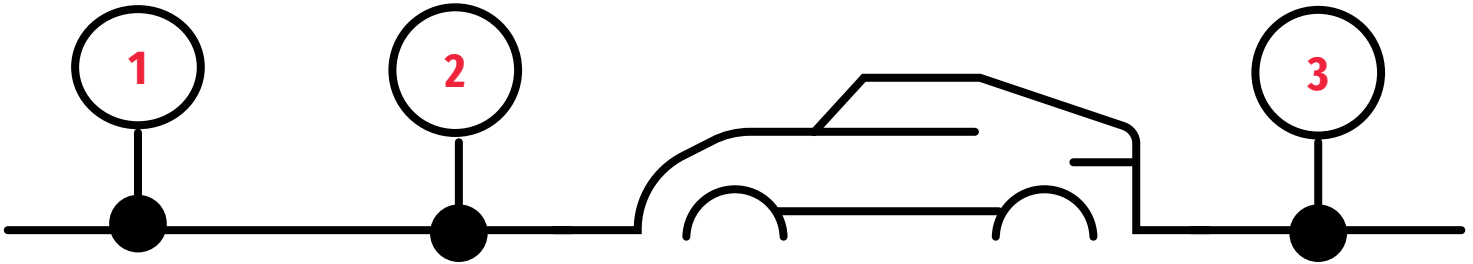
Car orientation detection

04/09/2022

Car orientation detection



Project background & description

Results

Conclusions, things we learned / didn't expect

Project Description

In this project we set out to use deep learning to solve a multiclass classification problem: Given an image of a car, detect it's orientation (Front, Front-Right, Front-Left, Back, Back-Right, Back-Left, Left ,Right).

Back-Right

Project purpose & Possible uses

* Can have some uses in autonomous driving (collision detection, detecting car turns etc.)

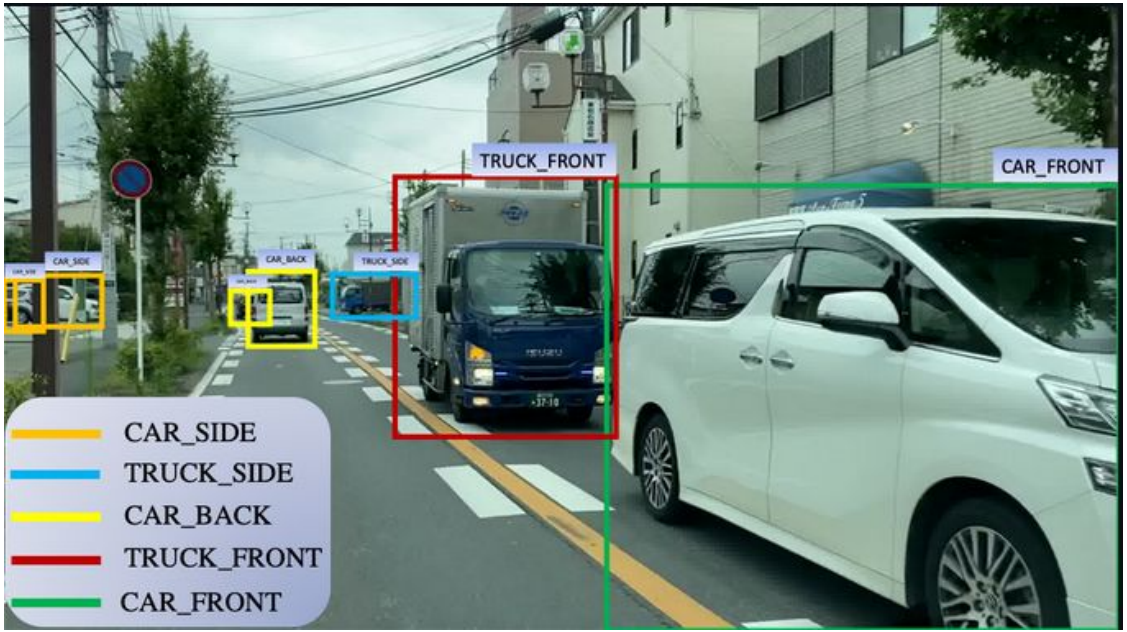
* Can be used by, for example, second hand selling websites, to validate that every side of the car was shown in the pictures taken by the seller.

* Counting the number of cars the are leaving / entering a place

Initial Idea

A dataset exists which contains images of cars labeled with Front, Back or Side.

This dataset had a few problems:

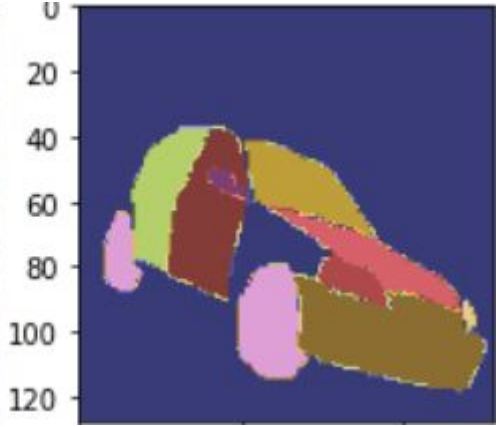


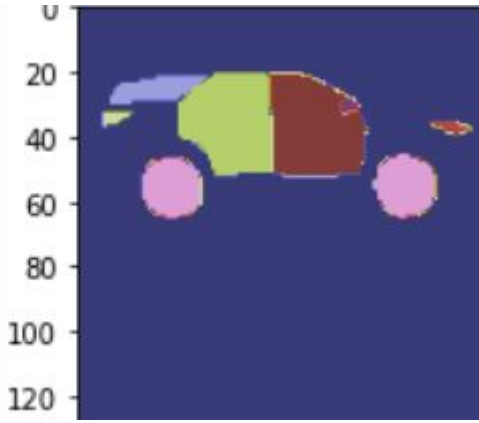
The problems with the dataset

1. Labels contained only Front, Side or Back.
2. Since the data was taken on the road in Japan:
   a. Road data is very noisy / messy
   b. Cars in Japan tend to be “boxier” than what we would consider the common car
3. Since the labeling was done semi automatically (Initial labels were given by humans however after a certain amount the rest were done automatically), not all labels were accurate.

New Idea

Since we couldn’t use the existing dataset, we started looking for other datasets and found an instance segmentation dataset for car parts, containing 500 images and 19 classes(Background, Front window, Back Left Door, Wheels etc.)





New Idea

Can we use the car parts to determine which sides of the car are visible ?

We probably could however we still have only 500 images, and these images are “cleaner” than images taken on the road like in the previous dataset.

We decided to try and use these images to create a general algorithm which would be able to label new images for us, and then we would only have to validate our algorithms labels instead of labeling all the data ourselves.

The Algorithm

First, train an instance segmentation network to detect car parts, then given the car parts detected, try and decide which car directions are visible. We would use this algorithm to label the existing images in this dataset and other images from other datasets in order to generate enough data in order to train a CNN for our original classification problem.

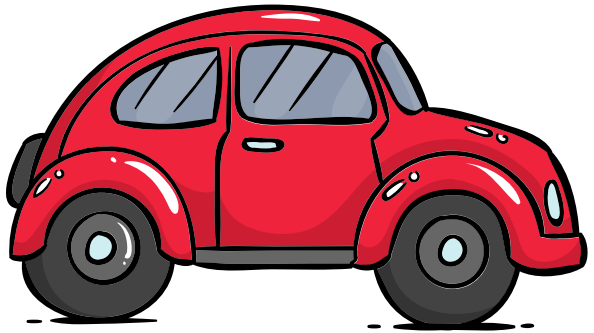
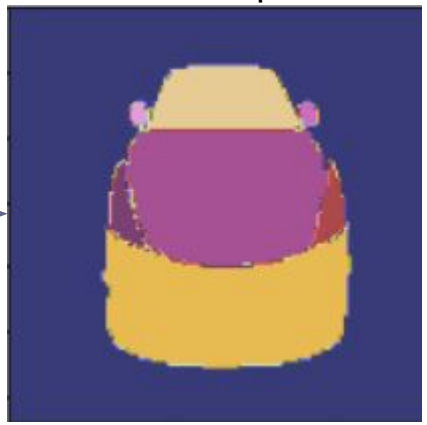


Image Segmentation networks

We are trying to solve a semantic segmentation problem: Given an image of a car, assign each pixel in it to one of 19 classes (Background, Wheel, Front Left door etc.). For this we'll use an Image segmentation network.

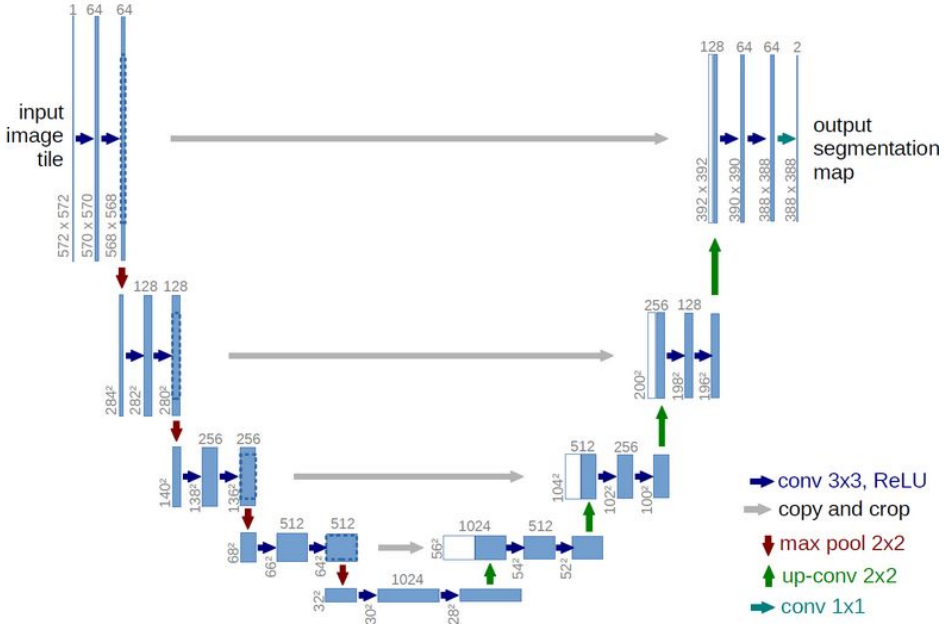
Input

Desired Output



The Network (U-Net)

The network architecture we will use is a U-Net. A U-Net is an Encoder-Decoder network where the decoders inputs are concatenated with their equivalent encoder levels in what are called “skip connections”.



The Network (U-Net)

Eventually, we settled on the U-Net from the official tensorflow guide on image segmentation. The network uses a pre-existing model (MobileNetV2) to learn the images features (the encoder) and existing layers from a model called pix2pix as the upsampler (decoder).

Before training, the encoder and decoder are connected with skip connections using the Concatenate() layer implemented in Keras.

The model will return n_classes probability matrices, where each value denotes the corresponding pixels probability of being of class n.

Preparing the data

Surprisingly, preparing the data for training took much longer than we anticipated.

The data was stored in a known format called COCO, however we couldn’t find any existing functions / libraries that could fit our labeling needs.

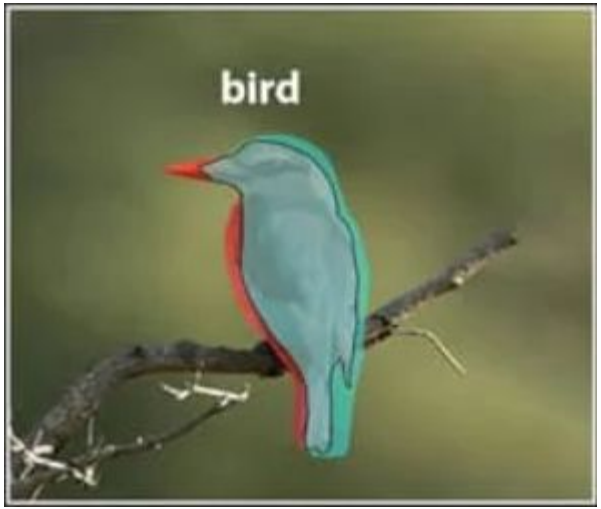
Since we had more than 2 classes (19!), we had to manually iterate over each class and paint it’s category_ID onto an empty image until we have the final label.

This became a bigger problem later when we tried to omit certain classes, since our network required the labels to be continues (0, 1, 2, 3 … 19) without any gaps (1, 3, 5… 13) would throw an error.

Metrics

While training, we used two main metrics:

1. Accuracy: How many pixels were classified correctly
2. IoU: The mean class wise overlap between the true label and our predicted label



Implementing metrics

While using accuracy required no special attention (simply use metrics = [‘accuracy’]), the IoU required a special implementation, since the baseline tensorflow metrics expects two n x m images, not an n x m image and a 19 x n x m image !

```
class MyMeanIOU(tf.keras.metrics.MeanIoU):
    def update_state(self, y_true, y_pred, sample_weight=None):
        return super().update_state(y_true, tf.argmax(y_pred, axis=-1), sample_weight)
```

Metrics can be deceiving

Since one of the classes we were trying to predict was “background”, by default, a network that would only predict background could reach around ~67% accuracy !

Using Mean-IoU gave a much better indicator of our networks performance and how much progress we we’re making.

First 2 epochs

```
- val_accuracy: 0.7919 - val_my_mean_iou: 0.1179

val_accuracy: 0.8260 - val_my_mean_iou: 0.1513
```

Last 2 epochs

```
val_accuracy: 0.8722 - val_my_mean_iou: 0.3588

val_accuracy: 0.8715 - val_my_mean_iou: 0.3592
```

Training

One of the biggest problems we encountered while training was NaN loss / No learning. This happened on multiple occasions and with multiple networks, what helped was:

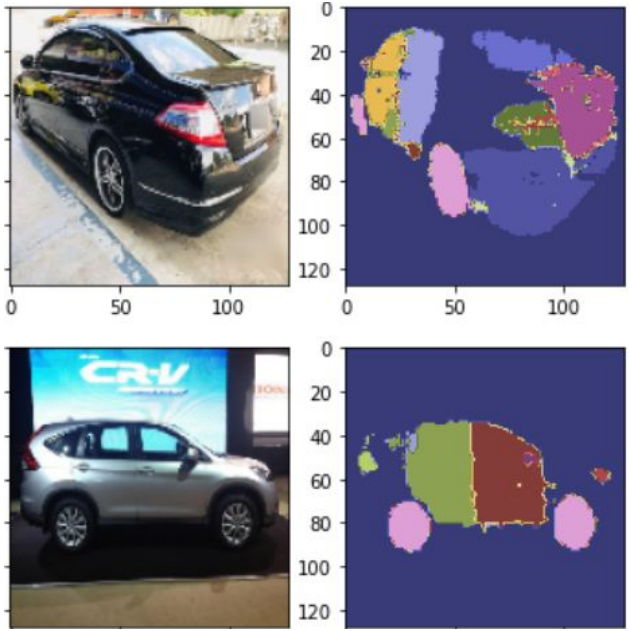
- Changing the complexity of the problem (changing image resolution, changing the number of classes etc.)
- Changing hyperparameters: Batch Size, Learning Rate.
- Something that helped us diagnose the problem: Try overfitting on a single sample, this did not always guarantee proper learning would occur on the whole data, but would give a good and quick indicator when trying different things.

```
Epoch 1/10
236/236 [==============================] - 3s 11ms/sample - loss: nan - accuracy: 0.0932 - val_loss: nan -
val_accuracy: 0.0000e+00
Epoch 2/10
236/236 [==============================] - 1s 4ms/sample - loss: nan - accuracy: 0.0000e+00 - val_loss: na
n - val_accuracy: 0.0000e+00
```

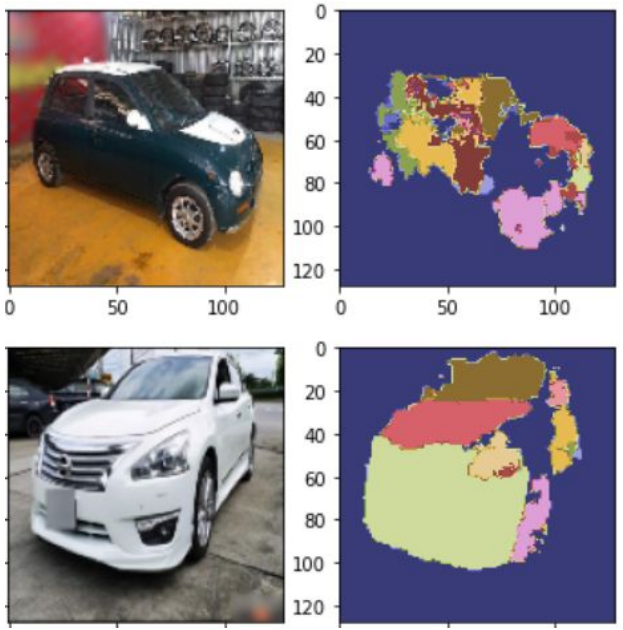
Results

So, now that we trained our network, let's see how the classified samples look…

Training Set

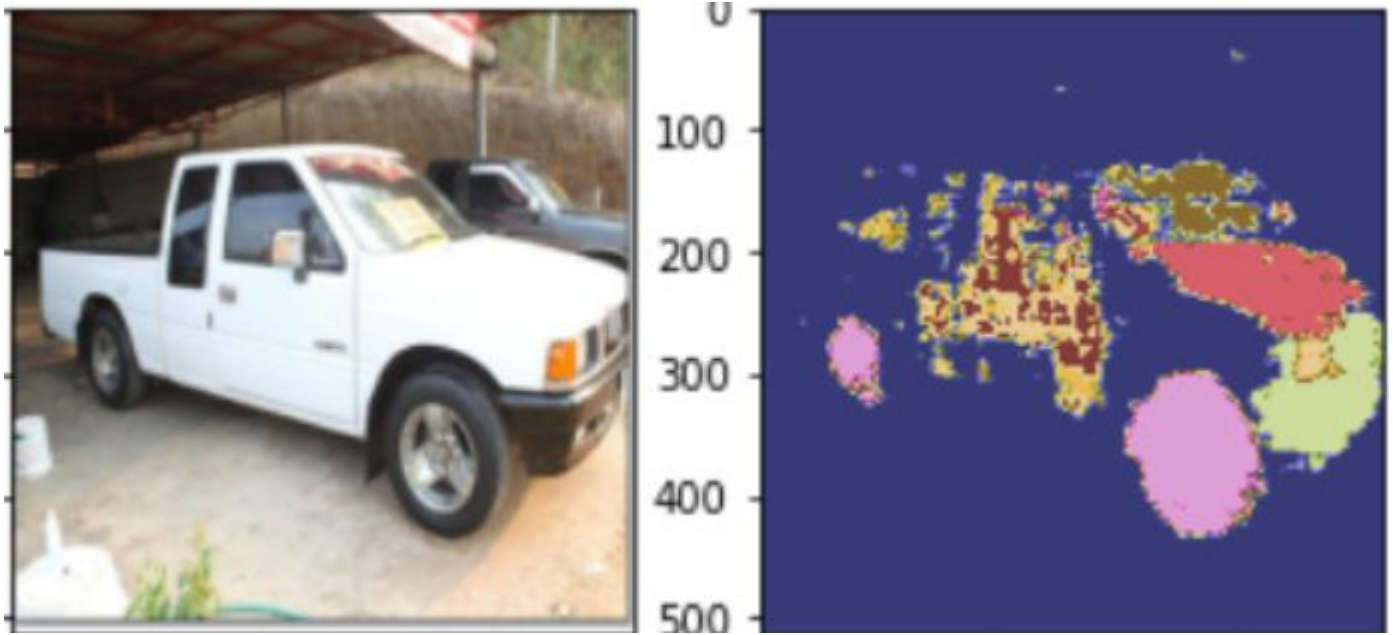


Testing Set



Dealing with noise

How do we take a prediction like this and figure out which classes were actually there, and which were just noise ?



Idea #1 / Intuition - Check which classes appear

Our first idea was the most obvious one, just look at the prediction and check which classes appear, if any pixel of, for example, a wheel, was predicted, assume a wheel is in the picture.

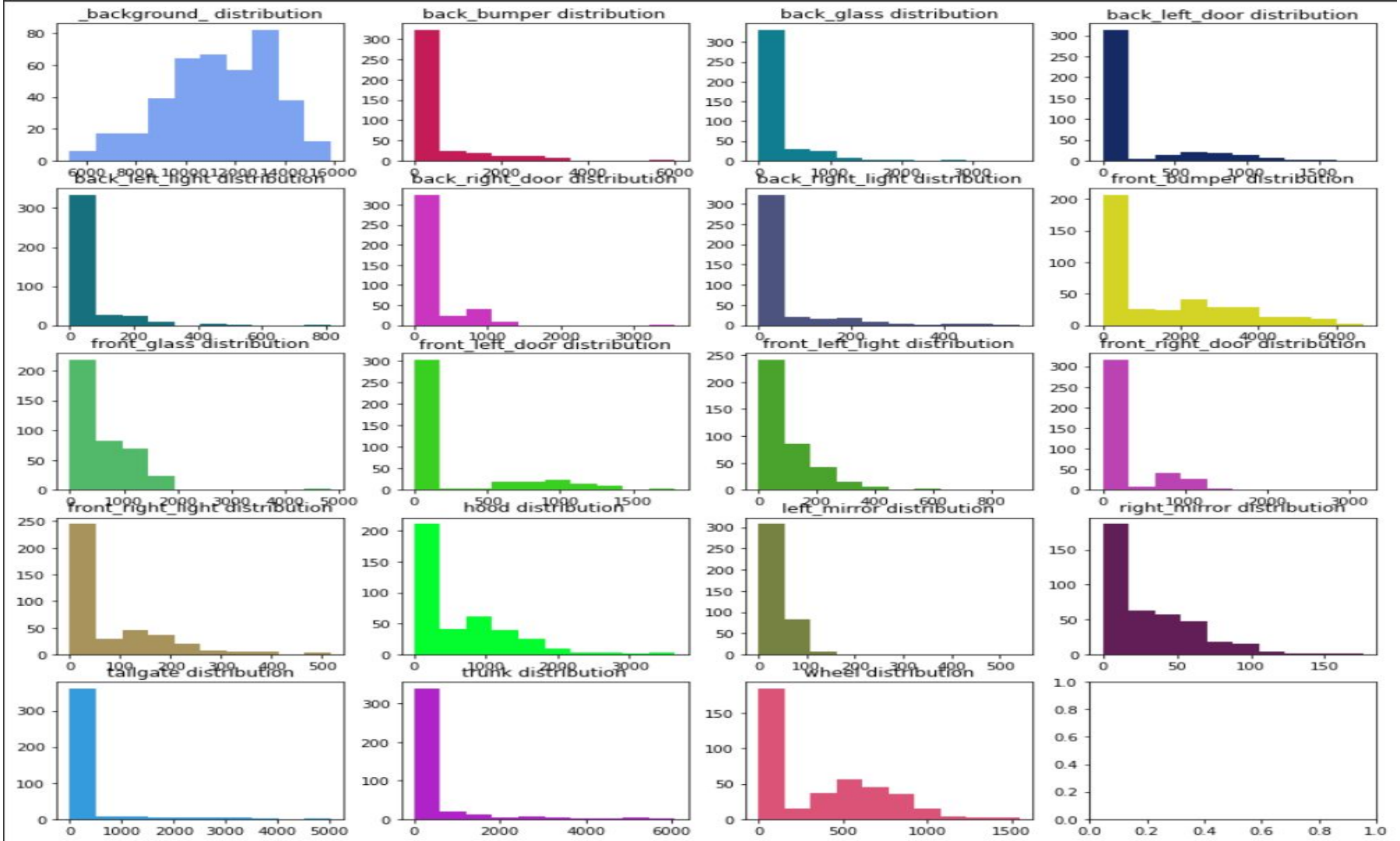
Unsurprisingly, this did not work, since nearly every prediction contained at least a single pixel of each of the 19 classes…

Idea #2 - Check distribution across labels

Next we figured we'd check how each class distributes among the original labels, for example, if we know that the front bumper usually takes 1,000 pixels in the image, we'd check the amount of front bumper pixels our network predicted, and if the number was ~1,000, we'd consider the front bumper present.

This idea has multiple problems:

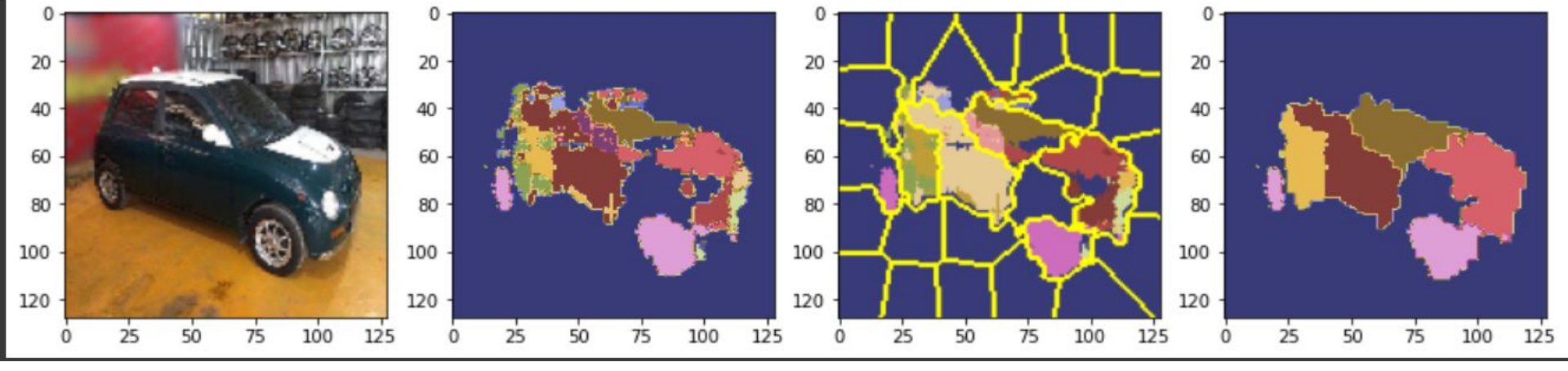
- Doesn't consider the cars scale in the image: cars further away would be judged like cars closer to the camera.
- Does not take into account the pixels locality, just their presence.
- Pixels distribution was less than ideal…


background distribution
back_bumper distribution
back_glass distribution
back_left_door distribution
back_left_light distribution
back_right_door distribution
back_right_light distribution
front_bumper distribution
front_glass distribution
front_left_door distribution
front_left_light distribution
front_right_door distribution
front_right_light distribution
hood distribution
left_mirror distribution
right_mirror distribution
tailgate distribution
trunk distribution
wheel distribution

Idea #3 - More segmentation !

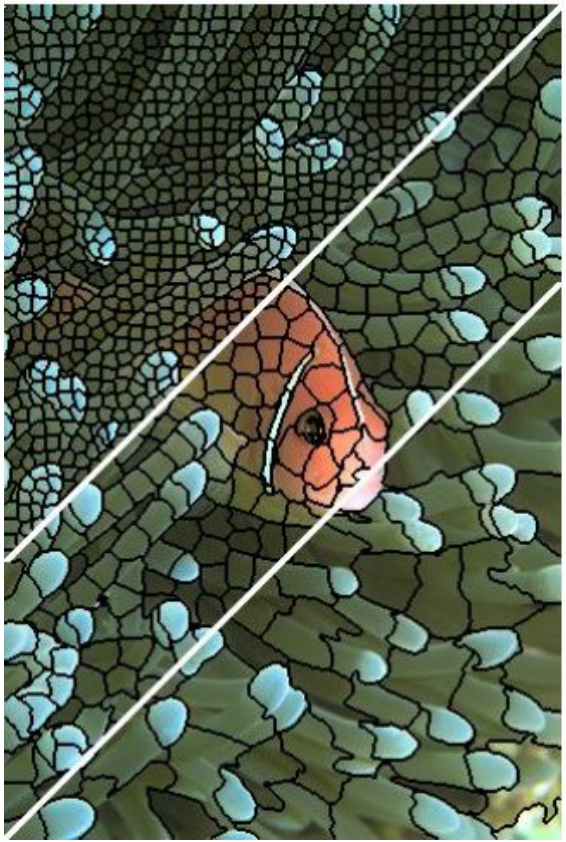
Instead of trying to make sense of the noise, we decided to try and remove it entirely while losing minimal useful information with an algorithm called SLIC, which is very similar to K-Means.

```
SLIC number of segments: 27
Wheel Found : True
{'front_glass', 'wheel', '_background_', 'front_right_door', 'hood', 'front_left_door'}
front_count : 2 back_count : 0 left_count : 1 right_count : 1
'front-left'
```



SLIC (Simpler Iterative Linear Clustering)

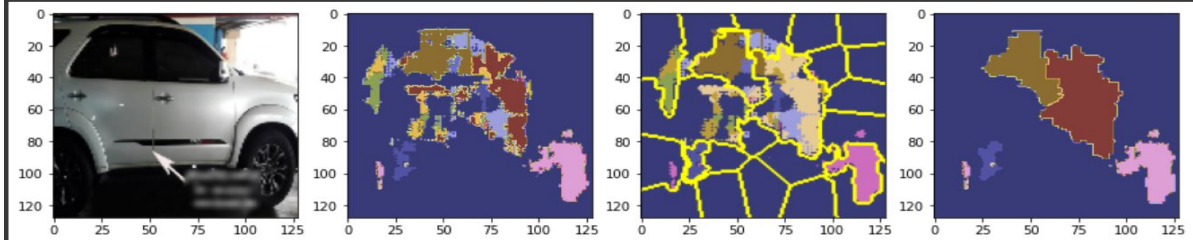
SLIC is a clustering algorithm that works similarly to K-Means: Pixels are clustered in the [R, G, B, x, y] space into “Superpixels”. A Superpixel is a collection of normal pixels which, after clustering, are all given the same value. Parameters can control the approximate size and count of superpixels, and the amount of importance given to the space proximity in relation to color proximity.



SLIC Hyperparameters (n_segments)

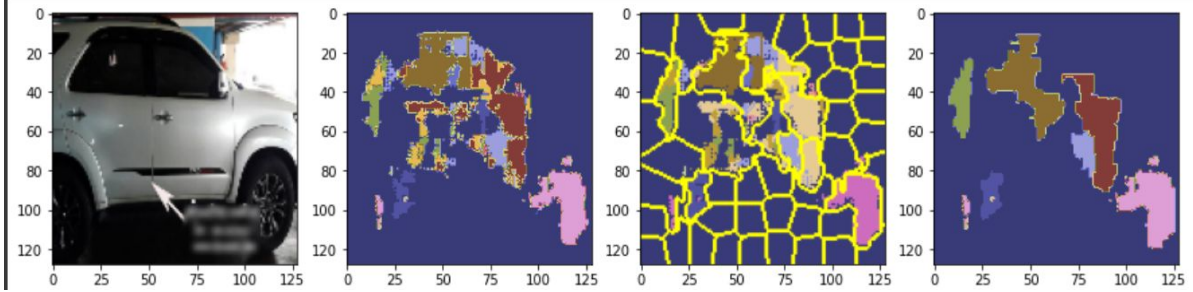
N_segments = 50:: Wrong prediction

```
SLIC number of segments: 23
Wheel Found : True
{'front_glass', 'front_right_door', 'wheel', '_background_', 'back_bumper'}
front_count : 1 back_count : 1 left_count : 0 right_count : 1
'front-left'
```



n_segments = 100+: Right prediction

```
SLIC number of segments: 50
Wheel Found : True
{'front_glass', 'back_right_door', 'front_right_door', 'wheel', '_background_', 'back_left_door', 'back_bumper'}
front_count : 1 back_count : 1 left_count : 1 right_count : 2
'right'
```

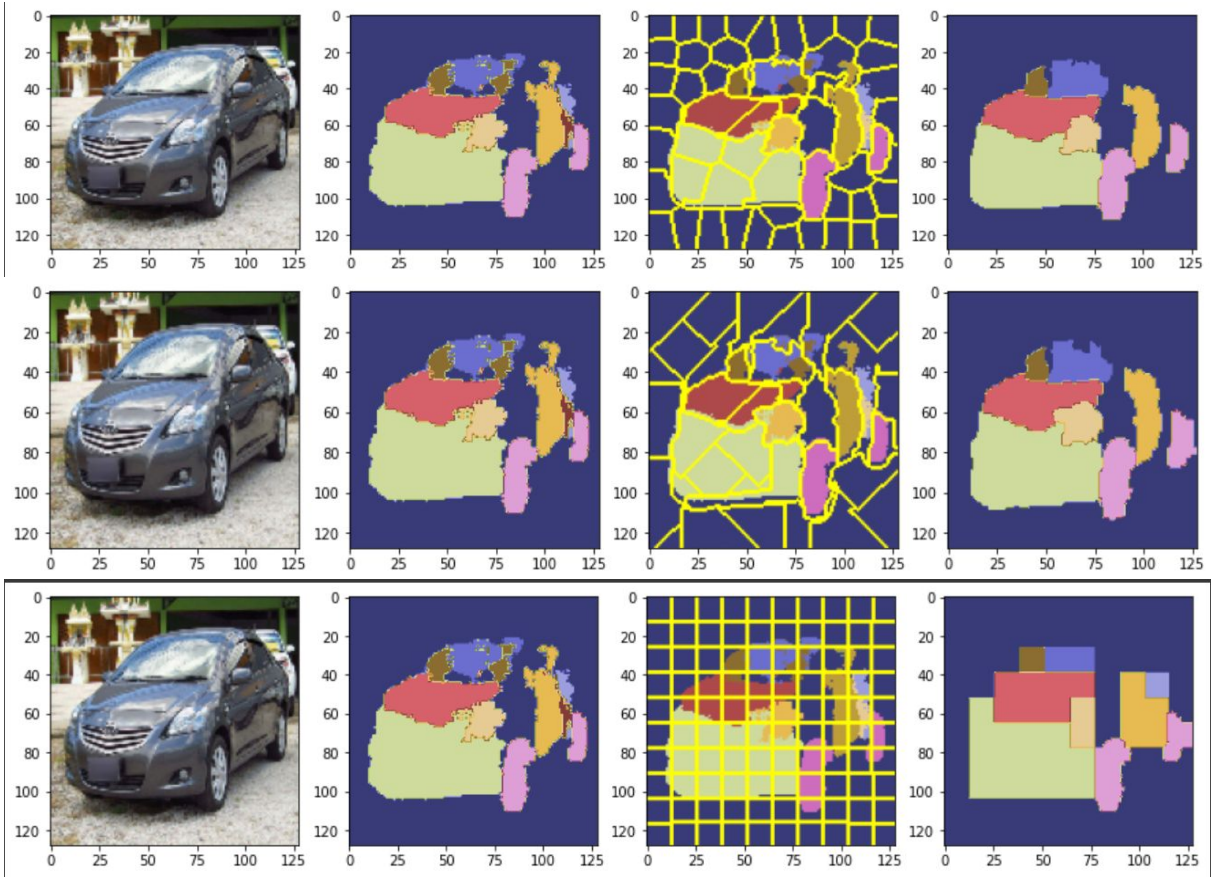


SLIC Hyperparameters (compactness)

Compactness: 1 (Default)

Compactness: 0.0001

Compactness: 10,000



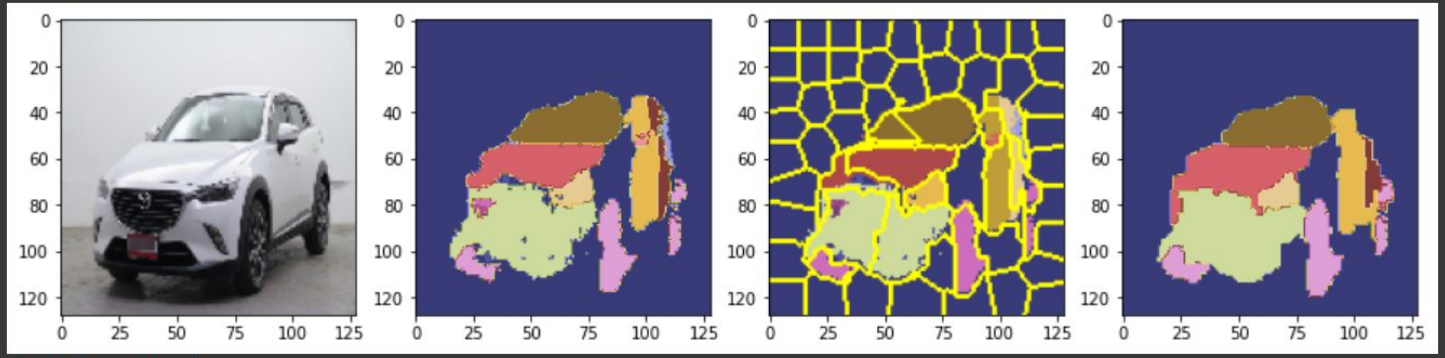
Now that we have cleaner predictions..

How do we turn the data we have into a final prediction ?

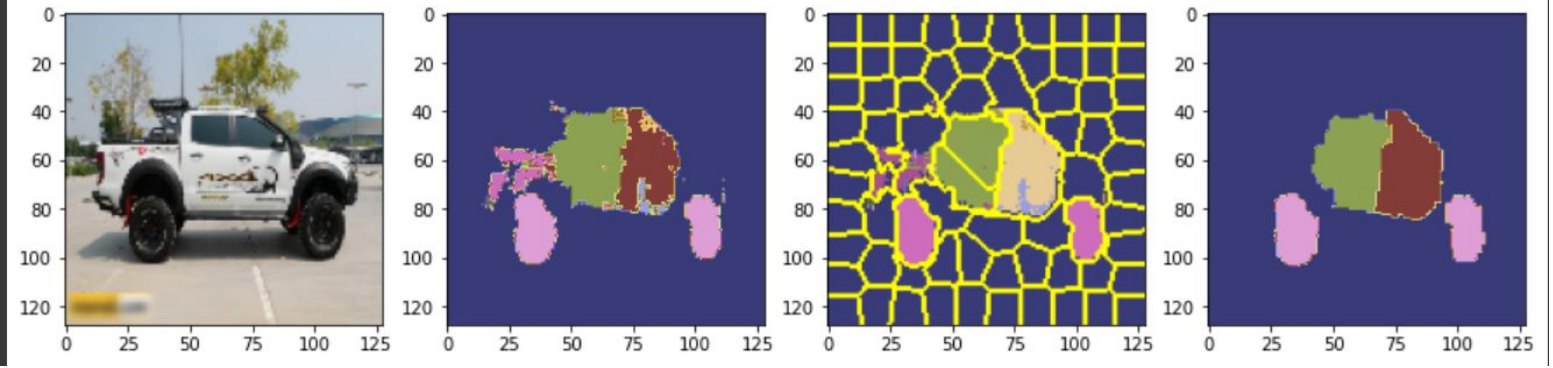
Ruleset:

1. Put all predicted parts into 4 bins: Front, Back, Left, Right.
2. If the biggest bin is left or right, predict the biggest bin.
3. If the biggest bin is front or back, check if a wheel was detected, if there is no wheel, pick the biggest bin.
4. If there's a wheel, calculate the wheels centroid and the front / back centroid, and us their location to determine the final prediction.
5. Any equality (e.g front_count == back_count) is decided by counting overall pixels for each bin.

```
Wheel Found : True
{'front_bumper', 'front_left_door', 'front_glass', 'front_right_door', 'wheel', '_background_', 'front_left_light', 'hood'}
front_count : 3 back_count : 0 left_count : 1 right_count : 1
'front-left'
```



```
Wheel Found : True
{'back_right_door', 'front_right_door', 'wheel', '_background_'}
front_count : 0 back_count : 0 left_count : 0 right_count : 2
'right'
```

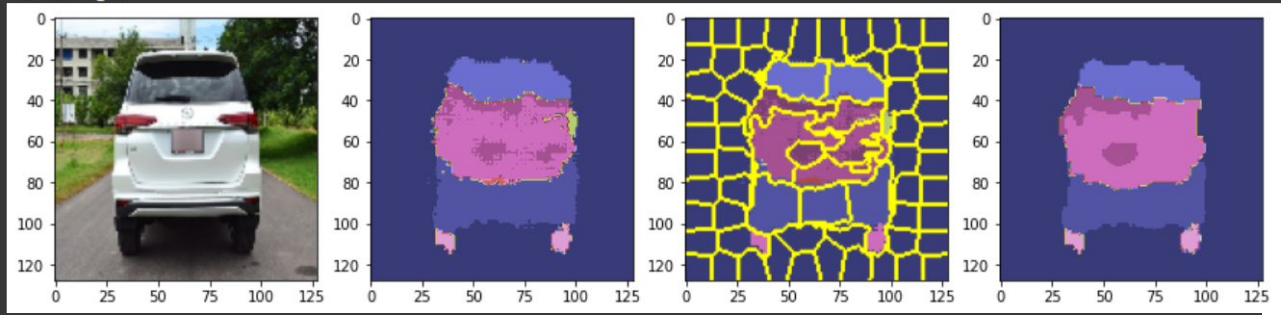


Left vs Right count equality is decided by wheel location

Right is decided by the biggest bin

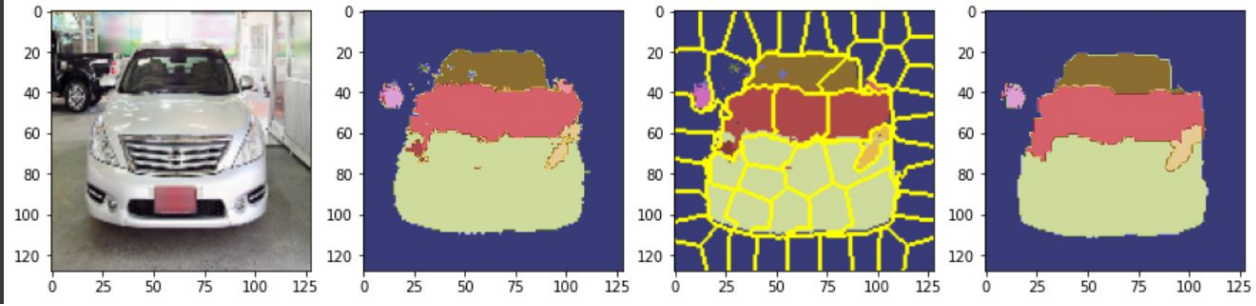
Where does it go wrong ?

```
Wheel Found : True
{'back_glass', 'trunk', '_background_', 'tailgate', 'back_bumper'}
front_count : 0 back_count : 4 left_count : 0 right_count : 0
'back-right'
```



Back right was predicted since wheels were found.

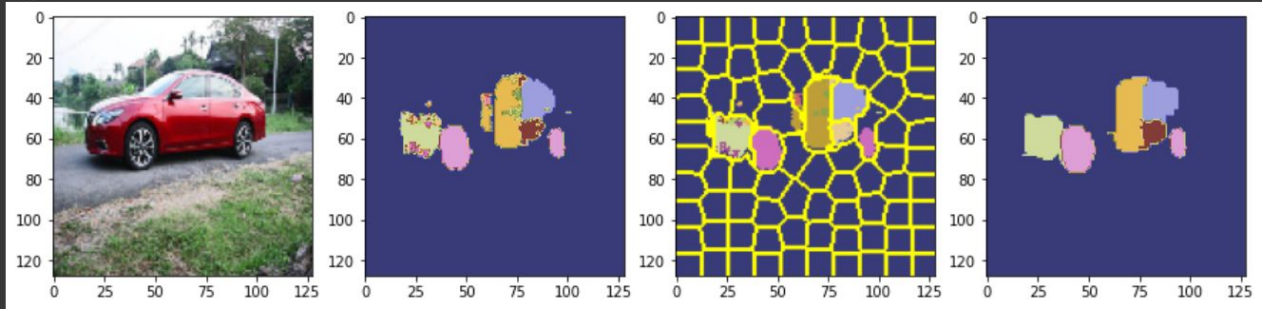
```
Wheel Found : True
{'front_bumper', 'front_glass', '_background_', 'front_left_light', 'hood'}
front_count : 3 back_count : 0 left_count : 0 right_count : 0
'front-right'
```



Other cars wheel was detected…

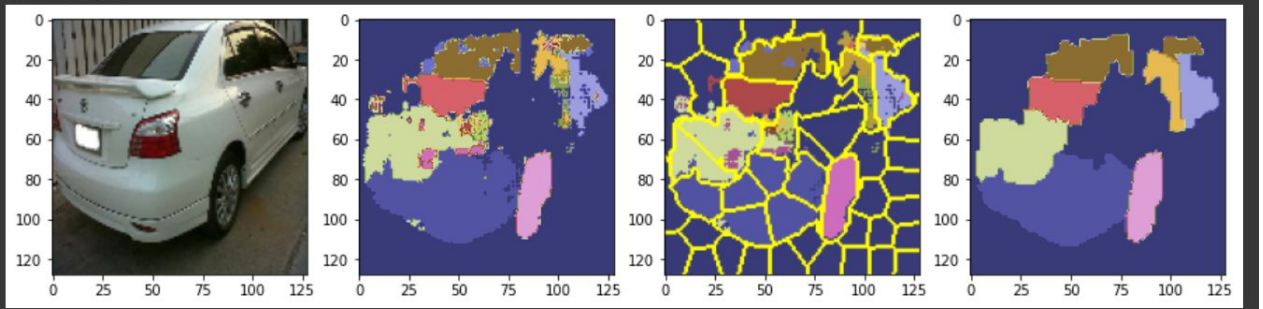
Where does it go wrong

```
Wheel Found : True
{'front_bumper', 'front_left_door', 'front_right_door', 'wheel', '_background_', 'back_left_door'}
front_count : 1 back_count : 0 left_count : 2 right_count : 1
'left'
```



A Front segment was detected, but since there were more left detections left was decided.

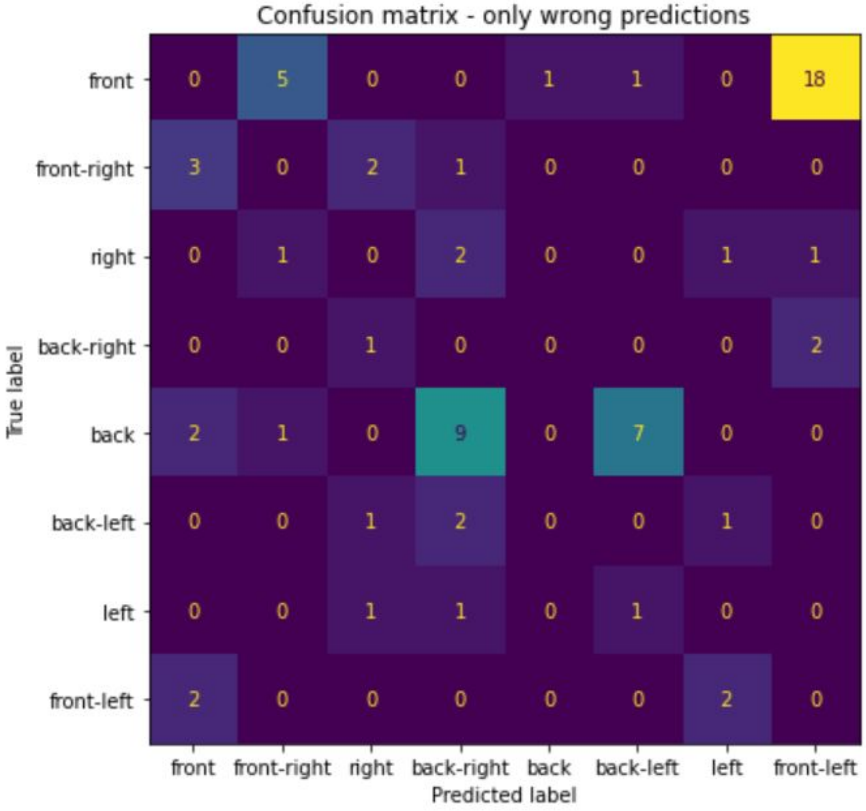
```
SLIC number of segments: 47
Wheel Found : True
{'front_bumper', 'front_left_door', 'front_glass', 'wheel', '_background_', 'back_left_door', 'hood', 'back_bumper'}
front_count : 3 back_count : 1 left_count : 2 right_count : 0
'front-left'
```



Front was (incorrectly) the majority class…

Overall accuracy

On this dataset, our algorithm achieved 86% accuracy (it's important to note that 80% of the data was the segmentation networks training data, so this score is positively biased).

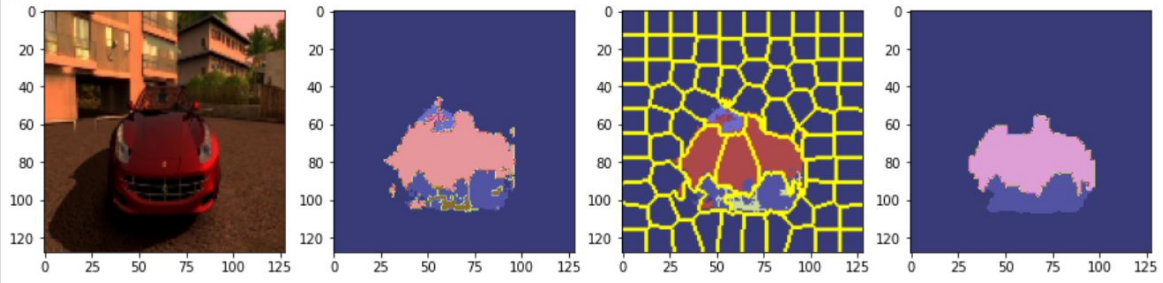


New Data (Stanford cars dataset)

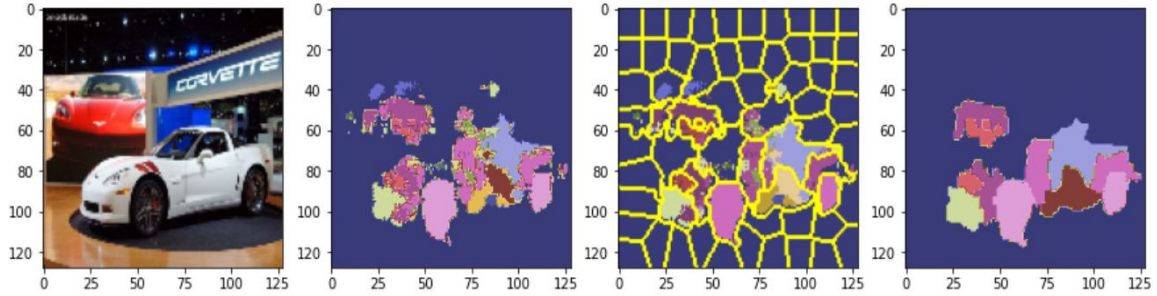
For our next dataset, we chose to test our algorithm on the stanford cars dataset, which contains images similar to our original dataset, however completely new to our network.

While some of the images were less than ideal...

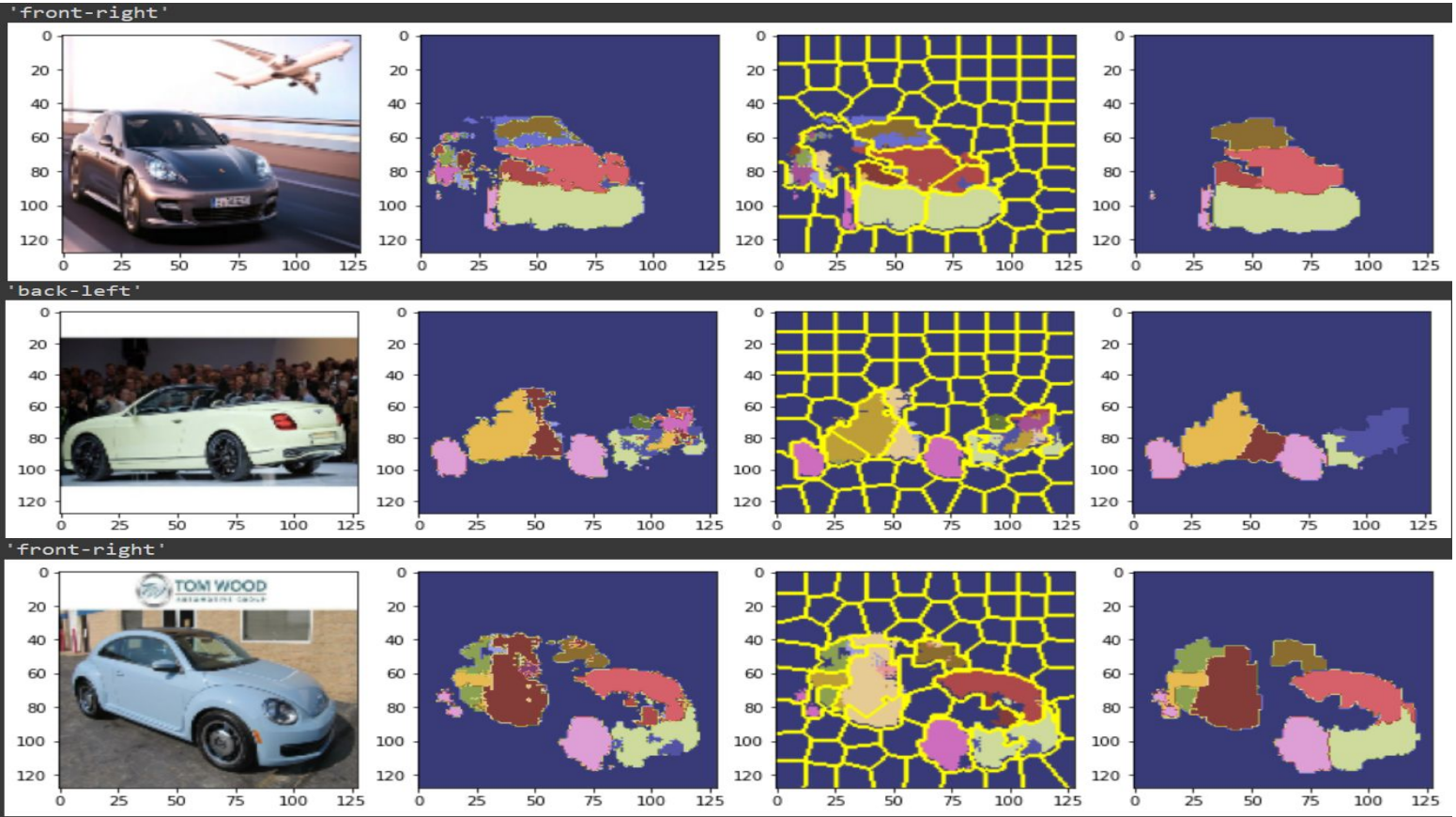
```
Wheel Found : False
{'hood', '_background_', 'back_bumper'}
front_count : 1 back_count : 1 left_count : 0 right_count : 0
'back'
```



```
Wheel Found : True
{'tailgate', 'hood', 'front_bumper', 'wheel', '_background_', 'trunk', 'back_left_door', 'front_right_door'}
front_count : 2 back_count : 2 left_count : 1 right_count : 1
'back-right'
```

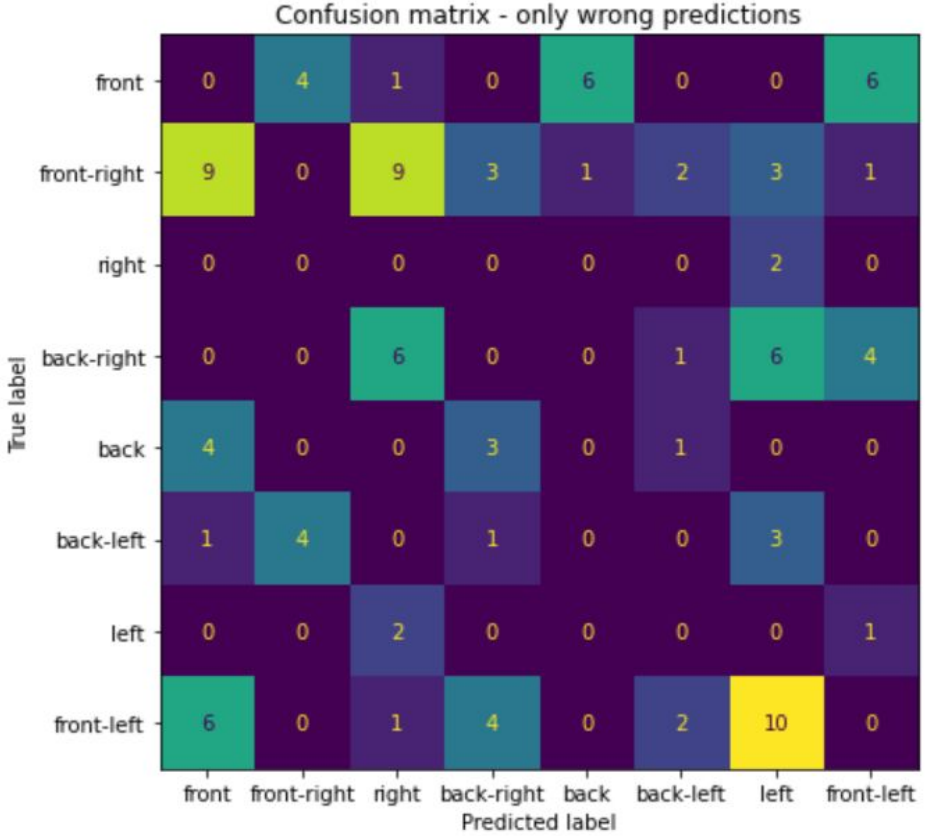


Our algorithm still got most of them right



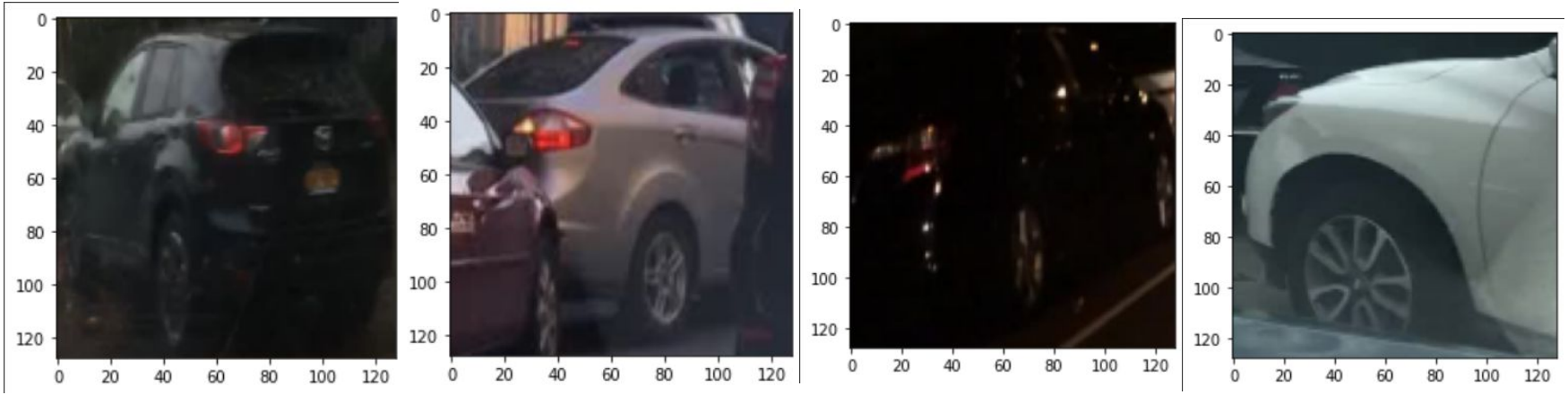
Stanford dataset results

On this dataset, our algorithm achieved 74% accuracy.



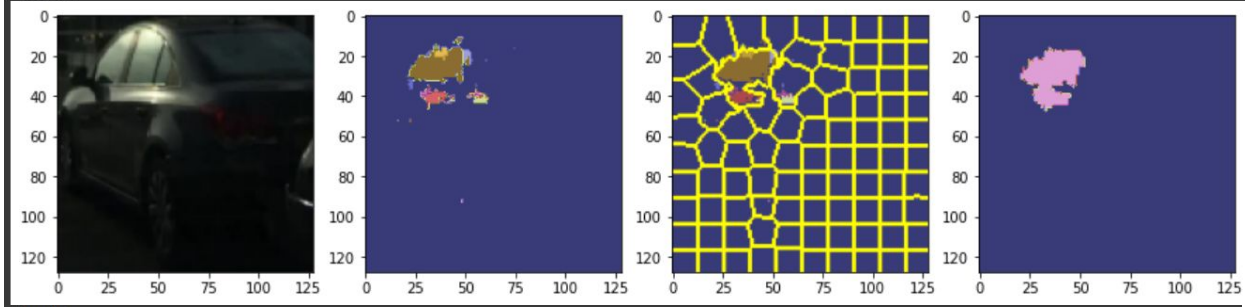
Final Dataset: Berkeley Deep Drive

The final dataset we tested our algorithm on was a collection of images we extracted from the Berkeley Deep Drive dataset, which is a collection of videos and images taken from cars on the road for use in autonomous driving tasks.

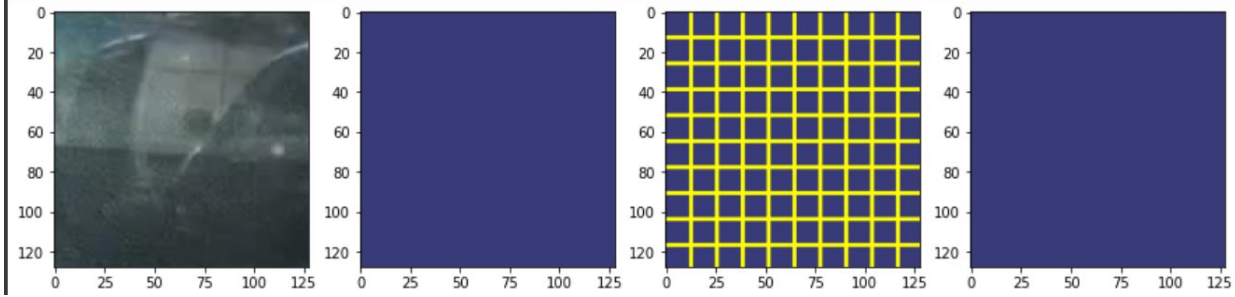


These images were much more challenging for our algorithm..

```
Wheel Found : False
{'front_glass', '_background_'}
front_count : 1 back_count : 0 left_count : 0 right_count : 0
'front'
```

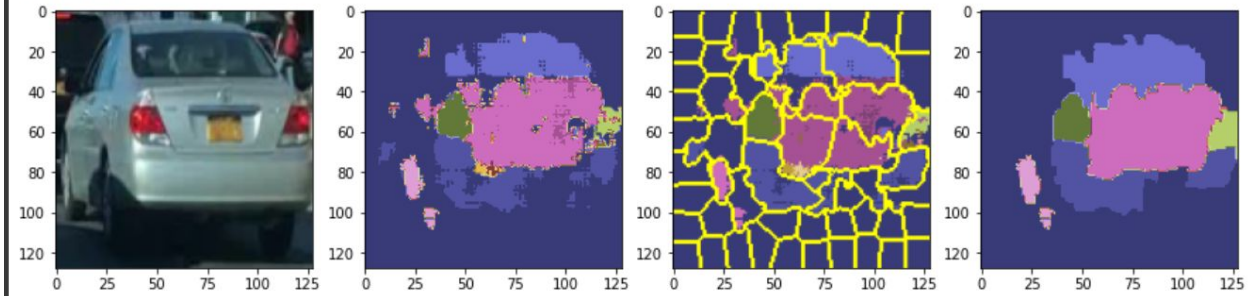


```
'back'
```

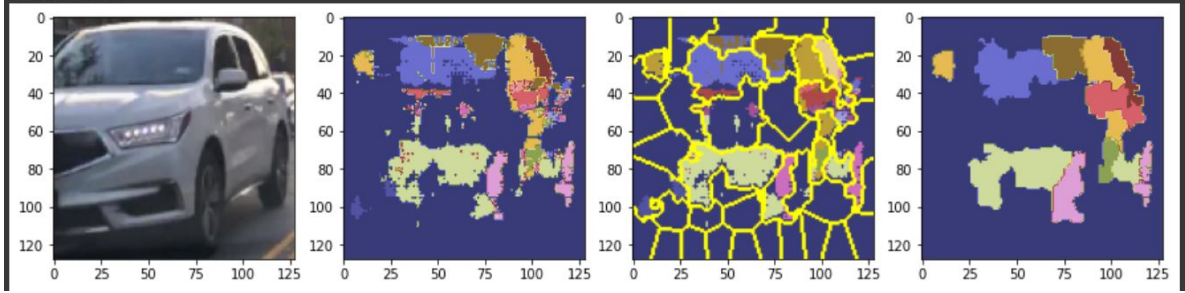


But some still came out fine

```
Wheel Found : True
{'back_glass', 'trunk', 'wheel', '_background_', 'back_left_light', 'back_right_light', 'back_bumper'}
front_count : 0 back_count : 3 left_count : 0 right_count : 0
'back-left'
```

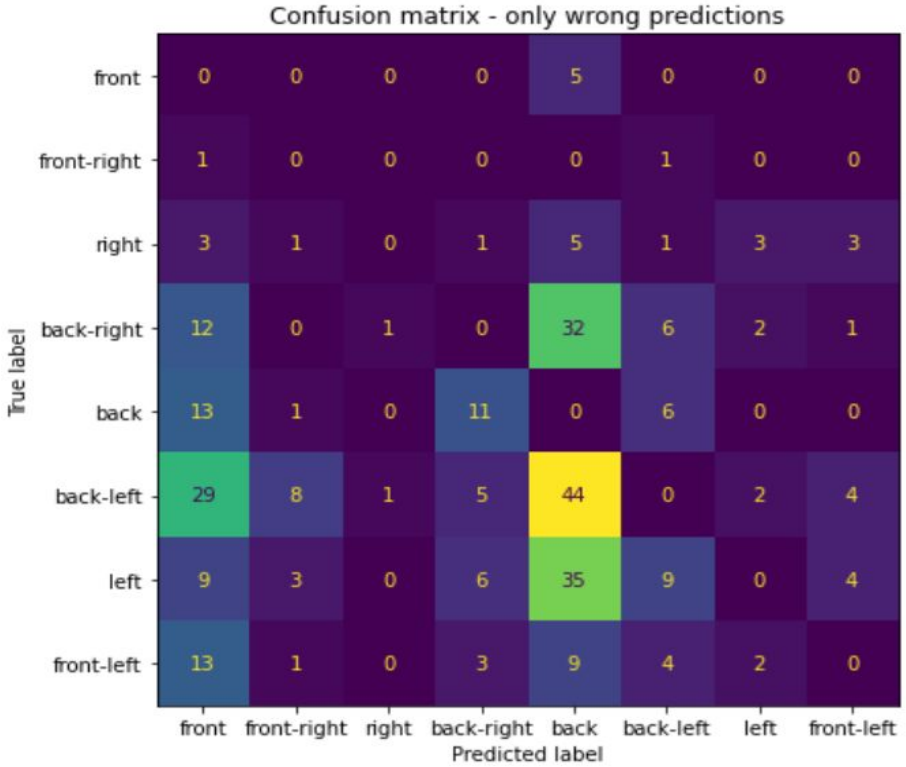


```
Wheel Found : True
{'back_glass', 'front_bumper', 'front_left_door', 'front_glass', 'back_right_door', 'front_right_door', 'wheel', '_background_', 'hood'}
front_count : 3 back_count : 1 left_count : 1 right_count : 2
'front-left'
```



Berkeley Dataset results

On this dataset, our algorithm achieved 42% accuracy.



Overall results of algorithm #1

Orginal Segmentation Dataset: 86% accuracy

Stanford Cars Dataset: 74% accuracy

Berkley Dataset: 42% accuracy

Overall Accuracy: ~67.3%

AVG inference time: 0.12s (Model prediction + SLIC algorithm + ruleset prediction)

Segmentation network: ~96.5% / ~87% / ~88% accuracy and ~77% / ~35% / ~43.5% mean IOU on the train / validation / test sets respectively.

number of Trainable Params: 4,683,379

What's next ?

After relabeling all wrong images (total accuracy was ~67.3% on 1,420 images so we had to relabel a third, or ~473 images), we trained a classic CNN on the labeled data to see if we could get a more general and accurate algorithm.

Algorithm #2 - CNN

Now that we have labeled data, we'll train a CNN to classify our images into the 8 classes. Before training, we'll split our data into training, testing and validation sets, making sure we use stratified splits to keep the proportion of samples consistent across classes.

The Network

```
conv_model = Sequential()
conv_model.add(Input(shape=(128, 128, 3)))

conv_model.add(Conv2D(32, kernel_size=(3, 3), activation="relu"))
conv_model.add(MaxPooling2D(pool_size=(2, 2)))
conv_model.add(BatchNormalization())

conv_model.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
conv_model.add(MaxPooling2D(pool_size=(2, 2)))
conv_model.add(BatchNormalization())

conv_model.add(Conv2D(128, kernel_size=(3, 3), activation="relu"))
conv_model.add(MaxPooling2D(pool_size=(2, 2)))
conv_model.add(BatchNormalization())

conv_model.add(Conv2D(256, kernel_size=(3, 3), activation="relu"))
conv_model.add(MaxPooling2D(pool_size=(2, 2)))
conv_model.add(BatchNormalization())

conv_model.add(Conv2D(512, kernel_size=(3, 3), activation="relu"))
conv_model.add(MaxPooling2D(pool_size=(2, 2)))

conv_model.add(Flatten())
conv_model.add(Dropout(0.5))
conv_model.add(Dense(8, activation="softmax"))

conv_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Notes about training

After testing, we found that pretty heavy regularization improved performance significantly, especially when we switched from Dropout to BatchNorm.

Batch size also had a major effect on overall accuracy, and after around 25 epochs the network would stop learning and would settle around 78% accuracy.

Overall results of Algorithm #2

Algorithm 2:

	Train / Val / Test
Original Segmentation Dataset:	99.65% / 93.75% / 88%
Stanford Cars Dataset:	99.61% / 80% / 74%
Berkeley Dataset:	97.26% / 63.85% / 74%
Overall accuracy:	98.84% / 79.2% / 78.7%

AVG inference time: 0.03s

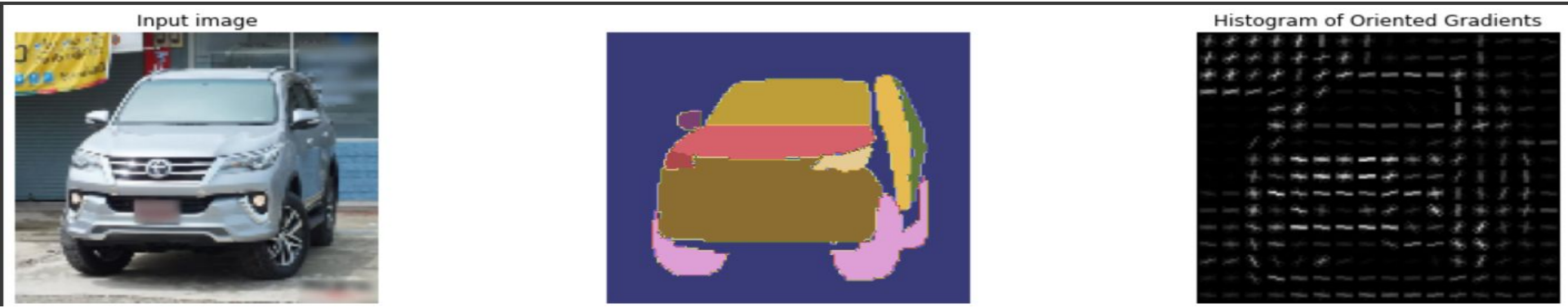
number of trainable params: 1,585,928

Comparison on the test sets

	Original	Stanford	Berkley
Algorithm 1	86%	74%	42%
Algorithm 2	88%	74%	74%
Difference	+2%	+0%	+32%

Things that failed

1. HOG: At one point we tried to use HOG instead of image segmentation to see how it performed, the problem was that we turned a complex segmentation problem into a complex tabular data problem (millions of lines of data, 64 columns each, 19 classes with extreme imbalance between the different labels). We tried to use RF and XGboost on the data, however learning took an extremely long time (hours) and was very inaccurate.



Things that failed

2. Removing redundant classes: We tired to remove redundant classes from our segmentation problem (e.g mirrors, back doors etc.), since these classes would barely be classified anyway. We hoped that removing these classes would allow our segmentation network to learn better, however in practice this did not improve our networks learning ability and even slightly harmed it.

Things we learned / didn't expect

1. Working with image data is hard. Operations are slow, resolutions need to be matched, RAM space is limited so VM sessions crash frequently. In addition, we had to write a significant amount of helper functions to handle even basic operations such as displaying the networks output in image form, or to find out which images our network failed to classify.
2. Experimenting takes a lot of time:
    a. Testing out HOG took a significant amount of time (setting up and preparing data, fitting models, analyzing results etc.) for minimal results.
    b. Trying to remove some classes also took a lot of time, we had to rewrite a lot of our helper functions and predefined variables, and normalize the data since the network expects continues labels.


Thank You
The End