Dolphin Hands

Brian Beckman

[2019-03-25 Mon]

Contents

1 Control Shadow Hands with Dolphin

1.1 Dimensions and Degrees of Freedom

1.1.1 One Hand

The following code gets the original, old `gym` environment from `OpenAI` to tell us the dimensions.

```
import gym
import numpy as np
env = gym.make("HandManipulateBlock-v0")
_= env.reset()
action = env.action_space.sample() # your agent here
obn, reward, done, info = env.step(action)
result = list({"shape(hand + cube state)": np.shape(obn['observation']),
               "shape(achived cube goal)": np.shape(obn['achieved_goal']),
               "shape(desired cube goal)": np.shape(obn['desired_goal']),
               "shape(action)": np.shape(action)}.items())
return result
```

Table 1: Shapes of Prominent Quantities for Dolphin Hands

shape(hand + cube state)	(61)
shape(achived cube goal)	(7)
shape(desired cube goal)	(7)
shape(action)	(20)

Account for all the numbers in the table above:

1. Actions

   Action for one hand is a row vector of length twenty, the twenty tendons on the robot. The code clips each action component to the interval $[-1, +1]$.

2. Joints in Hand

   There are twenty-four joints in each hand:

   (a) two wrist joints, `WRJ1` and `WRJ0` (in reverse order, intentionally)

   (b) four first-finger joints, `FFJ3` through `FFJ0` (reverse order)

   (c) four middle-finger joints, `MFJ3` through `MFJ0` (reverse order)

   (d) four ring-finger joints, `RFJ3` through `RFJ0` (reverse order)

   (e) five little-finger joints, `LFJ4` through `LFJ0` (reverse order)

   (f) five thumb joints, `THJ4` through `THJ0` (reverse order)

   The following symbols in the code build up the state:

qpos a generic word for a 7-vector (three Cartesian position coordinates concatenated to four components of a normalized quaternion *versor*, in pos-quat order despite the name), but `qpos` is abused, in this case, to contain twenty-four generalized coordinates encoding joint configuration.

qvel concatenation of velocity and angular velocity, in that order despite the name, but abused to contain twenty-four generalized velocities.

Total is forty-eight.

3. Object (Cube)

   object usually means "cube on the left;" we retained the old name to minimize disruption in the intricate XML files that specify the models.

   object_right means "cube on the right."

   The code defines the following informal (non-machine-checked) types:

   goal a flattened or *raveled* 14--vector: seven components of *pose* for the left cube concatenated to seven components of pose for the right cube.

   pose a flattened 7--vector: three components of position concatenated to four components of normalized quaternion, which has only three degrees of freedom.

   pos three components of position, (x, y, z)

   quat four components of normalized quaternion, sometimes spelled `qut` to have the same number of letters as `pos` and enhance vertical alignment of code.

   qvel six components, surmised to be $(\dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$, where ϕ, θ, ψ are *roll*, *pitch*, and *yaw*, respectively.

4. Total: Sixty-One Dimensions

   $\boxed{48 + 13 = 61}$

1.2 Piecewise Linear Controllers

The objective of a **controller** is to orient the cube to a target. Generally, a controller takes in a state and a time and returns an action. In full generality, then, a controller or **agent** $\mathcal{A}$ for the hands takes in two, 61-dimensional states and a time, and returns a 20--dimensional action vector:

$$\mathcal{A} : \mathbb{R}^{123} \to \mathbb{R}^{20} \tag{1}$$

The objective of the Dactyl-hands controller, however, is to convert the actual pose of a cube into actions that bring it closer to the goal pose. Thus, the inputs to the controller number only seven, not 123.

A *linear regulator* is a special kind of controller that takes in a **residual** pose (7-vector), that is, the difference between the desired pose and the actual pose, and outputs actions. The difference of the *pos* part of a pose is just the vector difference of the Cartesian components. The difference of the *quat* part is the quaternion product of one quaternion and the quaternion conjugate of the

second quaternion, namely $q_1 \star q_2^\star$. A linear regulator, $\mathcal{L}$, being linear, is just a linear-control matrix (**LCM**), a linear function of the following type:

$$\mathcal{L} : \mathbb{R}^7 \to \mathbb{R}^{20} \tag{2}$$

This matrix has 140 numbers in it. By A/B preferences alone, Dolphin can learn a controller containing 140 numbers in about $7,000$ trials:

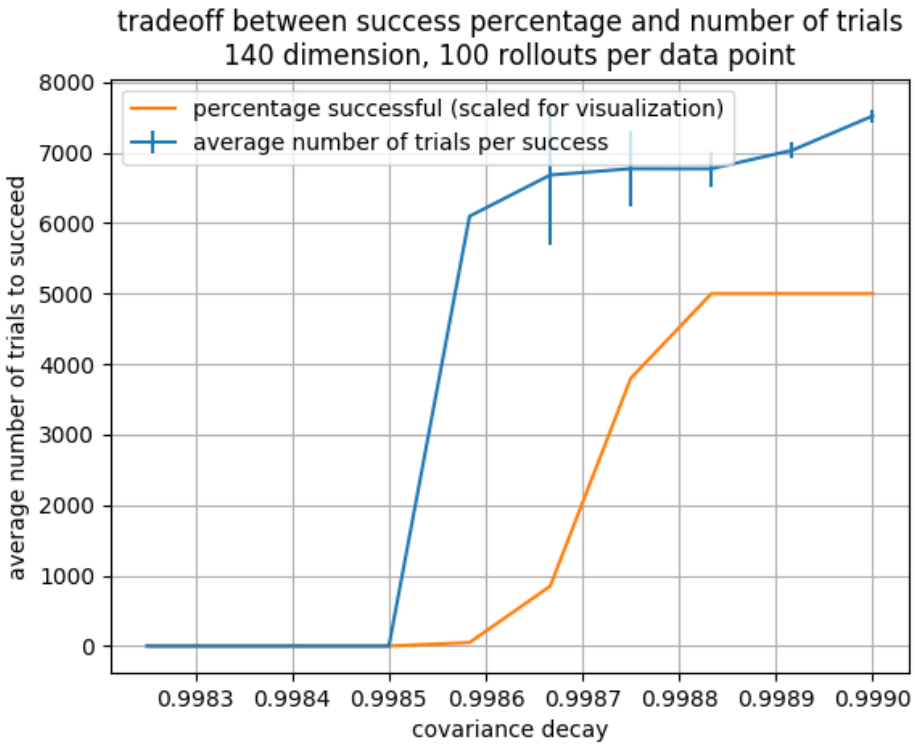

Figure 1: Trials versus covariance reduction for 140 dimensions

The terms *trial*, *rollout*, *episode*, and *trajectory* are synonyms until we learn otherwise.

A *piecewise linear controller* is a time-sequence of LCMs $\mathcal{L}_1, \mathcal{L}_2, \ldots$. We will need one new LCM, each containing 140 numbers, every few steps, say every ten. In 250 steps, we will need 25 LCMs, or $3,500$ numbers. Therefore, the entire, piecewise linear controller for a trial of length 250 steps has $3,500$ dimensions, taking $7,000^{10}$ trials. Without reduction of this number, learning by this method is not feasible.

However, because the individual LCMs in a piecewise linear controller will likely vary smoothly and slowly (with respect to the natural frequencies of the mechanics of the hand and the cube) over time, the number of trials needed to learn them all will likely be much smaller. An experiment is under way to find out.

1.3 Learn a Neural Network

Alternatively, we can have Dolphin learn a Dactyl neural network directly.

2 Get the Hands Going

2.1 Start Here (Ubuntu)

`https://github.com/rebcabin/baselines`

The instructions for setting up the Python environment are pretty good. Alternatives include PyCharm and pipenv. I got them all to work. Here is what I ended up with following the instructions above:

```
env PYTHONPATH=/usr/lib/tensorflow/lib/python3.6:$PYTHONPATH pytest \
    ./baselines/common/tests/test_serialization.py -k test_serialization
```

That test *FAILS* [TODO]. The immediate goal, however, is to get the hands going with mujoco, graphics, CUDA, and Tensorflow going.

To get the hands going on Linux, you will need the following.

2.2 Graphics Display (Ubuntu)

If graphics don't work for you, you may have to do some things in this section. I broke graphics by doing `sudo apt install libglew-dev`. To fix it, I had to chase down `glfw`, which doesn't have an obvious name. This took time to figure out: you don't want to discover all this on your own.

```
https://github.com/glfw/glfw/issues/808
https://github.com/openai/mujoco-py/issues/268
https://www.reddit.com/r/learnprogramming/comments/51u1bg/
    how_to_install_glew_on_ubuntu/
```

At one point, I had to add an untrusted `.deb` repository to `apt`, but that step no longer appears necessary. If the following doesn't work for you,

```
sudo apt update
sudo apt install libglfw3-dev
sudo apt install libglfw3
```

Then *temporarily* add the following line to `/etc/apt/sources.list` using `sudo.nano` (that's the easiest way to add the line).

```
deb http://ppa.launchpad.net/keithw/glfw3/ubuntu trusty main
```

and try again. Then comment out that line in `sources.list` because that `.deb` repo is not digitally signed and your automatic update software will stall on it. You may need to uncomment and recomment it later to reinstall `glfw`, however, so don't remove the line from the file; leave it as a reminder of what to install.

2.3 Mujoco

`https://www.roboti.us/index.html`

Install mujoco 150 and 200 (I leave that to you — there is a license involved, but everything goes in a directory named `~/.mujoco`.

2.4 Environment Variables

It's difficult to get the versions of mujoco, CUDA, Tensorflow, glfw, glew that will work with multiple applications. If you get all tied in knots, go here:

```
https://docs.nvidia.com/deeplearning/sdk/cudnn-install/
  index.html#ubuntu-network-installation
```

Add these lines to your `.zshrc`, or similar ones (in a different syntax) to `.bashrc`

```
export PATH=/usr/local/cuda-9.0/bin\
  ${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda/lib64\
  ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export LD_LIBRARY_PATH=~/.mujoco/mjpro150/bin\
  ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export LD_LIBRARY_PATH=~/.mujoco/mujoco200/bin\
  ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export PYTHONPATH=/usr/lib/tensorflow/lib/python3.6\
  ${PYTHONPATH:+:${PYTHONPATH}}
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libGLEW.so
```

Add them to PyCharm under `Run->Edit Configuration->Environment Variables` or use the `EnvFile` plugin for PyCharm.

Use PyCharm. Its debugger integration is worth the trouble.

2.5 Python

Make sure your Python is working (it must be Python 3.6, at least if you're following along with me):

```
python --version
```

Make a virtual environment. I called mine "shadow-hand-venv". Make sure it has at least this stuff:

```
pip freeze
```

absl-py==0.7.0
astor==0.7.1
atari-py==0.1.7
atomicwrites==1.3.0
attrs==19.1.0
-e git+https://github.com/rebcabin/baselines.git@1b0924#egg=baselines
box2d-py==2.3.8
certifi==2019.3.9
cffi==1.12.2
chardet==3.0.4
Click==7.0
cloudpickle==0.8.0
Cython==0.29.6
dill==0.2.9
filelock==3.0.10
future==0.17.1
gast==0.2.2
glfw==1.7.1
grpcio==1.19.0
-e git+https://github.com/rebcabin/baselines.git@1b0924#egg=gym
h5py==2.9.0
idna==2.8
imageio==2.5.0
joblib==0.13.2
Keras-Applications==1.0.7
Keras-Preprocessing==1.0.9
lockfile==0.12.2
Markdown==3.0.1
mock==2.0.0
more-itertools==6.0.0
mpi4py==3.0.1
mujoco-py==2.0.2.0
numpy==1.16.2
opencv-python==4.0.0.21
pbr==5.1.3
Pillow==5.4.1
pkg-resources==0.0.0
pluggy==0.9.0
progressbar2==3.39.3
protobuf==3.7.0
py==1.8.0
pybullet==2.4.8
pycparser==2.19
pyglet==1.3.2
PyOpenGL==3.1.0
pytest==4.3.1
pytest-forked==1.0.2
python-utils==2.3.0
requests==2.21.0
scipy==1.2.1
six==1.12.0
tensorboard==1.9.0
tensorflow==1.9.0
tensorflow-estimator==1.13.0
termcolor==1.1.0
tqdm==4.31.1
urllib3==1.24.1
Werkzeug==0.14.1

Activate your environment:

```
source ./shadow-hand-venv/bin/activate
```

Make sure again that Python 3.6 is working with an f-string example (f-strings don't work in Python 3.5)

```
import time
return f"Hello, today's date is {time.ctime()}"

Hello, today's date is Sun Apr  7 12:25:07 2019
```

Run Emacs in the background from a terminal where that environment is active. If you start Emacs without the environment, you won't be able to run the Python code below. Here is how I do it.

```
$ nohup ~/usr/bin/emacs-26.1 &> /dev/null &
```

2.6 See Hands Run; Run, Hands, Run!

If you use a `:session` header in the following, mujoco will hang, so don't.

Give it a go, and best of luck:

```
import gym
env = gym.make("TwoHandsManipulateBlocks-v0")
# env = gym.make("CartPole-v1")
# env = gym.make("Zaxxon-v0")

observation = env.reset() # BOGUS! env.reset returns zoquetes!
for _ in range(25):
    env.render()
    action = env.action_space.sample() # your agent here (this takes random actions)
    observation, reward, done, info = env.step(action)
    if done:
        observation = env.reset() # BOGUS! env.reset returns zoquetes!
env.close()
```

2.7 User Interface

2.7.1 Two Mujoco Windows

[2019-03-28 Thu 09:10] getting Mujoco to show two windows.

Suspending this out of bias-for-action. Turns out to require many changes inside `mujoco_py`. Mujoco assumes it controls one screen, one process. We can implement two mujocos, but it's more work. For now, I will put two hands, two cubes in one mujoco process.

Here is a comment recording my problems with it.

```
def render(self, mode='human', width=DEFAULT_SIZE, height=DEFAULT_SIZE):
    self._render_callback()
    if mode == 'rgb_array':
        self._get_viewer(mode).render(width, height)
        # window size used for old mujoco-py:
        data = self._get_viewer(mode).read_pixels(width, height, depth=False)
        # original image is upside-down, so flip it
        return data[::-1, :, :]
    # [[[ bbeckman --- human mode is ignoring width and height. The ignoring
    # happens way down deep in the mujoco layer. mujoco_py.MjViewer ignores
    # the width and height from here and opens a window full-screen. ]]]
    elif mode == 'human':
        self._get_viewer(mode).render(width, height)
```