Super Mario Bros DQN

Virak Pond-Tor
William Sun

April 2021

Abstract

In single-player video-games most players pay no attention to the time it takes to get through and simply focus on the plot and/or challenges presented to them. For a select set of games, however, there exists a dedicated community of players who aim to complete the game as fast as possible and these players are known as speedrunners. Players who speedrun will use anything within their power to shave off a couple of seconds, including glitches, exploits, and neat tricks that can be near impossible for humans to perform. The problem with the speedrunning process is that new optimizations often depend on the trial and error of current speedrunners and as a result, the rate of improvement is directly tied to human creativity. As of today, there exist speedrunning bots called TASBots which follow a sequence of hard-coded inputs, but since this sequence comes from humans, we can't be certain that more optimal routes do not exist outside of the community's knowledge. Our plan is to create a speedrunning bot for the original Super Mario Bros game using a DQN algorithm attuned to focus on the time it takes to complete a level. This will be achieved by giving the DQN a custom reward function that highlights time while also allowing for the completion of the level. In addition, the underlying algorithm will be a temporal convolutional network, allowing for the algorithm to understand both spatial position and the importance of frames as a time series.

Introduction

Reinforcement learning is an important field of machine learning that has been a crucial area of research in fields such as robotics and game AI development, as reinforcement learning is the standard for learning how to perform a task in an arbitrary action space and environment. While most current reinforcement learning algorithms and improvements are focused on how to get the training time to decrease, how to train algorithms that learn from very limited environment information, or how to get an algorithm to complete increasingly complex tasks, one area that has not been explored as much is how to get an algorithm to improve at completing a task quickly. While this is not a top priority for most fields of robotics, as current reinforcement learning-based agents perform tasks in a reasonable amount of time steps, this speed task has specific importance in the realm of video games, where there are dedicated bases of people who spend time trying to figure out how to beat a game as fast as possible.

This application of reinforcement learning is mostly unresearched; we could find few sources of people attempting to apply reinforcement learning to this task, and no papers or scholarly articles on it. The sources we did manage to find on this all talked about completing this task on a game with a small action space or with very limited ways to interact with the world. However, the tasks for which a reinforcement learning algorithm would help improve the most are tasks with large discrete action spaces, i.e. games with more freedom of movement, such as 2d platformers or racing games. Thus, in this paper we explore different ways to create an algorithm that will optimize more for speed of completion than number of or consistency of completion on the game Super Mario Brothers for the Nintendo NES.

Our intended approach is to implement our own algorithm as well as modify an existing reinforcement learning algorithm that is a reinforcement learning standard: the Deep-Q Network, shortened as DQN. DQNs are algorithms that train an agent to complete a task by working through a task step by step, using a neural network algorithm to predict the best next action to take and giving feedback at each step dependent on a reward policy. We intend to provide this DQN with a custom reward policy that rewards the agent heavily for the speed at which it completes a task, as well as testing a number of neural network structures to see which is the optimal neural network structure to complete this task as fast as possible. This will start with creating a DQN on a task with a much smaller action space, namely CartPole, and then scaling it up to the original Super Mario Brothers.

We created a baseline algorithm for the CartPole task and then applied a similar model to the Super Mario Bros environment. In the CartPole environment, we can test the effectiveness of this algorithm on the task by looking at a graph of its performance, in this case how many steps it takes before the pole tips past 20 degrees from the vertical, and see if the agent improves by keeping the pole balanced for a

larger number of steps. Similarly, in the Super Mario Bros environment we can generate a graph that displays the performance of each run, where a run consists of a single attempt by Mario to reach the flag and performance during is based on how close Mario can get to the flag.

Although much of our project revolves around the effect of a custom reward function using DQN, a large portion of it also involves familiarizing ourselves with common deep learning techniques. We are aware that applying learning algorithms to Super Mario Bros has already been covered at length, but we learned a great deal about deep learning algorithms and hopefully brought an interesting speedrunning perspective.

Related Work

There are a number of resources online related to reinforcement learning and also learning algorithms specifically for Super Mario Bros. The first video we took interest in was called "MarI/O - Machine Learning for Video Games" by SethBling [7], which used the NEAT [9] genetic algorithm to learn Super Mario World. NEAT is a genetic algorithm which uses evolving neural networks and while we understood that a genetic algorithm such as NEAT could be applied to Super Mario Bros, we wanted an algorithm that would align better with our speedrunning goals. The next resource we consulted was a blog post about deep learning in Super Mario Bros called "Play Super Mario Bros with a Double Deep Q-Network" by Andrew Grebenisan [5], which provided an overview of reinforcement learning and an overview of the algorithm's components. This includes the neural network module the author created, their method of storing results, and their method of updating weights in the neural network, which were all fairly standard in reinforcement learning algorithms. The author also introduced us to the Super Mario Bros OpenAI gym environment by Christian Kauten [6] and since this environment provides the inputs/outputs, a method for stepping through states, and a reward function, it saves us a lot of trouble.

Experimentation with the Super Mario Bros OpenAI gym environment eventually led us to the CartPole environment where we could test our model with an easier problem. Since the CartPole problem is a rather standard and popular problem in reinforced learning, we had no problem finding more resources to base our own model off of. Much of our model is based off the article "Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning)" by Greg Surma [10] which outlines the process of creating a network and using the cartpole environment. Note that the implementation outlined in Surma's article uses the Keras framework, whereas we want to use the more familiar PyTorch.

Technical Approach

A. State Observations

Our approach for this project consisted of three main parts: a CartPole DQN implementation, a Super Mario Bros DQN implementation, and a custom reward function tested on an already existing model. In the CartPole environment each state can be described with four observations, namely the position of the cart, velocity of the cart, angle of the pole, and rotation rate of the pole. As a result, the input for our model consists of these same observation, which will be denoted as X, V, θ, and ω respectively. The output for our model will be a Q-value that indicates whether the cart moves left or moves right to try and balance the pole.

B. Model and Training

The model itself consists of one linear layer with 4 nodes for the 4 corresponding observations, one hidden linear layer with 24 nodes, and lastly another linear layer of 2 nodes for the 2 possible actions that can be taken in a given state. We decided to keep a relatively small model with fewer nodes and layers to decrease the amount of time spent training, but at the cost of granularity, meaning changes in the model will be more drastic. We also apply ReLu [1], the rectified linear action function, between the first and second. ReLu is a piecewise function that leaves positive input unchanged, while mapping everything else to 0 and in many cases, results in models that train faster and perform better than models without it.

To train our model we must also define the loss function, labels, and optimizer which are all used to update the Q-values. The loss function we use is called the SmoothL1Loss function which exaggerates the loss when the difference between the out-

put and the ground truth is below a certain threshold and dampens the effect when the difference is above that same threshold, resulting in resiliance to outliers. When training a model, people use some sort of baseline to determine how accurately their model can make predictions, often referered to as labels or ground truth. They apply to the loss function to their model's output alongside the ground truth, which gives a measure of how accurate the model's output. In an evironment such as the CartPole task or Super Mario Bros the ground truth isn't provided, and so we use the Bellman equation to evaluate the "true" Q-value that the model should try to output.

$$Q(s,a) = r + \gamma(max(Q(s',a))) \tag{1}$$

In the above equation $Q(s,a)$ represents the Q-value for taking the action a in state s, r represents the reward of state s, γ is the discount factor, and $max(Q(s',a))$ is the maximum Q-value that can be attained in the next state. The discount factor is used to discount the worth of Q-values many time steps in the future, causing immediate Q-values to hold more weight.

As for the optimizer, we decided to use the Adam algorithm [2], a close relative of stochastic gradient descent, with a learning rate of 0.001. We also tried other optimizers for our implementation, including an SGD optimizer [8] which yielded similar results to the Adam optimizer, and so we arbitrarily chose to move forward with the Adam optimizer.

C. Exploration

Another aspect that influences our model is our method of exploration. We are using an epsilon greedy Q learning model, and thus tuning the hyperparameters for epsilon decay is crucial to get the algorithm working properly. The epsilon value describes the probability that we choose to take a random action from the domain as opposed to an action that the model tells us to take. Since we want the model to observe the result of many random actions at first, we start the training loop with an epsilon value of 0.9 and decrease this value to 0.05 over the course of our training session.

D. Super Mario Bros

The DQN implementation in Super Mario Bros closely resembles that of the implementation in CartPole with most major changes relating to the input and output of the model. In Super Mario Bros, each state is defined by a 2D array, which often contains more information than necessary. To improve training times, we preprocess the environment through SkipFrame [4], GrayScaleObservation [4], and ResizeObservation [4], while maintaining the important information from each state. After preprocessing, we flatten the 2D state array into a 1D array and pass it into an adjusted model with nodes that correspond to the input and output of Super Mario Bros, rather than CartPole. The output of the model in the Super Mario Bros correspond to the 256 different actions Mario can take, including moving right, moving left, crouching, and jumping.

(Grayscale image of Super Mario Bros level)

We also worked with modifying an existing implementation of a Super Mario Bros DQN. The reason we made this decision is that a pre-existing model would be more reliable and more bug-tested than our own model, and this would ensure that the improvements made are due to our own changes to the model rather than inconsistencies with the model itself. In addition, we could then use pre-trained weights to both explore the differences between our changes to the model and the original model and speed up training times. We used specifically used roclark's double DQN implementation on Super Mario Bros [3].

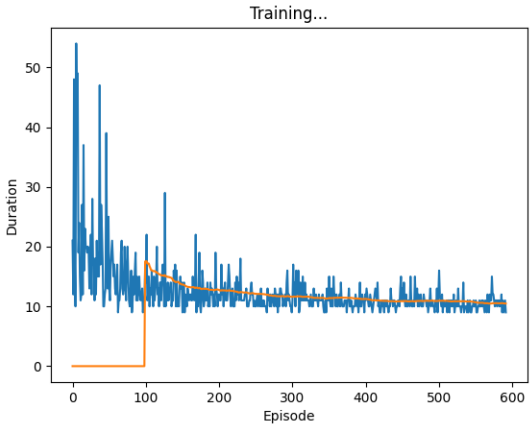
We looked into modifying 3 parts of the model: the reward function, the underlying model, and the exploration values. With the reward function, the goal was to modify the reward function to put a larger emphasis on the time it took to complete the level. The reward functions tested were a reward added every time Mario completed a level based on how quickly he completed it, a negative reward added every time an in-game second passed, and a reward added every step based on how long Mario had been in the level. In addition, modifying the underlying model to a temporal convolutional network (TCN) as opposed to a convolutional network (CNN) was tested, and various different starting exploration values were looked into as well. In particular, we looked at how modifying the epsilon value, which determines how often the bot took an exploration step as compared to a non-exploration step, to see if this had any effect on the model.

Evaluation

Although difficulties with our own implementation of DQN in Super Mario Bros prevented us from integrating it with the custom reward function, our success with the reward function demonstrated that reinforced learning could be an interesting prospect for the speedrunning community.

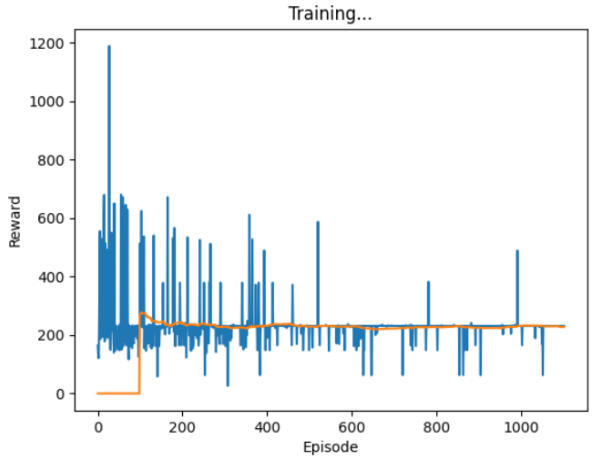
A. Our Implementations

Seeing as the Super Mario Bros DQN implementation was based off of our CartPole DQN, it makes sense that the following results are very similar. In CartPole, we measured success through the number of time steps the cart could keep the pole balanced for, while in Super Mario Bros, we measured success through how close Mario could get to the flag in a single life. When we ran our model in the CartPole environment, the cart initially had a relatively high success score, suggesting that random inputs can have some degree of success in this task. As the model continued to train and the epsilon value decreased, we noticed that the duration of each episode converged to one value at around 10-12 time steps. At this point in the training loop the epsilon value is low enough such that most of the cart's actions are provided by the model, indicating that our model was learning the incorrect behavior.



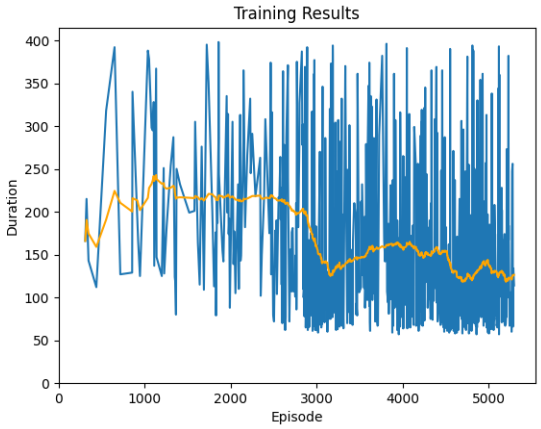
If we look at the figure above we can observe this correlation more clearly. The blue line represents the scatter plot generated when the duration of each episode is plotted and those points are connected, while the orange line represents the average duration of the last 100 episodes. During the early episodes, the pole can remain standing for over 50 time steps, but as the number of episodes increases, such long durations disappear completely. This downward trend of durations is what leads us to believe that something odd is occurring when weights are updated in the network. We still have yet to find the source of this error and we imagine that it would require many more hours of tuning our parameters to resolve. However, the convergence of the durations is a promising sign that with adjustments to our method of updating weights, our model can learn the correct behavior.

For the Super Mario Bros environment we had very similar results. The initial random actions would always show some degree of success, but eventually converge to a relatively low point.

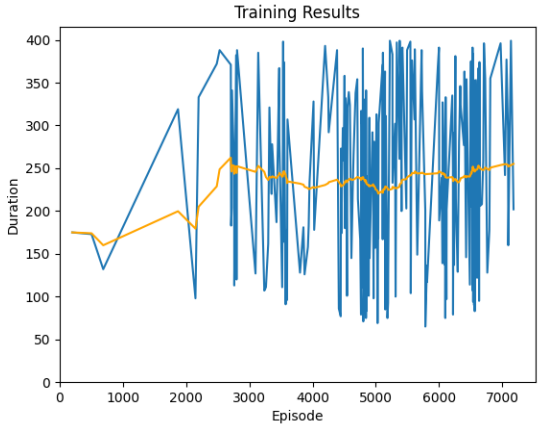


In the above figure, we can see that the orange line representing the average stayed at the same value of around 220-230 and deviated very little from that value. When we went to check the environment through the emulator, we noticed that Mario would continuously collide with a Goomba in the same spot, which is again indicative of incorrect learning. In the Super Mario bros environment, this behavior was much more prominent since we had a visual that was easier to understand, as opposed to the the CartPole environment where small adjustments are harder to notice. Overall, our own DQN implementations yielded poor results, but promised a future where we can hopefully correct the behavior of our model.

B. Custom Reward Function



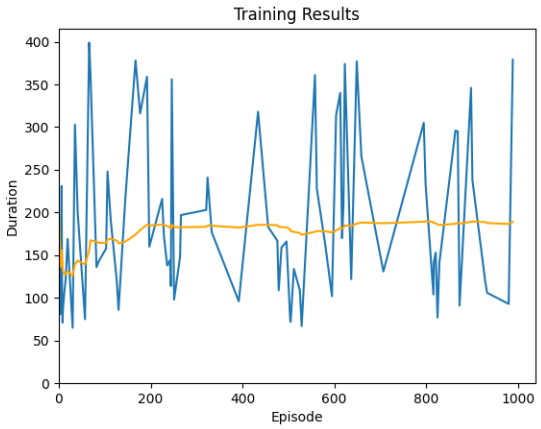
The above figure shows the amount of time it took Mario to complete a level running the default model. As shown in the figure above, the initial test on the default model showed little learning of how to complete the level for about 3000 episodes of training, at which point the model started to learn how to complete the level more and more often. However, the average time (as indicated by the orange line, which tracked the average time it took to complete the level for the past 100 episodes) actually increased over time, settling to around 250 in-game seconds out of a max of 400 possible.



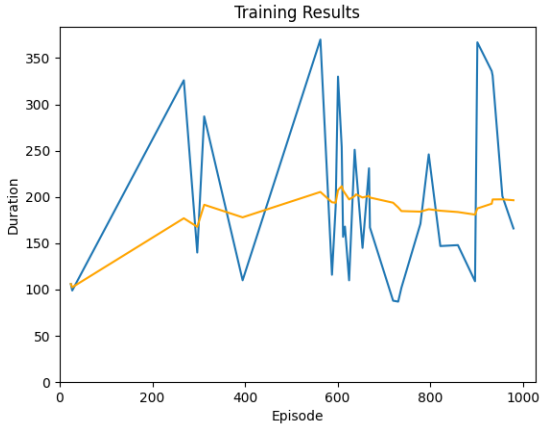
This figure, however, shows the amount of time it took Mario to complete a level using the custom reward function that added a negative reward every time an in-game second passed. As is seen, the model started to learn how to complete the level a lot earlier on, finding some early successes as early as episode 500, and started to reach a decent success rate by episode 2000. By episode 3000, the success rate of Mario completing a level was significantly higher in the custom reward model than the default model. In addition, we see that the model steadily improves in average time, as by the end of training, the model reached an average time of around 100 in-game seconds, an improvement of over 50% as compared to the default model.

However, while the figures are not shown here, the other custom rewards and model modifications did not work quite as well. Using a custom reward that was added at the end of every level and using a custom reward that was added based on the current time of the level, the DQN found that it would receive a higher reward dying early rather than completing

the level quickly, and this proved a major roadblock for it completing the level, let alone completing the level as quickly. In addition, using a TCN proved to slow down training significantly, as the model was significantly deeper, and ultimately it was not able to complete the level frequently, only completing the level approximately once every 500-1000 episodes.



The effects of using the pre-trained weights on the custom model were looked into as well. The above figure shows the completion times of the bot on the level when using the default weights as a starting point for training on the custom reward model, with a starting epsilon of 0.2. We see that in fact, Mario does not see much success in completing the level, actually completing the level less often as time went on. In addition, the time that the average run converged to was around 100 in-game seconds, which is significantly slower than the model trained from a starting epsilon of 1.0 with no transfer learning. This is indicative that transfer learning would not be very effective as the weights for the default model were not similar to the weights of the model with the custom reward. However, it is worth noting that the algorithm did complete the level faster than the default model, which is a sign that the weights were somewhat changed.



Another run was attempted with the transfer learning from the default model's weights, this time with a starting epsilon of 0.9, to see if the bot simply needed more exploration. What we see is that the bot performed even worse, completing the level significantly less often over the course of 1000 episodes and having a high average time of around 200 in-game seconds. This adds to the idea that the custom reward's weights are significantly different from the default model's reward, and thus the custom reward model struggled when starting at the default model's weights.s

Conclusion

In our project we observed a fair amount of success with adjusting a reward function to account for time, but fell short in our own implementation of DQN. While it would have been ideal to create our own implementation and integrate that with the reward function, we still achieved many of the goals we set out for ourselves, which includes learning more about machine learning and demonstrating how it could be viable for speedrunning. Since machine learning in Super Mario Bros is nothing new, most of our work involved experimenting with experimenting with already implemented models/algorithms and expanding our own knowledge. We read through numerous blog posts, articles, and GitHub repos highlighting the DQN architecture, created an implementation in the CartPole environment, adapted this implementation for Super Mario Bros, and finally observed the effect of an altered reward function to account for

time. As of right now, our model appears to perform worse as the epsilon value decreases and Mario depends more on actions given by the model, but the existance of this pattern in the first place signifies that the weights of the network are being updated in response to the environment. Although these aren't the implementation results we were hoping for, we still observed success for the reward function portion.

Going forward we hope to solve the issues with our current model and eventually integrate the model with the reward function. We suspect that most of our DQN problems lie in how we evaluate the model's accuracy. The next area to look at beyond the loss function is the structure of the actual model. There are an endless amount of combinations we could use arrive at a suffiecient model, but this would require much more research and much more time as well. Additionally, we could explore machine learning methods other than DQN, such as double DQN which may solve our issues with a more advanced algorithm.

After integrating our model with the reward function, we would of course try to optimize this reward function as much as possible. For our machine learning implementation to provide value to the Super Mario Bros community of speedrunners, it will have to be fast enough to at least compete with the current strategies in speedrunning. If our program were to reach this level of success we would love to see how machine learning fares against even the best speedrunners in the world.

References

[1] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. Machine Learning Mastery, Apr. 2019. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[2] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Machine Learning Mastery, July 2017. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/#:~:text=Adam%20is%20a%20replacement%20optimization.

[3] Robert Clark. *roclark/super-mario-bros-dqn*. GitHub, Mar. 2021. URL: https://github.com/roclark/super-mario-bros-dqn (visited on 04/23/2021).

[4] Yuansong Feng et al. *Train a Mario-playing RL Agent — PyTorch Tutorials 1.8.1+cu102 documentation*. pytorch.org, Dec. 2020. URL: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html (visited on 04/23/2021).

[5] Andrew Grebenisan. *Building a Deep Q-Network to Play Super Mario Bros*. Paperspace Blog, Sept. 2020. URL: https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/ (visited on 04/02/2021).

[6] Christian Kauten. *Kautenja/gym-super-mario-bros*. GitHub, May 2020. URL: https://github.com/Kautenja/gym-super-mario-bros.

[7] . *MarI/O - Machine Learning for Video Games*. www.youtube.com, June 2015. URL: https://www.youtube.com/watch?v=qv6UVOQ0F44&t=140s (visited on 04/02/2021).

[8] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. Sebastian Ruder, Jan. 2016. URL: https://ruder.io/optimizing-gradient-descent/.

[9] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10 (June 2002), pp. 99–127. DOI: 10.1162/106365602320169811.

[10] Greg Surma. *Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning)*. Medium, Nov. 2019. URL: https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288 (visited on 04/02/2021).