https://gist.github.com/zeyademam/0f60821a0d36ea44eef496633b4430fc

Should I use reinforcement learning for my problem?

What is the action space - what can the agent choose to do - does the action change the environment - continuous or discrete

What is the reward function - does it incentivize behaviour

It is a complex problem - classical optimization techniques such as linear programming or cross entropy may offer a simpler solution

Can I sample efficiently / cheaply - do you have a simulator

Reinforcement learning is hard

Debugging implementations is hard - very easy to have subtle bugs that don't break your code

Tuning hyperparameters is hard - tuning hyperparameters can also cover over bugs!

Results will succeed and fail over different random seeds (same hyperparameters!)

Machine learning is an empirical science, where the ability to do more experiments directly correlates with progress

Modern reinforcement learning suffers from sample inefficiency - agents often consume multiple lifetimes of human experience. This is required.

Mistakes and lessons

My own mistakes

Normalizing targets - a high initial target that occurs due to the initial weights can skew the normalization for the entire experiment

Doing multiple epochs over a batch

Not keeping batch size the same for experience replay & training

Not setting `next_observation = observation`

Not setting online & target network variables the same at the start of an experiment

Not gradient clipping - clip the norm of the gradient (I've seen between 1 - 5)

Mistakes of DSR students

Saving agent brain - not saving the optimizer state

Using too high a learning rate - learning rate is always important!!!

Building both an agent and environment

Building a state of the art agent (A3C)

Lessons learnt in DSR practicals

Lowering discount rate helps to reduce the magnitude of $Q(s, a)$.

Hyperparameters

Policy gradients - learning rate - clipping of distribution parameters (stochastic PG) - noise for exploration (deterministic PG) - network structure

Value function methods - learning rate - exploration (i.e. epsilon) - updating target network frequency - batch size - space discretization

Resources used for best practices

John Schulman – Berkley Deep RL Bootcamp 2017

How can I test if the training process of a reinforcement learning algorithm work correctly? Quora

(http://amid.fish/reproducing-deep-rl)

Best practices

Quick experiments on small test problems - CartPole for discrete action spaces - Pendulum for continuous action spaces

Compare to baselines

- a random agent is a good idea
- better is a good quality human designed agent

Be careful not to overfit these simple problems

- use low capacity neural networks
- large networks can potentially memorize sequences rather than learn generalizable rules
- less parameters means faster learning

Interpret & visualize learning process

- state visitation, value functions

Make it easier to get learning to happen (initially)

- input features, reward function design

Always use multiple random seeds

- rl is often not stable over random seeds (same hyperparam + different random seed -> different result)

Automate experiments

- don't waste time watching them run!
- making experiments easy and cheap (in terms of your time) means you will do more of them
- ML is an empirical science - more experiments -> more understanding

In reinforcement learning we often don't know the true min/max/mean/standard deviation of observations/actions/rewards/returns

Standardize data

- if observations in unknown range, estimate running average mean & stdev
- use the min & max if known

Rescale rewards

- but don't shift mean

- Writing Great Reward Functions - Bonsai has great info about how to write a reward function, including scaling.

Standardize prediction targets (i.e. value functions) the same way

Batch size matters

Policy gradient methods – weight initialization matters determines initial state visitation (i.e. exploration)

DQN converges slowly

Compute useful statistics - explained variance (for seeing if your value functions are overfitting), - computing KL divergence of policy before and after update (a spike in KL usually means degradation of policy) - entropy of your policy

Visualize statistics - running min, mean, max of episode returns - KL of policy update - explained variance of value function fitting - network gradients

Gradient clipping is helpful - dropout & batchnorm not so much

amid fish

(http://amid.fish/reproducing-deep-rl)

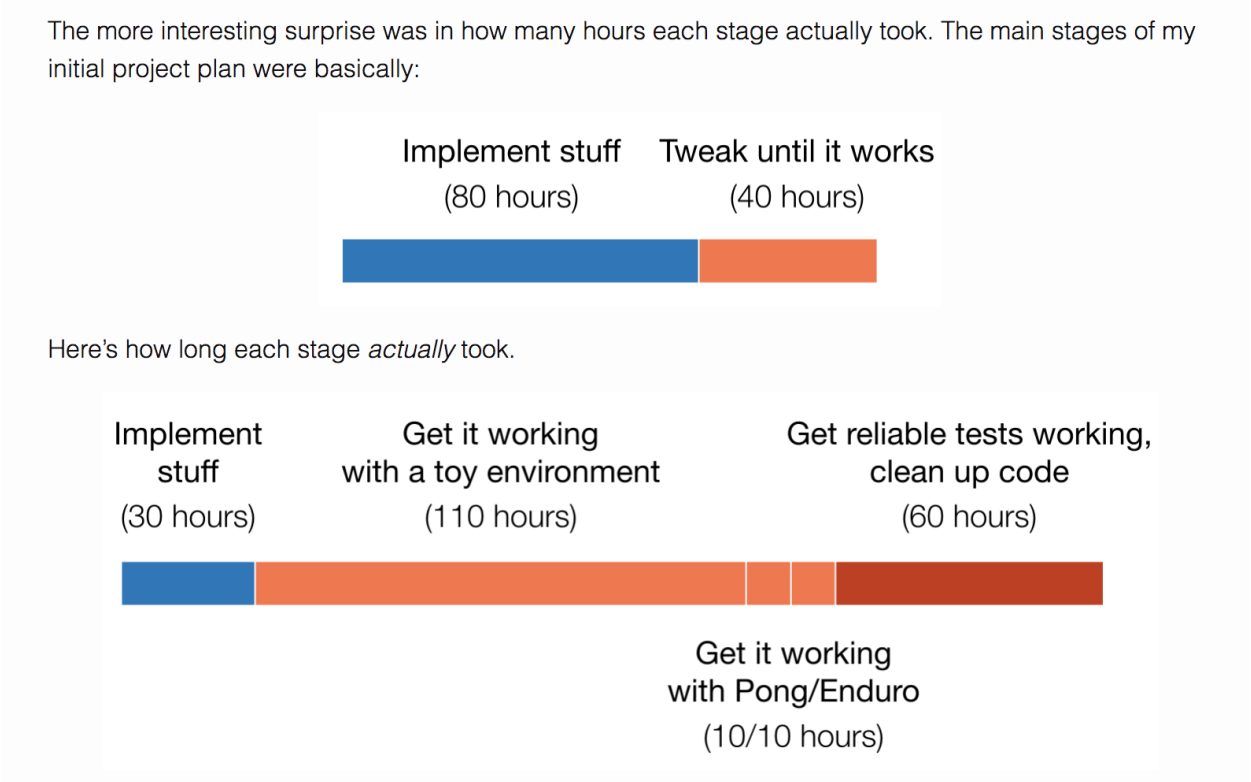


Figure 1: fig

> Reinforcement learning can be so unstable that you need to repeat every run multiple times with different seeds to be confident.
>
> It's not like my experience of programming in general so far where you get stuck but there's usually a clear trail to follow and you can get unstuck within a couple of days at most.

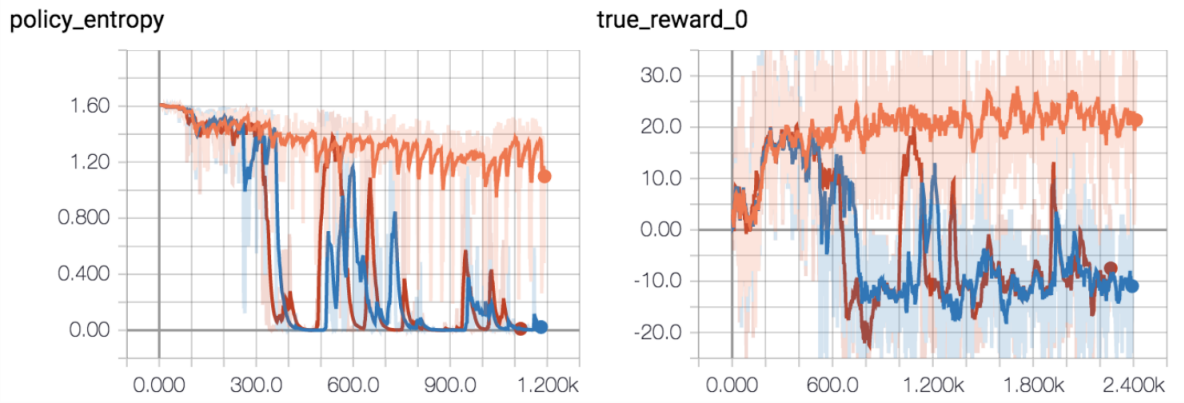
In total, the project took:

- **150 hours of GPU time and 7,700 hours (wall time × cores) of CPU time** on Compute Engine,
- **292 hours of GPU time** on FloydHub,
- and **1,500 hours (wall time, 4 to 16 cores) of CPU time** on my university's cluster.

I was horrified to realise that in total, that added up to **about $850** ($200 on FloydHub, $650 on Compute Engine) over the 8 months of the project.

Figure 2: fig

For example, once I thought everything was basically working, I sat down to make end-to-end tests for the environments I'd been working with. But I was having trouble getting even the simplest environment I'd been working with, training a dot to move to the centre of a square, to train successfully. I went back to the FloydHub job that had originally worked and re-ran three copies. It turned out that the hyperparameters I thought were fine actually only succeeded one out of three times.



It's not uncommon for two out of three random seeds (red/blue) to fail.

Figure 3: fig

It's more like when you're trying to solve a puzzle, there are no clear inroads into the problem, and the only way to proceed is to try things until you find the key piece of evidence or get the key spark that lets you figure it out.

Debugging

Debugging in four steps

1. evidence about what the problem might be
2. form hypothesis about what the problem might be (evidence based)
3. choose most likely hypothesis, fix
4. repeat until problem goes away

Most programming involves rapid feedback

- you can see the effects of changes very quickly
- gathering evidence can be cheaper than forming hypotheses

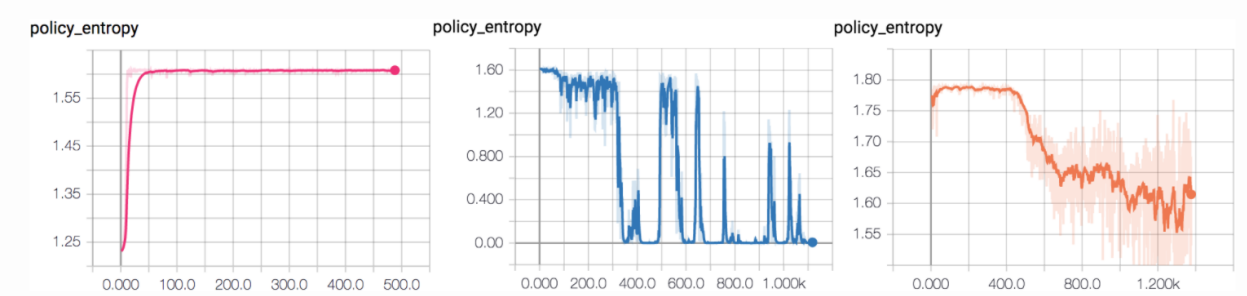
In RL (and supervised learning with long run times) gathering evidence is expensive

- suggests spending more time on the hypothesis stage
- switch from experimenting a lot and thinking little to **experimenting a little and thinking a lot**
- reserve experiments for after you've really fleshed out the hypothesis space

Get more out of runs

Recommends keeping a detailed work log - what output am I working on now - think out loud - what are the hypotheses, what to do next - record of current runs with reminder about what each run is susposed to answer - results of runs (i.e. TensorBoard)

Log all the metrics you can - policy entropy for policy gradient methods



Examples of unhealthy and healthy policy entropy graphs. Failure mode 1 (left): convergence to constant entropy (random choice among a subset of actions). Failure mode 2 (centre): convergence to zero entropy (choosing the same action every time). Right: policy entropy from a successful Pong training run.

Figure 4: fig

Try to predict future failures

RL specific - end to end tests of training - gym envs: -v0 environments mean 25% of the time action is ignored and previous action is repeated. Use -v4 to get rid of the randomness

General ML - for weight sharing, be careful with both dropout and batchnorm - you need to match additional variables - spikes in memory usages suggest validation batch size is too big - if you are struggling with the Adam optimizer, try an optimizer without momentum (i.e. RMSprop)

TensorFlow - `sess.run()` can have a large overhead. Try to group session calls - use the `allow_growth` option to avoid TF reserving memory it doesn't need - don't get addicted to TensorBoard - let your expts run!

Reward engineering

Important for any optimization system!

> Be careful what you wish for, you might just get it - Nick Bostrom