AUTOMATING RETRO GAMES USING REINFORCEMENT LEARNING DEEP Q-LEARNING MODELS

Aadhaar Koul , Arjun Charak
Department of Computer secience and Egineering
Model Institute of engineering and Technology
Kot Bhalwal , Jammu , Jammu and Kashmir
2020a1r040 , 2020a1r058 - mietjammu.in

Dr. Arjun Puri
Department of Computer secience and Egineering
Model Institute of engineering and Technology
Kot Bhalwal , Jammu , Jammu and Kashmir
arjun.cse@mietjammu.in

***Abstract*— General game testing relies on the use of human play testers, play test scripting, and prior knowledge of areas of interest to produce relevant test data. Using deep reinforcement learning (DRL), we introduce a self-learning mechanism to the game testing framework. With DRL, the framework is capable of exploring and/or exploiting the game mechanics based on a user-defined, reinforcing reward signal. As a result, test coverage is increased and unintended game play mechanics, exploits and bugs are discovered in a multitude of game types. In this paper, we show that DRL can be used to increase test coverage, find exploits, test map difficulty, and to detect common problems that arise in the testing of Retro games.In this paper, we study applying Reinforcement Learning to design a automatic agent to play the game Super Mario Bros. One of the challenge is how to handle the complex game environment. By abstracting the game environment into a state vector and using Q learning — an algorithm oblivious to transitional probabilities — we achieve tractable computation time and fast convergence. After training for 5000 iterations, our agent is able to win about 90 percent of the time. We also compare and analyze the choice of different learning rate (alpha) and discount factor (gamma).[1][3]**

***Index Terms*— DRL, Q-Learning , Reinforcement Learing , CUDA**

I. INTRODUCTION

Using artificial intelligence (AI) and machine learning (ML) algorithms to play computer games has been widely discussed and investigated, because valuable observations can be made on the ML play pattern vs. that of a human player, and such observations provide knowledge on how to improve the algorithms. Mario AI Competition [1] provides the framework [2] to play the classic title Super Mario Bros, and we are interested in using ML techniques to play this game. Reinforcement Learning (RL) [3] is one widely-studied and promising ML method for implementing agents that can simulate the behavior of a player [4]. In this project, we study how to construct an RL Mario controller agent, which can learn from the game environment. One of the difficulties of using RL is how to define state, action, and reward. In addition, playing the game within the framework requires realtime response, therefore the state space cannot be too large. We use a state representation similar to [4], that abstracts the whole environment description into several discrete-valued key attributes. We use the Q-Learning algorithm to evolve the decision strategy that aims to maximize the reward. Our controller agent is trained and tested by the 2012 Mario AI Competition evaluation system. The rest of this report is organized as follows: Section 2 provides a brief overview of the Mario AI interface and the Q-Learning algorithm; Section 3 explains how we define the state, action, reward to be used in RL; Section 4 provides evaluation results; Section 5 concludes and give some possible future work.[3]

II. MOTIVATION

We the team members of the Trex group always wanted to break things in order to make them better. We have always tried to learn programming in the most fun way. We always came up with ideas like building retro game like mario , Doom , Pacman etc. But were never able to make one maybe due to the lack of motivation. But in the recent days we got an opportunity to not only make a game but also to automate it using a concept called as the reinforcement learning as a part of our current semester final project. We gave in our best efforts to make this project only because the team members wanted to learn something new , fun and give something back to the gaming industry. The idea of building a game on our own excited us so much that we decided to automate 4 Retro games that included the MARIO , SNAKE , DOOM , FLAPPY BIRD , the detals of which the reader can find in the contents of the paper.

III. CONTRIBUTION

The team members of the project group Trex accompanied by the class coordinator - Dr. Arjun Puri had their utmost contribution to the project as well as the paper. The Members have given in countless hours in the making of the project and which comprises of the internet scraping for required snippets of code as well as the Model and image content for the project. We found ourselves lucky enough to be mentored by our class coordinator Dr. Arjun Puri sir who guided us throughout the project and directed the team members in the right direction.

IV. RELATED WORK

Many of the open source contributors in the tech world have tried to automate retro games using a reinforcement model . Given below is the list of open source contributed who have also been automating the retro games using a matching reinforcement learning model.

- Nicholas renotte - Youtube cannel - Title[Build a Mario AI Model with Python | Gaming Reinforcement Learning]
- Sebastian Heinz7 - Using reinforcement learning to play Super mario bros.
- [10]
- [9]

V. BLOCK DIAGRAM

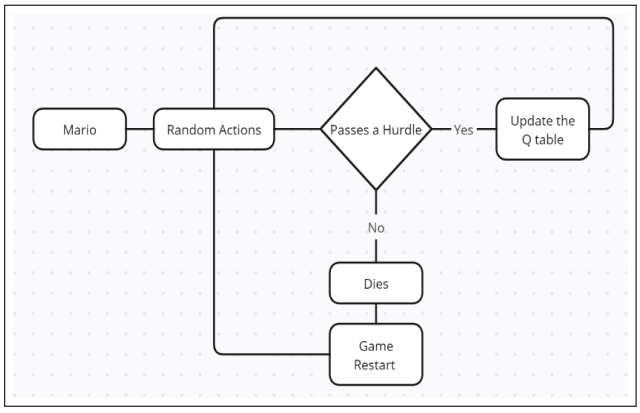

Fig. 1. Game logic for the Mario-Bros

In the above given block diagram the reinforcement execution of the mario game is depicted using a control flow for the mario avatar. The mario avatar when faces an 9impact or failure , the incident is recorded as a punishment and the game restarts . While on the other hand if the mario avatar makes the correct moves that take the avatar forward in the game , those are considered as the rewards , respectively as proposed in the reinforcement learning concept.

VI. NOTATIONS , ABBREVIATIONS AND ASSUMPTIONS

- pip - PIP is a package manager for Python packages, or modules if you like.
- gym - Gym is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API.
- nes - Project description. nes-py is an NES emulator and OpenAI Gym interface for MacOS, Linux, and Windows based on the SimpleNES emulator.
- stable-baselines3 - Stable-Baselines3 provides open-source implementations of deep reinforcement learning (RL) algorithms in Python. The implementations have been benchmarked against reference codebases, and automated unit tests cover 95
- matplotlib - Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK
- pyplot - Pyplot is an API (Application Programming Interface) for Python's matplotlib that effectively makes matplotlib a viable open source alternative .
- PPO - PPO is a policy gradient method and can be used for environments with either discrete or continuous action spaces. It trains a stochastic policy in an on-policy way. Also, it utilizes the actor critic method
- joypadspace -
- env - Abbrivation for teh mario environment that comprises of the multiple functionalities and data to play around with
- state - The curent position or condition of the environment or the mario avatar at any instant.
- reward - In reinforcement learning, the agent is rewarded for taking controls that lead to successful states. The rewards can be immediate, such as receiving a point for each step taken in the right direction, or they can be delayed, such as receiving a point at the end of the episode if the goal was reached.
- torch - PyTorch is an open source machine learning (ML) framework based on the Python programming language and the Torch library. Torch is an open source ML library used for creating deep neural networks and is written in the Lua scripting language. It's one of the preferred platforms for deep lear ning research
- plt - Short form / abbrivation for using the notation of matplotlib.pyplot as plt to use in the code . In the current research we have used the pla as a substitute for the matplotlib.pyplot . example - (import matplotlib.pyplot as plt).
- idx - index for a certain value in a looping condition

- RL Model - Short for reinforcement learning model. Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

VII. Conceptual Frameworks

A. Gym Environments

In order to build a reinforcement learning aplication, we need two things: (1) an environment that the agent can interact with and learn from (2) the agent, that observes the state(s) of the environment and chooses appropriate actions using Q-values, that (ideally) result in high rewards for the agent. An environment is typically provided as a so called gym, a class that contains the neecessary code to emulate the states and rewards of the environment as a function of the agent's actions as well further information, e.g. about the possible action space. Here is an example of a simple environment class in Python: [10]

Algorithm 1: Sequential Barnes-Hut Algorithm

```
1:  Function Environment() is
2:      Function –init–(self) is
3:          Pass
4:      Function step(self,action) is
5:          return next-state, reward, done, info
6:      Function reset(self) is
7:          Pass
8:      Function render(self) is
9:          Pass
10: Environment()
```

B. Environmet Wrappers

Gym environments contain most of the functionalities needed to use them in a reinforcement learning scenario. However, there are certain features that do not come prebuilt into the gym, such as image downscaling, frame skipping and stacking, reward clipping and so on. Luckily, there exist so called gym wrappers that provide such kinds of utility functions. An example that can be used for many video games such as Atari or NES can be found here. For video game gyms it is very common to use wrapper functions in order to achieve a good performance of the agent. The example below shows a simple reward clipping wrapper. [10]

Algorithm 2: Sequential Barnes-Hut Algorithm

```
1: import gym Function
   CliprewardEnv(gym.RewardWrapper) is
2:     """ Example wrapper for reward clipping """
3:     Function –init–(self,env) is
4:         gym.RewardWrapper.–init–(self, env)
5:     Function reward(self, reward) is
6:         return np.sign(reward)
```

C. The Super Mario Bros NES environment

For our Super Mario Bros reinforcement learning experiment, I've used gym-super-mario-bros. The API ist straightforward and very similar to the Open AI gym API. The following code shows a random agent playing Super Mario. This causes Mario to wiggle around on the screen and – of course – does not lead to a susscessful completion of the game.

1) Setup a mario environment:

- pip install gym-super-mario-bros nes-py

We are Installing gym-super-mario-bros nes-py. gym-super-mario-bros package is used to set up our gaming environment where our Mario will save the Princess Peach from Bowser and have you remembered the controls in this game.[4]

- Left dpad – Move left, enter pipe to the left of Mario.
- Right dpad – Move right, enter pipe to the right of Mario.
- Up dpad – Enter pipe above Mario or enter door behind Mario.
- Down dpad – Crouch, Ground Pound (In the air), enter pipe below Mario.
- A button – Jump, confirm menu selection.
- B button – Jump, exit menu.
- X button/Y button – Dash (Hold), launch fireball (Fire Mario only).

Nes-py help us to build a virtual joypad for python for our model to control Mario to do certain tasks.[9] [10]

2) Importing the required dependence:

- import gym-super-mario-bros - is importing the game itself.
- from nes-py.wrappers import JoypadSpace -
- from gym-super-mario-bros.actions import SIMPLE-MOVEMENT

By default, gym-super-mario-bros environments use the full NES action space of 256 discrete actions. To constrain

this, gym-super-mario-bros.actions provide three actions lists (RIGHT-ONLY, SIMPLE-MOVEMENT, and COMPLEX-MOVEMENT) we are using SIMPLE-MOVEMENT which has only 7 actions that help us to reduce the data we are going to process. actions are the combination of controls in the game.

3) Setting up and Running the game:

We are Installing gym-super-mario-bros nes-py. gym-super-mario-bros package is used to set up our gaming environment where our Mario will save the Princess Peach from Bowser and have you remembered the controls in this game. [5] [6]

- Left dpad – Move left, enter pipe to the left of Mario.
- Right dpad – Move right, enter pipe to the right of Mario.
- Up dpad – Enter pipe above Mario or enter door behind Mario.
- Down dpad – Crouch, Ground Pound (In the air), enter pipe below Mario.
- A button – Jump, confirm menu selection.
- B button – Jump, exit menu.
- X button/Y button – Dash (Hold), launch fireball (Fire Mario only).

Algorithm 3: Sequential Barnes-Hut Algorithm

```
import gym-super-mario-bros
from nes-py.wrappers import JoypadSpace
from gym-super-mariobros.actions import
  SIMPLE-MOVEMENT
env = gym-super-mario-bros.make('S.M.B-v0')
env = JoypadSpace(env, SIMPLE-MOVEMENT)
Function run() is
    done = True
    foreach step in range(100000): do
        if done then
            env.reset()
        state, reward, done, info =
          env.step(env.action-space.sample())
        env.render()
    env.close()
run()
```

Gym-super-mario-bros has various environments where we are going to train our model to learn and play. Feel free to check out different environments to play with. I am going to use SuperMarioBros-v0as the default classical environment.[8][9]

Fig. 2. SuperMarioBros-v0

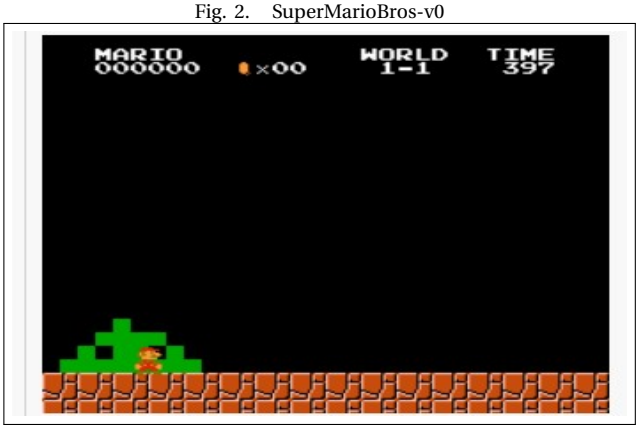

Fig. 3. SuperMarioBros-v1

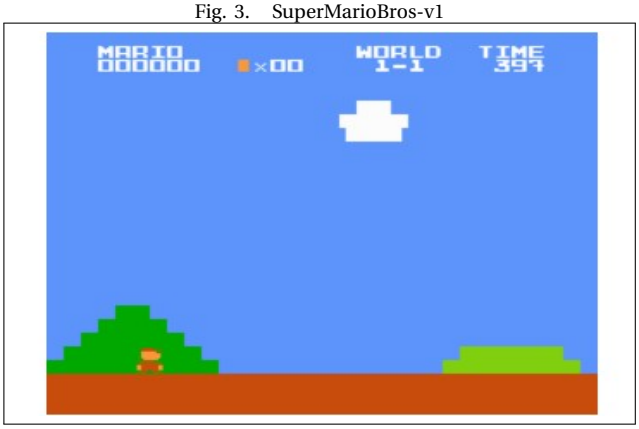

Fig. 4. SuperMarioBros-v3

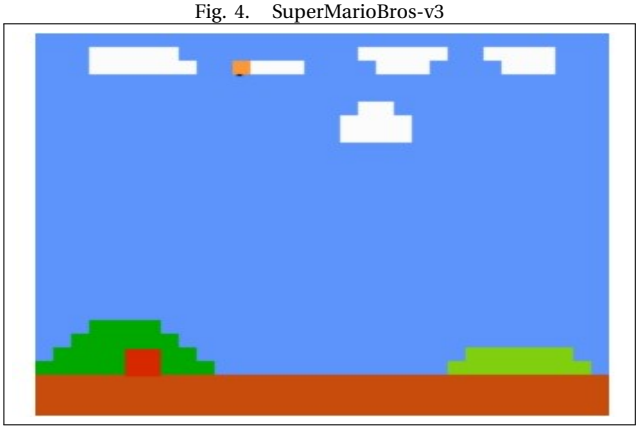

Fig. 5. SuperMarioBros-v0

D. Preprocessing Environment

In order to make the gym-syper-mario-bros reinforcement leraning model ready we actually need to to prepocess teh game so as to implement the learning mdoel to the game as better as possible . Our AI model must be able to see in what direction teh mario is moving or what enemies we are iteracting with etc. Packages like fram stack etc ca help us us out here.

Algorithm 4: Sequential Barnes-Hut Algorithm

```
!pip install torch==1.10.1+cu113
  torchvision==0.11.2+cu113
  torchaudio===0.10.1+cu113 -f
  https://download.pytorch.org/whl/cu113/torch-
  stable.html
!pip install stable-baselines3[extra]
from gym.wrappers import GrayScaleObservation
from stable-baselines3.common.vec-env import
  VecFrameStack, DummyVecEnv
from matplotlib import pyplot as plt
env = gym-super-mario-bros.make('S.M.B-v0')
env = JoypadSpace(env, SIMPLE-MOVEMENT)
env = GrayScaleObservation(env, keep-dim=True)
env = DummyVecEnv([lambda: env])
env = VecFrameStack(env, 4, channels-order='last')
state = env.reset()
state, reward, done, info = env.step([5])
plt.figure(figsize=(20,16))
Function Process() is
    foreach idx in range(state.shape[3]): do
        plt.subplot(1,4,idx+1)
        plt.imshow(state[0][:,:,idx])
    plt.show()
Process()
```

E. Train the reinforcemnt Learning Model

Now After the prepossing the game and makig it ready to be tested out with the reinforcemet learning model we import the necessary packages and also install the Stable-BAselines3 modeule form the open web whose process of installation can be seen in the upcoming columns of this research.

1) Installing the Stable-Baselines3 for the model.: Stable-Baselines3 provides open-source implementations of deep reinforcement learning (RL) algorithms in Python. The implementations have been benchmarked against reference codebases, and automated unit tests cover 95percent of the code. The algorithms follow a consistent interface and are accompanied by extensive documentation, making it simple to train and compare different RL algorithms. Our documentation, examples, and source-code are available at https://github.com/DLR-RM/stable-baselines3.

The Sable -Baselines3 by simply installing the pytorch first. The Givem below steps are given to follow along the installation of pytorch as well as the Stable-Baselines3.

- Head over the website : https://www.pytorch.org
- press the install button
- Select the appropriate configuration of the pytorch that is compatible with your system.

In Our case we have used the following configuration to test out the reinforcement learning model

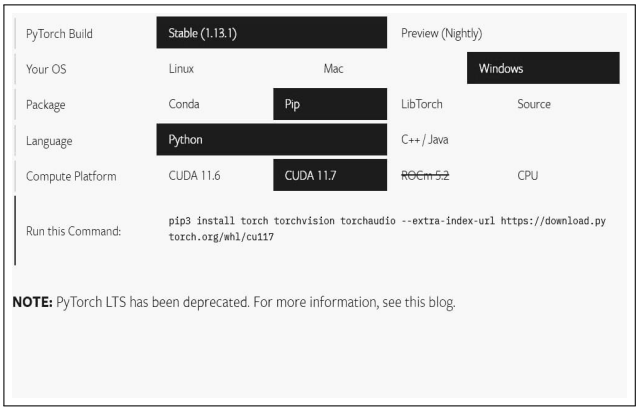

Fig. 6. Game logic for the Mario-Bros

- After selecting the required configurations copy the command that is generated at the bottom of the configuration and paste the command in the terminal.
- For the changes to actually take place it is a requiremet to restart the IDE and test out whether the pcakage has been imported successfully or not.
- The Stable-baselines3 is a part of the pytorch library so there is no need to download some other configuration files for the Stable-Baselines 3 it comes with the pytorch package only.

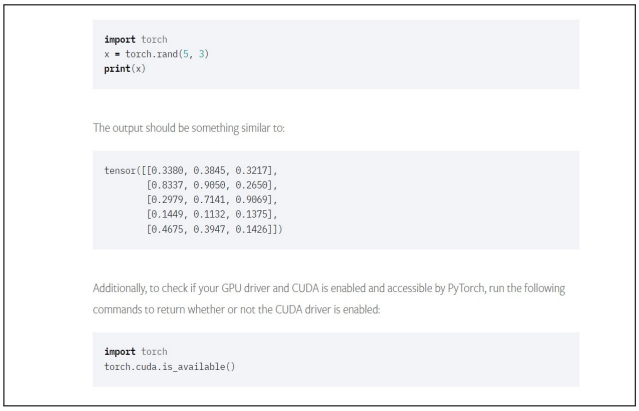

Fig. 7. You can test out the pytorch module by executing the above code

After the successful intallation of the package we can move over to the part where we form the algorithm around the imported packages .

Algorithm 5: Sequential Barnes-Hut Algorithm

```
import os
from stable-baselines3 import PPO
from stable-baselines3.common.callbacks import
  BaseCallback
Function TrainAndLoggingCallback(BaseCallback) is
    Function –init–(self, check-freq, save-path,
      verbose=1) is
        super(TrainAndLoggingCallback,
          self).–init–(verbose)
        self.check-freq = check-freq
        self.save-path = save-path
    Function -init-callback(self): is
        if self.save-path is not None: then
            os.makedirs(self.save-path,
              exist-ok=True)
    Function -on-step(self): is
        if self.n-calls (Rem / percent) self.check-freq
          == 0: then
            model-path = os.path.join(self.save-path,
              'best-model-'.format(self.n-calls))
            self.model.save(model-path))
TrainAndLoggingCallback()
CHECKPOINT-DIR = './train/'
LOG-DIR = './logs/'
callback =
  TrainAndLoggingCallback(check-freq=10000,
  save-path=CHECKPOINT-DIR)
model = PPO('CnnPolicy', env, verbose=1,
  tensorboard-log=LOG-DIR, learning-rate=0.000001,
  n-steps=512)
model.learn(total-timesteps=1000000,
  callback=callback)
model.save('thisisatestmodel')
```

F. Framestacking

Stack of 4 frames is simply to catch information like velocity of objects. The Max wrapper is because in Atari, sometimes the screen gets buggy and has stray pixels getting turned on/off in some frames which messed up the training. For e.g., in Pong, the DQN could get confused where the ball actually is if there are stray white pixels on the screen.It adds a huge amount of critical information. How else are you going to gauge direction of movement or velocity? Also many important states are shown through sprite animations, like ghost vulnerability in pacman. [9]

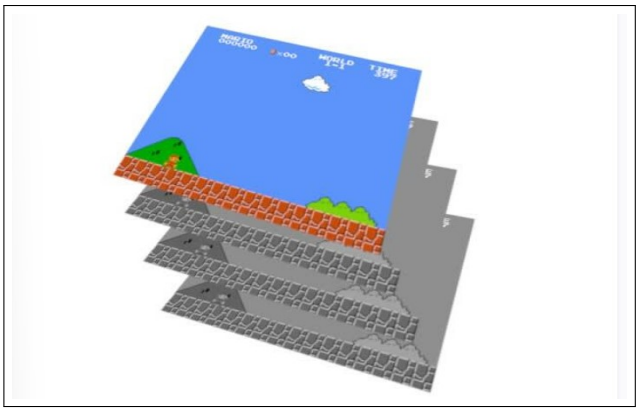

Fig. 8. Game logic for the Mario-Bros

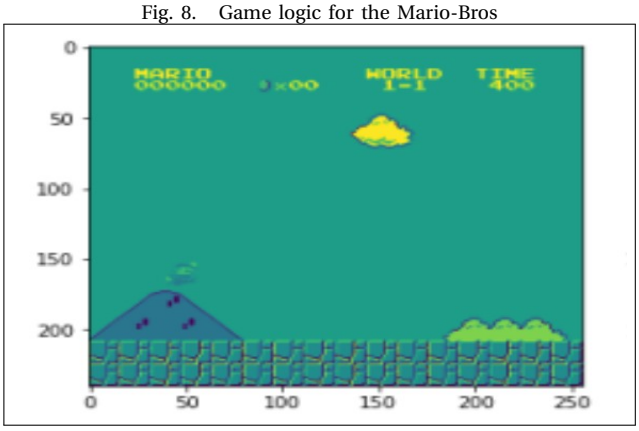

Fig. 9. Game logic for the Mario-Bros

Algorithm 6: Sequential Barnes-Hut Algorithm

```
foreach idx in range(state.shape[3]) do
    plt.subplot(1,4,idx+1)
    plt.imshow(state[0][:,:,idx])
plt.show()
```

G. Preparing the Agent

First, we need to instantiate the environment. Here, we use the first level of Super Mario Bros, SuperMarioBros-1-1-v0 as well as a discrete event space with RIGHT-ONLY action space. Additionally, we use a wrapper that applies frame resizing, stacking and max pooling, reward clipping as well as lazy frame loading to the environment. When the training starts, the agent begins to explore the environment by taking random actions. This is done in order to build up initial experience that serves as a starting point for the actual learning process. After burin = 100000 game frames, the agent slowly starts to replace random actions by actions determined by the CNN policy. This is called an epsilon-greedy policy. It is important to not completely eliminate random actions from the training process in order to avoid getting stuck on locally optimal solutions.

Algorithm 7: Sequential Barnes-Hut Algorithm

```
import torch
import random
import numpy as np
from collections import deque
from game import Direction, Point
from model import Linear-QNet, QTrainer
from helper import plot
MAX-MEMORY = 100-000
BATCH-SIZE = 1000
LR = 0.001
Function train(): is
    plot-scores = []
    plot-mean-scores = []
    total-score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    done = True
    if done : then
        state-old = agent.get-state(game)
        final-move = agent.get-action(state-old)
        reward, done, score =
          game.play-step(final-move)
        state-new = agent.get-state(game)
        agent.train-short-memory( state-old,
          final-move, reward, state-new, done)
        agent.remember(state-old, final-move,
          reward, state-new, done)
        if done then
            game.reset()
            agent.n-games += 1
            agent.train-long-memory()
        if score > record: then
            record = score
            agent.model.save()
        print('Game', agent.n-games, 'Score', score,
          'Record:', record)
        plot-scores.append(score)
        total-score += score
        mean-score = total-score / agent.n-games
        plot-mean-scores.append(mean-score)
        plot(plot-scores, plot-mean-scores)
if –name– == '–main–': then
    train()
```

H. Setting up the Q-trainer

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken.

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken.

The objective of the model is to find the best course of action given its current state. To do this, it may come up with rules of its own or it may operate outside the policy given to it to follow. This means that there is no actual need for a policy, hence we call it off-policy.

1) Important Terms in Q-Learning:

- States: The State, S, represents the current position of an agent in an environment.
- Action: The Action, A, is the step taken by the agent when it is in a particular state.
- Rewards: For every action, the agent will get a positive or negative reward.
- Episodes: When an agent ends up in a terminating state and can't take a new action.
- Q-Values: Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
- Temporal Difference: A formula used to find the Q-Value by using the value of current state and action and previous state and action.

While running our algorithm, we will come across various solutions and the agent will take multiple paths. How do we find out the best among them? This is done by tabulating our findings in a table called a Q-Table.

A Q-Table helps us to find the best action for each state in the environment. We use the Bellman Equation at each state to get the expected future state and reward and save it in a table to compare with other states.[7]

Algorithm 8: Sequential Barnes-Hut Algorithm

```
Function class Linear-QNet(nn.Module): is
    Function –init–(self, input-size, hidden-size, output-size): is
        super().–init–()
        self.linear1 = nn.Linear(input-size, hidden-size)
        self.linear2 = nn.Linear(hidden-size, output-size)
    Function forward(self, x): is
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x
    Function save(self, file-name='model.pth'): is
        model-folder-path = './model'
        if not os.path.exists(model-folder-path): then
            os.makedirs(model-folder-path)
        file-name = os.path.join(model-folder-path, file-name)
        torch.save(self.state-dict(), file-name)
Function class Qtrainer is
    Function –init–(self, model, lr, gamma): is
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
    Function train-step(self, state, action, reward, next-state, done): is
        state = torch.tensor(state, dtype=torch.float)
        next-state = torch.tensor(next-state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.long)
        reward = torch.tensor(reward, dtype=torch.float)
    if len(state.shape) == 1: then
        state = torch.unsqueeze(state, 0)
        next-state = torch.unsqueeze(next-state, 0)
        action = torch.unsqueeze(action, 0)
        reward = torch.unsqueeze(reward, 0)
        done = (done, )
    pred = self.model(state)
    target = pred.clone()
    foreach idx in range(len(done)): do
        Q-new = reward[idx]
        if not done[idx]: then
            Q-new = reward[idx] + self.gamma
            torch.max(self.model(next-state[idx]))
        target[idx][torch.argmax(action[idx]).item()] = Q-new
    self.optimizer.zero-grad()
    loss = self.criterion(target, pred)
    loss.backward()
    self.optimizer.step()
```

I. 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: [8]

1) 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

J. Testing the reinforcement learning model

After Loading up all the environment variables and importing all the reqired packages we can move ahead and test out the game for its Q- values as well as the graph plots which will help us visualize the rewareds and punishments attained by the mario avatar.
TO test out the model we will be needing to execute the following code which comprises of attaining the received input like reward state etc so that we are able to plot the graph at every instant , resetting the environment while we do so.

Algorithm 9: Sequential Barnes-Hut Algorithm

```
done = True
Function Run() is
    if done: then
        action, - = model.predict(state)
        state, reward, done, info = env.step(action)
        env.render()
        state = env.reset()
```

VIII. SIMULATION AND RESULT

Now with all the setting up and making the necessaerty changes to the environment as well as the package directory we can actually go out and run the model / game.

1) Looping through the game: One of the things that we need to make sure is that the mario avatar if or when encounters a punishment state the game should restart it and the mario should be able to play the game again in to order to get rewards so that it is able to make progress and plot the graphs for reinforced learning.
We can do so by executing the followig line with the rest of the looping condition code :

- state = env.reset()

Algorithm 10: Sequential Barnes-Hut Algorithm

```
model = PPO.load('./train/best-model-10000')
state = env.reset()
done = True
Function Run() is
    if done: then
        action, - = model.predict(state)
        state, reward, done, info = env.step(action)
        env.render()
        state = env.reset()
```

After the program execution the Mario Avatar should be able to make random moves and try to overcome the hurdles as well as the enemies. The working render of the MArio automation can be seen in the below picture.

We can also set multiple values of the pretrained models , Like a model that is trained for 100000 steps or 1000000 steps . we can stretch it upto 10000000 that is 10 million steps . At the maximum training the Mario avatar would be able to pass the hurdles very quickly and reach the end.
The procedure to chane the training model for the mario avatar is simply to change the identity of the model that is used to trai the reinforcement learning model for the mario i.e. in the 1st line of the following code:

- You can set the PPO.load model value to your desire . Like 10000 , 100000 , 10000000.

NOTE : The number that you enter into the code if the number of times the mario model has been trained to pass the hurdle.

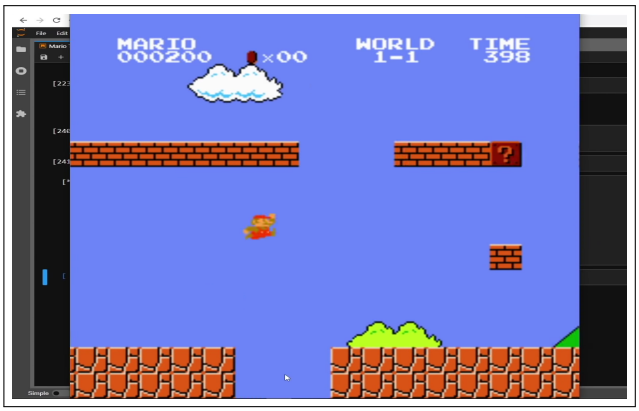

Fig. 10. Game logic for the Mario-Bros

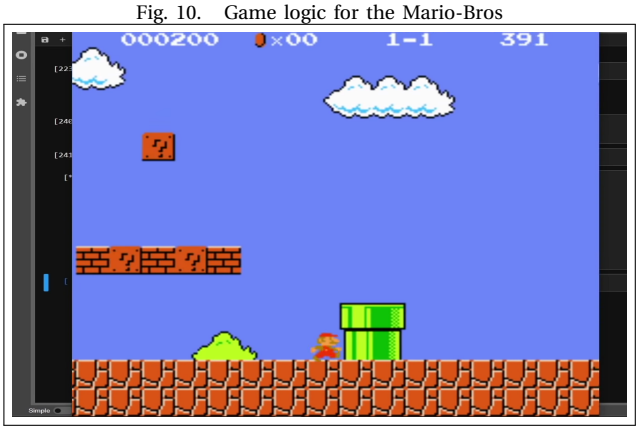

Fig. 11. Game logic for the Mario-Bros

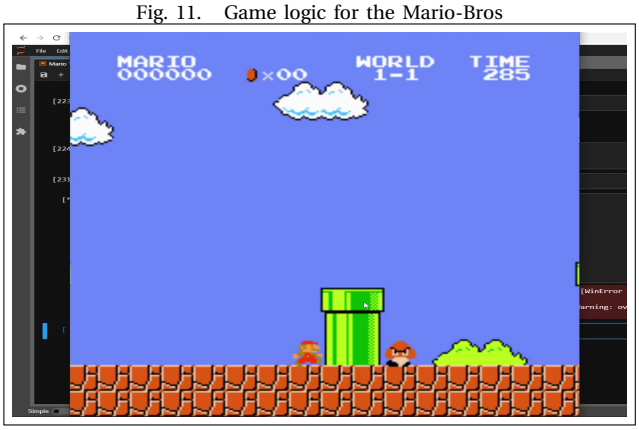

Fig. 12. Game logic for the Mario-Bros

Algorithm 11: Sequential Barnes-Hut Algorithm

```
model = PPO.load('./train/best-model-10000') <===
state = env.reset()
done = True
Function Run() is
    if done: then
        action, - = model.predict(state)
        state, reward, done, info = env.step(action)
        env.render()
        state = env.reset()
```

IX. Review of literature

A. Game Mechanics and the Mario AI Interface

The goal of the game is to control Mario to pass the finish line, and gain a high score one the way by collecting items and beating enemies. Mario is controlled by six keys: UP (not used in this interface), DOWN, LEFT, RIGHT, JUMP, and SPEED. In the game, Mario has three modes: SMALL, BIG, and FIRE. He can upgrade by eating certain items(mushroom and flower), but he will lose the current mode if attacked by enemies. Other than Mario, the world consists of different monsters (Goomba, Koopa, Spiky, Bullet Bill, and Piranha Plant), platforms (hill, gap, cannon, and pipe) and items (coin, mushroom, bricks, flower). The game is over once SMALL Mario is attacked, or when Mario falls into a gap. For more specific descriptions, please see [4] and [1]. When performing each step, the Mario AI framework interface call could return the complete observation of 22 x 22 grids of the current scene, as shown in Figure 1. That is, an array containing the positions and types of enemies/items/platforms within this range. This is the whole available information for our agent. The benchmark runs the game in 24 frames per second. The environment checking functions are called every frame. Therefore, while training we have to train and update the Qtable within 42 milliseconds.

1) Greedy Q-Learning: Q-learning treats the learning environment as a state machine, and performs value iteration to find the optimal policy. It maintains a value of expected total (current and future) reward, denoted by Q, for each pair of (state, action) in a table. For each action in a particular state, a reward will be given and the Q-value is updated by the following rule:

Q(st, at) $\leftarrow$ (1(Alpha)s,a)Q(st, at)+(Alpha)s,a(r+ max(Q(st+1, at+1)))(1)

In the Q-learning algorithm, there are four main factors: current state, chosen action, reward and future state. In (1), Q(st, at) denotes the Q-value of current state and Q(st+1, at+1) denotes the Q-value of future state. (Alpha) [0, 1] is the learning rate, [0, 1] is the discount rate, and r is the reward. (1) shows that for each current state, we update the Q-value as a combination of current value, current rewards and max possible future value.

B. We Choose Q. Learning for two reasons

1) Although we model the Mario game as approximately Markov Model, the specific transitional probabilities between the states is not known. Had we used the normal reinforcement learning value iteration, we will have to train the state transitional probabilities as well. On the other hand, Q-learning can converge without using state transitional probabilities ("model free"). Therefore Q-learning suits our need well.

2) When updating value, normal value iteration needs to calculate the expected future state value, which requires reading the entire state table. In comparison, Q learning only needs fetching two rows (values for st and st+1) in the Q table. With the dimension of the Q table in thousands, Q learning update is a lot faster, which also means given the computation time and memory constraint, using Q table allows a larger state space design.

The learning rate alpha affects how fast learning converges. We use a decreasing learning rate alpha(s),a [5] different for different (s, a) pairs. Specifically,

the equation is chosen based the criteria of proposed by Watkins' original Q-learning paper. He shows the following properties of alpha is sufficient for the Q values to converge.

1) alpha(st, at) $\rightarrow$ 0 as t $\rightarrow$ alpha
2) alpha(st, at) monotonically decreases with t.
3) Sigma of t=1 to infinity (s*t, a*t) = infinity.

One can easily verify the series (2) satisfy all the properties. The discount factor gamma denotes how much future state is taken into account during optimization. We evaluate under several gamma values and chooses 0.6 as the final value. We will show that non-optimal learning parameters lead to highly degenerated performance in the evaluation section. When training our agent, we actually used greedy Q-learning to explore more states. The algorithm is a small variation of Q-learning: each step the algorithm chooses random action with probability or the best action according to the Q table with probability 1. After performing the action, the Q table is updated as in 1.

X. Need and significance

Reinforcement learning is an exciting field in machine learning that offers a wide range of possible applications in science and business likewise. However, the training of reinforcement learning agents is still quite cumbersome and often requires tedious tuning of hyperparameters and network architecture in order to work well. There have been recent advances, such as RAINBOW (a combination of multiple RL learning strategies) that aim at a more robust framework for training reinforcement learning agents but the field is still an area of active research. Besides Q-learning, there are many other interesting training concepts in reinforcement learning that have been developed. If you want to try different RL agents and training approaches, I suggest you check out Stable Baselines, a great way to easily use state-of-the-art RL agents and training concepts.

If you are a deep learning beginner and want to learn more, you should check our brandnew STATWORX Deep Learning Bootcamp, a 5-day in-person introduction into the field that covers everything you need to know in order

to develop your first deep learning models: neural net theory, backpropagation and gradient descent, programming models in Python, TensorFlow and Keras, CNNs and other image recognition models, recurrent networks and LSTMs for time series data and NLP as well as advanced topics such as deep reinforcement learning and GANs.

If you have any comments or questions on my post, feel free to contact me! Also, feel free to use my code (link to GitHub repo) or share this post with your peers on social platforms of your choice. [2]

XI. Performance Comparison Graph Results

For training the Q-learning algorithm, we firstly initialize the Q-table entries with a uniform distribution, i.e. Q U (0.1, 0.1). Then we trained the agent by 15000 iterations on a fixed level for the three Mario modes. The order is: train the level/episode 20 times for small Mario, 20 times for large Mario, 20 times for fire Mario, repeat, etc. We believe in this order the fire Mario will be able to use the Q table information from previous runs even when he is attacked and downgraded. During training the learning parameters are:

$$(Alpha)(st,at) = \frac{(Alpha)0}{oftimesatperformedinst} \quad (1)$$

- (Alpha)0 = 0.8
- = 0.6
- (Theta) = 0.3

For every 20 training episodes of each mode, we evaluate the performance by running 100 episodes on the current Q table and use the average metrics as the performance indicators. The evaluation episodes are run with = 0, = 0.01, so the learning is turned off and random exploration is minimal. It is purely a test of how good the policy is. We use 4 metrics to evaluate performance

- A composite "score" combining weighted win status, kills total, distance passed and time spent.
- The probability the agent beats the level.
- The percentage of monster killed.
- The time spent on the level.

Figure. 2 shows the learning curves of evaluation score using the previously described optimized parameters. Thediscount factor, , is 0.6, meaning that the Q-learning algorithm tried to maximize the long-term reward. Obviously, our algorithm demonstrates a learning curve and quickly converges to the optimal solution after about 3000 training iterations. At the end of training cycles, our average evaluation score is around 8500. For some evaluation cycles, we can even achieve 9000 points, which is nearly the highest score human can achieve in one episode. In order to show the generalization, we also tested the

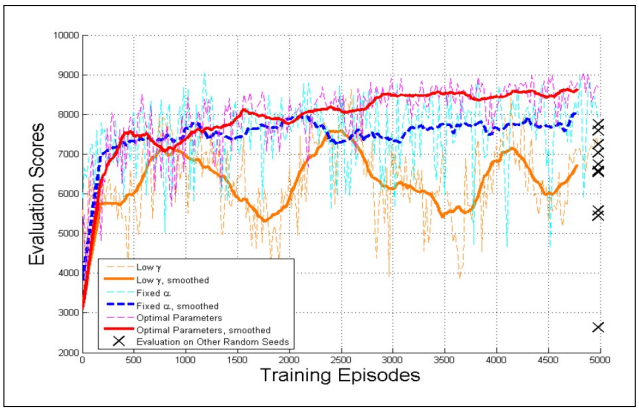

Fig. 13. Game logic for the Mario-Bros

trained Q-learning algorithm with random episode seeds. The results show that for most random seeds, our trained algorithm performs reasonably good. The reason that one test performs bad is that there always be some unknown situations, to which Mario is not trained. In addition, we plot the learning curves with fixed learning rate (= 0.15 throughout training) and low discount factor (= 0.2). As discussed above, in our training algorithm, we keep decreasing the learning rate. The figure indicates that with a fixed learning rate, when it converges, the converged solution is not optimal and the variance in scores is larger. A low discount factor means that the learning algorithm maximizes the short-term reward. In our learning algorithm, we gave positive reward for right movement and negative reward for left movement. If the algorithm tries to maximize short-term reward, Mario will always move the right. However, in some situations, Mario should stay or go left to avoid monsters. In this case, the short-term reward maximization is not the optimal solution. Figure. 3 shows the winning probability learning curve. As the early stage of learning process, the winning probability is as low as 0.3. With the increase of training cycles, the winning probability increases and converges to around 0.9. For the low learning, even the average probability is increasing but the variance is very high. For the fixed learning, the learning curve converges but the converged value is not optimal. The curve demonstrates that our trained agent is consistently good on beating the level! Figure. 4 shows the percentage of monster killed. Since we generated the training episode by the same seed and setup, the total number of monsters within a episode is the same over training. At the beginning, the Q-values are generated randomly. Therefore, the killing percentage is very low. The learning curve shows fast convergence of the killing probability within a few training episodes, due to the high reward we gave. There are two reasons that we give high reward for killing. Firstly, the killing action is given high score in evaluation and therefore, we can achieve a high score in evaluation. Secondly, the Mario is safer when he kills more monsters. In this figure, the learning curve with fixed shows a similar

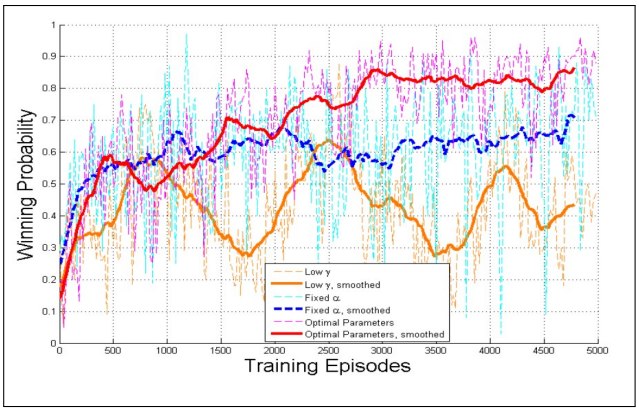

Fig. 14. Game logic for the Mario-Bros

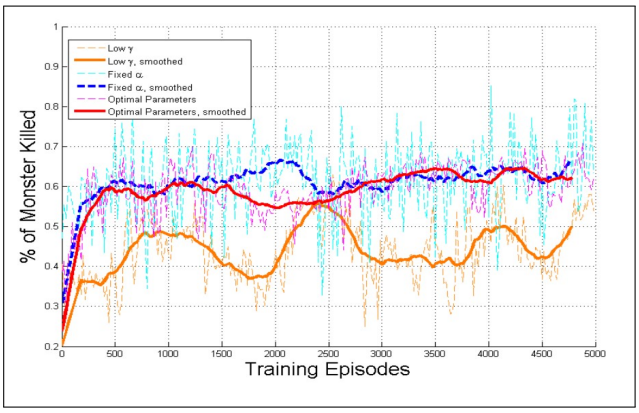

Fig. 15. Game logic for the Mario-Bros

performance to our optimal learning curve. The learning curve with low has very low mean and high variance. Figure. 5 shows the time spent for the Mario to pass a episodes successfully. The optimal learning curve shows a fast convergence within 250 iterations. The learning curve with low has a similar performance as the optimal learning curve. The reason is that the short-term reward maximization forces the Mario to keep moving rightward. However, as we discussed above, it is not optimal since the Mario will collide creatures with high probability. The learning algorithm with fixed needs more time to win because the converged policy is not the best.

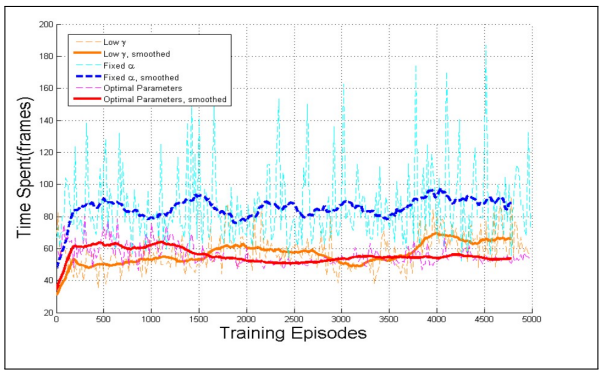

Fig. 16. Game logic for the Mario-Bros

XII. Conclusion

In this project, we designed an automatic agent using Q-learning to play the Mario game. Our learning algorithm demonstrates fast convergence to the optimal Q-value with high successful rate. The optimal policy has high killing rate and consistently beat the level. In addition, our results show that the state description is general enough, that our optimal policy can tackle different, random environments. Further, we observe that long term reward maximization overperforms short term reward maximization. Finally, we show that using decaying learning rate converges to a better policy than using fixed learning rate.

Our RL approach could be extended in a number of ways. For example, in our learning algorithm, we did not design the state for the Mario to grab mushroom and flowers. In addition, our algorithm focuses on optimizing the successful rate. Possible future work may include, but is not limited to:

- Extend our state to allow grabbing coins and upgrading.
- Vary the reward function to realize different objectives (e.g. killer-type Mario)
- Make the state design more precise to cope with the position rounding problem.
- Explore other RL approaches such as SARSA[6] and fuzzy-SARSA[7] to reduce state space and increase robustness.

References

[1] J. Bergdahl, C. Gordillo, K. Tollmar and L. Gisslén, "Augmenting Automated Game Testing with Deep Reinforcement Learning," 2020 IEEE Conference on Games (CoG), 2020, pp. 600-603, doi: 10.1109/CoG47356.2020.9231552.

[2] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in Proceedings of the IEEE Congress on Evolutionary Computation, Citeseer, 2010.

[3] R. Sutton and A. Barto, Reinforcement learning: An introduction, vol. 1. Cambridge Univ Press, 1998.

[4] J. Tsay, C. Chen, and J. Hsu, "Evolving intelligent mario controller by reinforcement learning," in Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on, pp. 266–272, IEEE, 2011.

[5] L. Jouffe, "Fuzzy inference system learning by reinforcement methods," Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, vol. 28, pp. 338 –355, aug 1998.

[6] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," tech. rep., 1994.

[7] Barnes, J., & Hut, P. (1986). A hierarchical O (N log N) force-calculation algorithm. nature, 324(6096), 446.

[8] Christian Kauten (2018). Title :: Super Mario Bros for OpenAI Gym [Online]. Available: `https://github.com/Kautenja/gym-super-mario-bros`. [Accessed: 07- Jan- 2018].

[9] @126522 (2022-08-11). Title :: Reinforcement learning in Super Mario bros [Online]. Available: `https://devpress.csdn.net/python/62f51e037e6682346618a0ad.html`. [Accessed: 07- Jan- 2018].

[10] Sebastian Heinz7 (2019). Title :: Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow [Online]. Available: `https://www.statworx.com/en/content-hub/blog`. [Accessed: 07- Jan- 2018].