**Submitted by:**

Lior Shilon 316126143

Yair Davidof 314792714

**Introduction:**

**The problem:**

Creating an AI model that will be able to play Mario.

**Methods for solving the problem:**

Two Reinforcement Learning algorithms:

1. PPO
2. DQN

Both algorithms were taken from the AI RL library Stable Baselines3 and the environment of the game was taken from <https://pypi.org/project/gym-super-mario-bros/> which is an Mario environment for OpenAI Gym.

We chose to use Reinforcement Learning because it’s a familiar way to solve those kind of problems, the agent will interact with the environment by doing actions in the game and he will learn from its past experience what he should and shouldn’t do in a specific state, which action is the best for a state.

We chose the algorithms above and the environment itself because the it is well documented, there are explanations of what to, how to do it, how the algorithms work and when the learning process of each algorithm ends, you get access to graphs that demonstrate parameters about the learning process you can use for comparison.

As for the algorithms they perform very well in solving this kind of problems.

**Environment setup:**

“Jupyter Notebook allows users to **compile all aspects of a data project in one place** making it easier to show the entire process of a project to your intended audience. Through the web-based application, users can create data visualizations and other components of a project to share with others via the platform”

We chose to work with Jupyer Notebook because it is convenient and friendly, it allows you to compile all aspect of a data project in one place.

We were able to use a lot of different libraries and make the libraries interact with each other.

Among the libraries:

* gym\_super\_mario\_bros
* nes\_py.wrappers
* PyTorch(in order to use stable\_baselines3).

**1)Setup Mario:**

a) First we installed gym\_super\_mario\_bros and nes-py which is an [OpenAI Gym](https://github.com/openai/gym) environment for Super Mario Bros. & Super Mario Bros. 2 (Lost Levels) on The Nintendo Entertainment System (NES) using [the nes-py emulator](https://github.com/Kautenja/nes-py), which let you play the game with python.

The code:

! pip install gym\_super\_mario\_bros==7.4.0 nes\_py

b)Than we imported some modules:

import gym\_super\_mario\_bros

from nes\_py.wrappers import JoypadSpace

from gym\_super\_mario\_bros.actions import SIMPLE\_MOVEMENT

c) We initialized the game:

env **=** gym\_super\_mario\_bros**.**make('SuperMarioBros-v0')

\*Gym is a toolkit for developing and comparing reinforcement learning algorithms.

By default gym\_super\_mario\_bros environment use actions space of 256 discrete actions which takes a lot of time for an AI model to learn and a lot of space in our computer, at least if we wanted the AI to make it through the first level, so we used the SIMPLE\_MOVEMENT actions list that contains 7 actions: [ [‘NOOP’], [‘right’], [‘right’, A], [‘right’, B], [‘right’, A, B], [A], [‘left’] ]

Which will decrease the time it takes for our agent to “learn”.

After that we wrapped (using nes\_py.wrappers) our current environment with the JoypadSpace

Providing him with the game and the allowed actions to perform.

env **=** JoypadSpace(env, SIMPLE\_MOVEMENT)

The JoypadSpace plays the game with the actions provided to him, It uses a method called: “actions\_space.sample()” which chooses a random action from the actions list and performs it in the game.

For our AI to learn we used a Reinforcement algorithm which uses a reward as a guideline.

The reward function:

The reward function assumes the objective of the game is to move as far right as possible (increase the agent's *x* value), as fast as possible, without dying. To model this game, three separate variables compose the reward:

1. *v*: the difference in agent *x* values between states
   * in this case this is instantaneous velocity for the given step
   * *v = x1 - x0*
     + *x0* is the x position before the step
     + *x1* is the x position after the step
   * moving right ⇔ *v > 0*
   * moving left ⇔ *v < 0*
   * not moving ⇔ *v = 0*
2. *c*: the difference in the game clock between frames
   * the penalty prevents the agent from standing still
   * *c = c0 - c1*
     + *c0* is the clock reading before the step
     + *c1* is the clock reading after the step
   * no clock tick ⇔ *c = 0*
   * clock tick ⇔ *c < 0*
3. *d*: a death penalty that penalizes the agent for dying in a state
   * this penalty encourages the agent to avoid death
   * alive ⇔ *d = 0*
   * dead ⇔ *d = -15*

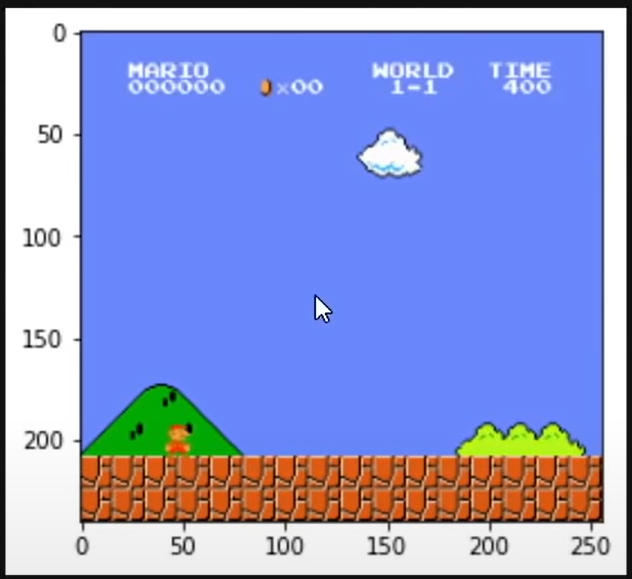
*r = v + c + d*

The reward is clipped into the range *(-15, 15)*.

The state:

The state is simply one frame of the game, the RGB values represented by matrices of numbers, here is an image representation of an actual state

from our program using matplotlib.



We used the following line of code to analyze the actions of our agent in the world:

state, reward, done, info **=** env**.**step(env**.**action\_space**.**sample())

The env variable is the JoypadSpace wrapping the Mario game.

This line of code performs a random step in the game and the variables: state, reward, done, info holds information about how the random step effected us.

State and reward are mentioned and explained above, “done” is a Boolean indicating if we are alive or not,

And info contains information about the world for example (x and y axis of our agent, collected coins, etc..).

**2)Preprocess Environment:**

The focus in this section is how we enabled our model to actually learn and how we improved the learning of our AI by neglecting variables that doesn’t contribute to our learning(insufficient data).

First we installed PyTorch:

**!**pip install torch**==**1.10.1+cu113 torchvision**==**0.11.2+cu113 torchaudio**===**0.10.1+cu113 -f https://download.pytorch.org/whl/cu113/torch\_stable.html

Than we installed stable-baselines3:

!pip install stable-baselines3[extra]

And installed other modules(explanation below):

from gym.wrappers import GrayScaleObservation

from stable\_baselines3.common.vec\_env import VecFrameStack, DummyVecEnv

pre-explanation:

We need to pre-process our Mario game data before we “AI-fy” it, we are going to apply two key pre-processing steps, gray scaling and frame stacking.

Our AI is going to be taking images of the Mario game to learn, a colored image has 3 times as many pixels to process, so converting it to gray scale cuts down the data it has to learn from.

Frame stacking helps out AI have a context, by stacking consecutive frames we are effectively giving our AI model memory, it will be able to see Mario and his enemies movements.

GrayScaleObservation allows us to convert our colored game into a gray scale version, which will cut down the amount of information our AI need to produce.

A colored image is effectively the height\*width\*3channels, because we need one channel per a color to represent red, green and blue. If we make it grayscale we actually cut down the amount of data by third.

Vectorization Wrappers:

When implemented our reinforcement learning model we needed to vectorize it in order to be able to actually use it with our AI.

So we used stable-baselines3, which is an AI library for reinforcement learning containing different algorithms.

VecFrameStack allows us to work with our stacked environments, it allows us to capture a couple of frames while we are playing Mario, this means that the AI model will be able to see what happened in the last x frames, where x is a number we define.

It will be able to see movements, otherwise lets say we pass just 1 frame to our AI it’s only going to know what’s happened right in this frame, so it doesn’t have any concept of movement or velocity, so we are going to stack frames together in order to train our AI.

DummyVecEnv just wraps out our base environment inside of a vectorization wrapper.

That’s is basically how we needed to transform our model in order to be able to pass it to the AI model.

Now lets take a look of how our environment looked so far:

*# 1. Create the base environment*

env **=** gym\_super\_mario\_bros**.**make('SuperMarioBros-v0')

*# 2. Simplify the controls*

env **=** JoypadSpace(env, SIMPLE\_MOVEMENT)

Using above modules our improved environment looks like this:

*# 3. Grayscale*

env **=** GrayScaleObservation(env, keep\_dim**=True**)

*# 4. Wrap inside the Dummy Environment*

env **=** DummyVecEnv([**lambda**: env])

*# 5. Stack the frames*

env **=** VecFrameStack(env, 4, channels\_order**=**'last')

So basically, we wrapped our environment with a grayscale which will improve learning performance ,transformed our model in order to be able to pass it to the AI model and added frame stacking(4 different frames stacked together) to give our AI better context.

**3)Train The RL Model:**

We trained the model using 2 different reinforcement learning algorithms form stable-baselines3:

1. PPO - Proximal Policy Optimization
2. DQN – Deep Q Network.

Explanation about the algorithms:

**PPO:**

Article:

<https://arxiv.org/pdf/1707.06347.pdf>

**Problems in reinforcement learning:**

1. The training data that is generated, is itself dependent on the current policy because our agent is generating its own training data by interacting with the environment rather than relying on a static data set as in the case of supervised learning do.

this means that the data distribution of our observations and rewards are constantly changing as our agent learns which is a major cause of instability in the whole training process

1. Reinforcement learning also suffers from a very high sensitivity to hyper parameter tuning and thing like initialization for example, in some cases it is kind of intuitive to understand why this happens, for example if our learning rate is to large we could have a policy update that pushes your policy network into a region of the parameter space where it’s going to collect the next batch of data under a very poor performance policy, causing it to never recover again

To address many of these problems in the reinforcement learning, the algorithm called PPO was designed by the Open Ai team.

The core purpose behind the PPO algorithm was to strike a balance between:

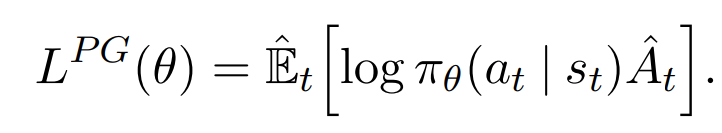
* + Ease of implementation
  + Sample Efficiency
  + Ease of tuning

PPO is a policy gradient method it means that unlike DQN for example that can learn from stored offline data PPO learns online means it doesn’t use a replay buffer to store past experiences but instead it learns directly from whatever its agent encounters in the environment and once a batch of experience was used to do a gradient update the experience is than discarded and the policy moves on.

This also means that policy gradient methods are typically less sample efficient than queue learning methods because they only use the collected experience once for doing an update.

**Defining the default policy gradient loss:**

**1.1**

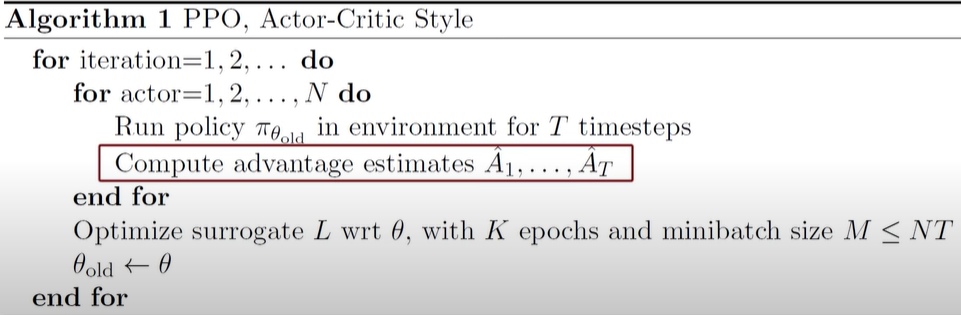


πθ - our policy, a neural network that takes the observed states in the environment as an input (sₜ) and suggests actions to take as an output(aₜ)(in our case the states are frames of the game, and the output is the action Mario should take in that frame)

Aₜ - The advantage function that tries to estimate what the relative value is of the selected action in the current state. where the little t is the current timestep.

In order to compute the reward, we need the discounted sum of rewards and a baseline estimate.

**The discounted sum of rewards** is a weighted sum of all the rewards the agent got during each timestep in the current episode , the calculation uses a gamma variable that goes between 0.9 and 0.99 accounts to the fact that the agent cares more about rewards that its going to get very quickly versus the same reward it would get a 100 timesteps from now.



The advantage is calculated after the episode sequence was collected form the environment therefore, we already know all the rewards and can calculate the discounted sum of rewards.

**The baseline estimate,** also called the value function, tries to estimate the discounted sum of rewards from this point onward, so basically it’s trying to guess what the final return is going to be in this episode starting from the current state.

During training the neural network, that’s representing the value function is going to be frequently updated using the experience that our agent collects from the environment.

So to summarize, we are taking states as an input and the neural network tries to predict what the discounted sum of rewards will be from this state onward, kind of like in supervised learning.

The estimate is going to be a noisy estimate because our network is not going to always predict the exact value of that states.

The advantage estimate provided by the advantage function is:

Aₜ = discounted sum of rewards - baseline estimate.

Which will point on how much better was the action that the agent took based on the expectation of what would normally happen in this specific state.

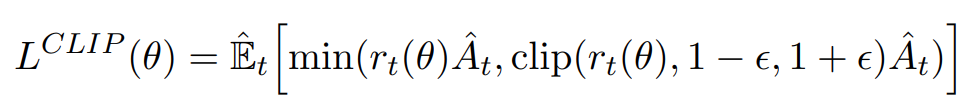
So basically when Aₜ is positive, it means the discounted sum of rewards is better than the baseline estimate, which from the equation above (See 1.1) will result to a positive outcome, therefore will increase the probability of selecting the actions that the agent took in said trajectory again in the future when the agent encounters the same state.

And when Aₜ is negative it implies that the actions the agent took during the trajectory are worse than the estimated return, which from the equation above (See 1.1) will result to a negative outcome, resulting decrease of the probability that said actions will be taken again in the future while encountering the same states.

To avoid to large policy updates, which can cause the returned value of the advantage function to be very noisy and lead to destroying your policy until the point it can no longer recover, PPO uses clipping, The main idea is that after an update, the new policy should be not too far from the old policy in order to obtain stable learning that can converge to a good model.

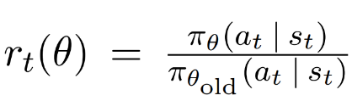
**Main objective function in PPO:**

**1.2**



PPO Computes this over a batches of trajectories

**Parameters explanation:**



a.

Probability ratio between the new updated policy outputs and the outputs of the previous, old version of the policy network.

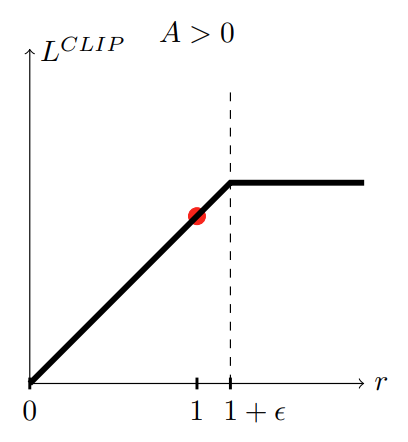
Means that the value will be larger than 1 if the actions of the agent are more likely now than it was in the old policy and the value will be between 0 and 1 if the actions are less likely now than it was in the old policy.

b. clip(x, a, b)

clips the value of x between a and b, it returns a if x < a, b if x > b, and x otherwise.

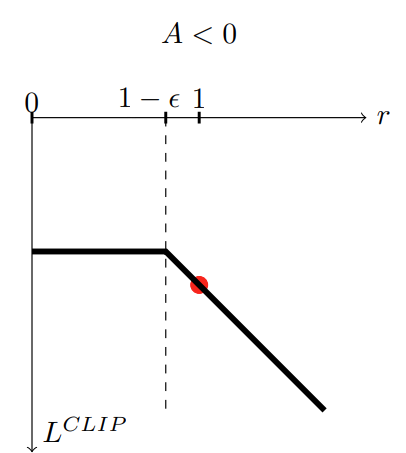
The whole purpose of this formula (see 1.2) is to prevent too big policy updates, if the update is to big it gets clipped to limit the effect of the gradient update.

The effect of the main objective function changes when the advantage is negative and when it is positive.



When the advantage is positive (A > 0), means the selected actions of the agent had a better than expected effect on the outcome, when r gets to big (x axis), means that the actions of the agent are **much** more likely now than it was in the old policy, clip the loss function value, in order to prevent a to big policy update/limit the effect of the gradient update.

Or in a simpler term, if the action was good and it became a lot more probable after the last gradient step don’t keep updating to much or else it might get worse. (Because we know that we have a very noisy advantage function and we don’t want to destroy a policy based on a single estimate).



When the advantage is negative (A < 0), means the selected actions of the agent had a worse than expected effect on the outcome

1. when r gets closer to 0, means that the actions are less likely now than it was in the old policy, clip the loss function value, prevent doing an update that will hurry to reduce the probability of the selected actions.
2. When r gets bigger, means that the likelihood of those actions is higher in the current policy than it was in the old policy, than we would like to undo the last gradient step therefore L\_clip ends up being negative thus telling us to go the other direction and make the action less probable by an amount proportional to the growth of r (seen by the linear line, when R is bigger than L\_clip is smaller).

This region in the graph is the only region where the left parameter of the min function is smaller than the right parameter (see 1.2) and thus get returned by the min function, which will cause the reduction in the likelihood of said actions.

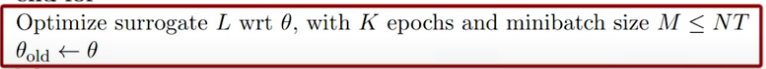
So, to summarize, the PPO objective forces the policy updates to be conservative if they move very far away from the current policy with a very simple objective function that don’t require much calculations.

Now back to the algorithm itself

Graphical user interface, text, application

Description automatically generated

There are 2 alternating threads in PPO, In the first one the current policy is interacting with the environment, generating episode sequences for which we immediately calculate the advantage function using the fitted baseline estimate for the state values.



And in every so many episodes the second thread is going to collect all this experience and run gradient descent on the policy network using the clipped PPO objective.

And now for the final loss function that is used to train an agent.



The loss function explained above that forces the policy updates to be conservative if - they move very far away from the current policy.

in charge of updating the baseline network based on how good it is to be in a certain state or more specifically what is the average amount of discounted rewards that I expect to get from this point onwards

the entropy term, in charge for making sure that the agent does enough exploration during training, so in contrast to discrete action policies that output the action choice probabilities the PPO policy head outputs the parameters of a gaussian distribution for each available action type and when running the agent in training mode the policy will sample from this distributions to get a continuous output value for each action head. Adding an entropy term will pus the policy to behave a little bit more randomly until the other parts of the objective start dominating, or in other words the model as started to converge to a good place.

In summary, PPO is a policy gradient method, the algorithm has a good stability and reliability, is simple to implement and can be used in a for a wide range of reinforcement learning tasks.

PPO is an on-policy algorithm that directly updates the policy network using the current trajectory of the agent. It uses a trust region optimization method to ensure that the updates to the policy network are conservative and do not deviate too much from the previous policy. This allows the agent to learn more efficiently and stabilize its learning process.

**DQN:**

Article:

<https://arxiv.org/pdf/1312.5602.pdf>

<https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>

**Problems in reinforcement learning:**

1. We don’t have a labeled data set, so how do we know which action is the correct action to perform.
2. Let’s say we have an agent and it has a task to achieve in the environment, if the environment is a small one and the task is relatively an easy task, we could easily traverse the environment multiple times to build a map of the world which will guide the agent to achieving his target, and create a lookup table which gives you the best option to perform in a certain state the agent is in, This approach is known as Q-learning.

But, if the environment is very complicated and there are a lot of actions the agent can perform and every combination of the allowed actions differentiate by the order of the actions the agent took, meaning different actions in different orders will result a different state, than we can’t create an infinite (or finite by way too large) lookup table for each state and the best action to perform.

**How does DQN handles the first problem:**

RL agent learns by trial and error, in our case the data will be generated by having Mario move around the environment and interact with it, every step that Mario takes in the environment generates a 4-tuple called “Experience” <s, a, s\_, r>.

Where s is the current state of the agent, a is the action that he performed, s\_ the new state as a result of the previous state and the action, and r is the reward obtained by performing said action.

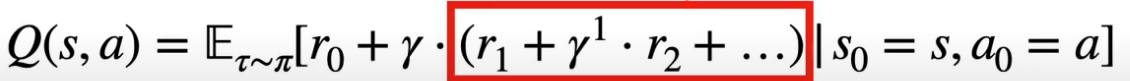
In DQN, we build our dataset by storing these experiences in a **replay Memory (the replay buffer).**

The replay buffer holds the last N interactions that the agent has with the environment, causing the agent to see this multiple times thus forcing it to learn from its past experiences.

So this is basically how the agent can learn from past experience, but how does it label the data.

DQN encodes the information via a state-action value function, also known as **Q-values** which are a numerical indicator determining how useful it is to perform a certain action in a given state in moving the agent toward its goal.

More formally, it is the expected discounted return of performing action ‘a’ in state ‘s’ .



We can see by the equation that the Q value of the current state and action depend on the Q value of the next state and action, which will depend on the next state and action and so on.

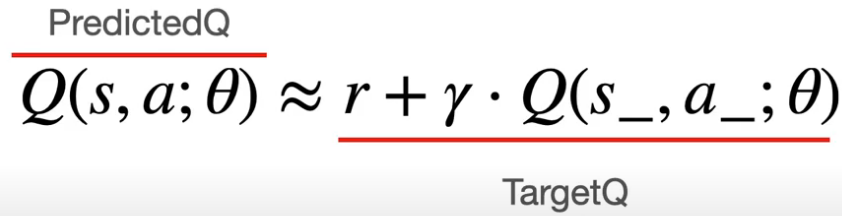
This recursive property helps us to define the labels to learn on.

**How does DQN handles the second problem:**

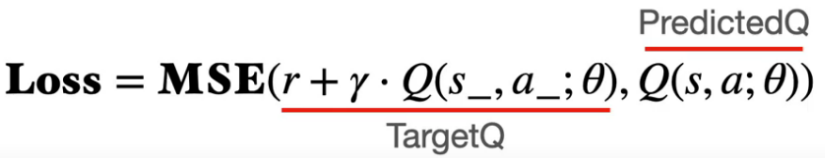
DQN handles the first problem by using neural networks to track the vast amount of information.

In DQN, the Q-value is estimated by the neural network, so instead of directly telling which action to perform, the neural network gives us an estimate of the Q-value of each of the possible actions, the agent than selects the action with the highest Q-value.

So the equation above can be updated to:



Now it takes into account the neural network, and after we know the target result and the predicted result the network learns by computing the mean squared error between the target value and the prediction value.



Thus, uses bootstrapping (A technique where an estimate updates another estimate)

It is important to note that this bootstrapping setup lead to some instability issues.

We have a network which is updated by computing the error between the predicted-Q and the target-Q but both this value are estimated using the same network, which means after the update the target-Q value on the very same state will be completely different.

This led to the predicted-Q value and the target-Q value constantly having to play catch up with each other.

To get over this problem, the developers of the algorithm introduced a second network, called the target network.

The target network is an exact copy of the already in use online network with one key difference- the target network never undergoes gradient updates.

Introducing a second network meant that the predicted-Q and target-Q values could now be computed by a separate network, thus keeping the target Q-values relatively constant.

**One more key element:**

We said earlier that the agent selects the action with the highest Q-value in a certain state, but to be more precise, this is what happens during the testing of the model.

In training, we do need the agent to explore the environment, to ensure that he can find a better possible route to the goal, DQN handles this by using **epsilon greedy policy**, means that based on a certain criteria, the agent decides whether to act randomly or to perform the action suggested by the network.

So, to summarize:

* + Data is generated via agent-environment interaction.
  + Data is stored on the Replay Memory.
  + Neural Network: Input = State; Output = Q-value per action
  + Bootstrapping can cause instability => use Target Network, whose sole purpose is to compute the target values.

**Dive deep into DQN algorithm:**

Text, letter

Description automatically generatedThe algorithm (from the second article mentioned above)

The Replay Memory:

We already established that the Replay Memory is used to store each of the interactions that the agent has with the environment, the Replay Memory is limited to a certain size defined by the user, as the memory fills up all the experiences are removed to make space for the newer ones.

What to do with the samples In the Replay Memory?

We can look at the Replay Memory as an equivalent to the dataset in a supervised learning setting, thus for the agent to learn from this data, at each iteration we need to feed the neural network mini batches of data which are the samples for it to optimize on, this mini batch is created by randomly sampling from the Replay Memory, the samples are chosen in a random manner because it helps to break correlations in data samples, thus it becomes more sample efficient.

So, the use of the Replay Memory ensures that the agent sees each data point multiple times before the data point is removed from memory.

The Neural Network:

Uses Online Network and a Target Network to ensure stability in training, these two are instances of the same network with the update technique being the only difference between them.

The output of the neural network determines how the agent will act, the selected action is the action with the highest Q-value, at least during testing. And during training the algorithm also encourage exploration by an Epsilon-Greedy Policy.

The Epsilon-Greedy Policy works by initializing a threshold value called epsilon, at each step we generate a random number, if the random number is smaller than epsilon than we perform a random action (exploring the environment) otherwise the algorithm will perform the action suggested by the neural network.

Over time, as the model progresses, we would like to rely more on the information we gathered than on random exploration. It is achieved by annealing the epsilon value, so as the agent learns more about the environment, we reduce the epsilon value until it finally reaches a very small number, which will cause the choosing of the random actions (exploration actions) to decrease and relying more on the neural network outputs.

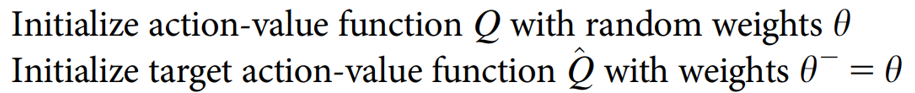
The Q-values:

Numerically indicate how good certain action is in a given state by computing the total discounted reward the agent can expect in the future if it follows the current policy.

The algorithm (See above):



1. initialize the replay memory, done by executing s completely random actions for a few timesteps.



1. creating the online network and the target network and set the weights of the target network to be the same as the weights of the online network.

Logo

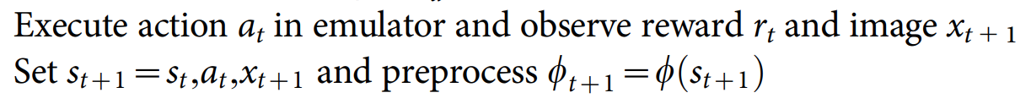
Description automatically generated with medium confidence

1. as part of the training loop, start gathering current observation ‘s, the outer for loop tracks the total number of steps that need to be performed, while the inner for loop tracks the steps in the current episode until terminal state reached.

Text

Description automatically generated

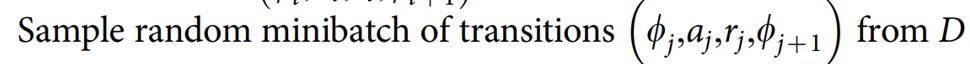
1. At each timestep use the epsilon greedy strategy to determine whether to perform a random action or act as the neural network output



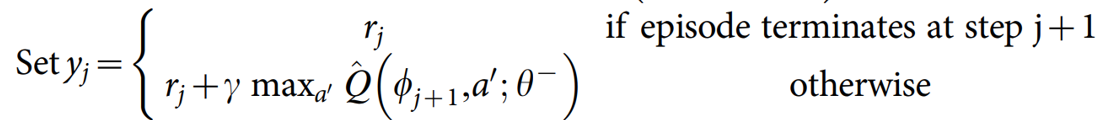
1. Execute the selected action, to obtain the immediate reward and the next state

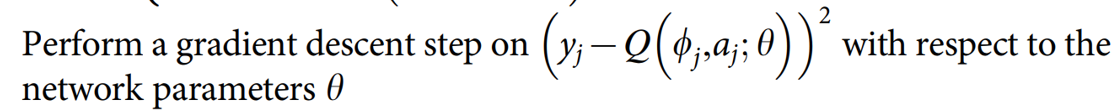


1. Store this experience of state, action, reward and next state into the replay memory.



1. Draw a mini batch of random samples from the memory



1. Compute the target-Q value using the target network, by accounting for terminal states
2. Compute the predicted Q-value using the online network and compute the loss between the target and predicted Q’s to update the weights of the online network.



1. At regular intervals, copy the weights of the online network into the target network.

In summary, DQN is an off-policy algorithm that uses a replay buffer to store past experiences and samples a batch of these experiences to compute the Q-values for each action. It then uses these Q-values to update the action-value function using the Bellman equation. This allows the agent to learn from past experiences and improve its policy over time.

So after this long explanation about both algorithms let’s go to back to training the RL model:

**Training the model using PPO algorithm:**

After we have our environment set up, we will now import some stuff.

*# Import os for file path management*

**import** os

*# Import PPO for algos*

**from** stable\_baselines3 **import** PPO

*# Import Base Callback for saving models*

**from** stable\_baselines3.common.callbacks **import** BaseCallback

We imported “os” and BaseCallback to save our model and information about the training, and we imported the PPO algorithm which will be used for the learning process.

**class** TrainAndLoggingCallback(BaseCallback):

**def** \_\_init\_\_(self, check\_freq, save\_path, verbose**=**1):

super(TrainAndLoggingCallback, self)**.**\_\_init\_\_(verbose)

self**.**check\_freq **=** check\_freq

self**.**save\_path **=** save\_path

**def** \_init\_callback(self):

**if** self**.**save\_path **is** **not** **None**:

os**.**makedirs(self**.**save\_path, exist\_ok**=True**)

**def** \_on\_step(self):

**if** self**.**n\_calls **%** self**.**check\_freq **==** 0:

model\_path **=** os**.**path**.**join(self**.**save\_path, 'best\_model\_{}'**.**format(self**.**n\_calls))

self**.**model**.**save(model\_path)

**return** **True**

Than we defined a callback which will allow us to save our model every so and so steps, the number of steps will be determined by the value of the argument check\_freq and the place where our model will be saved will be determined by the value of the argument save\_path.

Than we define to variables that will hold a string representation of the folders names to save the model and the log files.

CHECKPOINT\_DIR **=** './ppo\_train/'

LOG\_DIR **=** './ppo\_logs/'

And defined the callback variable which will cause the model to be saved automatically every 50,000 steps in the ppo\_train folder.

*# Setup model saving callback*

callback **=** TrainAndLoggingCallback(check\_freq**=**50000,save\_path**=**CHECKPOINT\_DIR)

Than we initialized the model providing it the neural network CnnPolicy and the environment, and overridden some default parameters.

*# This is the AI model started*

model **=** PPO('CnnPolicy', env, verbose**=**1, tensorboard\_log**=**LOG\_DIR, learning\_rate**=**0.000001,n\_steps**=**512)

* + CnnPolicy – a neural network that is very fast in processing images (the frames of the game).

And started the learning process of the model by telling it to run for 1,000,000 steps

*# Train the AI model, this is where the AI model starts to learn*

model**.**learn(total\_timesteps**=**1000000, callback**=**callback)

In the second part of the learning, we loaded the 1,000,000 model that was saved after the first part of training and told it to learn for another 1,000,000 steps so in total we ran 2,000,000 steps.

And saved the model in a folder called ppo\_train2

CHECKPOINT\_DIR **=** './ppo\_train2/'

callback **=** TrainAndLoggingCallback(check\_freq**=**50000,save\_path**=**CHECKPOINT\_DIR)

*# load the model*

model **=** PPO.load(‘./ppo\_train/best\_model\_1000000’)

model.set\_env(env)

model**.**learn(total\_timesteps**=**1000000, callback**=**callback)

• Side Note:

Those are the default parameters of the PPO algorithm we used

PPO(policy, env, learning\_rate=0.0003, n\_steps=2048, batch\_size=64, n\_epochs=10, gamma=0.99, gae\_lambda=0.95, clip\_range=0.2, clip\_range\_vf=None, normalize\_advantage=True, ent\_coef=0.0, vf\_coef=0.5, max\_grad\_norm=0.5, use\_sde=False, sde\_sample\_freq=-1, target\_kl=None, tensorboard\_log=None, policy\_kwargs=None, verbose=0, seed=None, device='auto', \_init\_setup\_model=True)

We overridden only below values:

verbose=1,

tensorboard\_log=LOG\_DIR,

learning\_rate=0.000001,

n\_steps=512

**Training the model using DQN algorithm:**

First, we imported the DQN algorithm

*# Import DQN for algos*

**from** stable\_baselines3 **import** DQN

Then we created the base environment again

*# 1. Create the base environment*

env **=** gym\_super\_mario\_bros**.**make('SuperMarioBros-v0')

*# 2. Simplify the controls*

env **=** JoypadSpace(env, SIMPLE\_MOVEMENT)

*# 3. Grayscale*

env **=** GrayScaleObservation(env, keep\_dim**=True**)

*# 4. Wrap inside the Dummy Environment*

env **=** DummyVecEnv([**lambda**: env])

*# 5. Stack the frames*

env **=** VecFrameStack(env, 4, channels\_order**=**'last')

And defined new folders for the training data and the place where we save the model

CHECKPOINT\_DIR **=** './dqn\_train/'

LOG\_DIR **=** './dqn\_logs/'

Then, we initialized the callback again with the new folders

callback **=** TrainAndLoggingCallback(check\_freq**=**50000,save\_path**=**CHECKPOINT\_DIR)

And initialized the model with the same neural network and learning rate and overridden the default buffer size from 1,000,000 to be 10,000 (because our computer didn’t have enough RAM).

model **=** DQN('CnnPolicy', env, buffer\_size=10000, verbose**=**1, tensorboard\_log**=**LOG\_DIR, learning\_rate**=**0.000001)

Then, started the learning process of the model by telling it to run for 1,000,000 steps(just like as we did with the PPO algorithm)

*# Train the AI model, this is where the AI model starts to learn*

model**.**learn(total\_timesteps**=**1000000, callback**=**callback)

In the second part of the learning, we loaded the 1,000,000 model that was saved after the first part of training and told it to learn for another 1,000,000 steps so in total we ran 2,000,000 steps.

And saved the model in a folder called dqn\_train2

CHECKPOINT\_DIR **=** './dqn\_train2/'

callback **=** TrainAndLoggingCallback(check\_freq**=**50000,save\_path**=**CHECKPOINT\_DIR)

*# load the model*

model **=** DQN.load(‘./dqn\_train/best\_model\_1000000’)

model.set\_env(env)

model**.**learn(total\_timesteps**=**1000000, callback**=**callback)

• Side Note:

Those are the default parameters of the DQN algorithm we used

DQN(policy, env, learning\_rate=0.0001, buffer\_size=1000000, learning\_starts=50000, batch\_size=32, tau=1.0, gamma=0.99, train\_freq=4, gradient\_steps=1, replay\_buffer\_class=None, replay\_buffer\_kwargs=None, optimize\_memory\_usage=False, target\_update\_interval=10000, exploration\_fraction=0.1, exploration\_initial\_eps=1.0, exploration\_final\_eps=0.05, max\_grad\_norm=10, tensorboard\_log=None, policy\_kwargs=None, verbose=0, seed=None, device='auto', \_init\_setup\_model=True)

We overridden only below values:

buffer\_size=10000,

verbose**=**1,

tensorboard\_log**=**LOG\_DIR,

learning\_rate**=**0.000001

So, to summarize, we let each algorithm (PPO, DQN) run for total of 2,000,000 steps in 2 learning parts, where each part contained 1,000,000 steps, and we saved our model each 50,000 steps, therefore we have a total of 40 models created by each algorithm.

We will load and run those models on the game environment for making a comparison between the real time results each algorithm achieved.

**4) Test out The RL Models:**

In order to test and run the two models we trained we ran again the following lines of code in jupyter notebook:

# Import the game

import gym\_super\_mario\_bros

# Import the Joypad wrapper

from nes\_py.wrappers import JoypadSpace

# Import the SIMPLIFIED controls

from gym\_super\_mario\_bros.actions import SIMPLE\_MOVEMENT

# Import GrayScaling Wrapper

from gym.wrappers import GrayScaleObservation

# Import Vectorization Wrappers

from stable\_baselines3.common.vec\_env import VecFrameStack, DummyVecEnv

# Import DQN for algos

from stable\_baselines3 import DQN

# Import PPO for algos

from stable\_baselines3 import PPO

Than we loaded the environment of the game the model will be tested on

# Create the base environment

env = gym\_super\_mario\_bros.make('SuperMarioBros-v0')

# Simplify the controlls

env = JoypadSpace(env, SIMPLE\_MOVEMENT)

# Grayscale, we need the keep\_dim=True to be able to use are Stack

env = GrayScaleObservation(env, keep\_dim=True)

# Wrap inside the dummy environment

env = DummyVecEnv([lambda: env])

# Stack the frames

env = VecFrameStack(env, 4, channels\_order='last')

**Loading and running the DQN model:**

As we mentioned our DQN models were saved in 2 different directories, each directory had 20 model and was storing the information of 1,000,000 timesteps.

The names of the directories:

* + dqn\_train
  + dqn\_train2

each model was called best\_model\_x , where x is the number of timesteps in the current run when the model was saved.

So, in order to load the DQN model we used the following line of code:

# Load model

model = DQN.load('./dqn\_train2/best\_model\_1000000')

In this example we loaded the 1,000,000 timesteps model of the second part of the training(dqn\_train2), which means the 2,000,000 timesteps model.

And in order to run the model and test it we ran the following lines of code :

obs = env.reset()

# Loop through the game

while True:

# Predict the action that the agent should do in the current state # based on the model

action, \_states = model.predict(obs, deterministic=False)

# make the action the model predicted in the game

obs, reward, done, info = env.step(action)

env.render()

# If Mario died after the last action reset the game

if done:

obs = env.reset()

**Loading and running the PPO model:**

As we mentioned our PPO models were saved in 2 different directories, each directory had 20 model and was storing the information of 1,000,000 timesteps.

The names of the directories:

* + ppo\_train
  + ppo\_train2

each model was called best\_model\_x , where x is the number of timesteps in the current run when the model was saved.

So, in order to load the PPO model we used the following line of code:

# Load model

model = PPO.load('./ppo\_train2/best\_model\_1000000')

In this example we loaded the 1,000,000 timesteps model of the second part of the training(ppo\_train2), which means the 2,000,000 timesteps model.

And in order to run the model and test it we ran the following lines of code :

# Start the game

state = env.reset()

# Loop through the game

while True:

# Predict the action that the agent should do in the current state # based on the model

action, \_ = model.predict(state)

# make the action the model predicted in the game

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

env.render()

**Design dilemmas we had:**

Because we used a very well known and documented RL library we didn’t really have design dilemmas, we followed the documentation of how to set the environment and the AI model.

But we did encounter some problems in the process of the environment set up and the reloading of the model for the purpose of re-training it.

When we tried to set up the environment, because the program uses a lot of different libraries and tools, we had to install them from the cmd using pip (python), as we started installing more and more modules we encountered software compatibility issues, some modules were needed other modules in old versions in order to function together so we had to read about the versions needed to be installed in order to achieve full project compatibility.

When we tried to reload the model and re-train it , first we loaded the model

for example:

model = PPO.load('./ppo\_train2/best\_model\_1000000')

Then we tried to start re-train it, so we used:

model.learn(total\_timesteps=1000000, callback=callback)

And an error jumped to the screen, we did a little bit of web research and discovered that after loading the model, in order to re-train it, we needed to set the environment of the model, because the environment isn’t saved in the model.

So, to solve this problem, we needed to use the below line of code between the two above lines of code.

model.set\_env(env)

Where the env variable is the game itself wrapped by all the different wrappers mentioned above (grayscale, framestack …).

**PPO vs DQN:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **PPO** | **DQN** | **Implications** |
| Type of algorithm | On-policy | Off-policy | PPO is more sample efficient because it uses the current policy to generate data for the update. DQN is off-policy algorithm that uses a replay buffer to store and reuse past experiences. |
| Sample efficiency | High | Medium | PPO can achieve better performance with less data, DQN requires more data to achieve similar performance. ( PPO uses the current policy to generate data for the update, which makes it more sample efficient. DQN uses a replay buffer to store and reuse past experiences, which requires more data to achieve similar performance) |
| Robustness to hyperparameters | High | Medium | PPO is less sensitive to the choice of hyperparameters, which makes it easier to train and tune. DQN may require more fine-tuning of the hyperparameters. (PPO uses a trust region optimization method which controls the step size) |
| Handling high-dimensional action spaces | Good | Not recommended | PPO can handle high-dimensional action spaces more effectively than DQN, which makes it suitable for a wide range of tasks. DQN is not recommended for tasks with high-dimensional action spaces. |
| Suitable for complex dynamics | Yes | No | PPO is known to have good performance in tasks that have complex dynamics, which makes it a great choice for many real-world applications. DQN is not suitable for tasks with complex dynamics. |
| Suitable for real-world applications | Yes | Yes | PPO is a good choice for real-world applications with complex dynamics and high-dimensional action spaces. DQN is good for tasks without those characteristics and it's more suitable for Atari games. |
| Trust region optimization | Yes | No | PPO uses a trust region optimization method which controls the step size of the update, allowing for more stable and robust training. DQN does not use this method. |
| Learning rate sensitivity | Low | Medium |  |

**Differences between the models:**

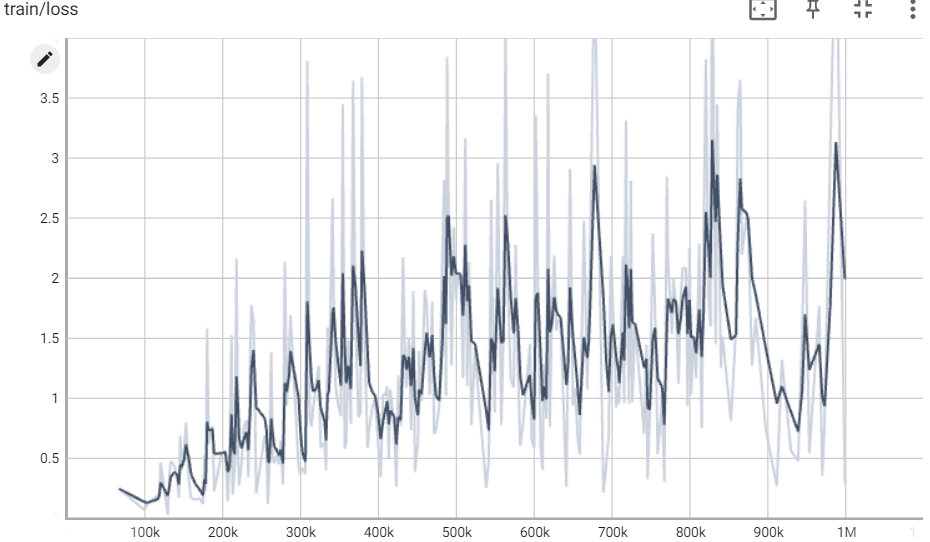
**Chart, line chart

Description automatically generated**

**DQN iteration 1 (0 – 1,000,000 timesteps):**

Chart, line chart

Description automatically generated



**DQN iteration 2 (1,000,000 – 2,000,000 timesteps):**

Chart, line chart

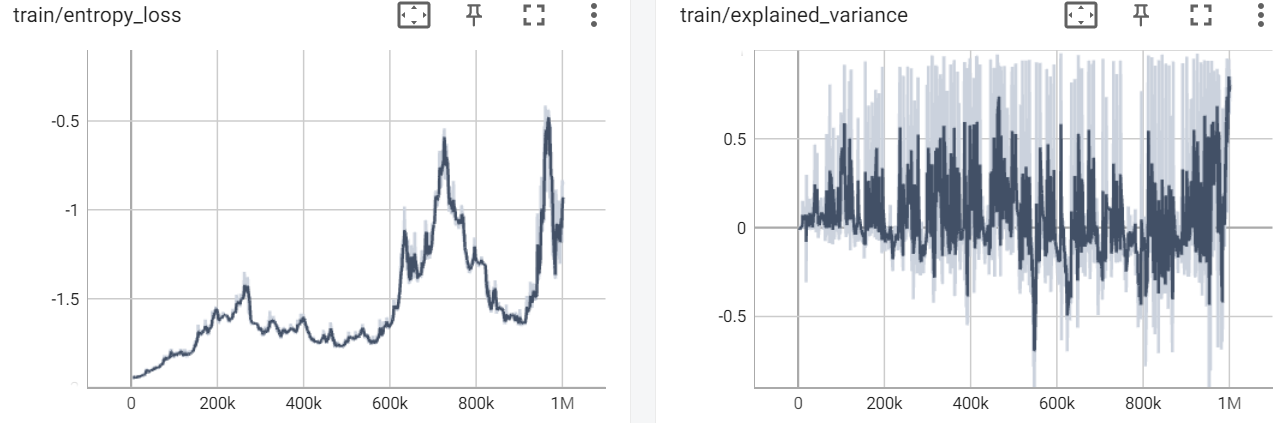
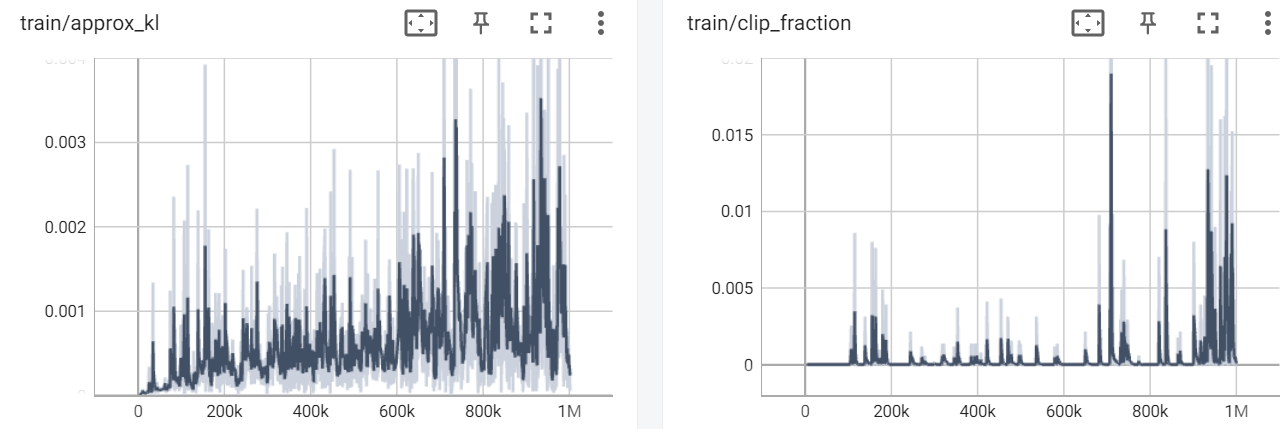
Description automatically generated

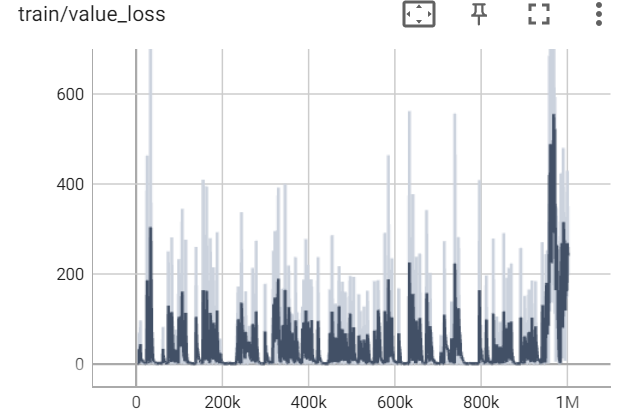
Chart, line chart

Description automatically generated

Chart, histogram

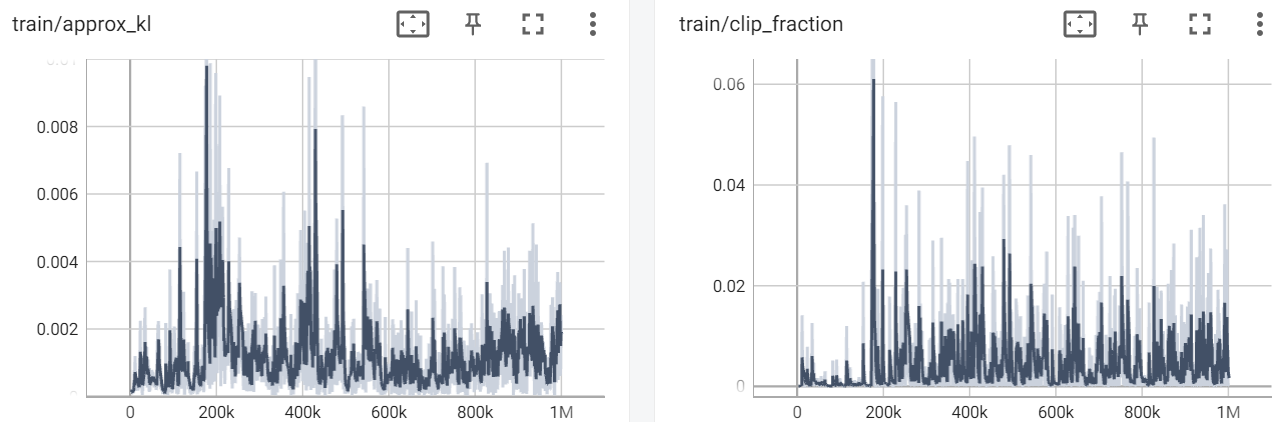
Description automatically generated

**PPO iteration 1 (0 – 1,000,000 timesteps):**

Chart, histogram

Description automatically generatedChart

Description automatically generated

**PPO iteration 2 (1,000,000 – 2,000,000 timesteps):**

Chart, bar chart

Description automatically generatedChart

Description automatically generated

Chart, histogram

Description automatically generatedChart, bar chart

Description automatically generated

* + Note:

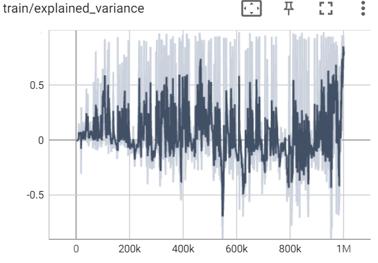
We took the graphs from the log files each algorithm saved during the training, we installed (using pip) tensorboard and tesnorflow, in the cmd went to the path of the log files and ran following command:

tensorboard --logdir=.

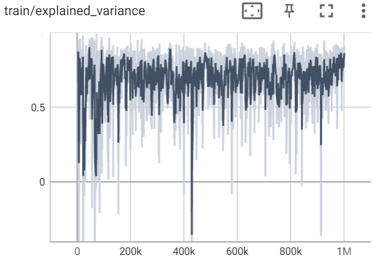
**Our Assumptions regarding the models based on the graphs:**

**PPO:**

The explained\_variance graph after 1,000,000 timesteps:



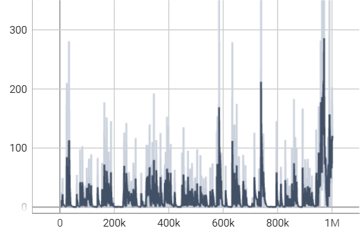
The explained\_variance graph after 2,000,000 timesteps:



The explained\_variance parameter represents the explained variance of the value function. It measures the degree to which the value function is able to predict the expected returns.

In the first graph we can see that the explained variance is very unstable, means that the model doesn’t make such a good prediction of expected returns, as the learning process continues, in the second graph we can see that the explained variance went up and become more stable, means that the model improved it’s ability to predict the expected returns, thus indicates that the learning process is going well (The model learns).

The loss graph after 1,000,000 timesteps:



The loss graph after 2,000,000 timesteps:

**Chart, bar chart

Description automatically generated**

This parameter represents the overall loss of the model, which is a combination of the policy gradient loss, value loss, and entropy loss.

We can see in both graphs that the loss value doesn’t decrease over time and actually it increases over time which can suggest on a poor learning or a need in change of the hyperparameters.

Even above statements, we severe it’s ok and only natural for an agent that is learning from trial and error and that is always expected to encounter new states (whenever it reaches to a point in the game he has never seen before) to make mistakes when it tries to predict the best action to take in these states.

**DQN:**

The loss parameter graph after 1,000,000 steps:

A picture containing bar chart

Description automatically generated

The loss parameter graph after 2,000,000 steps:

Chart, histogram

Description automatically generated

The loss parameter is a measure of how well the model is able to predict the Q-values of the state-action pairs. It is an indicator of how well the model is learning. If the loss is high, it means that the model is not able to predict the Q-values well, and if the loss is low, it means that the model is able to predict the Q-values well.

According to this graph, we can see the loss value goes up instead of down which points on a problem in the learning process, the model doesn’t predict the Q-Values well,

We think it arises from the change of the DQN Memory Buffer, it’s default (and recommended) size is 1,000,000 and we reduced it to 10,000 because of the available RAM size our computers have.

We think this change led to the fact that the stored experience DQN hold in the memory should be used fir learning purposes, so maybe decreasing its size caused some experience to not be in the memory for long enough for the model to learn from it, we severe the experience is deleted from the memory to fast for making room to newer experience.

But we can that the second graph is more stable than the first one, which can point on the fact that the model improved it’s ability to predict the Q-Values.

**Real Time Comparison Between the Models:**

The comparison of both model will be done as follows:

We will compare each DQN model with its equivalent PPO model, equivalency will be based on the timesteps.

Because of time limitations, instead of comparing and analyzing every model we saved, we will compare only models that were saved after : 50,000 ; 500,000 ; 1,000,000 ; 2,000,000 timesteps.

We will conduct regular tests and 2 special tests.

1. **Regular Tests:**

Each regular test will include a link to a google drive folder that holds the videos of the 2 compared models running and playing the game.

The factors for comparison in each test will be:

* + Most distance passed in the x-axis.
  + Dealing with obstacles (pipes, enemies, pits).
  + Highest score.
  + Rare staff findings (affected by the exploration rate).

1. **Special Tests**:

Each special tests will include a link to a google drive folder that holds the videos of the test.

The factors for comparison in each test will be:

* + Can the model reach to level 2 of the game?
  + Testing each model on a new Mario environment the model never played or seen before (switching the environment from “Super Mario Bros1” to “Super Mario Bros2”) and testing according to regular test parameters.

**Test 1:**

DQN vs PPO (50,000 timesteps models):

Link to videos: <https://drive.google.com/drive/folders/15LIt3X1o0mWvSrYlCl6KVjY8fK7f4GAI>

**Most distance passed in the x-axis:**

* + DQN:

Reached to the first green pipe.

* + PP0:

Reached to the second green pipe.

**Dealing with obstacles:**

* + DQN:

Killed the first enemy mushroom and got stuck in the first green pipe.

* + PP0:

Avoided the first enemy and Reached to the second green pipe.

**Highest score:**

* + DQN:

100 points.

* + PP0:

0 points.

**Rare staff findings:**

None found a rare item yet.

**Conclusions:**

Both algorithms are in the start of their learning process though we can see they managed to learn that jumping is good and that they need to jump in order to avoid some obstacles, PPO, in contrast to DQN learned how to jump higher and pass the first pipe, but DQN learned that he can jump on enemies, kill them and earn reward for that.

**Test 2:**

DQN vs PPO (500,000 timesteps models):

Link to videos: <https://drive.google.com/drive/folders/1IgGfwj4y-lDYyMOPbEvrP5b6DUcX85I1>

**Most distance passed in the x-axis:**

* + DQN:

Reached to the first pit, couldn’t pass it.

* + PP0:

Reached to the second pipe again, couldn’t pass it.

**Dealing with obstacles:**

* + DQN:

Deals with pipes better than in the first test, learned to jump high.

Deals with enemies pretty good, jumps over them or kills them, but It’s still unstable, meaning sometimes he just run to them and die.

* + PP0:

Deals with pipes very weakly, still couldn’t pass the second pipe means it can’t jump high yet.

Because it didn’t make it through the second pipe we can’t test dealing with enemies.

**Highest score:**

* + DQN:

200 points.

* + PP0:

100 points.

**Rare staff findings:**

None found a rare item yet.

**Conclusions:**

DQN is better than PPO on every aspect of the parameters and act the same when talking about finding rare items.

DQN is dealing with jumps and enemies better thus making it the better model.

**Test 3:**

DQN vs PPO (1,000,000 timesteps models):

Link to videos: <https://drive.google.com/drive/folders/1FeGSLYuTDzc4-o8MR_FRU0zwIjjCYLLu>

**Most distance passed in the x-axis:**

* + DQN:

Reached to the first pit, couldn’t pass it but in comparison to the last test tried to jump above the pit and failed .

* + PP0:

Passed the first and second pits.

**Dealing with obstacles:**

* + DQN:

Ability to deal with pipes improved from the second test, learned to jump higher and smoother (See second 16 in video DQN\_1M).

Deals with enemies pretty good, jumps over them or kills them, but It’s still unstable, meaning sometimes he just run to them and die.

Tries to jump over the pit instead of just walking and falling to it, but still couldn’t pass it.

* + PP0:

Dealing with pipes is good but not stable.

Dealing with enemies improved from the last test, kills and jumps over enemies.

Deals with pits pretty good when encounters them.

**Highest score:**

* + DQN:

400 points.

* + PP0:

1000 points.

**Rare staff findings:**

* + DQN:

Couldn’t find any rare items.

* + PP0:

Found 2 rare mushrooms that 1 of them was invisible, but didn’t take them

**Conclusions:**

DQN remained pretty stable with a little better adjustments, it leaned to coordinate jumps according to the agent’s velocity, which looks like almost a human made that jump and tried to jump over the second pit but still couldn’t pass it

On the other hand, PPO performance has spiked, in the previous test it couldn’t pass the second green pipe, and now it passed the DQN model performance in most parameter aspects, except that the DQN model jumps better.

**Test 4:**

DQN vs PPO (2,000,000 timesteps models):

Link to videos: <https://drive.google.com/drive/folders/1DahLwhwmQkhMt68W-FHu1cHRI16JpsK2>

**Most distance passed in the x-axis:**

* + DQN:

Reached to the second pit and couldn’t pass it, managed to pass the first pit.

* + PP0:

Passed the second pit and the turtle and couldn’t pass the 3 enemy mushrooms wave (See second 2:11 in video PPO\_2M).

**Dealing with obstacles:**

* + DQN:

Dealing with pipes and enemies very smoothly.

Succeeded to pass the first pit but couldn’t pass the second pit.

* + PP0:

Dealing with enemies and pipes is unstable, sometimes deals with them in a good way and sometimes doesn’t.

Deals with the pits very good.

Overall performance is good but unstable.

**Highest score:**

* + DQN:

500 points.

* + PP0:

500 points.

**Rare staff findings:**

* + DQN:

Couldn’t find any rare items.

* + PP0:

Couldn’t find any rare items.

**Conclusions:**

The DQN model is stable, handles enemies and pipes very well, still has problems passing the pits, but overall, its performance is good.

The PPO model handles enemies and jumps in an unstable way, sometimes it does better than other times, delas with the pits very good and manages to pass the biggest distance (x-axis) so, overall had better performance than the DQN.

**Special Test 1:**

Question: Can the model reach to level 2 of the game?

Link to video: <https://drive.google.com/drive/folders/16b4b_WWJjH9LUT7HvEqmsma83dRdXtH1>

We ran the models for a lot of time in order to see if one of them can make it to level 2.

**DQN:**

The model never succeeded to pass the second pit.

**PPO:**

After 1 hour of model testing the model made it to level 2 (See the link to the video)

**Conclusions:**

PPO won the test.

**Special Test 2:**

Testing each model on a new Mario environment the model has never played or seen before called Super Mario Bros 2.

Link to videos: <https://drive.google.com/drive/folders/1P8qXPh-4HX2zOYbFn1F_yWgETQvADqKd>

* + Note: the test will be done on the 2,000,000 timesteps models.

**Most distance passed in the x-axis:**

* + DQN:

Passed the first long green pipe.

* + PP0:

Passed the first long green pipe, passed the first pit failed at the second pit.

**Dealing with obstacles:**

* + DQN:

Dealing with pipes and enemies was pretty good, we could see that when the agent encounters a pipe, he knows that he should jump, but it had quite hard time dealing with the planes going out of the pipe that wasn’t exist in the training environment (Super Mario Bro 1).

* + PP0:

Dealing with enemies and pipes was pretty good, did a good job on jumping near enemies and pipes and even jumped above the plants that go out from the pipes even though he is not familiar with them.

The model knows that when it encounters a pit it should jump over it, so it managed learn that.

The overall performance of the model was very good.

**Highest score:**

* + DQN:

1700 points.

* + PP0:

1600 points.

**Rare staff findings:**

* + DQN:

Found 1 rare mushroom (See second 0:46 In the video DQN\_2M\_SUPERMARIO2).

* + PP0:

Found 1 rare mushroom and 1 rare star.

**Conclusions:**

PPO performance was better in the new environment, it manages to understand that he needs to jump above the plants, it manages to pass more distance on the x-axis, which is the main target reward function, so it made a better job in achieving the target.

**Final Conclusions:**

**Performance:**

Regarding the overall comparison, PPO performance are definitely better than DQN performance.

DQN is more stable in its performance, it handles pipes and enemies very well but failed miserably passing the pit.

On the other hand, PPO isn’t stable in it’s performance, most of the times it passe pipes and enemies and sometimes it can make stupid mistakes, but it reached level 2 of the game and passed things in the game that the DQN model never reached to. It even had better results while playing Super Mario Bros 2 (unfamiliar environment), thus it has better results overall and it achieved better convergence.

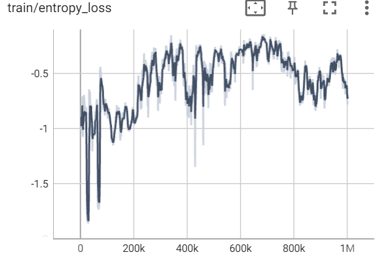
**Exploration rate:**

The tests prove that PPO found more rare things in the game, so we assume it has better exploration of the environment that arises from the stochasticity of the policy to explore the environment, PPO choses the actions based on the states by a Bell distribution, meaning that even if it has found a good action for a certain state it will use it most of the times it encounters said state but sometimes it will choose other actions with lower odds of occurrence for the purpose of exploration.

Even in the graphs we can see that:

Chart, line chart

Description automatically generatedDQN PPO

****

We think that the PPO performance came on the strong hand because it is an on-policy algorithm that generates updates using the current policy as the agent explores the environment, which lead to a better and faster convergence, On the other hand the DQN is an off-policy algorithmthat uses a memory buffer to store and reuse past experience so it’s learning is a bit slower.

One more important thing to add is that DQN memory buffer we used for training had relatively small size so we think a smaller buffer size may result in faster training but could also limit the ability of the algorithm to learn from long-term dependencies in the environment, which can explain the fact that DQN 500,000 timesteps model results were better than the PPO 500,000 timesteps model results (See regular test 2) but in the end PPO has converged to a better model and achieved better results.

The last conclusion is that more research needs to be done in order to determine solid facts, both models should be trained a lot more and achieve more stability.

**How our solution is different from past solutions:**

In the past people trained AI models by letting the AI watch a person play the game (supervised learning), which can result a good model with good performance, but can also limit the model performance, the model can’t get better than the human than he learns from.

So to overcome this, the reinforcement learning approach was invented which lets the model learn by interacting with the environment with series of trials and error which he learns from and develop on.

The model creates its own labeled data (like in supervised learning) from interacting with the environment and getting a rewards that can be positive and negative which lets him know if the action is good for a certain state not (of course it’s not that easy ,the model has to face the “credit assignment problem” which lets the model know which actions in an episode were good and bad and which actions actually impacted the reward it sees, but we won’t go into that).

By doing so, it enables the evolution of models which are far better than humans, it enables the model to explore and discover things humans didn’t know about before (for example find a way to break the game somehow) and it enables a self-learning model that is responsible for it’s own learning that a person should only ran and can leave it to be.

**Personal insights from the project:**

**Lior:**

* + The whole project was very fascinating, I didn’t think I can achieve so much knowledge about RL learning in the short amount of time given us for completing the project, I find this field very fascinating and before the project I thought to take the Devops route but after the project I’m definitely going to the AI route.
  + AI projects requires a lot of space and time, and the learning quality can vary by the hyperparameters provided to the model.
  + I gained general knowledge about RL algorithms, how they work, how the model perceives the environment (state, action, reward) but I didn’t have time to go deeper in order to understand the math behind the algorithms, how to set the actual environment for the model to learn etc..., which is knowledge I find very interesting and hope to achieve it in the future.

**Yair:**

Our project dealt with a Mario game, which from my childhood memory was a fun and fascinating game, and as time went by in developing the project, I realized how complex a simple and friendly game can be with so many things that can be studied and analyzed.

After the project, I learned a lot about artificial intelligence about the distinctive styles to approach problems, the analysis of the problem, and the way to the solution.

I marked for myself during the project three crucial points I had enrichment in during the project.

1. At first, I didn't think it would happen to me, but I fell in love with the algorithms, I saw myself sitting after running 50,000 runs and 500,000 runs and was dying to know how the algorithms dealt with the problem, every time it failed to try to understand from the data I have how much it progressed and where it is Failing and what he does to improve it, it reminded me of a fan watching his football team.

In the academic field, I feel that as time went by, I found more and more interest in the profit function, which is the measure for the algorithms, how to move forward, the understanding that it is crucial, if not the biggest, what is meant by profit, progress? Score? are you alive or dead and so on.

1. The simplicity with which she understood works, before work I always thought about overly complicated things, full of incomprehensible calculations that only math doctors could analyze.

And I confessed that it was not, it is a complex thing, but applicable and possible, very logical!

The combination of analysis and analysis of the data after a certain run and the way to access them gave me a lot of motivation for a deep understanding of the field.

1. The last insight I derived from the project is that this world we call artificial intelligence is still in such a young and fresh place, the game we built was complex and not simple, but it is overall an incredibly old computer game, and it was complex and very educational!

And yet the algorithms we used can answer much more extensive things, and this world of intelligence can provide a solution to the problems of all humanity and make life much easier**.**

**Bibliography:**

* + **Gym Super Mario Bros:**

<https://pypi.org/project/gym-super-mario-bros/>

* + **The two algorithms documentation:**

PPO - <https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>

DQN - <https://stable-baselines3.readthedocs.io/en/master/modules/dqn.htm>

* + **How does the algorithms work:**

PPO - <https://arxiv.org/pdf/1707.06347.pdf>?

DQN 2013 - <https://arxiv.org/pdf/1312.5602.pdf>

DQN 2015 - <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>

* + **Environment setup code:**

<https://github.com/nicknochnack/MarioRL/blob/main/Mario%20Tutorial.ipynb>