Background Separation using U-Net

Aakaash Jois [1] and Alp Aygar [1]

[1]Department of Electrical and Computer Engineering, Tandon School of Engineering
New York University

December 22, 2017

1 Introduction

The aim of this project is to perform pixel-wise classification on a car image dataset. Each pixel on the image should be classified as either 'background' or 'car' class, forming a mask on the car image. This type of image classification is called Image Segmentation. A neural network model with U-Net architecture [1] has been implemented to achieve this goal. The U-Net performance was tested on 4 different training setups with changing input sizes and added image augmentations. Our best model has achieved an accuracy of 98.96% in this image segmentation problem.

1.1 Dataset

The data used in this project has been taken from Carvana Image Masking Challenge on Kaggle [2]. It consists of 1918×1280 resolution images of 318 different cars taken in front of a similar background. Images consist of 3 color channels with 8 bits each, adding up to a total 24 bit colors. Each car has 16 different pictures taken from various angles resulting in a total of 5088 images. Each image is labeled with a corresponding masks. There are also more than 20,000 unlabeled images which were not used in this project.

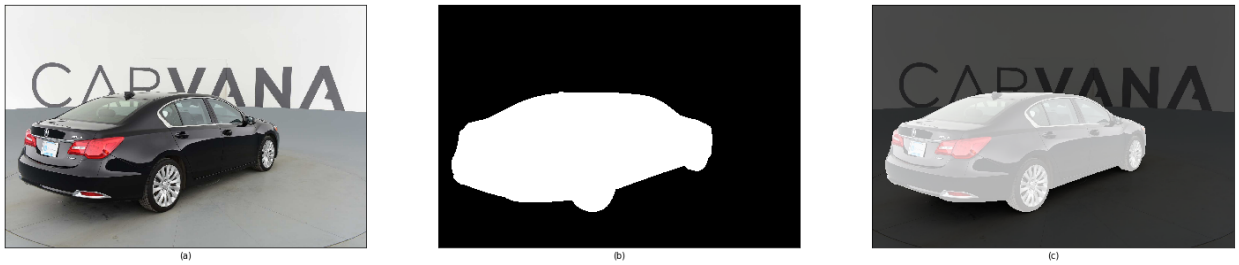

Figure 1: Sample image (a) and its corresponding mask (b). Shown together in (c).

1.2 U-Net Architecture

U-Net architecture is a state of the art convolutional neural network architecture for image segmentation tasks. It has been introduced in [1] for image segmentation on medical images. The

U-Net is a type of Deep Residual Network. In addition to the traditional convolutional and pooling layers, the U-Net architecture introduces the upsampling and concatenate layers.

The network initially works like a traditional neural network which does feature detection using convolutional and pooling layers. This path is called *the contractive path* as it concentrates contextual information on a smaller tile with each iteration. With every passing layer the receptive field of the neurons increase on the input image indicating a very low accuracy of location towards the bottom of the network. However the feature extraction performance is very high due to the repeated convolution and pooling operations with a high amount of feature map channels. After this point the network tries to locate the detected features by doing upsampling operations on the dense feature channels and concatenating the output with the dense feature channels from the previous layer. This path is called as the *expansive path*. Upsampling is a simple transpose convolution operation in which for each pixel of the input image, new pixels equal to the kernel size are created by multiplying the pixel value with the kernel. This operation reduces the contextual information, while increasing the localization. Concatenating layers in between allow the network to process the contextual and localization information together which allows the network to detect the features together with their locations.

Final layer of the network is a simple 1×1 kernel convolution followed by *sigmoid* activation. This allows the network to do classification on every pixel.

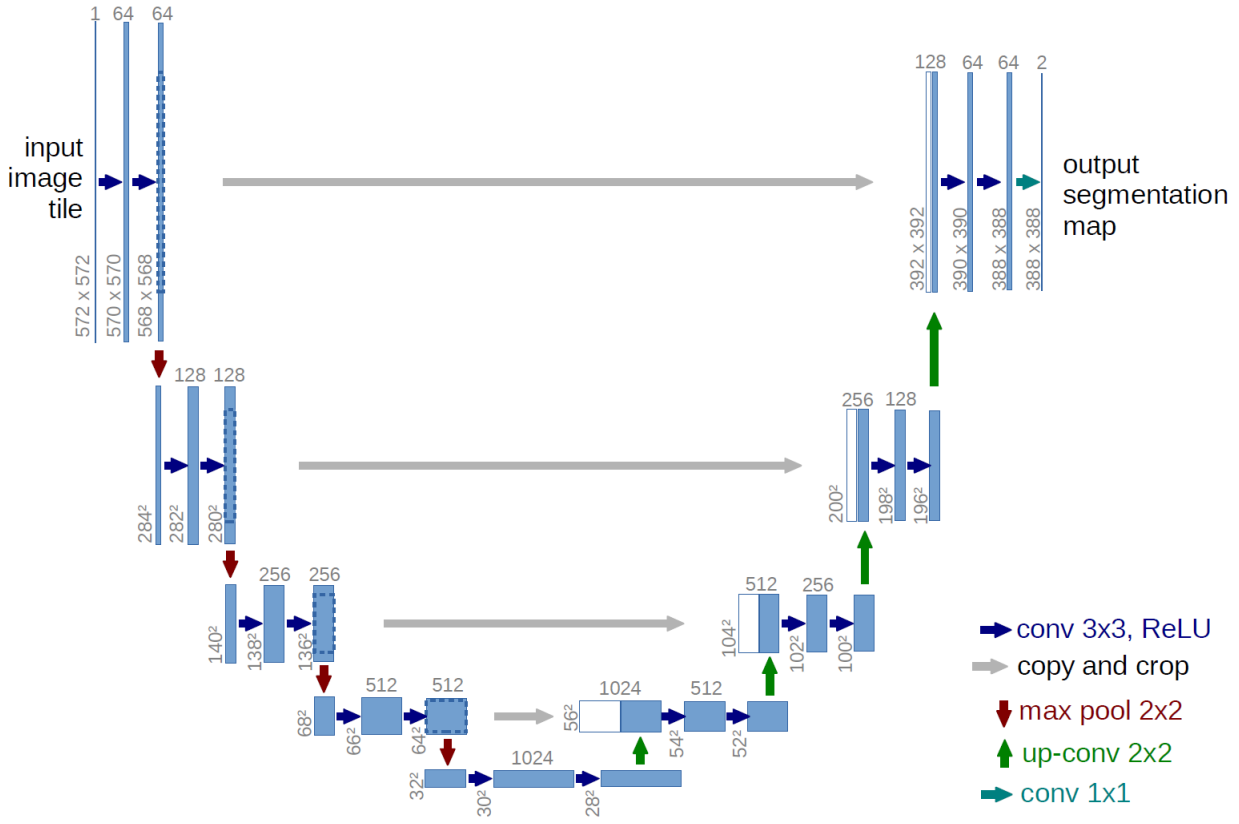

Figure 2: U-net architecture taken from [1]

2 Implementation

The complete implementation code can be found on GitHub:
`https://github.com/aakaashjois/Background-Separation-using-U-Net`

2.1 Data Augmentation Methods

Data augmentation methods are implemented to avoid overfitting and bias in the training. Random adjustments on the input data cause the network to be better tuned towards processing unexpected or slightly different data points. It also prevents the network from doing generalizations based on data frequently exposed.
In this project we implemented 5 different image augmentations. While training with augmentations each augmentation had a probability of 0.1 occurrence. It was possible for images to have multiple augmentations at the same time. The augmentation methods were written by using the sample code in [4], but large modifications were made to make them compatible with our code.
The only augmentation that was implemented with probability 1 was the k-means color quantization. On the trainings with augmentations present, all of the input images had their colors clustered into 8 sections.
Five different data augmentations methods were created and a function to control all of them together was implemented. Note that each of the augmentation functions except for quantization (random_shear, random_zoom etc.) were directly taken from [4] but we have implemented them together in one function in order to directly call them from the generator. Generator and color quantization will be explained in greater detail on the proceeding sections. Details can be found on the github code.

2.1.1 Flip

Simple flip operation that returns the symmetry of the image around vertical axis.

Figure 3: Sample image, its corresponding mask and their flipped states.

2.1.2 Rotate

Rotates the image using the transformation matrix T:

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

θ was uniformly sampled from $(-20, 20)$.

Figure 4: Sample image, its corresponding mask and their rotated states.

2.1.3 Zoom

Zooms towards the center of the image using the transformation matrix T:

$$T = \begin{bmatrix} z_x & 0 & 0 \\ 0 & z_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

z_x, z_y was uniformly sampled from $(0.8, 1)$.

Figure 5: Sample image, its corresponding mask and their zoomed states.

2.1.4 Shear

Shears the image using the transformation matrix T:

$$T = \begin{bmatrix} 1 & R\sin(\theta) & 0 \\ 0 & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

R determines the rotation direction for the shear and sampled as -1 or 1 with probability $1/2$. θ was uniformly sampled from $(-20, 20)$.

2.1.5 Shift

Shifts the image using the transformation matrix T:

$$T = \begin{bmatrix} 1 & 0 & r \times s_x \\ 0 & 1 & c \times t_y \\ 0 & 0 & 1 \end{bmatrix}$$

where r is the row number and c is the column number of the image. s_x and t_y were uniformly sampled from $(-0.2, 0.2)$.

Figure 6: Sample image, its corresponding mask and their sheared states.

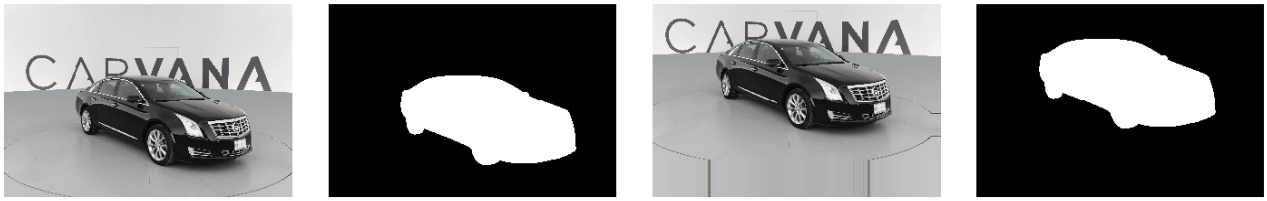

Figure 7: Sample image, its corresponding mask and their shifted states.

2.2 K-Means Color Quantization

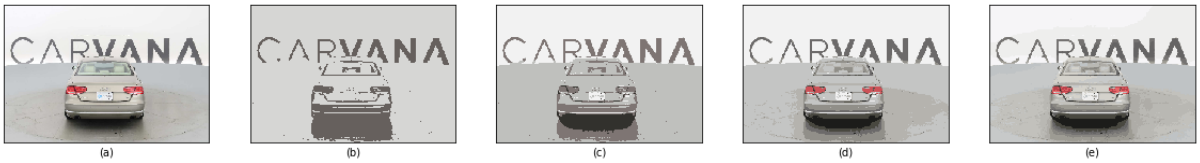

Figure 8: Original image (a) with 2, 4 , 8 and 16 color quantizations, demonstrated respectively in (b), (c), (d), (e).

K-means clustering for color quantization was implemented for the project in an attempt to increase performance as segmentation on a color clustered image is usually easier for the human eye, even though there is loss of data. It is also possible to reduce the actual input image size using k-means clustering. Figure 8 demonstrates the results of the K-means color quantization algorithm. Algorithm was based on [3], with modifications made for compatibility with the rest of the code. The code in [3] was one illustration on a single image. We have modified that code into a method to work with batches of images and masks with an option to select the number of required colors. Then this method was used together with other augmentation methods. We have implemented Note that clustering was done on each image by itself instead of by batches, so the clustered color means vary between images.

2.3 U-Net Implementation

The network was implemented on Python 3.6.2 using Keras with TensorFlow backend. Our goal was to implement the architecture in [1] and we were inspired by the implementation in [5], however we have taken a different approach to implement an architecture compatible with the data augmentation methods and different input sizes. Resulting network is the same in [5], but instead of

using functions to generate layers we have directly created the layers using built-in keras functions. Accuracy function was the *dice coefficient* given by the formula:

$$\frac{2 \times |X \cap Y|}{|X| + |Y|}$$

where X is the prediction mask and Y is the ground truth. *Binary cross entropy* was taken as the loss function. The computation graph of the model is demonstrated in Figure 9. Generator function have been used instead of loading all the images at once due to memory constraints.

2.4 Training

A Google Cloud Platform [6] virtual instance with 4 Intel Haswell CPUs, 15GB RAM and one NVIDIA Tesla K80 GPU was used for training. Adam optimizer with initial learning rate of 10^{-4} was used. Four different training setups were implemented. Their specifics are demonstrated in table 1. Each training was run for 10 epochs with a batch size of 2. Due to processing constraints,

	Input Image Dimensions	Augmentations	Run Time
Training 1	(320, 320)	✓	4.87 hours
Training 2	(320, 320)	✗	4.29 hours
Training 3	(640, 640)	✓	18.66 hours
Training 4	(640, 640)	✗	18.60 hours

Table 1: Training parameters

the batch size could not be increased to a higher value.

3 Results

	Training Accuracy	Validation Accuracy	Test Accuracy
Training 1	0.9748	0.9451	0.9459
Training 2	0.9776	0.9764	0.9769
Training 3	0.9866	0.9892	0.9896
Training 4	0.9838	0.9874	0.9877

Table 2: Comparison of accuracies with different training parameters. Accuracy represents the dice coefficient. For detailed information about training setup, refer to Table 1.

Results from our different training setups are demonstrated on figures 9, 10, 11 and 12. We can see that the validation accuracy convergence occurs more rapidly with larger input image size. The difference between the convergence speeds of the setups is caused by the increased input size. Since U-Nets are doing pixel based classifications, increasing the input image size of the dataset results in an increase of the overall training data. This is unlike the classical convolutional networks where classification is done per image, so the dataset size remains irrelevant of the input image dimensions. In our case, a 640×640 image contains 4 times the pixel amount of a 320×320 image which accounts to a quadruple increase of the training data. Thus, faster convergence for each

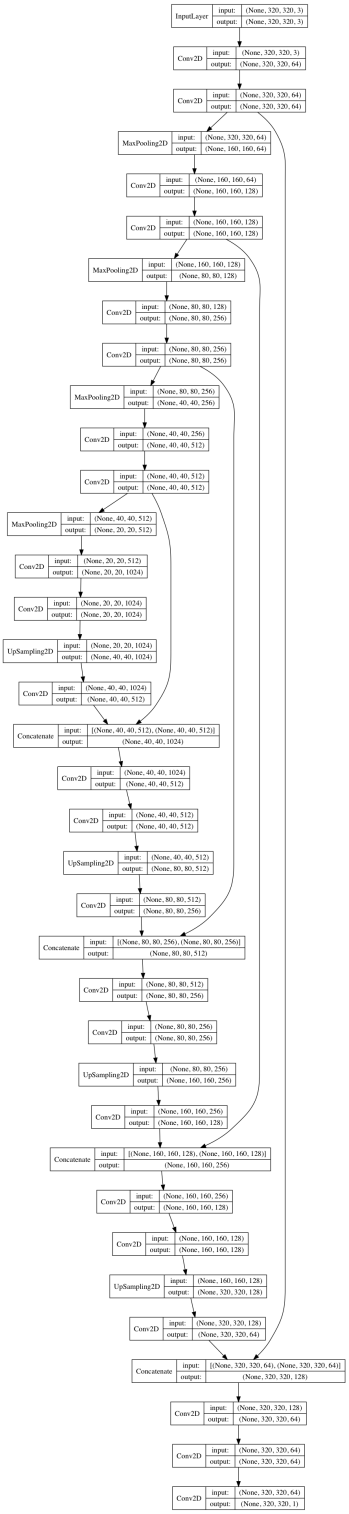

Figure 9: Architecture of the implemented U-Net with all the layers.

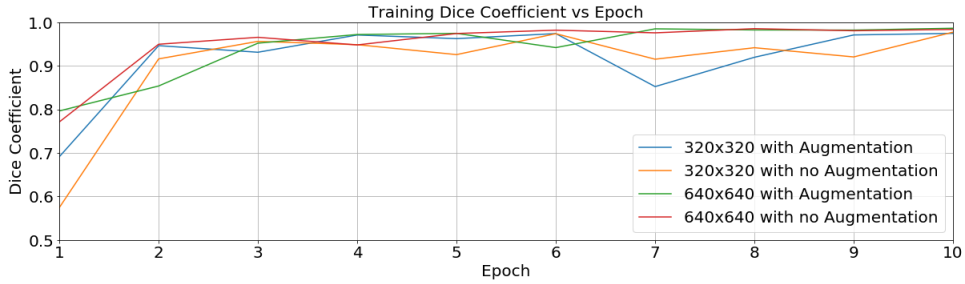

Figure 10: Training dice coefficient change with each epoch for various training setups.

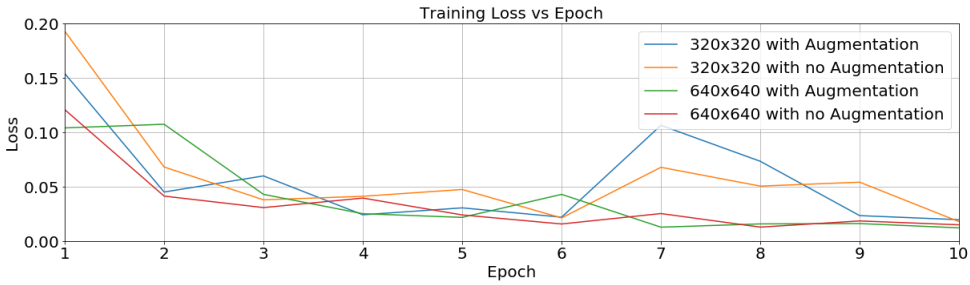

Figure 11: Training loss function change with each epoch for various training setups.

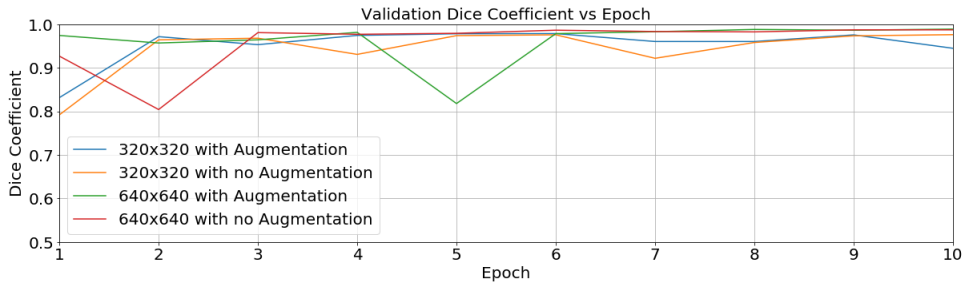

Figure 12: Validation dice coefficient change with each epoch for various training setups.

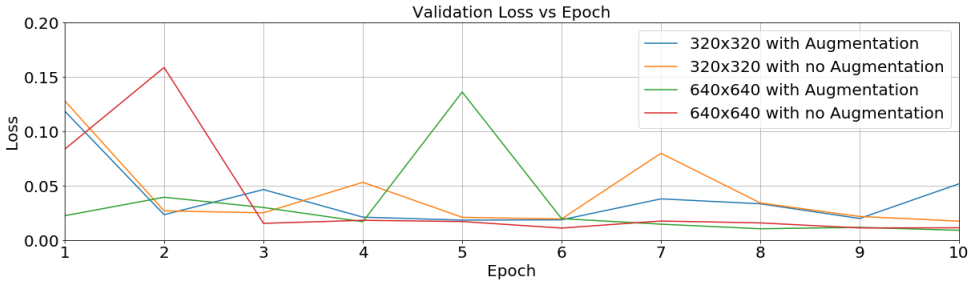

Figure 13: Dice coefficient change with each epoch.

epoch is not surprising. However on average 640×640 images were about 4 times slower to train. Another observation from these training sessions is the small impact of augmentations. We believe

this is due to the fact that our training and validation data are very similar. Our data consists of commercial photographs taken with great care so the images with small rotations or any kind of typical error are very rare. Thus training our network with augmentations that are not present in the dataset does not have a significant impact. Not only the augmentation impact is small, it also takes slightly longer to train the network when augmentations are present. However if we were to use regular street images to validate our network, we believe a network trained on augmentations would perform significantly better. Notice that the augmentations' impact was much higher for 320×320 training.

Overall our best performing network was training with 640×640 with augmentations which is as expected as it consists of the largest input data and with the added positive impact of the augmentations. Predictions based on this model are demonstrated on figure 14. Note that the output of the sigmoid function was ceiled and floored to 0 and 1 in order to get the actual predicted mask.

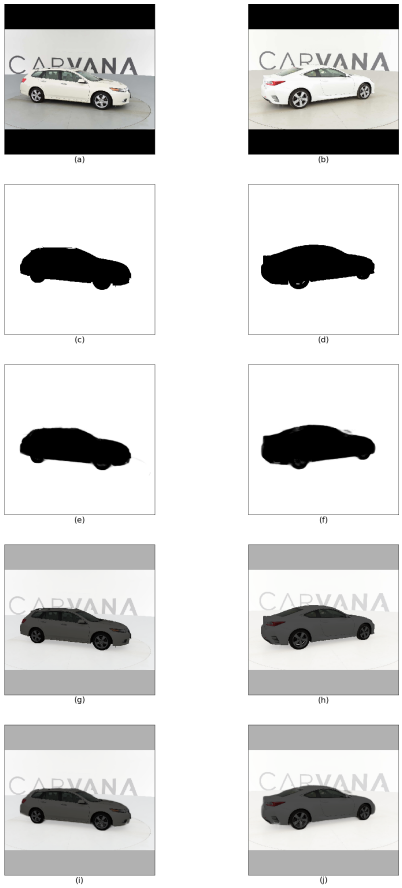

Figure 14: The 640 pixel network's performance compared to the ground truth. Input images (a), (b) and their ground truth masks (c), (d) compared to the predicted masks (e), (f). The ground truth masks together with the images are demonstrated on (g), (h) and the predictions with the input image are demonstrated on (i), (j).

References

[1] Olaf Ronneberger, Philipp Fischer, Thomas Brox *U-Net: Convolutional Networks for Biomedical Image Segmentation*. Medical Image Computing and Computer-Assisted Intervention (MICCAI), Springer, LNCS, Vol.9351: 234–241, 2015, available at arXiv:1505.04597 [cs.CV]

[2] Dataset from Carvana at Kaggle Competitions *Carvana Image Masking Challenge: Automatically identify the boundaries of the car in an image* available at https://www.kaggle.com/c/carvana-image-masking-challenge

[3] Scikit-Learn *Color Quantization using K-Means* available at https://scikit-learn.org/stable/auto_examples/cluster/plot_color_quantization.html

[4] Post by Beluga at Kaggle Competitions *Augmentation methods* available at https://www.kaggle.com/gaborfodor/augmentation-methods

[5] Post by Ecobill at Kaggle Competitions *U-Nets with Keras* available at https://www.kaggle.com/ecobill/u-nets-with-keras

[6] Google Cloud Platform available at https://cloud.google.com