GPT2PPO: Auto Regressive Proximal Policy Optimization

1st Andrei Cozma
Department of Electrical Engineering & Computer Science
University of Tennessee
Knoxville, United States
acozma@vols.utk.edu

2nd Hunter Price
Department of Electrical Engineering & Computer Science
University of Tennessee
Knoxville, United States
hprice7@vols.utk.edu

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—Reinforcement Learning, PPO, GPT2**

I. Introduction

In this project we explore the use of transformers in the context of Reinforcement Learning. The majority of theoretical works assume that problems follow a Markovian process, which is not always the case. Some problems need the contxt of previous states and actions to make an informed decision on the next decision. As a result, we propose an addition to the basic Proximal Policy Optimization (PPO) algorithm by using the Generative Pre-trained Transformer 2 (GPT2) model as the encoder for the critic network. This will allow the critic network to take into account the context of previous states and actions as well as apply attention to past states and actions that may be important. We will test this model on the LunarLander-v2 and Acrobot-v1 OpenAi Gym environments with discrete action spaces and compare it to the original PPO algorithm. Additionally we will test the model on BipedalWalker-v3 with continuous action spaces and compare it to the original PPO algorithm.

II. Previous Work

III. Background

All of the environments used in this project are from the OpenAI Gym library [1]. The environments are described below.

A. Lunar Lander

The Lunar Lander environment is a rocket trajectory optimization problem[1] shown in Figure 1. The goal is to actuate the lander to the landing pad at coordinates (0,0) without crashing. OpenAI Gym offers two versions of the environment: discrete or continuous. In this work we only use the discrete version. The state space is a 8-dimensional vector containing the x and y positional coordinates of the agent, its x and y linear velocities, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not. The action space is a single discrete scalar with values ranging from 0 to 3. The values corresponspond to the following actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. The reward structure contains both positive and negative rewards. If the lander moves away from the landing pad, it gains a negative reward. If the lander crashes, it receives an -100 reward. If it comes to rest, it receives an +100 reward. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Firing the side engine is -0.03 points each frame. The landers initial state is at the top center of the environment with a random intial force applied to its center. The episode ends if the lander crashes, goes outside of the viewport, or comes to a resting position.

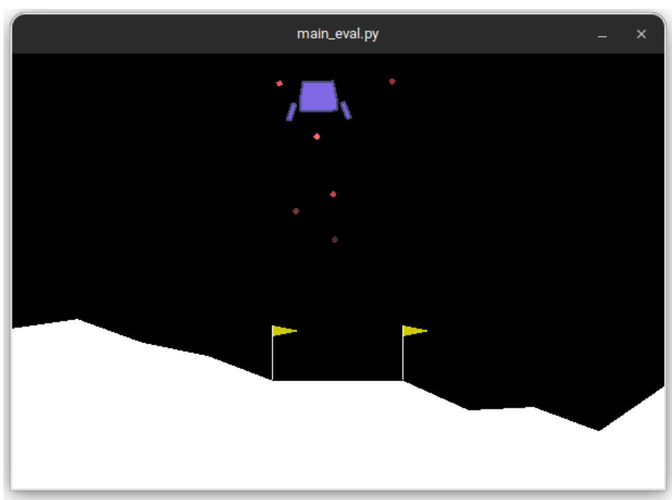

Fig. 1. The Lunar Lander environment.

B. Acrobot

Open AI Gym's implementation of the Acrobot environment[2] is based off the work of Sutton and Barto [2] shown

[1]OpenAI Gym Lunar Lander: https://www.gymlibrary.dev/environments/box2d/lunar_lander

[2]OpenAI Gym Acrobot: https://www.gymlibrary.dev/environments/classic_control/acrobot

in Figure 2. The environment is a 2-link pendulum with only the second joint actuated with a discrete action space. The goal is to swing the end of the pendulum up to a given height. The state space is a 6-dimensional vector containing the sin and cos of the two joint angles and the joint angular velocities. The action space is a single discrete scalar with values ranging from 0 to 5. The values corresponspond to the following actions: apply +1, 0, or -1 torque to the actuated joint. The reward structure contains only negative rewards. At each timestep the agent receives a reward of -1 for each step that does not reach the goal. If the goal is reached the agent receives a reward of 0. The episode ends if the agent reaches the goal height or if the episode exceeds the maximum number of timesteps.

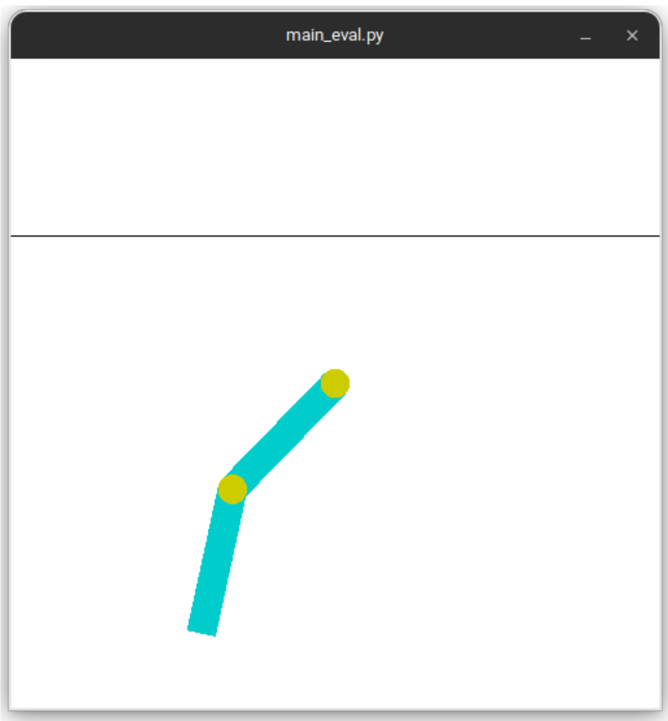

Fig. 2. The Acrobot environment.

C. Bipedal Walker

The Bipedal Walker environment is a 2D simulation of a bipedal walker robot with 4-joints[3] shown in Figure 3. The goal is to keep the walker upright for as long as possible. The state space is a 24-dimensional vector containing: the hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. Notably, there are no coordinates given in the state vector. The action space is a 4 dimensional vector containing continuous values of each joints motor speed between -1 and 1. The reward structure contains both positive and negative rewards. A positive reward is given for moving forward. If the agent falls it receives a reward of -100. Applying motor torque costs a small amount of reward. The agent's intial state is standing at the left of the terrain with the hull horizontal, with both legs in the same position with a slight knee angle. The episode terminates if the agent's hull makes contact with the ground or if the agent reaches the end of the terrain length.

Fig. 3. The Bipedal Walker environment.

IV. Methodology

A. Reinforcement Learning Methods

V. Code Design

A. Utilized Libraries

In this work we chose to use pytorch as our main deep learning library. Pytorch is a popular deep learning library that is well documented and has a large community. To help maintain the structure of the models we also used the Pytorch-Lightning library. Pytorch-Lightning is a high-level library that allows for easy training and testing of models. Pytorch-Lightning also provides a module named Lightning-Bolts that contains many pre-built models and utilities. For the GPT2 implementation we used the HuggingFace Transformers library. We use the OpenAI Gym Library to provide the environments for our models to train and test on [1]. Finally, we use wandb to log the results of all of our experiments [3].

B. File Structure

This works file structure is as follows. The main directory contains the **main_train.py** and **main_eval.py** files. These files are used to train and test the models respectively. The core code is contained within the **rllib** directory. The Within the **rllib/examples** contains an A2C and PPO baseline implementations that were taken from the Lighting-Bolts library, copied into our project.

Within the **rllib** directory there are many files. The important files are the files that are primarily used are: **Model.py**, **GPT2PPO.py**, **CommonGPT2.py**, **CommonBase.py**, and

[3]OpenAI Gym Acrobot: https://www.gymlibrary.dev/environments/box2d/bipedal_walker

GPT2.py. These files will be discussed in more detail in the following sections.

C. Training and Testing Tools

To train a model we use the **main_train.py** file. This file will train a model on a given environment and log the results. To train a model we use the following command:

```
python3 main_train.py \
    -m ppo_gpt \
    -e LunarLander-v2
```

For the **main_train.py** file we have the following additional command line arguments:

- `-e` or `--env` - The Open AI Gym environment [Default: LunarLander-v2]
- `-m` or `--model_name` - The name of the model to train [Default: ppo_ex (PPO)]
- `-ne` or `--num_epochs` - The number of epochs to train for [Default: 150]
- `--al_check_interval` - The itervale of epochs to run validation [Default: 5]
- `--wandb_project` - The wandb project to log to [Default: rl_project]
- `--wandb_entity` - The wandb entity to log to [Default: ece517]
- `--seed` - The seed value for training [Default: 123]

The training command will train the model then save the trained model to the checkpoints directory with the following structure:

```
checkpoints/
<model_name>/
<env_name>/
<model_hash>.pt
```

To test a model (watch it interact in the environment) we use the **main_eval.py**. This file will load a model from the checkpoints directory and run it in the environment. To test a model we use the following command:

```
python3 main_eval.py -f <model_checkpoint>
```

For the **main_eval.py** file we have the following additional command line arguments:

- `-f` or `--file_path` - The path to the model checkpoint
- `-ne` or `--num_episodes` - The number of episodes to run [Default: 5]
- `--running_rew_len` - The length of rewards to store at once [Default: 50]
- `--seed` - The seed value for testing [Default: 123]

Both the **main_train.py** and **main_eval.py** files utilize the **rllib/Model.py** file. This file abstracts the training and evaluation process from the training and testing commands into a class named Model. It simply loads in the correct model and trains or tests it.

D. Models

The baseline PPO model we use to compare against our results is in the **rllib/examples/PPOExample.py** file. In it contains code taken from the Lighting-Bolts library with no modifications. We store this model locally to protect against any future api changes in the library. The file contains a class named PPO that inherits from the LightningModule class. Our implementation is builds upon this class.

This works implementation resides at **rllib/GPT2PPO.py**. This file contains the GPT2PPO class which implements the same interface as the reference PPO. Apart from the existing MLP, ActorContinuous, and ActorCategorical models used in the baseline PPO implementation that are given by the Lighting-Bolts library, we use 2 additonal models that are implmented in the **rllib/CommonBase.py** file and the **rllib/CommonGPT2.py** file.

The **rllib/CommonBase.py** file implements the CommonBase class. This class implements a simple Sequential model containing a Linear layer with a relu activation function followed by a Linear layer with no activation function. This class is used as the the base of the ActorCategorical and ActorContinuous models. States are fed through this and immediately given to the actor.

The **rllib/CommonGPT2.py** file implements the CommonGPT2 class. This class implements a usage of the GPT2 model which is stored in the **rllib/GPT2.py** file. The GPT2 model is taken from the Decision Transformer implementation [4]. In that work the authors took the huggingface GPT2 model and removed the logic that implment positional embeddings [5], [6]. Our CommonGPT2 shares large similarities with the Decision Transformer implementation [4]; however, we chose to only feed the states and actions to GPT2 rather than the states, actions, and rewards to go. As input, CommonGPT2 will take a history of timesteps, states, and actions as input of shape (batch_size, sequence_length,:). It will then return an embedding of the state and action. The embedding of the state is then fed to the critic and the critic predicts the value of the state.

GPT2PPO has some differences from the baseline PPO in how the internals are structured. Because we are using a Transformer we need to keep track of the history of the states, actions, timesteps, and attention masks used by the model. We do this by storing each of the respective histories in a buffer of a length equal to the context length of the GPT2 model. When the buffer is full, the oldest entry is removed and the new entry is added to the end of the buffer. The buffer is then fed to the GPT2 model and the most current state is fed through the CommonBase then the actor. The GPT2 model will then return an embedding of the state and action. The embedding of the state is given to the critic. All inputs are saved in a history array which is later used for training. There are two main functions for data generation and training. The **generate_trajectory_samples** function is used to generate data for training. It runs a configured number of timesteps then yields the model inputs, attention masks, actions, log probabilities, Q values, and Advantages for all timesteps. The **training_step** function takes a batch of data and calculates and returns the loss for the actor or critic. This loss is is used by the Pytorch-Lightning framework to update the actor or critic.

VI. Results

In this work we compare the results of our GPT2PPO model against the baseline PPO model. We compare the results of both models on the Lunar Lander, Acrobot, and environments. To keep all of our results consistent, when comparing models we ran each model with the same hyperparemters for the same number of epochs. The hyperparameters for both models are listed in Table I. We ran each model 3 times and averaged the results.

TABLE I
Model Hyperparameters

Hyperparameters	
Param	***Value***
Environment Seed	123
Epochs	150
Gamma	0.99
Lambda	0.95
Batch Size	128
Actor Learning Rate	3e-4
Critic Learning Rate	1e-3
Max Episode Length	500
Steps Per Epoch	2048
Training Steps Per Epoch	5
Clip Ratio	0.2
Hidden Size	64
Context Length*	64

*Context Length only used for GPT2PPO.

A. Lunar Lander

Fig. 4. Lunar Lander - PPO vs GPT2PPO - Critic Loss

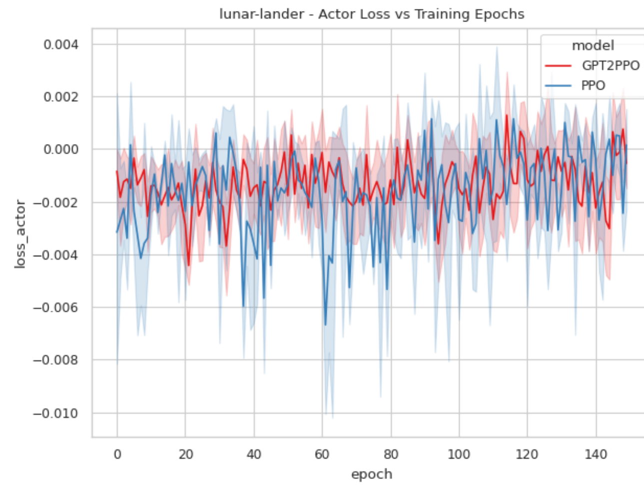

Fig. 5. Lunar Lander - PPO vs GPT2PPO - Actor Loss

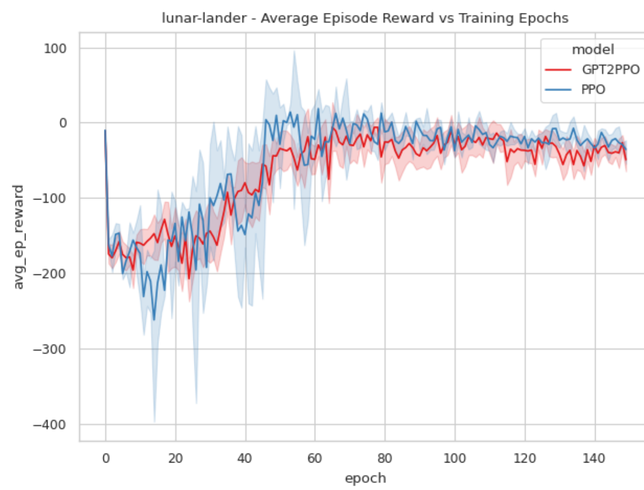

Fig. 6. Lunar Lander - PPO vs GPT2PPO - Average Episode Reward

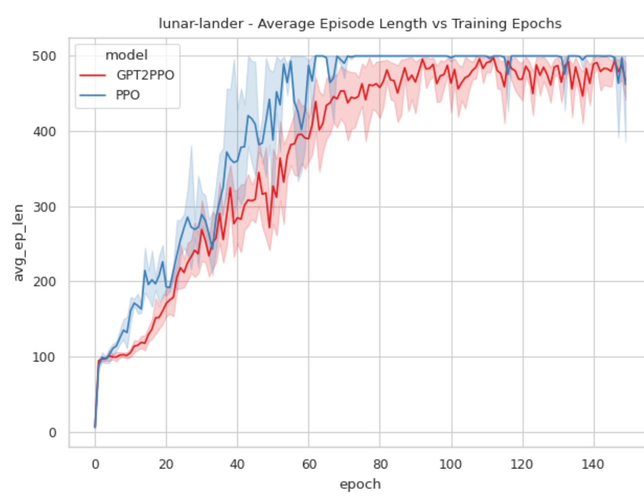

Fig. 7. Lunar Lander - PPO vs GPT2PPO - Average Episode Length

Fig. 8. Lunar Lander - GPT2PPO Context Length - Critic Loss

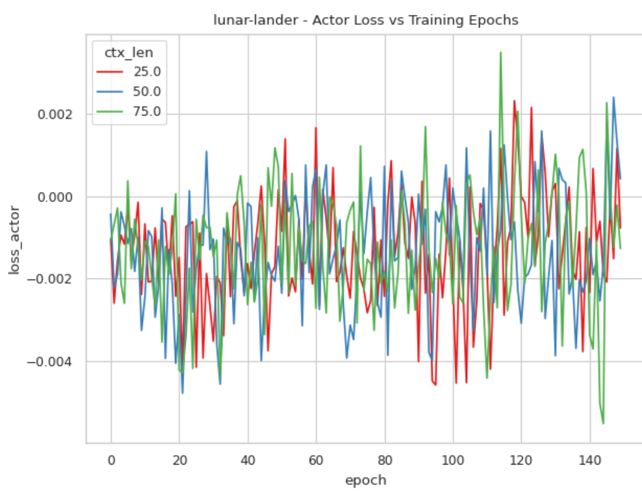

Fig. 9. Lunar Lander - GPT2PPO Context Length - Actor Loss

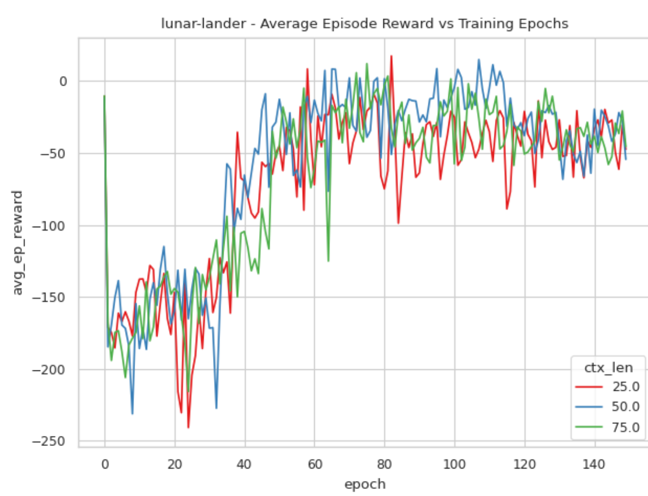

Fig. 10. Lunar Lander - GPT2PPO Context Length - Average Episode Reward

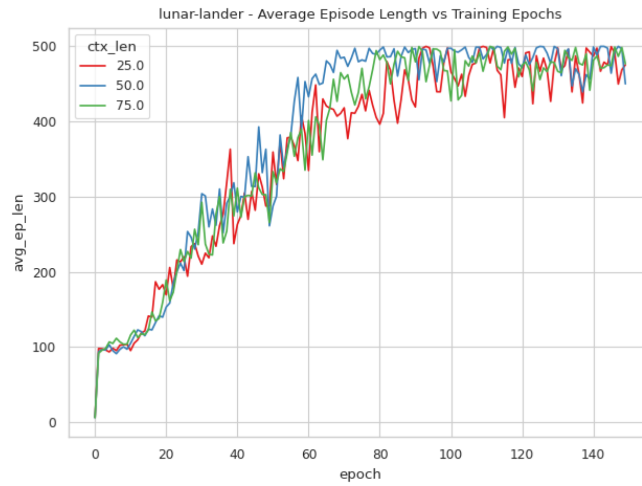

Fig. 11. Lunar Lander - GPT2PPO Context Length - Average Episode Length

Fig. 12. Acrobot - PPO vs GPT2PPO - Critic Loss

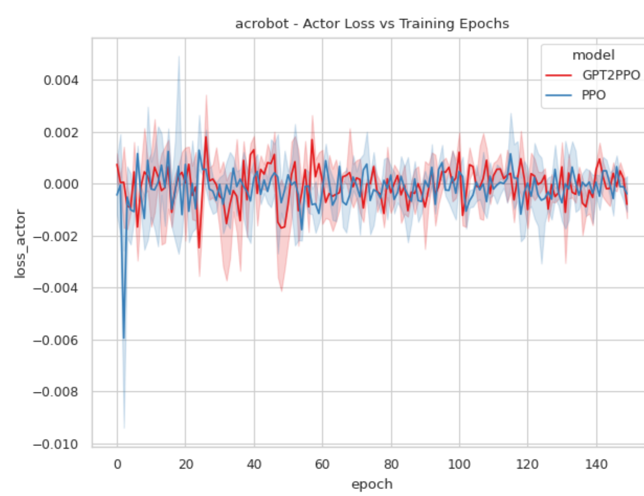

Fig. 13. Acrobot - PPO vs GPT2PPO - Actor Loss

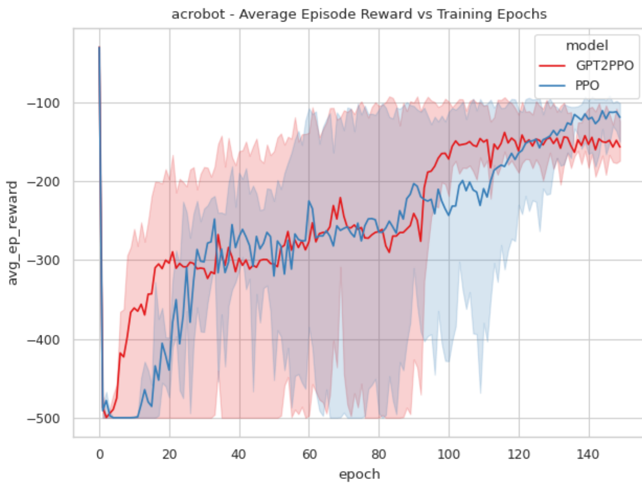

Fig. 14. Acrobot - PPO vs GPT2PPO - Average Episode Reward

Fig. 15. Acrobot - PPO vs GPT2PPO - Average Episode Length

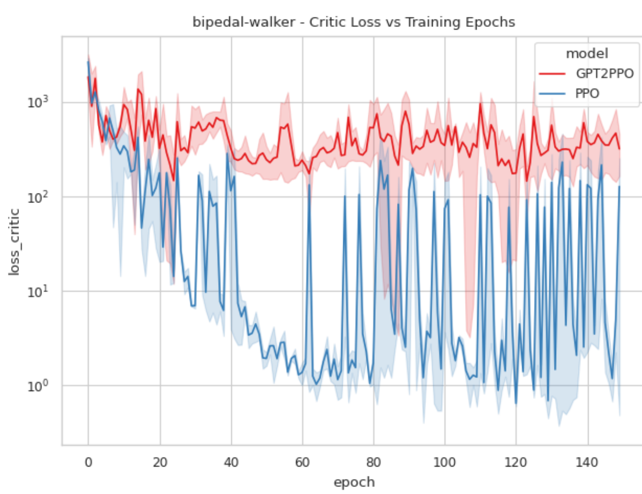

Fig. 16. Bipedal Walker - PPO vs GPT2PPO - Critic Loss

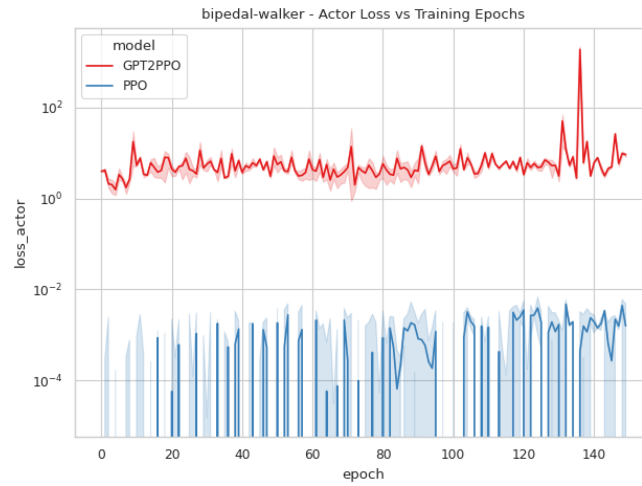

Fig. 17. Bipedal Walker - PPO vs GPT2PPO - Actor Loss

Fig. 18. Bipedal Walker - PPO vs GPT2PPO - Average Episode Reward

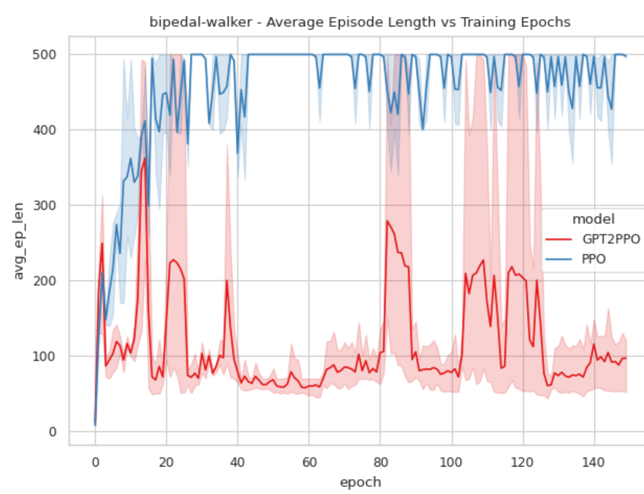

Fig. 19. Bipedal Walker - PPO vs GPT2PPO - Average Episode Length

B. Acrobot

C. Bipedal Walker

VII. Conclusion

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/

[4] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," *Advances in neural information processing systems*, vol. 34, pp. 15 084–15 097, 2021.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[6] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

Appendix

A. Team Member Contributions

Andrei Cozma. text here.
Hunter Price. text here.