Deep Reinforcement Learning

372.2.5910

Ben-Gurion University of the Negev

Lecture Notes

Contents

1. Hello world

Reinforcement Learning (RL) is the study of agents and how they learn by trial and error. The two main components of RL are the *agent* and the *environment* - The agent interacts with the environment (also known as taking a "step") by seeing a (sometimes partial) *observation* of the environment's *state*, and then decides which *action* should be. The agent also perceives a reward from the environment, which is essentially a number that tells the agent how good the state of the world is, and the agent's goal is to maximize the *cumulative reward*, called *return*.

$$
s_t,\ r_t \quad \begin{matrix} \textit{Agent} \\ \textit{Env} \end{matrix} \quad a_t
$$
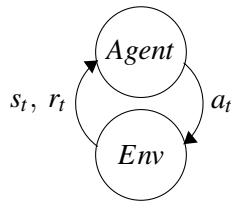
1.1 Terminology

let's introduce some additional terminology

1.1.1 State & Observation

The complete description of the environment/world's state is the *state s*, and an *observation o* is a partial description of *s*. We usually work with an observation, but wrongly denote it as *s* - we are going to stick with this convention.

R if $o = s$ we say that the environment is *fully observed*. Otherwise, it is *partially observed*.

1.1.2 Action spaces

Is the set of all valid actions in the environment. The action space could be discrete (like the action to move in one of the direction $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$) or continuous (like the action to move the motor with $\alpha \in \mathbb{R}$ Newtons of force)

1.1.3 Policy

Is a set of rules used by our agent to decide on the next action. It can be either deterministic of stochastic $a_t \sim \pi(\cdot|s_t)$. Under the scope of deep RL, the policy is a parameterized function, i.e it is a mapping with parameters θ that should be learned in some optimization process. A deterministic Policy could be implemented, for example, using some basic MLP architecture. For a stochastic policy, the two most common types are *Categorical* policy (for discrete action space) and *Diagonal-Gaussian* policy (for continuous action space)

Categorical (stochastic) Policy

Is essentially a classifier, mapping discrete states to discrete actions. For example, you could build a basic NN that takes in the observation and outputs action probabilities (after applying softmax). Denoting the last layer as $P_\theta(s)$, we can treat the actions as indices so the log-likelihood for action a is

$$\log \pi_\theta(a|s) = \log[P_\theta(s)]_a \tag{1.1}$$

Given $P_\theta(s)$, we can also sample from the distribution (one can use PyTorch `Categorical` to sample from a probability vector)

Diagonal-Gaussian (stochastic) Policy

Is a policy that can be implemented using a neural network that maps observations to *mean* actions, under the assumption that the action probability space can be represented by some multivariate Gaussian with diagonal covariance matrix, which can be represented in two ways

- we use $\log diag(\Sigma) = \log \sigma$ which is *not* a function of the state s (σ is a vector of standalone parameters)
- we use a NN that maps from $s \rightarrow \log \sigma_\theta(s)$

we use $\log \sigma$ and not σ as the log takes any value $\in (-\infty, \infty)$, unlike σ that only takes values in $[0, \infty)$, making it harder to train.

Once the mean action $\mu_\theta(s)$ and the std $\sigma_\theta(s)$ are obtain, the action is sampled as $a = \mu_\theta(s) + \sigma_\theta(s) \otimes z$, where $z \sim N(0, 1)$ and $\otimes$ is element-wise multiplication (This is similar to VAEs).

The log-likelihood of a k-dimensional action $a \in \mathbb{R}^k$ for a diagonal-Gaussian with mean μ_θ and std σ_θ can be simplified if we remember that when Σ is diagonal, a k-multivariate Gaussian's PDF is equivalent to the product of k one-dimension Gaussian PDF, hence

$$\log[\pi_\theta(a|s)] = \log \left[\frac{\exp\left[-\frac{1}{2}(a-\mu)^T \Sigma^{-1}(a-\mu)\right]}{\sqrt{(2\pi)^k |\Sigma|}} \right] = ... = -\frac{1}{2}\left[\sum_{i=1}^{k} \left(\frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2\log \sigma_i \right) + k\log 2\pi \right] \tag{1.2}$$

1.1.4 Trajectories

we denote the trajectory as $\tau = (s_0, a_0, s_1, a_1, ...)$ where the first state s_0 is randomly sampled from some start-state distribution $s_0 \sim \rho_0$. A new state is obtained from the previous state and action in either a stochastic or deterministic process.

(R) τ is also noted as "episode" or "rollout"

1.1.5 Reward

The reward r_t is some function of our states and action, and the goal of the agent is to maximize the cumulative reward over some trajectory τ.

- Finite-horizon un-discounted return: $R(\tau) = \sum\limits_{t=0}^{T} r_t$
- Infinite-horizon discounted return: $R(\tau) = \sum\limits_{t=0}^{\infty} \gamma^t r_t$, $\gamma \in (0,1)$

Not adding a converging term γ^t means that our infinite sum may diverge, but also it manifests the concept of "reward now > reward later"

1.1.6 The goal of RL

We always wish to find a policy π^* which maximizes the expected return when the agent acts according to it. Under the assumption of stochastic environment and policy, we can write the probability to obtain some trajectory τ of size T, given a policy π as

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} \underbrace{P(s_{t+1}|s_t,a_t)}_{\substack{\text{Pr. to reach } s_{t+1} \\ \text{from } s_t \text{ when applying } a_t.}} \cdot \underbrace{\pi(a_t|s_t)}_{\substack{\text{Pr. to choose action } a_t \\ \text{when in state } s_t.}} \tag{1.3}$$

The expected return is by definition the sum of returns given all possible trajectories, weighted by their probabilities

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{1.4}$$

and w.r.t to this objective, we wish to find

$$\pi^* = \underset{\pi\in\Pi}{\arg\max}\, J(\pi) \tag{1.5}$$

1.1.7 Value function

We can think of the expected return given some specific state, or some specific state-action pair as the "value" of the state, or the state-action pair. Those are simply the expected return conditioned with some initial state or action

- On-policy Value function $V^\pi(s)$: if you start from s and act according to π, the expected reward is

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

- On-policy Action-Value function $Q^\pi(s)$: if you start from s, take an action a (which may or may not come from π) and only then act according to π, the expected reward is

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

when finding a value function or an action-value function that maximizes the expected reward, we scan various policies and extract $V^*(s) = \max\limits_{\pi \in \Pi} V^\pi(s)$ or $Q^*(s,a) = \max\limits_{\pi \in \Pi} Q^\pi(s,a)$.

We can also find a relation between V and Q:

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s] \\
&= \sum_{\tau \sim \pi} Pr[R(\tau)|s_0 = s]R(\tau) \\
&= \sum_{\tau \sim \pi}\sum_{a \sim \pi} Pr[R(\tau),a|s_0 = s]R(\tau) \text{ [Total prob.]} \\
&= \sum_{a \sim \pi} Pr[a|s_0 = s] \sum_{\tau \sim \pi} Pr[R(\tau)|s_0 = s, a_0 = a]R(\tau) \\
&= \sum_{a \sim \pi} Pr[a|s_0 = s]\mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \\
&= \sum_{a \sim \pi} Pr[a|s_0 = s]Q^\pi(s,a) \\
&= \mathbb{E}_{a \sim \pi}Q^\pi(s,a)
\end{aligned}
\tag{1.6}
$$

where in the 4'th line we used the fact that the probability of both $R(\tau)$ and a is the same as summing over all possible a and conditioning the probability $Pr[R(\tau)]$ given a. In terms of optimality, notice that as $V^*(s)$ is the optimal value function for a specific s, and for any a, and $Q^*(s,a)$ is the optimal value for a specific s and a, taking $\max Q$ over all a is exactly $V(s)$. Specifically

$$
V^*(s) = \max_a Q^*(s,a)
\tag{1.7}
$$

1.1.8   The optimal Q-Function and the optimal action

$Q^*(s,a)$ gives the expected return for starting in s and taking action a, and then acting according to the optimal policy. As the optimal policy will select, when in s, the action that maximizes the expected return for when the initial state is s, we can obtain the optimal action a^* by simply maximizing over all values of Q^*

$$
a^*(s) = \operatorname*{argmax}_a Q^*(s,a)
\tag{1.8}
$$

We also note that if there are many optimal actions, we may choose one randomly

1.1.9   Bellman Equations

An important idea for all value functions is that the value of your starting point is the reward you expected to get from being there + the value of wherever you land next

$$
V^\pi(s) = \mathbb{E}_{a \sim \pi, s' \sim P}\left[r(s,a) + \gamma V^\pi(s')\right]
\tag{1.9}
$$

$$
Q^\pi(s,a) = \mathbb{E}_{s' \sim P}\left[r(s,a) + \gamma\mathbb{E}_{a' \sim \pi}[Q^\pi(s',a')]\right]
\tag{1.10}
$$

and optimality is obtained for

$$
V^*(s) = \max_{a \sim \pi} \mathbb{E}_{s' \sim P}\left[r(s,a) + \gamma V^*(s')\right]
\tag{1.11}
$$

$$
Q^*(s,a) = \mathbb{E}_{s' \sim P}\left[r(s,a) + \gamma\max_{a' \sim \pi}[Q^*(s',a')]\right]
\tag{1.12}
$$

1.1.10 Advantage function

Sometimes we only care if an action is better than others on average, and do not care as much for its' value on its own. The advantage function $A^\pi(s,a)$ describes how better is taking action a (that can be from π or not) when in s compared to selecting some random action $a' \sim \pi$, assuming you act according to π afterwards.

$$
\begin{aligned}
A^\pi(s,a) &= Q^\pi(s,a) - V^\pi(s) \\
&= Q^\pi(s,a) - \mathbb{E}_{a \sim \pi}[Q^\pi(s,a)]
\end{aligned}
\tag{1.13}
$$

By calculating the advantage, we ask if the Q function of some candidate action a (given some state s) is larger then the average Q-function associated with the examination of all other actions taken by our policy.

1.2 Kinds of RL Algorithms

1.2.1 Model-Free vs Model-Based RL

In some cases we have a closed form of how our environment behaves. For example, we may know the probability space $P[s'|s,a]$, i.e we know that probability to transition from some s to some other s' given arbitrary action a. It may also be the case that our our model can be described using some equation of motion. Either way, we can use this knowledge to formulate an optimal solution, which in many cases translate to some greedy approach of scanning various states and thinking ahead.
Some problems are that such model may not even be available (or even known), as for example, I do not have a model of how my chess opponent may play. Furthermore, greedy approaches usually mean brute-forcing all possible solutions.
In Model-Free RL, the model is not available, and we are trying to understand how the environment behaves by exploring it in some non-exhaustive manner. This means that model-free RL are likely to be not sample-efficient, though they are usually easier to implement

1.2.2 What do we learn in RL

After going through the taxonomy, we can ask whether we wish to learn the Q-function, the value-function, the policy or the environment model itself.

Under model-free RL
- Policy optimization: we parameterize the policy $\pi_\theta(a|s)$ and find optimum w.r.t the return $J(\pi_\theta)$. Such optimization is usually *on-policy*, meaning that the data used in the training process is only data given while acting according to the most recent version of the policy. In policy optimization we also find an approximator value function $V_\phi(s) \approx V^\pi(s)$. Some examples are *A2C,A3C,PPO*.
- Q-Learning: approximate $Q_\theta(s,a) \approx Q^*(s,a)$. Usually the objective is some form of the bellman equation. Q-Learning is usually *off-policy*, meaning that we use data from any point during training. Some examples are *DQN, C51*.

Compared to Q-Learning, that approximates Q^*, policy optimization finds exactly what we wish for - how to act optimally in the environment. Also, there are models that combine the two approaches, as *DDPG* for example, which learns both a Q function and an optimal policy.

Under model-based RL

cannot be clustered as easily, though some of the (many) approaches include methods of planning techniques to select actions that are optimal w.r.t to the model.

1.3 Intro to policy optimization

We aim to maximize the expected return $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$, and we assume the finite-horizon undiscounted return (∞-horizon is nearly identical). Our goal is to optimize π_θ with a gradient step

$$\theta_{k+1} = \theta_k + \alpha \underbrace{\nabla_\theta J(\pi_\theta)}_{\text{Policy gradient}} |_{\theta_k} \tag{1.14}$$

and to do so, we must find a numerical expression for the policy gradient. As $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \int_\tau P(\tau|\theta)R(\tau)$, we might as well write down a term for the probability of a trajectory

$$P(\tau|\theta) = \rho_0(s_0)\prod_{t=0}^{T}P(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t) \tag{1.15}$$

Using the log-derivative trick, $\frac{d}{dx}\log x = \frac{1}{x}$, meaning that $x\frac{d\log x}{dx} = 1$. rewrite 1 as $\frac{d}{dx}x$, Substitute $x \leftrightarrow P(\tau|\theta)$ and $\frac{d}{dx} \leftrightarrow \nabla_\theta$ and we have that $P(\tau|\theta)\nabla_\theta\log P(\tau|\theta) = \nabla_\theta P(\tau|\theta)$. We will use this later. Now, lets expand the log term

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T}\left[\log P(s_{t+1}|s_t,a_t) + \log \pi_\theta(a_t|s_t)\right] \tag{1.16}$$

When deriving w.r.t θ, we are only left with the last term (the others only depend on the environment and not our agent), hence

$$\nabla_\theta \log P(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T}\log \pi_\theta(a_t|s_t) = \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) \tag{1.17}$$

Notice the use of linearity in the second transition. Consequently, we re-write the expected return using eq 1.17 -

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\ &= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) \\ &= \mathbb{E}_{\tau \sim \pi_\theta}\left[\nabla_\theta \log P(\tau|\theta)R(\tau)\right] \\ &= \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right]\end{aligned} \tag{1.18}$$

In the 3rd transition we used the log-derivative trick, and in the last transition we used the expression from 1.17.

The last term is an expectation, hence can be estimated using mean - given a collected set $D =$

$\{\tau_1, \tau_2, ..., \tau_N\}$ of trajectories obtained by letting our agent act in the environment using π_θ we can write

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{1.19}$$

(R) It should be stated that this "Loss" term is not really a "loss" like we know from supervised learning. First of all, it does not depend on a fixed data distribution - here, the data is sampled from the recent policy. More importantly, it does not measure performance! the only thing it makes sure of is that given the *current* parameters, it has the negative gradient of performance. After this first step of gradient descent, there is no more connection to performance. This means that the loss minimization has no guarantee to improve expected return. This should come as a warning to when we look at the loss going down thinking that all is well - in policy gradients, this intuition is wrong, and we should only look at the average return.

2. RL basics

2.1 Motivation

Current malware detection platforms often deploy an ensemble of detectors to increase overall performance. This approach creates lots of redundancy, as in most cases one detector is enough, and it of course computationally expensive and time consuming, compared to one detector.

We can come up with a simple improvement - query a subset of detectors and decide based on their classification if more detectors are needed. If we observe our approach under the scope of classification, it may very well be the case that training a model w.r.t to every set of detectors is needed, as we cannot evaluate the performance of a subset of detectors without actually learning how they performed. As this is computationally hard for large detectors, this is not a preferred approach. Instead, we can use RL:

Suppose we use four detectors, and our agent takes as input the vector $[-1, -1, -1, -1] \in \mathbb{R}^4$, which is considered an initial state. The agent will choose a set of detectors/detector configurations, and a classification measurement of either "*malicious*" or "*benign*" will be taken. The decisions of the agent will be based on a reward mechanism that takes uses the values of TP, FP (correctly classified the content as "*malicious*" or "*benign*") and FP, FN (incorrectly classified to "*malicious*" or "*benign*"). We will "punish" using $C(t)$, which is a function that depends on the time it took for the detectors to run. We can see that regardless of how many detectors were used, if we are right - the

Exp. #	TP	TN	FP	FN
1	1	1	-C(t)	-C(t)
2	10	10	-C(t)	-C(t)
3	100	100	-C(t)	-C(t)

Table 2.1: Three suggested reward mechanisms for a malware detection platform

reward is constant (experiment with $1, 10, 100$). On the other hand, if we were wrong, we subtract

$C(t)$ which increases with the time t that has passed. As it is now "painful" to use more detectors, the reward incentives our model to only use more detectors if that addition translated to higher success rates. As our model efficiently scans through the state space, it is able to outperform (at least conceptually) the suggested "check-all" classification approach that was previously introduced.

2.2 When to use RL?

Not all cases adhere to the framework of RL. Here are some rules
- our data should be in the form of trajectories - a set of distinguishable states $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_N$
- need to make a sequence of related decisions - if every decision is independent, like classifying 4 images of cats-vs-dogs, don't use RL.
- the actions we perform result in some feedback - either positive or negative
- Tasks that are more suitable include both learning and planning - we learn our environment and plan an optimal behaviour
- the data is not provided apriori, and its' distribution changes with our action choice. This means that we must make sure our agent effectively explores the entire data, and does not settle in some small subspace.

2.3 Markov Decision Processes (MDP)

consists of the following
- States: that make up the environment. could be either discrete or continuous.
- Actions: by taking an action we transition from one state to another. In a deterministic process, we have that $P(s'|s,a) = 1$
- Reward: taking action $a \in A$ from state $s \in S$ results in reward $R(s,a) \in \mathbb{R}$

In finite MDP, s, a, r are all final.

2.3.1 The Markov Property

the distribution over the future states depends only on the present state and action

$$Pr[s_{t+1}|s_1, a_1, s_2, a_2, ..., s_t, a_t] = Pr[s_{t+1}|s_t, a_t] \tag{2.1}$$

In poker, for example, the markovian property does not hold as a player's current hand depends on the actions and hands of previous hands and/or players. A traffic light, on the other hand, is completely markovian as it is based on deterministic rules.

Using the markovian property, we can define the probability to reach a certain state s' with a certain reward r, as

$$Pr[s', r|s, a] \equiv Pr[s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a] \tag{2.2}$$

The probabilities induced by all event in S and R make up a probability space, hence

$$\forall s \in S \ \forall a \in A: \ \sum_{s' \in S} \sum_{r \in R} Pr[s', r|s, a] = 1 \tag{2.3}$$

The expected reward for state-action pairs, namely, what should we anticipate (in terms of reward) when performing the action a from the state s is

$$r(s, a) \equiv \mathbb{E}[r_{t+1}|s_t = s, a_t = a] = \sum_{r \in R} r \sum_{s' \in S} Pr[s', r|s, a] \tag{2.4}$$

where notice that the probability for a specific reward is the sum over all states, given the specific r (hence the sum over $s' \in S$).

We can also phrase our reward in terms of state-action-next state triplets, namely, what should we anticipate (in terms of reward) when performing the action a that takes us from state s to state s'

$$r(s,a,s') \equiv \mathbb{E}[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'] = \sum_{r \in R} r \frac{Pr[s',r|s,a]}{Pr[s'|s,a]} \tag{2.5}$$

Where we can think of the probability fraction as the number of events that reach s' (from s after performing a) and provide reward r, out of all the event that reach s' (from s after performing a) given any reward.

MDPs are very flexible

- both states and actions could be either abstract (s="*sad*", s="*happy*", a="*take a nap*") or well-defined (like s=sensor readings, or a=turn on a switch).
- the time intervals may not be constant (some transitions are slow while other are fast)
- the setting of an MDP does not need to be an exact copy of the real-world model. For example, a set of sensors may be enough to describe a robotic arm, even though there are many more aspects that the arm is made up of (that are not as relevant).

(R) At this point is may be helpful to look at what is known as the "*Backup Diagram*", That describes how the states are propagated based on the actions chosen by π and the probabilities induces by the environment.
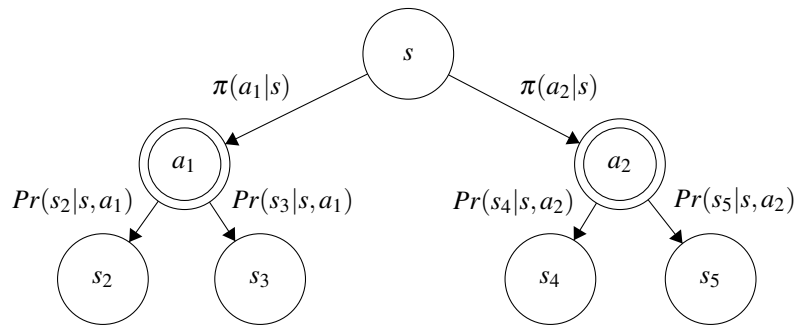
We can write, for example, the probability to transition from s to s_5 by performing an action a_s as $P(s_5|s,a_2) = \pi(a_2|s)Pr(s_5|s,a_2)$. In general term,, the probability to move to any state by performing any action is the probability to take some action a and sum all probabilities of states s' reachable from s using a, and finally sum over all such actions

$$Pr(\text{reach any state using any action}|s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} Pr(s'|s,a) \tag{2.6}$$

Equivalently, we can write the probability to reach any state using any action and receiving any reward

$$Pr(\text{any state any action any reward}|s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} \sum_{r \in R} Pr(s',r|s,a) \tag{2.7}$$

2.4 Goals and Rewards

The agent's goal is to maximize the expected return.
For a finite horizon of size T, the undiscounted sum of rewards is

$$G_t = R_{t+1} + R_{t+2} + ... + R_{t+T} = \sum_{k=0}^{T} R_{t+k+1} \tag{2.8}$$

For an infinite horizon, we add a discount factor, as otherwise infinite sum would result in an agent that does not really care for the reward mechanism. As previously stated, the intuition here is reward now > future reward.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.9}$$

where $\gamma \in (0,1)$

2.5 Policies,Value-function and Q-function

The policy π defines out strategy, namely, what we choose to do at every step

$$\pi(s,a) = Pr[a_t = a | s_t = s] \tag{2.10}$$

The goal of π is to maximize the value function, which is the cumulative expected return of following π starting from some state s

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s \right] \tag{2.11}$$

Notice that the expectation is w.r.t π, meaning that after the initial state $s_t = s$, the next states are fully determined by π. We can think of V_π as a measurement of "*How good is π?*", as intuitively, we can choose the policy that provides us the maximal expected return.
We can also define the Q-function, which is the same as V except the fact that we start from s *and* perform an initial action a (that may or may not be one of π's options)

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a \right] \tag{2.12}$$

2.6 The Bellman equation

> **Theorem 2.6.1** The value function can be written as
>
> $$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t | s_t = s] \\ &= \sum_{a \in A} \pi(a|s) \sum_{s' \in S} \sum_{r \in R} Pr[s',r|s,a][r + \gamma V_\pi(s')] \\ &= \mathbb{E}_{a \in A}[\mathbb{E}_{s',r}[r + \gamma V_\pi(s')]] \end{aligned} \tag{2.13}$$

Proof. [1] The reward and time t can be rewritten as

$$
\begin{aligned}
G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\
&= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\
&= R_{t+1} + \gamma G_{t+1}
\end{aligned}
\tag{2.14}
$$

Therefore the can re-write the Value-function as

$$
\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | s_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} | s_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} | s_t = s]
\end{aligned}
\tag{2.15}
$$

Focusing on the second term, we will use the law of iterated expectation

$$
\mathbb{E}[Y|X = x] = \mathbb{E}[\mathbb{E}[Y|X = x, Z = z]|X = x]
$$

with $Y = G_{t+1}, X = S_t, x = s, Z = S_{t+1}$ and $z = s'$, hence

$$
\begin{aligned}
\mathbb{E}_\pi[G_{t+1}|s_t = s] &= \mathbb{E}[\mathbb{E}[G_{t+1}|S_t = s, S_{t+1} = s']|S_t = s] \\
&= \mathbb{E}[\mathbb{E}[G_{t+1}|S_{t+1} = s']|S_t = s] \\
&= \mathbb{E}[V_\pi(S_{t+1} = s')|S_t = t]
\end{aligned}
\tag{2.16}
$$

In the 2_{nd} transition we removed the inner condition for $S_t = s$ as $G_{t+1} = R_{t+2} + \gamma R_{t+3} + ...$ does not depend on S_t. This is the case as every reward term R_{t+i} is only a function of the current state and action, so since R_t is not present, no term in G_{t+1} is related to S_t (only to S_{t+1}, S_{t+2}...). In the last transition we use the fact that the inner $\mathbb{E}$ term is nothing but the value function for $t \leftarrow t + 1$. Substituting all to 2.15 we have

$$
\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} | s_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} | s_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} | s_t = s] + \gamma \mathbb{E}[V_\pi(S_{t+1} = s')|S_t = t] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1} = s')|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s' \in S} \sum_{r \in R} Pr(s', r|s, a) \left[r(s, a, s') + \gamma V_\pi(S_{t+1} = s') \right]
\end{aligned}
\tag{2.17}
$$

R_{t+1} describes the reward obtained when moving from s to s' using a, so it can be written as $r(s, a, s')$. Furthermore, the expectation $\mathbb{E}_\pi$ is w.r.t to the states, actions and rewards induced by π (so the probability associated with every term is the one introduced in 2.7), and by summing over $r \in R$ we also indicate the fact that the transition $s \to s'$ could be rewarded with multiple different rewards (more than one option is plausible).

∎

The bellman optimality equation is the bellman equation for the optimal Value-function

$$
V^*(s) = \max_a \mathbb{E}[r(s, a) + \gamma V^*(s')]
\tag{2.18}
$$

---

[1] https://stats.stackexchange.com/questions/243384/deriving-bellmans-equation-in-reinforcement-learning

Theorem 2.6.2 The Q-function can be written as

$$Q_\pi(s,a) = \sum_{s',r}[r(s,a) + \gamma\sum_{a'}Q_\pi(s',a')] \tag{2.19}$$

Proof. Not included   ∎

The bellman optimality equation is the bellman equation for the optimal Q-function

$$Q^*(s,a) = \mathbb{E}[r(s,a) + \gamma\max_{a'}Q^*(s',a)] \tag{2.20}$$

2.7 Policy Iteration

Is a method of finding an optimal policy by performing two steps - evaluation and improvement. After a random initialization of both π and V (Step I), we evaluate the Value-function given some policy π (Step II)[2]. We do this by constantly sampling our environment and updating the value function for every state respectively. The loop stops when the change in value function (for all s) is smaller than some tolerance ε. We use the notation $\pi(a|s)$ to indicate the probability $Pr(\pi(s) = a)$.

Next, in step III, we observe the current Value function and choose an action that maximizes it. This will be considered our new returned action for every single state

Finally, we combine policy evaluation and policy improvement in an iterative process. More specifically, we evaluate V and improve π until a fixed point is reached (for all s, $\pi(s)$ has not changed, possibly up to some margin δ)[3].

2.8 Value iteration

In some cases it may not be efficient or even possible to scan all states $s \in S$, but scanning all possible actions $a \in A$, is. In VI, we choose the value of some state s as the maximal V function over all $a \in A$ and when the needed tolerance was reached, our returned policy would be the action that maximizes the final V function

2.9 Monte-Carlo

In cases where the dynamics $Pr[s'|s,a]$ and the reward G_t are unknown (model-free setting, 1.2.1), we can use a Monte-Carlo approach to sample the environment and come up with approximations for the value-function and Q-function. To do so, one must make sure that the episodes are finite (the number of transition until termination is $<$ inf). Another important factor is how shall we behave when encountering the same state more than once, and there are two common variants
  • First-Visit-Monte-Carlo (FVMC): estimates the return obtained only after the first visit to s (ignore future visits to s)
  • Every-Visit-Monte-Carlo (EVMC): estimates the average returns obtained after all visits to s (average all rewards obtained from s in the episode)

---

[2]The convergence of PE is the result of the Policy evaluation convergence theory. see HERE for more info
[3]I highly recommend checking out THIS implementation, by Denny Britz

---

Algorithm 1 Policy Iteration

Require: tolerance $\varepsilon > 0$ and an MDP $\langle S, A, R, Pr : S \times R \to \mathbb{R}, \gamma \rangle$

   $V, \pi \leftarrow$ Random Initialization $\in \mathbb{R}^{|S|}$ $\triangleright$ Step I: Initialization

   while True **do** $\triangleright$ Step II: Policy Evaluation

      $\Delta \leftarrow 0$

      for $s \in S$ **do**

         $v \leftarrow V(s)$

         $V(s) \leftarrow \sum\limits_{a \in A} \pi(a|s) \sum\limits_{s' \in S} \sum\limits_{r \in R} Pr[s', r|s, a][r + \gamma V(s')]$ $\triangleright$ using thrm. 2.6.1

         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

      end for

      if $\Delta < \varepsilon$ **then** break

      end if

   end while

   policy_stable←True $\triangleright$ Step III: Policy Improvement

   for $s \in S$ **do**

      old_$\pi \leftarrow \pi(s)$

      $\pi(s) \leftarrow \underset{a \in A}{\mathrm{argmax}} \sum\limits_{s' \in S} \sum\limits_{r \in R} Pr[s', r|s, a][r + \gamma V(s')]$

      if old_$\pi \neq \pi(s)$ **then** policy_stable←False

      end if

   end for

   if policy_stable **then return** V, π

   else go to step II

   end if

Algorithm 2 Value Iteration

Require: tolerance $\varepsilon > 0$ and an MDP $\langle S, A, R, Pr : S \times R \to \mathbb{R}, \gamma \rangle$

   $V \leftarrow$ Random Initialization $\in \mathbb{R}^{|S|}$

   while True **do**

      $\Delta \leftarrow 0$

      for $s \in S$ **do**

         $v \leftarrow V(s)$

         $V(s) \leftarrow \underset{a \in A}{\max} \sum\limits_{s' \in S} \sum\limits_{r \in R} Pr[s', r|s, a][r + \gamma V(s')]$

         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

      end for

      if $\Delta < \varepsilon$ **then** break

      end if

   end while

   return $\pi(s) = \underset{a \in A}{\mathrm{argmax}} \sum\limits_{s' \in S} \sum\limits_{r \in R} Pr[s', r|s, a][r + \gamma V(s')$ for all $s \in S$

2.9.1 Approximating Value-function

To approximate the value function V_π of a given policy π, we will sample a trajectory and update the cumulative discounted reward given the reward we have received. In the case of FVMC, we save the resulted reward for a given state *iff* we did not encounter it previously. The above is also described in alg. 3

Algorithm 3 First-Visit MC for Value function approximation

Require: a policy π to be evaluated

    $V(s) \leftarrow$ arbitrary initialization $\in \mathbb{R}$ for all $s \in S$

    $Returns(s) \leftarrow$ empty list [] for all $s \in S$

    while True **do**

        $\tau \leftarrow \{s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\}$ ▷ generate an episode using π

        $G \leftarrow 0$

        for step $t = T-1, T-2, ..., 0$ **do**

            $G \leftarrow \gamma G + r_{t+1}$

            if $s_t \notin \{s_0, s_1, ..., s_{t-1}\}$ **then**   ▷ Verifying first-visit

                $Returns(s_t).append(G)$

                $V(s_t) \leftarrow avg(Returns(s_t))$

            end if

        end for

    end while

2.9.2 Approximating policies

When a model of the world is available, we have already seen (in Policy Iteration, for example) that given the state-values only, one could generate a policy. On the other hand, when the model is not available - the state values alone does not contain enough information to formulate a policy, and one must estimate the action values in order to come up with a good policy. This means that approximating the Q-function could be useful to determine a policy π.

The formalism of FVMC for Q-function approximation is almost identical to alg. 3, except the fact that in a First-Visit, we make sure that both the state s and the action a were not encountered yet in the trajectory. It is also important to understand that if π is deterministic, following it we only observe returns for one specific action from each state. This means that there are no returns to average, hence we in fact do not allow our estimate to explore the state-action space properly. To solve this we must enforce exploration by, for example, setting a random state-action starting point for every episode.

To understand how a policy can be generated, we will go through the following steps:

Consider a Monte-Carlo version of classical policy iteration - Given some initial policy π_0, we approximate G_π over and over again using MC sampling (with additional starting point exploration). From here, for any action-value function Q, the greedy policy is the one that chooses a maximal action, i.e $\pi(s) = \text{argmax}_a Q(s, a)$. The above is neatly described in the following pseudo-code

2.10 On/Off-Policy methods

There are two types of policy methods

Algorithm 4 MCFV with Exploring Starts (ES) for estimating π_*

$\pi(s) \in A$ arbitrary for all $s \in S$ ▷ Initialization
$Q(s,a) \in \mathbb{R}$ arbitrary for all $s \in S$ and $a \in A$
$Returns(s,a) \leftarrow$ empty list [] for all $s \in S$ and $a \in A$
while True **do**
    $s_0 \in S$ and $a_0 \in A$ randomly chosen s.t all pairs (s_i,a_i) are reachable from (s_0,a_0) with prob>0
    $\tau \leftarrow \{s_0,a_0,r_1,s_1,a_1,r_2,...,s_{T-1},a_{T-1},r_T\}$ ▷ generate an episode using π from (s_0,a_0)
    $G \leftarrow 0$
    for step $t = T-1, T-2, ..., 0$ **do**
        $G \leftarrow \gamma G + r_{t+1}$
        if $(s_t,a_t) \notin \{(s_0,a_0),(s_1,a_1),...,(s_{t-1},a_{t-1})\}$ **then**   ▷ Verifying state-action first-visit
            $Returns(s_t,a_t).append(G)$
            $Q(s_t,a_t) \leftarrow avg(Returns(s_t,a_t))$
            $\pi(s_t) \leftarrow \text{argmax}_a Q(s_t,a)$
        end if
    end for
end while

- On-policy methods: attempts to evaluate or improve the policy that is being used to make decisions. As stated before, if π does not attain all state-action pairs with probability>0, we will poorly explore the space.
- Off-policy methods: attempt to evaluate or improve a policy other than the one used to generate the data (the one that selects actions).
  We work with two distinct policies:
  - The target policy π - the one that we wish to learn
  - the behavior policy b - the one used to generate the data

While On-policy methods tend to be more data efficient, they require new samples with each change of policy. Off-policy, on the other hand, are slower but more powerful and general, as they can be used to learn from various sources (like from a human expert)

2.10.1 Importance sampling for Off-Policy methods

Importance sampling is a technique for estimating expected values under one distribution given samples from another. It is performed by weighting returns according to the fraction of probabilities to a trajectory under some policy.

Lets assume that the behavior policy b is stochastic and the target policy π is deterministic. This means that the trajectories in the data (that were chosen by b) may be different than those chosen by π, which begs the question of how to calculate the expected return? The solution would be to weigh the return based on how it resembled the actual values returned by the target policy.

Consider the trajectory $\tau = \{s_t,a_t,s_{t+1},a_{t+1},...,s_T\}$. The probability to obtain τ given the starting state s_t and the actions $a_{t:T-1} \sim \pi$ is

$$Pr[\tau|s_t,a_{t:T-1} \sim \pi] = \prod_{k=t}^{T-1} \pi(a_k|s_k)Pr[s_{k+1}|s_k,a_k] \qquad (2.21)$$

Denoting the importance sampling for the time window $[t, t+1, ..., T-1]$ as $\rho_{t:T-1}$, we take the relative probability of the trajectory for the target and behaviour policy

$$\rho_{t:T-1} \equiv \frac{Pr[\tau|s_t, a_{t:T-1} \sim \pi]}{Pr[\tau|s_t, a_{t:T-1} \sim b]} = \prod_{k=t}^{T-1} \frac{\pi(a_k|s_k)}{b(a_k|s_k)} \tag{2.22}$$

Notice that even though the probabilities $P[s'|s, a]$ may be unknown, they cancel out in 2.22. From here, we can use $\rho_{t:T-1}$ and the return G_t of the behaviour policy b to obtain V_π, as

$$V_\pi(s) = \mathbb{E}[\rho_{t:T-1} G_t | s_t = s] \tag{2.23}$$

For example, if some trajectory τ is twice as plausible under b than it is under π, the expected return for π would be $1/2$ (in expectation) the return under b, which can also be seen as $\rho = 1/2$.

From here, we can take the MC algorithm (that averages returns), provide it with episodes following b but still estimate V_π.

Let $\mathcal{T}(s)$ be all time steps state s was visited (over all episodes), and $T(t)$ be the time of termination after time t for a given episode, then $\{G_t\}_{t \in \mathcal{T}(s)}$ is the set of returns associated with s across all episodes, and $\{\rho_{t:T(t)=1}\}_{t \in \mathcal{T}(s)}$ are the corresponding IS ratios. To estimate V_π we can use

$$V_\pi(s) = \frac{\sum_{t \in \mathcal{T}(t)-1} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|} \tag{2.24}$$

2.11 ε-Greedy Algorithms

In RL we often need to balance between
- Exploration - experimenting with multiple actions to better assess the expected reward
- Exploitation - attempt to maximize the reward by choosing the optimal action. We can do this by, for example, estimating our Q function with MC as

$$\hat{Q}_t(a, s) = \frac{1}{N_t(a)} \sum_{t=1}^{T} (r_t | a_t = a, s_t = s)$$

where $N_t(a)$ is the number of times the action a was chosen, and then choose an action

$$a_t^* = \underset{a \in A}{\operatorname{argmax}} \hat{Q}_t(a, s)$$

Notice how these two may collide, as when we explore the environment we also may not always choose an optimal action. In an ε-greedy approach, we explore with probability ε, and in all other cases we choose the optimal action:

- With probability $1 - \varepsilon$: select $a_t^* = \underset{a \in A}{\operatorname{argmax}} \hat{Q}_t(a, s)$
- with probability ε: choose a random $a \in A$

Intuitively speaking, as we maintain the ability to explore forever, we will eventually find an optimal policy. We also make sure to not include the randomness in the testing phase.

2.12 Temporal Difference (TD) Learning

When discussing MC learning, we showed how sampling the environment without knowing the dynamics of the system could be enough to learn V, Q, and a policy π directly. On the other hand,

when using MC we must make sure that the episodes are finite, and we could only learn based on complete episodes. In TD learning, on the other hand, both infinite environments and incomplete sequences learning is possible, as we can update our approximation after every step (compared to after every episode in MC). We define the TD-error as the difference between the optimal value function V_t^* and the current prediction V_t:

$$\delta_t^{TD}(s) = V_t^*(s) - V_t(s)$$
$$= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - V_t(s)$$
$$= r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+2} - V_t(s) \qquad (2.25)$$
$$= r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} - V_t(s)$$
$$= r_{t+1} + \gamma V_{t+1}^* - V_t(s)$$

As we do not know V_{t+1}^*, we can approximate it using the predicted V_{t+1}

$$\delta_t^{TD} \approx r_{t+1} + \gamma V_{t+1} - V_t \qquad (2.26)$$

This error is used as the general update rule, where we also add a learning rate α (much like in gradient descent, where we can think of $V_{t+1} - V_t$ as the term ∇V_t)

$$V_t \leftarrow V(t) + \alpha \left[r_{t+1} + \gamma V_{t+1} - V_t \right] \qquad (2.27)$$

(R) notice that we do not use an expectation term (as seen in policy iteration for example), as the update rule is the result of looking only one step into the future, given some episode rollout.

Algorithm 5 One-Step TD [TD(0)]

Require: π the policy to evaluate, $\alpha \in \mathbb{R}$
  $V(s) \in \mathbb{R}$ arbitrary initialized for all $s \in S$
  for each episode E **do**   ▷ sampling a trajectory like in MC
    for each $s \in E$ **do**
      $a \leftarrow \pi(s)$
      $r, s' \leftarrow$ take action a and observe r, s' ▷ taking one step to the future
      $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$ ▷ using eq. 2.27
      $s \leftarrow s'$
    end for
  end for
  return V

2.12.1 On-Policy TD Control: SARSA

SARSA is an on-policy (remember 2.10) algorithm that used TD(0) in order to approximate the Q-function. The idea will be similar to the value function update error, and the MC sampling would

---
Algorithm 6 SARSA

Require: $\alpha \in \mathbb{R}$

  $Q(s,a) \in \mathbb{R}$ arbitrary initialized for all $s \in S$ and for all $a \in A$

  for each episode E **do**   ▷ sampling a trajectory like in MC

    for each $s \in E$ **do**

      choose a from s given Q ▷ like in ε-greedy (2.11)

      $r,s' \leftarrow$ take action a and observe r,s' ▷ taking one step to the future

      choose a' from s' given Q ▷ like in ε-greedy (2.11)

      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$

      $s \leftarrow s', a \leftarrow a'$

    end for

  end for

  return Q

---

consider both the next state *and* the next action. Do notice that, as always, to approximate Q we must find its optimal value for every $s \in S$, and $a \in A$ - which is computationally difficult.

    (R)   the name SARSA stems from the idea that the update rule uses the quintuple $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} \rangle$

2.12.2 Off-Policy TD Control: Q-Learning

In Q-Learning, we do not look at the next $Q(s',a')$, but on the Q-function that has maximal value over all possible actions $\max_a Q(s',a)$. This means that the corresponding maximal Q-function was provided given an action that may or may not be the result of our policy π (hence, it is Off-Policy). More specifically - Q-learning is based on a greedy approximation of the optimal policy - which is the behaviour policy (compared to SARSA, that only used the current policy). Notice that we still use the current policy, as it determines which state-action pairs are visited and updated.

Algorithm 7 Q-Learning

Require: $\alpha \in \mathbb{R}$

  $Q(s,a) \in \mathbb{R}$ arbitrary initialized for all $s \in S$ and for all $a \in A$

  for each episode E **do**   ▷ sampling a trajectory like in MC

    for each $s \in E$ **do**

      choose a from s given Q ▷ like in ε-greedy (2.11)

      $r,s' \leftarrow$ take action a and observe r,s' ▷ taking one step to the future

      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_a Q(s',a) - Q(s,a))$

      $s \leftarrow s'$

    end for

  end for

  return Q

    Also notice we did not track down the next action a', as we did not use it (scanned all $a \in A$ instead). It should also be stated that Q-Learning usually converges quicker, due to the optimal choice of $\max_a Q(s',a)$. Having said that, as we do not take into consideration the next state a', using ε-greedy actions might mean that we take a step into a state with very bad reward (like falling down a cliff), simply as we are less conservative (due to exploration and not admitting to a next action).

3. DQN & it's derivatives

This section is based on sources I found online (because the lecture was not uploaded). Our new goal would be to use a parameterized function approximator to represent the state-action Q-function, instead of representing it with a table. In other words, we wish to find a function $\hat{Q}(s, a, \theta) \approx Q(s, a)$, where θ is our function's parameters and $Q(s, a)$ is the true function/oracle.

Lets start of with the ideal assumption, in which the oracle $Q(s, a)$ is accessible. Our function approximator could be learned using SGD, that is, by minimizing the squared loss $J(\theta)$ w.r.t to the oracle over batches sampled from our environment

$$J(\theta) = \mathbb{E}\left[(Q(s, a) - \hat{Q}(s, a, \theta))^2\right] \tag{3.1}$$

We update our parameter vector θ using $\theta \leftarrow \theta - \Delta\theta$, where

$$
\begin{aligned}
\Delta\theta &= -\frac{1}{2}\alpha\nabla_\theta J(\theta) \\
&= -\frac{1}{2}\alpha\nabla_\theta \mathbb{E}\left[(Q(s, a) - \hat{Q}(s, a, \theta))^2\right] \\
&= -\frac{1}{2}\alpha \cdot 2\mathbb{E}\left[Q(s, a) - \hat{Q}(s, a, \theta)\right] \cdot (-\nabla_\theta\hat{Q}(s, a, \theta)) \\
&= \alpha\mathbb{E}\left[Q(s, a) - \hat{Q}(s, a, \theta)\right]\nabla_\theta\hat{Q}(s, a, \theta)
\end{aligned}
\tag{3.2}
$$

As $Q(s, a)$ is generally unknown, it must be replaced with some approximated target. Recall that in SARSA for example, our target was based on the temporal difference $r + \gamma\hat{Q}(s', a', \theta)$. In classic Q-learning, on the other hand, we used an off policy approach, in which our target was $r + \gamma\max_{a' \in A}\hat{Q}(s', a', \theta)$. As we now proceed to describe DQN, we will follow the update rule of

$$\Delta\theta = \alpha\mathbb{E}\left[r + \gamma\max_{a' \in A}\hat{Q}(s', a', \theta) - \hat{Q}(s, a, \theta)\right]\nabla_\theta\hat{Q}(s, a, \theta) \tag{3.3}$$

3.1   Deep Q-Network

We now ask how shall we describe our approximated $\hat{Q}(s, a, \theta)$, and the answer is a deep neural network.

3.1.1   Architecture

In the original paper, the DQN architecture was based on convolutional neural network. The network takes in an input of shape $84 \times 84 \times 4$ (a processed batch of images, which is considered the state)[1], and propagates this input via three convolutional layers. To finally come up with an action value, there are two fully connected layer, where the last one has a single output for every possible Notice how our function approximates an action for every state, hence can be written as $\forall\, s \in S :$ $\hat{Q}(s, \theta) : \mathbb{R}^{|S|} \to \mathbb{R}^{|A|}$.

3.1.2   Training DQN

Those of us who are familiar with supervised learning, might think that the kind of "true-label" introduced in 3.3 is not much of a true label at all, as it is based on the network itself, which is constantly changing. This fact makes the learning process unstable, therefore few "tricks" were introduced in the paper - the *Experience Replay*, and a separate *Target Network*. Summarizing the two in a short sentence, our Q-network is learned by minimizing

$$J(\theta) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1}} \left[y_t^{DQN} - \hat{Q}(s_t, a_t, \theta) \right] \tag{3.4}$$

where y_t^{DQN} is the one-step TD component

$$y_t^{DQN} = r_t + \gamma \max_{a' \in A} \hat{Q}(s_{t+1}, a', \theta^-) \tag{3.5}$$

Notice how we use both θ and θ^-, where θ is the original parameters vector of the network but θ^- represents the parameters of the target network. By doing so we update θ based on values from a previous version of our model. Furthermore, the batches we sample are provided from all past transition tuples.

Experience Replay

We store the transitions $\langle s_t, a_t, r_t, s_{t+1} \rangle := e_t$ at each time-step in a fixed buffer $D_t = \{e_1, e_2, ..., e_t\}$, and During SGD, we only sample uniformly from D. This is because
- Data efficiency: each experience e_i can potentially be used in many updates
- De-correlation: randomly sampling leads to de-correlation between consecutive experiences. As correlation breaks, expected values will fluctuate less, meaning that the variance in sampling is reduced, hence stability is increased.
- Smoother divergence: As our training process is based on a large number of experiences, outliers tend to average out with the rest of the samples, leading to less oscillations in the training process.

Note that in a more sophisticated experience replay, we might weigh experiences based on their importance and keep the relevant ones for longer.

---

[1] Originally, images were $210 \times 160 \times 3$, but the preprocessing applied spacial dimension reduction to 84×84, extracted the Y channel from the RGB and concatenated 4 consecutive images as "memory"

Target Network

Another stability improvement comes from the fact that we use two neural nets. Ideally, we would like to minimize the effect of our targets being *Non-Stationary*, and we do so by setting a network that is only updated after $C \gg 1$ steps. This means that our non-stationary target will, in a sense, be stationary for at least C steps, hence stabilizing our learning process even further.

Algorithm 8 Deep Q-Learning (DQN)

Initialize experience replay D with fixed size
Initialize network $\hat{Q}$ with random weights θ
Initialize target network $\hat{Q}_T$ with random weights $\theta^- = \theta$
for episode $m = 1, 2, ..., M$ **do**
    observe environment to get s_1
    take action $a_t \leftarrow \begin{cases} \text{random,} & \text{with probability } \varepsilon \\ \operatorname{argmax}_{a \in A} \hat{Q}(s_t, a, \theta), & \text{otherwise} \end{cases}$
    $r_t, s_{t+1}, done_i \leftarrow$ Take action a_t in the environment
    $D \leftarrow D \cup \{\langle s_t, a_t, r_t, s_{t+1}, done_i \rangle\}$
    $B \leftarrow$ sample a random batch $\{\langle s_i, a_i, r_i, s_{i+1}, done_i \rangle\}_{i=1}^N$ from D
    for every experience $\langle s_i, a_i, r_i, s_{i+1}, done_i \rangle \in B$ **do**
        $y_i \leftarrow r_i$ **if** $done_i$ **else** $r_i + \gamma \max_{a' \in A} \hat{Q}_T(s_{i+1}, a', \theta^-)$
    end for
    perform SGD on $J(\theta) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{Q}(s_i, a_i, \theta))^2$ w.r.t θ
    every C steps $\theta^- \leftarrow \theta$
end for

3.2 Double Deep Q-Network (DDQN)

We'd like to state that the max operator is prone to over estimation. Consider the following example:[2]

∎ **Example 3.1** Say $N \gg 1$ people have equal weight of $80kg$, and we'd like to measure all of their weights using a weighing scale that is off by ± 1 kg (equal probability to measure > 80 and < 80). Lets run two sets of measurements:

- Denote the weight measurement of the i'th person as X_i, and set $Y = \max_i X_i$. We can intuitively understand that almost surely $Y > 80$, as almost surely there exists some j for which $X_j > 80$. This really tells us that the max operator is prone to over estimation when noise is introduced to the system.
- As a second experiment, we will measure each person's weight twice and store the values in X_1^i, X_2^i. to estimate Y, we first calculate $n = \operatorname{argmax}_i X_1^i$. Next, we take the second measurement $Y = X_2^n$ as our maximal value. Notice that as X_2^n is independent from X_1^n, it is equally likely to overestimate the real value or underestimate it, hence it is not systematically over-optimistic. So everyone weighs $80kg$ and $X_1^n > 80$ with high probability, but X_2^n is both > 80 and < 80 with even probability.

∎

---

[2]from CIS 522 YouTube channel

   With the example above in mind, notice how in the DQN algorithm we both choose an action a_t using $\hat{Q}(s_t, a, \theta)$ and evaluate it (when calculating y_i), which means that we tend to overestimating the target values. To address this issue we replace the current update of y_i (when not $done_i$) with

$$y_i^{DoubleDQN} = r_i + \gamma \hat{Q}\left(s_{i+1}, \operatorname*{argmax}_{a' \in A} \hat{Q}(s_{i+1}, a', \theta), \theta^-\right) \tag{3.6}$$

In other words, we choose an action (argmax) using a network with parameters θ (that is currently being trained), but evaluate the action using a network with parameters θ^- (that is not being trained)

3.3   Dueling network

Starting off with a new definition, the *Advantage* function that is associated with a policy π and a state-action function Q is given by

$$A^\pi(s, a) = Q^\pi(s, a) - V_\pi(s) \tag{3.7}$$

We can think of the advantage as some measurement to how important is some action, as in - take the importance of a given state-action pair ($Q(s, a)$) and subtract the importance of the state ($V(\pi)$) to get the importance of a given action. Note that as $V^\pi(s) = \mathbb{E}_{a \sim \pi}[Q^\pi(s, a)]$ (see eq. 1.6), we have that $\mathbb{E}_{a \sim \pi}[A^\pi(s, a)] = 0$.

A *Dueling* network approximates the Q-function by first separating the output of the network into two distinct elements - one is $V(s)$ and the other is $A(s, a1), A(s, a_2), ...,$ and then extracts from both the value of Q. Recall that in regular DQN, for every state we estimated the value of all action
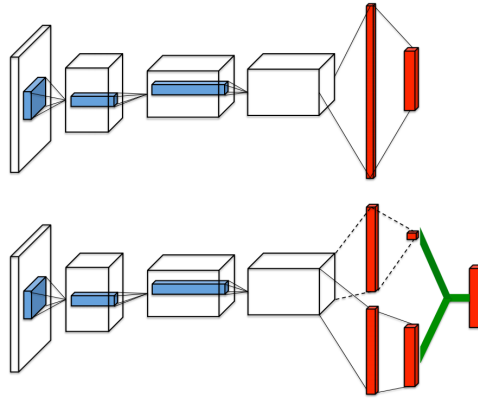

Figure 3.1: (Top) a standard DQN architecture, where the last layer is a vector representing $Q(s, a_1), Q(s, a_2), ...$ (Bottom) a Dueling DQN, where the upper branch represents the value of $V(s)$, the lower branch represents the values of $A(s, a_1), A(s, a_2), ...$ and the final layer represents $Q(s, a_1), Q(s, a_2), ...$ As the input/output are the same, Alg. 8 can be applied given the Dueling architecture as well.

choices. This may be unnecessary in cases where an action has little to no effect on the outcome. For example - if you fell down a cliff, it is really irrelevant what are the action values of steering the wheel. As we split our architecture to two distinct components, we can learn state values without having to learn the effect of each action on those states - as those may be an unnecessary computation for some states.

3.3.1 Implementation

Denoting the output of the upper branch as $V(s,\theta,\theta_V)$ and the output of the lower branch as $A(s,\theta,\theta_A)$ where θ are shared parameters and θ_V, θ_A are distinct parameters for every branch, it may seem reasonable to add these values to obtain $Q(s,a,\theta,\theta_V,\theta_A)$. The immediate problem that arises is that given such Q, we cannot recover the exact values of both A and V, as for example it may be the case that $Q = (A+x) + (V-x)$, and x can be chosen freely.
To distinguish A and V, the following formula was applied:

$$\hat{Q}(s,a,\theta,\theta_A,\theta_V) = V(s,\theta,\theta_V) + \left[A(s,a,\theta,\theta_A) - \max_{a' \in Actions} A(s,a',\theta,\theta_A) \right] \tag{3.8}$$

This trick forces the Q value associated with the maximizing action to equal V (as for the maximizing action, which is the action that is choose, the squared brackets zero out). This means that the upper stream (V) can be identified as the Q function value, and the lower stream would be the advantage function.

(R) Alternatively, one can use

$$\hat{Q}(s,a,\theta,\theta_A,\theta_V) = V(s,\theta,\theta_V) + \left[A(s,a,\theta,\theta_A) - \frac{1}{|Actions|} \sum_{a' \in Actions} A(s,a',\theta,\theta_A) \right] \tag{3.9}$$

which was shown to improve stability, as was stated in the article

4. Policy Gradients

(see also 1.3).

Instead of value function of state-action value function approximation, we can aim to model the policy itself. Again, we take the path of function approximation, rather than a look-up table approach that assign every state an action. Generally speaking, we define a parameterized policy function π_θ and try to optimize it using a gradient ascent optimization process

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{4.1}$$

where $\nabla J(\theta_t)$ is an estimated performance measurement.

π_θ itself may have various kinds of outputs, given the problem setup (see 1.1.3 for more info)

- For a discrete action space - $\pi_\theta(s) \in \mathbb{R}^{|A|}$, where each entry may represent a probability associated with an action
- For continuous action space - the output may be mean and covariance of a Gaussian distribution (from which we sample an action), or a floating point value as commonly used in regression tasks

Policy-gradient methods can learn an appropriate balance between exploration/exploitation, hence are preferred over ε-greedy approaches. This also means that they provide us with a smoother action selection, as changes in final action choice are based in changes to the parameters of the model, rather than some random value exceeding ε

4.1 The Policy-Gradient theorem

Let us define the performance measure $J(\theta)$ as the value function associated with the parameterized policy

$$J(\theta) = V_{\pi_\theta}(s) \tag{4.2}$$

The PG theorem states that the gradient of $J(\theta)$ is proportional to the gradient of our policy, weighted by the Q function over all states, and by the average number of times each state is visited

in a trajectory $\mu(s)$.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a Q_{\pi_\theta}(s,a) \nabla \pi_\theta(a|s) \tag{4.3}$$

The proof can be seen in S&B2020, page 325.

The PG theorem provides us with an analytic term to the change in performance w.r.t the parameters of our policy.

4.2 The REINFORCE Algorithm

If we follow a target policy π, the weighted sum over all states in eq. 4.3 can be replaced with expected value over the policy, as the number of times $\mu(s)$ some state was reached under π is the same as summing all values under the policy associated with the state s. In other words, $\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_a Q_{\pi_\theta}(s,a) \nabla \pi_\theta(a|s) \right]$.

Supposedly, this is enough to formulate a gradient-ascent algorithm, as the mean value is replaced with sampling a set of experiences - $\theta_{t+1} = \theta_t + \alpha \sum_a \hat{Q}(s_t, a, \theta_Q) \nabla \pi_\theta(a|s_t)$. Having said that, this approach means that an approximation of Q is needed as well, and the update rule takes into consideration *all* actions, which is not ideal.

A preferred option would be to articulate a learning scheme that is based only on taking one step in our environment, i.e taking an action a_t and understanding from it alone how to change the weights:

$$\begin{aligned}
\nabla J(\theta) &= \mathbb{E}_\pi \left[\sum_a Q_{\pi_\theta}(s_t, a) \nabla \pi_\theta(a|s_t) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi_\theta(a|s_t) Q_{\pi_\theta}(s_t, a) \frac{\nabla \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} \right] \text{ [multiply and divine the same value]} \\
&= \mathbb{E}_\pi \left[Q_{\pi_\theta}(s_t, a_t) \frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right] \text{ [replace } \sum_a[...] \text{ with sampling } a_t \sim \pi] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right] [Q_\pi(s_t, a_t) = \mathbb{E}_\pi[G_t|s_t, a_t]]
\end{aligned} \tag{4.4}$$

Hence we can now rewrite the update rule as $\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}$. Notice that this approach is only applicable in the episodic setting, as to calculate the return G_t we need to sum all future rewards until the end is reached. Intuitively, we can decompose this expression as

- $\nabla \pi_\theta(a_t|s_t)$ is a vector pointing to the direction (in parameter space) that increases the probability of taking a_t the most, given the state s_t
- the "amplitude" is $\frac{G_t}{\pi_\theta(a_t|s_t)}$, meaning that the update tends to increase G_t - actions are chosen if they maximize the return, and decrease $\pi_\theta(a_t|s_t)$ - this makes sense as otherwise, maximizing $\pi_\theta(a_t|s_t)$ means that actions that are frequent (high $\pi_\theta(a_t|s_t)$) will be chosen even if they do not yield high return.

One last trick is to notice that $\nabla \ln x = \nabla x / x$, so the final update rule is

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi_\theta(a_t|s_t) \tag{4.5}$$

From here, we can generate an episode from our initialized policy function, calculate the return as a sun of discounted (or undiscounted) reward (remember that we assume finite episodes setting) and update the network parameters using the update rule from above.

Algorithm 9 REINFORCE: MC Policy-Gradient control (episodic case) for control

---

Require: a (differentiable), randomly initialized policy $\pi_\theta(a|s)$
Require: learning rate α
  while did not converge **do**
    generate an episode $\{s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T\}$ given π_θ
    for each step $t = 0, 1, ..., T-1$ **do**
      $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
      $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi_\theta(a_t|s_t)$
    end for
  end while

4.2.1 REINFORCE with Baseline

Sampling an episode may cause fluctuations and slow convergence properties. To mitigate the problem we can include a baseline, which can be any function that we compare our predicted return to. We use this baseline in the update rule as $\theta_{t+1} = \theta_t + \alpha(G_t - b(s_t)) \frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}$. We also must ensure that our baseline does not depend on the actions, as those are what we aim to learn. More specifically, a reasonable baseline would be the value function $\hat{V}(s_t, \theta_V)$ (recall that V is the mean value of G, starting from s_t and following the policy π). $\hat{V}$ should be calculated using some method that was previously discussed.

Algorithm 10 REINFORCE: MC Policy-Gradient control (episodic case) for control

---

Require: a (differentiable), randomly initialized policy $\pi_\theta(a|s)$
Require: a (differentiable), randomly initialized value function $\hat{V}(s, \theta_V)$
Require: learning rate α_V, α
  while did not converge **do**
    generate an episode $\{s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T\}$ given π_θ
    for each step $t = 0, 1, ..., T-1$ **do**
      $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
      $\delta \leftarrow G - \hat{V}(s_t, \theta_V)$
      $\theta_V \leftarrow \theta_V + \alpha_V \delta \nabla \hat{V}(s_t, \theta_V)$
      $\theta \leftarrow \theta + \alpha \gamma^t \delta \nabla \ln \pi_\theta(a_t|s_t)$
    end for
  end while

Intuitively, $G_t - \hat{V}(s_t)$ tells us how good we did after time step t (G_t) compared to what was expected to be achieved for that same timestamp ($\hat{V}(s_t)$). If, for example, the performance was better than expected, the log-likelihood term for the action that was used is now weighted based on that performance, encouraging our agent to continue with those behaviors. If we anticipated the return exactly, then the actions we chose are ideal, hence no more updates are required ($G(t) - b(s_t) = 0$). Comparing this to the previous method with no baseline, where we always multiplied by the actual return G_t, unnecessary updates would have happened, which hinders convergence

4.3 Actor-Critic methods

let's go over two new definitions

- Actor-only methods: use a parameterized representation + gradient ascent and relies on sampling the environment (aka acting on the environment). such methods have high variance due to sampling and do not accumulate older information (meaning that once the policy, for example, is updated, we start over again)
- Critic-only methods: directly solving for a value function / optimizing a parameterized function (aka creating a value criterion based on states and actions). Theoretically, these methods can yield optimal policies, though it is not a guarantee

We can think of an actor method as a method that, well, acts on the environment, whereas the critic provides some measurement of that performance.

The concept of combining the two means that we use our critic to evaluate the state and reward given by the system, provide a value that will be inserted to the actor, and the actor will eventually output some action to be performed on the environment. AC methods are the intersection between Value-based methods and Policy-based methods:

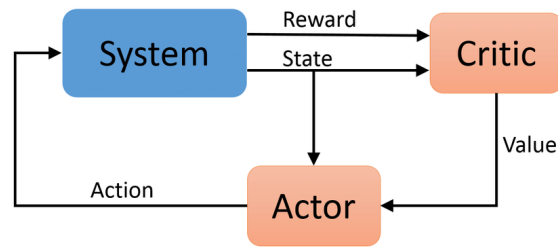|	The policy is	The Value function is
Value-based	implicit (ε-greedy)	estimated
Policy-based	estimated	no use of Value function
Actor-Critic based	estimated	estimated

Figure 4.1: Actor-Critic diagram. Notice that w.r.t to the table above, both the actor and the critic are estimated (i.e parameterized)

The formalism of AC methods is similar to Policy iteration (recall alg. 1), as we alternate between policy evaluation, where the value function is being estimated, and a policy improvement, where given the evaluated policy we improve our policy. More specifically, the actor attempts to improve the policy (using Policy gradients for example or argmax Q), and the critic evaluates the current policy.

4.3.1 One-Step AC algorithm

Recall our td-update decomposition (see eq. 2.27), $V_t \leftarrow V(t) + \alpha [r_{t+1} + \gamma V_{t+1} - V_t]$. We can use this update rule to reformat the REINFORCE w/Baseline:

$$
\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha(G_t - \hat{V}(s_t, \theta_V))\nabla \ln \pi_\theta(a_t|s_t) \\
&= \theta_t + \alpha(r_{t+1} + \gamma \hat{V}(s_{t+1}, \theta_V) - \hat{V}(s_t, \theta_V))\nabla \ln \pi_\theta(a_t|s_t) \\
&= \theta_t + \alpha \delta_t^{TD} \nabla \ln \pi_\theta(a_t|s_t)
\end{aligned}
\tag{4.6}
$$

Algorithm 11 One-Step Actor-Critic

Require: a (differentiable), randomly initialized policy $\pi_\theta(a|s)$
Require: a (differentiable), randomly initialized value function $\hat{V}(s, \theta_V)$
Require: learning rate α_V, α
  while did not converge **do**
      sample an initial state s
      define γ term $I \leftarrow 1$
      while s is not terminal **do**
         take action $a \sim \pi_\theta(\cdot|s)$ and observe s', r
         calculate TD error $\delta \leftarrow r + \gamma\hat{V}(s', \theta_V) - \hat{V}(s, \theta_V)$
         update $\hat{V}$'s weights $\theta_V \leftarrow \theta_V + \alpha_V \delta \nabla_{\theta_V} \hat{V}(s, \theta_v)$
         update π_θ's weights $\theta \leftarrow \theta + \alpha_\theta I \delta \nabla_\theta \ln \pi_\theta(a|s)$
         $I \leftarrow \gamma I, s \leftarrow s'$
      end while
  end while

The above can be summarized to the following pseudo-code Notice how compared to MC control (like in the original REINFORCE), the update rule happens after every step, meaning we do not have to simulate an entire episode. This makes the process less variable, allowing for faster convergence.

4.3.2 Asynchronous Advantage Actor-Critic (A3C)

Note that we are not bounded to only one critic, and multiple critics can operate simultaneously to output a singular value. Using multiple critics also means that we can further explore the environment, as for example the initialization can be different for every one of them. We can think of A3C as some sort of committee method between critics, but other versions exists (weighted sum of critics, or even an attention mechanism). Broadly speaking, A3C is implemented as a wrapper to the basic Actor-Critic, where every critic generates an update rule $\Delta\theta, \Delta\theta_V$, and once all critics finished, the global update rule for θ, θ_V is generated.

5. Imitation Learning

Imitation learning is a learning process that is based on data provided from an "expert" - a being from which we understand how to perform some task/movement/etc..
Let us start with a new definition - the *Regret*, which is the difference between following the optimal policy and some other policy.

5.1 The Regret

(R) This part seems to use slightly different notations, but hopefully still be understandable

We define an action value as the expected reward for an action a

$$Q(a) = \mathbb{E}[r|a] \tag{5.1}$$

Given an optimal policy π^*, the optimal action value is

$$V^* = \max_a Q(a) \tag{5.2}$$

The Regret at time t is the expected loss of all action values w.r.t to the optimal value

$$\ell_t = \mathbb{E}[V^* - Q(a_t)] \tag{5.3}$$

and the *total* regret up to time T is

$$L = \mathbb{E}\left[\sum_{t=1}^{T}(V^* - Q(a_t))\right] \tag{5.4}$$

We sometimes refer to $V^* - Q(a)$ as the *gap*, and denote it with Δ_a.
Notice how when we maximize the rewards the action value $Q(a)$ increases, getting closer to V^*. In other words - maximizing the rewards $\leftrightarrow$ minimizing the regret. With this intuition in mind, our

goal would be to perform some iterative optimization process in which every in every iteration we examine some π_i and hope it will be as close to π^* as possible.

Next, we define the count $N_t(a)$, which is the number of times action a is selected by the t'th time step, and use it to reformulate the total regret

$$
\begin{aligned}
L &= \mathbb{E}\left[\sum_{t=1}^{T}(V^* - Q(a_t))\right] \\
&= \sum_{a \in A} \mathbb{E}[N_t(a)](V^* - Q(a)) \\
&= \sum_{a \in A} \mathbb{E}[N_t(a)]\Delta_a
\end{aligned}
\tag{5.5}
$$

Let's observe the regret value under various algorithms:

- A greedy algorithm that always chooses $\mathrm{argmax}_a \hat{Q}_t(a)$: as we do not explore the environment, we may lock into some sub-optimal policy. This means that the action values remain constant, so as T increases we sum over more constant terms $(V^* - Q(a_t)) \sim$const, i.e the regret increases linearly over time.
- An ε-greedy algorithm: will also lead to linear behaviour of the regret, but with a slower trend (slope). This is due to the fact that with probability ε, an exploration is made.
- Decaying ε-greedy: over time, we control the trade-off between exploration and exploitation, closing the gap between $Q(a)$ and V^* in a logarithmic fashion (needs clarification)

5.2 Imitation learning

In some cases, the task that is to be solved is "too difficult" for a model to learn from scratch, so we'd like to convert it to a series of prediction problems instead, based on some input provided by an expect demonstrator. This means that our problem is decomposed to a set of actions that are to be learned in a supervised fashion, which may simplify the task and improve overall results. For example, if we are interested to teach a model to fly a helicopter, we will provide it with features that represent a set of flights performed by a human (videos/control parameters over time/etc..) and expect our model to learn the observed behaviour. Notice that this approach comes with problems, and the most prominent of those is the fact that it is difficult to learn failure recovery. More precisely, our model is only as good as the expect, which means that new states (that the expert did not encounter) will be difficult to recover from. In a sense, we want to train our model on all possible states, but as this is not feasible, we must consider which states are more relevant than others.

In the following section, we use these notations:

1. T - the tasks' horizon
2. d_π^t - the states distribution for times$\in [1, t-1]$ when following π
3. d_π - the average state distribution $\frac{1}{T}\sum_{t=1}^{T} d_\pi^t$
4. $C(s,a)$ - the expected immediate cost of performing action a in state s (can be thought of as the opposite of the reward)
5. $C_\pi(s)$ - expected immediate policy cost from state s. $\mathbb{E}_{a \sim \pi(s)}[C(s,a)]$
6. $J(\pi)$ the total cost of the policy. $J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{s \sim d_\pi^t}[C_\pi(s)] = T \cdot \mathbb{E}_{s \sim d_\pi}[C_\pi(s)]$
7. $\ell(s,\pi)$ - observed surrogate loss, which is the gap of π compared to π^* given state s

5.2.1 Apprenticeship Learning

In the most basic form of imitation learning, we can perform this hand-wavy approach:

1. watch an expert perform a task and record state-action trajectories
2. use those trajectories to learn the model dynamics (the transitions matrix $P \in \mathscr{M}^{|S| \times |A|}$)
3. use some RL approach to find a near-optimal policy
4. if the policy is good enough - finish. otherwise, start over again

Apprenticeship learning is completely greedy, as there is no exploration term - this means that it is more suitable for cases where exploration is dangerous or costly. Also notice how we try to learn both the dynamics of the environment *and* the transitions function. Ideally, we'd prefer to disentangle the two, as such mechanism is less likely to generalize to new states.

5.2.2 Supervised learning

As a first, naive, solution - lets reduce the sequence trajectory to many decouples supervised learning problems. With this framework in hand, our objective may be to find a policy that minimizes the observed surrogate loss, under some current state distribution

$$\pi^* = \operatorname*{argmin}_{\pi \in \Pi} \mathbb{E}_{s \sim d_\pi}[\ell(s, \pi)] \tag{5.6}$$

As this is a sequential learning problem, we are faced with non i.i.d samples (d_π depends on π and the action now may affect the states in the next sequence), hence our optimization is not as easy. The problem with eq. 5.6 is that the states are sampled under a distribution induced by some non-optimal policy, which results in a total cost that diverges from the cost of the optimal policy's cost (namely, $J(\pi) \leq J(\pi^*) + T^2 \varepsilon$, where ε is an error rate). This is all related to our initial intuition - when a classifier makes a mistake, a new unseen state is being introduced to the agent. From this point on, w.h.p all actions chosen by the agent will be wrong, and the error compounds.

5.2.3 Forward Training

the algorithm trains a non-stationary policy, meaning that at each iteration t we train a new policy π_t in the following manner

*   at the t'th iteration, we sample a trajectory generated by the latest policy π_{t-1}
*   we ask our expect (π^*) to provide state action for that provided trajectory
*   we train a new classifier to provide the policy for $t+1$
*   we use π_{t+1} to advance the system one step forward

More formally, we perform the following pseudo-code

Algorithm 12 Forward Training

Require: Initialize non-stationary policy $\{\pi_i^0\}_{i=1}^T$
Require: expect policy π^*
   for $i = 1, 2, ..., T$ **do**
      sample T-step trajectories by following π^{i-1}
      generate data $\mathscr{D} = \{\langle s_i, \pi^*(s_i) \rangle\}$ taken by the expert at step i
      train the policy $\pi_i^i = \operatorname*{argmin}_{\pi \in \Pi} \mathbb{E}_{s \sim \mathscr{D}}[e_\pi(s)]$
      update policy $\pi_j^i = \pi_j^{i-1}$ for all $j \neq i$
   end for
   return $\{\pi_i^T\}_{i=1}^T$
---

where $e_\pi(s) = \mathbb{E}_{a \sim \pi(s)}[e(s, a)]$ and $e(s, a) = \mathbb{1}[a \neq \pi^*(s)]$. Examining the bounds of the algorithm, the regret is $\leq O(uT\varepsilon)$, where u is the diversion from the optimal policy ($u \leq T$).

As we train the algorithm on currently visited states, we can make better decisions and recover from mistakes. But if T is large, the algorithm is generally impractical for real-life applications.

5.2.4 Dataset Aggregation (DAgger)

Compared to a pre-defined set of expert demonstrations, which causes compound error accumulation, DAgger aims to mitigate compounding errors by constantly letting the expert tag the new experiences our agent encounters.

Algorithm 13 DAagger

Initialize dataset $\mathscr{D} \leftarrow \emptyset$
Initialize random policy $\hat{\pi}_1 \in \Pi$
for $i = 1, 2, ..., N$ **do**
    sample T-step trajectory using π_i
    get dataset $\mathscr{D}_i = \{(s, \pi^*(s)\}$ of visited states by π_i and actions given by expert
    aggregate $\mathscr{D} \leftarrow \mathscr{D} \cup \mathscr{D}_i$
    train $\hat{\pi}_{i+1}$ on $\mathscr{D}$
end for

Though it was shown to work well for both simple and complex problems (linear regret), one disadvantage is that the expert must be available throughout the training process, which may not always be the case. Furthermore, another shortcoming is when the learner's policy is drastically different from the expert's policy. Think for example of a person who has just received his driver's license - if that person would observe the driving performance of a Formula 1 racer driving a track, not much would have been processed and transferred to the new driver, and he will probably not be able to reconstruct the experienced driver's patterns. Obviously, to properly teach the new driver the complicated set of skills, there has to be some learning curve, allowing for an increasing level of complexity.

(R) In some versions of the dagger algorithm, the policy used is a convex sum of the expert's policy and the current trained policy, i.e $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$. Furthermore, the β_i term usually decays, as to account for the expert less over time. in terms of the pseudo-code in alg. 13, we can add this line right at the beginning of the **for** loop

5.2.5 DAgger with coaching

Intuitively speaking, we train our policy by presenting it with examples that are increasingly challenging. Formally we define the *hope* action

$$\tilde{\pi}_i(s) = \underset{a \in A}{\operatorname{argmax}} \left[\lambda_i \cdot score_{\pi_i}(s, a) - L(s, a) \right] \tag{5.7}$$

This is for
- $\lambda_i \geq 0$ specifies how close the coach is to the expert
- $score_{\pi_i}$ is the likelihood of our agent to choose an action a in state s
- $L(s, a)$ is the immediate cost

As the oracle's action choices are minimizers of the cost L, when our policy's actions result in a small $L(s, a)$ for a given s, we understand that the actions that were chosen are close to the oracle

actions. Furthermore, if a score of an action $score_{\pi_i}(s,a)$ is high, it is more likely to be chosen by our current policy π_i. We can use this intuition to formulate the following table:

cost↓/score→	Low	High
Low	a is unlikely but is near-optimal $(score - L \approx 0)$	a is likely and near-optimal $(score - L \gg 1)$
High	a is unlikely and far from optimal $(score - L \ll 1)$	a is likely and far from optimal $(score - L \approx 0)$

as we wish to maximize the difference $(score - L)$, we tend towards a likely action that is also optimal, and we control λ_i to make sure that the actions are likely enough to be chosen.

When adding this functionality to our dagger implementation (alg. 13), we only need to change the tagging of π^* to $\tilde{\pi}$

6. Multi-Arm Bandit

We'd like to cover the concept of RL in the most simplified setting, which does not involve learning to act in more than one situation and avoids some complexities introduced in other RL problems.

Imagine a row of K slot machines ("arms) with unknown and variable payoffs. A player must choose which machines to play given a finite time horizon H to maximize his profits. Our player must balance *exploration* and *exploitation*, he should also understand the behavior of multiple arms, but also focus on those that provide a better reward (or minimal regret).

> **R** Before moving on, We define the expected cumulative regret $\mathbb{E}[Reg_n]$ to be the difference between the optimal expected cumulative reward and the expected cumulative reward of our strategy at time n. If the optimal reward at every time step is R^*, after n steps we can write

$$\mathbb{E}[Reg_n] = nR^* - \sum_{i=1}^{n} \mathbb{E}[r_i] \tag{6.1}$$

6.1 Basic bandit algorithms

In the most basic Bandit algorithm, we initialize the value of an action $Q(a)$ and the number of times we performed an action $N(a)$, and repeat the following process:

- choose an ε greedy action (random or $\mathrm{argmax}_a Q(a)$)
- obtain a reward R by taking an action a (pulling the a'th bandit's lever, if you will)
- update $N(a) \leftarrow N(a) = 1$
- update the value $Q(a) \leftarrow Q(a) + \frac{1}{N(a)}[R - Q(a)]$

The update rule says that the expected reward for action a should be updated to be a weighted average of the previously expected reward and the newly observed reward. The weight given to the new observed reward is inversely proportional to the number of times action a has been taken so far. In other words, the more times action a has been taken, the less weight is given to the new observed reward. This encourages the exploration of other actions. Notice also how the update rule resembles gradient ascend - we take in the old estimate for the value and update it w.r.t to the difference from

the target reward, up to some step size.

We can also weigh the update rule with some generic step size term α, to add more versatility.

We can improve on the naive action exploration by exploring actions according to their potential of actually being optimal. formally we can define

$$a_t = \underset{a \in A}{\operatorname{argmax}} \left[Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right] \qquad (6.2)$$

This means that we not only choose the greedy action based on the action that maximizes $Q_t(a)$ for all $a \in A$, but also based on the visit rate - here like the above, if $N_t(a)$ increases, we are less likely to choose the action. we also add a $\sqrt{\ln t}$ term, to indicate a measurement of time increment. If $N_t(a)$ is large but also $\ln t$ is, the term will be not as small, indicating that it is reasonable to choose an action many times if much time has passed since the beginning.

eq. 6.2 helps us replace the ε-greedy random choice, that decouples the exploration and exploitation with one single term that governs both in a more elegant way.

> **R** The above can then be reformulated to an algorithm called *The Upper Confidence Bound* (UCB), named after the fact that the expected number of pulls that was required to achieve an optimal policy was bounded from above

Some problems with UCB and other stationary algorithms:
- Non-stationary problems, where the expected return of every arm can change over time, are not supported as we can't really rely on what had already happened.
- if the state space is too large, it is unfeasible to learn.

6.2 Advanced bandits algorithm

Let's briefly go over some of the more advanced methods

6.2.1 Gradient bandit algorithms

Again we come to the conclusion that estimating action values to select ones is a good approach to support large state/action spaces and generalize to more problem setups.

Let us define the numerical preference for action a at time t as $H_t(a)$, where large $H_t(a)$ means that a is more likely to be taken. Our goal would be to learn $H_t(a)$ using iterative gradient descend, where

$$H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) \qquad (6.3)$$

with $H_0(a) = 0$, and

$$\pi_t(a) = Pr[a_t = a] = softmax(H_t(a)) = \frac{\exp(H_t(a))}{\sum_{a_i \in A} \exp(H_t(a_i))} \qquad (6.4)$$

6.2.2 Contextual Bandits

We would like to take into account the context in which the action sampling is being made. By doing so, we are essentially modeling the state in our environment, thus getting closer to the initial setting of reinforcement learning. we can write a schematic iterative process as follows - in round t:

- Observe user u_t and a set A of arms, alongside their features (context) $x_{t,a}$
- based on reward from previous iterations, choose an arm $a \in A$ and receive a reward $r_{t,a}$
- improve the arm selection strategy with each observation of $\langle x_{t,a}, a, r_{t,a} \rangle$. Notice here that different rewards from the same arm are possible, as many contexts are available.

For example, if we were to implement a recommendation system, we could ideally identify the context of our users (some tabular traits for example) and use those to provide a selection that is more specific.

In some cases, we can model the relation between the reward and the context in a linear fashion, and in *The Linear UCB algorithm*m they did just that. Let us define the expected reward conditioned on the context as

$$\mathbb{E}[r_{t,a}|x_{t,a}] = (x_{t,a})^T \theta^* \tag{6.5}$$

where θ^* is the unknown coefficient vector that we aim to learn. Our goal is to minimize the regret, which can be defined as the expected difference between the observed reward for the best arms and the expected reward for the selected arms

$$R_t(T) = \mathbb{E}\left[\sum_{t=1}^{T} r_{t,a_t^*} - r_{t,a_t}\right] \tag{6.6}$$

where $a_t^* = \text{argmax}_{a \in A} x_{t,a}^T \theta^*$ is the best action at step t according to θ^*, and a_t is the action selected at step t. We then minimize the regret using linear regression minimization (the math formalism is discarded here)

6.2.3 Thompson Sampling

We assume that each arm's reward is sampled from a constant unknown distribution. Over time, we find an estimation for those distributions by sampling the arms.

let's say that the agent is trying to choose between K different actions, and let's call the probability that action k is the best action p_k. At each step, the agent will sample a value from the distribution for each action and choose the action with the highest sample. The distribution for each action is updated using Bayes' rule, which states that the posterior probability (the updated probability after observing new data) is equal to the prior probability (the initial probability before observing new data) times the likelihood (the probability of observing the new data given the prior probability). In this case, the prior probability is the current estimate for the probability that action k is the best action, and the likelihood is the probability of observing the reward that the agent received for taking action k. So the updated probability for action k after observing a reward r is:

$$p_k = p_k \cdot P(r|p_k) \tag{6.7}$$

This process is repeated for each action at each step, with the distribution for each action being updated based on the rewards that the agent receives. The advantage of using this approach is that it allows the agent to balance exploration and exploitation, as it will choose actions that have a high probability of being the best action based on the current information, while also trying out other actions to learn more about them and improve its estimates.

More specifically, in the algorithm, they've assumed beta distribution in the following manner
- for each arm $i = 1, 2, ..., N$ set $S_i = 0, F_i = 0$ (we think of S_i as #*Success* and F_i as #*Failures*)
- for every time $t = 1, 2, ...$

- for each arm $i = 1, 2, ..., N$: sample from $\beta(S_i + 1, F_i + 1)$ and play arm $i(t) = \text{argmax}_i \theta_i(t)$ to observe reward r_t.
- if $r = 1$ then $S_{i(t)} += 1$, else $F_{i(t)} += 1$

R   as a reminder, we write the relation between the posterior and the prior based on the Bayes rule

$$posterior = \frac{likelihood \times prior}{evidence} \longleftrightarrow P(\theta|X) = \frac{P(X|\theta)P(\theta)}{\sum_\theta P(X|\theta)} \tag{6.8}$$

7. Advanced RL use case - AlphaGo

Using some of the concepts from previous chapters, we will explore the story of AlphaGo, a groundbreaking artificial intelligence program developed by Google Deep Mind that made history in 2016 when it defeated Lee Sedol, a world champion Go player, in a highly publicized five-game match. The victory was significant because Go is a complex and challenging board game that has long been considered a pinnacle of the human intellect, and it was the first time that a computer program had been able to defeat a human professional player in the game. In this chapter, we will delve deeper into the techniques and strategies used by AlphaGo to achieve this impressive feat.

Before we can fully understand how AlphaGo was able to defeat a human champion at Go, it is important to first understand the concept of Monte Carlo tree search, which was a key component of the program's strategy. In the following sections, we will discuss the basics of Monte Carlo tree search and how it was used by AlphaGo to make informed decisions on the Go board.

7.1 Monte Carlo Tree Search (MCTS)

MCTS combines both sampling and tree search. We use the vertices of a tree to represent states, and the edges to represent actions that take us to the next state. The choice of action at any given vertices is governed by the tree's policy π. At any given iteration, if a leaf node was reached, we expand the tree by sampling the environment. Formally, we can write MCTS in 4 steps:

- Selection: we navigate to a leaf using the upper confidence bound (UCB) formula, (similar to eq 6.2): $\pi_{UCB}(s) = \text{argmax}_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$ where $Q(s,a)$ is the value of the state-action pair, $n(s)$ is the number of times a state s was visited, $n(s,a)$ is the number of times a state-action pair (s,a) was visited and c is a calibration parameter. Remember that the first term is an exploitation term, whereas the other term is an exploration term.
- expansion: if a leaf was reached, we either reached the end of the game (and then jump to back-propagation) or were still mid-game, but in an unknown situation. In the latter case, we use another policy called the *rollout* policy (which is a simpler policy) to choose which note to move to, and add that note to our tree.

- simulation (play-out): one simulated game is played from the leaf node reached in the previous step, and a reward is observed at the end of the game. The actions during the simulation step are either randomly chosen or given using the rollout-policy
- back-up: the results of the game are back-propagated in the tree, updating the value of $Q(s,a)$ for each of the nodes that participated in the game.

We emphasize the MCTS is only aware of the "rules" of the game, and hence might be outperformed by other approaches that use tailor-made heuristics. Furthermore, the algorithm can be halted at any given time, therefore training it for a fixed time is plausible.
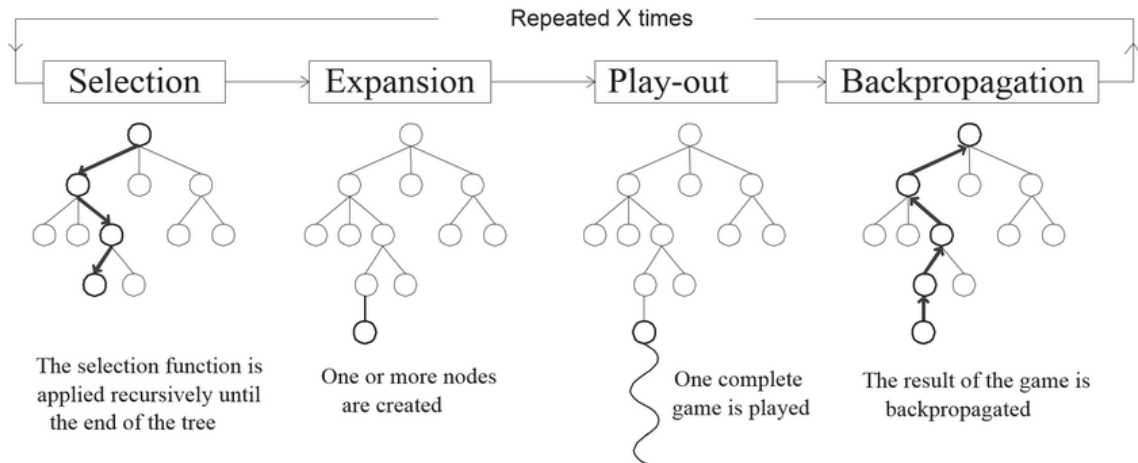

Figure 7.1: MCTS Diagram

Some improvements were suggested over the years, to enhance the basic functionality of MCTS. some of which are:

- Pruning policy - once an expanded node has had a sufficient number of simulations, we use a hand-crafted policy to determine whether it should remain the tree or be removed
- Improving the Value function - we define the value function as an MLP layer $Q(s,a) = \sigma(\phi(s,a)^T \theta)$ where ϕ are binary features, θ are the weights and σ is some activation probability function like *softmax*. Furthermore, instead of randomly sampling a state in the expansion step, a few policies were tested
  - ε-greedy one
  - greedy policy with noisy value function $\pi(s,a) = 1 \iff a = \underset{a' \in A}{\operatorname{argmax}} [Q(s,a') + \eta(s,a')]$ (otherwise $\pi(s,a) = 0$))
  - a smoothed softmax $\pi_\tau(s,a) = softmax(Q(s,a)/\tau)$

7.2   RL for the game of *Go*

Go is a *perfect information* game, meaning that all the information is given at any time step (unlike poker, for example). The optimal value function can be computed recursively on a search tree of b^d sequences, where b is the number of legal moves possible in each turn and d is the length of the game

(R) In chess, for example, $b \approx 35$ and $d \approx 80$. In a game of Go, on the other hand, $b \approx 250$ and $d \approx 150$

7.2.1 Alpha-Go

check out **this** blog post for more information.

The AlphaGo architecture consists of three networks: supervised learning policy network, RL policy network, and RL value network. In addition, the linear softmax model was used as a rollout policy. The state consisted of a few parameters such as stone color, turn, viable stone positions, etc. To make our arguments cleaner, the parameterized policy that outputs an action given a state as $p_\theta(a|s)$

The supervised learning policy network

the goal - recommend good moves by predicting those performed by Go grans-masters (similar to imitation learning). The policy network was trained on $\sim 30M$ positions from $\sim 160K$ games, and the data was augmented by rotating the board. The goal function was maximizing the log-likelihood function of taking the "human" action. More formally, the step we take is

$$\propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} \tag{7.1}$$

where m is a batch size, α is a step size, θ_{SL} are the network parameters and $\log p_\theta[a_k|s_k]$ is the log probability of taking an expert action a_k given the state s_k

The rollout policy

During game simulation (discussed later), we need a fast approach to narrow down the moves options. Therefore, we create a rollout policy π which is a simple linear softmax classifier. This policy is trained exactly like the linear model one, but as it is simpler it is much faster.

The RL policy network

In the next phase, we play the game of Go with the current policy network itself, to improve the overall results. To do so, we start off with a duplicate of the SL policy network and call it the RL policy network p_ρ (with parameters ρ). We use REINFORCE with a baseline to improve p_ρ iteratively. The opponent p_ρ plays against is a previous version of p_ρ itself (chosen randomly), and the game goes on until it is finished. We denote the output of the game at time t as z_t, where $z_t = 1$ if p_ρ won and -1 otherwise. The update rule for p_ρ is

$$\propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t \tag{7.2}$$

where notice that the direction we move is governed by $sign(z_t)$

The RL value network

In the last stage, we want to add the capabilities of the grand-master board position evaluation, so we train a deep network to estimate the value of the current position (which is 1 if that position can be translated to a win and -1 otherwise), and call it v_π. To train the value network, which has similar architecture as the RL policy one except that it outputs a single value, we again perform self-play using the RL policy network. we compute the MSE w.r.t to the actual game outcome, meaning that the update rule is

$$\propto \frac{\partial \log v_\theta(s)}{\partial \theta} (z - v_\theta(s)) \tag{7.3}$$

Throughout the games, we collect board positions. More specifically, one board position is collected for every game (as all board positions in a game lead to the same result - win or lose) as different positions of the same game are highly correlated.

Our last goal is to use the policy network and the value network to complement each other

MCTS

We'd like to search for actions that translate to as many wins as possible, and we do so by examining the four steps of MCTS:

- Selection: remember that we aim to choose actions that are also greedy but also explore. in terms of the exploitation part, we take advantage of our value function network and define

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbb{1}(s,a,i) V(s_i) \tag{7.4}$$

  with $V(s_i) = (1-\lambda)v_\theta(s_i) + \lambda z_i$. In words - we set the value of a state action pair as the weighted sum (in terms of #visits) of the actual value we give to the state and whether that state is translated to a win condition. We use a convex sum to account for the two factors combined. In terms of the exploration part, we define

$$u(s,a) = \frac{p_\sigma(a|s)}{1+N(s,a)} \tag{7.5}$$

  meaning that we normalize "how good it is to take action a" by the number of visits, as it makes visited actions less likely to be chosen (which is the meaning of exploration)
  Finally, we aggregate the two and choose the action

$$a_t = \operatorname*{argmax}_a [Q(s_t,a) + u(s_t,a)] \tag{7.6}$$

- Expansion: we add more positions into the tree to reflect what moves we have tried. every new node is initialized with a predefined value of $N(s,a) = Q(s,a) = 0$, an associated probability $p_\sigma(a|s)$ (set by the SL policy network), and a value $v_\pi(s)$ (set by the RL value network)
- Simulation: we simulate the rest of the game using MC rollout starting from the current leaf node. more formally, we sample action from our rollout policy $a \sim p_\pi$. recall that p_π is very fast, as many game roll-outs are necessary.
- back-up: after the roll-out we know if our game resulted in a win or a loss, so we can compute Q. over time, our Q estimate will be good enough to choose a good action

7.2.2 Alpha-Zero

for more info, see HERE
In the next generation of AlphaGo, we only use self-play to learn, taking into consideration nothing but the rules of the game (no expert demonstrations). Furthermore, the state's representation is only the stones on the board (with some history saved as well). Lastly, only one neural net was used. We start off with a high level description:

- Self-Play: we create a training set by using self-play, where in each move the game state, the search probabilities (from MCTS) and the winner are saved.
- Network optimization: sample a mini batch from the training set (of the previous step) and train the current network on these board positions. the loss function has two terms

- the value function (how probable is winning from the given board states), that is compared to the actual win condition (win or lose) using MSE
- the action probabilities for each legal state

More specifically, we write that

$$\ell = \sum_t (v_\theta(s_t) - z_t)^2 - \pi_t \log(p_\theta(s_t)) \tag{7.7}$$

and in some cases, a regularization term, $\lambda||\theta||$, was added to normalize the weights
- evaluate network: play 400 games between the latest neural network and the current best neural network, where both networks use MCTS to select their moves. the network who wins 55% or more is declared the new best network.

Similar to AlphaGo, each node contains a $V(s)$, that represents how likely the player to win from the current state, and each edge contains the action value $Q(s,a)$, the visit count $N(s,a)$ and the probability to visit $P(s,a)$.

Next, we describe the four steps of MCTS

- Selection: again the exploitation term is $Q(s,a)$, but the exploration is $c \cdot P(s,a) \frac{\sqrt{\sum_{a' \in A} N(s,a')}}{1+N(s,a)}$, which is similar to what we had in Alpha-Go, up to the factorization of all possible actions $\sqrt{\sum_{a' \in A} N(s,a')}$
- Expansion + simulation: when a leaf is reached, all possible states are initialized. Then, a single node is expanded and evaluated using $Q(s,a)$. For this node we also calculate $V(s)$ and *immediately* return, that is no rollout is being performed.
- backup - traverse up the tree, update the values $N+=1, V+=v, Q=V/N$

after around ~ 1600 simulations, we select a move. In the case of the test phase, we chose the node for which N is largest. for the training phase, we choose

$$\pi(a|s_0) = \frac{N(s_0,a)^{1/\tau}}{\sum_{a' \in A} N(s_0,a')^{1/\tau}} \tag{7.8}$$

7.2.3 Alpha-Zero in other domains

Some later projects attempted to apply Alpha-Zero's framework to the field of automatic pipeline generation, the prominent one was Alpha-D3M. More specifically, instead of Go/Chess pieces, the basic unit was some pipeline primitive (say, random forest, neural net, etc...), the state was some metadata associated with the task, and the task itself, the actions were replacing pipelines, adding more aggregated ones, etc, and the reward was the pipeline performance. To encode a pipeline we may perform the following abstract steps:
- encode dataset D_i as metadata features $f(D_i)$
- encode the task T_j
- encode the current pipeline at time t by a vector S_t (S_t is a pipeline sequence)
- encode action $f_a(S_t)$ so policy π maps $(f(D_i), T_j, S_t) \to (f_a(S_1), ..., f_a(S_n))$

The general learning scheme is similar to Alpha-Zero, where the action choice is given using the same exploitation + exploration term, and the loss function is a sum of both a cross-entropy term for the probability function and an MSE term for the value function (with some additional regularization). Another line of research is known as *Neural architecture search*, in which the idea is to automatically learn a more efficient network architecture representation. One approach to do so is to learn a string representation of the network's architecture and train an RNN to generate the next architecture block

sequence. In the original paper, they defined the reward as the accuracy result over the validation set, and user REINFORCE to optimize the loss term.

8. Meta and Transfer Learning

In the following chapter, we distinguish between two similar yet different learning tasks
- Meta-learning (sometimes refers to as "*Learning to learn*"): is the process of learning how to model our problem and use the generalized result over multiple sets of tasks with similar setups. For example - we train a robotic arm with two joints to do an arbitrary task and then use the same agent on a different robotic arm with three joints.
- Transfer learning: is the process of learning some knowledge from one task and using it to improve the performance of a model on a different but related task. For example - we can use a pre-trained ResNet to achieve better image classifiers.

the main difference between meta-learning and transfer learning is that meta-learning focuses on learning generalizable knowledge that can be applied to a wide range of tasks, while transfer learning focuses on transferring knowledge from one specific task to another related task.

8.1 Meta-Learning

Let us define a set of tasks $\{\tau_1, ..., \tau_n\}$ where each τ_i is episodic (with length H_i) and defined by a set of states $\{s_t^i\}_{t=0}^{H_i}$, actions $\{a_t^i\}_{t=0}^{H_i}$, a loss L_i, and a transition distribution P_i. A meta-learner with parameters θ models the distribution $\pi(a_t|s_1, ..., s_t; \theta)$ with the objective of minimizing the expected loss over all tasks

$$\min_{\theta} \mathbb{E}_{\tau_i} \left[\sum_{t=0}^{H_i} L(s_t, a_t) \right] \tag{8.1}$$

where $s_t \sim P_i(s_t|s_{t-1}, a_{t-1})$, and $a_t \sim \pi(a_t|s_1, ..., s_t; \theta)$

There are a few desired properties of a Meta-RL algorithm:
- consistency: the ability to continually improve as we get more data
- expressive: the ability to represent multiple types of tasks
- Structured exploration: the ability to explore the problem space efficiently, utilizing as few samples as possible
- Efficient and off-policy: this way we can run the Meta algorithm on real-world problems

8.1.1 Memory-Augmented Networks

(Based on the paper *Meta-Learning with Memory-Augmented Neural Networks*)

In some cases, we wish to perform meta-learning when we are only provided with small (or no) data. For example - in one/few-shot learning we are expected to provide a label with only a few tagged samples. To do so, we use two techniques

- External memory module - we initialize a container of knowledge that is called upon to respond to challenging circumstances
- Label shuffling - labels are presented one time-step after their corresponding sample, which helps with dealing with simply learning the mapping *sample* → *label* without generalizing. More specifically, we feed the network with $(x_1, null), (x_2, y_1), ..., (x_{t+1}, y_t)$

In addition, samples are shuffled across different datasets (samples from different datasets may appear in the same sequence). This encourages the network to use the memory module and extract the relevant label once the corresponding sample is provided.

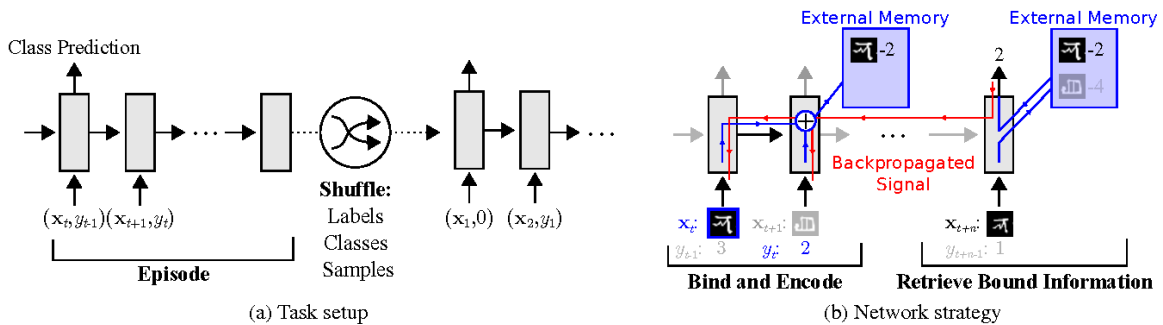(a) Task setup   (b) Network strategy

Figure 8.1: meta-learning with memory-augmented network diagram. Left (a): the (lagged) episodes from various datasets are shuffled. Right (b): during the learning process, the sample is first saved to the external memory and later retrieved when the relevant label is presented

We can also represent the pipeline using the following block diagram
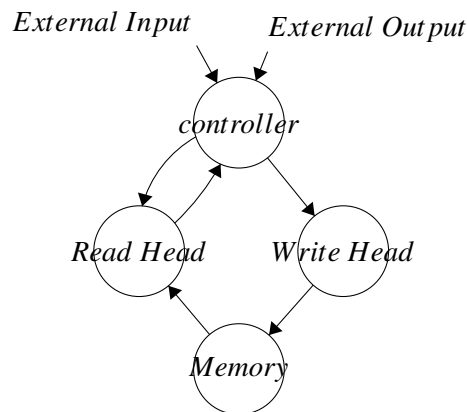
Our main question would be - which memories should we read? We use a similarity measure to generate a weights vector - given an input x_t, the controller (the network) produces a key k_t which is then stored in a row of a matrix M_t or used to retrieve the particular memory i from the i'th row

$M_t(i)$. When retrieving a memory, we use cosine similarity

$$K(k_t, M_t(i)) = \frac{k_t \cdot M_t(i)}{||k_t|| \cdot ||M_t(i)||} \tag{8.2}$$

Meaning that we compare the current key k_t with another key described in the i'th row of M_t. We use The similarity between all keys to produce the *read-weight vector* (with superscript r)

$$w_t^r(i) = softmax(K(k_t, M_t(i)))) = \frac{\exp\left(K(k_t, M_t(i))\right)}{\sum_j \exp\left(K(k_t, M_t(j))\right)} \tag{8.3}$$

$w_t^r(i)$ is stored in memory, from which the read value is retrieved as

$$r_t = \sum_i w_t^r(i) M_t(i) \tag{8.4}$$

Over time, new information is written into rarely-used locations, preserving recently encoded information, or it is written to the last used location, which can function as an update of the memory with newer, possibly more relevant information. The distinction between these two options is accomplished with interpolation between the previous read weights and weights scaled according to usage weights w_t^u. Those are updated as

$$w_t^u \leftarrow \gamma w_{t-1}^u + w_t^r + w_t^w \tag{8.5}$$

where γ is a decaying parameter, w_t^r is computed as in eq. 8.3, and w_t^w will be defined later. The least-used weights $w_t^{lu}(i)$ is then given as 0 if $w_t^u(i) > m(w_t^u, n)$ or 1 otherwise, where $m(w_t^u, n)$ denotes the n'th smallest item in w_t^u (and we set n to the number of read to memory). This definition allows us to recursively define the write weights

$$w_t^w \leftarrow \sigma(\alpha) w_{t-1}^r + (1 - \sigma(\alpha)) w_{t-1}^{lu} \tag{8.6}$$

where σ is the sigmoid function and α is a hyper-parameter.
Eventually, we update the memory using

$$M_t(i) \leftarrow M_{t-1}(i) + w_t^w(i) k_t \tag{8.7}$$

for all i

(R)   we did not specify how to formulate the above as an RL task

8.1.2 Simple Neural Attentive Meta-Learner

The previous approach struggled to identify historic events that are most relevant to the current timestamp, as it was hard-coded to only account for one step difference. In a newer version, attention layers and dilated temporal convolutions allowed just that.

Dilated + Causal convolutions

Instead of a regular $k \times k$ kernel, we use some form of inherent stride, where each convolution pixel is distant from its neighbors, hence increasing the filter's receptive field while using the same number of parameters as the original Conv layer.
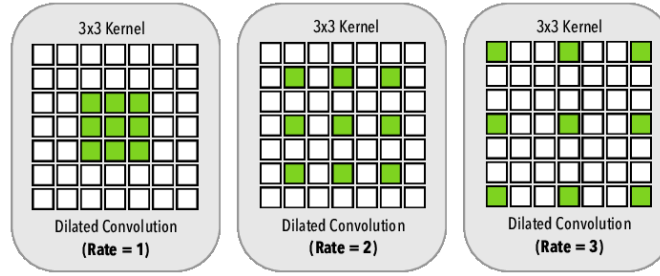
Figure 8.2: Dilated Conv layer for various rates (strides)

Attention mechanism (in brief)

Attention mechanisms are a key component in many natural language processing and machine learning models. They allow the model to focus on specific parts of the input, rather than processing the entire input equally. Mathematically, an attention mechanism can be represented as a function $\text{Attention}(Q, K, V)$, where Q, K, and V are matrices representing the query, key, and value, respectively. The attention function returns a weighted sum of the values, where the weight for each value is determined by the dot product of the corresponding query and key.

$$\text{Attention}(Q, K, V) = \sum_{i=1}^{n} \frac{\text{dot}(Q, K_i)}{\sqrt{d_k}} V_i \tag{8.8}$$

Here, n is the number of values, d_k is the dimension of the keys, and $\text{dot}(Q, K_i)$ is the dot product of the query and the ith key. The division by $\sqrt{d_k}$ is included to scale the dot product, as it can become large for large values of d_k.

To add an attention mechanism to an LSTM, for example, we can modify the computation of the output at each time step to incorporate the attention weights. Let's consider an LSTM with outputs $o_1, o_2, \ldots, o_T$, where T is the number of time steps. The output at each time step is typically computed as a function of the previous output, the current input, and the current hidden state:

$$o_t = g(o_{t-1}, x_t, h_t)$$

To incorporate an attention mechanism, we can modify this equation to include the attention weights:

$$o_t = g(o_{t-1}, x_t, h_t, \text{Attention}(o_{t-1}, K, V))$$

Here, $\text{Attention}(o_{t-1}, K, V)$ is the attention function described in a previous answer, with Q set to the previous output o_{t-1}. The matrices K and V are typically fixed and are learned during training.

This modified equation allows the output at each time step to be a weighted sum of the previous output, the current input, and the current hidden state, with the weights determined by the attention mechanism. This allows the model to focus on specific parts of the input when computing the output, rather than processing the entire input equally.

It's also possible to use the attention mechanism in conjunction with the hidden state of the LSTM, rather than the output. In this case, the attention function would be applied to the hidden state at each time step, rather than the output.

Furthermore, we use causal attention that makes sure no future input is used when calculating the output of the current state (to avoid data leakage). This means that every output is generated by looking over only previous samples.

The architecture

we use two blocks of temporal convolution layers (orange) that are interleaved with two causal attention layers (green). In reinforcement-learning settings, it receives a sequence of observation-action-reward tuples $(o_1, null, null), ..., (o_t, a_{t-1}, r_{t-1})$. At each time t, it outputs a distribution over actions a_t based on the current observation o_t as well as previous observations, actions, and rewards. Furthermore, the internal state across episode boundaries is preserved, which allows it to have a memory that spans multiple episodes. The observations also contain a binary input that indicates episode termination.
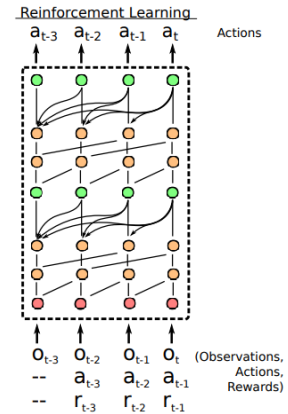
SNAIL achieves state-of-the-art performance by significant margins on all of the most widely bench-marked meta-learning tasks in both supervised and reinforcement learning, without relying on any application-specific architectural components or algorithmic priors.

8.1.3 Model Agnostic Meta-Learning

If we write a generic learning process as $\theta \leftarrow \theta - \alpha \nabla_\theta L_{train}(\theta)$, a generalized approach over many tasks minimizes the objective $\sum_{task_i} \sum_i (\theta - \alpha \nabla_\theta L^i_{train}(\theta))$. Intuitively, we take a step in the *averaged* direction based on the gradients of all tasks' losses.
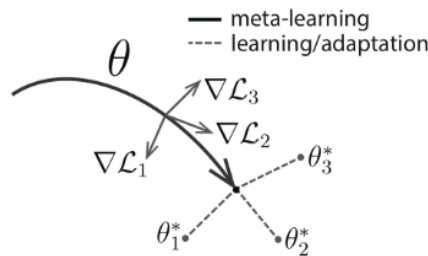
Figure 8.3: MAML - update θ w.r.t to the expected direction induced by the losses of all tasks

Under the scope of RL, we will sample a batch of tasks, and for each task sample a set of trajectories from our environment. once a trajectory was sampled, a loss gradient will be computed based on the expected episodic return, followed by an update rule for the specific tasks' parameters. once all tasks were iterated through, the final update rule will be performed w.r.t the mean loss values over all tasks. We can compare MAML and SNAIL in terms of their properties:

	SNAIL	MAML
consistent	As a heavy-duty model, less likely to improve with only new data	easily adjusted given new data as it is only gradient-based
expressive	uses memory therefore can obtain deeper understanding of tasks	has no memory, so is generally less expressive
structured exploration	does not enforce a smart exploration scheme, though still not very inefficient	same as SNAIL
efficient and off-policy	is on policy	same as SNAIL

(R) [Adaptations in real-life] We briefly discussed Adaptations in real-life, which is a meta-learning process of adapting an agent to environmental changes, usually encountered in real-life problems. To adapt our model to real life, we first make the assumption that any new time step is potentially a new environment, with different dynamics though a common structure. For each of the environments, we learn a trajectory that lasts as far as the environment does and store the recent history in a database. The adaptation takes hold in a cyclic fashion - a model is learned for the current environment, and once a new environment was encountered, it is either acted upon using experience stored in the database or adapted to and then stored.

8.2 Transfer Learning

TL is, again, a process where a model trained on one problem/dataset is applied to another. TL is mostly used when multiple models are harder to train and when we'd like to leverage knowledge across problems or domains. Few options are common

- Classic approach: take a trained model from one task and re-train it on another task. Another possibility is to freeze some of the weights.
- Diversity learning: pre-train the model for diversity by learning various solutions to a similar task

8.2.1 Training the model for diversity

A key point in diversity learning is the fact that a stochastic policy is required, as a deterministic one cannot easily consider multiple possible action choices.