Time: solving the variable-dimension batch problem in log segmentation

Almost every published segmentation result quietly assumes that training images can be stacked into a rectangular batch tensor. Scanned well logs break that assumption at the source: a raster archive contains images spanning from roughly 3,200 to 12,800 pixels wide and 480 to 640 pixels tall, in no fixed ratio, because the physical logs were printed at different scales, photographed at different times, and scanned by different operators across decades. You cannot stack tensors of different shapes, so the naive training loop fails before the first forward pass. This whitepaper is a pure engineering account of how we trained VeerNet, our encoder-decoder segmentation network with a transformer attention bottleneck, under exactly this constraint. The binary-segmentation stage was forced to a batch size of 1, which trains correctly but starves the optimiser of the gradient averaging that makes large-batch training stable, and it took 110 minutes for 50 epochs over 2,000 instances. For the harder three-class multiclass stage on 15,000 synthetic instances we engineered a custom PyTorch collate function that pads a mini-batch to the largest member, carries a validity mask so padding never contributes to the loss, and pairs it with gradient accumulation to reach an effective batch of 16. That run completed 50 epochs in 550 minutes. We cover the memory arithmetic that sets the real ceiling, why padding-and-masking beats resizing for one-pixel-wide curve traces, the GroupNorm choice that keeps normalisation honest at batch size 1, and the throughput numbers an engineer needs to plan a training budget. The geoscience is upstream of this document; what follows is the systems problem of getting a variable-dimension dataset through a fixed-shape training loop without corrupting the gradient.

Narendra Patwardhan
Research Collaborator

Quamer Nasim
ML Research Engineer

Tarry Singh
Founder & CEO

VEERNET / VARIABLE-DIMENSION / BATCH-SIZE / COLLATE-FN / GRADIENT-ACCUMULATION / GROUPNORM

CONTENTS

A deep-learning training loop has one structural assumption baked so deep that most engineers never have to think about it: a batch is a single rectangular tensor. You stack `N` images of shape `(C, H, W)` into a tensor of shape `(N, C, H, W)`, hand it to the GPU, and the matrix multiplies that follow are only well-defined because every image in the batch has the same `H` and the same `W`. Resize everything to 224 by 224, stack, train. The assumption is so reliable on natural-image benchmarks that it disappears into the framework defaults.

Scanned well logs break it at the source, and they break it badly enough that the naive loop does not produce a worse model, it produces no model at all. The raster archive we trained VeerNet on contains images that run from roughly **3,200 to 12,800 pixels wide** and **480 to 640 pixels tall**, in no fixed aspect ratio. The physical logs were printed at different vertical scales, by different service companies, across roughly five decades, then photographed and scanned by different operators on different equipment. There is no natural common shape to resize to, and as we will argue, resizing is exactly the wrong move for this target anyway. The first time you call `DataLoader` with the framework default collate function, it tries to stack tensors of different shapes and throws before the first forward pass.

This whitepaper is the engineering account of getting that variable-dimension dataset through a fixed-shape training loop without corrupting the gradient. It is a companion to our VeerNet architecture and loss-ablation work, and it deliberately stays in its lane: the architecture is upstream of this document, the geoscience is upstream of the architecture, and what follows is purely the systems problem of throughput, memory, and the variable-dimension constraint. We built VeerNet as an encoder-decoder segmentation network, residual blocks in the encoder [1], an upsampling decoder in the U-Net lineage [2], and a two-layer transformer attention refinement on the bottleneck [3]. None of that is the subject here. The subject is the batch.

Why you cannot just resize the logs

The reflex fix is to resize every log to a common shape and move on. For natural images that reflex is correct and costs nothing the model cares about. For well-log segmentation it quietly destroys the target.

The thing VeerNet is trained to find is a curve trace: a printed line that is, on the source raster, often **one pixel wide**. The entire supervisory signal lives in those thin traces, and the segmentation mask is sparse, almost all background with a few hairline foreground pixels. Resize a 12,800-pixel-wide log down to a few hundred pixels and the interpolation kernel does precisely what it is designed to do, which is smear a one-pixel feature across its neighbours and blend it into the background. The curve does not survive the downsample as a clean trace, and neither does its ground-truth mask. You would be training the network to reproduce an artefact of the resize filter rather than the curve.

Resizing also distorts geometry. A well log encodes value as horizontal deflection against a depth axis, so anisotropic resizing, which is what you get when you force wildly different aspect ratios to a common shape, changes the apparent gradient of every curve. The shape of the trace is the signal. You cannot stretch it on one axis and keep the meaning.

So resizing is off the table on quality grounds, not convenience grounds. That decision is what forces every downstream problem in this paper. If you could resize, none of the rest of this would be necessary. Because you cannot, the variable-dimension constraint is load-bearing, and it has to be solved in the data path, not designed away.

The binary stage: forced to batch size 1

The first training stage was binary segmentation, foreground curve against background, on an initial dataset of **2,000 instances**. Here we took the honest, blunt solution: train at a batch size of **1**.

Batch size 1 is the one setting that makes the variable-dimension problem evaporate, because a batch of one image never needs two shapes to agree. There is no stacking, so there is nothing to break. You feed one `(1, 1, H, W)` tensor through the network, compute the loss against one mask, backpropagate, step. The grayscale log is a single input channel, so the tensor is genuinely `(1, 1, H, W)`, and the model simply accepts whatever `H` and `W` arrive. The data path becomes trivial. Everything you give up, you give up at the optimiser.

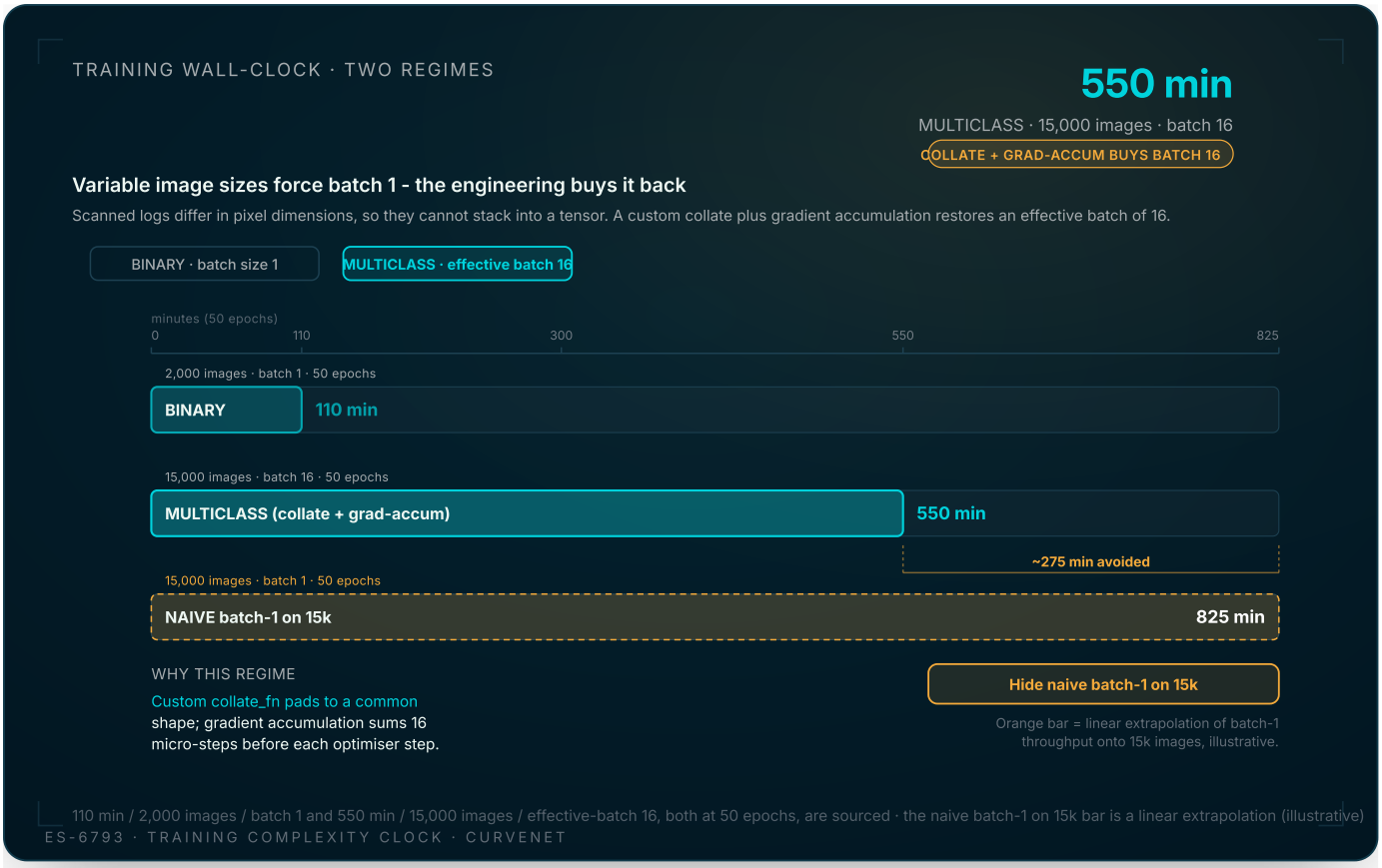
What you give up is gradient averaging. A batch of `N` images produces a gradient that is the mean of `N` per-example gradients, and that averaging is most of why mini-batch training is stable: it smooths the stochastic noise of any single example and lets you take

confident steps. At batch size 1 every step is driven by one image, the gradient is as noisy as the noisiest log in the set, and the optimiser jitters. You can partly compensate with a lower learning rate and more epochs, which is what we did, but you are paying for the variable-dimension constraint in wall-clock time and in training stability.

The other casualty at batch size 1 is normalisation, and it is worth being precise about why. Batch Normalization estimates the mean and variance of each feature channel across the batch dimension and normalises by those statistics [5]. With a single image in the batch, the "batch statistic" is computed over one example. The variance estimate is degenerate, the running statistics are driven by whatever single log happened to come through, and the normalisation that is supposed to stabilise training instead injects noise. BatchNorm at batch size 1 is not a small inefficiency, it is a correctness problem.

Our answer was Group Normalization [4]. GroupNorm computes its statistics across groups of channels within a single image and never looks at the batch dimension at all, so its behaviour is identical whether the batch is 1 or 64. That batch independence is the entire reason it belongs in this pipeline. We size the groups with a simple `half_or_16` rule, taking the larger of half the channel count and a floor of **16** channels per group, which keeps each group statistically meaningful without assuming anything about batch size. With GroupNorm in place, batch size 1 trains correctly. It trains slowly, but the gradient it computes is honest.

The numbers from that stage: **110 minutes for 50 epochs** over the 2,000 binary instances. That is a clean, reproducible training run, and it is the baseline every later optimisation is measured against. It is also a ceiling. Batch size 1 was never going to carry the harder problem.



Scanned well-log images arrive at wildly different pixel dimensions, so they cannot be stacked into a tensor; the naive fix is batch size 1, which under-feeds the GPU and stretches the wall-clock. CurveNet's answer is a custom collate_fn plus gradient accumulation, which buys back an effective batch of 16 without the memory blowup. Pick a regime; toggle the naive batch-1 on 15k bar to see the wall-clock the fix avoided. The 110 min / 2,000 images / batch 1 and 550 min / 15,000 images / effective-batch 16 figures (both at 50 epochs) are the handover's own; the orange naive bar is a linear extrapolation of the batch-1 throughput onto 15k images and is flagged as illustrative.

The multiclass stage: 15,000 instances, three classes, and a real batch

The multiclass stage is where the constraint stopped being tolerable. The task expanded to **three classes**, background plus two curves, on a synthetic dataset of **15,000 instances**, and the synthetic generator deliberately spanned the full dimensional range of the real archive, from the 3,200-pixel minimum width to the 12,800-pixel maximum, 480 to 640 pixels tall. The dataset was harder, larger, and dimensionally wider than the binary set, and the sparse-foreground imbalance was worse: with two thin curves against a vast background, the loss is dominated by background pixels unless you fight for the foreground. That imbalance is its own well-studied problem [7], and it is why our loss ablation landed on a Tversky objective [6] for this stage, but the imbalance only gets a

chance to matter if the batch is large enough to average over. At batch size 1 the noisy single-example gradient and the foreground imbalance compound each other. We needed a real batch.

The obstacle was unchanged: you still cannot stack a 3,200-wide log and a 12,800-wide log into one tensor. The framework default collate function will not do it, and resizing is still forbidden for the same one-pixel-trace reason. So we wrote our own.

The custom collate function: pad to the max, mask the pad

A PyTorch `DataLoader` builds each mini-batch by passing the list of sampled examples to a `collate_fn`, whose job is to turn a Python list of samples into the batched tensors the training step consumes. The default implementation assumes uniform shapes and stacks. We replaced it with a collate function that does three things in order.

First, it inspects the sampled mini-batch and finds the maximum height and maximum width across its members. The target shape for this batch is whatever the largest member needs, computed per batch rather than fixed for the whole dataset, so a mini-batch of small logs pads to a small box and a mini-batch that happens to catch a 12,800-wide log pads to a large one. The batch sizes itself to its contents.

Second, it pads every image and every mask in the mini-batch up to that per-batch maximum, bottom-and-right, with a constant background value. Now every member has the same shape and `torch.stack` is legal. The variable-dimension problem is gone at the tensor level.

Third, and this is the step that makes padding safe, it builds a per-pixel validity mask that marks which pixels are real and which are padding, and stacks that alongside the images. The padded regions are fiction, they carry no curve and no supervision, and if they reached the loss they would teach the network that vast rectangles of background-valued pixels are the correct answer, biasing it toward predicting nothing. The validity mask is multiplied into the loss so that **every padded pixel contributes exactly zero gradient**. The network sees a clean rectangular batch; the optimiser only ever learns from real pixels. Pad to enable the stack, mask to protect the gradient.

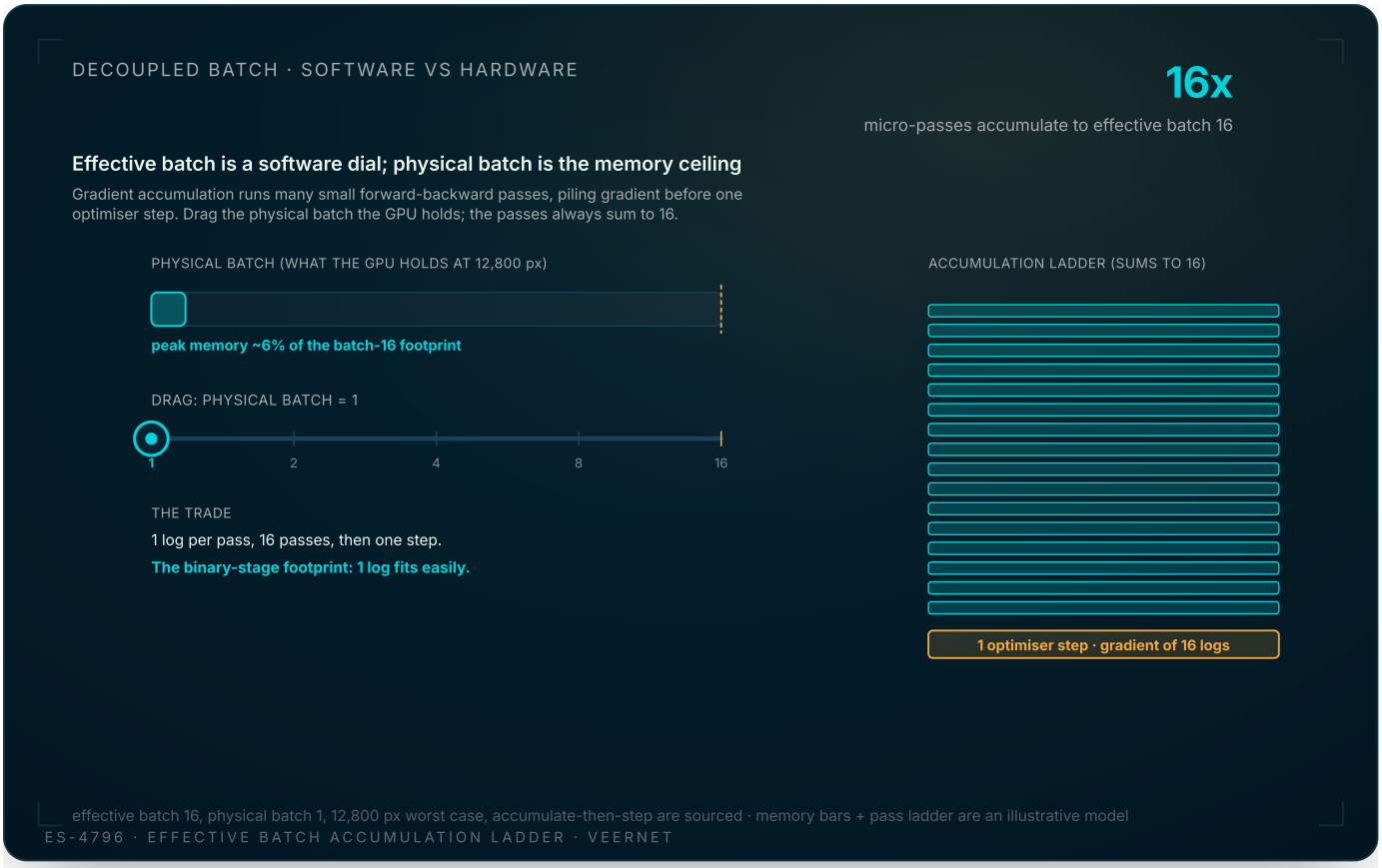
The padding-and-masking choice is the crux of the whole approach, so it is worth stating plainly why it beats the alternative. Resizing changes the data and corrupts the thin-trace target. Padding adds nothing to the data and, because of the mask, adds nothing to the loss. It costs memory, the batch tensor is as large as its biggest member times the batch size, but it preserves every pixel of every real curve at native resolution. For a target that lives in one-pixel traces, preserving native resolution is not a nicety, it is the difference between a usable mask and a smeared one.

Gradient accumulation: an effective batch of 16 without the memory bill

The collate function makes a batch of more than 1 legal. It does not make a batch of 16 affordable. A mini-batch padded to a 12,800-pixel-wide log, times the encoder-decoder activation footprint of a deep residual network with a transformer bottleneck, plus the space for backpropagation, is large, and the genuine ceiling on this whole problem is GPU memory, not arithmetic. The widest logs are what set that ceiling: a single 12,800-wide example already commits a serious slice of the device, and a literal batch of 16 such examples held in memory at once is not something the hardware budget allows.

Gradient accumulation decouples the batch the optimiser sees from the batch the GPU holds. Instead of forwarding 16 images at once, we forward a small number that fits in memory, compute the loss, scale it, and call backward, which accumulates gradients into the parameter buffers without taking an optimiser step. We repeat until 16 images' worth of gradient has piled up, then take one step and zero the buffers. The optimiser updates on the averaged gradient of 16 examples, an **effective batch of 16**, while the device never holds more than the small physical batch that fits. You trade wall-clock, because you run more forward and backward passes per optimiser step, for an effective batch the memory could never hold directly.

That effective batch of 16 is what bought back the gradient averaging the binary stage went without. The per-step gradient is now the mean of 16 logs, the single-example jitter is damped, the foreground imbalance has enough examples per step to average over, and the training is stable enough to push the harder three-class objective.



Gradient accumulation decouples the batch the optimiser sees from the batch the GPU holds. The effective batch of 16 is a software variable tuned for gradient quality; the physical batch is a hardware one pinned to the memory ceiling that the 12,800-pixel worst-case log sets. Drag the physical batch from 1 to 16 and watch the network run (16 / physical) forward-backward micro-passes, accumulating gradient before a single optimiser step on the averaged gradient of all 16 logs. A small physical batch means many passes and more wall-clock but a footprint the device can hold; a physical batch of 16 of the widest logs is over the memory ceiling. The effective batch of 16, the binary-stage physical batch of 1, the 12,800-pixel width, and the accumulate-then-step mechanic are the handover's own; the relative memory bars and the pass ladder are an illustrative linear model and are flagged as such.

The number from that stage: **550 minutes for 50 epochs** over the 15,000 multiclass instances. It is a longer run than the binary stage by exactly the factors you would predict, more than seven times the data and the overhead of accumulation, and it is a run that simply was not reachable at batch size 1 on a dataset this large and this dimensionally wide.

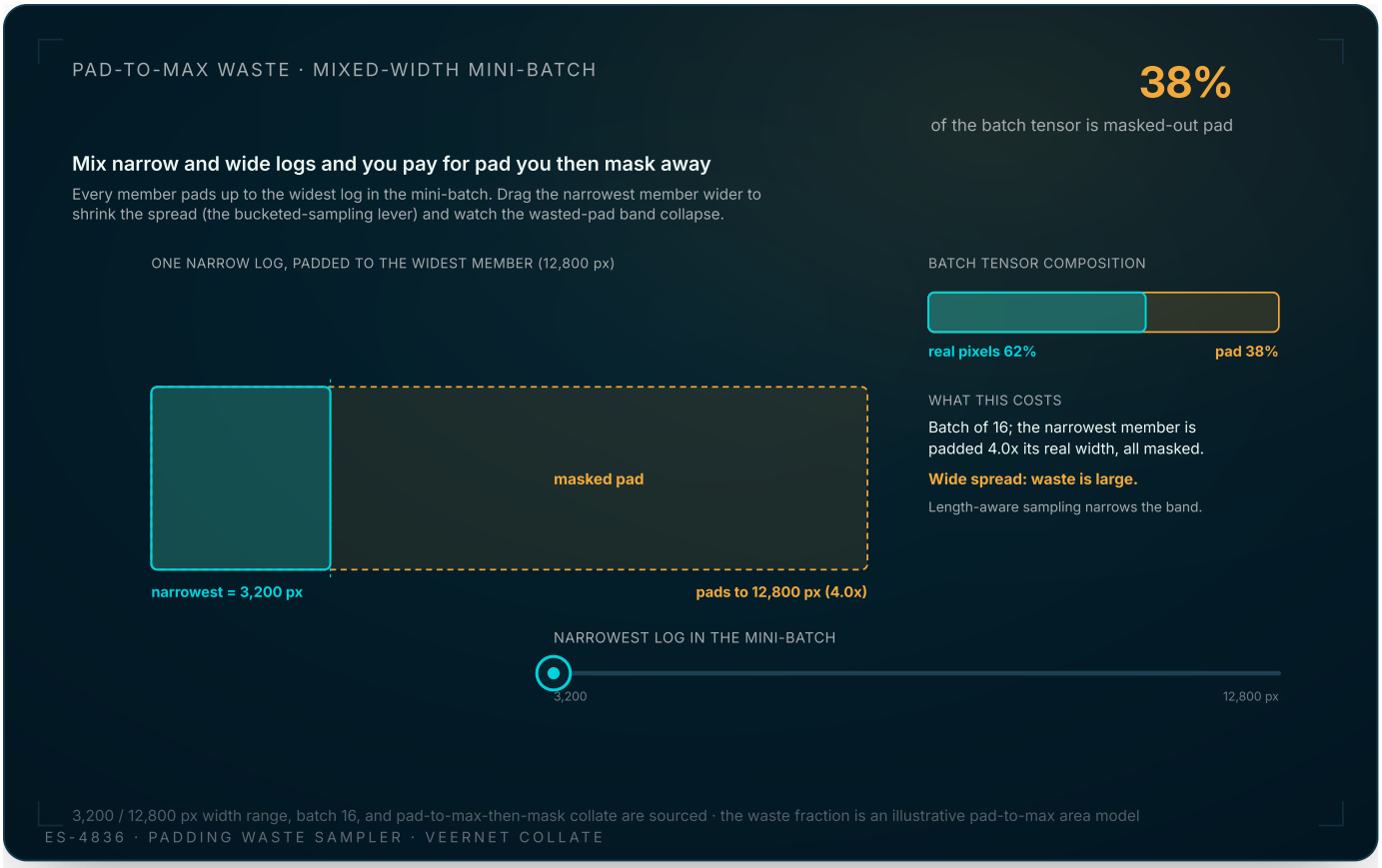
The memory arithmetic an engineer actually needs

Strip the pipeline to the constraint that governs it and you get one inequality. The peak memory of a training step is roughly the activation footprint of the network evaluated at the largest padded image in the mini-batch, multiplied by the physical batch size, plus the

parameter and gradient and optimiser-state buffers, plus the backpropagation workspace. Every term except the first is fixed. The first term is set by your widest log.

That single observation reorganises how you plan a training budget for variable-dimension data:

- **The widest image, not the average, sets your physical batch.** Planning around the mean log width is the classic mistake. A batch sampler can pull a 12,800-wide log at any moment, and the physical batch has to survive that worst case. We size the physical batch to the widest plausible member, which means the physical batch is small.
- **Effective batch is a software variable, physical batch is a hardware one.** Gradient accumulation makes effective batch a free parameter you tune for optimisation quality, while physical batch stays pinned to the memory ceiling. Decoupling them is the whole point. You pick the effective batch the optimiser wants and the physical batch the GPU tolerates, independently.
- **Padding waste is bounded by how you sample.** A mini-batch that mixes a 3,200-wide log with a 12,800-wide one pads the small one to four times its real width, and all of that pad is masked-out compute you paid for and threw away. Length-aware or bucketed sampling, grouping similarly sized logs into the same mini-batch, shrinks the padding overhead and is the natural next lever once correctness is in hand.
- **GroupNorm makes the small physical batch harmless.** Because the physical batch is forced small by the widest-log ceiling, a normalisation scheme that degrades at small batch size would quietly poison the run. GroupNorm's batch independence [4] is what lets you run a physical batch of one or two without the normalisation falling apart, which is exactly the regime the memory ceiling pushes you into.



The custom collate function pads every log in a mini-batch up to the widest member, then masks that pad out of the loss so the gradient stays honest. But masked pad is not free compute: a mini-batch that mixes a 3,200-pixel-wide log with a 12,800-pixel-wide one pads the narrow log to four times its real width, and every padded column is compute you paid for and threw away. Drag the narrowest member wider to shrink the width spread (the bucketed, length-aware sampling lever) and watch the orange wasted-pad share collapse. The 3,200 and 12,800 pixel widths, the batch of 16, and the pad-then-mask collate are the handover's own; the per-batch waste fraction is an illustrative pad-to-max area model and is flagged as such.

The clean way to hold the whole result in your head: the binary stage paid for the variable-dimension constraint in optimiser quality, accepting a noisy batch-size-1 gradient to keep the data path trivial. The multiclass stage paid for it in engineering and wall-clock instead, a custom collate function plus gradient accumulation, and bought back a real averaged gradient at an effective batch of 16. Same constraint, two different prices, and the second one is the one you want to pay once the dataset is large enough to be worth it.

What this buys, and what is still on the table

The payoff is direct. A 15,000-instance, three-class, dimensionally-unbounded dataset trains correctly and stably in 550 minutes for 50 epochs, with a gradient that honestly averages over 16 examples per step, on hardware that could never hold 16 of the widest

logs at once. None of the curve resolution is sacrificed to a resize filter, no padded pixel ever contaminates the loss, and the normalisation behaves identically regardless of how small the memory ceiling forces the physical batch to be. The model the loss ablation and the architecture work depend on is only trainable because the data path underneath it respects the variable-dimension constraint instead of papering over it.

The levers still unpulled are the obvious ones. Bucketed, length-aware batching would cut the masked-pad waste that mixed-width mini-batches incur today, recovering throughput we currently spend computing on padding we then mask away. Mixed-precision training would relax the memory ceiling that the widest logs impose, which would in turn allow a larger physical batch and fewer accumulation steps per optimiser update. Both are throughput optimisations on top of a loop that is already correct, and correctness was always the harder half. The variable-dimension batch problem is not a footnote you handle in the `DataLoader` and forget. On a real scanned-log archive it is the constraint that decides whether you have a training run at all, and engineering around it honestly, padding to enable the stack, masking to protect the gradient, accumulating to recover the batch, is what turns a pile of differently-shaped images into a model.

Get the full whitepaper

This page is the long-form summary. The complete whitepaper includes the full collate-function reference implementation, the per-stage memory profile at the 12,800-pixel worst case, the GroupNorm group-sizing derivation, and the accumulation-schedule timing breakdown.

References

[1] He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep Residual Learning for Image Recognition. CVPR. The residual-block design VeerNet's encoder is built on. https://arxiv.org/abs/1512.03385

[2] Ronneberger, O., Fischer, P., Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. MICCAI. The encoder-decoder shape for dense per-pixel prediction. https://arxiv.org/abs/1505.04597

[3] Vaswani, A., et al. (2017). Attention Is All You Need. NeurIPS. The self-attention mechanism at VeerNet's bottleneck. https://arxiv.org/abs/1706.03762

[4] Wu, Y., He, K. (2018). Group Normalization. ECCV. The normalisation scheme that stays valid when the batch is a single image. https://arxiv.org/abs/1803.08494

[5] Ioffe, S., Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML. The batch-statistic dependence that batch size 1 breaks. https://arxiv.org/abs/1502.03167

[6] Salehi, S. S. M., Erdogmus, D., Gholipour, A. (2017). Tversky Loss Function for Image Segmentation Using 3D Fully Convolutional Deep Networks. MLMI. The loss that carried the multiclass curve recovery. https://link.springer.com/chapter/10.1007/978-3-319-67389-9_44

[7] Lin, T.-Y., et al. (2017). Focal Loss for Dense Object Detection. ICCV. Class-imbalance handling for sparse foreground targets. https://arxiv.org/abs/1708.02002

EARTHSCAN

Training on One Image at a Time: solving the variable-dimension batch problem in log segmentation

Published February 2023. Generated 2026-07-22.
https://www.earthscan.io/whitepapers/training-one-image-at-a-time-variable-dimension-batches