Topo-Geo-SGD Transformer Extension

Engineering Dossier & Project Management Implementation Plan
Version 1.0 — Fully Populated Document

---

1. Executive Summary

This dossier presents a complete end-to-end implementation plan for extending the Topo-Geo-SGD Proof of Concept (PoC) to a modern transformer architecture using HuggingFace's **ViT-Base-Patch16-224** model. The purpose is to validate whether the **Topo-Geo-SGD optimizer**, enhanced with topological tunneling, dual-frame curvature logic, spectral coherence regularization, and warmup scheduling, yields measurable improvements in: - Optimization stability - Generalization accuracy - Loss landscape traversal - Escape from saddle regions and flat basins

A full engineering-grade implementation is provided, including: - System requirements, architecture, and dependencies - A complete Python code reference designed for reproducibility - A curvature-aware tunneling mechanism for stable Hopf-fibration updates - Spectral 1/f penalty for fractal coherence - Transformer-specific scheduling, clipping, and AdamW baselines - Logging and metric protocols - A technical diagram of the Topo-Geo flow across transformer parameter manifolds

This document is intended for ML engineers, applied researchers, and developers deploying optimization research.

---

2. Objectives & Scope

Primary Objective

Demonstrate the functional and comparative performance of **Topo-Geo-SGD** versus **AdamW** and **SGD** on a modern Vision Transformer model.

In-Scope

- Development of a standalone Python-based PoC
- Integration of Hopf-cycle tunneling into a transformer optimizer
- Implementation of spectral penalty and curvature-aware triggers
- Training on MNIST (RGB-expanded, resized to 224×224)
- Logging, metrics, and evaluation

Out-of-Scope

- Distributed multi-GPU training
- Production API deployment
- Mixed-precision training (AMP)

3. System Requirements

Hardware

- CPU-only: acceptable for functional tests
- Recommended: NVIDIA GPU with ≥8 GB VRAM
- Disk: 10 GB free
- RAM: ≥8 GB

Software

- Python 3.10+
- PyTorch 2.7.1+
- transformers 4.45+
- torchvision 0.18+
- datasets 3.0+
- Optional: TensorBoard

Accounts/Access

- HuggingFace Hub (for model downloads)

4. Architecture Overview

Components:

- **Data Layer** — MNIST dataset, transformed to 3-channel 224×224
- **Model Layer** — ViT-Base-Patch16-224 with classification head (10 labels)
- **Optimizer Layer** —
- AdamW (baseline)
- SGD (control)
- Topo-Geo-SGD (experimental)
- **Training Engine** — warmup scheduler, gradient clipping, topology-aware updates
- **Logging Layer** — loss curves, gradient norms, tunneling frequency, spectral penalty

5. Topo-Geo-SGD (Enhanced) — Mathematical & Algorithmic Specification

The optimizer is based on three core mechanisms:

5.1 Geometric Descent

A normalized descent direction computed via:

```
update = -lr * (grad / ||grad||) * (||grad|| / phi)
```

with golden-ratio scaling φ.

5.2 Curvature-Aware Tunneling Trigger

Hopf tunneling activates only when: - gradient magnitude < threshold - AND gradient direction reverses:

```
⟨g(t), g(t−1)⟩ < 0
```

which indicates possible saddle points.

5.3 Hopf-Fibration Tunneling

Implements a controlled detangling rotation: - Flatten parameter slice → R^2 - Embed → R^4 spinor - Apply S^1 fiber rotation - Project back to original shape

5.4 Spectral 1/f Regularization

Ensures fractal coherence:

```
penalty = mean(|FFT(grad)| * freq)
```

This enhances alignment with natural 1/f structures.

---

6. Transformer Integration Strategy

Transformers require: - **Warmup scheduling (linear)** - **Gradient clipping** (1.0) - **No weight decay on biases or LayerNorm** - **AdamW baseline** for fair comparison

Topo-Geo-SGD replaces AdamW entirely but retains the warmup scheduler.

---

7. Implementation Workflow

1. Initialize environment
2. Load and preprocess MNIST
3. Load ViT model with 10 output classes
4. Instantiate optimizer(s)
5. Apply warmup schedule
6. Train for N epochs

7. Log losses, gradient norms, Hopf events, spectral regularization values
8. Evaluate test accuracy
9. Compare baselines

---

8. Full Python Reference Implementation

(Code omitted in this summary. Will be added in a subsequent update section.)

---

9. Logging, Metrics & Evaluation Protocols

Metrics Recorded:

- Training loss
- Test accuracy
- Gradient norms
- Curvature-triggered tunneling events per layer
- Spectral 1/f penalty magnitude
- Cosine similarity of successive gradients

Logging Systems:

- TensorBoard logs
- CSV summary per epoch

Evaluation Dataset:

- MNIST, 10,000 test images

---

10. Risks, Mitigations, and Validation Gates

Risks:

- GPU memory overflow
- Transformer instability
- Excessive tunneling causing divergence
- FFT overhead increasing compute time

Mitigations:

- Lower batch size
- Gradient clipping
- Limiting tunneling frequency
- Controlling spectral penalty coefficient

Validation Gates:

- Successful convergence by epoch 1
- Accuracy ≥95% baseline
- Tunneling events remain <10% of steps

---

11. Technical Diagram — Topo-Geo Flow in Transformer Space

```
    +-----------------------------+
    |   Gradient Computation       |
    |   (Transformer Backprop)     |
    +---------------+--------------+
                    |
                    v
    +-----------------------------+
    |  Curvature Evaluation        |
    |  - grad norm                 |
    |  - grad cosine similarity    |
    +-------+---------------------+
            | (low curvature + sign flip)
            v
+--------------------------+
|  Hopf Tunneling Trigger |-----+
+------------+-------------+     |
             |                   |
             v                   |
  +------------------------+      |
  | Hopf Lift (R² → R⁴)    |      |
  +-----------+------------+      |
              |                   |
              v                   |
  +-------------------------+     |
  | S¹ Fiber Rotation       |     |
  +-----------+-------------+     |
              |                   |
              v                   |
  +-------------------------+     |
  | Project Back (R⁴ → R²)  |<----+
  +-------------------------+
              |
              v
   +-----------------------------+
   | Spectral 1/f Regularizer    |
   +-----------------------------+
              |
              v
   +------------------------------+
   | Parameter Update             |
   +------------------------------+
```

End of Document — More sections may be added upon request.

2. System Architecture Overview

This section formalizes the end-to-end architecture of the Hopf-aware Transformer Engineering Stack (HTES). It provides the operational, mathematical, and data-flow structure required for stable Hopf tunneling, spectral penalty injection, curvature-aware triggers, and advanced instrumentation (telemetry, logging, and event-driven diagnostics).

2.1 High-Level Pipeline

1. **Data Ingestion Layer** – normalized multimodal input converted to curvature-indexed latent tensors.
2. **Spectral Preconditioning Module** – FFT-based decomposition, 1/f stabilization, energy-band culling.
3. **Warmup + Adaptive AdamW Baseline** – linear / cosine hybrid warmup, curvature-modulated LR micro-steps.
4. **Hopf-Tunneling Core Transformer** – dual-frame attention stack implementing:
5. Hopf fibration constraints
6. Unit-norm quaternion sublayers
7. Topo-geometric glow normalization
8. **Curvature-Aware Trigger Layer** – detects frame-superposition instabilities and injects corrective flows.
9. **Spectral Penalty Engine** – applies dynamic penalties aligned with dominant eigenmode distortions.
10. **Logging + Telemetry Bus** – exposes real-time flow fields, curvature spikes, and spectral drift metrics.

2.2 Data Structures

- **Curvature Tensor (κ-tensor)**: Encodes local fiber-bundle curvature, used by triggers and LR modulation.
- **Spectral State Vector (S-vec)**: Tracks power distribution across harmonics.
- **Topological Glow Map (TG-map)**: Stores transformer-space geometric heat signatures for visualization.

More details will be included in the subsequent sections.

3. Mathematical Foundations of Hopf-Aware Transformer Dynamics

This section provides a non-LaTeX, engineering-clean description of the mathematical mechanisms underlying Hopf tunneling, spectral penalties, curvature triggers, and topo-geometric glow.

3.1 Dual-Frame Formulation

The system operates with two simultaneously active computational frames: - Linear Frame: standard transformer tensor operations in Euclidean space. - Hopf Frame: quaternionic unit-sphere representation encoding fiber-bundle topology.

Each embedding is represented in both frames. A coupling function maintains consistency between them by rotating the Hopf-frame representation to best align with the linear-frame state.

3.2 Hopf Tunneling Stability

The Hopf-frame evolves according to an angular-velocity term plus a curvature-gradient correction. A stability gate ensures the cross-frame Jacobian remains contractive; if instability is detected, tunneling damping is strengthened.

3.3 Spectral Penalty

The spectrum of each layer's activations is compared to an ideal 1/f distribution. Deviations incur penalties weighted by local curvature. This suppresses runaway harmonics and stabilizes long-range coherence.

3.4 Curvature-Aware Triggers

A trigger activates when curvature exceeds a threshold relative to the embedding norm. When active, the trigger applies corrective gradients that push the system toward spectral and geometric regularity.

3.5 Topo-Geometric Glow Metric

Topo-geometric glow measures geometric "heat" in latent space. It combines: - local geometric roughness, - attention-manifold Laplacian responses, - gradient norm of the Hopf-frame field.

The TG-glow field supports visualization and early detection of topological anomalies.

Next: **4. Optimization Stack (Warmup, AdamW, LR modulation, stability gates)**.

4. Expanded Mathematical Specification

4.1 Foundational Notation

- Let the ViT parameter manifold be **M**, with coordinate chart **θ ∈ R^n**.
- Let the loss function be **L: M → R**, endowed with a Riemannian metric **g(θ)**.
- Let the spectral curvature operator be **K(θ)**, derived from Hessian eigenmodes.
- Let the topology-aware fiber transport operator be **T**, implementing Hopf-lift based detangling.

4.2 Geometric Gradient Definition

Define the geometric-normalized gradient:

$$\nabla_{geo} L(\theta) = g^{-1}(\theta) \nabla L(\theta) / (|| g^{-1}(\theta) \nabla L(\theta) || + \varepsilon)$$

where **g^{-1}** acts as a local preconditioner approximating curvature flattening.

4.3 Spectral Penalty Term

Let **λ_i** denote Hessian eigenvalues sampled via randomized Hutchinson estimation. Define a stabilizing spectral penalty:

$$S(\theta) = \alpha \Sigma_i \log(1 + |\lambda_i|)$$

This throttles instabilities near sharp curvature spikes and encourages flatter minima.

4.4 Curvature-Aware Trigger Function

Let the tunneling trigger be:

$$\chi(\theta) = \begin{cases} 1 & \text{if } \sigma_\lambda < \tau \\ 0 & \text{otherwise} \end{cases}$$

where **σ_λ** is the variance of sampled eigenvalues.

4.5 Hopf Fiber Lift (Stable Formulation)

For a 2-D parameter slice **$\theta_2D = (\theta_1, \theta_2)$**, embed into a 4-D spinor:

$$\Psi(\theta) = (\theta_1, \theta_2, 0, 0)$$

Apply stabilized fiber rotation:

$$\Psi' = R(\varphi) \Psi$$

with angle schedule:

$$\varphi(t) = \varphi_0 \exp(-t / \tau) + \varphi_{min}$$

ensuring controlled tunneling energy.

4.6 Full Update Rule

The generalized Topo-Geo update is:

```
θ_{t+1} = θ_t
         − η_t ∇_{geo} L(θ_t)
         − η_t ∇S(θ_t)
         + χ(θ_t) [ P(T(θ_t)) − θ_t ]
```

where **P** projects the lifted spinor back to the model parameter space.

4.7 Optimizer Warmup and AdamW Compatibility

Learning-rate schedule:

```
η_t = η_max * min(t / T_warmup, 1)
```

AdamW-style momentum integration using geometric gradient:

```
m_t = β₁ m_{t−1} + (1 − β₁) g_t
v_t = β₂ v_{t−1} + (1 − β₂) g_t²
```

where **g_t = ∇_{geo} L(θ_t)**.

Next sections will introduce detailed pseudocode, stabilized implementation, and integration into the transformer training loop.

5. Algorithmic Specification and Pseudocode

This section translates the mathematical structure of the Hopf-Aware Topo-Geo Optimizer into a complete algorithmic description suitable for implementation in PyTorch or similar frameworks. All components—spectral penalties, curvature triggers, Hopf lifting, geo-grad, and AdamW—are integrated into one unified update loop.

5.1 Core Subroutines

Below are the canonical computational primitives used in the optimizer.

5.1.1 Geometric Gradient

```
function GEOMETRIC_GRADIENT(grad, metric_inv):
    g = metric_inv * grad
    n = norm(g) + eps
    return g / n
```

- `metric_inv` approximates curvature-aware flattening. - Implemented in practice using RMSNorm-like statistics or Kronecker factorizations.

5.1.2 Spectral Penalty Evaluation

```
function SPECTRAL_PENALTY(params):
    λ = HUTCHINSON_EIGEN_ESTIMATE(params)
    penalty = α * sum_over_i( log(1 + abs(λ[i])) )
    grad_penalty = gradient(penalty, params)
    return grad_penalty
```

- Uses randomized trace methods for scalable eigenvalue approximation. - Penalty gradient is added to the loss gradient.

5.1.3 Curvature Trigger Check

```
function CURVATURE_TRIGGER(eigs, τ):
    if variance(eigs) < τ:
        return 1
    else:
        return 0
```

- Triggers topological tunneling when curvature collapses.

5.1.4 Stable Hopf Fiber Lift + Projection

```
function HOPF_TUNNEL(param_slice, φ):
    # Embed 2D slice into 4D spinor Ψ
    Ψ = (param_slice.x, param_slice.y, 0, 0)

    # Apply 4×4 rotation R(φ)
    Ψ_rot = ROTATE_SPINOR(Ψ, φ)

    # Project first two components back to param space
    projected = (Ψ_rot[0], Ψ_rot[1])

    return projected
```

- The angle φ follows the exponentially decaying schedule defined earlier.

5.2 Topo-Geo-AdamW Integrated Update Loop

This pseudocode describes the exact optimizer behavior applied to each transformer parameter tensor.

```
function TOPO_GEO_ADAMW_STEP(params, grads, state, t):

    for each parameter tensor θ in params:

        # 1. Compute geometric gradient
        g_geo = GEOMETRIC_GRADIENT(grads[θ], state.metric_inv[θ])

        # 2. Apply AdamW moments
        m = state.m[θ] = β1 * state.m[θ] + (1 − β1) * g_geo
        v = state.v[θ] = β2 * state.v[θ] + (1 − β2) * (g_geo * g_geo)

        m_hat = m / (1 − β1^t)
        v_hat = v / (1 − β2^t)

        # 3. Learning rate warmup
        η_t = η_max * min(t / T_warmup, 1)

        # 4. Compute spectral penalty gradient
        grad_spec = SPECTRAL_PENALTY(θ)

        # 5. Compute curvature trigger
        λ = state.eigenvalues[θ]
        χ = CURVATURE_TRIGGER(λ, τ)

        # 6. Hopf tunneling (only if triggered)
        if χ == 1:
            φ = STABLE_HOPF_ANGLE(t)
            θ_tunnel = HOPF_TUNNEL( EXTRACT_2D_SLICE(θ), φ )
            hopf_correction = θ_tunnel − θ
        else:
            hopf_correction = 0

        # 7. Weight decay
        wd_term = weight_decay * θ

        # 8. Final update
        Δθ = −η_t * m_hat / (sqrt(v_hat) + eps)
              −η_t * grad_spec
              +η_t * hopf_correction
              −η_t * wd_term

        θ ← θ + Δθ
    return params, state
```

5.3 Complexity Analysis

- **Hopf Lift → O(n)** per parameter slice
- **Spectral Penalty → O(k n)** where *k* is sampled Hutchinson probe count
- **AdamW Moments → O(n)**
- **Overall per step → O(n + kn)** (k ≈ 4 in practice)

This remains practical for ViT-base and ViT-large models.

Next section: **6. Full PyTorch Implementation (final code)**.

```
# topo_geo_transformer_poc.py
# Full PyTorch implementation of Topo-Geo-AdamW optimizer + ViT training on
MNIST

import os
import math
import time
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.optimizer import Optimizer
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from transformers import AutoModelForImageClassification, AutoImageProcessor

# --------------------------- Utilities ---------------------------

def set_seed(seed=42):
    import random
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


# --------------------------- Topo-Geo-AdamW Optimizer
---------------------------
class TopoGeoAdamW(Optimizer):
    """
    AdamW-style optimizer augmented with:
      - geometric-normalized gradient preconditioning
      - spectral penalty (1/f style via FFT of parameter slices)
      - curvature-aware tunneling trigger using gradient norm + cosine flip
      - lightweight Hopf spinor rotation applied to small random 2D slices
    This implementation is intentionally pragmatic for PoC use.
    """

    def __init__(
        self,
        params,
        lr=1e-3,
        betas=(0.9, 0.999),
        eps=1e-8,
        weight_decay=0.01,
        phi=(1 + math.sqrt(5)) / 2,
        hopf_threshold=1e-3,
        hopf_prob=0.02,
        spectral_coeff=1e-4,
        warmup_steps=100,
    ):
        defaults = dict(
            lr=lr,
            betas=betas,
            eps=eps,
            weight_decay=weight_decay,
            phi=phi,
            hopf_threshold=hopf_threshold,
            hopf_prob=hopf_prob,
            spectral_coeff=spectral_coeff,
            warmup_steps=warmup_steps,
        )
        super().__init__(params, defaults)

    @staticmethod
    def _flatten(tensor):
        return tensor.view(-1)

    def _geometric_grad(self, grad, state):
        # approximate local preconditioning via RMS-style running avg stored in
state
        if 'rms' not in state:
            state['rms'] = torch.zeros_like(grad)
        state['rms'].mul_(0.99).addcmul_(grad, grad, value=0.01)
        rms = (state['rms'].sqrt() + 1e-6)
        g = grad / rms
        norm = g.norm() + 1e-12
        return g / norm

    def _spectral_penalty_grad(self, p):
```

```
# compute spectral penalty gradient via FFT magnitude deviation from 1/f shape
        x = self._flatten(p.data)
        n = x.shape[0]
        if n < 8:
            return torch.zeros_like(p.data)
        x_cpu = x.detach().cpu()
        # small practical FFT on CPU to avoid extra CUDA memory churn
        X = torch.fft.rfft(x_cpu)
        mag = X.abs()
        freqs = torch.linspace(1.0, mag.shape[0], mag.shape[0])
        ideal = 1.0 / freqs
        # normalize shapes
        mag_n = mag / (mag.mean() + 1e-12)
        ideal_n = ideal / (ideal.mean() + 1e-12)
        diff = (mag_n - ideal_n)
        # penalty in parameter space approximated by inverse FFT of scaled diff
        diff_complex = diff * torch.exp(1j * torch.angle(X))
        corr = torch.fft.irfft(diff_complex, n=n)
        grad = torch.from_numpy(corr.numpy()).to(p.device).view_as(p.data)
        return grad

    def _hopf_tunnel(self, p, state, angle):
        # select a small random 2D slice and apply 4D spinor rotation then
project back
        flat = self._flatten(p.data)
        n = flat.shape[0]
        if n < 4:
            return torch.zeros_like(p.data)
        # pick slice start
        start = torch.randint(0, max(1, n - 2), (1,)).item()
        # ensure 2 elements
        idx = torch.tensor([start, min(start + 1, n - 1)], dtype=torch.long,
device=p.device)
        slice_vals = flat[idx].clone()
        # create 4D spinor
        Ψ = torch.zeros((4,), device=p.device)
        Ψ[0] = slice_vals[0]
        Ψ[1] = slice_vals[1]
        # rotation matrix (block on first 2 components)
        cos_t = math.cos(angle)
        sin_t = math.sin(angle)
        R = torch.tensor(
            [[cos_t, -sin_t, 0, 0], [sin_t, cos_t, 0, 0], [0, 0, cos_t, -sin_t],
[0, 0, sin_t, cos_t]],
            device=p.device,
        )
        Ψ_rot = R @ Ψ
        projected = torch.zeros_like(flat)
        projected[idx] = Ψ_rot[:2]
        return projected.view_as(p.data)

    def step(self, closure=None, global_step=1):
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            lr = group['lr']
            betas = group['betas']
            eps = group['eps']
            wd = group['weight_decay']
            phi = group['phi']
            hopf_threshold = group['hopf_threshold']
            hopf_prob = group['hopf_prob']
            spectral_coeff = group['spectral_coeff']
            warmup_steps = group['warmup_steps']

            # warmup factor
            warmup_factor = min(1.0, global_step / float(max(1, warmup_steps)))

            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('TopoGeoAdamW does not support sparse
gradients')

                # init state
                state = self.state[p]
                if len(state) == 0:
                    state['step'] = 0
                    state['m'] = torch.zeros_like(p.data)
                    state['v'] = torch.zeros_like(p.data)

                state['step'] += 1

                # geometric grad
                g_geo = self._geometric_grad(grad, state)

                # AdamW moments (apply on geometric grad)
                state['m'].mul_(betas[0]).add_(g_geo, alpha=(1 - betas[0]))
                state['v'].mul_(betas[1]).addcmul_(g_geo, g_geo, value=(1 -
betas[1]))

                m_hat = state['m'] / (1 - betas[0] ** state['step'])
                v_hat = state['v'] / (1 - betas[1] ** state['step'])

                # spectral penalty (cheap heuristic)
                try:
                    grad_spec = self._spectral_penalty_grad(p)
                except Exception:
                    grad_spec = torch.zeros_like(p.data)

                # curvature-aware trigger (norm small and cosine flip)
                prev_grad = state.get('prev_grad')
                grad_norm = grad.norm()
                trigger = False
                if prev_grad is not None:
                    cos_sim = torch.dot(self._flatten(prev_grad),
self._flatten(grad)) / (
                        (self._flatten(prev_grad).norm() + 1e-12) *
(self._flatten(grad).norm() + 1e-12)
                    )
                    if grad_norm < hopf_threshold and cos_sim < -0.1:
                        trigger = True
                state['prev_grad'] = grad.clone()

                hopf_correction = torch.zeros_like(p.data)
                if trigger and (torch.rand(1).item() < hopf_prob):
                    angle = (math.pi / 4) * math.exp(-state['step'] / 1000.0) +
1e-3
                    hopf_correction = self._hopf_tunnel(p, state, angle)

                # adam update direction
                step_direction = m_hat / (v_hat.sqrt() + eps)

                # weight decay
                if wd != 0:
                    step_direction = step_direction + wd * p.data

                # combine updates
                update = -lr * warmup_factor * step_direction
                update = update - lr * warmup_factor * spectral_coeff *
grad_spec.to(p.device)
                update = update + lr * warmup_factor * hopf_correction

                p.data.add_(update)

        return loss


# --------------------------- Training Loop ---------------------------
```

```
def build_dataloaders(data_dir='./data', batch_size=16):
    model_name = 'google/vit-base-patch16-224'
    processor = AutoImageProcessor.from_pretrained(model_name)
    transform = transforms.Compose([
        transforms.Grayscale(num_output_channels=3),
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=processor.image_mean,
std=processor.image_std),
    ])

    train_ds = datasets.MNIST(root=data_dir, train=True, download=True,
transform=transform)
    test_ds = datasets.MNIST(root=data_dir, train=False, download=True,
transform=transform)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,
num_workers=2)
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,
num_workers=2)
    return train_loader, test_loader


def train_one_epoch(model, optimizer, loader, device, epoch, max_grad_norm=1.0):
    model.train()
    total_loss = 0.0
    criterion = nn.CrossEntropyLoss()
    for step, (x, y) in enumerate(loader):
        x = x.to(device)
        y = y.to(device)
        optimizer.zero_grad()
        out = model(pixel_values=x).logits
        loss = criterion(out, y)
        loss.backward()
        # gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
        optimizer.step(global_step=epoch * len(loader) + step + 1)
        total_loss += loss.item()
        if step % 200 == 0:
            print(f"Epoch {epoch} Step {step} Loss {loss.item():.4f}")
    return total_loss / len(loader)


@torch.no_grad()
def evaluate(model, loader, device):
    model.eval()
    correct = 0
    total = 0
    for x, y in loader:
        x = x.to(device)
        y = y.to(device)
        logits = model(pixel_values=x).logits
        preds = logits.argmax(dim=1)
        correct += (preds == y).sum().item()
        total += y.size(0)
    return 100.0 * correct / total


def main():
    set_seed(42)
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    train_loader, test_loader = build_dataloaders(batch_size=16)

    model_name = 'google/vit-base-patch16-224'
    model = AutoModelForImageClassification.from_pretrained(model_name,
num_labels=10, ignore_mismatched_sizes=True)
    model.to(device)

    # Baseline AdamW for comparison
    base_optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=0.01)

    # Experimental Topo-Geo-AdamW
    topo_optimizer = TopoGeoAdamW(
        model.parameters(),
        lr=2e-4,
        betas=(0.9, 0.999),
        weight_decay=0.01,
        hopf_threshold=1e-4,
        hopf_prob=0.03,
        spectral_coeff=5e-5,
        warmup_steps=200,
    )

    # choose optimizer to run
    optimizer = topo_optimizer

    epochs = 3
    for epoch in range(1, epochs + 1):
        t0 = time.time()
        avg_loss = train_one_epoch(model, optimizer, train_loader, device,
epoch)
        acc = evaluate(model, test_loader, device)
        print(f"Epoch {epoch} | Loss {avg_loss:.4f} | Test Acc {acc:.2f}% |
Time {time.time() - t0:.1f}s")

    # save model
    model.save_pretrained('./topo_vit_mnist')


if __name__ == '__main__':
    main()
```

Notes: - This file is a PoC: create a virtualenv, `pip install torch torchvision transformers datasets`, and run. - The spectral penalty uses a practical FFT heuristic rather than a full Hessian spectral estimate to reduce overhead. - Hopf tunneling is applied sparsely to small 2D slices to avoid excessive parameter distortion. - For rigorous research comparisons, add AdamW baseline runs, repeat random seeds, and use CIFAR-10 for better ViT benchmarking.

```
```