ChatGPT

Day 3, Session 2: Flood Mapping with U-Net and Sentinel-1 SAR

Learning Objectives

- Implement a **U-Net** convolutional neural network to perform flood **segmentation** on Sentinel-1 SAR imagery.
- Load and preprocess **dual-polarization** SAR data (VV & VH) and corresponding flood **mask** labels for training.
- Apply data **augmentation** techniques (flips, rotations) tailored to SAR to improve model generalization.
- Construct a **small U-Net** architecture with encoder–decoder layers and **skip connections** to preserve spatial details.
- Train the U-Net using a suitable **loss function** (Dice loss and/or cross-entropy) and **optimizer** (Adam), monitoring training and validation performance.
- Evaluate the trained model on a test set using **segmentation metrics** (IoU, precision, recall, F1) and understand their significance.
- **Visualize** predictions by overlaying model output masks on SAR images for qualitative assessment of flood extent mapping.

Case Context: Central Luzon Flooding and SAR for Disaster Mapping

In this session, we tackle a real-world scenario of flood mapping in Central Luzon, Philippines. In recent years, intense typhoons have caused severe flooding in this region. For example, Category-4 *Typhoon Vamco* (local name *Ulysses*) struck Central Luzon in November 2020, resulting in widespread floods [1]. Likewise, *Typhoon Noru* (local name *Karding*) in 2022 brought torrential rains and flooded over 140 areas, mostly in Central Luzon [2]. Rapidly identifying inundated areas is critical for disaster response, but often challenging due to cloud cover and power outages limiting ground surveys.

Why SAR for flood mapping? Unlike optical satellites (e.g. Sentinel-2) which cannot penetrate thick clouds, **Synthetic Aperture Radar (SAR)** can image the Earth **day or night and through clouds**, making it ideal for floods occurring during storms [3]. Sentinel-1's C-band SAR has a ~6-day revisit and operates in dual polarizations (VV and VH), providing frequent, weather-independent observations [3]. Floodwater typically appears as dark (low backscatter) regions on SAR images (especially if the water surface is calm), so SAR can delineate water extent even when optical sensors are blinded by clouds [4]. In this session, we use **Sentinel-1 SAR** imagery (VV and VH channels) of a flood event (such as the Ulysses 2020 flood) and aim to map the flood extent with a deep learning model.

By using a **U-Net** model, a type of fully convolutional network originally developed for biomedical segmentation, we can train a network to classify each pixel of the SAR image as "flood" or "no-flood". Deep learning approaches have shown promise in improving flood mapping accuracy by reducing false alarms and missed detections compared to simple threshold methods [5]. The U-Net's encoder-decoder structure and skip connections are well-suited to learn the **context** of flooding (e.g., water along river valleys) while preserving **fine details** (exact flood boundaries), even in the presence of SAR speckle noise or complex terrain.

Data Setup: We assume you have preprocessed **patches** of Sentinel-1 SAR data and corresponding flood masks. Each SAR patch has two channels (VV and VH intensities) and a binary mask marking flooded areas (1) vs non-flood (0). These patches may be of size 128×128 or 256×256 pixels for manageable model input. The SAR data might be provided as NumPy arrays or image files (after radiometric calibration and terrain correction), and the masks are likewise in a convenient format. Our goal is to load this data, build and train a U-Net to segment flood pixels, and evaluate its performance.

Module 1: Load SAR Patches and Flood Masks

In this module, we'll load the SAR input data (VV and VH channels) and the corresponding flood mask for each patch. We assume the data is stored in a structured way (e.g., separate files for each patch's VV, VH, and mask, or a combined NumPy file per patch).

Step-by-Step Instructions:

1. **Locate the data files:** Identify where the SAR patch data and masks are stored. For example, you might have a directory with filenames like `patch_001_VV.npy`, `patch_001_VH.npy`, and `patch_001_mask.npy`, or perhaps a single `.npz` file containing all patches in arrays.
2. **Load the SAR channels:** Use `numpy.load` or an image reading library to load the VV and VH channel data for a patch. Combine them into a single multi-channel array (shape `[2, H, W]`). If the data is in image format (e.g., TIFF or PNG), use PIL (`Image.open`) or OpenCV to read and then convert to a NumPy array.
3. **Load the mask:** Load the flood mask for the same patch, which should be a binary 2D array of shape `[H, W]` (or `[1, H, W]` if stored with a channel dimension). Verify that the mask aligns with the SAR data spatially.
4. **Preprocess values:** Convert data to floating point tensors. If the SAR values are not normalized, consider converting them to a suitable scale. Commonly, SAR backscatter is given in decibels (dB); if so, you might normalize by a fixed range or mean/standard deviation. At minimum, scaling the inputs to [0,1] or standardizing can help stable training. (For this session, if data is pre-normalized, you can skip additional normalization.)
5. **Batch and dataset preparation:** If you have many patches, you may create a PyTorch `Dataset` class to load them on the fly, and then a `DataLoader` to batch them. For example, a custom `FloodDataset` could implement `__len__` and `__getitem__` to return `(sar_tensor, mask_tensor)` for each index. This allows convenient shuffling and batching during training.

Below is a **code snippet** demonstrating how you might load one sample (assuming NumPy files for simplicity). In practice, you'll loop this for all training patches or use a DataLoader.

```
import numpy as np
import torch

# Example: load a single patch (VV, VH, and mask)
```

```
vv = np.load("data/patch_001_VV.npy")         # shape: (H, W)
vh = np.load("data/patch_001_VH.npy")         # shape: (H, W)
mask = np.load("data/patch_001_mask.npy")     # shape: (H, W), values 0 or 1

# Stack VV and VH into a 2-channel array
sar_patch = np.stack([vv, vh], axis=0)        # shape: (2, H, W)

# Convert to torch tensors
sar_tensor = torch.from_numpy(sar_patch).float()
mask_tensor = torch.from_numpy(mask).float()

print("SAR tensor shape:", sar_tensor.shape)
print("Mask tensor shape:", mask_tensor.shape)
```

When you run this, you should see the SAR tensor shape as **(2, H, W)**, confirming the two channels, and the mask tensor shape as **(H, W)**. Before training, if using PyTorch, you might also add a batch dimension (e.g., shape becomes (N, 2, H, W) for a batch of N patches) – but the DataLoader can handle that batching automatically.

Think-Through: Why do we use **both VV and VH polarizations** as input? Consider that VV (vertical transmit, vertical receive) and VH (vertical transmit, horizontal receive) channels capture different backscatter characteristics. For open water, both VV and VH often show low backscatter (dark areas) due to specular reflection away from the sensor. However, certain flooded areas (e.g., water under vegetation or rough water surfaces) might appear differently in VH vs VV. Using both channels gives the model more information to distinguish flood water from other dark features (like shadows or asphalt). *Question:* If one polarization sometimes shows the flood more clearly than the other, how might the network learn to use the channels? (Hint: It could learn to weight the more informative channel for a given pixel, effectively learning from both.)

Mini-Challenge: The data here is already pre-sliced into patches. In a real scenario, you might need to **tile a large Sentinel-1 image into patches** for processing, or handle streaming data. As a challenge, think about how you would modify the loading process if the data wasn't pre-divided. For instance, how could you use a sliding window to generate overlapping patches from a big image? Additionally, try writing a simple `torch.utils.data.Dataset` that yields data from a list of file paths, and test it by iterating and checking tensor shapes. This will set the stage for feeding data into the network efficiently.

Module 2: Data Augmentation for SAR Imagery (Optional)

To improve model robustness, we can apply **data augmentation** – transformations that create new plausible training samples from existing ones. Augmentation helps the model generalize better, especially with limited data, by exposing it to varied orientations and scenarios. For SAR flood mapping, we should choose augmentations that **preserve the semantic content** (flood vs non-flood) while adding diversity.

Suitable augmentations for SAR flood data:

- **Flips:** Horizontal and vertical flips are typically safe for SAR flood images. Flipping a SAR patch left-right or up-down doesn't change the fact that certain pixels are flooded, but gives the model a new perspective (since flood patterns have no preferred orientation). Many flood datasets use horizontal/vertical flips during training [6].
- **Rotations:** 90-degree rotations (or any multiples of 90°) are also good augmentations, as they essentially permute image orientation without altering pixel labels. Random 90° rotations can quadruple the variety of patch orientation.
- **Random Crops or Zoom:** If the context allows, randomly cropping a patch or zooming can teach the model to handle different framing, though care must be taken not to cut out critical flood areas entirely.
- **Noise augmentation:** SAR inherently has **speckle noise**. We might simulate slight speckle or add Gaussian noise to make the model more noise-resistant. However, be cautious: adding too much noise might confuse training, so minor noise augmentation is usually enough if used.
- **Intensity shifts:** Generally, you would not arbitrarily change SAR intensity because the absolute backscatter values carry meaning (e.g., water's low backscatter). We usually avoid brightness shifts or contrast adjustments that are common in optical image augmentation. Instead, focusing on geometric transformations (flips/rotations) is safer for SAR.

When applying augmentation for segmentation, **ensure the mask undergoes the same transform** as the image. For example, if you flip an image horizontally, you must also flip its mask horizontally so that the flood pixels remain aligned.

Example Code for Augmentations: We can use PyTorch's `torchvision.transforms` or write custom augmentation functions. Here's an example using simple horizontal flip and 90° rotation:

```
import torch
import random

def random_flip_and_rotate(sar_tensor, mask_tensor):
    # Horizontal flip with 50% chance
    if random.random() < 0.5:
        sar_tensor = torch.flip(sar_tensor, dims=[2])       # flip horizontally
(dim=2 assuming shape (C,H,W))
        mask_tensor = torch.flip(mask_tensor, dims=[1])     # flip horizontally
(dim=1 for shape (H,W))
    # Vertical flip with 50% chance
    if random.random() < 0.5:
        sar_tensor = torch.flip(sar_tensor, dims=[1])       # flip vertically
(dim=1 for SAR tensor)
        mask_tensor = torch.flip(mask_tensor, dims=[0])     # flip vertically
(dim=0 for mask)
    # 90-degree rotations (0, 90, 180, 270 degrees)
    k = random.randint(0, 3)  # random choice
    if k:
        sar_tensor = torch.rot90(sar_tensor, k, dims=[1, 2])
        mask_tensor = torch.rot90(mask_tensor, k, dims=[0, 1])
    return sar_tensor, mask_tensor

# Usage on a sample patch:
aug_sar, aug_mask = random_flip_and_rotate(sar_tensor, mask_tensor)
```

In this snippet, `torch.flip` and `torch.rot90` are used for convenience. We flip along appropriate dimensions (taking care that the channel dimension is not flipped, only spatial dims), and rotate both SAR and mask by the same random `k*90` degrees. This augmented output can then be fed into the model. If using a DataLoader, you could incorporate such augmentation in the `__getitem__` of the Dataset (for training data only).

Think-Through: What **augmentations might be unnecessary or risky** for SAR flood data? One example: applying a random brightness or contrast change (often done with optical images) could be problematic – in SAR, absolute pixel intensity matters (water is dark, so artificially brightening a water pixel could confuse the model). Also, extreme geometric transforms like large skew or perspective warp aren't typical for SAR patches and may create unrealistic images. By focusing on flips and rotations, we ensure the physical characteristics of SAR floods remain plausible. *Question:* SAR images have a look direction (the satellite side from which radar illuminates the scene). Do you think flipping an image left-right (which changes the look direction relative positions) could ever cause an inconsistency? (Hint: If the SAR image had a subtle shadow or layover pattern due to look angle, a horizontal flip might create a situation not physically observed, but for flood segmentation this is usually negligible. Flips are commonly used as noted in literature [6].)

Mini-Challenge: Augmentations can be expanded. Try adding a **random rotation by an arbitrary angle** (not just multiples of 90°) – this is more complex because you'd need to interpolate pixels and it may blur the mask edges slightly. If you're up for it, implement a random small rotation (say ±10°) on the SAR image and mask. Another idea: simulate **speckle noise** by multiplying the SAR patch by a random noise field (e.g., pixelwise independent noise drawn from a Rayleigh distribution to mimic speckle). Keep the noise level low so as not to overwhelm the signal. Does the model trained with such noise augmentation perform better on real noisy SAR images?

Module 3: Define a U-Net Model in PyTorch

Now for the heart of the session – implementing the **U-Net architecture**. U-Net is a specialized CNN that features an **encoder-decoder** structure with symmetric skip connections, forming a "U" shape. The encoder (contracting path) consists of convolutional layers and pooling that progressively **downsample** the feature maps, capturing higher-level context but losing spatial resolution. The decoder (expanding path) uses upsampling (transpose convolutions) to **increase resolution** and make per-pixel predictions. **Skip connections** directly connect corresponding encoder and decoder layers, **copying encoder features** to the decoder to help recover fine-grained details that were lost in downsampling [7]. This combination allows U-Net to precisely **localize** where the features of interest (flooded pixels) are, while still using the encoder's abstract understanding of the scene [7].

Let's break down the U-Net components for our flood mapping task:

- **Input & Output:** The model input is a 2-channel SAR patch (VV,VH). The output is a single-channel prediction map (of the same spatial size) with values that can be interpreted as flood probability per pixel.
- **Encoder:** We'll have a few convolutional blocks that gradually reduce spatial size. Each block typically has two 3×3 convolutions (with ReLU activations) followed by a downsampling (max pooling) to halve the H and W. We often start with a certain number of feature **filters** and double the number of filters after each pooling (to capture more abstract features in deeper layers). A small U-Net might have 2–4 downsampling steps. For example:
- Block1: input 2 channels -> conv -> conv -> output 16 feature maps, then 2×2 pool (down to 1/2 size).
- Block2: input 16 -> conv -> conv -> output 32 feature maps, then pool (down to 1/4 original size).
- Block3: input 32 -> conv -> conv -> output 64 features, then pool (down to 1/8 size).
- (Maybe one more block if using 256x256 images, going to 128 features and 1/16 size).
- Bottleneck: convs at the lowest resolution (e.g., 64 or 128 features, 1/8 or 1/16 of input size).
- **Decoder:** For each downsampling step, we have a corresponding upsampling step. We use a transposed convolution (learnable upsampling, also called deconvolution) to double the spatial size and reduce the number of feature maps (typically halve the filters compared to the encoder at that level). Then we **concatenate** the upsampled features with the features from the encoder **skip connection** (i.e., attach the encoder's feature map from the matching level). Then apply two 3×3 convs to fuse them and generate the decoder's output for that level.
- E.g., the decoder corresponding to Block3: upsample bottleneck output (64 features -> 64 features but double size) then concatenate with encoder Block3's output (64 features), resulting in 128 channels, then conv -> conv to output, say, 32 features (because we expect to match Block2's size).
- Continue for Block2: upsample 32 -> concat with 32 from encoder Block2 -> convs -> output 16.
- Block1 decoder: upsample 16 -> concat with 16 from Block1 encoder -> convs -> output maybe 16 again.
- **Final layer:** A 1×1 convolution at the end to map the last decoder output to 1 channel (the segmentation map). This is typically followed by a non-linear activation. For binary segmentation, we will use a **sigmoid** activation to get probabilities between 0 and 1 for "flood".

We will implement this step-by-step. Below is a **code snippet** for a simplified U-Net. This model has an encoder with two downsampling steps (so 3 levels: 128→64→32 spatial, if input is 128, it ends at 32) and a decoder mirroring it. We use small numbers of filters for simplicity. You can expand this structure as needed (e.g., add another down/up pair for larger images).

```python
import torch.nn as nn
import torch

class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        # Encoder: Conv block 1
        self.enc_conv1 = nn.Sequential(
            nn.Conv2d(2, 16, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, padding=1), nn.ReLU(inplace=True)
        )
        self.pool1 = nn.MaxPool2d(2)  # 128 -> 64

        # Encoder: Conv block 2
        self.enc_conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.ReLU(inplace=True)
        )
        self.pool2 = nn.MaxPool2d(2)  # 64 -> 32

        # Bottleneck
        self.bottleneck = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1), nn.ReLU(inplace=True)
        )

        # Decoder: Upsample block 2
        self.upconv2 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
# 32 -> 64
        self.dec_conv2 = nn.Sequential(
            nn.Conv2d(32 + 32, 32, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.ReLU(inplace=True)
        )
        # Decoder: Upsample block 1
        self.upconv1 = nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2)
# 64 -> 128
        self.dec_conv1 = nn.Sequential(
            nn.Conv2d(16 + 16, 16, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, padding=1), nn.ReLU(inplace=True)
        )
        # Final output layer
        self.final_conv = nn.Conv2d(16, 1, kernel_size=1)

    def forward(self, x):
        # Encoder forward
        e1 = self.enc_conv1(x)        # e1: (16, H, W)
        e2 = self.enc_conv2(self.pool1(e1))  # e2: (32, H/2, W/2)
        b  = self.bottleneck(self.pool2(e2))  # b: (64, H/4, W/4)
        # Decoder forward with skips
        d2 = self.upconv2(b)                   # upsample bottleneck to (32, H/
2, W/2)
        d2 = torch.cat([d2, e2], dim=1)        # concat skip from e2 -> shape:
(64, H/2, W/2)
        d2 = self.dec_conv2(d2)
# process after skip -> (32, H/2, W/2)
        d1 = self.upconv1(d2)                  # upsample to (16, H, W)
        d1 = torch.cat([d1, e1], dim=1)        # concat skip from e1 -> shape:
```

```
(32, H, W)
        d1 = self.dec_conv1(d1)                # -> (16, H, W)
        out = self.final_conv(d1)              # -> (1, H, W)
        # Note: we'll apply sigmoid later (in loss or evaluation)
        return out

# Instantiate the model and test a forward pass
model = UNet()
dummy_input = torch.randn(1, 2, 128, 128)  # batch=1, channels=2, size=128x128
output = model(dummy_input)
print("Output shape:", output.shape)
```

This `UNet` class defines the encoder and decoder as described. Key things to notice:

- We used `padding=1` on conv layers so that the spatial size remains the same after each conv (easier concatenation, as no cropping needed). The pooling halves the size, and ConvTranspose with stride=2 doubles it back.
- **Skip connection concatenation:** `torch.cat([d2, e2], dim=1)` is where the magic happens for skip connections. We concatenate along the channel dimension (dim=1 for NCHW tensor) so the decoder stage gets both its current feature maps and the encoder's feature maps of the same spatial size. For example, at `d2` stage, we had 32 channels from upsampling, and we concatenate the 32 channels from `e2`, giving 64 channels which `dec_conv2` then reduces back to 32.
- The final conv outputs 1 channel. We did not apply `torch.sigmoid` in the `forward` because we will use a loss that expects raw logits (like `BCEWithLogitsLoss` which combines a sigmoid internally for numerical stability). We can always apply sigmoid at prediction time to get probabilities.

Running the above snippet should show `Output shape: torch.Size([1, 1, 128, 128])`, meaning for a 128×128 input, we recovered a 128×128 mask prediction.

U-Net Diagram & Mnemonic: Think of the U-Net in two halves. The **encoder** is like a funnel narrowing down: it's learning "**what**" is in the image (e.g., "somewhere here is water"). The **decoder** is like a magnifying glass expanding out: it's learning "**where**" those things are in the image, constructing a pixel map. The skip connections are like **bridges** carrying high-resolution details from the encoder to decoder. A simple mnemonic: *"Down to understand, up to pinpoint, skip to refine."* The skip connections ensure that the **fine edges of floods** (which the early layers see clearly) are not lost by the time we get to the output [7]. They effectively add the encoder's **fine-grained shallow features** to the decoder's coarse feature map, enabling precise localization of flood boundaries [7].

Illustratively, a skip connection in U-Net can be shown as:

```
Encoder feature map -->=====================>--| concat in decoder
                      (skip connection)
```

Where the `-->===...===>--` line represents copying the feature map from an encoder layer over to a decoder layer of the same level.

Think-Through: What would happen if we **omit skip connections**? Our decoder would then rely solely on the bottleneck output, which is very low resolution, to infer fine details. Likely, the output flood mask would be blob-like and miss precise boundaries – small flooded fields or narrow water channels could vanish. Skip connections retain spatial context, helping differentiate, say, a narrow river vs the background. *Question:* U-Net has many feature maps (16, 32, 64 in our model). Why do we increase the number of filters as we go deeper? (Hint: As spatial size decreases, we can afford more feature channels. These deeper layers learn more abstract, high-level features – increasing channels lets them capture a richer variety of patterns. At the same time, the combination of many lower-resolution features is needed to reconstruct details when upsampling.)

Mini-Challenge: The provided U-Net is a basic version. Try to **enhance the model** in one of the following ways: - **Depth:** Add an additional downsampling and upsampling pair (i.e., a 3rd encoder block and its corresponding decoder). This would create a larger U-Net that might capture features at an even more global scale (especially useful if input patches are 256×256 or larger). Keep in mind to adjust filter counts and skip connections accordingly. - **Activation functions:** Swap `ReLU` with `LeakyReLU` (with a small negative slope) in all conv layers. LeakyReLU can help if ReLU dead neurons are a concern. See if it affects training stability. - **Batch Normalization:** Insert `nn.BatchNorm2d` after conv layers (before ReLU) to normalize feature distributions. This can sometimes speed up training. - **Dropout:** Add a `nn.Dropout` layer in the bottleneck or decoder (for example, after the first conv in bottleneck or between decoder convs) to regularize the model and mitigate overfitting. Flood mapping often has limited training data, so dropout might help generalization. - **Final activation:** For inference convenience, you might incorporate a `nn.Sigmoid` at the end of `forward` (if not using `BCEWithLogitsLoss`). If you do this, remember to adjust the training loss accordingly (use plain BCE not logits version, or compute Dice on the sigmoid output).

Experiment with one or two of these modifications. Does the model size or changes impact the training speed or accuracy? This exercise will deepen your understanding of U-Net's design choices.

Module 4: Compile the Model and Train

With the U-Net defined, we move to the training phase. We need to set up a **loss function**, an **optimizer**, and loop over epochs to train the model on our dataset. We will also monitor the training and validation performance to ensure the model is learning and to avoid overfitting.

Loss Function: For segmentation, especially with class imbalance (usually far fewer flood pixels than background pixels), a common choice is **Dice loss**, **binary cross-entropy (BCE)**, or a combination of both. - **Binary Cross-Entropy (with logits):** Treat each pixel's prediction vs label as a binary classification, and take the average loss. This is a standard choice and is implemented in PyTorch as `nn.BCEWithLogitsLoss` (which applies a sigmoid internally). - **Dice Loss:** This is based on the Dice coefficient (overlap measure). Dice coefficient for sets A (prediction) and B (ground truth) is `2|A∩B|/(|A|+|B|)`. As a differentiable loss, we use `1 - DiceCoefficient` (since we want to minimize loss = maximize overlap). It effectively measures the overlap between predicted and true mask, and is particularly useful when the target class is sparse. Dice loss directly optimizes for overlap and is less sensitive to class imbalance (because if there are few positives, the overlap term still dominates relative to negative space). - **Combo Loss:** Many state-of-the-art approaches combine BCE (or focal loss, a variant of weighted BCE) with Dice loss, to get benefits of both [8]. For instance, `Loss = α * BCE + (1-α) * DiceLoss`. BCE ensures overall precision/recall are learned, while Dice focuses on the structure of the segmentation. In literature, adding Dice loss has been shown to improve flood segmentation results by addressing data imbalance [9].

For our initial training, we can use **BCEWithLogitsLoss** as a simple choice. Optionally, you can implement Dice loss and add it. Below, we'll illustrate using a combination: compute BCE loss and Dice loss each batch, and sum them (with equal weight for simplicity, α=0.5 each, or adjust as needed).

Optimizer: We'll use **Adam** optimizer (which usually performs well for CNNs) with a moderate learning rate (e.g., 1e-3). Adam adapts the learning rate per parameter and often converges faster than plain SGD.

Training Loop: We iterate for a number of epochs. In each epoch: - Switch model to `train` mode. - Loop over training batches: get images and masks, move to device (GPU if available), do `forward` pass to get outputs, compute loss, do `backward()` to compute gradients, and `optimizer.step()` to update weights. Don't forget `optimizer.zero_grad()` each iteration. - Optionally accumulate the training loss to monitor average. - After training epoch, switch model to `eval` mode and compute metrics on the **validation set** (or a hold-out part of training if no separate val). Compute the loss on val data (without gradient) and perhaps compute the Dice or IoU to see how well it's segmenting. Monitoring validation helps catch overfitting: if val loss starts increasing while train loss decreases, the model might be memorizing training data. - Print epoch summary: training loss, validation loss, and maybe validation IoU/F1.

Code Snippet for Training: Here is a skeletal training loop. (We assume `train_loader` and `val_loader` are PyTorch DataLoader objects prepared from our dataset.)

```
import torch.optim as optim

# Initialize model, loss, optimizer
model = UNet().to(device)
criterion_bce = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

num_epochs = 20
for epoch in range(1, num_epochs+1):
    model.train()
    running_loss = 0.0
    for images, masks in train_loader:
        images = images.to(device)            # shape [N,2,H,W]
        masks = masks.to(device)             # shape [N,H,W]
        masks = masks.unsqueeze(1)           # shape [N,1,H,W] for loss
compatibility

        # Forward + loss
        logits = model(images)              # output shape [N,1,H,W]
        loss = criterion_bce(logits, masks)
        # If using Dice loss as well:
        # pred_probs = torch.sigmoid(logits)
        # dice = 1 - (2 * (pred_probs * masks).sum() + 1) / ((pred_probs +
masks).sum() + 1)
        # loss = 0.5 * bce_loss + 0.5 * dice   # example combined loss

        # Backprop and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)  # accumulate sum of loss
    train_loss = running_loss / len(train_loader.dataset)

    # Validation
    model.eval()
    val_loss = 0.0
    val_intersection = 0.0
    val_union = 0.0
    with torch.no_grad():
        for images, masks in val_loader:
            images = images.to(device)
            masks = masks.to(device)
            masks = masks.unsqueeze(1)
            logits = model(images)
            # Compute validation loss
            val_loss += criterion_bce(logits, masks).item() * images.size(0)
            # Compute IoU components for validation
            preds = torch.sigmoid(logits)  # convert to probabilities 0-1
            pred_mask = (preds >= 0.5).float()  # threshold to 0/1
            # Intersection and Union for IoU
            intersection = (pred_mask * masks).sum().item()
            union = pred_mask.sum().item() + masks.sum().item() - intersection
            val_intersection += intersection
            val_union += union
    val_loss = val_loss / len(val_loader.dataset)
    val_iou = val_intersection / (val_union + 1e-8)
    print(f"Epoch {epoch}/{num_epochs} - Train Loss: {train_loss:.4f}  Val
Loss: {val_loss:.4f}  Val IoU: {val_iou:.4f}")
```

This code prints the training loss, validation loss, and validation IoU each epoch. Note: - We `unsqueeze(1)` on masks to make them shape `[N,1,H,W]` to match the output's shape for the loss function. - In the commented section, we show how you might compute Dice loss (`dice = 1 - 2*|pred∩mask|/(|pred|+|mask|)` with a smoothing of +1 in numerator and denominator to avoid division by zero). - We threshold predictions at 0.5 for IoU calculation on validation. IoU isn't directly used in training here (we're just monitoring it). - `device` should be defined (like `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`). Ensure model and data are on the same device.

Feel free to adjust `num_epochs`. If you see that by epoch 20 the model hasn't converged (loss still going down and IoU going up steadily), you can train for more epochs. Conversely, if it plateaus early, you might stop earlier.

Training Tips: - If the loss is not decreasing after a few epochs, try reducing the learning rate (e.g., 1e-4) or check if there's a bug (like forgetting to set model.train()). - Ensure the **mask values are 0/1** (and not 0/255 or something) before computing loss. If masks are boolean or 0-255, convert to float 0.0/1.0. - Using **Dice loss** can significantly help when flood pixels are sparse. You might observe training with combined Dice + BCE leads to better IoU in validation. (Dice loss essentially forces overlap maximization.) - Monitor not just IoU but also **Precision** and **Recall** on validation (we'll do that in Module 5). For instance, if IoU is low, checking precision/recall will tell if the model is over-predicting flood (low precision, many false positives) or under-predicting (low recall, missing floods). - Save the model with best validation IoU using `torch.save` if you plan to use it later or for inference on test data.

Think-Through: Why do we separate the data into training and validation sets? Couldn't we just look at training performance? Consider that a model might memorize training examples (especially if they are few). Validation performance tells us how well the model generalizes to unseen data. If training loss keeps dropping but validation loss stops improving or worsens, it's a sign of **overfitting**. We might then use strategies like early stopping or regularization. *Question:* If you find the model starts overfitting after epoch 15 (val loss rising), what could you do to mitigate it? (Hint: You could stop training early, use a smaller learning rate as training progresses, add more augmentation, or incorporate regularization like dropout.)

Additionally, think about the choice of **loss function**. We used BCE (and optionally Dice). If flood pixels are very rare, plain BCE might get overwhelmed by the majority class (background). Dice loss helps because it essentially normalizes by size of regions [9]. Another approach is **class weighting** in BCE (assign higher weight to flood class). Focal loss is a variant that down-weights easy negatives and focuses on hard pixels. In summary, the loss design can significantly affect training for imbalanced segmentation.

Mini-Challenge: Implement the **Dice loss** function yourself and experiment with it. For example, write a function `dice_loss(pred_probs, true_masks)` that computes the soft dice loss for a batch. Use it in training (either alone or combined with BCE). See if the convergence (in terms of IoU) improves. Another challenge: try using a **learning rate scheduler** from PyTorch (e.g., `ReduceLROnPlateau` to reduce LR when validation loss stagnates, or `StepLR` to decay LR every few epochs). Integrating a scheduler can yield better final performance. Observe how adjusting the learning rate during training impacts the model's learning curve.

Module 5: Evaluate Model Performance on Test Data

After training the model (hopefully to a point of good validation performance), it's time to evaluate it on the **test dataset**. This dataset contains SAR patches that the model has never seen – likely from different areas or times of the flood event – to truly assess how well the model generalizes in mapping floods.

We will use several **metrics** to evaluate segmentation quality: - **Intersection over Union (IoU)**, also known as the Jaccard Index. - **Precision** and **Recall** (also called Positive Predictive Value and Sensitivity). - **F1 Score**, the harmonic mean of precision and recall. - (And optionally Pixel Accuracy, though we will see why accuracy can be misleading.)

Definitions: Let's define: - **TP** (True Positives): number of pixels correctly predicted as flood. - **TN** (True Negatives): number of pixels correctly predicted as non-flood. - **FP** (False Positives): number of pixels predicted as flood but actually non-flood. - **FN** (False Negatives): number of pixels predicted as non-flood but actually flood.

Using these: - **Precision** = TP / (TP + FP) – Of the pixels we predicted as flood, what fraction were actually flood? High precision means few false alarms. - **Recall** = TP / (TP + FN) – Of the actual flooded pixels, what fraction did we correctly detect? High recall means we missed very little (few false negatives). - **F1 Score** = 2 * Precision * Recall / (Precision + Recall) – This combines precision and recall into one number (it's the harmonic mean). It's 1.0 if precision and recall are both perfect, and declines if either is low. - **IoU (Jaccard)** = TP / (TP + FP + FN) – This directly measures overlap between the predicted flood area and true flood area. It can also be seen as `TP / (union of predicted and true flood areas)`. IoU of 1 means prediction exactly matches truth; IoU of 0 means no overlap at all.

These metrics focus on the **flood class** (since that's the positive class). We usually don't emphasize TN or overall accuracy in flood mapping because the background class dominates. For example, if only 5% of pixels are flood, a trivial model that labels everything "non-flood" gets 95% accuracy – but is useless. That's why we use metrics like precision, recall, F1, IoU which concentrate on the positive (flood) detection performance [10] .

Evaluate on test set: We will run the model on each test patch, apply a threshold (say 0.5) on the sigmoid output to get predicted mask, then count TP, FP, FN. Summing over the whole test set, we compute the above metrics. It's often helpful to compute metrics per-patch as well, to see variability (but we'll at least get the aggregate).

Here's a code example to evaluate metrics on a `test_loader`:

```
model.eval()
total_tp = total_fp = total_fn = 0

for images, masks in test_loader:
    images = images.to(device)
    masks = masks.to(device)  # shape [N,H,W]
    masks = (masks > 0.5).float()   # ensure binary 0/1
    with torch.no_grad():
        logits = model(images.to(device))
        preds = torch.sigmoid(logits)
    pred_mask = (preds >= 0.5).float()
    # Calculate TP, FP, FN for this batch
    tp = (pred_mask * masks.unsqueeze(1)).sum().item()
    fp = (pred_mask * (1 - masks.unsqueeze(1))).sum().item()
    fn = ((1 - pred_mask) * masks.unsqueeze(1)).sum().item()
    total_tp += tp
    total_fp += fp
    total_fn += fn
```

```
# Now compute metrics
precision = total_tp / (total_tp + total_fp + 1e-8)
recall = total_tp / (total_tp + total_fn + 1e-8)
f1 = 2 * precision * recall / (precision + recall + 1e-8)
iou = total_tp / (total_tp + total_fp + total_fn + 1e-8)

print(f"Test Precision: {precision:.3f}, Recall: {recall:.3f}, F1: {f1:.3f},
IoU: {iou:.3f}")
```

(We add a small epsilon 1e-8 to denominators to avoid division-by-zero issues in case of no positives.)

This will output the overall Precision, Recall, F1, IoU on the test set. We hope to see values fairly high (e.g., IoU might be somewhat lower since it's strict, but F1 could be > 0.8 if model is good, etc.). The exact numbers depend on the difficulty of the dataset.

After computing, interpret the results: - If **Precision** is significantly higher than Recall, the model is conservative – it only marks something as flood when it's very sure (few false positives, but potentially more false negatives). This might be acceptable if you want high confidence flood maps but it means some flooded pixels were missed. - If **Recall** is higher than Precision, the model is aggressive – it finds most floods (few misses) but also flags more non-flood areas as flood (more false alarms). In disaster response, missing floods (low recall) might be worse than some false alarms, but there's a balance. - **F1** provides a single measure of balance between precision and recall. **IoU** is a stricter measure than F1 (for instance, an F1 of 0.8 corresponds roughly to IoU of ~0.67). IoU is often used as the primary metric in segmentation challenges.

Many publications report **IoU or F1** as the key metric for flood segmentation. For context, IoUs for flood mapping can vary – a model might achieve IoU 0.6–0.8 on a challenging test set (which is decent given noise and label uncertainty) [11] [12]. An IoU above 0.8 is quite good and suggests very high agreement with ground truth, whereas an IoU below 0.4 indicates the model is struggling.

Understanding metric significance: Precision and recall can hint at the type of errors: - Low Precision, High Recall: Many false positives, model over-predicts flood (maybe confused by other dark features like shadows or water-like surfaces). - High Precision, Low Recall: Model under-predicts, maybe only capturing the core flood areas and missing edges or under vegetation. - Both Low: The model might be misclassifying a lot – possibly learned wrong features or dataset was very challenging. - Both High: The model is performing well.

Think-Through: Why is **pixel accuracy** not a reliable metric here? Imagine only 1% of pixels are flood. If the model classifies all pixels as non-flood, accuracy = 99% (because it got all the background right) [10]. But recall = 0 (it missed all floods). This illustrates that accuracy can be very high even when the model fails the task that matters. We focus on metrics that specifically evaluate the class of interest (flood). *Question:* If you had to prioritize one – would you aim for higher recall or higher precision in flood mapping? (This might depend on application: for emergency response, missing a flooded area could be critical – so many prefer high recall, ensuring all flooded regions are found, even if it means some false alarms that can be later checked. On the other hand, in damage assessment, false positives might waste resources, so precision can be important. Ideally, F1 or IoU combines both aspects.)

Mini-Challenge: Extend the evaluation to output **per-patch metrics**. Modify the code to compute precision, recall, IoU for each patch in the test set and then you can examine the distribution of scores. Are there certain patches where the model performed poorly (e.g., maybe patches with heavy forest or urban areas)? This can give insight into failure modes. Another challenge: create a simple **confusion matrix** for pixels (a 2x2 matrix of [[TN, FP],[FN, TP]] counts) and print it. This can sometimes be more tangible for understanding, e.g., "Model mislabeled X non-flood pixels as flood (FP) and missed Y flood pixels (FN) out of Z total flood pixels."

Finally, ensure you save the model if you want to reuse it. You can do: `torch.save(model.state_dict(), "unet_flood_model.pth")` after training. Then in future, load with `model.load_state_dict(torch.load("unet_flood_model.pth"))` and run predictions.

Module 6: Visualize Predictions on SAR Images

Quantitative metrics are important, but **visualizing the model's predictions** gives a visceral sense of its performance. In this module, we will take some test patches and overlay the predicted flood mask on the SAR image. This visual check can highlight if the model's flood extents align with the SAR features (usually dark regions for water) and where it might be going wrong (e.g., false positives on dark non-water areas).

Preparing images for display: Sentinel-1 SAR images are essentially grayscale (one channel per polarization). To visualize, we might take the VV channel (or a composite) as a grayscale image. The predicted mask is binary. Overlaying means we will color the mask and superimpose it with some transparency on the SAR image.

A common approach: - Display the SAR image in grayscale. - Overlay the flood mask in a color (say red or blue) with an alpha (transparency) so you can see underlying SAR. - Alternatively, create an RGB image where the SAR is in the gray channel and flood pixels highlighted in a color.

We can use Matplotlib for quick visualization. Here's an example for one patch:

```
import matplotlib.pyplot as plt

# Get one batch of test data (or one sample)
images, masks = next(iter(test_loader))
model.eval()
with torch.no_grad():
    preds = torch.sigmoid(model(images.to(device)))
pred_mask = (preds.cpu() >= 0.5).numpy().astype('uint8')  # binary mask, shape
(N,1,H,W)

# Visualize the first sample in the batch
sar_vv = images[0, 0].numpy()        # VV channel of the first sample
true_mask = masks[0].numpy()
pred_mask0 = pred_mask[0, 0]

plt.figure(figsize=(10,4))
```

```
# Left: SAR with true mask overlay
plt.subplot(1,2,1)
plt.title("SAR + True Flood Mask")
plt.imshow(sar_vv, cmap='gray')
# overlay true mask in green
plt.imshow(true_mask, cmap='Greens', alpha=0.3)
# Right: SAR with predicted mask overlay
plt.subplot(1,2,2)
plt.title("SAR + Predicted Flood Mask")
plt.imshow(sar_vv, cmap='gray')
plt.imshow(pred_mask0, cmap='Reds', alpha=0.3)
plt.show()
```

In this code: - We take the first image's VV polarization for visualization. (You could also combine VV and VH for display, e.g., as two different grayscale images or an RGB composite, but using one is fine to see water.) - The true mask is overlaid in semi-transparent green, predicted in red. The choice of color is arbitrary; the idea is to differentiate flood areas from the gray background. - The result shows side-by-side the ground truth vs model prediction. Green areas (left) are actual floods, red (right) are model's floods. Ideally, these overlap a lot (yellowish if overlapping green/red, if we had them on one image).

When you display these, inspect: - Are the predicted flood areas generally where the SAR is dark (flood-prone areas)? If yes, the model learned the relationship. - Do you see red patches in places with bright SAR returns (which likely aren't water)? Those would be false positives. - Are there green areas with no red on the right (false negatives)? Where are those? Perhaps under heavy vegetation or near bright features that confused the model. - If possible, look at multiple examples to see consistency.

Visualizing an overlay on the original SAR image also helps communicate results to others. It's easier to explain "red highlight shows where the model thinks flooding is, on top of the SAR image" rather than just looking at a binary mask alone.

Conceptual Hurdle (EO context): SAR images have speckle and textured terrain, which can make interpretation tough. The U-Net's ability to integrate multi-scale features means it can ignore some pixel-level noise while focusing on the larger shape of water bodies. The **skip connections** ensure that the model's output aligns well with actual image features: e.g., the boundary of a river in the SAR image might be a sharp line between dark (water) and bright (land). The encoder will capture the presence of a water region, and through skip connections, the decoder can get the exact boundary from the earlier layer to draw the flood outline accurately [7]. This is how U-Net delineates floods despite the "salt-and-pepper" noise in SAR – it learns to average out noise in deeper layers, but uses the fine details from early layers to place boundaries correctly.

Think-Through: When overlaying predictions, why might we prefer the SAR background over, say, an optical image or just showing the mask alone? The reason is that the SAR background gives context – we can see rivers, urban areas, etc., and verify if the model's predicted floods make sense in context. Also, an optical image might not even be available at flood time due to clouds. By using the SAR image itself, we see exactly what the model saw. *Question:* If you notice some systematic errors (e.g., the model always thinks a certain radar-dark feature like a shadow is water), how could you refine the model or input to address that? (Hint: Perhaps providing additional inputs like a DEM or doing temporal change detection could help. Or using context – e.g., if certain areas are permanently water or permanently non-water, that info can be fused. In fact, some advanced models incorporate elevation or historical water maps as additional channels [13] [14].)

Mini-Challenge: Pick 2–3 random test patches and create a small **gallery of results**. For each patch, plot: (a) SAR image, (b) true mask, (c) predicted mask. You can do this in a 3-column subplot for each patch. This gives a qualitative summary of performance. Another idea: if geo-referenced data is available, you could try to overlay the flood mask on a map or satellite base layer (this is more advanced and typically done in a GIS, but conceptually interesting). Finally, consider using different colormaps or opacities to make a nice visualization – perhaps the predicted mask in semi-transparent blue over SAR for a report graphic.

By completing these modules, you have implemented a full pipeline for **SAR flood mapping with U-Net** – from data loading and augmentation, through model building, training, and evaluation, to visualization of results. This workflow is powerful for many Earth Observation tasks, especially in disaster management where rapid, automated mapping is needed. You've also learned how U-Net's architecture (especially those skip connections) helps tackle the challenges of SAR data (noise and resolution) to produce accurate flood extent maps [15] [7]. Great job on reaching this point, and feel free to experiment further with the model or try it on new events!

---

[1] (PDF) REMOTE SENSING AND GOOGLE EARTH ENGINE FOR RAPID FLOOD MAPPING AND DAMAGE ASSESSMENT: A CASE OF
AND VAMCO (ULYSSES)
https://www.researchgate.net/publication/
380142998_REMOTE_SENSING_AND_GOOGLE_EARTH_ENGINE_FOR_RAPID_FLOOD_MAPPING_AND_DAMAGE_ASSESSMENT_A_CASE_OF_TYPHOON_GONI

[2] Typhoon Noru - Wikipedia
https://en.wikipedia.org/wiki/Typhoon_Noru

[3] [6] [7] [8] [9] [13] [14] Frontiers | Deep attentive fusion network for flood detection on uni-temporal Sentinel-1 data
https://www.frontiersin.org/journals/remote-sensing/articles/10.3389/frsen.2022.1060144/full

[4] [10] [11] [12] [15] ntrs.nasa.gov
https://ntrs.nasa.gov/api/citations/20220000062/downloads/Accepted_final.pdf

[5] Sen1Floods11: A Georeferenced Dataset to Train and Test Deep Learning Flood Algorithms for Sentinel-1
https://openaccess.thecvf.com/content_CVPRW_2020/papers/w11/
Bonafilia_Sen1Floods11_A_Georeferenced_Dataset_to_Train_and_Test_Deep_Learning_CVPRW_2020_paper.pdf