Flood Extent Extraction

This notebook is designed to be an end to end solution that is able to extract flood extents from supplied imagery. Alot of the techniques and files regarding Unet for this NoteBook were obtained from this git repository. https://github.com/bnsreenu/python_for_microscopists

All configurations are contained within the `Notebook settings` cell. There are no other configurable options outside of the Notebook settings cell.

NoteBook settings

All user configurable elements are contained in this cell.

- Input Image settings
  - `nrg_image_s3_source` - String: Amazon S3 source directory containing flood images destined for processing eg. s3://ss-csu-dataset/raw/Brewarrina_Flood_2021_04_15cm_NRG/
  - `nrg_image_storage_directory` - String: Storage directory where flood images will be stored for processing.
  - `image_scale` - Int: Image size to scale image to during preprocessing. `patch_size` needs to evenly divide into this (eg and image 1024x1024 will get 4 patches at 512x512). Default value is: 1024
  - `patch_size` - Int: Image size that the Unet model is expecting. After image is scaled to `image_scale`. Preprocessing will patchify images into `patch_size`.
  - `ignore_nonsquare_images` Boolean: Notebook will ignore and not process non square images. Otherwise non square images will be rescaled to `image_scale`. Scaling non square images can cause unexpected results.
- Unet model settings
  - `unet_model` - String: Directory containing the pre trained Unet model
- Contouring / polygon settings
  - `minimum_contour_size` - Int: Minimum pixel area for a contour / polygon to be considered valid
- Logging settings
  - `log_file_prefix` - String: Prefix that will be attached to all log files.
  - `log_storage_directory` - String: Location to store log files
- Output settings
  - `raster_shp_output_prefix` - String: Prefix for output shape and raster files. Output files names will also include date and time the file was created. eg "brewarrina_20211010_07-08-01.jp2"
  - `output_directory` - String: Location to store shape files, inspection and zip file.

Imports and system setup

Below cells configure the python / conda environment and performs setup tasks such as configuring the logging mechanisms. Please note it can take some time for the conda environment to completes its setup.

In [ ]:

```
# Enable python module autoreload
%load_ext autoreload
%autoreload 2
```

In [ ]:

```
#Image constants
nrg_image_s3_source = "s3://live-demo-images/brewarrina/"
nrg_image_storage_directory = "../../Images/"
patch_size = 512 # in pixels, value needs to match that of the UNet CNN
image_scale = 1024
ignore_nonsquare_images = False
#Unet model constants
unet_model = "../../Models/CNN/flood_standard_unet_512x512_7epochs_binaryloss"
#Contour constants
minimum_contour_size = 100 # pixels
#Log file constants
log_file_prefix = "NOTEBOOK_LOG"
log_storage_directory = "../../Logs/"
#Output settings
raster_shp_output_prefix = "brewarrina"
output_directory = "../../Out/"
```

In [ ]:

```
!pip install patchify
!pip install 'opencv-python>=4.5.3.56' # required as older versions fail opening some jp2 images
!conda install -y -c conda-forge gdal fiona rasterio # this will take a long time ~25mins
```

In [ ]:

```
# imports modules - this could be moved to a seperate file once modules have stabilised.
#-- Standard Libraries
import os
from os import environ
environ["OPENCV_IO_ENABLE_JASPER"] = "true"
import sys
import datetime
import logging
import shutil
import gc

#-- Utility Libraries
import numpy as np
import matplotlib.pyplot as plt
import pickle
import boto3
try:
    from urlparse import urlparse
except ImportError:
    from urllib.parse import urlparse
from IPython.display import clear_output

#-- Mapping
from osgeo import gdal
from osgeo import osr
import fiona
import rasterio
from rasterio.io import MemoryFile
import rasterio.merge

#-- image processing
import cv2
from patchify import patchify
from PIL import Image

#-- CNN
import tensorflow as tf
from keras.models import load_model

#-- patch smoothing
import scipy.signal
from tqdm import tqdm
from smooth_tiled_predictions import predict_img_with_smooth_windowing
```

In [ ]:

```
cell_error = False
# Configure logging

# safeguard to prevent accidently running this more than once, running more than once breaks logging functionality
if not 'logger_intiated' in globals():
    logger = logging.getLogger(log_file_prefix)
    # To use different't log level for file and console
    timestamp = datetime.datetime.utcnow().strftime('%Y%m%d_%H-%M-%S')
    formatter = logging.Formatter('[%(asctime)s] %(name)s %(levelname)s - %(message)s')
    # File settings

    filename = os.path.join(log_storage_directory , f"{log_file_prefix}_{timestamp}.log" )
    try:
        file_handler = logging.FileHandler(filename=filename)
        file_handler.setLevel(logging.DEBUG)
        file_handler.setFormatter(formatter)
        #stream stdout settings
        stream_handler = logging.StreamHandler(sys.stdout)
        stream_handler.setLevel(logging.INFO)
        stream_handler.setFormatter(formatter)
        # The handlers have to be at a root level since they are the final output
        logger.addHandler(stream_handler)
        logger.addHandler(file_handler)
        logger.setLevel(logging.DEBUG)

        # Log global debug to a seperate file. Log file will include debug logs from import modules
        # File settings
        global_filename=os.path.join(log_storage_directory , f"{log_file_prefix}_EXTRA_{timestamp}.log" )
        file_handler = logging.FileHandler(filename=global_filename)
        file_handler.setLevel(logging.DEBUG)
        file_handler.setFormatter(formatter)
        logging.basicConfig(
            level=logging.DEBUG,
            handlers=[
                file_handler
            ]
        )
        logger.info(f"Finished configuring logging. Log file: {filename}, Global log file: {global_filename}")
        logger_intiated = True
    except Exception as e:
        cell_error = True
        logger.error(f"Unable to configure logging: {e}.", exc_info=True)
else:
    logger.warning(f"Logger is already configured. Log file: {filename}, Global log file: {global_filename}")

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
# Log settings to file
logger.debug(f"Notebook settings:\n"
f"nrg_image_s3_source  {nrg_image_s3_source} {type(nrg_image_s3_source)} \n"
f"nrg_image_storage_directory  {nrg_image_storage_directory} {type(nrg_image_storage_directory)} \n"
f"image_scale {image_scale} {type(image_scale)}\n"
f"patch_size  {patch_size} {type(patch_size)}\n"
f"ignore_nonsquare_images  {ignore_nonsquare_images} {type(ignore_nonsquare_images)}\n"
f"unet_model  {unet_model} {type(unet_model)}\n"
f"minimum_contour_size  {minimum_contour_size} {type(minimum_contour_size)}\n"
f"log_file_prefix  {log_file_prefix} {type(log_file_prefix)}\n"
f"log_storage_directory  {log_storage_directory} {type(log_storage_directory)}\n"
f"raster_shp_output_prefix  {raster_shp_output_prefix} {type(raster_shp_output_prefix)}\n"
f"output_directory  {output_directory} {type(output_directory)}")
```

Download images

Download images from s3 bucket and store in directory `nrg_image_storage_directory`. Once downloaded image path will be enumerated for images of jpeg2000. Extension must be `.jp2`.

In [ ]:

```
cell_error = False

s3_client = boto3.client('s3')

def download_dir(prefix, local, bucket, client=s3_client):
    """
    params:
    - prefix: pattern to match in s3
    - local: local path to folder in which to place files
    - bucket: s3 bucket with target contents
    - client: initialized s3 client object
    """
    keys = []
    dirs = []
    next_token = ''
    base_kwargs = {
        'Bucket':bucket,
        'Prefix':prefix.lstrip('/'),
    }
    while next_token is not None:
        kwargs = base_kwargs.copy()
        if next_token != '':
            kwargs.update({'ContinuationToken': next_token})
        results = client.list_objects_v2(**kwargs)
        contents = results.get('Contents')
        for i in contents:
            k = i.get('Key')
            if k[-1] != '/':
                keys.append(k)
        next_token = results.get('NextContinuationToken')
    for k in keys:
        fn = os.path.split(k)[-1]
        dest_pathname = os.path.join(local, fn)
        logger.debug(f"Downloading {fn} to {local} from bucket {bucket}")
        client.download_file(bucket, k, dest_pathname)

# split the s3 path in to components
s3_path_components = urlparse(nrg_image_s3_source, allow_fragments=False)
# download files
logger.info(f"Started downloading all files from {nrg_image_s3_source} to {nrg_image_storage_directory}")
try:
    download_dir(s3_path_components.path, nrg_image_storage_directory, s3_path_components.netloc)
    logger.info(f"Finished downloading files from {nrg_image_s3_source}")
except Exception as e:
    logger.error(f"Unable to download images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
cell_error = False

#populate list of image paths
nrg_image_paths = []

# Get image paths
try:
    # get all file names
    for file in os.listdir(nrg_image_storage_directory):
        if file.endswith(".jp2"):
            path = os.path.join(nrg_image_storage_directory, file)
            nrg_image_paths.append(path)
    # Count images
    if len(nrg_image_paths) > 0:
        logger.info(f"Directory {nrg_image_storage_directory} contains {len(nrg_image_paths)} images")
    else:
        # Raise exception if no images found
        raise Exception(f"No images in directory {nrg_image_storage_directory}")
        cell_error = False
except Exception as e:
    logger.error(f"Unable to open images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

Open images and perfrom predictions

Below will load the 'pretrained' Unet model. Each jp2 image contained in `nrg_image_storage_directory` will be opend and preprocessed by scaling into `image_scale` and further patchifying images in `patch_size` images. Patches will then be given to the CNN for predictions.

In [ ]:

```
cell_error = False
# open the unet model
try:
    unet = load_model(unet_model, compile=False)
    logger.info(f"Opened Unet model {unet_model}")
except Exception as e:
    logger.error(f"Unable to open Unet model {unet_model}: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
cell_error = False

binary_masks = []
used_nrg_image_paths = []
# loop through all images
logger.info(f"Started opening {len(nrg_image_paths)} images in {nrg_image_storage_directory}.")
for i in range(len(nrg_image_paths)):
    try:
        # build image path
        image_name = os.path.split(nrg_image_paths[i])[-1]
        # open the image
        nrg_raw_image = cv2.imread(nrg_image_paths[i])
        #reject non square images
        if nrg_raw_image.shape[0] != nrg_raw_image.shape[1] and ignore_nonsquare_images:
            logger.debug(f"Rejected {nrg_image_paths[i]}, image not square. Image shape {nrg_raw_image.shape}")
            continue
        shape = nrg_raw_image.shape
        nrg_raw_image = np.array(nrg_raw_image)
        used_nrg_image_paths.append(nrg_image_paths[i])
        logger.debug(f"Opened {nrg_image_paths[i]}. Image shape {nrg_raw_image.shape}")
        #scale image if required
        nrg_raw_image = cv2.resize(nrg_raw_image, (image_scale, image_scale), interpolation=cv2.INTER_AREA)
        nrg_raw_image = cv2.cvtColor(nrg_raw_image, cv2.COLOR_RGB2BGR)
        logger.debug(f"Scaled image {image_name} to {image_scale}. New image shape {nrg_raw_image.shape}")
        # perform patching and prediction on source images
        logger.debug(f"Performing prediction on image")
        prediction = predict_img_with_smooth_windowing(
        nrg_raw_image,
        window_size=patch_size,
        subdivisions=2,  # Minimal amount of overlap for windowing. Must be an even number.
        nb_classes=2,
        pred_func=(
            lambda img_batch_subdiv: unet.predict((img_batch_subdiv))
            )
        )
        logger.debug(f"Finished predictions, final prediction shape {prediction.shape}")

        tmp_img = cv2.resize(prediction[:,:,1], shape[0:2], interpolation=cv2.INTER_LINEAR) #todo remove resize hard code

        binary_masks.append(tmp_img >= 0.001 )

    except Exception as e:
        logger.error(f"Unable to process file {nrg_image_paths[i]}: {e}", exc_info=True)
        cell_error = True
        continue

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished opening and processing {len(used_nrg_image_paths)} images")
```

Combine Extracted Flood Extents

After predicting is performed on supplied images. Returned predictions are joined together to form one large binary image. The binary image is then 'contoured' to extract edges of flood extent.

In [ ]:

```
# open images as geo data sets
cell_error = False
logger.info(f"Combining images")
datasets = []
try:
    for i in range(len(used_nrg_image_paths)):
        logger.debug(f"Opening {used_nrg_image_paths[i]}")
        ds = rasterio.open(used_nrg_image_paths[i])
        # get dataset profile
        profile = ds.profile
        profile.update(
            dtype=rasterio.uint8,
            count=1)
        ds.close()
        # append data set to array
        logger.debug(f"Opening temporary memory file for binary image")
        memfile = MemoryFile()
        mem_ds = memfile.open(**profile)
        mem_ds.write(binary_masks[i], 1)
        datasets.append(mem_ds)
    #combine datasets and save to disk
    timestamp = datetime.datetime.utcnow().strftime('%Y%m%d_%H-%M-%S')
    raster_file = os.path.join(output_directory, f"{raster_shp_output_prefix}_{timestamp}.jp2")
    logger.debug(f"Merging {len(datasets)} binary images and writing to {raster_file}")
    rasterio.merge.merge(datasets, dst_path=raster_file, dst_kwds=profile)
    # open dataset and read flood layer
    logger.debug(f"Opening binary raster image as raster dataset")
    merged_dataset = rasterio.open(raster_file)
    logger.debug(f"Reading flood extent raster")
    flood_layer = merged_dataset.read(1)
    logger.info(f"Finished combining images")

except Exception as e:
    logger.error(f"Error combining images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished combining images")
```

In [ ]:

```
cell_error = False
try:
    contour_results = []
    i = 0
    logger.info(f"Started contouring binary image")
    contours, hierachy = cv2.findContours(np.uint8(flood_layer), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    # eliminate contours less than minimum area
    for c in range(len(contours)):
        if cv2.contourArea(contours[c]) > minimum_contour_size:
            contour_results.append(contours[c])
            # reverse orientation to create holes in shape
            if hierachy[0][c][3] != -1:
                contour_results[-1] = np.flipud(contour_results[-1])

    logger.info(f"Found {len(contour_results)} contours, {len(contours) - len(contour_results)} eliminated")
except Exception as e:
    logger.error(f"Error contouring images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished contouring / creating polygons")
```

Shape file

Create ESRI shape files using polygons created from the contouring process.

In [ ]:

```
def pixel2location(dx,dy):
    '''Convert pixel coordinates to spatial coordinates
    dx: x axis pixel
    dy: y axis pixel
    '''
    x = dx * pixel_x_size + pixel_x_offset
    y = dy * pixel_y_size + pixel_y_offset
    return x,y

# define shp schema
schema = {
    'geometry':'MultiPolygon',
    'properties':[('tag','str')]
}

cell_error = False
logger.info("Started converting contours / polygons to shape file")

# get the transform data
transform = merged_dataset.transform
pixel_x_offset = transform[2]
pixel_x_size = transform[0]
pixel_y_offset = transform[5]
pixel_y_size = transform[4]
logger.debug(f"Spatial data: X pixel width = {pixel_x_size}m, Y pixel height = {pixel_y_size}m, pixel (0, 0) location = {pixel_x_offset}, {pixel_y_offset}")

# loop through each contour
if len(contour_results) > 0:
    logger.debug(f"{len(contour_results)} polygons will be created")
# Convert pixel data points to spatial
multi_polygon = []
for contour in range(len(contour_results)):
    polygon = []
    multi_polygon.append(polygon)
    # convert polygon pixel points to spatial points and store
    for pixel in contour_results[contour]:
        #convert
        x, y = pixel2location(pixel[0][0], pixel[0][1])
        #store
        polygon.append([x, y])

# write to shape file
shp_file_output = os.path.join(output_directory, f"{raster_shp_output_prefix}_{timestamp}.shp")
logger.debug(f"Writing {shp_file_output}")
try:
    with fiona.open(shp_file_output, 'w', 'ESRI Shapefile', schema=schema, crs=merged_dataset.crs) as shp_file:
        logger.debug(f"Shape file espg: {shp_file.crs}")
        shp_file.write({
            'geometry' : {'type':'MultiPolygon',
            'coordinates': [multi_polygon]}, # Here the xyList is in brackets
            'properties': {
                'tag': ' flood extent'
            }
        })

except Exception as e:
    logger.error(f"Unable to write shape file {shp_file_output}: {e}", exc_info=True)
    cell_error = True
else:
     logger.debug(f"No polygons. No shape file will be generated")

if cell_error:
    logger.warn(f"Finished converting polygons to shape file with errors")
else:
    logger.info(f"Finished converting polygons to shape file")
```

Compress and clean up

Compress (zip) the output files and remove any temporary files created or downloaded onto the local machine. A zip file containing the output results will be left in the respository root.

In [ ]:

```
try:
    x = datetime.datetime.now()
    zip_file_name = (f"{x.year}{x.month}{x.day}{x.hour}{x.minute}_Outputs")
    zip_file = os.path.join(f"{output_directory}/../", zip_file_name)
    logger.info(f"Starting compression of {output_directory}, writing to {zip_file}.zip")
    shutil.make_archive(f"{zip_file}", 'zip', output_directory)
    logger.info(f"Success compressing and writing outputs to {zip_file}.zip")
    logger.info(f"Cleaning up")
    logger.info(f"Removing files from images storage {nrg_image_storage_directory}")
    for file in os.listdir(nrg_image_storage_directory):
        if file.endswith(".zip") or os.path.isdir(file):
            pass
        else:
            os.remove(os.path.join(nrg_image_storage_directory, file))
    logger.info(f"Removing files from outputs storage {output_directory}")
    for file in os.listdir(output_directory):
        if file.endswith(".zip") or os.path.isdir(file):
            pass
        else:
            os.remove(os.path.join(output_directory, file))
except Exception as e:
    logger.error(f"Unable to compress and clean up: {e}", exc_info=True)
```