A Scalar Regression Pipeline Based on DINOv3

Code: LinkinPony/CSIRO

Abstract

This solution is driven by a simple premise: as visual foundation models like DINOv2/DINOv3 mature, many downstream problems can be addressed by **reusing a strong, general-purpose backbone** and learning only a **lightweight task head** (optionally with PEFT/adapters).

Concretely, we treat this biomass regression task as "representation + readout": extract DINOv3 patch-token features and train a small regression head on top. We optionally adapt the backbone via LoRA while keeping pretrained weights frozen. Our final submission reached 0.78 (rank 8) on the public leaderboard and 0.65 (rank 6) on the private leaderboard; our best single submission achieved 0.76/0.69 on public/private.

1. Introduction

1.1 Motivation and problem setting

Estimating pasture biomass is a recurring decision point in grazing management: it affects when to graze, how intensely to stock, and how to maintain pasture productivity over seasons. However, **accurate biomass measurement is expensive and operationally difficult at scale**. Traditional approaches like clip-and-weigh are reliable but slow; field instruments such as plate or capacitance meters are faster yet can be noisy under variable conditions; remote sensing provides broad coverage but typically requires ground calibration and may not separate biomass into fine-grained components.

This work targets the CSIRO Pasture Biomass task: given a pasture image (optionally with auxiliary supervision such as NDVI/height during training), predict five biomass components in **grams**: `Dry_Green_g`, `Dry_Dead_g`, `Dry_Clover_g`, `GDM_g`, and `Dry_Total_g`. Performance is evaluated using a **weighted average** R^2 across targets, computed in a **log-transformed space** (the repository follows `log1p(clamp(x, min=0))`), with `Dry_Total_g` carrying the largest weight.

Our approach follows a simple "representation + readout" premise: modern self-supervised visual foundation models (DINOv3) provide strong, general-purpose representations that transfer well even when labeled data is limited. Instead of end-to-end finetuning of the full backbone, we primarily train a **lightweight regression head** on top of frozen DINOv3 features, optionally enabling **parameter-efficient finetuning (LoRA)** for the last few transformer blocks. This design keeps the system modular, compute-efficient, and amenable to rapid iteration over head architectures, objectives, and inference-time ensembling.

To further improve generalization on a small, noisy dataset, we also apply **manifold mixup** in feature space: during training, we linearly interpolate DINOv3 representations (patch tokens in our final run) between samples and mix supervision targets in a mask-aware way (§3.4.1).

1.2 Contributions

Concretely, this repository contributes:

- **A DINOv3 feature-extraction backbone with reproducible loading**: the backbone can be resolved from a vendored local copy (for offline runs) or via `torch.hub`, and supports extracting both last-layer and intermediate-layer features for multi-layer heads. See `src/models/backbone.py`.

- **Optional PEFT/LoRA finetuning for ViT backbones**: LoRA adapters are injected into selected attention/MLP projections (typically in the last K blocks), with a "train adapters only" policy and optional EVA initialization support. See `src/models/peft_integration.py`.
- **A unified regression model with multiple lightweight heads**: a single training stack supports different readout heads (MLP / ViTDet-style pyramid / FPN / Mamba / DPT / EoMT), patch-aware modes, and multi-layer feature fusion, enabling systematic ablations under consistent data and optimization settings. See `src/models/regressor/biomass_regressor.py` and `src/models/regressor/heads/`.
- **Feature-space augmentation via manifold mixup**: we apply mixup directly on the shared representation (patch tokens in our final run) and mix regression/ratio targets consistently (including missing-label masks), providing a strong regularizer without pixel-level mixing. See `src/models/regressor/manifold_mixup.py`.
- **Practical weights management for deployment**: backbone weights and head weights are treated as separate artifacts (head-only checkpoints and packaging), enabling fast swapping and ensembling without duplicating large backbone checkpoints. See `src/callbacks/head_checkpoint.py`, `package_artifacts.py`, and `README.md`.
- **An end-to-end inference pipeline for competition submission**: inference loads backbone + head weights, supports test-time augmentation with ViT patch-grid alignment, optional ensembling (configured in JSON), and writes the required long-format `submission.csv`. See `infer_and_submit_pt.py`, `src/inference/pipeline.py`, and `configs/ensemble.json`.

2. Task, Data, and Metric

2.1 Targets, identities, and a constraint-preserving parameterization

The competition requires predicting five dry-mass quantities (in grams) per image:

- `Dry_Clover_g`: dry mass of clover biomass.
- `Dry_Green_g`: dry mass of green vegetation **excluding clover** (primarily grasses).
- `Dry_Dead_g`: dry mass of dead material (senesced/litter).
- `GDM_g`: green dry matter (total green biomass).
- `Dry_Total_g`: total dry biomass.

These targets are not independent. By definition,

$$\begin{aligned}\mathrm{GDM}_g &= \mathrm{Dry\_Clover}_g + \mathrm{Dry\_Green}_g \\ \mathrm{Dry\_Total}_g &= \mathrm{Dry\_Clover}_g + \mathrm{Dry\_Green}_g + \mathrm{Dry\_Dead}_g\end{aligned}$$

To make model outputs satisfy these identities **by construction**, the training configuration documented in this writeup
(`configs/train.yaml`) parameterizes biomass using:

- **Total** t: we directly supervise only `Dry_Total_g` (`data.target_order: [Dry_Total_g]`).
- **Composition ratio** p: a 3-way proportion vector over **(clover, dead, green)**. In the dataset, when all four components are available and $t > 0$, we define:

$$p = \begin{bmatrix} \frac{\text{Dry\_Clover}_g}{t} \\ \frac{\text{Dry\_Dead}_g}{t} \\ \frac{\text{Dry\_Green}_g}{t} \end{bmatrix}.$$

Given (t, p), we reconstruct the canonical 5D vector (repo order `[Dry_Clover_g, Dry_Dead_g, Dry_Green_g, GDM_g, Dry_Total_g]`) as:

$$\begin{aligned} \hat{c} &= p_0 t, \\ \hat{d} &= p_1 t, \\ \hat{g} &= p_2 t, \\ \widehat{\text{GDM}} &= \hat{c} + \hat{g}, \\ \widehat{\text{Total}} &= \hat{c} + \hat{d} + \hat{g}. \end{aligned}$$

This lets us train and evaluate on the official 5D targets while keeping predictions consistent without adding an extra
constraint loss.

Repo anchors

- Ratio targets + masks from raw CSV labels: `src/data/datamodule.py`
- 5D reconstruction and constraint-aware fusion (training losses): `src/models/regressor/steps/losses.py`
- 5D reconstruction and constraint-aware fusion (inference): `src/inference/pipeline.py`

2.2 Dataset and preprocessing (what `configs/train.yaml` uses)

2.2.1 From long-format CSV to image-level records

The official CSIRO `train.csv` is provided in a **long format**: each image appears multiple times (one row per target
name). We derive an `image_id` by stripping the `__<target_name>` suffix from `sample_id` and pivot the table into an
image-level dataframe (one row per image).

The same pivot also carries over metadata and auxiliary labels (`Sampling_Date`, `State`, `Height_Ave_cm`, `Pre_GSHH_NDVI`). The `(Sampling_Date, State)` pair is used for grouped cross-validation (§2.3), and `Height_Ave_cm` / `Pre_GSHH_NDVI` / `State` are used as auxiliary targets when enabled (§3.4).

Repo anchors

- CSV pivot + `image_id` derivation: `src/data/csiro_pivot.py`
- CSIRO dataset / dataloader wiring: `src/data/datamodule.py`

2.2.2 Image preprocessing

Images are loaded as RGB, resized to a fixed `data.image_size: [960, 480]` (width × height), converted to tensors, and
normalized with ImageNet statistics (`mean=[0.485,0.456,0.406]`, `std=[0.229,0.224,0.225]`).

During training we use the repo's `legacy` augmentation policy, which targets **robustness to framing, viewpoint, and
capture-condition variation** while keeping the overall pasture layout intact. Concretely, `configs/train.yaml` enables:

- **Crop-and-resize (framing jitter)**: a `RandomResizedCrop` with area scale 0.8–1.0 to emulate modest zoom / distance
  changes, then resizing back to the fixed resolution. (For highly-rectangular images, this is implemented robustly and
  falls back to a center crop when a random crop is not feasible under the chosen constraints.)
- **Geometric jitter**: horizontal and vertical flips (each $p = 0.5$), plus a light random affine transform (±5° rotation, up to 2% translation, 0.95–1.05 scaling, small shear) to cover camera pose and slight misalignment.
- **Photometric jitter**: `ColorJitter` applied with $p = 0.8$ (brightness/contrast ±0.2, saturation ±0.1) to reflect
  illumination and exposure changes.
- **Blur and sensor noise**: Gaussian blur ($p = 0.5$, kernel 3–7, sigma 0.1–1.0) and additive Gaussian noise ($p = 0.5$, std 0.01) to mimic motion blur, compression, and sensor artifacts.
- **Capture artifacts**: synthetic watermark/timestamp overlays ($p = 0.3$) and light-spot overlays ($p = 0.3$) so the
  model is less sensitive to incidental UI or glare-like effects.
- **Occlusion robustness**: random erasing ($p = 0.25$, removing 2–10% of pixels per sample) to discourage over-reliance
  on any single local patch.
- **Clean-sample mixing**: with probability 0.1, augmentation is bypassed and the clean evaluation preprocessing is used,
  keeping a fraction of training inputs close to the test-time distribution.

Validation uses deterministic resize + normalization only.

Repo anchors

- Train/eval transforms + overlays (watermark/light-spot/noise/blur/erasing): `src/data/augmentations.py`
- Image size parsing (`[width,height] → (height,width)`): `src/training/single_run.py`

2.2.3 Label preprocessing: units, z-score standardization, and masks

All biomass targets are reported in grams for a fixed plot size of $0.7\,\mathrm{m} \times 0.3\,\mathrm{m}$. We convert grams to **areal density** g/m^2 by dividing by plot area:

$$a = 0.7 \times 0.3 = 0.21\ \mathrm{m}^2$$
$$y_{g/m^2} = \frac{y_g}{a}.$$

We then apply per-target **z-score standardization** (mean/std computed on the training split):

$$y_z = \frac{y_{g/m^2} - \mu}{\sigma}.$$

In `configs/train.yaml`, `model.log_scale_targets: false`, so no log transform is applied to training labels. The
computed statistics are saved to `z_score.json` and reused at inference time to invert the normalization.

Besides the primary regression target (`Dry_Total_g`), we also z-score NDVI (`Pre_GSHH_NDVI`) with its own mean/std.
Height (`Height_Ave_cm`) is supervised in raw centimeters.

Not all labels are available for every image, so we use masks:

- **ratio supervision** is enabled only when the three components and total are all finite and `Dry_Total_g > 0`.
- **5D supervision** ignores missing components dimension-wise.

Repo anchors

- Gram $\rightarrow g/m^2$ conversion, z-score, and mask construction (CSIRO dataset): `src/data/datamodule.py`
- Dataset plot area resolution (meters $\rightarrow m^2$): `src/training/single_run.py`

2.3 Validation protocol

Our reported offline validation uses **5-fold cross-validation**.

- **Fold construction**: we build folds on the image-level dataframe and (by default) group by `(Sampling_Date, State)`, i.e. all images sharing the same date–state pair are assigned to the same fold. This reduces metadata leakage (samples collected under the same conditions are not split across train/val).
- **Per-fold training**: for fold j, we train on the other 4 folds and evaluate on the held-out fold using the metric
  using the competition metric (computed from the reconstructed 5D predictions in grams; §2.1).

Because R^2 depends on the baseline mean used in `SS_tot`, fold-to-fold scores can shift simply because each fold has a slightly different target distribution. For stable comparisons across folds, we also compute a **global-baseline** variant where the baseline mean is computed from the full dataset in grams and mapped to log space via `log1p(mean_grams)`, rather than from one fold only.

Repo anchors

- Grouped k-fold split generation: `src/training/splits.py`
- k-fold training loop + per-fold metrics extraction: `src/training/kfold_runner.py`
- Global-baseline R^2 computation: `src/models/regressor/steps/epoch_metrics.py`

3. Method

3.1 Backbone: DINOv3 features

This repository uses **DINOv3** as the visual backbone. Conceptually, we treat DINOv3 as a general-purpose image encoder that converts an input pasture image into a set of dense visual features; a lightweight regression head then maps those features to biomass targets (§3.3).

In our final configuration, we use **DINOv3 ViT-7B/16** (`dinov3_vit7b16`) pretrained on LVD-1689M, and we use it primarily as a **feature extractor** for downstream regression (with optional lightweight adaptation via LoRA; §3.2).

Repo anchors

- Backbone loading + patch-token extraction + intermediate layers: `src/models/backbone.py`

3.1.1 Patch tokens as a spatial representation

Vision Transformers split an image into non-overlapping patches and embed each patch into a feature vector. With a 16×16 patch size, an image of size $H \times W$ becomes a patch grid of size $H_p \times W_p$, where $H_p = H/16$ and $W_p = W/16$. The backbone outputs a sequence of **patch tokens** with shape (B, N, C), where B is batch size, $N = H_p \cdot W_p$ is the number of patches, and C is the embedding dimension.

For the DINOv3 ViT-7B/16 backbone, $C = 4096$. At our typical resolution (960×480, width × height), $H_p = 30$, $W_p = 60$, so $N = 1800$ patch tokens.

We focus on **patch tokens** rather than the **CLS token**, because our readout is spatial: it reshapes tokens back into a 2D feature map and builds multi-scale features to support robust scalar prediction (§3.3). Patch tokens preserve locality (e.g., separating vegetation from background) while still encoding global context through self-attention.

3.1.2 Multi-layer feature extraction and fusion

Instead of taking features only from the final transformer block, we extract patch tokens from a small set of **intermediate layers**. In our final setup we use layers **[23, 31, 39]**. Intuitively, different depths capture different information (lower-level texture/structure vs higher-level semantics), and combining them can improve transfer on small, domain-shifted datasets.

To combine multiple layers, we first align their feature spaces with lightweight per-layer projections (bottlenecks), then apply a **learned fusion** to produce a single fused token representation passed to the downstream head. (When using multi-layer heads, we may also run a separate head per selected layer and fuse their predictions; see §3.3.3.)

Repo anchors

- Multi-layer token extraction in training: `src/models/regressor/steps/predict.py`

3.2 Backbone adaptation: frozen backbone + LoRA (optional)

The CSIRO biomass dataset is small compared to the capacity of a multi-billion-parameter ViT backbone, so we avoid full end-to-end finetuning by default. Instead, we train in a **"frozen backbone + lightweight readout"** regime: DINOv3 provides stable patch-token features (§3.1), and nearly all task learning happens in the head (§3.3).

When the domain gap matters (pasture imagery differs from the natural-image distribution used for pretraining), we optionally add **LoRA** adapters as a controlled way to adapt the backbone **without** updating its original weights. In our setup, LoRA keeps all pretrained DINOv3 weights frozen and learns only a small set of adapter parameters injected into the attention **QKV** projections of the **last 16 transformer blocks**. We always train the readout head; when LoRA is enabled, adapter parameters are optimized in a separate parameter group.

Repo anchors

- LoRA injection + “train adapters only” policy: `src/models/peft_integration.py`
- LoRA optimizer parameter groups (`group_type: lora`) and LR settings: `src/models/regressor/optim.py`

3.3 Readout heads

In the final run (`configs/train.yaml`), we use a **ViTDet-style SimpleFeaturePyramid scalar head** (`model.head.type: vitdet`) to map DINOv3 patch tokens to scalar outputs. The key design choice is to keep the prediction path *explicitly spatial* until the very end: patch tokens are reshaped into a 2D grid, processed with a small multi-scale pyramid, and only then pooled into a global descriptor for scalar regression.

Repo anchors

- ViTDet SimpleFeaturePyramid scalar head (single + multi-layer): `src/models/regressor/heads/vitdet.py`
- Patch-grid inference from (H, W) + patch size: `src/models/regressor/heads/spatial_fpn.py`

3.3.1 Per-layer readout: patch tokens → pyramid → global descriptor

For one backbone layer, the head takes patch tokens T (shape $B \times N \times 4096$) (we do not use the CLS token; `use_cls_token: false`) and:

1. **Reshapes tokens into a 2D token map.** Using the ViT patch size (16×16), we reshape tokens into X (shape $B \times 4096 \times H_p \times W_p$), where (H_p, W_p) is the patch grid implied by the input image size.
2. **Builds a SimpleFeaturePyramid over the token map.** We form four pyramid levels at scales $\{2.0, 1.0, 0.5, 0.25\}$ (relative to the base grid):
   - **2.0×**: upsample by 2 using a transposed convolution (with channel reduction in the upsampling stage),
   - **1.0×**: identity (no resampling),
   - **0.5×**: downsample by 2 using max-pooling,
   - **0.25×**: downsample by 4 using repeated max-pooling (repo extension beyond Detectron2 ViTDet).

   Each level is then projected to a common width of **512 channels** via lightweight conv blocks ($1 \times 1 \to \mathrm{LN} \to 3 \times 3 \to \mathrm{LN}$), matching the ViTDet convention of channel-wise LayerNorm in NCHW.
3. **Pools and concatenates multi-scale features.** We global-average-pool each 512-channel pyramid level to a vector in $\mathbb{R}^{512}$ and concatenate them into a single descriptor g in $\mathbb{R}^{2048}$ (4×512).
4. **Applies an MLP to obtain the shared bottleneck.** We pass g through a small scalar MLP ($2048 \to 4096$ with **SiLU** and strong dropout) to produce z in $\mathbb{R}^{4096}$. This z is the shared representation used by all scalar prediction heads.

3.3.2 Predicting total + ratio (and auxiliary scalars)

From the shared bottleneck z, we predict:

- **Main scalar (`reg3`)**: in the final config, `data.target_order: [Dry_Total_g]`, so `reg3` predicts a single **total** term t. Softplus (`use_output_softplus`) is only applied when predicting in a non-log, non-z-scored space; with z-scored targets (our final config), it is disabled in code.
- **Ratio logits (`ratio`)**: a 3-way logit vector for the composition **(clover, dead, green)**. With `ratio_head_mode: shared`, ratio logits are produced from the same z as the total (shared trunk).

- **NDVI**: because `mtl.tasks.ndvi: true`, NDVI is predicted directly inside the ViTDet head from z.
- **Height / state**: because `mtl.tasks.height: true` and `mtl.tasks.state: true`, we attach lightweight linear heads on top of z for these auxiliary targets.

3.3.3 Multi-layer inference: per-layer heads → learned fusion → 5D biomass

The final config enables multi-layer extraction (`model.backbone_layers.enabled: true`) from DINOv3 layers **[23, 31, 39]**. We apply the same per-layer readout described above (§3.3.1–§3.3.2) to each layer's patch tokens, producing per-layer predictions (total t_l, ratio logits r_l, and bottleneck z_l).

We then fuse layers with **learned fusion weights** (softmax-normalized weights over layers; `layer_fusion: learned`). Rather than averaging in logit space, we fuse in **component space** to preserve non-negativity and the biomass constraints in §2.1: convert each layer's (t_l, r_l) into non-negative component masses, take a weighted average across layers, and finally reconstruct the 5D outputs using the identities in §2.1. This keeps the final prediction consistent by construction.

Repo anchors

- Learnable layer fusion weights (`layer_fusion: learned`): `src/models/regressor/heads/vitdet.py`
- Component-space fusion (training-time; used for ratio supervision when per-layer outputs exist): `src/models/regressor/steps/losses.py`
- Component-space fusion (inference-time; builds final 5D grams): `src/inference/pipeline.py`

3.4 Training objective and multi-task optimization

Our final model is trained as a **multi-task** system: besides the primary biomass prediction, we supervise a ratio decomposition head (to enforce additive constraints by construction) and attach small auxiliary heads (height/NDVI/state) that act as additional learning signals. This section explains (i) the loss terms we optimize, and (ii) how we combine and optimize them in a stable way.

This section covers: (i) feature-space regularization via manifold mixup (§3.4.1), (ii) the loss terms (§3.4.2), (iii) how we balance them (UW; §3.4.3), and (iv) how we mitigate gradient interference (PCGrad; §3.4.4).

3.4.1 Feature-space augmentation: manifold mixup

The CSIRO training set is small relative to the capacity of modern ViT backbones and can exhibit substantial nuisance variation (illumination, viewpoint, camera artifacts). Beyond pixel-level augmentation (§2.2.2), we therefore regularize the model with **manifold mixup**, i.e. mixup performed on intermediate representations rather than on raw pixels.

In our final configuration, we apply manifold mixup to DINOv3 **patch tokens** (the input to the spatial readout head). For a mini-batch of token tensors T_i and supervision targets y_i, we sample a coefficient $\lambda \sim \mathrm{Beta}(\alpha, \alpha)$ and a random pairing permutation π, then form mixed tokens:

$$T_i' = \lambda T_i + (1-\lambda) T_{\pi(i)}$$

Targets are mixed with the same coefficient:

$$y_i' = \lambda y_i + (1-\lambda) y_{\pi(i)}$$

and missing-label masks are combined conservatively (logical AND) so that a mixed target is supervised only when both originals are valid.

Because our biomass formulation couples a total scalar with a ratio decomposition (§2.1), we apply mixing in a **constraint-aware** way: when 5D component masses are available, we mix them and recompute the ratio target from the mixed components, ensuring the ratio remains on the simplex and avoiding inconsistent ratio supervision.

Concretely, when 5D component masses are available $y_i^{(5)} = [c_i, d_i, g_i, \mathrm{GDM}_i, t_i]$, we **mix masses first** and derive ratio targets from the mixed masses (with a small epsilon for numerical safety):

$$\mathbf{y}_i^{(5)\prime} = \lambda \mathbf{y}_i^{(5)} + (1-\lambda)\mathbf{y}_{\pi(i)}^{(5)}$$

$$\mathbf{p}_i' = \left[\frac{c_i'}{\max(t_i', \epsilon)}, \frac{d_i'}{\max(t_i', \epsilon)}, \frac{g_i'}{\max(t_i', \epsilon)} \right].$$

This keeps ratio supervision consistent with the same mixed total t_i'. Masks are ANDed, so ratio loss is applied only when both paired samples have valid (c, d, g, t) and $t_i' > 0$.

In `configs/train.yaml`, manifold mixup is enabled with high probability (≈0.96) and a Beta parameter $\alpha \approx 3.9$ (strong mixing). It is applied during training only and disabled for validation/inference.

Repo anchors

- Manifold mixup implementation: `src/models/regressor/manifold_mixup.py`
- Where mixup is applied to backbone tokens/features: `src/models/regressor/steps/predict.py`

3.4.2 Loss terms

In the final configuration (`configs/train.yaml`), training optimizes a collection of supervised objectives. In code, these loss components are keyed by task name (used consistently by Uncertainty Weighting and PCGrad): **`reg3`**, **`ratio`**, **`biomass_5d`**, **`height`**, **`ndvi`**, and **`state`**.

Repo anchors

- Supervised losses + masked objectives (reg3/ratio/5D/aux): `src/models/regressor/steps/losses.py`
- Training/validation step orchestration (CutMix + mixup + loss wiring): `src/models/regressor/steps/shared_step.py`

Below we define each raw task loss L_t before describing how we combine them (§3.4.3) and how we handle gradient interference (§3.4.4).

Let i index samples in a mini-batch $\mathcal{B}$. The model predicts:

- a primary scalar $\hat{t}_i$ (our run uses `data.target_order: [Dry_Total_g]`, so t is "total"),
- ratio logits $\hat{\mathbf{r}}_i \in \mathbb{R}^3$ for (clover, dead, green), with probabilities $\hat{\mathbf{p}}_i = \mathrm{softmax}(\hat{\mathbf{r}}_i)$,
- auxiliary predictions $\hat{h}_i$ (height), $\hat{n}_i$ (NDVI), and class logits $\hat{s}_i$ (state).

We also build a derived 5D biomass vector (in the repo's canonical order `[ Dry_Clover, Dry_Dead, Dry_Green, GDM, Dry_Total ]`) from $(\hat{t}_i, \hat{\mathbf{p}}_i)$ using the constraints in §2.1:

$$\hat{y}_{i,\mathrm{clover}}^{(5)} = \hat{p}_{i,0}\hat{t}_i,$$
$$\hat{y}_{i,\mathrm{dead}}^{(5)} = \hat{p}_{i,1}\hat{t}_i,$$
$$\hat{y}_{i,\mathrm{green}}^{(5)} = \hat{p}_{i,2}\hat{t}_i.$$

$$\hat{y}^{(5)}_{i,\text{gdm}} = \hat{y}^{(5)}_{i,\text{clover}} + \hat{y}^{(5)}_{i,\text{green}},$$
$$\hat{y}^{(5)}_{i,\text{total}} = \hat{y}^{(5)}_{i,\text{clover}} + \hat{y}^{(5)}_{i,\text{dead}} + \hat{y}^{(5)}_{i,\text{green}}.$$

Because not every label is present for every sample (and because some values can be invalid), the dataloader provides binary masks; we use these masks to compute **masked** losses.

(a) Main regression (`reg3`). We use a masked mean squared error:

$$L_{\text{reg3}} = \frac{1}{\sum_{i\in\mathcal{B}} m_i^{(\text{reg3})}} \sum_{i\in\mathcal{B}} m_i^{(\text{reg3})} \left(\hat{t}_i - t_i\right)^2.$$

(b) Ratio supervision (`ratio`). The ratio head is supervised in probability space (softmaxed logits) with an MSE on the simplex:

$$L_{\text{ratio}} = \frac{1}{\sum_{i\in\mathcal{B}} m_i^{(\text{ratio})}} \sum_{i\in\mathcal{B}} m_i^{(\text{ratio})} \left\|\hat{\mathbf{p}}_i - \mathbf{p}_i\right\|_2^2.$$

(c) 5D weighted loss (`biomass_5d`). Even though the model's *direct* supervised target is 1D (`Dry_Total_g`), we additionally supervise the reconstructed 5D outputs to stabilize the decomposition. We compute a per-dimension masked MSE and then take a weighted mean:

$$L_{\text{5d}} = \frac{1}{\sum_k w_k} \sum_{k=1}^{5} w_k \cdot \frac{1}{|\mathcal{B}_k|} \sum_{i\in\mathcal{B}} m_{i,k}^{(5)} \left(\hat{y}_{i,k}^{(5)} - y_{i,k}^{(5)}\right)^2,$$
$$\mathcal{B}_k = \{i : m_{i,k}^{(5)} = 1\}.$$

In `configs/train.yaml`, we set the weight vector w to match the leaderboard emphasis:

$$w = [0.1, 0.1, 0.1, 0.2, 0.5],$$

corresponding to `[ Dry_Clover, Dry_Dead, Dry_Green, GDM, Dry_Total ]`.

(d) Auxiliary tasks (`height`, `ndvi`, `state`). We add small auxiliary heads on the shared bottleneck z and supervise them with standard objectives:

- height: $L_{\text{height}} = \text{MSE}(\hat{h}_i, h_i)$,
- NDVI: a masked MSE L_{ndvi} analogous to L_{reg3},
- state: a multi-class cross-entropy $L_{\text{state}} = \text{CE}(\hat{s}_i, s_i)$.

3.4.3 Uncertainty Weighting (UW) for loss balancing

With multiple loss terms, a naïve sum requires hand-tuned coefficients (and those coefficients can change when we add/remove tasks, change normalization, or change batch composition). In `configs/train.yaml` we enable **Uncertainty Weighting (UW)** via `loss.weighting: uw` and learn the balancing automatically.

We maintain one learnable scalar **log-variance** per task, $s_t = \log(\sigma_t^2)$, for all enabled tasks $\mathcal{T} = \{\text{reg3}, \text{ratio}, \text{biomass\_5d}, \text{height}, \text{ndvi}, \text{state}\}$.
Given the raw task losses L_t from §3.4.2, the total scalar loss is:

$$L = \frac{1}{|\mathcal{T}|} \sum_{t\in\mathcal{T}} \frac{1}{2}(\exp(-s_t)\, L_t + s_t).$$

Intuitively, $\exp(-s_t)$ acts as a **learned weight** (proportional to $1/\sigma_t^2$): tasks that look "noisier" (higher uncertainty) are down-weighted, while tasks with more confident supervision are up-weighted. The additive s_t term prevents the trivial solution of driving all weights to zero.

In implementation, these UW parameters are stored as a `ParameterDict` keyed by task name and are optimized as their own optimizer group (controlled by `optimizer.uw_lr` and `optimizer.uw_weight_decay`). See the source-of-truth hooks in `src/models/regressor/steps/uw.py` and UW parameter creation in `src/models/regressor/biomass_regressor.py`.

3.4.4 PCGrad (primary-anchored) for gradient conflict mitigation

Even with UW balancing, multi-task learning can suffer from **gradient interference**: one task's update direction can slow down or harm another task. We mitigate this with **PCGrad** (Projected Conflicting Gradients), enabled by `pcgrad.enabled: true` in `configs/train.yaml`.

Primary vs auxiliary tasks (our choice)

Not all tasks are equally important to the final leaderboard score. In `configs/train.yaml` we treat:

- **primary tasks**: `{reg3, ratio, biomass_5d}`
- **auxiliary tasks**: `{height, ndvi, state}`

and run PCGrad in **primary-anchored** mode (`pcgrad.mode: primary_anchored`).

Primary-anchored PCGrad

Let g_p be the combined "primary" gradient (an anchor direction), and g_a an auxiliary task gradient. For each auxiliary task, if it conflicts with the primary direction (negative dot product), we project out the component along g_p:

$$g_a \leftarrow g_a - \frac{\langle g_a, g_p \rangle}{\|g_p\|^2 + \varepsilon} g_p \quad \text{if} \quad \langle g_a, g_p \rangle < 0.$$

The final update is then $g = g_p + \sum g_a$ (with `pcgrad.reduction: sum`), so auxiliary tasks contribute only when they do not oppose the primary direction.

What PCGrad is applied to (important implementation choice)

In our repo, PCGrad is **not** applied to every parameter:

- It is applied only to optimizer parameter groups tagged `group_type: head` (`pcgrad.apply_to_group_types: [head]`).
- We explicitly exclude the UW parameter group (`pcgrad.exclude_group_types: [uw]`) so the learned log-variances are not affected by cross-task gradient surgery.
- LoRA parameters live in `group_type: lora` groups and are therefore optimized with standard gradients in this config (they are not selected by `apply_to_group_types`).

UW-aware per-task terms

PCGrad operates on per-task **terms that sum to the exact scalar loss used for backprop**, not on the raw L_t. Concretely, when UW is enabled, each task term becomes $0.5 \cdot \left(\exp(-s_t)\, L_t + s_t\right)$ (and we keep the same normalization by number of tasks). This makes the PCGrad procedure consistent with the true UW objective, rather than a separate reweighted surrogate.

The source-of-truth implementation lives in `src/models/regressor/biomass_regressor.py` (custom `backward` hook) and `src/training/pcgrad.py` (projection routines).

4. Implementation Details

This section is intentionally brief. It records only the **reproducibility knobs** and the **weight artifacts/contract** needed to run and deploy the solution. The modeling and optimization choices are described in §2–§3.

4.1 Configuration system and reproducibility

Run the legacy single-file YAML configuration (`configs/train.yaml`):

```
python train.py --config configs/train.yaml
```

At startup we:

- seed all RNGs via `pl.seed_everything(seed, workers=True)`,
- snapshot the loaded config to `<log_dir>/train.yaml` (default: `outputs/<version>/train.yaml`),
- write logs/checkpoints under `logging.log_dir` and `logging.ckpt_dir` (default: `outputs/` and `outputs/checkpoints/`, namespaced by `version`).

Repo anchors

- Legacy launcher: `train.py`
- Training entrypoint: `src/training/entrypoint.py`
- Legacy YAML training routine: `src/training/single_run.py`

4.2 Training stack (high-level)

Training uses PyTorch Lightning with a single `LightningModule` (`BiomassRegressor`). The surrounding orchestration is
thin: build the datamodule and module from config, configure a `Trainer` (precision/accumulation/clipping), and attach a
small callback set (checkpointing, head export, EMA, NaN/Inf guard).

Repo anchors

- Core LightningModule: `src/models/regressor/biomass_regressor.py`
- Training runner: `src/training/single_run.py`
- Callbacks: `src/callbacks/`
- EMA callback: `src/callbacks/ema.py`
- NaN/Inf guard callback: `src/callbacks/nonfinite_guard.py`

4.3 Weights management: backbone vs head-only artifacts

Because the DINOv3 backbone is large and frozen by default, we treat it as a **shared dependency** and version only the
lightweight, run-specific pieces (the regression head and minimal metadata):

- **Backbone weights (shared)**: stored once under `dinov3_weights/` (e.g. `dinov3_weights/dinov3_vitl16_pretrain_lvd1689m-8aa4cbdd.pt`). Training refers to this via `model.weights_path` and the
  entrypoint best-effort copies provided weights into `dinov3_weights/`.
- **Lightning `last.ckpt` (resume)**: kept for backward compatibility/resume, but made lightweight by stripping frozen
  backbone tensors from the checkpoint state_dict while retaining LoRA adapter tensors.
- **Head-only checkpoint (portable)**: `outputs/checkpoints/<version>/head/head-epochXXX*.pt` exported by
  `HeadCheckpoint` (by default only the final epoch). Contains `state_dict` plus a minimal `meta` block (and optional
  `peft`) to reconstruct the head at inference time.
- **Packaged inference folder**: `package_artifacts.py` copies a selected head checkpoint to `weights/head/infer_head.pt`
  and snapshots configs (including per-version snapshots for ensembles).

Inference consumes **two inputs**: a backbone weights file (or directory) and a head weights file/directory (usually
`weights/head/`).

Repo anchors

- Head checkpointing callback: `src/callbacks/head_checkpoint.py`
- Lightning checkpoint slimming: `src/models/regressor/checkpointing.py`
- Packaging script: `package_artifacts.py`
- Inference loader for head weights: `src/inference/torch_load.py`

5. Inference and Ensembling

This section documents the **submission-time inference path used in this repository**, with emphasis on the final
multi-model ensemble defined in `configs/ensemble.json`.

5.1 Entrypoint (`infer_and_submit_pt.py`)

Inference runs fully offline via `infer_and_submit_pt.py`. Set the required variables at
the top of the script and execute:

```
python infer_and_submit_pt.py
```

At minimum you must provide:

- `PROJECT_DIR`: repository root (contains `configs/` and `src/`), or a packaged weights directory that also contains
  `configs/` and `head/`.
- `INPUT_PATH`: dataset directory (containing `test.csv`) or a direct `test.csv` path.
- `OUTPUT_SUBMISSION_PATH`: where to write the long-format `submission.csv`.
- `DINO_WEIGHTS_PT_PATH`: a backbone weights file (`.pt`/`.pth`) or a directory containing official DINOv3 weights.
- `HEAD_WEIGHTS_PT_PATH`: head weights directory (default: `weights/head/`).

The inference pipeline reads the **model/data definition from YAML** (image size, normalization, dataset plot geometry,
target parameterization). In the default setup this is `configs/train.yaml`, and in ensemble mode each member may use a
per-version snapshot (see §5.2.2).

5.2 The submission ensemble we used (`configs/ensemble.json`)

Our final submission uses a **4-model ensemble** configured in `configs/ensemble.json`:

- `tune-vitdet-v1-12h-56-c-05-ema-0.99`
- `tune-vitdet-v1-12h-56-10-vit7b`
- `tune-vitdet-v1-12h-56-c-05-ema-0.99-vit7b-v2`
- `tune-vitdet-v1-12h-56-c-05-ema-0.99-vit7b-v3`

To activate the ensemble, set `"enabled": true`. (The repo keeps it `false` by default to preserve single-model behavior.)

5.2.1 Expected packaged layout (per version)

When `enabled=true`, the pipeline resolves each member by its `version` string and expects:

- **Head weights base dir**:
  - running from repo root: `weights/head/<version>/`
  - running from a packaged weights dir: `head/<version>/`

Each `<version>/` directory should contain the exported head checkpoint(s), typically a single `infer_head.pt` produced by
`package_artifacts.py` (see §4.3).

5.2.2 Per-version config snapshots

For correct preprocessing/geometry at inference time, each member can load a per-version `train.yaml` snapshot (best
effort resolution order):

1. `weights/configs/versions/<version>/train.yaml` (repo root)
2. `configs/versions/<version>/train.yaml` (packaged weights dir)
3. fallback to `configs/train.yaml`

Repo anchors

- Per-version config + head dir resolution: `src/inference/paths.py`
- Ensemble config parsing + per-model caching (`outputs/ensemble_cache/`): `src/inference/ensemble.py`
- TTA view generation with ViT patch-grid alignment: `src/inference/pipeline.py`
- MC-dropout implementation (dropout-only train mode): `src/inference/predict.py`

5.2.3 Aggregation (how predictions are combined)

Each model produces a per-image 5D vector in grams in the canonical repo order `[Dry_Clover_g, Dry_Dead_g, Dry_Green_g, GDM_g, Dry_Total_g]`. The ensemble first computes a **base weighted mean**
(equal weights by default), and then applies the configured aggregation:

- In our config: `aggregation: "per_sample_weighted_mean"` with **MC-dropout uncertainty weighting** (head-only) as
  specified under `per_sample_weighting.mc_dropout` (64 samples). Disagreement-based reweighting is disabled in the
  current config.

Per-model outputs are cached under `outputs/ensemble_cache/` to avoid recomputing model forward passes when rerunning.

6. Limitations and Future Work

Due to limited compute and time, we performed almost no rigorous ablation studies; most decisions were driven by score changes on the public leaderboard. This makes it hard to explain which design choices were truly beneficial. The most obvious example is multi-task learning (MTL): we spent substantial effort on MTL and additional datasets, but we cannot conclusively tell whether they improved generalization. We also considered introducing synthetic data (as used by the rank-2 solution), but did not find a practical way to implement it.

If we had the opportunity to continue this project, we would like to explore the approach on a more complete dataset (potentially including mask annotations). We believe an R^2 of 0.69 is far from the upper bound of this task. We would also like to try lighter models (e.g., ViT-H or even ViT-S) to achieve comparable performance—imagine an edge-deployable model that can predict biomass locally inside a mobile app. We sincerely hope that our models, as well as those from other teams, can eventually be applied in the real world.

7. Conclusion

In summary, this repository demonstrates a pragmatic “representation + readout” recipe for scalar regression on small, noisy vision datasets: reuse a strong DINOv3 ViT backbone to produce dense patch-token features (kept frozen by default, optionally adapted with LoRA) and train a lightweight but explicitly spatial readout head (a ViTDet-style SimpleFeaturePyramid) that aggregates multi-scale evidence into a global descriptor for biomass prediction. Key engineering choices include a constraint-preserving parameterization (predict total mass + composition ratio to satisfy biomass identities by construction), feature-space regularization via manifold mixup, stable multi-task optimization with auxiliary supervision, Uncertainty Weighting for loss balancing, and primary-anchored PCGrad to reduce cross-task gradient interference. With grouped 5-fold validation and an inference-time ensemble (including head-only MC-dropout uncertainty weighting), our final

submission reached 0.78 (rank 8) on the public leaderboard and 0.65 (rank 6) on the private leaderboard; our best submission achieved 0.76/0.69 on the public/private leaderboard.

8. Postscript

I want to share a short story about competition strategy. Early on, we had a very promising architecture: it was the first one that pushed our public leaderboard score to 0.76, and the same idea achieved 0.68–0.69 on the private leaderboard—enough to beat the eventual rank-1 solution. The core idea was simple: during preprocessing, we converted biomass from grams to *density*, which allows arbitrary image cropping while still assigning a global density label to any region. DINOv3 outputs a sequence of patch tokens; each patch token can also be viewed as a kind of "crop", while carrying both local and global semantic information. So we can transform the supervision from (image → biomass in grams) to (patch token → biomass density), train an MLP to predict the density of each token, and then average over all tokens to recover the full-image prediction. (In fact, this idea likely originated from a discussion thread or a public notebook, but I no longer remember the link.) Unfortunately, at that time I was not running strict k-fold validation for each submission; I was operating with a gambler-like loop of "manual tuning → submit → watch public leaderboard changes". This lack of rigor led me to overlook an architecture that clearly had potential, and I ultimately missed the chance to reach rank 1.

On the final day, I made another short-sighted mistake: I was intimidated by the rapidly rising public leaderboard scores. Imagine being rank 1 on the public leaderboard for almost two months, only to slide quickly to around rank 20 in the last month—others keep improving, while you never surpass your own submission from two months ago. In that situation, how tempting is it when a last-minute training run suddenly jumps you back to rank 6? That "submit the highest-score model" gambler mindset ultimately took over. Instead of submitting a k-fold-validated ensemble (local k-fold score ~0.67), I submitted a single ViT-7B model plus an ensemble of a few of its variants. Another reason behind that decision was my belief that a larger backbone would always perform better—which did hold as we upgraded from ViT-S/16+ to ViT-H/16+. This second mistake cost me a top-3 finish.

That is the end of this slightly regretful story. In fact, I only started learning deep learning for computer vision about a year ago due to work needs, and this was my first Kaggle competition that I fully committed to. As a beginner, winning a gold medal was genuinely surprising and exciting.

This competition also taught me a lot. I have since applied the same strategy—DINOv3 + LoRA for parameter-efficient fine-tuning, plus a lightweight regression head—to my work, and achieved striking agreement with human experts even when training only on pseudo-labeled data. In industry, many problems are posed without clear requirements, sometimes even without a clear definition—you often have to start by building the dataset and the evaluation metrics. Kaggle and the organizers provided an excellent platform and a well-defined task for testing ideas. The discussions and public notebooks were also a great source of interesting techniques and insights. Without this competition, I doubt I could have applied similar approaches so quickly in practice. Finally, I would like to express my sincere thanks to Kaggle, the organizers, and all participants.

9. Third-party code (`third_party/`)

All third-party code referenced/used in this project is vendored under `third_party/` (primarily as git submodules) for reproducibility. The full list is:

- `third_party/DI-Retinex` (sunshangquan/DI-Retinex)
- `third_party/DPT` (isl-org/DPT)
- `third_party/F-SAM` (nblt/F-SAM)

- `third_party/GSAM` (juntang-zhuang/GSAM)
- `third_party/LibMTL` (median-research-group/LibMTL)
- `third_party/TabPFN` (PriorLabs/TabPFN)
- `third_party/albumentations` (albumentations-team/albumentations)
- `third_party/augmix` (google-research/augmix)
- `third_party/c-mixup` (huaxiuyao/C-Mixup)
- `third_party/causal-conv1d` (Dao-AILab/causal-conv1d)
- `third_party/d2` (terrastruct/d2)
- `third_party/detectron2` (facebookresearch/detectron2)
- `third_party/dinov3` (facebookresearch/dinov3)
- `third_party/dinov3-finetune` (RobvanGastel/dinov3-finetune)
- `third_party/dinov3-src` (source snapshot of DINOv3; upstream: facebookresearch/dinov3)
- `third_party/eomt` (tue-mps/eomt)
- `third_party/fixmatch` (google-research/fixmatch)
- `third_party/lightly-train` (lightly-ai/lightly-train)
- `third_party/mamba` (state-spaces/mamba)
- `third_party/multi_task_learning_pytorch` (SimonVandenhende/Multi-Task-Learning-PyTorch)
- `third_party/peft` (huggingface/peft)
- `third_party/peft-src` (source snapshot of PEFT; upstream: huggingface/peft)
- `third_party/pytorch-lightning` (Lightning-AI/pytorch-lightning)
- `third_party/ray` (ray-project/ray)
- `third_party/sam` (davda54/sam)
- `third_party/uda` (google-research/uda)

10. Papers

- **DINOv3** — Oriane Siméoni et al., 2025. arXiv:2508.10104
- **Gradient Surgery for Multi-Task Learning (PCGrad)** — Tianhe Yu et al., 2020. arXiv:2001.06782
- **Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics** — Alex Kendall, Yarin Gal, Roberto Cipolla, 2018. arXiv:1705.07115
- **LoRA: Low-Rank Adaptation of Large Language Models** — Edward J. Hu et al., 2021. arXiv:2106.09685
- **Exploring Plain Vision Transformer Backbones for Object Detection (ViTDet)** — Yanghao Li et al., 2022. arXiv:2203.16527