discussion from that point. The typical control problem in dynamical systems aims to minimise the following objective function,

$$J = \min_{u_i} \sum_{i=1}^{\infty} \left[w_x (r_i - x_i)^2 + w_u \, u_i^2 \right] \tag{12}$$

$$\text{subject to } x_{i+1} = f(x_i, u_i) \text{ and } u_i \in \mathcal{U}.$$

Here x_i is the state of the system at discrete time points[4], and u_i are the control inputs or actions. The w's are user-specified weights, and r_i define the ideal or *reference* trajectory. In the case of MPC, f is assumed to be specified in the problem. The mathematical challenge is in solving the sequential optimisation problem exactly, and so MPC typically solves the problem for n steps to get $(u_1, u_2, \ldots, u_n)$. We implement only u_1 from this computation, then re-solve the problem as the window rolls forward. Under the usual assumptions about smooth dynamics and sufficient actuator power, this approach has been established to be extremely competitive. The rolling horizon allows MPC to handle stochastic problems as well as ones with approximate models.

How does this translate to the Deep RL context? For the time being, assume that we have a fairly good idea of the one-step transition probability distribution $P(s_{t+1}|s_t, a_t)$. Given the current state s_t, we can sample $a_t \sim \pi(\cdot|s_t)$. In model-free RL, the sampled action is applied to the environment. Instead, in MBRL we roll out the next state s_{t+1} using P. We may choose to do this once or multiple times if P is highly stochastic. At this point we can compute $V(s_{t+1})$ over all the next steps generated by P, and compute the average to give a more 'correct' estimate of $Q(s_t, a_t)$. If we sample multiple values of a_t from the policy, we can choose the optimal action by maximising over the rolled out values rather than instantaneous ones. Furthermore, we can choose to repeat the action sampling and next state estimation for multiple time steps before computing the value estimates. As we describe below, there are myriad variations of this basic concept.

Let us first consider the basic question of where P (the transition model) comes from. Besides the trivial route of specifying it externally, the following methods are available in literature:

- Episodic memory [FTF+19]: In deterministic environments, the agent can retrieve previous experiences which coincide with the current state and proposed action.
- Bayesian estimation [DFA13]: One can model prior and posterior probabilities over the next state using newly generated experiences. This is sample efficient but computationally costly.

[3]RL comes very close to Model Predictive Control in this scenario

[4]If the actual dynamics are continuous, it is assumed that u_i is held constant for one time step. This is an accurate assumptions for most digital controllers.

- Function approximation: This is how the vast majority of MBRL operates, by building a model of the environment through supervised learning of (s_t, a_t, s_{t+1}) tuples.

- Simulation: Where computationally tractable, one may generate next states through simulation of low-level dynamics.

We cannot hope to cover all of MBRL in these notes, so we will focus on a handful of popular studies. One of the earliest papers to use neural networks for model dynamics was by Nagabandi et al [NKFL18]. In their learning regime, they first run episodes with random exploration to generate enough data for model estimation. Then they train a neural network $\hat{f}_\theta$ in a supervised fashion to predict the next state, given the current state and action. Instead of applying RL based improvement to the policy, the basic approach uses $\hat{f}_\theta$ to estimate the effect of multiple candidate actions, and choose the action whose trajectory has the highest reward. The authors improve on this baseline by computing actions using DAGGER [RB10], where the actions are computing using MPC (and in this case, the model for MPC is $\hat{f}_\theta$). Finally, they also show results by using the trained policy as the initial policy for TRPO, following which the regime is model-free RL. The advantage of this multi-step training regime is that the slow improvement of TRPO is accelerated by having a very good initial guess.

In Dreamer [HLBN19], the authors go one step further by actually training the RL policy assuming the learnt model is the ground truth. The idea is to (i) collect some experiences using the current policy, (ii) train the dynamics model in a supervised manner, (iii) perform an RL training run using the updated model, and then go back to (i) and repeat. This is an effective approach when the learnt model has low multi-step error, but it is difficult to use in more stochastic environments.

The first studies to really solve a complex environment using MBRL were AlphaGo [SHM+16] (and its close successor AlphaGo Zero [SSS+17]). These are important papers not so much for their theoretical contributions as for their engineering effort. They are also the papers that made the terms *self-play* and *neural MCTS* highly popular in the RL community. In short, the four stages of AlphaGo are as follows,

1. Train a supervised learning based policy p_σ from a vast store of human games, to predict next opponent moves (with about 57% accuracy).

2. Since p_σ is a deep network and expensive to use for rollouts, train a smaller policy p_π for rapid sampling (with about 24% accuracy).

3. This is the first RL step. Initialise the RL policy p_ρ using p_σ, and use self-play to improve it.

4. Simultaneously train a win probability value function v_θ in all three phases.

AlphaGo Zero follows a roughly similar regime, but it does not start with human games (hence 'Zero'). Both agents have several engineering tweaks for speed and accuracy of rollouts, which we will cover during the lecture. At this point, we have completed all the basic concepts required for working in Deep RL, at least at the single-agent level. In the next two sections, we will cover an eclectic set of more recent work.

8 Approaching RL as a sequence prediction problem

Everyone working in the area of ML/AI is familiar with the revolution introduced by transformer architectures in sequence prediction problems. It has not taken long for the RL community to realise that the sequential decision-making task is also a sequence prediction problem, but with a difference.

Interestingly, what we may consider to be the first significant paper in this area came before the transformer became popular. Upside-down RL [Sch19] proposed that instead of thinking of rewards as a quantity to be regressed, we could use rewards as *inputs*. The idea is to provide the algorithm with a reward in the form of an instruction (or a reference in control terminology), and let it predict an action that achieves the specified reward. This makes it a supervised learning problem.

Two related groups from Berkeley connected this concept with the power of a transformer, simultaneously proposing the decision transformer [CLR+21] and trajectory transformer [JLL21]. The intuition behind both architectures is similar, while the engineering solution is different. Here we briefly describe the decision transformer (DT). If one thinks of the RL trajectory as a sequence $(s_t, a_t, r_t, s_{t+1}, a_{t+1}, \ldots)$, then it should be possible to model using transformers. The basic DT is an offline learning algorithm, which means it processes a previously collected data set. To stabilise learning, it uses the return-to-go ($\sum_t^T r_t$) rather than the step reward r_t. It also assumes that separate encoders for the state, action, and reward spaces have been trained for mapping all three quantities to the same latent space. Once the DT is trained, it can be used online by seeding with a required return (similar to UDRL) and observing the current state to infer the next action. More recent online versions of the DT have also been proposed in literature.

Two applications of transformers in the RL context stand out for their achievement as well as their potential. The first is a Nature paper [FBH+22] on discovering faster matrix multiplication algorithms using RL. The reason this work is distinct is because rather than handling an online problem, the study actually tackles a static optimisation problem and uses RL as a search algorithm. The authors begin with the known result that any matrix product can be written as a 3D tensor, and decomposed into a summation of rank-1 operations (which themselves are tensor outer products). Given a matrix product, we can think of the rank-1 operations as actions, and the number of terms required to form the matrix as the penalty (shorter is better). This makes it an RL problem. Using a transformer to predict the sequence of actions, the authors use a combination of supervised and reinforcement learning regimes to discover faster multiplication rules (fewer rank-1 terms) than the known state-of-the-art results for several $(n \times m)$ problems.

Since the introductions of LLMs and other foundational models with in-context learning, researchers in RL have considered extending the transformer architecture even further. Instead of just predicting (s, a, r) tuples within an episode, we can actually consider the entire training history (over all episodes) as one large sequence. In this case, training an RL agent can itself be done in-context (without updating the weights of the generative model). Algorithm distillation [LWO+22] is one such approach. The authors train the model itself on RL training histories for a diverse set of tasks. This results in model parameters that 'understand' how RL works in general. Note that the model must be a causal transformer, i.e. should not be conditioned on future tokens (unlike in the case of translation tasks). When introduced to a new task, the model can generate data that mimics RL training, including the tradeoff between exploration and exploitation. However, the specifics of the approach and its potential should be taken with a bit of caution. These are very early days for foundational RL models.

9 List of things we could not cover

At this point, we have covered most of the Deep RL basics required to read and understand literature as of the end of 2023. We have not had a chance to even introduce very large portions of cutting-edge work that builds on these fundamentals. In the following list, we will only mention some of the areas that we could not touch. The studies that are cited should be thought of as possible starting points, and are not meant to be exhaustive.

- Multi-Agent RL (MARL): This is probably the most important body of RL literature that we have not covered. MARL [ZYB21] addresses very important practical problems, including cooperative (team of agents such as drone swarms), non-cooperative (multiple individuals, such as vehicles in a city), and adversarial (competing agents, such as auctions) settings. Algorithms in this setting are fundamentally similar to the ones we have covered so far, but include concepts from game theory and distributed optimisation.

- Offline RL: A common challenge in RL settings is the development of a reasonably accurate simulation or environment for training the agent. Frequently, all we have to work with is a previously captured historical data set using some unknown behavioural policy. Offline RL [ASN20] is a set of algorithms specifically designed to learn without interaction with the environment. It should not be confused with off-policy RL, which is online (has live interactions) but also uses out-of-date data points for training.

- Safe RL: Finally, a word about explainability, performance guarantees, and guardrails. Safe RL [GF15] is a class of algorithms with parallels to robust control, where the objective is to balance performance improvement with failure-averse behaviours. These algorithms are especially important when training is to be done on a real system, and not on a simulation.

References

[ASN20] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning*, pages 104–114. PMLR, 2020.

[AWR$^+$17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[Bel54] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[BHP17] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31/1, 2017.

[BPK$^+$20] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *International conference on machine learning*, pages 507–517. PMLR, 2020.

[BSO$^+$16] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[BSV$^+$20] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, et al. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

[CLR$^+$21] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[DFA13] Richard Dearden, Nir Friedman, and David Andre. Model-based bayesian exploration. *arXiv preprint arXiv:1301.6690*, 2013.

[DOB20] Will Dabney, Georg Ostrovski, and André Barreto. Temporally-extended {\epsilon}-greedy exploration. *arXiv preprint arXiv:2006.01782*, 2020.

[FBH$^+$22] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[FTF$^+$19] Meire Fortunato, Melissa Tan, Ryan Faulkner, Steven Hansen, Adrià Puigdomènech Badia, Gavin Buttimore, Charles Deck, Joel Z Leibo, and Charles Blundell. Generalization of reinforcement learners with working and episodic memory. *Advances in neural information processing systems*, 32, 2019.

[GF15] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[HLBN19] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[HMVH$^+$18] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32/1, 2018.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[JLL21] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34:1273–1286, 2021.

[KSK23] Durgesh Kalwar, Omkar Shelke, and Harshad Khadilkar. Using general value functions to learn domain-backed inventory management policies. *arXiv preprint arXiv:2311.02125*, 2023.

[KSN$^+$22] Durgesh Kalwar, Omkar Shelke, Somjit Nath, Hardik Meisheri, and Harshad Khadilkar. Follow your nose: Using general value functions for directed exploration in reinforcement learning. *arXiv preprint arXiv:2203.00874*, 2022.

[LHP$^+$15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[LWO$^+$22] Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, et al. In-context reinforcement learning with algorithm distillation. *arXiv preprint arXiv:2210.14215*, 2022.

[MBM$^+$16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[MGFL20] Russell Mendonca, Xinyang Geng, Chelsea Finn, and Sergey Levine. Meta-reinforcement learning robust to distributional shift via model identification and experience relabeling. *arXiv preprint arXiv:2006.07178*, 2020.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[ML99] Manfred Morari and Jay H Lee. Model predictive control: past, present and future. *Computers & chemical engineering*, 23(4-5):667–682, 1999.

[NKFL18] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[PN17] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[Pow07] Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

[RB10] Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668. JMLR Workshop and Conference Proceedings, 2010.

[Ros14] Sheldon M Ross. *Introduction to stochastic dynamic programming.* Academic press, 2014.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[Sch19] Juergen Schmidhuber. Reinforcement learning upside down: Don't predict rewards–just map them to actions. *arXiv preprint arXiv:1912.02875*, 2019.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SL08] Alexander L Strehl and Michael L Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331, 2008.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[SMD+11] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

[SPS99] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[ZYB21] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pages 321–384, 2021.