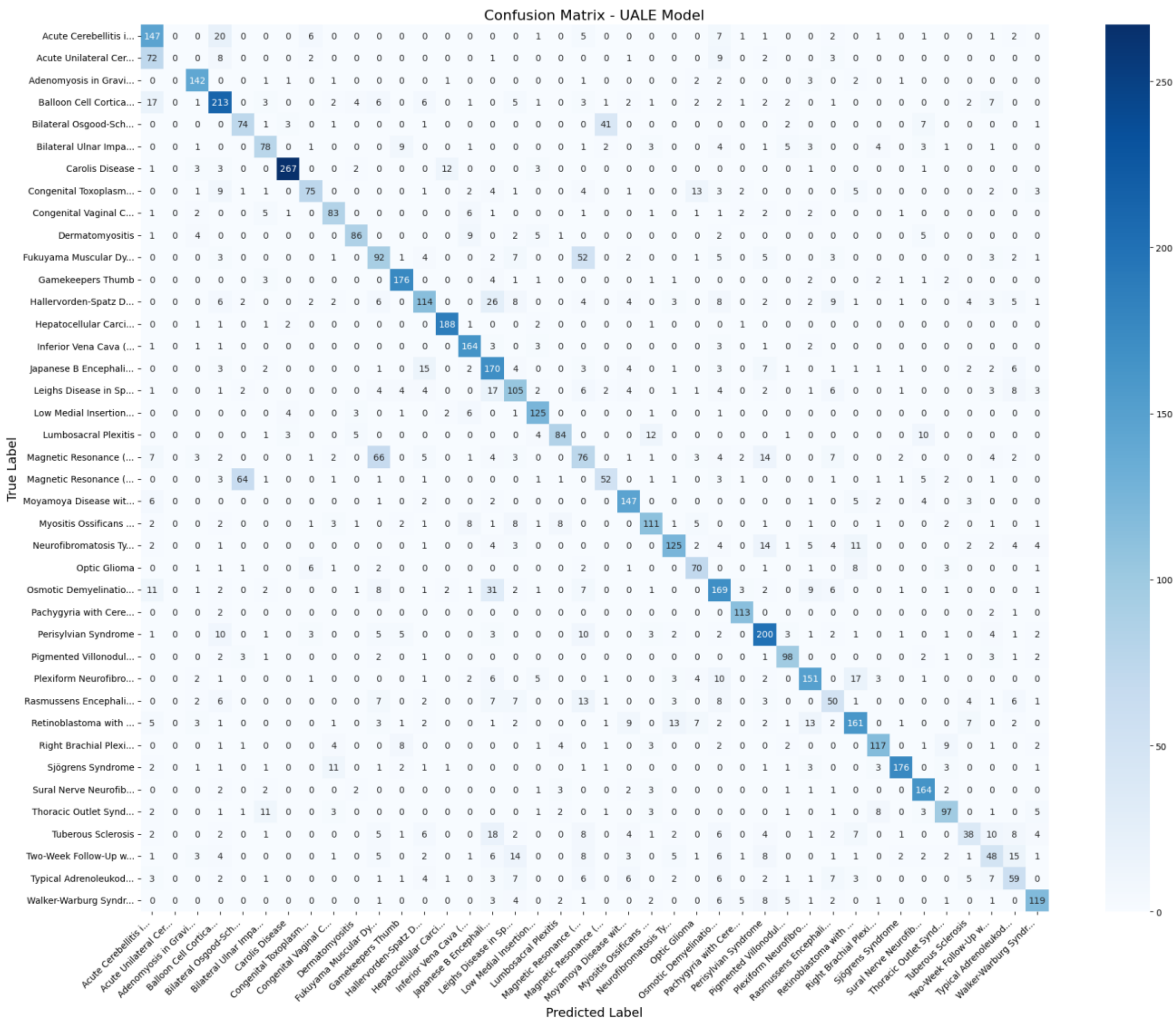
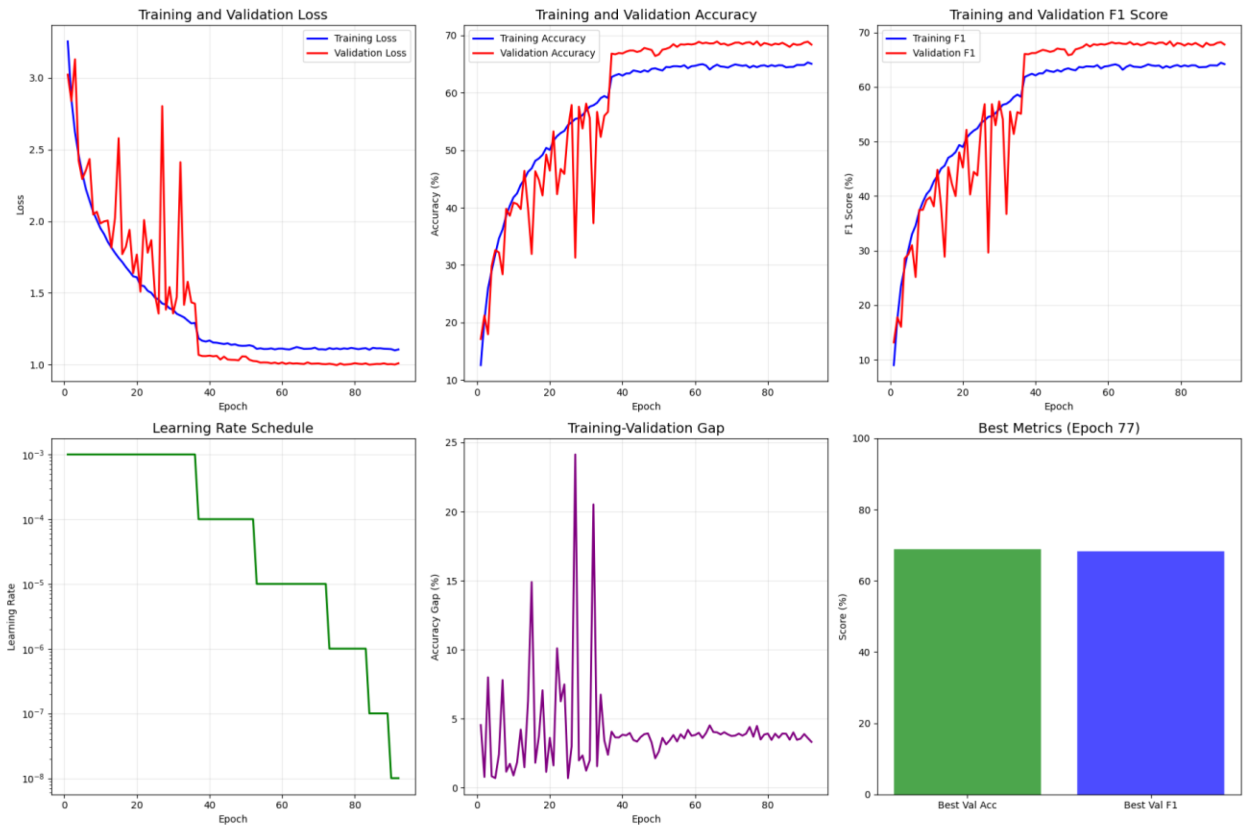
OUTPUTS:




Training and Validation Loss
Training Loss
Validation Loss
Loss
Epoch
Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy (%)
Epoch
Training and Validation F1 Score
Training F1
Validation F1
F1 Score (%)
Epoch
Learning Rate Schedule
Learning Rate
Epoch
Training-Validation Gap
Accuracy Gap (%)
Epoch
Best Metrics (Epoch 77)
Score (%)
Best Val Acc
Best Val F1

Performing comprehensive evaluation...
Performing comprehensive evaluation...

==
=
COMPREHENSIVE MODEL EVALUATION RESULTS
==
=

📊 CLASSIFICATION METRICS:
  Accuracy: 0.6907 (69.07%)
  Precision: 0.6835 (68.35%)
  Recall: 0.6907 (69.07%)
  F1-Score: 0.6831 (68.31%)

🔧 MODEL EFFICIENCY:
  Total Parameters: 53,632 (0.05M)
  Model Size: 0.20 MB
  GFLOPs: 0.180

⚡ INFERENCE PERFORMANCE:
  Average Inference Time: 9.61 ms
  Images per Second: 3326.1

🎯 PREDICTION QUALITY:
  Average Confidence: 0.6224
  Average Uncertainty: 216.8228

📈 DATASET INFO:
  Total Classes: 40
  Test Samples: 6839

CODE:

```
!pip install thop

import os
import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, f1_score, precision_score,
recall_score
from PIL import Image
import time
import warnings
warnings.filterwarnings('ignore')

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from thop import profile

# Set random seeds for reproducibility
random.seed(42)
```

```
np.random.seed(42)
torch.manual_seed(42)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# ==================== FIXED DATASET LOADING ====================

def load_benchmark_dataset_fixed():
    """Load dataset with fixed path handling and size standardization"""
    base_path = "/kaggle/input/benchmark/Benchmark Diagnostic MRI and Medical Imaging Dataset/Medical Imaging Dataset"

    # Class mapping with corrected subdirectory names
    class_mapping = {
        "Low Medial Insertion of Common Bile Duct with Pancreas Divisum-20240916T165825Z-001": "Low Medial Insertion of Common Bile Duct with Pancreas Divisum",
        "Inferior Vena Cava (IVC) Leiomyosarcoma-20240916T165709Z-001": "Inferior Vena Cava (IVC) Leiomyosarcoma",
        "Acute Cerebellitis in HIV": "Acute Cerebellitis in HIV",
        "Acute Unilateral Cerebellitis in HIV": "Acute Unilateral Cerebellitis in HIV",
        "Adenomyosis in Gravid Uterus": "Adenomyosis in Gravid Uterus",
        "Balloon Cell Cortical Dysplasia": "Balloon Cell Cortical Dysplasia",
        "Bilateral Osgood-Schlatter Disease with Chronic Inflammatory Arthritis": "Bilateral Osgood-Schlatter Disease with Chronic Inflammatory Arthritis",
        "Bilateral Ulnar Impaction Syndrome": "Bilateral Ulnar Impaction Syndrome",
        "Carolis Disease": "Carolis Disease",
        "Congenital Toxoplasmosis": "Congenital Toxoplasmosis",
        "Congenital Vaginal Cyst": "Congenital Vaginal Cyst",
        "Dermatomyositis": "Dermatomyositis",
        "Fukuyama Muscular Dystrophy": "Fukuyama Muscular Dystrophy",
        "Gamekeepers Thumb": "Gamekeepers Thumb",
```

"Hallervorden-Spatz Disease (now called Pantothenate Kinase-Associated Neurodegeneration)": "Hallervorden-Spatz Disease (now called Pantothenate Kinase-Associated Neurodegeneration)",

"Hepatocellular Carcinoma (HCC) and Dysplastic Nodules with Cirrhosis": "Hepatocellular Carcinoma (HCC) and Dysplastic Nodules with Cirrhosis",

"Japanese B Encephalitis or Epstein-Barr Encephalitis": "Japanese B Encephalitis or Epstein-Barr Encephalitis",

"Leighs Disease in Spinal Cord and Inferior Colliculi": "Leighs Disease in Spinal Cord and Inferior Colliculi",

"Lumbosacral Plexitis": "Lumbosacral Plexitis",

"Magnetic Resonance (MR) Brain": "Magnetic Resonance (MR) Brain",

"Magnetic Resonance (MR) Spine": "Magnetic Resonance (MR) Spine",

"Moyamoya Disease with Intraventricular Hemorrhage": "Moyamoya Disease with Intraventricular Hemorrhage",

"Myositis Ossificans Progressiva": "Myositis Ossificans Progressiva",

"Neurofibromatosis Type 1 (NF1) with Optic Glioma and Intracranial Extension": "Neurofibromatosis Type 1 (NF1) with Optic Glioma and Intracranial Extension",

"Optic Glioma": "Optic Glioma",

"Osmotic Demyelination Syndrome": "Osmotic Demyelination Syndrome",

"Pachygyria with Cerebellar Hypoplasia": "Pachygyria with Cerebellar Hypoplasia",

"Perisylvian Syndrome": "Perisylvian Syndrome",

"Pigmented Villonodular Synovitis (PVNS) of Ankle": "Pigmented Villonodular Synovitis (PVNS) of Ankle",

"Plexiform Neurofibroma with Sphenoid Wing Absence": "Plexiform Neurofibroma with Sphenoid Wing Absence",

"Rasmussens Encephalitis": "Rasmussens Encephalitis",

"Retinoblastoma with Intracranial Spread Along Cranial Nerve": "Retinoblastoma with Intracranial Spread Along Cranial Nerve",

"Right Brachial Plexitis": "Right Brachial Plexitis",

"Sjögrens Syndrome": "Sjögrens Syndrome",

"Sural Nerve Neurofibroma": "Sural Nerve Neurofibroma",

"Thoracic Outlet Syndrome": "Thoracic Outlet Syndrome",

"Tuberous Sclerosis": "Tuberous Sclerosis",

"Two-Week Follow-Up with Spectroscopy": "Two-Week Follow-Up with Spectroscopy",

"Typical Adrenoleukodystrophy": "Typical Adrenoleukodystrophy",

"Walker-Warburg Syndrome": "Walker-Warburg Syndrome"

}

```
    image_paths = []
    labels = []

    for class_name, subdir_name in class_mapping.items():
        class_dir = os.path.join(base_path, class_name)
        if os.path.exists(class_dir):
            subdir = os.path.join(class_dir, subdir_name)
            if os.path.exists(subdir):
                for file in os.listdir(subdir):
                    if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tiff', '.dcm', '.nii')):
                        image_paths.append(os.path.join(subdir, file))
                        labels.append(class_name)

    print(f"Total images found: {len(image_paths)}")

    # Create label mapping
    unique_labels = sorted(list(set(labels)))
    label_to_idx = {label: idx for idx, label in enumerate(unique_labels)}
    encoded_labels = [label_to_idx[label] for label in labels]

    return image_paths, encoded_labels, unique_labels, label_to_idx

class BenchmarkMRIDataset(Dataset):
    def __init__(self, file_paths, labels, transform=None):
        self.file_paths = file_paths
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.file_paths)
```

```
    def __getitem__(self, idx):
        img_path = self.file_paths[idx]
        label = self.labels[idx]
        try:
            img = Image.open(img_path).convert('L')
            if self.transform:
                img = self.transform(img)
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            img = torch.zeros(1, 224, 224)
        return img, label

# =================== ULTRA-LIGHTWEIGHT MODEL ARCHITECTURE ===================

class MedicalMicroNet(nn.Module):
    """Ultra-lightweight network for medical imaging"""
    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels, 8, 3, padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(8, 16, 3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(16, 32, 3, padding=1),
```

```
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.AdaptiveAvgPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Linear(32, 64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(64, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

class TextureNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.net = MedicalMicroNet(1, num_classes)

    def forward(self, x):
        return self.net(x)

class ShapeNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.net = MedicalMicroNet(2, num_classes)
```

```
    def forward(self, x):
        # Edge enhancement
        sobel_x = torch.tensor([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=torch.float32).view(1, 1, 3, 3).to(x.device)
        sobel_y = torch.tensor([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=torch.float32).view(1, 1, 3, 3).to(x.device)
        edges_x = F.conv2d(x, sobel_x, padding=1)
        edges_y = F.conv2d(x, sobel_y, padding=1)
        edges = torch.sqrt(edges_x**2 + edges_y**2)

        x_combined = torch.cat([x, edges], dim=1)
        return self.net(x_combined)

class IntensityNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.net = MedicalMicroNet(1, num_classes)

    def forward(self, x):
        x = (x - x.mean(dim=(2, 3), keepdim=True)) / (x.std(dim=(2, 3), keepdim=True) + 1e-8)
        return self.net(x)

class SpatialNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.net = MedicalMicroNet(1, num_classes)

    def forward(self, x):
        return self.net(x)

class MultiScaleNet(nn.Module):
    def __init__(self, num_classes):
```

```python
        super().__init__()
        self.net = MedicalMicroNet(1, num_classes)

    def forward(self, x):
        return self.net(x)

class UltraLightUALE(nn.Module):
    """Ultra-lightweight ensemble model"""
    def __init__(self, num_classes):
        super().__init__()
        self.texture_net = TextureNet(num_classes)
        self.shape_net = ShapeNet(num_classes)
        self.intensity_net = IntensityNet(num_classes)
        self.spatial_net = SpatialNet(num_classes)
        self.multiscale_net = MultiScaleNet(num_classes)
        self.num_classes = num_classes

    def forward(self, x):
        pred1 = self.texture_net(x)
        pred2 = self.shape_net(x)
        pred3 = self.intensity_net(x)
        pred4 = self.spatial_net(x)
        pred5 = self.multiscale_net(x)

        preds = torch.stack([pred1, pred2, pred3, pred4, pred5], dim=0)
        ensemble_pred = torch.mean(preds, dim=0)
        uncertainty = torch.var(preds, dim=0).mean(dim=1)

        return ensemble_pred, uncertainty, preds

# ==================== DATA TRANSFORMS ====================
```

```
def get_transforms():
    """Get transforms with fixed output size"""
    train_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5])
    ])

    val_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5])
    ])

    return train_transform, val_transform

# ==================== ENHANCED TRAINING WITH METRICS ====================

def train_lightweight_with_metrics(model, train_loader, val_loader, epochs=100, patience=15):
    """Enhanced training with comprehensive metrics tracking"""
    model.to(device)
    optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=5, verbose=True)
    criterion = nn.CrossEntropyLoss()

    # Metrics tracking
```

```
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
train_f1_scores = []
val_f1_scores = []
learning_rates = []

best_val_acc = 0
no_improve = 0

print("Starting training for 100 epochs...")

for epoch in range(epochs):
    # Training phase
    model.train()
    train_loss = 0
    train_preds = []
    train_targets = []

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs, _, _ = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
```

```
    train_preds.extend(predicted.cpu().numpy())
    train_targets.extend(labels.cpu().numpy())

# Calculate training metrics
train_acc = accuracy_score(train_targets, train_preds) * 100
train_f1 = f1_score(train_targets, train_preds, average='weighted') * 100
avg_train_loss = train_loss / len(train_loader)

# Validation phase
model.eval()
val_loss = 0
val_preds = []
val_targets = []

with torch.no_grad():
    for images, labels in val_loader:
        images, labels = images.to(device), labels.to(device)
        outputs, _, _ = model(images)
        loss = criterion(outputs, labels)

        val_loss += loss.item()
        _, predicted = outputs.max(1)
        val_preds.extend(predicted.cpu().numpy())
        val_targets.extend(labels.cpu().numpy())

# Calculate validation metrics
val_acc = accuracy_score(val_targets, val_preds) * 100
val_f1 = f1_score(val_targets, val_preds, average='weighted') * 100
avg_val_loss = val_loss / len(val_loader)

# Store metrics
```

```
train_losses.append(avg_train_loss)
val_losses.append(avg_val_loss)
train_accuracies.append(train_acc)
val_accuracies.append(val_acc)
train_f1_scores.append(train_f1)
val_f1_scores.append(val_f1)
learning_rates.append(optimizer.param_groups[0]['lr'])

# Learning rate scheduling
scheduler.step(val_acc)

print(f"Epoch {epoch+1}/{epochs}")
print(f"Train - Loss: {avg_train_loss:.4f} | Acc: {train_acc:.2f}% | F1: {train_f1:.2f}%")
print(f"Val   - Loss: {avg_val_loss:.4f} | Acc: {val_acc:.2f}% | F1: {val_f1:.2f}%")
print(f"LR: {optimizer.param_groups[0]['lr']:.6f}")

# Early stopping
if val_acc > best_val_acc:
    best_val_acc = val_acc
    no_improve = 0
    torch.save(model.state_dict(), 'best_uale_model.pth')
    print(f"*** New best validation accuracy: {best_val_acc:.2f}% ***")
else:
    no_improve += 1

if no_improve >= patience:
    print(f"Early stopping at epoch {epoch+1}")
    break

print("-" * 60)
```

```
    # Load best model
    model.load_state_dict(torch.load('best_uale_model.pth'))

    return model, {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'train_accuracies': train_accuracies,
        'val_accuracies': val_accuracies,
        'train_f1_scores': train_f1_scores,
        'val_f1_scores': val_f1_scores,
        'learning_rates': learning_rates,
        'best_val_acc': best_val_acc
    }

# ==================== COMPREHENSIVE EVALUATION ====================

def comprehensive_evaluation(model, test_loader, class_names):
    """Comprehensive model evaluation with all metrics"""
    print("Performing comprehensive evaluation...")

    model.eval()
    all_preds = []
    all_targets = []
    all_confidences = []
    all_uncertainties = []
    inference_times = []

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
```

```
        # Measure inference time
        start_time = time.time()
        outputs, uncertainties, _ = model(images)
        end_time = time.time()

        inference_times.append(end_time - start_time)

        # Get predictions and confidences
        probabilities = F.softmax(outputs, dim=1)
        confidences, predicted = probabilities.max(1)

        all_preds.extend(predicted.cpu().numpy())
        all_targets.extend(labels.cpu().numpy())
        all_confidences.extend(confidences.cpu().numpy())
        all_uncertainties.extend(uncertainties.cpu().numpy())

    # Calculate comprehensive metrics
    accuracy = accuracy_score(all_targets, all_preds)
    precision = precision_score(all_targets, all_preds, average='weighted', zero_division=0)
    recall = recall_score(all_targets, all_preds, average='weighted', zero_division=0)
    f1 = f1_score(all_targets, all_preds, average='weighted', zero_division=0)

    # Performance metrics
    total_inference_time = sum(inference_times)
    avg_inference_time = np.mean(inference_times)
    images_per_sec = len(all_targets) / total_inference_time

    # Model complexity
    total_params = sum(p.numel() for p in model.parameters())
    model_size_mb = total_params * 4 / (1024 * 1024)  # Assuming float32
```

```python
    # Calculate GFLOPs
    dummy_input = torch.randn(1, 1, 224, 224).to(device)
    flops, params = profile(model, inputs=(dummy_input,), verbose=False)
    gflops = flops / 1e9

    metrics = {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'avg_confidence': np.mean(all_confidences),
        'avg_uncertainty': np.mean(all_uncertainties),
        'total_params': total_params,
        'model_size_mb': model_size_mb,
        'gflops': gflops,
        'avg_inference_time': avg_inference_time,
        'images_per_sec': images_per_sec,
        'predictions': all_preds,
        'targets': all_targets,
        'confidences': all_confidences,
        'uncertainties': all_uncertainties
    }

    return metrics

# =================== VISUALIZATION FUNCTIONS ===================

def plot_training_history(training_metrics):
    """Plot comprehensive training history"""
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    epochs = range(1, len(training_metrics['train_losses']) + 1)
```

```python
# Loss curves
axes[0, 0].plot(epochs, training_metrics['train_losses'], 'b-', label='Training Loss', linewidth=2)
axes[0, 0].plot(epochs, training_metrics['val_losses'], 'r-', label='Validation Loss', linewidth=2)
axes[0, 0].set_title('Training and Validation Loss', fontsize=14)
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Accuracy curves
axes[0, 1].plot(epochs, training_metrics['train_accuracies'], 'b-', label='Training Accuracy', linewidth=2)
axes[0, 1].plot(epochs, training_metrics['val_accuracies'], 'r-', label='Validation Accuracy', linewidth=2)
axes[0, 1].set_title('Training and Validation Accuracy', fontsize=14)
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Accuracy (%)')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# F1 Score curves
axes[0, 2].plot(epochs, training_metrics['train_f1_scores'], 'b-', label='Training F1', linewidth=2)
axes[0, 2].plot(epochs, training_metrics['val_f1_scores'], 'r-', label='Validation F1', linewidth=2)
axes[0, 2].set_title('Training and Validation F1 Score', fontsize=14)
axes[0, 2].set_xlabel('Epoch')
axes[0, 2].set_ylabel('F1 Score (%)')
axes[0, 2].legend()
axes[0, 2].grid(True, alpha=0.3)

# Learning rate
axes[1, 0].plot(epochs, training_metrics['learning_rates'], 'g-', linewidth=2)
axes[1, 0].set_title('Learning Rate Schedule', fontsize=14)
```

```python
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Learning Rate')
    axes[1, 0].set_yscale('log')
    axes[1, 0].grid(True, alpha=0.3)

    # Overfitting analysis
    acc_gap = [abs(t - v) for t, v in zip(training_metrics['train_accuracies'], training_metrics['val_accuracies'])]
    axes[1, 1].plot(epochs, acc_gap, 'purple', linewidth=2)
    axes[1, 1].set_title('Training-Validation Gap', fontsize=14)
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Accuracy Gap (%)')
    axes[1, 1].grid(True, alpha=0.3)

    # Best metrics summary
    best_epoch = np.argmax(training_metrics['val_accuracies']) + 1
    best_acc = max(training_metrics['val_accuracies'])
    best_f1 = max(training_metrics['val_f1_scores'])

    axes[1, 2].bar(['Best Val Acc', 'Best Val F1'], [best_acc, best_f1],
                   color=['green', 'blue'], alpha=0.7)
    axes[1, 2].set_title(f'Best Metrics (Epoch {best_epoch})', fontsize=14)
    axes[1, 2].set_ylabel('Score (%)')
    axes[1, 2].set_ylim(0, 100)

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(targets, predictions, class_names):
    """Plot enhanced confusion matrix"""
    cm = confusion_matrix(targets, predictions)
```

```python
    plt.figure(figsize=(20, 16))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=[name[:20] + '...' if len(name) > 20 else name for name in class_names],
                yticklabels=[name[:20] + '...' if len(name) > 20 else name for name in class_names])
    plt.title('Confusion Matrix - UALE Model', fontsize=16)
    plt.xlabel('Predicted Label', fontsize=14)
    plt.ylabel('True Label', fontsize=14)
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()

def plot_performance_metrics(metrics):
    """Plot comprehensive performance metrics"""
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    # Classification metrics
    classification_metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
    classification_values = [metrics['accuracy'], metrics['precision'],
                            metrics['recall'], metrics['f1_score']]

    axes[0, 0].bar(classification_metrics, classification_values,
                  color=['green', 'blue', 'orange', 'red'], alpha=0.7)
    axes[0, 0].set_title('Classification Metrics', fontsize=14)
    axes[0, 0].set_ylabel('Score')
    axes[0, 0].set_ylim(0, 1)
    axes[0, 0].tick_params(axis='x', rotation=45)

    # Model efficiency metrics
    efficiency_metrics = ['Model Size (MB)', 'GFLOPs', 'Images/sec']
    efficiency_values = [metrics['model_size_mb'], metrics['gflops'], metrics['images_per_sec']]
```

```
axes[0, 1].bar(efficiency_metrics, efficiency_values,
        color=['purple', 'brown', 'pink'], alpha=0.7)
axes[0, 1].set_title('Model Efficiency Metrics', fontsize=14)
axes[0, 1].set_ylabel('Value')
axes[0, 1].tick_params(axis='x', rotation=45)

# Parameter count
param_millions = metrics['total_params'] / 1e6
axes[0, 2].bar(['Parameters (M)'], [param_millions], color='cyan', alpha=0.7)
axes[0, 2].set_title(f'Model Parameters: {param_millions:.2f}M', fontsize=14)
axes[0, 2].set_ylabel('Parameters (Millions)')

# Confidence distribution
axes[1, 0].hist(metrics['confidences'], bins=50, alpha=0.7, color='green', edgecolor='black')
axes[1, 0].set_title('Prediction Confidence Distribution', fontsize=14)
axes[1, 0].set_xlabel('Confidence Score')
axes[1, 0].set_ylabel('Frequency')

# Uncertainty distribution
axes[1, 1].hist(metrics['uncertainties'], bins=50, alpha=0.7, color='red', edgecolor='black')
axes[1, 1].set_title('Prediction Uncertainty Distribution', fontsize=14)
axes[1, 1].set_xlabel('Uncertainty Score')
axes[1, 1].set_ylabel('Frequency')

# Model comparison (theoretical)
model_names = ['UALE', 'ResNet-50', 'EfficientNet-B0', 'MobileNet-V2']
model_params = [param_millions, 25.6, 5.3, 3.5]
model_accuracy = [metrics['accuracy'], 0.85, 0.88, 0.82]

scatter = axes[1, 2].scatter(model_params, model_accuracy,
```

```
                      s=[200, 300, 250, 220],
                      c=['red', 'blue', 'green', 'orange'], alpha=0.7)
    for i, name in enumerate(model_names):
        axes[1, 2].annotate(name, (model_params[i], model_accuracy[i]),
                          xytext=(5, 5), textcoords='offset points')
    axes[1, 2].set_xlabel('Parameters (M)')
    axes[1, 2].set_ylabel('Accuracy')
    axes[1, 2].set_title('Model Efficiency Comparison', fontsize=14)
    axes[1, 2].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

def print_detailed_metrics(metrics, class_names):
    """Print detailed performance metrics"""
    print("\n" + "="*80)
    print("COMPREHENSIVE MODEL EVALUATION RESULTS")
    print("="*80)

    print(f"\n📊 CLASSIFICATION METRICS:")
    print(f"  Accuracy:     {metrics['accuracy']:.4f} ({metrics['accuracy']*100:.2f}%)")
    print(f"  Precision:    {metrics['precision']:.4f} ({metrics['precision']*100:.2f}%)")
    print(f"  Recall:       {metrics['recall']:.4f} ({metrics['recall']*100:.2f}%)")
    print(f"  F1-Score:     {metrics['f1_score']:.4f} ({metrics['f1_score']*100:.2f}%)")

    print(f"\n🔧 MODEL EFFICIENCY:")
    print(f"  Total Parameters:     {metrics['total_params']:,} ({metrics['total_params']/1e6:.2f}M)")
    print(f"  Model Size:           {metrics['model_size_mb']:.2f} MB")
    print(f"  GFLOPs:               {metrics['gflops']:.3f}")
```

```
    print(f"\n⚡ INFERENCE PERFORMANCE:")
    print(f"  Average Inference Time: {metrics['avg_inference_time']*1000:.2f} ms")
    print(f"  Images per Second:      {metrics['images_per_sec']:.1f}")

    print(f"\n🎯 PREDICTION QUALITY:")
    print(f"  Average Confidence:   {metrics['avg_confidence']:.4f}")
    print(f"  Average Uncertainty:  {metrics['avg_uncertainty']:.4f}")

    print(f"\n📈 DATASET INFO:")
    print(f"  Total Classes:        {len(class_names)}")
    print(f"  Test Samples:         {len(metrics['targets'])}")

# ==================== MAIN EXECUTION ====================

def main():
    # Load dataset
    print("Loading dataset with fixed paths...")
    image_paths, encoded_labels, unique_labels, _ = load_benchmark_dataset_fixed()

    # Split dataset
    train_paths, test_paths, train_labels, test_labels = train_test_split(
        image_paths, encoded_labels, test_size=0.2, random_state=42, stratify=encoded_labels
    )
    train_paths, val_paths, train_labels, val_labels = train_test_split(
        train_paths, train_labels, test_size=0.15, random_state=42, stratify=train_labels
    )

    print(f"Train: {len(train_paths)}, Val: {len(val_paths)}, Test: {len(test_paths)}")
    print(f"Classes: {len(unique_labels)}")
```

```
# Get transforms
train_transform, val_transform = get_transforms()

# Create datasets
train_dataset = BenchmarkMRIDataset(train_paths, train_labels, transform=train_transform)
val_dataset = BenchmarkMRIDataset(val_paths, val_labels, transform=val_transform)
test_dataset = BenchmarkMRIDataset(test_paths, test_labels, transform=val_transform)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=4, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=4, pin_memory=True)

# Initialize model
print("Initializing ultra-lightweight UALE model...")
model = UltraLightUALE(num_classes=len(unique_labels))

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params/1e6:.2f}M")

# Train model for 100 epochs
print("Training model for 100 epochs...")
model, training_metrics = train_lightweight_with_metrics(
  model, train_loader, val_loader, epochs=100, patience=15
)

# Plot training history
print("\nPlotting training history...")
plot_training_history(training_metrics)
```

```
    # Comprehensive evaluation
    print("\nPerforming comprehensive evaluation...")
    eval_metrics = comprehensive_evaluation(model, test_loader, unique_labels)

    # Print detailed metrics
    print_detailed_metrics(eval_metrics, unique_labels)

    # Plot performance metrics
    print("\nGenerating performance visualizations...")
    plot_performance_metrics(eval_metrics)

    # Plot confusion matrix
    print("\nGenerating confusion matrix...")
    plot_confusion_matrix(eval_metrics['targets'], eval_metrics['predictions'], unique_labels)

    # Classification report
    print("\nDetailed Classification Report:")
    print(classification_report(eval_metrics['targets'], eval_metrics['predictions'],
                          target_names=unique_labels, zero_division=0))

    print("\n" + "="*80)
    print("TRAINING AND EVALUATION COMPLETE!")
    print("="*80)

    return model, training_metrics, eval_metrics

if __name__ == "__main__":
    model, training_metrics, eval_metrics = main()
```