ChatGPT

Day 3: Advanced Deep Learning – Semantic Segmentation & Object Detection

Session 1: Semantic Segmentation with U-Net for Earth Observation (1.5 hours)

Concept of Semantic Segmentation

Semantic segmentation is the task of classifying each pixel in an image into a class label, effectively producing a **pixel-wise labeled map**. This is different from image classification (which assigns one label to an entire image or region) and object detection (which draws bounding boxes around individual objects) [1] [2]. In segmentation, the output is an image where every pixel is labeled (e.g. water, building, forest), allowing precise delineation of different land cover types. This level of detail is especially useful in geospatial contexts for creating thematic maps and analyzing spatial patterns. For example, whereas an image classification might tell us an entire satellite patch is “urban” or “agriculture,” **semantic segmentation** can highlight exactly which pixels are buildings, roads, vegetation, water, etc., giving a much richer understanding of the scene [1]. This pixel-level approach is crucial for Earth Observation tasks like mapping floods, land cover, or burn scars, where knowing the exact extent and shape of features matters.

U-Net Architecture for Segmentation

The **U-Net** is a popular convolutional neural network architecture designed for semantic segmentation, originally developed for biomedical images but now widely used in Earth Observation as well. It has a characteristic U-shape with two main components [3]:

- **Encoder (Contracting Path):** A series of convolutional layers and downsampling (via pooling) that progressively reduces the spatial resolution of the feature maps while learning higher-level features. As we move down the encoder, image details are compressed and abstracted, capturing the *context* of what is in the image. (Recall from Day 2: concepts like convolution, activation, and pooling – with techniques like padding to control output size – all apply here. In U-Net, often *same padding* is used so that feature maps align when combining encoder and decoder outputs.)
- **Decoder (Expansive Path):** A series of upsampling or transposed convolution layers (sometimes called deconvolutions) that gradually restore the spatial resolution, using the encoded features to construct a pixel-wise prediction map [3]. This upsampling mirrors the encoder’s steps in reverse, enabling *precise localization* of features (recall: upsampling can be done via learned transposed conv filters or interpolation followed by conv layers, as introduced in Day 2).

Illustration of the U-Net architecture. The network has an encoder (left down-sampling path) that compresses the image into abstract features, and a decoder (right up-sampling path) that reconstructs a segmentation mask at full image resolution. Crucially, **skip connections** (horizontal arrows) copy feature maps from the encoder to the decoder at corresponding levels, providing high-resolution details to the decoder for more precise outputs. (Image source: Ronneberger *et al.*, 2015)

A key innovation of U-Net is these **skip connections** linking matching encoder and decoder layers [4]. The feature maps from the encoder (which contain fine-grained spatial details from earlier layers) are concatenated with the upsampled features in the decoder. This allows the model to “skip over” the bottleneck and directly inject high-resolution context into the decoding process [4]. The result is improved detail and accuracy in segmentation outputs, since the decoder doesn’t have to rely solely on the coarse feature maps after upsampling – it can leverage the original fine details as well. These skip connections help preserve edges and small structures (e.g., the exact boundary of a flooded area or building) that might otherwise be lost during downsampling [5]. The central part of U-Net is the **bottleneck layer** (the bottom of the “U”), where the feature representation is most compressed. This is where the network holds a condensed encoding of the image’s content (maximum context, minimum spatial detail) before the decoder begins expanding it. Notably, U-Net implementations must handle the alignment of feature map sizes for concatenation – often using appropriate padding on convolutions so that each encoder output matches the size of the corresponding decoder feature map. By combining encoder and decoder features, U-Net captures both the *what* (context) and the *where* (location) for each class in the image.

Recap of Key CNN Concepts: U-Net uses the same building blocks introduced earlier (convolutions, ReLU activations, pooling, etc.). Participants should recall the role of **padding** in convolutions: padding helps preserve spatial dimensions through layers (e.g. “same” padding keeps width and height constant), which U-Net often uses to ensure that skip-connected feature maps align in size. Similarly, **upsampling** (through transpose convolutions) is essentially the inverse of pooling – it increases spatial dimensions, allowing the network to produce an output mask the same size as the input image. We briefly revisit these concepts so that participants recognize them inside the U-Net architecture rather than seeing them as entirely new operations.

Applications of U-Net in Earth Observation

U-Net has become a go-to architecture for many EO segmentation tasks due to its accuracy and efficiency in learning from limited data. Some notable applications include:

- **Flood Mapping:** Identifying water inundation extent from satellite imagery (optical or SAR). U-Net has been used to segment flooded areas in Sentinel-1 SAR images and Sentinel-2 optical images with high accuracy, proving invaluable for disaster response [6]. For example, during floods, a U-Net can produce a binary map of floodwater vs. non-flood for each pixel, enabling rapid assessment of flood extent.
- **Land Cover Mapping:** Creating detailed land use/land cover maps (forest, agriculture, urban, water, etc.) from high-resolution imagery. U-Net’s ability to preserve fine details helps delineate boundaries between different land cover types (like the exact shape of urban districts or small water bodies) [6].
- **Road and Infrastructure Extraction:** Segmenting road networks or railways from aerial images or SAR. U-Net can trace continuous structures like roads by learning the linear patterns, useful for map updating and transportation planning.
- **Building Footprint Delineation:** Extracting buildings from overhead images to produce building footprint maps. With appropriate high-res data, U-Net can outline individual buildings or dense informal settlements, aiding urban planning and risk assessment. (In practice, variants like Residual U-Net or Attention U-Net are also popular for building segmentation, but the core idea remains an encoder-decoder with skips.)
- **Vegetation and Crop Segmentation:** Identifying crop fields, forests, or tree cover at the pixel level for agricultural monitoring or forestry management.

Across these examples, U-Net's strength is in combining the broad **context** (e.g., distinguishing an urban area from a forest in general) with **precise boundaries** (e.g., exactly where the forest stops and urban area begins). Research and practice have shown U-Net achieves high segmentation accuracy in remote sensing, often outperforming older pixel-based or patch-based methods [6]. It has even been demonstrated that U-Net can achieve robust results with relatively small training datasets, thanks to the efficiency of the architecture (one reason it was originally successful in medical imaging with limited training images [7]). We will leverage these strengths in our hands-on flood mapping exercise.

Loss Functions for Segmentation

Training a segmentation model requires choosing an appropriate **loss function** that compares the predicted mask to the ground truth mask. Several loss functions are common in segmentation, each with different strengths:

- **Pixel-wise Cross-Entropy Loss:** This is the standard approach for multi-class segmentation (or binary cross-entropy for two classes). It treats each pixel's classification as an independent prediction and penalizes the difference between predicted class probability and true class (using the negative log-likelihood). Cross-entropy is effective and straightforward, providing strong gradients for learning. However, in segmentation tasks with class imbalance (e.g., far more background pixels than flood pixels), vanilla cross-entropy can be dominated by the majority class. A common remedy is to use **weighted cross-entropy**, assigning higher weight to under-represented classes so the model pays attention to them [8]. Cross-entropy focuses on pixel-level accuracy but does not directly ensure good overlap or boundary alignment – that's where overlap-based losses come in.
- **Dice Loss (F1 score loss):** Dice loss measures the overlap between the predicted mask and the true mask, essentially **2 * (intersection) / (sum of areas)**. Training the network to maximize Dice (or minimize *1 – Dice*) encourages it to get the segmentation overlap as high as possible. Dice loss is particularly well-suited for imbalanced datasets because it focuses on the *relative* overlap; correctly segmenting that one small flooded patch contributes as much to Dice as some large region of non-flood [9]. It's known to be effective when the target objects occupy a small fraction of the image or when false negatives and false positives need to be weighted equally. In other words, Dice loss helps the model *not* ignore small structures or minority classes [10] [11]. (Many medical image segmentation models, for instance, use Dice loss to segment tumors that occupy only a tiny area.)
- **Jaccard Loss (Intersection over Union, IoU loss):** Very similar to Dice, IoU loss directly optimizes the IoU metric (intersection divided by union of prediction and truth). It's also robust to class imbalance and has an interpretation closely tied to segmentation quality – it penalizes false positives and false negatives at the region level. IoU (or Jaccard) loss is often mentioned for improving boundary accuracy, since maximizing IoU requires the predicted region to align well with the true region boundaries [12] [13]. This makes it useful in applications like geographic mapping where boundary delineation is crucial [14].
- **Combined Losses:** In practice, you don't have to pick just one. Many implementations use a combination (e.g., a weighted sum of cross-entropy and Dice losses) to get the benefits of both – cross-entropy ensuring overall pixel-wise correctness and Dice focusing on overlap and balance [15]. For example, a **Dice + Cross-Entropy** loss can improve both per-pixel accuracy and overall region accuracy. Some also use **Focal Loss** (a modified cross-entropy that down-weights easy/background examples) especially for extremely imbalanced cases, though that's more common in object detection.

Why these losses? In segmentation of EO data, class imbalance is a typical issue (think of mapping floods: the flooded pixels are usually far fewer than non-flooded). Losses like Dice are naturally robust in such cases [9]. Meanwhile, accurately capturing boundaries (say the edge of a flood or the outline of a building) is often vital – IoU-based loss directly rewards aligning shapes [12]. We will discuss when to use each loss. For our flood mapping case (binary segmentation with imbalance), we might choose a Dice loss or a combo (Dice + Cross-Entropy) to handle the imbalance and get sharp boundaries. The goal is to demonstrate to participants that the choice of loss can significantly affect model training: a poor choice might lead the model to predict all pixels as the majority class (trivial but useless result), whereas the right loss will push it to get the minority class right as well [8]. We highlight these considerations so that participants learn not just to accept the default loss, but to think about the nature of their segmentation problem and pick (or tune) a loss accordingly for the best results.

Session 2: *Hands-on:* Flood Mapping with U-Net on Sentinel-1 SAR (DRR Focus) – 2.5 hours

Case Study Overview: Flood Mapping in Central Luzon, Philippines

In this session, we put theory into practice by tackling a **disaster risk reduction (DRR)** use-case: mapping flood inundation extents from satellite imagery. The case study centers on the **Pampanga River Basin** in Central Luzon, a region prone to severe flooding. We will focus on a specific flood event caused by a major typhoon (e.g., Typhoon *Ulysses* in November 2020, locally known as Vamco). Typhoon Ulysses brought torrential rains that led to extensive flooding in Central Luzon, affecting millions of people and causing billions of pesos in damage [16]. In particular, the Pampanga River and its tributaries overflowed, submerging large areas of Pampanga and Bulacan provinces. Rapid and accurate flood mapping in such scenarios is crucial for emergency response and damage assessment.

Our goal is to use **Sentinel-1 SAR** data and a U-Net model to automatically delineate flooded areas, as a step toward faster disaster response. SAR (Synthetic Aperture Radar) is invaluable for flood mapping because it penetrates cloud cover and provides day-night imaging – critical since floods often occur during storms with clouds. We will use Sentinel-1's dual polarization (VV and VH) data, which provides two channels of information that can help distinguish water from land. This case exemplifies how advanced deep learning (U-Net) can enhance traditional remote sensing workflows for DRR by extracting pixel-level information (flood extent) from raw imagery.

Relevance to DRR: Flood maps derived from SAR help authorities identify which communities are underwater and how the flood is spreading, often well before field reports can paint a full picture. In a country like the Philippines, where typhoons and monsoon floods are regular threats, automating flood extent extraction means more timely evacuations, targeted relief, and better recovery planning. By focusing on a real event in the Philippines, participants can appreciate the real-world impact of the technology they're learning.

Data and Preparation (SAR Imagery & Ground Truth)

Working with raw SAR data can be complex and time-consuming, so we have **prepared the dataset** in advance to be “analysis-ready.” This allows us to concentrate on building and training the model, rather than on image preprocessing quirks. The dataset consists of:

- **Sentinel-1 SAR image patches** (grayscale images, 128×128 or 256×256 pixels) covering various parts of the Pampanga River basin during the flood. Each patch has two bands (VV and VH polarizations), so effectively it’s like a 2-channel image. These have already undergone the necessary preprocessing: **radiometric calibration** (to convert raw SAR signal to backscatter values), **speckle filtering** (to reduce SAR’s grainy noise while preserving features), and **geometric terrain correction** (orthorectification to remove distortion from topography and sensor angle). These steps are essential – e.g., radiometric calibration ensures pixel values truly represent radar backscatter so that water and land are distinguishable, and terrain correction ensures the flood map overlays correctly on a map. We highlight to participants that preparing SAR data *is* a significant effort in practice: one must apply orbit files, remove thermal noise, filter speckle, and use a DEM for terrain correction [17] [18]. By providing preprocessed patches, we save time and focus on the deep learning aspect.
- **Binary flood mask patches** (ground truth labels) corresponding to each SAR patch. In these masks, a pixel value of 1 indicates **flooded** area and 0 indicates **non-flooded** (dry land or normal water bodies). These masks were derived from authoritative sources or expert annotation for the chosen event (for example, through manual analysis of SAR and validation with reports on which areas were flooded). Generating accurate flood masks is itself challenging – it might involve thresholding the SAR, using optical imagery when available, and a lot of cleaning by analysts. We note that this ground truth creation is a critical step: a machine learning model can only be as good as the training labels. In real deployments, building such a labeled dataset requires significant domain knowledge and effort.

Pitfall Highlight – Data Preparation: We will stress that one often spends **much more time** preparing remote sensing data than actually training the model. For SAR in particular, raw data can’t be fed to a CNN without correction and noise reduction. Misclassifications can easily arise if, say, an area looks dark in SAR not because it’s water but because of a sensor artifact – hence the need for careful calibration. Similarly, creating ground truth for floods may involve days of expert work per event (sometimes drawing polygons around floods on imagery or using algorithms plus manual fixes). Participants should understand that while our session gives them “clean” numpy arrays to plug into a network, a lot of unseen groundwork underpins that. This is a reality of EO AI projects: **“80% of the work is data cleaning”**. By providing prepared data, we enable them to get hands-on with the model within a limited time, but in professional settings they must budget time for these steps.

Workflow Steps

We will walk through a structured workflow to implement and evaluate the flood mapping model. The process is broken down into clear steps, which we will execute in a Google Colab notebook (with GPU enabled for faster training):

1. **Load Data:** Begin by loading the SAR image patches (as input features) and their corresponding binary flood masks (as labels). We’ll verify dimensions (e.g., each image is 128×128 with 2 channels, and each mask is 128×128 with 1 channel) and perhaps visualize a few examples. It’s insightful to show a SAR patch and its flood mask overlay, so participants see what the model is trying to learn

(e.g., the SAR might show a dark meandering shape along a river – indicating low backscatter from smooth water – and the mask highlights that region as flooded). This also ensures everyone understands the data format.

2. **Data Augmentation (if time permits):** We'll discuss augmentation techniques to synthetically expand the training data. Simple flips or rotations can be applied to SAR images and masks (taking care to apply the same transform to both). SAR-specific augmentation might include slight intensity scaling or adding noise, but to keep things simple we might stick to geometric transforms. Augmentation can help the model generalize better – for instance, a rotation might help it handle river flooding patterns from different orientations. We'll implement a couple of augmentations and possibly let participants decide if they want to use them (e.g., randomly flip horizontally/vertically). This step is optional depending on time, but it's good practice to mention it.

3. **Define the U-Net Model:** Using the chosen deep learning framework (whether PyTorch or TensorFlow/Keras, consistent with Day 2's usage), we'll construct the U-Net architecture. This involves defining convolutional blocks for the encoder, downsampling (pooling) layers, upsampling layers for the decoder (we might use conv transpose or upsampling + conv), and the skip connections that concatenate encoder features to decoder features at the same resolution. We'll ensure the final layer is a 1-channel output with a sigmoid activation (for binary segmentation of flood vs non-flood). Coding this from scratch helps participants see the inner workings (e.g., how the tensor dimensions halve then double, how concatenation works). We'll also pay attention to **padding** here – if using PyTorch, for example, we might use `padding=1` on conv layers to keep sizes consistent, or explicitly handle cropping as original U-Net did. This is a chance to briefly reinforce how padding affects output size (a key idea from earlier sessions).

4. **Compile Model with Loss and Optimizer:** We will choose an appropriate loss function for training – likely **Dice loss** or a **Dice + Cross-Entropy combo** – because flood pixels are relatively sparse compared to background. We'll explain our choice: e.g., *"Dice loss will help because only a small fraction of pixels are floods, so we want the model to focus on getting those right"*. If using TensorFlow, we'll specify this loss and perhaps metrics like IoU or F1-score to monitor. If using PyTorch, we'll implement the loss and prepare an optimizer (e.g., Adam) and learning rate. We'll also set aside a validation set of patch data to monitor performance during training (ensuring we check for overfitting).

5. **Train the U-Net:** We'll train the model on the prepared dataset, using the training set patches. As the model trains (likely for a modest number of epochs given time constraints and dataset size), we'll watch the training vs validation loss curves and metric scores. Participants will see the model gradually learn to segment floods. We'll encourage them to note if the validation Dice/IoU improves and if it plateaus or diverges. We'll discuss any challenges that arise (for instance, if it's not learning well initially, maybe the learning rate is too high, etc., though with a prepared example it should converge reasonably). This hands-on training solidifies how the theory translates to actual model fitting.

6. **Evaluate Performance:** Once trained, we evaluate the model on a **test set** of patches (images it hasn't seen before). We will compute metrics like **Intersection over Union (IoU)**, **F1-score (Dice)**, **precision**, and **recall** for the flood class. For example, IoU tells us what fraction of the area of true flood plus predicted flood was correctly overlapped [13], and F1-score tells the harmonic mean of

precision and recall (how well we balance false positives vs false negatives). We'll interpret these: *"The model achieved an IoU of 0.75 on the test data, meaning it correctly overlaps 75% of the flooded area, which is quite good for satellite-based mapping* [19] *. Precision is X, recall is Y, indicating it (for example) sometimes over-predicts water in some areas (false positives) but catches most of the true floods."* We'll tie this back to the loss function choice too (Dice optimization tends to maximize F1-score). If available, we might compare to a baseline (say, a thresholding approach or a simpler CNN) to appreciate the improvement deep learning provides.

7. **Visualize Predictions:** Finally, the most rewarding part – we will visualize some of the U-Net's predicted flood maps on the test patches. We can overlay the predicted flood mask on the SAR image (perhaps coloring predicted flooded pixels in blue on top of the grayscale SAR). This gives a qualitative feel for performance: participants can see, for example, that the model nicely detected the flooded river spilling onto nearby fields, or it marked a certain dark region as flood which corresponds to known flood water. We'll also check any obvious mistakes visually – maybe the model thought a dark forest patch (low SAR return) was water and falsely labeled it. These visuals help connect the numeric metrics to reality and are great for discussion. Participants can relate them to what they learned: *"Notice how the U-Net's skip connections allowed it to outline the flood boundary along this river very closely – that's the spatial detail preservation at work!"*.

Through these steps, participants get end-to-end exposure: data → model → training → validation → results. We will encourage questions at each step, such as why certain parameters were chosen or what might be different if this were a multi-class segmentation (e.g., mapping water, urban, vegetation simultaneously).

Key Concepts and Hurdles for EO Users

Understanding SAR Data vs Optical: One conceptual hurdle is appreciating that a CNN like U-Net can be applied to SAR images *despite* SAR's peculiarities (speckle noise, grayscale intensity, etc.). We will explain that convolutional networks don't inherently care if an image "looks like a photograph" – they find patterns in the pixel intensities. In SAR, water typically appears very dark (low backscatter) because it's smooth and reflects signal away from the sensor, whereas rough land or urban surfaces appear brighter. The U-Net can learn these patterns (dark = water) even if SAR images are not visually intuitive to us. We reassure participants that the same Conv/Pooling operations they learned work on SAR data since it's just arrays of numbers. We might mention that using both VV and VH polarizations gives the network more information (some floods show up better in one polarization than the other, and having both can improve robustness).

Role of U-Net Architecture in Delineating Floods: We tie back the architecture to the task: the encoder learns to distinguish flooded vs non-flooded textures/contexts (e.g., floodwater often occurs in flat regions near rivers, which the network can contextualize). The decoder, with its skip connections, uses high-resolution features to precisely mark *where* the flood is. For instance, skip connections deliver the fine details of the riverbank outlines from the encoder's early layers, so the decoder can draw the flood extent with pixel-level accuracy. We highlight: if we only had a coarse decoder without skips, the flood outline might come out blob-like or miss thin flooded channels. The skip connections ensure that the model knows the exact original layout of features while labeling them [5] . This is particularly important in flood mapping, where the difference between a flood overtopping a road or not could be just a few pixels difference. U-Net's design is well-suited to capture such nuances.

Practical Insight – Generalization: We will discuss how the model might generalize to other flood events or areas. Does it learn something general about water in SAR, or is it mostly memorizing this specific event's characteristics? Likely, with enough training data diversity, it can generalize, but if one only trained on Pampanga 2020 flood, applying to a different region's flood might need fine-tuning. We mention this to temper expectations and reinforce the need for diverse training data if we want a broadly applicable model.

By the end of Session 2, participants will have **trained a deep learning model on real EO data** and understood not just the mechanics, but also the interpretation of results. They see the full pipeline from satellite images to a useful information product (flood map), which is a powerful takeaway. It also serves as a template for other segmentation tasks they might encounter (like those mentioned in Session 1's applications).

Session 3: Object Detection Techniques for EO Imagery (1.5 hours)

Concept of Object Detection

Object detection is about **finding and classifying multiple objects in an image**. In contrast to image classification (one label per image) or semantic segmentation (label every pixel), object detection must both identify *what* objects are present and *where* they are located in the image [2]. Typically this is achieved by predicting a **bounding box** (an x,y,width,height rectangle) around each object along with a class label. For example, in a satellite image, an object detector could locate and label all the cars in a parking lot or all the ships in a harbor. This task is essentially a combination of localization and classification.

We will clarify the hierarchy of vision tasks for participants: - *Image Classification:* "Is there a ship in this image or not?" (No location, just presence/absence or overall category). - *Object Detection:* "There is a ship at these coordinates, and another ship over there." (Each instance is localized by a box). - *Semantic Segmentation:* "This specific pixel is ship or water," producing a mask (no distinct instance separation, but precise outline). - *Instance Segmentation:* (mentioned for completeness) is like a combination of detection and segmentation – e.g., outline each individual ship. (Mask R-CNN does this, as we'll mention later.)

For our purposes, **object detection** will refer to the classic bounding-box detection of instances. It's critical in geospatial applications when we need to count objects or pinpoint their locations (e.g., count how many buildings in an area, find coordinates of all airplanes in an airfield, etc.).

We'll illustrate with a simple example: imagine a high-resolution image of a port – classification might just say "port" vs "non-port"; segmentation might color water vs land vs ships in different colors (merging all ships into one category region); but object detection will draw a box around each ship and perhaps label each as "ship." This distinction helps participants understand why and when we use detection. It also helps avoid confusion as we proceed to architectures: the output of a detector is not a full mask, but a list of boxes with class scores.

Overview of Popular Detection Architectures

Object detection has been a hot research area, and a few **families of architectures** have become prominent. We will give a high-level overview of these, focusing on the intuition of how they work rather than deep technical details (given time constraints). The main categories are:

- **Two-Stage Detectors:** These models break the task into two steps. First, they generate **region proposals** – rough candidate regions in the image that might contain an object. Next, they classify each proposed region into object categories (or background) and refine the bounding box. The seminal examples are the R-CNN family:
- **R-CNN (2014):** Extract ~2000 region proposals using an algorithm (Selective Search), then run a CNN on each and classify with an SVM. This was slow but a proof of concept.
- **Fast R-CNN:** Improved R-CNN by running the CNN on the whole image once (to get a feature map) and then sampling features for each proposal (ROI Pooling), instead of CNN per proposal – much faster.
- **Faster R-CNN:** Introduced a small integrated neural network (Region Proposal Network, RPN) to predict proposals, effectively learning where to look for objects, and made the system end-to-end trainable [20] [21].
- **Mask R-CNN:** Extends Faster R-CNN by adding a third branch that outputs a pixel mask for each detected object (instance segmentation). Worth mentioning as an evolution, though it's beyond just detection.

Two-stage detectors are known for **high accuracy** and good handling of difficult scenarios (small or overlapping objects) because that second stage can scrutinize each candidate region carefully [22]. They tend to be **slower** in inference because of the multiple stages and per-region processing. Faster R-CNN, for instance, might process 100 proposals per image; if the image is large or many objects, it can be computationally heavy (though "fast" and "faster" R-CNN have made it reasonably efficient for many uses). In EO context, two-stage detectors can be beneficial when precision is paramount – e.g., detecting small cars in a congested area where you can't afford to miss any or have false alarms.

- **Single-Stage Detectors:** These models do localization and classification in one go – **one stage**. They typically divide the image into a grid and predict bounding boxes and class probabilities *directly*, without a separate proposal generation step. The hallmark examples:
- **YOLO (You Only Look Once):** A family of models (v1 through v5 and beyond) that frame detection as a regression problem. YOLOv1 (2016) divided the image into an S×S grid and each grid cell predicted a fixed number of boxes and class probabilities. YOLO is designed for speed – it can run in real-time even on video [21]. Over iterations, YOLO has drastically improved accuracy, nearly closing the gap with two-stage methods while retaining speed. It's popular in many practical applications due to its simplicity and performance.
- **SSD (Single Shot MultiBox Detector):** Another one-stage approach that uses multiple feature map scales to detect objects of different sizes (e.g., early layers for small objects, later for large). SSD also predicts multiple boxes per location with different aspect ratios (so-called anchor boxes concept, which we'll explain shortly).
- **RetinaNet:** A one-stage detector that introduced the focal loss to address the class imbalance between background and object examples, improving one-stage accuracy.

Single-stage detectors **prioritize speed and simplicity** [22]. By not having a separate proposal stage, they streamline the process and can often run in a single forward pass of the network. The trade-off historically was slightly lower accuracy, especially for small objects or crowded scenes, but modern one-stage models (like YOLOv5, v7 etc.) are very accurate. In scenarios where real-time or near-real-time inference is needed (say, processing live drone footage, or running on an embedded device), one-stage detectors are preferred. For EO, consider monitoring a busy port for ships: a one-stage detector on each new satellite image could quickly give counts and locations without heavy computation – valuable for frequent monitoring.

- **Transformer-Based Detectors:** The newest wave, exemplified by **DETR (DEtection TRansformer)** by Facebook AI. These models use Transformers (as in the attention mechanism from NLP) to detect objects, removing the need for many heuristic components:
- DETR treats object detection as a **set prediction** problem. It uses a CNN backbone to extract image features, then a Transformer encoder-decoder to globally reason about object relationships, and directly outputs a set of object bounding boxes and classes.
- Notably, DETR does **not use predefined anchor boxes or a proposal stage**, and it doesn't require non-maximum suppression to remove duplicates [23]. The Transformer decoder is fed a fixed number of "object query" vectors which it uses to attend to the image features and output that many object predictions (with a special class for "no object" to allow fewer detections than queries).
- DETR's innovation also includes a unique loss that performs bipartite matching between predicted and ground truth boxes, so each predicted box is assigned to at most one true object.

The advantage of transformer-based detectors is a **simpler pipeline** (no manual anchors, no NMS) and the ability to capture global context via attention (potentially helping in complex scenes). DETR achieved comparable results to strong CNN detectors on benchmarks, though one downside was that it required a lot of training data and time to converge. Researchers are actively improving on it (e.g., deformable DETR to speed up training, etc.). For our advanced audience, mentioning DETR shows where the field is heading. For instance, in remote sensing, DETR could handle cases with arbitrary orientations and dense scenes by inherently learning spatial relations without needing rotation-specific tweaks. We'll note that some very recent works apply transformers to aerial object detection with success, although they may not be as widely used yet as YOLO/Faster R-CNN in practice.

To sum up this module, we will contrast the approaches: - Two-stage = "scan then zoom in": **high accuracy**, more computation. - One-stage = "all at once, like magic": **fast**, good accuracy (especially modern ones), may struggle on extreme small objects without customization. - Transformer-based = "new kid on the block": **no prior assumptions** (like anchor shapes), promising results by letting the model learn to propose and refine by itself [23].

This overview gives participants context if they read about "Faster R-CNN vs YOLO" or see detection frameworks in the wild. It's also a primer for the next hands-on where we will likely choose a one-stage model (for ease of use and speed in Colab) or a pre-trained two-stage via an API.

Applications of Object Detection in Earth Observation

Object detection has *numerous* applications in EO, especially for tasks where counting or pinpointing specific features is important:

- **Vehicle and Aircraft Detection:** Counting cars in parking lots, detecting moving vehicles from aerial video, or finding airplanes on runways in satellite images (for airport activity monitoring). This can support traffic analysis, security, or military intelligence.

- **Ship Detection:** Identifying ships in ports, coastal waters, or open ocean. Useful for maritime traffic monitoring, illegal fishing detection, border security, and rescue operations. (E.g., using SAR imagery to detect ships at night or in all-weather.)
- **Building Detection:** Locating buildings in high-res imagery to aid in mapping urbanization, estimating population (each house detected could imply something about population if other data are absent), or rapid damage assessment (detecting collapsed buildings after an earthquake, for instance).
- **Infrastructure Mapping:** Detecting specific structures like oil and gas storage tanks, wind turbines, solar panel arrays, communication towers, etc. This has applications in economic monitoring and asset mapping. For example, there are projects detecting all the round oil tank farms from satellite images using object detectors.
- **Airplane or Vehicle Counting in Disaster Response:** Post-disaster, counting how many vehicles are in an area (to gauge evacuation or recovery activity) or how many aircraft are at an airport (for relief logistics) can be automated with detection.
- **Deforestation or Environmental Monitoring:** Though often approached via segmentation or classification, certain aspects can use detection – e.g., detecting individual trees in high-res images (if we treat tree crowns as objects) to assess tree count and health. Also detecting animals in aerial images (wildlife surveys using drones).
- **Informal Settlements and Urban Features:** As a segue to our next session, detecting clusters of small buildings or shanties as distinct objects in a scene to map informal settlements expansion.

We will mention a couple of real examples to ground these ideas. For instance, the xView dataset (by DIUx) contains annotated objects in satellite images across 60 classes (vehicles, buildings, etc.), demonstrating the breadth of EO detection. Another example: during the COVID-19 pandemic, satellite-based car detection in parking lots was used to infer retail activity or social distancing compliance. Essentially, whenever a question asks "How many X are in this area and where are they?", object detection is the tool to use.

Participants should appreciate that detection gives a different kind of output than segmentation: a list of object coordinates and types. This is directly usable for databases (e.g., geolocating all ships detected, then tracking them over time) and feeding into GIS analyses (each detection can be a point or polygon on a map with attributes). In DRR and NRM (natural resource management), detection can help quantify exposure (e.g., how many buildings lie in a floodplain by detecting them on imagery) or monitor illegal activities (detecting illegal mining or logging sites from imagery). It's a versatile technique that complements pixel-wise mapping.

Challenges of Object Detection in EO

While object detection in natural images (like the COCO dataset of everyday photos) is mature, applying it to satellite or aerial imagery introduces specific challenges that we will highlight:

- **Small Object Sizes:** Often, targets in EO images are tiny relative to the image. A car in a 30 cm resolution image might be ~20 pixels long; a small hut in a 1.5 m resolution image could be only a few pixels. Detecting such small objects is difficult because they contain limited pixels for the model to recognize features [24]. Models may miss them or misclassify them. Specialized techniques (like using higher-resolution feature maps, focal loss, super-resolving the image, or simply using higher resolution sensors) are often needed.
- **Scale Variations:** EO imagery can cover a huge range of scales. In one image, you might have both large objects (an airplane, ~50 m long) and very small ones (cars, 4 m). Detectors need to handle this

scale variation. Feature pyramid approaches (like SSD's multi-scale feature maps or FPN in Faster R-CNN) were developed for this reason. Still, if scale range is extreme, it's challenging – e.g., detecting both a big ship and a small fishing boat in the same scene.

- **Orientation and Rotation:** Unlike ground photos, aerial objects can appear in any orientation (there is no "upright" for a building or car in a nadir image). A car could point north-south, east-west, or anywhere in between. Standard detectors use axis-aligned bounding boxes which often end up covering a lot of background for diagonally oriented objects. There's been research on rotated bounding boxes or rotation-invariant features [25]. We'll mention that in many EO tasks (e.g., detecting airplanes), one often uses data augmentation by random rotation so the model learns to detect objects regardless of orientation. Nonetheless, orientation remains a challenge – a model trained only on north-oriented ships might struggle on east-oriented ones if not properly addressed.
- **Complex and Cluttered Backgrounds:** EO images, especially in urban areas, are extremely dense with detail. For example, detecting buildings in a dense city: the background is "other buildings", which are very similar to the target building. The detector has to distinguish individual instances in aggregations. Or detecting vehicles in a city: vehicles might be adjacent or partly occluded by trees, etc. The high-detail background can confuse models or yield many false positives (e.g., a patterned roof might look like a cluster of small objects). We mention the need for high-quality training data and perhaps higher model capacity to handle this.
- **Atmospheric and Sensor Effects:** In optical images, lighting, shadows, haze, or clouds can obscure objects. A plane under a cloud might not be detectable. In SAR images, speckle noise or layover (distortions from side-looking radar) can create artifacts that look like objects or hide objects. These are issues unique to EO data collection. For example, if using Sentinel-2 for car detection, a shadow of a building might trick the detector or a bright reflective rooftop might saturate and hide detail. Robust detection in EO often must be resilient to these factors, potentially through preprocessing (e.g., shadow detection) or leveraging multi-temporal data.
- **Limited Labeled Data:** Creating large labeled datasets for detection in EO is a big challenge [26]. While ImageNet or COCO have millions of annotations, an EO project might have to manually label thousands of objects on satellite images, which requires expertise (knowing what a small blob of pixels is) and is time-consuming. There are some public datasets (like xView, DOTA, etc.), but for specific tasks (like informal settlements in Manila) you likely have to label your own. This scarcity of data can make training detection models from scratch difficult – hence the importance of **transfer learning** (using models pre-trained on other data) which we will apply in Session 4. We'll stress that models like YOLO trained on COCO (everyday images) can still be fine-tuned to detect, say, buildings, because they have learned to recognize shapes and patterns in general; you just need a smaller dataset to adapt them. But if only a very small dataset is available, performance might be limited. Few-shot or weakly supervised detection research is ongoing to tackle this [27].

Addressing these challenges often requires tailoring detection approaches for EO. For instance, researchers have developed *rotation-sensitive* detectors for aerial images that output rotated bounding boxes [28] to better fit objects like buildings or ships at an angle. Others use *context modeling*: e.g., if you detect one ship, you expect more nearby if in a shipyard (using spatial context to reduce false alarms). Small object detection is an active area – tricks like zooming into image tiles or using super-resolution on small proposals can help. We want participants to be aware that while the algorithms fundamentally are the same, applying them effectively to EO data might require extra considerations (like those above).

Overall, this module equips participants with conceptual knowledge of object detection, preparing them for the hands-on session where they'll see these ideas applied. We emphasize that by clearly distinguishing detection from segmentation, and by understanding the types of models and challenges, they will be better prepared to choose or tweak detection methods for their own geospatial tasks.

Session 4: *Hands-on:* Object Detection on Sentinel-2 Imagery (Urban Monitoring Focus) – 2.5 hours

Case Study: Informal Settlement Detection in Metro Manila (Urban DRR/NRM)

For our final session, we dive into object detection with a real-world scenario relevant to the Philippines: monitoring **informal settlements (slums) in Metro Manila**. Informal settlements are densely built, unplanned communities that often lack official infrastructure and are vulnerable to hazards (floods, fires, earthquakes). Identifying and monitoring the growth of these settlements is crucial for disaster risk reduction (knowing where vulnerable populations are) and for urban planning and resource allocation. However, keeping track of them on the ground is challenging due to their often rapid change and sometimes hidden nature.

Why this case: Metro Manila has numerous informal settlements, sometimes along rivers, coastal areas, or idle lands – places often at higher risk of flooding or other disasters. Local authorities and organizations need up-to-date maps of these communities to target interventions (e.g., flood early warnings, housing upgrades, provision of services). By using **satellite imagery**, we can periodically detect where new settlements are emerging or existing ones expanding. This is a clear example where AI can assist policy: instead of waiting for surveys, an object detection model could flag new clusters of rooftops in outskirts or along a river.

Specifically, we focus on an area like **Quezon City or along the Pasig River** corridor, where informal settlements are known to exist. Our object of interest might be defined as "building-like structures that likely indicate an informal settlement." We will treat detection of these structures as a proxy for locating informal communities. Once detected, the data can be used to measure changes (are settlements growing?) or to assess exposure (how many homes are in a flood-prone riverbank zone, for instance).

This case study ties back to DRR because informal settlements often suffer disproportionate damage in disasters – they may be in hazard-prone zones and have less resilient construction. By being able to detect and map them, agencies can plan risk reduction measures (like relocations or infrastructure improvements) more proactively [29]. It also touches NRM, as informal settlements relate to land use patterns and often to environmental issues (like settlements in protected wetlands, etc.).

We clarify that we're using **Sentinel-2 optical imagery (10 m resolution)** for this exercise. Sentinel-2's resolution means individual small houses are not directly visible as separate tiny objects (10 m pixels might cover several houses). However, larger settlement clusters (tens of meters across) do show up as distinct urban patches (usually a grayish or bluish roof-colored blotch in a sea of green vegetation, for example). Ideally, detecting informal settlements would be done with higher resolution images (e.g., PlanetScope at 3 m or commercial 0.5 m imagery). But given data accessibility and computational ease, Sentinel-2 provides a free and manageable dataset for our hands-on. We can frame the task as detecting "built-up patches indicative of informal housing" rather than individual huts. The model might actually be learning to detect small clusters of built-up area (a group of bright pixels) against vegetated or planned urban backgrounds.

Platform and Data Setup

As with Session 2, we'll use **Google Colab with GPU** as our development environment. The deep learning framework will be the same as we used in previous sessions (ensuring consistency so participants aren't confused by different syntax – e.g., if we used TensorFlow so far, we continue with that, or likewise with PyTorch). Consistency helps them re-use code patterns (like how to define a model or run training loops) from prior hands-on work.

Data provided: We have prepared a set of **Sentinel-2 image patches** focusing on specific areas of Metro Manila. Each patch might be, say, 256×256 pixels at 10 m resolution, covering a neighborhood scale area (~2.5 km across). These patches are taken from both known informal settlement areas and non-informal areas for contrast. They could be multi-spectral (Sentinel-2 has many bands), but for simplicity we might use a composite like RGB or RGB+NIR (4 channels) so that the model can utilize spectral differences (vegetation vs built-up have distinct signatures: vegetation is very bright in NIR, whereas buildings are not). Alternatively, we might already have them converted to a simpler feature set or indices (like an NDVI layer plus visual bands) – but likely sticking with a few raw bands is fine.

Annotations: Along with the images, we provide **bounding box annotations** for informal settlements or buildings in those patches. These annotations could have been drawn manually or using some reference data (e.g., building footprints from OpenStreetMap or a previous analysis). The annotation format will list coordinates of boxes and a class label (in our case maybe just one class: "settlement" or "building"). For example, in one image patch covering a riverside, we might have 5 boxes each encircling a dense group of small roofs that constitute a slum cluster. We'll ensure the format is compatible with whatever detection API or model we use (could be a CSV of box coordinates or a TFRecord or COCO-style JSON, depending on our approach, but we'll abstract away too much detail by likely loading them as needed).

Note on Sentinel-1 vs Sentinel-2: The TOR suggested Sentinel-1 could also be used for built-up change detection. Indeed, SAR can detect urban changes, but interpreting SAR for fine features is harder for beginners. We choose Sentinel-2 for the hands-on partly because it's easier to visually understand (participants can actually see the imagery and the boxes on them in a familiar way). We will mention that alternative approach: one could also do object detection on SAR (for example, detecting ships in Sentinel-1, or changes in built-up areas using SAR time series). But optical data is more intuitive for our first go at detection.

Approach and Workflow

Given the limited time and the complexity of implementing detection models from scratch, we present two options for the hands-on, emphasizing that **Option A (pre-trained model fine-tuning)** is the recommended one for most participants, while **Option B (from-scratch simple model)** is for those who are more adventurous or if time/instructor capacity allows a deeper dive:

Option A: Fine-Tune a Pre-Trained Detector – This is the practical, efficient path. We will leverage a model **pre-trained on a large dataset** (like COCO or Open Images) and fine-tune it on our specific task of settlement detection. Many libraries make this easy: - In TensorFlow, one can use models from the TensorFlow Model Zoo or TF Hub. For example, a SSD MobileNet or Faster R-CNN model pre-trained on COCO (which includes classes like "house" or "building" conceptually) can be loaded with a few lines of code.

- In PyTorch, we have `torchvision.models.detection` which includes pre-trained Faster R-CNN and SSD, or using a library like YOLOv5 which allows transfer learning.

We will pick a lightweight model for speed – e.g., **SSD with MobileNet backbone** or **YOLOv5s** – something that can train a few epochs on Colab within our session and produce results. We'll explain that since the model already knows general features for object detection (edges, shapes, etc.), we only need to train it a bit on our specific class. This drastically reduces training time and data requirements. It's a prime example of **transfer learning**, which we highlight as a key strategy in EO AI due to often limited labels.

Workflow for Option A: 1. **Load Pre-trained Model:** e.g., `model = tfhub.load('SSD_MobileNet_V2_COCO')` and adjust it to our single-class ("settlement") output. Or in PyTorch, use `models.detection.fasterrcnn_resnet50_fpn(pretrained=True)` and then replace the head with a new one with num_classes=2 (settlement vs background). 2. **Prepare Training Data:** Convert our annotated data into the format needed (maybe we'll use a DataLoader that yields images and target dicts of boxes and labels). We ensure shuffling, batching, etc. 3. **Fine-Tune:** Train the model for a short duration on our dataset. Because it's pre-trained, even a few epochs might suffice to get it to detect our target class. We watch the loss decrease. If using a high-level API, we might not see per-iteration metrics easily, but we could evaluate periodically. 4. **Evaluate:** After fine-tuning, run the model on a held-out test set of patches. Use detection metrics – e.g., **mean Average Precision (mAP)** at certain IoU thresholds (the standard for detectors). Given our single-class scenario, this simplifies to average precision for that class. We might compute the precision-recall curve if feasible or simply report mAP. If coding this is too involved, we can at least calculate precision/recall at a chosen IoU threshold (like IoU>0.5 to count a detection as correct). We'll discuss results in terms of "It found X% of the true settlements (recall) with Y% precision, and the average precision is Z." 5. **Visualize Detections:** Take some test images and plot the predicted bounding boxes on them (with confidence scores). This is always exciting for participants – seeing the model put boxes around what it thinks are settlements. We'll likely see it outline the clusters of tin roofs or houses. We can overlay the ground truth boxes as well for comparison (to see if it missed or mislocated any). Qualitatively, this shows strengths and weaknesses – maybe it found the obvious large clusters but missed a very small one, or maybe it confused a bright industrial roof as a slum.

This approach emphasizes getting something working quickly and is akin to what one would do in a real project (use a pre-trained model to save time).

Option B: Implement a Simplified Detector (YOLO-like) from Scratch – For a deeper understanding, one could attempt to code a basic one-stage detector. This is quite ambitious for a short session, so we frame it as optional or conceptual. Key ideas we might demonstrate or pseudocode: - Dividing the image into a grid (say 16×16 grid for a 256px image, each grid cell responsible for predicting objects that fall into it). - Using a few anchor boxes per grid cell (pre-defined box sizes) to predict adjustments and confidences. - Loss function consisting of localization loss (difference between predicted and true box), confidence loss, and classification loss. - Non-Maximum Suppression to filter overlapping boxes in the end.

Actually implementing this end-to-end and training might be too much, but we can show an **illustrative snippet** for one forward pass or how predictions are encoded. Alternatively, if we had more time, we could implement a very tiny model (like a 2-layer conv net that outputs a grid of predictions) just to grasp the mechanics, and train on a very small subset. However, due to complexity, we expect most will stick to Option A, and we'll primarily ensure Option A works.

Regardless of option, participants should learn how to go from an annotated image dataset to a trained detection model and then to inference.

Training considerations: We'll mention that detection models often need longer training and more data to really shine, and hyperparameters like learning rate are crucial. In our fine-tuning, we might freeze early backbone layers and only train the detector heads initially (common practice to avoid overfitting small data). Or use a lower learning rate for pre-trained layers. We simplify these details but mention them for completeness.

Running the Workflow (Step-by-Step in Colab)

We will structure the Colab similar to Session 2:

1. **Setup and Data Loading:** Load image patches and annotations into memory. Perhaps convert them to a suitable dataset object. Print a sample annotation to demystify it (e.g., "Image1 has 3 boxes: [[x1,y1,x2,y2], ...]"). Possibly display one image with ground truth boxes as a sanity check.
2. **Model Selection:** If Option A, load the pre-trained model. Show the structure and how we modify the final layer. If Option B, define our custom model architecture (a small conv network).
3. **Training Loop:** Train the model. For TF, maybe use `model.fit()` if using a high-level API, or a custom train loop for PyTorch. Because detection outputs are more complex, using built-in libraries might be easier (like TensorFlow's Object Detection API or PyTorch Lightning). But we aim to keep it understandable rather than treat it as a black box. We'll log the training progress.
4. **Evaluation:** Use the model to predict on test images. Then compute metrics. We may use an existing library function to compute mAP if available to save time.
5. **Visualization:** Plot results as mentioned.

Throughout, we encourage participants to think about why the model makes certain errors. For example, if the model flagged a false positive (FP) – maybe it saw a cluster of trees with a pattern that it mistook for roofs. This leads to discussion: how to reduce such FPs? Perhaps incorporate multi-spectral info (trees would be green in NIR, roofs not), or filter by shape (trees have different texture). Or if it missed a true settlement (FN), was it because it was too small or faint? That might indicate the resolution is a limitation or more training examples of that kind are needed.

Conceptual Hurdles and Clarifications

During this hands-on, we revisit and clarify several concepts to ensure EO folks grasp them:

- **Classification vs Detection vs Segmentation (revisited):** By now, we've done classification (Day 2 likely), segmentation (Day 3 Session 1-2), and detection (Session 3-4). We will take a moment to clearly contrast these outputs with visual examples, so participants are not confused. For instance, show a simple image and say what each task's output would look like. This helps in reinforcing the unique role of detection. It's common for newcomers to mix up segmentation masks and detection boxes – we rectify that clearly.
- **Anchor Boxes:** For those new to detection, the idea of anchor (prior) boxes can be perplexing. We will explain it in intuitive terms: *"Instead of the model freely drawing any box, we give it a few default box shapes/sizes at each location to start from. The model then only needs to adjust those to fit the object and decide if an object is present."* This simplifies learning because the model doesn't start from scratch for

box size. We might draw a grid and show anchor boxes (like one large, one medium, one small rectangle at a grid cell) to demonstrate.

- **Non-Maximum Suppression (NMS):** We introduced this as a post-processing step: *“The model might predict multiple overlapping boxes for the same object (especially if anchors overlap). NMS is an algorithm that keeps the best prediction and removes others that overlap significantly with it, so you get one clean detection per object.”* In many frameworks, NMS is done under the hood, but we tell participants it’s happening so they understand why they don’t see duplicate boxes in results. We can even show what it would look like before NMS (many boxes clustered on one object) vs after (one box).
- **Performance Metrics (mAP):** Unlike segmentation where IoU or Dice is straightforward, detection uses mAP which can be abstract initially. We will break it down: how for each class, the precision/recall curve is computed from the ranked detections, and AP is the area under that curve, and mAP is mean AP over classes. Since we essentially have one class, AP = mAP for us. We’ll emphasize interpretation: an AP of 0.5 means on average the model is doing okay but missing quite a bit; an AP of 0.9 would mean it’s almost perfect at that IoU threshold. We likely won’t dive into computing it manually (unless we have a tool), but we explain what the reported number means in plain language.
- **Transfer Learning Benefits:** We point out how quickly we fine-tuned a pre-trained model to our task, as opposed to training from scratch which would be impractical with our small dataset. This teaches participants a strategy for their own projects: start from pre-trained weights whenever possible. It also subtly shows them that these models learn general features that transfer to different imagery – a fascinating aspect (e.g., a model originally trained on everyday pictures can still help detect buildings in satellite images because edges and shapes are universal features to some extent).

In discussing these concepts, we keep linking back to the EO context. For example, *“We used a pre-trained model that was trained on a dataset like COCO (which has images of everyday scenes, not satellites). Yet it provides a strong starting point – it had filters that detect lines, corners, etc., which are useful for satellite images too. We only had to fine-tune it to understand our specific aerial perspective and what our ‘building’ class looks like from above.”* This demystifies the idea that models trained on one domain can help another (transfer learning across domains, which is common in EO because labeled EO datasets are relatively small).

By the end of Session 4, participants will have built or fine-tuned an object detector and applied it to real satellite imagery of their region of interest. They will see the outcome as boxes on an image, bringing a satisfying conclusion to the three-day course: they’ve covered classification (Day 2 likely), segmentation (Day 3 morning), and detection (Day 3 afternoon) – a comprehensive tour of deep learning in EO. We’ll have one more reflection on how these techniques can be combined (for instance, one could first detect buildings, then run segmentation to delineate each building’s footprint precisely, or detect ships then segment the water vs land, etc.).

Finally, we wrap up by encouraging them to think of other EO problems where these advanced techniques could be applied, and to continue exploring with the foundational knowledge they’ve gained. By maintaining consistency with Day 2’s style (hands-on exercises, clear conceptual modules) and briefly revisiting key CNN concepts (padding, upsampling, etc.) when relevant, we ensure the learning experience is cohesive and reinforcing across the days. The participants should leave Day 3 feeling empowered to tackle their own geospatial AI challenges using the advanced tools of semantic segmentation and object detection.

Sources: The concepts and techniques above are drawn from widely-used deep learning models and recent advancements in the field, applied to Earth Observation contexts [1] [4] [13] [23] [29].

---

[1] [2] Difference between Image Classification, Object Detection, and Image Segmentation
https://www.geowgs84.com/post/image-classification-vs-object-detection-vs-image-segmentation

[3] [4] [5] [6] How does the U-NET architecture leverage skip connections to enhance the precision and detail of semantic segmentation outputs, and why are these connections important for backpropagation? - EITCA Academy
https://eitca.org/artificial-intelligence/eitc-ai-adl-advanced-deep-learning/advanced-computer-vision/advanced-models-for-computer-vision/examination-review-advanced-models-for-computer-vision/how-does-the-u-net-architecture-leverage-skip-connections-to-enhance-the-precision-and-detail-of-semantic-segmentation-outputs-and-why-are-these-connections-important-for-backpropagation/

[7] UNet and its Family: UNet++, Residual UNet, and Attention UNet
https://www.vizuaranewsletter.com/p/unet-and-its-family-unet-residual

[8] [10] [11] [13] [15] Common Loss Functions in Modern Machine Vision
https://www.unitxlabs.com/resources/loss-function-machine-vision-system-common-loss-functions/

[9] [12] [14] Understanding Loss Functions for Deep Learning Segmentation Models | by Devanshi Pratiher | Medium
https://medium.com/@devanshipratiher/understanding-loss-functions-for-deep-learning-segmentation-models-30187836b30a

[16] CAT-i Bulletin: Typhoon Vamco (Ulysses) - Guy Carpenter
https://www.guycarp.com/insights/2020/11/cat-i-bulletin-typhoon-vamco-ulysses.html

[17] [18] PowerPoint Presentation
https://dges.carleton.ca/courses/IntroSAR/Winter2019/SECTION%207A%20-%20Carleton%20SAR%20Training%20-%20SAR_processing_Sentinal1%20-%20FINAL.pdf

[19] Full article: Flooded area detection and mapping from Sentinel-1 ...
https://www.tandfonline.com/doi/full/10.1080/22797254.2024.2414004

[20] [21] [22] Two-Stage Object Detectors Explained | Ultralytics
https://www.ultralytics.com/glossary/two-stage-object-detectors

[23] DETR: Overview and Inference
https://learnopencv.com/detr-overview-and-inference/

[24] Small object detection: A comprehensive survey on challenges ...
https://www.sciencedirect.com/science/article/pii/S2667305325000870

[25] [26] [28] Tobler’s First Law in GeoAI: A Spatially Explicit Deep Learning Model for Terrain Feature Detection Under Weak Supervision
https://arxiv.org/html/2508.03745v1

[27] Few-shot Object Detection with Feature Attention Highlight Module ...
https://arxiv.org/abs/2009.01616

[29] Satellite images accurately map all informal settlements in Manila | GIM
https://www.gim.be/en/about-us/customers/what-our-clients-have-to-say/satellite-images-accurately-map-all-informal-settlements-in-manila