Reinforcement learning fundamentals

Petar Veličković

Computer Laboratory, University of Cambridge, UK
Nokia Bell Labs, Cambridge, UK

About me

- Research Assistant in Computational Biology/PhD student in the Artificial Intelligence Group of the University of Cambridge's Computer Laboratory;
- Industrial/research experience with *Microsoft*, *Jane Street* and *Nokia Bell Labs*.
- Interested in integrating machine learning techniques with complex networks, particularly in low-data environments—one of which is (early-stage) reinforcement learning.

Introduction

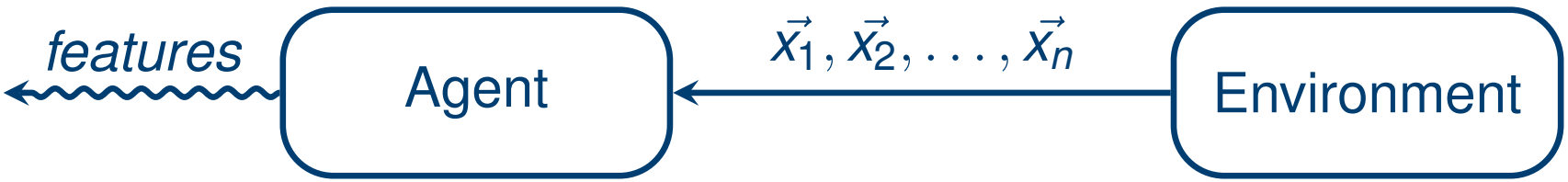
- Contrary to my previous talks on the topic, this talk will introduce reinforcement learning from essential first concepts.
- Assuming one is familiar with reinforcement learning and deep neural networks, simply plugging in a neural network at the correct moment in the pipeline. . .
- . . . *and deriving a "proper" supervision signal for it. . .*
- . . . is sufficient for making the leap to deep RL!

The three “flavours” of machine learning

- Unsupervised learning
- Supervised learning
- Reinforcement learning

Unsupervised learning

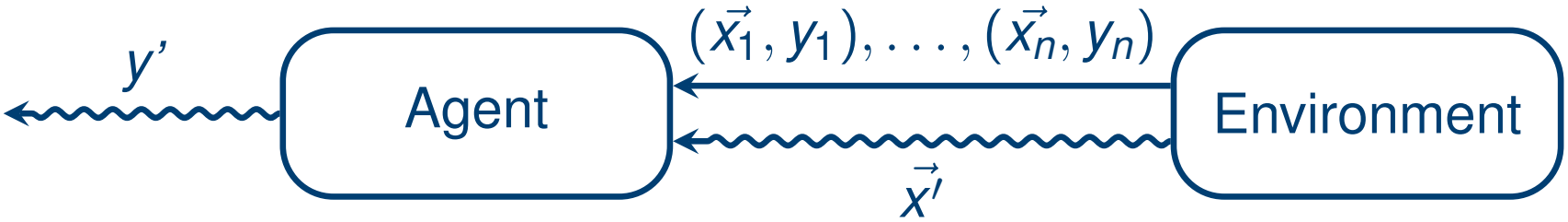
- The environment gives you *unlabelled data*—and asks you to assign useful features/structure to it.



- Example: study data from patients suffering from a disease, in order to discover different (previously unknown) types of it.

Supervised learning

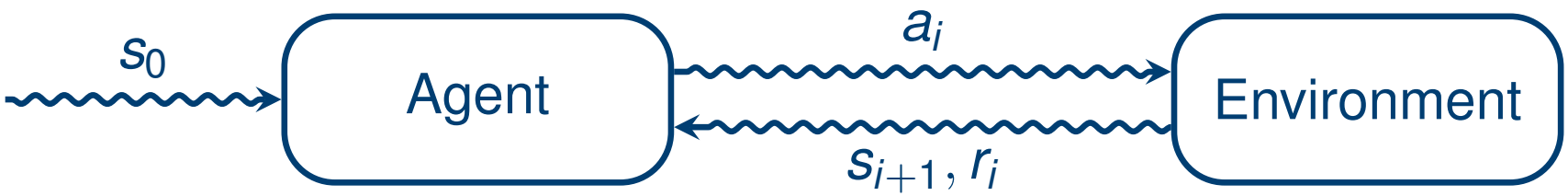
- The environment gives you *labelled data* ($\sim$ known *input/output pairs*)—and asks you to learn the underlying function.



- Example: determining whether a person will be likely to return their loan, given their credit history (and a set of previous data on issued loans to other customers).

Reinforcement learning

- This time, you are allowed to perform *actions* within the environment, triggering a state change and a reward signal—your objective is to maximise future rewards.



- Example: playing a video game—*states* correspond to RAM/framebuffer contents, *actions* are available key presses (including NOP), *rewards* are changes in score.

Markov Decision Processes (MDPs)

Problem

- States, $s \in \mathcal{S}$, and actions, $a \in \mathcal{A}(s)$.
- Transition model, $\mathcal{T}(s, a, s') \sim \mathbb{P}(s'|s, a)$.
- Reward model, $\mathcal{R}(s)/\mathcal{R}(s, a)/\mathcal{R}(s, a, s') \in \mathbb{R} \longrightarrow$ *equivalent*.

Solution

- Policy, $\pi(s) = a \in \mathcal{A}(s)$, learnt from observed (s, a, s', r) tuples.

Properties of MDPs

+ Markov property (can encode all past states within members of $\mathcal{S}$, if necessary to make this work)
+ Stationary (model does not change with time)
– Delayed rewards (lack of *immediate feedback*)
– Minor parameter changes may have a significant influence on the optimal policies!
? Temporal credit assignment problem: determining which actions in a sequence were *most responsible* for the observed reward sequence.

Assumptions

- Infinite horizon ($\sim$ can *live forever*).
  - for *finite horizons*, the policy function also takes into account the **remaining time**, i.e. $\pi(s, \underline{t})$.
- Sequence utilities:

$$\begin{aligned} & \mathcal{V}(s_0, s_1, \ldots, s_n, \ldots) > \mathcal{V}(s_0, s'_1, \ldots, s'_n, \ldots) \\ \Longrightarrow\ & \mathcal{V}(s_1, s_2, \ldots, s_n, \ldots) > \mathcal{V}(s'_1, s'_2, \ldots, s'_n, \ldots) \end{aligned}$$

- Defining utilities in the naïve way

$$\mathcal{V}(s_0, \ldots s_n, \ldots) = \sum_{t=0}^{+\infty} \mathcal{R}(s_t)$$

does not work for infinite horizons! (why?)

Discounted cumulative reward

- To remedy the issue, we introduce a *discount factor*, $\gamma \in [0, 1)$, which scales all future rewards:

$$\mathcal{V}(s_0, \dots s_n, \dots) = \sum_{t=0}^{+\infty} \gamma^t \mathcal{R}(s_t)$$

  giving rise to the *discounted cumulative reward*, which is the typical metric to optimise in an RL setting.
- Assuming rewards are bounded by $\mathcal{R}_{\text{max}}$, we can easily show that this metric fixes the previous issue:

$$\sum_{t=0}^{+\infty} \gamma^t \mathcal{R}(s_t) \leq \sum_{t=0}^{+\infty} \gamma^t \mathcal{R}_{\text{max}} = \frac{\mathcal{R}_{\text{max}}}{1 - \gamma}$$

Optimal policy

- Our agent seeks to learn an *optimal policy*:

$$\pi^* = \underset{\pi}{\text{argmax}}\, \mathbb{E}\left(\sum_{t=0}^{+\infty} \gamma^t \mathcal{R}(s_t) \middle| \pi \right)$$

- Define the *value* of a given state s under a policy π as:

$$\mathcal{V}^{\pi}(s) = \mathbb{E}\left(\sum_{t=0}^{+\infty} \gamma^t \mathcal{R}(s_t) \middle| \pi, s_0 = s \right)$$

- This allows us to re-express the optimal policy as:

$$\pi^*(s) = \underset{a}{\text{argmax}} \sum_{s'} \mathcal{T}(s, a, s') \mathcal{V}^{\pi^*}(s')$$

Bellman equation

- Setting $\mathcal{V}(s) \equiv \mathcal{V}^{\pi^*}(s)$, we arrive at:

$$\boxed{\mathcal{V}(s) = \mathcal{R}(s) + \gamma \max_a \sum_{s'} \mathcal{T}(s, a, s')\mathcal{V}(s')}$$

  an identity also known as the **Bellman equation.**

- Determining $\mathcal{V}(s)$ is sufficient for learning the optimal policy! However, the max operator makes the formula nonlinear, and therefore hard to *directly* solve.

Value iteration

- Starting with random initial values $\mathcal{V}_0(s)$, and updating until convergence as follows:

$$\mathcal{V}_{t+1}(s) = \mathcal{R}(s) + \gamma \max_a \sum_{s'} \mathcal{T}(s, a, s') \mathcal{V}_t(s')$$

  we arrive at the **value iteration** algorithm.

Aside: Policy iteration

- While not the preferred approach by deep RL, we may also consider *policy iteration*, where we optimise the policy directly:
  1. Start with a random-guess policy, π_0.
  2. Evaluate by computing $\mathcal{V}_t = \mathcal{V}^{\pi_t}$.
  3. Improve by computing $\pi_{t+1}(s) = \text{argmax}_a \sum_{s'} \mathcal{T}(s, a, s')\mathcal{V}_t(s')$.
  4. If not converged, return to step 2.

- Note that the computation rule for $\mathcal{V}_t$ is fully linear now:

$$\mathcal{V}_t(s) = \mathcal{R}(s) + \gamma \sum_{s'} \mathcal{T}(s, \pi_t(s), s')\mathcal{V}_t(s')$$

Hang on...

- I haven't been completely honest with you!
- In fact, recalling the obtained expression for π^*

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} \mathcal{T}(s, a, s')\mathcal{V}(s')$$

  we find that learning $\mathcal{V}(s)$ is *not sufficient* for deriving π^*—we need knowledge of the underlying transition model ($\mathcal{T}$) as well!
- Generally speaking, the agent *has no access to* $\mathcal{T}$!
- Did I say "Bellman *equation*"?

I meant **equations**!

Using $\mathcal{R}(s, a)$ for the reward model from now on:

- First Bellman Equation ("value"):

$$\mathcal{V}(s) = \max_a \left\{ \mathcal{R}(s, a) + \gamma \sum_{s'} \mathcal{T}(s, a, s') \mathcal{V}(s') \right\}$$

- Second Bellman Equation ("quality"):

$$\mathcal{Q}(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s'} \mathcal{T}(s, a, s') \max_{a'} \left\{ \mathcal{Q}(s', a') \right\}$$

- Third Bellman Equation ("continuation"):

$$\mathcal{C}(s, a) = \gamma \sum_{s'} \mathcal{T}(s, a, s') \max_{a'} \left\{ \mathcal{R}(s', a') + \mathcal{C}(s', a') \right\}$$

Significance?

- All three equations are *semantically equivalent*!
- $\mathcal{V}(s)$ rule very useful when we have (direct or inferred) knowledge of the $\mathcal{T}/\mathcal{R}$ functions.
- $\mathcal{Q}(s, a)$ rule does not require knowing the underlying model once the values are learned: the optimal policy is simply

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

  This procedure is often referred to as the *$\mathcal{Q}$-learning algorithm*.
- $\mathcal{C}(s, a)$ has the same property, but delays reward evaluation by one step—*useful when reward signals are difficult to compute*.

The three “flavours” of reinforcement learning

Three ways of learning from sequences $\left(s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots \right)$:

- Model-based

$$(s, a, r)^* \rightarrow \boxed{\textit{Learner}} \leftrightarrow \mathcal{T}/\mathcal{R} \rightarrow \boxed{\textit{Solver}} \rightarrow \mathcal{Q} \rightarrow \boxed{\text{argmax}} \rightarrow \pi$$

- Model-free

$$(s, a, r)^* \rightarrow \boxed{\textit{Value update}} \leftrightarrow \mathcal{Q} \rightarrow \boxed{\text{argmax}} \rightarrow \pi$$

- Policy search

$$(s, a, r)^* \rightarrow \boxed{\textit{Policy update}} \leftrightarrow \pi$$

Temporal difference

- We still need to derive an update rule, as the Bellman equation for $\mathcal{Q}(s, a)$ still requires knowledge of $\mathcal{T}$.

- A very popular approach is $TD(\lambda)$ (*temporal difference*) learning; I will focus on the special case of $TD(0)$, which is usually a good choice for real-time control—for more info, ref. Sutton, 1988.

$TD(0)$ update rule

- Key idea: after observing a transition $s \xrightarrow[r]{a} s'$, based on our current beliefs of the $\mathcal{Q}$ function, this transition has a value of $r + \gamma \max_{a'} \mathcal{Q}(s', a')$.
- Therefore we can update our belief of $\mathcal{Q}(s, a)$ based on this value, according to a learning rate $\alpha_t \in (0, 1]$, as follows:

$$\mathcal{Q}_{t+1}(s, a) = \mathcal{Q}_t(s, a) + \alpha_t \left(r + \gamma \max_{a'} \mathcal{Q}_t(s', a') - \mathcal{Q}_t(s, a) \right)$$

- **N.B.** we have now eliminated the requirement of knowing $\mathcal{T}/\mathcal{R}$ and are learning only from observed transitions in training.

Aside: learning rate and convergence properties

- To guarantee *convergence* of the $TD(0)$ algorithm, the following three properties have to hold (regardless of initialisation):
  1. $\sum_{t=0}^{+\infty} \alpha_t = +\infty$
  2. $\sum_{t=0}^{+\infty} \alpha_t^2 < +\infty$
  3. Each state/action pair must be visited *infinitely often*.
- The parameter α_t's purpose is taking account of *stochasticity* of the model—therefore avoiding full overwriting of the old values. It is commonly set to $\alpha_t = \frac{1}{t^p}$ for $\frac{1}{2} < p \leq 1$.
- However, if the environment is *deterministic* (next state is always the same for a given (s, a) pair)—it is optimal to set $\alpha_t = 1$ and fully overwrite. We will assume this hereinafter.

(Deterministic) $\mathcal{Q}$-learning

We have arrived at the familiar verison of the $\mathcal{Q}$-learning algorithm!

Initialise the $\mathcal{Q}'$ table with random values.

1. Choose an action a to perform in the current state, s.
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Update:
$$\mathcal{Q}'(s, a) \leftarrow \mathcal{R}(s, a) + \gamma \max_{\alpha} \left\{ \mathcal{Q}'(\mathcal{S}(s, a), \alpha) \right\}$$
5. If the next state is not terminal, go back to step 1.

Exploration vs. exploitation

- *How* to choose an action to perform?
- Ideally, would like to start off with a period of *exploring* the environment's characteristics, converging towards the policy that fully *exploits* the learnt $\mathcal{Q}$ values (greedy policy).
- A very active topic of research. Two common approaches:
  - *Softmax* ($\tau \to 0$):

$$\mathbb{P}(a|s) = \frac{\exp(\mathcal{Q}(s,a)/\tau)}{\sum_{\alpha} \exp(\mathcal{Q}(s,\alpha)/\tau)}$$

  - ε*-greedy* ($1 \to \varepsilon \to 0$):

$$\mathbb{P}(a|s) = \varepsilon \cdot \frac{1}{|A|} + (1-\varepsilon) \cdot \mathbb{I}(a = \operatorname*{argmax}_{\alpha} \mathcal{Q}(s,\alpha))$$

Issues with $\mathcal{Q}$-learning

- **Major issue**: *we often cannot store the entire $\mathcal{Q}$ table in memory!* (e.g. for video game-playing, consider the set of all possible input framebuffers $\sim$ matrices of pixels)
- It needs to be **approximated** somehow...
- By using a *deep neural network* as the approximator, and the most-recent belief of the $\mathcal{Q}$ value as its supervision signal, we arrive at the familiar **D$\mathcal{Q}$N** architecture.

Deep $\mathcal{Q}$-learning

Initialise an empty replay memory.
Initialise two D$\mathcal{Q}$Ns, $\mathcal{Q}'$ and $\mathcal{Q}''$, with random (small) weights.

1. Choose an action a to perform in the current state, s, using an ε-greedy strategy (with ε annealed from 1.0 to 0.1).
2. Perform a and receive reward $\mathcal{R}(s, a)$.
3. Observe the new state, $\mathcal{S}(s, a)$.
4. Add $(s, a, \mathcal{R}(s, a), \mathcal{S}(s, a))$ to the replay memory.

Deep $\mathcal{Q}$-learning, *cont'd*

5. Sample a minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from the replay memory, and perform stochastic gradient descent on the D$\mathcal{Q}$N $\mathcal{Q}'$, minimising the loss function

$$\left(\mathcal{Q}'(s_i, a) - \left(r_i + \gamma \max_{\alpha}\{\mathcal{Q}''(s_{i+1}, \alpha)\}\right)\right)^2$$

   where $\mathcal{Q}'(s, \cdot)$ and $\mathcal{Q}''(s, \cdot)$ are computed by feeding s into each of the respective D$\mathcal{Q}$Ns.
6. If the next state is not terminal, go back to step 1.
7. Occasionally set $\mathcal{Q}'' = \mathcal{Q}'$.

Concluding remarks

- It should be clear that, given that our supervision signal *likely does not represent ground truth*, this procedure does not preserve mathematical guarantees of the original $\mathcal{Q}$-learning algorithm.
- In fact, there are *absolutely no convergence guarantees* to this approach (or any of the improved approaches such as DDQN and Dueling DQN)!
- However, it still works quite well in practice (as you have doubtlessly already seen). :)

Thank you!

Questions?

petar.velickovic@cl.cam.ac.uk