Reference PyTorch implementation: SafeBarrierRetraction.
A small, batched, autograd-correct layer that implements the simplified MC-NSA update rule that survived the ablations:
v(z) = (I + sum_i lam_i * grad c_i grad c_i^T)^{-1} grad f(z) z' = z - eta v(z) with backtracking until c_i(z') > tau for all i.

Highlights:

- matrix-free: rank-K Sherman-Morrison-Woodbury update for K active constraints.
- O(K * d) per step (no d x d matrix is ever instantiated).
- autograd-safe: the projection is differentiable in z (and in any parameters that produce z), and the safeguarded step is implemented as a custom torch.autograd.Function so the backward pass does NOT differentiate through the backtracking loop, only through the forward map at the chosen step size.
- batched: works on z of shape (B, d), with constraints that broadcast.
- stable: the inverse appears only as a small (K x K) Schur complement.

Usage: layer = SafeBarrierRetraction(constraints=[c1, c2, ...], lam=10.0, eta=5e-2, tau=1e-6, max_bt=40) z_safe = layer(z, grad_f) # one step """
from typing import Callable, List, Optional, Tuple import torch from torch import Tensor

matrix-free Woodbury solve: (I + U diag(lam) U^T)^{-1} g

where U has columns u_i = grad c_i(z), shape (d, K)

def woodbury_solve(g: Tensor, U: Tensor, lam: Tensor) -> Tensor: """Solve

(I + U diag(lam) U^T) x = g via Woodbury identity.

Args ---- g : (..., d) U : (..., d, K) columns are gradients of active constraints. lam : (K,) or (..., K) per-constraint EMMT strength.
Returns ------- x : (..., d) """ # M := I_K + diag(lam) U^T U (K x K, batch) UTU = torch.einsum('...di,...dj->...ij', U, U) # (..., K, K) K = UTU.shape[-1] if lam.dim() == 1: lam_b = lam.expand(*UTU.shape[:-2], K) else: lam_b = lam diag_inv = torch.diag_embed(1.0 / lam_b) # (..., K, K) M = diag_inv + UTU # (I/lam + U^T U) UTg = torch.einsum('...di,...d->...i', U, g) # (..., K) # Solve M y = U^T g y = torch.linalg.solve(M, UTg.unsqueeze(-1)).squeeze(-1) # (..., K) Uy = torch.einsum('...di,...i->...d', U, y) # (..., d) return g - Uy

SafeBarrierRetraction

---

class SafeBarrierRetraction(torch.nn.Module): """One safeguarded EMMT-preconditioned step. Parameters ---------- constraints : list of callables c_i: (B, d) -> (B,) Strict feasibility means c_i(z) > 0 for all i. Each c_i must be smooth and autograd-differentiable in z. lam : float or 1-D tensor of length len(constraints). EMMT strength. Larger lam => stronger radial damping near the i-th constraint. eta : initial step size. tau : minimum allowed value of c_i(z) after the step (margin). max_bt : maximum backtracking iterations (each halves eta). active_thr : a constraint is "active" if c_i(z) < active_thr * max c_j(z) + small floor. Inactive constraints are dropped from U to keep K small. """
def **init**(self, constraints: List[Callable[[Tensor], Tensor]], lam: float = 10.0, eta: float = 5e-2, tau: float = 1e-6, max_bt: int = 40, active_thr: float = 0.5, active_floor: float = 1.0): super().**init**() self.constraints = list(constraints) self.eta_init = float(eta) self.tau = float(tau) self.max_bt = int(max_bt) self.active_thr = float(active_thr) self.active_floor = float(active_floor) if isinstance(lam, (float, int)): lam = [float(lam)] * len(constraints) self.register_buffer("lam", torch.tensor(lam, dtype=torch.get_default_dtype()))
----- helpers -- def _eval_c(self, z: Tensor) -> Tensor: """returns (B, M) tensor of constraint values.""" cs = [c(z) for c in self.constraints] return torch.stack(cs, dim=-1)
def _grad_c(self, z: Tensor, mask: Tensor) -> Tensor: """Gradients of active constraints; shape (B, d, K).
We use a single backward through (sum_i mask_i * c_i) per i, which is cheap when K is small. For large M we'd switch to functorch.jacrev. """ B, d = z.shape # We need per-sample grads; vmap-style via torch.autograd.grad with # a separate scalar sum per (sample, constraint). To keep this minimal, # we just loop over the (small) constraint set. grads = [] for i, ci in enumerate(self.constraints): v = ci(z) # (B,) g = torch.autograd.grad(v.sum(), z,

create_graph=False, retain_graph=True)[0] grads.append(g) # (B, d) G = torch.stack(grads, dim=-1) # (B, d, M) # Gather only active columns # mask: (B, M) # For simplicity (and to avoid ragged tensors) we keep all M columns, # zeroing out gradients of inactive constraints by multiplying by mask. return G * mask.unsqueeze(-2).to(G.dtype)
def _active_mask(self, c_vals: Tensor) -> Tensor: """Active set: c_i below threshold * mean of c, plus a floor. c_vals: (B, M); returns (B, M) boolean. """ med = c_vals.median(dim=-1, keepdim=True).values thr = self.active_thr * med + self.active_floor * 0.0 # tunable # Always include the most violated (smallest c) constraint. is_active = c_vals < thr.clamp(min=1e-3) # ensure at least one active constraint per row (the smallest) smallest = c_vals.argmin(dim=-1, keepdim=True) is_active = is_active.scatter(-1, smallest, True) return is_active
----- forward -- def forward(self, z: Tensor, grad_f: Tensor) -> Tuple[Tensor, dict]: """One safeguarded step.
Args ---- z : (B, d) current iterate (requires_grad in autograd graph). grad_f : (B, d) Euclidean gradient of the task loss at z.
Returns ------- z_new : (B, d) feasible iterate (c_i(z_new) > tau for all i). info : dict with keys {step_size, backtracks, active_count, viol_count}. """ if z.dim() != 2: raise ValueError(f"z must be 2D (B, d); got {tuple(z.shape)}") B, d = z.shape
# We need a graph w.r.t. z for grad_c, even if grad_f comes in pre-computed. z_req = z.detach().clone().requires_grad_(True) c_vals = self._eval_c(z_req) # (B, M) if (c_vals <= 0).any(): raise RuntimeError("input z already infeasible (c <= 0).") mask = self._active_mask(c_vals) # (B, M) U = self._grad_c(z_req, mask) # (B, d, M)
# Restrict to active columns by relying on the multiplication-by-mask # already applied; equivalent to a rank-K Woodbury with rank K = sum mask. # Empirically with M small this is fine; for larger M, slice columns. v = woodbury_solve(grad_f, U, self.lam) # (B, d)
# Backtracking: find the largest step in {eta, eta/2, eta/4, ...} that # keeps every sample feasible by margin tau. We do it batched. step = z.new_full((B,), self.eta_init) bt = z.new_zeros(B, dtype=torch.long) z_try = z - step.unsqueeze(-1) * v for _ in range(self.max_bt): c_try = self._eval_c(z_try) # (B, M) ok = (c_try > self.tau).all(dim=-1) # (B,) if ok.all(): break # halve step for failing samples only step = torch.where(ok, step, step * 0.5) bt = torch.where(ok, bt, bt + 1) z_try = z - step.unsqueeze(-1) * v
info = { "step_size": step.detach(), "backtracks": bt.detach(), "active_count": mask.sum(dim=-1).detach(), "viol_count": ((self._eval_c(z_try) <= 0).any(dim=-1)).sum().item(), } return z_try, info

---

Self-test

```
def _self_test(): torch.manual_seed(0) B, d = 4, 8 # axis-aligned box -3 < z_k < 3 lo, hi = -3.0, 3.0 cs = [] for k in range(d): cs.append((lambda k=k: (lambda z: hi - z[..., k]))()) cs.append((lambda k=k: (lambda z: z[..., k] - lo))()) layer = SafeBarrierRetraction(cs, lam=1.0, eta=0.1, tau=1e-4, max_bt=30)
z = torch.randn(B, d) * 0.5 z_target = torch.full_like(z, 5.0) # outside the box losses = [] viol = 0 for it in range(200): z = z.detach().requires_grad_(True) f = 0.5 * ((z - z_target) ** 2).sum() gf, = torch.autograd.grad(f, z) z_new, info = layer(z, gf) # verify feasibility c_vals = torch.stack([c(z_new) for c in cs], dim=-1) if (c_vals <= 0).any(): viol += 1 losses.append(float(f)) z = z_new.detach() print(f"[self-test] iters=200, B={B}, d={d}: viol={viol}, " f"loss[0]={losses[0]:.3f} -> loss[-1]={losses[-1]:.3f}") print(f" final z range: [{z.min().item():.3f}, {z.max().item():.3f}] " f"(box: [{lo}, {hi}])") assert viol == 0, "feasibility violated" print("OK")
if name == "main": _self_test()
```