Pusher Ortamında PPO Algoritmasının Uygulanması

1. Giriş

PPO, yüksek performans sunan ve politika optimizasyonuna dayalı bir derin pekiştirmeli öğrenme algoritmasıdır. Bu çalışmada, Pusher ortamında bir nesneyi belirlenen hedefe itmek için PPO kullanılarak bir ajan eğitilmiştir.

2. PPO Algoritması

PPO, politika tabanlı pekiştirmeli öğrenme algoritmalarından biridir. Geleneksel politika gradyan yöntemlerinin iyileştirilmiş bir versiyonu olan PPO, ajanların eğitim sürecinde daha kararlı ve verimli öğrenmelerini sağlamak için kayıp fonksiyonunu sınırlı güncellemelerle optimize eder. PPO'nun temel avantajları:

- Politikayı güncellerken ani değişiklikleri sınırlayan **klip yöntemi** ile daha güvenilir bir eğitim süreci sağlar.
- Hedef fonksiyonunda politika değişimlerini aşırı büyütmeyi engeller, böylece model daha kararlı olur.

3. Pusher Ortamı

Pusher ortamı, OpenAI Gym'deki fizik tabanlı bir simülasyon ortamıdır. Bu ortamda bir robot kol, belirli bir nesneyi hedef konuma itmeye çalışır. Ajan, doğru eylemleri seçerek nesneyi hedefe yaklaştırmaya çalışır ve ödül sinyalleriyle hedefe ulaşmak için yönlendirilir.

Amaç:

- Ajanın, belirlenen hedef konumuna nesneyi başarılı bir şekilde itmesi.

Ödül Fonksiyonu:

- Nesnenin hedefe yakınlığı arttıkça ödüllendirme yapılır. Hedefe uzaklık azaldıkça pozitif ödül alır ve amaç, toplam ödülü maksimize etmektir.

4. Avantajlar ve Dezavantajlar

Avantajlar

1. Stabil ve Güvenilir Eğitim Süreci
2. Yüksek Performans
3. Basit ve Etkili Uygulama
4. Geniş Uygulanabilirlik

Dezavantajlar

1. Hiperparametre Hassasiyeti
2. Verimlilik Sorunları
3. Görev Karmaşıklığına Göre Performans Dalgalanmaları
4. Yüksek Hesaplama Maliyeti

5. Sonuçlar

PPO algoritması, Pusher ortamında nesneyi hedefe itme görevinde başarılı sonuçlar vermiştir. Eğitim süreci boyunca ajan, hedefe ulaşma oranını artırarak toplam ödülü maksimize etmiştir. İlk başlarda rasgele hareket eden ajan, eğitim ilerledikçe daha kararlı ve doğru hamleler yapmayıöğrenmiştir.

Başarı Kriteri:

- Başarı oranı, ajan nesneyi hedefe yaklaştırabildiğinde artış göstermiştir.
- Eğitim sonrasında ajan, yüksek bir başarı oranına ulaşmıştır.

6. Kod ve Değerlendirme

```
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy
import os
```

gymnasium, RL eğitim ortamlarını oluşturmamızı sağlayan kütüphanedir.

- stable_baselines3 paketinden PPO (Proximal Policy Optimization) algoritmasını ve RL değerlendirme işlevini içerir.
- DummyVecEnv, çevreyi vektörize ederek paralel simülasyon yapabilmemizi sağlar.
- os ise dosya ve klasör işlemleri için kullanılan bir Python kütüphanesidir.

```
env_name='Pusher-v5'
env =gym.make(env_name,render_mode="human")
```

env_name değişkenine Pusher-v5 ortamını tanımlıyoruz.

- gym.make() işlevi, belirtilen isme göre bir ortam oluşturur. Burada ortamın görsel olarak render_mode="human" ile gösterilmesi sağlanır.

```
episodes=5
for episode in range(1,episodes+1):
    obs=env.reset()
    done=False
    score=0
    while not done:
        env.render()
        action=env.action_space.sample()
        obs,reward,done, _,info=env.step(action)
        score+=reward
    print('episode:{} score{}'.format(episode,score))

env.close()
```

pisodes değişkeni, ortamın kaç kez simüle edileceğini belirler.

- Döngü her simülasyon (episode) için çalışır.
- env.reset() her bir simülasyon başında ortamı sıfırlar ve başlangıç gözlem değerini (obs) döndürür.
- done değişkeni, simülasyonun bitip bitmediğini izler; score ise her bölüm için toplam puanı hesaplar.

while not done döngüsü, simülasyon bitene kadar devam eder.

- env.render() ortamın görsel olarak çizilmesini sağlar.
- env.action_space.sample() rastgele bir aksiyon seçer.
- env.step(action) seçilen aksiyonu uygular ve ortamın bir sonraki durumunu (obs), ödülü (reward), ve bitiş durumunu (done) döndürür.
- score += reward ile ödül toplanır.

Her bölümün sonunda toplam skor yazdırılır.

- env.close() ortamı kapatarak görsel pencereleri sonlandırır.

```
env.action_space
env.observation_space
log_path=os.path.join('Trainig','Logs')
model =PPO('MlpPolicy',env,verbose=1,tensorboard_log=log_path)
os.makedirs(log_path, exist_ok=True)
model.learn(total_timesteps=1000000)
```

log_path, TensorBoard kayıtlarının saklanacağı yolu belirler.

- PPO modelini MlpPolicy (Çok Katmanlı Algılayıcı Politika) ile başlatır, verbose=1 eğitimin ilerlemesini ayrıntılı olarak gösterir.
- os.makedirs(log_path, exist_ok=True) ile log klasörünün var olup olmadığını kontrol eder, yoksa oluşturur.
- Modeli 1 milyon adım boyunca eğitir.

```
mean_reward, std_reward = evaluate_policy(model, env, n_eval_episodes=10)
print(f"Mean Reward: {mean_reward}, Std: {std_reward}")
```

evaluate_policy işlevi, eğitilen politikayı 10 bölüm üzerinde değerlendirir ve ortalama ödül (mean_reward) ile standart sapmayı (std_reward) hesaplar.

- Değerlendirme sonuçları yazdırılır.

```
ppo_path=os.path.join('Trainig','Saved_Model','PPO_pusher_model_4')
model.save(ppo_path)
del model
```

Model ppo_path yolunda kaydedilir.

- Model bellekte silinir (del model), ardından dosyadan yeniden yüklenir.

```
model=PPO.load(ppo_path,env)
evaluate_policy(model,env,n_eval_episodes=5, render=True)
```

Modeli yeniden yükledikten sonra 5 bölüm üzerinde değerlendirir ve sonuçları görsel olarak (render=True) gösterir.

Değerlendirme :

ilk iterasyon:

```
----------------------------------------
| rollout/              |              |
|    ep_len_mean        | 100          |
|    ep_rew_mean        | -120         |
| time/                 |              |
|    fps                | 59           |
|    iterations         | 2            |
|    time_elapsed       | 69           |
|    total_timesteps    | 4096         |
| train/                |              |
|    approx_kl          | 0.013347901 |
```

```
|    clip_fraction        | 0.133       |
|    clip_range           | 0.2         |
|    entropy_loss         | -9.89       |
|    explained_variance   | -0.012      |
|    learning_rate        | 0.0003      |
|    loss                 | 26          |
|    n_updates            | 10          |
|    policy_gradient_loss | -0.0198     |
|    std                  | 0.99        |
|    value_loss           | 171         |
-----------------------------------------
```

son iterasyon:

```
-----------------------------------------
| rollout/              |               |
|    ep_len_mean        | 100           |
|    ep_rew_mean        | -32.4         |
| time/                 |               |
|    fps                | 62            |
|    iterations         | 489           |
|    time_elapsed       | 15910         |
|    total_timesteps    | 1001472       |
| train/                |               |
|    approx_kl          | 0.028509943   |
|    clip_fraction      | 0.288         |
|    clip_range         | 0.2           |
|    entropy_loss       | 4.91          |
|    explained_variance | 0.979         |
|    learning_rate      | 0.0003        |
|    loss               | 0.244         |
|    n_updates          | 4880          |
|    policy_gradient_loss | -0.00719    |
|    std                | 0.121         |
|    value_loss         | 1.12          |
-----------------------------------------
```

Başlangıç iterasyonunda , model henüz çevreyi keşfetme ve temel öğrenme adımlarını gerçekleştirme aşamasındadır. Bu aşamada ep_rew_mean değerinin -120 gibi düşük bir seviyede olması, modelin ödüllendirme sistemiyle henüz etkin bir şekilde uyum sağlayamadığını gösterir. Ayrıca, explained_variance değerinin negatif olması ve loss ile value_loss değerlerinin yüksek çıkması, modelin ödül tahmininde zorlandığını ve henüz stabil bir politika geliştiremediğini ifade eder. Entropi kaybının negatif olması ve standart sapmanın yüksek olması da, başlangıç iterasyonunda keşfe odaklı bir politikanın izlendiğini ve modelin çevrede daha fazla deneme-yanılma yoluyla öğrenim gerçekleştirdiğini gösterir.
Son iterasyonda (ilk raporda) ise modelin ödüllendirme mekanizmasıyla daha uyumlu hale geldiğini ve politikayı optimize ettiğini görebiliyoruz. ep_rew_mean değeri -32.4'e yükselmiş, bu da ödül performansında belirgin bir iyileşme anlamına gelir. explained_variance değeri 0.979 ile oldukça yüksek, bu da modelin ödülleri doğru bir şekilde tahmin edebildiğini ve çevreyi daha iyi anlayarak politikalarını geliştirdiğini gösterir. loss ve value_loss değerlerinin düşmesi, modelin daha stabil hale geldiğini ve ödül tahminlerinde daha az hata yaptığını yansıtır. Ayrıca, policy_gradient_loss ve düşük std değeri, modelin daha kararlı ve optimize edilmiş bir politika izlediğini göstermektedir. Bu gelişmeler, modelin eğitim süreci boyunca verimli bir öğrenme gerçekleştirdiğini ve öğrenme sürecinin sonunda performansını ciddi oranda artırdığını kanıtlar.

Car Racing Ortamında PPO Algoritmasının Uygulanması

1. Giriş

Car Racing ortamında Proximal Policy Optimization (PPO) algoritmasının uygulanmasına odaklanmıştır. PPO, yüksek performans odaklık ve optimizasyonuna dayalı bir derin peliştirmeli öğrenme algoritmasıdır. Bu projede, PPO algoritması ile eğitilen bir ajan, Car Racing ortamında parkuru başarıyla tamamlamak için eğitilmiştir.

2. PPO Algoritması

PPO, politika tabanlı pekiştirmeli öğrenme algoritmalarından biridir ve politika güncellemelerini güvenilir bir şekilde yapmak için klipli bir optimizasyon yöntemi kullanır. PPO'nun avantajları arasında stabil eğitim süreci ve yüksek performans sayılabilir. Bu algoritma, sürekli aksiyon alanlarına sahip görevlerde başarılı sonuçlar verebilir.

PPO'nun Temel Özellikleri:

- **Klipli Güncelleme**: Politika güncellemelerinde aşırı değişiklikleri önler ve stabil bir öğrenme sağlar.
- **Yüksek Performans**: Hem sürekli hem de ayrık aksiyon alanlarında yüksek başarı gösterir.
- **Basit Yapı**: Diğer pekiştirmeli öğrenme algoritmalarına kıyasla daha kolay uygulanabilir.

3. Car Racing Ortamı

Car Racing ortamı, OpenAI Gym'deki sürekli aksiyon alanına sahip bir simülasyon ortamıdır. Bu ortamda bir yarış aracı, rastgele oluşturulmuş parkur üzerinde ilerlemeye çalışır. Ajanın doğru eylemleri seçerek parkur dışına çıkmadan ilerlemesi ve toplam ödülü maksimize etmesi beklenir.

Amaç:

- Aracın parkur dışına çıkmadan mümkün olduğunca hızlı ve verimli bir şekilde ilerlemesi.

Ödül Fonksiyonu:

- Araç parkurda ilerledikçe pozitif ödül alır, ancak parkur dışına çıktığında ödül kaybeder. Amacımız, toplam ödülü maksimize etmektir.

4. Avantajlar ve Dezavantajlar

Avantajlar

1) Stabil Eğitim: PPO algoritmasında, güncellemelerde sınır olduğu için eğitim süreci daha kararlı haldedir.

2) Performans: PPO, hem sürekli hem de ayrık eylem alanlarında etkili sonuçlar vermesi onu daha performanslı var.

3) Uygulama Kolaylığı: PPO, diğer politika optimizasyon yöntemleri gibi karmaşık değildir.

4) Genelleme Kapasitesi: Farklı tür görevlerde iyi sonuçlar verebilir ve çeşitli ortamlarda uygulanabilir.

Dezavantajlar

1) Hiperparametre Hassasiyeti : PPO'nun öğrenme oranı, klip oranı, batch boyutu gibi çeşitli hiperparametrelere karşı hassasiyeti yüksektir.

2) Veri Verimliliği Eksikliği: PPO, eski deneyimlerin tekrar kullanımında çok verimli değildir.

3) Yüksek Hesaplama Maliyeti: PPO, özellikle büyük ağ yapılarında veya karmaşık görevlerde yüksek hesaplama maliyetine sahiptir.

4) Performans Dalgalanmaları: PPO algoritması belirli karmaşık görevlerde dalgalanan performans gösterebilir.

5. Sonuçlar

PPO algoritması, Car Racing ortamında başarılı bir eğitim süreci geçirmiş ve aracı parkur üzerinde etkili bir şekilde kontrol etmeyi öğrenmiştir. Eğitim sırasında ajan, hızlanma, fren yapma ve dönüşlerde daha doğru kararlar alarak toplam ödülü artırmıştır.

Eğitim Sonuçları:

- **Başarı Oranı**: Eğitim sonuna doğru ajan, parkurda daha uzun süre kalmaya başlamış ve ödül kazanma oranı artmıştır.
- **Öğrenme Eğrisi**: İlk başlarda ajan parkur dışına sıkça çıkarken, eğitim ilerledikçe daha kararlı bir sürüş sergilemiştir.

Performans Göstergeleri:

- **Toplam Ödül**: Eğitim boyunca ajan, ödülü maksimize edecek stratejiler geliştirmiştir.
- **Kararlılık**: PPO'nun klip yöntemi, eğitim sırasında aracın dengesiz manevralar yapmasını engelleyerek daha stabil bir sürüş performansı sağlamıştır.

6. Kod ve Değerlendirme

```
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy
import os
```

gymnasium, stable_baselines3, ve os kütüphanelerini içe aktarıyoruz:

- *gymnasium: OpenAI Gym ortamlarını kullanmak için.*
- *stable_baselines3: Önceden tanımlanmış, kullanıma hazır derin takviye öğrenme algoritmalarını içeriyor. Burada **PPO (Proximal Policy Optimization)** kullanacağız.*
- *os: Dosya yollarını yönetmek için gerekli.*

In [2]:

```
env_name='CarRacing-v3'
env =gym.make(env_name,render_mode="human")
env.action_space
env.observation_space
```

CarRacing-v3 adlı Gym ortamını başlatıyoruz:

- *gym.make: Belirtilen ortamı (CarRacing-v3) başlatıyor.*
- *render_mode="human": Ortamın görselleştirilmesini sağlıyor.*
- *env.action_space: Ortamın eylem alanını kontrol ediyoruz (örneğin, hangi tür eylemler yapılabilir).*

- *env.observation_space: Ortamın gözlem alanını kontrol ediyoruz (örneğin, ortamdan alınabilecek gözlemler).*

```
episodes=5
for episode in range(1,episodes+1):
    obs=env.reset()
    done=False
    score=0
    while not done:
        env.render()
        action=env.action_space.sample()
        obs,reward,done, _,info=env.step(action)
        score+=reward
    print('episode:{} score{}'.format(episode,score))

env.close()
```

eş bölüm için eğitim gerçekleştireceğiz:

- *episodes: Eğitim yapılacak bölüm sayısını belirliyoruz.*
- *env.reset(): Her yeni bölümde ortamı sıfırlıyoruz, başlangıç gözlemi (obs) alıyoruz.*
- *done: Bölüm bitmediği sürece bu değer False olarak kalacak.*
- *score: Bu bölümde elde edilen toplam ödülü tutmak için kullanılan değişken.*

Bölüm sonuna kadar ortamı simüle ediyoruz:

- *env.render(): Ortamı görselleştirir.*
- *env.action_space.sample(): Rastgele bir eylem seçer (temel bir eğitim süreci değil, sadece gözlem yapıyoruz).*
- *env.step(action): Ortama seçilen eylemi uygular ve yeni gözlem, ödül, bölüm durumu (done) ve bilgi döner.*
- *score += reward: Toplam ödülü güncelleriz.*
- *Bölüm bittiğinde, toplam puanı ekrana yazdırır.*

```
env=gym.make(env_name,render_mode="human")
env=DummyVecEnv([lambda:env])
```

Ortamı kapatıp PPO modeli için yeni bir ortam hazırlıyoruz:

- *env.close(): Mevcut ortamı kapatır.*
- *DummyVecEnv: Ortamı vektörleştirilmiş bir biçimde sararak paralel öğrenmeyi mümkün kılar.*

```
log_path=os.path.join('Trainig','Logs')
model =PPO('CnnPolicy',env,verbose=1,tensorboard_log=log_path)
os.makedirs(log_path, exist_ok=True)
```

PPO modelini oluşturuyoruz:

- *log_path: TensorBoard loglarının kaydedileceği klasör yolunu oluşturuyoruz.*
- *PPO('CnnPolicy', env, verbose=1, tensorboard_log=log_path): PPO modelini CNN tabanlı bir politika ile oluşturuyoruz.*
- *os.makedirs(log_path, exist_ok=True): Log klasörünü oluşturur (varsa tekrar oluşturmaz).*

```
model.learn(total_timesteps=1000000)
```

Modeli 1.000.000 zaman adımı için eğitiyoruz.

```
ppo_path=os.path.join('Trainig','Saved_Model','PPO_driving_model')
model.save(ppo_path)
del model
```

```
model=PPO.load(ppo_path,env)
```

Modeli kaydetme ve yükleme:

- *model.save: Eğitilen modeli belirtilen yola kaydeder.*
- *del model: Mevcut model nesnesini bellekte siler.*
- *PPO.load: Kaydedilen modeli yükler.*

```
evaluate_policy(model,env,n_eval_episodes=5, render=True)
env.close()
```

evaluate_policy: Modeli 5 bölüm boyunca değerlendirir, performansını gözlemlemek için ortamı görselleştirir

```
episodes=5
for episode in range(1,episodes+1):
    reset_result = env.reset()
    obs = reset_result[0] if isinstance(reset_result, tuple) else reset_result
    done=False
    score=0
    while not done:
        env.render()
        action, _=model.predict(obs)
        obs,reward,done, _,info=env.step(action)
        score+=reward
    print('episode:{} score: {}'.format(episode,score))
env.close()
```

ğitilen modelle test yapıyoruz:

- *reset_result = env.reset(): Ortamı sıfırlar.*
- *obs = reset_result[0] if isinstance(reset_result, tuple) else reset_result: Gymnasium ile uyumluluk için gözlemi ayarlar.*

Modelin tahminlerini kullanarak bölüm simülasyonları yapıyoruz:

- *action, _ = model.predict(obs): Modelin tahmin ettiği eylemi alırız.*
- *env.step(action): Eylemi uygulayıp yeni gözlem, ödül ve bölüm durumunu döner.*
- *score += reward: Toplam ödülü güncelleriz ve bölüm tamamlandığında skoru yazdırırız.*

DEĞERLENDİRME

ilk iterasyon:

```
-----------------------------------------
| time/                |            |
|    fps               | 38         |
|    iterations        | 2          |
|    time_elapsed      | 107        |
|    total_timesteps   | 4096       |
| train/               |            |
|    approx_kl         | 0.009457771 |
|    clip_fraction     | 0.0882     |
|    clip_range        | 0.2        |
|    entropy_loss      | -4.25      |
|    explained_variance | -0.0054    |
|    learning_rate     | 0.0003     |
|    loss              | 0.214      |
```

| n_updates | 10 |
| policy_gradient_loss | -0.00657 |
| std | 0.997 |
| value_loss | 0.599 |
----

son iterasyon

| time/ | |
| fps | 31 |
| iterations | 489 |
| time_elapsed | 32043 |
| total_timesteps | 1001472 |
| train/ | |
| approx_kl | 0.056289725 |
| clip_fraction | 0.416 |
| clip_range | 0.2 |
| entropy_loss | -5.05 |
| explained_variance | 0.963 |
| learning_rate | 0.0003 |
| loss | 0.581 |
| n_updates | 4880 |
| policy_gradient_loss | 0.00904 |
| std | 1.3 |
| value_loss | 6.57 |

Eğitim sürecinin başından sonuna kadar modelin performansında önemli bir gelişim gözlemlenmiştir. Başlangıçta, model çevreyi keşfetme aşamasında olup, entropy_loss değeri -4.25 civarındayken, eğitim sonunda bu değer -5.05'e düşerek, ajanın daha belirgin bir şekilde sömürüye (exploit) geçtiğini gösteriyor. Ayrıca, approx_kl değeri başlangıçta 0.0095 iken, sonrasında 0.0563'e yükselmiş; bu, ajan politikasının eski politikadan önemli ölçüde farklılaştığını ve modelin daha karmaşık bir strateji geliştirdiğini ortaya koyuyor.
Modelin performansını daha da gösteren bir diğer önemli metrik ise explained_variance'dir. Eğitim başlangıcında -0.0054 gibi düşük bir değere sahipken, eğitim sonrasında 0.963 gibi yüksek bir değere ulaşmıştır. Bu, modelin ödül tahminlerinin oldukça güvenilir olduğunu ve çevreyi başarılı bir şekilde öğrendiğini gösteriyor. loss değeri ise eğitim sürecinde artarak 0.581'e ulaşmış, bu da modelin daha karmaşık ödül yapısını öğrenmeye çalıştığını ve çevreyle etkileşimde derinleştiğini gösteriyor.
Diğer bir önemli metrik olan clip_fraction, başlangıçta %8.82 iken eğitim sonunda %41.6'ya çıkmış. Bu, modelin başlangıçta daha az klipslenmiş güncellemeler yaptığı, ancak eğitim süreci ilerledikçe klipslenen güncellemelerin arttığını gösteriyor. Bu durum, modelin keşif yaparken daha fazla güncelleme gerçekleştirdiğini ve daha geniş bir strateji setini değerlendirdiğini düşündürüyor.
Son olarak, std (standart sapma) değeri başlangıçta 0.997 iken, eğitim sonunda 1.3'e çıkmış. Bu, modelin aksiyon seçiminde daha fazla risk almaya başladığını ve çevrede daha kapsamlı bir keşif yapmayı tercih ettiğini gösteriyor.
Özetle, model eğitim sürecinde oldukça stabil bir şekilde gelişmiş ve çevresine daha uyumlu hale gelmiştir. Eğitim boyunca ajanın keşiften sömürüye geçişi, ödül tahminlerinin doğruluğu ve aksiyon seçimindeki risk artışı modelin çevreyi öğrenme başarısını ortaya koymaktadır.

Pusher Ortamında DDPG Algoritması

(Deep Deterministic Policy Gradient)

1. Kütüphaneler ve Sınıflar

Kodun başında, PyTorch ve NumPy gibi gerekli kütüphaneler içe aktarılır. Ayrıca, MLPActorCritic ve ReplayBuffer adında iki özel sınıf da yüklenir:

*Bu belirtilen sınıflar, aynı dosyadaki farklı kodlardır.

- **torch** ve **torch.nn**: PyTorch kütüphanesi; sinir ağları oluşturmak, eğitim yapmak ve GPU kullanarak hızlandırma sağlamak için kullanılır.
- **numpy**: Matematiksel işlemler ve diziler için kullanılır.
- **MLPActorCritic** ve **ReplayBuffer**: DDPG için özel olarak oluşturulmuş sınıflardır. MLPActorCritic, aktör ve critic ağlarını tanımlar; ReplayBuffer, deneyimleri depolayarak eğitimde tekrar kullanmamızı sağlar.

2. DDPGAgent Sınıfının Özellikleri

Sınıf, bir ajan oluşturur ve aşağıdaki özellikleri içerir:

- **env**: Ajanın içinde hareket edeceği çevre.
- **gamma**: 0.99 olarak belirlenen indirim faktörü. Gelecekteki ödüllerin önemini azaltır ve ajanın uzun vadeli ödülleri dikkate almasını sağlar.
- **tau**: 0.005 olarak belirlenmiş soft güncelleme katsayısı. Hedef ağı (target network) güncellemek için kullanılır.
- **device**: GPU veya CPU'da çalışmak üzere cihazı seçer.

3. Aktör ve Kritik Ağ

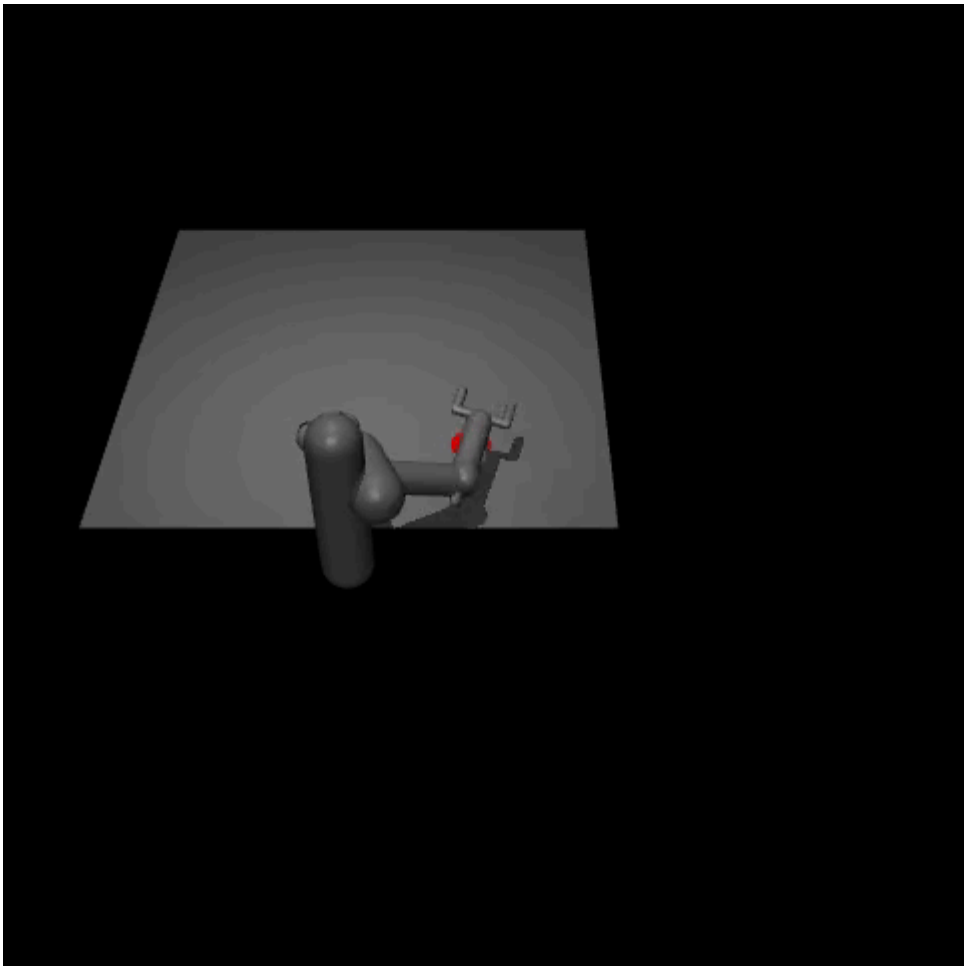
Bu ajan, bir aktör ve kritik ağ ile hedef versiyonlarını oluşturur:

- **self.ac**: Aktör-critic modeli MLPActorCritic sınıfından türetilir. Aktör, bir politika belirler ve critic, Q değerini tahmin eder.
- **actor_optimizer** ve **critic_optimizer**: Adam optimizasyon algoritması kullanılarak aktör ve critic için optimizasyon işlemi yapılır. self.ac.pi aktörün parametrelerini, self.ac.q_network ise critic ağını temsil eder.
- **self.target_ac**: Aktör-critic modelinin kopyası olan hedef ağdır. Soft güncellemeler için kullanılır.
- **self.update_target()**: Başlangıçta hedef ağı günceller.

4. Metodlar

- **replay_buffer**: Deneyimleri saklayacak olan buffer (tampon bellek) burada tanımlanır.
- **set_replay_buffer(replay_buffer)**: Ajan için harici bir replay buffer ataması yapılır.
- **update_target()**: Hedef ağı güncelleme işlemi yapılır.
- **next_action**: Hedef aktör ağı üzerinden bir aksiyon tahmin edilir.
- **target_q**: Critic ağının hedef değeri hesaplanır. Bu hesaplama, bir sonraki Q değeriyle ödülün indirimli toplamını kullanır.
- **action**: Aktörden alınan deterministik bir aksiyondur. detach işlemiyle grafikten ayrılır, bu da geri yayılım sırasında kullanılmasını engeller.
- **critic_loss**: Kritik kaybı hesaplanır; bu, tahmin edilen Q değeri ile hedef Q değeri arasındaki farktır.
- **Geri yayılım ve güncelleme işlemi**: Critic ağının parametreleri optimize edilir.
- **actor_loss**: Aktör kaybı negatif bir Q değeridir; çünkü aktör, Q değerini maksimize etmeye çalışır.
- **Aktör ağı güncellemesi**: Geri yayılım ve optimizasyon işlemiyle aktör ağı güncellenir.

• **act(obs)**: Verilen bir gözlemden eylem seçimi yapar.
• **update(batch_size=64)**: Replay buffer'dan örnekler alarak ağları günceller. Kritik kaybı ve aktör kaybı hesaplanır, ardından optimizasyon işlemleri gerçekleştirilir.
• **save(filepath)** ve **load(filepath)**: Model parametrelerini dosyaya kaydeder veya yükler.



Car Racing Ortamında DDPG Algoritması

DDPG (Deep Deterministic Policy Gradient)

1. Kütüphaneler ve Sınıflar Başlangıçta, PyTorch ve NumPy gibi gerekli kütüphaneler içe aktarılır. Ayrıca, özel olarak DDPG algoritması için oluşturulmuş *MLPActorCritic* ve *ReplayBuffer* sınıfları kullanılır. Bu sınıflar:

- PyTorch: Derin öğrenme modelleri oluşturmak ve GPU hızlandırması sağlamak için.
- NumPy: Matematiksel işlemler ve dizi yönetimi için.
- MLPActorCritic: Aktör ve kritik ağlarını tanımlar. Aktör, çevreye dair bir politika üretirken, kritik ağ Q değerlerini tahmin eder.
- ReplayBuffer: Deneyimleri depolayarak eğitimde tekrar kullanılmasını sağlar. Bu, ajanın daha etkin öğrenmesini destekler.

2. DDPGAgent Sınıfının Özellikleri Bu sınıf, ajanı (agent) oluşturur ve çeşitli hiperparametrelerle çalışır:

- *env*: Ajanın içinde hareket edeceği ortam.

- *gamma*: Ajanın uzun vadeli ödülleri dikkate almasını sağlayan indirim faktörü.
- *tau*: Hedef ağın (target network) güncellenmesi için kullanılan katsayı.
- *device*: Modelin GPU veya CPU'da çalışıp çalışmayacağını belirler.

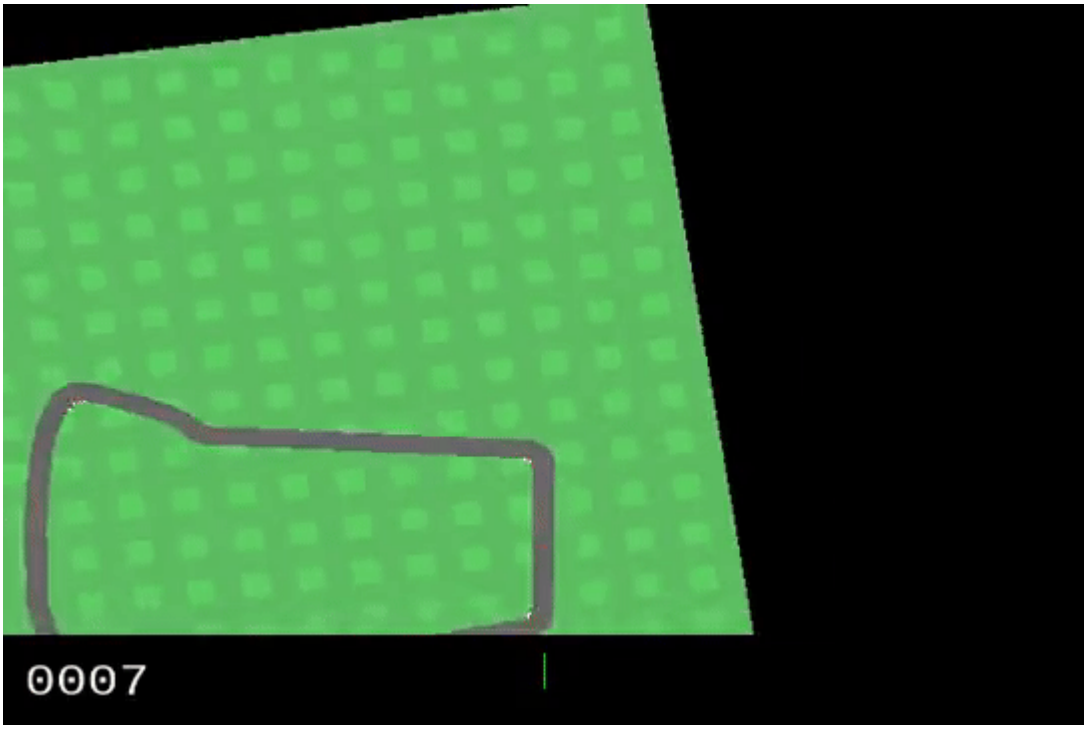
3. Aktör ve Kritik Ağ Bu ajan, aktör ve kritik ağlarını oluşturur ve hedef versiyonlarını tanımlar:

- *self.ac*: MLPActorCritic sınıfı kullanılarak tanımlanan aktör ve kritik modelidir. Aktör, bir eylem belirlerken kritik ağ Q değerlerini tahmin eder.
- *actor_optimizer* ve *critic_optimizer*: Aktör ve kritik ağları optimize etmek için Adam optimizasyon algoritması kullanılır.
- *self.target_ac*: Hedef ağı temsil eder, bu ağ soft güncellemelerle güncellenir.

4. Temel Metodlar

- *replay_buffer*: Eğitim verisini saklayan tampon bellektir.
- *update_target()*: Hedef ağı güncellemek için kullanılır, böylece ajan daha tutarlı hale gelir.
- *next_action* ve *target_q*: Hedef aktör ağı bir aksiyon önerir, kritik ağ ise bu aksiyona dair tahmin yaparak Q değerini hesaplar. Bu tahminler, ödüller ve indirim faktörleri ile birlikte gelecekteki tahminler için kullanılır.
- *critic_loss* ve *actor_loss*: Kritik ağ, tahmin edilen Q değerini iyileştirirken aktör, ajan için ideal aksiyonları seçer. Bu kayıplar minimize edilirken her iki ağ da optimize edilir.
- *act(obs)*: Ajan gözlemlerden aksiyon seçimi yapar.
- *update()*: Replay buffer'dan örnekler alır, kritik ve aktör kayıplarını günceller ve optimizasyonu gerçekleştirir.
- *save()* ve *load()*: Ajanın model parametrelerini kaydetme ve yükleme işlemlerini yönetir.

Bu yapı ve metodlarla DDPG algoritması, bir aktör-critic modeli üzerinden, sürekli aksiyon uzaylarına sahip ortamlarda politika belirlemeye odaklanır.



Pusher Ortamında SAC Algoritması

core.py dosyasında, sürekli aksiyon uzayına sahip ortamlarda çalışabilen bir MLPActorCritic sınıfı tanımlanır. Bu sınıf, SAC algoritmasında kullanılan bir aktör-kritik yapısıdır. SAC algoritması, belirli bir gözlem ve aksiyon çifti için beklenen ödülleri tahmin etmeye yönelik bir kritik ağı ve optimal aksiyonları seçmeye yönelik bir aktör ağı içerir. core.py içeriği, bu bileşenlerin doğru yapılandırılması ve öğrenme sürecine uygun bir şekilde optimize edilmesini sağlar.

Kod Yapısı ve Yöntemler
MLPActorCritic Sınıfı:

- **Aktör Ağı (self.pi)**: Çevreden gelen gözlemlerden aksiyon önerileri üretir. Tanh aktivasyon fonksiyonu ile -1 ile 1 arasında çıkış verir.
- **Kritik Ağı (self.q_network)**: Q değerlerini hesaplar, yani verilen gözlem ve aksiyon çiftlerinin beklenen ödüllerini tahmin eder.
- **Standart Sapma Parametresi (self.log_std)**: Aksiyonların olasılık dağılımındaki standart sapmayı kontrol eder.

Metodlar:

- **act**: Bir gözlem (obs) alarak ortalama ve standart sapma değerleriyle aksiyon dağılımı oluşturur ve bir aksiyon örnekler.
- **q**: Belirli bir gözlem ve aksiyon çifti için Q değerini hesaplar.

Sonuç
MLPActorCritic sınıfı, aktör-kritik yapısında esneklik ve performans sağlamak için doğru bir şekilde yapılandırılmıştır. Bu yapı, SAC algoritması gibi sürekli aksiyon uzayında çalışan algoritmalar için uygundur ve özellikle kompleks ortamlarda stabilite sağlar.

SAC.PY

1. Amaç

SACAgent sınıfı, SAC algoritmasının temel bileşenlerini içeren bir ajan oluşturur. Bu ajan, sürekli aksiyon uzayına sahip ortamlarda çalışabilir ve çevreden aldığı ödüllerle kendi politikasını geliştirir. SAC algoritmasının güçlü bir yönü, entropi katsayısını kullanarak öğrenme sürecinde bir denge sağlamasıdır.

2. Yapı ve Yöntemler

Aktör-Kritik Ağı: MLPActorCritic modeli kullanılarak oluşturulmuştur.

Replay Buffer: Deneyimleri depolamak ve mini-batch örnekler almak için dışarıdan tanımlanır.

Güncellemeler: Kritik ağ için target_q değerleriyle kayıp fonksiyonu minimize edilirken, aktör ağı için log_prob değeri minimize edilir.

Hedef Ağ: Yumuşak güncellemelerle kritik ağı taklit eder.

3. Avantajlar ve Dezavantajlar

Avantajlar:

Stabilite: Yumuşak güncelleme ve entropi katsayısı, öğrenmeyi stabilize eder.

Verimlilik: Sürekli aksiyon uzayında çalışabilir ve daha iyi performans gösterir.

Dezavantajlar:

Hesaplama Maliyeti: SAC algoritması karmaşıktır ve GPU üzerinde çalıştırılması önerilir.

MAİN.PY

main.py, SAC algoritmasıyla eğitilecek ajan için ana çalıştırma dosyasıdır. Bu dosya, eğitim sürecini başlatır ve eğitim sırasında gerekli kontrol adımlarını içerir.

Kod Yapısı ve Yöntemler

- Kayıt Dizini: checkpoints adlı bir dizin oluşturur ve eğitim sırasında model parametrelerinin burada saklanmasını sağlar.
- Eğitim Süreci: train modülünü içeri aktararak eğitim döngüsünü başlatır.

Sonuç ve Öneriler

main.py, eğitim sürecini yönetmek ve modeli belirli aralıklarla kaydetmek için temel bir yapı sunar. Bu dosya, eğitim sürecinin başlatılmasında gerekli adımları sağlamakla birlikte, eğitim verimliliğini artırmak için farklı ortam yapılandırmaları veya hiperparametre ayarları üzerinde değişiklikler yapılabilir. Eğitim çıktılarını izlemek ve değerlendirmek için ara kayıt dosyalarının belirli aralıklarla kaydedilmesi yararlı olabilir.

REPLAY_BUFFER.PY

replay_buffer.py, deneyim tekrar oynatma (experience replay) için kullanılan ReplayBuffer sınıfını içerir. SAC gibi algoritmalar için deneyim tekrar oynatma, öğrenme sürecinde geçmiş deneyimlerden faydalanarak daha stabil ve verimli bir eğitim sağlar.

Kod Yapısı ve Yöntemler

ReplayBuffer Sınıfı:

- Yapıcı Metot (__init__): Gözlem (state) ve aksiyon (action) boyutları ile maksimum boyutta bir deque yapısı oluşturur.
- add Metodu: Yeni bir deneyim örneğini (state, action, reward, next_state, done) buffer'a ekler. Eklenen verinin boyutlarını doğrular.
- sample Metodu: Rastgele bir mini-batch örnekleme yaparak eğitim sırasında kullanılmak üzere veri döndürür. done bayrağını tersine çevirerek not_done olarak kullanır.
- __len__ Metodu: Replay buffer'daki mevcut örnek sayısını döndürür.

Sonuç

ReplayBuffer sınıfı, SAC gibi algoritmalar için geçmiş deneyimlerin yeniden kullanılmasını sağlar ve öğrenme sürecini stabilize eder.

TRAİN.PY

train.py, SAC algoritmasını kullanarak ajanı Pusher-v4 ortamında eğitmek için gereken ana eğitim döngüsünü içerir. Bu dosya, deneyimleri toplamak, replaybuffer'ı doldurmak, model güncellemelerini yapmak ve belirli aralıklarla modeli kaydetmek için yapılandırılmıştır.

Eğitim Döngüsü: Eğitim num_episodes kadar bölümde gerçekleştirilir. Her bölüm için:

- Ajan, env.reset() ile ortama başlatılır.
- Ajanın ortamla etkileşiminden elde edilen her gözlem, aksiyon, ödül ve bir sonraki gözlem replay buffer'a kaydedilir.
- Ajan Güncellemesi: Replay buffer'da yeterli veri olduğunda agent.update() çağrılır ve ajan güncellenir.

- Eğitim İlerlemesinin Kaydedilmesi:

Eğitim ilerlemesi ve her bölümde elde edilen toplam ödül, training_log.txt dosyasına kaydedilir. Bu dosya, ajan performansının izlenmesi için kullanılabilir.

- Model Kaydetme: Belirli aralıklarla (save_interval bölümler) model checkpointsdizinine kaydedilir. Bu, eğitim sırasında modelin ara sürümlerinin saklanmasını sağlar ve gerektiğinde yeniden başlatılabilir.,

Sonuç ve Öneriler
train.py, ajanı eğitmek için gerekli tüm adımları içerir. Eğitim sırasında training_log.txt dosyasına kaydedilen performans metrikleri, ajan performansının değerlendirilmesinde faydalıdır. Eğitim süresini optimize etmek için num_episodes, batch_size, ve save_interval gibi parametreleri denemekuygun olabilir.

RENDER.PY

render.py, eğitim tamamlandıktan sonra modelin performansını görselleştirmek için tasarlanmıştır. Bu dosya, kaydedilmiş bir modeli yükler ve ajan ile Pusher-v4 ortamında etkileşime girer. env.render() fonksiyonu, ajan davranışının görsel çıktısını sağlar.

Ortam ve Ajan Kurulumu:
gymnasium kullanılarak Pusher-v4 ortamı başlatılır ve SACAgent sınıfı ile bir ajan tanımlanır. agent.load() çağrısı ile eğitimde elde edilen model parametreleri yüklenir. Bu yapı, ajanın eğitimde öğrendiği yetenekleri değerlendirmek için kullanılır.

Ajan Etkileşimi ve Görselleştirme:
Bu kısımda, ajan gözlem alır, aksiyon seçer ve ortamda adım atar. env.render() fonksiyonu, ajanın ortamda nasıl davrandığını görsel olarak izlemeye imkan tanır. render.py eğitim tamamlandıktan sonra ajanın performansını incelemek için önemlidir.

Sonuç ve Öneriler
render.py, eğitim sonrası ajan davranışlarını gözlemlemeye olanak tanır. Bu dosya, ajanın başarılı bir şekilde öğrendiği yetenekleri değerlendirmek için kullanılabilir. Modelin performansını test ederken ajanın aldığı ödülleri kaydetmek, görselleştirme sürecinde ilerlemesini daha detaylı anlamak için faydalı olabilir.

CONFİG.YAML

config.yaml dosyası, Pusher-v4 ortamında Soft Actor-Critic (SAC) algoritmasıyla eğitim gerçekleştirmek için gereken hiperparametreleri ve eğitim ayarlarını içerir. Bu dosya, ajan ve ortam arasındaki etkileşim sırasında kullanılacak yapılandırmaları belirler ve eğitim sürecini optimize etmeye olanak sağlar.

1. Ortam Ayarları (environment)
Bu bölüm, eğitim ortamı ve bazı temel eğitim parametrelerini içerir:

- name: "Pusher-v4", ajan için eğitim ortamı olarak gymnasium platformundaki Pusher-v4 ortamını belirtir.
- total_steps: Eğitim boyunca atılacak toplam adım sayısını belirtir.
- batch_size: Mini-batch boyutu olup, her güncelleme adımında replay buffer'danalınacak örnek sayısını belirtir.

- log_interval: Eğitim sırasında log'ların yazdırılma sıklığını (adım sayısı cinsinden) belirtir
- save_interval: Modelin belirli adım sayılarında kaydedilme sıklığını belirler.

- 2. SAC Ayarları

Bu bölüm, Soft Actor-Critic algoritmasına özgü hiperparametreleri tanımlar:

- gamma: İskonto oranı olup, ajan ödülleri hesaplarken gelecekteki ödüllerin etkisini azaltmak için kullanılır.
- tau: Yumuşak hedef güncelleme katsayısı olup, kritik ağın hedef ağ ile güncellenmesinde kullanılır
- alpha: Entropi katsayısıdır ve ajan politikasının rasgeleliğini kontrol eder.
- actor_lr ve critic_lr: Aktör ve kritik ağlar için öğrenme oranlarıdır.
- replay_buffer_size: Replay buffer'ın alabileceği maksimum örnek sayısını belirtir

train_pusher.py

```
import gymnasium as gym
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import ProgressBarCallback
import time

# Pusher ortamını oluşturun
env = gym.make('Pusher-v4', render_mode="rgb_array")

# Modeli başlatın ve buffer boyutunu ayarlayın
model = SAC('MlpPolicy', env, buffer_size=100000, verbose=1)

# İlerlemenin her adımda gösterilmesi için ProgressBarCallback kullanın
callback = ProgressBarCallback()

# Eğitim
print("Eğitim başlıyor...")
model.learn(total_timesteps=5000, callback=callback)  # Eğitim adım sayısını isteğinize
göre artırabilirsiniz

# Eğitim tamamlandığında modeli kaydedin
model.save("sac_pusher")
print("Eğitim tamamlandı ve model kaydedildi.")

# Modeli yükleyin
model = SAC.load("sac_pusher")

# Test döngüsü
print("Model test ediliyor...")
obs, _ = env.reset()

for i in range(100):
  action, _ = model.predict(obs, deterministic=True)
  obs, reward, done, _, _ = env.step(action)
  env.render()  # render() çağrısı görselleştirmeyi ekrana getirir
  time.sleep(0.01)  # Görselleştirmeyi yavaşlatmak için isteğe bağlı olarak ekleyin
  if done:
     obs, _ = env.reset()

env.close()
print("Test tamamlandı.")
```

Çıktı Değerlerinin Anlamları

1. **1.** **rollout/** - Bu bölüm, simülasyon sırasında toplanan gözlem verilerini temsil eder.
    - ○ **ep_len_mean**: Ortalama bölüm uzunluğu. İlk çıktıda 100, son çıktıda da 100 olarak aynı kalmış, yani her bölümün ortalama adım sayısı sabit.
    - ○ **ep_rew_mean**: Ortalama bölüm ödülü. İlk çıktıda -159 iken son çıktıda -139'a yükselmiş. Bu, ajan için ortalama ödülün arttığını, yani performansın iyileştiğini gösterir.
    - **2.** **time/** - Bu bölümde eğitim süresiyle ilgili bilgiler bulunur.
    - ○ **episodes**: Gerçekleşen bölüm sayısı. İlk çıktıda 4, son çıktıda 48 olmuş. Bu, eğitim sırasında daha fazla bölüm oynatıldığını gösteriyor.
    - ○ **fps**: Eğitim sırasında saniyede işlenen kare sayısı. FPS ilk çıktıda 72 iken, son çıktıda 55'e düşmüş. Bu, eğitim süresi arttıkça FPS'nin biraz yavaşladığını gösteriyor.
    - ○ **time_elapsed**: Geçen süre (saniye cinsinden). Eğitim boyunca toplam geçen süre ilk çıktıda 5 saniyeyken, son çıktıda 86 saniyeye çıkmış.
    - ○ **total_timesteps**: Ajanın aldığı toplam adım sayısı. İlk çıktıda 400 adım alınmışken, son çıktıda bu sayı 4800'e yükselmiş.
    - **3.** **train/** - Bu bölümde, eğitim sırasında ajanın parametrelerinde yapılan güncellemeler hakkında bilgiler yer alır.
    - ○ **actor_loss**: Ajanın politik (actor) kaybı. Bu değer, ajan eylemlerini nasıl seçtiğini optimize eder. İlk çıktıda -10.1, son çıktıda -18.1 olmuş. Ajanın seçtiği eylemlerle ilgili daha fazla deneyim topladığını gösterir.
    - ○ **critic_loss**: Ajanın değer fonksiyonu (critic) kaybı. İlk çıktıda 0.272, son çıktıda 0.0411’e düşmüş. Bu düşüş, modelin değer fonksiyonunun daha doğru tahminler yaptığını ve daha iyi öğrendiğini gösterir.
    - ○ **ent_coef**: Entropi katsayısı. Bu, ajanı keşif yapmaya teşvik eden katsayıdır. İlk çıktıda 0.914 iken, son çıktıda 0.246’ya düşmüş. Bu düşüş, ajan keşiften daha belirgin eylemlere geçiş yapmaya başlamış demektir.
    - ○ **ent_coef_loss**: Entropi katsayısı kaybı. Ajanın keşif eğilimini optimize eder. İlk çıktıda -1.05, son çıktıda -15.5 olmuş.
    - ○ **learning_rate**: Öğrenme oranı, 0.0003 olarak sabit kalmış.
    - ○ **n_updates**: Yapılan toplam güncelleme sayısı. İlk çıktıda 299, son çıktıda 4699’a yükselmiş, yani ajan daha fazla öğrenme döngüsü gerçekleştirmiş.

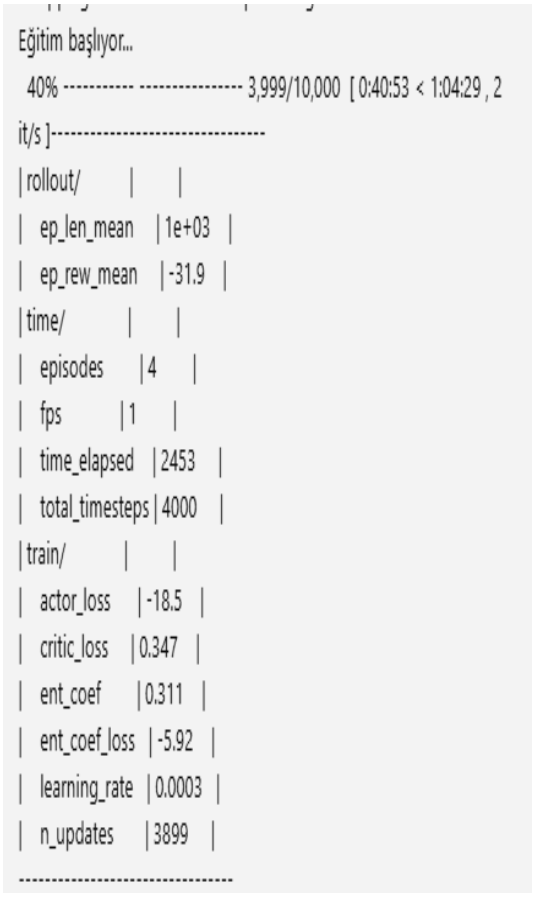
Gelişmeler ve Sonuç

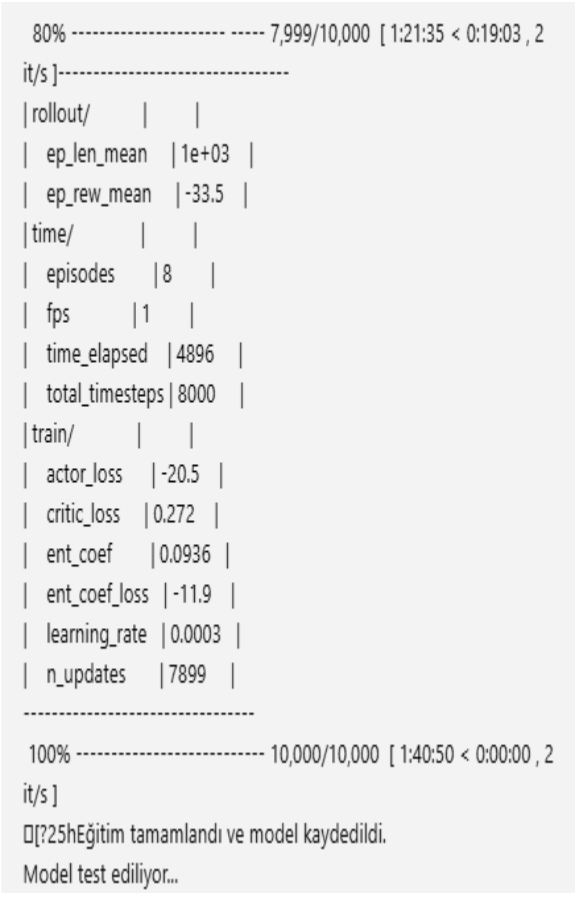
İlk ve son çıktılara bakıldığında, ajan ödül açısından performansını artırmış, daha uzun bir eğitim sürecinden geçmiş ve daha fazla güncelleme ile modelini iyileştirmiştir. Ortalama ödüldeki artış (-159’dan -139’a) ve critic kaybındaki azalma (0.272’den 0.0411’e) ajan performansının yükseldiğini gösterir. Ayrıca, entropi katsayısının düşmesi, ajan keşif aşamasını tamamlayıp daha belirgin eylemlere yönelmeye başladığını işaret eder.

Rapor Sonucu

Ajan, eğitim boyunca ödül toplama performansını artırmış ve simülasyon boyunca aldığı kararların doğruluğunu geliştirerek daha başarılı sonuçlara ulaşmıştır. Entropi katsayısının düşmesi, keşiften daha belirgin eylemlere yönelme eğilimini yansıtır. Eğitim sürecinde yapılan güncellemeler, ajanı optimize etmiş ve modelin doğruluğunu artırmıştır. Bu bulgular, ajanın çevreyi daha iyi öğrenmeye başladığını ve eğitim sürecinde performansının belirgin şekilde geliştiğini göstermektedir.

Car Racing Ortamında SAC Algoritmasının Uygulanması





Eğitim Sürecine Ait Metirkler

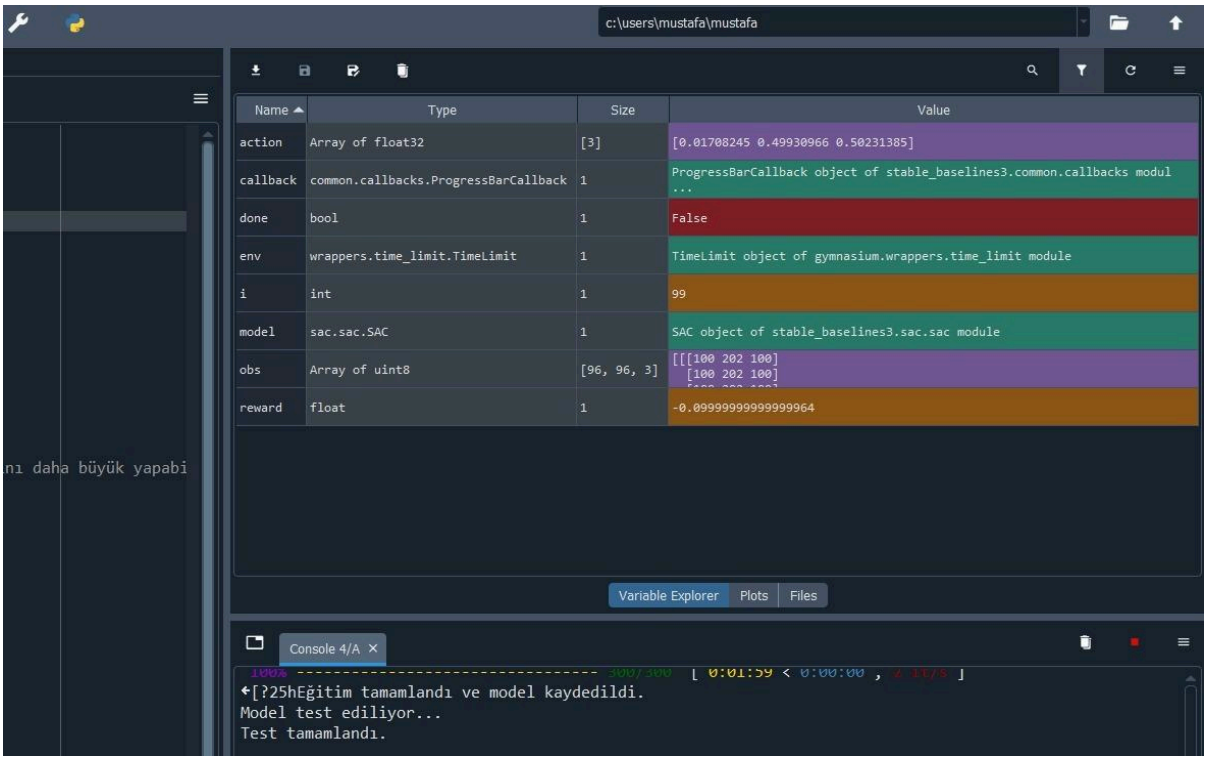
1. **rollout / ep_len_mean**: Ortalama bölüm uzunluğunu gösterir. `1e+03` değeri, her bölümün yaklaşık 1000 adım sürdüğünü gösteriyor. Bu, `CarRacing-v2` ortamının varsayılan zaman sınırı olduğundan, her bölümün maksimum uzunluğa ulaştığını gösteriyor.
2. **rollout / ep_rew_mean**: Ortalama bölüm ödülünü ifade eder. Örneğin, -31.9 veya -33.5 gibi değerler, modelin ortalama olarak her bölümde aldığı ödülü gösteriyor. Daha yüksek (pozitif) ödüller, genellikle daha iyi performansı gösterir.
3. **time / episodes**: Bu, eğitim sırasında kaç bölüm tamamlandığını belirtir. Örneğin, `episodes | 4` değeri, eğitim başladığından beri toplam 4 bölümün tamamlandığını gösterir.
4. **time / fps**: Bu, eğitimin ne kadar hızlı ilerlediğini gösterir. `fps | 1` değeri, saniyede yaklaşık bir adım işlendiğini belirtir, yani eğitim oldukça yavaş ilerliyor. Bu, modelin CPU'da çalışmasından kaynaklanıyor olabilir; GPU kullanımı, bu değeri artırarak daha hızlı eğitim sağlar.
5. **time / time_elapsed**: Eğitim başladığından beri geçen süreyi (saniye olarak) gösterir. Örneğin, `2453` değeri, eğitimde geçen sürenin yaklaşık 2453 saniye olduğunu ifade eder.
6. **time / total_timesteps**: Şu ana kadar gerçekleştirilen toplam adım sayısını gösterir. Örneğin, `total_timesteps | 4000`, modelin şu ana kadar 4000 adımlık bir eğitim gördüğünü gösterir.

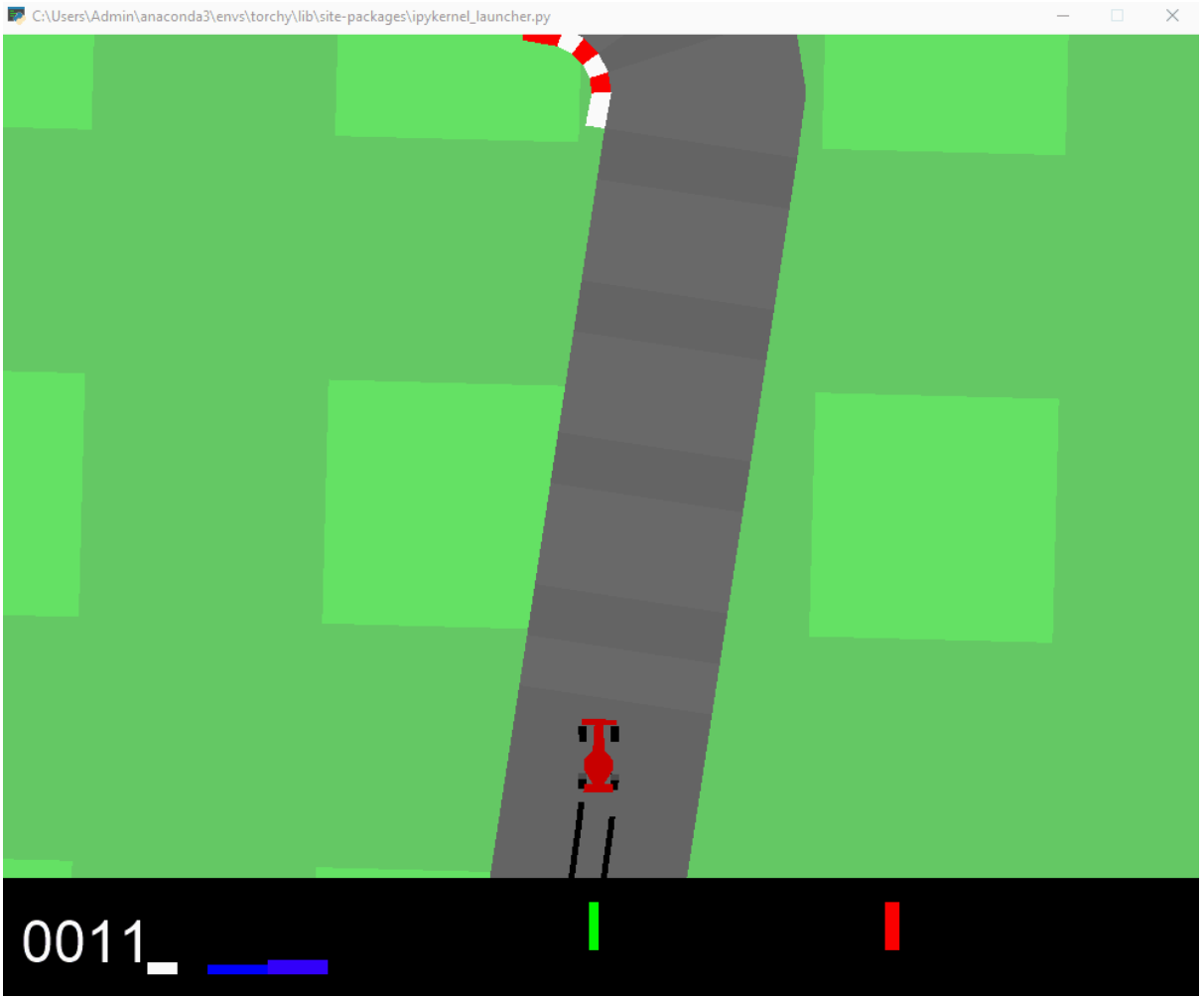
Eğitim Metirkleri

7. **train / actor_loss**: Aktör ağırlığının kaybını ifade eder. Bu değer negatifse, bu genellikle aktörün daha iyi politikalar öğrenmesi için yapılan ayarlamaları gösterir. Negatif kayıp değerleri, politikanın ödülleri artırmaya çalıştığını gösterir.
8. **train / critic_loss**: Eleştirmen ağırlığının kaybını ifade eder. Kritik kayıp değeri, eleştirmenin ödülleri doğru tahmin etme yeteneğini gösterir. Daha düşük bir değer, modelin daha isabetli ödül tahminleri yaptığı anlamına gelir.
9. **train / ent_coef**: Entropi katsayısını gösterir. Entropi, politikada keşif derecesini ifade eder; bu nedenle daha yüksek bir entropi değeri, modelin farklı eylemler denemeye daha açık olduğunu gösterir.
10. **train / ent_coef_loss**: Entropi katsayısının kaybını gösterir. Entropi kaybı, modelin keşfetme ve sömürü dengesini optimize etmeye çalıştığı anlamına gelir.
11. **train / learning_rate**: Öğrenme oranını gösterir. Bu, modelin parametrelerini güncellerken yaptığı değişikliklerin hızını ifade eder.
12. **train / n_updates**: Modelin ağırlıklarının kaç kez güncellendiğini gösterir. Bu değer arttıkça, model daha fazla eğitim görmüş olur.

Özet

- **Eğitim Performansı**: Ortalama ödül (`ep_rew_mean`) değeri negatif, yani model henüz oldukça düşük bir performansa sahip. Bu durum, `CarRacing` ortamının zorluğundan kaynaklanabilir.
- **Eğitim Hızı**: Eğitim sürecinin CPU üzerinde çalışması nedeniyle yavaş ilerliyor (`fps | 1`).
- **Modelin Güncellemeleri**: Model her adımda güncellemeler yaparak ödülleri daha doğru tahmin etmeye ve daha iyi eylemler seçmeye çalışıyor.





KAYNAK KODU

```
import gymnasium as gym
from stable_baselines3 import SAC
from stable_baselines3.common.callbacks import ProgressBarCallback
import time

# CarRacing ortamını render_mode ile oluşturun
env = gym.make('CarRacing-v2', render_mode="human")

# Modeli başlatın ve buffer boyutunu ayarlayın
model = SAC('CnnPolicy', env, buffer_size=100000, verbose=1)

# İlerlemenin her adımda gösterilmesi için ProgressBarCallback kullanın
callback = ProgressBarCallback()
```

```
# Eğitim
print("Eğitim başlıyor...")
model.learn(total_timesteps=300, callback=callback)  # Eğitim adım sayısını daha büyük yapabilirsiniz

# Eğitim tamamlandığında modeli kaydedin
model.save("sac_car_racing")
print("Eğitim tamamlandı ve model kaydedildi.")

# Modeli yükleyin
model = SAC.load("sac_car_racing")

# Test döngüsü
print("Model test ediliyor...")
obs, _ = env.reset()

for i in range(100):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, _, _ = env.step(action)
        env.render()  # render() çağrısı görselleştirmeyi ekrana getirir
        time.sleep(0.01)  # Görselleştirmeyi yavaşlatmak için isteğe bağlı olarak ekleyin
        if done:
        obs, _ = env.reset()

env.close()
print("Test tamamlandı.")
```