Multi-Agent Load Balancing and Collaboration via Reinforcement Learning

compiled by D.Gueorguiev with the help of Claude Opus 4.5, 1/4/2026

Table of Contents

[Overview of the System 2](#_Toc222346394)

[Agents Roles and Hierarchy 4](#_Toc222346395)

[Understanding the Action Space for the PPO Model 4](#_Toc222346396)

[Action Structure 4](#_Toc222346397)

[1. Continuous Actions for Nuanced Control 7](#_Toc222346398)

[2. Decentralized Yet Coordinated 7](#_Toc222346399)

[3. Sparse Collaboration Signal 7](#_Toc222346400)

[4. Emergent Team Behavior 7](#_Toc222346401)

[Initializing the Agents 8](#_Toc222346402)

[Task Generation 8](#_Toc222346403)

[Create PPO Model 9](#_Toc222346404)

[Set up Model Training Callbacks 10](#_Toc222346405)

[Train the PPO Model 10](#_Toc222346406)

[Task Processing 10](#_Toc222346407)

[Collaboration Matrix 11](#_Toc222346408)

[Purpose of the Collaboration Matrix 11](#_Toc222346409)

[Update of the Collaboration Matrix 12](#_Toc222346410)

[Observation Generation 13](#_Toc222346411)

[Step Execution 14](#_Toc222346412)

[Evaluate the PPO Model 15](#_Toc222346413)

[Problem with the PPO Model using real LLM calls in the Agentic Workflow 16](#_Toc222346414)

[Class Diagrams 17](#_Toc222346415)

[Core Classes 18](#_Toc222346416)

[Smart Agent and Tool System Classes 19](#_Toc222346417)

[System Flowchart and Training Process 19](#_Toc222346418)

[Training Sequence 21](#_Toc222346419)

[Training Loop 22](#_Toc222346420)

[Parallel Vectorized Environments 23](#_Toc222346421)

[Initialize the PPO model 23](#_Toc222346422)

[Key PPO hyperparameters 24](#_Toc222346423)

[PPO Hyperparameters deep dive 24](#_Toc222346424)

[More Details on Task Processing and Task Management 24](#_Toc222346425)

[Task Driven Work Distribution 25](#_Toc222346426)

[**Task Distribution Strategy** 25](#_Toc222346427)

[Agent Task Execution Flowchart 26](#_Toc222346428)

[Task Lifecycle Flowchart 27](#_Toc222346429)

[Task States 29](#_Toc222346430)

[Interpreting the Results of Agentic RL System Execution 30](#_Toc222346431)

[References 30](#_Toc222346432)

[Appendix 31](#_Toc222346433)

[Key Concepts in Reinforcement Learning 31](#_Toc222346434)

[Bellman’s Equations for Markov Decision Processes 33](#_Toc222346435)

[Actor-Critic Reinforcement Learning Algorithms 33](#_Toc222346436)

[Policy Gradients 36](#_Toc222346437)

[Generalized Advantage Estimation 36](#_Toc222346438)

[Proximal Policy Optimization 37](#_Toc222346439)

[A Note on nn.Sequential implemented with PyTorch Deep Learning library 37](#_Toc222346440)

[Example 1: MNIST-style digit classification (0-9) using nn.Sequential 37](#_Toc222346441)

[Example 2: Deterministic Policy in continuous action space modeled with nn.Sequential 41](#_Toc222346442)

# Overview of the System

We will discuss application of RL algorithms such as Proximal Policy Optimization (PPO) to implement adaptive workflow which will manage learning-based task allocation. In this document we will discuss the application of the PPO for the *simulated* agentic workflow illustrated in the notebook [1].

Key structures are AgentRole, AgentState, and Task for tracking agent roles, agent states and tasks in the system.

class AgentRole(Enum):

"""Defines specialized agent roles in the system"""

RESEARCHER = "researcher"

ANALYZER = "analyzer"

EXECUTOR = "executor"

VALIDATOR = "validator"

COORDINATOR = "coordinator"

@dataclass

class Task:

"""Represents a task in the workflow"""

id: str

type: str

complexity: float

requirements: List[str]

deadline: float

priority: float

status: str = "pending"

assigned\_agent: Optional[str] = None

completion\_time: Optional[float] = None

quality\_score: Optional[float] = None

@dataclass

class AgentState:

"""Tracks individual agent state"""

id: str

role: AgentRole

capacity: float

expertise: Dict[str, float]

current\_load: float = 0.0

completed\_tasks: int = 0

success\_rate: float = 1.0

collaboration\_score: float = 1.0

## Agents Roles and Hierarchy

A screenshot of a computer

AI-generated content may be incorrect.

## Understanding the Action Space for the PPO Model

Recall, the Action Space is defined as

self.action\_space = spaces.Box(

low=0, high=1, shape=(n\_agents \* 3,), dtype=np.float32

)

That is, the PPO model outputs a flat vector of continuous values between 0 and 1, with a total size of n\_agents × 3.

### Action Structure

When the step() function receives this flat action vector, it reshapes it:

action = action.reshape(self.n\_agents, 3)

This creates a 2D matrix where each row represents one agent's actions:

|  |  |  |  |
| --- | --- | --- | --- |
| Agent | Action 0 (Task Assignment) | Action 1 (Resource Allocation) | Action 2 (Collab Request) |
| 0 | action[0,0] | action[0,1] | action[0,2] |
| 1 | action[1,0] | action[1,1] | action[1,2] |
| 2 | action[2,0] | action[2,1] | action[2,2] |
| 3 | action[3,0] | action[3,1] | action[3,2] |

Details on the Action Components

**Action Component 0: Task Assignment Decision** (action[i, 0])

Determines whether agent i should take a task from the queue.

Use:

if len(self.task\_queue) > 0 and action[i, 0] > 0.5:

task = self.task\_queue.popleft()

self.\_assign\_task(agent, task)

Interpretation:

* action[i, 0] > 0.5: Agent accepts the next task from the queue
* action[i, 0] <= 0.5: Agent declines to take a task this step

**Example**:

**Scenario**: Queue has tasks [research\_task, analysis\_task, validation\_task]

**PPO outputs**: action[0, 0] = 0.82 → Agent 0 TAKES research\_task (first in queue)

action[1, 0] = 0.34 → Agent 1 SKIPS (doesn't take any task)

action[2, 0] = 0.91 → Agent 2 TAKES analysis\_task (now first in queue)

action[3, 0] = 0.12 → Agent 3 SKIPS

**Result**:

- research\_task assigned to Agent 0

- analysis\_task assigned to Agent 2

- validation\_task remains in queue

**Action Component 1: Resource Allocation** (action[i, 1])

Intended for how much of the agent's capacity to allocate to current work.

**Current Implementation**: This component is defined in the action space but not currently used in the step() function. It's a placeholder for potential extensions like:

* Throttling agent effort (energy management)
* Splitting focus between multiple tasks
* Quality vs. speed trade-offs

**Future Use Example**:

# Potential implementation:

effective\_capacity = agent.capacity \* action[i, 1]

progress = agent.expertise.get(task.type, 0.5) \* effective\_capacity

**Action Component 2: Collaboration Request Signal** (action[i, 2])

Signals whether the agent wants to collaborate with other agents.

Here is how it is used:

for i in range(self.n\_agents):

for j in range(self.n\_agents):

if i != j and action[i, 2] > 0.7 and action[j, 2] > 0.7:

self.collaboration\_matrix[i, j] \*= 1.01

reward += 0.5 # Collaboration bonus

Interpretation:

action[i, 2] > 0.7: Agent is signaling willingness to collaborate

action[i, 2] >= 0.7: Agent is working independently

**Note**: Collaboration only happens when BOTH agents signal collaboration (mutual consent).

**Example**:

PPO outputs: action[0, 2] = 0.85 → Agent 0 WANTS to collaborate

action[1, 2] = 0.92 → Agent 1 WANTS to collaborate

action[2, 2] = 0.45 → Agent 2 working solo

action[3, 2] = 0.78 → Agent 3 WANTS to collaborate

Collaboration Analysis:

- Agents 0 & 1: BOTH > 0.7 → collaboration\_matrix[0,1] \*= 1.01, +0.5 reward

- Agents 0 & 2: Only 0 > 0.7 → NO collaboration (Agent 2 declined)

- Agents 0 & 3: BOTH > 0.7 → collaboration\_matrix[0,3] \*= 1.01, +0.5 reward

- Agents 1 & 2: Only 1 > 0.7 → NO collaboration

- Agents 1 & 3: BOTH > 0.7 → collaboration\_matrix[1,3] \*= 1.01, +0.5 reward

- Agents 2 & 3: Only 3 > 0.7 → NO collaboration

Total collaboration reward this step: pairs directions =

**Complete Action Example**

Let's trace through a full example with 4 agents:

Raw PPO Output (flat vector):

ppo\_action = np.array([

0.82, 0.65, 0.85, # Agent 0: take task, medium resource, collaborate

0.34, 0.90, 0.92, # Agent 1: skip task, high resource, collaborate

0.91, 0.45, 0.45, # Agent 2: take task, low resource, work solo

0.12, 0.70, 0.78 # Agent 3: skip task, medium resource, collaborate

])

After Reshape:

action = ppo\_action.reshape(4, 3)

# array([[0.82, 0.65, 0.85],

# [0.34, 0.90, 0.92],

# [0.91, 0.45, 0.45],

# [0.12, 0.70, 0.78]])

What Happens This Step:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agent | Task Assignment | Resource | Collaboration | Result |
| 0 | 0.82 > 0.5 ✓ | 0.65 (unused) | 0.85 > 0.7 ✓ | Takes task, signals collab |
| 1 | 0.34 ≤ 0.5 ✗ | 0.90 (unused) | 0.92 > 0.7 ✓ | No task, signals collab |
| 2 | 0.91 > 0.5 ✓ | 0.45 (unused) | 0.45 ≤ 0.7 ✗ | Takes task, works alone |
| 3 | 0.12 ≤ 0.5 ✗ | 0.70 (unused) | 0.78 > 0.7 ✓ | No task, signals collab |

Collaboration matrix updates:

Before: After:

[[1.0, 1.0, 1.0, 1.0], [[1.00, 1.01, 1.00, 1.01],

[1.0, 1.0, 1.0, 1.0], → [1.01, 1.00, 1.00, 1.01],

[1.0, 1.0, 1.0, 1.0], [1.00, 1.00, 1.00, 1.00], ← Agent 2 didn't collab

[1.0, 1.0, 1.0, 1.0]] [1.01, 1.01, 1.00, 1.00]]

**Benefits of this design**:

### 1. Continuous Actions for Nuanced Control

Using Box space with values 0-1 allows the PPO policy to learn subtle preferences:

* 0.51 = barely willing to take a task
* 0.99 = very eager to take a task

### 2. Decentralized Yet Coordinated

Each agent has its own action components, but collaboration requires mutual agreement. The policy must learn to:

* Coordinate collaboration timing
* Match complementary agents

### 3. Sparse Collaboration Signal

The 0.7 threshold for collaboration creates a clear distinction:

* Below 0.7: Focus on individual work
* Above 0.7: Actively seeking collaboration

### 4. Emergent Team Behavior

Over training, the PPO policy learns patterns like:

* "When facing complex tasks, signal collaboration"
* "Agent 1 (Analyzer) should collaborate with Agent 0 (Researcher)"
* "Don't overload any single agent with tasks"

┌──────────────────────────────────────────────────────────────────┐

│ PPO ACTION VECTOR │

│ [a0\_task, a0\_res, a0\_col, a1\_task, a1\_res, a1\_col, ...] │

└──────────────────────────────────────────────────────────────────┘

│

▼ reshape(n\_agents, 3)

┌──────────────────────────────────────────────────────────────────┐

│ AGENT 0 AGENT 1 │

│ ┌─────────┬─────────┬─────────┐ ┌─────────┬─────────┬────────┐ │

│ │ Task │Resource │ Collab │ │ Task │Resource │ Collab │ │

│ │ >0.5? │(future) │ >0.7? │ │ >0.5? │(future) │ >0.7? │ │

│ │ ↓ │ │ ↓ │ │ ↓ │ │ ↓ │ │

│ │ Assign │ │ Update │ │ Assign │ │ Update │ │

│ │ Task │ │ Matrix │ │ Task │ │ Matrix │ │

│ └─────────┴─────────┴─────────┘ └─────────┴─────────┴────────┘ │

└──────────────────────────────────────────────────────────────────┘

## Initializing the Agents

def **\_initialize\_agents**(self) -> List[AgentState]:

"""Initialize agents with diverse roles and capabilities"""

agents = []

roles = list(AgentRole)[:self.n\_agents]

for i, role in enumerate(roles):

expertise = {

"research": np.random.uniform(0.5, 1.0),

"analysis": np.random.uniform(0.5, 1.0),

"execution": np.random.uniform(0.5, 1.0),

"validation": np.random.uniform(0.5, 1.0)

}

# Boost expertise based on role

if role == AgentRole.RESEARCHER:

expertise["research"] = min(1.0, expertise["research"] + 0.3)

elif role == AgentRole.ANALYZER:

expertise["analysis"] = min(1.0, expertise["analysis"] + 0.3)

agents.append(AgentState(

id=f"agent\_{i}",

role=role,

capacity=np.random.uniform(0.8, 1.0),

expertise=expertise

))

return agents

## Task Generation

The environment of this problem generates tasks probabilistically (30% chance per step) . The code excerpts below illustrate the task generation workflow

def **\_generate\_task**(self) -> Task:

"""Generate a new task with random properties"""

task\_types = ["research", "analysis", "execution", "validation"]

task\_type = np.random.choice(task\_types)

return Task(

id=f"task\_{np.random.randint(10000)}",

type=task\_type,

complexity=np.random.uniform(0.3, 1.0),

requirements=[np.random.choice(task\_types) for \_ in range(np.random.randint(1, 3))],

deadline=np.random.uniform(10, 50),

priority=np.random.uniform(0.1, 1.0))

def **step**(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, bool, Dict]:

"""Execute action and return new state"""

self.current\_step += 1

...

# Generate new tasks

if np.random.random() < 0.3:

self.task\_queue.append(self.\_generate\_task())

...

## Create PPO Model

def **\_create\_model**(self) -> PPO:

"""Create PPO model with custom architecture"""

policy\_kwargs = dict(

net\_arch=[

dict(pi=[256, 256, 128], vf=[256, 256, 128])

],

activation\_fn=nn.ReLU

)

model = PPO(

"MlpPolicy",

self.env,

learning\_rate=3e-4,

n\_steps=2048,

batch\_size=64,

n\_epochs=10,

gamma=0.99,

gae\_lambda=0.95,

clip\_range=0.2,

clip\_range\_vf=None,

ent\_coef=0.01,

vf\_coef=0.5,

max\_grad\_norm=0.5,

policy\_kwargs=policy\_kwargs,

verbose=1,

tensorboard\_log="./tensorboard\_logs/”)

return model

...

self.model = self.\_create\_model()

## Set up Model Training Callbacks

def **\_setup\_callbacks**(self):

"""Setup training callbacks for monitoring and checkpointing"""

eval\_callback = EvalCallback(

self.eval\_env,

best\_model\_save\_path="./models/best\_model/",

log\_path="./logs/",

eval\_freq=5000,

deterministic=True,

render=False,

n\_eval\_episodes=10

)

checkpoint\_callback = CheckpointCallback(

save\_freq=10000,

save\_path="./models/checkpoints/",

name\_prefix="agentic\_rl\_model"

)

return [eval\_callback, checkpoint\_callback]

...

self.callbacks = self.\_setup\_callbacks()

## Train the PPO Model

def train(self, total\_timesteps: int = 100000):

"""Train the multi-agent system"""

logger.info(f"Starting training for {total\_timesteps} timesteps")

self.model.learn(

total\_timesteps=total\_timesteps,

callback=self.callbacks,

log\_interval=10,

progress\_bar=True

)

logger.info("Training completed")

return self.model

## Task Processing

def \_assign\_task(self, agent: AgentState, task: Task):

"""Assign a task to an agent"""

task.assigned\_agent = agent.id

task.status = "active"

self.active\_tasks[task.id] = (task, agent)

agent.current\_load += task.complexity

def \_complete\_task(self, task: Task, agent: AgentState):

"""Mark a task as completed"""

task.status = "completed"

task.completion\_time = self.current\_step

task.quality\_score = agent.expertise.get(task.type, 0.5) \* agent.success\_rate

self.completed\_tasks.append(task)

del self.active\_tasks[task.id]

agent.current\_load = max(0, agent.current\_load - task.complexity)

agent.completed\_tasks += 1

agent.success\_rate = 0.95 \* agent.success\_rate + 0.05 \* task.quality\_score

## Collaboration Matrix

Purpose of the Collaboration Matrix

1. **Tracking Agent Collaboration Patterns**

The matrix is an n\_agents × n\_agents array that records collaboration history between agent pairs:

# Initialized as all 1s (neutral starting point)

self.collaboration\_matrix = np.ones((n\_agents, n\_agents))

2. **Part of the Observation Space**

The flattened collaboration matrix is included in the state observation, allowing the RL policy to "see" collaboration patterns:

# In \_get\_observation()

obs.extend(self.collaboration\_matrix.flatten())

This means the observation dimension includes n\_agents \* n\_agents values from the matrix, letting the policy learn which agent pairs have historically collaborated well.

3. **Incentivizing Collaboration via Rewards**

When two agents both signal high collaboration intent (action value > 0.7), the system:

* Strengthens their collaboration bond (1% increase)
* Provides a reward bonus (+0.5)

# In step()

for i in range(self.n\_agents):

for j in range(self.n\_agents):

if i != j and action[i, 2] > 0.7 and action[j, 2] > 0.7:

self.collaboration\_matrix[i, j] \*= 1.01 # Bond strengthens

reward += 0.5 # Collaboration bonus

4. Action Space Design

Each agent's action has 3 components:

# action.reshape(n\_agents, 3)

# action[i, 0] = task assignment decision

# action[i, 1] = resource allocation

# action[i, 2] = collaboration request signal ← Used for collaboration

| **Benefit** | **How the Matrix Helps** |
| --- | --- |
| **Emergent teamwork** | Agents learn which pairs work well together over time |
| **Memory of relationships** | The matrix persists within an episode, encoding collaboration history |
| **Informed decisions** | Policy can use past collaboration success to make future task assignments |
| **Reward shaping** | Explicit bonus encourages agents to coordinate rather than work in isolation |

Example Flow

Episode Start:

collaboration\_matrix = [[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1]]

After Agent 0 & Agent 2 collaborate multiple times:

collaboration\_matrix = [[1.00, 1.00, 1.05, 1.00],

[1.00, 1.00, 1.00, 1.00],

[1.05, 1.00, 1.00, 1.00], ← Strengthened bond

[1.00, 1.00, 1.00, 1.00]]

The RL policy can then learn to leverage this strengthened bond, preferring to assign related tasks to agents 0 and 2 since their collaboration has been historically successful.

### Update of the Collaboration Matrix

# inside evaluate(n\_episodes)

action, \_ = self.model.predict(obs, deterministic=True)

...

# inside function step()

...

# Update collaboration matrix based on actions

for i in range(self.n\_agents):

for j in range(self.n\_agents):

if i != j and action[i, 2] > 0.7 and action[j, 2] > 0.7:

self.collaboration\_matrix[i, j] \*= 1.01

reward += 0.5 # Collaboration bonus

## Observation Generation

def **\_get\_observation**(self) -> np.ndarray:

"""Construct observation vector from current state"""

obs = []

# Agent states

for agent in self.agents:

obs.extend([

agent.capacity,

agent.current\_load,

agent.completed\_tasks / max(1, self.current\_step),

agent.success\_rate,

agent.collaboration\_score,

agent.expertise.get("research", 0),

agent.expertise.get("analysis", 0)

])

# Task queue state

for i in range(self.max\_tasks):

if i < len(self.task\_queue):

task = list(self.task\_queue)[i]

obs.extend([

task.complexity,

task.priority,

task.deadline,

1.0, # task exists

0.0 # not yet assigned

])

else:

obs.extend([0, 0, 0, 0, 0])

# Collaboration matrix (flattened)

obs.extend(self.collaboration\_matrix.flatten())

return np.array(obs, dtype=np.float32)

## Step Execution

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, bool, Dict]:

"""Execute action and return new state"""

self.current\_step += 1

# Parse actions

action = action.reshape(self.n\_agents, 3)

# Process task assignments

reward = 0

for i, agent in enumerate(self.agents):

if len(self.task\_queue) > 0 and action[i, 0] > 0.5:

task = self.task\_queue.popleft()

self.\_assign\_task(agent, task)

# Calculate immediate reward for assignment

match\_score = agent.expertise.get(task.type, 0.5)

reward += match\_score \* task.priority

# Process active tasks

completed\_this\_step = []

for task\_id, (task, agent) in list(self.active\_tasks.items()):

progress = agent.expertise.get(task.type, 0.5) \* agent.capacity

# Check if task completed

if np.random.random() < progress:

self.\_complete\_task(task, agent)

completed\_this\_step.append(task)

# Calculate completion reward

time\_bonus = max(0, 1 - (self.current\_step / task.deadline))

quality = agent.expertise.get(task.type, 0.5) \* agent.success\_rate

reward += (quality \* task.priority \* (1 + time\_bonus)) \* 10

# Update collaboration matrix based on actions

for i in range(self.n\_agents):

for j in range(self.n\_agents):

if i != j and action[i, 2] > 0.7 and action[j, 2] > 0.7:

self.collaboration\_matrix[i, j] \*= 1.01

reward += 0.5 # Collaboration bonus

# Generate new tasks

if np.random.random() < 0.3:

self.task\_queue.append(self.\_generate\_task())

# Calculate penalties

queue\_penalty = len(self.task\_queue) \* 0.1

overdue\_penalty = sum(1 for t in self.active\_tasks.values()

if self.current\_step > t[0].deadline) \* 0.5

reward -= (queue\_penalty + overdue\_penalty)

# Check termination

done = self.current\_step >= self.max\_steps

truncated = False

# Prepare info

info = {

"completed\_tasks": len(completed\_this\_step),

"queue\_length": len(self.task\_queue),

"active\_tasks": len(self.active\_tasks),

"avg\_success\_rate": np.mean([a.success\_rate for a in self.agents])

}

return self.\_get\_observation(), reward, done, truncated, info

## Evaluate the PPO Model

def evaluate(self, n\_episodes: int = 10) -> Dict[str, float]:

"""Evaluate the trained model"""

env = self.env\_class()

episode\_rewards = []

episode\_lengths = []

task\_completion\_rates = []

for episode in range(n\_episodes):

obs, \_ = env.reset()

done = False

episode\_reward = 0

episode\_length = 0

while not done:

action, \_ = self.model.predict(obs, deterministic=True)

obs, reward, done, truncated, info = env.step(action)

episode\_reward += reward

episode\_length += 1

episode\_rewards.append(episode\_reward)

episode\_lengths.append(episode\_length)

if len(env.completed\_tasks) > 0:

completion\_rate = len(env.completed\_tasks) / (len(env.completed\_tasks) + len(env.task\_queue))

task\_completion\_rates.append(completion\_rate)

metrics = {

"mean\_reward": np.mean(episode\_rewards),

"std\_reward": np.std(episode\_rewards),

"mean\_episode\_length": np.mean(episode\_lengths),

"task\_completion\_rate": np.mean(task\_completion\_rates) if task\_completion\_rates else 0

}

return metrics

## Problem with the PPO Model using real LLM calls in the Agentic Workflow

The problem is that PPO's on-policy, batch-mode design demands an enormous volume of live environment interactions to produce each policy update which will incur substantial cost for the related LLM invocations as well as substantial time to complete. Hence the approach using the robust policy-based PPO becomes impractical in this specific scenario.

Thus, after the introduction of real LLM invocations, the PPO-based RL algorithm has to be replaced with much less training-intensive approach utilizing Q learning. Also the collaboration matrix becomes infeasible with this design changed so it has been dropped from consideration which means the agents will not be collaborating.

Both PPO and Q-Learning are **online RL** algorithms — they both learn by actively interacting with a live environment, not from a pre-collected static dataset. The critical difference is **how much** online interaction each algorithm demands before producing a robust policy. PPO's batch-mode, on-policy design requires **tens of thousands** of live environment interactions per policy update, making it economically prohibitive when each interaction is a paid LLM API call. Q-Learning's incremental, off-policy design learns from **every single interaction** and converges with orders of magnitude fewer samples.

A screenshot of a computer

AI-generated content may be incorrect.

**Problem** : Catastrophic Sample Inefficiency with Real API Calls

The Core Issue

PPO is designed for environments where millions of interactions are cheap (video games, robotics simulators). When each interaction is a real LLM API call costing money and taking seconds, PPO's sample requirements become economically prohibitive.

PPO's Sample Requirements (Original Notebook)

The original notebook configured PPO for 50,000 timesteps.

With evaluation episodes added, the total interaction count exceeds 60,000. At real API prices this results in an estimated cost north of $150.

For comparison if we replace the PPO algorithm with Q Learning we will have no more than 30 iterations and a sample cost of no more than 9 cents per training.

For more details refer to this markdown: [problems\_with\_ppo\_algorithm.md](https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/multi_agent_colab/problems_with_ppo_algorithm.md)

# Class Diagrams

## Core Classes

A diagram of a computer program

AI-generated content may be incorrect.

## Smart Agent and Tool System Classes

A diagram of a software company

AI-generated content may be incorrect.

# System Flowchart and Training Process

A diagram of a process

AI-generated content may be incorrect.

## A diagram of a program AI-generated content may be incorrect.Training Sequence

## A diagram of a company AI-generated content may be incorrect.Training Loop

The training loop in the agentic reinforcement learning system combines Proximal Policy Optimization method for obtaining optimal policy and multi-agent coordination to learn optimal task allocation and collaboration strategies.

The training process begins by creating an instance of the AgenticRLSystem class which serves as an orchestrator for the entire training pipeline.

system = AgenticRLSystem(

env\_class=MultiAgentTaskEnvironment,

n\_envs=4 # Number of parallel environments

)

### Parallel Vectorized Environments

The training process creates vectorized environments. These environments are updated in parallel for efficient experience collection. The vectorized environment paradigm allows to run multiple environment instances independently and simultaneously. Thus we speed up the training by collecting more diverse experiences per training step. Thus the vectorized environment workflow collects n\_envs times more experience per step.

Each environment evolves independently, reducing correlation in training data. More diverse experiences lead to more stable policy updates due to better gradient estimates.

Example implementation:

self.env = DummyVecEnv([lambda: env\_class() for \_ in range(n\_envs)])

Each environment maintains its own agent states (capacity, expertise, workload), task queue with varying priorities and complexities, collaboration matrix tracking inter-agent relationships, independent random seeds for task generation.

### Initialize the PPO model

Creates the neural network architecture and optimization components.

The PPO model consists of two main networks:

Policy Network (Actor): outputs action probabilities

Value Network (Critic): estimates state values for advantage calculation

**Architecture configuration**

policy\_kwargs = dict(

net\_arch=[

dict(pi=[256, 256, 128], # Policy network layers

vf=[256, 256, 128]) # Value network layers

],

activation\_fn=nn.ReLU

)

### Key PPO hyperparameters

learning\_rate=3e-4: Step size for gradient descent

n\_steps=2048: Steps before policy update (trajectory length)

batch\_size=64: Minibatch size for optimization

n\_epochs=10: Number of optimization epochs per update

gamma=0.99: Discount factor for future rewards

gae\_lambda=0.95: GAE parameter for advantage estimation

clip\_range=0.2: PPO clipping parameter to limit policy changes

### PPO Hyperparameters deep dive

**Learning rate**

The learning rate can be viewed as a step size for the gradient descent updates to both policy and value networks.

The custom policy and value networks in the example in [1] have both 3 layers with size .

Conservative learning rate prevents destabilizing multi-agent coordination patterns the network learns.

Too high learning rate would cause the agents to “forget” the successful collaboration strategies.

Too low learning rate means slow adaptation to new task distributions.

With agents × multiple task types, strikes a balance between learning speed and stability. For simpler environments, one might use ; for more complex, .

**Trajectory length**

The trajectory length n\_steps controls the number of environmental steps collected before each policy update.

The environment of this problem generates tasks probabilistically (30% chance per step) . For details refer to Task Generation in the first section of this document.

The reward structure benefits from long trajectories:

- Completion rewards (quality \* priority \* time\_bonus) \* 10

- Collaboration bonuses accumulate

- Queue penalties are time-dependent

//TODO: finish the section on key PPO parameters

# More Details on Task Processing and Task Management

Refer to the markdown [task\_driven\_agentic\_workflow\_with\_rl\_loop.md](https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/task_driven_agentic_workflow_with_rl_loop.md)

## Task Driven Work Distribution

A diagram of a flowchart

AI-generated content may be incorrect.

### **Task Distribution Strategy**

1. **PPO Policy** decides which agent receives each task based on:
   * Agent capacity and current load
   * Task complexity and requirements
   * Historical success rates
   * Collaboration opportunities
2. **Q-Learning** enables each agent to learn optimal tool selection:
   * ε-greedy exploration balances learning vs exploitation
   * Rewards shape tool preferences per task type
   * Tool effectiveness tracking improves over time
3. **Collaboration Matrix** captures inter-agent synergies:
   * Agents requesting collaboration simultaneously get bonuses
   * Matrix values increase with successful collaborations
   * Enables emergent team behaviors

**Reward Structure**

A screenshot of a computer

AI-generated content may be incorrect.

## Agent Task Execution Flowchart

A diagram of a task

AI-generated content may be incorrect.

A diagram of a computer

AI-generated content may be incorrect.

A diagram of a number of objects

AI-generated content may be incorrect.

A diagram of a record in task history

AI-generated content may be incorrect.

## Task Lifecycle Flowchart

A diagram of a process flow

AI-generated content may be incorrect.

A diagram of a process flow

AI-generated content may be incorrect.

## Task States

A diagram of a process

AI-generated content may be incorrect.

# Interpreting the Results of Agentic RL System Execution

# References

[1] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/agentic_rl_workflow.ipynb>

[2] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/agentic_rl_diagrams.md>

[3] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/agentic_rl_training_loop.md>

[4] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/multi_agent_colab/problems_with_ppo_algorithm.md>

[5] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/multi_agent_colab/agentic_rl_workflow_LLM_calls_lb.ipynb>

[6] <https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/notebooks/reinforcement_learning/multi_agent_colab/agentic_rl_workflow_LLM_calls_lb.md>

[7] [Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems, Andrew G. Barto et al, 1983](https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/articles/reinforcement_learning/Neuronlike_adaptive_elements_that_can_solve_difficult_learning_control_problems_Barto1983.pdf)

[8] [Actor-Critic Algorithms, Vijay Konda, John N. Tsitsiklis, 1999](https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/articles/reinforcement_learning/NIPS-1999-actor-critic-algorithms-Paper.pdf)

[9] [Natural Actor-Critic, Jan Peters et al, 2005](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/NaturalActorCritic_Peters_2005.pdf)

[10] [Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients, Ivo Grondman et al, 2012](https://github.com/dimitarpg13/agentic_architectures_and_design_patterns/blob/main/articles/reinforcement_learning/Survey_of_Actor-Critic_Reinforcement_Learning-Standard_and_Natural_Policy_Gradients_Grondman_2012.pdf)

[11] [Policy Gradient Methods for Reinforcement Learning with Function Approximation, Richard Sutton et al, AT&T, 1999](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/NIPS-1999-policy-gradient-methods-for-reinforcement-learning-with-function-approximation-Paper.pdf)

[12] [Going Deeper Into Reinforcement Learning: Fundamentals of Policy Gradients, Seita’s Place Blog, March 2017](https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/)

[13] [Notes on the Generalized Advantage Estimation Paper, Seita’s Place blog, April 2017](https://danieltakeshi.github.io/2017/04/02/notes-on-the-generalized-advantage-estimation-paper/)

[14] [High-Dimensional Continuous Control Using Generalized Advantage Estimation, John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, Pieter Abbeel, 2015](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/High-Dimensional_Continuous_Control_Using_Generalized_Advantage_Estimation_Schulman_2015.pdf)

[15] [Human-level control through deep reinforcement learning, Volodymyr Mnih et al, Nature, 2015](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/Human-level_control_through_deep_reinforcement_learning_Mnih_2015.pdf)

[16] [Asynchronous methods for deep reinforcement learning, Volodymyr Mnih et al, Google DeepMind, 2016](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/Asynchronous_Methods_for_Deep_Reinforcement_Learning_Mnih_2016.pdf)

[17] [Trust Region Policy Optimization, John Schulman et al, OpenAI, 2017](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/TrustRegionPolicyOptimization_Schulman_2015.pdf)

[18]  [Proximal Policy Optimization Algorithms, John Schulman et al, OpenAI, 2017](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/articles/ReinforcementLearning/Proximal_Policy_Optimization_Algorithms_Shulman_2017.pdf)

[19] [PPO for LLMs: A Guide for Normal People, Cameron Wolfe, Substack, Oct 27th, 2025](https://cameronrwolfe.substack.com/p/ppo-llm)

[20] [Key Concepts in Reinforcement Learning, OpenAI Spinning Up (Part1)](https://spinningup.openai.com/en/latest/spinningup/rl_intro.html)

[21] [Kinds of Reinforcement Learning Algorithms, OpenAI Spinning Up (Part 2)](https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html)

[22] [Intro to Policy Optimization, OpenAI Spinning Up (Part 3)](https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html)

[23] [Vanila Policy Gradient, OpenAI docs](https://spinningup.openai.com/en/latest/algorithms/vpg.html)

[24] [Trust Region Policy Optimization, OpenAI docs](https://spinningup.openai.com/en/latest/algorithms/trpo.html)

[25] [Proximal Policy Optimization, OpenAI docs](https://spinningup.openai.com/en/latest/algorithms/ppo.html)

[26] [Note on Q functions and V functions in Reinforcement Learning, D. Gueorguiev, 2023](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/docs/Note_on_Q_functions_in_Reinforcement_Learning.pdf)

[27] [Notes on Reinforcement Learning by David Silver (notes taken by D. Gueorguiev), 2026](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/docs/Notes_on_Reinforcement_Learning_by_David_Silver.pdf)

[28] [Notes on Reinforcement Learning and Deep Reinforcement Learning , D. Gueorguiev, 2025](https://github.com/dimitarpg13/reinforcement_learning_and_game_theory/blob/main/docs/Notes_on_Reinforcement_Learning_and_Deep_Reinforcement_Learning.pdf)

# Appendix

## Key Concepts in Reinforcement Learning

What is Reinforcement Learning: branch of machine learning concerned with making decisions and taking sequences of actions based on some current state thereby maximizing some reward objective over time.

action

Environment

Agent

state, reward

Figure a1.f1: Feedback loop between the Agent and the Environment in RL

The main components of Reinforcement Learning are the *Agent* and the *Environment*. The Agent lives and interacts in the Environment. At every step (or moment in time) of the interaction the Agent sees a (possibly partial) observation of the state of the world and then it decides on an action to take. The environment changes when the agent acts on it but may also change on its own.

The agent also perceives a reward signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward - *the return*. Reinforcement learning teaches the agent patterns leading to a decision making achieving this goal.

**Definition** *State*

A state is a complete description of the state of the environment.

**Definition** *Observation*

An observation is a possibly partial description of a state , which may omit some of the information pertaining to the state .

**Definition** *Fully observed and partially observed environment*

Environment in which the agent can see the complete state of it. Otherwise, it is partially observed.

**Definition** *Action space*

*Action space* is the set of all valid actions in a given environment. The action space can be discrete or continuous. In discrete action spaces only a finite number of actions are available to the agent. In continuous action spaces the actions are real-valued vectors.

Note on continuous vs discrete state and action space models

The distinction between discrete and continuous state and action spaces have profound consequences for the methods of deep RL. Some families of algorithms can only be applied directly in one case and need substantial rework which would impose additional limitations on their performance in the other case. In this expose we will limit our attention to discrete state and action spaces governed by discrete Markov Decision Processes in order to make the presentation of the discussed algorithms as simple as possible.

**Definition** *Policy*

A *policy* is a rule used by an agent to decide on actions to take. It can be deterministic in which case it is denoted with :

or it may be stochastic in which case it is usually denoted by :

The policy defines the agent’s behavior, and it is not uncommon to substitute the notion of an “agent” with that of a “policy”.

**Definition** *Parametrized policy*

Policy as a computable function which depends on a set of parameters which are obtained through some sort of optimization procedure (e.g. neural network training). We will denote the parameters of such policy by or and then we will use subscript to highlight the connection

**Deterministic Policies**

**Example**: Deterministic Policy in continuous action space in PyTorch deep learning library torch.nn

pi\_net = nn.Sequential(

nn.Linear(obs\_dim, 64),

nn.Tanh(),

nn.Linear(64, 64),

nn.Tanh(),

nn.Linear(64, act\_dim)

)

This builds a MLP network with two hidden layers of size and tanh activation functions.

Here obs is a NumPy array which contains a batch of observations and pi\_net can be used to obtain a batch of actions as:

obs\_tensor = torch.as\_tensor(obs, dtype=torch.float32)

actions = pi\_net(obs\_tensor)

For details on the MLP network pi\_net constructed with the expression above see the section Example 2: Deterministic Policy in continuous action space modeled with nn.Sequential in the Appendix.

**Stochastic Policies**

The two most common types of stochastic policies are *categorical policies* and *diagonal Gaussian policies*.

//TODO: finish key concepts in RL (mainly from [17] and [18])

## Bellman’s Equations for Markov Decision Processes

//TODO: finish Bellman’s Equations for MDP (mainly from [23])

## Actor-Critic Reinforcement Learning Algorithms

**Definition of MDP**

MDP is a tuple where denotes the *state space*, the *action space* , is the *state transition probability density function* and the *reward function* which represents the reward of going from state to state taking action . Stationary MDP is such MDP in which all of the elements of the tuple are constants in time. Note that in case state space is discrete we will use state transition distribution instead of density function .

For the purpose of the current exposition we will assume that the state space is *continuous*. This assumption implies that it is only possible to define a probability of reaching a certain state region since the probability of reaching a particular state is 0.

The probability of reaching a state in the region from state after applying action is

(ac.mdp.1)

After each transition to a state , the controller receives an immediate reward

(ac.rwr.1)

which depends on the previous state, the current state and the action taken. The reward function is assumed to be bounded.

**Policy**

The action taken in a state is drawn from a stochastic policy . At any state the actor will choose an action by drawing it from a stochastic parametrized policy with parameters and the system transfers to a new state drawn from the state transition density or state transition distribution in the discrete case.

Policy is a map from state to action – a function which represents mapping from states to probabilities of selecting each possible action.

If the agent is following policy at time , then is the probability that if . Note that is an ordinary function which defines a probability distribution over for each .

The goal here is to find the policy which maximizes the expected value of a certain function of the immediate rewards (usually total return) received while following the policy. This expected value is known as the cost-to-go function

(ac.ctg.1)

In most cases, the function is either the discounted sum of rewards or the average reward received , as explained in the next two paragraphs.

**Discounted Reward**

In the discounted reward setting, the reward function is equal to the expected value of the discounted sum of rewards when starting with the initial state drawn from an initial state distribution , also called the discounted return:

(ac.dsc.1)

We denote with the the expected reward which is given with

(ac.expr.1)

Thus we get

(ac.cfun.1)

Here is the discounted state distribution under the policy and denotes the reward discount factor. Note that is a probability density function. Here we are not making an assumption of stationarity for ; that is does not necessarily exist.

During learning the agent will have to estimate the cost-to-go function for a given policy . The procedure is called *policy evaluation*. The resulting estimate of is called the *value function*. We distinguish two kinds of value function: the *state value function* and the *state-action value function*. The state value function is given with

(ac.svf.1)

Notice that depends only on the state and the policy is followed starting from that state.

Basically, is , parameterized with respect to which is renamed to . This can be expressed as

(ac.svf.2)

where

(ac.dsd.1)

is the discounted state distribution.

The state-action value function is given with

(ac.savf.1)

also depends on the state , but makes the action chosen in this state a free variable instead of having it generated by the policy .

Looking at (ac.cfun.1) we write

(ac.savf.2)

also depends on the state , but makes the action chosen in this state a free variable instead of having it generated by the policy . Once the first transition onto a next state has been made governs the rest of the action selection. The relationship between the state value function and the state-action value function is given with

(ac.svf.2)

We can derive the recursive form of the latter by manipulating further (ac.svf.1) and (ac.savf.1):

(ac.bel.1)

This equation is known as the Bellman’s equation. The derivation of (ac.bel.1) is shown below:

From (ac.svf.1) and (ac.cfun.1)

We rewrite (dsd.1) splitting it into two terms

(ac.dsd.2)

Substituting (ac.dsd.2) into (ac.svf.2) gives us

//TODO: finish the appendix section on actor-critic algorithms (mainly from [5] , [6] and [7])

## Policy Gradients

**Definition** *Trajectory*

We define a trajectory of length as

where starts from the starting distribution of states

//TODO: finish the appendix section on the policy gradients (mainly from [8] and [9])

## Generalized Advantage Estimation

//TODO: finish the appendix section on generalized advantage estimation (mainly from [10], [11])

## Proximal Policy Optimization

//TODO: finish the appendix section on the PPO (mainly from [15] and [16])

## A Note on nn.Sequential implemented with PyTorch Deep Learning library

nn.Sequential is a container class in PyTorch deep learning library that stacks neural network layers in a linear order. It allows you to build models by simply passing a sequence of modules to its constructor, where the output of each module is automatically passed as the input to the next one.

This class is *linear stacking*, that is - it is designed for networks where data flows through the layers in strictly sequential manner, with one input and one output for each layer. The entire nn.Sequential container acts as a single nn.Module allowing it to use it just like any other layer in the network.

For more details see [PyTorch docs on nn.Sequential](https://docs.pytorch.org/docs/stable/generated/torch.nn.Sequential.html).

### Example 1: MNIST-style digit classification (0-9) using nn.Sequential

import torch.nn as nn

import torch

# Define the sequential model

model = nn.Sequential(

nn.Flatten(), # Flattens the input image from 2D to 1D

nn.Linear(28\*28, 128), # A linear (fully connected) layer

nn.ReLU(), # A ReLU activation function (introduces non-linearity)

nn.Linear(128, 10), # Another linear layer to output final class scores

nn.Softmax(dim=1) # Softmax activation for probability distribution

)

# Example input (batch of 1, 28x28 image)

input\_image = torch.rand(1, 28, 28)

# Pass the input through the model

output = model(input\_image)

print(output)

Architecture Flow and Tensor Shape Transformations for the network on Example 1 shown below

A diagram of a software program

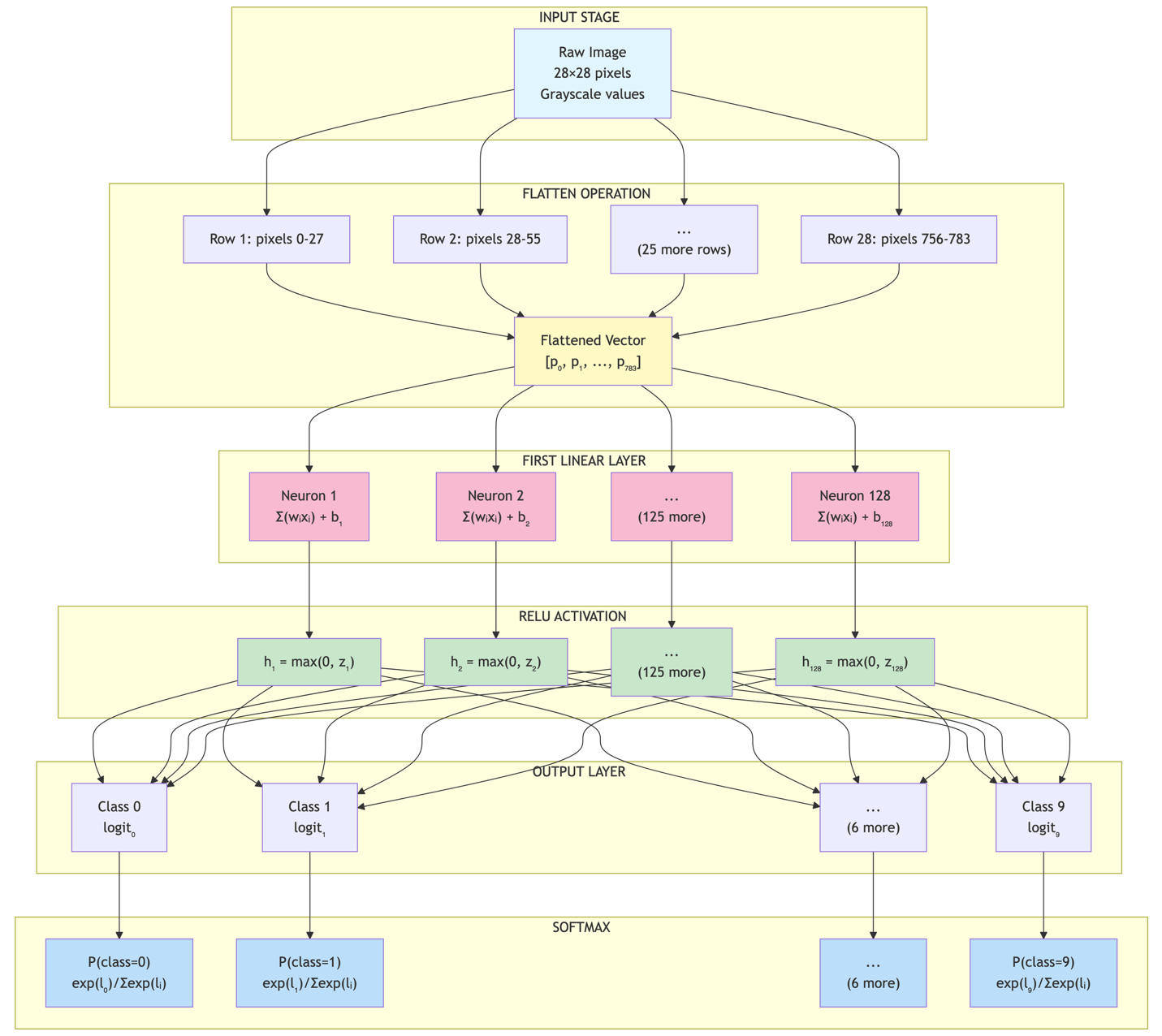
AI-generated content may be incorrect.A diagram of a diagram

AI-generated content may be incorrect.

Architecture Flow

Tensor Shape Transformations

Network layer stack and Layer operations for the network on Example 1 shown below:

A diagram of a computer

AI-generated content may be incorrect.

Information Flow with Dimensions for the network on Example 1 shown below:

A diagram of a function

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.Computation Graph for the network on Example 1 shown below:

Summary Table for the network on Example 1

A screenshot of a graph

AI-generated content may be incorrect.

Equations for the Example 1 Network

Layer 2 (First linear)

where ,

//TODO: finish the equations for the example network

For details see the markdown [pytorch\_example\_neural\_network\_diagrams.md](https://github.com/dimitarpg13/reinforcement_learning_in_agentic_workflows/blob/main/docs/pytorch/pytorch_example_neural_network_diagrams.md)

### Example 2: Deterministic Policy in continuous action space modeled with nn.Sequential

pi\_net = nn.Sequential(

nn.Linear(obs\_dim, 64),

nn.Tanh(),

nn.Linear(64, 64),

nn.Tanh(),

nn.Linear(64, act\_dim)

)

This builds a MLP network with two hidden layers of size and tanh activation functions.

Architecture Flow and Tensor Shape Transformations for the network on Example 2

A diagram of a layer

AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.

Tensor Shape Transformations

Architecture Flow

Network layer stack for the network on Example 2:

A screenshot of a computer

AI-generated content may be incorrect.

A diagram of a flowchart

AI-generated content may be incorrect.Detailed Network Architecture for Example 2:

//TODO: finish the diagrams for the deterministic policy in continuous action space

For details see the markdown [pytorch\_policy\_net\_diagrams.md](https://github.com/dimitarpg13/reinforcement_learning_in_agentic_workflows/blob/main/docs/pytorch/pytorch_policy_net_diagrams.md)