[0.3] Optimization, Hyperparameters, and Scale

David Quarel

ARENA

June 9, 2026

What's today about?

- Yesterday: you built ResNet34. Today: we explore what the optimizer does, build some from scratch, and train with them.
- The road there:
  1. Optimizers — SGD, momentum, Adam, AdamW, and why they exist.
  2. Experiment tracking + hyperparameter sweeps with Weights & Biases.
  3. Distributed training — when one GPU isn't enough.

The training loop

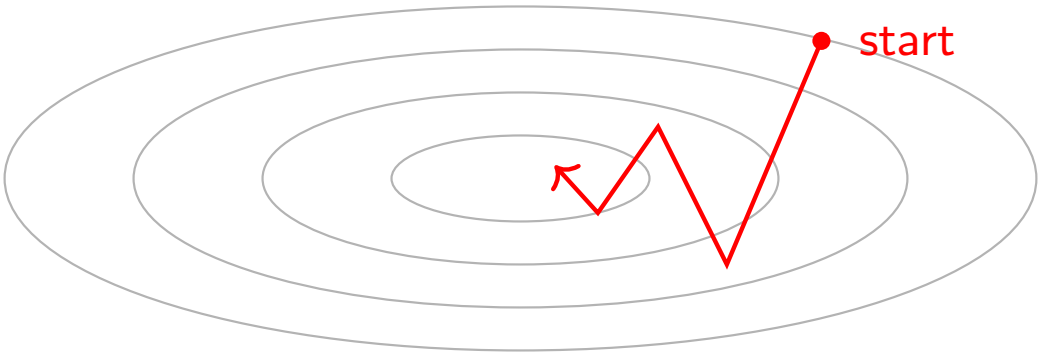
- Same four steps every iteration: *forward*, *loss*, *backward*, *step*.

```
for x, y in dataloader:
    y_hat = model(x)
    loss = loss_fn(y,y_hat)
    loss.backward()
    optimizer.step()          # Build this today
    optimizer.zero_grad()     # Build this today
```

- `loss.backward()` fills `.grad` on every parameter via autograd.
- `optimizer.step()` applies an update rule (e.g. Adam: momentum + per-parameter adaptive learning rate).
- `zero_grad()` before backward: gradients **accumulate** by default in PyTorch.

Loss landscapes & why this is hard

- Loss is a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ over millions of parameters. We never see the full thing — only 1D/2D slices through it.
- Common pathologies:
  - **Ravines:** one direction much steeper than another. Naive SGD oscillates across, crawls along.
  - **Saddle points:** gradient is zero but you're not at a minimum.
  - **Plateaus:** gradient is tiny; nothing moves.



ill-conditioned "ravine" — SGD zigzags

Gradient descent recap

Gradient descent

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t)$$

- Learning rate α is the single most important hyperparameter. Too high $\rightarrow$ divergence; too low $\rightarrow$ glacial progress.
- In practice we don't compute $\nabla\mathcal{L}$ on the full dataset — we use **batches**. The gradient becomes a noisy estimate, but each step is much cheaper.
- If the batch size is too large, one can do several forward passes over mini-batches and accumulate gradients to emulate a larger batch size.
- **Batch size:**
  - Larger $\rightarrow$ (+) lower-variance gradient, better GPU utilization
  - Larger $\rightarrow$ (-) more memory, fewer optimizer steps per epoch, often worse generalization at scale

SGD

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

```
class SGD(Optimizer):
    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    @torch.no_grad()
    def step(self):
        for p in self.params:
            p = p - self.lr * p.grad  # SGD update

    def zero_grad(self):
        for p in self.params:
            p.grad.zero_()
```

Momentum

SGD with momentum

$$v_{t+1} = \mu\, v_t + \nabla_\theta \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha\, v_{t+1}$$

- v_t is an exponentially weighted moving average of past gradients, decayed by $\mu \in [0, 1)$ (typically 0.9).
- In a ravine, gradient directions across the ravine cancel on average; directions along it accumulate. Smooth, fast progress toward the minimum.
- Equivalent picture: a ball with mass rolling down the landscape. Reaches a *terminal velocity* along consistent directions.
- distill.pub/2017/momentum has a beautiful interactive on this.

RMSprop

RMSprop

$$s_{t+1} = \beta\, s_t + (1-\beta)\left(\nabla_\theta \mathcal{L}\right)^2$$

$$\theta_{t+1} = \theta_t - \alpha\, \frac{\nabla_\theta \mathcal{L}}{\sqrt{s_{t+1}} + \varepsilon}$$

- s_t is a moving average of squared gradients (per-parameter).
- Effect: each parameter gets its own **adaptive learning rate**. Parameters with large historical gradients get smaller updates; parameters with tiny gradients get amplified.
- Implicitly handles ill-conditioning — it's a cheap, online approximation of dividing by the diagonal of the Hessian.
- **Memory cost:** s_t is parameter-sized $\Rightarrow 2\times$ memory vs SGD.

Adam

Adam

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\,\nabla\mathcal{L} && \text{(first moment)} \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)\,(\nabla\mathcal{L})^2 && \text{(second moment)} \\
\hat{m}_{t+1} &= m_{t+1}/(1-\beta_1^{t+1}) && \text{(bias correction)} \\
\hat{v}_{t+1} &= v_{t+1}/(1-\beta_2^{t+1}) \\
\theta_{t+1} &= \theta_t - \alpha\,\hat{m}_{t+1}/(\sqrt{\hat{v}_{t+1}}+\varepsilon)
\end{aligned}
$$

- **Momentum + RMSprop + bias correction**, all at once.
- Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$. These usually just work.
- Bias correction matters early in training when m, v are still near zero — without it the first few steps are tiny.
- **Memory cost:** m_t and v_t are both parameter-sized $\Rightarrow 3\times$ memory vs SGD.

Adam vs AdamW: the weight-decay subtlety

- **Naive weight decay:** add $\frac{\lambda}{2}\|\theta\|^2$ to the loss. Looks innocent, but for adaptive methods, the L2 gradient $\lambda\theta$ gets divided by $\sqrt{\hat{v}_t}$ along with everything else — so parameters with large historical gradients are decayed less. Not what you want.

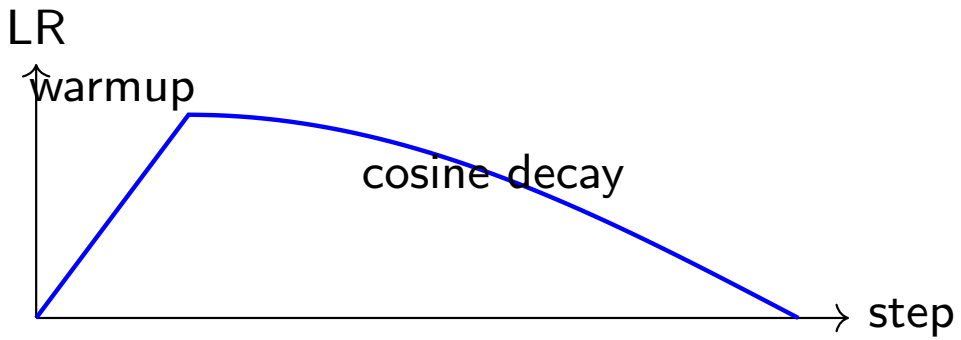
AdamW: decoupled weight decay

$$\theta_{t+1} = \theta_t - \alpha\,\hat{m}_{t+1}/(\sqrt{\hat{v}_{t+1}} + \varepsilon) - \alpha\,\lambda\,\theta_t$$

- The $\lambda\theta$ term sits *outside* the adaptive scaling — every parameter shrinks toward zero by the same relative amount each step.
- In practice: just use `AdamW`.

Schedules & parameter groups

- Fixed LR is fine for toy problems. Real training uses a **schedule**: warmup (linear ramp from 0) + decay (cosine or linear).

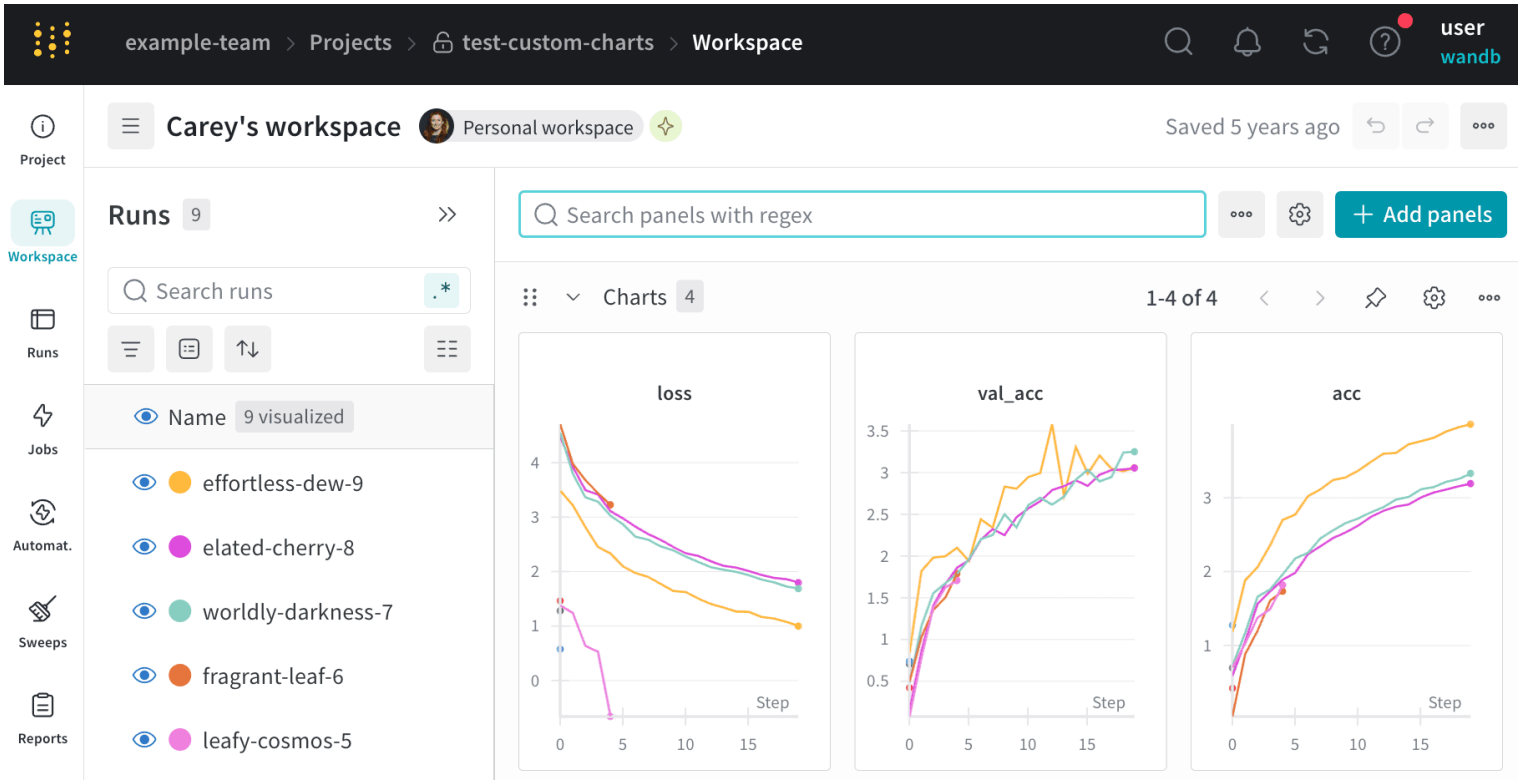


- **Parameter groups:** different learning rates for different parts of the model. Common in fine-tuning (slow backbone, fast head):

```
optimizer = torch.optim.AdamW([
    {"params": backbone.parameters(), "lr": 1e-5, "weight_decay": 0.01},
    {"params": head.parameters(),     "lr": 1e-3, "weight_decay": 0.0},
])
```

Why track experiments? Weights & Biases

- WandB is the de-facto tool: can easily track and log loss, accuracy etc. while the model is training

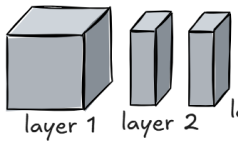


Parallel processes:

process 1
process 2
process 3
⋮

(each represents computations happening on a different GPU)

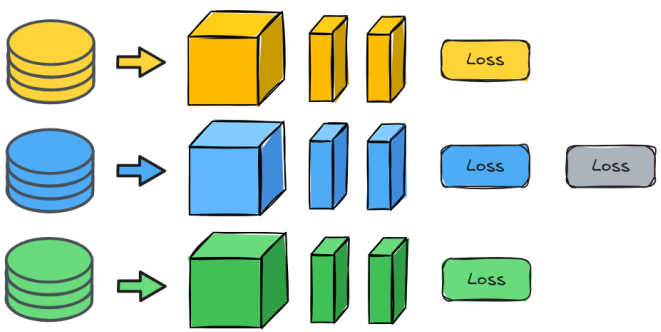
Model:



(each layer is a sequential step)

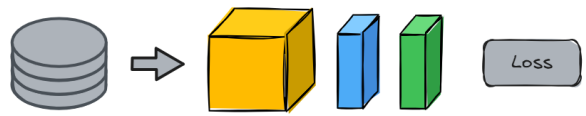
Data Parallelism

- Splitting batches of data across processes
- Easiest form to implement & manage; supported by PyTorch



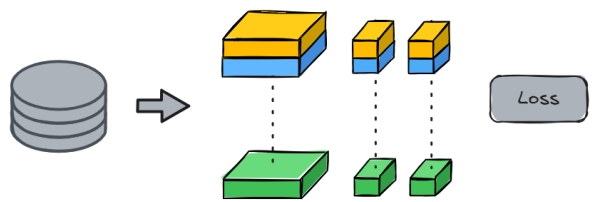
Pipeline Parallelism / Vertical Model Parallelism

- Splitting model into serialized stages, e.g. layers
- Requires careful management of inter-stage latency
- Natively supported by PyTorch: torch.distributed.pipelining



Tensor Parallelism / Horizontal Model Parallelism

- Splitting model into parallel stages (often splitting parameters into shards)
- Most complex to implement (requires fine-grained control over computations)
- Experimentally supported by PyTorch: torch.distributed.tensor.parallel



Glossary

- **Optimizer**: the rule for turning $\nabla\mathcal{L}$ into a parameter update.
- **Learning rate** α: step size. The most important dial.
- **Momentum** (μ): exponentially weighted average of past gradients.
- **Weight decay** (λ): shrink parameters toward zero each step.
- **Adam**: momentum + per-parameter adaptive LR + bias correction.
- **AdamW**: Adam with decoupled weight decay. Use by default.
- **LR schedule**: warmup then decay; pretty much always.
- **Parameter group**: subset of parameters with its own LR / weight decay.
- **WandB**: experiment-tracking dashboard
- **Sweep**: automated hyperparameter search