6.100 Spring18 Project report

Wilson Li
{litianyi@mit.edu}
May 17,2018

1.Poject discription

Augmented Reality and Machine Learning for 3D Printable Robots:

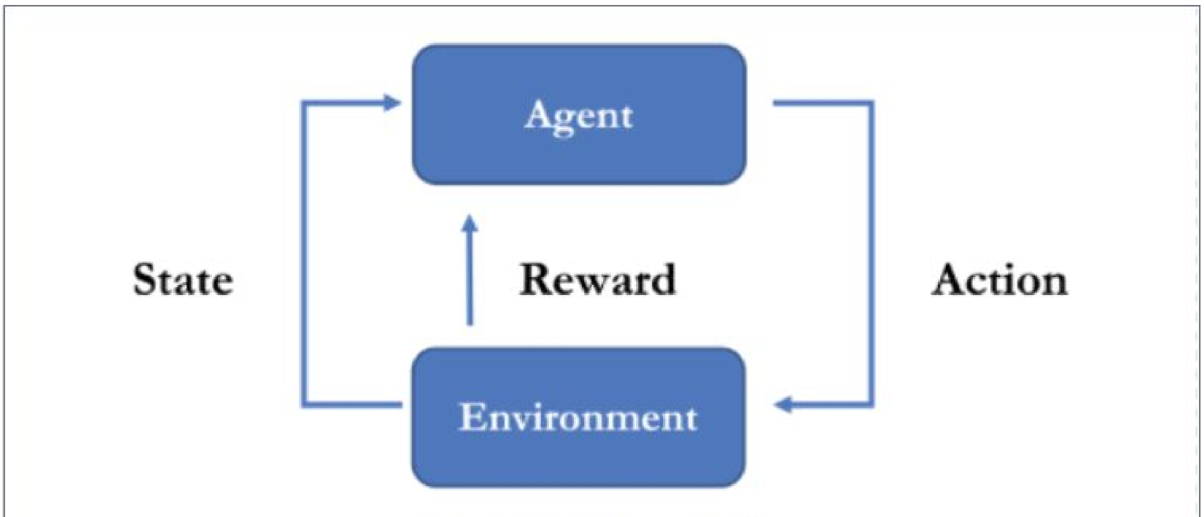
We are working on developing an augmented-reality based environment for in-situ design of 3D printable robots. This work is part of the larger printable robotics initiative at CSAIL. Augmented reality allows for the interactive design of objects, allowing users to test simulated designs directly in target environments. Robots are a particularly interesting application. Robots are fully dynamic and possess very many degrees of freedom.This both makes their interactions with their environments more complicated to model, as well as making it difficult to interactively modify motions using traditional input modes. This project will involve the development of a software stack for simulating, interacting, and optimizing robots, and will ideally tie into a number of ongoing research efforts in our lab. This project will tie in with efforts in our lab for coupling results in reinforcement learning with design.

2.Context

Reinforcement learning:Reinforcement learning can be viewed as a form of learning for sequential decision making that is commonly associated with controlling robots (but is, in fact, much more general). Consider an autonomous firefighting robot that is tasked with navigating into an area, finding the fire and neutralizing it. At any given moment, the robot perceives the environment through its sensors (e.g. camera, heat, touch), processes this information and produces an action (e.g. move to the left, rotate the water hose, turn on the water). In other words, it is continuously making decisions about how to interact in this environment given its view of the world (i.e. sensors input) and objective (i.e. neutralizing the fire). Teaching a robot to be a successful firefighting machine is precisely what reinforcement learning is designed to do.

More specifically, the goal of reinforcement learning is to learn a policy, which is essentially a mapping from observations to actions. An observation is what the robot can measure from its environment (in this case, all its sensory inputs) and an action, in its most raw form, is a change to the configuration of the robot (e.g. position of its base, position of its water hose and whether the hose is on or off).

The last remaining piece of the reinforcement learning task is the reward signal. When training a robot to be a mean firefighting machine, we provide it with rewards (positive and negative) indicating how well it is doing on completing the task. Note that the robot does not *know* how to put out fires before it is trained. It learns the objective because it receives a large positive reward when it puts out the fire and a small negative reward for every passing second. The fact that rewards are sparse (i.e. may not be provided at every step, but only when a robot arrives at a success or failure situation), is a defining characteristic of reinforcement learning and precisely why learning good policies can be difficult (and/or time-consuming) for complex environments.



The reinforcement learning cycle.

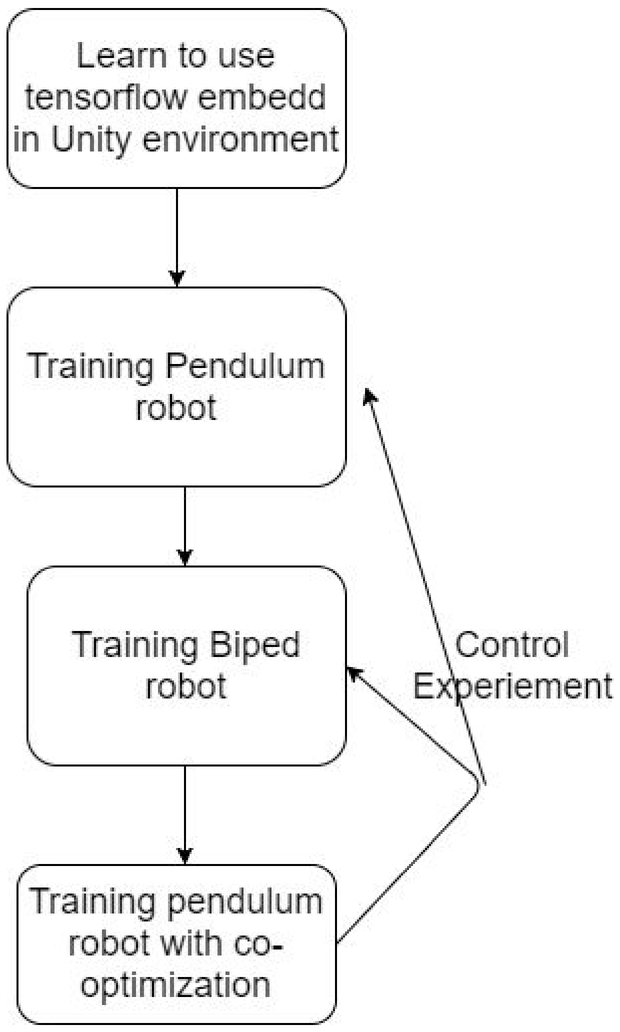
Tensorflow:TensorFlow is an open source library for performing computations using data flow graphs, the underlying representation of deep learning models. It facilitates training and inference on CPUs and

GPUs in a desktop, server, or mobile device. Within ML-Agents, when you train the behavior of an Agent, the output is a TensorFlow model (.bytes) file that you can then embed within an Internal Brain. Unless you implement a new algorithm, the use of TensorFlow is mostly abstracted away and behind the scenes.

One component of training models with TensorFlow is setting the values of certain model attributes (called *hyperparameters*). Finding the right values of these hyperparameters can require a few iterations. Consequently, we leverage a visualization tool within TensorFlow called TensorBoard. It allows the visualization of certain agent attributes (e.g. reward) throughout training which can be helpful in both building intuitions for the different hyperparameters and setting the optimal values for your Unity environment.

Unity:We use Unity engine to provide physics-like environment where robots being trained exist and receive feedback from after taking certain actions.

3.Workflow


Learn to use tensorflow embedd in Unity environment
Training Pendulum robot
Training Biped robot
Training pendulum robot with co-optimization
Control Experiement

We first train pendulum and biped robot in each Unity physical environment with PPO(Proximal Policy Optimization) to achieve higher reward.After effective training,we derive certain policies.We then add the parameters about the configuration of the robot like mass into the learning input and try to retrain agents to learn more about the metadata about itself rather than just the environment feedback so that we can generalize the derived policy to more flexible configurations.We refer to this process as co-optimization.

We hypothesize that learning with co-optimization will strengthen the ability for robots to adapt to flexible configurations without having to retrain.

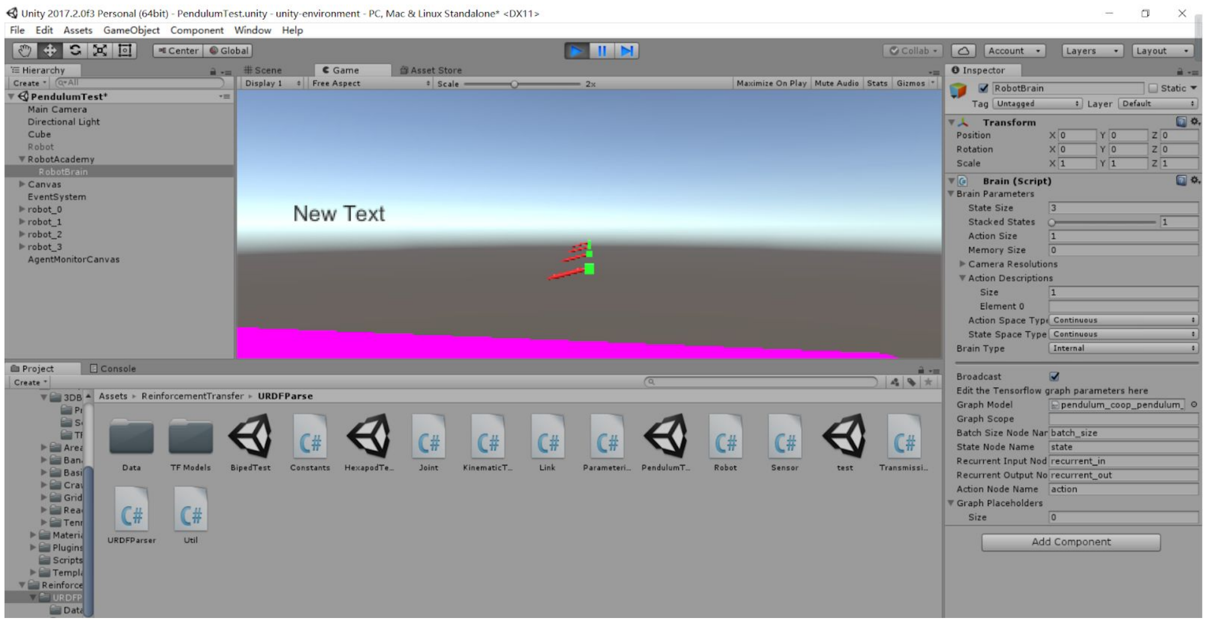
We finally test the performance of policies with co-optimization and without co-optimization under certain scenarios for comparison to evaluate our hypothesis.

4.Training without co-optimization

4.1 Training Overview

PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state.Considering that our environment provides reasonable amount of direct sensor feedback like the current position and rotational velocity to the robot,we don't need to process the frame to derive information.Thus,CNN or complicated neural network structure is not required.We maintain a three-layer network with 128 hidden units and regularization for all following trainings.Appendix lists all PPO hyperparameters and their influence on training.

4.2 Pendulum



RobotBrain
Tag Untagged Layer Default

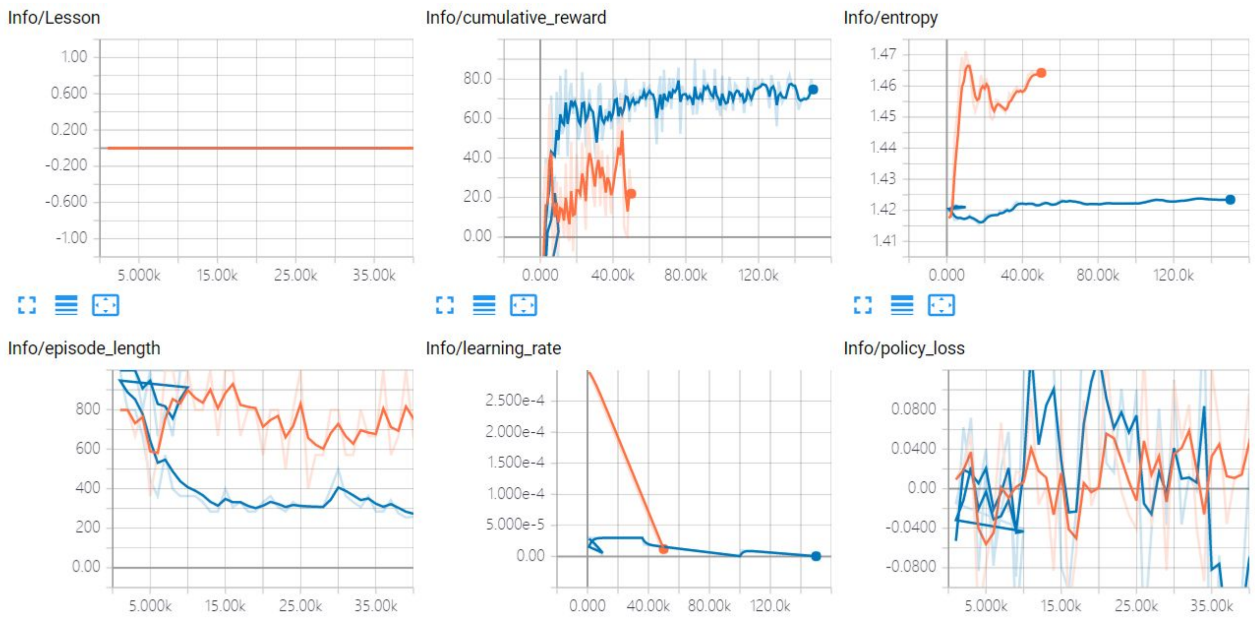
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Brain (Script)
Brain Parameters
State Size 3
Stacked States 1
Action Size 1
Memory Size 0
Camera Resolutions
Action Descriptions
Size 1
Element 0
Action Space Type Continuous
State Space Type Continuous
Brain Type Internal

Broadcast ✓
Edit the Tensorflow graph parameters here
Graph Model pendulum_coop_pendulum_
Graph Scope
Batch Size Node Nar batch_size
State Node Name state
Recurrent Input Nod recurrent_in
Recurrent Output No recurrent_out
Action Node Name action
Graph Placeholders
Size 0

Pendulum robot learns to sway its body as fast and as high as possible to achieve higher reward.Learning step is reset when it sways over its top.

(1)Hyperparameters



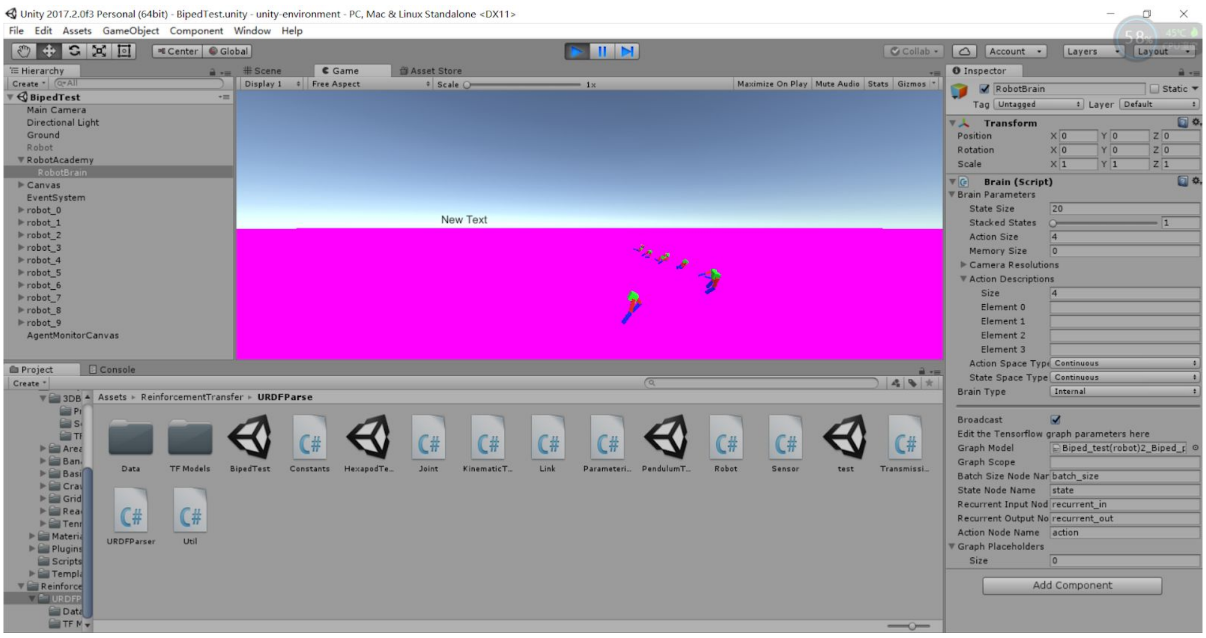
Blue curve clearly does better than orange curve.As shown in entropy,orange curve prefer exploring new actions to exploiting learned actions more than blue curve,which might cause unstable learning.We make following major changes in hyperparameters to generate more stable learning:

	learning rate	beta	eplison
Orange	3e-04	0.005	0.2
Blue	3e-05	0.01	0.1

(2)Demo

A video clip of applying trained policy(blue curve) is attached with this report.

4.3 Biped test


Unity 2017.2.0f3 Personal (64bit) - BipedTest.unity - unity-environment - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Hierarchy
Scene
Game
Asset Store
Inspector
New Text
Project
Console
Add Component

RobotBrain
Tag: Untagged
Layer: Default

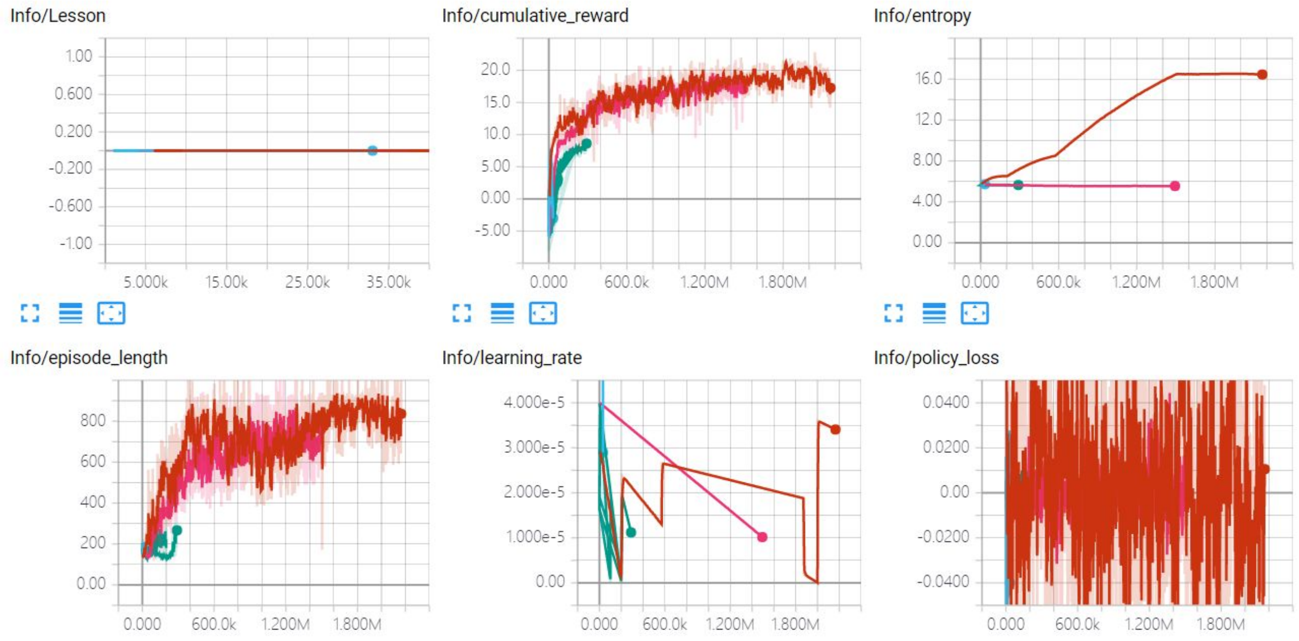
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Brain (Script)
Brain Parameters
State Size: 20
Stacked States: 1
Action Size: 4
Memory Size: 0
Camera Resolutions
Action Descriptions
Size: 4
Element 0
Element 1
Element 2
Element 3
Action Space Type: Continuous
State Space Type: Continuous
Brain Type: Internal

Broadcast ✓
Edit the Tensorflow graph parameters here
Graph Model: Biped_test(robot)2_Biped_
Graph Scope
Batch Size Node Name: batch_size
State Node Name: state
Recurrent Input Node: recurrent_in
Recurrent Output Node: recurrent_out
Action Node Name: action
Graph Placeholders
Size: 0

Biped robot learns to keep balanced and walk farther to achieve higher reward.Learning step is reset when it falls and hits the ground.

(1)Hyperparameters



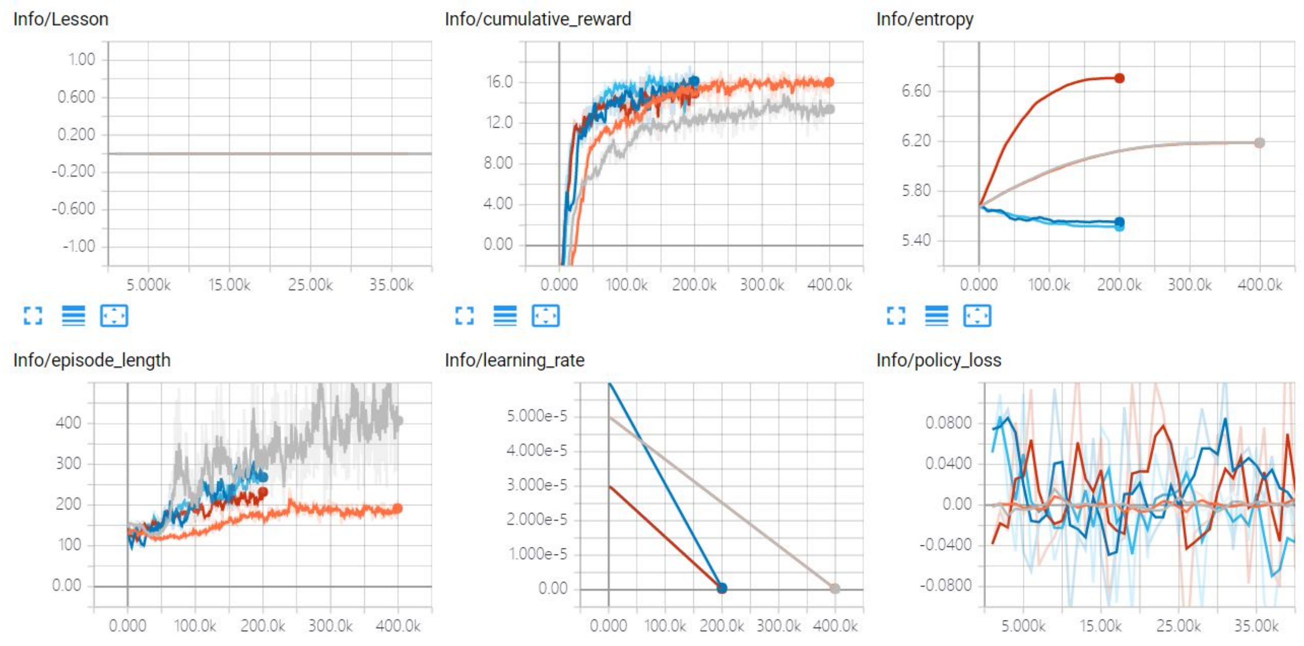
Red curve performs the best.And as the video demo shows,it's able to run for a reasonable amount of time and distance before losing balance completely.

Major hyperparameters for red curve are set as the following table:

	batch-size	buffer -size	time-horizon	epsilon	gamma	lambd	learing rate	beta
Red Curve	32	512	64	0.25	0.99	0.93	6e-5	0.0002

(2)"Fell" detection

We discover that previous training takes an enormous amount of time to converge with learning rate still not dropped to zero.After reexamining the code,we add one more condition to detect "fell" where robot's head lowered beneath half of its height.This way we can detect felling faster and reset much more frequent in response to save much training time.For the training rounds in the following curve, the learning rate all drop to 0 with only approximately 20 percent of the training time when no extra fell detection is implemented.



(3)Demo

A video clip of applying trained policy(orange curve) is attached with this report.

(4)Reward function

With fell detection,we can see that under current setting tuning parameters already lose its effects in improving performance.Limit is probably reached here.As we notice from the demo,robots tend to start with a huge step and stagger with unbalanced position until fall since our reward function only equals to the integral of velocity and doesn’t reward slow but steady gait.As suggested in other biped robot control paper,we could reward ideal ZMP position rather than just the distance and penalize high torque which could generate big step.But current policy is sufficient for us to proceed to co-optimization testing.

6.Training with co-optimization

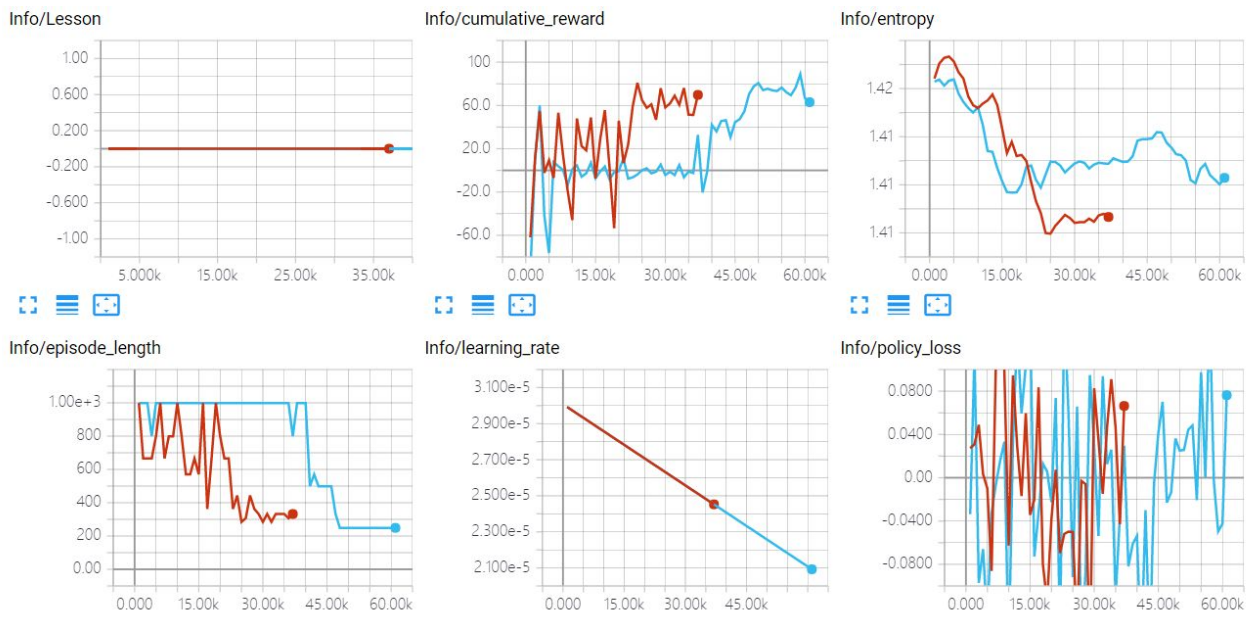
Co-optimization allows us to turn the mass of the robot into a learning input in PPO training other than the feedback from environment.We’ll initiate the following four experiments on pendulum robot with the set of hyperparameters that performs the best from previous pendulum training :

(1) Train the robot on random masses between 0.1-1.0 kg, using cooptimization.

(2) Train a robot on hand-picked values between 0.1-1.0 kg, without co-optimization.We pick 0.1 kg in this case.

(3) Take the policy learned from (1). Set the values to the same as those used in (2). Compare the rewards of (3) to those of (2).

(4) Take a robot that was trained at 0.1 kg. Change its mass to 1.0kg and see how it performs. Compare it to the robot trained from experiment (1),having its mass changed to 1.0kg.



Red curve and blue curve shown in the above graph corresponds to the learning process in (1) and (2) experiments.We can observe that with a randomized mass,co-optimization actually speeds up the training process by about 20%(convergence time are 25 mins and 34 mins relatively).Co-optimization generates a more unstable learning process before convergence.Red curve converge to a slightly higher reward than blue curve.

We'll then apply trained policies on pendulum robots and see how long it takes to reset and how the cumulative reward within one step is like.It's clear that the faster it takes to reset and higher the reward is,the better the performance of the policy is.

We run these process ten times.We take the approximated average and derive the following result:

	Reset time	Cumulative reward
Policy in (2) on 0.1kg robot	17s	78
Policy in (1) on 0.1kg robot	19s	75
Policy in (1) on 1.0kg	20s	66

Policy in (2) on 1.0kg	31s	45

Two following key observations can be made.

A:Policy in (1) on 0.1kg outperforms policy in (2) on 1.0kg both on Reset time and Cumulative reward.Since policy in (1) learns a fixed mass(0.1kg) robot and policy in (2) learns a randomized mass robot with co-optimization,it's reasonable that policy in (2) will be more adaptive and can generalize to robots with different mass while policy in (1) performs better on the same robots it trains on.This observation is consistent with our reasoning.

B:Policy in (1) on 1.0kg outperforms policy in (2) on 0.1kg both on Reset time and Cumulative reward.It shows that policy in (2) with co-optimization does generalize better than policy in (1) without co-optimization.

Except for co-optimization,two policies also differ in the mass learning pattern.To test the effect of randomized mass learning without co-optimization,we conduct an extra experiment where we train the robot on random masses between 0.1-1.0 kg without using co-optimization.And then we see how this new policy performs on 1.0kg,we derive that:

Reset time	Cumulative reward
28s	49

Clearly,policy in (1) on 1.0kg robots outperforms this policy both on Reset time and Cumulative reward.Thus,we show that it's mostly co-optimization that contribute to better generalization in observation B rather than just randomized mass.

7.Conclusion

We implement PPO to successfully train Pendulum robot and Biped robot in the unity environment.With a good set of hyperparameters,we proceed to test the effect of co-optimization on learning generalizations and show that this way of transferring knowledge about robot configuration could be effectively learned by the agent robots.

8.Future extensions

Due to the limited time scope of this project(during regular term with courses workload),we haven't test effects of co-optimization on more complex configurations of robots like Biped robots or even Hexapod robot.Besides,we could also try to embed more knowledge about the robot like size,moment of inertia,etc into co-optimization process to improve the level of generalizations.These two part of the research could be potentially conducted in the future.

Appendix

Hyperparameters in PPO

Gamma

`gamma` corresponds to the discount factor for future rewards. This can be thought of as how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. In cases when rewards are more immediate, it can be smaller.

Typical Range: `0.8` - `0.995`

Lambda

`lambd` corresponds to the `lambda` parameter used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate (which can be high bias), and high values correspond to relying more on the actual rewards received in the environment (which can be high variance). The parameter provides a trade-off between the two, and the right value can lead to a more stable training process.

Typical Range: `0.9` - `0.95`

Buffer Size

`buffer_size` corresponds to how many experiences (agent observations, actions and rewards obtained) should be collected before we do any learning or updating of the model. This should be a multiple of `batch_size`. Typically larger `buffer_size`correspond to more stable training updates.

Typical Range: `2048` - `409600`

Batch Size

`batch_size` is the number of experiences used for one iteration of a gradient descent update. This should always be a fraction of the `buffer_size`. If you are using a continuous action space, this value should be large (in the order of 1000s). If you are using a discrete action space, this value should be smaller (in order of 10s).

Typical Range (Continuous): `512` - `5120`

Typical Range (Discrete): `32` - `512`

Number of Epochs

`num_epoch` is the number of passes through the experience buffer during gradient descent. The larger the `batch_size`, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning.

Typical Range: `3` - `10`

Learning Rate

`learning_rate` corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase.

Typical Range: `1e-5` - `1e-3`

Time Horizon

`time_horizon` corresponds to how many steps of experience to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimate is used to predict the overall expected reward from the agent's current state. As such, this parameter trades off between a less biased, but higher variance estimate (long time horizon) and more biased, but less varied estimate (short time horizon). In cases where there are frequent rewards within an episode, or episodes are prohibitively large, a smaller number can be more ideal. This number should be large enough to capture all the important behavior within a sequence of an agent's actions.

Typical Range: `32` - `2048`

Max Steps

`max_steps` corresponds to how many steps of the simulation (multiplied by frame-skip) are run during the training process. This value should be increased for more complex problems.

Typical Range: `5e5` - `1e7`

Beta

`beta` corresponds to the strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase `beta`. If entropy drops too slowly, decrease `beta`.

Typical Range: `1e-4` - `1e-2`

Epsilon

`epsilon` corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process.

Typical Range: `0.1` - `0.3`

Normalize

`normalize` corresponds to whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems.

Number of Layers

`num_layers` corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation. For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems.

Typical range: `1` - `3`

Hidden Units

`hidden_units` correspond to how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the observation inputs, this should be small. For problems where the action is a very complex interaction between the observation variables, this should be larger.

Typical Range: `32` - `512`