CMPUT 652, Fall 2019 - Assignment #3

November 7, 2019

Version:	1.0. Nov 7, 2019
Due:	November 21, 23:59 MST
Submission:	Email a single zip file to both armahmood@ualberta.ca and sherstan@ualberta.ca
Questions:	Post questions on the eclass discussion forum, not email.

In this assignment you will program the PPO algorithm (Schulman et al., 2017) from scratch and run it on a Dynamixel servo. The goals are: 1) to gain experience implementing a state-of-the-art policy gradient algorithm on a real robot, 2) to think through the practical concerns of data synchronization and the interaction between action and sense loops on a real robot. There are three primary components for this assignment: 1) completing the ReacherEnv 2) PPO implementation and testing 3) Creating a PID baseline.

1 Setup

You are to use the Dynamixel servo kit you were provided in class. You are free to use the SenseAct driver that we previously used in the lab: `dxl_utils.py`. Other than that you are expected to program everything from scratch. You may NOT use OpenAI baselines or any other implementations of PPO. Please use PyTorch.

Refer to https://eclass.srv.ualberta.ca/mod/resource/view.php?id=3853364 for setting up an environment for the Dynamixels. I do not intend to run your code, so you are free to create your python environment as you like. However, you will need at least: pytorch, numpy, matplotlib, pyserial, tensorboard.

You can modify the communications delay between your computer and the USB device as follows (thanks to Kris De Asis):

```
cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
>> 16
sudo apt install setserial
setserial /dev/ttyUSB0 low_latency.
cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
>> 1
```

On one of my computers this noticeably reduced the variability in my cycle time.

There are two types of drivers. The ctypes driver comes from DynamixelSDK and is the default returned by `get_driver`. To use the pyserial driver call `get_driver(False)`. While it has been recommended that we use the ctypes drivers I have had difficulties with it. On one of my computers it regularly causes segmentation faults. Also, after setting the low latency as described above I observed similar variation in my cycle times using either pyserial or ctypes drivers. In both cases my PPO agent performed similarly. Additionally, there are other issues which can impact performance. On my home computer my system would occasionally lock up and produce cycle times up to 500s, yes that's close to 10 minutes. Obviously, if this happened too frequently then nothing would work, but despite this issue, overall my agent was still able to learn—these periods appear as just a blip and because they only present for a small number of training transitions they only have a limited affect on the weights.

2 Deliverables

2.1 Reacher Environment (20 %)

Your first task is to finish writing the class `ReacherEnv` which is found in `env.py`. This class is meant to be compatible with Gym environments. This task is meant to be the same as SenseAct's reacher task. At the start of every episode the motor is reset to position 0 and a new target is selected in [-75, 75] degrees.

The Dynamixel motors operate in one of three modes: position (joint), speed, or torque. You are free to choose any of the methods. Please write which method you chose. Unlike assignment #2 I will not be trying to run your code or restore it. Thus, you are more free to change the class as you see fit. However, I expect it to still conform to the Gym interface specification. I also expect the reward function to be unmodified.

You are responsible for implementing the following:

1. `_reset_motor`: This function is called every time the environment is reset. It needs to move the motor to the start position and then switch to your chosen control method.

2. `self.observation_space`: Define the observation space here.

3. `_step`: This function is called every time the environment's main `step` function is called. This function takes an action, executes it and returns `observation, reward, done, _`, where _ is just extra info the function may return (it must be included for compatibility with Gym). Thus, this function expects to execute an action and only return once it has observed the outcome of the action. As discussed in class it is here that you must maintain your cycle time. Statistics of the cycle time of the `step` function are already maintained and reported. With both ctypes and pyserial drivers and the low latency setting described previously I was able to achieve a standard deviation in cycle time of about 1 ms (until such time that my computer would freeze up).

4. `read`: Reads the registers of the Dynamixel motor.

5. `_make_observation`: This function takes in a Dynamixel observation dictionary, which you are responsible for writing elsewhere. You can then use any fields from the dxl observation. You do not need to use them directly, you can calculate different features as you see fit. You may also use additional fields stored in the environment such as the target or last action. Recall that it is a good idea to normalize your features.

2.2 PPO (70 %)

To implement PPO complete the following files: network.py, ppo.py, ppo_agent.py, test_ppo_agent.py. Unlike the previous assignment you have much more flexibility to modify the code as you see fit. However, I expect it to be clean and easy to follow. Please provide comments to help me understand what you are doing.

For simplicity we are performing all of our learning in an offline manner. So we will collect a batch of transitions, then stop the motor and run our PPO update on the collected batch. Then we clear our batch and collect a new batch. Each batch will be composed of transitions from many different episodes.

Each episode is the same length $\sim$2s. From your selected cycle time the env will calculate the number of steps per episode and use this value to determine the number of transitions in the episode. To simplify things further we will select batch sizes which are a multiple of the number of steps in each episode. That way you will only learn on complete episodes. It is up to you to correctly count transitions and ensure that your batch is set correctly. To say that again, if each episode is N timesteps in length, you should select a batch size $b = N * m$, where m is the number of episodes.

2.2.1 PPO Training Code (30 marks)

We have provided `ppo.py` and `ppo_agent.py` as very light skeletons. You are free to change these as you like, but for the agent I will be looking for at least the following three functions: `compute_return`, `step`, `learn`.

There are 9 steps to go from REINFORCE to PPO. Implement as many as you can. In your code please add a hash tag for each step, example: Add the comment`#PPO1` by your loss function to indicate you have dropped the γ^t. You need to implement all of these for full marks, but can get partial marks for those you do complete.

1. **#PPO1** Drop γ^t from the update. The version of the update given in the RL textbook is:

$$\theta \leftarrow \theta + \alpha\gamma^t \nabla ln\pi(A_t|S_t, \theta).$$

   While we continue to use bootstrapping in computing the return, we drop the γ^t term from the update.

2. **#PPO2** Subtract a baseline. Like REINFORCE with Baseline (Section 13.4 of the RL text), we subtract a baseline from the return. Here our baseline is the value estimate—thus, we compute the advantage function:

$$H_t = G_t - V(S_t)$$

3. **#PPO3** Batch update over multiple episodes. In the original PPO paper they have many agents operating simultaneously, using the same policy. They collect samples for a batch update. We can reproduce the same thing with a single agent by running the agent over numerous episodes to collect our batch before performing a learning update.

4. **#PPO4** Minibatch updates. For each transition in the batch the return and advantage are first computed. Then the batch is shuffled and divided into minibatches of approximately equal size. For a given minibatch the objective and gradients are computed and applied. Training is done over all the minibatches—thus, all transitionsa in the batch are trained.

5. **#PPO5** Multiple Epochs. The agent performs minibatch updates multiple times. So for each epoch the batch is divided into minibatches (again each is populated randomly) and the network is trained on each minibatch.

6. **#PPO6** After each minibatch update the policy of our network is slightly different than the one used to generate the data. To account for this change in the probability of drawing each of the actions will apply an importance sampling ratio, ρ, and use the surrogate objective (k indicates the epoch number and b the minibatch number):

$$\mathcal{L}_{k:b} = -\mathbb{E}_b[\rho_t G_t^k] \tag{1}$$

   where:

$$\rho_t = \frac{\pi_\theta(A_t^k|S_t^k)}{\pi_{\theta_{old}}(A_t^k|S_t^k)}. \tag{2}$$

   ρ should be computed at the time you process the minibatch, not at the start of each epoch.

7. **#PPO7** Use λ-return:

$$G_t^\lambda = R_{t+1} + \gamma[(1-\lambda)V(S_{t+1}) + \lambda G_{t+1}].$$

   We can use this return for updating both the policy and value estimate. Note that this is a bit of a departure from the approach given in the PPO paper. You are free to use either.

8. **#PPO8** Normalize the advantage of the batch, prior to update. Only do this once. Here we simply subtract the mean of the advantage and divide by its standard deviation.

9. **#PPO9** Apply a penalty for change in policy. This is given as follows:

$$\mathcal{L}_{k:b} = -\mathbb{E}_b\Big[min[\rho_t H_t, clip(\rho_t, 1-\epsilon, 1+\epsilon)H_t]\Big]. \tag{3}$$

Below are guidelines for parameter settings you should use:

- **batch_size**: Choose a batch size that is a multiple of the episode length and as close to 2048 as possible.
- **mini_batch_div**: Try 32.
- **epoch_count**: Try 10.

- **baud**: Use 1M.
- **cycle_time**: Somewhere between 30-60 ms (0.03-0.06s) is a good idea.

As I evaluate your code I will be looking for a few things, some are more subjective than others, but here are some general guidelines.

- Is the code easy to follow? Commenting will help here, so will breaking code into smaller functions.
- Are the pieces of PPO implemented correctly? There are multiple variations for implementing these pieces. If yours is reasonable, it is fine; it doesn't need to match mine. In your writeup explain any design choices that you make.
- Are the order of operations correct/reasonable? In particular looking at the `step` function in ReacherEnv, and your main training loop in `ppo.py`.
- Does the environment still maintain the Gym interface?
- Does the code match your pseudocode?

2.2.2 Pseudocode (5 %)

Write up your pseudocode for the algorithm you actually implement in LaTex.

2.2.3 NeuralNetwork (5 %)

Create your model in `network.py` You are free to modify this as you need. It is recommended that you create separate networks for policy and value. That is, the policy and value networks should not share parameters. Both can be placed inside a single module or inside two different ones. The choice is yours. Your network forward pass should work correctly whether you pass in a single state input or a batch of them.

The policy should be parameterized by a normal distribution (torch.distributions.normal.Normal). To be clear your policy network should output the mean and stddev which are then fed into the Normal which can then be sampled. Care should be given to how your network outputs the stddev and how it is initialized. We require $\sigma > 0$. This can be achieved by any number of ways. Say that f is the output of our network then you could do: f^2, $abs(f)$, $log(1 + e^f)$, $sigmoid(f)$, e^f. Large values of stddev will encourage exploration, but will produce high variance, which may make learning difficult. Additionally stddev can be implemented in either a state dependent way—as the output of your network—or as a single fixed parameter that is independent of state (this is how OpenAI baselines does it). Stddev can even be fixed, but this will limit final performance.

2.2.4 Unit Test (5 %)

At some point in almost every project I've run I stop and write unit tests, and I always regret that I didn't do it right at the beginning—it would have saved me so much headache. At a minimum I want you to write a unit test to evaluate your function which computes the return for your agent. Write this in test_ppo_agent.py. You are welcome to write additional tests, but I am only specifically requiring a test for the computation of the return. Again, please write this code clearly and add comments to tell me what you are testing for.

2.2.5 Plot: Average Episode Returns (25 %)

You will need to create a plot of the average episode returns vs. timesteps. We simplify this again by taking your averages over the update batches. Thus, your averages will be in line with your learning updates and will always be taken over the same number of episodes. For a given data point you will take the average of the returns in the batch at that point in time.

You will need to save this information to disk for 5 runs (at least). Create a plot of this data. The plot should show: 1) the mean over all the runs. This should be shown in the foreground. 2) The standard deviation across the runs shown as a lightly shaded region. 3) Each of the runs themselves. These lines should be visible against the shading and should be easy to separate from the mean. The y-axis should be the average episode return and the x-axis should list both

transition count and wall clock time. To be clear wall clock time will include the time for learning and resets. This is how long it actually took to run. Runs should be at least 50k timesteps in length, but I found my performance to start plateauing around 100k.

Note: You can compare your results to those published in Mahmood et al., 2018 by dividing your results by 0.04. My results matched those in the paper fairly well.

Here's an idea of what I'm looking for:

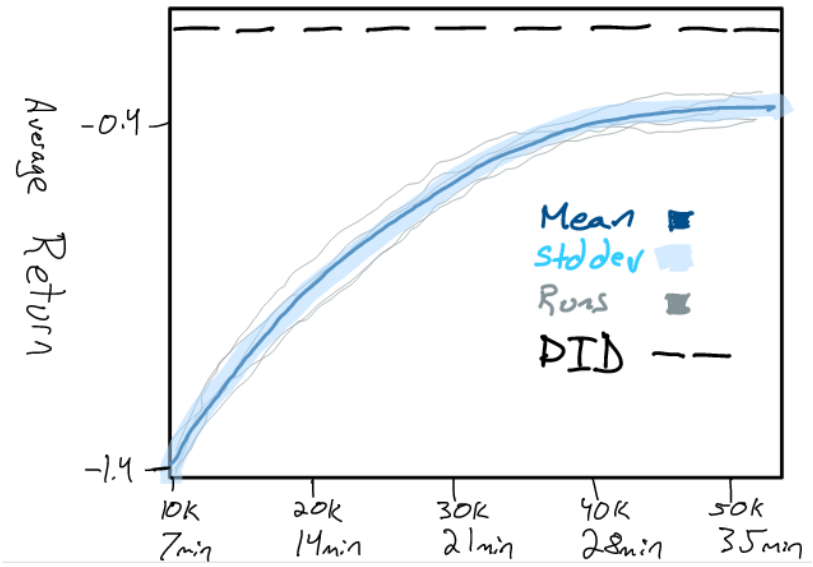


Figure 1: Average Return

2.3 PID (10 %)

The final task for this assignment is to create a PID controller to perform the same job as your PPO agent. Let's recap PID control. At time t the agent observes the target position, T_t and current position P_t. It computes error: $E_t = T_t - P_t$. It also computes the integral of the error from time 0 to the current time, $\int_0^t E_t dt$, and the derivative, $\frac{dE_t}{dt}$. Each of these terms is then multiplied by a constant scalar and summed to produce the action:

$$A_t = K_p E_t + K_i \int_0^t E_t dt + K_d \frac{dE_t}{dt}.$$

This action is sent to the motor in whatever control mode you are operating in. For example if you are using torque mode then A_t is a torque commmand written directly to the motor and the scalars have units like $[K_p] = \frac{mA}{rad}$, where position is given in rad and torque is given in mA.

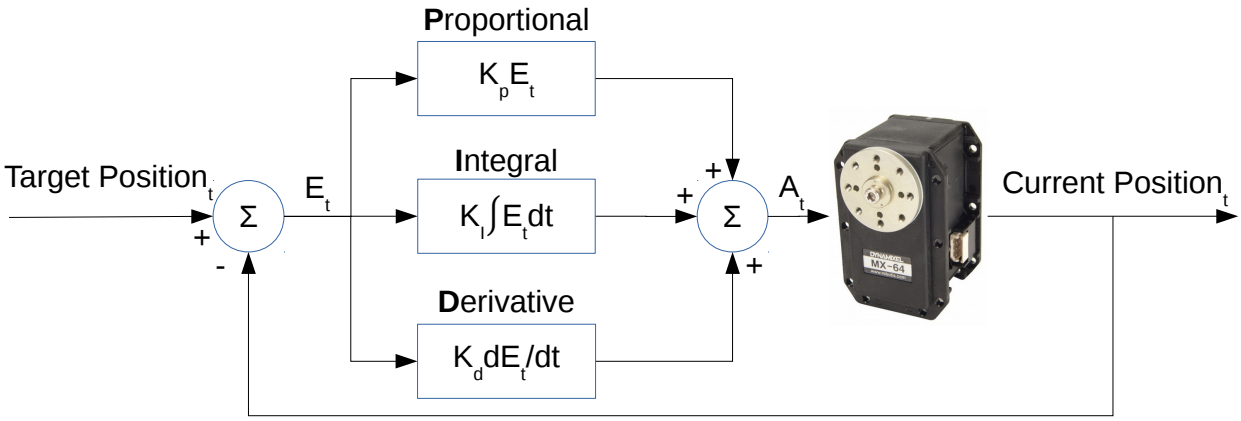


Figure 2: PID Control

Fill in the details of `pid.py`. This file has a class PIDReacherEnv which will inherit the ReacherEnv you created and return observations: [current position, target position]. Your PID controller must use the same control mode (position, speed, or torque), the same cycle time, and the same action limit (e.g. 100 mA for torque) as your PPO implementation.

It is up to you to select your scalar values. Wikipedia describes a process you can use https://en.wikipedia.org/wiki/PID_controller#Manual_tuning. Try to get a reasonable curve, which achieves the target quickly, and without too much overshoot or oscillation. I will not be grading you on optimality of your parameters, but I would like you to spend a bit of time on this. Play with the parameters. Can you cause oscillation or overshooting? Can you stop oscillation and overshooting?

1. (3%) Compute the average return over 100 episode (and stddev). Report those numbers in your write up. Add these to your previous PPO plot as a fixed horizontal line (`axhline`).

2. (3%) Response plot. Provide a plot of a single episode. You should indicate the target position and plot the current position as a function of timesteps.

3. (4%) Submit your code.

2.4 Guidance

1. You will need to limit the actions sent to the motor. For torque the original SenseAct implementation uses [-100, 100]. For speed you will need to determine your own values. For position based control the motor itself will impose limits of $[-\pi, \pi]$. Some ways of providing these bounds are better than others (Hint: some ways may interfere with the probabilities you use for your updates). In your code mark the location where you have implemented these bounds with **#LIMITS**.

2. When you look at the objective defined in Eq.(3) which variable does the gradient need to flow through? No gradients should flow through the other variables.

3. When you parameterize a policy using a Gaussian distribution the stddev controls the amount of stochasticity in the action selection. This is the exploration of your algorithm. Large exploration will lead to high variability, which can make learning very difficult. When stddev is a variable in a policy gradient method it will behave in certain ways with respect to policy performance. When the policy is doing poorly, like in early learning, stddev will increase and when the policy is doing well the stddev will shrink. In the limit stddev will shrink towards zero. In reality it can never achieve this and if the correct action is 0.0 then some other mechanism is needed to ensure this. You should think about how/if you want to control the magnitude of stddev. Some methods, such as parameterizing by a sigmoid will naturally keep stddev bounded. Some developers choose to clip the value. In my own code I simply initialized the value to be reasonable (around 0.5) by initializing the network bias for the stddev output appropriately.

Below are stats from my own experiments. We see that initially the stddev increases and then consistently decreases. Also the average magnitude of the means increases with time. It is no longer uncertain which direction to go, which would be indicated by 0.0, but now has a clear directionality.

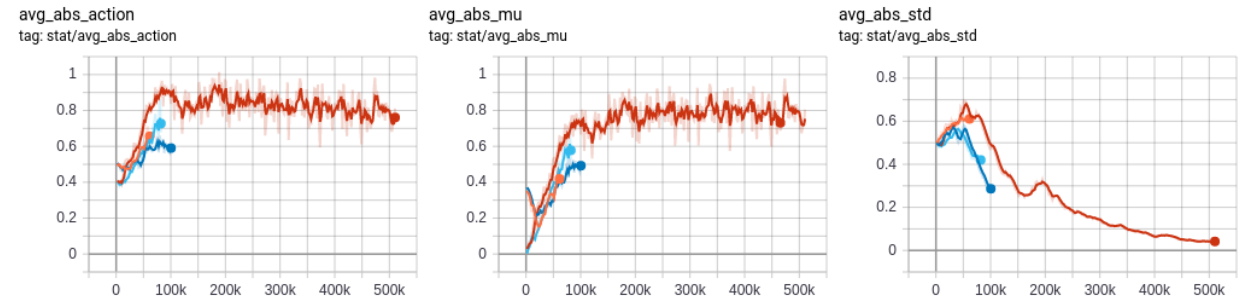


Figure 3: Stats from my own runs.

4. There are certain things to watch for that will help indicate that your system is learning. One I found helpful was to watch the average value loss over time. If the value loss was consistently decreasing it was an indicator that my value network was working, even if the policy wasn't or didn't yet show improvement. So too the bias value of the value estimator would decrease.

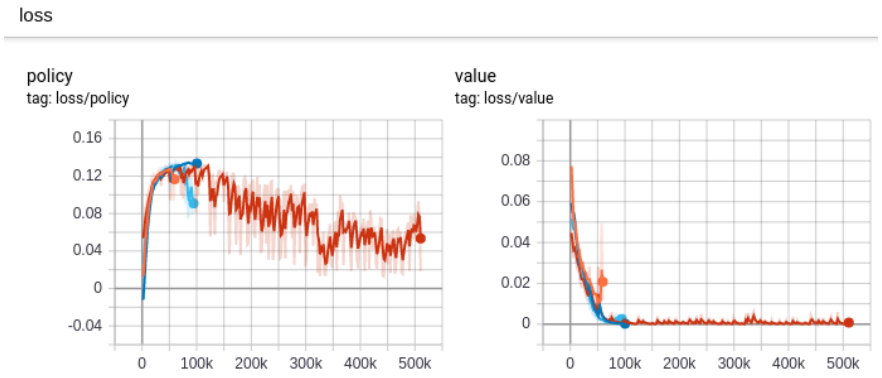


Figure 4: Losses

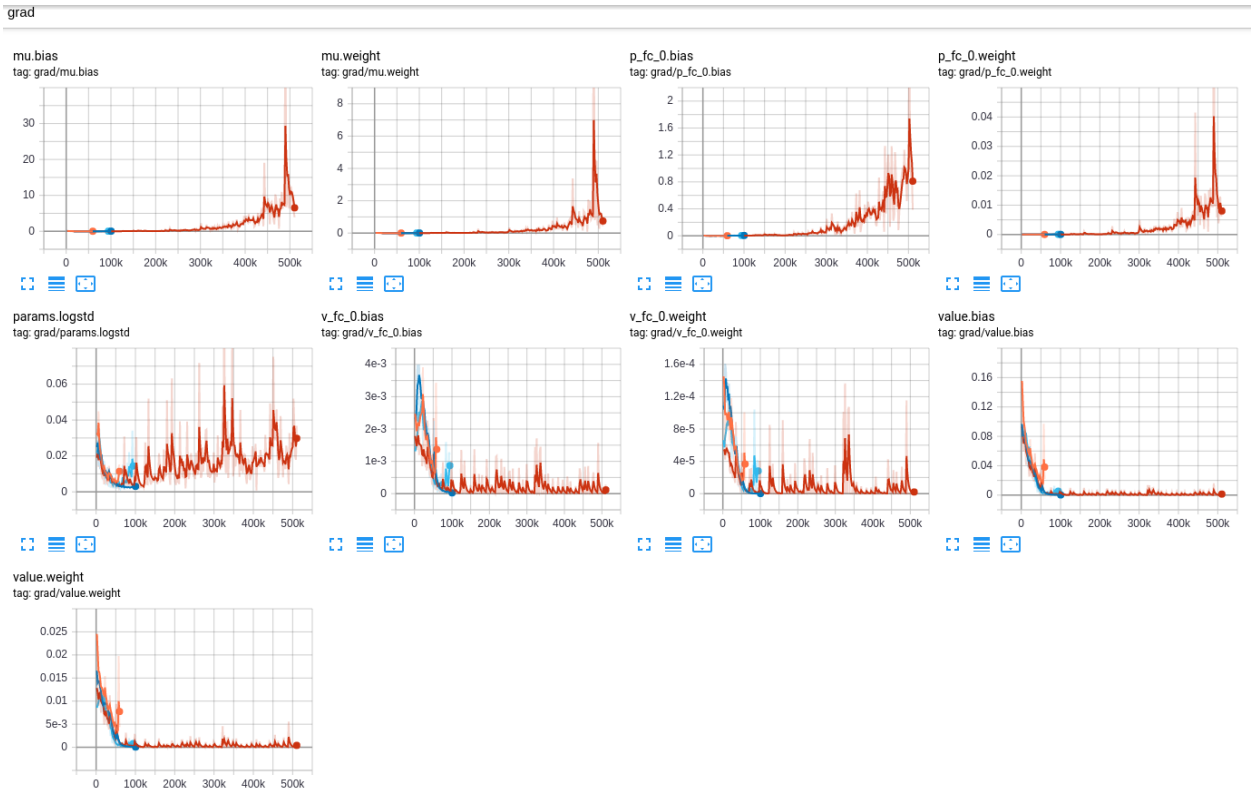


Figure 5: Grads

5. For reference here are plots of my own results. Some of these runs were done with ctypes drivers and some with pyserial. Notice that the results are fairly consistent. Random seeds were used for each run. If you divide these results by 0.04 they match the results reported by Mahmood et al., 2018 fairly well up to 50k. The orange run used 10 epochs, while the others used 20—there doesn't seem to be much difference.

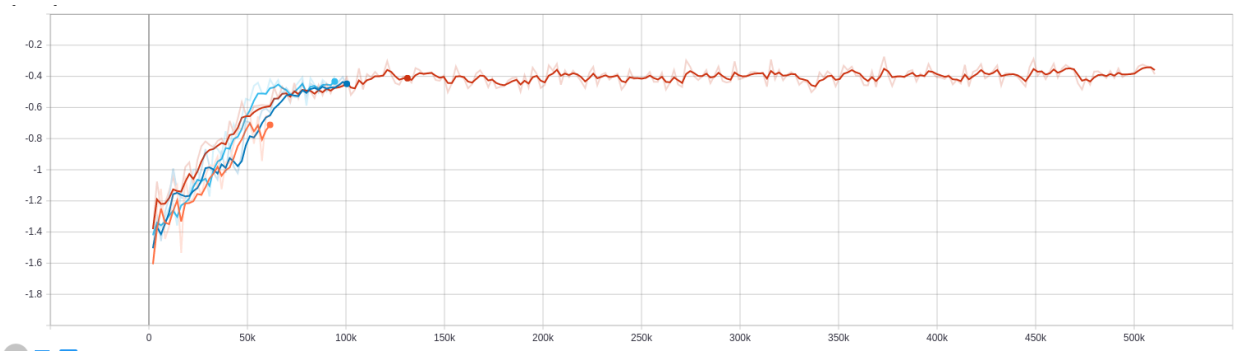
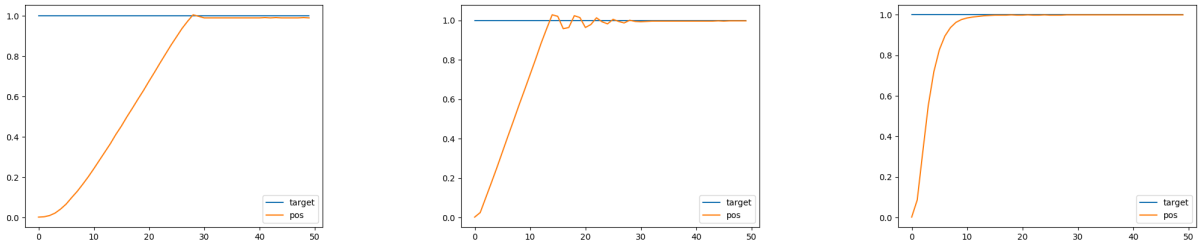


Figure 6: Returns

My model used two fully connected layers of 64 nodes for both the policy and value networks (so do the ones in SenseAct). I initialized the bias for the policy mean to zero. I used a single stddev parameter which was independent of state (like SenseAct). My stddev was computed by e^f where f is the network variable. I initialized the stddev to $log(0.5)$ to give an initial stddev of 0.5.

I used torque control with a cycle time of 40ms, max torque 100, a batch size of 2050, mini_batch_div of 32, epoch_count of 10 and 20, $\gamma = 0.99$, $\lambda = 0.95$, $\epsilon = 0.2$. I used an Adam optimizer with default settings. For now I'll keep my observation vector to myself, but I will say that it was simpler than the one used by SenseAct,

but also less general in that I engineered it a bit more. Using the same cycle time, same max torque my PID implementation, which also used torque control, achieved an average return of around -0.37 over 100 episodes. Removing the torque limit I was able to achieve an average of -0.14. For position based PID I was able to achieve -0.19 and for speed control PID with no current limits I achieved -0.07. By far the speed control based PID was the easiest to tune and produced the smoothest curves.



(a) PID Torque (max torque=100) (b) PID Position (c) PID Speed

Figure 7: PID responses for a fixed target at position 1, starting from position 0.

3 Submission Checklist

I will try to mark each part independently. Example: evaluating your plot of PPO will not depend on your code.

- ☐ **Report** For each of the sections above I want you to provide a **short** write up explaining what you did and specifically highlighting any design choices you made and parameter settings that you used. Your write up should be in PDF format. I do not require your LaTex code.
- ☐ **4.1 Reacher Environment** Completed `env.py`
- ☐ **4.2 PPO**
  - PPO implementation: `network.py`, `ppo.py`, `ppo_agent.py`.
  - Report cycle time. I will NOT be grading you on this—there was too much weirdness on my own machine.
  - Network: `network.py`
  - Unit test of return: `test_ppo_agent.py`
  - Pseudocode.
  - Plot averaging runs of 5 seeds.
- ☐ **4.3 PID**
  - PID implementation: `pid.py`
  - Plot of one run.
  - Report average return.

References

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., & Bergstra, J. (2018). Benchmarking Reinforcement Learning Algorithms on Real-World Robots. (CoRL, pp. 1–31). Retrieved from http://arxiv.org/abs/1809.07731

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv*, *1707.06347*.