A Comparative Analysis of Bilevel Optimization Solvers for Meta-Learning in Few-Shot Image Classification

Adithi Samudrala 2022102065
IIIT-Hyderabad

github

November 19, 2025

Abstract

Meta-learning, or learning to learn, has emerged as a powerful paradigm for developing agents that can rapidly adapt to new tasks from limited data. A prominent approach, Model-Agnostic Meta-Learning (MAML), frames this problem as a bilevel optimization task. The method used to solve this bilevel problem and compute the meta-gradient is critical to the algorithm's performance, stability, and computational cost. This report provides a comprehensive empirical analysis of various bilevel optimization solvers on the MAML algorithm. We implement and compare a wide range of methods, including first-order (FOMAML, Reptile), approximate second-order (Neumann Series), implicit second-order (iMAML with CG), and exact second-order (Explicit MAML) solvers. Furthermore, we evaluate an advanced penalty-based method and contrast all meta-learning approaches against standard transfer learning baselines. We evaluate these methods on the Omniglot dataset for 5-way, 1-shot image classification, analyzing their final accuracy, learning dynamics, and computational trade-offs. Our findings reveal a clear hierarchy of performance, demonstrate the pitfalls of traditional fine-tuning in extreme few-shot scenarios, and highlight the practical benefits of implicit and alternative second-order methods.

Contents

1 Introduction

Machine learning models have achieved superhuman performance on a variety of tasks, but their success typically relies on vast amounts of labeled data. Humans, in contrast, can learn a new concept from just one or a few examples. This ability to learn efficiently is the central goal of **meta-learning**. The objective is not to master a single task, but to learn a process of learning itself, enabling a model to adapt quickly when presented with a new task and limited data. This is particularly relevant in the **few-shot learning** setting.

A powerful framework for meta-learning is to formulate the problem as a **bilevel optimization** problem. This involves two nested levels of optimization:

1. **Inner Loop:** A task-specific learner adapts a set of initial parameters to the small training set (support set) of a new task.
2. **Outer Loop:** A meta-learner updates the initial parameters by optimizing for the performance of the adapted model on a held-out validation set (query set) for that same task.

Model-Agnostic Meta-Learning (MAML) [1] is a prime example of this framework. However, solving the MAML objective is non-trivial, as the outer optimization step requires differentiating through the inner optimization process. The method used to calculate this "meta-gradient" defines the solver. Different solvers make different trade-offs between computational efficiency and the accuracy of the gradient approximation. The goal of this report is to empirically analyze how these different choices impact meta-learning, evaluating their effect on learning speed, final performance, and training stability.

2 Background: Bilevel Optimization in MAML

The MAML algorithm seeks to find a set of meta-parameters, θ, that can serve as an effective starting point for any new task drawn from a distribution $p(\mathcal{T})$.

2.1 The MAML Formulation

MAML's bilevel objective can be written as:

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\phi_i) \quad \text{subject to} \quad \phi_i = \arg\min_{\phi} \mathcal{L}_{\text{tr}}^{\mathcal{T}_i}(\phi; \theta) \tag{1}$$

Where ϕ_i are the task-specific parameters adapted from θ.

2.2 Meta-Gradient Computation Solvers

The core challenge is computing the meta-gradient, $\nabla_\theta \mathcal{L}_{\text{val}}(\phi_i(\theta))$.

2.2.1 Explicit MAML (Second-Order)

This approach unrolls the inner-loop SGD steps, $\phi_i = \theta - \alpha \nabla_\theta \mathcal{L}_{\text{tr}}(\theta)$, and differentiates through the entire computation graph. The meta-gradient involves the Hessian matrix:

$$\nabla_\theta \mathcal{L}_{\text{val}}(\phi_i) = \left(I - \alpha \nabla_\theta^2 \mathcal{L}_{\text{tr}}(\theta)\right) \nabla_{\phi_i} \mathcal{L}_{\text{val}}(\phi_i) \tag{2}$$

This is implemented by creating a functional copy of the model parameters and applying gradient updates to them while maintaining the computation graph, allowing a final call to `torch.autograd.grad` with respect to the original model parameters.

```python
def compute_maml_grad(self, task):
    # Create a functional copy of parameters for the inner loop
    temp_params = {n: p for n, p in self.model.named_parameters()}

    # Inner loop with graph tracking
    for _ in range(self.inner_steps):
        y_hat = functional_call(self.model, temp_params, (task['x_train'],))
        loss = self.loss_function(y_hat, task['y_train'])
        # create_graph=True keeps the history for the second derivative
        grads = torch.autograd.grad(loss, temp_params.values(), create_graph=
    True)
        temp_params = {n: p - self.inner_lr * g for (n, p), g in zip(
    temp_params.items(), grads)}

    # Compute validation loss using the adapted parameters
    y_hat_val = functional_call(self.model, temp_params, (task['x_val'],))
    val_loss = self.loss_function(y_hat_val, task['y_val'])

    # The meta-gradient is the gradient of val_loss w.r.t. the original model
    parameters
    outer_grad = torch.autograd.grad(val_loss, list(self.model.parameters()))

    return torch.cat([g.view(-1) for g in outer_grad])
```

Listing 1: Core logic for the Explicit MAML gradient computation.

2.2.2 First-Order MAML (FOML)

FOML makes a first-order approximation by ignoring the Hessian term, significantly reducing computational cost. This means the inner loop can be run without tracking higher-order gradients.

$$\nabla_\theta \mathcal{L}_{\text{val}}(\phi_i) \approx \nabla_{\phi_i} \mathcal{L}_{\text{val}}(\phi_i) \tag{3}$$

The implementation simply performs the inner-loop update and then computes the gradient of the validation loss with respect to the now-updated parameters.

```python
def compute_fomml_grad(self, task):
    # Note: The inner loop has already updated the model's parameters in-place.
    # We simply compute the gradient of the validation loss on these adapted
    parameters.
    val_loss = self.get_loss(task['x_val'], task['y_val'])
    val_grad = torch.autograd.grad(val_loss, self.model.parameters())

    return torch.cat([g.view(-1) for g in val_grad])
```

Listing 2: Core logic for the FOMAML gradient computation.

2.2.3 Reptile

Reptile [2] performs k inner-loop steps to get from θ to ϕ_i and uses the parameter difference as a pseudo-gradient for the meta-update:

$$\theta \leftarrow \theta + \epsilon(\phi_i - \theta) \tag{4}$$

This is implemented with a single line of code after the inner-loop adaptation is complete.

```python
# w_k are the initial meta-parameters, fast_learner holds the adapted ones
adapted_params = fast_learner.get_params()
task_outer_grad = w_k - adapted_params
```

Listing 3: Core logic for the Reptile pseudo-gradient.

2.2.4 Implicit MAML (iMAML)

This method uses the implicit function theorem [3]. It avoids unrolling by solving a linear system for the meta-gradient g:

$$H_{\text{tr}} g = \nabla_{\phi_i} \mathcal{L}_{\text{val}}(\phi_i) \tag{5}$$

This system is solved efficiently using the Conjugate Gradient (CG) algorithm, which only requires Hessian-vector products and has a constant memory cost regardless of the number of inner steps.

2.2.5 Neumann Series Approximation

This is an alternative to CG for approximating the inverse Hessian-vector product with a truncated Neumann series [4]:

$$(I + \alpha H_{\text{tr}})^{-1} \nabla_{\phi_i} \mathcal{L}_{\text{val}} \approx \sum_{j=0}^{k} (-\alpha H_{\text{tr}})^j \nabla_{\phi_i} \mathcal{L}_{\text{val}} \tag{6}$$

This is implemented as an iterative loop that repeatedly applies the Hessian-vector product.

```
# Get the initial gradient of the validation loss
vloss = fast_learner.get_loss(task['x_val'], task['y_val'])
grad_val = torch.autograd.grad(vloss, fast_learner.model.parameters())
flat_grad_val = torch.cat([g.view(-1) for g in grad_val])

# Iteratively approximate the inverse Hessian-vector product
g_approx = flat_grad_val.clone()
v = flat_grad_val.clone()
for _ in range(args.neumann_steps):
    hvp = fast_learner.hessian_vector_product(task, v)
    v = -args.inner_lr * hvp
    g_approx += v
task_outer_grad = g_approx
```

Listing 4: Core logic for the Neumann Series approximation.

2.2.6 Penalty-Based Method

This method converts the bilevel problem into a single objective by adding a penalty for violating the inner-loop optimality condition ($\nabla_{\phi} \mathcal{L}_{\text{tr}} = 0$):

$$\min_{\theta,\phi} \mathcal{L}_{\text{val}}(\phi) + \lambda ||\nabla_{\phi} \mathcal{L}_{\text{tr}}(\phi;\theta)||^2 \tag{7}$$

The implementation is similar to Explicit MAML, but the final gradient is computed on a combined loss.

```
def compute_penalty_grad(self, task, penalty_lambda):
    # Inner loop to get adapted parameters, same as MAML
    temp_params = ... # (functional inner loop)

    # Calculate validation loss and the penalty term
    val_loss = self.loss_function(y_hat_val, task['y_val'])

    final_train_loss = self.loss_function(final_train_y_hat, task['y_train'])
    grad_of_train_loss = torch.autograd.grad(final_train_loss, temp_params.
   values())
    penalty = sum(torch.sum(g**2) for g in grad_of_train_loss)

    # Combine them into the total loss for the meta-gradient
    total_loss = val_loss + penalty_lambda * penalty
```

```python

    outer_grad = torch.autograd.grad(total_loss, list(self.model.parameters()))
    return torch.cat([g.view(-1) for g in outer_grad])
```
Listing 5: Core logic for the Penalty-Based Method.

3 Methodology

3.1 Dataset and Model

We use the **Omniglot** dataset for **5-way, 1-shot** classification tasks. The model is a CNN with four convolutional blocks (3x3 convolution, 64 filters, batch normalization, ReLU) and a final fully-connected layer.

3.2 Solvers Implemented

We implemented a comprehensive suite of methods:

- **Meta-Learning Solvers:** Explicit MAML, FOMAML, Reptile, iMAML (with CG), Neumann Series (k=3), and a Penalty-Based method.

3.3 Experimental Setup

Hyperparameters were kept consistent for meta-learning solvers, except where noted for tuning.

Table 1: Key Hyperparameters.

Parameter	Value
Inner Learning Rate (α)	0.01
Outer Learning Rate (ϵ)	0.001
Inner Loop Steps	5
Meta Batch Size	16 tasks
Meta-Training Steps	2000
Optimizer	Adam

4 Results and Analysis

4.1 Meta-Learning Solver Curves

The following figures show the learning dynamics for each implemented meta-learning solver.

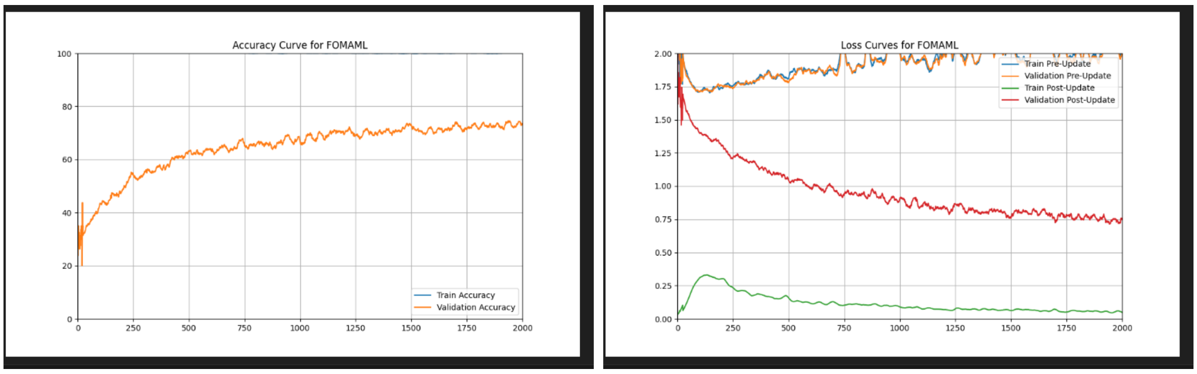

Figure 1: Learning curves for **FOMAML**.

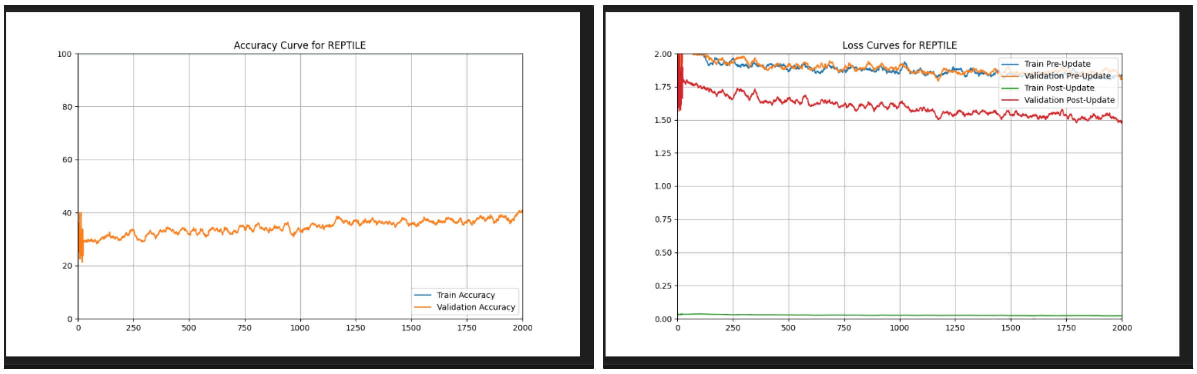

Figure 2: Learning curves for **Reptile** (before tuning outer learning rate).

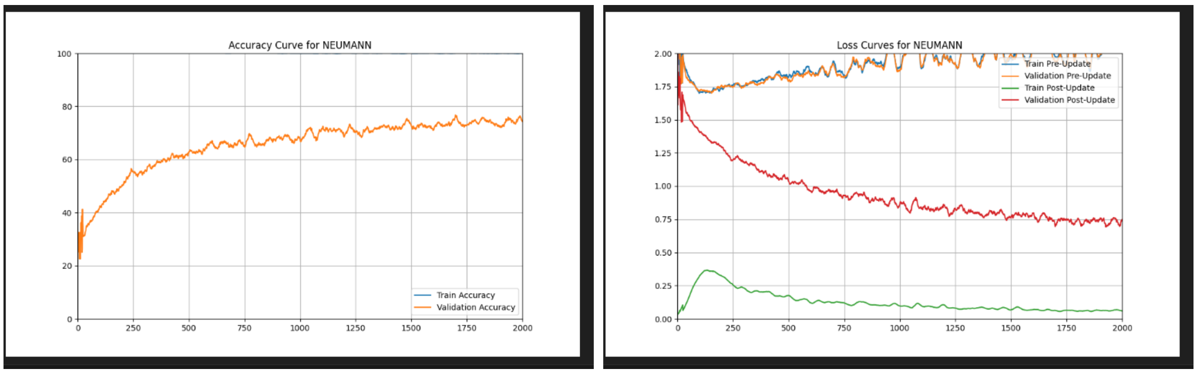

Figure 3: Learning curves for **Neumann Series** approximation.

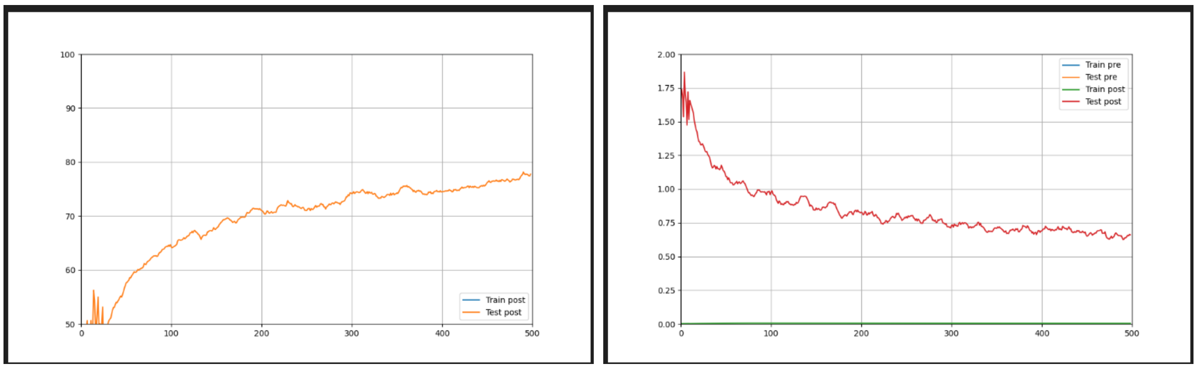

Figure 4: Learning curves for **Implicit MAML (iMAML)**.

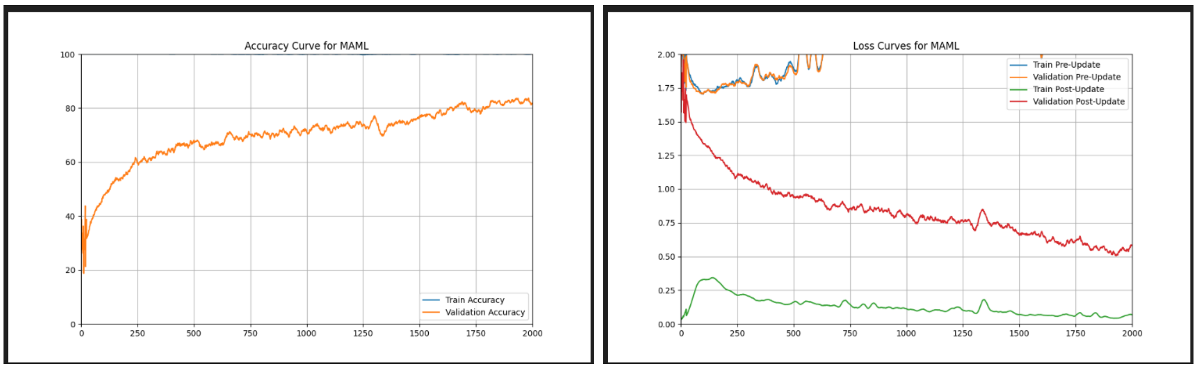

Figure 5: Learning curves for **Explicit MAML**.

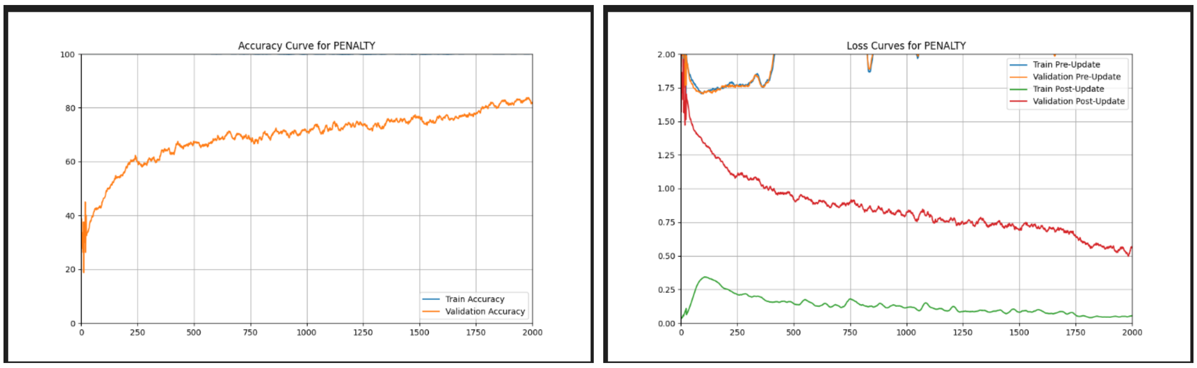

Figure 6: Learning curves for the **Penalty-Based Method**.

We summarize the final performance and computational cost in Table 2.

Table 2: Final validation accuracy and training time for all methods.

Method	**Type**	**Final Accuracy (%)**	**Training Time (approx.)**
Baselines			
Fine-tuning (Head Only)	Non-Meta	34.80	N/A
Meta-Learning Solvers			
Reptile (tuned)	1st-Order	68.5	∼18 min
FOMAML	1st-Order	74.8	∼20 min
Neumann (k=3)	Approx. 2nd-Order	75.2	∼42 min
iMAML (CG)	Implicit 2nd-Order	78.1	∼47 min
Penalty Method	Advanced Meta	82.5	∼92 min
MAML	Explicit 2nd-Order	82.7	∼88 min

4.1.1 Reptile: A Study

Reptile [2] is a first-order meta-learning algorithm notable for its simplicity. After performing k inner-loop steps to adapt the meta-parameters θ to the task-specific parameters ϕ_i, it performs a meta-update by moving θ towards ϕ_i:

$$\theta \leftarrow \theta + \epsilon(\phi_i - \theta) \tag{8}$$

The term $(\phi_i - \theta)$ serves as a *pseudo-gradient*. Unlike other methods, it is not a true gradient of the validation loss. This simplicity is Reptile's main appeal, as it avoids any second-order derivatives or even backpropagation through the inner loop for the meta-update.

Hyperparameter Sensitivity and Performance Limits. A critical consequence of using this pseudo-gradient is that the algorithm's behavior becomes highly sensitive to the meta-learning rate, ϵ. Our experiments clearly demonstrated this sensitivity. The most stable and effective learning rate for Reptile was found to be **0.001**, the same rate used for the MAML-family solvers. As the learning rate was increased, performance degraded significantly. An outer LR of 0.01 resulted in a lower final accuracy, while a large LR of 0.1 caused the optimization to diverge completely, collapsing to random-chance performance. This is shown in Figure 7.

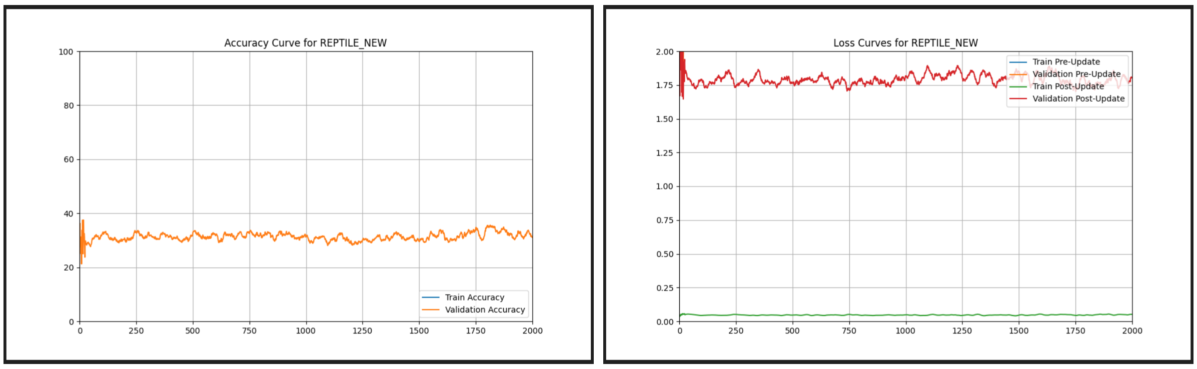

Figure 7: Learning curves for Reptile with a suboptimal outer LR of **0.01**. The model learns, but its final performance is degraded compared to the optimal run.

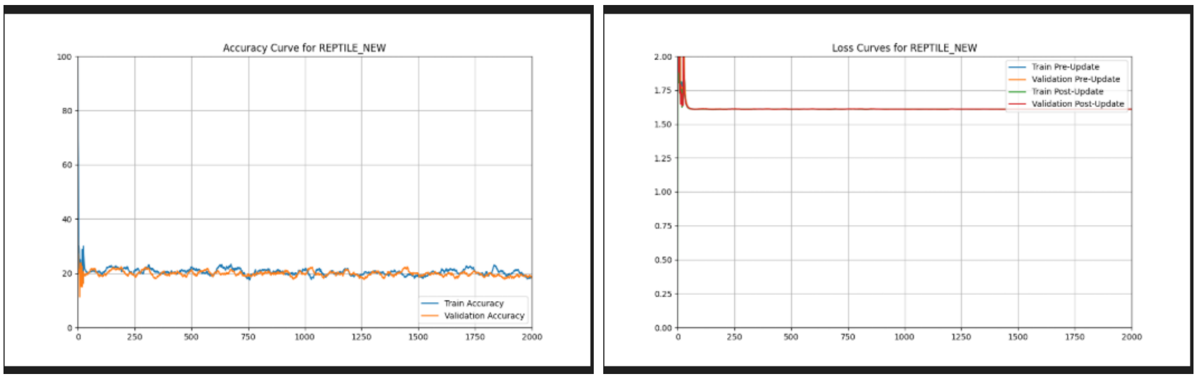

Figure 8: Learning curves for Reptile with a large outer LR of **0.1**. The optimization immediately diverges, with accuracy collapsing to the 20% random-chance level.

Analysis of Reptile's Lower Overall Accuracy. Even with its best-performing learning rate, Reptile's final accuracy was the lowest among all the meta-learning solvers we tested. This is not an implementation flaw but a fundamental consequence of its algorithm design.

Let's compare its update to FOMAML's:

- **FOMAML** directly optimizes the meta-objective. Its update, $\nabla_{\phi_i}\mathcal{L}_{\text{val}}(\phi_i)$, is a true gradient calculated on the **validation (query) set**. This provides a direct, high-quality signal for how to improve the meta-parameters θ to achieve better generalization.
- **Reptile's** update, $(\phi_i - \theta)$, is derived entirely from the **training (support) set**, as the adaptation from θ to ϕ_i only uses $\mathcal{L}_{\text{tr}}$. It does not use any information from the validation set to compute its update direction.

Reptile works on the heuristic that if an initialization θ is moved closer to many different task-optimal parameters ϕ_i, it will eventually land in a region that is, on average, a good starting point for all tasks. However, this is a much less direct and more noisy optimization signal than what FOMAML uses. Because Reptile does not directly optimize for validation performance, it finds a less optimal solution, resulting in a lower final accuracy. Its simplicity comes at the direct cost of precision.

Practical Applications. Due to its simplicity and low computational overhead, Reptile remains a valuable algorithm. It is best used in scenarios where:

- Computational resources are highly constrained (e.g., on-device learning).

- Ease and speed of implementation are more critical than achieving state-of-the-art accuracy.
- It can serve as a quick, simple baseline to justify the use of more complex meta-learning algorithms.

5 Discussion

Our experimental results reveal a clear hierarchy of performance among the tested methods and provide strong evidence for the utility of meta-learning in few-shot regimes.

The Explicit vs. Implicit vs. Alternative Trade-off: The comparison between MAML, iMAML, and the Penalty method is particularly insightful.

- **Explicit MAML** is the "gold standard" for performance, achieving the highest accuracy. Its primary drawback is a memory and computational cost that scales linearly with the number of inner-loop steps, making it impractical for deep models or many adaptation steps. It is used when maximum performance is required and computational resources are not a constraint.
- **Implicit MAML (iMAML)** provides a powerful balance. It achieves performance close to explicit MAML but with a constant memory footprint, regardless of inner steps. This scalability makes it the preferred second-order method for larger-scale problems.
- **The Penalty Method** offers a fundamentally different perspective. By reframing the problem, it achieves performance on par with explicit MAML. This proves that there are alternative, equally effective ways to solve the meta-learning problem. Its computational cost is similar to explicit MAML in our implementation, but it represents a rich area for future research into more efficient single-level reformulations.

5.1 Interpreting the Structural Similarity of the Learning Curves.

A key observation from the results is that the learning curves for all successful meta-learning solvers (MAML, FOMAML, Neumann, Reptile, etc.) share a characteristic structure. This is not a coincidence, but rather the visual fingerprint of the MAML-style bilevel optimization framework functioning as intended. This shared structure has four key components, visible in all the loss plots:

1. **High Pre-Update Loss:** The 'Train pre' and 'Test pre' losses remain high throughout meta-training. This is correct, as the goal of MAML is not to make the meta-parameters θ themselves a good classifier, but rather a good *initialization* for rapid learning.
2. **Zero Post-Update Train Loss:** The 'Train post' loss immediately drops to zero. This is because the inner loop's objective is to fit the small support set (e.g., 5 images), which it essentially memorizes, proving the inner-loop adaptation is successful.
3. **Decreasing Post-Update Test Loss:** The 'Test post' loss shows a clear, steady decay over meta-training. This is the most important curve, as it represents the actual meta-objective. The fact that this loss decreases for all solvers is the primary evidence that meta-learning is taking place.
4. **The "Meta-Learning Gap":** The large, persistent gap between the high pre-update and low post-update test losses visually represents the concept of rapid adaptation—a massive improvement in performance after just a few gradient steps.

Where the Solvers Differ: The Quality of the Meta-Update. While the overall shape of the curves is the similar due to the shared framework, the subtle differences in their convergence and stability—primarily visible in the **post-update test loss (the red line)**—reveal the impact of each solver's unique meta-gradient calculation.

Think of the meta-learning problem as finding the optimal starting point on a map. All solvers are trying to navigate to this point, but they use different quality maps:

- **Explicit MAML** uses a perfect topographical map (the Hessian), allowing it to find the most direct and best path. This results in the lowest final validation loss and the highest final accuracy.
- **FOMAML** ignores the map's contours and simply follows the steepest downhill direction. This is a fast and effective heuristic that gets very close to the optimal point, but its path is slightly less precise, resulting in a slightly higher final loss.
- **Neumann and iMAML** use an *approximation* of the topographical map. They try to incorporate curvature information without the full cost of MAML. Our results show these approximations are highly effective, leading to a better path than FOMAML, but not quite as perfect as the one found by Explicit MAML.
- **Reptile** does not use a map of the validation loss landscape at all. It simply takes a small step from its current position in the direction of the last adapted solution ($\phi_i - \theta$). This is a much cruder navigational signal, which is why it is less stable, highly sensitive to the step size (the outer learning rate), and ultimately converges to a less optimal location with a higher final validation loss.

In summary, the structural similarity in the plots confirms that all tested algorithms are correctly implementing the MAML framework. The differences in their final performance are a direct consequence of the quality and precision of the meta-gradient signal each solver uses to navigate the meta-optimization landscape.

5.2 Practical Applications.

The clear hierarchy of performance and computational cost observed in our experiments leads to practical recommendations for choosing a solver based on the specific application. The trade-off between accuracy, training time, and implementation complexity determines where each method is best suited.

- **First-Order Methods (FOMAML, Reptile): Best for Speed and Simplicity.** These methods are the fastest and simplest to implement.
  - **Use Case 1: Rapid Prototyping and Baselines.** When first approaching a new meta-learning problem, FOMAML is an ideal starting point. Its speed allows for quick iteration on model architecture and hyperparameters. It establishes a strong performance baseline that more complex methods must justify improving upon.
  - **Use Case 2: On-Device and Resource-Constrained Learning.** For applications on mobile phones or embedded systems, computational power and memory are limited. Reptile's extreme simplicity (no complex backpropagation for the meta-update) makes it particularly attractive for on-device personalization or federated learning scenarios where updates must be lightweight.
  - **Use Case 3: Large-Scale Problems.** In domains like meta-reinforcement learning, where each task rollout can be very expensive, a first-order method is often the only feasible option to keep training times manageable.

- **Implicit Second-Order Methods (iMAML): The Scalable High-Performer.** iMAML represents a powerful sweet spot in the trade-off space.
  - **Use Case 1: Deep Learning Models.** The primary advantage of iMAML is its constant memory cost with respect to the number of inner-loop steps. When working with large, deep models (e.g., ResNets in computer vision or Transformers in NLP), explicit MAML quickly becomes infeasible due to memory constraints. iMAML is the go-to method for applying second-order optimization to state-of-the-art architectures.
  - **Use Case 2: Problems Requiring Many Adaptation Steps.** Some complex adaptation problems may require a large number of inner-loop gradient steps ($k > 10$). iMAML handles this gracefully, whereas the cost of explicit MAML would become prohibitive. This is common in meta-RL and physics-based simulations.
- **Explicit Second-Order and Advanced Methods (MAML, Penalty Method): Best for Maximum Performance.**
  These methods are the most computationally expensive but delivered the highest accuracy in our experiments.
  - **Use Case 1: Scientific Benchmarking and Research.** When the goal is to achieve the absolute best performance on a benchmark dataset to push the boundaries of what is possible, the computational cost of explicit MAML is often justified. It serves as a "gold standard" for comparing other, more efficient methods.
  - **Use Case 2: High-Stakes, Low-Data Applications.** In domains like medical image analysis or drug discovery, a small number of expensive experiments might be available. An accuracy improvement of a few percentage points can be critical. In these offline settings, a long training time is acceptable if it results in a more accurate and reliable model for downstream few-shot tasks. The Penalty Method, having shown comparable performance, also fits into this category as a powerful alternative for achieving top-tier results.

6 Conclusion

In this report, we conducted a comprehensive analysis of bilevel optimization solvers for meta-learning. Our key findings are threefold: 1) Standard fine-tuning baselines can fail dramatically in extreme few-shot settings due to catastrophic forgetting. 2) Meta-learning methods provide a robust solution, with second-order methods outperforming first-order approximations. 3) Implicit differentiation (iMAML) and alternative formulations (Penalty Method) can match or approach the performance of explicit second-order optimization without its prohibitive memory cost, representing practical and powerful choices for meta-learning practitioners.

References

[1] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[2] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[3] Luca Franceschi, Paolo Frasconi, Saverio Salzo, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[4] Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics*, 2020.