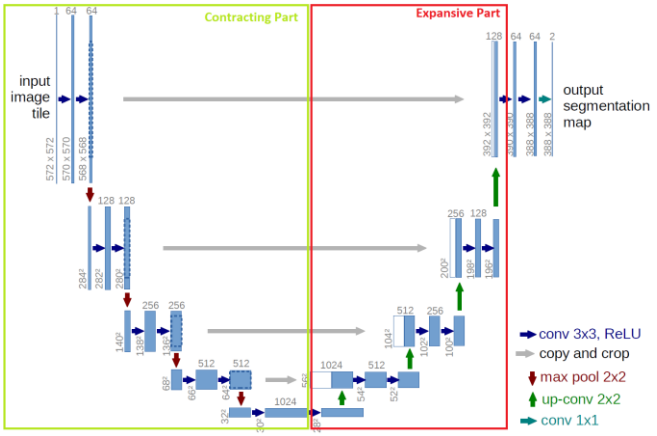
An in-deep analysis of the U-Net Segmentation

Sandu Cristian
student at the Computer Science Department
Technical University of Cluj-Napoca

U-Net is a C-NN architecture design for segmenting medical images. It was designed for biomedical image segmentation at the Computer Science Department of the University of Freiburg, Germany.

I. Original Architecture

The original architecture, without my notations, can be found here and the related paper, here. The following architecture will contain the author notations.



The system is composed of 2 parts:

- Contracting part, which acts like a usual CNN, extracting features from the image, and down sampling the image to prevent the overfitting,
- Expansive part, where the intermediary results from the previous part, are grouped together, and up sampling to allow the reconstruction of the image and process the extracted features.

As the reader can see, the components used in this architecture are simple, if the simple is the right term.

➡ a convolution with unpadded kernel of size 3x3, followed by a RELU activiation layer.

➡ because of the fureture downsampling, the current intermediar result must be adjusted for the concatenation, to a smaller size.

The others are self-explanatory. Up-conv and max pooling are using stride of 2 and no padding.

This is a general view of the architecture. For more details, please the articles provided previously.

The loss function used for this architecture is the Cross-Entropy. Because the loss is computed only a binary class, the function can be augmented to Binary Cross-Entropy.

The activation function is the Soft-max. The optimized was Stochastic Gradient Descent.

A. Implementation Changes

My implementation differs from the original one, as follows:

- the kernel convolution will have a padding of one and stride of 2. In this way, the output image size will not differ from the input one. Padding is added around the image, left or right.
- Up-convolution was replaced with a transpose convolution. In this way, we can train this level too.
- The activation function is not longer SoftMax, it was replaced by sigmoid with a threshold.
- The optimization function was changed to RMS, which takes into consideration the direction of the gradient, when is prepared to optimize it.

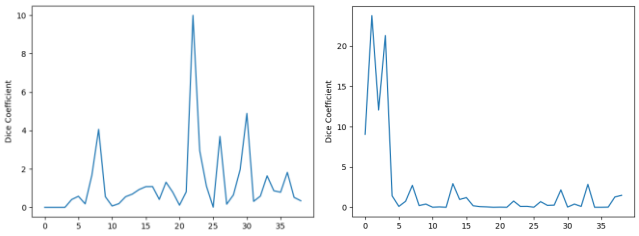
Also, it is worth to mention at this part that my implementation was done using the PyTorch framework.

B. Optimization problem (comparation)

There are many optimization algorithms which may be used to speed up the process & find the optimal solution.
For image segmentation, two algorithms are recommended: RMSProp and Adam.

The first thing to change is the optimization function. RMS take too big steps when going to the minimum gradient. It is going fast to minimum but missing it many times. This will cause many oscillations in system, creating spikes in the gradient and loss function. This condition is known as gradient exploding.
RMS may work efficiently, if we limit our learning rate & decrease the gradient before the backward propagation.

By contrast, the Adam optimizer, take into the consideration, velocity, acceleration and tend to go with smaller steps to the solution.
A graph comparation for Adam vs RMS is presented below, first image is RMS, followed by Adam:



Images are self-explanatory for the reason of why I've chosen to implement the Adam Optimizer.

II. Training and Loss Functions used

The system was tested on two categories of images. Binary and multi-class images (in progress). Please, take notice that all the results for this section were obtained using:

- batch of size 2, and 400 iterations.
- evaluation was run every next 10 trained images. The evaluation was executed on 30 images from the final of the batch.
- activation threshold of Sigmond was of 0.5 for Adam Optimizer
- all data is initially shuffled

For the training, there have been used 2 loss functions. One for the backward propagation, in the training part, and one for the testing parting, or validation part, to trace the real results of the algorithm.

A. *Dice function (validation part)*

The function used to trace the results of the algorithm is the Dice Loss Function. Mathematically, it can be written as:

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

where A and B are two images.

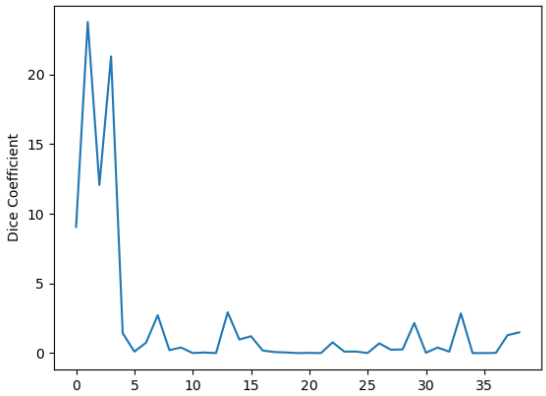
This algorithm was augmented to ignore following form. In this way, if the dice algorithm is applied on float point images, the small errors are ignored. To flatten the curve, a S variable was added:

$$Dice = 2|A \cap B + S| / (|A*A| + |B*B| + S)$$

The dice coefficient expresses how similar two images are, as a percentage. Their common pixels (from A and B) divided by the total number of the pixel in both images.

The dice coefficient is applied on the result max, after the SoftMax activation, and on the expected mask.

The results of the Dice Loss function for Segmentation into 2 classes:



As you can see, further the training goes, the general loss is decreasing.

B. *Binary images*

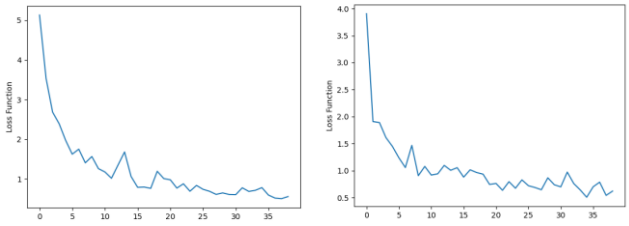
For the binary images, the first function is named Binary Cross-Entropy, and it is a faster form of Cross-Entropy for binary classes. The article will not enter to technical details about this function. The code is open-source, and well explained on many websites.

Anyways, since I need to document the most part of the things I have used, I will explain it in a few rows. The Cross-Entropy will calculate the distance between two vectors, according to the formula

$$h(x) = SUM(-log(P(x)) * Q(x))$$

where the P is the value from the target vector, and Q is the expected value, as a binary representation.

The results of lost function were good, for both optimizers. The following is the loss of the Adam optimizer, followed by RMSProp optimizer:



C. *Feature implementation for multi-classes:*

Currently, there is not official implementation for U-Net when dealing with more than 2 classes for segmentation problem. The approach, which is in progress, is to add N final layers, for every class. Break the mask for this layer and apply Cross-Entropy to calculate the distance vector for these classes.

III. Image Transformation (Binary Mask)

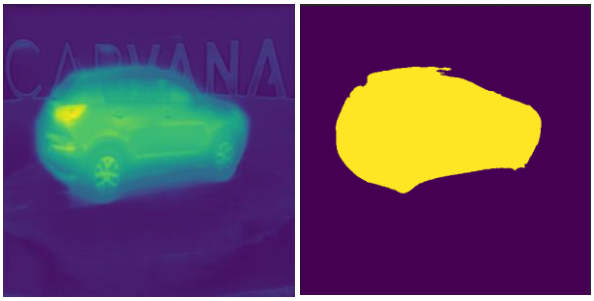
In this part we will analyze the image transform, thought the CNN, over the training procedure, and the output after applying the normalization (Sigmond + Threshold) on it.

The image dataset was obtained from a Kaggle Contest. This contains the original image and the mask. We will show the results of the algorithm, on the following image, having the following mask:

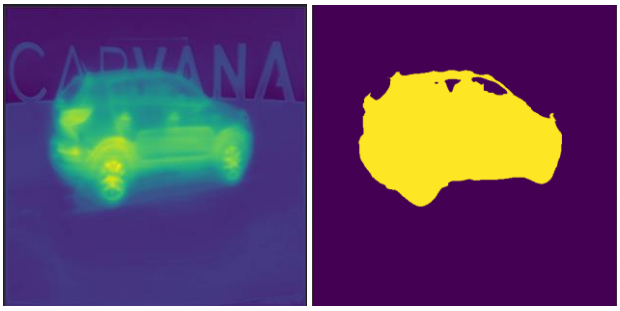
Training took place on a single epoch on 400 batches of 2 images. The following intermediary results are between 100 batches (in an incremental order), and the final one.

The results were as displayed as followings:

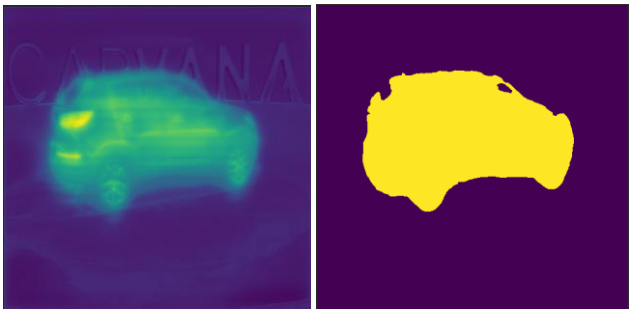
- first image, the feature map, obtained from the CNN.
- second image, the feature map after the sigmoid activation function was applied



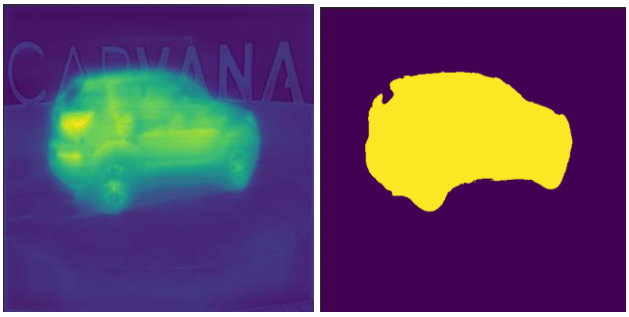
Results after traning on 100 batches of 2 images. As you can see, the image car image is difusing with the background. A part of the background is also active. (CARANVA word) The wheels are no defined, and other details are omitted.



Results after traning on 200 batches of 2 images. The background becomes more and more inactive. The car starts to be well evidentiated. The wheels are showing.



The neurons learn the car shape well. As you can see, on feature map, the background is starts to be no active and the car shape glow. Still, during the activation, the features are not strong enough to contour the whole car.



After processing 400 batches of 2 images from the input data, we can see that the features of the car are strong enough to contour the object. Please, take into consideration that the data is 20 times bigger than what it was processed for this example.

IV. Data Augmentation and Results

A. General Information

Data augmentation should prevent the system from overfitting, by applying a series of transformations on the given image.

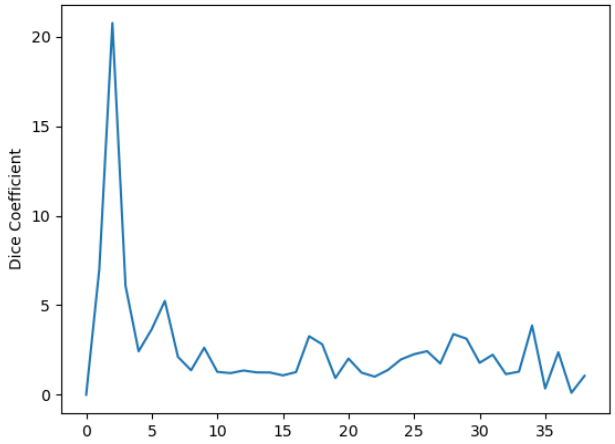
There were not many augmentations applied on the images. The only used operation was flipping the image. By having a 20% change of being flipped on vertical, or on horizontal, or on both axes.

B. Results after augmentation

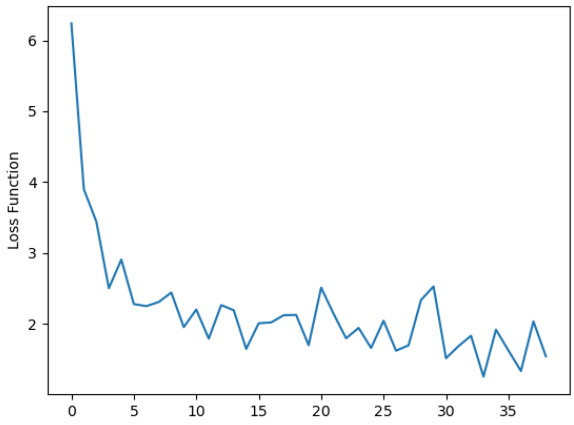
The tests for data augmentation were executed after the training with 400 batches of 2 images, like in non-augmented data, to be able to make a comparation. The following data is non-deterministic, because of the shuffle process. Considering the big number law, we can consider the similarity between results.

The other hyper-variables were taken the same as the previously data.

The global value of the **dice coefficient** was increased slightly, but start decreasing more on the future iterations, until a global minimum. It is logic that results can not be same as the non-augmented data, where all cars are facing a single direction, not rotated upside down.



Loss function was hardly affected by training on augmented images, as it was expected. Because the data differs very much, spikes are created.



As you can see, the image result is better, for the same number of train data, than the previous training.