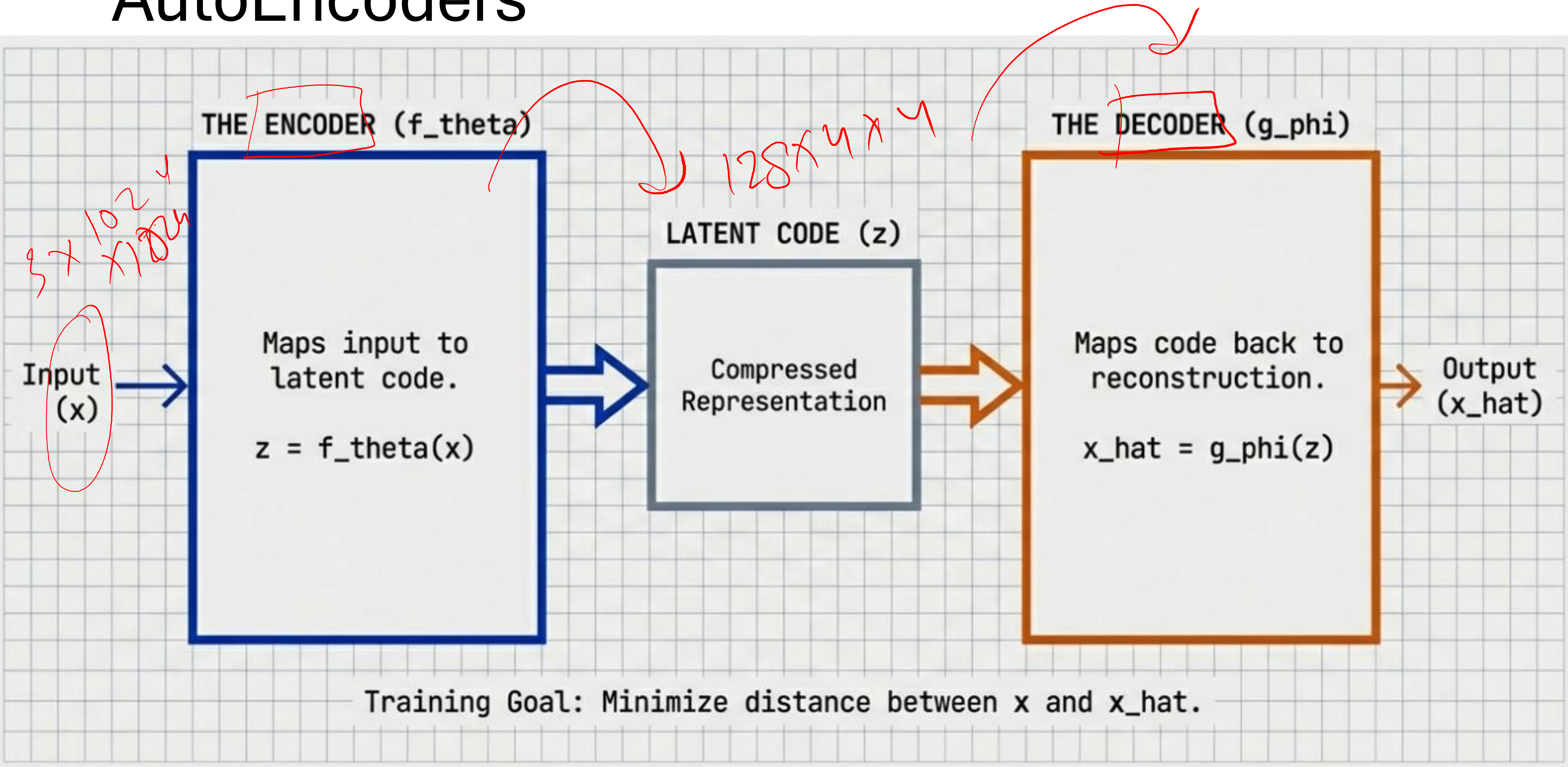
Deep Learning Frameworks

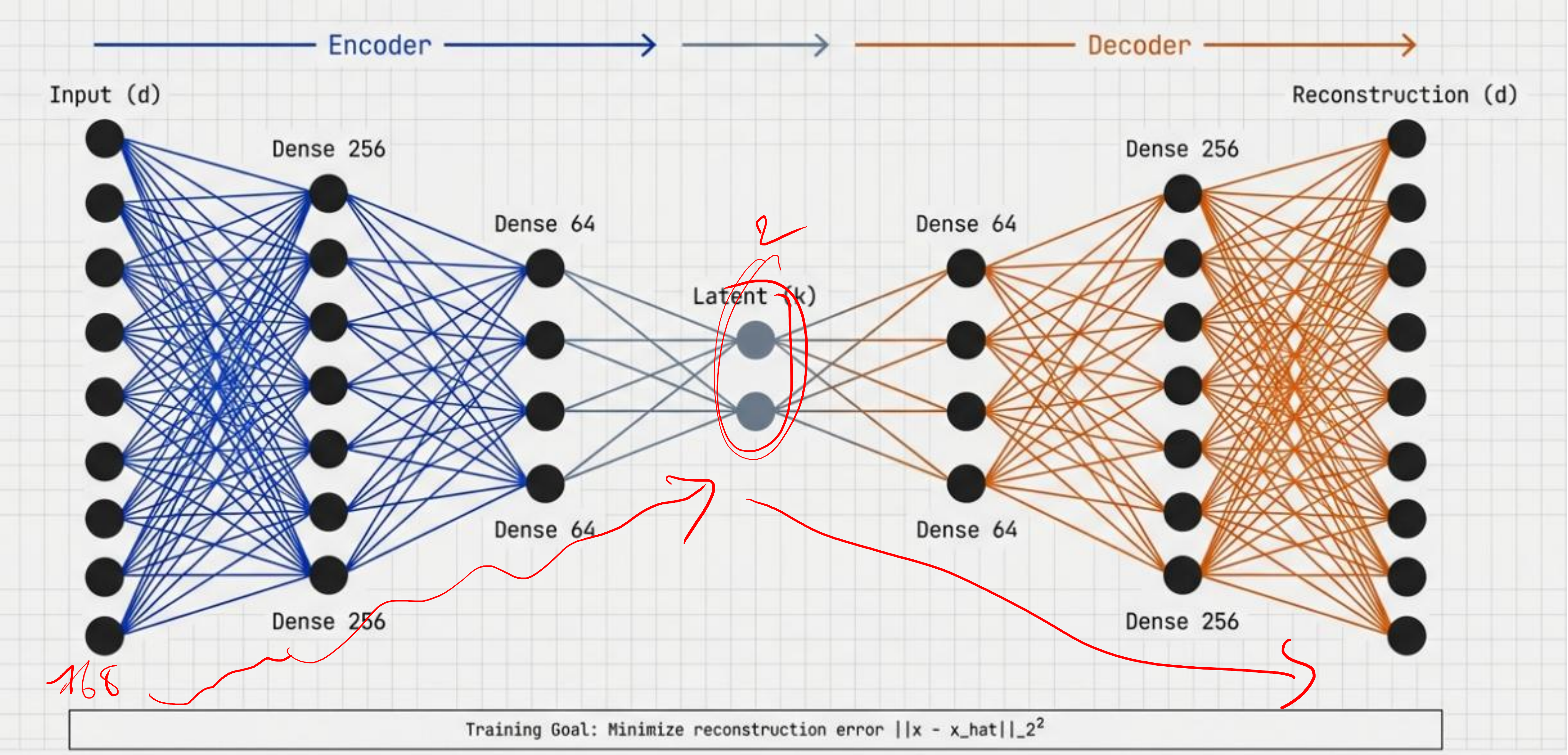
Autoencoders, Gradient Checkpointing, Optimizers

https://tinyurl.com/dlframeworks
https://github.com/sakharamg/DeepLearningFrameworks

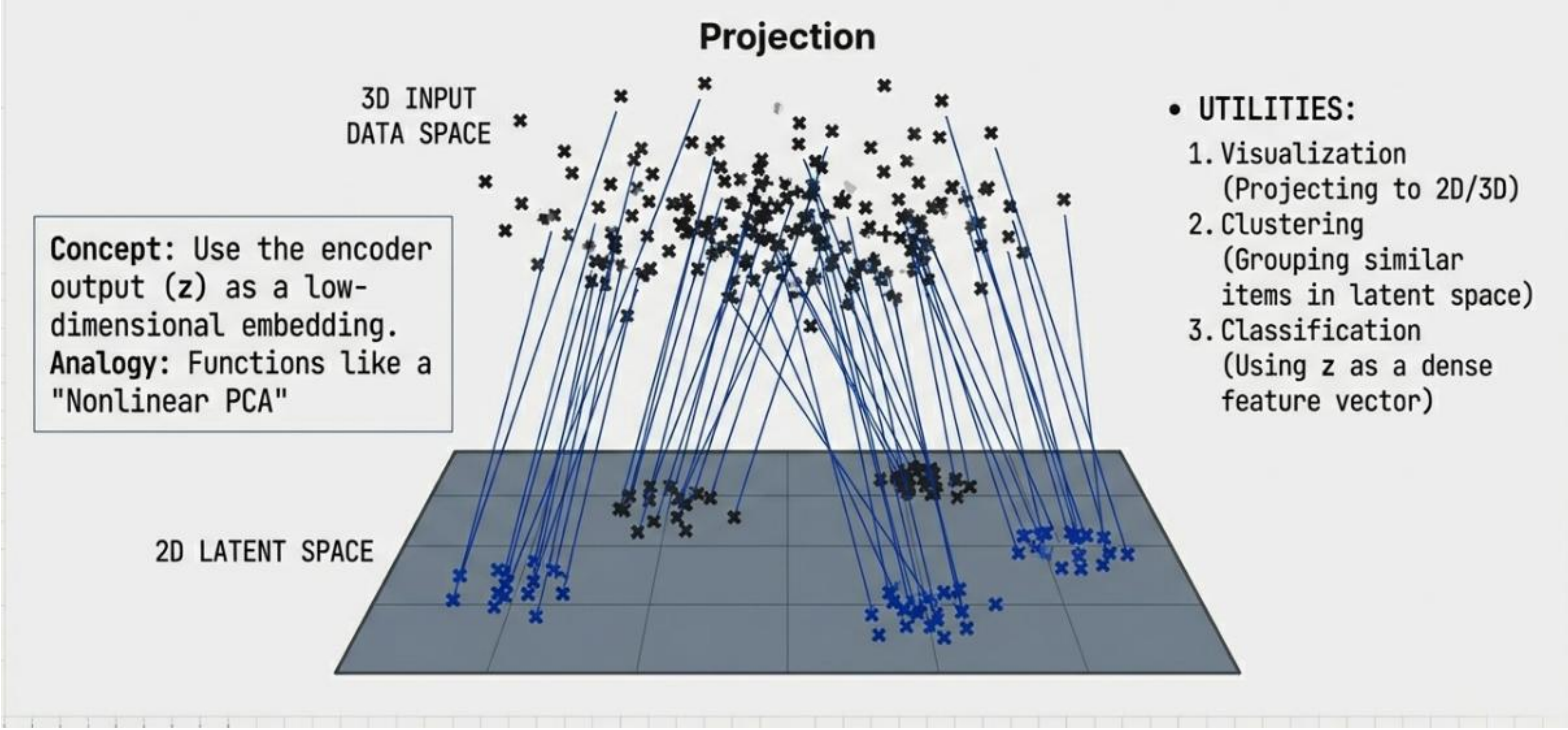
AutoEncoders



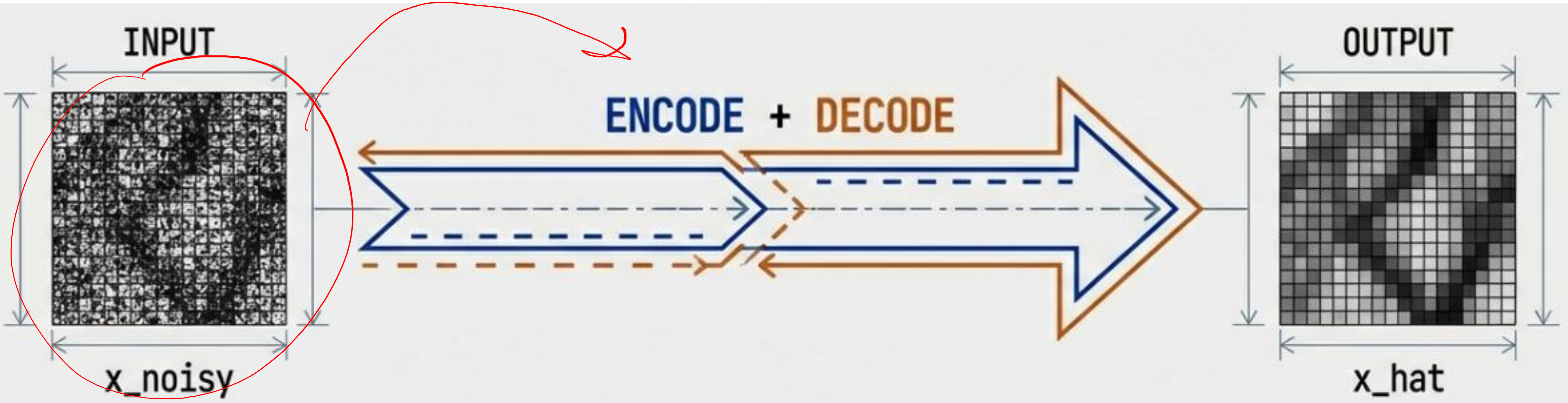
Architecture



Dimensionality Reduction



Denoising



The Modification: Train with a mismatch between Input and Target.

Input: Corrupted Data (x_noisy)

Target: Clean Data (x)

Loss Function: $L(x, g(f(x\_noisy)))$

Result: The network learns to map corrupted inputs back to the clean data manifold.

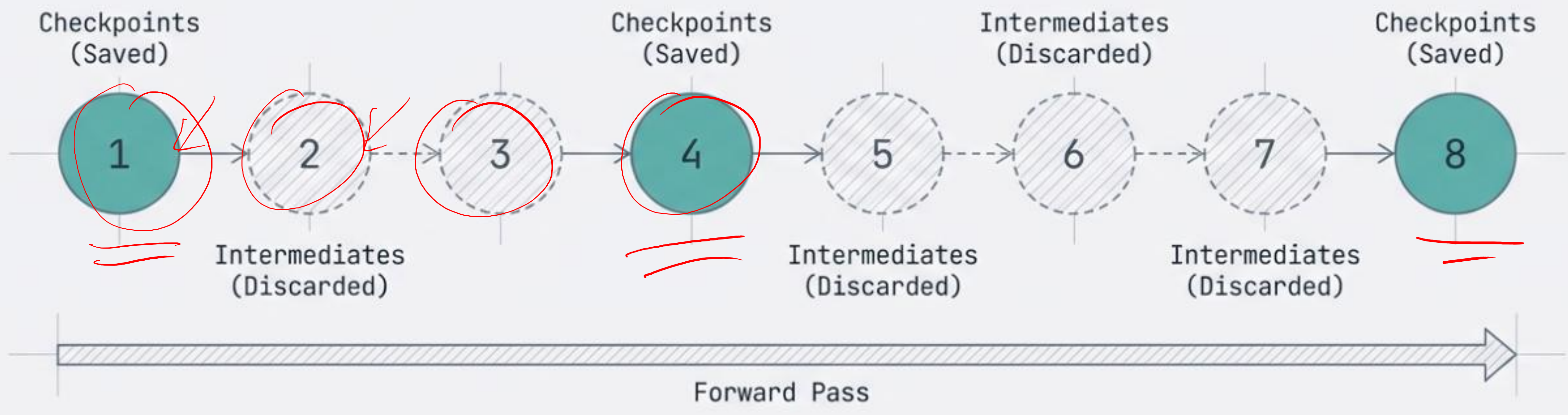
Lab

https://tinyurl.com/dlframeworks
https://github.com/sakharamg/DeepLearningFrameworks

Gradient Checkpointing

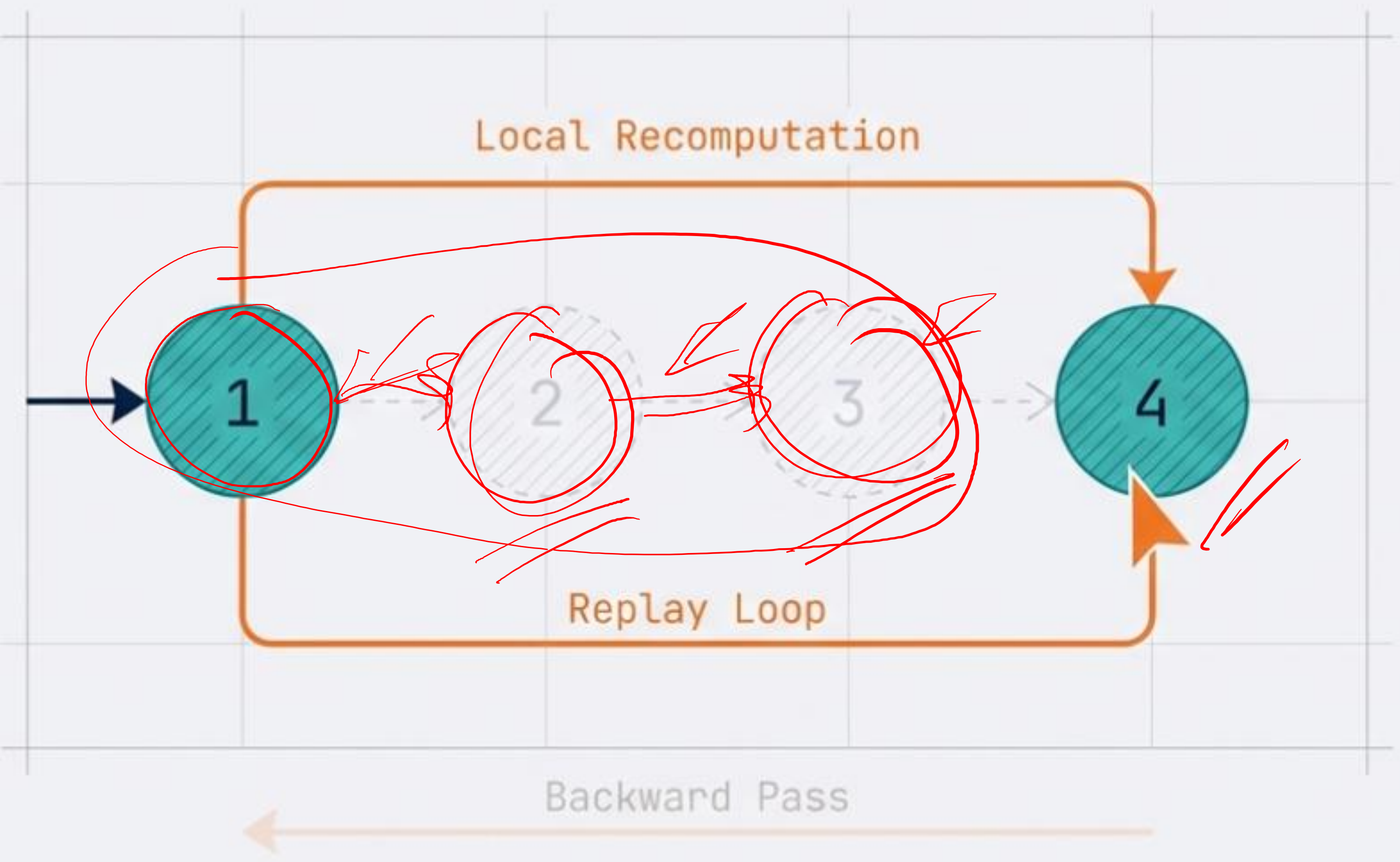
The Core Concept: A strategic decision to not store all intermediate activations. We prioritize memory availability by choosing to recompute specific values on demand during the backward pass.

Checkpoints and Discards



Mechanism: The system traverses the network, but only commits the "Checkpoint" nodes to long-term memory. All dashed nodes are computed, used for the next step, and immediately erased.

Backward Prop



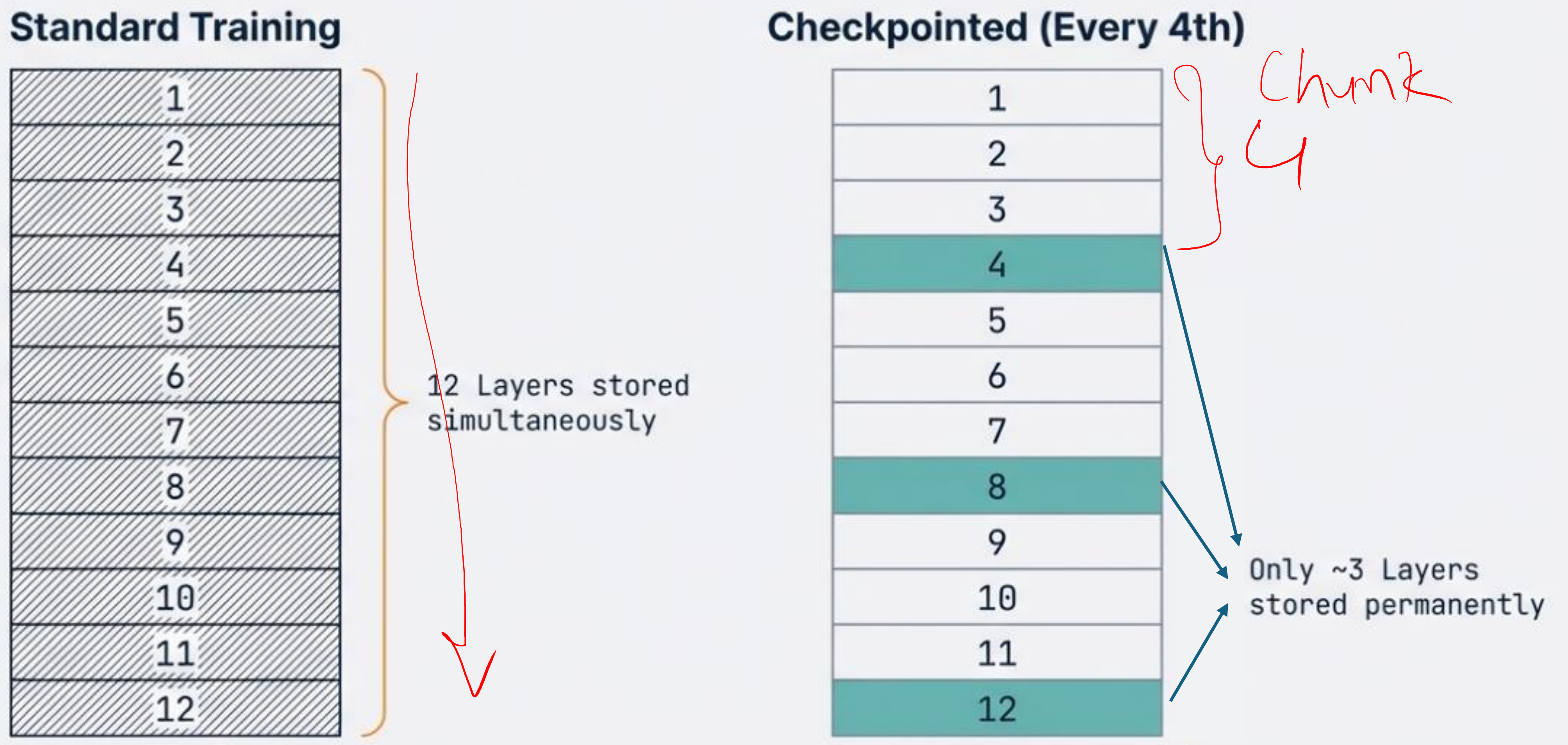
1. Pause: Backprop requires gradients for discarded Layer 3.

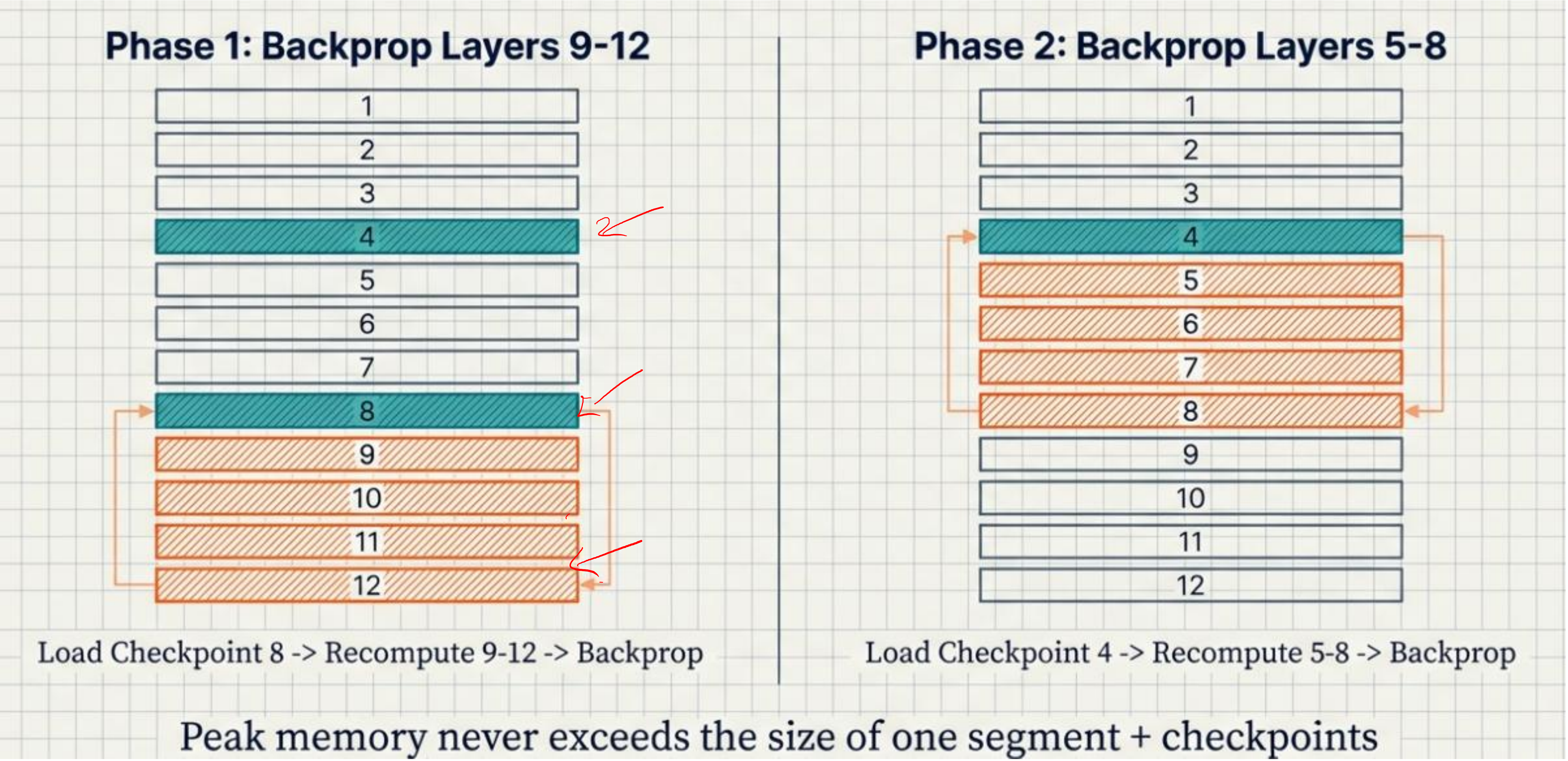
2. Rewind: System loads state from Checkpoint 1.

3. Replay: Forward pass runs again for Layers 2 & 3.

4. Discard: Data is used and immediately freed.

Visualizing the Memory Footprint



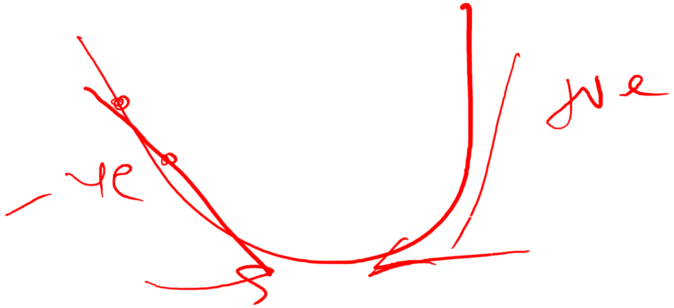

Phase 1: Backprop Layers 9-12
1
2
3
4
5
6
7
8
9
10
11
12
Load Checkpoint 8 -> Recompute 9-12 -> Backprop
Phase 2: Backprop Layers 5-8
1
2
3
4
5
6
7
8
9
10
11
12
Load Checkpoint 4 -> Recompute 5-8 -> Backprop
Peak memory never exceeds the size of one segment + checkpoints

Lab

https://tinyurl.com/dlframeworks

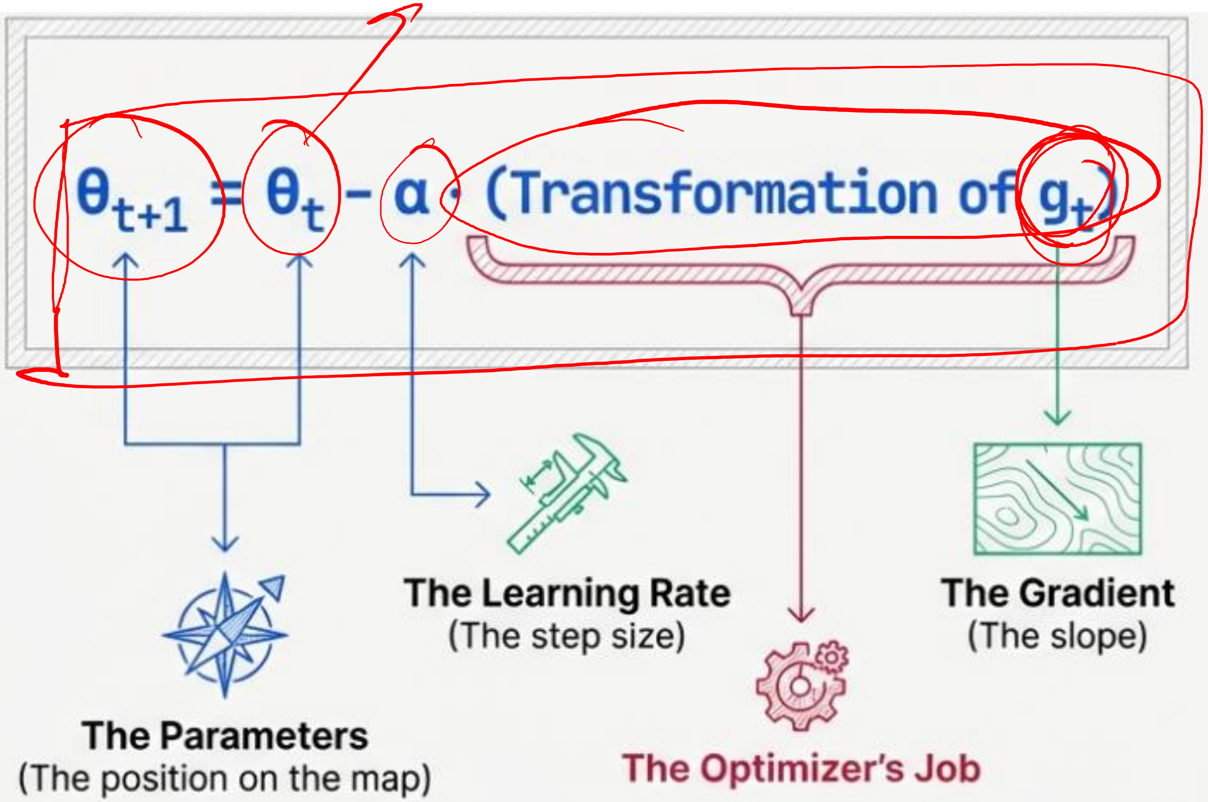
https://github.com/sakharamg/DeepLearningFrameworks

Optimizers



Deep learning optimization is fundamentally a search problem. We possess a set of parameters (θ) and a loss function (L). The gradient (g_t) acts as a compass, pointing in the direction of the steepest ascent—we want to go the opposite way.

Our objective is to minimize loss by iteratively updating these parameters.



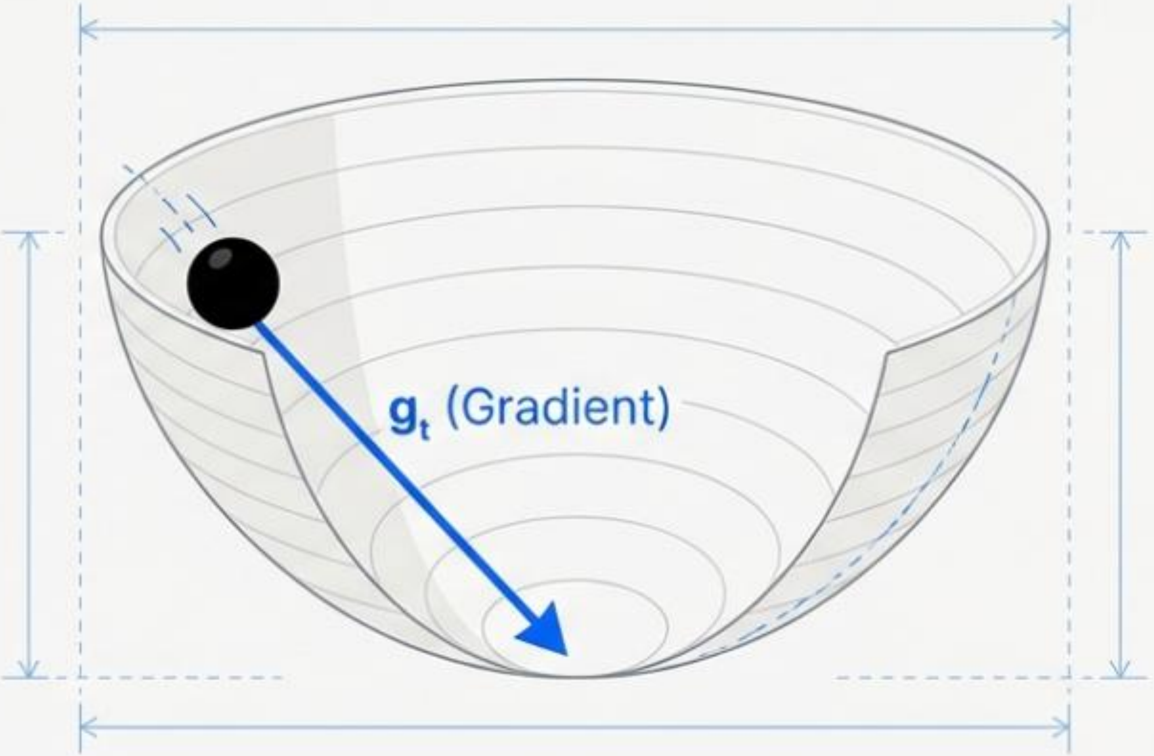
Stochastic Gradient Descen~~d~~t

MSE

Concept & Math

The concept is simple: take a small step directly opposite the gradient. There is no memory of past steps and no scaling of the future—just the raw slope.

$$\underbrace{\theta_{t+1}}_{\text{Updated Parameters}} = \underbrace{\theta_t}_{\text{Current Parameters}} - \underbrace{\alpha}_{\text{The Update Step}} \underbrace{g_t}_{\text{The Gradient}}$$



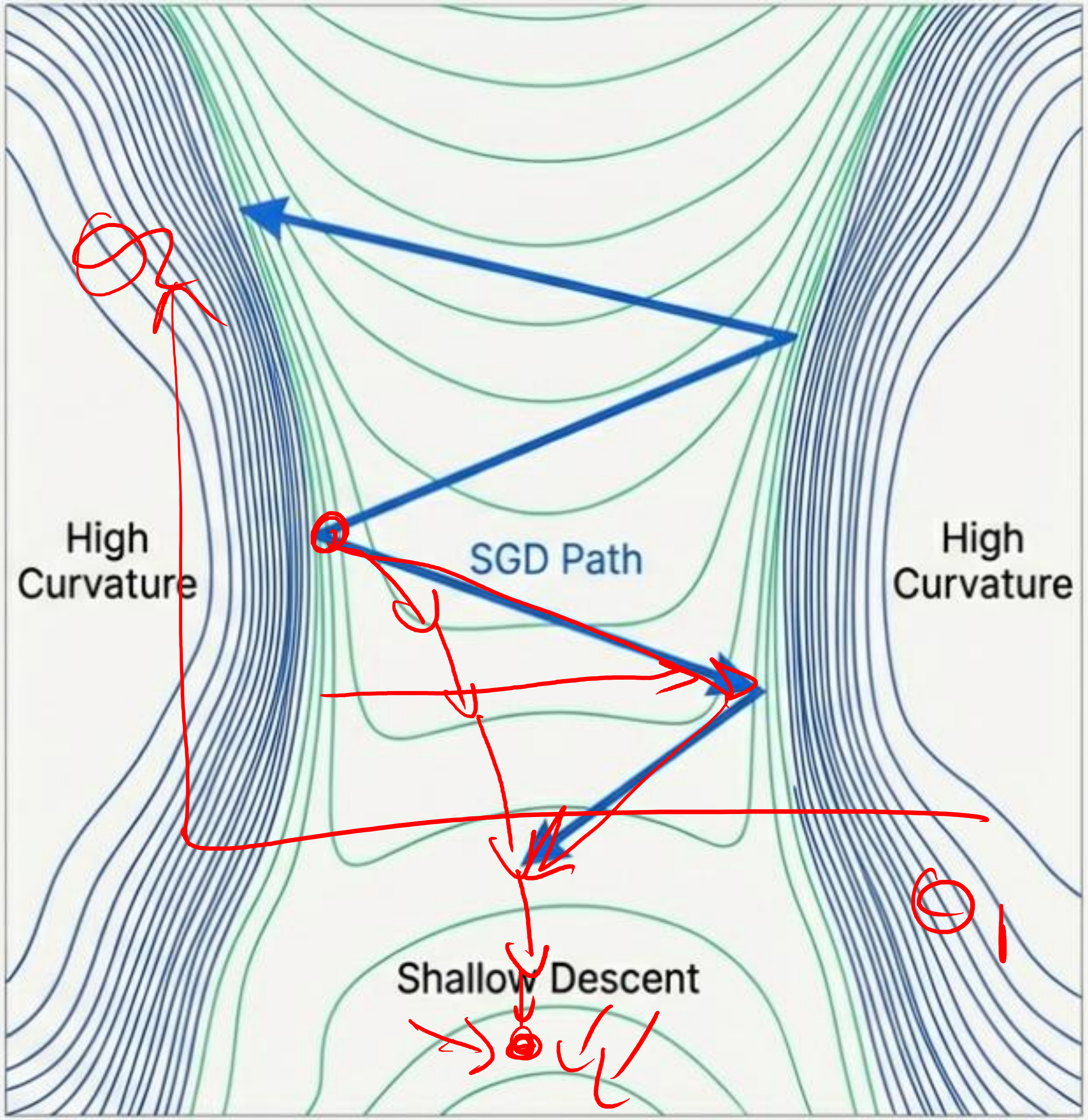
Key Insight: Simple, cheap, and capable. In mini-batch settings, the inherent noise helps escape sharp local minima.

The Failure Mode: The Ravine

Real loss landscapes aren't smooth bowls; they are often ravines—steep on the sides, shallow along the floor.

- **Zig-Zagging**: SGD oscillates in directions with high curvature.

- **Sensitivity**: Extremely sensitive to learning rate. Too high, it diverges; too low, it stalls.
- **Inefficiency**: Steps are wasted bouncing sideways rather than moving forward.

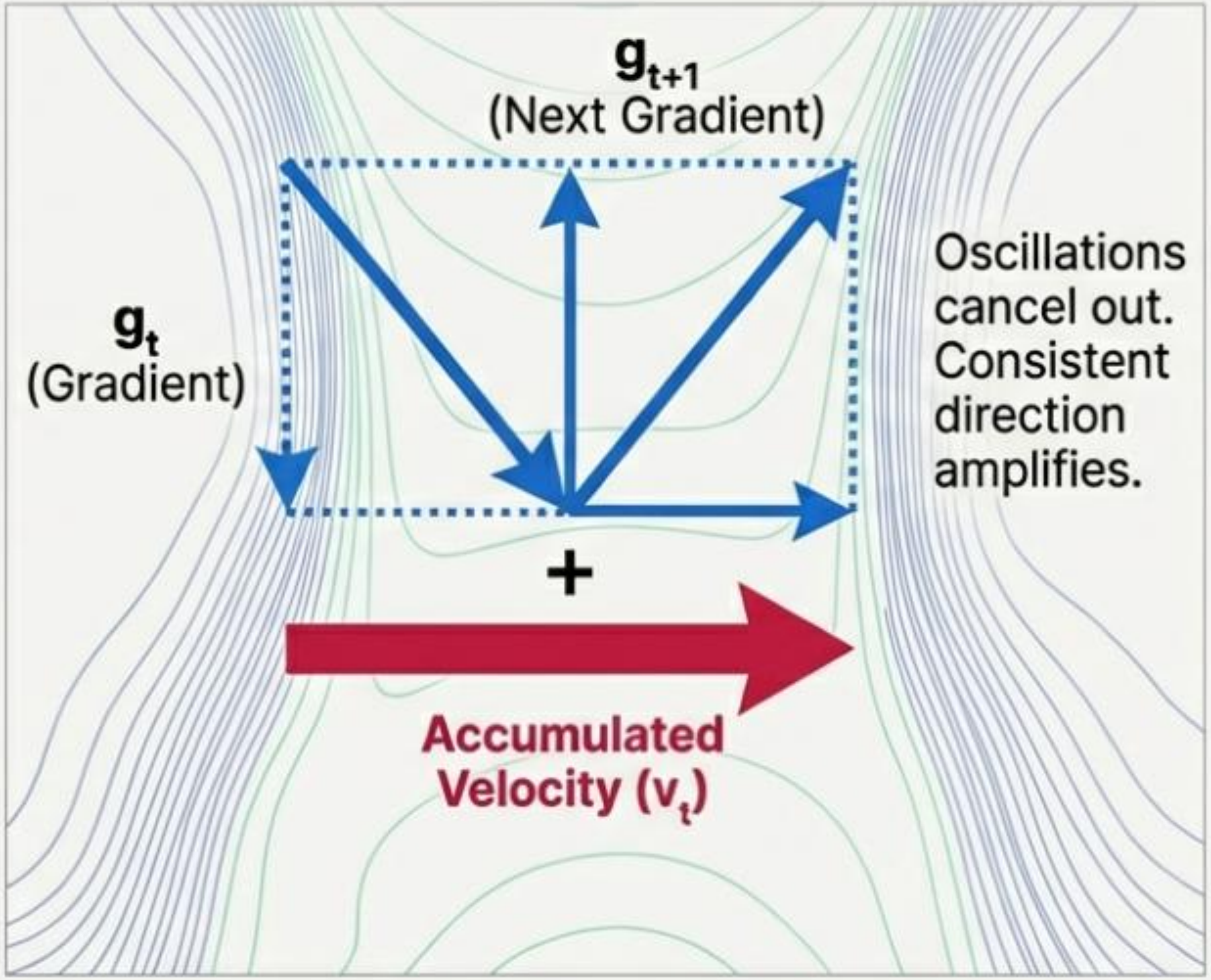


The Velocity Upgrade: Momentum

What if the optimizer had mass?

Intuition

To fix the zig-zag, we introduce physics.
If gradients flip signs (oscillate), they should cancel out. If they point in the same direction, they should accumulate speed.
We introduce a new variable: **Velocity** (v_t)



Momentum Equations

$0.9 \leftarrow \beta \, v_{t-1} + \underbrace{(1-\beta)}_{0.1} g_t$

$v_0 = 0$

$v_1 = \beta v_0 + g_1$

Velocity Update

Friction / Memory (typically 0.9)

$$v_t = \beta \, v_{t-1} + g_t$$

Previous Velocity

Current Gradient

Parameter Update

$$\theta_{t+1} = \theta_t - \alpha \, v_t$$

Step is now based on **Velocity**, not just Gradient

Note on Nesterov: A variant called Nesterov Momentum "peeks ahead" by calculating the gradient at the predicted future position, often yielding better results in computer vision tasks.

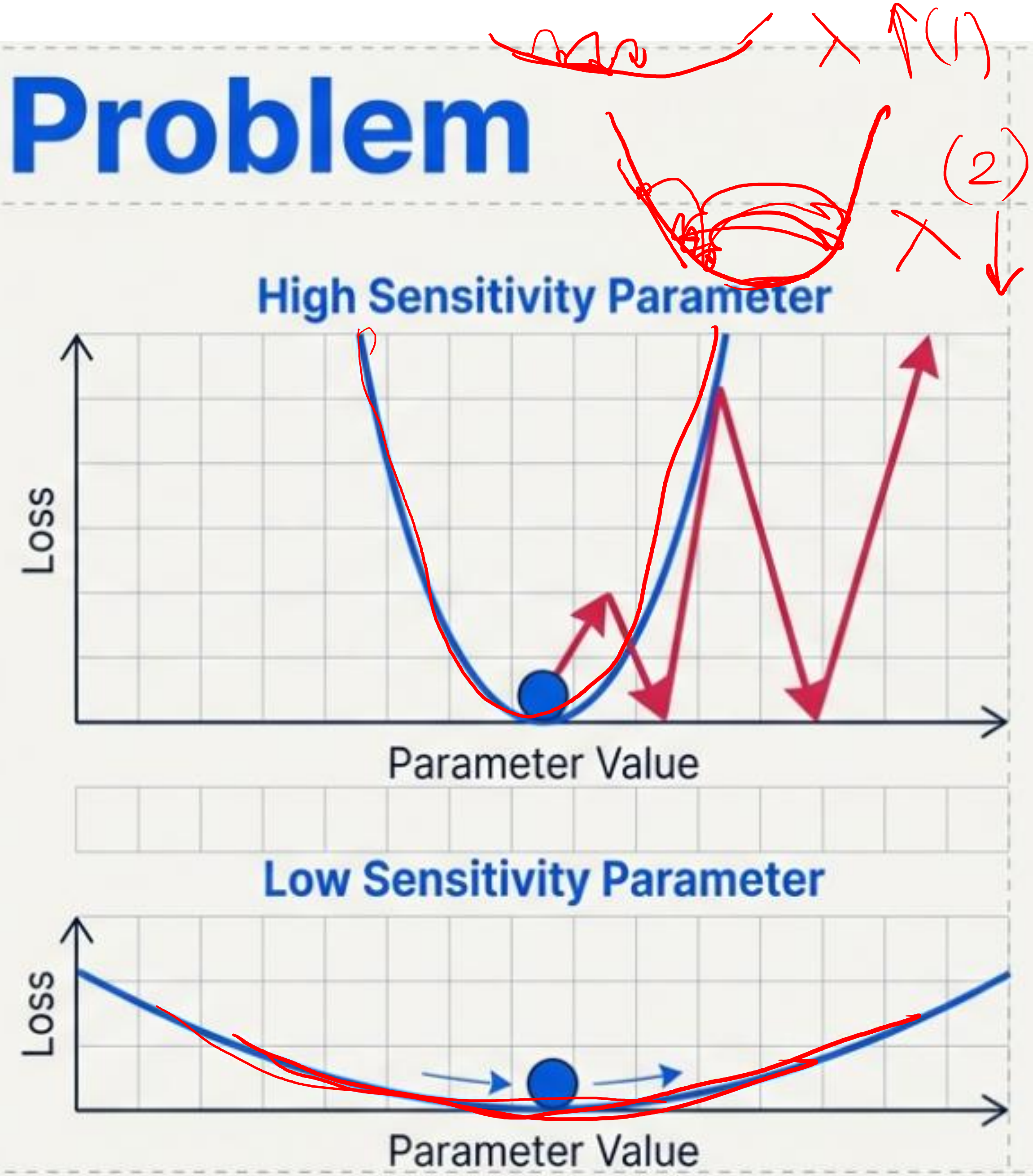
The Scale Problem

The Conflict

Momentum solved the direction problem, but what about scale? Not all parameters are created equal. Some weights have massive gradients; others have tiny, subtle ones.

A single global learning rate (α) cannot satisfy both:

- **Too small:** Tiny gradients learn too slowly.
- **Too big:** Large gradients explode or oscillate.



RMSProp: Adaptive Learning Rates

The Solution

RMSProp tracks the volatility of each parameter to adjust the step size individually.

- **High Variance (Steep Slope):** We hit the brakes (divide by a large number).
- **Low Variance (Flat Slope):** We hit the gas (divide by a small number).

The Math

$|g_t|$

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2$$

Running Average of Squared Gradients

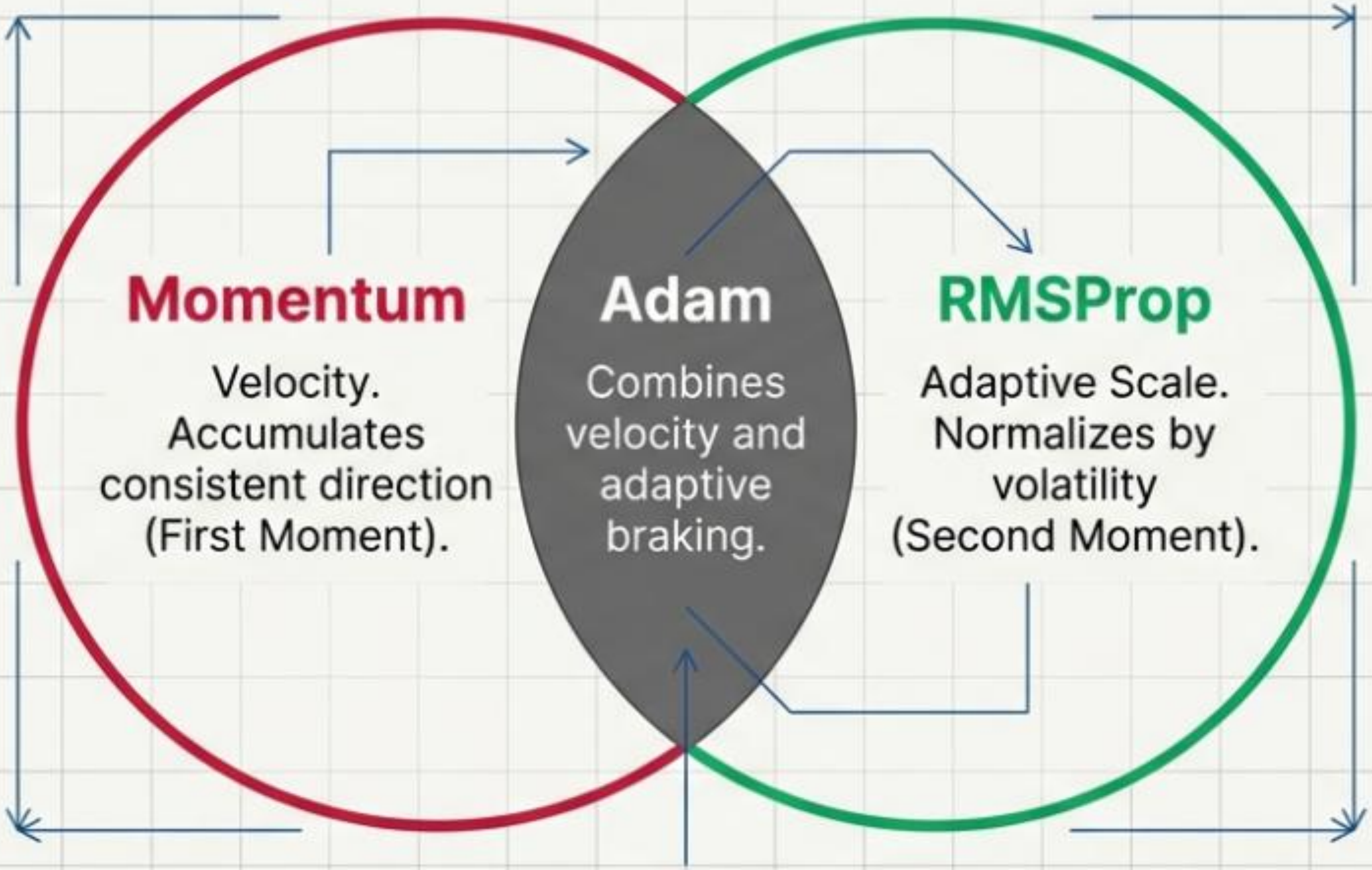
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{s_t} + \varepsilon} g_t$$

10^{-8}

Normalization / Adaptive Scaling

Adam: The Synthesis

Adaptive Moment Estimation



Why it works: Adam keeps a running mean of gradients (Momentum) AND a running mean of squared gradients (RMSProp). It works “out of the box” for messy landscapes.

Inside the Adam Engine

1. Momentum (First Moment)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

2. Adaptive Scale (Second Moment)

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

3. Bias Correction

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Prevents estimates from being biased toward zero at the start of training.

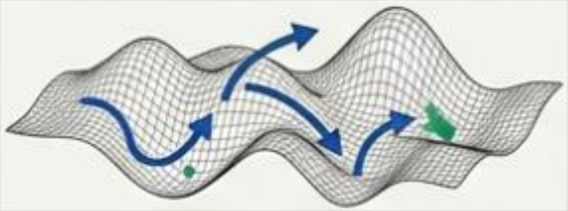
4. The Update

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

The Generalization Gap $-(L + \lambda\theta^2)$

Why Adam Wins

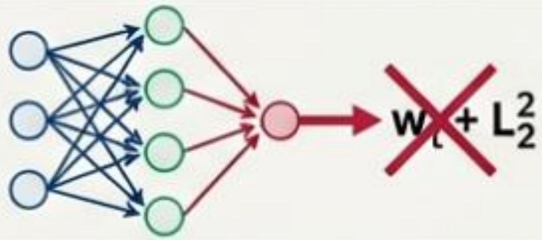
- Handles messy curvature and varying parameter scales.
- Requires significantly less tuning than SGD.
- The best "first try" optimizer for new problems.



Adam: Robust & Efficient

The Limitations

- **Generalization:** Sometimes SGD + Momentum generalizes better on classic vision tasks.
- **Critical Flaw:** The way Adam handles L2 Regularization (Weight Decay) is mathematically incorrect.



L2 Regularization: The Bug

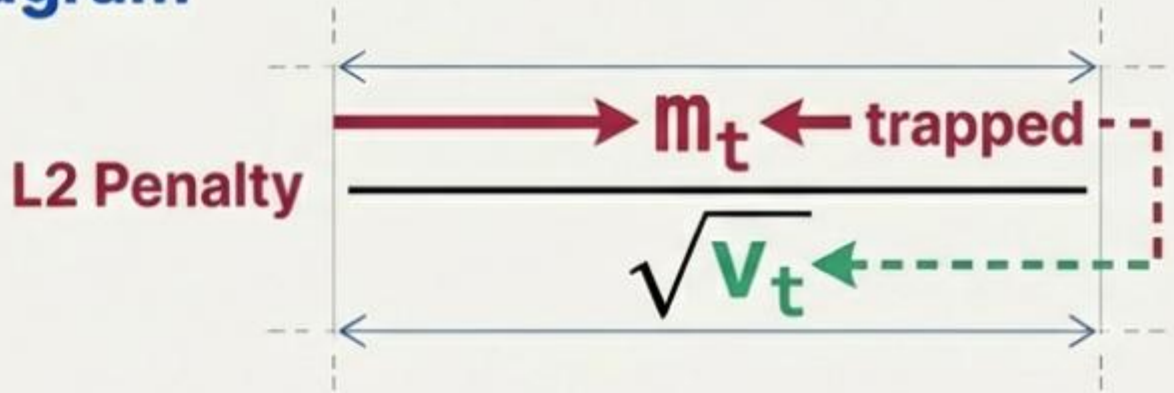
$$\theta_{t+1} = \cancel{\theta}\ \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Incorrect Weight Decay handling leads to suboptimal generalization.

The Bug in L2 Regularization

In SGD, “L2 Regularization” and “Weight Decay” are mathematically identical—they both shrink weights slightly at every step.

The Conflict Diagram



In classic Adam implementation, the L2 penalty is added to the gradient. This means the penalty gets scaled by the adaptive term. The shrinkage becomes uneven: parameters with large gradients (large v_t) get LESS shrinkage than intended.

The regularization is distorted.

AdamW: Decoupled Weight Decay

The Fix

AdamW decouples weight decay from the gradient update. It applies the

"shrinkage" ***after*** the adaptive step, ensuring a consistent force pulling weights to zero.

The Visual Comparison

Adam

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t + \lambda\theta_t}{\sqrt{v_t}}$$

Trapped inside

AdamW

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t}} - \alpha\lambda\theta_t$$

Decoupled / Pure Shrinkage

AdamW = Adam + Correct Weight Decay

Thank You