A Project Report

On

Transport Layer Multipath TCP with Deep Reinforcement Learning

By

MUGDHA GUPTA

2021B3A72724H

Under the Supervision of

Prof PARESH SAXENA

SUBMITTED IN THE FULFILLMENT OF THE REQUIREMENTS OF CS F366:

LABORATORY PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

(HYDERABAD CAMPUS)

(December 2024)

Acknowledgments

My deepest gratitude goes to my mentors, Professor Paresh Saxena and Shravan Kumar from the Department of Computer Science at BITS Pilani, Hyderabad Campus. Their persistent advice and insightful oversight were invaluable as I worked on the "Transport Layer Multipath with Deep Reinforcement Learning" project. Their guidance, assistance, and recommendations about relevant research material and methods significantly improved the project's efficacy and provided me with a first-hand research experience.

Furthermore, I would like to thank BITS Pilani Hyderabad Campus for enabling me to participate in this research as required by my academic program. I've been able to put my academic knowledge to use and deepen my grasp of computer networks and deep reinforcement learning owing to this practical experience.

Birla Institute of Technology and Science – Pilani

Hyderabad Campus

Certificate

This is to certify that the project report titled “Transport Layer Multipath TCP with Deep Reinforcement Learning”, submitted by Ms. Mugdha Gupta (ID No. 2021B3A7PS2724H) in partial fulfillment of the requirements of the course CS F366: Laboratory Project Course, embodies the work done by her under my supervision and guidance.

Date: 12-12-2024

(Prof. Paresh Saxena)
BITS-Pilani, Hyderabad Campus

ABSTRACT

The multipath transmission control protocol (MPTCP) is a transport mechanism that allows peers to use multiple TCP subflows via their existing IP addresses at the same time. Each subflow encounters the bottleneck connection status of its path, so scheduling an outgoing packet with the best subflow is crucial to multipath performance. Poor scheduling decisions prevent users from utilizing the pooling potential of available subflows, even though good scheduling decisions can significantly boost throughput. DRL algorithms like the Soft-Actor Critic, Model Agnostic Meta Learning, and RL2 (Fast Reinforcement Learning via Slow Reinforcement Learning) are used to analyze packet scheduling in the MPTCP in detail to address this issue. These algorithms can help optimize the decision-making process by taking into account observed network states and rewards.

Table of Contents

Introduction

Each subflow that makes up an MPTCP connection operates differently based on the path's current state. One of the primary issues with multipath transport protocols is packet scheduling, which determines how to divide packets across many channels.

The path condition is determined by the bottleneck link's throughput capacity, inflight packets, congestion window, and round trip time. To manage multipath diversity, a scheduler must select the optimal subflow to transmit each packet to. Scheduling choices have a big influence on improving multipath performance since the scheduler's objective dictates what the optimal subflow is. [1].

An MPTCP scheduler must generate significant packet routing schedules based on the current condition of pathways (sub-flows) in terms of loss rate, bandwidth, and jitter to optimize the network thoroughput. MPTCP extends TCP by enabling a single byte stream to be split into many byte streams and transmitted via multiple different network paths or subflows. The performance of each subflow inside an MPTCP connection depends on the path's state, which includes variables like throughput capacity, packet loss rate, and queue delay.

Unreliable packet scheduling can lead to critical networking issues like out-of-order (OFO) packets, which require the receiver to maintain a large queue to reorganize the received packets, and head-of-line (HoL) blocking, where packets scheduled on the low-latency path must wait for packets on the high-latency path to ensure in-order delivery. This project's objective is to investigate and assess MPTCP scheduling on dynamic networks, including cellular networks, and to suggest an MPTCP schema that can successfully handle dynamic networks' performance limitations. [2]

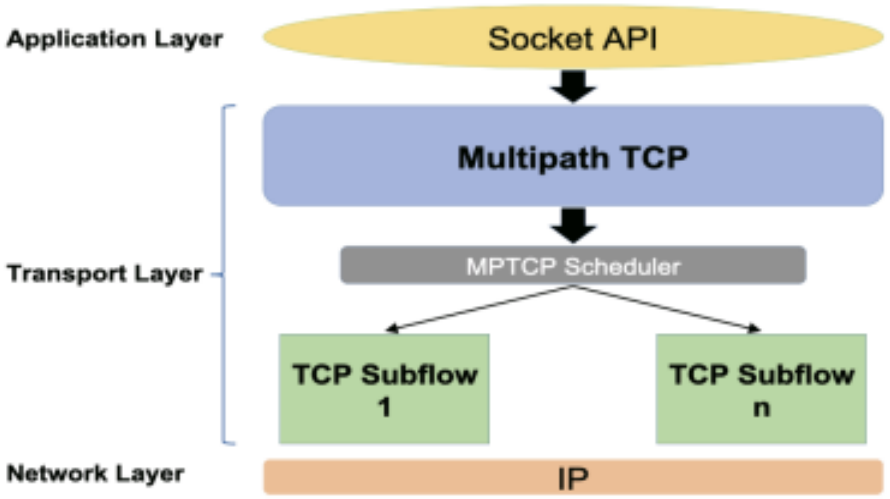


Fig. 1: Multi-path TCP architecture. [3]

This project implements a Deep Reinforcement Learning-based MPTCP scheduler. Using deep reinforcement learning (DRL) techniques, a neural network is trained only via experience to develop the control strategy for packet scheduling. It uses a full reward function that schedules packets by taking into account the round-trip time, congestion window, and in-flight packets.

Because DRL algorithms may optimize for changing congestion windows, RTT, and in-flight packets, they aid in adaptive decision-making by allowing packet routing decisions to be modified in real-time as network conditions change. DRL can adapt to these changes and keep producing wise choices even as the number of possible routes or network factors increases.

Deep Reinforcement Learning algorithms are more resilient than rule-based schedulers because they can learn over time and generalize to unknown network situations. 3 such DRL algorithms, Soft Actor-Critic (SAC), MAML, and RL^2 are ideally suited for this application because of their ability to manage continuous operations (such as regulating packet flow rates) and optimize efficiency while balancing exploration and exploitation using an entropy term.

SAC Implementation

Key Components of a DRL-Based MPTCP Scheduler:

1. **Environment:**

   The environment is defined by the network conditions, including round-trip times (RTT), in-flight packets, and numerous pathways with varying congestion windows (CWND). The scheduler receives these dynamic parameters. The state space includes the previously described metrics (CWND, RTT, and in-flight packets) for any available path.

2. **Action:**

   The action space is used to select a path for the next packet (or flow). For instance, for every packet, the DRL agent has the option of delivering it via path 1 or path 2.

   Possible actions include the ability to switch between paths in response to real-time situations or more accurate control over the quantity of packets sent on each path.

3. **Policy and Value Networks (Actor and Critic Networks):**

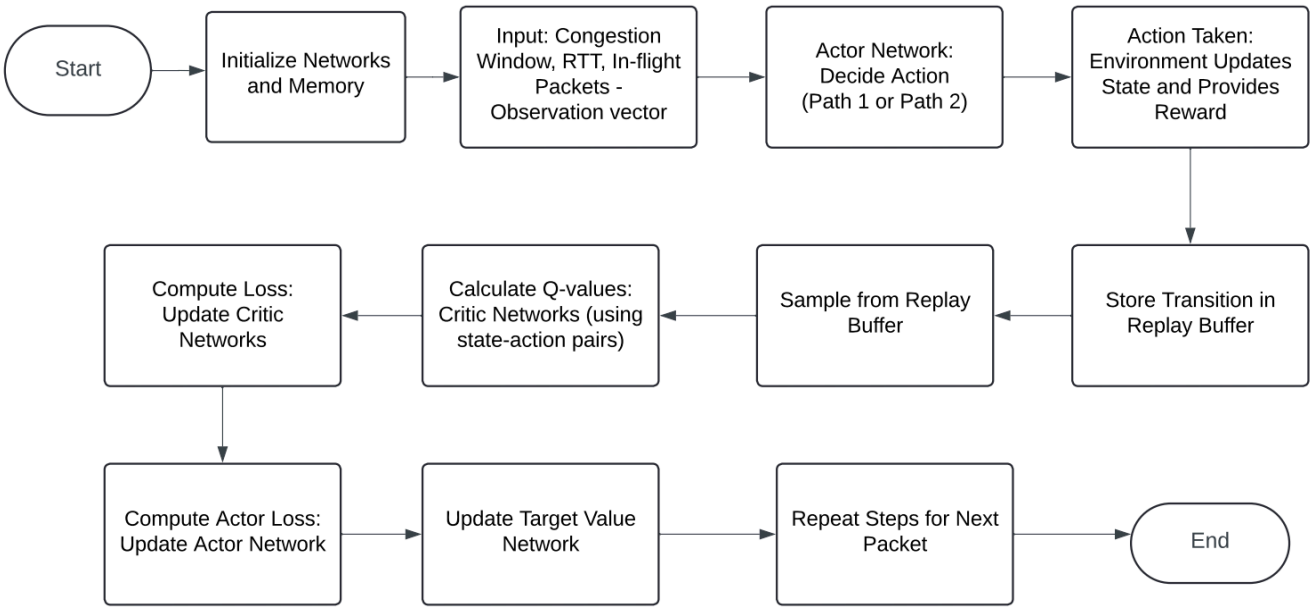
   The actor-network uses the state (CWND, RTT, in-flight packets) as input to create a probability distribution across actions (which path to take). The critic network evaluates the actor's action by calculating the Q-value, or expected future reward, for the action selected in the given conditions.

   The SAC algorithm adapts the actor and critic networks to the reward of the environment by using policy gradients to make better decisions over time.

4. **Reward:**

   The reward is often determined by throughput, delay, packet loss, or a combination of these variables. Reduced RTT and increased throughput may result in beneficial incentives. Delays or packet loss may result in negative rewards.The scheduler must select the most efficient data transmission mechanism while accounting for network conditions to maximize cumulative rewards.

BLOCK DIAGRAM



Flow of the Algorithm:

1. Initialize Networks and Memory:

Input: None (initialization step).

Process:

- Actor-Network: Set up to produce actions according to the observation vector, which is the input state.
- Critical Networks (Critic 1 and Critic 2): Started to provide state-action pair Q-values.
- Value Network: Set up to output the current state's value.
- Target Value Network: Used for steady target Q-value computations during training, this network is initially a replica of the Value Network.
- Replay Buffer: Set up to hold previous state, action, reward, and new state transitions.

Output: Initialisation of the networks and replay buffer.

2. Input: Congestion Window, RTT, In-flight Packets - Observation vector

Input: The network environment's current state.

Process: Collect the observation vector, which includes the following: ● Congestion Window: Congestion window size.

Round Trip Time (RTT): The amount of time it takes for a packet to go to the destination and back.

● In-Flight Packets: The total number of packets in motion at any one time.

The observation vector (congestion window, RTT, in-flight packets) **is the output.**

3. Actor Network Decision

Input: Step 2 observation vector.

Process:

After processing the observation vector, the actor-network produces an action (a_t) that denotes the choice of path (Path 1 or Path 2).

Output: Action (a_t) denotes the data packet's selected path.

4. Execute Action

Input: Action (a_t), which is the Actor Network's selected path.

Process: By processing this action, the environment modifies the dynamics of the network.

Output: A new state that reflects the effect of the action on the network, including updated values for the congestion window, RTT, and in-flight packets, as well as reward (r_t).

5. Store Transition in Replay Buffer

Input: Action, reward, new state, done flag (if the episode has ended), and previous state.

Process: Store the state, action, reward, new state, and completed transition into the Replay Buffer.

Output: The Replay Buffer stores the transition.

6. Sample from Replay Buffer

Condition: There are enough transitions in the replay buffer for a batch.

Process: Take a sample of the Replay Buffer's transitions.

Output: A collection of transitions (done, new state, action, reward, and state).

7. Critic Network Evaluation

A sampled batch of states and actions is the input.

Process:

Determine the Q-values for the state-action pairs in the sampled batch using the Critic Networks (Critic 1 and Critic 2) method. The expected return for doing the specified action in the specified state is represented by these Q-values.

b. Target Value Network: Determines the value of the sampled batch's subsequent state, or new state. This is how the target Q-value is calculated.

Output: Target Q-value and Critic Networks Q-values.

8. Compute Loss and Update Critic Networks

Input:
a. Critic Networks' Q-values, which were calculated in Step 7.
b. Target Q-values, which were calculated in Step 7 using the Target Value Network.

Process:
c. Determine each Critic Network's loss function. The mean squared error (MSE) between the Q-values and the intended Q-values is usually used for this.
d. Use the gradients from the loss function to update the Critic Networks.

Updated Critic Networks (Critic 1 and Critic 2) are the output.

9. Compute Actor Loss and Update Actor Network

Input:

- Current policy (actor network-generated actions).
- Critic Networks Q-values (from Step 7).

The Actor Network calculates the log probability of actions.

Process: Determine the actor loss, taking into account the policy gradient and add an entropy element to promote exploration.

Utilising the gradients obtained from the actor loss, update the actor network.

Actor Network is the updated output.

10. Update Target Value Network

Weights from the Target Value Network and the existing Value Network are input.

The soft update rate is controlled by the update parameter tau.

Process: Use the following formula to soft update the Target Value Network weights:

Target Value Network Weights= $\tau \times$Value Network Weights + $(1-\tau) \times$Target Value Network Weights

Updated Target Value Network is the output.

11. Repeat Steps for Next Packet

- **Condition:** Until the episode is finished, repeat the "Observe Environment" instructions for every new packet.

12. End of Episode

- Input: Determine whether the episode is over.
- Output: If finished, start a new episode and reset the environment.

Reward Function:

The reward function should be designed to incentivise the agent to choose the path with the least amount of expected delay and congestion.

Consideration is given to how effective the chosen course is in contrast to the alternative.

This reward function's objective is to encourage selecting the path with the lowest total score of weighted metrics, which typically corresponds to a path with superior network performance.

```python
def calculate_reward(path_1, path_2, action):
    # Calculate a score for each path (example: weighted sum)
    weights = [0.5, 0.3, 0.2]  # Example weights for RTT, CWND, and in-flight packets
    score_1 = sum(w * p for w, p in zip(weights, path_1))
    score_2 = sum(w * p for w, p in zip(weights, path_2))

    # Determine the reward based on the chosen action and relative scores
    if action == 0:  # Chose Path 1
        reward = score_2 - score_1
    else:  # Chose Path 2
        reward = score_1 - score_2
```

Let's assume the following values for the three metrics (RTT, CWND, and in-flight packets) for both paths in order to demonstrate how the reward function operates with numerical values:

Path 1 Metrics: RTT: 80 ms CWND: 10 In-flight packets: 6 packets

Path 2 Metrics: RTT: 100 ms CWND: 12 In-flight packets: 8 packets

Normalization:To make sure the values are on the same scale, we must normalise them before calculating the reward. To keep things simple, let's assume that the numbers are now between 0 and 1 after we have normalised them.

Normalized Path 1 Metrics: RTT_norm: 0.8 CWND_norm: 0.5 In-flight packets_norm: 0.3

Normalized Path 2 Metrics: RTT_norm: 1.0 CWND_norm: 0.6 In-flight packets_norm: 0.4

Weights: RTT weight: 0.5 CWND weight: 0.3 In-flight packets weight: 0.2

Scoring: The weighted total of the normalised metrics is used to determine the score for each path.

Score for Path 1: Score_1 = (0.5 * 0.8) + (0.3 * 0.5) + (0.2 * 0.3) Score_1 = 0.4 + 0.15 + 0.06 Score_1 = 0.61

Score for Path 2: Score_2 = (0.5 * 1.0) + (0.3 * 0.6) + (0.2 * 0.4) Score_2 = 0.5 + 0.18 + 0.08 Score_2 = 0.76

Reward Calculation: If we choose Path 1 (action == 0), the reward is calculated as: Reward = Score_2 - Score_1 Reward = 0.76 - 0.61 → Reward = 0.15

If we choose Path 2 (action != 0), the reward is calculated as: Reward = Score_1 - Score_2 Reward = 0.61 - 0.76 → Reward = -0.15

Interpretation: In this example, selecting Path 1 results in a positive reward, suggesting that, according to the weighted sum of the normalised metrics, Path 1 is better than Path 2. The reward is negative if we select Path 2, meaning that, based on the same criterion, Path 2 is not as desirable as Path 1. Therefore, the path with the lowest aggregate score for the weighted metrics is chosen using the reward function as a guide.

Environment Initialization:

```
class NetworkEnv(gym.Env):
    def __init__(self, data):
        super(NetworkEnv, self).__init__()
        self.data = data
        self.current_step = 0
        self.observation_space = spaces.Box(low=0, high=np.inf, shape=(3,), dtype=np.float32)
        self.action_space = spaces.Discrete(2)  # Two possible paths: 0 for Path 1, 1 for Path 2

    def step(self, action):
        current_data = self.data.iloc[self.current_step]

        # Parameters for both paths
        path_1 = [current_data['P1_RTT'], current_data['P1_CWND'], current_data['P1_inflight']]
        path_2 = [current_data['P2_RTT'], current_data['P2_CWND'], current_data['P2_inflight']]

        # Calculate reward based on the chosen action
        reward = calculate_reward(path_1, path_2, action)

        # Increment step
        self.current_step += 1
        done = self.current_step >= len(self.data)

        # Next state is based on the selected path parameters
        new_state = np.array([path_1 if action == 0 else path_2], dtype=np.float32).squeeze()

        return new_state, reward, done, {}
```

Networks:

```
class ActorNetwork(nn.Module):
    def __init__(self, alpha, input_dims, fc1_dims=256, fc2_dims=256, n_actions=2):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dims, fc1_dims)
        self.fc2 = nn.Linear(fc1_dims, fc2_dims)
        self.mu = nn.Linear(fc2_dims, n_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=alpha)
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        mu = torch.tanh(self.mu(x))  # Output between -1 and 1
        return mu
```

Agent Training:

The agent receives several episodes of training. The agent goes through several phases in each episode, including:

- Selecting an action through the actor-network.
- The environment provides feedback in the form of a reward and a new state.
- The replay buffer is used to store the experience.
- Replay buffer experiences are sampled in order to update the networks.

```
class SACAgent:
    def __init__(self, alpha, beta, input_dims, n_actions, gamma=0.99, tau=0.005, buffer_size=100000, batch_size=64):
        self.gamma = gamma
        self.tau = tau
        self.batch_size = batch_size
        self.memory = ReplayBuffer(buffer_size, input_dims, n_actions)

        # Networks
        self.actor = ActorNetwork(alpha, input_dims, n_actions=n_actions)
        self.critic_1 = CriticNetwork(beta, input_dims, n_actions)
        self.critic_2 = CriticNetwork(beta, input_dims, n_actions)
        self.value = ValueNetwork(beta, input_dims)
        self.target_value = ValueNetwork(beta, input_dims)

        self.update_network_parameters(tau=1)

    def choose_action(self, state):
      state = torch.tensor([state], dtype=torch.float32).to(self.actor.device)
      logits = self.actor(state)
      action_probs = torch.softmax(logits, dim=-1)
      action_distribution = torch.distributions.Categorical(action_probs)
      action = action_distribution.sample()
      return action.item()


    def store_transition(self, state, action, reward, new_state, done):
        self.memory.store_transition(state, action, reward, new_state, done)

    def learn(self):
        if self.memory.mem_cntr < self.batch_size:
            return
```

Loss Computation:

- **Value Loss**: The value network is trained to predict the target value of a state, which is calculated using rewards and the value of the next state. The difference between the target value and the predicted value is the value loss.
- **Critic Loss**: The critic networks estimate the Q-values of the state-action pairs; the target for the critic networks is the reward plus the discounted value of the next state. The critics' loss is the mean squared error between the target and predicted Q-values.
- **Actor Loss:** The actor is trained to maximise the expected Q-value of its selected actions while preserving exploration. The advantage function is used to calculate the loss.

Visualizing the Rewards:

```
env = NetworkEnv(dataset)
agent = SACAgent(alpha=0.001, beta=0.001, input_dims=3, n_actions=2)
train_sac(env, agent, episodes=100)
```

```
Episode 2/100 - Total Reward: 1.2296033150611372
Episode 3/100 - Total Reward: -1.506571520324154
Episode 4/100 - Total Reward: 1.798845349174171
Episode 5/100 - Total Reward: 4.849266697580164
Episode 6/100 - Total Reward: 3.1542001060649
Episode 7/100 - Total Reward: -6.0731930910512535
Episode 8/100 - Total Reward: -5.173126579283965
Episode 9/100 - Total Reward: 0.13468801711104006
Episode 10/100 - Total Reward: -4.346080609358543
Episode 11/100 - Total Reward: 5.815910899912529
Episode 12/100 - Total Reward: -7.6697290021582685
Episode 13/100 - Total Reward: -13.147588870476138
Episode 14/100 - Total Reward: -16.890283266799962
Episode 15/100 - Total Reward: -3.7043036826865254
Episode 16/100 - Total Reward: -0.2512982829439213
Episode 17/100 - Total Reward: 15.868659455569748
Episode 18/100 - Total Reward: 8.865308952394985
Episode 19/100 - Total Reward: 3.9806269751394416
Episode 20/100 - Total Reward: 15.856106203238149
Episode 21/100 - Total Reward: 7.288451410876132
Episode 22/100 - Total Reward: 12.9209382148658
Episode 23/100 - Total Reward: 15.635509418554307
Episode 24/100 - Total Reward: 16.352165747861097
Episode 25/100 - Total Reward: 24.569727879429827
Episode 26/100 - Total Reward: 24.92301409220383
Episode 27/100 - Total Reward: 15.486549477026674
Episode 28/100 - Total Reward: 25.24383539430474
Episode 29/100 - Total Reward: 12.83879214897076
Episode 30/100 - Total Reward: 26.75659580239458
```

Model Agnostic Meta-Learning

Model-Agnostic Meta-Learning (MAML), is designed to create models that can quickly adapt to new tasks or environments with minimal retraining. This algorithm is particularly powerful in scenarios where the environment is dynamic and unpredictable.

Why Meta-RL Might Be Suitable:

1. **Rapid Adaptation**:

   Meta-RL algorithms are designed to enable rapid adaptation to new or changing environments. In your case, if the network conditions (like congestion, round trip time, etc.) change frequently, a Meta-RL model could quickly adjust the scheduler's policy to optimize packet transfer across paths.

2. Learning to Learn:

   Instead of learning a specific policy for a static environment, Meta-RL algorithms learn how to quickly adapt to new situations. This means that the scheduler wouldn't just learn how to route packets in one specific scenario but would learn a strategy that works well across a wide range of network conditions.

3. Reduced Training Time:

   Once trained, a meta-RL model can adapt to new network conditions with minimal additional training. This is especially beneficial if the network environment changes too frequently to allow for complete retraining each time.

Algorithm

Algorithm 3 MAML for Reinforcement Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using $f_{\theta_i'}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
11: **end while**

1. Initialize Policy Network

This step involves defining the architecture of the policy network, which will be used by the agent to make decisions based on the input (network state).

The input layer takes in the network state, which in this case is composed of three key parameters: RTT, CNWD, and Inflight Packets. The network state input is a vector containing these three parameters for both paths being considered by the agent.

The output layer generates the final prediction—in this case, the action probabilities for path selection.

2. Initialize MAML Algorithm

This step is crucial for setting up the meta-learning process in the MAML (Model-Agnostic Meta-Learning) framework.

Set Learning Rates

- **α (Alpha): Inner Loop Learning Rate**
  - **Purpose**: Alpha represents the learning rate used for task-specific adaptation. It controls how much the model's parameters (policy network weights) are

adjusted for each task during the inner loop of MAML.

- **Inner Loop**: This is where the model adapts to a specific task by taking a few gradient steps to minimize the loss for that task. A smaller learning rate ensures smoother and more controlled updates for task-specific fine-tuning.

β (Beta): Outer Loop Learning Rate

- **Purpose**: Beta represents the learning rate used in the meta-update (outer loop) to adjust the base model's parameters after task-specific adaptations. It controls how much the model learns from the combined experience of all tasks.
- **Outer Loop**: After the model has adapted to a batch of tasks, the outer loop updates the global policy using the accumulated knowledge from task-specific adaptations. Beta controls how much influence the meta-loss (average loss across tasks) has on updating the global policy.

Define Parameters

- **Number of Inner Steps**: Task-Specific Adaptation
  - **Purpose**: This defines how many steps of gradient descent are taken during task-specific adaptation (inner loop).
  - E**xample**: Setting the number of inner steps to 5 means the model will perform 5 iterations of gradient descent for each task during adaptation.

Number of Meta Iterations: Overall Meta-Training

- **Purpose**: This defines how many outer loop iterations are performed for meta-training. Each meta-iteration involves sampling a batch of tasks, performing task-specific adaptation for each, and then updating the global policy based on the accumulated meta-loss.

3. Sample Tasks

This step involves creating multiple task environments that the model will use for training in the MAML algorithm.

Generate Task Environments: multiple instances of the NetworkEnv class are created, with each instance representing a different task.

MAML aims to learn a meta-policy that can be generalized across different tasks. By sampling multiple tasks, the model learns how to adapt quickly to new conditions, even ones it hasn't seen before.

Each task presents a unique challenge, requiring the model to adjust its parameters in

task-specific ways during the inner loop of adaptation.

4. Meta-Training Loop

- **Task-specific adaptation**

```
def adapt(self, task, initial_params):
    adapted_params = initial_params

    # Perform task-specific adaptation (inner loop)
    for _ in range(self.num_inner_steps):
        # Get a trajectory from the task using the standalone function
        states, actions, rewards, _ = sample_trajectory(task, self.policy)

        # Compute loss
        loss = self.compute_loss(states, actions, rewards, adapted_params)

        # Compute gradients with allow_unused=True
        grads = torch.autograd.grad(loss, adapted_params, create_graph=True, allow_unused=True)

        # Update parameters using gradient descent
        adapted_params = [param - self.alpha * grad if grad is not None else param
                          for param, grad in zip(adapted_params, grads)]

    return adapted_params
```

- **Compute loss and update task parameters**

```
def compute_loss(self, states, actions, rewards, params):
    log_probs = []
    for state, action in zip(states, actions):
        action_probs = self.policy(state)


        action_probs = action_probs.squeeze()
        if len(action_probs.shape) == 0:
            action_probs = action_probs.unsqueeze(0)

        log_prob = torch.log(action_probs[action])
        log_probs.append(log_prob)

    log_probs = torch.stack(log_probs)
    rewards = torch.FloatTensor(rewards)

    # Compute the loss
    loss = -torch.sum(log_probs * rewards)
    return loss
```

- **Meta-update**

After task-specific adaptation (the inner loop) has been completed for each task, the **meta-update** step takes place. The goal of the meta-update is to aggregate the knowledge gained from each task and update the global policy parameters (θ) in a way that improves

the model's ability to generalize across tasks.

```
def meta_update(self, tasks):
    meta_loss = 0
    total_rewards = 0  # To accumulate rewards across tasks

    for task in tasks:
        initial_params = list(self.policy.parameters())

        # Adapt the policy to the task
        adapted_params = self.adapt(task, initial_params)

        # Get a new trajectory with the adapted policy
        states, actions, rewards, _ = sample_trajectory(task, self.policy)

        # Compute the loss for the adapted policy
        loss = self.compute_loss(states, actions, rewards, adapted_params)

        # Accumulate the meta-loss
        meta_loss += loss

        # Calculate the total reward for the task
        total_rewards += sum(rewards)  # Sum of rewards for this task

    # Perform meta-update using the accumulated meta-loss
    self.optimizer.zero_grad()
    meta_loss.backward()
    self.optimizer.step()
```

7. Repeat Until Convergence

This is the final step in the MAML meta-training process, where the model continues to go through meta-iterations until it either converges (i.e., the policy stops improving) or the training reaches a predefined number of iterations.

Convergence: Convergence occurs when the policy stops improving across meta-iterations, meaning that the meta-loss and task-specific performance metrics (such as rewards) no longer show significant improvement.

How to Detect Convergence:

Stable Meta-Loss: If the meta-loss (the total loss accumulated across tasks) reaches a plateau, it indicates that the policy has likely converged.

Stable Task-Specific Rewards: If the average reward across tasks stabilizes and stops increasing, the policy may no longer be improving.

Why Convergence Matters:

Once the model has converged, continuing meta-training may not result in significant improvements, and it may even overfit to the tasks seen during training. Therefore, stopping training at convergence saves computational resources while ensuring the policy has been generalized well.

Visualizing Rewards

```
Meta iteration 1/50 starting...
Average reward for this meta-update: -2.95
Meta iteration 1/50 complete
Meta iteration 2/50 starting...
Average reward for this meta-update: -0.02
Meta iteration 2/50 complete
Meta iteration 3/50 starting...
Average reward for this meta-update: 2.70
Meta iteration 3/50 complete
Meta iteration 4/50 starting...
Average reward for this meta-update: -1.92
Meta iteration 4/50 complete
Meta iteration 5/50 starting...
Average reward for this meta-update: 2.89
Meta iteration 5/50 complete
Meta iteration 6/50 starting...
Average reward for this meta-update: 1.26
Meta iteration 6/50 complete
Meta iteration 7/50 starting...
Average reward for this meta-update: 2.59
Meta iteration 7/50 complete
Meta iteration 8/50 starting...
Average reward for this meta-update: 4.00
Meta iteration 8/50 complete
Meta iteration 9/50 starting...
Average reward for this meta-update: 4.63
Meta iteration 9/50 complete
Meta iteration 10/50 starting...
Average reward for this meta-update: 7.57
Meta iteration 10/50 complete
Meta iteration 11/50 starting...
Average reward for this meta-update: 3.90
Meta iteration 11/50 complete
Meta iteration 12/50 starting...
Average reward for this meta-update: 6.96
Meta iteration 12/50 complete
Meta iteration 13/50 starting...
Average reward for this meta-update: 8.87
Meta iteration 13/50 complete
```

RL^2

(Fast Reinforcement Learning via Slow Reinforcement Learning)

RL² aims to make an agent that learns more quickly when facing a new task. Instead of creating an algorithm that learns fast by itself, the idea is to use a neural network (specifically a recurrent neural network or RNN) to "learn how to learn" from experience. The goal is to take knowledge from previous tasks and use that to solve new tasks more efficiently.

How does RL² work?

Two Levels of Learning:

- **Fast Learning**: This is the quick learning the agent does while interacting with a specific task. It's the kind of learning that happens as it explores, tries different actions, and learns from rewards.
- **Slow Learning**: This happens over many tasks, where the agent updates its general knowledge of how to learn new tasks efficiently. This "meta-learning" is the slow process of learning how to perform fast learning better.

The Role of Recurrent Neural Networks (RNNs):

- The key idea is to use an RNN, which is a neural network designed to handle sequential data. The RNN will receive inputs (observations, actions, rewards, etc.) over time and update its internal state.
- **Why an RNN?** Because RNNs can remember past information, they are well-suited to handle the history of actions and rewards across multiple episodes. This lets the RNN "remember" what it has learned in previous interactions and carry that memory forward to new tasks.

Information Flow:

- In each task, the RNN gets observations (like the state of the environment), the action taken, the reward received, and whether the episode ended. This input helps the RNN decide what to do next.
- The difference is that the RNN retains this information across episodes, meaning it gets

better at solving the task as it gathers more information.

How Tasks Are Structured:

- RL^2 works by exposing the agent to a set of tasks (which could be different MDPs). Each task is presented for a series of episodes, and the agent must quickly learn to perform well on these tasks.
- After experiencing several tasks, the agent improves its general ability to adapt to new, unseen tasks because its RNN has learned to quickly learn.

Learning Across Episodes: The RNN-based agent stores "fast learning" information in its internal states. Instead of resetting the internal state after each episode, the state is carried over between episodes in the same task. This allows the RNN to remember what it learned from one episode and apply it to the next one, making it a quick learner for that specific task.

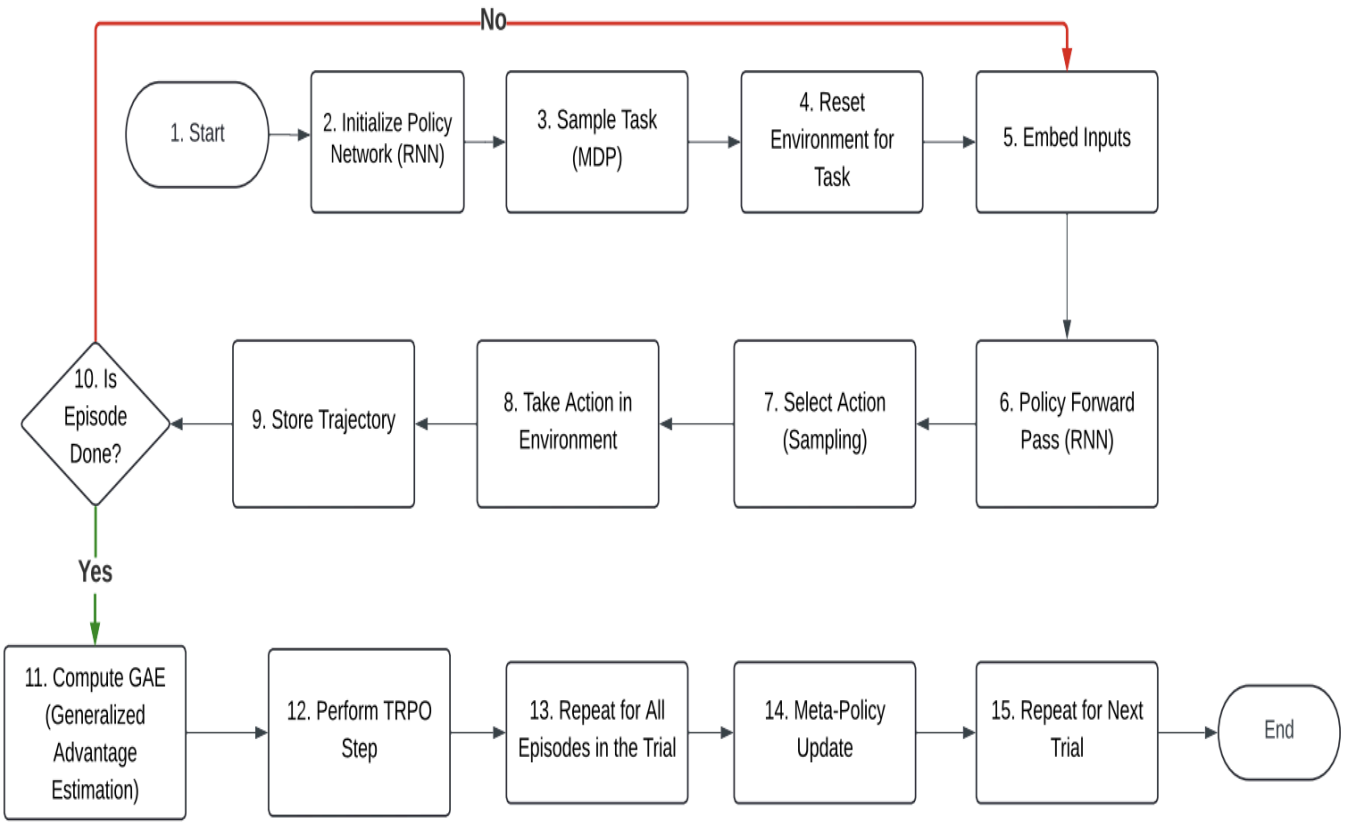
Example:

Imagine you're trying to teach a robot to find a hidden treasure in different mazes. You don't want to reprogram it every time it enters a new maze. Instead, you want it to "learn how to explore mazes" in general. You expose the robot to several mazes, and over time, it gets better and better at quickly figuring out where the treasure might be, based on what it learned from previous mazes. After enough practice, when you give it a brand-new maze, it can adapt quickly without needing a long time to explore. That's essentially what RL^2 does—it teaches the agent how to learn new tasks faster by retaining knowledge from previous tasks.

Key Points:

- **Fast learning** happens during a task (e.g., a specific maze).
- **Slow learning** happens over many tasks (e.g., several different mazes).
- The RNN remembers important information across episodes, allowing it to get better at new tasks.
- The agent is optimized using a slow RL algorithm (like Trust Region Policy Optimization - TRPO), but the RNN itself acts as a fast learner.

BLOCK DIAGRAM



Algorithm

1. **Define the RNN-based policy using GRU.**

```
# RNN policy with GRU
class RNNPolicy(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNPolicy, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        x = x.view(-1, 1, self.gru.input_size)
        out, hidden = self.gru(x, hidden)
        logits = self.fc(out.squeeze(0))
        return logits, hidden

    def init_hidden(self):
        return torch.zeros(1, 1, self.gru.hidden_size)
```

2. **Sample Task (MDP)**
3. **Multiple tasks were sampled for each trial**
4. **Embed Inputs**

In this step, the agent's current observation (state), action, reward, and termination flag (indicating whether the episode has ended) are combined and converted into a format suitable for input into the RNN-based policy network. This process is known as **embedding the inputs**.

```
# Embedding inputs for RNN
def embed_input(state, action, reward, done):
    action_vec = np.zeros(2)
    action_vec[action] = 1.0
    termination_flag = 1.0 if done else 0.0
    # Ensure the input is in the correct shape (1, input_size)
    input_tensor = np.concatenate([state, action_vec, [reward, termination_flag]])
    return torch.FloatTensor(input_tensor).unsqueeze(0)  # Add an extra dimension for the sequence length
```

5. **Policy Forward Pass (RNN)**
6. **Select Action (Sampling)**
7. **Take Action in the Environment**
8. **Store Trajectory**
9. **Compute GAE (Generalized Advantage Estimation)**

In this step, we calculate the **returns** for each step in the trajectory using **Generalized Advantage Estimation (GAE)**. GAE helps reduce variance in the policy gradient while keeping the bias minimal

```
# GAE computation for variance reduction
def compute_gae(rewards, values, gamma=0.99, lam=0.95):
    returns = []
    gae = 0
    next_value = 0
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * next_value - values[step]
        gae = delta + gamma * lam * gae
        returns.insert(0, gae + values[step])
        next_value = values[step]
    return torch.FloatTensor(returns)
```

10.Perform TRPO Step

```
# TRPO Policy Optimization (using a constraint on KL divergence)
def trpo_step(policy, trajectories, max_kl=0.01):
    states, actions, returns, old_log_probs = zip(*trajectories)

    # Convert to tensors
    states = torch.stack(states)
    actions = torch.LongTensor(actions)
    returns = torch.FloatTensor(returns)
    old_log_probs = torch.FloatTensor(old_log_probs)

    def compute_loss():
        logits, _ = policy(states, policy.init_hidden())
        new_log_probs = Categorical(logits=logits).log_prob(actions)
        loss = -torch.mean((returns - returns.mean()) * new_log_probs)
        return loss

    # Backprop and gradient update
    loss = compute_loss()
    policy.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(policy.parameters(), 1.0)
    optimizer.step()
```

11. Repeat for all episodes in a trial

12.Meta-Policy Update

```
# Train the meta-policy using RL²
def train_rl2(env, policy, optimizer, num_trials, num_episodes):
    for trial in range(num_trials):
        total_rewards = []
        for _ in range(num_episodes):
            state = env.reset()
            hidden = policy.init_hidden()
            trajectory = []

            done = False
            while not done:
                state_tensor = embed_input(state, 0, 0, done)
                logits, hidden = policy(state_tensor, hidden)
                action = Categorical(logits=logits).sample().item()

                next_state, reward, done, _ = env.step(action)
                log_prob = Categorical(logits=logits).log_prob(torch.tensor(action))

                trajectory.append((state_tensor, action, reward, log_prob))
                state = next_state

            rewards = [x[2] for x in trajectory]
            values = [0] * len(rewards)
            returns = compute_gae(rewards, values)

            total_rewards.append(sum(rewards))
            trpo_step(policy, trajectory)
```

Visualizing Rewards

```
# Hyperparameters
input_size = 10
hidden_size = 128
output_size = 2
num_trials = 10
num_episodes = 10
learning_rate = 0.001

# Initialize environment, policy, and optimizer
env = NetworkEnv(dataset)
policy = RNNPolicy(input_size, hidden_size, output_size)
optimizer = optim.Adam(policy.parameters(), lr=learning_rate)

# Train the RL² meta-policy
train_rl2(env, policy, optimizer, num_trials, num_episodes)
```

```
Trial 1/10, Avg Reward: 5.11
Trial 2/10, Avg Reward: 29.03
Trial 3/10, Avg Reward: 34.20
Trial 4/10, Avg Reward: 34.19
Trial 5/10, Avg Reward: 34.19
```

Conclusion

The Deep Reinforcement Learning (DRL) algorithms—Soft Actor-Critic (SAC), Model-Agnostic Meta-Learning (MAML), and RL^2 were implemented — for packet scheduling in MPTCP. Each algorithm showed strengths in managing dynamic network conditions, with SAC performing well in handling continuous actions and balancing exploration and exploitation. However, its adaptability to rapidly changing environments was limited compared to MAML and RL^2.

MAML proved to be the most effective, excelling in scenarios with high variability by quickly adapting to new network conditions with minimal data. Its rapid learning and ability to optimize scheduling policies across different network states resulted in the best overall performance in terms of reward maximization. MAML's fast adaptation made it particularly useful in highly dynamic environments, where responsiveness is critical.

RL^2 demonstrated strong convergence and was highly effective at generalizing across tasks. Its ability to retain knowledge from previous experiences and apply it to new situations allowed for quick adaptation and improved stability, especially in networks with repeated, slightly varying conditions. While MAML showed the best improvement overall, RL^2's fast convergence and learning efficiency made it a competitive alternative.

In comparison, SAC struggled with sample efficiency and slower adaptation, requiring more careful tuning to match the performance of MAML and RL^2. Overall, MAML and RL^2 offer superior adaptability and efficiency, making them better suited for real-time network scheduling in dynamic environments. Future work would focus on further refining these meta-learning approaches to handle large-scale network deployments.

References

1. Bruno Y. L. Kimura; Demetrius C. S. F. Lima and Antonio A. F. Loureiro “Packet Scheduling in Multipath TCP: Fundamentals, Lessons, and Opportunities” 2. Mohammed Asiri “Novel Multipath TCP Scheduling Design for Future IoT Applications ”
2. Vivek Adarsh; Paul Schmitt; and Elizabeth Belding “MPTCP Performance over Heterogenous Subpaths”
3. Han Zhang, Wenzhong Li, Shaohua Gao, Xiaoliang Wang, and Baoliu Ye. 2019. ReLeS: A neural adaptive multipath scheduler based on deep reinforcement learning. In Proceedings of the IEEE Conference on Computer Communications (INFOCOM’19). IEEE, 1648–1656.
4. Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How hard can it be? Designing and implementing a deployable multipath TCP. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI’12). 399–412.
5. Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In International conference on machine learning, pages 1126--1135. PMLR, 2017.
6. Duan, Yan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever and P. Abbeel. “RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning.” *ArXiv* abs/1611.02779 (2016).