Modeling an Optimizable Processing Pipeline

August 11, 2025

Abstract

This document describes the modeling of a processing pipeline using the PipeOptz framework. The system is designed to allow users to construct workflows by combining processing steps (nodes) into a Directed Acyclic Graph (DAG). It provides a structure for creating, executing, and tuning these workflows, particularly for image processing.

Contents

1 Introduction

The primary goal of the PipeOptz framework is to provide a modular library for building and optimizing processing workflows. Users can define independent processing steps as **Nodes**, connect them to form a **Pipeline**, and then use the **PipelineOptimizer** to tune their parameters automatically. This architecture separates the logic of individual tasks from the overall workflow structure.

2 Core Concepts

The framework is built upon 4 components:

- **Node:** The basic building block. A node is a wrapper around a Python function that performs a single, atomic task. It has a unique ID and a set of fixed parameters.
- **Pipeline:** A container for a collection of nodes and their dependencies, forming a DAG. It manages the execution order and the flow of data between nodes.
- **Parameter:** An object that defines the search space for a tunable parameter within a node. Subclasses like `IntParameter` or `ChoiceParameter` allow the optimizer to know the valid range of values for a given parameter.
- **PipelineOptimizer:** The engine that runs metaheuristic algorithms (e.g., Genetic Algorithm, Bayesian Optimization) to find the optimal set of tunable parameters that minimize a given loss function.

3 The Pipeline: Defining Connections

`predecessors` dictionary in the `pipeline.add_node()` method.

3.1 Basic Connections

A standard connection maps the output of a source node (predecessor) to a named input parameter of the target node.

```python
# The output of 'node_A' becomes the 'image_input' for 'node_B'
pipeline.add_node(
    Node(id='node_B', func=my_func),
    predecessors={'image_input': 'node_A'}
)
```

Listing 1: Basic node connection

3.2 Runtime Inputs

To provide input to the pipeline when it starts, use the `run_params:` prefix. The value will be taken from the dictionary passed to the `pipeline.run()` method.

```
# 'image' input for 'blur_node' comes from runtime parameters
pipeline.add_node(
    Node(id='blur_node', func=gaussian_blur),
    predecessors={'image': 'run_params:source_image'}
)

# Execute the pipeline with the required runtime parameter
pipeline.run(run_params={'source_image': my_image_data})
```

Listing 2: Providing a runtime input

3.3 Advanced Structures and Connections

The pipeline also supports more complex workflow patterns like conditional logic, nested pipelines, and loops.

3.3.1 Conditional Execution with `NodeIf`

A `NodeIf` allows for branching logic. It contains a condition function and two sub-pipelines: one for `True` and one for `False`. Inputs for the condition function are specified with the `condition_func:` prefix.

```
# An 'if' node that checks an image's size
if_node = NodeIf(
    id='check_size',
    condition_func=is_large_image,
    true_pipeline=large_image_pipeline,
    false_pipeline=small_image_pipeline
)

# Wire the runtime image to the condition function's 'img' parameter
pipeline.add_node(if_node, predecessors={
    'condition_func:img': 'run_params:source_image'
})
```

Listing 3: Using NodeIf for conditional logic

3.3.2 Nested Pipelines (Sub-Pipelines)

To create modular and reusable workflows, an entire `Pipeline` object can be added as a node to a parent pipeline. This promotes hierarchical design.

```
# 'preprocessing_pipeline' is a complete Pipeline object
main_pipeline.add_node(
```

```
    preprocessing_pipeline,
    predecessors={'input_image': 'run_params:raw_image'}
)
```

Listing 4: Adding a pipeline as a sub-pipeline

3.3.3 Iteration (Loops)

To execute a node for each element in a list produced by a predecessor, wrap the target input parameter name in square brackets `[]`. Like that you don't need to create a map manually.

```
# 'isolate_objects' node outputs a list of image masks
pipeline.add_node(Node(id='isolate_objects', ...))

# 'process_mask' node will be executed for each mask in the list
pipeline.add_node(
    Node(id='process_mask', func=process_one_mask),
    predecessors={'[mask]': 'isolate_objects'}
)
```

Listing 5: Iterating over a list of elements

3.3.4 Combinatorial Execution (Product)

To execute a node over the Cartesian product of multiple input lists, wrap the target input parameter name in curly braces .

```
# 'get_kernels' outputs [[3,3], [5,5]]
# 'get_sigmas' outputs [1.0, 1.5]
pipeline.add_node(Node(id='get_kernels', ...))
pipeline.add_node(Node(id='get_sigmas', ...))

# 'apply_blur' will run 4 times with all combinations:
# (k=(3,3), sigma=1.0), (k=(3,3), sigma=1.5), etc.
pipeline.add_node(
    Node(id='apply_blur', func=gaussian_blur),
    predecessors={
        '{k}': 'get_kernels',
        '{sigma}': 'get_sigmas'
    }
)
```

Listing 6: Combinatorial execution

4 Serialization with JSON

To ensure persistence, interoperability, and reusability, pipelines can be serialized to and from a JSON format. The `pipeline.to_json()` and `Pipeline.from_json()` methods handle this process.

4.1 High-Level JSON Structure

The root of a serialized pipeline is a JSON object with four main keys:

- `name`: The string name of the pipeline.
- `description`: A string description.
- `nodes`: An array of node objects, ordered topologically.
- `edges`: An array of edge objects defining the connections.

4.2 The `nodes` Array

Each object in the `nodes` array represents a node or a complex structure.

4.2.1 Standard `Node`

A standard node is defined by its ID, its function (as a string), and its fixed parameters.

```
{
  "id": "blur_image",
  "type": "pipeoptz.utils.gaussian_blur",
  "fixed_params": { "k": 5, "sigma": 1.5 }
}
```

Listing 7: JSON for a standard Node

The `type` field is a string that `from_json` resolves to a callable Python function.

4.2.2 `NodeIf`

A conditional node has a special type and recursively contains its true and false sub-pipelines.

```
{
  "id": "conditional_branch",
  "type": "NodeIf",
  "condition_type": "my_module.my_condition_func",
  "fixed_params": {},
  "true_pipeline": {
    "name": "true_branch",
    "description": "",
    "nodes": [ ... ],
    "edges": [ ... ]
```

```
  },
  "false_pipeline": {
    "name": "false_branch",
    "description": "",
    "nodes": [ ... ],
    "edges": [ ... ]
  }
}
```

Listing 8: JSON for a NodeIf

4.2.3 Sub-Pipeline

A nested pipeline is also represented with a special type and a recursive definition.

```
{
  "id": "preprocessing_steps",
  "type": "SubPipeline",
  "pipeline": {
    "name": "preprocessing_pipeline",
    "description": "Applies blur and threshold.",
    "nodes": [ ... ],
    "edges": [ ... ]
  }
}
```

Listing 9: JSON for a Sub-Pipeline

4.3 The `edges` Array

The `edges` array defines the DAG structure. Each edge object is a simple, declarative link. The DSL syntax (e.g., brackets for loops) is stored in the `to_input` field.

```
{
  "from_node": "source_node_id",
  "to_node": "target_node_id",
  "to_input": "target_parameter_name"
}
```

Listing 10: JSON for an Edge

Example for an iterative connection:

```
{
  "from_node": "isolate_objects",
  "to_node": "process_mask",
  "to_input": "[mask]"
}
```

Listing 11: JSON for an Iterative Edge

4.4 Function Resolution

When loading a pipeline from JSON, the framework uses a resolver to convert the function path strings (e.g., `"pipeoptz.utils.gaussian_blur"`) back into actual Python functions. This is done via dynamic module importing, allowing the framework to reconstruct the exact pipeline if all the functions nodes can be accessible (by importating a library or by hard coding in the current file).