Reading: Assignment Overview: Agricultural Land Classifier

Estimated time: 6 minutes

Learning objectives:

- Build and compare binary image classification models using Keras and PyTorch to evaluate agricultural land use from satellite imagery.
- Analyze and optimize model performance using precision, recall, F1-score, and ROC-AUC to improve real-world geospatial AI applications for fertilizer sales forecasting.

Introduction

This reading helps you explore how you to build a CNN-based model for classifying satellite images as agricultural or non-agricultural, enabling fertilizer sales forecasting. You'll learn about models in Keras and PyTorch, compare workflows, tune hyperparameters, and evaluate performance using precision. You will also recall F1-score and ROC-AUC-building real-world skills in geospatial AI and binary image classification.

Overview

As a data scientist at a fertilizer company which is preparing to expand sales into an unfamiliar region, you are tasked with createing a projection of fertilizer sale in the new territory. Management's first question is simple: "What share of this territory is actually farmland?". To answer this, you plan to build an AI model that scans hundreds of square kilometres of satellite image data and labels each agricultural or non-agricultural tile. To train this model, you have been provided with a labeled dataset with labels stored in two folders "class_1_agri" (3,000 images) and "class_0_non_agri" (3,000 images). You also plan to train two independent Convolutional Neural Networks (CNNs): one written in Keras and the other in PyTorch, to:

1. Validate the models
2. To understand their distinct workflows, giving you wider career flexibility.

Why CNNs are a good choice for satellite image classification

Satellite pixels contain subtle textural cues such as furrow patterns, uniform crop canopies, and irrigation circles that are hard to code in regular ML rules. CNNs automatically discover and layer these features, beginning with edge detectors in the first layer, moving to texture patterns in the middle layers, and finishing with high-level concepts such as "patchwork field" or "urban block." Their hierarchical learning is very useful for these type of classification tasks.

CNN architecture for binary classification

The architecture depth and width in a computer-vision CNN model is usually dependent of the complexity of visual features of the images. If the within-class variance is high, the backbone needs several convolutional blocks and bigger feature maps to encode for this variance. You can start with a small backbone, with 4-6 convolutional layers. If the training and validation losses diverge, signifying underfitting, you can widen the filters or add more residual blocks. If they converge quickly and plateau, you can keep the depth lower for faster inference.

Hyperparameters

Hyperparameters are model configuration which are chosen before training a model. They shape either the network's structure or the optimization process.

Common architecture hyperparameters include the number of hidden layers, filters, or fine-tuned blocks in a CNN. Increasing the filters in early layers can help if classes have diverse appearances. Increasing filters also doubles GPU memory, so they need to be expanded carefully.

Optimization hyperparameters

Optimization hyperparameters are used in learning. The learning-rate controls weight update size, momentum smooths gradient directions, and learning-rate decay schedules gradual reductions for stable convergence. Usually, a learning rate of 0.001 with Adam optimizer is a good starting point. Too high learning rate can cause divergence; too low can slow the convergence. A dynamic learning rate, such as step decay after 5-7 epochs, tends to produce smoother loss curves.

Regularization hyperparameters

Regularization hyperparameters, such as dropout rate or L2 weight decay, help in controlling over-fitting in a neural network. They modify the loss, the data, or the training loop so the model learns simpler, more general patterns by randomly disabling neurons or penalizing large weights.

Batch size

The batch size affects gradient noise and GPU memory usage, 32 is a safe default. Larger batches make better use of GPU throughput but may reduce generalization.

Epoch count

Epoch count decides how many times the model sees the full dataset. With low number of images, such as 6000 in our case, 15-20 epochs are usually enough. Use early stopping on validation loss to avoid needless training once the curve plateaus.

Building the model in Keras

Keras's Sequential API lets you create a full image-classification pipeline with minimal code. To create a CNN model in Keras, you use the following steps:

1. **Define a sequential model**: This includes stacking layers such as Rescaling, Conv2D, MaxPooling2D, Flatten, and Dense; input shape is declared only on the first layer.
2. **Call compile**: This includes choosing optimizer (for example, Adam), loss (binary_crossentropy or sparse_categorical_crossentropy).
3. **Feed data**: Feed the relevant data through a tf.data.Dataset that performs lazy loading and on-the-fly augmentation, then launch training with a single fit, optionally adding callbacks for early stopping and checkpointing.

4. **Exporting the model**: Finally, export the trained model with save for deployment.

Building a CNN in PyTorch

1. **Define a network class** that inherits from nn.Module; declare convolution, pooling, and dense layers in __init__, then wire them in a forward method to specify how tensors flow through the model.
2. **Construct a Dataset** (e.g., ImageFolder or custom) with torchvision.transforms for resizing, normalization, and augmentation, and wrap it in a DataLoader that batches, shuffles, and prefetches data via multiple CPU workers.
3. **Instantiate an optimizer** such as Adam or SGD and a criterion like nn.CrossEntropyLoss; optional components include a learning-rate scheduler and automatic-mixed-precision (torch.cuda.amp).
4. **Write an explicit training loop** for each epoch, iterate over the DataLoader, move inputs and labels to GPU, zero gradients, run the forward pass, compute loss, call backward, update weights with optimizer.step, log metrics, and periodically save checkpoints.

Evaluating the classifier

You can compare the performance of a classifier or between different classifiers using various performance metrics. Some of these metrics are:

Precision: The fraction of predicted farmland tiles that are truly farmland. High precision avoids over-estimating potential customers.

Recall or sensitivity: The fraction of true farmland tiles that your model catches. High recall prevents lost sales opportunities.

F1-Score: The harmonic mean of precision and recall; a balanced single number.

Confusion matrix: The visual check for systematic errors-e.g., confusing dry stubble with urban rooftops.

ROC-AUC: Shows how well the model separates the two classes across all thresholds; useful when the optimal probability cut-off is a business decision.

Summary

In this reading, you learned that as a geospatial AI specialist, you can choose between a CNN implementation using either Keras or PyTorch.

- Keras lets you prototype a working classifier quickly, while PyTorch can be useful for controlling fine details in the model.
- Both frameworks handle sequential data loading and on-the-fly augmentation and are crucial for large datasets, such as satellite imagery volumes.
- Both frameworks supply utilities to track precision, recall, F1, and ROC-AUC, translating raw pixel predictions into the business language of acreage and market-share potential.
- Using the performance metrics for finetuning the hyperparameters of the model can help you create a model for accurate land usage classification.