

AADS-ULoRA v5.5 Implementation Guide – Part 1

Environment Setup, Crop Router, Phase 1 Training, Dynamic OOD Configuration

Agricultural AI Development Team

March 2026–Version

Contents

| | | |
|----------|--|-----------|
| 1 | Critical Changes from v5.4 | 2 |
| 1.1 | What Was Enhanced | 2 |
| 1.2 | What Was Kept | 2 |
| 2 | Environment Setup | 2 |
| 3 | Simple Crop Router Implementation | 3 |
| 3.1 | Crop Router Architecture | 4 |
| 3.2 | Training the Crop Router | 6 |
| 4 | Independent Crop Adapter Architecture | 7 |
| 5 | Phase 1 Training Details | 9 |
| 6 | Mahalanobis Prototype and Dynamic OOD Computation | 11 |
| 7 | Summary | 13 |

1 Critical Changes from v5.4

AADS-ULoRA v5.5 enhances v5.4’s independent architecture with improved OOD detection through dynamic, per-class Mahalanobis thresholds. This eliminates the need for manual threshold tuning while improving detection accuracy across variable disease classes.

1.1 What Was Enhanced

Table 1: v5.4 to v5.5 Enhancements

| Component | v5.4 | v5.5 |
|-----------------------|------------------------|--------------------------------------|
| OOD Thresholds | Fixed global values | Dynamic per-class computation |
| Threshold Computation | Manual tuning | Automatic from validation statistics |
| Detection Sensitivity | Uniform across classes | Adapted to class variability |

1.2 What Was Kept

- Simple crop router (replaces SEMA)
- Independent crop adapters (no cross-adapter interference)
- DoRA for Phase 1 base initialization
- SD-LoRA for Phase 2 class-incremental learning
- CONEC-LoRA for Phase 3 data-incremental learning
- Mahalanobis++ OOD detection framework

2 Environment Setup

```
1 """
2 =====
3 AADS-ULoRA v5.5 Independent Multi-Crop - Environment Setup
4 =====
5 Run this first in Google Colab Pro
6 """
7
8 import torch
9 import sys
10 import os
11
12 print("=="*60)
13 print("AADS-ULoRA v5.5 Independent Multi-Crop Setup")
14 print("=="*60)
15
16 # Step 1: Verify GPU
17 print("\n[Step 1] Verifying GPU...")
18
19 assert torch.cuda.is_available(), "ERROR: No GPU available!"
20 gpu_name = torch.cuda.get_device_name(0)
21 print(f"CUDA available: {torch.cuda.is_available()}")
22 print(f"GPU: {gpu_name}")
23
24 if "A100" not in gpu_name:
25     print(f"WARNING: Not an A100! Got {gpu_name}")
26     print("Training may be slower or OOM")
```

```

27
28 total_mem = torch.cuda.get_device_properties(0).total_memory / 1e9
29 print(f"GPU Memory: {total_mem:.1f} GB")
30
31 BATCH_SIZE = 8 if total_mem < 45 else 16
32
33 # Step 2: Install Dependencies
34 print("\n[Step 2] Installing dependencies...")
35
36 !pip install -q transformers==4.56.0 --upgrade
37 !pip install -q peft>=0.8.0 --upgrade
38 !pip install -q accelerate bitsandbytes
39 !pip install -q scikit-learn matplotlib seaborn
40 !pip install -q tqdm pillow opencv-python
41 !pip install -q albumentations
42
43 import transformers
44 import peft
45
46 print(f"Transformers: {transformers.__version__}")
47 print(f"PEFT: {peft.__version__}")
48
49 # Verify PEFT supports DoRA
50 peft_version = tuple(map(int, peft.__version__.split('.'))[:2])
51 assert peft_version >= (0, 8), "PEFT must be >= 0.8.0 for DoRA!"
52 print(f"PEFT {peft.__version__} supports DoRA")
53
54 # Step 3: Mount Google Drive
55 print("\n[Step 3] Mounting Google Drive...")
56
57 from google.colab import drive
58 drive.mount('/content/drive')
59
60 # Step 4: Create v5.5 Directory Structure
61 ROOT = '/content/drive/MyDrive/AADS_v55_Independent'
62 folders = [
63     'data',           # Per-crop data
64     'adapters',       # Independent crop adapters
65     'router',          # Crop router model
66     'checkpoints',    # Training checkpoints
67     'prototypes',      # Per-crop prototypes with OOD stats
68     'logs',            # Training logs
69     'ood_stats'        # Per-class OOD threshold statistics
70 ]
71
72 for folder in folders:
73     os.makedirs(f"{ROOT}/{folder}", exist_ok=True)
74     print(f"Created {ROOT}/{folder}")
75
76 print("\nEnvironment setup complete!")
77 print(f"Batch size: {BATCH_SIZE}")
78 print(f"Project root: {ROOT}")

```

Listing 1: Environment Setup for v5.5

3 Simple Crop Router Implementation

Unlike v5.3's SEMA-based expansion, v5.5 uses a simple crop classifier (unchanged from v5.4). This is a much easier task (crop identification vs. disease detection) and achieves 98%+ accuracy with minimal training.

3.1 Crop Router Architecture

```
1 import torch
2 import torch.nn as nn
3 from transformers import AutoModel
4 from typing import List
5
6 class SimpleCropRouter:
7     """
8         Lightweight crop classifier using DINoV2 linear probe.
9         Much simpler than SEMA - just identifies which crop.
10
11     Literature: Standard transfer learning
12     Target: 98%+ crop classification accuracy
13     """
14
15     def __init__(self, crops: List[str], device='cuda'):
16         self.crops = crops
17         self.device = device
18
19         # Load DINoV2 as feature extractor (smaller base model)
20         self.backbone = AutoModel.from_pretrained(
21             'facebook/dinov2-base' # 86M params vs 1.1B for giant
22         ).to(device)
23         self.backbone.eval()
24
25         # Freeze backbone
26         for param in self.backbone.parameters():
27             param.requires_grad = False
28
29         # Linear classifier for crop types
30         self.classifier = nn.Linear(768, len(crops)).to(device)
31
32         print(f"Crop router initialized for {len(crops)} crops")
33
34     def train(self, crop_dataset, epochs=10, lr=1e-3):
35         """
36             Train crop classifier on labeled crop images.
37
38             Args:
39                 crop_dataset: DataLoader with (images, crop_labels)
40                 epochs: Training epochs
41                 lr: Learning rate
42
43             Target: 98% accuracy (easy task)
44             """
45
46         optimizer = torch.optim.AdamW(
47             self.classifier.parameters(),
48             lr=lr
49         )
50         criterion = nn.CrossEntropyLoss()
51
52         best_acc = 0.0
53
54         for epoch in range(epochs):
55             self.classifier.train()
56             total_loss = 0
57             correct = 0
58             total = 0
59
60             for images, labels in crop_dataset:
61                 images = images.to(self.device)
62                 labels = labels.to(self.device)
```

```

62     # Extract features (frozen backbone)
63     with torch.no_grad():
64         features = self.backbone(images).last_hidden_state[:, 0]
65
66     # Classify crop
67     logits = self.classifier(features)
68     loss = criterion(logits, labels)
69
70     # Backward
71     optimizer.zero_grad()
72     loss.backward()
73     optimizer.step()
74
75     # Metrics
76     total_loss += loss.item()
77     _, predicted = logits.max(1)
78     total += labels.size(0)
79     correct += predicted.eq(labels).sum().item()
80
81     acc = 100.0 * correct / total
82     avg_loss = total_loss / len(crop_dataset)
83
84     print(f"Epoch {epoch+1}/{epochs}: Loss={avg_loss:.4f}, Acc={acc:.2f} %")
85
86     if acc > best_acc:
87         best_acc = acc
88         self.save_checkpoint(f'router_epoch{epoch+1}.pth')
89
90     print(f"\nBest crop routing accuracy: {best_acc:.2f}%")
91     return best_acc
92
93 def route(self, image: torch.Tensor) -> str:
94     """
95     Predict crop type from image.
96
97     Args:
98         image: Tensor [1, 3, H, W]
99
100    Returns:
101        crop_name: One of self.crops
102    """
103    self.classifier.eval()
104
105    with torch.no_grad():
106        features = self.backbone(image).last_hidden_state[:, 0]
107        logits = self.classifier(features)
108        crop_idx = logits.argmax(dim=1).item()
109
110    return self.crops[crop_idx]
111
112 def save_checkpoint(self, path):
113     """Save router checkpoint."""
114     torch.save({
115         'classifier': self.classifier.state_dict(),
116         'crops': self.crops
117     }, path)
118
119 def load_checkpoint(self, path):
120     """Load router checkpoint."""
121     checkpoint = torch.load(path)
122     self.classifier.load_state_dict(checkpoint['classifier'])

```

```
123     self.crops = checkpoint['crops']
```

Listing 2: Simple Crop Router Class

3.2 Training the Crop Router

```
1 from torch.utils.data import Dataset, DataLoader
2 from PIL import Image
3 import albumentations as A
4 from albumentations.pytorch import ToTensorV2
5 import numpy as np
6
7 class CropClassificationDataset(Dataset):
8     """
9         Dataset for crop type classification.
10        Can use PlantCLEF or custom crop images.
11    """
12    def __init__(self, image_paths, crop_labels, transform=None):
13        self.image_paths = image_paths
14        self.crop_labels = crop_labels # Numeric labels
15        self.transform = transform
16
17    def __len__(self):
18        return len(self.image_paths)
19
20    def __getitem__(self, idx):
21        image = Image.open(self.image_paths[idx]).convert('RGB')
22        image = np.array(image)
23
24        if self.transform:
25            image = self.transform(image=image) ['image']
26
27        return image, self.crop_labels[idx]
28
29 # Augmentation for crop router
30 crop_transform = A.Compose([
31     A.Resize(224, 224),
32     A.HorizontalFlip(p=0.5),
33     A.VerticalFlip(p=0.3),
34     A.RandomBrightnessContrast(p=0.2),
35     A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
36     ToTensorV2()
37 ])
38
39 # Create dataset
40 crops = ['tomato', 'pepper', 'corn']
41 train_dataset = CropClassificationDataset(
42     image_paths=train_image_paths,
43     crop_labels=train_crop_labels, # 0=tomato, 1=pepper, 2=corn
44     transform=crop_transform
45 )
46
47 train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
48
49 # Train router
50 router = SimpleCropRouter(crops=crops, device='cuda')
51 router.train(train_loader, epochs=10)
52
53 # Validate
54 val_acc = evaluate_router(router, val_loader)
55 print(f"Validation accuracy: {val_acc:.2f}%")
```

```
56 assert val_acc >= 95.0, "Crop router accuracy too low!"
```

Listing 3: Crop Router Training Script

4 Independent Crop Adapter Architecture

```
1 from peft import LoraConfig, get_peft_model, TaskType
2 import torch
3 import torch.nn as nn
4 from typing import Dict, List, Optional
5 import numpy as np
6 from sklearn.covariance import EmpiricalCovariance
7
8 class IndependentCropAdapter:
9     """
10         Self-contained adapter for one crop with dynamic OOD detection.
11         No communication with other crop adapters.
12
13     Lifecycle:
14         - Phase 1: DoRA base initialization
15         - Phase 2: SD-LoRA add new diseases
16         - Phase 3: CONEC-LoRA fortify existing diseases
17
18     OOD Detection:
19         - Dynamic per-class Mahalanobis thresholds
20         - Computed from validation data statistics
21     """
22     def __init__(self, crop_name: str, device='cuda'):
23         self.crop_name = crop_name
24         self.device = device
25         self.phase = 0
26         self.disease_classes = []
27
28         # Load frozen DINOV2 backbone
29         from transformers import AutoModel
30         self.backbone = AutoModel.from_pretrained(
31             'facebook/dinov2-giant'
32         ).to(device)
33
34         # Freeze backbone
35         for param in self.backbone.parameters():
36             param.requires_grad = False
37
38         # DoRA adapter (initialized in Phase 1)
39         self.adapter = None
40         self.classifier = None
41         self.prototypes = None
42
43         # Dynamic OOD statistics
44         self.ood_stats = {
45             'class_means': {},          # Per-class Mahalanobis mean
46             'class_stds': {},           # Per-class Mahalanobis std
47             'threshold_factor': 2.0    # k sigma factor (95% confidence)
48         }
49
50         print(f"Initialized adapter for {crop_name}")
51
52     def phase1_initialize(self, train_data, val_data, config):
53         """
54             Phase 1: Base initialization with DoRA.
55             Computes dynamic OOD thresholds from validation data.
56
```

```

57     Literature: Liu et al. (2024) - DoRA
58     Target: 95%+ clean accuracy
59     """
60
61     # DoRA configuration
62     lora_config = LoraConfig(
63         task_type=TaskType.FEATURE_EXTRACTION,
64         r=config['lora_r'],
65         lora_alpha=config['lora_alpha'],
66         use_dora=True, # Enable DoRA decomposition
67         target_modules=['query', 'value'],
68         lora_dropout=0.1
69     )
70
71     # Apply DoRA to backbone
72     self.adapter = get_peft_model(self.backbone, lora_config)
73
74     # Add classifier head
75     self.classifier = nn.Linear(
76         1536, # DINOv2-giant output dim
77         len(train_data.classes)
78     ).to(self.device)
79
80     # Train Phase 1
81     self._train_phase1(train_data, config)
82
83     # Compute Mahalanobis prototypes
84     self.prototypes = self._compute_prototypes(train_data)
85
86     # Compute dynamic OOD thresholds from validation data
87     self._compute_dynamic_ood_thresholds(val_data)
88
89     self.disease_classes = train_data.classes
90     self.phase = 1
91
92     print(f"\n{self.crop_name} Phase 1 complete")
93     print(f"Disease classes: {self.disease_classes}")
94     print(f"OOD thresholds computed for {len(self.disease_classes)} classes")
95 )
96
97 def _compute_dynamic_ood_thresholds(self, val_data):
98     """
99     Compute per-class Mahalanobis distance statistics on validation data.
100    These determine dynamic OOD thresholds.
101
102    self.adapter.eval()
103    self.classifier.eval()
104
105    # Collect features and predictions per class
106    class_distances = {cls: [] for cls in range(len(self.disease_classes))}
107
108    with torch.no_grad():
109        for images, labels in val_data:
110            images = images.to(self.device)
111
112            # Forward pass
113            features = self.adapter(images).last_hidden_state[:, 0]
114            logits = self.classifier(features)
115            predictions = logits.argmax(dim=1)
116
117            # Compute Mahalanobis distance for each sample
118            for i, (feat, pred) in enumerate(zip(features, predictions)):
119                # Distance to predicted class prototype
120                cls = pred.item()

```

```

119     mean = self.prototypes['means'][cls]
120     cov = self.prototypes['covariances'][cls]
121
122     diff = feat.cpu() - mean
123     # Mahalanobis distance: sqrt(diff^T * inv(cov) * diff)
124     try:
125         cov_inv = torch.inverse(cov.cpu())
126         dist = torch.sqrt(diff @ cov_inv @ diff.T).item()
127         class_distances[cls].append(dist)
128     except:
129         # Fallback if covariance is singular
130         dist = torch.norm(diff).item()
131         class_distances[cls].append(dist)
132
133     # Compute statistics per class
134     for cls in range(len(self.disease_classes)):
135         if len(class_distances[cls]) > 0:
136             distances = np.array(class_distances[cls])
137             self.ood_stats['class_means'][cls] = float(np.mean(distances))
138             self.ood_stats['class_stds'][cls] = float(np.std(distances))
139         else:
140             # Fallback if no samples predicted as this class
141             self.ood_stats['class_means'][cls] = 0.0
142             self.ood_stats['class_stds'][cls] = 1.0
143
144     # Save OOD statistics
145     self._save_ood_stats()
146
147     print(f"Dynamic OOD thresholds computed:")
148     for cls, name in enumerate(self.disease_classes):
149         mean = self.ood_stats['class_means'][cls]
150         std = self.ood_stats['class_stds'][cls]
151         threshold = mean + self.ood_stats['threshold_factor'] * std
152         print(f" {name}: mean={mean:.2f}, std={std:.2f}, threshold={threshold:.2f}")
153
154     def get_ood_threshold(self, class_idx: int) -> float:
155         """Get dynamic OOD threshold for specific class."""
156         mean = self.ood_stats['class_means'].get(class_idx, 0.0)
157         std = self.ood_stats['class_stds'].get(class_idx, 1.0)
158         return mean + self.ood_stats['threshold_factor'] * std
159
160     def _save_ood_stats(self, path=None):
161         """Save OOD statistics to disk."""
162         if path is None:
163             path = f"./ood_stats/{self.crop_name}_ood_stats.pt"
164         torch.save(self.ood_stats, path)
165
166     def load_ood_stats(self, path):
167         """Load OOD statistics from disk."""
168         self.ood_stats = torch.load(path)

```

Listing 4: Independent Crop Adapter Class with Dynamic OOD

5 Phase 1 Training Details

```

1 def _train_phase1(self, train_data, config):
2     """
3         DoRA training with LoRA+ optimizer.
4
5         LoRA+: Use 16x learning rate for B matrices
6         Target: 95% clean accuracy

```

```

7     """
8     # LoRA+ optimizer configuration
9     optimizer_params = [
10         {
11             'params': [p for n, p in self.adapter.named_parameters()
12                         if 'lora_B' in n],
13             'lr': config['base_lr'] * config['loraplus_lr_ratio'] # 16x
14         },
15         {
16             'params': [p for n, p in self.adapter.named_parameters()
17                         if 'lora_A' in n or 'lora_magnitude' in n],
18             'lr': config['base_lr']
19         },
20         {
21             'params': self.classifier.parameters(),
22             'lr': config['base_lr']
23         }
24     ]
25
26     optimizer = torch.optim.AdamW(optimizer_params, weight_decay=1e-4)
27     criterion = nn.CrossEntropyLoss()
28
29     best_acc = 0.0
30
31     for epoch in range(config['phase1_epochs']):
32         self.adapter.train()
33         self.classifier.train()
34
35         epoch_loss = 0
36         correct = 0
37         total = 0
38
39         for images, labels in train_data:
40             images = images.to(self.device)
41             labels = labels.to(self.device)
42
43             # Forward pass
44             features = self.adapter(images).last_hidden_state[:, 0]
45             logits = self.classifier(features)
46             loss = criterion(logits, labels)
47
48             # Backward
49             optimizer.zero_grad()
50             loss.backward()
51             optimizer.step()
52
53             # Metrics
54             epoch_loss += loss.item()
55             _, predicted = logits.max(1)
56             total += labels.size(0)
57             correct += predicted.eq(labels).sum().item()
58
59             # Evaluate
60             val_acc = self._evaluate(train_data.val_loader)
61             avg_loss = epoch_loss / len(train_data)
62
63             print(f"Epoch {epoch+1}/{config['phase1_epochs']}: "
64                  f"Loss={avg_loss:.4f}, Val Acc={val_acc:.2f}%")
65
66             # Save best
67             if val_acc > best_acc:
68                 best_acc = val_acc
69                 self._save_checkpoint('phase1_best.pth')

```

```

70     # Load best checkpoint
71     self._load_checkpoint('phase1_best.pth')
72     print(f"\nPhase 1 best accuracy: {best_acc:.2f}%")
73
74     assert best_acc >= 0.95, f"Phase 1 accuracy {best_acc:.2f}% < 95%!"

```

Listing 5: Phase 1 DoRA Training

6 Mahalanobis Prototype and Dynamic OOD Computation

```

1 def _compute_prototypes(self, train_data):
2     """
3         Compute class-conditional Mahalanobis prototypes.
4
5     Returns:
6         prototypes: Dict with 'means', 'covariances', and 'num_classes'
7     """
8     from sklearn.covariance import EmpiricalCovariance
9
10    self.adapter.eval()
11
12    # Collect features per class
13    class_features = {cls: [] for cls in range(len(self.disease_classes))}
14
15    with torch.no_grad():
16        for images, labels in train_data:
17            images = images.to(self.device)
18
19            features = self.adapter(images).last_hidden_state[:, 0]
20
21            for feat, label in zip(features.cpu().numpy(), labels.numpy()):
22                class_features[label].append(feat)
23
24    # Compute means and covariances
25    means = {}
26    covariances = {}
27
28    for cls in range(len(self.disease_classes)):
29        features = np.array(class_features[cls])
30
31        # Mean
32        means[cls] = torch.tensor(features.mean(axis=0)).to(self.device)
33
34        # Covariance with regularization for numerical stability
35        cov_estimator = EmpiricalCovariance().fit(features)
36        cov_matrix = torch.tensor(cov_estimator.covariance_).to(self.device)
37
38        # Add small regularization to prevent singularity
39        cov_matrix += torch.eye(cov_matrix.shape[0], device=self.device) * 1e-4
40        covariances[cls] = cov_matrix
41
42    return {
43        'means': means,
44        'covariances': covariances,
45        'num_classes': len(self.disease_classes)
46    }
47
48 def compute_mahalanobis_distance(self, features, class_idx=None):
49     """
50         Compute minimum Mahalanobis distance to class prototypes.
51         If class_idx specified, compute distance to that class only.

```

```

52
53     Returns:
54         distance: Mahalanobis distance
55         predicted_class: Index of nearest class (if class_idx is None)
56     """
57     min_distance = float('inf')
58     predicted_class = class_idx if class_idx is not None else 0
59
60     classes_to_check = [class_idx] if class_idx is not None else range(self.
61 prototypes['num_classes'])
62
63     for cls in classes_to_check:
64         mean = self.prototypes['means'][cls]
65         cov = self.prototypes['covariances'][cls]
66
67         diff = features - mean
68
69         try:
70             cov_inv = torch.inverse(cov)
71             distance = torch.sqrt(diff @ cov_inv @ diff.T).item()
72         except:
73             distance = torch.norm(diff).item()
74
75         if class_idx is None and distance < min_distance:
76             min_distance = distance
77             predicted_class = cls
78         elif class_idx is not None:
79             min_distance = distance
80
81     return min_distance, predicted_class
82
83 def detect_oop_dynamic(self, image):
84     """
85     OOD detection using dynamic per-class thresholds.
86
87     Returns:
88         result: Dict with 'is_oop', 'predicted_class', 'confidence',
89                 'mahalanobis_distance', 'threshold'
90     """
91     self.adapter.eval()
92     self.classifier.eval()
93
94     with torch.no_grad():
95         features = self.adapter(image).last_hidden_state[:, 0]
96         logits = self.classifier(features)
97
98         # Get prediction
99         probs = torch.softmax(logits, dim=1)
100        confidence, predicted_class = probs.max(1)
101        predicted_class = predicted_class.item()
102        confidence = confidence.item()
103
104        # Compute Mahalanobis distance to predicted class
105        maha_dist, _ = self.compute_mahalanobis_distance(
106            features[0], class_idx=predicted_class
107        )
108
109        # Get dynamic threshold for predicted class
110        threshold = self.get_oop_threshold(predicted_class)
111
112        # OOD decision
113        is_oop = maha_dist > threshold

```

```

114     return {
115         'is_ood': is_ood,
116         'predicted_class': predicted_class,
117         'disease_name': self.disease_classes[predicted_class],
118         'confidence': confidence,
119         'mahalanobis_distance': maha_dist,
120         'threshold': threshold,
121         'ood_score': maha_dist / threshold if threshold > 0 else maha_dist
122     }

```

Listing 6: Prototype and Dynamic OOD Computation

7 Summary

Part 1 covered:

- Environment setup for v5.5
- Simple crop router implementation (replaces SEMA)
- Independent crop adapter architecture with dynamic OOD
- Phase 1 DoRA training with LoRA+ optimizer
- Mahalanobis prototype computation
- Dynamic per-class OOD threshold computation from validation data

Part 2 will cover:

- Phase 2: SD-LoRA for class-incremental learning
- Phase 3: CONEC-LoRA for data-incremental learning
- Complete multi-crop pipeline integration with dynamic OOD
- Gradio demonstration interface