CS 159 PROJECT REPORT: MULTI-TASK LEARNING IN CONTINUOUS CONTROL ENVIRONMENTS

A PREPRINT

Ayya Alieva, Tim Liu, Vaishnavi Shrivastava, Bianca Yang

August 4, 2019

ABSTRACT

In recent years, reinforcement learning (RL) has become the front line of the Machine learning research. However, majority of the RL algorithms suffer from instability and sampling inefficiency, which severely limits their real-world applications. RL tends to perform particularly poorly on tasks with a large number of states. One way to address this issue is to allow the algorithms to learn from a few similar tasks. Inspired by the successes of such an approach for discrete spaces, we adapted the existing Multi-task methods to continuous control environments. We show that this approach greatly reduces the training time and increases stability of the models.

1 Introduction

Reinforcement learning is typically used to accomplish specific tasks, such as automating airplanes. One of the biggest problems with this approach is that it is extremely difficult for models trained on one task to perform well on other tasks, even if the other tasks are similar to the task it was originally trained on. For example, if we train a model to recognize faces in images, the model would not perform well on related tasks of identifying people or objects within images. Another challenge is that the training data in reinforcement learning often comes from expert demonstrations. Acquiring these demonstrations for a range of tasks and environments can be difficult and expensive. In these cases sharing samples across related tasks is extremely beneficial. The samples can be used in a supportive way as well as in an adversarial way, to highlight both the similarities and differences between tasks. This would allow us to increase the models' ability to perform well on multiple related tasks while maintaining its environment-specific performance.

The idea of sharing samples to simultaneously train related tasks is what motivates multi-task reinforcement learning. In multi-task learning, we create a shared representation of the tasks and learn on all of the tasks at the same time by sharing our samples across tasks. By increasing the data available for each individual task, multi-task learning enables us to generally learn more efficiently and have greater predictive accuracy and generalizability. These are extremely valuable features for creating machine learning models that can replicate human-like intelligence. Currently, most multi-task reinforcement learning has been developed for discrete action space MDP models. However, the majority of interesting real-world applications of machine learning involve continuous action spaces. As a famous example, autonomous vehicles can represent agents that have control over continuous parameters such as acceleration and wheel angles.

In our project, we show how multi-task approach can be utilized in the continuous RL framework. There exist algorithms that focus on discretizing continuous action spaces, but they tend to underperform in high-precision tasks because slight deviations from the optimal action can lead to low utility values. Moreover, very fine discretization suffers from a dimensionality problem, as the algorithm will have to traverse all actions before picking the optimal policy. [7] One possible approach to overcome these issues is to adapt the actor-critic architecture and represent a policy followed by an agent as a continuous probability distribution (often Gaussian) over the action space. However, that usually suffers from poor sampling efficiency and high sensitivity to model parameters (e.g. learning rate). In addition, the learning processes for these algorithms tends to be unstable. While sampling inefficiency is not a problem for toy environments like Mujoco, it can provide a significant setback for some real applications when collecting new data is expensive or dangerous.

In our project, we assess the extent to which some existing multitask learning methods can alleviate the above problems for RL algorithms on continuous action spaces. As previous work exploring multi-task learning in continuous control environments is limited, with this project we wanted to focus on establishing the baseline for the future models. Below we adapt widely used discrete control methods to continuous control scenarios, and demonstrate their performance. We focused on the A2C algorithm, as it was empirically shown to perform as well as its stochastic version A3C on Mujoco environments. [8] We adapted 6 multitask algorithms to continuous A2C and tested them on 6 variants of Mujoco InvertedPendulum-v2. The specific task of the environment is to balance a pole on a cart, and the action space is a one-dimensional continous subspace that controls the force applied to the cart. Our goal was to empirically evaluate and compare the performances of these multitask techniques and to provide baselines for future work in the field. While this is just a proof-of-concept, our findings show that incorporating multi-task learning with continuous control RL could improve the sampling efficiency and stability of the resulting models.

2 Previous Work

The main work in this area is by Henderson et al. [1], who recently created a series of Continuous Mujoco Modified OpenAI gym environments. We use these environments for our experiments.

Arora et al. [2] use the actor-critic and knowledge distillation algorithms to perform multitask learning. Actor critic is an on-policy algorithm which uses actor networks to improve policies and critic networks to evaluate these policies. The algorithm tries to sample rollouts based on a current version of the policy and estimate the future reward. After training the actor-critic model, Arora et al. use it as a "teacher" policy to reduce training time for the untrained "student" policy. The algorithm captures similarity between the two policies using KL-divergence. Arora et al. showed that this approach generally reduces variance and improves learning efficiency.

3 Algorithm

In this section, we first present the details of our continuous control algorithm. Afterwards, we discuss how our continuous framework can be incorporated into known multitasking methods.

3.1 Policy Gradient Algorithm

We first give a brief review of the reinforcement learning structure and a quick derivation of the basic policy gradient algorithm. The goal is to discover the optimal policy π_θ for a Markov Decision Process, which is parameterized as follows: $\langle S, A, P, r, \rho_0, \gamma, T\rangle$, where S is a set of states, A is a set of actions, $P : S \times A \times S \to R_+$ is a transition function, $r : S \times A \to [R_{min}, R_{max}]$ is a reward function, $\gamma \in [0, 1]$ is a discount factor, and T is a time horizon.

Concretely, the objective is to is to find policy parameters θ^* that maximize the expected discounted reward: $\theta^* = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}[\sum_t \gamma^t r(s_t, a_t)]$, where τ is a trajectory $\tau = (s_0, a_0, ...)$ sampled according to the policy π_θ.

Using the standard policy gradient theorem, $\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[(\sum_t \nabla_\theta log\pi_\theta(a_t|s_t))\sum_t \gamma^t(r(s_t, a_t))]$.

Expanding the expectation operator: $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i(\sum_t \nabla_\theta log\pi_\theta(a_t|s_t))\sum_t \gamma^t(r(s_t, a_t))]$, where i indexes the rollouts.

Then the basic policy gradient reinforcement learning algorithm is the following:

1. sample $\{\tau^i\}$ from $\pi_{\theta(a_t|s_t)}$
2. $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i(\sum_t \nabla_\theta log\pi_\theta(a_t|s_t))\sum_t \gamma^t(r(s_t, a_t))]$
3. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

3.2 Actor-Critic Algorithm (A2C)

We can improve the estimate for $\hat{Q}_{i,t} = \sum_t r(s_{i,t}, a_{i,t})$ with an actor-critic structure. Define

$$Q(s_t, a_t) = \sum_t E_{\pi_\theta}[r(s_t, a_t)|s_t, a_t]$$

$$V(s_t) = E_{a_t\sim\pi_\theta(a_t|s_t)}[Q(s_t, a_t)]$$

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

$\hat{Q}_{i,t}$ can be interpreted as an estimate of the expected reward for taking action $a_{i,t}$ in state $s_{i,t}$, while $Q(s_t, a_t)$ is the true expected reward for taking action a_t in state s_t. In this framework, $V(s_t)$ is then the total expected reward from state s_t and $A^\pi(s_t, a_t)$ is the impact of action a_t on the future reward.

Now the gradient and the expected reward can be rewritten as

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}(\tau)[(\sum_t \nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t}))(Q(s_{i,t}, a_{i,t})) - V(s_{i,t}))]$$

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + E_{s_t+1\sim p(s_{t+1}|s_t,a_t)}[V^\pi(s_{t+1})]$$

Thus, we can write $A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$. To somewhat reduce variance, in our implementation we first rollout the policy for at most $t_{\max}$ steps, and use the empirical discounted reward in our calculations. Combing the equations above, the updated actor-critic algorithm (with reward discounting) becomes:

1. sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$
2. estimate $\hat{V}^\pi(s)$ (value function approximation) with $r + \gamma\hat{V}^\pi(s')$
3. evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma\hat{V}^\pi(s'_i) - \hat{V}^\pi(s_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta log\pi_\theta(a_i|s_i))\hat{A}^\pi(s_i, a_i)$
5. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

We used this algorithm as the baseline for experiments, and as a starting point for our multitasking methods. Specifically, we use feed-forward neural networks to approximate π_θ and V_{θ_v}. As we are working in the continuous action space, we parametrize π_θ by a Normal distribution with mean μ and variance σ^2. Thus, our feed forward neural network has three heads (mean, variance, state-value function), and given a state s, an action is sampled according to $N(\mu, \sigma)$. In order to discourage convergence to sub-optimal policies, we also introduce an entropy penalty. The entropy is minimized when the probability distribution is uniform, and thus subtracting entropy from the loss encourages exploration.

3.3 Multi-tasking

We adapted our above algorithm to 5 different multi-tasking frameworks, and empirically tested their relative performance on continuous Mujoko environments. The rest of the paper is structured as follows: first, we discuss each of multi-tasking methods considered, and briefly describe the simulation environments we used for testing. Then, we present our findings and discuss the implications of our results.

3.3.1 Hard parameter sharing

In hard parameter sharing, a single neural net learns to solve multiple tasks. The state feeds into a shared layer, and the shared layer feeds into multiple additional layers that correspond to each task. Thus, the number of heads on the neural net is dependent on the number of tasks. For the continuous setting, each task is determined by the parameters of the chosen probability function class. In our case, we used a μ and a σ head that the actor samples from and a dedicated value head for the critic.

During training, the two environments are sampled from on an alternating basis. The rewards for the two environments are recorded and used to update the weights of the neural net. Hard parameter sharing involves the two tasks using the same shared layer, but with different additional layers stacked on the shared one.

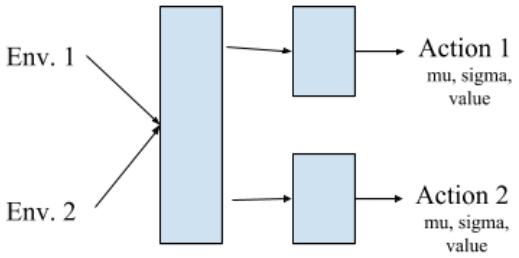


Figure 1: Hard parameter sharing. The various environments have a shared layer which feeds into multiple heads. The heads are sampled from to determine an action.

3.3.2 Soft parameter sharing

In soft parameter sharing, each task has a dedicated neural net that generates its own action from its own environment. However, the neural nets are constrained so that the weights in the same layers of different nets are similar to each other. An L_2 normalization term, added to the loss function of the neural nets during training, accomplishes this. An L_2 norm term is added for every pair of weights within every layer of the networks. In order for soft-parameter sharing to work, we need the networks for the different tasks to have nearly identical structure, so that we can have similar weights between networks. Similar to regularization, the term constrains the weights of the neural nets so that they are near each other in value, by punishing the neural networks by for weights that have a great deal of discrepancy.

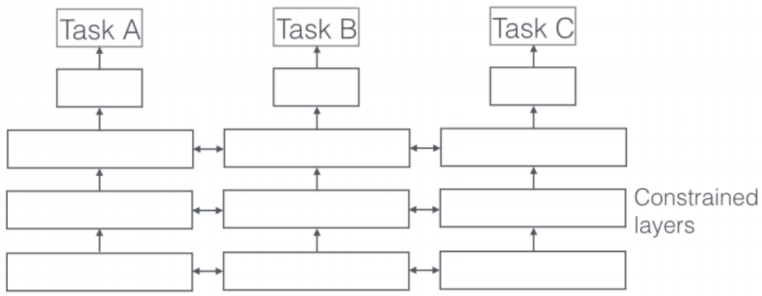


Figure 2: Soft parameter sharing. Each task has a separate neural network, with weights across the layers of all networks being constrained to be close to each other.

3.3.3 Fine tuning

Fine tuning is one of the simplest forms of multitask learning. This technique uses a single neural network which is configured to provide predictions for one environment. The neural net first trains on a single environment. Once training is completed, the weights of the bottom layer are frozen and not allowed to change. This part of the neural network is considered to be "solved" and is now fixed. Then, the neural net trains on the second environment, keeping the bottom layer fixed; only the output layers to the actor and critic heads are allowed to update. Since the neural net being trained on the second environment has been partially solved for a related problem, training on the second environment should take fewer episodes than if it was trained on a randomly initialized neural net.

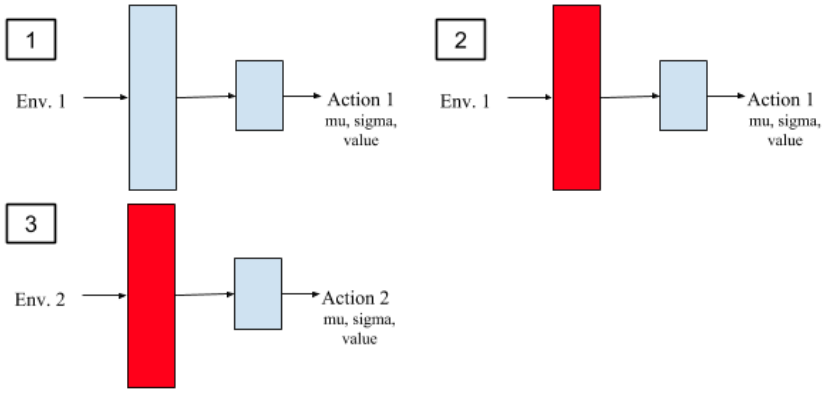


Figure 3: Fine tuning. The neural net first trains on a single environment. Once the environment is trained, the weights of the first layer are frozen. The neural net is then trained on a second, different environment.

3.3.4 Knowledge Distillation

In this method, the goal is to distill (or transfer) knowledge from a pre-trained teacher network T to a student network S. In our project, we pretrained 6 teacher networks $\{T_1, ..., T_6\}$ using the baseline continuous A2C algorithm on the six environments, and then trained one multitask student network S on $n = 6$ datasets $D_i = \{(X_j^i, Y_j^i)\}$ for each environment i. At each optimization step, for each environment, we sample a trajectory τ_i of maximum length $t_{\max}$, which follows teacher policy with probability p_t and student policy with probability p_s. For each state j in the sampled trajectory, (x_j^i, y_j^i) contains the values $([\mu_j^s, \sigma_j^s], [\mu_j^t, \sigma_j^t])$ which parameterize the student policy $\pi_i^s \mid j$ and teacher policy $\pi_i^t \mid j$ respectively. The joint minimization objective for each step is defined as a sum of KL divergences between the teacher and student distributions: $f = \sum_{i=1}^{n} \sum_{j \sim \tau_i} D_{KL} \{N(\mu_j^t, \sigma_j^t), N(\mu_j^s, \sigma_j^s)\}$, where j denotes a state passed by a trajectory. In our case of normal distributions, $D_{KL} \{N(\mu^t, \sigma^t), N(\mu^s, \sigma^s)\} = \log \frac{\sigma^t}{\sigma^s} + \frac{(\sigma^s)^2 + (\mu^s - \mu^t)^2}{2(\sigma^t)^2} - \frac{1}{2}$. KL divergence was chosen as it has performed well on discrete spaces [1].

3.3.5 Distral

Distral, as proposed in Teh et al. [6], is a generalization of distillation learning. In the original paper, there are n neural networks N_i for each of n environments, and one neural network N_0 which acts as a regularizer for the other networks (see Figure 5). The idea behind this approach is that the shared neural network learns the common behaviour, while each of the environment-specific neural networks learns the difference between the common policy and its specific environment. In our implementation, each N_i has three heads: policy head V_i, mean head μ_i and variance head σ_i^2. N_0 has 2 heads: μ_0 and σ_0^2. The algorithm follows the actor-critic framework with the following modifications:

1) During the rollout of the policy for each environment i, given a state s, and action is sampled from a a distribution $\pi_i \mid s = N(\alpha\mu_0 + (1-\alpha)\mu_i, \alpha\sigma_0 + (1-\alpha)\sigma_i)$, where α is a customizable parameter.

2) The loss function is regularized by the sum of KL divergences between $\pi_i \mid s$ and $\pi_0 \mid s$.

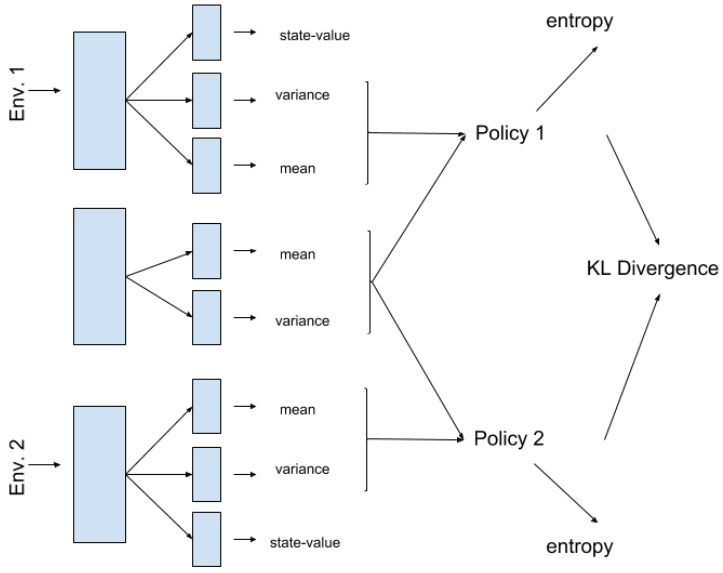


Figure 4: Distral adaptation for the case of 2 environments. The neural network in the middle is the regularizer neural network N_0, with two enviromnent-specific networks N_1, N_2 on the sides.

4 Methods

4.1 Setup

Tests were done on OpenAI gym environments. We selected the Inverted Pendulum environment from Mujoco because the low dimensionality of the problem makes it more manageable. Multiple environments with different pole lengths and different friction levels were used. The goal for the agent in each environment was to learn to balance the pole on the moving cart. (Figure 5) During the simulation, the algorithm controls the movement of the pendulum by applying horizontal force to the cart and nudging it in either right or left direction. In these simulations, the force can be any value on the uncountable $[-3, 3]$ interval, hence these are continuous control environments. In our adaptations, the algorithms take the position of the inverted pendulum as input, and are trained to output a mean μ and variance σ^2. The mean and variance parametrize the Gaussian probability distribution $N(\mu, \sigma^2)$, and at the next step, the force applied to the cart is then sampled from this distribution. If the pendulum has not fallen during this step, the reward for this episode increases by 1. The episode ends when the pendulum falls, and the neural networks regress on the total reward from the episode to improve their estimates for μ, σ^2. Figure 5 shows a trained Standard Inverted Pendulum simulation.

To simplify our calculations and experiments, we also included a finite time horizon, so the episode terminated with the highest reward after 999 steps even if the pendulum was upright. This assumption is justified, since from our observations, the algorithm that is able to keep the pendulum upright consistently for over 900 steps can be considered "trained". To ensure that the resulting networks would be stable and consistent with their task performances, we introduced a notion of running reward, defined as 1% of the number of steps the most recent run lasted (capped at 999 steps) plus 99% of the previous running reward. Each neural network was trained until it reached a running reward of 950 steps out of possible 999.

Since the eventual goal of this line of research is an ability to transfer knowledge from a simpler to a more difficult task, six different variations of the inverted pendulum problem were selected for training. Each of these environments are similar enough such that we expected that the neural networks will be able to learn from each other. However, despite the similarities between these simulations, some were much "harder" for the baseline A2C algorithm than others. Figure 6 lists all of the environments used for simulations.

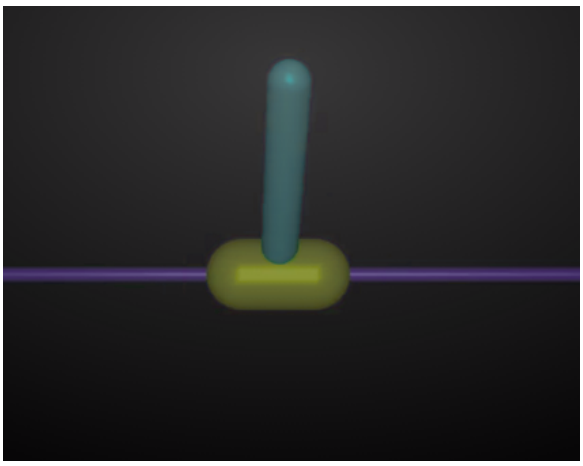


Figure 5: Trained Inverted Pendulum environment

1. Standard version
2. High friction (100% higher friction)
3. Low friction (50% lower friction)
4. High gravity (1.5 g - Super Earth gravity)
5. Low gravity (0.38 g - Martian gravity)
6. Long pole (50% longer)

Figure 6: Variations of Inverted Pendulum

5 Results

The purpose of our research was to determine if we could use multitask learning to resolve two issues with continuous action space reinforcement learning algorithms: instability and sampling inefficiency. As evidenced by the graphs in Figure 8, all techniques with the exception of soft-parameter sharing converge faster than the baseline A2C algorithm.

We tested each of the 6 algorithms (including a baseline single-task learning algorithm) on 6 environments. The algorithms were trained until they reached a running reward of 950 steps. The running reward is defined as 1% of the number of steps the most recent run lasted (capped at 999 steps) plus 99% of the previous running reward. This benchmark emphasizes stability and an algorithm's ability to consistently balance the pole.

Fine-tuning (Figure 7f, 9b) was our simplest model, and it performed the best on running-reward for all environments except the full pendulum. This is because fine-tuning was first trained on the regular pendulum environment. Thus, we expect the performance of fine-tuning on that environment to be similar to the performance of the baseline, which is confirmed in Figure 7f and Figure 10. The advantages of this approach are its simplicity and remarkably superior convergence times on all "student" environments. However, fine-tuning requires careful assumptions about the similarity of different environments. Moreover, the prerequisite to fine-tuning student models is the ability to train an agent in at least one environment without any multi-tasking. Both of these hurdles can pose significant challenges for real life applications. In our low-dimensional experiments, fixing one layer of the neural network proved to be successful. However, in other settings it is entirely possible that the environments do not relate to each other in such straightforward way.

Knowledge distillation (Figure 7c, 9c) is similar to fine-tuning in the sense that both require some expert policies to train. The difference is that knowledge distillation combines as many experts as there are environments into a single generalized model while fine-tuning continually refines a single model to generalize to the new environments. Since the objective for knowledge distillation is much more robust than for all other algorithms, its convergence time is markedly faster, with the exception of fine-tuning. In addition, out of all algorithms considered, knowledge distillation

is best at reducing the variance. However, while the resulting knowledge distillation model outperforms the baseline, the time to train each teacher/expert model may outweigh the benefits of faster convergence in the aggregate model. Yet these experiments show that there might be some benefit in distilling the knowledge from a few expert models trained on similar environments, as this algorithm's training process was more stable than those for all other algorithms.

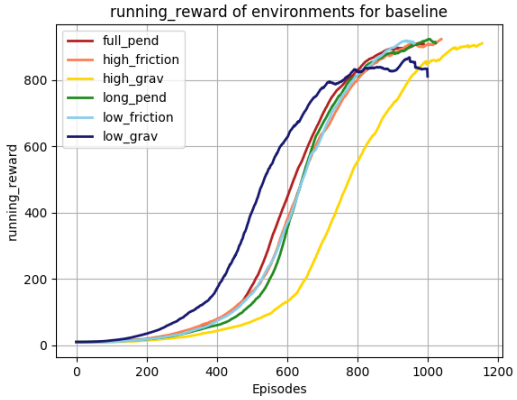
Distral's performance lags that of distillation learning or fine-tuning, but is still notably better than the baseline. (Figures 7d, 9f) Distral's advantage is that there is no need to have pretrained expert models. The distilled policy trains at the same time as the task-specific neural networks. In addition, unlike hard- and soft- parameter sharing, Distral does not assume that the neural networks have dependencies only across the same levels. Instead, Distral works only with the outputs of the neural networks. It is arguably the most generalizable algorithm that we have attempted, as it hard-codes the least about the nature of similarity between different environments.

Hard and soft parameter sharing (Figures 7b, 7e, 9c, 9d) appear to have the least improvement relative to the baseline. As discussed above, a possible explanation for this underwhelming result is that soft- and hard- parameter sharing assumes that the individual weights are similar across the different neural networks. Thus, while Distral (Figure 13) puts a penalty on dissimilar output, soft- and hard- parameter sharing attempt to bind the weights in layers as well. Such a setup is likely to outperform Distral only if there are some unknown, but specific similarities across different environments. As evidenced by our experiments, our learned models were not so homogeneous across layers. As real-world scenarios are more complicated, it is unlikely that the many of them would exhibit such layered structure.

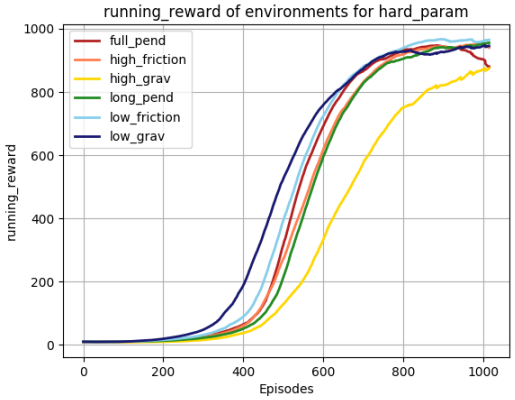
Moreover, while both hard- and soft- parameter sharing suffer from the above disadvantages, the latter algorithm is also more complex than compared to the former, and thus takes longer to train, as evident in Figure 9 and convergence tables. Unlike hard-parameter sharing that trains a single neural network, soft-parameter sharing consists of n separate neural networks. The soft parameter regularization evidently did not encourage knowledge transfer between environments. As a result, for most environments, the performance of the soft-parameter sharing is comparable to or even worse than the baseline A2C algorithm. The failure of soft-parameter sharing is particularly evident when the two methods are compared on a difficult environment such as High Gravity. In Figure 9d, the soft-parameter sharing model is the only model that did not finish training after 1200 episodes. Moreover, it is the only model to exhibit such variance in the performance.

Hard parameter sharing outperformed the baseline measurements on all of the environments, but its convergence times are not dramatically faster, as can be seen in Figure 9. Moreover, we were surprised to find that hard-parameter algorithm did not reduce the instability of the learning process, as compared to the baseline. Additionally, out of the algorithms that we implemented, hard-parameter sharing was the most sensitive to changes in model's hyper-parameters such as the learning rate or the type of optimizer (Adam vs SGD).

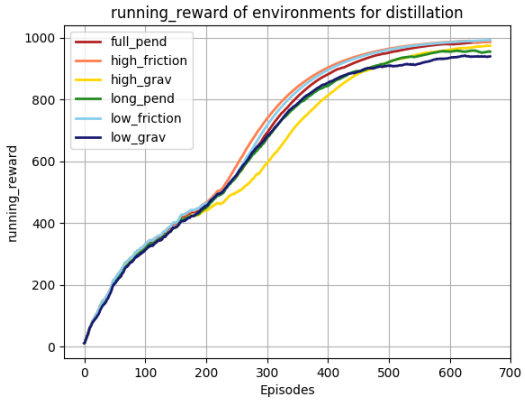
5.1 Performance during training



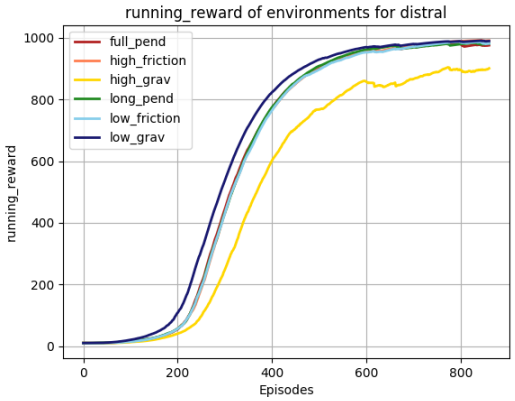
(a) Performance of the baseline model.



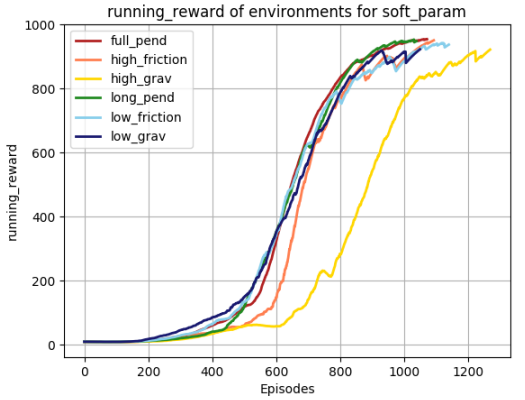
(b) Performance of the Hard-Parameter model.



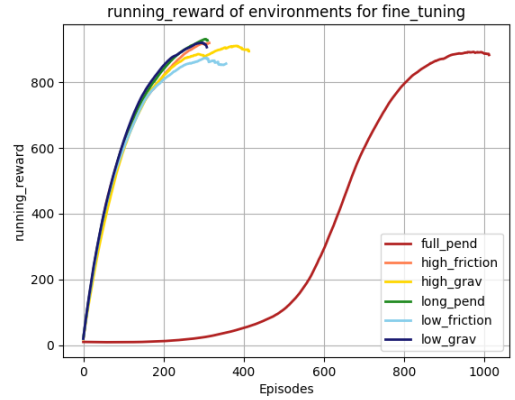
(c) Performance of the Distillation model.



(d) Performance of the Distral model.

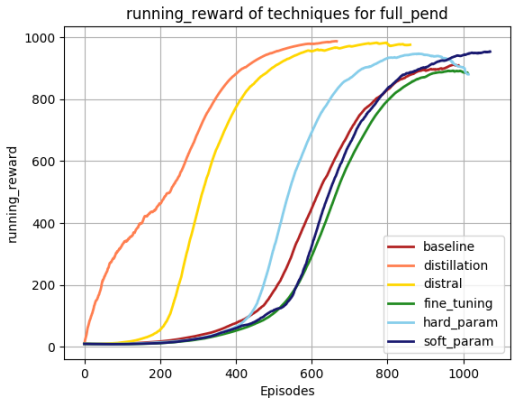


(e) Performance of the Soft-Parameter model.

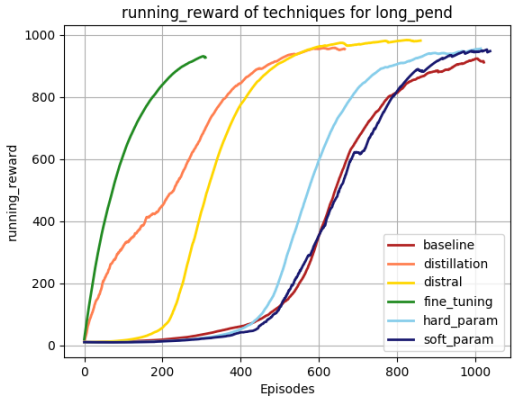


(f) Performance of the Fine-tuning model.

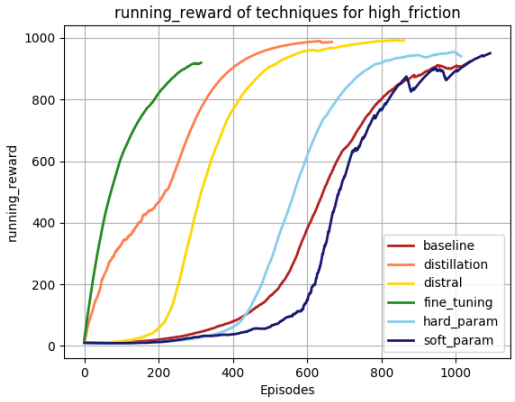
Figure 7: The comparison of the relative difficulty of each environment for each of the 5 Multi-tasking methods discussed above. The x-axis is the number of episodes the agent has spent training. The y-axis is the running reward, which measures the average performance of the neural network across all episodes. The higher the running reward as a function of the number of episodes, the faster the algorithm trains to succeed in the task.



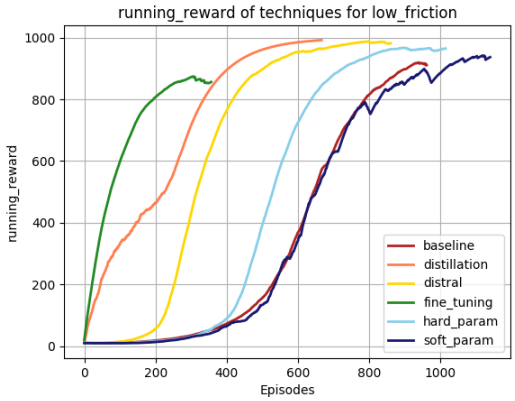
(a) Performance of the Multi-task agents on Full Pendulum environment.



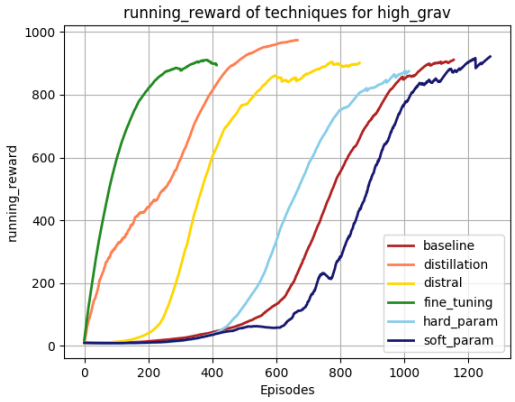
(b) Performance of the Multi-task agents on Long Pendulum environment.



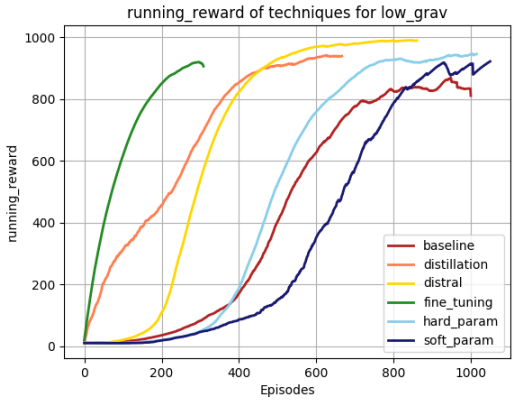
(c) Performance of the Multi-task agents on High Friction environment.



(d) Performance of the Multi-task agents on Low Friction environment.

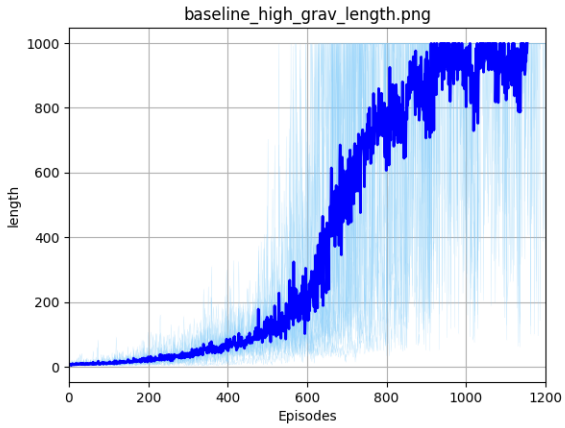


(e) Performance of the Multi-task agents on High Gravity environment.

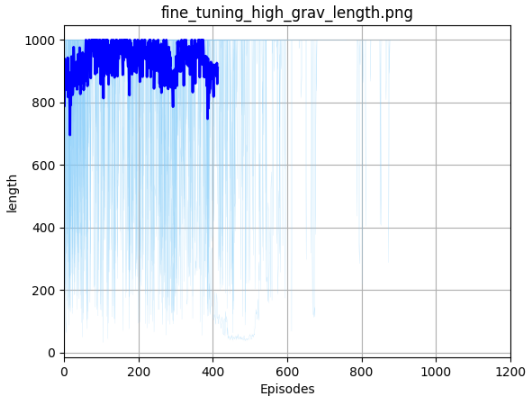


(f) Performance of the Multi-task agents on Low Gravity environment.

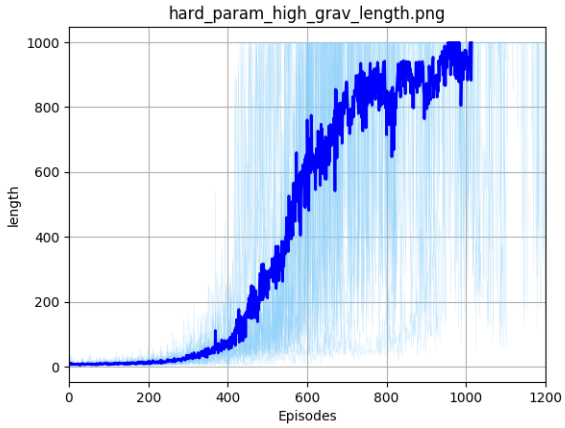
Figure 8: The comparison of the performance of the 5 Multi-tasking methods for each of the Inverted Pendulum environments. The x-axis is the number of episodes the agent has spent training. The y-axis is the running reward, which measures the average performance of the neural network across all episodes. The higher the running reward as a function of the number of episodes, the faster the algorithm trains to succeed in the task.



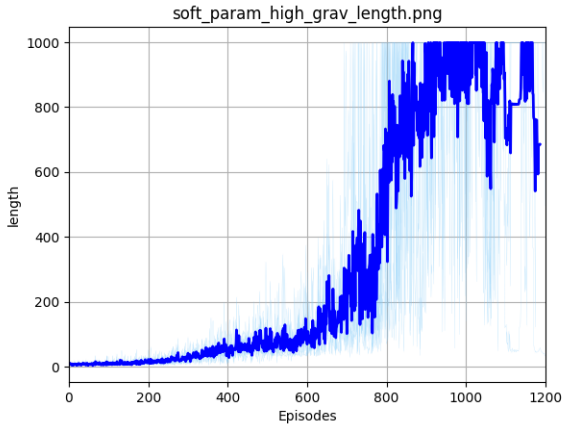
(a) Performance of the baseline model.



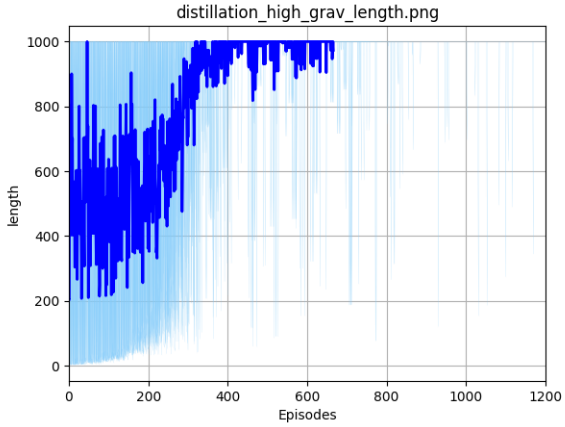
(b) Performance of the Fine-Tuning model.



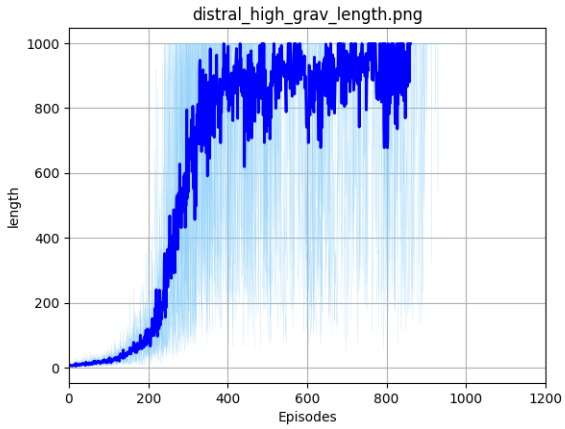
(c) Performance of the Hard-parameter model.



(d) Performance of the Soft-parameter model.



(e) Performance of the Distillation model.



(f) Performance of the Distral model.

Figure 9: Running length as a function of the number of roll-outs (episodes) for the high gravity environment. The light blue lines indicate the spread in the length of the episode, and the dark blue lines show the average length per episode.

5.2 Comparison of algorithms

The tables below give the convergence time in episodes for each technique and for each environment. The mean convergence time, the standard deviation, and the number of trials for each is given. Convergence time is defined as the number of episodes needed for the running reward to reach 950. This means the algorithm has been trained to consistently keep the pendulum upright for at least 950 steps.

Figure 10: Convergence Performance for full pendulum

	Mean	Std. Dev	Trials
hard param	886.7	210.9	15.0
soft param	1050.6	188.1	22.0
fine tuning	1165.1	319.8	85.0
distral	572.5	42.2	10.0
distillation	494.1	8.6	10.0
baseline	1149.1	327.1	28.0

Figure 11: Convergence Performance for high friction

	Mean	Std. Dev	Trials
hard param	971.3	466.4	15.0
soft param	1080.5	11.5	2.0
fine tuning	366.7	75.0	18.0
distral	578.3	40.8	10.0
distillation	466.8	6.5	10.0
baseline	1066.1	101.6	11.0

Figure 12: Convergence Performance for high gravity

	Mean	Std. Dev	Trials
hard param	1043.7	177.3	15.0
soft param	1525.8	616.6	5.0
fine tuning	515.3	156.4	15.0
distral	869.1	133.0	10.0
distillation	558.4	10.1	10.0
baseline	1399.0	590.6	11.0

Figure 13: Convergence Performance for long pendulum

	Mean	Std. Dev	Trials
hard param	896.8	88.2	15.0
soft param	1032.2	147.4	5.0
fine tuning	351.2	43.7	17.0
distral	568.7	40.4	10.0
distillation	564.5	20.2	10.0
baseline	1082.5	99.9	11.0

Figure 14: Convergence Performance for low friction

	Mean	Std. Dev	Trials
hard param	810.9	57.3	15.0
soft param	1161.4	109.2	5.0
fine tuning	615.7	751.8	17.0
distral	575.6	41.1	10.0
distillation	474.5	6.6	10.0
baseline	1034.0	98.0	11.0

Figure 15: Convergence Performance for low gravity

	Mean	Std. Dev	Trials
hard param	847.0	223.9	15.0
soft param	1146.8	291.1	5.0
fine tuning	423.1	205.7	18.0
distral	536.3	37.2	10.0
distillation	768.4	171.7	10.0
baseline	1247.9	486.5	8.0

6 Discussion

As self-driving cars and autonomous robots become a reality, the development of sample efficient, robust continuous RL models is increasingly important. The aim of this project was to show that multitask learning can help achieve these goals. We picked five state-of-the-art techniques - fine-tuning, hard- and soft-parameter sharing, distillation learning and distral - and incorporated them into the continuous actor-critic algorithm that learned to balance a pole on a cart. Our experiments confirmed that combining the objectives of multiple environments decreases training time. Moreover, some models (Distral, fine tuning, and knowledge distillation) were more robust to hyperparameter change than the baseline A2C algorithm. Surprisingly, the multitask learning did not significantly increase the stability during training. Overall, our project demonstrates a potential for research in understanding which factors and architectures result in the largest gain for multi-task learning.

We found that training 'harder' environments in combination with 'easier' environments considerably reduces the training time for the more difficult cases, while only slightly increasing the training times for the simpler environments. More sophisticated RL techniques (e.g. the ones that learn a policy to determine which environments to sample from) are likely to further reduce the required number of samples by choosing to train on easier cases.

During the roll-outs of different multi-tasking versions of A2C, several factors such as the learning rate and the optimizer were fixed across all trials. The optimal learning rate may vary based on the technique, which could significantly improve learning time. A fairly modest learning rate of $1e^{-3}$ was used to improve the stability of training and prevent the running reward from suddenly plummeting. However, some algorithms may be robust enough to train at a higher learning rate.

The experiments were done on a low dimensional problem with several fairly similar environments. As seen in the baseline tests, only the high gravity and low gravity environments were notably more and less difficult than the rest. The four other environments converged at roughly the same rate for baseline tests. An extension of our work would be to attempt training on a wider variety of environments with a greater range of difficulty. There may be a threshold of similarity where multitask learning is no longer effective.

A more complex challenge would be to attempt multitask learning on a higher dimensional problem. Originally, we planned to experiment on a human standing environment. However, this task was rejected due to its complexity and the time constraints of the project. An extension of our work would be to study multitask learning techniques in more sophisticated continuous environments. Previous works have shown that frequently, RL algorithms that perform well for low-dimensional tasks do not have adequate stability to accommodate more complex environments. Thus, while the conclusions of our project are promising, a lot more work is required before continuous action space RL can be successfully applied in practice.

7 Our code

All of our experiments can be found at https://github.com/timsliu/CS159Project.

8 References

[1] Benchmark Environments for Multitask Learning in Continuous Domains. Henderson, P., Wei-Di, C., Shkurti, F., Hansen, J., Meger, D., Dudek, G. Lifelong Learning: A Reinforcement Learning Approach Workshop 2017.
[2] Multi-task Learning for Continuous Control. Arora, H., Kumar, R., Krone, J., Li, C. arXiv preprint arXiv:1802.01034v1 2017.
[3] An Overview of Multi-Task Learning in Deep Neural Networks. Sebastian Ruder. arXiv preprint arXiv:1706.05098v1 2017.
[4] Multi-Task Deep Reinforcement Learning for Continuous Action Control. Zhaoyang Yang, Kathryn Merrick1, Hussein Abbass1, Lianwen Jin. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence. 2017.
[5] Deep multi-task learning with low level tasks supervised at lower layers. Anders Sogaard, Yoav Goldberg. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics. 2016
[6] Distral: Robust Multitask Reinforcement Learning. Yee Whye Teh, Victor Bapst, Wojciech Marian Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, Razvan Pascanu DeepMind, London, UK 2017
[7] Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods. Alessandro Lazaric, Marcello Restelli, Andrea Bonarini. 2016
[8] Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, Jimmy Ba, 2017