Reinforcement Learning:

Training a Deep RL Model to Play Mario Bros

Zack Strathe
Kansas State University – CIS 730

Introduction

In the field of reinforcement learning (RL), models can be trained that are capable of learning to complete tasks with only a reward function to guide positive behavior. In real-world applications, there seem to be countless use cases where reinforcement learning could be used to automate tasks—for example, with self-driving cars that are capable of learning to drive more safely, or software-agents attempting to maximize efficiency of an electrical grid. The concept of artificial intelligence that can continually learn to become better is very promising, but there exist drawbacks and limitations. And to explore the concept of reinforcement-learning, it is helpful to simplify an implementation with a toy example, such as a game-playing agent. Therefore, the focus of this project will be to train a RL agent to play a simple retro game from the Nintendo Entertainment System: Mario Bros.

The overall goal of this project is to utilize a deep-reinforcement policy to train a game-playing agent to successfully play the classic NES game Mario Bros and complete the first level. This game is very simple, with only three controls available for the agent (move left, move right, jump). The action space in this game is discrete and the environment is fully-observable. Further details about the game environment are presented in Appendix I. My research objective is to research multiple reinforcement-learning (RL) algorithms and implement the best candidate for training a game-playing agent. After selecting an algorithm to utilize, in order to develop the best possible model, I will experiment with modifying the reward function, modifying the action space, modifying the observation space, and modifying algorithm hyperparameters.

Background and Related Work

The implementation for controlling a reinforcement-learning agent in Mario Bros already exists as a pre-built environment in OpenAI Gym Retro (OpenAI n.d.). A YouTube video (World of Longplays 2012) shows game-play by a human player, from which I will use the score as a baseline for how many points the RL agent should score to complete the first level. In the video, the player completes the first level with a score of 2,430 points. Therefore, if a trained RL agent achieves 2,430 points or greater, it will be assumed that it can complete the first level.

To become more familiar with some basic RL algorithms I researched the documentation from OpenAI SpinningUp (OpenAI n.d.). My findings were that either Proximal Policy Optimization (PPO) or Trust Region Policy Automation (TRPO) algorithms would be an ideal choice for a game-playing agent, as they are currently leading-edge when handling discrete action spaces. And further, the paper "Proximal Policy Optimization Algorithms" (Schulman et. Al. 2017), indicates that both PPO and TRPO achieve similar results, but with PPO having a more simplified implementation and sampling complexity, so I believe that PPO will be the ideal choice for this implementation of reinforcement learning. In addition, I researched into examples of others' work that implement game-playing agents utilizing OpenAI Gym Retro; a webpage (Poliquin 2019) provides some in-depth information about doing this. The website's author linked to a number of videos showing the results of game-playing agents trained using OpenAI Gym Retro, utilizing the PPO2 algorithm from OpenAI Baselines. The website also shows how to graph metrics such as the training reward score over time.

Methodology

To train a RL agent to play Mario Bros, I utilized the OpenAI Gym Retro library for Python, which contains a fairly simple interface for conducting reinforcement-learning on a supported game environment. Located in the 'MarioBros-Nes' files within Gym Retro directory is a 'scenario.json' file which defines the reward function for the game. The 'scenario' file defines memory locations of variables that are to be tracked and used as an input to the reward function. By default, the Mario Bros scenario file defines the reward function as only the game score. The obvious benefit of this reward function is that it should work in most cases, as the goal of most games is generally to score more points. However, this reward function does have an obvious drawback: it can lead the agent to behavior that may not contribute to the goal. In many games, actions can reward the player with points, though the action may not contribute toward progression in completing the game. As noted in gameplay notes in Appendix I, Mario Bros especially features gameplay elements that may lead a RL agent to engage in repetitive behavior that does not contribute to completing the level.

In addition to utilizing the Open AI Retro library, for choosing a state-of-the-art algorithm, I decided to utilize the Open AI Baselines library, which supplies of number of implementations of leading-edge algorithms. And in particular, Baselines contains the PPO algorithm, which my research indicates to be a leading algorithm when it comes to performance as well as simplified implementation. And although there exist many alternative libraries that also provide PPO algorithms, I determined through informal testing that the Baselines library provided the fastest training speed and best results with the "PPO2" algorithm, which uses TensorFlow 1.x for execution. One downside to using the Baselines library is that it is not coded very modular and can be difficult to utilize without using the command-line interface that is provided with the library, which makes building a custom model from scratch to be a difficult approach. However, I found that utilizing the command line interface worked well, and any code customizations could be simply done by downloading the library and modifying those files as-needed.

For monitoring the training process with real-time graphs of the mean reward score, utilized TensorBoard. Further, TensorBoard provides a simple method of saving training logs as ".csv" files for further analysis or development of visualizations. For the development of visualizations, first made manual modifications to the logs (to rescale the number of steps logged, as some algorithm parameters will provide different intervals for the same total number of steps). Then, I imported the logs into Tableau to build comparative graphs.

One very important consideration with training a RL agent is the availability of computing resources. Because it is commonplace for RL training to take millions of steps, training time without sufficient computing resources can be a very long process. In my initial testing with a laptop (equipped with an AMD Ryzen 5 3500U CPU and no dedicated GPU), the training time to reach 1 million steps was measured in days. Therefore, it was quickly apparent that I needed to utilize cloud computing, and preferably with the option of utilizing a GPU since the Baselines "PPO2" algorithm enables GPU calculations for a significant speedup (OpenAI 2017). Therefore, I selected to utilize the Google Cloud platform, which offers dedicated VM instances of Colab notebooks, with the option of adding a dedicated GPU. After informal testing, I opted to utilize a VM with an 8-core vCPU and a dedicated Nvidia Tesla T4 GPU. With this VM, I was able to train a model to 5 million steps in approximately 2 hours.

Experiment Design

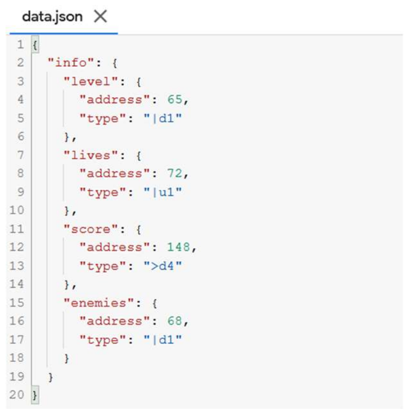
To evaluate whether a trained model met the goal of completing the first level, I used a score benchmark because it is easy to extract from the game environment. So, if during training a model, the mean training reward score met or exceeded the baseline score of 2,430 points, determined from a video of a human player (World of Longplays 2012), the assumption was that the model had met the goal. Further, if a model did achieve a training mean reward score above 2,430 points, I saved the model and recorded a video to verify whether it actually could complete the level.

To attempt to improve upon a model trained with default parameters from the Gym Retro and Baselines libraries, I attempted improvements in four categories: modifying the reward function, modifying the observation space, modifying the action space, and last, modifying the PPO algorithm hyperparameters. Each training run consisted of training a model to 5 million timesteps, which based on informal testing, seemed to be an approximate point where models will either begin to converge to a successful strategy or not.

Results and Conclusions

Modifying the Reward Function

To modify the reward function, I updated the default "scenario.json" file for the Mario Bros game in the Gym files to calculate the reward based on killing enemies, providing a reward of 1 for each time the number of enemies is reduced. To do this, I utilized the OpenAI Gym Integration Tool to find the memory location where the variable for the number of enemies is stored. With this variable located, I updated the "data.json" file to include the new variable, and then updated the scenario file to give a reward of 1

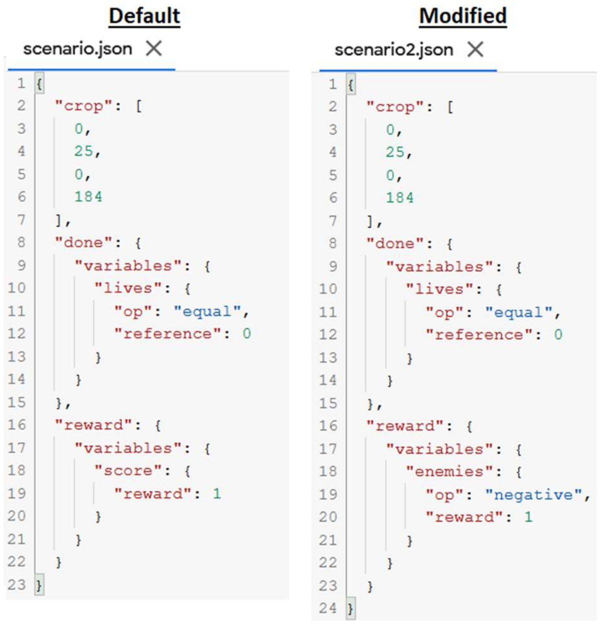


data.json

```
{
  "info": {
    "level": {
      "address": 65,
      "type": "|d1"
    },
    "lives": {
      "address": 72,
      "type": "|u1"
    },
    "score": {
      "address": 148,
      "type": ">d4"
    },
    "enemies": {
      "address": 68,
      "type": "|d1"
    }
  }
}
```

Figure 1: Modified "data.json" file to include the "enemies" variable and memory address

each time the number of enemies variable decreases. Screenshots of the modified data and scenario files are shown in Figures 1 and 2.


Default

scenario.json

```
{
  "crop": [
    0,
    25,
    0,
    184
  ],
  "done": {
    "variables": {
      "lives": {
        "op": "equal",
        "reference": 0
      }
    }
  },
  "reward": {
    "variables": {
      "score": {
        "reward": 1
      }
    }
  }
}
```

Modified

scenario2.json

```
{
  "crop": [
    0,
    25,
    0,
    184
  ],
  "done": {
    "variables": {
      "lives": {
        "op": "equal",
        "reference": 0
      }
    }
  },
  "reward": {
    "variables": {
      "enemies": {
        "op": "negative",
        "reward": 1
      }
    }
  }
}
```

Figure 2: Showing the default versus modified scenario files, with the modified reward function calculated to provide of reward of 1 every time the number of enemies is reduced

The PPO model was trained to 5 million steps using the modified reward function, and the results are shown in Figure 3. By the shape of the graph, it is apparent that the agent was unable to learn any beneficial behavior since there is no trend shown in the mean reward value increasing over time. This is likely because rewards are too sparse with the modified reward function, which does not allow the model to generalize state and action pairs, since most actions are accompanied with zero change to the reward function. Therefore, for subsequent testing of other model settings, I reverted to the default reward function (which uses the game score only).

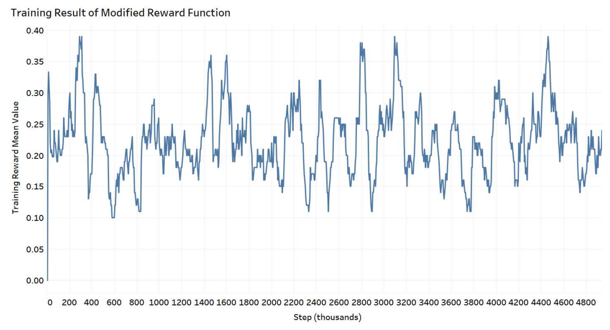


Figure 3: Training results to 5 million steps with the modified reward function

Modifying the Observation Space

While looking into modifying the reward function, I noticed that Gym Retro allows for cropping of the observation space with the “scenario.json” file (with cropping shown in the screenshots in Figure 2). Because using a convolutional neural network (CNN) utilizes images of the game environment (IBM 2020), it seems sensible that a model would better be able to generalize states and actions if useless information/noise is removed from the inputs. For example, the score at the top of the screen will update throughout the training process and would be sampled by the CNN. At best, I believe the model training process may be only slightly slowed down by the extra erroneous input data; and at worst, the score info could cause the model to make incorrect state-action pair associations. Therefore, I experimented with training the model and cropping out the score info (as well as some unchanging space at the bottom of the screen). At this stage I also compared results from using the “CnnPolicy” and “ImpalaCnnPolicy” that are provided with the Baselines library. The results are shown in Figure 4.

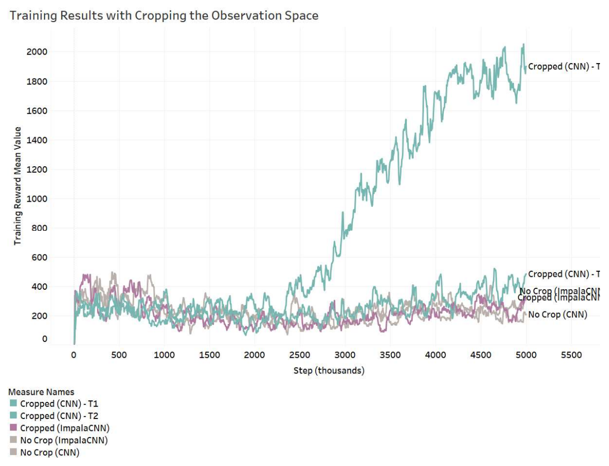


Figure 4: Training results to 5 million steps, showing comparison of cropping the environment, as well as comparison between the CNN and ImpalaCNN networks provided with the OpenAI Baselines library

While the first training run with the cropped environment and CNN policy seemed to be extremely promising (shown as “Cropped (CNN) – T1” in Figure 4), the subsequent training run (noted as “T2” in Figure 4) using the same parameters and seed displayed results much closer to the average. While this does show that stochasticity is certainly a factor in these training experiments (despite using an unchanging seed number), the graph does seem to indicate that the cropped observation space generally outperforms the uncropped observation space, and similarly, that the “CnnPolicy” outperforms the “ImpalaCnnPolicy.” While a larger sampling size would significantly help with making these determinations, the significant computing cost and time of each training run severely limits feasibility of additional sampling. Accordingly, for the following model-training experiments, each will utilize the cropped observation space and "CnnPolicy."

In modifying the observation space, I also looked at results from OpenAI's Retro Contest (OpenAI 2018), where teams competed to train agents to play the game Sonic the Hedgehog. Specifically, I looked to what the winning team—Team Dharmaraja—did (Yang 2018). The default Baselines PPO2 algorithm samples the environment as an 84x84 pixel greyscale image, however Team Dharmaraja modified that to utilize an RGB image. Therefore, I also attempted to train my model with an RGB image input, which I accomplished by modifying one of the default "wrappers" in the Baselines library—specifically the "WarpFrame" wrapper which changes how the environment image is sampled. In addition, I also tested modifying parameters of the WarpFrame wrapper to double the size of images, using 168x168 pixel images rather than the default of 84x84 pixels. To implement these modifications with the Baselines library, I modified some of the Baselines library files. When using a Gym Retro environment, the Gym environment is built from the Baselines/common/retro_wrappers.py file. Within this file, I modified the call within the "wrap_deepmind_retro" function to build the environment using the WarpFrame wrapper, specifying optional parameter values for "width", "height", and "greyscale." The results of training the models with these settings to 5 million timesteps are shown in Figure 5.

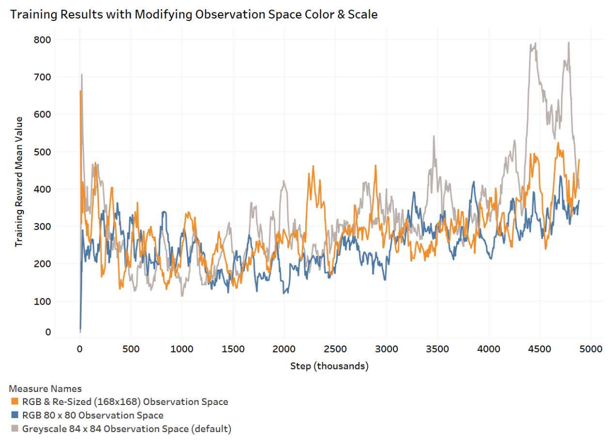


Figure 5: Training results to 5 million steps, showing comparison of training models with an RGB observation space, RGB observation space re-sized to 168x168 pixels, and the default observation space (greyscale 84x84 pixels)

From observing the graph in Figure 5, it's easy to determine that there seems to be no benefit from both using a RGB-colored observation space and re-sizing the observation space. And though I don't have metrics for it, the training speed with both of these modifications were significantly slower than using the default greyscale 84x84 image of the game environment. Therefore, no further tests will utilize these settings, and the only observation space modification for subsequent tests will be a cropped game image.

Modifying the Action Space

To modify the action space, I again looked at what successful teams did in the OpenAI Retro Contest. I observed in the Baselines code library that, while the default Retro Gym environment is built defining the action space using a "retro.actions.Discrete" class, the successful teams utilized a Gym wrapper class called "SonicDiscretizer" which specifically defined the buttons for the game that could be utilized by the model. Therefore, for this project, I copied that class and called it "MarioDiscretizer," and provided a mapping of the three possible actions in the game ("Left", "Right", and "Jump"). The training results from utilizing the modified action space are displayed in Figure 6.

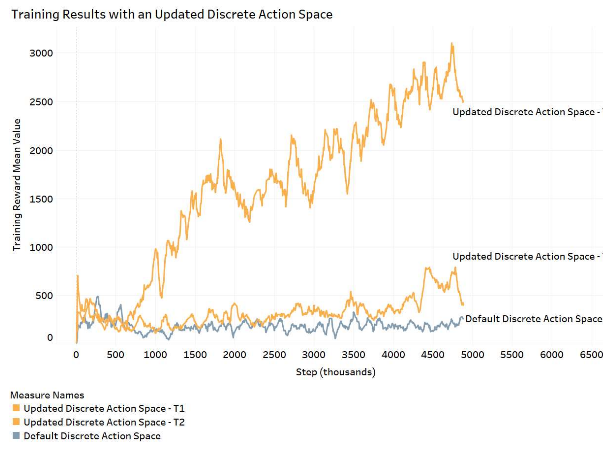


Figure 6: Training results to 5 million steps, showing comparison of using "MarioDiscretizer" wrapper versus the default gym.actions.Discrete action space

While the sample size is very small, it does seem that the updated discrete action space from using a wrapper (that was also utilized by many successful OpenAI Retro Contest teams), provided better training results than the default "gym.actions.Discrete" action space. Most notably, the first training attempt with the updated action space achieved a mean training score of approximately 2,508, indicating that this model may have achieved the goal of completing the first level. The second training attempt ("T2") with the updated discrete action space and unchanged parameters and seed value yielded a much lower score. Again, similar to testing with cropping the observation space, these results indicate that stochasticity plays a factor in these results (since all settings were exactly the same with both the "T1" and "T2" training runs). So, while it would be ideal to gather many more samples to account for randomness, that unfortunately is not feasible due to the time and expense of computing these models. Nevertheless, the updated discrete action space, using the "MarioDiscretizer" wrapper, will be used for subsequent testing of PPO hyperparameters.

Modifying PPO Algorithm Hyperparameters

For tuning hyperparameters, I again looked to what successful teams in the OpenAI Retro Contest did. Specifically, I compared the Baselines PPO2 algorithm hyperparameter defaults with what Team Dharmaraja used (Yang 2018). The table of comparisons is shown in Figure 7. In particular, Team Dharmaraja used different hyperparameter values for nsteps (8,192 versus the default of 2,048), nminibatches (8 versus the default of 4), ent_coef (0.001 versus the default of 0), and lr (0.00002 versus the default of 0.0003). It's also worth noting that Team Dharmaraja appears to have trained their model to 10 billion timesteps, which is infeasible for the scope of this project, and therefore may affect whether some of these hyperparameters work well on the models for this project.

	Baselines PPO2 Default	Team Dhajarama PPO Hyperparameters (bolded if different from default)
policy	n/a	CnnPolicy
nsteps	2048	**8192**
nminibatches	4	**8**
lam	0.95	0.95
gamma	0.99	0.99
noptepochs	4	4
ent_coef	0	**0.001**
lr	3.00E-04	**2.00E-05**
cliprange	0.2	0.2
total_timesteps	n/a	1.00E+10

Figure 7: Baselines PPO2 default hyperparameters versus those used by Team Dharmaraja (Retro Contest winners)

The table of hyperparameters that were tested are shown in Figure 8. The V1 hyperparameters are all copied exactly from what Team Dharmaraja used, except for training to 5 million timesteps instead of 10 billion. For subsequent versions of hyperparameters, only one setting was changed each time (shown in bold in Figure 8). For hyperparameters V2, the learning rate parameter was updated to the Baselines default of 0.0003, because the learning rate used in V1 hyperparameters may be accounting for a significantly longer training time than is possible here. And for hyperparameters V3, the "nsteps" parameter, which is the number of steps in the environment between policy updates, was updated to the Baselines default of 2,048. The graphed results of the models using those hyperparameters are displayed in Figure 9.

While the models trained with the V1 and V2 versions of hyperparameters did not show much divergence from

	V1	*V2*	*V3*
policy	CnnPolicy	CnnPolicy	CnnPolicy
nsteps	8192	8192	**2048**
nminibatches	8	8	8
lam	0.95	0.95	0.95
gamma	0.99	0.99	0.99
noptepochs	4	4	4
ent_coef	0.001	0.001	0.001
lr	2.00E-05	**3.00E-04**	**3.00E-04**
cliprange	0.2	0.2	0.2
total_timesteps	5.00E+06	5.00E+06	5.00E+06

Figure 8: hyperparameter settings used for V1, V2, and V3 models.

average mean scores from prior testing, the V3 version of hyperparameters appears to consistently generate a model that converges on a successful strategy by the end of the 5 million step training. In particular, the first test with V3 hyperparameters was very successful, ending with a mean reward score of 2,742 points. Subsequent tests with the V3 hyperparameters were not nearly as successful as the first, though in general they appear to be an improvement over the other model settings.

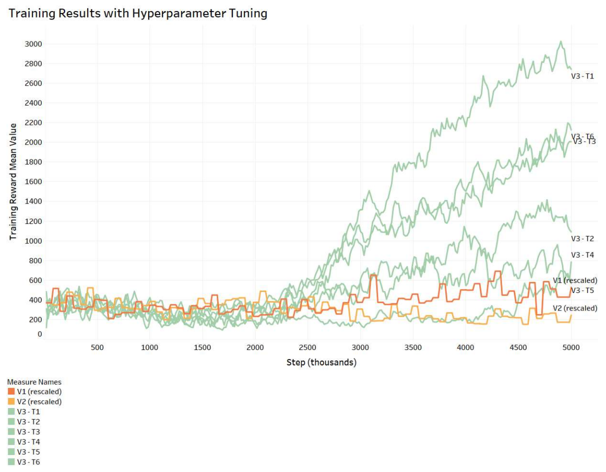


Figure 9: Training results to 5 million steps, showing models using hyperparameter versions V1, V2, and V3.

Conclusions

To determine whether any of the trained models were actually capable of completing the first level, I utilized the recording function from the Baselines library to save video output of the more successful trained models. From viewing the videos, I was able to determine whether the model could complete the first level, which is listed on the first table in Figure 10.

The takeaway from this data is that none of the models which had achieved a mean training score of approximately 2,400 or more points (which was set as the baseline score for completing the level), was actually able to complete the level.

Results of viewing model video playback

	Cropped model (A)	Updated Discrete Action Space (B)	Model using V3 Hyperparameters (C)
Training Steps	5 million	5 million	5 million
Mean training score	1,901	2,508	2,742
Able to complete level 1	**No**	**No**	**No**
With model training continued to 20 million steps:			
Mean training score	6,705	n/a (didn't test)	6,471
Able to complete level 1	**Yes**	**n/a (didn't test)**	**Yes**

Figure 10: Results table from viewing video of models

As such, I took two of the models and continued their training to 20 million steps, and viewed the resulting videos of those models. In both instances, the models had been able to complete the first level in an extremely efficient manner. So, for this project, the conclusion is that a model can be trained to successfully complete the first level of Mario Bros.

An interesting observation from recording videos of the trained models is that, with any alteration to the observation space that the model was trained on, the quality of the model's actions is significantly decreased. For example, all of the models had been trained with the score cropped from the environment, but because it would be helpful to see the score when viewing video playback of the models, I initially recorded videos with the score cropped back into the observation space. But unfortunately, each model performed very poorly when the score information was added back to the display.

Finally, to look at whether a model trained to 20 million timesteps from the beginning would perform similarly to the models that I had continued from previous training, I trained a model using the default Baselines settings (with only cropping applied) to 20 million steps, and graphed the results alongside the two other models. That graph is displayed in Figure 11.

An inference that can be made from the graph in figure 11 is that a model in a somewhat difficult environment (in this case, one with sparse rewards for shaping desired behavior) may not always converge to a successful strategy of state-action pairs, even when trained for a fairly large number of steps. So, a takeaway with this case is that an ideal strategy for developing reinforcement learning models may be to initially train many models for fewer number of steps, then selectively choose the models that performed the best within that limit, to continue training to whatever the ultimate goal may be.

And further, while it ultimately appears from the graph in Figure 11 that, after 20 million steps of training, the model with tuned hyperparameters did not significantly outperform the model with default parameters, the model with tuned hyperparameters does seem to show a higher likelihood of initially converging to a successful strategy in a shortened training window, so there seems to be clear benefit to hyperparameter tuning, as well as other tweaks to the observation and action space.

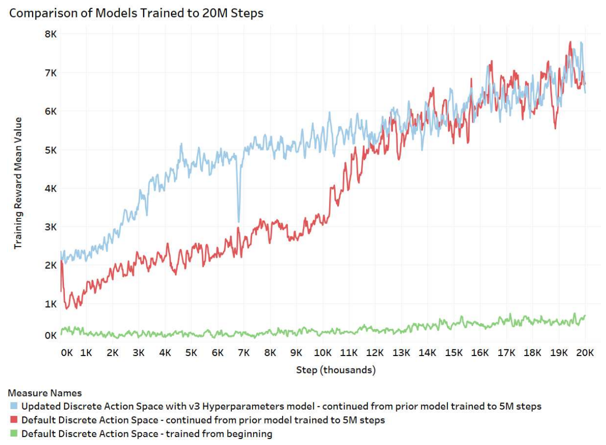


Figure 11: Training results of models trained to 20 million steps

Future Work

Because there was an obvious element of stochasticity with the reinforcement learning experiments conducted for this project, it would be beneficial to further explore alternate implementations of the PPO algorithm, to determine whether other implementations suffer from the same issue.

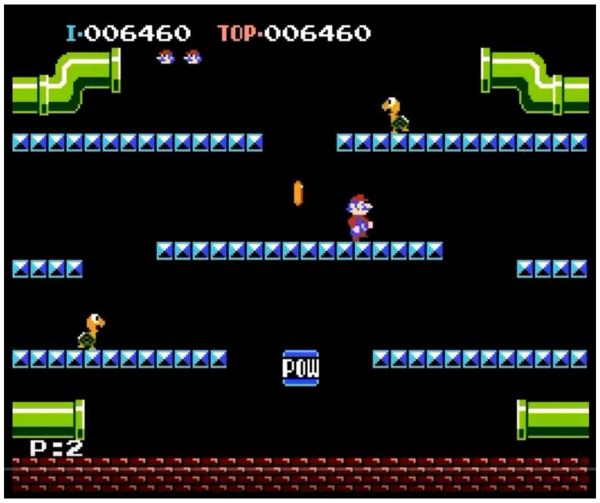
While the PPO algorithm was capable of learning to complete the first level of Mario Bros, I believe that there may exist other algorithm implementations that could improve upon the long training time. In particular, implementations of reinforcement learning algorithms that utilize behavioral cloning or imitation learning seem to be ideal candidates for improving upon the results of this project. Ideally, such an algorithm could be "pre-trained" on how to kill an enemy, and thus avoid the long, uncertain process of learning that behavior in an unsupervised manner.

Appendix: Gameplay Details

The game Mario Bros (from the Nintendo Entertainment System) has a fully-observable environment and a discrete action space. A screenshot of the game is included below. From my initial research, it appears that the gameplay progresses the same each time (such as, enemies' behavior will always be predictable unless affected by the player).

Regarding movement, the environment is static, but the player, as well as "enemies," do move. At each vertical level (see game screenshot below), moving to the edge of the screen will move the player to the opposite edge. To defeat enemies, the player must "stun" the enemy by jumping up and hitting it from the platform below, then run into the enemy while it is still stunned to kill it. An enemy will only stay in the stunned state for approximately 9 seconds, before it returns to normal function. For scoring, stunning an enemy returns 10 points, killing an enemy returns 800 points, and an additional 800 points can be received by collecting a coin that emits after an enemy is defeated. The emitted coins travel through the level and disappear if they reach the end before the player collects them. There is no time limit on each level, though enemies will increase in speed after the first two instances of being stunned.

Due to the two-step process of defeating an enemy, and the time factor in that process, there exists a challenge in developing a RL agent that can develop optimal behavior in the game. While an agent may occasionally kill an enemy from random sampling of actions (and accordingly get a big reward), a RL algorithm may fail to learn the full set of actions that contributed to receiving the reward. However, there are rewards for every step of completing a level: stunning an enemy, killing an enemy, and then repeating until all enemies are killed. So, given unlimited training iterations, a deep RL agent should eventually be able to learn to complete the entire game. However, realistically, training iterations will be limited for an RL agent.



Appendix Figure 1: a screenshot of the Mario Bros game on the first level

In addition, there is the possibility of an agent learning to farm points by stunning an enemy over and over again. In manual tests, there does not appear to be a limit on how many times a player can stun an enemy to receive 10 points, so a trained agent could learn to repeat this behavior but not ever kill the enemy. However, the game does have a built-in defense against this behavior, which is that enemies will transform into a faster version after being stunned twice. Accordingly, point-farming behavior will lead to the game becoming more difficult, and so it seems unlikely for a RL to achieve a large score from this behavior.

References

IBM. 2020. Convolutional Neural Networks. https://www.ibm.com/cloud/learn/convolutional-neural-networks.

OpenAI. n.d.. http://spinningup.openai.com

OpenAI. n.d.. http://gym.openai.com

OpenAI. 2017. Proximal Policy Optimization https://openai.com/blog/openai-baselines-ppo/

OpenAI. 2018. Retro Contest: Results. https://openai.com/blog/first-retro-contest-retrospective/

Poliquin, M. 2019. How to setup Open AI Baselines + Retro. http://www.videogames.ai/2019/01/29/Setup-OpenAI-baselines-retro.html

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG].

World of Longplays. 2012. NES Longplay [191] Mario Bros. https://www.youtube.com/watch?v=WFptXdODy7k

Yang, Yu. 2018. main_ppo.py [Source code]. https://github.com/eyounx/RetroCodes/blob/master/A3gent/main_ppo.py.