Playing Super Mario Bros. with Reinforcement Learning

Sohum Padhye · Follow
9 min read · Nov 28, 2022

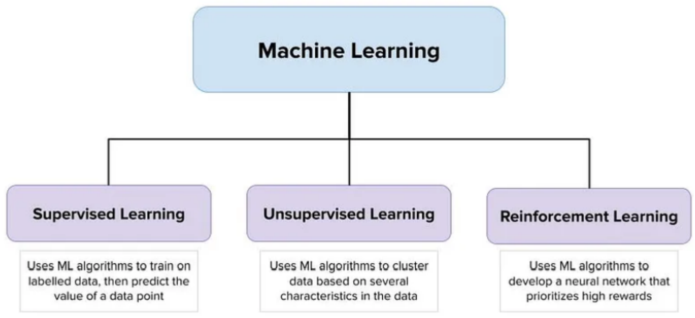
After posting my previous article on neural networks, I decided to start looking into reinforcement learning, a subset of machine learning that caught my eye. In this article, I will go through my experience of training a reinforcement learning agent to play Super Mario Bros. I have used "Train a Mario-playing RL Agent" as a guide to learn how double Q networks, a type of reinforcement learning work, as well as how they can be applied to this use case.

Overview:

- **How Reinforcement Learning Works**
- **Deep Q Learning & Double Deep Q Networks (DDQNs)**
- **MarioNet**
- **Running the MarioNet**
- **Challenges**
- **Changes Made to the Code**

How Reinforcement Learning Works

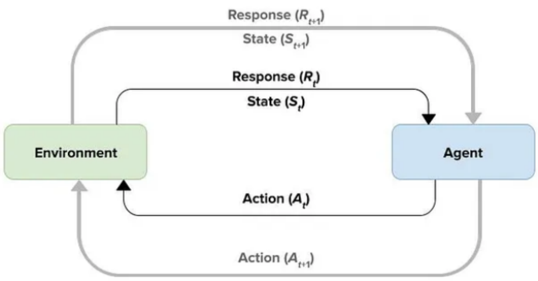
Machine learning can be broken up into three categories: supervised learning, unsupervised learning, and reinforcement learning.


Machine Learning
Supervised Learning
Uses ML algorithms to train on labelled data, then predict the value of a data point
Unsupervised Learning
Uses ML algorithms to cluster data based on several characteristics in the data
Reinforcement Learning
Uses ML algorithms to develop a neural network that prioritizes high rewards

In supervised learning, the model is given a training dataset with data that contains the input and which output (label) it ties to. It learns from this data to understand how to derive the output from the input. Once it has been trained, it goes onto a testing dataset that has inputs the model has never seen before. Then, the model must try to predict the label each data point should be given.

In unsupervised learning, the model is given a dataset, but none of the data points have labels. Instead, it is given data points and a number of clusters to group them in. As the number of clusters changes, so does the algorithm's output. This type of learning is used to discover patterns in given data.

Reinforcement learning is different than these other two training methods:



The t means the step (or time) that's being processed. In one step, the environment gives the agent a response (either a reward or penalty) to the agent's actions, along with the current state. With this information, the agent determines which action to take.

Its training is unsupervised, but instead of trying to group data points, it attempts to gain a reward by performing favourable actions in an environment, while trying to avoid making errors which penalize it (R_t).

The terms below are important to understand when it comes to reinforcement learning:

- **Agent:** an agent is a neural network that learns by interacting with its environment and understanding which actions lead to good outcomes and bad outcomes
- **Environment:** an environment is a world in which the agent interacts and learns
- **Action:** an action is what the agent, given its circumstances (state), decides to do
- **State:** a state is a capture of what is in the environment at a given moment
- **Response:** a response is something that is given to an agent when it does something beneficial or harmful toward its next state

Deep Q Learning & Double Deep Q Networks (DDQNs)

Deep Q learning is a type of reinforcement learning that uses deep neural networks. It uses a Q (or quality of state) value, which helps it determine whether its actions were beneficial or harmful to the environment's state.

A deep Q network (DQN) is a multi-layered neural network that for a given state outputs a vector of action values based on the weights and biases of the network. For an *n*-dimensional state space and an action space containing *m* actions, the neural network is a function from *R^n* to *R^m*.

The Q-learning algorithms are known to overestimate action values under certain conditions, which could affect their performance. Double deep Q-learning, is a specific adaptation of the DQN algorithm that not only reduces the overestimations but also leads to improved performances. The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation.

A detailed understanding of deep Q learning and double deep Q networks (DDQNs) involves complex mathematics. I recommend you read this paper to have a better understanding.

MarioNet

I want to acknowledge that all the learning here is my reflection on this tutorial named "Train A Mario-Playing RL Agent". I am thankful to Yuanson Feng, Suraj Subramanian, Howard Wang, and Steven Guo for making this tutorial.

Context

In the context of training an agent to play Mario:

- The agent is Mario.
- The environment is Mario's world — it contains enemies, obstacles, and powerups, such as tubes, mushrooms, etc.
- The action is what Mario (the agent) decides to do.
- The action space is all of the possible actions that Mario can carry out: running right, jumping right, running left, and jumping left. In this case, I have restricted Mario to right movement only, for faster training.
- The state includes several things: Mario's position, where the obstacles, powerups, and enemies are, Mario's current score, and the direction that everything is moving in.
- The state space is a set of all of the possible states in the environment.
- The reward is the feedback from the environment to the agent, which drives Mario to learn and change his future action.
- The return is a cumulative reward over the course of several play-throughs or episodes.
- The quality of action in a state is the expected return for the given state and action.

Preprocessing the Environment

Before we start training the neural network, we have to optimize the environment so that the training is not as computationally heavy.

Grayscaling: The size of the environment is a 3x240x256 window, where the 3 represents the RGB channels, and the 240x256 represents the dimensions of the window. However, the colour of each object in the state doesn't really contribute to the final action made by the agent. The fact that Mario is wearing red or that the bricks are brown won't change the agent's actions. So, instead of having a computationally heavy 3x240x256 window to process, we can turn the entire image into grayscale, so now it's just 1x240x256 values.

Resizing: Next, it's probably not the easiest for the computer to process a 240-by-256-pixel window at 15 frames per second (more on this in a moment). So, we can resize the image that the agent is taking in, to 84 pixels by 84 pixels.

Frame Skipping: Finally, it doesn't add much value to make our agent process every single frame, because by looking at consecutive frames the agent doesn't gain much information because consecutive frames contain very similar states. Instead, we can skip a given number of intermediate frames without losing much information.

Frame Stacking: Frame stacking is used to combine consecutive frames into one consolidated input for the learning model. Using this, it's easier to identify what actions previously occurred by looking at the given frame.

In this case, we are stacking four consecutive, grayscaled, resized frames, giving us a size of 4x84x84. This is the size of our response for every action that Mario takes.

Defining the Agent

Mario (our agent) needs to be able to act, remember, and learn.

Act: The action of the agent is based on the current state of the environment and the optimal action policy. In every state, Mario either performs an action (explore) or uses his neural network, MarioNet, to perform the action for him (exploit). Mario decides whether to explore or exploit based on his exploration rate.

At the beginning of training, the exploration rate is set to one, which means that Mario will definitely do a random action. Then, as each stacked frame goes by, the exploration rate is reduced by a number called the exploration rate decay, which slowly introduces Mario to exploit by using his neural network rather than explore with random actions.

Remember (cache and recall): Mario remembers his experiences based on the current state, the reward, and the next state. For each action, Mario caches his experiences (stores them in memory). Then, he recalls (randomly samples a batch of experiences) from the memory cache, and uses it to learn how to play the game better.

Learn: Over time, Mario needs to be able to use his experiences to improve his actions (or action policy). To accomplish this task, we are using the DDQN algorithm. In this case, the DDQN uses two convolutional neural networks that approximate the optimal action-value function. The temporal difference (TD) method is used to compute the TD_estimate and TD_target values to optimize the neural network parameters.

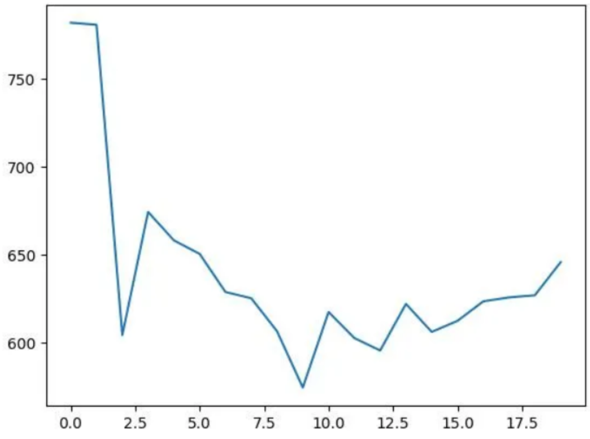
Running the MarioNet

I downloaded the code and reviewed each line of the file to understand how it works. The code was extremely well-commented. It allowed me to learn how all of the mathematical concepts in double Q learning were translated into code and enabled me to get more acquainted with the PyTorch framework, Torch library, and various associated libraries.

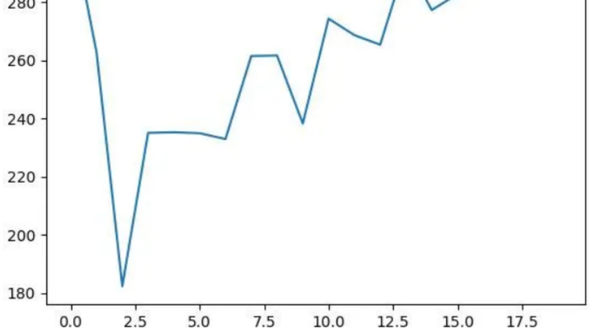
Running the network and observing how the network works was a lot of fun. However, I wasn't able to fully train the model because that would take 40,000 episodes.

It took me roughly nine minutes to complete 100 episodes on my laptop. At that rate, it would have required about 60–70 hours nonstop to train 40,000 episodes.

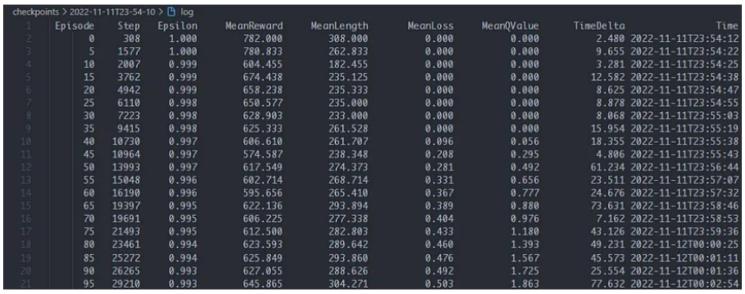
Below, I have put some output graphs and logs that were generated throughout the training. By looking at this graph, I could observe how neural networks learn progressively. Note that during the first few episodes, Mario's actions were completely random. It was only around episode 40 (8 on the graph) that Mario started exploiting his neural network.



A plot of the rewards throughout episodes. The x-axis values multiplied by five are the number of episodes. The y-axis is the mean reward through each of those five episodes.



A plot of the mean length for every five episodes. The y-axis represents the time, in ticks, of each episode.



```
checkpoints > 2022-11-11T23-54-10 > log
   Episode    Step  Epsilon     MeanReward     MeanLength       MeanLoss     MeanQValue      TimeDelta                Time
         0     308    1.000        782.000        308.000          0.000          0.000          2.480 2022-11-11T23:54:12
         5    1577    1.000        780.833        262.833          0.000          0.000          9.655 2022-11-11T23:54:22
        10    2007    0.999        604.455        182.455          0.000          0.000          3.281 2022-11-11T23:54:25
        15    3762    0.999        674.438        235.125          0.000          0.000         12.582 2022-11-11T23:54:38
        20    4942    0.999        658.238        235.333          0.000          0.000          8.625 2022-11-11T23:54:47
        25    6110    0.998        650.577        235.000          0.000          0.000          8.878 2022-11-11T23:54:55
        30    7223    0.998        628.903        233.000          0.000          0.000          8.068 2022-11-11T23:55:03
        35    9415    0.998        625.333        261.528          0.000          0.000         15.954 2022-11-11T23:55:19
        40   10730    0.997        606.610        261.707          0.096          0.056         18.355 2022-11-11T23:55:38
        45   10964    0.997        574.587        238.348          0.208          0.295          4.806 2022-11-11T23:55:43
        50   13993    0.997        617.549        274.373          0.281          0.492         61.234 2022-11-11T23:56:44
        55   15048    0.996        602.714        268.714          0.331          0.656         23.511 2022-11-11T23:57:07
        60   16190    0.996        595.656        265.410          0.367          0.777         24.676 2022-11-11T23:57:32
        65   19397    0.995        622.136        293.894          0.389          0.880         73.631 2022-11-11T23:58:46
        70   19691    0.995        606.225        277.338          0.404          0.976          7.162 2022-11-11T23:58:53
        75   21493    0.995        612.500        282.803          0.433          1.180         43.126 2022-11-11T23:59:36
        80   23461    0.994        623.593        289.642          0.460          1.393         49.231 2022-11-12T00:00:25
        85   25272    0.994        625.849        293.860          0.476          1.567         45.573 2022-11-12T00:01:11
        90   26265    0.993        627.055        288.626          0.492          1.725         25.554 2022-11-12T00:01:36
        95   29210    0.993        645.865        304.271          0.503          1.863         77.632 2022-11-12T00:02:54
```

A log which shows the step number, the exploration rate, the mean reward, length, loss, and Q value of the five episodes, the time it took to go through those five episodes, and the time that they completed.

A snippet of the MarioNet undergoing training.

Challenges

I faced several challenges throughout the creation of the MarioNet:

Loading and Saving the Model

While experimenting with the program, I quickly realized that every time I ran the application, it started from scratch and didn't save the last trained state of the neural network. So, I needed to find a way to train the network over multiple runs so I wouldn't have had to overload the computer's CPU for 60 hours. In PyTorch, there are ways that you can save and load the model. While the saving was properly implemented in the code, I found that there wasn't really any code for loading previously saved neural network states at the start of the program.

Another thing that I wanted to change was the fact that every time I ran it, it started at episode 1. I wanted to keep track of the total number of episodes over the several times that I ran it rather than having to tally it all up at the end to see how many episodes it took to get there.

I ended up modifying the code so that:

1. At the end of the program, the state of the neural network, the epsilon (exploration rate) value, and the episode number were saved

2. At the start of the program, the previously saved state of the neural network, the epsilon value, and the episode number were retrieved so the training could start from that point onwards.

CPUs and GPUs

Essentially, when you install PyTorch you can take it to be with compute unified device architecture (CUDA) support, or with just the CPU. The problem for me was that training with the CPU was slow, but whenever I tried to train with the GPU it would run out of memory. The reason for the GPU running out of memory was that it kept filling its cache to the point where it got full, but it never emptied any data out of it while training. I'm still working on this problem.

Final Thoughts

I am thankful to Yuanson Feng, Suraj Subramanian, Howard Wang, and Steven Guo for making such an amazing tutorial that was easy to follow and taught me so much about reinforcement learning in the process. I got to learn so much more about using PyTorch, OpenAI Gym (which is the dependency that provided these awesome environments to train my neural network), and this motivates me to work even more and create my own applications of reinforcement learning.

Thank you for reading my article, and I'll catch you in the next one!

Reinforcement Learning · Deep Neural Networks · Double Q Learning · Artificial Intelligence · Machine Learning

33 · 2

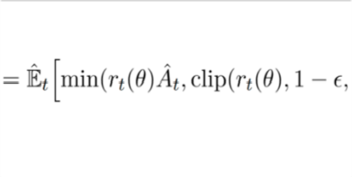
Written by Sohum Padhye

Follow

85 Followers

Passionate about AI and Web Dev.

More from Sohum Padhye



$= \hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon,$

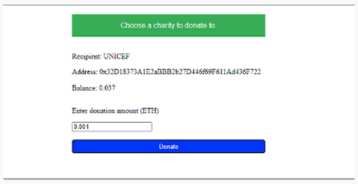
Sohum Padhye

Building a Reinforcement Learning Agent that can Play Rocket League

In this article, I'll show you how I made a reinforcement learning agent that could lear...

12 min read · Mar 25, 2023

50



Choose a charity to donate to.

Recipient: UNICEF

Address: 0x32D18373A1E2aBBB2b27D446f69F611Ad436F722

Balance: 0.037

Enter donation amount (ETH)

0.001

Donate

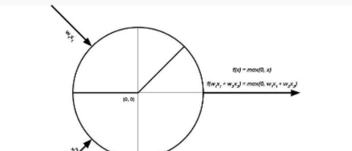
Sohum Padhye

Creating a Web3 Charity Application

In my previous article, I went over how I created a GIF portal on the Solana blockchai...

10 min read · Jun 26, 2022

75 · 1



Sohum Padhye

Neural Networks — The Underlying Technology of Artificial…

Everything from the fundamentals of tensors and perceptrons, to how deep neural…

12 min read · Nov 1, 2022

59 1

Sohum Padhye

Building a GIF Portal on Solana

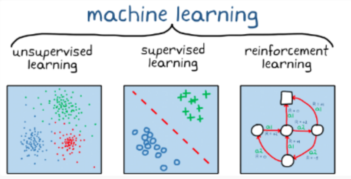
Read about how I created a GIF portal on the Solana blockchain with Rust!

9 min read · May 22, 2022

59

See all from Sohum Padhye

Recommended from Medium



Nicholsonjm

VAEs in Reinforcement Learning

In the realm of machine learning, Variational Autoencoders (VAEs) and Reinforcement…

16 min read · Feb 27, 2024

4

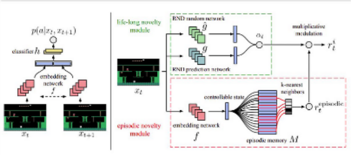


Figure 1: (left) Training architecture for the embedding network (right) NGU's reward generator.

Chadha Sridi

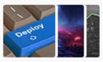
Exploration strategies in Reinforcement Learning

Reinforcement learning is a type of machine learning where an agent learns to make…

14 min read · May 5, 2024

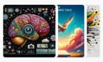
7

Lists



Predictive Modeling w/ Python

20 stories · 1252 saves



Natural Language Processing

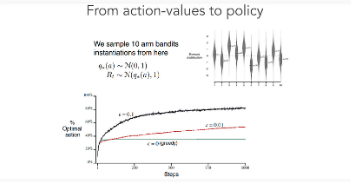
1489 stories · 1003 saves

AI Regulation

6 stories · 472 saves

Practical Guides to Machine Learning

10 stories · 1507 saves



Amansinghalml

Multi-armed bandits

Are the number of possible arms in multi-armed bandit discrete or continuous?…

3 min read · Mar 30, 2024

Henry Wu

Intro to Reinforcement Learning: Monte Carlo to Policy Gradient

This post is an intro to reinforcement learning, in particular, Monte Carlo methods,…

16 min read · Feb 15, 2024

196

Wouter van Heeswijk, Ph... in Towards Data Scien...

Proximal Policy Optimization (PPO) Explained

The journey from REINFORCE to the go-to algorithm in continuous control

✦ · 13 min read · Nov 30, 2022

326 5

Dagang Wei

Demystifying Q-Learning

Estimating the Long-Term Reward of an Action in a Given State

7 min read · May 13, 2024

See more recommendations