Machine Learning model that detects road network and collects relevant data from satellite imagery (Road network inference)

Project Advisor: Huaizhu Gao (hg55)

Project Collaborators: Dr. Mohammad Tayarani (mt789)

Team: Logan Vegna (lnv8), Saharat Rodjanamongkol (sr865)

Introduction

Understanding important information about road networks can allow us to further our understanding about a multitude of travel and traffic-related topics, and possibly improve them. Such is the case in the upcoming research project Smart and Healthy Cities (“Smart and Healthy Cities.”) which aims to study the impact of disruptive technologies like autonomous vehicles on our current road networks. Two road network features that are important to studying this topic are road width, and the number of lanes a road has. Unfortunately, no datasets exist that have reliable information about these two topics. Additionally, manually sampling data such as these at a scale necessary for research is extremely time-consuming, taking at the very minimum 250 extremely tedious man-hours to collect 10 large cities' worth of data. Thus a scalable, automated system is necessary to efficiently collect enough data for future research projects and road network modeling. In this paper, we introduce a pipeline that, with minimal human interaction, can automatically sample road width, and lane counts for thousands of roads from one or multiple cities. The methods in this project rely heavily on Deep Machine Learning techniques and Open Street Maps. We provide working code to re-train the models, as well as pre-trained models and code to sample modern cities.

Dataset Creation

In order to train the lane count and width models, we need to first create a dataset. The lane count model requires satellite images and the associated number of lanes, while the road width model requires satellite images and the corresponding road centerline images, both with the dimension of 512 x 512 pixels. The satellite images are sourced from Google Static Map API while the road centerline images and the number of lanes are sourced from a Python library called OSMnx. See https://developers.google.com/maps/documentation/maps-static/start and

https://osmnx.readthedocs.io/en/stable/index.html for more details. We used OSMnx to pull the data from Open Street Map (OSM) and represented the road network data as a graph. Information about the road is stored on each edge and node of the graph. All the graph data is then saved in graphml format for future use.

We created the dataset of 11,000 data points by sampling 1000 edges that met certain conditions from 11 cities: Washington, Baltimore, Baltimore, Philadelphia, New York City, San Francisco, San Jose, Austin, Denver, Portland, Seattle, and Salt Lake City. The conditions are edges with properly formatted lane data, not a junction, tunnel, nor bridge, and have lengths between 20 and 1000 meters. These conditions help standardize the satellite image to be a straight road that is further away from the intersection and doesn't have roads on top of each other. We use zoom level 20 to get high-quality images of the roads with a lot of details on the lane marking and to reduce noises from other parts of the image.

Figure 1: Satellite image and Road centerline image at zoom level 20

	1	2	3	4	5	6	7	8	9	10
washington	8881	35586	6289	8936	813	1596	170	192	0	0
baltimore	10163	56368	6992	7600	1390	262	14	1	1	0
philadelphia	8805	32274	7680	4996	2209	583	266	2	0	0
new_york_city	19873	39932	19500	11841	2937	1058	320	10	0	0
san_francisco	4154	26158	7728	5569	1257	104	14	9	0	0
san_jose	6858	22565	17248	7112	4635	614	531	29	0	0
austin	4708	28071	8957	6015	3710	385	93	2	0	0
denver	9165	67488	12787	5767	4640	1067	424	222	40	16
portland	9666	74305	9835	7842	3349	390	40	18	0	0
seattle	32673	89645	6395	9025	4037	1148	504	18	0	0
salt_lake_city	2951	16971	4250	3539	2375	422	174	22	16	0

Table 1: Lanes distribution for each city in the graph

	1	2	3	4	5	6	7	8	9	10
washington	5	730	56	166	13	23	4	3	0	0
baltimore	15	811	39	104	29	2	0	0	0	0
philadelphia	11	629	87	181	69	18	5	0	0	0
new_york_city	67	563	49	227	66	25	3	0	0	0
san_francisco	5	755	101	101	36	2	0	0	0	0
san_jose	0	478	257	127	104	10	22	2	0	0
austin	1	689	29	143	126	8	4	0	0	0
denver	20	753	95	42	56	18	7	5	4	0
portland	19	751	84	93	41	10	2	0	0	0
seattle	8	755	54	96	55	20	12	0	0	0
salt_lake_city	1	592	126	140	109	12	16	2	2	0

Table 2: Lanes distribution of dataset for each city

While we tried to make the data and satellite images standardized and less ambiguous, there can still be some bad images in the dataset. Not all road edges have reliable lane number data and less than 1% of the road edges have width data. The width data that does exist is also very inconsistent, thus we decide not to use them. Conversely, the lane count data is relatively accurate, and usable for model training. However, there are some lane labeling inconsistencies with the turn lane and merge lane being counted as an additional lane some of the time, and not others. Some of the satellite images of the roads are occluded by trees and buildings or lack lane markings. We tried to avoid intersections and bridges but we couldn't fully eliminate them without manually reviewing every image. Additionally, the data we sampled is very imbalanced, with significantly more 2 lanes roads (around 60% of the roads) than other number of lanes and very few roads with more than 4 lanes.

Initially, we also look into lane detection models to detect where the road and centerline are in the satellite image. However, the OSMnx already provides accurate data on the road network, so we just use data from OSMnx to get the road network. We wanted to create a new dataset where the number of lanes is balanced and group lanes larger

than 4 together as one category. However, we ran out of time to do so at the end of the semester.

Running Dataset Creation

The notebook DataExtraction.ipynb handles dataset generation. Before running the whole notebook, change some of the variables under Configs and Constants. Important variables to look out for are:

- GOOGLE_MAP_API_KEY: the key is required for calling the Google static map API, see https://developers.google.com/maps/documentation/javascript/get-api-key#creating-api-keys on how to create your own key.
- places: list of locations to sample the data from. Add, remove, or edit the list to choose where to sample the data.
- numSample: number of edges to sample at each location.

The satellite images will be saved in the folder with the path defined by the variable satelliteImageFolder and under the subfolder for each city, ex *satelliteImage/new_york/new_york_city_1_sat.png*. Similarly, the centerline images will be saved in the folder defined by the variable centerLineImageFolder, ex *centerLineImage/new_york/new_york_city_1_osm.png*. The output data will be saved as a csv file with name and location defined by the variable csvPath. Each row of the csv file represents the data and features of a road edge. The data includes the path to satellite image, path to centerline image, one hot encoded highway type, whether the road is one way, the grade (slope) of the edge, the length of the edge, the node ids associated with the edge, the pixel and lat/long coordinate of the nodes, the elevation of the nodes, the points perpendicular to the edge, the name of the road, the city it is sampled from, the sample index, the edge id on OSM, and the number of lanes. The number of lanes is the label for the model to predict.

There are a couple of things to note about the dataset creation. Calling the API to download the satellite images and centerline images takes a very long time. It took around 2 days to download 11,000 images. The code does support rerunning the download and won't redownload the same data. Be aware that the data in the saved graphml files will be used by default but the centerline images will be fetched from the API. Therefore, the saved graph data can be stale which can result in coordinate mismatch between the saved graph and the freshly fetched centerline. The API call is used to get the centerline because it's much faster than using the saved graph data of the whole city. The "fix bad edges" section of the notebook can detect and fix the data where the edge changes, moves, or the number of lanes changes.

Number of Lanes Methods

One of the major portions of this project was the creation of a model which could automatically count the number of lanes of a given road at the center an image. At first, we looked into an existing model called RoadTagger (He), which combines both Convolutional Neural Networks (CNN) and Graph Neural Networks (GNN) to predict the number of lanes and road types. However, we decided not to pursue that model further and train our own model. The first reason was that we couldn't get it to run locally and on Google Collab. Second, the model requires the dataset to be a graph of road data so it also requires data on the surrounding road, which makes the dataset difficult to create, requires significantly more data, and doesn't align as well with our goal to predict random roads in the city not predicting the whole road network. Third, the model was trained using Tensorflow and we want to keep all our code in PyTorch for consistency and potential compatibility with ScRoadExtractor. Lastly, the CNN + GNN model doesn't perform much better than the CNN baseline to be worth the added complexity. In hindsight, we guess that the RoadTagger model might suffer from the same problem as our model as we will discuss later.

The first attempt to train the model was using transfer learning to train on top of the PyTorch model pretrained on imagenet dataset. The classifier of the pretrained model has to be modified to change the number of outputs. While the imagenet dataset is significantly different from the road dataset, we hope that the weights of the lower-level layer might still be transferable. We trained on pretrained AlexNet https://arxiv.org/abs/1404.5997, VGG16 https://arxiv.org/abs/1409.1556, Vit_l_32 https://arxiv.org/abs/2010.11929, and Convnext_large https://arxiv.org/abs/2201.03545. AlexNet and VGG16 have straightforward architectures similar to the CNN baseline used in RoadTagger but with less depth. Vit_l_32 and Convnext_large are fairly recent models with complex architecture that are the best performing models on imagenet available on PyTorch.

We settled on using non-pretrained VGG because the model is straightforward and performs fairly well on imagenet. Furthermore, the complex model doesn't perform better than the simple models. We also want to modify the architecture to incorporate non-image data such as the highway type and whether the road is one way to provide more informative features to the model. The VGG model we used was based on https://medium.com/@tioluwaniaremu/vgg-16-a-simple-implementation-using-pytorch-7850be4d14a1.

To make the model generalize better, we first tried data augmentation. We double the training data size and randomly rotate and flip the satellite image to create more data from the existing image. The augmented image is still valid because the number of

lanes of the road at the center of a satellite image is still the same when rotated or flipped. This doesn't help the model much so we didn't pursue it further.

To improve the model performance, we first tried to filter out roads that are shorter than 34 meters in hopes of acquiring more consistent roads which are far from intersections. Next, we created a new dataset with more zoomed-in images to train the model. Finally we include the numerical road conditions data mentioned in the Dataset Creation section. The first dataset we used had a zoom level of 17, similar to the zoom level used by ScRoadExtractor's training data. However, the images are too zoomed out and details of the lanes aren't clear. It was even hard for humans to evaluate the number of lanes at this zoom level. Therefore, we increased the zoom level to 20, similar to the zoom level used in RoadTagger, in order to increase the lane detail resolutions and reduce noises from other parts of the images.

Figure 2: Satellite image and Road centerline image at zoom level 17

To combat imbalanced data, we tried reducing the number of 2 lanes in the dataset to be about the same number as 4 lanes. Then we tried using WeightedRandomSampler to make the less common lanes be resampled for training more often so that all lanes are trained by the model with similar frequency. We also switched from CrossEntropyLoss to FocalLoss which addresses class imbalance by putting more focus on the hard, misclassified samples.

Running Lanes Model

The notebook Train_Lane_Extraction.ipynb trains a model to predict the number of lanes. Before running the whole notebook, change some of the variables under Configs and Constants. The dataset is read from the csv file defined in the variable data_path.

The csv dataset is expected to be output from the DataExtraction.ipynb notebook. base_satellite_image_path defines the path the satellite image folder is on, which can be empty if it is in the same directory. The best model with the highest validation accuracy will be saved at the location with the file name defined by the variable best_model_path. Similarly, checkpoint_path defined the training checkpoint save location. The checkpoint file saves the training state after each epoch so the training can resume from the checkpoint at the next rerun without retraining from beginning in case of an error. You can see the loss and accuracy plot of training and validation set at the end of the training. After training, you can also see samples of the satellite images, what the model predicts, and the true label. Finally it'll run the model against the test set and report the overall accuracy and the accuracy breakdown for each number of lanes.

The notebook Infer_Num_Lanes.ipynb contains the code to use our pre-trained model to make predictions on the dataset. Change some of the variables under Configs and Constants before running the whole notebook. The variables are similar to the ones in Train_Lane_Extraction.ipynb with the addition of output_csv. The predicted number of lanes will be appended to the csv file under the column predicted_lane_nums and the file should be the same file as the output in width detection.

Number of Lanes Results

Even after all the approaches we mentioned earlier, none of the models we trained generalized well to the test set. The best model achieved around 70%+ accuracy on the test set, which is similar to the performance in the RoadTagger paper. However, the results are misleading because the model seems to predict based heavily on the distribution of the training dataset, which happens to be the same in the test set. Most of the roads are 2 lanes. Generally, the models predicted 2 lanes well with almost 100% accuracy but fails to predict other lane counts, with around 0% to 50% accuracy. We guess that a similar situation happened in the RoadTagger. The complex pretrained model doesn't perform well and didn't show any sign of improvement after 1 epoch while the simpler model did show some signs of improvement as we continued to train. Additionally, data balancing didn't seem to help. Reducing the number of 2 lanes to be similar to 4 lanes results in the model predicting 2 and 4 lanes somewhat well, around 70%+ while it still fails to predict other lanes. The WeightedRandomSampler and FocalLoss results in the model performing worse overall. This is because even if the dataset was now somewhat balanced, the model sees fewer images and less variance of common lanes such as 2 lanes, and sees the same image of uncommon lanes such as 6 lanes multiple times, which doesn't help the model generalize as well. The performance of the 2 lanes dropped significantly while other lanes doesn't improve much.

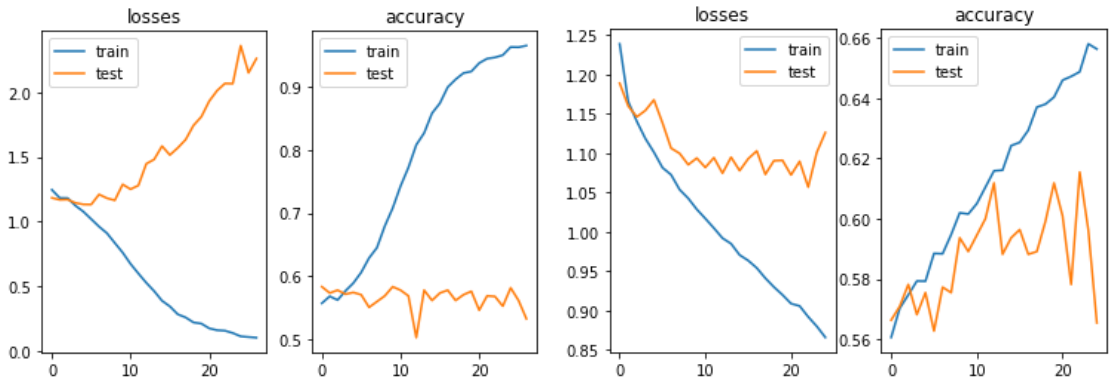

Figure 3: Pretrained Alexnet vs Pretrained Alexnet with data augmentation

# Lanes	Accuracy
1	42%
2	88%
3	23%
4	17%
5	16%
7	28%
others	0%

Table 3: Accuracy break down for Alexnet with data augmentation

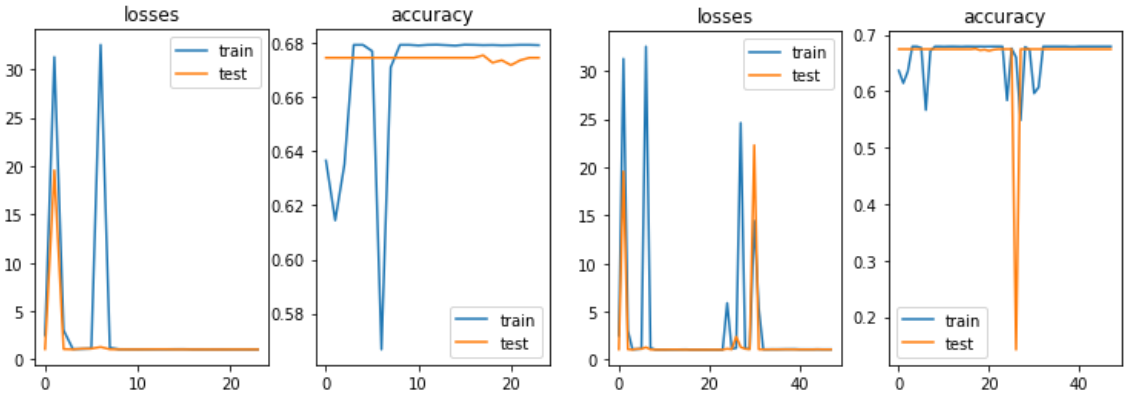

Figure 4: Vit_l_32 vs Convnext_large

# Lanes	Accuracy Vit_l_32	Accuracy Convnext_large
2	99%	100%
others	0%	0%

Table 4: Accuracy break down for Vit_l_32 and Convnext_large

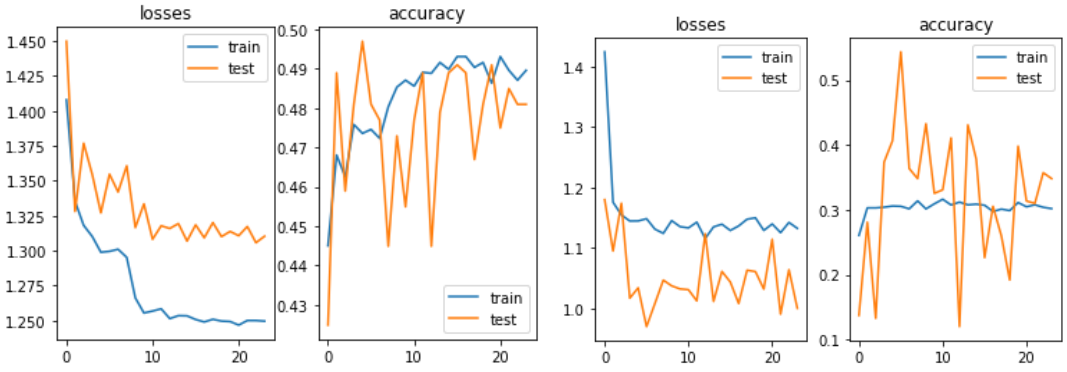

Figure 5: VGG16 reduced 2 lanes vs VGG16 weighted random sampler

# Lanes	VGG16 reduced 2 lanes	VGG16 weighted sample
1	29%	59%
2	72%	77%
3	17%	10%
4	75%	0%
5	0%	4%
8	0%	50%
others	0%	0%

Table 5: Accuracy break down for VGG16 with reduced lane and weighted sample

The general poor performance is likely due to the imbalanced dataset. Even with WeightedRandomSmpler, there's still not enough variability in the data for the model to generalize well. Data augmentation with WeightedRandomSmpler might help the model generalize more. Also, there are just not that many roads with greater than 4 lanes so grouping all of them together as a greater than 4 group might help with the imbalance. Creating a balanced dataset with an equal number of 1, 2, 3, 4, and 4+ lanes should

help the model to learn to better predict all the lanes. A totally different approach might be worth exploring such as detecting lane markers, but it would limit the use case just to the ones with lane markings. Additionally, using attention units, or masking the road which we want to classify using ScRoadExtractor might improve performance. Unfortunately, most of the existing lane detection research is focused on car camera images with clear road markings instead of satellite images, so we couldn't find much reference.

Road Width Methods

One of the most important parts of our project was automatically extracting the width of the randomly sampled roads. There are various approaches one could take to this problem however many of them have major issues, or barriers. For instance most basic machine learning approaches will require a large dataset labeled with road widths for direct road width extraction, which unfortunately a reliable one does not exist. Creating a dataset like this at the scale we would need to effectively train a model is nearly impossible since at least 6,000-10,000 roads would have to be tediously measured by hand. Hence why there is very little research into this subject. Because of this limitation we had to choose a method which could either be fully unsupervised or semi-supervised in a way in which we could automatically generate labels. Enter ScRoadExtractor (Wei): a machine learning method from Wuhan university which utilizes a semi-supervised approach to automatically segment and mask roads. Rather than using road widths, or road masks as the data labels, ScRoadExtractor utilizes road centerlines, as seen in the right image in figures 2, to loosely mark the center of the road to be masked. The use of centerlines allows us to overcome the dataset creation hurdle since accurate centerlines exist for a multitude of cities, and training data can be created in a scalable way. It should be noted that ScRoadExtractor only uses centerlines to train, and only needs the raw satellite images to make a measurement of road widths from other cities.

We heavily based our implementation on the original ScRoadExtractor implementation, however we made some major changes to increase the performance and to utilize the model to measure road widths rather than just segmenting the road from the surroundings. The reader can assume that any part of the ScRoadExtractor model not referenced in the following sections functions identically as described in the original ScRoadExtractor paper (Wei).

The first major change we made was in the training data augmentation phase which takes the centerline and satellite image and generates data which the model can be trained on. One of the key steps in this phase is the generation of rough proposal masks as seen in Figure 6, which marks the smallest possible width in white and the widest possible width in gray. It is important that these masks are as close to the theoretical

actual mask as possible, although occasional errors are not devastating to performance. Typically these masks are generated using the ground sampling distance and a 95% confidence interval for the maximum and minimum road widths for a city. We modified this step by instead creating a 95% confidence interval for the width of an individual lane and multiplying this interval by the number of lanes for each image in the OSM data. This change significantly reduced the error in our implementation, due to our high zoom data being less tolerant to variations in the proposal mask than the original paper's lower zoom data.

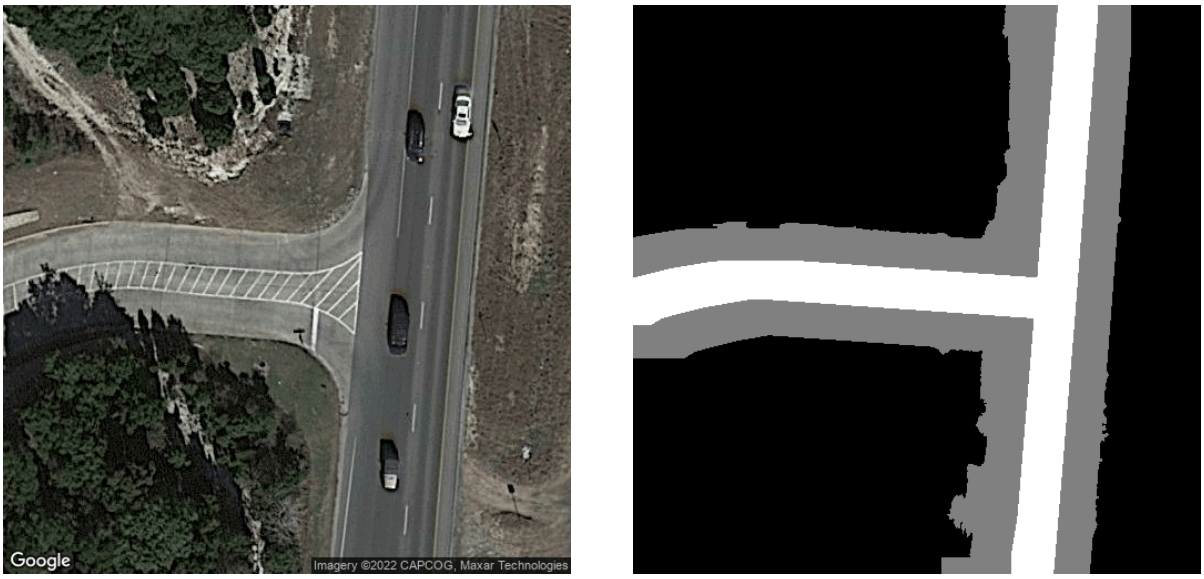

Figure 6: Satellite image and its rough proposal mask for ScRoadExtractor

The second change we made was to preload the data into memory to significantly decrease the training time of the model such that we could train it in a reasonable time. To do this we wrote a custom PyTorch dataloader and dataset class which preloads the dataset and generates a memory mapped pickle file which allows the dataset to be quickly loaded into memory in subsequent runs without having to do a full preload from the hard storage. This change cut down on training time by a factor of 5 and made the training of the model feasible on our lesser hardware.

The final change to the method did not modify any part of the original ScRoadExtractor implementation, but instead added a final step to the pipeline which extracted the road width in meters using the ground sampling distance. To do so, we first calculated a line perpendicular to the center line halfway up the road. Using this line we then simply iterate over the pixels starting from the center of the road in both directions. We then

note the pixels at which the road mask ends and calculate the distance between the points in meters using the ground sampling distance.

Running Road Width

We were unable to include the pretrained model in our repository, thus one must manually download it and place it in the ScRoadExtractor/Weights/ folder before running. The pretrained model was trained utilizing 10 major American cities and performed very well in other western and modern cities. To run our code, start by cloning the road-network-inference repo into the top level of your google drive or environment. To use our model to automatically measure road width data, start by creating a new dataset for the desired cities (or utilizing our pre-generated dataset). Once you have this, simply run the cells in the file Infer_Road_Width.ipynb in order. Note that at the top of the notebook you may need to update the paths to various folders which contain the data.

To retrain our model after generating a new dataset, run the cells in order in the Train_width_extraction.ipynb file. The first few cells in the notebook are responsible for augmenting the input data to train the model. Similar to running the inference, you may need to update some file paths. Also, keep in mind that if it is the first time running the data loader cell you will need to restart the notebook, and run all cells again before training. Otherwise, the memory will overflow since Colab does not dump the excess memory used to write the initial pickle file unless it is restarted.

Road Width Results

Measuring the actual performance of our road width methods proved to be somewhat difficult, and ambiguous. To analyze the model outputs we need a gold label road measurement. The first ambiguity comes from the error involved in measuring the actual road width. To create the gold label we utilized the Google maps measure function, which is accurate up to about 10 centimeters or 0.1 meters, thus giving us a standard error. We verified this by manually measuring a few roads in Ithaca on google maps and in real life using tape measures, and comparing them to make sure they agreed, which they did. Unfortunately, there is also ambiguity in defining exactly what the boundaries of some roads are, something which the model also struggles with highly ambiguous roads. Thus, we report our results separately for normal, well-defined roads, and ambiguous roads. Although our dataset creation attempts to minimize ambiguous roads as much as possible, our pipeline cannot yet fully eliminate them. Our results are reported in Table 1 as the mean absolute error of 50 randomly sampled roads (43 normal, 7 ambiguous). We ensured that the mean number of lanes in our samples was 3 lanes through weighted random sampling. Additionally, the images in Figure 7 are two examples of our model outputs with the road mask overlaying the original satellite image in red, and the measurement points indicated by the center of the yellow circles.

	Mean Absolute Error	MAE/Lane
Normal Roads	0.4m ± 0.1m	0.13m ± 0.03m
Ambiguous Roads	2.6m ± 0.1m	0.86m ± 0.03m

Table 6: Results From Road Width Extraction

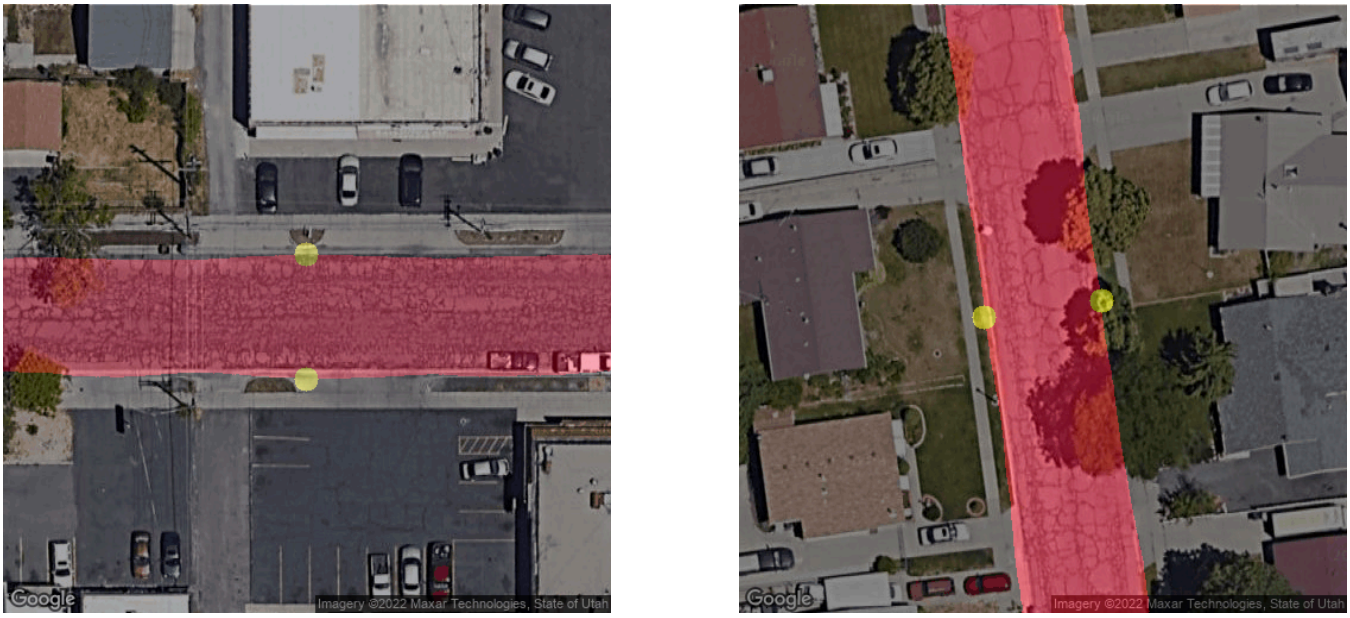

Figure 7: Selected Normal Road Test Examples From Road Width Extraction

Figure 8: Examples of poorly measured ambiguous roads

Overall, for normal roads, the MAE was extremely good, and subjectively the overall road masks and the measurement points are very precise. Rarely did the model mispredict a normal-looking road. However, the model did tend to overestimate the width of normal roads, likely due to the inclusion of very wide ambiguous roads in the dataset. On the other hand, as shown in figure 8, some ambiguous roads were wildly off causing the MAE to be somewhat high and subjectively un-precise. The error was not as high as expected, however, this is mainly due to the large number of ambiguous roads with connected sidewalks and parking spaces which the model tended to over-predict by their typical width of 1-2m. Keep in mind that the two examples in Figure 8 are only minorly ambiguous roads and other roads in the dataset are significantly more ambiguous, although those roads should be removed under ideal conditions.

Conclusion

Overall our project effectively lays the groundwork for creating a fully functional automatic road sampling system. The dataset creation and road width estimation portions of our project work extremely well, and will immediately allow users to collect road width information as well as generate datasets for further training and other research. The number of lanes section, although not perfect, still allows the user to sample road data at a middling level of accuracy either directly from OSM or using our automated method. However, the section also lays out a solid framework for future improvements to be made. Road width extraction from satellite images has never been directly performed in such a manner before, and the number of lanes extraction from satellite images has never been performed and achieved a significantly better result than what we have achieved. Future improvements on this project could include the use of attention-based lane number extraction methods, as well as automatic elimination of ambiguous roads either through the use of simple statistical methods or machine learning methods. Also, a more balanced dataset, different models, or different approaches are worth further investigating. Our methods should allow future research into road networks such as that in Smart and Healthy Cities and others to obtain meaningful results.

Bibliography

Aremu, Toluwani. “VGG-16: A Simple Implementation Using Pytorch.” Medium, Medium, 30 July 2021, https://medium.com/@tioluwaniaremu/vgg-16-a-simple-implementation-using-pytorch-7850be4d14a1.

Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. Computers, Environment and Urban Systems 65, 126-139. doi:10.1016/j.compenvurbsys.2017.05.004

Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." arXiv preprint arXiv:2010.11929 (2020).

He, Songtao, et al. "RoadTagger: Robust road attribute inference with graph neural networks." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. No. 07. 2020.

Krizhevsky, Alex. "One weird trick for parallelizing convolutional neural networks." arXiv preprint arXiv:1404.5997 (2014).

Liu, Zhuang, et al. "A ConvNet for the 2020s." arXiv preprint arXiv:2201.03545 (2022).

“Smart and Healthy Cities.” Frontiers, https://www.frontiersin.org/research-topics/23056/smart-and-healthy-cities#overview.

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

Wei, Yao, and Shunping Ji. "Scribble-based weakly supervised deep learning for road surface extraction from remote sensing images." IEEE Transactions on Geoscience and Remote Sensing 60 (2021): 1-12.