Run, Sonic, Run!:
Exploring Reinforcement Learning Approaches in Retro Sonic Game

Likai Wei, Chester Mu, Hasith Rajakarunanayake

Abstract

In this report, we will explore different reinforcement learning algorithms and their performances on the video game "Sonic, the Hedgehog-Genesis" in the OpenAI gym retro environment. The report will discuss and present results on models trained on a single level and multiple levels of the game, and will also demonstrate the results of transferring models into new levels that the agent has not been trained on.

Building from JERK and PPO baselines introduced in OpenAI's technical report[1], we implemented an improved version of JERK called JERK2 to better suit the Sonic environment. This algorithm is designed to compensate for a few key deficiencies of the JERK algorithm, and enable it to be easily integrated with a deep learning model, such as PPO2. It gets rid of JERK's tendency to repeat past mistakes that are seen as rewarding, and it promotes further exploration of the environment where rewards are more plentiful. This new agent is shown to outperform the baseline JERK algorithm and is able to achieve states that the PPO algorithm normally would not reach. However, it still does not have any perception of the environment other than the rewards, so it fails to surpass certain geographical features.

To further improve upon the JERK2 algorithm, we have come up with a new algorithm called JERK2_PPO which integrates the JERK2 agent and a pre-trained PPO model. This algorithm essentially allows the JERK2 algorithm to perceive its surroundings with a model trained on several levels and reduces the randomness. This results in more stable behavior. However, when it tends to get stuck, it will act randomly again, which allows the agent to reach new states. This algorithm is meant to mitigate the shortcomings in both algorithms. This agent was implemented with a PPO model trained on a single level and yields better reward comparing to only use a single JERK or PPO model.

To achieve better performance on different levels, we jointly trained PPO and A2C on 13 different Sonic levels on different workers. The jointly trained PPO model has decent performance, even completing some of the levels. However, the jointly trained A2C model does not perform that well and even with worse performance as training timesteps increases. The jointly trained PPO model performed well on its training set, but it fails to perform with zero-shot transfer test due to overfitting.Based on the same concept of JERK2_PPO, we integrate JERK2 with our jointly trained PPO and test the agent on both training levels and brand new levels. The findings were that JERK2_PPO performed similarly to PPO on levels that were in the training set, and outperformed it in levels in the evaluation set.

To further explore the transfer learning performance of our model, we added a replay buffer to allow the joint PPO model sample and learn from past experience. We also attempted to substitute the CNN network we used with a recurrent network (RNN, Lstm, CNNLstm, etc.).

1. Motivation:

Current existing benchmarks for deep reinforcement learning tend to encourage training and testing the agent in the same environment, which can lead to overfitting. Video games can be considered as a great benchmark for testing the trained agents in environments that they have never seen. OpenAI Gym[5] Retro environment provides a collection of video games to test RL agents' generalization ability. In April 2018, OpenAI gym released OpenAI Retro Contest which focused on reinforcement and transfer learning in the "Sonic, the Hedgehog" game franchise. Most of the participants are focusing on swiping different hyperparameters on the provided baselines in order to get higher scores. Our project tries to improve the algorithms to better suit the environment instead of sweeping hyperparameters and doing architecture searches. Our report will compare different algorithms for training a Sonic agent and compares their results to the provided baselines from OpenAI.

2. Introduction:

Figure 1: Screenshots from the game "Sonic the Hedgehog-Genesis", the game character is running towards the right to finish the level.

Gym Retro is a project by OpenAI aimed at emulating different retro video games. Specifically, in "Sonic the Hedgehog-Genesis", the overall goal is to finish the level by reaching the destination located at the right end. Levels in the game are complicated and different, consisting of enemies and obstacles which requires other commands than simply moving to the right.

Observations: A RGB image with size 320 by 224 pixels.

Actions: Valid button combinations selected from {['LEFT'], ['RIGHT'], ['LEFT', 'DOWN'], ['RIGHT', 'DOWN'], ['DOWN'], ['DOWN', 'B'], ['B']}.

Rewards: At each timestep, the reward is given by the horizontal displacement compared to the previous timestep. In other words, there will be a positive award for Sonic to move toward the right and negative to the left. In addition, there is a bonus reward of 1000 for completing each level. The bonus reward decreases as timesteps increase in each episode, encouraging the agent to finish each level as fast as possible. The total reward of each episode is the cumulative rewards of each timestep and the maximum reward for every level is 10000.

3. Training & Testing Single Level

3.1. JERK (Just Enough Retained Knowledge)

The JERK algorithm mimics prior experience to achieve improved rewards. It solely relies on the rewards to make decisions about what actions should be taken. This is done by randomly sampling a trajectory until the rewards stop increasing. When this happens, the agent randomly samples a trajectory in the opposite direction for a short time. Once an episode has ended, the reward and sequence of actions are saved to a list of solutions. When beginning a new episode, the agent has a chance to select the best-saved solution, and repeat it, and stop when it has reached the end of that episode [1].

Require: initial exploitation fraction, β.
Require: consecutive timesteps for holding the jump button, J_n.
Require: probability of triggering a sequence of jumps, J_p.
Require: consecutive timesteps to go right, R_n.
Require: consecutive timesteps to go left, L_n.
Require: evaluation timestep limit, T_{max}.
$S \leftarrow \{\}$, $T \leftarrow 0$.
repeat
  if $|S| > 0$ and $RandomUniform(0, 1) < \beta + \frac{T}{T_{max}}$ **then**
    Replay the best trajectory $\tau \in S$. Pad the episode with no-ops as needed.
    Update the mean reward of τ based on the new episode reward.
    Add the elapsed timesteps to T.
  else
    repeat
      Go right for R_n timesteps, jumping for J_n timesteps at a time with J_p probability.
      if cumulative reward did not increase over the past R_n steps **then**
        Go left for L_n timesteps, jumping periodically.
      end if
      Add the elapsed timesteps to T.
    until episode complete
    Find the timestep t from the previous episode with the highest cumulative reward r.
    insert (τ, r) into S, where τ is the action sequence up to timestep t.
  end if
until $T \geq T_{max}$

Figure 2: Detailed Description of JERK Algorithm[1].

In the sonic environment, this agent performs very well on levels in which randomly moving forward is sufficient. However, it struggles considerably when faced with certain obstacles where specific actions are necessary. For example, some levels require a significant amount of time moving backward, and there are also times where no horizontal movement is desired.

Another glaring weakness of this algorithm is that once a prior agent has repeated a previous solution if the episode is not yet over, it simply waits for the episode to end. Because of this, only randomly sampled trajectories will result in an improvement from prior experience.

A third weakness of the JERK algorithm is that it blindly replays the best previous solution that it previously achieved. Sometimes, the best prior solution is still a very bad one. For example, when running this baseline algorithm, sonic would run into spikes. Because the spikes were the most rewarding location sonic had reached in the past, he would constantly run into the same spikes. In this project, we aim to eliminate these faults.

3.2. JERK2

We have used the JERK algorithm as a starting point to develop a new algorithm which mitigates the weaknesses discussed in section 3.1. The main strength of JERK is that once it learns a good trajectory, it can reliably repeat that trajectory, and very quickly starts exploring new areas. It does this through the saved best trajectory mechanism. A feature of this new algorithm is that instead of simply replaying the old trajectory out of the box, it randomly selects a segment of the trajectory. This segment starts at the beginning, and the end is determined by a triangular probability distribution, with the highest probability being at the end, and the lowest probability being somewhere in the middle. The result of this is that the algorithm favors further exploration near the end of the good trajectory, resulting in fewer deaths and reducing the number of times sonic gets stuck in one area. A triangular probability distribution was selected because it favors using a larger portion of the chosen trajectory.

In the JERK algorithm, there is a random exploration to search for good trajectories, and then when the agent exploits a previous trajectory, it plays the full episode and stops. In this modified algorithm, we promote further exploration after the trajectory has been replayed. In order to do this, after the commands from the exploit have been exhausted, the agent returns to a state of random exploration, and begins appending new moves to its current trajectory. This, when combined with the fact that the agent only replays a segment of the trajectory, allows the agent to train much quicker than standard JERK, because it always explores new areas around the area that resulted in the highest reward previously.

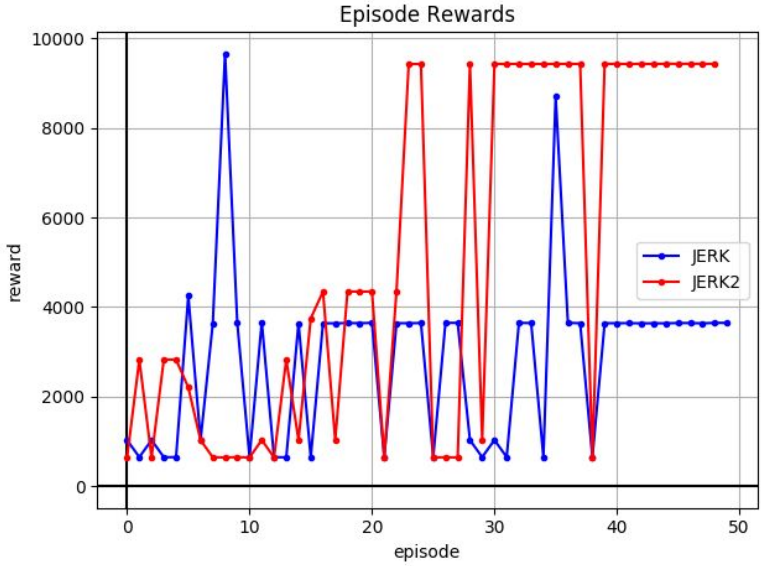

Figure 3: Performance of JERK2 vs JERK. Over the course of many episodes, the JERK2 algorithm explores more and finds easier ways to reach the end of the level. This results in more consistent rewards than the baseline JERK algorithm.

3.3. PPO

Proximal Policy Optimization(PPO or PPO2)[2] is an algorithm which is based on Policy Gradient with a constrained objective function to prevent excessively large policy updates. The objective function for PPO is as follows:

$$L^{CLIP}(\theta) = \widehat{\mathrm{E}_t}\left[\min\left(r_t(\theta)\widehat{A_t}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A_t}\right)\right]$$

The probability ratio is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$$

The PPO baseline on a single level is provided by OpenAI for the contest. Inspired by other participants, we decide to pre-process the images for our training. The observations will be resized into 96x96 pixels and converted into grayscale for faster training, and we scale the reward by 0.01 in order make the advantages within a reasonable range for our neural networks. The single level model is trained on "GreenHillZone Act1" for 1 million timesteps.

Running the model on the training level, we get returns around 5000 with very small variance. The reason is that the trained model gets stuck when the agent encounters a loop, which requires the agent to go backward and then keep going right. This result exposes a weakness of the PPO method: The agent will easily get stuck where the level requires moving left to pass obstacles because of the negative reward for moving left.

Figure 4: Loop in GreenHillZone.Act1, where requires the agent to go left and then continuously move right to pass it.

3.4. JERK2_PPO

In order to mitigate the drawbacks from JERK and PPO algorithms, we have come up with a new approach called JERK2_PPO. JERK2_PPO integrates a pre-trained PPO model inside the JERK2 algorithm. When the agent is planning to move towards the right, it will choose from either generating a random right & jump actions sequence as in JERK2 or predicting actions based on its current observation using the pre-trained PPO model. This approach not only reduces the randomness introduced by the JERK while picking actions but also enable the PPO model to backtrack and exploit saved best trajectories. In our current setup, the probability of

choosing a random action sequence by JERK is set to 30%, and 70% for predicting by PPO model. It will then continue with the decided upon policy for a set number of timesteps, before making the decision again. If a move does not result in a positive reward overall, the agent will backtrack a certain number of timesteps, jumping with a given probability, currently set to 2.5%. Furthermore, the agent will memorize its best trajectory and exploit it again with a probability.

We have tested our JERK2_PPO algorithm on “GreenHillZone Act1”, and compared its performance with the case of solely using JERK2 or PPO.

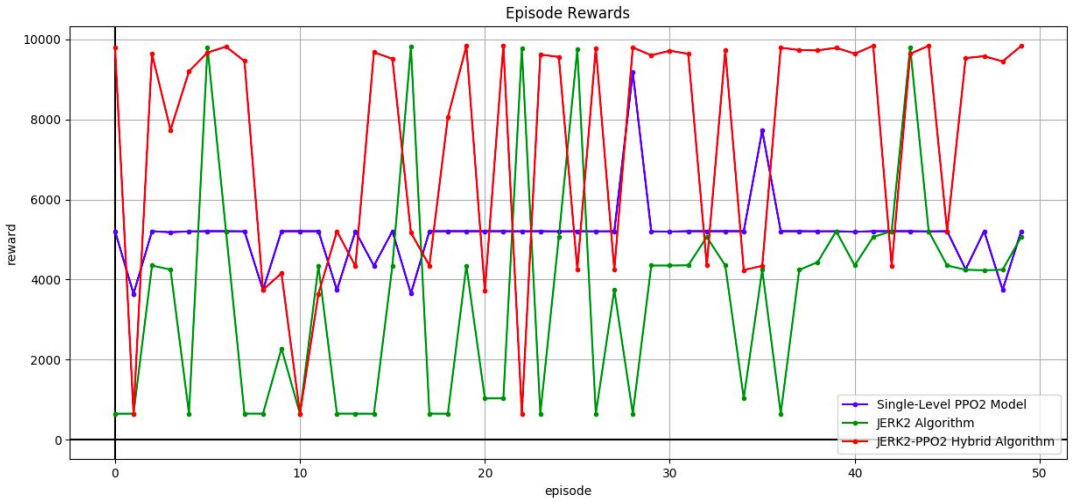

Figure 5: Plot showing the returns for three algorithms over 50 episodes. The PPO2 model (blue) is very consistent, but rarely passes the middle of the level. The JERK2 algorithm successfully reaches new states but it is rare. The combined JERK2_PPO algorithm is able to reach the end of the level and consistently repeat the performance.

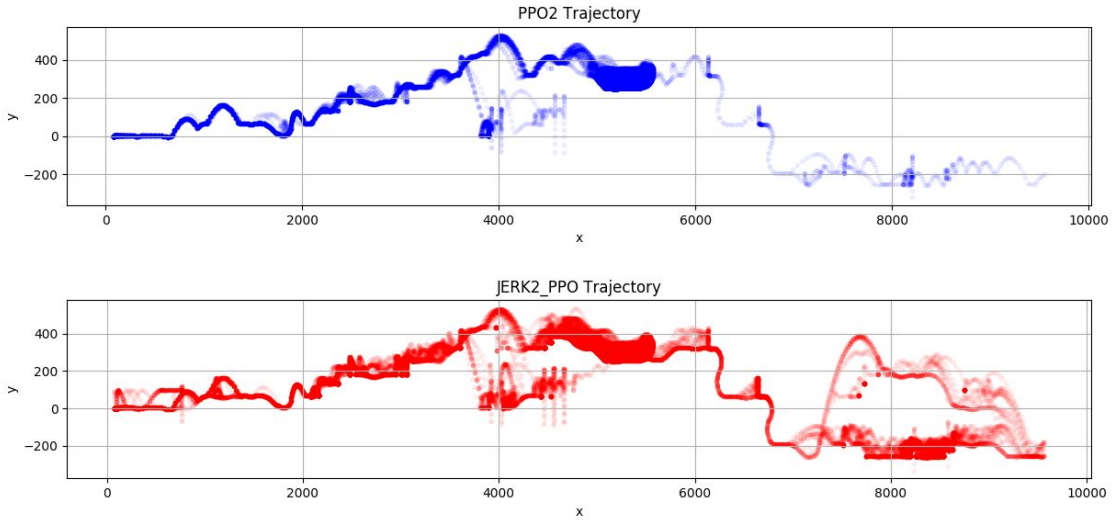

Figure 6: This plot shows the trajectory of two agents when running for 50 episodes. Darker colors correspond to areas that have been visited more often. The PPO2 model (top) tends to get stuck in the middle. However, the JERK2_PPO2 model (bottom) is able to reach more states and finish the level.

4. Training & Testing on Multiple Levels

4.1. Jointly trained A2C

At this stage of the project, it has been pretty obvious for us that using a value-based policy is not a good idea as it would be susceptible to all of the sudden reward changes from predicted actions as the sonic game environment is exceptionally complicated. But before we move on to modify and train the PPO model, we first implemented and tested a relatively more basic algorithm, A2C, an actor-critic method fitting on the advantage function. There are mainly two motivations behind this. Firstly, it is in general faster to train an A2C model. Since we have limited computing resources and need to play with the policy update algorithm, changing the game environments (reward and action space) and tuning model hyperparameters, training A2C models gives us more chances to do all this within less time. Secondly, we want to set up a fair comparison with the PPO policy that is to be trained on multiple training levels and justify how the clipping range performs comparing with A2C.

Similar to the actor-critic model introduced in the class earlier in the semester, A2C is just an actor-critic model but trains on the advantage function with a baseline of average reward value at each state to make the learning process experience less variance. For example, for the actor policy update, we have:

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s,a))A(s,a)$$

where the advantage function is:

$$A(s,a) = Q(s,a) - \tilde{V}(s)$$

And we can further approximate the above function with TD error:

$$A(s,a) = r + \Upsilon * V(s') - V(s)$$

Since here we are training on multiple levels, it may be helpful to briefly mention A3C, an asynchronous model of the Advantage actor-critic network here to further justify our implementation choice. Unlike A2C where all the agents wait to update the policy together, in A3C, all the runner agents get a copy of the policy and update it asynchronously. We chose A2C over A3C because we think it should be better to update the policy with one averaged gradient from all runner agents for a more generalized policy since we are training the policy on multiple levels. Considering that A2C/A3C does not have a clipping range to regularize policy updates like PPO, we chose A2C over A3C to expect to avoid unrecoverable change to the policy from each update.

We trained and tested the A2C on a single level first and then did a 13-level joint training. The results and some observations for the jointly trained models are shown below.

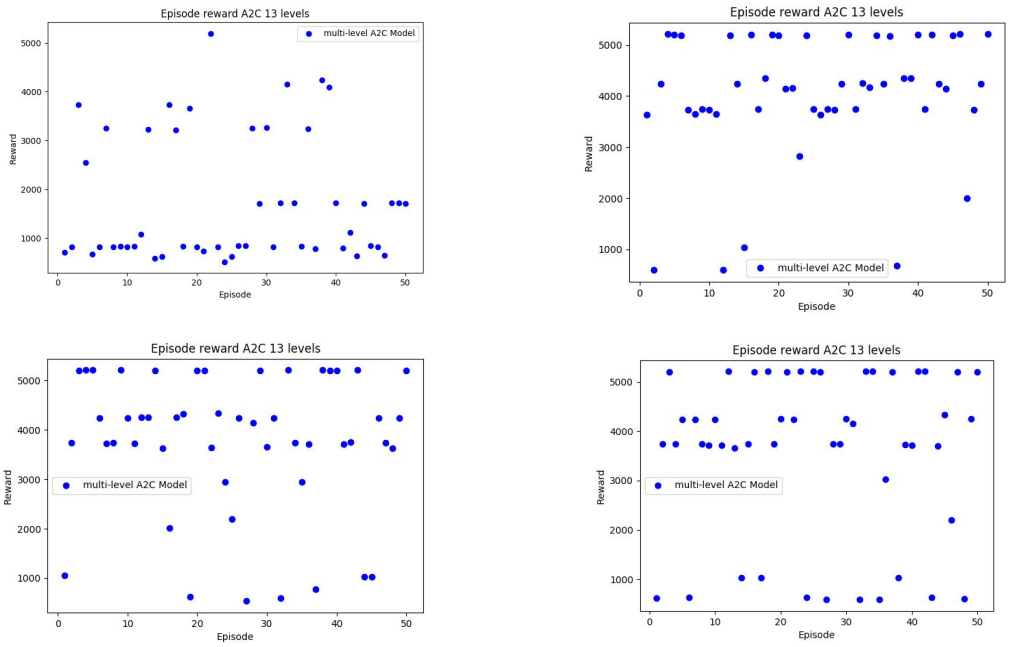

Figure 7: Rewards over 50 episodes run on the level "GreenHillZone Act 1" for the model trained on 13,000,000 total time steps, 10,000,000 total time steps, 8,000,000 total time steps, and 6,500,000 total time steps, respectively.

From the results from training and testing on a single level (not shown here), we observed serious overfitting at larger training time steps over 500,000. So we paid special attention here to the jointly trained model. And from the results shown above in Figure 6, we can see that the A2C model trained on 13 levels is also susceptible to overfit, the model trained on 13,000,000 total time steps gets stuck in an early stage of the level and is outperformed by the models trained with less total timesteps. This is also further proved by testing on unseen validation levels, where the most trained model achieves lower scores, and thus we choose to report the testing results from a 10,000,000 time steps trained model in the Appendix.

Finally, we ran the A2C model trained on 10,000,000 total time steps on new levels that belong to the validation set to see how this model transfers. From the results shown in the appendix Table 4, the joint A2C model does not run as well as a joint PPO model to be shown in later sections. One reason might be that without a clipping range, A2C model might experience big updates that cannot be recovered, which might makes the model worse in generalization.

Overall, these results set a good baseline for us to compare with the joint PPO, joint PPO-Jerk, joint PPO with replay buffer models to be introduced later.

4.2. Jointly trained PPO

As we learned from the previous section, PPO agent trained on a single level has its limitations on playing the levels. In order to generalize more information about different levels

and achieve the goal of transfer learning, we want to train our model on multiple levels. Researching technical reports from other participants, we learned that meta-learning algorithms such as MAML[3] and Reptile[4] cannot yield good results on the Sonic environment.

As a result, we implemented a simple meta-learning approach as described in [1] to jointly train a PPO model on 13 different Sonic levels. We ran all training levels in parallel on 13 workers for 13 million total timesteps with same image preprocessing and rewards scaling as in 3.3. The gradients will be averaged across all 13 workers after each gradient steps, making sure the policy is not biased towards any level.

The jointly trained PPO model performance has better performance on the training set compared to the single level one. Take "GreenHillZone Act1"(the same level as in section 3.3) as an example, the jointly trained model is able to finish the level every time, which almost doubles the reward of the single-level model.

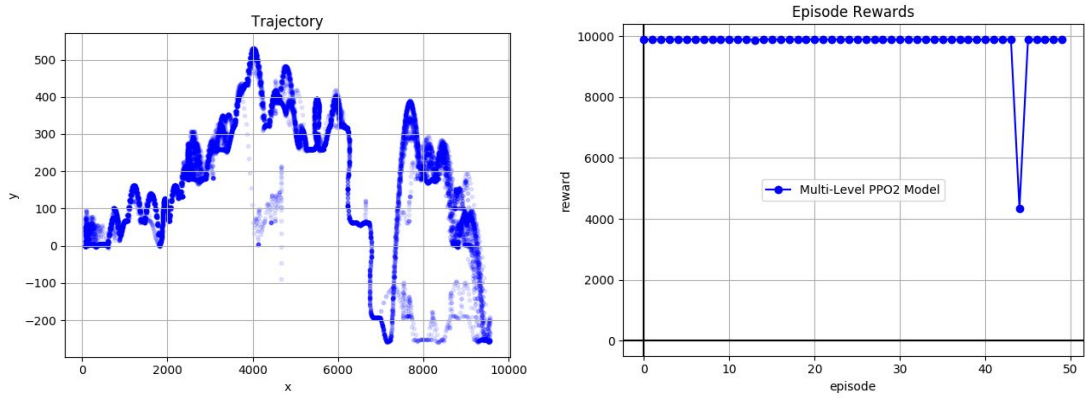

Figure 8: Tractories and Episode Rewards Plots over 50 episodes for jointly trained PPO on "GreenHillZone Act1"

However, the jointly trained PPO model is having difficulty while testing on new levels. Overfitting might be the cause of this issue, preventing us from running zero-shot transfer learning by this model. Unlike the joint PPO model by OpenAI which is trained by 188 workers on 47 levels, our model is only trained on 13 workers with 13 levels due to hardware limitations. It is possible that 13 levels are not enough for generalizing Sonic level. The average performance of this model tested on different levels is listed in Appendix B.

4.3. JERK2 with Jointly trained PPO Agent

After verifying JERK2_PPO is able to yield positive outcomes in the single level case, we wanted to extend this approach by replacing the PPO model with the one that we mentioned in section 4.2. In this section, we will discuss the performance of this method on both seen and unseen levels.

The JERK2_PPO agent with the jointly trained PPO agent performs similarly to the PPO agent on the testing set. However, when transferring to the evaluation set, the combined agent

outperforms it significantly. This is because the jointly trained PPO model is very good at making decisions on environments it has seen before, but not on new ones. However, integration with the JERK2 algorithm allows the agent to explore areas in the stage that it could not reach before. Also, the exploitation of prior rewards allows the agent to reliably repeat its successes when it luckily passes certain obstacles. The performance is illustrated in tables 2-5 in the Appendix.

For example, the JERK2_PPO model outperforms jointly trained PPO2 on StarLight Zone Act2. The PPO model is not able to pass or kill the enemy at the very beginning of the level, yielding a average rewards around 0. In contrast, JERK2_PPO is able to pass by enemy by introducing randomness in it action and exploiting recorded trajectories.

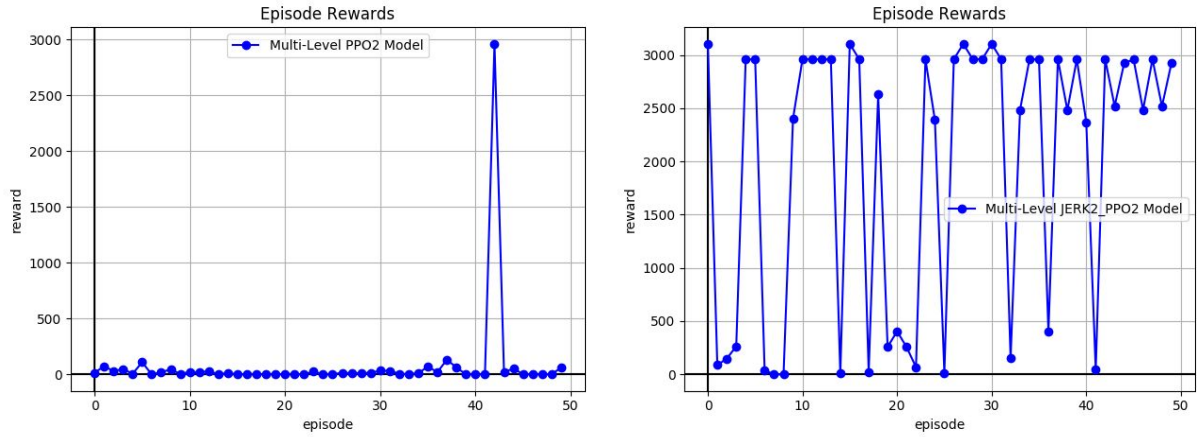

Figure 9: Episodes Reward Plots on unseen level StarLight Zone Act2.

4.4. Jointly trained PPO agent with replay buffer

At a later stage of our project, we further proceeded to modify our model taking inspiration from **HW5c: Meta-Learning** to see if we could get the model better generalized and thus perform better in transfer learning.

Inspired by HW5, we first tried to use a Recurrent Neural Network instead of the modified CNN used previously. Firstly we implemented a regular RNN. However, although we seemed to be able to initiate and pass the hidden states in the network, the code does not run properly with multiple errors. Since we were running short on time, we adapted the **LSTM** network provided by the OpenAI baseline package. After making some minor tweaks and fixing all the dimension related problems, we managed to get the code running to train. However, we constantly got OOM (out of memory) errors when storing tensors at certain steps. The problem also happens with a similar **CnnLSTM** network. We did some further digging but still could not get the training to work.

Eventually, we put this approach on hold to move on to a simpler approach also inspired by HW5, building a **replay buffer**. Now we have the previous joint PPO model sample batches from its experience memory and train on that. With this replay buffer, we hope for the PPO

agent to learn some transitions from what happened before and thus do a better job at unseen sonic game levels. Additionally, sampling from memory allow us to mitigate much of the correlation of samples since we would be training our model on more independent samples.

Unfortunately, this part takes a long time to train, and we could not finish all the training we intended by the due date of this report. The results we obtained so far (at ~11,000,000 total time steps) are reported in the Appendix Table 7 of this report. With 11,000,000 time steps, our model is not "fully trained" and thus the reported performance does not show improvement over the regular joint PPO model in all levels. However, in certain levels, like StarLightZone.Act3, where an agent could easily die at the very start of the level, a joint PPO agent seems to benefit from learning older experience and can perform better in general by avoiding dying at the beginning.

Another observation is that the model we have so far for a joint PPO agent trained with replay buffer, seems to perform much better in the beginning than later in some levels. One possible reason could be that our current implementation has a very large size of replay buffer, and with the sampling from past experience, policy would need more time steps to train in order to gain more experience deeper into the level.

We will wait for the training to finish and fully compare the performance further.

5. Conclusion

In this project, we have analyzed a number of different reinforcement learning algorithms in the context of the Sonic the Hedgehog benchmark. We identified several weaknesses and strengths in existing algorithms, and came up with new algorithms that combined the strengths and minimized the weaknesses of each algorithm. Using JERK as a baseline, we created a JERK2 algorithm which was more robust and able to explore new environments, and learn from past successes. We then integrated this with a PPO2 model in order to reduce randomness and allow the agent to perform more consistently. The resulting agent was much more effective than the base PPO2 or JERK2 algorithms when transferred to new environments. We also added replay buffers to the joint PPO model for transfer learning purpose.

6. Future Work

There is still room for improvement in the JERK2 algorithm. In several levels, sonic is required to move left for long periods of time. A possible direction of research would be to explore a JERK algorithm that does not hard-code any behavior, but moves in one direction freely, and only switches directions if the rewards decrease significantly.

The PPO model can also be improved in many aspects. First of all, we have the model trained on more levels to generalize better about the Sonic game franchise. To improve the performance of transfering, instead of doing zero-shot tests, we can fine-tune the model on the testing set in order to provide an initialization for the transferred model. More image preprocessing, reward function modification and changes in action space can be done to make

the trained model better suit the Sonic environment. Lastly, different meta-learning and multi-task learning strategies can also be explored and tested to make further improvements.

We adapted the A2C algorithm to train on multiple levels and tested its transfer learning performance. For future work, we could also try to add the replay buffer to it and see if and how the performance would improve on unseen levels.

We could also devote more time into further exploring the properties of replay buffer. Future work may include testing how the size of the buffer influence model performance. Additionally, we could also proceed to prioritize certain experience to make the learning from memory more effective and efficient. For instance, we could stress on training on experience with higher policy loss.

Last but not least, we can spend more time in the future to get a recurrent neural network to run properly with our current models. The current existed packages have poor support for recurrent network implementations.

7. Acknowledgments

Thanks to Stable Baselines(*stable-baselines.readthedocs.io*) for creating an easy-to-use packages which helps with our implementations.

Thanks to OpenAI for detailed blogs A2C(*blog.openai.com/baselines-acktr-a2c/*) and PPO(*blog.openai.com/openai-baselines-ppo/*)algorithms.

8. Work Distribution

Likai Wei: Reviewed literatures for exploring possible approaches; worked on development, tuning and testing of single-level and multi-level PPO models training; helped with testing and debugging JERK & PPO integration; analyzing results from each algorithm.

Chester Mu: Worked on development of A2C training on single and multiple levels, worked on implementing replay buffers for joint PPO model, worked on attempts to implement various recurrent network, and also worked on model testing and comparison.

Hasith Rajakarunanayake: Worked on development of JERK2 algorithm, and integrated J ERK2 algorithm with PPO2 models. Also worked on testing of various models to determine failures and strengths in each type of agent.

References

1. A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta Learn Fast: A New Benchmark for Generalization in RL", 2018. eprint: arXiv:1804.03720.
2. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and Oleg Klimov, "Proximal Policy Optimization Algorithms", 2017. eprint: arXiv:1707.06347.
3. C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," 2017. eprint: arXiv:1703.03400.
4. A. Nichol and J. Schulman, "On first-order meta-learning algorithms," 2018. eprint: arXiv:1803.02999
5. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016. eprint: arXiv:1606.01540.
6. Adam Stooke, Pieter Abbeel, "Accelerated Methods for Deep Reinforcement Learning". March 2018. eprint: arXiv:1283.02811v1
7. Ruishan Liu, James Zou, "The Effect of Memory Replay in Reinforcement Learning", Oct 2017. eprint: arXiv:1710.06574v1
8. Chiyuan Zhang, Oriol Vinyals, Remi Munos, Samy Bengio, "A Study on Overfitting in Deep Reinforcement Learning", Apr 2018. eprint: arXiv:1804.06893v2

Appendix

A Hyperparameters

PPO(single level)		PPO(13 levels, w/ replay buffer)	
Hyperparameters	Value	Hyperparameters	Value
Horizon	4096	Horizon	2048
Epochs	3	Epochs	4
Minibatch Size	8	Minibatch Size	16
Discount(Gamma)	0.99	Discount(Gamma)	0.99
GAE parameter(Lambda)	0.95	GAE parameter(Lambda)	0.95
Clipping parameter(Epsilon)	0.1	Clipping parameter(Epsilon)	0.1
Entropy Coefficient	0.01	Entropy Coefficient	Value
Learning Rate	0.0002	Hyperparameters	2048
Reward Scaling	0.01	Horizon	4

Table 1: Hyperparameters used for PPO trainings

B Performances of different algorithms

Level	Average Reward	Level	Average Reward
SpringYardZone.Act 3	2578±390	MarbleZone.Act2	2265±274
SpringYardZone.Act 2	2210±1597	MarbleZone.Act1	5177±1107
GreenHillZone.Act3	7555±3954	MarbleZone.Act3	2927±495
GreenHillZone.Act1	9333±1936	ScrapBrainZone.Act2	1257±139
StarLightZone.Act2	1805±188	LabyrinthZone.Act2	2967±19
StarLightZone.Act1	5313±1166	LabyrinthZone.Act1	3906±675
LabyrinthZone.Act3	2157±926	**Aggregated Mean**	**3803**

Table 2: PPO(trained on 13 levels)on training levels over 50 episodes:

Level	Average Reward	Level	Average Reward
SpringYardZone.Act 3	2413±362	MarbleZone.Act2	2299±131
SpringYardZone.Act 2	2104±1252	MarbleZone.Act1	3985±1562
GreenHillZone.Act3	7552±3607	MarbleZone.Act3	2798±590
GreenHillZone.Act1	7673±3357	ScrapBrainZone.Act2	1238±60
StarLightZone.Act2	4193±1974	LabyrinthZone.Act2	2728±642
StarLightZone.Act1	5301±1163	LabyrinthZone.Act1	3278±1002
LabyrinthZone.Act3	2104±970	**Aggregated Mean**	**3665**

Table 3: JERK2_PPO(trained on 13 levels)on training levels over 50 episodes:

Level	Average Reward	Level	Average Reward

SpringYardZone.Act 1	542±431	StarLightZone.Act3	2397±825
GreenHillZone.Act2	2662±520	ScrapBrainZone.Act1	797±413

Table 4: A2C(trained on 13 levels 10,000,000 timesteps)on unseen levels over 50 episodes

Level	Average Reward	Level	Average Reward
SpringYardZone.Act 1	310±427	StarLightZone.Act3	79±416
GreenHillZone.Act2	4490±1117	ScrapBrainZone.Act1	915±506

Table 5: PPO(trained on 13 levels)on unseen levels over 50 episodes

Level	Average Reward	Level	Average Reward
SpringYardZone.Act 1	949±885	StarLightZone.Act3	1917±1294
GreenHillZone.Act2	3438±1779	ScrapBrainZone.Act1	1355±726

Table 6: JERK2_PPO(trained on 13 levels)on unseen levels over 50 episodes

Level	Average Reward	Level	Average Reward
SpringYardZone.Act 1	648±654	StarLightZone.Act3	1969±1153
GreenHillZone.Act2	2244±1271	ScrapBrainZone.Act1	662±277

Table 7: PPO with replay buffer(trained on 13 levels 11,000,000 timesteps) on unseen levels over 50 episodes