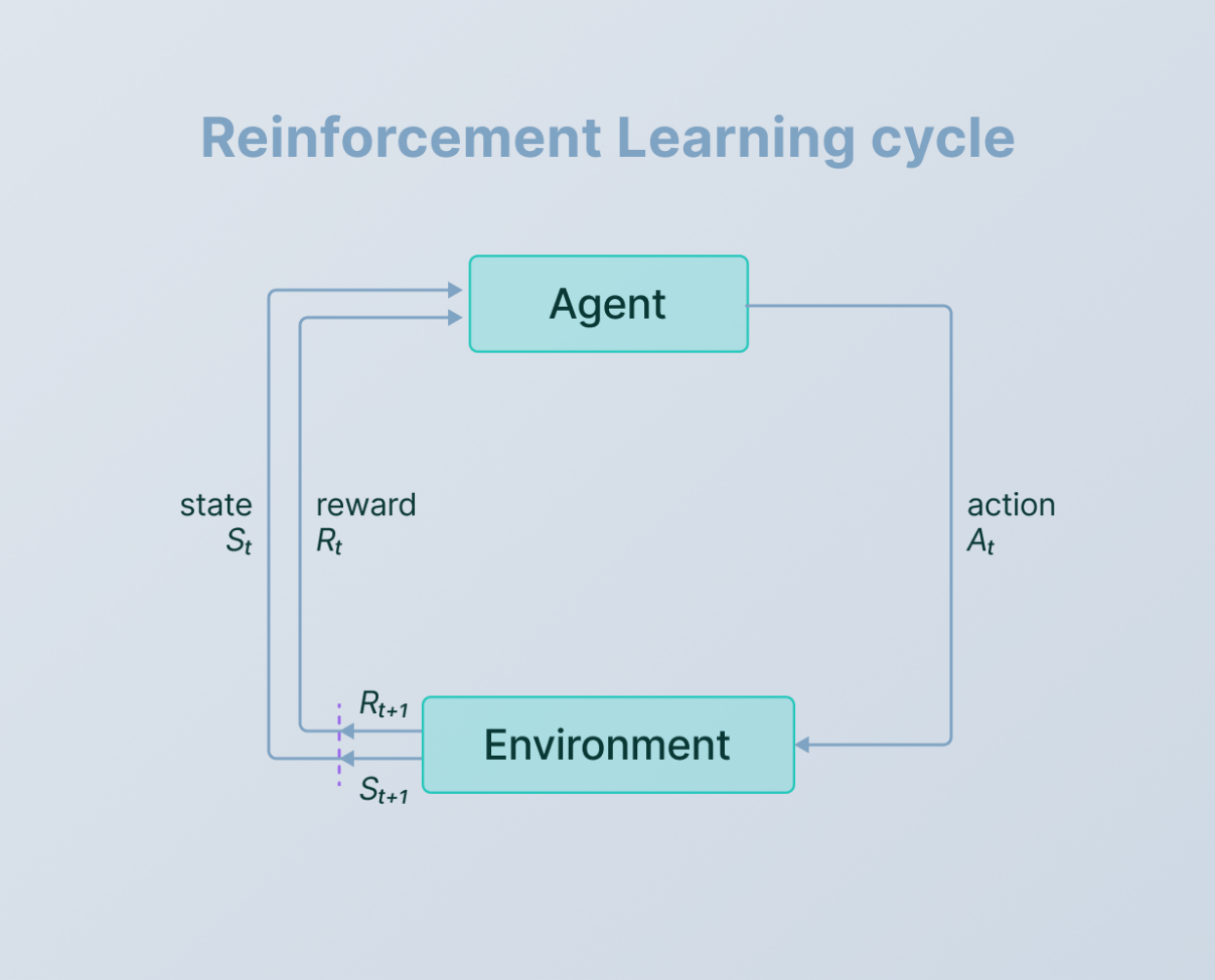
Reinforcement Learning For Playing Super Mario Bros

Reinforcement Learning is combined by the following factors: Environment, States, Agent, Set of Possible Actions, Reward.

In our case of playing the game Super Mario Bros:

- `Environment`: The Game itself
- `State`: the pixels input of the current frame
- `Agent`: Mario himself
- `Action`: Moving right, left, jumping, etc.
- `Reward`: Progressing the game without dying or losing lives.

After the Agent takes an action, the environment moves to the next state **S'**.



A key difficulty in creating algorithms for playing games is determining the optimal action to maximize the reward. Essentially, these algorithms can be broadly classified into two main categories:

- **Value Based**: (e.g. Q-learning, Deep Q-Network, Double Deep Q-Network, etc.,). These algorithms `estimate the value of being in a state` or `the expected cumulative reward` (Q-value) of taking a specific action in a state. The policy that drives the agent's behavior is derived from these values.
- **Policy-Based**: (e.g. PPO, REINFORCE, A2C, etc,.). These algorithms `directly` aim to `optimize the policy without calculating a value function`. The `policy is parameterized`, and the learning process refines these parameters.

Expanding on these classifications, there exist `hybrid` approaches referred to as `Actor-Critic algorithms`. In these algorithms, the `actor suggests actions according to a policy`, while the `critic evaluates the worth of states` or `state-action pairs` to inform the actor's decision-making. The fundamental idea behind actor-critic methods is the concurrent optimization of both the policy (handled by the actor) and the value function (assessed by the critic).

PPO (Proximal Policy Optimization) is a `policy-based` algorithm, but it can be viewed as a `hybrid approach` that incorporates some elements of actor-critic methods. PPO was introduced to address some of the challenges and limitations of traditional policy gradient methods, and it stands for both "Proximal Policy Optimization" and "Policy Policy Optimization," reflecting its focus on optimizing policies.

Within the PPO framework, the actor generates a policy distributed over actions, while the critic evaluates the value of states. Subsequently, the PPO algorithm refines the policy by employing a surrogate objective, leveraging the critic's value assessments. This process ensures that the updated policy remains close to the original one, preventing significant deviations.

1. Environment Setting

```
import gym
from nes_py.wrappers import JoypadSpace

env = gym.make('SuperMarioBros-v0', apply_api_compatibility = True, render_mode = "human")
env = JoypadSpace(env, [["right"], ["right", "A"]])
```

In this setup, we've defined our action space to include only two inputs: **right** and **right and A** (right and jump). This choice simplifies our training process. While it reduces the range of possible actions, it remains sufficient for competing most levels, striking a balance between complexity and performance.

2. Reward Structure

The goal in RL is to select actions that maximizes rewards. The reward function built into the `gym-super-mario-bros` library is designed with the objective that the agent should `move as far to the right` as possible, as `quick` as possible and `without dying`.

The reward is composed of three separate components:

1. **Instantaneous Velocity**: Represents the difference in the agent's x-values between two consecutive states:

$$v = x_1 - x_0$$

where:

- x_0 is the x-position before the step
- x_1 is the x-position after the step.

The possible conditions are:

- $v > 0 \implies$ Moving right
- $v < 0 \implies$ Moving left
- $v = 0 \implies$ Stay still

2. **Clock Penalty**: Represents the differences in the game clock between frames. This penalty `prevents the agent from standing still`.

   $c = c_0 - c_1$

   where:

   - c_0 is the clock reading before the step
   - c_1 is the clock reading after the step.

   Conditions:

   - $c = 0 \implies$ No clock tick
   - $c < 0 \implies$ Clock tick

3. **Death Penalty**: Penalizes the agent for dying in a state, encouraging the agent to avoid death.

   $d = 0 \implies$ Alive

   $d = -15 \implies$ Dead

The overall Reward is then written as:

$r = v + c + d$

which is clipped to the range [-15, 15].

3. PPO

PPO, or Proximal Policy Optimization, is a Reinforcement Learning algorithm designed for training policies in environments where an agent makes decisions to maximize cumulative rewards. It belongs to the family of policy gradient methods and is known for its stability and efficiency in optimizing complex policies.

Structure:

1. **Actor-Critic Framework:**
   - `PPO` follows an `actor-critic` structure, where the `actor` is responsible for `generating a policy` (Probability distribution over actions), and the `critic` `estimates the value of states` or state-action pairs.
2. **Objective Function:**
   - `PPO` employs a `surrogate objective function` that combines elements of policy improvement and value estimation. This objective function is `designed to prevent large policy changes` and `stabilize the learning process`.
3. **Policy Optimization:**
   - The `actor` proposes actions based on a policy, and the `algorithm` seeks to `optimize this policy`. PPO introduces a clipping mechanism in the objective function to ensure that the updated policy does not deviate significantly from the previous one.
4. **Value Function Estimation:**

- The `critic` `estimates the value of states`, providing `feedback` on the `expected cumulative rewards` associated with being in particular states or taking specific actions. This value estimation is crucial for guiding the actor's policy updates.

5. **Clipping Mechanism:**
   - One distinctive feature of PPO is the use of a `clipping mechanism` in the objective function. This mechanism `constrains the policy update to a certain range`, preventing overly aggressive changes that could destabilize training.
6. **Iteration:**
   - PPO typically involves multiple iterations or epochs of collecting experiences in the environment, updating the policy and value function estimates, and iterating to improve the policy further.

Training Process:

- PPO uses a form of stochastic gradient ascent to maximize the surrogate objective function. The algorithm collects samples from the environment, computes the objective function, and then performs gradient-based updates to improve the policy.

Advantages:

- PPO is known for its stability and sample efficiency, making it suitable for a wide range of Reinforcement Learning tasks. The use of a clipping mechanism contributes to preventing large policy changes, which can be beneficial for training robust policies.

PPO Algorithm Steps:

- **Sampling:**
  - The agent interacts with the environment, collecting experiences (states, actions, rewards, etc.).
  - For each time step, the actor's current policy is used to sample actions.
  - The advantages and GAE are calculated based on the rewards and values obtained from the critic.
- **Training:**
  - The collected experiences are used to update the actor and critic models.
  - The PPO loss function is computed, including the policy loss, value function loss, and entropy loss.
  - The loss is minimized using back propagation and optimization.
- **Checkpointing:**
  - Periodically, the model's weights are saved as checkpoints during training.
  - This allows the training process to be resumed from a specific episode.

About GAE and Advantages role in PPO:

1. **Advantages:**

- **Definition:** In the context of reinforcement learning, advantages represent the `difference` `between` the `observed returns` (rewards) and the `expected returns` estimated by the value function.
- **Purpose:** Advantages `indicate how much better or worse an agent's actual experience was` `compared to what the policy expected`. Positive advantages indicate better-than-expected outcomes, while negative advantages indicate worse-than-expected outcomes.

2. **Generalized Advantage Estimation (GAE):**
   - **Definition:** GAE is a method to `estimate advantages` that incorporates not only the immediate rewards but also the expected future rewards.
   - **Purpose:** GAE provides a more nuanced measure of advantage by taking into account the temporal structure of the environment. It considers the cumulative effect of rewards over time, allowing for more informative advantages.
3. **Role in PPO:**
   - **Policy Update:** The PPO algorithm updates the policy based on the advantages and GAE. The objective is to `encourage` actions that lead to `higher-than-expected returns` and `discourage` actions that lead to `lower-than-expected returns`.
   - **Clipping Mechanism:** PPO introduces a `clipping mechanism` in the objective function to `prevent overly large policy updates`. The clipped objective is a compromise between the surrogate objective and a conservative objective, which helps stabilize training.

4. Neural Networks

We can use the `nn` class form the `torch` library to create our actor and critic networks:

```
# This type of architecture is common in environments where the state space is
# represented as images (e.g., Games environments in OpenAI Gym).

import torch
import torch.nn as nn

# Assuming 'env' has an attribute 'action_space.n' for the number of actions

# Define the actor network
actor = nn.Sequential(
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
    nn.ReLU(),
    nn.Flatten(),  # Flatten the output for the fully connected layers
    nn.Linear(3136, 512),
    nn.ReLU(),
    nn.Linear(512, env.action_space.n)  # Output layer for action probabilities
)

# Define the critic network
critic = nn.Sequential(
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4),
    nn.ReLU(),
```

```
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
    nn.ReLU(),
    nn.Flatten(),  # Flatten the output for the fully connected layers
    nn.Linear(3136, 512),
    nn.ReLU(),
    nn.Linear(512, 1)  # Output layer for state value estimation
)

# Example usage (assuming input tensor 'state' with shape (batch_size, 4, height, width))
action_probs = actor(state)
state_value = critic(state)

# Print the network architectures
print("Actor Network:\n", actor)
print("\nCritic Network:\n", critic)
```

We can see that both the actor and critic networks share a common structure, comprising three convolutional layers followed by two linear layers.

- **Convolutional Layers**: Foundational to Convolutional Neural Networks (CNNs), these layers are `optimized for processing grid-like data structures`, such as images. Operating through convolution, a mathematical operation, these layers enable the network to recognize and learn spatial hierarchies of features. Their strength lies in discerning patterns in the spatial or temporal domain, making them well-suited for tasks like image and video recognition.
- **Linear Layers**: Also referred as fully connected layers, linear layers are fundamental components present in conventional neural networks. They play a pivotal role in `generating output` predictions `based on the features acquired from preceding layers`.

  The network is specifically `designed to handle inputs with four channels`, representing four stacked grayscale frames from the game (further explained in the preprocessing section). The `actor network`'s output channels correspond to the game's action space, allowing the `modeling of a categorical distribution over possible actions`. Conversely, the `critic network` produces a singular output channel, providing an `estimate of the state’s value`.

5. Training Process

The training process can be broken into three fundamental steps:

Iterate:

- **Sample**
- **Evaluate**
- **Optimize**

The initial stage of the training process involves sampling, where the model interacts with the game, accumulating experience that is later evaluated and used to optimize performance for improvement.

3.1 Pre-processing

Training a Reinforcement Learning model can be computationally intensive, but there are ways to enhance efficiency through adjustments to the data sampling process. These adjustments, collectively known as `pre-processing`, involve providing the model with `minimal input`, `reducing computational calculations`, and `expediting the learning process`. The pre-processing steps include reducing the action space, resizing frames, and converting frames to grayscale.

- **Reducing Action Space:** This entails limiting the number of possible actions the agent can take. For instance, in a traditional Super Mario Bros setting, action like moving left, right, up, and down, as well as buttons A (jump) and B (use items), can be streamlined to a more focused set, such as [”right”] and [”A”, “right”]. This reduction in the action space accelerates the training process.

```
env = JoypadSpace(env, [["right"], ["right", "A"]])
```

- **Resizing Observation Space**: Changing the observation space dimensions minimizes the input that the neural network needs to process. Resizing can be achieved using TorchVision’s transforms.

```
class ResizeObservation(gym.ObservationWrapper):
    def __init__(self, env, shape):
        super().__init__(env)
        # Set the desired shape for observation resizing
        self.shape = (shape, shape)
        # Adjust the observation space shape to accommodate the resized frames
        obs_shape = self.shape + self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def observation(self, observation):
        # Define a sequence of transformations using TorchVision
        transformations = transforms.Compose([
            transforms.Resize(self.shape),
            transforms.Normalize(0, 255)  # Normalize pixel values between 0 and 255
        ])
        # Apply the defined transformations and remove singleton dimension
        return transformations(observation).squeeze(0)
```

- **Converting to Grayscale**: Converting images to grayscale involves using a single color channel instead of three (RGB). This reduces the amount of data for interpretation. This task can be done with TorchVisions `Grayscale()` function.

```
class GrayScaleObservation(gym.ObservationWrapper):
  def __init__(self, env):
    super().__init__(env)
    # Set the observation space to grayscale, reducing data dimensions
    self.observation_space = Box(low = 0, high = 255, shape = self.observation_space.shape[:2], dtype = np.uint8)

  def observation(self, observation):
    # Apply TorchVision's Grayscale transformation
    transform = transforms.Grayscale()

    # Transpose the observation to match TorchVision's expected format and convert to a PyTorch tensor
```

```
    return transform(torch.tensor(np.transpose(observation, (2, 0, 1)).copy(), dtype = torch.float))
```

- **Frame Skipping**: This is the final method of optimizing performance. In numerous games or simulations, consecutive frames can exhibit almost identical content, making it computationally inefficient to run an agent's decision process on every single frame. To address this, we can skip a certain number of frames, presenting the agent with every nth frame. When the agent takes an action, instead of applying the action to the environment once, the action is executed for skip consecutive frames. The rewards from all these frames are aggregated, and only the last observation (frame) is returned to the agent. If the episode terminates before all skips are completed, it concludes prematurely.

```
class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        super().__init__(env)
        # Set the frame skip parameter
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = False
        trunc = False
        # Execute the action for skip consecutive frames
        for _ in range(self._skip):
            obs, reward, done, trunc, info = self.env.step(action)
            total_reward += reward
            # Break if the episode terminates
            if done:
                break
        return obs, total_reward, done, trunc, info
```

- **Frame Stacking**: The last step in the pre-processing pipeline. This involves stacking frames and providing them to the network, aiding the agent in understanding movement within the frame. As noticed earlier, our `networks have four input channels`, where `each channel corresponds to a frame in a group of four`. With these four elements implemented - reducing action space, resizing frames, converting frames to grayscale, and frame stacking - we can now more efficiently sample and train our model.

```
class FrameStack(gym.Wrapper):
  def __init__(self, env, num_stack):
    super().__init__(env)

    # Number of frames to stack
    self.num_stack = num_stack

    # Circular buffer to store the frames
    self.frames = deque(maxlen = num_stack)

    # Adjust the observation space to account for stacked frames
    low = np.repeat(env.observation_space.low, num_stack, axis = -1)
    high = np.repeat(env.observation_space.high, num_stack, axis = -1)
    self.observation_space = Box(low = low, high = high, dtype = env.observation_space.dtype)

  def reset(self):
```

```
    # Reset the environment and initializethe frame buffer
    observation = self.env.reset()
    self.frames.extend([observation] * self.num_stack)
    return np.array(self.frames).transpose(1, 2, 0)

  def step(self, action):
    # Take a step in the environment
    observation, reward, done, info = self.env.step(action)

    # Add the current observation to the frame buffer
    self.frames.append(observation)

    # Return the stacked frames as the modified observation
    return np.array(self.frames).transpose(1, 2, 0), reawrd, done, info
```

```
env = JoypadSpace(env, [["right"], ["right", "A"]])
env = SkipFrame(env, skip = 4)
env = GrayScaleObservation(env)
env = ResizeObservation(env, shape = 84)
env = FrameStack(env, num_stack = 4)
```

3.2 Sampling

Now that we've completed the data pre-processing, we can initiate the `sampling` phase. This involves `employing the model to play` the game and `acquire knowledge` from experience. Initially, we define arrays to hold the results from our training, continuously appending to these array while the game is in progress.

Pseudocode Implementation of Sampling:

```
Input: batch_size, policy_old, env
Initialize arrays: rewards, actions, done, obs, log_pis, values of size batch_size

FOR i from 0 to batch_size - 1 DO
  Get current observation obs[i]
  Calculate action probability distribution p1 and value estimate v using policy_old from obs[i]
  Store v in values[i]
  Sample action a from pi
  Store a in actions[i]
  Store log probability of a in log_pis[i]
  Execute action a in the environment to get new obs, reward, and done status
  Store the reward in rewards[i]
  Render the environment
  IF episode is done THEN
    Reset the episode and environment
  END IF
END FOR
```

Python Implementation:

```
import torch
import torch.nn.functional as F
import torch.optim as optim

# Assume the existence of a neural network class representing the policy
class PolicyNetwork(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(PolicyNetwork, self).__init__()
```

```
        # Define your neural network layers here

    def forward(self, x):
        # Define the forward pass of the neural network
        return pi, v  # Assuming pi is the action distribution and v is the value estimate

# Function to sample the environment
def sample_environment(batch_size, policy_old, env):
    # Initialize arrays
    rewards = torch.zeros(batch_size)
    actions = torch.zeros(batch_size, dtype=torch.long)
    done = torch.zeros(batch_size, dtype=torch.bool)
    obs = [None] * batch_size
    log_pis = torch.zeros(batch_size)
    values = torch.zeros(batch_size)

    for i in range(batch_size):
        # Get current observation
        obs[i] = env.get_observation()

        # Calculate action probability distribution pi and value estimate v using policy_old
        pi, v = policy_old(obs[i])

        # Store value estimate
        values[i] = v

        # Sample action a from pi
        a = torch.multinomial(F.softmax(pi, dim=-1), 1).item()

        # Store action and log probability
        actions[i] = a
        log_pis[i] = F.log_softmax(pi, dim=-1)[a]

        # Execute action in the environment to get new obs, reward, and done status
        obs[i], rewards[i], done[i] = env.step(a)

        # Render the environment (if needed)
        env.render()

        # If episode is done, reset the episode and environment
        if done[i]:
            obs[i] = env.reset()

    return rewards, actions, done, obs, log_pis, values

# Example usage
env = MyEnvironment()  # Replace with your environment class
policy_old = PolicyNetwork(input_size, output_size)  # Replace with your policy network
optimizer = optim.Adam(policy_old.parameters(), lr=0.001)

# Sampling
batch_size = 32
rewards, actions, done, obs, log_pis, values = sample_environment(batch_size, policy_old, env)

# Now use the collected data to calculate loss and update the model using backpropagation and optimization
# (Note: The actual implementation depends on your specific RL algorithm)
```