Machine Learning Engineer Nanodegree

Capstone Project

Nitish Puri
August 31st, 2017

Using Deep Learning for Image Masking

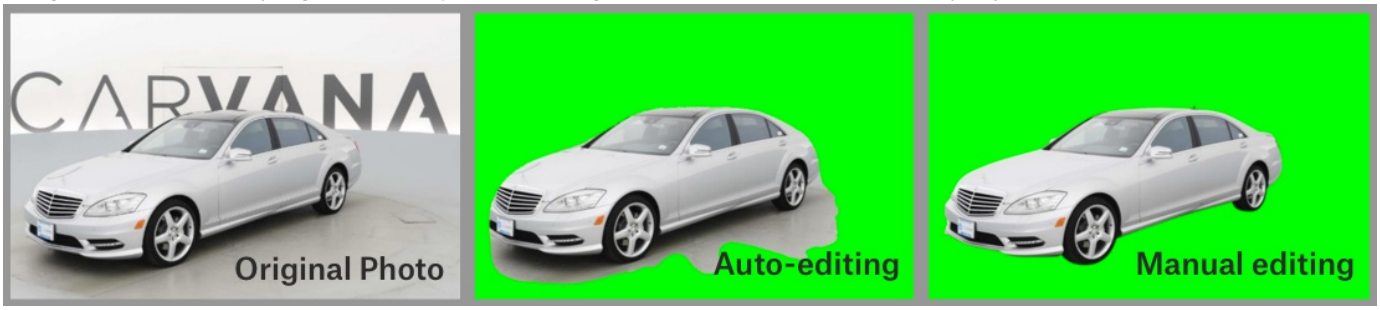
I. Definition

Project Overview

One of the reason I decided to do the Machine Learning Nanodegree was my growing interest in computational perception. There are several closely related perception problems that are being addressed using the rapidly emerging field of deep learning. This project is one variant of the problem of Image Segmenation inspired from the Kaggle Image Masking Challenge.

Historically, the problem of image segmentation has been solved by using traditional Computer vision principles like k-means clustering, thresholding, edge detection and even lossy compression techniques. These techniques require careful engineering of the image pipeline, which can differ a lot between different problems or different datasets.The problem of image segmentation has been studied in various domains ranging from robot perception to medical image analysis and the results obtained and models created are transferable to these domains as well.

The challenge is organized by Carvana. An interesting part of their innovation is a custom rotating photo studio that automatically captures and processes 16 standard images of each vehicle in their inventory. While they capture high quality photos, bright reflections and cars with similar colors as the background cause automation errors, which requires a skilled photo editor to fix. In this project I am going to develop an algorithm that automatically removes the photo studio background. This will allow Carvana to superimpose cars on a variety of backgrounds. I will be analyzing a dataset of photos, covering different vehicles with a wide variety of year, make and model combinations.



Problem Statement

As mentioned in the previous section, we are provided with 16 standard images(1918 X 1280) of each vehicle(318 vehicle categories for the training data) in their inventory. We also have a corresponding image mask(1918 X 1280) for each input image. The problem is in automatically create an image mask for unseen images of automobiles in a similar setting. One potential solution that I could immediately come think of was using Deep Neural Network that would label each pixel of the input image as belonging to background or automobile. The background pixels can then be masked to create an image mask for the given image.
Specifically I will be using a U-net architecture that use Convolution layers to generate pixel level classification on the input image. These architectures are discussed in the following sections in more detail.

Metrics

Dice coefficient can be used as an evaluation metric for the problem. Dice coefficient is a statistic used for comparing the similarity of two samples. Its range goes from 0 meaning no similarity to 1 meaning maximum similarity. It can be used to compare the pixel-wise agreement between a predicted segmentation and its corresponding ground truth. The formula is given by:

$$QS = \frac{2|X \cap Y|}{|X| + |Y|}$$

where X is the predicted set of pixels and Y is the ground truth. The Dice coefficient is defined to be 1 when both X and Y are empty. For example consider these two example 5X5 image masks.

Ground Truth

0	0	0	0	0
0	1	1	1	0
0	1	0	1	0
0	1	1	1	0
0	0	0	0	0

Predicted Mask

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	0	0
0	0	0	0	0

```
Here,
X = 8,
Y = 8,
X & Y = 6

So,
QS = (2 * 6)/(8 + 8)
QS = 0.75
```

This is an appropriate statistic for image segmentation task as it takes into account and penalizes the result for false positives as well as false negatives. This is not the case with the precision or recall statistics. Dice coefficient is mathematically equivalent to F1 score which is the harmonic mean of precision and recall.

The final score can be calculated as the mean of the Dice coefficients for each image in the test set.

We can convert this to a loss function by using

```
dice_loss = 1 - QS
```

This can be augmented with binary_crossenntropy to calculate a more robust loss function.

II. Analysis

Data Exploration

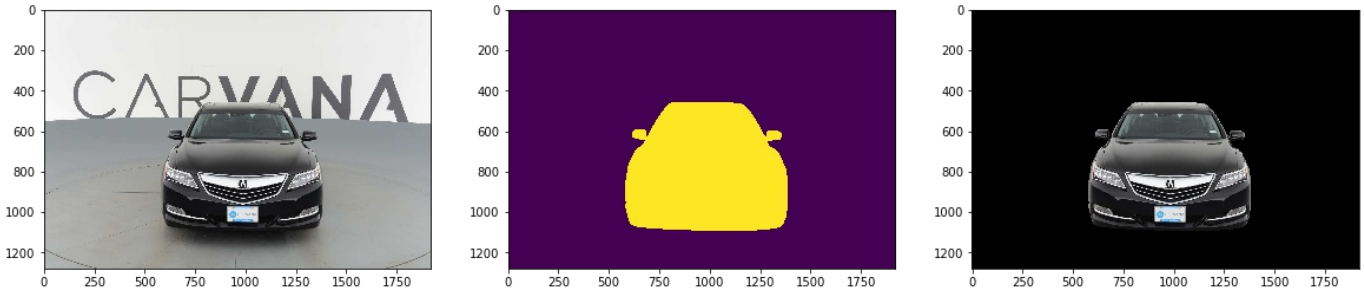
Here are some of the questions that i wanted to answer while progress7ing through this project.

- Can I train a model to predict the manufacturer?
- Background is always the same. how can i take advantage of that?
- How do we/network deal with the reflections?

Since this is a Kaggle competition, we already have separate test and train data available here.
Training data is in the form of 318 vehicle samples with 16 poses each. Each car has an associated metadata,

id	year	make	model	trim1	trim2
0004d4463b50	2014.0	Acura	TL	TL	w/SE
00087a6bd4dc	2014.0	Acura	RLX	RLX	w/Tech
000aa097d423	2012.0	Mazda	MAZDA6	MAZDA6	i Sport
000f19f6e7d4	2016.0	Chevrolet	Camaro	Camaro	SS
00144e887ae9	2015.0	Acura	TLX	TLX	SH-AWD V6 w/Advance Pkg

Corresponding to each car image we have a single channel mask. Our goal is to model a neural network that can learn how to generate this mask image given the RGB input.



Here are some samples shown with masks overlaid,

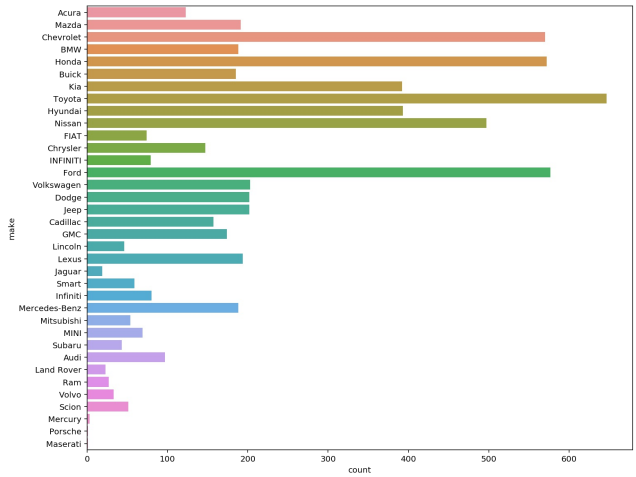
Corrupt data

During random visualization of the dataset i was able to find out a few samples that do not have a (nearly)perfect mask. These anomalies are mostly because of very thin appendages such as spoilers or antennas, regular patterns such as wheel spokes or translucent features such as glasses. These samples may cause issues, and we can try to improve the score by removing these samples from the train dataset. However, I have decided to not remove these samples and rely on data augmentation to provide regularization effects against these samples.

Exploratory Visualization

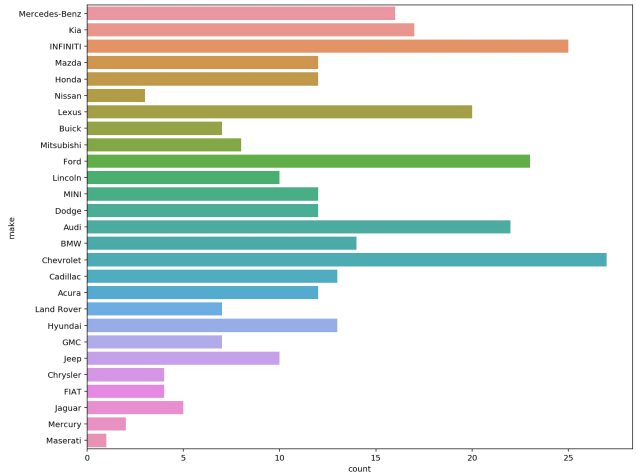
Car manufacturer distribution

Also, we can get a distribution of car manufacturers in the complete dataset.

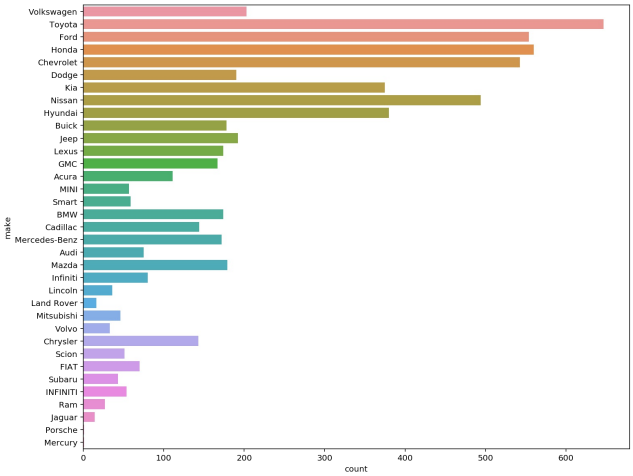


These results show the distribution in train and test sets.

Training data



Test data

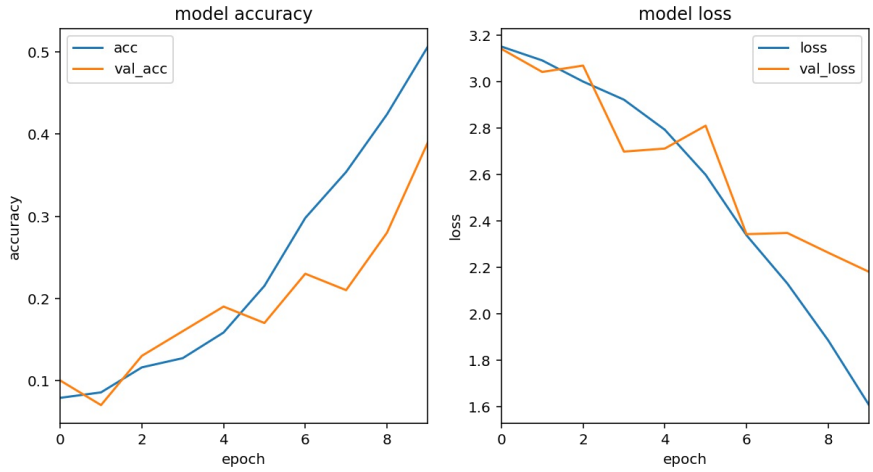


As a *side quest* I decided to train a very crude model that can predict car manufacturer given a car image. Here is the model summary,

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 126, 126, 16)      448
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 63, 63, 16)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 61, 61, 32)        4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 30, 30, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 12, 12, 128)       73856
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
global_average_pooling2d_2 ( (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 36)                4644
=================================================================
Total params: 102,084
Trainable params: 102,084
Non-trainable params: 0
_________________________________________________________________
```

We use categorical_crossentropy loss and optimize this model using Adam.

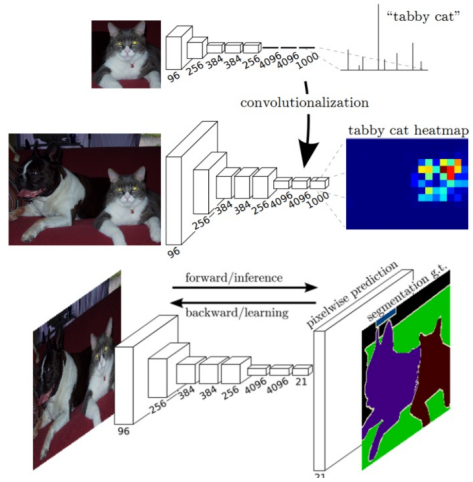
Here is the learning graph for the model.



This model can also be improved by using data augmentation methods used in the following sections.

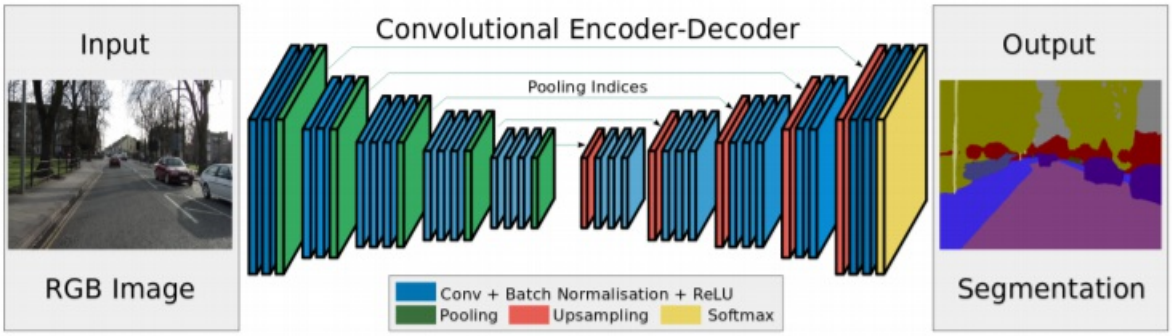
Algorithms and Techniques

CNNs have been known to perform well for image recognition problems. The next step after classification is segmentation, which has its applications in autonomous driving, human-machine interaction, computational photography, image search engines, augmented reality and medical image diagnostics to name a few. A brief overview of various deep learning techniques is done here. The main idea behind these networks is to output spatial maps instead of classification scores. This is accomplished by replacing *fully connected* layers with *convolution* layer. Here is an illustration from one of the earlier works to use this technique, Fully Convolution Networks for Semantic Segmentation.

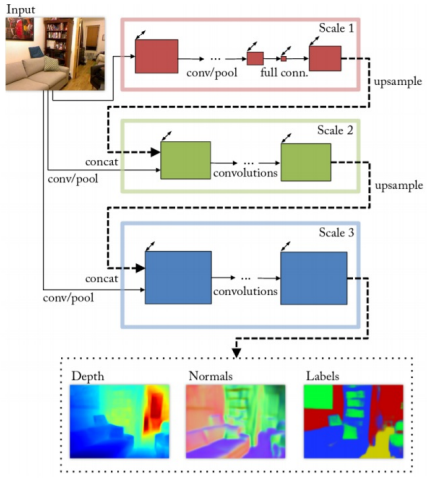


Despite the power and flexibility of the FCN model, it still lacks various features which hinder its application to certain problems and

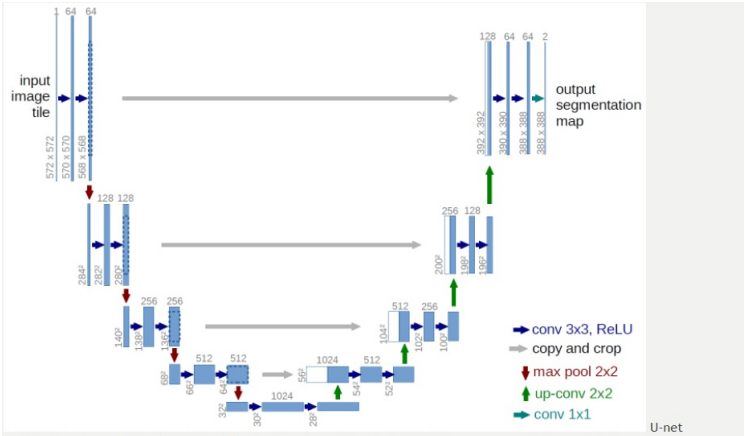
situations: its inherent spatial invariance does not take into account useful global context information, no-instance awareness is present, efficiency is still far from real time execution at high resolutions, and it is not completely suited for unstructured data such as 3D point clouds or models.
There have been various different approaches to address these issues, viz. SegNet which uses Encoder-Decoder type network to output a high resolution map.



These methods have received significant success since fine-grained or local information is crucial to achieve good pixel-level accuracy. However, it is also important to integrate information from the global context of the image to be able to resolve local ambiguities. Vanilla CNNs struggle to keep this balance, pooling layers being one of the sources that dispose of global context information. Many approaches have been taken to make Vanilla CNNs aware of the global information. One such approach is multi-scale aggregation.



Here, inputs at different scales is concatenated to output from convolution layers and fed further into the network.
In this project we are going to use *u-net*, a variation of the above approach for training and inference. This model has been applied to medical image segmentation on data with segmentation masks. This is very similar to our problem and is a simplified version of the general problem of image segmentation and multiple instance learning with multiple(hundreds) classes.



U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Benchmark

As a benchmark model we can use a simplified CNN architecture containing only a couple of fully convolution layers, without any pooling or

downsampling. We use this simple architecture to show that even such a simple model improve its results on the given problem. Though the size of the network and number of learnable parameters may not be sufficient to learn the complexity of the problem at hand.
Here is the benchmark network,

```
baseline_model = Sequential()
baseline_model.add( Conv2D(16, kernel_size= (3, 3), activation='relu',
                    padding='same', input_shape=(INPUT_SIZE, INPUT_SIZE, 3)) )
baseline_model.add( Conv2D(32, kernel_size= (3, 3), activation='relu', padding='same') )
baseline_model.add( Conv2D(1, kernel_size=(5, 5), activation='sigmoid', padding='same') )

baseline_model.compile(Adam(lr=1e-3), bce_dice_loss, metrics=['accuracy', dice_coef])
```

where, INPUT_SIZE = 128 while training for the benchmark. And, the model summary,

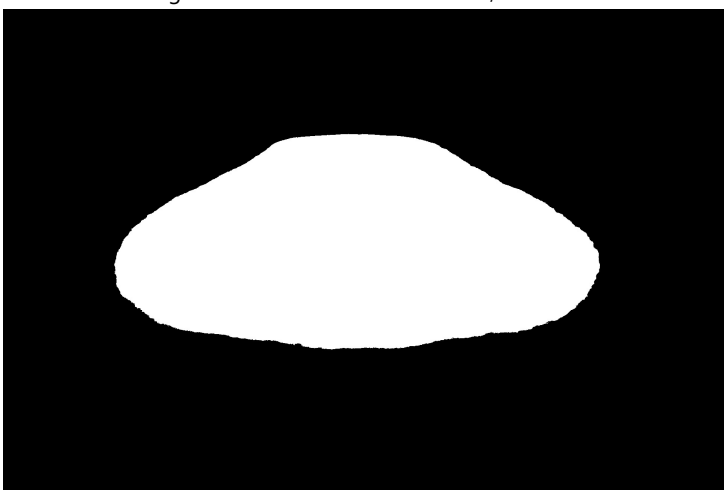
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 128, 128, 16)      448
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 128, 128, 32)      4640
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 128, 128, 1)       801
=================================================================
Total params: 5,889
Trainable params: 5,889
Non-trainable params: 0
_________________________________________________________________
```

However, before discussing further about this benchmark, I would like to show another *simple* benchmark model.

Another Benchmark

We can also use a second benchmark, not based on an ML based model. Here we take samples from the training set and calculate an average mask from the corresponding masks. Now this average mask can be used as a predicted mask for each test data. This benchmark is taken directly from a kaggle post.

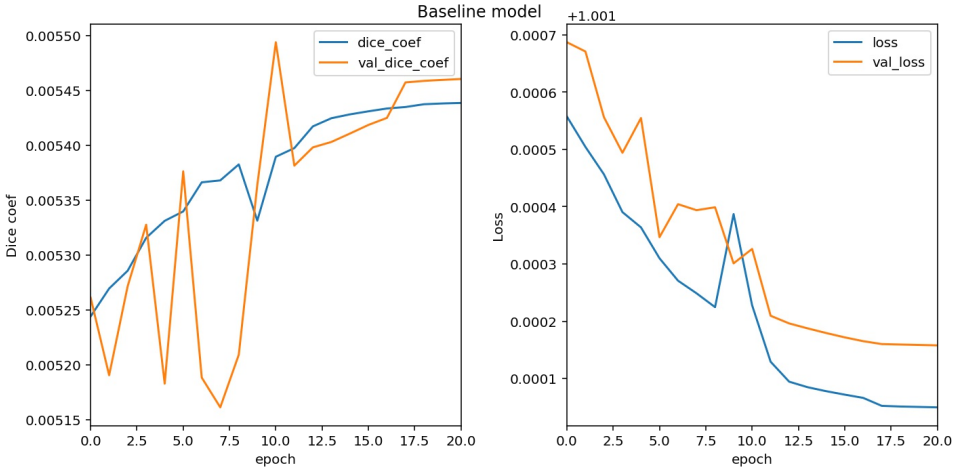
Here is the average mask created as a benchmark,



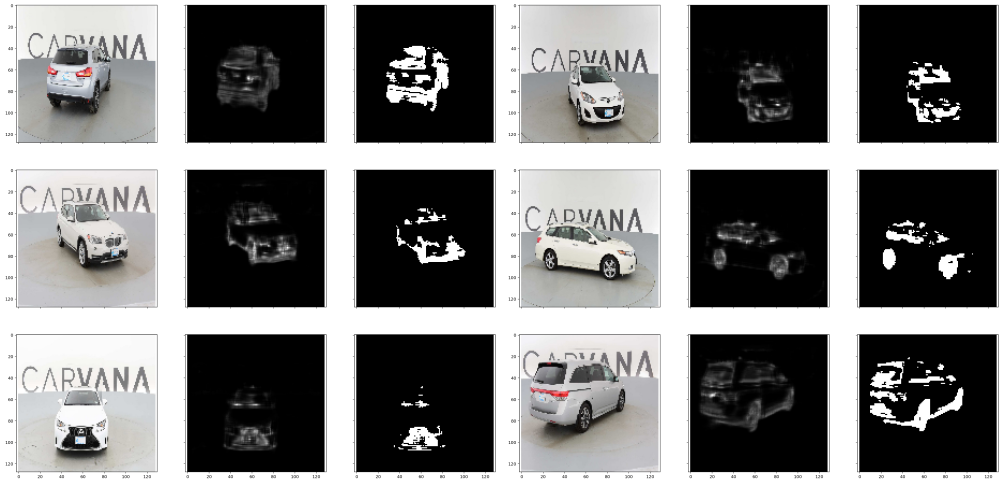
This mask gets a score of **0.7491** on the Kaggle public leaderboard and a score of **0.743401** on the validation set(600 random images from validation set).

Coming back to our benchmark model

Here are the results obtained after *20 epochs* of the above model.



Training for this model is done without using any of the augmenting functions discussed later in preprocessing step. This model took ~5 min per epoch on NVidia GTX 960M with 2GB Memory. The model received a public score of **0.8848** on Kaggle and **0.889419** on our validation set. Here are some predictions visualized from the network(*Input, Softmax prediction, Prediction>0.001*).



Here, we can see that even this simple 3 layer model is able to detect a general shape of the automobile. Also, this simple model shows a significant improvement over the previous benchmark score of **0.7491**. This proves that using CNN's can be fruitful for the problem at hand.

III. Methodology

Data Preprocessing

The training data comes in the form of 1918X1280 RGB images. Our U-Net model is composed of Fully Convolutional layers, which expect the input data in the form of 3D Matrices(multiple 2D stacked layers).So, there is no necessity for preprocessing the data for the model to start learning.
However, we need to consider other problems like computational complexity and overfitting.

- Convolution networks can be very compute intensive, and running through the full res input images would take a lot of time to converge. So, we start with a low res input (128X128) to see if it offers any improvement over the baseline models.
- To help prevent large updates and variations in our gradients, we generally use data with zero mean and unit variance. Here, we have decided to not use that approach as our first step and start with a crude approximation to just normalize the input imagery so that its range is now between (0, 1).
- We have 5088 samples in our training data and 100064 samples in the test set. This shows that the training set *might* not be able to provide a good range of variations in the inputs. We can try to overcome this issue by augmenting our training data using random shifts.
    - Randomly shift pixel values in HSV space. This is done to add variations in the image in the form of tint and brightness. Using HSV space for this is generally better because it is more uniform i.e. less noisy w.r.t. local changes, and also slight shifts in HSV space might look more *natural* than similar shifts in RGB space.
    - Randomly scale and rotate the input image while preserving the input dimensions. This would also help us make our model invariant changes in the scale(or distance from camera) of vehicle. We need to make sure that we don't use very large values

which might cause the automobile to get cropped or change the orientation drastically.

- We introduce a random horizontal flip to the inputs. Here are some samples produced after augmentation.

Implementation

The project is implemented in python 3 using Keras for building and training neural network and OpenCV for image processing. Different parts of the project are divided into modules in the form of .py files. These are explained below,

- data.py : Utility for reading in csv files for metadata, train masks and listing testing and training datasets.
- encoder.py : Utility for converting image mask to *run length encoding*.
- filename.py : Utility for getting car id and angle id from filename and vice versa.
- generator.py : Generator functions used while training the network.
- image.py : Utilities to open and show image given car code and angle code.
- losses.py : Accuracy and loss functions for segmentation model. Defines Dice accuracy and loss function.
- models.py : Functions to generate the UNet model, baseline model and manufacturer model.
- params.py : Some global parameters and constants used throughout different files and utilities.
- preprocess.py : Functions to add random variations to a given image for data augmentation.
- read_activations.py : Generate and display activations produced by the given input in the intermediate layers of the model.
- vis.py : Provides various visualization utils for data exploration, plotting results and visualizing model filters.
- zf_baseline.py : File taken from a Kaggle kernel to produce avg baseline mask.

The project is initially implemented in *Jupyter Notebooks* for exploration and then plugged into a console program using run.py.

Training summary

```
def trainUnet128Model():
    unet_model = models.get_unet_128()

    callbacks = [ModelCheckpoint(filepath='models/unet_128.best_weights.hdf5',
                                 monitor = 'val_loss', verbose=2, save_best_only=True),
                 CSVLogger('./logs/unet_128_history.csv'),
                 EarlyStopping(monitor='val_loss', patience=8, verbose=1, min_delta=1e-4),
                 ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=4, verbose=1, epsilon=1e-
4),
                 TQDMCallback()]

    steps_per_epoch = int(np.ceil(float(len(train_images)) / float(BATCH_SIZE)))
    validation_steps = int(np.ceil(float(len(validation_images)) / float(BATCH_SIZE)))

    unet_model_history = unet_model.fit_generator(generator = train_generator(),
                         steps_per_epoch = steps_per_epoch, verbose = 0,
                         epochs = 100, validation_steps = validation_steps,
                         validation_data=valid_generator(), callbacks = callbacks)

    return unet_model_history
```

The above code snippet shows the method trainUnet128Model() from train_val.py. This method summarizes the complete trining process along with the relevant parameters.

- Get the model from get_unet_128() defined in utils/models.py. Model architecture is explained thoroughly in the *Model Architecture* section.
- Add callbacks for monitoring, backup and tweak the learning process.
- Start the learning process with data generated using train_generator() and valid_generator() defined in utils/generator.py for a maximum of 100 epochs.

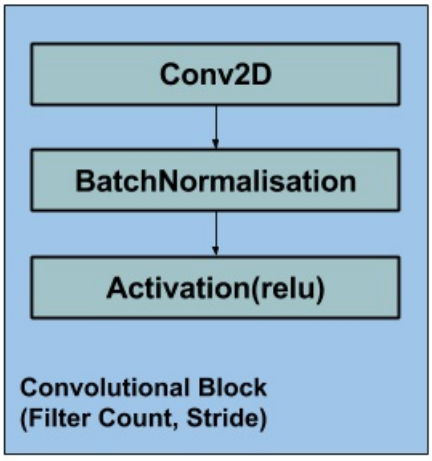
Metrics

As previously discussed, we are using dice coefficient as our metric for accuracy. This is converted to a loss function by using dice_loss = 1 - dice_coeff.

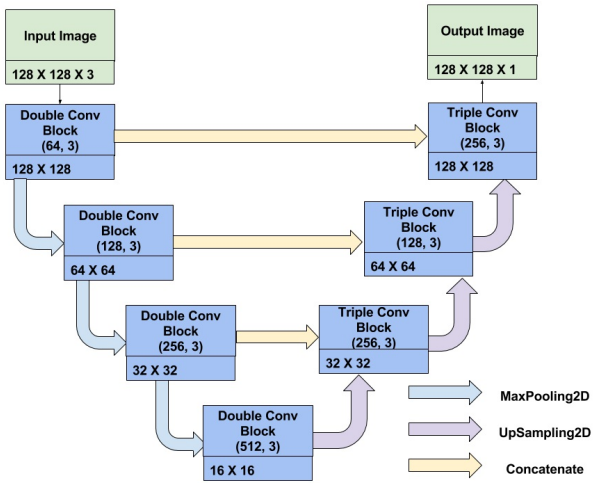
These metrics are implemented in losses.py.
I was getting some weird type mismatch errors if i used the same implementation of dice coefficient while calculating it manually, so I had to keep two different implementations of the function, one that uses Keras/tfconstructs, and one that uses numpy constructs.

Model Architecture

For our final model implementation we are going to use a UNet based model as discussed previously. The model is defined in models.py.
Here is a high level illustration for the model,
This is a simple **Conv block** consisting of a single convolution layer followed by a batch normalization layer and a relu activation layer.



These conv blocks can be chained together to form *double* or *triple* convolution blocks which I am using in the following illustration.

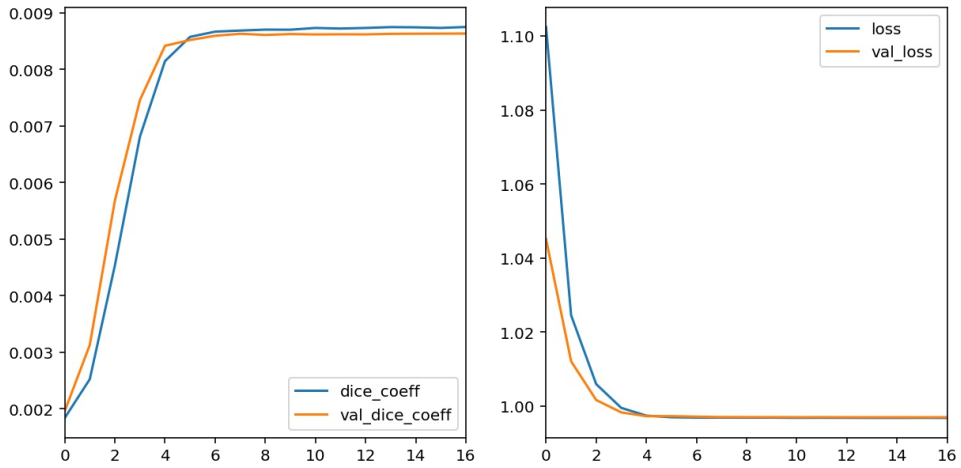


The UNet architecture is composed of encode-decode architecture along with concatenation from the previous high res layers. The last layer of the model is a (1 X 1) convolution followed by a sigmoid activation function. We train this model using RMSprop with an initial learning rate of 0.0001.

The initial model that I implemented consisted of deeper layers with 1024 filters. However my GPU was not able to handle that kind of memory even with a batch size of 8. So, I decided to use remove those layers from in between and use a center layer of 512 filters instead, with 16 samples per batch.
Further, I added the following callbacks to the fit function to monitor the learning process,

- ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=4, verbose=1, epsilon=1e-4) : This reduces the learning rate by a factor of 0.1 if val_loss does not improve by more than 1e-4 for 4 iterations.
- EarlyStopping(monitor='val_loss', patience=8, verbose=1, min_delta=1e-4) : This callback stops the learning process if val_loss does not improve by 1e-4 for 8 consecutive iterations.

Using this configuration, I was able to achieve the following learning graph.



Here we can clearly see that the model did not improve much after the first 4 epochs, and stopped after 20 epochs.

Note: The dice coefficient score that we see here is actually a score based on sigmoid activations from the last layer. For generating the final mask, we used a threshold value of 0.001 generated by trail and error. Because of such low values, this dice score is not in the same scale as the defined metric. However, it is still an acceptable representation for calculating the loss during training. For validating our model we have two options,

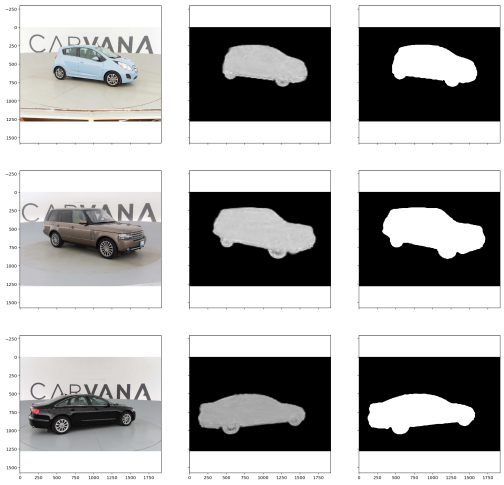
- Manually score the predictions on validation set. *We only use this score because of the absence of labeled test data.*
- Score on test data from Kaggle public leaderboard. We are going to use both these options.

IV. Results

Model Evaluation and Validation

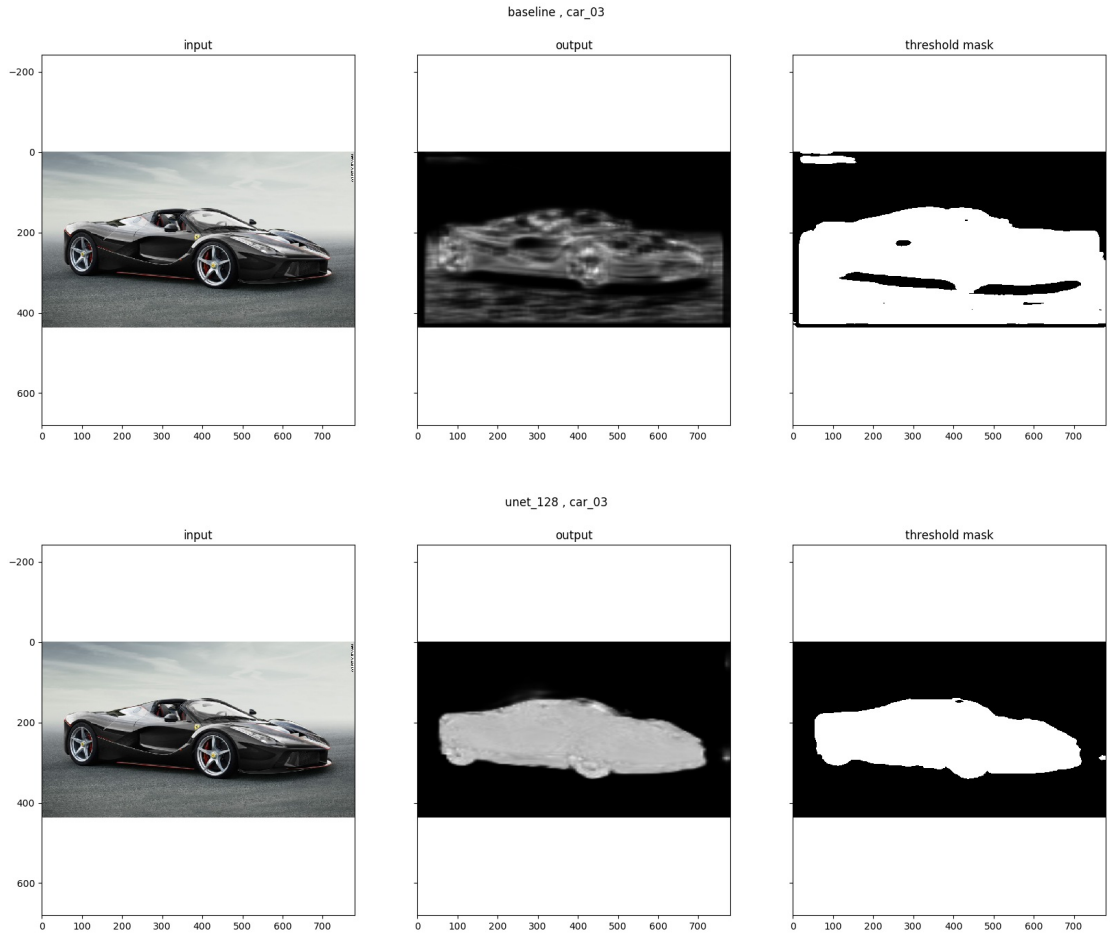
Since this is a Kaggle competition, the test data that we have is not labeled, i.e. we don't have correct output masks available for the test dataset. So, for evaluating our model we can either remove some data from the training set and keep aside for testing(before the train-validation split). However, I decided to use the score provided by Kaggle public leaderboard for model evaluation. To do this, I needed to predict the masks for each of the images available in the test set(/input/test), and convert them to rle encoding for submission. This model achieves a score of **0.9886** on the public leaderboard and the final score of **0.989057** on the sample validation set.

Masks generated by our final model.
Original Image, Sigmoid Prediction, Prediction with threshold(0.001)

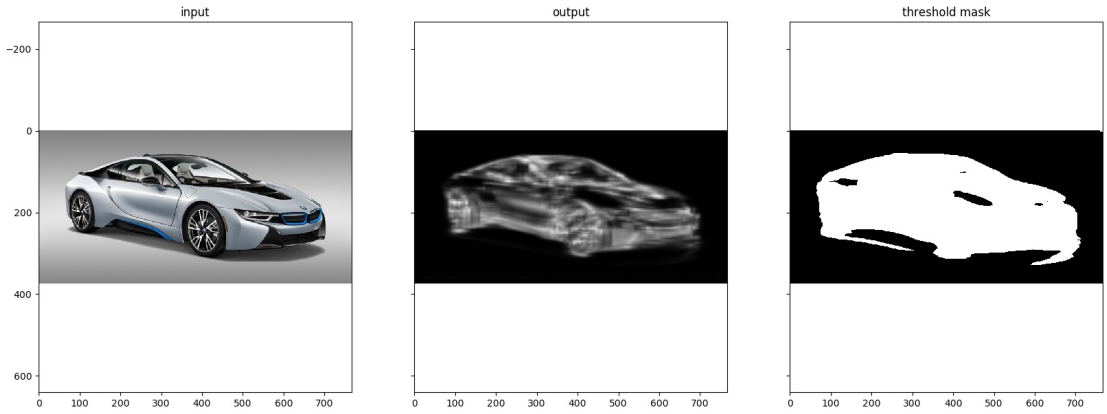


The masks produced by our final model look reasonable to the naked eye, and they are a huge improvement over our last result using a simple 3 layer CNN model.

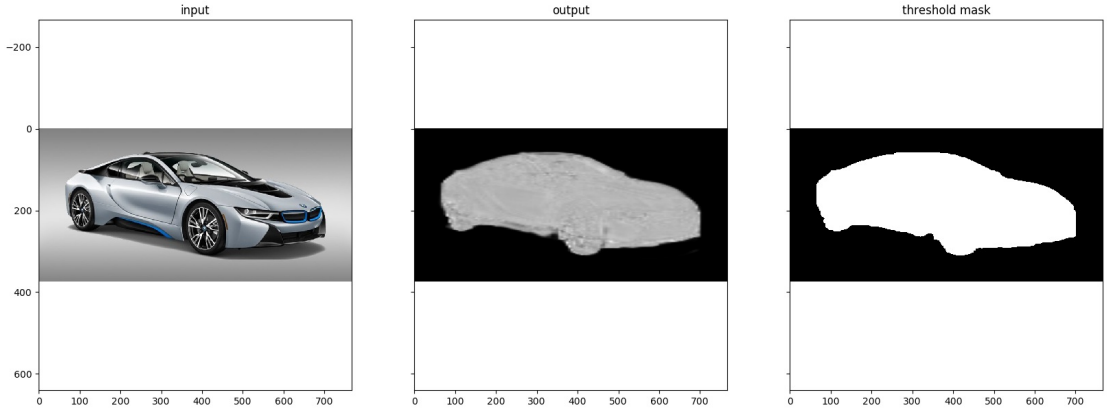
Further, we can also validate how good our model is in generating masks for images external to the dataset. This is not a part of the kaggle challenge and hence the input images are just some freely available stock photos taken from google images.



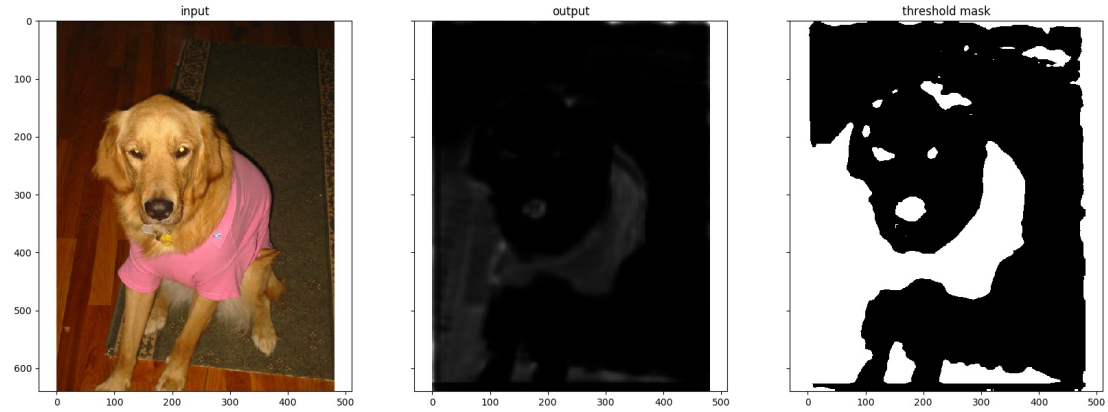
baseline , car_08



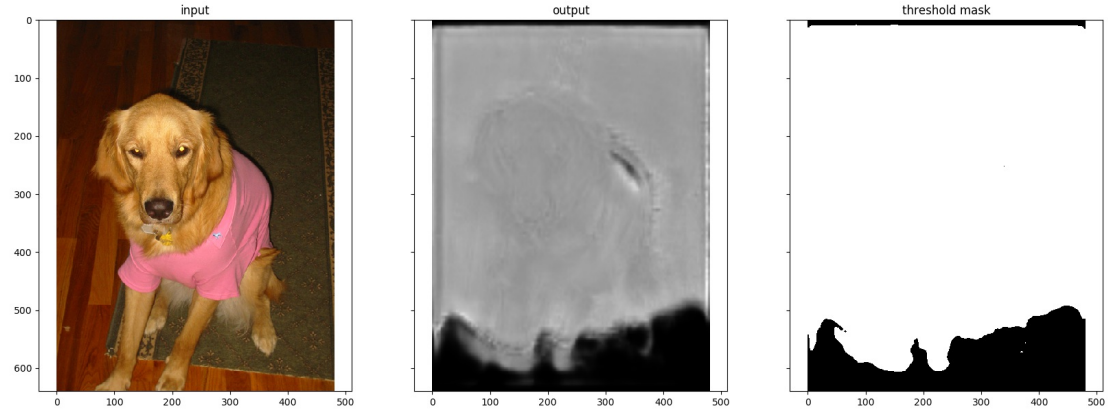
unet_128 , car_08

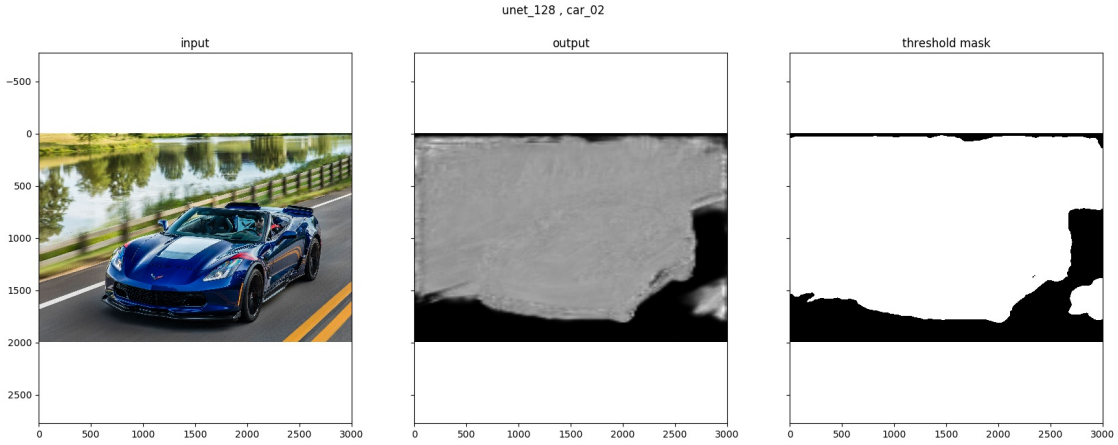


baseline , notcar_08



unet_128 , notcar_08





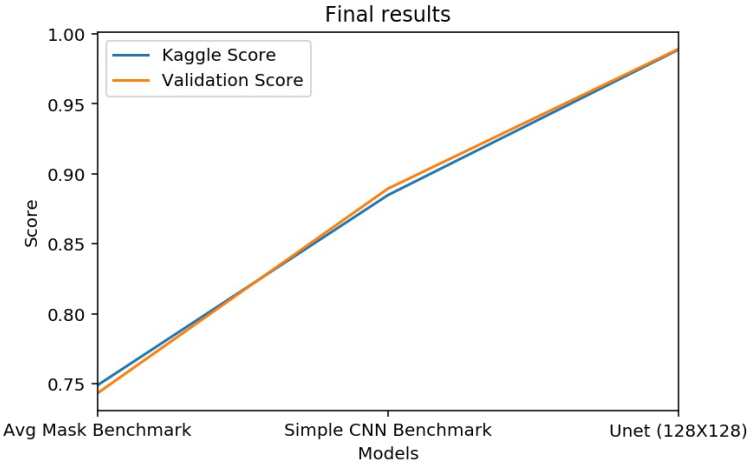
More results can be seen by running the program.

Here we can see that the model is able to segment a vehicle, given the background has a relatively uniform colors, with no particularly noticeable features.

With the given model performance, we can use it for the original problem of finding optimal masks for vehicles photographed in a studio setting, which would then be used for overlaying models in various different settings for demonstrative visualizations.

Justification

We can now plot our final scores.



This shows a clear improvement of our selected model over the specified benchmark models.
The final solution does solve the original problem. However, better overall performance can certainly be achieved by just using higher resolution input images(this is discussed further in the *Improvement* section).
Moreover, this model is not suitable for dealing with *real world* images with non uniform backgrounds, but this problem can also be minimized if we train on *real world* data, or maybe augment our studio based data with outdoor scenery/landscapes and train on that. This can be another extension for the project.

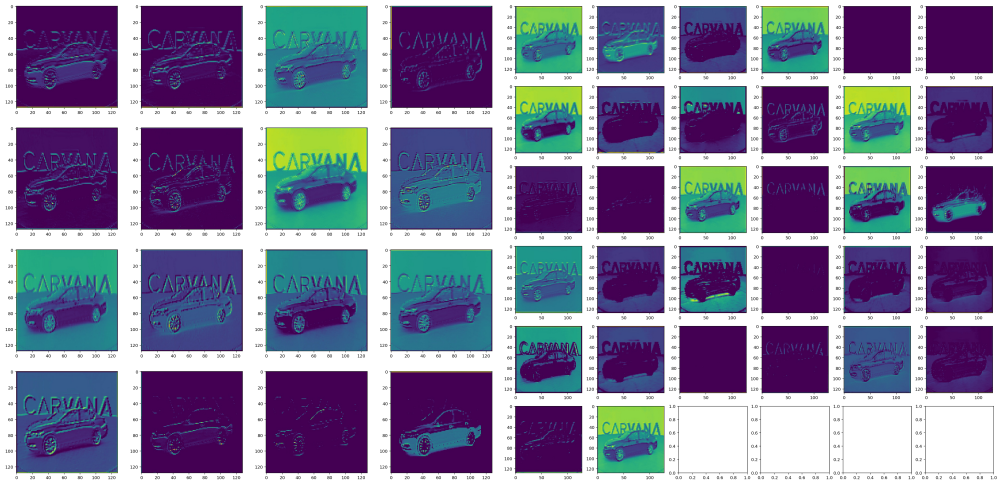
V. Conclusion

Free-Form Visualization

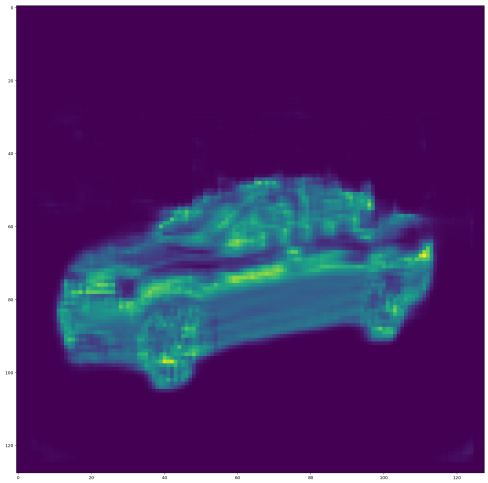
One of the major issues with Neural networks is that of reasonability, i.e. given the model architecture and weights it is very difficult to reason *why* the model behaves the way it does. Some of these issues can be addressed by visualizing the intermediate layer activations for a given input.
In this section we provide some of the intermediate layer activations for the baseline model and our final unet model.

Baseline activations for a sample image

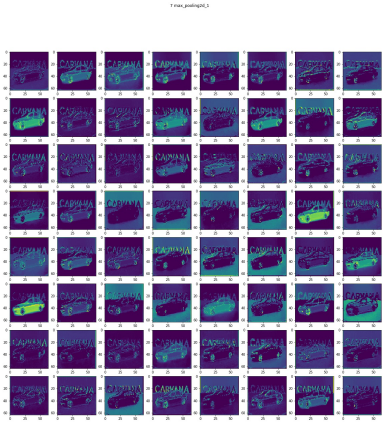
Layer 1 *Layer 2*

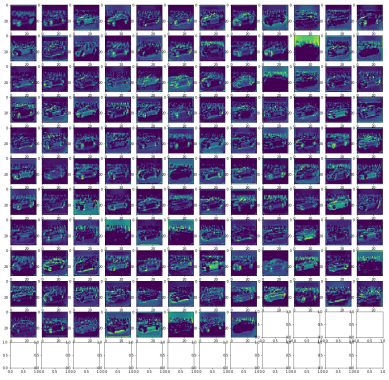


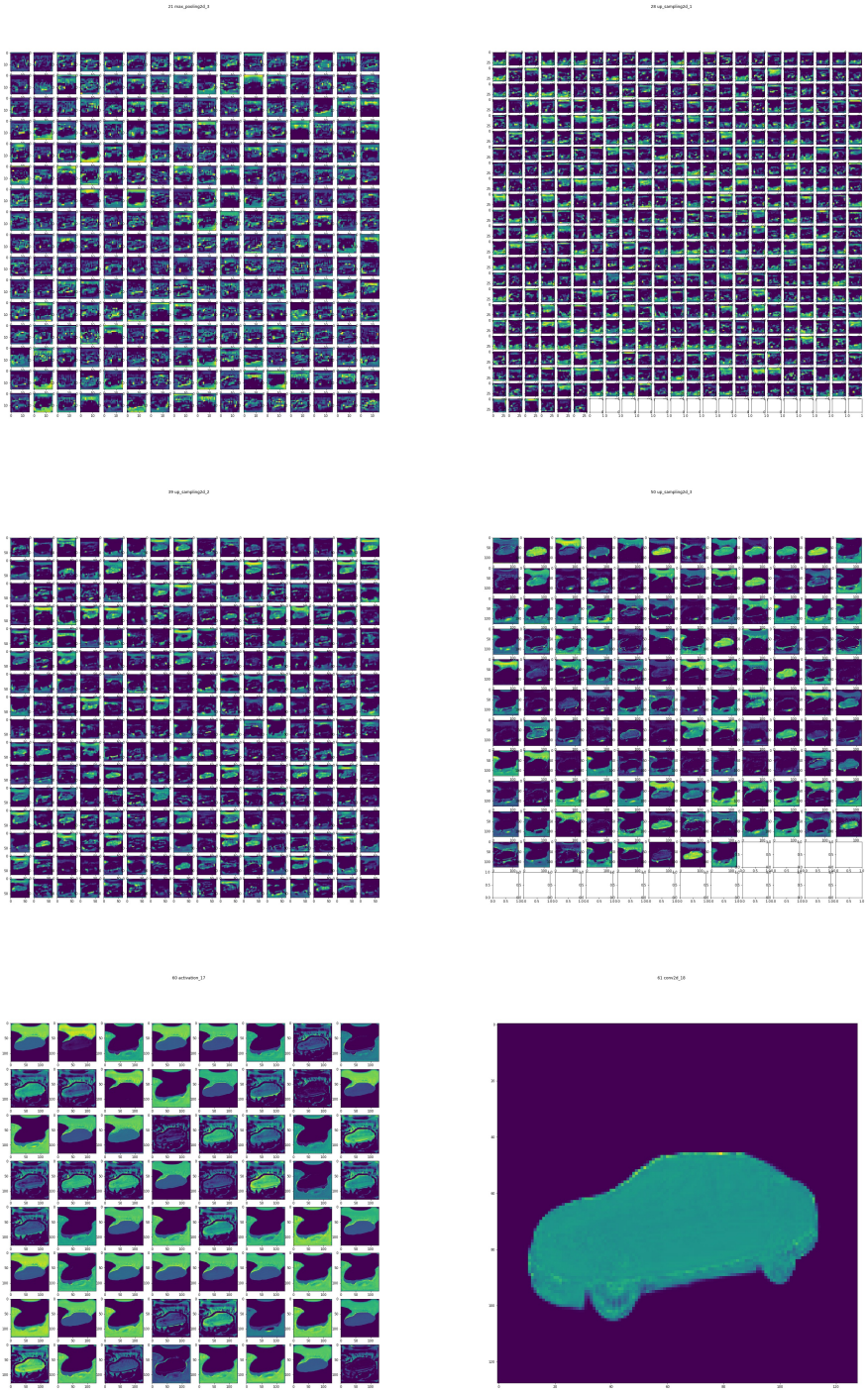
Final Output Layer



UNet 128 activations for a sample image(only pooling and upsampling layers are shown here)







These images show that the filters are progressively extracting a more generalized shape from the sample, instead of hard features.

Reflection

In this project, I tackle a simpler form of the more general object segmentation problem found in many computer vision applications. The approach I have taken is also extendable to those problems.
We are provided with input images of automobiles photographed in a studio setting. Sp, the images input images have a uniform background. This also makes the learning process much simpler than the general problem. I use a UNet based CNN model to learn pixel-wise mask from a given image. This helps in maintaining local information about image regions and hence can give much better results for segmentation tasks than vanilla CNNs. Also, we augment this data to provide some invariance towards image(background) color as long as it is uniform, and towards variations in scale and rotation of the automobile. We have used a very low resolution input image to make these predictions and still were able to get a *good* score.

Here are the final results,

Model	Kaggle Score	Validation Score
Baseline : Avg Mask	**0.7491**	**0.743401**
Baseline : Simple CNN	**0.8848**	**0.889419**
Unet (128X128)	**0.9886**	**0.989057**

U-Nets and CNNs in general are taking over the domain of computer vision applications and are expected to drive the domain much further in the coming future.
One of the most interesting parts about this project was to see how well even simple CNN models are able to learn computer vision tasks. Also, during the implementation phase I became much more familiar with some salient aspects of neural network API's in general and this knowledge would be immensly useful in future projects.

The final model produced here is well beyond my initial expectations, with room left for scope of improvement. I would say that model can be used in the real world application of this problem with some of the improvements mentioned in the next section to achieve a better result in even higher resolution imagery.

Improvement

In the current form, there are a lot of improvements that can be employed to improve the score on the test set,

- Use higher resolution input images. The original input image and masks are 1918X1280 resolution. The problem setters(from Kaggle) have provided a separate dataset of even higher input images. Using higher resolution inputs would certainly improve the final score. However, we might have to add layers more hidden layers to our model to accommodate a higher size input to learn.
- We could use more sophisticated neural network architectures, viz. R-CNN and its derivatives, particularly Mask R-CNN.
- There are pre-trained SegNet and ResNet models available that can be built upon for image segmentation tasks. We can leverage the initial layers of those models to train our network. A combination of one or more of these approaches can certainly give a better result for our problem, however these solutions come with there own complexities in terms of computational resources, which can be dealt with on a certain extent if we use transfer learning.

References

1. Carvana Image Masking Chalenge
2. A Review of Deep Learning Techniques Applied to Semantic Segmentation
3. Convolution Networks for Visual Recognition
4. Fully Convolution Networks for Semantic Segmentation
5. SegNet
6. Multi-Scale Convolutional Architecture for Semantic Segmentation
7. U-Net: Convolutional Networks for Biomedical Image Segmentation
8. DeepLab
9. Rich feature hierarchies for accurate object detection and semantic segmentation
10. Mask R-CNN