# NREIP TestRL Notes

Rohan Pandey

November 20, 2024

## Explanation of the MATLAB Script

The MATLAB script is designed to set up and train a Deep Deterministic Policy Gradient (DDPG) agent within a Simulink environment for a reinforcement learning task. Here's a breakdown of what each part of the script does:

### 1. Define Observation and Action Specifications

- `obsInfo = rlNumericSpec([3, 1], LowerLimit = 0, UpperLimit = 1);`
  - This line creates an observation specification object defining the shape and range of the observations the agent will receive. It specifies a 3-dimensional vector with values ranging from 0 to 1.
- `actInfo = rlNumericSpec([3, 1], LowerLimit = -0.5 , UpperLimit = 0.5);`
  - Similarly, this line creates an action specification object defining the shape and range of the actions the agent can take. It specifies a 3-dimensional vector with values ranging from -0.5 to 0.5.

### 2. Create the Reinforcement Learning Environment

- `env = rlSimulinkEnv("RL_Rough_Structure5", "RL_Rough_Structure5/RL Agent", obsInfo, actInfo);`
  - This line initializes the Simulink environment for reinforcement learning. It specifies the Simulink model `RL_Rough_Structure5` and the block `RL Agent` within that model where the agent interfaces.
  - It also passes the observation and action specifications to the environment.

### 3. Configure the Agent Options

- `opts = rlDDPGAgentOptions("DiscountFactor", 0.6);`
  - This line creates an options object for the DDPG agent, setting the discount factor (which determines the importance of future rewards) to 0.6.

## 4. Create the DDPG Agent

- `agent = rlDDPGAgent(obsInfo, actInfo, opts);`
  - This line initializes the DDPG agent using the previously defined observation and action specifications and the agent options.

## 5. Set Training Options

- `opts = rlTrainingOptions("MaxEpisodes", 2000, "MaxStepsPerEpisode", 10, "ScoreAveragingWindowLength", 100, "StopTrainingCriteria", "AverageReward", "StopTrainingValue", 10000);`
  - This line sets up the training options, including:
    * Maximum number of episodes: 2000
    * Maximum steps per episode: 10
    * Window length for averaging the reward: 100
    * Criteria to stop training when the average reward reaches 10000

## 6. Train the Agent

- `train(agent, env, opts);`
  - This line starts the training process of the agent within the environment using the specified training options.

# Python Implementation

Now, let's walk through the equivalent Python code using the `gym` and `stable_baselines3` libraries.

## 1. Import Necessary Libraries

```
import gym
import numpy as np
from stable_baselines3 import DDPG
from stable_baselines3.common.noise import NormalActionNoise
from stable_baselines3.common.callbacks import BaseCallback
```

## 2. Define the Custom Environment

Since we don't have the Simulink environment in Python, we'll need to create a custom environment that mimics the behavior.

```
class CustomEnv(gym.Env):
    def __init__(self):
        super(CustomEnv, self).__init__()
```

```
        # Observation space: 3-dimensional vector with red↩
            red→ values between 0 and 1
        self.observation_space = gym.spaces.Box(low=0, highred↩
            red→ =1, shape=(3,), dtype=np.float32)
        # Action space: 3-dimensional vector with values red↩
            red→ between -0.5 and 0.5
        self.action_space = gym.spaces.Box(low=-0.5, highred↩
            red→ =0.5, shape=(3,), dtype=np.float32)
        self.state = None
        self.current_step = 0
        self.max_steps_per_episode = 10

    def reset(self):
        self.state = np.random.uniform(0, 1, size=(3,))
        self.current_step = 0
        return self.state

    def step(self, action):
        # Apply action to the environment (define your red↩
            red→ dynamics here)
        # For demonstration, we'll just generate a random red↩
            red→ next state
        self.state = np.random.uniform(0, 1, size=(3,))
        # Define a reward function (customize this based on red↩
            red→ your problem)
        reward = -np.sum(np.square(action))  # Example: red↩
            red→ Penalize large actions
        self.current_step += 1
        done = self.current_step >= self.red↩
            red→ max_steps_per_episode
        info = {}
        return self.state, reward, done, info
```

### 3. Create the Environment Instance

```
env = CustomEnv()
```

### 4. Configure the Agent Options

```
# Number of actions
n_actions = env.action_space.shape[0]

# Add action noise for exploration
action_noise = NormalActionNoise(mean=np.zeros(n_actions), red↩
    red→ sigma=0.1 * np.ones(n_actions))

# Create the DDPG agent
```

```
agent = DDPG(
    "MlpPolicy",
    env,
    gamma=0.6,  # Discount factor equivalent to MATLAB's red↩
        red→ DiscountFactor
    action_noise=action_noise,
    verbose=1,
)
```

## 5. Implement Custom Training Callback for Stopping Criteria

We need to implement a callback to stop training when the average reward reaches 10000 over a window length of 100 episodes.

```
class StopTrainingOnRewardThreshold(BaseCallback):
    def __init__(self, reward_threshold, window_length, red↩
        red→ verbose=0):
        super(StopTrainingOnRewardThreshold, self).__init__(red↩
            red→ verbose)
        self.reward_threshold = reward_threshold
        self.window_length = window_length
        self.episode_rewards = []

    def _on_step(self):
        if self.locals.get('dones'):
            # Episode finished
            episode_reward = self.locals['infos'][0].get('red↩
                red→ episode')['r']
            self.episode_rewards.append(episode_reward)
            if len(self.episode_rewards) > self.red↩
                red→ window_length:
                self.episode_rewards = self.episode_rewardsred↩
                    red→ [-self.window_length:]
                average_reward = np.mean(self.red↩
                    red→ episode_rewards)
                if average_reward >= self.reward_threshold:
                    print(f"Stopping training as average red↩
                        red→ reward {average_reward} "
                              f"over {self.window_length} red↩
                                red→ episodes >= {self.red↩
                                red→ reward_threshold}")
                    return False  # Returning False stops red↩
                        red→ training
        return True
```

## 6. Train the Agent with the Training Options

```
# Training options
```

```
callback = StopTrainingOnRewardThreshold(reward_thresholdred↩
    red→ =10000, window_length=100)

# Train the agent
agent.learn(total_timesteps=2000 * 10, callback=callback)  #red↩
    red→  Total timesteps equivalent to MaxEpisodes * red↩
    red→ MaxStepsPerEpisode
```

## 7. Save the Trained Agent

```
agent.save("ddpg_agent")
```

## 8. Load and Test the Trained Agent

```
# Load the trained agent
agent = DDPG.load("ddpg_agent", env=env)

# Test the agent
obs = env.reset()
for _ in range(10):
    action, _states = agent.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    if done:
        obs = env.reset()
```

## Explanation

In the Python implementation:

- We create a custom environment `CustomEnv` that defines the observation and action spaces equivalent to the MATLAB specifications.
- The `reset` and `step` methods are placeholders and should be implemented based on the specific dynamics of your environment. The reward function and transition dynamics need to reflect your actual problem.
- We configure the DDPG agent with a discount factor (`gamma`) of 0.6, matching the MATLAB agent options.
- A custom callback `StopTrainingOnRewardThreshold` is implemented to stop training when the average reward over a specified window reaches a certain threshold, similar to MATLAB's `StopTrainingCriteria` and `StopTrainingValue`.
- We train the agent for a total number of timesteps calculated by multiplying `MaxEpisodes` and `MaxStepsPerEpisode` to mirror MATLAB's training duration.

## Additional Notes

- The `stable_baselines3` library does not natively support all the training options available in MATLAB's `rlTrainingOptions`, so custom implementations (like the callback) are necessary.
- Ensure that the reward function and the environment dynamics in the `step` method accurately represent your specific use case for meaningful training results.
- The action noise is added to encourage exploration during training, similar to the behavior of the DDPG algorithm in MATLAB.
- Saving and loading the agent allows you to persist the trained model and reuse it without retraining.