

---

# White Paper: A Dynamic Gray-Box Methodology for Analyzing and Simplifying PyTorch Models

---

**Jifeng Wu**  
jifengwu2k@gmail.com

## Abstract

Machine learning researchers and practitioners often face challenges when analyzing, interpreting, and repurposing deep learning models, particularly those built with popular, highly dynamic frameworks such as PyTorch. Existing static analysis and tracing techniques offer limited support for the breadth of constructs found in real-world codebases. This work-in-progress proposes a dynamic, gray-box methodology for systematically analyzing PyTorch models, extracting human-interpretable module hierarchies, and automatically synthesizing minimal, skeletal module classes that preserve functional equivalence and remain compatible with the original model weights. This document outlines a general strategy, open to further extensions, for approaching these challenges in a principled manner.

## 1 Introduction

Python’s inherently dynamic language features and PyTorch’s dynamic computation graph pose unique hurdles to programmatic model analysis and simplification. Existing static analysis tools, such as those used for model tracing (e.g., TorchScript [1]), struggle with dynamic control flow, custom model compositions, or complex dependencies.

To empower both researchers and practitioners with tools to better understand and repurpose existing PyTorch models, we outline herein a dynamic gray-box approach that enables both structural analysis and faithful simplification of arbitrary PyTorch models. This methodology is agnostic to the specific problem domain and prioritizes practical applicability over theoretical completeness.

## 2 Methodology Overview

This section provides an overview of the proposed analysis and simplification approach.

### 2.1 Project Acquaintance

The process begins with a survey of available project documentation (e.g., README files) to determine model invocation patterns and relevant entry points.

### 2.2 Model Snapshotting

The next phase consists of locating the moment in execution when the PyTorch model is fully constructed (initialized and loaded with trained weights), but before it is used for any inference or training. We temporarily disable any model tracing or scripting, and serialize (pickle) the model at this point for downstream analysis.

## 2.3 Unit Test Carving

At this point, the process performs unit test carving [2], which, in this context, means dynamically extracting and preserving self-contained test cases for each component module of the PyTorch model from a full model invocation.

For every unique, user-defined subclass of `torch.nn.Module` participating in the model, the methodology identifies at least one live instance. For this representative instance:

- Its full set of parameters and persistent buffers is saved by serializing its `state_dict`.
- The instance's `forward` method is wrapped with logic (see Listing 1) that, on first invocation:
  - Records the inputs (`*args, **kwargs`) as presented to `forward`, and the output (the return value of `forward`).
  - Serializes the inputs and outputs.

```
import pickle
import torch
from collections import OrderedDict
from inspect import signature
from types import MethodType

class ForwardWrapper(object):
    def __init__(self, module_name, forward):
        self.module_name = module_name
        if not isinstance(forward, MethodType):
            raise TypeError('not isinstance(forward, MethodType)')
        self.forward = forward
        self.parameters = list(signature(forward).parameters.keys())
        self.instance = forward.__self__
        self.inputs_output_recorded = False

    def __call__(self, *args, **kwargs):
        if not self.inputs_output_recorded:
            inputs_outputs = OrderedDict()
            for (parameter, arg) in zip(self.parameters, args):
                inputs_outputs[parameter] = pickle.loads(
                    pickle.dumps(arg)
                )
            for (parameter, arg) in kwargs.items():
                inputs_outputs[parameter] = pickle.loads(
                    pickle.dumps(arg)
                )
            result = self.forward(*args, **kwargs)
            inputs_outputs['return'] = pickle.loads(
                pickle.dumps(result)
            )
            torch.save(inputs_outputs, '%s.pt' % (self.module_name,))
            self.inputs_output_recorded = True
            return result
        else:
            return self.forward(*args, **kwargs)
```

Listing 1: The `ForwardWrapper` class for capturing inputs and outputs

After the rest of the script finishes execution, this process, for each user-defined model class, extracts a minimal unit test comprising model state, inputs, and outputs.

## 2.4 Building the Module Hierarchy Trie

Next, the pickled model is loaded and recursively traversed to build a module hierarchy trie (using the `tinytrie` package) describing the entire module hierarchy (see Listing 2). Each node in the trie corresponds to a unique instance of a `torch.nn.Module` (including standard PyTorch modules and user-defined subclasses).

```

import torch
from tinytrie import TrieNode

def build_module_hierarchy_trie(module):
    root = TrieNode()
    root.is_end = True
    root.value = module
    for child_module_name, child_module in module.named_children():
        child_root = build_module_hierarchy_trie(child_module)
        root.children[child_module_name] = child_root
    return root

```

Listing 2: Building a module hierarchy trie

In the context of a module hierarchy trie, we define node height as the length of the longest path from that node down to a leaf node (i.e., a submodule with no further children). Analyzing node heights (see Listing 3) provides a systematic way to stratify the model’s structure, facilitating bottom-up analysis and simplification.

```

from typing import Dict

import torch
from tinytrie import TrieNode

def analyze_node_heights(module_hierarchy_trie_root):
    node_heights = {}

    def analyze_node_height(node):
        if node.children:
            children_heights = []
            for child_node in node.children.values():
                analyze_node_height(child_node)
                children_heights.append(node_heights[child_node])
            node_heights[node] = max(children_heights) + 1
        else:
            node_heights[node] = 0

    analyze_node_height(module_hierarchy_trie_root)
    return node_heights

```

Listing 3: Analyzing node heights in the module hierarchy trie

## 2.5 Extracting Invariants

For each user-defined `torch.nn.Module` subclass, the methodology iterates over all its instances and records the following invariants, assembling them into a module signature:

- Node heights
- Child module names and types
- Parameters (names, `requires_grad` properties, and values)
- Non-persistent buffer names and values
- Persistent buffer names and shapes
- forward method implementations

A module signature thus captures the essential blueprint for reconstructing any conformant instance of a given module class.

## 2.6 Concolic Analysis of forward Methods

To determine precisely how each user-defined module generates its outputs from inputs, the methodology employs concolic analysis [3] for the forward method of each user-defined module class, starting with those with node heights of 0 and proceeding bottom-up. The core steps are:

- (a) Execute the forward method concretely on the carved unit test inputs while instrumenting the code to record control flow branches, data dependencies, and external variable accesses.
- (b) Where feasible, lift observed computations into a symbolic domain-specific language (DSL) representing tensor operations.
- (c) For module attributes accessed by forward, check if their values are invariant across all module instances; if not, these are promoted to arguments in the module’s constructor.
- (d) Analyze called auxiliary methods; inline them if possible.

This ensures accurate extraction of computational behavior under Python’s dynamic features.

## 2.7 Synthesis and Testing of Minimal Module Classes

Concurrently with forward method analyses, the methodology synthesizes minimal class definitions of every user-defined `torch.nn.Module` subclass in the model:

- A bespoke constructor (`__init__`) only initializes child modules, parameters, buffers, and attributes proven to be necessary.
- The forward method is programmatically generated from the extracted, simplified DSL logic, guaranteeing functional alignment with the original.

Correctness is validated via previously carved unit tests: for each synthesized class definition,

- An instance is created.
- The previously saved `state_dict` is loaded.
- The previously captured input is fed to the forward method.
- The output is compared to the previously captured output, confirming functional equivalence.

## 3 Current Status

The methodology described here, especially the concolic analysis of forward methods, is largely work in progress. An implementation is under exploratory development, and all steps are intended to be generalizable to arbitrary PyTorch models.

## 4 Conclusion

This approach delineates a principled, dynamic route to both analyzing and simplifying complex PyTorch models, producing minimal but fully functional model definitions. By relying on unit test carving, invariant extraction, concolic analysis, and unit testing rather than static analysis, it accommodates the dynamic and evolving idioms of real-world model code, promising useful applications in model understanding, refactoring, and hardware- or deployment-oriented optimization.

## References

- [1] Zachary DeVito. TorchScript: Optimized execution of PyTorch programs. *Retrieved January, 2022*.
- [2] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering*, pages 253–264, 2006.
- [3] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on automated software engineering*, pages 571–572, 2007.