05b

January 23, 2024

[1]:
```
%pylab inline
import torch
import sys, os
import pystk
import ray
device = torch.device('cuda') if torch.cuda.is_available() else torch.
↪device('cpu')
print('device = ', device)
ray.init(logging_level=50)
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
device =  cuda
```

[1]:
```
RayContext(dashboard_url='', python_version='3.10.13', ray_version='2.9.1',
ray_commit='cfbf98c315cfb2710c56039a3c96477d196de049', protocol_version=None)
```

```
(raylet) [2024-01-23 09:44:28,547 E 7315 7315] (raylet)
node_manager.cc:3022: 2 Workers (tasks / actors) killed due to memory pressure
(OOM), 0 Workers crashed due to other reasons at node (ID:
c390692744c464f918a4ee414045f7d2158877bae42e6c8503ee03be, IP: 10.64.35.146) over
the last time period. To see more information about the Workers killed on this
node, use `ray logs raylet.out -ip 10.64.35.146`
(raylet)
(raylet) Refer to the documentation on how to address the out of memory
issue: https://docs.ray.io/en/latest/ray-core/scheduling/ray-oom-
prevention.html. Consider provisioning more memory on this node or reducing task
parallelism by requesting more CPUs per task. To adjust the kill threshold, set
the environment variable `RAY_memory_usage_threshold` when starting Ray. To
disable worker killing, set the environment variable
`RAY_memory_monitor_refresh_ms` to zero.
```

[2]:
```
@ray.remote
class Rollout:
    def __init__(self, screen_width, screen_height, hd=True,␣
↪track='lighthouse', render=True, frame_skip=1):
        # Init supertuxkart
        if not render:
```

```python
            config = pystk.GraphicsConfig.none()
        elif hd:
            config = pystk.GraphicsConfig.hd()
        else:
            config = pystk.GraphicsConfig.ld()
        config.screen_width = screen_width
        config.screen_height = screen_height
        pystk.init(config)

        self.frame_skip = frame_skip
        self.render = render
        race_config = pystk.RaceConfig(track=track)
        self.race = pystk.Race(race_config)
        self.race.start()

    def __call__(self, agent, n_steps=200):
        torch.set_num_threads(1)
        self.race.restart()
        self.race.step()
        data = []
        track_info = pystk.Track()
        track_info.update()

        for i in range(n_steps // self.frame_skip):
            world_info = pystk.WorldState()
            world_info.update()

            # Gather world information
            kart_info = world_info.players[0].kart

            agent_data = {'track_info': track_info, 'kart_info': kart_info}
            if self.render:
                agent_data['image'] = np.array(self.race.render_data[0].image)

            # Act
            action = agent(**agent_data)
            agent_data['action'] = action

            # Take a step in the simulation
            for it in range(self.frame_skip):
                self.race.step(action)

            # Save all the relevant data
            data.append(agent_data)
        return data

def show_video(frames, fps=30):
```

```
    import imageio
    from IPython.display import Video, display

    imageio.mimwrite('/tmp/test.mp4', frames, fps=fps, bitrate=1000000)
    display(Video('/tmp/test.mp4', width=800, height=600, embed=True))

viz_rollout = Rollout.remote(400, 300)
def show_agent(agent, n_steps=600):
    data = ray.get(viz_rollout.__call__.remote(agent, n_steps=n_steps))
    show_video([d['image'] for d in data])

rollouts = [Rollout.remote(50, 50, hd=False, render=False, frame_skip=5) for i␣
 ↪in range(10)]
def rollout_many(many_agents, **kwargs):
    ray_data = []
    for i, agent in enumerate(many_agents):
         ray_data.append( rollouts[i % len(rollouts)].__call__.remote(agent,␣
 ↪**kwargs) )
    return ray.get(ray_data)

def dummy_agent(**kwargs):
    action = pystk.Action()
    action.acceleration = 1
    return action
```

[3]:

```
def three_points_on_track(distance, track):
    distance = np.clip(distance, track.path_distance[0,0], track.
 ↪path_distance[-1,1]).astype(np.float32)
    valid_node = (track.path_distance[..., 0] <= distance) & (distance <= track.
 ↪path_distance[..., 1])
    valid_node_idx, = np.where(valid_node)
    node_idx = valid_node_idx[0] # np.random.choice(valid_node_idx)
    d = track.path_distance[node_idx].astype(np.float32)
    x = track.path_nodes[node_idx][:,[0,2]].astype(np.float32) # Ignore the y␣
 ↪coordinate
    w, = track.path_width[node_idx].astype(np.float32)

    t = (distance - d[0]) / (d[1] - d[0])
    mid = x[1] * t + x[0] * (1 - t)
    x10 = (x[1] - x[0]) / np.linalg.norm(x[1]-x[0])
    x10_ortho = np.array([-x10[1],x10[0]], dtype=float32)
    return mid - w / 2 * x10_ortho, mid, mid + w / 2 * x10_ortho


def state_features(track_info, kart_info, absolute=False, **kwargs):
    f = np.concatenate([three_points_on_track(kart_info.distance_down_track +␣
 ↪d, track_info) for d in [0,5,10,15,20]])
```

```
    if absolute:
        return f
    p = np.array(kart_info.location)[[0,2]].astype(np.float32)
    t = np.array(kart_info.front)[[0,2]].astype(np.float32)
    f = f - p[None]
    d = (p-t) / np.linalg.norm(p-t)
    d_o = np.array([-d[1], d[0]], dtype=float32)
    return np.stack([f.dot(d), f.dot(d_o)], axis=1)

# Let's load a fancy auto-pilot. You'll write one yourself in your homework.
from _auto_pilot import auto_pilot
data, = rollout_many([auto_pilot], n_steps=400)

figure()
f = state_features(**data[50])
plot(f[:,1].flat, f[:,0].flat, '*')
axis('equal')
gca().invert_yaxis()

figure()
for d in data:
    f = state_features(**d, absolute=True)
    plot(f[:,1].flat, f[:,0].flat, '*')
axis('equal')
gca().invert_yaxis()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[3], line 29
     26     return np.stack([f.dot(d), f.dot(d_o)], axis=1)
     28 # Let's load a fancy auto-pilot. You'll write one yourself in your␣
  ↪homework.
---> 29 from _auto_pilot import auto_pilot
     30 data, = rollout_many([auto_pilot], n_steps=400)
     32 figure()

ModuleNotFoundError: No module named '_auto_pilot'
```

[4]:
```
from torch.distributions import Bernoulli

def new_action_net():
    return torch.nn.Linear(2*5*3, 1, bias=False)

class Actor:
    def __init__(self, action_net):
        self.action_net = action_net.cpu().eval()
```

```
    def __call__(self, track_info, kart_info, **kwargs):
        f = state_features(track_info, kart_info)
        output = self.action_net(torch.as_tensor(f).view(1,-1))[0]

        action = pystk.Action()
        action.acceleration = 1
        steer_dist = Bernoulli(logits=output[0])
        action.steer = steer_dist.sample()*2-1
        return action

class GreedyActor:
    def __init__(self, action_net):
        self.action_net = action_net.cpu().eval()

    def __call__(self, track_info, kart_info, **kwargs):
        f = state_features(track_info, kart_info)
        output = self.action_net(torch.as_tensor(f).view(1,-1))[0]

        action = pystk.Action()
        action.acceleration = 1
        action.steer = output[0]
        return action
```

[5]:
```
action_net = new_action_net()
show_agent(Actor(action_net))
```

```
IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by
macro_block_size=16, resizing from (400, 300) to (400, 304) to ensure video
compatibility with most codecs and players. To prevent resizing, make your input
image divisible by the macro_block_size or set the macro_block_size to 1
(risking incompatibility).
```

```
<IPython.core.display.Video object>
```

[ ]:
```
many_action_nets = [new_action_net() for i in range(10)]

data = rollout_many([Actor(action_net) for action_net in many_action_nets],␣
 ↪n_steps=600)

good_initialization = many_action_nets[ np.argmax([d[-1]['kart_info'].
 ↪overall_distance for d in data]) ]
```

[ ]:
```
show_agent(Actor(good_initialization))
```

Recall what we're trying to do in RL: maximize the expected return of a policy π (or in turn minmize a los L)

$$-L = E_{\tau\sim P_\pi}[R(\tau)],$$

where $\tau = \{s_0, a_0, s_1, a_1, ...\}$ is a trajectory of states and actions. The return of a trajectory is then defined as the sum of individual rewards $R(\tau) = \sum_k r(s_k)$ (we won't discount in this assignment).

Policy gradient computes the gradient of the loss L using the log-derivative trick

$$\nabla_\pi L = -E_{\tau\sim P_\pi}[\sum_k r(s_k)\nabla_\pi \sum_i \log \pi(a_i|s_i)].$$

Since the return $r(s_k)$ only depends on action a_i in the past $i < k$ we can further simplify the above equation:

$$\nabla_\pi L = -E_{\tau\sim P_\pi}\left[\sum_i (\nabla_\pi \log \pi(a_i|s_i))\left(\sum_{k=i}^{i+T} r(s_k)\right)\right].$$

We will implement an estimator for this objective below. There are a few steps that we need to follow:

- The expectation $E_{\tau\sim P_\pi}$ are rollouts of our policy
- The log probability $\log \pi(a_i|s_i)$ uses the `Categorical.log_prob`
- Gradient computation uses the `.backward()` function
- The gradient $\nabla_\pi L$ is then used in a standard optimizer

[ ]:
```
import copy

n_epochs = 20
n_trajectories = 10
n_iterations =50
batch_size = 128
T = 20

action_net = copy.deepcopy(good_initialization)

best_action_net = copy.deepcopy(action_net)

optim = torch.optim.Adam(action_net.parameters(), lr=1e-3)


for epoch in range(n_epochs):
    eps = 1e-2

    # Roll out the policy, compute the Expectation
    trajectories = rollout_many([Actor(action_net)]*n_trajectories, n_steps=600)
    print('epoch = %d   best_dist = '%epoch, np.max([t[-1]['kart_info'].
 ↪overall_distance for t in trajectories]))

    # Compute all the reqired quantities to update the policy
    features = []
    returns = []
    actions = []
    for trajectory in trajectories:
        for i in range(len(trajectory)):
```

```
            # Compute the returns
            returns.append( trajectory[min(i+T,␣
↪len(trajectory)-1)]['kart_info'].overall_distance -
                                trajectory[i]['kart_info'].overall_distance )
            # Compute the features
            features.append( torch.as_tensor(state_features(**trajectory[i]),␣
↪dtype=torch.float32).cuda().view(-1) )
            # Store the action that we took
            actions.append( trajectory[i]['action'].steer > 0 )

    # Upload everything to the GPU
    returns = torch.as_tensor(returns, dtype=torch.float32).cuda()
    actions = torch.as_tensor(actions, dtype=torch.float32).cuda()
    features = torch.stack(features).cuda()

    returns = (returns - returns.mean()) / returns.std()

    action_net.train().cuda()
    avg_expected_log_return = []
    for it in range(n_iterations):
        batch_ids = torch.randint(0, len(returns), (batch_size,), device=device)
        batch_returns = returns[batch_ids]
        batch_actions = actions[batch_ids]
        batch_features = features[batch_ids]

        output = action_net(batch_features)
        pi = Bernoulli(logits=output[:,0])

        expected_log_return = (pi.log_prob(batch_actions)*batch_returns).mean()
        optim.zero_grad()
        (-expected_log_return).backward()
        optim.step()
        avg_expected_log_return.append(float(expected_log_return))

    best_performance, current_performance =␣
↪rollout_many([GreedyActor(best_action_net), GreedyActor(action_net)],␣
↪n_steps=600)
    if best_performance[-1]['kart_info'].overall_distance <␣
↪current_performance[-1]['kart_info'].overall_distance:
        best_action_net = copy.deepcopy(action_net)
```

[ ]:
```
show_agent(GreedyActor(best_action_net))
```

[ ]:
```
```