Mastering AI Agents

Agentic Swarms & Google's A2A Protocol

A 3-Day Intensive Course

11 Modules | Hands-on Labs | Capstone Project

Course Overview

Day 1: Foundations

- AI Agent Architecture
- Swarm Intelligence
- Multi-Agent Systems

Day 2: Protocols & Tools

- Google A2A Protocol
- Agent Development
- Frameworks (Mesa, Ray)

Day 3: Applications

- Swarm Optimization
- Real-World Applications
- Ethics & Capstone

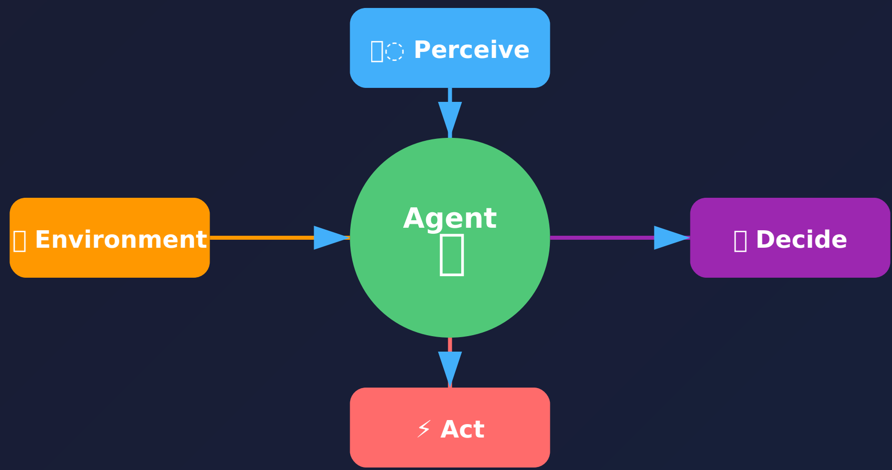
Prerequisites:

- Python proficiency
- Basic ML knowledge
- OOP experience

Module 1

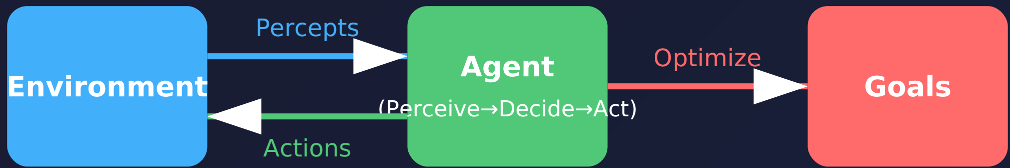
Foundations of AI Agents

Understanding autonomous intelligent systems



What is an AI Agent?

Definition: An AI agent is an autonomous entity that perceives its environment through sensors, processes information, and takes actions to achieve specific goals.



Key Properties: Autonomy, Reactivity, Proactiveness, Social Ability

Agent Types

Reactive Agents

Simple stimulus-response

- No internal state
- Fast response
- Limited reasoning

Example: Thermostat

Deliberative Agents

Planning and reasoning

- World model
- Goal-directed
- Slower but smarter

Example: Chess AI

Hybrid Agents

Best of both worlds

- Reactive layer
- Deliberative layer
- Meta-control

Example: Autonomous car

BDI Architecture

Beliefs, Desires, Intentions - A cognitive agent model

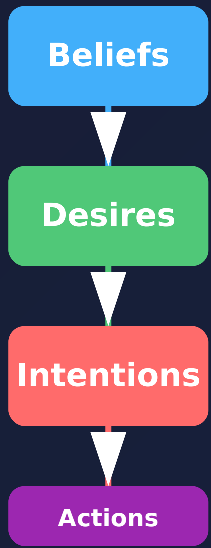
Beliefs: Agent's knowledge about the world

Desires: Goals the agent wants to achieve

Intentions: Committed plans of action

```
class BDIAgent:
    def __init__(self):
        self.beliefs = {}      # World state
        self.desires = []      # Goals
        self.intentions = []   # Active plans

    def deliberate(self):
        # Update beliefs from percepts
        # Generate options (desires)
        # Filter to intentions
        # Execute current intention
        pass
```



Environment Types (PEAS)

Performance, **E**nvironment, **A**ctuators, **S**ensors

Property	Type A	Type B	Example
Observability	Fully Observable	Partially Observable	Chess vs Poker
Determinism	Deterministic	Stochastic	Calculator vs Weather
Episodic	Episodic	Sequential	Image classifier vs Game
Dynamics	Static	Dynamic	Puzzle vs Trading
Agents	Single-agent	Multi-agent	Crossword vs Auction

Simple Reactive Agent

```
class ReactiveAgent:
    """A simple stimulus-response agent"""

    def __init__(self, rules: dict):
        self.rules = rules  # condition -> action mapping

    def perceive(self, environment) -> dict:
        """Get current state from sensors"""
        return {
            'temperature': environment.get_temp(),
            'humidity': environment.get_humidity(),
            'motion': environment.detect_motion()
        }

    def decide(self, percepts: dict) -> str:
        """Match percepts to rules"""
        for condition, action in self.rules.items():
            if self._matches(percepts, condition):
                return action
        return 'idle'

    def act(self, action: str, environment):
        """Execute the chosen action"""
        environment.execute(action)

    def run_cycle(self, environment):
        """One perception-action cycle"""
        percepts = self.perceive(environment)
        action = self.decide(percepts)
        self.act(action, environment)
        return action

# Example: Thermostat agent
thermostat = ReactiveAgent({
    ('temp < 18',): 'heat_on',
    ('temp > 24',): 'cool_on',
    ('18 <= temp <= 24',): 'maintain'
})
```

Module 2

Swarm Intelligence

Collective behavior from simple rules

Biological Inspiration

Ant Colonies

- Pheromone trails
- Shortest path finding
- Division of labor

Bee Swarms

- Waggle dance
- Collective decision
- Nest site selection

Bird Flocks

- No leader
- Local rules only
- Emergent patterns

Key Insight: Complex global behavior emerges from simple local interactions without central control.

Core Principles

1. Decentralization

No single point of control or failure

2. Self-Organization

Order emerges from local interactions

3. Emergence

Whole is greater than sum of parts

4. Stigmergy

Indirect communication via environment

Advantages:

- Robust to failures
- Scalable
- Flexible/adaptive
- Simple individual agents

Challenges:

- Hard to predict
- Difficult to design
- May converge slowly

Boids Flocking Algorithm

Craig Reynolds' 1986 algorithm for realistic flocking

Separation

Avoid crowding neighbors

Steer away from nearby boids

Alignment

Match velocity of neighbors

Steer towards average heading

Cohesion

Move toward group center

Steer towards average position

```
class Boid:
    def __init__(self, x, y):
        self.position = np.array([x, y])
        self.velocity = np.random.randn(2)

    def update(self, boids, weights):
        neighbors = self.get_neighbors(boids, radius=50)

        sep = self.separation(neighbors) * weights['separation']
        ali = self.alignment(neighbors) * weights['alignment']
        coh = self.cohesion(neighbors) * weights['cohesion']

        self.velocity += sep + ali + coh
        self.velocity = self.limit_speed(self.velocity, max_speed=5)
        self.position += self.velocity
```

Stigmergy: Indirect Communication

Definition: Agents communicate by modifying the environment rather than direct messaging.

Ant Example:

1. Ant finds food
2. Deposits pheromone on return
3. Other ants follow trail
4. Stronger trails = more ants
5. Shortest path emerges

```
class AntColony:
    def __init__(self, grid_size):
        self.pheromones = np.zeros(grid_size)
        self.evaporation = 0.1

    def deposit(self, x, y, amount):
        self.pheromones[x, y] += amount

    def evaporate(self):
        self.pheromones *= (1 - self.evaporation)

    def get_probability(self, x, y):
        # Higher pheromone = higher probability
        return self.pheromones[x, y] ** alpha
```

Flocking Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class FlockSimulation:
    def __init__(self, n_boids=50):
        self.boids = [Boid(
            np.random.rand() * 100,
            np.random.rand() * 100
        ) for _ in range(n_boids)]

        self.weights = {
            'separation': 1.5,
            'alignment': 1.0,
            'cohesion': 1.0
        }

    def step(self):
        for boid in self.boids:
            boid.update(self.boids, self.weights)
            boid.wrap_edges(100, 100)  # Wrap around boundaries

    def animate(self, frames=200):
        fig, ax = plt.subplots(figsize=(8, 8))
        scatter = ax.scatter([], [], c='blue', s=20)

        def update(frame):
            self.step()
            positions = np.array([b.position for b in self.boids])
            scatter.set_offsets(positions)
            return scatter,

        anim = FuncAnimation(fig, update, frames=frames, interval=50)
        plt.show()

# Run simulation
sim = FlockSimulation(n_boids=100)
sim.animate()
```

Module 3

Multi-Agent Systems Architecture

Designing collaborative agent networks

System Topologies

Centralized

- Single coordinator
- Easy to manage
- Single point of failure

Decentralized

- Peer-to-peer
- No single failure point
- Complex coordination

Hierarchical

- Tree structure
- Clear authority chain
- Scalable

Mesh/Hybrid

- Mixed connections
- Flexible routing
- Best for complex systems

Communication Patterns

Direct Messaging

```
agent_b.receive(
    sender=agent_a,
    message=msg
)
```

Point-to-point

Publish-Subscribe

```
broker.publish(
    topic="prices",
    data=update
)
# Subscribers notified
```

Decoupled

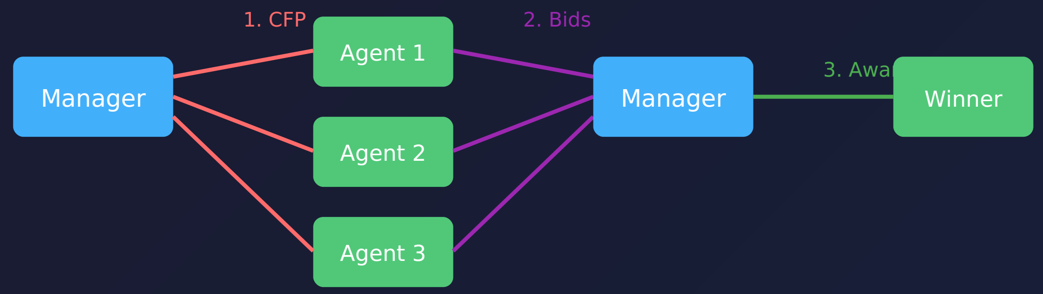
Blackboard

```
blackboard.post(
    key="solution",
    value=partial
)
# All agents see it
```

Shared memory

Contract Net Protocol

A protocol for task allocation among agents



CFP = Call For Proposals

Agents bid based on capability and cost

Message Passing Implementation

```
import asyncio
from dataclasses import dataclass
from typing import Any

@dataclass
class Message:
    sender: str
    receiver: str
    performative: str  # REQUEST, INFORM, PROPOSE, ACCEPT, REJECT
    content: Any

class Agent:
    def __init__(self, name: str):
        self.name = name
        self.inbox = asyncio.Queue()
        self.directory = {}  # Other agents

    async def send(self, receiver: str, performative: str, content: Any):
        msg = Message(self.name, receiver, performative, content)
        await self.directory[receiver].inbox.put(msg)

    async def receive(self) -> Message:
        return await self.inbox.get()

    async def run(self):
        while True:
            msg = await self.receive()
            await self.handle_message(msg)

    async def handle_message(self, msg: Message):
        if msg.performative == "REQUEST":
            # Process request and respond
            response = self.process_request(msg.content)
            await self.send(msg.sender, "INFORM", response)

# Example usage
async def main():
    agent_a = Agent("A")
    agent_b = Agent("B")
    agent_a.directory["B"] = agent_b
    agent_b.directory["A"] = agent_a

    await agent_a.send("B", "REQUEST", {"task": "calculate", "data": [1,2,3]})
```

Coordination Strategies

Cooperative

- Shared goals
- Information sharing
- Task decomposition
- Example: Search & rescue

Competitive

- Conflicting goals
- Strategic behavior
- Game theory applies
- Example: Trading agents

Hybrid (Coopetition)

- Cooperate on some goals
- Compete on others
- Common in real systems
- Example: Supply chains

Key Mechanisms:

- Voting protocols
- Auction mechanisms
- Negotiation
- Social choice

Module 4

Google's A2A Protocol

Agent-to-Agent Communication Standard

A2A Overview

Agent-to-Agent (A2A) is Google's open protocol for agent interoperability, enabling agents built on different frameworks to communicate and collaborate.

Key Features

- Framework agnostic
- HTTP(S) + JSON-RPC 2.0
- Capability discovery
- Task delegation
- Streaming support

Design Principles

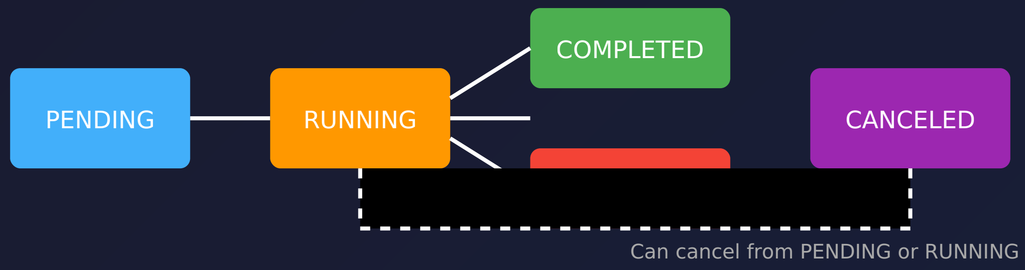
- **Opaque:** Internal details hidden
- **Secure:** Auth & encryption
- **Async:** Long-running tasks
- **Extensible:** Custom capabilities

Agent Cards

Every A2A agent exposes an Agent Card describing its capabilities

```json
{
    "name": "DataAnalysisAgent",
    "description": "Analyzes datasets and generates insights",
    "url": "https://agent.example.com/a2a",
    "version": "1.0.0",
    "capabilities": {
        "streaming": true,
        "pushNotifications": false
    },
    "skills": [
        {
            "id": "analyze_data",
            "name": "Data Analysis",
            "description": "Perform statistical analysis on datasets",
            "inputSchema": {
                "type": "object",
                "properties": {
                    "data": {"type": "array"},
                    "analysis_type": {"type": "string"}
                }
            },
            "outputSchema": {
                "type": "object",
                "properties": {
                    "results": {"type": "object"},
                    "visualizations": {"type": "array"}
                }
            }
        }
    ],
    "authentication": {
        "type": "bearer"
    }
}
```

Task Lifecycle



Streaming Updates:

Long-running tasks can send progress updates via Server-Sent Events (SSE)

Error Handling:

Failed tasks include error codes and messages for debugging

A2A Implementation

```
from flask import Flask, request, jsonify
import json

app = Flask(__name__)

# Agent Card endpoint
@app.route('/.well-known/agent.json')
def agent_card():
    return jsonify({
        "name": "CalculatorAgent",
        "description": "Performs mathematical calculations",
        "url": "http://localhost:5000/a2a",
        "skills": [{
            "id": "calculate",
            "name": "Calculate",
            "inputSchema": {
                "type": "object",
                "properties": {
                    "expression": {"type": "string"}
                }
            }
        }]
    })

# JSON-RPC endpoint
@app.route('/a2a', methods=['POST'])
def handle_request():
    data = request.json
    method = data.get('method')
    params = data.get('params', {})

    if method == 'tasks/create':
        task_id = create_task(params)
        return jsonify({"jsonrpc": "2.0", "result": {"taskId": task_id}, "id": data['id']})

    elif method == 'tasks/get':
        task = get_task(params['taskId'])
        return jsonify({"jsonrpc": "2.0", "result": task, "id": data['id']})

    return jsonify({"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}})
```

A2A Client

```
import requests
import json

class A2AClient:
    def __init__(self, agent_url: str):
        self.agent_url = agent_url
        self.agent_card = self._fetch_agent_card()

    def _fetch_agent_card(self):
        response = requests.get(f"{self.agent_url}/.well-known/agent.json")
        return response.json()

    def _rpc_call(self, method: str, params: dict):
        payload = {
            "jsonrpc": "2.0",
            "method": method,
            "params": params,
            "id": 1
        }
        response = requests.post(f"{self.agent_url}/a2a", json=payload)
        return response.json()

    def create_task(self, skill_id: str, input_data: dict):
        return self._rpc_call("tasks/create", {
            "skillId": skill_id,
            "input": input_data
        })

    def get_task(self, task_id: str):
        return self._rpc_call("tasks/get", {"taskId": task_id})

    def list_skills(self):
        return self.agent_card.get('skills', [])

# Usage
client = A2AClient("http://localhost:5000")
print("Available skills:", client.list_skills())
result = client.create_task("calculate", {"expression": "2 + 2"})
```

Module 5

Advanced Agent Development

Memory, learning, and fault tolerance

Agent Memory Systems

Short-Term

Current context

- Conversation history
- Working memory
- Limited capacity

Long-Term

Persistent knowledge

- Vector databases
- Semantic search
- Facts & procedures

Episodic

Past experiences

- Event sequences
- Temporal context
- Learning from history

Agent with Memory

```python
from langchain.memory import ConversationBufferWindowMemory
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings

class MemoryAgent:
    def __init__(self):
        # Short-term: Last K conversations
        self.short_term = ConversationBufferWindowMemory(k=5)

        # Long-term: Vector store for semantic retrieval
        self.long_term = Chroma(
            embedding_function=OpenAIEmbeddings(),
            persist_directory="./agent_memory"
        )

        # Episodic: List of past experiences
        self.episodic = []

    def remember_short(self, input_text: str, output_text: str):
        self.short_term.save_context(
            {"input": input_text},
            {"output": output_text}
        )

    def remember_long(self, text: str, metadata: dict = None):
        self.long_term.add_texts([text], metadatas=[metadata or {}])

    def remember_episode(self, episode: dict):
        self.episodic.append({
            "timestamp": datetime.now(),
            **episode
        })

    def recall(self, query: str, k: int = 3):
        # Combine short-term context with long-term retrieval
        context = self.short_term.load_memory_variables({})
        relevant = self.long_term.similarity_search(query, k=k)
        return {"context": context, "relevant_docs": relevant}
```

Fault Tolerance Patterns

Retry with Backoff

```
import time
from functools import wraps

def retry(max_attempts=3, backoff=2):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise
                    time.sleep(backoff ** attempt)
        return wrapper
    return decorator

@retry(max_attempts=3)
def risky_operation():
    # May fail temporarily
    pass
```

Circuit Breaker

```
class CircuitBreaker:
    def __init__(self, threshold=5):
        self.failures = 0
        self.threshold = threshold
        self.state = "CLOSED"

    def call(self, func, *args):
        if self.state == "OPEN":
            raise Exception("Circuit open")

        try:
            result = func(*args)
            self.failures = 0
            return result
        except Exception:
            self.failures += 1
            if self.failures >= self.threshold:
                self.state = "OPEN"
            raise
```

State Management

```
from enum import Enum, auto
from typing import Optional

class AgentState(Enum):
    IDLE = auto()
    PERCEIVING = auto()
    REASONING = auto()
    ACTING = auto()
    WAITING = auto()
    ERROR = auto()

class StatefulAgent:
    def __init__(self):
        self.state = AgentState.IDLE
        self.state_data = {}
        self.history = []

    def transition(self, new_state: AgentState, data: dict = None):
        self.history.append({
            "from": self.state,
            "to": new_state,
            "timestamp": datetime.now()
        })
        self.state = new_state
        self.state_data = data or {}

    def save_checkpoint(self, path: str):
        checkpoint = {
            "state": self.state.name,
            "state_data": self.state_data,
            "history": self.history[-100:]  # Last 100 transitions
        }
        with open(path, 'w') as f:
            json.dump(checkpoint, f)

    def restore_checkpoint(self, path: str):
        with open(path) as f:
            checkpoint = json.load(f)
        self.state = AgentState[checkpoint["state"]]
        self.state_data = checkpoint["state_data"]
```

Module 6

Frameworks and Tools

Mesa, PySwarm, Ray, and RLlib

Framework Comparison

Framework	Best For	Key Features	Scale
Mesa	Agent-based modeling	Visualization, data collection	Single machine
PySwarm	Swarm optimization	PSO algorithms	Single machine
Ray	Distributed computing	Actors, scaling	Cluster
RLlib	Multi-agent RL	Training, policies	Cluster

Mesa: Agent-Based Modeling

```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
from mesa.datacollection import DataCollector

class SchellingAgent(Agent):
    def __init__(self, unique_id, model, agent_type):
        super().__init__(unique_id, model)
        self.type = agent_type

    def step(self):
        neighbors = self.model.grid.get_neighbors(
            self.pos, moore=True, include_center=False
        )
        similar = sum(1 for n in neighbors if n.type == self.type)

        if len(neighbors) > 0 and similar / len(neighbors) < 0.3:
            self.model.grid.move_to_empty(self)

class SchellingModel(Model):
    def __init__(self, width, height, density):
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)

        for i, cell in enumerate(self.grid.coord_iter()):
            if self.random.random() < density:
                agent_type = 0 if self.random.random() < 0.5 else 1
                agent = SchellingAgent(i, self, agent_type)
                self.grid.place_agent(agent, cell[1:])
                self.schedule.add(agent)

    def step(self):
        self.schedule.step()
```

Ray: Distributed Agents

```python
import ray

ray.init()

@ray.remote
class DistributedAgent:
    def __init__(self, agent_id):
        self.agent_id = agent_id
        self.state = {}

    def perceive(self, environment_ref):
        env = ray.get(environment_ref)
        return env.get_local_state(self.agent_id)

    def decide(self, percepts):
        # Decision logic
        return {"action": "move", "direction": "north"}

    def act(self, action, environment_ref):
        env = ray.get(environment_ref)
        return env.apply_action(self.agent_id, action)

# Create distributed agents
agents = [DistributedAgent.remote(i) for i in range(100)]

# Run in parallel
futures = [agent.perceive.remote(env_ref) for agent in agents]
percepts = ray.get(futures)

# All agents decide and act in parallel
actions = ray.get([agent.decide.remote(p) for agent, p in zip(agents, percepts)])
results = ray.get([agent.act.remote(a, env_ref) for agent, a in zip(agents, actions)])
```

RLlib: Multi-Agent RL

```
from ray.rllib.algorithms.ppo import PPOConfig
from ray.tune.registry import register_env

# Define multi-agent environment
def env_creator(env_config):
    return MultiAgentTrafficEnv(env_config)

register_env("traffic", env_creator)

# Configure multi-agent training
config = (
    PPOConfig()
    .environment("traffic")
    .multi_agent(
        policies={
            "traffic_light": (None, obs_space, act_space, {}),
            "vehicle": (None, obs_space, act_space, {}),
        },
        policy_mapping_fn=lambda agent_id, *args:
            "traffic_light" if "light" in agent_id else "vehicle"
    )
    .training(
        gamma=0.99,
        lr=0.0003,
        train_batch_size=4000
    )
)

# Train
algo = config.build()
for i in range(100):
    result = algo.train()
    print(f"Iteration {i}: reward = {result['episode_reward_mean']}")
```

Module 7

Swarm Optimization

PSO, ACO, and Consensus Algorithms

Particle Swarm Optimization

> PSO optimizes by having particles fly through the search space, attracted to their personal best and the global best positions.

```
import numpy as np

class PSO:
    def __init__(self, n_particles, dimensions, bounds):
        self.n_particles = n_particles
        self.dimensions = dimensions
        self.bounds = bounds

        # Initialize particles
        self.positions = np.random.uniform(
            bounds[0], bounds[1], (n_particles, dimensions)
        )
        self.velocities = np.zeros((n_particles, dimensions))
        self.personal_best_pos = self.positions.copy()
        self.personal_best_val = np.full(n_particles, np.inf)
        self.global_best_pos = None
        self.global_best_val = np.inf

    def optimize(self, objective_func, iterations=100, w=0.7, c1=1.5, c2=1.5):
        for _ in range(iterations):
            # Evaluate fitness
            fitness = np.array([objective_func(p) for p in self.positions])

            # Update personal bests
            improved = fitness < self.personal_best_val
            self.personal_best_pos[improved] = self.positions[improved]
            self.personal_best_val[improved] = fitness[improved]

            # Update global best
            best_idx = np.argmin(fitness)
            if fitness[best_idx] < self.global_best_val:
                self.global_best_val = fitness[best_idx]
                self.global_best_pos = self.positions[best_idx].copy()

            # Update velocities and positions
            r1, r2 = np.random.rand(2)
            self.velocities = (w * self.velocities +
                c1 * r1 * (self.personal_best_pos - self.positions) +
                c2 * r2 * (self.global_best_pos - self.positions))
            self.positions += self.velocities
```

Ant Colony Optimization

```python
class AntColonyOptimization:
    def __init__(self, distances, n_ants, alpha=1, beta=2, evaporation=0.5):
        self.distances = distances
        self.n_cities = len(distances)
        self.n_ants = n_ants
        self.alpha = alpha  # Pheromone importance
        self.beta = beta    # Distance importance
        self.evaporation = evaporation
        self.pheromones = np.ones((self.n_cities, self.n_cities))

    def run(self, iterations=100):
        best_path = None
        best_distance = np.inf

        for _ in range(iterations):
            paths = [self._construct_path() for _ in range(self.n_ants)]

            # Update pheromones
            self.pheromones *= (1 - self.evaporation)
            for path in paths:
                distance = self._path_distance(path)
                if distance < best_distance:
                    best_distance = distance
                    best_path = path
                self._deposit_pheromones(path, distance)

        return best_path, best_distance

    def _construct_path(self):
        path = [np.random.randint(self.n_cities)]
        while len(path) < self.n_cities:
            probs = self._transition_probs(path[-1], path)
            next_city = np.random.choice(self.n_cities, p=probs)
            path.append(next_city)
        return path
```

Raft Consensus Algorithm

Key Concepts

- **Leader Election:** One leader per term
- **Log Replication:** Leader sends entries
- **Safety:** Committed = replicated to majority

Node States

- **Follower:** Default state
- **Candidate:** Seeking votes
- **Leader:** Handles requests

```
class RaftNode:
    def __init__(self, node_id, peers):
        self.id = node_id
        self.peers = peers
        self.state = "follower"
        self.term = 0
        self.voted_for = None
        self.log = []
        self.commit_index = 0

    def request_vote(self, term, candidate_id):
        if term > self.term:
            self.term = term
            self.voted_for = None

        if (self.voted_for is None and
            term >= self.term):
            self.voted_for = candidate_id
            return True
        return False

    def become_candidate(self):
        self.state = "candidate"
        self.term += 1
        votes = 1  # Vote for self
        # Request votes from peers
        # If majority: become leader
```

Module 8

Real-World Applications

Case studies in production systems

Warehouse Automation (Kiva/Amazon)

System Overview

- 1000s of robots per warehouse
- Decentralized path planning
- Dynamic task assignment
- Collision avoidance

Key Algorithms

- A* for path planning
- Hungarian algorithm for assignment
- Auction-based task allocation

Results:

- 4x faster order fulfillment
- 50% less floor space needed
- Near-zero collision rate

Multi-Agent Techniques:

- Stigmergy (virtual paths)
- Swarm coordination
- Distributed consensus

Smart City Applications

Traffic Control

- Adaptive signals
- Emergency priority
- Congestion prediction

Energy Grid

- Demand response
- Renewable integration
- Peer-to-peer trading

Emergency Response

- Resource dispatch
- Route optimization
- Cross-agency coord.

Financial Trading Systems

```
class TradingAgent:
    def __init__(self, strategy: str, capital: float):
        self.strategy = strategy
        self.capital = capital
        self.portfolio = {}
        self.orders = []

    def analyze_market(self, market_data: dict):
        if self.strategy == "momentum":
            return self._momentum_signal(market_data)
        elif self.strategy == "mean_reversion":
            return self._mean_reversion_signal(market_data)

    def _momentum_signal(self, data):
        # Buy if price trending up
        returns = data['returns'][-10:]
        if np.mean(returns) > 0.01:
            return {"action": "buy", "confidence": 0.8}
        return {"action": "hold", "confidence": 0.5}

    def execute(self, signal, exchange):
        if signal['action'] == 'buy' and signal['confidence'] > 0.7:
            order = exchange.submit_order(
                symbol="AAPL",
                quantity=self.calculate_position_size(),
                order_type="market"
            )
            self.orders.append(order)
```

Module 9

Monitoring and Troubleshooting

Observability for multi-agent systems

Key Metrics

Agent-Level

- Response time
- Success/failure rate
- Resource usage (CPU, memory)
- Message queue depth

System-Level

- Throughput (tasks/second)
- Latency distribution
- Convergence time
- Network traffic

```
from prometheus_client import Counter, Histogram

agent_requests = Counter(
    'agent_requests_total',
    'Total agent requests',
    ['agent_id', 'action']
)

response_time = Histogram(
    'agent_response_seconds',
    'Agent response time',
    ['agent_id']
)

class MonitoredAgent:
    def act(self, action):
        with response_time.labels(
            self.agent_id
        ).time():
            result = self._execute(action)

        agent_requests.labels(
            self.agent_id, action
        ).inc()

        return result
```

Debugging Distributed Behavior

Common Issues:

- **Deadlock:** Agents waiting on each other
- **Livelock:** Agents repeatedly changing state but making no progress
- **Starvation:** Some agents never get resources
- **Message loss:** Network failures

```
import logging

class DebugAgent:
    def __init__(self, agent_id):
        self.logger = logging.getLogger(f"Agent-{agent_id}")
        self.logger.setLevel(logging.DEBUG)

    def send_message(self, recipient, message):
        self.logger.debug(f"SEND -> {recipient}: {message[:100]}")
        # Add correlation ID for tracing
        message['correlation_id'] = str(uuid.uuid4())
        message['timestamp'] = datetime.now().isoformat()
        self._send(recipient, message)

    def receive_message(self, message):
        self.logger.debug(f"RECV <- {message['sender']}: correlation={message['correlation_id']}")
        self._process(message)
```

Module 10

Ethics and Future Directions

Responsible AI agent development

Ethical Considerations

Accountability

- Who is responsible for agent actions?
- Audit trails and logging
- Human oversight requirements

Transparency

- Explainable decisions
- Clear agent identification
- Disclosed limitations

Safety

- Bounded autonomy
- Kill switches
- Graceful degradation

Privacy

- Data minimization
- Consent for data use
- Secure communication

Emerging Trends

LLM Agents

Large language models as agent brains

- Natural language understanding
- Reasoning capabilities
- Tool use

Neuromorphic

Brain-inspired computing

- Event-driven processing
- Low power consumption
- Real-time learning

Quantum Agents

Quantum-enhanced optimization

- Faster search
- Better optimization
- Novel algorithms

Module 11

Capstone Project

Build a complete multi-agent system

Project Options

Option A: Warehouse

- Multi-robot coordination
- Task allocation
- Path planning
- Performance metrics

Option B: Smart Grid

- Energy producers/consumers
- Price negotiation
- Load balancing
- Renewable integration

Option C: Trading

- Multiple strategies
- Market simulation
- Risk management
- Performance analysis

Requirements

Technical

- Minimum 5 agents
- A2A protocol for communication
- At least 2 agent types
- Visualization dashboard
- Monitoring and logging

Deliverables

- Working code (GitHub)
- Architecture documentation
- Performance analysis report
- 10-minute demo presentation
- Lessons learned

Evaluation Criteria: Functionality (40%), Code Quality (20%), Documentation (20%), Presentation (20%)

Thank You!

Mastering AI Agents

You now have the foundation to build intelligent multi-agent systems

Key Takeaways:

- Agent architectures: Reactive, Deliberative, BDI
- Swarm intelligence: Emergence from simple rules
- Communication: A2A protocol for interoperability
- Frameworks: Mesa, Ray, RLlib for production
- Applications: From warehouses to trading floors