GDO: Fast, pitch and smooth

Jan Wilczek

Published on 2026-02-28 | Updated on 2026-03-01 | Reading time: 15 min

GDO: Fast, pitch and smooth

Jan Wilczek

Gradient descent is the workhorse of machine learning. Every neural network, every regression model, every recommendation system learns by following the gradient of a loss function downhill. Vanilla gradient descent does this well in principle, but in practice it struggles with noisy gradients, ill-conditioned curvature, and saddle points. Adaptive optimizers like Adam address these problems by maintaining per-parameter statistics, but they do so at the cost of memory and complexity. Momentum methods smooth the gradient signal, but they require tuning a separate hyperparameter and can overshoot when the curvature changes rapidly.

This post introduces GDO (Gradient Descent Optimizer), a single-parameter optimizer that combines momentum, adaptive scaling, and curvature-aware damping into a unified update rule. We derive the update from first principles, implement it in PyTorch, and benchmark it against Adam on a suite of test problems. The goal is not to claim that GDO is universally better than Adam, but to show that a simpler optimizer can recover most of the benefits of adaptive methods with less overhead and fewer hyperparameters.

Gradient descent and its limitations

Given a loss function $L(\theta)$ over parameters θ, gradient descent updates the parameters by:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

where η is the learning rate. This converges to a local minimum for convex L and sufficiently small η. In practice, three problems arise:

- **Noisy gradients.** When $\nabla L(\theta_t)$ is estimated from a mini-batch, the signal is corrupted by sampling noise. Large η amplifies this noise into parameter jitter.
- **Ill-conditioned curvature.** When the loss surface is steep in one direction and flat in another, a single η must be small enough for the steep direction to avoid overshooting, which makes progress in the flat direction slow.
- **Saddle points.** In high-dimensional spaces, most critical points are saddle points rather than local minima. Gradient descent slows down near saddle points because the gradient magnitude shrinks.

Momentum addresses the first problem by accumulating a velocity vector:

$$v_t = \beta v_t + \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta v_t$$

Adaptive methods address the second problem by scaling each parameter by its gradient history. Adam does both:

$$m_t = \beta_1 m_t + (1 - \beta_1) \nabla L(\theta_t)$$
$$v_t = \beta_2 v_t + (1 - \beta_2) \nabla L(\theta_t)^2$$
$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$

Adam has three hyperparameters (β_1, β_2, ϵ) and stores two buffers per parameter. For a model with N parameters, the optimizer state occupies $2N$ memory. Adam also does not address saddle points directly.

GDO: Fast, pitch and smooth

GDO combines momentum, adaptive scaling, and curvature-aware damping into a single update rule:

$$g_t = \nabla L(\theta_t)$$
$$m_t = \beta m_t + g_t$$
$$v_t = \max(\beta v_t, |g_t|)$$
$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{v_t + \epsilon}$$

The gradient g_t is computed from the current mini-batch. The momentum m_t accumulates gradients with decay β. The denominator v_t tracks the maximum of the decayed gradient magnitude and the current gradient magnitude. When gradients are consistent, v_t grows slowly and the step size is large. When gradients are noisy or the curvature changes rapidly, v_t grows quickly and the step size shrinks. The ϵ prevents division by zero.

The single hyperparameter β controls both the momentum decay and the damping rate. Typical values are $\beta = 0.9$ or $\beta = 0.99$. Compared to Adam, GDO has one hyperparameter instead of three and one buffer instead of two.

Implementation

```
class GDO(Optimizer):
    def __init__(self, params, lr=1e-3, beta=0.9, eps=1e-8):
        if lr < 0:
            raise ValueError(f"Invalid learning rate: {lr}")
        if beta < 0 or beta > 1:
            raise ValueError(f"Invalid beta: {beta}")
        if eps < 0:
            raise ValueError(f"Invalid eps: {eps}")
        defaults = dict(lr=lr, beta=beta, eps=eps)
        super().__init__(params, defaults)

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()
        for group in self.param_groups:
            lr = group['lr']
            beta = group['beta']
            eps = group['eps']
            for p in group['params']:
                if p.grad is None:
                    continue
                g = p.grad
                if g.is_sparse:
                    raise RuntimeError("GDO does not support sparse gradients")
                state = self.state[p]
                if len(state) == 0:
                    state['step'] = 0
                    state['m'] = torch.zeros_like(p)
                    state['v'] = torch.zeros_like(p)
                m = state['m']
                v = state['v']
                m.mul_(beta).add_(g)
                torch.maximum(v.mul(beta), g.abs(), out=v)
                p.addcdivv_(m, v + eps, -lr)
                state['step'] += 1
        return loss
```

The implementation follows the standard PyTorch optimizer interface. The state dictionary stores the momentum buffer `m` and the damping buffer `v` for each parameter. The update uses `addcdivv_` to fuse the division and subtraction into a single operation.

Benchmarks

We benchmark GDO against Adam on three test problems: a quadratic bowl, the Rosenbrock function, and a small neural network trained on synthetic data. For each problem we sweep the learning rate and report the best stable run.

Quadratic bowl

The quadratic bowl is $L(\theta) = \frac{1}{2}\theta^T A \theta$ where A is a diagonal matrix with condition number 100. This tests how well the optimizer handles ill-conditioned curvature.