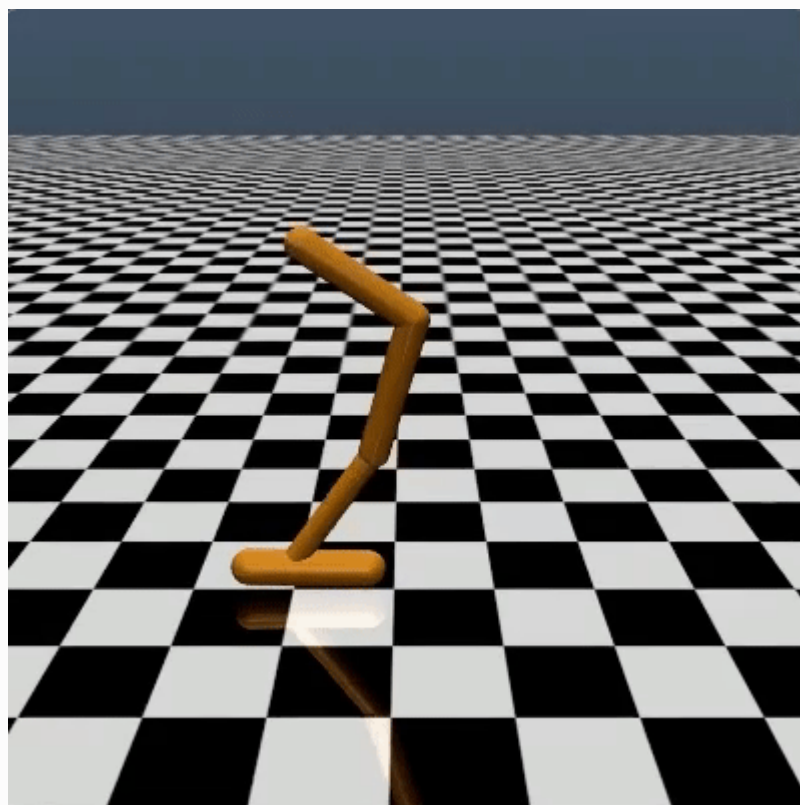


Amid Fish

Lessons Learned Reproducing a Deep Reinforcement Learning Paper

Apr 6, 2018

There are a lot of neat things going on in deep reinforcement learning. One of the coolest things from last year was OpenAI and DeepMind's work on training an agent using feedback from a human rather than a classical reward signal. There's a great blog post about it at [Learning from Human Preferences](#), and the original paper is at [Deep Reinforcement Learning from Human Preferences](#).



Learn some deep reinforcement learning, and you too can train a noodle to do backflip. From [Learning from Human Preferences](#).

I've seen a few recommendations that reproducing papers is a good way of levelling up machine learning skills, and I decided this could be an interesting one to try with. It was indeed a [super fun project](#), and I'm happy to have tackled it - but looking back, I realise it wasn't exactly the experience I thought it would be.

If you're thinking about reproducing papers too, here are some notes on what surprised me about working with deep RL.

First, in general, **reinforcement learning turned out to be a lot trickier than expected.**

A big part of it is that right now, reinforcement learning is really sensitive. There are a lot of details to get *just* right, and if you don't get them right, it can be difficult to diagnose where you've gone wrong.

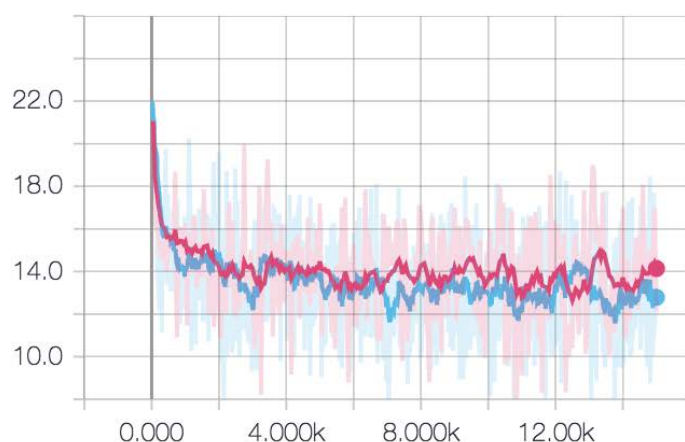
Example 1: after finishing the basic implementation, training runs just weren't succeeding. I had all sorts of ideas about what the problem might be, but after a couple of months of head scratching, it turned out to be because of problems with normalization of rewards and pixel data at a key stage¹. Even with the benefit of hindsight, there were no obvious clues pointing in that direction: the accuracy of the reward predictor network the pixel data went into was just fine, and it took a long time to occur to me to examine the rewards predicted carefully enough to notice the reward normalization bug. Figuring out what the problem was happened almost accidentally, noticing a small inconsistency that eventually lead to the right path.

Example 2: doing a final code cleanup, I realised I'd implemented dropout kind of wrong. The reward predictor network takes as input a pair of video clips, each processed identically by two networks with shared weights. If you add dropout and you're not careful about giving it the same random seed in each network, you'll drop out differently for each network, so the video clips won't be processed identically. As it turned out, though, fixing it completely broke training, despite prediction accuracy of the network looking exactly the same!

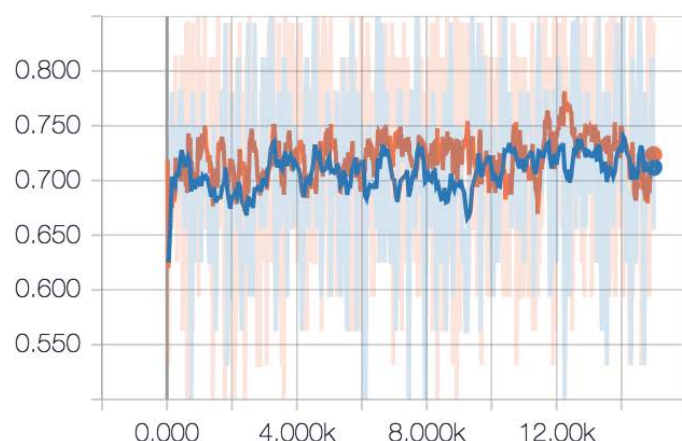
reward_predictor_accuracy_0



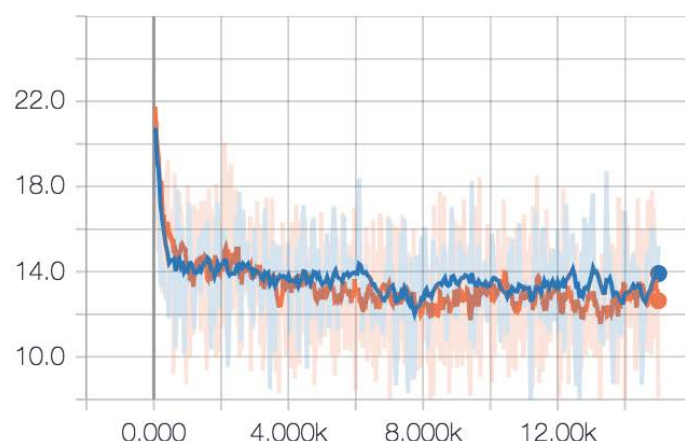
reward_predictor_loss_0



reward_predictor_accuracy_0



reward_predictor_loss_0



Spot which one is broken. Yeah, I don't see it either.

I get the impression this is a pretty common story (e.g. [Deep Reinforcement Learning Doesn't Work Yet](#)). My takeaway is that, starting a reinforcement learning project, you should **expect to get stuck like you get stuck on a math problem**. 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. 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.

A corollary is to **try and be as sensitive as possible in noticing confusion**.

There were a lot of points in this project where the only clues came from noticing some small thing that didn't make sense. For example, at some point it turned out that taking the difference between frames as features made things work much better. It was tempting to just forge ahead with the new features, but I realised I was confused about *why* it made such a big difference for the simple environment I was working with back then. It was only by following that confusion and realising that taking the difference between frames zeroed out the background that gave the hint of a problem with normalization.

I'm not entirely sure how to make one's mind do more of this, but my best guesses at the moment are:

- Learn to **recognise what confusion feels like**. There are a lot of different shades of the "something's not quite right" feeling. Sometimes it's code you know is ugly. Sometimes it's worry about wasting time on the wrong thing. But sometimes it's that *you've seen something you didn't expect*: confusion. Being able to recognise that exact shade of discomfort is important, so that you can...
- Develop the habit of following through on confusion. There are some sources of discomfort that it can be better to ignore in the moment (e.g. code smell while prototyping), but confusion isn't one of them. It seems important to really **commit yourself to always investigate whenever you notice confusion**.

In any case: expect to get stuck for several weeks at a time. (And have confidence you will be able to get to the other side if you keep at it, paying attention to those small details.)

Speaking of differences to past programming experiences, a second major learning experience was the **difference in mindset required for working with long iteration times**.

Debugging seems to involve four basic steps:

- Gather evidence about what the problem might be.
- Form hypotheses about the problem based on the evidence you have so far.
- Choose the most likely hypothesis, implement a fix, and see what happens.
- Repeat until the problem goes away.

In most of the programming I've done before, I've been used to rapid feedback. If something doesn't work, you can make a change and see what difference it makes within seconds or minutes. Gathering evidence is very cheap.

In fact, in rapid-feedback situations, gathering evidence can be a lot cheaper than forming hypotheses. Why spend 15 minutes carefully considering everything that could be causing what you see when you can check the first idea that jumps to mind in a fraction of that (and gather more evidence in the process)? To put it another way: if you have rapid feedback, you can narrow down the hypothesis space a lot faster by trying things than thinking carefully.

If you keep that strategy when each run takes 10 hours, though, you can easily waste a *lot* of time. Last run didn't work? OK, I think it's this thing. Let's set off another run to check. Coming back the next morning: still doesn't work? OK, maybe it's this other thing. Let's set off another run. A week later, you still haven't solved the problem.

Doing multiple runs at the same time, each trying a different thing, can help to some extent, but a) unless you have access to a cluster you can end up racking up a lot of costs on cloud compute (see below), and b) because of the kinds of difficulties with reinforcement learning mentioned above, if you try to iterate too quickly, you might never realise what kind of evidence you actually need.

Switching from **experimenting a lot and thinking a little** to **experimenting a little and thinking a lot** was a key turnaround in productivity. When debugging with long iteration times, you really need to *pour* time into the hypothesis-forming step - thinking about what all the possibilities are, how likely they seem on their own, and how likely they seem in light of everything you've seen so far. Spend as much time as you need, even if it takes 30 minutes, or an hour. Reserve experiments for once you've fleshed out the hypothesis space as thoroughly as possible and know which pieces of evidence would allow you to best distinguish between the different possibilities.

(It's especially important to be deliberate about this if you're working on something as a side project. If you're only working on it for an hour a day and each iteration takes a day to run, the number of runs you can do per week ends up feeling a precious commodity you have to make the most of. It's easy to then feel a sense of pressure to spend your working hour each day rushing to figure out something to do for that day's run. Another turnaround was being willing to spend several days just *thinking*, not starting any runs, until I felt really confident I had a strong hypothesis about what the problem was.)

A key enabler of the switch to thinking more was **keeping a much more detailed work log**. Working without a log is fine when each chunk of progress takes less than a few hours, but anything longer than that and it's easy to forget what you've tried so far and end up just going in circles. The log format I converged on was:

- Log 1: what specific output am I working on right now?
- Log 2: thinking out loud - e.g. hypotheses about the current problem, what to work on next
- Log 3: record of currently ongoing runs along with a short reminder of what question each run is supposed to answer
- Log 4: results of runs (TensorBoard graphs, any other significant observations), separated by type of run (e.g. by environment the agent is being trained in)

I started out with relatively sparse logs, but towards the end of the project my attitude moved more towards "log absolutely everything going through my head". The overhead was significant, but I think it was worth it - partly because some debugging required cross-referencing results and thoughts that were

days or weeks apart, and partly for (at least, this is my impression) general improvements in thinking quality from the massive upgrade to effective mental RAM.

19/03/2018

- Diagnosed problem with Pong run (3M45)
- Fix preference and segment communication (2h17)

Analysis:

- Catch-up on where we're at (9:23 to 9:30 – 7 minutes)
- Woke up and did end Pong run and check memory logs from 9:20:30 to 10:02 – 33 minutes
- Test hypothesis about why Pong run failed (10:11 to 10:30 – 19 minutes)
- Test hypothesis about failed Pong run (10:47 to 11:51 – 7 minutes)
- Figure out problem with failed Pong run (12:33 to 14:08 – 1:35)
- Fix segment/preference receiving (14:29 – 15:30 – 16:00)
- Test fixed segment/preference receiving (15:57 to 16:24 – 16:27)

OK, what's the state of affairs?

- The refactored code has passes moving dot end-to-end tests, but not Pong end-to-end tests.
- I'm not sure what the best way of moving segments/preferences between processes is.
- I need to check whether the Pong runs had a memory leak.
- I need to figure out whether moving dot still works with a simpler network.
- I need to create an end-to-end test for Enduro.

OK, so first, let's write up the results of the Pong and end-to-end runs, and check for memory leaks then.

OK, so what are my thoughts about the last Pong run (15:02:56)?

- Dropout somehow didn't get passed through.
- Learning rates were different compared to the last run (which worked).
- The way that segments/preferences are generated or sent is broken now.

OK, it is easy to test Nept, if the reward predictor are definitely being created without dropout 0.5 and backdoor false.

Learning rate? Reward predictor learning rate was 2e-4. A2C learning rate definitely 7e-4. And yep, that's definitely what they were for the master branch run which worked.

OK, so I guess it must really be something to do with the segments.

The master branch code (15:02:56) for segments:

```
seg_pair = (0x00000000, 0x00000000)
pref_pair = (0x00000000, 0x00000000)
```

Mentions of `recv_segments`:

```
while True:
    seg_pair = self.sample_seg_pair()
    if len(segments) == 2:
        break
    print("Waiting for segments")
    time.sleep(1.0)
```

while True:

```
pair_idx = 0
# If we've tested all the possible pairs of
segments so far.
# We might have to wait
while len(pair_idx) == 0:
    self.recv_segments(segments, seg_pair,
                        seg_pair_idx)
    pair_idx = self.sample_pair_idx(segments,
                                    exclude_pair_idx_tested_pairs)
```

OK, so it definitely calls `recv_segments` on every iteration of the main loop.

```
recv_prefs:
```

```
x1, x2, m = pref_pair.get(timeout=0.5)
```

Mentions of `recv_prefs`: well, the important one is

```
while True:
```

So yeah, it definitely calls `recv_prefs` every loop.

How about 15:02:56? (Code was 15:02:56)

```
recv_prefs = (0x00000000, 0x00000000)
pref_pair = (0x00000000, 0x00000000)
```

Mentions of `recv_prefs`:

```
while True:
    seg_pair = None
    while seg_pair is None:
        try:
            seg_pair = self.sample_seg_pair()
        except IndexError:
            # If we've tested all possible pairs of
            segments so far.
            # We'll have to wait for more segments
            time.sleep(1.0)
```

OK, looks fine.

```
recv_prefs:
```

```
x1, x2, m = pref_pair.get(timeout=0.5)
```

Mentions of `recv_prefs`:

```
while True:
    new_prefs = train_prefs(seg_train, pref_R_val,
                           val_prefs)
    if 1 and 1 % 100000 == 0:
        new_prefs.save()
        new_prefs(seg_train, pref_R_val,
                  new_prefs)
```

Huh. Also looks fine.

Hmm.

OK, is there any difference between the code that passed moving dot and this Pong code? Between `moving_dot_2` and 15:02:56?

OK, yeah, I have no idea what happened there. The tag is just pointing to the wrong commit.

OK, yeah, so far as I can tell, the Pong run was done with exactly the same code as the successful moving dot run.

Was the policy definitely training? Well, we got out metrics from the training stage, including policy entropy, which is generated while training, so yep.

Could segment generation or reward replacement be broken in the new code? Well, moving dot worked fine, though. Nept, can't see anything wrong.

Definitely using the right networks? Reward predictor using `net_con.A2C` using `ConPolicy`. So yep.

OK, what do the rewards look like from the one that trained well?

But wait, what would that tell me?

- If the rewards look fine in both cases: it's A2C's fault.
- If the rewards are clearly broken for the broken run: it's the reward predictor's fault.
- But it doesn't help me narrow down where it's broken.

OK, still, let's check that it's definitely the reward predictor that's broken.

What do the actual rewards look like? Well, it definitely gives a negative reward when we're about to miss:

OK, stop. What are my hypotheses?

- The new code isn't sending preferences about newer segments, whereas the old code is sending preferences about new segments.
- The new code is sending preferences about newer segments, but they're not being trained on. (I'd be very surprised if the training mechanism was broken. More likely, they're not making it to the preference database for some reason.)

Alright, how could I test which of these it is?

- Get to a state where the segment database is full.
- Have the policy generate different frames.
- Get to a state where the segment database should be full again.
- See if the preference interface is generating different frames.

OK, and we'll do it with the new code.

OK, that does indeed seem to be the problem.

Specifically, it's happening like this:

- In the old code, `recv_segments` doesn't stop once it reaches a maximum number of segments.
- Before the segment database is full, this is fine: the loop body is quick enough that it'll wait if a segment isn't available on the queue.
- But once the segment database gets full, each iteration of the loop body takes slightly longer. And the extra is just long enough for an extra segment to be serialised to the queue.
- So once the segment database gets full, `pref_interface` just gets stuck in that loop, and never sends any more preferences.

OK, fine, but then why doesn't it happen in the old codebase? Surely `del` (`del`) should be just as expensive?

OK, yes, it turns out that the `del` is expensive, but for whatever reason, serialisation takes longer in the old codebase, so it still breaks.

So let's think, how do we fix this?

So worst case, segments are generated faster than they can be serialised. That means we have no way of saying "get segments until there aren't any left".

Could we just fetch a certain number of segments each time? The problem with that would be:

- It severely hobbles the number of artificial preferences we can generate per second. (Maybe this isn't such a big problem. It just means we have to wait longer for initial preferences.)
- We would have to know how long it took to generate each segment in advance in order to be able to calculate how many to get. (Nah, we could get around this by just saying "Get as many segments as you can in 1 second".)
- What would be the alternative? Having a background thread constantly fetching segments. I don't like that because it'll use a lot of CPU. And maybe it would be a bad thing if all the segments are too recent. Probably we want some which are a little old.
- How do we figure out what value is right for that?

OK, let's not be too perfectionist. Let's just get as many segments as we can in 0.5 seconds.

OK, that's segments. What about preferences?

For preferences, we shouldn't lose any at all. If the queue has a finite size, there could always come a point where the reward predictor is being and preventing new prefs from being sent. So I think there we should have a thread always receiving them.

OK, so how does it work now? In summary:

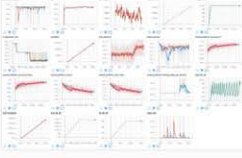
- A2C equips segments onto a queue of length 1 without blocking. Between preferences, `pref_interface` gets as many as it can in 0.5 seconds. Segments are cheap, so we don't mind if some are dropped.
- Preferences, though, are more expensive. So `pref_interface` sends them while blocking, also on a queue of length 1. A background process runs on the reward predictor receiving them all the time. Accesses to the preference database goes through a lock.

Alright, let's check moving dot still works on gpu:

(Side note: god, I wonder whether that's why it didn't work without dropout.

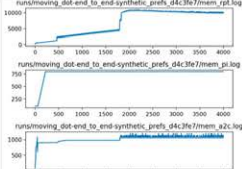
before. Because it adjusted the timings such that it didn't get new prefs. Shoulder shoulder shudder.)

On 8/13/18, new code passes moving dot.



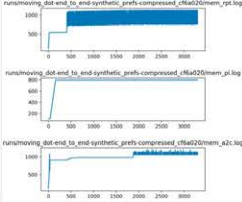
And yep, a checkpoint look good.

Total memory usage for 3 runs was about 36 GB.




So it looks like most of it is in the preference interface. That's not too surprising, seeing as we're keeping two copies of the preference dictionary.

But then, on 7/18/18: compress the values of the dictionary. Uses about 3 GB per run:



Still learns fine:



Took 55 minutes total, compared to 1h15 when running 3 of them simultaneously. So yeah, runtime doesn't seem to take a hit either. Success!

A typical day's log.

In terms of **getting the most out of the experiments you do run**, there are two things I started experimenting with towards the end of the project which seem like they could be helpful in the future.

First, adopting an attitude of **log all the metrics you can** to maximise the amount of evidence you gather on each run. There are obvious metrics like training/validation accuracy, but it might also be worth spending a good chunk of time at the start of the project brainstorming and researching which other metrics might be important for diagnosing potential problems.

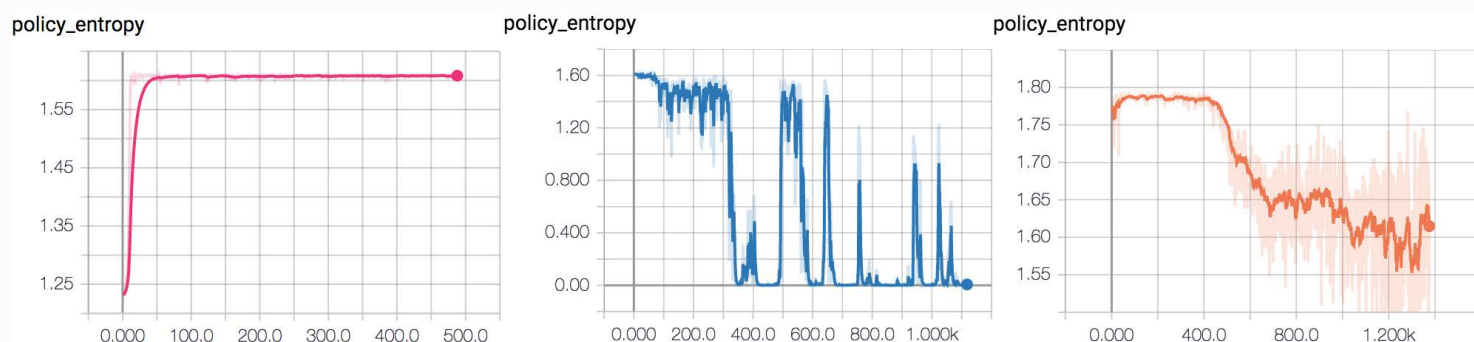
I might be making this recommendation partly out of hindsight bias where I *know* which metrics I should have started logging earlier. It's hard to predict which metrics will be useful in advance. Still, heuristics that might be useful are:

- For every important component in the system, consider what *can* be measured about it. If there's a database, measure how quickly it's growing in size. If there's a queue, measure how quickly items are being processed.
- For every complex procedure, consider how long different parts of it take. If you've got a training

loop, measure how long each batch takes to run. If you've got a complex inference procedure, measure how long each sub-inference takes. Those times are going to help a lot for performance debugging later on, and can sometimes reveal bugs that are otherwise hard to spot. (For example, if you see something taking longer and longer, it might be because of a memory leak.)

- Similarly, consider profiling memory usage of different components. Small memory leaks can be indicative of all sorts of things.

Another strategy is to look at what other people are measuring. In the context of deep reinforcement learning, John Schulman has some good tips in his [Nuts and Bolts of Deep RL talk](#) ([slides](#); [summary notes](#)). For policy gradient methods, I've found policy entropy in particular to be a good indicator of whether training is going anywhere - much more sensitive than per-episode rewards.



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.

When you do see something suspicious in metrics recorded, remembering to *notice confusion*, err on the side of assuming it's something important rather than just e.g. an inefficient implementation of some data structure. (I missed a multithreading bug for several months by ignoring a small but mysterious decay in frames per second.)

Debugging is much easier if you can see all your metrics in one place. I like to have as much as possible on TensorBoard. Logging arbitrary metrics with TensorFlow can be awkward, though, so **consider checking out [easy-tf-log](#)**, which provides an easy `tflog(key, value)` interface without any extra setup.

A second thing that seems promising for getting more out of runs is **taking the time to try and predict failure in advance**.

Thanks to hindsight bias, failures often seem obvious in retrospect. But the *really* frustrating thing is when the failure mode is obvious *before you've even observed what it was*. You know when you've set off a run, you come back the next day, you see it's failed, and even before you've investigated, you realise, "Oh, it must have been because I forgot to set the frobulator"? That's what I'm talking about.

The neat thing is that sometimes you can trigger that kind of half-hindsight-realisation in advance. It does take conscious effort, though - really stopping for a good five minutes before launching a run to think about what might go wrong. The particular script I found most helpful to go through was: [2](#)

1. Ask yourself, "How surprised would I be if this run failed?"

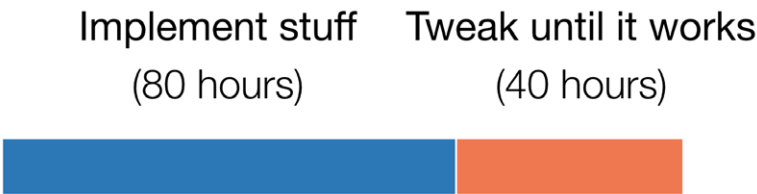
- 2. If the answer is ‘not very surprised’, put yourself in the shoes of future-you where the run *has* failed, and ask, “If I’m here, what might have gone wrong?”
- 3. Fix whatever comes to mind.
- 4. Repeat until the answer to question 1 is “very surprised” (or at least “as surprised as I can get”).

There are always going to be failures you couldn’t have predicted, and sometimes you still miss obvious things, but this does at least seem to *cut down* on the number of times something fails in a way you feel *really* stupid for not having thought of earlier.

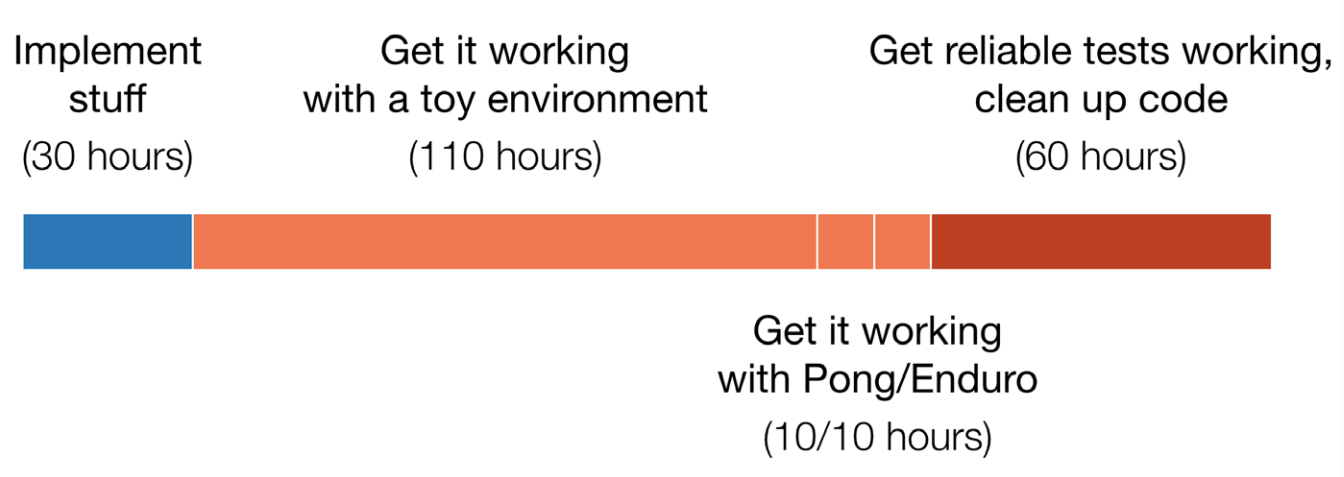
Finally, though, **the biggest surprise with this project was just how long it took** - and related, the amount of compute resources it needed.

The first surprise was in terms of calendar time. My original estimate was that as a side project it would take about 3 months. It actually took around *8 months*. (And the original estimate was supposed to be pessimistic!) Some of that was down to underestimating how many hours each stage would take, but a big chunk of the underestimate was failing to anticipate other things coming up outside the project. It’s hard to say how well this generalises, but **for side projects, taking your original (already pessimistic) time estimates and doubling them** might not be a bad rule-of-thumb.

The more interesting surprise was in how many hours each stage actually took. The main stages of my initial project plan were basically:



Here’s how long each stage *actually* took.



It wasn’t writing code that took a long time - it was debugging it. In fact, getting it working on even a [supposedly-simple environment](#) took *four times* as long as initial implementation. (This is the first side project where I’ve been keeping track of hours, but experiences with past machine learning projects have been similar.)

(Side note: be careful about designing from scratch what you hope should be an ‘easy’ environment for reinforcement learning. In particular, think carefully about a) whether your rewards really convey the right information to be able to solve the task - yes, this is easy to mess up - and b) whether rewards depend only on previous observations or also on current action. The latter, in particular, might be relevant if you’re doing any kind of reward prediction, e.g. with a critic.)

Another surprise was the amount of compute time needed. I was lucky having access to my university’s cluster - only CPU machines, but that was fine for some tasks. For work which needed a GPU (e.g. to iterate quickly on some small part) or when the cluster was too busy, I experimented with two cloud services: VMs on [Google Cloud Compute Engine](#), and [FloydHub](#).

Compute Engine is fine if you just want shell access to a GPU machine, but I tried to do as much as possible on FloydHub. FloydHub is basically a cloud compute service targeted at machine learning. You run `floyd run python awesomecode.py` and FloydHub sets up a container, uploads your code to it, and runs the code. The two key things which make FloydHub awesome are:

- Containers come preinstalled with GPU drivers and common libraries. (Even in 2018, I wasted a good few hours fiddling with CUDA versions while upgrading TensorFlow on the Compute Engine VM.)
- Each run is automatically archived. For each run, the code used, the exact command used to start the run, any command-line output, and any data outputs are saved automatically, and indexed through a web interface.

Find projects, datasets, and people...

JobsProjectsDatasetsExplore

Success

mrahtz/projects/learning-from-human-preferences/345

mrahtz submitted 3 weeks ago

Train moving dot end-to-end, ent_coef=0.02, n_initial_epochs=200, seed 3

GPU

1:07:20

Command

Success

mrahtz/projects/learning-from-human-preferences/344

mrahtz submitted 3 weeks ago

Train moving dot end-to-end, ent_coef=0.02, n_initial_epochs=200, seed 2

GPU

1:09:48

Command

Success

mrahtz/projects/learning-from-human-preferences/343

mrahtz submitted 3 weeks ago

Train moving dot end-to-end, ent_coef=0.02, n_initial_epochs=200, seed 1

GPU

1:08:25

Command

Shutdown

mrahtz/projects/learning-from-human-preferences/342

mrahtz submitted 3 weeks ago

Train Pong, 4 workers, starting from pretrained reward predictor

GPU

9:51:02

Command

Success

mrahtz/projects/learning-from-human-preferences/340

mrahtz submitted 3 weeks ago

CPU

3

Command

Success

mrahtz/projects/learning-from-human-preferences/339

mrahtz submitted 3 weeks ago

CPU

3

Command

LP

mrahtz / projects / learning-from-human-preferences / 345

Success

OverviewCodeDataOutputCLISettings

Train moving dot end-to-end, ent_coef=0.02, n_initial_epochs=200, seed 3

mrahtz submitted 3 weeks ago1:07:20GPUCliTensorflow

floyd run --gpu --env tensorflow-1.5 --message 'Train moving dot end-to-end, ent_coef=0.02, n_initial_epochs=200, seed 3' --data mrahtz/datasets/bases/4/gpu_tf15 --tensorboard 'bash floyd_wrapper_gpu_tf15.sh python run.py --random_queries --headless --env MovingDotNoFrameskip-v0 --log_dir /output --network easyfeatures --n_envs 1 --million_timesteps 0.2 --n_initial_epochs 200 --ent_coef=0.02 --seed 3'

System Metrics3 weeks ago (Metrics collected every min)

Training Metrics

LogsViewDownload

2018-03-07 03:48:44 PST Training batch 3
2018-03-07 03:48:45 PST Trained reward predictor for 23069 steps
2018-03-07 03:48:45 PST Training batch 4
2018-03-07 03:48:45 PST Trained reward predictor for 23070 steps
2018-03-07 03:48:45 PST Training batch 5
2018-03-07 03:48:45 PST Trained reward predictor for 23071 steps
2018-03-07 03:48:45 PST Training batch 6

LP

mrahtz / projects / learning-from-human-preferences / 345

Success

OverviewCodeDataOutputCLISettings

mrahtz/projects/learning-from-human-preferences/345/code1.04 MBLast updated 3 weeks agoDownload

.vscode

baselines

idea

Pipfile.lock18 KB

TODOs.md589 B

utils.py6.1 KB

pref_interface.py6.3 KB

enduro_wrapper.nv702 B

LP

mrahtz / projects / learning-from-human-preferences / 345

Success

OverviewCodeDataOutputCLISettings

mrahtz/projects/learning-from-human-preferences/345/output/output134.91 MBLast updated 3 weeks agoCreate DatasetDownload

policy_checkpoints

reward_pred

baselines

reward_predictor_checkpoints

misc

args.txt1.4 KB

FloydHub's web interface. Top: index of past runs, and overview of a single run. Bottom: both the code used for each run and any data output from the run are automatically archived.

I can't stress enough how important that second feature is. For any project this long, detailed records of what you've tried and the ability to reproduce past experiments are an absolute must. Version control software can help, but a) managing large outputs can be painful, and b) requires extreme diligence. (For example, if you've set off some runs, then make a small change and launch another run, when you commit the results of the first runs, is it going to be clear which code was used?) You could take careful notes or roll your own system, but with FloydHub, *it just works* and you save so much mental energy.

Other things I like about FloydHub are:

- Containers are automatically shut down once the run is finished. Not having to worry about checking runs to see whether they've finished and the VM can be turned off is a big relief.
- Billing is much more straightforward than with cloud VMs. You pay for usage in, say, 10-hour blocks, and you're charged immediately. That makes keeping weekly budgets much easier.

The one pain point I've had with FloydHub is that you can't customize containers. If your code has a lot of dependencies, you'll need to install them at the start of every run. That limits the rate at which you can iterate on short runs. You *can* get around this, though, by creating a 'dataset' which contains the changes to the filesystem from installing dependencies, then copying files from that dataset at the start of each run (e.g. `create_floyd_base.sh`). It's awkward, but still probably less awkward than having to deal with GPU drivers.

FloydHub is a little more expensive than Compute Engine: as of writing, \$1.20/hour for a machine with a K80 GPU, compared to about \$0.85/hour for a similarly-specced VM (though less if you don't need as much as 61 GB of RAM). Unless your budget is really limited, I think the extra convenience of FloydHub is worth it. The only case where Compute Engine can be a lot cheaper is doing a lot of runs in parallel, which you can stack up on a single large VM.

(A third option is Google's new [Colaboratory](#) service, which gives you a hosted Jupyter notebook with free access to a single K80 GPU. Don't be put off by Jupyter: you can execute arbitrary commands, and set up shell access if you really want it. The main drawbacks are that your code doesn't keep running if you close the browser window, and there are time limits on how long you can run before the container hosting the notebook gets reset. So it's not suitable for doing long runs, but can be useful for quick prototyping on a GPU.)

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.

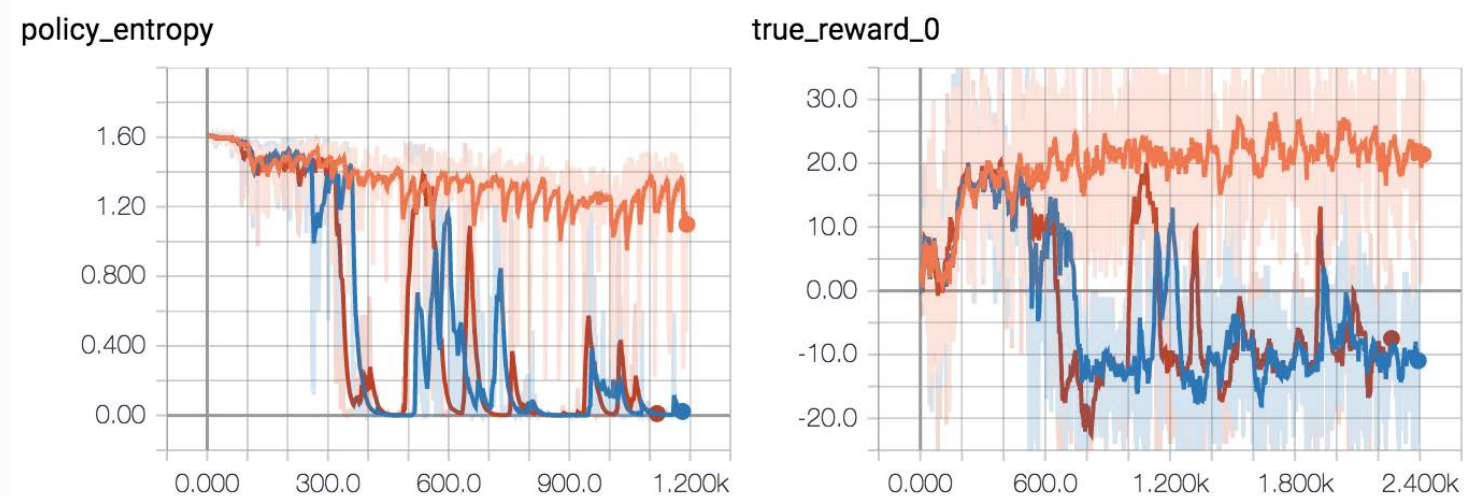
I was horrified to realise that in total, that added up to **about \$850** (\$200 on FloydHub, \$650 on Compute Engine) over the 8 months of the project.

Some of that's down to me being ham-fisted (see the above section on mindset for slow iteration).

Some of it's down to the fact that reinforcement learning is still so sample-inefficient that runs do just take a long time (up to 10 hours to train a Pong agent that beats the computer every time).

But a big chunk of it was down to a horrible surprise I had during the final stages of the project: **reinforcement learning can be so unstable that you need to repeat every run multiple times with different seeds to be confident.**

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.

To give a visceral sense of how much compute that means you need:

- Using A3C with 16 workers, Pong would take about 10 hours to train.
- That's 160 hours of CPU time.
- Running 3 random seeds, that 480 hours (20 days) of CPU time.

In terms of costs:

- FloydHub charges about \$0.50 per hour for an 8-core machine.
- So 10 hours costs about \$5 per run.
- **Running 3 different random seeds at the same time, that's \$15 per run.**

That's, like, 3 sandwiches every time you want to test an idea.

Again, from [Deep Reinforcement Learning Doesn't Work Yet](#), that kind of instability seems normal and accepted right now. In fact, even "Five random seeds (a common reporting metric) may not be enough to argue significant results, since with careful selection you can get non-overlapping confidence intervals."

(All of a sudden the \$25,000 of AWS credits that the [OpenAI Scholars programme](#) provides doesn't seem quite so crazy. That probably *is* about the amount you need to give someone so that compute

isn't a worry at all.)

My point here is that **if you want to tackle a deep reinforcement learning project, make sure you know what you're getting yourself into**. Make sure you're prepared for how much time it could take and how much it might cost.

Overall, reproducing a reinforcement learning paper was a fun side project to try. But looking back, thinking about which skills it actually levelled up, I'm also wondering whether reproducing a paper was really the best use of time over the past months.

On one hand, I definitely feel like my machine learning *engineering* ability improved a lot. I feel more confident in being able to recognise common RL implementation mistakes; my workflow got a whole lot better; and from this particular paper I got to learn a bunch about Distributed TensorFlow and asynchronous design in general.

On the other hand, I don't feel like my machine learning *research* ability improved much (which is, in retrospect, what I was actually aiming for). Rather than implementation, the much more difficult part of research seems to be coming up with ideas that are interesting but also *tractable and concrete*; ideas which give you the best bang-for-your-buck for the time you *do* spend implementing. Coming up with interesting ideas seems to be a matter of a) having a large vocabulary of concepts to draw on, and b) having good 'taste' for ideas (e.g. what kind of work is likely to be useful to the community). I think a better project for both of those might have been to, say, read influential papers and write summaries and critical analyses of them.

So I think my main meta-takeaway from this project is that **it's worth thinking carefully whether you want to level up engineering skills or research skills**. Not that there's no overlap; but if you're particularly weak on one of them you might be better off with a project specifically targeting that one.

If you want to level up both, a better project might be to read papers until you find something you're really interested in that comes with clean code, and trying to implement an extension to it.

If you *do* want to tackle a deep RL project, here are some more specific things to watch out for.

Choosing papers to reproduce

- Look for papers with few moving parts. Avoid papers which require multiple parts working together in coordination.

Reinforcement learning

- If you're doing anything that involves an RL algorithm as a component in a larger system, don't try and implement the RL algorithm yourself. It's a fun challenge, and you'll learn a lot, but RL is unstable enough at the moment that you'll never be sure whether your system doesn't work because of a bug in your RL implementation or because of a bug in your larger system.
- Before doing anything, see how easily an agent can be trained on your environment with a baseline

algorithm.

- Don't forget to normalize observations. *Everywhere* that observations might be being used. ³
- Write end-to-end tests as soon as you think you've got something working. Successful training can be more fragile than you expected.
- If you're working with OpenAI Gym environments, note that with `-v0` environments, 25% of the time, the current action is ignored and the previous action is repeated (to make the environment less deterministic). Use `-v4` environments if you don't want that extra randomness. Also note that environments by default only give you every 4th frame from the emulator, matching the early DeepMind papers. Use `NoFrameSkip` environments if you don't want that. For a fully deterministic environment that gives you exactly what the emulator gives you, use e.g. `PongNoFrameskip-v4`.

General machine learning

- Because of how long end-to-end tests take to run, you'll waste a lot of time if you have to do major refactoring later on. Err on the side of implementing things well the first time rather than hacking something up and saving refactoring for later.
- Initialising a model can easily take ~ 20 seconds. That's a painful amount of time to waste because of e.g. syntax errors. If you don't like using IDEs, or you can't because you're editing on a server with only shell access, it's worth investing the time to set up a linter for your editor. (For Vim, I like [ALE](#) with *both* [Pylint](#) and [Flake8](#). Though Flake8 is more of a style checker, it can catch some things that Pylint can't, like wrong arguments to a function.) Either way, every time you hit a stupid error while trying to start a run, invest time in making your linter catch it in the future.
- It's not just dropout you have to be careful about implementing in networks with weight-sharing - it's also batchnorm. Don't forget there are normalization statistics and extra variables in the network to match.
- Seeing regular spikes in memory usage while training? It might be that your validation batch size is too large.
- If you're seeing strange things when using Adam as an optimizer, it might be because of Adam's momentum. Try using an optimizer without momentum like RMSprop, or disable Adam's momentum by setting β_1 to zero.

TensorFlow

- If you want to debug what's happening with some node buried deep in the middle of your graph, check out `tf.Print`, an identity operation which prints the value of its input every time the graph is run.
- If you're saving checkpoints only for inference, you can save a lot of space by omitting optimizer parameters from the set of variables that are saved.
- `session.run()` can have a large overhead. Group up multiple calls in a batch wherever possible.
- If you're getting out-of-GPU-memory errors when trying to run more than one TensorFlow instance on the same machine, it could just be because one of your instances is trying to reserve all the GPU memory, rather than because your models are too large. This is TensorFlow's default behaviour. To tell TensorFlow to only reserve the memory it needs, see the `allow_growth`

option.

- If you want to access the graph from multiple things running at once, it looks like you *can* access the same graph from multiple threads, but there's a lock somewhere which only allows one thread at a time to actually do anything. This seems to be distinct from the Python global interpreter lock, which TensorFlow is [supposed to](#) release before doing heavy lifting. I'm uncertain about this, and didn't have time to debug more thoroughly, but if you're in the same boat, it might be simpler to just use multiple processes and replicate the graph between them with [Distributed TensorFlow](#).
- Working with Python, you get used to not having to worry about overflows. In TensorFlow, though, you still need to be careful:

```
> a = np.array([255, 200]).astype(np.uint8)
> sess.run(tf.reduce_sum(a))
199
```

- Be careful about using `allow_soft_placement` to fall back to a CPU if a GPU isn't available. If you've accidentally coded something that can't be run on a GPU, it'll be silently moved to a CPU. For example:

```
with tf.device("/device:GPU:0"):
    a = tf.placeholder(tf.uint8, shape=(4))
    b = a[..., -1]

sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True))
sess.run(tf.global_variables_initializer())

# Seems to work fine. But with allow_soft_placement=False

sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=False))
sess.run(tf.global_variables_initializer())

# we get

# Cannot assign a device for operation 'strided_slice_5':
# Could not satisfy explicit device specification '/device:GPU:0'
# because no supported kernel for GPU devices is available.
```

- I don't know how many operations there are like this that can't be run on a GPU, but to be safe, do CPU fallback manually:

```
gpu_name = tf.test.gpu_device_name()
device = gpu_name if gpu_name else "/cpu:0"
with tf.device(device):
    # graph code
```

Mental health

- Don't get addicted to TensorBoard. I'm serious. It's the perfect example of addiction through unpredictable rewards: most of the time you check how your run is doing and it's just pootling away, but as training progresses, sometimes you check and all of the sudden - jackpot! It's doing something super exciting. If you start feeling urges to check TensorBoard every few minutes, it might be worth setting rules for yourself about how often it's reasonable to check.

If you've read this far and haven't been put off, awesome! If you'd like to get into deep RL too, here are some resources for getting started.

- Andrej Karpathy's [Deep Reinforcement Learning: Pong from Pixels](#) is a great introduction to build motivation and intuition.
- For more on the theory of reinforcement learning, check out [David Silver's lectures](#). There isn't much on deep RL (reinforcement learning using neural networks), but it does teach the vocabulary you'll need to be able to understand papers.
- John Schulman's [Nuts and Bolts of Deep RL talk](#) ([slides](#); [summary notes](#)) has lots more tips about practical issues you might run into.

For a sense of the bigger picture of what's going on in deep RL at the moment, check out some of these.

- Alex Irpan's [Deep Reinforcement Learning Doesn't Work Yet](#) has a great overview of where things are right now.
- Vlad Mnih's talk on [Recent Advances and Frontiers in Deep RL](#) has more examples of work on some of the problems mentioned in Alex's post.
- Sergey Levine's [Deep Robotic Learning](#) talk, with a focus on improving generalization and sample efficiency in robotics.
- Pieter Abbeel's [Deep Learning for Robotics](#) keynote at NIPS 2017 with some of the more recent tricks in deep RL.

Good luck!

Thanks to [Michal Pokorný](#) and Marko Thiel for thoughts on a first draft on this post.

1. Observations are fed into two different training loops, policy training and reward predictor training, and I'd forgotten to normalize observations for the second one. Also, calculating running statistics (specifically, variance) is tricky. Check out [John Schulman's code](#) for a good reference.
2. This is basically [CFAR's](#) 'MurphyJitsu' script.
3. As mentioned above, I was stuck for a good while because of forgetting to normalize observations used for training the reward predictor. Derp.

Amid Fish

is Matthew Rahtz's blog

[GitHub](#), [LinkedIn](#), or say hello at matthew.rahtz@gmail.com!

