Deep Pipes: Deep Reinforcement Learning to Solve Super Mario Bros

Emmanuel Guzman
eguzman3@gatech.edu

John Michael Burke
jburke65@gatech.edu

Haitham M. Dawood
hdawood6@gatech.edu

Asma Beevi Kuriparambil Thekkumpate
abkt3@r@gatech.edu

Abstract

Modern deep Reinforcement Learning (RL) has a wide range of approaches for solving problems including game playing, robotics, and control systems. We consider the game playing scenario where training data is abundant and can be sampled from a virtual environment. In particular, we selected the Super Mario Bros (SMB) [14] game and explored the performance of various deep reinforcement learning algorithms in this environment. Namely, we adopted Deep-Q-Network (DQN), Path Consistency Learning (PCL), Proximal Policy Optimization (PPO), and Soft Actor Critic (SAC). The objective of this analysis is to evaluate and juxtapose the results of these algorithms to the Super Mario Bros simulator.

1. Introduction/Background/Motivation

Super Mario Bros (SMB) is a classic game made for the Nintendo Entertainment System (NES) and has become a brand which has been iterated on countless times even recently [25]. The game features a collection of worlds each containing a set of smaller levels referred to as stages [4]. We experiment with applying four deep RL algorithms to the first stage of the first world to evaluate sample efficiency, model stability, as well as train and evaluation performance. The ability to successfully train an agent to complete a challenging game is an impressive feat. We chose this simulator because it seemed relatively challenging and is frequently referred to in the RL community in addition to being chosen as the framework for PyTorch's RL Tutorial demonstrating Double Q-Learning [11].

The simulator environment, Gym [8], produces samples online in the form of a pixel-based state (height, width, channels), reward varying between [-15, 15], a boolean of whether or not the environment is done, and a map of various additional information about the current state of the game. This data is provided whenever a step is taken which requires providing the input action. Two frames taken from different sections of World 1 Stage 1 are shown in Figure 1. We selected the default action space, SIMPLE_ACTIONS (7 total actions) for this experiment. In the upcoming sections, we describe the algorithms we have chosen to solve SMB game as well as our approach towards implementing them. Challenges and results from each algorithm are then presented. The paper concludes with some comparisons and remarks.

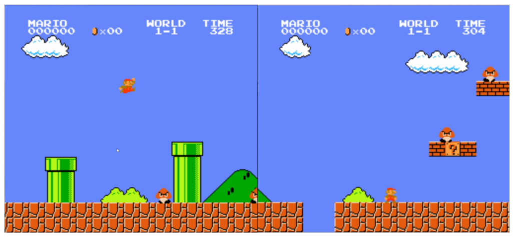

Figure 1. Two frames from the Super Mario Bros simulator showing characteristics of the environment.

2. Approach

There exists a multitude of algorithms in RL. They vary primarily based on learning styles such as on-policy or off-policy, referring to whether or not the algorithm estimates value from discounted rewards or from state-action value estimates. The algorithms we selected to solve SMB game include a mixture of on-policy and off-policy (personal interests of team members have also been a selection criteria). They are: Deep-Q Network (DQN), Path Consistency Learning (PCL), Proximal Policy Optimization (PPO), and Soft Actor Critic (SAC). In the following subsections, a brief summary and implementation approaches for each of these algorithms are described.

2.1. Deep Q-Network (DQN) Algorithm

The Deep Q network agent implemented in this project is inspired by the seminal work on deep RL by Mnih et. al [16]. The agent actions are chosen to maximize the reward for the given state using a neural network-based optimal action value function approximation Q(s,a). The RL model is evaluated using the mean squared difference between the neural network predicted expected reward $Q(s,a;\theta)$ and a more accurate expected reward approximation $r + \gamma * max_{a'} Q(s', a'; \theta')$.

We start by exploring state-action space randomly at first and slowly reduce this exploration rate to let the agent further learn from the optimal state-action mapping it discovered. The state, action and observation information are stored in an experience replay buffer. This replay buffer is randomly sampled to ensure independently and identically distributed inputs. To further reduce the correlation between observations and recently learned state action mapping, a target network was used to gather experiences while an online network was used to learn from these observations as in [17]. Further, to speed up the simulation, frame-skipping [16] was implemented where the same action is applied for a consecutive number of frames and the cumulative reward is collected.

2.1.1 Deep-Q Neural Network Models

RL and DQN in particular requires a large number of steps for the algorithm to converge. Hence, light-weight models were used to speed up the sequential action-space exploration. Further, to understand how much higher representation ability of neural network helps in the Q learning, a shallow and wide 2 layer fully connected neural network is compared against a deeper but less wide 4 layer neural network (2 layers of this deeper network are convolutional layers).

The RL algorithm's performance was contrasted and compared between 2 models where one has a simpler action space (but sufficient to solve the game) and the other one which has a more diverse action space.

2.1.2 Trade offs between performance and computational complexity

RL often requires a large number of steps for the model even to show signs of robust learning. Human in the loop hyperparameter tuning strategy was used. To speed up the simulation, cloud computing was utilized. Tensor-board library was utilized to get immediate visual feedback of monitored metrics like loss, reward etc. Many hyper parameters like replay buffer size etc. were limited by compute memory.

2.2. Path Consistency Learning (PCL) Algorithm

The main disadvantage of value based (off-policy) methods is their instability. This requires going through a tedious hyper-parameter tuning to achieve stability of the results. Moreover, lack of theoretical understanding of how these methods approach optimum behavior is another drawback.

While policy based (on-policy) methods show more stability than off-policy methods, they typically go through many policy iterations (PI) while training the agent. Rollout is the simplest of these PI methods where at each iteration, only one improved/modified policy is obtained given a priori policy [6]. Therefore, the variance associated with these policies is typically large which leads to inefficiency in training these agents.

PCL Algorithm was proposed [19] to reconcile the gap between value and policy based RL approaches and combine their respective advantages. PCL can be viewed as a generalization of actor-critic and Q-learning algorithms.

The PCL algorithm considers an on-policy trace [by including a policy, $\pi(\theta)$] and an off-policy trace [by including a state value function, V(ϕ)]. The PCL paper [19] provides a notion of what they call "soft consistency error" known as "C". In essence, "C" represent the error/distance between the on-policy and off-policy components of the algorithm if a specific sequences of actions are taken (see also [18]). The objective function while training the PCL algorithm is to minimize the square of the "soft consistency error". Based on this optimization, gradients are calculated to update the on-policy and off-policy components of the algorithm. As this error decreases, the agent is trained to select the best possible actions given a certain state of the environment.

2.2.1 Implementation

Given the complexity of the algorithm, we decided to use a readily available implementation of PCL as coded in ChainerRL [12]. ChainerRL is a deep reinforcement learning library in Python built on top of Chainer framework [1]. In setting up the code for training, we started with two example code [2, 3]. The ChainerRL code was originally setup to train the *CartPole-v0* gym environment. While most of the logic was maintained, multiple changes were made to the base code as well as some of the underlying functions in ChainerRL to allow for easier experimentation.

More specifically: the use of SMB environment is incorporated in the code, multiple wrappers were used to process the frames obtained from the environment (i.e., *MaxAndSkipEnv* to skip frames and *WarpFrame_variable* to resize frames); ChainerRL *WarpFrame_variable* function was modified to use the dimension of the resized frame as an input; and the input parameters of multiple functions were modified to explore beyond the standard values of some inputs.

2.2.2 Challenges and Lessons Learned

Originally, using a readily available library was motivated by the interest in spending more time exploring the results while learning a state-of-the-art RL library (rather than coding the algorithm from scratch). The main challenge faced was to understand the ins and outs of the library since the PCL algorithm uses a wide range of inputs (and calls several Chainer and ChainerRL functions in the process). Achieving a stable run was a tedious process and needed multiple iterations Depending on the complexity of the NN, runs could take between few hours to several days.

2.3. Proximal Policy Optimization (PPO) Algorithm

The Proximal Policy Optimization algorithm is an on-policy algorithm which learns policy gradients while trying to optimize a custom loss function. This loss function is comprised of a few interchangeable surrogate objective functions. PPO bulids off of Trust Region Policy Optimization (TRPO)'s surrogate objective function of the advantaged ratio of policy probabilities but adds clipping and drops the Kullback–Leibler (KL) divergence objective [22]. The final objective loss function adds the advantaged clipped policy ratio, mean squared error of the critic value function with some constant, and finally an entropy bonus of the actor's probability distribution; seen below [22]:

$$L_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF} + c_2 S[\pi_\theta](s_t)]$$

This experiment uses the alternative advantage function, Generalized Advantage Estimation (GAE), proposed in another paper [21]. The GAE advantage involves performing a TD(λ) style optimization across multiple epochs of batches of trajectories collected recently with the current policy. The simplified formula for the GAE equation [21]:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l}^V$$

Although the implementation of this algorithm can be quite thorough, PPO is often the preferred RL algorithm due to it's ease of setup and sample efficiency [10]. Finally, there have been many variations to the PPO algorithm but getting the setup correct seems to be key to it's success. Two additions to the algorithm used were reward scaling from **[-15, 15]** to **[-1,1]** and global gradient clipping the norm to **0.2**; these were not all the suggestions from this paper but appeared to make a difference [10]. The implementation of this algorithm was adapted using portions of an existing codebase for PPO and the same SMB environment but in a non-concurrent manner [20].

2.3.1 Hyper-parameters and Implementation Details

Outside of the main algorithm the only major differences are the non-concurrent implementation of the algorithm and the various hyper-parameters. In order to process the environment state, a frame skip and stack was implemented where all actions would be repeated N times, in this case 4, and each frame would be stacked together across a new dimension. Each frame was originally in RGB but converted to grayscale by taking the mean. Each stacked observation was fed to an actor and critic model consisting of $3x32x(7-2j)x(7-2j)$ convolutional neural networks and 2x512 fully connected layers. Each network was assigned an Adam optmizer with a learning rate of **1e-4**. Although other optimizers, Adagrad and Adam with AMSGrad, and a learning rate schedule were experimented with, these alternatives' results were unsuccessful and dropped in the end. A buffer was used to store recent memories which would be cleared after every update, or upon the buffer filling up, which stored up to **512** observations. When the buffer filled, **8** minibatches of **32** length trajectories would be randomly selected. This policy optimization was repeated **20** times per update which would typically occur every 2-5 games early in training and almost 2-4 times per game approaching an optimal policy.

2.4. Soft Actor Critic (SAC) Algorithm

The Soft Actor-Critic algorithm was implemented from the ground up referencing the the original paper [13] and accompanying OpenAI supplemental [5] documentation. Other publications and online lectures [23, 9] helped clarify ambiguities as well as provided ideas for mitigating training instability.

SAC leverages an actor network (policy $\pi_\phi(a|s)$) and four critic networks (two primary Q-functions $Q_{1,2}(s,a)$ and two target Q-functions $Q_{target,1,2}(s,a)$ similar to DDPG [24]), and in place of external exploration strategies relies on internally trained stochasticity in the actor's neural network. The loss functions between the actor and critic use SGD in competing maximization and minimization, as well as introducing entropy-based compensations to the reward for exploration. We explore what this mean in the following sections.

2.4.1 Implementation

State-Action Value Function: the network architecture of choice was a mostly standard CNN/Fully-Connected layer combination as the input states were raw pixel values and the outputs were the joypad actions in the NES emulator. The loss function for the critics was Mean Squared Error (MSE) and the optimization was stochastic gradient *descent*, thus we needed a target $y_t(s, \tilde{a}, r, s', d)$ to compare against. This could be computed as follows:

$$r + \gamma(1-d)\mathbb{E}\left[\min_{i=1,2} Q_{target,i}(s',\cdot) - \alpha \log \pi_\phi(\cdot|s')\right]$$

We use the $Q_{target,1,2}(s,\tilde{a})$ models, which we shall expound on soon. The $\tilde{a}$ indicates the probabilities are sampled from the actor and we use $\cdot$ to denote the vector of all actions. Conceptually we can see that when the actor $\pi_\phi(a|s)$ predicts low probabilities for certain actions, the log-probabilities will be high, thus signaling the target critic that there may be be higher values through other actions, this is then scaled by the action probabilities. MSE is then computed between $Q_{1,2}(s,a)$ and $y_t(s,\tilde{a},r,s',d)$ where a is taken from the replay buffer. The objective then is to minimize the distance between the primary critic's state-action value and the expected future reward.

Policy Function: this network shares the same core architecture as the critics, but ends with a softmax layer whose probability distribution can be used to sample actions from. The loss function for the actor was computed as the mean value of the following:

$$\mathbb{E}\left[\min_{i=1,2} Q_i(s,\cdot) - \alpha \log \pi_\phi(\cdot|s')\right]$$

This optimization uses stochastic gradient *ascent* which can simply be the negative of the final value for backpropagation.

Target State-Action and Temperature: the aforementioned target critics aren't updated with backpropagation and are instead updated as follow:

$$Q_{target,1,2} \leftarrow \tau Q_{1,2} + (1-\tau) Q_{target,1,2}$$

The temperature value α which we have seen accompanying the log-probabilities gives a semblance of how much exploration we encourage the model to partake in. We followed the method described by Christodoulou [9] to apply stochastic gradient *descent* on the mean of the following:

$$\mathbb{E}\left[-\alpha(\log \pi_\phi(\cdot|s) + \alpha_{target})\right]$$

The value α_{target} has empirically been found to be $0.98\log(N)$ where N is the number of actions, thus for the SIMPLE_ACTION set for SMB $\alpha_{target} \approx 0.82819$.

2.4.2 Challenges and Lessons Learned

Christodoulou [9] noted SAC had mixed results in discrete-action space (up to a 99% regression on very simple environments like Pong), and SAC being an actor-critic model meant there would be inherent instability to overcome. Precision issues were what plagued us the most, often times ending the learning process altogether from overly-pessimistic action probabilities that were problematic when computing log-probabilities, as well as near-zero temperatures grinding exploration to a halt. Through careful computation log tracings some remedies such as clamping functions were introduced to prevent these undesired effects. Data augmentation [15] was leveraged modifying the provided code to apply the same transformation to both state and next state pairs.

3. Experiments and Results

We discussed the implementation details of our 4 different RL algorithms in Section 2. In this section, we present the results of various experiments we performed and analysis we conducted.

3.1. Deep-Q Network

Three deep-Q models were chosen for detailed experiments after preliminary architectural and hyper-parameter search. Model A is a 7 action space 4 layer convolutional NN with a total parameters of about 1.56 million. Model B has an action space of only 4 and is otherwise the same as Model A. Model C is a 2 layer fully connected neural network with a total number of parameters of about 4 million.

The deep-Q loss curves for the three models are shown in Figure 2(a). The deep-Q training rewards (with a minimum explore-exploit ratio of 0.1) obtained using these models are shown in Figure 2(b). The non-stationary statistics of MSE loss is clearly evident from the loss curves. The reward curves for Model A and Model B shows convergence behavior after about 4 million steps. But such a conclusion cannot be derived by purely looking at MSE loss. By referring to reward plot as well as loss plot, we can infer that Model A and Model B achieves convergence while Model C does not. Model C has only 2 layers despite having about 4 times the number of parameters as compared to Model A & B. This shows why depth is important to achieve parameter efficiency and representation learning in Deep Learning - shallow network while being faster lacks the higher dimensional representation ability deeper networks possess.

The baseline performance that the deep-Q agent needs to beat is the average reward of a random acting agent. Random agent collects an average reward of 550 while not completing the game level. Model A achieves average training reward greater than 1200 and average validation reward greater than 2200. The training rewards were collected with a minimum explore exploit ratio of 0.1. Interestingly, Model B has a training reward of only 1000 and average validation reward of only 1500. Model B has a reduced, and simpler action space - but Model A with a richer action space and is performing better. This might be due to the neural network having enough capacity to model a bigger state-action space.

Another interesting observation is that we cannot compare the three RL models based on the loss metric. We can see that Model C has a lower MSE loss as compared to Model A and B while its collected reward is no better than that of a random acting agent. This is again due to the non-stationary statistics of DQN.

3.2. Path Consistency Learning (PCL)

One of the main disadvantages of the off-policy algorithms mentioned by the PCL developers was the tedious

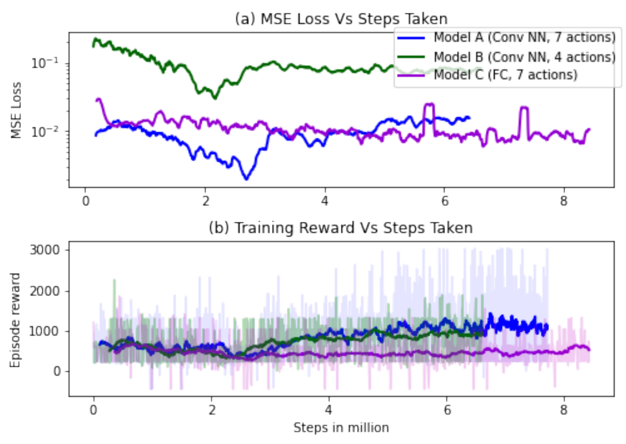

Figure 2. DQN loss and reward curves across training steps for the three models.

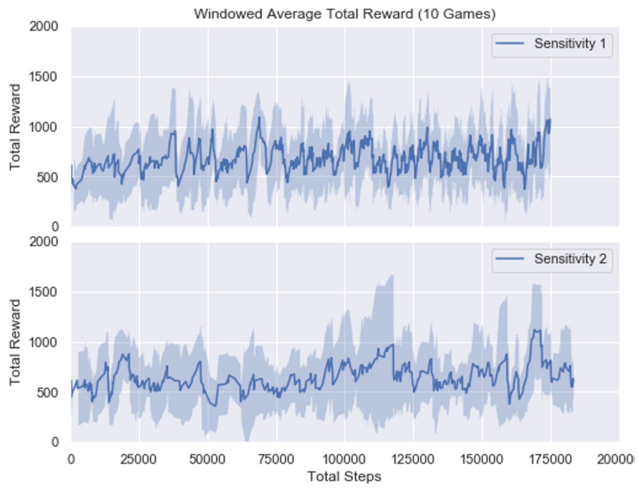

Figure 3. PCL Total Game Reward Accumulation as Training Progressed for- Top: Batch size=200, Bottom: Batch size=5.

process of hyper-parameter tunning. The irony is that PCL itself needs many parameters to be tuned (see section B.3 [19]).

The main parameters changed in the sensitivity analyses are listed below. The values between parentheses are the values used in the runs presented herein. The final hyper-parameter selection was based on preliminary analyses and are within within the range of hyper-parameters cited in the PCL paper:

1. Number of frames to be skipped [skipped 4 frames],
2. Dimension of resized frames [128x128],
3. NN architecture (number of channels and hidden FC layers) [5 FC layers with 500 channels and ReLU in between],
4. Learning rate [0.001],
5. The (decaying) epsilon-greedy behavior of the agent [epsilon starts at 1 and linearly decays with the steps. It reaches 0.1 at 200k steps],
6. Batch of on-policy trajectories [5 and 200], and
7. Frequency of updating the agent [at end of each game].

In total, about 50 sensitivities were performed with different combinations of hyper-parameters. Figure 3 shows the rewards accumulated overtime for two sensitivities. The solid line represent the moving average for rewards over 10 consecutive games. The light shaded area represent the +/- standard deviation. The only difference between both sensitivities is the batch (200 for Sensitivity 1 and 5 for Sensitivity 2). Using a larger batch size considerably impacted the stability of obtained rewards because the algorithm is checking several trajectories. Nevertheless, both presented sensitivities do not appear to converge, likely because they needed much more steps to converge.

The training was not a complete success. The trend of reward was going upwards for several games, but the algorithm was not stable enough to converge within the number of steps used. This lack of convergence can be attributed to: 1) There are too many hyper-parameters to tune, so possibly the tested combinations were not optimum. While the range of hyper-parameters tested by the authors of the PCL paper were provided, no specific values were vetted, 2) We decided to explore more combinations of parameters instead of training runs with millions of steps, 3) There could be a bug in the implementations that was not caught, and 4) The decision not to use CONV layers could have impacted the capacity of the NN (as noted in DQN section).

The main lesson learned is that it is a challenge to achieve steady results with complex algorithms. Achieving results similar to the ones presented in research papers can be tricky. This becomes more obvious when the algorithm is applied to a different game/application.

Lastly, many parameters had an impact on the training behavior including: 1) The rate of decay of the epsilon (probability of adopting a random action), 2) The size of reduced frame, and 3) The number of skipped frames. It is worth noting that the optimum number of skipped frames varies significantly from game to another [7].

3.3. Proximal Policy Optimization (PPO)

In Figure 4, two plots can be seen depicting the loss and reward curves of the algorithm during training. The PPO algorithm was difficult to start learning and the loss curves show constant fluctuation. The L^{vf} loss appears to increase more than anything typically followed by a catastrophic decrease in rewards. Although the variance between games is not too drastic which aligns with the on-policy nature of PPO, learning incrementally against policies off only recent experiences. As mentioned earlier not all of the suggestions

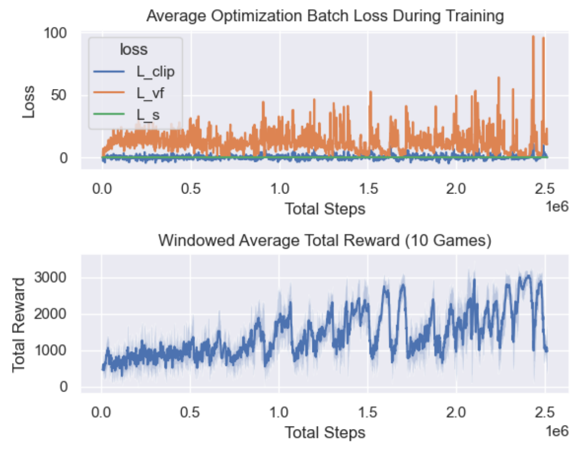

Figure 4. A) PPO policy surrogate, L^{clip}, value function, L^{vf}, and entropy bonus, L^{s} loss curves during training. B) PPO average total reward using a rolling window of 10 games during training.

from PPO case study were included which may be evident here [10]. Also the lack of a learning rate schedule could have contributed to the slow learning and the catastrophic declines observed. The loss curves may also be misinterpreted because as they are the average of each minibatch across each iteration for the most recent optimization at the end of the game. The loss significantly decreases in the beginning backprops of the update step versus the final few backprops of the last iteration but it was not intuitive as to how best to log and chart these loss values. As for sample efficiency, the SAC algorithm seems to learn much quicker than PPO where PPO doesn't reach roughly the same high reward until **1** million steps but training eventually reaches over **3000** at **2.4** million. This could be related to the SAC algorithm performing more frequent optimization steps. An interesting find between training was the eval model would get stuck because of taking argmax of the actor. When adding a random move after a max action repeat, the model would learn to maximize the right action far away from holes such that a random move would always land in a safe space. With this model and repeat penalty, the final evaluation model achieves an average of **2887.5** winning **90%** of games played.

3.4. Soft Actor-Critic (SAC)

The measure of success can be derived from questions Haarnoja *et al.* [13] originally asked, which was if a randomly behaving actor could solve the environment it was in. A realistic first benchmark to beat is that of an agent taking random actions; as previously notes, random actions achieve approximately 550 reward game points, beating this is a great entry point. Secondly we wanted to see if the agent could beat the SMB first level, the SAC agent did

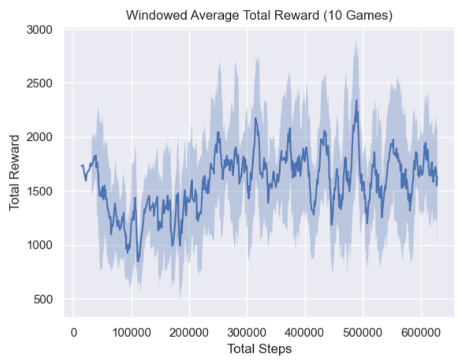

Figure 5. SAC total game reward accumulation during training

indeed beat these two benchmarks and on occasion would be seen flying through the environment executing near perfect jumps. It wasn't without it's flaws though, at times the agent would become overconfident in seemingly poor action choices like sprinting into Goombas/holes upon reviving or running into pipes until time ran out leading to entire sequences of games with poor rewards until the actor's probability distribution adjusted accordingly.

Altogether Figure 5 shows a sample of rewards accumulated over the training duration, we note that this is well over the territory of random movements and at time well into scores expected from a completed first level.

4. Conclusion

Using readily available libraries (e.g., ChainerRL) to run the RL algorithms is not a trivial task. While some established ML libraries are flexible and powerful, RL libraries are not necessarily as stable. Deep Q learning was showing relatively stable behavior as compared to the other algorithms while it took significantly more steps to achieve convergence behavior. We found that between PPO, SAC and PCL there was a fair amount of instability where the algorithm seemed to do well and then drastically fail indicating there was an opportunity for better control of the learning via learning schedules or regularization.

In future work, we would like to redo this analysis by constraining more of the shared hyper-parameters such as the game random seed, memory buffer sizes, or how frequently an optimization step is performed. Running our learned models on other world stages showed worse than random performance so potentially training models on a mixture of world stages could lead to more general performance.

5. Work Division

This project was tackled as a survey, thus each member was responsible for an entire model implementa-

Student Name	Contributed Aspects	Details
Asma	DQN	Implemented DQN and trained on cloud resources.
Emmanuel	SAC	Implemented SAC and trained on local accelerators.
Haitham	PCL	Adopted PCL from ChainerRL and trained on local machine
John Michael	PPO	Implemented PPO with some adoption from [20] and ran on a local GPU.

Table 1. Contributions of team members.

tion/analysis as well as all the training/plotting. We have a more specific breakdown in Table 1 for the work division. Altogether the group supported the analysis of each other's algorithms, code, and came together to prepare this report. Weekly, and at times daily, meetings were conducted to ensure the project was moving forward, we had the opportunity to review each other's code, and work through any bugs or implementation difficulties that came up.

References

[1] Chainer Library. https://github.com/chainer/chainer. Accessed: 2021-12-13. 2

[2] ChainerRL Example Code. https://github.com/chainer/chainerrl/blob/master/examples/gym/train_pcl_gym.py. Accessed: 2021-12-13. 2

[3] Colab MarioRL Code. https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/54f5097c720c6f2656219ab14a4e7431/mario_rl_tutorial.ipynb. Accessed: 2021-12-13. 2

[4] Super mario bros game online. https://supermario-game.com/. Accessed: 2021-12-13. 1

[5] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. 3

[6] Dimitri Bertsekas. Multiagent reinforcement learning: Rollout and policy iteration. *IEEE/CAA Journal of Automatica Sinica*, 8(2):249–272, 2021. 2

[7] Alexander Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *AAAI Workshop: Learning for General Competency in Video Games*, 2015. 5

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 1

[9] Petros Christodoulou. Soft actor-critic for discrete action settings. *CoRR*, abs/1910.07207, 2019. 3, 4

[10] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on ppo and trpo, 2020. 3, 6

[11] Yuansong Feng, Suraj Subramanian, Howard Wang, and Steven Guo. Train a mario-playing rl agent. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html. Accessed: 2021-12-13. 1

[12] Yasuhiro Fujita, Toshiki Kataoka, Prabhat Nagarajan, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *CoRR*, abs/1912.03905, 2019. 2

[13] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer G. Dy and Andreas Krause, editors, *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018. 3, 6

[14] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. 1

[15] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *CoRR*, abs/2004.14990, 2020. 4

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, and Wierstra. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. 1, 2

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518, 2015. 2

[18] Ofir Nachum, Yinlam Chow, and Mohammad Ghavamzadeh. Path consistency learning in tsallis entropy regularized mdps. *CoRR*, abs/1802.03501, 2018. 2

[19] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. *CoRR*, abs/1702.08892, 2017. 2, 5

[20] Viet Nguyen. [pytorch] proximal policy optimization (ppo) for playing super mario bros. https://github.com/uvipen/Super-mario-bros-PPO-pytorch, 2013. 3, 7

[21] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. 3

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 3

[23] Olivier Sigaud. Soft actor critic (v2). 3

[24] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015. cite arxiv:1511.06581Comment: 15 pages, 5 figures, and 5 tables. 3

[25] Wikipedia contributors. List of video games featuring mario. https://en.wikipedia.org/wiki/List_of_video_games_featuring_Mario, 2021. 1