AI Agent за Super Mario Bros чрез Reinforcement Learning

Документация

От: Александър Македонски

github: https://github.com/alemaki/super_mario_bros_AI

Въведение

Целта на този проект е да се създаде изкуствен интелект (AI) агент, който да може да играе и да преминава нива в класическата игра Super Mario Bros. Това се постига чрез използването на Reinforcement Learning (обучение с подсигуряване), където агентът се учи да изпълнява действия, които максимизират наградата в играта. Проектът е мотивиран от любовта към видеоигри и интереса към изкуствения интелект, като съчетаването на двете области предоставя уникална възможност за изследване и развитие.

Мотивация

Избрах този проект, защото видеоигрите са неразделна част от моя живот, а изкуственият интелект е една от най-вълнуващите и бързо развиващи се области в технологиите и представлява голям мой интерес. Създаването на AI агент, който да играе Super Mario Bros, е предизвикателство, което обединява двете ми страсти. Този проект ми позволява да разбера по-добре как функционират алгоритмите за reinforcement learning и да ги приложа в реална среда.

Алгоритми за сравнение

В проекта ще бъдат сравнени три алгоритма за reinforcement learning:

1) DQN (Deep Q-Network)
   - Базиран на Q-learning, използва невронна мрежа за приближаване на Q-стойности.
   - Използва experience replay и target network за стабилизиране на обучението.
2) A2C (Advantage Actor-Critic)
   - Комбинира подход на actor-critic с изчисляване на advantage функция.
   - Работи в синхронна среда, като актьорът и критикът се обучават едновременно.
3) A3C (Asynchronous Advantage Actor-Critic)
   - Подобен на A2C, но работи асинхронно с множество агенти, които се обучават паралелно.
   - По-ефективен от A2C по отношение на времето за обучение.

Среда за тренировка

За тренировката на AI агента е използвана средата gym-super-mario-bros, която предоставя интерфейс за взаимодействие с играта Super Mario Bros. Средата е интегрирана в Python 3.8 и позволява на агента да извършва действия и да получава обратна връзка под формата на награди и състояния на играта.

Библиотека PyTorch

За изграждането и тренирането на невронните мрежи е използвана библиотеката PyTorch. PyTorch е мощна библиотека за машинно обучение, която предоставя гъвкавост и лекота при създаването на модели. Тя позволява дефинирането на невронни мрежи, оптимизацията на параметрите и изчисляването на градиенти чрез автоматично диференциране.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class DQN(nn.Module):
def __init__(self, input_shape, n_actions: int, use_leaky_relu: bool = False,
channel_multiplier: int = 1):
super(DQN, self).__init__()

self.conv = nn.Sequential(
nn.Conv2d(input_shape[0], 32*channel_multiplier, kernel_size=8, stride=4),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
nn.Conv2d(32*channel_multiplier, 64*channel_multiplier, kernel_size=4, stride=2),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
nn.Conv2d(64*channel_multiplier, 64*channel_multiplier, kernel_size=3, stride=1),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
)
self.fc = nn.Sequential(
nn.Linear(self._feature_size(input_shape), 512*channel_multiplier),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
nn.Linear(512*channel_multiplier, n_actions),
)

def _feature_size(self, shape):
with torch.no_grad():
return self.conv(torch.zeros(1, *shape)).view(1, -1).size(1)

def forward(self, x):
x = self.conv(x)
return self.fc(x.view(x.size(0), -1))
```

Обработка на изображението в проекта

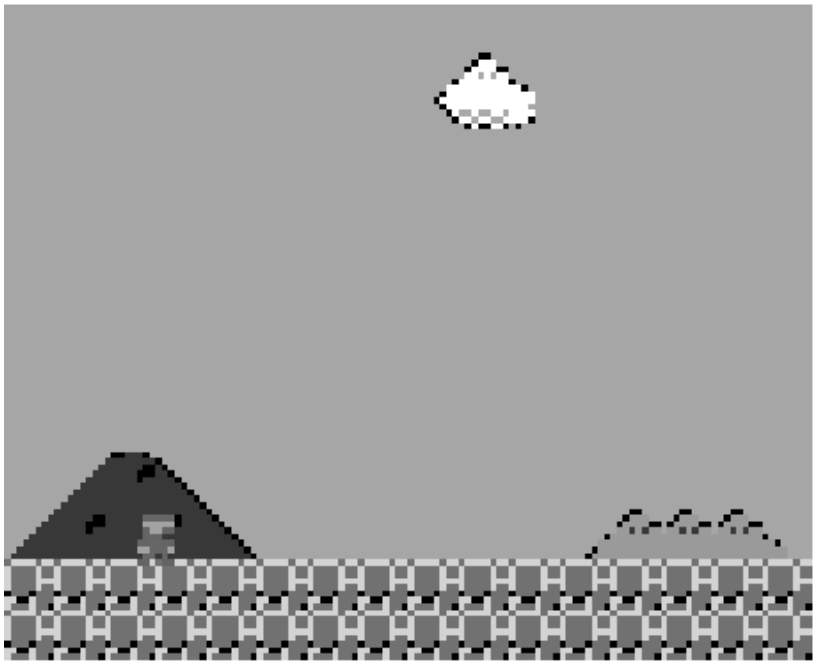
В проекта за създаване на AI агент, който играе Super Mario Bros, обработката на изображението е критична стъпка, тъй като входните данни за невронната мрежа са визуални кадри от играта. Тези кадри трябва да бъдат предварително обработени, за да намалят сложността и да ускорят обучението на модела. Обработката включва няколко ключови стъпки:

1. Преобразуване в черно-бяло (Grayscale)
Играта Super Mario Bros генерира цветни изображения:

Но за да се намали размерността на данните и да се улесни обработката, цветните кадри се преобразуват в черно-бели:

Намаляване на резолюцията (Downsampling)

Оригиналните кадри от играта имат висока резолюция, което ги прави прекалено големи за обработка от невронната мрежа. Затова резолюцията се намалява чрез премахване на ненужни детайли (горната част на снимката с ненужни данни и намаляне на резолюцията):



Документация за основния скрипт на проекта

Този скрипт е основната част от проекта за обучение на AI агент да играе Super Mario Bros чрез reinforcement learning. Той използва Deep Q-Network (DQN) алгоритъм за обучение на агента. Скриптът включва инициализация на средата, конфигурация на хиперпараметри, обучение на модела и записване на резултатите.

Основен цикъл за обучение

Скриптът изпълнява основния цикъл за обучение, който включва:

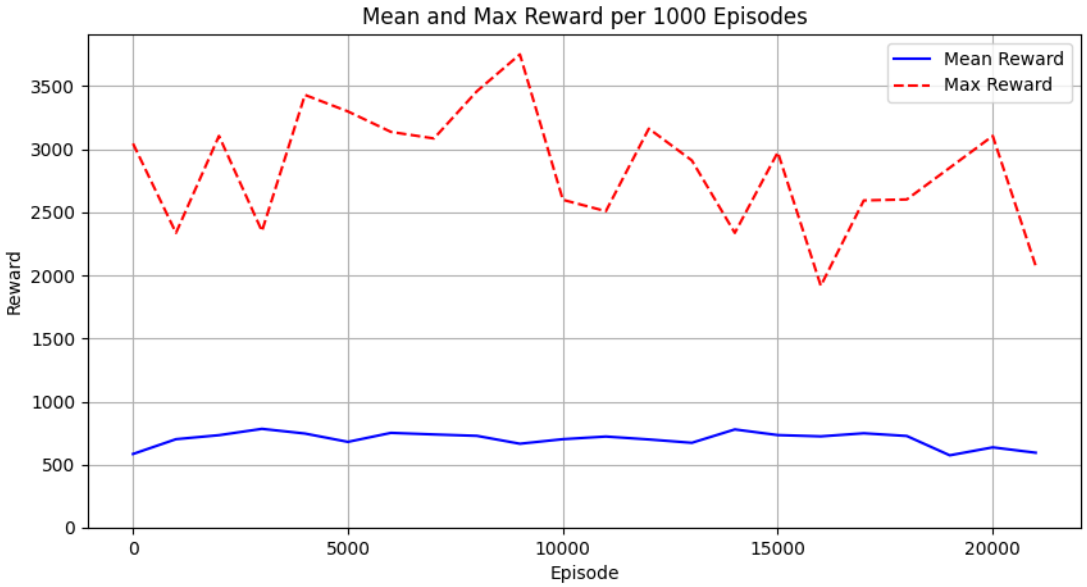
1. Инициализация на епизода:
   - Нулиране на състоянието и подготовка за нов епизод.
   - Инициализация на променливи за награди и време.
2. Изпълнение на стъпки в средата:
   - Избор на действие чрез epsilon-greedy стратегия.
   - Изпълнение на действието и получаване на ново състояние, награда и информация.
   - Запазване на преживяването в буфера.
3. Обучение на модела:
   - Ако буферът е достатъчно голям, моделът се обучава с данни от буфера.
   - Актуализиране на целевата мрежа на определени интервали.
4. Записване на резултатите:
   - Записване на информация за епизода в лог файл.
   - Запазване на модела на определени интервали.

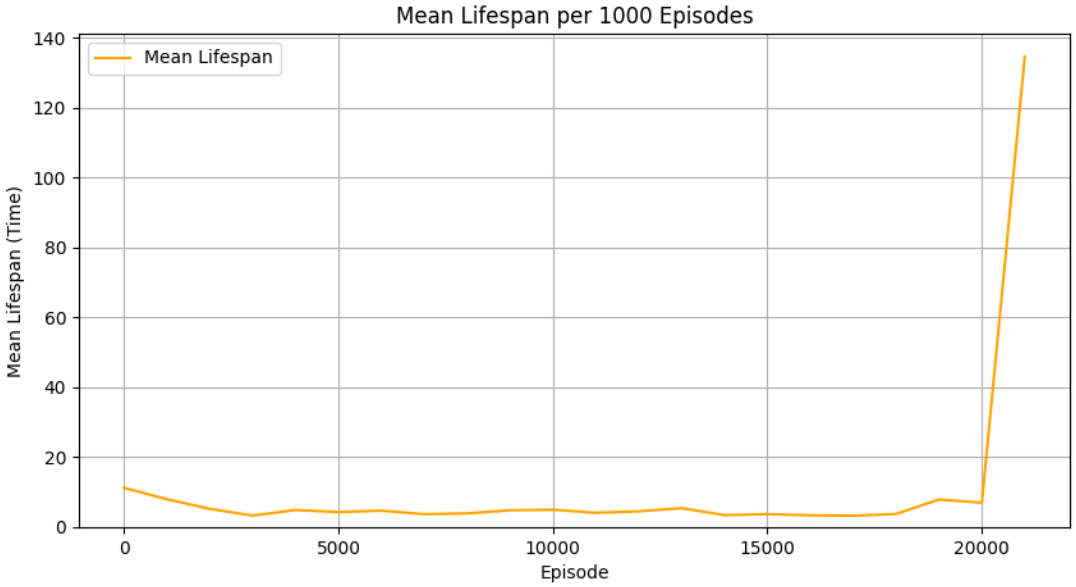
```
policy_net = DQN(input_shape, n_actions, True, CHANNEL_MULTIPLIER).to(device)
target_net = DQN(input_shape, n_actions, True, CHANNEL_MULTIPLIER).to(device)
if START_MODEL_EPISODE != -1:
load_dqn_models(START_MODEL_EPISODE, policy_net, target_net, SAVE_DIR)

target_net.load_state_dict(policy_net.state_dict())
target_net.eval()
optimizer = optim.Adam(policy_net.parameters(), lr=LEARNING_RATE)
memory = deque(maxlen=MEMORY_SIZE)

epsilon = EPSILON_START
epsilon = max(EPSILON_MIN,
epsilon*(EPSILON_DECAY**(START_MODEL_EPISODE*MAX_STEPS/EPSILON_UPDATE)))

target_update_frames_left = TARGET_UPDATE

for episode in range(START_MODEL_EPISODE + 1, EPISODE_STOP + 1):
start_time = time.time()
state = preprocess_smaller_state(env.reset(), device=device)
state = state.unsqueeze(0).repeat(4, 1, 1)

done = False
total_reward = 0
remaining_lives = 2

for frame in range(MAX_STEPS + 1):
#env.render()
if np.random.rand() < epsilon:
action = env.action_space.sample()
Else:
state_tensor = state.unsqueeze(0) # Add batch dimension
action = policy_net(state_tensor).argmax(dim=1).item()

next_state, reward, done, truncated, info = env.step(action)
next_state = preprocess_smaller_state(next_state, device=device)
next_state = torch.cat((state[1:], next_state.unsqueeze(0)), dim=0) # Update frame stack

if remaining_lives > info['life']:
remeaining_lives = info['life']
reward-=50
memory.append((state, action, reward, next_state, done))
state = next_state
total_reward += reward
```

```
# Training
if len(memory) > BATCH_SIZE:
policy_net.train_mario(memory,
target_net,
optimizer,
GAMMA,
BATCH_SIZE)

target_update_frames_left -= 1
if target_update_frames_left <= 0:
target_net.load_state_dict(policy_net.state_dict())
target_update_frames_left = TARGET_UPDATE

if frame % EPSILON_UPDATE == 0:
epsilon = max(EPSILON_MIN, epsilon*EPSILON_DECAY)

if done or truncated:
break

if ONE_LIFE and info['life'] < 2:
break

elapsed_time = time.time() - start_time
print(f"Episode {episode}, Total Reward: {total_reward}, Time Elapsed: {elapsed_time:.2f} seconds, epsilon
{epsilon}")

record_info_for_episode(LOG_FILE_NAME, episode, total_reward, elapsed_time, info)

if episode % EPISODE_SAVE == 0:
save_dqn_model(episode, policy_net, SAVE_DIR)


env.close()
```

Резултати

Резултатите от обучението на DQN агента:





Документация за A2C (Advantage Actor-Critic) модел

A2C (Advantage Actor-Critic) е алгоритъм за reinforcement learning, който комбинира два основни компонента:

1. Actor: Отговаря за избора на действия според текущата политика.
2. Critic: Оценява стойността на текущото състояние, което помага на актьора да подобри своята политика.

```
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class ActorCritic(nn.Module):
def __init__(self, input_shape, n_actions: int, use_leaky_relu: bool = False, channel_multiplier: int = 1):
super(ActorCritic, self).__init__()

self.conv = nn.Sequential(
nn.Conv2d(1, 32*channel_multiplier, kernel_size=8, stride=4), # Change input_shape[0] to 1
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
nn.Conv2d(32*channel_multiplier, 64*channel_multiplier, kernel_size=4, stride=2),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU(),
nn.Conv2d(64*channel_multiplier, 64*channel_multiplier, kernel_size=3, stride=1),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU()
)
self.fc = nn.Sequential(
nn.Linear(self._feature_size(input_shape), 512*channel_multiplier),
nn.LeakyReLU() if use_leaky_relu else nn.ReLU()
)

self.actor = nn.Sequential(
nn.Linear(512*channel_multiplier, n_actions),
nn.Softmax(dim=-1) # Add softmax here
)
self.critic = nn.Linear(512*channel_multiplier, 1)

def _feature_size(self, shape):
with torch.no_grad():
return self.conv(torch.zeros(1, *shape)).view(1, -1).size(1)

def forward(self, x):
x = self.conv(x)
x = x.view(x.size(0), -1)
x = self.fc(x)

# Stabilize softmax input
actor_output = self.actor[0](x) # Get logits before softmax
actor_output = actor_output - torch.max(actor_output, dim=-1, keepdim=True).values
action_probs = self.actor[1](actor_output) # Apply softmax

state_value = self.critic(x)
```

Дефиниране на A2C модела

Класът ActorCritic дефинира архитектурата на A2C модела. Той се състои от два основни компонента: Actor и Critic

1. Actor: Генерира вероятности за всяко действие.
2. Critic: Оценява стойността на текущото състояние.

Forward метод

Методът forward дефинира как данните преминават през мрежата:

- action_probs: Вероятности за всяко действие.
- state_value: Оценка на стойността на текущото състояние.

Този A2C модел е проектиран да обработва визуални данни от играта и да генерира действия, които максимизират наградата. Моделът се състои от конволюционни и пълносвързани слоеве, които извличат характеристики от изображенията и ги преобразуват в вероятности за действия и оценка на състоянието. Този подход е ефективен за reinforcement learning задачи, където входните данни са сложни и многомерни.

A3C

```python
# Set the multiprocessing start method (only once)
if __name__ == "__main__":
    try:
        mp.set_start_method('spawn')  # Use 'spawn' for multiprocessing
    except RuntimeError:
        print("Multiprocessing context already set.")

if __name__ == "__main__":
    global_counter = mp.Value('i', -1)
    episode_count_lock = mp.Lock()
    global_model_lock = mp.Lock()
    stop_flag = mp.Value('b', False)

# Define the save directory
BASE_DIR = Path(__file__).resolve().parent
SAVE_DIR = BASE_DIR / "a2c_simple_movement_models"
LOG_FILE_NAME = BASE_DIR / "a2c_simple_movement_models" / "episodes_log.log"
LOAD_MODEL_EPISODE = -1
LEARNING_RATE = 1e-4
GAMMA = 0.99
ENTROPY_COEF = 0.05
MODEL_SAVE_EPISODES = 1000
N_STEPS = 10
MAX_STEPS_ENV = 5000
MAX_EPISODES = 50000
ONE_LIFE = True

if not os.path.exists(SAVE_DIR):
    os.makedirs(SAVE_DIR)

def save_model(global_model, episode, save_dir=SAVE_DIR):
    model_path = save_dir / f"global_model_episode_{episode}.pth"""   """
    torch.save(global_model.state_dict(), model_path)
    print(f"Model saved at episode {episode} to {model_path}")

def load_model(global_model, episode, save_dir=SAVE_DIR):
    model_path = save_dir / f"global_model_episode_{episode}.pth"
    global_model.load_state_dict(torch.load(model_path))
    print(f"Model loaded from episode {episode} from {model_path}")
```

```python
def compute_advantages_and_update(
        global_model,
        local_model,
        optimizer,
        states,
        actions,
        rewards,
        next_states,
        done,
        global_model_lock):

    # Compute loss on LOCAL model
    states = torch.stack(states).squeeze(dim=1).to(device)
    actions = torch.tensor(actions, dtype=torch.long, device=device)
    rewards = torch.tensor(rewards, dtype=torch.float32, device=device)

    # Forward pass with LOCAL model
    action_probs, values = local_model(states)
    values = values.squeeze()

    # Compute returns and advantages (as before)
    R = local_model(next_states[-1].unsqueeze(0).unsqueeze(0))[1].item()
    returns = []
    for r in reversed(rewards):
        R = r + GAMMA * R
        returns.insert(0, R)
    returns = torch.tensor(returns, device=device)

    advantages = returns - values

    # Loss calculation
    log_probs = torch.log(action_probs.gather(1, actions.unsqueeze(1)))
    policy_loss = -(log_probs * advantages.detach()).mean()
    value_loss = 0.5 * (returns - values).pow(2).mean()
    entropy = -(action_probs * torch.log(action_probs)).sum(dim=1).mean()
    entropy_loss = -ENTROPY_COEF * entropy

    total_loss = policy_loss + value_loss + entropy_loss

    optimizer.zero_grad()
    optimizer.zero_grad()
    total_loss.backward()

    torch.nn.utils.clip_grad_norm_(local_model.parameters(), 0.5)

    with global_model_lock:
        for local_param, global_param in zip(local_model.parameters(),
                                             global_model.parameters()):
            if global_param.grad is None:
                global_param.grad = torch.zeros_like(global_param.data)
            global_param.grad.data.copy_(local_param.grad.data)

        optimizer.step()

        local_model.load_state_dict(global_model.state_dict())
```

```python
def worker(global_model,
           optimizer,
           worker_id,
           env_name,
           n_actions,
           global_counter,
           episode_count_lock,
           global_model_lock,
           stop_flag):

    env = gym_super_mario_bros.make(env_name, max_episode_steps=MAX_STEPS_ENV)
    env = JoypadSpace(env, SIMPLE_MOVEMENT)
    input_shape = preprocess_smaller_state(env.reset(), device).shape
    local_model = ActorCritic(input_shape, n_actions, True, 2).to(device)
    local_model.train()

    print(f"Worker {worker_id} started.")

    with global_model_lock:
        local_model.load_state_dict(global_model.state_dict())

    while not stop_flag.value and global_counter.value <= MAX_EPISODES:
        # old_state_dict = {}
        # for key in global_model.state_dict():
        #     old_state_dict[key] = global_model.state_dict()[key].clone()
        start_time = time.time()
        state = preprocess_smaller_state(env.reset(), device)
        done = False
        truncated = False
        total_reward = 0
        estimated_reward = 0
        remaining_lives = 2

        while not done and not truncated:
            #print(f"Worker {worker_id} calls")
            states, actions, rewards, next_states = [], [], [], []
            for _ in range(N_STEPS):
                env.render()
                state = state.unsqueeze(0).unsqueeze(0) # add batch and channel dimensions.
                with torch.no_grad():
                    action_probs, value = local_model(state)
```

```
        while not done and not truncated:
            #print(f"Worker {worker_id} calls")
            states, actions, rewards, next_states = [], [], [], []
            for _ in range(N_STEPS):
                env.render()
                state = state.unsqueeze(0).unsqueeze(0) # add batch and channel dimensions
                with torch.no_grad():
                    action_probs, value = local_model(state)
                action = torch.multinomial(action_probs, 1).item()

                next_state, reward, done, truncated, info = env.step(action)
                next_state = preprocess_smaller_state(next_state, device)

                if remaining_lives > info['life']:
                    remaining_lives-=1
                    reward -= 100

                total_reward += reward
                estimated_reward += value.item()
                states.append(state)
                actions.append(action)
                rewards.append(reward)
                next_states.append(next_state)  # Store next_state for bootstrapping

                state = next_state
                if done or truncated:
                    break


            compute_advantages_and_update(global_model,
                                          local_model,
                                          optimizer,
                                          states,
                                          actions,
                                          rewards,
                                          next_states,
                                          done,
                                          global_model_lock)
```

```
            if ONE_LIFE and remaining_lives <= 1:
                break



        elapsed_time = time.time() - start_time

        with episode_count_lock:
            global_counter.value += 1
            current_episode = global_counter.value
            record_info_for_worker(LOG_FILE_NAME, current_episode, worker_id, elapsed_time, total_reward, info)
            print(f"Worker {worker_id} finished episode: {current_episode}. Time: {elapsed_time:.2f}. Total reward: {

        if current_episode % MODEL_SAVE_EPISODES == 0:
            with global_model_lock:
                save_model(global_model, current_episode)

    print(f"Worker {worker_id} stopped.")
```

```python
if __name__ == "__main__":
    print("Main thread started")

    env = gym_super_mario_bros.make('SuperMarioBros-v0')
    env = JoypadSpace(env, SIMPLE_MOVEMENT)
    input_shape = preprocess_smaller_state(env.reset(), device).shape
    n_actions = env.action_space.n

    global_model = ActorCritic(input_shape, n_actions, True, 2).to(device)
    if LOAD_MODEL_EPISODE != -1:
        load_model(global_model, LOAD_MODEL_EPISODE, SAVE_DIR)
    global_model.share_memory()
    global_model.train()

    optimizer = SharedAdam(global_model.parameters(), lr=LEARNING_RATE)

    num_workers = 8 #mp.cpu_count() - 8
    processes = []
    for worker_id in range(num_workers):
        p = mp.Process(target=worker,
                       args=(global_model,
                             optimizer,
                             worker_id,
                             'SuperMarioBros-v0',
                             n_actions,
                             global_counter,
                             episode_count_lock,
                             global_model_lock,
                             stop_flag))
        p.start()
        processes.append(p)
```

Този код имплементира **A3C (Asynchronous Advantage Actor-Critic)**, алгоритъм за reinforcement learning, който използва множество паралелни агенти (workers) за обучение на един глобален модел. Всеки агент взаимодейства със своя копие на средата и изпраща градиентите на глобалния модел, който се актуализира асинхронно. Този подход позволява бързо и ефективно обучение, особено в сложни среди като видеоигрите.

Какво прави A3C?

A3C е алгоритъм, който комбинира A2C и **Асинхронност.**

Основните характеристики на A3C са:

- **Асинхронност**: Множество агенти работят паралелно, като всеки агент има своя копия на средата и локален модел.
- **Паралелно обучение**: Всеки агент изчислява градиентите на локалния си модел и ги изпраща на глобалния модел, който се актуализира асинхронно.
- **Ефективност**: Паралелното обучение значително ускорява процеса на обучение, тъй като множество агенти събират опит едновременно.

Кодът се състои от няколко ключови части:

1. Инициализация на средата и модела

- **Среда**: Използва се gym_super_mario_bros за създаване на средата на Super Mario Bros.
- **Модел**: Глобалният модел (global_model) е инстанция на ActorCritic, който се споделя между всички работници (workers).
- **Оптимизатор**: Използва се SharedAdam, който позволява споделяне на параметрите на оптимизатора между процесите.

2. Работници (Workers)

Всеки работник (worker) е отделен процес, който:

- Има своя копия на средата и локален модел.
- Взаимодейства със средата, събира преживявания и изчислява градиенти.
- Актуализира глобалния модел с изчислените градиенти.

3. Изчисляване на advantages и актуализация на модела

Функцията compute_advantages_and_update изчислява загубите за актьора и критика, и актуализира глобалния модел:

4. Паралелно обучение

Главният процес стартира множество работници и управлява процеса на обучение:

5. Спиране на обучението

Обучението може да бъде спряно чрез натискане на клавиша **SPACE** или чрез сигнал **Ctrl+C**:

Колко бързо се справя със задачата?

A3C е много ефективен алгоритъм за reinforcement learning, особено в сложни среди като видеоигрите. Паралелното обучение позволява на множество агенти да събират опит едновременно, което значително ускорява процеса на обучение. В зависимост от броя на работниците и сложността на средата, A3C може да достигне добри резултати за сравнително кратко време.

- **Бързина**: Паралелното обучение позволява на A3C да се справя със задачата много по-бързо от синхронните алгоритми като A2C.
- **Ефективност**: A3C е по-ефективен от DQN, тъй като използва advantages за по-стабилно обучение.
- **Скалируемост**: Алгоритъмът може да се мащабира до голям брой работници, което го прави подходящ за сложни задачи.

Заключение

Този код имплементира A3C за обучение на AI агент да играе Super Mario Bros. Паралелното обучение и асинхронната актуализация на модела правят A3C много ефективен и бърз алгоритъм за reinforcement learning. Той е подходящ за сложни задачи, където е необходим голям брой епизоди за обучение.