AI-Driven Reconfigurable Intelligent Surface (RIS) Control Using Reinforcement Learning

Deep Deterministic Policy Gradient (DDPG)-based phase optimization for dynamic 6G wireless environments

Author

Jay Gautam, February 2026

RIS Conceptual Reference / Inspiration

Inspired by many foundational research on RIS physics and propagation, including some RIS-physics-based formulations from

“Intelligent Reflecting Surfaces: Physics, Propagation, and Pathloss Modeling by Özgecan Özdogan, Emil Björnson, and Erik G. Larsson.”

Content

About me:

Who Am I?

> I am a wireless and RF engineer working on 5G and 6G technologies, with hands-on experience in RIS, metasurfaces, Antennas and wireless system modeling. Over time, my work has naturally moved toward AI-driven methods for wireless communication, while staying strongly connected to the underlying physics.
>
> Today, I actively work with machine learning, deep learning, and reinforcement learning, and apply these tools to wireless and 6G-related problems.

My Introduction to the AI World:

> In 2021, just before starting my master's thesis at TU Darmstadt, Germany, I coincidentally watched a few movies — Zoe (2018), Archive (2020), and Ex Machina (2014). For the first time, I started seeing the future in a very different way. That moment pushed me to decide that I wanted to move toward AI based engineering.
>
> At that time, I had almost no background in programming and certainly no background in ML/DL/AI. Still, I chose a machine-learning-based (RF/Microwave domain) topic for my master's thesis, completed it successfully, and continued learning step by step by myself. I consciously mapped my AI learning to wireless and communication problems.
>
> Now, I am comfortable implementing ML, DL, and RL algorithms — including DQN, PPO (Discrete and Continuous), DDPG, TD3, and SAC — for tasks related to 6G wireless technology.

My Future Research Direction- my future focus is on AI-driven implementations aligned with the 6G vision, especially:

- Reinforcement learning–based AI control for RIS and wireless systems
- Autonomous and adaptive 6G environments
- Sustainable and intelligent resource management
- Energy-aware AI methods for wireless networks

AI Driven Problem statement:

This work implements intelligent phase control of Reconfigurable Intelligent Surfaces (RIS) for future 6G wireless networks under dynamic channel conditions. The goal is to maximize spectral efficiency, SNR and reliability (through outage prediction) by formulating RIS control as a model-free reinforcement learning problem, using a DDPG agent to learn continuous phase configurations directly from interaction with the environment. Performance is evaluated in terms of throughput, SNR, and outage probability, and compared against closed-form analytical optimization and no-RIS baselines.

Brief Explanation: RL Agent (DDPG) and RIS

Reconfigurable Intelligent Surfaces (RIS) are considered a important technology for future 6G networks. However, to address the radio environment to support the communication, dynamically configuring RIS phase shifts of RIS panel, remains a challenging Goal.

In this work, RIS phase control is achieved using a Model Free Deep Reinforcement Learning, DDPG-Agent to learn continuous phase configurations directly from interaction with the environment.

The wireless channel is modeled using a time-correlated Rayleigh fading process ($\rho = 0.95$), representing a highly dynamic and non-ideal propagation scenario rather than an idealized static channel.

The objective is to maximize spectral efficiency, improve SNR, and reduce outage probability under time-varying conditions.

The learned policy is evaluated against a closed-form phase-alignment baseline and a no-RIS scenario.

Reinforcement Learning (RL) Elements

State: Current wireless channel observations, including the source–RIS, RIS–destination, and direct link information.

Action: Continuous phase adjustments applied to each RIS element.

Reward: Communication performance in terms of spectral efficiency- SNR and outage behavior.

AI Driven Implementation: DDPG using PyTorch and Python

Section1: Libraries and Parameters

- Deep learning libraries are used to model the wireless environment, implement deep reinforcement learning, and visualize training and evaluation results.
- Devices choose GPU acceleration when available to improve training efficiency.
- System-level parameters: **RIS size**, **episode length**, and **number of training episodes**, etc.
- Channel dynamics for a **time-correlation (fading) parameter**, controlling how rapidly the wireless channel evolves over time.
- Noise power to reflect non-ideal communication conditions.
- Training-related parameters: batch size and moving-average windows
- Evaluation parameters, including outage thresholds and evaluation episodes, for performance comparison among different strategies.

```
# Libraries, Import
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from collections import deque
import random
import matplotlib.pyplot as plt
import time

# Reproducibility
SEED = 0
np.random.seed(SEED)
random.seed(SEED)
torch.manual_seed(SEED)

# Device choices
```

```
if torch.cuda.is_available():
  torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
print("Computing Device:", device)


# Parametrs required for the project
N_ELEMENTS = 16 # Number of RIS Elements
EPISODES = 600 # Number of training episodes
EPISODE_LENGTH = 50 # Length of each training episodes

BATCH_SIZE = 32 # Batch Size for Training
NOISE_POWER = 1e-3 # Noise Power in Watt
RHO = 0.5 # 0.95 # Fading parameter
MOV_WINDOW = 50 # Moving Average Window

EVAL_EPISODES = 100 # Number of episodes for the evaluation
process.
OUTAGE_SNR_DB = 0.0 # Threshold for SNR to determine Outage
condition. in dB
OUTAGE_SNR_LIN = 10**(OUTAGE_SNR_DB/10) # Threshold for SNR to
determine Outage condition. in Linear
```

Section 2: RIS Environment for Reinforcement learning (DDPG) Agent: No Direct link between Base Station and User

- This environment models a **pure RIS-assisted communication scenario**, where the base station and user communicate **only through the RIS**, with no direct propagation path.

- The wireless channel consists of two cascaded links: **source to RIS** and **RIS-to-user**, both modeled as **Rayleigh fading channels**, including Rho factor to indicate the strength of the fading.

- Channel dynamics are captured using a **time-correlated fading model (RHO)**, allowing the channel to evolve realistically across time steps within an episode.

- The **state representation (Channel)** includes the real and imaginary components of both channel links, providing the agent with instantaneous channel observations.
- The **action space (Phase)** is continuous and corresponds to the phase shifts applied to each RIS element, enabling fine beamforming control.
- The **reward (SNR or Capacity)** is defined based on the achievable spectral efficiency, encouraging the agent to align RIS phases to maximize SNR- to bring better signal quality.

```
# Define a class: RIS Environment for the Agent. Without
Direct path. h_sr, h_rd, no h_bd
# With this class. the agent steps and get the reward based on
the definition.

class RISEnv:
  # Constructor
  def __init__(self, N = 16, noise_power = 1e-3, rho = 0.95,
episode_length = 50):
    self.N = N # number of RIS Elements. NXN Panel
    self.noise_power = noise_power
    self.rho = rho # Temporal relation coefficients. Fading.
Dynamic Challel modelling.
    self.episode_length = episode_length
    self.state_dim = 4*N # Real and maginary parts for BS->RIS
and RIS->User
    self.action_dim = N # Phase. action dim = number of RIS
Elements.
    self.step_count = 0 # Counter

  # Initialize Random channel and reset step counter. Rayleigh
model.
  def reset(self):
    self.step_count = 0
    self.h_sr = (np.random.randn(self.N) +
1j*np.random.randn(self.N)) / np.sqrt(2) # BS(s)-> RIS(r)
    self.h_rd = (np.random.randn(self.N) +
1J*np.random.randn(self.N)) / np.sqrt(2) # RIS(r)-> Device(d)
```

```
    return self._get_state() # Private function. Hidden for
outside users.

  def _get_state(self): # Channel Matrix. Private function.
OOP-Polymorphism concept.
    return np.concatenate([np.real(self.h_sr),
np.imag(self.h_sr),
                           np.real(self.h_rd),
np.imag(self.h_rd)])

  # Rayleigh Cahhel + Fast fading (rho factor) channel.
Private function. OOP-Polymorphism concept.
  def _update_channel(self):
    self.h_sr = self.rho*self.h_sr + np.sqrt(1-
self.rho**2)*(np.random.randn(self.N) +
1j*np.random.randn(self.N)) / np.sqrt(2)
    self.h_rd = self.rho*self.h_rd + np.sqrt(1-
self.rho**2)*(np.random.randn(self.N) +
1j*np.random.randn(self.N)) / np.sqrt(2)

  def step(self, action):
    # Take action
    theta = (action + 1)* np.pi # Phase shift by the element.
[0, 2*pi]
    Phi = np.exp(1j*theta) # RIS Phase shift Matrix

    # Get reward for the action
    gain = np.abs(np.dot(self.h_rd.conj().T, Phi * self.h_sr))
** 2
    snr = gain / self.noise_power
    reward = np.log2(1 + snr/self.N) # Shanons Capacity
formula.

    # Get the next state, return done flag and rewrd
    self._update_chnnel()
    self.step_count +=1

    done = self.step_count >= self.episode_length # done flag,
if it exceed the episode length.
```

```
    next_state = self._get_state() # Get new state (channel)

    return next_state, reward, done, {"snr": snr}
```

Section 3: RIS Environment for Reinforcement learning (DDPG) Agent: With Direct link between Base Station and User

- This environment extends the RIS-assisted system model by **a direct propagation path** between the base station and the user, in addition to the RIS-assisted reflected path.
- The overall received signal is modeled as the **superposition of the direct link and the RIS-reflected link**, representing a more realistic wireless scenario.
- All channel components—source-to-RIS, RIS-to-user, and base-station-to-user—are modeled as **time-correlated Rayleigh fading channels**, enabling dynamic channel evolution across time steps.
- The **state representation** is concatenated to include the real and imaginary parts of the direct channel, allowing the agent to jointly account for both direct and reflected signal contributions.
- The **action space** remains continuous and corresponds to RIS phase adjustments, enabling adaptive beamforming in the presence of a competing direct signal.
- The **reward** is defined using spectral efficiency based on the total received signal power, encouraging intelligent phase control that constructively combines reflected and direct components.
- This environment is primarily used for **training and fair performance evaluation**, enabling direct comparison between DDPG learning-based control, closed-form RIS optimization, and no-RIS baselines under identical channel dynamics.

```
# Author: Jay Gautam
# Define a class: RIS Environment for the Agent. With Direct path. h_sr, h_rd, h_bd
# With this class. the agent steps and get the reward based on the definition.

class RISEnvFull:
```

```
    def __init__(self, N=16, noise_power=1e-3, rho=0.95,
episode_length=50):
        self.N = N # number of RIS Elements. NXN Panel
        self.noise_power = noise_power
        self.rho = rho # Temporal relation coefficients.
Fading. Dynamic Challel modelling.
        self.episode_length = episode_length
        self.state_dim = 4 * N + 2 # Real and maginary parts
for BS->RIS and RIS->User, and BS(b)->User(d)
        self.action_dim = N # Phase. action dim = number of
RIS Elements.
        self.step_count = 0 # Counter

    # Initialize Random channel and reset step counter.
Rayleigh model.
    def reset(self):
        self.step_count = 0
        self.h_sr = (np.random.randn(self.N) +
1j*np.random.randn(self.N))/np.sqrt(2) # BS(s)-> RIS(r)
        self.h_rd = (np.random.randn(self.N) +
1j*np.random.randn(self.N))/np.sqrt(2) # RIS(r)-> Device(d)
        self.h_bd = (np.random.randn() +
1j*np.random.randn())/np.sqrt(2) # BS(b)->User(d)
        return self._get_state()

    # Channel Matrix
    def _get_state(self):
        return np.concatenate([np.real(self.h_sr),
np.imag(self.h_sr),
                               np.real(self.h_rd),
np.imag(self.h_rd),
                               np.array([np.real(self.h_bd),
np.imag(self.h_bd)])])
    # Rayleigh Cahhel + Fast fading (rho factor) channel.
Private function. OOP-Polymorphism concept.
    def _update_channel(self):
        self.h_sr = self.rho * self.h_sr + np.sqrt(1 -
self.rho**2) * (
```

```
            (np.random.randn(self.N) + 1j *
np.random.randn(self.N)) / np.sqrt(2)
        )
        self.h_rd = self.rho * self.h_rd + np.sqrt(1 -
self.rho**2) * (
            (np.random.randn(self.N) + 1j *
np.random.randn(self.N)) / np.sqrt(2)
        )
        self.h_bd = self.rho * self.h_bd + np.sqrt(1 -
self.rho**2) * (
            (np.random.randn() + 1j * np.random.randn()) /
np.sqrt(2)
        )

    # Take action and get reward and new state.
    def step(self, action):
        # Take action, phase change.
        theta = (action + 1.0) * np.pi   # theta in [0, 2pi]
        Phi = np.exp(1j * theta)         # element-wise phase
shift (length N)

        # Get reward for the action
        reflected = np.dot(self.h_rd.conj().T, Phi *
self.h_sr)
        total = self.h_bd + reflected
        gain = np.abs(total)**2
        snr = gain / self.noise_power
        reward = np.log2(1 + snr)  # spectral efficiency per
step

        # Get the next state, return done flag and rewrd
        self._update_channel()
        self.step_count += 1
        done = self.step_count >= self.episode_length # done
flag, if exceed predefined episode length.
        next_state = self._get_state() # reset the state
        info = {"snr": snr, "gain": gain, "reflected":
np.abs(reflected)**2, "direct": np.abs(self.h_bd)**2}
        return next_state, reward, done, info
```

Section 4: Actor–Critic Architecture for Continuous RIS Control (DDPG)

- The RIS phase control problem involves a **continuous, high-dimensional action space**.
- The **actor network** represents the control policy and maps observed wireless channel states directly to continuous RIS phase control actions.
- The **critic network** evaluates the quality of each state–action pair by estimating the expected long-term return, providing a learning signal to guide policy improvement.
- Both networks are implemented as **fully connected deep neural networks**, enabling nonlinear function approximation for complex channel–action relationships.
- A bounded activation function is used at the actor output to ensure that actions remain within physically meaningful limits for RIS phase control.
- Xavier **weight initialization** is applied to stabilize training.
- This actor–critic structure enables **stable learning in continuous action spaces**, forming the foundation of the Deep Deterministic Policy Gradient (DDPG) algorithm used in this work.

```
# Author: Jay Gautam
# Actor-Critic Network: Policy network
# Condition for stable training: Initialize weight and bias
# .. for linear network using Xavier function.
def init_weights(layer):
    if isinstance(layer, nn.Linear):
        nn.init.xavier_uniform_(layer.weight)
        nn.init.constant_(layer.bias, 0.0)

# Policy network: States to Action- to find RIS phases.
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=256):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, action_dim)
        self.apply(init_weights)
```

```
    def forward(self, s): # states as input, action as output
        x = F.relu(self.fc1(s))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))

# Q-Value network
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=256):
        super().__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, 1)
        self.apply(init_weights)
    def forward(self, s, a): # States and Actions as input.
Q-value for (s,a) pair as output.
        x = torch.cat([s, a], dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Section 5: DDPG Agent Design and Learning Mechanism

- The RIS control problem: using a **model-free Deep Deterministic Policy Gradient (DDPG) agent**, which learns optimal behavior purely from interaction with the environment.
- The agent consists of **actor and critic networks**, along with their corresponding **target networks**, enabling stable learning in continuous action spaces.
- **Target networks** are used to decouple learning targets from rapidly changing network parameters, improving training stability.
- An **experience replay buffer** stores past interactions, state, action, reward, done flag, next state, allowing the agent to learn from randomized mini-batches and break temporal correlations in the data.
- Exploration is encouraged by injecting **noise** into the actor's actions during training, enabling effective exploration of the continuous phase space.

- The **critic** learns to evaluate long-term returns for state–action pairs, while the **actor** is optimized to select actions that maximize the critic's assessment.
- A **soft-update mechanism (Polyak averaging)** is applied to the target networks, ensuring smooth parameter updates and preventing training oscillations.
- This learning mechanism enables the agent to gradually discover RIS phase configurations that perform well under **time-varying wireless channel conditions**.

```
# Author: Jay Gautam
# Deep Deterministic Policy Gradient (DDPG) Agent
# DDPG Agent (Model Free) learns purely based on experinece
## actor, actor_target, actor_opt
## critc, critic_target, critic_opt
class DDPGAgent:
    def __init__(self, state_dim, action_dim, device):
        self.device = device
        self.actor = Actor(state_dim, action_dim).to(device) #
actor
        self.actor_target = Actor(state_dim,
action_dim).to(device) # actor target

self.actor_target.load_state_dict(self.actor.state_dict()) #
actor target

        self.critic = Critic(state_dim, action_dim).to(device)
# Critic
        self.critic_target = Critic(state_dim,
action_dim).to(device) # Critic Target

self.critic_target.load_state_dict(self.critic.state_dict()) #
Critic Target

        self.actor_opt = optim.Adam(self.actor.parameters(),
lr=1e-4) # Actor: Adam optimizer
        self.critic_opt = optim.Adam(self.critic.parameters(),
lr=1e-3) # Critic: Adam Optimizer
        self.buffer = deque(maxlen=100000) # Experience replay
buffer, size = maxlen
```

```
        self.gamma = 0.99 # Discount factor
        self.tau = 0.005 # Polyak averaging parameter

    def select_action(self, state, noise_scale=0.02):
        s = torch.FloatTensor(state.reshape(1,
-1)).to(self.device)# state tensor data type
        a = self.actor(s).cpu().data.numpy().flatten()  #
action tensor data type
        # Add Gaussian noise to the action for more
exploration
        a += np.random.normal(0, noise_scale, size=a.shape)
        return np.clip(a, -1, 1) # Return the Clipped action
(bounded value)

    # Experience replay buffer. Store the Transitions.
    def store(self, transition):
        self.buffer.append(transition)

    def train(self, batch_size=64):
        if len(self.buffer) < batch_size: # check the
sufficient memory size for training
            return
        s, a, r, s2, d = map(np.stack,
zip(*random.sample(self.buffer, batch_size)))# extract RL
elements
        s, a, r, s2, d = [torch.FloatTensor(x).to(self.device)
                          for x in [s, a, r.reshape(-1, 1),
s2, d.reshape(-1, 1)]] # Tensor form data

        with torch.no_grad():
            next_a = self.actor_target(s2)
            target_Q = r + (1 - d) * self.gamma *
self.critic_target(s2, next_a) # Bellman Equation
        # Critic Network Update
        loss_c = F.mse_loss(self.critic(s, a), target_Q)
        self.critic_opt.zero_grad(); loss_c.backward();
self.critic_opt.step()

        # Actor Network Update
```

```
        loss_a = -self.critic(s, self.actor(s)).mean()
        self.actor_opt.zero_grad(); loss_a.backward();
self.actor_opt.step()

        # Soft update of actor/critic target networks- Polyak
averaging.
        for net, tgt in [(self.actor, self.actor_target),
(self.critic, self.critic_target)]:
            for p, tp in zip(net.parameters(),
tgt.parameters()):
                tp.data.copy_(self.tau * p.data + (1 -
self.tau) * tp.data)
```

Section 6: Closed Form Phase Optimization: Analytical Approach (Baseline Methods for Performance Comparison)

- This section implements a **closed-form analytical baseline** for RIS phase optimization, serving as a reference point for evaluating learning-based DDPG control.
- The analytical approach computes RIS phase shifts by **explicitly aligning the phases of the cascaded channels**, aiming to coherently combine reflected signals at the receiver.
- This method assumes **instantaneous channel knowledge** and applies a deterministic mapping from channel phases to RIS phase configurations.
- Unlike reinforcement learning, the closed-form solution does **not involve exploration or learning over time**, and it cannot adapt beyond its predefined analytical structure.
- The closed-form policy provides an **upper-bound-style benchmark** under ideal assumptions and is useful for quantifying the performance gap between analytical optimization and model-free learning. All the real world channel complexity is ignored
- Comparing DDPG against this baseline helps highlight the **strengths and limitations of learning-based RIS control** in dynamic wireless environments.

```
# Author: Jay Gautam
```

```
# Closed-form implementation: Analytical function- to compare
with DDPG agent
## Analytical implementation: RIS Phase tuning. BS-> RIS,
RIS->Device
def closed_form_action_from_channels(h_sr, h_rd):
    theta = np.angle(h_rd) - np.angle(h_sr)
    theta = np.mod(theta, 2*np.pi)
    a = theta / np.pi - 1.0
    return np.clip(a, -1.0, 1.0) # used to find analytical
phase shift matrix.
```

Section 7: Evaluation Module: Closed Form Phase Optimization: Analytical Approach

- This section evaluates the **analytical (closed-form) RIS phase optimization** in a controlled and repeatable manner.
- The closed-form policy is applied **step by step** over multiple episodes using the same environment dynamics as the RL agent.
- For each episode, the environment is reset and the RIS phases are computed directly from the instantaneous channel realizations.
- At every time step, the resulting **reward, SNR, and outage condition** are recorded to analyze communication performance.
- **Throughput** is computed as the average spectral efficiency over the episode length.
- **SNR** is averaged across time steps and converted to dB for easier interpretation.
- **Outage probability** is measured as the fraction of time steps where the SNR falls below a predefined threshold.
- This evaluation provides an analytical **baseline**, helping to clearly compare closed-form optimization against learning-based RIS control.

```
# Author Jay Gautam
# Analysis and Evaluation of the Closed Form Imlementation.
# It will return Throughput, SNR and Outage count value.

def evaluate_closed_form(env, episodes=200):
```

```
    througputs = [] # KPI
    snrs_db = []
    outages = [] # KPI: Outage Probability
    for ep in range(episodes):
        _ = env.reset() # Get envitonment parameters, based on
the choice.
        ep_th = 0.0
        ep_snrs = []
        ep_outage_count = 0
        for _step in range(env.episode_length):
            a = closed_form_action_from_channels(env.h_sr,
env.h_rd) # Action
            _, r, _, info = env.step(a) # Step in the
environment with action value "a", get reward
            ep_th += r
            snr = float(info["snr"])
            ep_snrs.append(snr)
            if snr < OUTAGE_SNR_LIN: # Check for Outage with
the defined threshold value.
                ep_outage_count += 1 # Outage counter.
        througputs.append(ep_th / env.episode_length) #
throughput append
        snrs_db.append(10*np.log10(np.maximum(1e-12,
np.mean(ep_snrs)))) # snr append
        outages.append(ep_outage_count / env.episode_length) #
outage count append
    return np.array(througputs), np.array(snrs_db),
np.array(outages)
```

Section 8: Evaluation Module: no-RIS scenarios- Direct link between BS and User

- This section evaluates the **baseline case where no RIS is used** in the communication system.
- Communication happens only through the **direct channel between the base station and the user**, without any reflected path.

- The environment is still reset and updated in the same way, so **channel dynamics remain consistent** with other evaluation cases.
- At each time step, the received signal quality is computed purely from the **direct link gain**.
- **Throughput** is calculated as the average spectral efficiency over the episode duration.
- **SNR** values are tracked and converted to dB for performance analysis.
- **Outage probability** is measured based on how often the SNR falls below the predefined threshold.
- This scenario serves as a **lower-bound reference**, helping to clearly show the performance gain achieved by RIS-assisted and RL-based approaches.

```
# Author Jay Gautam
# Analysis and Evaluation: Case when no RIS is used in the
communication.
# .. Base station and the user. Direct Channel.
# It will return Throughput, SNR and Outage count value.

def evaluate_noRIS(env, episodes=200):
    # Use only direct link; still evolve channels like env
does
    througputs = []
    snrs_db = []
    outages = []
    for ep in range(episodes):
        _ = env.reset()
        ep_th = 0.0
        ep_snrs = []
        ep_outage_count = 0
        for _step in range(env.episode_length):
            gain = np.abs(env.h_bd)**2 # Based on direct
channel BS->UserDevice
            snr = gain / env.noise_power
            r = np.log2(1 + snr)
            ep_th += r
            ep_snrs.append(snr)
            if snr < OUTAGE_SNR_LIN: # Count Outage
```

```
                ep_outage_count += 1
            env._update_channel()
        througputs.append(ep_th / env.episode_length)
        snrs_db.append(10*np.log10(np.maximum(1e-12,
np.mean(ep_snrs))))
        outages.append(ep_outage_count / env.episode_length)
    return np.array(througputs), np.array(snrs_db),
np.array(outages)
```

Section 9: Evaluation Module: DDPG Agent Performance

- This section evaluates the performance of the **trained DDPG agent** in the RIS-assisted wireless environment.
- During evaluation, the agent operates in **pure exploitation mode**, meaning no exploration noise is added to the actions.
- For each episode, the environment is reset and the agent selects RIS phase configurations based only on the learned policy.
- At every time step, the agent interacts with the environment and observes the resulting **reward, SNR, and outage condition**.
- **Throughput** is computed as the average spectral efficiency over the episode length.
- **SNR** values are averaged and converted to dB for performance comparison.
- **Outage probability** is calculated as the fraction of time steps where the SNR falls below the defined threshold.
- This evaluation reflects the **actual capability of the learned policy**, allowing a fair comparison with analytical and no-RIS baselines under identical channel dynamics.

```
# Author Jay Gautam
# Evaluation Module: Perfomrmance with DDPG Agent
# It will return Throughput, SNR and Outage count value.
def evaluate_agent(env, agent, episodes=200, device='cpu'):
    througputs = []
    snrs_db = []
    outages = []
    for ep in range(episodes):
```

```
        s = env.reset()
        ep_th = 0.0
        ep_snrs = []
        ep_outage_count = 0
        done = False # Done flag- check- copleted or not
completed
        while not done:
            a = agent.select_action(s, noise_scale=0.0) # DDPG
Agents action
            s2, r, done, info = env.step(a) # DDPG steps to
the environment with the action "a"
            ep_th += r
            snr = float(info["snr"])
            ep_snrs.append(snr)
            if snr < OUTAGE_SNR_LIN:
                ep_outage_count += 1
            s = s2 # update state.
        througputs.append(ep_th / env.episode_length) # Append
throughput
        snrs_db.append(10*np.log10(np.maximum(1e-12,
np.mean(ep_snrs)))) # Append SNR
        outages.append(ep_outage_count / env.episode_length) #
Append Outage count
    return np.array(througputs), np.array(snrs_db),
np.array(outages)
```

Section 10: Training: DDPG Agent on RIS assisted communication and Direct link- Logging for: SNR, Critics Learning statistics, Throughput and Outage

- In this section, the **DDPG agent is trained on the full RIS-assisted environment**, including both the RIS-reflected paths and the direct BS–user link.
- Training is performed over multiple episodes, where each episode represents a sequence of time-varying wireless channel realizations.
- At the start of every episode, the environment is reset and new channel states are generated.

- At each time step, the agent selects RIS phase actions based on the current channel state and interacts with the environment.
- The agent stores each interaction in the **replay buffer** and updates the actor and critic networks using mini-batch learning.
- Multiple **key performance indicators (KPIs)** are logged during training to understand both learning behavior and communication performance.

Logged metrics during training include:

- **Average reward per episode**, indicating overall learning progress.
- **Average SNR**, reflecting signal quality achieved by the learned RIS policy.
- **Critic Q-values**, used as a learning signal to monitor value function stability.
- **Throughput**, measured as average spectral efficiency per episode.
- **Outage probability**, showing robustness under dynamic channel conditions.
- Periodic logging during training helps track convergence behavior and detect instability or performance saturation.
- This training process allows the DDPG agent to gradually learn **adaptive RIS phase control policies** that perform well under realistic, time-varying wireless channels.

```
# Author Jay Gautam
# Training DDPG on RISEnvFull- All three Channels.

env = RISEnvFull(N=N_ELEMENTS, episode_length=EPISODE_LENGTH,
rho=RHO, noise_power=NOISE_POWER)
agent = DDPGAgent(env.state_dim, env.action_dim, device)

# Evaluation Metrics using DDPG
reward_history = []
accumulated_reward_history = [] # Reward accumulation over
episodes
snr_history = []
critic_q_history = [] # Q-value of the Critic Network
throughput_history = []
outage_history = []
```

```
start_time = time.time()
for ep in range(EPISODES): # Start the training with the
episodes
    s = env.reset() # reset the state in the start of new
episodes
    ep_reward = 0.0
    ep_snr_list = []
    q_values_in_episode = []
    ep_throughput = 0.0
    ep_outage_count = 0

    while True:
        a = agent.select_action(s) # take action with the
state (s = Channel)
        with torch.no_grad():
            s_t = torch.FloatTensor(s.reshape(1,
-1)).to(device)
            a_t = torch.FloatTensor(a.reshape(1,
-1)).to(device)
            q_val = agent.critic(s_t, a_t).cpu().item() # Q
value from the Value Network.
            q_values_in_episode.append(q_val) # Q value append

        s2, r, done, info = env.step(a)
        snr = float(info["snr"]) # get snr from the info flag

        ep_throughput += np.log2(1.0 + snr)
        if snr < OUTAGE_SNR_LIN: # Outage counter test
            ep_outage_count += 1

        agent.store((s, a, r, s2, float(done))) # Replay
Buffer
        agent.train(BATCH_SIZE)

        ep_reward += r
        ep_snr_list.append(snr)
        s = s2 # State update
        if done: # done flag check
```

```
            break

    # Per episode: Update metrics/KPI
    reward_history.append(ep_reward / env.episode_length)
    accumulated_reward_history.append(ep_reward)
    snr_history.append(np.mean(ep_snr_list) if ep_snr_list
else 0.0)
    critic_q_history.append(np.mean(q_values_in_episode) if
q_values_in_episode else 0.0)
    throughput_history.append(ep_throughput /
env.episode_length)
    outage_history.append(ep_outage_count /
env.episode_length)

    # Display Training progress data
    if (ep + 1) % 50 == 0 or ep == 0:
        elapsed = time.time() - start_time # Totl time of the
training.
        print(f"Episode {ep+1}/{EPISODES} | Avg
Reward={reward_history[-1]:.4f} | "
              f"Avg SNR={10*np.log10(np.maximum(1e-12,
snr_history[-1])):.2f} dB | "
              f"Mean Q={critic_q_history[-1]:.3f} | "
              f"Throughput={throughput_history[-1]:.3f} |
Outage={outage_history[-1]:.3f} |
Training_time={elapsed:.1f}s")

print("Training done. Total Training time:
{:.1f}s".format(time.time()-start_time))
```

Section 11: Visualization- Reward vs Episodes, Average Spectral efficiency vs Episode, Mean Critic_Q_value vs Episode, Average SNR (dB) vs Episode

- This section visualizes how the **DDPG agent learns over time** during training.

- Raw training metrics can be noisy due to stochastic channel behavior, so a **moving average window** is used to smooth the curves and show clear trends.

- Both raw values and averaged values are plotted to maintain transparency of learning behavior.

Visualized metrics include:

- **Average reward vs episodes:** Shows overall learning progress of the agent. A rising and stabilizing curve indicates improved RIS phase control.

- **Average spectral efficiency vs episodes:** Since Shannon capacity is used as the reward, this plot closely follows the reward curve and reflects communication performance improvement.

- **Mean critic Q-value vs episodes:** Represents the learning behavior of the critic network. Stable and smooth Q-values indicate healthy value-function learning.

- **Average SNR (dB) vs episodes:** Shows how signal quality improves as the agent learns better RIS configurations under time-varying channels.

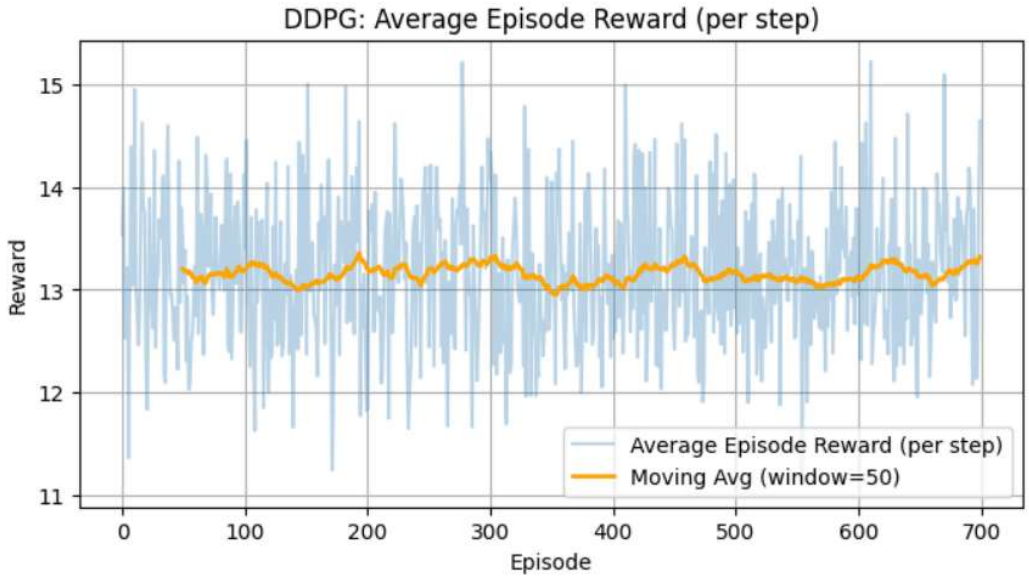
- The moving-average visualization helps identify **convergence behavior**, performance saturation, and possible training instability.

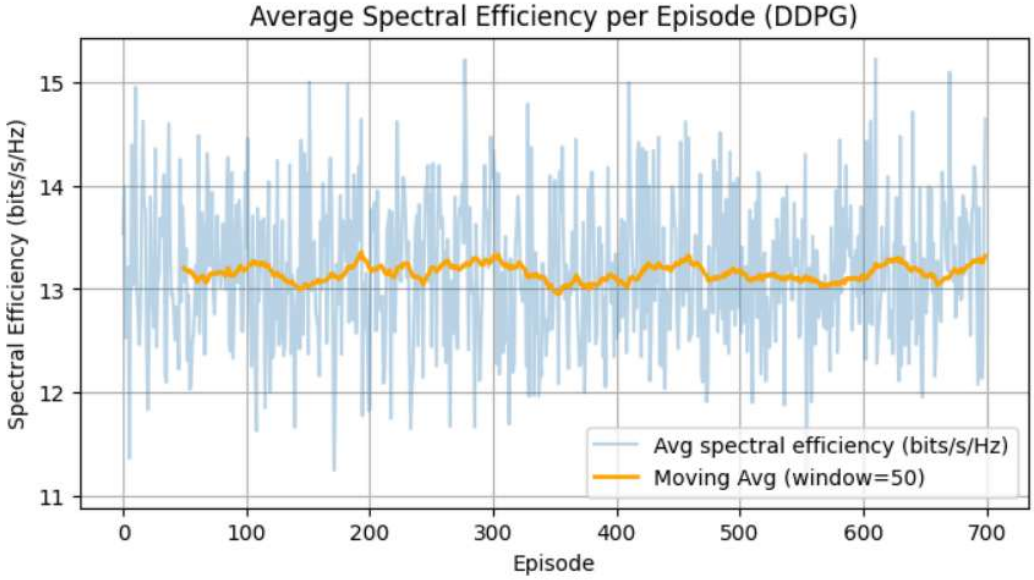
- Together, these plots provide clear insight into both **learning dynamics** and **wireless performance gains** achieved by the DDPG-based RIS control.

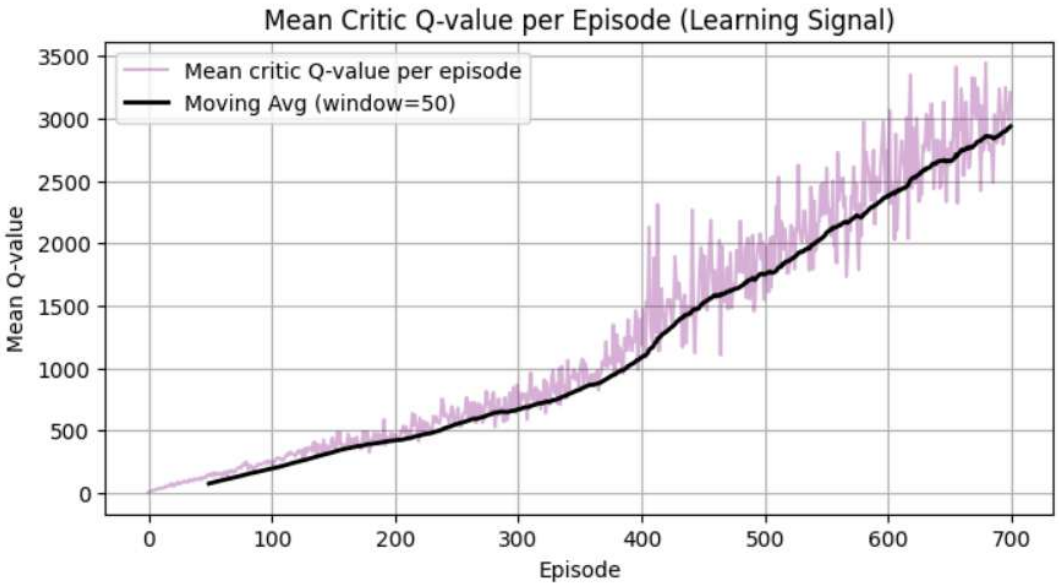
```
# Author: Jay Gautam
# Visualization: Reward, Average Spectral efficiency, Mean
Critic_Q_value and Average SNR (dB)
# Get the average of the metrics in a certain training window.

def moving_avg_with_alignment(x, window): # window-> size of
the window.
    y = np.array(x)
    n = len(y)
    x_raw = np.arange(n)
    if n == 0:
        return x_raw, y, np.array([]), np.array([])
    if n >= window and window > 0:
        mov = np.convolve(y, np.ones(window)/window,
mode='valid')
```

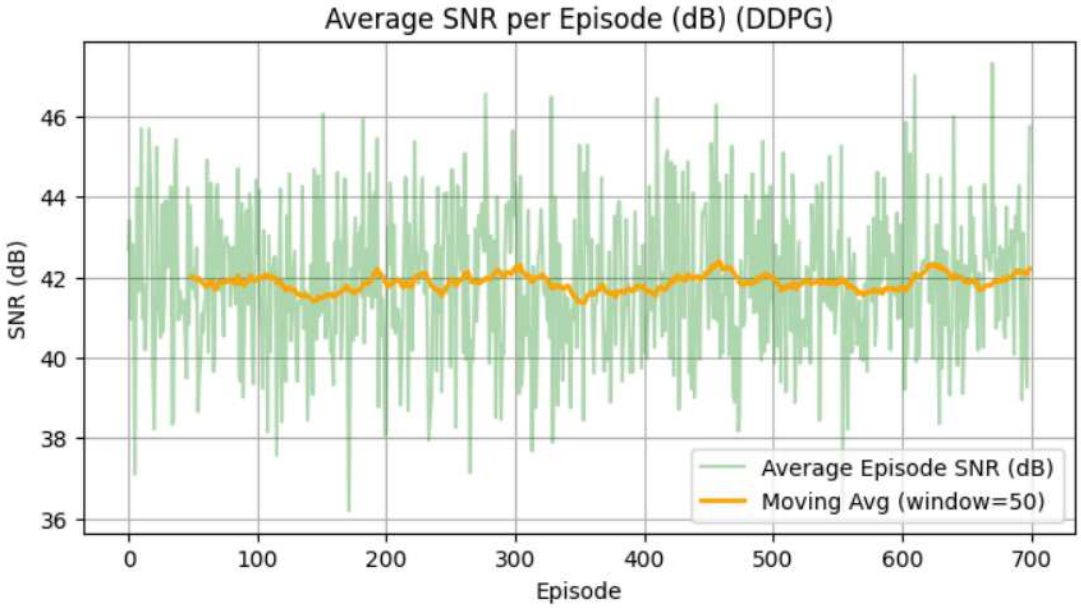
```
        x_mov = np.arange(window-1, n)
        return x_raw, y, x_mov, mov
    else:
        cma = np.cumsum(y) / (np.arange(n) + 1)
        x_mov = x_raw
        return x_raw, y, x_mov, cma


# Average Episode Reward (per step)- average of the window.
# Plot raw reward and moving average reward
x_raw, y_raw, x_mov, y_mov =
moving_avg_with_alignment(reward_history, MOV_WINDOW)
plt.figure(figsize=(8,4))
plt.plot(x_raw, y_raw, alpha=0.3, label='Average Episode
Reward (per step)')
if len(x_mov) > 0:
    plt.plot(x_mov, y_mov, color='orange', linewidth=2,
label=f'Moving Avg (window={MOV_WINDOW})')
plt.title("DDPG: Average Episode Reward (per step)")
plt.xlabel("Episode"); plt.ylabel("Reward")
plt.grid(True); plt.legend(); plt.show()

# Average Spectral Efficiency per Episode
# Plot: Raw spectrum effiiency and averaged spectral
efficiency over a window
# It should be same as rewardplot- as Shanon capacity is
chosen as reward function.
x_raw, y_raw, x_mov, y_mov =
moving_avg_with_alignment(throughput_history, MOV_WINDOW)
plt.figure(figsize=(8,4))
plt.plot(x_raw, y_raw, alpha=0.3, label='Avg spectral
efficiency (bits/s/Hz)')
if len(x_mov) > 0:
    plt.plot(x_mov, y_mov, color='orange', linewidth=2,
label=f'Moving Avg (window={MOV_WINDOW})')
plt.title("Average Spectral Efficiency per Episode (DDPG)")
plt.xlabel("Episode"); plt.ylabel("Spectral Efficiency
(bits/s/Hz)")
plt.grid(True); plt.legend(); plt.show()
```

```
# Mean Critic Q-value per Episode- Indicates Learning
characteristic of the Critic network
x_raw, y_raw, x_mov, y_mov =
moving_avg_with_alignment(critic_q_history, MOV_WINDOW)
plt.figure(figsize=(8,4))
plt.plot(x_raw, y_raw, alpha=0.3, label='Mean critic Q-value
per episode', color='purple')
if len(x_mov) > 0:
    plt.plot(x_mov, y_mov, color='black', linewidth=2,
label=f'Moving Avg (window={MOV_WINDOW})')
plt.title("Mean Critic Q-value per Episode (Learning Signal)")
plt.xlabel("Episode"); plt.ylabel("Mean Q-value")
plt.grid(True); plt.legend(); plt.show()

# Average SNR (dB) per Episode
snr_db = 10 * np.log10(np.maximum(1e-12,
np.array(snr_history)))
x_raw, y_raw, x_mov, y_mov =
moving_avg_with_alignment(snr_db.tolist(), MOV_WINDOW)
plt.figure(figsize=(8,4))
plt.plot(x_raw, y_raw, alpha=0.3, label='Average Episode SNR
(dB)', color='green')
if len(x_mov) > 0:
    plt.plot(x_mov, y_mov, color='orange', linewidth=2,
label=f'Moving Avg (window={MOV_WINDOW})')
plt.title("Average SNR per Episode (dB) (DDPG)")
plt.xlabel("Episode"); plt.ylabel("SNR (dB)")
plt.grid(True); plt.legend(); plt.show()
```


DDPG: Average Episode Reward (per step)
Reward
Episode
Average Episode Reward (per step)
Moving Avg (window=50)
15
14
13
12
11
0
100
200
300
400
500
600
700


Average Spectral Efficiency per Episode (DDPG)
Spectral Efficiency (bits/s/Hz)
Episode
Avg spectral efficiency (bits/s/Hz)
Moving Avg (window=50)
15
14
13
12
11
0
100
200
300
400
500
600
700





Interpretation of Training Results:

- The average episode reward shows an overall stable trend with natural fluctuations due to the time-varying wireless channel. The moving average indicates that the DDPG agent has learned a consistent policy rather than oscillating or diverging.
- The average spectral efficiency per episode closely follows the reward behavior, which is expected since the reward is based on Shannon capacity. This confirms that the agent is effectively optimizing communication performance.
- The mean critic Q-value increases steadily over training episodes, indicating that the critic network is learning a more confident and consistent value estimate for the state–action pairs. The smooth growth of the moving average suggests stable learning without collapse.

- The average SNR per episode remains stable and gradually improves, showing that the agent learns RIS phase configurations that maintain strong signal quality under dynamic channel conditions.
- Overall, these results indicate stable convergence of the DDPG agent, with no signs of training instability, divergence, or performance degradation, despite the stochastic and time-varying nature of the wireless environment.

Section 12: Evaluation of different strategies on the same environment- DDPG Agent, Closed-form, no-RIS

- This section performs a **fair and direct comparison** of different communication strategies under the **same environment dynamics**.
- All methods are evaluated using the **same RIS-assisted environment with a direct BS–user link**, ensuring consistency.
- Three strategies are compared:
    - **DDPG-based RIS control**
    - **Closed-form analytical RIS phase optimization**
    - **No-RIS (direct link only) baseline**
- Each strategy is evaluated over multiple episodes, where the wireless channels evolve dynamically across time.
- For every method, **throughput, average SNR, and outage probability** are collected as key performance indicators.
- The evaluation is performed without exploration noise, reflecting the **true performance** of each strategy.
- Using a common evaluation environment allows a **clean comparison**, highlighting the impact of learning-based control versus analytical optimization and no-RIS operation.

```
# Author Jay Gautam
# Evaluation of diferent startegies on the same environment.
## Evaluates.. DDPG Agent, Closed-form, no-RIS
# Env-> RISEnvFull: Contains all three channels.

print("Evaluation Process..")
```

```
eval_env = RISEnvFull(N=N_ELEMENTS,
episode_length=EPISODE_LENGTH, rho=RHO,
noise_power=NOISE_POWER)

t0 = time.time()
ddpg_th, ddpg_snr_db, ddpg_out = evaluate_agent(eval_env,
agent, episodes=EVAL_EPISODES, device=device)
cf_th, cf_snr_db, cf_out = evaluate_closed_form(eval_env,
episodes=EVAL_EPISODES)
nr_th, nr_snr_db, nr_out = evaluate_noRIS(eval_env,
episodes=EVAL_EPISODES)
print("Evaluation done in {:.1f}s".format(time.time() - t0)) #
Time calculator
```

Section13: Numerical Results- Throughput, SNR and Outage: DDPG, Closed_form, no-RIS

- This section presents the **numerical performance results** of all evaluated strategies.
- Three methods are compared:
    - **DDPG-based RIS control**
    - **Closed-form analytical RIS optimization**
    - **No-RIS (direct link only) baseline**
- For each method, results are collected over multiple evaluation episodes to ensure statistical reliability.
- The **mean and standard deviation** are reported for all key performance indicators.

Reported metrics include:

- **Average throughput**, measured in bits/s/Hz, indicating spectral efficiency.
- **Average SNR (dB)**, reflecting signal quality at the receiver.

- **Outage probability**, representing the fraction of time where the received SNR falls below the defined threshold.

- Reporting both mean and variance helps understand not only average performance, but also **stability and robustness** of each strategy.

- These numerical results clearly show the performance gap between **learning-based RIS control**, **analytical optimization**, and **no-RIS operation** under dynamic channel conditions.

```
# Jay Gautam
# Numerical Results: Throughput, SNR and Outage
#.. for DDPG, Closed_form, no-RIS
# Calculate the mean and std of the metrics: different methods

def summary_stats(arr):
    return np.mean(arr), np.std(arr)

methods = ["DDPG (RL)", "Closed-form", "No-RIS"]
means_th = [summary_stats(ddpg_th), summary_stats(cf_th),
summary_stats(nr_th)]
means_snr = [summary_stats(ddpg_snr_db),
summary_stats(cf_snr_db), summary_stats(nr_snr_db)]
means_out = [summary_stats(ddpg_out), summary_stats(cf_out),
summary_stats(nr_out)]

print("\nNumeric summary (over {} eval
episodes):".format(EVAL_EPISODES))
for i, m in enumerate(methods):
    print(f"{m}: Throughput mean={means_th[i][0]:.4f} ±
{means_th[i][1]:.4f} bits/s/Hz, "
          f"SNR mean={means_snr[i][0]:.2f} ±
{means_snr[i][1]:.2f} dB, "
          f"Outage fraction={means_out[i][0]:.3f} ±
{means_out[i][1]:.3f}")
```

Section 14:6G KPI: Visualization- Bar Plot: Throughput, SNR (dB), Outage

- This section presents a **visual comparison of key 6G performance indicators** using bar plots.
- The comparison includes three different strategies:
    - **DDPG-based RIS control**
    - **Closed-form analytical RIS optimization**
    - **No-RIS (direct link only) baseline**
- All results are obtained from the **same evaluation environment**, ensuring fair comparison.

Visualized KPIs include:

- **Throughput (spectral efficiency):**
  Shows how efficiently each strategy uses the available spectrum. Error bars indicate variability across evaluation episodes.
- **Average SNR (dB):**
  Highlights the improvement in received signal quality when RIS is intelligently controlled.
- **Outage probability:**
  Indicates the robustness of each strategy under dynamic channel conditions, with lower values representing more reliable communication.
- The use of bar plots makes performance differences **easy to interpret at a glance**, which is especially useful for high-level 6G discussions.
- These results clearly demonstrate the **benefits of RIS-assisted and learning-based control** compared to static analytical methods and no-RIS operation.

```
# Jay Gautam
# 6G KPI- Bar Plot: Throughput, SNR (dB), Outage

# Bar plot: throughputs by three methods- DDPG, Closed_form,
no-RIS
plt.figure(figsize=(6,4))
```

```
x = np.arange(len(methods))
means = [means_th[i][0] for i in range(len(methods))]
stds = [means_th[i][1] for i in range(len(methods))]

plt.bar(x, means, yerr=stds, capsize=6, alpha=0.8)
plt.xticks(x, methods)
plt.ylabel("Avg spectral efficiency (bits/s/Hz)")
plt.title("Throughput comparison")
plt.grid(axis='y', alpha=0.3)
plt.show()

# Bar plot: SNR (dB) by three methods- DDPG, Closed_form,
no-RIS
plt.figure(figsize=(6,4))
smeans = [means_snr[i][0] for i in range(len(methods))]
sstds = [means_snr[i][1] for i in range(len(methods))]
plt.bar(x, smeans, yerr=sstds, capsize=6, color='orange',
alpha=0.8)
plt.xticks(x, methods)
plt.ylabel("Average SNR (dB)")
plt.title("Average SNR (dB) comparison")
plt.grid(axis='y', alpha=0.3)
plt.show()

# Bar plot: outage by three methods- DDPG, Closed_form, no-RIS
plt.figure(figsize=(6,4))
omeans = [means_out[i][0] for i in range(len(methods))]
ostds = [means_out[i][1] for i in range(len(methods))]
plt.bar(x, omeans, yerr=ostds, capsize=6, color='red',
alpha=0.8)
plt.xticks(x, methods)
plt.ylabel("Outage fraction (SNR < {:.1f}
dB)".format(OUTAGE_SNR_DB))
plt.title("Outage comparison")
plt.grid(axis='y', alpha=0.3)
plt.show()
```

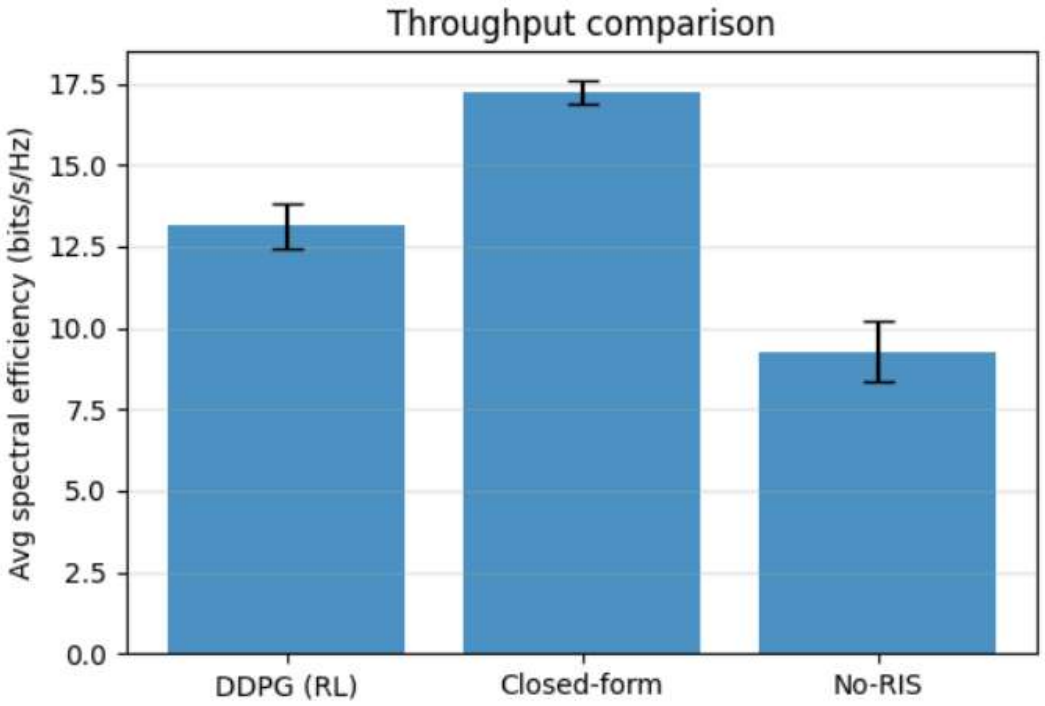
Results:

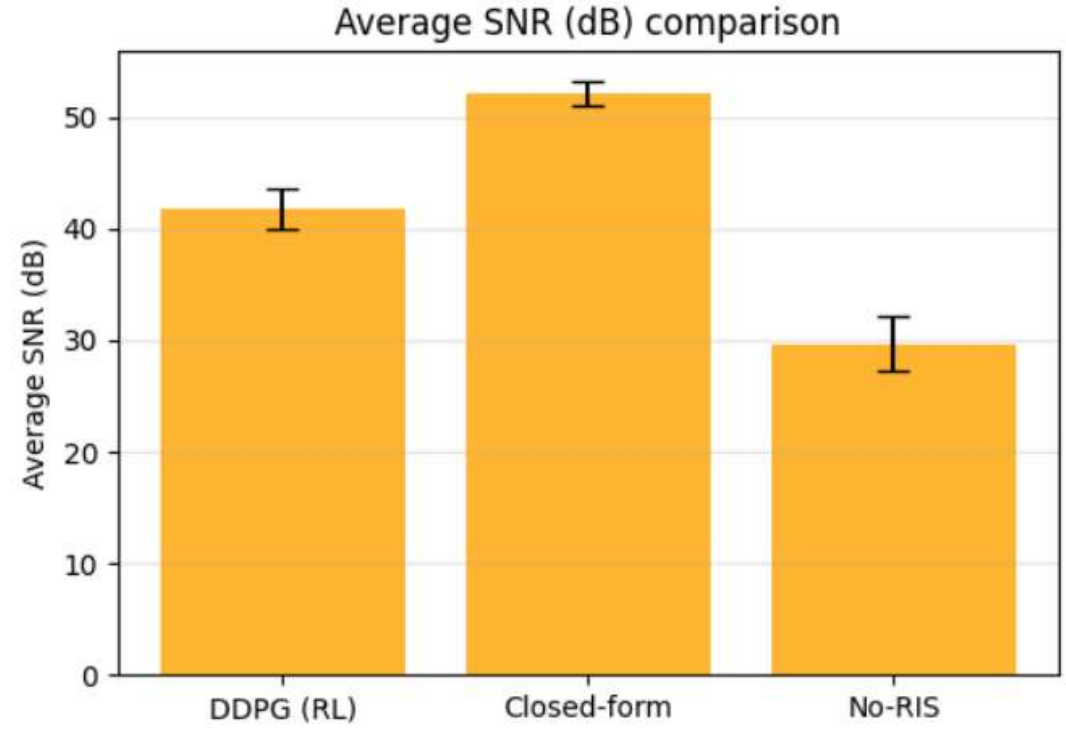
Numeric summary (over 100 eval episodes):

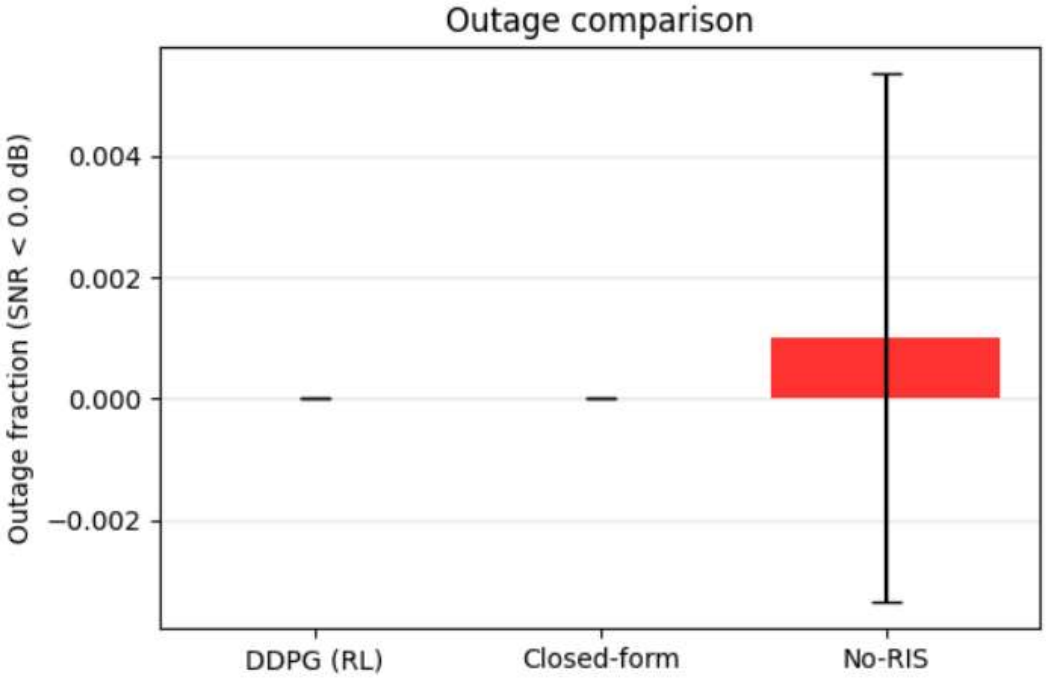
DDPG (RL): Throughput mean=13.1335 ± 0.6882 bits/s/Hz, SNR mean=41.85 ± 1.81 dB, Outage fraction=0.000 ± 0.000

Closed-form: Throughput mean=17.2309 ± 0.3636 bits/s/Hz, SNR mean=52.14 ± 1.08 dB, Outage fraction=0.000 ± 0.000

No-RIS: Throughput mean=9.2635 ± 0.9347 bits/s/Hz, SNR mean=29.70 ± 2.44 dB, Outage fraction=0.001 ± 0.004




Average SNR (dB) comparison
Average SNR (dB)
0
10
20
30
40
50
DDPG (RL)
Closed-form
No-RIS


Outage comparison
Outage fraction (SNR < 0.0 dB)
0.004
0.002
0.000
−0.002
DDPG (RL)
Closed-form
No-RIS

- The **closed-form method** achieves the **highest throughput and SNR**, which is expected as it directly aligns RIS phases using perfect instantaneous channel knowledge. This represents an **ideal analytical upper-bound** scenario.

- The **DDPG-based RIS control** achieves **significant performance gain compared to the no-RIS case**, both in terms of throughput and SNR.

    - Even without explicit channel optimization formulas, the RL agent learns to **adapt RIS phases effectively** under dynamic channel conditions.

    - Zero outage indicates that the learned policy is **stable and reliable** in the evaluated SNR regime.

- The **no-RIS baseline** shows the **lowest throughput and SNR**, with a small but non-zero outage probability. This clearly highlights the **benefit of RIS-assisted communication**, even without learning.

- While DDPG does not outperform the closed-form solution in this setup, it operates in a **model-free and adaptive manner**, making it more suitable for **realistic, time-varying, and partially known environments**, where closed-form solutions may not be available or practical.

Remark: RL Agent vs Closed Form solution- Why Closed-Form Wins Here, and Where RL Really Matters

- Closed-form RIS phase alignment is **analytically optimal** under perfect CSI, continuous phase control, and ideal hardware assumptions.

- In this setup, the environment exactly matches the assumptions under which closed-form solutions are derived.

- Reinforcement learning is **sample-based** and may not fully converge to the analytical optimum within a limited training resource.

- RL becomes advantageous when **CSI is imperfect, delayed, partial, or noisy**, where closed-form methods degrade quickly.

- Hardware constraints such as **quantized phase shifts, nonlinear amplitude–phase coupling, or mutual coupling** break closed-form optimality but can be learned by RL.

- RL naturally supports **complex objectives** (multi-user, fairness, energy efficiency, latency) where no closed-form solution exists.

- Closed-form inference is cheap, but **CSI acquisition and feedback overhead** can be costly in practice.

- RL has **high offline training cost**, but **very low online inference cost**, making it attractive for edge and low-latency deployment after pretraining.

- RL can be **computationally and energetically sustainable** in realistic 6G scenarios with proper plan of training and deployment.

Resource & Links

GitHub (code, implementations, updates):
https://github.com/jskgautam

LinkedIn (professional updates and discussions):
https://www.linkedin.com/in/jay-gautam-203362a2/recent-activity/shares/

Disclaimer

This work is an independent technical effort, developed using my own programming and AI skills. The implementation, experiments, and analysis presented here are entirely my own and are done using my own resources.

Theoretical understanding and practical insights have naturally been shaped by many publicly available research papers, courses, books, and discussions, as is common in academic and engineering practice. No proprietary code or confidential material has been used.

This document is shared for learning, discussion, and open technical exchange. It is not intended to claim ownership of any foundational theories or prior research.

In the future, for any of my content, if I use resources from others, I will certainly give proper credit. If any content raises concerns or objections, I kindly request that you contact me directly via LinkedIn, so the matter can be discussed and addressed constructively.