TorchRL Framework Tutorial

Guven Gergerli

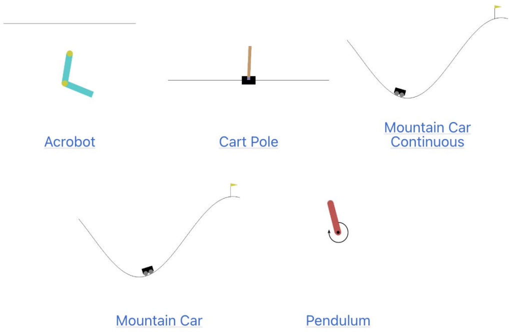
TorchRL

- Reinforcement Learning (RL) library for PyTorch
  - High modularity
  - Few dependencies
  - Aimed at supporting research in RL
- Developed by Meta

```
$ pip install torchrl
```

Environments

- Reinforcement Learning training loops involves a model -> π(a,s)
  - Trained to achieve a given goal via rewards. -> $R_a(s,s')$
- Often, this environment is a simulator that accepts actions as input and produces an observation along with some metadata as output.
- There are many environment frameworks
  - Gymnasium
  - DeepMind Lab
  - …
- We will use gymnasium (previously as gym)
- Using a wrapper for gymnasium




```python
from torchrl.envs import GymEnv

env = GymEnv("Pendulum-v1")
```

TEDs (TorchRL Episode Data format)

- Environments function over `reset()` and `step()` through agent action.
- Normally gym environments returns *observation*, *reward*, and *done info*.
- But in TorchRL, the environments read and write `tensordict`.
    - Tensordict: “generic key-based data carrier for tensors”
    - This is the novelty of TorchRL.

- Let’s see what it looks like
  (collab part 1)

```
TensorDict(
    fields={
        action: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
        done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
        next: TensorDict(
            fields={
                done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
                observation: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
                reward: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
                terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
                truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False)},
            batch_size=torch.Size([]),
            device=None,
            is_shared=False),
        observation: Tensor(shape=torch.Size([3]), device=cpu, dtype=torch.float32, is_shared=False),
        terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False)},
    batch_size=torch.Size([]),
    device=None,
    is_shared=False)
```

Transforming and Environment

- This modularity gives us the flexibility to modify the environment
- Example: step_counter
  - Adds another entry that tracks the number of steps since the last reset.

- Here is the example:
  (collab part 2)

```python
from torchrl.envs import StepCounter, TransformedEnv

transformed_env = TransformedEnv(env, StepCounter(max_steps=10))
rollout = transformed_env.rollout(max_steps=100)
```

```python
print(rollout["next", "step_count"])
```

```
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4],
        [ 5],
        [ 6],
        [ 7],
        [ 8],
        [ 9],
        [10]])
```

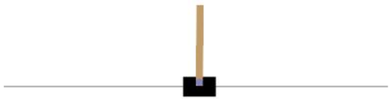
TorchRL Modules

- TorchRL is modular
  - Just like transforming tensordict, you can transform policies.
- A module with input and output can become a policy.
- TorchRL provides wrappers: `Actor`, `ProbabilisticActor`, `ActorValueOperator` or `ActorCriticOperator`
- TorchRL provides networks: `MLP`, `ConvNet`, or `LSTMModule`
- We even can use `EGreedyModule` for exploration

- Lets cover these in the code: (collab part 3)

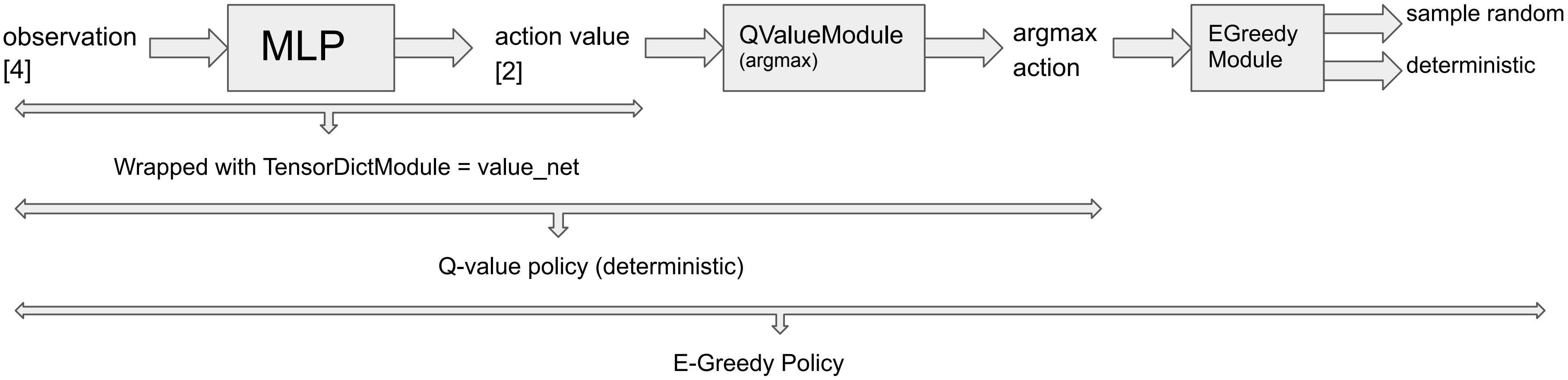
(collab part 4)

Bringing It All Together: Q-Value actors



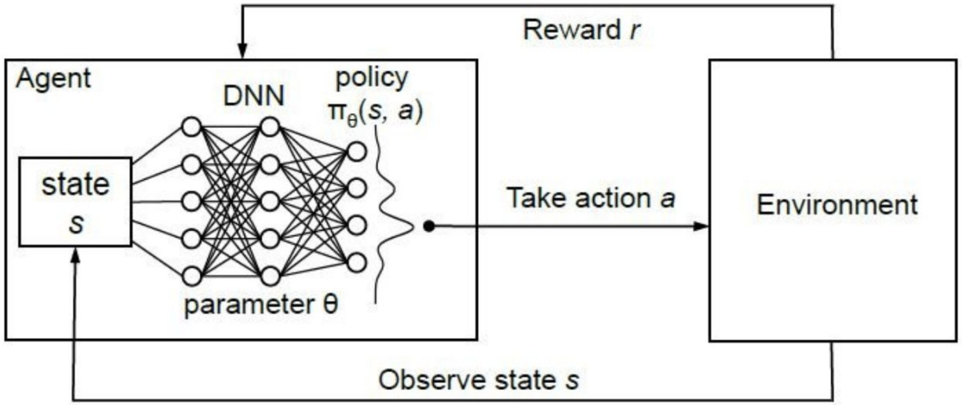
Environment = GymEnv(“Cartpole-v1”)

Action Space	`Discrete(2)`
Observation Space	`Box([-4.8 -inf -0.41887903 -inf], [4.8 inf 0.41887903 inf], (4,), float32)`
import	`gymnasium.make("CartPole-v1")`



Model Optimization

- We know how to create agents now, but how about training an agent?
- What value function predicts: return/target; $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$
- In Value-based methods (DQN) we approximate Q(s,a) (MSE between pred Q value and the target)
  - $L(\theta) = (Q(s_t,a_t;\theta)-y)2$
- In policy gradient methods, maximize expected reward (negative expected return)
  - $L(\theta) = -E_{\tau\sim\pi\theta}[\sum_t log\pi_\theta(a_t \mid s_t)G^{\wedge}_t]$



Model Optimization

- In torch we use the `loss.backward()` followed with `optimizer.step()`
- Back to TorchRL: dedicated loss modules for optimizing a model
- Example: off-policy DDPG algorithm
  - We need a policy defined as TensorDictModule:
    - our policy will be a value network for predicting the value of state-action pair.
  - DDPG loss will find the policy parameters that output actions that maximize the value for a given state
    - so we will feed the actor and the value net.
  - Then we will train the LossModule.
  - Bonus: Soft update

- (collab part 5)

Data Collection and Replay Buffers

- Data = learning
  - We need to collect and/or store data for training
- Data collectors
  - Role: execute policy within the environment, reset when necessary, and provide batches
  - Requires: size of the batches, length of the iterator, policy, and environment

```
>>> for data in collector:
...     # your algorithm here
```

- Replay Buffers
  - Role: Store temporary data for training
  - Variables: storage type, sampling technique, writing heuristic, transforms…

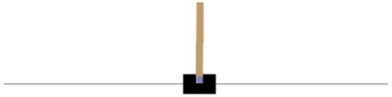
```
>>> for data in collector:
...     storage.store(data)
...     for i in range(n_optim):
...         sample = storage.sample()
...         loss_val = loss_fn(sample)
...         loss_val.backward()
...         optim.step() # etc
```

- (collab part 6)

Logging

- Required to check your algorithms and compile results
  - CSVLogger
  - TensorBoard
  - Wandb
  - …
- Important: Watch your agents
  - Plots might look good but, reward hacking? efficiency?
- (collab part 7)

Training Loop - DQN



- (collab tutorial 2)

Transformed Environment = GymEnv(“Cartpole-v1”) + StepCounter + VideoRecorder