

A Simple Super Mario Bot using RL

A machine learning project

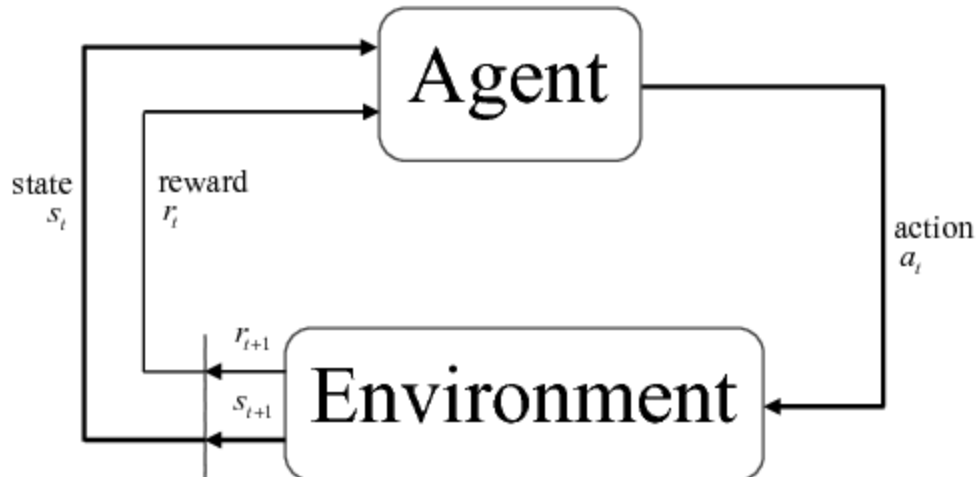
What is Reinforcement Learning?

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning is considered as one of three machine learning paradigms, alongside supervised learning and unsupervised learning.

It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

Reinforcement learning, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms. In the operations research and control literature, reinforcement learning is called approximate dynamic programming, or neuro-dynamic programming. Basic reinforcement is modeled as a Markov decision process.

- a set of environment and agent states, \mathcal{S} ;
- a set of actions, A , of the agent;
- $P_o(S, S') = P_r(S_{t+1} = S' \mid S_t = S, a_t = a)$ is the probability of transition from state S to state S' under action a .
- $R_o(S, S')$ is the immediate reward after transition from S to S' with action a .

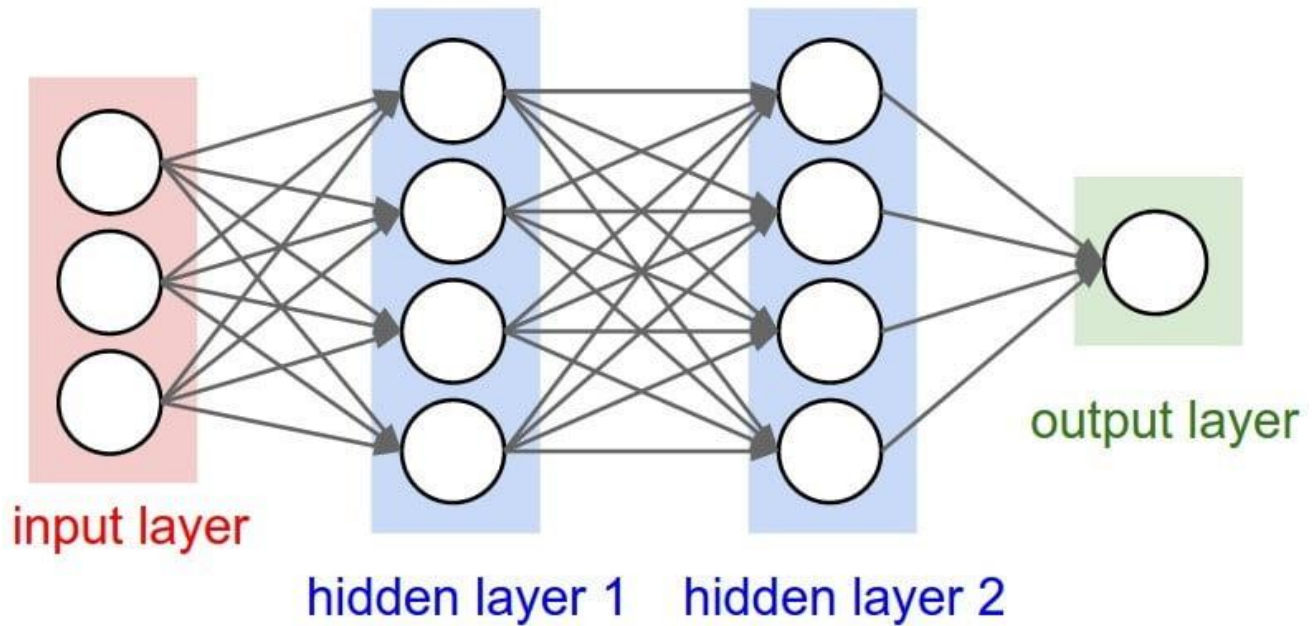


A reinforcement learning agent interacts with its environment in discrete time steps. At each time t , the agent receives an observation $o_{\{t\}}$, which typically includes the reward $r_{\{t\}}$. It then chooses an action $a_{\{t\}}$ from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state $S_{\{t+1\}}$ and the reward $r_{\{t+1\}}$ associated with the transition $(S_{\{t\}}, a_{\{t\}}, S_{\{t+1\}})$ is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can (possibly randomly) choose any action as a function of the history.

What is a Neural Network (ANN)?

Artificial neural networks (ANN) or connectionist systems are computing systems vaguely inspired by the biological neural networks that constitute animal brains. The neural network itself is not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.

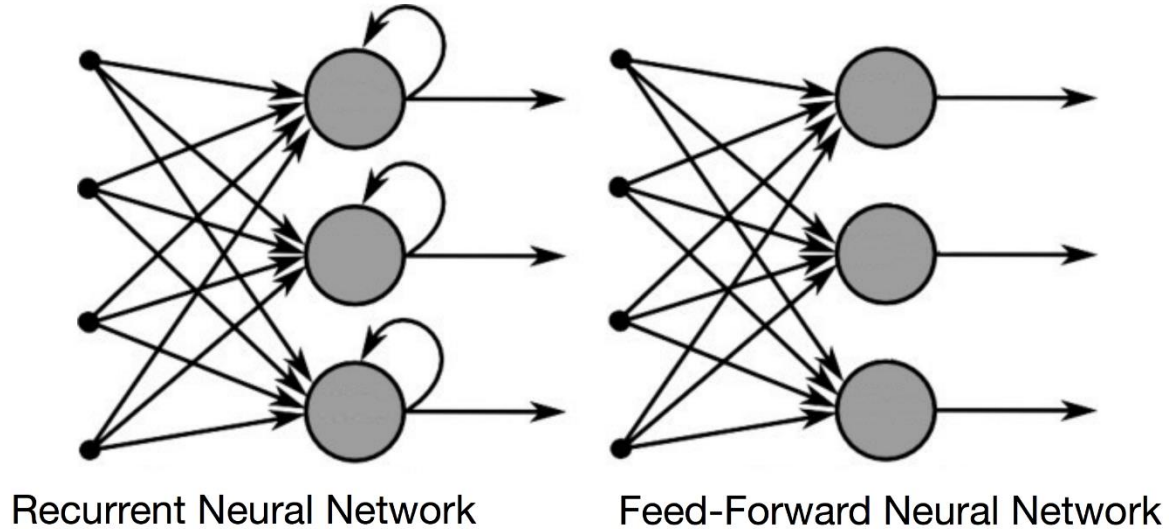
An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.



What is a Recurrent Neural Network (RNN)?

A recurrent neural network (RNN) is a class of artificial neural network where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.

The term "recurrent neural network" is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled.



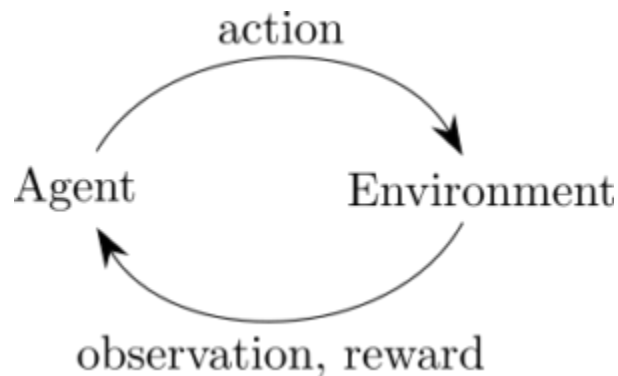
What is OpenAI Gym?

OpenAI is a non-profit research company that is focused on building out AI in a way that is good for everybody. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like pong or pinball. Gym is an open source interface to reinforcement learning tasks. Gym provides an environment and it is up to the developer to implement any reinforcement learning algorithms.

- `gym.make("ENVIRONMENT_NAME")`: returns the environment that was passed as parameter.
- `env.reset()`: This command will reset the environment. It returns an initial observation.
- `for _ in range(1000)`: This line in python code will run an instance of an environment for 1000 time steps.
- `env.render()`: This command will display a popup window. Since it is written within a loop, an updated popup window will be rendered for every new action taken in each step.
- `env.step()`: This command will take an action at each step. The action is specified as its parameter. `env.step` function returns four parameters, namely observation, reward, done and info. These four are explained below:
 1. observation/state: an environment-specific object representing your observation of the environment.

2. reward: amount of reward achieved by the previous action. It is a floating data type value. The scale varies between environments.
3. c) done: A Boolean value stating whether it's time to reset the environment again.
4. d) info (object): diagnostic information useful for debugging.

Each time step, the agent chooses an action, and the environment returns an observation and a reward.



Note: For the Q-learning example we'll be using gym, while for the Mario bot we'll use gym-retro (Refer: <https://github.com/openai/retro>); this is meant for old NES games.

Let look into some of RL algorithms we can use. In this code, we'll use Q-learning method and NEAT algorithm.

What is Q-learning?

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.

We shall implement this algorithm for a simple game called the Frozen lake. The docs for this are present at:

`<project-directory>/Docs/Frozen_Lake_Docs`

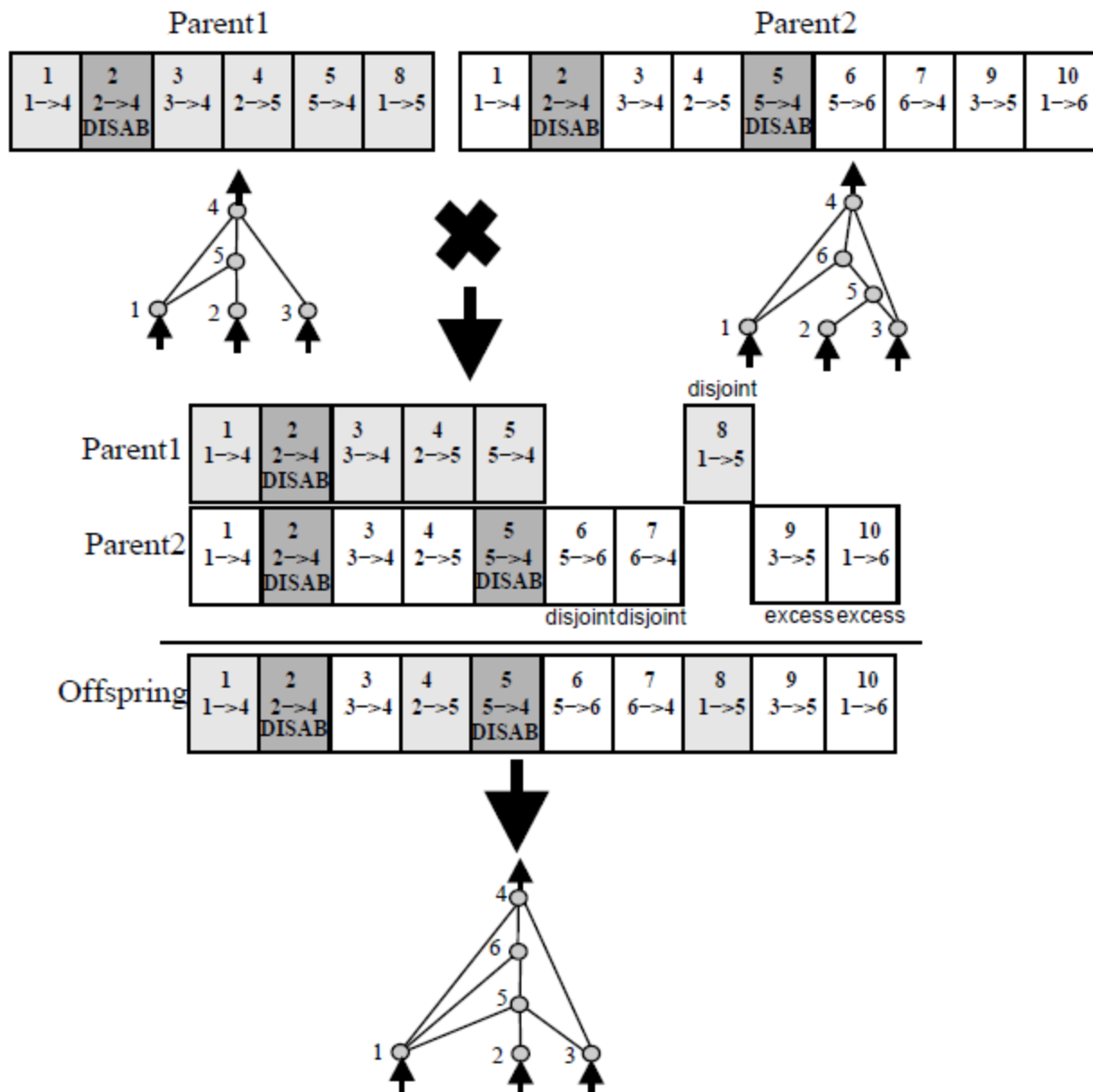
As mentioned in the above doc, frozen lake is a discrete game, i.e., it has fixed actions which are not linear functions of key presses. However the opposite holds true for SuperMarioBros, which returns a 256x256x3 state matrix depicting the frame of the game. Hence we have to process the image before using it in a dense layer, but the main problem being overcoming the continuous action sets of Mario (along with other disadvantages mentioned in the above doc).

What is NEAT?

Neuro Evolution of Augmenting Topologies (NEAT) is a genetic algorithm (GA) for the generation of evolving artificial neural networks (a neuro-evolution technique) developed by Ken Stanley in 2002 while at The University of Texas at Austin. It alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. It is based on applying three key techniques: tracking genes with history markers to allow crossover among topologies, applying speciation (the evolution of species) to preserve innovations, and developing topologies incrementally from simple initial structures ("complexifying").

The NEAT approach begins with a perceptron-like feed-forward network of only input neurons and output neurons. As evolution progresses through discrete steps, the complexity of the network's topology may grow, either by inserting a new neuron into a connection path, or by creating a new connection between (formerly unconnected) neurons.

The competing conventions problem arises when there is more than one way of representing information in a phenotype. For example, if a genome contains neurons A, B and C and is represented by [A B C], if this genome is crossed with an identical genome (in terms of functionality) but ordered [C B A] crossover will yield children that are missing information ([A B A] or [C B C]), in fact 1/3 of the information has been lost in this example. NEAT solves this problem by tracking the history of genes by the use of a global innovation number which increases as new genes are added. When adding a new gene, the global innovation number is incremented and assigned to that gene. Thus the higher the number the more recently the gene was added. For a particular generation if an identical mutation occurs in more than one genome they are both given the same number, beyond that however the mutation number will remain unchanged indefinitely.



To implement this, we'll be using neat-python, which has the necessary modules.

(Refer: <https://github.com/CodeReclaimers/neat-python>)

In the current implementation of NEAT-Python, a population of individual genomes is maintained. Each genome contains two sets of genes that describe how to build an artificial neural network:

- Node genes, each of which specifies a single neuron.
- Connection genes, each of which specifies a single connection between neurons.

To evolve a solution to a problem, the user must provide a fitness function which computes a single real number indicating the quality of an individual genome: better ability to solve the problem means a higher score. The algorithm progresses through a user-specified number of generations, with each generation being produced by reproduction (either sexual or asexual) and mutation of the most fit individuals of the previous generation.

The reproduction and mutation operations may add nodes and/or connections to genomes, so as the algorithm proceeds genomes (and the neural networks they produce) may become more and more complex. When the preset number of generations is reached, or when at least one individual (for a fitness criterion function of max; others are configurable) exceeds the user-specified fitness threshold, the algorithm terminates.

Explanation of parameters in the configuration file:

- [NEAT] section:
 1. `fitness_criterion`: The function used to compute the termination criterion from the set of genome fitnesses.
 2. `fitness_threshold`: When the fitness computed by `fitness_criterion` meets or exceeds this threshold, the evolution process will terminate, with a call to any registered reporting class' `found_solution` method.
 3. `pop_size`: The number of individuals in each generation.
 4. `reset_on_extinction`: if this evaluates to True, when all species simultaneously become extinct due to stagnation, a new random population will be created. If False, a `CompleteExtinctionException` will be thrown.
- [DefaultGenome] section:
 1. `activation_default`: The default activation function attribute assigned to new nodes; sigmoid.
 2. `Aggregation_default`: The probability that mutation will replace the node's aggregation function with a randomly-determined member of the `aggregation_options`. Valid values are in [0.0, 1.0].
 3. `bias_init_type`: If set to gaussian or normal, then the initialization is to a normal/gaussian distribution. If set to uniform, a uniform distribution from $\max(\text{bias_min_value}, (\text{bias_init_mean} - (\text{bias_init_stdev} * 2)))$ to $\min(\text{bias_max_value}, (\text{bias_init_mean} + (\text{bias_init_stdev} * 2)))$. (Note that the standard deviation of a uniform distribution is not $\text{range}/0.25$, as implied by this, but the range divided by a bit over 0.288 (the square root of 12));

however, this approximation makes setting the range much easier.) This defaults to “gaussian”.

4. `compatibility_threshold`: Individuals whose genomic distance is less than this threshold is considered to be in the same species.
5. `feed_forward`: False (using recurrent nn)
6. `response_init_mean`: The mean of the normal/gaussian distribution, if it is used to select response multiplier attribute values for new nodes.
7. `weight_init_mean`: The mean of the normal/gaussian distribution used to select weight attribute values for new connections.

- [DefaultStagnation] section:

1. `max_stagnation`: Species that have not shown improvement in more than this number of generations will be considered stagnant and removed. This defaults to 15.
2. `species_elitism`: The number of species that will be protected from stagnation; mainly intended to prevent total extinctions caused by all species becoming stagnant before new species arise.

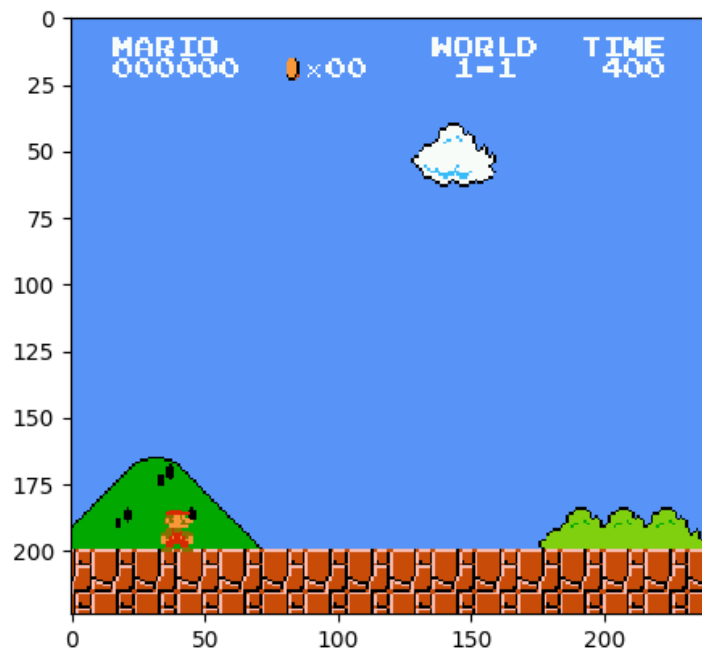
- [DefaultReproduction] section:

1. `Elitism`: The number of most-fit individuals in each species that will be preserved as-is from one generation to the next.
2. `survival_threshold`: The fraction for each species allowed to reproduce each generation.

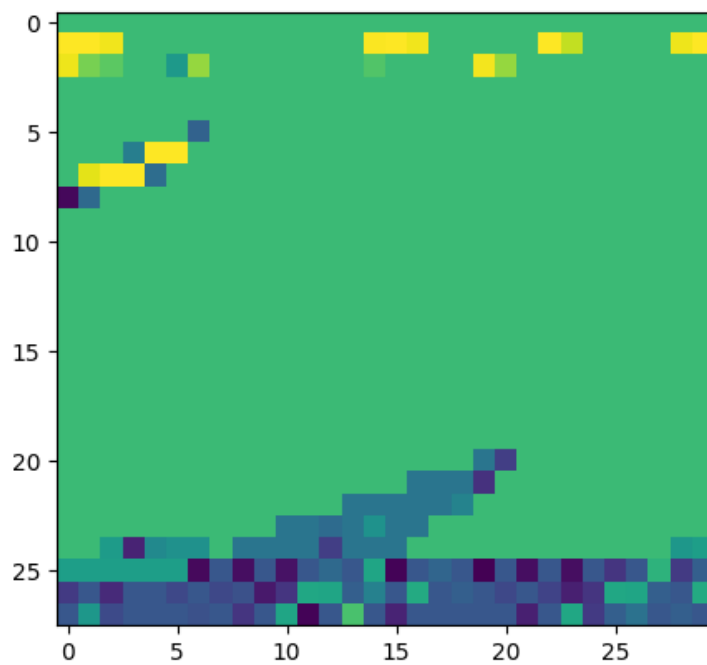
What is OpenCV?

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. We use this to reduce the 256x256x(RGB) pixel map into a smaller image by combining 8 into 1 pixel. This enables faster calculations.

Original:

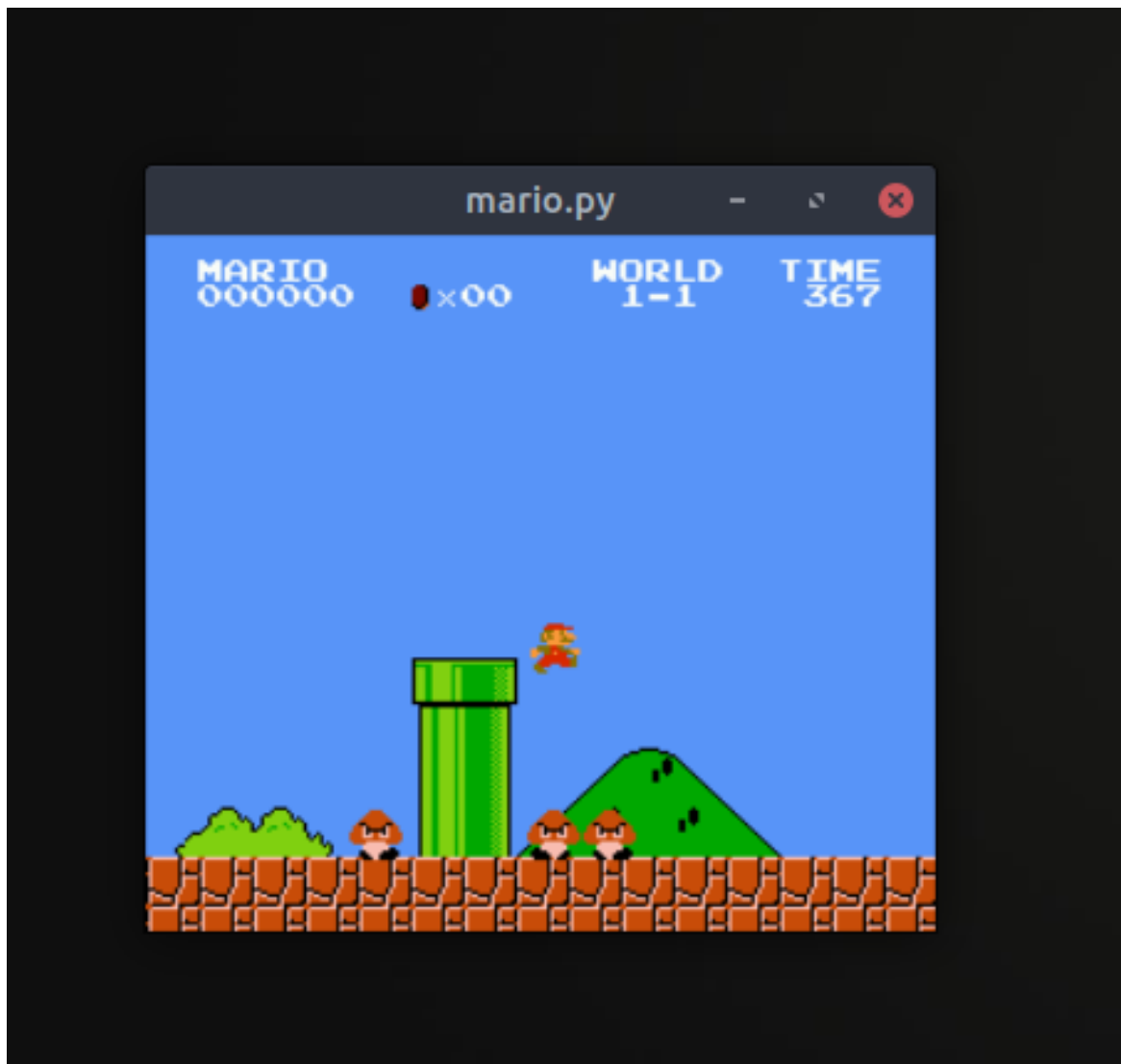


Compressed (via opencv):

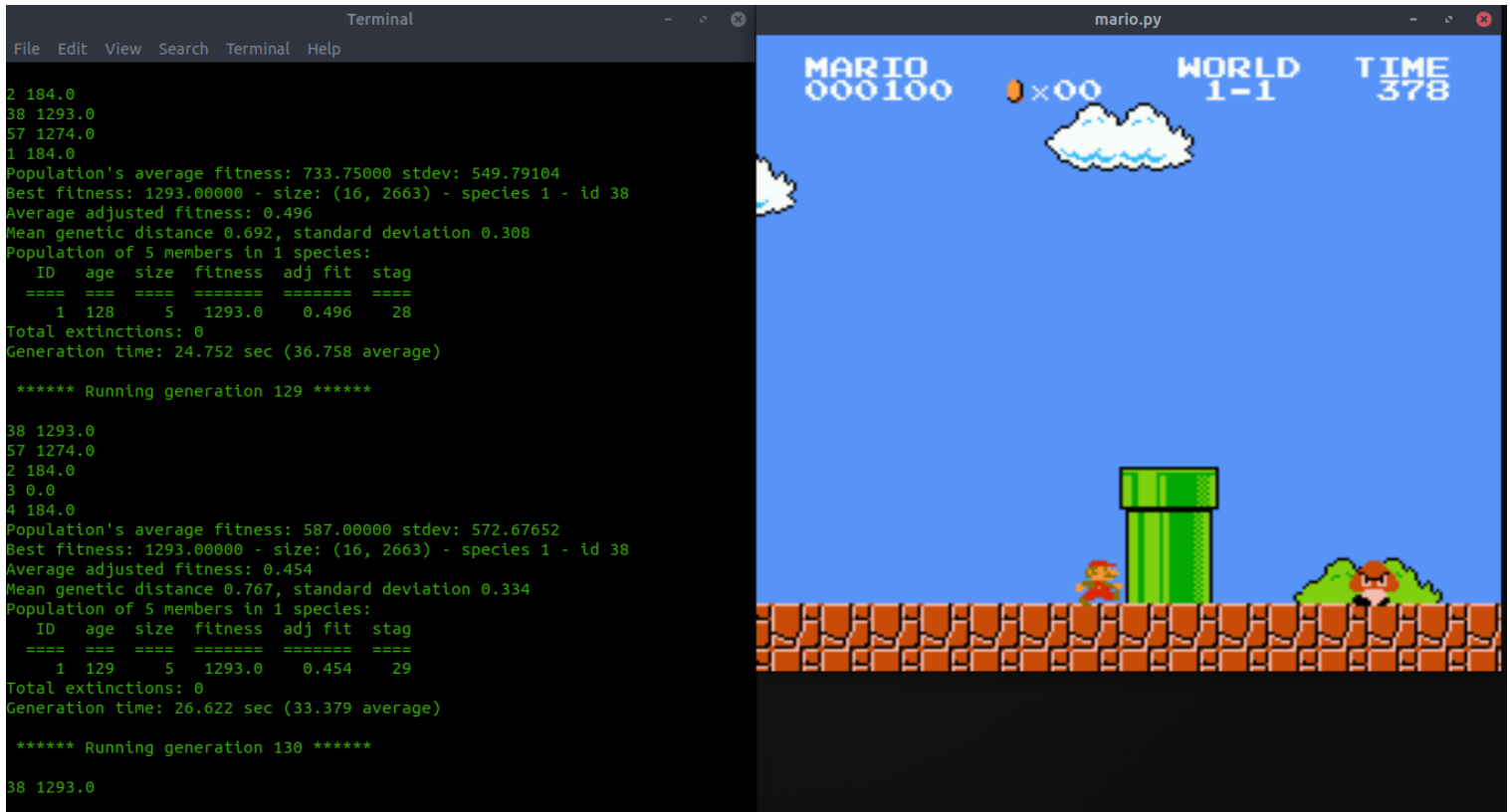


What is FCEUX?

FCEUX is a Nintendo Entertainment System (NES), Famicom, Famicom Disk System (FDS), and Dendy emulator. The FCEUX concept is that of an "all in one" emulator that offers accurate emulation and the best options for both casual play and a variety of more advanced emulator functions. For pro users, FCEUX offers tools for debugging, rom-hacking, map making, Tool-assisted movies, and Lua scripting.



Our objective is to make Mario complete the level1-1 of the world, by maximizing reward. Although in the beginning the generations don't get much rewards, over multiple cross-breedings the population racks up a lot of rewards.



```
Terminal
File Edit View Search Terminal Help

2 184.0
38 1293.0
57 1274.0
1 184.0
Population's average fitness: 733.75000 stdev: 549.79104
Best fitness: 1293.00000 - size: (16, 2663) - species 1 - id 38
Average adjusted fitness: 0.496
Mean genetic distance 0.692, standard deviation 0.308
Population of 5 members in 1 species:
  ID  age  size  fitness  adj fit  stag
====  ==  ====  =====  =====  ==
   1  128   5   1293.0    0.496   28
Total extinctions: 0
Generation time: 24.752 sec (36.758 average)

***** Running generation 129 *****

38 1293.0
57 1274.0
2 184.0
3 0.0
4 184.0
Population's average fitness: 587.00000 stdev: 572.67652
Best fitness: 1293.00000 - size: (16, 2663) - species 1 - id 38
Average adjusted fitness: 0.454
Mean genetic distance 0.767, standard deviation 0.334
Population of 5 members in 1 species:
  ID  age  size  fitness  adj fit  stag
====  ==  ====  =====  =====  ==
   1  129   5   1293.0    0.454   29
Total extinctions: 0
Generation time: 26.622 sec (33.379 average)

***** Running generation 130 *****

38 1293.0
```

mario.py

MARIO 000100 x00 WORLD 1-1 TIME 378

- Note: The values of the config are tweaked to give multiple winners by the end of ~200 generations.
- The neat-checkpoint-* are binary files which help use to restore the learnt model.
- Winners stored in winner.pkl (binary file)
- Please read the README.md for installation guide.
- rom.nes == SuperMarioBros-Nes (v0).
- Check out <project-directory>/Frozen_Lake/ for implementation of Frozen lake using Deep Q Learning.
- Read the READ present under this directory for installation guide.

Disadvantages of NEAT:

Evolutionary algorithm optimizers are global optimization methods and scale well to higher dimensional problems. They are robust with respect to noisy evaluation functions, and the handling of evaluation functions which do not yield a sensible result in given period of time is straightforward.

The algorithms can easily be adjusted to the problem at hand. Almost any aspect of the algorithm may be changed and customized. On the other hand, although lots of research has been done on which evolutionary algorithm is best suited for a given problem, this question has not been answered satisfactorily. While the standard values usually provide reasonably good performance, different configurations may give better results. Furthermore, premature convergence to a local extremum may result from adverse configuration and not yield (a point near) the global extremum.

Thank you

