SCRIPTS_EXPLAINED
Training Scripts — Technical Documentation

> This document explains **every script, every parameter, every decision** we made across 3 training phases — why we chose what we chose, what we expected, and what actually happened.

Table of Contents

Script Overview

Each training script follows the same pipeline:

```
1. Config & Hyperparameters
2. Dataset class (loads images + masks, applies transforms)
3. Augmentation pipeline (albumentations)
4. Model architecture (backbone + segmentation head)
5. Loss function
6. Training loop (with AMP, gradient accumulation)
7. Validation loop (IoU, Dice, Accuracy per epoch)
8. Checkpointing (save best model by val_iou)
9. Plotting (training curves, per-class IoU)
10. Metrics export (text + JSON)
```

All scripts output to `TRAINING AND PROGRESS/PHASE_X_NAME/` and are self-contained — no imports between them.

Phase 1 Script: `train_phase1_baseline.py`

Purpose

The **original hackathon-provided script**, run without modifications. Establishes the baseline metric that all improvements are measured against.

Data Pipeline

```python
# Mask encoding: raw pixel values → class IDs
value_map = {0:0, 100:1, 200:2, 300:3, 500:4, 550:5, 700:6, 800:7, 7100:8, 10000:9}
# 10 classes: Background, Trees, Lush Bushes, Dry Grass, Dry Bushes,
#             Ground Clutter, Logs, Rocks, Landscape, Sky
```

Image loading: OpenCV reads BGR → converted to RGB. Masks are uint16 single-channel.
Resize: 476×266 — chosen as nearest multiple of 14 (ViT patch size) to the original 960×540 at ~50% scale.
Normalization: ImageNet mean/std `[0.485, 0.456, 0.406]` / `[0.229, 0.224, 0.225]`
Augmentations: **None at all** — this is the baseline's biggest weakness.

Backbone — DINOv2 ViT-Small

```python
backbone = torch.hub.load("facebookresearch/dinov2", "dinov2_vits14")
backbone.eval()  # frozen — no gradients
```

Why ViT-Small? It's the hackathon's default. DINOv2 is a self-supervised vision transformer pre-trained on 142M images. ViT-Small has 384-dim embeddings with 14×14 patches.

Why frozen? With only 2857 training images, fine-tuning the entire 22M-parameter backbone would cause massive overfitting. Only the segmentation head (~500K params) is trained.

Feature extraction: `forward_features()` returns `x_norm_patchtokens` — a tensor of shape `[B, N, 384]` where N = (H/14) × (W/14) = 19 × 34 = 646 patch tokens.

Segmentation Head — ConvNeXt

Python

```python
# Simple stack: project tokens → reshape → conv layers → upsample
nn.Sequential(
    nn.Conv2d(384, 256, 3, padding=1),
    nn.BatchNorm2d(256),
    nn.GELU(),
    nn.Conv2d(256, 256, 3, padding=1),
    nn.BatchNorm2d(256),
    nn.GELU(),
    nn.Conv2d(256, 10, 1),  # classifier
)
```

How it works: Patch tokens are reshaped from (B, 646, 384) → (B, 384, 19, 34), then passed through conv layers, and bilinearly upsampled to the original 476×266.

Problem: This head only sees features at **one scale**. A 3-pixel Log and a 10,000-pixel Sky region get the exact same processing. There's no multi-scale context.

Loss & Optimizer

Python

```python
loss_fn = nn.CrossEntropyLoss()  # unweighted — treats all classes equally
optimizer = optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
# No LR scheduler
```

Why this is bad: Sky (34% of pixels) contributes 34% of the loss. Logs (0.07%) contributes 0.07%. The model has almost zero incentive to learn rare classes.

SGD at 1e-4: Extremely slow convergence. SGD needs large learning rates and momentum to work well, but this LR is conservative. AdamW would be 3-5× faster.

Training Loop

Python

```python
for epoch in range(10):  # only 10 epochs!
    for imgs, labels in train_loader:
        tokens = backbone.forward_features(imgs)["x_norm_patchtokens"]
        logits = model(tokens)
        out = F.interpolate(logits, size=imgs.shape[2:], mode="bilinear")
        loss = loss_fn(out, labels.squeeze(1).long())
        loss.backward()
```

```
        optimizer.step()
        optimizer.zero_grad()
```

No mixed precision — runs in fp32, using more VRAM than necessary.
No gradient accumulation — effective batch = actual batch = 2.
No checkpointing — only the final model is saved, not the best.

Expected vs Actual

Aspect	Expected	Actual
IoU	0.25-0.35 (typical for frozen ViT-S)	**0.2971** ✅ in range
Convergence	Should plateau by epoch 10	**Still improving** ❌ — underfitting
Overfitting	Possible without augmentations	**None** ✅ — train ≈ val
Rare classes	Very low IoU expected	Logs=0.05, Rocks=0.16 — **terrible** as expected
Training time	~1-2 hours	**83 min** ✅

Key takeaway: The model was nowhere near converged. Loss decreased linearly through all 10 epochs — we left massive performance on the table.

Phase 2 Script: `train_phase2_improved.py`

What Changed and Why

Every change was a **direct response to Phase 1's failures**.

Change 1: SGD → AdamW

Python

```python
# Phase 1
optimizer = optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)

# Phase 2
optimizer = optim.AdamW(model.parameters(), lr=5e-4, weight_decay=1e-4)
```

Why: SGD at 1e-4 was painfully slow. AdamW uses per-parameter adaptive learning rates — parameters with small gradients get larger effective LR. This is critical for segmentation

where different parts of the head learn at different rates.

Why lr=5e-4? Standard starting LR for AdamW in fine-tuning tasks. Higher than SGD's 1e-4 because AdamW self-adjusts.

Why weight_decay=1e-4? Light regularization to prevent overfitting without being too aggressive. Common default for vision tasks.

Change 2: CosineAnnealingLR

Python

```python
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=30, eta_min=1e-6)
```

Why: A constant LR means the model either learns too aggressively (high LR → unstable) or too slowly (low LR → never converges). Cosine annealing starts high for fast initial learning, then gradually decreases to fine-tune.

Why T_max=30? Matches the total epoch count so LR reaches its minimum at the last epoch.

Why eta_min=1e-6? Nearly zero but not exactly zero — allows tiny parameter updates in the final epochs for boundary refinement.

Expected: IoU should improve faster initially and plateau more gracefully.
Actual: ✅ Worked perfectly. Best IoU at epoch 26 (LR ≈ 2.3e-5), showing the schedule found the sweet spot.

Change 3: Data Augmentations

Python

```python
A.Compose([
    A.Resize(h, w),
    A.HorizontalFlip(p=0.5),              # Scenes are symmetric
    A.VerticalFlip(p=0.1),                # Rare but adds variety
    A.ShiftScaleRotate(                   # Simulates camera movement
        shift_limit=0.08, scale_limit=0.15, rotate_limit=20,
        border_mode=cv2.BORDER_CONSTANT, value=0, p=0.4
    ),
    A.OneOf([
        A.GaussianBlur(blur_limit=(3,5)),  # Simulates defocus
        A.MedianBlur(blur_limit=5),        # Simulates sensor noise
    ], p=0.2),
```

```
        A.OneOf([
            A.RandomBrightnessContrast(...),   # Lighting variation
            A.HueSaturationValue(...),         # Color shift
        ], p=0.4),
        A.Normalize(...),
        ToTensorV2(),
    ])
```

Why these specific augmentations?

- **HFlip (p=0.5)**: Desert scenes look the same left-right flipped. Free 2× data.
- **ShiftScaleRotate**: Simulates different camera angles and distances. `border_mode=CONSTANT` fills edges with black rather than reflecting.
- **Blur**: Simulates camera defocus and motion blur in offroad conditions.
- **Color jitter**: Desert lighting varies hugely — sunrise, midday, sunset all look different.

What we deliberately avoided:

- `RandomRotate90` — crashed training because it swaps dimensions on non-square images, breaking `torch.stack` in the DataLoader. We discovered this bug and removed it.
- Heavy geometric distortions — would distort object shapes too much for segmentation.

Expected: 10-15% IoU improvement from augmentations alone.
Actual: Augmentations + AdamW together gave +35.8%, hard to isolate individual contribution.

Change 4: Weighted CrossEntropy

Python

```python
# Compute class weights from pixel frequency
counts = torch.zeros(10)
for _, masks in train_loader:
    for c in range(10):
        counts[c] += (masks == c).sum()
weights = total / (10 * counts)  # Inverse frequency
weights = weights / weights.max() * 5.0  # Cap at 5.0

loss_fn = nn.CrossEntropyLoss(weight=weights)
```

Why inverse frequency? If Sky has 34% of pixels, its weight becomes 0.01 (near-zero). If Logs has 0.07%, its weight becomes 5.0 (maximum). This forces the model to "care" about rare classes proportionally.

Why cap at 5.0? Without capping, Logs would get weight ~72. This would make the model hallucinate Logs everywhere because the loss rewards even false Logs detections heavily.

Expected: Big boost for rare classes (Logs, Rocks).
Actual: Partial success. Logs went from unlearnable to IoU=0.05 (barely visible). The fundamental problem is that weighted CE still treats each pixel independently — it doesn't optimize for region overlap.

Change 5: Mixed Precision

Python

```python
scaler = torch.amp.GradScaler('cuda')
with torch.amp.autocast('cuda'):
    logits = model(tokens)
    loss = loss_fn(logits, labels)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Why: Our RTX 3050 has only 6GB VRAM. Mixed precision (fp16 forward, fp32 backward) reduces memory by ~30% and speeds up training by ~15% on Tensor Cores.

Expected: Faster training, no accuracy loss.
Actual: ✅ ~15% speedup, identical metrics to fp32 training in tests.

Change 6: Checkpointing & Early Stopping

Python

```python
if val_iou > best_iou:
    best_iou = val_iou
    torch.save(model.state_dict(), 'best_model.pth')
    no_improve = 0
else:
    no_improve += 1
    if no_improve >= 10:  # patience
        break
```

Why checkpointing? Phase 1 only saved the final model. If the best epoch was #26 but training ran to #30, we'd lose the best weights.

Why patience=10? Gives the model time to escape local minima. Too low (3-5) might stop too early; too high (20+) wastes time.

Expected: Save the true best model.
Actual: ✅ Best at epoch 26, saved correctly. Early stopping not triggered in 30 epochs.

Expected vs Actual (Phase 2)

Aspect	Expected	Actual
IoU	0.45-0.50	**0.4036** ❌ lower than expected
Best epoch	~20-25	**26** ✅ close
Training time	~3-4 hours	**4.1 hours** ✅
Rare class boost	Logs > 0.15	**Logs = 0.05** ❌ still terrible
Overfitting	Low risk with augs	**None** ✅

Why lower than expected? We overestimated what a simple ConvNeXt head could do. The bottleneck wasn't optimization — it was **architecture**. The single-scale head fundamentally cannot handle the 500× size difference between Sky and Logs.

Phase 3 Script: `train_phase3_advanced.py`

What Changed and Why

Every change was a **direct response to Phase 2's architectural limitations**.

Change 1: ViT-Small → ViT-Base

Python

```python
# Phase 2
backbone = torch.hub.load("facebookresearch/dinov2", "dinov2_vits14")
# 384-dim embeddings, 22M params

# Phase 3
backbone = torch.hub.load("facebookresearch/dinov2", "dinov2_vitb14_reg")
# 768-dim embeddings, 86M params, with register tokens
```

Why ViT-Base? Phase 2's per-class analysis showed failure on texturally similar classes:

- Dry Bushes (0.28) confused with Dry Grass (0.48) — both brownish
- Lush Bushes (0.41) confused with Trees (0.50) — both green
- Ground Clutter (0.22) confused with Background (0.45)

ViT-Small's 384-dim features don't have enough capacity to encode **texture differences between similar classes**. ViT-Base doubles to 768-dim, giving the model more "vocabulary" to describe what it sees.

Why `_reg` variant? The `dinov2_vitb14_reg` model includes register tokens that reduce attention artifacts in ViT, producing smoother feature maps.

VRAM concern: ViT-Base uses ~2× more VRAM than ViT-Small for feature extraction. We solved this with gradient accumulation and mixed precision.

Expected: +5-10% IoU from richer features alone.
Actual: Epoch 1 IoU=0.42 (already above Phase 2's best 0.40!) confirming the backbone quality. ✅

Change 2: ConvNeXt → UPerNet

Python

```python
class PPM(nn.Module):
    """Pyramid Pooling Module — captures context at multiple scales."""
    def __init__(self, in_channels, pool_sizes=(1, 2, 3, 6)):
        # For each pool size, global average pool → 1×1 conv → upsample back
        # pool_size=1: global context (entire image summary)
        # pool_size=2: 2×2 grid (quadrant-level context)
        # pool_size=3: 3×3 grid (region-level context)
        # pool_size=6: 6×6 grid (local context)

class UPerNetHead(nn.Module):
    """Multi-scale FPN with dilated convolutions."""
    def __init__(self, in_channels=768, mid_channels=256, out_channels=10):
        # 1. Input projection: 768 → 256 channels
        # 2. PPM: captures global-to-local context
        # 3. Multi-scale FPN:
        #    - dilation=1: 3×3 receptive field (fine details like Logs)
        #    - dilation=2: 5×5 receptive field (medium objects like Rocks)
        #    - dilation=4: 9×9 receptive field (large areas like Landscape)
        # 4. Fusion: concat 3 scales → 1×1 conv → classifier
```

Why UPerNet? Phase 2's ConvNeXt head failed because:

- It processed all spatial locations with the same 3×3 conv
- A 3-pixel Log and a 10K-pixel Sky get identical processing

- No global scene understanding — can't use "this is the top of the image" as context for Sky

UPerNet fixes this with:

1. **PPM**: At pool_size=1, the model sees the ENTIRE image summarized as one vector. This provides **scene understanding** — "this is a desert scene with sky at top."
2. **Multi-scale FPN**: dilation=1 detects fine edges (Logs, small Rocks), dilation=4 captures wide patterns (Landscape boundaries).

Why GroupNorm instead of BatchNorm?

Python

```python
# PPM creates 1×1 spatial tensors via AdaptiveAvgPool2d(1)
# BatchNorm requires spatial_size > 1 → CRASHES
nn.BatchNorm2d(channels)  # ❌ fails at 1×1

# GroupNorm normalizes across channel groups, works at ANY spatial size
nn.GroupNorm(32, channels)  # ✅ always works
```

We discovered this bug when Phase 3 crashed on the first forward pass. GroupNorm(32, channels) splits the 256 channels into 32 groups and normalizes each group independently.

Expected: +10-15% IoU from multi-scale features.
Actual: Hard to isolate, but the per-class improvements in Landscape (+51%), Rocks (+92%), and Logs (+382%) strongly suggest the multi-scale processing was the key driver. ✅

Change 3: CrossEntropy → Focal + Dice

Python

```python
class FocalLoss(nn.Module):
    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets, weight=self.alpha, reduction='none')
        pt = torch.exp(-ce_loss)   # probability of correct class
        focal_loss = ((1 - pt) ** self.gamma) * ce_loss
        # When pt ≈ 1 (easy pixel): focal_weight ≈ 0 → ignored
        # When pt ≈ 0 (hard pixel): focal_weight ≈ 1 → full weight
        return focal_loss.mean()

class DiceLoss(nn.Module):
    def forward(self, inputs, targets):
```

```
        # Computes per-class Dice coefficient and averages
        # Unlike CE, Dice treats each class equally regardless of pixel count
        # Logs (0.07%) gets the same weight as Sky (34%)
        for c in range(num_classes):
            dice += (2 * intersection + smooth) / (prediction + target + smooth)
        return 1 - dice / num_classes

# Combined: Focal handles hard pixels, Dice handles class balance
loss = 1.0 * FocalLoss(alpha=class_weights, gamma=2.0) + \
       0.5 * DiceLoss(num_classes=10)
```

Why γ=2.0 for Focal? Standard choice from the original Focal Loss paper. γ=2 means a pixel classified with 90% confidence gets its loss reduced by 100× compared to a 50% confidence pixel. This forces the model to focus on uncertain boundaries.

Why 0.5 weight for Dice? Dice Loss tends to produce larger gradients than Focal Loss. Weighting it at 0.5 prevents it from dominating the combined gradient.

Why not just Dice alone? Dice Loss can be unstable early in training when predictions are near-random. Focal Loss provides stable pixel-level gradients while Dice provides the region-level optimization signal.

Expected: Major boost for Logs and Rocks.
Actual: Logs 0.05 → 0.25 (+382%), Rocks 0.16 → 0.32 (+92%). ✅ Exactly what we hoped.

Change 4: Higher Resolution (644×364)

Python

```
# Phase 1-2: 476×266 → 34×19 = 646 patch tokens
# Phase 3:   644×364 → 46×26 = 1196 patch tokens
w = 46 * 14  # 644 (must be multiple of 14 for ViT)
h = 26 * 14  # 364
```

Why higher resolution? At 476×266, a Log might occupy only 3-5 pixels. That's too small for ANY architecture to segment reliably. At 644×364 (84% more pixels), Logs become 5-8 pixels — still small but detectable.

Trade-off: 1196 patch tokens (vs 646) means ~1.85× more computation in the backbone. Combined with ViT-Base's larger model, each epoch takes ~10 min vs ~8 min in Phase 2.

Why not even higher (e.g., 960×540)? Would exceed 6GB VRAM even with AMP. The 644×364 balance was chosen to maximize resolution within our GPU constraints.

Change 5: Gradient Accumulation

Python

```python
accum_steps = 2  # accumulate gradients over 2 batches
for step, (imgs, labels) in enumerate(train_loader):
    loss = loss_fn(output, labels) / accum_steps  # scale loss
    scaler.scale(loss).backward()

    if (step + 1) % accum_steps == 0:  # update every 2 steps
        scaler.step(optimizer)
        scaler.update()
        optimizer.zero_grad()
```

Why? Actual batch_size=2 (VRAM limit), but effective batch_size=4. Larger effective batches provide more stable gradient estimates, especially important with class-weighted Focal Loss where rare-class gradients can be noisy.

Why accum=2 (not 4 or 8)? Each accumulation step doubles memory for stored activations. 2 steps keeps us well within 6GB.

Change 6: Warmup Scheduler

Python

```python
def lr_lambda(epoch):
    if epoch < warmup_epochs:  # epochs 0, 1, 2
        return (epoch + 1) / warmup_epochs  # 0.33, 0.67, 1.0
    progress = (epoch - warmup_epochs) / (n_epochs - warmup_epochs)
    return 0.5 * (1 + np.cos(np.pi * progress))

scheduler = optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)
```

Why warmup? Phase 2's CosineAnnealing started at peak LR immediately. This was fine for Phase 2's simpler head, but Phase 3's UPerNet has more parameters (PPM + FPN + fusion). Starting with full LR on randomly initialized weights causes large, unstable gradients.

Why 3 epochs? Standard heuristic: warmup for 5-10% of total training. 3/40 = 7.5%.

LR progression: 1e-4 → 2e-4 → 3e-4 (warmup) → cosine decay → 0.

Change 7: Stronger Augmentations

Python

```
# New in Phase 3:
A.RandomShadow(p=0.15),  # Simulates shadow from trees/cliffs
A.CLAHE(clip_limit=2.0, p=0.15),  # Enhance local contrast
```

Why RandomShadow? Desert offroad scenes have harsh shadows from boulders, cliffs, and vegetation. Training with synthetic shadows makes the model robust to lighting changes.

Why CLAHE? Contrast-Limited Adaptive Histogram Equalization enhances local detail, making subtle textures (Ground Clutter, Dry Bushes) more distinct during training.

Expected vs Actual (Phase 3)

Aspect	Expected	Actual
IoU	0.55+	**0.5161** ❌ close but short
Epoch 1 IoU	> Phase 2 best (0.40)	**0.4219** ✅ already above
IoU > 0.50	By epoch 10-15	**Epoch 16** ✅ close
Logs IoU	> 0.20	**0.2495** ✅ exceeded
Rocks IoU	> 0.25	**0.3167** ✅ exceeded
Training time	~6-8 hours	**~7 hours** ✅
Convergence	By epoch 35	**Still improving at 40** ❌ — more epochs needed

Why we didn't reach 0.55? The model was still improving at epoch 40 (IoU went from 0.515 → 0.516 in the last 5 epochs). With 60-80 epochs, we'd likely reach 0.52-0.53. The remaining gap to 0.55 would require:

- **Copy-paste augmentation** for Logs/Rocks (paste rare objects onto training images)
- **Backbone fine-tuning** (carefully unfreeze last 2-3 ViT layers)
- **Test-time augmentation** (average predictions from flipped/scaled inputs)

Inference: `test_segmentation.py`

Runs the best trained model on test images from `DATASET/Offroad_Segmentation_testImages/`.

What it does:

1. Loads the best `segmentation_head.pth` weights
2. For each test image: resize → normalize → backbone → head → argmax → color map

3. Saves predicted mask overlays for visual inspection

Key parameters:

- Uses the same image size as the training phase (644×364 for Phase 3)
- Same normalization (ImageNet mean/std)
- No augmentations at inference (deterministic)

Visualization: `visualize.py`

Helper utilities:

- **Overlay**: Blends RGB image with predicted mask at configurable opacity
- **Color map**: Maps class IDs to distinct colors for visualization
- **Side-by-side**: Shows image | ground truth | prediction for comparison

---

Parameter Evolution Table

Parameter	Phase 1	Phase 2	Phase 3	Why Changed
Backbone	ViT-S (384d)	ViT-S (384d)	**ViT-B (768d)**	P2 couldn't distinguish similar textures
Head	ConvNeXt	ConvNeXt	**UPerNet**	P2's single-scale missed small objects
Loss	CE	Weighted CE	**Focal + Dice**	Weighted CE didn't help Logs enough
Optimizer	SGD 1e-4	AdamW 5e-4	AdamW 3e-4	P1's SGD was too slow
LR Schedule	None	CosineAnnealing	**Warmup + Cosine**	UPerNet needs warmup for stability
Image Size	476×266	476×266	**644×364**	Small objects (Logs) were sub-pixel
Batch	2	2	2 (eff. 4)	ViT-B needs more stable gradients
Epochs	10	30	40	P1 undertrained, P2 converged at 26
Augmentations	None	5 types	**7 types**	P1 had none, P3 adds shadow/CLAHE

Parameter	Phase 1	Phase 2	Phase 3	Why Changed
Normalization	BatchNorm	BatchNorm	**GroupNorm**	BatchNorm fails at 1×1 spatial (PPM)
Mixed Precision	No	Yes	Yes	6GB VRAM constraint
Checkpointing	Final only	Best by IoU	Best by IoU	P1 didn't save best
Early Stopping	None	Patience=10	Patience=12	P3 trained longer, needed more patience

---

Lessons Learned

1. Architecture > Optimization

Phase 2 proved that better optimization (AdamW vs SGD, 30 vs 10 epochs) can only go so far. The ConvNeXt head held us at 0.40 regardless of how well we optimized. Phase 3's UPerNet broke through to 0.52. **A better model architecture provides gains that no amount of hyperparameter tuning can match.**

2. Rare Class Performance Requires Multiple Strategies

Weighted CE alone gave Logs IoU=0.05. Adding Focal Loss raised it to 0.25. But even with Focal + Dice + higher resolution + stronger augmentations, Logs is still the worst class (0.25). **Extreme class imbalance (72× rarer than average) requires data-level solutions** like copy-paste augmentation or synthetic oversampling, not just loss-level fixes.

3. Resolution Matters for Small Objects

Going from 476×266 → 644×364 (+84% pixels) gave disproportionate gains for small-object classes (Rocks +92%, Logs +382%). At low resolution, these objects are 2-4 pixels — below the detection threshold for any model.

4. Warmup Prevents Wasted Early Epochs

Phase 3's warmup prevented the early instability we saw in some Phase 2 test runs. The first 3 epochs with gradually increasing LR let the randomly initialized UPerNet head "warm up" before receiving full-strength gradient updates.

5. GroupNorm is Mandatory for PPM

This was a production bug: BatchNorm crashes silently when spatial size = 1×1 (from AdaptiveAvgPool2d(1)). GroupNorm has no spatial-size dependency. **Always use GroupNorm in pooling-heavy architectures.**

6. The Model Was Still Improving

Phase 3 reached IoU=0.5161 at epoch 40 — but it was still improving (0.5154 → 0.5161 in last 4 epochs). Early stopping never triggered (patience=12). With 60+ epochs, we'd expect 0.52+ IoU. **Don't cut training short if the model is still learning.**