# Self-Learning Mario with DDQN

learn it i will need your help "Demo & Intro
0:00
hey what's up guys I have something that will absolutely blow your mind this is
0:05
the computer playing Super Mario Bros and guess what the computer has no knowledge about the internal State or
0:11
the mechanics of the video game rather it's simply looking at the frames and then deciding what buttons to press
0:18
exactly how humans do it impressive right well if you want to learn how to
0:24
implement this stay tuned because that's exactly what I'll be sharing with you guys today and I cannot wait
0:30
this was done using a subclass of machine learning algorithms called reinforcement learning reinforcement
0:36
learning is the coolest thing ever in fact it's been used to train drones and
0:41
robots find more efficient ways for matrix multiplication and even make the training process for other machine
0:48
learning algorithms faster oh and not to mention it was also used to train
0:53
everyone's favorite chatbot chat GPT reinforcement learning is pretty different from its cousin's supervised
0:59
and unsupervised learning in supervised learning the model is given a data set
1:04
where each point has a correct answer associated with it whether it's categorical or numerical the model's job
1:11
is then to predict these values accurately an unsupervised learning the data points
1:17
don't have a correct answer but rather the models tasked with extracting General Trends or insights reinforcement
1:24
learning is its own Beast the objective of reinforcement learning is for the agent to maximize its rewards inside of
1:31
the environment we give the agent a positive reward whenever it exhibits behaviors we want
1:36
to persist and negative ones for behaviors that we don't want it's kind of like teaching a dog new

tricks
1:42
another thing to note is that in supervised and in unsupervised learning you must have the data pre-collected
1:49
before you feed it to the model whereas in reinforcement learning we'll see today our agent go out into the world
1:55
collect its own experiences and then learn from it it's a completely different Paradigm when it comes to
2:01
training I wanted to share an implementation of the ddqn algorithm or the double deep Q Network I'm not going
2:09
to dive into the nitty-gritty of reinforcement learning theory I'd rather just share a more intuitive
2:14
understanding for those that have a base level understanding of machine learning but haven't really been exposed to
2:19
reinforcement learning before however for those that are interested I'm going to leave some of my favorite
2:25
reinforcement learning resources down in the description below along with the paper that originally pioneered the
2:31
double deep Q Network algorithm I first implemented this algorithm when I was a sophomore in college and I still
2:37
remember the excitement that I felt when I first watched the model start to train learn and eventually beat the level
2:44
I was absolutely awestruck I mean I understood all of the math and each line of code that I wrote to implement the
2:50
algorithm however there's just something that's still magical about reinforcement learning that gets me excited every
2:56
single time and my goal today is to be able to package that excitement and share it with you guys through this
Key Reinforcement Learning Vocabulary
3:02
video let's start off with the basics in order to understand the fundamentals of reinforcement learning we need to Define
3:08
these following terms let's understand each of them in the context of Super Mario Bros first is the agent the agent
3:17
today will be our neural network that's controlling Mario it's the one responsible for making decisions and
3:23
taking the optimal action the environment is what our agent will be interacting with it's going to be

level
3:30
one one of Super Mario Bros on the NES a state is a snapshot of the environment
3:36
at any given time it's essentially what our agent sees today we'll be using four
3:41
consecutive frames from the game the reason we're using consecutive frames is so our agent can see motion for example
3:49
is the Goomba coming towards us or away from us well for us to know we need to
3:54
know the previous frames oh well hopefully our agent knows to jump actions are the inputs that the
4:01
agent has available to it to give to the environment today the agent will have access to these five button combinations
4:09
to press first one is with no buttons pressed and then the agent is able to move right a is used to jump and the
4:16
longer it's held the higher Mario will jump and then B is used to move faster
4:22
once the environment gives the agent a state and the agent responds with an
4:27
action the environment also gives the agent a reward a reward essentially
4:33
helps the agent determine whether or not the action it took inside of that state was good or bad today we'll be giving
4:40
the agent a plus one reward for each unit to the right that it makes in the video game it'll receive a negative one
4:46
reward for each second that passes by in the games clock this is to disincentivize the agent from standing
4:52
still and then the agent will get a massive negative 15 points if it dies
4:59
now the concept of a negative reward may seem foreign to some however remember the goal of the rewards is to guide the
5:06
agent to Optimal actions and negative rewards still help the agent do just
5:11
that everything that we've talked about so far is summarized in this diagram it
5:17
essentially outlines how the agent and the environment interact with one another the agent gives the environment
5:23
actions and the environment returns States and rewards an episode for us
5:29
today will just be one attempt at the level an episode will end whether Mario dies reaches the flag or Runs Out of
5:37

Time a policy is a function that takes in a state and returns an action it's
5:43
what our agent will use to make its decisions some reinforcement learning algorithms will have the policy be a
5:50
probability distribution however today in our ddqn algorithm implementation
5:56
we'll be using the Epsilon greedy approach more on that later a value function takes in a state and returns
6:03
how valuable that state is in reinforcement learning it's natural to
6:08
view some states as more valuable than others for example the state where you're right next to the flag is
6:15
probably a little bit more valuable than the one where you're falling down a hole we're not going to be working directly
6:20
with the value function today however I thought I'd mention it because a lot of other reinforcement learning literature
6:26
brings it up an action value function takes in a state and an action and then
6:33
returns how valuable this pair is as you can see on the screen this state action
6:39
pair is probably more valuable than this state action pair I'm sure you can see why approximating this function
6:44
accurately today is our goal because it's going to let our agent take the
6:49
action inside of in state that has the highest value alright quick review we're
6:55
using reinforcement learning to train the computer to play Super Mario Bros our agent or the neural network that's
7:01
controlling Mario will receive a state from the environment remember a state is
7:06
four consecutive frames from the video game the agent will then take an action
7:11
if the action was good we'll give it a positive reward if the action was bad it will give it a negative reward and then
7:18
the agent will use these rewards to train itself over time time to take better and better actions eventually
7:25
we're hoping that the agent will learn how to beat the level and reach the flag that's waiting for it at the end that
7:32
was a lot to take in thank you guys for bearing with me we have three more Concepts to cover first is the Epsilon
7:37

greedy approach second is the replay buffer and third are some more details about the action value function after
7:44
that we'll assemble the algorithm and then implement it in code the Epsilon greedy approach is the strategy our
Epsilon-Greedy Approach
7:50
agent will be using today to choose its actions however to understand Epsilon greedy we first must take a step back
7:57
and understand the explore exploit dilemma a problem that's at the heart of reinforcement learning let's say our
8:04
agent has learned that jumping over Koopas is the most effective way to not die good I mean I guess it gets the job
8:11
done right however as I'm sure my fellow Gamers know that jumping on a Koopa
8:17
actually turns its shell into a weapon that can be used to eliminate other enemies if our agent constantly exploits
8:24
or takes the action that it deems to be the best in the current moment then it will never get a chance to explore other
8:30
potentially Superior strategies so how do we navigate this explore exploit dilemma enter the world of the Epsilon
8:36
greedy approach a clever solution to the explore exploit dilemma with probability Epsilon our agent Ventures into the
8:43
unknown taking random actions and with the remaining probability 1 minus Epsilon it'll take the action that it's
8:49
confident is the best one it's essential to note that this best action of the agent is dependent only on what it knows
8:55
in the current moment it's not necessarily the best action overall initially when our agent knows nothing
9:01
about the environment we'll set Epsilon to 1 ensuring maximum exploration and
9:06
then over time as our agent continues to learn about the environment and its Dynamics will start to taper off Epsilon
9:13
from one to a very small non-zero number we'll always keep Epsilon greater than zero just so our agent is always on the
9:20
lookout for new strategies the this means that if our agent did learn to jump over Koopas we're hoping that with
9:26
the Epsilon greedy approach it'll eventually randomly jump on one and then realize the benefits of

doing so now
Replay Buffer
9:33
let's cover the replay buffer it's essentially a storage for our past experiences and rewards it'll be used to
9:39
train the neural network there are a couple of reasons why we use a replay buffer rather than training on the
9:44
experiences as we collect them the first is because sequential experiences have a lot of correlation which can lead to a
9:52
lot of instability when training the model to eliminate this instability we randomly sample from our replay buffer
9:58
and the second reason is that we're now able to reuse data which actually improves data efficiency in the replay
10:05
buffer we'll be storing tuples of the state action taken reward received next
10:11
state and a Boolean flag indicating if the episode is done for any reason we'll see how each of these come into play
10:17
when we sample the replay buffer to train our Network now let's pull up the action value function that we mentioned
Action-Value Function Intuition
10:23
earlier this is often referred to as the Bellman equation remember our primary objective today is to approximate this
10:30
function enabling our agent to make optimal decisions in any given State the goal isn't just about seeking immediate
10:36
gratification instead the agent aims to maximize its rewards over the long run humans intuitively do this as well for
10:44
example we may take a slightly longer route with less coins if it leads us to an item box at the end that contains the
10:50
invincibility star math notation is scary so let me translate this equation to English the value of taking action a
10:58
in state s is equal to the reward you get in the next time step plus the value
11:03
of taking the best action in the next state S Prime multiplied by a discount
11:09
Factor the discount Factor makes rewards from future States less valuable in
11:14
reinforcement learning this is important because future rewards are less predictable than the current one due to
11:20

the environment's stochastic in the Stanford marshmallow experiment children were given a marshmallow and
11:26
were told that they could either eat it right now or wait 15 minutes for a second marshmallow it was to measure
11:33
their self-control when it came to delayed gratification essentially the researchers were measuring the
11:39
children's Gamma or their discount Factor one marshmallow now or two in the
11:44
future our agent is kind of in a similar situation except for the children their
11:50
second marshmallow was guaranteed unfortunately our agent does not have that luxury as such it's ideal for it to
11:58
value a guaranteed marshmallow in the present more than a potential marshmallow in the future a gamma of one
12:05
means that there is no discounting while a gamma of zero means the agent is completely myopic we want neither but
12:13
rather to strike a balance between the two this action value function is recursive in nature this means that the
12:20
value of a specific State action pair is not just based on the immediate reward rather it also depends on the value of
12:27
the subsequent State action pair dive deeper and you'll see that this subsequent value in turn depends on its
12:33
immediate reward and the value of the next state action pair and the chain continues we assume that we take the
12:39
optimal action the action with the highest Q value at each recursive level thus the max function when you unravel
12:46
this recursive structure you'll find that the value of any state action pair is essentially the sum of its immediate
12:52
reward and the accumulated discounted feature rewards until the end of the episode this first equation once you
12:59
assume you're taking the best action at each time step condenses down to this after you distribute the Gammas
13:05
appropriately the further out a reward is the more times it's multiplied by gamma still a little bit confused don't
13:12
worry I was too when I first learned this let's try to tackle this from my more visual approach take each ribbon on

13:18
the screen to be a reward the agent will get until it reaches the flag remember that these rewards are granted after
13:24
each step the agent takes in the environment and the number on the screen is arbitrary the ribbons are getting
13:30
progressively smaller to signify the discounting assuming the agent is currently in state s and takes action a
13:36
the value of this state action pair is the sum of this entire sequence of rewards discounted appropriately if S
13:44
Prime is the next state after taking action a in state s and a prime is the
13:49
best possible action in S Prime then the value of S Prime a prime will be this
13:54
sum again discounted appropriately we know the reward R we got from going from
13:59
s to S Prime so we can use that to help tune the estimate of the current state action pair's value to train our agent
14:07
We compare its predicted value of a state action pair to the Target value remember the target value is the
14:13
immediate reward from a move plus the disc discounted value of the best possible next move essentially We
14:19
compare its predicted value of the sum of these ribbons to the predicted value of the sum of these ribbons plus the
14:26
ribbon whose value we know with certainty because the agent just received that reward while both the
14:32
prediction and the target are estimates the target is considered to be more reliable why is that the case well
14:39
because the target is based on fewer future assumptions as a component of the target is the reward directly observed
14:45
from the agent's recent action making it grounded in immediate experience rather
14:50
than speculative projection once we have our predicted value and our Target value we can use the following equation to
14:57
update our estimate for those of you familiar with gradient descent you might recognize this to be exactly that Alpha
15:04
is our learning rate or the size of the steps our Network's parameters take as they converge towards their optimal
15:09

values the part inside of the square brackets is the error or the difference between our predicted and Target values
15:15
this is exactly the derivative of the mean squared error loss function I'm sorry if all of that was overwhelming I
The DDQN Algorithm
15:21
know it was for me the very first time I learned it however the intuition of the action value function is the hardest
15:27
part I swear it gets easier from here so how do we even begin to approximate this function that has such a vast space of
15:33
potential State action pairs well we turn to our friend the neural network remember neural networks are great at
15:40
approximating functions and extremely high Dimensions the specific architecture we'll be using today is the
15:47
convolutional neural network or a CNN CNN's excel at extracting meaningful
15:52
visual features from its inputs whether its enemies holds or the flag if you're not too familiar with CNN's I highly
15:59
recommend nvidia's learning deep learning book it'll cover literally everything you need to know about CNN
16:05
the input size will be the dimensions of our state which we'll see in just a sec the number of neurons in the output
16:12
layer will be the number of actions available to us which in this case will be five the value in each neuron
16:19
represents the predicted Q value for that Associated action paired with the
16:24
input State when running this algorithm we're actually going to have two identical copies of our neural network
16:30
the first one will be the online Network this is the one that will actually be training the second one will be the
16:37
target Network the one we'll be using for the ground Truth for our predictions to give to our loss function the target
16:43
Network won't be trained however we'll be intermittently copying the weights from the online network over to the
16:49
Target Network this is to increase stability during the training process to train the online Network we sample
16:56
the replay buffer to get a random Tuple containing a state action reward next state and done flag

we then pass in the
17:04
state to the online Network to get the predicted Q values for the five actions the different sized controllers
17:10
represent how the network values each action differently however we only care about the action that was taken and is
17:17
present in the sampled Tuple this is our predicted value for this state action
17:22
pair to see how accurate it is we calculate our Target value to do so we pass in the
17:29
next state into our Target Network to get the predicted Q values for the five actions again this time we take the
17:37
highest predicted Q value represented by the largest controller to compute our
17:42
Target value we take the reward from our sample and add it to the predicted Q value of the next state multiplied by
17:49
gamma our discount Factor we would then pass these two values into our loss
17:54
function and perform one learning step for our online Network this technique is known as bootstrapping because our poor
18:02
little network is essentially pulling itself up by its bootstraps it's using a crappy Target Network to train our
18:09
crappy online Network and then as the online Network improves just a little bit we'll copy over the weights from the
18:15
online Network to the Target Network and now we're using a slightly less crappy Target Network to continue training our
18:22
online Network we'll repeat keep this process tens of thousands of times and eventually we'll start to see our agent
18:28
actually start to learn I know this seems very Jank like we're using a moving estimate to train another
18:34
estimate however in reality we'll see how stable it actually is so let's put
DDQN Pseudocode
18:40
together all of these moving parts and finally assemble our ddq1 algorithm we start off with initializing our starting
18:46
variables and resetting our environment which gives us the starting State second given the state we'll choose an action
18:53
with our Epsilon greedy approach third we'll use this action to take one
18:59

step inside of the environment as mentioned previously the environment will return a new state and a reward
19:05
we'll then take the current state action reward the new state and the done flag
19:13
and store them all in our replay buffer then we'll do one learning step which involves sampling the replay buffer and
19:20
training the online Network then we'll take our next state and assign it to our
19:25
current state we'll repeat all of this until the episode is done and finally
19:31
we'll repeat this inside the loop for however many episodes we want to train for so let's finally get to coding this
Implementation in Code
19:39
up for the NES emulator we'll be using a library called gym Super Mario Bros it
19:45
handles all of the complexities of the emulator side of things for us it gives us neatly packaged python objects for
19:51
the States along with a simple way to input actions it does the this via a
19:56
simple API that's based on openai's gem API openai's gym library is a wildly
20:03
popular collection of different reinforcement learning environments that researchers or enthusiasts like us can
20:10
use to try out different algorithms they're incredibly easy to set up and use these days the most common approach
20:17
is to use gymnasium which is a maintained Fork of openai's gym Library by an outside team I wonder what openai
20:25
is working on these days also all of the code will be linked on GitHub right down in the description and if you ever need
20:32
to reference it throughout this video or even afterwards please feel free to do so so let's start off with a simple
20:38
example let's first import the libraries we'll be limiting our agent to only press these following button
20:44
combinations which is why we're importing write only then we create our environment object
20:50
and wrap it with the joypad space wrapper ensuring that the actions in write only are the only ones the
20:57
environment can accept then we set our done flag to false and
21:02
reset our environment this while loop will run for as long as the agent is still alive

21:08
first let's have our agent only press right on the d-pad we'll then take one step in the environment with that action
21:15
and get our done flag ignoring the other return values from the step function for now and finally because on line 6 we
21:24
have our render mode set to human calling env.render will display our
21:29
environment to our screen so we can see what our agent is doing as expected our
21:35
agent is not doing so well to add some dynamism let's have it randomly choose an action from our action space
21:42
unfortunately our agent still sucks good thing we have an entire reinforcement
21:48
learning algorithm up Our Sleeve let's implement the wrappers first rappers are essentially a way that we can modify our
21:55
environment's outputs if you want some more information about writing your own wrappers I highly recommend checking out
22:01
the gymnasium documentation which is linked below we're going to be using four different
22:06
wrappers today the wrapper we're implementing will skip frames meaning it'll just take the action inputted for
22:12
the first frame and reapply it for however many frames we want to skip we'll aggregate the rewards over those
22:19
four frames this is useful for us because consecutive frames have a lot of overlap so it's redundant to reprocess
22:26
everything the next three rappers are already implemented by gymnasium resize
22:32
observation will change the dimensions of a frame from 240 by 256 pixels to 84
22:40
by 84. this is just to reduce the computational load similarly grayscale
22:46
observation will turn the frame from having a red green and blue channel to just one reducing the amount of data our
22:53
network will have to crunch finally so our Network can see motion we're going to take four consecutive
23:00
frames and stack them on top of each other to finally create our state object that will pass into our Network each
23:07
final State object comprised of a stack of four processed frames encapsulates data from 16 original game frames due to
23:15

the combined effects of frame skipping resizing and stacking in our
23:20
pre-processing steps the code for the wrappers is pretty straightforward as we're only implementing one of them we first
23:27
override the Constructor to take the number of frames to skip then we override the step function whenever step
23:34
is called we use the for Loop to take skip number of steps aggregating the
23:39
rewards the apply wrappers function applies our wrappers one by one
23:44
encapsulating our base environment like the layers of an onion at the end it
23:50
returns our fully wrapped environment object we'll use this function in our main.pi
23:56
let's next create our neural network with pi torch as mentioned previously we'll be using a convolutional neural
24:03
network we're going to have three convolutional layers followed by two linear layers the input shape will be 4
24:10
by 84 by 84 which is the dimensions of our state the number of actions will be five the
24:16
underscore get conv out function performs a dummy forward pass through the convolutional layers to get the
24:23
number of neurons we need in our first linear layer it's 3136 with this specific architecture but
24:31
the function allows us to dynamically calculate it where we to change anything in the future
24:37
the freeze flag allows us to prevent pie Torch from calculating the gradients which we'll need for the Target Network
24:43
remember we only use the target Network to calculate the correct values we want
24:48
our online Network to predict then finally we add our forward pass and then move the network to whichever device
24:54
we're training on whether it's the CPU or GPU let's next create the agent class
25:00
these will be our Imports along with pytorch and numpy we'll also import our
25:05
neural network that we just created for the replay buffer we'll be using pi torch's built-in tensor dict replay
25:12
buffer which will hold python dictionaries with tensors as the values
25:17
the storage mechanism it'll be using is pi torch's lazy mem map storage which
25:23
will use memory mapped files for easy access to our experiences along with

25:28
alleviating RAM usage we could have just used a python list for our experiences from which we sample
25:35
but that means sampling would be really slow and we'd be using a lot of RAM
25:40
for the Constructor we'll take in the dimensions of our state and the number of actions our agent has access to we'll
25:47
also use a learn step counter which keeps track of how many times we've trained our Network
25:53
we then set up our hyper parameters first is the learning rate or Alpha this
25:59
is the size of the steps the network will take when it's updating its weights second is gamma which is our discount
26:05
Factor remember we want the agent to Discount future Rewards
26:10
third is the starting value of Epsilon which will be set to 1 initially to encourage exploration
26:16
fourth is the Epsilon Decay factor which will multiply Epsilon with after every
26:21
time step fifth is our minimum Epsilon so we always maintain some likelihood to
26:27
explore sixth is the batch size for our training and finally is how often we'll sync the
26:35
target Network weights with the online Network after the hyper parameters comes the networks again making sure the
26:42
target Network's parameters are frozen after adding our Optimizer and loss we create our replay buffer with the
26:49
capacity to hold a hundred thousand experiences then let's write our choose
26:55
action function that uses the Epsilon greedy approach as we can see when the random number is less than our Epsilon
27:02
value it'll choose a random action since Epsilon starts off at 1 the action
27:07
chosen will always be random as the value of Epsilon decays the probability
27:12
of using our online Network to choose the action with the highest Q value increases also because we want the index
27:19
itself not the estimated value of that action we're taking the ARG Max our
27:25
Decay Epsilon function multiplies the current value of Epsilon with our Decay factor and ensures it doesn't go below
27:32
the minimum Epsilon threshold the store in memory function takes in the tensors that we want to put into our

27:39
replay buffer organizes them in a dictionary and adds them to the buffer
27:44
the sync networks function checks if enough learning steps have passed and if so it'll copy over the weights of the
27:51
online Network to the Target Network now for the most important function the
27:57
learn function in the agent class we first validate that there are enough experiences in our replay buffer to
28:03
sample a batch from then we call the sync networks function we then clear our
28:08
gradients followed by sampling the replay buffer we store the results in the states actions rewards next States
28:15
and duns variables we passed the state's tensor through the online Network to get
28:22
our predicted values we index by the actions that we actually took because we only perform back propagation on those
28:29
values then we calculate our Target values we pass in the next States through our
28:36
Target Network and get the value that the best action yields we then multiply
28:41
it by gamma and added to the rewards we got in the current state the one minus
28:46
duns part sets all future rewards to zero if we're in a terminal state
28:52
we then calculate the loss using our predicted and Target Q values performing
28:57
back propagation to calculate the gradients and then perform a step of gradient descent with those gradients we
29:04
also then increment our learn step counter and Decay Epsilon finally let's create our main dot Pi in this file
29:12
we'll set up our environment as we saw previously and then create the full training loop again starting off with
29:18
our Imports this time adding our apply wrappers function and our agent class
29:24
we then create our environment applying the joypad space wrapper plus the other four that we talked about earlier we
29:31
also create our agent object and pass in the dimensions of our state and the number of actions from our environment
29:37
finally the training Loop looks very similar to our pseudo code for each episode we collect experiences and learn
29:44

from them until the episode is done first we choose an action given the starting State again this will be
29:50
completely random at first then we take a step in the environment and store the state action reward new state and done
29:58
flag in our replay buffer we then perform one learning update then we set
30:04
our current state to our new state and repeat fair warning training for 50 000
30:09
iterations can take a while I have a RTX 3080 and it took me about two days of
30:14
training non-stop however I'm excited to see what optimizations you guys can uncover by tooting the hyper parameters
The AI Beats the Level!
30:21
but now the moment you've all been waiting for after we finish training
30:27
this is the final result
30:39
thank you
Conclusion
30:56
well we did it guys we finally trained the computer to play Super Mario Bros I
31:02
hope you guys feel the magic of reinforcement learning that I've always felt in fact even bigger than that I
31:08
hope you realize that machine learning is so much bigger than just predicting labels from a data set or trying to
31:14
Cluster some points anyways if you have any questions about anything that we talked about today
31:21
please leave a comment down below and until then see you guys next time

"

Here's a concise version of your provided transcript, focusing on implementing a reinforcement learning algorithm to play Super Mario Bros using a Double Deep Q Network (DDQN):

---

**Introduction to Reinforcement Learning with Super Mario Bros:**

- **Demo & Context**: The computer plays Super Mario Bros by looking at frames and deciding button presses without knowledge of the game's mechanics.
- **Reinforcement Learning (RL)**: Used in various advanced applications, RL involves training an agent to maximize rewards within an environment through trial and error.

**Key Concepts in RL**:

1. **Agent**: The neural network controlling Mario.
2. **Environment**: The game level being played.
3. **State**: Snapshot of the game at a given time (4 consecutive frames for motion understanding).

Now tell  me the complete working from scratch how does program executes to self learn itself

Certainly! I'll break down how a reinforcement learning algorithm, specifically the Double Deep Q-Network (DDQN), can be used to train a neural network to play Super Mario Bros. I'll explain each step in detail, from setting up the environment to training the agent and observing its progress.

### Overview

Reinforcement learning involves training an agent to make decisions by rewarding it for desirable actions and penalizing it for undesirable ones. The agent interacts with an environment, learns from the outcomes of its actions, and aims to maximize its cumulative reward over time.

### Step-by-Step Explanation

#### 1. Set Up the Environment

First, we need to set up the environment in which our agent will operate. We'll use the `gymnasium` library, a fork of OpenAI's Gym, along with `gym-super-mario-bros` to simulate the Super Mario Bros game.

```python
import gymnasium as gym
import gym_super_mario_bros
from gymnasium.wrappers import FrameStack
from nes_py.wrappers import JoypadSpace
import numpy as np
import random
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque

env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, [['right'], ['right', 'A'], ['A']])
```

#### 2. Define Wrappers

To preprocess the frames, reduce the computational load, and handle the state representation, we use wrappers. Wrappers will skip frames, resize the frames, convert them to grayscale, and stack multiple frames together.

python

```
import gymnasium as gym
from gymnasium.wrappers import GrayScaleObservation, ResizeObservation

class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        for _ in range(self._skip):
            state, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return state, total_reward, done, info

def apply_wrappers(env):
    env = SkipFrame(env, skip=4)
    env = GrayScaleObservation(env)
    env = ResizeObservation(env, shape=84)
    env = FrameStack(env, num_stack=4)
    return env

env = apply_wrappers(env)
```

#### 3. Define the Neural Network

The neural network will take the state (stacked frames) as input and output the Q-values for each possible action. This network will be used to approximate the action-value function.

python

```
class MarioNet(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(MarioNet, self).__init__()
        self.conv1 = nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc1 = nn.Linear(7*7*64, 512)
        self.fc2 = nn.Linear(512, n_actions)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

input_shape = (4, 84, 84)
n_actions = env.action_space.n
net = MarioNet(input_shape, n_actions)
```

#### 4. Define Replay Buffer

The replay buffer stores the agent's experiences, which will be used to train the neural network.

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def add(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.buffer,
batch_size))
        return np.stack(state), action, reward, np.stack(next_state), done

    def size(self):
        return len(self.buffer)
```

### 5. Define the Training Loop

The training loop involves the following steps:

- Initialize variables and the environment.
- Choose an action using the epsilon-greedy policy.
- Take a step in the environment.
- Store the experience in the replay buffer.
- Sample a batch of experiences from the replay buffer.
- Compute the target Q-values using the Bellman equation.
- Update the network's weights using backpropagation.

```python
def compute_td_loss(batch_size, gamma=0.99):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state = torch.FloatTensor(state).to(device)
    next_state = torch.FloatTensor(next_state).to(device)
    action = torch.LongTensor(action).to(device)
    reward = torch.FloatTensor(reward).to(device)
    done = torch.FloatTensor(done).to(device)

    q_values = net(state)
    next_q_values = target_net(next_state)

    q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = next_q_values.max(1)[0]
    expected_q_value = reward + gamma * next_q_value * (1 - done)

    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    return loss

replay_buffer = ReplayBuffer(10000)
optimizer = optim.Adam(net.parameters())
target_net = MarioNet(input_shape, n_actions).to(device)
target_net.load_state_dict(net.state_dict())
target_net.eval()

epsilon = 1.0
epsilon_decay = 0.99
```

```
min_epsilon = 0.1
batch_size = 32
gamma = 0.99

num_episodes = 500

for episode in range(num_episodes):
    state = env.reset()
    total_reward = 0

    while True:
        if random.random() < epsilon:
            action = env.action_space.sample()
        else:
            with torch.no_grad():
                action = net(torch.FloatTensor(state).unsqueeze(0).to(device)).max(1)
[1].item()

        next_state, reward, done, info = env.step(action)
        replay_buffer.add(state, action, reward, next_state, done)

        state = next_state
        total_reward += reward

        if replay_buffer.size() > batch_size:
            loss = compute_td_loss(batch_size, gamma)

        if done:
            break

    if epsilon > min_epsilon:
        epsilon *= epsilon_decay

    if episode % 10 == 0:
        target_net.load_state_dict(net.state_dict())
        print(f'Episode {episode}, Total Reward: {total_reward}')

env.close()
```

### Summary

1. **Setup**: Import necessary libraries and initialize the environment.
2. **Preprocess Data**: Use wrappers to preprocess the frames.
3. **Neural Network**: Define a CNN to approximate the Q-value function.
4. **Replay Buffer**: Implement a replay buffer to store experiences.
5. **Training Loop**: Implement the training loop that:
    - Chooses actions using the epsilon-greedy policy.
    - Steps through the environment.
    - Stores experiences in the replay buffer.
    - Samples from the replay buffer to train the network using the Bellman equation.
    - Updates the target network periodically.

By following these steps, the agent will learn to play Super Mario Bros through reinforcement learning, improving its performance over time by interacting with the environment and learning from the rewards and penalties it receives.

i need each step of working not code

Sure, let's walk through each step of how a Double Deep Q-Network (DDQN) reinforcement learning algorithm works to train an agent to play a game like Super Mario Bros, without diving into the code.

### Step 1: Set Up the Environment

**Description**: The environment is a simulated version of the game that the agent interacts with. It provides observations (states), responds to actions taken by the agent, and provides rewards based on the agent's performance.

**Process**:

- Load the game using a simulation library.
- Define the action space (possible moves the agent can make).
- Set up the initial state of the game.

### Step 2: Preprocess Data

**Description**: To make the data manageable and useful for the neural network, we preprocess the game frames.

**Process**:

- **Frame Skipping**: Skip frames to reduce the frequency of decisions, which simplifies the problem and speeds up learning.
- **Grayscale Conversion**: Convert frames to grayscale to reduce complexity (from 3 color channels to 1).
- **Resizing**: Resize the frames to a smaller resolution to reduce computational load.
- **Frame Stacking**: Stack multiple consecutive frames together to capture motion information.

### Step 3: Define the Neural Network

**Description**: A neural network approximates the Q-value function, which estimates the expected reward of taking an action in a given state.

**Process**:

- **Input Layer**: Takes the processed game frames as input.
- **Convolutional Layers**: Extract features from the input frames.
- **Fully Connected Layers**: Process these features to produce Q-values for each action.
- **Output Layer**: Outputs Q-values for all possible actions, indicating their expected future rewards.

### Step 4: Initialize the Replay Buffer

**Description**: The replay buffer stores the agent's experiences, allowing the network to learn from a diverse set of past interactions.

**Process**:

- **Experience Storage**: Each experience consists of the current state, action taken, reward received, next state, and a done flag indicating if the episode ended.
- **Sampling**: Randomly sample experiences from the buffer to break correlations and stabilize learning.

### Step 5: Implement the Training Loop

**Description**: The training loop is where the agent interacts with the environment, collects experiences, and updates the neural network.

**Process**:

**a. Initialize Variables**

- Set initial values for parameters like the exploration rate (epsilon), discount factor (gamma), batch size, and learning rate.

**b. Interaction with the Environment**

- **Action Selection**: Use an epsilon-greedy policy to balance exploration (random actions) and exploitation (choosing the best-known action).
  - With probability epsilon, select a random action.
  - With probability (1 - epsilon), select the action with the highest Q-value predicted by the neural network.
- **Take Action**: Perform the selected action in the environment.
- **Observe Outcome**: Record the next state, reward, and whether the episode ended.

**c. Store Experience**

- Add the experience (current state, action, reward, next state, done) to the replay buffer.

**d. Sample from Replay Buffer**

- Once the buffer has enough experiences, sample a batch of experiences for training.

**e. Compute Target Q-Values**

- For each experience in the batch, compute the target Q-value using the Bellman equation:
  - If the episode ended (done flag is true), the target Q-value is the reward.
  - If the episode did not end, the target Q-value is the reward plus the discounted maximum future Q-value of the next state.

**f. Update the Network**

- Calculate the loss between the predicted Q-values and the target Q-values.
- Perform a gradient descent step to minimize this loss and update the network's weights.

**g. Update Target Network**

- Periodically copy the weights from the main network to the target network to stabilize training.

### Step 6: Adjust Exploration Rate

**Description**: Gradually reduce the exploration rate (epsilon) to shift from exploration to exploitation as the agent learns.

**Process**:

- Decrease epsilon by a small factor after each episode until it reaches a minimum threshold.

### Step 7: Monitor and Evaluate

**Description**: Track the agent's performance over time to ensure it is learning.

**Process**:

- Print or log metrics such as total reward per episode.
- Optionally, visualize the agent's progress by rendering the game or plotting performance graphs.

### Summary

1. **Set Up the Environment**: Initialize the game simulation and define the action space.
2. **Preprocess Data**: Simplify and format game frames for the neural network.
3. **Define Neural Network**: Create a network to estimate Q-values for actions.
4. **Initialize Replay Buffer**: Store and sample past experiences for training.
5. **Implement Training Loop**: Interact with the environment, collect experiences, and update the network.
6. **Adjust Exploration Rate**: Gradually shift from exploration to exploitation.
7. **Monitor and Evaluate**: Track the agent's performance and ensure learning is effective.

By iteratively interacting with the environment and learning from the collected experiences, the agent improves its ability to play the game, aiming to maximize its cumulative reward.

How does agent know which is the obstacle does it learn itself by multiple trying?

Yes, the agent learns to recognize obstacles and other important elements in the game environment through its interactions and experiences. Here's how this process works in detail:

### Learning Through Interaction

1. **Exploration and Exploitation**:
    - Initially, the agent explores the environment by taking random actions (exploration). This allows it to encounter various situations, including encountering obstacles.
    - As the training progresses, the agent increasingly exploits its knowledge by choosing actions that it has learned yield higher rewards (exploitation).

2. **Collecting Experiences**:

   - Each time the agent takes an action, it observes the outcome in terms of the new state of the environment, the reward received, and whether the game is over.
   - These interactions are stored as experiences in the replay buffer. Each experience consists of:
     - The current state (a representation of the game screen, which includes obstacles, characters, etc.).
     - The action taken.
     - The reward received.
     - The next state (the new game screen after the action).
     - A flag indicating whether the game ended.

3. **Learning from Rewards**:

   - The agent receives positive rewards for actions that bring it closer to the goal (e.g., moving forward, collecting items) and negative rewards for actions that lead to undesirable outcomes (e.g., hitting an obstacle).
   - By learning which actions lead to higher cumulative rewards, the agent begins to understand the importance of avoiding obstacles.

## Training the Neural Network

4. **Processing Visual Input**:

   - The agent processes game frames through a neural network, which is designed to extract relevant features from the visual input.
   - Convolutional layers in the network help identify patterns in the pixels, such as shapes and edges that correspond to obstacles and paths.

5. **Predicting Q-Values**:

   - The neural network predicts Q-values for each possible action, representing the expected future reward of taking that action in the current state.
   - Over time, the network learns to associate specific visual patterns (like the appearance of an obstacle) with lower Q-values for actions that would lead to collisions.

6. **Updating the Network**:

   - Using the experiences sampled from the replay buffer, the network is trained to minimize the difference between predicted Q-values and target Q-values.
   - Target Q-values are calculated based on actual rewards received and the maximum Q-value of the next state, incorporating the presence of obstacles in these calculations.

## Recognizing Obstacles

7. **Pattern Recognition**:

   - Through repeated exposure and training, the neural network improves its ability to recognize visual patterns that represent obstacles.
   - When the agent encounters a similar state in the future, the network's learned weights help it predict the potential negative outcomes of colliding with an obstacle, thereby guiding it to

take safer actions.

8. **Behavior Adaptation**:
   - The agent's policy (strategy for selecting actions) evolves as it learns from the accumulated experiences.
   - The agent becomes adept at navigating the environment, avoiding obstacles, and making decisions that maximize long-term rewards.

### Continuous Learning and Refinement

9. **Adjusting Strategies**:
   - The agent continuously refines its strategies as it encounters new situations and obstacles.
   - It balances the need to explore new possibilities with exploiting known strategies that work well, gradually improving its performance.

10. **Feedback Loop**:
    - The feedback loop of taking actions, receiving rewards, and updating the network allows the agent to progressively learn more sophisticated behaviors, such as anticipating obstacles and planning moves in advance.

### Summary

- **Initial Exploration**: The agent starts by exploring the environment and taking random actions.
- **Experience Collection**: It collects experiences that include information about obstacles.
- **Reward-Based Learning**: It learns from rewards, understanding that avoiding obstacles often leads to higher rewards.
- **Neural Network Training**: The neural network is trained to recognize visual patterns corresponding to obstacles.
- **Policy Improvement**: The agent's policy improves as it learns to navigate the environment more effectively, avoiding obstacles and optimizing for rewards.

Through this iterative process, the agent learns to recognize obstacles and other critical elements in the game environment, improving its performance over time.

Now summarize the working in a cyclic flow

### Cyclic Flow of Agent Self-Learning in Reinforcement Learning

1. **Initialization**:
   - The agent starts with random actions and an untrained neural network.

2. **Exploration**:
   - The agent explores the environment by taking random actions to gather diverse experiences.
3. **Experience Collection**:
   - The agent collects experiences in the format (state, action, reward, next state, done) and stores them in a replay buffer.
4. **Action Execution**:
   - The agent takes an action based on either exploration or exploitation of its current knowledge (predicted Q-values).
5. **State Transition and Reward**:
   - The environment responds with a new state, a reward, and a flag indicating if the episode is done.
6. **Experience Storage**:
   - The current experience (state, action, reward, next state, done) is stored in the replay buffer.
7. **Experience Sampling**:
   - A batch of experiences is sampled from the replay buffer for training the neural network.
8. **Q-Value Prediction**:
   - The neural network predicts Q-values for the current state-action pairs.
9. **Target Calculation**:
   - Target Q-values are calculated using the reward and the maximum Q-value of the next state.
10. **Network Update**:
    - The neural network is trained by minimizing the difference between predicted and target Q-values using backpropagation.
11. **Policy Improvement**:
    - The agent updates its policy based on the improved Q-values, favoring actions that lead to higher expected rewards.
12. **Loop Back**:
    - The process repeats from step 3, with the agent continually improving its understanding and policy through interaction with the environment.

### Cyclic Flow Diagram

1. **Initialization →**
2. **Exploration →**
3. **Experience Collection →**
4. **Action Execution →**
5. **State Transition and Reward →**
6. **Experience Storage →**
7. **Experience Sampling →**
8. **Q-Value Prediction →**
9. **Target Calculation →**
10. **Network Update →**
11. **Policy Improvement →**
12. **Loop Back to Step 3**

This cyclic process continues until the agent has sufficiently learned to navigate the environment effectively, recognizing and avoiding obstacles to maximize rewards.

define all the key terms for reinforcement learning from the give video tanscript

Certainly! Here's a list of key terms for reinforcement learning derived from the video transcript:

1. **Agent**:
   - The entity that makes decisions and interacts with the environment to achieve a goal.
2. **Environment**:
   - The external system with which the agent interacts. It provides the state and reward in response to the agent's actions.
3. **State**:
   - A representation of the current situation or configuration of the environment as perceived by the agent.
4. **Action**:
   - A decision made by the agent that affects the state of the environment.
5. **Reward**:
   - A scalar value received by the agent from the environment as feedback for the action taken. It indicates the immediate benefit of an action.
6. **Policy**:

- A strategy or mapping from states to actions that defines the agent's behavior. It can be deterministic or stochastic.

7. **Q-Value (Quality Value)**:
   - A measure of the expected future rewards for taking a particular action in a given state and following the current policy thereafter.
8. **Q-Learning**:
   - An off-policy reinforcement learning algorithm where the agent learns the value of the optimal policy independently of the agent's actions.
9. **Neural Network**:
   - A computational model used to approximate complex functions, such as Q-values in reinforcement learning.
10. **Experience Replay Buffer**:
    - A memory that stores past experiences (state, action, reward, next state, done) for training the neural network. It helps in breaking the correlation between consecutive experiences.
11. **Exploration**:
    - The process of taking random actions to discover new states and rewards, crucial for learning in unknown environments.
12. **Exploitation**:
    - The process of taking actions based on the current policy to maximize rewards, utilizing the knowledge already gained.
13. **Epsilon-Greedy Strategy**:
    - A method of balancing exploration and exploitation by choosing a random action with probability epsilon and the best-known action with probability 1-epsilon.
14. **Target Q-Value**:
    - The updated estimate of the Q-value, calculated using the reward and the maximum Q-value of the next state.
15. **Backpropagation**:
    - A method used to train neural networks by adjusting weights to minimize the difference between predicted and actual Q-values.
16. **Learning Rate**:
    - A hyperparameter that controls the size of weight updates during training.

17. **Discount Factor (Gamma)**:
    - A parameter that determines the importance of future rewards. A value close to 1 emphasizes long-term rewards, while a value close to 0 emphasizes immediate rewards.
18. **Done**:
    - A flag indicating whether the current episode has ended, typically when the agent reaches a terminal state or a maximum number of steps is reached.

These key terms provide a comprehensive understanding of the components and concepts involved in reinforcement learning, as described in the video transcript.

why didn't we directly train the neural network for current  datarather we used replay-buffer to train it

Using a replay buffer to train a neural network in reinforcement learning offers several key advantages over training the network on the current data alone. Here's a detailed explanation of why this approach is preferred:

1. **Breaking Temporal Correlation**:
    - In reinforcement learning, consecutive states and actions are highly correlated. Training a neural network on these consecutive experiences can lead to poor learning and convergence because the updates to the network weights will be highly correlated.
    - A replay buffer stores experiences (state, action, reward, next state, done) and samples them randomly during training. This random sampling breaks the temporal correlations and makes the learning process more stable and efficient.
2. **Efficient Use of Data**:
    - In many reinforcement learning environments, especially in simulation or real-world scenarios, gathering data can be costly and time-consuming.
    - A replay buffer allows the agent to reuse past experiences multiple times, making the learning process more data-efficient. This leads to better learning from fewer interactions with the environment.
3. **Reduced Variance in Updates**:
    - When training directly on the latest data, the updates to the neural network can be noisy because the agent might not have seen enough examples to generalize well.
    - By sampling a diverse set of experiences from the replay buffer, the updates to the neural network become smoother and less variable, leading to more reliable learning.
4. **Stability in Training**:

- Reinforcement learning algorithms can be unstable, especially when combined with function approximation like neural networks. The replay buffer helps in stabilizing training by providing a more consistent gradient estimation for the updates.

5. **Avoiding Catastrophic Forgetting**:

    - If the agent were to train only on the most recent experiences, it might forget important information from past experiences, especially if the environment changes or if the agent's policy changes significantly.
    - A replay buffer helps mitigate this by ensuring that experiences from different parts of the agent's exploration history are retained and used for training, preventing the network from forgetting valuable past information.

6. **Improved Convergence**:

    - Due to the reasons mentioned above (breaking temporal correlation, efficient use of data, reduced variance, stability, and avoiding catastrophic forgetting), using a replay buffer often leads to improved convergence of the neural network. The agent is more likely to learn a robust policy that performs well across a variety of states and actions.

### How Replay Buffer Works in Practice

1. **Experience Collection**:

    - The agent interacts with the environment and collects experiences, which are tuples of (state, action, reward, next state, done).

2. **Storing in Replay Buffer**:

    - Each experience tuple is stored in the replay buffer. The buffer typically has a fixed size, and when it is full, the oldest experiences are discarded to make room for new ones.

3. **Sampling for Training**:

    - During the training phase, a batch of experiences is randomly sampled from the replay buffer. This batch is used to update the neural network.

4. **Updating the Neural Network**:

    - The network is trained using the sampled batch, where the loss is calculated (e.g., using the difference between predicted Q-values and target Q-values) and backpropagation is performed to update the network weights.

By incorporating a replay buffer, reinforcement learning algorithms can achieve more stable and efficient learning, ultimately leading to better performance and more robust policies.

why we need two networks online and target network

Using two networks—an online network and a target network—is a technique commonly employed in deep reinforcement learning, particularly in algorithms like Deep Q-Networks (DQN). This method addresses several issues that arise when training neural networks to approximate value functions. Here's why having both an online network and a target network is beneficial:

### 1. Stabilizing Training

- **Non-stationary Targets**: In reinforcement learning, the target values for the Q-function are not stationary because they depend on the current policy, which is constantly being updated. This can lead to instability as the network is trying to chase a moving target.
- **Solution**: By using a target network, which is a delayed copy of the online network, the targets for the Q-values are updated less frequently. This creates more stable targets, which helps stabilize the training process.

### 2. Reducing Oscillations

- **Feedback Loop**: Without a target network, the Q-values used for training the online network are directly influenced by the same network's weights. This feedback loop can cause large oscillations in Q-value estimates, leading to poor performance and divergence.
- **Solution**: The target network mitigates this issue by providing a consistent target over several updates. This helps smooth out the learning process, reducing oscillations in the Q-value estimates.

### 3. Improving Convergence

- **Consistent Learning Signal**: When the target network is updated at a slower rate compared to the online network, the learning signal becomes more consistent. This allows the online network to converge more reliably to an optimal Q-function.
- **Solution**: The target network provides a relatively stable baseline for the updates, which improves the convergence properties of the algorithm.

### How It Works in Practice

1. **Online Network**:

   - The online network (also called the policy network or the Q-network) is the one that is actively learning. It is updated every time a new batch of experiences is sampled from the replay buffer.
   - It generates the Q-values for the current state-action pairs, which are used to select actions and compute the loss during training.

2. **Target Network**:

   - The target network is a copy of the online network but with its weights updated less frequently. Typically, the target network is updated to match the online network's weights either periodically (every fixed number of steps) or softly (by slowly mixing the weights of the online network into the target network using a parameter $\tau$).

- It generates the Q-values for the next state, which are used to calculate the target Q-values for the Bellman update.

### Example of Update Mechanism

1. **Experience Collection**:
    - The agent interacts with the environment and stores experiences (state, action, reward, next state, done) in the replay buffer.
2. **Sampling from Replay Buffer**:
    - A batch of experiences is randomly sampled from the replay buffer.
3. **Q-Value Computation**:
    - **Online Network**: Computes the current Q-values for the state-action pairs from the sampled batch.
    - **Target Network**: Computes the target Q-values for the next state. This is typically done by taking the maximum Q-value across all actions for the next state.
4. **Loss Calculation and Backpropagation**:
    - The loss is computed as the difference between the predicted Q-values from the online network and the target Q-values from the target network.
    - The online network is updated by backpropagating this loss.
5. **Target Network Update**:
    - Periodically, the weights of the target network are updated to match the weights of the online network. This can be done either by hard updates (copying the weights directly) or by soft updates (blending the weights using a small parameter τ).

### Benefits Summarized

- **Stability**: The target network helps stabilize training by providing a consistent target, reducing the risk of oscillations and divergence.
- **Convergence**: By providing a more stable learning signal, the use of a target network improves the convergence properties of the algorithm, leading to better performance.
- **Efficiency**: This approach makes the learning process more efficient by reducing the variance in updates, allowing the network to learn from past experiences more effectively.

By incorporating both an online network and a target network, deep reinforcement learning algorithms can achieve more stable and reliable learning, ultimately leading to better policies and performance in various tasks.

what are wrappers

In the context of reinforcement learning (RL), wrappers are a useful tool provided by many RL frameworks (such as OpenAI's Gym) to modify environments in a flexible and reusable way. Wrappers allow you to add additional functionality to environments without altering their underlying code. They are commonly used for preprocessing observations, modifying rewards, or logging additional information during training.

### Key Types of Wrappers

1. **Observation Wrappers**:
    - Modify the observations returned by the environment.
    - Example: Normalizing pixel values in an image-based observation.
2. **Action Wrappers**:
    - Modify the actions taken by the agent before they are passed to the environment.
    - Example: Discretizing a continuous action space into a set of discrete actions.
3. **Reward Wrappers**:
    - Modify the rewards returned by the environment.
    - Example: Scaling rewards to a specific range or clipping rewards to prevent extreme values.
4. **Environment Wrappers**:
    - Modify the environment's behavior or add new functionalities that do not fit into the above categories.
    - Example: Adding a time limit to episodes or combining multiple environments for multi-task learning.

### How Wrappers Work

Wrappers typically inherit from the base environment class and override specific methods to add their functionality. The most commonly overridden methods include `reset()`, `step(action)`, and `render()`.

### Example Workflow Using Wrappers

1. **Initialize Base Environment**:
    - Create the base environment instance that you want to wrap.

```python
import gym
env = gym.make('CartPole-v1')
```

2. **Apply Observation Wrapper**:

   - Wrap the environment to modify the observations.

```python
from gym.wrappers import TransformObservation

def normalize_observation(obs):
    return obs / 255.0

env = TransformObservation(env, normalize_observation)
```

3. **Apply Action Wrapper**:

   - Wrap the environment to modify the actions.

```python
from gym.wrappers import TransformReward

def scale_reward(reward):
    return reward * 0.01

env = TransformReward(env, scale_reward)
```

4. **Apply Reward Wrapper**:

   - Wrap the environment to modify the rewards.

```python
from gym.wrappers import TimeLimit

env = TimeLimit(env, max_episode_steps=200)
```

5. **Use the Wrapped Environment**:

   - Use the wrapped environment in the RL algorithm as you normally would.

```python
observation = env.reset()
done = False
while not done:
    action = env.action_space.sample()  # Sample random action
    observation, reward, done, info = env.step(action)
    env.render()
env.close()
```

## Benefits of Using Wrappers

- **Modularity**: Wrappers allow you to encapsulate changes and add new functionalities in a modular fashion. You can stack multiple wrappers together, each handling a specific aspect of the environment.
- **Reusability**: Once defined, wrappers can be reused across different environments and projects, saving time and effort.
- **Maintainability**: Wrappers help keep the environment code clean and maintainable by separating preprocessing and modification logic from the core environment.

### Common Use Cases

- **Preprocessing Observations**: Normalizing pixel values, resizing images, converting color spaces, etc.
- **Action Modification**: Converting discrete actions to continuous ones, action clipping, etc.
- **Reward Shaping**: Scaling rewards, reward clipping, adding custom reward functions, etc.
- **Episode Management**: Adding time limits, early stopping conditions, episodic logging, etc.

### Conclusion

Wrappers are a powerful and flexible tool in reinforcement learning, enabling the modification and extension of environments in a clean, modular, and reusable manner. By leveraging wrappers, you can efficiently preprocess data, modify environment interactions, and add custom functionalities to better suit your RL tasks.

why you think ddqn network is a good choice for this project

Double Deep Q-Network (DDQN) is a good choice for the project due to several key reasons that address some of the common challenges in reinforcement learning, particularly when dealing with Q-learning algorithms. Here's why DDQN is advantageous:

### 1. Addressing Overestimation Bias

Traditional Q-learning and Deep Q-Networks (DQNs) suffer from overestimation bias, where the algorithm tends to overestimate the value of certain actions due to the maximization step. This bias can lead to suboptimal policies and unstable learning.

- **DDQN Solution**: DDQN reduces this bias by decoupling the action selection from the action evaluation. Instead of using the same Q-network for both selecting and evaluating an action, DDQN uses the online network to select the action and the target network to evaluate it. This results in more accurate Q-value estimates and improves learning stability.

### 2. Improved Stability and Convergence

Reinforcement learning algorithms, especially those involving deep neural networks, can be unstable and slow to converge. This instability often stems from the correlation between consecutive updates and large updates that shift the policy abruptly.

- **DDQN Solution**: By having two networks (the online network and the target network) and periodically updating the target network with the weights of the online network, DDQN smoothens the updates, leading to more stable and reliable learning. This periodic update helps in reducing the variance in updates and contributes to better convergence properties.

### 3. Sample Efficiency

Efficient use of experience data is crucial for training deep reinforcement learning agents, especially in environments where collecting new data is costly or time-consuming.

- **Replay Buffer**: DDQN uses a replay buffer to store experiences (state, action, reward, next state) and samples mini-batches from this buffer to update the network. This mechanism ensures that experiences are reused multiple times, improving sample efficiency and helping the agent learn from past experiences more effectively.

### 4. Handling Complex Environments

DDQN, with its improved stability and accuracy, is well-suited for environments with high-dimensional state and action spaces or environments where precise action selection is critical.

- **Complex Decision-Making**: In environments like robotic control, autonomous driving, or complex games, the ability to accurately evaluate action outcomes is vital. DDQN's reduced bias and improved stability make it a good choice for such complex decision-making tasks.

### 5. Practical Implementation

DDQN has been extensively tested and validated across various benchmarks, showing its robustness and effectiveness in different scenarios.

- **Community and Support**: As a well-established method, there is ample community support, documentation, and existing implementations for DDQN, which can significantly aid in troubleshooting and enhancing the project.

### 6. Flexibility and Extendibility

DDQN can serve as a strong foundation for further enhancements and experimentation.

- **Extensions**: Once a DDQN-based agent is working well, it can be extended with other techniques such as Prioritized Experience Replay, Dueling Networks, or multi-step learning, further enhancing its performance.

### Summary

DDQN is a good choice for this project because it effectively addresses the overestimation bias inherent in traditional Q-learning, leading to more accurate Q-value estimates and improved stability. Its sample efficiency and robust performance in complex environments make it suitable for a wide range of applications. Additionally, the practical benefits of community support and the potential for further enhancements make DDQN a solid foundation for reinforcement learning projects.

where did you got the idea for the  Super Mario Bros to implement it with ddqn

The idea to implement a Double Deep Q-Network (DDQN) for Super Mario Bros. comes from the broader field of reinforcement learning (RL) and its applications to video games. Here's a breakdown of how this idea likely evolved and the reasons behind it:

### Background in Reinforcement Learning and Video Games

1. **Reinforcement Learning (RL)**:
    - RL is a type of machine learning where an agent learns to make decisions by performing actions in an environment to maximize cumulative reward.
    - The agent learns through trial and error, using feedback from its actions and the resulting states of the environment.
2. **Video Games as RL Benchmarks**:
    - Video games have long been used as benchmarks for testing and developing RL algorithms because they offer complex, dynamic, and diverse environments.
    - Games like Atari 2600 (via the Arcade Learning Environment) and others have been popular choices for testing due to their structured yet challenging scenarios.

### Deep Q-Network (DQN)

- **DQN**: Developed by DeepMind, DQN combines Q-learning with deep neural networks, allowing RL to handle high-dimensional sensory inputs like raw pixels from video games.
- **Successes**: DQN demonstrated impressive performance on Atari games, where the agent learned to play directly from pixel inputs, achieving superhuman performance on many games.

### Double DQN (DDQN)

- **Overestimation Bias**: Traditional DQNs suffer from overestimation bias, where they tend to overestimate action values, leading to unstable learning.
- **DDQN**: Proposed to address this issue by using two networks: the online network for action selection and the target network for action evaluation, leading to more stable and accurate learning.

### Super Mario Bros. and RL

- **Suitability**: Super Mario Bros. is a well-known, classic platformer game that presents a challenging environment for RL agents. It includes diverse levels, varying difficulties, and the need for precise control and timing.
- **Research and Community Interest**: Given its popularity and the complexity of its gameplay, Super Mario Bros. has attracted interest from researchers and hobbyists in the RL community.

### Implementing DDQN for Super Mario Bros.

1. **Challenges of Super Mario Bros.**:

   - The game involves high-dimensional input (pixel data), complex physics, and a variety of enemies and obstacles.
   - It requires both short-term precision (avoiding obstacles, timing jumps) and long-term strategy (completing levels, maximizing score).

2. **Benefits of DDQN**:

   - **Stability**: DDQN's reduced overestimation bias leads to more stable learning, which is crucial for mastering complex levels in Super Mario Bros.
   - **Sample Efficiency**: Using replay buffers helps the agent learn more effectively from limited data, which is beneficial given the complexity of the game environment.

### Influence from Existing Work

- **Previous Implementations**: The idea of applying RL to Super Mario Bros. isn't new. Previous works have implemented basic RL algorithms like Q-learning and DQN on the game.
- **Improvement through DDQN**: Building on these earlier works, implementing DDQN is a natural progression to improve performance and stability.

### Summary

The idea to use DDQN for Super Mario Bros. comes from a combination of the general success of RL in video games, the specific improvements that DDQN offers over traditional DQN, and the suitability of Super Mario Bros. as a challenging benchmark environment. The approach leverages the strengths of DDQN to handle the complexities of the game, building on prior research and the known benefits of using RL in video game settings.

agent.py

agent_nn.py

generate_clips.py

main.py

requirements.txt

simple_example.py

simplified_main.py

utils.py

wrappers.py tell me what each file does

Sure, here's a brief description of the purpose of each file in the context of a project implementing a Double Deep Q-Network (DDQN) for Super Mario Bros.:

1. **agent.py**:
   - This file typically defines the main agent class responsible for interacting with the environment. It includes methods for choosing actions based on the policy (using the neural network), updating the policy based on experience, and managing the agent's state and transitions.
2. **agent_nn.py**:
   - This file usually contains the definition of the neural network architecture used by the agent. It includes the implementation of the online and target networks, and the forward pass method that predicts Q-values for given states.
3. **generate_clips.py**:
   - This script is likely used to generate gameplay clips, possibly for visualization or analysis. It may record the agent's performance during training or evaluation to create video clips showing how the agent interacts with the environment.
4. **main.py**:
   - The main entry point for training the DDQN agent. This script sets up the environment, initializes the agent, and runs the training loop. It handles the high-level flow of the training process, including episodes, steps within episodes, and logging progress.
5. **requirements.txt**:
   - A list of Python packages required to run the project. This file ensures that all necessary dependencies are installed, typically used with package managers like pip.
6. **simple_example.py**:
   - A simplified example script to demonstrate basic functionality. This file likely contains a minimal setup to show how the agent interacts with the environment, useful for understanding the core concepts without the complexity of the full training loop.
7. **simplified_main.py**:
   - A streamlined version of `` `main.py` ``, possibly for testing or debugging. This script may contain a more straightforward implementation of the training loop or focus on a subset of the full functionality for quicker iterations.

8. **utils.py**:

    - A utility module that provides helper functions used across the project. It may include functions for preprocessing data, managing replay buffers, saving/loading models, logging, and other common tasks that support the main functionality.

9. **wrappers.py**:

    - This file likely defines environment wrappers. In reinforcement learning, wrappers are used to modify the environment to make it more suitable for training. This could involve preprocessing observations, modifying reward structures, or adding additional logging and monitoring capabilities.

### Detailed Flow and Function of Each File:

#### 1. agent.py:

- **Functions**:
    - `` `select_action(state)` ``: Uses the policy to select an action based on the current state.
    - `` `optimize_model()` ``: Updates the online network using experiences sampled from the replay buffer.
    - `` `store_transition(state, action, reward, next_state, done)` ``: Stores the experience tuple in the replay buffer.

#### 2. agent_nn.py:

- **Functions**:
    - `` `DQN` ``: Defines the architecture of the neural network (e.g., convolutional layers for processing pixel data).
    - `` `forward(state)` ``: Passes the input state through the network to predict Q-values.

#### 3. generate_clips.py:

- **Functions**:
    - `` `record_episode(agent, environment, file_name)` ``: Runs an episode with the trained agent and records the gameplay to a video file.
    - `` `save_clip(video, path)` ``: Saves the recorded video clip to the specified path.

#### 4. main.py:

- **Functions**:
    - `` `train()` ``: Orchestrates the training process, including initializing the environment and agent, running episodes, and periodically updating the target network.
    - `` `evaluate(agent)` ``: Evaluates the performance of the agent after training or during intermediate checkpoints.

#### 5. requirements.txt:

- **Content**:
    - Lists packages like `` `gym` ``, `` `numpy` ``, `` `torch` ``, `` `opencv-python` ``, and others required for the project.

**6. simple_example.py:**

- **Functions**:
  - `run_example()`: Sets up a simple environment and agent, runs a few steps to demonstrate interaction, and prints/logs basic results.

**7. simplified_main.py:**

- **Functions**:
  - `train_simple()`: A more straightforward version of the training loop with fewer features and potentially less logging for quick testing.

**8. utils.py:**

- **Functions**:
  - `preprocess_state(state)`: Preprocesses the raw state from the environment (e.g., resizing and normalizing pixel data).
  - `ReplayBuffer`: A class implementing the replay buffer to store and sample experiences.
  - `save_model(agent, path)`: Saves the trained model to disk.
  - `load_model(path)`: Loads a saved model from disk.

**9. wrappers.py:**

- **Functions**:
  - `ObservationWrapper`: Modifies the observations returned by the environment (e.g., converting to grayscale, resizing).
  - `RewardWrapper`: Modifies the rewards returned by the environment to potentially shape the learning process.
  - `Monitor`: Records episodes for later playback and analysis.

These descriptions outline the general structure and purpose of each file, providing a cohesive overview of how they fit together to implement a DDQN agent for Super Mario Bros.

explain main.py code import torch

import gym_super_mario_bros
from gym_super_mario_bros.actions import RIGHT_ONLY

from agent import Agent

from nes_py.wrappers import JoypadSpace
from wrappers import apply_wrappers

import os

from utils import *

model_path = os.path.join("models", get_current_date_time_string())

```
os.makedirs(model_path, exist_ok=True)

if torch.cuda.is_available():
    print("Using CUDA device:", torch.cuda.get_device_name(0))
else:
    print("CUDA is not available")

ENV_NAME = 'SuperMarioBros-1-1-v0'
SHOULD_TRAIN = True
DISPLAY = True
CKPT_SAVE_INTERVAL = 5000
NUM_OF_EPISODES = 50_000

env = gym_super_mario_bros.make(ENV_NAME, render_mode='human' if DISPLAY else 'rgb',
apply_api_compatibility=True)
env = JoypadSpace(env, RIGHT_ONLY)

env = apply_wrappers(env)

agent = Agent(input_dims=env.observation_space.shape, num_actions=env.action_space.n)

if not SHOULD_TRAIN:
    folder_name = ""
    ckpt_name = ""
    agent.load_model(os.path.join("models", folder_name, ckpt_name))
    agent.epsilon = 0.2
    agent.eps_min = 0.0
    agent.eps_decay = 0.0

env.reset()
next_state, reward, done, trunc, info = env.step(action=0)

for i in range(NUM_OF_EPISODES):
    print("Episode:", i)
    done = False
    state, _ = env.reset()
    total_reward = 0
    while not done:
        a = agent.choose_action(state)
        new_state, reward, done, truncated, info  = env.step(a)
        total_reward += reward

        if SHOULD_TRAIN:
            agent.store_in_memory(state, a, reward, new_state, done)
            agent.learn()

        state = new_state

    print("Total reward:", total_reward, "Epsilon:", agent.epsilon, "Size of replay buffer:",
len(agent.replay_buffer), "Learn step counter:", agent.learn_step_counter)
```

```
    if SHOULD_TRAIN and (i + 1) % CKPT_SAVE_INTERVAL == 0:
        agent.save_model(os.path.join(model_path, "model_" + str(i + 1) + "_iter.pt"))

    print("Total reward:", total_reward)

env.close()
```

Let's break down and explain the provided `main.py` script for training a Double Deep Q-Network (DDQN) agent to play Super Mario Bros using the gym environment:

### Import Statements

- **torch**: Imports the PyTorch library for building and training neural networks.
- **gym_super_mario_bros**: Imports the Super Mario Bros gym environment.
- **RIGHT_ONLY**: Imports an action set where the agent can only move to the right.
- **Agent**: Imports the Agent class from the `agent` module, which contains the implementation of the DDQN agent.
- **JoypadSpace**: Imports a wrapper from `nes_py` to restrict the action space.
- **apply_wrappers**: Imports a function from the `wrappers` module to apply a series of predefined wrappers to the environment.
- **os**: For handling directory creation and file paths.
- **get_current_date_time_string**: Imports a utility function to generate a timestamped string.

### Setting Up the Model Path

- **model_path**: Constructs the path to save the models using the current date and time.
- **os.makedirs**: Creates the directory for saving models if it doesn't already exist.

### Checking for CUDA Availability

- **torch.cuda.is_available()**: Checks if a CUDA-compatible GPU is available.
- Prints the name of the CUDA device if available, otherwise states that CUDA is not available.

### Constants and Configuration

- **ENV_NAME**: The name of the Mario environment to use.
- **SHOULD_TRAIN**: Flag to indicate if the model should be trained.
- **DISPLAY**: Flag to indicate if the environment should be rendered.
- **CKPT_SAVE_INTERVAL**: Interval for saving model checkpoints.
- **NUM_OF_EPISODES**: Number of episodes to run for training or evaluation.

### Environment Setup

- **env**: Creates and configures the Super Mario Bros environment with rendering options.
- **JoypadSpace**: Restricts the action space to only rightward movements.
- **apply_wrappers(env)**: Applies additional wrappers to the environment to modify observations and rewards.

### Agent Initialization

- **agent**: Initializes the agent with the environment's observation and action space dimensions.

### Loading Pre-trained Model (if not training)

- **SHOULD_TRAIN**: If `False`, loads a pre-trained model and sets epsilon-related parameters for evaluation.

### Main Training Loop

- **env.reset()**: Resets the environment to the initial state.
- **env.step(action=0)**: Takes an initial step in the environment.
- **for i in range(NUM_OF_EPISODES)**: Iterates over the number of episodes.
  - **state, _ = env.reset()**: Resets the environment at the beginning of each episode.
  - **while not done**: Loops until the episode is done.
    - **agent.choose_action(state)**: Chooses an action based on the current state.
    - **env.step(a)**: Executes the action in the environment and returns the new state, reward, and done status.
    - **agent.store_in_memory(state, a, reward, new_state, done)**: Stores the transition in the agent's replay buffer if training.
    - **agent.learn()**: Trains the agent by sampling from the replay buffer and updating the network if training.
    - **state = new_state**: Updates the current state.
  - **print()**: Outputs episode results, including total reward, epsilon, replay buffer size, and learning step counter.
  - **agent.save_model()**: Saves the model at regular intervals during training.

### Closing the Environment

- **env.close()**: Closes the environment after all episodes have been completed.

This script essentially sets up the environment and agent, then runs a training loop where the agent interacts with the environment, stores experiences, and learns from them to improve its policy over time. Checkpoints are saved periodically to allow for recovery and further training at a later time.

how the game starts

The game starts within the `main.py` script through a sequence of initialization steps and entering the main training loop. Here's a detailed breakdown of how the game starts:

1. **Import Required Libraries and Modules**:
   The script begins by importing necessary libraries and modules, including PyTorch, gym, and custom modules like the agent and wrappers.

2. **Setup Model Saving Directory**:

```python
model_path = os.path.join("models", get_current_date_time_string())
os.makedirs(model_path, exist_ok=True)
```

This creates a directory to save models using the current date and time for easy tracking.

3. **Check for CUDA Availability**:

```python
if torch.cuda.is_available():
    print("Using CUDA device:", torch.cuda.get_device_name(0))
else:
    print("CUDA is not available")
```

This checks if a CUDA-compatible GPU is available for faster computation and prints the device name if available.

4. **Define Constants and Configuration**:
Constants like the environment name, whether to train the model, display options, checkpoint saving intervals, and the number of episodes are defined.

```python
ENV_NAME = 'SuperMarioBros-1-1-v0'
SHOULD_TRAIN = True
DISPLAY = True
CKPT_SAVE_INTERVAL = 5000
NUM_OF_EPISODES = 50_000
```

5. **Initialize the Environment**:

```python
env = gym_super_mario_bros.make(ENV_NAME, render_mode='human' if DISPLAY else 'rgb',
apply_api_compatibility=True)
env = JoypadSpace(env, RIGHT_ONLY)
env = apply_wrappers(env)
```

This code sets up the Super Mario Bros environment, configures it to only allow rightward movements, and applies additional wrappers for observation and reward modifications.

6. **Initialize the Agent**:

```python
agent = Agent(input_dims=env.observation_space.shape, num_actions=env.action_space.n)
```

The agent is initialized with the dimensions of the observation space and the number of possible actions.

7. **Load Pre-trained Model (if not training)**:
   If `SHOULD_TRAIN` is set to `False`, a pre-trained model is loaded, and epsilon-related parameters are adjusted for evaluation.

```python
if not SHOULD_TRAIN:
    folder_name = ""
    ckpt_name = ""
    agent.load_model(os.path.join("models", folder_name, ckpt_name))
    agent.epsilon = 0.2
    agent.eps_min = 0.0
    agent.eps_decay = 0.0
```

8. **Initial Environment Reset and Step**:

```python
env.reset()
next_state, reward, done, trunc, info = env.step(action=0)
```

The environment is reset to the initial state, and an initial step is taken to begin the interaction.

9. **Main Training Loop**:

```python
for i in range(NUM_OF_EPISODES):
    print("Episode:", i)
    done = False
    state, _ = env.reset()
    total_reward = 0
    while not done:
        a = agent.choose_action(state)
        new_state, reward, done, truncated, info  = env.step(a)
        total_reward += reward

        if SHOULD_TRAIN:
            agent.store_in_memory(state, a, reward, new_state, done)
            agent.learn()

        state = new_state

    print("Total reward:", total_reward, "Epsilon:", agent.epsilon, "Size of replay
buffer:", len(agent.replay_buffer), "Learn step counter:", agent.learn_step_counter)

    if SHOULD_TRAIN and (i + 1) % CKPT_SAVE_INTERVAL == 0:
```

```
        agent.save_model(os.path.join(model_path, "model_" + str(i + 1) + "_iter.pt"))

    print("Total reward:", total_reward)
```

This loop iterates over the number of episodes defined. Within each episode:

- The environment is reset.
- The agent interacts with the environment by choosing actions and receiving rewards.
- If training is enabled, the agent stores experiences in memory and updates its policy through learning.
- Periodic checkpoint saves occur based on the defined interval.

10. **Closing the Environment**:

```python
env.close()
```

After completing all episodes, the environment is closed.

In summary, the game starts by initializing the environment, setting up the agent, and entering a loop where the agent interacts with the game, learns from its experiences (if training), and periodically saves its progress. The environment is displayed if `DISPLAY` is set to `True`, allowing you to see the agent's progress in playing the game.

do i need to install the super mario.exe game in my PC to run the game

No, you don't need to install the "Super Mario Bros" game on your PC to run the code. The code you provided is using the `gym_super_mario_bros` library, which provides a Python interface to the Super Mario Bros game environments. This library allows you to interact with the game environment directly through Python code, without the need for a separate game installation.

However, you do need the ROM files for the Super Mario Bros game, which contain the game data, such as level layouts, sprites, and game mechanics. These ROM files are not included with the `gym_super_mario_bros` library due to legal reasons. You can obtain the ROM files separately and configure the library to use them.

Once you have the ROM files and set up the environment properly, you can run the code to train your agent to play Super Mario Bros.

so how does it display the game on the screen

The code utilizes the `render_mode` parameter when creating the Super Mario Bros environment instance:

```python
env = gym_super_mario_bros.make(ENV_NAME, render_mode='human' if DISPLAY else 'rgb',
apply_api_compatibility=True)
```

When `DISPLAY` is set to `True`, the render mode is set to `'human'`, which allows the game environment to render and display the game graphics on the screen using the standard display capabilities of your Python environment. This typically involves opening a window or a viewport where the game frames are rendered.

Under the hood, the `gym_super_mario_bros` library leverages a rendering backend such as `pygame` or `OpenGL` to display the game frames. These libraries provide functions for drawing graphics to a window or viewport, which are used by the game environment to render the game frames.

If `DISPLAY` is set to `False`, the render mode is set to `'rgb'`, which means the game frames are rendered as RGB arrays but not displayed on the screen. This mode is useful when you want to train the agent without the overhead of rendering and displaying the game graphics.

then exactly what what things i needd in my PC for the project execution

To run the Super Mario Bros reinforcement learning project on your PC, you'll need:

1. **Python Environment**: Ensure you have Python installed on your PC. The code is written in Python.
2. **Required Python Libraries**: You'll need to install the necessary Python libraries specified in the `requirements.txt` file. You can do this by running `pip install -r requirements.txt` in your command line interface.
3. **NES Emulator**: The project uses an NES emulator to run the Super Mario Bros game. This is provided by the `gym_super_mario_bros` library. You don't need to install the NES emulator

separately, as it's included as part of the Gym environment.

4. **Super Mario Bros ROM**: The project requires the Super Mario Bros ROM file. This file contains the game data that the emulator uses to run the game. You'll need to legally acquire the ROM file and place it in the appropriate directory. The Gym environment will automatically locate and load the ROM file when you create the game environment.

5. **CUDA**: If you have an NVIDIA GPU and want to accelerate computations using CUDA, ensure you have the appropriate CUDA toolkit installed.

6. **Display**: If you want to visually observe the game while it's running, ensure you have a display available. The code can render the game frames on the screen if the `DISPLAY` flag is set to `True`.

With these components in place, you should be able to execute the project and train your reinforcement learning agent to play Super Mario Bros.

how does he code knows its agent ? is the agent is the character of the super Mario?

In the context of reinforcement learning, an "agent" refers to the entity that learns and interacts with the environment. In this Super Mario Bros reinforcement learning project, the "agent" is not the character of Super Mario himself; rather, it's the algorithmic entity that learns to play the game.

Here's how it works:

1. **Environment**: The Super Mario Bros game environment is provided by the `gym_super_mario_bros` library. This environment includes the game world, the actions that can be taken within the game (e.g., moving left, right, jumping), and the observations (e.g., screen pixels) provided to the agent.

2. **Agent**: The `Agent` class in the code represents the reinforcement learning agent. This is where the learning algorithm resides. The agent interacts with the environment by observing its state (e.g., game screen) and taking actions based on its policy (e.g., a neural network policy).

3. **Training Loop**: In the `main.py` file, there's a loop where the agent interacts with the environment over multiple episodes. During each episode, the agent observes the environment, selects actions, receives rewards, and learns from its experiences. The agent's goal is to maximize the cumulative reward it receives over time.

So, while the agent is not Super Mario himself, it learns to control Super Mario within the game environment to achieve a specific goal, which in this case is maximizing the score or completing the level.

OpenAI

agent slowly learns about the character ?

In a reinforcement learning setting like this Super Mario Bros project, the agent doesn't necessarily "learn about the character" in the same way a human player might. Instead, the agent learns a policy—a strategy for selecting actions in different situations—that allows it to play the game effectively.

Here's how the learning typically progresses:

1. **Exploration**: At the beginning of training, the agent often explores different actions randomly or according to some exploration strategy (like epsilon-greedy). During this phase, the agent gathers information about the consequences of its actions and the structure of the game environment.

2. **Exploitation**: As training progresses, the agent starts to exploit its learned knowledge to maximize rewards. It begins to choose actions that it predicts will lead to higher rewards based on its learned policy.

3. **Refinement**: Through a process of trial and error (reinforcement learning), the agent refines its policy over time. It learns from both successful and unsuccessful experiences, gradually improving its ability to navigate the game world and achieve its goals.

While the agent doesn't "learn about the character" in a cognitive sense, it does learn to effectively control the character within the game environment to accomplish specific objectives, such as completing levels or achieving high scores.

can you tell what happens when the main code is run for the first time

When the main code is run for the first time, here's what happens step by step:

1. **Imports**: The necessary libraries and modules are imported, including `` `torch` ``, `` `gym_super_mario_bros` ``, `` `Agent` `` from `` `agent` ``, `` `JoypadSpace` `` from `` `nes_py.wrappers` ``, and `` `apply_wrappers` `` from `` `wrappers` ``.

2. **Model Path Creation**: A directory path for saving the model is created. If it doesn't exist, a new directory is created.

3. **CUDA Availability Check**: The code checks if CUDA (NVIDIA's parallel computing platform) is available. If CUDA is available, it prints the name of the CUDA device being used; otherwise, it indicates that CUDA is not available.

4. **Environment Setup**: The Super Mario Bros environment is created using `gym_super_mario_bros.make()`, with the specified environment name (`SuperMarioBros-1-1-v0`) and render mode. If `DISPLAY` is set to `True`, the render mode is set to `'human'` (for human-readable rendering); otherwise, it's set to `'rgb'` for RGB rendering. The environment is then wrapped using `JoypadSpace` to map the discrete actions to the game's button presses.

5. **Wrapper Application**: The `apply_wrappers()` function is called to apply additional wrappers to the environment. These wrappers might include preprocessing steps or modifications to the observation space.

6. **Agent Initialization**: An instance of the `Agent` class is created, with the input dimensions taken from the observation space of the environment and the number of actions from the environment's action space.

7. **Model Loading (Optional)**: If `SHOULD_TRAIN` is set to `False`, the code loads a pre-trained model from the specified folder and checkpoint. It also sets the exploration parameters (`epsilon`, `eps_min`, `eps_decay`) accordingly.

8. **Environment Reset**: The environment is reset using `env.reset()`, returning the initial state and additional information.

9. **Initial Step**: An initial step is taken in the environment with a default action (in this case, action `0`). This returns the next state, reward, termination flag (`done`), truncation flag (`trunc`), and additional information (`info`).

10. **Training Loop**: The main training loop begins, running for a specified number of episodes (`NUM_OF_EPISODES`). Within each episode, the following steps are repeated:

    - Reset the environment and obtain the initial state.
    - Run a loop until the episode terminates:
        - Choose an action using the agent's policy.
        - Take a step in the environment using the chosen action.
        - Update the total reward.
        - If training, store the transition in memory and update the agent's parameters.
    - Print episode information, such as total reward, epsilon value, replay buffer size, and learn step counter.
    - Optionally save the model checkpoint at specified intervals.

11. **Environment Closure**: After training completes, the environment is closed using `env.close()`.

Throughout this process, the agent interacts with the environment, learns from the experiences, and updates its policy to improve its performance in the game.

Now tell the whole process in simple words for a 5 year old boy

Sure! Imagine you have a robot friend who wants to learn how to play a video game like Super Mario Bros. Here's what happens:

1. **Getting Ready**: First, we make sure the robot's brain is ready. We check if it can use a special helper called CUDA, which makes learning faster.
2. **Setting Up the Game**: We open the Super Mario Bros game and make sure it's ready for the robot to play. We also put on some special glasses so the robot can see the game.
3. **Getting Help**: The robot needs some extra help to play the game, so we give it a special controller. This controller helps the robot know which buttons to press to make Mario move.
4. **Putting on More Helpers**: We give the robot some extra tools to help it play the game even better. These tools make the game easier for the robot to understand.
5. **Starting to Learn**: The robot starts playing the game. It tries to figure out which moves are the best to help Mario win. Every time it plays, it learns something new.
6. **Playing Lots of Times**: The robot keeps playing the game over and over again. Each time, it learns a little bit more about how to win.
7. **Saving Progress**: Sometimes, when the robot does really well, we save its progress. This way, if it needs to stop and come back later, it can start from where it left off.
8. **Finishing Up**: After the robot has played enough and learned a lot, we stop the game. It's like saying, "Okay, we're all done for today."
9. **Turning Off**: Finally, we turn everything off and say goodbye to our robot friend until next time.

So, the robot learns how to play Super Mario Bros by practicing a lot and getting better each time!

now explain it to a cse students

Of course! Let's dive into a more technical explanation for computer science students:

1. **Setting Up Environment**: We start by setting up the environment for our reinforcement learning task. This involves importing necessary libraries like PyTorch for neural network operations and gym_super_mario_bros for accessing the Super Mario Bros game environment.

2. **Initializing Agent**: We create an agent object that will learn to play Super Mario Bros. This agent will use a neural network to make decisions about which actions to take in the game.

3. **Preparing Game Environment**: We configure the Super Mario Bros game environment, including setting the rendering mode and applying wrappers to preprocess the game data.

4. **Training Loop**: If specified, we enter a training loop where the agent learns to play the game. This loop involves episodes of gameplay, where the agent interacts with the environment, observes states, takes actions, and receives rewards.

5. **Agent-Environment Interaction**: During each episode, the agent chooses actions based on its current understanding of the game environment. It then receives feedback in the form of rewards and observes how the environment changes.

6. **Experience Replay**: To improve learning stability and efficiency, we use a replay buffer to store past experiences (state, action, reward, next state). The agent samples from this buffer to learn from a diverse set of experiences.

7. **Neural Network Training**: The agent's neural network is trained using a variant of the DQN (Deep Q-Network) algorithm. This involves updating the network's weights to minimize the difference between predicted and target Q-values.

8. **Model Saving**: Periodically, we save the agent's model parameters to disk to track its progress and enable resuming training from a checkpoint.

9. **Game Rendering**: If specified, we render the game screen to display the agent's gameplay in real-time.

10. **Completion and Cleanup**: Once the training loop is complete, or if training is disabled, we close the game environment and perform any necessary cleanup.

In summary, the code sets up the environment, initializes the agent, orchestrates the agent-environment interaction, trains the agent using reinforcement learning techniques, and manages the overall execution of the learning process.

when does te game stops

The game typically stops when one of the following conditions is met:

1. **Goal Achievement**: If the game has a specific goal or objective, such as reaching the end of a level, collecting a certain number of items, or defeating a final boss, the game stops when the goal is achieved.

2. **Player Failure**: If the player character (controlled by the agent in this case) loses all its lives or health points, falls into a pit, or encounters an enemy without successfully overcoming it, the game stops due to player failure.

3. **Time Limit Exceeded**: Some games impose a time limit for completing a level or task. If the time runs out before the player achieves the goal, the game stops due to time limit expiration.

In the case of Super Mario Bros, the game typically stops when Mario loses all his lives (e.g., gets hit by enemies too many times, falls into a pit), indicating player failure. However, the exact conditions for game termination can vary depending on the specific game settings and objectives.

what happens after it

After the game stops, the program might perform several actions:

1. **Evaluation**: It could evaluate the performance of the agent during the game. This evaluation might include metrics like the score achieved, the time taken to complete the level, or the number of enemies defeated.

2. **Learning**: If the program is using reinforcement learning, it may update the agent's policies and strategies based on the outcomes of the game. This involves analyzing what actions led to success or failure and adjusting the agent's behavior accordingly to improve its performance in future games.

3. **Recording**: The program might record data from the game, such as gameplay footage, states of the game environment, actions taken by the agent, and corresponding rewards received. This recorded data can be used for analysis, debugging, or training purposes.

4. **Decision Making**: Depending on the program's design, it might decide whether to start a new game, adjust certain parameters or strategies, or terminate the program altogether.

Overall, what happens after the game stops depends on the specific objectives and functionalities of the program, as well as its overall design and architecture.