ChatGPT

Day 2, Session 4: CNN Hands-on Lab (Earth Observation Images)

Session Overview: In this hands-on session, we will reinforce convolutional neural network (CNN) concepts by implementing simple CNN models for remote sensing image classification (and an optional segmentation task). We will practice using both TensorFlow/Keras and PyTorch frameworks (with optional use of the TorchGeo/GeoTorch library) on Earth observation (EO) data. Two example datasets are provided: (1) a subset of the **EuroSAT** land use dataset (RGB Sentinel-2 patches), and (2) **Sentinel-2 image patches from a Palawan (Philippines) area of interest** prepared via Google Earth Engine (GEE). Students can choose either dataset for the classification task. The lab will guide you through building a basic CNN (with 2–3 convolutional layers) in both Keras and PyTorch, training it to perform scene-level classification (patch-based land cover classification), and evaluating its performance. An optional section is included for implementing a simple semantic segmentation model (e.g. a tiny U-Net) for pixel-wise classification.

The Math Academy Way: This lab is structured with clear learning goals, a layered learning approach, and scaffolded code walkthroughs. Throughout the lab, you will encounter **think-through prompts** (questions to ponder) and **mini-challenges** at the end of sections to test your understanding and encourage experimentation.

Learning Goals

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

- **Load and preprocess** satellite image patch data in Python, using Keras data utilities or PyTorch `Dataset` / `DataLoader` (optionally leveraging geospatial libraries like Google Earth Engine or TorchGeo for data handling).
- **Define a basic CNN architecture** for image classification, including convolutional layers with ReLU activation, pooling layers, and fully-connected (dense) output layers.
- **Explain the role of each layer** in a CNN (convolution for feature extraction, pooling for downsampling, flattening, dense layers for classification) [1].
- **Compile and train** a CNN model in both TensorFlow/Keras and PyTorch, using an appropriate loss function (categorical cross-entropy for multi-class classification) and optimizer (Adam) [2].
- **Evaluate model performance** using accuracy and confusion matrices, and interpret the results beyond a single accuracy metric [3].
- **Visualize predictions** by displaying sample input images with the model's predicted labels vs. true labels.
- *(Optional)* **Implement a simple semantic segmentation model**, understanding how an encoder-decoder CNN (like U-Net) can perform pixel-level classification [4], and train it to predict a binary mask (e.g., distinguishing one class vs. background).

0. Setup and Dataset Options

Before we begin, make sure you have access to the datasets and the proper environment:

- **Google Colab with GPU:** It is recommended to run this lab in Google Colab with a GPU runtime (Navigate to *Runtime* > *Change runtime type* > *Hardware accelerator* > *GPU*). This will speed up training significantly.
- **Dataset Option 1 – EuroSAT (RGB subset):** The EuroSAT dataset consists of Sentinel-2 satellite images (64×64 pixels, 3-band RGB) categorized into 10 land cover classes [5]. There are 27,000 images total, but we will use a subset for quicker training. The 10 classes include: *Annual Crop, Forest, Herbaceous Vegetation, Highway, Industrial, Pasture, Permanent Crop, Residential, River,* and *Sea/Lake* [6]. EuroSAT is a popular benchmark for land use classification [7].
- **Dataset Option 2 – Palawan Sentinel-2 Patches:** This is a custom set of Sentinel-2 image patches from an area in Palawan, Philippines, prepared via Google Earth Engine. *(Google Earth Engine is a cloud-based platform providing petabytes of satellite imagery and geospatial data for analysis [8].)* These patches may cover specific land cover types or an application (e.g., forest vs. non-forest, or urban vs. rural areas). If using this dataset, ensure you have the image patches and their labels (the data might be organized similarly to EuroSAT, e.g., images in folders per class for classification).

Selecting a Dataset: You may choose **either** dataset for the classification lab. The code provided can be adapted to either. For illustration, we will assume the EuroSAT subset by default (as it has well-known classes), but we will highlight any adjustments needed for the Palawan data if applicable.

Important: Make sure the data is available in your Colab environment. For EuroSAT, you might have a folder with subdirectories for each class. For the Palawan dataset, you might have something similar or a different structure – follow your instructor's guidance on accessing the data (e.g., downloading from a provided link or mounting Google Drive).

1. Loading and Preprocessing EO Image Patches

The first step is to load the image data and prepare it for training. We need to read the image files, assign labels, and split the data into training and test sets (and possibly a validation set). We will also perform basic **preprocessing**: ensuring the images are the same size, converting them to arrays or tensors, scaling pixel values, etc.

1.1 Using TensorFlow/Keras for Data Loading

In Keras, a convenient way to load images stored in directories is to use `image_dataset_from_directory` or **ImageDataGenerator**. Given a directory structure where images are organized by class (each class in a subfolder), Keras can create a `tf.data.Dataset` for training and validation easily.

```
import tensorflow as tf
from tensorflow.keras.utils import image_dataset_from_directory

# Define paths
```

```
data_dir = "path/to/eurosat_subset/"  # this directory should have subfolders
per class (0,1,... or named by class)
img_size = (64, 64)
# EuroSAT images are 64x64, but you could set a target size if needed
batch_size = 32

# Create training and validation datasets (80/20 split for example)
train_ds = image_dataset_from_directory(
    data_dir,
    labels="inferred",
    label_mode="categorical",        # one-hot encode the labels
    batch_size=batch_size,
    image_size=img_size,
    shuffle=True,
    seed=42,
    validation_split=0.2,
    subset="training",
)
val_ds = image_dataset_from_directory(
    data_dir,
    labels="inferred",
    label_mode="categorical",
    batch_size=batch_size,
    image_size=img_size,
    shuffle=False,
    seed=42,
    validation_split=0.2,
    subset="validation",
)
```

In the code above, `image_dataset_from_directory` will read images from the folders under `data_dir`. We specify `label_mode="categorical"` to get labels as one-hot vectors (since we'll use categorical cross-entropy loss). The images are automatically resized to 64×64 if they aren't already. We use 20% of data for validation.

Note: If using the **Palawan dataset**, ensure `img_size` matches the patch size (e.g., they might be 64×64 or 128×128 depending on how they were exported). The class folders and number of classes might differ (e.g., if it's a binary classification like "forest" vs "not forest", etc., you might use `label_mode="binary"` or still categorical with 2 classes). Adjust accordingly.

Data Normalization: By default, the pixel values will be loaded as 0–255 integers. It's generally beneficial to scale these to [0,1] for neural network training. We can do this by adding a normalization layer in the model or mapping the dataset:

```
# Normalize the pixel values to [0,1]
normalization_layer = tf.keras.layers.Rescaling(1./255)
```

```
train_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
val_ds   = val_ds.map(lambda x, y: (normalization_layer(x), y))
```

This will scale all images by 1/255. Alternatively, we could include `tf.keras.layers.Rescaling` as the first layer of our model.

Memory and Performance: The `image_dataset_from_directory` returns a `tf.data.Dataset` which loads data on the fly. We can improve performance by using `.cache()` if the dataset is small enough to fit in memory, and `.prefetch()` to overlap data loading with model training:

```
AUTOTUNE = tf.data.AUTOTUNE
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds   = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Now, `train_ds` and `val_ds` are ready to be fed into a Keras model via `model.fit()`.

1.2 Using PyTorch for Data Loading (with optional TorchGeo)

In PyTorch, data loading is handled by creating a custom `Dataset` (which returns an image tensor and label for each index) and then wrapping it in a `DataLoader` for batching and shuffling. We can either write our own `Dataset` class or use torchvision or TorchGeo utilities if available.

Option A: Using torchvision Dataset – If our data is organized in folders, we can use `torchvision.datasets.ImageFolder`, which automatically assigns labels from folder names, similar to Keras. We then apply transforms for preprocessing.

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

data_dir = "path/to/eurosat_subset/"

# Define transforms: convert images to tensor and normalize pixel values
transform = transforms.Compose([
    transforms.Resize((64, 64)),           # ensure images are 64x64
    transforms.ToTensor(),                 # convert to tensor [0,1]
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
    # normalize to [-1,1] (optional; here we subtract 0.5 and divide by 0.5 for
each channel)
])

# Create datasets
full_dataset = datasets.ImageFolder(root=data_dir, transform=transform)
# Split into train and val
```

```
train_size = int(0.8 * len(full_dataset))
val_size   = len(full_dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(full_dataset,
[train_size, val_size])

# Create DataLoaders
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Here, `ImageFolder` expects a directory structure where each subfolder name is a class name or label. It will assign labels 0,1,2,... in the order of the subfolder names. If using EuroSAT classes, you might have folder names like "AnnualCrop", "Forest", etc., but often for simplicity they might be numbered. In any case, `ImageFolder` will handle it. We define a transform to resize (if needed) and convert images to tensor. We also normalize: in this example, we normalized to a mean of 0.5 and std of 0.5 to yield pixel values roughly in [-1,1]. (You could also simply do `transforms.ToTensor()` which already scales 0–255 to 0–1, and then maybe `transforms.Normalize([0.5,...],[0.5,...])` or skip normalization for simplicity.)

If using the **Palawan dataset**, adjust the `transforms.Resize` to the patch size and possibly the normalization. If it's a binary classification (two classes), `ImageFolder` will still assign labels 0/1 accordingly.

Option B: Using TorchGeo (GeoTorch) – *Optional:* The TorchGeo library (sometimes referred to as GeoTorch) is a PyTorch domain library for geospatial data [9]. It provides ready-made Dataset classes for certain EO datasets and handles geospatial specifics. For example, TorchGeo has `torchgeo.datasets.EuroSAT` which can download and prepare EuroSAT. If you wanted, you could use it like:

```
# Optional TorchGeo approach (if library is installed)
!pip install torchgeo  # (in Colab, install the library first)
from torchgeo.datasets import EuroSAT

dataset = EuroSAT(root="data", bands=["RGB"])  # this will load the EuroSAT
dataset
```

TorchGeo can simplify data handling, but for learning purposes, it's just as good to use the standard approach above. We will proceed with the `ImageFolder` approach for clarity.

Think: Why do we normalize pixel values? Neural networks train faster and more reliably when inputs are on a consistent scale (e.g., roughly -1 to 1 or 0 to 1). If we left pixel values as 0–255, the optimization would have to adjust to large input values. Normalization also helps when using pretrained models.

Mini-Challenge: If you have time, try to **load the other dataset** as well. For example, if you did EuroSAT first, attempt to load the Palawan patches with similar code. Are there any differences in preprocessing needed (different image size or number of classes)? This will give you practice in adapting data loaders.

2. Building a Basic CNN Model

Now that our data is ready, we can define a CNN model for classification. We will start with a **basic CNN architecture**: a few convolutional layers for feature extraction, each followed by a non-linear activation (ReLU) and a downsampling (pooling) step, then some fully connected layers for classification. This kind of model is often called a *feature extractor* + *classifier* design, and is a common baseline for image classification [1].

> A basic CNN consists of **Convolutional layers**, **Pooling layers**, and a **fully-connected (Dense) layer** (or layers) before the output [1]. The conv layers act as feature extractors, the pooling layers reduce spatial resolution (making computation efficient and providing some translational invariance), and the dense layers learn to map the extracted features to class probabilities.

We'll implement the same conceptual model in both Keras and PyTorch.

2.1 Defining the CNN in TensorFlow/Keras

In Keras, we can use the Sequential API to stack layers. Our model will have the following structure for **scene classification**:

- **Input layer:** expects 64×64 RGB images (so shape `(64,64,3)`).
- **Conv2D layer 1:** e.g., 32 filters, 3×3 kernel, ReLU activation.
- **MaxPooling2D layer 1:** 2×2 pool (reduces image size by 2).
- **Conv2D layer 2:** e.g., 64 filters, 3×3, ReLU.
- **MaxPooling2D layer 2:** 2×2 pool.
- **Conv2D layer 3:** e.g., 128 filters, 3×3, ReLU.
- **MaxPooling2D layer 3:** 2×2 pool.
- **Flatten:** flatten the 2D feature maps into a 1D vector.
- **Dense layer:** e.g., 128 units, ReLU.
- **Output Dense layer:** number of units = number of classes (e.g., 10 for EuroSAT), with **softmax** activation (to produce class probabilities).

Let's code this:

```
from tensorflow.keras import layers, models

num_classes = 10  # EuroSAT has 10 classes (adjust if using a different dataset)
model = models.Sequential([
    layers.Input(shape=(64, 64, 3)),                        # input layer
    layers.Conv2D(32, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2)),
```

```
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

model.summary()
```

This prints a summary of the model architecture, showing each layer's output shape and number of parameters. Notice how the spatial dimensions reduce at each pooling: 64×64 → 32×32 → 16×16 after three poolings (since 64/(2*2*2) = 8 if we did three, but we used padding='same' which keeps dimensions before pooling, so actually: 64→32→16→8 after three pools). The final Flatten turns the 8×8 feature map with 128 filters into 8*8*128 = 8192 features, which the dense layer then processes.

Why ReLU? ReLU (Rectified Linear Unit) is a popular activation that sets negative values to 0 and keeps positive values linear. It introduces non-linearity so that the network can learn complex patterns, while being simple and avoiding the vanishing gradient problem associated with sigmoid/tanh for deep networks.

Think: What might happen if we add another Conv layer or increase the number of filters? More layers/filters can allow the model to learn more complex features, but also increase computation and may require more data to avoid overfitting. There is a trade-off. Our 2-3 conv layers are enough to grasp the concept.

2.2 Defining the CNN in PyTorch

In PyTorch, we define a subclass of `nn.Module` to create the model, or use `nn.Sequential`. We'll demonstrate using a custom class for clarity, which requires writing a `forward` method describing how data flows through the layers.

```
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)   # input 3
channels (RGB), 32 filters
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool  = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flat  = nn.Flatten()
        self.fc1   = nn.Linear(128 * 8 * 8, 128)  # 128 filters * 8*8 spatial
(assuming input 64x64 and 3 pools)
        self.fc2   = nn.Linear(128, num_classes)

        # Activation (ReLU and softmax can be applied in forward or defined
here)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)
```

```
    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = self.relu(self.conv3(x))
        x = self.pool(x)
        x = self.flat(x)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        # We will apply softmax after computing loss, so returning logits is
fine
        return x

model_pt = SimpleCNN(num_classes=10)
print(model_pt)
```

A few notes on this PyTorch implementation: - The input to `forward` is expected to be a tensor of shape `(batch_size, 3, 64, 64)` since PyTorch convolution expects channels-first format. This is why we set `nn.Conv2d(3, 32, ...)` with in_channels=3. - We used `padding=1` on conv layers to preserve spatial dimensions (so 64 stays 64 before pooling, then pooling halves it). After 3 pool layers, 64 → 32 → 16 → 8, so the feature map is 8×8 with 128 channels. We then flatten that for the linear layer. - We included `nn.Softmax(dim=1)` in the class, but note: for training, it's better to use raw logits and apply `CrossEntropyLoss` which internally applies softmax. We can choose to not use `self.softmax` in `forward` (as commented) to have the model output logits. - Printing `model_pt` will show the structure.

Device: If using a GPU, remember to move the model and data to the GPU. For example:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model_pt = model_pt.to(device)
```

Think: In Keras, we didn't explicitly add an activation after the final dense layer because we specified `activation='softmax'` inside the layer. In PyTorch, we typically output raw scores (logits) and apply softmax only for inference or let the loss function handle it. Always be mindful of whether your final layer outputs need an activation for your chosen loss.

Mini-Challenge: Try modifying the CNN architecture slightly: - What if you remove one Conv layer? (The model will be shallower – try adjusting the flatten dimensions accordingly.) - What if you add a fourth conv-pool pair? (The spatial size would go down to 4×4, and you'd need to adjust the linear layer input size.) - Experiment with these changes (in code or just mentally) and predict how it might affect model capacity and performance.

3. Compiling and Training the Model

With the models defined, the next step is to train them on our dataset. This involves specifying a **loss function** and **optimizer**, then running the training loop for several epochs.

Loss Function: For multi-class classification with one-hot labels, we use **Categorical Cross-Entropy** (also known as softmax cross-entropy). This loss measures the difference between the predicted probability distribution and the true distribution (which for one-hot is 0 for all classes except 1 for the true class). If using integer labels, we would use sparse categorical cross-entropy. Cross-entropy loss increases as the predicted probability of the wrong class increases – it penalizes confident wrong predictions heavily, forcing the model to improve [10].

Optimizer: We will use **Adam** (Adaptive Moment Estimation) as our optimizer. Adam is a variant of stochastic gradient descent that adapts the learning rate for each parameter, combining advantages of momentum and RMSprop [2]. It's a good default optimizer for many problems.

We also will track **metrics** like accuracy during training.

3.1 Training in TensorFlow/Keras

Keras makes training straightforward with `model.compile` and `model.fit`. Let's compile our Keras model:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

We passed the string `'categorical_crossentropy'` which is fine since our labels are one-hot (if using `label_mode='categorical'` as above). If our labels were integers (sparse), we'd use `'sparse_categorical_crossentropy'`.

Now, train the model with the training dataset and validate on the validation dataset:

```
epochs = 10  # you can start with 10, and later increase if needed
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

This will output the training loss and accuracy for each epoch, as well as validation loss and accuracy. For example, you might see something like:

```
Epoch 1/10: loss=... accuracy=... val_loss=... val_accuracy=...
Epoch 2/10: ...
...
```

Keep an eye on the accuracy: if it's increasing and validation accuracy is also increasing (or at least not diverging too much), training is proceeding well. If validation accuracy plateaus or starts decreasing while training accuracy keeps increasing, the model may be overfitting.

The `history` object contains the per-epoch metrics which you can use to plot learning curves if desired (e.g., `history.history['accuracy']` and `history.history['val_accuracy']`).

Training Time: With 10 epochs on ~20k images (if using full EuroSAT subset), this may take a few minutes on GPU. If using a smaller subset, it will be faster. You can adjust `epochs` or `batch_size` to tune speed vs. thoroughness.

Think: What does the loss value represent? It's the average cross-entropy across the batch. It doesn't directly correspond to accuracy, but generally a lower loss should correlate with higher accuracy. During training, we want to see loss decreasing over epochs.

3.2 Training in PyTorch

Training in PyTorch requires writing our own loop to iterate over batches, compute loss, and update weights. We'll use `torch.optim.Adam` and `nn.CrossEntropyLoss` (which combines softmax and cross-entropy in one).

```
import torch.optim as optim

# Move model to device (GPU if available)
model_pt = model_pt.to(device)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()  # for multi-class classification
optimizer = optim.Adam(model_pt.parameters(), lr=0.001)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model_pt.train()   # set to training mode
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()           # clear gradients
```

```
        outputs = model_pt(images)      # forward pass
        loss = criterion(outputs, labels)   # compute loss (expects raw outputs
and class indices)
        loss.backward()                 # backpropagate
        optimizer.step()                # update weights

        running_loss += loss.item() * images.size(0)
        # For accuracy: compare predicted class (max logit) with true label
        _, predicted = torch.max(outputs, 1)  # get index of max logit
        correct += (predicted == labels).sum().item()
        total += labels.size(0)
    epoch_loss = running_loss / total
    epoch_acc = correct / total

    # Validation
    model_pt.eval()  # evaluation mode
    val_correct = 0
    val_total = 0
    val_loss = 0.0
    with torch.no_grad():  # no gradient needed for eval
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model_pt(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            val_correct += (predicted == labels).sum().item()
            val_total += labels.size(0)
    val_loss = val_loss / val_total
    val_acc = val_correct / val_total

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Acc:
{epoch_acc:.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
```

This code manually loops through training data (`train_loader`) and does the forward-backward-update steps for each batch. We accumulate the total loss and correct predictions to compute average loss and accuracy for the epoch. Then we do a validation loop to compute val loss and accuracy.

PyTorch requires explicit zeroing of gradients (`optimizer.zero_grad()`) and stepping the optimizer. We use `torch.no_grad()` and `model.eval()` during validation to turn off dropout (if any) and gradient computation.

The printout will be similar to Keras, giving epoch metrics. You should see training loss decrease and training accuracy increase over epochs, and hopefully validation accuracy following suit.

Tip: If the model is training very slowly or not improving, ensure that the data is normalized properly and that the learning rate isn't too high or too low. Our choice of 0.001 for Adam is usually fine.

Mini-Challenge: The training loop above collects accuracy manually. For a challenge, try to also compute a **confusion matrix** on the validation set after training. You can do this by accumulating all `predicted` and `labels` for the val set and using `sklearn.metrics.confusion_matrix`. This will give deeper insight into which classes are confused by the model.

4. Evaluating Model Performance

After training, we want to evaluate how well our model performs on unseen data. We've been tracking validation accuracy, but now let's measure performance on a test set (if provided) or use the validation as a proxy. Key evaluation metrics for classification include **accuracy**, **precision/recall**, **F1-score**, and the **confusion matrix**.

For simplicity, we'll focus on accuracy and confusion matrix in this lab.

Confusion Matrix: A confusion matrix is a table that summarizes how often each actual class was predicted as each class by the model [3]. Rows might represent true classes and columns predicted classes (or vice versa). Diagonal elements are correct predictions, off-diagonals are mistakes. This helps identify if certain classes are often confused with each other.

4.1 Evaluation in Keras

If you have a separate test dataset, you can simply do:

```
test_loss, test_acc = model.evaluate(test_ds)
print(f"Test Accuracy: {test_acc:.3f}")
```

This gives overall accuracy. To get a confusion matrix, we need to get the predicted labels for each test image.

We can use `model.predict` on a dataset and then use `tf.math.confusion_matrix` or sklearn.

Example using TensorFlow:

```
import numpy as np
# Suppose we have a test dataset similar to val_ds
pred_probs = model.predict(val_ds)  # predict probabilities on validation set
y_pred = np.argmax(pred_probs, axis=1)  # predicted class indices
y_true = []
for _, labels in val_ds:
    # labels are one-hot vectors
    class_indices = np.argmax(labels.numpy(), axis=1)
```

```
    y_true.extend(class_indices)
y_true = np.array(y_true)

# Compute confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:")
print(cm)
```

This assumes `val_ds` is small enough to loop or that you have all labels. In practice, one might gather predictions batch by batch.

If using EuroSAT's 10 classes, the confusion matrix will be 10x10. Ideally, it should have large numbers on the diagonal. If certain off-diagonal entries are high, those classes are being confused. For instance, maybe **Highway** vs **River** might sometimes confuse if a river looks like a road in an image, etc., as an example scenario (EuroSAT classes can have some similarities).

We can also derive metrics like per-class accuracy from the confusion matrix.

4.2 Evaluation in PyTorch

In PyTorch, after training, we might use the validation or a test DataLoader to compute metrics. We partially did this in the validation loop. To get a confusion matrix:

```
# Assuming we have a test_loader or reuse val_loader
all_preds = []
all_labels = []
model_pt.eval()
with torch.no_grad():
    for images, labels in val_loader:
        images = images.to(device)
        outputs = model_pt(images)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds.cpu().numpy())
        all_labels.append(labels.numpy())
all_preds = np.concatenate(all_preds)
all_labels = np.concatenate(all_labels)

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(all_labels, all_preds)
print("Confusion Matrix:")
print(cm)
```

Now `cm[i,j]` is the number of samples of true class i predicted as class j.

Interpreting Results: - Check the **overall accuracy** (we calculated `val_acc` earlier). - Look at the confusion matrix to see which classes have lower accuracy. Are there specific confusions? For example, maybe in Palawan dataset, if classes were "water" vs "land", is the model sometimes predicting water as land or vice versa in certain conditions? Or in EuroSAT, maybe *Residential* vs *Industrial* get confused, etc. This analysis can guide future improvements (like collecting more data for those classes or using a more complex model).

Think: Why do we use more than just accuracy to evaluate? Accuracy gives an overall sense, but if the dataset is imbalanced or if certain classes matter more, we need more detail. The confusion matrix (and derived metrics like precision and recall) provides a class-wise breakdown. For instance, a high overall accuracy could hide the fact that one class is never predicted correctly [11]. Always examine where your model is "confused" [12]!

Mini-Challenge: Calculate the **per-class accuracy** from the confusion matrix. (Hint: per-class accuracy = correct predictions for class i / total samples of class i, which is `cm[i,i] / sum(cm[i,:])`.) Which class has the highest accuracy and which has the lowest? Can you think of why, based on the class characteristics?

5. Visualizing Sample Predictions

To cement understanding, let's visualize some predictions. We will take a few sample images from the test/validation set, run the model on them, and plot the image with its **predicted label** and **true label**. This helps connect the numerical performance with actual image content.

In Keras, we can directly index into our `val_ds` (which is a tf.data.Dataset). Alternatively, since we have numpy arrays from before, we can use those.

Here's an example using Matplotlib for a few images:

```
import matplotlib.pyplot as plt

# Get 5 examples from the validation set
for images, labels in val_ds.take(1):
    images = images.numpy()
    labels = labels.numpy()
    preds = model.predict(images)
    preds_cls = np.argmax(preds, axis=1)
    true_cls = np.argmax(labels, axis=1)

# Plot 5 images with predictions
class_names = train_ds.class_names  # this attribute exists if using
image_dataset_from_directory
plt.figure(figsize=(10, 8))
for i in range(5):
    ax = plt.subplot(2, 3, i+1)
    plt.imshow(images[i].astype("uint8"))  # images were scaled 0-1, but
```

```
Rescaling was applied, so should be 0-255 now
    pred_label = class_names[preds_cls[i]]
    true_label = class_names[true_cls[i]]
    color = "green" if preds_cls[i] == true_cls[i] else "red"
    plt.title(f"Pred: {pred_label}\nTrue: {true_label}", color=color)
    plt.axis("off")
plt.show()
```

This will display 5 images. The title above each image shows the predicted class and true class. If the prediction is correct, we color it green; if incorrect, red. Inspect these results: - Are the mistakes on images that look ambiguous even to you? - Are the correct predictions on images that have clear features?

For PyTorch, the approach is similar, but we need to get some images from `val_loader`. We also need to inverse-transform them because we normalized the images. If we used `Normalize(mean=0.5, std=0.5)`, we can invert that by `img * 0.5 + 0.5` to get back to 0-1 scale, then multiply 255 for display:

```
import numpy as np
# Get one batch from val_loader
data_iter = iter(val_loader)
images, labels = next(data_iter)
images, labels = images.to(device), labels.to(device)
outputs = model_pt(images)
_, preds = torch.max(outputs, 1)
images = images.cpu().numpy()
labels = labels.cpu().numpy()
preds = preds.cpu().numpy()

# Inverse normalize (if we normalized with mean=std=0.5)
images = images * 0.5 + 0.5  # this brings pixels back to [0,1]
images = np.transpose(images, (0,2,3,1))  # reshape from (N,C,H,W) to (N,H,W,C)
for plotting

plt.figure(figsize=(10,8))
for i in range(5):
    ax = plt.subplot(2,3,i+1)
    plt.imshow(images[i])
    pred_label = full_dataset.classes[preds[i]]
# ImageFolder assigns a classes attribute with names
    true_label = full_dataset.classes[labels[i]]
    color = "green" if preds[i] == labels[i] else "red"
    plt.title(f"Pred: {pred_label}\nTrue: {true_label}", color=color)
    plt.axis('off')
plt.show()
```

This does the same for PyTorch: we take one batch of images, get predictions, then plot a few. Again, analyze which ones are wrong and why.

Think: If you see a systematic pattern in the mistakes (e.g., all misclassified water images are turbid or have clouds), it might hint at limitations of the model or data issues. Visualization is a powerful tool to gain intuition beyond the numbers.

Mini-Challenge: Find one example where the model is **incorrect** and hypothesize why. Is the image low quality, or the class definitions overlapping, or maybe the model just hasn't learned that feature? Write down your thoughts.

6. Optional: Semantic Segmentation (Advanced)

If you are comfortable with the classification task, an exciting extension is **semantic segmentation** – predicting a class for each pixel in an image, instead of one label for the whole image. In EO contexts, segmentation could mean creating a land cover map from an image (each pixel labeled as forest, water, urban, etc.) or extracting specific features (e.g., roads or buildings as a binary mask).

For this optional section, we'll outline a simple segmentation experiment: predicting a **binary mask** from an image patch. For instance, given a satellite image, predict which pixels belong to water (1) vs land (0), or some similar binary segmentation.

Data for Segmentation: You would need pairs of input images and target masks of the same size. If using the Palawan data, perhaps a mask is available (for example, a coastline mask or forest cover mask). For illustration, let's assume we have images and corresponding binary masks (of shape 64×64 as well).

Model Architecture: A common architecture for segmentation is the **U-Net**, which has an encoder-decoder structure (downsampling path and upsampling path) [4]. Due to time constraints, we'll implement a simplified version: - **Encoder:** similar to our classification CNN, conv layers with pooling to extract features. - **Decoder:** conv-transpose (upsampling) layers to bring the spatial resolution back to original and predict a mask.

A very simple segmentation model could be:

- Conv(16 filters) + ReLU
- Conv(32 filters) + ReLU (with downsampling via stride 2 or pooling)
- ConvTranspose(16 filters) + ReLU (upsampling)
- ConvTranspose(1 filter) for output mask (with sigmoid activation to get probabilities 0-1 per pixel)

This is **not** a full U-Net (which would have skip connections from encoder to decoder), but it's a start.

6.1 Segmentation Model in PyTorch (example)

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

```
        # Encoder
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        # We will use stride=2 in conv2 to downsample instead of pooling
        self.conv2.stride = (2,2)  # downsample to half resolution

        # Decoder
        self.convT1 = nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2,
padding=1, output_padding=1)
        self.convT2 = nn.ConvTranspose2d(16, 1, kernel_size=3, padding=1)

        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Encoder
        x = self.relu(self.conv1(x))    # out: 64x64x16
        x = self.relu(self.conv2(x))    # out: 32x32x32 (due to stride 2
downsampling)
        # Decoder
        x = self.relu(self.convT1(x))   # out: 64x64x16 (upsampled back to
original size)
        x = self.convT2(x)             # out: 64x64x1 (mask logits)
        x = self.sigmoid(x)            # output mask in [0,1]
        return x
```

This model takes a 64×64 image and outputs a 64×64 single-channel mask. We used `ConvTranspose2d` (also known as deconvolution) to upsample. The `output_padding=1` in `convT1` ensures the output size matches 64 when stride=2.

Loss for Segmentation: For binary segmentation, we use **Binary Cross-Entropy (BCE)** loss per pixel. In PyTorch, `nn.BCELoss()` if the output is already sigmoid probabilities, or better `nn.BCEWithLogitsLoss()` if output logits (and skip the sigmoid in model). Since we applied sigmoid in model, we'll use BCELoss.

Training loop: Very similar to classification, except: - Labels will be 2D masks (0/1 per pixel). - We compute loss = BCELoss(output, target) which averages over all pixels. - We might track metrics like IoU (intersection over union) or simply pixel accuracy for evaluation.

Due to time, we won't detail the full training loop here. It would loop over image-mask pairs.

Usage of GeoTorch/TorchGeo: If your segmentation task is geospatial, libraries like TorchGeo can handle georeferenced tiling and loading. But conceptually, it's the same deep learning process.

6.2 (Optional) Segmentation in Keras

If you prefer Keras, you could similarly define a model using `Conv2D` and `Conv2DTranspose`. Keras even has an example of a simple U-Net in its documentation. The main difference is using `'sigmoid'` activation on the final Conv2D (with 1 filter) and compile with `loss=tf.keras.losses.BinaryCrossentropy()`.

6.3 Testing the Segmentation Model

After training a segmentation model, you would test it by giving it an image and visualizing the predicted mask vs. the true mask: - You could use `matplotlib.pyplot.imshow()` with a color map (e.g., `cmap='gray'`) to show the mask. - Overlaying the mask on the image (with some transparency) is also insightful, to see if the model highlights the correct regions.

For example, after training:

```
model_seg.eval()
test_image, test_mask = segmentation_dataset[0]  # first sample
pred_mask = model_seg(test_image.unsqueeze(0).to(device))  # add batch dim
pred_mask = pred_mask.squeeze().cpu().detach().numpy()
plt.subplot(1,3,1); plt.imshow(test_image.permute(1,2,0).numpy());
plt.title("Image")
plt.subplot(1,3,2); plt.imshow(test_mask.numpy(), cmap='gray'); plt.title("True
Mask")
plt.subplot(1,3,3); plt.imshow(pred_mask, cmap='gray'); plt.title("Predicted
Mask")
plt.show()
```

(This assumes `segmentation_dataset` yields (image, mask) and both are normalized appropriately.)

If done correctly, you would see the model's mask highlighting, say, water areas similarly to the true mask.

Think: Segmentation is more challenging than classification because the model must effectively do classification at every pixel. This requires more complex reasoning and often more data or more sophisticated models. Our simple model may not be very accurate, but it demonstrates the principle of an encoder-decoder CNN.

Mini-Challenge: If you have segmentation data, try training the above model for a few epochs. Monitor a metric like mean IoU if possible. Does the model seem to be learning? If not, consider: is the model capacity enough? Is the dataset too small? Segmentation often requires a lot of data or pretraining.

Conclusion and Next Steps

Congratulations on making it through this intensive lab! You have built and trained CNN models for image classification using two different frameworks and even dipped your toes into segmentation. Through this process, you practiced the end-to-end workflow of deep learning on EO data: loading data, designing a model, training, evaluating, and interpreting results.

Key takeaways: - **CNN Architecture:** Convolutional layers extract localized features (like edges, textures) which are crucial for images, and stacking conv layers allows hierarchical feature learning (simple features combine into complex patterns). Pooling reduces spatial size and helps focus on important features. - **Framework Differences:** Keras provides high-level simplicity (quick to prototype, with less code for training loop), while PyTorch offers low-level control (which is useful for debugging and custom training procedures). Both are widely used – understanding both makes you a more versatile practitioner. - **Geospatial Considerations:** We used generic CNNs, but real EO applications often involve multispectral data, varying resolutions, and georeferenced tiling. Tools like Google Earth Engine (for data prep) and TorchGeo (for data loading and transforms) can significantly ease the handling of such complexities [9]. For example, TorchGeo includes datasets and models specialized for remote sensing. - **Evaluation:** Always look beyond accuracy. Use confusion matrices to identify class confusions [3], and visualize predictions to gain intuition on what the model gets right or wrong.

Further Exploration: - Try using **data augmentation** (e.g., random flips or rotations) to see if it improves robustness. Keras allows adding augmentation layers (like `layers.RandomFlip`, `layers.RandomRotation`) in the model or using `ImageDataGenerator`. In PyTorch, you can add transforms like `transforms.RandomHorizontalFlip()` in the dataset pipeline. - Experiment with a **pretrained model** (transfer learning). For instance, Keras has applications like `tf.keras.applications.MobileNetV2` you could fine-tune on this data, or use a pretrained ResNet in PyTorch. This often boosts performance significantly on small datasets. - For segmentation, explore a full **U-Net** architecture and try a multi-class segmentation if you have the data (e.g., labeling multiple land cover types, not just binary).

Keep these materials as a reference. The concepts and patterns you practiced here will apply to many AI for Earth Observation tasks. Happy coding and exploring!

---

[1] Convolution Neural Network for image classification — Research Computing and Data Workshop
https://clemsonciti.github.io/rcde_workshops/python_deep_learning/07-Convolution-Neural-Network.html

[2] Adam Optimizer in Tensorflow - GeeksforGeeks
https://www.geeksforgeeks.org/python/adam-optimizer-in-tensorflow/

[3] [11] [12] Confusion Matrix: How To Use It & Interpret Results [Examples]
https://www.v7labs.com/blog/confusion-matrix-guide

[4] Improve Image Segmentation with U-Net's Deep Learning
https://viso.ai/deep-learning/u-net-a-comprehensive-guide-to-its-architecture-and-applications/

[5] [6] MuafiraThasni/eurosat-dataset-with-image · Datasets at Hugging Face
https://huggingface.co/datasets/MuafiraThasni/eurosat-dataset-with-image

7 Satellite images classification using the EuroSAT dataset | by Roman Nagy | Medium
https://rmnng.medium.com/satellite-images-classification-using-the-eurosat-dataset-13f7fd1edc42

8 About Google Earth Engine
https://developers.google.com/earth-engine/guides

9 Geospatial deep learning with TorchGeo – PyTorch
https://pytorch.org/blog/geospatial-deep-learning-with-torchgeo/

10 Binary and Categorical Cross-entropy with Keras | by Francesco Franco | CodeX | Medium
https://medium.com/codex/binary-and-categorical-cross-entropy-with-keras-c0fce246e572