Carvana Image Masking Challenge

Project Report

October 1, 2017

I. Definition

Project Overview

This project comes from the e-commerce domain. Carvana is a company that allows consumers to view a catalog of cars and make purchases online. Because their business is focused on selling these cars online, they have a strong interest in how the images of their inventory are presented. Preparation of these images can often required manual intervention by skilled editors, and so Carvana has provided a dataset for this Kaggle competition in the hopes of finding ways of automating part of the image preparation process using machine learning[1].

The dataset consists of car images in jpg format. There are 16 images for each car. In addition to the images, some basic metadata about the car, like year, make, and model, are provided. The dataset was obtained from Carvana through the Kaggle platform. For the training dataset, Carvana has also provided a image in gif format that contains the hand-labeled correct mask of the car. This will allow for the use of supervised learning techniques, since labeled training data has been made available. In addition to the training data, Kaggle has split out a test set for which no labels are provided. For the competition, one is supposed to generate a mask for this test data, and then submit to Kaggle for evaluation on the withheld correct test labels.

Problem Statement

The problem to be solved is image masking for car images. Specifically, the goal is, for each image, to determine a mask (or group of pixels) that contains only the image of the car, and excludes all the pixels belonging to the background. I will attempt to solve this

[1] https://www.kaggle.com/c/carvana-image-masking-challenge#description

problem with the following strategy: first, apply any necessary preprocessing to the images that is necessary for training the model; second, split the data into training and validation sets (the test set is withheld by Kaggle); third, train a model (to be described in more detail later in this paper) using supervised learning with the training images and the provided ground-truth masks; fourth, try to improve the model, and compare the models under consideration using the validation score.

The intended solution for this problem will be a model that can take as input an image of a car, and outputs an image of the same width and height as the input image, where all the pixels from the original that are part of the car are represented as pixels with value 1 in the output image, and all other pixels have value 0.

Metrics

The evaluation metric that Kaggle is using for the competition is mean Dice coefficient.[2] The way that Dice coefficient is calculated is the number of pixels in the intersection of the predicted mask and the ground truth, multiplied by 2, divided by the sum of the number of pixels in the predicted mask and the ground truth. The formula can be seen here:

$$\frac{2 * |X \cap Y|}{|X| + |Y|},$$

where X is the predicted mask and Y is the true labels[3]. The performance of a model is evaluated by the mean of the Dice coefficient on each image in the test set.

The Dice coefficient is useful here because the label and the prediction are both sets (in this case, sets of pixels). The Dice coefficient is often used in the context of comparing a predicted image segmentation to the ground truth mask, for example in a previous Kaggle competition on Ultrasound Nerve Segmentation[4]. It is also used in academic

[2] https://www.kaggle.com/c/carvana-image-masking-challenge#evaluation

[3] https://www.kaggle.com/c/ultrasound-nerve-segmentation#evaluation

[4] https://www.kaggle.com/c/ultrasound-nerve-segmentation#evaluation

papers on segmentation of medical imagery[5]. Because the problem in this project is also a segmentation problem, I believe that the Dice coefficient metric will be ideal for evaluating the model’s performance.

II. Analysis

Data Exploration

The dataset consists of car images in jpg format. There are 16 images for each car. The original images have height 1280 pixels and width 1918 pixels, and consist of 3 channels, representing red, green, and blue.

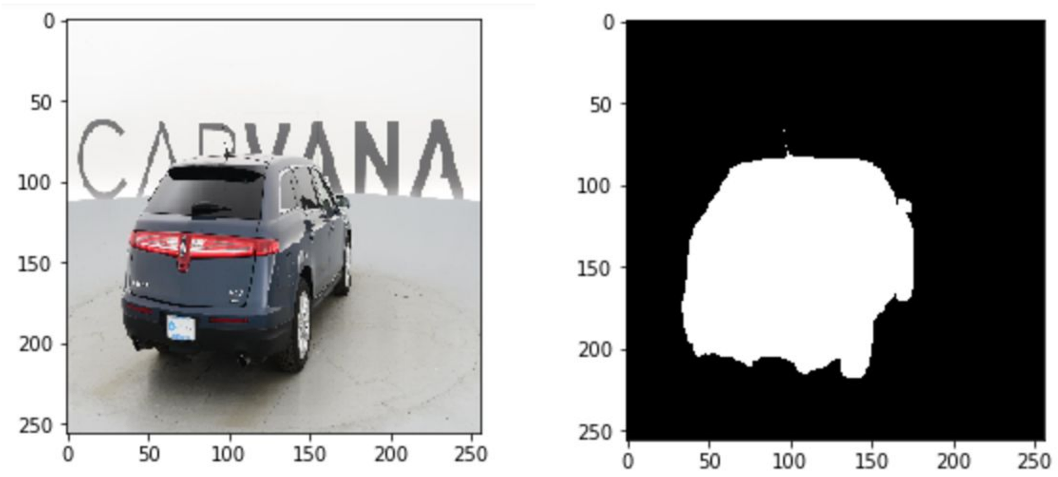
In addition to the images, some basic metadata about the car, like year, make, and model, are provided. The dataset was obtained from Carvana through the Kaggle platform, because Carvana is sponsoring a current Kaggle competition.

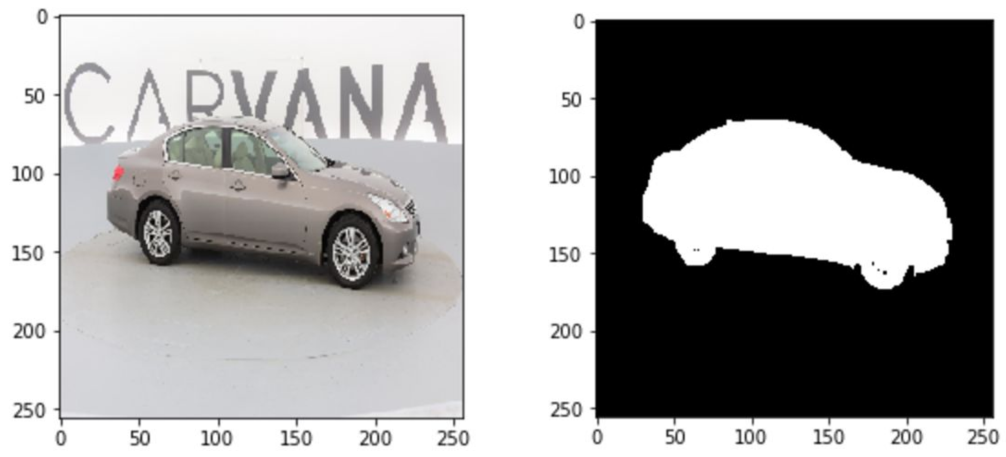
For the training dataset, Carvana has also provided a .gif image that contains the hand-labeled correct mask of the car. This will allow for the use of supervised learning techniques, since labeled training data has been made available. The masks are the same size as the car images, but consist of a single channel, with a value of 1 representing pixels that belong to the correct mask, and a value of 0 representing pixels outside the mask.

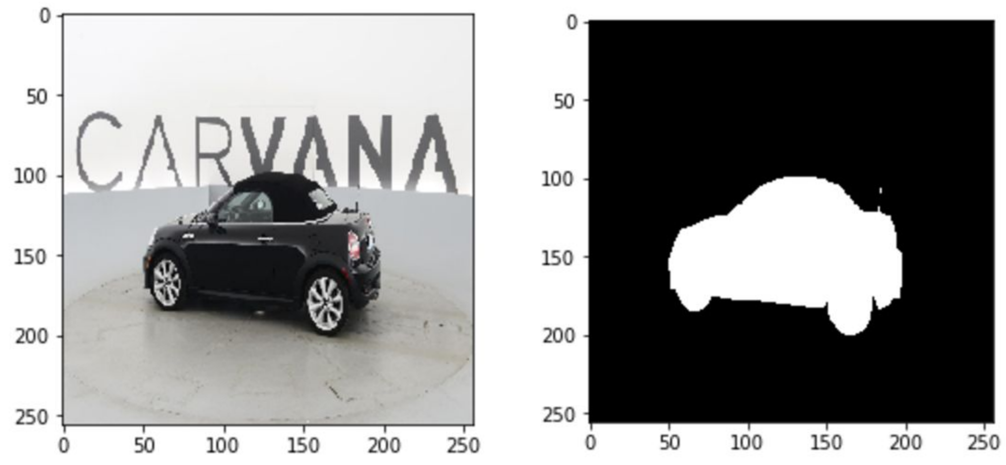
In addition to the training data, Kaggle has split out a test set for which no labels are provided. For the competition, one is supposed to generate a mask for this test data, and then submit to Kaggle for evaluation on the withheld correct test labels.

Here is a sample of the images and ground-truth masks:

[5] https://arxiv.org/pdf/1701.08816.pdf


CARVANA


CARVANA

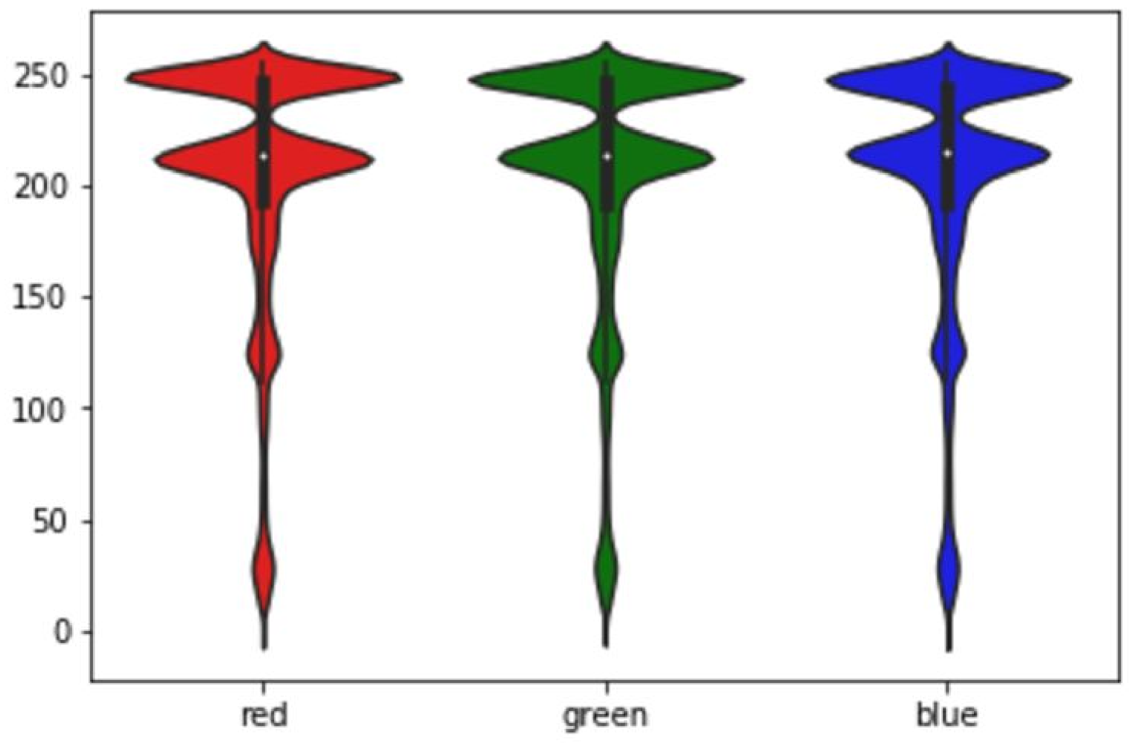

CARVANA

The mean pixel values for each color channel were 178.26337597, 176.48881409, and 174.95852306 for red, green, and blue, respectively.

The fact that there are 16 images of each individual car is worth noting—this is important when splitting the training and validation sets, as including the same car in the training and validation sets might make the validation score artificially high compared to the test score, since the test set does not share any cars of the same make and model with the training set.

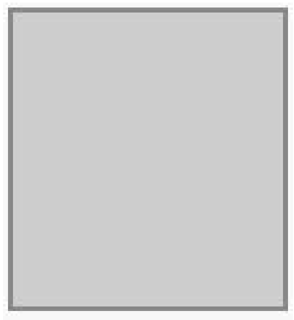
It’s also worth noting the size of the images -- the width and height are larger than the images in other datasets commonly used in machine learning (like MNIST or ImageNet). Because of this, I ended up having to scale down the images in order to achieve efficient training of a model on a GPU.

Exploratory Visualization

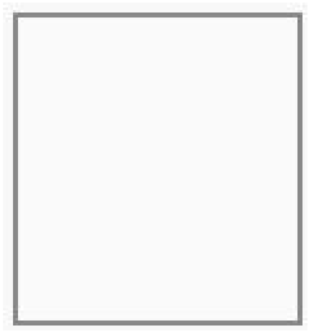


For my exploratory vizualization, I made a violin plot of the pixel intensities for each of the three color channels (red, green, and blue). From this visualization, we can observe some interesting aspects of the dataset. The first thing to note is the similarity of the plots for each color channel. At first, it might even appear as if they were the same plot, but examination of the pixel data shows that they do differ in some places. The similarities in the distribution of intensities is due to the fact that the pictures have large numbers of gray and white pixels, in which the intensities of red, green, and blue are equal (or extremely similar).

Beyond just the similarity of the plots, we can see two large peaks that appear in each plot, one around 205 and the other near 250. Upon examining a few sample images, one can determine the cause of these peaks: the peak around 205 corresponds to the color of the floor, which takes up roughly half the area of each training image. The RGB color for (205, 205, 205) looks like this:



The second peak near 250 corresponds to the other feature that takes up a large area of each image, which is the off-white wall upon which the Carvana name is written in the background of each image. The RGB color for (250, 250, 250) looks like this:



From making this visualization, we've been able to get some insights into what color pixels make up a large part of each training image.

Algorithms and Techniques

The model that I will use for my solution is a U-Net. The U-Net paper was published in 2015 by Olaf Ronneberger, Philipp Fischer, and Thomas Brox, who used their model to achieve state-of-the-art results for a variety of biomedical imaging tasks, like “segmentation of neuronal structures in electron microscopic stacks”[6]. The U-Net is a deep neural network architecture that uses convolutional blocks, making it appropriate for the field of image processing, since convolutional networks currently hold some of the best performances in image-related tasks.

The convolutional blocks of the U-Net are made up of 2 convolutional layers, followed by a pooling layer (in this case, max pooling is used). A convolutional layer is a set of learned filters with a given size (the U-Net uses 3x3 convolutions) that are slid over the image, or the output of the previous network in the layer, in order to generate a 2-dimensional “activation map”[7]. Each weight in the activation map is generated by taking the dot product of the convolutional filter with the pixels from the input that it is currently overlapping.

Pooling layers are inserted between convolutional layers in order to reduce the size of the layer’s input. Specifically, max pooling works by just taking the maximum activation of the set of activations in its filter size[8]. So a 2x2 max pooling with a stride of 2 (with stride being how far to slide it before computing the next activation) would reduce a 256x256xD input to 128x128xD output, where D is the depth, which is unchanged by the max pooling operation.

What the U-Net adds to the standard convolutional network is what the authors call an “expansive path”, which applies upsampling layers to turn the feature map generated by the convolutional layers back into the same size of the image, allowing it to output a segmentation mask. This mask can be compared to the ground truth label in order to measure the model’s performance. Once the model has been trained, and all the weights of the different layers determined, it should predict the same mask for the same image each time.

In order to train the net, we also need to select an optimizer. We will train our U-Net using gradient descent, which is an algorithm can be used to minimize the loss function by calculating the gradients (derivatives) on the network based on the difference between the expected output and the predicted output for a training step, and then using the gradients to take a small step in a direction that will lead to a lower loss. However, there are many different ways of determining exactly how to take that step, which is the job of an optimizer. For this project, I selected Adam, which stands for

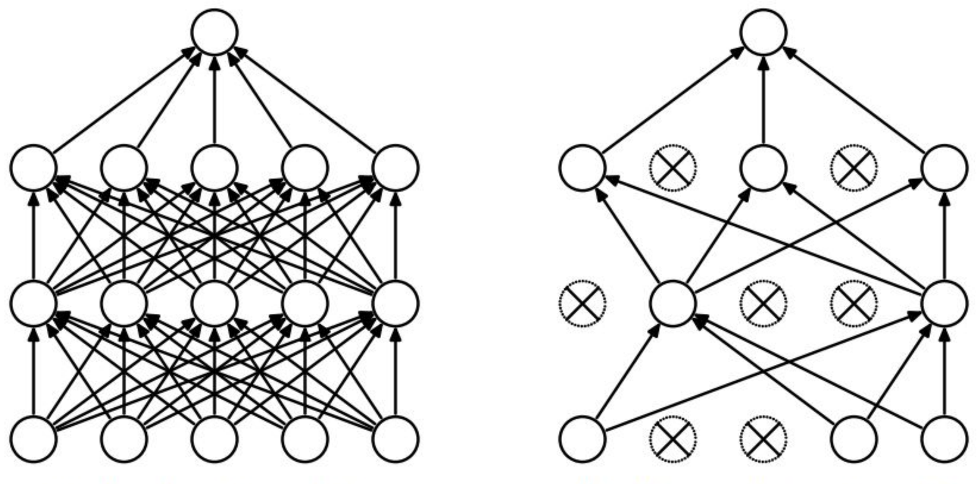
[6] https://arxiv.org/abs/1505.04597

[7] http://cs231n.github.io/convolutional-networks/#conv

[8] http://cs231n.github.io/convolutional-networks/#pool

“Adaptive Movement Estimation”[9]. By using some information about past gradients, Adam is able to efficiently optimize the parameters of the network to achieve a low value for the loss function.[10]

One advanced regularization technique that I explored using is called Dropout. Dropout works in this way: during model training, some of the units of the network are randomly dropped, producing a “thinned” network[11]. By repeatedly dropping different sets of units randomly during training, the network learns to make predictions using different “thinned” networks, and thus does not learn to rely too heavily on certain individual units or certain connections (which can cause overfitting). Then, when testing the net, all the units are included, with their weights scaled in such a way that accounts for the fact that all units are present, rather than having some be absent. A diagram from the original Dropout paper[12] provides a good visual explanation for how this works:



(a) Standard Neural Net (b) After applying dropout.

Benchmark

[9] https://arxiv.org/abs/1412.6980

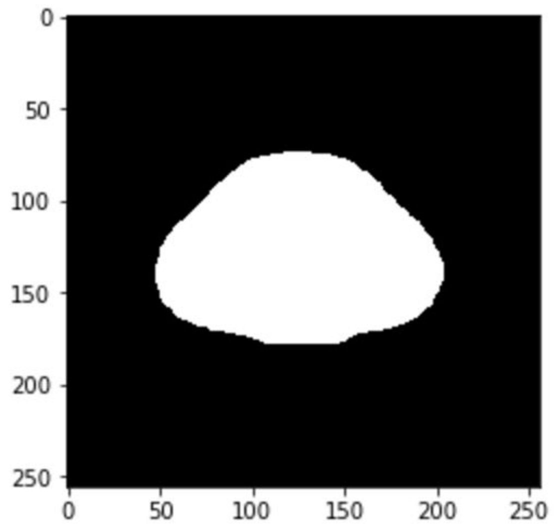
[10] https://arxiv.org/pdf/1609.04747.pdf

[11] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[12] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

The benchmark model that I used for this project was to take the average mask of the training data set. That is, always predict a pixel is in the mask if it appears in the ground truth label of at least 50% of the training images, otherwise predict that it is not in the mask. This makes sense as a benchmark, given that the images have a high degree of similarity, taking the pixels that appear in most of the training images gives a baseline idea of what the mask should look like. Since this benchmark model predicts a mask, we can evaluate the benchmark model using the average Dice coefficient, which we will also apply to our proposed solution.

The average mask of the training dataset looks like this:



The average Dice coefficient for the average mask benchmark model was 0.7387.

Credit for the idea for this benchmark is due to a Kaggle kernel by user ZFTurbo, though the implementation is my own[13].

III. Methodology

Data Preprocessing

For data preprocessing, I subtracted the mean pixel value for each channel. This is a common preprocessing step for machine learning models with images, as it serves to

[13] https://www.kaggle.com/zfturbo/baseline-optimal-mask/code

center the pixel values in each channel around zero[14]. I considered also dividing each pixel value by the standard deviation for further normalization, but found some evidence that this step is not particularly useful for image data. Stanford's CS231n course, "Convolutional Neural Networks for Visual Recognition", teaches "In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step."[15]

I also scaled the images down to 256x256, in order to train the model more quickly by fitting larger batches of images on the GPU.

Implementation

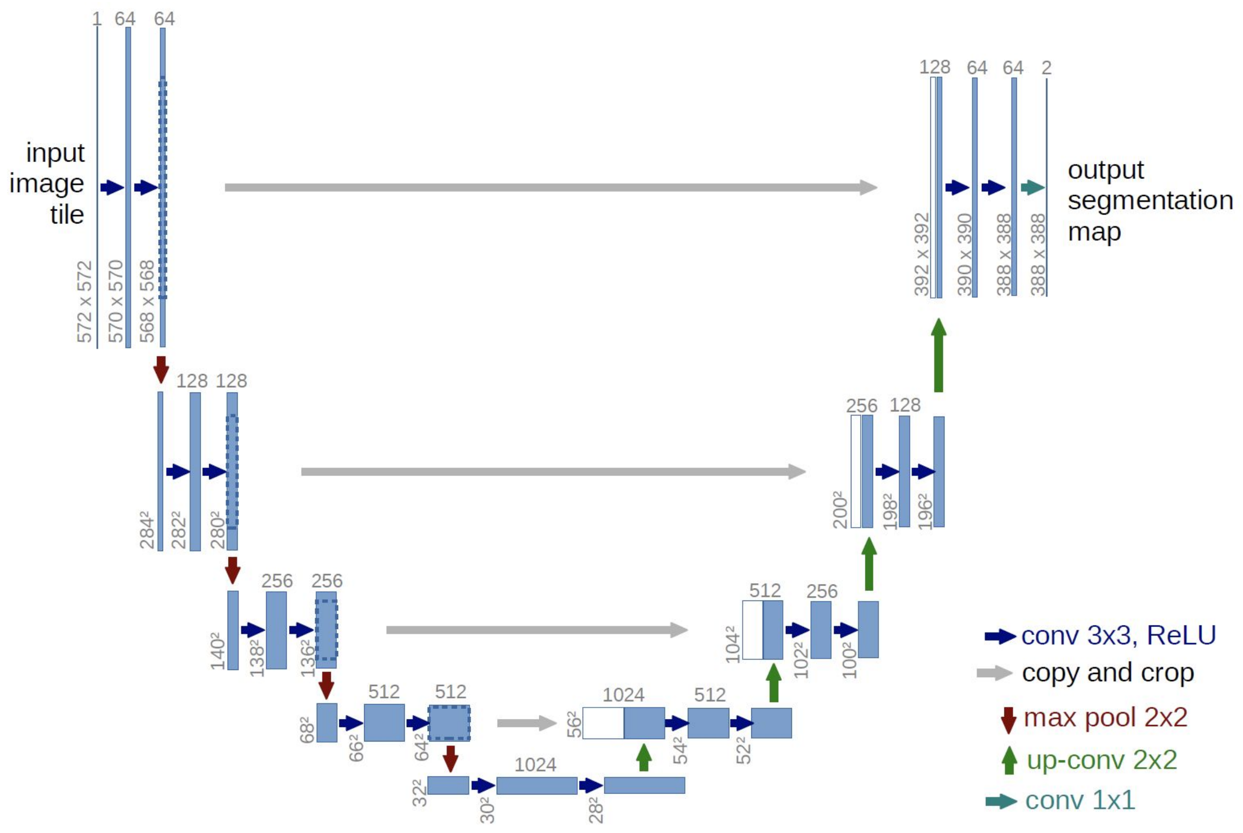
First, I downloaded the data from Kaggle. I split the data into training and validation sets to avoid overfitting—when considering changes to my model's architecture or hyperparameters, I wanted to be able use the validation set to make sure that those changes actually show improvement on data they weren't trained on. Kaggle has already split out a test set, which will be used for evaluating the final model.

After determining the appropriate preprocessing of images, I built the model architecture. As I state in the Algorithms and Techniques section, I am planning on using the U-Net architecture. Here is a diagram of the architecture (from the original authors)[16]:

[14] http://cs231n.github.io/neural-networks-2/#datapre

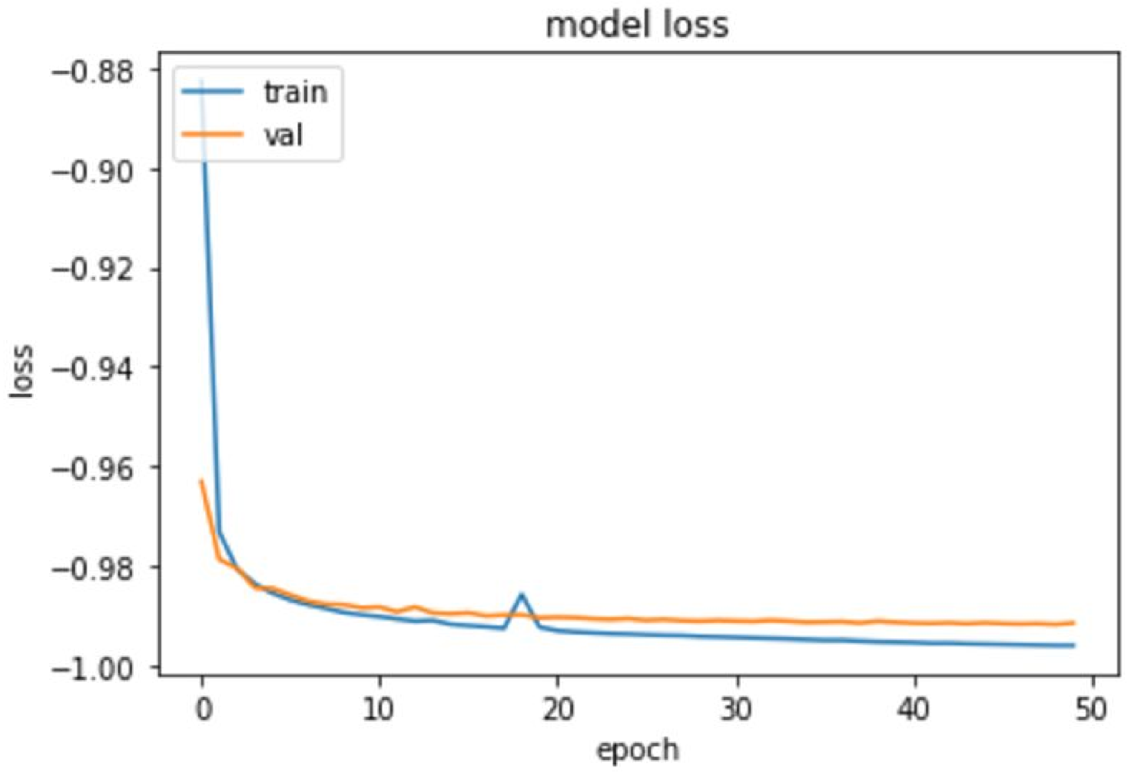
[15] http://cs231n.github.io/neural-networks-2/#datapre

[16] https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/



I used Keras (with Tensorflow backend) to build my model. Keras supports the necessary layers, including Conv2D, MaxPooling2D, and Upsampling2D.

After building the model, I trained the model for 50 epochs using a batch size of 32, with the Adam optimizer and a learning rate of 1E-5. My loss function was negative Dice coefficient. Over 50 epochs of training, the training and validation loss curves looked like this:



The model achieved an average Dice coefficient of 0.9917 on the validation set, representing a significant improvement over the benchmark. However, there still appeared to be possible improvements to the model, as you can seen from the graph above that the model was increasingly overfitting to the training set starting around epoch 8 or 9.

One coding difficulty that challenged me consistently in the implementation phase was keeping track of what each axis of the dataset array corresponds to. In my implementation, the shape of the training data was (4096, 256, 256, 3), which corresponded to (number of images, height, width, channels). But in order to, for example, calculate the mean pixel for each channel, it was initially a challenge to reason about this 4-dimensional array. Additionally, some of the image utilities seemed to sometimes reverse height and width, leading to some challenges in resizing and plotting the images correctly.

Refinement

In order to address this overfitting, I applied two techniques: first, I implemented data augmentation on the training set. Using Keras's ImageDataGenerator, I was able to apply random shifts to the training images, including rotations of up to 15 degrees, height shifts of up to 5% of the image height, width shifts of up to 10% of the original width, shears of up to 0.1 radians, and random horizontal flips.

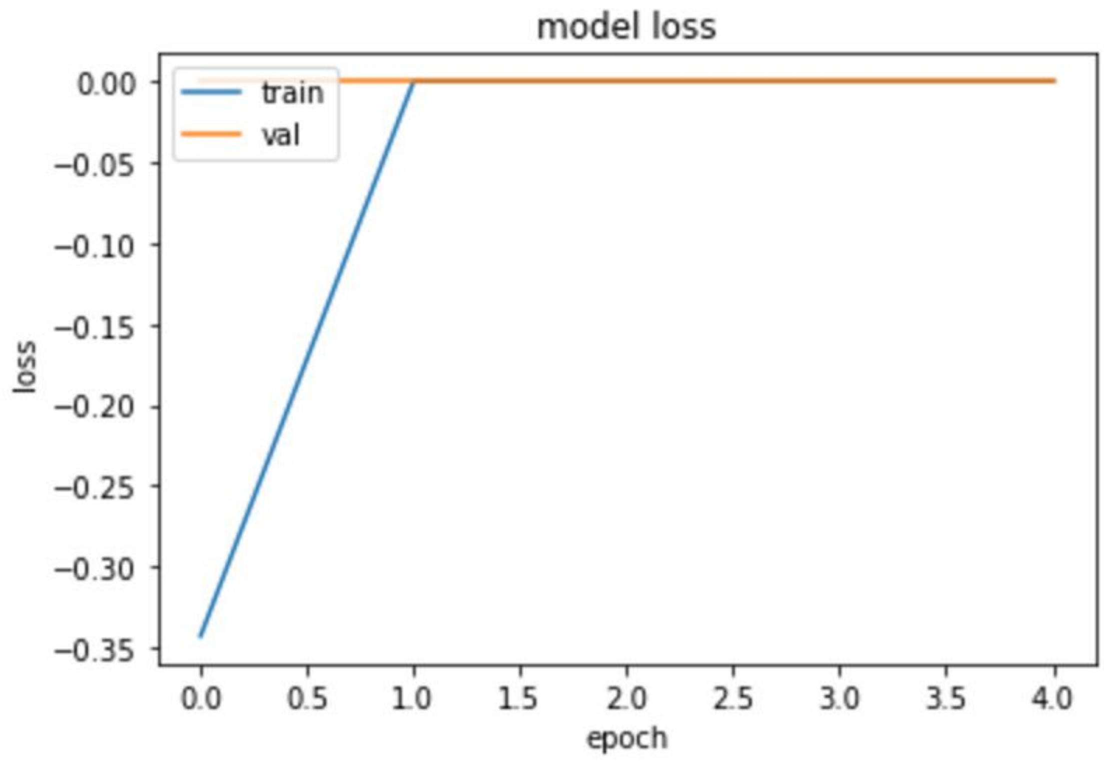
Additionally, I experimented with augmenting the architecture of the U-Net with Dropout layers. Dropout[17] is a technique for reducing overfitting by randomly dropping out units during the training phase. It has been applied successfully to U-Net architectures for segmentation applied to tasks like chest radiographs[18]. In combination with the the Dropout layers, I applied a maxnorm constraint, which acts as a further regularization by keeping the weight of a unit below a certain constant (in this case, I used C=6).

I was also concerned that the scaling down of the images might affect the accuracy, so I attempted to train the network on the full images. This proved prohibitively expensive in terms of GPU memory, so I settled for scaling them down 4x to 384x576, which at least preserved the aspect ratio of the original images.

After making these changes, I attempted to train my new model. However, this attempt was unsuccessful, as one can see from the graph below that after a single epoch the model was getting an average Dice coefficient of 0 on the training and validation sets:

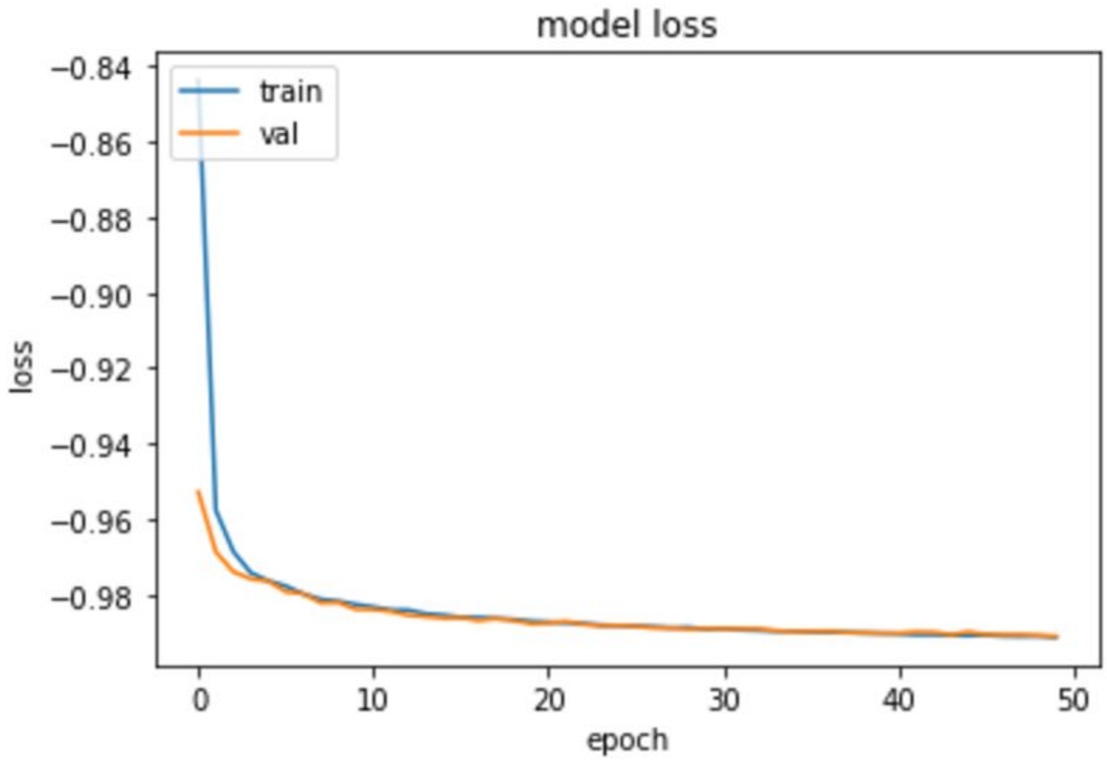
[17] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[18] https://arxiv.org/pdf/1701.08816.pdf



My theory was that, since I made so many changes to the model to prevent overfitting, I might have caused the training to go in the complete other direction, basically preventing the model from learning anything. So I decided to start with the simplest technique: just the data augmentation, and see if I could get an improvement from that.

So I trained using my original U-Net architecture, and the same 256x256 scaling, but with data augmentations applied to the dataset. The results were mixed. As you can see, the overfitting was greatly reduced, as the training and validation curves remain close together even after 50 epochs of training:



However, the addition of data augmentation appears to have made the model train more slowly, as after 50 epochs it still did not achieve a better score on the validation set than the original model without data augmentation. The average Dice coefficient for the model trained with data augmentation was 0.9911.

IV. Results

Model Evaluation and Validation

After experimenting with some refinements, the model that achieved the best validation score was still my original U-Net architecture without data augmentation. The final characteristics of the model line up well with the characteristics of the dataset—it has learned a set of parameters that allow it to identify different parts of the image, and segment out the pixels belonging to the cars. Training an architecture based on convolutional blocks in this case led to a robust solution to the problem—it can pick out detailed parts of different kinds of cars in order to make the correct segmentation.

To test how well this model generalizes to unseen data, I submitted its predictions on the test set (which has its ground truth withheld) to Kaggle for evaluation. My score on the private leaderboard (calculated with 75% of the test data) was 0.9896. This makese sense compared to my validation score of 0.9917, as I knew that the model had some issues with overfitting. In addition, there was some decrease to be expected because my model was working with scaled-down images, so the output masks were 256x256, and had to be scaled up to submit to Kaggle (which evaluated on full-scale ground truth maps).

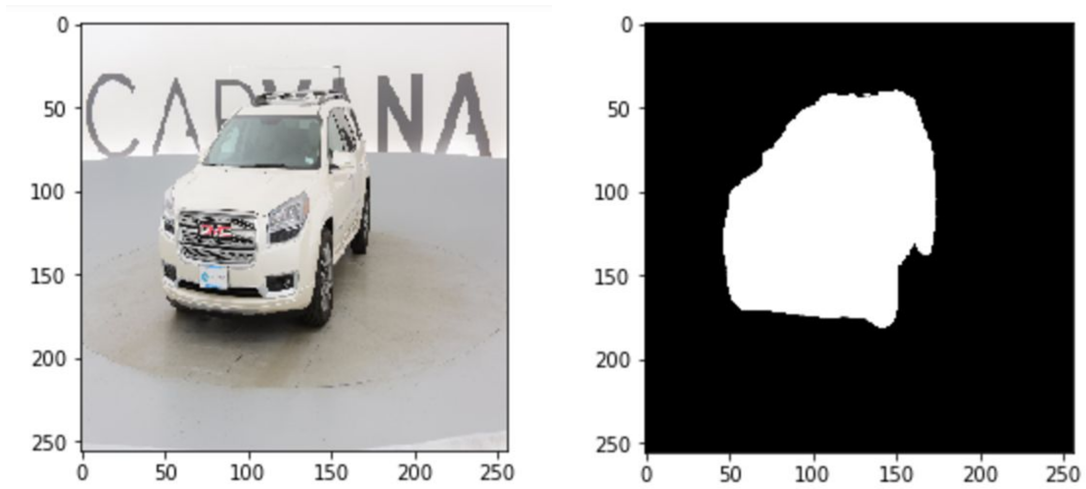
Justification

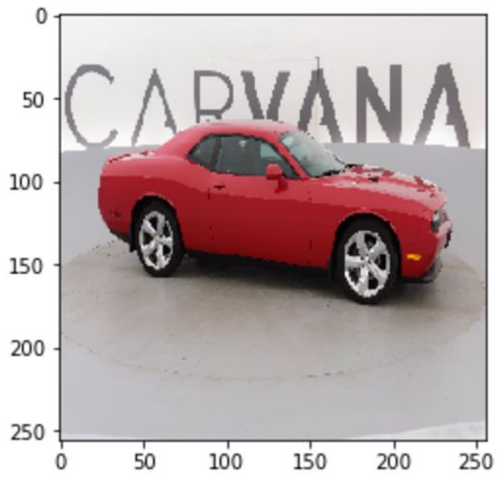
My model represents a significant improvement over the benchmark, achieved as validation score of 0.9917, compared to the benchmark's 0.7387. I believe that my solution is significant enough to have solved the problem. An average Dice coefficient of 0.9917 means that the predicted masks are quite close to the ground truth.

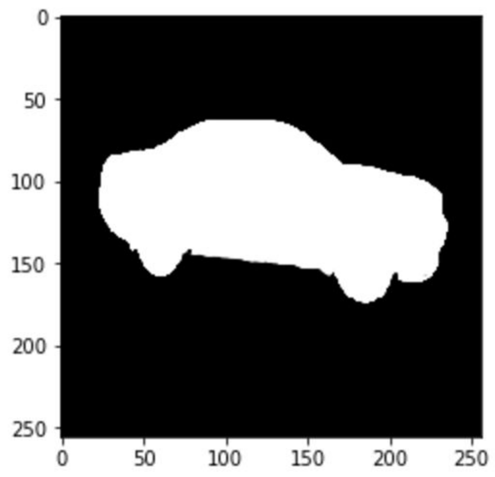
V. Conclusion

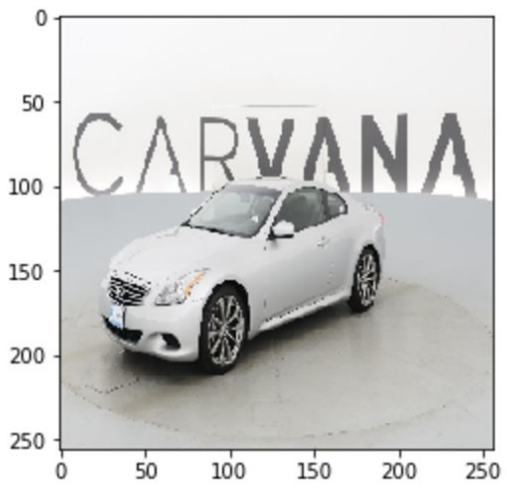
Free-Form Visualization

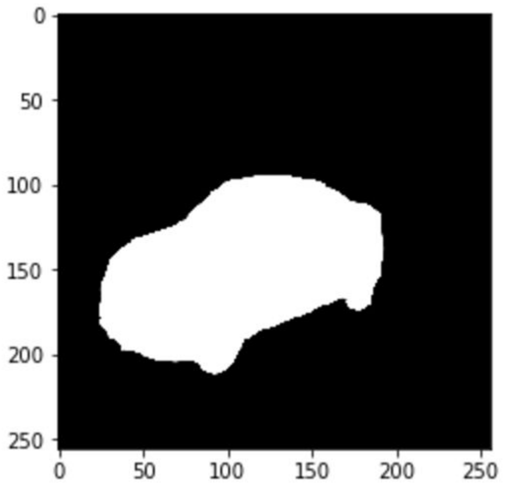
In order to visualize the performance of this model, here are some example images from the validation set, with the masks that the model predicts for them. As you can see, the model has learned not only to find the general shape of the car, but can pick out fairly granular details that may be present in only some makes or models, like the shape of the side mirrors or wheel wells.











Reflection

The process I used to come up with a solution to this problem was as follows: first, I obtained the dataset of car images, along with labels of the ground truth masks for each image. Then, I implemented a U-Net in Keras, and trained the model using the images and labels, while also tracking performance on a validation set I separated from the images used for training. An attempt to use data augmentation was successful at reducing overfitting, but did not achieve a better validation score compared to the original model.

Some of the difficult aspects of the project involved the performance of loading the images and training the model. As stated in the Implementation section, I experimented with training the images at different scales, but ultimately chose 256x256 in order to speed up training. Even with that speedup, it still took on the order of 10 hours to train the basic model, and even longer than that to train the more advanced architecture. The training time definitely limited my ability to experiment more with the architecture, or to do an effective hyperparameter search.

The final model and solution fit my expectations for the problem—looking at the visualization of some of the predicted masks, they look very close to what I would expect from a manually-applied mask. I think the solution is good enough to be used in a general setting to solve these types of problems.

Improvement

I think there are further improvements that could be made to the model. I think that successfully applying Dropout could improve the result of the model.

Another improvement that I think could be made is training several U-Net models with different random initializations, and then ensembling their results together.

Another neural network solution used for segmentation is The One Hundred Layers Tiramisu[19], which applies a kind of convolutional neural network called a DenseNet to the task of segmentation. The network is more complicated than the U-Net, but if implemented successfully, it may yield a better result on this task.

I do think an even better solution than mine exists. Given that this is a Kaggle competition, I can see that other teams on the leaderboard achieved better average Dice coefficients than I did.

[19] https://arxiv.org/abs/1611.09326