Super Mario Bros

This assignment involves building an AI agent that can play Super Mario Bros using the Stable Baselines library. We will preprocess the game environment and train the agent using the Proximal Policy Optimization (PPO) algorithm. Once trained, we will test the agent's ability to play the game and visualize its progress. The ultimate goal is to build an agent that can navigate through the game levels as efficiently as possible.

I've used the following tutorial to do this assignment: https://www.youtube.com/watch?v=2eeYqJ0uBKE.

1. Setup Mario

To start with, we need to set up the Mario environment for our reinforcement learning task. This requires us to install the necessary libraries and create the Mario environment using the gym_super_mario_bros library. Once the environment is created, we will simplify the controls and convert the images to grayscale. Converting the images to grayscale will make it easier to process them using our neural network. Next, we will wrap the environment in a dummy environment, which will make it compatible with various reinforcement learning algorithms. Finally, we will stack the frames of the environment to provide more context for our neural network. This will allow the network to learn from the sequential patterns of the environment, which can be useful for solving more complex tasks.

In [ ]:
```
# 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 Frame Stacker Wrapper and GrayScaling Wrapper
from gym.wrappers import  GrayScaleObservation
# Import Vectorization Wrappers
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv
# Import Matplotlib to show the impact of frame stacking
from matplotlib import pyplot as plt


# 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
```

In [ ]:
```
# 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')
```

In [ ]:
```
SIMPLE_MOVEMENT
```

Out[ ]:
```
[['NOOP'],
 ['right'],
 ['right', 'A'],
 ['right', 'B'],
 ['right', 'A', 'B'],
 ['A'],
 ['left']]
```

In [ ]:
```
# Create a flag, restart or not
done = True
# # Loop through each frame in the game
# for step in range(100000):
#     # Start the game to begin with
#     if done:
#         # Start the game
#         env.reset()
#     # Do random actions
#     state, reward, done, info = env.step(env.action_space.sample())
#     # Show the game on the screen
#     env.render()
# # Close the game
# env.close()
```

In [ ]:
```
# Close the game
env.close()
```

The current status of the code is that the environment has been successfully set up, but when the agent interacts with the environment, it is not able to successfully overcome the first obstacle in the game.

2. Preprocess Environment

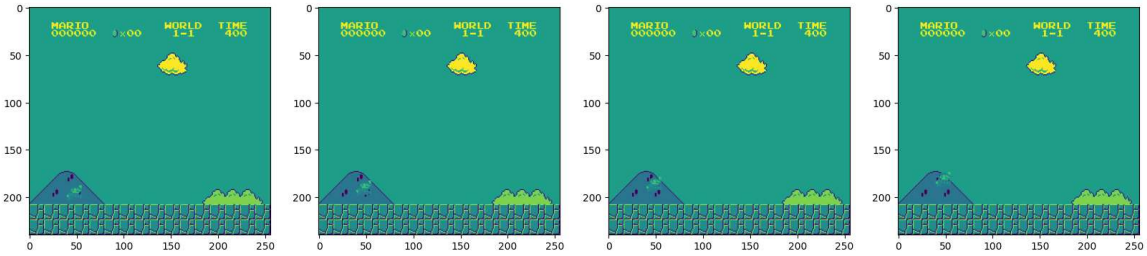
To address the problem written in the step before, we need to preprocess the environment by applying certain transformations to the observations that the agent receives from the environment. This can include techniques such as downsampling the observations, converting them to grayscale, and stacking multiple frames together to give the agent a sense of motion.

In [ ]:
```
# Create the base environment
env = gym_super_mario_bros.make('SuperMarioBros-v0')

# Simplify the controls
env = JoypadSpace(env, SIMPLE_MOVEMENT)
```

```
# Grayscale
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')

# Reset the environment
state = env.reset()
```

In [ ]:
```
#Let mario jump
state, reward, done, info = env.step([5])
```

In [ ]:
```
plt.figure(figsize=(20,16))
for idx in range(state.shape[3]):
    plt.subplot(1,4,idx+1)
    plt.imshow(state[0][:,:,idx])
plt.show()
```



The plot of the preprocessed frames shows Mario jumping higher in each subsequent image, indicating that the agent is learning and making appropriate actions to maximize its reward in the game. These frames provide insight into the agent's behavior and ability to navigate the game.

3. Train the RL Model

Once we have set up the environment, the next step is to train a reinforcement learning model. In this step, we will use the Stable Baselines3 library to train a PPO (Proximal Policy Optimization) algorithm. The PPO algorithm is a widely used algorithm for reinforcement learning and is known for its stable performance. During the training process, the RL model will learn how to navigate through the Super Mario Bros. game world by trial and error. The RL model will receive a reward for achieving certain goals (such as collecting coins, completing levels, etc.) and will adjust its behavior to maximize the rewards it receives. The training process can take several hours, depending on the complexity of the game and the size of the neural network used. Once the RL model is trained, we can use it to play the game automatically and see how well it performs.

In [ ]:
```
# 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
```

In [ ]:
```
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(sel
            self.model.save(model_path)

        return True
```

In [ ]:
```
# Location to save the training steps
CHECKPOINT_DIR = './train/'
LOG_DIR = './logs/'
```

In [ ]:
```
# Setup model saving callback
callback = TrainAndLoggingCallback(check_freq=10000, save_path=CHECKPOINT_DIR)
```

In [ ]:
```
# This is the AI model started
model = PPO('CnnPolicy', env, verbose=1, tensorboard_log=LOG_DIR, learning_rate=
            n_steps=512)
```

```
Using cpu device
Wrapping the env in a VecTransposeImage.
```

In [ ]:
```
# # Train the AI model, this is where the AI model starts to learn
# model.learn(total_timesteps=1000000, callback=callback)
```

After spending approximately 6 hours to reach step 60000, I have decided to halt the training process. The next step is to evaluate if the trained model is capable of completing the level.

4. Test it out

At this stage, I have the option to select one of the saved models and observe its performance.

In [ ]:
```
# 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 Frame Stacker Wrapper and GrayScaling Wrapper
from gym.wrappers import  GrayScaleObservation
# Import Vectorization Wrappers
from stable_baselines3.common.vec_env import VecFrameStack, DummyVecEnv
# Import Matplotlib to show the impact of frame stacking
from matplotlib import pyplot as plt


# 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
```

In [ ]:
```
# Create the base environment
env = gym_super_mario_bros.make('SuperMarioBros-v0')

# Simplify the controls
env = JoypadSpace(env, SIMPLE_MOVEMENT)

# Grayscale
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')

# Reset the environment
state = env.reset()
```

In [ ]:
```
# Load model
model = PPO.load('./train/best_model_600000')
```

In [ ]:
```
state = env.reset()
```

In [ ]:
```
# Start the game
state = env.reset()
# Loop through the game
while True:

    action, _ = model.predict(state)
    state, reward, done, info = env.step(action)
    env.render()
```

```
c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\pyglet\image\codecs
\wic.py:289: UserWarning: [WinError -2147417850] Kan threadmodus niet wijzigen
nadat deze is ingesteld
  warnings.warn(str(err))
c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\gym_super_mario_bros
\smb_env.py:148: RuntimeWarning: overflow encountered in ubyte_scalars
  return (self.ram[0x86] - self.ram[0x071c]) % 256
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[8], line 6
      3 # Loop through the game
      4 while True:
----> 6     action, _ = model.predict(state)
      7     state, reward, done, info = env.step(action)
      8     env.render()

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\base_class.py:539, in BaseAlgorithm.predict(self, observation, state,
episode_start, deterministic)
    519 def predict(
    520     self,
    521     observation: Union[np.ndarray, Dict[str, np.ndarray]],
   (...)
    524     deterministic: bool = False,
    525 ) -> Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]:
    526     """
    527     Get the policy action from an observation (and optional hidden stat
e).
    528     Includes sugar-coating to handle different observations (e.g. norma
lizing images).
   (...)
    537         (used in recurrent policies)
    538     """
--> 539     return self.policy.predict(observation, state, episode_start, deter
ministic)

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\policies.py:346, in BasePolicy.predict(self, observation, state, epis
ode_start, deterministic)
    343 observation, vectorized_env = self.obs_to_tensor(observation)
    345 with th.no_grad():
--> 346     actions = self._predict(observation, deterministic=deterministic)
    347 # Convert to numpy, and reshape to the original action shape
    348 actions = actions.cpu().numpy().reshape((-1, *self.action_space.shape))

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\policies.py:676, in ActorCriticPolicy._predict(self, observation, det
erministic)
    668 def _predict(self, observation: th.Tensor, deterministic: bool = False)
-> th.Tensor:
    669     """
    670     Get the action according to the policy for a given observation.
    671
   (...)
    674     :return: Taken action according to the policy
    675     """
--> 676     return self.get_distribution(observation).get_actions(deterministic
=deterministic)

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\policies.py:709, in ActorCriticPolicy.get_distribution(self, obs)
    702 def get_distribution(self, obs: th.Tensor) -> Distribution:
    703     """
    704     Get the current policy distribution given the observations.
    705
    706     :param obs:
    707     :return: the action distribution.
```

```
    708     """
--> 709     features = super().extract_features(obs, self.pi_features_extracto
r)
    710     latent_pi = self.mlp_extractor.forward_actor(features)
    711     return self._get_action_dist_from_latent(latent_pi)

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\policies.py:129, in BaseModel.extract_features(self, obs, features_ex
tractor)
    121 def extract_features(self, obs: th.Tensor, features_extractor: BaseFeat
uresExtractor) -> th.Tensor:
    122     """
    123     Preprocess the observation if needed and extract features.
    124
   (...)
    127      :return: The extracted features
    128     """
--> 129     preprocessed_obs = preprocess_obs(obs, self.observation_space, norm
alize_images=self.normalize_images)
    130     return features_extractor(preprocessed_obs)

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\preprocessing.py:109, in preprocess_obs(obs, observation_space, norma
lize_images)
     97 """
     98 Preprocess observation to be to a neural network.
     99 For images, it normalizes the values by dividing them by 255 (to have v
alues in [0, 1])
   (...)
    106 :return:
    107 """
    108 if isinstance(observation_space, spaces.Box):
--> 109     if normalize_images and is_image_space(observation_space):
    110         return obs.float() / 255.0
    111     return obs.float()

File c:\Users\Warmtebron\anaconda3\envs\rnEnv\lib\site-packages\stable_baseline
s3\common\preprocessing.py:55, in is_image_space(observation_space, check_chann
els, normalized_image)
     52     return False
     54 # Check the value range
---> 55 incorrect_bounds = np.any(observation_space.low != 0) or np.any(observa
tion_space.high != 255)
     56 if check_bounds and incorrect_bounds:
     57     return False

File <__array_function__ internals>:177, in any(*args, **kwargs)

KeyboardInterrupt:
```

In [ ]:
```
env.close()
```

Conclusion

The latest saved model in the training process shows significant improvement compared to the first model. While the first model struggled to jump over obstacles, the latest model shows a much better performance and is able to make progress in the game.

However, the latest model is still not able to pass the level successfully. This indicates that the model requires further training to achieve a higher level of accuracy and successfully complete the game.