FOUNDATION TIER

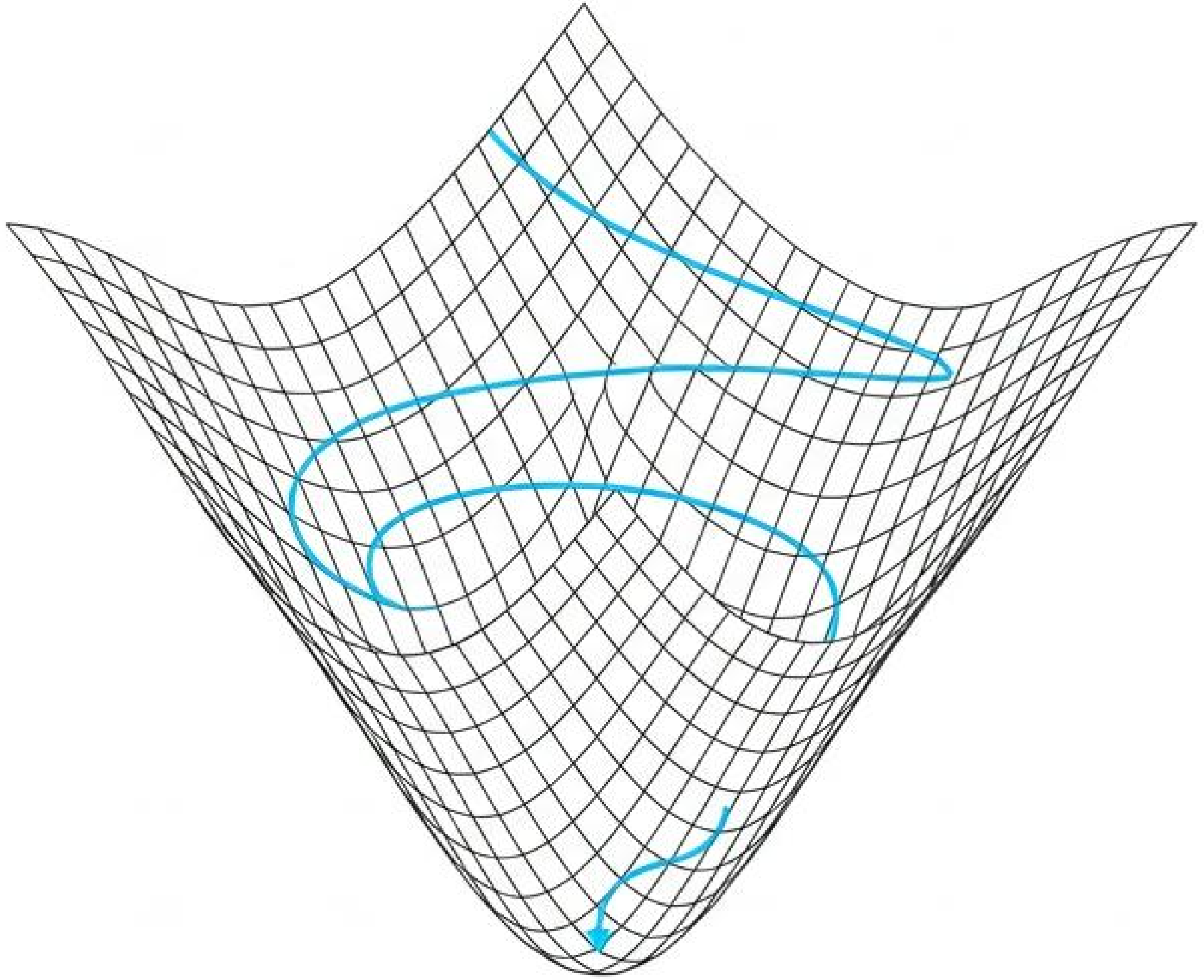
MODULE 07

Optimizers

The engines of learning that update model parameters

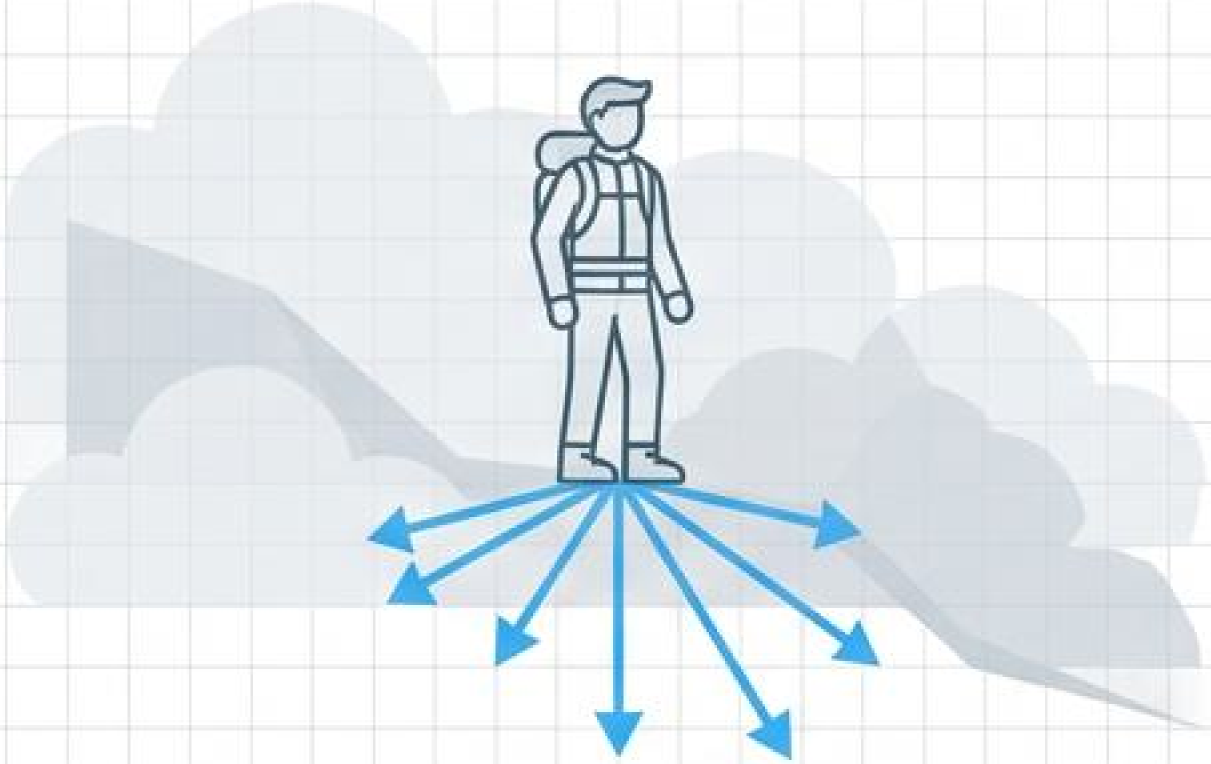
Module 07: Optimizers

Algorithms for Learning



Intuition: Hiking in the Fog

Module 06: Autograd



Sensing the slope (∇L)

Knowing which way is up.

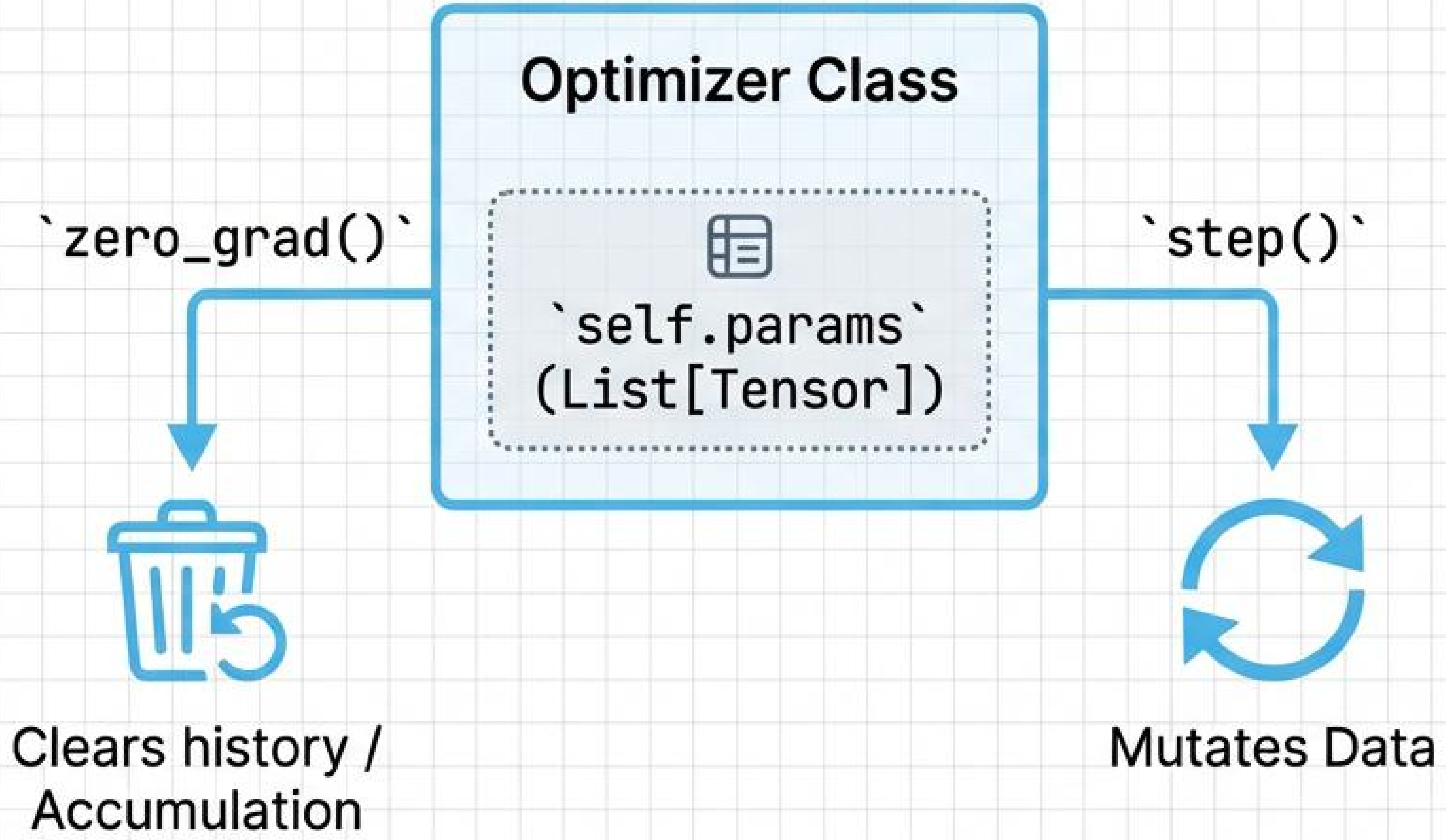
Module 07: Optimizers

Taking the step (θ_{new})

Deciding speed and strategy.

The Goal: Reach the valley floor (**Minimum Loss**) efficiently.

The Optimization Interface



- **Separation of Concerns:** Models compute gradients; Optimizers update weights.
- **State Management:** Tracks history (momentum) independent of the model.
- **Mutation:** The only component allowed to modify `param.data` in-place.

Implementing the Base Class

tinytorch/core/optimizers.py

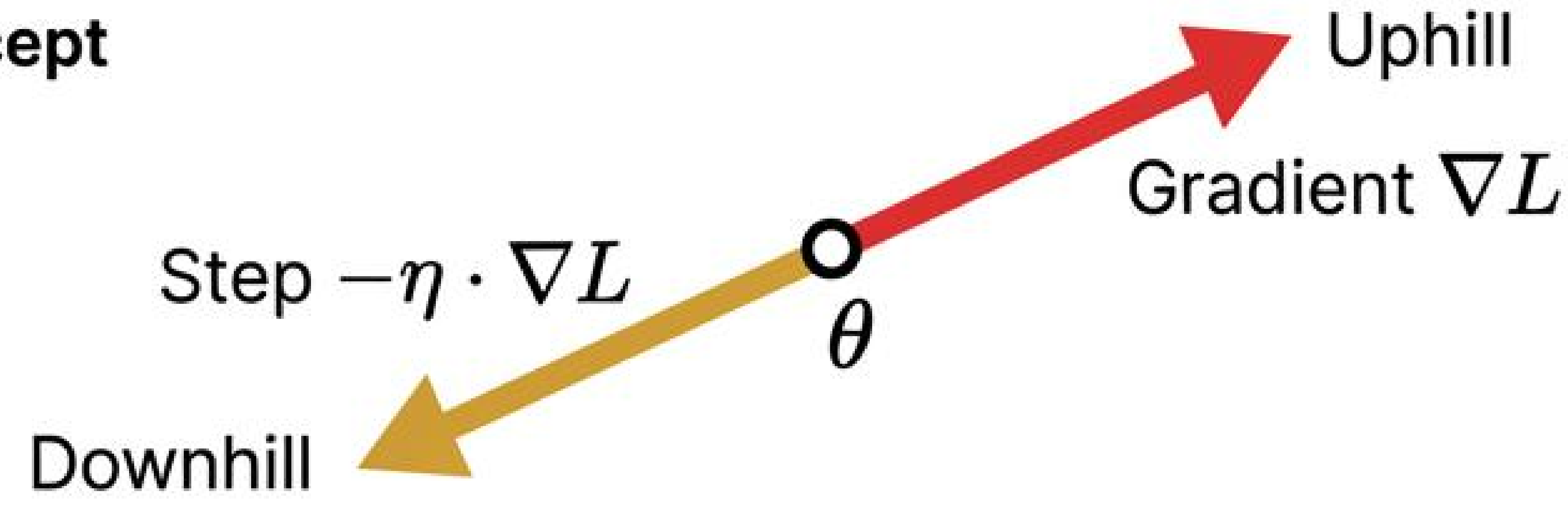
```python
class Optimizer:
    def __init__(self, params: List[Tensor]):
        self.params = list(params)
        self.step_count = 0

    def zero_grad(self):
        for param in self.params:
            param.grad = None  # Crucial: Clear accumulation

    def step(self):
        raise NoTImplementedError()
```

Must explicitly wipe gradients.
Autograd accumulates by default.

Stochastic Gradient Descent (SGD)

The Baseline

Concept



$$\theta_{new} = \theta - \eta \cdot \nabla L$$

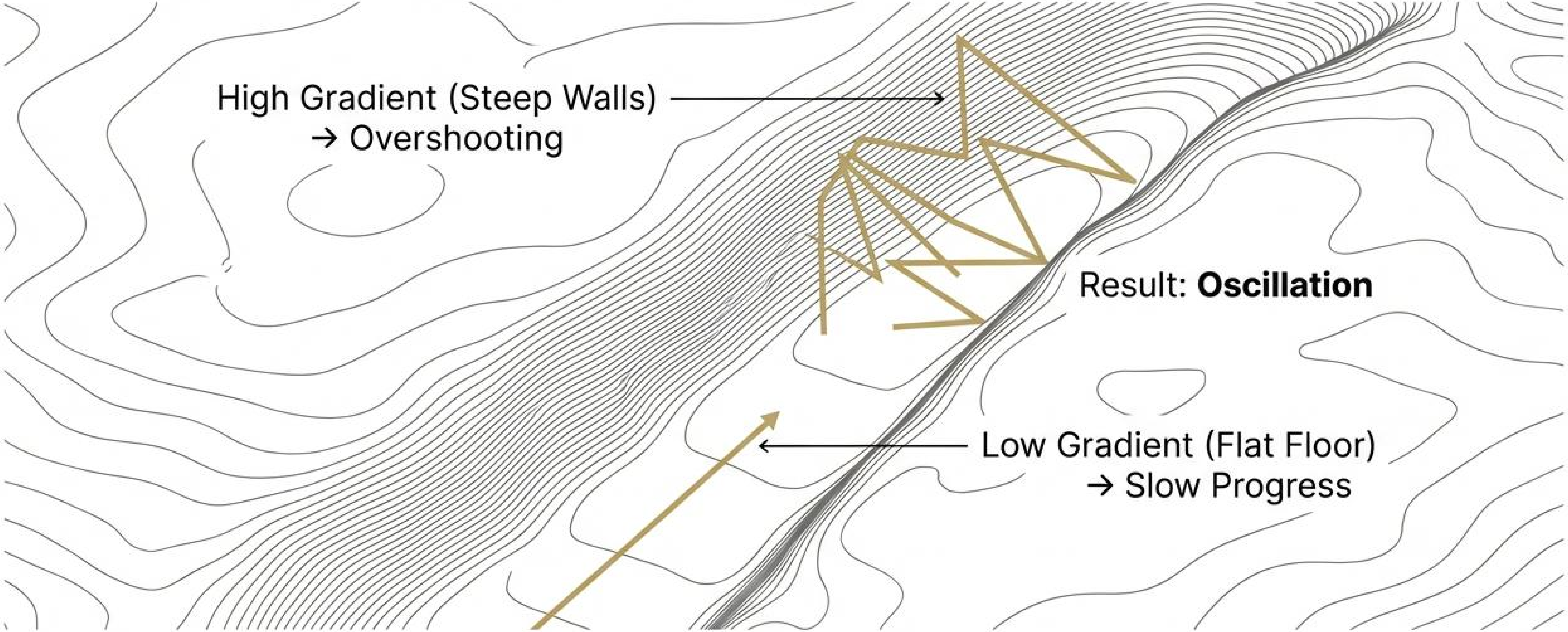
```
# Inside SGD.step()
for param in self.params:
    if param.grad is None: continue

    # Basic update rule
    param.data = param.data - self.lr * param.grad.data
```

Side Note

- **Strategy:** Always move opposite to the gradient.
- **Pros:** Minimal memory.
- **Cons:** Struggles in ravines.

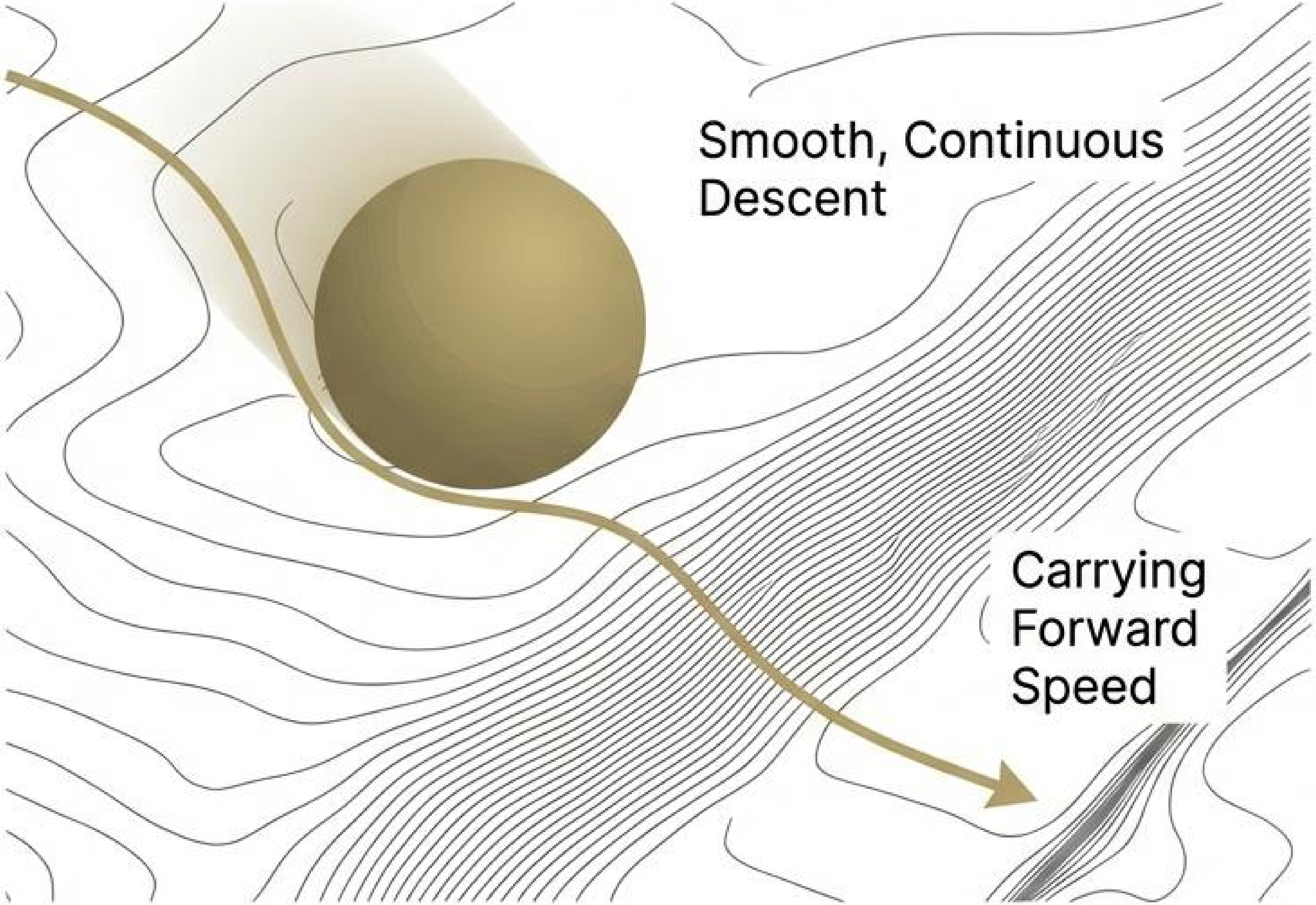
The Problem: Ravines

Why simple SGD fails



The Solution: Momentum

Physics to the Rescue



Equations and Logic

- **Velocity Buffer** (v): Accumulates direction history.
- **Friction** (β): Usually 0.9.

1. $v_t = \beta v_{t-1} + \nabla L$
2. $\theta_t = \theta_{t-1} - \alpha \cdot v_t$

Key Insight: Oscillations cancel out. Forward speed sums up.

Implementing SGD with Momentum

Systems Insight: Lazy Initialization

```
# Inside SGD.step loop
if self.momentum_buffers[i] is None:
    # Lazy initialization saves memory
    self.momentum_buffers[i] = np.zeros_like(param.data)

# Update velocity: v = momentum * v_prev + grad
self.momentum_buffers[i] = (self.momentum *
self.momentum_buffers[i]) + grad_data

# Update parameter using velocity
param.data = param.data - self.lr * self.momentum_buffers[i]
```

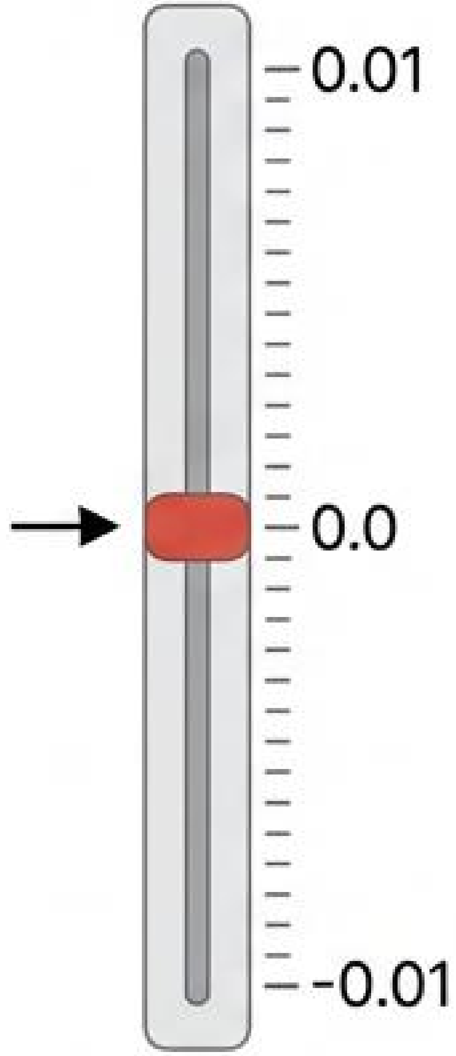
Physics: Blend history (90%) with current slope (10%)

The Challenge of Scale

Why one learning rate doesn't fit all

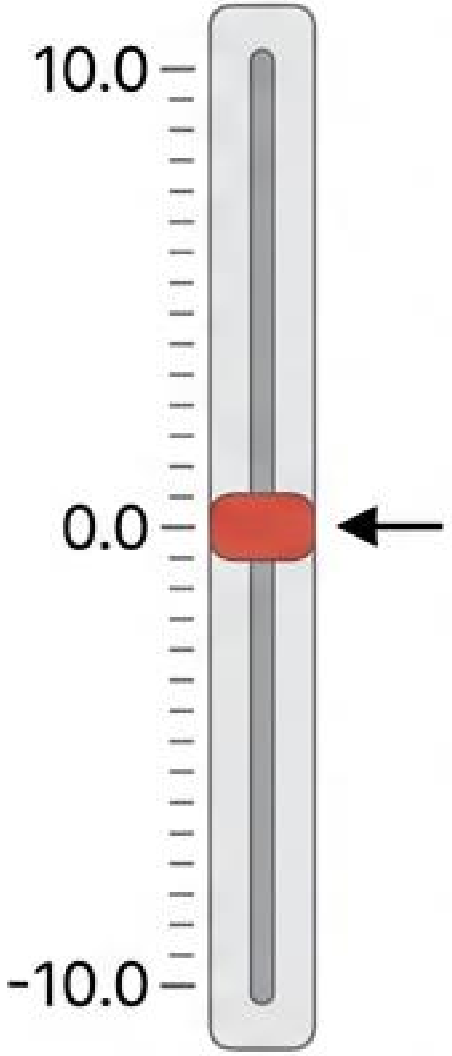
Parameter A
(e.g., Embedding)

Small Gradients.
Needs **BIG** steps.



SGD Dilemma:
A learning rate that works for A will **explode B**.
A rate safe for B will **freeze A**.

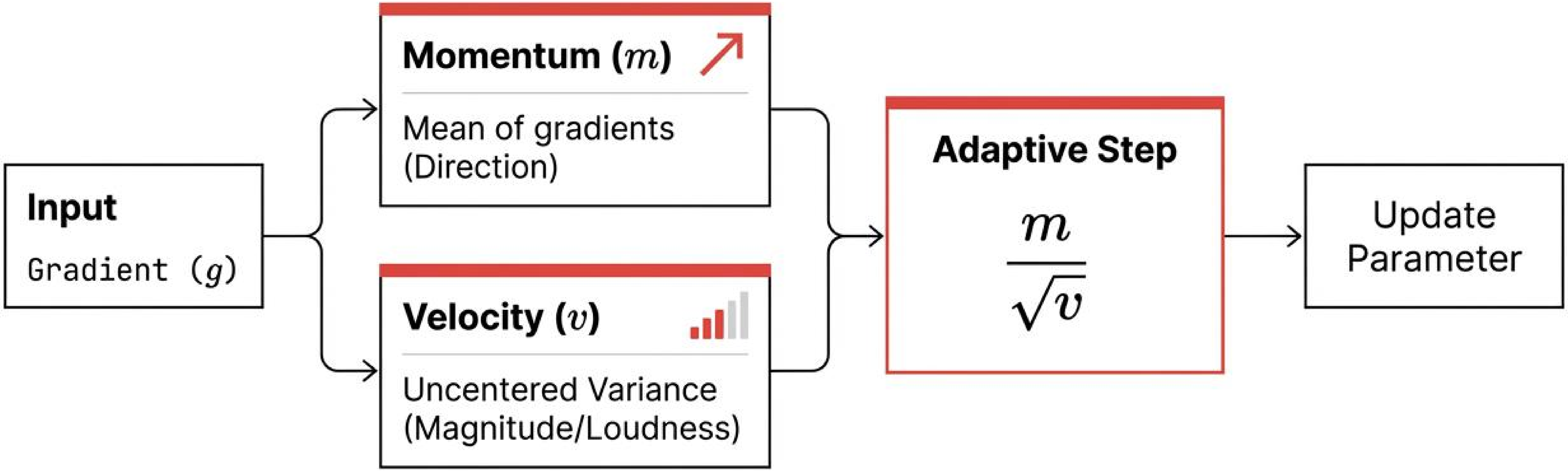
Parameter B
(e.g., Output Bias)



Massive Gradients.
Needs **TINY** steps.

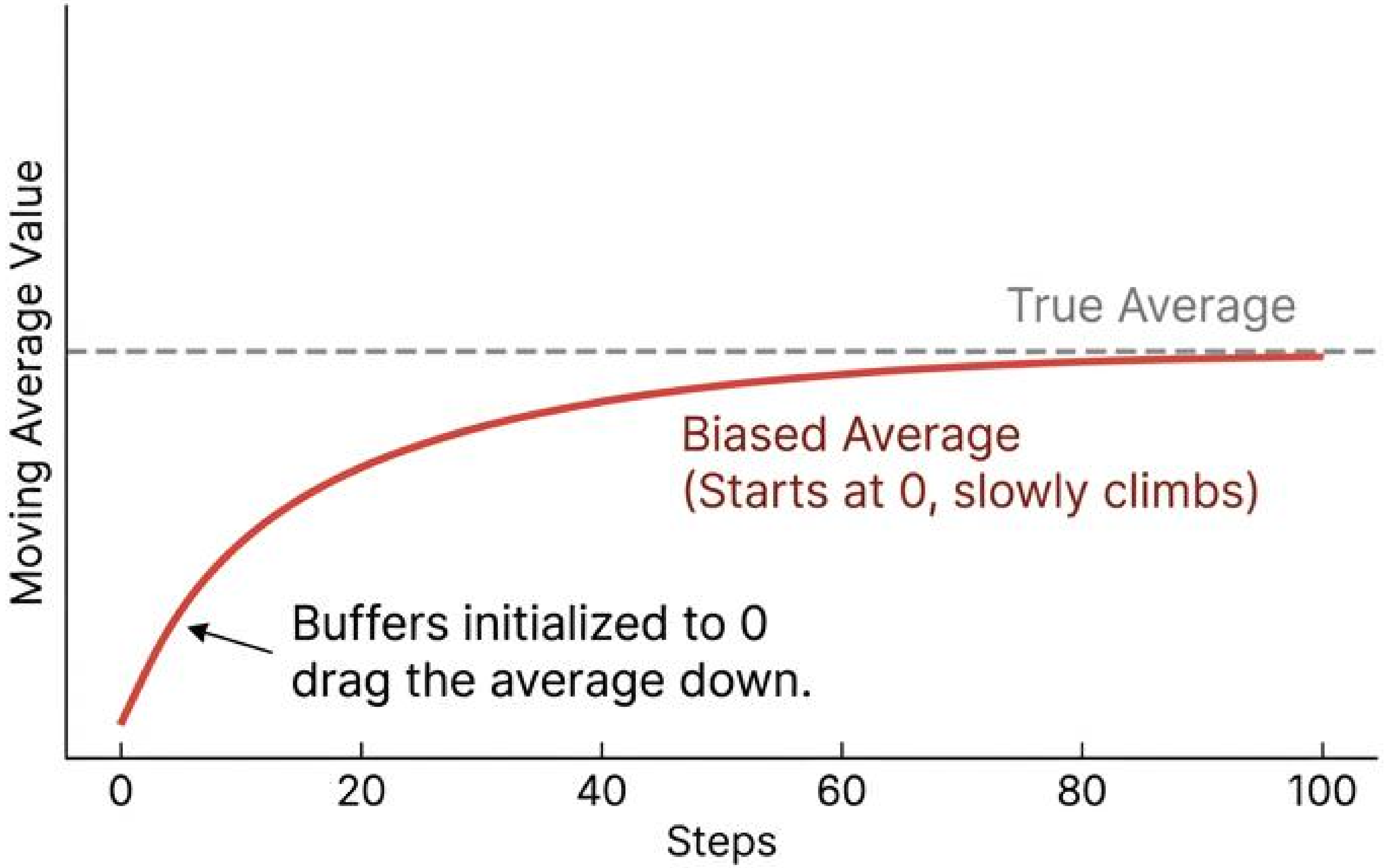
Adam: Adaptive Moment Estimation

A personal trainer for every parameter



Divide step by $\sqrt{v}$. Large gradients get throttled down. Small gradients get boosted up.

System Detail: The Cold Start Problem



The Math Fix: Bias Correction

$$m_{hat} = \frac{m}{1 - \beta^t}$$

Mathematically inflates values in early steps.

```
bias_correction1 = 1 - self.beta1 ** self.step_count
m_hat = self.m_buffers[i] / bias_correction1
```

Implementing Adam

tinytorch/core/optimizers.py

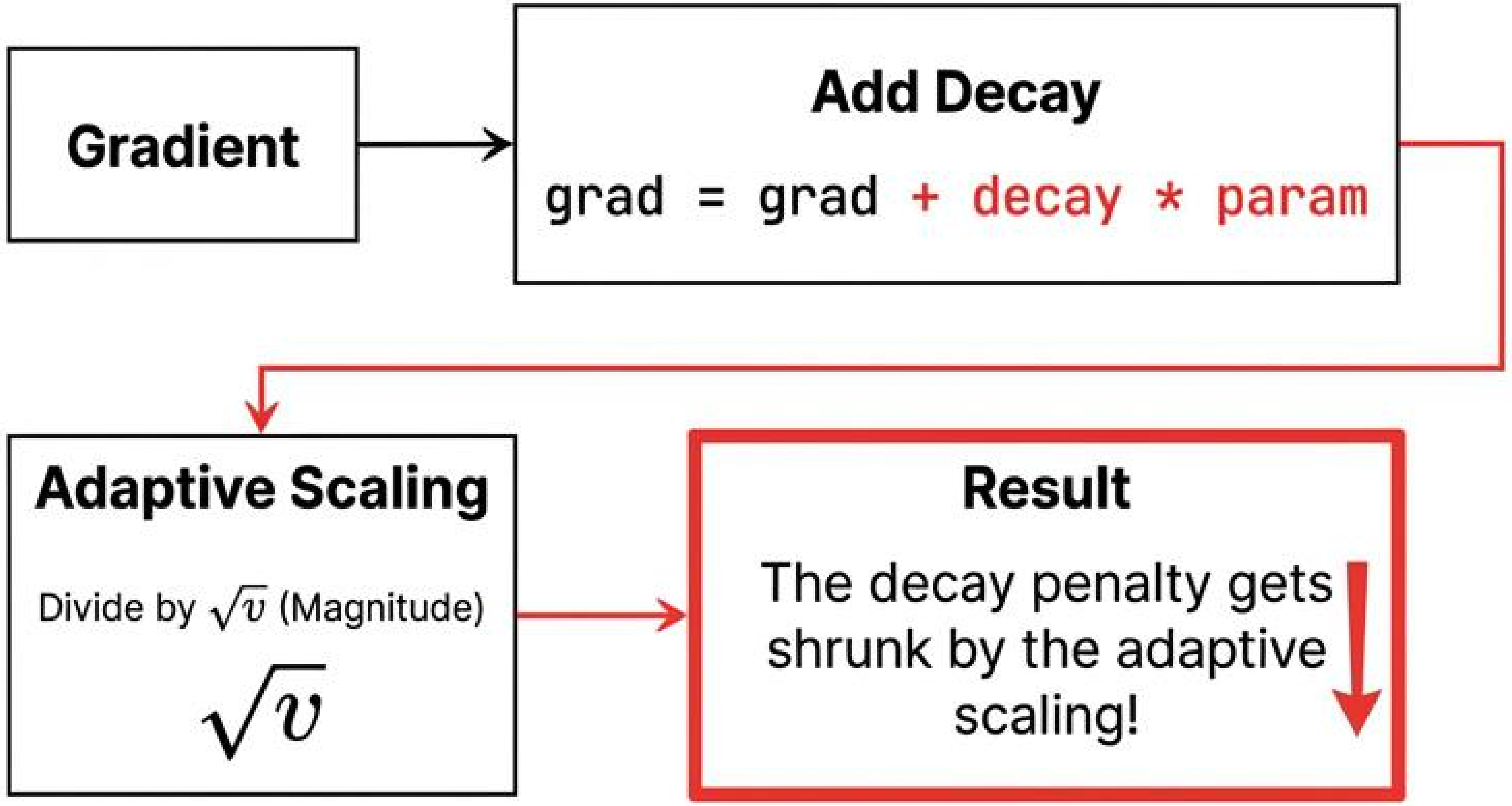
```
# 1. Update biased moments
self.m_buffers[i] = self.beta1 * self.m_buffers[i] + (1 - self.beta1) * grad_data
self.v_buffers[i] = self.beta2 * self.v_buffers[i] + (1 - self.beta2) * (grad_data ** 2)

# 2. Compute bias-corrected moments
m_hat = self.m_buffers[i] / bias_correction1
v_hat = self.v_buffers[i] / bias_correction2

# 3. Update parameter (Adaptive Step)
param.data = param.data - self.lr * m_hat / (np.sqrt(v_hat) + self.eps)
```

Tracks Magnitude

Normalization

The Bug in Adam

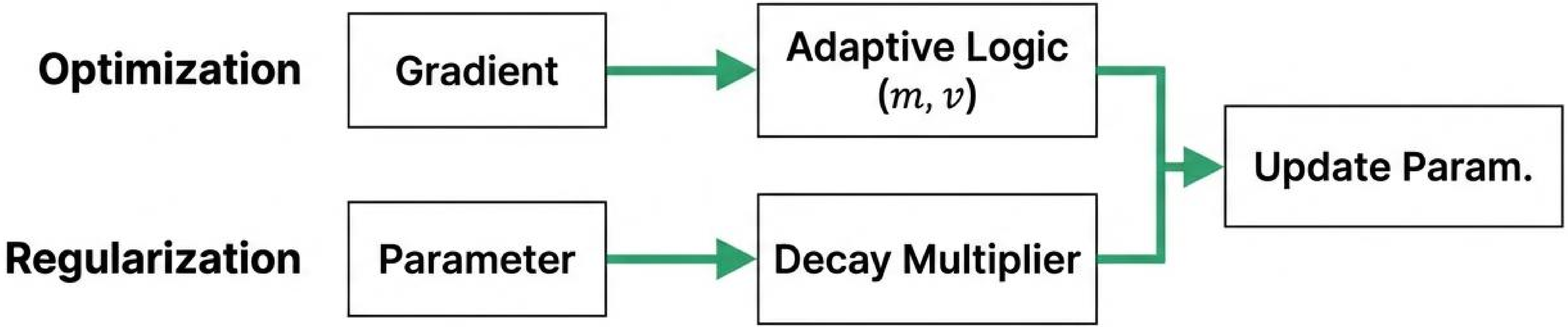
Why standard Adam fails at Weight Decay



- L2 Regularization (Decay) is meant to be a constant pressure.
- In Adam, highly active parameters (large v) accidentally shrink the decay penalty.
- **Consequence:** Poor generalization compared to SGD.

The Solution: AdamW

Decoupled Weight Decay



Separate the **optimization step** from the **regularization** step.
Apply decay *after* the adaptive update, directly to the weights.

Implementing AdamW

Decoupled Weight Decay Implementation

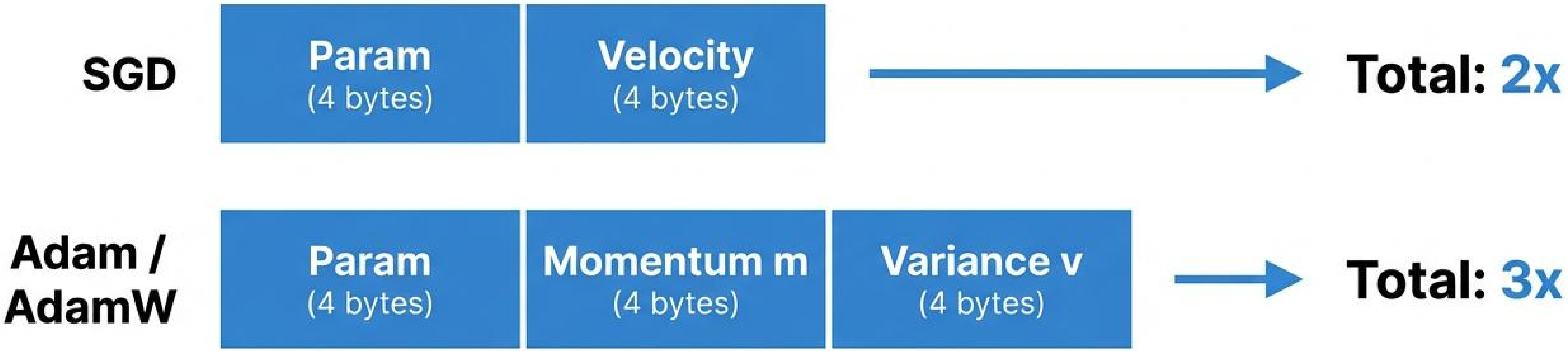
```
# 1. Apply gradient-based update (Standard Adam logic)
param.data = param.data - self.lr * m_hat / (np.sqrt(v_hat) + self.eps)
```

```
# 2. Apply decoupled weight decay (The Fix)
if self.weight_decay != 0:
    # Shrink parameter directly
    param.data = param.data * (1 - self.lr * self.weight_decay)
```

This **simple decoupling** is why Transformers train successfully.

The Memory Tax

The cost of intelligence



10B Parameter Model:

SGD State = 80GB
Adam State = 120GB **(+50% Increase)**

The Trade-off Matrix

Algorithm	Memory	Convergence	Generalization
SGD	Low (2x)	⚠️ Slow / Risky	Good
SGD + Momentum	Low (2x)	Fast	Good
Adam	High (3x)	Very Fast	⚠️ Poor (due to bug)
AdamW	High (3x)	Very Fast	✅ Excellent (Standard for LLMs)

TinyTorch → PyTorch

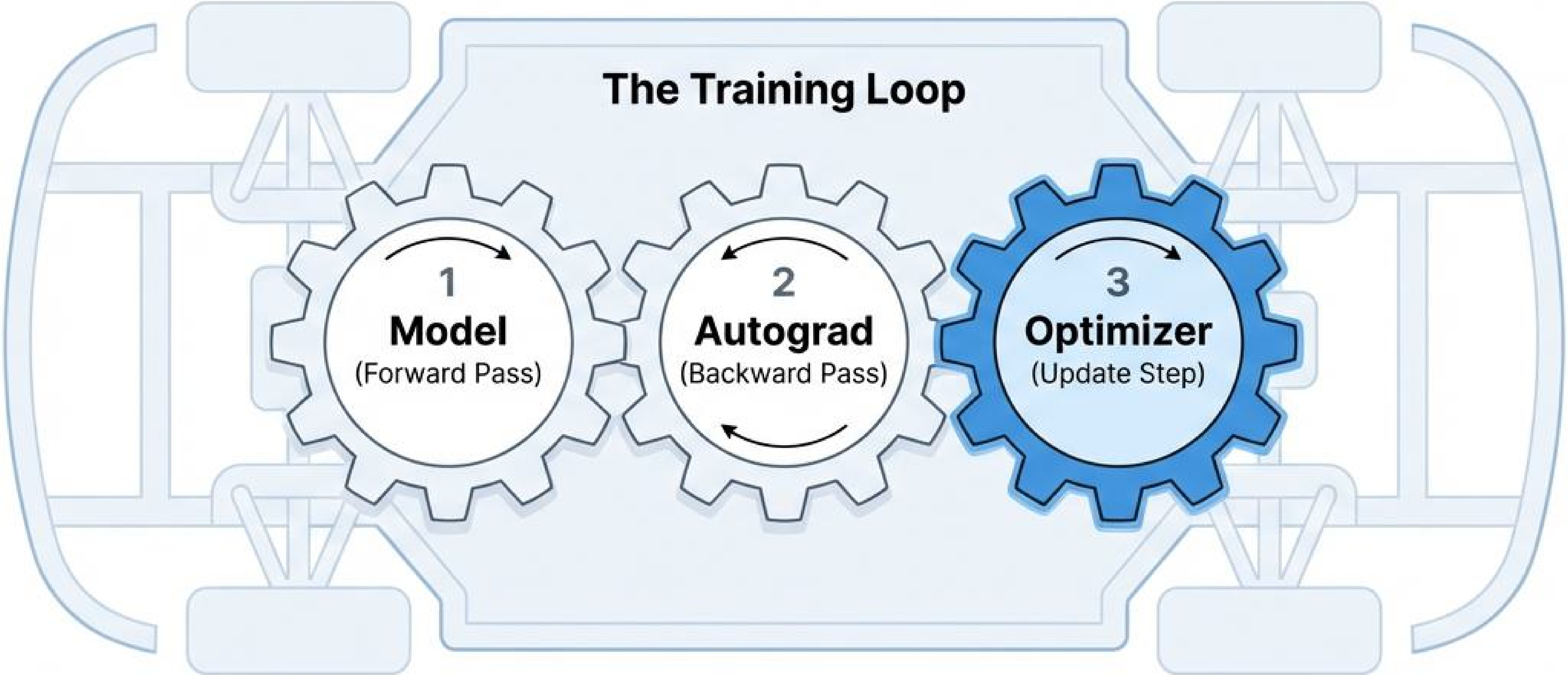
Identical API. Identical Math.

tinytorch	pytorch

`from tinytorch.core.optimizers import Adam`
 	
`# Initialize`
`optimizer = Adam(model.parameters(),`
`  lr=1e-3)`
 	
`# Training Step`
`loss.backward()`
`optimizer.step()`
`optimizer.zero_grad()`
	`import torch.optim as optim`
	
	`# Initialize`
	`optimizer = optim.Adam(model.parameters(),`
	`   lr=1e-3)`
	
	`# Training Step`
	`loss.backward()`
	`optimizer.step()`
	`optimizer.zero_grad()`

What's Next: Module 08

Building the Training Loop



We have the engine. Next, we build the car.