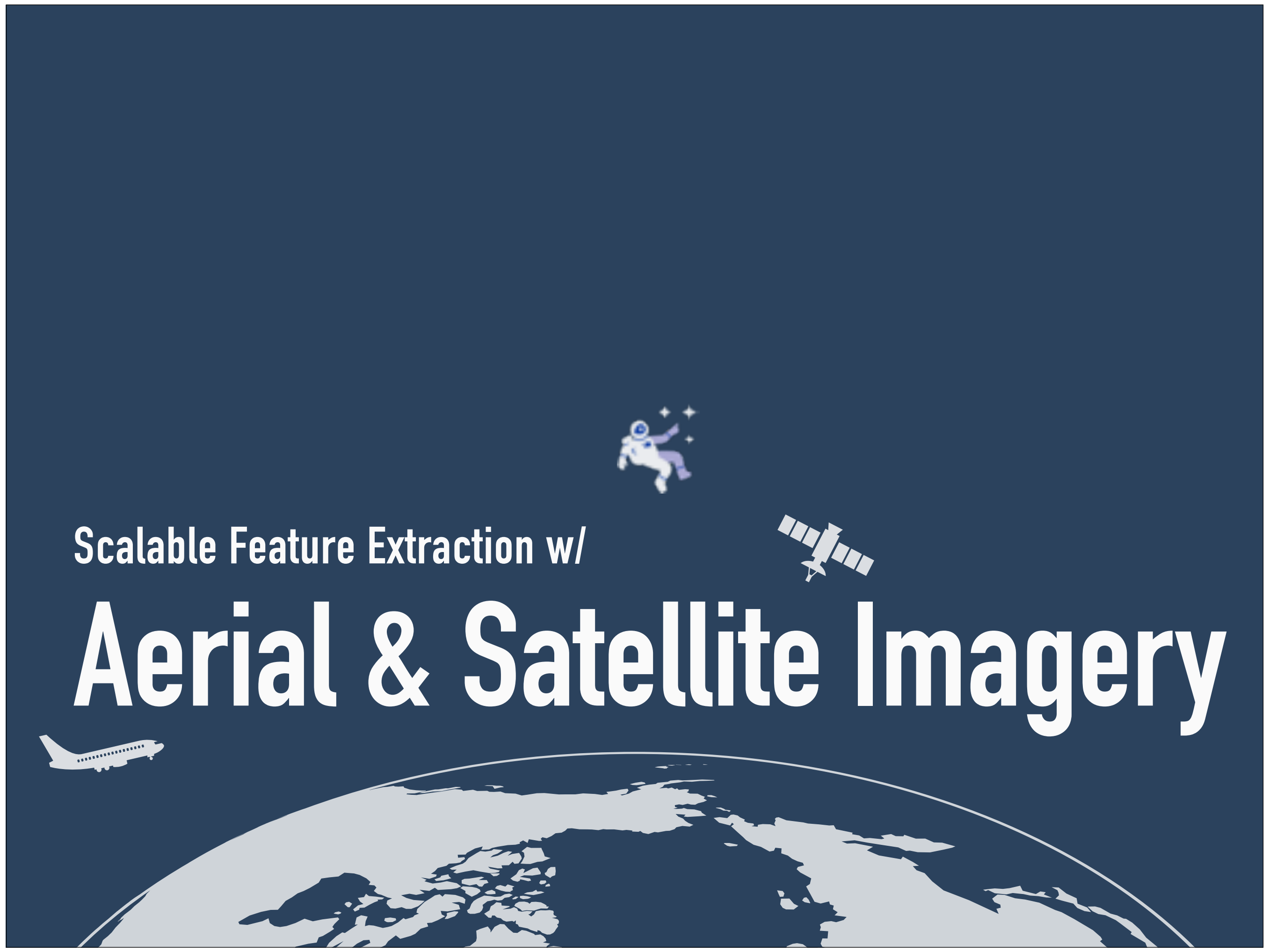
Scalable Feature Extraction w/

Aerial & Satellite Imagery



MAPBOX

The world is connected — from mobile, to cars, to disaster response.

Location underpins it all.

Mapbox is the location data platform for mobile and web applications. We provide building blocks to add location features like maps, search, and navigation into any experience you create.

Location is built into the fabric of our daily experiences. Whether you're exploring a city with Lonely Planet, sharing with friends on Snapchat, seeing if it's going to rain on Weather.com, tracking breaking news on Bloomberg — location is essential to every one of these applications, and they're powered by Mapbox.

DESIGNED WITH OPEN-SOURCE TOOLS

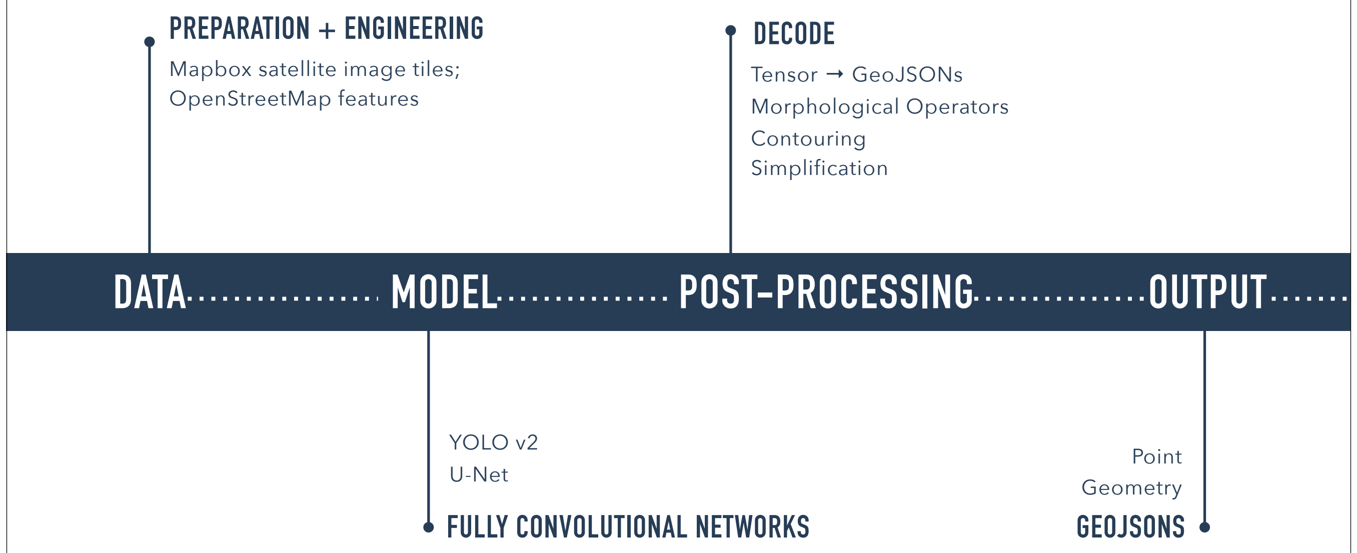
- Training data compiled from OSM and Mapbox Maps API
- Designed our processing pipelines and tools with open-source libraries like Scipy, Rasterio, Fiona, Osium, JSOM, Keras, PyTorch, OpenCV etc

OpenStreetMap

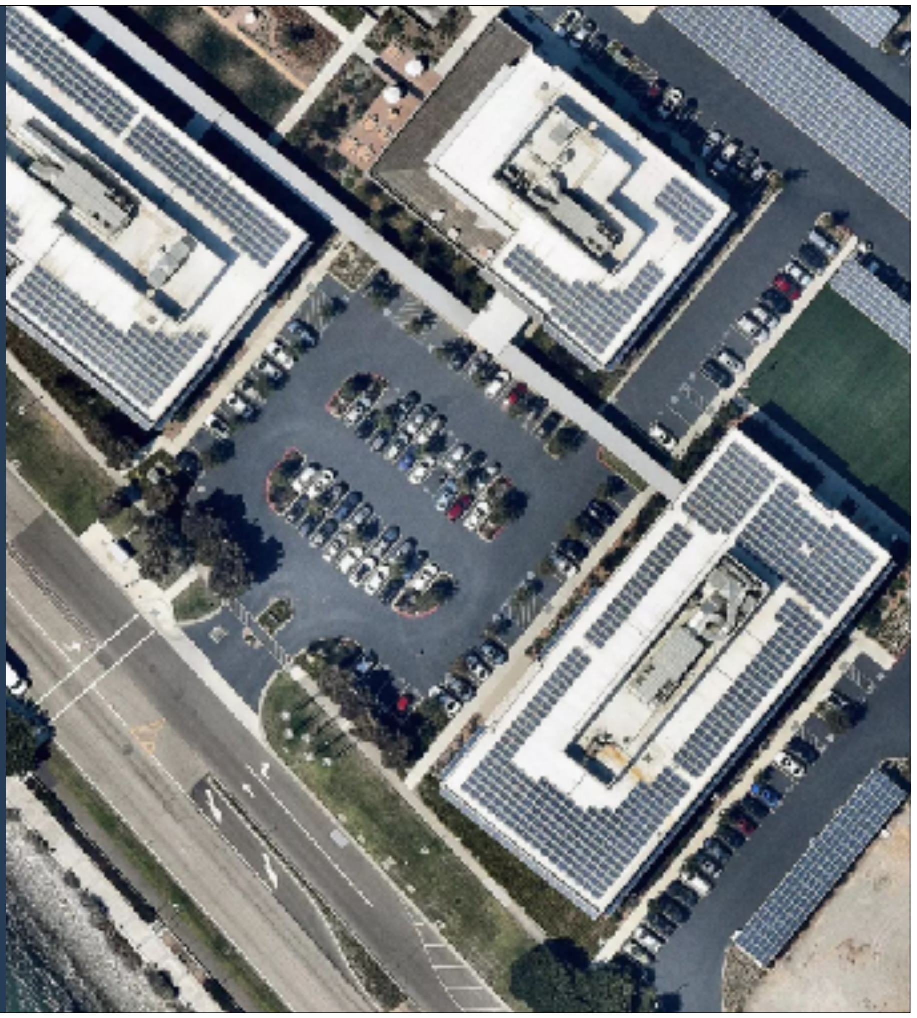
NAVIGATION

Navigation:
In particular, our navigation products are focused on providing smart turn-by-turn routing based on real-time traffic.

Valuable Assets For Routing, Maps, and Geocoding:
Turn Restrictions, Turn Lane Arrows, Parking Lots, Buildings, Grass, Trees, Parks, Water, Bridges.
We can manually map them, or we can abstract these assets from imagery.


PREPARATION + ENGINEERING
Mapbox satellite image tiles;
OpenStreetMap features
DECODE
Tensor → GeoJSONs
Morphological Operators
Contouring
Simplification
DATA
MODEL
POST-PROCESSING
OUTPUT
YOLO v2
U-Net
FULLY CONVOLUTIONAL NETWORKS
Point
Geometry
GEOJSONS

DATA



OBJECT DETECTION

- Locating and classifying a variable number of objects in an image.
- Other practical applications of object detection include face detection, counting, visual search engine.

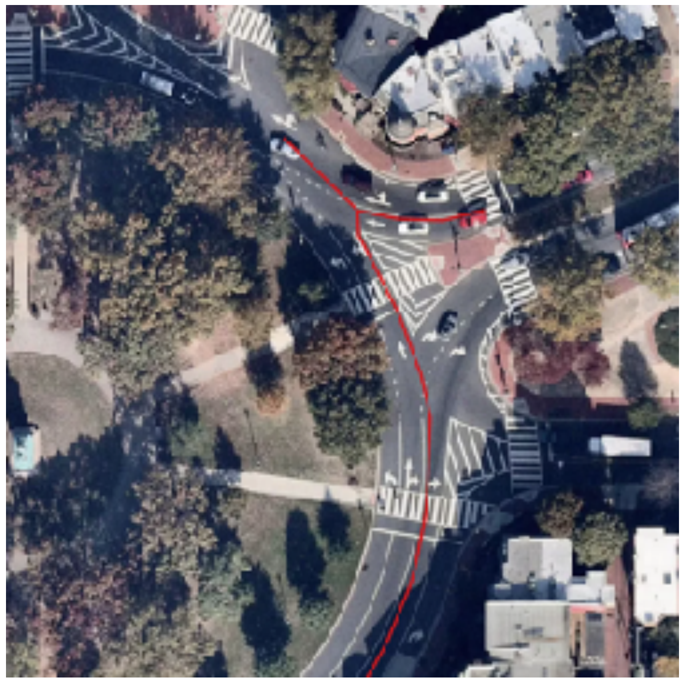
SEMANTIC SEGMENTATION

- Understanding an image at pixel level.
- Apart from recognizing the road from the buildings, we also have to delineate the boundaries of each object.

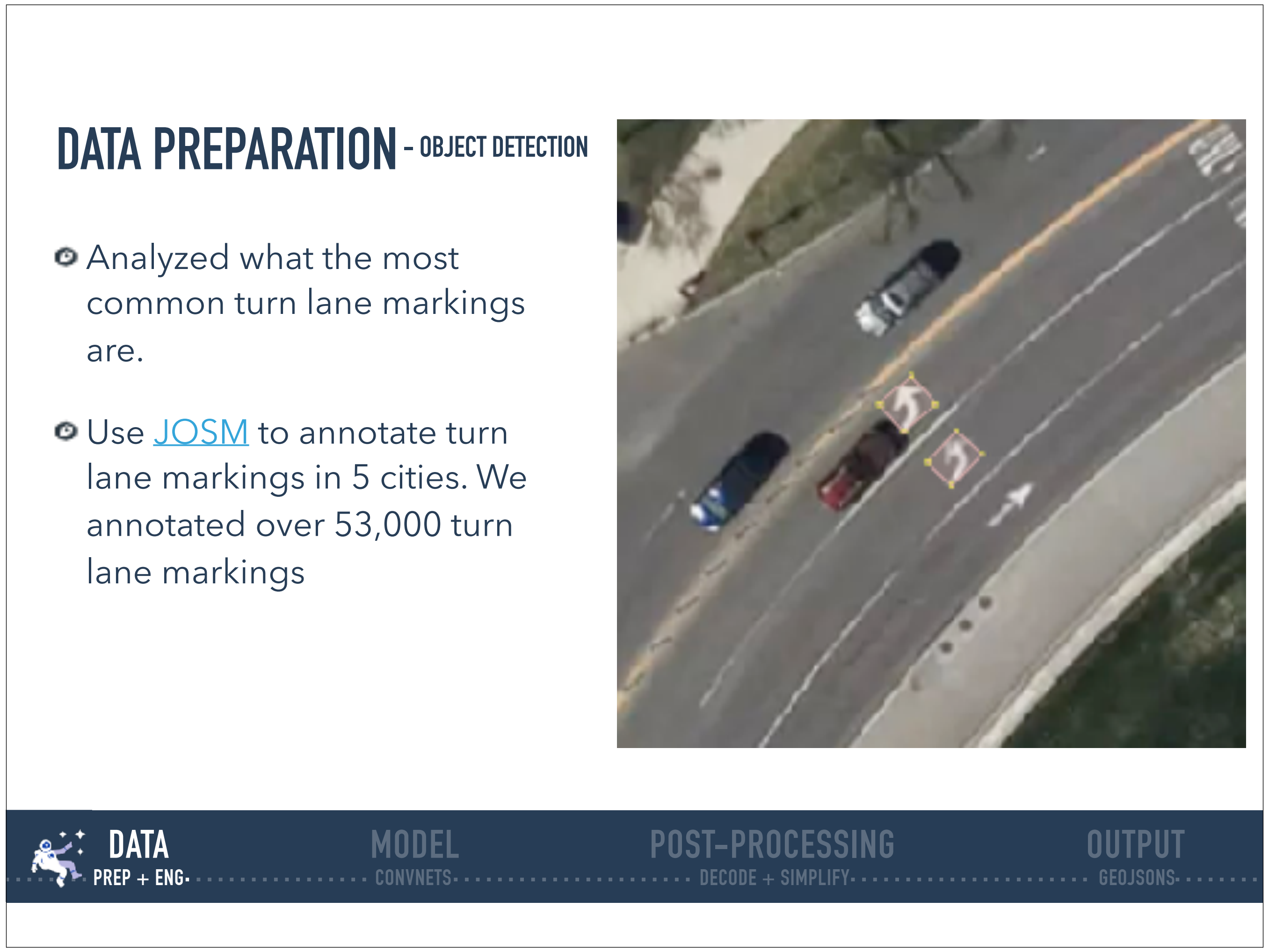
DATA PREPARATION - OBJECT DETECTION

- Where are the turn lane markings? Turn lane markings on OpenStreetMap are tagged as "ways" (line-strings)
- OpenStreetMap tags:

```
turn:lane=*
turn:lane:forward=*
turn:lane:backward=*
```

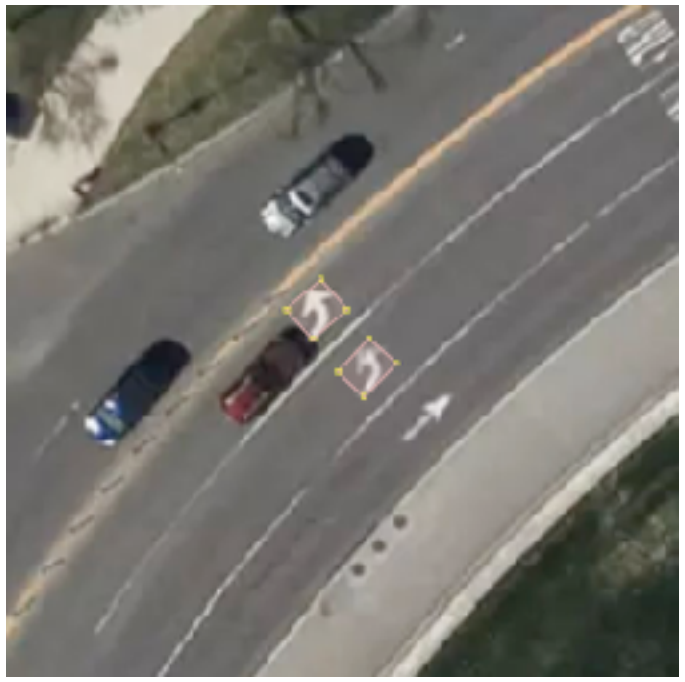


First find Where the Turn Lane Markings Are: Turn lane markings on OSM are recorded as "ways" (line-strings)
Used https://overpass-turbo.eu/ to query OpenStreetMap data. OpenStreetMap is a collaborative project to create a free editable map of the world. Extracted GeoJSONs in 5 cities from OSM that have one of the following attributes (aka OSM tags) and created a custom layer over mapbox.satellite layer:
turn:lane=*
turn:lane:forward=*
turn:lane:backward=*



DATA PREPARATION - OBJECT DETECTION

- Analyzed what the most common turn lane markings are.
- Use JOSM to annotate turn lane markings in 5 cities. We annotated over 53,000 turn lane markings

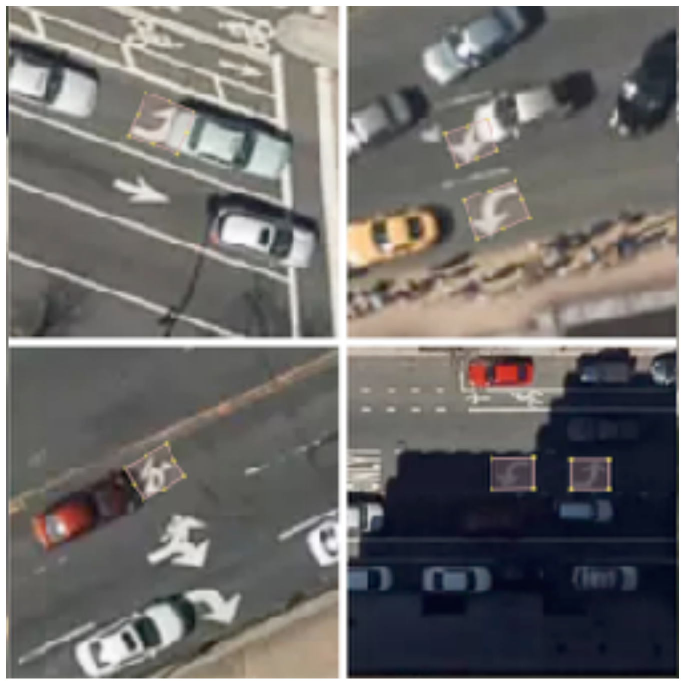


Annotated 6 classes of turn lane markings: Left, Right, Through (straight), ThroughLeft, ThroughRight, Other
Use JOSM (https://josm.openstreetmap.de/) to annotate (draw bounding box around the turn lane) turn lane signs in 5 cities. We annotated over 53K turn lane signs with bounding boxes.

JOSM is an extensible editor for OpenStreetMap (OSM) for Java 8+.
It supports loading GPX tracks, background imagery and OSM data from local sources as well as from online sources and allows to edit the OSM data (nodes, ways, and relations) and their metadata tags.
JOSM is open source and licensed under GPL

DATA PREPARATION - OBJECT DETECTION

- Included Turn Lanes Markings:
  - Of all shapes and sizes
  - Partially covered by cars
  - Covered by shadows
- Excluded Turn Lane Markings:
  - Fully covered by cars
  - Erased



DATA PREPARATION - SEGMENTATION

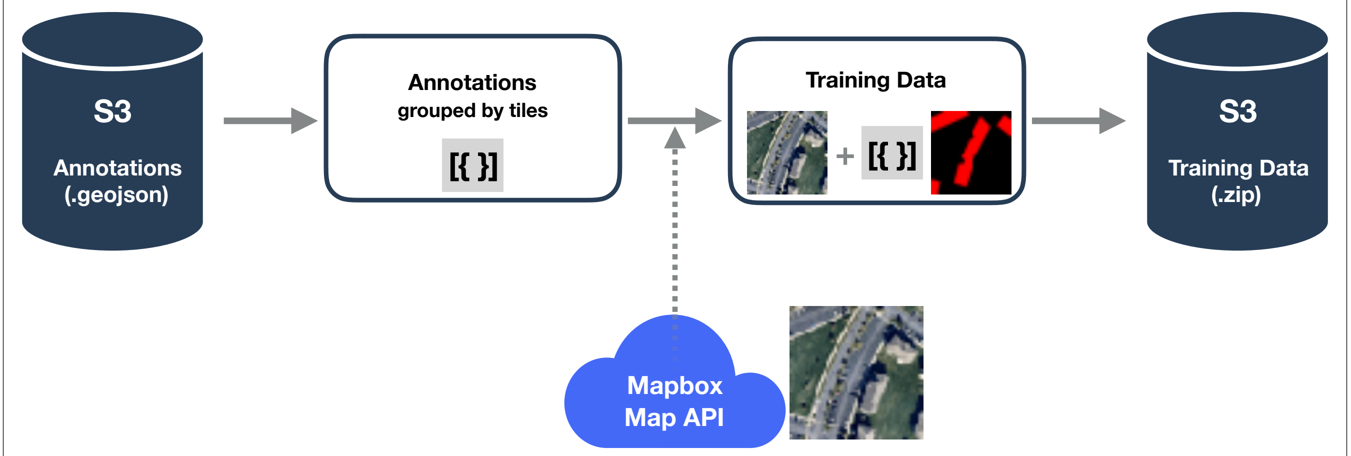
- Generate polygons from OpenStreetMap tags:
  `amenity=parking=*`
  `building=*`
- Excluded features that are not visible in satellite imagery.
- Use Osmium to annotate parking lots.

DATA PREP + ENG · MODEL CONVNETS · POST-PROCESSING DECODE + SIMPLIFY · OUTPUT GEOJSONS

Generate polygons from OpenStreetMap tags excluding features that are not visible in satellite imagery.
`Tag:amenity=parking=*` except underground, sheds, carports, garage_boxes
`building=*` except construction, houseboat, static_caravan, stadium, conservatory , digester, greenhouse, ruins

Scalability: our pipelines are fully generic. Users can generate training sets with any OpenStreetMap feature simply by writing their own osmium handler to turn OSM geometries into polygons.

DATA ENGINEERING

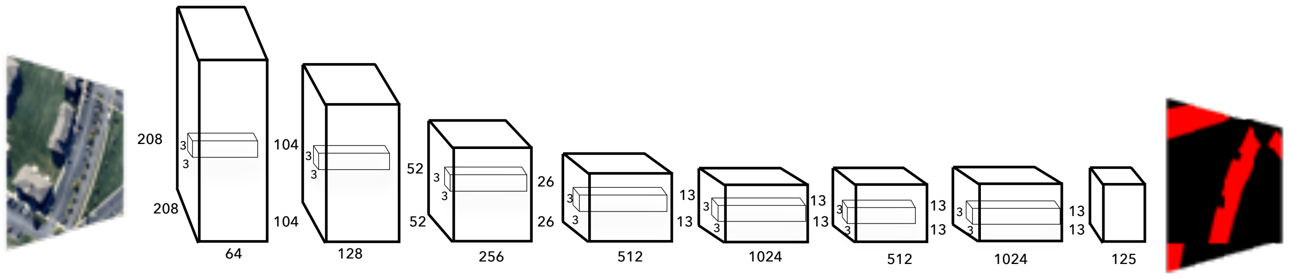


We built a data engineering pipeline within the larger object detection pipeline. This data engineering pipeline streams any set of prefixes off of s3 into prepared training sets.
First, we stream OSM features out of the GeoJSON files on S3 and merge classes and geographic bounding boxes into the feature attributes.
Next, we convert these into JSON image annotations grouped by tile. During this step, the feature bounding boxes are converted to image pixel coordinates.
The annotations are then randomly assigned to training and testing sets (80/20 split) and written to disk, joined by imagery fetched from the Mapbox Maps API. This is where the abstract tile in the pipeline is replaced by real imagery.
Finally, the training data is zipped and uploaded to S3.
In our first iteration, we wrote scripts for our data preparation steps (Python library and CLI). These scripts were then ran at large scale in parallel (multiple cities at once) by on AWS Amazon Elastic Container Service (Amazon ECS). ECS is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster (grouping of container instances).

ECS-watchbot
A library to help run a highly-scalable AWS service that performs data processing tasks in response to external events. We provide the messages and the logic to process them, while this library handles making sure that our processing task is run at least once for each message. This library is similar in many regards to AWS Lambda, but is more configurable, more focused on data processing, and not subject to several of Lambda's limitations.

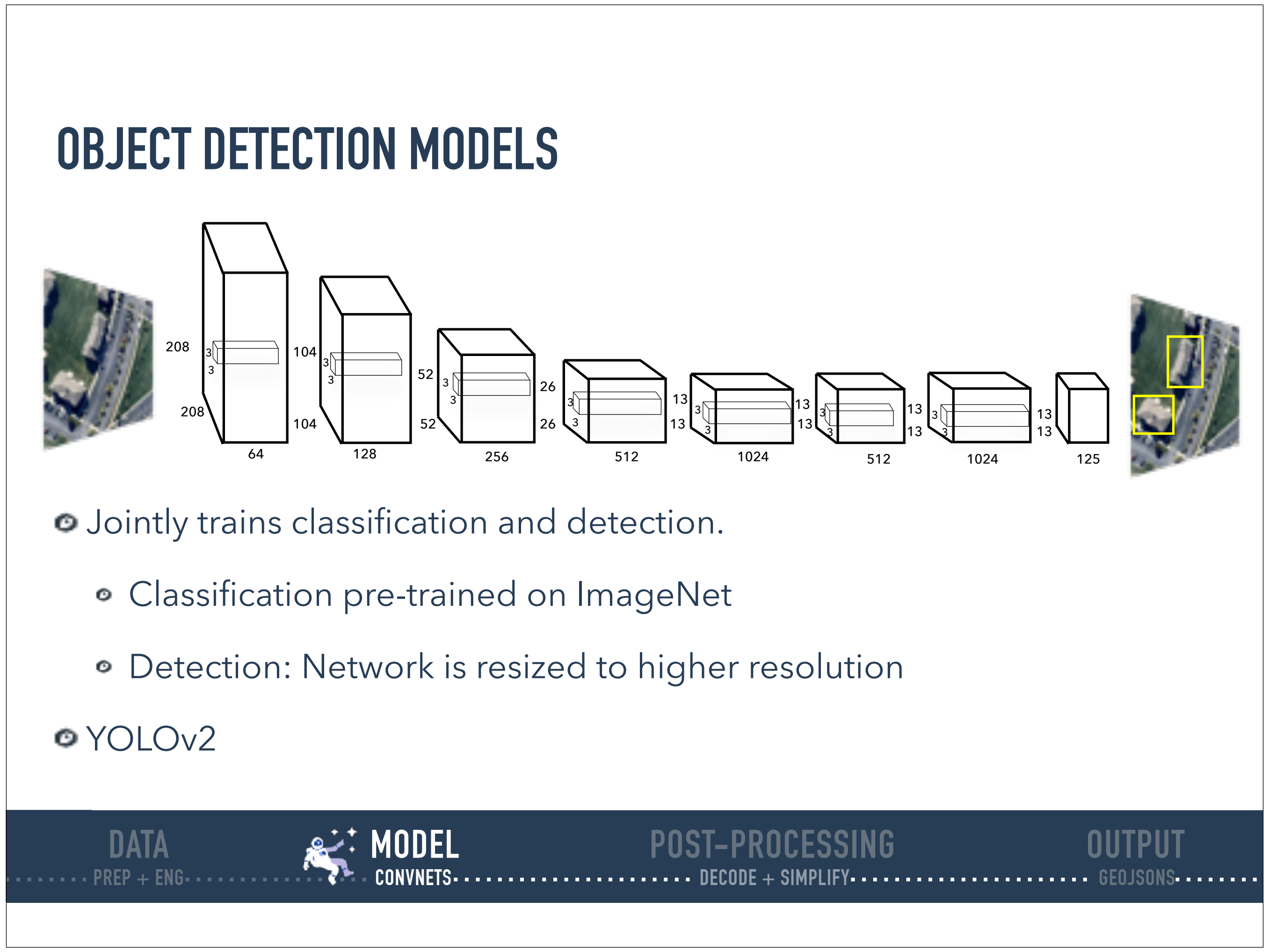
MODEL

FULLY CONVOLUTIONAL NEURAL NETWORKS

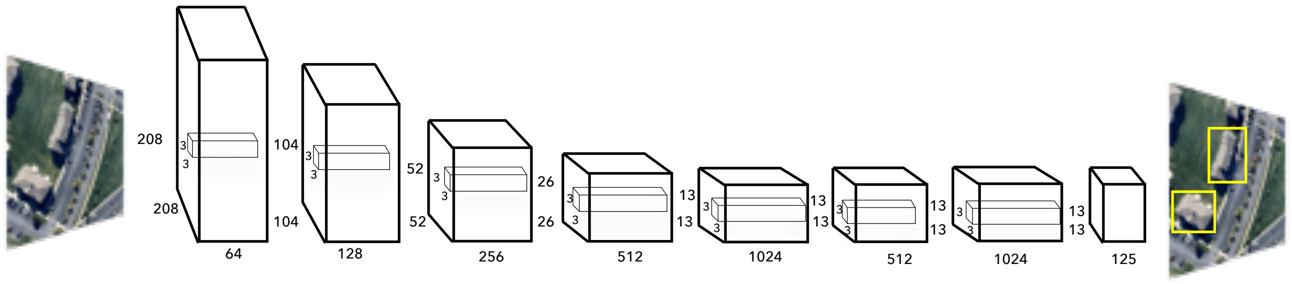


- Fully convolutional neural networks are composed of convolutional layers without any fully-connected layers or MLP usually found at the end of a neural network.
- Decision-making layers at the end of full convolutional networks are filters.
- Capable of managing different input sizes and are faster at this task.

Fully convolutional are neural networks composed of convolutional layers without any fully-connected layers or MLP usually found at the end of the network. A CNN with fully connected layers is just as end-to-end learnable as a fully convolutional one. The main difference is that the fully convolutional net is learning filters everywhere. Even the decision-making layers at the end of the network are filters. Traditional Convolutional neural networks containing fully connected layers cannot manage different input sizes , whereas fully convolutional networks can have only convolutional layers or layers which can manage different input sizes and are faster at that task.
A fully convolutional net tries to learn representations and make decisions based on local spatial input. Appending a fully connected layer enables the network to learn something using global information where the spatial arrangement of the input falls away and need not apply.

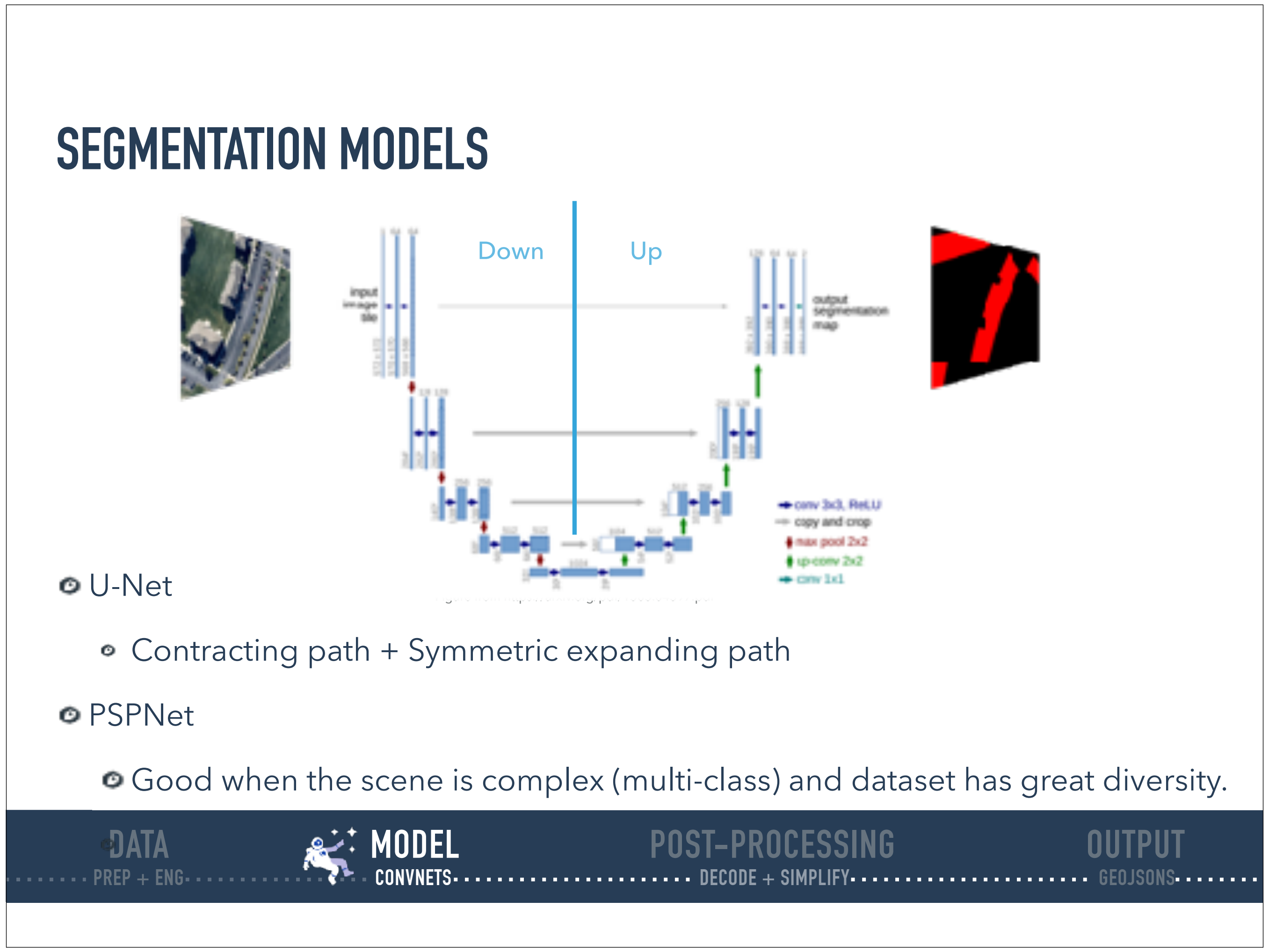


OBJECT DETECTION MODELS

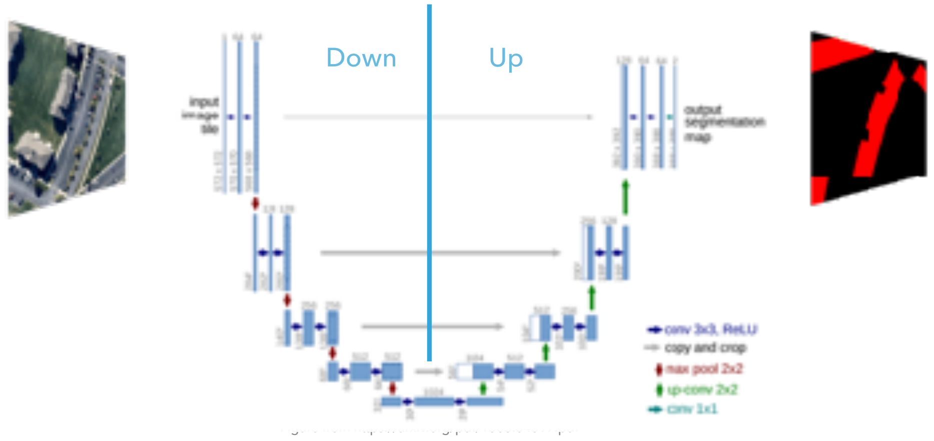


- Jointly trains classification and detection.
  - Classification pre-trained on ImageNet
  - Detection: Network is resized to higher resolution
- YOLOv2

The general way in which object detection works is, the model is pre-trained on ImageNet for classification. Then for detection, the network is resized to higher resolution especially to detect smaller objects in a scene. Fully convolutional models jointly trains these two steps. We implemented YOLOv2, a real-time object detection system and is the improved version of YOLO, which was introduced in 2015. YOLOv2 outperforms all the other state-of-the-art methods like Faster RCNN with ResNet and SSD in both speed and detection accuracy. Improvements made to YOLOv2 included batch normalization, which helped the model converge while regularizing it. Another change that was made to YOLO was the image resolution of which the network did resizing and fine-tuning. In generally, object detection models are pre-trained on ImageNet for classification. The network is then resized for higher resolution for detection. This has worked particular well on detecting smaller objects in a scene. YOLOv2 was first pre-trained on ImageNet (224x224) and then fine-tuned on (448x448). A major feature of the YOLO family is the use of anchor boxes to run prediction. There are two ways of predicting the bounding boxes- directly predicting the bounding box of the object or using a set of predefined bounding boxes (anchor box) to predict the actual bounding box of the object. YOLO predicts the coordinates of bounding boxes directly using fully connected layers on top of the convolutional feature extractor. But, it makes a significant amount of localization error. It is easier to predict the offset based on anchor boxes than to predict the coordinates directly. Instead of using pre-defined anchor boxes, YOLOv2 authors performed K-means clustering on bounding boxes from the training data set.



SEGMENTATION MODELS



- U-Net
  - Contracting path + Symmetric expanding path
- PSPNet
  - Good when the scene is complex (multi-class) and dataset has great diversity.

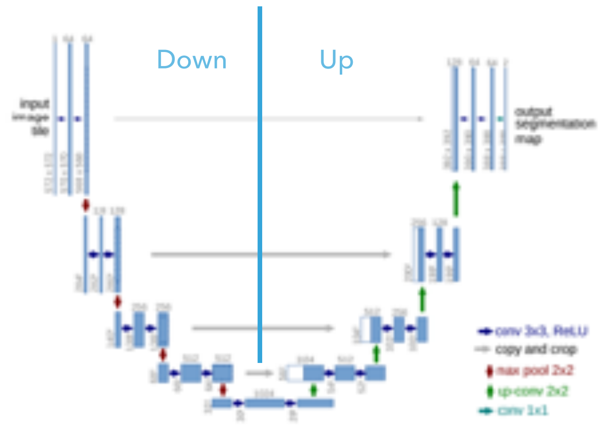
DATA PREP + ENG · MODEL CONVNETS · POST-PROCESSING DECODE + SIMPLIFY · OUTPUT GEOJSONS

Segmentation Models. We implemented U-Net for parking lot segmentation. The U-Net architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization. This type of network can be trained end-to-end with very few training images and yields more precise segmentations than prior best method such as the sliding-window convolutional network. (Above figure) This first part is called down or you may think it as the encoder part where you apply convolution blocks followed by a maxpool downsampling to encode the input image into feature representations at multiple different levels. The second part of the network consists of upsample and concatenation followed by regular convolution operations. Upsampling in CNNs may be a new concept to some of the readers but the idea is fairly simple: we are expanding the feature dimensions to meet the same size with the corresponding concatenation blocks from the left. While upsampling and going deeper in the network we are concatenating the higher resolution features from down part with the upsampled features in order to better localize and learn representations with following convolutions. For parking lots segmentation, we are doing binary segmentation distinguishing parking lots from the background.

We also experimented with Pyramid Scene Parsing Network (PSPNet). PSPNet is good when the scene is complex (multi-class) and dataset has great diversity. It’s redundant when the number of categories are less and dataset are more simple (such as self-driving car). PSP adds a multi-scale pooling on top of the backend model to aggregate different scale of global information. The upsample layer is implemented by bilinear interpolation. After concatenation, PSP fuse different level of feature with a 3x3 convolution.

HARDING NEGATIVE MINING

HARD NEGATIVES

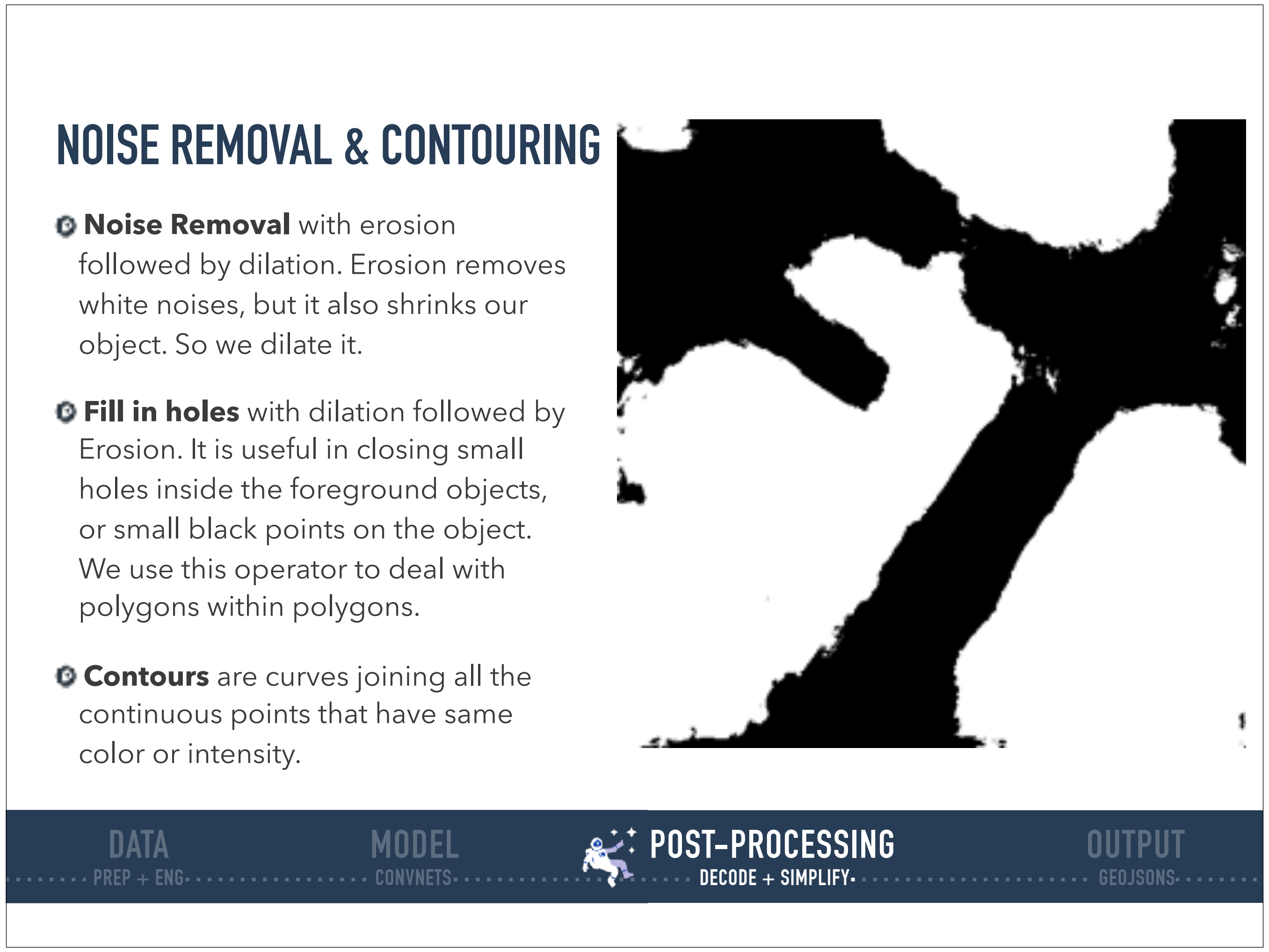


- A technique used to improve model performance by reducing the negative samples.
- Explicitly create a negative example out of a falsely detected patch, and add that negative to your training set.

Figure from https://arxiv.org/pdf/1505.04597.pdf

Hard Negative Mining. This is a technique we used to improve model performance by reducing the negative samples. A hard negative is when we take that falsely detected patch, and explicitly create a negative example out of that patch, and add that negative to our training set. When we retrain your model it should perform better with this extra knowledge, and not make as many false positives.

POST-PROCESSING



NOISE REMOVAL & CONTOURING

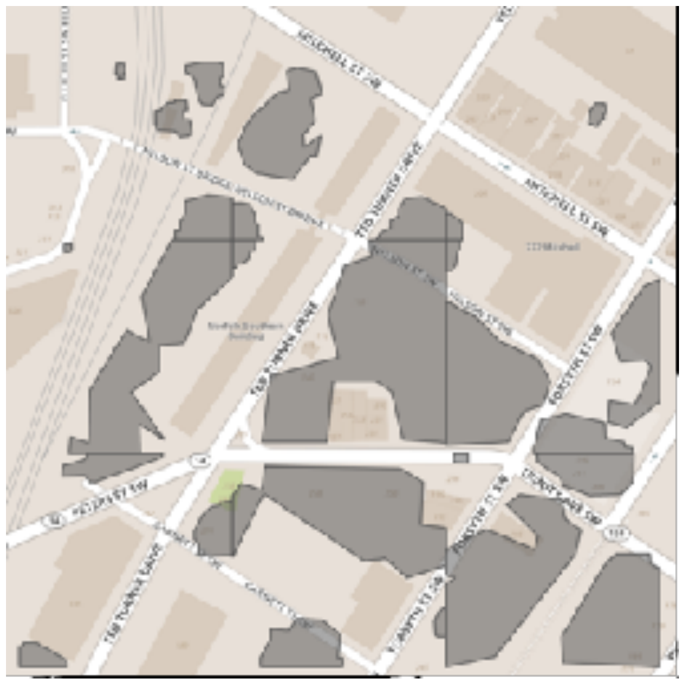
- **Noise Removal** with erosion followed by dilation. Erosion removes white noises, but it also shrinks our object. So we dilate it.
- **Fill in holes** with dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small black points on the object. We use this operator to deal with polygons within polygons.
- **Contours** are curves joining all the continuous points that have same color or intensity.

DATA PREP + ENG · MODEL CONVNETS · POST-PROCESSING DECODE + SIMPLIFY · OUTPUT GEOJSONS

Noise Removal. We remove noise in the data by performing two morphological operations: erosion followed by dilation. Erosion removes white noises, but it also shrinks our object. So we dilate it.
Fill in holes. We fill holes in the mask by performing dilation followed by erosion. It is especially useful in closing small holes inside the foreground objects, or small black points on the object. We use this operator to deal with polygons within polygons.
Contouring. Contours are curves joining all the continuous points that have same color or intensity.
Simplification. Douglas-Peucker Simplification takes a curve compared of line segments and finds a similar curve with fewer points. We get simple polygons that can be ingested by OSM as “nodes” and “ways”
Transform Data. Convert detection or segmentation results from pixel space back into GeoJSONs (world coordinate).
Removing tile border artifacts. Query and match neighboring image tiles.
Deduplication. Deduplicate by matching GeoJSONs with OSM data

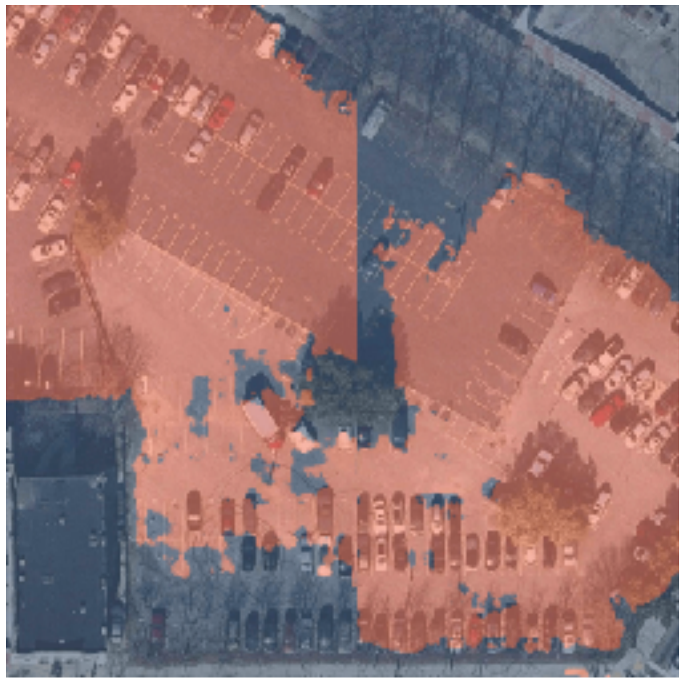
SIMPLIFY

- Douglas-Peucker Simplification takes a curve compared of line segments and finds a similar curve with fewer points. We get simple polygons that can be ingested by OSM as “nodes” and “ways”

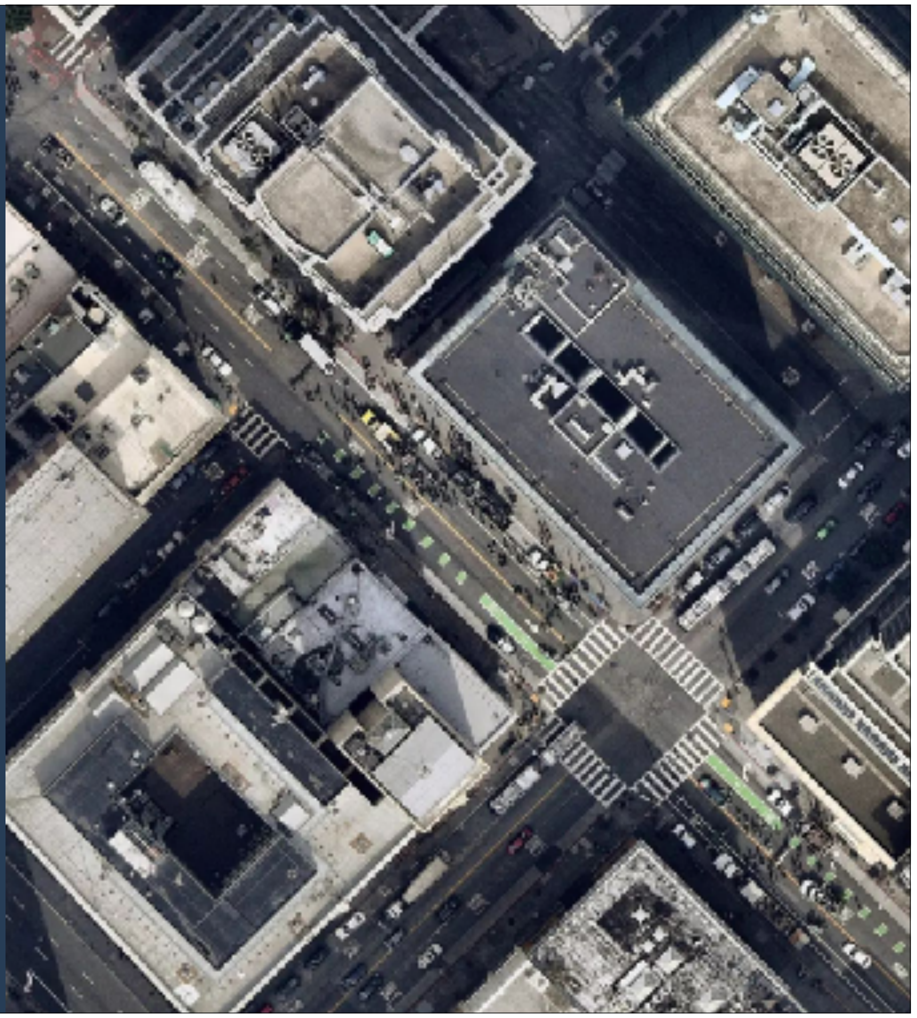


HANDLING TILE BORDER ARTIFACTS

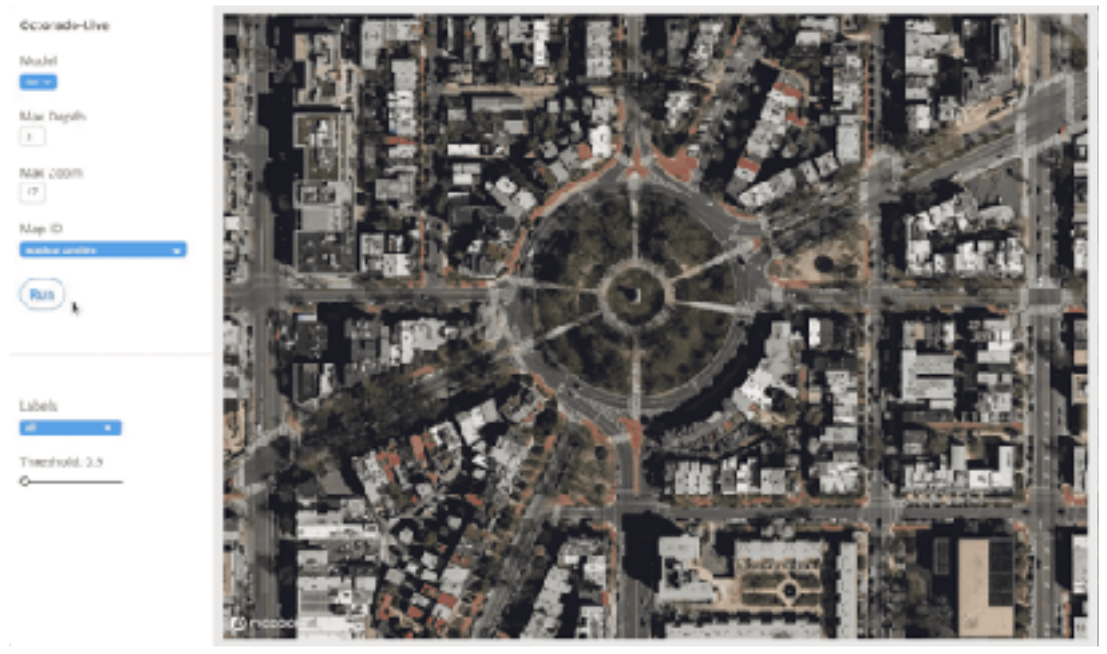
- Convert detection or segmentation results from pixel space back into GeoJSONs (world coordinate).
- Query and match neighboring image tiles
- Deduplicate by matching GeoJSONs with OSM data



OUTPUT



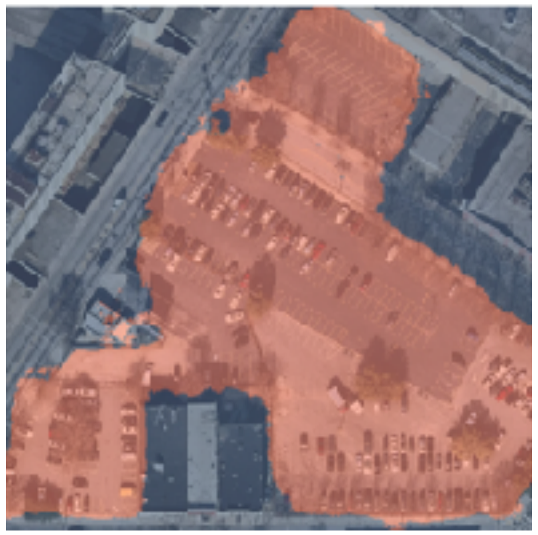
TURN LANE MARKING DETECTION



DATA PREP + ENG · MODEL CONVNETS · POST-PROCESSING DECODE + SIMPLIFY · OUTPUT GEOJSONS

With this pipeline design, we are able to run batch prediction at large scale. The output of these processing pipelines are turn lane markings and parking lots in the form of GeoJSONs. We can then add these GeoJSONs back into OpenStreetMap as turn lane and parking lot features. Our routing engines then take these OpenStreetMap features into account when calculating routes. We also built a front-end UI that allows users to pan around for instant turn lane markings detection

PARKING LOT SEGMENTATION



Raw Mask

Clean Mask

DATA
PREP + ENG

MODEL
CONVNETS

POST-PROCESSING
DECODE + SIMPLIFY

OUTPUT
GEOJSONS

Scalable Feature Extraction w/

Aerial & Satellite Imagery