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

I. Introduction

In this project, we explore the use of transformers in the context of Reinforcement Learning. Most theoretical works assume that problems follow a Markovian process, which is not always the case. Some problems need the context of previous states and actions to make an informed decision on the next action to take. 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

Reinforcement Learning (RL) is a field that has been dominated by several approaches. One of the most notable is the Advantage Actor Critic (A2C) model [1], which uses two networks for the actor and critic. These models are used to estimate an optimal policy and value function, respectively. The advantage function is calculated on a prediction and is used to compute the loss.

Proximal Policy Optimization (PPO) [2] builds upon A2C and uses a similar architecture. It implements clip loss for the actor loss, which takes the minimum of the ratio of new and old log probabilities, and the clipped ratio times the advantage. PPO is now a widely used algorithm in RL and is often used with models like ChatGPT [3].

Transformers have also been explored in the field of RL. One example is Decision Transformer, which uses an autoregressive language model (GPT2) to take a series of time steps, states, actions, and rewards-to-go and predict the next action [4], [5]. The flexibility of using transformers is made possible by Huggingface Transformers [6], which provides an easy-to-use API for state-of-the-art transformer models. In this work, we combine the PPO algorithm and the Decision Transformer.

III. Background

All of the environments used in this project are from the OpenAI Gym library [7]. The environments are described below.

A. Lunar Lander

The Lunar-Lander environment[1] shown in Figure 1 is a rocket trajectory optimization problem offered by OpenAI Gym. 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 of the environment 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, corresponding to the following actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

The reward structure of the environment is designed to encourage the lander to land on the pad without crashing. If the lander moves away from the landing pad, it receives a negative reward. If the lander crashes, it receives an -100 reward. If it comes to rest on the landing pad, it receives an +100 reward. Each leg with ground contact receives +10 points. Firing the main engine costs -0.3 points each frame, while firing the side engines costs -0.03 points each frame.

The episode ends when the lander crashes, goes outside of the viewport, or comes to a resting position on the landing pad. The lander's initial state is at the top center of the environment, with a random initial force applied to its center.

B. Acrobot

The Acrobot environment[2] shown in Figure 2 is a 2-link pendulum with only the second joint actuated, offered by OpenAI Gym [7]. The goal is to swing the end of the

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

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

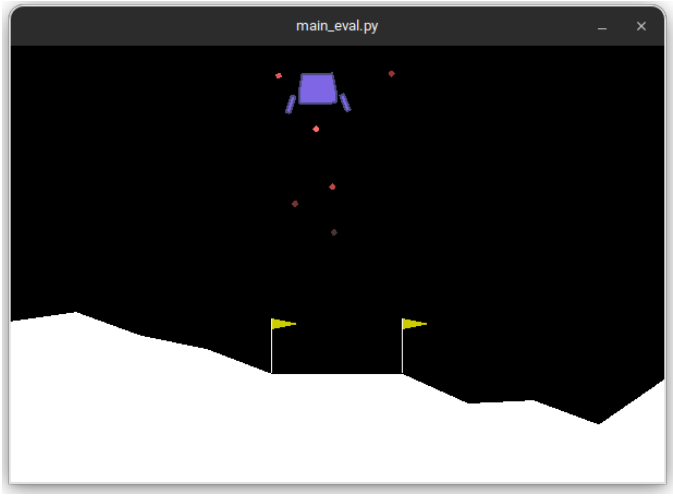

Fig. 1. The Lunar Lander environment.

pendulum up to a given height. The environment is based on the work of Sutton and Barto [8].

The state space of the environment 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, corresponding to the following actions: apply +1, 0, or -1 torque to the actuated joint.

The reward structure of the environment is designed to encourage the agent to reach the goal height. 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 when the agent reaches the goal height or if the episode exceeds the maximum number of timesteps.

C. Bipedal Walker

The Bipedal Walker environment[3] shown in Figure 3 is a challenging control task offered by OpenAI Gym [7]. The goal is to control a two-legged robot to walk as far as possible without falling. This environment has continuous action and state spaces, making it more challenging than the discrete action spaces of the Lunar Lander and Acrobot environments.

The state space of the environment is a 24-dimensional vector containing the position, velocity, and angle of the robot's body and legs. The action space is a continuous 4-dimensional vector, with each dimension representing the torque applied to one of the robot's joints.

The reward structure of the environment is designed to encourage the robot to move forward without falling. The robot receives a reward of +1 for each timestep it is able to remain upright and move forward. If the robot falls, it receives a large negative reward. The episode ends when the robot falls or reaches the maximum number of timesteps.

[3]Bipedal Walker: https://www.gymlibrary.dev/environments/box2d/bipedal_walker

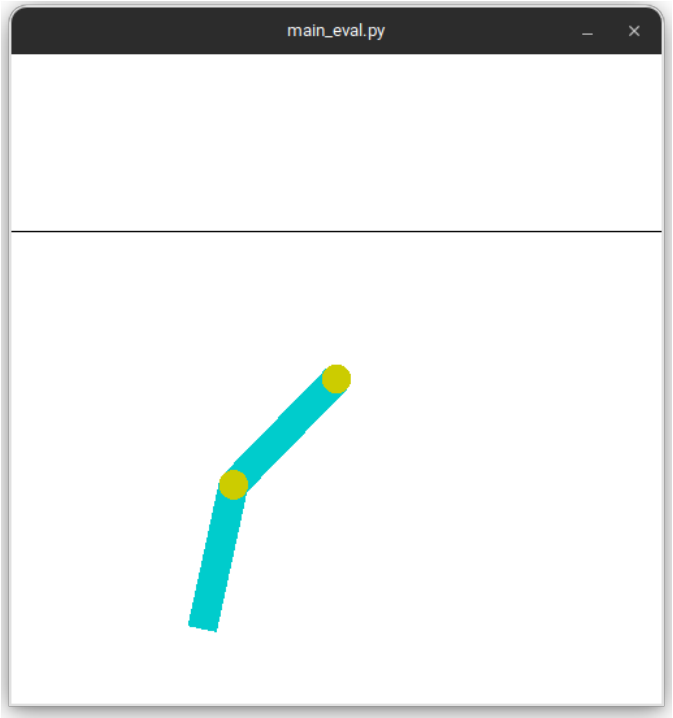

Fig. 2. The Acrobot environment.

Overall, the BipedalWalker environment presents a more challenging control task for our reinforcement learning agents compared to the Lunar Lander and Acrobot environments. This allows us to evaluate the performance of our agents on a wider range of tasks and environments.

Fig. 3. The Bipedal Walker environment.

IV. Methodology

Our agents implement the Proximal Policy Optimization (PPO) algorithm to learn the optimal policy for each environment. PPO is a popular policy gradient method that uses a clipped surrogate objective function to improve the policy.

To train the PPO algorithm, we generate trajectory samples by keeping track of both the states and actions of the environment, as well as the outputs of the network for each sample in the trajectory. We then use these trajectory samples to train the network in batches, where each batch of data is used to run several training steps. The number of training steps of optimization that we run for each batch of data is defined as a configurable hyperparameter.

The network architecture used by our agents is composed of two main components: the actor and the critic. The actor part of the network generates the policy distribution for the state-action pairs, while the critic uses the Generative Pre-trained Transformer 2 (GPT2) model as an encoder to generate predictions for the state-action pairs and generate the value function.

To prevent the actor network from overfitting, we use a simpler feed-forward network that only takes in the last state of the trajectory as input. This allows the agent to continue exploring the environment and avoids overfitting, while the critic network uses the full sequence data to make more informed predictions about the actor's behavior over time and under specific conditions.

A. Reinforcement Learning Methods

In this project, we use the Proximal Policy Optimization (PPO) algorithm [2] to find the optimal policies for agents to follow in unknown environments. PPO is a state-of-the-art reinforcement learning algorithm that is both effective and efficient. It works by using a policy to determine the actions that an agent should take in a given state, and then using a value function to evaluate the potential outcomes of those actions. The algorithm then uses this information to adjust the policy in a way that maximizes the expected reward for the agent.

We use the PPO-clip variant of the algorithm, which uses a "clipping" mechanism to prevent the algorithm from making too large of a change to the policy in a single iteration. This helps to stabilize the learning process and prevent the algorithm from becoming too "jumpy" or erratic.

In addition to the PPO algorithm, we also use the transformer architecture [9] as part of our network. The transformer is a type of neural network architecture that is commonly used in natural language processing tasks, such as machine translation and text summarization. It is able to process large amounts of data very efficiently, and uses self-attention mechanisms to "focus" on different parts of the input data at different times.

In the context of our PPO algorithm, the transformer is used to generate predictions for the state-action pairs. The transformer takes in the sequence of state-action pairs as input, and then uses self-attention to capture long-term dependencies in the data. The output of the transformer is then used by the critic head of the network to generate the value function, which is used to evaluate the potential outcomes of the actions taken by the agent.

Overall, our approach to reinforcement learning combines the powerful PPO algorithm with the efficient transformer architecture to learn the optimal policies for agents in unknown environments. This allows us to take advantage of the strengths of both methods.

V. Code Design

A. Utilized Libraries

One of the main libraries we use in this work is PyTorch [10]. PyTorch is a popular deep learning library that is well-documented and has a large community. We chose to use PyTorch because it is easy to use and has a wide range of powerful tools for building and training deep learning models.

In addition to PyTorch, we also use the PyTorch-Lightning library [11]. PyTorch-Lightning is a high-level library that makes it easy to train and test deep learning models. It provides a module named Lightning-Bolts that contains many pre-built models and utilities, which we use to help maintain the structure of our models.

To implement the GPT2 model, we use the HuggingFace Transformers library [6]. This library provides an intuitive interface for working with the GPT2 model, allowing us to easily incorporate it into our reinforcement learning algorithms.

We use the OpenAI Gym library [7] to provide the environments for our models to train and test on. The Gym library offers a wide range of environments that are well-suited to reinforcement learning tasks.

Finally, we use Weights & Biases (WandB) [12] to log the results of our experiments. WandB allows us to track and visualize the performance of our models over time, helping us to understand how they are performing and identify areas for improvement.

B. File Structure

The main directory of this project 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. Within this directory, there are many files that are important to the project. The primary files that are used are *Model.py*, *GPT2PPO.py*, *CommonGPT2.py*, *CommonBase.py*, and *GPT2.py*. These files will be discussed in more detail in the following sections.

The *rllib/examples* directory contains A2C and PPO baseline implementations that were taken from the Lightning-Bolts library and copied into our project.

Finally, the *archive* directory holds all the iterations of old code that we have tried and discarded. This directory provides a history of our development process and helps us to keep track of what we have tried and why we made certain decisions.

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 interval 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

In this work, we use the *Proximal Policy Optimization (PPO)* algorithm to learn the optimal policies for agents in unknown environments. The baseline PPO model we use to compare against our results is in the *rllib/examples/PPOExample.py* file. It contains code 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 built upon this class.

Our implementation is in the *rllib/GPT2PPO.py* file. This file contains the *GPT2PPO* class which implements the same interface as the reference PPO. In addition to the MLP, ActorContinuous, and ActorCategorical models used in the baseline PPO implementation from the Lighting-Bolts library, we use two additional models implemented in the *rllib/CommonBase.py* and *rllib/CommonGPT2.py* files.

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 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 [5]. In that work, the authors took the HuggingFace GPT2 model and removed the logic that implment positional embeddings [4], [6]. Our CommonGPT2 shares large similarities with the Decision Transformer implementation [5]; 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 and 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 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 hyperparameters 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
Number of Layers*	2
Number of Heads*	8
Dropout*	0.5
Number of Positions*	1024

*Only used for GPT2PPO.

A. Lunar Lander

Lunar Lander is a simple environment that we assume will be easy for both models to learn. We do not believe recurrence or attention will offer any superior advantage to base PPO in this environment. As described above, we trained both PPO and GPT2PPO for 150 epochs three times and averaged the results. Through our experiments, we found the models to be comparable in performance.

Figure 4 shows the critic loss for both models. It can be seen that the GPT2PPO model critic learns slower than the Baseline PPO model. This is likely due to the fact that the GPT2PPO model has far more weights to train, and the input to the model is much larger. However, the GPT2PPO critic does learn and eventually converges to a similar critic loss as the baseline PPO model.

Figure 5 shows the actor loss for both models. It can be seen that the loss for both models is very similar. This is likely because the actor in both models is very similar. The only difference is that the actor has two additional layers.

Figure 6 shows the average episode reward for both models. On average, the GPT2PPO model gains increases in average episode rewards at a similar pace as the Baseline PPO. However, the GPT2PPO model does not reach the same average reward as the Baseline PPO model; instead, it converges upon a slightly lower episode reward.

Figure 7 shows the average episode length for both models. On average, the GPT2PPO model gains increases in average episode length slower than the Baseline PPO. Additionally, the GPT2PPO model does not reach the same average episode length as the Baseline PPO model. The GPT2PPO model seems to learn up to a certain point but fails to converge upon an optimal policy. This could be because we needed to train the model for more epochs Overall, the GPT2PPO model performs similarly to the baseline PPO model. However, the GPT2PPO model converges to slightly worse values as compared to the Baseline PPO model.

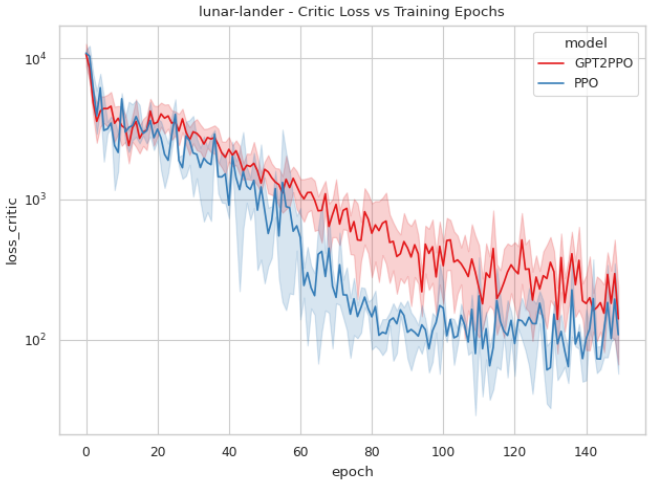

Fig. 4. Lunar Lander - PPO vs GPT2PPO - Critic Loss

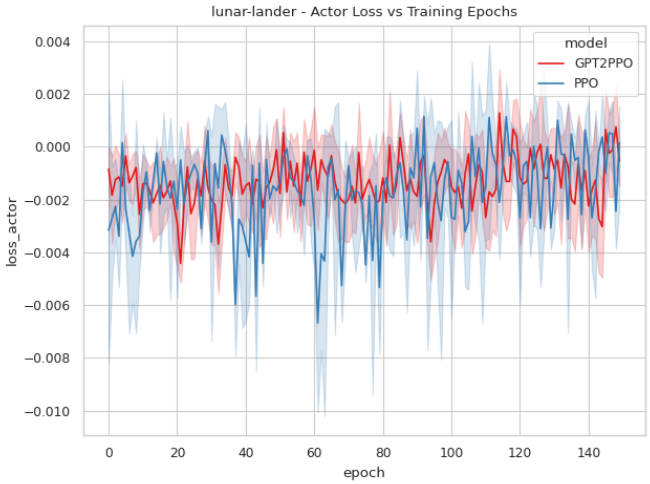

Fig. 5. Lunar Lander - PPO vs GPT2PPO - Actor Loss

We also experimented with different context lengths for the GPT2PPO model for the Lunar Lander environment. We kept the same hyperparameters shown in Table I, but altered the context length to be 25, 50, and 75. Figures 8, 9, 10, and 11 show these results. We can see that all context lengths perform similarly. This is likely because of the structure of the environment.

B. Acrobot

The Acrobot environment is slightly more complex than the Lunar Lander environment. We stipulate that some sense of

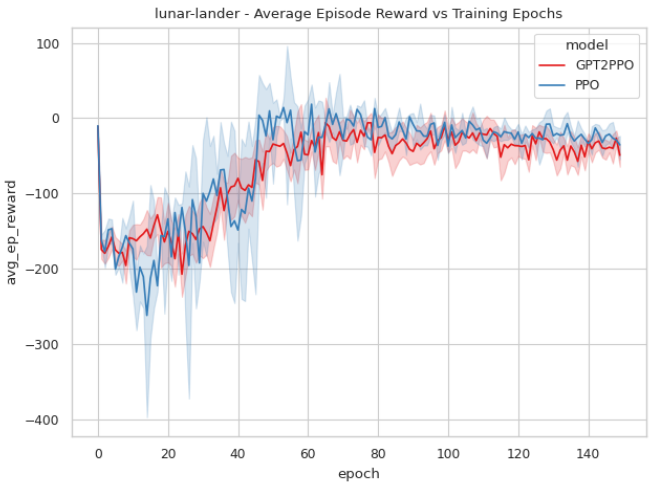

Fig. 6. Lunar Lander - PPO vs GPT2PPO - Average Episode Reward

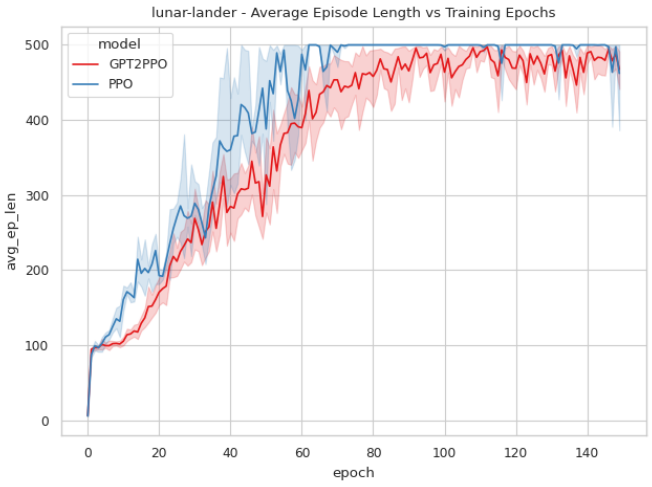

Fig. 7. Lunar Lander - PPO vs GPT2PPO - Average Episode Length

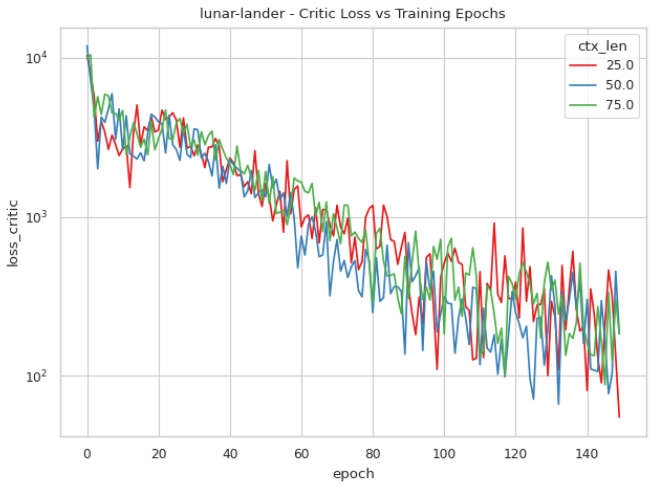

Fig. 8. Lunar Lander - GPT2PPO Context Length - Critic Loss

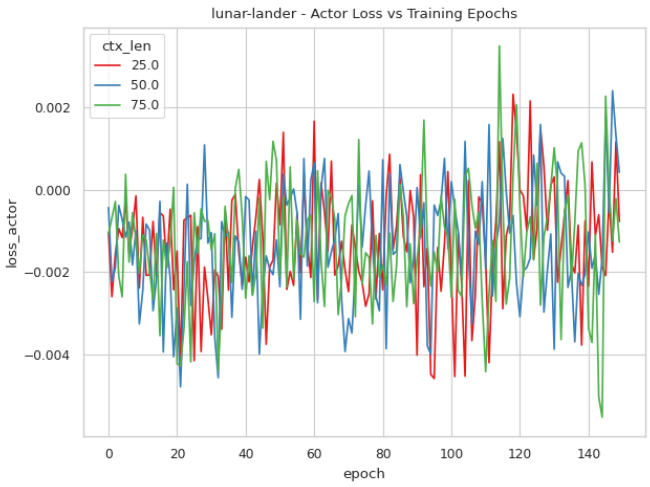

Fig. 9. Lunar Lander - GPT2PPO Context Length - Actor Loss

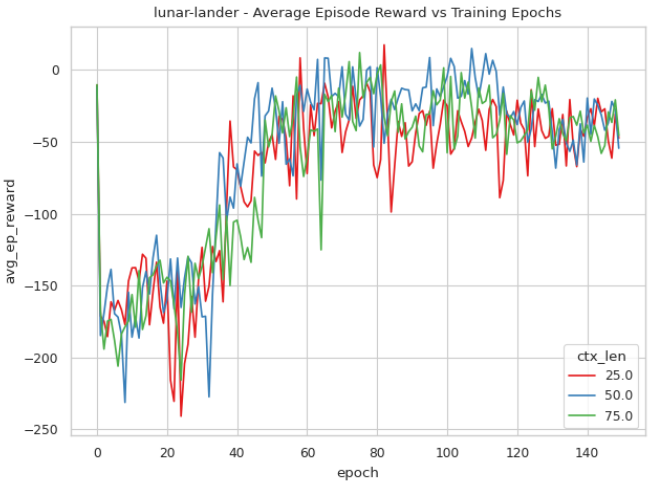

Fig. 10. Lunar Lander - GPT2PPO Context Length - Average Episode Reward

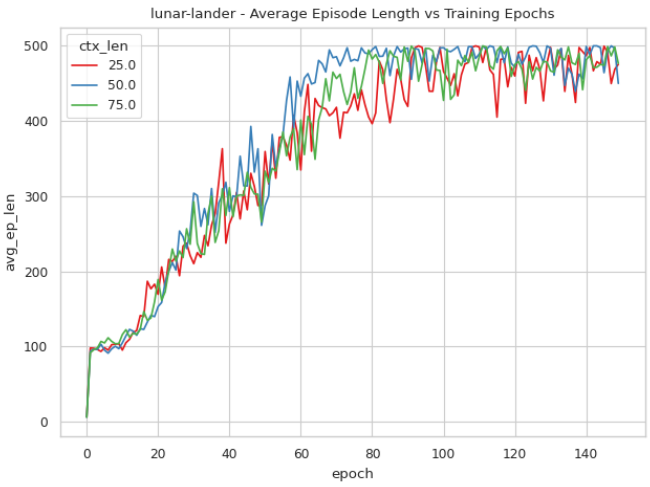

Fig. 11. Lunar Lander - GPT2PPO Context Length - Average Episode Length

recurrence or attention would be beneficial as the agent should likely take into account its previous states and actions when making a decision to conserve its momentum. Through our experiments, we find that the GPT2PPO model tends to learn faster than the Baseline PPO model but tends to converge to poorer values. Additionally, we found that our model is hugely dependent on starting conditions of the GPT2 model.

Figure 12 shows the critic loss for both models. We can see that the GPT2PPO model learns faster on average than the Baseline PPO model, but after 90 epochs, it will diverge to a worse value. Figure 13 shows the actor loss for both models. The average results are similar between each model. Our model seems to have a higher variance as compared to the Baseline PPO model.

Figure 14 shows the average episode reward for both models. The shaded regions show the minimum and maximum values for each model across the three runs. Here we can see that the results for our model vary. Our model's minimum average episode reward stays equal to the maximum episode length of 500. It does not seem to improve until the 90th epoch, where it quickly improves toward the results of other runs. On the other side of the spectrum, the maximum average episode reward learns much quicker than the Baseline PPO model. Moreover, our model, on average, will learn at a similar pace to the Baseline PPO model but will converge to a worse value.

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

C. Bipedal Walker

The Bipedal Walker represents the most complex environment we have trained our model on. This is because the state space is large, and the action space is continuous. We hypothesize that the configuration of our model used to compare against the Baseline PPO is insufficient to learn the environment. To learn better, we would need to increase the size of the GPT2 model and the number of epochs.

Figure 15 shows the critic loss for both models. We can see that the converged GPT2PPO critic loss is higher than the Baseline PPOs by a factor of 10. This is consistent with Figure

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

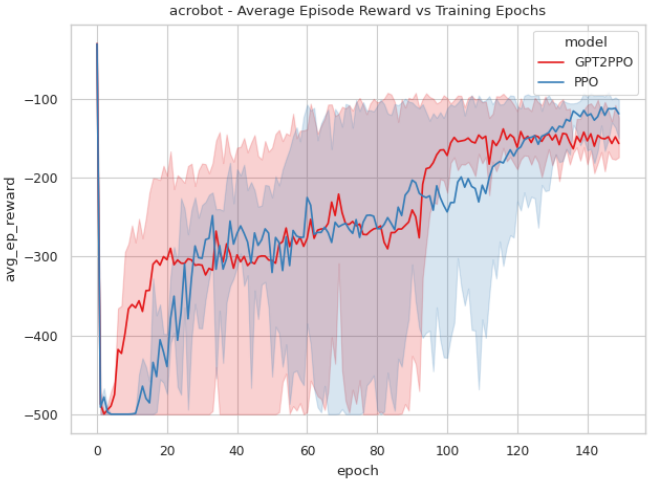

Fig. 14. Acrobot - PPO vs GPT2PPO - Average Episode Reward

16, where the GPT2PPO actor loss is a factor of 1000 higher than the converged Baseline PPO actor loss. Figure 17 shows the average episode reward for both models. Similar to the Acrobot environment, the results of our model significantly vary. Our model quickly switches between good and bad policies, whereas the Baseline PPO model converges much more smoothly. Overall, the Baseline PPO model significantly outperforms our model in this environment.

VII. Conclusion

In this work, we propose an addition to the Proximal Policy Optimization (PPO) algorithm used in reinforcement learning, named GPT2PPO. This addition includes adding two layers to the actor network and an autoregressive model (GPT2) to the base of the critic network. We show that in simple environments, our model performs similarly to the basic PPO implementation but ultimately converges to a slightly poorer outcome. We also show that our model can learn in more

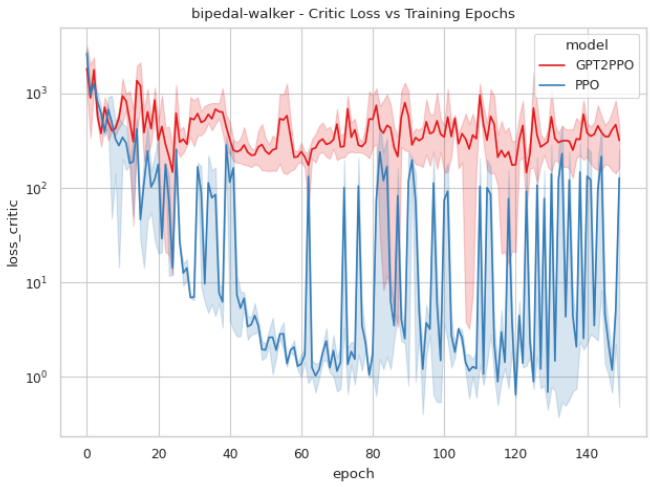

Fig. 15. Bipedal Walker - PPO vs GPT2PPO - Critic Loss

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

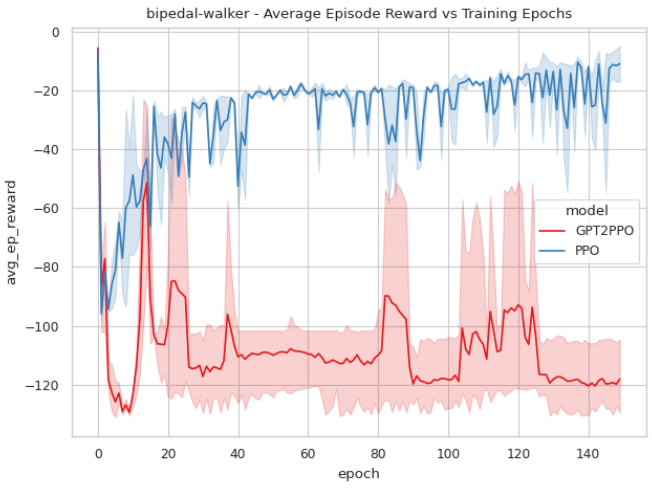

Fig. 17. Bipedal Walker - PPO vs GPT2PPO - Average Episode Reward

complex environments, but additional model finetuning and training are needed to be effective in those environments.

When using a shared network layer for both the actor and critic, we find that the critic loss converges faster than the Baseline PPO, but the actor loss diverges after a small number of epochs. If given more time, we would like to validate the correctness of the model's inputs and outputs, finetune our model, and experiment on more environments to determine the viability of our approach.

Overall, we have learned about the inner workings of PPO and that applying attention in the form of transformers can be effective when done correctly. However, more work is needed to optimize our network architecture for complex environments.

References

[1] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[3] OpenAI, "Chatgpt: Optimizing language models for dialogue," Dec 2022. [Online]. Available: https://openai.com/blog/chatgpt/

[4] 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.

[5] 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.

[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

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

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

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[11] W. Falcon et al., "Pytorch lightning," *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, vol. 3, 2019.

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

Appendix

A. Team Member Contributions

Andrei Cozma contributed to the overall project with a working implementation of simple transformer encoder to be

used as a common network for a baseline A2C implementation. Since A2C and PPO are similar in that they utilize actor and critic heads to predict the optimal policy and value function, respectively, he integrated it into the PPO implementation from the Lightning-Bolts library, which already provided functionality for creating batches of trajectories into an experience dataset. This provided us with a starting point for using one of the transformer models from HuggingFace as a swap-in replacement to the initial transformer model that was used for implementation testing. He also worked with Hunter to set up the rest of the training and evaluation framework in PyTorch and the pipeline for data collection and visualization of the results. He ran many variations of the initial PPO implementation with the simple transformer to better understand the sequence model's behaviors under different conditions. This was useful in determining the best ways we can integrate the final GPT2 model into the PPO algorithm. Finally, he worked with Hunter to write the final report.

Hunter Price contributed a working modification of the Decision Transformer that used only the time steps, states, and actions. He utilized Andrei's modification of PPO that stored a buffer of the past timesteps, states, and actions to feed to the CommonGPT2 model. He then modified this code to use consistent matrix shapes. He also added the ability for GPT2PPO to support continuous action spaces. He worked with Andrei to set up the training and testing framework in PyTorch. He found the baseline A2C and PPO implementation in the Lighting-Bolt library. He worked with Andrei to run many experiments used and shown in the report. He wrote code to visualize the results outside of WandB and save them locally. Worked with Andrei to write the report. Additionally, He ran many experiments with different versions of PPO, such as using VGG16 as a base for both the actor and critic when working with environments that gave image data as the state and a model that used a shared encoder with both the actor and critic networks.