Sigil - Network Mesh Maker

AD18702 – Reinforcement Learning

ASSIGNMENT II

Authored by:
AKASH S - 2127210502004
DHANUSH G - 2127210502012
KARTHICK L - 2127210502021
RASHEEM KHAN R -2127210502035

Sigil – Network Mesh Maker

Introduction

Sigil is an innovative project aimed at optimizing mobile network connectivity in areas affected by floods. In times of natural disasters, reliable communication can be the difference between safety and risk. Sigil uses reinforcement learning techniques to help users identify mobile network hotspots with the best possible data speeds, ensuring that they stay connected even when network stability is compromised.

"In the face of disaster, the ability to communicate can mean the difference between life and death. Our mobile networks ensure that people stay connected, informed, and supported when they need it the most."

- **Barack Obama**

Description

Sigil is intended to be used during floods, where mobile network stability is crucial. Using Reinforcement Learning, the system analyzes mobile data speeds across different areas and suggests the best locations for optimal network access. This information is presented to users through a responsive Next.js frontend, making it accessible and easy to use during emergencies.

Algorithms:

The project, we leverage two reinforcement learning techniques—**Deep Q-Network (DQN)** and **Q-Learning**—to optimize the decision-making process for selecting mobile network hotspots in emergency scenarios like floods.

Deep Q-Network (DQN)

DQN is an advanced version of Q-Learning that combines a neural network with reinforcement learning. In DQN, a neural network approximates the Q-value function, which estimates the expected reward for taking a particular action in a given state. This approach is particularly useful when dealing with large or continuous state spaces.

Key Components:

- **Q-values Prediction:**

  The DQN model uses a neural network to predict the Q-values for each possible action in a given state. It consists of a sequential model with dense layers, trained to minimize the mean squared error (MSE) between predicted Q-values and target Q-values.

- **Exploration vs. Exploitation:**

  The agent balances between exploring new actions (`epsilon-greedy` policy) and exploiting the current knowledge (choosing the action with the highest predicted Q-value).

- **Experience Replay**:

  The agent stores experiences (state, action, reward, next state, done) in memory and trains using random samples from this memory to break the correlation between consecutive updates.

- **Bellman Equation**:

  The agent updates its Q-values using the Bellman equation, where it combines the immediate reward with the discounted future reward.

The neural network has three layers:

- **Input Layer**: Accepts the state representation.
- **Hidden Layers**: Two fully connected layers with 24 neurons each using ReLU activation.
- **Output Layer**: Outputs Q-values for all possible actions.

Q-Learning

Q-Learning is a simpler, tabular reinforcement learning algorithm where the agent maintains a Q-table to store the value of each state-action pair. It updates the table using the following update rule:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max aQ(s',a) - Q(s,a)]Q(s,a)$$

Here, α is the learning rate, γ is the discount factor, r is the reward, and s' is the next state.

Key Components:

- Action Selection**: The agent selects actions based on either exploration or exploitation. In the exploration phase, the agent picks a random action, while during exploitation, it selects the action with the highest Q-value from the Q-table.**
- State Transitions**: After taking an action, the agent transitions to a new state and receives a reward based on the network performance at that location. This information is used to update the Q-value for the selected action.**

Reward Function:

The reward is calculated based on the improvement in upload and download speeds, weighted as follows:

$$Reward = 0.7 \times (Predicted\ Download\ Speed - Current\ Download\ Speed) + 0.3 \times (Predicted\ Upload\ Speed - Current\ Upload\ Speed)$$

Combining DQN and Q-Learning:

In the project, DQN predicts network performance (upload/download speeds) in unexplored locations, while Q-Learning optimizes the movement between locations by selecting the best direction (e.g., up, down, left, right) based on historical data.

Code:

q-learning.py

```python
import random
from pymongo import MongoClient

def calculate_direction(current_position, next_position):
    """Calculate movement directions based on position changes."""
    delta_x = next_position[0] - current_position[0]
    delta_y = next_position[1] - current_position[1]
    direction = []

    if delta_y > 0:
        direction.append(f"Move north by {abs(delta_y)} meter(s)")
    elif delta_y < 0:
        direction.append(f"Move south by {abs(delta_y)} meter(s)")

    if delta_x > 0:
        direction.append(f"Move east by {abs(delta_x)} meter(s)")
    elif delta_x < 0:
        direction.append(f"Move west by {abs(delta_x)} meter(s)")

    return direction

class SignalRLAgent:
    def __init__(self, actions, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.q_table = {}
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.actions = actions
    def get_q_value(self, state, action):
        return self.q_table.get((tuple(state), action), 0.0)

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(self.actions)  # Explore: random action
        q_values = [self.get_q_value(state, action) for action in self.actions]
        max_q = max(q_values)
        return self.actions[q_values.index(max_q)]  # Exploit: choose best action

    def update_q_value(self, state, action, reward, next_state):
        max_q_next = max([self.get_q_value(next_state, a) for a in self.actions])
        old_q = self.get_q_value(state, action)
        new_q = old_q + self.alpha * (reward + self.gamma * max_q_next - old_q)
        self.q_table[(tuple(state), action)] = new_q

    def update_agent_with_speed_data(self, current_position, current_upload_speed,
current_download_speed, mesh_collection):
```

```
        action = self.choose_action(current_position)

        next_position = current_position[:]
        if action == 'left':
            next_position[0] -= 1
        elif action == 'right':
            next_position[0] += 1
        elif action == 'up':
            next_position[1] += 1
        elif action == 'down':
            next_position[1] -= 1

        next_position_data = mesh_collection.find_one({'position': next_position})

        if next_position_data:
            predicted_download_speed = next_position_data.get('download_speed', 0.0)
            predicted_upload_speed = next_position_data.get('upload_speed', 0.0)
        else:
            predicted_download_speed = 0.0
            predicted_upload_speed = 0.0

        reward = self.get_reward(current_download_speed, current_upload_speed,
predicted_download_speed, predicted_upload_speed)

        self.update_q_value(current_position, action, reward, next_position)

        direction = calculate_direction(current_position, next_position)

        return {
            'recommended_action': action,
            'next_position': next_position,
            'direction': direction,
            'predicted_download_speed': predicted_download_speed,
            'predicted_upload_speed': predicted_upload_speed,
        }

    def get_reward(self, current_download_speed, current_upload_speed,
predicted_download_speed, predicted_upload_speed):
        download_speed_change = predicted_download_speed - current_download_speed
        upload_speed_change = predicted_upload_speed - current_upload_speed

        reward = (0.7 * download_speed_change) + (0.3 * upload_speed_change)
        return reward

client = MongoClient('mongodb://localhost:27017/'
db = client['signal_map_db']
mesh_collection = db['mesh_points']
if __name__ == '__main__':
    actions = ['left', 'right', 'up', 'down']
```

```
    q_agent = SignalRLAgent(actions)

    current_position = [0, 0]
    current_upload_speed = 10.0
    current_download_speed = 20.0

    result = q_agent.update_agent_with_speed_data(
        current_position, current_upload_speed, current_download_speed,
mesh_collection
    )

    print(f"Next Position: {result['next_position']}, Action:
{result['recommended_action']}, "
          f"Predicted Download Speed: {result['predicted_download_speed']}, "
          f"Predicted Upload Speed: {result['predicted_upload_speed']}, Directions:
{result['direction']}")
```

dqn.py

```
import numpy as np
import random
from collections import deque
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

class DQNAgent:
    def __init__(self, state_size, action_size, alpha=0.001, gamma=0.95,
epsilon=1.0, epsilon_min=0.01, epsilon_decay=0.995):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.learning_rate = alpha
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))  # Output layer
        model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate))
```

```
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def choose_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        q_values = self.model.predict(state)
        return np.argmax(q_values[0])

    def replay(self, batch_size=32):
        if len(self.memory) < batch_size:
            return

        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = reward + self.gamma *
np.amax(self.model.predict(next_state)[0])  # Bellman equation
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def save(self, dqn_saved):
        self.model.save(dqn_saved)

    def load(self, dqn_saved):
        self.model.load_weights(dqn_saved)

if __name__ == '__main__':
    state_size = 4
    action_size = 4
    agent = DQNAgent(state_size, action_size)

    state = np.reshape([1, 0, 0, 0], [1, state_size])
    action = agent.choose_action(state)
    print(f"Chosen Action: {action}")
```

app.py

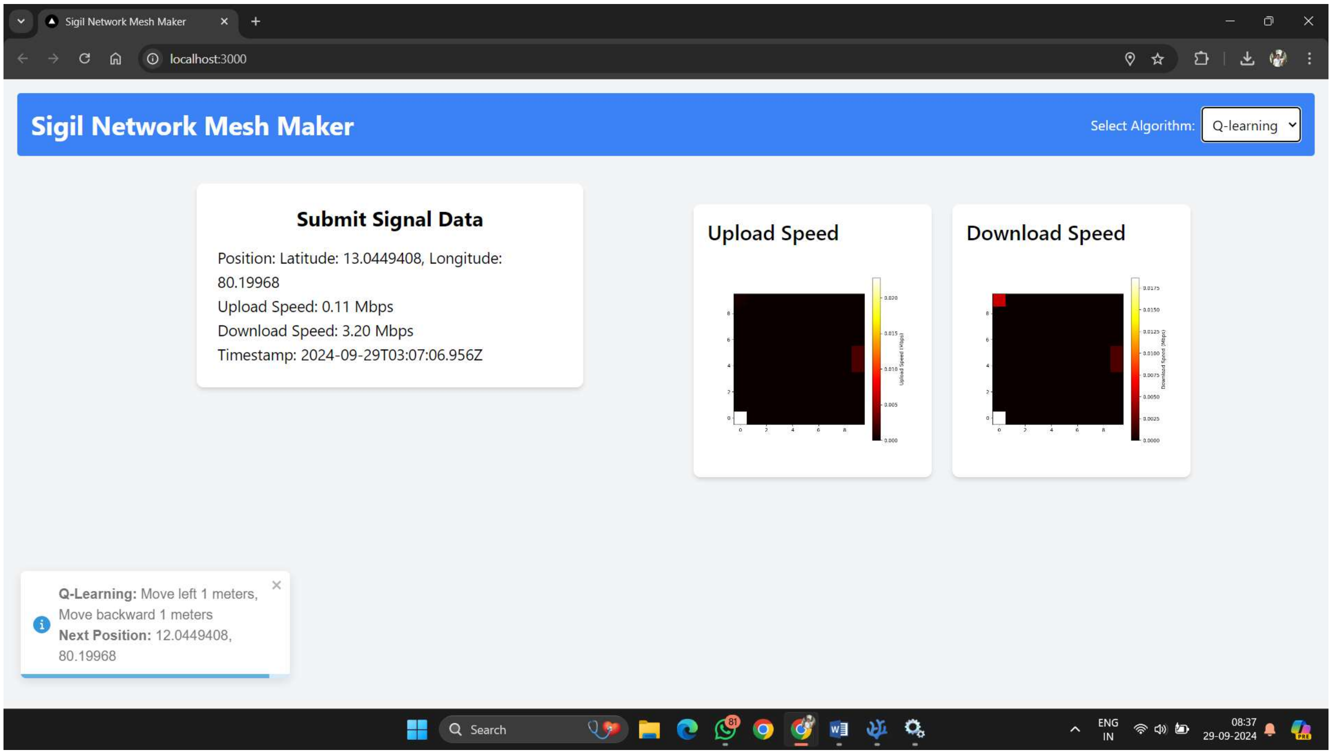
```
from flask import Flask, request, jsonify
from flask_cors import CORS
from pymongo import MongoClient
import matplotlib
matplotlib.use('Agg')  # Use the Agg backend for Matplotlib
import matplotlib.pyplot as plt
import numpy as np
import io
import base64
from q_learning_agent import SignalRLAgent  # Q-learning agent
from dqn_agent import DQNAgent  # DQN agent
import random
from waitress import serve

app = Flask(__name__)
CORS(app)

client = MongoClient('mongodb://localhost:27017/')
db = client['signal_map_db']
mesh_collection = db['mesh_points']

ACTIONS = ['left', 'right', 'up', 'down']
state_size = 2
action_size = len(ACTIONS)

q_agent = SignalRLAgent(ACTIONS)  # Q-learning agent
dqn_agent = DQNAgent(state_size, action_size)  # DQN agent

def insert_mesh_point(mesh_point):
    mesh_collection.insert_one(mesh_point)

@app.route('/submit_data', methods=['POST'])
def submit_data():
    data = request.json
    position = data.get('position')
    upload_speed = data.get('upload_speed')
    download_speed = data.get('download_speed')
    timestamp = data.get('timestamp')

    if not all([position, upload_speed, download_speed, timestamp]):
        return jsonify({"error": "Missing required fields"}), 400

    mesh_point = {
        "position": position,
        "upload_speed": upload_speed,
        "download_speed": download_speed,
        "timestamp": timestamp
```

```
    }
    insert_mesh_point(mesh_point)

    return jsonify({"status": "success"}), 201

@app.route('/get_q_learning_recommendation', methods=['GET'])
def get_q_learning_recommendation():
    try:
        mesh_points = list(mesh_collection.find({}))
        if not mesh_points:
            return jsonify({"error": "No data available"}), 404

        current_point = mesh_points[-1]
        current_position = current_point.get('position')
        current_upload_speed = current_point.get('upload_speed')
        current_download_speed = current_point.get('download_speed')

        if current_position is None or current_upload_speed is None or
current_download_speed is None:
            return jsonify({"error": "Invalid data in the database"}), 400

        result = q_agent.update_agent_with_speed_data(
            current_position, current_upload_speed, current_download_speed,
mesh_collection
        )

        recommendation = {
            "current_position": current_position,
            "recommended_action": result['recommended_action'],
            "next_position": result['next_position'],
            "predicted_upload_speed": result['predicted_upload_speed'],
            "predicted_download_speed": result['predicted_download_speed']
        }

        return jsonify(recommendation), 200

    except Exception as e:
        print(f"Error occurred: {e}")
        return jsonify({"error": "Server encountered an issue", "details": str(e)}),
500

@app.route('/get_dqn_recommendation', methods=['GET'])
def get_dqn_recommendation():
    try:
        mesh_points = list(mesh_collection.find({}))
        if not mesh_points:
            return jsonify({"error": "No data available"}), 404

        current_point = mesh_points[-1]
```

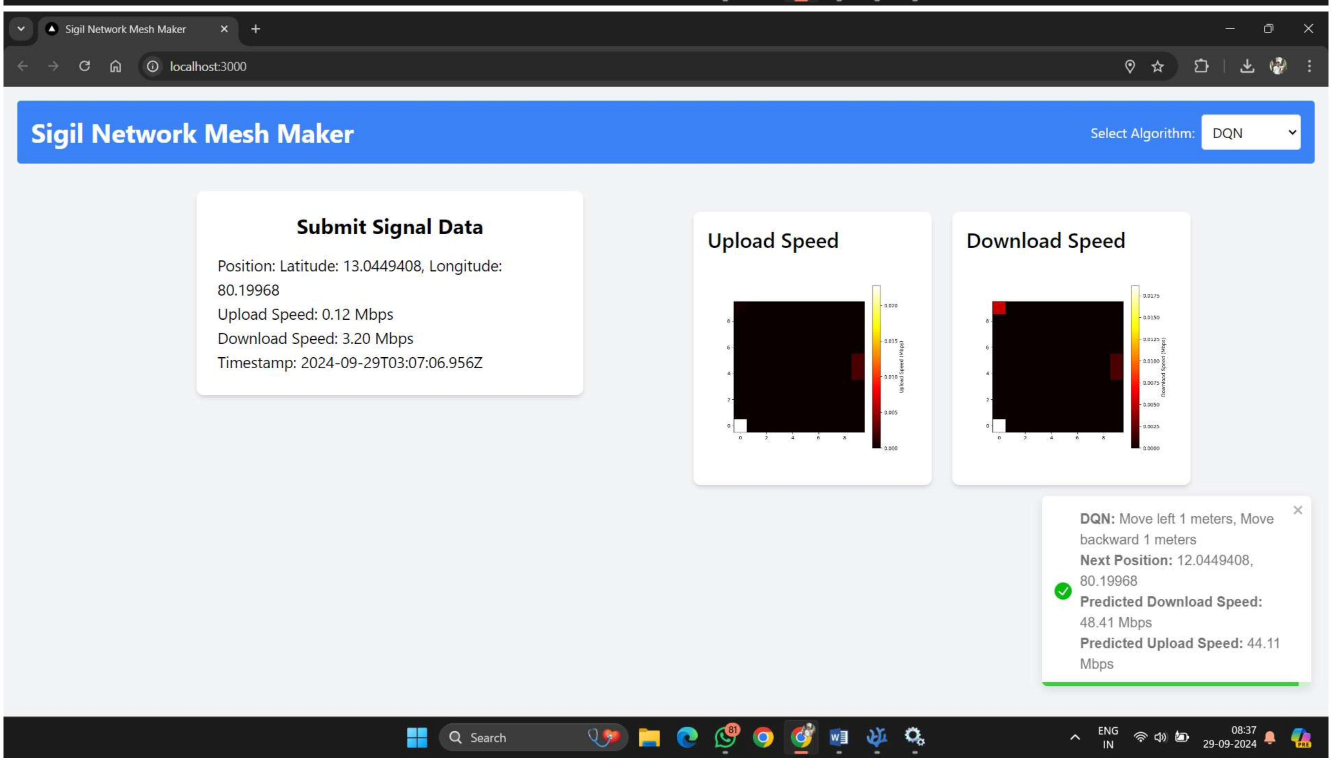
```
        current_position = current_point['position']
        current_upload_speed = current_point['upload_speed']
        current_download_speed = current_point['download_speed']

        state = np.reshape(current_position, [1, state_size])

        action = dqn_agent.choose_action(state)

        next_position = current_position[:]
        if action == 0:
            next_position[0] -= 1
        elif action == 1:
            next_position[0] += 1
        elif action == 2:
            next_position[1] += 1
        elif action == 3:
            next_position[1] -= 1

               next_upload_speed = random.uniform(5, 50)
        next_download_speed = random.uniform(5, 50)


        next_state = np.reshape(next_position, [1, state_size])
        dqn_agent.remember(state, action, next_download_speed, next_state, False)


        dqn_agent.replay()

        recommendation = {
            "current_position": current_position,
            "recommended_action": ACTIONS[action],
            "next_position": next_position,
            "predicted_upload_speed": next_upload_speed,
            "predicted_download_speed": next_download_speed
        }

        return jsonify(recommendation), 200

    except Exception as e:
        print(f"Error occurred: {e}")
        return jsonify({"error": "Server encountered an issue", "details": str(e)}),
500
@app.route('/get_heatmap', methods=['GET'])
def get_heatmap():
    try:
        mesh_points = list(mesh_collection.find({}))
        if not mesh_points:
            return jsonify({"error": "No data available"}), 404
```

```python
        x = [point['position'][0] for point in mesh_points]
        y = [point['position'][1] for point in mesh_points]
        upload_speed = [point['upload_speed'] for point in mesh_points]
        download_speed = [point['download_speed'] for point in mesh_points]


        plt.figure(figsize=(6, 6))
        heatmap_upload, xedges, yedges = np.histogram2d(x, y, bins=(10, 10),
weights=upload_speed, density=True)
        plt.imshow(heatmap_upload.T, origin='lower', cmap='hot',
interpolation='nearest')
        plt.colorbar(label='Upload Speed (Mbps)')


        buf = io.BytesIO()
        plt.savefig(buf, format='png')
        buf.seek(0)
        img_data_upload = base64.b64encode(buf.getvalue()).decode('utf-8')
        buf.close()
        plt.clf()


        plt.figure(figsize=(6, 6))
        heatmap_download, xedges, yedges = np.histogram2d(x, y, bins=(10, 10),
weights=download_speed, density=True)
        plt.imshow(heatmap_download.T, origin='lower', cmap='hot',
interpolation='nearest')
        plt.colorbar(label='Download Speed (Mbps)')

        buf = io.BytesIO()
        plt.savefig(buf, format='png')
        buf.seek(0)
        img_data_download = base64.b64encode(buf.getvalue()).decode('utf-8')
        buf.close()

        return jsonify({
            "upload_heatmap_image": img_data_upload,
            "download_heatmap_image": img_data_download
        }), 200

    except Exception as e:
        print(f"Error occurred: {e}")          return jsonify({"error": "Server
encountered an issue", "details": str(e)}), 500

if __name__ == '__main__':
    serve(app, host='0.0.0.0', port=5000)
```

Output:



Sigil Network Mesh Maker
localhost:3000
Sigil Network Mesh Maker
Select Algorithm:
Q-learning
Submit Signal Data
Position: Latitude: 13.0449408, Longitude: 80.19968
Upload Speed: 0.11 Mbps
Download Speed: 3.20 Mbps
Timestamp: 2024-09-29T03:07:06.956Z
Upload Speed
Download Speed
Q-Learning: Move left 1 meters, Move backward 1 meters
Next Position: 12.0449408, 80.19968
Search
ENG
IN
08:37
29-09-2024



Sigil Network Mesh Maker
localhost:3000
Sigil Network Mesh Maker
Select Algorithm:
DQN
Submit Signal Data
Position: Latitude: 13.0449408, Longitude: 80.19968
Upload Speed: 0.12 Mbps
Download Speed: 3.20 Mbps
Timestamp: 2024-09-29T03:07:06.956Z
Upload Speed
Download Speed
DQN: Move left 1 meters, Move backward 1 meters
Next Position: 12.0449408, 80.19968
Predicted Download Speed: 48.41 Mbps
Predicted Upload Speed: 44.11 Mbps
Search
ENG
IN
08:37
29-09-2024

Real-World Applications

The combination of **Deep Q-Network (DQN)** and **Q-Learning** in this project has several practical applications, especially in emergency situations like floods where mobile networks are crucial for communication. Here are a few real-world scenarios where such an approach could be applied:

1. **Disaster Management and Response**:
   - In the aftermath of natural disasters such as floods, hurricanes, or earthquakes, mobile network infrastructure can become disrupted. Our system helps emergency responders and civilians navigate toward areas with the strongest available mobile signals, enabling communication for rescue efforts and information dissemination.
2. **Network Optimization in Rural Areas**:
   - In remote or rural areas where mobile network coverage can be inconsistent, this system could dynamically suggest the best locations for setting up temporary communication hotspots (like portable towers or signal boosters), ensuring better connectivity for residents.
3. **Smart City Planning**:
   - As cities grow smarter, mobile networks need to be optimized for areas with high data usage. This reinforcement learning system can analyze network usage patterns in real-time and suggest optimal tower placements or signal improvements to maintain strong coverage during events such as concerts or emergencies.
4. **Military Operations**:
   - In defense operations where real-time communication is vital, such a system can help soldiers or operatives move toward areas with better network reception, ensuring continuous connection to central command, even in rugged or unpredictable terrain.
5. **Vehicular Communication Systems**:
   - For autonomous vehicles or mobile units (like drones or trucks), having real-time access to mobile networks is critical for navigation and data transmission. The system could guide vehicles toward better signal coverage areas, ensuring uninterrupted connectivity while on the move.

Conclusion

This project demonstrates the power of combining **Deep Q-Network (DQN)** and **Q-Learning** algorithms to solve real-world challenges in mobile network optimization during emergencies. By dynamically suggesting locations with stronger signals, this system has the potential to save lives, improve communication during critical situations, and optimize network usage in various environments.

The use of reinforcement learning techniques ensures that the system learns over time, continuously improving its ability to navigate and predict optimal network conditions. This adaptability makes it an ideal solution for rapidly changing and unpredictable environments, such as during natural disasters, where reliable mobile connectivity can be the difference between life and death.