**Introduction**

Generalization remains one of the most fundamental challenges in deep reinforcement learning. To benchmark both sample efficiency and provide a direct measure of how quickly a RL agent can learn generalizable skill in RL, OpenAI introduced Procgen ― a suite of 16 procedurally generated game-like environments. We attempted to train an agent in the Procgen FruitBot Environment, where the RL Agent had to learn how to pick up fruit items while avoiding non-fruit items and navigating itself through partitions. The RL agent was rewarded points when it picked up fruit items and was deducted points when it picked up non-fruit items. The RL agent was not awarded any points when it successfully navigated through barriers, however, failure to do so resulted in the termination of the game. We first attempted to approach the problem via Hierarchical RL before diverting to more classical RL techniques. We experimented with multiple reinforcement learning architectures, built upon its work and evaluated our agent’s performance on unseen testing environments.

**The 3-component Model**

We set out to create an algorithm with separate components that would each learn to overcome a different hurdle, which includes a perception network to learn meaningful representation of the game, a planning network to set short term goals, and an executioner network to control actions according to the state dynamics to reach these goals. The separate components would learn their respective roles by each being trained on different losses. The perception network would be pretrained without rewards to simply learn a semantic representation of the environment, for example through an autoencoding task. The executioner network would learn to achieve its designated short term goals rather than maximizing the environment’s reward. The planner would learn how to communicate plans by learning from past attempts, relabeling the goal states with the achieved states, as well as learn to maximize reward. None of our models were able to achieve satisfactory results. We reviewed previous literature (Ofir et al., 2019; Danijar et al. 2019) that resembled our ideas. We planned on testing various learning schedules to train different portions of the model by changing tasks and learning rates. We experimented with meta learning libraries and hierarchical RL libraries, but each had conflicting deprecated dependencies. Despite days of debugging and countless attempts on different virtual environments, we were unable to get the libraries to work. We understood that in order to get any meaningful results, we needed to at least create a model that could solve the environment.

**Baselines Experiments**

As a first step, we ran experiments using common RL algorithms, training 1 million timesteps of different configurations of ACER, DQN, and PPO2. Our save files from this part of experimentation run on remote machines were deleted when our sessions ended, but we did copy down some of the best achieved results by hand. These algorithms failed to solve the environment, even when measuring training performance on a 1 level environment.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | PPO 1 lvl | PPO 50 lvl | ACER 1 lvl | ACER 50 lvl | DQN 1 lvl | DQN 50 lvl |
| Final Train Reward Mean | -1.14 | -2.03 | -1.28 | -2.21 | -1.60 | -2.51 |

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | PPO\_CNN 50 lvl | ACER\_CNN 50 lvl | PPO\_IMPALA 50 lvl |
| Final Train  Reward Mean | 0.00 | 1.09 | 0.37 |

**Rethinking the problem**

When these models weren’t able to solve the environment, we reexamined the task. We realized that generalization is not so difficult. A policy that has learned well should be able to perform on new levels, since they are practically within distribution. From analyzing the random generation code in ProcGen, we found that the rest of a level was almost indistinguishable from a new level with extremely minor distribution changes such as the number of walls spread throughout a level. That being said, our models had difficulty solving the levels we had trained on, so generalization wasn’t the issue. Exploration is certainly an issue, but this can be brute forced with the number of timesteps we were training for. ProcGen environments are rendered at 64x64 pixels, if our perception is slow to learn and dissect the large amount of data meaningfully, this could compound with random exploration to take many orders of magnitude more to learn. We determined that the crux of the problem did not lie in good planning, precise control, or generalization. With a good perception network, a policy as simple as moving towards positive reward objects and away from negative reward objects (or walls) would be sufficient for performing far better than anything we had managed to create. It is important that perception is able to isolate the correct information relevant to rewards. The space between 2 walls at the top of the screen is an important learned feature for recreating the image, but it is not relevant to the rewards acquired by the bot at the bottom of the screen. Now, the focus of our training was to learn good perception in conjunction with a simple policy.

**Background**

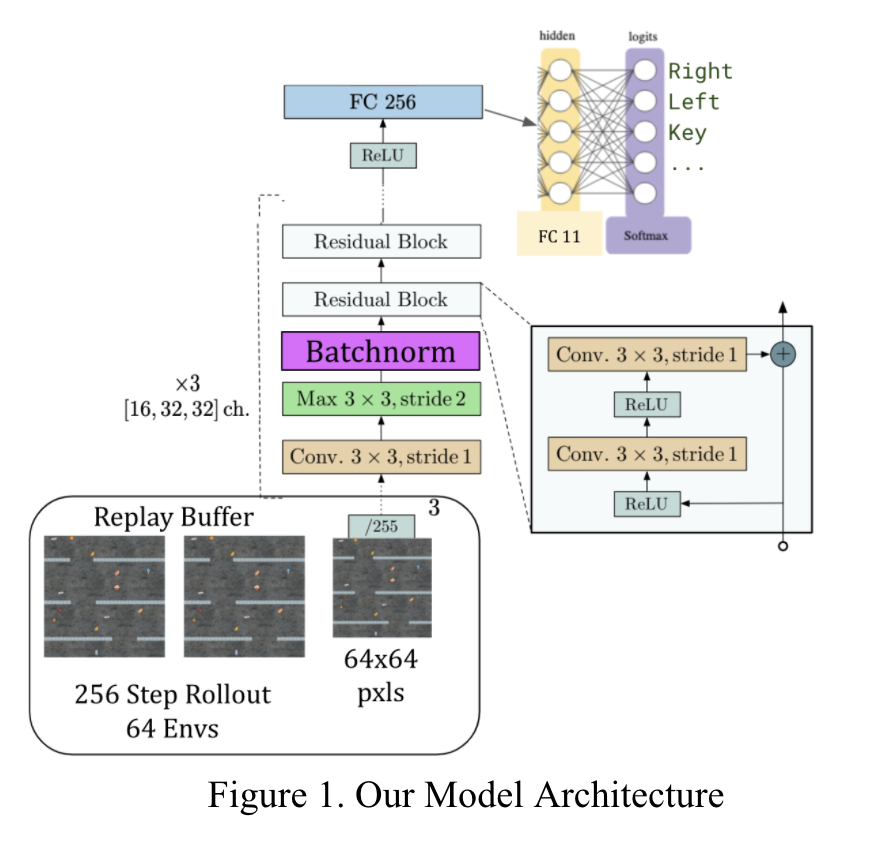
Trust Region Policy Gradient is a variation of Policy Gradient which constrains (or penalizes) the gradient update to remain close to the original policy using the KL divergence as a measure. Proximal Policy Optimization achieves a similar goal by maximizing a clipped objective that approximates TRPO’s KL penalty. Furthermore, it adds an entropy objective to incentivize exploration. PPO2 is a GPU optimized implementation of the algorithm suited for vectorized environments.

A common feature of most RL algorithms is an Experience Replay Buffer. This is a buffer of observations, actions, and rewards continuously collected throughout rollouts during training. By sampling the replay buffer, the dynamic distribution of RL training data is smoothed out leading to less correlated data samples, a critical property for neural network training. Importantly, replay buffer algorithms have achieved the best sample efficiency on the standard Atari environments [ACER].

Batch normalization was created to reduce internal covariate shift. It normalizes a layer’s activations by finding the distribution of those activations across a batch. This allows for higher learning rates, and ensures the activations aren’t stuck in saturated ranges. Batch normalization also offers some regularization effect, reducing generalization error.

**Approach and Experiments**

In order to learn better perception of our task, we decided to use a policy gradient algorithm with a policy network consisting of a convolutional network followed by a fully connected layer. This would be able to model our idea of a simplistic policy that maximizes the probability of actions such as right or left when they lead to rewards and minimizes their probability when they don’t. Most of the parameters would lie in the convolutional model that would learn meaningful small embeddings of the environment that would be relevant to the policy. In addition, we took in account various output distributions. Ultimately, our actions are discrete, but they have semantic meaning which can be used to translate continuous directions into a mix of possible key presses. We chose to use a categorical probability distribution, which misses out on the meaning of actions (Left/Right/Up/Down) , especially its correlation to the structure of our input data: 2D images, but does have the modelling capacity to output any action distribution. Swerving around negative objects is absolutely critical in this task because those are the source of so much negative reward. While the model learns, it is still uncertain both in perception and the policy’s decision making, in which case having a bimodal distribution that can choose one side to veer to is extremely important so it does not simply pick the average and head right into negative reward.

While this setup achieved nonnegative mean rewards during training, we got increased performance by utilizing modern convolutional architecture: a ResNet. In order to address the lack of a replay buffer in policy gradient, we used a rollout to gather more samples each iteration. At this point (unfortunately late), we discovered the procgen-train repository, which helped us find tuned hyperparameters (Appendix B) for these components.

In order to better increase sample efficiency as well as improve our testing performance, we added batch normalization layers into our ResNet. However, even with the replay buffer, the samples in our batch were far from independent. The biggest increase in performance came from using vectorized environments. We found very little success (at most 2.7 mean reward) training on 1 environment, throughout experiments with ACER, DQN, PPO, etc. By using 64 parallel simulations, we are able to take advantage

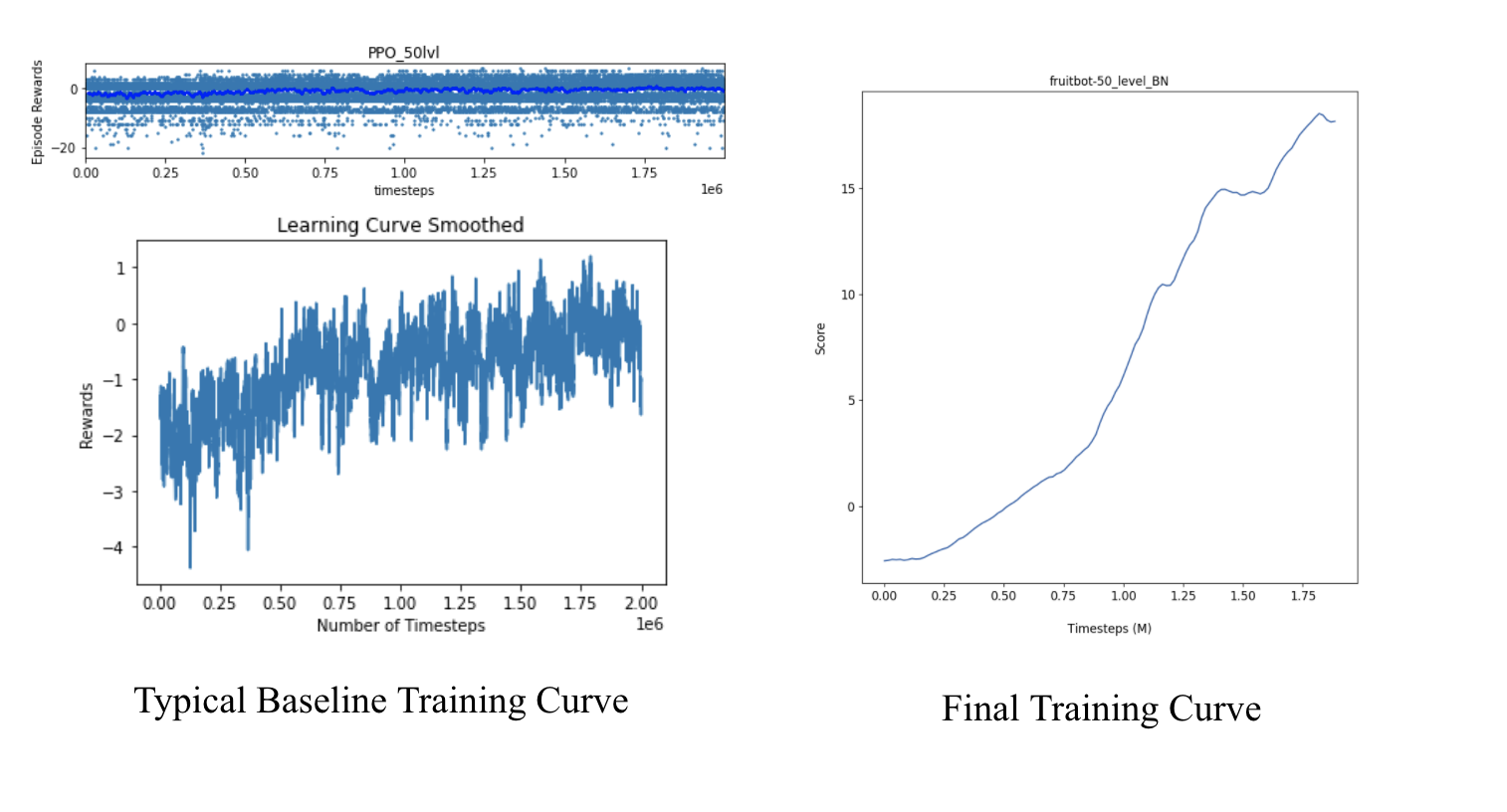
of vectorized numerical operations for faster training, as well

as collect a more inter-independent set of samples for our experience replay buffer.

However, there are still glaring biases. For the first half million timesteps, our models have yet to survive for over 100 timesteps per episode. This means our states are still almost entirely around initial states prior to passing the first wall. The reason this is dangerous is because when we begin seeing samples further into the level, our shifted distribution will shift our latent activations through the batchnorm layer, even for the unchanged signals of initial states that we had learned to perform well on. By having many parallel simulations, this smooths out the phenomenon, and a high enough learning rate allows us to adjust the model as the input data distribution shifts.

Our final model architecture is displayed in Figure 1. For baselines, we used ACER, DQN, and PPO. The baseline models with CNNs utilized the architecture in [[Nature](https://www.nature.com/articles/nature14236)].

**Results**

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Mean Test Reward**  **after 1e6 steps** | Baseline PPO CNN | 64 Envs  Model | 64 Envs + Batch Norm |
| All lvls | -0.32 | 9.53 | 11.04 |
| 50 lvls | -0.49 | 8.67 | 8.14 |
| 100 lvls | -0.61 | 9.11 | 8.98 |
| 250 lvls | 0.12 | 10.02 | 12.66 |

Figure 2. Baseline and final training curves

(See Appendix A for additional results)

The data imply that training with more levels yields higher mean reward, likely due to better generalization. Batchnorm appears to have a marginal improvement, but this may also just be explained by the noisiness of training. Most of our logs were deleted from our remote machines, but the results from those experiments guided our approach, as discussed above.

**Conclusion**

After attempts at building a hierarchical RL model and running various baseline experiments, we concluded that the crux of the problem lies in learning meaningful representations of the game. Our final architecture, a policy gradient algorithm with a policy network consisting of a ResNet with batchnorm followed by a fully connected layer, was able to yield significantly better results than baseline models. Through the course of this project, we spent a frustrating amount of time debugging codebases from existing literature. The short life span of ML library, although semi-anticipated, was nonetheless surprising. Our biggest breakthrough was finding a set of hyperparameters used by OpenAI to train their ProcGen models, without which we would not have been able to yield meaningful results. This speaks to the fragility of RL models – evidence against their generality. With the amassed number of tunable hyperparameters, perhaps it is disingenuous to study only the (non-hyper)parameter search, namely “training”, when analyzing learning.

**Team Contributions**

Work related to the initial proposal and project milestones were split equally between the team. For the final paper, both team members contributed equally to experiments related to the 3-component model and the baseline model. Osher led the design and training of the final model architecture. Both members contribute to the writing of the final writeup.

**References**

John Schulman and Filip Wolski and Prafulla Dhariwal and Alec Radford and Oleg Klimov. Proximal Policy Optimization Algorithms. arXiv:1707.06347, 2017.

Ziyu Wang and Victor Bapst and Nicolas Heess and Volodymyr Mnih and Remi Munos and Koray Kavukcuoglu and Nando de Freitas. Sample Efficient Actor-Critic with Experience Replay. arXiv:1611.01224, 2016.

Danijar Hafner and Timothy Lillicrap and Ian Fischer and Ruben Villegas and David Ha and Honglak Lee and James Davidson. Learning Latent Dynamics for Planning from Pixels.arXiv:1811.04551, 2018.

Yoshua Bengio and Aaron Courville and Pascal Vincent. Representation Learning: A Review and New Perspectives. arXiv:1206.5538, 2012.

Karl Cobbe and Christopher Hesse and Jacob Hilton and John Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning. arXiv:1912.01588, 2019.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167, 2015

Aboudy Kreidieh, h-baselines, (2021), GitHub repository, <https://github.com/AboudyKreidieh/h-baselines>

OpenAI, baselines, (2018), GitHub repositories, <https://github.com/openai/baselines>

Hill-a, stable baselines, (2021), GitHub repository, <https://github.com/hill-a/stable-baselines>

Lasse Espeholt and Hubert Soyer and Remi Munos and Karen Simonyan and Volodymir Mnih and Tom Ward and Yotam Doron and Vlad Firoiu and Tim Harley and Iain Dunning and Shane Legg and Koray Kavukcuoglu, IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures, arXiv:1802.01561, 2018

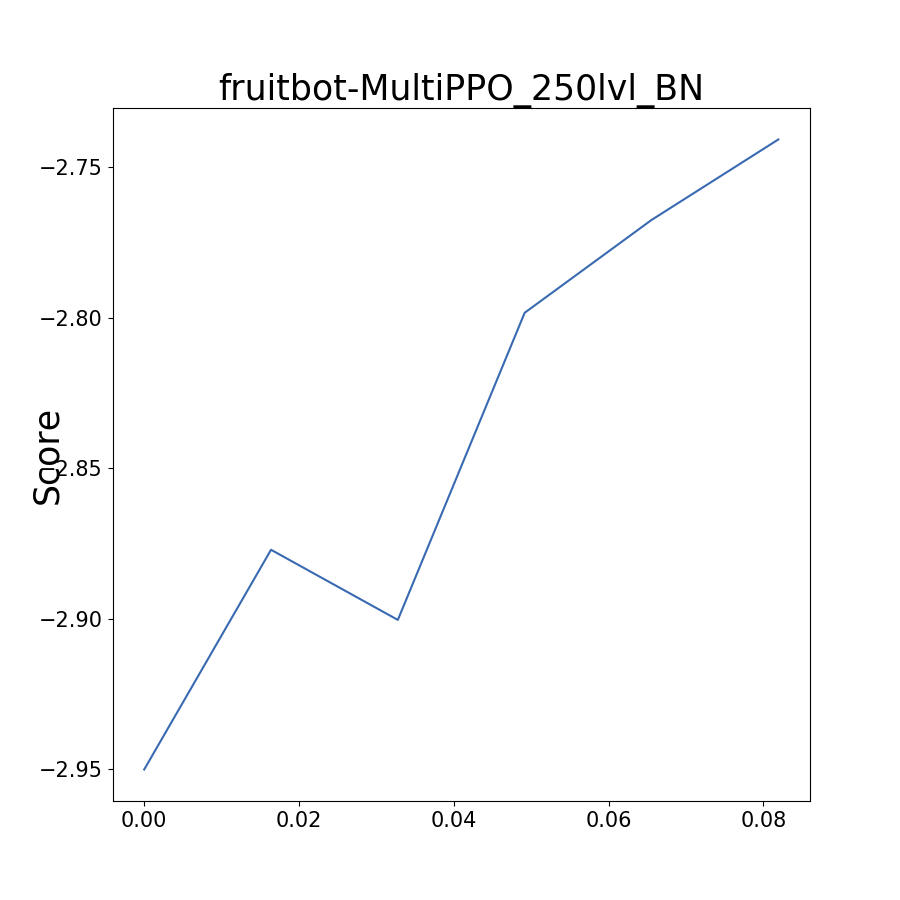
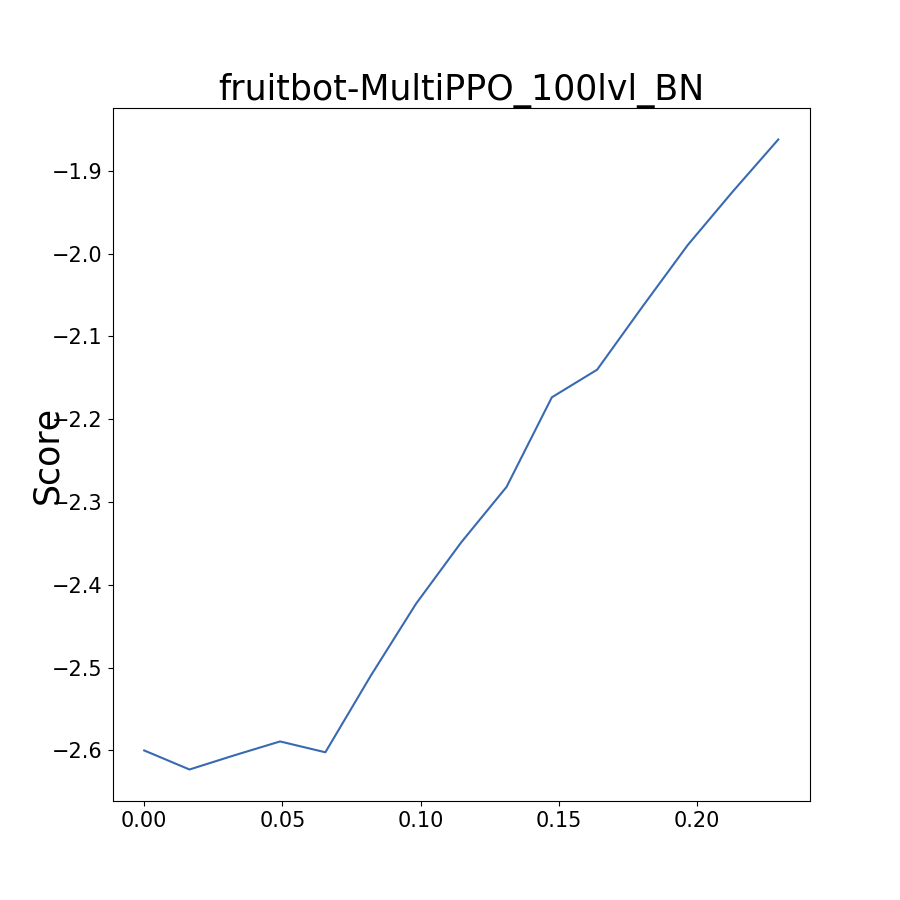
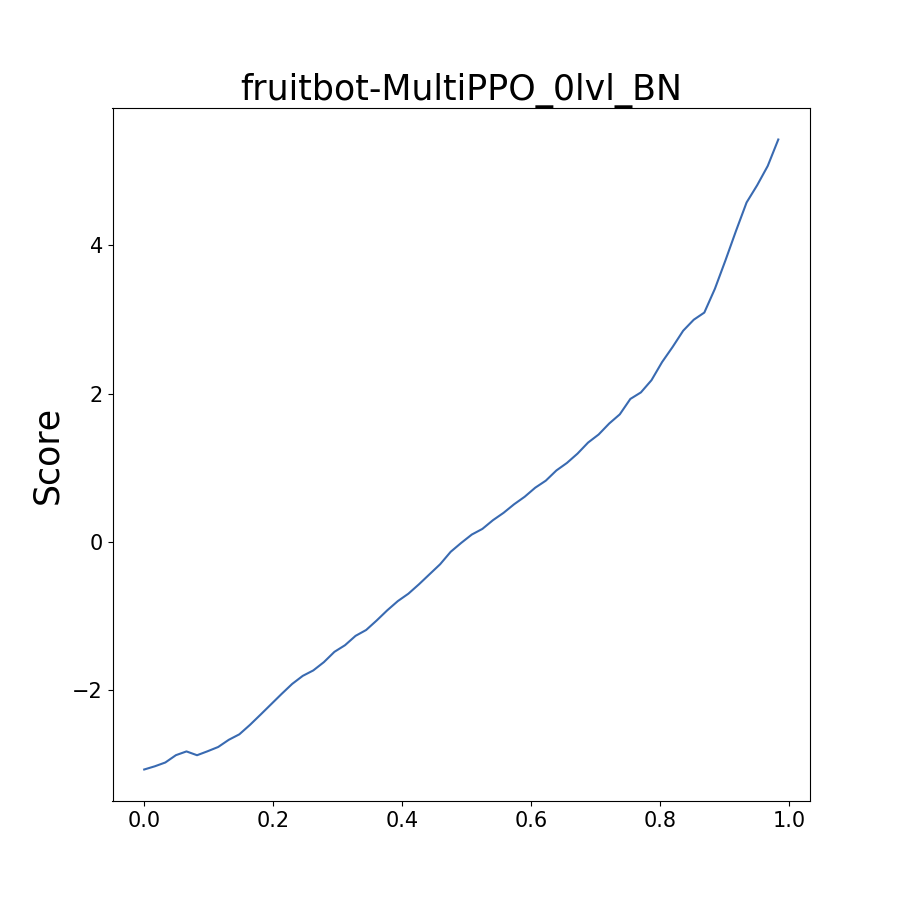
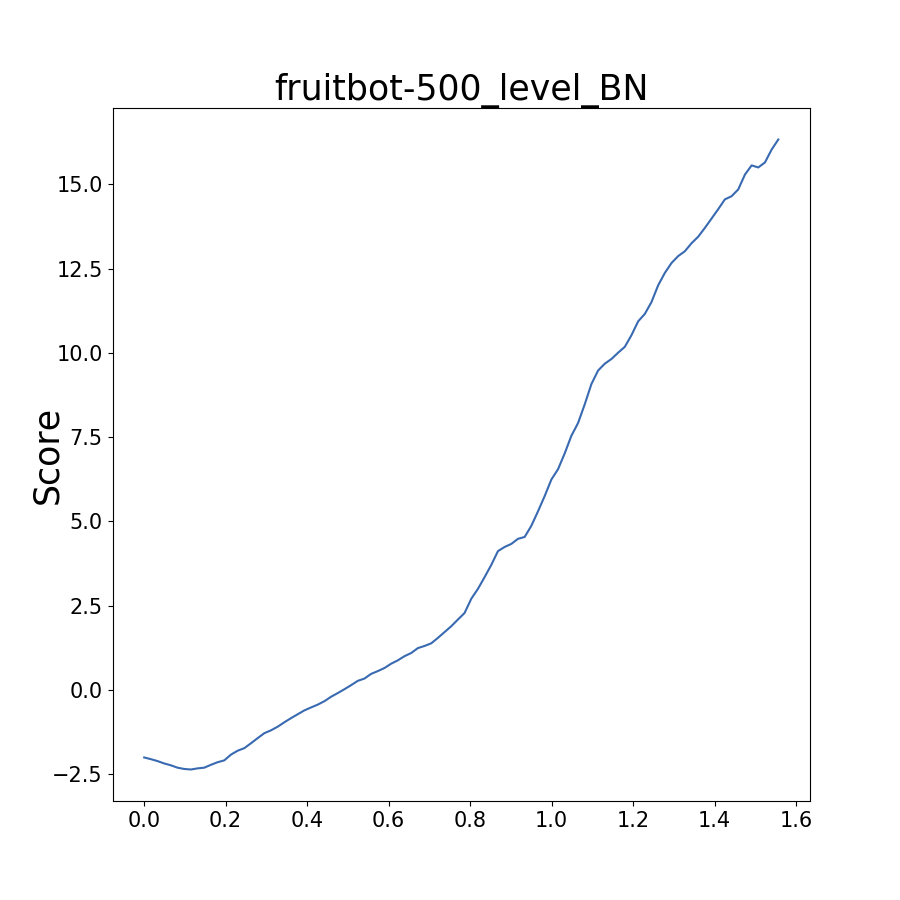
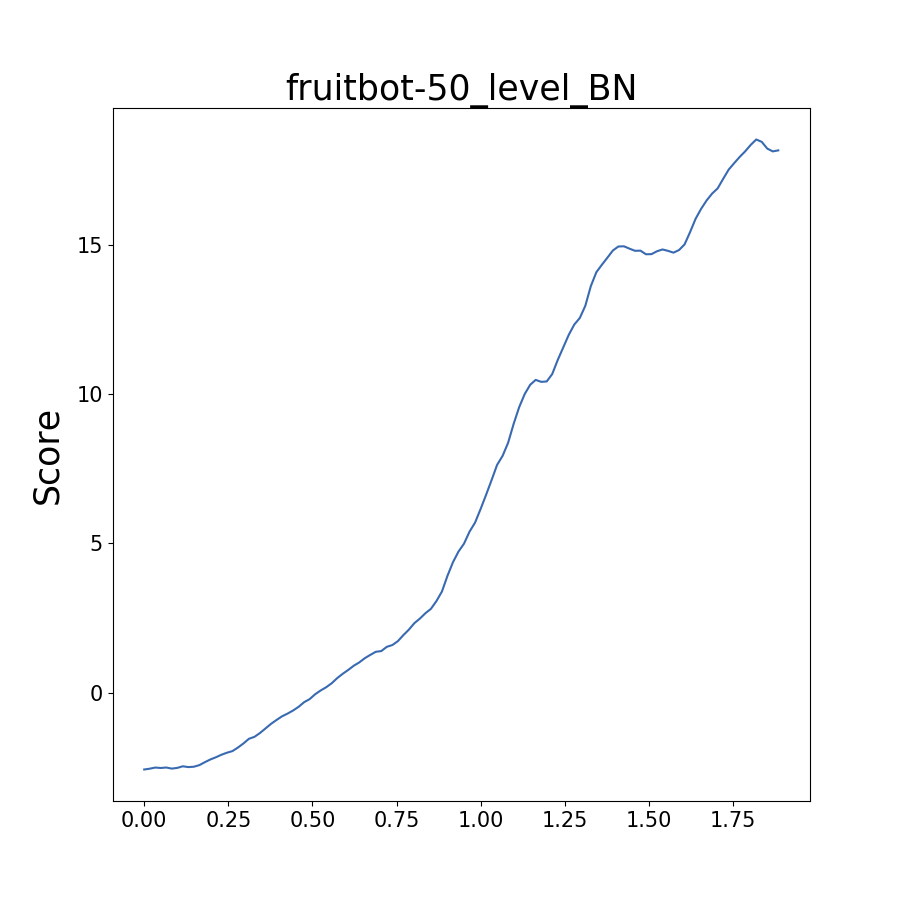
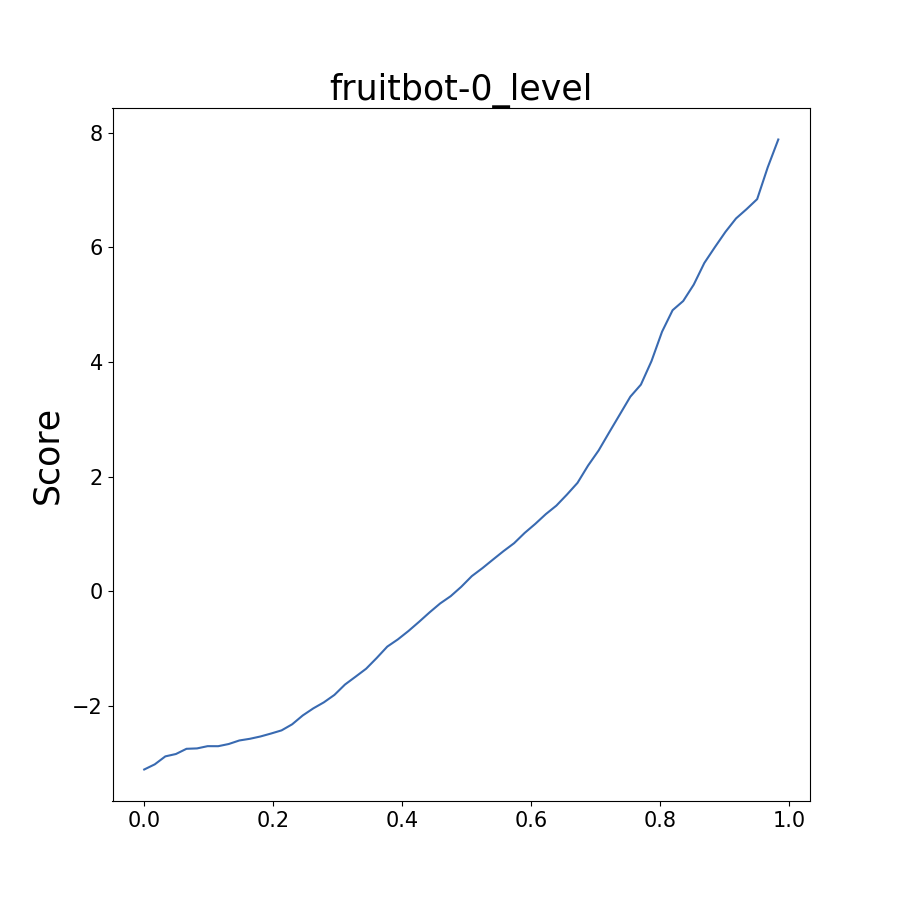
Volodymyr Mnih, Koray Kavukcuoglu. Human-level control through deep reinforcement learning. Nature, 2015.

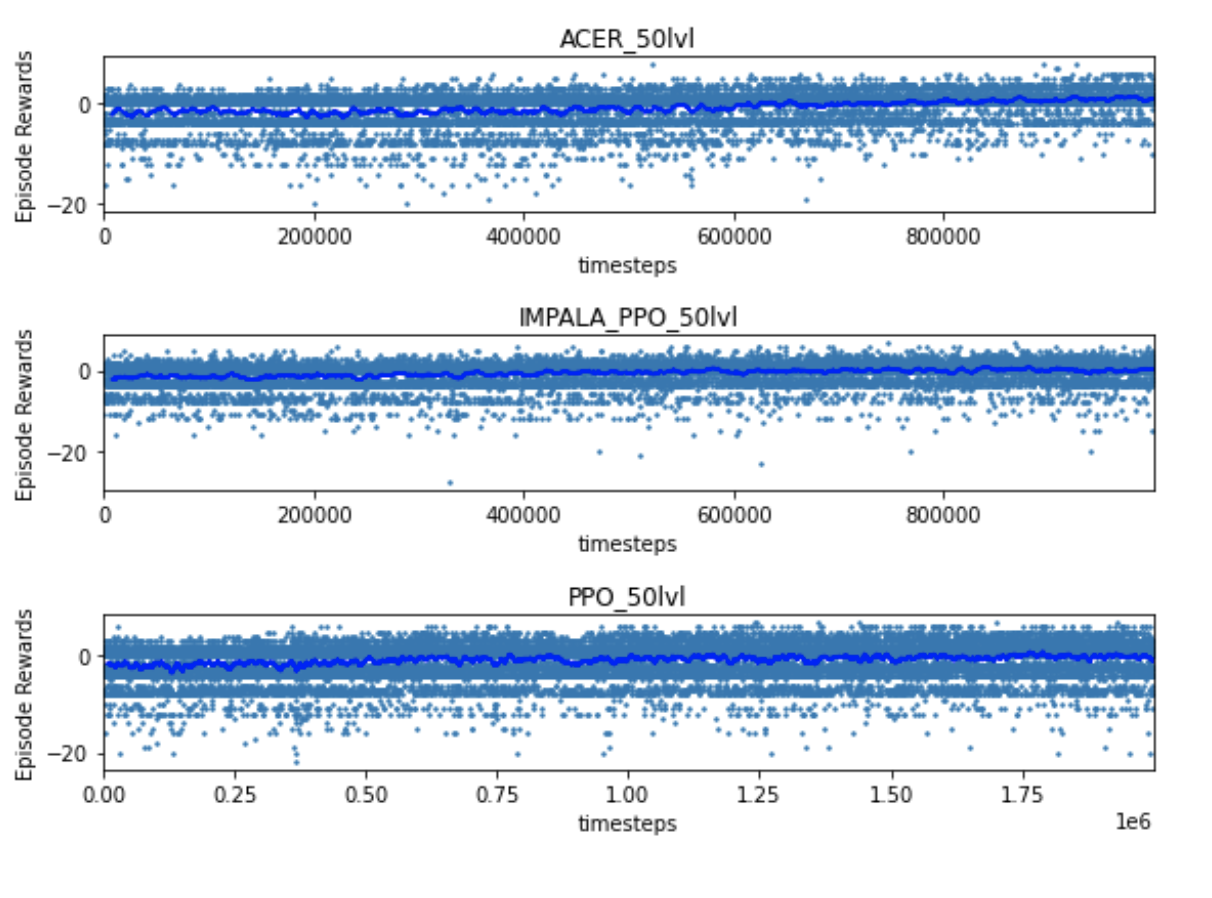
Ofir Nachum and Shixiang Gu and Honglak Lee and Sergey Levine. Near-Optimal Representation Learning for Hierarchical Reinforcement Learning. arXiv:1810.01257, 2018.

Alexander C. Li and Lerrel Pinto and Pieter Abbeel. Generalized Hindsight for Reinforcement Learning. arXiv:2002.11708, 2020.

**Appendix**

**A: Training Curve by Architecture**

****



**B: Hyperparameters for PPO training**

|  |  |
| --- | --- |
| Learning Rate | 5e-4 |
| ent\_coef | 0.01 |
| gamma | 0.999 |
| lam | 0.95 |
| nsteps | 256 |
| nminibatches | 8 |
| ppo\_epochs | 3 |
| clip\_range | 0.2 |
| Use\_vf\_clipping | True |