```python
# ppo_pytorch_gymnasium.py
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
from torch.distributions import Categorical
from collections import deque

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class ActorCritic(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden=64):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(obs_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU()
        )
        self.policy = nn.Linear(hidden, act_dim)
        self.value = nn.Linear(hidden, 1)

    def forward(self, x):
        h = self.shared(x)
        logits = self.policy(h)
        value = self.value(h).squeeze(-1)
        return logits, value


def compute_gae(rewards, masks, values, next_value, gamma=0.99, lam=0.95):
    values = np.append(values, next_value)
    gae = 0
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
        gae = delta + gamma * lam * masks[step] * gae
        returns.insert(0, gae + values[step])
    return np.array(returns)


def ppo_update(model, optimizer, obs, actions, log_probs_old, returns, advantages,
               clip_epsilon=0.2, vf_coef=0.5, ent_coef=0.01, epochs=4, batch_size=64):
    dataset_size = obs.shape[0]
    for _ in range(epochs):
        indices = np.arange(dataset_size)
        np.random.shuffle(indices)
        for start in range(0, dataset_size, batch_size):
            batch_idx = indices[start:start+batch_size]
            b_obs = torch.tensor(obs[batch_idx], dtype=torch.float32, device=device)
            b_actions = torch.tensor(actions[batch_idx], device=device)
            b_old_logp = torch.tensor(log_probs_old[batch_idx], device=device)
            b_returns = torch.tensor(returns[batch_idx], dtype=torch.float32, device=device)
            b_adv = torch.tensor(advantages[batch_idx], dtype=torch.float32, device=device)

            logits, values = model(b_obs)
            dist = Categorical(logits=logits)
            entropy = dist.entropy().mean()
            new_logp = dist.log_prob(b_actions)

            ratio = torch.exp(new_logp - b_old_logp)
            surr1 = ratio * b_adv
            surr2 = torch.clamp(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * b_adv
            policy_loss = -torch.min(surr1, surr2).mean()

            value_loss = ((b_returns - values) ** 2).mean()
            loss = policy_loss + vf_coef * value_loss - ent_coef * entropy

            optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 0.5)
            optimizer.step()


def train(env_name='CartPole-v1', total_timesteps=200, rollout_len=2048, lr=3e-4):
    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
```

```
    act_dim = env.action_space.n

    model = ActorCritic(obs_dim, act_dim).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [], [], [], [], []

    obs, _ = env.reset(seed=42)
    ep_rewards = deque(maxlen=100)
    timestep = 0

    while timestep < total_timesteps:
        for _ in range(rollout_len):
            obs_tensor = torch.tensor(obs, dtype=torch.float32, device=device).unsqueeze(0)
            logits, value = model(obs_tensor)
            dist = Categorical(logits=logits)
            action = dist.sample().item()
            logp = dist.log_prob(torch.tensor(action, device=device)).item()

            next_obs, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            obs_buf.append(obs.copy())
            action_buf.append(action)
            reward_buf.append(reward)
            done_buf.append(0.0 if done else 1.0)
            value_buf.append(value.item())
            logp_buf.append(logp)

            obs = next_obs
            timestep += 1
            if done:
                obs, _ = env.reset()

        obs_tensor = torch.tensor(obs, dtype=torch.float32, device=device).unsqueeze(0)
        _, next_value = model(obs_tensor)
        next_value = next_value.item()

        returns = compute_gae(reward_buf, done_buf, value_buf, next_value)
        advantages = returns - np.array(value_buf)
        advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

        ppo_update(model, optimizer,
                   np.array(obs_buf), np.array(action_buf), np.array(logp_buf),
                   returns, advantages)

        total_reward = sum(reward_buf[-rollout_len:])
        ep_rewards.append(total_reward)

        print(f"Timesteps: {timestep}\tAverage Rollout Reward: {np.mean(ep_rewards):.2f}")

        obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [], [], [], [], []

    env.close()
    return model


if __name__ == "__main__":
    train()

```

```
Timesteps: 2048 Average Rollout Reward: 2048.00
```

```python
# ppo_tensorflow_gymnasium.py
import gymnasium as gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model, optimizers

class ActorCritic(Model):
    def __init__(self, obs_dim, act_dim, hidden=64):
        super().__init__()
        self.shared1 = layers.Dense(hidden, activation='relu')
        self.shared2 = layers.Dense(hidden, activation='relu')
        self.logits = layers.Dense(act_dim)
        self.value = layers.Dense(1)

    def call(self, inputs):
        x = tf.convert_to_tensor(inputs, dtype=tf.float32)
        x = self.shared1(x)
        x = self.shared2(x)
        return self.logits(x), tf.squeeze(self.value(x), axis=-1)


def compute_gae_tf(rewards, masks, values, next_value, gamma=0.99, lam=0.95):
    values = np.append(values, next_value)
    gae = 0
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values
        [step]
        gae = delta + gamma * lam * masks[step] * gae
        returns.insert(0, gae + values[step])
    return np.array(returns)


@tf.function
def ppo_loss(model, obs, actions, old_logp, returns, adv, clip_epsilon=0.2,
vf_coef=0.5, ent_coef=0.01):
    logits, values = model(obs)
    dist = tf.nn.softmax(logits)

    # new log probabilities
    action_masks = tf.one_hot(actions, logits.shape[-1])
    new_probs = tf.reduce_sum(dist * action_masks, axis=1) + 1e-8
    new_logp = tf.math.log(new_probs)

    ratio = tf.exp(new_logp - old_logp)
    s1 = ratio * adv
    s2 = tf.clip_by_value(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * adv
    policy_loss = -tf.reduce_mean(tf.minimum(s1, s2))
    value_loss = tf.reduce_mean(tf.square(returns - values))
    entropy = -tf.reduce_mean(tf.reduce_sum(dist * tf.math.log(dist + 1e-8),
    axis=1))

    total_loss = policy_loss + vf_coef * value_loss - ent_coef * entropy
    return total_loss, policy_loss, value_loss, entropy


def ppo_update_tf(model, optimizer, obs, actions, old_logp, returns, adv,
epochs=4, batch_size=64):
    n = obs.shape[0]
    idx = np.arange(n)
    for _ in range(epochs):
        np.random.shuffle(idx)
        for start in range(0, n, batch_size):
            batch = idx[start:start + batch_size]
            with tf.GradientTape() as tape:
                loss, pl, vl, ent = ppo_loss(
                    model,
                    tf.convert_to_tensor(obs[batch], dtype=tf.float32),
                    tf.convert_to_tensor(actions[batch], dtype=tf.int32),
                    tf.convert_to_tensor(old_logp[batch], dtype=tf.float32),
                    tf.convert_to_tensor(returns[batch], dtype=tf.float32),
                    tf.convert_to_tensor(adv[batch], dtype=tf.float32),
                )
            grads = tape.gradient(loss, model.trainable_variables)
            grads, _ = tf.clip_by_global_norm(grads, 0.5)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

```

def train(env_name='CartPole-v1', total_timesteps=200, rollout_len=2048,
lr=3e-4):
    env = gym.make(env_name)
    obs_dim = env.observation_space.shape[0]
    act_dim = env.action_space.n

    model = ActorCritic(obs_dim, act_dim)
    dummy = tf.zeros((1, obs_dim))
    model(dummy)  # build network
    optimizer = optimizers.Adam(learning_rate=lr)

    obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [], [],
    [], [], [], []

    obs, _ = env.reset(seed=42)
    timestep = 0
    ep_rewards = []

    while timestep < total_timesteps:
        for _ in range(rollout_len):
            logits, value = model(np.expand_dims(obs, axis=0))
            logits = logits.numpy()[0]
            value = value.numpy()[0]
            probs = tf.nn.softmax(logits).numpy()
            action = np.random.choice(len(probs), p=probs)
            logp = np.log(probs[action] + 1e-8)

            next_obs, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            obs_buf.append(obs.copy())
            action_buf.append(action)
            reward_buf.append(reward)
            done_buf.append(0.0 if done else 1.0)
            value_buf.append(value)
            logp_buf.append(logp)

            obs = next_obs
            timestep += 1
            if done:
                obs, _ = env.reset()

        # compute last value for GAE
        _, next_value = model(np.expand_dims(obs, axis=0))
        next_value = next_value.numpy()[0]

        returns = compute_gae_tf(reward_buf, done_buf, value_buf, next_value)
        advantages = returns - np.array(value_buf)
        advantages = (advantages - advantages.mean()) / (advantages.std() +
        1e-8)

        ppo_update_tf(model, optimizer,
                      np.array(obs_buf), np.array(action_buf), np.array
                      (logp_buf),
                      returns, advantages)

        avg_reward = np.sum(reward_buf[-rollout_len:]) / rollout_len
        print(f"Timesteps {timestep}\tAvgRolloutReward {avg_reward:.2f}")

        obs_buf, action_buf, reward_buf, done_buf, value_buf, logp_buf = [],
        [], [], [], [], []

    env.close()
    return model


if __name__ == "__main__":
    train()

```

```
Timesteps 2048  AvgRolloutReward 1.00
```