DQN

usando torchrl

BRUNO MENEZES

DQN
usando torchrl

BRUNO MENEZES

Sobre o autor

Bruno Menezes é Engenheiro Eletricista, formado na Universidade Federal do Ceará, em 2014. Tem especialização em Gerenciamento de Projetos, Engenharia de Segurança do Trabalho e Segurança da Informação. Além de seu trabalho como Engenheiro e pesquisador na área de Inteligência Artificial (IA), é Técnico em Logística de Transporte (TLT) na Petrobras. Compartilha seus conhecimentos no GitHub, Youtube e publicações do LinkedIn, com o objetivo de disseminar informações sobre Aprendizado por Reforço Profundo (ARP) e promover o conhecimento prático de Aprendizado de Máquina na comunidade. Seu compromisso é ajudar os outros a dominar o uso de IA.

Índice

Observação!

Para executar este tutorial em um notebook, adicione uma célula de instalação no início contendo:

```
!pip install tensordict
!pip install torchrl
```

Usamos o DQN com um ambiente *CartPole* como exemplo prototípico.

Capítulo 1 - Construindo o ambiente

Vamos usar um ambiente de simulação do Gym com uma transformação *StepCounter* .

```
import torch

torch.manual_seed(0)

import time

from torchrl.envs import GymEnv, StepCounter, TransformedEnv

env = TransformedEnv(GymEnv("CartPole-v1"), StepCounter())
env.set_seed(0)

from tensordict.nn import TensorDictModule as Mod, TensorDictSequential as Seq
```

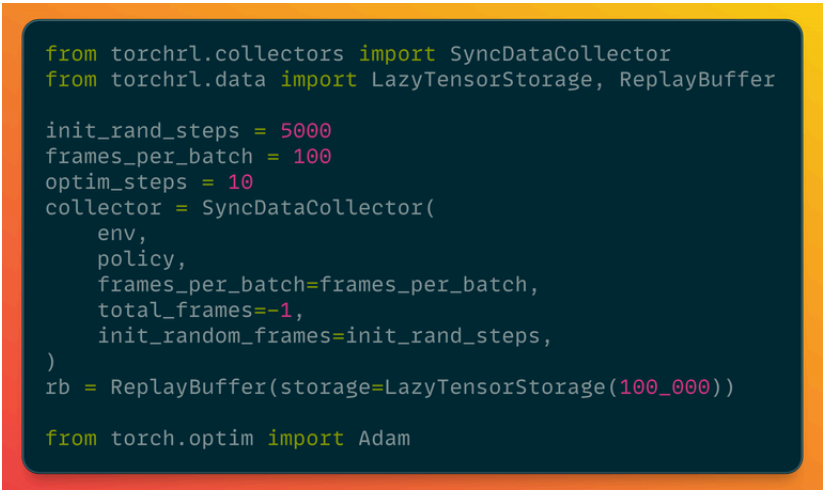
Capítulo 2 - Projetando uma política

O próximo passo é construir nossa política. Faremos uma versão regular e determinística do ator para ser usada no módulo de perda e durante a avaliação . A seguir, iremos aumentá-lo com um módulo de exploração para inferência .

```
from torchrl.modules import EGreedyModule, MLP, QValueModule

value_mlp = MLP(out_features=env.action_spec.shape[-1], num_cells=[64, 64])
value_net = Mod(value_mlp, in_keys=["observation"], out_keys=["action_value"])
policy = Seq(value_net, QValueModule(spec=env.action_spec))
exploration_module = EGreedyModule(
    env.action_spec, annealing_num_steps=100_000, eps_init=0.5
)
policy_explore = Seq(policy, exploration_module)
```

Capítulo 3 - Coletor de dados e buffer de reprodução

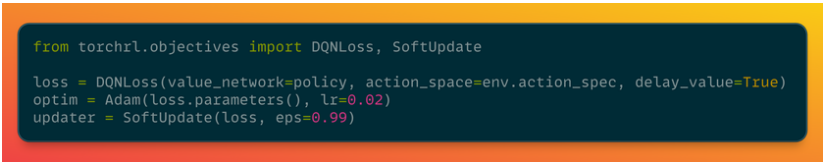
Aí vem a parte dos dados: precisamos de um coletor de dados para obter facilmente lotes de dados e um buffer de repetição para armazenar esses dados para treinamento.



```
from torchrl.collectors import SyncDataCollector
from torchrl.data import LazyTensorStorage, ReplayBuffer

init_rand_steps = 5000
frames_per_batch = 100
optim_steps = 10
collector = SyncDataCollector(
    env,
    policy,
    frames_per_batch=frames_per_batch,
    total_frames=-1,
    init_random_frames=init_rand_steps,
)
rb = ReplayBuffer(storage=LazyTensorStorage(100_000))

from torch.optim import Adam
```

Capítulo 4 - Módulo de perda e otimizador

Construímos nossa perda baseada no DQN, com otimizador e atualizador de parâmetros de rede alvo conforme abaixo:



```
from torchrl.objectives import DQNLoss, SoftUpdate

loss = DQNLoss(value_network=policy, action_space=env.action_spec, delay_value=True)
optim = Adam(loss.parameters(), lr=0.02)
updater = SoftUpdate(loss, eps=0.99)
```

Capítulo 5 - Registrador

Usaremos um registrador CSV para registrar nossos resultados e salvar vídeos renderizados.

```
from torchrl._utils import logger as torchrl_logger
from torchrl.record import CSVLogger, VideoRecorder

path = "./training_loop"
logger = CSVLogger(exp_name="dqn", log_dir=path, video_format="mp4")
video_recorder = VideoRecorder(logger, tag="video")
record_env = TransformedEnv(
    GymEnv("CartPole-v1", from_pixels=True, pixels_only=False), video_recorder
)
```

Capítulo 6 - Ciclo de treinamento

Em vez de fixar um número específico de iterações a serem executadas, continuaremos treinando a rede até que ela atinja um determinado desempenho (definido arbitrariamente como 500 etapas no ambiente – com *CartPole*, sucesso é definido como ter trajetórias mais longas).

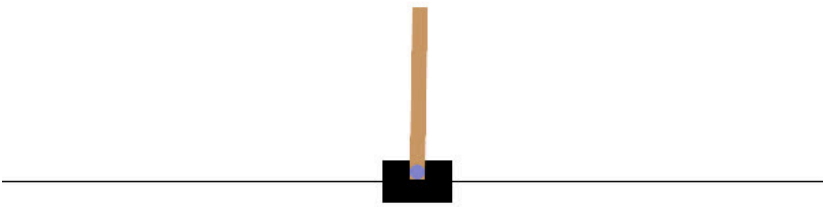
```
total_count = 0
total_episodes = 0
t0 = time.time()
for i, data in enumerate(collector):
    # Write data in replay buffer
    rb.extend(data)
    max_length = rb[:]["next", "step_count"].max()
    if len(rb) > init_rand_steps:
        # Optim loop (we do several optim steps
        # per batch collected for efficiency)
        for _ in range(optim_steps):
            sample = rb.sample(128)
            loss_vals = loss(sample)
            loss_vals["loss"].backward()
            optim.step()
            optim.zero_grad()
            # Update exploration factor
            exploration_module.step(data.numel())
            # Update target params
            updater.step()
            if i % 10:
                torchrl_logger.info(f"Max num steps: {max_length}, rb length {len(rb)}")
            total_count += data.numel()
            total_episodes += data["next", "done"].sum()
    if max_length > 500:
        break

t1 = time.time()

torchrl_logger.info(
    f"solved after {total_count} steps, {total_episodes} episodes and in {t1-t0}s."
)
```

Capítulo 7 - Renderização

Por fim, executamos o ambiente em tantas etapas quanto possível e salvamos o vídeo localmente (observe que não estamos explorando).

```
record_env.rollout(max_steps=1000, policy=policy)
video_recorder.dump()
```

Esta é a aparência do seu vídeo CartPole renderizado após um ciclo de treinamento completo:



BRUNO MENEZES

@bruno_f.menezes