Lecture 9: Reinforcement Learning (I)

In which we introduce the basics of reinforcement learning.

Reinforcement Learning (I)

Throughout this course, we have primarily focused on supervised learning (building a prediction function from labeled data), and briefly also discussed unsupervised learning (generative models and word embeddings). In both cases, we have assumed that the data to the machine learning algorithm is *static* and the learning is performed *offline*.

Neither assumption is true in the real world! The data that is available is often influenced by previous predictions that you have made. (Think, for example, of stock markets.) Moreover, data is continuously streaming in, so one needs to be able to adapt to uncertainties and unexpected pitfalls in a potentially adverse environment.

Applications that fall into this category include:

- AI for games (both computer/video games as well as IRL games such as Chess or Go)
- teaching robots how to autonomously move in their environment
- self-driving cars
- algorithmic trading in markets

among others.

This set of applications motivates a third mode of ML called *reinforcement learning* (RL). The field of RL is broad and we will only be able to scratch the surface. But several of the recent success stories in deep learning are rooted in advances in RL – the most high profile of them are Deepmind's AlphaGo and OpenAI's DOTA 2 AI, which were able to beat the world's best human players in Go and DOTA 2 respectively. These AI agents were able to learn winning strategies entirely automatically (albeit by leveraging massive amounts of training data; we will discuss this later.)

To understand the power of RL, consider – for a moment – how natural intelligence works. An infant presumably learns by continuously interacting with the world, trying out different actions in possibly chaotic environments, and observing outcomes. In this mode of learning, the input(s) to the learning module in the infant's brain is decidedly *dynamic*; learning has to be done *online*; and very often, the environment is *unknown* before hand.

For all these reasons, the traditional mode of un/supervised learning does not quite apply, and new ideas are needed.

A quick aside: the above questions are not new, and the formal study of these problems actually classical. The field of control theory is all about solving optimization problems of the above form.

But the approaches (and applications) that control theorists study are rather different compared to those that are now popular in machine learning.

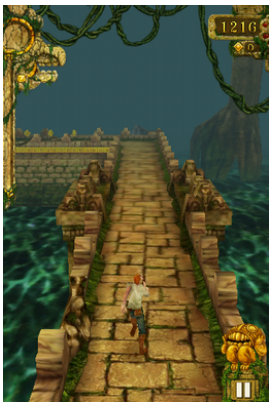

Figure 1: Temple Run

Setup

We will see that RL is actually "in-between" supervised and unsupervised learning.

The basis of RL is an environment (modeled by a dynamical system), and a learning module (called an *agent*) makes *actions* at each time step over a period of time. Actions have consequences: actions periodically lead to *reward*, or *penalty* (equivalently, negative reward). The goal is for the agent to learn the best *policy* that maximizes the cumulative reward. All fairly intuitive!

Here, the "best policy" is application-specific – it could refer to the best way to win a game of Space Invaders, or the best way to allocate investments across a portfolio of stocks, or the best way to navigate an autonomous vehicle, or the best way to set up a cooling schedule for an Amazon Datacenter.

All this is a bit abstract, so let us put this into concrete mathematical symbols, and interpret them (as an example) in the context of the classic iOS game *Temple Run*, where your game character is either Guy Dangerous or Scarlett Fox and your goal is to steal a golden idol from an Aztec temple while being chased by demons. (Fun game. See Figure 1.) Here,

- The environment is the 3D game world, filled with obstacles, coins, etc.
- The agent is the player.
- The agent receives a sequence observations (in the form of e.g. image pixels) about the environment.
- The state at time t, s_t, is the instantaneous relevant information of the agent (e.g. the 2D position and velocity of the player).
- The agent can choose an action, a_t, at each time step t (e.g. go left, go right, go straight). The next state of the game is determined by the current state and the current action:

  $$s_{t+1} = f(s_t, a_t).$$

  Here, f is the *state transition* function that is entirely determined by the environment. In control theory, we typically call this a *dynamical system*.

- The agent periodically receives rewards (coins/speed boosts) or penalties (speed bumps, or even death!). Rewards are also modeled as a function of the current state and action, $r(s_t, a_t)$.
- The agent's goal is to decide on a strategy (or policy) of choosing the next action based on all past states and actions: $s_t, a_{t-1}, s_{t-1}, \ldots, s_1, a_1, s_0, a_0$.
- The sequence of state-action pairs $\tau_t = (s_0, a_0, s_1, a_1, \ldots, a_t, s_t)$ is called a *trajectory* or *rollout*. Typically, it is impractical to store and process the entire history, so policies are chosen only over a fixed time interval in the past (called the *horizon length* L).

So a policy is simply defined as any function π that maps τ to a_t. Our goal is to figure out the best policy (where "best" is defined in terms of maximizing the rewards).

But as machine learning engineers, we can fearlessly handle minimization/maximization problems! Let us try and apply the ML tools we know here. Pose the cumulative negative reward as a loss function, and minimize this loss as follows:

$$\begin{aligned} \text{minimize } R(\tau) &= \sum_{t=0}^{L-1} -r(s_t, a_t), \\ \text{subject to } s_{t+1} &= f(s_t, a_t) \\ a_t &= \pi(\tau_t). \end{aligned}$$

The cumulative reward function $R(\tau)$ is sometimes replaced by the *discounted* cumulative reward, in case we exponentially decay the reward across time with some factor $\gamma > 0$:

$$R_{\text{discounted}}(\tau) = \sum_{t=0}^{L-1} -\gamma^t r(s_t, a_t).$$

OK, this looks similar to a loss minimization setting that we are all familiar with. We can begin to apply any of our optimization tools (e.g. SGD) to solve it. Several caveats emerge, however, and we have to be more precise about what we are doing.

First, what are the optimization variables? We are seeking the best among all *policies* π (which, above, are defined as functions from trajectories to actions), so this means that we will have to parameterize these policies somehow. We could imagine π to be a linear model that maps trajectories to actions, or kernel model, or a deep neural network. It really does not matter conceptually.

Second, what are the "training samples" provided to us and what are we trying to learn? The key assumptions in RL is that everything in the general case is probabilistic:

- the policy is stochastic. So what π is actually predicting from a given trajectory is not a *single best* action but a *distribution* over actions. More favorable actions get assigned higher probability and vice versa.
- the environment's dynamics, captured by f, can be stochastic.
- the reward function itself can be stochastic.

The last two assumptions are not critical – for example, in simple games, the dynamics and the reward are deterministic functions; but not so in more complex environments, such as the stock market – but the first one (stochastic policies) is fundamental in RL. This also hints to why we are optimizing over probabilistic policies in the first place: if there was no uncertainty and everything was deterministic, an oracle could have designed an optimal sequence of actions for all time before hand. (In older

Atari-style or Nintendo video games, this could indeed be done and one could play an optimal game pretty much from memory).

Since policies are probabilistic, they induce probability distribution over trajectories, and hence the cumulative negative reward is also probabilistic. (It's a bit hard to grasp this, considering that all the loss functions that we have talked about until now in deep learning have been deterministic, but the math works out in a similar manner.) So to be more precise, we will need to rewrite the loss in terms of the *expected value* over the randomness:

$$\begin{aligned}
&\text{minimize } \mathbb{E}_{\pi(\tau)} R(\tau) = \sum_{t=0}^{L-1} -r(s_t, a_t), \\
&\text{subject to } s_{t+1} = f(s_t, a_t) \\
&\qquad\qquad a_t = \pi(\tau_t), \text{ for } t = 0, \ldots, L-1.
\end{aligned}$$

This probabilistic way of thinking makes the role of ML a bit more clear. Suppose we have a yet-to-be-determined policy π. We pick a horizon length L, and execute this policy in the environment (the game engine, a simulator, the real world, ...) for L time steps. We get to observe the full trajectory τ and the sequence of rewards $r(s_t, a_t)$ for $t = 0, \ldots, L-1$. This pair is called a *training sample*. Because of the randomness, we simulate multiple such rollouts, and compute the cumulative reward averaged over all such rollouts, and adjust our policy parameters until this expectation is maximized.

We now return to the first sentence of this subsection: why RL is "in-between" supervised and unsupervised learning. In supervised learning we need to build a function that predicts label y from data features x. In unsupervised learning there is no separate label y; we typically wish to predict some intrinsic property of the dataset of x. In RL, the "label" is the action at the next time step, but once taken, this action becomes *part of the training data* and influences the subsequent action. This issue of intertwined data and labels (due to the possibility of complicated feedback loops across time) makes RL considerably more challenging.

Policy gradients

Let us now discuss a technique to numerically solve the above optimization problem. Basically, it will be akin to 'trial-and-error' – sample a rollout with some actions; if the reward is high then make those actions more probable (i.e., "reinforce" these actions), and if the reward is low then make those actions less probable.

In order to maximize expected cumulative rewards, we will need to figure out how to take gradients of the reward with respect to the policy parameters.

Recall that trajectories/rollouts τ are a probabilistic function of the policy parameters θ. Our goal is to compute the gradient of the expected reward, $\mathbb{E}_{\pi(\tau)} R(\tau)$ with respect to θ. To do so, we will need to take advantage of the *log-derivative trick*. Observe the following fact:

$$\begin{aligned}
\frac{\partial}{\partial \theta} \log \pi(\tau) &= \frac{1}{\pi(\tau)} \frac{\partial \pi(\tau)}{\partial \theta}, \text{ i.e.} \\
\frac{\partial \pi(\tau)}{\partial \theta} &= \pi(\tau) \frac{\partial}{\partial \theta} \log \pi(\tau).
\end{aligned}$$

Therefore, the gradient of the expected reward is given by:

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}_{\pi(\tau)}R(\tau) &= \frac{\partial}{\partial\theta}\sum_{\tau}R(\tau)\pi(\tau)\\ &= \sum_{\tau}R(\tau)\frac{\partial\pi(\tau)}{\partial\theta}\\ &= \sum_{\tau}R(\tau)\pi(\tau)\frac{\partial}{\partial\theta}\log\pi(\tau)\\ &= \mathbb{E}_{\pi(\tau)}[R(\tau)\frac{\partial}{\partial\theta}\log\pi(\tau)].\end{aligned}$$

So in words, the gradient of an expectation can be converted into an expectation over a closely related quantity. So instead of computing this expectation, like in SGD we *sample* different rollouts and compute a stochastic approximation to the gradient. The entire pseudocode is as follows.

Repeat:

1. Sample a trajectory/rollout $\tau = (s_0, a_0, s_1, \ldots, s_L)$.
2. Compute $R(\tau) = \sum_{t=0}^{L-1} -r(s_t, a_t)$
3. $\theta \leftarrow \theta - \eta R(\tau)\frac{\partial}{\partial\theta}\log\pi(\tau)$

There is a slight catch here, since we are reinforcing actions over the entire rollout; however, actions should technically be reinforced only based on future rewards (since they cannot affect past rewards). But this can be adjusted by suitably redefining $R(\tau)$ in Step 2 to sum over the t^{th} time step until the end of the horizon.

That's it! This form of policy gradient is sometimes called REINFORCE. Since we are sampling rollouts, this is also called *Monte Carlo Policy Gradient*.

In the above algorithm, notice that we never require direct access to the environment (or more precisely, the model of the environment, f) – only the ability to sample rollouts, and the ability to observe corresponding rewards. This setting is therefore called *model-free reinforcement learning*. A parallel set of approaches is model-based RL, which we will briefly touch upon next week.

Second, notice that since we don't require gradients, this works even for non-differentiable reward functions! In fact, the reward can be anything – non-smooth, non-differentiable, even discontinuous (such as a 0-1 loss).

Connection to random search

In the above algorithm, in order to optimize over rewards, observe we only needed to access function evaluations of the reward, $R(\tau)$, but *never its gradient*. This is in a departure from the regular gradient-based backpropagation framework we have been using thus far. The REINFORCE algorithm is in fact an example of *derivative free optimization*, which involves optimizing functions without gradient calculations.

Another way to do derivative free optimization is simple: just random search! Here is a quick introduction. If we are minimizing any loss function $f(\theta)$, recall that gradient descent updates θ along the negative direction of the gradient:

$$\theta \leftarrow \theta - \eta\nabla f(\theta).$$

But in random search, we pick a *random* direction v to update θ, and instead search for the (scalar) step size that provides maximum decrease in the loss along that direction. This is a rather inefficient way to minimize a loss function (for the same intuition that if we are trying to walk to the bottom of a valley, it is much better to follow the direction of steepest descent, rather than bounce around randomly.) But in the long run, random search does provably work as well. The pseudocode is as follows:

- Sample a random direction v
- Search for the step size (positive or negative) that minimizes $f(\theta + \eta v)$. Let that step size be η_{opt}.
- Set $\theta \leftarrow \theta + \eta_{\text{opt}} v$.

Again, observe that the gradient of f never shows up! The only catch is that we need to do a step size search (also called *line search*). However, this can be done quickly using a variation of binary search. Notice the similarity of the update rules (at least in form) to REINFORCE.

Let us apply this idea to policy gradients. Instead of the log-derivative trick, we will simply assume deterministic policies (i.e., a particular choice of policy θ leads to a deterministic rollout τ) use the above algorithm, with f being the reward function. The overall algorithm for policy gradient now becomes the following.

Repeat:

1. Sample a new policy update direction v.
2. Search for the step size η that minimize $R(\theta + \eta v)$.
3. Update the policy parameters $\theta \leftarrow \theta + \eta v$.

Done!

Details and extensions

We have only touched upon the bare minimum required to understand policy gradients in RL. This is a very vast area of emerging work and we cannot unfortunately do justice to all of it. Let us touch upon some practical aspects/concerns that may be of importance while trying to build RL systems.

First, the problem with REINFORCE is that we are replacing the expected value with a sample average in the gradient calculation, but unlike in standard SGD-type training, the *variance* of the sample average will be typically too high. This means that vanilla policy gradients will be far too slow and unreliable.

The standard solution is to perform *variance reduction*. One way to adjust the variance is via insertion of a quantity called the *reward baseline*. To understand this, observe that unlike regular gradient descent type training methods (which by definition depend on the slope/gradient of the loss), REINFORCE depends on the *absolute value*, not the *change*, of the reward function $R(\tau)$. This does not quite make sense: if a constant bias (of say +1000) is added uniformly to the reward function, the problem does not change fundamentally (we are just rewriting the reward on a different scale) but the algorithm changes quite a bit: in every iteration, every set of weights is likely to be reinforced positively no matter whether the action taken was good or bad.

A simple fix is to baseline-adjusted descent: subtract a baseline b from the reward function $R(\tau) - b$. Here is the method: we *learn* a baseline such that good actions are always associated with positive

reward, and bad actions are associated with negative reward. This is hard to do properly, and it is important to re-fit the baseline estimate each time. In the discounted reward case, we have to re-adjust the baseline depending on γ.

Also, note that in policy gradients we do not require a differentiable reward/loss, but we *do* require that the mapping π from trajectories to actions is differentiable — that's the only way we can properly define $\partial \log \pi$ in the policy gradient update step (and that's where standard neural net training methods such as backprop enter the picture).

To fix this, there is a class of techniques in RL called Evolutionary search (ES) that removes backprop entirely. The idea is to define the choice of *policy* itself as probabilistic functions (so π itself can be viewed as being drawn from a distribution over functions) and apply the log-derivative trick there. It's a bit complicated (and the gains over policy gradient are somewhat questionable) so we will not discuss this in detail here.