Fostering Emergent Communication in Multi-Agent Systems with Ray RLlib 2.9.0: A Researcher's Guide

Part I: Foundations of Emergent Communication in MARL

This initial part of the report establishes the theoretical groundwork necessary to understand *why* emergent communication is a critical and challenging research area in multi-agent reinforcement learning (MARL). It provides the conceptual basis for the practical implementations that follow.

Section 1: The Rationale for Learned Communication

The central challenge in cooperative multi-agent systems is effective coordination. When agents must work together to achieve a common goal, their ability to share information and align their actions becomes paramount. This section introduces the core problem that emergent communication (EC) aims to solve: achieving sophisticated coordination under the constraints of uncertainty and partial information.

1.1 The Challenge of Partial Observability

Most realistic multi-agent scenarios, from autonomous vehicle fleets and robotic warehouse operations to future 6G wireless networks, can be modeled as a Partially Observable Markov Decision Process (POMDP) or, more accurately, a Decentralized POMDP (Dec-POMDP). In this framework, each agent possesses only a limited, local view of the global environment state. For instance, one autonomous vehicle can only observe nearby traffic and road conditions, not the state of the entire city's traffic network. This partial observability is the fundamental driver for communication. To form a more coherent and complete understanding of the world state and thereby coordinate their actions effectively, agents must exchange information.

In the absence of a communication channel, each agent is forced to treat its peers as unpredictable components of the environment. From the perspective of a single agent, the actions of other learning agents cause the environment's dynamics to become non-stationary. This non-stationarity violates the core Markov assumption underlying many single-agent RL algorithms and is a well-known cause of poor performance and convergence failure in MARL. Communication provides a mechanism to mitigate this non-stationarity by allowing agents to explicitly signal their intentions or share their observations, making the collective system more predictable and learnable. The environment itself, through its complexity and the nature of its rewards, creates a selective pressure that determines the value and sophistication of the communication that arises. The structure of the environment and the task directly shapes the structure of the emergent protocol. A simple, fully-observable task may not require any communication, leading to "babbling" or protocol collapse, whereas a complex task with sparse rewards and significant partial observability makes meaningful communication a prerequisite for success.

1.2 Limitations of Hard-Coded Protocols

A straightforward approach to enabling coordination is to design a communication protocol manually. In this paradigm, a human designer pre-defines a set of messages and their meanings (e.g., "message 1 means 'go to location X'," "message 2 means 'obstacle detected'"). However, this approach has significant drawbacks that limit its applicability in complex, dynamic systems.

First, manual design is brittle. A protocol designed for one specific task or environment may fail completely if the conditions change, as it lacks the ability to adapt. Second, it is often sub-optimal. It is exceptionally difficult for a human designer to know *a priori* precisely what information is most critical for agents to share to maximize their collective performance. The optimal communication strategy may be non-intuitive or highly dependent on the learned policies of the agents themselves. Finally, designing such protocols is a labor-intensive and challenging engineering task, especially as the number of agents and the complexity of the state-action space grows.

1.3 Emergent Communication as a Solution

Emergent communication offers a powerful alternative to manual design. EC is the process whereby a population of autonomous agents, through interaction with each other and their environment, develops a shared communication protocol without any pre-programmed linguistic rules. The communication protocol is learned concurrently with the agents' policies for acting in the environment, typically guided by a shared, task-related reward signal.

In this framework, the "meaning" of a message is not defined by a human but emerges from its utility. If sending a particular signal in a particular context leads to coordinated actions that result in a high reward, that signal-meaning association is reinforced for all participating agents. This process allows for the development of communication protocols that are:

- **Adaptive:** The protocol can evolve as the environment or task changes.
- **Task-Specific:** The communication is optimized for the specific collaborative problem the agents are trying to solve, often leading to highly efficient and compressed representations of information.
- **Autonomous:** It removes the bottleneck of human design, enabling the deployment of multi-agent systems in complex domains where optimal coordination strategies are unknown.

The study of EC, therefore, lies at the intersection of multi-agent systems, reinforcement learning, and linguistics, seeking to understand how functional, structured communication can arise from the simple objective of maximizing cumulative reward.

Section 2: A Survey of Seminal and Modern EC Architectures

To implement emergent communication, specific architectural and algorithmic choices must be made. Over the years, researchers have proposed several influential paradigms that form a "design space" for EC models. Understanding these foundational and modern architectures is crucial for making informed decisions when building custom communication-enabled agents in RLlib.

2.1 Foundational Paradigms

Three early works laid the groundwork for much of the subsequent research in learned multi-agent communication.

- **RIAL (Reinforced Inter-Agent Learning):** Proposed by Foerster et al. (2016), RIAL treats communication as an explicit action within the agent's action space. An agent's policy network outputs a probability distribution over both its environmental actions and a vocabulary of discrete messages. The agent then samples from both to choose what to do and what to say. The entire system is trained using independent Q-learning (or other policy gradient methods) for each agent, where the only learning signal comes from the shared environmental reward. While simple and fully decentralized at execution, RIAL often struggles with the credit assignment problem. The impact of a single message on the final team reward is often highly delayed and noisy, making it difficult for agents to discover a useful communication protocol.
- **DIAL (Differentiable Inter-Agent Learning):** As a direct successor to RIAL, DIAL introduced a key innovation to address the weak learning signal: a differentiable communication channel. Instead of passing discrete, one-hot message vectors (which have no gradient), DIAL agents pass continuous, real-valued message vectors between each other during training. These continuous messages are then discretized (e.g., through a simple rounding or a specialized unit) before being fed into the listener's policy network. The critical insight is that because the channel is continuous during the forward pass of training, gradients from the listener agent can flow backward through the communication channel and directly into the weights of the speaker agent's network. This provides a much richer, more direct learning signal about how a message influences a listener's behavior. DIAL is a canonical example of the Centralized Training, Decentralized Execution (CTDE) paradigm: the end-to-end differentiability is exploited during a centralized training phase, but at execution time, agents can operate decentrally by passing the discretized messages.
- **CommNet (Communication-based Neural Network):** Proposed by Sukhbaatar et al. (2016), CommNet takes a different approach based on message aggregation. At each step, each agent's hidden state is considered its "message."

These hidden state vectors are then averaged (or summed) to create a single communication vector. This aggregated vector is then broadcast to all agents and used as an input to their policy network for the next time step. This architecture allows for continuous communication and multiple rounds of communication before an action is taken. A typical implementation of CommNet uses a single, shared policy network for all agents and assumes a fully connected communication graph.

2.2 Modern Approaches and Enhancements

Building on these foundations, more recent work has incorporated advanced deep learning architectures to enable more sophisticated communication patterns.

- **Attention-Based Communication:** Algorithms like TarMAC (Targeted Multi-Agent Communication) and MAAC (Multi-Actor-Attention-Critic) leverage attention mechanisms, inspired by their success in natural language processing. Attention allows agents to learn *who* to listen to. Instead of simply averaging all incoming messages as in CommNet, an attention mechanism allows a listener to dynamically compute weights for each speaker's message based on its relevance to the listener's current context. TarMAC, for example, uses a signature-based soft attention mechanism where speakers broadcast a "key" and listeners attend to messages whose keys best match their own "query". This enables targeted communication in a broadcast medium.
- **Graph-Based Communication:** For problems with an inherent graph structure (e.g., traffic networks, power grids, or spatially distributed robots), Graph Neural Networks (GNNs) have emerged as a powerful tool. In this paradigm, agents are nodes in a graph, and communication links are edges. A GNN-based policy allows agents to learn communication policies by performing message passing, where information is propagated and aggregated from an agent's local neighborhood in the graph. Algorithms like MAGIC (Multi-Agent Graph-Attention Communication) combine GNNs with attention to learn effective, structured communication protocols that respect the underlying topology of the problem.

The following table provides a structured comparison of these key EC algorithms, summarizing their core properties and design choices.

Algorithm	Communication	Gradient Flow	Training	Key Mechanism

	Type		Paradigm	
RIAL	Discrete	Blocked	Decentralized / CTDE	Agents learn to select messages as discrete actions via standard RL (e.g., Q-learning).
DIAL	Continuous (during training), Discrete (during execution)	Differentiable	Centralized Training, Decentralized Execution (CTDE)	Gradients from the listener are backpropagated through a continuous channel to the speaker.
CommNet	Continuous	Differentiable	Centralized Training, Centralized Execution	Agents' hidden states are averaged and broadcast to all other agents as the communication signal.
TarMAC	Continuous	Differentiable	Centralized Training, Decentralized Execution (CTDE)	Uses a signature-based soft attention mechanism for agents to learn targeted communication.
MAGIC	Continuous	Differentiable	Centralized Training, Decentralized Execution (CTDE)	Employs a Graph Neural Network (GNN) with attention to learn communication over a defined graph topology.

Part II: The Ray RLlib 2.9.0 Toolkit for Multi-Agent RL

This part of the report transitions from the theory of emergent communication to the practical tools provided by Ray 2.9.0 and its reinforcement learning library, RLlib. A deep understanding of these APIs is a prerequisite for implementing the complex, custom models required for EC research. All examples and API descriptions adhere to RLlib's "new API stack," which has been the default since Ray 2.4.0 and is standard in version 2.9.0.

Section 3: Architecting Multi-Agent Environments with MultiAgentEnv

The foundation of any MARL experiment is the environment itself. RLlib provides a powerful and flexible base class, ray.rllib.env.multi_agent_env.MultiAgentEnv, for creating custom environments that can host multiple, interacting agents.

3.1 The MultiAgentEnv Class

The MultiAgentEnv is a subclass of gymnasium.Env but is specifically designed to handle the complexities of multi-agent interaction. Agents within the environment are identified by unique strings called AgentIDs. The environment developer has full control over which agents are active at any given time, allowing for scenarios where agents can join or leave an episode dynamically.

To create a custom multi-agent environment, one must inherit from this class and implement its core methods, primarily reset() and step().

Python

```python
import gymnasium as gym
from ray.rllib.env.multi_agent_env import MultiAgentEnv, AgentID
from typing import Dict, Tuple

class MyCustomMultiAgentEnv(MultiAgentEnv):
    def __init__(self, config=None):
        super().__init__()
        # Agent IDs can be defined here
        self._agent_ids = {"agent_0", "agent_1"}

        # Define observation and action spaces for each agent
        # These are crucial for defining heterogeneous agents
        self.observation_space = gym.spaces.Dict(
            {
                "agent_0": gym.spaces.Box(low=0, high=1, shape=(4,)),
                "agent_1": gym.spaces.Box(low=0, high=1, shape=(8,)),
            }
        )
        self.action_space = gym.spaces.Dict(
            {
                "agent_0": gym.spaces.Discrete(2),
                "agent_1": gym.spaces.Discrete(4),
            }
        )

    def reset(self, *, seed=None, options=None):
        #... implementation...
        pass

    def step(self, action_dict: Dict):
        #... implementation...
        pass
```

3.2 The Dictionary-Based API

Unlike single-agent gymnasium environments, MultiAgentEnv uses dictionaries to pass information for each agent. This is the fundamental interaction pattern.

- **reset()**: This method is called at the beginning of each episode. It must return a tuple of (observations, infos). The observations element is a dictionary mapping the AgentID of each agent that is ready to act to its corresponding initial observation. The infos dictionary can contain auxiliary diagnostic information for each agent.

- **step(action_dict)**: This method advances the environment by one timestep. It receives an action_dict, which maps AgentIDs to the actions chosen by their respective policies. It must return a tuple of (observations, rewards, terminateds, truncateds, infos).
    - observations: A dictionary mapping AgentIDs to the next observation for each agent.
    - rewards: A dictionary mapping AgentIDs to the scalar reward received at this step.
    - terminateds: A dictionary mapping AgentIDs to a boolean indicating if that agent's episode has ended. It **must** also contain a special key, __all__, which is True if the entire episode is over for all agents, and False otherwise.
    - truncateds: A dictionary similar to terminateds, indicating if an episode was ended due to a time limit rather than a terminal state. It also requires the __all__ key.
    - infos: A dictionary for per-agent diagnostic information.

The structure of these dictionaries is not merely a data format; it is the primary mechanism for controlling the flow of interaction. The set of keys present in the observations dictionary returned by reset() or step() explicitly defines which agents are expected to compute an action for the *next* step. If an agent's ID is absent from the observations dictionary, RLlib will not query its policy for an action. This allows the environment designer to implement any interaction pattern, from fully simultaneous (all agent IDs are always in the obs dict) to strictly turn-based (only one agent ID is in the obs dict at a time) or any complex, asynchronous combination thereof. This level of control is essential for many EC paradigms, such as a speaker-listener setup where the listener should only act *after* receiving a message from the speaker.

3.3 Defining Heterogeneous Agents

A key requirement for many EC setups is that agents can be heterogeneous, meaning they have different capabilities and thus different observation and action spaces. For example, a "speaker" agent might have an action space corresponding to a vocabulary of messages, while a "listener" agent has an action space for navigating the environment.

MultiAgentEnv supports this natively. As shown in the code example above, the self.observation_space and self.action_space attributes can be set to gym.spaces.Dict objects. These dictionaries map each AgentID to its individual gym.Space. RLlib will automatically use this information to construct appropriately-sized policies for each agent type.

3.4 Integrating with PettingZoo

For many standard MARL benchmarks, implementing an environment from scratch is unnecessary. The PettingZoo library provides a wide array of pre-built multi-agent environments with a standardized API. RLlib provides a seamless integration through its ray.rllib.env.wrappers.pettingzoo_env.PettingZooEnv wrapper. This allows researchers to easily test their custom communication models on established EC tasks like SimpleSpeakerListener or SimpleReference with minimal boilerplate code.

Python

```python
from ray.rllib.env.wrappers.pettingzoo_env import PettingZooEnv
from pettingzoo.sisl import simple_speaker_listener_v4
from ray.tune.registry import register_env

# Define a creator function
def env_creator(config):
    return PettingZooEnv(simple_speaker_listener_v4.parallel_env())

# Register the environment with a custom name
register_env("speaker_listener", env_creator)

# This name can now be used in the AlgorithmConfig
# config.environment("speaker_listener")
```

Section 4: Configuring and Launching MARL Experiments

Once an environment is defined, the next step is to configure the training algorithm. RLlib's new API stack centers around the AlgorithmConfig object, which provides a fluent, chainable interface for setting up all aspects of an experiment.

4.1 The AlgorithmConfig Object

Instead of manipulating a large, nested dictionary of configuration keys, the modern approach is to instantiate a config object for a specific algorithm (e.g., PPOConfig, DQNConfig) and modify it using its methods. This provides better static analysis, auto-completion, and validation.

Python

```python
from ray.rllib.algorithms.ppo import PPOConfig

# Start with a default config for the PPO algorithm
config = PPOConfig()

# Chain methods to modify the configuration
config = config.environment("my_multi_agent_env")
config = config.training(lr=0.0005, gamma=0.99)
config = config.resources(num_gpus=1)

# Finally, build the Algorithm instance from the config
algo = config.build()
```

4.2 The .multi_agent() Method

The .multi_agent() method is the control center for configuring MARL-specific settings. It takes several key arguments that define how agents are mapped to policies and which policies are trained.

- **policies**: This argument defines the set of policies (i.e., neural networks) that will be created for the experiment. It can be provided as a set of policy ID strings (e.g., {"policy_0", "policy_1"}) for simple cases where all policies are identical. For heterogeneous policies (essential for EC), it should be a dictionary mapping policy IDs to PolicySpec objects, where each spec can define the policy class, observation/action spaces, and custom configurations.
- **policy_mapping_fn**: This is arguably the most critical component of the multi-agent setup. It is a Python callable (e.g., a lambda function) that takes an agent_id from the environment as input and returns the policy_id that should be used to compute actions for that agent. This function is the explicit link between the agents defined in the MultiAgentEnv and the policies defined in the policies argument. It allows for flexible schemes like policy sharing (multiple agents mapping to the same policy) or distinct policies for each agent.
- **policies_to_train**: This optional argument takes a list of policy IDs. Only the policies whose IDs are in this list will have their weights updated during training. Any policy not in this list will be frozen. This is extremely useful for training an agent against a fixed, heuristic opponent or for curriculum learning scenarios where different agents are trained at different times.

4.3 Scaling with .env_runners() and .learners()

Ray's core value proposition is scalable distributed computing, and RLlib exposes this power through the AlgorithmConfig.

- **.env_runners(num_env_runners=N)**: This method configures the "rollout workers" or EnvRunner actors. Setting num_env_runners to a value greater than 0 will create N parallel Ray actors, each with its own copy of the environment, to collect experience (i.e., run simulation steps) in parallel. This is the primary way to scale up the data collection bottleneck in RL.
- **.learners(num_learners=M, num_gpus_per_learner=G)**: This method, part of the new API stack, configures the LearnerGroup. Setting num_learners to M > 1 creates M parallel actors responsible for performing the neural network updates (gradient descent). If GPUs are available, setting num_gpus_per_learner=1 will place each Learner on its own GPU, enabling distributed data-parallel training.

This is the primary way to scale up the model training bottleneck.

The following table summarizes the key configuration methods essential for setting up a MARL experiment, with a specific focus on their role in emergent communication setups.

Method	Key Parameters	Purpose in MARL/EC Context	Example Snippet
.environment()	env, env_config	Specifies the custom MultiAgentEnv to be used and passes any necessary configuration to its constructor.	.environment(env=MyCommEnv, env_config={"vocab_size": 10})
.multi_agent()	policies, policy_mapping_fn, policies_to_train	Defines the policies (e.g., "speaker", "listener"), maps environment agents to them, and specifies which ones are trainable. This is the core of the MARL setup.	.multi_agent(policies={"speaker", "listener"}, policy_mapping_fn=lambda agent_id,...: agent_id)
.rl_module()	rl_module_spec	Specifies the custom RLModule class to be used. This is necessary for implementing the custom neural network architectures required for communication.	.rl_module(rl_module_spec=RLModuleSpec(module_class=MyCommModule))
.env_runners()	num_env_runners	Scales experience collection by creating parallel environment actors. Crucial for reducing wall-clock time in complex environments.	.env_runners(num_env_runners=4)
.learners()	num_learners, num_gpus_per_learn	Scales model training by creating parallel	.learners(num_learners=2,

	er	learner actors, often on separate GPUs, for distributed training.	num_gpus_per_learner=1)

Section 5: Custom Model Architectures with RLModule

The "brain" of an RL agent is its policy model. To implement emergent communication, where agents must learn to generate and interpret novel signals, standard feed-forward or convolutional networks are insufficient. RLlib's RLModule API provides the necessary flexibility to build the bespoke neural network architectures required for this task.

5.1 The New API Stack: RLModule vs. ModelV2

It is critical for developers using Ray 2.9.0 to understand the shift from the legacy ModelV2 API to the new RLModule API. The RLModule is the standard for defining models in the new API stack and offers a cleaner separation of concerns, better modularity, and is designed to be usable even outside of RLlib. Many older RLlib examples found online may use ModelV2; these are not directly compatible with the modern AlgorithmConfig and training workflows without using a migration wrapper. For new projects, and for the purposes of this report, all custom models will be implemented as RLModules.

5.2 Anatomy of an RLModule

A custom RLModule is a Python class that inherits from ray.rllib.core.rl_module.RLModule and typically wraps a torch.nn.Module. It has a well-defined structure:

- **setup()**: This method is the equivalent of a constructor. It is where the layers of the neural network (e.g., torch.nn.Linear, torch.nn.LSTM, torch.nn.Embedding) are defined and initialized.

- **Forward Methods**: The RLModule API separates the model's forward pass into three distinct methods, each for a different purpose:
    - _forward_exploration(batch, **kwargs): This method is called by EnvRunner actors during experience collection. It should implement the logic for generating actions used to explore the environment (e.g., sampling from a categorical distribution).
    - _forward_inference(batch, **kwargs): This method is called during evaluation or deployment. It should implement the logic for generating the optimal, deterministic action (e.g., taking the argmax of the policy logits).
    - _forward_train(batch, **kwargs): This method is called by Learner actors during a training update. Its purpose is to take a batch of training data and compute the values necessary to calculate the loss function (e.g., action logits and value function predictions). The output of this method is passed directly to the algorithm's loss function.

This separation allows for different network behaviors during training and inference (e.g., using dropout only during training) and is a key design principle of the new API stack.

5.3 Implementing a Communication-Aware RLModule

The RLModule is the ideal place to implement the custom logic for communication. The environment defines the *possibility* of communication by structuring the observation and action spaces, but the RLModule defines the *mechanism*. For an EC algorithm, the implementation pattern involves creating a custom RLModule that contains the specific architectural components needed for communication, such as message-generation heads or message-aggregation layers.

For a speaker-listener task, a single RLModule could contain two sub-networks: one for the speaker and one for the listener. The forward methods would then need to handle the flow of information between them. For instance, _forward_exploration would first pass the speaker's observation through its network to generate a message, then pass that message along with the listener's local observation to the listener's network to generate an action. This entire sequence happens within a single RLModule, which is then mapped to both the speaker and listener agents.

5.4 Configuring a Custom RLModule

To instruct RLlib to use a custom model, the AlgorithmConfig's .rl_module() method is used. This method accepts an RLModuleSpec object, which specifies the class of the custom module and any configuration parameters it might need.

Python

```python
from ray.rllib.core.rl_module.rl_module import RLModuleSpec
from my_comm_module import MyCommunicationRLModule # Our custom class

# In the AlgorithmConfig setup:
config.rl_module(
    rl_module_spec=RLModuleSpec(
        module_class=MyCommunicationRLModule,
        model_config_dict={"vocab_size": 10, "hidden_dim": 64},
    )
)
```

This configuration tells RLlib to instantiate MyCommunicationRLModule instead of a default MLP or CNN, passing the model_config_dict to its setup() method. This clean, explicit configuration is a hallmark of the new API stack.

Part III: Practical Implementation of Emergent Communication in RLlib

This part synthesizes the theoretical concepts and API knowledge from the previous sections into complete, runnable code examples. The goal is to provide a practical, step-by-step guide for implementing and analyzing an emergent communication system using Ray RLlib 2.9.0.

Section 6: Foundational Example: A Speaker-Listener Task

We begin with a classic emergent communication problem: the speaker-listener task. In this cooperative scenario, a "speaker" agent observes a goal state but cannot act, while a "listener" agent can act but cannot see the goal. To succeed, the speaker must learn to generate a meaningful message that guides the listener to the correct location. This task isolates the need for communication, making it an ideal starting point. We will use the SimpleSpeakerListener environment from the PettingZoo library as a reference for our custom implementation.

6.1 Environment Setup: SpeakerListenerEnv

First, we define a custom MultiAgentEnv. The speaker (speaker_0) observes a one-hot vector indicating the goal, and its action is to choose a message from a discrete vocabulary. The listener (listener_0) observes its own position and the message from the speaker, and its action is to move.

Python

```python
import gymnasium as gym
import numpy as np
from ray.rllib.env.multi_agent_env import MultiAgentEnv, AgentID

class SpeakerListenerEnv(MultiAgentEnv):
    """A custom speaker-listener environment.

    - Speaker observes a goal ID (0, 1, or 2).
    - Speaker communicates a message (0, 1, or 2).
    - Listener observes its own position (always 0) and the speaker's message.
    - Listener must move to the goal location to receive a reward.
    """
    def __init__(self, config=None):
        super().__init__()
        config = config or {}
        self.num_goals = 3
        self.vocab_size = 3

        self._agent_ids = {"speaker_0", "listener_0"}

        # Speaker: observes goal, acts by sending a message
        self.observation_space_speaker = gym.spaces.Box(0, 1, shape=(self.num_goals,),
dtype=np.float32)
        self.action_space_speaker = gym.spaces.Discrete(self.vocab_size)

        # Listener: observes message, acts by moving
        self.observation_space_listener = gym.spaces.Dict({
            "pos": gym.spaces.Box(-1, 1, shape=(1,), dtype=np.float32),
            "comm": gym.spaces.Box(0, 1, shape=(self.vocab_size,), dtype=np.float32)
        })
        self.action_space_listener = gym.spaces.Discrete(self.num_goals)

        # RLlib requires these to be set
        self.observation_space = gym.spaces.Dict({
            "speaker_0": self.observation_space_speaker,
            "listener_0": self.observation_space_listener,
        })
        self.action_space = gym.spaces.Dict({
            "speaker_0": self.action_space_speaker,
            "listener_0": self.action_space_listener,
        })

        self.goal = None
        self.message = None

    def reset(self, *, seed=None, options=None):
        super().reset(seed=seed)
        self.goal = self.np_random.integers(0, self.num_goals)
        self.message = None # Reset message

        # Speaker acts first
        obs = {
```

```python
            "speaker_0": np.eye(self.num_goals)[self.goal].astype(np.float32)
        }
        infos = {"speaker_0": {}, "listener_0": {}}
        return obs, infos

    def step(self, action_dict):
        # If speaker acted, we now get listener's observation
        if "speaker_0" in action_dict:
            self.message = action_dict["speaker_0"]

            obs = {
                "listener_0": {
                    "pos": np.array([0.0], dtype=np.float32),
                    "comm": np.eye(self.vocab_size)[self.message].astype(np.float32)
                }
            }
            # No rewards yet, episode is not over
            rewards = {"speaker_0": 0.0, "listener_0": 0.0}
            terminateds = {"__all__": False}
            truncateds = {"__all__": False}

        # If listener acted, the episode is over
        elif "listener_0" in action_dict:
            listener_action = action_dict["listener_0"]

            # Both agents get reward if listener chose the correct goal
            reward = 1.0 if listener_action == self.goal else -1.0
            rewards = {"speaker_0": reward, "listener_0": reward}

            # Episode ends
            obs = {} # No agent acts next
            terminateds = {"__all__": True}
            truncateds = {"__all__": True}

        else:
            # Should not happen
            raise ValueError("Invalid action_dict")

        infos = {"speaker_0": {}, "listener_0": {}}
        return obs, rewards, terminateds, truncateds, infos
```

This environment implements a strict turn-based interaction. reset() only provides an observation for the speaker. The step() function checks which agent acted and returns an observation for the next agent in the sequence, demonstrating the power of the dictionary-based API for controlling interaction flow.

6.2 Custom RLModule for Communication

Next, we need a custom RLModule that contains separate networks for the speaker and listener. We will use a MultiRLModule to encapsulate this logic cleanly.

Python

```
import torch
from torch import nn
from ray.rllib.core.rl_module.torch import TorchRLModule
from ray.rllib.core.rl_module.rl_module import RLModule
from ray.rllib.core.rl_module.multi_rl_module import MultiRLModule
from ray.rllib.policy.sample_batch import SampleBatch
from ray.rllib.utils.annotations import override
from ray.rllib.utils.framework import try_import_torch
from ray.rllib.models.torch.torch_distributions import TorchCategorical

class SpeakerListenerModule(TorchRLModule):
    """A module for either a speaker or a listener."""
    def __init__(self, config):
        super().__init__(config)

        # Unpack custom model config
        model_config = config.model_config
        self.is_speaker = model_config["is_speaker"]
        obs_dim = model_config["obs_dim"]
        action_dim = model_config["action_dim"]
        hidden_dim = 64

        # Shared encoder
        self.encoder = nn.Sequential(
            nn.Linear(obs_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        # Policy and value heads
        self.pi_head = nn.Linear(hidden_dim, action_dim)
        self.vf_head = nn.Linear(hidden_dim, 1)

    @override(RLModule)
    def _forward_train(self, batch: SampleBatch, **kwargs):
        # For PPO, we need logits and value function output
        obs = batch

        # For listener, obs is a dict, so we flatten it
        if not self.is_speaker:
            obs = torch.cat([obs["pos"], obs["comm"]], dim=-1)

        features = self.encoder(obs)
        logits = self.pi_head(features)
        vf_out = self.vf_head(features)

        return {
            SampleBatch.ACTION_DIST_INPUTS: logits,
            SampleBatch.VF_PREDS: vf_out.squeeze(-1),
        }

    # Exploration and inference forwards are similar for this simple case
    @override(RLModule)
    def _forward_exploration(self, batch, **kwargs):
        return self._common_forward(batch)

    @override(RLModule)
    def _forward_inference(self, batch, **kwargs):
        return self._common_forward(batch)
```

```python
    def _common_forward(self, batch):
        obs = batch
        if not self.is_speaker:
            obs = torch.cat([obs["pos"], obs["comm"]], dim=-1)

        features = self.encoder(obs)
        logits = self.pi_head(features)

        # Return dict with logits for action distribution
        return {SampleBatch.ACTION_DIST_INPUTS: logits}
```

This RLModule can be configured to act as either a speaker or a listener. The listener part handles the dictionary observation space by concatenating the features.

6.3 The Full AlgorithmConfig

Finally, we tie everything together in a configuration script. We will use the PPO algorithm, which works well with discrete action spaces and is a standard choice for MARL.

Python

```python
from ray import tune
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.core.rl_module.rl_module import RLModuleSpec
from ray.tune.registry import register_env

# Register the custom environment
register_env("speaker_listener_env", lambda config: SpeakerListenerEnv(config))

# Get the env's spaces for configuration
env = SpeakerListenerEnv()
speaker_obs_space = env.observation_space["speaker_0"]
```

```
speaker_action_space = env.action_space["speaker_0"]
listener_obs_space = env.observation_space["listener_0"]
listener_action_space = env.action_space["listener_0"]

# PPO Algorithm Configuration
config = (
   PPOConfig()
  .environment(env="speaker_listener_env")
  .framework("torch")
  .training(
      # Standard PPO hyperparameters
      train_batch_size=4000,
      sgd_minibatch_size=128,
      num_sgd_iter=10,
      lr=5e-5,
      vf_loss_coeff=0.5,
      entropy_coeff=0.01,
   )
  .multi_agent(
      # Define the two policies
      policies={
          "speaker_policy": (
              None, speaker_obs_space, speaker_action_space, {}
          ),
          "listener_policy": (
              None, listener_obs_space, listener_action_space, {}
          ),
      },
      # Map agents to policies
      policy_mapping_fn=lambda agent_id, episode, **kwargs:
          "speaker_policy" if agent_id == "speaker_0" else "listener_policy",
   )
  .rl_module(
      # Use a MultiRLModule to hold our custom modules
      rl_module_spec=MultiRLModuleSpec(
          module_specs={
              "speaker_policy": RLModuleSpec(
                  module_class=SpeakerListenerModule,
                  model_config_dict={
                      "is_speaker": True,
                      "obs_dim": speaker_obs_space.shape,
                      "action_dim": speaker_action_space.n,
                  },
              ),
              "listener_policy": RLModuleSpec(
                  module_class=SpeakerListenerModule,
                  model_config_dict={
                      "is_speaker": False,
                      "obs_dim": listener_obs_space["pos"].shape +
listener_obs_space["comm"].shape,
                      "action_dim": listener_action_space.n,
                  },
              ),
          }
      )
   )
  .resources(num_gpus=0)
  .env_runners(num_env_runners=1, num_envs_per_env_runner=1)
)

# To run the training:
# tune.Tuner("PPO", param_space=config.to_dict()).fit()
```

6.4 Code Walkthrough and Expected Results

This configuration script performs several key actions:

1. It registers our custom SpeakerListenerEnv.
2. It uses PPOConfig to set up the trainer.
3. The .multi_agent() method defines two policies, speaker_policy and listener_policy, and the policy_mapping_fn correctly maps speaker_0 to the former and listener_0 to the latter.
4. The .rl_module() method is configured with a MultiRLModuleSpec. This tells RLlib to create two instances of our SpeakerListenerModule, one for each policy, and passes the appropriate configuration (is_speaker, dimensions) to each.

When this experiment is run, we expect to see the episode_reward_mean metric increase over training iterations. Initially, the listener will act randomly, resulting in a mean reward close to -1. As the speaker learns to send messages that are correlated with the goal, and the listener learns to interpret these messages correctly, the mean reward should climb towards +1, indicating that a successful, albeit simple, communication protocol has emerged.

Section 7: Advanced Technique: Implementing DIAL with a Differentiable Channel

The previous example used discrete messages and standard reinforcement, similar to RIAL. To implement a more advanced algorithm like DIAL, we need to establish a differentiable communication channel to allow for richer gradient flow from the listener to the speaker. This can be achieved in RLlib not by selecting a "DIAL" algorithm, but by designing a custom RLModule and environment that implement the DIAL pattern. The flexibility of RLlib's components, particularly the ability to define arbitrary PyTorch computational graphs within an RLModule, makes this possible.

7.1 Modifying for a Continuous Communication Channel

First, we adapt the environment and model to handle continuous messages.

- **Environment Changes**: The speaker's action space and the listener's communication observation space are changed from Discrete to Box.

Python

```python
# In SpeakerListenerEnv.__init__
self.message_dim = 16 # A 16-dimensional continuous message vector
self.action_space_speaker = gym.spaces.Box(low=-1, high=1,
shape=(self.message_dim,), dtype=np.float32)
self.observation_space_listener["comm"] = gym.spaces.Box(low=-1, high=1,
shape=(self.message_dim,), dtype=np.float32)
```

```
# In SpeakerListenerEnv.step
# The speaker's action is now a continuous vector, no one-hot encoding needed
self.message = action_dict["speaker_0"]
obs["listener_0"]["comm"] = self.message
```

- **Algorithm Change**: Since the speaker now has a continuous action space, we should use an algorithm that supports it well, like PPO or SAC. We will stick with PPO for consistency. The RLModule will now need to output parameters for a continuous distribution (e.g., mean and standard deviation for a Gaussian).

7.2 Implementing a DIAL-like RLModule

The core of the DIAL implementation lies in a single, unified RLModule that computes the loss for both agents in a single _forward_train call, creating an end-to-end differentiable graph.

```python
from ray.rllib.models.torch.torch_distributions import TorchDiagGaussian

class DIALModule(TorchRLModule):
    def __init__(self, config):
        super().__init__(config)
        # Define speaker and listener networks (encoders, pi/vf heads)
        # Speaker's pi_head outputs 2 * message_dim for mean and log_std
        # Listener's pi_head outputs action_dim (discrete)

        # Example for speaker's network
        self.speaker_encoder =...
        self.speaker_pi_head = nn.Linear(hidden_dim, 2 * self.message_dim)
        self.speaker_vf_head = nn.Linear(hidden_dim, 1)

        # Example for listener's network
        self.listener_encoder =...
        self.listener_pi_head = nn.Linear(hidden_dim, self.num_goals)
        self.listener_vf_head = nn.Linear(hidden_dim, 1)

    @override(RLModule)
    def _forward_train(self, batch, **kwargs):
        # This forward pass is for the whole multi-agent batch

        # --- Speaker Pass ---
        speaker_batch = batch.policy_batches["speaker_policy"]
        speaker_obs = speaker_batch
        speaker_features = self.speaker_encoder(speaker_obs)

        # Speaker's value prediction
        speaker_vf_pred = self.speaker_vf_head(speaker_features).squeeze(-1)

        # Speaker's action (message) distribution
        speaker_pi_out = self.speaker_pi_head(speaker_features)

        # --- Listener Pass ---
        listener_batch = batch.policy_batches["listener_policy"]
        listener_obs_pos = listener_batch["pos"]

        # CRITICAL STEP: Get the message from the speaker's action
        # The `actions` in the speaker_batch are the messages.
        # We need to ensure gradients can flow, so we don't detach.
        message_from_speaker = speaker_batch

        # Add noise for regularization, as in DIAL
        noise = torch.randn_like(message_from_speaker) * 0.1
        noisy_message = message_from_speaker + noise

        # Listener's observation is its position + the (noisy) message
        listener_input = torch.cat([listener_obs_pos, noisy_message], dim=-1)

        listener_features = self.listener_encoder(listener_input)

        # Listener's value prediction and action logits
        listener_vf_pred = self.listener_vf_head(listener_features).squeeze(-1)
        listener_logits = self.listener_pi_head(listener_features)

        # Return a dictionary of dictionaries for the MultiAgentLoss
        return {
            "speaker_policy": {
                SampleBatch.ACTION_DIST_INPUTS: speaker_pi_out,
                SampleBatch.VF_PREDS: speaker_vf_pred,
            },
            "listener_policy": {
                SampleBatch.ACTION_DIST_INPUTS: listener_logits,
                SampleBatch.VF_PREDS: listener_vf_pred,
            },
        }
```

In this setup, message_from_speaker is a tensor that is part of the computational graph. When it is used to compute the listener's loss, and loss.backward() is called by the RLlib Learner, the gradients will flow back through listener_input into message_from_speaker, and subsequently into the weights of the speaker's network. This implicitly implements the core mechanism of DIAL without needing a special algorithm class.

7.3 Auxiliary Losses for Better Communication

The _forward_train method is also the perfect place to add auxiliary losses to "bias" the communication, as explored in the literature. For example, to encourage "positive signaling" (making messages informative about the speaker's observation), we can add a loss term that tries to reconstruct the speaker's observation from its message.

Python

```python
# In DIALModule.__init__
self.reconstruction_decoder = nn.Linear(self.message_dim, self.num_goals)
self.reconstruction_loss_fn = nn.CrossEntropyLoss()
```

```
# In DIALModule._forward_train
#... after speaker pass...

# --- Auxiliary Loss Calculation ---
reconstructed_goal_logits = self.reconstruction_decoder(message_from_speaker)
true_goal_labels = torch.argmax(speaker_obs, dim=-1)
reconstruction_loss = self.reconstruction_loss_fn(reconstructed_goal_logits,
true_goal_labels)

# PPO's loss is computed by RLlib from the returned dict.
# We need to add our custom loss to the total loss.
# This can be done by overriding the algorithm's loss function.

# In the PPOConfig:
def custom_loss_fn(policy, model, dist_class, train_batch):
    # Standard PPO loss
    ppo_loss = ppo_torch_loss(...) # RLlib's standard loss function

    # Get the custom loss from the model's forward pass
    # (Requires modifying _forward_train to return it)
    reconstruction_loss = model.get_reconstruction_loss() # Hypothetical getter

    return ppo_loss + 0.1 * reconstruction_loss

config.training(loss_fn=custom_loss_fn)
```

While the exact mechanism for adding the loss requires more detailed subclassing of the PPOTorchLearner, the principle remains: the custom RLModule computes all necessary tensors, and the Learner combines them to form the final loss. This modularity allows researchers to easily experiment with different communication biases.

Section 8: Analyzing the Emergent Protocol

Achieving a high task reward is the first step. The second, more scientific step is to understand the communication protocol that the agents have learned. An effective protocol might be discovered, but agents could also find loopholes or "cheats" that solve the task without meaningful communication. Therefore, post-hoc analysis is crucial.

8.1 The Problem of Opaqueness

The communication channel in EC is often a "black box". The messages are vectors of numbers or discrete symbols whose meanings are not immediately apparent to a human observer. The primary goal of analysis is to "decode" this emergent language and verify that it is grounded in the semantics of the task.

8.2 Extracting Communication Data

The first step in analysis is to collect the data. This can be done by adding a custom callback to the AlgorithmConfig that logs the relevant information during evaluation rollouts.

Python

```python
from ray.rllib.algorithms.callbacks import DefaultCallbacks
import json

class CommunicationCallback(DefaultCallbacks):
    def on_episode_step(self, *, worker, base_env, episode, **kwargs):
        # This callback is for the old API stack, but the principle is the same.
        # In the new stack, one would use a custom EnvRunner.
        # Let's assume we can access the last observation and action.
        # For our env, the speaker's action IS the message.
        if "speaker_0" in episode.last_action_for():
            obs = episode.last_observation_for("speaker_0")
            msg = episode.last_action_for("speaker_0")
            goal = np.argmax(obs)

            if "comm_data" not in episode.user_data:
```

```
            episode.user_data["comm_data"] =
        episode.user_data["comm_data"].append((goal, msg.tolist()))

    def on_episode_end(self, *, worker, base_env, policies, episode, **kwargs):
        if "comm_data" in episode.user_data:
            with open(f"comm_data_{episode.episode_id}.json", "w") as f:
                json.dump(episode.user_data["comm_data"], f)

# In AlgorithmConfig:
# config.callbacks(CommunicationCallback)
```

8.3 Quantitative Analysis

Simple quantitative metrics can provide a first-pass assessment of the protocol. For discrete messages, one can calculate the entropy of the message distribution.
H(M)=−Σm∈Mp(m)log2p(m)
A high entropy suggests the agent is using its full vocabulary, while an entropy near zero indicates "protocol collapse," where the agent sends the same message regardless of its observation.

8.4 Visualization with t-SNE

A more powerful qualitative analysis technique is to visualize the semantic structure of the message space using dimensionality reduction. t-Distributed Stochastic Neighbor Embedding (t-SNE) is an excellent tool for this, as it is effective at revealing underlying cluster structures in high-dimensional data. The goal is to see if messages corresponding to similar world states are "closer" in the learned embedding space.

The workflow is as follows:

1. **Collect Data:** Run the trained policy in evaluation mode and use a callback or custom loop to save a large number of (observation, message) pairs. The observation represents the "meaning" or "concept" the speaker is trying to convey.
2. **Prepare Data:** Extract the ground-truth concepts (e.g., the goal ID from the speaker's observation) and the corresponding message vectors.

3. **Apply t-SNE:** Use a library like scikit-learn to fit t-SNE to the collection of message vectors, reducing their dimensionality to 2.
4. **Plot:** Create a 2D scatter plot of the resulting embeddings, coloring each point according to its ground-truth concept.

Python

```python
import json
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# 1. Load data collected from the callback
with open("all_comm_data.json", "r") as f:
    data = json.load(f)

goals = np.array([item for item in data])
messages = np.array([item for item in data])

# 2. Apply t-SNE
tsne = TSNE(n_components=2, perplexity=30, learning_rate=200, random_state=42)
message_embeddings_2d = tsne.fit_transform(messages)

# 3. Plot the results
plt.figure(figsize=(10, 8))
scatter = plt.scatter(
    message_embeddings_2d[:, 0],
    message_embeddings_2d[:, 1],
    c=goals,
    cmap='viridis'
)
plt.legend(handles=scatter.legend_elements(), labels=['Goal 0', 'Goal 1', 'Goal 2'])
plt.title("t-SNE Visualization of Emergent Message Embeddings")
plt.xlabel("t-SNE Dimension 1")
plt.ylabel("t-SNE Dimension 2")
plt.show()
```

If a meaningful protocol has emerged, this plot will show distinct clusters of points. For example, all the points corresponding to goal=0 will be grouped together in one region of the plot, while points for goal=1 will form a separate cluster. This provides strong visual evidence that the agent has learned to map distinct concepts to distinct, separable representations in its communication protocol.

Part IV: Advanced Topics and Future Directions

Having explored the theoretical foundations and practical implementations of emergent communication in Ray RLlib, this final part addresses broader considerations such as scalability, the persistent challenges in the field, and promising directions for future research.

Section 9: Scalability, Challenges, and Open Problems

While the examples provided demonstrate the core mechanics on a small scale, real-world applications require scaling to potentially thousands of agents and much more complex environments. This section discusses how Ray facilitates this scaling and outlines the key research challenges that remain.

9.1 Scaling Experiments with Ray

A primary advantage of using the Ray ecosystem is its native support for distributed computing, which is essential for tackling the high computational demands of MARL research. The RLlib APIs discussed previously provide direct access to this scalability:

- **Scaling Data Collection:** As shown, config.env_runners(num_env_runners=N) allows for the parallelization of environment simulations across N CPU-based actors. For complex environments where simulation is the bottleneck, this can lead to a near-linear speedup in data collection throughput.
- **Scaling Model Training:** Similarly, config.learners(num_learners=M,

num_gpus_per_learner=G) enables distributed data-parallel training across M learners, each potentially utilizing its own GPU. This is critical for training large, complex communication models or when the training batch size becomes too large for a single device.

Ray's architecture handles the underlying task scheduling, data movement, and fault tolerance, allowing researchers to scale their EC experiments from a single laptop to a multi-node cluster with minimal code changes.

9.2 Key Challenges in EC Research

Despite significant progress, several fundamental challenges remain at the forefront of emergent communication research.

- **Scalability and Non-stationarity:** While Ray provides the tools to scale computation, the algorithmic challenges of scaling to a large number of agents persist. As the number of learning agents grows, the environment becomes increasingly non-stationary from any single agent's perspective, which can destabilize learning. Developing communication protocols that are robust in large, dynamic populations is an active area of research, with some work exploring mean-field approaches to model the aggregate behavior of large groups.
- **Convergence Speed and Sample Efficiency:** EC algorithms are notoriously sample-inefficient. Discovering a communication protocol from scratch through trial-and-error reinforcement learning can require millions or even billions of environment steps. This makes research slow and computationally expensive. Techniques to improve convergence speed, such as curriculum learning (starting with simpler tasks), transfer learning (reusing a protocol on a new task), or incorporating prior knowledge, are crucial.
- **Generalization and Compositionality:** A major open question is whether a learned protocol can generalize. Can a protocol learned for one task be effectively applied to a new, unseen task? Can agents trained with one population successfully communicate with new, ad-hoc teammates?. A related, and even more profound, question is that of compositionality: can agents learn to combine basic learned "words" or symbols to express novel, more complex ideas, a hallmark of human language?. Current methods often learn "holistic" protocols where each message corresponds to a specific state, rather than a compositional grammar.

9.3 The Frontier: Human-AI Communication

The ultimate goal for many is to create AI agents that can not only communicate with each other but also with humans. The "black box" nature of most emergent protocols makes them unintelligible to human partners. This has spurred a new and exciting research direction focused on making emergent communication more interpretable.

This involves moving beyond post-hoc analysis and designing systems that are encouraged to learn human-understandable languages from the outset. Some approaches aim to make the *subject* of communication transparent, even if the message itself is not, by having agents learn to selectively gate what information from their observation they share. Other, more recent work explores grounding the emergent language in natural language by using Large Language Models (LLMs) to generate synthetic training data or to provide an auxiliary reward signal that encourages alignment between the agent protocol and human language embeddings.

Conclusion

This report has provided a comprehensive guide to implementing multi-agent reinforcement learning experiments focused on emergent communication using Ray 2.9.0 and its RLlib library. It began by establishing the theoretical necessity of learned communication in partially observable, cooperative settings and surveyed the landscape of seminal algorithms like RIAL, DIAL, and CommNet.

The core of the report detailed the practical toolkit provided by RLlib's new API stack. We have demonstrated how to construct complex multi-agent environments using the dictionary-based MultiAgentEnv API, which grants fine-grained control over agent interactions. We have explored the fluent AlgorithmConfig interface, focusing on the .multi_agent() method for defining policies and mapping agents, and the .learners() and .env_runners() methods for scaling experiments. A critical takeaway is the central role of the custom RLModule, which serves as the locus for all bespoke communication logic, from message generation and processing to the implementation of auxiliary losses for biasing communication.

Through a series of detailed, end-to-end code examples, this report illustrated how these components are composed to build functional EC systems. We implemented a foundational speaker-listener task with discrete messaging and then advanced to a more sophisticated DIAL-like pattern with a continuous, differentiable communication channel. This demonstrated that RLlib's flexibility allows researchers to implicitly implement complex, state-of-the-art algorithms by correctly structuring the computational graph within a custom RLModule. Finally, we provided practical techniques for analyzing the resulting protocols, using t-SNE to visualize the semantic structure of the emergent message space.

The field of emergent communication is vibrant and filled with open challenges, from algorithmic scalability and sample efficiency to the grand challenge of fostering true compositional language and human-interpretable protocols. By providing a powerful, scalable, and flexible framework, Ray and RLlib empower researchers to effectively tackle these challenges, pushing the boundaries of what is possible in multi-agent coordination and artificial intelligence.