We wanted a project that’s **fast, doable in a couple of days, but actually impressive**. Most of the ideas we brainstormed either took too long or were too simple to matter. Then it clicked: why not focus on **hurricane flood detection and population impact**? This way, we can combine **AI, satellite imagery, and real population data** to show something meaningful like how many people were affected.

We chose a **Siamese U-Net model** for flood detection because it’s powerful enough to catch changes between pre- and post-hurricane images, but still lightweight for a short demo. We also included **NDWI and MDWI baseline methods** to provide a classic comparison, ensuring our results are grounded. Finally, we integrated **real WorldPop population data** to calculate affected populations and create detailed visualizations. In just two days, we can go from raw satellite images to a full, human-readable analysis maps, risk scores, and numbers that tell a real story.

**Data Gathering and Processing**

**Hurricane Data Pipeline for Florida Flood Analysis**

**Initial Questions and Challenges**

**Why Hurricane Ian?**

I had to choose a hurricane for the project, and Ian immediately stood out:

1. **Recent and well-documented**: September 2022 landfall, so a lot of satellite coverage exists.
2. **Severe impact**: Category 4 storm, massive flooding, perfect for a case study.
3. **Clear timeline**: Landfall date is precise, so I could define exact pre/post periods.
4. **Data availability**: HLS2 (Harmonized Landsat Sentinel-2) seemed like a reliable source.

Initially, I didn’t know if I could actually access all the required imagery, but I had to start somewhere.

**Learning HLS2 and Choosing Bands**

At first, I considered using raw Landsat or Sentinel imagery. But after reading NASA documentation, I realized **HLS2 is already “analysis-ready”**:

* Atmospheric correction
* Geometric alignment
* Radiometric calibration
* Cloud masking

This saved me **weeks of preprocessing**.

**Band selection reasoning:** For flood detection and NDWI computation:

* **B02 (Blue, 490nm)** – water sensitivity
* **B03 (Green, 560nm)** – NDWI numerator
* **B04 (Red, 665nm)** – vegetation contrast
* **B8A (NIR, 865nm)** – vegetation/water separation
* **B11 (SWIR1, 1610nm)** – soil moisture
* **B12 (SWIR2, 2190nm)** – cloud discrimination

**The Authentication Nightmare**

NASA Earthdata requires authentication. At first, the **documentation didn’t match reality**, and earthaccess.login() threw errors until I carefully set credentials. After some trial-and-error and Stack Overflow help, I got it working reliably.

import earthaccess

auth = earthaccess.login()

**Defining the Study Area**

I needed a realistic bounding box for **computational efficiency**:

* Chose **Southwest Florida** (Fort Myers/Naples) – ground zero for Ian
* Mix of urban, suburban, natural areas
* Bounding box: [-82.8, 25.8, -81.2, 27.5]
* Covers ~150km x 200km – manageable for local processing

**Temporal Sampling Strategy**

At first, I was only thinking of **single pre/post periods**, but I realized flooding evolution matters. I iterated through different strategies:

1. **Single pre/post period** – too simplistic
2. **Weekly sampling** – captures dynamics, manageable data size
3. **Monthly sampling** – optional for larger trends

Ultimately, **weekly sampling** worked best:

* 4 weeks before → pre-event
* 4 weeks after → post-event
* Gives 8 periods for modeling

**Download Strategy and API Learning Curve**

Each HLS2 granule is **200-800 MB**. My home network couldn’t handle all files at once.

* Strategy: limit downloads to top 5 per period
* Filtered by cloud coverage
* Batch download overnight

I also discovered a **key API nuance**: bounding\_box=(\*bbox,) instead of bbox= and temporal strings instead of datetime objects. These little things were huge blockers at first.

**Preprocessing Pipeline Evolution**

**Iteration 1:** Tried to do atmospheric correction, geometric alignment myself → redundant and time-consuming

**Iteration 2:** Focused on handling:

* Fill values (-9999 → NaN)
* Scaling (HLS2 ×0.0001)
* Median composite creation (robust to clouds)
* Normalization (0–1 range for deep learning)

**Median composite** was chosen over mean:

* Robust to cloud outliers
* Preserves surface reflectance
* Handles partial cloud coverage well

**Normalization:** Tried min-max and z-score, but **percentile normalization** was most robust:

p2, p98 = np.nanpercentile(image\_array, (2, 98))

normalized = (image\_array - p2) / (p98 - p2)

normalized = np.clip(normalized, 0, 1)

**Handling Real-World Imperfections**

* Sometimes no valid files for a band → created **dummy bands** (0.1 values) to keep array shapes consistent
* Even preprocessed HLS2 can have **NaNs, small patches, or unusual ranges**
* Quality checks: flagged excessive NaNs, constant bands, unusual ranges

**Final Pipeline Decisions**

* **Scalable pipeline** for multiple hurricanes & flexible temporal sampling
* Automated **search → download → composite → normalize → quality check → save**
* Metadata tracking for all hurricanes, periods, and band statistics
* Backup plan: handle missing or problematic bands

This led directly to the final data\_pipeline\_2.py. Every function now has **robust handling** for real-world messy data, without crashing the workflow.

Configured and ran a **scalable hurricane data pipeline**  
Implemented **weekly temporal sampling** pre/post hurricane  
**Downloaded, processed, and normalized** HLS2 imagery  
Created **median composites** for each band  
Performed **quality checks and dummy band handling**  
Saved processed arrays and full **metadata for reproducibility**

**Files created:**

* processed.npy composites for each period
* processing\_metadata.json – full log of hurricanes, periods, and processing
* Patch-ready arrays for deep learning

A screenshot of a computer

AI-generated content may be incorrect.

**Deep Learning Model Design and Training**

**1. Project Goals**

* Build a **robust deep learning pipeline** to detect hurricane-induced floods using pre/post-event satellite composites.
* Ensure the **network is resilient to noisy, partially missing data**.
* Incorporate **temporal dynamics** and **patch-level sampling** to balance flood vs non-flood pixels.
* Make training **stable and reproducible**, avoiding NaNs and gradient explosions.

**2. Model Architecture Decisions**

**2.1 Why a Siamese U-Net?**

* **Siamese encoder**: Processes pre- and post-event composites through shared weights, ensuring **feature consistency**.
* **U-Net structure**: Combines **hierarchical features** with skip connections to preserve **spatial context** of floods.
* **Feature fusion at bottleneck**: Concatenating the deepest features from pre/post images allows the network to learn **change representations** rather than absolute reflectance.

**2.2 Robustness Enhancements**

* **BatchNorm2d after convolutions** → stabilizes training across varying satellite scenes.
* **Kaiming initialization** → prevents vanishing/exploding gradients at the start.
* **Dropout2d in encoder** → reduces overfitting on small flood areas.
* **Adaptive padding** → ensures input sizes divisible by 16, prevents shape mismatch in upsampling.

**2.3 Decoder Choices**

* Skip connections from **post-event encoder only** → network learns change relative to baseline.
* **ConvTranspose2d** for upsampling → preserves spatial resolution while merging features.
* Single **sigmoid output** per pixel → flood probability map.

**3. Data Handling & Patches**

**3.1 Patch Generation Strategy**

* Full satellite images are too large for GPU memory → generate **256×256 patches**.
* Balanced sampling:
  + 50% patches contain flood pixels
  + 50% patches contain no-flood pixels
* Labels generated from **NDWI differences** between pre/post composites.
* Avoided inefficient random searching by **sampling directly from flood/non-flood coordinates**.

**3.2 Dataset Robustness**

* NaN values → replaced with 0.0
* Inf values → clipped to 1.0
* Ensures **network never receives invalid inputs**, preventing training crashes.

**4. Loss Function Choice**

* **Balanced Focal Loss**: Handles **class imbalance** (flood pixels are rare).
* Parameters tuned:
  + alpha=0.5 → balance between foreground/background
  + gamma=2.0 → focus on hard examples
  + pos\_weight\_factor=10.0 → explicit weighting of rare flood pixels
* This improves **sensitivity to small flood areas** while preventing the network from collapsing to predicting "no flood everywhere."

**5. Training Pipeline Enhancements**

**5.1 Data Splitting**

* **Critical fix**: Split pre/post composites separately to ensure **balanced validation sets**.
* 80/20 split with **at least one file in validation** → avoids empty validation sets on small datasets.

**5.2 Trainer Design**

* **AdamW optimizer** → better generalization and weight decay handling.
* **ReduceLROnPlateau scheduler** → lowers learning rate on plateau, improves convergence.
* Metrics tracked per epoch: F1, precision, recall → essential for imbalanced classification.

**5.3 Device Flexibility**

* Supports **MPS (Apple Silicon GPU)** or CPU fallback → ensures portability.

**6. Debugging & Iterative Decisions**

1. **Padding issues**:
   * Initially, upsampled feature maps did not match encoder skip shapes → added AdaptivePadding to fix mismatches.
2. **NaN losses**:
   * Encountered during early training → solved by np.nan\_to\_num in dataset loader.
3. **Flood patch scarcity**:
   * Early random sampling yielded mostly no-flood patches → switched to **coordinate-based sampling** for guaranteed representation.
4. **Validation stability**:
   * Early F1 scores were erratic → separated pre/post files during splitting to avoid leakage.
5. **Memory and speed optimizations**:
   * Limited patch generation per image (patches\_per\_image) → avoids GPU OOM
   * Minimized unnecessary copies and numpy array slicing → faster batch loading.

**7. Training Outcomes**

* **Training runs for 15 epochs** → convergence observed in loss and F1 metrics.
* **Best model saved automatically** on highest F1-score.
* **Metrics visualization**: Training/validation loss, F1, precision, recall plotted and saved.
* Provides **immediate insight into performance** and potential overfitting.

**8. Next Steps**

* Expand to **multiple hurricanes** (already supported by pipeline)
* Include **population impact maps** and overlay flood predictions
* Test **other change detection metrics** (e.g., MDWI) as additional supervision
* Potential **real-time deployment** for new hurricane events

A screenshot of a computer

AI-generated content may be incorrect.

**What we were trying to do**

We wanted to figure out **how many people were affected by a hurricane** using satellite images and population data. The idea is:

1. Detect flooded areas with a **Siamese U-Net**.
2. Overlay **WorldPop population data** to see how many people are in those flooded regions.
3. Do some basic statistics: total population affected, urban vs rural breakdown, etc.

Sounds simple, but the real data is messy.

**Problems we ran into**

* **WorldPop data is huge** (1 km resolution for the whole USA). Can’t just load it all at once.
* Files sometimes have **NaNs, negative numbers, or absurdly high values**.
* Clipping the data to the hurricane area is tricky because of **coordinate systems** and **windowing**.
* Memory limits meant we couldn’t just brute-force process the whole dataset.

So we had to come up with a **smart, reliable way to handle population data**.

**What we did**

1. **Download & Progress**
   * download\_with\_progress() downloads files in chunks, shows progress.
   * Deletes partially downloaded files if something fails.
   * Why: we need **clean data**, because one corrupted file breaks the analysis.
2. **Validation**
   * smart\_validate\_geotiff() samples multiple regions (corners, center, quarter points) to check the raster.
   * Looks for:
     + Negative-only datasets
     + Missing positive values
     + Extreme outliers (>100,000 people per pixel)
   * Why: just checking the top-left corner isn’t enough—**population data is inconsistent**.
3. **Fixing data**
   * fix\_population\_data() reads the raster in blocks (so we don’t blow up memory).
   * Fixes:
     + NoData values → 0
     + Negative values → 0
     + Extreme values → capped at 99.9th percentile
   * Outputs a **clean GeoTIFF** ready for analysis.
   * Why: prevents garbage in → garbage out in population counts.
4. **Clipping**
   * clip\_to\_bbox\_smart() clips the raster to the hurricane area.
   * Handles coordinate transforms correctly.
   * Returns the clipped raster + total population in that area.
   * Why: We only care about the affected area, not the whole country.
5. **Putting it all together**
   * get\_worldpop\_data() automates the full workflow:
     + Download → Validate → Fix → Clip
   * Keeps track of whether we’re using the **original or fixed file**.

**Integration with Flood Maps**

* Flood maps come from the **Siamese U-Net**.
* Multiply flood mask with population raster → population affected.
* Can compute:
  + Total population affected
  + Non-zero pixels (cells with people)
  + Urban vs rural risk

Basically, **population-at-risk numbers come out automatically**.

**Testing**

* We tested with **Hurricane Ian** ([-82.8, 25.8, -81.2, 27.5]).
* Clipped raster worked, total population > 100k (reasonable for that area).
* Shapes, min/max, non-zero pixels all make sense.
* The pipeline is **ready to plug into the flood model**.

**Goal**

We wanted to **figure out how many people are affected by a hurricane** using satellite imagery + population data. Not just “look at flooded pixels” we want **real-world population impact**, urban vs rural insights, and a baseline comparison with classical methods (NDWI/MDWI).

Basically: **flood maps + population → people at risk**.

**Design Choices / What We Did**

**1. Real Population Data Handling**

* **Data source:** WorldPop (~1km resolution global population grids).
* **Problem:** The raw data is huge, messy, sometimes has NaNs or extreme values.
* **Solution:**
  + Wrote ImprovedWorldPopHandler to **download, validate, and fix** the data automatically.
  + Block processing to handle memory limits (don’t load entire USA at once).
  + Sample multiple regions to **make sure it’s not corrupted** (top-left only isn’t enough).
* **Why:** Garbage in → garbage out. If population is wrong, our flood impact numbers are meaningless.

**2. Flood Detection**

We use **two approaches**:

**A) Deep Learning**

* **Model:** Siamese U-Net (6-channel input for pre/post images).
* **Patch-based processing:**
  + Big images → split into 256×256 patches with 64px overlap.
  + Handles edges and padding to avoid size mismatches.
  + Aggregates patch probabilities into a full flood map.
* **Why:** Large satellite images don’t fit GPU memory. Patches let us scale without crashing.

**B) Baselines**

* **NDWI:** (Green – NIR)/(Green + NIR)
  + Simple, classical method.
  + Thresholded post-event high & pre-event low NDWI → flooded.
* **MDWI:** (Green – SWIR)/(Green + SWIR)
  + Needs SWIR band.
  + Falls back to NDWI if SWIR isn’t available.

**Why:** Compare DL performance to established methods. If DL “fails,” we have a sanity check.

**3. Population Impact Calculation**

* Multiply **flood probability map** with **population raster**.
* Handles mismatched sizes using **resampling/zoom**.
* Calculates:
  + Affected population (# of people in flooded pixels)
  + % of total affected
  + Flooded area (km²)
  + Population density in flooded areas

**Why:** Directly links flood detection to real human impact. Easy to explain to a professor or decision-maker.

**4. Visualization**

* 6-panel figure:
  1. Real population density
  2. Deep Learning flood map
  3. NDWI/MDWI/DL overlay (RGB)
  4. DL-affected population heatmap
  5. Population × flood probability (“risk map”)
  6. Text panel with stats & comparisons
* **Why:** Professors like visuals. Overlay helps compare methods at a glance. Text panel makes numbers easy to read.

**5. File Outputs**

* Downsampled flood probability map → GeoTIFF (dl\_flood\_probability.tif)
* Affected population map → GeoTIFF (dl\_affected\_population.tif)
* **Why:** Can use in GIS software, presentations, or further analysis.

**Why We Made These Choices**

1. **Robustness:** Auto-download/validate/fix population data → no “file missing” errors.
2. **Scalability:** Patch-based DL inference → works on large satellite images without crashing GPU.
3. **Flexibility:** NDWI/MDWI baselines → sanity checks + method comparison.
4. **Accuracy:** Population-weighted flood maps → real human impact, not just pixels.
5. **Usability:** Visualization + GeoTIFF outputs → easy for sharing, reporting, and further analysis.

**What This Pipeline Really Does**

* Takes **satellite images before/after a hurricane**.
* Runs **DL flood detection** (patch-based) and baseline methods (NDWI/MDWI).
* Loads **WorldPop population data**, validates & fixes it.
* Calculates **population affected**, % affected, flooded area, and density.
* Generates **visualizations + GeoTIFFs**.
* Outputs **ready-to-use results for research or reports**.

**Why This Matters**

* Most flood studies stop at “flooded pixels.”
* We **link it directly to people**. This is useful for emergency response, urban planning, and disaster risk assessment.
* With baseline comparison, we can **justify our DL model** as more accurate or reliable than classical indices.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a graph

AI-generated content may be incorrect.