Reinforcement Learning Models for playing Super Mario Bros

Written by Anirudha Shastri, Varadh Kaushik
shastri.an@northeastern.edu
kaushik.var@northeastern.edu

Abstract

There has been vast increase of research conducted in the field of AI learning how to play basic games. This is due to the fact that more optimized algorithms are available and the computation power of computers has increased. In our project we have focused on doing a comparative study of the performance of a few Re-enforcement learning algorithms and their capacity to beat levels of the game Super Mario Bros. We have specifically taken into account 3 algorithms

- Deep Q-Learning (DQN)
- Double Deep Q-learning (DDQN)
- Proximal Policy optimization (PPO)

Index terms: Reinforcement Learning, Proximal Policy Optimization, Deep Q-Learning, Double Deep Q-Learning, Super Mario Bros

Introduction

With the growing popularity and power of Neural networks as powerful tools for AI and ML, it is still one which can be expensive to work with due to the hardware cost like that of GPUs to train these models on and the data acquisition costs involved. It is therefore important that one has an idea of what these resources are being used towards to get the best value for the time and money invested into training and implementing selected algorithms.

Keeping the above idea in mind in this project we have drawn comparisons between: 1)A generic RL policy update technique know as PPO: 2)Deep Neural network with Q-learning: 3)Deep Neural network with Double Q-Learning:

We decided to test these algorithms on the Super Mario Bros game as there is python support already present for this game. Being a relatively small and simple game the number of actions performed and computation overhead is less as well. This made it possible for us to experiment with the hyper parameters and also train our own models and test their performance against other algorithms and other models of their own algorithm.

Since all the three algorithms varied slightly we tried to keep the training parameters as standard as possible to make the comparison reliable.

Background

Super Smash Bros Environment

The goal of Super Smash Bros is to reach the end of each level while avoiding drops, enemies and enemy attacks. We have used OpenAI Gym environment for Super Mario Bros. [SMB] & Super Mario Bros. 2 (Lost Levels) [SMB2] on The Nintendo Entertainment System (NES) using the nes-py emulator.

Environments We are allowed 3 attempts (lives) to make it through the 32 stages in the game. The environments only send reward-able game-play frames to agents; No cut-scenes, loading screens, etc. are sent from the NES emulator to an agent nor can an agent perform actions during these instances. If a cut-scene is not able to be skipped by hacking the NES's RAM, the environment will lock the Python process until the emulator is ready for the next action.

Steps Information about the rewards and info returned by the 'step' method.

Reward Function

The reward function assumes the objective of the game is to move as far right as possible (increase the agent's x value), as fast as possible, without dying. To model this game, three separate variables compose the reward:

1. v: the difference in agent x values between states in this case this is instantaneous velocity for the given step
   - v = x1 - x0
     - x0 is the x position before the step
     - x1 is the x position after the step
   - moving right: v ¿ 0
   - moving left: v ¡ 0
   - not moving: v = 0
2. c: the difference in the game clock between frames
   - the penalty prevents the agent from standing still
   - c = c0 - c1
     - c0 is the clock reading before the step
     - c1 is the clock reading after the step

- no clock tick: c = 0
- clock tick: c ¡ 0

3. d: a death penalty that penalizes the agent for dying in a state
   - this penalty encourages the agent to avoid death
   - alive: d = 0
   - dead: d = -15

   r = v + c + d

The reward is clipped into the range (-15,15).

Environment Setup

All the training was conducted on a local machine having the following specification,

- CPU: Intel i7-11700
- GPU: Nvidia GeForce RTX-3080 10GB
- RAM: 16GB
- OS: Windows 10

The implementation of PPO and DQN(Deep Q-Network) were done using the help of the stable_baselines library in python.
This provided us with a basic function of PPO and DQN to work with. For Double Deep Q-Network we had to use a stand-alone implementation of DDQN. The models were trained and experimented with different hyper Parameters.

On training the model all three algorithms were run on 100,000 time-steps of the game and the avg score over every 1000 steps was calculated. Higher the score implied more successful the agent was in beating the game.

Related Work

In our implementation of these algorithms we have used only v0 of the Super Mario Bros environment, there are a total of 4 versions, each with different level of details in them. Eg: v2 is pixelated and v3 has the environment converted to rectangles. The environment has different movement sets, namely RIGHT_ONLY, SIMPLE_MOVEMENT and COMPLEX_MOVEMENT.
We have used SIMPLE_MOVEMENT as COMPLEX_MOVEMENT would require a lot more training time.

Approach

We used the gym Super Mario bros environment and used SIMPLE_MOVEMENTS only. This environment lets us run through the game frame by frame. We trained each model on frame by frame for a desired number of time steps. This means at each time step we go through one frame, the model tries and learns all information at that stage, then the next. This way over time after going through the same set of frames and action, as it trains it will have a better understanding of which move at which stage or condition will give the best outcome. This is the basic idea behind how RL works.

The process of training a RL model isn't as straightforward as there is no ground truth based on which the loss function or gradient decent can take place, therefore there are policies that help train the model. One such policy which we evaluated in this project is the PPO. In PPO the computation overhead is minimum and easy to work with. It computes an update at each step that minimizes the cost function and at the same time the deviation from the previous policy is very small.

The second RL algorithm we used is Q-Learning. This a model-Free RL algorithm, this means that it doesn't depend on the transition probability distribution in Markov process. This means the training of the model doesn't depend on any hidden variables or even on the model of the environment presented to it.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{temporal difference}} \underbrace{- Q(s_t, a_t)}_{\text{current value}}$$

$$\text{new value (temporal difference target)}$$

Figure 1: Q-Learning equation for Deep Q-Networks

The last algorithm used is a Double Q-learning. This helps making the Q-learning algorithm more robust and less susceptible to noise. Here two Q-learning algorithms work together to better tune the parameters by having a feedback loop of updated Q values.

$$Q^A_{t+1}(s_t, a_t) = Q^A_t(s_t, a_t) + \alpha_t(s_t, a_t) \left(r_t + \gamma Q^B_t \left(s_{t+1}, \arg\max_a Q^A_t(s_{t+1}, a) \right) - Q^A_t(s_t, a_t) \right)$$

Figure 2: Eqn 1 for Double Deep Q-Networks

$$Q^B_{t+1}(s_t, a_t) = Q^B_t(s_t, a_t) + \alpha_t(s_t, a_t) \left(r_t + \gamma Q^A_t \left(s_{t+1}, \arg\max_a Q^B_t(s_{t-1}, a) \right) - Q^B_t(s_t, a_t) \right).$$

Figure 3: Eqn 2 for Double Deep Q-Networks

Experiments and results

Experiments and results for Proximal Policy Optimization:
During the training process the model was saved for every 10,000th model generated. This way we could look through different models and their losses and see which one performed better on the game. The below graph is plotted based on a model built at the 2,500,000 time-steps. This graph shows us that the maximum average score it was able to achieve was around 130. This value was even lower on model that were trained for lower number of times steps.

PPO Result: The training of this model took close to 6 hours even after using PyTorch GPU compute. What was observed is that as the number of time steps where increased and the learning rate is reduced the model Is able to better play the game.

Experiments and results for Deep Q-Networks:
This model was trained for 3,000,000 time-steps as well with a learning rate of 0.000001. For this training process as well, the model was saved every 10,000 time-steps. The below three graphs show the performance on the Mario games on

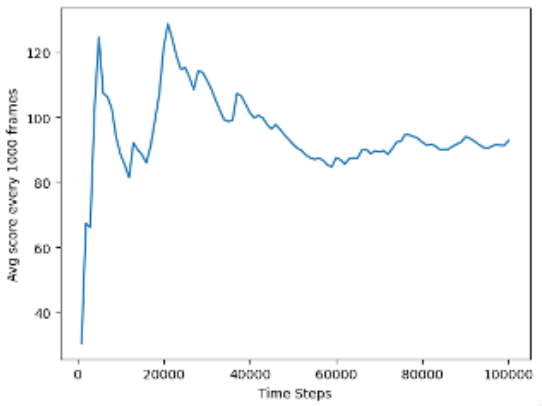

Figure 4: Graph for PPO trained for 3,000,000 time-steps with a learning rate of 0.000001

3 different models, as the number of time steps increased the chance of having a higher score increases as well.

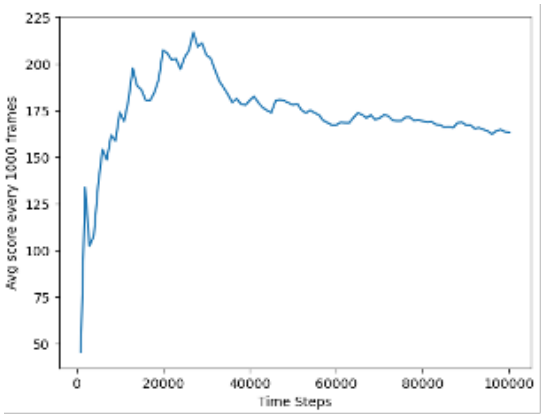

Figure 5: Graph for DQN trained for 1,050,000 time-steps with a learning rate of 0.000001

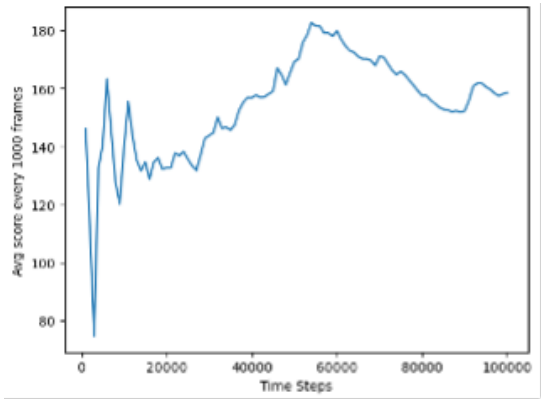

Figure 6: Graph for DQN trained for 1,740,000 time-steps with a learning rate of 0.000001

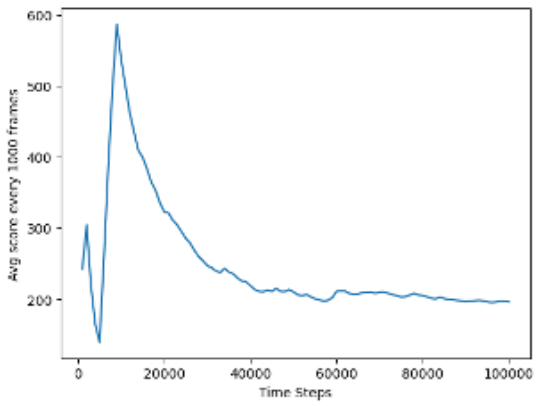

Figure 7: Graph for DQN trained for 2,560,000 time-steps with a learning rate of 0.000001

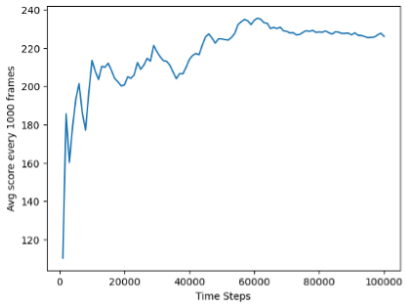

Figure 8: Graph for DDQN trained for 2000 time-steps with a learning rate of 0.000001

DQN Result: The training of this model took close to 6 hours as while using PyTorch GPU compute. As seen in the graph above it can happen that the model has run for higher number of iterations but might perform slightly worse than the one trained for less. This is because the Q-learning model is highly susceptible to noise present in the data set. But the more one trains it will be more capable to get higher scores as showed by the model which has been trained for 2560000 iterations has a spike of scores close to 600. Since it takes about the same time to train as PPO this might be a better option between the two.

Experiments and results for Double Deep Q-Networks:
DDQN: This model did not use the stable_baselines library. We used a stand-alone implementation of the Deep Double Q-network. There is a small change in how the model is trained, instead of number of time steps here we trained for number of episodes. Each episode is when the game starts to end either cause Mario died, the episode timed out or won the level. The training of this model was done for 2000 episodes with a learning rate of 0.00025 and batch size of 128 samples. As seen in the graph below the average scores of this network is much higher than PPO and DQN.

DDQN Result: The training of this model for 2000 episodes took close to 3hrs. Half the time taken by the DQN

and PPO. The average performance reflected is much better than that of both PPO and DQN. If trained for the same amount of time as PPO and DQN then this model will definitely perform much better than both of them by a significant margin.

Conclusion

From the above experiments, we can conclude that Double Deep Q-Networks is the most powerful algorithm when compared to Deep Q-Networks and Proximal Policy Optimization. Training on better hardware for longer would give us the actual capabilities of all these algorithms.

Author contributions

The implementation of Proximal Policy Optimization and Deep Q-Network was done by Anirudha Shashtri. The implementation of Double Deep Q-Network was done by Varadh Kaushik. The presentations/ reports/ remaining code had equal contributions from both authors.

Bibliography

Beating the world's best at Super Smash Bros. with deep reinforcement learning
Firoiu, Vlad and Whitney, William F and Tenenbaum, Joshua B *arXiv preprint* 1702.06230

OpenAI Gym
Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba A. eds. 2016. *arXiv* 1606.01540