Session 2: Hands-on Flood Mapping with U-Net

Practical Implementation for Philippine Disaster Risk Reduction

Stylianos Kotsopoulos
EU-Philippines CoPhil Programme

Hands-on Lab: Flood Mapping with U-Net

Applying Deep Learning to Philippine Disaster Response

Implement semantic segmentation for real-world flood extent mapping using Sentinel-1 SAR data

Session Overview

Duration: 2.5 hours **Format:** Hands-on Coding Lab **Platform:** Google Colab with GPU

What You'll Build:

- Complete flood mapping system using U-Net
- Trained model on Typhoon Ulysses data
- Automated flood detection from SAR imagery
- GIS-ready outputs for disaster response

Learning Objectives:

- Load and preprocess Sentinel-1 SAR data
- Implement U-Net architecture in TensorFlow
- Train with appropriate loss functions (Dice, Combined)
- Evaluate using IoU, F1-score, precision/recall
- Export results for GIS integration

Prerequisites

Required Setup:

✓ Google account with Colab access ✓ Google Drive (~500MB free space) ✓ GPU runtime enabled (T4 or better) ✓ Basic Python & NumPy knowledge ✓ Understanding of U-Net (Session 1)

Estimated Time: 2.5 hours

Access the Notebook:

Option 1: Open in Colab (Recommended)

Option 2: Download and Upload - Download from course materials - Upload to Google Drive - Open with Google Colab

Enable GPU: Runtime → Change runtime type → GPU → Save

Case Study: Central Luzon Flood Mapping

Philippine Disaster Context

Event Details:

- **Location:** Pampanga River Basin, Central Luzon
- **Event:** Typhoon Ulysses (Vamco) - November 2020
- **Impact:** Severe flooding across Bulacan, Pampanga, and surrounding provinces

Data Source:

- **Sensor:** Sentinel-1 SAR (Synthetic Aperture Radar)
- **Advantage:** Cloud-penetrating (day/night, all-weather)
- **Resolution:** 10m Ground Range Detected (GRD)
- **Polarizations:** VV and VH (dual-polarization)

Why This Matters:

Central Luzon experiences recurring floods during typhoon season. Rapid, accurate flood extent mapping is critical for:

- **Emergency response** - Identifying affected communities
- **Resource allocation** - Directing relief operations
- **Damage assessment** - Quantifying impact for recovery
- **Early warning** - Improving future prediction systems

The Challenge

Traditional flood mapping methods are slow and labor-intensive.

Deep learning with U-Net enables:

- **Automated detection** from raw SAR imagery
- **Rapid processing** of large areas within hours
- **Consistent methodology** across multiple events
- **Scalable approach** for operational disaster response

Understanding SAR for Flood Detection

VV Polarization: Vertical transmit, vertical receive

- Better for detecting open water
- Low backscatter (-30 to 10 dB)
- Flooded areas appear **dark**

VH Polarization: Vertical transmit, horizontal receive

- Sensitive to volume scattering
- Helps distinguish water from wet soil
- Values typically -30 to 0 dB

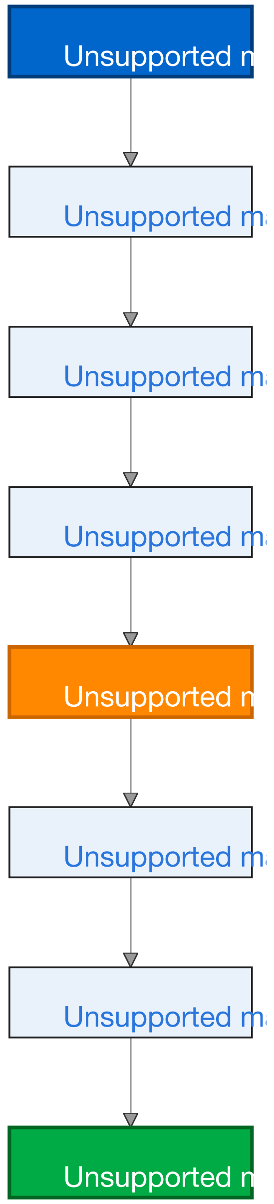
Why SAR Works for Floods:

- **All-weather imaging** - Penetrates clouds
- **Day/night capability** - Active sensor
- **Water detection** - Smooth water = low backscatter
- **Consistent response** - Physical interaction with surface

Key Principle: Flooded areas appear **dark** (low backscatter) in both VV and VH polarizations

Lab Workflow (8 Steps)

Complete Workflow



We’ll follow this systematic approach to build a complete flood mapping system.

Step 1: Setup and Data Loading

Import Libraries

```python
# Standard libraries
import numpy as np
import matplotlib.pyplot as plt
import os
from glob import glob
import random

# Deep learning framework (TensorFlow/Keras)
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models, callbacks

# Metrics and evaluation
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
import seaborn as sns

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)
random.seed(42)

print(f"TensorFlow version: {tf.__version__}")
print(f"GPU Available: {tf.config.list_physical_devices('GPU')}")
```

Dataset Information

Size: ~450MB compressed

Contents:

- ~800 training image patches (256×256, VV+VH)
- ~200 validation patches
- ~200 test patches
- Binary flood masks for all patches

Pre-processing Applied:

- Speckle filtering (Lee filter, 7×7 window)
- Radiometric calibration to σ0 (dB)
- Geometric terrain correction
- Resampling to 10m resolution

Directory Structure:

```
/data/flood_mapping_dataset/
  train/
    images/
    masks/
  val/
    images/
    masks/
  test/
    images/
    masks/
```

Download and Extract:

```python
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Dataset setup
DATA_DIR = "/content/data/flood_mapping_dataset"
```

Step 2: Data Exploration

Load Sample Data

```python
def load_sample_data(data_dir, subset='train', n_samples=5):
    """Load sample SAR images and masks"""
    img_dir = os.path.join(data_dir, subset, 'images')
    mask_dir = os.path.join(data_dir, subset, 'masks')

    img_files = sorted(glob(os.path.join(img_dir, '*.npy')))[:n_samples]
    mask_files = sorted(glob(os.path.join(mask_dir, '*.npy')))[:n_samples]

    images = [np.load(f) for f in img_files]
    masks = [np.load(f) for f in mask_files]

    return np.array(images), np.array(masks)

# Load samples
sample_images, sample_masks = load_sample_data(DATA_DIR, 'train', n_samples=5)
print(f"Sample images shape: {sample_images.shape}")  # (5, 256, 256, 2)
print(f"Sample masks shape: {sample_masks.shape}")    # (5, 256, 256, 1)
```

Visualize SAR Data

```python
def visualize_sar_samples(images, masks, n_samples=3):
    """Visualize SAR images (VV, VH) and flood masks"""
    fig, axes = plt.subplots(n_samples, 4, figsize=(16, n_samples*4))

    for i in range(n_samples):
        # VV polarization
        axes[i, 0].imshow(images[i, :, :, 0], cmap='gray', vmin=-25, vmax=5)
        axes[i, 0].set_title(f'Sample {i+1}: VV (dB)')
        axes[i, 0].axis('off')

        # VH polarization
        axes[i, 1].imshow(images[i, :, :, 1], cmap='gray', vmin=-30, vmax=0)
        axes[i, 1].set_title(f'Sample {i+1}: VH (dB)')
        axes[i, 1].axis('off')

        # Flood mask (ground truth)
        axes[i, 2].imshow(masks[i, :, :, 0], cmap='Blues', vmin=0, vmax=1)
        axes[i, 2].set_title(f'Ground Truth Mask')
        axes[i, 2].axis('off')

        # Overlay on VV
        overlay = images[i, :, :, 0].copy()
        overlay_rgb = plt.cm.gray((overlay + 25) / 30)[:, :, :3]
        mask_overlay = masks[i, :, :, 0]
        overlay_rgb[mask_overlay > 0.5] = [0, 0.5, 1]  # Blue for flood
        axes[i, 3].imshow(overlay_rgb)
```

Data Statistics

Understanding your data is crucial before training:

```
print("SAR Data Statistics:")
print(f"VV min: {sample_images[:,:,:,0].min():.2f} dB")
print(f"VV max: {sample_images[:,:,:,0].max():.2f} dB")
print(f"VV mean: {sample_images[:,:,:,0].mean():.2f} dB")
print(f"VH min: {sample_images[:,:,:,1].min():.2f} dB")
print(f"VH max: {sample_images[:,:,:,1].max():.2f} dB")
print(f"VH mean: {sample_images[:,:,:,1].mean():.2f} dB")

print("\nFlood Mask Statistics:")
flood_ratio = sample_masks.mean() * 100
print(f"Flood pixels: {flood_ratio:.2f}%")
print(f"Non-flood pixels: {100-flood_ratio:.2f}%")
print(f"Class imbalance ratio: 1:{(100-flood_ratio)/flood_ratio:.1f}")
```

Key Insight: Class imbalance requires special handling in loss function!

Step 3: Data Preprocessing

Normalization Strategy

SAR data requires proper normalization for neural network training:

```python
def normalize_sar(image, method='minmax'):
    """
    Normalize SAR backscatter values

    Methods:
    - 'minmax': Scale to [0, 1] based on typical SAR range
    - 'zscore': Standardize to mean=0, std=1
    """
    if method == 'minmax':
        # Typical SAR range: -30 to 10 dB
        vv_normalized = (image[:, :, 0] + 30) / 40  # Scale VV
        vh_normalized = (image[:, :, 1] + 35) / 35  # Scale VH
        return np.stack([vv_normalized, vh_normalized], axis=-1)

    elif method == 'zscore':
        # Standardize each channel
        mean = image.mean(axis=(0, 1), keepdims=True)
        std = image.std(axis=(0, 1), keepdims=True)
        return (image - mean) / (std + 1e-8)
```

Data Augmentation

Critical: Augment Image AND Mask Together

For segmentation, **both the image and mask must receive identical transformations**. Augmenting only the image will cause misalignment.

```python
def augment_data(image, mask, augment=True):
    """Apply data augmentation to image and mask"""
    if not augment:
        return image, mask

    # Random horizontal flip
    if np.random.random() > 0.5:
        image = np.fliplr(image)
        mask = np.fliplr(mask)

    # Random vertical flip
    if np.random.random() > 0.5:
        image = np.flipud(image)
        mask = np.flipud(mask)

    # Random 90-degree rotations (valid for nadir satellite views)
    k = np.random.randint(0, 4)  # 0, 90, 180, 270 degrees
    image = np.rot90(image, k)
    mask = np.rot90(mask, k)

    return image, mask
```

Create TensorFlow Datasets

```python
def create_tf_dataset(data_dir, subset='train', batch_size=16, augment=False):
    """Create TensorFlow dataset with preprocessing"""
    img_dir = os.path.join(data_dir, subset, 'images')
    mask_dir = os.path.join(data_dir, subset, 'masks')

    img_files = sorted(glob(os.path.join(img_dir, '*.npy')))
    mask_files = sorted(glob(os.path.join(mask_dir, '*.npy')))

    def load_and_preprocess(img_path, mask_path):
        img = np.load(img_path.numpy().decode('utf-8'))
        mask = np.load(mask_path.numpy().decode('utf-8'))
        img = normalize_sar(img, method='minmax')
        if augment:
            img, mask = augment_data(img, mask, augment=True)
        return img.astype(np.float32), mask.astype(np.float32)

    dataset = tf.data.Dataset.from_tensor_slices((img_files, mask_files))
    dataset = dataset.map(
        lambda x, y: tf.py_function(load_and_preprocess, [x, y], [tf.float32, tf.float32]),
        num_parallel_calls=tf.data.AUTOTUNE
    )
    dataset = dataset.batch(batch_size).prefetch(tf.data.AUTOTUNE)
    return dataset
```

Step 4: U-Net Model Implementation

U-Net Architecture Recap

Encoder-Decoder with Skip Connections

- **Encoder (Contracting Path):** Extract features at multiple scales
- **Bottleneck:** Deepest representation
- **Decoder (Expansive Path):** Reconstruct spatial resolution
- **Skip Connections:** Preserve fine-grained details

Input: 256×256×2 (VV + VH) **Output:** 256×256×1 (Flood probability)

Define U-Net Model (Part 1)

```python
def unet_model(input_shape=(256, 256, 2), num_classes=1):
    """U-Net architecture for binary flood segmentation"""
    inputs = keras.Input(shape=input_shape)

    # Encoder (Contracting Path)
    # Block 1
    c1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
    c1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c1)
    p1 = layers.MaxPooling2D((2, 2))(c1)

    # Block 2
    c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
    c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c2)
    p2 = layers.MaxPooling2D((2, 2))(c2)

    # Block 3
    c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
    c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c3)
    p3 = layers.MaxPooling2D((2, 2))(c3)

    # Block 4
    c4 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(p3)
    c4 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c4)
    p4 = layers.MaxPooling2D((2, 2))(c4)
```

Define U-Net Model (Part 2)

```
# Bottleneck
c5 = layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(p4)
c5 = layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(c5)

# Decoder (Expansive Path)
# Block 6
u6 = layers.Conv2DTranspose(512, (2, 2), strides=(2, 2), padding='same')(c5)
u6 = layers.concatenate([u6, c4])  # Skip connection
c6 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(u6)
c6 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c6)

# Block 7
u7 = layers.Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(c6)
u7 = layers.concatenate([u7, c3])  # Skip connection
c7 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(u7)
c7 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c7)

# Block 8
u8 = layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c7)
u8 = layers.concatenate([u8, c2])  # Skip connection
c8 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(u8)
c8 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c8)
```

Define U-Net Model (Part 3)

```python
    # Block 9
    u9 = layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c8)
    u9 = layers.concatenate([u9, c1])  # Skip connection
    c9 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(u9)
    c9 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c9)

    # Output layer
    outputs = layers.Conv2D(num_classes, (1, 1), activation='sigmoid')(c9)

    model = keras.Model(inputs=[inputs], outputs=[outputs], name='U-Net')
    return model

# Build model
model = unet_model(input_shape=(256, 256, 2), num_classes=1)
model.summary()
```

Total Parameters: ~31 million trainable parameters

Loss Functions

Implementing specialized loss functions for segmentation:

```python
def dice_coefficient(y_true, y_pred, smooth=1e-6):
    """Dice coefficient for evaluation"""
    y_true_f = tf.keras.backend.flatten(y_true)
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (
        tf.keras.backend.sum(y_true_f) + tf.keras.backend.sum(y_pred_f) + smooth
    )

def dice_loss(y_true, y_pred):
    """Dice loss for training"""
    return 1 - dice_coefficient(y_true, y_pred)

def combined_loss(y_true, y_pred):
    """Combined Binary Cross-Entropy + Dice Loss"""
    bce = tf.keras.losses.binary_crossentropy(y_true, y_pred)
    dice = dice_loss(y_true, y_pred)
    return 0.5 * bce + 0.5 * dice

def iou_score(y_true, y_pred, smooth=1e-6):
    """IoU metric (Intersection over Union)"""
    y_true_f = tf.keras.backend.flatten(y_true)
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true_f * y_pred_f)
    union = tf.keras.backend.sum(y_true_f) + tf.keras.backend.sum(y_pred_f) - intersection
    return (intersection + smooth) / (union + smooth)
```

Step 5: Model Training

Compile Model

```python
# Compile with combined loss
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-4),
    loss=combined_loss,
    metrics=['accuracy', dice_coefficient, iou_score]
)
```

Why Combined Loss?

- **Binary Cross-Entropy:** Pixel-wise classification accuracy
- **Dice Loss:** Handles class imbalance effectively
- **Combination:** Best of both worlds for flood segmentation

Setup Callbacks

```
# Create directories
os.makedirs('/content/models', exist_ok=True)
os.makedirs('/content/logs', exist_ok=True)

# Callbacks for training
checkpoint_cb = callbacks.ModelCheckpoint(
    '/content/models/unet_flood_best.h5',
    monitor='val_iou_score',
    mode='max',
    save_best_only=True,
    verbose=1
)

early_stop_cb = callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True,
    verbose=1
)

reduce_lr_cb = callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=5,
    min_lr=1e-7,
    verbose=1
```

Train the Model

⚠ **Training Time Estimate**

- **With GPU (T4):** 15-25 minutes for 50 epochs
- **With CPU:** 4-6 hours (not recommended)

The model will likely converge in 20-30 epochs with early stopping.

```
# Train model
EPOCHS = 50

history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS,
    callbacks=callback_list,
    verbose=1
)
```

Monitor: Loss, Dice Coefficient, IoU Score (train & validation)

Visualize Training History

```python
def plot_training_history(history):
    """Plot training and validation metrics"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Loss
    axes[0, 0].plot(history.history['loss'], label='Train Loss')
    axes[0, 0].plot(history.history['val_loss'], label='Val Loss')
    axes[0, 0].set_title('Model Loss')
    axes[0, 0].legend()

    # Dice Coefficient
    axes[0, 1].plot(history.history['dice_coefficient'], label='Train Dice')
    axes[0, 1].plot(history.history['val_dice_coefficient'], label='Val Dice')
    axes[0, 1].set_title('Dice Coefficient')
    axes[0, 1].legend()

    # IoU Score
    axes[1, 0].plot(history.history['iou_score'], label='Train IoU')
    axes[1, 0].plot(history.history['val_iou_score'], label='Val IoU')
    axes[1, 0].set_title('IoU Score')
    axes[1, 0].legend()

    # Accuracy
    axes[1, 1].plot(history.history['accuracy'], label='Train Acc')
    axes[1, 1].plot(history.history['val_accuracy'], label='Val Acc')
    axes[1, 1].set_title('Pixel Accuracy')
```

Step 6: Model Evaluation

Load Best Model

After training completes, load the best model weights:

```
# Load the best model
best_model = keras.models.load_model(
    '/content/models/unet_flood_best.h5',
    custom_objects={
        'combined_loss': combined_loss,
        'dice_coefficient': dice_coefficient,
        'iou_score': iou_score
    }
)

print("✓ Best model loaded successfully")
```

Evaluate on Test Set

```python
# Evaluate on test dataset
test_results = best_model.evaluate(test_dataset, verbose=1)

print("\n" + "="*50)
print("TEST SET RESULTS")
print("="*50)
print(f"Loss: {test_results[0]:.4f}")
print(f"Pixel Accuracy: {test_results[1]:.4f}")
print(f"Dice Coefficient: {test_results[2]:.4f}")
print(f"IoU Score: {test_results[3]:.4f}")
print("="*50)
```

Expected Results:

- IoU: 0.65-0.80
- Dice: 0.70-0.85
- Accuracy: 0.85-0.95

Detailed Metrics Calculation

```python
def calculate_detailed_metrics(model, dataset):
    """Calculate comprehensive segmentation metrics"""
    y_true_all = []
    y_pred_all = []

    for images, masks in dataset:
        predictions = model.predict(images, verbose=0)
        y_true_all.append(masks.numpy().flatten())
        y_pred_all.append((predictions > 0.5).astype(np.float32).flatten())

    y_true = np.concatenate(y_true_all)
    y_pred = np.concatenate(y_pred_all)

    from sklearn.metrics import precision_score, recall_score, f1_score

    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)

    # Confusion matrix components
    tp = np.sum((y_true == 1) & (y_pred == 1))
    tn = np.sum((y_true == 0) & (y_pred == 0))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))

    return {'precision': precision, 'recall': recall, 'f1_score': f1,
```

Interpreting Results

Good Performance Indicators:

- **IoU > 0.70:** Strong overlap
- **High Precision:** Few false alarms
- **High Recall:** Catches most floods
- **F1 > 0.75:** Balanced performance

For Disaster Response:

- **Precision matters:** Avoid sending resources to non-flooded areas
- **Recall matters more:** Don't miss flooded communities
- Trade-off depends on operational priorities

Example: Higher recall (0.85) with moderate precision (0.75) may be preferred

Step 7: Visualization

Predict on Test Samples

```python
def visualize_predictions(model, dataset, n_samples=5):
    """Visualize model predictions vs ground truth"""
    images, masks = next(iter(dataset))
    predictions = model.predict(images[:n_samples], verbose=0)

    fig, axes = plt.subplots(n_samples, 4, figsize=(20, n_samples*5))

    for i in range(n_samples):
        # Original SAR VV
        axes[i, 0].imshow(images[i, :, :, 0], cmap='gray', vmin=0, vmax=1)
        axes[i, 0].set_title(f'SAR VV (Normalized)')

        # Ground Truth
        axes[i, 1].imshow(masks[i, :, :, 0], cmap='Blues', vmin=0, vmax=1)
        axes[i, 1].set_title('Ground Truth Mask')

        # Prediction
        iou_val = iou_score(masks[i:i+1], predictions[i:i+1]).numpy()
        axes[i, 2].imshow(predictions[i, :, :, 0], cmap='Blues', vmin=0, vmax=1)
        axes[i, 2].set_title(f'Prediction (IoU: {iou_val:.3f})')

        # Error Overlay: TP=Green, FP=Red, FN=Yellow
        overlay = np.zeros((256, 256, 3))
        gt = masks[i, :, :, 0] > 0.5
        pred = predictions[i, :, :, 0] > 0.5
        overlay[gt & pred] = [0, 1, 0]
```

Error Analysis

Common Error Patterns:

False Positives (Red):

- Wet soil after rain
- Shadows in mountainous terrain
- Calm water bodies (pre-flood)

False Negatives (Yellow):

- Flooded vegetation
- Mixed pixels at boundaries
- Speckle noise in SAR data

Improvement Strategies:

- Multi-temporal data (before/after)
- Incorporate DEM (elevation data)
- Ensemble multiple models
- Post-processing with GIS constraints
- Contextual filtering

Step 8: Export for GIS

Save Trained Model

```
# Save model in different formats
best_model.save('/content/models/unet_flood_final.h5')
best_model.save('/content/models/unet_flood_final.keras')

# Save to Google Drive for persistence
!cp /content/models/unet_flood_final.h5 /content/drive/MyDrive/flood_mapping/

print("✓ Model saved successfully")
```

Export Predictions

```python
def export_predictions(model, dataset, output_dir='/content/outputs'):
    """Export predictions as NumPy arrays"""
    os.makedirs(output_dir, exist_ok=True)

    batch_idx = 0
    for images, masks in dataset:
        predictions = model.predict(images, verbose=0)

        for i in range(len(images)):
            # Save prediction (probability map)
            pred_file = os.path.join(output_dir, f'prediction_{batch_idx:04d}.npy')
            np.save(pred_file, predictions[i])

            # Save binary mask (threshold at 0.5)
            binary_file = os.path.join(output_dir, f'binary_mask_{batch_idx:04d}.npy')
            binary_mask = (predictions[i] > 0.5).astype(np.uint8)
            np.save(binary_file, binary_mask)

            batch_idx += 1

    print(f"✓ Exported {batch_idx} predictions to {output_dir}")
```

GIS Integration

1. Georeferencing:

- Match predictions to SAR coordinates
- Use Sentinel-1 GRD metadata

2. Vectorization:

```python
# Requires rasterio and geopandas
from rasterio.features import shapes
import geopandas as gpd

mask = (prediction > 0.5).astype(np.uint8)
shapes_gen = shapes(mask,
                    transform=affine_transform)
polygons = [shape(s) for s, v in shapes_gen
            if v == 1]

gdf = gpd.GeoDataFrame(
    {'geometry': polygons},
    crs='EPSG:4326'
)
gdf.to_file('flood_extent.geojson')
```

3. Export Formats:

- **GeoTIFF:** Raster for GIS
- **Shapefile/GeoJSON:** Vector polygons
- **KML:** Google Earth

4. Integration:

- Load in QGIS/ArcGIS
- Overlay with admin boundaries
- Calculate affected area/population
- Generate response maps

Troubleshooting

Common Issues & Solutions

Out of Memory:

```
# Reduce batch size
BATCH_SIZE = 8

# Mixed precision
from tensorflow.keras import mixed_precision
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_global_policy(policy)

# Clear session
from tensorflow.keras import backend as K
K.clear_session()
```

Model Not Learning:

```
# Check normalization
print(f"Range: {images.min()}, {images.max()}")

# Verify labels
print(f"Masks: {np.unique(masks)}")

# Adjust learning rate
optimizer = keras.optimizers.Adam(lr=5e-4)
```

Overfitting:

```
# Stronger augmentation
image = image * np.random.uniform(0.8, 1.2)
image += np.random.normal(0, 0.05, image.shape)

# Add dropout
c1 = layers.Dropout(0.2)(c1)
```

Colab Disconnections:

```
# Save frequently
checkpoint_cb = callbacks.ModelCheckpoint(
    filepath='model.h5',
    save_freq='epoch'
)

# Save to Drive
drive.mount('/content/drive')
model.save('/content/drive/MyDrive/model.h5')
```

Key Takeaways

What You've Accomplished

Technical Skills:

✓ Loaded and preprocessed Sentinel-1 SAR data ✓ Implemented complete U-Net architecture ✓ Trained with appropriate loss functions ✓ Evaluated using IoU, Dice, F1 metrics ✓ Visualized and interpreted predictions ✓ Exported results for GIS integration

Conceptual Understanding:

✓ SAR backscatter → flood detection ✓ Skip connections → precise boundaries ✓ Class imbalance → special loss functions ✓ Precision vs recall trade-offs ✓ Error patterns and improvements

Philippine DRR Context:

✓ Applied to Typhoon Ulysses data ✓ Operational disaster response ✓ Integration with PAGASA/DOST systems

Impact:

Your skills can now contribute to **saving lives** through rapid, accurate flood mapping for Philippine disaster response.

Critical Lessons

1. **Data Quality >> Model Complexity**
   - Well-prepared SAR data more important than model tweaks
   - Ground truth quality directly impacts performance
2. **Loss Function Selection Matters**
   - Combined loss (BCE + Dice) best for imbalanced data
   - Pure cross-entropy fails when flood pixels <10%
3. **Evaluation Beyond Accuracy**
   - Pixel accuracy misleading for imbalanced classes
   - IoU and Dice give true performance picture
4. **Operational Considerations**
   - For disaster response, recall > precision
   - Speed matters: Train once, inference in minutes
   - GIS integration essential for actionable outputs

Expected Results

Metric	Expected Range	Interpretation
IoU (Test)	0.65 - 0.80	Good to excellent overlap
Dice Coefficient	0.70 - 0.85	Strong agreement
Precision	0.70 - 0.90	Few false alarms
Recall	0.75 - 0.95	Catches most floods
F1-Score	0.72 - 0.88	Balanced performance
Training Time	15-30 min	With GPU (T4)

If Results Are Lower:

- Check data quality and normalization
- Adjust learning rate or loss function
- Increase training epochs or data augmentation

Discussion Questions

Reflection Questions

1. **Real-World Application:**
   - How would you deploy this for real-time disaster response?
   - What infrastructure and pipelines needed?
2. **Model Limitations:**
   - What types of floods might this model miss?
   - How to validate predictions without ground truth?
3. **Improvements:**
   - How to use multi-temporal data (before/after)?
   - How to incorporate elevation data (DEM)?
4. **Operational Challenges:**
   - What's acceptable latency for disaster response?
   - How to handle uncertainty quantification?
5. **Ethical Considerations:**
   - What if the model misses a flooded community?
   - How to balance automation with human expertise?

Resources

Datasets and Tools

Flood Mapping:

- Sen1Floods11
- FloodNet
- UNOSAT Flood Portal

SAR Data:

- Copernicus Hub
- Alaska Satellite Facility
- Google Earth Engine

Philippine EO:

- PhilSA Space+ Data Dashboard
- DOST-ASTI DATOS
- NAMRIA GeoPortal
- PAGASA

Code Repositories:

- Segmentation Models
- TorchGeo
- RasterVision

Papers

U-Net:

- Original U-Net Paper (Ronneberger et al., 2015)
- TensorFlow Segmentation Tutorial

SAR Flood Mapping:

- Flood Detection with SAR: A Review
- Deep Learning for SAR
- Automated Flood Mapping

Loss Functions:

- Dice Loss
- Focal Loss

Next Steps

Preparation for Session 3

Session 3: Object Detection

Topics:

- R-CNN, YOLO, SSD architectures
- Bounding box regression
- Anchor boxes and NMS
- Applications: Ship, building, vehicle detection

Key Differences:

- **Segmentation:** Pixel-wise classification
- **Detection:** Object localization with boxes

Preparation:

- Review CNN concepts from Day 2
- Understand segmentation vs detection
- Consider EO applications

Think About:

- When to use detection vs segmentation?
- How to combine both approaches?
- Philippine DRR applications?

Lab Completion Checklist

Before finishing, ensure you've completed:

- [ ] Successfully trained U-Net model
- [ ] Achieved IoU > 0.60 on test set
- [ ] Visualized predictions vs ground truth
- [ ] Analyzed error patterns
- [ ] Saved trained model to Google Drive
- [ ] Exported predictions
- [ ] Understood troubleshooting strategies
- [ ] Thought about operational deployment

Congratulations!

You’ve Built a Production-Ready Flood Mapping System

What You Built:

- Trained semantic segmentation model
- Automated flood detection system
- Export pipeline for GIS integration
- Performance evaluation framework

Impact:

Your skills can contribute to **saving lives** through rapid, accurate flood extent mapping for Philippine disaster response operations.

Time Plan

Block	Minutes	Cumulative
Setup + Exploration	25	0:25
Preprocessing	20	0:45
U-Net Implementation	15	1:00
Model Training	35	1:35
Evaluation	15	1:50
Visualization	10	2:00
Export + GIS	10	2:10
Troubleshooting	10	2:20
Buffer	10	2:30

Total: 2.5 hours

Start the Lab!

Access the notebook and begin your flood mapping journey.

Questions? Ask your instructor or teaching assistants.

Good luck!

This hands-on lab is part of the CoPhil 4-Day Advanced Training on AI/ML for Earth Observation, funded by the European Union under the Global Gateway initiative. Materials developed in collaboration with PhilSA, DOST-ASTI, and the European Space Agency.