```
1 # ppo_pytorch_gymnasium.py
2 import gymnasium as gym
 3 import numpy as np
4 import torch
5 import torch.nn as nn
 6 from torch.distributions import Categorical
7 from collections import deque
9 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
10
11 class ActorCritic(nn.Module):
      def __init__(self, obs_dim, act_dim, hidden=64):
12
           super().__init__()
14
           self.shared = nn.Sequential(
15
              nn.Linear(obs_dim, hidden),
16
              nn.ReLU(),
              nn.Linear(hidden, hidden),
17
18
              nn.ReLU()
19
          )
20
          self.policy = nn.Linear(hidden, act dim)
21
           self.value = nn.Linear(hidden, 1)
22
      def forward(self, x):
23
24
          h = self.shared(x)
25
          logits = self.policy(h)
26
           value = self.value(h).squeeze(-1)
27
          return logits, value
28
29
30 def compute_gae(rewards, masks, values, next_value, gamma=0.99, lam=0.95):
31
      values = np.append(values, next value)
      gae = 0
32
33
      returns = []
34
      for step in reversed(range(len(rewards))):
35
          delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
36
           gae = delta + gamma * lam * masks[step] * gae
          returns.insert(0, gae + values[step])
37
38
      return np.array(returns)
39
40
41 def ppo_update(model, optimizer, obs, actions, log_probs_old, returns, advantages,
42
                 clip_epsilon=0.2, vf_coef=0.5, ent_coef=0.01, epochs=4, batch_size=64):
43
      dataset_size = obs.shape[0]
      for _ in range(epochs):
44
          indices = np.arange(dataset_size)
45
46
           np.random.shuffle(indices)
47
           for start in range(0, dataset_size, batch_size):
48
              batch_idx = indices[start:start+batch_size]
49
              b_obs = torch.tensor(obs[batch_idx], dtype=torch.float32, device=device)
50
              b_actions = torch.tensor(actions[batch_idx], device=device)
51
              b_old_logp = torch.tensor(log_probs_old[batch_idx], device=device)
52
              b_returns = torch.tensor(returns[batch_idx], dtype=torch.float32, device=device)
              b adv = torch.tensor(advantages[batch idx], dtype=torch.float32, device=device)
53
54
55
              logits, values = model(b_obs)
56
              dist = Categorical(logits=logits)
57
              entropy = dist.entropy().mean()
58
              new_logp = dist.log_prob(b_actions)
59
60
              ratio = torch.exp(new_logp - b_old_logp)
61
              surr1 = ratio * b_adv
              surr2 = torch.clamp(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * b_adv
62
63
              policy_loss = -torch.min(surr1, surr2).mean()
64
65
              value_loss = ((b_returns - values) ** 2).mean()
              loss = policy_loss + vf_coef * value_loss - ent_coef * entropy
66
67
68
              optimizer.zero grad()
69
              loss.backward()
70
              nn.utils.clip_grad_norm_(model.parameters(), 0.5)
               optimizer.step()
71
72
74 def train(env_name='CartPole-v1', total_timesteps=200, rollout_len=2048, lr=3e-4):
      env = gym.make(env name)
75
       obs_dim = env.observation_space.shape[0]
```

```
77
        act_dim = env.action_space.n
 78
 79
       model = ActorCritic(obs_dim, act_dim).to(device)
       optimizer = torch.optim.Adam(model.parameters(), lr=lr)
80
       obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [], [], [], []
 82
 83
       obs, _ = env.reset(seed=42)
84
85
       ep_rewards = deque(maxlen=100)
 86
       timestep = 0
 87
 88
       while timestep < total_timesteps:</pre>
 89
            for _ in range(rollout_len):
                obs\_tensor = torch.tensor(obs, \ dtype=torch.float32, \ device=device).unsqueeze(0)
90
                logits, value = model(obs_tensor)
 91
 92
                dist = Categorical(logits=logits)
                action = dist.sample().item()
 93
 94
                logp = dist.log_prob(torch.tensor(action, device=device)).item()
 95
                next_obs, reward, terminated, truncated, _ = env.step(action)
 96
97
                done = terminated or truncated
98
99
                obs_buf.append(obs.copy())
100
                action_buf.append(action)
101
                reward_buf.append(reward)
102
                done_buf.append(0.0 if done else 1.0)
103
                value_buf.append(value.item())
104
                logp_buf.append(logp)
105
               obs = next_obs
106
107
                timestep += 1
108
                if done:
109
                    obs, _ = env.reset()
110
111
           obs_tensor = torch.tensor(obs, dtype=torch.float32, device=device).unsqueeze(0)
112
            _, next_value = model(obs_tensor)
           next_value = next_value.item()
113
114
           returns = compute_gae(reward_buf, done_buf, value_buf, next_value)
115
116
           advantages = returns - np.array(value buf)
117
           advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)
118
119
           ppo_update(model, optimizer,
120
                       np.array(obs_buf), np.array(action_buf), np.array(logp_buf),
121
                       returns, advantages)
122
123
           total_reward = sum(reward_buf[-rollout_len:])
124
            ep_rewards.append(total_reward)
125
126
           print(f"Timesteps: {timestep}\tAverage Rollout Reward: {np.mean(ep_rewards):.2f}")
127
           obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [], [], [], []
128
129
130
       env.close()
131
       return model
132
133
134 if __name__ == "__main__":
135
        train()
136
```

https://colab.research.google.com/drive/1aKzpFZouyn54SxU47b39UMNH5a9FKrp1#scrollTo=ADlukhsERtPk&printMode=true

Timesteps: 2048 Average Rollout Reward: 2048.00

```
1 # ppo_tensorflow_gymnasium.py
2 import gymnasium as gym
    import numpy as np
    import tensorflow as tf
    from tensorflow.keras import layers, Model, optimizers
7
    class ActorCritic(Model):
8
        def __init__(self, obs_dim, act_dim, hidden=64):
9
             super().__init__()
10
             self.shared1 = layers.Dense(hidden, activation='relu')
11
             self.shared2 = layers.Dense(hidden, activation='relu')
12
             self.logits = layers.Dense(act_dim)
13
            self.value = layers.Dense(1)
14
        def call(self, inputs):
15
16
            x = tf.convert_to_tensor(inputs, dtype=tf.float32)
17
            x = self.shared1(x)
18
            x = self.shared2(x)
19
            return self.logits(x), tf.squeeze(self.value(x), axis=-1)
20
21
22 def compute_gae_tf(rewards, masks, values, next_value, gamma=0.99, lam=0.95):
23
        values = np.append(values, next_value)
24
         gae = 0
        returns = []
25
26
        for step in reversed(range(len(rewards))):
27
            delta = rewards[step] + gamma * values[step + 1] * masks[step] - values
28
            gae = delta + gamma * lam * masks[step] * gae
29
            returns.insert(0, gae + values[step])
30
         return np.array(returns)
31
32
33 @tf.function
    def ppo_loss(model, obs, actions, old_logp, returns, adv, clip_epsilon=0.2,
34
     vf_coef=0.5, ent_coef=0.01):
35
         logits, values = model(obs)
        dist = tf.nn.softmax(logits)
36
37
         # new log probabilities
38
39
         action masks = tf.one hot(actions, logits.shape[-1])
40
         new_probs = tf.reduce_sum(dist * action_masks, axis=1) + 1e-8
41
        new_logp = tf.math.log(new_probs)
42
        ratio = tf.exp(new_logp - old_logp)
43
44
         s1 = ratio * adv
45
         s2 = tf.clip_by_value(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * adv
         policy_loss = -tf.reduce_mean(tf.minimum(s1, s2))
46
47
         value_loss = tf.reduce_mean(tf.square(returns - values))
         entropy = -tf.reduce_mean(tf.reduce_sum(dist * tf.math.log(dist + 1e-8),
48
         axis=1))
49
         total_loss = policy_loss + vf_coef * value_loss - ent_coef * entropy
50
51
         return total_loss, policy_loss, value_loss, entropy
52
53
54
   def ppo_update_tf(model, optimizer, obs, actions, old_logp, returns, adv,
     epochs=4, batch_size=64):
55
         n = obs.shape[0]
56
        idx = np.arange(n)
57
         for _ in range(epochs):
58
            np.random.shuffle(idx)
59
            for start in range(0, n, batch size):
60
                batch = idx[start:start + batch_size]
61
                with tf.GradientTape() as tape:
62
                     loss, pl, vl, ent = ppo loss(
63
                         model,
64
                         tf.convert_to_tensor(obs[batch], dtype=tf.float32),
65
                         tf.convert_to_tensor(actions[batch], dtype=tf.int32),
66
                         tf.convert_to_tensor(old_logp[batch], dtype=tf.float32),
67
                         tf.convert_to_tensor(returns[batch], dtype=tf.float32),
68
                         tf.convert_to_tensor(adv[batch], dtype=tf.float32),
69
70
                 grads = tape.gradient(loss, model.trainable_variables)
71
                 grads, _ = tf.clip_by_global_norm(grads, 0.5)
72
                 optimizer.apply_gradients(zip(grads, model.trainable_variables))
73
```

```
def train(env_name='CartPole-v1', total_timesteps=200, rollout_len=2048,
  75
       1r=3e-4):
  76
            env = gym.make(env name)
  77
            obs_dim = env.observation_space.shape[0]
            act_dim = env.action_space.n
  78
  79
  80
           model = ActorCritic(obs_dim, act_dim)
  81
            dummy = tf.zeros((1, obs dim))
  82
            model(dummy) # build network
           optimizer = optimizers.Adam(learning_rate=lr)
  83
  84
  85
           obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [],
           [], [], [], []
  86
           obs, _ = env.reset(seed=42)
  87
  88
           timestep = 0
  89
           ep_rewards = []
  90
  91
           while timestep < total_timesteps:</pre>
                for _ in range(rollout_len):
  92
  93
                    logits, value = model(np.expand_dims(obs, axis=0))
  94
                   logits = logits.numpy()[0]
                   value = value.numpy()[0]
  95
  96
                    probs = tf.nn.softmax(logits).numpy()
                    action = np.random.choice(len(probs), p=probs)
  97
  98
                   logp = np.log(probs[action] + 1e-8)
  99
                    next_obs, reward, terminated, truncated, _ = env.step(action)
 100
 101
                    done = terminated or truncated
 102
 103
                    obs_buf.append(obs.copy())
 104
                    action_buf.append(action)
 105
                    reward buf.append(reward)
                    done_buf.append(0.0 if done else 1.0)
 106
                    value_buf.append(value)
 107
 108
                   logp_buf.append(logp)
 109
                   obs = next_obs
 110
 111
                    timestep += 1
                    if done:
 112
 113
                        obs, _ = env.reset()
 114
 115
                # compute last value for GAE
                _, next_value = model(np.expand_dims(obs, axis=0))
 116
                next_value = next_value.numpy()[0]
 117
 118
 119
                returns = compute_gae_tf(reward_buf, done_buf, value_buf, next_value)
 120
                advantages = returns - np.array(value_buf)
 121
                advantages = (advantages - advantages.mean()) / (advantages.std() +
                1e-8)
 122
 123
                ppo_update_tf(model, optimizer,
 124
                              np.array(obs_buf), np.array(action_buf), np.array
                              (logp_buf),
 125
                              returns, advantages)
 126
 127
                avg_reward = np.sum(reward_buf[-rollout_len:]) / rollout_len
                print(f"Timesteps {timestep}\tAvgRolloutReward {avg_reward:.2f}")
 128
 129
                obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [],
 130
                [], [], [], [], []
 131
            env.close()
 132
 133
           return model
 134
 135
 136
       if __name__ == "__main__":
 137
           train()
Timesteps 2048 AvgRolloutReward 1.00
```

10/23/25, 9:49 AM RL Lab-09 - Colab