FOUNDATION TIER

MODULE 07

Optimizers

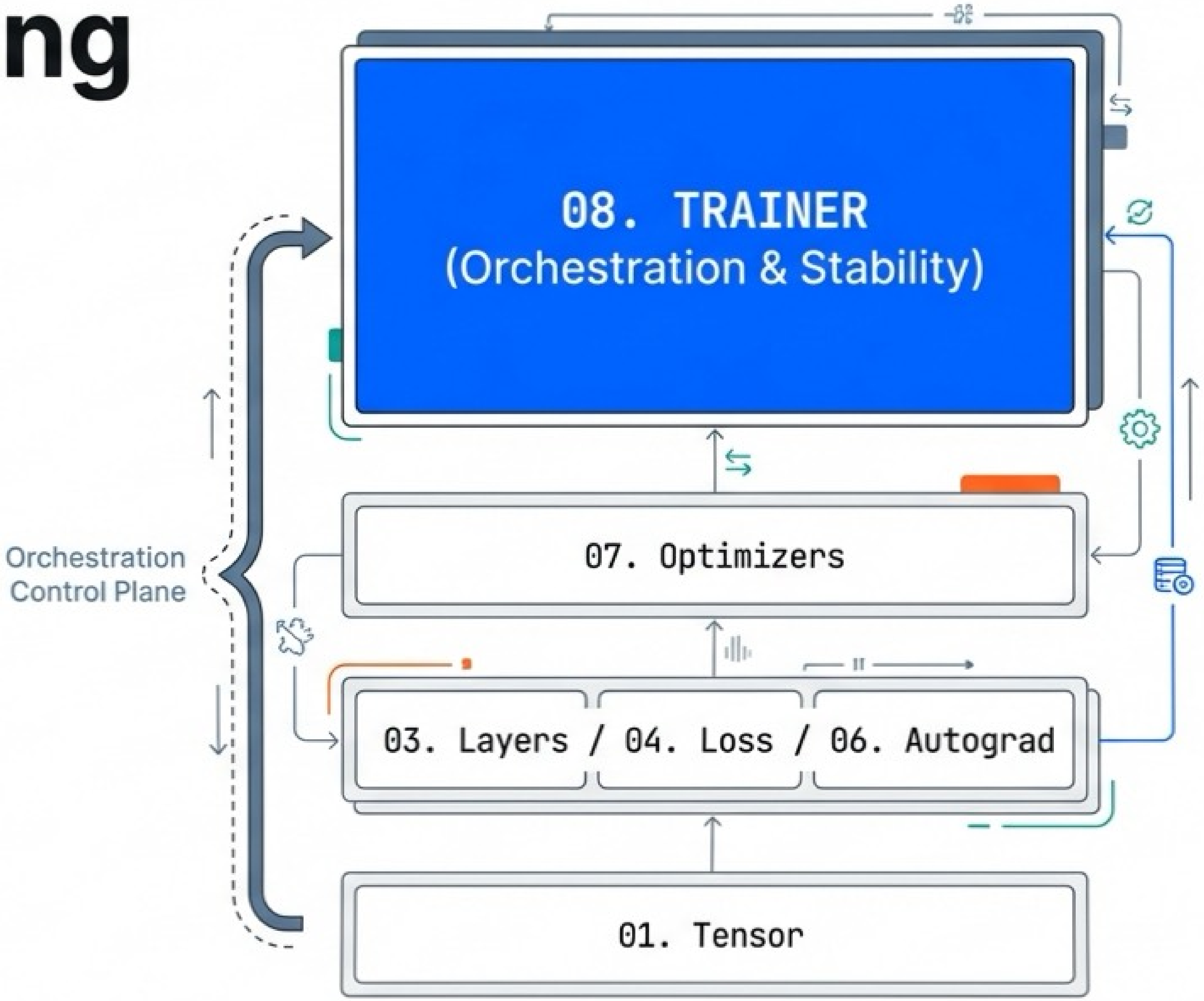
The engines of learning that update model parameters

Module 08: Training Infrastructure

Foundation Tier

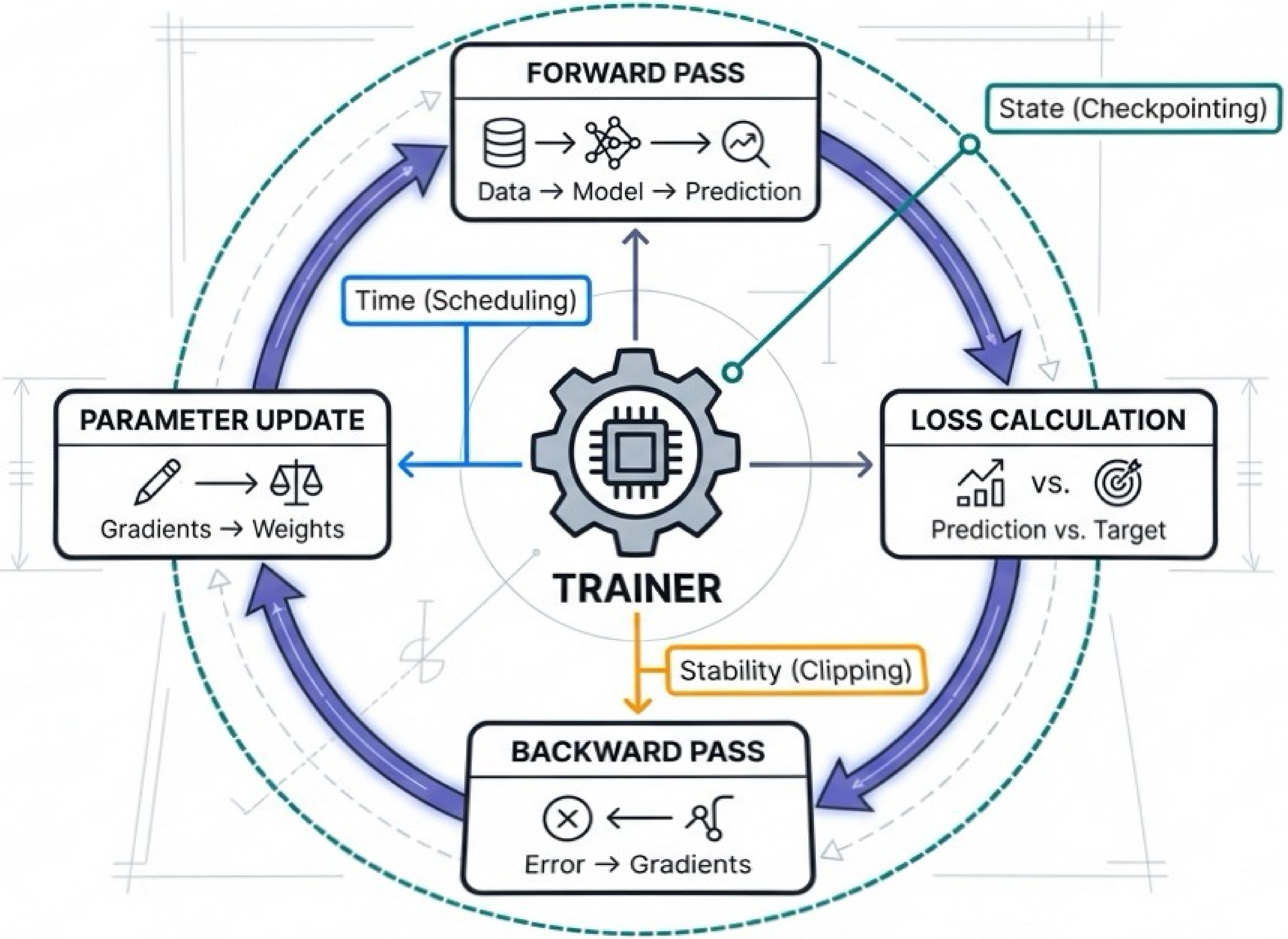
Prerequisites: Modules 01–07 (Tensor to Optimizer)

Components do not coordinate themselves. We are moving from component design to systems orchestration.



From Components to Symphony

The Lifecycle of a Batch



Why "Simple" Loops Fail in Production

in Inter Tight

The Naive Loop in Inter Tight

```
for batch in data:
    loss = model(batch)
    loss.backward()
    optimizer.step()
```

Fragile & Limited

Production Reality

Convergence in Inter Tight
Fixed learning rates are too slow to start, too volatile to finish.

Stability in Inter Tight
Gradients can explode (NaN), destroying progress.

Memory in Inter Tight
Physical batch size is capped by VRAM.

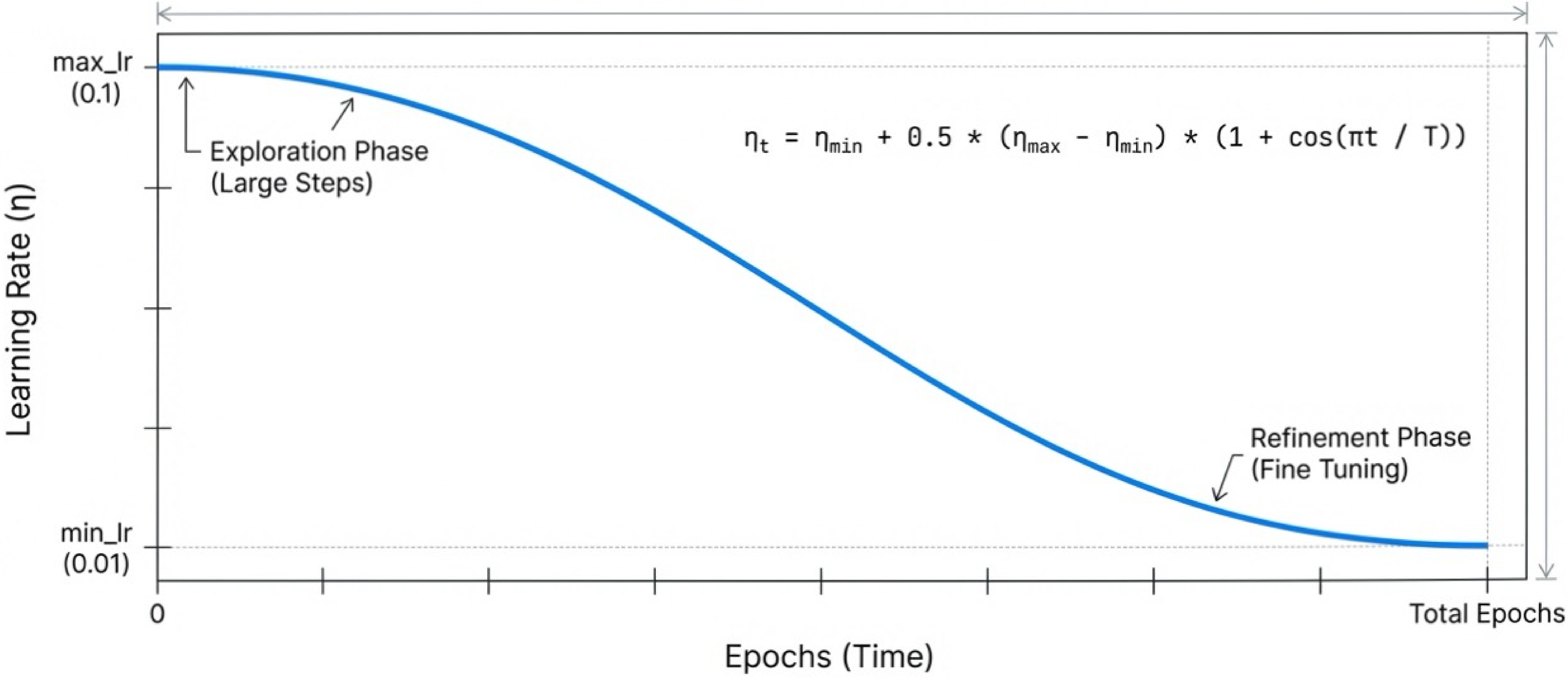
Fault Tolerance in Inter Tight
No persistence means complete data loss on crash.

The Solution: We build specific subsystems—Scheduler, Clipper, Checkpointer—before the Trainer.

Adaptive Training Dynamics

Concept: Cosine Annealing

Invariant: High LR for exploration; Low LR for convergence.



Implementation: CosineSchedule

tinytorch/core/training.py

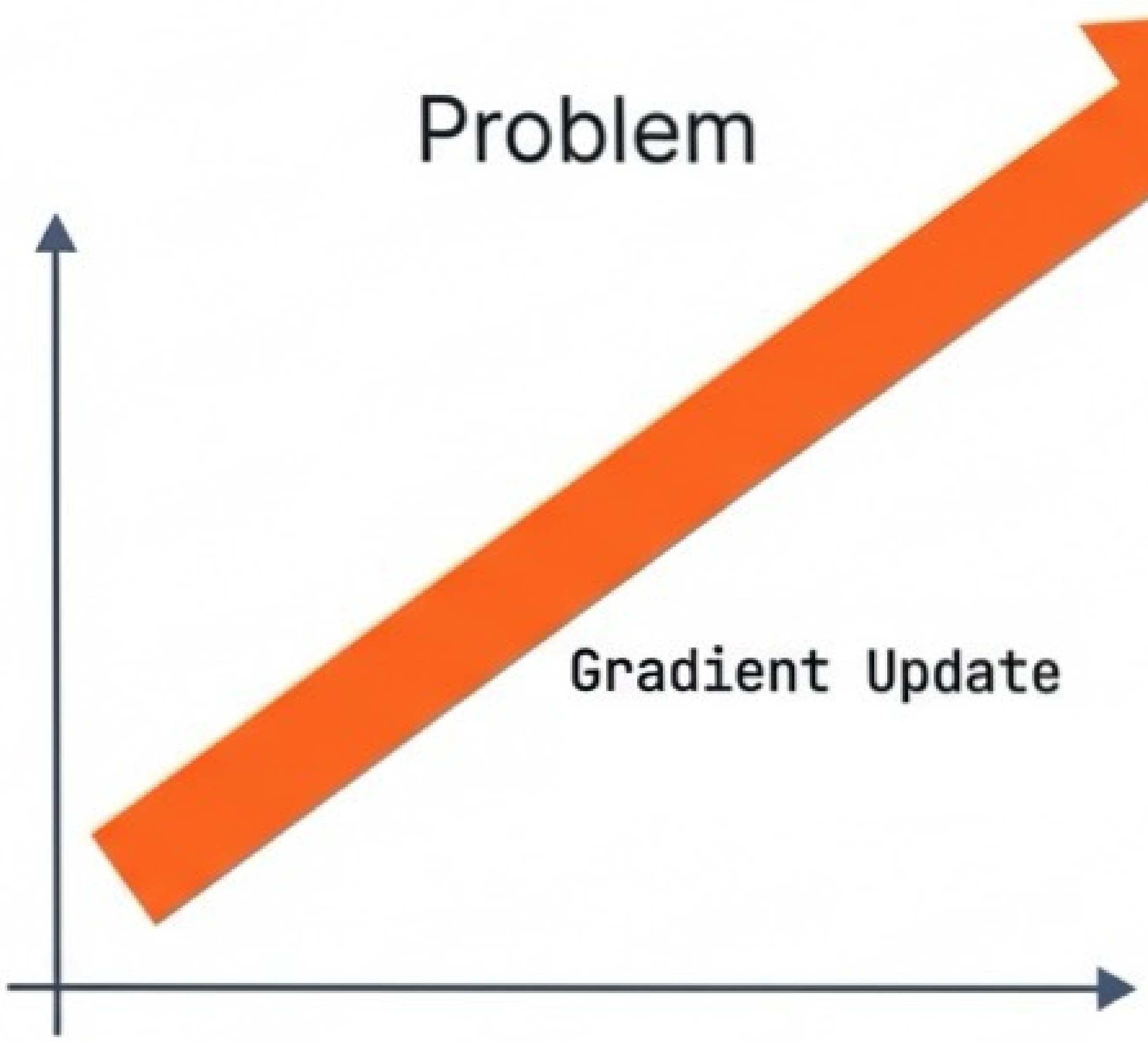
```
class CosineSchedule:
    def __init__(self, max_lr, min_lr, total_epochs):
        self.max_lr = max_lr
        self.min_lr = min_lr
        self.total_epochs = total_epochs

    def get_lr(self, epoch: int) -> float:
        # Boundary condition
        if epoch >= self.total_epochs:
            return self.min_lr

        # Cosine annealing formula
        cosine_factor = (1 + np.cos(np.pi * epoch / self.total_epochs)) / 2
        return self.min_lr + (self.max_lr - self.min_lr) * cosine_factor
```

Stateless Logic

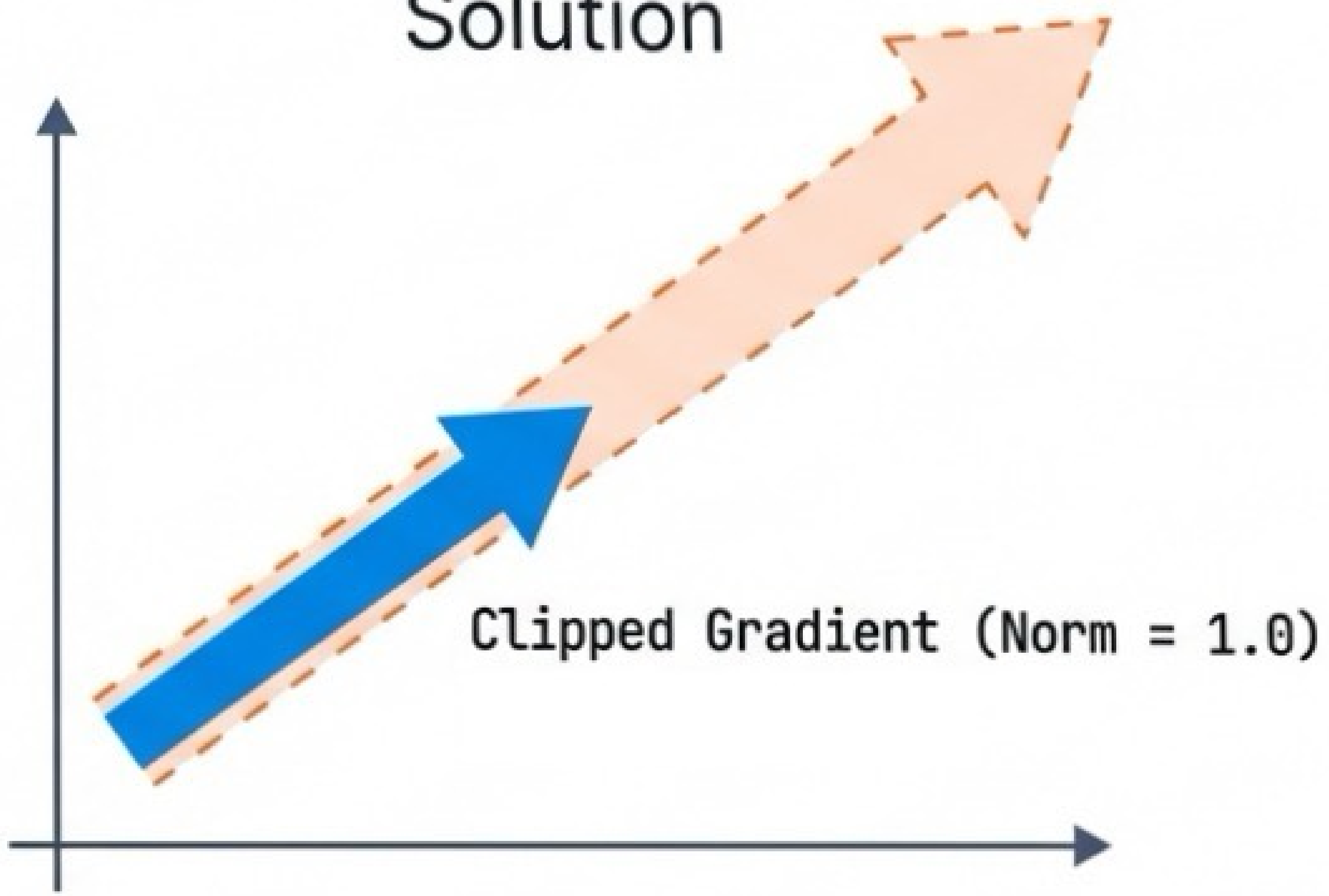
This class is a pure function of time. It does not access the model or optimizer directly.

The Exploding Gradient Problem

Global Norm Clipping



Result: Parameters hit Infinity or NaN.



Mathematical Logic:

1. Measure Total Norm (L2) of all params.
2. If Norm > Threshold: Scale by (Threshold / Norm).
3. Direction is preserved. Magnitude is safe.

Implementation: Gradient Clipping

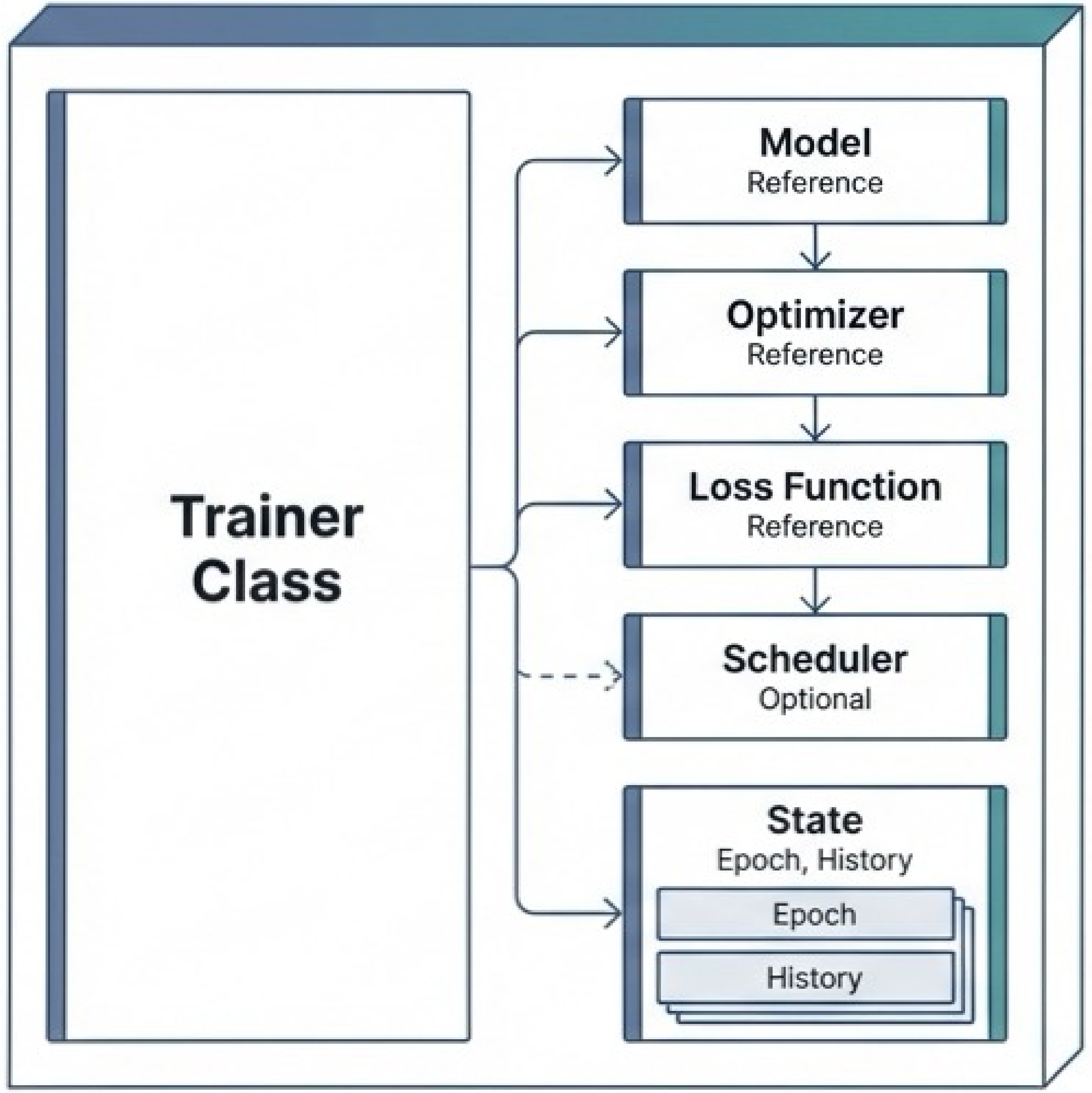
tinytorch/core/training.py

```python
def clip_grad_norm(parameters: List, max_norm: float = 1.0) -> float:
    # 1. Compute global norm across all parameters
    total_norm = 0.0
    for param in parameters:
        if param.grad is not None:
            # Access raw data to avoid graph overhead
            grad_data = param.grad.data if hasattr(param.grad, 'data') else param.grad
            total_nom += np.sum(grad_data ** 2)
    total_norm = np.sqrt(total_norm)

    # 2. Scale uniformly if norm exceeds threshold
    if total_norm > max_norm:
        clip_coef = max_norm / total_norm
        for param in parameters:
            if param.grad is not None:
                # Modify gradients in-place
                if hasattr(param.grad, 'data'):
                    param.grad.data *= clip_coef
                else:
                    param.grad *= clip_coef
    return float(total_norm)
```

In-place modification preserves memory.

The Trainer Abstraction

Encapsulating the Lifecycle



API Signature

```
class Trainer:
    def __init__(self, model,
                       optimizer,
                       loss_fn,
                       scheduler=None,
                       grad_clip_norm=None):

        self.model = model
        self.optimizer = optimizer
        self.loss_fn = loss_fn
        self.scheduler = scheduler
        self.grad_clip_norm = grad_clip_norm

        # State tracking
        self.epoch = 0
        self.history = {'train_loss': [], 'eval_loss': []}
```

The Training Loop: Forward & Accumulate

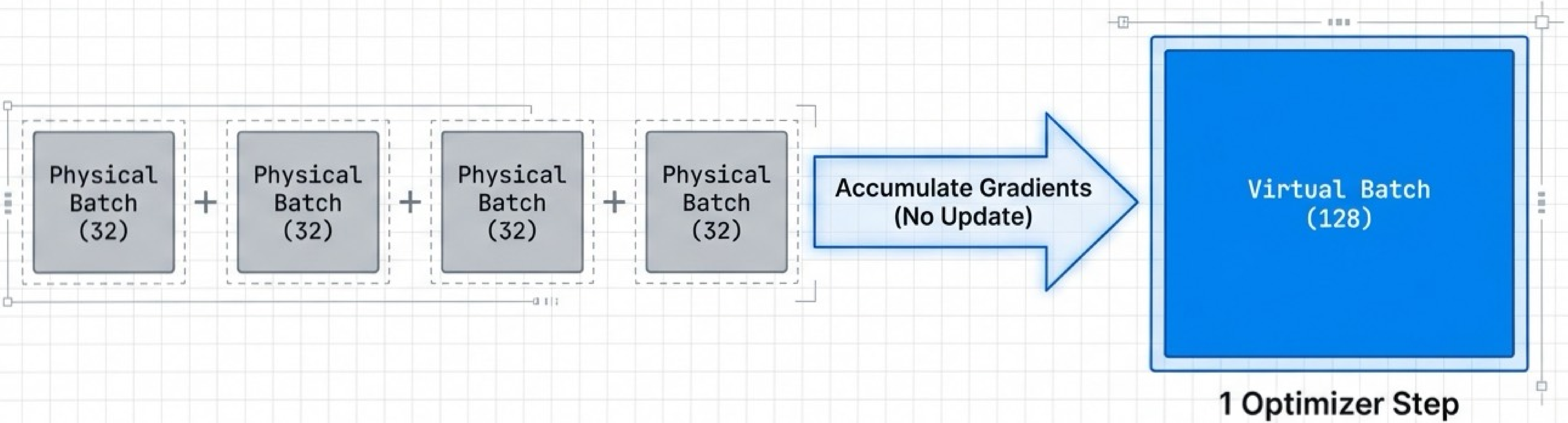
Inside **Trainer.train_epoch()**

```
def train_epoch(self, dataloader, accumulation_steps=1):
    self.model.training = True   # Enable Dropout/BatchNorm

    for batch_idx, (inputs, targets) in enumerate(dataloader):
        # 1. Forward pass
        outputs = self.model.forward(inputs)
        loss = self.loss_fn.forward(outputs, targets)

        # 2. Scale loss for accumulation
        # We divide by N so the sum of N gradients equals the mean
        scaled_loss = loss.data / accumulation_steps
        accumulated_loss += scaled_loss

        # 3. Backward pass (accumulates into .grad)
        loss.backward()
```

Accumulates gradients.
Does NOT overwrite.

System Insight: Gradient Accumulation

Decoupling Batch Size from Memory



- **Problem:** GPU Memory limits physical batch size.
- **Solution:** Accumulate gradients over N steps before updating.
- **Trade-off:** Saves Memory vs. Increases Time.

The Training Loop: Update & Schedule

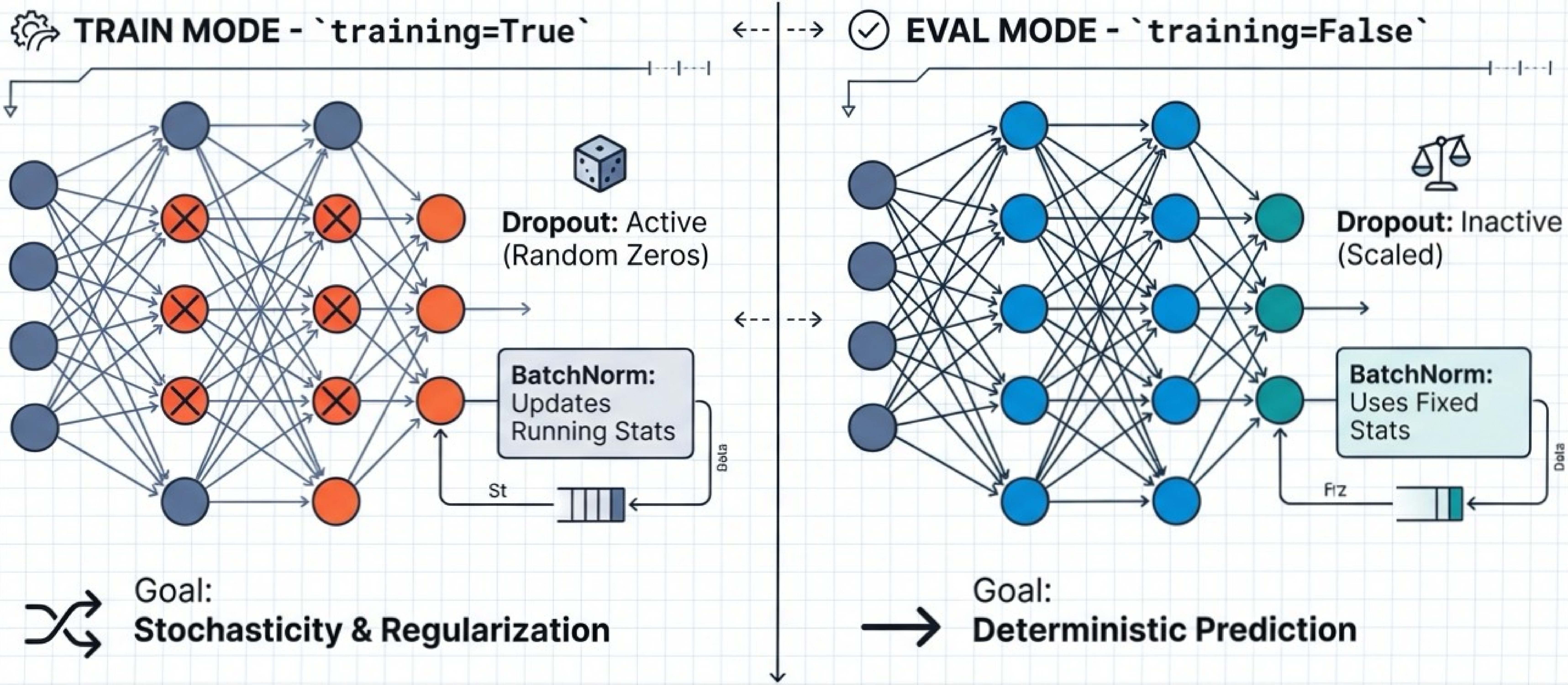
Inside **Trainer.train_epoch()**

```
# Only update every 'accumulation_steps'
if (batch_idx + 1) % accumulation_steps == 0:

    # 4. Gradient Clipping (Safety)
    if self.grad_clip_norm is not None:
        clip_grad_norm(self.model.parameters(), self.grad_clip_norm)

    # 5. Optimizer Step (Update)
    self.optimizer.step()
    self.optimizer.zero_grad()  # CRITICAL: Clear buffers

# End of epoch: Scheduler Update
if self.scheduler is not None:
    self.optimizer.lr = self.scheduler.get_lr(self.epoch)
    self.epoch += 1
```

Must clear buffers after step, or gradients will double.

Mode Switching: Train vs. Eval

Why models have two personalities



Implementation: Evaluate in Inter Tight

Observing without Learning

```
def evaluate(self, dataloader):
    self.model.training = False  # DISABLE Dropout/Update BN stats
    self.training_mode = False

    total_loss = 0.0
    for inputs, targets in dataloader:
        # Forward pass only
        outputs = self.model.forward(inputs)
        loss = self.loss_fn.forward(outputs, targets)

        # Metrics Only
        # NO backward()           # NO optimizer.step()
        total_loss += loss.data

    return total_loss / len(dataloader)
```

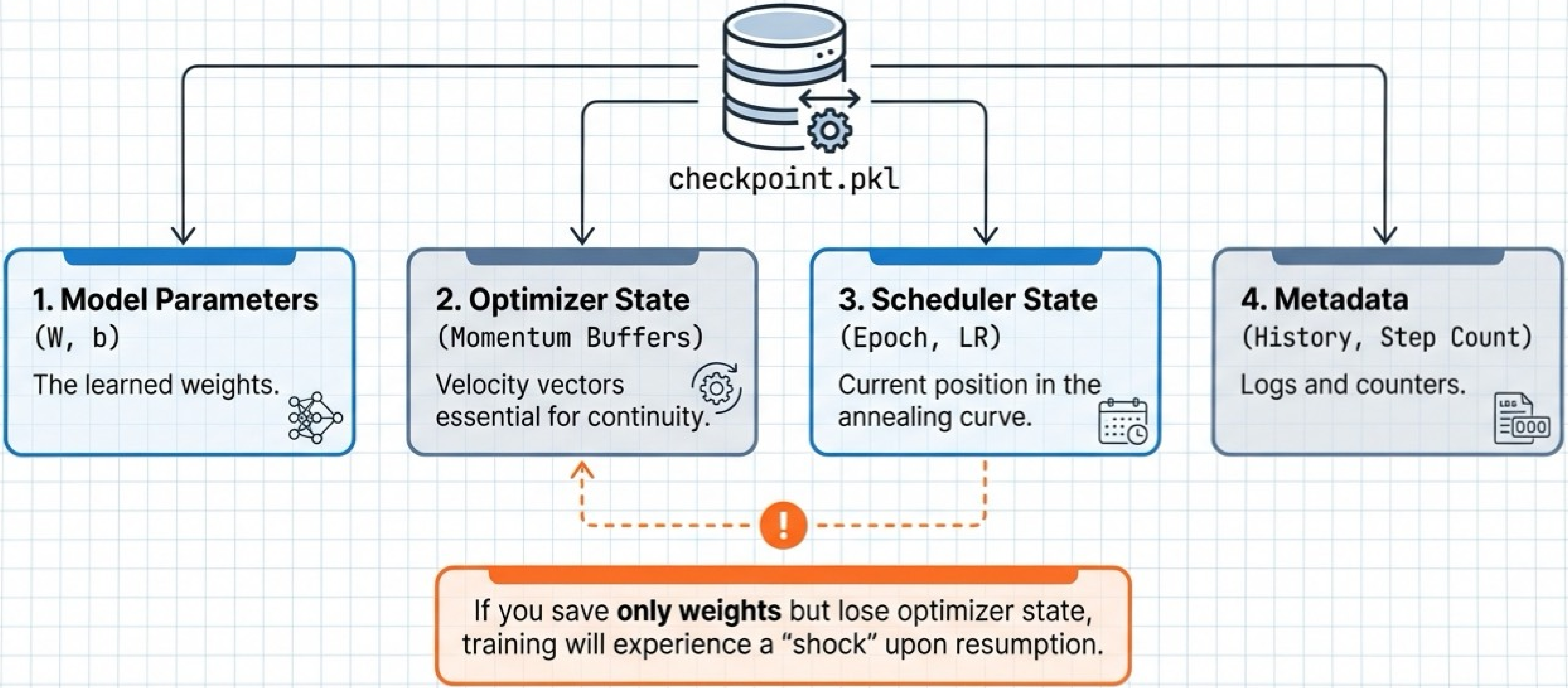
Critical Action

Skipping backward() & step() avoids training.

By skipping backward() and step(), we reduce computational cost and ensure we don't train on test data.

Persistence and Fault Tolerance

Anatomy of a Checkpoint



Implementation: Checkpointing

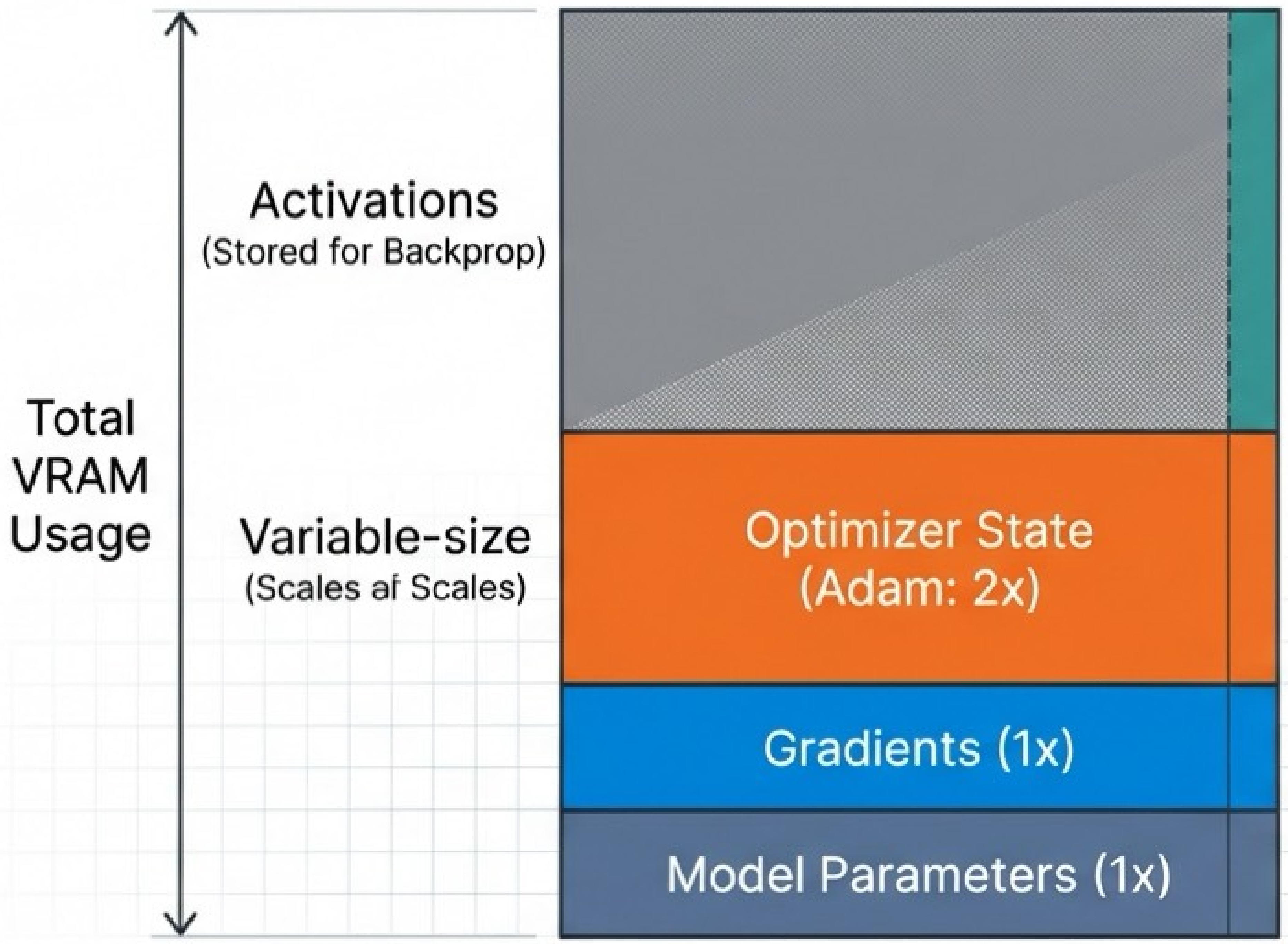
tinytorch/core/training.py

```
def save_checkpoint(self, path: str):
    checkpoint = {
        'epoch': self.epoch,
        'step': self.step,
        'model_state': self._get_model_state(),         # Weights
        'optimizer_state': self._get_optimizer_state(), # Momentum Buffers
        'scheduler_state': self._get_scheduler_state(), # Current LR info
        'history': self.history
    }

    with open(path, 'wb') as f:
        pickle.dump(checkpoint, f)
```

We use helper methods like `_get_optimizer_state` to extract internal buffers from SGD or Adam instances.

The Cost of Training: Memory Hierarchy

Why Training Needs 4-6x More RAM Than Inference



- **Implication:** A 1GB Model requires ~5GB+ VRAM to train.
- **Bottleneck:** Activations scale with Batch Size.

TinyTorch vs. The World

Concepts are identical; Scale is different.

Feature	TinyTorch	PyTorch Lightning / HF
Loop	Explicit “`for`” loop	Abstracted “`trainer.fit()`”
Scheduling	`CosineSchedule`	Pluggable Schedulers
Clipping	`clip_grad_norm`	`gradient_clip_val`
Precision	**Float32** Only	Mixed Precision (FP16/BF16)
Scale	Single Device	Distributed Multi-GPU

“You have built the engine. Production frameworks just add the turbocharger.”

The Complete Pipeline

Putting it all together

```
# 1. Setup Components (Modules 01-07)
model = MyModel()
optimizer = SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = CosineSchedule(max_lr=0.1, min_lr=0.01, total_epochs=100)

# 2. Inject into Trainer (Module 08)
trainer = Trainer(model, optimizer, MSELoss(),
                  scheduler, grad_clip_norm=1.0)

# 3. Execution
for epoch in range(100):
    train_loss = trainer.train_epoch(train_data)
    eval_loss, acc = trainer.evaluate(val_data)

    if epoch % 10 == 0:
        trainer.save_checkpoint(f'ckpt_{epoch}.pkl')
```

Setup

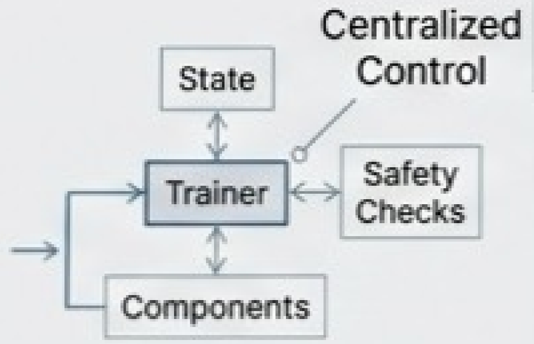
Trainer Injection

Execution Loop

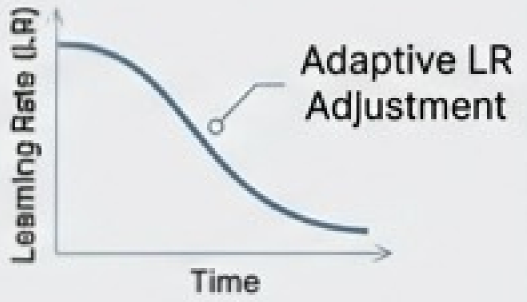
Summary & Next Steps

Takeaways

1. **Orchestration**
   The `Trainer` centralizes state and safety.

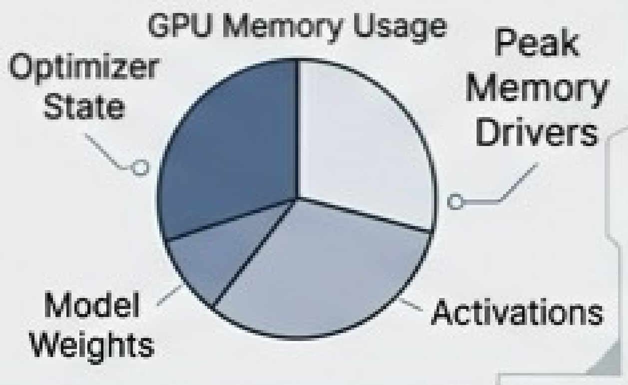


2. **Dynamics**
   Schedulers adapt learning rates for convergence.

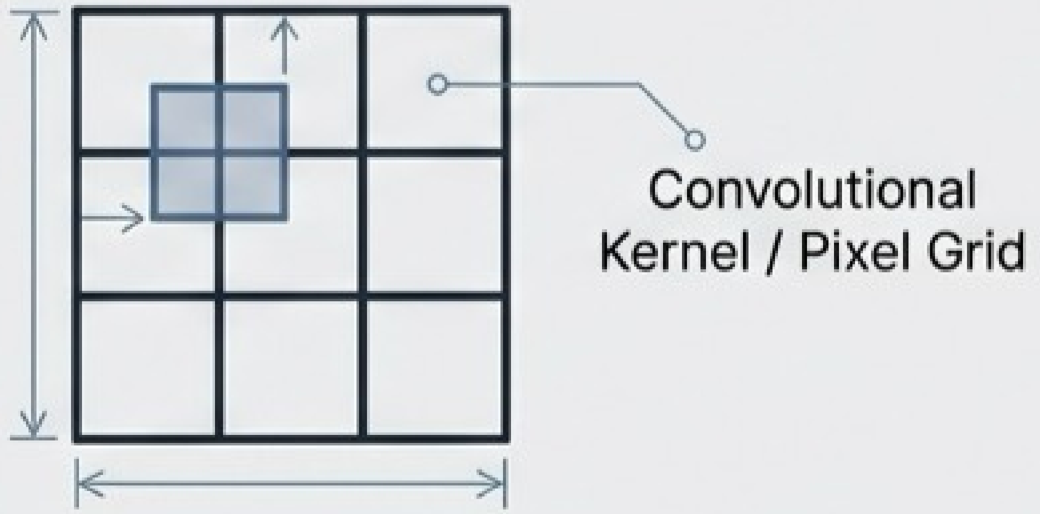


3. **Stability**
   Gradient clipping prevents numeric explosion.

4. **Reality**
   Memory is dominated by optimizer states and activations.



What's Next



Module 09: Architecture Tier

- Applying this infrastructure to Computer Vision. Vision Application
- Building Convolutions (CNNs). CNN Architecture

End of Foundation Tier.