SMART Goals : Algorithm

March 2020

1 Base Algorithm

Algorithm 1 Core execution loop

procedure SMART_PLAN(s, ϕ)
  if IS_TRIVIAL(s, ϕ) **then return** TRIVIAL_EXECUTE(s, ϕ)
  Φ = GENERATE_OPTIONS(s, ϕ)
  ϕ' = SELECT_OPTION(Φ, s, ϕ)
  $\tau_1,\ s$ = SMART_PLAN(s, ϕ')
  $\tau_2,\ s$ = SMART_PLAN(s, ϕ)
  return $\{\tau_1||\tau_2\},\ s$

Where the variables are defined as follows:

- $s \in S$: The representation of the state, should be vector
- ϕ, ϕ' : Representations of the objective, should be a vector
- Φ: set of objectives
- τ_1, τ_2 : Trajectories, returned for usage in memory later

Given this algorithm, we obviously have 4 fairly major holes that we need to fill in: the definitions of our four functions. Not shown here, we also need a distance function between states and goals In plain English terms, we would like them to do the following:

- IS_TRIVIAL : Should be simple and predictable, just serving as a base case for the function. Some possibilities include:
  - Recursion depth (would need to pass as argument)
  - If distance between state and goal is less than some threshold
  - If goal node is less than some distance from an adjacent node
  - Some learned function

- TRIVIAL_EXECUTE : Again, intended to be simple. Preferably has the ability to early exit, which would allow us to have kinda' a cascade of early exits. Would strongly prefer that this portion remains as static as possible, to simplify problem. Some possibilities are listed below.
- GENERATE_OPTIONS : Some function that generates options from the space of all possible goals. If tractable, could literally return all of the possible states. In practice, will need to return some subset of all possible goals. In the current formulation, this takes the form of N sample draws from some distribution conditioned on the current state and the goal. Under this formulation, it would ideally like to generate goals that are more likely to have been selected if all goals were possible.
- SELECT_OPTION : Some function that maps from a list of goals to a single goal. For end-to-end differentiability, I think that making this probabilistic makes a lot of sense.

I would like to keep IS_TRIVIAL and TRIVIAL_EXECUTE as simple as possible. For brevity of notation, we'll choose some function for IS_TRIVIAL, $T(s, \phi) => [0, 1]$, giving the probability that reaching a goal from a given state is trivial. Note that we only ever will need to compute whether or not reaching a goal from a given state is trivial, we will never actually need to evaluate two goals. We'll write TRIVIAL_EXECUTE as π_ϵ, because it looks vaguely like an E for Execute. We'll define a generator $G(s, \phi)$, and currently assume that SELECT_OBJECTIVE simply returns N i.i.d. samples from a list. Additionally, we'll make a simplifying assumption that our goal selection policy is $\pi_g(\phi', \phi, s, \Phi) = \frac{\exp(\lambda V(\phi', \phi, s)}{\sum_{\phi^* \in \Phi} \exp(\lambda V(\phi^*, \phi, s))}$, for some hyperparameter λ. Obviously, as $\lambda \to \infty$, this approaches a hard policy, but I think a soft policy is better for a couple of reasons. First, it provides gradients for more samples from the generator (as opposed to only the selected sample)

I do want to briefly say that I'm not sure that this is the best way to set up the problem, just probably the most dumb and simple. I could very easily see a formulation that uses variational inference or Metropolis-Hastings to try to approximate some joint distribution $\pi_g \odot G$, and try to learn this joint distribution to maximize our expected reward. Another interesting way to view this could be to use actor critic methods. In this scenario, the generator generates a single goal, which is chosen with 100% probability, and the function is trained with standard policy gradient methods. I'm writing down the below method simply because I think it's the simplest and (probably) the easiest to get working and understand, although I think that the other 2 methods probably have a little bit better of a theoretical foundation.

Anyways, with these definitions given, I'll try to provide some useful loss functions. The first is the generator loss, which vaguely resembles a generative adversarial network. Following that inspiration, we assume the generator to be of the form $G(*, s, \phi)$, where $*$ is a vector of normally distributed random noise.

The loss for the generative network is given as

$$L_G(\phi, s, \Phi) = -\sum_{\phi^* \in \Phi} \pi_G(\phi^*, \phi, s, \Phi) V(\phi^*, \phi, s) \tag{1}$$

And, naturally, the full loss is given

$$L_G = E_{s \sim \rho_s, \phi \sim \rho_\phi, * \ \mathcal{N}(0,I)}[L_G(\phi, s, \{G(*_1), ..., G(*_N)\}] \tag{2}$$

Where ρ_s and ρ_ϕ are the discounted visitation probability of states and the probability of goals. I was able to check out that all of these converge in the non-recursive case, but I'm having some difficulties proving it in the recursive case. You could use any number of loss functions for V, but I think the thing that seems the simplest is to just use the pathwise sum of rewards.

$$V(\phi^*, \phi, s) = E_{s' \sim \rho_\phi(s, \phi^*)}[V(\phi^*, s) + V(\phi, s')] \tag{3}$$

Where $\rho_\phi(s, \phi^*)$ is the discounted terminal state frequency of π given current state s and goal ϕ^*. With the slight bit of abuse of notation that

$$V(\phi, s) = T(\phi, s) V_\epsilon(\phi, s) + (1 - T(\phi, s)) E_{\phi' \sim \pi_g \odot G}[V(\phi', \phi, s)] \tag{4}$$

Where V_ϵ is the expected (discounted) reward for executing π_ϵ with parameters ϕ, s. In short, this is is simply the expected reward from setting out to go to ϕ^* from s, then the expected reward of going from wherever we actually ended up (discounted by time) to our second goal. While searching for difficulties in this, I was able to come up with a couple of potential places where this doesn't converge. First, assume that, for some ϕ, s, $T(\phi, s)$ ¡ 0.5. Assume that our goal function is devolved and is deterministic, such that our goal is always the same ϕ^* given this state and this goal. Finally, assume that, for whatever reason, executing $V_\epsilon(\phi, s)$ always leaves us back where we started at s. In this case, our function boils down to $V(\phi, s) = T(\phi, s)V_\epsilon(\phi, s) + (1 - T(\phi, s))[V(\phi, s) + V(\phi, s)]$, which clearly diverges.

One simple fix I came up with was a penalization term for not reaching our goals - we adjust the first equation to be

$$V(\phi^*, \phi, s) = E_{s' \sim \rho_\phi(s, \phi^*)}[V(\phi^*, s) + V(\phi, s') - V(\phi^*, s')] \tag{5}$$

This has a couple of benefits. First, it causes the equation to converge in the aforementioned scenario. Second, it forces our goal to be more than just a message passing mechanism, because we are penalized for our 'distance' to the goal, which is nice for interpretability. And finally, it potentially provides more data for us to learn from per observation, although this may be a small benefit.

Regarding how to learn π_ϵ, the simplest method (to me), would be to have some model of the world (either hard-coded or learned via unsupervised learning), and simply select the option that has the least expected distance from the goal. Alternately, you could probably learn how to achieve arbitrary goals in an unsupervised setting as well. From the literature that I've read thus far, it

seems like the most common choice for termination function is simply recursion depth, and that seems like a reasonable place to start, if not a reasonable place to end. In some problems, we may be able to get away with literally using state adjacency as our criterion.