MetaBark Stable: Pseudocode Documentation

# Appendix for Research Paper

# Appendix for Research Paper

## Abstract

This appendix provides comprehensive pseudocode documentation for the MetaBark Stable framework, a controlled meta-learning approach for few-shot tree species classification from bark images. The pseudocode covers the complete system architecture including the neural network model, training algorithms, and evaluation procedures. This documentation serves as a technical reference for implementation and reproducibility of the research findings presented in the main paper.

# 1. Model Architecture Pseudocode

## 1.1 Main MetaBark Stable Model

ALGORITHM 1: MetaBark Stable Model Architecture  
INPUT: RGB bark images of size (3, 128, 128)  
OUTPUT: Classification logits for query samples  
  
CLASS StableMetaBark:  
 INITIALIZE(feature\_dim = 256):  
 // Initialize ResNet18 backbone with ImageNet weights  
 backbone ← ResNet18(pretrained=ImageNet)  
  
 // Freeze early layers for stability  
 FREEZE(backbone.conv1)  
 FREEZE(backbone.bn1)  
 FREEZE(backbone.layer1)  
 FREEZE(backbone.layer2)  
  
 // Remove final classification layer  
 backbone.fc ← Identity()  
  
 // Feature processing network  
 feature\_processor ← Sequential(  
 Linear(512, feature\_dim),  
 BatchNorm1d(feature\_dim),  
 ReLU(),  
 Dropout(0.6),  
 Linear(feature\_dim, feature\_dim),  
 BatchNorm1d(feature\_dim),  
 ReLU(),  
 Dropout(0.5)  
 )  
  
 // Multi-head self-attention mechanism  
 self\_attention ← MultiHeadAttention(  
 embed\_dim = feature\_dim,  
 num\_heads = 4,  
 dropout = 0.3,  
 batch\_first = True  
 )  
  
 // Feature enhancement network  
 enhancer ← Sequential(  
 Linear(feature\_dim, feature\_dim),  
 ReLU(),  
 Dropout(0.4),  
 Linear(feature\_dim, feature\_dim)  
 )  
  
 // Temperature parameter for confidence calibration  
 temperature ← Parameter(5.0)  
  
 FUNCTION extract\_features(x):  
 // Extract backbone features  
 features ← backbone(x)  
  
 // Process features  
 processed ← feature\_processor(features)  
  
 // Apply self-attention  
 batch\_size ← processed.size(0)  
 processed\_reshaped ← processed.unsqueeze(1)  
 attended, \_ ← self\_attention(processed\_reshaped, processed\_reshaped, processed\_reshaped)  
 attended ← attended.squeeze(1)  
  
 // Enhance features with residual connection  
 enhanced ← enhancer(attended)  
 final\_features ← processed + 0.5 \* enhanced  
  
 // L2 normalization  
 final\_features ← L2\_normalize(final\_features, dim=1)  
  
 RETURN final\_features  
  
 FUNCTION prototypical\_forward(support\_images, support\_labels, query\_images):  
 // Extract features for support and query sets  
 support\_features ← extract\_features(support\_images)  
 query\_features ← extract\_features(query\_images)  
  
 // Create class prototypes  
 unique\_labels ← unique(support\_labels)  
 prototypes ← []  
  
 FOR each label IN unique\_labels:  
 mask ← (support\_labels == label)  
 prototype ← mean(support\_features[mask], dim=0)  
 prototypes.append(prototype)  
  
 prototypes ← stack(prototypes)  
  
 // Calculate distances and apply temperature scaling  
 distances ← euclidean\_distance(query\_features, prototypes)  
 temp ← clamp(temperature, min=1.0, max=10.0)  
 logits ← -distances / temp  
  
 RETURN logits

## 1.2 Feature Extraction Pipeline

ALGORITHM 2: Feature Extraction Pipeline  
INPUT: Batch of bark images B = {x₁, x₂, ..., xₙ}  
OUTPUT: Normalized feature vectors F = {f₁, f₂, ..., fₙ}  
  
FUNCTION feature\_extraction\_pipeline(image\_batch):  
 // Stage 1: Backbone feature extraction  
 raw\_features ← ResNet18\_backbone(image\_batch) // Shape: (N, 512)  
  
 // Stage 2: Feature processing with regularization  
 processed\_features ← Linear\_layer1(raw\_features) // (N, 256)  
 processed\_features ← BatchNorm(processed\_features)  
 processed\_features ← ReLU(processed\_features)  
 processed\_features ← Dropout(processed\_features, p=0.6)  
  
 processed\_features ← Linear\_layer2(processed\_features) // (N, 256)  
 processed\_features ← BatchNorm(processed\_features)  
 processed\_features ← ReLU(processed\_features)  
 processed\_features ← Dropout(processed\_features, p=0.5)  
  
 // Stage 3: Multi-head self-attention  
 attention\_input ← processed\_features.unsqueeze(1) // (N, 1, 256)  
 attended\_features, attention\_weights ← MultiHeadAttention(  
 query=attention\_input,  
 key=attention\_input,  
 value=attention\_input,  
 num\_heads=4  
 )  
 attended\_features ← attended\_features.squeeze(1) // (N, 256)  
  
 // Stage 4: Feature enhancement with residual connection  
 enhanced\_features ← Linear\_enhancement1(attended\_features)  
 enhanced\_features ← ReLU(enhanced\_features)  
 enhanced\_features ← Dropout(enhanced\_features, p=0.4)  
 enhanced\_features ← Linear\_enhancement2(enhanced\_features)  
  
 // Residual connection with reduced weight  
 final\_features ← processed\_features + 0.5 \* enhanced\_features  
  
 // Stage 5: L2 normalization  
 normalized\_features ← L2\_normalize(final\_features, dim=1)  
  
 RETURN normalized\_features

# 2. Training Algorithm Pseudocode

## 2.1 Conservative Training Strategy

ALGORITHM 3: Conservative Training Strategy  
INPUT: Training dataset D, Validation dataset V  
OUTPUT: Trained model parameters θ\*  
  
CLASS ConservativeTrainer:  
 INITIALIZE(model, device):  
 // Ultra-conservative optimizer settings  
 optimizer ← Adam(  
 parameters = model.parameters(),  
 learning\_rate = 5e-6,  
 weight\_decay = 1e-4,  
 betas = (0.9, 0.999)  
 )  
  
 // Learning rate scheduler  
 scheduler ← ReduceLROnPlateau(  
 optimizer,  
 mode = 'min',  
 factor = 0.5,  
 patience = 5  
 )  
  
 // Early stopping parameters  
 best\_val\_loss ← infinity  
 patience\_counter ← 0  
 patience ← 10  
  
 // Training history tracking  
 history ← {  
 'train\_loss': [],  
 'val\_loss': [],  
 'val\_accuracy': [],  
 'val\_std': [],  
 'learning\_rate': [],  
 'temperature': []  
 }  
  
 FUNCTION train\_epoch(train\_loader):  
 model.train()  
 total\_loss ← 0  
 num\_batches ← 0  
  
 FOR each batch IN train\_loader:  
 support\_images, support\_labels, query\_images, query\_labels ← batch  
  
 // Forward pass  
 logits ← model.prototypical\_forward(support\_images, support\_labels, query\_images)  
 loss ← CrossEntropyLoss(logits, query\_labels)  
  
 // Backward pass with gradient clipping  
 optimizer.zero\_grad()  
 loss.backward()  
 clip\_grad\_norm(model.parameters(), max\_norm=1.0)  
 optimizer.step()  
  
 total\_loss ← total\_loss + loss.item()  
 num\_batches ← num\_batches + 1  
  
 // Memory cleanup every 10 batches  
 IF num\_batches % 10 == 0:  
 clear\_gpu\_cache()  
  
 RETURN total\_loss / num\_batches  
  
 FUNCTION validate(val\_loader):  
 model.eval()  
 total\_loss ← 0  
 accuracies ← []  
  
 WITH no\_gradient():  
 FOR each batch IN val\_loader:  
 support\_images, support\_labels, query\_images, query\_labels ← batch  
  
 // Forward pass  
 logits ← model.prototypical\_forward(support\_images, support\_labels, query\_images)  
 loss ← CrossEntropyLoss(logits, query\_labels)  
  
 // Calculate accuracy  
 predictions ← argmax(logits, dim=1)  
 accuracy ← mean((predictions == query\_labels).float())  
  
 total\_loss ← total\_loss + loss.item()  
 accuracies.append(accuracy)  
  
 val\_loss ← total\_loss / length(val\_loader)  
 val\_accuracy ← mean(accuracies)  
 val\_std ← std(accuracies)  
  
 RETURN val\_loss, val\_accuracy, val\_std  
  
 FUNCTION train(train\_loader, val\_loader, epochs=50):  
 PRINT("Starting conservative training...")  
  
 FOR epoch IN range(epochs):  
 // Training phase  
 train\_loss ← train\_epoch(train\_loader)  
  
 // Validation phase  
 val\_loss, val\_accuracy, val\_std ← validate(val\_loader)  
  
 // Update learning rate scheduler  
 scheduler.step(val\_loss)  
  
 // Record training history  
 history['train\_loss'].append(train\_loss)  
 history['val\_loss'].append(val\_loss)  
 history['val\_accuracy'].append(val\_accuracy \* 100)  
 history['val\_std'].append(val\_std \* 100)  
 history['learning\_rate'].append(optimizer.param\_groups[0]['lr'])  
 history['temperature'].append(model.temperature.item())  
  
 // Early stopping mechanism  
 IF val\_loss < best\_val\_loss:  
 best\_val\_loss ← val\_loss  
 patience\_counter ← 0  
 save\_model(model, "best\_model.pth")  
 ELSE:  
 patience\_counter ← patience\_counter + 1  
  
 // Print progress  
 PRINT(f"Epoch {epoch+1}: Train Loss: {train\_loss:.4f}, "  
 f"Val Loss: {val\_loss:.4f}, "  
 f"Val Acc: {val\_accuracy\*100:.1f}% ± {val\_std\*100:.1f}%")  
  
 // Early stopping condition  
 IF patience\_counter >= patience:  
 PRINT(f"Early stopping at epoch {epoch+1}")  
 BREAK  
  
 // Memory cleanup  
 clear\_gpu\_cache()  
 garbage\_collect()  
  
 // Load best model  
 load\_model(model, "best\_model.pth")  
 PRINT("Training completed! Best model loaded.")  
  
 RETURN history

## 2.2 Episode Generation for Meta-Learning

ALGORITHM 4: Episode Generation for Meta-Learning  
INPUT: Dataset D, n\_way, n\_shot, n\_query, n\_episodes  
OUTPUT: Generated episodes for training/validation  
  
CLASS EpisodeDataLoader:  
 INITIALIZE(dataset, n\_way=5, n\_shot=5, n\_query=3, n\_episodes=100):  
 // Group images by species  
 species\_to\_images ← defaultdict(list)  
 FOR each (image, label) IN dataset:  
 species\_to\_images[label].append(image)  
  
 available\_species ← keys(species\_to\_images)  
  
 FUNCTION generate\_episode():  
 // Sample species for this episode  
 episode\_species ← random\_sample(available\_species, n\_way)  
  
 support\_images ← []  
 support\_labels ← []  
 query\_images ← []  
 query\_labels ← []  
  
 FOR class\_idx, species IN enumerate(episode\_species):  
 species\_images ← species\_to\_images[species]  
  
 // Ensure sufficient images available  
 IF length(species\_images) >= n\_shot + n\_query:  
 sampled\_images ← random\_sample(species\_images, n\_shot + n\_query)  
  
 // Create support set  
 FOR image IN sampled\_images[:n\_shot]:  
 support\_images.append(image)  
 support\_labels.append(class\_idx)  
  
 // Create query set  
 FOR image IN sampled\_images[n\_shot:n\_shot + n\_query]:  
 query\_images.append(image)  
 query\_labels.append(class\_idx)  
  
 // Convert to tensors  
 support\_images ← stack(support\_images)  
 support\_labels ← tensor(support\_labels)  
 query\_images ← stack(query\_images)  
 query\_labels ← tensor(query\_labels)  
  
 RETURN support\_images, support\_labels, query\_images, query\_labels  
  
 FUNCTION \_\_iter\_\_():  
 FOR episode IN range(n\_episodes):  
 YIELD generate\_episode()

# 3. Data Processing Pseudocode

## 3.1 Conservative Data Augmentation

ALGORITHM 5: Conservative Data Augmentation Strategy  
INPUT: Raw bark images  
OUTPUT: Augmented and normalized images  
  
FUNCTION create\_conservative\_transforms():  
 // Training transforms with minimal augmentation  
 train\_transform ← Compose([  
 Resize((128, 128)),  
 RandomHorizontalFlip(probability=0.3),  
 RandomRotation(degrees=10),  
 ColorJitter(  
 brightness=0.1,  
 contrast=0.1,  
 saturation=0.1,  
 hue=0.05  
 ),  
 ToTensor(),  
 Normalize(  
 mean=[0.485, 0.456, 0.406],  
 std=[0.229, 0.224, 0.225]  
 )  
 ])  
  
 // Validation transforms (no augmentation)  
 val\_transform ← Compose([  
 Resize((128, 128)),  
 ToTensor(),  
 Normalize(  
 mean=[0.485, 0.456, 0.406],  
 std=[0.229, 0.224, 0.225]  
 )  
 ])  
  
 RETURN train\_transform, val\_transform  
  
CLASS BarkDataset:  
 INITIALIZE(data\_dir, transform=None, is\_training=True):  
 images ← []  
 labels ← []  
 species\_names ← []  
  
 // Load images from species folders  
 species\_folders ← sorted(list\_directories(data\_dir))  
  
 FOR species\_idx, species\_folder IN enumerate(species\_folders):  
 species\_path ← join(data\_dir, species\_folder)  
 species\_names.append(species\_folder)  
  
 FOR img\_file IN list\_files(species\_path):  
 IF img\_file.endswith(('.jpg', '.jpeg', '.png')):  
 img\_path ← join(species\_path, img\_file)  
 images.append(img\_path)  
 labels.append(species\_idx)  
  
 FUNCTION \_\_getitem\_\_(idx):  
 img\_path ← images[idx]  
 label ← labels[idx]  
  
 // Load and convert image  
 image ← load\_image(img\_path).convert\_to\_RGB()  
  
 // Apply transforms if available  
 IF transform is not None:  
 image ← transform(image)  
  
 RETURN image, label

# 4. Evaluation Pseudocode

## 4.1 Episode-based Evaluation

ALGORITHM 6: Episode-based Model Evaluation  
INPUT: Trained model, Test dataset, Evaluation configuration  
OUTPUT: Performance metrics and statistics  
  
FUNCTION minimal\_real\_evaluation(model, test\_dataset, config):  
 // Set model to evaluation mode  
 model.eval()  
  
 // Initialize evaluation parameters  
 n\_episodes ← config.n\_episodes  
 n\_way ← config.n\_way  
 n\_shot ← config.n\_shot  
 n\_query ← config.n\_query  
  
 accuracies ← []  
  
 PRINT(f"Running {n\_episodes} episodes ({n\_way}-way {n\_shot}-shot)...")  
  
 WITH no\_gradient():  
 FOR episode IN range(n\_episodes):  
 TRY:  
 // Generate evaluation episode  
 support\_images, support\_labels, query\_images, query\_labels ←  
 generate\_evaluation\_episode(test\_dataset, n\_way, n\_shot, n\_query)  
  
 // Forward pass through model  
 logits ← model.prototypical\_forward(  
 support\_images, support\_labels, query\_images  
 )  
  
 // Calculate episode accuracy  
 predictions ← argmax(logits, dim=1)  
 accuracy ← mean((predictions == query\_labels).float())  
  
 accuracies.append(accuracy \* 100)  
 PRINT(f"Episode {episode+1}: {accuracy\*100:.1f}%")  
  
 EXCEPT Exception as e:  
 PRINT(f"Episode {episode+1} failed: {e}")  
 CONTINUE  
  
 // Calculate final statistics  
 IF length(accuracies) > 0:  
 mean\_accuracy ← mean(accuracies)  
 std\_accuracy ← std(accuracies)  
  
 PRINT(f"Mean Accuracy: {mean\_accuracy:.1f}% ± {std\_accuracy:.1f}%")  
 PRINT(f"Episodes Completed: {length(accuracies)}/{n\_episodes}")  
  
 // Save results  
 results ← {  
 'mean\_accuracy': mean\_accuracy,  
 'std\_accuracy': std\_accuracy,  
 'episodes\_completed': length(accuracies),  
 'individual\_accuracies': accuracies  
 }  
  
 save\_results(results, "evaluation\_results.json")  
  
 RETURN results  
 ELSE:  
 PRINT("No successful episodes!")  
 RETURN None  
  
FUNCTION generate\_evaluation\_episode(dataset, n\_way, n\_shot, n\_query):  
 // Select random species for episode  
 available\_species ← get\_available\_species(dataset)  
 episode\_species ← random\_sample(available\_species, n\_way)  
  
 support\_images ← []  
 support\_labels ← []  
 query\_images ← []  
 query\_labels ← []  
  
 FOR class\_idx, species IN enumerate(episode\_species):  
 // Get images for current species  
 species\_images ← get\_species\_images(dataset, species)  
  
 // Sample images for support and query sets  
 IF length(species\_images) >= n\_shot + n\_query:  
 sampled\_images ← random\_sample(species\_images, n\_shot + n\_query)  
  
 // Support set  
 FOR image IN sampled\_images[:n\_shot]:  
 support\_images.append(image)  
 support\_labels.append(class\_idx)  
  
 // Query set  
 FOR image IN sampled\_images[n\_shot:n\_shot + n\_query]:  
 query\_images.append(image)  
 query\_labels.append(class\_idx)  
  
 // Convert to tensors  
 support\_images ← stack(support\_images)  
 support\_labels ← tensor(support\_labels)  
 query\_images ← stack(query\_images)  
 query\_labels ← tensor(query\_labels)  
  
 RETURN support\_images, support\_labels, query\_images, query\_labels

## 4.2 Performance Metrics Calculation

ALGORITHM 7: Performance Metrics Calculation  
INPUT: Predictions P, Ground truth labels Y, Episode results E  
OUTPUT: Comprehensive performance metrics  
  
FUNCTION calculate\_performance\_metrics(predictions, ground\_truth, episode\_results):  
 // Basic accuracy metrics  
 accuracy ← mean((predictions == ground\_truth).float()) \* 100  
  
 // Episode-wise statistics  
 episode\_accuracies ← [result.accuracy for result in episode\_results]  
 mean\_episode\_accuracy ← mean(episode\_accuracies)  
 std\_episode\_accuracy ← std(episode\_accuracies)  
  
 // Confidence interval (95%)  
 confidence\_interval ← calculate\_confidence\_interval(episode\_accuracies, 0.95)  
  
 // Coefficient of variation  
 coefficient\_of\_variation ← (std\_episode\_accuracy / mean\_episode\_accuracy) \* 100  
  
 // Classification metrics per class  
 precision\_scores ← []  
 recall\_scores ← []  
 f1\_scores ← []  
  
 unique\_classes ← unique(ground\_truth)  
 FOR class\_label IN unique\_classes:  
 // True positives, false positives, false negatives  
 tp ← sum((predictions == class\_label) AND (ground\_truth == class\_label))  
 fp ← sum((predictions == class\_label) AND (ground\_truth != class\_label))  
 fn ← sum((predictions != class\_label) AND (ground\_truth == class\_label))  
  
 // Calculate metrics  
 precision ← tp / (tp + fp) if (tp + fp) > 0 else 0  
 recall ← tp / (tp + fn) if (tp + fn) > 0 else 0  
 f1 ← 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0  
  
 precision\_scores.append(precision)  
 recall\_scores.append(recall)  
 f1\_scores.append(f1)  
  
 // Macro-averaged metrics  
 macro\_precision ← mean(precision\_scores) \* 100  
 macro\_recall ← mean(recall\_scores) \* 100  
 macro\_f1 ← mean(f1\_scores) \* 100  
  
 // Compile results  
 metrics ← {  
 'accuracy': accuracy,  
 'precision': macro\_precision,  
 'recall': macro\_recall,  
 'f1\_score': macro\_f1,  
 'mean\_episode\_accuracy': mean\_episode\_accuracy,  
 'std\_episode\_accuracy': std\_episode\_accuracy,  
 'confidence\_interval': confidence\_interval,  
 'coefficient\_of\_variation': coefficient\_of\_variation,  
 'min\_accuracy': min(episode\_accuracies),  
 'max\_accuracy': max(episode\_accuracies)  
 }  
  
 RETURN metrics

# 5. Utility Functions Pseudocode

## 5.1 Reproducibility and Seed Setting

ALGORITHM 8: Reproducibility Setup  
INPUT: Random seed value  
OUTPUT: Configured environment for reproducible results  
  
FUNCTION set\_seeds(seed=42):  
 // Set PyTorch seeds  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
 torch.cuda.manual\_seed\_all(seed)  
  
 // Set NumPy seed  
 numpy.random.seed(seed)  
  
 // Set Python random seed  
 python\_random.seed(seed)  
  
 // Configure CUDNN for deterministic behavior  
 torch.backends.cudnn.deterministic ← True  
 torch.backends.cudnn.benchmark ← False  
  
 PRINT(f"All seeds set to {seed} for reproducibility")

## 5.2 Memory Management

ALGORITHM 9: Memory Management Utilities  
INPUT: Current system state  
OUTPUT: Optimized memory usage  
  
FUNCTION memory\_cleanup():  
 // Clear GPU cache  
 IF cuda\_is\_available():  
 torch.cuda.empty\_cache()  
  
 // Force garbage collection  
 gc.collect()  
  
 PRINT("Memory cleanup completed")  
  
FUNCTION monitor\_memory\_usage():  
 // Get current memory usage  
 IF cuda\_is\_available():  
 allocated ← torch.cuda.memory\_allocated() / (1024\*\*3) // GB  
 cached ← torch.cuda.memory\_reserved() / (1024\*\*3) // GB  
  
 PRINT(f"GPU Memory - Allocated: {allocated:.2f}GB, Cached: {cached:.2f}GB")  
  
 // System memory  
 import psutil  
 system\_memory ← psutil.virtual\_memory()  
 PRINT(f"System Memory Usage: {system\_memory.percent:.1f}%")

## 5.3 Model Checkpointing

ALGORITHM 10: Model Checkpointing System  
INPUT: Model state, Training metrics, Checkpoint path  
OUTPUT: Saved model checkpoint  
  
FUNCTION save\_checkpoint(model, optimizer, epoch, metrics, filepath):  
 checkpoint ← {  
 'epoch': epoch,  
 'model\_state\_dict': model.state\_dict(),  
 'optimizer\_state\_dict': optimizer.state\_dict(),  
 'metrics': metrics,  
 'timestamp': current\_timestamp()  
 }  
  
 torch.save(checkpoint, filepath)  
 PRINT(f"Checkpoint saved: {filepath}")  
  
FUNCTION load\_checkpoint(model, optimizer, filepath):  
 IF file\_exists(filepath):  
 checkpoint ← torch.load(filepath)  
  
 model.load\_state\_dict(checkpoint['model\_state\_dict'])  
 optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])  
  
 epoch ← checkpoint['epoch']  
 metrics ← checkpoint['metrics']  
  
 PRINT(f"Checkpoint loaded: {filepath}")  
 RETURN epoch, metrics  
 ELSE:  
 PRINT(f"Checkpoint not found: {filepath}")  
 RETURN 0, None

# 6. Configuration Management

## 6.1 Hyperparameter Configuration

ALGORITHM 11: Hyperparameter Configuration  
INPUT: Configuration requirements  
OUTPUT: Optimized hyperparameter settings  
  
CLASS MetaBarkConfig:  
 INITIALIZE():  
 // Model architecture parameters  
 feature\_dim ← 256  
 num\_attention\_heads ← 4  
 dropout\_rate ← 0.6  
 temperature\_init ← 5.0  
  
 // Training parameters  
 learning\_rate ← 5e-6  
 weight\_decay ← 1e-4  
 batch\_size ← 16  
 max\_epochs ← 50  
  
 // Meta-learning parameters  
 n\_way ← 2  
 n\_shot ← 3  
 n\_query ← 5  
 n\_episodes\_train ← 100  
 n\_episodes\_val ← 20  
  
 // Regularization parameters  
 gradient\_clip\_norm ← 1.0  
 early\_stopping\_patience ← 10  
 lr\_scheduler\_patience ← 5  
 lr\_scheduler\_factor ← 0.5  
  
 // Data parameters  
 image\_size ← (128, 128)  
 augmentation\_probability ← 0.3  
 rotation\_degrees ← 10  
  
 // Evaluation parameters  
 evaluation\_episodes ← 5  
 confidence\_level ← 0.95  
  
 FUNCTION validate\_config():  
 // Validate parameter ranges  
 ASSERT feature\_dim > 0, "Feature dimension must be positive"  
 ASSERT 0 < learning\_rate < 1, "Learning rate must be in (0, 1)"  
 ASSERT 0 <= dropout\_rate < 1, "Dropout rate must be in [0, 1)"  
 ASSERT n\_way >= 2, "n\_way must be at least 2"  
 ASSERT n\_shot >= 1, "n\_shot must be at least 1"  
  
 PRINT("Configuration validation passed")  
  
 FUNCTION save\_config(filepath):  
 config\_dict ← convert\_to\_dict(self)  
 save\_json(config\_dict, filepath)  
 PRINT(f"Configuration saved: {filepath}")  
  
 FUNCTION load\_config(filepath):  
 config\_dict ← load\_json(filepath)  
 update\_attributes(self, config\_dict)  
 PRINT(f"Configuration loaded: {filepath}")

# Conclusion

This pseudocode documentation provides a comprehensive technical reference for implementing the MetaBark Stable framework. The algorithms cover all major components including the neural network architecture, training procedures, data processing, and evaluation methods. The pseudocode follows standard algorithmic notation and includes detailed parameter specifications that align with the experimental setup described in the main research paper.

The conservative design principles emphasized throughout the pseudocode ensure stable training dynamics and reproducible results, which are critical for few-shot learning applications in ecological monitoring and forest management systems.