Week 3: Teaching Machines to See Landscapes — Building CNNs for Earth Observation

Learning Objectives

By the end of Week 3, students will be able to:

1. **Build and train a convolutional neural network** specifically for land-cover classification using Sentinel-2 imagery
2. **Preprocess geospatial imagery** for deep learning, including tiling, normalization, label encoding, and spatial train/test splitting
3. **Visualize and interpret learned features** through filter visualization, activation maps, and feature space analysis
4. **Evaluate model performance** using appropriate metrics for geospatial classification (accuracy, precision, recall, F1, IoU, confusion matrices)
5. **Document a reproducible training pipeline** that serves as the foundation for your capstone project

Opening Discussion and Review

Review of Week 2 Key Concepts

Before diving into building your own CNN, reflect on Week 2's insights:

- What did you learn about how ImageNet models fail on satellite imagery?
- Why does domain shift occur, and what does it reveal about what models have learned?
- What are the fundamental differences between natural images and Earth observation data?
- How do ethical considerations shape the way we approach Earth observation AI?

Week 2 demonstrated that pre-trained ImageNet models, while powerful for natural images, struggle with satellite imagery due to fundamental differences in scale, spectral information, temporal dynamics, and geographic context. This week, you'll address these challenges by training a CNN specifically on Earth observation data.

Today's Learning Journey

This week represents a major milestone in your GeoAI education: you'll build and train your first deep learning model for Earth observation. This is where theory meets practice, where conceptual understanding transforms into working code, and where you begin to see how machines can learn to interpret satellite imagery.

The journey has four interconnected stages:

First, you'll master the technical foundations of CNNs—not just what they are, but how they work at a detailed level. You'll understand convolution operations, pooling strategies, activation functions, and how these components combine to create hierarchical feature representations. This deep understanding is essential for debugging, tuning, and innovating beyond standard approaches.

Second, you'll learn the craft of preparing geospatial data for machine learning. This isn't simply downloading imagery and feeding it to a model. It requires careful attention to spatial resolution, spectral band selection, normalization strategies, label quality, and—critically—spatial train/test splitting that accounts for autocorrelation. Poor data preparation is the most common cause of model failure.

Third, you'll experience the training process firsthand—watching loss curves descend, monitoring validation accuracy, diagnosing overfitting, and making decisions about when to stop training. You'll learn that training neural networks is as much art as science, requiring intuition developed through experience.

Fourth, you'll learn to interpret what your model has learned through visualization techniques. Filter visualization shows what patterns individual neurons detect. Activation maps reveal which image regions influence predictions. Feature space analysis shows how the model represents different land-cover types. These interpretability tools are essential for building trust in AI systems and diagnosing failures.

By week's end, you'll have a trained CNN, a documented training pipeline, and the skills to evaluate and interpret model performance. This forms the foundation for everything that follows: transfer learning (Week 4), self-supervised learning (Weeks 5-6), and foundation models (Weeks 7-10).

Core Content: Convolutional Neural Networks for Earth Observation

Deep Dive into Convolutional Operations

To truly understand CNNs, we must understand convolution—the mathematical operation at their heart. While Week 2 introduced CNNs conceptually, this week we'll examine how

convolution works in detail.

The Mathematics of Convolution

A **convolution** is a mathematical operation that combines two functions to produce a third function. In image processing, we convolve an image with a **kernel** (also called a filter)—a small matrix of numbers that slides across the image.

Consider a simple 3×3 kernel applied to a grayscale image:

```
Plain Text

Kernel:           Image patch:        Result:
[-1  0  1]        [100 120 110]
[-2  0  2]   *    [105 125 115]   =  (sum of element-wise products)
[-1  0  1]        [110 130 120]
```

The convolution operation:

1. Places the kernel over a 3×3 region of the image
2. Multiplies each kernel value by the corresponding image pixel
3. Sums all products to produce a single output value
4. Slides the kernel to the next position and repeats

This particular kernel detects vertical edges—it responds strongly when there's a brightness change from left to right.

Why Convolution Works for Images

Convolution has several properties that make it ideal for image analysis:

Local connectivity: Each output value depends only on a small local region of the input. This matches the structure of visual information, where nearby pixels are related.

Parameter sharing: The same kernel is applied across the entire image. This means the network can detect the same pattern (like an edge or texture) anywhere in the image, regardless of position. This dramatically reduces the number of parameters compared to fully connected layers.

Translation invariance: Because the same kernel slides across the image, the network can recognize patterns regardless of their location. A forest looks like a forest whether it's in the top-left or bottom-right of the image.

Hierarchical composition: By stacking multiple convolutional layers, networks build hierarchical representations. Early layers detect simple patterns (edges, colors). Middle

layers combine these into textures and shapes. Deep layers recognize complex structures and objects.

Convolution Hyperparameters

Several hyperparameters control how convolution operates:

Kernel size: The spatial dimensions of the filter (typically 3×3, 5×5, or 7×7). Larger kernels capture more spatial context but have more parameters and computational cost. Modern architectures favor small (3×3) kernels stacked in multiple layers.

Stride: How many pixels the kernel moves between applications. Stride 1 means the kernel moves one pixel at a time. Stride 2 means it skips every other pixel, reducing output dimensions by half. Larger strides reduce computational cost but lose spatial resolution.

Padding: Adding pixels around the image border to control output size. "Valid" padding means no padding—output is smaller than input. "Same" padding adds enough border pixels so output has the same spatial dimensions as input. Padding preserves spatial resolution through the network.

Dilation: Spacing between kernel elements. Standard convolution has dilation 1 (kernel elements are adjacent). Dilation 2 means kernel elements are separated by one pixel, expanding the receptive field without adding parameters. Dilated convolutions are useful for capturing large-scale spatial context.

Multi-Channel Convolution

Satellite imagery has multiple spectral bands—Sentinel-2 provides 10+ channels. How does convolution handle multi-channel input?

For an input with C channels and a kernel with F filters:

- Each filter is a 3D tensor: (kernel_height, kernel_width, C)
- The filter convolves across all input channels simultaneously
- Each filter produces one output channel
- With F filters, output has F channels

For example, with 10-channel Sentinel-2 input and 64 filters:

- Each filter is 3×3×10 (spatial dimensions × input channels)
- Convolution produces 64 output channels
- Each output channel represents a different learned feature

This multi-channel convolution allows CNNs to learn combinations of spectral bands—essentially learning custom spectral indices optimized for the task.

Building Blocks of CNN Architectures

A complete CNN architecture combines multiple types of layers, each serving specific purposes.

Convolutional Layers: Feature Detection

Convolutional layers are the workhorses of CNNs, learning to detect spatial patterns. In PyTorch:

```python
conv_layer = nn.Conv2d(
    in_channels=10,       # Input channels (e.g., Sentinel-2 bands)
    out_channels=64,      # Number of filters to learn
    kernel_size=3,        # 3×3 kernels
    stride=1,             # Move one pixel at a time
    padding=1             # "Same" padding to preserve dimensions
)
```

This layer has $10 \times 64 \times 3 \times 3 = 5{,}760$ parameters (plus 64 bias terms). Each of the 64 filters learns to detect a different pattern in the 10-channel input.

Design considerations for Earth observation:

- More input channels than natural images (10+ spectral bands vs. 3 RGB)
- May want larger receptive fields to capture spatial context
- Consider dilated convolutions for multi-scale features

Activation Functions: Introducing Non-Linearity

Without activation functions, stacking multiple linear layers would still produce a linear transformation. Activation functions introduce non-linearity, enabling networks to learn complex patterns.

ReLU (Rectified Linear Unit): The most common activation function.

```python
output = max(0, input)
```

ReLU is simple, computationally efficient, and works well in practice. It addresses the vanishing gradient problem that plagued earlier activation functions like sigmoid and tanh.

Variants:

- **Leaky ReLU**: `output = max(0.01 * input, input)` — allows small negative values, preventing "dead neurons"
- **ELU (Exponential Linear Unit)**: Smooth negative values, can improve learning
- **GELU (Gaussian Error Linear Unit)**: Used in transformers, smooth approximation of ReLU

For most applications, standard ReLU works well. Use variants if you encounter training difficulties.

Pooling Layers: Downsampling and Invariance

Pooling reduces spatial dimensions, making the network more computationally efficient and providing some translation invariance.

Max Pooling: Takes the maximum value in each local region (typically 2×2).

```python
pool_layer = nn.MaxPool2d(kernel_size=2, stride=2)
```

This reduces spatial dimensions by half. A 64×64 image becomes 32×32.

Average Pooling: Takes the mean value in each region. Less common than max pooling but sometimes useful for smoother representations.

Global Average Pooling: Averages across entire spatial dimensions, reducing each channel to a single value. Often used before final classification layers instead of flattening + fully connected layers.

Design considerations for Earth observation:

- Pooling reduces spatial resolution—important for small features like parcels
- Consider using stride in convolutions instead of pooling for more learnable downsampling
- Global average pooling reduces parameters and overfitting

Normalization Layers: Stabilizing Training

Batch Normalization standardizes layer inputs, making training more stable and enabling higher learning rates.

```python
bn_layer = nn.BatchNorm2d(num_features=64)
```

Batch normalization computes mean and standard deviation across the batch for each channel, then normalizes:

```
Python

output = (input - mean) / sqrt(variance + epsilon)
output = gamma * output + beta  # Learnable scale and shift
```

Benefits:

- Reduces internal covariate shift (changing input distributions during training)
- Allows higher learning rates
- Provides some regularization effect
- Reduces sensitivity to initialization

Design considerations:

- Essential for training deep networks
- Batch size affects batch norm statistics—use larger batches if possible
- For small batches, consider Group Normalization or Layer Normalization

Dropout: Preventing Overfitting

Dropout randomly sets a fraction of neuron activations to zero during training, preventing the network from relying too heavily on any particular feature.

```
Python

dropout_layer = nn.Dropout(p=0.5)  # Drop 50% of activations
```

During training, each neuron has probability p of being set to zero. During evaluation, all neurons are active but outputs are scaled by (1-p) to maintain expected values.

Benefits:

- Reduces overfitting by preventing co-adaptation of neurons
- Ensemble effect—training many sub-networks simultaneously
- Simple and effective regularization

Design considerations:

- Typical dropout rates: 0.2-0.5
- Use higher dropout for larger networks or smaller datasets
- Apply dropout after activation functions, before final layers

Fully Connected Layers: Final Classification

After convolutional and pooling layers extract features, fully connected layers combine them for classification.

Python

```
fc_layer = nn.Linear(in_features=512, out_features=num_classes)
```

This layer connects every input feature to every output class. For a 4-class land-cover problem, the final layer has 512 × 4 = 2,048 parameters.

Modern trend: Minimize fully connected layers to reduce parameters. Global average pooling + single linear layer is common.

Designing Your First CNN for Land-Cover Classification

Now let's design a complete CNN architecture for classifying Sentinel-2 imagery into land-cover classes.

Architecture Design Principles

Start simple: Begin with a shallow network (3-4 convolutional layers) and increase complexity only if needed. Simple models train faster, are easier to debug, and often generalize better on small datasets.

Progressive downsampling: Gradually reduce spatial dimensions while increasing channel depth. This creates a pyramid of features from high-resolution/low-level to low-resolution/high-level.

Consistent pattern: Use repeating blocks (conv → batch norm → ReLU → pool) for clean, maintainable code.

Appropriate output: Match output dimensions to your task (number of land-cover classes).

Example Architecture: SimpleCNN

Here's a complete architecture for land-cover classification:

Python

```
import torch
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self, num_bands=10, num_classes=4):
```

```
        super(SimpleCNN, self).__init__()

        # Block 1: 10 → 32 channels
        self.conv1 = nn.Conv2d(num_bands, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU(inplace=True)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Block 2: 32 → 64 channels
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU(inplace=True)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Block 3: 64 → 128 channels
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.relu3 = nn.ReLU(inplace=True)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Global average pooling
        self.global_pool = nn.AdaptiveAvgPool2d(1)

        # Classifier
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(128, num_classes)

    def forward(self, x):
        # Block 1
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Block 2
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # Block 3
        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu3(x)
        x = self.pool3(x)

        # Global pooling and classification
        x = self.global_pool(x)
```

```
        x = x.view(x.size(0), -1)  # Flatten
        x = self.dropout(x)
        x = self.fc(x)

        return x
```

Architecture summary:

- Input: (batch, 10, H, W) — 10 Sentinel-2 bands
- After block 1: (batch, 32, H/2, W/2)
- After block 2: (batch, 64, H/4, W/4)
- After block 3: (batch, 128, H/8, W/8)
- After global pool: (batch, 128, 1, 1)
- Output: (batch, num_classes)

Parameter count: ~50,000 parameters (very small by modern standards)

This architecture is simple but effective for land-cover classification with limited training data.

Data Preparation for Geospatial Machine Learning

Proper data preparation is critical for success. Poor data quality or inappropriate preprocessing is the most common cause of model failure.

Defining Land-Cover Classes

The first step is defining what you want to classify. For your study region, consider:

Parcelización case study (Los Lagos):

- Forest (native and plantation)
- Agriculture (active fields)
- Parcels (subdivided residential)
- Water bodies
- (Optional: Bare soil, Urban, Grassland)

Megadrought case study (Central Chile):

- Healthy vegetation
- Stressed vegetation
- Bare soil / dry areas

- Water bodies
- Urban areas

Lake ecosystem case study (Lake Llanquihue):

- Clear water
- Turbid water / algal blooms
- Vegetation (riparian, aquatic)
- Agriculture (watershed)
- Forest

Design principles:

- 4-6 classes is manageable for initial experiments
- Classes should be spectrally and spatially distinct
- Balance between detail and data availability
- Consider hierarchical classification (coarse → fine)

Collecting Training Data

You need labeled examples for each class. Several approaches:

Manual digitization:

- Use QGIS to draw polygons around representative areas
- Verify with high-resolution imagery (Google Earth, Planet)
- Ensure examples span your study region
- Aim for 100-500 pixels per class minimum

Existing land-cover products:

- ESA WorldCover (10m global land cover)
- Dynamic World (10m near-real-time land cover)
- National datasets (if available for Chile)
- Caveat: May have errors; verify samples

Stratified sampling:

- Distribute samples across your AOI
- Include examples from different seasons/conditions
- Avoid spatial clustering
- Balance class representation

Quality control:

- Visually inspect all training areas
- Remove ambiguous or mixed pixels
- Document label sources and confidence
- Consider multiple labelers for agreement

Image Tiling and Patch Extraction

Satellite imagery is too large to process as single images. We tile it into smaller patches.

Tile size considerations:

- 64×64 to 256×256 pixels common
- Larger tiles: More spatial context, more memory, fewer training examples
- Smaller tiles: Less context, less memory, more examples
- For 10m Sentinel-2: 128×128 tiles = 1.28km × 1.28km

Extraction strategy:

Python

```python
def extract_patches(image, labels, patch_size=128, stride=64):
    """
    Extract patches from image and corresponding labels.

    Args:
        image: (C, H, W) numpy array
        labels: (H, W) numpy array
        patch_size: Size of square patches
        stride: Step size between patches (overlap if < patch_size)

    Returns:
        patches: List of (C, patch_size, patch_size) arrays
        patch_labels: List of (patch_size, patch_size) arrays
    """
    patches = []
    patch_labels = []

    C, H, W = image.shape

    for i in range(0, H - patch_size + 1, stride):
        for j in range(0, W - patch_size + 1, stride):
            patch = image[:, i:i+patch_size, j:j+patch_size]
            label = labels[i:i+patch_size, j:j+patch_size]
```

```
            # Only keep patches with valid labels
            if np.any(label > 0):
                patches.append(patch)
                patch_labels.append(label)

    return patches, patch_labels
```

Overlap strategy: Using stride < patch_size creates overlapping patches, increasing training examples but introducing correlation.

Normalization and Preprocessing

Proper normalization is essential for neural network training.

Sentinel-2 reflectance values range from 0-10,000 (scaled surface reflectance). We need to normalize to a range suitable for neural networks.

Approach 1: Min-Max normalization to [0, 1]:

Python

```
normalized = (image - image.min()) / (image.max() - image.min())
```

Simple but sensitive to outliers (clouds, shadows).

Approach 2: Percentile clipping + normalization:

Python

```
p2, p98 = np.percentile(image, [2, 98])
clipped = np.clip(image, p2, p98)
normalized = (clipped - p2) / (p98 - p2)
```

More robust to outliers.

Approach 3: Standardization (zero mean, unit variance):

Python

```
mean = image.mean(axis=(1, 2), keepdims=True)
std = image.std(axis=(1, 2), keepdims=True)
standardized = (image - mean) / (std + 1e-8)
```

Matches typical neural network assumptions.

Recommended for Sentinel-2:

Python

```
# Compute statistics from training set
train_mean = train_images.mean(axis=(0, 2, 3))  # Per-band mean
train_std = train_images.std(axis=(0, 2, 3))    # Per-band std

# Apply to all data
normalized = (image - train_mean[:, None, None]) / (train_std[:, None, None]
+ 1e-8)
```

This standardizes each band independently using training set statistics.

Label Encoding

Convert class names to integers:

- Forest: 0
- Agriculture: 1
- Parcels: 2
- Water: 3

For pixel-wise classification, labels are 2D arrays with class indices at each pixel.

For patch-level classification, you might use:

- **Single label per patch**: Majority class within patch
- **Pixel-wise labels**: Full 2D label array for semantic segmentation

Spatial Train/Test Splitting

Critical consideration: Spatial autocorrelation means nearby pixels are similar. Random splitting can leak information.

Wrong approach (random split):

Python

```
# DON'T DO THIS
train_patches, test_patches = train_test_split(all_patches, test_size=0.3)
```

If training and test patches are spatially close, the model can memorize locations rather than learn features.

Correct approach (spatial split):

Python

```
# Split by geographic regions
# Option 1: Split AOI into spatial blocks
train_region = aoi.filter(ee.Filter.bounds(train_bbox))
test_region = aoi.filter(ee.Filter.bounds(test_bbox))

# Option 2: Use spatial clustering
from sklearn.cluster import KMeans
coords = np.array([patch.centroid for patch in patches])
clusters = KMeans(n_clusters=5).fit_predict(coords)
train_idx = np.where(clusters < 4)[0]  # 4 clusters for training
test_idx = np.where(clusters == 4)[0]   # 1 cluster for testing
```

This ensures training and test data are spatially separated, providing a more realistic evaluation.

The Training Process: From Random Weights to Learned Features

Training a neural network involves iteratively adjusting parameters to minimize a loss function. Understanding this process is essential for successful model development.

Loss Functions for Classification

The **loss function** quantifies how wrong the model's predictions are. For classification, we use:

Cross-Entropy Loss (most common):

Python

```
loss_fn = nn.CrossEntropyLoss()
```

Cross-entropy measures the difference between predicted probability distributions and true labels. For a single example:

Plain Text

```
Loss = -log(p_correct_class)
```

Where p_correct_class is the predicted probability for the true class.

If the model predicts the correct class with 90% confidence, loss = -log(0.9) = 0.105. If it predicts with 10% confidence, loss = -log(0.1) = 2.303.

Lower loss = better predictions.

Weighted Cross-Entropy (for imbalanced classes):

```python
class_weights = torch.tensor([1.0, 2.0, 3.0, 1.5])  # Higher weight for rare classes
loss_fn = nn.CrossEntropyLoss(weight=class_weights)
```

This penalizes errors on rare classes more heavily, encouraging the model to learn them.

Optimization Algorithms

The **optimizer** updates model parameters to minimize loss. Several algorithms exist:

SGD (Stochastic Gradient Descent):

```python
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Classic algorithm: move parameters in the direction opposite to the gradient.

- `lr` (learning rate): Step size for updates
- `momentum`: Accumulates gradients over time, smoothing updates

Adam (Adaptive Moment Estimation):

```python
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Adaptive learning rates for each parameter. Generally works well with default settings.

- Combines momentum and adaptive learning rates
- Less sensitive to learning rate choice
- Most popular optimizer for deep learning

AdamW (Adam with Weight Decay):

```python
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)
```

Adam with improved weight decay regularization. Often performs better than standard Adam.

Recommendation: Start with Adam or AdamW with default settings. Adjust learning rate if needed.

Learning Rate Scheduling

The learning rate controls how quickly the model learns. Too high: unstable training. Too low: slow convergence.

Learning rate schedules adjust the learning rate during training:

Step decay:

```python
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10,
gamma=0.1)
```

Reduces learning rate by factor of gamma every step_size epochs.

Cosine annealing:

```python
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50)
```

Smoothly decreases learning rate following a cosine curve.

ReduceLROnPlateau:

```python
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5
)
```

Reduces learning rate when validation loss stops improving. Adaptive and effective.

The Training Loop

A complete training loop:

```python
def train_epoch(model, train_loader, criterion, optimizer, device):
    """Train for one epoch."""
    model.train()  # Set to training mode
    running_loss = 0.0
    correct = 0
```

```
    total = 0

    for images, labels in train_loader:
        # Move data to device (GPU if available)
        images = images.to(device)
        labels = labels.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        loss.backward()

        # Update parameters
        optimizer.step()

        # Track metrics
        running_loss += loss.item() * images.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / total
    epoch_acc = 100. * correct / total

    return epoch_loss, epoch_acc

def validate(model, val_loader, criterion, device):
    """Validate model."""
    model.eval()  # Set to evaluation mode
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # No gradient computation
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = outputs.max(1)
```

```
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / total
    epoch_acc = 100. * correct / total

    return epoch_loss, epoch_acc

# Complete training loop
num_epochs = 50
best_val_acc = 0.0

train_losses = []
train_accs = []
val_losses = []
val_accs = []

for epoch in range(num_epochs):
    # Train
    train_loss, train_acc = train_epoch(
        model, train_loader, criterion, optimizer, device
    )

    # Validate
    val_loss, val_acc = validate(
        model, val_loader, criterion, device
    )

    # Update learning rate
    scheduler.step(val_loss)

    # Save metrics
    train_losses.append(train_loss)
    train_accs.append(train_acc)
    val_losses.append(val_loss)
    val_accs.append(val_acc)

    # Print progress
    print(f'Epoch {epoch+1}/{num_epochs}:')
    print(f'  Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%')
    print(f'  Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%')

    # Save best model
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model.state_dict(), 'best_model.pth')
        print(f'  → New best model saved!')
```

Monitoring Training: Loss Curves and Overfitting

Loss curves show how training and validation loss change over epochs. They reveal important patterns:

Healthy training:

- Both train and val loss decrease
- Val loss tracks train loss closely
- Model is learning and generalizing

Underfitting:

- Both losses remain high
- Model hasn't learned the task
- Solutions: Train longer, increase model capacity, check data quality

Overfitting:

- Train loss decreases but val loss increases
- Model memorizes training data but doesn't generalize
- Solutions: More data, regularization (dropout, weight decay), early stopping

Early stopping: Stop training when validation loss stops improving:

Python

```python
patience = 10  # Wait 10 epochs for improvement
best_val_loss = float('inf')
epochs_without_improvement = 0

for epoch in range(num_epochs):
    train_loss, train_acc = train_epoch(...)
    val_loss, val_acc = validate(...)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        epochs_without_improvement = 0
        # Save model
    else:
        epochs_without_improvement += 1

    if epochs_without_improvement >= patience:
        print(f'Early stopping at epoch {epoch+1}')
        break
```

Model Evaluation: Beyond Accuracy

Accuracy alone is insufficient for evaluating geospatial classification models. We need multiple metrics to understand performance.

Classification Metrics

Accuracy: Fraction of correct predictions.

```
Plain Text

Accuracy = (TP + TN) / (TP + TN + FP + FN)
```

Simple but misleading for imbalanced classes.

Precision: Of predicted positives, how many are actually positive?

```
Plain Text

Precision = TP / (TP + FP)
```

Important when false positives are costly.

Recall (Sensitivity): Of actual positives, how many did we predict?

```
Plain Text

Recall = TP / (TP + FN)
```

Important when false negatives are costly.

F1 Score: Harmonic mean of precision and recall.

```
Plain Text

F1 = 2 * (Precision * Recall) / (Precision + Recall)
```

Balances precision and recall.

Per-class metrics: Compute precision, recall, F1 for each class separately. Reveals which classes the model struggles with.

Confusion Matrix

A confusion matrix shows predicted vs. actual classes:

```
Plain Text
```

```
                Predicted
              F    A    P    W
Actual   F  [85    5    8    2]
         A  [ 3   92    4    1]
         P  [12    6   78    4]
         W  [ 1    2    1   96]
```

Rows: Actual classes (F=Forest, A=Agriculture, P=Parcels, W=Water) Columns: Predicted classes Diagonal: Correct predictions Off-diagonal: Confusion between classes

This reveals:

- Forest sometimes confused with Parcels (8 cases)
- Parcels sometimes confused with Forest (12 cases)
- Water is well-classified (96% accuracy)

Intersection over Union (IoU)

For semantic segmentation (pixel-wise classification), IoU measures spatial overlap:

Plain Text

```
IoU = (Predicted ∩ Actual) / (Predicted ∪ Actual)
```

IoU ranges from 0 (no overlap) to 1 (perfect overlap).

Mean IoU (mIoU): Average IoU across all classes. Common metric for segmentation.

Spatial Evaluation

Beyond pixel-wise metrics, consider spatial patterns:

Visual inspection: Overlay predictions on imagery. Do predictions make spatial sense?

Boundary accuracy: How well does the model delineate boundaries between classes?

Spatial coherence: Are predictions spatially smooth or noisy?

Scale appropriateness: Does the model capture features at the right scale?

These qualitative assessments complement quantitative metrics.

Interpretability: Understanding What Your Model Learned

Interpretability is essential for building trust in AI systems and diagnosing failures.

Filter Visualization: What Patterns Do Neurons Detect?

Visualize learned convolutional filters:

Python

```python
def visualize_filters(model, layer_name='conv1'):
    """Visualize filters from a convolutional layer."""
    # Get the layer
    layer = dict(model.named_modules())[layer_name]

    # Get filter weights: (out_channels, in_channels, H, W)
    filters = layer.weight.data.cpu().numpy()

    # Visualize first 16 filters
    fig, axes = plt.subplots(4, 4, figsize=(10, 10))
    for i, ax in enumerate(axes.flat):
        if i < filters.shape[0]:
            # Average across input channels for visualization
            filter_img = filters[i].mean(axis=0)
            ax.imshow(filter_img, cmap='gray')
            ax.set_title(f'Filter {i}')
            ax.axis('off')

    plt.tight_layout()
    plt.show()
```

Early layer filters often show edge detectors, color blobs, and simple textures. Deeper layers learn more complex, task-specific patterns.

Activation Maps: What Regions Influence Predictions?

Grad-CAM (introduced in Week 2) shows which image regions influence predictions:

Python

```python
def generate_gradcam(model, image, target_class):
    """Generate Grad-CAM activation map."""
    model.eval()

    # Forward pass
    output = model(image)

    # Backward pass for target class
    model.zero_grad()
    output[0, target_class].backward()

    # Get gradients and activations from last conv layer
    gradients = model.get_gradients()  # Requires hooks
```

```
    activations = model.get_activations()

    # Weight activations by gradients
    weights = gradients.mean(dim=(2, 3), keepdim=True)
    cam = (weights * activations).sum(dim=1, keepdim=True)

    # Apply ReLU and normalize
    cam = F.relu(cam)
    cam = cam / cam.max()

    return cam
```

Overlay the activation map on the original image to see what the model focuses on.

Feature Space Visualization

Visualize how the model represents different classes in feature space:

Python

```python
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

def visualize_feature_space(model, data_loader, device):
    """Visualize learned feature representations."""
    model.eval()

    features = []
    labels = []

    with torch.no_grad():
        for images, targets in data_loader:
            images = images.to(device)

            # Get features before final classification layer
            feats = model.get_features(images)  # Requires modification
            features.append(feats.cpu().numpy())
            labels.append(targets.numpy())

    features = np.concatenate(features)
    labels = np.concatenate(labels)

    # Reduce to 2D with t-SNE
    tsne = TSNE(n_components=2, random_state=42)
    features_2d = tsne.fit_transform(features)

    # Plot
    plt.figure(figsize=(10, 8))
```

```
    for class_id in np.unique(labels):
        mask = labels == class_id
        plt.scatter(
            features_2d[mask, 0],
            features_2d[mask, 1],
            label=f'Class {class_id}',
            alpha=0.6
        )

    plt.legend()
    plt.title('Feature Space Visualization (t-SNE)')
    plt.xlabel('t-SNE 1')
    plt.ylabel('t-SNE 2')
    plt.show()
```

Well-separated clusters indicate the model has learned discriminative features. Overlapping clusters suggest confusion between classes.

Documenting Your Training Pipeline

Reproducibility is essential for scientific work and collaborative projects. Document your pipeline thoroughly.

Configuration Files

Store hyperparameters in a configuration file:

YAML

```yaml
# config.yaml
model:
  architecture: SimpleCNN
  num_bands: 10
  num_classes: 4

data:
  patch_size: 128
  train_split: 0.7
  batch_size: 32
  num_workers: 4

training:
  num_epochs: 50
  learning_rate: 0.001
  optimizer: Adam
  weight_decay: 0.01
  scheduler: ReduceLROnPlateau
```

```yaml
  early_stopping_patience: 10

paths:
  data_dir: ./data/processed
  output_dir: ./outputs
  model_save_path: ./models/best_model.pth
```

Load with:

Python

```python
import yaml

with open('config.yaml', 'r') as f:
    config = yaml.safe_load(f)
```

Experiment Logging

Track experiments systematically:

Python

```python
import json
from datetime import datetime

experiment_log = {
    'experiment_id': datetime.now().strftime('%Y%m%d_%H%M%S'),
    'config': config,
    'results': {
        'best_val_acc': best_val_acc,
        'best_val_loss': best_val_loss,
        'final_train_acc': train_accs[-1],
        'num_epochs_trained': len(train_losses),
    },
    'metrics': {
        'train_losses': train_losses,
        'train_accs': train_accs,
        'val_losses': val_losses,
        'val_accs': val_accs,
    }
}

with open(f'experiments/{experiment_log["experiment_id"]}.json', 'w') as f:
    json.dump(experiment_log, f, indent=2)
```

Model Cards

Document your trained model:

Markdown

```
# Model Card: SimpleCNN for Los Lagos Land Cover

## Model Details
- **Architecture**: SimpleCNN (3 convolutional blocks)
- **Parameters**: ~50,000
- **Training Date**: 2025-10-15
- **Framework**: PyTorch 2.1.0

## Intended Use
- **Primary Use**: Land-cover classification in Los Lagos region, Chile
- **Out-of-Scope**: Other geographic regions, high-resolution imagery

## Training Data
- **Source**: Sentinel-2 Level-2A
- **Temporal Range**: 2023-01-01 to 2023-12-31
- **Spatial Extent**: Los Lagos region (41°S - 43°S)
- **Classes**: Forest (40%), Agriculture (30%), Parcels (20%), Water (10%)
- **Training Samples**: 5,000 patches (128×128 pixels)

## Performance
- **Validation Accuracy**: 87.3%
- **Per-Class F1 Scores**:
  - Forest: 0.91
  - Agriculture: 0.88
  - Parcels: 0.79
  - Water: 0.95

## Limitations
- Trained only on summer imagery; may not generalize to other seasons
- Small parcels (<1 hectare) may be misclassified
- Performance degrades in cloudy/shadowed areas

## Ethical Considerations
- Model may be used for parcelization monitoring
- Potential privacy concerns for residential parcels
- Should not be used for enforcement without human review
```

Hands-on Activities

Activity 1: Data Preparation and Exploration (90 minutes)

Objective: Prepare a high-quality training dataset for land-cover classification.

Tasks:

1. **Define Classes** (15 min)
   - Choose 4-6 land-cover classes relevant to your case study
   - Document class definitions and characteristics
   - Consider spectral and spatial distinctiveness
2. **Collect Training Labels** (30 min)
   - Use QGIS to digitize training polygons
   - Aim for 50-100 polygons per class
   - Distribute across your AOI
   - Verify with high-resolution imagery
3. **Extract Patches** (20 min)
   - Download Sentinel-2 imagery for labeled areas
   - Extract 128×128 pixel patches
   - Create corresponding label arrays
   - Save as numpy arrays or GeoTIFFs
4. **Explore Data** (15 min)
   - Visualize example patches for each class
   - Check spectral signatures (mean reflectance per band)
   - Identify potential confusion between classes
5. **Split Data** (10 min)
   - Implement spatial train/validation split
   - Verify split is spatially separated
   - Check class balance in each split

Deliverables:

- Training polygons: `data/labels/training_polygons.geojson`
- Extracted patches: `data/processed/patches_train.npy`, `data/processed/patches_val.npy`
- Data exploration notebook: `notebooks/Week3_Data_Preparation.ipynb`
- Documentation: `reports/Week3_Data_Documentation.md`

Self-Assessment:

- Do you have sufficient examples for each class?
- Are classes spectrally distinct?
- Is your train/val split spatially separated?
- Have you documented data sources and processing steps?

Activity 2: Build and Train Your CNN (120 minutes)

Objective: Implement, train, and evaluate a CNN for land-cover classification.

Tasks:

1. **Implement Architecture** (30 min)
   - Code the SimpleCNN architecture (or design your own)
   - Verify architecture with dummy input
   - Count parameters
   - Create configuration file
2. **Setup Training** (20 min)
   - Create PyTorch datasets and data loaders
   - Define loss function and optimizer
   - Implement learning rate scheduler
   - Setup logging and checkpointing
3. **Train Model** (40 min)
   - Run training loop for 30-50 epochs
   - Monitor train/val loss curves
   - Implement early stopping
   - Save best model
4. **Evaluate Performance** (20 min)
   - Compute accuracy, precision, recall, F1
   - Generate confusion matrix
   - Calculate per-class metrics
   - Visualize predictions on validation set
5. **Document Results** (10 min)
   - Save training curves
   - Record final metrics

- Note observations and issues

Deliverables:

- Training notebook: `notebooks/Week3_CNN_Training.ipynb`
- Trained model: `models/week3_best_model.pth`
- Configuration: `config/week3_config.yaml`
- Metrics: `reports/week3_metrics.csv`
- Training curves: `figures/week3_training_curves.png`

Self-Assessment:

- Does your model train without errors?
- Are loss curves decreasing?
- Is validation accuracy reasonable (>70%)?
- Have you saved all necessary files?

Activity 3: Model Interpretation (60 minutes)

Objective: Understand what your model has learned through visualization.

Tasks:

1. **Filter Visualization** (15 min)
   - Visualize filters from first convolutional layer
   - Examine what patterns are detected
   - Compare to Week 2 ImageNet filters
2. **Activation Maps** (25 min)
   - Generate Grad-CAM for 10-15 validation examples
   - Create multi-panel figures showing:
     - Original image
     - True label
     - Predicted label
     - Activation map
   - Analyze what regions influence predictions
3. **Feature Space Analysis** (15 min)
   - Extract features from validation set

- Visualize with t-SNE or UMAP
- Examine class separability
- Identify confused classes

4. **Write Interpretation** (15 min)
    - Document observations
    - Explain what features the model uses
    - Discuss successes and failures
    - Connect to domain knowledge

Deliverables:

- Interpretation notebook: `notebooks/Week3_Model_Interpretation.ipynb`
- Filter visualizations: `figures/week3_filters.png`
- Activation maps: `figures/week3_gradcam_examples.png`
- Feature space plot: `figures/week3_feature_space.png`
- Interpretation memo: `reports/Week3_Model_Interpretation.md` (500-750 words)

Self-Assessment:

- Do activation maps make spatial sense?
- Can you explain what features the model uses?
- Have you identified patterns in model errors?
- Does your interpretation connect to remote sensing principles?

Activity 4: Ethics Thread Reflection (45 minutes)

Objective: Reflect on biases and trust in land-cover classification models.

Reading: "Algorithmic Opacity in Environmental AI" (or related article on interpretability and trust)

Reflection Prompts:

1. **Seeing Through CNNs** (150-200 words)
    - How does a CNN's "vision" differ from human vision?
    - What does your model focus on that humans might not?
    - What do humans see that your model misses?
2. **Encoded Biases** (150-200 words)
    - What biases might be encoded in your training labels?

- How might sampling strategy affect model fairness?
- Could your model perform differently for different communities?

3. **Trust and Interpretability** (150-200 words)
   - How do interpretability tools (Grad-CAM, feature viz) build trust?
   - When would you trust your model's predictions? When wouldn't you?
   - What additional validation would you need before deploying this model?

Deliverable:

- Ethics reflection: `reports/Week3_Ethics_Thread.md` (450-600 words)

Self-Assessment:

- Have you connected technical choices to ethical implications?
- Do you discuss specific examples from your model?
- Have you considered multiple perspectives?
- Is your reflection thoughtful and critical?

Week 3 Checkpoint Assessment (Self-Assessment)

Technical Implementation (35 points)

Excellent (32-35):

- Complete, well-documented training pipeline
- Model trains successfully with good performance
- Clean, reproducible code
- Proper configuration management

Good (28-31):

- Training pipeline works with minor issues
- Model achieves reasonable performance
- Code is functional with some documentation
- Basic configuration management

Satisfactory (24-27):

- Basic training pipeline implemented
- Model trains but performance is limited

- Code works but lacks documentation
- Minimal configuration management

Needs Improvement (<24):

- Incomplete or non-functional pipeline
- Model fails to train or performs poorly
- Code has significant issues
- No configuration management

Evaluation and Visualization (25 points)

Excellent (23-25):

- Comprehensive evaluation with multiple metrics
- Clear, informative visualizations
- Insightful analysis of results
- Proper spatial evaluation

Good (20-22):

- Good evaluation with standard metrics
- Adequate visualizations
- Solid analysis
- Some spatial considerations

Satisfactory (17-19):

- Basic evaluation (accuracy, confusion matrix)
- Simple visualizations
- Limited analysis
- Minimal spatial evaluation

Needs Improvement (<17):

- Incomplete evaluation
- Missing or unclear visualizations
- Superficial analysis
- No spatial evaluation

Interpretation Depth (25 points)

Excellent (23-25):

- Thorough interpretation using multiple techniques
- Insightful connections to domain knowledge
- Clear explanations of model behavior
- Identifies both strengths and limitations

Good (20-22):

- Good interpretation with standard techniques
- Some connections to domain knowledge
- Explains model behavior adequately
- Discusses some limitations

Satisfactory (17-19):

- Basic interpretation
- Limited domain connections
- Surface-level explanations
- Minimal discussion of limitations

Needs Improvement (<17):

- Superficial or missing interpretation
- No domain connections
- Unclear explanations
- No discussion of limitations

Ethics Reflection (15 points)

Excellent (14-15):

- Thoughtful, critical reflection
- Concrete connections to your model
- Multiple perspectives considered
- Actionable insights

Good (12-13):

- Solid reflection
- Good connections to your work
- Some perspectives considered

- Some insights

Satisfactory (10-11):

- Basic reflection
- Limited connections
- Single perspective
- General observations

Needs Improvement (<10):

- Superficial reflection
- No connections to your work
- No critical thinking
- Vague statements

Reflection and Discussion (30 minutes)

Key Questions for Reflection

1. **Technical Mastery**: What was the most challenging aspect of building and training your CNN? How did you overcome it?
2. **Data Quality**: How did data quality and quantity affect your results? What would you do differently in data collection?
3. **Model Behavior**: What surprised you about what your model learned? Did it focus on features you expected?
4. **Performance Gaps**: Where does your model perform well vs. poorly? Can you explain these patterns?
5. **Interpretability Value**: How did visualization techniques help you understand your model? What insights did they provide?
6. **Ethical Awareness**: What ethical considerations arose during model development? How might your model impact different stakeholders?

Self-Reflection Prompt

Write 300-400 words in `reports/Week3_Reflection.md` addressing:

- **Most significant challenge**: What was hardest this week, and what did you learn from it?

- **Model insights**: What did you learn about how CNNs see landscapes?
- **Data lessons**: What did you learn about the importance of data quality and preparation?
- **Looking forward**: How will you build on this foundation in Week 4?

Preview of Week 4: Transfer Learning and Multi-Source Fusion

Next week, you'll enhance your CNN through transfer learning and multi-source data fusion:

What's Coming

Transfer Learning: Learn to leverage pre-trained models (ResNet, EfficientNet) to improve performance with limited training data. Understand when to freeze layers vs. fine-tune.

Multi-Source Fusion: Combine Sentinel-2 with other data sources (MODIS, Landsat, DEM) to create richer representations. Learn strategies for aligning multi-resolution data.

Domain Adaptation: Explore techniques for adapting models trained in one region to another, addressing geographic generalization.

Baseline Pipeline: Establish a documented, reproducible pipeline that serves as the foundation for your capstone project.

Preparation for Week 4

1. Review transfer learning concepts (30 min)

- What is transfer learning and why does it work?
- Feature extraction vs. fine-tuning
- Resources: Transfer Learning Tutorial

2. Explore pre-trained models (20 min)

- Browse TorchVision model zoo
- Understand different architectures (ResNet, EfficientNet, DenseNet)
- Consider which might work for your application

3. Research multi-source data (30 min)

- What other data sources are available for your region?
- MODIS (250m-1km resolution, daily)
- Landsat (30m resolution, 16-day)
- DEM (elevation data)

- Climate data
- Consider how these might complement Sentinel-2

4. Reflect on your baseline (15 min)

- What worked well in your Week 3 model?
- What limitations need addressing?
- How might transfer learning or data fusion help?

Additional Resources

Deep Learning Textbooks and Courses

Books:

- **Goodfellow, I., Bengio, Y., & Courville, A. (2016).** *Deep Learning*. MIT Press.
  - Chapter 9: Convolutional Networks (required reading)
  - https://www.deeplearningbook.org/
- **Zhang, A., et al. (2023).** *Dive into Deep Learning*.
  - Interactive book with code examples
  - https://d2l.ai/

Online Courses:

- **Fast.ai Practical Deep Learning**: https://course.fast.ai/
  - Practical, code-first approach
- **Stanford CS231n**: http://cs231n.stanford.edu/
  - Comprehensive computer vision course

PyTorch Resources

Official Tutorials:

- **Training a Classifier**: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- **Visualizing Models**: https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html

Libraries:

- **TorchGeo**: Geospatial datasets and models
  - https://torchgeo.readthedocs.io/

- **Segmentation Models PyTorch**: Pre-built architectures
  - https://github.com/qubvel/segmentation_models.pytorch

Remote Sensing with Deep Learning

Papers:

- **Oquab, M., et al. (2014).** Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks. *CVPR*.
  - https://arxiv.org/abs/1411.1792
- **Long, J., Shelhamer, E., & Darrell, T. (2015).** Fully Convolutional Networks for Semantic Segmentation. *CVPR*.
  - Foundation for semantic segmentation
  - https://arxiv.org/abs/1411.4038
- **Helber, P., et al. (2019).** EuroSAT: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification. *IEEE JSTARS*.
  - Benchmark dataset for land-cover classification
  - https://arxiv.org/abs/1709.00029

Interpretability and Visualization

Papers:

- **Selvaraju, R. R., et al. (2017).** Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. *ICCV*.
  - https://arxiv.org/abs/1610.02391
- **Olah, C., et al. (2018).** The Building Blocks of Interpretability. *Distill*.
  - https://distill.pub/2018/building-blocks/

Tools:

- **Captum**: PyTorch interpretability library
  - https://captum.ai/
- **Grad-CAM++**: Improved Grad-CAM
  - https://github.com/jacobgil/pytorch-grad-cam

Glossary Terms Introduced This Week

- **Convolution**: Mathematical operation combining an image with a kernel to detect spatial patterns
- **Kernel (Filter)**: Small matrix of learnable parameters that slides across images to detect features
- **Stride**: Step size for moving kernels across images during convolution
- **Padding**: Adding border pixels to control output dimensions after convolution
- **Pooling**: Downsampling operation reducing spatial dimensions (max pooling, average pooling)
- **Batch Normalization**: Technique standardizing layer inputs to stabilize training
- **Dropout**: Regularization technique randomly deactivating neurons during training
- **ReLU (Rectified Linear Unit)**: Activation function outputting max(0, x)
- **Global Average Pooling**: Averaging across entire spatial dimensions, reducing to one value per channel
- **Loss Function**: Quantifies prediction errors; guides parameter updates during training
- **Cross-Entropy Loss**: Standard loss function for classification tasks
- **Optimizer**: Algorithm updating parameters to minimize loss (SGD, Adam, AdamW)
- **Learning Rate**: Step size for parameter updates; controls training speed
- **Learning Rate Schedule**: Strategy for adjusting learning rate during training
- **Epoch**: One complete pass through the training dataset
- **Batch**: Subset of training data processed together
- **Overfitting**: Model memorizes training data but fails to generalize to new data
- **Underfitting**: Model fails to learn patterns in training data
- **Early Stopping**: Halting training when validation performance stops improving
- **Confusion Matrix**: Table showing predicted vs. actual classes for all examples
- **Precision**: Fraction of positive predictions that are actually positive
- **Recall (Sensitivity)**: Fraction of actual positives that are correctly predicted
- **F1 Score**: Harmonic mean of precision and recall
- **IoU (Intersection over Union)**: Metric measuring spatial overlap between predicted and actual regions
- **Semantic Segmentation**: Pixel-wise classification assigning each pixel to a class
- **Feature Space**: High-dimensional representation where each dimension corresponds to a learned feature

- **t-SNE (t-Distributed Stochastic Neighbor Embedding)**: Dimensionality reduction technique for visualization
- **Spatial Autocorrelation**: Statistical property where nearby locations have similar values
- **Spatial Train/Test Split**: Dividing data geographically rather than randomly to prevent information leakage

---

Notes for Self-Paced Learners

Time Management Suggestions

Week 3 is technically demanding, requiring hands-on coding and experimentation. Budget approximately **12-15 hours**:

Day 1-2 (4-5 hours):

- Read core content on CNN architectures and training
- Review PyTorch tutorials
- Understand convolution operations and training procedures

Day 3 (3-4 hours):

- Activity 1: Data preparation
- Collect and process training data
- Implement spatial train/test split

Day 4-5 (4-5 hours):

- Activity 2: Build and train CNN
- Implement architecture
- Run training loop
- Monitor and evaluate

Day 6 (2-3 hours):

- Activity 3: Model interpretation
- Generate visualizations
- Write interpretation memo

Day 7 (1-2 hours):

- Activity 4: Ethics reflection
- Write weekly reflection

- Organize repository

Common Challenges and Solutions

Challenge: Model won't train (loss doesn't decrease)

Solutions:

- Check data normalization—improper normalization is a common cause
- Verify labels are correct (0-indexed, matching number of classes)
- Try lower learning rate (0.0001 instead of 0.001)
- Check for NaN values in data
- Simplify model architecture temporarily to isolate the issue

Challenge: Out of memory errors

Solutions:

- Reduce batch size (try 16, 8, or even 4)
- Reduce patch size (try 64×64 instead of 128×128)
- Use fewer convolutional channels
- Enable gradient checkpointing
- Use mixed precision training (torch.cuda.amp)

Challenge: Overfitting (train accuracy high, validation low)

Solutions:

- Collect more training data
- Increase dropout rate (try 0.5 or 0.6)
- Add weight decay to optimizer
- Use data augmentation (flips, rotations)
- Simplify model (fewer layers or channels)
- Stop training earlier

Challenge: Poor performance on certain classes

Solutions:

- Check class balance—collect more examples of underperforming classes
- Use weighted loss function to emphasize rare classes
- Examine confusion matrix—which classes are confused?
- Visualize misclassified examples—what patterns do you see?

- Consider if classes are spectrally distinct enough

Challenge: Training is very slow

Solutions:

- Use GPU if available (check with `torch.cuda.is_available()`)
- Increase batch size (if memory allows)
- Reduce number of workers in DataLoader if CPU-bound
- Use smaller patches or fewer training examples for initial experiments
- Profile code to find bottlenecks

Challenge: Results aren't reproducible

Solutions:

- Set random seeds:
- Use deterministic algorithms (may be slower):
- Document all hyperparameters in configuration files
- Save exact package versions (`pip freeze > requirements.txt`)

Extension Activities

For Advanced Learners:

- **Architecture experiments**: Try different architectures (U-Net, DeepLabV3, FCN). Compare performance.
- **Data augmentation**: Implement augmentation (rotations, flips, brightness adjustments). Measure impact.
- **Multi-task learning**: Train model to predict both land cover and NDVI simultaneously.
- **Attention mechanisms**: Add attention layers to your CNN. Does it improve performance?

For Conceptual Thinkers:

- **Feature hierarchy analysis**: Visualize features at multiple layers. How does representation change?
- **Comparison with traditional methods**: Compare your CNN to Random Forest on same data. What are trade-offs?
- **Error analysis**: Deeply analyze model errors. What patterns emerge? What do they reveal about the task?

- **Literature review**: Read recent papers on land-cover classification. What techniques are state-of-the-art?

For Visual Learners:

- **Architecture diagram**: Draw detailed diagram of your CNN architecture with dimensions at each layer.
- **Training animation**: Create animated visualization of training progress (loss curves, example predictions over epochs).
- **Feature evolution**: Visualize how learned features change during training.
- **Comparison gallery**: Create visual comparison of predictions vs. ground truth for validation set.

For Hands-On Learners:

- **Hyperparameter tuning**: Systematically vary hyperparameters (learning rate, batch size, architecture). Document results.
- **Cross-validation**: Implement k-fold spatial cross-validation. How stable is performance?
- **Ensemble methods**: Train multiple models and combine predictions. Does it improve accuracy?
- **Real-time inference**: Implement inference pipeline for new imagery. Measure speed.

This comprehensive Week 3 study guide provides the technical depth, practical guidance, and conceptual framework you need to successfully build, train, and interpret your first CNN for Earth observation. Take your time with each section, experiment thoroughly, and don't hesitate to revisit concepts as needed.

Document Version: 1.0

Last Updated: October 15, 2025

Author: Manus AI

Course: GeoAI and Earth Vision: Foundations to Frontier Applications