AI SpillGuard: Oil Spill Detection Using Deep Learning

Author: Aditya Kumar Upadhayay

Affiliation: Infosys Springboard

Course/Project: Infosys AI Intern Project

Date: October 31 , 2025

Abstract

Oil spills pose a severe and immediate threat to marine ecosystems, coastal economies, and environmental health. Traditional methods for monitoring spills, which often rely on manual inspection of satellite or aerial imagery, are time-consuming, labor-intensive, and prone to human error. This project, **AI SpillGuard**, presents an automated, end-to-end solution for the rapid and accurate detection and segmentation of oil spills from satellite images.

We employ a **U-Net–based deep learning architecture**, a convolutional neural network (CNN) specifically designed for biomedical image segmentation, which we have adapted for this geospatial task. The model is trained on a public dataset of satellite images and corresponding binary masks, learning to differentiate complex oil-spill features from the surrounding ocean environment.

The system segments affected areas with high fidelity, achieving a **Dice Coefficient of 0.85** and an **Intersection over Union (IoU) of 0.81** on the validation set. This high accuracy supports near real-time environmental monitoring and emergency response.

To make this technology accessible, the project integrates model training, evaluation, and deployment into a single workflow. The final, trained model is deployed via a **Streamlit-based web interface**, allowing non-technical users to upload satellite imagery and receive an instantaneous, visualized segmentation mask of the predicted spill. This work provides a scalable framework for automated environmental monitoring.

Table of Contents

1. Abstract 2
2. Introduction 4
    1. Background and Context
    2. The Need for Automation
    3. Project Objectives
    4. Scope and Contribution
3. Problem Statement 5
4. Literature Review 6
    1. Traditional Image Processing Methods
    2. Classical Machine Learning Approaches
    3. Deep Learning for Semantic Segmentation
    4. U-Net in Geospatial Analysis
5. System Requirements 7
    1. Hardware Specifications
    2. Software and Libraries
6. Dataset Description 8
    1. Data Source and Structure
    2. Data Characteristics and Annotation
    3. Sample Images
7. Methodology 9
    1. System Architecture Overview
    2. Step 1: Data Preprocessing
    3. Step 2: Model Architecture (U-Net)

8. Model Architecture (Detailed) 10
    1. The Encoder (Contracting Path)
    2. The Bottleneck
    3. The Decoder (Expanding Path)
    4. Skip Connections
9. Training and Evaluation 11
    1. Step 3: Loss Function and Metrics
    2. Step 4: Training Configuration
    3. Step 5: Evaluation Protocol
10. Implementation Details 12
    1. Development Environment
    2. Key Code Modules and Libraries
11. Results and Analysis 13
    1. Quantitative Results
    2. Training Performance
    3. Qualitative (Visual) Results
12. Deployment via Streamlit 14
    1. Application Interface
    2. User Workflow
13. Conclusion 15
14. Future Enhancements 15
15. References 16

2. Introduction

2.1 Background and Context

Environmental disasters such as the Deepwater Horizon or Exxon Valdez incidents have demonstrated the catastrophic and long-lasting damage oil spills can inflict. Spills contaminate coastlines, destroy marine habitats, and devastate local economies reliant on fishing and tourism. The **speed of detection** is the single most critical factor in mitigating this damage. An early and accurate assessment of a spill's location, size, and drift direction allows containment and cleanup operations to be mobilized effectively.

2.2 The Need for Automation

Currently, many agencies rely on traditional detection methods, including manual inspection of satellite images (both optical and Synthetic Aperture Radar - SAR), aerial surveillance, and reports from ships. These methods are:

- **Labor-Intensive:** Requiring trained analysts to manually scan vast swaths of ocean imagery.
- **Slow:** The delay between image acquisition and manual analysis can be many hours, allowing the spill to spread.
- **Error-Prone:** Factors like sun glint, cloud cover, and "look-alike" biogenic slicks can easily confuse human interpreters, leading to false positives or missed detections.

Recent advances in computer vision, powered by deep learning, offer a transformative alternative. These models can analyze massive volumes of image data in seconds, identifying complex patterns that are often subtle to the human eye.

2.3 Project Objectives

The primary goal of this project is to design, build, and deploy an end-to-end system, AI SpillGuard, for the automated segmentation of oil spills.

The specific objectives are:

1. **Build:** To implement a robust deep learning model, specifically a U-Net architecture, for pixel-level segmentation of oil spills.
2. **Train:** To train and validate the model on a public dataset of satellite imagery to achieve high accuracy.
3. **Evaluate:** To rigorously assess the model's performance using standard segmentation metrics, including the Dice Coefficient and IoU.
4. **Deploy:** To create a user-friendly, interactive web application using Streamlit that allows for real-time inference on new images.
5. **Analyze:** To provide visualization tools for interpreting model predictions and performance analytics.

2.4 Scope and Contribution

This project's scope covers the complete machine learning lifecycle: data collection and preprocessing, model design and training, and final deployment. The primary contribution is not just the trained model itself, but the **integration of this model into a functional, accessible tool** that can serve as a prototype for operational environmental monitoring systems.

3. Problem Statement

The core technical challenge is to perform **semantic segmentation** on satellite images to isolate oil spills. This task is significantly more complex than simple image classification. The model must assign a class (oil spill or background) to **every pixel** in the image.

This presents several difficulties:

- **Visual Complexity:** Oil spills lack a fixed shape or size. Their appearance varies greatly based on thickness, weathering, and sea state.
- **Challenging Backgrounds:** The model must distinguish spills from a complex and dynamic ocean background, which includes waves, sun glint, cloud shadows, ship wakes, and natural biogenic slicks (e.g., algae blooms) that can mimic the appearance of oil.
- **Data Scarcity:** While satellite images are abundant, high-quality, pixel-perfect annotated *masks* for training are rare and expensive to create.
- **Variable Conditions:** The model must be robust to variable lighting, atmospheric noise, and different image resolutions from various satellite sensors.

The system must, therefore, be able_ to learn the unique textural and spatial features of oil spills and segment them with **high precision** (minimizing false positives) and **high recall** (minimizing missed spills).

4. Literature Review

4.1 Traditional Image Processing Methods

Early attempts at automated oil spill detection relied on traditional computer vision techniques. These include:

- **Thresholding:** Simple methods that classify pixels based on their intensity values. This fails because oil and water do not have a consistent, predictable intensity difference, especially with noise like sun glint.
- **Edge Detection:** Algorithms like Canny or Sobel were used to find the boundaries of spills. These are highly susceptible to noise and often result in fragmented, incomplete detections.
- **Feature-Based Methods:** These involved handcrafting features (e.g., texture, wavelet transforms, statistical properties) to describe image patches, which were then fed into classifiers.

These methods are fundamentally brittle, as they rely on predefined rules and features that cannot adapt to the high variability of real-world ocean conditions.

4.2 Classical Machine Learning Approaches

A step above traditional methods, classical ML models like **Support Vector Machines (SVMs)** and **Random Forests** were applied. Analysts would extract a large set of handcrafted features from image patches, and the ML model would learn a decision boundary. While more robust than simple thresholding, their performance is entirely dependent on the quality of the "handcrafted features." This feature engineering process is time-consuming and may miss crucial abstract features.

4.3 Deep Learning for Semantic Segmentation

The advent of Deep Learning, specifically **Convolutional Neural Networks (CNNs)**, revolutionized this field. CNNs automatically learn hierarchical spatial features directly from raw pixel data, eliminating the need for manual feature engineering. Architectures like **Fully Convolutional Networks (FCN)** demonstrated the ability to produce dense, pixel-level predictions for segmentation.

4.4 U-Net in Geospatial Analysis

The **U-Net** architecture (Ronneberger et al., 2015) was a pivotal development. Originally designed for biomedical image segmentation, its architecture is uniquely suited for tasks like ours:

- **Encoder-Decoder Structure:** It first downsamples the image to capture context (the "what") and then upsamples to reconstruct a high-resolution mask, localizing the features (the "where").
- **Skip Connections:** The key innovation of U-Net is the "skip connection," which concatenates feature maps from the encoder path directly to the decoder path. This allows the decoder to recover fine-grained spatial information (like precise spill boundaries) that is lost during the downsampling process.

Multiple studies have successfully applied U-Net and its variants to geospatial and remote sensing tasks, including oil spill detection from both SAR and optical imagery, consistently demonstrating state-of-the-art performance, especially when annotated data is limited.

5. System Requirements

This section details the hardware and software specifications required for both training and deploying the AI SpillGuard system.

5.1 Hardware Specifications

- **Processor:**
    - **Training:** Intel i7/AMD Ryzen 7 or above.
    - **Deployment:** Intel i5 or equivalent.
- **GPU (Recommended for Training):**
    - NVIDIA GeForce RTX 2060 (6GB VRAM) or higher.
    - Alternatively, cloud-based GPUs like Google Colab (NVIDIA T4 or P100) or AWS/GCP instances. Training is computationally intensive and extremely slow on a CPU.
- **RAM:**
    - **Training:** 16 GB or higher.
    - **Deployment:** 8 GB.
- **Storage:** 10 GB minimum free space for the dataset, virtual environment, and saved model files.

5.2 Software and Libraries

- **Programming Language:** Python 3.10+
- **Deep Learning Framework:** TensorFlow 2.x / Keras
- **Web Application Framework:** Streamlit
- **Image Processing:** OpenCV-Python (cv2)
- **Data Visualization:** Matplotlib, Seaborn

- **Numerical Computation:** NumPy
- **Data Handling:** Pandas
- **Development Environment:** Google Colab (for training), VS Code or Jupyter Notebook (for deployment development)

6. Dataset Description

6.1 Data Source and Structure

The dataset used for this project was obtained from the **Kaggle Oil Spill Detection Dataset**. It provides a clean, pre-structured collection of satellite images and their corresponding segmentation masks, which is ideal for this proof-of-concept project.

The data is organized into three distinct directories:

```
dataset/
 ├── train/
|    ├── images/ (e.g., image_001.png)
|   └── masks/  (e.g., image_001.png)
 ├── val/
|    ├── images/
|   └── masks/
└── test/
    ├── images/
   └── masks/
```

6.2 Data Characteristics and Annotation

- **Images:** The images are RGB satellite photographs of ocean regions. They contain a mix of clear water, oil-affected areas, and visual noise (e.g., small clouds, sun glint).
- **Masks:** The masks are 8-bit grayscale images that serve as the "ground truth." They are binary, where pixels with a value of **255 (white)** represent an oil spill, and pixels with a value of **0 (black)** represent the background (water, clouds, etc.).
- **Data Splits:**
  - **Training Set:** 200 images and masks
  - **Validation Set:** 50 images and masks
  - **Test Set:** 100 images and masks

The validation set is used during training to monitor performance on unseen data and prevent overfitting. The test set is held out until the very end to provide a final, unbiased evaluation of the model's generalization.

7. Methodology

7.1 System Architecture Overview

The methodology follows an end-to-end machine learning pipeline, as illustrated in the system architecture diagram below.

7.2 Step 1 – Data Preprocessing

Raw image data is not suitable for direct input into a neural network. A preprocessing pipeline was established to standardize the data:

1. **Loading Paths:** First, the file paths for all images and their corresponding masks were loaded from the directories and aligned.
2. **Resizing:** All images and masks were resized to a uniform dimension of **256×256 pixels**. This is essential as a CNN requires a fixed input size. This size provides a good balance between spatial detail and computational load.
3. **Normalization:** The pixel values of the RGB images (which range from 0–255) were normalized to a floating-point range of **[0, 1]** by dividing by 255.0. This stabilizes the training process and helps the model converge faster.
4. **Mask Conversion:** The ground-truth masks were converted to binary format (values of 0 or 1) and ensured to have a single channel, as the model will output a single-channel prediction.

7.3 Step 2 – Model Architecture (U-Net)

We selected the **U-Net** architecture due to its proven success in segmentation tasks with limited data. The U-Net consists of three main parts:

1. **Encoder (Contracting Path):** This path acts as a feature extractor. It uses stacked convolution and max pooling layers to capture the *context* of the image. As the image is downsampled, the spatial

dimensions decrease, but the number of feature channels (depth) increases, allowing the model to learn complex patterns.

2. **Bottleneck:** This is the bridge at the bottom of the "U," connecting the encoder and decoder. It learns the deepest spatial representations of the features.

3. **Decoder (Expanding Path):** This path's goal is to reconstruct the segmentation mask. It uses "up-sampling" (specifically, Transposed Convolutions) to progressively increase the spatial dimensions back to the original 256x256 size.

4. **Output Layer:** A final 1x1 convolution with a **sigmoid** activation function is used to produce the final output. The sigmoid function squashes the output of each pixel to a value between 0 and 1, which can be interpreted as the probability that the pixel belongs to the "oil spill" class.

8. Model Architecture (Detailed)

Figure 8.1: The U-Net Architecture, highlighting skip connections.

8.1 The Encoder (Contracting Path)

The encoder consists of multiple blocks. In our implementation, each block contains:

1. Two consecutive **3×3 Convolution** layers with ReLU activation.
2. One **2×2 Max Pooling** layer with a stride of 2 for downsampling.

This process is repeated, doubling the number of feature channels at each step (e.g., 16 → 32 → 64).

8.2 The Bottleneck

This layer sits at the lowest point of the U. It consists of two 3x3 convolution layers (with 64 filters in this summary) and serves as the point where the model transitions from downsampling to upsampling.

8.3 The Decoder (Expanding Path)

The decoder's job is to semantically project the learned features back onto the pixel space. Each block in the decoder path consists of:

1. A **2×2 Convolution Transpose** layer, which upsamples the feature map (halving the channels).
2. A **Concatenation** layer, which merges the upsampled feature map with the corresponding feature map from the encoder path via a skip connection.
3. Two consecutive **3×3 Convolution** layers with ReLU activation.

8.4 Skip Connections

The key feature of U-Net. By concatenating feature maps from the encoder to the decoder, the model directly passes **high-resolution spatial information** that was captured early in the network. This allows the decoder to combine deep, abstract contextual information (from the bottleneck) with shallow, fine-grained detail (from the encoder), resulting in highly precise segmentation boundaries.

Model Summary Table

(This is a simplified summary; a full U-Net has many more layers in its blocks)

Layer Type	**Filters**	**Kernel Size**	**Activation**	**Output Shape**
Encoder Block 1				
Conv2D	16	3×3	ReLU	256×256×16
MaxPooling2D	–	2×2	–	128×128×16
Encoder Block 2				
Conv2D	32	3×3	ReLU	128×128×32
MaxPooling2D	–	2×2	–	64×64×32
Bottleneck				
Conv2D	64	3×3	ReLU	64×64×64
Decoder Block 1				
Conv2DTranspose	32	2×2	–	128×128×32
Concat (from Enc 2)	–	–	–	128×128×64
Conv2D	32	3×3	ReLU	128×128×32
Decoder Block 2				
Conv2DTranspose	16	2×2	–	256×256×16
Concat (from Enc 1)	–	–	–	256×256×32
Conv2D	16	3×3	ReLU	256×256×16
Output Layer				
Output (Conv2D)	1	1×1	**Sigmoid**	256×256×1

9. Training and Evaluation

9.1 Step 3 – Loss Function and Metrics

- Loss Function: Binary Cross-Entropy (BCE)

We used BCE as our loss function. This function is ideal for binary (0/1) classification problems. It treats each pixel in the 256x256 mask as an independent classification task, comparing the model's predicted probability (0 to 1) with the actual ground truth (0 or 1).

- Evaluation Metrics:

Accuracy alone is a poor metric for segmentation, especially with imbalanced data (e.g., if only 5% of the image is a spill, a model that predicts "no spill" everywhere is 95% accurate). We, therefore, rely on metrics that measure overlap.

1. Dice Coefficient: A common metric for segmentation, it measures the overlap between the predicted and true masks. It is calculated as:

A score of 1.0 means a perfect overlap, while 0 means no overlap.

2. Intersection over Union (IoU / Jaccard Index): Similar to Dice, it measures the ratio of the overlap to the total combined area.

IoU is a stricter metric than Dice and is widely used in segmentation challenges.

9.2 Step 4 – Training Configuration

- **Optimizer: Adam** (Adaptive Moment Estimation). This is a robust, widely-used optimizer that combines the advantages of other optimizers (like RMSProp and AdaGrad) and requires minimal tuning.

- **Epochs: 15**. An epoch is one complete pass through the entire training dataset. We found that after 15 epochs, the model's performance on the validation set began to plateau.
- **Batch Size: 8**. The model was trained on batches of 8 images at a time. This size was chosen to fit within the VRAM of the Google Colab GPU.

9.3 Step 5 – Evaluation

After training, the saved model (with the best validation Dice score) was loaded and evaluated against the **Test Set**—the 100 images the model had never seen before. This provides a final, unbiased measure of the model's ability to generalize to new, real-world data.

10. Implementation Details

10.1 Development Environment

The project was developed in two stages:

1. **Training: Google Colab** was used for all model training and experimentation. Its free access to NVIDIA T4 and P100 GPUs was essential for training the deep U-Net model in a reasonable amount of time.
2. **Deployment:** The Streamlit application was developed locally using **Visual Studio Code** and a Python virtual environment to manage dependencies.

10.2 Key Code Modules and Libraries

- **Data Loading Module (data_loader.py):**
    - Used OpenCV (cv2) to read images (cv2.imread) and resize them (cv2.resize).
    - Used NumPy to create and manipulate the image arrays.
    - Used glob to find and list all image/mask file paths.
- **Model Module (model.py):**
    - Used tensorflow.keras.layers (e.g., Conv2D, MaxPooling2D, Conv2DTranspose, Concatenate, Input) to define the U-Net architecture.
    - Used tensorflow.keras.models (Model) to compile the final model.
- **Training Module (train.py):**
    - This script integrated the data loader and model modules.
    - It defined the Adam optimizer, loss function, and metrics.

    - It configured the ModelCheckpoint and ReduceLROnPlateau callbacks.
    - It called model.fit() to start the training process and model.save("unet_oilspill_final.h5") to save the final artifact.
- **Visualization Module (evaluate.py):**
    - Used Matplotlib to plot the training history (loss and accuracy curves) and to display side-by-side image comparisons.
- **Deployment Module (app.py):**
    - Used Streamlit for the entire web framework (st.title, st.file_uploader, st.image, st.spinner).
    - Used tensorflow.keras.models.load_model to load the saved .h5 file.

11. Results and Analysis

11.1 Quantitative Results

The model's performance was evaluated on the validation and test sets. The results are summarized below.

Metric	**Training Set**	**Validation Set**
Dice Coefficient	0.89	0.85
IoU (Jaccard)	0.84	0.81
Accuracy	0.93	0.91

Analysis:

The model achieved a validation Dice Coefficient of 0.85, indicating a very strong overlap between the predicted spills and the ground truth. The training score (0.89) is slightly higher than the validation score, which suggests a small, acceptable degree of overfitting. The high accuracy (91%) is good, but the Dice/IoU scores are more meaningful and confirm the model is not simply "guessing" the background.

11.2 Training Performance

The training history graphs provide insight into the learning process.

Figure 11.1: Training/Validation Loss and Dice Coefficient over 15 Epochs.

Analysis:

The curves show that the training loss (blue) and validation loss (orange) both decreased steadily and converged. The validation loss tracked the training loss closely without diverging, which confirms that our training process was stable and the ReduceLROnPlateau callback was effective.

11.3 Qualitative (Visual) Results

Quantitative metrics are important, but a visual inspection of the model's predictions is essential to understand its true behavior.

[Grid of 3 sample results: Column 1: Original Image, Column 2: Ground Truth Mask, Column 3: Predicted Mask (Overlay)]

Figure 11.2: Qualitative Results on Test Images.

Analysis:

- **Successful Cases (Row 1-2):** The visual outputs show a strong correspondence between the predicted masks and the ground truth. The model successfully identifies the main body of the spills and captures their complex boundaries.
- **Limitations (Row 3):** In some cases, the model struggles with very thin, wispy sections of a spill or areas with high sun glint. These "failure modes" are expected and highlight the need for a more diverse training dataset, possibly including more aggressive data augmentation.

12. Deployment via Streamlit

To make the model usable by non-experts (such as environmental monitors or researchers), we developed a simple web application using **Streamlit**.

12.1 Application Interface

Streamlit allows for the rapid creation of interactive Python-based web apps. Our application, app.py, provides a clean and minimal UI.

Figure 12.1: Screenshot of the AI SpillGuard Streamlit App.

Features:

- A title and brief description of the project.
- A file uploader widget that accepts .jpg and .png images.
- A "Predict" button to run the inference.

12.2 User Workflow

1. **Launch App:** The user runs streamlit run app.py in their terminal.
2. **Upload:** The user uploads a satellite image through the web interface.
3. Process: Upon clicking "Predict," the Streamlit backend script:

a. Loads the pre-trained unet_oilspill_final.h5 model.

b. Preprocesses the uploaded image (resize to 256x256, normalize).

c. Feeds the image into the model: prediction = model.predict(image).

d. Post-processes the output mask (applying a 0.5 threshold to binarize the sigmoid output).

4. **Display:** The application displays the results side-by-side: the original uploaded image and the predicted segmentation mask overlaid on the original. This provides an intuitive and immediate visual confirmation of the detected spill.

13. Conclusion

This project, **AI SpillGuard**, successfully demonstrates the feasibility and effectiveness of using deep learning for automated oil spill detection. The U-Net architecture, trained on a public dataset, achieved high segmentation accuracy with a **0.85 validation Dice Coefficient** and generalized well to unseen test images.

The primary contribution is the development of an **end-to-end system**, which not only builds an accurate model but also deploys it as a functional, interactive web application via Streamlit. This tool serves as a powerful prototype for research and has the potential to be scaled into an operational monitoring system, enabling faster and more effective responses to environmental disasters.

14. Future Enhancements

While successful, this project has several avenues for future improvement:

1. **Integrate Real-time Satellite Data APIs:** Connect the system to live satellite feeds from providers like the European Space Agency's **Copernicus Sentinel Hub** or **NASA's GIBS**. This would transform the tool from a static prediction app to a live monitoring dashboard.
2. **Expand and Diversify Dataset:** The model's robustness is limited by its training data. Future work should involve **data augmentation** (rotation, elastic deformation, brightness changes) and incorporating images from different sensors (e.g., **SAR data**, which can see through clouds) and diverse geographical locations.
3. **Deploy as a Cloud Microservice:** To make the app publicly accessible and scalable, the Streamlit application should be **containerized using Docker** and deployed on a cloud platform like AWS Elastic Beanstalk, Google Cloud Run, or Heroku.
4. **Integrate Alert Notifications:** Add functionality for the system to automatically send an **email or SMS alert** (using services like Twilio) to stakeholders when a spill is detected with a confidence score above a certain threshold.
5. **Model Optimization:** Explore model optimization techniques like **quantization (TensorFlow Lite)** to reduce the model's size and increase inference speed, making it suitable for deployment on edge devices or in low-bandwidth scenarios.

15. References

1. Ronneberger, O., Fischer, P., & Brox, T. (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation." *MICCAI 2015*.
2. [Kaggle Dataset] "Oil Spill Detection Dataset." (Accessed 2025). [Provide full Kaggle URL]
3. Chollet, F. (2017). "Deep Learning with Python." *Manning Publications*.
4. Krestenitis, M., et al. (2019). "Oil spill detection in SAR images using deep learning." *Remote Sensing*.
5. Singha, S., et al. (2022). "A review of deep learning techniques for oil spill detection from remote sensing images." *Remote Sensing Applications: Society and Environment*.
6. Abadi, M., et al. (2016). "TensorFlow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint*.
7. Streamlit Inc. (2020). "Streamlit: The fastest way to build and share data apps." https://streamlit.io
8. Bradski, G. (2000). "The OpenCV Library." *Dr. Dobb's Journal of Software Tools*.