ChatGPT

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

Overview: Building on the fundamentals from Day 2 (where we covered image classification with ML and basic CNNs), Day 3 dives into more advanced deep learning tasks in Earth Observation (EO): semantic segmentation and object detection. By the end of today, participants will have a **comprehensive tour of deep learning in EO**, having seen how to classify entire images, label each pixel in an image, and detect discrete objects in an image. Throughout, we maintain the hands-on, example-driven style of Day 2, and we'll briefly recall key CNN concepts (like padding to preserve dimensions, upsampling to increase resolution, etc.) to reinforce understanding.

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

Concept of Semantic Segmentation

Semantic segmentation is the task of **classifying every pixel** in an image into a category, producing a detailed, pixel-wise map of the image content. Unlike image classification (which assigns one label per image) or object detection (which locates objects with bounding boxes), segmentation provides a **fine-grained understanding** of the scene [1] [2]. For example, in a satellite image we might label each pixel as water, building, forest, road, etc., thus outlining the exact shapes and areas of these features. This pixel-level approach is crucial in EO applications where understanding boundaries and spatial extents (e.g. flood extent, land cover patches) is important. We contrast these tasks visually and conceptually so participants clearly grasp that **image classification** gives a coarse summary, **object detection** gives localized labels for individual items, whereas **semantic segmentation** yields a rich map of the entire scene.

U-Net Architecture for Segmentation

One of the most popular deep networks for segmentation in both medical imaging and EO is **U-Net** [3]. The U-Net has a distinctive "U" shaped architecture composed of two paths: an **encoder (contracting path)** and a **decoder (expansive path)**, with skip connections between matching levels of the two. The **encoder** is a series of convolutional and pooling layers that progressively **downsample the image**, extracting higher-level features while reducing spatial resolution (just as we learned with CNNs on Day 2). For instance, using 3×3 convolutions (with ReLU activations) and 2×2 max-pooling, the encoder learns rich features but shrinks the image size at each step [4]. (Recall from Day 2: pooling without padding reduces image size, losing some detail.) The **decoder** then performs the reverse: it uses upsampling (e.g. transpose convolutions) to **increase the spatial resolution**, gradually building the output segmentation map [5]. Crucially, U-Net's decoder is fed by **skip connections** from the encoder: the feature maps from the encoder are **copied and concatenated** onto the decoder's feature maps at corresponding levels [6]. These skip connections provide high-resolution context to the decoder, ensuring that fine details (like precise boundaries) are preserved even after the image was compressed by the encoder. In essence, the encoder captures *what* is in the image (context), and the decoder, aided by skips, ensures we know *where* those things are in the image (precise localization). The middle of the U ("bottleneck") contains the most compressed, abstract representation of the image – the **semantic essence** that the network has extracted [7] .

Figure: U-Net architecture example. The left side is the encoder (contracting path) which downsamples the input while increasing feature channels (numbers on boxes). The right side is the decoder (expanding path) which upsamples back to the original resolution. Purple arrows denote skip connections that bring high-resolution features from the encoder to the decoder, helping the network recover fine details and delineate precise boundaries. The U-shape comes from the encoder and decoder paths mirroring each other.

A few implementation notes: to make concatenation in skip connections seamless, we often use **padding** in convolutions to maintain equal sizes between encoder and decoder feature maps (Day 2 covered how "same" padding keeps dimensions). The original U-Net paper cropped feature maps instead, but modern frameworks simply pad zeros so that encoder outputs and decoder inputs align. Also, upsampling in U-Net can be done via learned transposed conv layers (sometimes called "deconvolution") or simpler interpolations – either way, it's essentially the inverse of pooling (as a refresher from Day 2) to expand the image back to full size. By the end of the decoder, a 1×1 convolution produces the final segmentation map, with as many channels as target classes, so that each pixel receives a class label.

Applications of U-Net in Earth Observation

Semantic segmentation has wide-ranging applications in EO, and U-Net variants are extensively used for these. In disaster risk reduction (DRR), a prime example is **flood mapping** from satellite data: given a Sentinel-1 SAR image or Sentinel-2 optical image, a U-Net can identify flooded areas at the pixel level, producing a flood extent map. Studies have shown U-Net is effective in capturing flood patterns in SAR imagery [8] , achieving high accuracy in delineating water from land. Other applications include **detailed land cover mapping** (assigning each pixel to land cover classes like forest, agriculture, water, urban), **road network extraction** (tracing out roads and highways from aerial images), and **building footprint delineation** (segmenting exact outlines of buildings from high-resolution imagery) [9] . For instance, U-Net-based models have been used to extract buildings in urban areas, even with complex backgrounds, and to map roads winding through forests or cities. In each case, the pixel-wise output of segmentation gives a much richer output than simple classification – instead of just saying "there are buildings in this image," we get a map of *where* each building is. This is invaluable for urban planning, environmental monitoring, and emergency response (e.g. identifying flooded pixels vs. dry pixels across an entire region). We will showcase a few compelling examples, such as a U-Net mapping floodwater spread in a typhoon aftermath, and a U-Net delineating every building in a town, to illustrate the power of segmentation in EO.

Loss Functions for Segmentation

Training segmentation models brings unique challenges, especially class imbalance (e.g. in a flood image, "non-flood" pixels vastly outnumber flood pixels) and the need for precise boundary alignment. We introduce common loss functions used in segmentation and their characteristics:

- **Pixel-wise Cross-Entropy Loss:** The classic approach treats each pixel's class prediction as an independent classification, comparing to the one-hot ground truth. For multi-class segmentation, this is categorical cross-entropy (or binary cross-entropy for two classes). Cross-entropy is straightforward and works well when classes are reasonably balanced. However, if one class is very rare, a model could achieve good pixel accuracy by mostly predicting the majority class (since getting many background pixels right outweighs missing a few small objects). In such cases, cross-entropy can be augmented or replaced by other losses to handle imbalance.

- **Dice Loss (F1 score-based):** Dice loss is based on the Dice coefficient (F1 score) between predicted and true masks. It directly measures overlap: essentially **2 * (Intersection) / (Sum of areas)** for two sets. Dice loss is **especially useful for imbalanced data** [10] – it doesn't let the model get complacent by only predicting the majority class, because it focuses on the overlap of the positive (target) class. It treats false negatives and false positives more equally. In practice, Dice loss helps the network pay attention to small structures (like thin flood lines or tiny objects) that might be lost with cross-entropy. Many medical and EO segmentation tasks rely on Dice loss for this reason.

- **Jaccard Loss (IoU loss):** Jaccard index, a.k.a. Intersection over Union (IoU), is another overlap-based measure. IoU loss = 1 – IoU. This loss is closely related to Dice (they are mathematically similar), and is very intuitive: it directly maximizes the percentage overlap between prediction and truth. **IoU is a popular metric for segmentation accuracy**, especially to evaluate how well boundaries are captured [11]. As a loss, it also emphasizes getting the shape and placement of segments right rather than just each pixel independently. IoU (or Dice) loss is often used in conjunction with cross-entropy (e.g. a weighted sum) to balance per-pixel accuracy with overall region overlap.

We will explain how these loss functions work with simple examples (like a tiny image with a small object) to show why IoU/Dice are better at boundary focus and imbalance handling than plain pixel-wise loss. For our segmentation tasks, we might choose a **combined loss** (e.g. cross-entropy + Dice) to leverage the strengths of both – cross-entropy for stable learning and Dice to ensure overlap/shape accuracy [12]. We'll also mention **accuracy metrics** for segmentation: besides IoU, metrics like **Pixel Accuracy**, **Precision/Recall** for a class, and **F1-score** can be computed to judge performance on a validation set.

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

Case Study: We now put theory into practice with a real-world-inspired case. Participants will implement a U-Net to perform **flood inundation mapping** for a disaster scenario in the Philippines. Specifically, we consider a past severe flood event in Central Luzon (Pampanga River Basin), such as caused by Typhoon Ulysses in 2020 or Karding in 2022. Flood mapping is critically important for disaster response and is an ideal test for semantic segmentation in EO.

Platform & Framework: We use Google Colab (with GPU acceleration) for the coding exercise, continuing with the same deep learning framework as Day 2 (whether PyTorch or TensorFlow, for consistency). This allows participants to re-use and extend the basic CNN knowledge from yesterday into building a U-Net today.

Data: For the sake of time, we provide **pre-processed Sentinel-1 SAR image patches** (tiles of say 128×128 or 256×256 pixels) covering the flood-affected area, along with corresponding **binary flood masks** (pixel value 1 for flooded water, 0 for non-flood). Each image patch has two channels (Sentinel-1's VV and VH polarizations), giving more information to distinguish water from land. The ground truth flood masks were derived from authoritative sources or careful manual annotation for that typhoon event. By providing ready-to-use patches and labels, we let participants focus on model building rather than satellite data preprocessing.

> **Data Preparation – A Crucial but Hidden Challenge:** We highlight that in real projects, getting SAR data to an "analysis-ready" state is a significant effort. Raw Sentinel-1 images require multiple preprocessing steps: **speckle noise filtering**, **radiometric calibration** to backscatter coefficients, and **geometric terrain correction** to align with map coordinates, among others [13]. This is computationally heavy and time-consuming. Likewise, obtaining accurate flood ground truth is hard – it might involve interpreting radar signals, using optical imagery, or field data. We make sure learners appreciate that behind the neat training patches provided, there's a lot of remote sensing expertise. (In our case, we assume that steps like removing thermal noise, calibrating to sigma-naught, speckle filtering, and terrain correcting have been done on the Sentinel-1 GRD imagery [13].) By handling these beforehand, the session can focus on the *deep learning* aspect, but we advise participants that operational flood mapping pipelines must include those preprocessing steps.

Workflow: The hands-on exercise will follow a structured workflow:

1. **Load Data:** Start by loading the SAR image patches (as numpy arrays or tensors) and their corresponding binary flood mask labels. We'll inspect a few examples to understand how floods appear in SAR (e.g. water usually gives low backscatter on SAR, appearing dark, while land is brighter – but this can invert for different polarization or if wind/waves on water). Visualizing input and label pairs helps confirm the data looks correct.

2. **Data Augmentation (optional):** If time permits, we introduce simple augmentations to make the model more robust. For EO, typical augmentations include random rotations or flips (since orientation might not matter for flood patterns), and perhaps slight cropping or adding noise. However, we are cautious with SAR data – radiometric characteristics should not be altered drastically. Augmentation is not a core focus here, so this step is optional.

3. **Define U-Net Model:** Participants will either code a U-Net architecture or use a high-level library if available. We specify the architecture: an encoder with several convolution blocks and pooling, a bottleneck, and a decoder with upsampling and skip connections. Given the small image patch size, we can use 3-4 levels in the U-Net (depending on image size). We'll remind them to use padding in convolutions so that the feature map sizes match when concatenating skip connections (a brief recall: padding avoids size reduction). If using TensorFlow/Keras, we might use `Conv2D` and `Conv2DTranspose` for down and up, or in PyTorch define the `forward` with `F.interpolate` for upsampling. This solidifies the understanding of how each part of U-Net is implemented in code.

4. **Compile Model:** We choose an appropriate loss function and optimizer. For this binary segmentation, a common choice is **Dice loss** or a combo like **BCE (binary cross-entropy) + Dice loss** to handle class imbalance (flood pixels are usually much fewer than non-flood). We discuss why Dice is helpful (as covered in Session 1). The optimizer can be Adam, with a moderate learning rate (since U-Net is a deeper model than the CNN from Day 2, and might need careful training). We also set up metrics like IoU and F1-score to monitor during training, in addition to plain accuracy.

5. **Train the U-Net:** We split patches into a training set and a validation set. During training (which may run for a few epochs on Colab's GPU), we monitor the training loss and validation IoU/accuracy each epoch. Participants will see the model gradually learn to segment water vs land. We talk about any signs of overfitting (if validation performance stalls or diverges from training). We keep training time reasonable by having a small dataset or a pre-trained backbone if necessary (but likely a small U-Net on 128×128 patches will train quickly).

6. **Evaluate on Test Set:** After training, we evaluate the model on a hold-out test set of patches. We calculate metrics: e.g., overall **IoU** for the flood class, the **precision** and **recall** (so they see trade-offs: did the model miss some floods (false negatives) or mark non-flood as flood (false positives)?), and the **F1-score** which summarizes precision/recall. Suppose the U-Net achieves, for example, 0.80 IoU and an F1 of 0.88 on the test – we'd discuss if that's good for this application (88% of flooded pixels correctly identified is quite good, though some misses may remain). We also emphasize looking at **per-patch performance** – maybe certain areas (e.g., urban or mountainous) are harder to classify for flood due to radar shadow or artifacts.

7. **Visualize Predictions:** A very important step is qualitative assessment. We will plot some test patches with the model's predicted flood mask overlaid on the SAR image. For example, show the SAR image in grayscale and color the predicted flooded pixels in blue. Participants can compare this to the ground truth mask. This visual check is compelling: they can see if the U-Net outlines the flooded river accurately, if it falsely lights up some areas, etc. We likely show cases of both success (the model neatly segmented the flooded fields) and failure (e.g., the model thought a lake was a flood, but maybe it was permanent water – pointing to how training data defines the classes). This cements understanding and gives a "real-world" feel of the output.

Throughout the hands-on, we keep an eye on a key **conceptual hurdle**: how a CNN-based model like U-Net can work on SAR data which looks very different from optical images. We explain that to the model, SAR is just another image (with a certain distribution of pixel values); the convolutional layers will learn to detect patterns like homogeneous dark regions (water) versus textured bright regions (urban or vegetation). The **physics of SAR** (penetration of clouds, sensitivity to surface roughness) is different from RGB images, but the network doesn't need to know physics – it learns from examples. We reassure participants that U-Net's success in flood mapping is well documented [8] and that its architecture (especially the skip connections) is well-suited to delineating **sharp boundaries** like where floodwater ends and dry land begins. The skip connections deliver the high-resolution details that ensure the flooded patches on the output map line up exactly with the SAR image features. By walking through this case study, participants not only practice coding a deep network but also gain insight into applying AI for a pressing DRR task in their region.

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

Concept of Object Detection

In this module, we introduce **object detection**, the next step up in complexity from image classification. Object detection involves **both** classification and localization: the model must find *where* each object is in the image (often via a bounding box) and *what* that object is [2]. For example, in a satellite photo of a city, object detection could locate every car (draw a box around each) and label them "car". This differs from **semantic segmentation**, which would mark each car's pixels, and from plain **classification**, which might only say "there are cars in this image" without location. We clarify these differences: detection gives discrete, instance-level results – each object instance gets its own box (and possibly a class label and confidence score). In contrast, semantic segmentation merges all pixels of a class into one mask without distinguishing instances. There's also **instance segmentation** (not our focus here) which is like a mix: it segments each object instance separately (e.g., masks for each car). For our needs, bounding-box detection is a practical approach to count objects or mark features in EO data.

We ensure everyone understands the output of an object detector: a list of bounding boxes (each defined by, say, x,y,width,height) with associated class labels and confidence scores. A common point of confusion is how detection algorithms can find multiple objects in a single pass – which leads us to discuss the architectures.

Overview of Popular Detection Architectures

Object detection has seen rapid evolution. We give a **high-level overview** of the main types of detectors, without delving into too much code detail (since implementation of these from scratch would be too advanced for this course).

- **Two-Stage Detectors:** These methods break the task into two phases. First, a **Region Proposal** step scans the image to propose candidate bounding boxes that might contain objects. Second, a classifier/refiner network examines those proposals to classify them and adjust the bounding box (and sometimes generate a mask, as in Mask R-CNN). The seminal example is **R-CNN** (Regions with CNN, 2014), which used an external algorithm to propose ~2000 regions and then a CNN to classify each [14]. This was slow, so **Fast R-CNN** and then **Faster R-CNN** (2015) improved the speed: Faster R-CNN introduced the **Region Proposal Network (RPN)** that is learned and runs inside the CNN itself, making proposals in a nearly seamless way. Two-stage detectors are known for **high accuracy** – because the second stage can scrutinize each candidate – and particularly they tend to handle **small objects** better (since proposals can be made at multiple scales and then a focused classifier examines each) [14]. However, they are slower, as they do two rounds of processing. In EO, where images can be very large and objects small, two-stage methods (like Faster R-CNN) have been popular for research, due to their accuracy.

- **Single-Stage Detectors:** These models do detection in one go, without an explicit second stage for proposals. Notable examples: **YOLO (You Only Look Once)** and **SSD (Single Shot MultiBox Detector)**. They treat detection as a **direct regression problem** over a dense set of potential locations [15]. In essence, the image is processed by a CNN and at multiple locations (and scales) the network outputs both the class prediction and bounding box coordinates *simultaneously*. YOLO divides the image into a grid and predicts boxes and class probabilities for each grid cell in one forward pass; SSD uses feature maps at multiple scales to detect both large and small objects in one shot. The advantage is **speed** and simplicity – these can run in real-time or on large images quickly [16] [14]. They historically had slightly lower accuracy than two-stage, especially for small or densely packed objects, but the gap has closed with newer versions (e.g., YOLOv5, v7 etc., and improvements like **RetinaNet** with focal loss to address class imbalance). In many practical cases, single-stage detectors are preferred when one needs fast results or to process lots of imagery quickly (e.g., scanning large areas for ships). We mention that modern one-stage detectors even dispense with predefined **anchor boxes** (the earlier YOLO/SSD used a set of preset box sizes at each location as templates). New **anchor-free** detectors predict keypoints or centers of objects instead, simplifying things further.

- **Transformer-Based Detectors:** An emerging class of detectors uses the power of Transformers (as seen in NLP) for detection. We briefly introduce **DETR (DEtection TRansformer)** by Facebook (2020) as an example. DETR removes the need for many hand-designed components: it uses a CNN backbone to extract features, then a Transformer encoder-decoder to directly output a set of object bounding boxes and classes, treating detection as a **set prediction problem** [17]. DETR does not require non-maximum suppression or anchor boxes; instead it uses a special loss (bipartite matching loss) to ensure each object is predicted once. The concept is advanced, but we want the audience to know the field is moving forward. Transformer-based models can be powerful (they can learn global relationships and context) but are computationally heavy. Some newer variants (Deformable DETR, etc.) have improved on the original to make it faster and better with small objects. If the audience is advanced enough, we'll mention how DETR could be the future of object detection research, but in practice, the well-established YOLO/Faster R-CNN paradigms are still very useful for EO tasks today.

By giving this overview, participants get the "lay of the land." Importantly, we will not dive into coding a detector from scratch (which would be complex), but understanding these concepts will help them use pre-built models intelligently.

We also clarify some common components in detectors:
- **Anchor boxes:** For those detectors that use them (like Faster R-CNN, YOLOv2/3, SSD), these are pre-defined box shapes (sizes and aspect ratios) tiled across the image. The network's job is to adjust and classify these. It's a bit like giving the model some starting guesses. We'll explain intuitively: imagine covering an image with a bunch of default boxes of various sizes; the model learns to pick which ones contain an object and tweak their boundaries to fit tightly.
- **Non-Maximum Suppression (NMS):** We describe how a detector might predict multiple overlapping boxes for the same object (especially one-stage detectors which operate locally). NMS is a simple algorithm that post-processes the output: it sorts detections by confidence and then removes lower-confidence ones that overlap too much with a higher one. This gives the final clean set of boxes. It's an important step to get from raw model outputs to a neat result, and even DETR had to replace it by a learned mechanism. We can give a quick example or visual: if two boxes overlap 80% and both say "car," NMS keeps the one with higher probability.

Understanding these concepts will demystify how detectors work under the hood when they use pre-trained ones.

Applications of Object Detection in EO

Object detection unlocks many impactful applications in the geospatial domain. We'll discuss several examples that resonate with local and global needs:

- **Vehicle and Aircraft Detection:** Using high-resolution imagery or aerial photos, detectors can find cars, trucks, or airplanes. This is useful for traffic monitoring (counting vehicles on roads, seeing congestion), or for activity monitoring (e.g., counting aircraft at an airport or vehicles in a parking lot as an indicator of activity). In a DRR context, vehicle detection can support evacuation planning or damage assessment (seeing where vehicles are stranded after a disaster, for instance).

- **Ship Detection:** In maritime surveillance or fisheries management, object detection is used on SAR or optical images to locate ships at sea [18]. This has applications in monitoring illegal fishing, tracking maritime traffic, or detecting ships for naval operations. Satellites can cover huge ocean areas, and an automated detector can flag where ships are, even small boats, which is much faster than an analyst scanning imagery.

- **Infrastructure and Building Detection:** Identifying structures like buildings, houses, or oil tanks in imagery is another common use. For example, in urban planning or in responding to informal settlements growth (as in our case study next session), detectors can rapidly pinpoint where new structures have appeared. In disaster damage mapping, detecting collapsed buildings (perhaps via changes or via direct detection of debris) is an active research area. Also, large storage tanks or solar panel arrays can be detected for industrial monitoring. In EO, often these objects are small in pixel size, so specialized techniques or high-res images are needed. Nonetheless, methods exist and are improving.

- **Environmental and Wildlife Monitoring:** Though less common, detection has been applied to count animals in aerial surveys (e.g., detecting elephants or whales in imagery), or detecting features like trees (for certain species counting) or even cloud features. Essentially, any time we need to count or locate instances of a class in an image, detection algorithms can be applied.

We will cite specific examples or success stories: e.g., an object detector that can find *ships, airplanes, and cars* in the large-scale xView or DOTA remote sensing datasets (which include those classes) [19]. Another example: using Sentinel-2 (10 m resolution) one can detect large structures like big buildings or circular tanks – perhaps coarse, but with deep learning it's been done. The key message is that detection broadens the questions we can ask of imagery: not just "what is the predominant class here?" but "how many X are there and where?".

We tie this to **DRR and NRM themes**: For disaster risk reduction, detecting buildings can help estimate populations at risk, detecting illegal structures can aid in disaster preparedness (e.g., houses built in flood-prone riverbeds), and detecting vehicles can assist in relief logistics. For natural resource management, detecting objects like trees (e.g., identifying individual tree crowns) or animals could be relevant. We encourage participants to think of detection problems in their work – maybe monitoring *small-scale mining* by detecting mining pits, or *ship traffic* in protected waters.

Challenges of Object Detection in EO

Object detection in EO imagery is challenging due to several factors that we highlight so participants have realistic expectations:

- **Small Objects:** Satellites often image large areas, so targets of interest (vehicles, small buildings, boats) may occupy only a few pixels or tens of pixels in the image [20]. Detecting such small objects is much harder than detecting, say, a large cat in a close-up photo. The object's features are limited and can easily be missed. This is where high resolution helps, and why research often uses aerial imagery or commercial satellite images for tasks like car detection. Techniques like using higher magnification in the detector, or tiling the image into smaller patches (as done in some XView challenge solutions), are employed to tackle this.

- **Varied Scales and Orientations:** Objects in EO can appear in very different sizes (a car vs. a building vs. a ship) and orientations (a ship could be oriented in any direction in the sea, cars can face any which way on roads). This variability means the detector must be scale-invariant and rotation-invariant to some degree. Standard detectors handle scale by multi-scale feature maps (SSD, FPN) and handle some rotation by data augmentation, but in EO sometimes **oriented bounding boxes** (rotated rectangles) are used in research to better fit angled objects (like diagonal oriented ships/buildings). We mention that while our mainstream detectors use axis-aligned boxes, EO datasets like DOTA include oriented boxes for better accuracy in dense scenes [19].

- **Complex and Cluttered Backgrounds:** Satellite images, especially of urban areas, are very busy – lots of buildings, shadows, roads, vegetation create a complex background. False alarms are a risk: a detector might think a strangely shaped roof is a ship, or a rock in a field is a vehicle. Distinguishing objects from background requires training on lots of examples and sometimes special tweaks (like context windows). The **background class** in detectors effectively is everything that is not of interest, and it's an endless variety. We explain this means detectors need a lot of diverse training data to not be confused by new scenery [20].

- **Viewing Angle and Atmospheric Effects:** Unlike ground photos, satellites often see objects from a top-down or oblique angle, and conditions vary. Haze or cloud cover, varying sunlight, etc., can all affect the appearance of objects. A car under a cloud shadow might be very hard to spot in an optical image; a building's height might cast a shadow that confuses things. In SAR, speckle noise or layover (tilted appearances of tall structures) add additional complexity. These are unique EO issues that CV algorithms must overcome, often by training with representative data or using data from multiple sensors.

- **Limited Labeled Data:** Creating large, annotated datasets for detection in EO is labor-intensive (imagine drawing boxes around thousands of tiny objects in hundreds of images!). While there are some public datasets (like xView, DOTA, COCO-Wildlife, etc.), for many niche tasks in EO, you might have only a small training set. This is why **transfer learning** is crucial – using models pre-trained on huge datasets like MS COCO or ImageNet and fine-tuning them on the EO task. It's also why we often prefer simpler models or augment heavily. We prepare the audience that results might not be as perfect as the examples they see in autonomous driving (where massive labeled datasets exist), but even a fine-tuned model can do impressive things with relatively few examples if done right.

By discussing these challenges, participants will understand why object detection can be tricky in practice and why the hands-on in the next session is scoped in a certain way (to be feasible). We might mention some cutting-edge solutions (like **super-resolution for small object detection**, or **slicing imagery into smaller tiles** to handle large images [21], or leveraging **synthetic data** to get more training examples). But the main takeaway: EO detection requires careful consideration of scale and context, and often more training data or augmentation to achieve robust results.

Session 4: *Hands-on:* Feature/Object Detection from Sentinel-2 Imagery (Urban DRR Focus) (2.5 hours)

Case Study: In this final hands-on session, participants will apply object detection to a local urban environment problem aligned with DRR and NRM. We focus on detecting and monitoring **informal settlements (unplanned housing)** and general **building structures** in the Greater Metro Manila area – for example, along the Pasig River or in parts of Quezon City known for dense, informal housing. This ties to disaster risk because informal settlements often are high-risk for fires, floods, or earthquakes due to building quality and density. Being able to automatically detect where new settlements are expanding can help in risk management and urban planning.

Data & Scenario: We provide **Sentinel-2 optical imagery** patches of the target area (Sentinel-2 has 10 m resolution in its best bands, which is coarse for individual buildings but large clusters of built-up area are identifiable). Each patch might cover a neighborhood (~a few kilometers square). Along with the images, we provide **annotation files** with bounding boxes around areas of interest – here, clusters of buildings or settlement areas – labeled appropriately (e.g., "building" or "settlement"). These annotations could be simplified, like one box per cluster of shanties, given Sentinel-2's resolution (we won't see individual houses clearly, but a cluster of bright pixels could indicate a settlement). If we had higher-res data (like Planet or aerial), we could detect individual houses, but we'll stick to Sentinel-2 for accessibility. We clarify that for real tasks, higher resolution is preferable for building detection, but Sentinel-2 can at least distinguish urban vs. non-urban patches and large structures.

We also mention that object detection can be done on SAR (Sentinel-1) for built-up areas (since SAR is sensitive to structures), but interpreting SAR detections is more complex for beginners. So, we choose Sentinel-2 for the hands-on due to its more intuitive, "photographic" appearance – participants will more easily see what the model is detecting.

Workflow: The exercise is designed to be achievable within a single session by leveraging pre-existing models. We outline two approaches and will likely follow **Option A (the simpler one)**, but we explain both for learning purposes:

1. **Load images and annotations:** We start by loading the Sentinel-2 patch images and the corresponding bounding box labels (perhaps in PASCAL VOC format or simple CSV). We will visualize a couple of images with their ground-truth boxes drawn (e.g., show an image with red rectangles around known settlement areas). This lets participants see the target outcome: "Ah, those highlighted areas are the informal settlements the model should detect." We'll ensure everyone understands the label format (each label might be the pixel coordinates of the box corners and a class name).

2. **Option A – Fine-tune a Pre-trained Detector:** We recommend using a **pre-trained model** (transfer learning) to save time. For example, using TensorFlow Hub or PyTorch Hub to grab a model like SSD Mobilenet or a YOLO model pre-trained on MS COCO (which knows about common objects). We can choose a lightweight model that runs fast on Colab. Since our task is "building/settlement detection," which is not exactly a COCO class, we'll have to map it – but COCO has "house" or "building" in some versions (if not, we treat it as a new single-class problem). We fine-tune the model on our dataset: this typically involves replacing the final layer to predict our class(es) and training for a few epochs on our labeled data. We emphasize how **transfer learning** helps: the model's early layers already learned to detect generic shapes and edges, which will help in detecting buildings even though it wasn't originally trained on satellite images [20]. We adjust the training so that it doesn't unlearn everything from COCO – e.g., by using a smaller learning rate and maybe freezing some backbone layers initially.

3. **Option B – Implement a Simple Detector Concept:** If the group is very advanced or curious, we discuss how one would implement a basic detector from scratch (conceptually). This could involve

using a CNN backbone (like a mini ResNet) and adding a prediction head that outputs, for each grid cell, some boxes and class probabilities (similar to YOLO). We clarify concepts like **grid cells** (dividing image into, say, an 8×8 grid, each cell responsible for detecting objects in that region), **anchor boxes** (preset box sizes that the network can choose to adjust), and how the network predicts coordinates (likely as offsets from anchors) plus an "objectness" score and class label. We won't code this fully (it's too time intense), but we might show a snippet or pseudocode for one forward pass. This helps demystify how YOLO works internally. We then likely steer back to Option A, noting that modern implementations are highly optimized and complex, so using them is the pragmatic approach in a short training.

4. **Training the Detector:** Using Option A's fine-tuning, we train the model on our dataset of images. Because the dataset might be small, we might only do a few epochs. If the pre-trained model is good, even a single epoch might show it adapting. We monitor a training loss (which in detection is a combination of classification loss and localization loss for the boxes). We also keep a validation set to see if we are overfitting. This training process is faster than training from scratch since the model already has good weights. We talk through what the loss means: e.g., localization loss measuring how far predicted boxes are from truth, classification loss for correctly labeling boxes vs background. Participants will see logs or simple metrics (like mean Average Precision if we compute it, or just loss decrease). We might train, say, for 10 epochs and see improvement.

5. **Evaluate Performance:** The standard metric for object detection is **mean Average Precision (mAP)** at certain IoU thresholds. We will simplify this if needed – possibly we compute the precision/recall for detected boxes vs ground truth at IoU 0.5 (50% overlap) as a success criterion, and derive an AP. Given we might have mostly one class (buildings), we can get an Average Precision for that class. We'll explain in simple terms: we sort model's detections by confidence and see how many true positives vs false positives as we go down the list, plotting a precision-recall curve and finding the area under it (AP) [22] . We'll likely not code the full mAP pipeline from scratch due to time; instead, we might use a library or just discuss results conceptually (e.g., "our model achieved an AP of 0.7 for building class, meaning it's pretty decent at finding them"). We also look at simpler metrics: what percentage of ground truth boxes were found (recall) and what fraction of detections were correct (precision). If multiple classes were involved (not likely here), we'd do per-class AP.

6. **Visualize Detected Outputs:** Finally, as a satisfying conclusion, we run the trained detector on some test images (or even on a larger image of Metro Manila that wasn't fully labeled, just to see what it finds). We display the images with the model's predicted bounding boxes drawn (perhaps in green). This qualitative check is exciting: participants can see, for example, the model put boxes around clusters of bright roofs along the river, correctly identifying new settlements; maybe it also put a box on a bright industrial building (false detection, if our label was only informal settlements). We discuss any false positives or negatives visible. Perhaps the model misses very small clusters or confuses some bright bare soil as a building. This opens a discussion – how to improve (maybe more training data, higher resolution, or adjusting the threshold). But overall, they get to see an **AI model in action on satellite imagery**, pinpointing features of interest, which is a big achievement from where we started.

Conceptual Reinforcement: During this exercise, we pay special attention to reinforcing the distinctions between the tasks: a single image can undergo classification (Day 2's task: e.g., is this image mostly urban or rural), segmentation (Day 3 morning: e.g., mark every pixel of water vs land), and detection (Day 3 afternoon: e.g., draw boxes around buildings). We might actually take one of our test areas and mentally overlay these different outputs to compare. For example, an area along Pasig River: a classification model might just say “urban” for the patch; a segmentation model could color in built-up area vs vegetation vs water in different colors; and our detection model places boxes on the denser building clusters. Each method gives a different level of detail and serves different purposes. We ensure participants can articulate these differences now. We also demystify anchor boxes and NMS as we encounter them in the code: if using a pre-trained model, we point out that “see, we didn’t have to write NMS – the library handled it after the model predictions”. If possible, we show one example of overlapping detections and how NMS resolved it.

By the end of this session, participants have **practical experience with an object detection workflow**: data preparation (with bounding boxes), using transfer learning, training a model, and interpreting the results. They also see how this connects to a real DRR problem in their context – monitoring urban growth. We emphasize that while our example is simplified (coarse resolution, maybe simplified labels), the same pipeline could be applied with finer data to do serious analyses, or combined with segmentation for even more detailed mapping (e.g., detect buildings then precisely segment their footprints for area measurements).

Conclusion & Next Steps

With the completion of Day 3, our training course has covered the three major computer vision approaches for EO:

- **Day 2:** Image Classification – assigning overall labels (e.g., land cover type) to image patches using ML (Random Forest) and basic deep learning (CNNs).
- **Day 3 (Morning):** Semantic Segmentation – classifying each pixel using U-Net, achieving detailed maps like flood extent or land cover maps.
- **Day 3 (Afternoon):** Object Detection – finding and localizing objects/features like buildings using advanced deep learning models (e.g., fine-tuned detectors).

This progression was intentional: starting from simpler concepts to increasingly complex tasks, we’ve given participants a **comprehensive tour of deep learning in EO**. Today’s hands-on results – segmented flood maps and detected settlement boxes – bring a satisfying conclusion, as they can visually see AI outputs overlayed on satellite images, which makes the power of these techniques tangible.

We take a moment to reflect on how these techniques can even be **combined** in practice. For example, one could first use object detection to find the approximate locations of objects (say, detect cars in a large image) and then apply segmentation to precisely outline each car or distinguish it from background. Or use detection to find buildings, then a segmentation model to get the exact building footprints. In disaster mapping, one might detect *damaged* buildings and also segment the debris extent around them, etc. Understanding all three tasks allows one to choose the right tool or mix of tools for a given problem.

Finally, we encourage participants to **continue exploring and applying** these methods. The field of AI for Earth Observation is rapidly evolving – new models (like transformer-based models we briefly saw) are emerging, and data is ever-growing. We’ve equipped them with the foundational knowledge: how CNNs learn from data, how to train and evaluate models, and how to approach typical EO challenges (from data preprocessing to class imbalance to interpretation of results). By maintaining consistency in our approach (each day blending theory modules with hands-on practice, and re-emphasizing key concepts like convolution, padding, upsampling across contexts), we've built a cohesive learning experience. The hope is that participants leave **empowered** – not just with some code recipes, but with an understanding of *why* these models work and *how* to adapt them to their own geospatial tasks. They are now ready to tackle their own challenges in disaster risk reduction, environmental monitoring, and beyond, using the advanced tools of semantic segmentation and object detection that they learned on Day 3.

We wrap up by inviting any final questions and discussions. What other EO problems do they have that might be addressed with these techniques? Perhaps segmenting burned areas after a forest fire, detecting illegal logging sites, mapping plastic debris in water via detection, etc. The possibilities are vast. The key takeaway: participants now have a strong foundation in modern AI as applied to EO – from concept to implementation – and are encouraged to keep practicing and learning. The journey doesn't end here, but they are well-prepared to continue it.

---

[1] [2] Segmentation vs Detection vs Classification in Computer Vision: A Comparative Analysis — Picsellia
https://www.picsellia.com/post/segmentation-vs-detection-vs-classification-in-computer-vision-a-comparative-analysis

[3] [4] [5] [6] [7] U-Net Architecture Explained - GeeksforGeeks
https://www.geeksforgeeks.org/machine-learning/u-net-architecture-explained/

[8] (PDF) Segmentation and Visualization of Flooded Areas Through Sentinel-1 Images and U-Net
https://www.researchgate.net/publication/379775578_Segmentation_and_Visualization_of_Flooded_Areas_Through_Sentinel-1_Images_and_U-Net

[9] CopPhil EO AI_ML Training Agenda - Final - 040725.docx
file://file_000000004c30620aa965644877a59b86

[10] [11] [12] 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

[13] General Steps for Sentinel-1 Data Preprocessing - Microwave Toolbox - STEP Forum
https://forum.step.esa.int/t/general-steps-for-sentinel-1-data-preprocessing/1907

[14] [15] [16] [22] One-Stage Object Detectors Explained | Ultralytics
https://www.ultralytics.com/glossary/one-stage-object-detectors

[17] End-to-End Detection Transformer (DETR) - - NeuralCeption -
https://neuralception.com/objectdetection-detr/

[18] Detecting Objects in Aerial Images (DOAI)
https://captain-whu.github.io/DOAI2019/cfp.html

[19] Enhanced Vehicle Detection and Segmentation Using the SAMRS ...
https://www.kjrs.org/journal/view.html?uid=1046&vmd=Full

[20] [21] Object Detection in Remote Sensing Images Using Deep Learning: From Theory to Applications in Intelligent Transportation Systems | Journal of Future Artificial Intelligence and Technologies
https://faith.futuretechsci.org/index.php/FAITH/article/view/114