Spinning Up as a Deep RL Researcher

By Joshua Achiam, October 13th, 2018

Table of Contents

If you're an aspiring deep RL researcher, you've probably heard all kinds of things about deep RL by this point. You know that it's hard and it doesn't always work. That even when you're following a recipe, reproducibility is a challenge. And that if you're starting from scratch, the learning curve is incredibly steep. It's also the case that there are a lot of great resources out there, but the material is new enough that there's not a clear, well-charted path to mastery. The goal of this column is to help you get past the initial hurdle, and give you a clear sense of how to spin up as a deep RL researcher. In particular, this will outline a useful curriculum for increasing raw knowledge, while interleaving it with the odds and ends that lead to better research.

The Right Background

Build up a solid mathematical background. From probability and statistics, feel comfortable with random variables, Bayes' theorem, chain rule of probability, expected values, standard deviations, and importance sampling. From multivariate calculus, understand gradients and (optionally, but it'll help) Taylor series expansions.

Build up a general knowledge of deep learning. You don't need to know every single special trick and architecture, but the basics help. Know about standard architectures (MLP, vanilla RNN, LSTM (also see this blog), GRU, conv layers, resnets, attention mechanisms), common regularizers (weight decay, dropout), normalization (batch norm, layer norm, weight norm), and optimizers (SGD, momentum SGD, Adam, others). Know what the reparameterization trick is.

Become familiar with at least one deep learning library. Tensorflow or PyTorch would be a good place to start. You don't need to know how to do everything, but you should feel pretty confident in implementing a simple program to do supervised learning.

Get comfortable with the main concepts and terminology in RL. Know what states, actions, trajectories, policies, rewards, value functions, and action-value functions are. If you're unfamiliar, Spinning Up ships with an introduction to this material; it's also worth checking out the RL-Intro from the OpenAI Hackathon, or the exceptional and thorough overview by Lilian Weng. Optionally, if you're the sort of person who enjoys mathematical theory, study up on the math of monotonic improvement theory (which forms the basis for advanced policy gradient algorithms), or classical RL algorithms (which despite being superseded by deep RL algorithms, contain valuable insights that sometimes drive new research).

Learn by Doing

Write your own implementations. You should implement as many of the core deep RL algorithms from scratch as you can, with the aim of writing the shortest correct implementation of each. This is by far the best way to develop an understanding of how they work, as well as intuitions for their specific performance characteristics.

Simplicity is critical. You should organize your efforts so that you implement the simplest algorithms first, and only gradually introduce complexity. If you start off trying to build something with too many moving parts, odds are good that it will break and you'll lose weeks trying to debug it. This is a common failure mode for people who are new to deep RL, and if you find yourself stuck in it, don't be discouraged—but do try to change tack and work on a simpler algorithm instead, before returning to the more complex thing later.

Which algorithms? You should probably start with vanilla policy gradient (also called REINFORCE), DQN, A2C (the synchronous version of A3C), PPO (the variant with the clipped objective), and DDPG, approximately in that order. The simplest versions of all of these can be written in just a few hundred lines of code (ballpark 250-300), and some of them even less (for example, a no-frills version of VPG can be written in about 80 lines). Write single-threaded code before you try writing parallelized versions of these algorithms. (Do try to parallelize at least one.)

Focus on understanding. Writing working RL code requires clear, detail-oriented understanding of the algorithms. This is because **broken RL code almost always fails silently,** where the code appears to run fine except that the agent never learns how to solve the task. Usually the problem is that something is being calculated with the wrong equation, or on the wrong distribution, or data is being piped into the wrong place. Sometimes the only way to find these bugs is to read the code with a critical eye, know exactly what it should be doing, and find where it deviates from the correct behavior. Developing that knowledge requires you to engage with both academic literature and other existing implementations (when possible), so a good amount of your time should be spent on that reading.

What to look for in papers: When implementing an algorithm based on a paper, scour that paper, especially the ablation analyses and supplementary material (where available). The ablations will give you an intuition for what parameters or subroutines have the biggest impact on getting things to work, which will help you diagnose bugs. Supplementary material will often give information about specific details like network architecture and optimization hyperparameters, and you should try to align your implementation to these details to improve your chances of getting it working.

But don't overfit to paper details. Sometimes, the paper prescribes the use of more tricks than are strictly necessary, so be a bit wary of this, and try out simplifications where possible. For example, the original DDPG paper suggests a complex neural network architecture and initialization scheme, as well as batch normalization. These aren't strictly necessary, and some of the best-reported results for DDPG use simpler networks. As another example, the original A3C paper uses asynchronous updates from the various actor-learners, but it turns out that synchronous updates work about as well.

Don't overfit to existing implementations either. Study existing implementations for inspiration, but be careful not to overfit to the engineering details of those implementations. RL libraries frequently make choices for abstraction that are good for code reuse between algorithms, but which are unnecessary if you're only writing a single algorithm or supporting a single use case.

Iterate fast in simple environments. To debug your implementations, try them with simple environments where learning should happen quickly, like CartPole-v0, InvertedPendulum-v0, FrozenLake-v0, and HalfCheetah-v2 (with a short time horizon—only 100 or 250 steps instead of the full 1000) from the OpenAI Gym. Don't try to run an algorithm in Atari or a complex Humanoid environment if you haven't first verified that it works on the simplest possible toy task. Your ideal experiment turnaround-time at the debug stage is <5 minutes (on your local machine) or slightly longer but not much. These small-scale experiments don't require any special hardware, and can be run without too much trouble on CPUs.

If it doesn't work, assume there's a bug. Spend a lot of effort searching for bugs before you resort to tweaking hyperparameters: usually it's a bug. Bad hyperparameters can significantly degrade RL performance, but if you're using hyperparameters similar to the ones in papers and standard implementations, those will probably not be the issue. Also worth keeping in mind: sometimes things will work in one environment even when you have a breaking bug, so make sure to test in more than one environment once your results look promising.

Measure everything. Do a lot of instrumenting to see what's going on under-the-hood. The more stats about the learning process you read out at each iteration, the easier it is to debug —after all, you can't tell it's broken if you can't see that it's breaking. I personally like to look at the mean/std/min/max for cumulative rewards, episode lengths, and value function estimates, along with the losses for the objectives, and the details of any exploration parameters (like mean entropy for stochastic policy optimization, or current epsilon for epsilon-greedy as in DQN). Also, watch videos of your agent's performance every now and then; this will give you some insights you wouldn't get otherwise.

Scale experiments when things work. After you have an implementation of an RL algorithm that seems to work correctly in the simplest environments, test it out on harder environments. Experiments at this stage will take longer—on the order of somewhere between a few hours and a couple of days, depending. Specialized hardware—like a beefy GPU or a 32-core machine—might be useful at this point, and you should consider looking into cloud computing resources like AWS or GCE.

Keep these habits! These habits are worth keeping beyond the stage where you're just learning about deep RL—they will accelerate your research!

Developing a Research Project

Once you feel reasonably comfortable with the basics in deep RL, you should start pushing on the boundaries and doing research. To get there, you'll need an idea for a project.

Start by exploring the literature to become aware of topics in the field. There are a wide range of topics you might find interesting: sample efficiency, exploration, transfer learning, hierarchy, memory, model-based RL, meta learning, and multi-agent, to name a few. If you're looking for inspiration, or just want to get a rough sense of what's out there, check out Spinning Up's key papers list. Find a paper that you enjoy on one of these subjects—something that inspires you—and read it thoroughly. Use the related work section and citations to find closely-related papers and do a deep dive in the literature. You'll start to figure out where the unsolved problems are and where you can make an impact.

Approaches to idea-generation: There are a many different ways to start thinking about ideas for projects, and the frame you choose influences how the project might evolve and what risks it will face. Here are a few examples:

Frame 1: Improving on an Existing Approach. This is the incrementalist angle, where you try to get performance gains in an established problem setting by tweaking an existing algorithm. Reimplementing prior work is super helpful here, because it exposes you to the ways that existing algorithms are brittle and could be improved. A novice will find this the most accessible frame, but it can also be worthwhile for researchers at any level of experience. While some researchers find incrementalism less exciting, some of the most impressive achievements in machine learning have come from work of this nature.

Because projects like these are tied to existing methods, they are by nature narrowly scoped and can wrap up quickly (a few months), which may be desirable (especially when starting out as a researcher). But this also sets up the risks: it's possible that the tweaks you have in mind for an algorithm may fail to improve it, in which case, unless you come up with more tweaks, the project is just over and you have no clear signal on what to do next.

Frame 2: Focusing on Unsolved Benchmarks. Instead of thinking about how to improve an existing method, you aim to succeed on a task that no one has solved before. For example: achieving perfect generalization from training levels to test levels in the Sonic domain or Gym Retro. When you hammer away at an unsolved task, you might try a wide variety of methods, including prior approaches and new ones that you invent for the project. It is possible for a novice to approch this kind of problem, but there will be a steeper learning curve.

Projects in this frame have a broad scope and can go on for a while (several months to a year-plus). The main risk is that the benchmark is unsolvable without a substantial breakthrough, meaning that it would be easy to spend a lot of time without making any progress on it. But even if a project like this fails, it often leads the researcher to many new insights that become fertile soil for the next project.

Frame 3: Create a New Problem Setting. Instead of thinking about existing methods or current grand challenges, think of an entirely different conceptual problem that hasn't been studied yet. Then, figure out how to make progress on it. For projects along these lines, a standard benchmark probably doesn't exist yet, and you will have to design one. This can be a huge challenge, but it's worth embracing—great benchmarks move the whole field forward.

Problems in this frame come up when they come up—it's hard to go looking for them.

Avoid reinventing the wheel. When you come up with a good idea that you want to start testing, that's great! But while you're still in the early stages with it, do the most thorough check you can to make sure it hasn't already been done. It can be pretty disheartening to get halfway through a project, and only then discover that there's already a paper about your idea. It's especially frustrating when the work is concurrent, which happens from time to time! But don't let that deter you—and definitely don't let it motivate you to plant flags with not-quite-finished research and over-claim the merits of the partial work. Do good research and finish out your projects with complete and thorough investigations, because that's what counts, and by far what matters most in the long run.

Doing Rigorous Research in RL

Now you've come up with an idea, and you're fairly certain it hasn't been done. You use the skills you've developed to implement it and you start testing it out on standard domains. It looks like it works! But what does that mean, and how well does it have to work to be important? This is one of the hardest parts of research in deep RL. In order to validate that your proposal is a meaningful contribution, you have to rigorously prove that it actually gets a performance benefit over the strongest possible baseline algorithm—whatever currently achieves SOTA (state of the art) on your test domains. If you've invented a new test domain, so there's no previous SOTA, you still need to try out whatever the most reliable algorithm in the literature is that could plausibly do well in the new test domain, and then you have to beat that.

Set up fair comparisons. If you implement your baseline from scratch—as opposed to comparing against another paper's numbers directly—it's important to spend as much time tuning your baseline as you spend tuning your own algorithm. This will make sure that comparisons are fair. Also, do your best to hold "all else equal" even if there are substantial differences between your algorithm and the baseline. For example, if you're investigating architecture variants, keep the number of model parameters approximately equal between your model and the baseline. Under no circumstances handicap the baseline! It turns out that the baselines in RL are pretty strong, and getting big, consistent wins over them can be tricky or require some good insight in algorithm design.

Remove stochasticity as a confounder. Beware of random seeds making things look stronger or weaker than they really are, so run everything for many random seeds (at least 3, but if you want to be thorough, do 10 or more). This is really important and deserves a lot of emphasis: deep RL seems fairly brittle with respect to random seed in a lot of common use cases. There's potentially enough variance that two different groups of random seeds can yield learning curves with differences so significant that they look like they don't come from the same distribution at all (see figure 10 here).

Run high-integrity experiments. Don't just take the results from the best or most interesting runs to use in your paper. Instead, launch new, final experiments—for all of the methods that you intend to compare (if you are comparing against your own baseline implementations)—and precommit to report on whatever comes out of that. This is to enforce a weak form of preregistration: you use the tuning stage to come up with your hypotheses, and you use the final runs to come up with your conclusions.

Check each claim separately. Another critical aspect of doing research is to run an ablation analysis. Any method you propose is likely to have several key design decisions—like architecture choices or regularization techniques, for instance—each of which could separately impact performance. The claim you'll make in your work is that those design decisions collectively help, but this is really a bundle of several claims in disguise: one for each such design element. By systematically evaluating what would happen if you were to swap them out with alternate design choices, or remove them entirely, you can figure out how to correctly attribute credit for the benefits your method confers. This lets you make each separate claim with a measure of confidence, and increases the overall strength of your work.

Closing Thoughts

Deep RL is an exciting, fast-moving field, and we need as many people as possible to go through the open problems and make progress on them. Hopefully, you feel a bit more prepared to be a part of it after reading this! And whenever you're ready, let us know.

PS: Other Resources

Consider reading through these other informative articles about growing as a researcher or engineer in this field:

Advice for Short-term Machine Learning Research Projects, by Tim Rocktäschel, Jakob Foerster and Greg Farquhar.

ML Engineering for AI Safety & Robustness: a Google Brain Engineer's Guide to Entering the Field, by Catherine Olsson and 80,000 Hours.

References

[1] Deep Reinforcement Learning Doesn't Work Yet, Alex Irpan, 2018

[2] Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control, Islam et al, 2017

[3] Deep Reinforcement Learning that Matters, Henderson et al, 2017

[4] Lessons Learned Reproducing a Deep Reinforcement Learning Paper, Matthew Rahtz, 2018

[5] UCL Course on RL

[6] Berkeley Deep RL Course

[7] Deep RL Bootcamp

[8] Nuts and Bolts of Deep RL, John Schulman

[9] Stanford Deep Learning Tutorial: Multi-Layer Neural Network

[10] The Unreasonable Effectiveness of Recurrent Neural Networks, Andrej Karpathy, 2015

[11] LSTM: A Search Space Odyssey, Greff et al, 2015

[12] Understanding LSTM Networks, Chris Olah, 2015

[13] Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, Chung et al, 2014 (GRU paper)

[14] Conv Nets: A Modular Perspective, Chris Olah, 2014

[15] Stanford CS231n, Convolutional Neural Networks for Visual Recognition

[16] Deep Residual Learning for Image Recognition, He et al, 2015 (ResNets)

[17] Neural Machine Translation by Jointly Learning to Align and Translate, Bahdanau et al, 2014 (Attention mechanisms)

[18] Attention Is All You Need, Vaswani et al, 2017

[19] A Simple Weight Decay Can Improve Generalization, Krogh and Hertz, 1992

[20] Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al, 2014

[21] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe and Szegedy, 2015

[22] Layer Normalization, Ba et al, 2016

[23] Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks, Salimans and Kingma, 2016

[24] Stanford Deep Learning Tutorial: Stochastic Gradient Descent

[25] Adam: A Method for Stochastic Optimization, Kingma and Ba, 2014

[26] An overview of gradient descent optimization algorithms, Sebastian Ruder, 2016

[27] Auto-Encoding Variational Bayes, Kingma and Welling, 2013 (Reparameterization trick)

[28] Tensorflow

[29] PyTorch

[30] Spinning Up in Deep RL: Introduction to RL, Part 1

[31] RL-Intro Slides from OpenAI Hackathon, Josh Achiam, 2018

[32] A (Long) Peek into Reinforcement Learning, Lilian Weng, 2018

[33] Optimizing Expectations, John Schulman, 2016 (Monotonic improvement theory)

[34] Algorithms for Reinforcement Learning, Csaba Szepesvari, 2009 (Classic RL Algorithms)

[35] Benchmarking Deep Reinforcement Learning for Continuous Control, Duan et al, 2016

[36] Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013 (DQN)

[37] OpenAI Baselines: ACKTR & A2C

[38] Asynchronous Methods for Deep Reinforcement Learning, Mnih et al, 2016 (A3C)

[39] Proximal Policy Optimization Algorithms, Schulman et al, 2017 (PPO)

[40] Continuous Control with Deep Reinforcement Learning, Lillicrap et al, 2015 (DDPG)

[41] RL-Intro Policy Gradient Sample Code, Josh Achiam, 2018

[42] OpenAI Baselines

[43] rllab

[44] OpenAI Gym

[45] OpenAI Retro Contest

[46] OpenAI Gym Retro

[47] Center for Open Science, explaining what preregistration means in the context of scientific experiments.