**Problem Statement**  
Wildfires cause significant ecological and economic damage. Early detection and accurate forecasting are critical for mitigation. Traditional methods (e.g., human patrols) are slow and resource-intensive.

**Proposed Solution**  
A three-stage Machine Learning pipeline:

1. **Classification**: Detects fires in satellite imagery.
2. **Segmentation**: Maps fire boundaries.
3. **Forecasting**: Predicts future fire spread.

**Background Concepts**

**1. Satellite Imaging**

Satellite imaging refers to the process of capturing images of the Earth from satellites equipped with specialized sensors. These images provide valuable data for various applications, including environmental monitoring, disaster management, land use analysis, and defense.

**Types of Satellite Imaging Sensors**

**Optical Imaging Sensors**

Capture images in the visible, infrared, and ultraviolet spectrum.

Example: NASA’s Landsat, Sentinel-2 for Earth observation.

**Radar Imaging (Synthetic Aperture Radar - SAR)**

Uses microwave signals to penetrate clouds, making it useful for all-weather, day-and-night imaging.

Example: Sentinel-1, RADARSAT.

**Thermal Imaging:**

Detects heat radiation from objects and surfaces, commonly used for wildfire detection.

Example: MODIS (Moderate Resolution Imaging Spectroradiometer), VIIRS (Visible Infrared Imaging Radiometer Suite).

**2. Machine Learning (ML):**

Machine Learning is a way to teach computers to *learn from experience* without being explicitly programmed. Instead of giving strict rules, you show the computer many examples, and it figures out patterns on its own.

Example**:**  
Imagine teaching a child to recognize cats:

1. Show them 100 pictures of cats and say "This is a cat."
2. Show them 100 pictures of dogs and say "This is not a cat."
3. Eventually, the child learns to *generalize* and identify cats in new pictures.

ML works the same way, but with computers.

|  |  |  |
| --- | --- | --- |
| Type of ML | What It Does | What It Does |
| Supervised Learning | Learns from labeled examples (input + correct answer) | Spam detection (emails labeled "spam" or "not spam") |
| Unsupervised Learning | Finds patterns in unlabeled data | Grouping customers by shopping habits |
| Reinforcement Learning | Learns by trial-and-error with rewards | Training a robot to walk |

**3. Deep Learning**

A *subset of ML* that uses artificial "brains" called **neural networks** to solve complex tasks. These networks are inspired by the human brain’s structure.

**Neural Network Analogy:**  
Think of it like layers of chefs in a kitchen:

1. **First Layer:** Identifies basic ingredients (edges, colors in an image).
2. **Middle Layers:** Combines ingredients into dishes (shapes, textures).
3. **Final Layer:** Decides the meal (e.g., "fire" or "no fire").

**What is a Neural Network?**

A neural network is a computational model inspired by the human brain, designed to recognize patterns and learn from data. It consists of layers of interconnected nodes, or neurons, that process and transform input data to produce meaningful outputs. Neural networks are fundamental in machine learning (ML) and deep learning (DL) and are widely used in tasks such as image recognition, natural language processing (NLP), forecasting, and autonomous systems.

**Structure of a Neural Network**

A neural network typically consists of three main layers:

1. Input Layer:
   * This is the first layer that receives raw data (e.g., images, text, or numerical data).
   * Each neuron in this layer represents a feature from the input data.
2. Hidden Layers:
   * These layers perform the actual learning by applying transformations to the data.
   * Each neuron in a hidden layer processes input from previous layers and passes the result forward.
   * These layers use activation functions (e.g., ReLU, Sigmoid, Tanh) to introduce non-linearity and help the model learn complex patterns.
3. Output Layer:
   * This layer produces the final prediction or classification.
   * For example, in a binary classification task (fire vs. no fire), the output layer might have a Sigmoid activation function that outputs a probability between 0 and 1.
   * In a multi-class classification task, a Softmax activation function is often used to assign probabilities to multiple classes.

**How a Neural Network Learns**

Neural networks learn through an iterative process known as backpropagation, which consists of:

1. Forward Propagation:
   * Data moves through the network, layer by layer, and produces an output prediction.
2. Loss Calculation:
   * The difference between the predicted output and the actual value is measured using a loss function (e.g., Mean Squared Error for regression or Cross-Entropy Loss for classification).
3. Backward Propagation (Backprop):
   * The error is propagated backward through the network to adjust weights and biases using an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam.
   * The goal is to minimize the loss by updating weights iteratively.

**4. Classification**

Classification is a type of supervised learning in machine learning where an algorithm learns to categorize input data into predefined classes or labels. It is widely used in applications such as image recognition, spam detection, medical diagnosis, and wildfire detection.

How Classification Works

1. Training Phase:

The model is trained on labeled data, meaning each input sample is associated with a specific category.

Features from the input data are extracted and learned through an algorithm.

1. Prediction Phase:

When a new, unseen input is provided, the model assigns it to one of the learned categories.

It outputs a probability score or direct class label based on its confidence in the prediction.

1. Evaluation:

Performance is measured using metrics like accuracy, precision, recall, and F1-score to determine how well the model classifies data.

**Types of Classification**

1. Binary Classification – The model categorizes data into two classes (e.g., fire vs. no fire in wildfire detection).
2. Multi-Class Classification – The model assigns data to one of multiple classes (e.g., classifying different species of plants).
3. Multi-Label Classification – Each input can belong to multiple categories simultaneously (e.g., an image may contain both smoke and flames).

**5. Convolutional Neural Networks (CNNs)**

A **Convolutional Neural Network (CNN)** is a type of deep learning model designed to process grid-like data (e.g., images, videos). It mimics how humans process visual information by detecting patterns hierarchically:

1. **Simple patterns**: Edges, colors, textures.
2. **Complex patterns**: Shapes, objects (e.g., fire plumes).

**Why CNNs Were Used in This Wildfire Research?**

Your goal was to classify satellite images as **"fire"** or **"no fire"**. Hence, Classification, and in order to achieve this MobileNetV2 CNN architecture was used.

**MobileNetV2:**

MobileNetV2 is a light weight Convolutional Neural Network (CNN) designed for efficient deep learning on mobile and edge devices. It is an improvement over MobileNetV1, offering higher accuracy with lower computational cost. MobileNetV2 was introduced by Google in 2018 and is widely used for tasks like image classification, object detection, and segmentation while being optimized for real-time applications.

**5. Segmentation:**

Segmentation is a fundamental task in computer vision where an image is divided into different regions or objects based on pixel-level classification. It is commonly used in medical imaging, satellite imagery, self-driving cars, and wildfire detection.

**U-Net**

U-Net is a convolutional neural network (CNN) architecture designed for image segmentation tasks, particularly in medical and satellite imaging. It follows a U-shaped structure with a contracting path (encoder) that extracts features and a symmetric expanding path (decoder) that reconstructs spatial information.

**How U-Net Works**

U-Net has a U-shaped architecture, consisting of two main parts:

1. Contracting Path (Encoder) – A downsampling process where the image is passed through a series of convolutional layers and pooling layers, extracting important features while reducing spatial dimensions.
2. Expanding Path (Decoder) – An upsampling process that reconstructs the image by using transposed convolutions and feature maps from the encoder, allowing for precise localization of objects.

A key feature of U-Net is skip connections, which transfer detailed features from the encoder to the decoder, helping to retain spatial details lost during downsampling.

**Why U-Net is Used for Wildfire Segmentation**

* High Accuracy in Pixel-Level Classification – U-Net can accurately distinguish fire-affected areas from background regions.
* Handles Small and Large Fires – The architecture captures both fine details and larger fire regions.
* Works with Limited Data – U-Net performs well even with small datasets, making it useful for wildfire segmentation from satellite images.

**Time Series Forecasting**

Time series forecasting is a machine learning technique used to predict future values based on previously observed data over time. It is widely applied in fields such as finance, weather prediction, stock market analysis, and wildfire forecasting.

**How Time Series Forecasting Works**

1. Data Collection – Gather historical data with timestamps (e.g., daily wildfire occurrences, temperature trends).
2. Data Preprocessing – Handle missing values, normalize data, and create lag features.
3. Model Selection – Choose a forecasting model such as ARIMA, LSTM, or Prophet.
4. Training & Validation – Fit the model on past data and evaluate it using metrics like MAE (Mean Absolute Error) and RMSE (Root Mean Squared Error).
5. Prediction – Use the trained model to forecast future values.

In this project, **FBProphet** (a time series forecasting tool developed by Facebook) was used to predict wildfire occurrences based on satellite data. It is designed to handle time series data with strong seasonal patterns, missing values, and outliers, making it a robust choice for real-world forecasting applications.

**Source code and explanations**

**Classification:**

Classification Code Documentation

File: `Classification.ipynb`

Goal: Train a CNN to detect fires in satellite images.

```python

# Install required libraries

!pip install folium tensorflow rasterio

```

- Purpose: Installs essential tools:

- `folium`: Map visualization

- `tensorflow`: Deep learning framework

- `rasterio`: Satellite image processing

1. Imports and Setup

```python

import os

import shutil

from sklearn.model\_selection import train\_test\_split

import tensorflow as tf

from tensorflow.keras import layers, models, regularizers, applications

import numpy as np

from tensorflow.keras.preprocessing import image\_dataset\_from\_directory

from pathlib import Path

import matplotlib.pyplot as plt

from sklearn.metrics import f1\_score, classification\_report, confusion\_matrix

```

- Purpose: Load libraries for:

- File handling (`os`, `shutil`)

- Data splitting (`train\_test\_split`)

- Model building (`tensorflow.keras`)

- Visualization (`matplotlib`)

2. Data Preparation

```python

# Configuration

DATA\_DIR = "/content/drive/MyDrive/FIRMS\_Data"

SPLIT\_RATIO = (0.7, 0.2, 0.1) # Train, Val, Test

SEED = 42 # For reproducibility

# Create folders

for split in ['train', 'val', 'test']:

for cls in ['fire', 'no\_fire']:

os.makedirs(os.path.join(DATA\_DIR, split, cls), exist\_ok=True)

```

- Purpose:

- Define paths and split ratios (70% train, 20% val, 10% test).

- Create folders to organize images into `train`, `val`, and `test` sets.

3. Split Data

```python

# Split files into train/val/test

for class\_name in ['fire', 'no\_fire']:

src\_dir = os.path.join(DATA\_DIR, class\_name)

files = [f for f in os.listdir(src\_dir) if f.endswith(('.png', '.jpg', '.jpeg'))]

# Split 1: Train+Val vs Test

train\_val\_files, test\_files = train\_test\_split(files, test\_size=SPLIT\_RATIO[2], random\_state=SEED)

# Split 2: Train vs Val

train\_files, val\_files = train\_test\_split(train\_val\_files,

test\_size=SPLIT\_RATIO[1]/(SPLIT\_RATIO[0]+SPLIT\_RATIO[1]),

random\_state=SEED)

# Copy files to folders

for f in train\_files:

shutil.copy(os.path.join(src\_dir, f),

os.path.join(DATA\_DIR, 'train', class\_name, f))

# Repeat for val\_files and test\_files...

```

- Purpose:

- Organizes raw images into structured folders for training.

- Uses `train\_test\_split` to ensure balanced splits.

- `random\_state=42` ensures reproducibility (same splits every run).

4. Load Datasets

```python

# Configuration

IMG\_SIZE = (256, 256) # Resize images to 256x256 pixels

BATCH\_SIZE = 32 # Process 32 images at once

# Load datasets

train\_ds = image\_dataset\_from\_directory(

f'{DATA\_DIR}/train',

image\_size=IMG\_SIZE,

batch\_size=BATCH\_SIZE,

label\_mode='binary' # Binary classification (fire/no\_fire)

)

val\_ds = image\_dataset\_from\_directory(f'{DATA\_DIR}/val', ...)

test\_ds = image\_dataset\_from\_directory(f'{DATA\_DIR}/test', ...)

```

- Purpose:

- `image\_dataset\_from\_directory` auto-labels images based on folder names (`fire`/`no\_fire`).

- `label\_mode='binary'` sets up a yes/no classification task.

- `BATCH\_SIZE=32` balances memory usage and training speed.

5. Handle Class Imbalance

```python

# Calculate class weights

fire\_count = len(os.listdir(f'{DATA\_DIR}/train/fire'))

no\_fire\_count = len(os.listdir(f'{DATA\_DIR}/train/no\_fire'))

total = fire\_count + no\_fire\_count

class\_weight = {0: (1 / no\_fire\_count) \* (total / 2.0), # Weight for "no\_fire"

1: (1 / fire\_count) \* (total / 2.0)} # Weight for "fire"

```

- Purpose:

- Assigns higher weight to the minority class (`fire`) to prevent model bias.

- Formula: `weight = (1 / class\_count) \* (total\_samples / 2)`

6. Build the Model

```python

# Use MobileNetV2 (pretrained on ImageNet)

base\_model = applications.MobileNetV2(

input\_shape=(\*IMG\_SIZE, 3),

include\_top=False, # Remove the original classifier

weights='imagenet' # Use pretrained weights

)

base\_model.trainable = False # Freeze pretrained layers

# Custom model head

model = models.Sequential([

# Data augmentation

layers.RandomFlip('horizontal\_and\_vertical'),

layers.RandomRotation(0.2),

layers.RandomZoom(0.2),

layers.RandomContrast(0.2),

# Preprocessing

layers.Rescaling(1./127.5, offset=-1), # Normalize for MobileNet

# Base model

base\_model,

# Custom layers

layers.GlobalAveragePooling2D(), # Convert 2D features to 1D

layers.Dropout(0.5), # Prevent overfitting

layers.Dense(1, activation='sigmoid') # Final decision layer

])

```

- Key Components:

- MobileNetV2: Pretrained on 1M+ general images (faster training).

- Data Augmentation: Artificially expands dataset with flipped/rotated images.

- GlobalAveragePooling2D: Summarizes spatial features into a single vector.

- Dropout: Randomly ignores 50% of neurons to prevent overfitting.

- Sigmoid Activation: Outputs probability between 0 (no fire) and 1 (fire).

7. Compile the Model

```python

# Focal loss (handles class imbalance)

def focal\_loss(gamma=2.0, alpha=0.25):

def loss(y\_true, y\_pred):

y\_true = tf.cast(y\_true, tf.float32)

ce = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

pt = tf.exp(-ce)

return tf.reduce\_mean(alpha \* (1 - pt)\*\*gamma \* ce)

return loss

# Custom learning rate schedule

lr\_schedule = tf.keras.optimizers.schedules.ExponentialDecay(

initial\_learning\_rate=1e-3,

decay\_steps=1000,

decay\_rate=0.9

)

model.compile(

optimizer=tf.keras.optimizers.Adam(lr\_schedule),

loss=focal\_loss(), # Use focal loss instead of standard binary\_crossentropy

metrics=['accuracy',

tf.keras.metrics.Precision(name='precision'),

tf.keras.metrics.Recall(name='recall'),

tf.keras.metrics.AUC(name='auc')]

)

```

- Key Choices:

- Focal Loss: Focuses on hard-to-classify fires (reduces false negatives).

- ExponentialDecay: Gradually reduces learning rate for stable training.

- Metrics: Track precision (avoid false alarms), recall (catch all fires), and AUC (overall performance).

8. Train the Model

```python

# Callbacks

callbacks = [

tf.keras.callbacks.EarlyStopping(

patience=5, # Stop if no improvement for 5 epochs

monitor='val\_auc', # Track validation AUC

mode='max', # Maximize AUC

restore\_best\_weights=True # Keep the best model

),

tf.keras.callbacks.ModelCheckpoint(

'best\_model.keras', # Save the best model

save\_best\_only=True,

monitor='val\_auc'

)

]

# Train

history = model.fit(

train\_ds,

validation\_data=val\_ds,

epochs=50,

class\_weight=class\_weight, # Apply class balancing

callbacks=callbacks

)

```

- Purpose:

- EarlyStopping: Prevents overfitting by halting training if no improvement.

- ModelCheckpoint: Saves the best model based on validation AUC.

- `class\_weight`: Prioritizes fire examples during training.

9. Evaluate the Model

```python

# Test set evaluation

test\_results = model.evaluate(test\_ds)

print(f"Test Loss: {test\_results[0]:.4f}")

print(f"Accuracy: {test\_results[1]:.4f}")

print(f"Precision: {test\_results[2]:.4f}")

print(f"Recall: {test\_results[3]:.4f}")

print(f"AUC: {test\_results[4]:.4f}")

# Generate predictions

test\_preds = model.predict(test\_ds)

test\_preds = (test\_preds > 0.5).astype(int) # Threshold at 0.5

# Classification report

print(classification\_report(test\_labels, test\_preds))

# Confusion matrix

print(confusion\_matrix(test\_labels, test\_preds))

```

- Metrics Explained:

- Accuracy: % of correct predictions.

- Precision: % of fire alerts that are real fires (avoid false alarms).

- Recall: % of actual fires detected.

- AUC: Overall performance (0.5 = random, 1.0 = perfect).

**Segmentation**

File: `Segmentation.ipynb`

Goal: Train a U-Net model to create pixel-level fire masks from satellite images.

1. Imports and Configuration

```python

import os

import cv2

import numpy as np

import shutil

import matplotlib.pyplot as plt

from tqdm import tqdm

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv2D, MaxPool2D, Dropout, concatenate, Conv2DTranspose

from tensorflow.keras.metrics import MeanIoU

```

- Purpose:

- `cv2` (OpenCV): Image processing (resizing, color conversion).

- `tensorflow`: Build and train the U-Net model.

- `MeanIoU`: Evaluates segmentation accuracy.

```python

# Configuration

DATASET\_ROOT = "/content/drive/MyDrive/FIRMS\_Data"

NEW\_SIZE = (512, 512) # Standard processing size

DEBUG\_MODE = True # Set to False for full dataset

# Fire color thresholds (HSV/LAB ranges)

HSV\_LOWER = np.array([0, 50, 50]) # Red/orange hues

HSV\_UPPER = np.array([20, 255, 255])

LAB\_LOWER = np.array([0, 130, 70]) # Brightness/chroma

LAB\_UPPER = np.array([255, 145, 90])

```

- Why:

- `NEW\_SIZE`: Balances detail and computation.

- `HSV/LAB`: Fire pixels fall in these color ranges.

2. Fire Mask Generation

```python

def enhanced\_fire\_mask(img\_path):

img = cv2.imread(img\_path)

img = cv2.resize(img, NEW\_SIZE)

# Convert to HSV & LAB color spaces

hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

lab = cv2.cvtColor(img, cv2.COLOR\_BGR2LAB)

# Thresholding

mask\_hsv = cv2.inRange(hsv, HSV\_LOWER, HSV\_UPPER)

mask\_lab = cv2.inRange(lab, LAB\_LOWER, LAB\_UPPER)

combined = cv2.bitwise\_or(mask\_hsv, mask\_lab)

# Cleanup small noise

kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (7,7))

refined = cv2.morphologyEx(combined, cv2.MORPH\_CLOSE, kernel)

refined = cv2.morphologyEx(refined, cv2.MORPH\_OPEN, np.ones((3,3)))

# Remove tiny detections

contours, \_ = cv2.findContours(refined, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

for cnt in contours:

if cv2.contourArea(cnt) < 50:

cv2.drawContours(refined, [cnt], -1, 0, -1)

return refined

```

- Step-by-Step:

1. Resize Image: Standardize to 512x512 pixels.

2. Color Thresholding: Isolate fire pixels using HSV (hue) and LAB (brightness).

3. Morphological Operations:

- `MORPH\_CLOSE`: Fills small holes in fire regions.

- `MORPH\_OPEN`: Removes thin noise.

4. Contour Filtering: Deletes tiny detections (<50 pixels).

3. Dataset Preparation

```python

def restructure\_dataset():

for split in ['train', 'val', 'test']:

# Create folders

img\_dir = os.path.join(DATASET\_ROOT, split, 'images')

mask\_dir = os.path.join(DATASET\_ROOT, split, 'masks')

os.makedirs(img\_dir, exist\_ok=True)

os.makedirs(mask\_dir, exist\_ok=True)

# Process fire images

fire\_src = os.path.join(DATASET\_ROOT, split, 'fire')

for fname in tqdm(os.listdir(fire\_src)):

src\_path = os.path.join(fire\_src, fname)

dest\_img = os.path.join(img\_dir, fname)

dest\_mask = os.path.join(mask\_dir, f"mask\_{fname}")

# Generate and save mask

mask = enhanced\_fire\_mask(src\_path)

cv2.imwrite(dest\_mask, mask)

# Save resized image

img = cv2.imread(src\_path)

img = cv2.resize(img, NEW\_SIZE)

cv2.imwrite(dest\_img, img)

# Process no-fire images (empty masks)

nofire\_src = os.path.join(DATASET\_ROOT, split, 'no\_fire')

for fname in tqdm(os.listdir(nofire\_src)):

cv2.imwrite(dest\_mask, np.zeros(NEW\_SIZE[::-1], dtype=np.uint8))

```

- Purpose:

- Organizes data into `images` and `masks` folders.

- Generates fire masks for fire images; empty masks for no-fire.

- Uses `tqdm` for progress bars.

4. Dataset Loading

```python

def parse\_image\_mask\_pair(image\_path, mask\_path):

# Load and normalize image

image = tf.io.read\_file(image\_path)

image = tf.image.decode\_png(image, channels=3)

image = tf.image.convert\_image\_dtype(image, tf.float32) # [0,1] range

# Load and binarize mask

mask = tf.io.read\_file(mask\_path)

mask = tf.image.decode\_png(mask, channels=1)

mask = tf.image.convert\_image\_dtype(mask, tf.float32)

mask = tf.where(mask > 0.5, 1.0, 0.0) # Binary mask (0 or 1)

return image, mask

def create\_dataset(split):

image\_dir = f"{DATASET\_ROOT}/{split}/images"

mask\_dir = f"{DATASET\_ROOT}/{split}/masks"

# List and pair image/mask paths

image\_paths = tf.data.Dataset.list\_files(f"{image\_dir}/\*", shuffle=False)

mask\_paths = tf.data.Dataset.list\_files(f"{mask\_dir}/\*", shuffle=False)

dataset = tf.data.Dataset.zip((image\_paths, mask\_paths))

# Parse pairs and augment training data

dataset = dataset.map(parse\_image\_mask\_pair, num\_parallel\_calls=tf.data.AUTOTUNE)

if split == 'train':

dataset = dataset.map(lambda x, y: (tf.image.random\_flip\_left\_right(x),

tf.image.random\_flip\_left\_right(y)))

dataset = dataset.map(lambda x, y: (tf.image.random\_brightness(x, 0.1), y))

dataset = dataset.shuffle(100)

return dataset.batch(8).prefetch(tf.data.AUTOTUNE)

```

- Key Points:

- `parse\_image\_mask\_pair`: Normalizes images and masks to [0,1].

- `create\_dataset`:

- Applies flips and brightness augmentation to training data.

- `prefetch`: Speeds up training by loading next batch in background.

5. U-Net Model

```python

def unet\_model(input\_size=(512, 512, 3)):

inputs = Input(input\_size)

# Encoder (Downsample)

c1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)

c1 = Conv2D(64, 3, activation='relu', padding='same')(c1)

p1 = MaxPool2D(2)(c1)

c2 = Conv2D(128, 3, activation='relu', padding='same')(p1)

c2 = Conv2D(128, 3, activation='relu', padding='same')(c2)

p2 = MaxPool2D(2)(c2)

# Bottleneck

c3 = Conv2D(256, 3, activation='relu', padding='same')(p2)

c3 = Conv2D(256, 3, activation='relu', padding='same')(c3)

# Decoder (Upsample)

u4 = Conv2DTranspose(128, 2, strides=2, padding='same')(c3)

u4 = concatenate([u4, c2])

c4 = Conv2D(128, 3, activation='relu', padding='same')(u4)

c4 = Conv2D(128, 3, activation='relu', padding='same')(c4)

u5 = Conv2DTranspose(64, 2, strides=2, padding='same')(c4)

u5 = concatenate([u5, c1])

c5 = Conv2D(64, 3, activation='relu', padding='same')(u5)

c5 = Conv2D(64, 3, activation='relu', padding='same')(c5)

outputs = Conv2D(1, 1, activation='sigmoid')(c5)

return tf.keras.Model(inputs, outputs)

```

- U-Net Architecture:

- Encoder: Captures context (what is fire?) via downsampling.

- Decoder: Restores spatial details (where is fire?) via upsampling.

- Skip Connections: Combine encoder/decoder features for precise boundaries.

6. Training

```python

model = unet\_model()

model.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy', MeanIoU(num\_classes=2)])

callbacks = [

tf.keras.callbacks.ModelCheckpoint('best\_model.h5', save\_best\_only=True),

tf.keras.callbacks.EarlyStopping(patience=10)

]

history = model.fit(train\_ds, epochs=50, validation\_data=val\_ds, callbacks=callbacks)

```

- Loss: `binary\_crossentropy` – standard for binary segmentation.

- Metrics:

- `accuracy`: % of correct pixel labels.

- `MeanIoU`: Intersection-over-Union (overlap between predictions and truth).

7. Evaluation

```python

test\_results = model.evaluate(test\_ds)

print(f"Test Loss: {test\_results[0]:.4f}")

print(f"Test Accuracy: {test\_results[1]:.4f}")

print(f"Test IoU: {test\_results[2]:.4f}")

# Visual predictions

def show\_predictions(dataset, num=3):

for images, masks in dataset.take(1):

preds = model.predict(images)

plt.figure(figsize=(15, num\*5))

for i in range(num):

plt.subplot(num, 3, i\*3+1)

plt.imshow(images[i])

plt.subplot(num, 3, i\*3+2)

plt.imshow(masks[i], cmap='gray')

plt.subplot(num, 3, i\*3+3)

plt.imshow(preds[i] > 0.5, cmap='gray')

plt.show()

```

- IoU Interpretation:

- 0.0 = No overlap, 1.0 = Perfect match.

- Your result: 0.7121 – good but can improve.

**Time-Series**

Time Series Forecasting Code Documentation

File: `Time\_series.ipynb`

Goal: Predict daily wildfire counts and acres burned using historical data.

1. Imports and Setup

```python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from prophet import Prophet

from prophet.diagnostics import cross\_validation, performance\_metrics

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error

from math import sqrt

import seaborn as sns

```

- Purpose:

- `pandas`: Data manipulation.

- `prophet`: Facebook’s time-series forecasting tool.

- `sklearn.metrics`: Evaluate prediction errors (MAE, RMSE).

2. Data Preparation

```python

def prepare\_fire\_data(file\_path):

# Load and parse dates

df = pd.read\_csv(file\_path)

df['ALARM\_DATE'] = pd.to\_datetime(df['ALARM\_DATE']).dt.tz\_localize(None)

df['CONT\_DATE'] = pd.to\_datetime(df['CONT\_DATE']).dt.tz\_localize(None)

# Create daily time series

daily\_fires = df.groupby(pd.Grouper(key='ALARM\_DATE', freq='D')).agg(

fire\_count=('INC\_NUM', 'count'),

total\_acres=('GIS\_ACRES', 'sum')

).reset\_index()

# Fill missing dates with zeros

date\_range = pd.date\_range(start=daily\_fires['ALARM\_DATE'].min(),

end=daily\_fires['ALARM\_DATE'].max())

daily\_fires = daily\_fires.set\_index('ALARM\_DATE').reindex(date\_range).fillna(0).reset\_index()

daily\_fires.columns = ['ds', 'y\_fire\_count', 'y\_acres\_burned']

return daily\_fires

```

- Steps:

1. Load Data: Read CAL FIRE’s historical wildfire records.

2. Aggregate Daily: Count fires and sum acres burned per day.

3. Handle Gaps: Fill missing dates (e.g., days with no fires) with zeros.

- Why?

- Time-series models require continuous daily data.

- Zeros represent "no fire" days.

3. Train-Test Split

```python

# Split into 80% train, 20% test

split\_date = fire\_data['ds'].quantile(0.8)

train = fire\_data[fire\_data.ds <= split\_date]

test = fire\_data[fire\_data.ds > split\_date]

```

- Rationale:

- Test on the most recent 20% of data to simulate real-world forecasting.

- Preserves temporal order (no future data leaks into training).

4. Prophet Forecasting Model

```python

def run\_prophet\_forecast(train\_data, test\_data, target\_col='y\_fire\_count'):

model = Prophet(

yearly\_seasonality=True, # Capture annual patterns (e.g., summer fires)

weekly\_seasonality=False, # Ignore weekly cycles (irrelevant for fires)

daily\_seasonality=False,

changepoint\_prior\_scale=0.05 # Control trend flexibility

)

# Format data for Prophet

prophet\_train = train\_data[['ds', target\_col]].rename(columns={target\_col: 'y'})

model.fit(prophet\_train)

# Predict on test dates

future = test\_data[['ds']].copy()

forecast = model.predict(future)

# Merge predictions with actual values

results = pd.merge(forecast[['ds', 'yhat']], test\_data[['ds', target\_col]], on='ds')

results.columns = ['date', 'predicted', 'actual']

return model, results

```

- Key Parameters:

- `yearly\_seasonality`: Fires peak annually (e.g., dry seasons).

- `changepoint\_prior\_scale`: Allows the trend to change direction (e.g., policy changes).

- Output:

- `yhat`: Predicted values.

- `actual`: Ground truth from test data.

5. Model Evaluation

```python

def evaluate\_model(results, target\_name):

print(f'\nEvaluation for {target\_name}:')

mae = mean\_absolute\_error(results.actual, results.predicted)

rmse = sqrt(mean\_squared\_error(results.actual, results.predicted))

print(f'MAE: {mae:.2f}') # Average error magnitude

print(f'RMSE: {rmse:.2f}') # Penalizes large errors

# Calculate MAPE only for days with fires

non\_zero = results[results.actual > 0]

if len(non\_zero) > 0:

mape = np.mean(np.abs((non\_zero.actual - non\_zero.predicted) / non\_zero.actual)) \* 100

print(f'MAPE (non-zero days): {mape:.2f}%') # % error

```

- Metrics:

- MAE: Average absolute error (e.g., ±3 fires/day).

- RMSE: Highlights large errors (e.g., missing a mega-fire).

- MAPE: % error on days with fires (ignores zeros).

6. Cross-Validation

```python

def perform\_cv(model, train\_data, horizon=365):

df\_cv = cross\_validation(

model,

initial='3650 days', # 10-year training window

period='180 days', # Retrain every 6 months

horizon=f'{horizon} days', # Predict 1 year ahead

parallel="processes"

)

df\_p = performance\_metrics(df\_cv)

return df\_cv, df\_p

```

- Purpose:

- Tests model reliability over multiple time periods.

- Initial: 10 years of training data.

- Horizon: Predict 1 year into the future.

7. Results and Visualization

Fire Count Forecast:

- MAE: 0.72 (predicts within ±1 fire/day).

- RMSE: 1.91 (rare large errors).

Acres Burned Forecast:

- MAE: 3,077 acres (underestimates major fires).

- MAPE: 85,900% (poor performance on non-zero days).

```python

def plot\_results(results, title):

plt.figure(figsize=(12, 6))

plt.plot(results.date, results.actual, label='Actual', alpha=0.7)

plt.plot(results.date, results.predicted, label='Predicted', linestyle='--')

plt.title(title)

plt.legend()

plt.show()

```

![Forecast Plot](https://i.imgur.com/xyZ1qD2.png)

Model captures seasonal trends but misses extreme events.

8. Why Prophet?

|  |  |
| --- | --- |
| Feature | Benefit |
| Automatic Seasonality | Detects yearly fire patterns without manual tuning. |
| Missing Data Handling | Works with gaps/zeros in historical data |
| Interpretability | Provides trend/seasonality breakdowns |

Alternatives:

- ARIMA: Requires manual parameter tuning.

- LSTM: Needs more data and computational power.

9. Improvement Strategies

1. Add Features:

```python

def add\_weather\_data(df):

df = df.merge(weather\_api\_data, on='ds')

df['drought\_index'] = calculate\_spi(df['precipitation'])

return df

```

2. Hybrid Model: Combine Prophet with anomaly detection for extreme fires.

3. Threshold Adjustment\*\*: Focus on predicting fires >100 acres.