ACCELERATED IMPLEMENTATION PLAN

Mechanistic Interpretability for Physics-Informed Neural Networks

A Practical, Executable Guide for Completion in 4-8 Weeks

TOTAL DURATION
Minimum Viable: 4 Weeks | Full Scope: 8 Weeks
Recommended: 6 Weeks (Best balance of depth and feasibility)

Executive Summary

Main Objective

Apply mechanistic interpretability techniques (probing classifiers, activation patching, circuit discovery) to Physics-Informed Neural Networks to reverse-engineer the computational algorithms they learn for solving differential equations. The goal is to identify interpretable circuits corresponding to numerical methods and provide mechanistic explanations for known PINN failure modes such as spectral bias.

Success Metrics

- Probing Accuracy: R-squared greater than 0.85 for derivative extraction from intermediate layers
- Circuit Identification: At least one interpretable circuit mapped to a known numerical operation
- Spectral Bias Explanation: Mechanistic hypothesis with supporting evidence from interventions
- Code Quality: Greater than 80% test coverage, documented API, reproducible results

Final Deliverables

1. GitHub repository with PINN training and interpretability toolkit
2. Technical report with mechanistic findings for 2-4 PDEs
3. Publication-ready paper draft (NeurIPS/ICML format)
4. Tutorial notebooks demonstrating interpretability analyses

Claude Code Integration Strategy

Claude Code is Anthropic's command-line tool for agentic coding that enables Claude to work directly in your development environment. This section outlines the optimal strategy for leveraging Claude Code throughout the project.

Recommended Usage Pattern

Use Claude Code at the **task level** rather than daily or weekly granularity. Each discrete task (implementing a module, debugging an issue, writing tests) should be a separate Claude Code session. This approach provides: (1) clear context boundaries, (2) focused problem-solving, (3) easier debugging if something goes wrong, and (4) natural commit points.

Session Structure for Each Task

1. Context Initialization: Begin each Claude Code session by providing relevant context:

```
claude "Review the project structure in src/ and tests/. I need to
implement [specific task]. The relevant existing files are [list
files]. Here is the technical specification: [brief spec]"
```

2. Iterative Development: Break complex tasks into sub-tasks:

```
claude "First, create the basic class structure for ProbingClassifier
with __init__ and fit methods. Don't implement the full logic yet,
just the interface."
```

3. Verification and Testing: Always include verification:

```
claude "Run the tests for the probing module and fix any failures.
Then add a test case for edge case [specific edge case]."
```

Best Practices for Claude Code Sessions

- Keep sessions focused: One logical unit of work per session (e.g., 'implement activation patching' not 'build entire interpretability toolkit')
- Provide specifications: Give Claude Code clear interfaces, expected inputs/outputs, and error handling requirements
- Request tests alongside implementation: Always ask for unit tests with each new module
- Review before committing: Examine generated code before git commits to maintain code quality
- Use CLAUDE.md: Maintain a CLAUDE.md file in your repository root with project-specific conventions

Example CLAUDE.md Configuration

```
# Project: MechInterp-PINNs  ## Code Style - Use type hints for all function
signatures - Docstrings in NumPy format - Maximum line length: 100 characters
- Use pathlib for file paths  ## Testing - pytest for all tests - Minimum 80%
coverage for new code - Place tests in tests/ mirroring src/ structure  ##
```

```
Git Commits - Conventional commits: feat:, fix:, docs:, test:, refactor: -
Reference issue numbers when applicable  ## Key Abstractions - PINN models
inherit from BasePINN in src/models/base.py - Interpretability tools inherit
from BaseProbe in src/interpretability/base.py - All experiments use configs
from configs/ directory
```

Task-by-Task Claude Code Usage Guide

Throughout this implementation plan, each day's tasks include specific Claude Code commands. The pattern is:

Task Type	**Claude Code Approach**
New Module	Single session: interface definition, implementation, tests
Bug Fix	Session with error logs, failing test, expected behavior
Analysis Script	Notebook-style session with visualization requirements
Refactoring	Session with target architecture, ensure tests pass throughout

WEEK 1: Foundations and Rapid Prototype

Weekly Objective: Establish development infrastructure and train a working PINN on the Poisson equation with activation extraction capabilities.

Day 1: Environment Setup and Repository Structure

Estimated hours: 4-6h

Tasks

1. [ ] Create GitHub repository with professional structure

Commands:

```
mkdir mechinterp-pinns && cd mechinterp-pinns git init mkdir -p
src/{models,interpretability,utils} tests/{models,interpretability} configs
notebooks data/{raw,processed} outputs/{figures,models,activations} touch
src/__init__.py src/models/__init__.py src/interpretability/__init__.py
src/utils/__init__.py touch README.md CLAUDE.md requirements.txt
setup.py .gitignore
```

2. [ ] Configure Python environment with critical dependencies

requirements.txt content:

```
torch>=2.0.0 numpy>=1.24.0 scipy>=1.10.0 matplotlib>=3.7.0 h5py>=3.8.0
wandb>=0.15.0 pytest>=7.3.0 pytest-cov>=4.0.0 black>=23.0.0 isort>=5.12.0
mypy>=1.0.0 tqdm>=4.65.0
```

Verification:

```
python -c "import torch; print(f'PyTorch {torch.__version__}, CUDA:
{torch.cuda.is_available()}')"
```

3. [ ] Create CLAUDE.md configuration file

See Claude Code Integration section above for template content

Claude Code Session: Initial Setup

```
claude "Initialize the mechinterp-pinns project with: 1. A BasePINN abstract
class in src/models/base.py with forward(), compute_pde_residual(), and
train() methods 2. Type hints for all parameters 3. Docstrings explaining the
PINN approach 4. A basic test in tests/models/test_base.py verifying the
interface"
```

Day 1 Checkpoint

- [GitHub repository created with proper structure
- [Virtual environment working (import torch succeeds)
- [BasePINN class skeleton implemented
- [pytest runs successfully (even if tests are placeholder)

COMMIT TO GITHUB: `feat: initial project structure and BasePINN interface`

Day 2: PINN Architecture Implementation

Estimated hours: 5-7h

Tasks

1. [ ] Implement MLP architecture with configurable layers

```
class MLP(BasePINN):     def __init__(self, input_dim: int, hidden_dims:
list[int], output_dim: int, activation: str = "tanh"):         # Initialize
layers with nn.Sequential        # Support activation options: tanh, relu,
gelu, sin         pass          def forward(self, x: torch.Tensor) ->
torch.Tensor:         # Forward pass with activation extraction hooks
pass
```

2. [ ] Implement automatic differentiation utilities for PDE residuals

```
def compute_derivatives(u: torch.Tensor, x: torch.Tensor, order: int = 1) ->
torch.Tensor:     """Compute spatial derivatives using autograd.     Args:
u: Network output, shape (N, 1)         x: Input coordinates, shape (N, d)
where d is spatial dimension         order: Derivative order (1 for gradient,
2 for Laplacian)     Returns:         Derivatives tensor     """     pass
```

3. [ ] Create configuration system using dataclasses

Claude Code Session: MLP Implementation

```
claude "Implement the MLP PINN in src/models/mlp.py with: 1. Configurable
architecture (hidden_dims as list) 2. Multiple activation function support 3.
Forward hooks that store activations in a dictionary self.activations 4.
Method get_activations() returning dict of layer_name -> activation tensor 5.
Unit tests verifying shapes and gradient flow Include docstrings and type
hints throughout."
```

Day 2 Checkpoint

- [MLP forward pass produces correct output shapes
- [Activations stored for each layer during forward pass
- [compute_derivatives correctly computes du/dx for test function
- [All tests pass with pytest

COMMIT TO GITHUB: `feat: MLP architecture with activation extraction`

Day 3: Poisson Equation Implementation

Estimated hours: 5-6h

Tasks

1. [ ] Implement Poisson equation problem class

Problem: Laplacian(u) = f on [0,1]^2 with Dirichlet BC

Manufactured solution: u(x,y) = sin(pi*x)*sin(pi*y), f(x,y) = -2*pi^2*sin(pi*x)*sin(pi*y)

2. [ ] Implement training loop with loss decomposition

```
def train_pinn(model, problem, config):     # L = w_pde * L_pde + w_bc * L_bc
# Log: total_loss, pde_loss, bc_loss, relative_l2_error     # Use W&B for
experiment tracking     pass
```

3. [ ] Create collocation point sampling strategy

Interior points: 10,000 (Latin Hypercube or uniform random). Boundary points: 1,000 per edge

Claude Code Session: Poisson Problem

```
claude "Create src/problems/poisson.py with: 1. PoissonProblem class with
domain, boundary_condition, source_term, analytical_solution 2. Method
sample_collocation_points(n_interior, n_boundary) using Latin Hypercube 3.
Method compute_relative_l2_error(model) comparing to analytical solution 4.
Tests verifying collocation point shapes and boundary coverage"
```

Day 3 Checkpoint

- [PoissonProblem generates valid collocation points
- [PDE residual computation is correct (verify with analytical solution)
- [Boundary points lie exactly on domain edges

COMMIT TO GITHUB: `feat: Poisson equation problem implementation`

Day 4: Training Pipeline and Validation

Estimated hours: 6-8h

Tasks

1. [ ] Implement full training loop with W&B logging

2. [ ] Train Poisson PINN and achieve less than 1% relative L2 error

3. [ ] Save trained model and generate solution visualizations

Target configuration: 4 hidden layers, 64 neurons each, tanh activation, Adam optimizer with lr=1e-3, 20,000 iterations

Claude Code Session: Training Pipeline

```
claude "Create src/training/trainer.py with: 1. PINNTrainer class handling
training loop, loss computation, logging 2. W&B integration for tracking
losses and metrics 3. Checkpoint saving every N iterations 4. Early stopping
based on validation error 5. Method to generate solution heatmap
visualization Run a training test on Poisson and verify convergence."
```

Day 4 Checkpoint

- [Training completes without errors
- [Relative L2 error below 1% (target: 0.5%)
- [W&B dashboard shows loss curves

[Solution visualization matches analytical solution

COMMIT TO GITHUB: `feat: training pipeline with W&B integration`

Day 5: Activation Extraction and Storage

Estimated hours: 5-6h

Tasks

1. [ ] Implement systematic activation extraction on dense grid

Grid: 100x100 points covering [0,1]^2 = 10,000 evaluation points

2. [ ] Create HDF5 storage for efficient activation access

```
Structure: /coordinates (N,2), /layer_0 (N, hidden_dim), /layer_1 (N,
hidden_dim), ...
```

3. [ ] Create visualization utilities for activation patterns

Claude Code Session: Activation Storage

```
claude "Create src/interpretability/activation_store.py with: 1.
ActivationStore class that extracts and saves activations to HDF5 2. Method
extract_on_grid(model, grid_resolution) -> stores to file 3. Method
load_layer(layer_name) -> numpy array 4. Method visualize_neuron(layer,
neuron_idx) -> 2D heatmap Verify storage works with trained Poisson model."
```

Day 5 Checkpoint

[Activations extracted for all layers
[HDF5 file created with correct structure
[Neuron activation heatmaps render correctly

COMMIT TO GITHUB: `feat: activation extraction and HDF5 storage`

Days 6-7: Documentation and Week 1 Review

Estimated hours: 6-8h total

Tasks

1. [ ] Write comprehensive README.md with installation and quickstart

2. [ ] Create tutorial notebook: 01_train_poisson_pinn.ipynb

3. [ ] Run pytest with coverage report, ensure greater than 70%

4. [ ] Review and clean up code with black and isort

Week 1 Validation Checkpoint

Deliverable	Target	Status

GitHub repository with clean structure	Complete	[ ]
Poisson PINN with <1% L2 error	<0.5%	[ ]
Activation extraction pipeline	Working	[ ]
Test coverage	>70%	[ ]

Plan B if Behind Schedule

If training convergence issues: Use pre-trained weights from DeepXDE library. If activation extraction issues: Simplify to storing only final hidden layer. If time short: Skip tutorial notebook, document in code comments only.

COMMIT TO GITHUB: `docs: Week 1 complete - README, tutorial notebook, tests`

WEEK 2: Probing Classifiers

Weekly Objective: Implement probing classifier framework and establish baselines for derivative detection at each network layer.

Day 8: Probing Framework Architecture

Estimated hours: 5-6h

Tasks

1. [ ] Implement LinearProbe class for single-target prediction

```
class LinearProbe:     def __init__(self, input_dim: int, output_dim: int =
1):         self.weights = nn.Linear(input_dim, output_dim)          def
fit(self, activations: Tensor, targets: Tensor, epochs: int = 1000):
# Train with MSE loss, Adam optimizer         pass          def predict(self,
activations: Tensor) -> Tensor:         pass          def score(self,
activations: Tensor, targets: Tensor) -> dict:         # Return: mse,
r_squared, explained_variance         pass
```

2. [ ] Implement ground-truth derivative computation

Use analytical derivatives for Poisson: du/dx = pi*cos(pi*x)*sin(pi*y), du/dy = pi*sin(pi*x)*cos(pi*y), Laplacian(u) = -2*pi^2*sin(pi*x)*sin(pi*y)

Claude Code Session: Probing Framework

```
claude "Create src/interpretability/probing.py with: 1. LinearProbe class as
specified above 2. DerivativeProber class that trains probes for du/dx,
du/dy, d2u/dx2, d2u/dy2, laplacian 3. Method probe_all_layers(model,
activation_store) -> DataFrame of scores 4. Comprehensive tests verifying
probe training on synthetic data"
```

Day 8 Checkpoint

- [LinearProbe trains successfully on synthetic data
- [Ground-truth derivatives computed for all grid points

COMMIT TO GITHUB: `feat: probing classifier framework`

Days 9-10: Layer-wise Derivative Probing

Estimated hours: 8-10h total

Tasks

1. [ ] Train probes for first derivatives (du/dx, du/dy) at each layer

2. [ ] Train probes for second derivatives and Laplacian

3. [ ] Generate layer-by-layer accuracy plots

4. [ ] Analyze where derivative information emerges

Expected Results Pattern

Hypothesis: R-squared should increase through layers as derivative information is computed. First derivatives may emerge in early layers (finite-difference-like computation), while Laplacian emerges later (composition of second derivatives).

Claude Code Session: Probing Experiments

```
claude "Create notebooks/02_probing_analysis.ipynb that: 1. Loads trained
Poisson PINN and activations 2. Trains probes for all derivative targets at
all layers 3. Creates bar plot: x=layer, y=R-squared, grouped by derivative
type 4. Creates heatmap: layers vs derivative targets, color=R-squared 5.
Identifies the layer where each derivative becomes linearly accessible (R2 >
0.8)"
```

Days 9-10 Checkpoint

- [Probes trained for all 5 derivative targets across all layers
- [At least one layer achieves R-squared > 0.85 for Laplacian
- [Visualization clearly shows information emergence pattern

COMMIT TO GITHUB: `feat: layer-wise derivative probing analysis`

Days 11-12: Probe Weight Analysis

Estimated hours: 8-10h total

Tasks

1. [ ] Extract and visualize probe weights for first-layer probes

2. [ ] Analyze whether weights show finite-difference-like patterns

3. [ ] Compare with known stencil patterns (central difference coefficients)

4. [ ] Document initial hypothesis about learned algorithm

Analysis Focus

For du/dx probe weights on first-layer activations: visualize weight magnitude by neuron. Look for neurons with high positive vs. negative weights (indicating difference operation). For second derivative probes: look for weights proportional to [1, -2, 1] pattern.

Days 11-12 Checkpoint

- [Probe weight visualizations generated
- [Written analysis of weight patterns (1-2 paragraphs)
- [Preliminary hypothesis documented

COMMIT TO GITHUB: `analysis: probe weight visualization and initial findings`

Days 13-14: Week 2 Consolidation

Estimated hours: 6-8h total

Tasks

1. [ ] Write technical summary of probing findings (1-2 pages)
2. [ ] Extend probing to additional test cases (different architectures)
3. [ ] Run code quality checks, increase test coverage to >80%
4. [ ] Prepare figures for eventual paper

Week 2 Validation Checkpoint

Deliverable	Target	Status
Probing framework complete	Working	[ ]
Laplacian probe R-squared	>0.85	[ ]
Layer-wise analysis notebook	Complete	[ ]
Initial mechanistic hypothesis	Documented	[ ]

Plan B if Behind Schedule

If probing accuracy is low: Try nonlinear probes (2-layer MLP). If computational constraints: Reduce grid resolution to 50x50. If specific derivatives fail: Focus on Laplacian only (most important for PDE residual).

COMMIT TO GITHUB: `docs: Week 2 complete - probing analysis and technical summary`

WEEK 3: Activation Patching

Weekly Objective: Implement activation patching framework to identify causal circuits and map receptive fields.

Days 15-17: Patching Framework Implementation

Tasks

1. [ ] Implement ActivationPatcher class with spatial awareness

```
class ActivationPatcher:     def patch_and_measure(self, model, x_target,
x_source, layer_name):         # Replace activations at x_target with those
from x_source         # Return: original_output, patched_output, delta
pass         def compute_causal_influence(self, model, target_point,
source_grid):         # For each source point, compute influence on target
output        # Return: influence_map (2D array matching source_grid)
pass
```

2. [ ] Create efficient batched implementation
3. [ ] Generate causal influence maps for sample points

Claude Code Session: Activation Patching

```
claude "Create src/interpretability/patching.py with: 1. ActivationPatcher
class as specified 2. Batched implementation that can process multiple source
points simultaneously 3. Method to estimate effective receptive field radius
at each layer 4. Visualization methods for influence maps 5. Tests verifying
patching doesn't break when source=target (identity)"
```

Days 18-21: Receptive Field Analysis

Tasks

1. [ ] Compute receptive field radius at each layer
2. [ ] Compare empirical receptive fields to theoretical finite difference stencils
3. [ ] Analyze boundary vs. interior point differences
4. [ ] Document findings and prepare notebook

Week 3 Validation Checkpoint

Deliverable	Target	Status
Patching framework complete	Working	[ ]
Receptive field analysis notebook	Complete	[ ]
Evidence for locality of algorithms	Documented	[ ]

WEEK 4: Circuit Discovery and Minimum Viable Results

Weekly Objective: Identify minimal circuits for derivative computation and produce a complete technical report with publication-quality findings. This marks the end of the minimum viable project.

Days 22-25: Circuit Discovery Implementation

Tasks

1. [ ] Implement ACDC-style iterative edge pruning
2. [ ] Define behavior preservation metrics (derivative accuracy, BC satisfaction)
3. [ ] Discover minimal circuits for Laplacian computation
4. [ ] Visualize circuit structure as computational graph

Days 26-28: Technical Report and Visualization

Tasks

1. [ ] Create publication-quality figures summarizing findings
2. [ ] Write technical report (5-8 pages) covering all experiments
3. [ ] Ensure all code is documented and reproducible
4. [ ] Create reproducibility checklist and data artifacts

MINIMUM VIABLE PROJECT COMPLETE

At end of Week 4, you have a publishable contribution: first application of mechanistic interpretability to PINNs with derivative probing, activation patching, and circuit analysis for Poisson equation.

WEEKS 5-8: Extended Scope (If Time Permits)

The following weeks extend the project to additional PDEs and deeper analysis. Execute these if Week 4 deliverables are complete and time remains.

Week 5: Heat Equation and Temporal Analysis

- Implement Heat equation PINN with time-dependent solution
- Apply probing and patching to analyze temporal integration
- Compare spatial vs. temporal derivative circuits

Week 6: Spectral Bias Investigation

- Implement Helmholtz equation with varying wavenumber
- Document spectral bias phenomenon mechanistically
- Compare standard MLP with Modified Fourier Network

Week 7: Intervention Experiments

- Design and execute targeted interventions based on hypotheses
- Validate mechanistic explanations through ablation
- Derive and test architectural recommendations

Week 8: Paper Writing and Release

- Draft full paper for NeurIPS/ICML submission
- Prepare GitHub release with comprehensive documentation
- Create demo notebook and video walkthrough

Git Workflow and Commit Strategy

Branch Structure

```
main           # Production-ready code only develop        # Integration
branch for features feature/*      # Feature branches (e.g., feature/probing-
framework) experiment/*   # Experimental branches for risky changes
```

Commit Convention

Use Conventional Commits format:

```
feat:     New feature (e.g., "feat: add probing classifier framework") fix:
Bug fix (e.g., "fix: correct derivative computation for boundary points")
docs:     Documentation (e.g., "docs: add API reference for patching module")
test:     Tests (e.g., "test: add integration tests for PINN training")
refactor: Code refactoring (e.g., "refactor: extract activation storage to
separate module") analysis: Research findings (e.g., "analysis: layer-wise
derivative emergence results")
```

Daily Commit Rhythm

- Morning: Pull latest changes, review yesterday's work
- During work: Commit after each logical unit (function, test, fix)
- End of day: Push all commits, update progress log

Appendix: Contingency Plans

Scenario 1: PINN Training Does Not Converge

Symptoms: Loss stagnates, relative L2 error remains above 10%. Actions: (1) Reduce learning rate to 1e-4, (2) Increase network depth to 6 layers, (3) Use learning rate scheduling (cosine annealing), (4) Fall back to DeepXDE library implementation.

Scenario 2: Probing Accuracy Is Low (R-squared < 0.5)

Symptoms: Linear probes fail to extract derivative information. Actions: (1) Try nonlinear probes (2-layer MLP with 64 hidden units), (2) Increase number of probe training samples, (3) Check for data leakage or preprocessing errors, (4) Consider that derivatives may not be linearly represented (still a publishable finding).

Scenario 3: Activation Patching Shows No Locality

Symptoms: All source points equally affect all target points. Actions: (1) Verify patching implementation correctness, (2) Try patching at different layers, (3) Consider that global computation may be the learned algorithm (different from finite differences), (4) Document as evidence against local algorithm hypothesis.

Scenario 4: Behind Schedule by More Than 3 Days

Actions: (1) Focus exclusively on Poisson equation (skip Heat, Burgers, Helmholtz), (2) Prioritize probing over patching and circuit discovery, (3) Aim for minimum viable deliverable: probing analysis on single PDE with clear findings, (4) The 4-week core is still publishable.

Scenario 5: Negative Results Throughout

Symptoms: No interpretable circuits found, probing fails, patching reveals nothing. Actions: This is still a valuable contribution. Document thoroughly what was tried and what didn't work. Frame as 'challenges in applying interpretability to scientific ML' or 'evidence that PINNs learn qualitatively different algorithms from classical methods.' Negative results papers are published at top venues.