DeepRLHacks

From a talk given by John Schulman titled "The Nuts and Bolts of Deep RL Research" (Aug 2017)
These are tricks written down while attending summer Deep RL Bootcamp at UC Berkeley.

Tips to debug new algorithm

1. Simplify the problem by using a low dimensional state space environment.
   - John suggested to use the Pendulum problem because the problem has a 2-D state space (angle of pendulum and velocity).
   - Easy to visualize what the value function looks like and what state the algorithm should be in and how they evolve over time.
   - Easy to visually spot why something isn't working (aka, is the value function smooth enough and so on).
2. To test if your algorithm is reasonable, construct a problem you know it should work on.
   - Ex: For hierarchical reinforcement learning you'd construct a problem with an OBVIOUS hierarchy it should learn.
   - Can easily see if it's doing the right thing.
   - WARNING: Don't over fit method to your toy problem (realize it's a toy problem).
3. Familiarize yourself with certain environments you know well.
   - Over time, you'll learn how long the training should take.
   - Know how rewards evolve, etc...
   - Allows you to set a benchmark to see how well you're doing against your past trials.
   - John uses the hopper robot where he knows how fast learning should take, and he can easily spot odd behaviors.

Tips to debug a new task

1. Simplify the task
   - Start simple until you see signs of life.
   - Approach 1: Simplify the feature space:
     - For example, if you're learning from images (huge dimensional space), then maybe hand engineer features first. Example: If you think your function is trying to approximate a location of something, use the x,y location as features as step 1.
     - Once it starts working, make the problem harder until you solve the full problem.
   - Approach 2: simplify the reward function.

- Formulate so it can give you FAST feedback to know whether you're doing the right thing or not.
- Ex: Have reward for robot when it hits the target (+1). Hard to learn because maybe too much happens in between starting and reward. Reformulate as distance to target instead which will increase learning and allow you to iterate faster.

Tips to frame a problem in RL

Maybe it's unclear what the features are and what the reward should be, or if it's feasible at all.

1. First step: Visualize a random policy acting on this problem.
    - See where it takes you.
    - If random policy on occasion does the right thing, then high chance RL will do the right thing.
        - Policy gradient will find this behavior and make it more likely.
    - If random policy never does the right thing, RL will likely also not.
2. Make sure observations usable:
    - See if YOU could control the system by using the same observations you give the agent.
        - Example: Look at preprocessed images yourself to make sure you don't remove necessary details or hinder the algorithm in a certain way.
3. Make sure everything is reasonably scaled.
    - Rule of thumb:
        - Observations: Make everything mean 0, standard deviation 1.
        - Reward: If you control it, then scale it to a reasonable value.
            - Do it across ALL your data so far.
    - Look at all observations and rewards and make sure there aren't crazy outliers.
4. Have good baseline whenever you see a new problem.
    - It's unclear which algorithm will work, so have a set of baselines (from other methods)
        - Cross entropy method
        - Policy gradient methods
        - Some kind of Q-learning method (checkout OpenAI Baselines as a starter or RLLab

Reproducing papers

Sometimes (often), it's hard to reproduce results from papers. Some tricks to do that:

1. Use more samples than needed.
2. Policy right... but not exactly
    - Try to make it work a little bit.
    - Then tweak hyper parameters to get up to the public performance.
    - If want to get it to work at ALL, use bigger batch sizes.
        - If batch size is too small, noisy will overpower signal.
        - Example: TRPO, John was using too tiny of a batch size and had to use 100k time steps.
        - For DQN, best hyperparams: 10k time steps, 1mm frames in replay buffer.

Guidelines on-going training process

Sanity check that your training is going well.

1. Look at sensitivity of EVERY hyper parameter

- If algo is too sensitive, then NOT robust and should NOT be happy with it.
- Sometimes it happens that a method works one way because of funny dynamics but NOT in general.

2. Look for indicators that the optimization process is healthy.
    - Varies
    - Look at whether value function is accurate.
        - Is it predicting well?
        - Is it predicting returns well?
        - How big are the updates?
    - Standard diagnostics from deep networks
3. Have a system for continuously benchmarking code.
    - Needs DISCIPLINE.
    - Look at performance across ALL previous problems you tried.
        - Sometimes it'll start working on one problem but mess up performance in others.
        - Easy to over fit on a single problem.
    - Have a battery of benchmarks you run occasionally.
4. Think your algorithm is working but you're actually seeing random noise.
    - Example: Graph of 7 tasks with 3 algorithms and looks like 1 algorithm might be doing best on all problems, but turns out they're all the same algorithm with DIFFERENT random seeds.
5. Try different random seeds!!
    - Run multiple times and average.
    - Run multiple tasks on multiple seeds.
        - If not, you're likely to over fit.
6. Additional algorithm modifications might be unnecessary.
    - Most tricks are ACTUALLY normalizing something in some way or improving your optimization.
    - A lot of tricks also have the same effect... So you can remove some of them and SIMPLIFY your algorithm (VERY KEY).
7. Simplify your algorithm
    - Will generalize better
8. Automate your experiments
    - Don't spend your whole day watching your code spit out numbers.
    - Launch experiments on cloud services and analyze results.
    - Frameworks to track experiments and results:
        - Mostly uses iPython notebooks.
        - DBs seem unnecessary to store results.

General training strategies

1. Whiten and standardize data (for ALL seen data since the beginning).
    - Observations:
        - Do it by computing a running mean and standard deviation. Then z-transform everything.
        - Over ALL data seen (not just the recent data).
            - At least it'll scale down over time how fast it's changing.

        - Might trip up the optimizer if you keep changing the objective.
        - Rescaling (by using recent data) means your optimizer probably didn't know about that and performance will collapse.
    - Rewards:
        - Scale and DON'T shift.
            - Affects agent's will to live.
            - Will change the problem (aka, how long you want it to survive).
    - Standardize targets:
        - Same way as rewards.
    - PCA Whitening?
        - Could help.
        - Starting to see if it actually helps with neural nets.
        - Huge scales (-1000, 1000) or (-0.001, 0.001) certainly makes learning slow.
2. Parameters that inform discount factors.
    - Determines how far you're giving credit assignment.
    - Ex: if factor is 0.99, then you're ignoring what happened 100 steps ago... Means you're shortsighted.
        - Better to look at how that corresponds to real time
            - Intuition, in RL we're usually discretizing time.
            - aka: are those 100 steps 3 seconds of actual time?
            - what happens during that time?
    - If TD methods for policy gradient of Value fx estimation, gamma can be close to 1 (like 0.999)
        - Algo becomes very stable.
3. Look to see that problem can actually be solved in the discretized level.
    - Example: In game if you're doing frame skip.
        - As a human, can you control it or is it impossible?
        - Look at what random exploration looks like
            - Discretization determines how far your browning motion goes.
            - If do many actions in a row, then tend to explore further.
            - Choose your time discretization in a way that works.
4. Look at episode returns closely.
    - Not just mean, look at min and max.
        - The max return is something your policy can hone in pretty well.
        - Is your policy ever doing the right thing??
    - Look at episode length (sometimes more informative than episode reward).
        - if on game you might be losing every time so you might never win, but... episode length can tell you if you're losing SLOWER.
        - Might see a episode length improvement in the beginning but maybe not reward.

Policy gradient diagnostics

1. Look at entropy really carefully
    - Entropy in ACTION space
        - Care more about entropy in state space, but don't have good methods for calculating that.
    - If going down too fast, then policy becoming deterministic and will not explore.

- If NOT going down, then policy won't be good because it is really random.
- Can fix by:
  - KL penalty
    - Keep entropy from decreasing too quickly.
  - Add entropy bonus.
- How to measure entropy.
  - For most policies can compute entropy analytically.
    - If continuous usually using a Gaussian so can compute differential entropy.

2. Look at KL divergence
   - Look at size of updates in terms of KL divergence.
   - example:
     - If KL is .01 then very small.
     - If 10 then too much.
3. Baseline explained variance.
   - See if value function is actually a good predictor or a reward.
     - if negative it might be over fitting or noisy.
       - Likely need to tune hyper parameters
4. Initialize policy
   - Very important (more so than in supervised learning).
   - Zero or tiny final layer to maximize entropy
     - Maximize random exploration in the beginning

Q-Learning Strategies

1. Be careful about replay buffer memory usage.
   - You might need a huge buffer, so adapt code accordingly.
2. Play with learning rate schedule.
3. If converges slowly or has slow warm-up period in the beginning
   - Be patient... DQN converges VERY slowly.

Bonus from Andrej Karpathy:

1. A good feature can be to take the difference between two frames.
   - This delta vector can highlight slight state changes otherwise difficult to distinguish.