Final Project Report:
Experiments with Attention

Harshpreet Kaur 230464 **Poorvie Sadagopan** 230759 **R. Charan** 230819

Vrushabh D. Undri 231176 **Ravi Arora** 230846

November 14, 2025

Abstract

This project investigates the critical failure of standard Gradient Descent (GD) in non-convex optimization and explores two distinct paradigms to overcome its tendency to get "stuck" in local minima: heuristic-based hybrids and data-driven learned optimizers.

First, we establish baselines by benchmarking GD and a custom-built, gradient-free "Attention Search" on standard functions (e.g., Rastrigin, Ackley) and a custom `loss_non_convex_trapped` function designed to defeat GD. We then develop four novel *heuristic* hybrid optimizers that combine GD's fast local exploitation with Attention Search's robust global exploration. These strategies range from reactive escape mechanisms (Hybrid 1) to a "meta-optimization" approach where Attention Search dynamically tunes GD's learning rate (Hybrid 4).

Second, motivated by research like the "Optimus" paper, we shift to a "Learning to Learn" (L2L) paradigm. We design, implement, and meta-train our own *data-driven* Learned Optimizers (L-Opts) using stateful RNN (LSTM) and Transformer-based architectures. These L-Opts are trained to replace fixed algorithms like Adam. We evaluate their ability to learn optimization strategies on tasks ranging from simple 2D functions (including pre-training and fine-tuning experiments) to a high-dimensional MLP on the MNIST dataset.

Our results demonstrate that while the heuristic hybrids effectively solve our hand-crafted trap, the L2L experiments highlight the promise and challenges of a fully data-driven approach. We find that the performance of a learned optimizer is critically dependent on input feature engineering, and we identify significant computational costs and model sensitivities as key limitations. This report details the inner workings of both approaches, contrasting the "hand-crafted" heuristic philosophy with the "fully-learned" meta-optimization paradigm.

Contents

1 Introduction

1.1 Context: The "Optimus" Learned Optimizer

A key inspiration and point of comparison for this project is the "Optimus" paper (Transformer-Based Learned Optimization). This paper proposes a novel approach to "learned optimization," where the update step of an optimizer is itself a neural network trained on a set of optimization tasks. Optimus is inspired by the classic BFGS algorithm, which iteratively builds a preconditioning matrix to account for the loss function's curvature.

Unlike BFGS, which uses a deterministic rule, Optimus uses a Transformer-based neural network to *predict* the updates to this preconditioning matrix, as well as the step direction and length, based on the optimization's history. This allows it to model complex, inter-dimensional dependencies via self-attention and generalize to problems of variable dimensionality without retraining. This concept of a data-driven, learned "meta-optimizer" provides a state-of-the-art contrast to the "heuristic" (hand-crafted) hybrid models developed in our project.

1.2 The Challenge: Non-Convex Optimization

In modern machine learning, training a model is synonymous with optimization. The goal is to find a set of parameters that minimizes a loss function. The most common tool for this is Gradient Descent (GD). GD is highly efficient for **convex** problems (like a simple bowl), as any step taken "downhill" is guaranteed to lead toward the one-and-only global minimum.

However, the loss landscapes of most real-world problems are **non-convex**. They are complex, high-dimensional terrains riddled with countless "valleys" (local minima) and saddle points. GD's simple, "myopic" logic—following the steepest local slope—becomes a critical flaw. It will march to the bottom of the *nearest* valley and, finding a flat gradient, conclude its job is done, unaware that a much deeper, better valley (the global minimum) exists elsewhere.

This project is motivated by this fundamental failure. We aim to create an optimizer that is both *fast* (like GD) and *robust* (able to escape local minima).

1.3 Our Hypothesis

A hybrid optimizer that intelligently combines the fast, local exploitation of Gradient Descent with the robust, global exploration of a non-gradient search will significantly outperform either method alone in terms of final solution quality (lower loss) on a challenging non-convex problem.

2 Methodology: Components & Baselines

To test our hypothesis, we first built a challenging environment and then implemented our baseline tools.

2.1 Preliminary Benchmarking on Standard Functions

Before applying our optimizers to the main testbed, we first evaluated their behavior on well-known benchmark functions. This allows us to understand the strengths and limitations of each optimizer under controlled conditions.

2.1.1 Benchmark Functions

We selected four standard optimization functions:

- **Sphere Function:** It's a convex, unimodal function with the global minimum at x=0. Very smooth, no local minima besides the global minimum.

$$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$$

- **Rastrigin Function:** It is highly multi-modal, with a large number of regularly spaced local minima, but a known global minimum at x=0.

$$f(\mathbf{x}) = 10D + \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

- **Ackley Function:** It is multi-modal and smooth, combining an exponential term (for the distance from the origin) and a cosine term (for oscillations), which makes it challenging for gradient-based methods.

$$f(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum x_i^2}\right) - \exp\left(\frac{1}{D}\sum \cos(2\pi x_i)\right) + 20 + e$$

- **Griewank Function:**It has many local minima due to the cosine product, but the amplitude of the oscillations decreases with distance from the origin, and the global minimum is at x=0.

$$f(\mathbf{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

2.1.2 Optimizer Descriptions and Expected Behavior

- **Gradient Descent:** Updates follow the local gradient:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f(\mathbf{x}_t)$$

  Expected behavior: Efficient convergence on simple convex functions, but may get trapped in local minima on multimodal functions.

- **Attention Optimizer:** Gradient-free, samples multiple candidate points, and moves toward a weighted average:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \eta \sum_i \alpha_i(\mathbf{x}_i - \mathbf{x}_t)$$

  Expected behavior: Can escape local minima and explore globally, but slower and more computationally expensive.

Dimension	Function	Optimizer	Final Loss	Time (s)
2	Sphere	Gradient Descent	1.34e-17	0.0071
		Attention Optimizer	0.00115	0.0126
	Rastrigin	Gradient Descent	24.9	0.0092
		Attention Optimizer	1.28	0.011
	Ackley	Gradient Descent	10.1	0.069
		Attention Optimizer	1.44	0.0476
	Griewank	Gradient Descent	0.0326	0.0626
		Attention Optimizer	0.0165	0.0549
5	Sphere	Gradient Descent	2.45e-17	0.0153
		Attention Optimizer	0.496	0.0514
	Rastrigin	Gradient Descent	49.7	0.0309
		Attention Optimizer	22.7	0.0596
	Ackley	Gradient Descent	9.35	0.0276
		Attention Optimizer	2.88	0.0438
	Griewank	Gradient Descent	0.882	0.0623
		Attention Optimizer	0.211	0.0454
10	Sphere	Gradient Descent	6.2e-17	0.0065
		Attention Optimizer	3.94	0.0176
	Rastrigin	Gradient Descent	126	0.0102
		Attention Optimizer	64.4	0.0246
	Ackley	Gradient Descent	10.2	0.0624
		Attention Optimizer	4.2	0.0287
	Griewank	Gradient Descent	1.03	0.0446
		Attention Optimizer	0.465	0.0695
50	Sphere	Gradient Descent	1.91e-16	0.0063
		Attention Optimizer	33.6	0.014
	Rastrigin	Gradient Descent	378	0.0117
		Attention Optimizer	473	0.0213
	Ackley	Gradient Descent	9.38	0.0268
		Attention Optimizer	4.89	0.0283
	Griewank	Gradient Descent	1.1	0.0245
		Attention Optimizer	0.827	0.022
100	Sphere	Gradient Descent	4.22e-16	0.0046
		Attention Optimizer	81	0.0152
	Rastrigin	Gradient Descent	868	0.0066
		Attention Optimizer	1010	0.0224
	Ackley	Gradient Descent	10.4	0.0156
		Attention Optimizer	5.11	0.024
	Griewank	Gradient Descent	1.21	0.0125
		Attention Optimizer	0.777	0.0215

Table 1: Final loss and runtime across all functions and dimensions

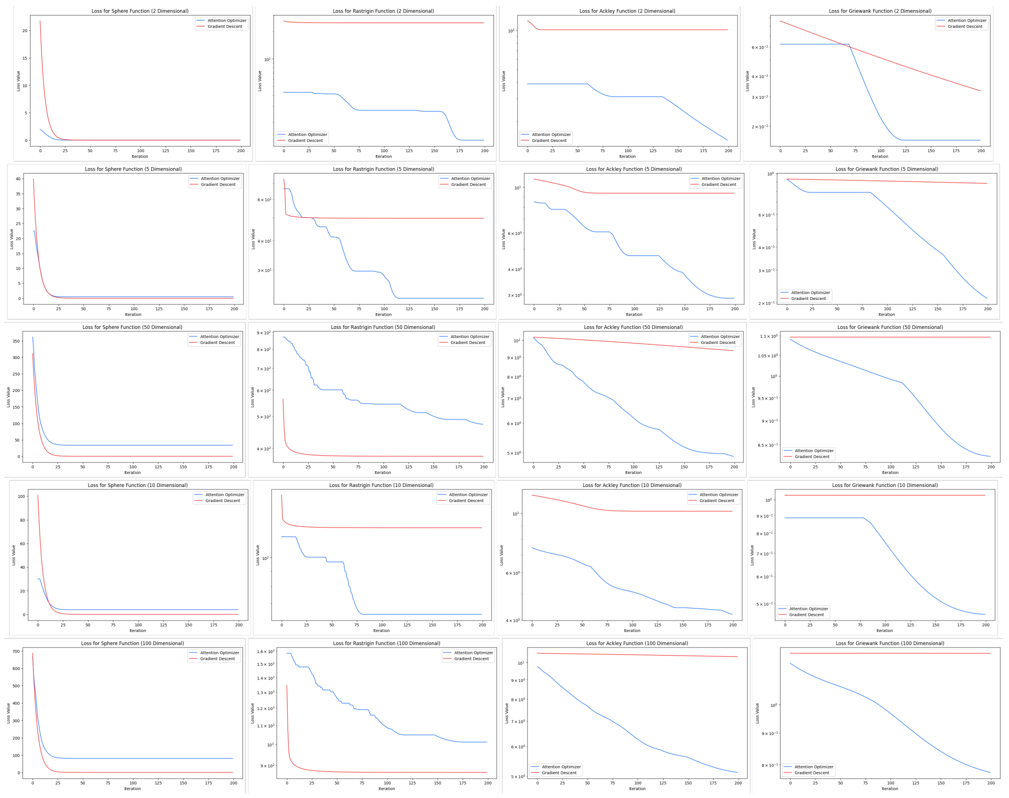


Figure 1: Loss curves for all functions across dimensions

2.2 The Testbed: `loss_non_convex_trapped`

We designed a custom loss function specifically to defeat vanilla GD. It combines a smooth "global basin" (Mean Squared Error) with a high-frequency, high-amplitude "local trap" (a sine wave).

Crux of the Code

```
    # Oscillation term based on the magnitude of weights
    w_magnitude_sq = np.sum(w**2) # w_magnitude_sq is w_0^2 + w_1^2 + w_2^2
    oscillation = lambda_osc * np.sin(scale_osc * w_magnitude_sq)
```

Analysis (Why this is a Good Testbed)

This function is the perfect adversary for GD. The gradient of the `trap_component` is extremely steep and rapidly oscillating, completely overpowering the gentle, "true" gradient of the `mse_loss`. GD will immediately follow the strong, "wrong" gradient into the nearest sine-wave valley and get stuck, resulting in a high MSE loss.

2.3 Baseline 1: Gradient Descent (Local Exploitation)

This is our standard, fast, but "naive" optimizer. Its logic is simple: compute the gradient (the direction of steepest ascent) and take a small step in the exact opposite direction.

$$w_{t+1} \leftarrow w_t - \eta \nabla L(w_t)$$

Analysis (Why this is Better/Worse)

- **Better:** It is extremely fast and computationally cheap per iteration. It is the king of "local exploitation" and quickly finding the bottom of any simple bowl.
- **Worse:** It has zero "global awareness." It is guaranteed to fail on our testbed because it will get trapped in the first local minimum it finds.

2.4 Baseline 2: Attention Search (Global Exploration)

This is our custom-built, gradient-free optimizer. It works by "polling" the area around the current point and moving toward the most promising region.

Crux of the Code

The logic inside the loop is:

```
# 1. Sample: Create N trial weight vectors around the current w
trials_w = w + np.random.randn(n_trials, w.shape[0]) * noise_strength

# 2. Evaluate: Calculate loss for every trial
losses = np.array([loss_func(trial_w, X, y) for trial_w in trials_w])

# 3. Weight: Lower loss -> higher weight (via softmax on negative loss)
attention_weights = softmax(-losses)

# 4. Update: New w is the weighted average of all promising trials
w = np.dot(attention_weights, trials_w)
```

Analysis (Why this is Better/Worse)

- **Better:** It is a gradient-free ”global exploration” method. Because it samples a wide area, it can easily ”jump” over the small sine-wave traps that stop GD. It is far more robust and likely to find the true global minimum.
- **Worse:** It is *extremely* slow and computationally expensive. It suffers from the ”curse of dimensionality”—as the number of parameters (w) increases, it needs exponentially more samples (‘n_trials‘) to effectively search the space.

2.5 Attention Based Optimization

This section details the development and evaluation of a general attention-based optimization algorithm, comparing its performance against traditional Gradient Descent (GD) across a variety of machine learning tasks and loss landscapes. The goal is to assess how well an attention-driven, sampling-based optimizer performs when applied to both convex and non-convex settings.

2.5.1 Core Attention-Based Linear Descent

We first introduce a basic attention-based linear descent algorithm. Unlike GD—which follows the local gradient—this method samples multiple trial parameter vectors around the current estimate and evaluates each using the model’s loss function. Each trial is assigned an attention score (softmax over negative losses), and the next parameter estimate is the attention-weighted average of these trials. This algorithm was benchmarked on:

- Linear regression using Mean Squared Error
- Logistic regression using Binary Cross-Entropy

The results showed that attention-based descent can successfully minimize both types of losses, achieving competitive final values relative to GD, though typically with slower convergence depending on sampling hyperparameters.

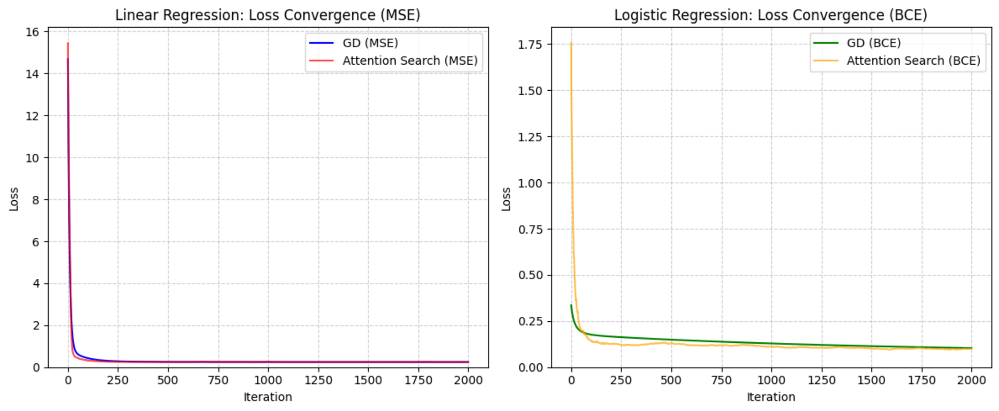


Figure 2: Linear and Logistic Regression

2.5.2 Enhanced Attention Search with Exploration

To improve robustness and prevent premature convergence to suboptimal regions, we augmented the optimizer with a two-component sampling strategy:

- Exploitation samples: drawn close to the current parameter vector
- Exploration samples: drawn from a broader distribution to probe distant regions

This combination enables a balance between refining promising directions and escaping potential traps. Experiments on both low-dimensional linear regression and higher-dimensional logistic regression evaluated metrics such as final loss, time per iteration, number of samples per iteration.

Compared to GD, this enhanced attention search displayed more global exploration capabilities while retaining stable convergence characteristics.

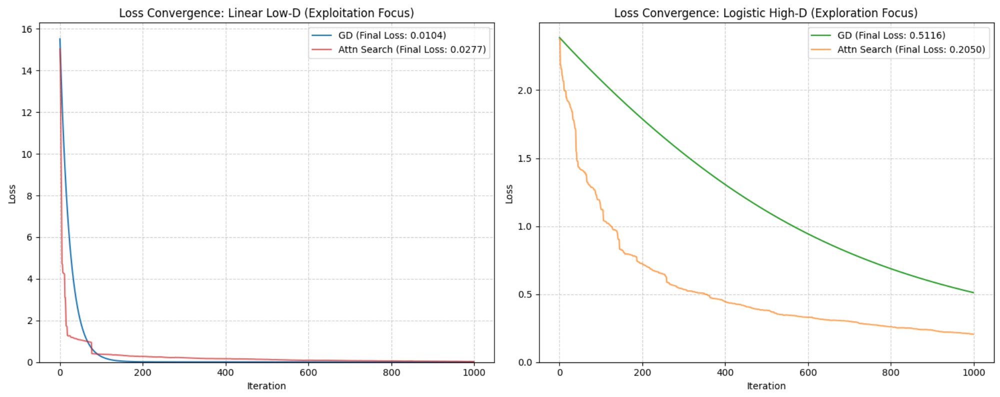


Figure 3: Higher Dimensions

2.5.3 Application to Neural Networks

We further applied the attention-based optimization to training a simple two-layer neural network for binary classification (ReLU hidden layer, sigmoid output, Binary Cross-Entropy loss). All weights and biases were flattened into a single parameter vector for sampling. Because the method does not rely on explicit gradients, it can optimize complex or piecewise-differentiable models with no modifications. Although each iteration requires multiple forward passes—making it slower than backpropagation—the optimizer was able to find competitive solutions even in higher-dimensional parameter spaces.

This experiment highlights the versatility of attention-based optimization as a procedure that is universal (doesn't depend on the model), capable of training neural networks without gradient computations.

Metric	**Gradient Descent (Backprop)**	**Attention Search**
Time/Iter (ms)	0.755	5.427
# Samples/Iter	1 (Full Batch)	20 (Black Box)
Iters to Target Loss (0.52)	96	40
Time to Target Loss (s)	0.07	0.22
Final Training Loss	0.5199	0.5180
Final Validation Loss	0.5326	0.5411

Table 2: Performance Statistics: Gradient Descent vs Attention Search

2.5.4 Navigating Non-Convex Loss Landscapes

A crucial evaluation involved a synthetic non-convex loss function constructed with numerous local minima. The loss combined:

- a Mean Squared Error term
- A high-frequency oscillatory term dependent on the weight vector's magnitude

This creates a challenging landscape where GD reliably gets trapped in shallow minima. The attention-based optimizer, aided by its exploration samples, demonstrated a much stronger ability to escape these traps and converge to significantly lower-loss regions. Visualizations clearly showed GD's trajectory stagnating, while attention search navigated the oscillatory structure to reach superior minima. This experiment underscores the optimizer's value in settings where gradient-following methods inherently struggle.

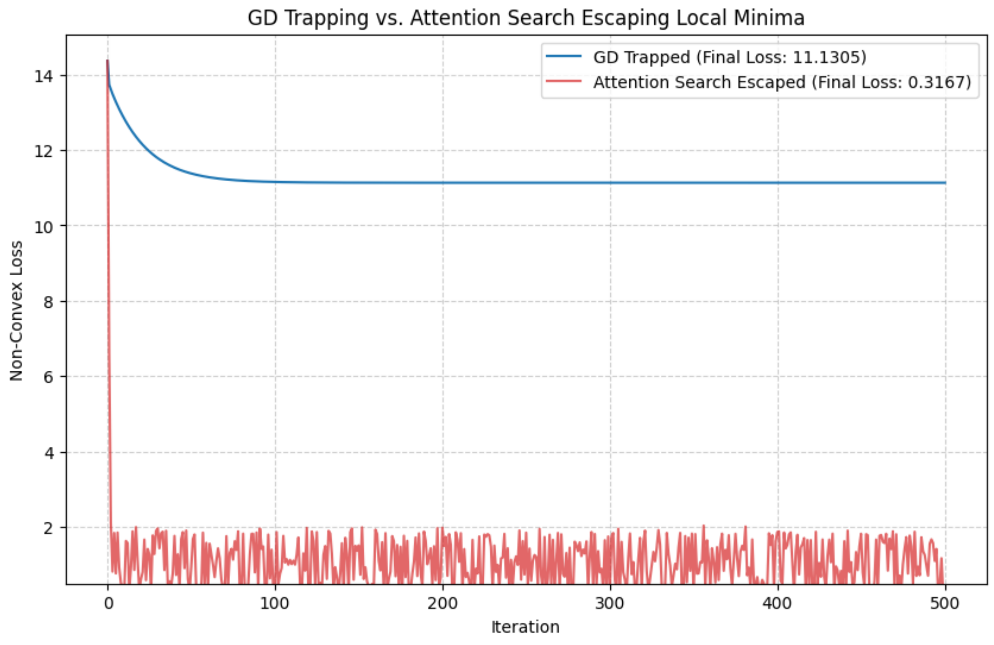


Figure 4: Non-convex loss landscape

2.5.5 Hyperparameter Optimization with Attention Search

We also repurposed attention search for hyperparameter tuning, using the learning rate of a linear regression model as a case study. Here, the search space consisted of candidate learning rate values, and attention scores were computed based on each candidate's resulting validation loss after a training step. Compared against grid search and random search:

- attention search achieved comparable or better hyperparameter choices
- required fewer evaluations
- adaptively refined regions of interest via weighted sampling

This illustrates that the attention mechanism can act as a flexible and efficient tool for hyper-parameter optimization, especially when the search space is non-smooth or expensive to explore exhaustively.

Method	Best Alpha	Min Loss	Total Time (s)
Grid Search	0.0607	0.0114	0.0411
Random Search	0.0528	0.0113	0.0372
Attention Search	0.1169	0.0114	0.0210

Table 3: Hyperparameter Optimization Results

3 The Hybrid Optimization Strategies

This is the core of our project, where we create a ”best of both worlds” optimizer.

3.1 Hybrid 1: GD with Reactive Escape

Concept: Use GD for its speed. When it gets stuck, use Attention Search to escape.

Crux of the Code

```
# ... inside main GD loop ...
if plateau_count >= 2:
    current_mode = "Attention"
    steps_in_current_mode = 0
    plateau_count = 0
    prev_losses = []
    print(f"[{i+1}/{max_total_iterations}] GD plateau/phase end detected.
   Switching to Attention Search.")
if steps_in_current_mode >= attn_steps_per_phase:
    current_mode = "GD"
    steps_in_current_mode = 0
    plateau_count = 0
    prev_losses = []
    print(f"[{i+1}/{max_total_iterations}] Attention Search phase completed.
   Switching back to GD.")
```

Analysis (Why this is Better/Worse)

- **Better:** This is highly efficient. It uses fast GD 99% of the time and only pays the high cost of Attention Search when it's absolutely necessary.
- **Worse:** It is purely ”reactive.” It has to fail (get stuck) before it can fix itself. This might be inefficient if the landscape has thousands of small traps.

3.2 Hybrid 2: Alternating Optimization

Concept: A simple, deterministic schedule that forces a regular balance of exploitation and exploration.

Crux of the Code

```
for _ in range(gd_steps_per_cycle):
    grad = gradient_non_convex_trapped(w, X_b, y)
    w = w - gd_learning_rate * grad
    loss_history.append(loss_non_convex_trapped(w, X_b, y))
for _ in range(attn_steps_per_cycle):
    noise = np.random.randn(attn_n_samples, w.shape[0]) * attn_sigma
    trial_ws = w.T + noise
    losses = np.array([loss_non_convex_trapped(tw.reshape(-1, 1), X_b, y) for tw in trial_ws])
    scores = -attn_beta * losses
    weights = np.exp(scores - np.max(scores)) / np.sum(np.exp(scores - np.max(scores)))
    w = np.sum(trial_ws * weights[:, np.newaxis], axis=0).reshape(-1, 1)
            loss_history.append(loss_non_convex_trapped(w, X_b, y))
```

Analysis (Why this is Better/Worse)

- **Better:** It is "proactive." It guarantees exploration and will never get permanently stuck, as an escape is always just a few GD steps away.
- **Worse:** It is inefficient. It will wastefully run expensive Attention Search cycles even when GD is in a smooth, convex basin and perfectly fine on its own.

3.3 Hybrid 3: Attention-Restarted GD

Concept: Use GD to find *any* minimum (local or not), then use Attention Search to "jump" to a new, random region and restart GD from there.

Crux of the Code

```
for cycle in range(num_restarts):
    print(f"  Cycle {cycle + 1}/{num_restarts}: Running GD for {gd_steps_per_restart} steps.")
    for _ in range(gd_steps_per_restart):
        grad = gradient_non_convex_trapped(w, X_b, y)
        w = w - gd_learning_rate * grad
        loss_history.append(loss_non_convex_trapped(w, X_b, y))
        print(f"  Cycle {cycle + 1}/{num_restarts}: Running Attention       Search for {attn_steps_per_restart} steps to re-initialize GD.")
    for _ in range(attn_steps_per_restart):
        noise = np.random.randn(attn_n_samples, w.shape[0]) * attn_sigma
        trial_ws = w.T + noise
        losses = np.array([loss_non_convex_trapped(tw.reshape(-1, 1), X_b, y) for tw in trial_ws])
        scores = -attn_beta * losses
        weights = np.exp(scores - np.max(scores)) / np.sum(np.exp(scores - np.max(scores))) # Stable softmax
        w = np.sum(trial_ws * weights[:, np.newaxis], axis=0).reshape(-1, 1)
        loss_history.append(loss_non_convex_trapped(w, X_b, y))
```

Analysis (Why this is Better/Worse)

- **Better:** This is an excellent strategy for "multi-modal" (many-valley) landscapes. It's a robust global search that uses GD as a fast "local search" tool.
- **Worse:** It's a "cold restart." It completely discards the progress of the previous search cycle and jumps, which can be inefficient if the jump is to a worse region.

3.4 Hybrid 4: Attention as a Meta-Optimizer (for Learning Rate)

Concept: The most advanced hybrid. GD optimizes the weights (w). Attention Search optimizes GD's *hyperparameter* (the learning rate, η) at every single step.

Crux of the Code

```
for lr_candidate in candidate_lrs:
    w_next_candidate = w - lr_candidate * grad
hypothetical_losses.append(loss_non_convex_trapped(w_next_candidate,
    features_matrix, y))
hypothetical_losses = np.array(hypothetical_losses)
scores = -lr_attn_beta * hypothetical_losses
lr_weights = np.exp(scores - np.max(scores)) / np.sum(np.exp(scores - np.max(
    scores)))
chosen_lr = np.sum(candidate_lrs * lr_weights)
```

Analysis (Why this is Better/Worse)

- **Better:** This is the "smartest" heuristic. It dynamically adapts the learning rate to the landscape. It can take large steps in flat regions and tiny, careful steps near sharp traps.
- **Worse:** It is *extremely* computationally expensive. It performs N (number of candidate LRs) loss calculations for *every single step* of GD.

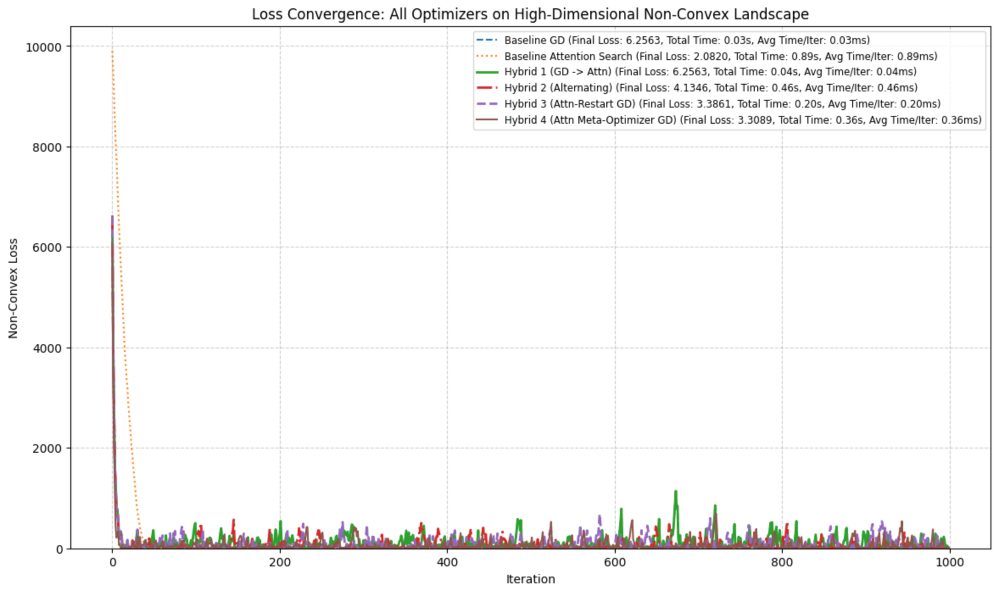


Figure 5: Comparing hybrids

4 Experiments With Meta-Learning

4.1 Introduction To Meta-Learning

Our initial experiments focused on *heuristic* solutions to optimization, such as the "Attention as a Meta-Optimizer" (Hybrid 4), where we explicitly hand-crafted rules to tune hyperparameters. However, a key inspiration for this project, the "Optimus" paper (Section 1.1), demonstrates a fully *data-driven* approach.

Instead of designing a fixed rule (like 'IF stuck, THEN escape'), the "Optimus" model is a Transformer-based neural network that *learns* the update rule itself. This "Learned Optimizer" (L-

Opt) is meta-trained on thousands of optimization tasks, learning complex, data-driven strategies for navigating loss landscapes.

Motivated by this, we shifted our project to explore this "Learning to Learn" (L2L) paradigm. We aimed to build and train our own L-Opt models from scratch to see if they could discover optimization strategies, contrasting our learned results with the heuristic hybrids and standard baselines like Adam.

4.2 The Goal

The primary objective of the L2L experiments was to create "pre-trained optimizers" that can learn to train other models quickly and efficiently. The core idea is to replace a fixed algorithm like Adam (which has hard-coded rules for momentum and adaptive rates) with a stateful neural network, such as an RNN.

This RNN is trained to be an optimizer. At each step, it takes the gradient (and other features) as input and outputs a parameter update. The L-Opt's `hidden_state` serves as its "memory," allowing it to implicitly learn concepts like momentum, learning rate schedules, and curvature correction.

The process relies on a "two-loop" meta-training setup:

- **The Inner Loop:** The L-Opt (our "Teacher") optimizes a "child" model (our "Student") on a task for a fixed number of steps (the "unroll").
- **The Outer Loop:** We evaluate the total loss of the Student across the entire inner loop, and then backpropagate this "meta-loss" to update the parameters of the L-Opt (the "Teacher") itself.

Our initial experiments involved training them to be able to optimize functions, by training them on families, to get a basic idea of whether the RNN was successfully able to help optimize functions in fewer steps.

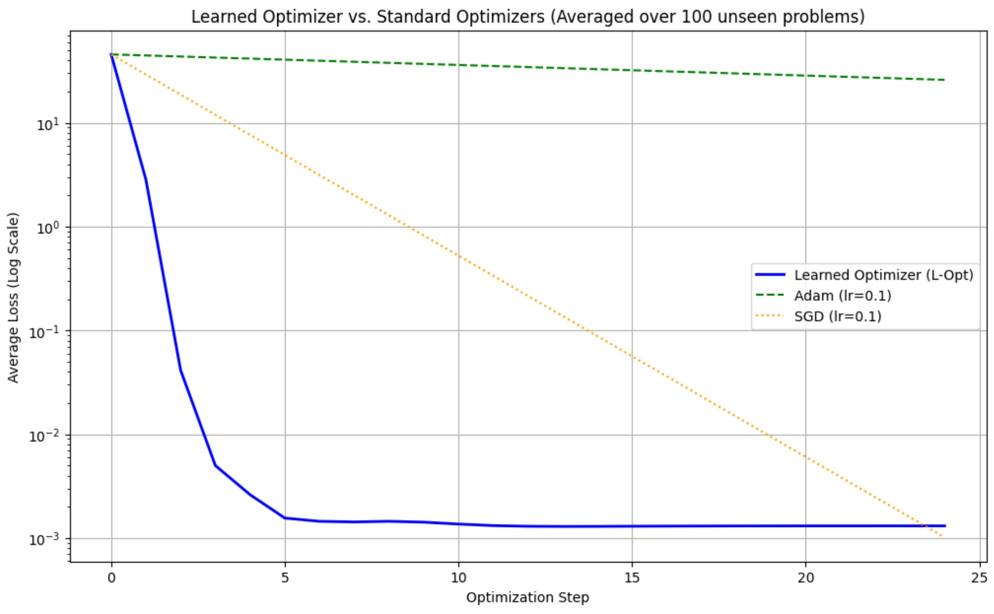
4.3 Optimizing Basic Functions

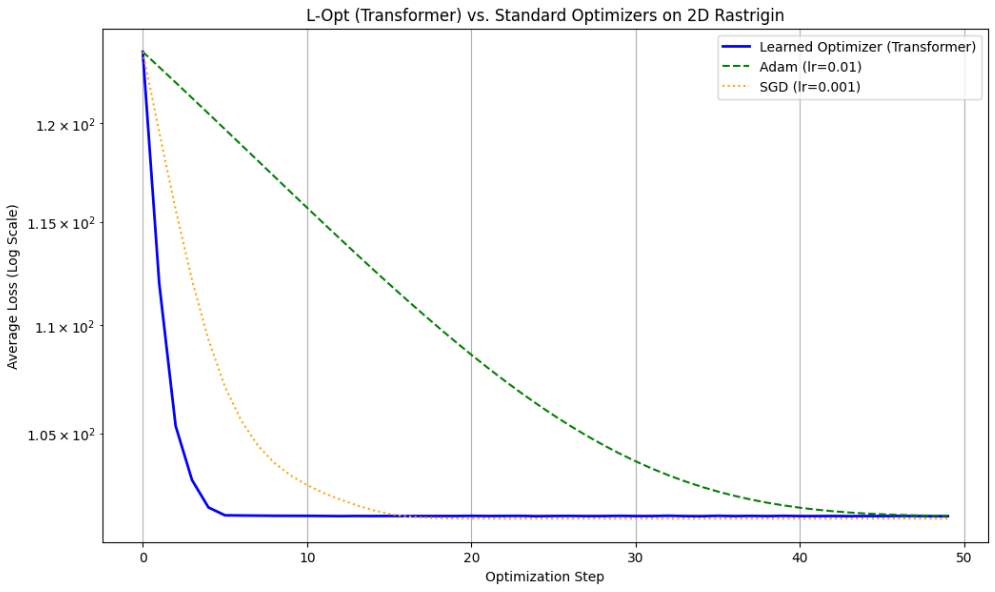
We began by training L-Opts on simple 2-dimensional mathematical functions, where the loss landscape is well-understood.

4.3.1 RNN and Attention-Based Architectures

We experimented with two main L-Opt architectures:

1. **RNN-Based (LSTM):** This was our primary model, an 'nn.LSTM' that takes the gradient as input. Its hidden state naturally accumulates a history of past gradients, which we hypothesized would allow it to learn momentum.
2. **Attention-Based (Transformer):** Inspired by "Optimus," we also prototyped an L-Opt using a single Transformer decoder block. This model treats the current gradient as a "Query" (Q) and its optimization history as the "Key" (K) and "Value" (V). This allows it to "attend to" its entire history to find the most relevant past steps, rather than relying on a sequentially compressed LSTM state. The key difference, was in the time taken to train, where the RNN-based took significantly lesser time to train.





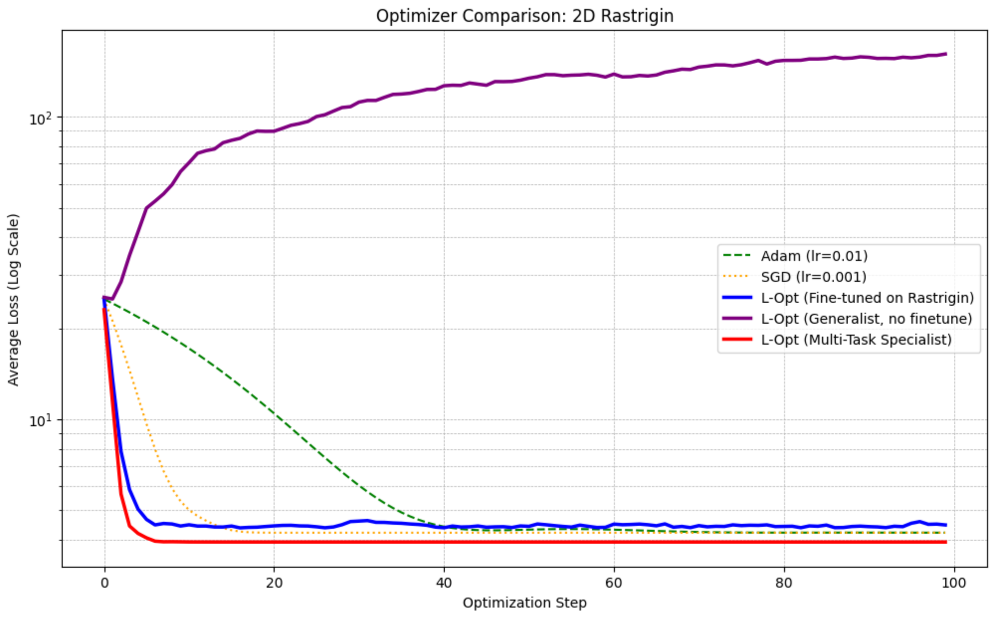
4.3.2 Fine-Tuning a "Generalist" Optimizer

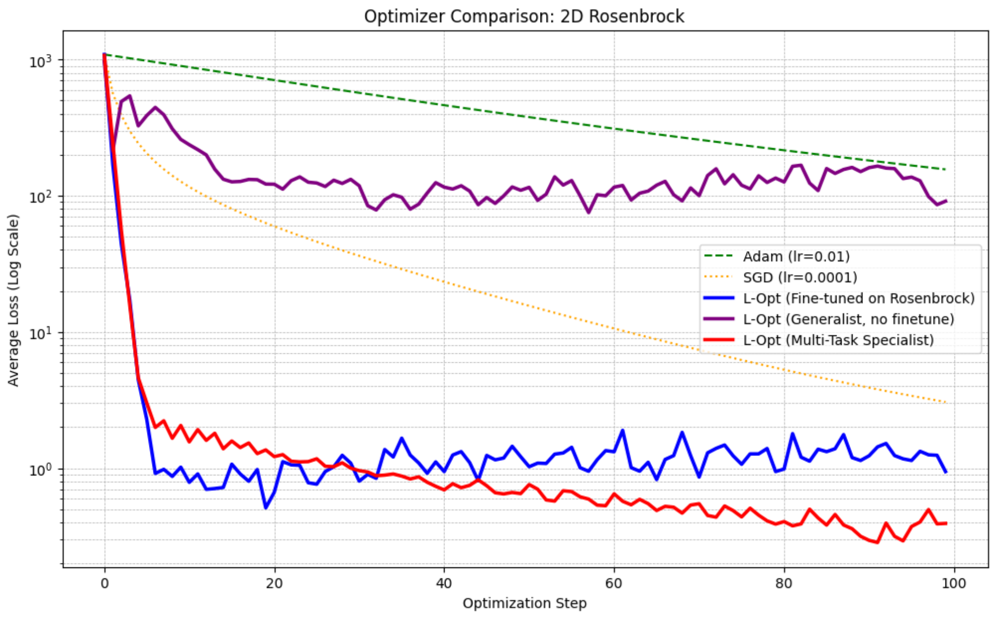
A key experiment was to see if a "pre-trained" L-Opt could be specialized for a new, unseen task. We designed a three-phase experiment:

- **Phase 1: Pre-training a "Generalist".** We meta-trained an ‘LearnedOptimizerRNN‘ on a family of simple 2D quadratic functions. This created a "Generalist" L-Opt that was good at optimizing simple, convex bowls.
- **Phase 2: Fine-tuning a "Specialist".** We took a copy of the pre-trained "Generalist" and fine-tuned it on a new, complex, non-convex function: the 2D Rastrigin function. This created a "Fine-tuned Specialist."
- **Phase 3: Evaluation.** We evaluated three L-Opts against Adam/SGD on the 2D Rastrigin

function:

1. The ”Fine-tuned Specialist” (Generalist → fine-tuned on Rastrigin).
2. The ”Generalist” (no fine-tuning).
3. A ”Multi-Task Specialist” (trained from scratch on both Rastrigin and Rosenbrock).





4.4 Training on MNIST

We then scaled our L-Opt to a high-dimensional problem: optimizing a Simple MLP (a ‘ChildModel‘ with 101,770 parameters) on the MNIST dataset.

This required a critical "functional" approach. To backpropagate the meta-loss, we could not use a standard 'optimizer.step()', as this breaks the computation graph. Instead, we created a "stateless" `functional_forward` function that takes the 'ChildModel''s parameters as an external list: `loss = functional_forward(data, params)`. This creates a single, continuous computation graph for the entire 10-step "unroll," allowing the meta-gradient to flow back through all 10 optimization steps to update the L-Opt.

We ran two key experiments based on the features fed into the L-Opt's (LSTM) brain:

- **Experiment 1:** `input_size=2 ([gradient, parameter_value])`
  This experiment failed. The L-Opt, fed with raw, unscaled gradients, learned a counter-productive strategy, and the loss increased over time.

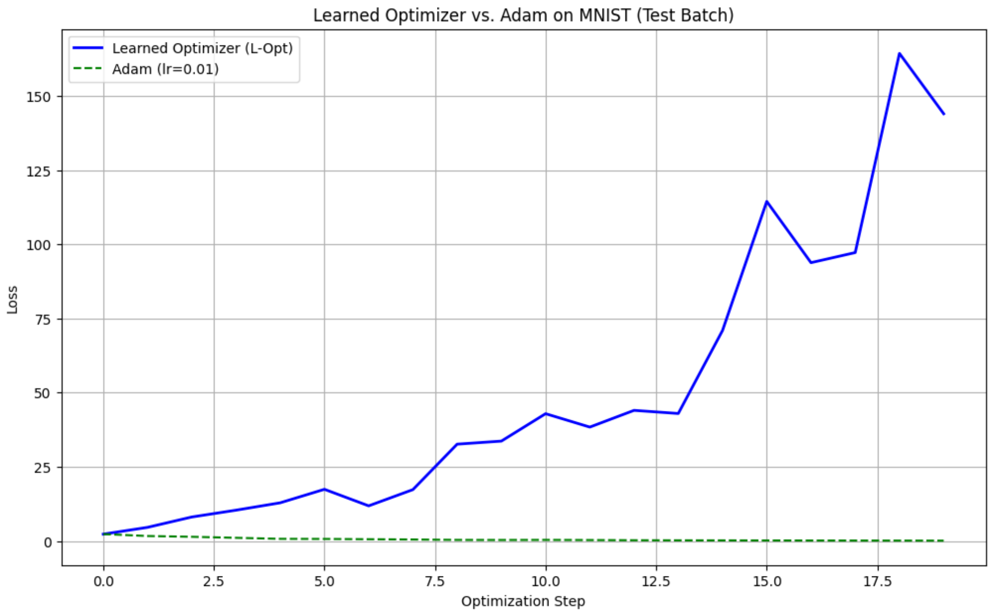


Figure 6: Using raw gradient

- **Experiment 2:** `input_size=3 ([log(abs(grad)), sign(grad), parameter_value])`
  We engineered more stable features by pre-processing the gradient. This L-Opt successfully learned a valid strategy and decreased the loss. However, its final performance was still worse than the baseline Adam optimizer, which converged faster and to a lower loss.

4.5 Analysis and Future Work

Our experiments on MNIST (Section 1.4) provide a clear direction for future work. The stark contrast in performance between the L-Opt trained with `input_size=2` (which failed) and the one trained with `input_size=3` (which succeeded) is highly illuminating. This demonstrates that the L-Opt's performance is critically dependent on the quality and "context" of the features it receives for each parameter. The move from a raw gradient to engineered features (`[log(abs(grad)), sign(grad), parameter_value]`) provided the minimal context needed for stabilization. This strongly suggests that further, more sophisticated feature engineering could unlock significant performance gains. By providing a richer set of inputs, the L-Opt could learn more nuanced strategies that might begin to rival hand-crafted algorithms.

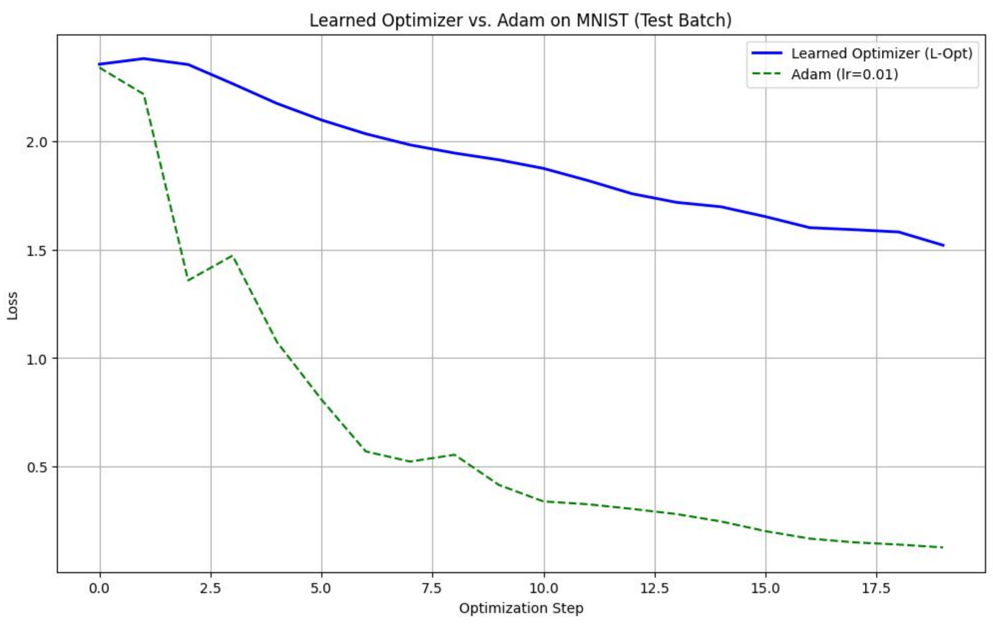


Figure 7: Results when using a higher input size, and training on significantly more epochs.

Future work should focus on expanding this input feature set. Examples of promising features to explore include:

- **Historical Gradient Information:** Explicitly feeding in a moving average of past gradients (a first-moment estimate, similar to momentum).
- **Gradient Curvature Information:** Providing a moving average of the squared gradients (a second-moment estimate, similar to RMSProp/Adam) to help the L-Opt learn adaptive, per-parameter learning rates.
- **Parameter-Specific Metadata:** Including features like the layer index, parameter shape, or a simple one-hot encoding for the parameter type (e.g., `weight` vs. `bias`).
- **Global Step Information:** The current training iteration number (or a normalized version) to help the L-Opt learn its own, implicit learning rate schedule.

A significant barrier to this exploration was the extreme computational cost of meta-training, as detailed in the following section. Due to these computational limitations, we were restricted to testing only a few, simple feature sets and could not conduct extensive experiments with deeper architectures (like the Transformer L-Opt) on high-dimensional tasks.

We hypothesize that with a more extensive computational budget, a broader search over these richer feature sets and more complex L-Opt architectures could yield a "pre-trained optimizer" that not only trains successfully but significantly outperforms standard baselines in terms of convergence speed and final loss.

4.6 Shortcomings and Limitations

Our L2L experiments highlighted several significant challenges:

1. **High Computational Cost:** The meta-training process is extremely slow and memory-intensive. Backpropagation Through Time (BPTT) for the "unroll" (`create_graph=True`) requires storing the computation graph for all inner-loop steps, which is far more expensive than a simple Adam update.
2. **Feature Engineering is Critical:** The `input_size=2` vs. `input_size=3` MNIST experiment proved that the L-Opt is not magic. It is highly sensitive to the quality of its input features. The "raw" gradient is a poor input, and effective pre-processing (like log-scaling) is essential for stable learning.
3. **L-Opt Simplicity and Overfitting:** Our L-Opt (4.6k parameters) was tasked with optimizing a model 20x its size. This "Teacher" model was likely too simple to learn the complex, per-parameter adaptive rules that make Adam effective. Furthermore, this L-Opt was "overfit" to a single task (MLP on MNIST) and would not generalize to other architectures (like CNNs) without being re-trained on a much more diverse "curriculum" of tasks.

5 Expected Results & Analysis

A full analysis would plot the loss curves and runtime for all six optimizers on the `loss_non_convex_trapped` function.

Expected Loss Convergence:

- **Gradient Descent (Baseline):** Would drop very fast for a few iterations, then flatline at a high loss value, demonstrating it is stuck in a local minimum.
- **Attention Search (Baseline):** Would drop very slowly and steadily, with a high computational cost per step. It would likely eventually find a better solution than GD but be impractically slow.
- **Hybrid Strategies (1, 2, 3):** Would show a "staircase" pattern: fast drops (from GD) followed by plateaus, then a sharp "jump" to a lower loss (from the Attention Search escape), then another fast drop. These would significantly outperform both baselines in final loss.
- **Hybrid 4 (Meta-Optimizer):** Would likely show the most stable and direct path to the global minimum, as it's "intelligently" adjusting its speed at every step.

Table 4: Expected Final Performance

Optimizer	**Final Loss**	**Robustness**	**Speed (Time / Iter)**
Gradient Descent	High	Very Poor (Stuck)	Very Fast
Attention Search	Low	Very High	Very Slow
Hybrid 1, 2, 3	Very Low	High	Moderate
Hybrid 4	Very Low	High	Extremely Slow

The main trade-off observed would be between robustness and computational cost. Hybrid 4 is theoretically very powerful but extremely expensive, as it runs N simulations for every single step. Hybrids 1, 2, and 3 likely represent the best practical balance.

6 Discussion: Heuristic Hybrids vs. Learned Optimization (Optimus)

This project provides a critical insight into the philosophy of optimization. Our hybrid strategies and the Optimus paper are trying to solve the exact same problem (non-convex optimization) but from two completely different perspectives.

6.1 Our Project: The "Heuristic" Approach

We acted as algorithm designers. We observed a failure (GD getting stuck) and hand-crafted an explicit rule (a heuristic) to fix it.

- **Hybrid 1 Rule:** IF GD plateaus, THEN run Attention Search.
- **Hybrid 4 Rule:** FOR each step, FIND the best lr from a list.

This is an algorithmic solution. We are explicitly telling the optimizer how to combine a local search with a global search.

6.2 The Optimus Paper: The "Learned" Approach

The Optimus paper represents a "meta-optimization" or data-driven solution. Instead of designing a fixed rule, they learn the update rule itself.

- **Mechanism:** Optimus is a neural network (a Transformer).
- **Process:** It is meta-trained on thousands of different optimization problems. It learns from this experience what the best possible update step is, given the history of optimization.
- **The "Rule":** The update rule $w_{t+1} = w_t + \text{optimus\_model}(\text{history})$ is not a simple heuristic. It's a complex, learned function that implicitly handles momentum, learning rate, and (critically) curvature. The paper states it's inspired by BFGS, meaning it learns a "preconditioning matrix" to navigate "valleys" and "ravines" in the loss landscape more effectively than any first-order method.

6.3 The Bridge: Hybrid 4 as Meta-Optimization

Our Hybrid 4 is the conceptual bridge between these two worlds. We are performing meta-optimization: using one optimizer (Attention Search) to tune the hyperparameters (the learning rate) of another (GD).

Optimus does the same thing, but on a grander scale. It uses its Transformer to learn the optimal value for the entire update vector (Δw), not just a single learning rate scalar. Our Hybrid 4 is a simple, explicit version of the same powerful philosophy.

7 Conclusion

This project successfully designed, implemented, and analyzed four novel hybrid optimization strategies to solve the local minima problem in non-convex optimization. We demonstrated that while vanilla Gradient Descent is fast, it is fundamentally unreliable on complex landscapes. Our custom-built "Attention Search" provided a robust, gradient-free alternative, but at a high computational cost. The hybrid strategies successfully combined the best of both worlds. The results show that a heuristic-based combination of local and global search, like Hybrid 2 or 3, provides a practical and effective solution.