Project Part 1
736 Neural Networks and Machine Learning
Nic Manoogian
Robert Bond III
Zach Lauzon

Problem Overview

Nintendo's *Super Mario Bros.* took the world by storm in 1985; it was one of the first video games of its genre that was appealing to the masses. Before this, most games focused on high scores and endless play. Mario is not about high scores, it's about beating the game and saving the Princess. Over the next few weeks, we will train a neural network to play *Super Mario Bros.* on an emulated Nintendo Entertainment System (NES).

Video games are an interesting application for machine learning algorithms due to their functional nature. Most machine learning systems learn from data that must first be collected. This raises some concerns about how representative this training data needs to be. Video games endlessly generate and output this data. In a sense, a video game is just a very complicated nonlinear function with pseudo-randomness sprinkled in.

Why did we spend time teaching computers to play chess? Why are we so interested in building a machine that passes the Turing Test? The solutions to these problems don't directly improve humanity but they are significant advancements in the field of artificial intelligence. They demonstrate the depth of human intelligence and the novel application of new technologies. Moreover, these efforts often reveal new methods for solving other problems.

In contrast to chess, video games are somewhat close to the real world. Is driving a simulated car in *Gran Turismo* very different from driving a real one? In teaching a neural network to play Mario effectively, we hope to uncover new methods of processing data and training our artificial neural network.

Prior Literature

Human-level Control through Deep Reinforcement Learning (DeepMind)

In this 2015 paper, Mnih and colleagues explore a machine learning technique that has been applied to video game play. The authors use deep learning to process raw pixel input and play Atari 2600 games, just as a human would.

Due to the easy-to-score nature of games, these problems lend themselves well to reinforcement learning. In this type of system, the agent selects an action from the set of legal actions that it could take at that point in time. That action, when executed by the emulator, changes the state of the game. For this reason, small changes in the order of executed actions can impact the reinforcement significantly. Thus, the goal of the agent is to interact with the emulator by selecting the action that will maximize future rewards (Mnih et al., 2015).

The authors note that this can be quite difficult due to the nature of these games. Often, actions that you take in one frame may not yield a direct reward in the next but are critical to overall success. To improve this association, the authors use a technique known as *experience replay* where the agent's experiences are stored at each time-step and pooled into a *replay memory*. Randomly-sampled experiences from memory are fed into a convolutional neural network as part of the training process. This specific network consists of a $84 \times 84 \times 4$ preprocessed inputs followed by three convolutional layers and two fully-connected layers with a single output for each action that the agent could take.

The full algorithm, called *deep Q-learning*, was very successful at learning to play Atari games. In almost all of the 49 games tested, the agent was able to perform better than or comparable to a human player. In the best cases, the agent was able to learn a long-term strategy. For example, in *Breakout*, the agent learned to dig a tunnel around the side of the wall which traps the ball and destroys many blocks in a short amount of time.

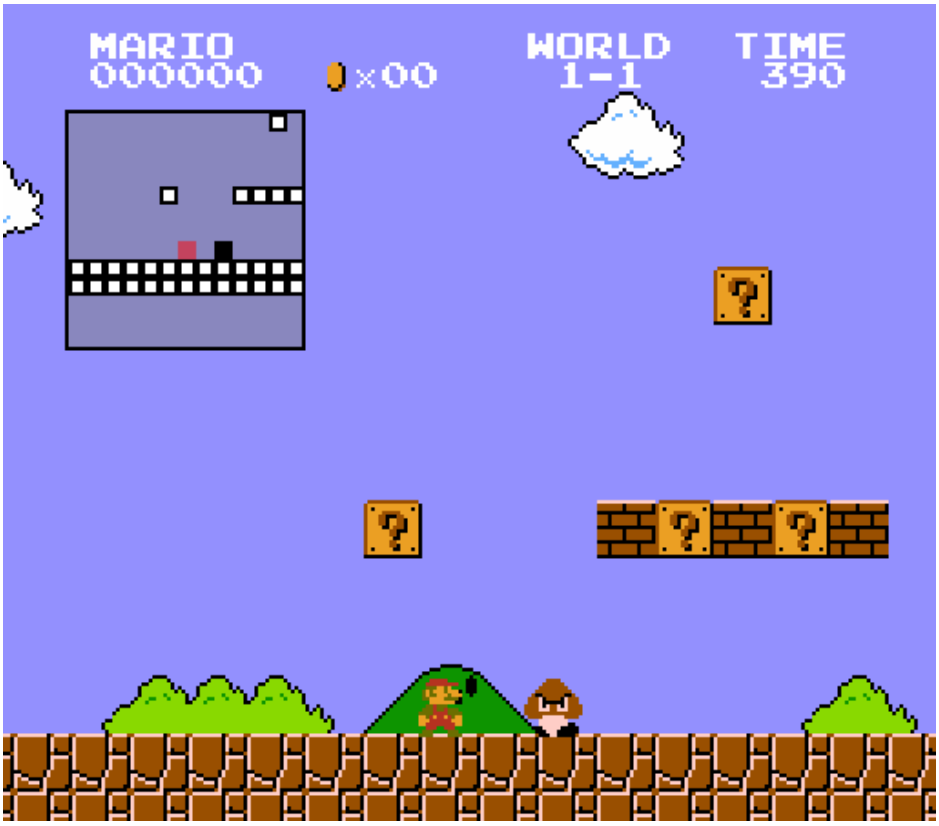
The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel . . . after that it gets a little tricky

Murphy takes on a novel approach to intelligently playing video games. Rather than observe pixel data, Murphy designs two algorithms: one to identify regions of memory that are lexicographically increasing (string order) for a given NES game, and another to produce controller outputs that maximize the values in these regions. These regions generally correspond with important values, such as health, lives, and forward progress. The playing algorithm is allowed to explore multiple futures in an attempt to avoid local minima (short-term planning). Murphy tested the algorithm on various NES games with mixed results: it was able to beat the first level of *Super Mario Bros.*, but does not perform well on games such as *Tetris* that require longer-term planning (Murphy VII, 2013). Murphy's approach may be considered even more generic than DeepMind's: while DeepMind uses raw pixels, it still tells the network the current score (i.e. how well the network is doing). Murphy's approach automatically determines its own fitness function for any NES game.

MarI/O - Machine Learning for Video Games

While not a peer-reviewed research publication, MarI/O (SethBling, 2015) does make a significant contribution in applying machine learning to video games. Rather than using raw pixels like DeepMind, MarI/O does further feature selection. Since we will be using an approach similar to MarI/O, we will detail this feature selection process.

Consider the beginning of the first Mario game, world 1-1. MarI/O will transform the frame into a feature matrix. Each neutral tile will be represented by a 0, each solid tile (ones that Mario can walk on) as a 1, and each enemy tile as a -1. Thus, the following frame is represented by the box in the upper-left corner:



These features are given as inputs to a neural network that is evolved via NEAT (Stanley & Miikkulainen, 2002).

As SethBling's research came to a halt, his results were cataloged through videos on his YouTube channel. He developed this technique for a Super Nintendo game called *Super Mario World*, which is similar to *Super Mario Bros.* but has many more outputs and other game complexities to take into account. He attempted to apply the same technique to *Super Mario Bros.* but could only get the network to complete the first level. Conversely, for *Super Mario World*, SethBling's script was able to beat the first and second level, as well another level further in the game. The network got stuck on certain levels due to complexities of the game that required more long-term planning (i.e. killing a specific enemy and using that enemy's platform to jump to a higher section of the level).

Current Solutions

DeepMind's system learned to play a broad array of Atari games using the deep Q-learning algorithm and pixel data. SethBling's system learned to play *Super Mario Bros.* using the NEAT algorithm and feature-processed, emulated RAM. We feel that the best solution for playing Mario games would be a combination of these two systems: rather than using pixel data as inputs to a deep Q-learning network, perform feature-processing in a manner similar to MarI/O. This level of downsampling should decrease training time significantly by bringing the NES's 254×240 pixel resolution down to 13×13 features. We hope that by simplifying the input we will be able to speed up the Q-learning process and yield long-term, generalized strategies in a short amount of time. We won't be utilizing Murphy's algorithm, as we are interested in solving the problem from a neural networking approach.

Techniques and Innovations

We believe there is great potential in MarI/O's approach, and we have identified several areas for improvement. Some of these may be considered errors, while others are better described as an increase in scope.

1. *Train on multiple levels.* Training on multiple levels would help prevent overfitting to the first level that Mario is trained on. As soon as it is able to beat the first level, the strategy it has learned does not help beat it the second. To mitigate this, the fitness function could be summed over multiple levels. A distributed training system would minimize the increase in training time, as the levels could be played in parallel.

2. *Prune unnecessary outputs.* MarI/O allows all possible buttons as outputs of the neural network: $\{Start, Select, Left, Right, Up, Down, A, B\}$. While some of these are crucial, others are clearly unnecessary. Removing *Start* and *Select*, for example, would speed up training rate with no drawbacks; these buttons are not useful in playing the game. Up and $Down$, may be removed for a slight tradeoff as they are rarely used in critical parts of the game, and B can be set to always be pressed because it is used to sprint.

3. *Fitness function tweaks.* Tweaking the fitness function to value progress over speed may change what types of strategies prevail.

4. *Timeout strategy refinement.* MarI/O employs a timeout rule when training, in order to quickly determine when a given neural network is not making progress on a level: if Mario hasn't increased his rightmost x position in N frames, then stop. We believe this to be too aggressive, however, as there are certain levels that require Mario to wait before continuing.

5. *Include x and y velocities as inputs.* Being a "platformer" game, there are many situations in Mario where velocity control is crucial. Adding x and y velocities as inputs could allow an improvement in training.

6. *A new input algorithm.* Through testing the MarI/O feature selection algorithm, we discovered many inconsistencies that should be addressed. Hammers, which are able to kill Mario, do not appear as -1. Depending on Mario's position, there are situations where enemies appear to float above the ground, even though they are clearly grounded. Platforms are represented as -1, even though they do not hurt Mario. Coins appear as 1, even though they are not solid (Mario can walk through them). Some enemies appear present when they are in an unreachable state (behind a pipe). At the end of the level, Mario jumps as high as he can onto the flagpole to earn extra points. The flagpole appears as a column of 1's which may be yielding strange obstacle avoidance strategies.

7. *Altering the view frame.* MarI/O's view frame is a square centered on Mario, covering about 75% of the visible screen in the x and y axis. Being relative to Mario, this causes Mario to be unable to see the ground from the apex of his jump. Shifting the view frame to the right and down could be beneficial, as the majority of the game is spent making rightward progress with carefully timed landings.

8. *Distinguish enemies.* Enemies in Mario vary wildly: from walking goombas, to jumping koopas, to flying cheep-cheeps, to fire-breathing Bowser, there is a wide array of enemy behavior to challenge Mario. By representing all enemies as -1, too much information may be lost.

9. *Add memory.* Since MarI/O can't remember past frames, he is severely limited in his possible strategies. Imagine being given random pictures from a video game and trying to select which buttons to press. Even

worse, MarI/O's view frame doesn't include the entire game screen, meaning he is blind to objects far below or above him–every long jump becomes a "leap of faith." In addition, he is unable to determine enemy velocities, since every event is a discrete point in time without context. Giving multiple frames as inputs could help. A deep-learning neural network in a fashion similar to DeepMind's approach may prove an even better solution.

ANN Design

For this project, the fitness of the network will be evaluated by this formula:

$$fitness = \sum_{i=1}^{32} = \begin{cases} difficulty(i) * (distance_i + bonus) - (t * frames_i) & victorious \\ difficulty(i) * distance_i - (t * frames_i) & defeated \end{cases}$$

Where $difficulty$ is a function that approximates how difficult a level is, $distance_i$ is the distance that Mario traveled into the level, $frames_i$ is how many frames Mario spent playing the level, t is a constant, and $bonus$ is a large constant for beating a level. Some levels may be omitted if they are deemed fundamentally different than the others (i.e. water levels). We will evaluate the fitness function in parallel to speed up training.

Part 2

For part 2, we will use *neuroevolution of augmenting topologies (NEAT)*, a genetic algorithm to evolve neural network topologies. We will use a set of inputs processed from the game state at a given moment in time. These inputs will include a 13×13 matrix relative to Mario. This box will be a tiled representation of the pixels around Mario, partitioned into 16×16 pixel boxes. These boxes will have either a 1, 0, or -1, depending on if that tile contains a solid (walkable) block, background tile, or enemy, respectively. We will also include Mario's x and y velocities. Thus, $13 \times 13 + 2$ for a total of 171 inputs.

These inputs are similar to MarI/O, with multiple modifications detailed in *Techniques and Innovations*. Most importantly, we will shift the box around Mario down and to the right in order to encompass more important

parts of the level. In addition, we will fix numerous issues with MarI/O's feature selection algorithm (coins are considered 1, deadly hammers are missing, floating enemies, etc).

These inputs will be driven through a neural network, with 3 outputs that represent the buttons to press ($Left, Right$, and A). Every k frames, the game is suspended as inputs are retrieved, given to the neural network, and processed. Once processed, the controller's buttons are set to the network's output, and the game is resumed. Since the NES runs at 60 frames per second, we choose $k = 4$ to provide a reasonable compromise between training speed and realistic gameplay.

Once Mario defeats the level, dies, or gets stuck for a certain amount of time, the level is stopped, and the network begins playing the next level. Once all levels have been played by a given neural network, its fitness is evaluated, and the NEAT algorithm produces a new neural network.

Part 3

For part 3, we will use a deep Q-Learning network, similar to Google's DeepMind paper. Our hope is that we will be able to generate an agent that is capable of learning optimal strategies – similar to DeepMind's *Breakout* agent. For example, some game experts generate sequences of controller inputs called *tool-assisted speedruns (TAS)* which are designed to play the game in the shortest time possible. In some cases, a TAS is even designed to leverage "glitches" to minimize time in ways that the game creators did not even anticipate (DarkEncodesRising, 2011).

We'll use the same inputs and fitness function from part 2 with a multi-layer convolutional neural network. Our network will consist of $13 \times 13 + 2$ inputs followed by three convolutional layers and two fully-connected layers with 3 outputs. The input and output layers will represent the same constructs as they did in part 2.

Tools

To accomplish these techniques for machine learning, we will be using an emulator called BizHawk, a multi-system, multi-platform. This emulator allows fast emulation while running Lua scripts through a built-in Lua console. The emulator itself is written in C# while the emulated NES is written in Visual

C++. BizHawk offers a powerful Lua API that allows reading from emulated NES memory, loading savestates, and setting controller inputs, among other features. This allows us to read specific parts of the memory that corresponds with values in the game. For instance, we can use the value read from memory address 0x001D to detect whether Mario is sliding down the flag pole at the end of a level, thus indicating victory. In addition, Bizhawk has a RAM watch feature that allows us to watch the memory addresses change while the game is being played. This will be instrumental in determining which memory values are relevant to this project.

Another tool that we may use is called Torch, a machine learning library written in Lua. Torch is the tool that DeepMind used for their deep Q-learning network. This allows us to train neural networks on a UNIX system (possibly Windows too, if built from source) and is extremely efficient.

References

DarkEncodesRising. (2011, Dec). *Tas: Snes super mario world "glitched" in 2:36.40 by masterjun.* YouTube. Retrieved from https://www.youtube.com/watch?v=Syo5sI-iOgY

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... et al. (2015, Feb). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529533. Retrieved from http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html

Murphy VII, T. (2013). The first level of super mario bros. is easy with lexicographic orderings and time travel ... after that it gets a little tricky. Retrieved from https://www.cs.cmu.edu/ tom7/mario/mario.pdf

SethBling. (2015, Jul). *Mari/o - machine learning for video games.* YouTube. Retrieved from https://www.youtube.com/watch?v=qv6UVOQ0F44

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99-127. Retrieved from http://nn.cs.utexas.edu/?stanley:ec02