Choosing an RL Example:
Classic Control Tasks (Gym): A great starting point is to explore OpenAI Gym's classic control environments like `CartPole-v1`, `LunarLander-v2`, or `Acrobot-v1`. These environments provide a well-defined reward structure and are relatively simple to learn, allowing you to focus on the core RL concepts.

Steps for Implementation:
Install required libraries: `pip install torch torchrl gym`
Import necessary modules.

Create an instance of the chosen Gym environment:
env = gym.make('CartPole-v1') # Or any other Gym environment

Policy Network: In reinforcement learning (RL), an agent interacts with an environment by acting and receiving rewards. The goal is to learn a policy, which maps states (observations of the environment) to actions.
A policy network is a neural network that approximates this policy. It takes the current state as input and outputs the probabilities for the agent's actions.
The agent selects an action based on these probabilities, often using a strategy that balances exploration (trying new actions) and exploitation (choosing actions with the highest predicted reward).

```
import torch
from torch import nn

class MLPPolicy(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(MLPPolicy, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)  # First hidden layer
        self.fc2 = nn.Linear(64, 32)  # Second hidden layer
```

```
        self.output = nn.Linear(32, action_dim)  # Output layer for
actions

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # Apply ReLU activation
        x = torch.relu(self.fc2(x))  # Apply ReLU activation
        return self.output(x)  # Return the predicted action
```

Adjust the network architecture (number of layers, neurons) based on the complexity of the environment.

Define the policy network using PyTorch modules. This network will take the agent's observation (state) as input and output an action (e.g., move left, right). Here's a basic example using a multilayer perceptron (MLP)
Multi-Layer Perception (MLP): An MLP is a type of artificial neural network architecture commonly used for various tasks, including function approximation and classification. It consists of multiple layers of interconnected nodes (artificial neurons).
Each layer transforms the input it receives from the previous layer using a mathematical function (activation function). Common activation functions include ReLU (Rectified Linear Unit) used in the provided code.
By stacking multiple layers, MLPs can learn complex relationships between the input and output, making them suitable for approximating the policy function in an RL setting.

We use network policy representation because it offers a powerful and adaptable way to learn policies directly from experience. It can capture non-linear relationships between states and actions, which are often present in real-world environments. The ability to train and update these networks allows the agent to continuously improve its policy through interaction with the environment.

RL Algorithm Integration with torchRL:

Import the chosen RL algorithm from torchRL (e.g., `torchrl.algorithms.PPO`).
Instantiate the RL algorithm, providing the environment, policy network, and hyperparameters:
from torchrl.algorithms import PPO

```
agent = PPO(
    env=env,
    policy_network=MLPPolicy(env.observation_space.shape[0],
env.action_space.n),
    learning_rate=0.001,  # Adjust hyperparameters as needed
    discount_factor=0.99,
    entropy_coefficient=0.01
)
```

Proximal Policy Optimization (PPO): PPO is a policy gradient reinforcement learning algorithm that aims to find a balance between achieving good performance (high rewards) and exploring different actions.
PPO does this by clipping the policy updates during training, ensuring the new policy doesn't deviate too much from the previous one. This helps maintain stability and prevents the policy from collapsing.
PPO has become a popular choice for RL tasks due to its effectiveness, stability, and ease of implementation.

Entropy Coefficient:
The entropy coefficient is a hyperparameter used in PPO to encourage exploration during training.
Entropy refers to the amount of uncertainty or randomness in the agent's policy. A high entropy policy explores more, while a low entropy policy tends to stick to the actions it has found successful.
By introducing an entropy bonus term in the objective function, PPO encourages the agent to maintain a diverse set of actions,

preventing it from getting stuck in local optima (suboptimal solutions).
The entropy coefficient controls the strength of this exploration encouragement. A higher value promotes more exploration, while a lower value focuses more on exploiting high-reward actions.

Training Loop:
Implement a training loop that interacts with the environment, collects experience (states, actions, rewards, next states), and uses torchRL's algorithm to update the policy network:

```
total_episodes = 1000
for episode in range(total_episodes):
    observation = env.reset()
    done = False
    while not done:
        # Take action based on observation and current policy
        action = agent.predict(observation)

        # Interact with the environment
        next_observation, reward, done, info = env.step(action)

        # Feed experience to the RL algorithm (may differ slightly for different algorithms)
        agent.experience(observation, action, reward, next_observation, done)
        observation = next_observation

    # Update the policy network after each episode (may differ)
    agent.update()

# Close the environment after training
env.close()
```

Example program

```
import gym
import torch
import torch.nn as nn
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor

class MLPPolicy(nn.Module):
    def __init__(self, observation_space, action_space):
        super(MLPPolicy, self).__init__()
        state_dim = observation_space.shape[0]
        action_dim = action_space.n
        self.fc1 = nn.Linear(state_dim, 64)  # First hidden layer
        self.fc2 = nn.Linear(64, 32)  # Second hidden layer
        self.output = nn.Linear(32, action_dim)  # Output layer for
actions

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # Apply ReLU activation
        x = torch.relu(self.fc2(x))  # Apply ReLU activation
        return self.output(x)  # Return the predicted action

class CustomFeatureExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space, features_dim=32):
        super(CustomFeatureExtractor, self).__init__(observation_space,
features_dim)
        state_dim = observation_space.shape[0]
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, features_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return torch.relu(self.fc2(x))

env = gym.make('CartPole-v1')
env = DummyVecEnv([lambda: env])  # Vectorized environment

policy_kwargs = dict(
    features_extractor_class=CustomFeatureExtractor,
    features_extractor_kwargs=dict(features_dim=32),
)
```

```
agent = PPO(
    "MlpPolicy",
    env,
    policy_kwargs=policy_kwargs,
    learning_rate=0.001,
    n_steps=2048,
    batch_size=64,
    n_epochs=10,
    gamma=0.99,
    ent_coef=0.01,
    verbose=1,
)

total_timesteps = 100000
agent.learn(total_timesteps=total_timesteps)

# Save the trained agent
agent.save("ppo_cartpole")

# Close the environment after training
env.close()
```

Here , the hyper-parameters of PPO

1. Learning Rate: Controls the step size for updating the policy and value function networks.
2. Clip Range (epsilon): Defines the threshold to clip the probability ratio between old and new policies to ensure updates are not too large.
3. Epochs: The number of passes over the data during each training update.
4. Batch Size: The number of samples used for each training update within an epoch.
5. Mini-batch Size: The number of samples used in each mini-batch when performing a training update.
6. Discount Factor (gamma): The factor by which future rewards are discounted back to the present.

7. Lambda (GAE): The parameter for Generalized Advantage Estimation, controlling the bias-variance trade-off in advantage calculation.
8. Entropy Coefficient: The weight of the entropy term in the loss function, encouraging exploration.
9. Value Function Coefficient: The weight of the value loss term in the total loss function.
10. Max Gradient Norm (gradient clipping): The threshold to clip gradients to avoid exploding gradients during backpropagation.
11. Timesteps per Batch: The number of timesteps of experience to collect per batch for updating the policy.

Expected output

| time/ | |

| fps | 146 |

| iterations | 1 |

| time_elapsed | 13 |

| total_timesteps | 2048 |

| time/ | |

| fps | 116 |

| iterations | 2 |

| time_elapsed | 35 |

| total_timesteps | 4096 |

| train/ | |

| approx_kl | 0.01443335 |

| clip_fraction | 0.186 |

| clip_range | 0.2 |

| entropy_loss | -0.682 |

| explained_variance | -0.0362 |

| learning_rate | 0.001 |

| loss | 4.38 |

| n_updates | 10 |

| policy_gradient_loss | -0.022 |

| value_loss | 21 |

It shows up to 16 iterations, This code sets up a custom Proximal Policy Optimization (PPO) agent using Stable Baselines3 to train on the 'CartPole-v1' environment. It defines a custom feature extractor and policy network architecture, specifies PPO hyperparameters,

and trains the agent for 100,000 timesteps. After training, the agent is saved to a file named "ppo_cartpole" and the environment is closed.