SIRC Torch-Pruning v2 (STP v2)

Internal Technical Report

February 2026

Abstract

Fork of Torch-Pruning extended with Pruning-Aware Training (PAT), new importance criteria (VBP, MAC-Aware), bias compensation, and regularization strategies.

Contents

1 Theoretical Foundations

This part presents the theoretical framework behind structured channel pruning in STP v2. We begin with the dependency-aware pruning mechanism, then introduce Pruning-Aware Training (PAT) — the main contribution — followed by pluggable importance criteria, regularization, and post-pruning compensation.

1.1 Structured Pruning Mechanism

1.1.1 Overview

Structured pruning removes entire channels (or filters) from a network, reducing both parameter count and computational cost. Unlike unstructured (weight-level) pruning, structured pruning produces dense models that run efficiently on standard hardware without specialized sparse kernels.

The pruning engine follows five core steps plus an optional compensation step (Figure~1):

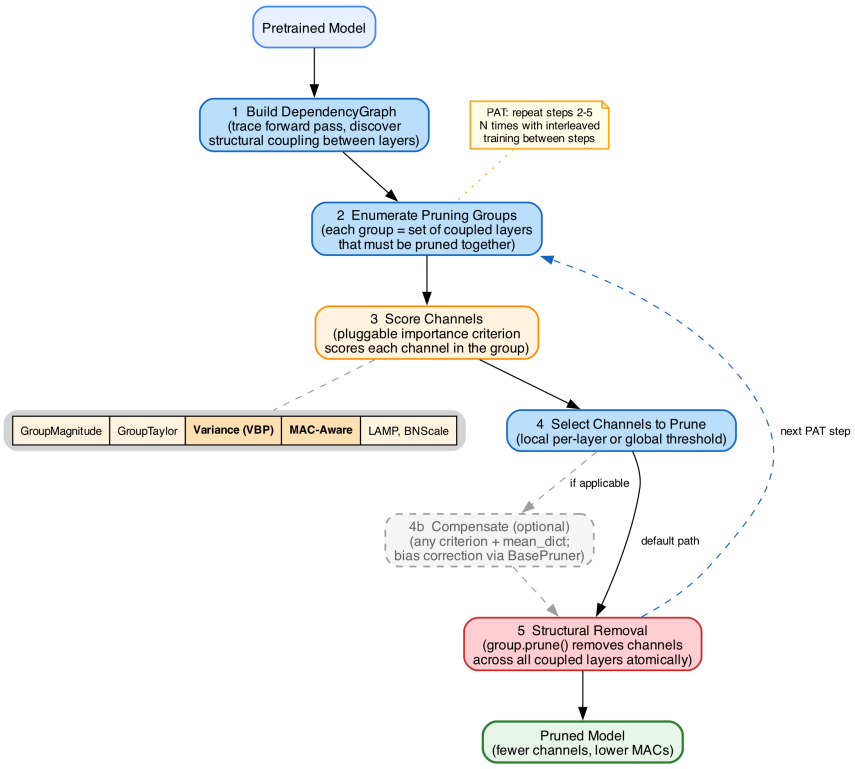

Figure 1: Pruning pipeline. Five core steps from pretrained model to pruned model, plus an optional compensation step for criteria that collect activation statistics. The dashed loop indicates PAT iterations (Section 1.2). Criteria highlighted in bold are new in STP v2.

Step 1: Build DependencyGraph. `DependencyGraph.build_dependency(model, example_inputs)` traces a dummy forward pass and records how layers are structurally coupled: if a Conv2d's output channels are pruned, the downstream BatchNorm and the next Conv2d's input channels must be pruned in lockstep.

Step 2: Enumerate Pruning Groups. `DG.get_all_groups()` yields `Group` objects — sets of coupled (layer, channel-indices) pairs that must be pruned together. This is the core contribution

of DepGraph [Fang et al., CVPR 2023]: automatic dependency discovery replaces manual group definitions.

Step 3: Score Channels. The importance criterion's `__call__(group)` method returns a score vector $\mathbb{R}^C$. The criterion is pluggable — `GroupMagnitudeImportance`, `VarianceImportance`, or any custom subclass can be swapped at initialization (Section~1.3).

Step 4: Select Channels. Channels below a threshold are marked for removal. The threshold can be local (per-layer ratio) or global (single threshold across all groups). Steps 2–4 are orchestrated by `BasePruner._prune()`, which is called from `BasePruner.step()`.

Step 4b: Compensate *(optional).* When a `mean_dict` is provided (activation means from a calibration pass), `BasePruner._apply_compensation(group, idxs)` corrects consumer biases before the structural cut (Section~1.5). This is criterion-agnostic — any pruner (magnitude, LAMP, VBP) can opt in. Without `mean_dict`, this step is skipped.

Step 5: Structural Removal. `Group.prune(idxs)` removes weight rows/columns, adjusts biases, and updates normalization parameters atomically across all coupled layers.

In PAT mode (Section~1.2), steps 2–5 repeat N times with interleaved training between steps. The `ChannelPruning` wrapper (Section~3.2) calls `step()` at the right epochs automatically.

1.2 Pruning-Aware Training (PAT)

PAT is the central contribution of STP v2. It wraps the single-pass pruning engine (Section~1.1) behind a thin interface and integrates it into a standard training loop, enabling iterative prune-then-train cycles that preserve accuracy far better than one-shot pruning.

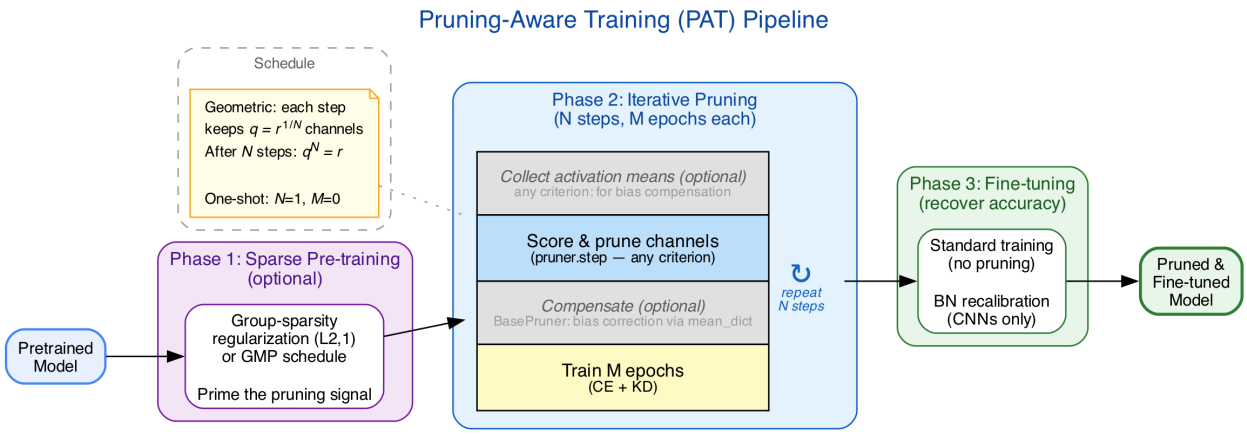

Figure 2: PAT pipeline. Three phases (all optional), with a repeated pruning loop in Phase 2. Grey rows in Phase 2 are criterion-specific (apply only to VBP or other statistics-based criteria); the blue and yellow rows are shared by all criteria. One-shot mode collapses Phase 2 to a single step with $M = 0$.

Phase 1: Sparse Pre-training *(optional).* Before any structural pruning, the model is trained with a group-sparsity regularizer ($L_{2,1}$ or GMP) that pushes unimportant channels toward zero. This primes the importance signal so that the pruner can make better decisions in Phase~2. Skipping this phase is valid but may reduce pruning quality at aggressive ratios.

Phase 2: Iterative Pruning. The core PAT loop, repeated N times:

1. *Collect stats* (optional) — criteria that rely on activation statistics (e.g., VBP) run a calibration forward pass to gather per-channel means and variances. Weight-only criteria (magnitude, LAMP) skip this.
2. *Score & prune channels* — the importance criterion scores every channel, a threshold selects the bottom fraction, and the pruner structurally removes them (Section 1.1, steps 3–5).
3. *Compensate* (optional) — VBP applies bias correction and BN variance update before the structural cut (Section 1.5). Other criteria skip this.
4. *Train M epochs* — standard training (cross-entropy, optionally knowledge distillation) to recover accuracy before the next pruning step.

Each step retains a fraction $q = r^{1/N}$ of channels, so after N steps the cumulative retention is $q^N = r$. One-shot mode is the degenerate case: $N = 1$, $M = 0$.

Phase 3: Fine-tuning. After all pruning steps, the pruned model is trained for several more epochs with no further pruning. For CNNs, BN running statistics are recalibrated at the start of this phase (Section~1.5.3).

1.2.1 Pruning Schedules

Rather than reimplementing a schedule externally, PAT leverages the upstream `BasePruner`'s built-in `iterative_steps` parameter. The pruner is created once with a target ratio and N iterative steps; each call to `step()` applies the next increment automatically. The schedule determines how the target ratio r is distributed across these N steps:

Linear (upstream default). The target pruning ratio $p = 1 - r$ is divided equally across N steps: step t prunes to cumulative ratio $p \cdot t/N$. Simple, but early steps are disproportionately aggressive — removing p/N channels from a full model is a smaller relative cut than removing p/N from an already-pruned one.

Geometric. Each step retains a constant fraction q of the *current* channels, so that the cumulative effect equals the target:

$$q = r^{1/N}, \qquad q^N = r$$

This ensures equal *proportional* pruning at each step, making each cut equally disruptive relative to the remaining capacity. Preferred over linear for aggressive ratios.

One-shot. The degenerate case of either schedule with $N = 1$: no interleaved training ($M = 0$), followed by fine-tuning only. Simpler and faster, but less accurate at aggressive ratios.

1.2.2 Training Loop Integration

The PAT wrapper exposes two calls — `regularize()` and `prune()` — to an otherwise unchanged training loop. Based on the current epoch and configuration (`start_epoch`, `end_epoch`, `epoch_rate`), the wrapper decides internally whether to apply regularization, trigger a pruning step, or do nothing (fine-tuning phase). The chosen schedule (linear or geometric) only affects the ratio applied at each step; the epoch-based control is the same regardless.

1.3 Channel Importance Criteria

1.3.1 Group-Aware Importance (DepGraph Contribution)

Prior work [Li et al., ICLR 2017] computed importance per-layer: $I_c = \|\mathbf{w}_c\|_p$ for each layer independently. This ignores inter-layer dependencies — pruning channel c from a Conv2d also affects the downstream BN and the next layer's input, but per-layer scoring does not account for this.

DepGraph introduced **group-level importance aggregation**. For a pruning group containing K coupled layers, each layer k contributes a local importance vector $\mathbf{i}^{(k)} \in \mathbb{R}^C$. These are aggregated via a configurable reduction over a shared *root channel space* (mapped via the DG's index tracking):

$$I_c = \text{reduce}\Big(\Big\{i_c^{(1)}, i_c^{(2)}, \ldots, i_c^{(K)}\Big\}\Big)$$

Available reductions include **mean** (default, $I_c = \frac{1}{K}\sum_k i_c^{(k)}$), **max**, **prod**, and **first/gate** (use only one layer in the group). This group reduction is orthogonal to the base criterion — it applies equally to magnitude, Taylor, Hessian, and VBP importance.

1.3.2 Magnitude-Based Importance

The base criterion computes per-channel weight norms. For layer m with weight tensor $W^{(m)}$:

$$i_c^{(m)} = \left\|\mathbf{w}_c^{(m)}\right\|_p$$

where $\mathbf{w}_c$ is the c-th output channel flattened to a vector ($\mathbb{R}^{C_{\text{in}}}$ for Linear, $\mathbb{R}^{C_{\text{in}} \cdot k^2}$ for Conv2d). Input-channel pruning transposes the weight before computing norms. BatchNorm/LayerNorm affine weights contribute their scale parameters.

With group aggregation (Section 1.3.1), this becomes `GroupMagnitudeImportance`: each coupled layer contributes its local magnitude, and the group reduction produces a single consensus score per channel. This is fast (no data required), but ignores activation statistics and gradient information.

Post-reduction normalization can further adjust scores: per-layer mean normalization, max normalization, Gaussian z-scores, or LAMP (Layer-Adaptive Magnitude Pruning) weighting.

1.3.3 Variance-Based Pruning (VBP)

VBP [arXiv 2507.12988] uses the variance of **post-activation outputs** as importance. A channel whose activations have near-zero variance produces a nearly constant signal, contributing little information to downstream layers.

Statistics collection. Forward hooks on target layers accumulate running sums $\sum x_c$ and $\sum x_c^2$ across a calibration set. After N total observations:

$$\mu_c = \frac{1}{N}\sum x_c, \qquad \sigma_c^2 = \frac{1}{N}\sum x_c^2 - \mu_c^2$$

The importance score for channel c is $I_c = \sigma_c^2$. Lower variance → lower importance → pruned first. The stored means μ_c are also used for bias compensation (Section~1.5).

Spatial reduction. For Conv2d feature maps (B, C, H, W), each spatial position is an independent observation ($N = B \cdot H \cdot W$). For Linear outputs (B, T, C), sequence positions play the same role ($N = B \cdot T$). The kernel size and spatial resolution affect only the sample count, not the dimensionality of the score.

Architecture-specific hooking. Statistics are collected after the activation function (post-GELU for ViT fc1 layers, post-BN+ReLU for CNN convolutions) via composable `post_act_fn` hooks.

1.3.4 MAC-Aware Importance

MAC-Aware importance is designed as a **composite criterion** that balances structural importance (accuracy) against computational cost (efficiency). The general formulation combines an accuracy-driven term with a cost-weighted term:

$$I_c = \alpha \cdot \left\| I_c^{\text{base}} \right\| + (1 - \alpha) \cdot \left(\frac{\text{cost}_m}{\text{cost}_{\max}} \right)^{\beta} \qquad \text{(additive mode)}$$

$$I_c = I_c^{\text{base}} \cdot \left(\frac{\text{cost}_m}{\text{cost}_{\max}} \right)^{\beta} \qquad \text{(multiplicative mode)}$$

Parameters α (importance vs. cost trade-off) and β (cost exponent) control the balance.

Current implementation uses MACs as the cost metric, but the cost term is intentionally designed to be replaceable. In deployment scenarios, MACs can be substituted with measured GPU latency, power consumption, memory bandwidth, or any hardware-specific PPA (Power–Performance–Area) metric — the formulation remains the same. This makes the criterion hardware-agnostic: optimize for whatever efficiency metric matters for the target platform.

1.4 Regularization for Structured Pruning

Structured pruning removes entire channels. For this to work well, the model should be trained so that unimportant channels are clearly distinguishable — ideally near-zero. Regularization during training achieves this by adding a penalty term to the loss.

1.4.1 DepGraph Group Regularization

DepGraph [Fang et al., CVPR 2023, Section 3.3, Eq.~4–5] defines an $L_{2,1}$ norm over pruning groups. For group G_k containing all coupled layers, let $\mathbf{g}_k^{(i)}$ be the concatenated parameters of the i-th channel across all layers in the group:

$$\mathcal{L}_{\text{reg}} = \sum_k \sum_{i=1}^{C_k} \left\| \mathbf{g}_k^{(i)} \right\|_2$$

This is L_2 within each channel (preserving internal structure), L_1 across channels (encouraging entire channels to zero) — the standard *group sparsity* formulation.

1.4.2 $L_{2,1}$ Weight Regularization

Our implementation applies the same $L_{2,1}$ norm, restricted to the prunable layers (MLP expansion or interior convolutions):

$$\mathcal{L}_{L_{2,1}} = \sum_{m} \sum_{i=1}^{C_{\text{out}}} \left\| \mathbf{w}_i^{(m)} \right\|_2$$

where $\mathbf{w}_i^{(m)}$ is the i-th output channel of layer m, flattened to a vector. For a Linear layer, $\mathbf{w}_i \in \mathbb{R}^{C_{\text{in}}}$; for a Conv2d layer with kernel size k, $\mathbf{w}_i \in \mathbb{R}^{C_{\text{in}} \cdot k^2}$. The total loss becomes:

$$\mathcal{L} = \mathcal{L}_{\text{CE}} + \lambda \cdot \mathcal{L}_{L_{2,1}}$$

This operates in **weight space** and is agnostic to the pruning criterion.

1.4.3 Variance Entropy Regularization

Designed specifically for VBP. Instead of penalizing weights, it shapes the **activation variance distribution** to sharpen the VBP importance signal.

For each target layer m with output activations $\mathbf{a}$, compute the per-channel variance σ_c^2 (across batch and spatial/sequence dimensions), normalize into a probability distribution, and compute its entropy:

$$p_c = \frac{\sigma_c^2}{\sum_j \sigma_j^2}, \qquad H_m = -\sum_c p_c \log p_c, \qquad \mathcal{L}_{\text{var}} = \sum_m H_m$$

Entropy is maximal when all channels have equal variance (uniform p) and minimal when variance is concentrated in few channels. **Minimizing** $\mathcal{L}_{\text{var}}$ forces the model to concentrate its representational capacity into fewer channels, making the remaining low-variance channels clearly expendable.

1.4.4 Comparison

Table 1: Comparison of regularization strategies.

	$L_{2,1}$ **Weight Regularization**	**Variance Entropy Regularization**
Domain	Weight space	Activation space
Mechanism	Group sparsity (push channels $\to 0$)	Entropy minimization (concentrate variance)
Alignment	Agnostic (benefits magnitude pruning)	Directly aligned with VBP
Phase	Sparse pre-training (before pruning)	PAT fine-tuning (during pruning)

1.5 Post-Pruning Compensation

1.5.1 Bias Compensation

When channel j is pruned from a layer's output, every downstream consumer loses that channel's contribution. For a consumer computing $y_i = \sum_j W_{ij} x_j + b_i$, the output after pruning channel set P becomes:

$$y_i^{\text{pruned}} = \sum_{j \notin P} W_{ij}\, x_j + b_i$$

The missing term has expected value $\sum_{j \in P} W_{ij}\, \mu_j$, where $\mu_j = \mathbb{E}[x_j]$ is the mean activation of the pruned channel (collected via a calibration forward pass). Compensation absorbs this into the bias:

$$b_i^{\text{new}} = b_i + \sum_{j \in P} W_{ij}\, \mu_j$$

This is implemented in `BasePruner._apply_compensation()` and is **criterion-agnostic**: any pruner (magnitude, LAMP, Taylor, VBP) can opt in by passing a `mean_dict` mapping modules to per-channel activation means. A standalone `collect_activation_means()` utility collects these means from a calibration set without requiring VBP's full variance statistics. When no `mean_dict` is provided, compensation is skipped with zero overhead.

This restores the correct **expected output**. However, the **variance** is not restored — the fluctuations $\sum_{j \in P} W_{ij}(x_j - \mu_j)$ are permanently lost:

$$\text{Var}\left(y_i^{\text{pruned}}\right) = \text{Var}(y_i) - \sum_{j \in P} W_{ij}^2\, \sigma_j^2$$

(assuming independent channels). This variance shift is the root cause of the CNN-specific normalization problem.

1.5.2 The BatchNorm Problem in CNNs

Why ViTs are resilient. In a ViT, every Linear layer is followed by **LayerNorm**, which computes normalization statistics from the current input at inference time:

$$\hat{x}_c = \frac{x_c - \mu_{\text{LN}}}{\sigma_{\text{LN}}}, \qquad \mu_{\text{LN}} = \frac{1}{C}\sum_c x_c, \quad \sigma_{\text{LN}}^2 = \frac{1}{C}\sum_c (x_c - \mu_{\text{LN}})^2$$

After pruning and bias compensation, the mean is corrected and the variance has shifted — but LayerNorm recomputes μ_{LN} and σ_{LN} from whatever the pruned network actually produces. It **automatically adapts**.

Why CNNs break. In a CNN, Conv2d layers are followed by **BatchNorm**, which normalizes using **stored population statistics** accumulated during the original training:

$$\hat{x}_c = \frac{x_c - \hat{\mu}_c}{\sqrt{\hat{\sigma}_c^2 + \epsilon}}$$

where $\hat{\mu}_c$ and $\hat{\sigma}_c^2$ are exponential moving averages frozen at inference. After pruning:

1. $\hat{\mu}_c$ **is stale** — bias compensation corrects the consumer's bias, but BN sits between producer and consumer. The actual per-channel mean has changed; BN still subtracts the old mean.
2. $\hat{\sigma}_c^2$ **is stale** — the variance reduction from pruning means the true activation variance is now smaller, but BN divides by the old (larger) standard deviation, *compressing* the signal.

3. **Errors compound** — each BN layer introduces a mean/variance mismatch. In a deep network these errors accumulate through the residual stream, leading to catastrophic accuracy collapse.

This is not specific to VBP — *any* structured pruning criterion applied to CNNs suffers from stale BN statistics.

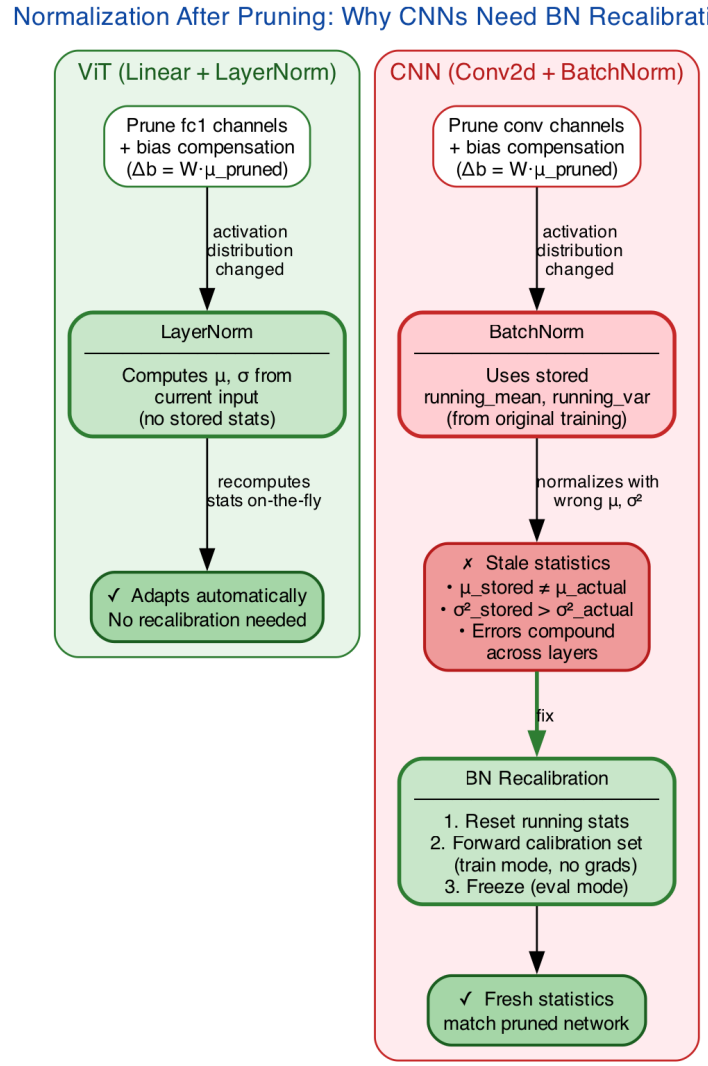

Figure 3: Normalization after pruning. **Left:** ViT with LayerNorm — statistics are recomputed on-the-fly, adapting automatically. **Right:** CNN with BatchNorm — stored running statistics become stale, requiring explicit recalibration.

1.5.3 BN Recalibration

The fix is empirical recalibration: after pruning, forward a calibration set through the pruned network with BN layers in training mode (recomputing statistics) but no gradient updates:

1. **Reset** all BN running statistics: $\hat{\mu} \leftarrow 0$, ; $\hat{\sigma}^2 \leftarrow 1$.
2. **Forward** K calibration batches in training mode (no backpropagation).

3. BN layers accumulate fresh $\hat{\mu}$ and $\hat{\sigma}^2$ reflecting the pruned network's actual activation distribution.
4. **Freeze** by switching to evaluation mode.

The reset step is critical — without it, BN's exponential moving average blends new (correct) statistics with old (stale) ones, and convergence is slow. With reset, a few hundred batches suffice for ResNet-50; MobileNetV2's narrow bottleneck layers need $\sim$5,000 samples.

2 STP v2 Implementation

This part summarizes the engineering changes in STP v2 relative to the upstream Torch-Pruning library.

2.1 Upstream Rebase

STP was rebased from Torch-Pruning v1.5.1 to **v1.6.1**, gaining a refactored dependency module with cleaner Node/Group/DependencyGraph separation and improved isomorphic and multi-head attention pruning support.

2.2 ChannelPruning Simplification

Major refactor of the pruning scheduler — both code cleanup and an architectural simplification.

Code cleanup. PEP~8 class naming (`channel_pruning` → `ChannelPruning`), a `PruningMethod` enum replacing magic strings, a unified logging helper, and bug fixes for `ignored_layers` accumulation and hardcoded device references.

Scheduling rewrite. Previously, the pruner was recreated every epoch with custom rate computation, MAC-target flags, optimizer resets, and JSON config writing. Now the pruner is created **once at initialization** with a fixed number of iterative steps, and each pruning epoch calls a single `step()`. The upstream linear scheduler splits the target ratio across steps automatically.

Regularization cleanup. The `regularize()` method was simplified: a single `epoch > end_epoch` guard replaces scattered conditionals, and the separate L1 group sparse pre-training code was removed (now handled by the training script). `GroupNormPruner` now uses `GroupMagnitudeImportance` internally (fixing an AttributeError with the non-existent `GroupNormImportance`), and `ConvTranspose2d` layers are skipped during regularization to avoid shape mismatches.

Regularization loss visibility. TP's regularization works by modifying gradients directly (`param.grad += ...`), not by adding a loss term. To enable separate logging of the regularization magnitude, `regularize()` now accumulates and returns the L_1 norm of the gradient modifications as a scalar. Callers can log this alongside the task loss without changing the optimization.

MAC-target estimation. An analytical conversion from MAC retention target to channel keep ratio was added. Each layer's MACs are classified as unpruned (U), 1-dimensional (S_1, only input or output pruned), or 2-dimensional (S_2, both pruned). The target equation:

$$\text{mac\_target} = \frac{U + S_1 \cdot q + S_2 \cdot q^2}{T}, \qquad q = 1 - p \text{ (keep ratio)}$$

For ViT (all 1-dim): linear solution $p = 1 - \text{mac\_target}$. For CNNs with 2-dim middle layers: quadratic formula. Called once at initialization; the pruner's iterative steps handle the splitting.

Constructor `train_loader` parameter. The `Pruning` and `ChannelPruning` constructors accept an optional `train_loader`, enabling statistics collection and compensation mean gathering in a single pass during initialization. A `_stats_fresh` flag prevents redundant re-collection on the first `prune()` call.

One-shot / PAT epoch config. The `vbp_imagenet_pat.py` config builder handles `pat_steps` $\leq$ `1` as one-shot (`end_epoch = start_epoch`, `iterative_steps = 1`) and `pat_steps > 1` as PAT

(`end_epoch = start_epoch + (pat_steps - 1) × epoch_rate`), ensuring `iterative_steps` exactly equals `pat_steps`.

2.3 Transformer Support

The original pruning utilities only handled Conv2d and BatchNorm. Extensions include:

- **Linear layer pruning** — proper in/out channel handling for `nn.Linear`.
- **LayerNorm pruning** — coupled with Linear in dependency groups (ViT uses LayerNorm before/after attention and MLP blocks).

Future work: MHA head pruning (Q/K/V/output projection simultaneously), embedding dimension pruning, and joint MLP + attention pruning modes.

2.4 Generalized Bias Compensation

Bias compensation was originally implemented inside `VBPPruner`. It has been **lifted to `BasePruner`** so that any pruning criterion can benefit from it:

- `BasePruner._apply_compensation(group, idxs)` handles Linear, Conv2d (standard and depthwise) consumers.
- Activated by passing `mean_dict` at construction or via `set_mean_dict()`.
- `collect_activation_means()` provides a standalone calibration utility (no VBP dependency).
- `ChannelPruning` auto-collects means when a `train_loader` is available, regardless of criterion.
- `VBPPruner` now inherits compensation from `BasePruner` and only adds mean-check diagnostics and BN variance updates.

2.4.1 Auto-Detected Target Layers

Target layers for statistics collection are now auto-detected by walking the `DependencyGraph` rather than requiring architecture-specific code. `build_target_layers()` discovers Conv2d→BN→Act and Linear→Act patterns automatically, supporting CNNs, ViTs, and ConvNeXt with a single code path.

2.4.2 CNN-Specific Challenges

Tested on ResNet-50 and MobileNetV2. Key challenges resolved:

- **BN recalibration** — running statistics must be explicitly reset before recalibrating (Section~1.5.3).
- **Depthwise conv group roots** — in MobileNetV2, the depthwise conv is the group root but activation statistics reside on the expand conv. The compensation routine searches the group for a module with matching statistics.
- **Depthwise convs and ignored layers** — depthwise convs must not be in the ignored-layers list, as they appear in expand conv groups with output-channel pruning, causing group rejection.
- **ReLU6 detection** — `nn.ReLU6` uses `HardtanhBackward0` internally, requiring an explicit mapping in activation auto-detection.

2.4.3 Unified Pruning Loop

All criteria (VBP, magnitude, LAMP, random) now share a single interactive pruning loop. Previously VBP had a separate non-interactive code path; now all criteria use the same loop with identical per-group logging ("Prune/Mask N/M channels" with a compensation indicator). VBP mean-check setup/teardown wraps the shared loop conditionally. This also fixed a bug where compensation was missing in the non-VBP physical pruning path — `_apply_compensation()` now runs before both mask and prune branches.

2.4.4 Multi-Criterion Support

The `PruningMethod` enum was extended with `LAMP` and `RANDOM` values. `init_channel_pruner()` branches on the enum to instantiate the appropriate importance (`GroupMagnitudeImportance`, `LAMPImportance`, or random). Both `vbp_imagenet.py` and `vbp_imagenet_pat.py` accept `-criterion` to select the criterion; VBP-specific logic (stats collection, variance loss) is automatically gated on `criterion=variance`.

2.5 Mask-Then-Prune Strategy

In PAT mode, intermediate pruning steps now use **mask-only mode** (zeroing channels but keeping the architecture intact), which allows the optimizer state and learning rate schedule to remain consistent across steps. The **final step** switches to physical structural removal, so fine-tuning operates on the actually smaller model with reduced MACs.

MAC logging. Mask-only and physical steps require different MAC measurement. Standard `count_ops_and_params` counts ops from tensor shapes, so zeroed channels still register as full MACs. Mask steps therefore use `measure_macs_masked_model()` which detects zeroed channels and subtracts their contribution. The final physical step compares current absolute MACs against the stored original. Both paths report a consistent ratio relative to the unpruned model.

2.6 Dependency Graph Visualization

Graphviz-based visualization of the DependencyGraph and pruning groups, providing three complementary views:

- **Computational graph** — data flow through the model.
- **Dependency graph** — pruning coupling between layers (direct dependencies vs. forced shape matches).
- **Combined** — both overlaid with consistent node layout.

Features: group cluster boxes, color-coded nodes by operation type, multi-group detection, and consistent layout across views. Output formats: PNG, SVG, PDF.