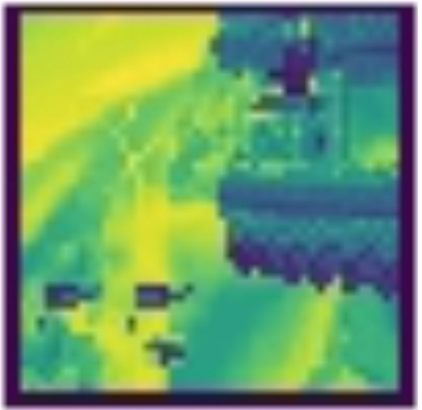
Playing Super Smash Bros 64 with RL

Varun Jana
Andrew Shen
Ruijie Mao

The challenge: Play against the in-game AI on the Dreamland stage, using only the screenshots as input. The goal is to knock the opponent off the stage through attacks.

Why this is an interesting problem: dynamic camera focus and zoom, sequential game, highly delayed rewards



Knock out! This is what our model sees. Each frame is also stacked with the past three frames. Otherwise the model cannot detect velocity.

Approach

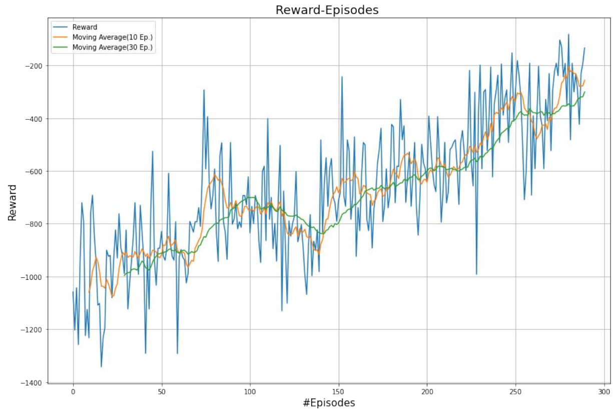
PPO (Actor Critic Algorithm):

Actor Policy directly outputs the action, maximizes the gain predicted by the Critic model. Poor performance due to sample inefficiency.

DQN:

On policy model, more sample efficient because the model stores all past transitions in a prioritized experience replay. Trained over 600,000 frames. Learnt to exploit the in game AI into falling off the map.

Total reward over training



Super Smash Bros. with CNNs

Varun Jana JVARUN@SEAS.UPENN.EDU
Ruijie Mao DREAMFLY@SEAS.UPENN.EDU
Andrew Shen ANDSHEN@SEAS.UPENN.EDU

Abstract

As of now, many new improvements in learning techniques have been developed in artificially intelligent game playing. This is especially true for more retro console games primarily developed for sub 32-bit/16-bit platforms such as the SNES, N64, Atari and Sega Genesis. In this paper, we will design and train various agents to play the classic crossover fighting game, Super Smash Bros (*SSB*) (for the N64 version) - where the winning objective is to knock opponents off the platform, while also causing damage. The main challenge in this exploration comes from the fact that *SSB* features a diverse array of move sets, multiple components to recognise and react to, especially with substantial noise in the data due to rapidly changing camera angles, zoom and in-game animations. We present our findings and an analysis of the different agents and variations.

1. Introduction

Super Smash Bros 64 is a fight game that revolves around knocking the opponent off the game stage. Dealing damage in SSB increases how far the opponent will be knocked by your attacks. So SSB players have to uniquely balance between dealing damage to the opponent player and delivering the right attack that will knock off the opponent. Movement and positioning is also very important in the game, most players have special moves to recover after being knocked into the air.

Our project focuses on designing a reinforcement learning agent for Super Smash Bros. Our model only uses screenshots of the game, similar to how an agent would play. Our aim was to show that a reinforcement learning agent can learn to play SSB at a decent level from scratch. This is a novel problem; previous attempts to solve the SSB game have used feature engineered RAM game states from the emulator (2017). We show our results from models using the Proximal Policy Optimization algorithm and Deep Q-Learning algorithm (DQN).

University of Pennsylvania, CIS 419/519 Course Project.
Copyright ©2020 by Varun Jana, Ruijie Mao, and Andrew Shen.

2. Background/Related Work

2.1. Q-Learning and MDP

Q Learning is one of the most common method of performing RL. Basically, how it works is that during each state, the model will calculate a Q function. Such Q function indicates the expected reward corresponding to each state and each action. In this case, we are able to update the Q function by dynamic programming using the formula below:

$$Q(S, a) = r + \gamma max Q(S', a')$$

Where r is the reward and S', a' is the state and action in the previous state. Therefore, the current Q value is obtained by the sum of the current reward (r) and the previous reward.

2.2. Deep Q-learning Modifications

In the Deep Q learning, instead of storing Q values for each state we approximate the Q value through a neural network. We optimize the network through minimizing the difference between estimated

$$Q(S_t, a_t)$$

and the target value

$$r + \gamma max Q(S'_t, a'_t)$$

DQN, in practice, struggles with instability issues and Q values are prone to explode to very high values. One method of stablizing training is to fix the target Q value by using a fixed old version of the agent (Mnih et al., 2013).

DQN also suffers from high correlation in the inputs when training during game-play. A common solution is to store past experiences in a experience replay buffer and train using mini-batches of past transitions, and this modification is implemented for our model as well.

2.3. Proximal Policy Optimization

PPO is an Actor Critic algorithm. It consists of two models, the actor takes in the current state of the game and chooses an action. That action is passed into the environment in exchange for the reward and next state. This is repeated for each episode. The sequence of rewards is used by the critic to estimate the gain for each step. The critic model minimizes

$$J^C(s_t) = (G_t - V(s_t)^2$$

Actor Critic algorithms calculate the advantage for each transition

$$A^t = G_t - V^\pi(s_t)$$

The actor model is then trained to maximize the probability of taking actions that have high advantage.

The PPO model is also extended by a clipping constraint that prevents the model from deviating too far from it's current policy. It does this by calculating a probability ratio

$$p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_\theta old(a_t|s_t)}$$

Then the clipped objective is maximized.

$$J^{CLIP}(\theta) = E^t[min(clip(p_t(\theta), 1\epsilon, 1+\epsilon)A^t, p_t(\theta)A^t)]$$

$$+\beta H(\pi_\theta(s_t))$$

Where H is the entropy of the action distribution, weighted by hyperparameter β.

3. Gameplay Framework

3.1. Emulator

Super Smash Bros 64 does not run on Atari, so we had to use an nintendo 64 emulator to train our agent. At the same time we wanted our emulator to be compatible with the OpenAI gym framework.

So we chose to use the Mupen64 emulator, which provided us with rendered pixel data as output and has a gym environment wrapper implementation available online (Zier, 2019). We were able to set up the emulator in a stable environment using docker containerization.

The emulator took continuous valued inputs for joystick controls. In order to reduce the complexity of our model, we discretized the joystick movements into 9 directions. Combined, the total action set consisted of the following moves:

Move	**Mapping Key**
Jump	C
Weak Attack	A
Strong Attack	Analog right + A
Shield	X
Grab enemy	Z + A
Special Attack 1	B
Special Attack 2	Analog up + B
Special Attack 3	Analog down + B
Move around	Analog stick

3.2. Data Collection

For training our reinforcement learning agents we recorded the frames from our agent's game-play, which we stored in an experience replay buffer. We recorded all of our games on the Dream Land stage. Because each Smash game is highly different, we wanted to focus on training our agent to play well on one stage.

4. Approach

4.1. Image Preprocessing

Decrease resolution to 84 x 84. Each imagine is in grayscale. Each frame is stacked together with the past three frames into 84 x 84 x 4 tensors.

4.1.1. Past Timesteps (Stacking)

Each frame is stacked together with the past three frames into 84 x 84 x 4 tensors. This allows computing object velocity since the algorithm has 'access' to the previous 4 frames (though this is still very difficult due to the velocity of the camera angle and rapidly changing camera zoom.)

4.2. Feature Generation

The architecture consists of 3 CNN layers and 1 fully connected layer to process the image data and generate around 1000 features. The feature space is very complex due to the enormous number of things that the agent may need to detect (eg. enemy movement, location on island, agent velocity). It also another fully connected layer to map the 1000 features into Q value predictions.

4.3. Deep Reinforcement Learning

We are using a fixed model to generate a Q values and use such value to generate our targets. Then we apply such target as our labels to calculate the Q value as our labels (fixed target) so that we run 50 iterations each time (propstep) to train the network and then copy the trained model to our target model.

Algorithm 1 DQN

```
Initialize $model$ and $target$ model.
for $i = 1$ to $maxstep$ do
    select $a_t = argmaxQ(\phi(S_t), a; \theta)$
    use the targetmodel to get the target Q value as our label based on our past experience
    calculate current Q value through $Q'(S, a) = r + \gamma maxQ(S', a')$
    use model to get the trained Q value
    minimize mean square error loss between Q' and Q
end for
```

Algorithm 2 PPO Algorithm

```
Input: policy parameters $\theta_0$, clipping threshold $\epsilon$
for $k = 0, 1, 2...$ do
    Collect trajectories $D_k$ using policy $\pi(\theta_k)$
    Compute advantage A = $\frac{\pi_{\theta k}(a_t|s_t)}{\pi_{\theta k-1} old(a_t|s_t)}$
    Maximize objective $J^{CLIP}(\theta) = E^t[min(clip(p_t(\theta), 1\epsilon, 1+\epsilon)A^t, p_t(\theta)A^t)]$
    By taking 4 steps of SGD (via Adam)
end for
```

4.3.1. PPO

We use the PPO (Policy Proximal Optimization) strategy to evaluate our data. That is, we use the neutral network to predict the likelihood of each action within each state and we increase the likelihood of the good actions (the actions with higher reward) and decrease likelihood of the bad actions.

5. Discussion and Results

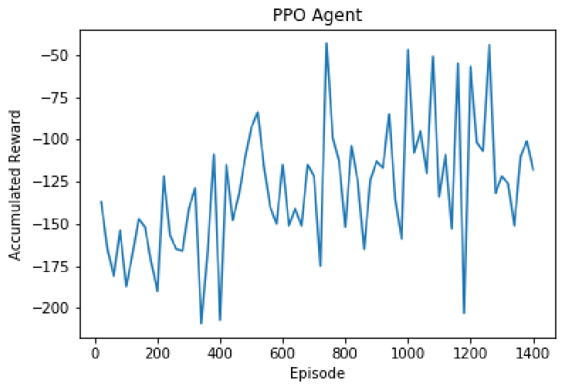
5.1. PPO

PPO is an online learning algorithm, which means that the optimizer uses actions from the current policy. PPO has produced state of the art results (Schulman et al., 2017) in various Atari games. However, PPO has mostly been applied to games with raw RAM states. Learning from visual data is a more complex task and requires heavy computation. In this domain PPO often performs less well than off policy methods like DQN that has higher sample efficiency.

Because PPO performs the update step at the end of episodes, we limited episodes to 300 time steps to increase update frequency. Unlike other methods, PPO is relatively insensitive to hyperparameter choices. So we only performed basic hyperparameter search around standard values. We evaluated hyperparameters after 30,000 frames.

We trained with the best hyperparameters, learning rate = 0.002, ϵ = 0.2, epochs=4 for 420,000 frames. The agent performed very erratically and barely better than random.

Figure 1. Accumulated reward



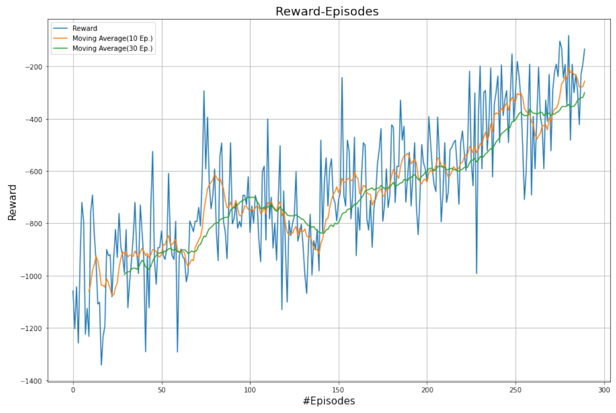


Figure 2. DQN with prioritized experience replay

5.2. DQN

The DQN algorithm allows off policy training on past transitions. This helped increase learning speed. Our biggest bottleneck in training was the emulator, which could only run on CPU. Because DQN allowed us to train on the same transition multiple times on GPU, we achieved better sample efficiency.

We implemented DQN with an experience replay and fixed target model as described in (Mnih et al., 2013). In addition, we implemented a prioritized experience replay (Schaul et al., 2015), which sampled important transitions more frequently. Without prioritized replay, we noticed that our agent did not learn to associate jumping at the edge of the map with negative rewards because the reward signal from falling off the edge of the map comes after a sequence of 20 frames. With prioritized replay, the agent better dealt with the delayed reward signals.

Table 1: Direct comparison of models (reward is averaged over last 4000 frames)

Algorithm	Accumulated Rew.	std
PPO	-534	51
DQN	-433	39
DQN + Prioritized Replay	-223	26

6. Conclusions and Future Work

Learning to play Super Smash Bros from scratch is a very challenging reinforcement learning problem. The combination of dynamic camera zoom, sequential game-play and delayed rewards makes SSB a good benchmark for visual only reinforcement learning algorithms.

Our project have shown that PPO and DQN with prioritized replay can learn the basics of the game. We believe that enhancing DQN with additional tricks like rainbow DQN (Hessel et al., 2017) will improve results further. Other future directions include novel zoom-invariant architectures and RNN to capture the sequential non MPD nature of the game.

References

Firoiu, Vlad, Whitney, William F., and Tenenbaum, Joshua B. Beating the world's best at super smash bros. with deep reinforcement learning, 2017.

Hessel, Matteo, Modayil, Joseph, van Hasselt, Hado, Schaul, Tom, Ostrovski, Georg, Dabney, Will, Horgan, Dan, Piot, Bilal, Azar, Mohammad, and Silver, David. Rainbow: Combining improvements in deep reinforcement learning, 2017.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning, 2013.

Schaul, Tom, Quan, John, Antonoglou, Ioannis, and Silver, David. Prioritized experience replay, 2015.

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

Zier, Brian. Gym mupen64plus. https://github.com/bzier/gym-mupen64plus/commits/master, 2019.