ChatGPT

Day 2, Session 3: Introduction to Deep Learning and CNNs for Earth Observation

Learning Objectives

By the end of this session, you will be able to:

- **Differentiate** between traditional machine learning approaches and the deep learning paradigm, especially in how features are extracted from Earth observation (EO) data.
- **Explain** the basic structure of a neural network and what makes a network "deep".
- **Identify** the key components of a Convolutional Neural Network (CNN) – convolutional layers, activation functions (ReLU), pooling layers, and fully connected layers – and describe their roles in image analysis.
- **Understand** how CNNs learn hierarchical features from imagery (e.g. edges in early layers, shapes in mid-layers, and complex patterns in deeper layers).
- **Recognize** common EO applications of CNNs, including scene classification (land use/land cover mapping), object detection (e.g. finding vehicles, buildings), and semantic segmentation (e.g. flood extent mapping, crop field delineation).
- **Define** a simple CNN model in a deep learning framework (PyTorch or TensorFlow) and interpret how each layer contributes to processing an EO image.

1. Recap: Neural Networks vs. Traditional ML in Remote Sensing

In previous sessions, we explored classification of remote sensing data using traditional machine learning (e.g. decision trees or Random Forests). Those classical methods typically required **manual feature engineering** – we had to provide input features such as spectral indices (NDVI, etc.), texture measures, or principal components to help the model distinguish classes. In contrast, **deep learning** with neural networks introduces a paradigm shift: the model can automatically learn relevant features from raw data through a series of **learnable layers** [1]. This ability to learn hierarchical feature representations is a key advantage of deep learning in image analysis.

Traditional ML vs Deep Learning:

- *Feature Extraction:* Traditional ML often relies on handcrafted features or domain knowledge (for example, we might compute vegetation indices or GLCM texture for land cover classification). Deep learning models, especially CNNs, automatically learn **both low-level and high-level features** directly from the input images [2]. The CNN's convolutional filters adapt during training to extract informative patterns (edges, textures, shapes) without explicit feature design by the analyst.
- *Performance on Complex Data:* With increasing complexity of remote sensing data (multi-spectral, high resolution, temporal stacks), classical models may struggle to capture intricate patterns or context. CNN-based deep learning has become the **dominant approach** for image classification, object detection, and segmentation tasks because of its capacity to extract rich, high-level

representations of scenes [2]. In other words, CNNs excel at capturing the **semantics** of EO imagery (e.g. recognizing an image as an "urban residential area" vs just pixels), whereas pixel-based classifiers using only low-level spectral features might miss that contextual understanding.

- *Data and Training:* Deep learning typically requires larger training datasets and more computation. In remote sensing, this has been mitigated by transfer learning (using models pre-trained on large datasets like ImageNet and fine-tuning them on EO data) and by the increasing availability of labeled EO datasets. For example, since the breakthrough of **AlexNet in 2012**, CNN-based deep learning has consistently defined the state-of-the-art in computer vision, and this trend has extended to Earth observation applications [3]. Today, powerful CNN architectures (ResNets, EfficientNet, etc.) pre-trained on massive image collections can be adapted to tasks like land cover mapping or object detection in satellite imagery with excellent results.

Think-Through: In the Random Forest classification exercise, we fed the model features like spectral band values or indices for each pixel. What might a deep CNN use as input, and how would it obtain the kinds of distinguishing information that we manually provided in the RF approach?

Neural Networks Refresher

A **neural network** is a computational model inspired by the human brain's network of neurons. At its core, each neuron (or node) performs a weighted sum of its inputs and passes the result through an activation function (often nonlinear). Neurons are organized in layers: an **input layer** (taking in data), some number of **hidden layers**, and an **output layer** that produces predictions. In training, the network adjusts the weights on connections via **backpropagation** to minimize prediction error. A network with multiple hidden layers is called a **deep neural network**, and "deep learning" refers to training such multi-layer networks.

For Earth observation data, a basic neural network (fully connected multilayer perceptron) could take as input the pixel values or pre-computed features for an image and learn to classify land cover. However, a fully connected network does not scale well to large images – the number of weights would explode if we connect every pixel to neurons in the first hidden layer. Moreover, fully connected layers lose the **spatial structure** of image data (they treat input as an unstructured vector). This is where **Convolutional Neural Networks (CNNs)** come in, as a specialized architecture to effectively handle image data by exploiting spatial locality.

2. CNN Architecture Basics

Convolutional Neural Networks are specifically designed for grid-like data such as images. Instead of full connections between layers, CNNs introduce **convolutional layers** that connect each neuron to a local region of the input, greatly reducing the number of parameters and preserving spatial relationships. A typical CNN for image analysis is composed of a sequence of operations serving as **feature extractors**, followed by one or more **fully connected layers** that act as a classifier [4]. The basic building blocks of a CNN are:

[5] **1. Convolutional Layer –** Applies learned filters/kernels to the input image (or feature maps from the previous layer). Convolutional layers slide these small filters across the image to produce **feature maps** highlighting various features. Each filter acts as a detector for a specific pattern in the input (for example, an edge oriented in a certain direction). The convolution operation preserves the spatial structure by only

combining values within a local neighborhood [6]. During training, the CNN learns the optimal weights of these filters so that they activate (produce strong feature map responses) for important visual features. Crucially, using convolutions means **weights are shared** across the image – the same filter is applied at every location – which makes the model much more efficient than a fully connected approach.

[7] **2. Activation Function (ReLU) –** After each convolution, a non-linear activation is applied. The most common is **ReLU (Rectified Linear Unit)**, which is an element-wise operation that sets all negative values in the feature map to zero [7]. This introduces non-linearity into the network (without which a stack of linear convolutions would collapse into a single linear operation). ReLU helps the network learn complex patterns and also tends to make training faster and more effective by avoiding some issues like vanishing gradients. Other activations like sigmoid or tanh can be used, but ReLU has been found to work better in practice for CNNs [8].

3. Pooling Layer – A pooling (subsampling) layer typically follows one or a few convolution layers + activations. **Spatial pooling** reduces the spatial dimensions (width and height) of the feature maps while retaining the most important information [9]. The most common is **Max Pooling**, which takes a window (e.g. 2×2 or 3×3) and outputs the maximum value in that region, essentially picking the strongest activation in that window. Pooling thus down-samples the image representation, which *compresses* data and provides two benefits: (a) it reduces the number of parameters and computations in later layers, and (b) it provides a degree of **translational invariance** – small shifts or slight distortions in the input have less effect on the pooled feature maps [10]. By taking the maximum activation in a local patch, the network becomes less sensitive to the exact position of a feature (for instance, whether an edge is slightly to the left or right). Pooling can be thought of as consolidating redundant information: after a convolution finds an interesting feature, we only need a rough spatial location of that feature for the next layers. (Note: some CNN architectures forego pooling or use alternatives like strided convolutions, but the effect is similar in reducing spatial resolution of feature maps.)

4. Fully Connected (Dense) Layer – After several layers of convolutions and pooling, a CNN will have transformed the original image into abstracted feature maps of smaller spatial size but greater depth (number of channels). At the end of the network, one or more fully connected layers take the final set of feature activations and produce an output classification (or regression). The fully connected layers treat the extracted features as input to a traditional neural network classifier. Essentially, **the convolution/pooling part of the CNN serves as an automatic feature extractor, and the fully connected part (multilayer perceptron) serves as the classifier** [4]. The output layer of a CNN for classification will typically use a Softmax activation to produce class probabilities. For example, if our CNN is meant to classify an EO image patch into land cover categories, the final layer might have *N* neurons (where *N* is the number of classes, e.g. 10 for EuroSAT land cover types) and Softmax to output a probability distribution over the classes.

Illustration: A simple CNN architecture for image classification. The input image passes through a stack of convolution + ReLU layers (feature extraction), each followed by pooling layers that reduce spatial size. Finally, the high-level features are flattened and fed into fully connected layers for classification (outputting probabilities for each class).

Each of these components plays a crucial role: - The **convolutional layers** learn *localized feature detectors* that respond to patterns in the imagery (edges, textures, etc.). - The **ReLU activations** inject non-linearity, allowing the network to learn complex decision boundaries. - The **pooling layers** progressively reduce the image representation size and make the detected features more robust to position changes. - The **fully**

connected layers combine all the extracted features to decide the best class label (or other task output) for the image [11].

These operations are the fundamental building blocks of virtually every CNN [5]. Modern CNN architectures simply layer these components in creative ways. For instance, a deep CNN might have multiple convolution-ReLU-convolution-ReLU-pooling sequences in a row to extract increasingly complex features at multiple scales, followed by a few dense layers at the end. **It's not necessary to have a pooling layer after every convolution** – sometimes multiple convolutions are stacked before a pooling step, especially in very deep networks [12]. But overall, the flow is as described: **feature extraction by conv filters, downsampling by pooling, and final classification by dense layers** [4].

> **Think-Through:** Why is it beneficial for a CNN to use small convolutional filters scanning over the image, rather than one huge filter covering the entire image at once? Consider the implications for the number of parameters and the ability to detect local patterns.

The Convolution Operation in Detail

Before moving on, let's develop an intuition for the **convolution operation**, since it is central to CNNs. A convolution in image processing terms involves a small matrix of weights (the *filter* or *kernel*) that is multiplied element-wise with a patch of the image, then summed to produce a single value in the output feature map. The filter is then slid to the next spatial position (according to the stride) and the process repeats, producing another output value, and so on for all positions.

Figure: Visualization of the convolution operation. A 3×3 filter (orange overlay) slides over the input image (green grid of pixel values). For each position, the element-wise products of the filter values and the corresponding image pixels are summed to yield one number in the output feature map (pink cell). In this example, the filter acts as an edge detector, producing higher values where there is a contrast in the input.

Several important concepts illustrated above:

- The **filter (kernel)** has a certain size (3×3 in the figure) which defines the receptive field it looks at in the input. Typical filter sizes in CNNs are 3×3 or 5×5 (occasionally 7×7 in first layers). Small filters are common because stacking multiple small convolutions is more parameter-efficient and effective than one large filter (e.g. two 3×3 conv layers have the effect of a 5×5 receptive field but with fewer parameters and an extra non-linearity in between).
- The filter moves across the image with a certain **stride** (the example uses stride 1, moving one pixel at a time). Stride controls how much we downsample during convolution; a stride of 2 would move the filter two pixels at a time, producing a smaller output feature map.
- The edges of the image are handled by either ignoring incomplete patches or by applying **zero-padding** (padding the image with zeros around the border so that the filter can fully cover every pixel even at the edges). Zero-padding is often used to preserve the input size (called "same" convolution in some frameworks) [13].
- The output of one convolutional filter is one **feature map**. A CNN typically uses many filters (say 16, 32, or even 256 filters) in a convolutional layer, so it produces that many output feature maps. The number of filters is often called the **depth** of the conv layer [14]. Each filter will learn to detect a different feature in the image (for example, one might become an edge detector, another might

activate on a green vegetation texture, another on a water texture, etc.). As a result, increasing the number of filters allows the network to extract a richer set of features [15].

An intuitive example of convolution's effect: If we choose a filter's weights to approximate an edge detector (say, a Sobel filter), then convolving it with an image will result in high values in positions where an edge is present in the original image. By learning the filter values from data, CNNs figure out the optimal edge detectors, blob detectors, texture detectors, etc., needed for the task. The **key power of CNNs** is that these filters are learned automatically via training, rather than designed by a human [15].

In summary, **convolutional layers learn to detect features**. Early in the network, filters typically learn very basic visual primitives (edges, corners). In later layers, filters can represent more complex shapes or semantic patterns (like rooftops, roads or ships in satellite imagery). We will discuss this hierarchical feature learning next.

The Role of Activation (ReLU) in CNNs

After computing the linear convolution, CNNs apply an **activation function** like ReLU. ReLU (Rectified Linear Unit) simply outputs 0 for any negative input and identity for positive input: `f(x) = max(0, x)`. Despite its simplicity, ReLU has a big impact: - It introduces **non-linearity** – without it, combinations of convolutions are just linear and the network could not model complex relationships. - It helps with the vanishing gradient problem (gradients flow better through ReLUs than through sigmoids/tanh in deep networks) and generally speeds up training convergence. - Empirically, ReLU often leads to better performance in vision tasks than older activations.

Almost every convolutional layer in modern CNNs is followed by a ReLU (or a variant like LeakyReLU or ELU). The pattern is: **Conv -> ReLU -> (Pooling) -> Conv -> ReLU -> (Pooling) -> …** and so on [16]. This pattern ensures that after each convolution, we keep only positive signals (features that excite the filter) and zero-out the rest, making the feature maps sparse and focused on prominent features.

> **Think-Through:** Imagine if we omitted the ReLU (or any non-linear activation) after convolution layers. How would that affect the CNN's ability to learn? (Hint: think about multiple linear operations composed together.)

Pooling Revisited: Why and How

Pooling often raises the question: by discarding information (only taking maxima or averages), are we losing useful detail? It's true that pooling is a trade-off between **spatial precision** and **invariance/efficiency**. Max pooling chooses the strongest feature in each region, which provides a degree of position and scale tolerance – e.g. if a target object moves slightly in the image, a max-pooled feature map might remain unchanged, which is desirable for detection robustness [10]. The network designers choose pooling size/stride to reduce data gradually. A 2×2 max pool with stride 2 (common choice) will reduce width and height by half. Doing this repeatedly drastically lowers the number of computations needed in deeper layers, enabling deeper networks to be trained.

However, if very fine spatial detail is required (say, for precise segmentation boundaries), one must be careful with pooling or use strategies to reclaim resolution (like upsampling later, or using small stride so as not to lose too much detail). In classification tasks, exact spatial positions are less critical than whether a

feature is present anywhere in the image, so aggressive pooling is fine. In dense prediction tasks (segmentation), modern architectures often still pool but then use techniques to recover resolution (e.g. U-Net or Fully Convolutional Networks use an encoder-decoder structure where pooling happens in the encoder and the decoder upsamples to produce a pixel-level output).

In summary, **pooling reduces dimensionality and makes the representation more abstract**, capturing the "gist" of features while discarding exact locations [9] [10]. It's one of the reasons CNNs can achieve **equivariance** to translation – shifting an image slightly results in correspondingly shifted feature maps up to before pooling, and after pooling, small shifts might not change the pooled outputs at all, making the network output stable under translations.

Fully Connected Layers: From Features to Decision

After a series of conv/ReLU/pool layers, a CNN will have a final set of feature maps. These are typically flattened into a one-dimensional vector of features which feed into one or more **fully connected (dense) layers**. The dense layers mix all the information together to make a global decision about the image. In a classification CNN, the last dense layer yields **class scores or probabilities** for each category.

Think of the convolutional part as producing a set of high-level descriptors: e.g., "this image has an edge here, a green patch there, a circular shape here, etc." – and the fully connected part as the logic that combines those descriptors: "green patch + round shape => likely a *forest* class" or "linear edges + right angles + gray color => *buildings* class". The fully connected layers are essentially the same as a standard multilayer perceptron, which can learn any arbitrary mapping from the extracted features to the output classes [11].

An important concept is that during training, **the convolutional layers will learn to produce features that are most useful for the final classification task**, and the fully connected layers learn how to weigh those features to produce the correct class. This end-to-end learning is what makes deep learning powerful: the features are optimized for the task at hand, not manually chosen. For EO images, that means a CNN might learn to detect things like "texture of water," "rectangular building-like shapes," or "rows of crops" as intermediate features because those help distinguish the classes, without anyone explicitly programming those features.

As a concrete example, consider a CNN trained to classify land cover in Sentinel-2 patches (like the EuroSAT dataset): The conv layers might develop filters that activate on spectral signatures of vegetation vs. urban materials, or on geometric patterns like straight lines (roads) vs. amorphous regions (fields/forest). The final fully connected layers would then use those signals (e.g. strong "green vegetation texture" feature + low "man-made edge" feature implies *Forest* class; whereas strong "built-up edges" + certain spectral signature implies *Residential* class).

Finally, it's worth noting that some modern architectures replace the fully connected layers with a global pooling (like Global Average Pooling) followed directly by the output. But the principle is similar: combine the information from all spatial locations and all feature channels into the final prediction.

> **Think-Through:** CNN feature extractors are often shared across different tasks via transfer learning. Why do you think the convolutional filters learned on a large dataset (like ImageNet)

might be useful for a remote sensing task, even if the images are very different? Which layers' filters (early vs. late) are more likely to be transferable [17] ?

3. Hierarchical Feature Learning in CNNs

One of the most fascinating aspects of CNNs is how they build up **hierarchical features** – a layered understanding of the image. This is a concept we hinted at: earlier layers detect simple patterns, and later layers detect complex concepts by combining the earlier patterns [18] . Let's break this down:

- **Layer 1 (Lower-level features):** The first conv layer sees the raw pixels (plus possibly all bands in multispectral images). Filters here usually learn to respond to very basic features like *edges of various orientations*, *spots*, or *color/spectral gradients*. In EO imagery, first-layer filters might detect things like a contrast between red and green channels (picking out vegetation vs non-vegetation), or simple textures. These are analogous to edge detectors or Gabor filters that computer vision researchers used to hand-craft in the past. The CNN learns them automatically. Notably, these features are generally **universal** – for example, edge or texture detectors are useful in almost any vision task, whether it's natural images or satellite images.

- **Intermediate Layers (Mid-level features):** As we go deeper (layer 2, 3, etc.), the CNN filters now take as input the feature maps from previous layers (which themselves highlight edges or simple patterns). These filters can activate on combinations of lower-level features. For instance, if certain edge features in layer 1 form a closed rectangular shape, a layer-2 filter might detect a *rectangle* (which could indicate a building footprint in an aerial image). Other mid-level features might be *corners, curves, blobs, textures (like "grid pattern" for a city block or "linear streak" for a road)*. Essentially, the network starts to capture parts of objects or more complex textures. **Hierarchical composition** is at work: edges combine to form shapes; spectral blobs combine to suggest materials; etc. [18] .

- **Deeper Layers (High-level features):** In the last convolutional layers, the receptive field of the filters is large (they can "see" a big portion of the image, thanks to prior pooling/strides). These filters respond to very complex, abstract concepts – essentially they can detect *specific objects or semantic regions* in the image. For example, a deep layer might activate strongly if it sees a pattern like "building-like arrangement of edges with a certain size" or "tree canopy texture" or "water texture with sun glint". In a CNN trained on general photos, deep layers might detect faces or legs or specific objects. In remote sensing, deep layers might pick up things like *urban vs rural scene*, *airplane shape*, *boat on water*, or *crop field row structure*. They are highly tuned to features relevant to the training classes. In summary, **earlier layers are general-purpose detectors, later layers are task-specific detectors** [17] .

A classic description of CNNs is: *Layer 1 finds edges; Layer 2 finds shapes or motifs; Layer 3 might find parts of objects; Layer 4 or 5 (if we have that many) might identify whole objects or scene categories.* This has been verified by visualizing CNN layers in many studies [18] . In fact, researchers have visualized that: - First layer might light up for "horizontal line" or "blue-green contrast" etc. - Second layer might light up for "textured patch" or "corner shape". - Third layer might detect "roof structure" or "road intersection". - Final conv layer might differentiate entire scene layouts ("forest patch" vs "buildings cluster").

This hierarchical learning is a big part of why CNNs perform so well – they **automatically learn the feature hierarchy** that a human might describe as first doing edge detection, then shape detection, etc., but without being explicitly programmed to do so.

To solidify this understanding, consider transfer learning: if you take a CNN trained on one image dataset and apply it to another, the early layers (edges, textures) are usually *reusable* because those features are useful everywhere [17]. The later layers are more specific to the original task and might need retraining to adapt to new classes. In practice, this is why we often freeze the first few layers of a pre-trained network and only fine-tune the later layers on our specific EO dataset.

In the context of Earth observation: - **General features (early layers):** detect basic elements like water-land boundaries, straight lines (which could be roads or field boundaries), coarse textures (smooth water vs rough forest canopy). - **Specific features (late layers):** combine those to detect high-level classes, e.g., a pattern that looks like "grid of roads plus rectangular roofs" = city, or "meandering boundary with smooth texture on one side and rough on other" = coastline, etc.

> **Think-Through:** Suppose you have a CNN that was trained to identify ships in high-resolution satellite images. What do you think the first convolutional layer's filters might look like (visually or in terms of pattern)? How about a filter in one of the last layers? Consider the differences in what they "focus on" at different depths.

4. CNN Applications in Earth Observation

With the fundamentals of CNNs understood, let's explore how they are applied in various Earth Observation tasks. Deep learning has made a huge impact in remote sensing across a range of applications:

4.1 Scene Classification (Image-Level Land Use/Land Cover Classification)

Scene classification involves assigning a label to an entire image or image patch, describing its predominant class or scene type. In EO, this could mean labeling a satellite image patch as "urban residential", "forest", "water", "agricultural field", etc. CNNs are **well-suited for this task** because they can take raw image pixels as input and learn both spectral and spatial features that characterize different land cover types [2]. This is a step up from traditional per-pixel classification; it considers the image holistically.

A prominent example is the **EuroSAT dataset** – which is a benchmark LULC (Land Use/Land Cover) classification dataset. EuroSAT consists of 27,000 labeled Sentinel-2 image patches (64×64 pixels each) across **10 classes** (categories such as annual crops, forest, herbaceous vegetation, highway, residential, industrial, pasture, permanent crop, river, and sea/lake) [19] [20]. Each patch is labeled with the land cover/use that dominates that scene. Researchers have trained CNNs on this dataset to very high accuracies (over 98% in some cases using transfer learning) [21], far surpassing what earlier techniques could achieve. The CNN automatically learns spectral-spatial features for each class (e.g., the distinct checkerboard pattern of agricultural fields vs. the chaotic texture of forests vs. the linear features in built-up areas).

For instance, using a ResNet CNN pre-trained on ImageNet and fine-tuning it on EuroSAT, Helber et al. achieved excellent classification accuracy, showing CNNs can effectively classify Sentinel-2 scenes [21]. The **combination of spectral bands and spatial context** is key – a CNN can utilize all 13 Sentinel-2 bands (as

an input with 13 channels) to pick up spectral signatures, while also recognizing shapes and textures like field boundaries or building patterns. This would be extremely hard to do with a traditional classifier unless one manually designed features for texture, shape, etc., which CNNs do automatically.

In practical terms, scene classification CNNs enable: - Mapping land use/land cover from patch-based classification of satellite tiling. - Change detection by classifying patches from different dates and seeing changes in class (e.g., forest to bare ground). - The basis for further analysis like feeding into geographic information systems for mapping.

> **Example:** A CNN trained on EuroSAT might take a 64×64 Sentinel-2 image patch and output probabilities: 90% Forest, 5% Herbaceous, 3% Pasture, etc., indicating it's most likely a forest scene. The network might have learned to identify the spectral signature of dense vegetation along with the texture that differentiates forest from agriculture (forest being more irregular canopy vs. agriculture often having structured rows or uniform color).

4.2 Object Detection in Aerial/Satellite Imagery

Object detection involves locating and classifying multiple objects within an image, usually by drawing bounding boxes around them. In EO contexts, objects of interest could be **vehicles, ships, airplanes, buildings, pools, solar panels, animals**, or any discrete item visible in the imagery. This is a more challenging task than scene classification because the model must not only say what is present but also where it is, and handle potentially many objects per image at varying scales.

CNNs have revolutionized object detection in overhead imagery as well. The typical approach is to use advanced CNN-based detectors (originating from computer vision research) like **Faster R-CNN, YOLO (You Only Look Once), or SSD (Single Shot Detector)**, sometimes with modifications for aerial images: - **Two-stage detectors** like Faster R-CNN use a CNN (e.g. ResNet or VGG) as a **backbone** to extract feature maps, then a region proposal network (RPN) suggests candidate bounding boxes, and finally another network head classifies those regions and refines the boxes. These tend to be more accurate and are popular when high precision is needed [22]. For example, Faster R-CNN has been widely used for detecting buildings, ships, and airplanes in satellite images, often with adjustments to handle the fact that objects can be arbitrarily oriented (not aligned to image axes) in overhead images. - **One-stage detectors** like YOLO or SSD skip the region proposal step and predict bounding boxes and classes in one go, making them faster. These are also applied in EO for things like real-time monitoring (e.g. a drone footage analysis).

A challenge in EO is that objects can be tiny (e.g. cars in 30 cm imagery are just ~20 pixels long) and densely packed (solar panel arrays or parked cars). CNN detectors have been adapted: for instance, using **rotated bounding boxes** (predicting angle of the box) so that elongated objects like airplanes can be tightly bounded even if rotated [22]. Also, high resolution images are very large, so methods like tiling the image and applying CNN on each tile, or using efficient architectures, are used.

Use cases include: - **Vehicle detection:** Identify and count cars in parking lots, or ships in harbors or at sea (useful for economic indicators, port traffic, illegal fishing monitoring, etc.). - **Building detection:** Automatically mapping buildings from aerial images for urban planning or disaster response (e.g. after an earthquake, detecting collapsed buildings). - **Aircraft detection:** Monitoring airfields, or detecting planes in imagery (e.g. for treaty compliance or reconnaissance). - **Wildlife detection:** E.g., using drones to detect

animals in savannah. - **Infrastructure detection:** Finding roads, power lines, etc., often approached either as object detection or segmentation.

CNNs are very effective for these tasks because they can learn the varied appearance of objects from examples, even under different imaging conditions. For example, a well-trained CNN can detect ships in Sentinel-2 or PlanetScope imagery despite varying water colors or ship orientations. Studies have shown that CNN-based detectors outperform earlier approaches (like template matching or classical computer vision methods) significantly.

One interesting note: because EO imagery often covers large areas with relatively sparse objects, a lot of research goes into how to efficiently apply CNNs. Strategies involve using image pyramids (to detect objects at multiple scales), or the CNN's feature maps at different layers to detect both small and large objects (feature pyramid networks), etc.

[23] [24] CNNs have been used to detect entities in transportation (vehicles, roads), settlements (buildings, houses), and agricultural contexts (animals, machinery) from above. In fact, **object detection in EO is one of the areas where deep learning's impact is most visible** – tasks that were once nearly impossible (like identifying every car in a 30 cm resolution image of a city) are now feasible with high accuracy using CNN detectors.

> **Example:** A YOLOv3 model is applied to a 1024×1024 px satellite image and outputs bounding boxes around several objects: 5 boxes labeled "Building", 2 boxes labeled "Truck", etc., with confidence scores. Under the hood, the CNN has learned to recognize the visual patterns of those objects (like the rectangular rooftop + shadow shape of a building, or the elongated shape of a truck). The output might be visualized as the original image with colored boxes drawn around each detected object.

4.3 Semantic Segmentation (Pixel-wise Classification)

Semantic segmentation is about labeling each pixel in an image with a class (as opposed to one label per image in scene classification, or one label+box per object in detection). In remote sensing, this translates to creating **thematic maps**: e.g., given a satellite image, label every pixel as water, forest, urban, etc., or delineate specific features like flooded areas vs non-flooded, crop type A vs crop type B, etc.

CNNs approach segmentation by an **encoder-decoder architecture**. The **encoder** is often a CNN similar to those for classification (it extracts features and progressively pools/downsamples), and the **decoder** uses upsampling (via transpose convolutions or interpolation + conv) to build back up a full-resolution output map. A seminal architecture is **U-Net**, which has an encoder and a symmetric decoder with skip connections (features from encoder layers are merged with decoder layers to help recover fine details). Another is the Fully Convolutional Network (FCN) which replaced dense layers with convolutional ones to output a grid of class predictions.

In EO, segmentation CNNs like U-Net have become extremely popular for tasks such as: - **Land cover mapping:** Producing pixel-level classification of land cover types in an image. For instance, mapping urban, vegetation, water, bare land in a single pass over a large satellite image. - **Building footprint extraction:** Labeling pixels that belong to buildings (useful for creating maps of all buildings). - **Road extraction:** Segmenting out road networks from high-res imagery or LiDAR. - **Flood inundation mapping:** Determining

which pixels are flooded water vs. normal (dry) land in a satellite image taken during a flood event [25]. - **Crop field segmentation:** Identifying boundaries of fields or segmenting different crop types based on temporal composites. - **Forest fire/burn scar mapping:** Marking burned area extent from imagery. - **Cloud and cloud shadow masking:** Identifying cloud pixels in optical imagery (though this is often a separate pre-processing task, it can be done with CNN segmentation too).

The advantage of CNNs here is their ability to integrate *both* local details and global context. A pixel's label can often be determined by the texture/pattern in its neighborhood and the larger structure it belongs to. CNNs can learn these context patterns (e.g., a pixel with moderate NDVI might be forest if surrounded by other high NDVI pixels and tree texture, but the same NDVI might indicate crops if in structured row patterns, etc.).

For example, in **flood mapping**, researchers have used CNN segmentation on SAR or optical images to isolate floodwater. A deep CNN (e.g., U-Net with a ResNet backbone) can achieve very high accuracy in labeling water vs land, significantly reducing the need for manual thresholding or visual analysis [25]. These CNNs learn to detect subtle differences in backscatter or reflectance that indicate water under flood conditions, even under clouds (when using SAR). One study noted deep CNN methods provided **80% reduction in time** to produce flood maps compared to semi-automated methods, while improving accuracy [26].

Another example: **crop type mapping** – using multi-temporal satellite data, a CNN could classify each pixel as corn, wheat, soy, etc. Traditional methods required computing summary statistics or indices then classifying; a CNN can take the raw time-series (converted to an image-like structure) and directly learn optimal features, capturing phenological differences.

[27] Notably, encoder-decoder CNNs like U-Net are widely used in EO segmentation because they combine precise localization (through skip connections restoring spatial detail) with the deep feature power of CNNs. U-Net and its variants have become go-to architectures for tasks like building footprint extraction and land cover segmentation, where knowing *exactly which pixels* belong to a class is required.

> **Example:** A U-Net model is given a 512×512 patch of a satellite image and produces a 512×512 mask labeling each pixel either "flooded" or "not flooded". The output might look like a binary image highlighting the flood extent following a river. The CNN learned from training data how flooded areas appear (in SAR: dark smooth surfaces, or in optical: certain color and texture differences) and was able to delineate them. Such a model could be used to quickly map flooding after a hurricane over large areas, something that would be tedious manually.

4.4 Other EO Applications

Beyond these main tasks, CNNs (and deep learning generally) have penetrated into many other areas of EO: - **Change Detection:** Using CNNs to detect changes between two images (possibly as a segmentation of change areas, or classifying change vs no-change). - **Super-Resolution or Image Enhancement:** CNNs can learn to upscale lower resolution satellite images or fuse data from multiple sources. - **Data Fusion:** Combining modalities (e.g., optical + SAR) by feeding multi-channel inputs into a CNN to leverage complementary information for classification or detection. - **Object Counting:** Counting objects (like trees, animals) by combining detection and density estimation with CNNs. - **Anomaly Detection:** Finding unusual

patterns in imagery (for example, new constructions or illegal mining) using autoencoders or one-class CNN models. - **Time-series forecasting with CNNs:** For example, CNNs can treat a time-series (like a sequence of images) in a convolutional manner (ConvLSTM or 3D CNN) for tasks like predicting crop yields or weather patterns.

The common theme is that **feature learning** by CNNs, often augmented with other deep learning layers (recurrent layers for time series, transformer layers in newer models, etc.), provides a powerful toolbox for EO analysts that goes far beyond what was possible with manual feature engineering. According to a 2019 review, CNNs were already in an "advanced transition phase from computer vision to EO", with a majority of EO studies focusing on method development and proof-of-concepts using CNNs [28]. Optical imagery with high spatial resolution has been the most common data for CNN applications, since CNNs shine with rich spatial details [29]. But CNNs have also been adapted to other data like hyperspectral (by 3D convolutions or treating spectral dimension specially) and SAR (where textural patterns are learned).

5. Introduction to TensorFlow and PyTorch for CNNs

To implement and experiment with CNNs, two popular deep learning frameworks are commonly used: **TensorFlow (with Keras)** and **PyTorch**. Both allow us to define neural network architectures and train them on data using high-level building blocks for layers, loss functions, and optimization algorithms. We will give a brief introduction to defining a simple CNN in each framework. (The hands-on lab in Session 4 will go deeper into using these frameworks for training a model on EO data.)

Both frameworks ultimately do similar things, but their style differs: - **Keras (TensorFlow)** provides a high-level, user-friendly API, where you can quickly stack layers in a `Sequential` model or use the functional API to build more complex models. It's very straightforward for beginners. - **PyTorch** provides a slightly lower-level feel, with a dynamic computation graph. You define `nn.Module` classes for models, and explicitly write the forward pass. This offers flexibility and is very popular in research.

We'll demonstrate defining a basic CNN with two convolutional layers and one fully connected layer in both frameworks.

Example: Building a Basic CNN in PyTorch

In PyTorch, we define a model by subclassing `nn.Module` and defining layers in `__init__` and the forward pass in `forward`. Below is a simple CNN for a classification task with RGB images of size 64×64 pixels, and say 10 output classes (like a simplified EuroSAT example):

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

# Convolutional layer 1: input channels = 3 (RGB), output channels = 16, kernel
```

```
size = 3
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
padding=1)

# Convolutional layer 2: input channels = 16, output = 32, kernel size = 3
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
padding=1)
        # Pooling layer: 2x2 window
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

# Fully connected layer: 32 feature maps of size 16x16 will be flattened ->
32*16*16 inputs, 64 hidden units
        self.fc1 = nn.Linear(32 * 16 * 16, 64)
        # Output layer: 64 inputs, 10 output classes
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        # Input x is [batch_size, 3, 64, 64]
        x = self.conv1(x)            # conv1 -> output shape [batch, 16, 64, 64]
        x = F.relu(x)               # activation
        x = self.pool(x)            # pool1 -> output shape [batch, 16, 32, 32]
        x = self.conv2(x)            # conv2 -> [batch, 32, 32, 32]
        x = F.relu(x)
        x = self.pool(x)            # pool2 -> [batch, 32, 16, 16]
        x = x.view(-1, 32 * 16 * 16) # flatten
        x = F.relu(self.fc1(x))      # fully connected + activation -> [batch,
64]
        x = self.fc2(x)
# output layer (no activation here if using CrossEntropyLoss)
        return x

# Instantiate the model and inspect
model = SimpleCNN()
print(model)
```

If you print this model, you'll see a structure listing the layers. The forward pass logic follows exactly the conceptual flow we discussed: Conv -> ReLU -> Pool -> Conv -> ReLU -> Pool -> Flatten -> FC -> output. In a training loop (which we'll cover in the lab session), you would pass image data through this `model`, compute loss against true labels, and use an optimizer to adjust weights. PyTorch handles backpropagation for us automatically when we call `loss.backward()`.

A few points to note: - We padded the conv layers (`padding=1`) so that a 3×3 filter doesn't reduce the spatial size (64 stays 64 after conv, until pooling halves it). This way after two poolings the spatial size went from 64 -> 32 -> 16. - The linear layer `self.fc1` needs to have the correct input size. We had 32 feature maps of size 16×16 at that point, hence 32*16*16 = 8192 inputs. We chose 64 hidden units arbitrarily for demonstration. - We used `F.relu` functional calls for activation; we could also define ReLU as a layer `self.relu = nn.ReLU()` and call it. PyTorch is flexible either way. - The final layer outputs raw scores

(logits) for 10 classes. In PyTorch, if we use `CrossEntropyLoss`, it expects these logits and internally applies Softmax, so we don't put Softmax in the model definition in that case.

Example: Building a Basic CNN in TensorFlow/Keras

Using TensorFlow's Keras API, we can define a similar model more succinctly with a Sequential container:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Conv2D(filters=16, kernel_size=3, padding='same', activation='relu',
input_shape=(64, 64, 3)),
    layers.MaxPooling2D(pool_size=(2, 2)),  # now output 32x32
    layers.Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),  # now output 16x16
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes, softmax gives
probability distribution
])
model.summary()
```

A few differences to note: - Keras by default uses channels-last image format (64,64,3) as input shape, whereas PyTorch uses channels-first (3,64,64). We specify `input_shape=(64, 64, 3)` for the first layer in Keras. - We included `activation='relu'` directly in the Conv2D layers for brevity. - After building the model, `model.summary()` will print out the layers and their output shapes, which is useful to verify our architecture. - We used a Softmax on the final layer since in Keras it's common to include it and compile the model with a loss like `categorical_crossentropy` or `sparse_categorical_crossentropy`.

This Keras model corresponds almost one-to-one with the PyTorch model we wrote: Conv2D -> ReLU -> Pool -> Conv2D -> ReLU -> Pool -> Flatten -> Dense -> Dense.

Both frameworks will ultimately perform the same computations under the hood. It’s a matter of preference and context which one to use. Since this course focuses on concepts, you're free to use either in the lab. We will provide guidance in both if needed.

Training: After defining a model, typically you would: - For PyTorch: write a training loop (loop over epochs, inner loop over batches, do `optimizer.zero_grad()`, `output = model(batch)`, compute loss, `loss.backward()`, `optimizer.step()`). - For Keras: call `model.compile(optimizer=..., loss=..., metrics=...)` then `model.fit(train_data, train_labels, epochs=..., batch_size=...)` which handles the loop internally.

We won't go into full training here (that's for Session 4 lab), but remember that training a deep CNN on imagery often requires GPU acceleration to be feasible in a reasonable time. During training, you'll monitor metrics like training loss, validation accuracy, etc., to ensure the model is learning well and not overfitting.

> **Think-Through:** In the simple CNN examples above, we arbitrarily chose 2 conv layers and then a dense layer. How would you decide on the number of layers or number of filters in a real problem? What considerations might guide increasing the depth or width of the network for classifying Earth observation images?

6. Summary of Key Concepts

In this session, we introduced the principles of deep learning and CNNs as they apply to Earth observation, laying the groundwork for the hands-on lab to follow. Let's summarize the key takeaways:

- **Deep Learning vs Traditional Methods:** Deep learning with neural networks (especially CNNs) automatically learns feature representations from raw data, which is a major advantage over traditional ML that required manual feature extraction. This capability has made CNNs the state-of-the-art for analyzing the complex spectral and spatial information in EO imagery [2].

- **CNN Building Blocks:** A Convolutional Neural Network typically consists of convolutional layers (with learnable filters), activation functions like ReLU, pooling layers for downsampling, and fully connected layers for final classification [5]. These components work together to extract multi-scale features and make predictions. Convolution preserves spatial structure by looking at local regions, ReLU adds non-linearity, pooling provides invariance and reduces dimensions, and dense layers synthesize the features into a decision.

- **Hierarchical Feature Learning:** CNNs learn a hierarchy of features – from edges and textures in early layers to shapes and objects in later layers [18]. This enables understanding of imagery at different levels of detail. Initial layers are general (often transferable across datasets [17]), while later layers become specialized to the classes of interest in the training data.

- **CNNs in EO Applications:** We covered three primary remote sensing tasks:

- *Scene Classification:* CNNs classify whole image patches into scene or land cover categories (e.g., classifying a Sentinel-2 patch as forest, water, urban, etc.). Example: Using CNNs on the EuroSAT dataset achieves high LULC classification accuracy with automatically learned spectral-spatial features [2].
- *Object Detection:* CNN-based detectors (Faster R-CNN, YOLO, etc.) can locate and identify objects in aerial images, from vehicles and ships to buildings. They leverage CNN backbones to propose and classify regions, and can be adapted for rotated objects and small targets common in overhead imagery [22].

- *Semantic Segmentation:* CNN architectures like U-Net provide pixel-wise classification, essential for mapping tasks (flood extent, crop types, land cover maps). These models output high-resolution masks and have proven extremely effective in capturing fine details while maintaining context [27]. For instance, CNNs have demonstrated excellent performance in mapping floods from SAR imagery, greatly improving speed and accuracy of flood delineation [25].

- **Frameworks (TensorFlow/PyTorch):** We introduced how to define a simple CNN in both PyTorch and TensorFlow-Keras. Key points include understanding input dimensions (channels first vs last), layer stacking, and how convolution/pooling alter image size. Both frameworks allow rapid prototyping of CNNs which we will leverage in the next session's lab.

- **Preparing for the Lab:** In the next session, you will implement and train a CNN for an EO classification task (likely patch-based classification). Be ready to think about practical aspects: data preparation (normalizing pixel values, handling multiple bands), model tuning (choosing architecture depth/width), and training considerations (preventing overfitting, using validation sets). The conceptual groundwork from this session should empower you to understand *why* the CNN is structured a certain way and *how* it is learning from the data.

Deep learning is a vast field, but the essentials you learned here – how a neural network functions, how CNNs exploit image structure, and the types of EO problems they can solve – will enable you to critically understand and build AI solutions for remote sensing. As you proceed to the hands-on part, remember that successfully training a deep model also requires experimentation and intuition. Use the "Math Academy" approach of breaking problems down, thinking through each component, and layering complexity gradually. Happy deep learning with Earth observation data!

---

1 2 17 20 21 Deep Transfer Learning for Land Use and Land Cover Classification: A Comparative Study
https://www.mdpi.com/1424-8220/21/23/8083

3 22 23 24 27 28 29 elib.dlr.de
https://elib.dlr.de/136742/1/2020-09-28_DLinEO_ESA_Phiweek_Hoeser.pdf

4 5 6 7 8 9 10 11 12 13 14 15 16 18 An Intuitive Explanation of Convolutional Neural Networks – Ujjwal Karn's blog
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

19 EuroSAT dataset sample images. The available classes are Forest ...
https://www.researchgate.net/figure/EuroSAT-dataset-sample-images-The-available-classes-are-Forest-Annual-Crop-Highway_fig1_356747729

25 Convolutional Neural Network-Based Deep Learning Approach for Automatic Flood Mapping Using NovaSAR-1 and Sentinel-1 Data
https://www.mdpi.com/2220-9964/12/5/194

26 UN Global Pulse
https://www.unglobalpulse.org/document/fully-convolutional-neural-network-for-rapid-flood-segmentation-in-synthetic-aperture-radar-imagery/