MŰEGYETEM 1782

Project work:

Automated Flood Detection based on Satellite Images

Deep Learning (BMEVITMMA19)

“Magellan”

Names	Neptun	E-mail
Szladek Máté Nándor	TGPZTT	szladekmate@gmail.com
Tóth Ádám László	TK6NT3	adamlaszlototh@edu.bme.hu

Introduction and Related Work

The main goal of this project is to detect flooded areas using satellite images from the SEN12-FLOOD dataset, which comes from the Sentinel-2 satellite. Flood detection is really important because it helps reduce damage from natural disasters and allows for better early warning systems. Sentinel-2's multispectral images contain detailed information about land surfaces, vegetation, and water, but these images can be tricky to analyze because they're so complex. By focusing on just a few of the satellite's spectral bands (Blue, Green, Red, and NIR), we can simplify the problem and more easily pinpoint flooded regions. The NIR (Near-Infrared) channel is useful because it can pass through thin clouds and is good at detecting plants and water. This makes it helpful for things like flood detection and classifying different types of land.

Earlier this year, Hungary had a major flood. We knew that flooding was likely, but it is also important to monitor areas that are ecologically sensitive, even if they are not populated. The damage caused by the flood made it clear that we need faster and more accurate ways to find the early stages of flooded areas. This event inspired the motivation for this project.

The SEN12-FLOOD dataset (Rambour et al., 2020) is great because it includes both radar (Sentinel-1) and optical (Sentinel-2) data specifically prepared for flood detection tasks. There are also other MSc projects and open-source efforts, such as the one by KonstantinosF on GitHub (https://github.com/), which provided a helpful reference for data organization and baseline model approaches. In our case, we're focusing solely on Sentinel-2 optical data to see how much we can achieve without radar inputs. With our solutions, we achieved better results compared to KonstantinosF's approach, but the performance could still be improved. It might be worth considering including radar images as well to further improve the results.

We explore two main approaches:

1. **Solution 1 (Upgraded Baseline):** A simple CNN built with TensorFlow/Keras, which evolved into an exceptionally effective solution after fine-tuning and further data engineering. This model provides a strong foundation and delivers excellent results.
2. **Solution 2 (Advanced Model):** A more complex PyTorch Lightning model incorporating additional techniques such as input normalization, class balancing, dropout for regularization, and early stopping. While the simpler model turned out to be superior, the advanced model has the potential to leverage larger and more detailed datasets more effectively, offering greater advantages with further optimization.

Dataset and Preprocessing

Data Source:

We used the SEN12-FLOOD dataset, which includes both radar (Sentinel-1) and optical (Sentinel-2) images. For this project, we focused only on the Sentinel-2 bands: Blue (B02), Green (B03), Red (B04), and NIR (B08). These four bands are commonly chosen because they help distinguish between different types of land cover—like telling water apart from vegetation—based on how they reflect or absorb light.

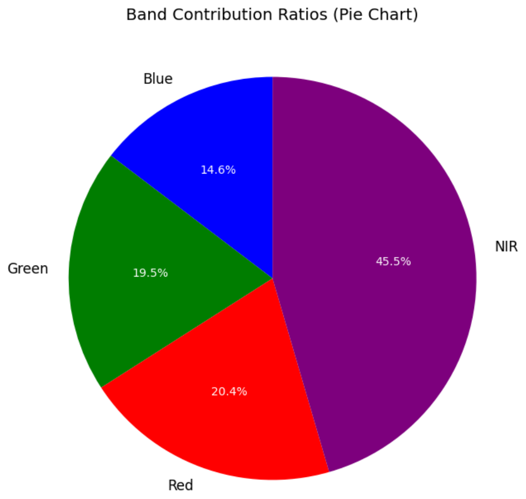
Preprocessing Steps:

- **Directory Restructuring:**
  We reorganized the dataset so that each scene’s Sentinel-2 bands are kept together, and we removed any files we weren’t going to use (like Sentinel-1 data).
- **Quality Check and Removing Empty Scenes:**
  Some scenes contain no actual information (for example, images that are basically all zeros). We identified and discarded these, ensuring our training data actually has meaningful content.
- **Band Stacking:**
  Instead of dealing with multiple separate files per scene, we combined the four bands (Blue, Green, Red, NIR) into one TIFF file (stack.tif). This makes loading and processing the data a lot simpler.
- **Label Insertion:**
  Each scene’s folder gets a small text file (flooding.txt) that tells us whether there’s flooding (“True”) or not (“False”). This gives us a quick and easy way to associate each image with the correct label.
- **Data Augmentation (Optional):**
  To boost our training data, we applied simple transformations like flips and rotations. This created an “augmented” version (astack.tif), effectively doubling the number of training samples and helping the model learn more robust features.

Data Analysis

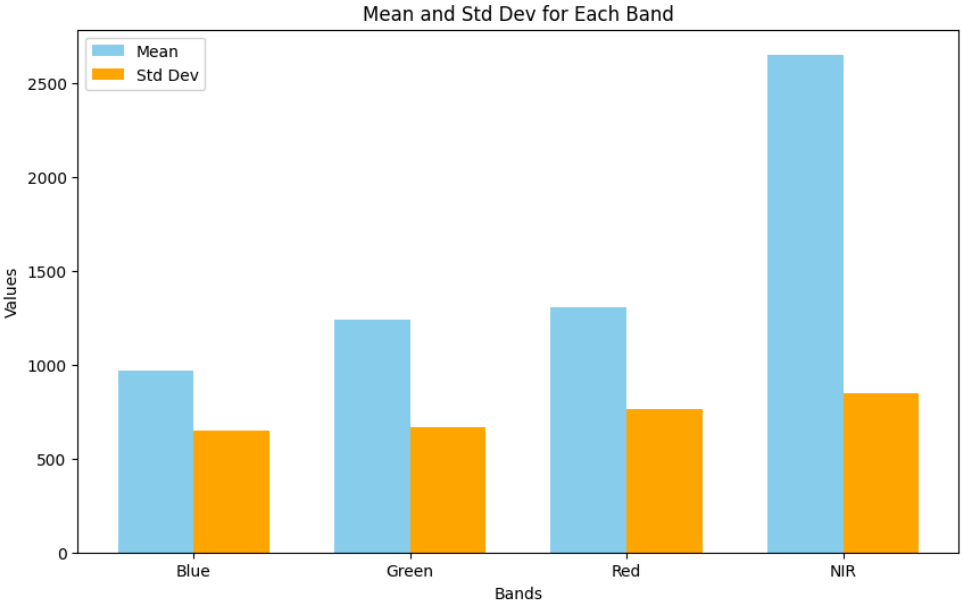
Before making any major changes to our approach, we took a closer look at the data to understand its characteristics and distribution. The data analysis took place after we had already built and tested our first, simpler model. Our main goal here was to figure out what could be improved in the second model. Ironically, as we’ll see later, even though we made adjustments based on these insights, the first model ended up performing better in the end.

Label Distribution:
We confirmed that around 23% of the scenes were labeled as flooded, making flood images the minority. This highlighted the class imbalance issue. Our initial (baseline) model didn’t explicitly address this, potentially skewing its predictions. We assumed that applying techniques like weighted sampling in our second model would help, but, as it turned out, addressing imbalance didn’t necessarily guarantee improved results. Sometimes, simpler models might benefit from natural dataset characteristics in ways we don’t fully anticipate.



Band Contributions and Statistics:
We also examined how each spectral band (Blue, Green, Red, and NIR) contributed to the final stacked images. We expected this information to

guide us in refining the second model’s approach and also simply provide interesting insights. The logic was that if each band was providing meaningful input, then normalizing or weighting bands differently might help. The below pie chart shows the contributions for each band, highlighting the strong influence of the near infrared band, which on it’s own contributed almost as much data to the stacked picture as the RGB bands all-together.



This chart shows both the average (mean) and the variation (standard deviation) of pixel intensities for each of the four spectral bands we’re using: Blue, Green, Red, and NIR. The light blue bars represent the mean values, while the orange bars represent the standard deviations.

- **Blue Band:**
  Has the lowest mean intensity compared to the others, and a moderate standard deviation. This suggests that the blue channel is generally dimmer than the others on average, but still varies quite a bit between scenes.
- **Green Band:**
  The mean intensity is higher than Blue, and the standard deviation is also significant. This indicates that the green channel often reflects more light than Blue does, but the amount of variability across different scenes or surfaces is still noticeable.
- **Red Band:**
  Falls somewhere between Green and NIR in terms of mean intensity. Its standard deviation is also in a moderate range, meaning it’s not as variable as some other bands but still changes enough to matter.
- **NIR Band:**
  Stands out with the highest mean intensity by a large margin, and it also has a relatively large standard deviation. This suggests that the NIR channel captures a lot more reflected light on average, but its values can differ quite a bit from one scene to another. This is actually expected since vegetation strongly reflects NIR light, so areas with dense vegetation or certain surface types can really push that average up.

Model Architectures

Solution 1 - Keras Model:

- **Model Architecture:** Two convolutional layers (32 filters, (3, 3) kernel, ReLU activation) with max-pooling, followed by a flattening layer, a dense layer with 128 ReLU units, and a softmax output layer. Input shape: (512, 512, 4).
- **Training Parameters:**
    - Optimizer: Adam (default parameters).
    - Loss: Sparse categorical crossentropy.
    - Metric: Accuracy.
    - Epochs: 10.
    - Batch size: 28 (optimized for memory**).**
- **Monitoring:** TensorBoard is integrated for real-time training visualization with dynamically generated log directories.
- **Gradio:** Real time classification. Uploaded images will be classified by the loaded model.

Solution 2 - PyTorch Model:

- **Model Architecture:** First layer: 16 filters, (3, 3) kernel, ReLU activation, and batch normalization. Second layer: 32 filters, (3, 3) kernel, ReLU activation, and batch normalization.
  Max-pooling layers after each convolution.
  Softmax activation for binary classification.
  Dropout (30%) for regularization.
  Xavier initialization applied to all convolutional and linear layers to stabilize gradient flow.
  Multichannel Input: Supports 4 Sentinel-2 spectral bands (blue, green, red, and infrared) The iInput shape: (512, 512, 4)
- **Training Parameters:**
    - Normalization: Pixel values are normalized using (ImageNet) mean=[0.485, 0.456, 0.406, 0.406] and std=[0.229, 0.224, 0.225, 0.225], extended for multichannel input.
    - Optimizer: Adam with a learning rate of 1e-5.
    - Loss Function: Sparse categorical cross-entropy.
    - Metric: Accuracy for both training and validation.
    - Batch Size: Fixed at 32 to fit memory constraints.
    - Class Balancing: A WeightedRandomSampler addresses class imbalance using inverse class counts [2186, 668].
    - Early Stopping: Training stops after 3 epochs of no validation loss improvement.
    - Epoch: 40. The model converged at epoch 3.
- **Learning Rate Scheduling:** ReduceLROnPlateau dynamically adjusts the learning rate when validation loss plateaus, halving it after two stagnating epochs.
- **Monitoring:** WandbLogger: Logs metrics to the "Magellan" project.
    - WandB Report WebURL: https://api.wandb.ai/links/szladek-mate-kutat-di-kok-mozgalma/u259pzzi

Splitting Data into Training and Test Sets

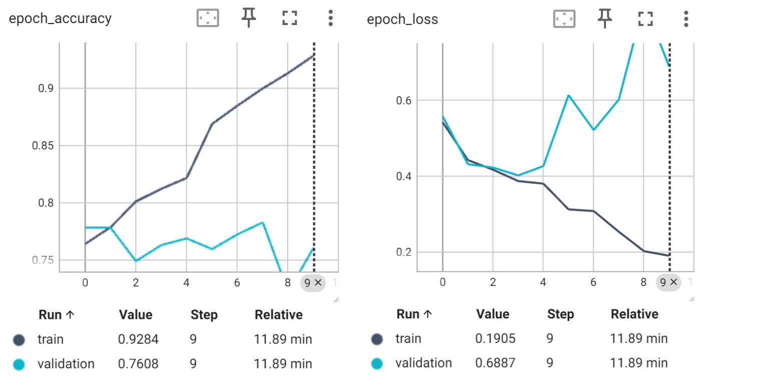
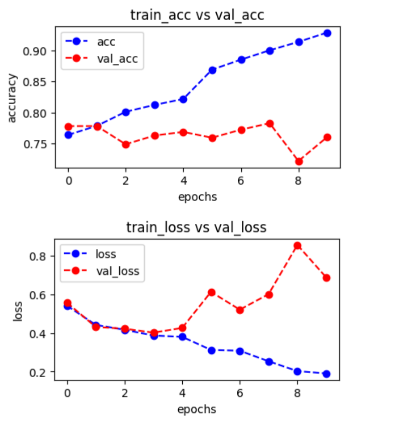
After all the preprocessing, we split the data into 70% for training and 30% for testing. We kept a fixed random seed to make the splitting reproducible (so we can always get the same train/test sets if needed). The training set includes both the original and the augmented images, while the test set just has the original data. This ensures that our test results reflect how well the model can handle unseen images without any artificial adjustments.

Training Process

For both solutions, image data is loaded into memory, normalized to [0,1]. The Keras Model is trained using the Keras fit() method, while the PyTorch Model leverages PyTorch Lightning's Trainer. During training, accuracy and loss are monitored. In case of the Keras Model Tensorboard is used for live visualization, while the PyTorch Model uses Weights & Biases (wandb) logging to track progress and visualize metric trends over epochs. For hyperparameter optimization we used a manual trial-and-error method. One of the main future improvement points would be adding automated hyperparameter optimization, this could lead to significantly better performance. In our final training we used the parameters mentioned in the previous chapter (Model Architectures).

We trained both models locally on personal computers using Jupyter Notebooks running in Visual Studio Code. Throughout the entire training process, we relied on CPU computation, no dedicated GPU resources were used. This choice meant that training times were longer than they would have been with GPU acceleration, but since our dataset and models were relatively manageable, the CPU approach was still feasible.

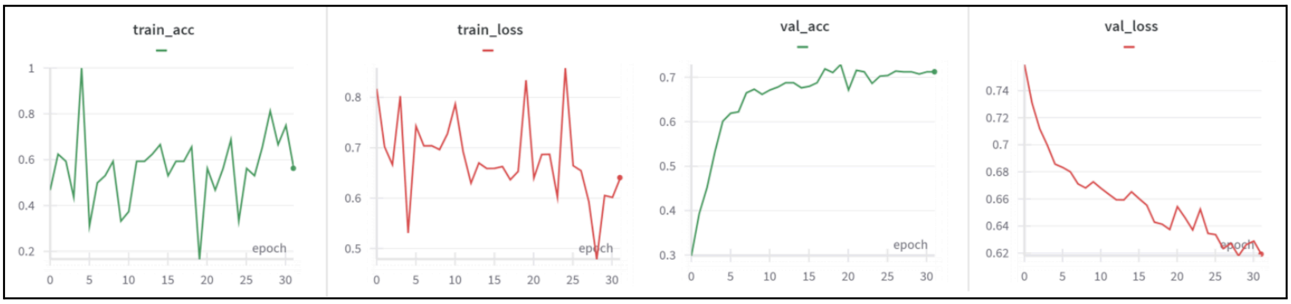
After the training was complete, we saved our trained models to files on our local machine. This step lets us easily load the models later without having to retrain them from scratch. With the saved model files, we can quickly run evaluations, generate predictions, and test the model’s performance. It also makes it simpler to share our trained models with others, since they don’t need to reproduce the entire training process—just load up the saved weights and start making predictions.

Keras Model training progress visualized in Tensorboard:	**Keras Model** training progress visualized locally:
	

Run ↑	Value	Step	Relative
● train	0.9284	9	11.89 min
● validation	0.7608	9	11.89 min

Run ↑	Value	Step	Relative
● train	0.1905	9	11.89 min
● validation	0.6887	9	11.89 min

PyTorch Model training progress visualized in WandB:



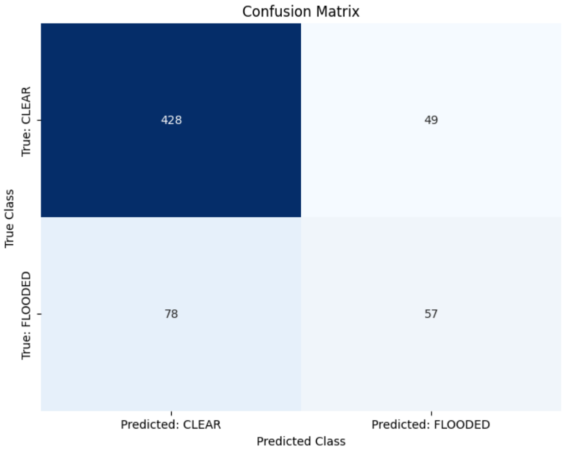
Evaluation Metrics and Analysis

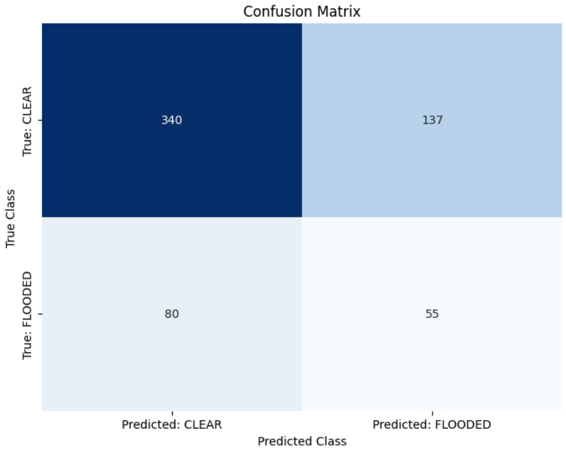
Flood detection is formulated as a binary classification task: the model predicts either "FLOODED" or "CLEAR." In most of the pictures, patterns that would suggest flooding are not visible to the human eye, so it was interesting to see how machine learning models are more capable of performing this task. By learning from large amounts of labeled training data, these models become sensitive to even the smallest cues, enabling them to identify conditions associated with flooding far more consistently and accurately than a human might by eye alone. Both models were evaluated using the same metrics and visualizations, which are explained and compared in the following sections.

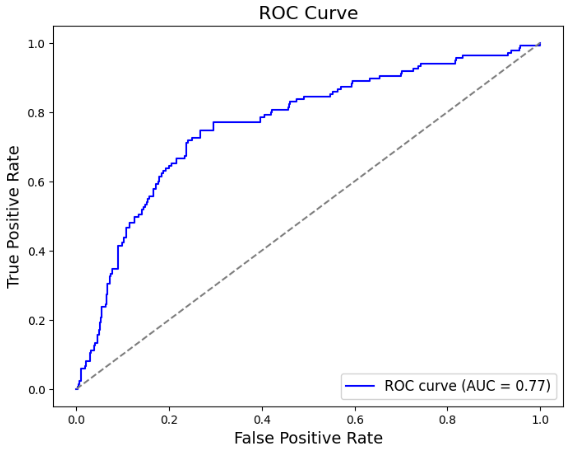
- **Accuracy:** The ratio of correctly classified samples.
- **Confusion Matrix:** Provides insight into the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).
- **Precision, Recall, and F1-Score:** Given class imbalance, these metrics are critical. Precision measures how many predicted floods are correct, recall measures how many actual floods are identified, and F1-score balances both.
- **ROC and PR Curves:** The ROC curve (with AUC) and Precision-Recall curve offer threshold-independent views of model performance. The PR curve is especially informative when dealing with class imbalance.

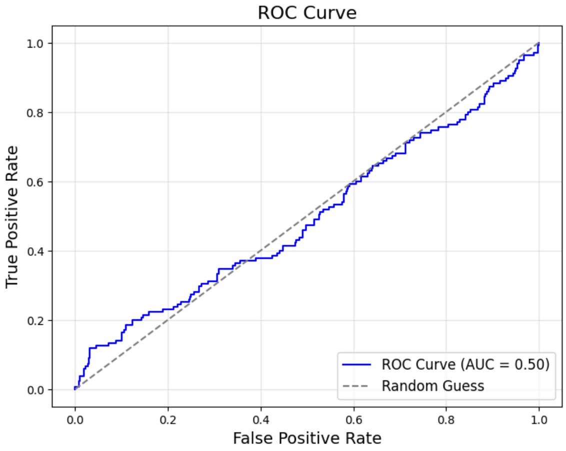
	Accuracy	Precision	Recall	F1-Score	AUC
Keras Model	**0.79**	**0.54**	**0.42**	**0.47**	**0.77**
PyTorch Model	**0.65**	**0.29**	**0.41**	**0.34**	**0.5**

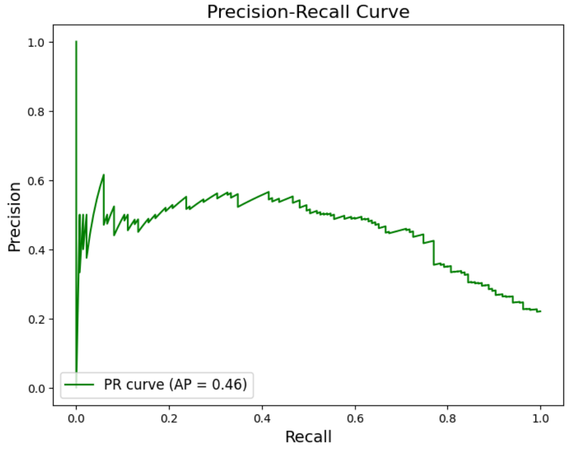
Comparing the two models reveals a clear advantage for the Keras approach in this specific task. The Keras model not only achieved a higher overall accuracy (0.79 vs. 0.65) but also showed stronger performance in precision (0.54 vs. 0.29) and F1-score (0.47 vs. 0.34), indicating it more effectively balanced identifying flooded scenes while minimizing false alarms. Although both models had room for improvement in recall (0.42 vs. 0.41), the Keras model’s relatively higher AUC (0.77 vs. 0.50) further emphasizes that it consistently ranked flooded versus clear samples better than random guessing.

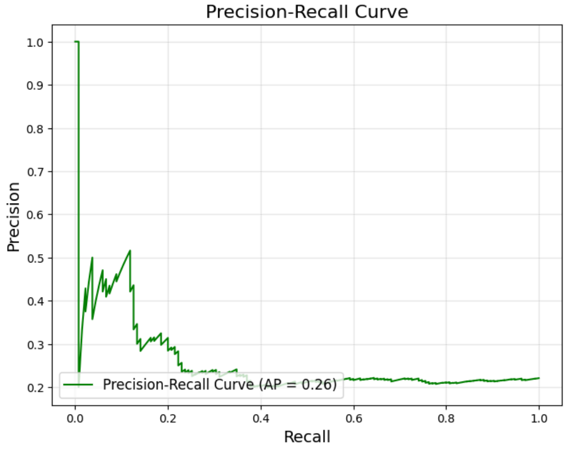

Confusion Matrix
True Class
True: CLEAR
True: FLOODED
428
49
78
57
Predicted: CLEAR
Predicted: FLOODED
Predicted Class


Confusion Matrix
True Class
True: CLEAR
True: FLOODED
340
137
80
55
Predicted: CLEAR
Predicted: FLOODED
Predicted Class


ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.77)


ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC Curve (AUC = 0.50)
Random Guess


Precision-Recall Curve
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0
PR curve (AP = 0.46)


Precision-Recall Curve
Precision
Recall
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0.0
0.2
0.4
0.6
0.8
1.0
Precision-Recall Curve (AP = 0.26)

The diagrams show how each model (Keras vs. PyTorch) performed at flood detection:

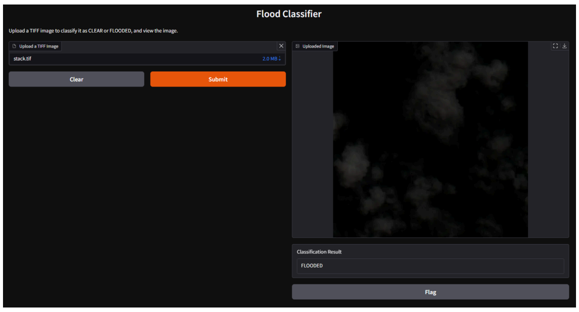
Confusion Matrices: The Keras model's matrix (left) has fewer errors, correctly classifying more flooded and clear scenes. The PyTorch model's matrix (right) shows more mistakes, with many clear scenes predicted as flooded and fewer floods identified correctly.

ROC Curves: The Keras model's curve (left) rises well above the random guess line, with an AUC of 0.77, indicating good discrimination. The PyTorch model's curve (right) stays near the diagonal (AUC ≈ 0.50), meaning it's barely better than guessing.

Precision-Recall Curves: The Keras model maintains higher precision at various recall levels (AP ≈ 0.46), while the PyTorch model's curve (AP ≈ 0.26) shows it struggles to balance correct flood detections with false alarms.

Overall, the Keras model outperforms the PyTorch model, identifying flooded scenes more reliably and making fewer incorrect predictions.

Gradio Interface

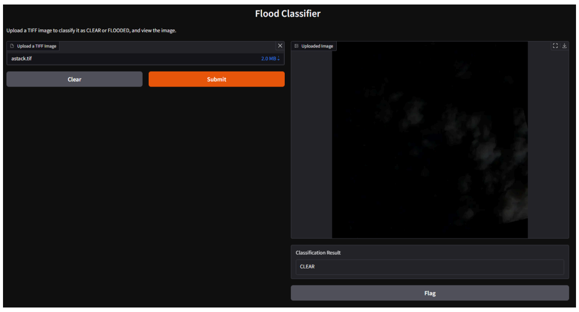


Flood Classifier

Upload a TIFF image to classify it as CLEAR or FLOODED, and view the image.

Upload a TIFF Image: stack.tif 2.0 MB

Classification Result: FLOODED



Flood Classifier

Upload a TIFF image to classify it as CLEAR or FLOODED, and view the image.

Upload a TIFF Image: astack.tif 2.0 MB

Classification Result: CLEAR

The implemented Gradio interface provides a user-friendly way to interact with the trained Keras model that classifies Sentinel-2 TIFF images into "CLEAR" or "FLOODED." When a user uploads a TIFF image, the interface automatically preprocesses it—normalizing its pixel values and ensuring it matches the model's expected input dimensions. Afterward, the model generates a prediction, and the interface displays both the original image and the predicted class label. This setup makes it easy for users to experiment with various TIFF files and quickly see classification results without any additional coding or command-line interaction.

In practical terms, this means that anyone, even without deep technical knowledge, can test the model's performance directly from their web browser. The uploaded image is analyzed on the spot, and the output is clearly displayed next to the input image. Users can visually confirm what they have uploaded and immediately review the model's guess—making the model evaluation process more intuitive, transparent, and accessible.

Conclusion

The goal of this project was to first define a strong baseline model for flood detection using Sentinel-2 imagery and then build a more advanced model to improve upon it. Our initial baseline, built with Keras, evolved into a highly effective solution as we refined it through further tuning and data engineering. Meanwhile, we designed a more complex PyTorch model incorporating advanced techniques such as class balancing, normalization, and dropout. Although the PyTorch model did not outperform the improved Keras baseline, this result highlights potential areas for further exploration rather than the limitations of the approach itself. The Keras model achieved better results across all key metrics, including accuracy, precision, and AUC (0.77 vs. 0.50), demonstrating that simplicity paired with iterative improvement can deliver strong performance. On the other hand, the PyTorch model has significant untapped potential. With more time and resources, hyperparameter optimization, better normalization strategies, and advanced data augmentation could improve its performance. Additionally, the quality and characteristics of the Sentinel-2 visual data play a critical role in training both models. If the data is inherently noisy or lacks sufficient representation of flooding, it could limit the model's ability to generalize. Exploring higher-quality datasets or supplementing with radar data could unlock the full potential of the PyTorch model. While the PyTorch model struggled in this case, the Keras model provided a reliable and practical solution for flood detection, demonstrating that simplicity and careful design can often deliver strong results. This project underscores the importance of understanding both the data and the models in use, and the lessons learned here offer valuable directions for future work, including improving data quality and further optimizing advanced model architectures.

References:

Rambour, C., Audebert, N., Koeniguer, E., Le Saux, B., Crucianu, M. & Datcu, M., 2020. *SEN12-FLOOD: a SAR and Multispectral Dataset for Flood Detection*. [online] Available at: https://dx.doi.org/10.21227/w6xz-s898 [Accessed 8 December 2024].

ClmRmb, 2020. *SEN12-FLOOD Dataset Description*. [online] Available at: https://github.com/ClmRmb/SEN12-FLOOD [Accessed 8 December 2024].

KonstantinosF, 2021. *Flood Detection in Satellite Images*. [online] Available at: https://github.com/KonstantinosF/Flood-Detection---Satellite-Images [Accessed 8 December 2024].

Rambour, C., Audebert, N., Koeniguer, E., Le Saux, B. & Datcu, M., 2020. *Flood Detection in Time Series of Optical and SAR Images*. *ISPRS Archives*, [online] Available at: https://isprs-archives.copernicus.org/articles/XLIII-B2-2020/1343/2020/isprs-archives-XLIII-B2-2020-1343-2020.pdf [Accessed 8 December 2024].

We have used ChatGPT for debugging the code, and occasionally here for text generation.