RL introduction: simple actor-critic for continuous actions

Andy Steinbach Follow
Dec 20, 2018 · 11 min read

Part 1 (Theory):

(Jump to the code implementation in part 2 or the TensorFlow code on Github if you prefer)

Reinforcement learning corresponds to our idea of what artificial intelligence should be: we drop a robotic "agent" into an environment it knows nothing about, and it learns to repeatably achieve an objective with *optimal* performance. First a little RL vernacular:

Our RL *agent* interacts with the world (called it's *environment*) by using a *policy* to choose at every time step from a set of *actions*. The environment responds by causing the agent to transition to its next *state*, and providing a *reward* attributed to the last action from the prior state.

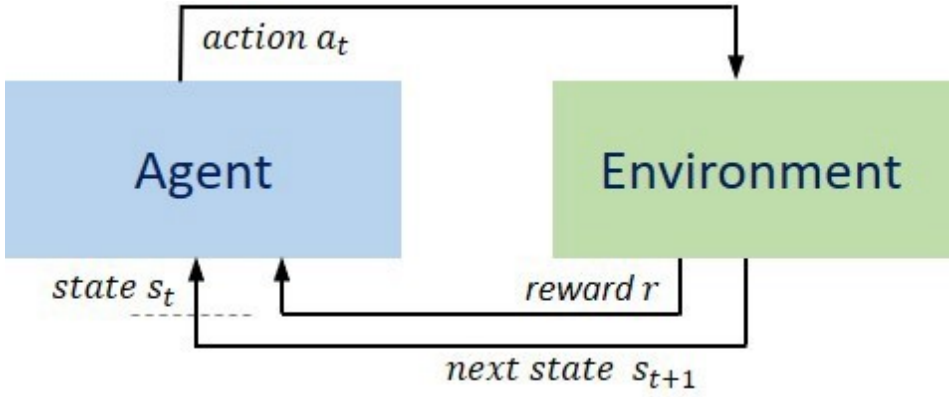

Figure 1

The space of allowed states and actions can be discrete or continuous and single or multi-variate, and the reward is scalar valued. Time is discrete. As the agent takes this series of actions through time, it therefore creates a state-action-reward sequence called the agent's *experience*:

$$\langle (s_0, a_0, r_1), (s_1, a_1, r_2), \ldots \quad (s_i, a_i, r_{i+1}), \ldots \qquad , \ldots (s_T, a_T, r_{T+1}) \rangle$$

(1) Agent's experience sequence

This sequence can be viewed as a "trajectory" through the state-action space of the system. For illustration purposes assume we have a simple game where our agent's objective is to drive a car and keep it on the road. There are only two discrete actions: "steer left" and "steer right", and only two discrete states: "on the road" and "off the road". One single-step cycle of agent-environment interaction looks like this:

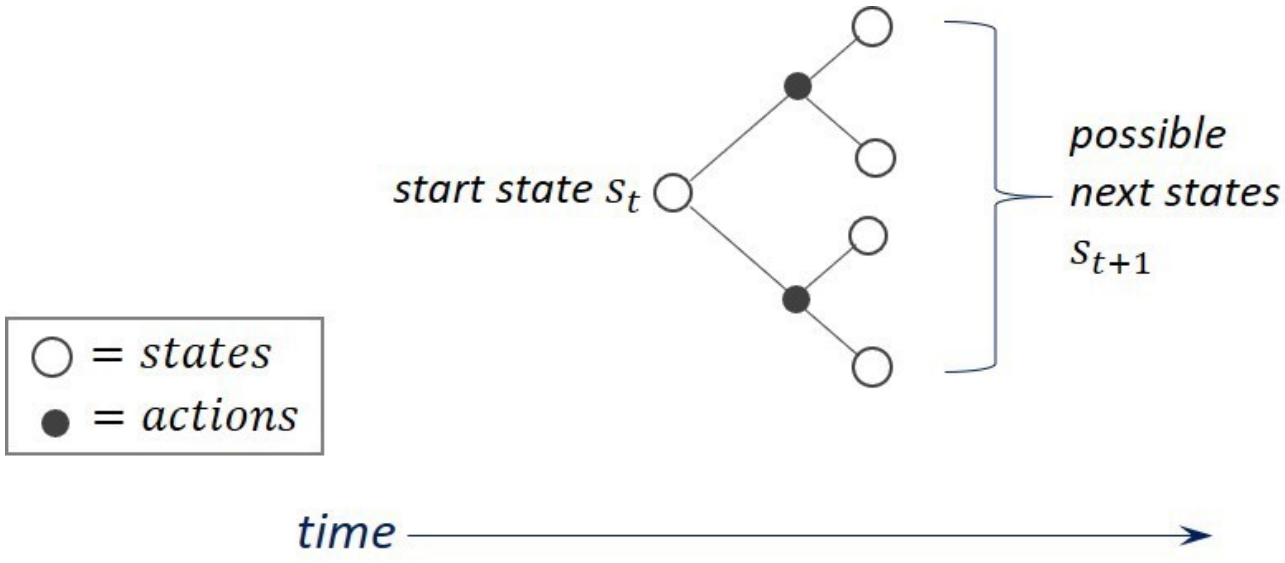

Figure 2. A single state-action-state' transition

The system starts from an arbitrary state, the agent chooses action "left" or "right", and the environment's dynamics determines what next state the system winds up in. Now we can stitch these diagrams together to show the multi-step branching tree of all possible trajectories the system could take:

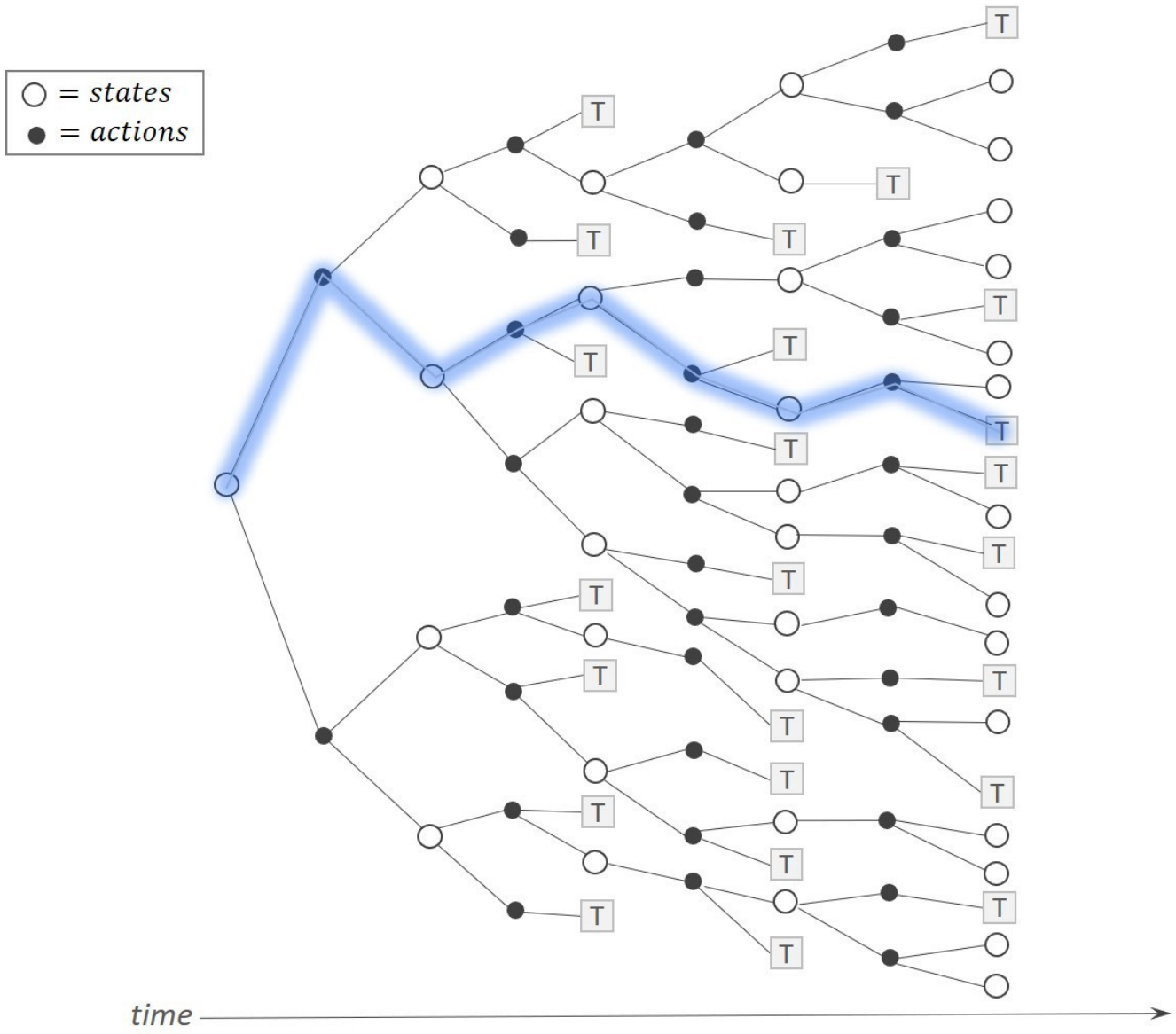

Figure 3. A full episodic trajectory (in blue)

We see that the number of possible paths exponentially blows up after a short time. (Side note: This was actually the exact problem for which Richard Bellman initially coined the phrase “the curse of dimensionality”, a term now widely used in machine learning referring to the exponential growth of state space volume with increasing input features.)

This branching tree allows us to visualize a number of RL concepts we’ll touch on as we go. For now, look at the trajectory in blue, which shows a

full trajectory of the agent from a start state until it reaches a "terminating state" (T), where the game is over. Each full trajectory represents one full play of the game (called an *episode*). The environment determines when the game has ended. In some environments, the game continues forever without terminating.

Let's introduce the bare minimum set of equations and concepts we'll need to implement our actor-critic algorithm in TensorFlow:

A *policy* π is a behavior function that maps states to actions:

$$a = \pi(s)$$

(2) A deterministic policy

If we consider a full episode (one of our complete trajectories) that starts at time t=0 and terminates at time t=T, then we define the *return Ro* from the starting state as:

$$R_0 = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t$$

(3) Definition of the **Return** from state=0. NOTE: it is easy to confuse reward (due to a single step) and Return (due to a whole trajectory)

Where r_t is the reward at time=t, and γ is a *discount factor* in the range [0,1], which allows us to adjust the time horizon we care about, and insure the return over long episodes remains finite. Note that an arbitrary

environmental transition from state $s \rightarrow s'$ is generally probabilistic (not deterministic), and so starting multiple episodes from the ***same*** state will generally result in ***different*** returns, since different episodes will follow different allowed trajectories. If we take the expectation value of return over *all* allowed trajectories, we can define the *expected return,* called the *state-value function* Vπ as:

$$V_\pi(s) = E[R_t \mid s_t = s]$$

(4) Definition of the **state-value function**

which describes the expected return starting from state *s* and then following policy π. Here R_t is the return of *a single, specific full episode* starting at time=t. The expectation operator $E[\cdot]$ averages over all possible individual episodes/trajectories starting from initial state *s* and following policy π. A closely related quantity is the action-*value function* Qπ

$$Q_\pi(s,a) = E[R_t \mid s_t = s, a_t = a]$$

(5) Definition of the **action-value function**

which describes the expected return starting in state *s*, taking action *a*, and thereafter following policy π.

"Deep" reinforcement learning simply means we will use deep neural networks as function approximators to estimate values for both Qπ and Vπ. To distinguish these neural network functions from the exact functions we use the following notation:

$$- \textit{The RL function } V_\pi \textit{ is estimated by neural network function } V_\pi^V$$

$$- \textit{ The RL function } Q_\pi \textit{ is estmated by neural network function } Q_\pi^U$$

where U and V represent the parameter vectors of the respective neural networks. We use stochastic gradient descent to train these networks. We still need to work out:

1. What quantities can we use as the "training labels" for training these estimators for *Q* and *V*?
2. How can we learn the policy behavior function π ?

We'll look at two methods for obtaining training labels because it gives us good insight into RL. In both cases, let's try to think of RL as a form of supervised machine learning, where we *generate our own set of training labels* by playing episodes of the game, starting with zero knowledge of the game. At first, we play poorly, but as our agent learns to play the game, the quality of the labels we generate improves progressively.

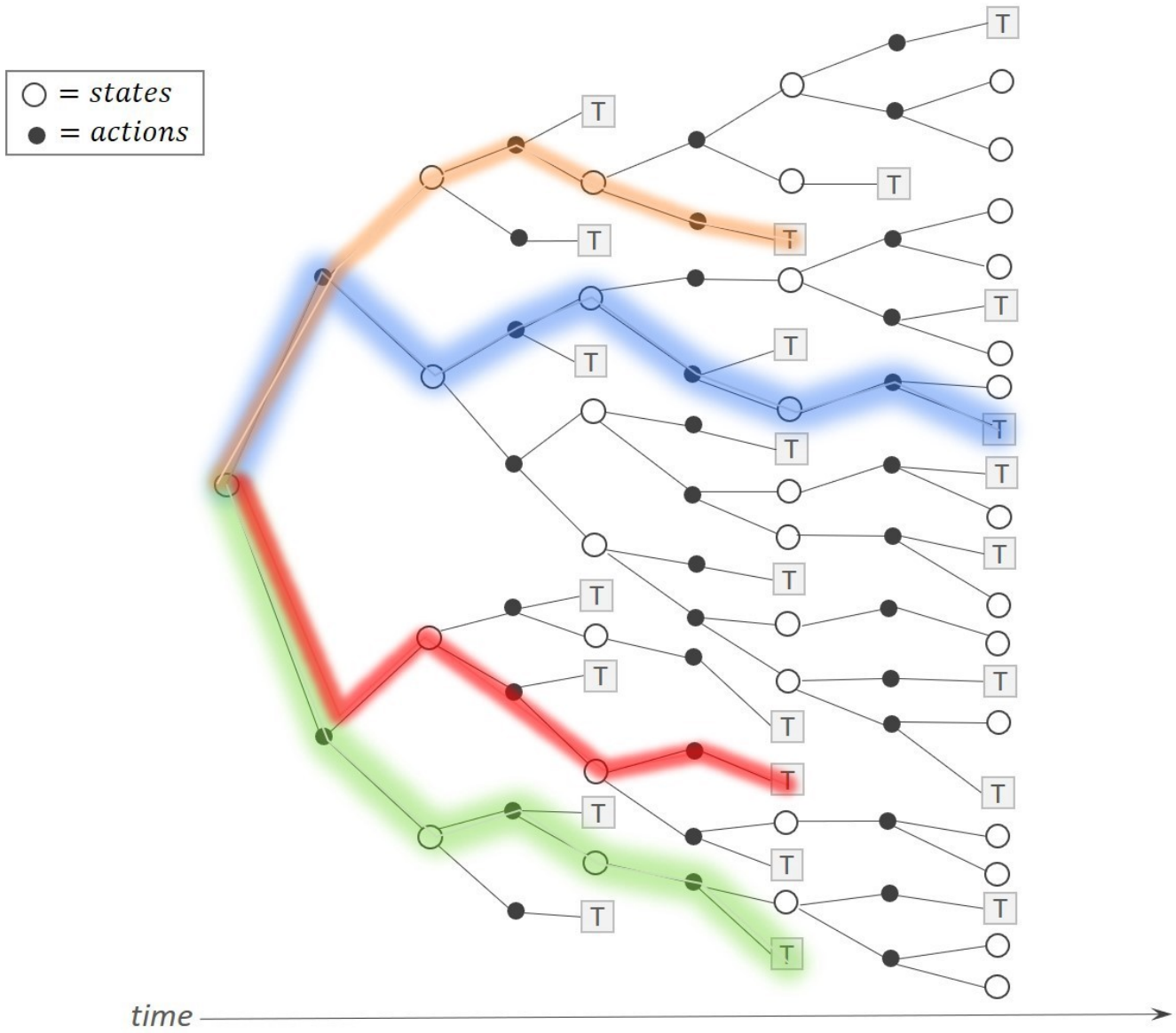

Figure 4. Several full episodic trajectories (each color)

The first method to obtain training labels is the *Monte Carlo (MC) method*. Here we play a *full* episode of the game — at every step we use our policy to choose an action, and the environment gives us back a next state and a reward, and so on, until the game ends. For example, each highlighted colored path in Fig. 4 represents a full MC trajectory. Now for each trajectory we simply calculate it's return R_t by applying Equation (3), and we use each value of R_t as an individual training label. Note each label is the *exact* value of return for that specific episodic path, but it is an *estimate*

of $V\pi(s)$ because we have not averaged over all possible game trajectories. We say the MC method is an *unbiased but noisy* estimator of expected return, because it uses exact, individual return samples (with no error for that path), but displays a lot of noise as training proceeds because the return jumps around as many different trajectories are sampled.

The second method for generating training labels uses the *Bellman expectation equation*:

$$V_\pi(s_t) = r + \gamma \cdot V_\pi(s_{t+1})$$

(6)

This recursive equation expresses $V\pi$ at $time=t$ in terms of the discounted $V\pi$ at $time=t+1$, plus the reward r when stepping from time t to $t+1$. This recursive property follows directly from the definition of return R_t in (3). Equation (6) gives us another way to obtain training labels. Let's substitute our state-value function estimator into (6) for the exact state-value function $V\pi$, we obtain:

$$V_\pi^V(s_t) \cong r + \gamma \cdot V_\pi^V(s_{t+1})$$

(7)

where the equality is now approximate since our estimator has error. In fact, rearranging (7) by putting both terms on the left-hand side, we obtain this error, which is called the temporal difference error, or *TD error*, represented by the symbol δ.

$$V_\pi^V(s_t) - r + \gamma \cdot V_\pi^V(s_{t+1}) = estimation\ error \equiv \delta$$

(8) Definition of the Temporal Difference Error (TD Error). (Author correction: the gamma term should have a negative sign)

We will use δ as our deep learning loss function, and drive this error toward zero using stochastic gradient ascent/descent, making our estimator a good approximator of the true $V\pi$.

The term on the right-hand side of (7) is called the temporal difference target, or *TD target,* and we use this as a training target (i.e. training label). In temporal difference learning we take a *single* step at time=t, obtaining from the environment the single step reward *r* and the next state (*s* at *time=t+1)*. We then use our function approximator to estimate $V\pi$ for the next state (at *time=t+1),* which by the definition of $V\pi$ gives the return for the rest of the trajectory from time=*t+1* until the terminating time. This TD learning method relies heavily on *bootstrapping* from fresh information contained *only* in the reward *r* at each step - a tenuous incremental process David Silver calls "updating a guess towards a guess". Nonetheless, with enough steps our estimator learns how to make accurate estimates of the state-value function $V\pi$.

Next, we also use a deep neural network with parameter vector θ to approximate the policy function π, and we adopt as the training objective J for policy π the expected return from the start state $V\pi(s)$. To optimize J, we need to find the gradient of J with respect to the policy parameters θ. This gradient is given by the *policy gradient theorem*:

$$\nabla_\theta J(\theta) = E\left[\nabla_\theta\left(log\pi^\theta(s,a)\right)\cdot Q_\pi(s,a)\right]$$

(9)

where π^θ represents the estimator of policy π parametrized by θ

and the loss function of policy π^θ is the start state expected return $J = V_{\pi^\theta}$

The proof is interesting but involved, so here we simply cite this central result. It is easy to get lost in all the subscripts, so consider a streamlined version of (9):

$$\nabla J = E[\nabla \log \pi \cdot Q]$$

(10)

A nice geometric interpretation of this policy gradient theorem (after Silver's RL course) is the following: at each sample of the expectation I take an action according to my policy and transition to a new state. The term $\nabla log\pi$ tells me the *direction* in θ-parameter space to move if I want "more" (toward the gradient) or "less" (opposite the gradient) of the current policy behavior due to the last step; and the term Q multiplying the gradient tells me if that action was in fact good (positive value) or bad (negative value). So, if we update at each step the parameters θ of our neural network policy π to follow this policy gradient, we train it to "take more of the good actions" and "less of the bad actions", incrementally improving its return.

We see clearly in (10) the concept of "actor-critic" emerge: the policy π is an actor taking different actions in the world, while the value function Q acts only as a critic, judging which of those actions are good or bad, and therefore which to encourage or discourage.

We're almost there!! Let's introduce just a few more improvements to help our algorithm achieve much better performance and greater simplicity. First, imagine that the action-value function Q (whose value is a scalar) had an offset and was only changing between 99 and 100 as we take our steps. It would be better to subtract off this average offset so the change would be between say *-0.5* and *+0.5*. We can subtract a baseline function of state (as long as it has no dependence on action) from Q in (5), and one choice is to use V as the baseline, giving us the so-called *advantage function* A:

$$A(s,a) = Q(s,a) - V(s)$$

V gives the expected return from a starting state *s averaged over all actions*, and Q gives the return of a specific action from that state. The difference is therefore the 'advantage' (positive or negative) of that specific action, compared to the average action. One last slight of hand: we pull out of the RL toolbox an equality that the advantage function A equals the expectation of our old friend, the TD error δ !

$$A_\pi(s,a) = E[\,\delta_\pi \mid s_t = s, a_t = a]$$

(11)

We are allowed to replace Q with A in (9) because the subtracted baseline V does not change the policy gradient, and finally using (11) our final expression for the policy gradient theorem becomes:

$$\nabla_\theta J(\theta) = E\left[\, \nabla_\theta \big(log\ \pi_\theta(s,a)\big) \cdot \delta_\pi \,\right]$$

(12)

This is our equation for training the neural network parametrized policy π. Note that now we only need to represent neural networks for π and $V\pi$ in our algorithm (since $\delta\pi$ is given in terms of $V\pi$) and we have eliminated the need to represent $Q\pi$ at all.

The final machinery we need is to represent our policy π not as a deterministic policy $a=\pi(s)$, but rather as a stochastic policy $\pi=P(a|s)$, where $P(a|s)$ is the probability distribution of taking action a given state s. Since we are interested in the case where action a is a continuous variable (not discrete), then P will be a probability density. We choose a simple Gaussian probability distribution to represent our stochastic policy:

$$\pi(a|\mu,\sigma) = \frac{1}{constant} e^{(a-\mu)^2/\sigma^2} \equiv \mathcal{N}(a|\mu,\sigma)$$

(13) Note mu and sigma are functions of state s

where μ is the mean and σ is the standard deviation of the Gaussian (normal) distribution abbreviated above as $script\ N(a|\mu,\sigma)$, and a is a continuous action drawn from $\mathbb{R}$ (real numbers). We can now write the

pseudo code of our algorithm, using all of our above relations:

Algorithm TD Advantage Actor-Critic

Randomly initialize critic network $V_\pi^U(s)$ *and actor network* $\pi^\theta(s)$ *with weights* U *and* θ
Initialize environment E
for *episode = 1, M* ***do***
  Receive initial observation state s_0 *from E*
  for *t=0, T* ***do***
    Sample action $a_t \sim \pi(a|\mu,\sigma) = \mathcal{N}(a|\mu,\sigma)$ *according to current policy*
    Execute action a_t *and observe reward* r *and next state* s_{t+1} *from E*
    Set TD target $y_t = r + \gamma \cdot V_\pi^U(s_{t+1})$
    Update critic by minimizing loss: $\delta_t = \left(y_t - V_\pi^U(s_t)\right)^2$
    Update actor policy by minimizing loss:
      $Loss = -log\left(\mathcal{N}(a \mid \mu(s_t), \sigma(s_t))\right) \cdot \delta_t$
    Update $s_t \leftarrow s_{t+1}$
  end for
end for

Each pass through the outer loop executes one full episode, and each pass through the inner loop will execute one environmental time step, until the current episode terminates.

This is a **online algorithm —** which means it can collect labels and train itself while it is also executing as an working agent (in the context of RL). This means it operates with a batch size of one: it executes one action and gets one reward and next state, and takes a training step/update based just on that one example. So, it is constantly learning with every step. The downside of this approach is that it discards every new data instance after just one use, it can't benefit from larger batch sizes (which is computationally efficient and has a stabilizing effect), and the sequential training data examples in one episode are highly correlated, which turns

out to be very problematic for training higher dimensional RL problems. More advanced actor-critic algorithms use an ***experience replay buffer*** to mitigate these shortcomings, by storing data for re-use. We will cover such an algorithm (DDPG) in a future part of this series, but you will notice that — at its heart — it nonetheless shares a very similar structure to our simpler algorithm here.

In part 2 of this series, we will implement this TD advantage actor-critic algorithm in TensorFlow, using one of the classic toy problems: **Continuous Mountain Car.** Get the code here now. My code here was initially inspired by the implementation of TD Advantage Actor-Critic in Denny Britz's GitHub RL repo (see here also for a wealth of great additional references).

(Note added 03–11–19: Here is also my own unpolished version of the more advanced algo ***DDPG for Continuous Mountain Car in TensorFlow***. Please note this code is hacky, but it does run. If/when I have time I will polish the code and provide some documentation).

Machine Learning Reinforcement Learning TensorFlow