LDP (Language Decision Processes)

Language Decision Processes (LDP)

GITHUB ↗ pypi package 0.27.0 ↗ Lint and Test passing ↗ License Apache 2.0
python 3.11 | 3.12 | 3.13

An framework for constructing language model agents and training on constructive tasks.

This repo models agent-environment interactions using aPartially Observable Markov Decision Process ↗ (POMDP).
Inspired by POMDP, this repo's name `ldp` stands for Language Decision Processes.

Installation

To install `ldp`:

```
pip install -e .
```

If you plan to export Graphviz visualizations,
make sure you also install the `graphviz` library into your OS via:

- Linux: `apt install graphviz`
- macOS: `brew install graphviz`

Agent

An agent is something that interacts with an environment (defined in our other GitHub repo Future-House/aviary ↗).

An agent uses tools in response to observations, which are just natural language observations. An agent has two functions:

```
agent_state = await agent.init_state(tools=tools)
new_action, new_agent_state, value = await agent.get_asv(agent_state,
obs)
```

`get_asv(agent_state, obs)` chooses an action (`a`) conditioned on the observation messages,
and returns the next agent state (`s`) and a value estimate (`v`).
The first argument, `agent_state`, is a state specific for the agent.
The state is outside of the agent so agents are functional, enabling batching across environments.
You can make the state `None` if you aren't using it. It could contain things like memory, as a list of previous observations and actions.

The `obs` are not the complete list of all prior observations, but rather the return of `env.step`.
Usually the state should keep track of these.

Value is the agent's state-action value estimate; it can default to 0.
This is used for training with reinforcement learning.

Computing Actions

You can just emit actions directly if you want:

```
from aviary.core import ToolCall


def get_asv(agent_state, obs):
    action = ToolCall.from_name("calculator_tool", x="3 * 2")
    return action, agent_state, 0
```

but likely you want to do something more sophisticated.
Here's how our `SimpleAgent` - which just relies on a single LLM call - works (typing omitted):

```
from ldp.agent import Agent
from ldp.graph import LLMCallOp


class AgentState:
    def __init__(self, messages, tools):
        self.messages = messages
        self.tools = tools


class SimpleAgent(Agent):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.llm_call_op = LLMCallOp()

    async def init_state(self, tools):
        return AgentState([], tools)

    async def get_asv(self, agent_state, obs):
        action = await self.llm_call_op(
            config={"name": "gpt-4o", "temperature": 0.1},
            msgs=agent_state.messages + obs,
            tools=agent_state.tools,
        )
        new_state = AgentState(
            messages=agent_state.messages + obs + [action],
tools=agent_state.tools
        )
        return action, new_state, 0.0
```

Notice how it's pretty simple. We have to do some bookkeeping - namely appending messages as they come and passing tools. There is no magic here.

Compute Graph

We do have a compute graph - which helps if you want to differentiate with respect to parameters inside your agent (including possibly the LLM). If your compute graph looks like the above example - where all you do is call an LLM directly, then don't worry about this.

If you want to do more complex agents and train them, then read on. Let's start with an example compute graph

```
from ldp.graph import FxnOp, LLMCallOp, PromptOp, compute_graph

op_a = FxnOp(lambda x: 2 * x)

async with compute_graph():
    op_result = op_a(3)
```

This creates a compute graph and executes it. The compute graph is silly - just doubles the input. The compute graph executions and gradients are saved in a context for later use, like training updates. For example:

```
print(op_result.compute_grads())
```

Now, inside the `SimpleAgent` example above, you can see some of the compute graph. Let's see a more complex example for an agent that has a memory it can draw upon.

```
@compute_graph()
async def get_asv(self, agent_state, obs):
    # Update state with new observations
    next_state = agent_state.get_next_state(obs)

    # Retrieve relevant memories
    query = await self._query_factory_op(next_state.messages)
    memories = await self._memory_op(query, matches=self.num_memories)

    # Format memories and package messages
    formatted_memories = await
self._format_memory_op(self.memory_prompt, memories)
    memory_prompt = await self._prompt_op(memories=formatted_memories)
    packaged_messages = await self._package_op(
        next_state.messages, memory_prompt=memory_prompt,
use_memories=bool(memories)
    )

    # Make LLM call and update state
    config = await self._config_op()
    result = await self._llm_call_op(
        config, msgs=packaged_messages, tools=next_state.tools
    )
    next_state.messages.extend([result])

    return result, next_state, 0.0
```

You can see in this example that we use differentiable ops to ensure there is a connection in the compute graph from the LLM result (action) back to things like the memory retrieval and the query used to retrieve the memory.

Why use a compute graph? Aside from a gradient, using the compute graph enables the tracking of all inputs/outputs to the ops and serialization/deserialization of the compute graph so that you can easily save/load them. The tracking of input/outputs also makes it easier to do things like fine-tuning or reinforcement learning on the underlying LLMs.

Generic Support

The `Agent` (as well as classes in `agent.ops`)
are generics ↗,
which means:

- `Agent` is designed to support arbitrary types
- Subclasses can exactly specify state types, making the code more readable

If you are new to Python generics (`typing.Generic`),
please read about them in Python typing ↗.

Below is how to specify an agent with a custom state type.

```
from dataclasses import dataclass, field
from datetime import datetime

from ldp.agents import Agent


@dataclass
class MyComplexState:
    vector: list[float]
    timestamp: datetime = field(default_factory=datetime.now)


class MyAgent(Agent[MyComplexState]):
    """Some agent who is now type checked to match the custom state."""
```

Complete Example

```
from ldp.agent import SimpleAgent
from aviary.env import DummyEnv

env = DummyEnv()
agent = SimpleAgent()

obs, tools = await env.reset()
agent_state = await agent.init_state(tools=tools)

done = False
while not done:
    action, agent_state, _ = await agent.get_asv(agent_state, obs)
    obs, reward, done, truncated = await env.step(action.value)
```

Tutorial

See a tutorial of building and running an agent for GSM8K ↗

packages

Language Model Interface (LMI)

GITHUB ↗ pypi package 0.27.0 ↗ Lint and Test passing ↗ License Apache 2.0
python 3.11 | 3.12 | 3.13

A Python library for interacting with Large Language Models (LLMs) through an unified interface.

Installation

```
pip install fhlmi
```

- SentenceTransformerEmbeddingModel

Quick start

A simple example of how to use the library with default settings is shown below.

```
from lmi import LiteLLMModel
from aviary.core import Message

llm = LiteLLMModel()

messages = [Message(content="What is the meaning of life?")]

result = await llm.call_single(messages)
# assert result.text == "42"
```

Documentation

LLMs

An LLM is a class that inherits from `LLMModel` and implements the following methods:

- `async acompletion(messages: list[Message], **kwargs) -> list[LLMResult]`
- `async acompletion_iter(messages: list[Message], **kwargs) -> AsyncIterator[LLMResult]`

These methods are used by the base class `LLMModel` to implement the LLM interface. Because `LLMModel` is an abstract class, it doesn't depend on any specific LLM provider. All the connection with the provider is done in the subclasses using `acompletion` and `acompletion_iter` as interfaces.

Because these are the only methods that communicate with the chosen LLM provider, we use an abstraction LLMResult ↗ to hold the results of the LLM call.

LLMModel

An `LLMModel` implements `call`, which receives a list of `aviary` `Message`s and returns a list of `LLMResult`s. `LLMModel.call` can receive callbacks, tools, and output schemas to control its behavior, as better explained below.
Because we support interacting with the LLMs using `Message` objects, we can use the modalities available in `aviary`, which currently include text and images. `lmi` supports these modalities but does not support other modalities yet.
Adittionally, `LLMModel.call_single` can be used to return a single `LLMResult` completion.

LiteLLMModel

`LiteLLMModel` wraps `LiteLLM` API usage within our `LLMModel` interface. It receives a `name` parameter, which is the name of the model to use and a `config` parameter, which is a dictionary of configuration options for the model following the LiteLLM configuration schema ↗. Common parameters such as `temperature`, `max_token`, and `n` (the number of completions to return) can be passed as part of the `config` dictionary.

```
import os
from lmi import LiteLLMModel

config = {
    "model_list": [
        {
            "model_name": "gpt-4o",
            "litellm_params": {
                "model": "gpt-4o",
                "api_key": os.getenv("OPENAI_API_KEY"),
                "frequency_penalty": 1.5,
                "top_p": 0.9,
                "max_tokens": 512,
                "temperature": 0.1,
                "n": 5,
            },
        }
    ]
}

llm = LiteLLMModel(name="gpt-4o", config=config)
```

`config` can also be used to pass common parameters directly for the model.

```
from lmi import LiteLLMModel

config = {
    "name": "gpt-4o",
    "temperature": 0.1,
    "max_tokens": 512,
    "n": 5,
}

llm = LiteLLMModel(config=config)
```

Cost tracking

Cost tracking is supported in two different ways:

1. Calls to the LLM returns the token usage for each call in `LLMResult.prompt_count` and `LLMResult.completion_count`. Additionally, `LLMResult.cost` can be used to get a cost estimate for the call in USD.
2. A global cost tracker is maintained in `GLOBAL_COST_TRACKER` and can be enabled or disabled using `enable_cost_tracking()` and `cost_tracking_ctx()`.

Rate limiting

Rate limiting helps regulate the usage of resources to various services and LLMs. The rate limiter supports both in-memory and Redis-based storage for cross-process rate limiting. Currently, `lmi` take into account the tokens used (Tokens per Minute (TPM)) and the requests handled (Requests per Minute (RPM)).

Basic Usage

Rate limits can be configured in two ways:

1. Through the LLM configuration:

```
from lmi import LiteLLMModel

config = {
    "rate_limit": {
        "gpt-4": "100/minute",  # 100 tokens per minute
    },
    "request_limit": {
        "gpt-4": "5/minute",  # 5 requests per minute
    },
}

llm = LiteLLMModel(name="gpt-4", config=config)
```

With `rate_limit` we rate limit only token consumption,
and with `request_limit` we rate limit only request volume.
You can configure both of them or only one of them as you need.

2. Through the global rate limiter configuration:

```
from lmi.rate_limiter import GLOBAL_LIMITER

GLOBAL_LIMITER.rate_config[("client", "gpt-4")] = "100/minute"  # 100
tokens per minute
GLOBAL_LIMITER.rate_config[("client|request", "gpt-4")] = (
    "5/minute"  # 5 requests per minute
)
```

With `client` we rate limit only token consumption,
and with `client|request` we rate limit only request volume.
You can configure both of them or only one of them as you need.

Rate Limit Format

Rate limits can be specified in two formats:

1. As a string: `"<count> [per|/] [n (optional)] <second|minute|hour|day|month|year>"`

```
"100/minute"  # 100 tokens per minute

"5 per second"  # 5 tokens per second
"1000/day"  # 1000 tokens per day
```

2. Using RateLimitItem classes:

```
from limits import RateLimitItemPerSecond, RateLimitItemPerMinute

RateLimitItemPerSecond(30, 1)  # 30 tokens per second
RateLimitItemPerMinute(1000, 1)  # 1000 tokens per minute
```

Storage Options

The rate limiter supports two storage backends:

1. In-memory storage (default when Redis is not configured):

```
from lmi.rate_limiter import GlobalRateLimiter

limiter = GlobalRateLimiter(use_in_memory=True)
```

2. Redis storage (for cross-process rate limiting):

```
# Set REDIS_URL environment variable
import os

os.environ["REDIS_URL"] = "localhost:6379"

from lmi.rate_limiter import GlobalRateLimiter

limiter = GlobalRateLimiter()  # Will automatically use Redis if
REDIS_URL is set
```

This `limiter` can be used in within the `LLMModel.check_rate_limit` method to check the rate limit before making a request, similarly to how it is done in the `LiteLLMModel` class ↗.

Monitoring Rate Limits

You can monitor current rate limit status:

```
from lmi.rate_limiter import GLOBAL_LIMITER
from lmi import LiteLLMModel
from aviary.core import Message

config = {
    "rate_limit": {
        "gpt-4": "100/minute",  # 100 tokens per minute
    },
    "request_limit": {
        "gpt-4": "5/minute",  # 5 requests per minute
    },
}

llm = LiteLLMModel(name="gpt-4", config=config)
results = await llm.call([Message(content="Hello, world!")])  # Consume
some tokens

status = await GLOBAL_LIMITER.rate_limit_status()

# Example output:
{
    ("client|request", "gpt-4"): {  # the limit status for requests
        "period_start": 1234567890,
        "n_items_in_period": 1,
        "period_seconds": 60,
        "period_name": "minute",
        "period_cap": 5,
    },
    ("client", "gpt-4"): {  # the limit status for tokens
        "period_start": 1234567890,
        "n_items_in_period": 50,
        "period_seconds": 60,
        "period_name": "minute",
        "period_cap": 100,
    },
}
```

Timeout Configuration

The default timeout for rate limiting is 60 seconds, but can be configured:

```
import os

os.environ["RATE_LIMITER_TIMEOUT"] = "30"  # 30 seconds timeout
```

Weight-based Rate Limiting

Rate limits can account for different weights (e.g., token counts for LLM requests):

```
await GLOBAL_LIMITER.try_acquire(
    ("client", "gpt-4"),
    weight=token_count,  # Number of tokens in the request
    acquire_timeout=30.0,  # Optional timeout override
)
```

Tool calling

LMI supports function calling through tools, which are functions that the LLM can invoke. Tools are passed to `LLMModel.call` or `LLMModel.call_single` as a list of `Tool` objects from `aviary` ↗, along with an optional `tool_choice` parameter that controls how the LLM uses these tools.

The `tool_choice` parameter follows `OpenAI`'s definition. It can be:

Tool Choice Value	Constant	Behavior
`"none"`	`LLMModel.NO_TOOL_CHOICE`	The model will not call any tools and instead generates a message
`"auto"`	`LLMModel.MODEL_CHOOSES_TOOL`	The model can choose between generating a message or calling one or more tools
`"required"`	`LLMModel.TOOL_CHOICE_REQUIRED`	The model must call one or more tools
A specific `aviary.Tool` object	N/A	The model must call this specific tool

	`LLMModel.UNSPECIFIED_TOO`	No tool choice preference is

When tools are provided, the LLM's response will be wrapped in a `ToolRequestMessage` instead of a regular `Message`. The key differences are:

- `Message` represents a basic chat message with a role (system/user/assistant) and content
- `ToolRequestMessage` extends `Message` to include `tool_calls`, which contains a list of `ToolCall` objects, which contains the tools the LLM chose to invoke and their arguments

Further details about how to define a tool, use the `ToolRequestMessage` and the `ToolCall` objects can be found in the Aviary documentation ↗.

Here is a minimal example usage:

```
from lmi import LiteLLMModel
from aviary.core import Message, Tool
import operator


# Define a function that will be used as a tool
def calculator(operation: str, x: float, y: float) -> float:
    """
    Performs basic arithmetic operations on two numbers.

    Args:
        operation (str): The arithmetic operation to perform ("+", "-",
"*", or "/")
        x (float): The first number
        y (float): The second number

    Returns:
        float: The result of applying the operation to x and y

    Raises:
        KeyError: If operation is not one of "+", "-", "*", "/"
        ZeroDivisionError: If operation is "/" and y is 0
    """
    operations = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv,
    }
    return operations[operation](x, y)


# Create a tool from the calculator function
calculator_tool = Tool.from_function(calculator)

# The LLM must use the calculator tool
llm = LiteLLMModel()
result = await llm.call_single(
    messages=[Message(content="What is 2 + 2?")],
    tools=[calculator_tool],
    tool_choice=LiteLLMModel.TOOL_CHOICE_REQUIRED,
)

# result.messages[0] will be a ToolRequestMessage with tool_calls
containing
# the calculator invocation with x=2, y=2, operation="+"
```

Embedding models

This client also includes embedding models. An embedding model is a class that inherits from `EmbeddingModel` and implements the `embed_documents` method, which receives a list of strings and returns a list with a list of floats (the embeddings) for each string.

Currently, the following embedding models are supported:

- `LiteLLMEmbeddingModel`
- `SparseEmbeddingModel`
- `SentenceTransformerEmbeddingModel`
- `HybridEmbeddingModel`

LiteLLMEmbeddingModel

`LiteLLMEmbeddingModel` provides a wrapper around LiteLLM's embedding functionality. It supports various embedding models through the LiteLLM interface, with automatic dimension inference and token limit handling. It defaults to `text-embedding-3-small` and can be configured with a `name`, `batch_size`, and `config` parameters. Notice that `LiteLLMEmbeddingModel` can also be rate limited.

```
import os
from lmi import LiteLLMEmbeddingModel

model = LiteLLMEmbeddingModel(
    name="text-embedding-3-small",
    batch_size=16,
    config={
        "rate_limit": "100/minute",
    },
)

embeddings = await model.embed_documents(["text1", "text2", "text3"])
```

HybridEmbeddingModel

`HybridEmbeddingModel` combines multiple embedding models by concatenating their outputs. It is typically used to combine a dense embedding model (like `LiteLLMEmbeddingModel`) with a sparse embedding model for improved performance. The model can be created in two ways:

```
from lmi import LiteLLMEmbeddingModel, SparseEmbeddingModel,
HybridEmbeddingModel

dense_model = LiteLLMEmbeddingModel(name="text-embedding-3-small")
sparse_model = SparseEmbeddingModel()
hybrid_model = HybridEmbeddingModel(models=[dense_model, sparse_model])
```

The resulting embedding dimension will be the sum of the dimensions of all component models. For example, if you combine a 1536-dimensional dense embedding with a 256-dimensional sparse embedding, the final embedding will be 1792-dimensional.

SentenceTransformerEmbeddingModel

You can also use `sentence-transformer`, which is a local embedding library with support for HuggingFace models, by installing `lmi[local]`.

src

ldp

nn

chat_templates

- llama3.1_chat_template_vllm.jinja: https://github.com/vllm-project/vllm/blob/4fb8e329fd6f51d576bcf4b7e8907e0d83c4b5cf/examples/tool_chat_template_llama3.1_json.jinja
- llama3.1_chat_template_hf.jinja: https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct/blob/main/tokenizer_config.json#L2053
- llama3.1_chat_template_thought.jinja: Fixes a typo in llama3.1_chat_template_ori.jinja
- llama3.1_chat_template_nothought.jinja: Derived from llama3.1_chat_template_ori.jinja, but removes thoughts
- llama*ori.jinja: TODOC - these were written by @kwanUm