CHAPTER 18

Reinforcement Learning

Chapter 18 introduces **reinforcement learning (RL)** and deep RL, explaining how agents learn by trial and error to maximize cumulative rewards, and presenting policy gradients, Q-learning/DQN, and modern TF-Agents–based methods.

RL setup: agent, environment, rewards

In RL, an **agent** interacts with an **environment** in discrete time steps: it observes a state (or observation), chooses an action, and receives a scalar **reward**, aiming to learn a **policy** that maximizes expected discounted return over time. This framework models tasks like robotics control, game playing (Atari, Go), trading, and thermostats, where feedback is sparse and delayed rather than labeled for each individual decision. Future rewards are typically discounted by factor $\gamma \in (0,1)$, so the **return** from time t is $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$; larger γ values make the agent care more about long-term outcomes.

OpenAI Gym and CartPole example

To standardize environments, **OpenAI Gym** provides many RL tasks (e.g., CartPole-v1, Atari games), each exposing:

- reset() → initial observation.
- step(action) → (next_obs, reward, done, info).
- action_space, observation_space describing valid actions and observations.

The **CartPole** task asks the agent to balance a pole by moving a cart left/right; the episode ends when the pole falls or a step limit is reached, with reward 1 per time step. A simple handcrafted policy "push left if pole leans left, else right" achieves ~42 steps on average, far from the 200-step maximum.

Policy gradients (REINFORCE) with neural policies

A **policy** can be parameterized as a neural network mapping observations to action probabilities (stochastic policy). The **REINFORCE**-style policy gradient algorithm works by:

1. Running the policy for multiple episodes, recording actions, rewards, and per-step gradients that would increase the log-probability of chosen actions.
2. Computing the **discounted return** for each time step, then **normalizing** returns across episodes to estimate **advantages** (how much better/worse than average each action was).
3. Weighting each stored gradient by its corresponding advantage (positive advantages reinforce actions; negative ones suppress them) and averaging to update policy

parameters via gradient ascent (implemented as gradient descent on a negated objective).

Applied to CartPole with a small dense network (4⟶5⟶1 with sigmoid output), this method learns policies that reliably achieve near-maximal reward (~200 steps), outperforming the hardcoded rule. Policy gradients are conceptually simple but **sample-inefficient**, requiring many episodes to estimate advantages accurately.

Markov decision processes and value-based methods

RL problems are modeled as **Markov decision processes (MDPs)** with states s, actions a, transition dynamics $P(s' \mid s, a)$, reward function $R(s, a)$, and discount γ. Value-based methods learn:

- **State-value** $V^{\pi}(s)$: expected return following policy π from state s.
- **Action-value** $Q^{\pi}(s, a)$: expected return starting from s, taking action a, then following π.

Q-learning iteratively updates a table of Q-values using the Bellman optimality equation; for large or continuous spaces, **Deep Q-Networks (DQN)** use a neural net $Q_{\theta}(s, a)$ instead of a table. Core DQN stabilizing tricks:

- **Experience replay**: store transitions in a replay buffer and train on random minibatches to decorrelate updates.
- **Target network**: maintain a slowly updated copy Q_{θ^-} to compute TD targets, reducing oscillations.

DeepMind’s DQN with these ideas achieved near-human and then superhuman control in many **Atari 2600** games directly from pixels.

DQN variants and TF-Agents

Several improvements to DQN are described:

- **Double DQN**: separate action selection and evaluation in TD targets to reduce overestimation bias.
- **Dueling DQN**: decompose $Q(s, a) = V(s) + A(s, a)$ (state-value + advantage), improving learning about state quality independent of action.
- **Prioritized experience replay**: sample transitions with probability proportional to TD error magnitude so the agent focuses on more informative experiences.

The chapter introduces **TF-Agents**, a TensorFlow RL library that provides reusable components: environments (including Gym wrappers), policies, replay buffers, agents (DQN, Actor–Critic, SAC, PPO, etc.), and drivers to collect experience. Example: using TF-Agents to train a DQN agent on **Breakout-v4**:

- Wrap the Gym Atari environment (frame preprocessing, stacking, frame skipping).

- Define a Q-network CNN, DQN agent, replay buffer, metrics, and driver.
- Run a training loop where the driver collects experience steps, the agent trains on replayed batches, and metrics are periodically logged.

Actor–Critic and recent advances

Actor–Critic methods combine a policy network (actor) with a value estimator (critic): the actor is updated via policy gradients using low-variance advantage estimates from the critic, improving sample efficiency over vanilla REINFORCE. Important modern algorithms:

- **A3C/A2C**: asynchronous or synchronous Advantage Actor–Critic with multiple parallel workers exploring copies of the environment, stabilizing and speeding training.
- **Soft Actor–Critic (SAC)**: off-policy Actor–Critic maximizing both expected return and action entropy, encouraging exploration and achieving excellent sample efficiency and robustness; available in TF-Agents.
- **Proximal Policy Optimization (PPO)** and **TRPO**: policy-gradient methods that constrain updates (via clipping or trust regions) to avoid destructive policy changes, widely used in continuous control.

The chapter closes with **curiosity-driven exploration**, where agents receive intrinsic rewards for being surprised by prediction errors on their own learned models, helping in environments with sparse external rewards.