Final Project Report

INTRODUCTION

BACKGROUND

Part of the work that the Stages of Construction team (SOC team) does at Reveal Global Consulting (Reveal) is to track the monthly progress of construction of homes. Instead of relying on surveys and other analog methods, the SOC team realized that they could use satellite imagery, and specifically high-resolution imagery, as their source of data. The benefits of using such imagery are many: the SOC team can cover more ground and work more quickly than any labor-intensive survey campaign could ever hope to achieve. This work, however, comes at a cost, namely, the procurement and analysis of satellite imagery quickly gets expensive, both in terms of money and compute.

To illustrate this further, consider a single target area, like Harris County in Texas. To get data on the construction occurring in this county, the SOC team would need to procure about 1,780 square miles of imagery. This is expensive for many reasons, including the cost to obtain the data, the time and processing cost for storage and retrieval, and the cost in processing the imagery. Even with all the data coming from a large metropolitan area, where a lot of construction would surely be taking place, there will be vast swaths of the landscape that will not get developed. This leads to unnecessary captures of high-resolution imagery. In the case of Harris County, construction activity may only be around 200 square miles, or just over 11% of the total area. Clearly, the SOC team's current data procurement and processing strategy involves a large outlay of resources, some of which could be better spent elsewhere.

PROBLEM STATEMENT

With the issues of procurement apparent, there is an opportunity to cut costs and diminish the time it takes to track construction. The easiest savings are to be found at the front end of the process, i.e., before the SOC team buys the expensive, high-resolution imagery.

To solve this problem, one solution, discussed in this report, would be to train model to act as a "filter" that could scan the proposed capture area and give the SOC team hotspots or change masks. These assets would help the SOC team target their resources for maximum coverage at the lowest cost. The project is hosted on GitHub and can be found here: https://github.com/RevealGC/low-res-change-detection.

SUCCESS CRITERIA

For this project to be a success, it will need to be run cheaply, with easily-obtainable imagery, and have the potential to slot into the existing workflow of the SOC team and other teams at Reveal that need to know where changes have occurred over time in an area.

DEFINITIONS

We should define some key terms that will appear throughout this text:

- A **geohash** is a type of geospatial index that organizes the entire Earth's surface into nested rectangles, each with a unique, short string of letters and numbers.[1] A geohash with the ID 9vgm2 is within the larger geohash of 9vgm and 9vg. Correspondingly, geohashes that share more characters in their prefix are spatially close to each other, e.g., 9vgm0 and 9vgm1 are neighbors, but 9vgm0 and 9vfm0, while being in the same region, are not as close to each other as 9vgm0 and 9vgm1.

- For the purposes of this project, a **dataset** (also known as a DataArray or datacube) is a multi-dimensional table (it can be imagined as a table of tables, a list of lists, or a dictionary of dictionaries) that has the potential to store multitudes of information, both in a spatial and temporal context. Often used in global climate models, this data structure can hold temperature, precipitation, latitude, longitude, and time, at every cell or pixel in the dataset (**Figure 1**).
- **NDVI**, or normalized difference vegetation index, is calculated using the red and near-infrared bands and is a measure of the health and density of vegetation. If vegetation were to be cleared from an area to create a parking lot, we'd expect the NDVI values to drop to near zero (**Figure 2**).
- t_0 and t_1 represent the "before" and "after" images in the image pair used in training the model and when running inference on a trained model. More on the model's architecture is discussed in the next section.

METHODOLOGY

Conceptually, there are three categories of scripts that we developed for this project:

1. **Dataset tools** that build, inspect, and prepare the dataset for the model;
2. **Model tools** that define the architecture, training setup, and run the training and hyperparameter search; and
3. an **Inference tool** to produce results.

We will discuss the data collection and processing, model architecture, and several workflows for this project in the following manner:

1. Dataset creation
2. Dataset inspection
3. Pre-Training
4. Model Architecture
5. Training Architecture
6. Model Training Loop
7. Model Inference

DATA COLLECTION AND PROCESSING

The data source for this project is Sentinel-2 imagery. We are using Google Earth Engine's (GEE) Python API to download this data and the service offers free and paid tiers. The Sentinel-2 dataset is in the free tier and is at 10m resolution. Weather permitting, new images with under 5% cloud cover are added to a region a couple of times a week, so over the course of a year, a region can be expected to have around 100 time steps of imagery. The imagery spans 13 total bands, consisting of true color (red, blue, and green), near infrared, infrared, and ultra blue (good for coastal and aerosol observation). In addition, the SRTM dataset (global elevation at 30m resolution) is available for free download. We chose to train the model on imagery from Dallas, Texas, as a lot of home construction has taken place in the surrounding suburbs over the past five years, ensuring that we can easily find changes in the imagery.

For this project, we chose to gather the following data into feature channels in every dataset:

- RGB bands (i.e., true-color imagery)
  - These are fundamental channels needed to perform satellite image analysis.
- NDVI
  - As part of the dataset preparation process, we normalize the NDVI bands to the seasonal changes so that the model is sensitive only to major changes, like the clearing of a field for new construction.
- Terrain indicators (elevation, slope, and aspect)
  - These are of secondary importance but could inform the model about why certain areas are prone to construction and others are not. For example, most construction does not take place on steep slopes.
- Time-based indicators (change in time, i.e., Δt and day-of-year)
  - The model could, depending on location, learn that meaningful changes in the landscape might only occur in larger time differences, or that changes in the landscape (in our case, detecting construction) may typically occur in the warmer months of the year, for example.

MODEL ARCHITECTURE

The model uses an unsupervised learning approach to detect changes in low-resolution satellite imagery. At the heart of the model is a variational autoencoder (VAE), which was inspired by the RaVAEn project.[2] VAEs compress the input data into a representation so that the model can learn to generalize what the key patterns and features are, without the use of labelled training data. This is important in anomaly and change detection problems such as our model. Furthermore, we also borrow the time pair-based change detection from RaVAEn. You can view the model architecture in **Table 1**.

We also used the time-embedding methods from the DSFANet project.[3] Theirs was an older approach, but their use of time as a factor in their model training architecture is helpful for our needs.

The inputs to the model are geohash-based, where the sample area consists of a five-precision geohash (e.g., "9vgm2" or "dr5ru"), which, depending on its location on the earth's surface, is on the order of 5 to 10 square miles. This level of precision was chosen for its balance of area and processing speed.

Metrics and Loss Function

The output consists of the following metrics:

- **Reconstruction Loss** (using mean-squared error) is used to measure how faithfully the VAE reconstructs the t_1 image. Lower values indicate better reconstruction performance.
- **KL Divergence Loss** is used to balance the reconstruction fidelity with a smooth and structured latent space. The KL divergence loss is scaled with β in the first 10% of epochs to avoid a failure of the model to learn (also called a "posterior collapse").
- **Slow-Feature Penalty** allows for the filtering of noise so that the model can focus on persistent changes. It is scaled by the slow_weight hyperparameter (discussion of the parameters and hyperparameters for each tool can be found later in this section).
- **SlowLoss** is the mean-squared error between the encoder's mean maps for t_0 and t_1. This is useful to encourage the model's latent representation to change gradually over short

time gaps instead of getting affected by noise. It helps the model learn genuine, lasting change.

- **Total VAE loss** equals the reconstruction loss + β * KLLoss + slow_weight * SlowLoss

The model also computes per-patch reconstruction error and latent-delta norms during validation. Latent-delta norms measure how much the latent code changed between t_0 and t_1.

Other outputs are created from the inference tool, described below. The tool creates a heatmap and change mask, along with summary tables and imagery of t_0 and t_1, to demonstrate how the model performed in detecting change.

WORKFLOWS

Dataset Creation Workflow

The script for creating the datasets **build_dataset.py**. The key tools in the script include GEE, geemap, xarray, rasterio, rio-xarray, zarr, s3fs/fsspec, gdal, numpy, and Pandas. The basic unit of the model training and testing workflow are the 5-precision geohash-labelled datasets. The user runs build_dataset.py and selects the bucket destination, AWS and GEE account credentials, and a list of geohashes to either update existing datasets or add new datasets for the specified geohashes. Working with one geohash at a time, the tool uses the input geohash to create a bounding box. This bounding box is sent to GEE and the resulting imagery is downloaded. The imagery is stacked on top of each other, converted to a Zarr store (a cloud-optimized, chunked, and compressed file format), and uploaded to the specified S3 bucket. For our project, we created a new bucket, called rgc-zarr-store. A metadata CSV is also uploaded, which has a registry of all the images contained within the dataset. At the same time the imagery is downloaded, other assets are created, which will be relevant further downstream. These assets are climatology arrays to normalize for NDVI seasonality and a band summary statistics file.

The parameters for the script are as follows:

- -- bucket/-b: The name of the target S3 bucket (no s3:// prefix), e.g., rgc-zarr-store
- -- folder/-f: The top-level folder under the bucket, e.g., data
- -- ee-account-key/-K: The GEE service account email, e.g. low-res-sat-change-detection@low-res-sat-change-detection.iam.gserviceaccount.com
- -- aws-creds-file/-A: Path to AWS credentials JSON, e.g. secrets/aws_rgc-zarr-store.json
- -- geohashes/-g: Comma-separated 5-precision geohash(s), e.g. 9vgm0,9vgm1

There are also three arguments best left to their defaults:

- -- coud-perc/-c: Cloud percentage filter for imagery to select below this threshold; defaults to 5%
- -- scale/-s: Scale for imagery requested from Google Earth Engine, defaults to 10
- -- end-date/-e: Sets the last date of imagery to be collected by the script, defaults to the date when the script is executed

Dataset Inspection Workflow

The **inspect_dataset.py** script is tasked with peeking into the dataset. The key tools in the script include xarray, s3fs, matplotlib, numpy, and Pandas. Although not required for the

model, the user can run the inspect_dataset.py script by inputting a date pair (if not selected, the model will default to the first and last time-steps in the dataset). This tool gives summary statistics on the channels within the dataset and outputs PDF-ready images of the data.

The parameters of the script are as follows:
- -- bucket/-b: Same as in build_dataset.py
- -- folder/-f: Same as in build_dataset.py
- -- geohash/-g: Same as in build_dataset.py
- -- output/-o: Base path, with no extension, for the output folder that will store the .PNG and .PDF outputs of the script
- -- aws-creds/-a: Same as the aws-creds-file argument in build_dataset.py
- -- patch-size/-p: Sets the patch size of the visualization, which defaults to 128, the default patch size for the model training
- -- patch-location/-pl: Defaults to 0 (top left), being the first index of the patch within a single time slice and calculated using row * column ordering

There is one group of mutually-exclusive arguments, i.e., the user can only choose to include one of the following per execution of the script:
- -- index/-i: Inspects the patch at the global index value, then inspects the same patch location in the next time slice
- -- dates/-d: Inspects by two dates, with each written either in YYYY-MM-DD or the keywords 'first' or 'last' to retrieve the first or last time slice in the dataset

Pre-Training Workflow
The **prepare_dataset.py** script is written to convert the Zarr file into a workable format that PyTorch can use. The key tools include xarray, s3fs/fsspec, tempfile, PyTorch utilities, numpy, and Pandas. Once the training, validation, and testing sets are assigned, the prepare_dataset.py tool, mainly through the **ZarrChangePairDataset** class, stages the datasets locally to speed up I/O and reduce S3 data egress costs. It loads the band statistics and climatology arrays and normalizes all the channels in each dataset according to a global minimum and maximum value from all the input datasets, so that each band now has a value between 0 and 1. Once ready, the tool divides the input image (roughly 500x500 pixels) into a patch (128x128) and feeds it to the next step downstream. This patch size was chosen to balance the need to incorporate the surrounding image context while maintaining an acceptable model training speed. The **build_dataset.py** and **prepare_dataset.py** scripts create a multi-dimensional dataset that provides the model cues beyond just visual imagery to help it detect change.

The script has no user-definable arguments as it merely defines modules imported by other scripts and functions.

Model Architecture
The script that gives the model its structure is **vae.py**. The key tool is PyTorch. The model is designed to generate pixel-wise change-detection heatmaps and change masks at a resolution equal to the input, so to do this, we use a fully convolutional approach to allow for a reconstruction at the same 128x128 resolution as the input. This allows the inference script to

run no upsampling of the reconstruction, which gives our heatmaps and change masks a higher resolution. A key feature is a "latent spatial grid", which is a two-dimensional, 16x16 lattice of variables, each cell having its own distribution, which allows the model to output a reconstruction with meaningful spatial information. Walking through each module in the script, we start with the **SEBlock**. This is a so-called "squeeze and excitation" module, which compresses the spatial dimensions of each channel into a single number. The SEBlock also adds a multi-layer perceptron to allow the model to learn which are the most informative spectral bands and suppress the less useful ones. **FCVAEEncoder** is a module that halves the spatial resolution three times when creating the latent grid, to convert the 128x128 patch down to 16x16 (128 to 64 to 32 to 16). ReLU (rectified linear unit, where negatives are set to zero and positives are kept) is employed to introduce non-linearity. We add skip-connections to introduce fine detail that was lost in the down-sampling. We stop at 16x16 so that every patch of the input has a corresponding latent value, which is critical for pixel-wise change detection. There is no flattening so that the decoder can upsample seamlessly. **FCVAEDecoder** is the FCVAEEncoder in reverse, with normalization, ReLU, skip-connections. The "down-then-up" architecture with skip-connections is considered a U-Net model structure. The **FCVAE** class wraps all the modules together and computes the loss. The FCVAE class never does per-band differencing (i.e., it does not compare the Red band of t_0 to the Red band of t_1); rather, it encodes the entire multi-channel tensor for t_0 and t_1 each into a latent grid, then computes a single L2 norm across the latent-channel dimension. It helps the model learn which combinations of band changes are most predictive of real change.

There are no user-definable arguments associated with this script as it defines modules imported by other scripts.

Training Architecture

The script involved in defining the training and validation of the model is **trainer.py**, and the key tools are PyTorch and tensorboard. **train_one_epoch** is the module that oversees training and uses the Adam optimizer. **validate_one_epoch** oversees validation and sends model training information to tensorboard, a PyTorch model visualization tool that helps the user understand the training performance during the training loop. Using tensorboard is optional and does not affect the model training behavior. **_rebuild_train_loader** and **_update_sample_weights** are a pair of tools that track which samples are getting reconstructed poorly as a proxy for labels. The assumption is that patches that are "hard" to reconstruct by the model are those that have undergone change. In future epochs, these patches are sampled more if the **hard_power** hyperparameter is above 1.0. **train** is the module that orchestrates the above training tools. It also adds early stopping functionality to prevent undue training if the loss does not improve.

This script only defines functions and modules used by other scripts, so there are no user-definable arguments associated with it.

Model Training

train_vae.py and **hp_search.py** are the scripts in charge of executing the training of the model. The key tools are PyTorch, s3fs, tempfile, optuna, smtplib as well as the tools defined by the above scripts. The **train_vae.py** script allows the user to initiate a single training run. **hp_search.py** defines the functions needed to run a hyperparameter search to explore the

parameter space to train the best possible model. **seed_all** sets a random seed for reproducibility and defaults to 42. The rest of the tools define the training, validation, and evaluation steps of the trainer. The hyperparameters are defined by the user or left as defaults and are incorporated into the training loop. **hp_search.py** is a hyperparameter search and tuning orchestration script that runs on top of the **train_vae.py** tools so that multiple training runs can be organized in sequence as the function explores the hyperparameter space. When the search is completed, the script sends an email notification. This allows the user to work on other tasks while the system is running. Optuna is the tool used to orchestrate that search process and uses a Tree-structured Parzen Estimator, which is a Bayesian optimizer. As the trials are running, the interim progress is saved to a database, and a training log is kept. The database allows the tool to pick up where it left off, should an interruption occur.

The parameters of **train_vae.py** are as follows:

- -- seed/-s: Sets the random seed for reproducibility and defaults to 42
- -- bucket/-b: Same parameter as defined in the above scripts
- -- folder/-f: Same parameter as defined in the above scripts
- -- train-geohashes/-tr: A list of comma-separated 5-digit geohash codes to train on, e.g. 9vgm0,9vgm1,9vgm2
- -- val-geohashes/-va: A list of comma-separated 5-digit geohash codes to validate the model on
- -- test-geohashes/-te: A list of comma-separated 5-digit geohash codes for testing the model, e.g. 9vgm3,9vgm4
- -- out/-o: Sets the save directory for the checkpoints and outputs of the model
- -- latent-dim/-l: Sets the dimensionality of the VAE latent space, defaults to 64
- -- aws-creds/-A: Same as the aws-creds arguments in previous scripts defined above
- -- patch-size: Sets the size, in pixels, of each square patch and defaults to 128
- -- patch-stride: The stride, in pixels, between patch windows, and if left unset, equals the patch-size argument. This means that all pixels are seen by the model
- -- num-workers: Sets the number of multiprocessing workers operating in parallel to speed up training. Note that this currently does not work if you run the script on an EC2 instance
- -- stage-zarr-root: Do not use. This defines a path to a directory containing pre-downloaded Zarr stores, one per geohash. Overrides —stage-zarr
- -- stage-zarr: Recommended to include. If set, the script will download each Zarr locally before training
- -- no-stage-zarr: Not recommended to use. If set, the script will not download Zarrs locally and will instead stream from S3
- -- hard-power: The exponent for hard-negative mining as a proxy for labels. If hard-power is set to 1.0 (the default), the sampling is uniform. Any value greater than 1.0 will reweight patches by reconstruction error * hard_power
- -- max-step: Maximum temporal gap (in number of time steps) to sample. If max_step is 1, only the adjacent pairs (i, i+1) pairs are used. If this argument is greater than 1, all (i, i+k) pairs with $1 \le k \le$ max_step are included. Defaults to 1, though is recommended to include a much larger number, like 60 or 180, to show the model time gaps of multiple months, to ensure the model has seen pairs that actually underwent change

- -- time-gap-exp: Sets the exponent that reflects how much to weigh the larger time-gapped (k) samples in (i, i+k) and defaults to 1.0
- -- num-time-pairs: Sets the fraction of all valid per-geohash pairs (i, i+k) to sample in each epoch. We recommend using small fractions, like the default of 0.05, to avoid huge numbers of training patches
- -- epochs: Defaults to 50 individual training runs
- -- batch-size: Sets the number of samples passing through the training loop at one time and defaults to 16
- -- lr: Sets the learning rate and defaults to $1x10^{-3}$
- -- weight-decay: Sets the optimizer weight decay and defaults to $1x10^{-5}$
- -- scheduler-step: Sets the learning rate scheduler step size and defaults to 10
- -- scheduler-gamma/-e: Sets the learning rate scheduler decay factor and defaults to 0.5
- -- slow-weight: Weight on the time-aware loss term and defaults to 0.1
- -- early-stopping: Sets the patience for early stopping of the training if no improvements are observed and defaults to 5 epochs

The parameters of **hp_search.py** are as follows:
- -- trials: Sets the number of Optuna runs with unique hyperparameters to train the model on
- -- bucket/-b: Same parameter as in other scripts
- -- folder/-f: Same parameter as in other scripts
- -- train-geohashes/-tr: Same parameter as in other scripts
- -- val-geohashes/-va: Same parameter as in other scripts
- -- test-geohashes/-te: Same parameter as in other scripts
- -- out/-o: Sets the save directory to place all assets from this script (i.e., training log, optuna log, tensorboard materials, hp_search database, and best_model.pt)
- -- aws-creds/-A: Uses the default from **train_vae.py** if not set here
- -- batch-size: Uses the default from **train_vae.py** if not set here
- -- patch-size: Uses the default from **train_vae.py** if not set here
- -- num-workers: Uses the default from **train_vae.py** if not set here
- -- stage-zarr: Recommended to use this argument flag, same as from other scripts
- -- no-stage-zarr: Not recommended to use this argument flag, same as from other scripts
- -- trial-epochs: Sets the number of epochs per trial and defaults to 5. Recommended to leave this number low when beginning a new hyperparameter search to remain nimble and search more of the hyperparameter space
- -- trial-patience: Sets the early-stopping patience per trial and defaults to 3. The lower the number, the more chances Optuna will have to explore the hyperparameter space

Model Inference

The script in charge of testing a trained model on new datasets is **inference.py**. The key tools are PyTorch, xarray, matplotlib, numpy, and Pandas. Once a model has been trained, the user directs the inference.py script to open the model and run it over a specified dataset and date range. The tool loads the climatology layers and the data, builds patches, and applies the trained model to the patched dataset. The script outputs a PDF report with figures showing

the two timesteps in the time pair, a heatmap, a change mask (with a tunable threshold), and summary statistics.

The parameters of the script are as follows:
- -- bucket/-b: Same as in previous scripts
- -- folder/-f: Same as in previous scripts
- -- geohash/-g: Same as in previous scripts
- -- model/-m: The path to the best_model.pt produced from the training or hyperparameter search scripts
- -- aws-creds-file/-A: Same as in previous scripts
- -- out/-o: Sets the path to where the outputs of this script will be saved
- -- stage-zarr: Same as in previous scripts
- -- no-stage-zarr: Same as in previous scripts
- -- patch-size: Same as in previous scripts
- -- stride: Same as in previous scripts
- -- out-size: Sets the number of patches per dimension in the output heatmap. Defaults to None
- -- batch-size: Same as in previous scripts
- -- latent-dim: Must match the latent-dim used at train time
- -- threshold: Sets the threshold value used when constructing the change mask. If unset, it will default to the mean + 2 * std of heatmap
- -- num-workers: Same as in previous scripts

The following are part of a mutually exclusive group, with definitions and defaults equal to **inspect_dataset.py**:
- -- index/-i
- -- dates/-d

Model Architecture Visualization
This small script, called **visualize_nn.py**, uses torchsummary and torchviz to help visualize the model's architecture.

A Note on Streamlit
To demo the project, we created a Streamlit app that included presentation slides on one side of the screen and an interactive demo on the other. The demo allows the user to input a 5-character geohash, select a date pair, view the resulting bounding box on a map, and then buttons to run **build_dataset.py**, **inspect_dataset.py**, and **inference.py** scripts in sequence. The datasets created are stored and retrieved on S3, just like when building datasets that train the model. Prior to the demo, the best model was copied and saved locally in the project's Streamlit folder on the repository. A Progress bar and "spinner" showed the user that the process at each step of the demo was working. Upon the completion of each step, the user gets a view of the assets created and buttons appear, which allows the user to download all the assets.

One Pixel's Journey through the Model

The Methodology section above gives a deep-dive explanation into how the model and training procedures were architected for this project. I find it helpful to take a step back and explain how the model runs by following a pixel's journey through the model. Let's choose the very top-left pixel on time t_0 at a geohash-dataset that is about ready to be run through the model for training.

We start at the Zarr store, on the S3 bucket. There, the pixel is stored in the geohash's Zarr folder, which are chunked for faster access. The Zarr dataset holds a 4D array for each band: band x time x lat x lon. At the start of the loop, the Zarr with the pixel is downloaded to the local machine.

When ZarrChangePairDataset.__getitem__ is called, for a specific index, we pick a time-pair (i, j) from those sampled. We pick a spatial window, extract a patch for each raw band (R, G, B, NDVI, elevation, slope, aspect, and aspect_mask), giving us arrays of shape (8, patch_size, patch_size) for t_0 and t_1. ZarrChangePairDataset also computes the sin and cosine of aspect and stacks the metadata channels (Δt, day-of-year, seasonal NDVI, z-NDVI) to yield two arrays each of shape (13, patch_size, patch_size). Finally, it returns a tuple of t_0 and t_1. Our pixel is in the t_0 tensor.

For the DataLoader step, these patches are batched into tensors of shape (B, 2, 13, patch_size, patch_size) plus a (B,1) delta-t tensor.

After this, the trainer splits the tensors into x_{t0} and x_{t1}, each with the shape (B, 13, patch_size, patch_size). The trainer calls mu_{t0} = encoder(x_{t0}), then recon, mu_{t1}, $logvar_{t1}$ = model(x_{t1}, Δt). From here, the pixel is sent to be compressed by the model, ceasing to exist, but contributing its information to teaching the model to observe change detection.

RESULTS

This model trained for around 24 hours on imagery from the greater Dallas, Texas area. We did not explore how the model performed when tested on imagery that was not from suburban Texas regions. We also did not test the model on geohashes with few changes, or those with small time gaps in between t_0 and t_1. The best parameters of the model are listed in **Table 2**. Viewing the results, we can see that the heatmaps and change masks are able to identify areas that underwent change. Refer to **Figure 3** to see the results of an inference run for geohash 9v1z2, which is from the San Antonio, Texas area.

Since the model breaks the input into 128x128 patches, the heatmap and change masks show these as artifacts, especially when viewing the full-scene heatmap: at the boundaries of the patches, some patches' masks reach the threshold while other neighboring patches do not, so we see sharp breaks in the heatmap and change mask.

The best trial value is high, at nine figures, but that is misleading, as we architected the metrics to report the sum error (i.e., the sum of all the reconstruction errors for every pixel). Understandably, even for very small per-pixel errors, this number can get very large. The summary statistics in Figure 3 show that the mean change in error (Δμ) is very small, which shows that per-pixel, the model does a good job at reconstructing the change.

DISCUSSION

In order to give the model the best chance of success in detecting change, we allowed Optuna to bias our sampling strategy towards harder samples and those with larger time gaps. Would a sample with very little change still get observed by the model? We are not sure.

POTENTIAL FUTURE WORK

If this project were to be developed further, there are a myriad of directions that we could go in. We could wire in the ability to select larger areas when building datasets and testing the model. This would be more beneficial for the team so that instead of testing an area seven square miles at a time, they could put in their entire proposed capture area and receive a much more helpful heatmap and change mask.

The change mask asset itself could use more improvement: we could run a clustering process on the mask layer so that we get very user-friendly bounding boxes that could be sent directly to the image provider (instead of blobs of changed areas). We could ask it to bound an area with at least 90% change detected, as an option.

We could move beyond pure change detection towards enhanced segmentation and intelligence: we could explore asking questions like, what *kind* of change is occurring? Is it construction? What kind of construction: residential, commercial, or industrial? This may require the use of other data sources, like a roads layer from OpenStreetMap or hyperspectral imagery (HSI).

Using a Kernel Density Estimation calculation derived from OpenStreetMap (to allow the vector points of a road network to become a raster surface), and given a sufficient training area and compute power, we could consider adding the road network to the dataset. The model might be able to understand that where there are roads, construction can follow. In the middle of a forest, without any roads, construction would very likely not occur. To state the obvious, cement trucks, bulldozers, work crews, etc. need to be able to use the road network to access a construction site. The weakness of utilizing this data source is that the road network does not change frequently. We would need to account for this by expanding the training area so that more road changes are observed more often by the model during training.

HSI has the potential to detect specific mineral content, plant species, or even plant subspecies type. Regarding mineral content in HSI, we may be able to figure out which pixels are asphalt, what are roof tiles, etc. This could improve the model's performance, as each component of the landscape could potentially have a different spectral signature that could be picked up by the satellite image sensor, and thus have an easier time being segmented by the model.

If we purchased higher-resolution imagery, or incorporated imagery that other teams have used, the model's performance would improve dramatically and would open even more doors for the project. Asking what kind of change would be easier with higher-resolution imagery, though this would come at a cost in compute time.

Finally, we could explore improving the model's architecture and incorporating pre-trained models and workflows from others. The world of data science is evolving rapidly, and the earth observation (EO) industry is no different. New satellites for EO are being launched practically every week, increasing the amount of data collected about our planet. Machine learning models and AI are advancing seemingly every day, which makes our job of

understanding the surface of the earth easier. Combining these two shows that there is a lot of room for growth and improvement of the model.

SUCCESS CRITERIA REVISITED

Let's circle back to our success criteria and discuss whether the project fulfilled the three goals: running cheaply, with easily-obtainable imagery, and having the potential to fit into other workflows.

Run Cheaply

This project was run extremely cheaply. There were very minor hosting costs with storing and accessing our data on the AWS S3 bucket: just a couple of dollars per month. I do not know the cost of running the training on the EC2 instance, but with a total running time of about 24 hours on a single process, we can assume that the processing was at least contained and did not use all threads and the full power of the GPU. All the packages we used were free and open source.

Easily-Obtainable Imagery

We only used the free imagery from GEE, and all it takes to access is a free account with GEE and keeping the account credentials in a safe place. Once the code is up and running without errors, we can put in any 5-character geohash and obtain the imagery from that location. By design, we wrote the scripts to make it as easy as possible for people to obtain the target imagery.

Potential to Fit into Other Workflows

We chose to use geohashes as the fundamental unit in organizing the datasets, as other existing workflows that the SOC team runs use that particular geospatial index. The Streamlit app could be run when the team wants to check whether a particular area did experience change over a specific time window. That being said, the areas are small, on the order of 7 or 8 square miles.

Given the above, we can conclude that this project was a success.

APPENDIX

Figure 1. Schematic of a typical DataArray, or datacube, used in global climate models showing how temperature, precipitation, latitude and longitude are registered to each point in the dataset. Time is also encoded, one time step for each layer. This project takes advantage of the information density and organized nature of datacubes for creating the data processed by the model in this project. Image taken from https://xarray.dev/xarray-datastructure.png.

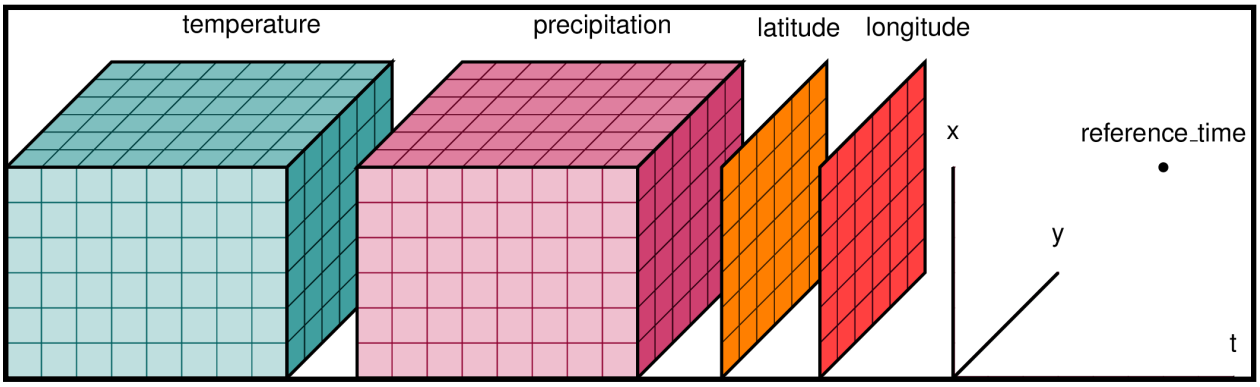


Figure 2. Example colorized NDVI image, with greens being higher NDVI values and reds being negative NDVI values. Yellow represents an NDVI of zero. Note how in the change image on the right, there are negative NDVI values are where what looks like new construction has taken place. Image taken from this project's work.

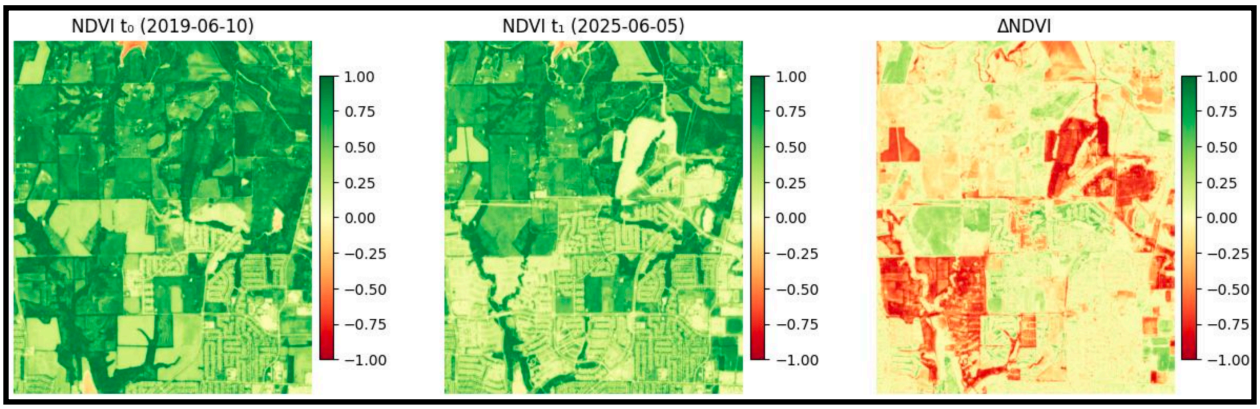


Table 1. The model architecture from torchsummary.

Layer (type)	Output Shape [batch_size*, channels, dimension_x, dimension_y]	Param #
Conv2d	[-1*, 32, 64, 64]	3776
BatchNorm2d	[-1, 32, 64, 64]	64
ReLU	[-1, 32, 64, 64]	0
AdaptiveAvgPool2d	[-1, 32, 1, 1]	0
Conv2d	[-1, 4, 1, 1]	132
ReLU	[-1, 4, 1, 1]	0
Conv2d	[-1, 32, 1, 1]	160
Sigmoid	[-1, 32, 1, 1]	0
SEBlock	[-1, 32, 64, 64]	0
Conv2d	[-1, 64, 32, 32]	18496
BatchNorm2d	[-1, 64, 32, 32]	128
ReLU	[-1, 64, 32, 32]	0
Conv2d	[-1, 128, 16, 16]	73856
BatchNorm2d	[-1, 128, 16, 16]	256
ReLU	[-1, 128, 16, 16]	0
Conv2d	[-1, 18, 16, 16]	2322

Notes:
*: The “-1” in the batch_size indicates that it can be of any size

Total params: 99,190
Trainable params: 99,190
Non-trainable params: 0

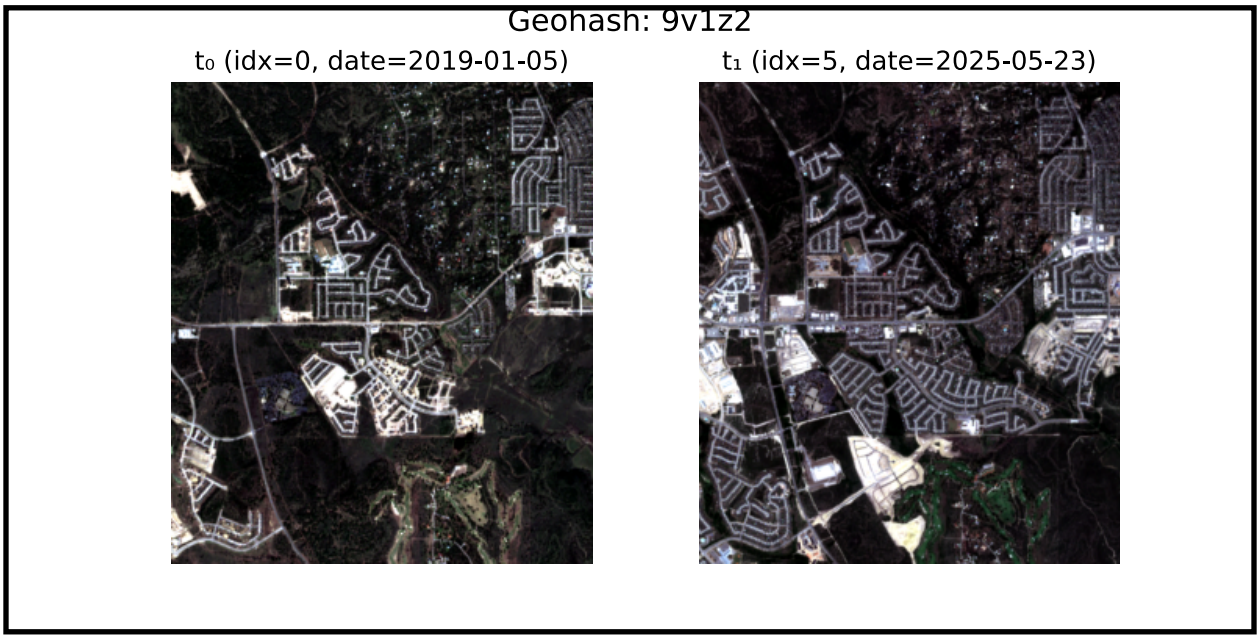
Input size (MB): 0.81
Forward/backward pass size (MB): 6.29
Params size (MB): 0.38
Estimated Total Size (MB): 7.48

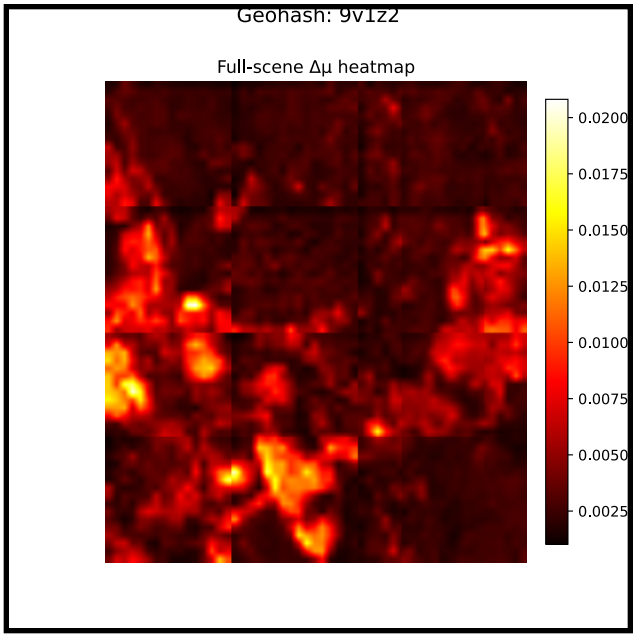
Table 2. The hyperparameter search arguments and best parameters found in the model.

Argument Name	Value
trials	30
bucket	rgc-zarr-store
folder	data
train_geohashes	9vgke,9vgks,9vgm0,9vgmd
val_geohashes	9vgm1
test_geohashes	9vgm6
out	results/hp_1
aws_creds	secrets/aws_rgc-zarr-store.json
batch_size	32
patch_size	128
num_workers	0
stage_zarr	True
no_stage_zarr	False
trial_epochs	30
trial_patience	5
Best parameters from hp_search	
lr	0.00014987952321776128
weight_decay	$6.490906952185921 \times 10^{-6}$
latent_dim	17
slow_weight	0.3737798946568667
scheduler_gamma	0.10182361080944241
hard_power	3.64675894131236
max_step	9
num_time_pairs	0.9070241116025707
time_gap_exp	1.9140789185236153
Best trial value*	264084863.0125

Notes:
*: The best trial value is the sum of all the reconstruction errors for every pixel, hence the very large number.

Figure 3. Inference outputs from running geohash 9v1z2 through the best model created during the project. There is a side-by-side image in true-color, a full-scene heatmap and change mask, overlays of the heatmap and change mask over t_0 and t_1, and a summary statistics table showing the error, threshold, and overall size of the input.





Geohash: 9v1z2

Δμ heatmap over t_0 (2019-01-05)

Change mask over t_1 2025-05-23

Change Detection Summary	
Metric	**Value**
Mean Δμ	0.004
Standard deviation Δμ	0.002
Max Δμ	0.021
Threshold	0.009
Percent patches above threshold	13.38%
Patch grid size	489 x 428

ENDNOTES

[1] https://www.geohash.es/

[2] https://github.com/spaceml-org/RaVAEn

[3] https://github.com/wwdAlger/DSFANet