ChatGPT

# Day 2: Advanced AI/ML for Earth Observation – Classification & CNNs

Welcome to **Day 2** of the CopPhil AI/ML for Earth Observation training. Today's focus is on advanced classification techniques and an introduction to deep learning for EO. Participants will build on their prior ML and EO experience to tackle supervised classification and Convolutional Neural Networks (CNNs) in practical Earth Observation scenarios. We will expand four key sessions into comprehensive ~3-hour modules with theory, hands-on labs, quizzes, and real-world datasets. Each module emphasizes applications in **Natural Resource Management (NRM) in the Philippines**, preparing you for upcoming modules on semantic segmentation and time series analysis.

## Module 1: Theory and Practice of Supervised Classification using Random Forest for EO

### Introduction and Objectives

Supervised land cover classification is a foundational task in Earth Observation, enabling the creation of maps that distinguish forests, water, agriculture, urban areas, and more. In the context of NRM in the Philippines, such classifications help track deforestation, monitor crop lands, and plan sustainable land use. For example, annual land cover maps allow stakeholders to **track urban expansion and its impact on natural habitats**, supporting sustainable development and conservation efforts [1]. In this module, we revisit supervised classification concepts and dive into the **Random Forest (RF)** algorithm, a powerful ensemble method widely used for EO data. We'll discuss why RF is popular for remote sensing (robustness, high accuracy, ability to handle many input features), and demonstrate how to apply it to satellite imagery. By the end of this module, you will understand how RF classifiers work and how to train and evaluate a model for land cover mapping.

**Learning Outcomes:** By the end of Module 1, participants will be able to:
- Explain the concept of supervised classification and how it applies to Earth Observation data.
- Describe the Random Forest algorithm and its advantages for classifying remotely sensed imagery.
- Outline the workflow for training an EO classification model (data collection, training, prediction, validation) [2].
- Understand how to assess classification accuracy (e.g. confusion matrix, accuracy metrics) and why this is important [3].
- **(Practical)** Interpret a simple Random Forest classification example on EO data (e.g., identify how changing parameters like number of trees affects results).

### Theoretical Concepts

**Supervised Classification in Earth Observation:** In supervised classification, the goal is to assign each observation (e.g. a pixel or an image patch) a label (class) based on input features. In EO, features are

typically spectral reflectance values from satellite bands (and possibly indices or ancillary data), and classes might be land cover types (forest, water, urban, etc.). "Supervised" means we require **training data**: examples where the class is known, used to train the model. In practice, training samples can come from ground truth surveys, existing maps, or manual labeling in tools like Google Earth Engine (GEE). The classification workflow generally involves: **(1)** collecting training data with class labels and predictor features, **(2)** choosing and configuring a classifier algorithm, **(3)** training the classifier on the labeled data, **(4)** using the trained model to classify the entire image or dataset, and **(5)** evaluating the results with independent validation data [2]. Key considerations include ensuring class labels are encoded properly (e.g. numeric codes in GEE), scaling or normalizing features if needed, and avoiding biased or overlapping samples. For EO data, supervised classification enables creation of thematic maps critical for NRM – for instance, mapping forest cover vs. agriculture to monitor land use change.

**Random Forest Algorithm:** Random Forest is an ensemble learning method that constructs a "forest" of many decision trees and aggregates their votes to make a final prediction. Each decision tree is trained on a random subset of the data and a random subset of features, which introduces diversity. The ensemble voting scheme typically yields higher accuracy and stability than a single tree, and it mitigates overfitting. RF is **non-parametric** (makes no strong assumptions about data distribution) and can handle high-dimensional data well – useful for satellite imagery which may have many spectral bands and derived indices. It's also relatively straightforward to train and tune, often achieving good classification results with minimal parameter tweaking [4]. Key hyperparameters include the number of trees (more trees generally improve performance up to a point), the number of features considered for splitting at each node, tree depth limits, etc. One great feature of RF is it can provide **feature importance** measures, indicating which bands or indices were most useful for the classification [5] [6]. In remote sensing, Random Forest has become one of the most popular classifiers due to its strong performance even with limited training data and its robustness to noisy labels and outliers [4]. It is **commonly applied for land cover classification and other remote sensing image analysis tasks** [7], making it an ideal choice for our initial models.

**Applying RF to Earth Observation Data:** When using RF for EO classification, we typically treat each pixel (or object/region) as one sample with multiple features. For a satellite like Sentinel-2, features could be reflectance values in various bands (e.g. Blue, Green, Red, NIR, SWIR, etc.), as well as derived indices (NDVI, NDWI, etc.) or auxiliary data (elevation from SRTM, slope, etc.). Including additional data often improves classification; for example, combining Sentinel-2 imagery with terrain data and even nighttime lights data can significantly boost land cover mapping accuracy [8]. Studies have found that using multi-temporal composites (e.g. monthly or seasonal median images rather than a single date) yields better performance, since it captures phenological changes and reduces cloud impact [9]. In practice, an EO classification with RF would involve preparing a composite image (to handle clouds and seasonality), gathering training points for each class, and then using an RF implementation (e.g. `ee.Classifier.smileRandomForest` in GEE or `sklearn.ensemble.RandomForestClassifier` in Python) to train on those points. It's important to use **independent validation** to assess performance: a common approach is splitting your sample data into a training set and a validation set (e.g. 80/20 split) before training [10]. After training, you can apply the classifier to the whole image to get a land cover map, and compute a confusion matrix (error matrix) comparing predicted vs. true labels for the validation set [11]. Accuracy metrics derived from this (overall accuracy, per-class accuracy, kappa, etc.) tell you how well the model is performing [3]. If accuracy is low for certain classes, you might need more training data, better features, or to adjust RF parameters. For example, increasing the number of trees or adding NDVI band as a feature could improve separation of vegetated classes. Overall, Random Forest offers a solid balance of ease-of-use and performance for EO tasks and will be a baseline for more advanced methods.

### Practical Demo: Random Forest Classification Workflow

*In this section, we walk through a simple example to illustrate how a Random Forest classification is set up. This is a precursor to the full hands-on exercise in Module 2.*

**Demo Scenario:** Suppose we want to classify land cover in a small region (e.g. a part of Luzon, Philippines) into basic classes: Water, Vegetation, and Built-up. We have a Sentinel-2 image composite of the area from the dry season (to minimize cloud cover). We will use Google Earth Engine for this demo due to its convenience with satellite data. The steps are:

- **1. Data preparation:** Define a region of interest (ROI) and retrieve a Sentinel-2 surface reflectance image or composite for a given date range. Apply cloud masking (using the QA60 band in Sentinel-2) and possibly select key bands. For example:

```
var roi = /* user-defined geometry for ROI */;
// Load Sentinel-2 SR images for Jan-Feb 2023 and filter clouds
function maskClouds(img) {
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var qa = img.select('QA60');
  // Keep pixels with both cloud and cirrus bits = 0
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0)
               .and(qa.bitwiseAnd(cirrusBitMask).eq(0));
  return img.updateMask(mask);
}
var s2 = ee.ImageCollection('COPERNICUS/S2_SR')
           .filterDate('2023-01-01', '2023-02-28')
           .filterBounds(roi)
           .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 20))
           .map(maskClouds);
var composite = s2.median().clip(roi);  // median composite
```

Here we loaded Sentinel-2 Level-2A data and made a median composite over two months to reduce clouds. The composite image will be used for classification. (In a real case, we might also add an NDVI band or include SRTM elevation as extra features to help differentiate land cover types).

- **2. Collect training samples:** Next, we need training data. Let's assume we have some reference data or we draw polygons for known land cover types. For demonstration, one might use the ESA WorldCover map 2020 (which has global land cover classes) as a source of labels, or manually create points. For simplicity, we'll use a small set of hand-picked sample points (in a real exercise, more systematic sampling would be done). For example:

```
// Example: using WorldCover as training label source
var landcover = ee.Image('ESA/WorldCover/v100/2020').select('Map').clip(roi);
// Remap WorldCover classes to simpler 0=Water, 1=Vegetation, 2=Built-up
var remapLC = landcover.remap(
```

```
    [40, 90, 50],   // water(40), urban(50), vegetation (90) class codes in
WorldCover
    [0, 2, 1]       // new labels 0,1,2
).rename('class');
// Sample 50 points per class within ROI
var trainingSample = composite.addBands(remapLC).stratifiedSample({
    numPoints: 50,
    classBand: 'class',
    region: roi,
    scale: 10,
    seed: 42,
    geometries: true
});
```

In this snippet, we use an existing land cover map to get sample labels: water, vegetation, built-up (urban). We then perform a **stratified random sampling** to pick 50 points of each class from within the ROI, ensuring our training set is balanced. Each sample captures the composite's band values and the class label. *(Note: In practice, if high-quality ground truth isn't available, participants might draw polygons over known areas to create training points using the GEE GUI. This requires time but ensures understanding of training data collection.)*

- **3. Train the Random Forest classifier:** With the `trainingSample` FeatureCollection ready, we configure a Random Forest. We might start with, say, 100 trees. In GEE, `ee.Classifier.smileRandomForest(n_trees)` returns an untrained classifier object. We then call `.train(features, classProperty, inputProperties)` on it:

```
var classifier = ee.Classifier.smileRandomForest({numberOfTrees: 100}).train({
    features: trainingSample,
    classProperty: 'class',
    inputProperties: composite.bandNames()
});
```

This trains the RF model on our samples. The result `classifier` now contains the trained model (decision trees). We can output some info like the out-of-bag error estimate or feature importance if needed (e.g. `print('Feature Importance:', classifier.explain().importance)` in JavaScript API).

- **4. Classify the image:** We apply the trained model to the composite image to generate a classified map:

```
var classified = composite.classify(classifier);
```

The result is a single-band image where each pixel has a class ID (0,1,2 corresponding to Water, Vegetation, Built-up in our scheme). We can add this to the GEE map with a color palette to visualize, or export it for further use.

- **5. Accuracy assessment:** It's crucial to evaluate how well the classifier performed. If we set aside some validation data, we would use that. In our example, we could split the `trainingSample` into 80% used for training and 20% for validation before training (as was done in the code example above with `randomColumn` in the sample dataset [10]). Alternatively, since we drew from an existing map, we could compare the classified result to that map. A confusion matrix can be computed in GEE by comparing the `classified` image to the reference labels on a set of sample points:

```
var validation = trainingSample;  // (for demonstration, using same sample for
validation; ideally use separate data)
var validated = validation.classify(classifier);
var testConfusionMatrix = validated.errorMatrix('class', 'classification');
print('Confusion Matrix:', testConfusionMatrix);
print('Overall Accuracy:', testConfusionMatrix.accuracy());
```

This would print a confusion matrix table and overall accuracy. Suppose we got an overall accuracy of 85% with most confusion between Built-up and Bare soil classes (which we merged into others) – this tells us our model is fairly good but might need improvement. We might increase training samples or refine features. **Accuracy assessment using a confusion matrix is crucial to gauge classifier performance** [3], and it guides us in refining the model.

Through this demo, we saw the end-to-end process of using RF for EO classification. In Module 2, you will perform a more in-depth **hands-on lab** tackling a real-world land cover classification problem, applying these same steps yourself.

## Quiz & Assessment

Test your understanding of the concepts from Module 1 with the following questions:

1. **Conceptual:** What is the difference between *supervised* and *unsupervised* classification in the context of EO data? Why do we need training data for the former?
2. **Random Forest Theory:** Random Forest combines multiple decision trees. How does this ensemble approach improve classification stability and accuracy compared to a single decision tree?
3. *A.* It uses deeper trees to capture all variations.
4. *B.* It averages results from many trees to reduce overfitting.
5. *C.* It uses a single best tree selected via cross-validation.
6. **Answer:** B – By averaging votes from many trees (each trained on random subsets), RF reduces overfitting and improves generalization.
7. **Data & Features:** You have multispectral imagery and want to classify mangroves vs. other land cover. Name two additional features or data sources you might include (besides the raw spectral bands) to improve the classification, and explain why.
   *(Expected answer: Potential features include NDVI (to highlight vegetation greenness), water indices (to distinguish water), texture measures, or ancillary data like elevation (mangroves occur in coastal low*

*elevation zones). Including these can help the classifier separate classes that might be spectrally similar in raw bands.)*

8. **Practical:** In GEE, why do we need to *remap* class values to consecutive integers (0,1,2,...)? What could go wrong if we use arbitrary class codes in the classifier training?
   *(Expected: GEE's classifier expects class labels as 0-indexed integers; non-consecutive or large code values might cause errors or inefficient model training.)*
9. **Evaluation:** If your RF model achieves 95% accuracy on training data but only 70% on validation data, what does that indicate and what steps could you take?
   *(Expected: This indicates overfitting – the model memorized training samples but generalizes poorly. Steps: gather more training data, reduce model complexity (fewer trees or shallower trees), ensure features are relevant, or use techniques like cross-validation.)*

### Summary

In Module 1, we covered the theory and practice of supervised classification with a focus on Random Forests. **Key takeaways:** Supervised classification turns raw EO data into meaningful categorical maps using labeled examples – a process highly relevant for NRM tasks like land cover mapping and forest monitoring in the Philippines. Random Forest is a powerful, user-friendly algorithm for this, offering high accuracy and robustness for mapping complex landscapes. We reviewed the end-to-end workflow: data preparation, model training, prediction, and validation [2]. Crucially, we emphasized the need for accuracy assessment (confusion matrices, etc.) to ensure the model's reliability [3]. By mastering RF classification, you've gained a baseline method to produce land cover maps that, for example, help **monitor key habitats and land use changes over time** [1]. In the next module, we will put this knowledge into action with a full hands-on lab using Sentinel-2 data for land cover classification in an NRM context.

---

## Module 2: Hands-on Land Cover Classification (NRM Focus) using Sentinel-2 in GEE/Python

### Introduction and Objectives

This module is a comprehensive **lab session** where you will apply supervised classification to create a land cover map, focusing on a Natural Resource Management scenario in the Philippines. Building on Module 1, we'll use **Sentinel-2 imagery** and possibly auxiliary data to classify land cover types (such as forest, agriculture, water, built-up, etc.) in a region of interest. The case study could be, for example, classifying land cover in a province or watershed that is important for biodiversity or resource management. Accurate land cover maps are vital – for instance, the Philippines has lost *1.42 million hectares of tree cover from 2001 to 2022* due to various drivers (urban expansion, agriculture, illegal logging) [12]. Maps derived from classification can help identify where such deforestation is happening and inform reforestation or protection efforts. In this hands-on module, you will go through the full workflow yourself, using either **Google Earth Engine (JavaScript API)** or a Python environment (e.g. Jupyter/Colab with libraries like rasterio, numpy, and scikit-learn). We will emphasize best practices like cloud masking, feature engineering (adding indices), training sample collection, classifier training, and accuracy assessment. The lab will illustrate how these techniques directly support NRM decision-making in the real world.

**Learning Outcomes:** After completing Module 2, participants will be able to:
- Set up an environment for satellite data processing (GEE Code Editor or Python in Google Colab) and load

Sentinel-2 imagery for a specified region/time.
- Perform image preprocessing steps such as cloud masking and creating composites (e.g. median or seasonal composites to handle clouds and capture phenology).
- Create or utilize training data for land cover classes, either by drawing sample polygons/points or using existing labeled datasets.
- Train a supervised classifier (Random Forest) on the prepared data and apply it to classify an image into a thematic map.
- Evaluate the classification result with validation data (confusion matrix, accuracy) and interpret the outcomes (identify which classes have confusion, discuss possible improvements).
- Understand and articulate how the resulting land cover map can be used in NRM applications (e.g. forest cover monitoring, urban planning, agricultural management in the Philippine context).

## Lab Setup and Data Preparation

**Choosing the Tool:** You may choose to conduct this lab in **Google Earth Engine (GEE)** or in Python. We recommend GEE for its simplicity in accessing Sentinel-2 and its built-in classifiers, especially if internet connectivity is stable. GEE also has an interactive map to visualize results. Python is an alternative if you prefer more control or integration with other data sources; however, you'd need to acquire the imagery (e.g. via Sentinel Hub, AWS, or GEE Python API) which can be time-consuming. For this training, instructions will focus on GEE Code Editor, with notes on how one could do similarly in Python. If using Python, a Google Colab notebook with pre-downloaded sample data will be provided to save time (with packages: `geemap` (for GEE access), `rasterio`, `numpy`, `sklearn`, etc.).

**Selecting Area of Interest (AOI):** First, decide on a study area relevant to NRM in the Philippines. This could be a region like **Sierra Madre** (for forest conservation), **Pantabangan-Carranglan Watershed** (forest and water resource management), or a province heavily used for agriculture. Ensure the AOI is not too large (to keep data manageable) – for example, an area of ~50km x 50km. Define this AOI in your code (as a polygon or by importing a boundary shapefile).

**Data Collection – Sentinel-2 Imagery:** Using Sentinel-2 **Level-2A (surface reflectance)** data is ideal because it's pre-corrected for atmospheric effects. We will retrieve images for a specific time window. If doing an annual land cover map for 2023, you might take imagery from an entire year and create a composite. However, for simplicity and time, we can use a **seasonal composite** or a few representative months. Given the Philippines' cloudiness, a common approach is to use dry season months for clearer images. For example, use **December 2022 to March 2023** images to build a composite for "2023 land cover". In GEE, this looks like:

```
var aoi = /* your AOI geometry */;
var start = '2022-12-01';
var end   = '2023-03-31';
var s2 = ee.ImageCollection('COPERNICUS/S2_SR')
           .filterBounds(aoi)
           .filterDate(start, end)
           .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
           .map(maskS2clouds);  // Apply cloud mask function (same as in Module
1 demo)
```

```
var composite = s2.median().clip(aoi);
Map.addLayer(composite, {bands: ['B4','B3','B2'], min:0, max:3000}, 'Sentinel-2
composite');
```

Here we filtered images by location and date, limited cloud percentage, and applied a cloud mask. We then take the median across images to get a representative composite. This reduces cloud contamination and averages out temporal changes, giving a clearer view of persistent land cover. Optionally, you can also add indices as new bands (e.g. NDVI, NDWI) to help the classifier. For example:

```
var addIndices = function(img) {
  var ndvi = img.normalizedDifference(['B8','B4']).rename('NDVI');
  var ndwi = img.normalizedDifference(['B3','B8']).rename('NDWI');
  return img.addBands([ndvi, ndwi]);
};
var composite_idx = addIndices(composite);
```

We now have an image ready for classification, with spectral bands and some indices. You could also merge other data like **SRTM elevation**: `var elev = ee.Image('USGS/SRTMGL1_003').clip(aoi); composite_idx = composite_idx.addBands(elev.rename('elev'));` – adding elevation can help, for instance, to distinguish high-elevation forests from lowland agriculture.

### Creating Training Data

Now, prepare training data for the classes of interest. Decide on a set of land cover classes relevant to NRM. For example, you may choose: **Forest, Water, Built-Up, Cropland, Grassland/Shrub, Bareland** (these cover major categories – you can adjust depending on the AOI's landscape). There are two main ways to get training samples: - **Manual Digitization:** Draw polygons on the map for areas you know represent each class (e.g. draw a few polygons over dense forest areas, water bodies, urban areas, etc.). Then use these as training regions to sample the underlying image. This can be done with the GEE drawing tools and `sampleRegions` function. - **Existing Maps:** Use an existing land cover map or dataset as reference. For example, the **ESA WorldCover 2020** map or a national land cover dataset (if available) can provide class labels. You can sample points from this map within the AOI. The GEE example in Module 1 demo showed how WorldCover's class codes can be remapped to a simpler scheme and then stratified sampling used [13] [14].

For this lab, we suggest a guided approach using a provided reference if available (to save time). Assume we provide a simplified land cover truth for 2023 for the area (this could be a small sample or a raster map prepared beforehand). If using WorldCover 2020 as a proxy, be aware it might not perfectly align with 2023 imagery, but it's a starting point.

**Example (using WorldCover for training):**

```
// Load ESA WorldCover 2020 (10m land cover map)
var lc2020 = ee.Image('ESA/WorldCover/v100/2020').select('Map').clip(aoi);
// Define a mapping from WorldCover classes to our target classes
```

```
// (e.g., 10=Tree cover->Forest, 20=Shrub->Grassland, 30=Grassland->Grassland,
40=Cropland->Cropland,
// 50=Built-up->Built, 60=Bare->Bare, 70=Snow/ice->Bare (not likely in PH),
80=Water->Water, 90=Wetlands->Water (or separate Wetland class), 95=Mangroves-
>Forest, 100=Flooded veg->Wetland)
var wcClasses = [10,20,30,40,50,60,70,80,90,95,100];
var myClasses = [0, 1, 1, 2, 3, 4, 4, 5, 5, 0, 5];
// (This mapping is just an example: 0=Forest, 1=Grass/Shrub, 2=Cropland,
3=Built, 4=Bare/Other, 5=Water/Wetland)
var lcMap = lc2020.remap(wcClasses, myClasses).rename('class').toByte();
// Stratified sampling of 500 points (about ~100 per class if evenly
distributed)
var trainingPoints = composite_idx.addBands(lcMap).stratifiedSample({
    numPoints: 500,
    classBand: 'class',
    region: aoi,
    scale: 10,
    seed: 30,
    geometries: true
});
print('Training points count per class:',
trainingPoints.aggregate_histogram('class'));
```

This will produce a FeatureCollection of training points with the band values and class label. We also printed a histogram to see how many samples per class we got (if some classes are rare in the AOI, you may get fewer points for them; ensure you have at least a reasonable number for each class or adjust `numPoints` or use `classWeights` parameter in stratifiedSample).

If you opted to draw polygons manually, you would do something like: draw polygons for each class, convert them to FeatureCollection, assign class labels, then use `sampleRegions` on the composite image to get samples.

### Training the Classifier and Classification

With training data in hand, training the Random Forest is straightforward. We will use GEE's RF (which by default uses the Smile RF implementation). Choose the number of trees – for example, 50 or 100 trees is a good start (more trees can improve accuracy but also increase computation time). We'll also set the `seed` for reproducibility.

```
var rf = ee.Classifier.smileRandomForest({
    numberOfTrees: 100,
    seed: 42
}).train({
    features: trainingPoints,
    classProperty: 'class',
```

```
    inputProperties: composite_idx.bandNames()
});
```

If the training set is large or has many features, training may take some seconds. After training, we apply the model to classify the composite:

```
var classifiedMap = composite_idx.classify(rf);
```

Now `classifiedMap` is our result image. We can define a visualization palette for the classes (e.g., forest=dark green, grass=light green, cropland=yellow, built-up=red, bare=gray, water=blue, etc.) and add to map:

```
var palette = ['006400','7fffd4','ffd700','ff0000','d3d3d3','0000ff'];  //
example colors for classes 0-5
Map.addLayer(classifiedMap, {min:0, max:5, palette: palette}, 'Land Cover
Classified');
```

Take a look at the map. Does it make sense? Forested areas should show up in the forest color, water in blue, etc. You might notice some obvious misclassifications – for example, perhaps some cloud shadows got classified as water or bare land (this is common if cloud masking wasn't perfect or shadows were interpreted as dark water). Or some urban areas might be confused with bare soil. These are points to analyze and discuss improvements.

**Assessment of runtime**: If the AOI is moderately sized, the classification should run quickly. If using Python with scikit-learn, you would have had to export the composite's bands as numpy arrays along with labels, then fit `RandomForestClassifier(n_estimators=100)`. The results should be similar, but GEE handles large images more gracefully.

### Accuracy Assessment

We need to quantitatively assess the classification accuracy. Ideally, we have **validation data** independent of our training. In a rigorous project, we'd set aside some of our labeled points as a test set or have an entirely separate ground truth dataset. Since this is a training exercise, one approach is to split our stratified sample into train/test by random column (like 80/20) before training. If you did not do that, another strategy is to compare with the reference map (WorldCover or other) and compute confusion, keeping in mind the reference might have errors itself.

To illustrate accuracy assessment, let's assume we did a split when sampling:

```
// Split into training and validation (if not already split)
var sampleWithRandom = trainingPoints.randomColumn('rand', 0);
var trainSet = sampleWithRandom.filter(ee.Filter.lt('rand', 0.8));
var testSet  = sampleWithRandom.filter(ee.Filter.gte('rand', 0.8));
```

```
print('Train size:', trainSet.size(), 'Test size:', testSet.size());

// Train on trainSet (instead of full trainingPoints)
var rf_model = ee.Classifier.smileRandomForest(100).train({
    features: trainSet,
    classProperty: 'class',
    inputProperties: composite_idx.bandNames()
});
// Classify testSet and get confusion matrix
var testPredictions = testSet.classify(rf_model);
var confusionMatrix = testPredictions.errorMatrix('class', 'classification');
print('Confusion Matrix:\n', confusionMatrix);
print('Overall Accuracy:', confusionMatrix.accuracy());
```

GEE will output a confusion matrix (rows = actual class, cols = predicted class) and overall accuracy. Inspect this matrix to see which classes are often confused. For example, you might find forest vs. shrubland have some mix-up (common if spectral difference is small, or if mixed pixels), or built-up vs bare soil might confuse each other. Note the per-class accuracies by normalizing the confusion matrix.

If overall accuracy is, say, 85%, that's a decent result for a first pass. Discuss with participants: which classes are least accurate and why? Perhaps *Grassland vs Cropland* were confused because both might just appear as generic vegetation without temporal info; including multi-season imagery could help differentiate crops (which get harvested) from perennial grass. Or *Mangroves (if present) vs Forest* might confuse if we lumped them together or if spectral difference is small – adding a coastal mask or a SAR band could help in that case. This thought process highlights how to improve the model.

**Tip:** It's also instructive to examine variable importance (if using scikit or the `.explain()` in GEE). If NDVI was among features, is it highly ranked? If elevation was added, see if it helped separate classes (e.g., built-up might concentrate at low elevations near cities, etc.). Understanding feature importance can guide adding or removing features.

Finally, once satisfied, you can export the classification map (GeoTIFF) for use outside GEE or to share with stakeholders.

*Example output map: The figure above shows an example land cover classification result for an area in Central Luzon, Philippines (Pampanga River delta). Different colors denote classes (e.g., green for forests/vegetation, yellow for croplands, red for built-up areas, blue for water). Such maps support NRM by indicating the distribution of land cover types; for instance, one can easily spot areas of expanding agriculture (yellow) or urban development (red) encroaching natural habitats. In this example, one could track changes year by year to see trends in land use.*

### Real-World Relevance to NRM

After producing your map, consider how it can be used. In the Philippines, accurate land cover maps feed into many applications: forest authorities use them to identify remaining forest pockets and prioritize protection in regions with high deforestation rates [12]; urban planners use them to see how built-up areas have expanded and plan infrastructure accordingly; agricultural agencies use crop land classifications to

estimate cultivation areas and manage food security; environmental agencies overlay land cover maps with biodiversity data to find critical habitats that need preservation. In our example output, if large areas that used to be forest are now classified as cropland or built-up, that signals land use change, which might warrant action or further monitoring. The ability to **observe land use evolution over time is fundamental to sustainable development and conservation** [1] – and the skills you practiced in this module are directly enabling that.

## Quiz & Assessment

Reflect on the lab with these questions:

1. **Data Prep:** Why did we choose a multi-month composite image for classification instead of a single Sentinel-2 scene?
   *A.* To reduce the data volume for processing.
   *B.* To minimize cloud cover and capture average reflectance, improving class separability.
   *C.* Sentinel-2 cannot be used for single scenes.
   **Answer:** B – Compositing over multiple dates reduces cloud effects and captures phenological differences, aiding classification [15].
2. **Indices & Features:** How did adding NDVI and NDWI bands likely help our classification? Which classes might these indices particularly assist in separating?
   *Answer:* NDVI highlights vegetated areas, helping to distinguish vegetated classes (forest, crops) from non-vegetation (urban, bare, water). NDWI highlights water or moisture, aiding detection of water bodies or wetlands. For example, water vs. shadow confusion can be reduced with NDWI, and crops vs. bare soil can be better separated with NDVI.
3. **Training Data:** If one land cover class in your AOI had very few training samples (e.g., wetlands), what strategies could improve the classifier's handling of that class?
   *Answer:* You could collect more samples for that class (targeted sampling), possibly use oversampling or synthetic samples to balance the classes, or apply class weighting in the classifier to not overlook the minority class. Additionally, ensure features that characterize that class (e.g., a water index for wetlands) are included.
4. **Accuracy:** You find that "Built-up" areas were often misclassified as "Bare soil" in your result. Give two possible reasons and how you might address them in a future iteration.
   *Expected points:* (a) Spectral similarity – built-up (concrete) and bare soil both have high reflectance in certain bands; solution: incorporate texture or temporal data (built-up remains constant, while soil might change with seasons), or use high-resolution data for urban areas. (b) Insufficient training data – maybe our built-up samples didn't cover all urban types (rooftops, roads); solution: add more diverse urban samples. Also, adding nighttime lights or infrastructure data as a feature could help separate urban surfaces.
5. **Tool Use:** For those who used Python: what are the challenges of doing this classification outside of GEE, and what advantages does GEE offer? Conversely, what flexibility might Python offer over GEE?
   *Discussion points:* GEE simplifies data access and processing large images in the cloud, avoiding local download. It also has built-in functions (e.g., `ee.Classifier`) for quick modeling. Python, while requiring data management (download or API usage) and possibly slower for large images, allows use of custom algorithms, integration of local datasets, and greater control over model parameters or different classifiers (e.g., XGBoost, etc.). Ideally, one can use GEE for rapid prototyping and Python for specialized modeling as needed.

### Summary

In Module 2, you completed a hands-on land cover classification, reinforcing the concepts learned earlier. **Key achievements:** you loaded and preprocessed Sentinel-2 data, engineered useful features (cloud-free composite, NDVI/NDWI, etc.), and trained a Random Forest to produce a land cover map. You also evaluated the map's accuracy and discussed its implications for NRM in the Philippines. The resulting classification provides valuable information – for example, identifying areas of deforestation, extent of agricultural lands, and urban growth – which is foundational for resource management and environmental planning [1] [12]. You've learned not only the "how" of classification but also the "why" – seeing how each step contributes to a reliable output that stakeholders can trust. This hands-on experience prepares you for the next step: leveraging more advanced AI techniques. In **Module 3**, we pivot to **Deep Learning and Convolutional Neural Networks**, which can further improve and automate feature extraction for tasks like classification (and later segmentation). Keep this module's lessons in mind, as many principles (data preparation, feature importance, validation) remain crucial even as models become more complex.

---

## Module 3: Introduction to Deep Learning and CNNs for Earth Observation

### Introduction and Motivation

Traditional machine learning (like the Random Forest used earlier) relies on human-defined features and has limitations in capturing complex patterns in imagery. **Deep Learning**, and specifically Convolutional Neural Networks (CNNs), have revolutionized how we analyze images by automatically learning hierarchical features directly from raw data [16]. In this module, we introduce the core concepts of deep learning with a focus on CNNs, and discuss how they apply to Earth Observation tasks. We'll bridge from familiar ideas (like image classification) to how CNNs perform these tasks more powerfully. This will set the stage for the hands-on CNN exercise in Module 4. We will also emphasize use cases in EO relevant to NRM – for example, using deep learning to detect subtle patterns such as disease stress in crops, identify buildings or roads in satellite images for urban planning, or classify coral reef health from high-resolution imagery. By understanding CNN fundamentals, participants will be equipped to harness these methods for more advanced applications like **semantic segmentation** (per-pixel classification, to be covered on Day 3) and **time-series analysis** (Day 4, combining temporal patterns with deep learning).

**Learning Outcomes:** By the end of Module 3, participants will:
- Grasp the basic concept of deep learning and how it differs from traditional ML.
- Understand the architecture of Convolutional Neural Networks (CNNs) and key components (convolution layers, activation functions, pooling, fully-connected layers).
- Recognize why CNNs are well-suited for image data and how they automatically learn features (edges, textures, objects) from pixel values.
- Be aware of common CNN architectures and terms (e.g. kernels/filters, strides, ReLU, CNN layers like Conv/Pool, etc., and examples like VGG, ResNet in context).
- Learn about typical EO applications of CNNs: image-level classification, object detection, and segmentation, with examples pertinent to NRM (e.g. land use classification, detecting deforestation or infrastructure, mapping habitats).
- Appreciate the requirements of deep learning – need for larger datasets, computational resources (GPUs),

and how techniques like transfer learning can help in EO where labeled data may be limited.
- Understand the difference between **image-level classification vs. pixel-level classification** and how CNNs can do both (image-level directly, and with modifications for pixel-level/segmentation) [17].
- Get introduced to the frameworks and tools (TensorFlow, Keras, PyTorch) that make building CNNs feasible, and how these integrate with EO data (e.g., using libraries to handle geospatial imagery).

## What is Deep Learning? (vs Traditional ML)

Deep learning refers to machine learning methods based on artificial neural networks with many ("deep") layers. Unlike a Random Forest which uses decision trees on manually chosen features, a deep neural network learns multiple levels of representation automatically. For images, a deep network can start from pixel values and learn low-level features like edges in early layers, then assemble those into higher-level features like shapes or object parts in deeper layers. The model's complexity allows it to capture very intricate patterns, given enough data. **Key point:** Deep learning often outperforms classical methods when there is a lot of training data, because it can exploit subtle variations and interactions in the data that are hard to capture with hand-crafted features.

However, deep learning models have many parameters and thus usually require **large datasets and significant computational power (especially GPUs)** to train. In the context of Earth Observation, we now have big data (e.g., decades of satellite imagery), which makes deep learning attractive. Cloud computing and libraries like TensorFlow and PyTorch have made it easier to train models on this data. Another important concept is **transfer learning** – using a model pre-trained on a large dataset (often on generic images like ImageNet or on a large EO dataset) and fine-tuning it on your specific task. This is particularly useful in EO where labeled data might be scarce for a specific region or class; a pre-trained CNN already learned general image features and can quickly adapt to remote sensing data with fewer samples.

## CNN Fundamentals: How Convolutional Neural Networks Work

**Why Convolutions?** Images have spatial structure – pixels close together form meaningful features (e.g. an edge or texture). A CNN is designed to take advantage of this by using **convolutional layers** that apply small filters (kernels) across the image to detect local patterns. Each convolutional layer produces feature maps: for example, the first conv layer might highlight edges in various orientations, the next might detect shapes or textures by combining edges, and so on. This hierarchical feature learning is powerful for imagery.

**Layers in a CNN:** A typical CNN architecture for image classification has a stack of layers:

- **Input layer:** The raw image data (for EO, this could be multi-band; CNNs can handle multiple channels, e.g., 3-channel RGB or even 13-band Sentinel-2, but often we might start with RGB or selected bands due to pre-trained models expecting 3 channels).
- **Convolutional layers:** Each convolutional layer uses a set of learnable filters (e.g., a 3x3 or 5x5 pixel window) that slide over the input and compute dot-products, producing feature maps. Each filter is like a detector for a certain pattern (one might activate on vertical edges, another on a green vegetation texture, etc.). The network learns the filter values during training.
- **Activation functions:** After each convolution, a non-linear activation (like **ReLU**, Rectified Linear Unit) is applied, which introduces non-linearity (so the model can learn complex patterns). ReLU simply zeroes out negative values, keeping positive signals.

- **Pooling layers:** Periodically, a pooling layer (e.g., max pooling) reduces the spatial resolution by taking a summary (max or average) over a region (like 2x2). Pooling makes the representation more abstract and invariant to small shifts (important for recognizing features anywhere in the image). It also reduces computation for subsequent layers.
- These conv + activation + pooling layers may repeat several times, each time capturing higher-level features and reducing image size. For example, an input 64x64 could be reduced to 32x32 with certain features, then 16x16, etc., as we go deeper.
- **Fully Connected (Dense) layers:** Towards the end, the CNN might flatten the feature maps and use one or more dense layers to combine features and make the final classification. However, modern architectures often use global pooling and avoid too many dense layers to reduce parameters.
- **Output layer:** For classification, a dense layer with softmax activation gives class probabilities (e.g., if we have 10 classes, a vector of length 10 with probabilities summing to 1).

During training, the CNN adjusts all those filter weights via backpropagation, trying to minimize classification error on the training set. The result is a model that can generalize to new images, ideally.

**Why CNNs excel at image tasks:** CNNs learn *what to look for* in images by themselves. In our earlier RF approach, we fed spectral bands and indices which is fine, but RF couldn't consider spatial texture easily beyond what indices might encode. A CNN on a pixel neighborhood can learn that "a cluster of bright pixels in a linear arrangement" might indicate a road, or "a roundish green shape with a shadow" could be a tree crown, etc., which a per-pixel classifier would miss. This makes CNNs particularly good for high-resolution imagery or complex pattern recognition (like identifying rooftops, cars, or disease patches in crops). Indeed, **deep learning has revolutionized analysis of satellite and aerial imagery, addressing challenges like vast image sizes and diverse object classes** [16] .

**Example Use Cases in EO:**
- *Scene Classification:* Assigning a label to an entire image (or patch), e.g., classifying a 256x256 px Sentinel-2 patch as "Forest" or "Urban". This is similar to what you'll do with EuroSAT in Module 4. CNNs have achieved high accuracy on such tasks, often surpassing classical methods, especially when there are subtle textures or combinations of features indicative of classes. - *Object Detection:* Finding and classifying objects within an image (e.g., detecting ships in harbors, finding illegal logging sites, counting houses for population estimates). CNNs (with architectures like YOLO or Faster R-CNN) can be trained to output bounding boxes around objects of interest. For NRM, an example is detecting *fish pens in lakes* or *small boats* in coastal waters for fisheries management, or detecting *burn scars* in forests after fires. - *Semantic Segmentation:* Classifying each pixel in an image (we will cover this in detail in a later module). This produces a detailed map much like our Module 2 output, but CNN-based segmentation (with models like U-Net or SegNet) can often produce more detailed and accurate maps by considering texture and context around each pixel. This is useful for delineating things like precisely mapping coral reef extents or identifying individual crop fields. Keep in mind, **image-level classification vs. pixel-level classification differ in output granularity** – CNNs can do both: image-level (one label per image/patch) and pixel-level (label per pixel) when structured appropriately [17] . - *Other advanced tasks: Change detection* (CNNs to compare images from two dates and find changes), *Super-resolution* (enhancing image resolution), *Data fusion* (combining radar and optical data in a network), etc. Deep learning is a rapidly evolving field in EO.

### CNNs in Practice: Frameworks and Training Considerations

To use CNNs, we rely on libraries like **TensorFlow/Keras or PyTorch**. These libraries handle the heavy math (matrix operations on GPUs) and provide high-level abstractions to define layers easily. For example, in Keras you can define a simple CNN like:

```
from tensorflow.keras import layers, models
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(64,64,3)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

This would be a small CNN for classifying 64x64 RGB images into 10 classes. In Earth Observation, you might plug in multi-band images; if using a pre-trained model, often you use just RGB or pan-sharpen images due to model expectations, or use transfer learning weights from a network trained on similar data (there are some pre-trained on EO datasets).

**Training Data needs:** CNNs typically need a large number of examples. In our Module 4, the EuroSAT dataset has 27,000 images which is decent. If you don't have thousands of samples, you might not want to train a CNN from scratch – that's when transfer learning is helpful (e.g., take a ResNet model pre-trained on millions of images, then fine-tune on your small EO dataset). We will actually mention this in Module 4 as a possible extension.

**Overfitting & Validation:** Just like any model (even more so here), monitoring performance on a validation set during training is vital to avoid overfitting. Techniques like data augmentation (random flips, rotations of images) are often used in training CNNs to artificially expand the dataset and make the model robust to variations. For satellite imagery, augmentations could include rotations (since orientation shouldn't change class) or slight color jitter.

**Computational aspect:** If training on a CPU, CNN training can be very slow. For Module 4, we will use Google Colab with GPU enabled, which significantly speeds up training. Participants should be aware that training large models or big datasets might require cloud GPU instances or local GPU machines.

**Interpreting CNNs:** Deep learning models are sometimes called "black boxes" because it's not as straightforward to interpret which feature caused a certain classification. However, tools like class activation maps or simply visualizing filters can provide some insight. Also, CNNs can output probabilities, which help gauge confidence in predictions.

**Ethical/Practical note:** Always remember that a model is only as good as the data – biases in training data will reflect in the model. For example, if all "urban" training images are from one city with white roofs, the

model might not recognize urban areas with different building materials. So we need diversity in training data and careful evaluation.

## EO and NRM Applications of CNNs

Now let's tie CNN capabilities to NRM examples in the Philippines: - **Forestry:** A CNN could be trained on high-resolution satellite or drone images to identify individual trees or classify forest types (e.g., CNN segmentation can differentiate mangrove species or detect illegal logging clearings in a dense forest). - **Agriculture:** Deep learning models can classify crop types from multi-temporal satellite data (combining CNN for spatial info and maybe recurrent layers for temporal). For instance, identifying rice vs. corn fields from satellite time series helps in crop area estimation and food security planning. - **Disaster Management:** After a typhoon, CNN-based damage assessment (detecting damaged buildings in imagery) can prioritize response. CNNs can also assist in flood mapping from SAR or optical images quickly by segmentation. - **Marine and Coastal:** Mapping coral reefs or seagrass meadows using CNNs on multispectral imagery can support marine resource management. The **coastal habitat monitoring** pilot in CopPhil likely uses CNNs/Segmentation on Sentinel-2 or Planet imagery to classify benthic habitats (coral, sand, algae) for reef health monitoring. - **Urban and Infrastructure:** For rapidly urbanizing areas, CNNs can identify new construction, road networks, or informal settlements from aerial images, information valuable for urban planning and disaster risk reduction (knowing where people live in flood-prone zones, etc.).

In summary, CNNs open up a wide array of advanced applications in EO by improving accuracy and extracting more information than classical methods. They are a key technology behind many modern remote sensing applications (from Google's AI-powered land cover maps to academic research on climate change indicators).

## Quiz & Thought Questions

1. **Conceptual:** What distinguishes a convolutional neural network from a regular fully-connected neural network on an image task?
   *Answer:* CNNs use convolutional layers that exploit spatial locality and weight sharing (the same filter applied across an image), whereas a fully-connected network treats each pixel independently as input weights. This makes CNNs far more efficient and effective for images since they learn local patterns (like shapes) rather than unrelated pixel-by-pixel parameters.
2. **CNN Layers:** In a CNN, what is the role of a pooling layer? What might happen if we removed pooling layers entirely?
   *Expected:* A pooling layer reduces spatial resolution and aggregates features, providing translation invariance and reducing overfitting by decreasing parameters. If we remove pooling, the network retains full resolution throughout, leading to very high dimensional feature maps and potentially overfitting, as well as much higher computation and memory usage. Some modern architectures replace pooling with strides in convolution, but the idea of progressively reducing resolution remains.
3. **Application:** Why might a CNN be better than a Random Forest for identifying **"coconut plantation"** versus **"natural forest"** in high-resolution imagery?
   *Answer:* A coconut plantation has a regular tree spacing and pattern that a CNN can recognize (textural pattern of tree crowns in a grid), whereas a natural forest is more irregular. A per-pixel RF

using spectral data might see both as just “green vegetation” and struggle to differentiate without spatial context. CNNs analyze texture and arrangement, capturing the plantation’s signature pattern.

4. **Data Requirements:** You have only 100 labeled images of a certain habitat type from satellites – what strategy could you use to still train a deep learning model effectively?
   *Answer:* Use transfer learning: take a CNN pre-trained on a large dataset (perhaps ImageNet or a similar land cover dataset) and fine-tune it on the 100 images. Also apply data augmentation (rotations, flips, color jitter) to get more mileage out of the 100 images. This way the model’s earlier layers, which already learned to detect edges and textures, can adapt to the new data with far fewer samples than training from scratch.
5. **Looking Ahead:** How do you think a CNN could be adapted to perform **semantic segmentation** (classifying each pixel) instead of whole-image classification? (This foreshadows tomorrow’s module.)
   *Expected discussion:* By using a fully convolutional approach – instead of ending in a dense layer for a single label, one can have the CNN output a map of class probabilities for each pixel. Architectures like U-Net accomplish this by an encoder-decoder structure: an encoder CNN compresses the image into feature representations, and a decoder upsamples those features back to the original image size, outputting a class label per pixel. We will learn more about this in the segmentation module.

### Summary

Module 3 provided a broad introduction to deep learning and CNNs, emphasizing their relevance to Earth Observation. We learned that **CNNs automatically learn spatial features** from imagery, enabling more powerful classification and detection capabilities than traditional methods. Key points include the structure of CNNs (convolutions, pooling, etc.) and why they are effective for images. We also discussed how CNN-based approaches are increasingly used in EO tasks ranging from land cover classification to object detection and segmentation, often yielding state-of-the-art results [16]. In terms of NRM in the Philippines, CNNs can enhance our analyses – from more accurately mapping land cover (distinguishing plantations vs. natural forests, or identifying small water bodies), to detecting changes and events important for resource management (like new illegal mining areas or coral bleaching events via image analysis). With this conceptual groundwork, you are now ready to get hands-on with a CNN in Module 4, where we’ll build and train a model for satellite image classification using a real dataset (and you’ll see these concepts in action).

---

## Module 4: Hands-on CNN for Image Classification using EuroSAT (Satellite Imagery)

### Introduction and Objectives

In this final module of Day 2, we transition from theory to practice with deep learning. You will train and evaluate a Convolutional Neural Network on an Earth Observation dataset – specifically, the **EuroSAT land use land cover dataset**, which consists of Sentinel-2 image patches. While EuroSAT contains European scenes, its classes (e.g. various crop types, urban, water, forest) are quite relevant to NRM contexts globally, including the Philippines (e.g., forests, agriculture, and water bodies exist in both contexts). This exercise will solidify your understanding of CNNs by having you preprocess data, define a model, train it, and assess its performance. It will also expose you to the practical aspects of using deep learning frameworks and handling EO data in that context. By doing this on a curated dataset, we avoid needing thousands of local labels – but keep in mind, the skills learned can be applied to your own datasets (with transfer learning, etc.). We’ll use Python (TensorFlow/Keras or PyTorch) in a Jupyter/Colab environment for this lab, leveraging

GPU acceleration. After completing this module, you'll not only have a working land cover classifier but also be prepared to tackle more advanced tasks like the upcoming **semantic segmentation** and incorporate temporal data for sequence models.

**Learning Outcomes:** By the end of Module 4, participants will:
- Become familiar with the EuroSAT dataset and its land cover classes, understanding how it parallels land cover issues in NRM (e.g., distinguishing different agricultural lands, urban areas, etc.).
- Learn how to load and preprocess image data for CNN training (including splitting into train/validation sets, normalizing pixel values, and data augmentation techniques to improve model generalization).
- Define a CNN architecture using a deep learning framework (we'll walk through a simple architecture, and mention how to use a pre-trained model for comparison).
- Execute the training process, monitoring training and validation accuracy/loss, and preventing overfitting (using the validation set and possibly early stopping).
- Evaluate the trained model on a test set: compute overall accuracy, per-class accuracy, and observe where the model performs well or poorly.
- Interpret the results and discuss how such a model could be used in practice (e.g., automated labeling of satellite images for mapping efforts), as well as limitations (what happens if applied to Philippine data without retraining?).
- Get introduced to the idea of transfer learning by optionally trying a pre-trained model (if time permits), to see the boost in performance and efficiency.
- Build confidence in using modern AI tools (Colab, Keras/PyTorch) on EO data, setting the stage for more complex modeling in later days.

### Dataset Overview: EuroSAT Land Cover Classification

**EuroSAT** is a public benchmark dataset for land cover classification, derived from Sentinel-2 imagery. It consists of 27,000 image patches (64 x 64 pixels each) covering 13 spectral bands of Sentinel-2, labeled into 10 classes [18]. The classes are: **Annual Crop, Permanent Crop, Pasture, Forest, Herbaceous Vegetation (like natural grasslands), Highway (road/path), Residential (urban buildings), Industrial (commercial/industrial buildings), River, and Sea/Lake**. These are clearly relevant to NRM – e.g., distinguishing crop types and vegetated land vs. built environment vs. water.

For our purposes, we will use the RGB version of the dataset (3-band images) for simplicity and since many pre-trained CNNs expect 3 channels. (Using all 13 bands is possible but would require a custom model from scratch; using RGB allows us to leverage architectures and weights from computer vision.) The dataset is already split into a training and testing set in some versions, but we can also create our own split. Typically, one might use ~80% of the images for training, 10% for validation, 10% for test.

**Note on dataset access:** In Colab, we can use `tensorflow_datasets` to load EuroSAT, or use a prepared archive. We have provided the dataset in the environment (for example, images in folders by class) to simplify. If using `tfds`:

```
import tensorflow_datasets as tfds
dataset = tfds.load('eurosat/rgb', split='train', as_supervised=True)
```

This would give a TensorFlow Dataset of (image, label) pairs. Alternatively, we might have a zip of images to manually load using `ImageDataGenerator` in Keras or PyTorch's `ImageFolder`. For clarity, we'll outline using Keras high-level API.

### Data Preparation

**1. Loading Data:** If using Keras, one convenient method is:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_dir = 'path/to/eurosat/train'  # organized by class subfolders
val_dir   = 'path/to/eurosat/val'
test_dir  = 'path/to/eurosat/test'

# Create an ImageDataGenerator for data augmentation
train_datagen = ImageDataGenerator(rescale=1/255.0, rotation_range=10,
width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
val_datagen   = ImageDataGenerator(rescale=1/255.0)

train_generator = train_datagen.flow_from_directory(train_dir, target_size=(64,
64), batch_size=32, class_mode='categorical')
val_generator   = val_datagen.flow_from_directory(val_dir, target_size=(64,64),
batch_size=32, class_mode='categorical')
```

Here we rescale pixel values to [0,1] (original Sentinel-2 reflectance was scaled to 0-255 in the RGB images). We also apply some augmentations to the training data: small rotations, shifts, and flips to make the model more robust. The `flow_from_directory` assumes a directory structure where each class has its own folder (which we have set up). The generators will yield batches of images and one-hot encoded labels.

**2. Splitting**: If the dataset isn't already split, we must split it. We could randomly shuffle and allocate 80/10/10. TensorFlow Datasets actually labels them all as 'train' in EuroSAT, but they expect users to create a split. Suppose we did that beforehand.

**3. Understanding Classes:** Ensure the class indices match known labels. The generator's `class_indices` attribute will show a dict mapping class names to numeric IDs. For reference:

```
print(train_generator.class_indices)
```

Might output: `{ 'AnnualCrop':0, 'Forest':1, 'HerbaceousVegetation':2, 'Highway':3, 'Industrial':4, 'Pasture':5, 'PermanentCrop':6, 'Residential':7, 'River':8, 'SeaLake':9 }` (or similar). Keep this for interpreting results later.

### Model Definition

We will start with a simple custom CNN to prove we can train from scratch. Given our input is 64x64 color images, a model architecture could be:

- Conv layer (16 filters, 3x3 kernel) + ReLU
- Conv layer (16 filters, 3x3) + ReLU
- Pooling (max pool 2x2)
- Conv layer (32 filters, 3x3) + ReLU
- Conv layer (32 filters, 3x3) + ReLU
- Pooling (2x2)
- Flatten -> Dense(64 units, ReLU) -> Dense(10 units, softmax)

This is a fairly small network suitable for demonstration. Alternatively, we could use a pre-built model like a small pre-trained MobileNet and fine-tune it, but that might hide some complexity from learning perspective. We'll do the simple one first.

Keras code for the model:

```
import tensorflow as tf
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(64,64,3)),
    tf.keras.layers.Conv2D(16, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

We compile with Adam optimizer and categorical crossentropy (since it's multi-class classification). The `summary()` will print the layer structure and number of parameters, which helps ensure the model isn't too huge.

**Explanation:** We use two Conv layers before each pool to allow the network to learn more features at each resolution. The number of filters doubled from 16 to 32 after the first pooling, as is common to increase feature maps as spatial size reduces. The final dense layer outputs 10 class probabilities. This model has on the order of tens of thousands of parameters – which is fine for 27k images.

### Model Training

We train the model using our generators:

```
history = model.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator
)
```

We might train for ~10 epochs (with 27k images, batch 32, that's about 840 steps per epoch). On a GPU this should be a few minutes. We monitor the `history` for training vs validation accuracy to watch for overfitting. Ideally, validation accuracy will climb and perhaps level off or drop if we overfit. If we had more time/compute, we could train more epochs or implement early stopping. For now, we watch: say after 10 epochs we get something like 90% training accuracy and 85% validation accuracy – indicating a well learning model that's not grossly overfitting.

If the model was underfitting (train acc low), we might increase complexity or epochs. If overfitting (val acc much lower than train and gap increasing), we might need more augmentation or fewer parameters or early stopping.

**Tip:** We can plot the loss and accuracy curves from `history` to illustrate the training process (though in a text document we just describe it). For example: - Epoch 1: train acc 60%, val acc 58% (learning basic patterns) - ... - Epoch 10: train acc 92%, val acc 87% (model has learned quite a lot; slight gap indicating some overfit but not severe).

### Model Evaluation

After training, we evaluate on the test set (the images not seen in training or validation):

```
test_datagen = ImageDataGenerator(rescale=1/255.0)
test_generator = test_datagen.flow_from_directory(test_dir, target_size=(64,64),
batch_size=32, class_mode='categorical', shuffle=False)
loss, accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {accuracy*100:.2f}%")
```

Suppose the test accuracy comes out around 85%. That is quite good for a 10-class satellite image problem. We can delve deeper: - Compute a confusion matrix to see per-class performance. We can use `model.predict(test_generator)` to get predictions and then compare with true labels from `test_generator.classes`. Then create a confusion matrix. - Likely observation: classes like **Forest vs Herbaceous** might sometimes confuse (both are "green" land covers); **Residential vs Industrial** might confuse if both show buildings (differences could be subtle like density or roof types); **River vs SeaLake** could confuse if the patch is small water vs big water body – though EuroSAT presumably distinguishes by context. - We check per-class accuracy. Perhaps we find: - Forest: 90% accurate (distinct in many cases), - AnnualCrop vs PermanentCrop: could be tricky if only using one date and RGB – might confuse with each

other or with Pasture, - Highway: might be easier as it has a strip of gray road usually, but could be small in patch so maybe moderate accuracy, - etc.

If we have time, we can demonstrate **transfer learning**: load a pre-trained model like VGG16 or ResNet50 (pretrained on ImageNet), replace final layer with 10 classes, and fine-tune on EuroSAT. This often would yield higher accuracy (maybe >90%) even with fewer epochs, thanks to learned features. However, doing so might be too advanced for this short module unless participants are already comfortable. We can mention it as an extension: *"If this were a production scenario, you might use a larger pre-trained CNN to boost performance. For example, using a ResNet50 pretrained on ImageNet and fine-tuning it on EuroSAT can push accuracy higher with fewer epochs, as shown in research [19] (transfer learning is very effective for EO data)."*

## Using the Model & Interpretation

Now that we have a trained model, how could it be used? One idea: feed it new Sentinel-2 image patches to automatically classify land use. For instance, you could slide a 64x64 window across a larger image (or use a pooling scheme) to generate a land cover map – this becomes more like segmentation when done densely. In fact, one could use this model to initialize a segmentation model or to tag large images with presence of certain classes.

In an NRM context, an automated classifier could speed up mapping. Imagine an organization needs to map all the **rice paddies vs. other crops** across the country. A model trained (or fine-tuned) on local data could examine new images and label regions quickly, alerting where changes occur (like new areas being cultivated or urbanized). It could also assist non-experts by providing a first-cut map that an analyst then cleans up, drastically reducing manual effort.

**Limitations:** If we take our EuroSAT-trained model and apply it directly in the Philippines, we must be cautious. The spectral characteristics or scenery might differ (e.g., different building materials, different crop types than in Europe). The model might misclassify if those differences are significant. Ideally, we'd retrain or fine-tune the model on some local samples for best performance.

## Wrap-up Discussion

Compare this deep learning approach with the classical one in Module 2: - The CNN learned features automatically, whereas the RF relied on NDVI/NDWI we fed. The CNN's internal filters might have effectively learned something akin to NDVI anyway (e.g., one filter could detect "greenness"). - The CNN considered spatial patterns – for instance, a highway patch has a unique linear feature which CNN can detect, whereas an RF on pixel values might not identify that without specialized feature engineering. - Training the CNN took more data and compute, but once trained, it might achieve better accuracy on certain nuanced classes. - We saw the importance of having a validation set to monitor training, which is analogous to what we did with RF but even more critical here to manage overfitting. - The deep learning workflow introduces new steps like tuning learning rates, deciding epoch counts, etc., which require practice.

Participants should now feel more comfortable with both worlds – using a tried-and-true machine learning method (RF) and a modern deep learning method (CNN). Both have their place in an EO analyst's toolbox. Often, a project might start with simple methods to get quick results, and then move to deep learning for improved accuracy or additional capabilities (like processing raw images without manual feature extraction).

## Quiz & Assessment

1. **Training Process:** During CNN training, you notice the validation loss stopped improving after 5 epochs, but you let it run to 15 epochs and the model started to overfit. What technique could you use to address this during training?
   *Answer:* Implement **early stopping** – monitor the validation loss and stop training when it ceases to improve for several epochs. This prevents overfitting by not over-training on the training data.
2. **Augmentation:** Give two examples of data augmentation applied in this module and explain why augmentation is important for training CNNs on limited EO data.
   *Expected:* Examples – random rotations, flips, shifts, brightness changes. Augmentation creates varied versions of the training images, helping the model generalize better to new data (e.g., a satellite image might appear at different orientations or slight differences in lighting; augmentation prepares the model for that). It effectively increases the training dataset size without needing new labeled images.
3. **Result Interpretation:** If our model has 95% accuracy on most classes but only 70% on the "River" class, what might be the reason? What could we do to improve the model for that class?
   *Answer:* Possibly the "River" class has less training data or is inherently confusing (maybe some river images look similar to "SeaLake" or are narrow and hard to recognize in small patches). Rivers might also often be accompanied by surrounding vegetation making them less visually distinct. To improve, we could gather more river examples, or include multi-spectral bands (like SWIR which highlights water), or use a larger model that might capture the subtle cues better. We could also check if the label is defined clearly (maybe the class includes small streams which are truly hard to detect at 10m resolution).
4. **Generalization:** Our model was trained on European Sentinel-2 imagery. If we want to use a CNN for classifying land cover in the Philippines, what steps should we take?
   *Answer:* We should fine-tune or train a model on Philippine data. That means collecting a local dataset (or using a global one that includes tropical environments) – could use transfer learning with the EuroSAT model as a starting point and then feed labeled Philippine scenes to adapt it. Also ensure any unique classes or features in the Philippines (like rice terraces, nipa huts, etc.) are accounted for in the training data so the CNN can learn them. Essentially, domain adaptation is needed for the model to generalize to a different region.
5. **Connection to Segmentation:** If you slide a 64x64 window across a large satellite image and classify each patch with your CNN, you get a coarse segmentation. How is this different from a true segmentation CNN (like U-Net), and why might the latter be more accurate or efficient?
   *Answer:* The sliding window approach classifies each patch independently, which is computationally intensive (lots of overlapping calculations) and doesn't guarantee consistency at patch boundaries. A U-Net or segmentation CNN processes the whole image (or large tiles) and produces a per-pixel map, considering context in a more seamless way. It can learn features at multiple scales and yields a smoother, more precise map without blocky artifacts. It's also more efficient because it avoids redundant computation on overlapping patches (thanks to shared convolution results). We'll explore this in the semantic segmentation module next.

## Summary

Module 4 was a deep dive into implementing a CNN for EO data. Congratulations – you have trained a deep learning model to classify land cover from satellite imagery! We covered the full pipeline: data loading, augmentation, model building, training, and evaluation. The EuroSAT exercise demonstrated how a CNN can achieve high accuracy (on the order of 85-90%) on a complex 10-class problem, reflecting the power of

learned features. We also discussed how these methods apply to real-world NRM tasks and what considerations are needed to adapt models to new regions or improve specific class performance. With this hands-on experience, you are well-prepared to tackle more advanced deep learning applications. In the subsequent days, we will build on this knowledge: moving to **semantic segmentation** (so you can create pixel-wise maps, not just classify patches) and exploring **time series analysis** (to utilize temporal dynamics from EO data). The skills and understanding from today – from Random Forest to CNNs – form a strong foundation for those upcoming topics. Keep this material handy as you progress, and always remember to connect these technical capabilities back to their purpose: supporting evidence-based Natural Resource Management and environmental protection in the Philippines and beyond.

---


1 12 Introducing the CopPhil Pilot Services: Improving Hazard and Environmental Monitoring in the Philippines - CopPhil
https://copphil.philsa.gov.ph/blog/introducing-the-copphil-pilot-services-improving-hazard-and-environmental-monitoring-in-the-philippines/

2 3 Supervised Classification | Google Earth Engine | Google for Developers
https://developers.google.com/earth-engine/guides/classification

4 Classification: Random Forests - EO4GEO
http://www.eo4geo.eu/training/classification-random-forests/

5 An assessment of the effectiveness of a random forest classifier for ...
https://www.sciencedirect.com/science/article/abs/pii/S0924271611001304

6 Introduction to Random Forest Machine Learning - UBC Blogs
https://blogs.ubc.ca/tdeenik/2021/04/22/introduction-to-random-forest-machine-learning/

7 10 11 13 14 ee.Classifier.smileRandomForest | Google Earth Engine | Google for Developers
https://developers.google.com/earth-engine/apidocs/ee-classifier-smilerandomforest

8 9 15 Frontiers | Optimal parameters of random forest for land cover classification with suitable data type and dataset on Google Earth Engine
https://www.frontiersin.org/journals/earth-science/articles/10.3389/feart.2023.1188093/full

16 17 GitHub - satellite-image-deep-learning/techniques: Techniques for deep learning with satellite & aerial imagery
https://github.com/satellite-image-deep-learning/techniques

18 eurosat | TensorFlow Datasets
https://www.tensorflow.org/datasets/catalog/eurosat

19 Applying Deep Learning on Satellite Imagery Classification. | by Wired Wisdom | DataDrivenInvestor
https://medium.datadriveninvestor.com/applying-deep-learning-on-satellite-imagery-classification-5f2588b932c1?gi=40526ec243cd