Applied Artificial Intelligence and Deep Learning Book

Tim Mayer   Biplov Bhandari   David Saah

2026-06-01

Table of contents

Introduction 7

Editors' Introduction 9

Background and Motivation 9

What This Book Is—and What It Is Not 9

Audience 10

How to Use This Book 10

Meetings, Workshops, and Community Events 10

Data Science and AI/ML Exchange (2019) 10

TensorFlow Technical Exchange (2020) 11

TensorFlow Exchange at Geo for Good (2022) 11

Data Science and AI/ML Exchange (2023) 12

Earth Observation Foundational Model Hackathon 12

Technical Work Planning: Geo-AI for Earth Science (2025) 12

American Geophysical Union Sessions 13

Acknowledgements 13

Preface 14

Foreword 14

About the Editors 15

1 Data Preparation 16

I Semantic Segmentation 17

2 Crop Mapping — Rice Mapping in Bhutan with U-Net 18

2.1 Note: This notebook is meant to be run in Colab. You can still run this locally, make sure you have Google Drive API installed and path adjusted in relevant places. 18

2.2 Setup environment 18

2.2.1 Download datasets 18

2.2.2 Setup config file variables 19

2.2.3 Update the config file programtically 20

2.3 U-Net Model 21

2.3.1 Load config file variables 21

2.3.2 Load `ModelTrainer` class 22

2.3.3 Train and Save U-Net model . . . 22
2.3.4 Save the config file . . . 22
2.3.5 Load the logs files via TensorBoard . . . 23
2.3.6 Load the Saved U-Net Model . . . 23
2.3.7 Inference using Saved U-Net Model . . . 23
2.4 DNN Model . . . 24
2.4.1 Setup any changes in the config file for DNN Model . . . 24
2.4.2 Update the config file programtically . . . 24
2.4.3 Load config file variables for DNN Model . . . 25
2.4.4 Load `ModelTrainer` class . . . 25
2.4.5 Train and Save DNN model . . . 25
2.4.6 Save the config file . . . 25
2.4.7 Load the logs files via TensorBoard . . . 25
2.4.8 Load the Saved DNN Model . . . 26
2.4.9 Inference using Saved DNN Model . . . 26
2.5 Independent Validation . . . 26
2.5.1 Update the config file . . . 26
2.5.2 Load config file variable . . . 27
2.5.3 Import earthengine and geemap for visualization . . . 27
2.5.4 Class Information and Masking . . . 27
2.5.5 Model: U-Net . . . 28
2.5.6 Model: DNN . . . 29
2.6 Figures and Plots . . . 31
2.6.1 Training and Validation Plot . . . 31
2.6.2 Probability Distribution Plot . . . 33

3 Selective Logging Detection with Deep Learning and Very-High Resolution Imagery 35
3.1 Part 2: Modeling . . . 35
3.2 1. Data collection and initial settings . . . 35
3.3 2. Data visualization . . . 38
3.4 3. Data processing . . . 40
3.4.1 3.1 Definition of variables . . . 40
3.4.2 3.2 Create a TFRecordDataset . . . 41
3.5 4. Deep Learning . . . 42
3.5.1 4.1 Model construction . . . 42
3.5.2 4.2 Model training . . . 45
3.5.3 4.3 Model evaluation . . . 46
3.5.4 4.4 Save the model . . . 47

4 Object Detection 48

II Time Series **49**

5 Phenology-Guided Deep Learning with Uncertainty Quantification for Soybean Yield Prediction **50**
5.1 Overview . . . 50
5.2 The Challenge with Current Yield Prediction Models . . . 50
5.3 Model Architecture and Approaches . . . 51
5.3.1 1. Probabilistic LSTM Model . . . 51
5.3.2 2. Probabilistic 1D-CNN Model . . . 51
5.3.3 3. Gaussian Process Regression (GPR) . . . 51
5.4 Key Functionalities . . . 51
5.4.1 Data Preprocessing . . . 51
5.4.2 Model Training and Evaluation . . . 52
5.4.3 Uncertainty Quantification . . . 52
5.4.4 Feature and Temporal Importance Analysis . . . 52
5.4.5 Spatial-Temporal Analysis . . . 52
5.5 1. Data Loading and Preprocessing . . . 54
5.5.1 Study Area . . . 54
5.5.2 Dataset Overview . . . 55
5.5.3 Data Structure . . . 55
5.5.4 Visualizing Input Feature Dynamics . . . 59
5.6 Why Phenology-Guided Time Series Modeling? . . . 62
5.7 Why Quantify Prediction Uncertainty? . . . 63
5.8 2. Model Development and Training . . . 63
5.8.1 Key Implementation Features: . . . 63
5.8.2 2.1 Probabilistic Long Short-Term Memory (LSTM) Model . . . 63
5.8.3 2.2 Probabilistic 1D Convolutional Neural Network (CNN) Model . . . 73
5.8.4 2.3 Gaussian Process Regression (GPR) Model . . . 77
5.9 3. Advanced Analysis and Interpretation . . . 80
5.9.1 3.1 Multi-Year Performance Analysis . . . 80
5.9.2 3.2 Feature and Temporal Importance . . . 80
5.9.3 3.3 Spatial-Temporal Visualization . . . 80
5.9.4 3.1 Multi-Year Model Performance Analysis . . . 80
5.9.5 3.2 Feature Importance Analysis . . . 84
5.9.6 3.3 Phenological (Temporal) Importance Analysis . . . 89
5.9.7 3.4 Spatial-Temporal Visualization and Uncertainty Mapping . . . 93
5.10 Conclusions and Limitations . . . 100
5.11 Looking Forward: Future Directions and Next Steps . . . 101

6 Understanding active fire detection uncertainty with Bayesian Neural Networks **102**
6.1 Important Notes (Read Before Running) . . . 102
6.1.1 Python Versioning . . . 102
6.1.2 Code Order . . . 102

6.1.3 Colab vs Jupyter . . . 103
6.1.4 How to Run This Notebook on Colab - Default setting . . . 103
6.1.5 How to Run This Notebook on Local . . . 104
6.2 Frequently Asked Questions . . . 105
6.2.1 Running this Notebook without Training Models . . . 106
6.3 Colab Specific Installations . . . 107
6.3.1 Import Statements . . . 108
6.4 Case Study A . . . 109
6.4.1 Define epochs . . . 109
6.4.2 Model One . . . 109
6.4.3 Model Two . . . 117
6.4.4 Model Three . . . 120
6.4.5 Define epochs . . . 120
6.4.6 Results (All Models) . . . 129
6.5 Case Study B . . . 134
6.5.1 Model One . . . 134
6.5.2 Model Two . . . 138
6.5.3 Results (All Models) . . . 141
6.5.4 Hypothesis review . . . 145

7 Transfer Learning **147**

8 Fusion **148**

9 Downscaling **149**

III Future of Deep Learning and Foundational Models **150**

10 Evaluating Foundation Models Trained with Earth Observation Data **151**
10.1 Introduction . . . 151
10.1.1 How foundation models improve efficiency in machine learning workflows 153
10.1.2 Limitations . . . 156
10.1.3 Challenges . . . 156
10.2 Run Prithvi . . . 158
10.2.1 Define Location and Date of Interest . . . 159
10.2.2 Natural Disaster in Focus . . . 160
10.2.3 Retrieve Data from the STAC Catalog . . . 160
10.2.4 Create a Bounding Box for the Area of Interest . . . 161
10.2.5 Retrieve the imagery data . . . 162
10.2.6 Review the Downloaded Imagery . . . 163
10.2.7 Load the Model . . . 163
10.2.8 Format Band Pixel Data for the Model . . . 164

10.2.9 Execute the Model . . . 165
10.2.10 Analyze the Embeddings . . . 166
10.2.11 Embeddings for Each Time Step . . . 167
10.2.12 Train a Model using the Embeddings as Features . . . 170
10.3 Clay . . . 171
10.3.1 Load the Model . . . 172
10.3.2 Prepare Band Metadata for Model Input . . . 173
10.3.3 Convert Band Pixel Data to Model Format . . . 174
10.3.4 Combine Metadata and Pixels . . . 175
10.3.5 Execute the Model . . . 175
10.3.6 Analyze the Embeddings . . . 176
10.3.7 Train a Model using the Embeddings as Features . . . 177
10.3.8 Change Detection Heatmap (Pre- vs. Post-Flood Distances) . . . 178
10.3.9 Discussion . . . 179
10.3.10 Conclusion . . . 183

11 Ethics and Artificial Intelligence **184**

Conclusions **185**

References **186**

Introduction

First edition. Published electronically June 1st 2026 DOI: XXXXXXX

This publication may be reproduced in whole or in part and in any form for educational or non-profit purposes without special permission from the copyright holder, provided acknowledgement of the source is made. No use of this publication may be made for resale or for any other commercial purpose.

DISCLAIMER: The views expressed in this publication are not necessarily those of the agencies cooperating in this project. Mention of a commercial company or product in this report does not imply endorsement by NASA. The use of information from this publication concerning proprietary products for advertising or publicity is not permitted. Trademark names and symbols are used in an editorial fashion with no intention of infringement on trademark or copyright laws.

Figures and maps in this handbook contain material from: Google Maps data: Imagery © 2018 Landsat/Copernicus, DigitalGlobe, Mapdata © 2018 Google. Sentinel-1 data: Earth Big Data, LLC 2018, contains modified Copernicus Sentinel data 2014–2018, processed by ESA. ALOS data: Earth Big Data, LLC 2018; includes Material © JAXA/METI 2007–2018

EarthRISE Program Office National Space Science and Technology Center 320 Sparkman Drive, Huntsville, AL 35805

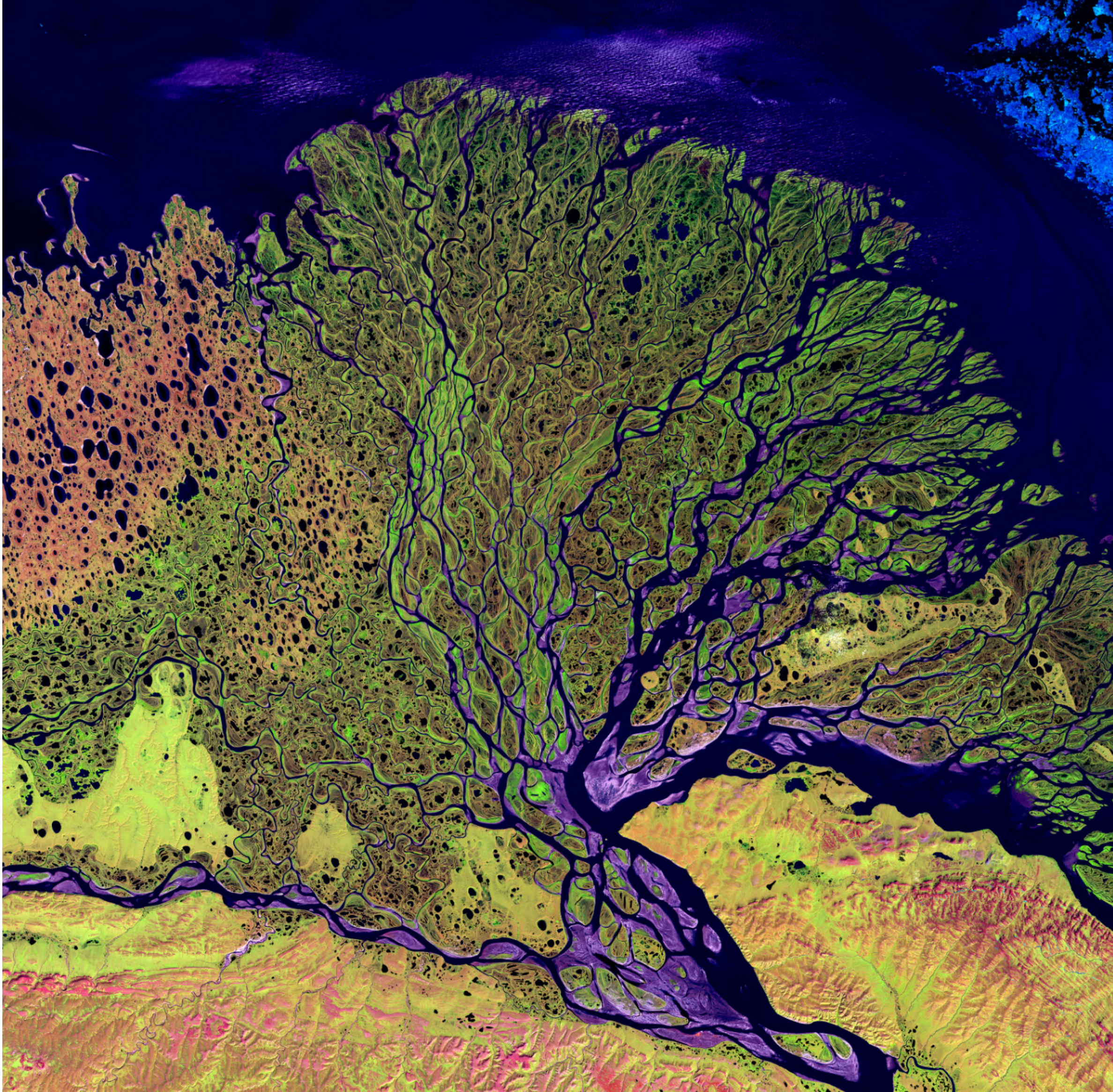


Figure 1: The Lena River, some 2,800 miles (4,400 km) long, is one of the largest rivers in the world. The Lena Delta Reserve is the most extensive protected wilderness area in Russia. It is an important refuge and breeding ground for many species of Siberian wildlife. This image was acquired by Landsat 7's Enhanced Thematic Mapper plus (ETM+) sensor on July 27, 2000. This is a false-color composite image made using shortwave infrared, infrared, and red wavelengths. Image provided by the USGS EROS Data Center Satellite Systems Branch. This image is part of the Landsat Earth as Art series.

Editors' Introduction

Background and Motivation

The intersection of deep learning, foundational models, and geospatial science is reshaping how we observe, interpret, and manage our planet. Remote sensing analysts, Earth scientists, environmental specialists, and GIS professionals increasingly encounter machine learning tools in their workflows—yet for many, the path into this rapidly evolving landscape remains a challenge. Comprehensive textbooks struggle to keep pace with the field's velocity, while research literature often presupposes a depth of AI expertise that most applied Earth scientists have not yet developed. This book was created to bridge that gap.

The *EarthRISE Applied Artificial Intelligence and Deep Learning Book* is the product of more than seven years of community-driven knowledge sharing within the Geospatial-Artificial Intelligence Working Group (Geo-AI WG). Rooted in three defining pillars: Building Capacity, Application Development, and Knowledge Sharing this volume represents the third pillar in action. A practitioner's guide distilled from real-world experience, peer collaboration, and a deep commitment to making advanced geospatial AI accessible to the broader Earth observation community. This book stands in a lineage of open, community-authored knowledge resources –alongside the SAR Handbook and the Google Earth Engine Book—sharing the same mission: to translate technical expertise into resources practitioners can immediately apply.

What This Book Is—and What It Is Not

This is not an encyclopedic survey of artificial intelligence. The field evolves too quickly for any single volume to claim comprehensiveness, and we make no such claim. Instead, this book presents a curated collection of applied deep learning (DL) use cases ranging from common analytical needs to broader considerations for practitioners, such as the ethics of artificial intelligence.

Think of this book as a "choose your own adventure." Readers with foundational knowledge can move directly to applied chapters aligned with their thematic interests—whether agriculture, forestry, water resources, urban environments, or disaster response. Each chapter pairs conceptual background with executable Jupyter notebooks designed to run across multiple platforms and frameworks, including TensorFlow, PyTorch, Google Cloud Platform, and Amazon Web Services. Our goal is to meet practitioners where they are and lower the barrier to experimentation.

This first collection of chapters is designed with the future in mind. As the field continues to evolve, the book remains open to future submissions and collaborations as a living resource that grows alongside the community it serves.

Audience

We assume readers arrive with meaningful geospatial expertise and familiarity with satellite imagery, raster and vector data, and common Earth observation workflows, but limited formal background in DL or AI. Our aim is to grow awareness, build practical skill, and empower readers to evaluate where and when DL approaches can meaningfully enhance their existing methods.

How to Use This Book

Readers may progress sequentially from foundational data preparation through increasingly advanced problem sets, or navigate directly to chapters most relevant to their domain. Applied examples are cross-cutting by design: a workflow developed for crop-type mapping may illuminate approaches applicable to flood mapping, urban land cover classification, or ecosystem monitoring. We encourage readers to run, modify, and share the accompanying code. All notebooks are provided under open licenses.

Meetings, Workshops, and Community Events

The chapters and methods presented in this book did not emerge in isolation. They are the product of years of structured exchanges, hands-on workshops, international collaborations, and community-driven hackathons that brought together practitioners, scientists, and thought leaders from across the scientific community. The events below trace the arc of that community from early capacity-building exchanges to the strategic planning now shaping the next generation of Geo-AI applications.

Data Science and AI/ML Exchange (2019)

Google Campus, Mountain View, CA, USA · Geo for Good 2019

The 2019 Data Science Exchange marked the beginning of a sustained, network-wide effort to build applied machine learning (ML) capacity within the Geo-AI Working Group community. Held in tandem with the Google Geo for Good Summit, the exchange brought together scientists

and practitioners to explore the use of data science, ML, and DL for Earth science applications. Participants worked hands-on with real-world problem sets, bringing label data from their own service areas to begin developing ML workflows using Google Earth Engine.

This event laid the organizational foundation for what would become the Geo-AI Working Group, establishing a shared vocabulary, collaborative norms, and a forward-looking agenda for applied AI within the network. We are grateful to Nick Clinton and Karin Tuxen-Bettman at Google for their support, as well as to partners from the University of San Francisco, the International Centre for Integrated Mountain Development (ICIMOD), the Asian Disaster Preparedness Centre (ADPC), Regional Centre for Mapping of Resources for Development (RCMRD), International Center for Tropical Agriculture (CIAT), and AGRHYMET.

TensorFlow Technical Exchange (2020)

Virtual · 21–25 September 2020

Building directly on the 2019 exchange, the 2020 TensorFlow Technical Exchange extended the network's capacity-building efforts into a fully virtual format during the COVID pandemic. Over five days, participants from partner organizations engaged in training sessions, lectures, and hands-on demonstrations led by experts in AI, DL and ML for Earth sciences.

The exchange focused on applying TensorFlow within the Google Earth Engine framework, with each team working toward a defined ML workflow for an identified service application. Participants also explored emerging methodologies and libraries aligned with the Geo for Good 2020 summit. The event continued efforts initiated by the TensorFlow working group to provide network-wide expertise and support, and contributed to virtual programming at the Geo for Good 2021 summit.

TensorFlow Exchange at Geo for Good (2022)

Google Bay View Event Center, Mountain View, CA · October 4–6, 2022

The 2022 exchange returned to an in-person format at Google's Bay View campus, deepening the technical work of previous years and formalizing the establishment of a Geo-AI Working Group. Participants focused on refining end-to-end ML workflows for solution/service co-development, with an emphasis on deploying trained models within Google Earth Engine and other open source platforms.

Data Science and AI/ML Exchange (2023)

Google Campus, Mountain View, CA · Geo for Good 2023

The 2023 exchange expanded scope to encompass the rapidly evolving landscape of foundational models, large language models, and integrated Geo-AI workflows. In addition to continued capacity building in DL methods, participants worked in a build-a-thon format to develop and refine ML workflows for specific service applications, culminating in presentations at a dedicated session during the Geo for Good 2023 summit.

Earth Observation Foundational Model Hackathon

Geo for Good Summit, Dublin · 2023

Earth Observation Foundational Model (EOFM) and Large Earth Model (LEM) Integration into Google Earth Engine

This hackathon challenged participants to explore how Earth observation foundational models could be effectively leveraged as part of operational geospatial workflows spanning Google Earth Engine and Google Cloud Platform. Working in collaborative teams, hackers hosted, deployed, and ran inference from models including NASA IMPACT and IBM's Prithvi and Meta's Segment Anything Model (SAM), sourced from Hugging Face. Two integration pathways were explored: batch prediction via data export from GEE with a custom I/O pipeline, and online prediction directly from Earth Engine using the EE–Vertex AI connector. The hackathon was a defining moment for the community's understanding of how foundational models can complement traditional Earth observation workflows.

Technical Work Planning: Geo-AI for Earth Science (2025)

National Space Science and Technology Center, Huntsville, AL · 3–7 February 2025

The February 2025 work planning meeting represented a strategic inflection point for the Geo-AI Working Group and Data Science area of NASA EarthRISE. Convened at the National Space Science and Technology Center, the meeting brought together organizational leads to formalize a shared strategic vision for Geo-AI and conduct roadmapping toward alignment with the Earth Action strategic framework.

American Geophysical Union Sessions

Members of the Geo-AI Working Group have convened and chaired dedicated sessions at the American Geophysical Union annual meeting since 2022, building a scientific forum for presenting applied results and advancing community dialogue on ML for Earth science. The sessions have grown in scope each year, reflecting the rapid evolution of the field itself.

In 2022 in Chicago, the group organized and chaired a full suite of sessions under the title *Addressing Environmental Challenges and Sustainable Development Through Earth Science Applications Utilizing Machine Learning IV*, comprising two oral sessions, a poster session, and a virtual poster session. The call for that year centered on novel applications of ML, DL, and AI in data-sparse regions, with an emphasis on open data access, cloud computing, capacity building, and data democracy—themes that run throughout this book.

By 2023 in San Francisco, the session framing had broadened to foreground the rise of Geospatial AI and its intersection with geoscience, with a specific focus on innovative technical solutions addressing pressing environmental issues while advancing FAIR data principles: Findability, Accessibility, Interoperability, and Reuse. The group presented *Applications of Machine Learning and Deep Learning to Address Environmental Challenges and Sustainable Development Goals*, reflecting the community's expanding scope beyond DL fundamentals toward integrated, deployment-ready workflows.

The 2024 session in Washington, D.C., *Bridging the Gap: Leveraging Geo-Artificial Intelligence to Address Environmental Challenges*, acknowledged the moment the field was entering: one shaped by foundational models, large language models, and a new generation of Earth observation AI tools. The session description captured the stakes clearly—groundbreaking efforts in foundational model development, near real-time deforestation alerts, precision agriculture, and air quality monitoring—and called for collaboration between experts, non-profits, and applied scientists to ensure these capabilities reach the regions and communities that need them most.

Taken together, these sessions document not just the group's contributions to the scientific community, but the arc of the field itself: from establishing ML workflows to deploying foundational models at scale.

Acknowledgements

We are deeply grateful to NASA Headquarters and the NASA EarthRISE Program for their vision and sustained commitment to open, community-driven capacity building. In particular, we wish to thank Natasha Sadoff (NASA Headquarters), Dan Irwin (EarthRISE Program Manager), Eric Anderson (EarthRISE Deputy Program Manager), Brent Roberts (EarthRISE Chief

Scientist), Ashutosh Limaye (NASA Headquarters), and Nancy Searby (NASA Headquarters Retired).

The following individuals provided invaluable peer review whose thoughtful feedback substantially strengthened every chapter: Chishan Zhang (University of Illinois Urbana-Champaign), Kyle Woodward (Delo Insurance Solutions / Spatial Informatics Group), Lilly Thomas (Development Seed), Ate Poortinga (Spatial Informatics Group), John Dilger (SynMax), Nick LaHaye (Spatial Informatics Group), Karlee Zammit (Natural Resources Canada), Alqamah Sayeed (University of Alabama in Huntsville / NASA EarthRISE), Aparna Phalke (University of Alabama in Huntsville / NASA EarthRISE), Deepali Bidwai (Alpha Square), Sandip Rijal (Florida Atlantic University), Andréa Nicolau Puzzi (Spatial Informatics Group), Silvia Arámbula (Creative Labs STEAM Education), Phoebe Odour (University of Alabama in Huntsville / NASA EarthRISE), Diana West (University of Alabama in Huntsville / NASA EarthRISE), Emil Cherrington (University of Alabama in Huntsville / NASA EarthRISE), Betzy Hernandez (University of Alabama in Huntsville / NASA EarthRISE), Kevin Horn (NASA MSFC / NASA ODSI), Micky Maganini (University of Alabama in Huntsville / NASA EarthRISE), Jacob Abramowitz (University of Alabama in Huntsville / NASA EarthRISE), Kristina Glass (NASA EarthRISE Developers Academy / Analytical Mechanics Associates), Isabelle Le Conche (NASA EarthRISE Developers Academy / Analytical Mechanics Associates), Jacob Ramthun (University of Alabama in Huntsville / NASA EarthRISE), and Lena Pransky (University of Alabama in Huntsville / NASA EarthRISE).

Above all, we thank the readers and practitioners who engage with this material, build upon it, and carry this work forward.

Preface

Forthcoming.

Foreword

Forthcoming.

About the Editors

Tim Mayer is a Research Scientist at the University of Alabama in Huntsville and Lead Data Scientist of the NASA EarthRISE program. Tim has worked at the intersection of applied research and Geospatial Artificial Intelligence for over a decade with NASA. Tim co-leads the Geospatial Artificial Intelligence Working Group. His primary research interest surrounds the applied intersection of Ecology and advanced deep learning techniques.

Biplov Bhandari — *Bio forthcoming.*

David Saah — *Bio forthcoming.*

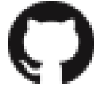
1 Data Preparation

This chapter is under development and will be available in a future edition of the book.

Part I

Semantic Segmentation

2 Crop Mapping — Rice Mapping in Bhutan with U-Net

Run in Colab  View on GitHub

https://www.youtube.com/embed/l3Tv5UBlw9I

2.1 Note: This notebook is meant to be run in Colab. You can still run this locally, make sure you have Google Drive API installed and path adjusted in relevant places.

This notebook is also available in this github repo: https://github.com/SERVIR/servir-aces. Navigate to the `notebook` folder.

2.2 Setup environment

```
from google.colab import drive
drive.mount("/content/drive")
```

```
!pip install servir-aces
```

```
!git clone https://github.com/SERVIR/servir-aces
```

2.2.1 Download datasets

For this chapter, we have already prepared and exported the training datasets. They can be found at the google cloud storage and we will use `gsutil` to get the dataset in our workspace. The dataset has `training`, `testing`, and `validation` subdirectory. Let's start by downloading these datasets in our workspace.

If you're looking to produce your own datasets, you can follow this notebook which was used to produce these training, testing, and validation datasets provided in this notebook.

```
!mkdir -p content/datasets
```

```
!gsutil -m cp -r gs://dl-book/chapter-1 content/datasets/
```

2.2.2 Setup config file variables

Now the repo is downloaded. We will create an environment file file to place point to our training data and customize parameters for the model. To do this, we make a copy of the `.env.example` file provided.

Under the hood, all the configuration provided via the environment file are parsed as a config object and can be accessed programatically.

Note current version does not expose all the model intracacies through the environment file but future version may include those depending on the need.

```
!cp servir-aces/.env.example servir-aces/config.env
```

Okay, now we have the `config.env` file, we will use this to provide our environments and parameters.

Note there are several parameters that can be changed. Let's start by changing the `BASEDIR` and `OUTPUT_DIR` as below.

```
BASEDIR = "/content/"
OUTPUT_DIR = "/content/drive/MyDrive/Colab Notebooks/DL_Book/Chapter_1/output"
```

We will start by training a U-Net model using the `dl-book/chapter-1/unet_256x256_planet_wo_indices` dataset inside the `dataset` folder for this exercise. Let's go ahead and change our DATADIR in the `config.env` file as below.

```
DATADIR = "datasets/unet_256x256_planet_wo_indices"
```

These datasets have RGBN from Planetscope mosiac. Since we are trying to map the rice fields, we use growing season and pre-growing season information. Thus, we have 8 optical bands, namely `red_before`, `green_before`, `blue_before`, `nir_before`, `red_during`, `green_during`, `blue_during`, and `nir_during`. In adidition, you can use `USE_ELEVATION` and `USE_S1` config to include the topographic and radar information. Since this datasets have toppgraphic and radar features, so we won't be settting these config values. Similarly, these datasets are tiled to 256x256 pixels, so let's also change that.

```
# For model training, USE_ELEVATION extends FEATURES with "elevation" & "slope"
# USE_S1 extends FEATURES with "vv_asc_before", "vh_asc_before", "vv_asc_during", "vh_asc_du
# "vv_desc_before", "vh_desc_before", "vv_desc_during", "vh_desc_during"
# In case these are not useful and you have other bands in your training data, you can do se
# USE_ELEVATION and USE_S1 to False and update FEATURES to include needed bands
USE_ELEVATION = False
USE_S1 = False

PATCH_SHAPE = (256, 256)
```

Next, we need to calculate the size of the traiing, testing and validation dataset. For this, we know our size before hand. But `aces` also provides handful of functions that we can use to calculate this. See this notebook to learn more about how to do it. We will also change the `BATCH_SIZE` to 32; if you have larger memory available, you can increase the `BATCH_SIZE`. You can run for longer `EPOCHS` by changing the `EPOCHS` paramter; we will keep it to 5 for now.

```
# Sizes of the training and evaluation datasets.
TRAIN_SIZE = 8531
TEST_SIZE = 1222
VAL_SIZE = 2404
BATCH_SIZE = 32
EPOCHS = 30
```

2.2.3 Update the config file programtically

We can also make a dictionary so we can change these config settings programatically.

```
BASEDIR   = "/content/" # @param {type:"string"}
OUTPUT_DIR = "/content/drive/MyDrive/Colab Notebooks/DL_Book/Chapter_1/output" # @param {type
DATADIR = "datasets/unet_256x256_planet_wo_indices" # @param {type:"string"}
# PATCH_SHAPE, USE_ELEVATION, USE_S1, TRAIN_SIZE, TEST_SIZE, VAL_SIZE
# BATCH_SIZE, EPOCHS are converted to their appropriate type.
USE_ELEVATION = "False" # @param {type:"string"}
USE_S1 = "False" # @param {type:"string"}
PATCH_SHAPE = "(256, 256)" # @param {type:"string"}
TRAIN_SIZE = "8531" # @param {type:"string"}
TEST_SIZE = "1222" # @param {type:"string"}
VAL_SIZE = "2404" # @param {type:"string"}
BATCH_SIZE = "32" # @param {type:"string"}
EPOCHS = "30" # @param {type:"string"}
MODEL_DIR_NAME = "unet_v1" # @param {type:"string"}
```

```
unet_config_settings = {
    "BASEDIR" : BASEDIR,
    "OUTPUT_DIR": OUTPUT_DIR,
    "DATADIR": DATADIR,
    "USE_ELEVATION": USE_ELEVATION,
    "USE_S1": USE_S1,
    "PATCH_SHAPE": PATCH_SHAPE,
    "TRAIN_SIZE": TRAIN_SIZE,
    "TEST_SIZE": TEST_SIZE,
    "VAL_SIZE": VAL_SIZE,
    "BATCH_SIZE": BATCH_SIZE,
    "EPOCHS": EPOCHS,
    "MODEL_DIR_NAME": MODEL_DIR_NAME,
}
```

```
import dotenv

config_file = "servir-aces/config.env"

for config_key in unet_config_settings:
    dotenv.set_key(dotenv_path=config_file,
                   key_to_set=config_key,
                   value_to_set=unet_config_settings[config_key]
                   )
```

2.3 U-Net Model

2.3.1 Load config file variables

```
from aces import Config, DataProcessor, ModelTrainer, EEUtils
```

Let's load our config file through the `Config` class.

```
unet_config = Config(config_file=config_file)
```

Most of the config in the `config.env` is now available via the config instance. Let's check few of them here.

```
unet_config.TRAINING_DIR, unet_config.OUTPUT_DIR, unet_config.BATCH_SIZE, unet_config.TRAIN_S
```

2.3.2 Load `ModelTrainer` class

Next, let's make an instance of the `ModelTrainer` object. The `ModelTrainer` class provides various tools for training, buidling, compiling, and running specified deep learning models.

```
unet_model_trainer = ModelTrainer(unet_config, seed=42)
```

2.3.3 Train and Save U-Net model

`ModelTrainer` class provides various functionality. We will use `train_model` function that helps to train the model using the provided configuration settings.

This method performs the following steps: - Configures memory growth for TensorFlow. - Creates TensorFlow datasets for training, testing, and validation. - Builds and compiles the model. - Prepares the output directory for saving models and results. - Starts the training process. - Evaluates and prints validation metrics. - Saves training parameters, plots, and models.

```
unet_model_trainer.train_model()
```

2.3.4 Save the config file

```
from pathlib import Path
import shutil

config_file = Path(config_file)
drive_config_file = Path(unet_config.MODEL_DIR / f"{str(config_file).split('/')[-1]}")

# Create the target directory if it doesn't exist
drive_config_file.parent.mkdir(parents=True, exist_ok=True)

# Copy the file
shutil.copy(config_file, drive_config_file)

print(f"File copied from {config_file} to {drive_config_file}")
```

2.3.5 Load the logs files via TensorBoard

Tensorboard provides a unique way to view and interact with the logs while the model is being trained. Learn more here. Here we only show you how you can load them to tensorboard with our training logs.

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

```
log_dir_unet = f"{str(unet_config.MODEL_DIR)}/logs"
log_dir_unet
```

```
%tensorboard --logdir "{log_dir_unet}"
```

2.3.6 Load the Saved U-Net Model

Load the saved model

```
import tensorflow as tf
```

```
unet_model = tf.keras.models.load_model(f"{str(unet_config.MODEL_DIR)}/trained-model")
```

```
print(unet_model.summary())
```

2.3.7 Inference using Saved U-Net Model

Now we can use the saved model to start the export of the prediction of the image. For prediction, you would need to first prepare your image data. We have already exported the image needed here, which we will use for now. See this notebook to understand how we did it.

In addition, this notebook shows how you can then use the image to predict from the saved Model.

In any case, you now have the prediction in the Earth Engine as image.

2.4 DNN Model

2.4.1 Setup any changes in the config file for DNN Model

There are few config variables that needs to be changed for running a DNN model. First would be the data itself so let's change the `DATADIR`. We also need to change our output directory using `MODEL_DIR_NAME`. This is the sub-directory inside the `OUTPUT_DIR` for this model run. We also need to specify this is the DNN model that we want to run. We have `MODEL_TYPE` parameter for that. Currently, it supports unet, dnn, and cnn (case sensitive) models; default being unet. Make other changes, as appropriate.

```
DATADIR = "datasets/dnn_planet_wo_indices"
MODEL_DIR_NAME = "dnn_v1"
MODEL_TYPE = "dnn"
```

2.4.2 Update the config file programtically

```
DATADIR = "datasets/dnn_planet_wo_indices" # @param {type:"string"}
# PATCH_SHAPE, USE_ELEVATION, USE_S1, TRAIN_SIZE, TEST_SIZE, VAL_SIZE
# BATCH_SIZE, EPOCHS are converted to their appropriate type.
MODEL_DIR_NAME = "dnn_v1" # @param {type:"string"}
MODEL_TYPE = "dnn" # @param {type:"string"}
BATCH_SIZE = "32" # @param {type:"string"}
EPOCHS = "30" # @param {type:"string"}
```

```
dnn_config_settings = {
    "DATADIR": DATADIR,
    "MODEL_DIR_NAME": MODEL_DIR_NAME,
    "MODEL_TYPE": MODEL_TYPE,
    "BATCH_SIZE": BATCH_SIZE,
    "EPOCHS": EPOCHS,
}
```

```
for config_key in dnn_config_settings:
    dotenv.set_key(dotenv_path=config_file,
                   key_to_set=config_key,
                   value_to_set=dnn_config_settings[config_key]
                   )
```

2.4.3 Load config file variables for DNN Model

```
dnn_config = Config(config_file=config_file, override=True)
```

Most of the config in the `config.env` is now available via the config instance. Let's check few of them here.

```
dnn_config.TRAINING_DIR, dnn_config.OUTPUT_DIR, dnn_config.BATCH_SIZE, dnn_config.MODEL_TYPE
```

2.4.4 Load `ModelTrainer` class

Next, let's make an instance of the `ModelTrainer` object. The `ModelTrainer` class provides various tools for training, buidling, compiling, and running specified deep learning models.

```
dnn_model_trainer = ModelTrainer(dnn_config, seed=42)
```

2.4.5 Train and Save DNN model

```
dnn_model_trainer.train_model()
```

2.4.6 Save the config file

```
drive_config_file = Path(dnn_config.MODEL_DIR / f"{str(config_file).split('/')[-1]}")

# Create the target directory if it doesn't exist
drive_config_file.parent.mkdir(parents=True, exist_ok=True)

# Copy the file
shutil.copy(config_file, drive_config_file)

print(f"File copied from {config_file} to {drive_config_file}")
```

2.4.7 Load the logs files via TensorBoard

```
log_dir_dnn = f"{str(dnn_config.MODEL_DIR)}/logs"
log_dir_dnn
```

```
%tensorboard --logdir "{log_dir_dnn}"
```

2.4.8 Load the Saved DNN Model

```
dnn_model = tf.keras.models.load_model(f"{str(dnn_config.MODEL_DIR)}/trained-model")
```

```
print(dnn_model.summary())
```

2.4.9 Inference using Saved DNN Model

Now we can use the saved model to start the export of the prediction of the image. For prediction, you would need to first prepare your image data. We have already exported the image needed here, which we will use for now. See this notebook to understand how we did it.

In addition, this notebook shows how you can then use the image to predict from the saved Model.

In any case, you now have the prediction in the Earth Engine as image.

2.5 Independent Validation

For independent validation, we will use a file that we have prepared. These files were collected using Collect Earth Online by SCO and NASA DEVELOP interns. We will be using GEE here. Before we do that, let's make changes in our config file.

We will make sure our `GCS_PROJECT` is setup correctly.

```
GCS_PROJECT = "servir-ee"
```

2.5.1 Update the config file

```
GCS_PROJECT = "servir-ee" # @param {type:"string"}
```

```
config_settings = {
    "GCS_PROJECT": GCS_PROJECT,
}
```

```
for config_key in config_settings:
    dotenv.set_key(dotenv_path=config_file,
                   key_to_set=config_key,
                   value_to_set=config_settings[config_key]
                   )
```

2.5.2 Load config file variable

```
config = Config(config_file=config_file, override=True)
```

2.5.3 Import earthengine and geemap for visualization

```
# Import, authenticate and initialize the Earth Engine library.
import ee
ee.Authenticate()
EEUtils.initialize_session(use_highvolume=True, project=config.GCS_PROJECT)
```

```
import geemap

Map = geemap.Map()
```

2.5.4 Class Information and Masking

```
# CLASS
# 0 - cropland etc.
# 1 - rice
# 2 - forest
# 3 - Built up
# 4 - Others (includes water body)
```

```
l1 = ee.FeatureCollection("projects/servir-sco-assets/assets/Bhutan/BT_Admin_1")
paro = l1.filter(ee.Filter.eq("ADM1_EN", "Paro"))


# mask the rice growing zone
# in Paro, rice grows upto 2600 m asl (double check to make sure??)
dem = ee.Image("MERIT/DEM/v1_0_3") # ee.Image('USGS/SRTMGL1_003')
dem = dem.clip(paro)
rice_zone = dem.gte(0).And(dem.lte(2600))
```

2.5.5 Model: U-Net

2.5.5.1 Load and visualize the prediction output

```
UNET_RGBN = ee.Image("projects/servir-ee/assets/dl-book/chapter-1/prediction/prediction_unet
UNET_RGBN = UNET_RGBN.updateMask(rice_zone)
Map.centerObject(UNET_RGBN, 11)
Map.addLayer(UNET_RGBN.clip(paro), {"bands": ["prediction"], "min":0, "max":4, "palette": ["
Map
```

2.5.5.2 Calculate classification metrics

Remapping to rice and non-rice output

```
UNET_RGBN_remapped = UNET_RGBN.remap([0, 1, 2, 3, 4], [0, 1, 0, 0, 0], 0, "prediction")
Map.addLayer(UNET_RGBN_remapped, {"min": 0, "max": 1, "palette": ["cfcf00", "267300"]}, "UNE
Map
```

```
sampling_geom = ee.FeatureCollection("projects/servir-ee/assets/dl-book/chapter-1/data/sample
ceo_final_data = ee.FeatureCollection("projects/servir-ee/assets/dl-book/chapter-1/data/ceoDa
ceo_final_data = ee.FeatureCollection(ceo_final_data.filter(ee.Filter.bounds(sampling_geom).
```

```
prediction_unet = UNET_RGBN_remapped.sampleRegions(
    collection = ceo_final_data,
    scale = 10,
    geometries = True
)

# print("predictionOutputUnet", prediction_unet.getInfo())
```

```
error_matrix_unet = prediction_unet.errorMatrix(actual="rice", predicted="remapped")
test_acc_unet = error_matrix_unet.accuracy()
test_kappa_unet = error_matrix_unet.kappa()
test_recall_producer_acc_unet = error_matrix_unet.producersAccuracy().get([1, 0])
test_precision_consumer_acc_unet = error_matrix_unet.consumersAccuracy().get([0, 1])
f1_unet = error_matrix_unet.fscore().get([1])
```

```
print("error_matrix_unet", error_matrix_unet.getInfo())
print("test_acc_unet", test_acc_unet.getInfo())
print("test_kappa_unet", test_kappa_unet.getInfo())
print("test_recall_producer_acc_unet", test_recall_producer_acc_unet.getInfo())
print("test_precision_consumer_acc_unet", test_precision_consumer_acc_unet.getInfo())
print("f1_unet", f1_unet.getInfo())
```

2.5.5.3 Calculate Probability Distribution

```
prob_output_unet = UNET_RGBN.select(["prediction", "others_etc", "cropland_etc", "urban", "fo
                            .rename(["prediction_class", "others_prob", "cropland_prob", "url
                            .sampleRegions(collection=ceo_final_data, scale=10, geometries=T

# print("prob_output_unet", prob_output_unet.getInfo())
```

```
prob_output_unet = prob_output_unet.getInfo()
```

2.5.6 Model: DNN

2.5.6.1 Load and visualize the prediction output

```
DNN_RGBN = ee.Image("projects/servir-ee/assets/dl-book/chapter-1/prediction/prediction_dnn_v
DNN_RGBN = DNN_RGBN.updateMask(rice_zone)
Map.centerObject(DNN_RGBN)
Map.addLayer(DNN_RGBN.clip(paro), {"bands": ["prediction"], "min":0, "max":4, "palette": ["FF
Map
```

2.5.6.2 Calculate classification metrics

```
DNN_RGBN_remapped = DNN_RGBN.remap([0, 1, 2, 3, 4], [0, 1, 0, 0, 0], 0, "prediction")
Map.addLayer(DNN_RGBN_remapped, {"min": 0, "max": 1, "palette": ["cfcf00", "267300"]}, "DNN_R
Map
```

```
prediction_dnn = DNN_RGBN_remapped.sampleRegions(
    collection = ceo_final_data,
    scale = 10,
    geometries = True
)

# print("predictionOutputDNN", prediction_dnn.getInfo())
```

```
error_matrix_dnn = prediction_dnn.errorMatrix(actual="rice", predicted="remapped")
test_acc_dnn = error_matrix_dnn.accuracy()
test_kappa_dnn = error_matrix_dnn.kappa()
test_recall_producer_acc_dnn = error_matrix_dnn.producersAccuracy().get([1, 0])
test_precision_consumer_acc_dnn = error_matrix_dnn.consumersAccuracy().get([0, 1])
f1_dnn = error_matrix_dnn.fscore().get([1])
```

```
print("error_matrix_dnn", error_matrix_dnn.getInfo())
print("test_acc_dnn", test_acc_dnn.getInfo())
print("test_kappa_dnn", test_kappa_dnn.getInfo())
print("test_recall_producer_acc_dnn", test_recall_producer_acc_dnn.getInfo())
print("test_precision_consumer_acc_dnn", test_precision_consumer_acc_dnn.getInfo())
print("f1_dnn", f1_dnn.getInfo())
```

2.5.6.3 Calculate Probability Distribution

```
prob_output_dnn = DNN_RGBN.select(["prediction", "others_etc", "cropland_etc", "urban", "fore
                            .rename(["prediction_class", "others_prob", "cropland_prob", "url
                            .sampleRegions(collection=ceo_final_data, scale=10, geometries=Ti

# print("prob_output_dnn", prob_output_dnn.getInfo())
```

```
prob_output_dnn = prob_output_dnn.getInfo()
```

2.6 Figures and Plots

2.6.1 Training and Validation Plot

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pickle

%matplotlib inline
```

```
with open(unet_config.MODEL_DIR / "model.pkl", "rb") as f:
    unet_model_metrics = pickle.load(f)

with open(dnn_config.MODEL_DIR / "model.pkl", "rb") as f:
    dnn_model_metrics = pickle.load(f)
```

```
# Create subplots for different metrics in a 3x4 grid
fig, axs = plt.subplots(2, 4, figsize=(4*7, 6*2))

colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]
metrics = ["loss", "precision", "recall", "categorical_accuracy"]
metrics_name = ["Loss", "Precision", "Recall", "Categorical Accuracy"]

epochs = range(1, config.EPOCHS + 1)

title_fontsize = 22
label_fontsize = 22
legend_fontsize = 15
tick_fontsize = 18
lw=1.5

for i in range(2):
    for y in range(len(metrics)):
        if i == 1:
            axs[i][y].plot(epochs, unet_model_metrics[f"val_{metrics[y]}"], color=colors[0],
            axs[i][y].plot(epochs, dnn_model_metrics[f"val_{metrics[y]}"], color=colors[1],
            axs[i][y].set_title(f"Validate {metrics_name[y]}", fontsize=title_fontsize)
            axs[i][y].set_xlabel("epochs", fontsize=label_fontsize)
            axs[i][y].set_ylabel(f"{metrics[y]}", fontsize=label_fontsize)
            axs[i][y].grid(linestyle="dotted", alpha=0.7)
```

```
            axs[i][y].legend(fontsize=legend_fontsize)
            axs[i][y].tick_params(axis="both", which="major", labelsize=tick_fontsize)
        else:
            axs[i][y].plot(epochs, unet_model_metrics[metrics[y]], color=colors[0], lw=lw, ma
            axs[i][y].plot(epochs, dnn_model_metrics[metrics[y]], color=colors[1], lw=lw, mar
            axs[i][y].set_title(f"Train {metrics_name[y]}", fontsize=title_fontsize)
            axs[i][y].set_xlabel("epochs", fontsize=label_fontsize)
            axs[i][y].set_ylabel(f"{metrics[y]}", fontsize=label_fontsize)
            axs[i][y].grid(linestyle="dotted", alpha=0.7)
            axs[i][y].legend(fontsize=legend_fontsize)
            axs[i][y].tick_params(axis="both", which="major", labelsize=tick_fontsize)


# Adjust layout and show the plot
plt.tight_layout()
# plt.savefig("metrics_plot_model_comparison.png", dpi=500, bbox_inches="tight")
plt.show()
```

```
# Create subplots for different metrics in a 3x4 grid
fig, axs = plt.subplots(1, 4, figsize=(4*7, 6*1))

colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]
metrics = ["loss", "precision", "recall", "categorical_accuracy"]
metrics_name = ["Loss", "Precision", "Recall", "Categorical Accuracy"]

epochs = range(1, config.EPOCHS + 1)

title_fontsize = 22
label_fontsize = 22
legend_fontsize = 15
tick_fontsize = 18
lw=1.5

for y in range(len(metrics)):
    axs[y].plot(epochs, unet_model_metrics[f"val_{metrics[y]}"], color=colors[0], marker="o"
    axs[y].plot(epochs, dnn_model_metrics[f"val_{metrics[y]}"], color=colors[1], lw=lw, marke

    axs[y].plot(epochs, unet_model_metrics[metrics[y]], color=colors[2], lw=lw, marker="o", l
    axs[y].plot(epochs, dnn_model_metrics[metrics[y]], color=colors[3], lw=lw, marker="o", la
    axs[y].set_title(f"{metrics_name[y]}", fontsize=title_fontsize)
    axs[y].set_xlabel("epochs", fontsize=label_fontsize)
    axs[y].set_ylabel(f"{metrics[y]}", fontsize=label_fontsize)
```

```
    axs[y].grid(linestyle="dotted", alpha=0.7)
    axs[y].legend(fontsize=legend_fontsize)
    axs[y].tick_params(axis="both", which="major", labelsize=tick_fontsize)


# Adjust layout and show the plot
plt.tight_layout()
# plt.savefig("metrics_plot_model_comparison.png", dpi=500, bbox_inches="tight")
plt.show()
```

2.6.2 Probability Distribution Plot

```
all_data = {}

unet_data = []
dnn_data = []

unet_rice_data = []
dnn_rice_data = []

unet_other_data = []
dnn_other_data = []

for i, feature in enumerate(prob_output_unet["features"]):
    unet_rice_prob = round(feature["properties"]["rice_prob"], 5)
    unet_other_prob = round(feature["properties"]["cropland_prob"] + round(feature["propertie
    unet_data.append([unet_rice_prob, unet_other_prob])
    unet_rice_data.append(unet_rice_prob)
    unet_other_data.append(unet_other_prob)

    dnn_feature = prob_output_dnn["features"][i]
    dnn_rice_prob = round(dnn_feature["properties"]["rice_prob"], 5)
    dnn_other_prob = 1. - round(dnn_feature["properties"]["rice_prob"], 5)
    # dnn_other_prob = round(dnn_feature["properties"]["cropland_prob"] + dnn_feature["prope
    dnn_data.append([dnn_rice_prob, dnn_other_prob])
    dnn_rice_data.append(dnn_rice_prob)
    dnn_other_data.append(dnn_other_prob)
```

```python
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8, 5))

title_fontsize = 22
label_fontsize = 10
tick_fontsize = 10

# rectangular box plot
bplot1 = ax1.boxplot([unet_rice_data, dnn_rice_data],
                     notch=True,
                     vert=True,  # vertical box alignment
                     patch_artist=True,  # fill with color
                     labels=["U-Net", "DNN"],
                     sym="k+")  # will be used to label x-ticks
ax1.set_title("Rice Probability")

# notch shape box plot
bplot2 = ax2.boxplot([unet_other_data, dnn_other_data],
                     notch=True,  # notch shape
                     vert=True,  # vertical box alignment
                     patch_artist=True,  # fill with color
                     labels=["U-Net", "DNN"],
                     sym="k+")  # will be used to label x-ticks
ax2.set_title("Other Probability")
```

3 Selective Logging Detection with Deep Learning and Very-High Resolution Imagery

3.1 Part 2: Modeling

This notebook shows the workflow used to train a deep learning model with the patches saved from the previous notebook (Notebook n.º1). We will load our TFRecords, train a model with them and save it for further use (inference). The notebook was developed and configured specifically to run on Google Colab, so its implementation is optimized to run on that platform. In addition, it is necessary to configure the environment to enable GPU computing by changing the runtime type. First, navigate to *Edit → Notebook Settings* and select GPU from the Hardware Accelerator options.

3.2 1. Data collection and initial settings

We are going to connect **Google Drive** with **Google Colab** to manage the exported data (patches) from the previous Notebook.

```
from google.colab import drive
drive.mount('/content/drive')
```

Get the data that has been staged Yupanqui Carrasco, O., & Quispe Sedano, M. J. (2026). Selective Logging Detection with Deep Learning and Very-High Resolution Imagery_TF_records [Data set]. Zenodo. https://doi.org/10.5281/zenodo.19614389

```
import os
import zipfile
import requests

# =========================
# CONFIG
# =========================
use_drive = True  # True = save to Google Drive, False = save to /content
```

```
if use_drive:
    from google.colab import drive
    drive.mount('/content/drive')
    base_path = "/content/drive/MyDrive/DL_book"
else:
    base_path = "/content/DL_book"

# Zenodo file URL (direct download)
url = "https://zenodo.org/records/19614389/files/dataset.zip?download=1"

zip_path = os.path.join(base_path, "dataset.zip")

# =========================
# SETUP DIRECTORY
# =========================
os.makedirs(base_path, exist_ok=True)

# =========================
# DOWNLOAD FILE
# =========================
print("Downloading dataset...")
response = requests.get(url, stream=True)

with open(zip_path, "wb") as f:
    for chunk in response.iter_content(chunk_size=8192):
        if chunk:
            f.write(chunk)

print(f"Downloaded to: {zip_path}")

# =========================
# UNZIP FILE
# =========================
print("Extracting dataset...")
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(base_path)

print(f"Extracted to: {base_path}")

# =========================
# (OPTIONAL) DELETE ZIP
# =========================
```

```
os.remove(zip_path)
print("Zip file removed.")
```

Let's see the contents of our Drive folder using the linux command `!ls`. Note: This command only runs on Google Colab

```
!ls -F /content/drive/MyDrive/DL_Book
```

Our folder should contain several files whose names begin with: - train - testing - validation

Those files are our exported patches from the previous Notebook. If you do not wish to run the previous notebook , you can create a copy from the original folder. Click here to go to the Drive folder. Make sure to modify the paths to match the current Notebook.

We will also import some libraries that will contribute to the training of the model. This notebook was built and tested using Google Colab, so the libraries used correspond to the latest version available on this platform at the time of construction and testing (e.g. Tensorflow 2.19.0). Check the latest versions of TensorFlow here.

```
import tensorflow as tf
print(tf.__version__)
```

Numpy is another important library, that allow us to perform operations with tensors, matrices and vectors.

```
import numpy as np
```

We will also import Matplotlib, for the data visualization.

```
import matplotlib.pyplot as plt
```

Finally, we will also import os, to read files and manipulate paths.

```
import os
```

3.3 2. Data visualization

We will start by viewing a single TFRecord file, if we explore the folder where the files were downloaded, we can see that they are "compressed" with a special `.gz` format, Tensorflow allows the use of this special format, without the need to decompress the files.
We will start by creating a variable called record, which will include in a container `tf.data.TFRecordDataset()` the path of an exported file. *Please verify that the export path is the same, in the previous example we exported our data to the DL_Book folder in Google Drive*

```
record = tf.data.TFRecordDataset('/content/drive/MyDrive/DL_Book/testing_0.tfrecord.gz', comp
print(record)
```

We can see that the TFRecordDataset does not show all the information, we cannot know the `shape` of the tensors, even though we know based on the previous code that they have a 128 by 128 shape. We also cannot know the `dtype` (the data type) neither the number of elements that our TFRecordDataset has.

Because TFRecords are binary storage files, we must transform them into a readable structure. To do this, we will create a dictionary that contains the bands that we exported in the previous code and assign each band a shape with `tf.io.FixedLenFeature()` and a data type.

```
features = {
    'b1': tf.io.FixedLenFeature([128, 128], tf.float32),
    'b2': tf.io.FixedLenFeature([128, 128], tf.float32),
    'b3': tf.io.FixedLenFeature([128, 128], tf.float32),
    'b4': tf.io.FixedLenFeature([128, 128], tf.float32),
    'class': tf.io.FixedLenFeature([128, 128], tf.float32),
}
```

We will create a function to read a serialized example into the structure defined by the dictionary created above.

```
def parse_tfrecord(example_proto):
    return tf.io.parse_single_example(example_proto, features) # This parses each TFRecord to
```

We will apply the newly created function to our TFRecordDataset using the `.map()` function, which allows iterating through each element of our TFRecordDataset.

```
serialized = record.map(parse_tfrecord) # With .map() we loop through each element of our TFR
print(serialized)
```

We can see that the newly created object named **serialized** has a different form than the original TFRecordDataset. This new object has the structure defined by the dictionary created. To display the numerical values of each of the bands of our new object, we will use the function `get_single_element()`.

```
element = serialized.get_single_element() # With .get_single_element() we retrieve an individ
print(element)
```

The result is a dictionary, from which we can extract each band using `get()` followed by the `key`, which in this case is the name of the exported band.

```
b1 = element.get('b1')
b2 = element.get('b2')
b3 = element.get('b3')
b4 = element.get('b4')
label = element.get('class')
```

We can now view the numerical values of any exported band. We can also transform the objects to a numpy array using the `numpy()` function.
Now we will create an "image" to be able to visualize it, concatenating the values of the created bands, into a single tensor.

```
img = tf.constant([b3.numpy(), b2.numpy(), b1.numpy()]) # This appends each band to a consta
print(img)
```

If we visualize the shape of the tensor, we see that it has the shape of (3, 128, 128), in order to plot it we must transform the shape to (128, 128, 3). To do this we will use the `.transpose()` function. In addition, since Matplotlib does only tolerate values from 0 to 255, we will normalize our image (tensor) with `.divide()`.

```
imgRGB = tf.transpose(tf.divide(img, 12000), [1, 2, 0])
```

Now we will plot the results using the matplotlib library. We will be able to contrast how the logging activity is visualized with a SkySat (0.5 m) image (on the left), and the hand-digitized label (on the right).

```
fig, axs = plt.subplots(1, 2, figsize = (15, 15))

axs[0].set_title('RGB Image')
axs[0].imshow(imgRGB)
```

```
axs[1].set_title('Label')
axs[1].imshow(label.numpy())

plt.show()
```

This plot compares the exported SkySat satellite image patch with the hand-digitized label. Performing the manual digitization process takes many hours, especially when the logging activity is intense. Deep Learning models greatly speed up this manual process and allow the user to focus on evaluating the output.

3.4 3. Data processing

3.4.1 3.1 Definition of variables

Before performing data processing, some variables will be defined for the training. - The variable `kernel_size` contains the size of the patch, from which we build its `shape`. - The variable `patch_path` refers to the folder containing the previously exported patches. - `buffer_size` this number is the amount of data that will be filled when shuffling. For example, if our data contains 100 records and we have a buffer_size of 10, then `shuffle` will select a random element from the first 10 records. The space of this element will be replaced by the 101-st element. Since our dataset contains less than 250 files, and we want the model to shuffle randomly the **whole dataset** then we will define this parameter to 1000 (it could be higher).

The model also needs hyperparameters, defining these parameters can sometimes be time consuming and experimental. Some of the most important are described below: - `batch_size`: The number of examples used in a pass throught the network. A high batch size can lead to faster training times but may result in lower accuracy and overfitting, while a small batch sizes can provide better accuracy, but can be computationally expensive and time-consuming. We will set our batch size to 4 (because this is a small dataset). - `epochs`: This is the amount of training cycles through all of the samples in the training dataset. This is how many times the model will see the entire training data before completing training. We will set this value to 10. - `learning_rate`: This controls how much the model's parameters are adjusted during each training step. This will be set to 0.1

```
bands = ['b1', 'b2', 'b3', 'b4']
response = ['class']
features = bands + response

kernel_size = 128
kernel_shape = [kernel_size, kernel_size]
```

```
columns = [
  tf.io.FixedLenFeature(shape = kernel_shape, dtype = tf.float32) for k in features
]

# Path to the folder with patches (Benchmark dataset)
patch_path = '/content/drive/MyDrive/DL_Book'
train_path = f'{patch_path}/train*'
val_path   = f'{patch_path}/validation*'
test_path  = f'{patch_path}/testing*'

features_dict = dict(zip(features, columns))

train_size = len([f for f in os.listdir(patch_path) if f.startswith('train')])
val_size = len([f for f in os.listdir(patch_path) if f.startswith('validation')])
test_size = len([f for f in os.listdir(patch_path) if f.startswith('testing')])

# Hyperparameters
batch_size = 4
epochs = 10
buffer_size = 1000
learning_rate = 0.1
```

3.4.2 3.2 Create a TFRecordDataset

TFRecordDataset are a collection of TFRecords, so they are very useful for storing large amounts of data, and are optimized to save memory.
First we will create a function called `.parse_tfrecord()`, which will perform the same functionality as in the data visualization, done previously. We will transform the structure of each TFRecord.

```
def parse_tfrecord(example_proto):
    return tf.io.parse_single_example(example_proto, features_dict)
```

Second, a function called `.to_tuple()` will be created, which will convert the tensor dictionary to a tuple, with the scheme (inputs, outputs).

```
def to_tuple(inputs):
    inputsList = [inputs.get(key) for key in features]
    stacked = tf.stack(inputsList, axis = 0) # This stacks the input list to a single tensor
    stacked = tf.transpose(stacked, [1, 2, 0]) # Transposition of the stacked tensor to the s
    return stacked[:, :, :len(bands)], stacked[:, :, len(bands):]
```

Third, a function will be created to read the tensors, this function will also apply the functions created previously: `.parse_tfrecord()` and `.to_tuple()`

```
def get_dataset(pattern):
    glob = tf.io.gfile.glob(pattern) # This reads the files in our path
    dataset = tf.data.TFRecordDataset(glob, compression_type = 'GZIP') # Add the files to a T
    dataset = dataset.map(parse_tfrecord, num_parallel_calls = 5)
    dataset = dataset.map(to_tuple, num_parallel_calls = 5)
    return dataset
```

We will create 3 TFRecordDataset, which will contain our training, validation and test data set, respectively. To do this we will use functions, within each one there will be an object called `glob` that will indicate the path of the data sets.

```
def get_patch_from_path(path, batch_size, buffer_size = 1000, shuffle = False, repeat = Fals
    dataset = get_dataset(path + '*')
    dataset = dataset.shuffle(buffer_size, seed = 42) if shuffle else dataset # We will shuf
    dataset = dataset.batch(batch_size)
    dataset = dataset.repeat() if repeat else dataset
    return dataset

training_data = get_patch_from_path(train_path, batch_size = batch_size, buffer_size = buffe
validation_data = get_patch_from_path(val_path, batch_size = 1, repeat = True)
testing_data = get_patch_from_path(test_path, batch_size = 1)
```

If we print the 3 sets of data we will see that the training and validation data we see are of type `RepeatDataset()`, that is, they are files that are constantly being generated, which based on the previous functions are being ordered randomly and the Outgoing data set has a size of 1 (based on batch size).

```
print(training_data)
print(validation_data)
print(testing_data)
```

3.5 4. Deep Learning

3.5.1 4.1 Model construction

In this notebook we will use a modification of the U-Net architecture developed by Ronneberger et al., (2015). The U-Net architecture was initially designed for use in biomedical imaging

but proved to be very efficient in the segmentation of satellite and drone images, and is currently one of the most widely used for that purpose. This architecture has demonstrated remarkable effectiveness in various environmental monitoring applications, including agricultural field boundary detection John & Zhang, 2022, urban land use classification, and coastal change detection. Ulmas & Liiv, 2020 further validated U-Net's superiority in crop type mapping from multi-spectral satellite imagery, showing that its ability to preserve spatial resolution through the decoder path results in more precise segmentation boundaries compared to traditional convolutional neural networks or patch-based classification approaches. The architecture's robustness to varying image resolutions and its capacity to learn from limited training data through effective data augmentation make it particularly well-suited for remote sensing applications where high-quality labeled data may be scarce Manos et al., 2022.

To represent the architecture we will use some layers of `tensorflow.keras`. This modified version of U-Net has more encoding and decoding blocks than the original architecture, which increases the depth of the model and its learnability. Unlike the original design, which starts and ends with 64 filters, this implementation starts and ends with a layer of 32 filters, slightly reducing the initial complexity before continuing with the standard progression. Below is a diagram of the modified U-Net architecture used.

This modified architecture allows a superior multi-scale feature extraction (due to the additional encoder and decoder levels). However, this comes with a cost, this architecture has much more parameters compared to the standard U-Net. This will negatively impact training and inference times.

```
def conv_block(input_tensor, num_filters):
    encoder = tf.keras.layers.Conv2D(num_filters, (3, 3), padding='same')(input_tensor)
    encoder = tf.keras.layers.BatchNormalization()(encoder)
    encoder = tf.keras.layers.Activation('relu')(encoder)
    encoder = tf.keras.layers.Conv2D(num_filters, (3, 3), padding='same')(encoder)
    encoder = tf.keras.layers.BatchNormalization()(encoder)
    encoder = tf.keras.layers.Activation('relu')(encoder)
    return encoder

def encoder_block(input_tensor, num_filters):
    encoder = conv_block(input_tensor, num_filters)
    encoder_pool = tf.keras.layers.MaxPooling2D((2, 2), strides=(2, 2))(encoder)
    return encoder_pool, encoder

def decoder_block(input_tensor, concat_tensor, num_filters):
    decoder = tf.keras.layers.Conv2DTranspose(num_filters, (2, 2), strides=(2, 2), padding='
    decoder = tf.keras.layers.concatenate([concat_tensor, decoder], axis=-1)
    decoder = tf.keras.layers.BatchNormalization()(decoder)
    decoder = tf.keras.layers.Activation('relu')(decoder)
```

```
    decoder = tf.keras.layers.Conv2D(num_filters, (3, 3), padding='same')(decoder)
    decoder = tf.keras.layers.BatchNormalization()(decoder)
    decoder = tf.keras.layers.Activation('relu')(decoder)
    decoder = tf.keras.layers.Conv2D(num_filters, (3, 3), padding='same')(decoder)
    decoder = tf.keras.layers.BatchNormalization()(decoder)
    decoder = tf.keras.layers.Activation('relu')(decoder)
    return decoder

def get_model():
    inputs = tf.keras.Input(shape=[kernel_size, kernel_size, len(bands)])
    encoder0_pool, encoder0 = encoder_block(inputs, 32)
    encoder1_pool, encoder1 = encoder_block(encoder0_pool, 64)
    encoder2_pool, encoder2 = encoder_block(encoder1_pool, 128)
    encoder3_pool, encoder3 = encoder_block(encoder2_pool, 256)
    encoder4_pool, encoder4 = encoder_block(encoder3_pool, 512)
    center = conv_block(encoder4_pool, 1024)
    decoder4 = decoder_block(center, encoder4, 512)
    decoder3 = decoder_block(decoder4, encoder3, 256)
    decoder2 = decoder_block(decoder3, encoder2, 128)
    decoder1 = decoder_block(decoder2, encoder1, 64)
    decoder0 = decoder_block(decoder1, encoder0, 32)
    outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(decoder0)

    model = tf.keras.models.Model(inputs=[inputs], outputs=[outputs])
    return model
```

Now we will define the model, using the created function `.get_model()` and compiling it.

The output of our model will be single channel and binary (0 and 1) because we used a `sigmoid`activation function. Therefore, we will use the `BinaryCrossentropy` loss function since it is ideal for binary classifications. In addition, we will use the `Adam` optimizer which is robust on both large and noisy data sets and is considered computationally efficient. Finally, we will use the `IoU` as a metric as it is ideal for binary image classifications because it evaluates the intersection between prediction and actual values.

```
model = get_model()
model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate = learning_rate),
    loss = tf.keras.losses.BinaryCrossentropy(),
    metrics = [tf.keras.metrics.IoU(num_classes = 2, target_class_ids = [0] , name = 'iou')]
```

Once the model is compiled, we can see the layers it has using the `.summary()` function

```
model.summary()
```

3.5.2 4.2 Model training

Once the model is defined, we will proceed to train it, adjusting it to the already processed data set, using the `.fit()` function. Feel free to change the parameters and use the GPU environment!

```
history = model.fit(
    x = training_data,
    epochs = epochs,
    batch_size = batch_size,
    verbose = 1,
    steps_per_epoch = int(train_size / batch_size),
    validation_data = validation_data,
    validation_steps = val_size)
```

We can plot the training results by epoch using matplotlib.

```
# Metrics graph
fig, ax = plt.subplots(nrows = 2, sharex = True, figsize = (15,10))

ax[0].plot(history.history['loss'], color = '#1f77b4', label = 'Training Loss')
ax[0].plot(history.history['val_loss'], linestyle = ':', marker = 'o', markersize = 3, color
ax[0].set_ylabel('Loss')
ax[0].set_ylim(0.0, 1)
ax[0].legend()

ax[1].plot(history.history['iou'], color = '#E5D31F', label = 'Training IoU')
ax[1].plot(history.history['val_iou'], linestyle = ':', marker = 'o', markersize = 3, color =
ax[1].set_ylabel('IoU')
ax[1].legend(loc="lower right")

ax[1].set_xticks(history.epoch)
ax[1].set_xticklabels(range(1, len(history.epoch) + 1, 1))
ax[1].set_xlabel('Epoch')
ax[1].set_ylim(0.0, 1)

plt.legend();
```

The training loss exhibits a clear downward trend, decreasing from approximately 0.08 to around 0.04 over the 10 epochs, indicating that the model is effectively learning and reducing error on the training dataset. The training IoU remains stable at about 0.97 throughout the process, while the validation IoU is consistently slightly higher at approximately 0.975, suggesting good generalization performance.

Although the validation loss shows some instability, particularly a very high value during the first epoch followed by fluctuations in subsequent epochs, it quickly stabilizes to low values, which may be attributed to initialization effects or data scaling issues rather than persistent model misfit. Overall, the model demonstrates strong and consistent segmentation performance.

Given these results, extending the training beyond 10 epochs may yield only marginal improvements. Considering the relatively small dataset, additional training could increase the risk of overfitting without significantly enhancing model performance.

3.5.3 4.3 Model evaluation

Now we will use the test dataset, reserved for this moment. We can get the total metrics for the entire test data set.

```
evaluation = model.evaluate(
    x = testing_data,
    verbose = 1,
    steps = test_size
)
```

We can also apply the model to the test dataset, and graph the results.

```
prediction = model.predict(
    x = testing_data,
    verbose = 1,
    steps = test_size
)
```

By running the following code block we can visualize 3 random images from our test data set, and the model predictions.

```
np.random.seed(42)
n_images = 3

listTest = np.random.choice(test_size, size = n_images, replace = False)
```

```
print(listTest)

fig, axs = plt.subplots(n_images, 2, figsize = (15, 15))

for i in range(len(listTest)):
    imgTest = tf.math.divide(tf.squeeze(list(testing_data)[listTest[i]][0]), 12000)[:, :, 0:3
    imgPred = prediction[listTest[i]].reshape(kernel_size, kernel_size)

    axs[i, 0].set_title('RGB Test Image ' + str(listTest[i]))
    axs[i, 0].imshow(imgTest)
    axs[i, 1].set_title('Prediction ' + str(listTest[i]))
    axs[i, 1].imshow(imgPred, cmap='coolwarm')

plt.subplots_adjust(wspace = -0.3, hspace = 0.4)
plt.show()
```

Despite being trained on a small dataset for only 10 epochs, the model produces high-quality segmentations with well-defined boundaries and contiguous regions, validating both the architecture choice and the training approach for this VHR satellite imagery application. This model proves that effective selective logging monitoring is achievable without massive datasets or extensive computational power. This modified U-Net architecture provides a powerful combination of high precision, data efficiency, and proven real-world effectiveness.

3.5.4 4.4 Save the model

Once we have an adequate model based on the metrics used, we will proceed to save it. To do this we will use the `tf.keras.save_model()` function and we will have to specify the path to save the model.

```
# If necessary
!mkdir /content/drive/MyDrive/DL_Book/model/
```

```
modelDir = '/content/drive/MyDrive/DL_Book/model/logging_model.keras'
tf.keras.models.save_model(model, modelDir)
```

In the next chapter, we will load our model again and apply it to the whole image!

4 Object Detection

This chapter is under development and will be available in a future edition of the book.

Part II

Time Series

5 Phenology-Guided Deep Learning with Uncertainty Quantification for Soybean Yield Prediction

Run in Colab

Authors: Chishan Zhang, Chunyuan Diao

5.1 Overview

This notebook demonstrates the application of multiple deep learning models for crop yield estimation using satellite-derived time series data. The project showcases probabilistic modeling approaches that provide both predictions and uncertainty estimates for agricultural yield forecasting.

5.2 The Challenge with Current Yield Prediction Models

While deep learning has opened new possibilities for crop yield estimation, current approaches face two significant limitations. First, most models provide only deterministic predictions, ignoring the inherent uncertainties in agricultural systems—such as weather variability, satellite noise, and spatial heterogeneity. Without uncertainty quantification, the practical utility of these models for risk assessment is severely limited. # Second, existing models often rely on simple calendar-based aggregation (e.g., weekly or monthly averages). This approach fails to account for the biological reality of crop development, as the impact of environmental factors on yield varies substantially depending on the specific growth stage. To address these gaps, this notebook implements a phenology-guided, probabilistic deep learning framework that explicitly accounts for both temporal crop developmental stages and prediction uncertainties.

5.3 Model Architecture and Approaches

5.3.1 1. Probabilistic LSTM Model

- **Architecture**: Long Short-Term Memory network with probabilistic output layer
- **Output**: Mean and uncertainty estimates using Normal distribution
- **Loss Function**: Negative Log-Likelihood (NLL)
- **Advantages**:
  - Captures temporal dependencies in crop growth
  - Provides uncertainty quantification
  - Handles sequential nature of agricultural data

5.3.2 2. Probabilistic 1D-CNN Model

- **Architecture**: Convolutional Neural Network for time series with probabilistic outputs
- **Layers**: Multiple Conv1D layers with MaxPooling and Dense layers
- **Output**: Mean and uncertainty estimates
- **Advantages**:
  - Captures local temporal patterns
  - Efficient computation
  - Good for detecting seasonal patterns

5.3.3 3. Gaussian Process Regression (GPR)

- **Type**: Bayesian non-parametric model
- **Kernel**: RBF + White noise kernel with automatic hyperparameter optimization
- **Output**: Mean predictions with uncertainty estimates
- **Advantages**:
  - Natural uncertainty quantification
  - Flexible modeling assumptions
  - Works well with limited data

5.4 Key Functionalities

5.4.1 Data Preprocessing

- **Normalization**: Z-score normalization using training data statistics
- **Missing Value Handling**: NaN-aware statistical computations

- **Data Flattening**: Conversion of time series to flat vectors for traditional ML models

5.4.2 Model Training and Evaluation

- **Train/Validation Split**: 80/20 split with fixed random state
- **Early Stopping**: Prevents overfitting in neural networks
- **Reproducibility**: Fixed random seeds for consistent results
- **Cross-Model Comparison**: Standardized evaluation metrics across all models

5.4.3 Uncertainty Quantification

- **Probabilistic Models**: LSTM, CNN, and GPR provide prediction uncertainties
- **Confidence Intervals**: 95% confidence interval coverage analysis
- **Uncertainty Visualization**: Scatter plots with uncertainty bands

5.4.4 Feature and Temporal Importance Analysis

- **Permutation Importance**: Identifies most influential input features
- **Phenological Analysis**: Determines critical growth stages for yield prediction
- **Temporal Patterns**: Visualizes importance across growing season

5.4.5 Spatial-Temporal Analysis

- **Multi-Year Evaluation**: Model performance across different years (2018-2021)
- **Geographic Visualization**: County-level prediction and uncertainty maps
- **Temporal Generalization**: Tests model robustness across different growing seasons

```
"""
=== COMPREHENSIVE IMPORTS FOR CROP YIELD ESTIMATION NOTEBOOK ===

All required packages and dependencies for the entire notebook.
Place this cell near the top after the overview/description section.

Compatible Package Versions (tested and verified):
Python: 3.11.12
TensorFlow: 2.18.0
Keras (within TensorFlow): 3.8.0
TensorFlow Probability: 0.25.0
NumPy: 2.0.2
Pandas: 2.2.2
```

```
Scikit-learn: 1.6.1
Matplotlib: 3.10.0
Seaborn: 0.13.2
GeoPandas: 1.0.1
"""

# Standard Library Imports
import os
import sys
import time
import random
import warnings
import importlib.util

# Core Data Science Libraries
import numpy as np
import pandas as pd

# Visualization Libraries
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import LinearSegmentedColormap

# Machine Learning Libraries
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, ConstantKernel as C

# Deep Learning Libraries
import tensorflow as tf
import tensorflow_probability as tfp
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import LSTM, Dense, Input, Conv1D, MaxPooling1D, Flatten, Dropou
from tensorflow.keras.callbacks import EarlyStopping

# Geospatial Libraries
try:
    import geopandas as gpd
    GEOPANDAS_AVAILABLE = True
```

```
    print("GeoPandas successfully imported")
except ImportError:
    GEOPANDAS_AVAILABLE = False
    print("GeoPandas not available - geographic visualizations will be skipped")

# TensorFlow Probability Distributions
tfd = tfp.distributions

# Suppress warnings for cleaner output
warnings.filterwarnings('ignore', category=FutureWarning)
warnings.filterwarnings('ignore', category=UserWarning, module='matplotlib')
warnings.filterwarnings('ignore', category=DeprecationWarning)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # Suppress TensorFlow warnings

# Configure TensorFlow for deterministic behavior
tf.config.experimental.enable_op_determinism()
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) > 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)

# Print Keras version
print(tf.keras.__version__)
```

```
# Google Drive mount is optional; skip when using local download_dir.
# If running in Colab with Drive storage, uncomment below:
# from google.colab import drive
# drive.mount('/content/drive')
```

5.5 1. Data Loading and Preprocessing

5.5.1 Study Area

The study area includes the major soybean-producing states of Illinois, Indiana, and Iowa in the U.S. Corn Belt. County-level soybean yields (bu/ac) and phenology-guided predictors are analyzed for the period 2018–2021.

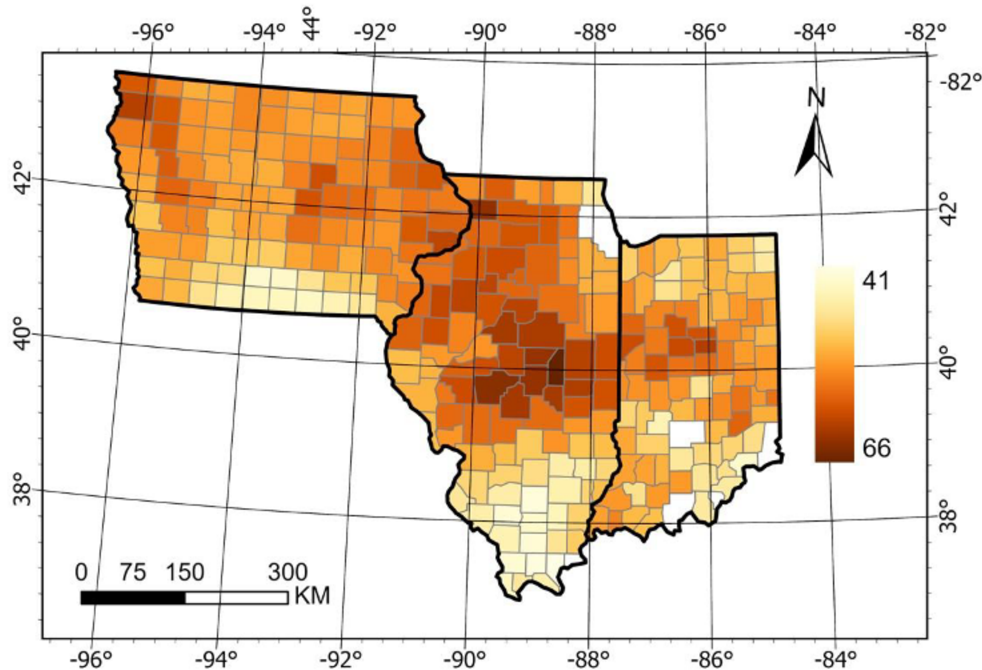


Figure 5.1: Study area map showing Illinois, Indiana, and Iowa with average soybean yields (bu/ac) from 2008 to 2021)

5.5.2 Dataset Overview

Data Download: The data can also be directly downloaded from: https://github.com/pouuh/phenology-guided-soybean-yield/releases/download/v1.0/soybean_phenology_guided_dataset.zip

This section loads the pre-processed satellite-derived time series data for soybean yield prediction. The dataset consists of:

- **Training Set**: Multi-temporal observations from multiple growing seasons
- **Test Set**: Holdout data for final model evaluation
- **Features**: 14 environmental variables captured at regular intervals throughout the growing season
- **Target**: County-level soybean yield measurements (bushels per acre)

5.5.3 Data Structure

- **Input Shape**: `(samples, timesteps, features)` - 3D array representing time series
- **Temporal Resolution**: Weekly to bi-weekly observations during growing season

- **Spatial Resolution**: County-level aggregated values
- **Years Covered**: 2018-2021 for model development and testing Loading

5.5.3.1 Input Features (X)

The input data for our models (`X_train.npy`, `X_test.npy`, `[year]_X.npy`) are time series with a shape of `(samples, timesteps, features)`, where `timesteps` represent 6 phenological stages and `features` include 14 predictors:

- **EVI2 (Enhanced Vegetation Index 2):**
  - *Description:* A satellite-derived measure of vegetation greenness and canopy health.
  - *Significance for Yield:* Higher EVI2 generally indicates a healthier, more photosynthetically active crop, which is essential for biomass accumulation and ultimately for producing higher yields through processes like pod development and seed fill. Stress events affecting EVI2 can signal potential yield reductions.
- **LST_Day (Daytime Land Surface Temperature in °C):**
  - *Description:* The temperature of the ground surface during the day.
- **LST_Night (Nighttime Land Surface Temperature in °C):**
  - *Description:* The temperature of the ground surface during the night.
- **tmax (Maximum daily air temperature in °C):**
  - *Description:* Highest air temperature recorded in a day.
- **tmin (Minimum daily air temperature in °C):**
  - *Description:* Lowest air temperature recorded in a day.
- **ppt (Precipitation in mm):**
  - *Description:* Accumulated rainfall.
- **vpdmax & vpdmin (Maximum/Minimum Vapor Pressure Deficit in hPa):**
  - *Description:* Measure of atmospheric dryness (the difference between how much moisture the air can hold and how much it actually holds).
- **Evap_tavg (Average Evapotranspiration in kg/m²/s or mm/day):**
  - *Description:* Actual water loss from the soil surface (evaporation) and plant tissues (transpiration).
- **PotEvap_tavg (Average Potential Evapotranspiration in W/m² or mm/day):**
  - *Description:* The amount of evapotranspiration that would occur if sufficient water were available.

- **SoilMoi0_10cm (Soil moisture at 0-10 cm depth in kg/m^2 or mm):**
  - *Description:* Water content in the top soil.
- **SoilMoi10_40cm, SoilMoi40_100cm, SoilMoi100_200cm (Soil moisture at increasing depths):**
  - *Description:* Water content in deeper soil layers.

5.5.3.2 Target Variable (Y)

The target variable for prediction (`Y_train.npy`, `Y_test.npy`) is: * **County-Level Soybean Yield:** * *Unit:* Bushels per acre (bu/ac). * *Significance:* This is the primary measure of agricultural productivity for soybeans. Accurate and timely yield prediction is crucial for farmers, policymakers, insurers, and commodity markets for planning, risk assessment, and economic forecasting.

```
# --- Automatic Data Download and Extraction ---
import urllib.request
import zipfile
import shutil

# Download URL
dataset_url = 'https://github.com/pouuh/phenology-guided-soybean-yield/releases/download/v1.
download_dir = './soybean_dataset'
zip_path = os.path.join(download_dir, 'dataset.zip')

# Create directory if it doesn't exist
os.makedirs(download_dir, exist_ok=True)

# Check if data already exists
required_files = ['X_train.npy', 'Y_train.npy', 'X_test.npy', 'Y_test.npy']
files_exist = all(os.path.exists(os.path.join(download_dir, file)) for file in required_files

if not files_exist:
    print("Downloading soybean phenology guided dataset...")
    try:
        urllib.request.urlretrieve(dataset_url, zip_path)
        print(f" Download completed: {zip_path}")

        print("Extracting dataset...")
        with zipfile.ZipFile(zip_path, 'r') as zip_ref:
            zip_ref.extractall(download_dir)
```

```
        print("  Extraction completed")

        # Remove zip file after extraction
        os.remove(zip_path)
        print("  Cleaned up zip file")

    except Exception as e:
        print(f"  Error downloading or extracting dataset: {e}")
        raise
else:
    print("  Dataset files already exist locally")

file_folder = download_dir + '/Soybean_Yield'

# Verify data files exist before loading
print("\nVerifying data files...")
for file in required_files:
    file_path = os.path.join(file_folder, file)
    if os.path.exists(file_path):
        print(f"  Found: {file}")
    else:
        print(f"  Missing: {file}")

# Load preprocessed time series data
print("\nLoading training and test datasets...")
X_train = np.load(os.path.join(file_folder, 'X_train.npy'))
Y_train = np.load(os.path.join(file_folder, 'Y_train.npy'))
X_test = np.load(os.path.join(file_folder, 'X_test.npy'))
Y_test = np.load(os.path.join(file_folder, 'Y_test.npy'))

print(f"Training data shape: X={X_train.shape}, Y={Y_train.shape}")
print(f"Test data shape: X={X_test.shape}, Y={Y_test.shape}")
print(f"Features per timestep: {X_train.shape[2]}")
print(f"Timesteps per sample: {X_train.shape[1]}")
```

```
# --- Displaying a sample of the input data (X_train) ---
print(f"Shape of X_train: {X_train.shape}") # Expected: (num_samples, 6 timesteps, 14 featur

if X_train.shape[0] > 0:
    print("\nFirst sample from X_train (all 6 timesteps, first 3 features shown for brevity)
    # This prints the data for the first county-year, across all 6 phenological stages,
    # but only for the first 3 out of the 14 features to keep the printout manageable.
```

```
    # Adjust slicing as needed to show a meaningful snippet.
    print(X_train[0, :, :3])
else:
    print("X_train appears to be empty or not loaded correctly.")
```

5.5.4 Visualizing Input Feature Dynamics

To provide a more intuitive understanding of the input data (`X`) fed into our models, the plots below showcase the time series for three key predictors—EVI2, Maximum Temperature, and Precipitation—across the six phenological stages. These visualizations are for three randomly selected county-year samples from the training dataset and display the data in its *[Specify: 'original (pre-normalization)' OR 'normalized']* form.

Observing these plots helps to: * Illustrate the temporal patterns of different environmental variables through the defined soybean growth stages. * Appreciate the sample-to-sample variability in these predictors. * Better understand the nature of the 3D input `(samples, timesteps, features)` that the LSTM and 1D-CNN models process. Each line in a subplot represents the sequence of values for one feature across the 6 phenological stages (timesteps) for one sample.

```
data_to_sample_from = X_train
data_source_label = "Original (Pre-Normalization)" # Label for titles

# Define feature names and their indices within your 14 features.
# These MUST match the actual order in your X_train.npy features.
# Example (VERIFY AND UPDATE THESE INDICES BASED ON YOUR ACTUAL DATA):
feature_map = {
    'EVI2': 0,
    'LST_Day (K)': 1,
    'LST_Night (K)': 2,
    'Max Temperature (°C)': 3,  # Using tmax as an example temperature
    'Min Temperature (°C)': 4,
    'Precipitation (mm)': 5,
    'VPDmax (hPa)': 6,
    'VPDmin (hPa)': 7,
    'Evapotranspiration': 8,
    'Potential Evapotranspiration': 9,
    'SoilMoi0_10cm': 10,
    'SoilMoi10_40cm': 11,
    'SoilMoi40_100cm': 12,
    'SoilMoi100_200cm': 13
}
```

```
# Select features to plot (ensure these names match keys in feature_map)
features_to_plot_names = ['EVI2', 'Max Temperature (°C)', 'Precipitation (mm)']
selected_feature_indices = [feature_map[name] for name in features_to_plot_names]

num_samples_to_plot = 3
if data_to_sample_from.shape[0] >= num_samples_to_plot:
    # Set seed for reproducibility of random sample selection if desired
    # np.random.seed(42) # Optional: for consistent random samples
    random_indices = np.random.choice(data_to_sample_from.shape[0], num_samples_to_plot, repl
else:
    print(f"Warning: Not enough samples in the dataset (found {data_to_sample_from.shape[0]}
    random_indices = np.arange(min(num_samples_to_plot, data_to_sample_from.shape[0]))


if len(random_indices) > 0:
    num_stages = data_to_sample_from.shape[1] # Should be 6
    pheno_stages_labels = [f"Stage {i+1}" for i in range(num_stages)]

    fig, axes = plt.subplots(nrows=num_samples_to_plot, ncols=len(features_to_plot_names),
                             figsize=(18, 4 * num_samples_to_plot), sharex=False) # sharex=Fa

    # Adjust for single sample plotting case
    if num_samples_to_plot == 1:
        axes = np.array([axes]) # Make it 2D for consistent indexing

    fig.suptitle(f"Example Input Feature Time Series ({data_source_label} Data)", fontsize=18

    for i, sample_idx in enumerate(random_indices):
        for j, feature_idx in enumerate(selected_feature_indices):
            ax = axes[i, j]
            time_series_data = data_to_sample_from[sample_idx, :, feature_idx]

            ax.plot(pheno_stages_labels, time_series_data, marker='o', linestyle='-')
            ax.set_ylabel(features_to_plot_names[j])
            if i == 0: # Set subplot titles for the first row (feature type)
                 ax.set_title(f"{features_to_plot_names[j].split(' (')[0]}")
            if j == 0: # Set y-axis title for the first column (sample identifier)
                 ax.text(-0.3, 0.5, f"Sample {i+1}", transform=ax.transAxes,
                         rotation=90, verticalalignment='center', fontsize=14)

            ax.grid(True, linestyle='--', alpha=0.7)
            ax.tick_params(axis='x', rotation=45)
```

```python
    # Common X-label (optional, if sharex=True)
    # fig.text(0.5, 0.01, 'Phenological Stage', ha='center', va='center', fontsize=16)

    plt.tight_layout(rect=[0, 0.03, 1, 0.97]) # Adjust layout to make space for suptitle & xl
    plt.show()
else:
    print("No samples to plot.")
```

```python
# === DATA NORMALIZATION FUNCTIONS ===
def normalize_data(data, mean, std):
    """
    Normalize time series data using z-score normalization.

    This function standardizes each feature across all samples and timesteps
    using the training set statistics to prevent data leakage.

    Parameters:
    -----------
    data : numpy.ndarray
        Input data of shape (samples, timesteps, features)
    mean : numpy.ndarray
        Feature-wise mean values from training data
    std : numpy.ndarray
        Feature-wise standard deviation from training data

    Returns:
    --------
    normalized_data : numpy.ndarray
        Z-score normalized data
    """
    # Prevent division by zero for constant features
    std_safe = std.copy()
    std_safe[std_safe == 0] = 1e-8

    normalized_data = (data - mean) / std_safe
    return normalized_data


# === COMPUTE NORMALIZATION STATISTICS ===
# Calculate feature-wise statistics from training data only
print("\nCalculating normalization statistics from training data...")
```

```
X_mean = np.nanmean(X_train, axis=(0, 1))  # Mean across samples and timesteps
X_std = np.nanstd(X_train, axis=(0, 1))    # Std across samples and timesteps

print(f"Features with zero variance: {np.sum(X_std == 0)}")
print(f"Features with NaN values: {np.sum(np.isnan(X_mean))}")

# Apply normalization to both training and test sets
print("Applying z-score normalization...")
X_train_normalized = normalize_data(X_train, X_mean, X_std)
X_test_normalized = normalize_data(X_test, X_mean, X_std)

# Update variables for consistency with rest of notebook
X_train = X_train_normalized
X_test = X_test_normalized

print("Data loading and preprocessing complete!")
print(f"Training data statistics: mean={np.nanmean(X_train):.3f}, std={np.nanstd(X_train):.3
print(f"Test data statistics: mean={np.nanmean(X_test):.3f}, std={np.nanstd(X_test):.3f}")
```

5.6 Why Phenology-Guided Time Series Modeling?

Crop growth and yield formation are fundamentally governed by **phenological development**, not by fixed calendar dates. Environmental stresses such as heat, water deficit, or excessive moisture can have vastly different impacts on final yield depending on *when* they occur during the growing season. For example, stress during flowering or grain filling often has a disproportionately large effect compared to similar conditions during early vegetative growth.

Many existing yield prediction approaches rely on calendar-based temporal aggregation (e.g., weekly or monthly averages). While convenient, these fixed intervals may mix multiple developmental stages or split critical growth phases, thereby obscuring biologically meaningful signals.

In this notebook, we adopt a **phenology-guided framework**, where satellite- derived phenological transition dates are used to divide the growing season into six biologically meaningful stages. Environmental and vegetation predictors are aggregated within these adaptive stage boundaries, ensuring that the input time series aligns with actual crop development rather than arbitrary time windows. This approach improves both interpretability and the model's ability to learn stage-specific yield sensitivities.

5.7 Why Quantify Prediction Uncertainty?

Crop yield prediction is inherently uncertain due to multiple sources of variability, including weather fluctuations, observation noise in remote sensing data, spatial heterogeneity in agricultural systems, and limitations in model structure. Deterministic point predictions alone do not convey how reliable a prediction is, which limits their usefulness for decision-making, risk assessment, and interpretation under atypical conditions.

To address this limitation, all models implemented in this notebook are formulated as **probabilistic models**, producing both a mean yield estimate and an associated uncertainty. These uncertainty estimates provide additional context about prediction confidence, highlighting conditions, regions, or yield ranges where the model is less certain. This is particularly important for applications such as agricultural monitoring, insurance, and climate risk assessment, where understanding prediction reliability is as critical as the prediction itself.

5.8 2. Model Development and Training

This section implements and compares three different approaches for probabilistic crop yield prediction:

1. **Probabilistic LSTM**: Captures temporal dependencies in crop development
2. **Probabilistic 1D-CNN**: Learns local temporal patterns and seasonal cycles
3. **Gaussian Process Regression (GPR)**: Provides natural uncertainty quantification

All models output both mean predictions and uncertainty estimates, enabling robust decision-making in agricultural applications.

5.8.1 Key Implementation Features:

- **Reproducible training**: Fixed random seeds across all models
- **Early stopping**: Prevents overfitting during neural network training
- **Validation split**: 80/20 train/validation split for hyperparameter tuning
- **Probabilistic outputs**: All models provide prediction uncertainty estimates

5.8.2 2.1 Probabilistic Long Short-Term Memory (LSTM) Model

Model Rationale: LSTM networks excel at modeling sequential data with long-term dependencies, making them ideal for crop yield prediction where: - Early season conditions influence late season development - Critical growth stages occur at different times - Weather patterns show temporal autocorrelation

Architecture Details: - **Input Layer**: Accepts time series of shape (timesteps, features) - **LSTM Layer**: 100 hidden units with built-in memory mechanisms - **Dense Layers**: 100-unit ReLU layer followed by 2-unit output - **Probabilistic Output**: Learns both mean ( ) and uncertainty ( ) parameters

Training Configuration: - **Loss Function**: Negative Log-Likelihood (NLL) for probabilistic training - **Optimizer**: Adam with learning rate 1e-4 - **Batch Size**: 64 samples - **Early Stopping**: Monitors validation loss with patience=10

```
# === PROBABILISTIC LSTM MODEL IMPLEMENTATION ===

# Split training data for validation
print("Creating train/validation split...")
X_train_split, X_val, Y_train_split, Y_val = train_test_split(
    X_train, Y_train,
    test_size=0.2,
    random_state=42,
    shuffle=True
)
print(f"Train samples: {X_train_split.shape[0]}, Validation samples: {X_val.shape[0]}")

# === PROBABILISTIC DISTRIBUTION SETUP ===

tfd = tfp.distributions

def normal_sp(params):
    """
    Create a Normal distribution from neural network outputs.

    This function transforms raw network outputs into a well-behaved
    probability distribution with learnable mean and variance.

    Parameters:
    -----------
    params : tf.Tensor
        Network output tensor with shape (batch_size, 2)
        First column: raw mean values
        Second column: raw scale parameters

    Returns:
    --------
    distribution : tfp.distributions.Normal
        Parameterized Normal distribution
```

```
    """
    return tfd.Normal(
        loc=params[:, 0:1],  # Mean parameter (no transformation)
        scale=1e-3 + tf.math.softplus(0.05 * params[:, 1:2])  # Ensure positive scale
    )

class ProbabilisticLayer(tf.keras.layers.Layer):
    """
    Custom Keras layer that outputs distribution parameters.

    This layer takes the raw outputs from the dense layer and
    formats them for probabilistic prediction.
    """
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def call(self, inputs):
        """Transform inputs into distribution parameters."""
        dist = normal_sp(inputs)
        return tf.concat([dist.loc, dist.scale], axis=-1)

def NLL(y_true, y_pred):
    """
    Negative Log-Likelihood loss for probabilistic training.

    This loss function encourages the model to:
    1. Make accurate mean predictions
    2. Provide appropriate uncertainty estimates
    3. Avoid overconfident or underconfident predictions

    Parameters:
    -----------
    y_true : tf.Tensor
        True target values
    y_pred : tf.Tensor
        Predicted distribution parameters [mean, scale]

    Returns:
    --------
    nll : tf.Tensor
        Negative log-likelihood loss value
    """
```

```
    dist = tfd.Normal(
        loc=y_pred[:, 0:1],
        scale=y_pred[:, 1:2]
    )
    return -tf.reduce_mean(dist.log_prob(y_true))

# === REPRODUCIBILITY SETUP ===

def set_seeds(seed_value=32):
    """Set all random seeds for reproducible results."""
    os.environ['PYTHONHASHSEED'] = str(seed_value)
    random.seed(seed_value)
    np.random.seed(seed_value)
    tf.random.set_seed(seed_value)

def create_probabilistic_lstm(input_shape, lstm_units=100, dense_units=100):
    """
    Create a probabilistic LSTM model for yield prediction.

    Parameters:
    -----------
    input_shape : tuple
        Shape of input time series (timesteps, features)
    lstm_units : int
        Number of LSTM hidden units
    dense_units : int
        Number of dense layer units

    Returns:
    --------
    model : tf.keras.Model
        Compiled probabilistic LSTM model
    """
    model = Sequential([
        Input(shape=input_shape),
        LSTM(lstm_units, name='lstm_layer'),
        Dense(dense_units, activation='relu', name='dense_hidden'),
        Dense(2, name='distribution_params'),  # Output: [mean, scale]
        ProbabilisticLayer(name='probabilistic_output')
    ])

    model.compile(
```

```
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
        loss=NLL,
        metrics=['mse']  # Additional metric for monitoring
    )

    return model

def train_and_evaluate_lstm(X_train_split, Y_train_split, X_val, Y_val, X_test, Y_test):
    """
    Complete training and evaluation pipeline for LSTM model.

    Parameters:
    -----------
    X_train, Y_train : numpy.ndarray
        Training data and labels
    X_val, Y_val : numpy.ndarray
        Validation data and labels
    X_test, Y_test : numpy.ndarray
        Test data and labels


    Returns:
    --------
    model : tf.keras.Model
        Trained model
    mean : numpy.ndarray
        Mean predictions for test data
    uncertainty : numpy.ndarray
        Uncertainty estimates for test data
    history : tf.keras.callbacks.History
        Training history
    """

    # Set seeds for reproducibility
    set_seeds()

    # Create model
    model = create_probabilistic_lstm(input_shape=(X_train_split.shape[1], X_train_split.shap

    # Define callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
```

```
        patience=10,
        restore_best_weights=True,
        mode='min'
    )

    # Train the model
    history = model.fit(
        X_train_split, Y_train_split,
        epochs=1000,
        batch_size=64,
        validation_data=(X_val, Y_val),
        callbacks=[early_stopping],
        verbose=1
    )

    # Get predictions
    predictions = model.predict(X_test)
    means = predictions[:, 0]
    uncertainties = predictions[:, 1]

    # Calculate metrics
    final_r2 = r2_score(Y_test, means)
    final_rmse = np.sqrt(mean_squared_error(Y_test, means))

    print(f"\nFinal Test Metrics:")
    print(f"R2 Score: {final_r2:.4f}")
    print(f"RMSE: {final_rmse:.4f}")
    print(f"Average Uncertainty: {np.mean(uncertainties):.4f}")

    return model, means, uncertainties, history

# Execute LSTM training and evaluation
lstm_model, lstm_means, lstm_uncertainties, lstm_history = train_and_evaluate_lstm(
    X_train_split, Y_train_split,
    X_val, Y_val,
    X_test, Y_test
)
```

```
# --- Plotting Training and Validation Loss (Full and Zoomed Views) ---

# Ensure 'history' is the Keras History object from model.fit()
```

```python
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 6)) # Adjust figsize as needed

history = lstm_history

# Subplot 1: Full Training and Validation Loss Curve
axes[0].plot(history.history['loss'], label='Training NLL')
axes[0].plot(history.history['val_loss'], label='Validation NLL')
axes[0].set_title('Full Training & Validation Loss Curve')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('NLL (Negative Log-Likelihood)')
axes[0].legend()
axes[0].grid(True, linestyle='--', alpha=0.6)

# Subplot 2: Zoomed-in View of Training and Validation Loss
start_epoch_for_zoom = 200  # Keep this or adjust if another start point is better
# Ensure there are enough epochs to make a zoomed plot meaningful
if len(history.history['loss']) > start_epoch_for_zoom:
    # X-axis range for the zoomed plot (actual epoch numbers)
    epoch_range_zoomed = range(start_epoch_for_zoom, len(history.history['loss']))

    axes[1].plot(epoch_range_zoomed,
                 history.history['loss'][start_epoch_for_zoom:],
                 label='Training NLL (Zoomed)')
    axes[1].plot(epoch_range_zoomed,
                 history.history['val_loss'][start_epoch_for_zoom:],
                 label='Validation NLL (Zoomed)')
    axes[1].set_title(f'Zoomed Loss (Epochs {start_epoch_for_zoom}+)')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('NLL (Negative Log-Likelihood)')
    axes[1].legend()
    axes[1].grid(True, linestyle='--', alpha=0.6)
else:
    # Fallback if not enough epochs for the defined zoom
    axes[1].text(0.5, 0.5, 'Not enough epochs for zoomed view.',
                 horizontalalignment='center', verticalalignment='center',
                 transform=axes[1].transAxes)
    axes[1].set_title(f'Zoomed View (Epochs {start_epoch_for_zoom}+)')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('NLL')


plt.tight_layout()
```

```
plt.show()
```

```
def evaluate_model(model, X_test, Y_test, model_name="Model"):
    """
    Comprehensive evaluation of a probabilistic model.

    Parameters:
    -----------
    model : tf.keras.Model or sklearn model
        Trained probabilistic model
    X_test, Y_test : numpy.ndarray
        Test data and labels
    model_name : str
        Name of the model for reporting

    Returns:
    --------
    predictions : numpy.ndarray
        Model predictions [mean, uncertainty]
    """
    predictions = model.predict(X_test)
    mean_pred = predictions[:, 0]
    std_pred = predictions[:, 1]

    # Calculate metrics
    final_r2 = r2_score(Y_test, mean_pred)
    final_rmse = np.sqrt(mean_squared_error(Y_test, mean_pred))

    print(f"\nFinal Test Metrics:")
    print(f"R2 Score: {final_r2:.4f}")
    print(f"RMSE: {final_rmse:.4f}")
    print(f"Mean Prediction Uncertainty: {np.mean(std_pred):.4f}")
    print(f"Min Uncertainty: {np.min(std_pred):.4f}")
    print(f"Max Uncertainty: {np.max(std_pred):.4f}")

    # Calculate prediction interval coverage
    within_interval = np.abs(Y_test - mean_pred) <= 2 * std_pred
    coverage = np.mean(within_interval) * 100
    print(f"95% CI Coverage: {coverage:.2f}%")

    return predictions
```

```python
def plot_predictions_with_uncertainty(Y_test, predictions, title="Predictions with 95% Confid
    """
    Create publication-quality prediction vs truth plot with uncertainty visualization.

    Parameters:
    -----------
    Y_test : numpy.ndarray
        True target values
    predictions : numpy.ndarray
        Model predictions [mean, uncertainty]
    """
    mean_pred = predictions[:, 0]
    std_pred = predictions[:, 1]

    plt.figure(figsize=(8, 8))

    # Calculate overall min and max for consistent axes
    all_values = np.concatenate([Y_test.flatten(), mean_pred.flatten()])
    min_val = all_values.min()
    max_val = all_values.max()

    # Add some padding to the limits
    padding = (max_val - min_val) * 0.05
    axis_min = min_val - padding
    axis_max = max_val + padding

    # Sort for proper uncertainty band plotting
    sort_idx = np.argsort(Y_test.flatten())
    Y_test_sorted = Y_test.flatten()[sort_idx]
    mean_pred_sorted = mean_pred.flatten()[sort_idx]
    std_pred_sorted = std_pred.flatten()[sort_idx]

    # Plot uncertainty bands
    plt.fill_between(Y_test_sorted,
                     mean_pred_sorted - 2*std_pred_sorted,
                     mean_pred_sorted + 2*std_pred_sorted,
                     alpha=0.2, color='coral', label='95% CI')

    # Plot predictions
    plt.scatter(Y_test, mean_pred, alpha=0.5, color='cornflowerblue', label='Predictions')

    # Plot diagonal line
```

```
    plt.plot([Y_test.min(), Y_test.max()],
             [Y_test.min(), Y_test.max()],
             'r--', label='Perfect Prediction')

    # Calculate slope using linear regression
    lr = LinearRegression()
    lr.fit(Y_test.reshape(-1, 1), mean_pred)
    slope = lr.coef_[0]
    intercept = lr.intercept_
    r2 = r2_score(Y_test, mean_pred)

    # Plot fitted regression line
    plt.plot([axis_min, axis_max],
             [slope * axis_min + intercept, slope * axis_max + intercept],
             'g-', alpha=0.8, linewidth=2, label=f'Fitted Line (slope={slope:.2f})')

    # Set consistent square axes limits
    plt.xlim(axis_min, axis_max)
    plt.ylim(axis_min, axis_max)
    plt.gca().set_aspect('equal', adjustable='box')  # Make the plot square

    plt.xlabel('True Vaules (bu/ac)', fontsize=18)
    plt.ylabel('Predictions (bu/ac)', fontsize=18)
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.title(title, fontsize=20)
    plt.legend(fontsize=16, loc='upper left',frameon=False)
    # plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

# Evaluate LSTM model
predictions = evaluate_model(lstm_model, X_test, Y_test, model_name="LSTM")

# Visualize LSTM results
plot_predictions_with_uncertainty(Y_test, predictions)
```

```
# === MODEL PERSISTENCE ===

# Save the trained LSTM model for future use
model_save_path = os.path.join(file_folder, 'model_lstm.h5')
lstm_model.save(model_save_path)
```

```
print(f"LSTM model saved to: {model_save_path}")
print("Model can be loaded later using:")
print("  model = tf.keras.models.load_model(path, custom_objects={'ProbabilisticLayer': Proba
```

5.8.3 2.2 Probabilistic 1D Convolutional Neural Network (CNN) Model

Model Rationale: 1D CNNs are particularly effective for time series analysis because they: - Detect local temporal patterns and seasonal cycles - Require fewer parameters than LSTM networks - Process data more efficiently for shorter sequences - Capture translation-invariant patterns across the growing season

Architecture Design: - **Multi-scale Feature Extraction**: Three convolutional blocks with different receptive fields - **Hierarchical Learning**: Filters progress from 32→32→16 to learn features at multiple scales - **Pooling Strategy**: MaxPooling reduces temporal dimension while preserving important features - **Probabilistic Head**: Same as LSTM - outputs mean and uncertainty estimates

Key Differences from LSTM: - **Computational Efficiency**: Faster training and inference - **Local Pattern Focus**: Better at detecting short-term temporal signatures - **Parameter Efficiency**: Fewer trainable parameters than equivalent LSTM - **Parallel Processing**: Convolutions can be parallelized more effectively

```
# === PROBABILISTIC 1D-CNN MODEL IMPLEMENTATION ===

def create_probabilistic_cnn(input_shape, filters=[32, 32, 16], kernel_size=3, dense_units=1
    """
    Create a probabilistic 1D-CNN model for yield prediction.

    Parameters:
    -----------
    input_shape : tuple
        Shape of input time series (timesteps, features)
    filters : list
        Number of filters for each convolutional layer
    kernel_size : int
        Size of convolutional kernels
    dense_units : int
        Number of dense layer units

    Returns:
    --------
    model : tf.keras.Model
```

```
        Compiled probabilistic CNN model
    """
    model = Sequential([
        Input(shape=input_shape),

        # First convolutional block - captures fine-grained patterns
        Conv1D(filters=filters[0], kernel_size=kernel_size,
               activation='relu', padding='same', name='conv1d_1'),
        MaxPooling1D(pool_size=2, name='maxpool_1'),

        # Second convolutional block - learns mid-level features
        Conv1D(filters=filters[1], kernel_size=kernel_size,
               activation='relu', padding='same', name='conv1d_2'),
        MaxPooling1D(pool_size=2, name='maxpool_2'),

        # Third convolutional block - high-level temporal abstractions
        Conv1D(filters=filters[2], kernel_size=kernel_size,
               activation='relu', padding='same', name='conv1d_3'),

        # Flatten and dense layers
        Flatten(name='flatten'),
        Dense(dense_units, activation='relu', name='dense_hidden'),
        Dense(2, name='distribution_params'),
        ProbabilisticLayer(name='probabilistic_output')
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
        loss=NLL,
        metrics=['mse']
    )

    return model

def train_and_evaluate_cnn(X_train, Y_train, X_val, Y_val, X_test, Y_test):
    """
    Complete training and evaluation pipeline for CNN model.
    """
    set_seeds(132)

    print("Creating 1D-CNN model architecture...")
    model = create_probabilistic_cnn(
```

```
        input_shape=(X_train.shape[1], X_train.shape[2])
    )
    model.summary()

    # Define callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        mode='min',
        verbose=1
    )

    print("\nStarting CNN training...")
    history = model.fit(
        X_train, Y_train,
        epochs=1000,
        batch_size=64,
        validation_data=(X_val, Y_val),
        callbacks=[early_stopping],
        verbose=1
    )

    print("\nEvaluating CNN on test set...")
    predictions = model.predict(X_test, verbose=0)
    mean_pred = predictions[:, 0]
    std_pred = predictions[:, 1]

    # Calculate performance metrics
    r2 = r2_score(Y_test, mean_pred)
    rmse = np.sqrt(mean_squared_error(Y_test, mean_pred))
    mae = np.mean(np.abs(Y_test.flatten() - mean_pred))

    print(f"\n=== CNN MODEL PERFORMANCE ===")
    print(f"R² Score: {r2:.4f}")
    print(f"RMSE: {rmse:.4f} bu/acre")
    print(f"MAE: {mae:.4f} bu/acre")
    print(f"Average Uncertainty: {np.mean(std_pred):.4f} bu/acre")
    print(f"Training completed in {len(history.history['loss'])} epochs")

    return model, predictions, history
```

```
# Execute CNN training and evaluation
cnn_model, cnn_predictions, cnn_history = train_and_evaluate_cnn(
    X_train_split, Y_train_split,
    X_val, Y_val,
    X_test, Y_test
)
```

```
# Save the trained CNN model
cnn_save_path = os.path.join(file_folder, 'model_cnn.h5')
cnn_model.save(cnn_save_path)
print(f"CNN model saved to: {cnn_save_path}")
```

```
# --- Plotting Training and Validation Loss (Full and Zoomed Views) ---

# Ensure 'history' is the Keras History object from model.fit()

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 6)) # Adjust figsize as needed

history = cnn_history

# Subplot 1: Full Training and Validation Loss Curve
axes[0].plot(history.history['loss'], label='Training NLL')
axes[0].plot(history.history['val_loss'], label='Validation NLL')
axes[0].set_title('Full Training & Validation Loss Curve')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('NLL (Negative Log-Likelihood)')
axes[0].legend()
axes[0].grid(True, linestyle='--', alpha=0.6)

# Subplot 2: Zoomed-in View of Training and Validation Loss
start_epoch_for_zoom = 200  # Keep this or adjust if another start point is better
# Ensure there are enough epochs to make a zoomed plot meaningful
if len(history.history['loss']) > start_epoch_for_zoom:
    # X-axis range for the zoomed plot (actual epoch numbers)
    epoch_range_zoomed = range(start_epoch_for_zoom, len(history.history['loss']))

    axes[1].plot(epoch_range_zoomed,
                 history.history['loss'][start_epoch_for_zoom:],
                 label='Training NLL (Zoomed)')
    axes[1].plot(epoch_range_zoomed,
                 history.history['val_loss'][start_epoch_for_zoom:],
                 label='Validation NLL (Zoomed)')
```

```
    axes[1].set_title(f'Zoomed Loss (Epochs {start_epoch_for_zoom}+)')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('NLL (Negative Log-Likelihood)')
    axes[1].legend()
    axes[1].grid(True, linestyle='--', alpha=0.6)
else:
    # Fallback if not enough epochs for the defined zoom
    axes[1].text(0.5, 0.5, 'Not enough epochs for zoomed view.',
                 horizontalalignment='center', verticalalignment='center',
                 transform=axes[1].transAxes)
    axes[1].set_title(f'Zoomed View (Epochs {start_epoch_for_zoom}+)')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('NLL')


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

```
# Evaluate CNN model
predictions = evaluate_model(cnn_model, X_test, Y_test, model_name="CNN")

# Visualize CNN results
plot_predictions_with_uncertainty(Y_test, predictions)
```

5.8.4 2.3 Gaussian Process Regression (GPR) Model

Model Rationale: Gaussian Processes provide a Bayesian approach to regression that: - Naturally quantifies uncertainty without architectural modifications - Works well with limited training data - Provides theoretical guarantees about uncertainty calibration - Serves as a strong baseline for comparison with neural networks

Key Advantages: - **Natural Uncertainty**: Uncertainty estimates are theoretically grounded - **Non-parametric**: Flexible model that adapts to data complexity - **Interpretable**: Kernel functions have clear mathematical meaning - **Calibrated**: Uncertainty estimates are typically well-calibrated

Implementation Notes: - **Computational Complexity**: $O(n^3)$ scaling requires subsampling for large datasets - **Kernel Choice**: RBF + White noise kernel for smooth trends with observation noise - **Hyperparameter Optimization**: Automatic optimization of kernel parameters - **Data Preprocessing**: Requires flattening of time series for traditional ML interface

```
### Gaussian Process Regression (GPR)
# Set random seed for reproducibility
seed_value=11
np.random.seed(seed_value)
random.seed(seed_value)

# Flatten the time series data for GPR
def flatten_time_series_data(X):
    """Flatten time series data for traditional ML models"""
    n_samples, n_timesteps, n_features = X.shape
    return X.reshape(n_samples, n_timesteps * n_features)

# Flatten training and test data
X_train_flat = flatten_time_series_data(X_train_split)
X_val_flat = flatten_time_series_data(X_val)
X_test_flat = flatten_time_series_data(X_test)

print(f"Original shape: {X_train_split.shape}")
print(f"Flattened shape: {X_train_flat.shape}")

# Due to computational constraints, we'll use a subset for GPR training
# GPR scales as O(n³) so we need to limit the training size
max_samples = 2000  # Adjust based on your computational resources
if X_train_flat.shape[0] > max_samples:
    print(f"Using subset of {max_samples} samples for GPR training")
    indices = np.random.choice(X_train_flat.shape[0], max_samples, replace=False)
    X_train_gpr = X_train_flat[indices]
    Y_train_gpr = Y_train_split[indices].ravel()
else:
    X_train_gpr = X_train_flat
    Y_train_gpr = Y_train_split.ravel()

# Define the kernel
# We use a combination of RBF (for smooth variations) and WhiteKernel (for noise)
kernel = C(1.0, (1e-3, 1e3)) * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2)) + Whit

print("Creating Gaussian Process Regressor...")
gpr_model = GaussianProcessRegressor(
    kernel=kernel,
    alpha=1e-6,  # Small regularization parameter
    normalize_y=True,  # Normalize target values
    n_restarts_optimizer=5,  # Number of restarts for optimization
```

```
    random_state=seed_value,  # Set random seed for reproducibility
    # verbose=1  # Verbosity level
)

# Train the GPR model
print("Training Gaussian Process Regressor...")
start_time = time.time()
gpr_model.fit(X_train_gpr, Y_train_gpr)
training_time = time.time() - start_time
print(f"Training completed in {training_time:.2f} seconds")

print(f"Optimized kernel: {gpr_model.kernel_}")
print(f"Log-marginal-likelihood: {gpr_model.log_marginal_likelihood_value_:.3f}")
```

```
# Make predictions with uncertainty
print("Making predictions...")

# Predictions on training set (subset)
gpr_pred_train, gpr_std_train = gpr_model.predict(X_train_gpr, return_std=True)

# Predictions on validation set
gpr_pred_val, gpr_std_val = gpr_model.predict(X_val_flat, return_std=True)

# Predictions on test set
gpr_pred_test, gpr_std_test = gpr_model.predict(X_test_flat, return_std=True)

# Calculate metrics
train_r2 = r2_score(Y_train_gpr, gpr_pred_train)
train_rmse = np.sqrt(mean_squared_error(Y_train_gpr, gpr_pred_train))

val_r2 = r2_score(Y_val, gpr_pred_val)
val_rmse = np.sqrt(mean_squared_error(Y_val, gpr_pred_val))

test_r2 = r2_score(Y_test, gpr_pred_test)
test_rmse = np.sqrt(mean_squared_error(Y_test, gpr_pred_test))

# Print results
print(f"\nGaussian Process Regression Model Performance:")
print(f"Test - R²: {test_r2:.4f}, RMSE: {test_rmse:.4f}")
print(f"Average Test Uncertainty: {np.mean(gpr_std_test):.4f}")

# Calculate prediction interval coverage
```

```
within_interval = np.abs(Y_test.flatten() - gpr_pred_test) <= 2 * gpr_std_test
coverage = np.mean(within_interval) * 100
print(f"95% CI Coverage: {coverage:.2f}%")

# # Create scatter plot for GPR predictions
plot_predictions_with_uncertainty(Y_test, np.array([gpr_pred_test, gpr_std_test]).T)
```

5.9 3. Advanced Analysis and Interpretation

This section provides deeper insights into model behavior and practical applications:

5.9.1 3.1 Multi-Year Performance Analysis

- **Temporal Generalization**: How well models perform across different years
- **Scatter Plot Analysis**: Visual assessment of prediction quality by year
- **Annual Yield Trends**: Comparison of predicted vs actual yield patterns

5.9.2 3.2 Feature and Temporal Importance

- **Permutation Importance**: Identifies most influential environmental variables
- **Phenological Analysis**: Determines critical growth stages for yield prediction
- **Agricultural Insights**: Translates model behavior into agronomic understanding

5.9.3 3.3 Spatial-Temporal Visualization

- **Geographic Mapping**: County-level prediction and uncertainty visualization
- **Uncertainty Patterns**: Spatial distribution of model confidence
- **Temporal Consistency**: Year-to-year prediction stability

5.9.4 3.1 Multi-Year Model Performance Analysis

This section evaluates how well our trained models generalize across different growing seasons and environmental conditions. By testing on individual years (2018-2021), we can:

- **Assess Temporal Robustness**: How consistent are predictions across different weather years?
- **Identify Model Weaknesses**: Which years or conditions challenge each model most?

Key Evaluation Aspects: - **Year-by-Year Performance**: R^2 and RMSE metrics for each test year - **Visual Pattern Analysis**: Scatter plots reveal model behavior and outliers

```
def plot_prediction_scatter(all_years_data, figsize=(20, 20)):
    """
    Create clean scatter plots of Predicted vs True Yield for each year with statistics
    """
    fig, axes = plt.subplots(2, 2, figsize=figsize)
    axes = axes.flatten()

    # Set font sizes
    plt.rcParams.update({
    'font.size': 28,           # Increased from 20
    'axes.labelsize': 30,      # Increased from 22
    'axes.titlesize': 32,      # Increased from 24
    'xtick.labelsize': 32,     # Increased from 20
    'ytick.labelsize': 32,     # Increased from 20
    })


    # Get overall min and max for consistent axes
    all_true = np.concatenate([data[0] for data in all_years_data.values()])
    all_pred = np.concatenate([data[1] for data in all_years_data.values()])
    min_val = min(all_true.min(), all_pred.min())
    max_val = max(all_true.max(), all_pred.max())

    for idx, year in enumerate(range(2018, 2022)):
        true_yield, pred_yield = all_years_data[year]
        ax = axes[idx]

        # Create scatter plot
        ax.scatter(true_yield, pred_yield,
                  alpha=0.6,
                  c='cornflowerblue',
                  s=100)

        # Calculate statistics
        r2 = r2_score(true_yield, pred_yield)
        rmse = np.sqrt(mean_squared_error(true_yield, pred_yield))

        # Add perfect prediction line
        ax.plot([min_val, max_val], [min_val, max_val],
                'r--', alpha=0.8)
```

```
        # Add labels and title
        ax.set_xlabel('True Yield (bu/ac)')
        ax.set_ylabel('Predicted Yield (bu/ac)')
        ax.set_title(f'Year {year}')

        # Set consistent axes limits
        ax.set_xlim(min_val, max_val)
        ax.set_ylim(min_val, max_val)

        # Add grid but make it lighter
        ax.grid(True, linestyle='--', alpha=0.3)

        # Remove frame
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)

        # Add statistics text box without frame
        stats_text = f'RMSE: {rmse:.3f} bu/acre\nR²: {r2:.3f}'
        ax.text(0.05, 0.95, stats_text,
                transform=ax.transAxes,
                verticalalignment='top',
                fontsize=26,
                bbox=dict(facecolor='white',
                         edgecolor='none',
                         alpha=0.8,
                         pad=0.5))

    plt.tight_layout()
    return fig, axes
```

```
# load the lstm model again
model = create_probabilistic_lstm(input_shape=(X_train_split.shape[1], X_train_split.shape[2
model.summary()
with tf.keras.utils.custom_object_scope({
    'ProbabilisticLayer': ProbabilisticLayer,
    'NLL': NLL
}):
    model = load_model(os.path.join(file_folder, 'model_lstm.h5'))

# Load the county data
save_path = file_folder + '/'
```

```
df_all = pd.read_csv(save_path + '/Yield_2018_2021.csv')

# Collect data for all years
all_years_data = {}

for year in range(2018, 2022):
    # Load and process X data
    X_test = np.load(save_path + str(year) + '_X.npy')
    X_test = normalize_data(X_test, X_mean, X_std)

    # Get predictions
    predictions = model.predict(X_test)
    pred_yield = predictions[:, 0]

    # Get actual values
    df_year = df_all[df_all['Year'] == year]
    true_yield = df_year['Yield'].values

    # Store data
    all_years_data[year] = (true_yield, pred_yield)

# Create scatter plots
fig, axes = plot_prediction_scatter(all_years_data)
plt.show()
```

```
# load the CNN model again
model = create_probabilistic_cnn(input_shape=(X_train_split.shape[1], X_train_split.shape[2]
model.summary()
with tf.keras.utils.custom_object_scope({
    'ProbabilisticLayer': ProbabilisticLayer,
    'NLL': NLL
}):
    model = load_model(os.path.join(file_folder, 'model_cnn.h5'))

# Load the county data
save_path = file_folder + '/'
df_all = pd.read_csv(save_path + '/Yield_2018_2021.csv')

# Collect data for all years
all_years_data = {}

for year in range(2018, 2022):
```

```
    # Load and process X data
    X_test = np.load(save_path + str(year) + '_X.npy')
    X_test = normalize_data(X_test, X_mean, X_std)

    # Get predictions
    predictions = model.predict(X_test)
    pred_yield = predictions[:, 0]

    # Get actual values
    df_year = df_all[df_all['Year'] == year]
    true_yield = df_year['Yield'].values

    # Store data
    all_years_data[year] = (true_yield, pred_yield)

# Create scatter plots
fig, axes = plot_prediction_scatter(all_years_data)
plt.show()
```

5.9.5 3.2 Feature Importance Analysis

Understanding which environmental variables most strongly influence crop yield predictions is crucial for:

Model Interpretation: - **Validate Agricultural Knowledge**: Confirm model learns biologically meaningful patterns - **Identify Model Biases**: Detect if model relies on spurious correlations - **Guide Feature Engineering**: Inform future data collection and preprocessing strategies

Scientific Understanding: - **Quantify Variable Importance**: Rank environmental factors by prediction contribution - **Regional Adaptation**: Understand which factors matter most in specific growing regions - **Climate Change Impacts**: Assess sensitivity to variables likely to change

Methodology:

We use **permutation importance** which measures how much model performance degrades when each feature is randomly shuffled, breaking its relationship with the target while preserving its distribution.

How the `permutation_importance` function works:

1. **Establish Baseline Performance:**

- First, the function calculates a `baseline_score` using a chosen evaluation `metric` (e.g., R^2 score) on the original, unaltered test dataset (`X`, `y`). This score represents the model's performance when all features have their true values.

2. **Iterate Through Features:**

   - The function then iterates through each input feature one by one (e.g., EVI2, LST_Day, specific soil moisture layers, etc.).

3. **Permute Feature Values:**

   - For the current feature being evaluated, its values are randomly shuffled across *all samples and all timesteps* in a *copy* of the test dataset (`X_permuted`). This shuffling breaks the relationship between that feature and the target variable (`y`), effectively making the feature noisy or uninformative while keeping other features intact.

4. **Evaluate Performance with Permuted Feature:**

   - The model then makes predictions using this `X_permuted` dataset (where one feature is shuffled).
   - A new performance score (`permuted_score`) is calculated using the same metric.

5. **Calculate Importance Score:**

   - The importance of the feature for that single permutation run is the difference between the `baseline_score` and the `permuted_score` (i.e., `baseline_score - permuted_score`).
   - A significant drop in performance (a high positive importance score) after permuting a feature suggests that the model relied heavily on that feature for making accurate predictions. If the score does not change much, the feature is considered less important.

6. **Repeat for Robustness:**

   - Steps 3-5 are repeated `n_repeats` times (e.g., 10 times by default) for each feature, with a new random permutation each time.
   - The average of these repeated importance scores is then taken as the final importance score for that feature. This averaging helps to provide a more stable and reliable estimate of feature importance.

The output is a list of importance scores, corresponding to each feature, indicating their relative contributions to the model's predictive power on the given dataset and metric.

```
# load the whole data again
file_folder = download_dir + '/Soybean_Yield'
X_train = np.load(os.path.join(file_folder, 'X_train.npy'))
Y_train = np.load(os.path.join(file_folder, 'Y_train.npy'))
```

```
X_test = np.load(os.path.join(file_folder, 'X_test.npy'))
Y_test = np.load(os.path.join(file_folder, 'Y_test.npy'))

# Calculate mean and standard deviation
X_mean = np.nanmean(X_train, axis=(0, 1))
X_std = np.nanstd(X_train, axis=(0, 1))

# Normalize the data
X_train = normalize_data(X_train, X_mean, X_std)
X_test = normalize_data(X_test, X_mean, X_std)

# load the LSTM model
model = create_probabilistic_lstm(input_shape=(X_train_split.shape[1], X_train_split.shape[2
model.summary()

with tf.keras.utils.custom_object_scope({
    'ProbabilisticLayer': ProbabilisticLayer,
    'NLL': NLL
}):
    model = load_model(os.path.join(file_folder, 'model_lstm.h5'))
```

```
# Set professional plotting parameters
plt.rcParams.update({
    'font.family': 'serif',
    'font.serif': ['Times New Roman', 'DejaVu Serif'],
    'font.size': 10,
    'axes.labelsize': 11,
    'axes.titlesize': 12,
    'xtick.labelsize': 9,
    'ytick.labelsize': 9,
    'legend.fontsize': 9,
    'legend.title_fontsize': 10,
    'figure.dpi': 300,
    'savefig.dpi': 300,
    'savefig.bbox': 'tight',
    'savefig.pad_inches': 0.1
})

def permutation_importance(model, X, y, metric, feature_names, n_repeats=10):
    """
    Computes feature importance and its stability (standard deviation)
    using the permutation importance method for time-series data.
```

```
    The importance of a feature is determined by measuring the decrease in a model's
    performance score when that feature's values are randomly shuffled. A larger drop
    in score indicates higher importance. This process is repeated multiple times
    (n_repeats) for each feature to get a mean importance and its standard deviation.

    Parameters:
    ----------
    model : keras.Model
        The trained Keras model. Assumes model.predict(X)[:, 0] gives mean predictions.
    X : numpy.ndarray
        Input data (samples, timesteps, features).
    y : numpy.ndarray
        True target values.
    metric : function
        Performance metric function (e.g., r2_score).
    feature_names : list of str
        Names of the features.
    n_repeats : int, optional
        Number of times to repeat permutation for each feature. Default is 10.

    Returns:
    -------
    feature_names : list of str
        The original list of feature names.
    mean_importances : list of float
        Mean importance score for each feature.
    std_importances : list of float
        Standard deviation of importance scores for each feature across repeats.
    """
    # 1. Calculate the baseline performance score
    baseline_score = metric(y, model.predict(X)[:, 0])

    mean_importances = [] # To store mean importance score for each feature
    std_importances = []  # To store std dev of importance scores for each feature

    # 2. Iterate over each feature
    for feature_idx in range(X.shape[2]):
        scores_for_current_feature = [] # Scores from multiple permutations for this feature

        # 3. Repeat permutation n_repeats times for the current feature
        for _ in range(n_repeats):
            X_permuted = X.copy()
```

```
            # 4. Permute the current feature across all samples and timesteps
            permuted_values = np.random.permutation(X_permuted[:, :, feature_idx].flatten())

            # 5. Reshape and assign permuted values
            X_permuted[:, :, feature_idx] = permuted_values.reshape(X.shape[0], X.shape[1])

            # 6. Calculate score with permuted feature
            permuted_score = metric(y, model.predict(X_permuted)[:, 0])

            # 7. Importance is the drop in score
            scores_for_current_feature.append(baseline_score - permuted_score)

        # 8. Calculate mean and std dev of importances from n_repeats
        mean_importances.append(np.mean(scores_for_current_feature))
        std_importances.append(np.std(scores_for_current_feature))

    return feature_names, mean_importances, std_importances

# Define feature names (as in your provided code)
# Ensure this list matches the order and number of features in X_test correctly
non_soil_vars = ['EVI2','LST_Day','LST_Night','tmax',
                 'tmin','ppt', 'vpdmax','vpdmin','Evap_tavg',
                 'PotEvap_tavg','SoilMoi0_10cm', 'SoilMoi10_40cm',
                 'SoilMoi40_100cm','SoilMoi100_200cm']

# Calculate importance scores and standard deviations
# Assuming 'model', 'X_test', 'Y_test', 'r2_score' are defined
# This function now returns feature_names as the first element
# to maintain correspondence if you pass them in.
ordered_feature_names, mean_importance_scores, std_importance_scores = permutation_importanc
    model, X_test, Y_test, r2_score, non_soil_vars, n_repeats=10
)

# Combine for sorting based on mean importance scores
results_with_std = list(zip(ordered_feature_names, mean_importance_scores, std_importance_sc
# Sort by mean importance score in descending order
results_with_std.sort(key=lambda x: x[1], reverse=True)

print("\nFeature Importance Rankings (Mean +/- Std Dev):")
for feature, score, std_dev in results_with_std:
    print(f"{feature:<30} importance: {score:.4f} +/- {std_dev:.4f}")
```

```python
# Unzip for plotting (features will be in sorted order)
sorted_features, sorted_scores, sorted_stds = zip(*results_with_std)

# Create a bar plot with error bars
plt.figure(figsize=(14, 7)) # Adjusted figsize for better readability with error bars
bars = plt.bar(sorted_features, sorted_scores, yerr=sorted_stds,
               capsize=4, alpha=0.8, color='cornflowerblue', ecolor='black') # Added yerr and

plt.xticks(rotation=45, ha='right', fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel('Features', fontsize=16)
plt.ylabel('Importance Score (Decrease in R²)', fontsize=16) # Clarified y-axis
plt.title('Feature Importance Based on Permutation Test (with Stability)', fontsize=18)
plt.grid(True, axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

5.9.6 3.3 Phenological (Temporal) Importance Analysis

Understanding **when** during the growing season environmental conditions most strongly influence final yield is crucial for:

Scientific Understanding: - **Phenological Insights**: Validate known critical periods (flowering, grain filling) - **Climate Sensitivity**: Understand when crops are most vulnerable to weather stress - **Yield Formation**: Quantify relative importance of different developmental stages

Model Interpretation: - **Temporal Attention**: Understand which time periods the model focuses on - **Seasonal Patterns**: Identify consistent vs. variable importance across the season - **Data Collection Priority**: Guide decisions about temporal resolution of monitoring

Methodology: Instead of permuting individual features, we permute **entire time steps**, disrupting the temporal relationships while preserving the feature distributions at each time point.

```python
def timestep_importance(model, X, y, metric, n_repeats=10):
    """
    Calculate importance scores for each time step using permutation importance.

    Parameters:
    -----------
    model : keras.Model
```

```
        Trained model
    X : numpy.ndarray
        Input data of shape (samples, timesteps, features)
    y : numpy.ndarray
        Target values
    metric : function
        Scoring metric (e.g., r2_score)
    n_repeats : int
        Number of times to repeat the permutation

    Returns:
    --------
    importances : list
        Importance score for each time step
    importances_std : list
        Standard deviation of importance scores for each time step
    timesteps : list
        List of time step indices
    """

    # Set random seed for reproducibility
    np.random.seed(32)

    baseline_score = metric(y, model.predict(X)[:, 0])
    importances = []
    importances_std = []
    timesteps = list(range(X.shape[1]))  # Get list of timestep indices

    # For each timestep
    for timestep_idx in range(X.shape[1]):
        scores = []
        for _ in range(n_repeats):
            X_permuted = X.copy()
            # Permute all features at this timestep
            permuted_values = np.random.permutation(X_permuted[:, timestep_idx, :])
            X_permuted[:, timestep_idx, :] = permuted_values
            permuted_score = metric(y, model.predict(X_permuted)[:, 0])
            scores.append(baseline_score - permuted_score)

        mean_score = np.mean(scores)
        std_score = np.std(scores)
```

```
        if mean_score > 0:
            importances.append(mean_score)
        else:
            importances.append(0)

        importances_std.append(std_score)

    return importances, importances_std, timesteps

def plot_timestep_importance(importances, importances_std, timesteps, window_size):
    """
    Plot timestep importance scores with error bars.

    Parameters:
    -----------
    importances : list
        Importance scores
    importances_std : list
        Standard deviation of importance scores
    timesteps : list
        Time step indices
    window_size : int
        Size of the input window in days
    """
    plt.figure(figsize=(8/1.5, 6/1.5))

    # Convert timestep indices to days before prediction
    days_before = [(window_size - i) for i in timesteps]

    plt.bar(days_before, importances, yerr=importances_std,
            # capsize=5, error_kw={'elinewidth': 2, 'alpha': 0.8})
            capsize=4, alpha=0.8, color='cornflowerblue', ecolor='black')
    plt.xlabel('Phenological Stage')
    plt.ylabel('Importance Score (decrease in R²)')
    plt.title('Phenological Stage Importance Based on Permutation Test')
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.show()

def analyze_timestep_importance(model, X, y, window_size, metric=r2_score, n_repeats=10):
    """
```

```
    Analyze and visualize time step importance.

    Parameters:
    -----------
    model : keras.Model
        Trained model
    X : numpy.ndarray
        Input data
    y : numpy.ndarray
        Target values
    window_size : int
        Size of the input window in days
    metric : function
        Scoring metric (default: r2_score)
    n_repeats : int
        Number of permutation repeats (default: 10)
    """
    print("Calculating timestep importance scores...")
    importance_scores, importance_stds, timesteps = timestep_importance(model, X, y, metric,

    # Print results sorted by importance
    results = list(zip(timesteps, importance_scores, importance_stds))
    results.sort(key=lambda x: x[1], reverse=True)

    print("\nPhenology Stage Importance Rankings:")
    print("Phenology stage| Importance Score | Std Dev")
    print("-" * 55)
    for timestep, score, std in results:
        days_before = window_size - timestep
        print(f"{days_before:^19} | {score:>15.4f} | {std:>7.4f}")

    # Create visualization
    plot_timestep_importance(importance_scores, importance_stds, timesteps, window_size)

    return importance_scores, importance_stds, timesteps

window_size = X_train.shape[1]  # Your input window size
importance_scores, importance_stds, timesteps = analyze_timestep_importance(model, X_test, Y_
```

5.9.7 3.4 Spatial-Temporal Visualization and Uncertainty Mapping

This section provides geographic visualization of model predictions and uncertainties, enabling:

Spatial Pattern Analysis: - **Regional Performance**: How well does the model perform in different geographic areas? - **Spatial Uncertainty Patterns**: Are there geographic regions where the model is less confident? - **Agricultural Zone Validation**: Do predictions align with known agricultural productivity zones?

Temporal Consistency: - **Year-to-Year Stability**: How consistent are predictions across different growing seasons? - **Climate Impact Visualization**: Spatial patterns of model response to variable weather - **Risk Assessment**: Geographic identification of high-uncertainty areas for targeted monitoring

Visualization Features: - **Choropleth Maps**: County-level prediction and uncertainty visualization - **Consistent Color Scales**: Enable cross-year comparison - **Uncertainty Quantification**: Visual representation of model confidence

```
def get_overall_ranges(df_all, years_range):
    """Calculate min/max values for yield and uncertainty across all years."""
    all_predictions = []
    all_uncertainties = []

    for year in years_range:
        X_test = np.load(save_path + str(year) + '_X.npy')
        X_test = normalize_data(X_test, X_mean, X_std)
        predictions = model.predict(X_test)
        all_predictions.extend(predictions[:, 0])
        all_uncertainties.extend(predictions[:, 1])

    return {
        'yield_min': np.min(all_predictions),
        'yield_max': np.max(all_predictions),
        'uncertainty_min': np.min(all_uncertainties),
        'uncertainty_max': np.max(all_uncertainties)
    }

def plot_yield_prediction(counties, year, value_ranges, figsize=(8, 6), show_coords=True):
    """
    Create yield prediction map with less dense ticks
    """
    yield_colors = ['#f7fcf0', '#e0f3db', '#ccebc5', '#a8ddb5', '#7bccc4',
                    '#4eb3d3', '#2b8cbe', '#0868ac', '#084081']
```

```
    yield_cmap = LinearSegmentedColormap.from_list('yield_professional', yield_colors)

    fig, ax = plt.subplots(1, 1, figsize=figsize)

    # Plot counties
    counties.boundary.plot(ax=ax, linewidth=0.2, color='#666666', alpha=0.8)
    counties.plot(column='Predicted_Yield',
                 ax=ax,
                 cmap=yield_cmap,
                 vmin=value_ranges['yield_min'],
                 vmax=value_ranges['yield_max'],
                 edgecolor='none')

    if show_coords:
        bounds = counties.total_bounds
        ax.set_xlim(bounds[0], bounds[2])
        ax.set_ylim(bounds[1], bounds[3])

        # Create less dense coordinate ticks
        lon_range = bounds[2] - bounds[0]
        lat_range = bounds[3] - bounds[1]

        # Increase spacing for less dense ticks
        if lon_range > 20:
            lon_step = 5    # Changed from 2 to 5
            lat_step = 2    # Changed from 1 to 2
        elif lon_range > 10:
            lon_step = 3    # For medium areas
            lat_step = 2
        else:
            lon_step = 2    # For smaller areas
            lat_step = 1

        lon_ticks = np.arange(np.ceil(bounds[0]/lon_step)*lon_step, bounds[2], lon_step)
        lat_ticks = np.arange(np.ceil(bounds[1]/lat_step)*lat_step, bounds[3], lat_step)

        ax.set_xticks(lon_ticks)
        ax.set_yticks(lat_ticks)
        ax.set_xticklabels([f'{int(abs(x))}°W' for x in lon_ticks])
        ax.set_yticklabels([f'{int(y)}°N' for y in lat_ticks])

        ax.grid(True, alpha=0.3, linewidth=0.5, color='gray')
```

```
        ax.set_xlabel('Longitude')
        ax.set_ylabel('Latitude')
    else:
        ax.axis('off')

    # Add colorbar
    sm = plt.cm.ScalarMappable(cmap=yield_cmap,
                               norm=plt.Normalize(vmin=value_ranges['yield_min'],
                                                  vmax=value_ranges['yield_max']))
    cbar = plt.colorbar(sm, ax=ax, shrink=0.8, aspect=30, pad=0.02)
    # cbar.set_label('Predicted Yield (bu/acre)', rotation=90, labelpad=15)

    ax.set_title(f'Predicted Crop Yield (bu/ac) - {year}', fontweight='bold', pad=15)

    return fig, ax

def plot_uncertainty(counties, year, value_ranges, figsize=(8, 6), show_coords=True):
    """
    Create uncertainty map with less dense ticks
    """
    uncertainty_colors = ['#053061', '#2166ac', '#4393c3', '#92c5de', '#d1e5f0',
                          '#f7f7f7', '#fddbc7', '#f4a582', '#d6604d', '#b2182b', '#67001f']
    uncertainty_cmap = LinearSegmentedColormap.from_list('uncertainty_professional', uncertai

    fig, ax = plt.subplots(1, 1, figsize=figsize)

    # Plot counties
    counties.boundary.plot(ax=ax, linewidth=0.2, color='#666666', alpha=0.8)
    counties.plot(column='Uncertainty',
                  ax=ax,
                  cmap=uncertainty_cmap,
                  vmin=value_ranges['uncertainty_min'],
                  vmax=value_ranges['uncertainty_max'],
                  edgecolor='none')

    if show_coords:
        bounds = counties.total_bounds
        ax.set_xlim(bounds[0], bounds[2])
        ax.set_ylim(bounds[1], bounds[3])

        # Create less dense coordinate ticks (same logic as yield plot)
        lon_range = bounds[2] - bounds[0]
```

```
        lat_range = bounds[3] - bounds[1]

        if lon_range > 20:
            lon_step = 5    # Less dense
            lat_step = 2
        elif lon_range > 10:
            lon_step = 3
            lat_step = 2
        else:
            lon_step = 2
            lat_step = 1

        lon_ticks = np.arange(np.ceil(bounds[0]/lon_step)*lon_step, bounds[2], lon_step)
        lat_ticks = np.arange(np.ceil(bounds[1]/lat_step)*lat_step, bounds[3], lat_step)

        ax.set_xticks(lon_ticks)
        ax.set_yticks(lat_ticks)
        ax.set_xticklabels([f'{int(abs(x))}°W' for x in lon_ticks])
        ax.set_yticklabels([f'{int(y)}°N' for y in lat_ticks])

        ax.grid(True, alpha=0.3, linewidth=0.5, color='gray')
        ax.set_xlabel('Longitude')
        ax.set_ylabel('Latitude')
    else:
        ax.axis('off')

    # Add colorbar
    sm = plt.cm.ScalarMappable(cmap=uncertainty_cmap,
                              norm=plt.Normalize(vmin=value_ranges['uncertainty_min'],
                                                 vmax=value_ranges['uncertainty_max']))
    cbar = plt.colorbar(sm, ax=ax, shrink=0.8, aspect=30, pad=0.02)
    # cbar.set_label('Prediction Uncertainty (bu/acre)', rotation=90, labelpad=15)

    ax.set_title(f'Prediction Uncertainty (bu/ac) - {year}', fontweight='bold', pad=15)

    return fig, ax


# Calculate overall ranges once
value_ranges = get_overall_ranges(df_all, range(2018, 2022))

# Create separate plots for each year
```

```
for year in range(2018, 2022):
    # Load the shapefile
    counties = gpd.read_file(save_path+'cb_2019_us_county_20m.shp')

    # Get data for current year
    df_year = df_all[df_all['Year'] == year]

    # Load and process X data
    X_test = np.load(save_path + str(year) + '_X.npy')
    X_test = normalize_data(X_test, X_mean, X_std)

    # Get predictions
    predictions = model.predict(X_test)
    mean_pred = predictions[:, 0]
    std_pred = predictions[:, 1]

    # Add predictions to dataframe
    df_year['Predicted_Yield'] = mean_pred
    df_year['Uncertainty'] = std_pred

    # Merge data
    df_year['GEOID'] = df_year['GEOID'].astype(str)
    counties['GEOID'] = counties['GEOID'].astype(str)
    counties = counties.merge(df_year, left_on='GEOID', right_on='GEOID')

    # Create and save yield prediction plot
    fig_yield, ax_yield = plot_yield_prediction(counties, year, value_ranges)
    # save_individual_figures(fig_yield, year, 'yield_prediction', save_path)
    plt.show()

    # Create and save uncertainty plot
    fig_uncertainty, ax_uncertainty = plot_uncertainty(counties, year, value_ranges)
    # save_individual_figures(fig_uncertainty, year, 'uncertainty', save_path)
    plt.show()

    # Close figures to save memory
    plt.close(fig_yield)
    plt.close(fig_uncertainty)
```

```
# Load the LSTM model
model = create_probabilistic_lstm(input_shape=(X_train_split.shape[1], X_train_split.shape[2
with tf.keras.utils.custom_object_scope({
```

```
    'ProbabilisticLayer': ProbabilisticLayer,
    'NLL': NLL
}):
    model = tf.keras.models.load_model(os.path.join(file_folder, 'model_lstm.h5'))

# Load the county data
save_path = file_folder + '/'
df_all = pd.read_csv(save_path + '/Yield_2018_2021.csv')

def plot_annual_average_yields_with_std():
    """
    Plot average actual vs predicted yields for each test year with standard deviation error
    """
    years = list(range(2018, 2022))
    avg_actual_yields = []
    avg_predicted_yields = []
    std_actual_yields = []
    std_predicted_yields = []

    # Calculate average and std yields for each year
    for year in years:
        # Load and process X data for the year
        X_test_year = np.load(save_path + str(year) + '_X.npy')
        X_test_year = normalize_data(X_test_year, X_mean, X_std)

        # Get predictions
        predictions = model.predict(X_test_year)
        pred_yield = predictions[:, 0]

        # Get actual yields for the year
        df_year = df_all[df_all['Year'] == year]
        true_yield = df_year['Yield'].values

        # Calculate averages and standard deviations
        avg_actual = np.mean(true_yield)
        avg_predicted = np.mean(pred_yield)
        std_actual = np.std(true_yield)
        std_predicted = np.std(pred_yield)

        avg_actual_yields.append(avg_actual)
        avg_predicted_yields.append(avg_predicted)
        std_actual_yields.append(std_actual)
```

```
    std_predicted_yields.append(std_predicted)

    print(f"Year {year}:")
    print(f"  Actual: {avg_actual:.2f} ± {std_actual:.2f} bu/ac")
    print(f"  Predicted: {avg_predicted:.2f} ± {std_predicted:.2f} bu/ac")

# Create the plot
plt.figure(figsize=(10, 6))

# Plot both lines with error bars
plt.errorbar(years, avg_actual_yields, yerr=std_actual_yields,
             fmt='o-', linewidth=3, markersize=8, capsize=5, capthick=2,
             color='darkblue', label='Actual Average Yield')
plt.errorbar(years, avg_predicted_yields, yerr=std_predicted_yields,
             fmt='s-', linewidth=3, markersize=8, capsize=5, capthick=2,
             color='red', label='LSTM Predicted Average Yield')

# Customize the plot
plt.xlabel('Year', fontsize=14)
plt.ylabel('Average Soybean Yield (bu/ac)', fontsize=14)
plt.title('Average Actual vs LSTM-Predicted Soybean Yield(2018-2021)\nError bars show ±
          fontsize=16, pad=20)

# Set x-axis to show all years
plt.xticks(years, fontsize=12)
plt.yticks(fontsize=12)

# Add grid for better readability
plt.grid(True, alpha=0.3, linestyle='--')

# Add legend
plt.legend(fontsize=12, loc='best')

# Set y-axis to start from a reasonable minimum
all_values = avg_actual_yields + avg_predicted_yields
all_stds = std_actual_yields + std_predicted_yields
y_min = min(all_values) - max(all_stds) - 2
y_max = max(all_values) + max(all_stds) + 2
plt.ylim(y_min, y_max)

# Tight layout to prevent label cutoff
plt.tight_layout()
```

```
    # Show the plot
    plt.show()

    # Calculate and print overall statistics
    mae = np.mean(np.abs(np.array(avg_actual_yields) - np.array(avg_predicted_yields)))
    rmse = np.sqrt(np.mean((np.array(avg_actual_yields) - np.array(avg_predicted_yields))**2
    r2 = r2_score(avg_actual_yields, avg_predicted_yields)

    print(f"\nOverall Performance Metrics (Annual Averages):")
    print(f"MAE: {mae:.2f} bu/ac")
    print(f"RMSE: {rmse:.2f} bu/ac")
    print(f"R²: {r2:.3f}")

    # Print standard deviation comparison
    print(f"\nStandard Deviation Comparison:")
    print(f"Average Std of Actual Yields: {np.mean(std_actual_yields):.2f} bu/ac")
    print(f"Average Std of Predicted Yields: {np.mean(std_predicted_yields):.2f} bu/ac")

    return years, avg_actual_yields, avg_predicted_yields, std_actual_yields, std_predicted_y

# Create the plot with standard deviations
years, actual_yields, predicted_yields, std_actual, std_predicted = plot_annual_average_yiel
```

5.10 Conclusions and Limitations

This notebook demonstrated a phenology-guided, probabilistic deep learning workflow for soybean yield prediction using satellite-derived time series and environmental data. By aligning predictors with crop developmental stages and employing probabilistic LSTM, 1D-CNN, and Gaussian Process models, the approach captures both temporal yield dynamics and associated prediction uncertainty. Results highlight the dominant role of vegetation condition (EVI2), shallow soil moisture, and late-season phenological stages in determining final yield.

Several limitations should be noted. The analysis is conducted at the county level and does not explicitly incorporate detailed management information such as planting dates, cultivar maturity groups, or within-county variability. Additionally, while probabilistic outputs are provided, uncertainty estimates are primarily aleatoric and would benefit from further calibration and the explicit inclusion of epistemic uncertainty. Readers are referred to the accompanying book chapter for a more detailed discussion of results, interpretation, and methodological considerations.

5.11 Looking Forward: Future Directions and Next Steps

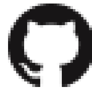
This notebook has walked through a phenology-guided deep learning pipeline for soybean yield prediction, including probabilistic modeling of prediction uncertainty. If you are relatively new to deep learning in agriculture, being able to train a time-series model that predicts yield and provides its own uncertainty estimates is already a significant step.

As discussed in the limitations and future directions of the chapter, this framework is meant to serve as a foundation rather than an endpoint. When adapting these ideas to your own work, you might consider the following directions:

- **Exploring Advanced Time Series Architectures:** You can experiment by varying the current LSTM architecture (e.g., adjusting the number of layers, units, or regularization) to observe how model complexity impacts performance and overfitting. Additionally, consider comparing the LSTM against other sequence models like Gated Recurrent Units (GRUs) or modern Attention/Transformer-based architectures. As an analytical exercise, you might also investigate early-season forecasting by training models using only the first few phenological stages to determine exactly when reliable yield signals first emerge.
- **Expanding Phenology Applications:** The satellite-derived phenology approach used here is highly adaptable. You can modify this pipeline to study different crops (such as corn or wheat) by swapping the MODIS transition dates, or test the integration of specific extreme weather proxies, like heat-stress degree days or drought indices.
- **Deepening Uncertainty Quantification (UQ):** The probabilistic models in this notebook output a mean and variance to capture *aleatoric* uncertainty (inherent noise in the data). A powerful next step is to explore *epistemic* uncertainty (uncertainty in the model's own knowledge). You could try implementing **Deep Ensembles** (training multiple models and combining their predictive distributions) or exploring **Bayesian Neural Networks** (e.g., using TensorFlow Probability) to generate even more robust, trustworthy confidence intervals for decision-makers.

Ultimately, the goal of these tools is to align with initiatives like the NASA Earth Science to Action framework—translating raw satellite data into actionable agricultural insights. We encourage you to adapt this code to your own local regions and continue exploring how machine learning can support sustainable, climate-resilient agriculture!

6 Understanding active fire detection uncertainty with Bayesian Neural Networks

Run in Colab  View on GitHub

This tutorial accompanies the work titled "Understanding active fire detection uncertainty with Bayesian Neural Networks". This notebook is intended to be accompanied by the written document and is not meant to be a standalone notebook. Please reference the written text to find further information and explanations of the steps and plots produced here.

This work was based off of the Keras Bayesian Neural Networks tutorial. Note that this tutorial uses specific versions of packages that are compatible with Keras 2.0, as Keras version 3 (at the time of development) did not support the tensorflow probabilistic layers. The import statements note the correct versions. We also provide a yml file which forces Keras 2 if you would like to run locally. Additionally, the Keras tutorial is set up for regression, and so this notebook altered the tutorial to be compatible for classification tasks.

https://www.youtube.com/embed/e6lqjTUAvmI

6.1 Important Notes (Read Before Running)

6.1.1 Python Versioning

This code is only compatible with Python 3.10, which is no longer the default version loaded by Colab. We have included the instructions on how to force the notebook to use Python 3.10, to ensure there are no errors related to version incompatabilties.

6.1.2 Code Order

The code in this tutorial is intended to be run start to finish in order. A user must also read the initial sections to properly load in the data before attempting to "Run All". Functions are defined in the section where they are first called in, and may be called again during additional sections without being re-defined in those sections. Therefore if you wish to run only a portion

of this code make sure to run all the function defining code blocks of previous sections to ensure all required functions have been defined.

6.1.3 Colab vs Jupyter

This tutorial was constructed so that every step can be run within the Google Colab environment to take advantage of the computing resources it provides. We highly recommend users who are new to Python use the option of running in Colab as it negates the need to have Python and the associated packages configured on their local machines. Therefore this is the default option of the code. However, we do recognize that for more advanced users a local workflow may be preferred, so we have included instructions for how to adapt this script to run locally if desired.

6.1.4 How to Run This Notebook on Colab - Default setting

---

1. Upload the recommened files using the code block below
2. Set colab_jupyter_flag to “c”
3. Hit the “Run all” button located above the script
4. A prompt will appear asking the user to select a python version. Select version 3.10 and see the notebook sectional labeled ‘Colab Specific Instructions’ for a more detailed explanation

####Files to upload: 1. **train-final-raw.csv:** this csv file contains raw VIIRS active fire data and contains imbalanced classes. This will be used in Case Study A, Model One to serve as a baseline dataset for this study. 2. **train-final-balanced.csv:** this is the class-balanced version of the train-final-raw.csv. It is used in Case Stdy A, Model Two to show the impact of balancing classes on model performance and in Case Study A, Model Three as an input for hyperparameter tuning. 3. **train-final-balanced-case-study-2.csv:** this dataset integrates data from both VIIRS and MODIS. It is used in Case Study B, Model One and Two to help test if fire detection varies based on satellite source.

For more information regarding the datasets, please refer to section 2.2 Data in the corresponding article.

```
#Upload the files
from google.colab import files
import os

# Define fixed upload directory
upload_dir = "/content/"
```

```
# Create directory if it doesn't exist
os.makedirs(upload_dir, exist_ok=True)

# Prompt user to upload files
uploaded = files.upload()

# Save uploaded files to the fixed directory
for filename, data in uploaded.items():
    filepath = os.path.join(upload_dir, filename)
    with open(filepath, "wb") as f:
        f.write(data)
    print(f"Saved: {filepath}")

print(f"\nAll files saved to: {upload_dir}")
```

6.1.5 How to Run This Notebook on Local

This notebook can be run locally on your machine IF you have a GPU and are at an intermediate Python programming level, otherwise you will need to use Google Colab. We suggest all users first use the Google Colab as the environment can be tricky to set up on local machines, and it is already set up here for you. The figures and numbers in the accompanying text will match the model files available for preloading with Colab. If you retrain the models (as is necessary within the Jupyter option), you will get slightly different statistics and figures when compared to the text due to some internal randomness in Keras. 1. Download: 1. the provided yml file and install to your local computer 2. the provided csv files that contain the data needed 3. Create your anaconda environment using the yml file 4. Import the anaconda environment into jupyter using these commands in the anaconda prompt: 5. conda install -c anaconda ipykernel 6. python -m ipykernel install –user –name=firstEnv 5. Update the pathway to the directory containing all csv files below 6. Set colab_jupyter_flag to "j"

Decision point: Would you like to run this code in colab or in a jupyter notebook on your local machine?

```
# set this flag to either "c" or "j" for running in jupyter or colab
colab_jupyter_flag = 'c'

# Update data directory to where you have stored the csv files
# Example data_dir = r"C:\Users\kzammit\Documents\DL-chapter\test-20251014"
if colab_jupyter_flag == 'j':
  data_dir = r"UPDATE"
```

6.2 Frequently Asked Questions

This section contains more in depth explanations regarding the code itself and how to work with it.

How do I upload the supporting files (input data, pre trained models etc) to the Colab enviroment? 1. Once the notebook has loaded on the left hand side of your screen will be a list of icons the bottom being a file that says "file". Click there to open the Files menu. 2. Identify the directory labeled "content" and drag and drop any files (input data, saved model files etc) needed for the run here. 3. When you first open the menu, content may not be an option, click on the directory labeled ".." to reveal more directories.

Where do I find the file path to assign to the variable "data_dir"? 1. Navigate to directory containing the input files using the file menu to the left. 2. Hover the cursor over the directory you need the path to point to and click the three dots that appear to the right of the directory name. 4. When the drop down menu appears click "Copy path" 5. Navigate to the box where "data_dir" is set. Under the section How to Run this Notebook in Colab. And use the keyboard command ctrl+v to paste wihtin the quotes after the r. Keyboard shortcut MUST be used. Right click drop down will not have a paste option.

Can I use this code with different input datasets than those provided?

Yes! We encourage it. Please refer to section 2.2 of the accompanying chapter for a detailed description on how we generated the datasets that have been provided as the sample datasets.

How long will this code take to run?

In general this code will take around 2 hours to run to completion when run in the Google Colab environment. Run time of local environments will vary based on the computing capabilities of the local machine. With the training of case study A model 3 and case study B model 1 as two of the longer steps at between 30 and 50 minutes. The SHAP figures also can take up to 30 minutes to render.

How to use the "Help Boxes"?

Through out the tutorial are markdown cells title Help Box. By navigating to a Help Box cell you will receive directions with how you can skip the most time intensive steps of the tutorial by loading a pretrained model. As long as the file paths associated with these help boxes are set up correctly and the "read_in_file" variable is set to True. The user can run the code from the beginning as normal. No additional actions are required from the user to skip model training.

Ran out of GPU time?

You may receive an error message from Colab saying you have run out of your allotted GPU time. This notebook does not require connection to a GPU to run. On the top right hand side of the note box is an arrow next to the words RAM and Disk. Clicking on the arrow opens a

drop down with the option “Change runtime type”. Opening that menu will show an option for Hardware accelerator. Select the CPU option and hit save. You should no longer receive GPU run time error messages.

Crashed?

Some users may encounter runtime timeouts or crashes near the end of the notebook when using Google Colab. If this occurs, we recommend using the Jupyter version of the notebook or utilizing the provided pretrained model files when working in Colab.

6.2.1 Running this Notebook without Training Models

Although the primary purpose of this code is to train a deep learning model from scratch, we recognize that a user may not wish to retrain their models every time they start a new session. Included in this code are directions for how to export a trained model so that it will be saved after a run is exited, as well as how to modify the code to import in these model files and bypass the code blocks that initiate model training. For convenience we have included sample model files so a user exploring this tutorial for the first time may bypass model training altogether. However, turning off model training is not the recommended way to interact with the tutorial. The outputs generated by Keras as it trains models are informative and offer valuable context to deepen your understanding of the models themselves.

Decision point: Would you like to run this code from “scratch” or with a pretrained model?

```
# Loading pretrained model files
# The provided model files are exported from Colab and do not run locally so we have forced
# the flag to be False if running in Jupyter
# Once you have ran once from scratch locally, feel free to switch this to "True" as the expo
# model files are compatible with Jupyter
if colab_jupyter_flag == 'j':
    read_in_file = False
else:
  # In Colab, set this to either True or False
    read_in_file = True
    # Define fixed upload directory
    upload_dir = "/content/"

    # Create directory if it doesn't exist
    os.makedirs(upload_dir, exist_ok=True)

    # Prompt user to upload files
    uploaded = files.upload()
```

```
    # Save uploaded files to the fixed directory
    for filename, data in uploaded.items():
        filepath = os.path.join(upload_dir, filename)
        with open(filepath, "wb") as f:
            f.write(data)
        print(f"Saved: {filepath}")

    print(f"\nAll models saved to: {upload_dir}")
```

6.3 Colab Specific Installations

Do not run these if running locally in a Jupyter notebook.

```
# Google colab will update its default version of Python without notice. Use this box to chec
# Included below are the steps to force the notebook back into Python 3.10, the last version
if colab_jupyter_flag == 'c':
    !python --version
```

```
# Click on the box where it says selection and choose 1 for python 3.10
if colab_jupyter_flag == 'c':
    !update-alternatives --config python3
```

```
# Run this cell to confim the notebook is now running Python 3.10
if colab_jupyter_flag == 'c':
    !python --version
```

```
# Must reinstall pip in the new python
# Note: It is expected to get warning messages from this cell (ex. WARNING: The scripts pip,
# Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no
if colab_jupyter_flag == 'c':
    !apt-get install python3-pip
    !python3 -m pip install --upgrade pip --user
```

```
# Confirm it worked. Tutorial required python 9 or 10 enviroment set up may fail if python 11
if colab_jupyter_flag == 'c':
    !python --version
    !pip --version
```

```
if colab_jupyter_flag == 'c':
  ! pip install tensorflow_probability==0.24.0
  ! pip install tf_keras
  ! pip install tfp-nightly
  ! pip install tf-nightly
  ! pip install tf-keras-nightly
  ! pip install shap
```

6.3.1 Import Statements

From here onwards, everything is compatible in both Jupyter and Colab.

```
import pandas as pd
import numpy as np
import tensorflow as tf
import tf_keras
import tensorflow_probability as tfp
from keras.utils import plot_model
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
from matplotlib.lines import Line2D
from matplotlib.colors import LinearSegmentedColormap
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
from scipy.stats import ttest_ind
import itertools
import importlib
import os
import shap
import plotly.graph_objects as go
import random

# Set random seed for reproducibility
tf.random.set_seed(42)
random.seed(42)
np.random.seed(42)

# Suppress tensorflow warnings for deprecated versions
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
# Print the version of the nonstandard versions for use in this tutorial (note "dev" in the r
print("TensorFlow version:", tf.__version__)
print("TensorFlow Probability version:", tfp.__version__)
print("TensorFlow keras version:", tf_keras.__version__)
```

6.4 Case Study A

Case Study A: Pre-screening VIIRS active fire data to highlight potential false positives using BNN.

This study uses a point prediction BNN model, which outputs a single probability score for each detection, indicating the likelihood it is a true positive. Applying a classification threshold to this score yields a binary indicator (true/false positive), primarily for use in pre-screening VIIRS data to identify and remove likely false positives.

Goal: The goal of the supervised Bayesian neural network developed in Case Study A is to predict whether a hotspot identified by a satellite-based active fire detection product corresponds to an actual fire on the ground. Note that the training and validation loss will be plotted individually after training each model, but all results (loss curves, confusion matrices, etc.) are presented at the end of the case study in the "Results" section.

6.4.1 Define epochs

```
# Define the number of epochs to train over for each run
epochs = 30
```

6.4.2 Model One

Model One closely follows the original Keras tutorial and does not include any hyperparameter tuning. You do not need to run the original Keras tutorial as a prerequiste to this notebook, as we repeat the necessary information here. It uses "raw" input data, defined here as data obtained directly from FIRMS, with no preprocessing beyond the basic steps outlined in Section 2.2. The purpose of this model is to serve as a baseline, demonstrating model performance without additional optimization or customization.

6.4.2.1 Importing and preparing the data

The notebook expects that the data has been downloaded from source and undergone basic processing steps before beginning this tutorial. See Section 2.2 for further details. We used a 80/20 ratio to split the complete input data set into separate training and testing data sets respectively.

```
def create_tr_test_data(data_file):
    """
    Split data into training and validation datasets
    :param data_file: file path to csv file, ex. 'train-final-raw.csv' if you've uploaded lo
    :return: X train, X test, y train, y test, feature names (used in model creation)
    """

    # Define raw data file and read
    train_data = pd.read_csv(data_file)

    print(train_data.head())

    # Drop the target variable
    X = train_data.drop(columns=['Class'])

    # Set the target variable
    y = train_data['Class']

    # Create the training and testing datasets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
    feature_names = X_train.columns

    X_train = X_train.to_numpy()
    X_test = X_test.to_numpy()
    y_train = y_train.to_numpy()
    y_test = y_test.to_numpy()

    return X_train, X_test, y_train, y_test, feature_names
```

```
# Note: You must add your data as described in the beginning sections:
# "How to Run This Notebook on Local" and "How to Run This Notebook on Colab"
if colab_jupyter_flag == 'j':
  cs1_m1_data = os.path.join(data_dir, 'train-final-raw.csv')
else:
  cs1_m1_data = '/content/train-final-raw.csv'
```

```
X_train_c1m1, X_test_c1m1, y_train_c1m1, y_test_c1m1, feature_names_c1m1 = create_tr_test_da
print("Preview of provided data to confirm sucessful upload")
train_size_c1m1 = X_train_c1m1.shape[0]
```

6.4.2.2 Create and train model

All of the model training criteria we defined in section 3.3 of the manuscript were set during this section of the code.

```
def prior(kernel_size, bias_size, dtype=None):
    """
    Define the prior distribution (not tunable)
    :param kernel_size: defined by model
    :param bias_size: defined by model
    :param dtype: defined by model
    :return: prior model
    """

    n = kernel_size + bias_size
    prior_model = tf_keras.Sequential(
      [
          tfp.layers.DistributionLambda(
              lambda t: tfp.distributions.MultivariateNormalDiag(
                  loc=tf.zeros(n), scale_diag=tf.ones(n)
              )
          )
      ]
    )
    return prior_model
```

```
def posterior(kernel_size, bias_size, dtype=None):
    """
    Define the posterior distribution (tunable)
    :param kernel_size: defined by model
    :param bias_size:
    :param dtype:
    :return: posterior model
    """

    n = kernel_size + bias_size
    posterior_model = tf_keras.Sequential(
```

```
      [
          tfp.layers.VariableLayer(
              tfp.layers.MultivariateNormalTriL.params_size(n), dtype=dtype
          ),
          tfp.layers.MultivariateNormalTriL(n),
      ]
    )
    return posterior_model
```

```
def create_model_inputs(feature_names):
    """
    Create inputs of tensorflow float type 32 for model input
    :return: inputs
    """
    inputs = {}
    for feature_name in feature_names:
      inputs[feature_name] = tf_keras.Input(name=feature_name, shape=(1,), dtype=tf.float32)
    return inputs
```

```
def create_bnn_model(train_size, prior_func, feature_names, activation_fun = "relu", unit_dim
    """
    Create bayesian neural network model
    :param train_size: number of samples in training dataset
    :param prior_func: name of prior function (posterior function never changes in this examp
    :param activation_fun: activation function for hidden layers
    :param unit_dim: number of units in hidden layers
    :return: model
    """

    inputs = create_model_inputs(feature_names)

    # Create one keras tensor for all inputs
    features = tf_keras.layers.concatenate(list(inputs.values()))
    features = tf_keras.layers.BatchNormalization()(features)
    hidden_units = [unit_dim,unit_dim]

    for units in hidden_units:
      features = tfp.layers.DenseVariational(
          units=units,
          make_prior_fn=prior_func,
          make_posterior_fn=posterior,
          kl_weight=1 / train_size,
```

```
            activation=activation_fun,
        )(features)

    # 2 for 2 classes
    distribution_params = tf_keras.layers.Dense(units=2)(features)

    outputs = tf_keras.layers.Dense(1, activation="sigmoid")(features)

    model = tf_keras.Model(inputs=inputs, outputs=outputs)
    return model
```

```
def run_experiment(model, loss, metrics, epochs, feature_names, X_train, y_train):
    """
    Run the experiment
    :param model: defined model to train
    :param loss: loss function to use during training
    :param metrics: metrics to output during training
    :param epochs: number of epochs
    :param X_train: training data
    :param y_train: training data labels
    :return: model
    """

    # Compile the model
    model.compile(optimizer='adam', loss=loss, metrics=metrics)

    X_train_format = {str(feature_names[i]): X_train[:, i] for i in range(X_train.shape[1])}

    # Train the model
    history = model.fit(
        x=X_train_format,
        y=y_train,
        batch_size=32,
        epochs=epochs,
        validation_split = 0.2,
        verbose=1
    )

    train_loss = history.history['loss']
    val_loss = history.history['val_loss']

    return model, train_loss, val_loss
```

6.4.2.3 Help Box - Skip model training

By setting the variable "read_in_file" to True at the beginning of the code, the user activates sections of the code provided to allow the time intensive model training steps to be skipped. It is assumed that if a user is activating these sections of the code they have uploaded the appropriately named files to the same directory that the input data is stored in. File names are set by default to call the sample files that have provided with this tutorial.

```
if read_in_file:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy1_model1_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m1.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m1.weights.h5"
    training_output_file = "casestudy1_model1_df.csv"

  df_c1m1 = pd.read_csv(training_output_file)
  reconstructed_model = create_bnn_model(train_size_c1m1, prior, feature_names_c1m1,
                                          activation_fun = 'relu', unit_dim = 8)
  reconstructed_model.load_weights(model_file_name)
  trained_model_c1m1 = reconstructed_model
  train_loss_c1m1 = df_c1m1['train_loss']
  val_loss_c1m1 = df_c1m1['val_loss']
```

```
# Compile an untrained model
model_c1m1 = create_bnn_model(train_size_c1m1, prior, feature_names_c1m1, activation_fun = 'r
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. If this i
else:
  # Train model
  # input order: model, loss, metrics, epochs, X_train, X_val, y_train, y_val
  trained_model_c1m1, train_loss_c1m1, val_loss_c1m1 = run_experiment(model_c1m1, 'binary_cro
    [['accuracy', tf_keras.metrics.Precision(), tf_keras.metrics.Recall()]], epochs,
                                          feature_names_c1m1, X_train_c1m1, y_train_c1m1)
```

6.4.2.4 Help Box - Saving model files

By default, once a model has been trained the notebook will export the model file and outputs from its training runs to files saved within the working directory. If the notebook is being run

on Colab these files WILL NOT be saved once the Colab run is exited. If a user wishes to retain these files download them to a local directory. If these users does not care about these outputs, they will automatically be deleted when the notebook is closed. If the user is running local on Jupyter the files will have to be deleted by hand.

```
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. Therefor
else:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy1_model1_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m1.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m1.weights.h5"
    training_output_file = "casestudy1_model1_df.csv"

  model_c1m1.save_weights(model_file_name)
  df_c1m1 = pd.DataFrame({'train_loss': train_loss_c1m1,
                    'val_loss': val_loss_c1m1})
  df_c1m1.to_csv(training_output_file)
```

```
### Should be a redundant cell. Leaving in for now just incase
# Save gate for experiment 1 outputs
if colab_jupyter_flag == 'j':
  dir_path = os.path.join(data_dir, 'output')
  os.makedirs(dir_path, exist_ok=True)
  if read_in_file:
      df_c1m1 = pd.read_csv(os.path.join(dir_path, "casestudy1_model1_df.csv"))
      reconstructed_model = create_bnn_model(train_size_c1m1, prior, feature_names_c1m1,
                                      activation_fun = 'relu', unit_dim = 8)
      reconstructed_model.load_weights(os.path.join(dir_path, "model_c1m1.weights.h5"))
      trained_model_c1m1 = reconstructed_model
      train_loss_c1m1 = df_c1m1['train_loss']
      val_loss_c1m1 = df_c1m1['val_loss']
  else:
      model_c1m1.save_weights(os.path.join(dir_path, "model_c1m1.weights.h5"))
      df_c1m1 = pd.DataFrame({'train_loss': train_loss_c1m1,
                    'val_loss': val_loss_c1m1})
      df_c1m1.to_csv(os.path.join(dir_path, "casestudy1_model1_df.csv"))
```

6.4.2.5 Compute predictions

```
def compute_predictions(model, feature_names, X_test, y_test):
    predictions = {str(feature_names[i]): X_test[:, i] for i in range(X_test.shape[1])}
    predictions = model(predictions)
    return predictions
```

```
# Generate predictions using the newly trained model to be used in the Results section of the
predictions_c1m1 = compute_predictions(trained_model_c1m1, feature_names_c1m1, X_test_c1m1, y
```

6.4.2.6 Visualize training and validation loss

Visualize the training and validation loss for Case Study A Model One.

```
def plot_training_loss(feature_names, model, X_val, y_val, train_loss, val_loss, predictions

    """
    Plot the loss curve and confusion matrix
    :param: feature_names: feature names used in model creation
    :param: model: trained model
    :param: X_val: validation data
    :param: y_val: validation data labels
    :param: train_loss: training loss over epochs
    :param: val_loss: validation loss over epochs
    :param: title: title of plot
    """
    plt.rcParams.update({ # set visualization values
    "font.size":            10,
    "axes.titlesize":       10,
    "axes.labelsize":       10,
    "xtick.labelsize":      10,
    "ytick.labelsize":      10,
    "legend.fontsize":      10,
    "figure.titlesize":     10})
    color_palette = ["#55C667FF","#20A387FF","#33638DFF","#481567FF"]
    custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)
    # # set up array for predictions
    # predictions = {str(feature_names[i]): X_val[:, i] for i in range(X_val.shape[1])}

    # # get model predictions
```

```
    # predictions = model(predictions)

    fig, (ax1) = plt.subplots(1, 1, figsize=(10,5))

    ax1.plot(train_loss, label='Training Loss', marker='o', color="#33638DFF")
    ax1.plot(val_loss, label='Validation Loss', marker='o', color="#481567FF")
    ax1.set_xlabel('Epochs')
    ax1.set_ylabel('Loss')
    #ax1.set_ylim(0, 1)
    ax1.grid()
    ax1.legend()

    fig.suptitle(title)
    plt.show()
```

```
plot_training_loss(feature_names_c1m1, trained_model_c1m1, X_test_c1m1, y_test_c1m1, train_lo
```

6.4.3 Model Two

For the second model, we expanded on the tutorial code by including more advanced data preprocessing functions. When applying machine learning methods, it is crucial to understand the characteristics of the input data, including any inherent limitations and biases. In Model One, no effort was made to balance the classes, which can significantly impair the network's ability to learn the key features needed to distinguish between true and false positives (Lee et al., 2016). In Model Two, we address this issue by balancing the dataset before training the model. To achieve this, we apply random undersampling to the original dataset, ensuring balance not only between classes (true and false positives) but also between satellite sources (NOAA-20 and Suomi-NPP). This helps prevent the model from learning satellite-specific noise patterns or resolution/angle biases instead of features associated with actual fire detections. Once data preprocessing was complete, model compiling and training were implemented in the same way as the previous model.

6.4.3.1 Create dataset

```
# Create X and y train and test data arrays
if colab_jupyter_flag == 'j':
  cs1_m2_data = os.path.join(data_dir, 'train-final-balanced.csv')
else:
  cs1_m2_data = '/content/train-final-balanced.csv'
```

```
X_train_c1m2, X_test_c1m2, y_train_c1m2, y_test_c1m2, feature_names_c1m2 = create_tr_test_dat
print("Preview of provided data to confirm sucessful upload")
train_size_c1m2 = X_train_c1m2.shape[0]
```

6.4.3.2 Create and train model

6.4.3.3 Help Box - Skip model training

By setting the variable "read_in_file" to at the beginning of the code, the user activates sections of the code provided to allow the time intensive model training steps to be skipped. It is assumed that if a user is activating these sections of the code they have uploaded the appropriately named files to the same directory that the input data is stored in. File names are set by default to call the sample files that have provided with this tutorial.

```
if read_in_file:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy1_model2_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m2.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m2.weights.h5"
    training_output_file = "casestudy1_model2_df.csv"

  df_c1m2 = pd.read_csv(training_output_file )
  reconstructed_model = create_bnn_model(train_size_c1m2, prior, feature_names_c1m2,
                                          activation_fun = 'relu', unit_dim = 8)
  reconstructed_model.load_weights(model_file_name)
  trained_model_c1m2 = reconstructed_model
  train_loss_c1m2 = df_c1m2['train_loss']
  val_loss_c1m2 = df_c1m2['val_loss']
```

```
model_c1m2= create_bnn_model(train_size_c1m2, prior, feature_names_c1m2, activation_fun = 're
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. If this i
else:
  # Train model
  trained_model_c1m2, train_loss_c1m2, val_loss_c1m2 = run_experiment(model_c1m2, 'binary_cro
    [['accuracy', tf_keras.metrics.Precision(), tf_keras.metrics.Recall()]], epochs,
                                          feature_names_c1m2, X_train_c1m2, y_train_c1m2)
```

6.4.3.4 Help Box - Saving model files

By default, once a model has been trained the notebook will export the model file and outputs from its training runs to files saved within the working directory. If the notebook is being run on Colab these files WILL NOT be saved once the Colab run is exited. If a user wishes to retain these files download them to a local directory. If these users does not care about these outputs, they will automatically be delated when the notebook is closed. If the user is running local on Jupyter the files will have to be deleted by hand.

```
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. Therefor
else:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy1_model2_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m2.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m2.weights.h5"
    training_output_file = "casestudy1_model2_df.csv"

  model_c1m2.save_weights(model_file_name)
  df_c1m2 = pd.DataFrame({'train_loss': train_loss_c1m2,
                      'val_loss': val_loss_c1m2})
  df_c1m2.to_csv(training_output_file )
```

6.4.3.5 Compute predictions

```
# Generate predictions using the newly trained model to be used in the Results section of the
predictions_c1m2 = compute_predictions(trained_model_c1m2, feature_names_c1m2, X_test_c1m2, y
```

6.4.3.6 Visualize training and validation loss

Visualize the training and validation loss for Case Study A Model Two.

```
plot_training_loss(feature_names_c1m2, trained_model_c1m2, X_test_c1m2, y_test_c1m2, train_lo
```

6.4.4 Model Three

For the third model, we utilized the balanced data that was used in Model Two, and expanded the model-building code beyond what was provided by the tutorial to include hyperparameter tuning into the workflow.

6.4.5 Define epochs

Number of epochs used for trianing is increased.

```
epochs = 100
```

6.4.5.1 Create and train model

```
def prior_tune(kernel_size, bias_size, dtype=None):
    """
    Define tunable prior distribution.
    :param kernel_size: defined by model
    :param bias_size: defined by model
    :param dtype: defined by model
    :return: prior model
    """

    n = kernel_size + bias_size

    prior_model = tf_keras.Sequential(
      [
          tfp.layers.DistributionLambda(
            apply_parameter(fun = active_prior_distribution, var_list = [n])
          )
      ]
    )
    return prior_model


def apply_parameter(fun, var_list):
    out = fun(var_list)
    return out
```

```
def distribution_param1(var_list):
    out = lambda t: tfp.distributions.MultivariateNormalDiag(
                     loc=tf.zeros(var_list[0]), scale_diag=tf.ones(var_list[0]))
    return out


def distribution_param2(var_list):
    out = lambda t: tfp.distributions.MultivariateNormalDiag(
                     loc=tf.zeros(var_list[0])*20, scale_diag=tf.ones(var_list[0])*20)
    return out


def ParameterGrid(input_dict):
    # Extract the lists corresponding to each key
    values = list(input_dict.values())

    # Use itertools.product to get all combinations of values
    combinations = itertools.product(*values)

    # Create a list of dictionaries from the combinations
    result = []
    for combo in combinations:
        result.append(dict(zip(input_dict.keys(), combo)))

    return result
```

6.4.5.2 Help Box - Prior distribution explanation

The prior distribution helps control how much uncertainty and flexibility the model has in its weights. The distribution_param1 defines a standard normal prior where each weight has mean 0 and standard deviation of 1. This indicates that the weights should stay near zero. While the distribution_param2 does the same, but with a standard deviation of 20, indicating that the weights could vary widely.

```
prior_distribution_dic = {'default': distribution_param1,
                          'stretch': distribution_param2}

BNN_activation_function_dic = {'default': 'relu',
                               'sigmoid': 'sigmoid',
                               'gelu':'gelu',
                               'ELU': 'elu',
```

```
                              'Leaky relu':'leaky_relu'}

BNN_hiddenunits_dim_dic = {'smaller': 4,
                           'default': 8,
                           'larger': 16}

# packed into one object, needed values (not keys for the parameteric grid)
param_grid = {
    "dist":list(prior_distribution_dic.values()),
    "activation":list(BNN_activation_function_dic.values()),
    # "class_id":list(class_id_threshold_dic.values()),
    "hidden_units":list(BNN_hiddenunits_dim_dic.values())
}
```

6.4.5.3 Help Box - Skip model training and tuning

This help box allows the user to skip both the training and the parameter tuning steps. In order to read in a saved model file, a user must pre-define structural hyperparameters of the model being inputted. The hyperparameters selected for the given sample data set may differ form the hyperparameters that would be selected for model's trained on different input data. A comment has been added to the code indicated the line that a user must be updated to match the hyperparameters selected for the new dataset in order for the model files to load properly.

By setting the variable "read_in_file" to at the beginning of the code, the user activates sections of the code provided to allow the time intensive model training steps to be skipped. It is assumed that if a user is activating these sections of the code they have uploaded the appropriately named files to the same directory that the input data is stored in. File names are set by default to call the sample files that have provided with this tutorial.

```
if read_in_file:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy1_model3_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m3.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m3.weights.h5"
    training_output_file = "casestudy1_model3_df.csv"

  df_c1m3 = pd.read_csv(training_output_file)
  # Following hyperparameters are set in accordance with optimizing the sample data.
```

```
  # If running with personal data set update these hyperparameters to match the results of th
  reconstructed_model = create_bnn_model(train_size_c1m2, prior, feature_names_c1m2, activati
  reconstructed_model.load_weights(model_file_name)
  trained_model_c1m3 = reconstructed_model
  train_loss_c1m3 = df_c1m3['train_loss']
  val_loss_c1m3 = df_c1m3['val_loss']
```

Estimated run time on CPU: 45 minutes

```
models_c1m3 = []
train_losses_c1m3 = []
val_losses_c1m3 = []
params_combo = []
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. If this i
else:
  grid = list(ParameterGrid(param_grid))
  total_runs = len(grid)
  for i, params in enumerate(grid, start =1): # creates all combinations of dictionary that i
      runs_left = total_runs - i
      print(f"[{i}/{total_runs} runs] Running with parameters: {params}")


      active_prior_distribution = params["dist"]

      model = create_bnn_model(train_size_c1m2, prior_tune, feature_names_c1m2,
                          activation_fun = params['activation'],
                          unit_dim = params['hidden_units'])

      model, train_loss, val_loss = run_experiment(model, 'binary_crossentropy', [['accuracy
                                              feature_names_c1m3, X_train_c1m3, y_train_c1m3)

      models_c1m3.append(model)
      train_losses_c1m3.append(train_loss)
      val_losses_c1m3.append(val_loss)
      params_combo.append(params)
      print(f'Finished run {i}/{total_runs}. Runs remaining: {runs_left}')
```

6.4.5.4 Evaluate Hyperparameters

The different models generated during hyperparameter tuning will be qualitatively evaluated using a Parallel Coordinates Plot (PCP), a scatter plot comparing training loss to training validation, and a SHapley Additive exPlanations (SHAP) kernel explainer.

```
# This code block outputs a parallel coordinates plot for Model Three.

if read_in_file:
  print("User has selected to read in saved models instead of training and tuning novel one.
else:
  df = pd.DataFrame(params_combo)

  # Calculate mean train and validation losses
  df["train_loss"] = [np.mean(loss) for loss in train_losses_c1m3]  # Averaging over epochs
  df["val_loss"] = [np.mean(loss) for loss in val_losses_c1m3]

  # Drop rows with missing values
  df.dropna(inplace=True)

  # Initialize an empty dictionary to store encoding maps for each column
  encoding_maps = {}

  # Loop through each categorical column that needs to be encoded
  for col in ["dist", "activation", "hidden_units"]:
    # Create a mapping from each unique value in the column to a unique integer index
    encoding_maps[col] = {val: idx for idx, val in enumerate(df[col].unique())}

    # Replace the original string values in the DataFrame with their corresponding integer en
    df[col] = df[col].map(encoding_maps[col])

  fig = go.Figure(data=
      go.Parcoords(
          line = dict(color = df['val_loss'],
                   colorscale = [[0,'purple'],[0.5,'lightseagreen'],[1,'gold']],),
          dimensions = list([
              dict(range = [0,int(len(param_grid["dist"])-1)],
                label = 'Dist', values = df['dist'], tickvals = [0,1], ticktext = ['Default'
              dict(range = [0,int(len(param_grid["activation"])-1)],
                label = 'Activation', values = df['activation'], tickvals = [0,1,2,3,4], tick
              dict(range = [0,int(len(param_grid["hidden_units"])-1)],
                label = 'Hidden Units', values = df['hidden_units'], tickvals = [0,1,2], tick
              dict(range = [0,1.6],
```

```
                label = 'Train Loss', values = df['train_loss']),
              dict(range = [0,1.6],
                label = 'Val Loss', values = df['val_loss'])
          ])
      )
  )

  fig.update_layout(
      # Font increased from 21
      font=dict(family="Arial, sans-serif", size=25),
      plot_bgcolor = 'white',
      paper_bgcolor = 'white')

  fig.show()
```

```
# This code block outputs the different training and validation losses for each trained model

if read_in_file:
  print("User has selected to read in saved models instead of training and tuning novel one.
else:
  # Prepare figure & axes
  fig, ax = plt.subplots(figsize=(10, 6))
  # Drop rows with missing values
  df.dropna(inplace=True)

  # select colors
  color_palette = ["#481567FF", "#33638DFF", "#20A387FF", "#55C667FF", "#DCE319FF"]

  # Plot
  sns.scatterplot(ax=ax, data=df,x="train_loss",y="val_loss",hue="activation", style="dist",

  #Diagonal line
  min_loss = min(df["train_loss"].min(), df["val_loss"].min())
  max_loss = max(df["train_loss"].max(), df["val_loss"].max())
  ax.plot([min_loss, max_loss], [min_loss, max_loss], color="gray", linestyle="--", linewidth


  activation_keys    = list(BNN_activation_function_dic.keys())
  dist_keys          = list(prior_distribution_dic.keys())
  hiddenunits_keys   = list(BNN_hiddenunits_dim_dic.keys())

  # color pallete
```

```
palette = sns.color_palette(color_palette, len(activation_keys))

# markers for the dist_keys
markers = ["o","X","P"][:len(dist_keys)]

# sizes for hidden_units
unit_vals = [BNN_hiddenunits_dim_dic[k] for k in hiddenunits_keys]
minu, maxu = min(unit_vals), max(unit_vals)
def map_size(u):
    return 50 + (u - minu) / (maxu - minu) * (300-50)

size_proxies = [
    Line2D(
      [0],[0],
      color="gray",
      marker="o",
      linestyle="None",
      markersize=np.sqrt(map_size(unit_vals[i])),
      label=f"{hiddenunits_keys[i]} ({unit_vals[i]} units)"
    )
    for i in range(len(hiddenunits_keys))
]

hue_proxies = [
    Patch(color=palette[i], label=activation_keys[i])
    for i in range(len(activation_keys))
]

style_proxies = [
    Line2D([0],[0], color="black",  marker=markers[i], linestyle="None", markersize=10, lab
    for i in range(len(dist_keys))
]

# legend of all the parameters
ax.legend(
    handles=hue_proxies + style_proxies + size_proxies + [Line2D([0], [0], color="gray", li
    title="Activation / Dist / Hidden Units",
    loc = "upper left",
    bbox_to_anchor=(1.02, 1),
    borderaxespad=0
)
ax.set_xlabel("Training Loss", fontsize =  18)
```

```
  ax.set_ylabel("Validation Loss", fontsize =  18)
  ax.set_title("Training vs. Validation Loss\n(color=activation, shape=dist, size=hidden_uni

  fig.subplots_adjust(right=0.75)
  plt.tight_layout()
  plt.show()
```

```
# This code block outputs a SHapley Additive exPlanations (SHAP) plot
if read_in_file:
  print("User has selected to read in saved models instead of training and tuning novel one.
else:
  #SHAP
  best_model_to_explain = df["val_loss"].idxmin()
  best_index = best_model_to_explain




  #BNN wrapper
  def bnn_predict_fn(model, X, feature_names, n_samples=100):
      # Ensure X is a numpy array
      X = np.array(X)
      examples = {str(feature_names[i]): X[:, i] for i in range(X.shape[1])}

      predicted = []
      for _ in range(n_samples):
        predicted.append(model(examples).numpy())
      return np.concatenate(predicted, axis=1)

  #plot color
  plt.rcParams.update({
      "font.size":        10,
      "axes.titlesize":   10,
      "axes.labelsize":   10,
      "xtick.labelsize":  10,
      "ytick.labelsize":  10,
      "legend.fontsize":  10,
      "figure.titlesize": 10,
  })
  color_palette = ["#DCE319FF", "#55C667FF","#20A387FF","#33638DFF","#481567FF"]
  custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)
  # Use a subset of X_train as background data
  background = X_train_c1m2[:50]
```

```
# Select and review the "best" model
best_model_to_explain = models_c1m3[best_index]

# Pass the model itself and the feature names to the wrapper
shap_explainer = shap.KernelExplainer(lambda x: bnn_predict_fn(best_model_to_explain, x, fe

# Compute SHAP values for test data
X_test_subset = X_test_c1m2[101:151]
shap_values = shap_explainer.shap_values(X_test_subset, nsamples=500)
shap_values_averaged = np.mean(shap_values, axis=2)

# Visualize the results
shap.summary_plot(shap_values_averaged, X_test_subset, feature_names=feature_names_c1m2, cm
```

6.4.5.5 Model selection

The hyperparameter combination that produced the model with the lowest validation loss value was selected to be used to evaluate against the models generated from other runs.

```
if read_in_file:
  print("User has selected to read in saved models instead of training and tuning novel one.
else:
  trained_model_c1m3 = models_c1m3[df["val_loss"].idxmin()]
  train_loss_c1m3 = train_losses_c1m3[df["val_loss"].idxmin()]
  val_loss_c1m3 = val_losses_c1m3[df["val_loss"].idxmin()]
```

6.4.5.6 Help Box - Saving model files

By default, once a model has been trained the notebook will export the model file and outputs from its training runs to files saved within the working directory. If the notebook is being run on Colab these files WILL NOT be saved once the Colab run is exited. If a user wishes to retain these files download them to a local directory. If these users does not care about these outputs, they will automatically be delated when the notebook is closed. If the user is running local on jupyter the files will have to be deleted by hand.

```
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. Therefor
else:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
```

```
    training_output_file = os.path.join(data_dir, "casestudy1_model3_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c1m3.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c1m3.weights.h5"
    training_output_file = "casestudy1_model3_df.csv"

  trained_model_c1m3.save_weights(model_file_name)
  df_c1m3 = pd.DataFrame({'train_loss': train_loss_c1m2,
                      'val_loss': val_loss_c1m2})
  df_c1m3.to_csv(training_output_file )
```

6.4.5.7 Compute predictions

```
# Generate predictions using the newly trained model to be used in the Results section of the
predictions_c1m3 = compute_predictions(trained_model_c1m3, feature_names_c1m2, X_test_c1m2, y
```

6.4.5.8 Visualize training and validation loss

Visualize the training and validation loss for Case Study A Model Three.

```
plot_training_loss(feature_names_c1m2, trained_model_c1m3, X_test_c1m2, y_test_c1m2, train_lo
```

6.4.6 Results (All Models)

Class label threshold values were optimized for each model. This was done by generating class labels using a range of threshold values from 0.25 - 0.75. Threshold value vs performance metrics were plotted on a muti-line plot, and the threshold value that maximized the most metrics for a model was selected. An accuracy assessment was implemented for every model, producing: accuracy, precision, recall, and F1 scores for each set of labeled predictions. Using these optimized threshold values, confusion matrices were generated and finalized accuracy, precision and recall scores were reported. A precision recall curve was also generated for each model, so the choice of epoch for model training could be qualitatively evaluated.

```
def analyze_threshold_value(df_all_stats):
# Plot all in one figure
  fig, ax = plt.subplots(figsize=(10, 5))
  palette = {
```

```
    'accuracy': '#33638DFF',
    'precision': '#481567FF',
    'recall':  '#20A387FF',
    'f1':  '#DCE319FF'
  }
  sns.set_style("whitegrid")
  sns.set_context("paper", font_scale=1.2)

  sns.lineplot(data=pd.melt(df_all_stats, id_vars=['threshold','model'], value_vars=['accura
              x='threshold', y='value', hue='variable', style='model', palette = palette)



# Titles & labels
  plt.title("Precision, Recall, F1 across Thresholds for All Models")
  plt.xlabel("Threshold", fontsize = 18)
  plt.ylabel("Score", fontsize = 18)
  ax.grid(True)

# legend
  ax.legend(title="Metric / Model",loc='upper left', bbox_to_anchor=(1.02, 1),borderaxespad=(

  fig.tight_layout(rect=[0, 0, 0.8, 1])

  plt.grid(True)
  plt.show()
```

```
# @title
def compute_results_grid(feature_names, model, predictions, y_val, title):
  """
  Compute the classifications and statistics (precision, recall, f1) for the trained model
  :param: feature_names: feature names used in model creation
  :param: model: trained model
  :param: X_val: validation data
  :param: y_val: validation data labels
  :param: title: title of plot
  :return: dataframe of classifications and statistics at each threshold
  """

  # NOTE: This is necessary to avoid issues with RAM, for full results please remove these su
  #predictions = predictions[0:1000]
  #y_val = y_val[0:1000]
```

```
  # create precision/recall curve plots
  thresholds = [0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75]

  df_classifications = pd.DataFrame()
  df_stats = pd.DataFrame(columns=['threshold', 'accuracy','precision', 'recall', 'f1'])

  for thresh in thresholds:
      # predicted, TP, TN, FP, FN = compute_metrics(y_val, predictions, thresh)
      prediction_label = np.where(predictions >= thresh, 1, 0)
      df_group = pd.DataFrame()

      df_group['label'] = y_val[:]
      # df_group['predicted'] = predicted[:]
      df_group['score'] = predictions[:]
      df_group['threshold'] = thresh

      accuracy = accuracy_score(y_val, prediction_label)
      #print(accuracy)
      precision = precision_score(y_val, prediction_label)
      #print(precision)
      recall = recall_score(y_val, prediction_label)
      #print(recall)
      try:
        f1 = (2 * float(precision) * float(recall)) / (float(precision) + float(recall))
      except ZeroDivisionError:
        f1 = 0

      df_stats.loc[len(df_stats)] = [thresh, accuracy, precision, recall, f1]
  df_stats['model'] = title
  return df_stats
```

```
# Combine all stats into one DataFrame to plot all of them
stats_c1m1 = compute_results_grid(feature_names_c1m1, trained_model_c1m1, predictions_c1m1, y
stats_c1m2 = compute_results_grid(feature_names_c1m2, trained_model_c1m2, predictions_c1m2, y
stats_c1m3 = compute_results_grid(feature_names_c1m2, trained_model_c1m3, predictions_c1m3, y
df_all_stats = pd.concat([stats_c1m1, stats_c1m2, stats_c1m3])
analyze_threshold_value(df_all_stats)
```

```
def plot_training_losses_all(runs):
    """
    Plot the loss curve for all three models in one figure.
```

```
    :param: feature_names: feature names used in model creation
    :param: model: trained model
    :param: X_val: validation data
    :param: y_val: validation data labels
    :param: threshold: threshold for classification
    :param: train_loss: training loss over epochs
    :param: val_loss: validation loss over epochs
    :param: title: title of plot
    """
    color_palette = ["#55C667FF","#20A387FF","#33638DFF","#481567FF"]
    _ = LinearSegmentedColormap.from_list("custom_gradient", color_palette)

    fig, axes = plt.subplots(1, 3, figsize=(18, 5))
    for ax, run in zip(axes, runs):
        train_loss = run["train_loss"]
        val_loss   = run["val_loss"]
        title      = run["title"]

        ax.plot(train_loss, label='Training Loss', marker='o', linewidth=1.5, color="#33638DI
        ax.plot(val_loss,   label='Validation Loss', marker='o', linewidth=1.5, color="#48156
        ax.set_xlabel('Epochs', fontsize = 18)
        ax.set_ylabel('Loss', fontsize = 18)
        ax.set_title(title, fontsize=18)
        ax.grid(True, alpha=0.3)
        ax.legend(loc='best')

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

```
runs = [
    {"train_loss": train_loss_c1m1, "val_loss": val_loss_c1m1, "title": "Case Study A Model (
    {"train_loss": train_loss_c1m2, "val_loss": val_loss_c1m2, "title": "Case Study A Model T
    {"train_loss": train_loss_c1m3, "val_loss": val_loss_c1m3, "title": "Case Study A Model T
]

plot_training_losses_all(runs)
```

```
def plot_confusion_matrix(models_data):
  """
  Plot the confusion matrix for all three models.
  :param: feature_names: feature names used in model creation
  :param: model: trained model
```

```
    :param: X_val: validation data
    :param: y_val: validation data labels
    :param: threshold: threshold for classification
    :param: train_loss: training loss over epochs
    :param: val_loss: validation loss over epochs
    :param: title: title of plot
    """
    color_palette = ["#55C667FF","#20A387FF","#33638DFF","#481567FF"]
    custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)

    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    for ax, m in zip(axes, models_data):
        y_true = np.asarray(m["y_true"]).astype(int).reshape(-1)
        preds  = m["predictions"]
        if hasattr(preds, "numpy"):  # TF tensor -> numpy
            preds = preds.numpy()
        preds = np.asarray(preds).reshape(-1)

        thr = float(m["threshold"])
        y_pred = (preds >= thr).astype(int)

        cm = confusion_matrix(y_true, y_pred)
        cm_norm = cm / np.sum(cm) if np.sum(cm) > 0 else cm

        sns.heatmap(cm_norm, annot=True, fmt='.2%',  annot_kws={"size": 18}, cmap=custom_cmap,
        ax.set_title(
            f"{m['title']}\nBNN "
            f"Accuracy:{accuracy_score(y_true, y_pred):.3f}, "
            f"Precision:{precision_score(y_true, y_pred, zero_division=0):.3f}, "
            f"Recall:{recall_score(y_true, y_pred, zero_division=0):.3f}",
            fontsize = 12
        )
        ax.set_xlabel('Predicted Label', fontsize = 20)
        ax.set_ylabel('True Label', fontsize = 20)
        ax.set_xticks([0.5, 1.5]); ax.set_xticklabels(['0', '1'])
        ax.set_yticks([0.5, 1.5]); ax.set_yticklabels(['0', '1'])
        ax.tick_params(axis='both', which='major', labelsize=20)
    fig.tight_layout()
    plt.show()
```

Define Thresholds Model 1 has maintained high accuracy, precision, recall and f1 values

across all of the thresholds; while Model 2 experiences an increase in precision and accuracy at 0.45, while Model 3 experiences an increase at about 0.5. Model One was unaffected by threshold so a default value of 0.5 was used.

```
thresh_c1m1 = 0.5
thresh_c1m2 = 0.5
thresh_c1m3 = 0.45

models_data = [{"y_true": y_test_c1m1, "predictions": predictions_c1m1, "threshold": thresh_c
    {"y_true": y_test_c1m2, "predictions": predictions_c1m2, "threshold": thresh_c1m2, "title
    {"y_true": y_test_c1m2, "predictions": predictions_c1m3, "threshold": thresh_c1m3, "title

plot_confusion_matrix(models_data)
```

6.5 Case Study B

Case Study B: Assigning multi-source EO active fire confidence using a probabilistic BNN.

This study uses a probabilistic BNN architecture that outputs a full probability distribution for each detection, not just a single point estimate. The resulting output includes a quantified measure of uncertainty (the standard deviation), which allows for the calculation of a confidence interval around the predicted probability. Furthermore, this case study integrates data from multiple Earth Observation sources (MODIS and VIIRS) to assess the impact of sensor differences on classification uncertainty.

Goal: The goal of Case Study B is to evaluate the null hypothesis that the uncertainty in the classification of an observed hotspot does not differ significantly in regard to the observation source. Case Study B analyzes fire hotspots detected by the VIIRS 375 m active fire product from the Suomi NPP and NOAA-20 satellites, alongside hotspots from the MODIS active fire product onboard the Aqua and Terra satellites. The data preparation methods were consistent with that of Case Study A, with the primary difference being the inclusion of standard-quality MODIS hotspots in addition to those from VIIRS. Note that instead of plotting results individually after training each model (loss curves, confusion matrices, etc.) , we present all results at the end of the case study in the “Results” section.

6.5.1 Model One

Model One uses the same architecture and hyperparameter tuning steps as Case Study A Model Three, but was trained with the updated data to contain both VIIRS and MODIS.

6.5.1.1 Create dataset

```
# Create X and y train and test data arrays
if colab_jupyter_flag == 'j':
  cs2_m1_data = os.path.join(data_dir, 'train-final-balanced-case-study-2.csv')
else:
  cs2_m1_data = 'train-final-balanced-case-study-2.csv'

X_train_c2m1, X_test_c2m1, y_train_c2m1, y_test_c2m1, feature_names_c2m1 = create_tr_test_da
train_size_c2m1 = X_train_c2m1.shape[0]
```

6.5.1.2 Create and train model

6.5.1.3 Help Box - Skip model training and tuning

Additional note if using your own input data. This help box allows the user to skip both the training and the parameter tuning steps. In order to read in a saved model file, a user must pre-define structural hyperparameters of the model being inputted. The hyperparameters selected for the given sample data set may differ form the hyperparameters that would be selected for model's trained on different input data. A comment has been added to the code indicated the line that a user must be updated to match the hyperparameters selected for the new dataset in order for the model files to load properly.

By setting the variable "read_in_file" to at the beginning of the code, the user activates sections of the code provided to allow the time intensive model training steps to be skipped. It is assumed that if a user is activating these sections of the code they have uploaded the appropriately named files to the same directory that the input data is stored in. File names are set by default to call the sample files that have provided with this tutorial.

```
if read_in_file:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy2_model1_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c2m1.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c2m1.weights.h5"
    training_output_file = "casestudy2_model1_df.csv"

  df_c2m1 = pd.read_csv(training_output_file)
  # Following hyperparameters are set in accordance with optimizing the sample data.
```

```
  # If running with personal data set update these hyperparameters to match the results of th
  reconstructed_model = create_bnn_model(train_size_c2m1, prior, feature_names_c2m1, activati
  reconstructed_model.load_weights(model_file_name)
  trained_model_c2m1 = reconstructed_model
  train_loss_c2m1 = df_c2m1['train_loss']
  val_loss_c2m1 = df_c2m1['val_loss']
```

```
if read_in_file:
  print("User has selected to read in saved models instead of training anf tu novel one. If t
else:
  models_c2m1 = []
  train_losses_c2m1 = []
  val_losses_c2m1 = []
  params_combo_c2m1 = []
  print(param_grid.keys())
  # SHouldnt need this because it is already set
  # epochs = 100
  grid = list(ParameterGrid(param_grid))
  total_runs = len(grid)
  for i, params in enumerate(grid, start =1): # creates all combinations of dictionary that i
    runs_left = total_runs - i
    print(f"[{i}/{total_runs} runs] Running with parameters: {params}")


    active_prior_distribution = params["dist"]

    model = create_bnn_model(train_size_c2m1, prior_tune, feature_names_c2m1,
                            activation_fun = params['activation'],
                            unit_dim = params['hidden_units'])

    model, train_loss, val_loss = run_experiment(model, 'binary_crossentropy', [['accuracy',
                                               feature_names_c2m1, X_train_c2m1, y_train_c2m1)

    models_c2m1.append(model)
    train_losses_c2m1.append(train_loss)
    val_losses_c2m1.append(val_loss)
    params_combo_c2m1.append(params)
    print(f'Finished run {i}/{total_runs}. Runs remaining: {runs_left}')
```

```
if read_in_file:
  print("User has selected to read in saved models instead of training and tuning novel one.
else:
```

```
  df = pd.DataFrame(params_combo_c2m1)

  # Calculate mean train and validation losses
  df["train_loss"] = [np.mean(loss) for loss in train_losses_c2m1]  # Averaging over epochs
  df["val_loss"] = [np.mean(loss) for loss in val_losses_c2m1]

  # Drop rows with missing values
  df.dropna(inplace=True)
  trained_model_c2m1 = models_c2m1[df["val_loss"].idxmin()]
  train_loss_c2m1 = train_losses_c2m1[df["val_loss"].idxmin()]
  val_loss_c2m1 = val_losses_c2m1[df["val_loss"].idxmin()]
  tuned_params = params_combo_c2m1[df["val_loss"].idxmin()]
  print(tuned_params)
```

6.5.1.4 Help Box - Saving model files

By default, once a model has been trained the notebook will export the model file and outputs from its training runs to files saved within the working directory. If the notebook is being run on Colab these files WILL NOT be saved once the Colab run is exited. If a user wishes to retain these files download them to a local directory. If these users does not care about these outputs, they will automatically be delated when the notebook is closed. If the user is running local on Jupyter the files will have to be deleted by hand.

```
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. Therefor
else:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy2_model1_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c2m1.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c2m1.weights.h5"
    training_output_file = "casestudy2_model1_df.csv"

  trained_model_c2m1.save_weights(model_file_name)
  df_c2m1 = pd.DataFrame({'train_loss': train_loss_c2m1,
                       'val_loss': val_loss_c2m1})
  df_c2m1.to_csv(training_output_file )
```

6.5.1.5 Compute predictions

```
predictions_c2m1 = compute_predictions(trained_model_c2m1, feature_names_c2m1, X_test_c2m1, y
```

6.5.1.6 Visualize training and validation loss

Visualize the training and validation loss for Case Study B Model One.

```
plot_training_loss(feature_names_c2m1, trained_model_c2m1, X_test_c2m1, y_test_c2m1, train_lo
```

6.5.2 Model Two

Model Two introduces a new architecture, a probabilistic Bayesian neural network (BNN), which outputs predictive distributions rather than single point estimates. From these distributions, we extract the mean as the predicted value and the standard deviation as a measure of model uncertainty. These predictions were grouped based on the source of the original observation (MODIS or VIIRS), and a Student's t-test was performed to compare the mean standard deviation between groups. A statistically significant result would suggest that the model's uncertainty differs depending on the data source. To further explore this, we generated histograms of the standard deviations for each group to visualize their distributions.

6.5.2.1 Create and train model

```
def create_probablistic_bnn_model(train_size, feature_names):
    """
    Create Bayesian neural network model
    :param train_size: number of samples in training dataset
    :param feature_names: list of input feature names
    :return: compiled Keras model
    """

    inputs = create_model_inputs(feature_names)

    features = tf_keras.layers.concatenate(list(inputs.values()))
    features = tf_keras.layers.BatchNormalization()(features)
    hidden_units = [8, 8]

    for units in hidden_units:
```

```
        features = tfp.layers.DenseVariational(
            units=units,
            make_prior_fn=prior,
            make_posterior_fn=posterior,
            kl_weight=1 / train_size,
            activation="relu",
        )(features)

    logits = tf_keras.layers.Dense(units=1)(features)
    outputs = tfp.layers.IndependentBernoulli(event_shape=1)(logits)

    model = tf_keras.Model(inputs=inputs, outputs=outputs)

    return model
```

6.5.2.2 Help Box - Negative Log-Likelihood Explanation

The output of our model is a probability distribution, not a point estimate. In order to understand how likely it is that our model explains the "true data," we use the negative log-likelihood as our loss function.

```
def negative_loglikelihood(targets, estimated_distribution):
    return -estimated_distribution.log_prob(targets)
```

6.5.2.3 Help Box - Skip model training

By setting the variable "read_in_file" to at the begining of the code, the user activates sections of the code provided to allow the time intensive model training steps to be skipped. It is assumed that if a user is activating these sections of the code they have uploaded the approrpiately named files to the same directory that the input data is stored in. File names are set by default to call the sample files that have provided with this tutorial.

```
if read_in_file:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy2_model2_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c2m2.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c2m2.weights.h5"
```

```
    training_output_file = "casestudy2_model2_df.csv"

  df_c2m2 = pd.read_csv(training_output_file)
  reconstructed_model = create_probablistic_bnn_model(train_size_c2m1, feature_names_c2m1)
  reconstructed_model.load_weights(model_file_name)
  trained_model_c2m2 = reconstructed_model
  train_loss_c2m2 = df_c2m2['train_loss']
  val_loss_c2m2 = df_c2m2['val_loss']
```

```
model_c2m2 = create_probablistic_bnn_model(train_size_c2m1, feature_names_c2m1)
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. If this i
else:
  trained_model_c2m2, train_loss_c2m2, val_loss_c2m2 = run_experiment(model=model_c2m2,
                                                    loss=negative_loglikelihood,
                                                    metrics=[['accuracy', tf_keras.metrics.Prec
                                                    epochs=epochs, feature_names=feature_names_
                                                    X_train=X_train_c2m1, y_train=y_train_c2m1)
```

6.5.2.4 Help Box - Saving model files

By default, once a model has been trained the notebook will export the model file and outputs from its training runs to files saved within the working directory. If the notebook is being run on Colab these files WILL NOT be saved once the Colab run is exited. If a user wishes to retain these files download them to a local directory. If these users does not care about these outputs, they will automatically be delated when the notebook is closed. If the user is running local on jupyter the files will have to be deleted by hand.

```
if read_in_file:
  print("User has selected to read in saved models instead of training a novel one. Therefor
else:
  if colab_jupyter_flag == 'j':
    # Path to files if running local using jupiter
    training_output_file = os.path.join(data_dir, "casestudy2_model2_df.csv")
    model_file_name  = os.path.join(data_dir, "model_c2m2.weights.h5")
  else:
    # Path to files if running in colab
    model_file_name = "model_c2m2.weights.h5"
    training_output_file = "casestudy2_model2_df.csv"

  trained_model_c2m2.save_weights(model_file_name)
```

```
  df_c2m2 = pd.DataFrame({'train_loss': train_loss_c2m2,
                          'val_loss': val_loss_c2m2})
  df_c2m2.to_csv(training_output_file)
```

6.5.2.5 Compute predictions

```
def compute_probablistic_predictions(model, feature_names, X_test, y_test):
    predictions = {str(feature_names[i]): X_test[:, i] for i in range(X_test.shape[1])}
    prediction_distribution = model(predictions)
    prediction_mean = prediction_distribution.mean().numpy().tolist()
    prediction_stdv = prediction_distribution.stddev().numpy()
    return prediction_mean, prediction_stdv
```

```
pred_mean_c2m2, pred_stdv_c2m2 = compute_probablistic_predictions(trained_model_c2m2, feature
predictions_c2m2 = compute_predictions(trained_model_c2m2, feature_names_c2m1, X_test_c2m1, y
```

6.5.2.6 Visualize training and validation loss

Visualize the training and validation loss for Case Study B Model Two.

```
plot_training_loss(feature_names_c2m1, trained_model_c2m2, X_test_c2m1, y_test_c2m1, train_lo
```

6.5.3 Results (All Models)

```
# Note this function only uses the first 1000 examples due to RAM limitations of Google Colab
stats_c2m1 = compute_results_grid(feature_names_c2m1, trained_model_c2m1, predictions_c2m1, y
stats_c2m2  = compute_results_grid(feature_names_c2m1, trained_model_c2m2, predictions_c2m2,

# Combine all stats into one DataFrame
df_case2_stats = pd.concat([stats_c1m3, stats_c2m1, stats_c2m2])

analyze_threshold_value(df_case2_stats)
```

Define Thresholds A threshold of 0.5 was selected for both Model 1 and Model 2 as both Models see a noticeable change with a threshold of 0.5.

```
def plot_results(feature_names, model, X_val, y_val, threshold, train_loss, val_loss, predic

    """
    Plot the loss curve and confusion matrix
    :param: feature_names: feature names used in model creation
    :param: model: trained model
    :param: X_val: validation data
    :param: y_val: validation data labels
    :param: threshold: threshold for classification
    :param: train_loss: training loss over epochs
    :param: val_loss: validation loss over epochs
    :param: title: title of plot
    """
    plt.rcParams.update({ # set visualization values
    "font.size":            15,
    "axes.titlesize":       12,
    "axes.labelsize":       15,
    "xtick.labelsize":      12,
    "ytick.labelsize":      12,
    "legend.fontsize":      10,
    "figure.titlesize":     15})
    color_palette = ["#55C667FF","#20A387FF","#33638DFF","#481567FF"]
    custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)
    # # set up array for predictions
    # predictions = {str(feature_names[i]): X_val[:, i] for i in range(X_val.shape[1])}

    # # get model predictions
    # predictions = model(predictions)

    # plot the results for a defined threshold
    predicted_classes = (predictions.numpy() >= threshold).astype(int)

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

    ax1.plot(train_loss, label='Training Loss', marker='o', color="#33638DFF")
    ax1.plot(val_loss, label='Validation Loss', marker='o', color="#481567FF")
    ax1.set_title('Training and Validation Loss Over Epochs')
    ax1.set_xlabel('Epochs', fontsize = 15)
    ax1.set_ylabel('Loss', fontsize = 15)
    #ax1.set_ylim(0, 1)
    ax1.grid()
    ax1.legend()
```

```
    cm = confusion_matrix(y_val, predicted_classes)
    sns.heatmap(cm / np.sum(cm), annot=True, annot_kws={"size": 16}, fmt='.2%', cmap=custom_c
    ax2.set_title('BNN \nAccuracy:{0:.3f}, Precision:{1:.3f}, Recall:{2:.3f} '.format(accurac
                  fontsize = 12)
    ax2.set_xlabel('Predicted Label')
    ax2.set_ylabel('True Label')

    fig.suptitle(title)
    plt.show()
```

```
def plot_probablistic_results(feature_names, model, X_test, y_test, threshold, train_loss, va
  """
  Plot the loss curve and confusion matrix
  :param: feature_names: feature names used in model creation
  :param: model: trained model
  :param: X_test: test data
  :param: y_test: test data labels
  :param: threshold: threshold for classification
  :param: train_loss: training loss over epochs
  :param: val_loss: validation loss over epochs
  :param: title: title of plot
  """
  plt.rcParams.update({ # set visualization values
    "font.size":          15,
    "axes.titlesize":     12,
    "axes.labelsize":     15,
    "xtick.labelsize":    12,
    "ytick.labelsize":    12,
    "legend.fontsize":    10,
    "figure.titlesize":   15})
  # # set up array for predictions
  # predictions = {str(feature_names[i]): X_val[:, i] for i in range(X_val.shape[1])}

  # # get model predictions
  # predictions = model(predictions)

  # plot the results for a defined threshold
  color_palette = ["#55C667FF","#20A387FF","#33638DFF","#481567FF"]
  custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)

  predicted_classes = (np.asarray(predictions) >= threshold).astype(int)
```

```python
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

ax1.plot(train_loss, label='Training Loss', marker='o', color="#33638DFF")
ax1.plot(val_loss, label='Validation Loss', marker='o', color="#481567FF")
ax1.set_title('Training and Validation Loss Over Epochs')
ax1.set_xlabel('Epochs', fontsize = 15)
ax1.set_ylabel('Loss', fontsize = 15)
ax1.grid()
ax1.legend(fontsize=10)

cm = confusion_matrix(y_test, predicted_classes)
sns.heatmap(cm / np.sum(cm), annot=True, annot_kws={"size": 16}, fmt='.2%', cmap=custom_cma
ax2.set_title('BNN \nAccuracy:{0:.3f}, Precision:{1:.3f}, Recall:{2:.3f} '.format(accuracy_
              fontsize = 12)
ax2.set_xlabel('Predicted Label')
ax2.set_ylabel('True Label')

fig.suptitle(title)
plt.show()
```

```python
thresh_c2m1 = 0.50
thresh_c2m2 = 0.50
plot_results(feature_names_c2m1, trained_model_c2m1, X_test_c2m1, y_test_c2m1, thresh_c2m1, t
plot_probablistic_results(feature_names_c2m1, trained_model_c2m2, X_test_c2m1, y_test_c2m1, t
```

```python
def predicted_class_confidence_intervals(prediction_mean, prediction_stdv, y_test):
    # The 95% CI is computed as mean ± (1.96 * stdv)
    upper = (prediction_mean + (1.96 * prediction_stdv))
    lower = (prediction_mean - (1.96 * prediction_stdv))
    diff = (upper - lower).round(4).tolist()
    prediction_stdv = prediction_stdv.tolist()
    upper = upper.round(4).tolist()
    lower = lower.round(4).tolist()

    prediction_df = pd.DataFrame({'actual': y_test,
                                'prediction_mean':  [element for innerList in prediction_mean
                                'upper_bound_95_C1': [element for innerList in upper for elem
                                'lower_bound_95_CI': [element for innerList in lower for elem
                                'interval_size': [element for innerList in diff for element i
                                'std_dev': [element for innerList in prediction_stdv for elem
                                  )
```

```
    return prediction_df
```

```
pred_df_c2m2 = predicted_class_confidence_intervals(pred_mean_c2m2, pred_stdv_c2m2, y_test_c2
print(pred_df_c2m2.head())
if colab_jupyter_flag == 'j':
  pred_df_c2m2.to_csv(os.path.join(dir_path, "pred_df_c2m2.csv"))
```

```
def compare_source_confidence_intervals(feature_names, X_test, prediction_df, comparison_feat
    # Extract source labels from input data
    feature_names = feature_names.to_numpy()
    id = np.where(feature_names=='source')[0]
    source_label= X_test[:, id].tolist()
    flatList = [element for innerList in source_label for element in innerList]
    source_df = pd.DataFrame({'source': flatList})
    # Merge it to the confidence interval df
    merged_df = pd.concat([source_df, prediction_df], axis=1)
    # Perform an independent t test to determine if there is a statistically significant dif
    source_1 = merged_df[merged_df['source']== 1.0]
    source_0 = merged_df[merged_df['source']== 0.0]
    t_stat, p_value = ttest_ind(source_1[comparison_feature], source_0[comparison_feature])
    return t_stat, p_value, source_1, source_0
```

6.5.4 Hypothesis review

```
t_stat, p_value, source_1, source_0 = compare_source_confidence_intervals(feature_names_c2m1
#print(f"p value: {round(p_value,8)}")
print(f"p value: {p_value:.8f}")
```

The p-value < 0.05, we can reject the null hypothesis and accept that the EO source does impact the standard deviation of the prediction confidence.

```
def feature_histograms(dist1,dist2, title):
  plt.hist(dist1, bins=20, color='#33638DFF', edgecolor='k', alpha=0.8, label ='source 1')
  plt.hist(dist2, bins=20, color='#481567FF', edgecolor='k', alpha=0.8, label = 'source 0')
  plt.legend(loc='upper left')
  plt.title(title)
  plt.show()
```

```
feature_histograms(source_1['std_dev'], source_0['std_dev'], 'Std Dev')
feature_histograms(source_1['prediction_mean'], source_0['prediction_mean'], 'Prediction')
print(source_1.head())
if colab_jupyter_flag == 'j':
  source_1.to_csv(os.path.join(dir_path, "source_1.csv"))
  source_0.to_csv(os.path.join(dir_path, "source_0.csv"))
```

```
def probablistic_feature_importance(trained_model, X_train, X_test, feature_names):
    def bnn_predict_fn(model, X, feature_names, n_samples=100):
        # Ensure X is a numpy array
        X = np.array(X)
        examples = {str(feature_names[i]): X[:, i] for i in range(X.shape[1])}

        predicted = []
        for _ in range(n_samples):
            predicted.append(model(examples).mean().numpy())
        return np.concatenate(predicted, axis=1)

    #plot color
    color_palette = ["#DCE319FF", "#55C667FF","#20A387FF","#33638DFF","#481567FF"]
    custom_cmap = LinearSegmentedColormap.from_list("custom_gradient", color_palette)
    # Use a subset of X_train as background data
    background = X_train[:50]

    # Pass the model itself and the feature names to the wrapper
    shap_explainer = shap.KernelExplainer(lambda x: bnn_predict_fn(trained_model, x, feature_

    # Compute SHAP values for test data
    X_test_subset = X_test[101:151]
    shap_values = shap_explainer.shap_values(X_test_subset, nsamples=500)

    # Visualize the results
    shap_values_averaged = np.mean(shap_values, axis=2)
    shap.summary_plot(shap_values_averaged, X_test_subset, feature_names=feature_names, cmap=
```

```
probablistic_feature_importance(trained_model_c2m2, X_train_c2m1, X_test_c2m1, feature_names_
```

7 Transfer Learning

This chapter is under development and will be available in a future edition of the book.

8 Fusion

This chapter is under development and will be available in a future edition of the book.

9 Downscaling

This chapter is under development and will be available in a future edition of the book.

Part III

Future of Deep Learning and Foundational Models

10 Evaluating Foundation Models Trained with Earth Observation Data

Run in Colab

Foundation models for Earth observation (EO) have the capacity to transform how critical real-world challenges are approached—from rapid flood mapping during disasters to monitoring deforestation and assessing agricultural health. By learning general-purpose representations from vast amounts of satellite imagery, these models can be quickly adapted to new tasks with minimal labeled data, dramatically reducing the time and cost of developing specialized applications. This chapter explores how to apply and evaluate these powerful tools for practical Earth science applications.

10.1 Introduction

A foundation model for Earth observation (EO) data is a large-scale machine learning model pre-trained on vast amounts of remote sensing data from various satellites and sensors, designed to learn general-purpose representations of the Earth's surface and atmospheric phenomena. These models can capture diverse features from a variety of EO datasets, such as optical, radar, thermal, and multispectral sensor imagery, and can be fine-tuned for specific tasks like environmental monitoring, land use classification, climate analysis, and disaster response.

Most current foundation models are developed using a technique called *self-supervised learning*, in which models learn useful representations by solving tasks that are derived from the structure of the input data itself, without relying on human-annotated labels. These tasks, known as *pretext tasks*, are designed to uncover internal patterns or latent features within the data distribution by creating a supervisory signal from the data alone. A common example is masked reconstruction, where parts of the input are intentionally hidden or removed, and the model is trained to predict or reconstruct the missing content based on the visible portions. This encourages the model to capture the underlying structure and context of the data. Other approaches to self-supervision include contrastive learning, where the model learns to distinguish between similar and dissimilar pairs of data, and generative methods, which learn to synthesize

or model entire input distributions. Unlike traditional supervised learning, which relies on labeled datasets, self-supervised learning does not require labeled examples (Chen et al., 2020). Self-supervised learning can be applied in many types of modeling tasks, including computer vision, and is gaining traction because it allows models to leverage vast amounts of unlabeled data whilst showing to be effective in improving performance across various domains.

Figure 1. A general self-supervised framework for remote sensing image classification, illustrating how models learn representations from unlabeled data through pretext tasks.

Source: "A General Self-Supervised Framework for Remote Sensing Image Classification" by Gao et al., 2022.

https://www.youtube.com/embed/kINuYpZroPA

Vision Transformers (ViTs) are a class of deep learning architectures that apply the transformer framework—originally developed for natural language processing—to visual data. By dividing images into fixed-size patches and treating them as a sequence of tokens, ViTs model patterns and relationships in the data using self-attention mechanisms. While ViTs are commonly used as the architectural backbone in many vision foundation models, they are not foundation models by themselves. Rather, ViTs serve as one of the key building blocks in constructing such models. ViTs require large datasets because transformers do not have the built-in inductive biases (like translation invariance) that convolutional neural networks (CNNs) have (Dosovitskiy et al., 2021). The data provided to train these models must be crafted with thoughtful sampling and stratification to ensure the correct representations are learned, and biases or artifacts reduced. Once that is ensured, it is worth considering that the effectiveness of these models greatly improves with larger datasets by provisioning more examples and greater diversity, thereby allowing the model to learn robust representations that generalize well across different tasks. More data improves the learning signal and helps the model differentiate between subtle visual cues (Chen et al., 2020).

Figure 2. Transformer-based architecture for land use classification. *Source: "Transformer-based land use and land cover classification with explainability using satellite imagery" by Khan et al., 2024*

A foundation model's extensive training regime enables its applicability across various downstream tasks. To achieve a properly trained FM, exposure to diverse and extensive data is needed in order to capture the variability and complexity inherent in real-world settings. For vision-specific foundation models, this means learning representations that can differentiate between millions of different phenomena, scenes, textures, lighting conditions, and other visual phenomena (Dosovitskiy et al., 2021). Large datasets help foundation models handle data bias and distribution shifts. For remote sensing vision tasks, biases can come from the sensor type, camera angle, geographic location, weather and lighting conditions under which images were captured (Radford et al., 2021). Training on diverse datasets that encompass multiple contexts and environments helps the model learn robust representations that generalize well across different conditions. Furthermore, training large models with millions (or billions) of parameters requires substantial data to avoid overfitting. When training vision-specific

models, such as those using ViTs as the underlying architecture or applying a self-supervised learning paradigm, having large datasets ensures that the model does not merely memorize the training data but learns generalized features that are useful across tasks (Dosovitskiy et al., 2021). Yet, while large datasets can support the learning of generalized features rather than memorization, increasing dataset size alone does not guarantee generalization—especially as models grow in parameter count. Since larger models also have a higher capacity to memorize training data, which can lead to overfitting, the task of properly sampling and splitting the data is crucial. For practitioners, this means that a biased or overfitted model could produce unreliable predictions—for example, a flood detection model trained primarily on temperate regions might fail catastrophically when applied to monsoon-affected areas in South Asia. Ensuring meaningful generalization requires careful dataset sampling and experimental design. Researchers must ensure that evaluation data (e.g., test sets) contain spatial and/or temporal slices not seen during training to mitigate data leakage and provide more robust evidence that performance stems from generalization, not memorization.

As stated, many vision foundation models rely on self-supervised learning, where they generate their own supervision signals from the data itself by solving pretext tasks—such as predicting masked patches in an image or distinguishing between augmented views. While large datasets offer the diversity needed to expose models to a broad range of visual patterns and contexts, dataset size alone is not sufficient. The quality, representativeness, and sampling strategy of the data are equally critical. Poorly sampled or imbalanced data can bias the model or lead to overfitting, especially in self-supervised setups where the supervision signal comes entirely from the data distribution. Careful attention must be paid to spatial and temporal coverage, stratification across relevant factors (e.g., class distributions, lighting conditions, sensor types), and avoidance of data leakage. High-quality, well-curated datasets not only enhance learning but also ensure that the representations learned are robust, transferable, and capable of capturing fine-grained details like textures, colors, and shapes across diverse conditions.

10.1.1 How foundation models improve efficiency in machine learning workflows

Foundation models improve efficiency in machine learning workflows by enabling tasks such as zero-shot inference, transfer learning, and general-purpose feature extraction with minimal task-specific data or training. These theoretical benefits have been demonstrated in practical applications across domains. For example, Prithvi (Prithvi-EO-1.0 and 2.0) has shown state-of-the-art performance in flood detection and above ground biomass estimation with limited labeled data (Jakubik et al., 2023 and Szwarcman et al., 2025). CLAY has demonstrated robust zero-shot generalization across different geographies and resolutions. For example, it enabled zero shot detection of deforestation events in Schroer et al., 2025. These efficiencies translate

to significant reductions in labeling costs and compute resources, especially in data-scarce or resource-constrained settings.

10.1.1.1 1. Reduced training costs

Traditional models often require training from scratch for every application, demanding significant computational resources and labeled datasets, which is especially challenging in earth observation contexts. Foundation models are pre-trained on massive datasets and can be fine-tuned for specific tasks using smaller labeled datasets, drastically reducing the costs of data annotation and training. Since these models undergo extensive pre-training on diverse datasets, they allow for the advantage of transfer learning to enable high performance with limited fine-tuning using smaller, domain-specific datasets. Suffice to say, foundation models reduce barriers to entry for users with limited resources.

Example: Instead of creating a wholly new model for deforestation detection, a pre-trained foundation model can be adapted in a fraction of the time and cost.

10.1.1.2 2. Faster development

Foundation models accelerate the development cycle by providing ready-to-use features or embeddings. This eliminates the need for extensive preprocessing and training, allowing users to focus on fine-tuning or directly deploying the model.

Example: A user working with PCA analysis on hurricane impacts can use pre-trained embeddings from a foundation model, reducing the need to engineer complex features.

10.1.1.3 3. Improved accuracy with limited data

Foundation models benefit from their extensive pre-training, making them robust and effective even when fine-tuned with limited domain-specific data. This reduces the need for expensive and time-consuming field campaigns or data collection.

Example: Foundation models trained with weather data may have learned to account for confounding weather variation that would otherwise have to be learned from the extensive collection and labeling of target features under different weather conditions.

10.1.1.4 4. Scalability across applications

A single foundation model can be repurposed for various tasks, avoiding the cost of building separate models for each application.

Example: A foundation model trained on global land cover data can be applied to soil health assessment, mangrove monitoring, or coral reef health analysis.

10.1.1.5 5. Democratizing access

With pre-trained foundation models available as open-source or through APIs, smaller organizations and researchers gain access to high-quality tools without investing in expensive infrastructure.

10.1.1.6 6. Good backbones (better than ImageNet or RGB-trained models)

Foundation models for EO are trained on specialized geospatial datasets (e.g. multi-spectral or radar satellite imagery), making their feature representations more aligned with EO tasks compared to models pre-trained on ImageNet or similar RGB datasets. EO data often includes multi-spectral bands (e.g., near-infrared) or radar data, which contain richer and more complex information than standard RGB images. - **Example:** A foundation model trained on multi-spectral imagery will perform better for tasks like vegetation health assessment than an RGB-trained ImageNet model.

10.1.1.7 7. Not as good as bespoke models, but get 90% there with 10% of the work

While foundation models may not achieve the same level of accuracy as task-specific bespoke models, they deliver competitive results with a fraction of the effort. Bespoke models demand extensive tuning, custom architectures, and domain expertise, which are costly and time-intensive. Foundation models offer a practical compromise. - **Example:** For deforestation detection, a bespoke model might achieve 95% accuracy, while a foundation model could achieve 90% accuracy with significantly less effort.

10.1.2 Limitations

All this said, foundation models are still a relatively new development in EO and have yet to see widespread validation across diverse real-world applications. Thus, while promising, their adoption is still in its early stages. Challenges like model bias, scalability, and performance in extreme conditions require further study. For instance, in precision agriculture, models may struggle with crop varieties not well-represented in training data; in urban planning, rapidly changing cityscapes may not be captured; and in polar research, the unique spectral signatures of ice and snow can confound models trained primarily on temperate regions. - **Example:** A foundation model trained on global land cover data might need more validation to ensure reliability in highly localized or unique environments.

10.1.3 Challenges

There are multiple aspects of Earth Observation Foundation Models (EOFM) that are unique to the broader scope of foundation models research. Earth observation imagery is very diverse in how it is captured. The instruments capture various frequencies at different wavelength and band widths, the ground sampling distance of the captured imagery ranges from a few centimeters to hundreds of meters, and the same location is often captured at multiple points in time. These substantial differences in the spectral, spatial, and temporal resolution make it harder for models to learn generalized patterns. In addition, earth observation data also has unique metadata characteristics that can be deterministic for the quality and the potential content of a dataset. Examples are the coordinate reference system used, the latitude and longitude location of the captured image, view angles that may be different from image to image, and data provenance artifacts such as how the data is stored and served across various providers.

Foundation models in EO would ideally capture and learn from as many of the key dimensions and metadata items as possible. An ideal model would be trained on a wide range of variation in these parameters (spatial, spectral, temporal resolutions, as well as geography, for example) during the learning process, via methods like global sampling over multiple years and seasons, as well as for a wide range of different sensors at different resolutions. Similarly, models would ideally accept a wide range of formats and image sizes during inference.

While no single model to date fully integrates all relevant dimensions of remote sensing complexity—such as temporal dynamics, multi-sensor fusion, and variable spatial and spectral resolutions—there are promising efforts in each area: for instance, Prithvi and Presto explore temporal modeling, DOFA addresses multi-sensor fusion, and Clay supports variable image resolutions and types. However, the goal need not be a single, monolithic model that captures all aspects simultaneously. In practice, combining multiple specialized models in modular or hierarchical architectures can offer a more scalable and adaptable approach. These modular

systems can collaboratively represent the rich diversity of the remote sensing domain, allowing for flexible subsetting, expansion, and fine-tuning across tasks. It's important to think creatively about architecture design, embracing hybrid strategies that align with the diverse and evolving nature of geospatial data.

> **GPU Required:** This notebook uses GPU acceleration for running foundation models. Please ensure you are using a GPU runtime (e.g., in Google Colab: Runtime → Change runtime type → GPU). If you are running low on compute units, be aware that model loading and inference steps will require GPU resources.

10.1.3.1 Using and evaluating EOFM

In the next portion, we will work through interactive demonstrations of how to use and evaluate EOFM for real domain-specific downstream tasks. Specifically, we will leverage two different EOFMs to generate embeddings for imagery before and after a natural hazard, and determine if these embeddings capture a descriptive pattern. We will also use these EOFMs in a fine tuning capacity to detect the natural hazard. After fine tuning, we'll calculate the performance scores. At the end, you will have worked through two different methods of leveraging EOFM using two different EOFM base models.

The two models we will work with include: - **Prithvi** — NASA/IBM's temporal Vision Transformer trained on Harmonized Landsat-Sentinel-2 (HLS) data, optimized for spatiotemporal analysis of Earth observation imagery. - **Clay** — A flexible EOFM supporting variable image resolutions and sensor types, designed for broad applicability across diverse remote sensing tasks.

To begin evaluating the EOFMs, we will start with the Prithvi model.

The Prithvi HLS foundation model is an EOFM developed collaboratively by IBM and NASA. It leverages a **Vision Transformer (ViT)** architecture and is pre-trained on the Harmonized Landsat-Sentinel-2 (HLS) dataset. The model incorporates a **Masked Autoencoder (MAE)** self-supervised learning approach. Its design is optimized for spatiotemporal analysis, using 3D patch embeddings to encode spatial and temporal features from input data formatted as a sequence of geospatial "videos" (`B, C, T, H, W` — bands, channels, time, height, width).

The architecture leverages temporal embeddings to model time-series dynamics and spatial embeddings to capture geospatial structure. Spectral inputs to Prithvi include six key spectral bands—Blue, Green, Red, Narrow NIR, SWIR1, and SWIR2—provided as geotiff files in reflectance units.

Figure 3. Prithvi foundation model architecture showing the masked autoencoder approach.

Source: Hugging Face - Prithvi-EO-2.0-300M

It has been fine-tuned for applications such as flood mapping, burn scar detection, and land cover classification. The pre-trained weights and fine-tuning workflows are openly available on platforms like Hugging Face, promoting accessibility and further development by the research community.

Prithvi 2.0 Citation:

```
@article{Prithvi-EO-V2-preprint,
    author          = {Szwarcman, Daniela and Roy, Sujit and Fraccaro, Paolo and Gíslason, Þo
    title           = {{Prithvi-EO-2.0: A Versatile Multi-Temporal Foundation Model for Eart
    journal         = {arXiv preprint arXiv:2412.02732},
    year            = {2024}
}
```

10.2 Run Prithvi

This exercise involves carrying out a complete analysis from beginning to end. The steps include: 1. Selecting a location and time range of interest. 2. Downloading Harmonized Landsat and Sentinel-2 (HLS) imagery for the specified parameters (make sure you have an Earth Data login and/or .netrc file for this). 3. Loading the model checkpoint. 4. Formatting the data to match the model's requirements. 5. Running the model on the prepared imagery. 6. Performing PCA and TSNE to analyze the model's output (embeddings). 7. Train a classifier on top of the model's embeddings.

Please start by installing the auxiliary libraries using the line below. This notebook requires Python version 3.10. If using Google Colab, you do not need any external requirements file, just these two lines below.

```
!pip install earthaccess>=0.14.0 pystac-client>=0.8.6 stackstac>=0.5.1 terratorch>=1.0.2
```

```
!pip install -q geopandas
# After running this cell, go to Runtime → Restart runtime, then continue
```

The imagery we will use requires authentification. As mentioned above, for this you will need to have an Earth Data login and create a .netrc file (which we will do below using a library called `earthaccess`).

```
import earthaccess
```

Calling `earthaccess.login` with `persist=True` will create a `.netrc` file for you.

```
earthaccess.login()
```

```
import os
import urllib

import geopandas as gpd
import math
import numpy as np
import pandas as pd
import pystac_client
import stackstac
import torch
import yaml
import time

from box import Box
from einops import rearrange, reduce
from matplotlib import pyplot as plt
from rasterio.enums import Resampling
from shapely.geometry import Point, mapping
from sklearn import decomposition, svm
from sklearn.manifold import TSNE
from sklearn.metrics import pairwise_distances
from torchvision.transforms import v2
from terratorch import BACKBONE_REGISTRY
```

```
os.environ["GDAL_HTTP_COOKIEFILE"] = os.path.expanduser("~/.urs_cookies")
os.environ["GDAL_HTTP_COOKIEJAR"] = os.path.expanduser("~/.urs_cookies")
os.environ["GDAL_HTTP_NETRC"] = "YES"
```

10.2.1 Define Location and Date of Interest

For this example, we analyze a region in Pakistan that experienced a monsoon flood. We will generate embeddings for the area with the model and analyze any relationships within those.

```
# Point over Padidan, Pakistan
lat, lon = 26.776567, 68.287374

# Dates surrounding a major monsoon flood (August 20, 2022)
start = "2022-06-01"
end = "2022-10-30"
```

10.2.2 Natural Disaster in Focus

The 2022 Pakistan floods stand as one of the most catastrophic flood events in recent history. Between July and August 2022, unprecedented monsoon rainfall triggered massive flooding across Pakistan, affecting approximately 33 million people and inundating much of the country. Vast agricultural lands were submerged, millions were displaced, and critical infrastructure was destroyed.

Organizations like ICIMOD played a vital role in disaster response, providing satellite-based assessments of crop losses and flood extent (ICIMOD Flood Assessment). By applying EOFMs to this use case, we demonstrate how foundation models can rapidly support flood mapping and damage assessment—capabilities that are essential for timely humanitarian response and post-disaster recovery planning.

10.2.2.1 Region of Interest: Padidan, Pakistan

Our study area is centered on Padidan in Pakistan's Sindh Province, one of the regions most severely impacted by the floods. During this period, the Indus River and its tributaries overflowed, submerging the surrounding agricultural lands and displacing local communities.

Key References: - FloodList: Pakistan Monsoon Floods August 2022 - AGU: Climate and Hydrological Analysis of the 2022 Pakistan Floods - World Bank: Pakistan Flood Damages Assessment - ICIMOD Library: Flood Assessment

This historical context underscores the importance of rapid, accurate flood mapping—a task well-suited for Earth Observation Foundation Models.

10.2.3 Retrieve Data from the STAC Catalog

Using the specified location and dates, let's obtain Harmonized Landsat and Sentinel-2 (HLS) imagery with `stackstac`. We'll parameterize our search query so as to only retrieve the desired STAC items for analysis.

```
STAC_API = "https://cmr.earthdata.nasa.gov/stac/LPCLOUD"
COLLECTION = "HLSS30.v2.0"

# Search the catalogue
catalog = pystac_client.Client.open(STAC_API)

search = catalog.search(
    collections=[COLLECTION],
    datetime=f"{start}/{end}",
    intersects=mapping(Point(lon, lat)) ,
```

```
    max_items=100,
    query={"eo:cloud_cover": {"lt": 50}},
)

all_items =  search.item_collection()
all_items
item = all_items[0]
item

# Use S3 links for downloading imagery
for item in all_items:
    for key in item.assets.keys():
        if "alternate" in item.assets[key].extra_fields:
            url = urllib.parse.urlparse(
                item.assets[key].extra_fields["alternate"]["s3"]["href"]
            )
            item.assets[key].href = f"https://{url.netloc}.s3.amazonaws.com{url.path}"
            item.assets[key].href = item.assets[key].extra_fields["alternate"]["s3"][
                "href"
            ]

items = []
dates = []
for item in all_items:
    items.append(item)
    dates.append(item.datetime.date())


print(f"Found {len(items)} items")
```

```
item.assets["B03"]
```

```
items[0]
```

10.2.4 Create a Bounding Box for the Area of Interest

We'll generate this bounding box in the projection of the dataset to ensure appropriately sized image chips.

```
# Extract coordinate system
epsg = 32642 # For Padidan, Pakistan.

# Convert point of interest into the image projection
# (assumes all images are in the same projection)
poidf = gpd.GeoDataFrame(
    pd.DataFrame(),
    crs="EPSG:4326",
    geometry=[Point(lon, lat)],
).to_crs(epsg)

# Get lat and lon in meters (UTM), where lat and lon are at the center of our intended image
coords = poidf.iloc[0].geometry.coords[0]

# Create bounds in projection
size = 256
gsd = 30
# This creates a bounding box of 256 × 30m = 7680m × 7680m (7.68 km square), centered on the
bounds = (
    coords[0] - (size * gsd) // 2,
    coords[1] - (size * gsd) // 2,
    coords[0] + (size * gsd) // 2,
    coords[1] + (size * gsd) // 2,
)
```

10.2.5 Retrieve the imagery data

```
# Retrieve the pixel values, for the bounding box in
# the target projection. In this example we use the
# RGB, NIR and SWIR bands.
# Clip the imagery to the defined bounds.
# Use nearest-neighbor resampling to avoid interpolation artifacts.
stack = stackstac.stack(
    items,
    bounds=bounds,
    snap_bounds=False,
    epsg=epsg,
    resolution=gsd,
    dtype="float64",
    rescale=False,
```

```
    fill_value=0,
    assets=["B02", "B03", "B04", "B05", "B06", "B07"],
    resampling=Resampling.nearest,
)

print(f"Working with stack of size {stack.shape}")

stack = stack.compute()

stack
```

10.2.6 Review the Downloaded Imagery

The imagery dataset contains four pre-flood images two during-flood images (one of which is cloudy) and 12 post-flood images.

```
stack.sel(band=["B04", "B03", "B02"]).plot.imshow(
    row="time", rgb="band", vmin=0, vmax=2000, col_wrap=6
)
```

```
stack.shape
```

10.2.7 Load the Model

Now that we have the imagery prepared, the next step is to load the model for analysis.

Estimated Time: Model loading typically takes 2-5 minutes depending on your internet connection, as the model weights (~1.3 GB) are downloaded from Hugging Face.

Hugging Face Authentication (Optional but Recommended): If you encounter authentication errors or timeouts when loading the model, you can authenticate with Hugging Face using one of these methods:

1. **Using `huggingface-cli` (recommended):**

   ```
   !pip install -q huggingface_hub
   !huggingface-cli login
   ```

   This will prompt you to enter your Hugging Face token. You can create a token at https://huggingface.co/settings/tokens.

2. **Using Google Colab Secrets:** Add your Hugging Face token as a secret named `HF_TOKEN` in Colab (click the key icon in the left sidebar), then restart your session.

3. **Programmatic login:**

```
from huggingface_hub import login
login(token="your_token_here")
```

```
# Find and print available Prithvi models in the model registry
print([model_name for model_name in BACKBONE_REGISTRY if "terratorch_prithvi" in model_name])

# Check that the model we want to use is in the registry
"terratorch_prithvi_eo_v2_300" in BACKBONE_REGISTRY

# instantiate our desired model
# NOTE: the backbone registry prefix (e.g., `terratorch`) is optional
model = BACKBONE_REGISTRY.build("prithvi_eo_v2_300", pretrained=True)
```

10.2.8 Format Band Pixel Data for the Model

Now we will transform the imagery stack into the format required by the model. Prithvi uses short time series of four images, “quadruplets,” as input. Due to limited data, overlapping dates are used to create quadruplets: the first spans dates 1 to 4, the second covers dates 2 to 5, and so forth.

```
input_size = [1,size,size]
patch_size = [1,16,16]
```

```
chips = []
for i in range(19):
    chips.append(stack.isel(time=slice(i, i + 4)).values)
chips[0].shape
```

```
chips = []
# Calculate number of valid quadruplets (sequences of 4 timesteps)
num_chips = max(0, stack.shape[0] - 3)  # stack.shape[0] is number of timesteps
print(f"Creating {num_chips} chips from {stack.shape[0]} timesteps")

for i in range(num_chips):
    chips.append(stack.isel(time=slice(i, i + 4)).values)
chips[0].shape
```

```
len(chips)
```

We get **n** chips to represent the number combinations of temporal sequences where the time series length is 4.

```
chip_time_indices = [[i, i+1, i+2, i+3] for i in range(num_chips)]
```

```
len(chip_time_indices)
```

```
chip_time_indices
```

10.2.9 Execute the Model

Now we pass the prepared datacube to the model to generate one embedding vector for each image.

```
embeddings = []
start_time = time.time()

# Iterate over each chip (sequence of 4 timesteps)
for ts in chips:
    # Rearrange tensor to match model input format: add batch dimension and put time in the r
    ts = rearrange(ts, "t c h w -> 1 c t h w")
    ts = ts.astype(np.float32)
    # Only process chips with 4 time steps (quadruplets)
    if ts.shape[2] == 4:
        embedding = model.forward(torch.from_numpy(ts)) # Run model to get embeddings
        # Extract the class token (represents a summary or global representation of the enti
        # `:` means all batches - here, size 1, `0` means the first token (the CLS token), `
        # so the shape of cls_embedding before unraveling is (batch_size, embedding_dim), and
        cls_embedding = embedding[0][:, 0, :].detach().cpu().numpy().ravel()
        embeddings.append(cls_embedding)
    else:
        print(f"Skipping chip with {ts.shape[2]} time steps")

embeddings = np.array(embeddings)
print(f"Generated {len(embeddings)} embeddings in {time.time() - start_time:.1f} seconds")
```

```
embeddiwngs[0].shape
```

```
len(embeddings)
```

10.2.10 Analyze the Embeddings

Now we will run a simple analysis to find any patterns in the model's learned representations of the data. PCA is applied to reduce each embedding to a single value. Embeddings, whether class or patch level, from models like transformers are usually high-dimensional vectors (e.g., 768, 1024 dims). PCA reduces this to fewer dimensions (e.g., 1 or 2) for easier analysis or visualization. By projecting embeddings into 2D with PCA, you can visualize how samples relate to each other—whether they cluster by class, time, or some other property. This is because PCA extracts principal components that capture the largest variance in the embeddings, potentially removing noise or irrelevant features and focusing on the most important aspects. As such, lower-dimensional representations from PCA can be used as input features for clustering, classification, or anomaly detection algorithms with less computational cost and potentially better generalization.

The PCA-reduced embeddings, as you'll see, appear to be grouped into four categories within PCA space: pre-flood images, during flood images, cloudy images, and post-flood images.

```
# Define flood phase indices for visualization
# These indices correspond to the temporal position of images in the stack
PRE_FLOOD_IDX = slice(0, 5)      # Images before the flood
FLOOD_IDX = slice(5, 9)          # Images during active flooding
POST_FLOOD_IDX = slice(9, None)  # Images after flood waters receded
CLOUDY_IDX = 7                   # Index of cloudy image

pca = decomposition.PCA(n_components=1)
pca_result = pca.fit_transform(embeddings)

plt.figure(figsize=(10, 5))
plt.xticks(rotation=-45)

# Plot points by flood phase
plt.scatter(stack.time[PRE_FLOOD_IDX], pca_result[PRE_FLOOD_IDX], color="blue", label="Pre-fl
plt.scatter(stack.time[FLOOD_IDX], pca_result[FLOOD_IDX], color="red", label="During flood",
plt.scatter(stack.time[CLOUDY_IDX], pca_result[CLOUDY_IDX], color="green", label="Cloudy", s=
plt.scatter(stack.time[POST_FLOOD_IDX][:num_chips-9], pca_result[POST_FLOOD_IDX], color="oran

plt.xlabel("Date")
```

```
plt.ylabel("PCA Component 1")
plt.title("PCA Projection of Prithvi Embeddings Over Time")
plt.legend()
plt.tight_layout()
plt.show()
```

10.2.11 Embeddings for Each Time Step

The model also generates embeddings for individual time steps. These embeddings are extracted for each input quadruplet and visualized using PCA, revealing similar clustering patterns.

The first version generates an embedding for each timesep in each quadruplet, which encounters some redundancy as a given timestep is inherently present in multiple quadruplets.

```
long_embeddings_quadruplets = []

# Iterate over each chip (sequence of 4 timesteps)
for ts in chips:
    # Rearrange tensor to match model input format: add batch dimension and put time in the r
    ts = rearrange(ts, "t c h w -> 1 c t h w")
    ts = ts.astype(np.float32)
    # Only process chips with 4 time steps (quadruplets)
    if ts.shape[2] == 4:
        embedding = model.forward(torch.from_numpy(ts)) # Run model to get embeddings
        # Extract class token (embedding vector for the whole chip (4 timesteps combined))
        cls_embedding = embedding[0][:, 0, :].detach().cpu().numpy().ravel()
        # Reshape to (t, n, d): time, patch tokens, embedding dim
        embedding = rearrange(embedding[0][:, 1:, :], "1 (t n) d -> 1 t n d", t=4)[0]
        # Average patch tokens per timestep: shape becomes (t, d)
        # gives us one averaged patch embedding per timestep
        embedding = reduce(embedding, "t n d -> t d", "mean").detach().numpy()
        # Store the embedding quadruplet
        t0, t1, t2, t3 = embedding
        long_embeddings_quadruplets.extend([t0] + [t1] + [t2] + [t3])
```

```
len(long_embeddings_quadruplets)
```

This second version stores only one embedding for a given timestep (appends the embedding based on its time index if that time index has not already been added).

```python
long_embeddings_dict = {}

# Iterate over each chip (sequence of 4 timesteps) and its index
for i, ts in enumerate(chips):
    time_ids = chip_time_indices[i]  # Global timestep indices for this chip, e.g., [0, 1, 2
    # Rearrange tensor to match model input format: add batch dimension and put time in the r
    ts = rearrange(ts, "t c h w -> 1 c t h w").astype(np.float32)

    # Only process chips with 4 time steps (quadruplets)
    if ts.shape[2] == 4:
        with torch.no_grad():
            embedding = model.forward(torch.from_numpy(ts)) # Run model to get embeddings

        # Extract CLS token, reshape to (t, n, d): time, patch tokens, embedding dim
        embedding = rearrange(embedding[0][:, 1:, :], "1 (t n) d -> 1 t n d", t=4)[0]
        # Average patch tokens per timestep: shape becomes (t, d)
        # gives us one averaged patch embedding per timestep
        embedding = reduce(embedding, "t n d -> t d", "mean").detach().numpy()

        # Store each timestep's embedding using the original (global) time index
        for j, t_idx in enumerate(time_ids):
            if t_idx not in long_embeddings_dict:
                long_embeddings_dict[t_idx] = embedding[j]

# Reconstruct a list of embeddings ordered by all 22 possible timesteps (if available)
all_timesteps = list(range(22))
long_embeddings = [long_embeddings_dict[t] for t in all_timesteps if t in long_embeddings_dic
```

```python
len(long_embeddings)
```

```python
# Run PCA
pca = decomposition.PCA(n_components=1)
pca_result = pca.fit_transform(long_embeddings)

plt.figure(figsize=(10, 5))
plt.xticks(rotation=-45)

# Plot points by flood phase
plt.scatter(stack.time[PRE_FLOOD_IDX], pca_result[PRE_FLOOD_IDX], color="blue", label="Pre-fl
plt.scatter(stack.time[FLOOD_IDX], pca_result[FLOOD_IDX], color="red", label="During flood",
plt.scatter(stack.time[CLOUDY_IDX], pca_result[CLOUDY_IDX], color="green", label="Cloudy", s=
plt.scatter(stack.time[POST_FLOOD_IDX], pca_result[POST_FLOOD_IDX], color="orange", label="Po
```

```
plt.xlabel("Date")
plt.ylabel("PCA Component 1")
plt.title("PCA Projection of Per-Timestep Embeddings Over Time")
plt.legend()
plt.tight_layout()
plt.show()
```

10.2.11.1 Visualizing Flood Patterns with t-SNE

While PCA provides a linear dimensionality reduction, t-SNE (t-distributed Stochastic Neighbor Embedding) captures non-linear relationships in the embedding space. t-SNE is particularly effective at revealing clustering patterns by preserving local neighborhood structure—making it well-suited for visualizing how the model distinguishes between different flood states (see Figure 5).

In the context of our flood analysis, t-SNE helps us understand whether the foundation model's embeddings naturally separate pre-flood conditions (e.g. normal agricultural land, urban areas and water bodies) from active flooding (e.g. inundated areas, turbid water) and post-flood recovery (e.g. receding waters, sediment deposits). If the model has learned meaningful representations of these hydrological states, we expect to see distinct clusters corresponding to each phase of the flood event.

```
long_embeddings_array = np.array(long_embeddings)

tsne = TSNE(n_components=2, random_state=42, perplexity=min(5, len(long_embeddings)-1))
tsne_result = tsne.fit_transform(long_embeddings_array)

plt.figure(figsize=(8, 6))
plt.scatter(tsne_result[PRE_FLOOD_IDX, 0], tsne_result[PRE_FLOOD_IDX, 1], color='blue', label
plt.scatter(tsne_result[FLOOD_IDX, 0], tsne_result[FLOOD_IDX, 1], color='red', label='During
plt.scatter(tsne_result[CLOUDY_IDX, 0], tsne_result[CLOUDY_IDX, 1], color='green', label='Cl
plt.scatter(tsne_result[POST_FLOOD_IDX, 0], tsne_result[POST_FLOOD_IDX, 1], color='orange', 
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.title('t-SNE Projection of Prithvi Embeddings')
plt.legend()
plt.tight_layout()
plt.show()
```

10.2.11.2 Interpretation: Prithvi Embeddings

The PCA and t-SNE visualizations above reveal that Prithvi's embeddings capture meaningful distinctions between flood phases. Pre-flood images (blue) cluster separately from active flooding images (red), indicating the model has learned to differentiate normal land conditions from storm-affected areas. The post-flood images (orange/yellow) form their own cluster. The cloudy image (green) appears as an outlier, demonstrating that the model's embeddings are also sensitive to atmospheric interference—an important consideration for operational flood mapping where cloud contamination is common.

10.2.12 Train a Model using the Embeddings as Features

Now we will train a classifier head for detecting flood events on top of the embeddings. Our dataset is tiny (22 samples), so we will choose a suitable method: Support Vector Machine. An SVM is a supervised machine learning model that tries to find the best boundary (or boundaries) between different classes of data in a feature space. At this point, we're treating the EOFM as a feature extractor (fixed), and only training a small classifier (the SVM) on top. Given that, SVM fits well — it learns fast and doesn't overfit as easily as deeper classifiers.

```
# Label the images we downloaded
# 0 = Cloud
# 1 = Pre-flood
# 2 = During-flood
# 3 = Post-flood

labels = np.array([1, 1, 1, 1, 1, 2, 2, 0, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])

# Split into fit and test manually, with an effort to have all 3 classes in both sets
# (since there is only one cloudy sample, we use it for training only)
fit = [0, 2, 4, 6, 7, 8, 10, 12, 14, 16, 18, 20]
test = [1, 3, 5, 9, 11, 13, 15, 17, 19, 21]

# Train a support vector machine model
long_embeddings = np.array(long_embeddings)
clf = svm.SVC()
clf.fit(long_embeddings[fit], labels[fit])

# Predict classes on test set
prediction = clf.predict(long_embeddings[test])

# Match for SVM
```

```
match = np.sum(labels[test] == prediction)
print(f"Matched {match} out of {len(test)} correctly")
```

10.3 Clay

The Clay Foundation model is another open source EOFM. It also utilizes a **Vision Transformer (ViT)** architecture with a **masked autoencoder** approach. The encoder of the model consists of several layers, each with multiple attention heads, allowing it to capture complex spatial and temporal relationships in the data. The self-attention mechanism enables it to understand both local and global features within satellite imagery. The decoder is structured similarly but with fewer layers and attention heads, and it is optimized for efficient reconstruction during the pre-training phase. The architecture uses an MLP (Multi-Layer Perceptron) layer for feature extraction.

The model is trained on a large dataset that combines multi-modal geospatial data from Landsat 8 and 9, Sentinel-2 L2A, Sentinel-1 RTC, NAIP and LINZ. A sampling strategy carefully curated to cover a variety of land types was implemented, ensuring diverse geographical and seasonal representation.

The architecture and data inputs make Clay highly flexible.

For more details, you can visit the official Clay Foundation website or the documentation website.

Let's get the necessary code for Clay.

```
!mkdir clay
```

```
cd clay
```

```
!git clone --depth=1 https://github.com/Clay-foundation/model
```

```
cd model
```

```
ls
```

```
import sys
sys.path.append("./claymodel/")
from claymodel.module import ClayMAEModule
```

The tile size for Prithvi was 256. For Clay, it is 224, so we will need to re-create our stacked dataset using that value.

```
# Create bounds in projection
size = 224
gsd = 30
bounds = (
    coords[0] - (size * gsd) // 2,
    coords[1] - (size * gsd) // 2,
    coords[0] + (size * gsd) // 2,
    coords[1] + (size * gsd) // 2,
)

# Retrieve the pixel values, for the bounding box in
# the target projection. In this example we use the
# RGB, NIR and SWIR bands.
stack = stackstac.stack(
    items,
    bounds=bounds,
    snap_bounds=False,
    epsg=epsg,
    resolution=gsd,
    dtype="float64",
    rescale=False,
    fill_value=0,
    assets=["B02", "B03", "B04", "B05", "B06", "B07"],
    resampling=Resampling.nearest,
)

print(f"Working with stack of size {stack.shape}")

stack = stack.compute()

stack
```

10.3.1 Load the Model

We now have the data to analyse, let's load the model. But first, we need to rename the bands to match what's expected in the config file.

```
stack.band.values
```

```
if "band" in stack.coords:
    stack = stack.assign_coords(band=["red" if b == "B04" else b for b in stack.band.values])
```

```
    stack = stack.assign_coords(band=["green" if b == "B03" else b for b in stack.band.values
    stack = stack.assign_coords(band=["blue" if b == "B02" else b for b in stack.band.values]
    stack = stack.assign_coords(band=["nir" if b == "B05" else b for b in stack.band.values])
    stack = stack.assign_coords(band=["swir16" if b == "B06" else b for b in stack.band.valu
    stack = stack.assign_coords(band=["swir22" if b == "B07" else b for b in stack.band.valu
```

Again, here you have the option to load the model the following way or with the HuggingFace library.

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
ckpt = "https://huggingface.co/made-with-clay/Clay/resolve/main/v1.5/clay-v1.5.ckpt"
torch.set_default_device(device)

model = ClayMAEModule.load_from_checkpoint(
    ckpt,
    model_size="large",
    metadata_path="./configs/metadata.yaml",
    mask_ratio=0.0,
    shuffle=False,
)
model.eval()

model = model.to(device)
```

10.3.2 Prepare Band Metadata for Model Input

This step is the most technical so far. It involves processing the imagery stack and converting its data into the format required by the model. This includes normalizing the latitude/longitude and the imagery's date values.

The Clay model is designed to accept any combination of bands in any order, even from different platforms. However, the model requires metadata about each band, including its central wavelength and normalization parameters. This information allows the model to standardize the data and correctly interpret each band based on its wavelength.

For Sentinel-2, we can extract these details from the model's metadata file. For other platforms, a custom approach may be needed to provide this information. For simplicity here, we utilize the Sentinel-2 parameters for our HLS imagery.

```
# Extract mean, std, and wavelengths from metadata
platform = "sentinel-2-l2a"
metadata = Box(yaml.safe_load(open("configs/metadata.yaml")))
```

```
mean = []
std = []
waves = []
# Use the band names to get the correct values in the correct order.
for band in stack.band:
    mean.append(metadata[platform].bands.mean[str(band.values)])
    std.append(metadata[platform].bands.std[str(band.values)])
    waves.append(metadata[platform].bands.wavelength[str(band.values)])

# Prepare the normalization transform function using the mean and std values.
transform = v2.Compose(
    [
        v2.Normalize(mean=mean, std=std),
    ]
)
```

10.3.3 Convert Band Pixel Data to Model Format

This step involves transforming the imagery stack into the format required by the model. It includes normalizing the latitude/longitude and the imagery's date values.

```
# Prep datetimes embedding using a normalization function from the model code.
def normalize_timestamp(date):
    week = date.isocalendar().week * 2 * np.pi / 52
    hour = date.hour * 2 * np.pi / 24

    return (math.sin(week), math.cos(week)), (math.sin(hour), math.cos(hour))


datetimes = stack.time.values.astype("datetime64[s]").tolist()
times = [normalize_timestamp(dat) for dat in datetimes]
week_norm = [dat[0] for dat in times]
hour_norm = [dat[1] for dat in times]


# Prep lat/lon embedding using a normalization function from the model code.
def normalize_latlon(lat, lon):
    lat = lat * np.pi / 180
    lon = lon * np.pi / 180

    return (math.sin(lat), math.cos(lat)), (math.sin(lon), math.cos(lon))
```

```
latlons = [normalize_latlon(lat, lon)] * len(times)
lat_norm = [dat[0] for dat in latlons]
lon_norm = [dat[1] for dat in latlons]

# Normalize pixels
pixels = torch.from_numpy(stack.data.astype(np.float32))
pixels = transform(pixels)
```

10.3.4 Combine Metadata and Pixels

All required inputs, including metadata and transformed pixel values, are compiled into a single dictionary.

```
# Prepare additional information
datacube = {
    "platform": platform,
    "time": torch.tensor(
        np.hstack((week_norm, hour_norm)),
        dtype=torch.float32,
        device=device,
    ),
    "latlon": torch.tensor(
        np.hstack((lat_norm, lon_norm)), dtype=torch.float32, device=device
    ),
    "pixels": pixels.to(device),
    "gsd": torch.tensor(stack.rio.resolution()[0], device=device),
    "waves": torch.tensor(waves, device=device),
}
```

10.3.5 Execute the Model

The datacube is passed to the model, which generates one embedding vector per image.

```
import time
start_time = time.time()

with torch.no_grad():
    unmsk_patch, unmsk_idx, msk_idx, msk_matrix = model.model.encoder(datacube)
```

```
# The first embedding is the class token, which is the
# overall single embedding. We extract that for PCA below.
embeddings = unmsk_patch[:, 0, :].cpu().numpy()

print(f"Generated {len(embeddings)} embeddings in {time.time() - start_time:.1f} seconds")
```

```
len(embeddings)
```

10.3.6 Analyze the Embeddings

A straightforward way to analyze the embeddings is by reducing each to a single value using Principal Component Analysis (PCA). This involves fitting a PCA model to the embeddings and performing dimensionality reduction. The resulting PCA space reveals three distinct groups: earlier images, cloudy images, and post-flood images, each occupying a different range within the PCA space.

```
# Run PCA
pca = decomposition.PCA(n_components=1)
pca_result = pca.fit_transform(embeddings)

plt.figure(figsize=(10, 5))
plt.xticks(rotation=-45)

# Plot points by flood phase
plt.scatter(stack.time[PRE_FLOOD_IDX], pca_result[PRE_FLOOD_IDX], color="blue", label="Pre-fl
plt.scatter(stack.time[FLOOD_IDX], pca_result[FLOOD_IDX], color="red", label="During flood",
plt.scatter(stack.time[CLOUDY_IDX], pca_result[CLOUDY_IDX], color="green", label="Cloudy", s=
plt.scatter(stack.time[POST_FLOOD_IDX], pca_result[POST_FLOOD_IDX], color="orange", label="Po

plt.xlabel("Date")
plt.ylabel("PCA Component 1")
plt.title("PCA Projection of Clay Embeddings Over Time")
plt.legend()
plt.tight_layout()
plt.show()
```

```
tsne = TSNE(n_components=2, random_state=42, perplexity=min(5, len(embeddings)-1))
tsne_result = tsne.fit_transform(embeddings)

plt.figure(figsize=(8, 6))
```

```
plt.scatter(tsne_result[PRE_FLOOD_IDX, 0], tsne_result[PRE_FLOOD_IDX, 1], color='blue', label
plt.scatter(tsne_result[FLOOD_IDX, 0], tsne_result[FLOOD_IDX, 1], color='red', label='During
plt.scatter(tsne_result[CLOUDY_IDX, 0], tsne_result[CLOUDY_IDX, 1], color='green', label='Cl
plt.scatter(tsne_result[POST_FLOOD_IDX, 0], tsne_result[POST_FLOOD_IDX, 1], color='orange',
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.title('t-SNE Projection of Clay Embeddings')
plt.legend()
plt.tight_layout()
plt.show()
```

10.3.6.1 Interpretation: Clay Embeddings

Similar to Prithvi, Clay's embeddings show clear separation between flood phases in both PCA and t-SNE projections. The model successfully distinguishes pre-flood conditions from active flooding and post-flood recovery states. Notably, Clay processes each timestep independently (unlike Prithvi's temporal quadruplets), yet still captures the flood signal effectively. This suggests that the spectral and spatial patterns of flooding are sufficiently distinctive that even single-image embeddings can support flood detection. The consistency between Prithvi and Clay results provides confidence that the observed patterns reflect genuine flood signatures rather than model-specific artifacts.

10.3.7 Train a Model using the Embeddings as Features

Finally, a classifier is again trained on the embeddings to identify flood events.

```
# Label the images we downloaded
# 0 = Cloud
# 1 = Pre-flood

# 2 = During-flood
# 3 = Post-flood

labels = np.array([1, 1, 1, 1, 1, 2, 2, 0, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])

# Split into fit and test manually, with an effort to have all 3 classes in both sets
# (since there is only one cloudy sample, we use it for training only)
fit = [0, 2, 4, 6, 7, 8, 10, 12, 14, 16, 18, 20]
test = [1, 3, 5, 9, 11, 13, 15, 17, 19, 21]
```

```
# Train a Support Vector Machine model
clf = svm.SVC()
clf.fit(embeddings[fit], labels[fit])

# Predict classes on test set
prediction = clf.predict(embeddings[test])

# Perfect match for SVM
match = np.sum(labels[test] == prediction)
print(f"Matched {match} out of {len(test)} correctly")
```

```
labels[test]
```

```
prediction
```

10.3.8 Change Detection Heatmap (Pre- vs. Post-Flood Distances)

We can generate a heatmap to visualize pairwise distances between embeddings from pre-flood and post-flood periods. Each cell's value represents the distance between a specific pre-flood and post-flood embedding.

We will also compute the average distance. A high average distance suggests that the model identifies significant differences between pre- and post-flood conditions. This could reflect meaningful changes in the landscape.

As for the heatmap, pattern consistency is key to look for. Uniformity in the heatmap (similar distances across all pairs) indicates a consistent model response to changes. Non-uniformity might suggest localized variations. Smaller distances between certain pairs might highlight areas unaffected by the event or areas with similarities across both periods.

Notice that the row impacted by the greatest distance includes the cloudy image at index 8. This makes sense as the cloudy image will bear greater dissimilarity to the other embeddings, both pre- and post-flood.

```
# Embeddings from pre- and post-flood
pre_event_embeddings = embeddings[0:5]
post_event_embeddings = embeddings[9:]

# Calculate pairwise distances between pre- and post-event embeddings
# This computes a matrix of distances between every pre-event embedding and every post-event
distance_matrix = pairwise_distances(pre_event_embeddings, post_event_embeddings)
# We're quantifying how much the embedding space has changed, assuming that semantic changes
```

```
# Get the average of the distance matrix
avg_distance = np.mean(distance_matrix)
print(f"Average embedding distance between pre- and post-event: {avg_distance:.4f}")
# One number representing how different the pre/post embeddings are overall.

# Visualize the differences
plt.imshow(distance_matrix, cmap="viridis", aspect="auto")
plt.colorbar(label="Embedding Distance")
plt.title("Pre- vs. Post-Event Embedding Distances")
plt.xlabel("Post-Event Embeddings")
plt.ylabel("Pre-Event Embeddings")
plt.show()
```

It might be helpful to look once more at the images chronologically ordered after visualizing the graph above.

```
stack.sel(band=["red", "green", "blue"]).plot.imshow(
    row="time", rgb="band", vmin=0, vmax=25000, col_wrap=6
)
```

10.3.9 Discussion

10.3.9.1 How do I know when EOFMs are the right tool?

EOFMs are particularly suitable when:

- You have **limited labeled data**
- Your task requires **generalizable spatial or temporal reasoning**
- You want to **transfer models across regions or sensors**
- You seek to build **multi-task pipelines** (e.g., classification + segmentation + anomaly detection)

They are less appropriate when:

- You have abundant labels and simple classification tasks
- Computational cost is a constraint
- Model interpretability is the top priority

10.3.9.2 How does the use of EOFMs compare to the performance of other approaches?

10.3.9.2.1 Zero-shot

This means of use is addressed in the chapter. Zero-shot performance (without task-specific training) gives a baseline for how well the pretrained EOFM has generalized from pretraining to your task domain. This is a strength of Prithvi and CLAY in particular, as they can encode rich geospatial patterns directly from input data.

10.3.9.2.2 Fine-tuning

Fine-tuning should be benchmarked alongside zero-shot usage in practice when using a specialized dataset, or when training for downstream tasks. Fine-tuning can enable domain-specific adaptation and often improves accuracy, particularly in edge cases or for features not well represented in pretraining. This comes with increased computational cost and overfitting risk if data is limited.

10.3.9.2.3 Simpler methods

It's essential to benchmark EOFMs against simpler baselines like:

- Classical remote sensing indices + Random Forest
- SVM
- Spectral-only MLPs
- Convolutional Neural Networks

These comparisons help quantify whether the added complexity of a foundation model is justified for your task.

10.3.9.2.4 Other feature reduction techniques

Other unsupervised methods like pure PCA, autoencoders, or temporal summaries may be compared to EOFM embeddings when used as input to downstream classifiers. This allows us to assess whether EOFMs truly encode more discriminative or transferable features than conventional feature engineering pipelines.

10.3.9.3 How does the performance of different models compare relative to each approach's resource consumption?

Model Type	Performance	Compute Cost	Training Data Needed	Suitability
EOFM (Prithvi/CLAY)	High (esp. zero/few shot)	High	Low–Moderate	Complex, generalizable tasks
Random Forest + Indices	Medium	Low	Moderate	Simple tasks
CNN from scratch	Medium–High	Medium–High	High	Supervised tasks with many labels
PCA + SVM	Low–Medium	Very Low	Moderate	Baselines, small-scale tasks

---

10.3.9.4 What kinds of interpretability can be done?

- **Attention maps** (e.g., from ViT-based EOFMs like Prithvi and Clay)
- **Similarity search** using embedding distances (semantic matching)
- **Visualization of embedding spaces** (UMAP/t-SNE for EO tasks)

For Prithvi and CLAY, embedding-based analysis (e.g., clustering, retrieval, similarity) is particularly powerful, though attribution methods are still emerging.

10.3.9.5 How can I use the information above to ensure I choose the right method?

Choosing between methods depends on:

- **Label availability** (EOFMs are strong when labels are scarce)
- **Task complexity** (fine-grained or abstract tasks benefit more from EOFMs)
- **Computation available** (EOFMs are heavier; classical methods are leaner)
- **Performance delta** (if EOFMs outperform simple methods only marginally, simpler methods may be preferable)

10.3.9.6 How can we stratify train, test, and validation to ensure proper evaluation?

- **Spatial stratification**: sample by region or tile, carefully to avoid spatial leakage
- **Temporal stratification**: ensure held-out time stamps in test set
- **Environmental conditions**: vary climate, terrain, and seasonality
- **Stratified sampling**: ensure class balance in all sets (important for rare class detection)

Avoid random pixel-based splits—these can overestimate performance due to spatial autocorrelation.

10.3.9.7 How can we evaluate at both the tile and map level to ensure proper continuity and qualitative "reasonability" of the output?

- **Tile-level metrics**: per-tile accuracy, F1, confusion matrices
- **Map-level coherence**: visually inspect maps for spatial discontinuities, fragmentation, or unreasonable transitions
- **Temporal consistency**: ensure predictions are temporally smooth where expected (e.g., gradual seasonal change)

Use moving window accuracy checks and expert review of map artifacts. EOFMs can occasionally "hallucinate" features if the embedding has been poorly grounded.

10.3.9.8 What to do when one's label set is limited?

- **Use EOFM embeddings as features**: they encode high-level patterns without requiring supervised training
- **Leverage weak supervision**: use coarsely labeled data
- **Few-shot fine-tuning**: tune only the last layer with a handful of labels
- **Self-training or pseudo-labeling**: train on confident predictions
- **Domain adaptation**: try transferring from well-labeled regions to unlabeled ones.

10.3.10 Conclusion

In this notebook, we demonstrated how to evaluate two Earth Observation Foundation Models—Prithvi and Clay—on the 2022 Pakistan floods, one of the most catastrophic flood events in recent history. Through these examples, we illustrated how EOFMs can generate meaningful embeddings from satellite imagery that capture distinct hydrological states: pre-flood land conditions, active flooding with inundated lands and turbid water, and post-flood recovery patterns.

The key insight is that these embeddings—learned representations of the imagery—encode rich semantic information without requiring manually labeled training data. In traditional flood mapping workflows, analysts must either manually delineate flood boundaries or train supervised classifiers on laboriously labeled datasets for each new event. EOFMs fundamentally change this pattern: because the models have already learned general-purpose representations of Earth's surface from vast amounts of satellite data, they can produce embeddings that distinguish flooded from non-flooded areas. As we showed, a simple classifier (like an SVM) trained on just a handful of labeled embeddings can achieve strong flood detection performance. This means disaster response teams can go from raw satellite imagery to preliminary flood maps in hours rather than days—critical time savings when lives and livelihoods are at stake.

For practitioners in flood monitoring and water resources management, this workflow offers a practical template: acquire imagery, generate embeddings with a pre-trained EOFM, and apply lightweight classification or clustering to produce actionable maps. As these foundation models continue to mature, their integration into operational flood forecasting and damage assessment systems will become increasingly valuable for climate adaptation efforts worldwide.

10.3.10.1 Next Steps

To build on the techniques demonstrated in this notebook, consider exploring:

- **Fine-tuning for flood segmentation:** Instead of using embeddings with a simple classifier, fine-tune the model with a segmentation head to produce pixel-level flood masks.
- **Multi-temporal change detection:** Leverage the temporal capabilities of models like Prithvi to detect flood onset and recession dynamics across longer time series.
- **Transfer to other disasters:** Apply the same embedding-based workflow to other natural hazards such as wildfires, landslides, or drought monitoring.
- **Operational integration:** Explore how to integrate EOFM-based flood mapping into existing disaster response pipelines, including near-real-time satellite data feeds.
- **Uncertainty quantification:** Investigate methods to estimate confidence in flood predictions, which is critical for decision-making in humanitarian response.

11 Ethics and Artificial Intelligence

This chapter is under development and will be available in a future edition of the book.

Conclusions

This chapter is under development and will be available in a future edition of the book.

References