Lyft 3D Object Detection for Autonomous Vehicles

Beuth University of Applied Sciences

Chandra Vamshi Dasyam(885931)

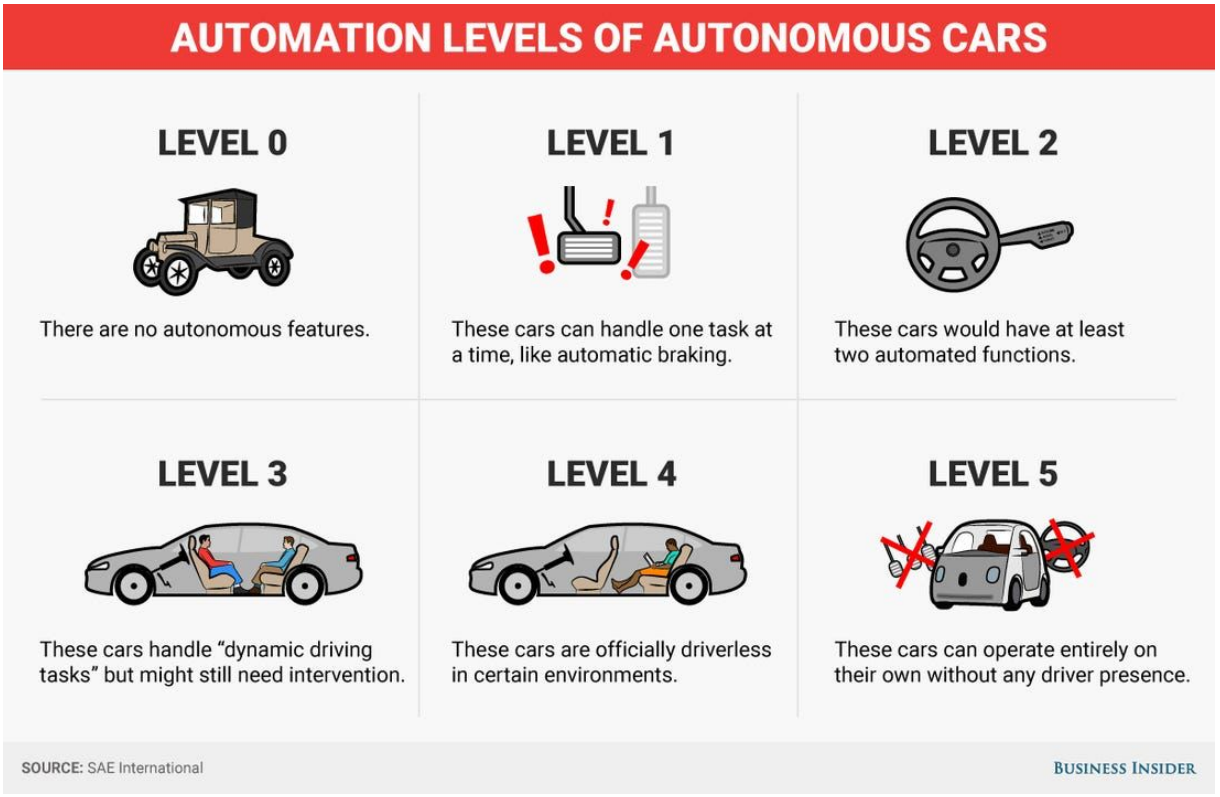
Muhammad Ahmad Falak (886100)

Abstract

With the rapid evolution in the world of technology, it becomes increasingly important for the automotive sector to keep up with the fast pace, just like any other industry. In recent times, self-driving cars have gained a lot of traction but there is a huge gap in expectation and the current state. On those lines, our project focuses on 3D Object Detection of Lyft's autonomous vehicles

Why we don't have an autonomous car yet?

Self Driving Vehicles are one of the most hyped technologies of the modern decade. Even though many companies may brand their driver assistance technology as "Autopilot" a truly self-driving vehicle, without a human driver on open roads, has not yet become a reality.



Source: http://post.toutptit-toutbio.com/page-3/self-driving-car-timeline-28811.html

Currently, vehicles that are labeled as "Autonomous" are Level 4 vehicles. There are 5 levels of self-driving cars:

1. The lowest level is Level 0, in which a vehicle has no driver assistance technology. It also constitutes a majority of vehicles on US roads.

2. Level 1 is the level in which the vehicle is equipped with driver assistance technology such as a blind spot monitoring system (BLIS), adaptive cruise control (ACC), and automatic emergency braking (AEB). It comprises of a majority of new cars sold, and has active or inactive systems but only intervene when they detect another vehicle.

3. Level 2 vehicles can control steering, acceleration, and braking and include systems such as GM's Supercruise or Tesla's Autopilot system.

4. Level 3 systems such as Audi Traffic Jam Assist are completely autonomous below 37 mph.

5. Level 4 vehicles are the current vehicles being tested around the world including Lyft, Uber, and Waymo vehicles. These vehicles are autonomous but must have a human driver available for difficult driving situations.

6. Level 5 comprises of completely autonomous vehicles with no human intervention, of which the most well known is the Google car, unveiled a few years ago. However, even this car is only able to operate on pre-defined routes and hence, can't be termed truly autonomous.

Given that there are many levels to evaluate self-driving vehicles, it leads us to wonder why there are no true self-driving vehicles on sale now at your local dealership (or online if you are Tesla). This is because of two main technological issues:

1. **Perception**: Identification and classification of objects around the vehicle

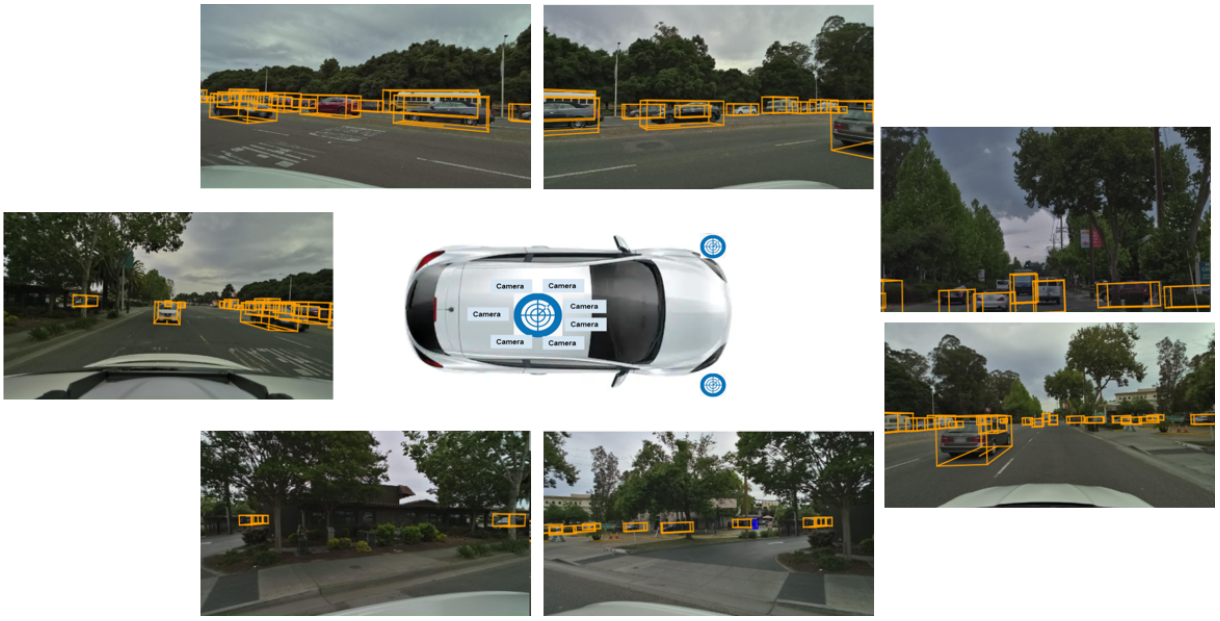
2. **Prediction:** Determination of the future position of predicted objects

Since perception was the cornerstone issue behind the slowed development of self-driving vehicles, we decided to focus our effort to improve perception around a vehicle.

Lyft Dataset

The Lyft dataset from the active Kaggle competition was a total of 85 GB. The data was split between testing and training sets and included a sample submission. The dataset gives a 3D point cloud and camera data from the Lyft test vehicles.

Our data was captured by 10 host cars on the roads of Palo Alto, California. Each of the host cars has seven cameras and one LiDAR sensor on the roof, and 2 smaller sensors underneath the headlights of the vehicle.



Each LiDAR sensor will shoot lasers 360 degrees to detect objects and get 3D spatial geometric information. These LiDAR sensors produce a point cloud each with 216,000 points at 10Hz. The data was captured across 13 files which served as inputs to our model. Let us see what the data comprised of, in detail:

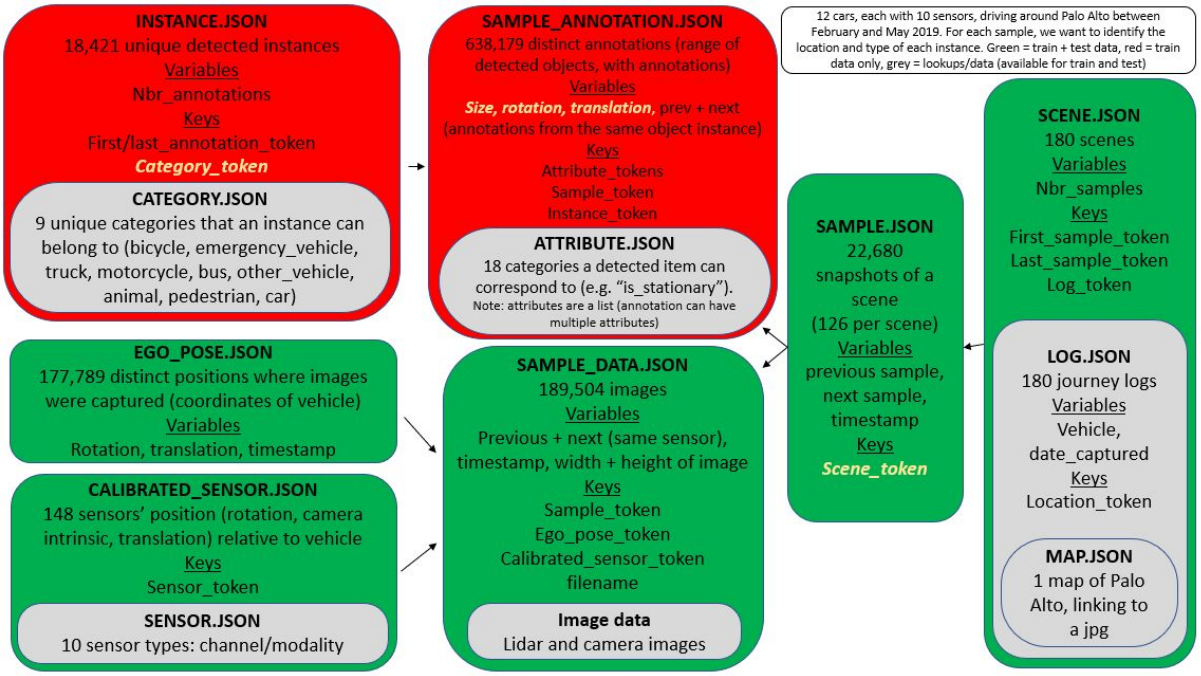
- **Sample_data** is the data collected from a particular sensor in the car.
- **Sample_annotation** is an annotated instance of an object of interest.
- An **instance** is an enumeration of all object instances we observed, such as a labeled truck on the image.
- **Categories** are different types of objects observed. Examples include cars, pedestrians, etc.,
- **Ego_pose** is the record of the vehicle at a certain point in time which could be mapped for each sample.

- **Map data** is a binary semantic map of the roads around the vehicle

- **Calibrated_sensor** is the definition of a particular sensor as calibrated on a particular vehicle. It provides information about the environment, such as the distance and bearing to a feature, or directly measure the sensor's position and orientation (pose). The intrinsic properties carried by calibrated_sensor are those that do not depend on the outside world and how the sensor is placed in it. The property and distance matrix will be used to combine three-point clouds produced at similar timestamp and map 3D bounding boxes.

- An **attribute** is a property of an instance that can change while the category remains the same.

All of these features along with the lidar and image data were split into training and testing sets and made up the large dataset for the competition.



The typical use of convolutional networks is on classification tasks, where the output to an image is a single class label. However, in many visual tasks, especially in Autonomous Vehicle image processing, the desired output should include localization, i.e., a class label is supposed to be assigned to each pixel.

Sementic segmentation

Image segmentation is a computer vision task in which we label specific regions of an image according to what's being shown.

"What's in this image, and where in the image is it located?"

Representing the task

Simply, our goal is to take a image and output a segmentation map where each pixel contains a class label represented as an integer.

Source : https://www.jeremyjordan.me/semantic-segmentation/#representing

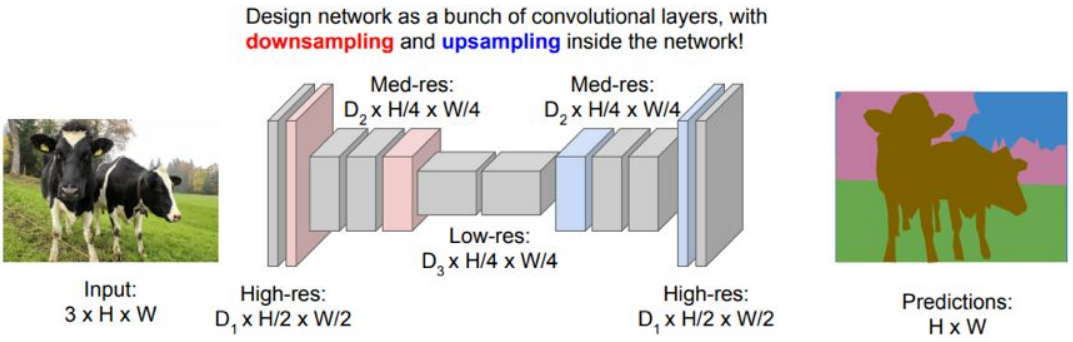
We can easily inspect a target by overlaying it onto the observation.

Source : https://www.jeremyjordan.me/semantic-segmentation/#representing

When we overlay a *single channel* of our target (or prediction), we refer to this as a **mask** which illuminates the regions of an image where a specific class is present.

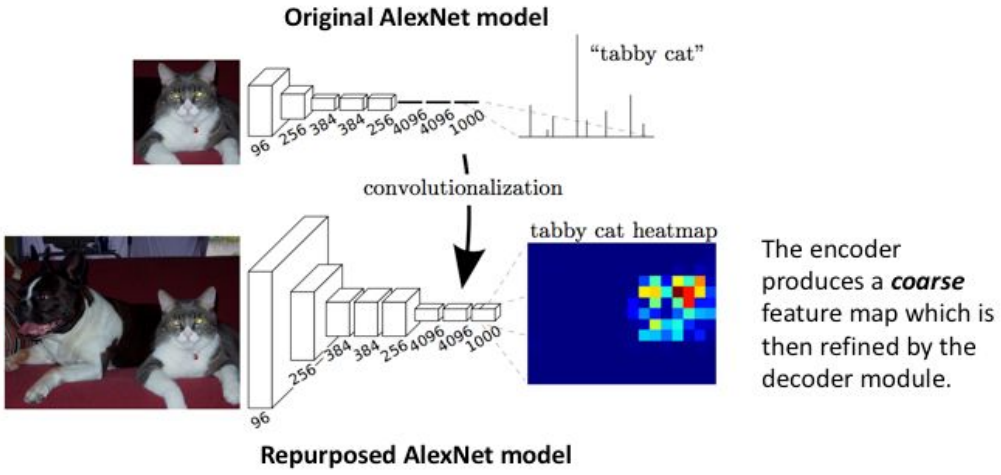
Constructing an architecture

One popular approach for image segmentation models is to follow an **encoder/decoder structure** where we *downsample* the spatial resolution of the input, developing lower-resolution feature mappings which are learned to be highly efficient at discriminating between classes, and the *upsample* the feature representations into a full-resolution segmentation map.



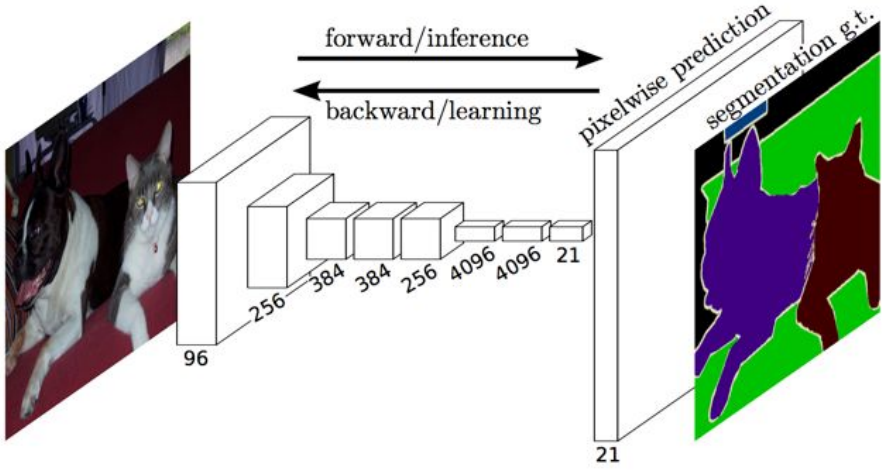
Source : http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf

The approach of using a "fully convolutional" network trained end-to-end, pixels-to-pixels for the task of image segmentation was introduced by Long et al. in late 2014. The paper's authors propose adapting existing, well-studied *image classification* networks (eg. AlexNet) to serve as the encoder module of the network, appending a decoder module with transpose convolutional layers to upsample the coarse feature maps into a full-resolution segmentation map.



Source : https://arxiv.org/abs/1411.4038

The full network, as shown below, is trained according to a pixel-wise cross entropy loss.



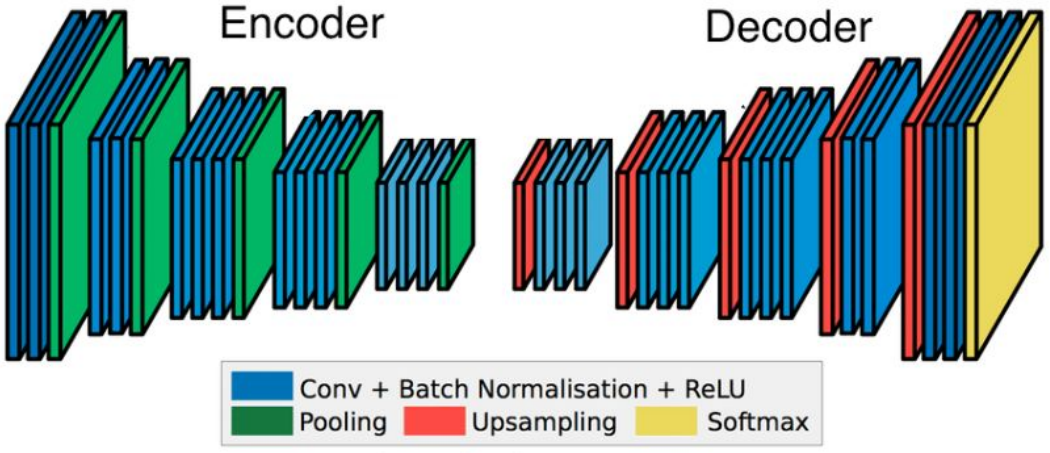
Source: https://arxiv.org/abs/1411.4038

Understading the UNET-Model

For the task of semantic segmentation, we need to retain the spatial information, hence no fully connected layers are used. That's why they are called **fully convolutional networks**. The convolutional layers coupled with downsampling layers produce a low-resolution tensor containing the high-level information.

Taking the low-resolution spatial tensor, which contains high-level information, we have to produce high-resolution segmentation outputs. To do that we add more convolution layers coupled with upsampling layers which increase the size of the spatial tensor. As we increase the resolution, we decrease the number of channels as we are getting back to the low-level information.

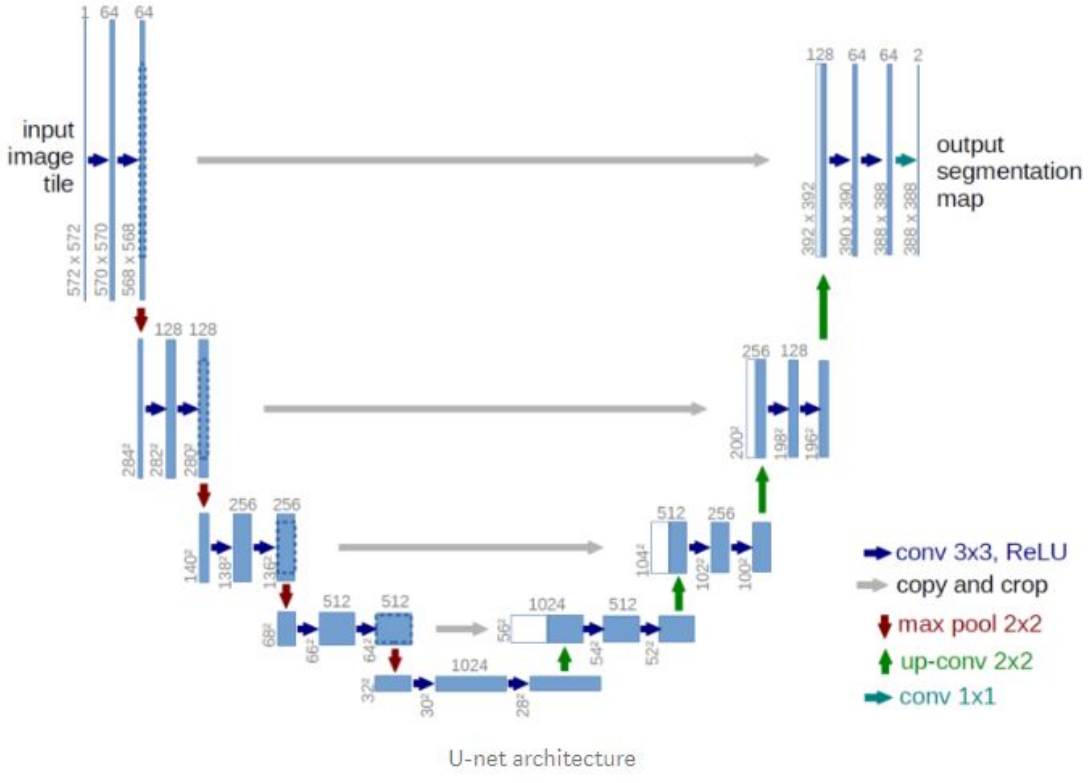
This is called an **encoder-decoder** structure. Where the layers which downsample the input are the part of the encoder and the layers which upsample are part of the decoder.



Source :
https://www.semanticscholar.org/paper/SSeg-LSTM%3A-Semantic-Scene-Segmentation-for-Syed-Morris/b5e4abb8c1bebf7d531202e57e431c638dd1c4af

When the model is trained for the task of semantic segmentation, the encoder outputs a tensor containing information about the objects, and its shape and size. The decoder takes this information and produces the segmentation maps

The architecture looks like a ‘U’ which justifies its name



U-net architecture

Source: https://arxiv.org/pdf/1505.04597.pdf

This architecture consists of three sections:

1. **The contraction**

   The contraction section is made of many contraction blocks. Each block takes an input and applies two 3X3 convolution layers followed by a 2X2 max pooling. The number of kernels or feature maps after each block doubles so that architecture can learn the complex structures effectively.

2. **The bottleneck**

   The bottom-most layer lies between the contraction layer and the expansion layer

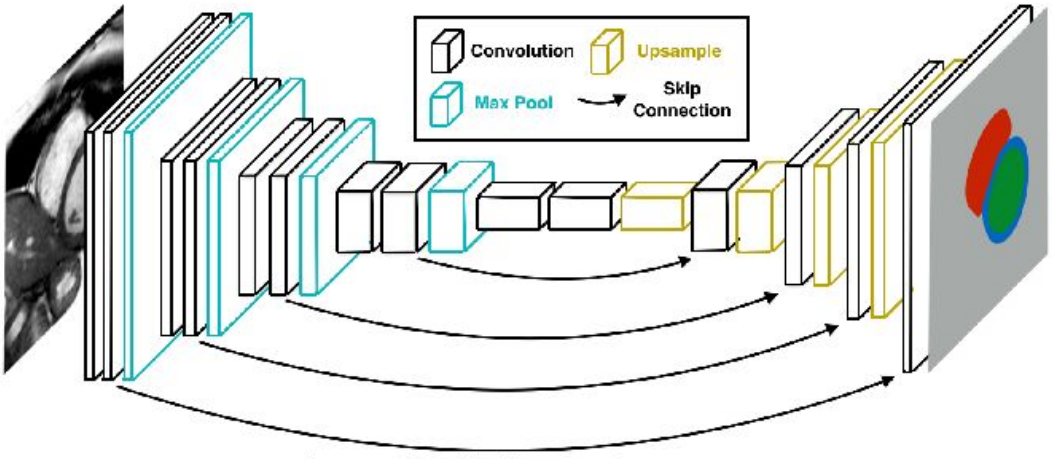
3. **Expansion section**

   But the heart of this architecture lies in the expansion section. Similar to contraction layer, it also consists of several expansion blocks. Each block passes the input to two 3X3 CNN layers followed by a 2X2 up-sampling layer. Also after each block number of feature maps used by convolutional layer get half to maintain symmetry. The number of expansion blocks is as same as the number of contraction block. After that the resultant mapping passes through another 3X3 CNN layer with the number of feature maps equal to the number of segments desired.

Skip Connection

If we simply stack the encoder and decoder layers, there could be loss of low-level information. Hence, the boundaries in segmentation maps produced by the decoder could be inaccurate.

To make up for the information lost, we let the decoder access the low-level features produced by the encoder layers. That is accomplished by **skip connections**. Intermediate outputs of the encoder are added/concatenated with the inputs to the intermediate layers of the decoder at appropriate positions



Encoder-Decoder with skip connections Image source

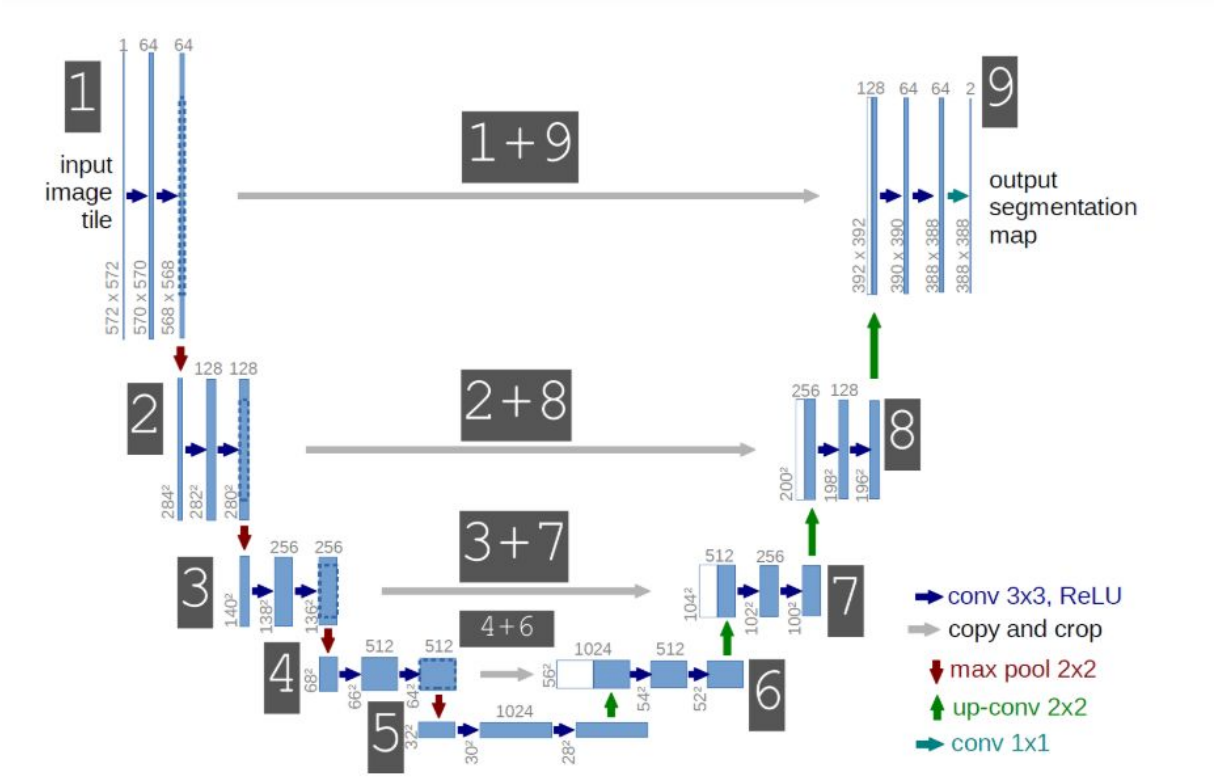
Source:
https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html

The skip connections from the earlier layers provide the necessary information to the decoder layers which is required for creating accurate boundaries.To understand the right part of the architecture, which is a decoder we must look at the whole architecture because a very important step called as concatenation happens in the decoder part of U-Net, where feature maps from the encoder are concatenated to the feature map of the decoder side for better learning. With the term better learning, I mean to say learning better contextual features. In image processing, the term contextual information means the relationship between or among two or more pixels in an image. As we all know that the initial layers of any convolution network are responsible for learning the features from the initial layer and carrying those features can help us preserve edge and many features in the final output.

In the image below, you can see that output from block 1 is concatenated to the input of block 9. In the same way, the output from block 2 is concatenated to the input of block 8, and also 3 and 7, 4 and 6. These skip connections from earlier layers in the network (prior to a downsampling operation) should provide the necessary detail in order to reconstruct accurate shapes for segmentation boundaries. Indeed, we can recover more fine-grain detail with the addition of these skip connections.

If you start looking at the right part of the network from the bottom. You will find a green upside arrow. That arrow stands for Transpose convolution. Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution. Achieve output of similar size as input.



Source :
https://medium.com/@pallawi.ds/understand-semantic-segmentation-with-the-fully-convolutional-network-u-net-step-by-step-9d287b12c852

- One very important aspect of this architecture is the fact that the upsampling path *does not* have a skip. The authors note that because the "upsampling path *increases* the feature maps spatial resolution, the linear growth in the number of features would be too memory demanding." Thus, only the *output* of a dense block is passed along in the decoder module.

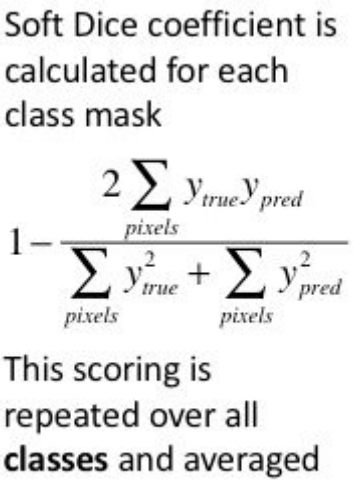
Loss Function

Dice coefficient performs better at class imbalanced problems.Loss function for image segmentation tasks is based on the Dice coefficient, which is essentially a measure of overlap between two samples. This measure ranges from 0 to 1 where a Dice coefficient of 1 denotes perfect and complete overlap. The Dice coefficient was originally developed for binary data, and can be calculated as:

$$Dice = 2|A \cap B| \ / \ |A| + |B|$$

where $|A \cap B|$ represents the common elements between sets A and B, and $|A||$ represents the number of elements in set A (and likewise for set B).
For the case of evaluating a Dice coefficient on predicted segmentation masks, we can approximate $|A \cap B|$ as the element-wise multiplication between the prediction and target mask, and then sum the resulting matrix.

In order to formulate a loss function which can be minimized, we'll simply use 1−Dice




```
def dice_loss(y_true, y_pred):
    smooth = 1.
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = y_true_f * y_pred_f
    score = (2. * K.sum(intersection) + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f)
+ smooth)
    return 1. – score
```

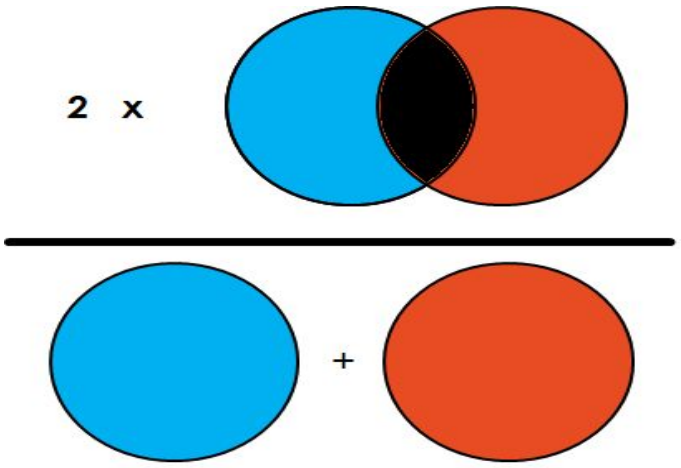
Optimizer

We have used Adam optimizer in my implementation. Adam is different from classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. In Adam, the learning rate is maintained for each network weight (parameter) and separately adapts as learning unfolds.

Adam is different from classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. In Adam, the learning rate is maintained for each network weight (parameter) and separately adapts as learning unfolds.

Metrices

Simply put, the Dice Coefficient is 2 * the Area of Overlap divided by the total number of pixels in both images



Source :
https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2

The Dice coefficient is very similar to the IoU. They are positively correlated, meaning if one says model A is better than model B at segmenting an image, then the other will say the same. Like the IoU, they both range from 0 to 1, with 1 signifying the greatest similarity between predicted and truth.

“””

```
def dice_coef(y_true, y_pred, smooth=1):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)
```

“””

Implementation

First join all lidars into one point cloud and apply annotations to make sure we're not missing anything.

Overview:

- we load lidar data (3d points), these points are in coordinate system of the lidar, rotated and translated relative to the car
- using sensor information of the lidar, we translate points from lidar coordinate frame to car coordinate frame, this allows us to merge data from all 3 lidars
- annotations are in global coordinates. We translate annotations into the car coordinates, which allows to have both the data and annotations in the same (car) coordinate frame

Before going to join all Lidars and converting into Car Coordinate System let's get a overview of Lidar point Cloud and Coordinate System

Lidar point Cloud

LiDAR systems send out pulses of light just outside the visible spectrum and register how long it takes each pulse to return. The direction and distance of whatever the pulse hits are recorded as a point of data. Different LiDAR units have different methods, but generally they sweep in a circle like a RADAR dish, while simultaneously moving the laser up and down.

Once the individual readings are processed and organised, the LiDAR data becomes point cloud data. The initial point clouds are large collections of 3D elevation points, which include x, y, and z along with its length, width, height and yaw.

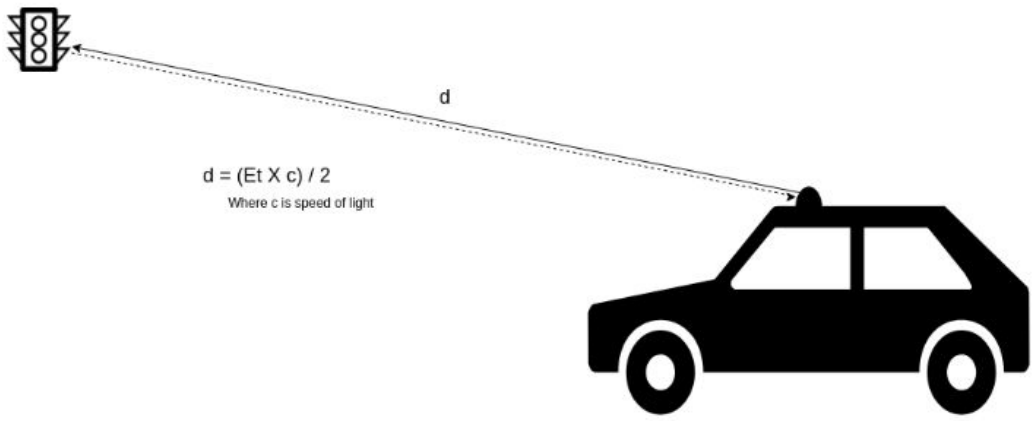
1. centre_x, centre_y**,** and centre_z correspond to the coordinates of an object's location on the XYZ plane.

2. yaw is the angle of the volume around the *z*-axis, making 'yaw' the direction the front of the vehicle/bounding box is pointing at while on the ground.

3. length, width, and height represent the bounding volume in which the object lies.

LiDAR Basics: The Coordinate System

Lidar is used to find the precise distance of objects in relation to us



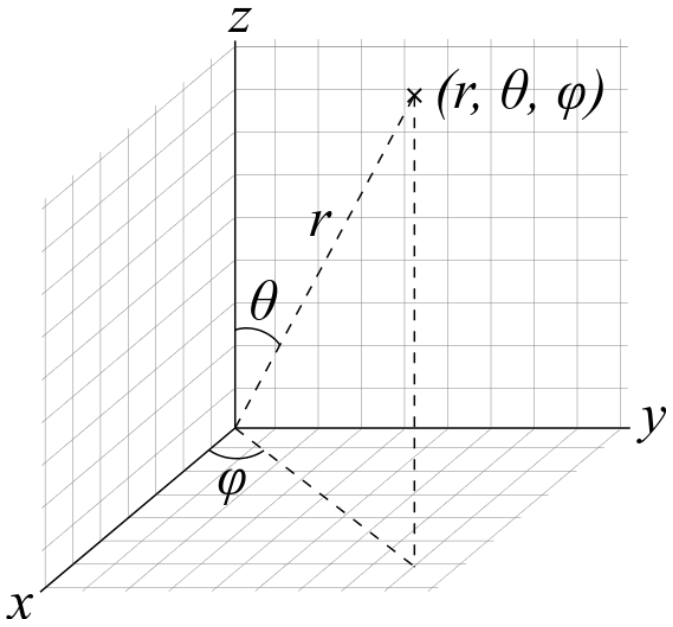
A LiDAR calculates the distance to an object using time of flight.

Source : https://hackernoon.com/lidar-basics-the-coordinate-system-a26529615df9.

As LiDAR sensor returns reading in spherical coordinates.

Spherical Coordinate System

In a spherical coordinate system, a point is defined by a distance and two angles. To represent the two angles we use azimuth(θ) and polar angle(ϕ) convention. Thus a point is defined by (r,θ,ϕ).

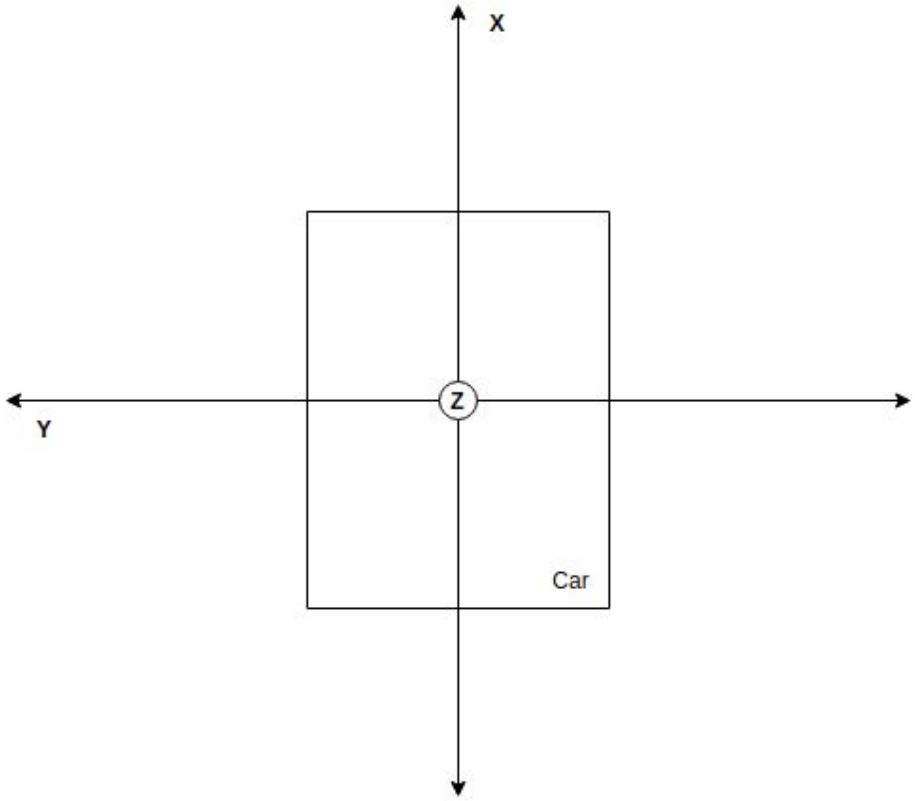


Source: https://hackernoon.com/lidar-basics-the-coordinate-system-a26529615df9

We can derive cartesian coordinates from spherical coordinates using below equations.

$$x = r \sin\theta \cos\varphi$$
$$y = r \sin\theta \sin\varphi$$
$$z = r \cos\theta$$

The Cartesian coordinate system is easy to manipulate and hence most of the time we need to convert spherical coordinates to a Cartesian system using the above equations.

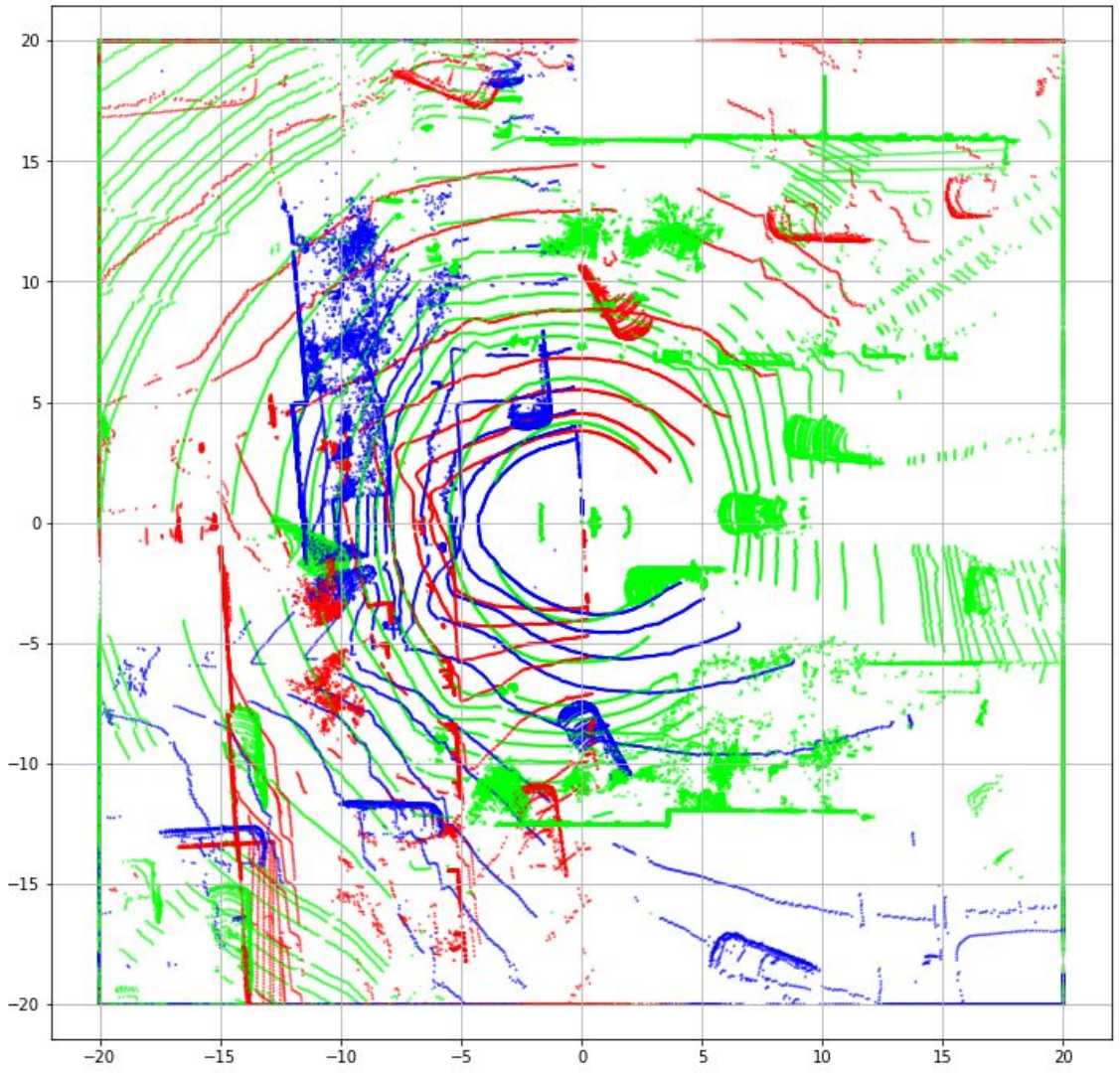


Source: https://hackernoon.com/lidar-basics-the-coordinate-system-a26529615df9

Above diagram shows the Cartesian coordinate system of a sensor mounted on a car.

Join Lidars and convert Coordinate System

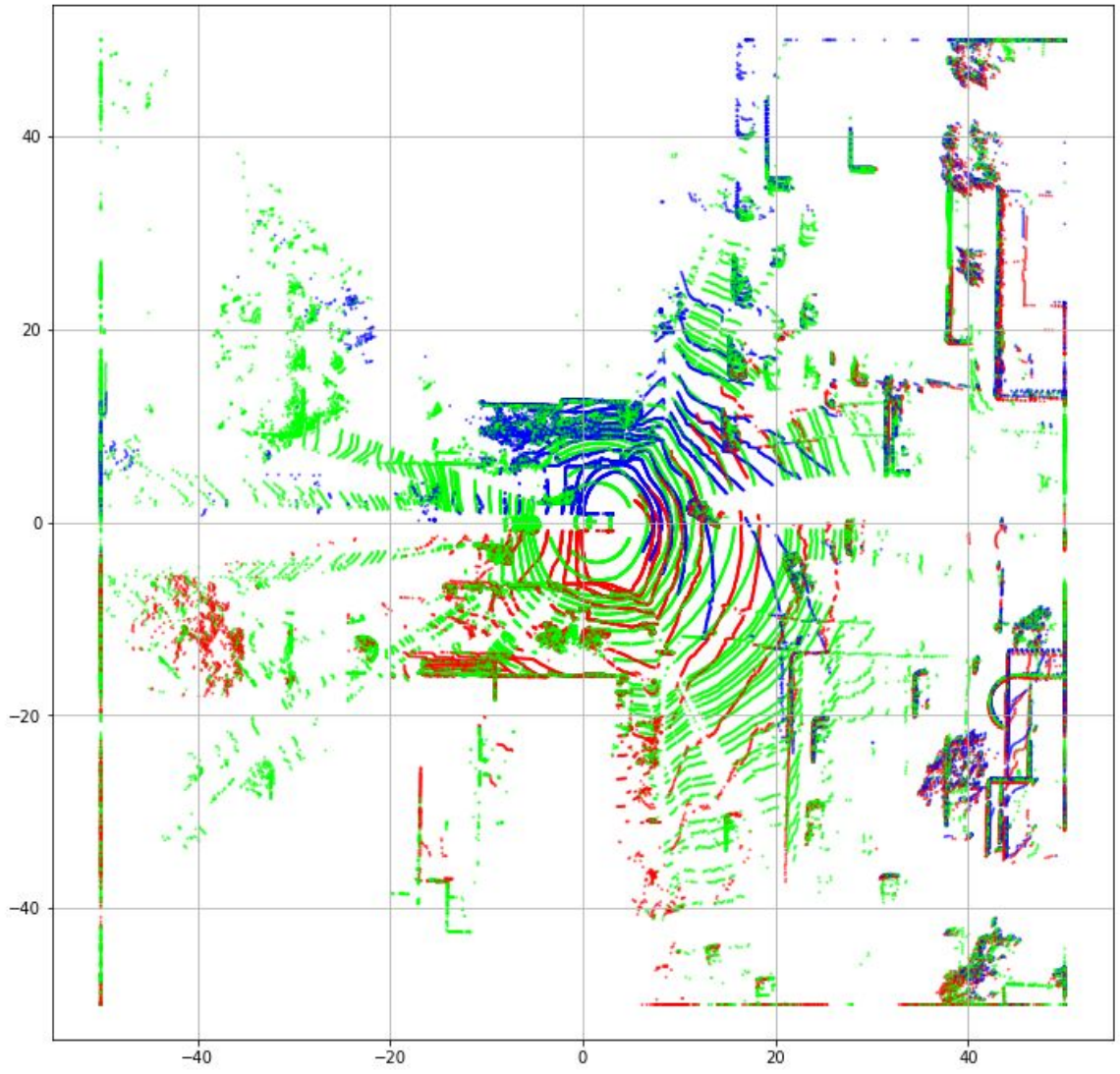
1. fetch lidar images related to the sample
2. load lidar's point data, we'll keep only first 3 columns (point coordinates)



Above lidar point look wrong, doesn't it? This is because each lidar is rotated and also slightly translated relative to the car, and lidar points are in lidar coordinates. Now let's enable translation to the car coordinates

1. translating all lidars into car coordinate system. For that we need Calibrated sensor info for each lidar

2. `transform_matrix(calibrated_sensor['translation'], Quaternion(calibrated_sensor['rotation']),`



We see that data from different lidars agrees, so all translations were right

load annotated data

Annotations are in local coordinate , we have to translate them first into global coordinate for that we'll use the ego_pose of the lidars

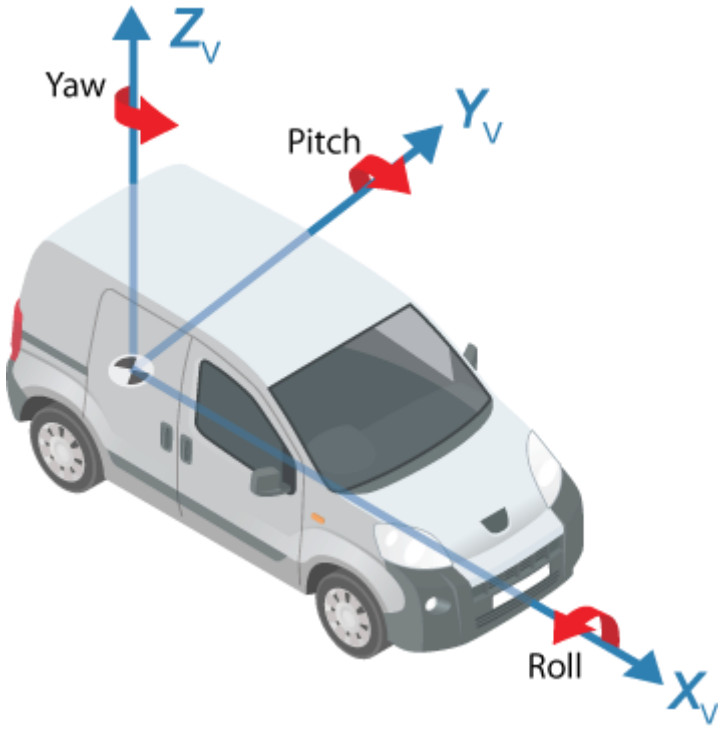
Before doing that first understand the car coordinate system and Global coordinate and why we have have to this translation?

Coordinate System in Autonomies Vehicle

The vehicle coordinate system (X_V, Y_V, $Z_{V)}$ used by Automated Driving is anchored to the ego vehicle. The term *ego vehicle* refers to the vehicle that contains the sensors that perceive the environment around the vehicle.

- The X_V axis points forward from the vehicle.

- The Y_V axis points to the left, as viewed when facing forward.
- The Z_V axis points up from the ground to maintain the right-handed coordinate system



Source: https://www.mathworks.com/help/driving/ug/coordinate-systems.html

Befor go further let's there are two concepts which are very important in Autonomous Vehicles

- Egocentric

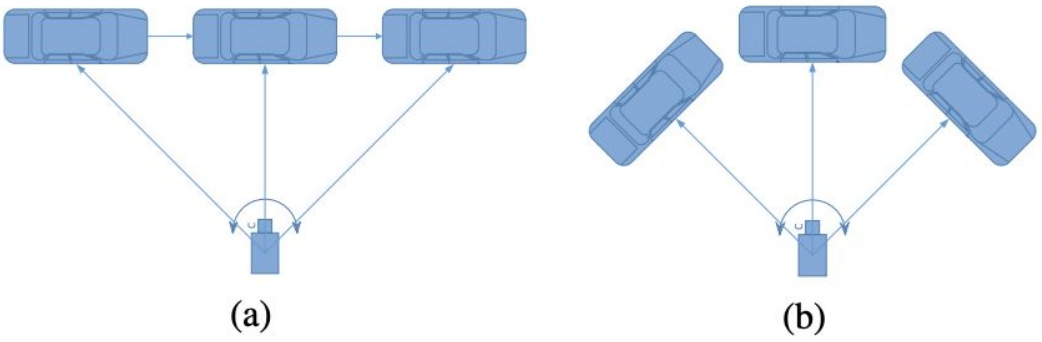
  egocentric orientation means orientation relative to camera.

  Egocentric orientation is sometimes referred to as **global orientation** of vehicles, as the reference frame is with respect to the camera coordinate system of the ego vehicle and does not change when the object of interest moves from one vehicle to another.

- Allocentric

  allocentric orientation is orientation relative to object (*i.e.*, vehicles other than the ego vehicle).

  Allocentric orientation is sometimes referred to as **local orientation** or **observation angle**, as the reference frame changes with the object of interest. For each of the object, there is one allocentric coordinate system, and one axis in the allocentric coordinate system aligns with the ray emitting from the camera to the object



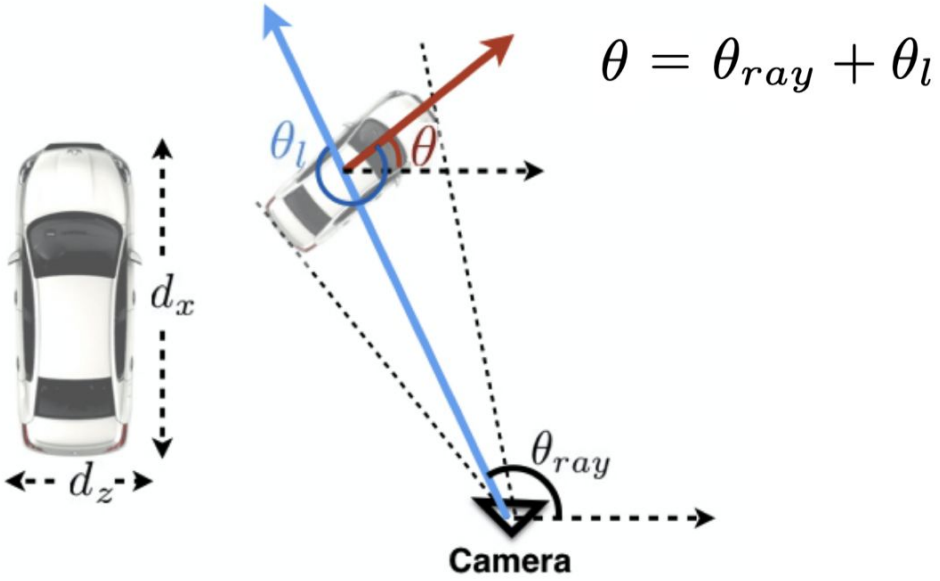
In (a), the global orientations of the car are all facing the right, but the local orientation and appearance will change when the car moves from the left to the right. In (b), the global orientations of the car differ, but both the local orientation in the camera coordinates and the appearance remain unchanged.

Source: https://arxiv.org/pdf/1612.00496.pdf

It is obvious to see that the appearance of an object in the monocular image only depends on the local orientation, and we can only regress the local orientation of the car based on the appearance.

Converting local to global yaw

To compute the global yaw using local yaw, we need to know the ray direction between the camera and the object, which can be calculated using the location of the object



Source: https://arxiv.org/pdf/1612.00496.pdf

The angle for the ray direction can be obtained using a keypoint from the bounding box position

Note that there are different choices for picking the keypoint.

In our case: projection of 3D bounding box on the image (can be obtained from lidar 3D bounding box ground truth)

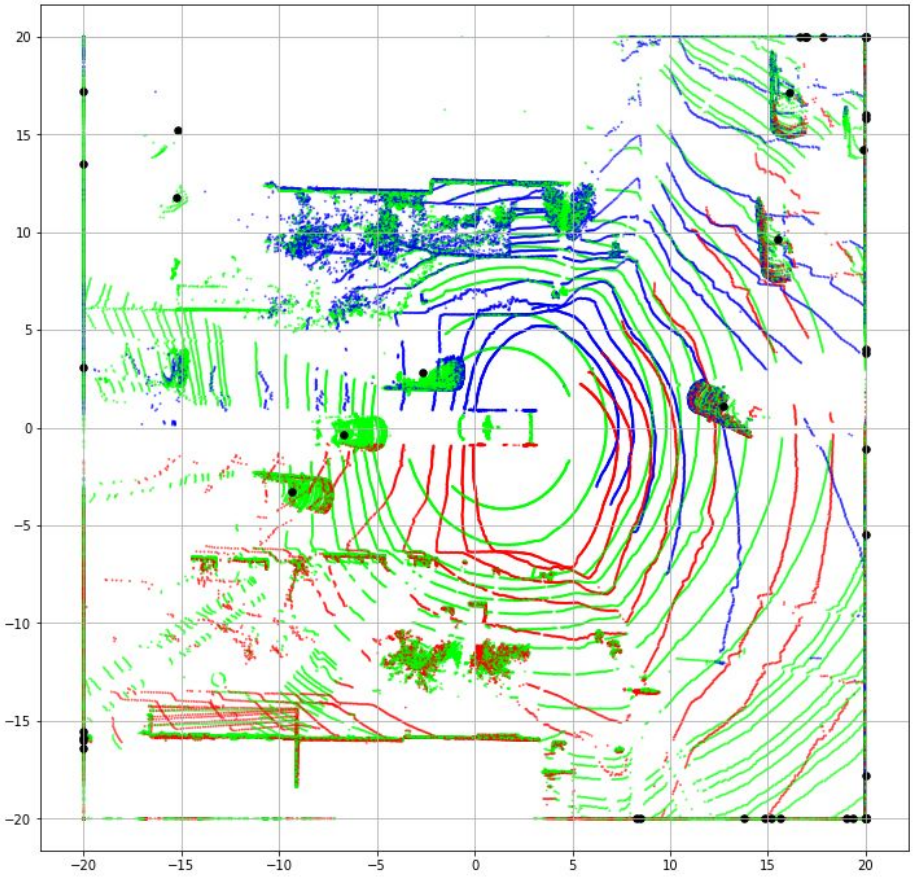
Takeaways

- It is possible to estimate the local (allocentric) orientation (yaw) from a local image patch.
- It is impossible to estimate the global (egocentric) orientation (yaw) from a local image patch.

After converting global coordinates into local coordinates we can **load annotated data**

Move boxes from world space to car space.

We will show only the car class

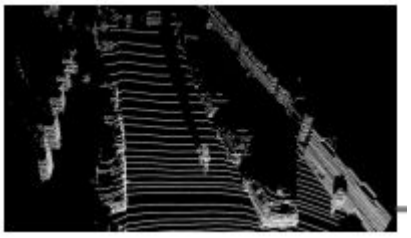


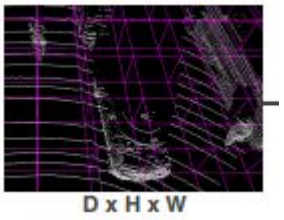
Voxels

divides a point cloud into equally spaced 3D voxels

voxel can exist it's a volume that's fills the spot in the three dimensional grid now.

Voxel Partition Given a point cloud, we subdivide the 3D space into equally spaced voxels Suppose the point cloud encompasses 3D space with range D, H, W along the Z, Y, X axes respectively. We define each voxel of size vD, vH, and vW accordingly. The resulting 3D voxel grid is of size $D' = D/vD$, $H' = H/vH$, $W' = W/vW$. Here, for simplicity, we assume D, H, W are a multiple of vD, vH, vW .





Source:
https://www.zpascal.net/cvpr2018/Zhou_VoxelNet_End-to-End_Learning_CVPR_2018_paper.pdf

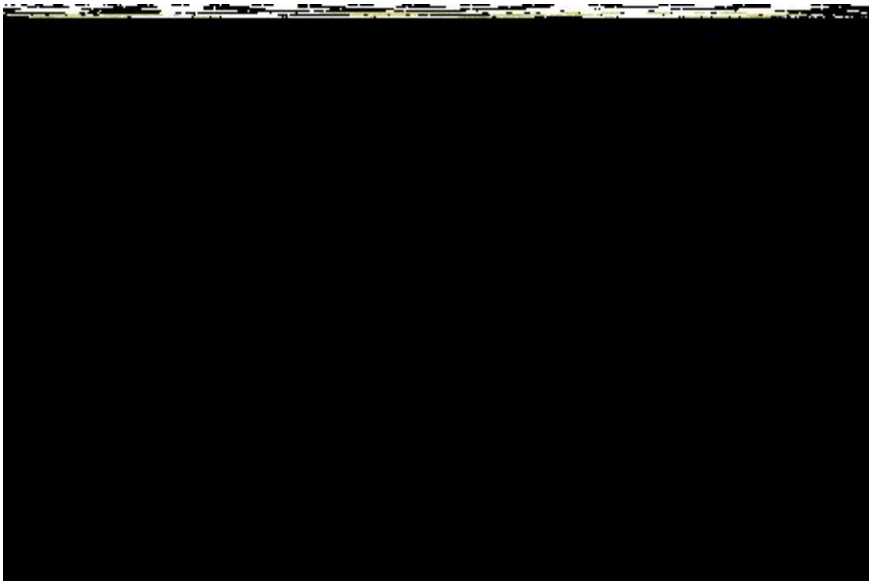
BirdEye View

we'll assemble the top-down view from a camera above the car. From this perspective we can immediately see the car's surroundings, which way the road goes, and the positions of other vehicles on the road.

the side cameras don't really cover the whole 360° angle around the car, therefore the model needs to infer the missing regions from those actually provided at input

A top-down view has a number of advantages:

- it condenses relevant information from four images into one;
- it can serve as a building block for localization/mapping algorithms;
- it can be used for path planning
- it allows for training a higher-level reinforcement learning agent that chooses an optimal path rather than atomic actions



Source:
https://medium.com/asap-report/from-semantic-segmentation-to-semantic-birds-eye-view-in-the-carla-simulator-1e636741af3f

Phase 1 of training

We have total 180 scenes from 12 hosts

We are using 140 for training and 40 for validation

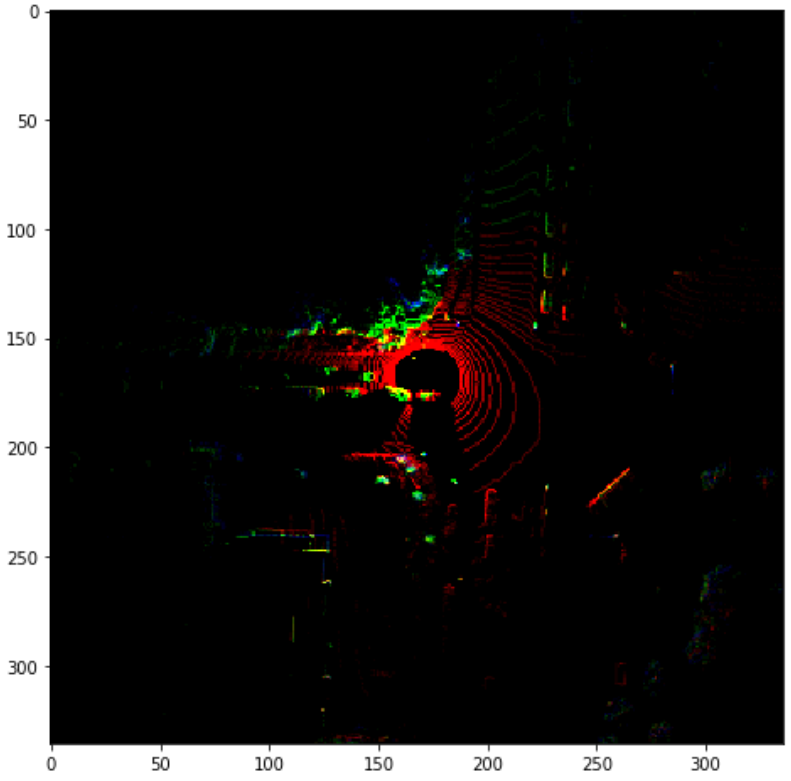
```
host-a005     1
host-a006     3
host-a007    26
host-a008     5
host-a009     9
host-a011    51
host-a012     2
host-a015     6
host-a017     3
host-a101    20
host-a102    12
Name: scene_token, dtype: int64
```

validation_hosts = ["host-a007", "host-a008", "host-a009"]

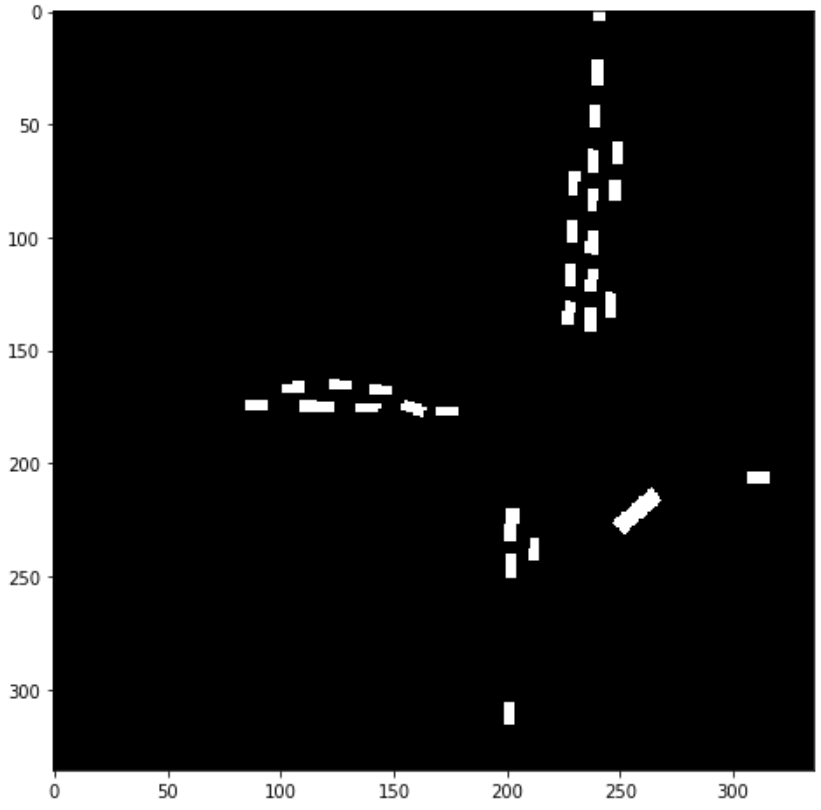
Below is an example of what the input for our network will look like. It's a top-down projection of the world around the car (the car faces to the right in the image). The height of the lidar points are separated into three bins, which visualized like this these are the RGB channels of the image

```
voxel_size = (0.4,0.4,1.5)
z_offset = -2.0
```

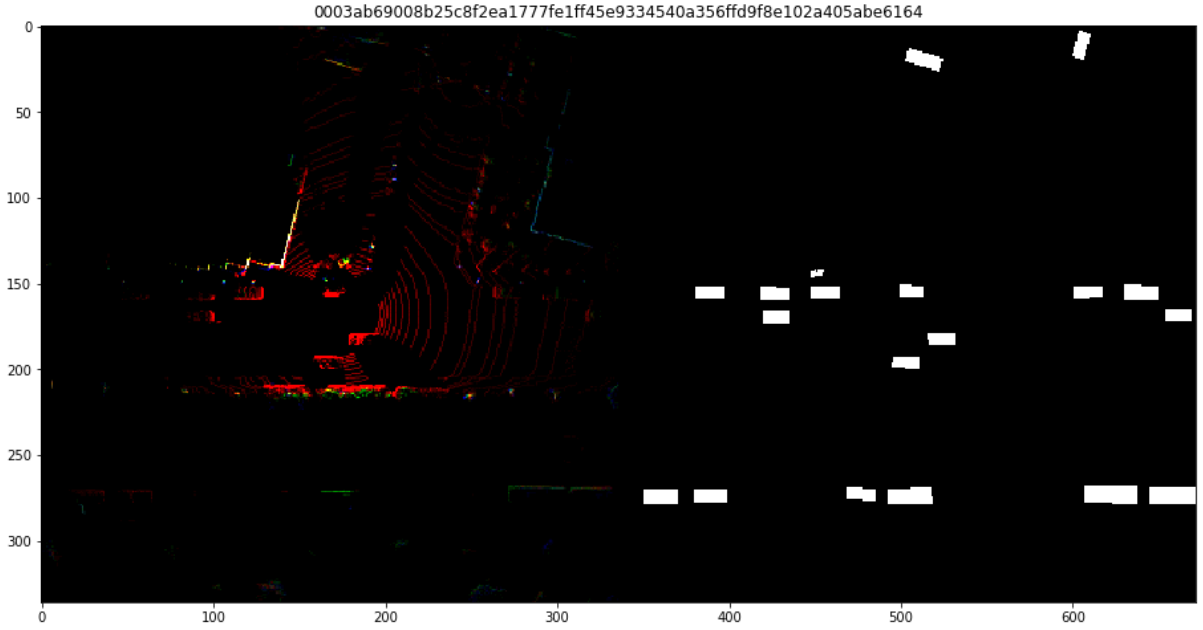
```
bev_shape = (336, 336, 3)
```



Targets



These inputs were passed on to the network to create the target on the right. Now we know the input to the model



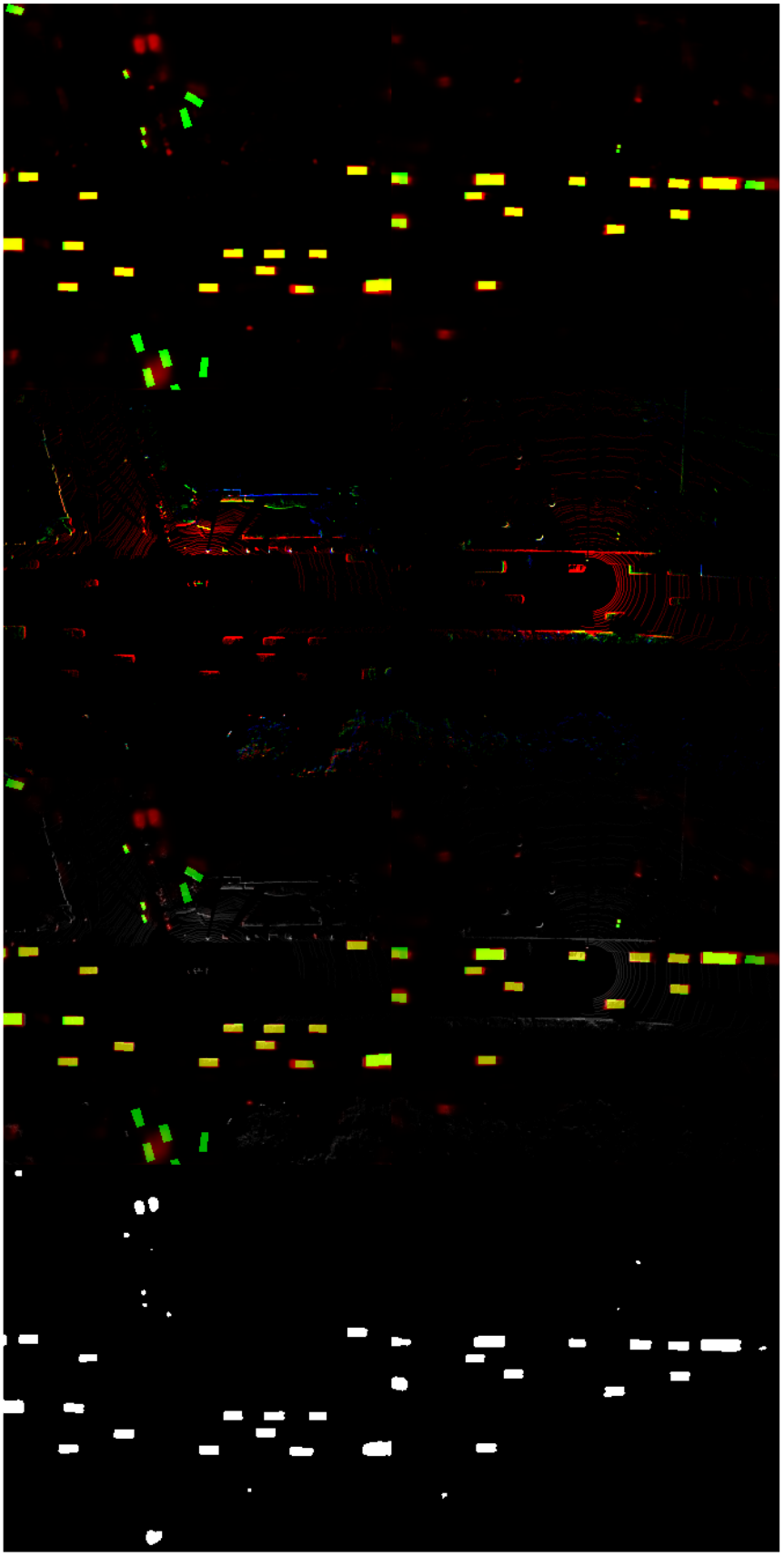
batch_size = 32

epochs = 25

Optimizer =adam
loss = binary_crossentropy

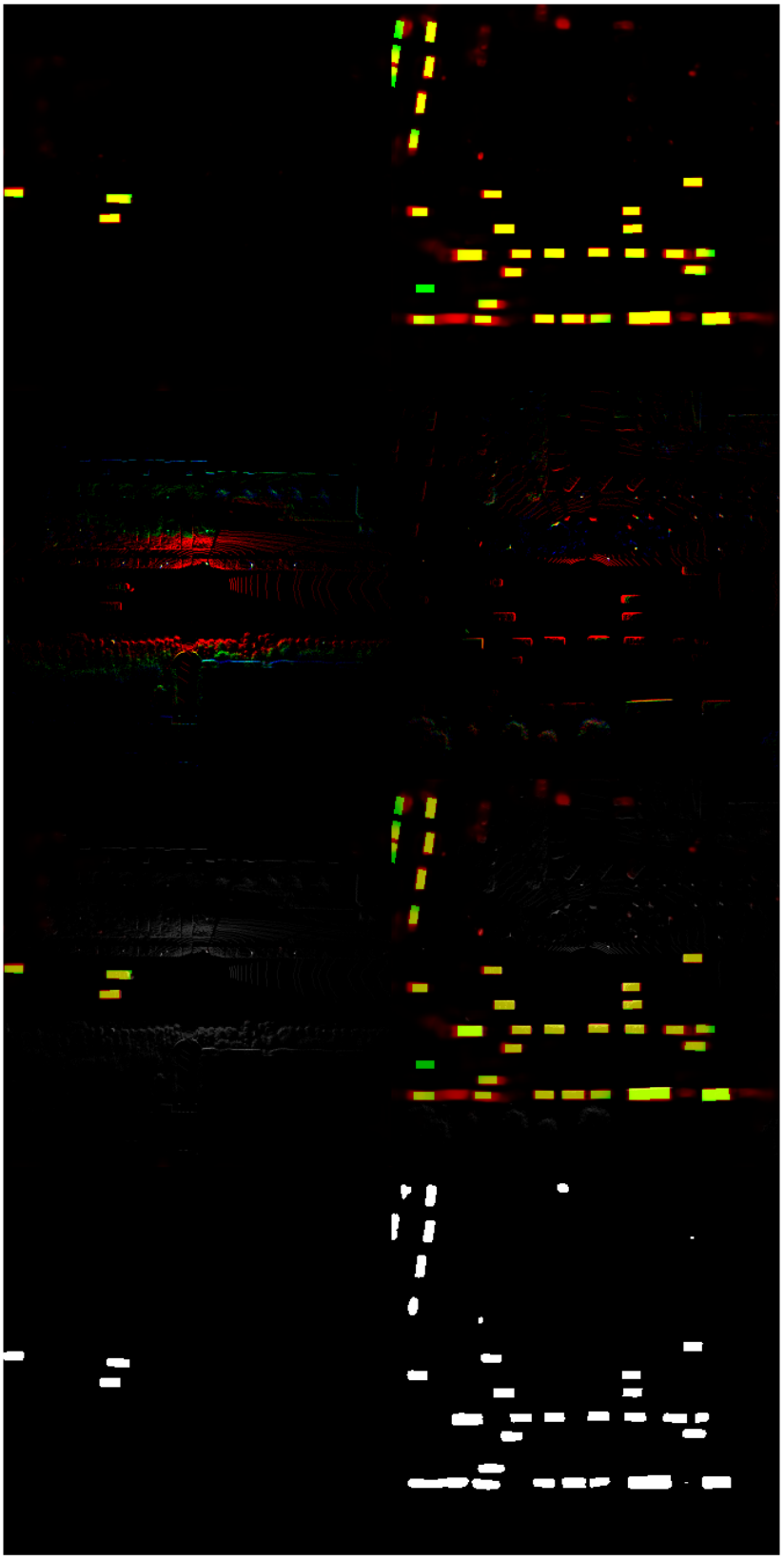
We are visualizing our result after every epoch

e.g after **epoch 5**



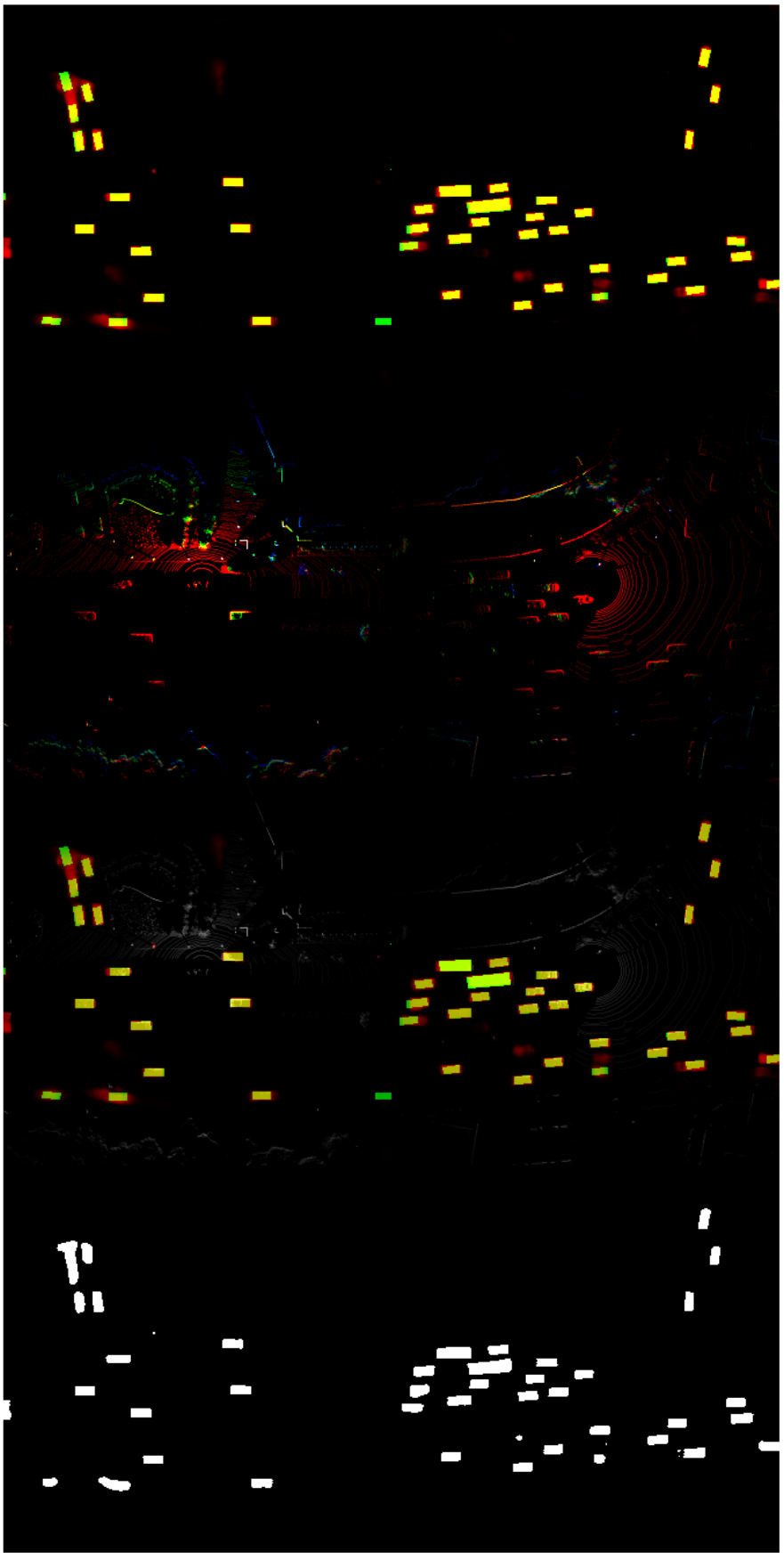
Loss: 0.049734123

After **epoch 10**

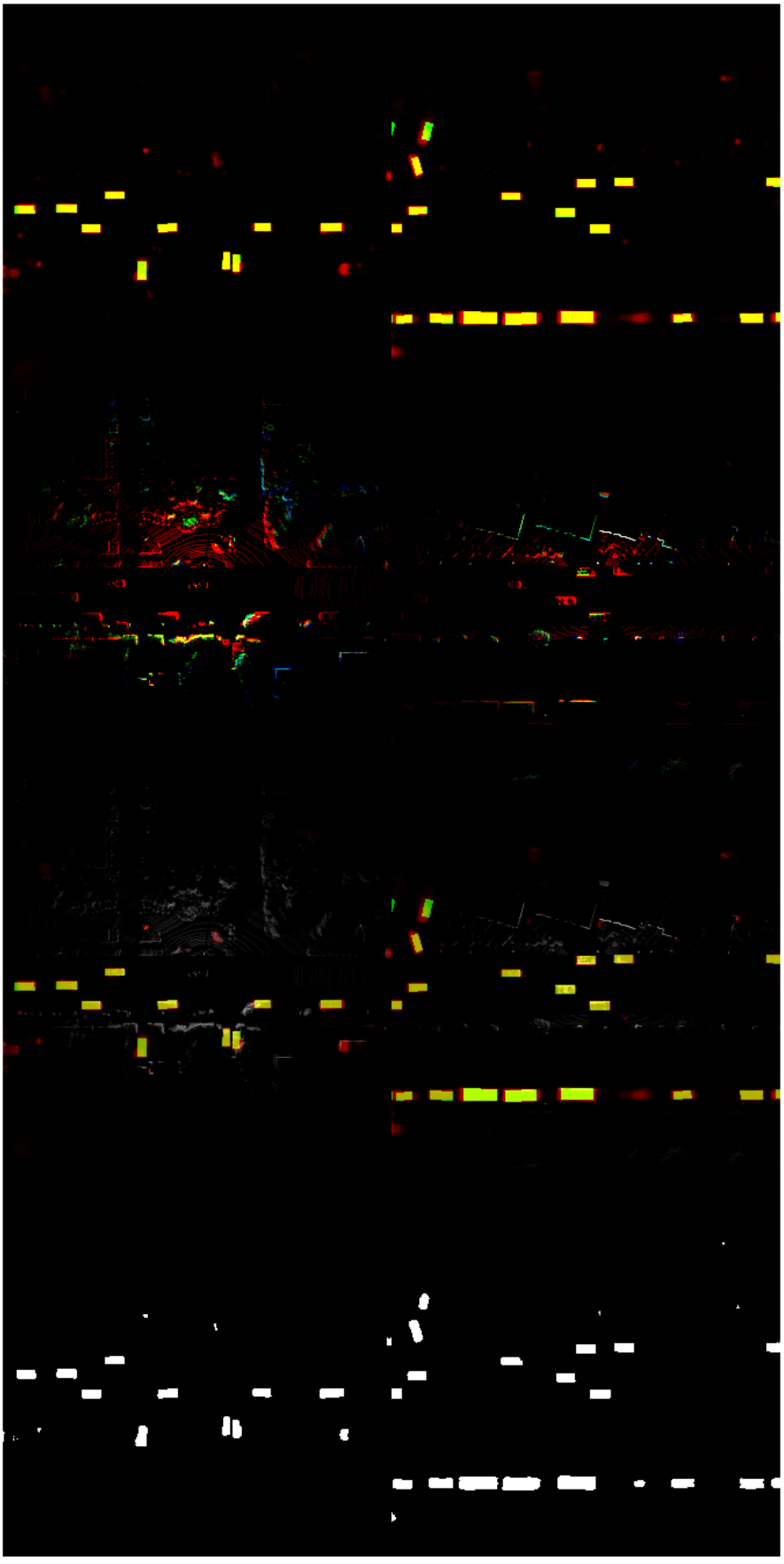


Loss: 0.03505975

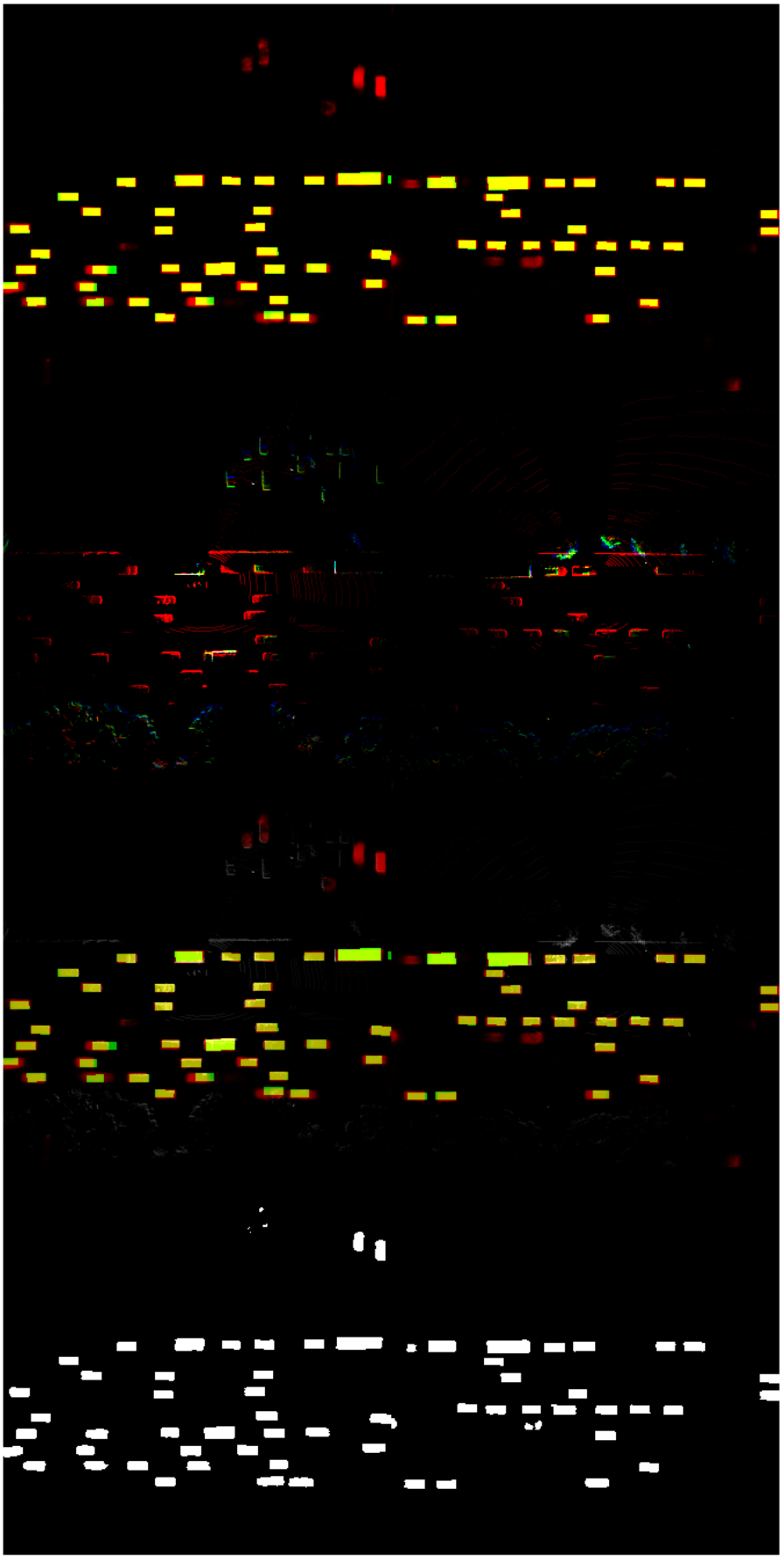
Epoch 15



Epoch 20



Epoch 25



There are four different visualizations stacked on top of eachother:

1. The top images have two color channels: red for predictions, green for targets. Note that red+green=yellow. In other words:

> **Black**: True Negative

Green: False Negative

Yellow: True Positive

Red: False Positive

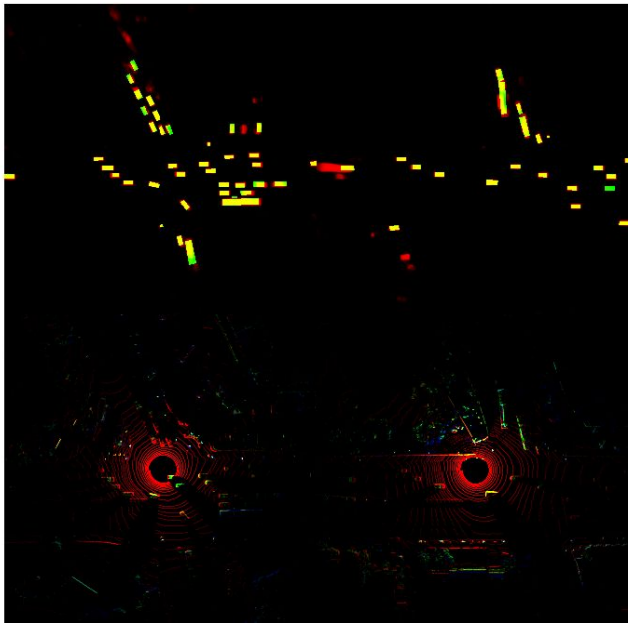
2. The input image

3. The input image (or semantic input map, not in this kernel) blended together with targets+predictions

4. The predictions thresholded at 0.5 probability.

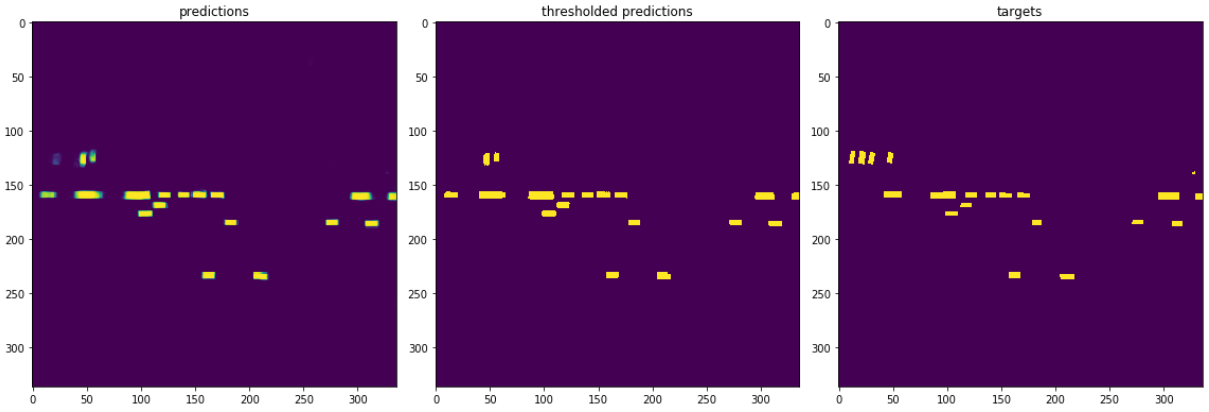
Validation

As we can see, there are a lot of red spots in the top-down projects. Those are the regions where the model is predicting the presence of an object, but there isn't any. The green area is the region where there is an object, but the model fails to predict an object. This is especially true for extremely small objects, such as pedestrians. Finally, the yellow-colored spots are the regions where our model matches the ground truth — True positives!



Visualize prediction:

The output from the network is the region of interest from the BEV. As you can see , not all the predictions are ground truths.



Conclusion:

Class_names = ['animal', 'bicycle', 'bus', 'car', 'motorcycle', 'other_vehicle', 'pedestrian', 'truck']
Average per class mean average precision = 0.10458624970399849
('animal', 0.0)
('bicycle', 0.014827256648354046)
('bus', 0.07114362365576843)
('car', 0.4644436416741515)
('motorcycle', 0.031743736796870606)
('other_vehicle', 0.240433992246761)

('pedestrian', 0.0022918037828294 82)
('truck', 0.011805942827252867)

Refrences:

- https://www.kaggle.com/c/3d-object-detection-for-autonomous-vehicles
- https://www.kaggle.com/gzuidhof/reference-model
- https://arxiv.org/abs/1505.04597
- https://www.mathworks.com/help/driving/ug/coordinate-systems.html
- https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47