Inference Software Guide for: Planetary-scale Surface Feature Detection and Mapping for Future Exploration Missions

Mark Wronkiewicz
January 18, 2021

1 Overview

This guide provides an overview to the inference software for the NASA STTR project entitled: "Planetary-scale Surface Feature Detection and Mapping for Future Exploration Missions". Specifically, this covers the architecture for detecting, mapping, and storing craters using for large satellite image data sets. This guide assumes access to a trained machine learning model capable of making high quality predictions on satellite images. It requires the user be reasonably familiar with using a variety of cloud resources on Google Cloud Platform (though analogs exist on Azure and AWS), Docker, and Kubernetes. It is meant to complement the final report with some extra technical details for users of our inference software.

2 Processing pipeline

The following section outlines the step-by-step processing needed to execute this inference pipeline at scale. See 2.1 for a graphical illustration. Here, we process and store results from each satellite image strip independently detecting and storing the craters for each. While the pipeline is meant to process global data sets, we do not store the prediction results in one single map. There are small but significant alignment discrepancies for the image sets we analyzed making global image alignment an extremely difficult task outside the scope of this work.

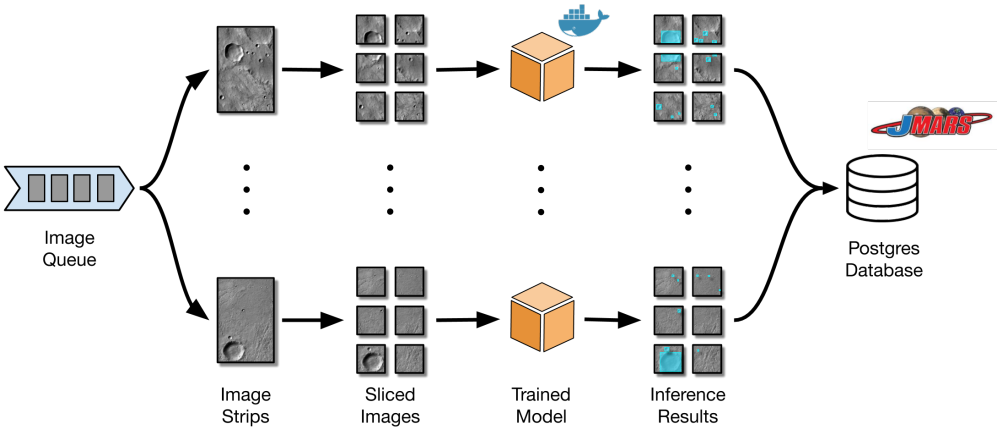


Figure 2.1: Illustration of all steps in the inference process.

2.1 Queue creation and population

The inference pipeline must be able to batch process arbitrarily large image sets. Even in the face of thousands to millions of satellite images to predict, it must remain as fault tolerant as possible to any problems or errors that arise during long running prediction jobs. Therefore, a reasonable strategy is to use a cloud-based processing pipeline meant to accommodate large scale processing. This begins with a queue where each item in the queue represents a unit of work for the inference pipeline. Queues store each prediction request as a "message" where each message in our pipeline contains simple information like the URL to a satellite image, projection information, and other metadata. Using a queue allows us to decouple the set of images that need to be processed from the software and compute that actually carries out the processing of those images. This gives us flexibility to update or scale up/down the cloud resources carrying out inference. We can also add new prediction requests to the queue on the fly.

We specifically use Google's PubSub API, which is the Google Cloud message queue tool. Within PubSub, we define a "Topic," which represents a group of messages. The create_messages.py script is capable of publishing messages to a named Topic to keep track of all inference requests. Once we initialize the compute resources to download and run inference over images, we create a "Subscription" to our Topic. With this Subscription, our inference code can pull messages one at a time, process them, and instruct Pub/Sub to delete them from the queue. Pub/Sub will only remove messages from the queue if they are acknowledged to have been completed successfully (see 2.2). This provides fault tolerance because we don't need to repopulate the queue and restart processing if an error occurs somewhere during the inference process.

2.1.1 Image acquisition and desired image properties

A comprehensive list of available images was obtained from Scott Dickenshied at ASU. This include the URL and all available metadata associated with MRO and LROC imagegs. The

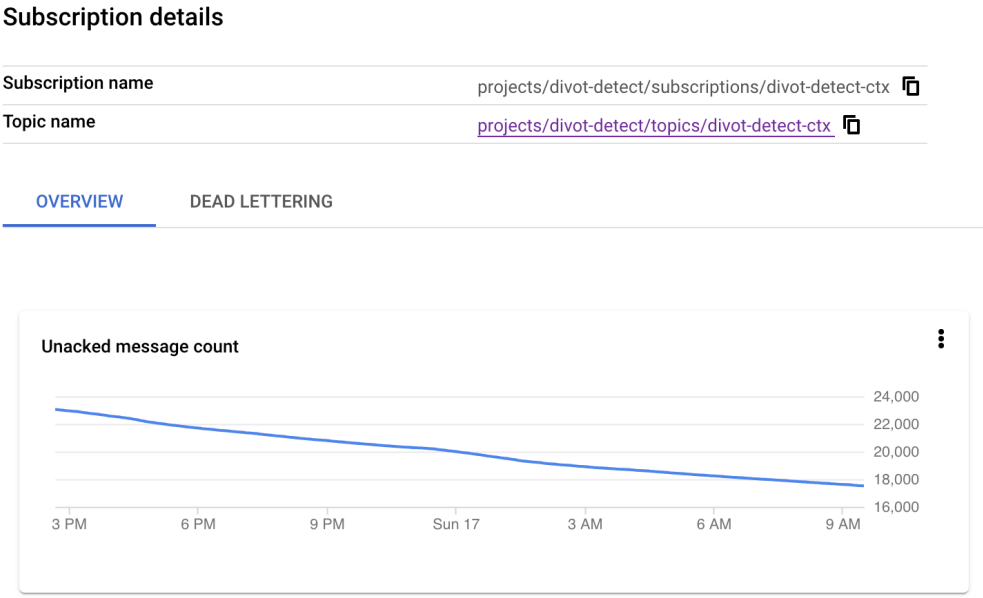


Figure 2.2: PubSub queue showing remaining messages (here, CTX images) that must be processed.

same information can also be obtained at `http://viewer.mars.asu.edu/` or `http://wms.lroc.asu.edu/lroc/search`. This information formed the basis of each item in the queue mentioned in 2.1.

Note that the full image lists from LRO or MRO are quite extensive (hundreds of thousands to millions of satellite images) and will take a significant amount of time to process. Therefore, it's useful to use some of the filters built into the message creation scripts to start small. After the inference pipeline is working as expected, it is easy to scale up by modifying a couple parameters in the kubernetes configuration files.

2.2 Database initialization

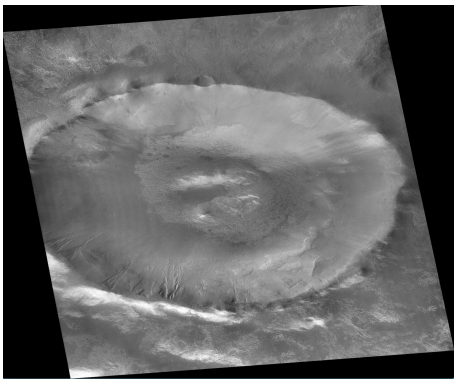
The inference pipeline is capable of generating millions of features. To store this efficiently, we use a PostgreSQL database with the PostGIS plugin. Using an SQL database permits manipulating data at the scales we're interested in and thhe PostGIS plugin allows us to manipulate geospatial data types. We chose to use a managed database sollution thhrough Google's Cloud Platform (the "SQL" API) for simplicity. The inference pipeline takes a URI input to connect to the database server and uses SQL Alchemy to manage connections and upload data. We currently use two tables within the database: Images and Craters. Images refer to individual satellite image strips (e.g., NAC or CTX images) and contain metadata about the location and PDS ID of said images. The Craters table holds individual crater boundaries as well as a reference to the image in which they were detected.

We used Google's SQL API to create a managed SQL instance where we could store the

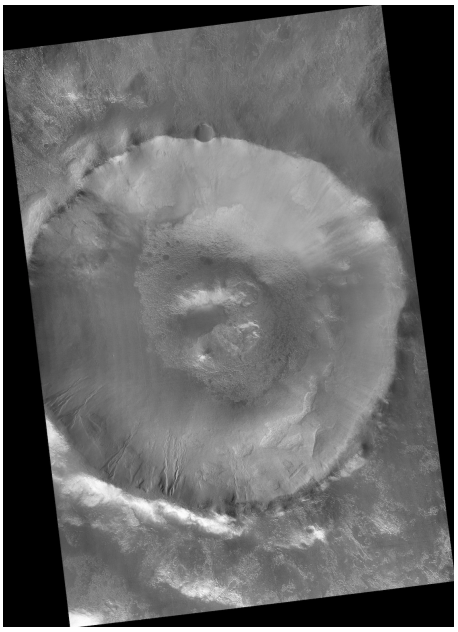
crater predictions. A relatively small and cheap SQL database instance should suffice here because each crater represents a relatively small amount of information. Even at 50 workers all simultaneously processing and uploading crater predictions, the smallest DB instance Google's SQL API had to offer was able to handle all of the traffic from our inference pipeline. For users with DB experience, it should also be relatively easy to run the database server locally as long as it is publicly accessible by the prediction pipeline.

2.2.1 (Optional) Projection

Our pipeline has the capability to reproject a satellite image using GDAL if necessary. This becomes increasingly necessary for images that are acquired at extreme latitudes. These images generally appear distorted or "squished" along one axis making craters appear like ovals instead of circles. If projection information is provided in the queue message (in the form of a center latitude), our software can apply an equidistant cylindrical projection to remove this distortion. This option must be specified for each message in the queue.



(a) Distorted image.



(b) Reprojected image

Figure 2.3: Example crater showing before (a) and after (b) reprojection using an equidistant cylinder projection centered on the image's geospatial coordinates.

2.2.2 Generate sliding windows

To process large images with our inference pipeline, we need to break these images into smaller sub-images that can be processed by our trained model. This is due to the fact that convolutional neural networks typically cannot handle arbitrary image sizes. Lunar and

Martian satellite images come in a variety of shapes and sizes though they are often thousands of pixels in each dimension. Therefore, we split the images up into approximately 1024x1024 pixel windows prior to inference. We also run the inference step at multiple scales (e.g., creating 2048x2048 windows and rescaling each dimension by a factor of 2). This helps the model capture craters at multiple scales because they might be missed using only one static resolution. After the inference step on each window, we reassemble the predictions back to the original image size.

During the message creation step, users can specify which specific scales to process the image at as well as the amount of overlap desired. Using a few different scales that differ by a factor of two is reasonable (e.g., [1.0, 0.5, 0.25] would process the image at (1) original resolution (2) half-scale and (3) quarter scale). We recommend testing a set of scales before proceeding to full-scale inference to make sure the returned crater predictions look as expected.

2.2.3 Feed sub-images to pool of inference models

Each pixel block is then sent to our inference pool for prediction. The pool consists of one or more trained ML models on GPU-accelerated machines sitting behind a load balancer. The load balancer accepts incoming images for prediction and distributes this traffic evenly to running ML models. This ensures that all ML trained models are fully utilized during the inference process. We can add or remove machines to or from this prediction cluster depending on how quickly we wish to process the images in the queue.

There are a few practical tips that we'll discuss here regarding the inference pool, which is specified as part of a Kubernetes deployment within the 'kfjobs' directory of the GitHub repository. Generally, a few dozen GPUs should be initiated as part of the inference pool if doing large scale processing. Otherwise, you may find that the ability to process large datasets is too slow. It's also worth monitoring the health checks and resource utilization throughout the processing to make sure the GPUs are fully utilized (see 2.4 and 2.5, which are also discussed in the next section). Finally, we suggest obtaining a static IP address for the load balancer and then specify that in the 'Service' section of this Kubernetes deployment. That way, it's not necessary to re-specify the inference endpoint (where the prediction requests are sent) in the PubSub messages every time the GPU pool is initialized. Finally, we recommend using some sort of tailing tool (like Stern) to monitor the logs of all your Kubernetes pods. That makes it much easier to quickly identify and debug any problems with the GPU images.

2.3 Process returned predictions

For each prediction request, a trained model will use the pixels as input and generate a set of crater predictions. Each prediction consists of a crater mask, a confidence value per crater, crater bounding boxes, and several other forms of metadata all in the form of a json dictionary. From this information, we first discard any craters with confidence values below a set threshold (e.g., 0.5). Then, we convert all crater masks from a dense image array into polygons. This involves using a contour detection algorithm to find the bounds of the crater, conversion of those points to a GDAL polygon composed of latitude and longitude points, and then polygon simplification (to reduce the size on disk).

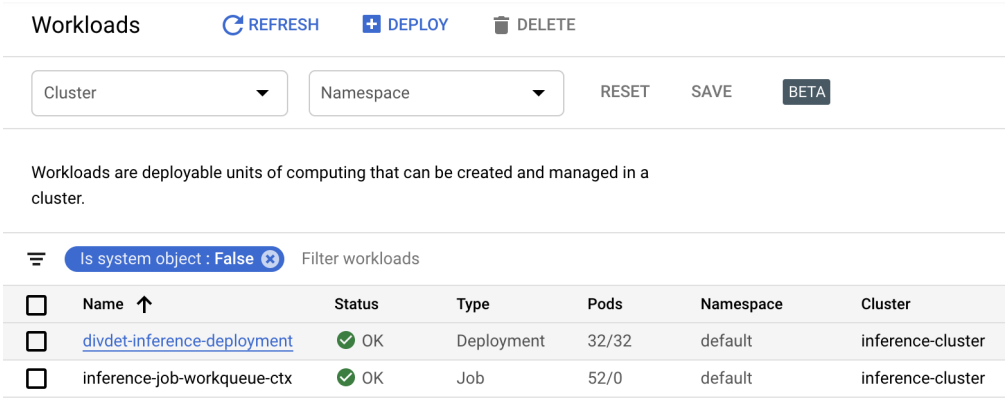


Figure 2.4: High level health monitoring of the cluster. This shows that both the GPU node pool, which responds to ML prediction requests, and the worker pool, which downloads and processes CTX images, are both healthy. From this page, we can also alter the number of GPUs and workers to better utilize our resources.

We can monitor the health of the Kubernetes cluster using metrics displayed by Google Kubernetes Engine. This is important to make sure the cluster is healthy and also operating efficiently given the compute resources available (see 2.4 and 2.5). This optimization is important because we don't want to pay for compute resources that we're not using.

2.4 Remove duplicate predictions

Most object detection models will produce repeated detections of some objects within an image. Therefore, we need to remove these repeats to avoid double counting any craters. Similar to most other object detection problems, we accomplish this through non-max suppression. In general, non-max suppression (NMS) involves finding pairs of overlapping polygons with an overlap exceeding some set Intersection Over Union (IOU) and discarding the prediction with the lower confidence. Computing all pairwise overlap is very computationally demanding – here, $O(n^2)$. The situation is made worse here because we must look for these overlaps across very large satellite image strips, which often contain thousands of predictions. We implemented a multithreaded version of polygon non-max suppression to handle large numbers of crater predictions, but this wasn't fast enough to support the processing speeds we desired. Therefore, we used Tensorflow's built-in (and optimized) NMS algorithm (specifically 'tensorflow.image.non_max_suppression') using only the bounding boxes as input.

2.5 Upload to database

Finally, we push all remaining crater detections to a database. Specifically, we use a Postgres database with the PostGIS plugin. The database allows us to store, manage, and query very large data sets and the PostGIS plugin provides geospatial functionality. Once the data is stored in a database, it's in a format that other many other geospatial platforms (including JMARS)

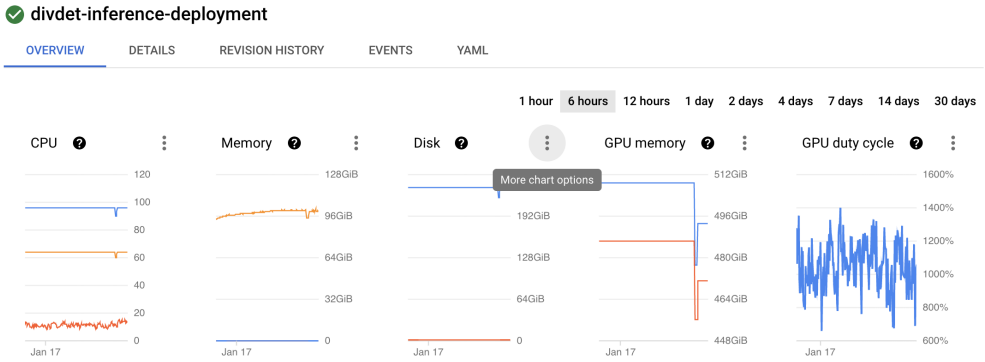


Figure 2.5: Compute resources being used by the GPU node pool. Here, we can look to make sure the GPU duty cycle and memory stays consistently high. This indicates we are using the GPUs to their full capacity.

can ingest and use.

The database schema consisted of two tables: Images and Craters. The Images table contains information about a single satellite image like the URL, instrument, latlon coordinates, and some other metadata. It also contains a database relationship to the Craters table. All craters are stored in Well-known Binary (WKB) format and a few pieces of metadata like the eccentricity and ML model's confidence in that prediction. We included this information so downstream users can build queries that include extra selection criteria.

3 Summary

In conclusion, this section provides some considerations when choosing whether or not to pursue a Kubernetes-based cloud pipeline. First, this architecture does require some up front time investment. Successful use requires that personnel have some experience with Kubernetes, Docker, and Cloud platforms (like GCP or AWS). Groups that are more research focused or consist mostly of students may find it difficult to successfully grasp and utilize all the techonlogies in required to take the approach described here. Second, cloud platforms also require significant financial investment. In our tests, we were forced to down-select the number of images to process in order to keep within our budget.

Use cases * want rapid processing of large image sets * have budget to support cloud computing * have personnel familiar with or patient enough to become familiar with cloud technologies like Kubernetes, Docker, and Cloud APIs * Requires some maintenance