ChatGPT

Session 4: Hands-on – Feature/Object Detection from Sentinel Imagery (Urban Monitoring Focus)

Case Study Overview: Informal Settlement Detection in Metro Manila

In this session, we will apply object detection techniques to a real-world scenario relevant to the Philippines: **monitoring informal settlement growth in Metro Manila**. Informal settlements (or slums) are densely built, unplanned communities often lacking official infrastructure and services [1]. These areas are highly vulnerable to hazards like floods, fires, or earthquakes [1] [2]. Keeping track of their expansion is crucial for disaster risk reduction and urban planning – authorities need up-to-date maps of where these communities are, to target interventions such as early warning systems, relocations, or infrastructure upgrades [3] [4]. However, monitoring them from the ground is challenging due to their rapid and sometimes hidden growth [5].

Our case study focuses on an area in Metro Manila (e.g. parts of Quezon City or along the Pasig River) known to have informal settlements [6]. We define our object of interest as **"building-like clusters that likely indicate an informal settlement"**, using detection of these clusters as a proxy for locating informal communities [6] [7]. Once detected, such data can reveal patterns: *Are settlements growing over time? How many homes are in a high-risk zone like a floodplain?* [8] [9]. This directly ties back to disaster risk reduction (DRR) – informal settlements often suffer disproportionate damage in disasters, being in hazard-prone zones with less resilient construction [10]. Mapping them allows agencies to plan risk reduction measures proactively (e.g. flood defenses or community relocation) [10]. It also relates to natural resource management (NRM) and urban planning, since these settlements impact land use and sometimes encroach on environmentally sensitive areas (like riverbanks or wetlands) [11].

Why Sentinel-2 imagery? We will use *Sentinel-2 optical satellite images* (10 m per pixel resolution) for this exercise [12]. At 10 m resolution, individual small houses are not directly visible as separate objects – a single 10 m pixel may cover several small structures [12] [13]. However, *larger clusters* of informal housing (tens of meters across) do appear as distinct urban patches in the imagery (often a gray or blue-gray blotch of rooftops amidst green vegetation in true-color images) [13] [14]. Ideally, one would use higher-resolution imagery (e.g. 3 m PlanetScope or sub-meter commercial images) to detect individual buildings or shanties [15]. But for our training session, Sentinel-2 provides a free, accessible, and manageable dataset. We frame the task as detecting **"built-up patches indicative of informal housing"** rather than pinpointing every house [15]. In essence, our model will learn to identify small clusters of built-up area (groups of bright roof pixels) against the background of vegetation or planned urban areas [16] [17]. This simplification makes the problem feasible given the data and time constraints, while still demonstrating the power of object detection for urban monitoring.

Object Detection vs. Classification vs. Segmentation

Before diving in, it's important to distinguish object detection from other computer vision tasks you've encountered (to avoid confusion between similar terminologies):

Figure: Illustration of different vision tasks. **Left:** Image Classification predicts an overall label for an image (e.g. identifying if **a** teddy bear is present or not, without locating it). **Middle:** Object Localization/Detection for a single object provides a bounding box around the object (e.g. finding **the** teddy bear and drawing a box). **Right:** Object Detection for multiple objects finds and classifies all instances (here, detecting both the teddy bear **and** the book, each with a bounding box and label). In **Semantic Segmentation** (not shown), every pixel is classified (e.g. every pixel of the teddy bear vs. background), producing precise shape masks rather than boxes. In **Instance Segmentation**, individual object masks are separated (e.g. separate masks for each object even if they are the same class).

In summary, **image classification** gives one label per image, **semantic segmentation** labels every pixel in the image (often grouping all objects of a class into one region), whereas **object detection** must both identify *what* objects are present and *where* they are in the image [18] [19]. Object detection outputs a set of bounding boxes with class labels, effectively combining localization and classification in one task [18]. For example, in a satellite image of a port, a classification model might simply output "port" or "no port" for the image; a segmentation model could color-code pixels as water, land, or ship (marking all ships in one color region); but an object detector would draw a box around *each* ship and label it as "ship" [20]. This distinction highlights why and when we use detection: it is essential for counting objects and pinpointing their locations (e.g. counting how many buildings are in an area, or finding coordinates of all ships in a harbor) [21]. In our case, a detector can locate each informal settlement cluster in a scene, instead of just telling us if one is present, or coloring an entire area as "built-up" without distinguishing separate clusters.

Understanding these differences is crucial as we proceed, especially since earlier sessions covered classification and segmentation. It ensures we interpret our model's output correctly: an object detector produces a list of boxes (with coordinates and labels) rather than a single label or a full pixel mask [22].

Platform and Data Setup

Environment: We will use **Google Colab** (with GPU enabled) as our development environment for this hands-on session. To minimize cognitive load and build on what you already know, we'll stick with the *same deep learning framework* used in previous sessions (TensorFlow or PyTorch, whichever you have been using so far) [23]. This consistency means you can reuse familiar code patterns – for example, the way we defined a model or set up a training loop earlier – rather than learning new syntax in the middle of the workshop [24]. Such continuity helps you focus on new concepts (object detection) instead of struggling with a new API or environment.

Data Provided: We have prepared a dataset of **Sentinel-2 image patches** covering parts of Metro Manila [25]. Each patch is a small image (for example, 256×256 pixels, which at 10 m resolution covers roughly a 2.5 km × 2.5 km area) [25] [26]. These patches include a variety of scenes: some contain known informal settlement areas, while others are non-settlement areas for contrast [27] [28]. The imagery is multi-spectral; however, for simplicity we might use only a subset of bands – for instance, an RGB composite or RGB + Near-Infrared (4 bands) – to provide the model with the most relevant information [29]. (Using the NIR band can be helpful since vegetation reflects strongly in NIR, making built-up areas stand out by contrast [29] [30]. Vegetated pixels will appear very bright in NIR, whereas man-made surfaces like roofs will appear darker, aiding the model in distinguishing urban vs. vegetated land [31] [32].)

Annotations: Alongside the images, we provide **bounding box annotations** for informal settlement clusters (and possibly other buildings) within those patches [33]. Each annotation entry gives the coordinates of a rectangle surrounding a cluster of interest, along with a class label. In our case, we essentially have a single object class (e.g. "settlement" or "building"), so each box is labeled with that class (and everything outside boxes is implicitly "background") [34]. For example, if one image patch covers a riverside barangay with informal housing, there might be ~5 bounding boxes, each encircling a dense group of small roofs that together form a slum cluster [35] [36]. These annotations could have been created manually by tracing the clusters, or derived from existing data (like known building footprints or previous surveys) [34]. The annotation format is chosen to be compatible with common object detection tools – it might be as simple as a CSV file of image names and box coordinates, or a COCO-style JSON, or a TFRecord, depending on the framework. We will not bog you down in the formatting details; the provided code will handle loading the annotations into the model's expected format [37].

Why not Sentinel-1 (radar)? The project Terms of Reference had suggested using Sentinel-1 SAR for built-up change detection. While radar data can indeed reveal urban areas (and is useful for change detection through time-series), interpreting SAR imagery can be challenging for beginners (due to speckle noise, and the lack of intuitive visual cues we have in optical images) [38]. We chose Sentinel-2 optical data for this hands-on because it's more straightforward to visually understand – you can literally see the communities and our model's boxes overlayed in a "photo-like" image, which makes the results intuitive to interpret [38] [39]. We will, however, mention to participants that object detection could also be applied to SAR or other data for advanced use cases (for instance, detecting ships in Sentinel-1 radar images, or using SAR change to flag new construction) [40]. But for our first foray into object detection, optical imagery is the more accessible route.

Hands-On Workflow Outline

We will follow a step-by-step workflow to train and apply an object detection model on our Sentinel-2 dataset. The process is simplified to fit in a 2.5-hour session, focusing on the key practical steps. We present two modeling approaches – **Option A** and **Option B** – but emphasize that **Option A (using a pre-trained model)** is the recommended path for most participants (it's faster and more feasible in a short session). **Option B (building a model from scratch)** is an optional, more advanced exercise for those who are curious and have time; it provides insight into how detection works under the hood, but it's not required to complete the session.

Workflow Steps:

1. **Load Images and Annotations:** Begin by loading the Sentinel-2 image patches and their corresponding bounding box annotations into your Colab notebook. This involves reading the images (e.g., as NumPy arrays or Tensors) and loading the annotation data structure (e.g., a list of boxes per image). We might have a utility function to do this. At this stage, familiarize yourself with the data: for a given image patch, you can plot it and draw the ground-truth boxes to see what an informal settlement cluster looks like in 10 m imagery (often a small cluster of grayish pixels) [13].

This helps in understanding the task visually. We ensure the data is split into a training set (for model training), and a test/validation set (for evaluating performance later).

2. **Select and Prepare an Object Detection Model:** Next, choose which object detection model approach to use. There are two options:

3. **Option A (Recommended) – Fine-tune a Pre-Trained Detector:** In this approach, we *leverage a model pre-trained* on a large generic dataset (for example, COCO, which includes common objects and might indirectly have learned features for buildings) [41] [42]. Using a pre-trained model is a form of **transfer learning**: the model has already learned to detect various objects in natural images, so it has useful feature detectors (for edges, textures, shapes) that we can adapt to our specific task [43]. Many libraries make this easy. For instance, in TensorFlow we could load a model from the TF *Model Zoo* or TensorFlow Hub (e.g. an SSD MobileNet or Faster R-CNN already trained on COCO) with just a few lines [44]. In PyTorch, we can use `torchvision.models.detection` (which provides pre-trained Faster R-CNN, SSD, etc.) or a lightweight YOLO implementation (e.g. YOLOv5 via a library) [42]. We will pick a **lightweight architecture** for speed – for example, SSD with a MobileNet backbone, or a small YOLO variant (like YOLOv5s) – something that can train quickly on Colab and still produce decent results [45] [46]. Once loaded, we **adjust the model's output layers** to our scenario. Since our dataset has only one object class ("settlement"), we configure the model to predict just two classes: "background" vs "settlement" (for example, by replacing the final classification layer to have 2 outputs, if using PyTorch's API) [47]. This way, the model will output only bounding boxes for our class of interest. By reusing a pre-trained model, we greatly reduce the training time and the amount of data needed – the model already *knows* a lot about detecting objects in general, so it just needs to learn to specialize to our specific object's appearance [43] [48]. This is a prime example of transfer learning in Earth Observation AI, where labeled data is often scarce: starting from pre-trained weights is almost always advantageous [48] [49].

4. **Option B (Advanced, Optional) – Implement a Simplified YOLO/SSD from Scratch:** This approach is for educational purposes if time allows, to demystify how object detection works internally. We **build a very simple one-stage detector** manually (akin to a toy version of YOLO). Key concepts include dividing the image into a grid and assigning responsibility for detections to grid cells [50]. For example, a 256×256 image might be divided into a 16×16 grid; each grid cell will predict whether an object exists in that cell and output a bounding box (or multiple boxes) for that cell [50]. We introduce **anchor boxes** (a few default box shapes/sizes per grid cell) – these serve as starting templates that the model tweaks to fit the objects [50] [51]. The model would predict adjustments to these anchors plus confidence scores and class labels. We also define a suitable **loss function** that penalizes errors in box coordinates, object confidence, and classification. Finally, during inference, we would apply **Non-Maximum Suppression** to eliminate redundant overlapping boxes [52] [53]. Fully implementing and training such a model from scratch is quite ambitious for a short session (it involves a lot of low-level detail and would train slowly), so this option would likely be exploratory. We might walk through pseudocode or implement just a **small portion** (say, one forward pass of a tiny network) to illustrate how predictions are encoded, rather than training a full model in class [54]. The goal here is to give interested participants a taste of the inner workings of detectors – e.g., how YOLO formulates detection as a regression problem on a grid – without expecting to reach the accuracy of a full-fledged detector in 2.5 hours. Most participants, especially those less comfortable with coding, can **skip Option B** and focus on Option A. The instructors will ensure Option A is fully working, while Option B is discussed conceptually or attempted only if time permits.

5. **Train (Fine-Tune) the Model:** After setting up the model (via Option A or B), we proceed to train it on our dataset. For Option A (pre-trained model fine-tuning), training will be relatively quick. We feed in our training images with their annotated boxes, and run the training loop for a small number of epochs (since the model only needs to adjust to our data). For example, even ~5–10 epochs might be enough to see significant learning, given the model's prior knowledge [55] [56]. During training, the model's loss will decrease as it gets better at predicting our settlement boxes. We'll monitor the process (e.g., watching loss values or printing intermediate evaluation metrics if available). If using a high-level API (like Keras or a Torch trainer), we might not see per-iteration details, but we can observe overall progress. In training from scratch (Option B), if attempted, we would likely train only a very limited number of iterations or on a subset, due to time constraints, just to demonstrate the concept. Either way, ensure that the training is running without errors and the model is learning something (e.g., the loss goes down).

6. **Evaluate Model Performance:** Once we have a trained model, we evaluate how well it performs at detecting informal settlements on unseen data (our validation/test set). We will run the model on several test patches and compare the predictions to the ground truth annotations. For a quantitative evaluation, we introduce the standard **object detection metrics**, primarily **mean Average Precision (mAP)** [57]. In essence, the model's predicted boxes are matched against true boxes to compute precision and recall, and the Average Precision (AP) summarizes the precision-recall curve for our class. Since we have only one class, the mAP (mean AP across classes) is effectively the same as the AP for the "settlement" class [58]. We won't delve into heavy math during the session, but conceptually we explain that AP is the area under the precision-recall curve, and mAP is the average of APs for all classes [59]. In practical terms, you can interpret the score: for instance, an AP of 0.5 (50%) indicates the model finds about half of the targets on average (or quality of detections is moderate), whereas an AP of 0.9 would mean it's nearly perfect at the given IoU threshold [60] [61]. If implementing the full metric calculation is complex, we may simplify by reporting precision and recall at a fixed IoU threshold (e.g., IoU > 0.5 to count a detection as correct) as a proxy. The main idea is to assess: *Does the model detect most of the clusters (high recall)? Does it avoid too many false alarms (precision)?* We will discuss the results in plain language, e.g., "The model correctly found X% of informal settlement areas in our test images, with Y% precision, meaning that out of all detections Y% were actual settlements" [57] [62]. This helps relate the numbers back to our goal.

7. **Visualize Detected Results:** This is the fun part – we will visualize the model's predictions on the test images to see how it performs qualitatively [63]. For a few test patches, we overlay the predicted bounding boxes on the satellite image (perhaps drawing them in a distinct color with the model's confidence score) [63]. We can also overlay the ground-truth boxes for comparison. This visual check gives immediate insight into the model's strengths and weaknesses: for example, we might see that it successfully outlines the larger clusters of tin roofs (informal houses) along the river, but maybe it **missed** a very small cluster tucked under trees, or it **confused** a bright industrial rooftop for a settlement (a false positive) [63] [64]. We might observe multiple overlapping boxes before the final output – recall that the detector may predict several candidate boxes for one object; the **Non-Maximum Suppression** step will have filtered those so only one box is shown per cluster [65]. We'll explain that this is why we see one clean box per object rather than many overlapping ones. The visualization step is not just for eye-candy – it's an important part of the validation because it allows us (as humans) to qualitatively assess if the detector is focusing on the right areas. Seeing the model "in action" – drawing boxes around what it thinks are settlements – is often very satisfying and solidifies the understanding of object detection for participants.

Throughout this workflow, we will keep an eye on runtime. Using a pre-trained model (Option A) ensures that training is feasible within the session (likely on the order of minutes). Option B, if attempted, will be limited to a demonstration due to time. The instructor will guide you through coding steps in the Colab notebook, but you are encouraged to *actively participate* – try changing a parameter, examine intermediate outputs, and ask questions whenever something isn't clear. This hands-on practice will help reinforce the concepts introduced.

Key Concepts and Terminology

During the session, we will revisit and clarify some important concepts to ensure everyone grasps the underlying ideas of object detection, especially as they differ from prior tasks:

- **Classification vs. Detection vs. Segmentation:** We touched on this earlier, but to reinforce: *Image classification* gives an image-level label (no location info), *object detection* finds *instances* with bounding boxes, and *semantic segmentation* assigns a class to every pixel (producing regions, often without distinguishing individual instances) [18] [19]. If you were confused between detection boxes and segmentation masks, remember that detection outputs discrete boxes per object, whereas segmentation would color the entire area of the object. We may show another simple visual or example to cement this difference [66] [67].

- **Anchor Boxes:** For those new to detection, the concept of anchor (prior) boxes can be perplexing. In intuitive terms: *Instead of allowing the model to propose any arbitrary box from scratch, we define a few default boxes of preset sizes/aspect ratios at each grid location, called anchor boxes.* The model then learns to adjust (shift/scale) these anchors to better fit the object and to predict a confidence score for each [51]. This provides a helpful starting point – the model doesn't begin with a blank slate for localization; it has some "anchors" to work with, which makes learning more efficient. For example, at a grid cell covering a certain area, we might have three anchors: one large rectangle, one medium, one small. If the object is small, the model will mostly tweak the small anchor; if the object is larger, it might use the large anchor, etc. We could draw a grid on an image and illustrate a couple of anchor boxes to demonstrate how an object might overlap with one of them [68] [69]. The take-home point: anchors are like **priors** that guide the model's bounding box predictions.

- **Non-Maximum Suppression (NMS):** This is a post-processing step used in virtually all object detectors. The model often predicts multiple boxes for the same actual object – especially because adjacent anchors or grid cells overlap. For example, a single settlement cluster might trigger several nearby anchors to each predict a box around it. NMS is an algorithm that **filters out redundant detections** [70] [65]. It works by selecting the highest-confidence prediction for a particular object and eliminating other boxes that overlap heavily with that selection (typically using the IoU – Intersection over Union – overlap measure). The result is that you get one clean detection per object rather than many overlapping boxes [65]. In many frameworks, NMS happens behind the scenes when you call the model's inference function. We explain this so you're aware: if you saw the raw outputs pre-NMS, you might see a cluster of, say, 5 boxes all around the same informal settlement, each with a slightly different position. After NMS, only the best one remains. Understanding NMS helps in interpreting why the model might output, say, *only one box* for a cluster even if multiple anchors detected it – the rest were pruned. We can even conceptually show a before-and-after to illustrate the effect [71] [72].

- **Detection Performance Metric – mAP:** Unlike classification (accuracy) or segmentation (IoU/Dice), object detection uses **mean Average Precision (mAP)** as the primary metric. We break this down in simple terms. **Precision** in detection is "of all boxes the model predicted, what fraction were correct?" and **recall** is "of all true objects, what fraction did the model detect?". The AP (Average Precision) combines precision and recall across different confidence thresholds into a single number (area under the precision-recall curve) [73] [74]. **mAP** is the mean of AP across all classes. In our single-class case, AP for "settlement" is equal to mAP [58]. We won't manually calculate a precision-recall curve here, but we'll conceptually explain that an AP of, say, 0.8 is quite good (the model finds most settlements with few false positives), whereas an AP of 0.3 would indicate a lot of missed detections or many false alarms. As mentioned, we may simplify evaluation by using a fixed IoU threshold for true/false positive and give you precision/recall values or an F1 score, depending on what's easier to compute live. The key is understanding what the numbers mean in practical terms for our task. If we report, for example, "Precision 0.9, Recall 0.6 at IoU 0.5", you should interpret that as "90% of detections were correct, but it only found 60% of all actual settlements" – indicating high precision, moderate recall.

- **Transfer Learning Benefits:** One big lesson from this session is *how quickly we achieved a working detector by using transfer learning*. Training an object detector from scratch on our small dataset would likely be impractical (it would require many images and a long training time to learn good features). By starting from a model pre-trained on a large dataset, we saw that in a short time we fine-tuned it to our specific task [43] [75]. This approach – using pre-trained weights whenever possible – is a **key strategy** in deep learning applied to Earth observation [49] [76]. The models we used were originally trained on everyday photos (e.g., COCO has pictures of dogs, cars, people), not satellite images. Yet, they provided a strong head start for our task. This might seem surprising, but it highlights that the low-level features these CNNs learn (edges, corners, textures) are quite general [77] [78]. Detecting a building in a satellite image is not fundamentally different in terms of edges and shapes than detecting a house in a street photo – so the pre-trained model's knowledge transfers well. We only had to teach it the "higher-level" specifics of what informal settlements look like from above [79] [80]. This concept of **cross-domain transfer learning** is very powerful in remote sensing AI, where labeled data in the target domain (satellite images) may be limited. Keep this in mind for your own projects: often the fastest path to success is to fine-tune an existing model rather than to reinvent the wheel.

Wrap-Up and Takeaways

By the end of this session, you have **built (or fine-tuned) an object detector and applied it to real satellite imagery of a local urban issue**. This is a significant achievement – in just a short time, we went from raw Sentinel-2 pixels to a model that can automatically highlight informal settlement areas on a map. Visually seeing those bounding boxes pinpoint clusters of makeshift housing in the images is a rewarding moment, as it demonstrates the AI's capability to extract meaningful information for societal applications. It also brings our three-day journey to a full circle: we've now covered the trifecta of deep learning for Earth observation – image classification (Day 2), semantic segmentation (earlier on Day 3), and object detection (this session) [81]. You've been exposed to how each technique can be used to turn satellite data into insights: from identifying whether a feature is present, to delineating its shape, to locating and counting multiple instances.

In real-world applications, these methods can even be combined – for example, one could first use object detection to find the locations of buildings, and then apply segmentation to delineate the precise footprint of each building, achieving instance segmentation. Or detect ships in an image and then segment the water vs. land to understand context. The possibilities are many, and you now have a foundational understanding to explore them [82] [83].

Lastly, we encourage you to **reflect and look forward**: think of other problems in disaster risk reduction, environmental monitoring, or urban planning that these advanced techniques could help solve. Perhaps automatically detecting *landslide scars* in mountains (segmentation), or *counting solar panel installations* across a city (detection), or *classifying crop types* from multispectral imagery (classification). With the knowledge gained here, you are empowered to continue experimenting and learning. We hope this hands-on experience, aligned with solid conceptual grounding (and not overloading you cognitively by following a step-by-step, scaffolded approach), has been both educational and inspiring. Keep practicing – the more you apply these skills, the more confident you'll become. And remember, whenever approaching a new problem, break it down into manageable steps, leverage pre-existing tools and models when available, and gradually build up complexity – this way, you'll maximize learning without feeling overwhelmed. Happy exploring in the world of geospatial AI!

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [75] [76] [77] [78] [79] [80] [81] [82] [83] Day 3_ Advanced Deep Learning – Semantic Segmentation & Object Detection.pdf
file://file_00000000626461f5969b1c65ddc5964a

[68] [69] [70] [71] [72] [73] [74] Day 3_ Advanced Deep Learning – Semantic Segmentation & Object Detection.pdf
file://file_000000001fd861f5933b737832065f78