Reinforcement Learning for Street Fighter II - A Study

Sunny Chen
Rutgers ECE Department
sunnychen2000@gmail.com

Anis Chihoub
Rutgers ECE Department
ac1906@scarletmail.rutgers.edu

Stanley Chou
Rutgers ECE Department
sc2029@scarletmail.rutgers.edu

***Abstract*—With Reinforcement Learning becoming a popular method to solve various problems, we aim to explore its use in playing classic video games such as Street Fighter. We use DQN and PPO to teach a reinforcement learning algorithm to play Street Fighter II. We find that the PPO algorithm outperforms DQN when playing Street Fighter II.**

***Index Terms*—Reinforcement Learning , DQN, PPO Street Fighter**

I. Introduction

Reinforcement learning (RL) at its core trains a model to perform a set of discrete options. The main goal is learning to take actions that maximize the cumulative rewards. Reinforcement learning problems are usually set up as a markov decision process, which is a mathematical framework to make decisions when the environment is random and partly in the hands of the decision maker. We can use a neural network as a function approximator for the policy, so we do not need explicit transition probabilities to make a decision. Using this information, RL has become a popular method for solving various problems related to control.

To this end, our group decided to take on the challenge of creating an RL algorithm to play Street Fighter. We selected street fighter because it is a retro game that shares similarities to some prior work that has been done by OpenAI. In addition, by obtaining a ROM, we can integrate OpenAI gym and do our development there.

II. Game

As stated before, our video game of choice is Street Fighter II. Street Fighter II is an arcade video game released in 1991 for the CP Arcade System [1]. At its core, Street Fighter II is a fighting game where the player can select a fighter and compete against an AI fighter. With the popularity of street fighter in the 1990's led to a golden age of competitive games and grassroots tournaments, cementing its legacy as one of the best arcade games of all time.

Fighting games are one of the most technically challenging genres within the realm of competitive gaming. At a glance, Street Fighter seems relatively straight forward when compared to other competitive multiplayer games, almost rudimentary due to its simplicity. The level is an extremely basic flat side-scroller and any gameplay elements involving luck or chance are stripped away. All that's left is the essence of a fighting game, a pure test of reaction time, spacing, and execution ability against the opponent. Such a heavy focus on the fundamentals is what makes fighting games notoriously difficult for beginners to pick up. On top of this, learning to predict what your opponent is going to do, how to force your opponent into unfavorable future positions, and the extremely large amount of possible options you need to consider cause Street Fighter to be a very mentally challenging game too. This difficulty combined with the fact that fighting games have little variation with almost no luck involved make Street Fighter II a perfect challenge for us. We wanted to see how strong of an agent we could create without the limitations of reaction time and human misinputs, in a game where success is entirely dependent on technical ability. The difficulty and high skill floor of Street Fighter II make this an especially challenging task.

Before we describe our methodology, we will describe the environment, actions, state, and rewards for Street Fighter II. The environment is the fighting situation we are in. In this case, we selected the character Ryu, who will face different opponents via trying to complete the campaign of Street Fighter 2. The state is the image on our screen. The fighter could be in the midst of an attack, moving around etc. This is described by a $200x256x3$ image that we can process for our Reinforcement Learning algorithm. We also have state of the buttons. The buttons in this case correspond to up, down, left, right, as well as eight input buttons for actions such as jabbing, kicking, and round housing. Our actions correspond to the buttons that we mentioned before and are represented via a 12x1 multi-binary array where 1 implies that button is being pressed. We want to either move or attack based on the current situation. Lastly, the rewards for this game can range from the health the fighter has to the time they are still alive. Manipulating the reward function will be one of the main objectives in order to maximize the success of our algorithm.

III. Methodology

A. Processing Input

As described before, each input image is a $200x256x3$ image. This is approximately 153600 entries that we would need to process. To simplify our analysis, we decided to resize each image to $64x64x1$. This ensured that the training time would not take too long, but preserve enough information to work with. In addition, this now means that each image is grayscale instead of RGB, so we trade-off some information loss for faster computation time. However, since we care more about the actions and surrounding environment, this is

a worthwhile trade off. To represent our actions as a vector, we will use one multi-binary encoding. Each vector is one by twelve, where a binary label of 1 at some index represents that the button/input represented at that index is being pressed. To ensure that our agent plays the game as naturally as possible, we limit the actions to a human based action. This means that we cannot take actions that are impossible for a human to enter, such as down and up on the joystick at the same time.

B. Methods

Now, we will describe our reinforcement learning algorithms. We considered two algorithms, Deep Q Learning (DQN) and Proximal Policy Optimization (PPO).

1) DQN: In Q-Learning methods, the goal is to approximate some Q function that represents the expected sum of future rewards for all state-action pairs (s, a). The policy (component which controls the agent actions) is inferred from this Q function as $\pi^*(s) = \text{argmax}_a\, Q(s, a)$, in other words choosing the action which maximizes the value of Q function. In Deep Q-Networks (DQN), a deep neural net is used as the Q approximator and will be the focus of our training. For our implementation we have an additional target network to serve as a stable point for computing loss, which will be periodically updated to match our main Q network. However, Q-Learning is a relatively old method of reinforcement learning first introduced in 1989 and has some drawbacks. It is slow to train and quite unstable, occasionally showing catastrophic collapses where the agent performance plummets. Additionally, Q-Learning is designed for deterministic state-action spaces where there will always be a "best" action for every state, such as the cart-pole problem. This is especially restrictive for multiplayer games, which cannot be deterministic as opponent behavior is not known. SF2 is then best modeled by a stochastic policy because the agent cannot make a guaranteed best move, only one that will have the highest probability of succeeding depending on the opponent's actions.

2) PPO: Policy gradient methods are a newer family of RL algorithms. Instead of approximating a Q function and inferring the optimal policy, policy gradient methods seek to optimize the policy itself to maximize reward. The policy will not be deterministic in nature (i.e. always selecting action that maximizes Q value) so each state-action decision has an associated probability and this offers better suitability for problems which require a stochastic policy. A neural net is used to represent the policy itself and each iteration, the observed reward and chosen actions update the policy gradient, which is used to adjust weights of the policy neural network. Proximity Policy Optimization (PPO) was introduced by OpenAI in 2017 as an improvement on previous policy gradient methods. While optimizing, we will want constrain our policy within a "trust region" to keep policy changes relatively small. This constraint involves computing a value called KL-divergence, which is an expensive second derivative computation that must be run every training step. PPO improves on this by using a soft constraint implemented as a penalty score instead of a hard constraint, bypassing the need for this expensive computation. The main benefit is that PPO is that it is much faster than older policy gradient methods and allows training for complex agents to be computationally feasible.

As we will discuss later, we started off using DQN but eventually switched to PPO for a multitude of reasons. PPO methods are newer and better reflect the current state of RL, and we also found empirical evidence supporting our switch.

C. Reward Function

One of the most important parts of any reinforcement learning algorithm is the reward function. In this case, we experiment with the change in score, the change in enemy health, and the change in the players health. For given player and enemy health x_1, x_2 respectively, and score y, we can write the following:

$$f(x) = y{+}50{*}(x_2{-}x_2'){-}50{-}{*}(x_1{-}x_1'){+}500{*}win{-}500{*}loss \tag{1}$$

In the above equation, the variables with a prime represent a new data point in regards to a character's health. We decided upon this score function because we wanted to penalize the losing health, but reward hurting your enemy or staying alive. The changes in health are multiplied by 50 to match the values of the score. The win and loss variables are just indicator variables on whether a win or loss just happened, We wanted to encourage our agent to win, and in practice we noticed by adding a positive reward for roud wins and negative reward for losses the agent learned a better policy.

D. Hyperparameter Tuning

Hyperparameter tuning is a necessary step of training ML models. For our PPO algorithm there were 5 hyper parameters that we had control over (of steps, discount factor gamma, learning rate, clip range, and GAE lambda). We used Optuna, an optimization toolkit that evaluates model performance across a range of hyperparameter values and selects the best performing ones. We train and evaluate a model on each set of intelligently chosen hyper parameters for 100,000 time steps and repeated this process 150 times for different sets of hyperparameters. Afterwards, we then tested each model hyper parameter pair on 10 full episodes of SF2 gameplay, we then took the 2 best average performing models based on reward after the 10 episodes to be the models we continued to train for our final models.

IV. Results

We trained the DQN and PPO algorithms on a PC with multiple NVIDIA GPUs. For the DQN algorithm, we trained the model for 250 episodes and the results showed that the algorithm was unstable and took too long to compute, 8 hours for only 250 episodes of training. These findings led the team to abandon the DQN methodology. For PPO, we used 10 million episodes as prior literature recommended training for millions of episodes. Below are images of the rewards for each result. We used tensorboard to log the cumulative reward function as we wanted to monitor the training over each episode to observe an overall trend.

A. DQN Results

Below are the results of the DQN algorithm. Notice that the training curve is very unstable for the 250 episodes we trained our algorithm for. In blue are the actual reward values, the yellow is the approximate mean of those values. Again, while we may have trained the model for too short a time to get valid results. These results alone took around 8 hours to compute and in the time span of the project we would finish with no results if we continued training the DQN.

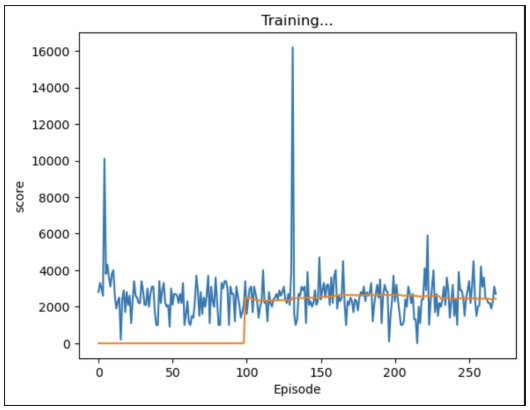

Fig. 1. DQN Rewards

B. PPO Results

Below are the results of the PPO algorithm we implemented via the stable baselines3 library. The first image describes the mean rewards at each episode. The second image describes the episode length for 10 million episodes. Notice that the results clearly indicate that the fighter survives for longer and the mean reward also generally increases. Furthermore, the results shown only took about 10 hours to compute 10 million time steps and over 1 million episodes compared to our DQN implementation's 250 episodes in 8 hours. Another thing to notice is how our model seemed to start performing worse after episode 5,100,000. We attribute this collapse to our learning rate being too high as what that collapse implies is that the model took too large of a step in how it played the game that caused it too perform worse. Generally we saw in literature that learning rates around 1e-7 performed well for reinforcement learning in an environment as complex as street fighter 2, so we chose a learning rate of 2.75e-7, but even this learning rate was possibly too high for the model to train with causing this collapse.

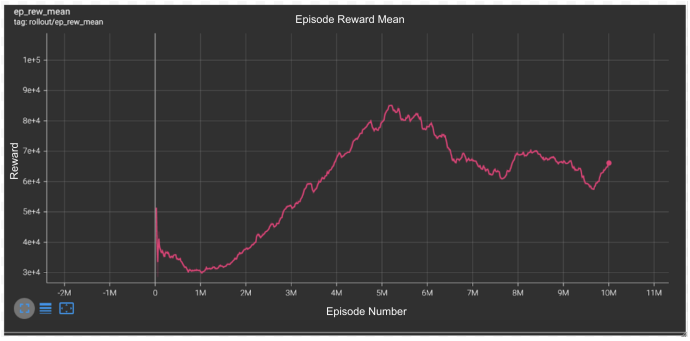

Fig. 2. PPO Rewards Mean for 10M episodes

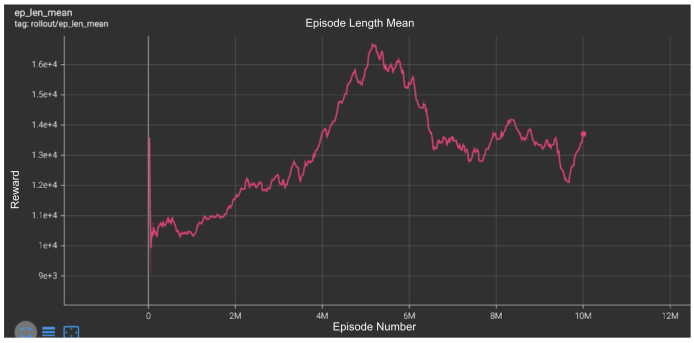

Fig. 3. PPO Episode Length for 10M episodes

In general, the main observation to note is that DQN did a lot worse than PPO. This is in part because DQN is unsuitable for extremely complex games like Street Fighter II. There are a lot of factors to consider such as whether to attack, jump, or do something else, and there are an almost infinite amount of game states. This means that there is not necessarily a best action that one can take and you will constantly see new never before seen situations while playing. This explains why we have the unstable curve in figure one. If the agent fails to win , DQN can collapse and fail to get any better. In this case, PPO is a much better choice. Since PPO aims to optimize the policy and is therefore non-deterministic in nature, it is a much better algorithm that we can pick. The second reason why we concluded that DQN was worse than PPO was because it took longer to train. It took 10 hours to train the DQN model for 500 episodes, whereas it took eight hours for 8 million episodes that PPO needed. In this case, it is rather clear that results wise and time wise, PPO is the better choice for beating Street Fighter II.

With regards to gameplay results, we saved the model of the PPO every 10000 time steps during training and found that the model trained at episode 5,180,000 performed the best overall. This model played the campaign and we recorded a video of it beating Guile, Ken, Chun Lee, Zangief, and Dhalsim in one campaign run. This performance is great and even out-performed models done by other tutorials on the internet that we saw. We learned a lot about reinforcement learning and PPO on SF2 from this tutorial https://www.youtube.com/watch?v=rzbFhu6So5U by Nicholas

Renotte. Much of ours and Nicholas' frame work is similar, but his model was only able to reach Dhalsim but never beat him after hyper parameter tuning for 2 entire weeks and training for days. One key difference, between our model and his was our reward function took into account damage taken by the agent and damage done to the enemy as well as explicitly rewarding wins and losses too, while his model only kept track of the agent's game score. Our model, in an effort to reduce computation time, also only read frames of size 64x64x1 compared to his larger feature space of 84x84x1. Our hyper parameters, and other coefficients were different as well. Thus, we were happy with our model beating Dhalsim given that we only were able to train it for 10 hours and only hyper parameter tuned for about 15 to 20 hours.

V. Future Work

With regards to the time constraints of the project we performed well, but corners cut due to time left several avenues for future improvements. One potential avenue for future work is to explore multiple models for fighting different characters. We use one model to try and beat the entire campaign which involves winning a best of three fight against every character in the games cast. This is not optimal because every character has their own strengths and weaknesses and the optimal way for beating Guile is different than the optimal way to beat Ken as Ryu. To this end, one could consider training different models to face each character and learn different strategies to beat them. This would actually be very similar to how real humans prepare for fighting game tournaments themselves, as humans intentionally try to learn the individual strategies they need to use for their character to beat each individual opponent character they face. This concept is called gathering match up knowledge in the fighting game community. Higher ranked players can often lose to lower ranked players in tournaments simply because their opponent plays a character they have little experience playing against, this is called being knowledge checked. Lowering the learning rate could also lead to an increase in performance at the cost of training the time. Our model's learning rates were always in the 1e-7 range, but lowering it even more could have made it learn better. Of course, we could not do this due to not having enough time to train the model sufficiently at that low of a learning rate. The other avenue for future work involves using the color data from the ROM instead of converting to grayscale. Increasing the resolution to 128x128 and including color is very likely a useful feature we can exploit to improve performance. Certain characters have projectile attack, which have different colors, but can have similar shapes. Thus, the lack of color channel information makes it hard for our agent to recognize when these particular attacks are coming. Better recognition of projectiles would help the agent learn how to beat a projectile heavy strategy, which was a weakness of our model in game play tests. Color channel data would also help the model learn what kind of ground attacks are coming at it too. The Last change we could implement is cropping out of the health bars, timer, and score from the frame data fed to the model. This is unnecessary data that provides no information and possibly distracts the model during training.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017; arXiv:1707.06347.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, 2013; arXiv:1312.5602.

[3] Yu-Jhe Li, Hsin-Yu Chang, Yu-Jing Lin, Po-Wei Wu and Yu-Chiang Frank Wang. Deep Reinforcement Learning for Playing 2.5D Fighting Games, 2018; arXiv:1805.02070.

[4] Yu-Jhe Li, Hsin-Yu Chang, Yu-Jing Lin, Po-Wei Wu and Yu-Chiang Frank Wang. Deep Reinforcement Learning for Playing 2.5D Fighting Games, 2018; arXiv:1805.02070.