Machine Learning Engineer Nanodegree

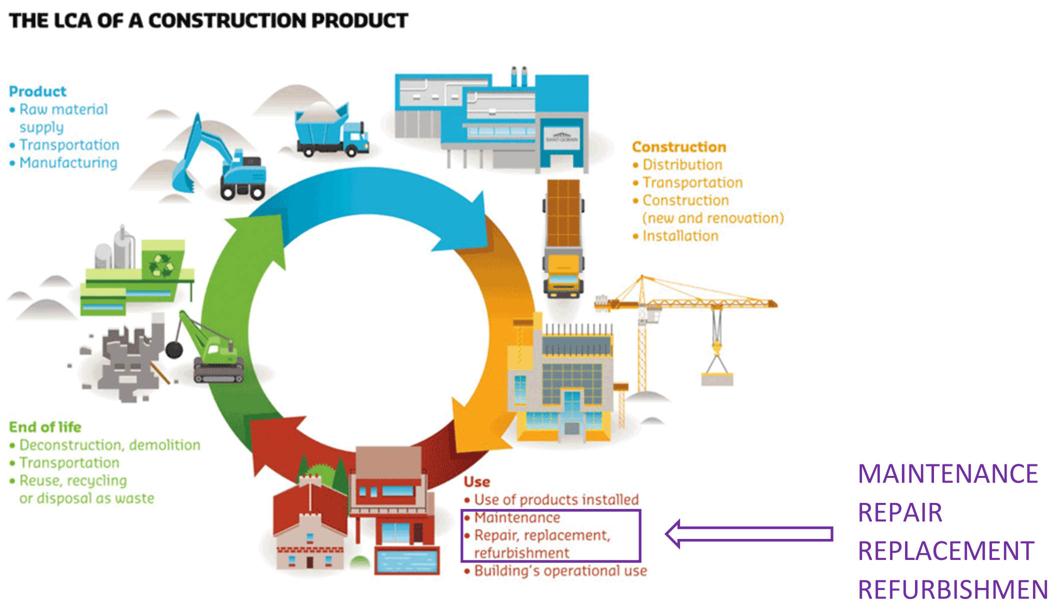
Capstone Project

Gaylord Marville
December 16th, 2018

I. Definition

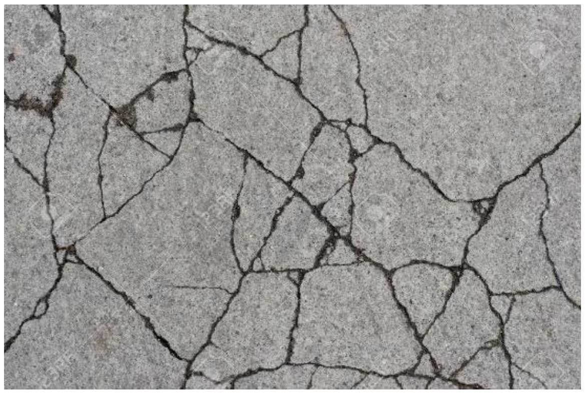
Project Overview

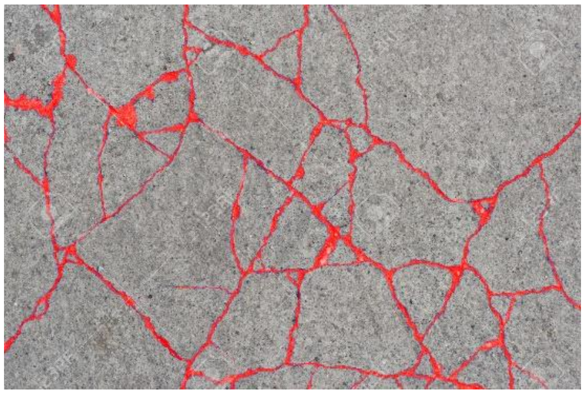
The life cycle of a construction product is roughly made of four parts. After the construction and before the end of life of the building, there is the most common and most extended part of this cycle – "the used part." During this part, the structure is held by a client who exploits it. Sometimes severe defects appear on this structure, and the client must repair it. For this purpose, the client engaged specialists to understand the cause and find a solution. Some clients are more cautious and inspect their property periodically before problems occur. Whatever the situation is, when it happens, the previously mentioned specialists are producing a written report stating the issues and how the client should handle them. Then the client engages a process of reparation, but that is something else.


THE LCA OF A CONSTRUCTION PRODUCT
Product
• Raw material supply
• Transportation
• Manufacturing
Construction
• Distribution
• Transportation
• Construction (new and renovation)
• Installation
Use
• Use of products installed
• Maintenance
• Repair, replacement, refurbishment
• Building's operational use
End of life
• Deconstruction, demolition
• Transportation
• Reuse, recycling or disposal as waste
MAINTENANCE
REPAIR
REPLACEMENT
REFURBISHMEN

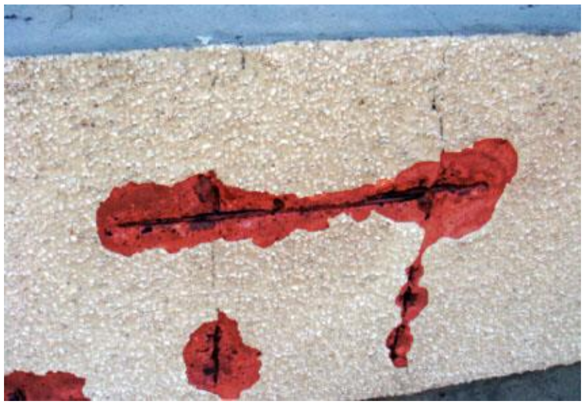
Life cycle of building and major civil engineering works such as tunnels or bridges

Recently the Morandi bridge collapsed in Genoa a city of Italy and did 41 casualties. To avoid those kinds of tragedy, a few years ago, French legislation fixed the terms of mandatory periodic diagnosis on public structures every 5 to 6 years. Those diagnosis consist of exhaustive inspections of the structural integrity, by detecting and locating and then rating all possible disorders a construction may be affected by. The written report I mentioned later is a 5 part one. One part of it prints out the disorder's pictures in A4 pages. Sometimes disorders like crack are barely visible due to the degradation of the printing process on the small sized images. When I was a student in training session, I had a mission among other which was to underline those disorders on the pictures and the only goal was to make them visible on the final paper report. It was a tedious task, especially for cracks.


Concrete cracks

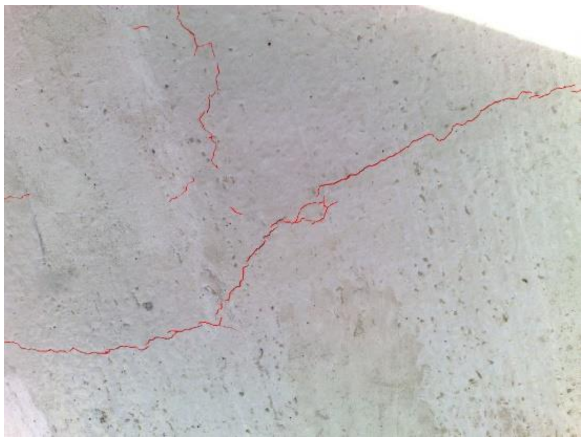

Underlined cracks

Spalling


Underlined spalling

The two cases above are easily readable in a report which is not necessecarily the case for the one below, even after it has been underlined.

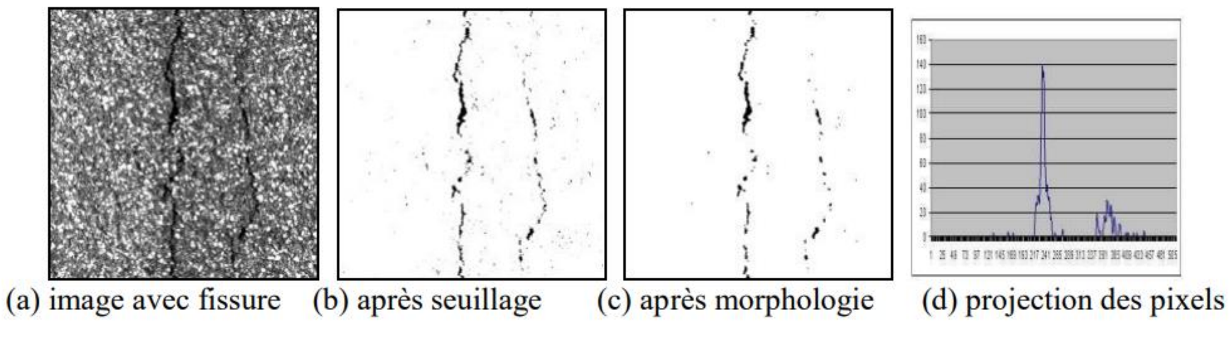
Concrete cracks


Underlined cracks

This goal may seem futile, but this is what motivated me first to find a solution to accomplish this underlining by computer. The use cases of such technics coupled with accurate localization of those disorders could radically change the way we are collecting them and presenting them to the client.

Problem Statement

The problem to be solved here is a computer vision one. It has been subjected to many research papers these past few years before the emergence of deep neural network and technics mainly based on convolutional neural networks to work on images. Before that, the problem was addressed using traditional image processing technics such as Canny edge detection or Hough transforms coupled with a non-destructive rigid transformation like rotation and flipping without success. It was mainly resulting in non-consistent results. It would have been hard to predict those kinds of tasks would need a new type of computer science.

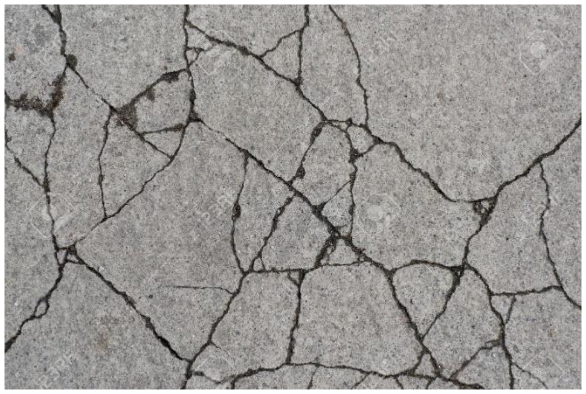


Traditional technics attempts

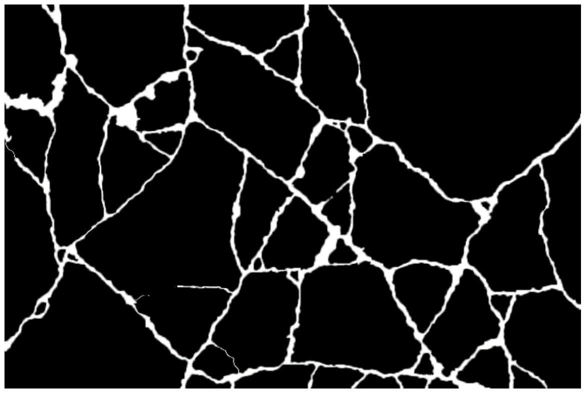
The idea will be to take a crack picture and use the Photoshop spot healing brush tool to remove the crack. We will then train a neural network in this task and later make the difference between the input image and the output one to obtain the crack.

Metrics

To evaluate our model, the best is to manually mask an arbitrary sample of images and, as a metric, calculate the dice or mean IoU between the model inference and the mask.



Input image



Masked image

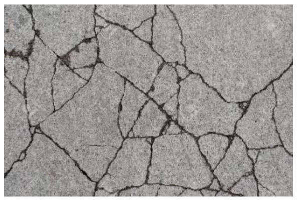
The IoU of a proposed set of object pixels and a set of true object pixels is calculated as:

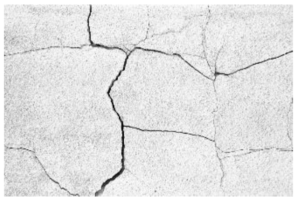
$$IoU(A, B) = \frac{A \cap B}{A \cup B}.$$

II. Analysis

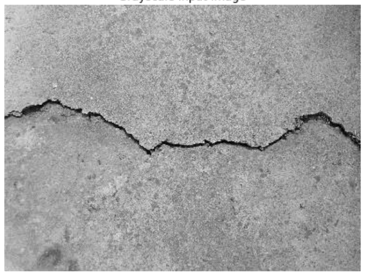
Data Exploration

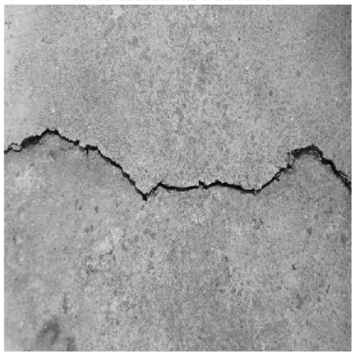
There are dozens of different disorders in civil engineering, but here I chose to work with the most representative one, the concrete crack. I had to build a dataset quickly, so I scrapped Google image using googleimagesdownload -k "concrete crack" (https://github.com/hardikvasa/google-images-download), in all the language Google could translate and only looking for large pictures.

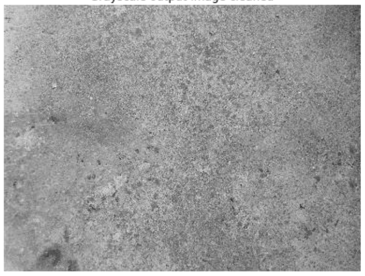




Exploratory Visualization


Grayscale input image


Grayscale input image resized 600x600.
Will be fed as batches in the network


Grayscale output image cleaned

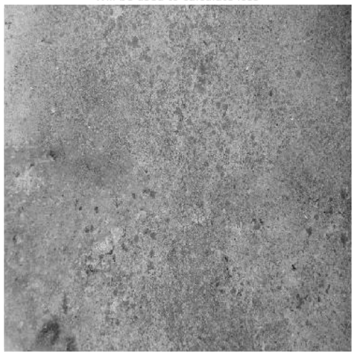
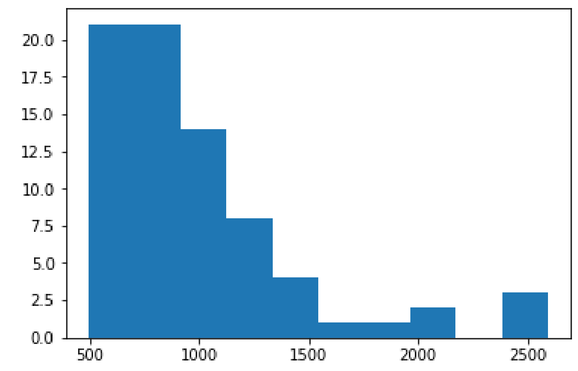
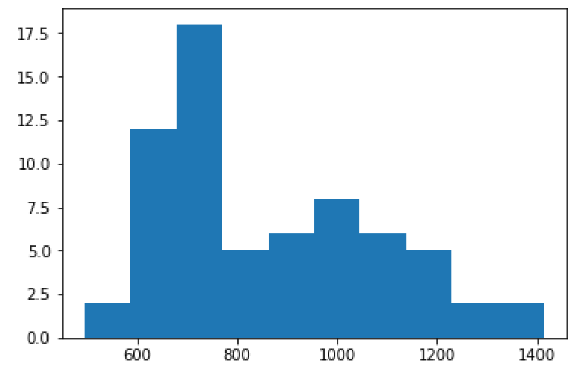
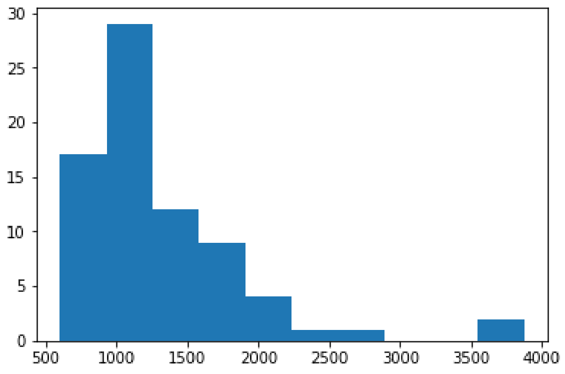
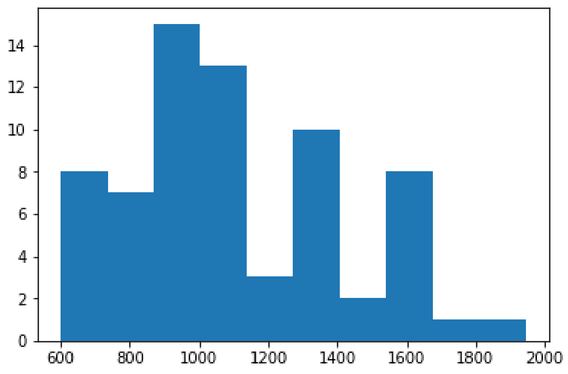
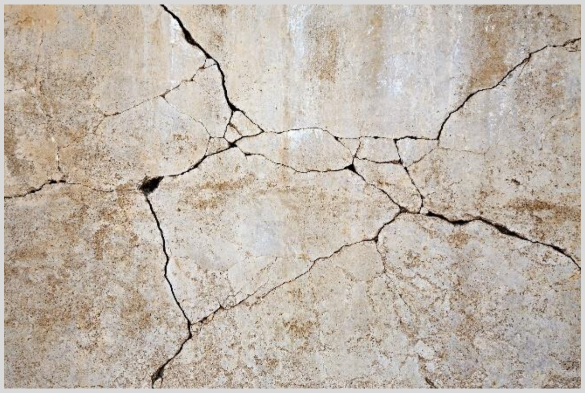
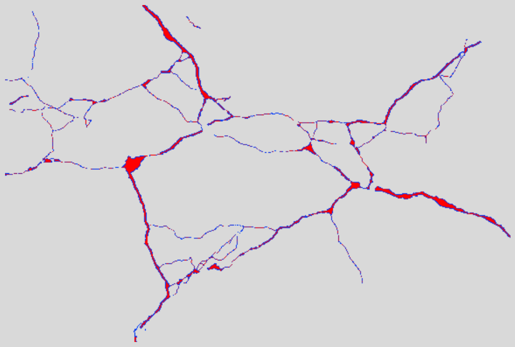
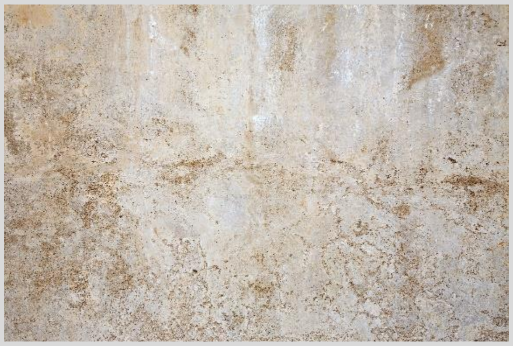
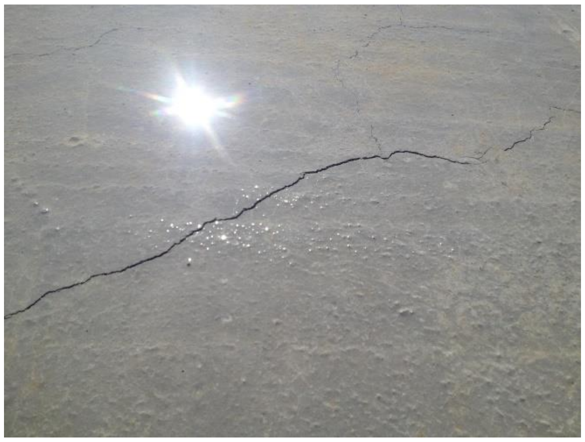
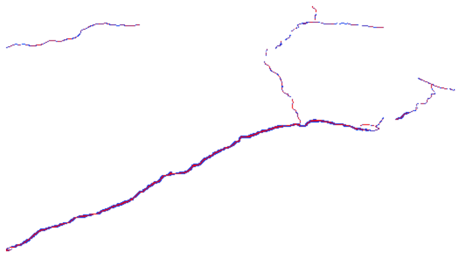
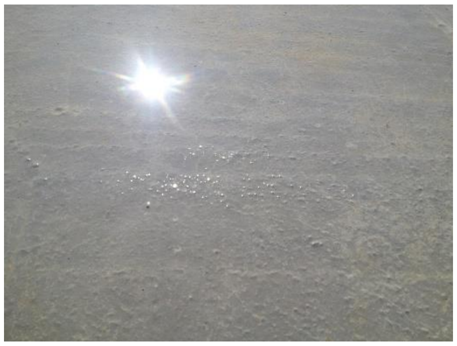
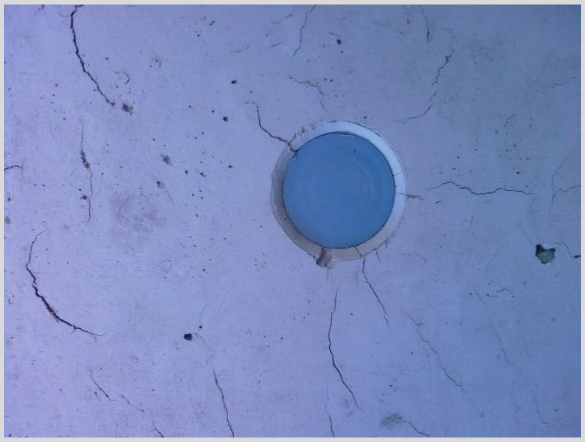
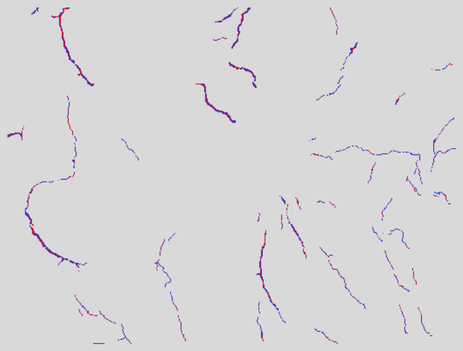
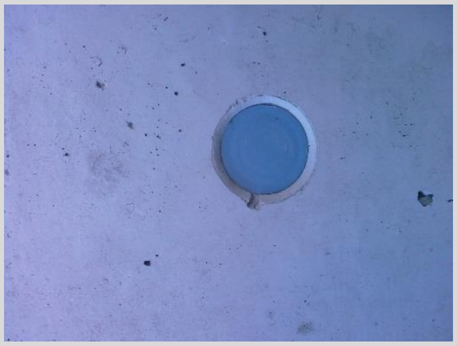
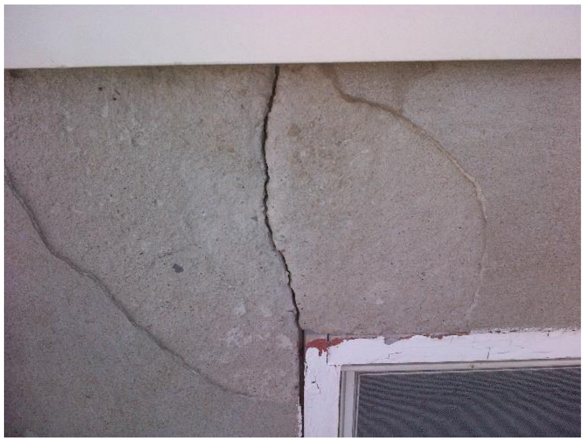
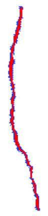
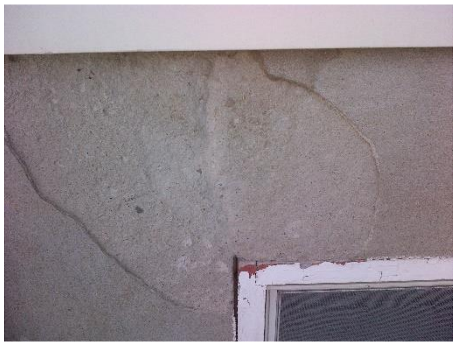

Grayscale output image cleaned
and resized 600x600.
Will be used to calculate loss

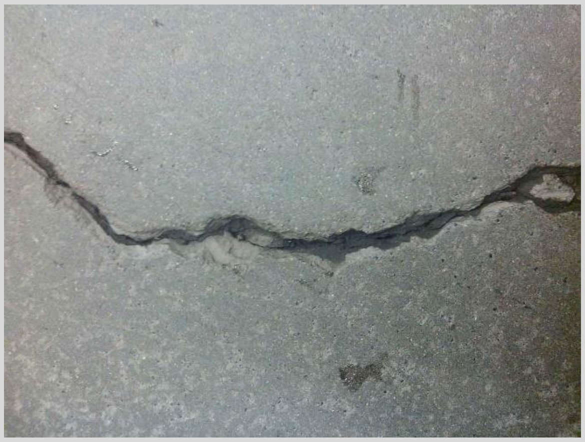
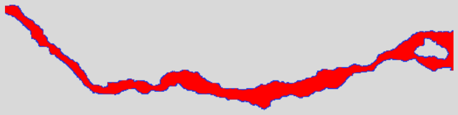
	Image size distribution	Image size distribution narrowed
Row		
Column		

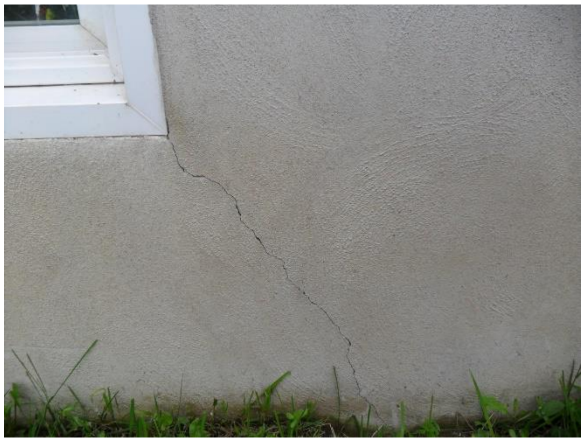
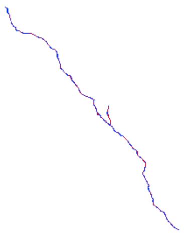
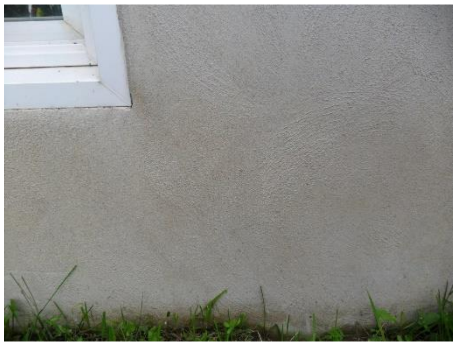
Looking at histograms above, we can observe that the widths of more than half of images are comprised between 600 and 800 pixels and 800 and 1200 pixels for their height. Therefore, if we want to resize those pictures without degrading it too much, we should pick values in those ranges. However, fully convolutional neural networks we want to use are massive and using such high-resolution images could lead to a memory error, even with the use of a generator. Thus, pictures will be arbitrarily resized to 600 x 600 pixels.

Above you can see the difference between a masked and a cleaned image and how we will use the clean solution to accelerate the definition of labeled datasets. You will also understand looking at the time it takes to mask a crack, why it is so essential to automate the process.

	Size: 2121 x 1414		
1	 *Input image*	 *Masked image*	**≥ 30 min**
		 *Cleaned image*	**9 – 10 min**
	Size: 1600 x 1200		
2	 *Input image*	 *Masked image*	**7 – 8 min**
		 *Cleaned image*	**≤ 1 min**

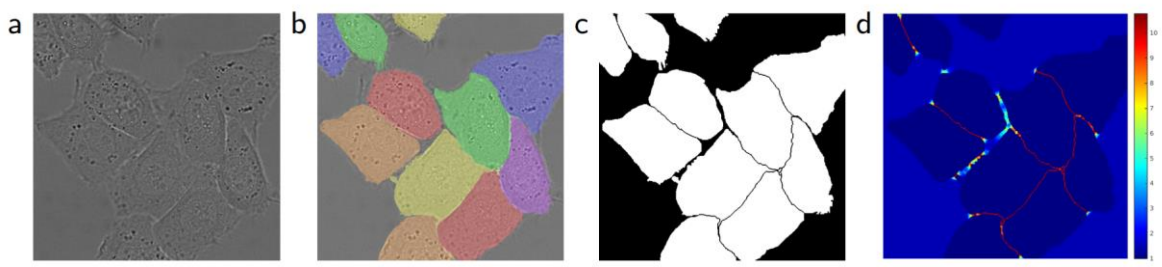
	Size: 2048 x 1536		
3	 *Input image*	 *Masked image*	**≥ 30 min**
		 *Cleaned image*	**7 – 8 min**
	Size: 2592 x 1944		
4	 *Input image*	 *Masked image*	**≤ 30 sec**
		 *Cleaned image*	**≤ 30 sec**

	Size: 1366 x 1025		
5	 *Input image*	 *Masked image*	**≤ 30 sec**
		Cleaned image	**≤ 30 sec**

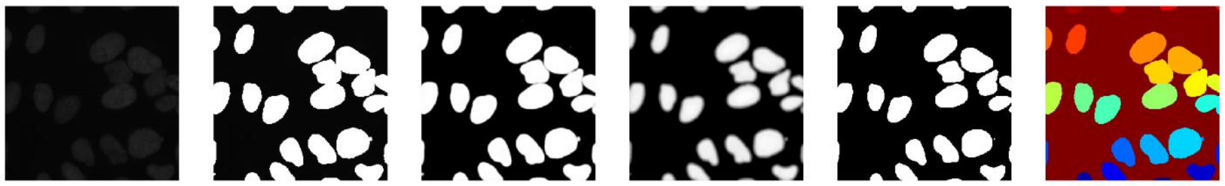
	Size: 1440 x 1080		
6	 *Input image*	 *Masked image*	**2 – 3 min**
		 *Cleaned image*	**30 – 45 sec**

Algorithms and Techniques

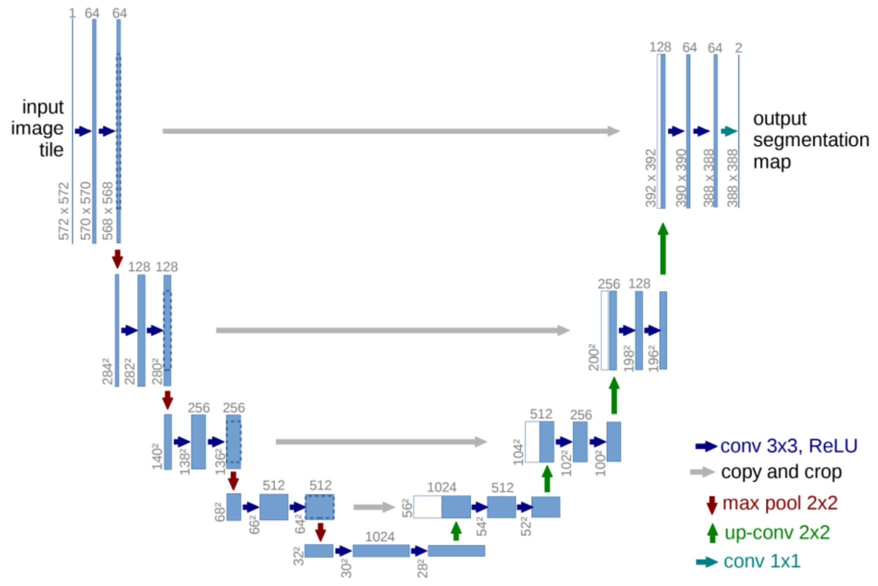
During the “2018 Data Science Bowl Kaggle competition”, U-Net an efficient architecture of image segmentation made the difference. We will reuse for cracks detection.



From the 2015 U-Net research paper https://arxiv.org/pdf/1505.04597.pdf

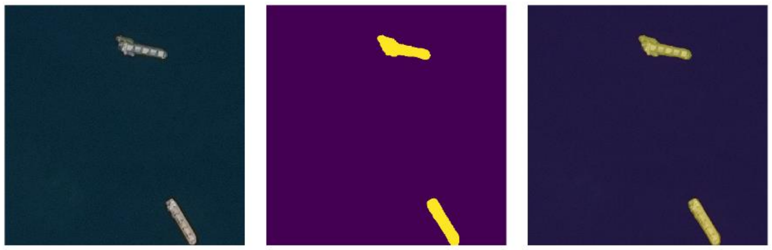


Kaggle data bowl challenge



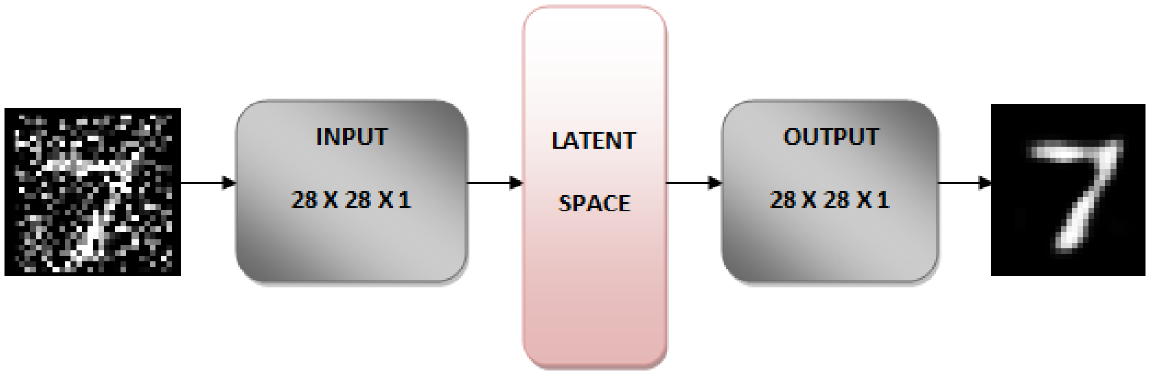
U-Net architecture

U-Net is composed of an encoder and a decoder linked in the middle by concatenated connections which improve the model contouring sharpness. U-Net can be used on different CNN architecture and we I'm planning to use it on the ResNet34 which is a lightweight network. I will not use a pre-trained model of ResNet34 which is increasing the implementation difficulty and may not be pertinent as no dataset today is close enough from cracks.


U-Net recently used on the Airbus/Kaggle challenge

Benchmark

Denoising Autoencoder



"The idea behind a denoising autoencoder is to learn a representation (latent space) that is robust to noise. We add noise to an image and then feed this noisy image as an input to our network. The encoder part of the autoencoder transforms the image into a different space that preserves the handwritten digits but removes the noise. As we will see later, the original image is 28 x 28 x 1 image, and the transformed image is 7 x 7 x 32. You can think of the 7 x 7 x 32 image as a 7 x 7 image with 32 color channels."

https://www.learnopencv.com/understanding-autoencoders-using-tensorflow-python/

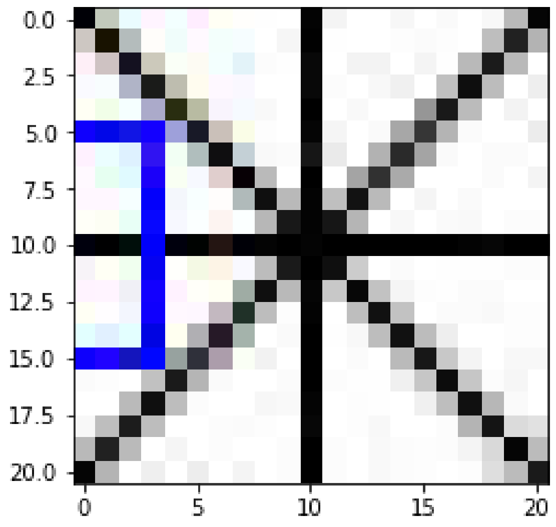
By lack of time I decided to not implement this simple denoiser solution in TensorFlow, but a tutorial can be found on the link above.

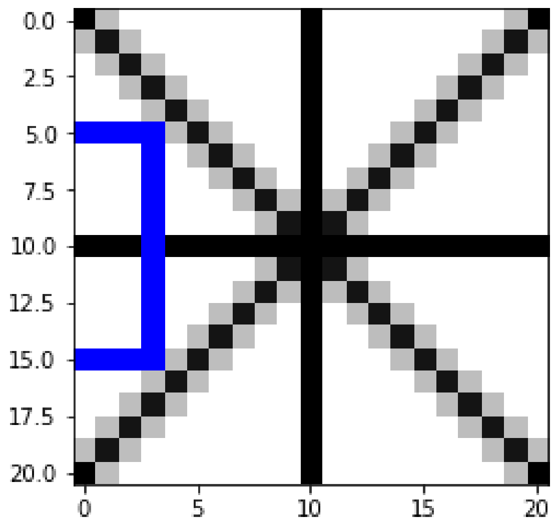
III. Methodology

Data Preprocessing

When working with images, it is more than wise to consider the PNG compression format instead of JPG since the last one is degrading images during compression. We want to delete noise due to compression during the process of creation of the labeled-masked images. Indeed, neural networks may be sensitive to those kinds of perturbations which could prevent it to converge to the solution or at best slow the progression.

	Image	First channel of the corresponding numpy array

JPG		`[[[222 241 255]]`
		`[[224 250 255]]`
		`[[246 255 255]]`
		`[[232 255 246]]`
		`[[   0  14  10]]`
		`[[240 255 239]]`
		`[[240 240 246]]`
		`[[243 246 254]]`
		`[[227 255 255]]]`

PNG		`[[[255 255 255]]`
		`[[255 255 255]]`
		`[[255 255 255]]`
		`[[255 255 255]]`
		`[[   0    0    0]]`
		`[[255 255 255]]`
		`[[255 255 255]]`
		`[[255 255 255]]`
		`[[255 255 255]]]`

Implementation

This project is an experiment; therefore, everything was kept as simple as possible to prevent unsolvable issues and help to debug.

Architecture

For the project we used a non-pretrained U-Net-34 architecture which removed the difficulty to manipulate a pretrained Resnet-34; extraction of the pretrained weight, choice of the trainable ones, and adapt it to a U-Net architecture. The architecture implementation was straightforward though. All the weights were retrained from scratch with no fine tuning or transfer learning.

Dataset

The choice of the dataset was simple due to its total size (75 images). I tried to choose it as heterogeneous as possible to help generalization but not too complex to prevent underfitting. Most of the crack where well defined, but some images where particularly tricky (see the model evaluation part above). Using Sklearn, I split it into two sub-datasets; 60 images for the training and 15 for the validation. I was conscient of the small size of the dataset but just wanted to evaluate it to start.

Generator

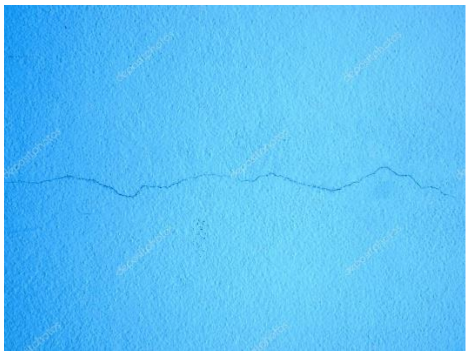
After having explored the size of the image, I decided to resize it all to 600 x 600 pixels. Working with a GTX 1080 Ti with 11Gb of memory, I decided to use a subclass of the generator provided by PyTorch. It brought to feed the model with batches of 3 pictures before reaching the out of memory error of the card.

Training

A usual with neural networks the goal is to flow data through the network (the forward pass), return an inference of the same size of the “labeled” image (the cleaned one in our case), compare them to calculate the loss and backpropagate it into the network. Reproduce the task through all the training dataset and for a predefined number of epochs. Take care to use well-suited learning rate to smoothly update the weights during the backpropagation part to help it learn.

Refinement

The model was first trained on RGB images, but it seemed to have trouble inference on colored concrete photos like this one:



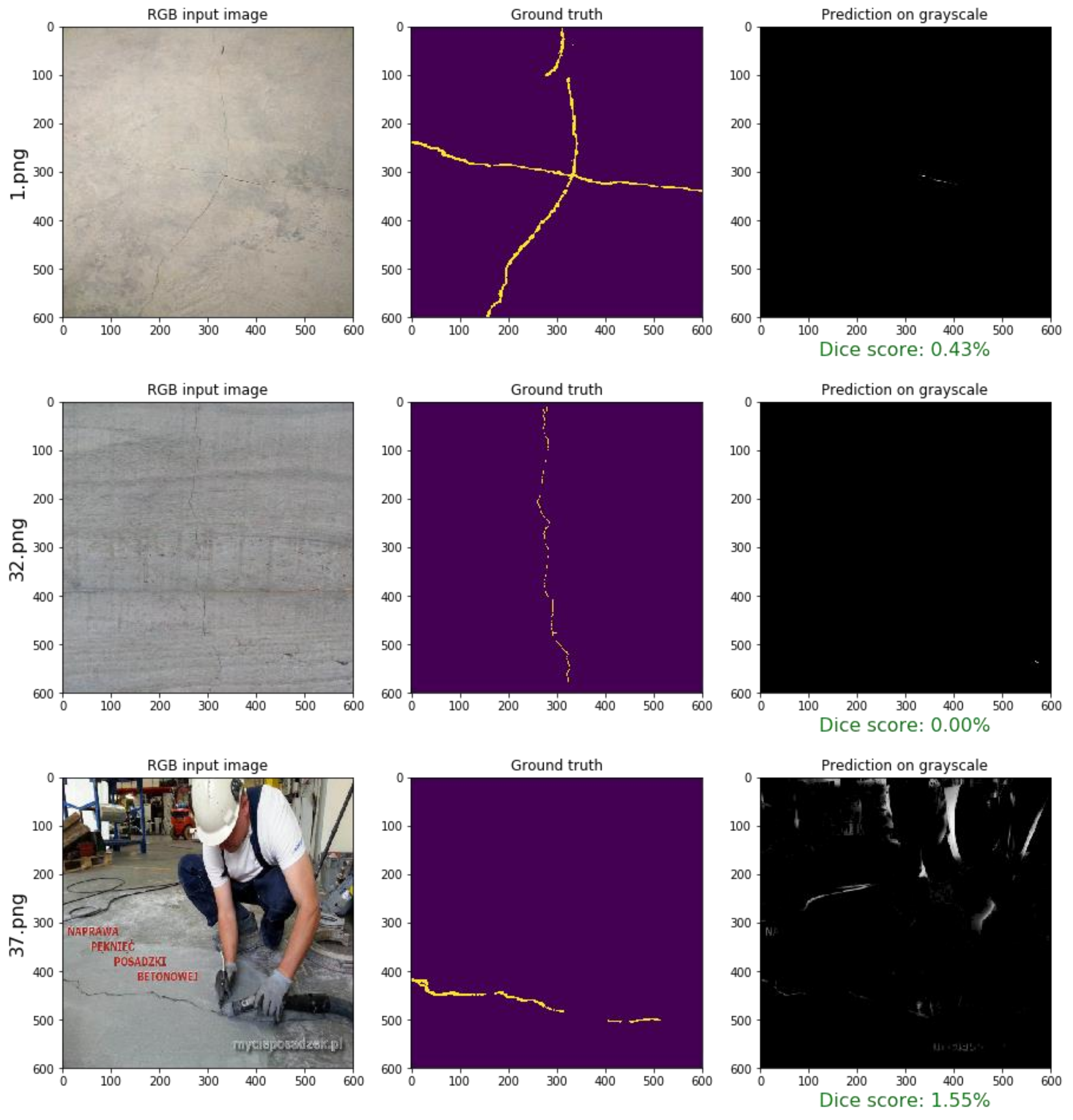
It motivated my decision to train it a second time on grayscale images to make it more resilient. The model trained a little faster and converged efficiently.

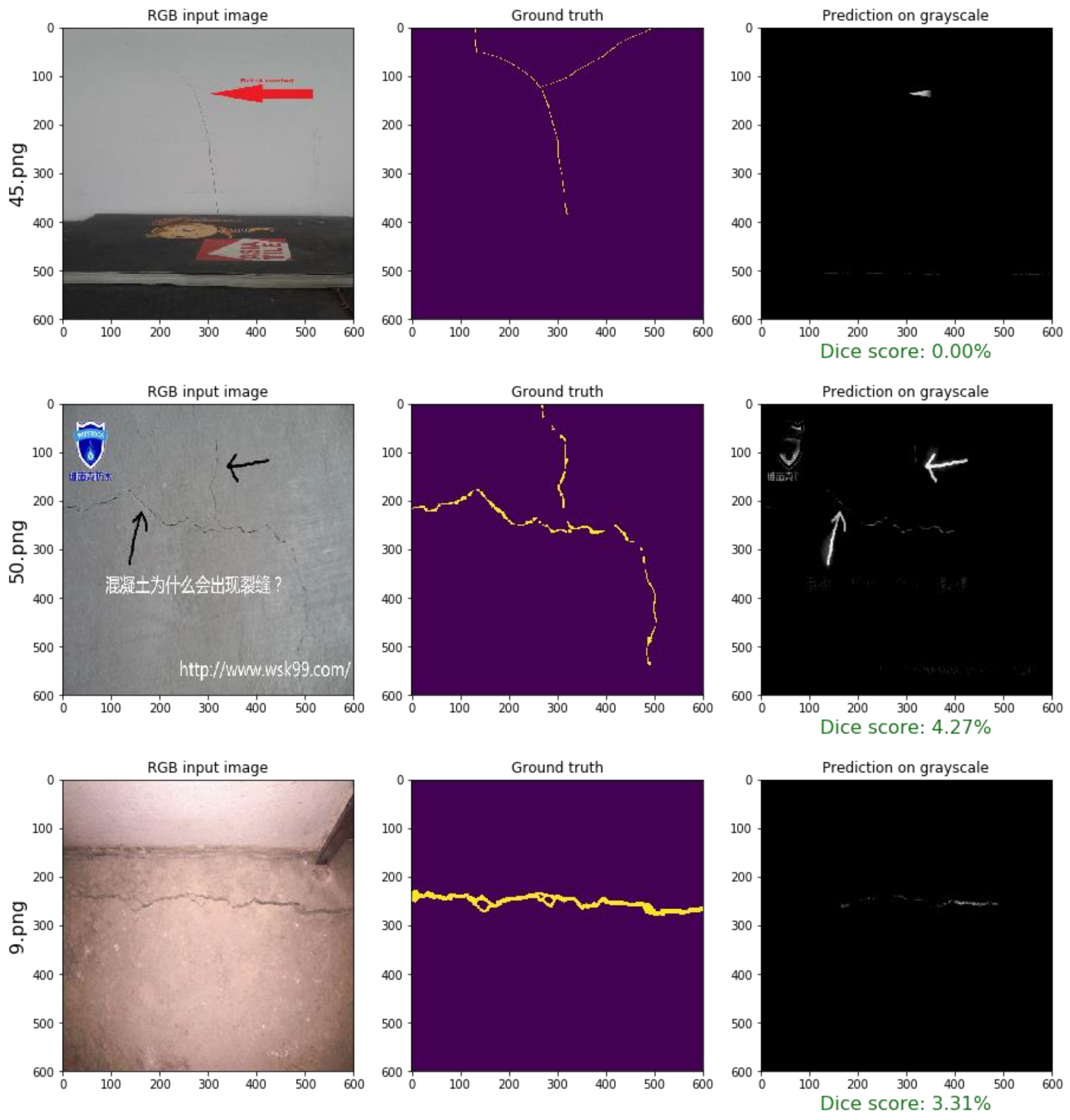
600 epochs on RGB	600 Epochs on grayscale
Epoch 1 Average loss: 0.04655402526259422	Epoch 1 Average loss: 0.03623107820749283
Epoch 50 Average loss: 0.0064216977916657925	Epoch 50 Average loss: 0.004056459292769432
Epoch 100 Average loss: 0.004951159469783306	Epoch 100 Average loss: 0.0033878725953400135
Epoch 150 Average loss: 0.004357353784143925	Epoch 150 Average loss: 0.003090395126491785
Epoch 200 Average loss: 0.003998121712356806	Epoch 200 Average loss: 0.0029050935991108418
Epoch 250 Average loss: 0.00377761316485703	Epoch 250 Average loss: 0.002823435701429844
Epoch 300 Average loss: 0.0035975740756839514	Epoch 300 Average loss: 0.0027590577956289053
Epoch 350 Average loss: 0.003458737162873149	Epoch 350 Average loss: 0.002718551317229867
Epoch 400 Average loss: 0.0033507065381854773	Epoch 400 Average loss: 0.002678843215107918
Epoch 450 Average loss: 0.0032665219623595476	Epoch 450 Average loss: 0.0026468904688954353
Epoch 500 Average loss: 0.00319574773311615	Epoch 500 Average loss: 0.0026138287503272295
Epoch 550 Average loss: 0.0031365426257252693	Epoch 550 Average loss: 0.002587314695119858
Epoch 600 Average loss: 0.003080792259424925	Epoch 600 Average loss: 0.0025665860157459974
Elapsed time: 05h-08m-55s seconds	Elapsed time: 04h-52m-37s seconds

IV. Results

Model Evaluation and Validation

Worst inference (Dice score < 5%)

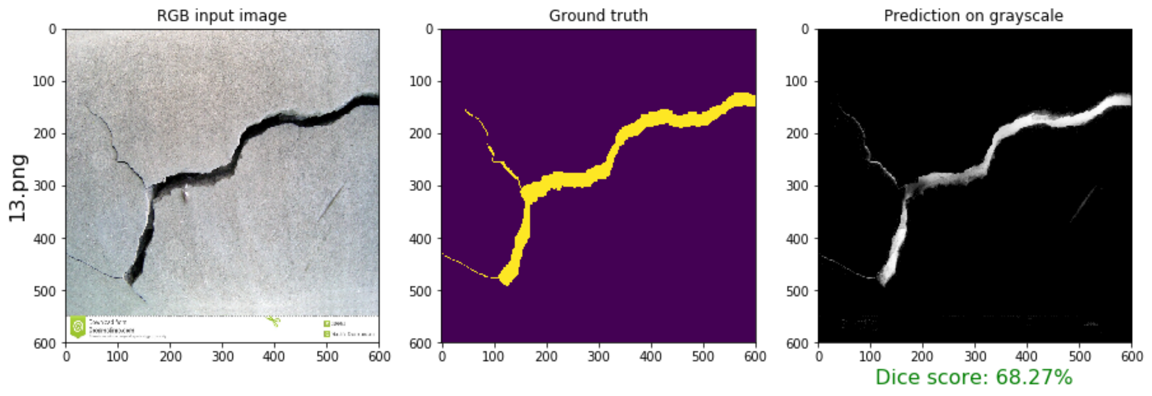


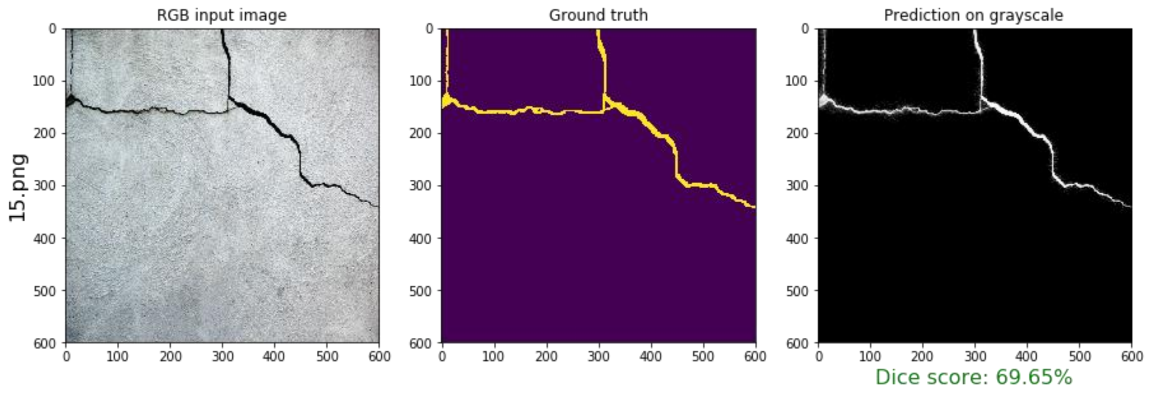


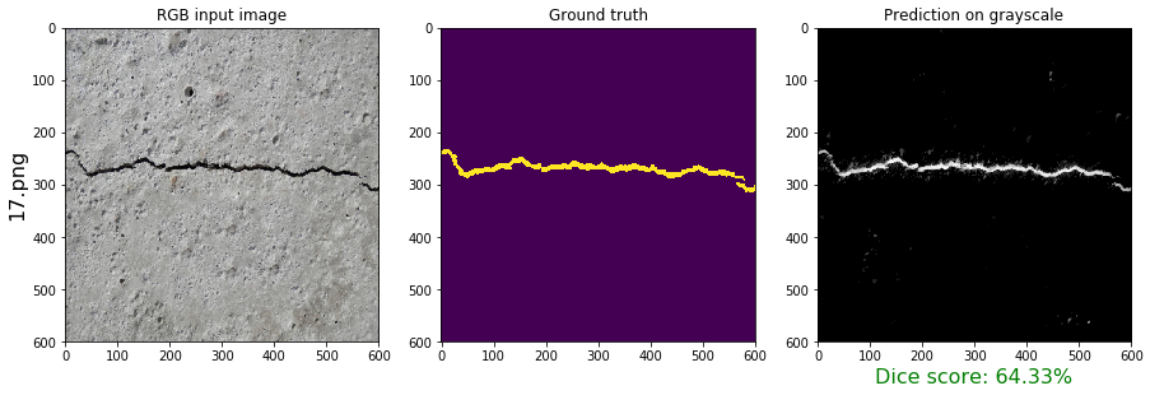
<u>Interpretation:</u> Images with the worst inference seems to suffer from the same problem. I denoted three of them; a small crack will be hardly detected, an environment with a lot of unrelevant and confusing pieces of information (arrow, text, people) or an unusual color of concrete may be challenging too.

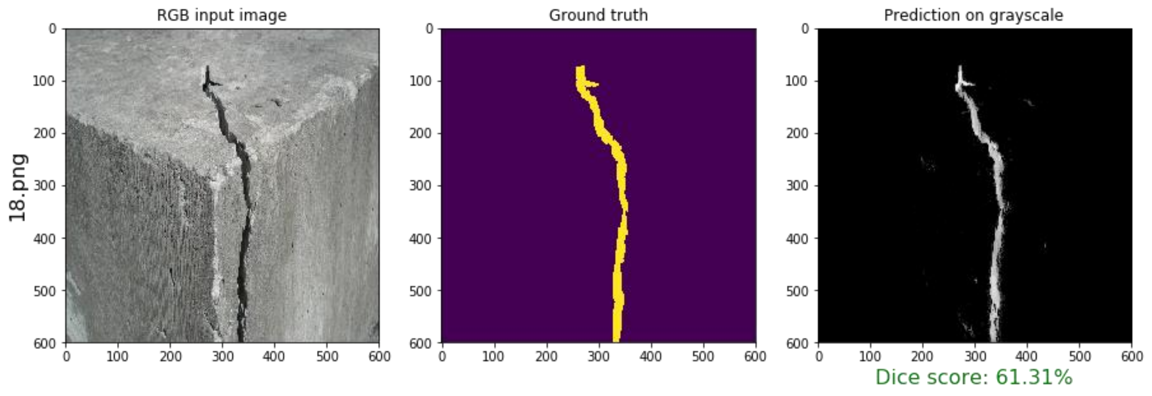
	Small crack	Confusing environment	Color of concrete
Small crack	32.png, 1.png	45.png, 50.png	
Confusing environment	45.png, 50.png	37.png	
Color of concrete			9.png

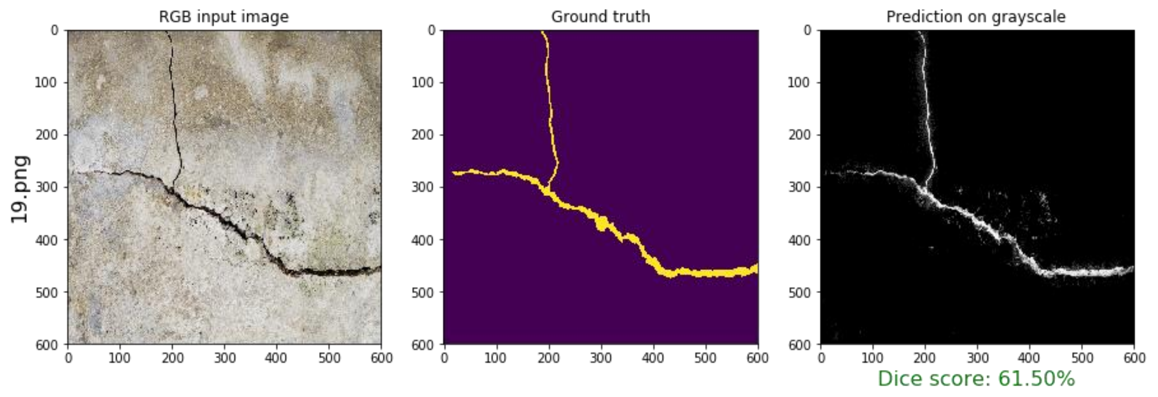
Best inference (Dice score > 60%)

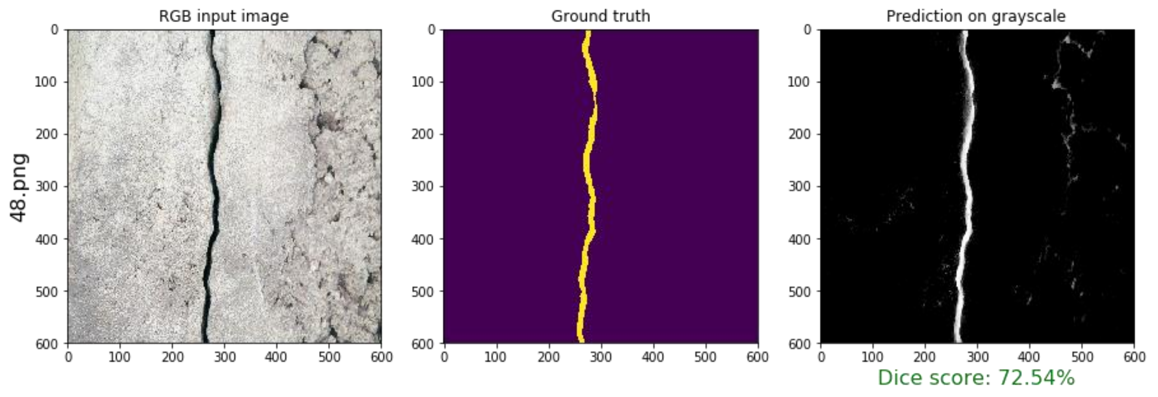








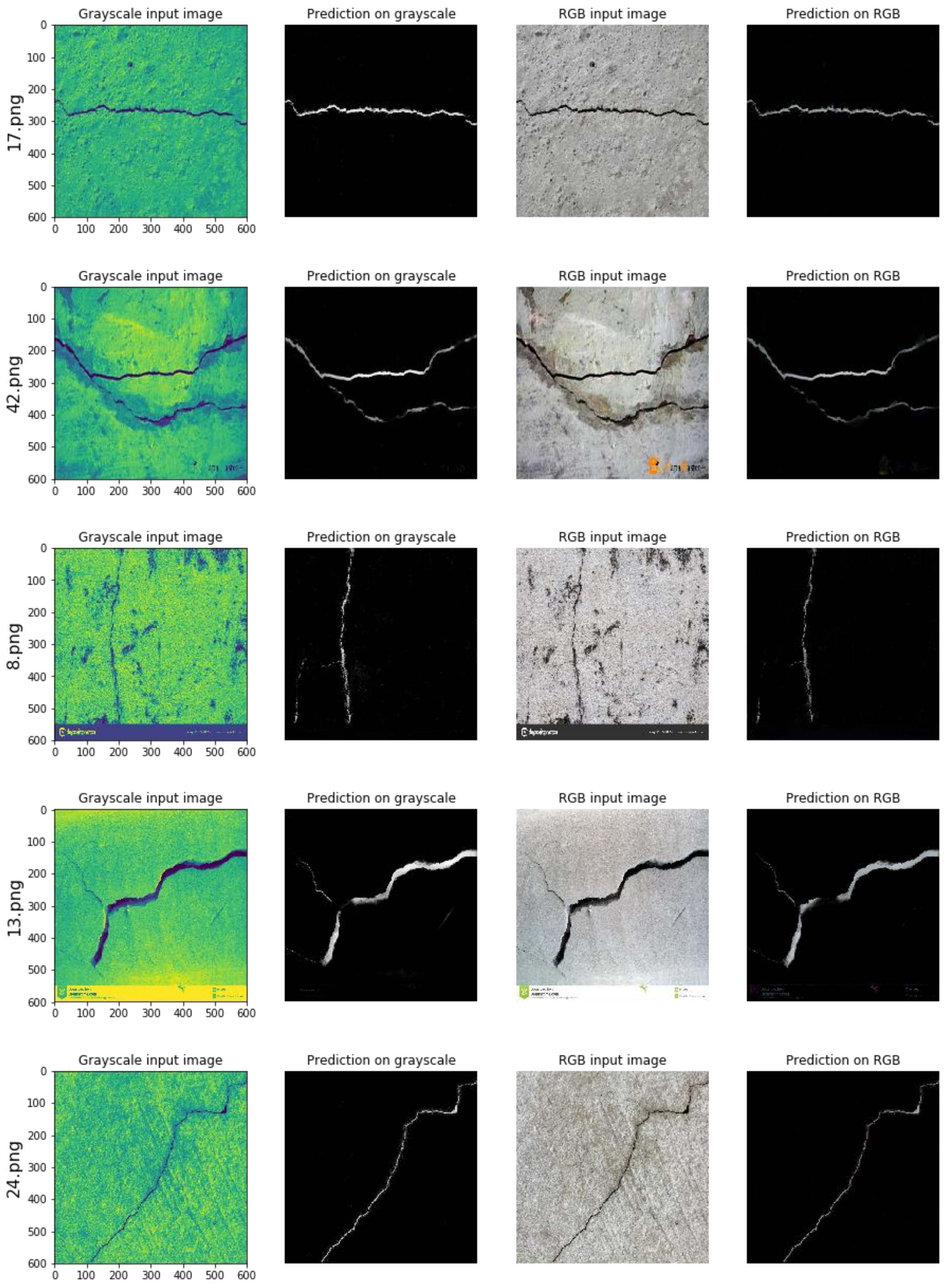




Interpretation: Images with the best inference are always thick cracks, on a smooth uniform concrete surface with the most common color of concrete, without confusing information around, often on the center of the image.

Conclusion: A future better dataset may benefit from data augmentation with focusing on the addition of randomly added shapes (arrow, stars, circle, line, rectangle…) in different colors, random modifications of color space. It is clear that large size images containing thin cracks suffered from the resizing operation to 600 x 600 pixels. For larger images some portion of 600 x 600 pixels must be randomly cropped to preserve the sensitive pieces of information, otherwise after image being resized crack may disappear from it.

Below you can see some randomly picked inferences.



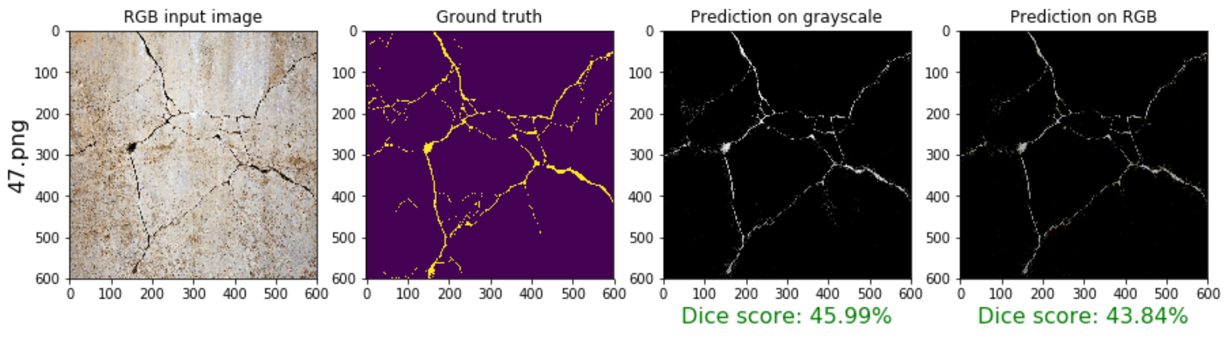
Justification

No benchmarks were used.

V. Conclusion

Free-Form Visualization

Satisfying detection which could take time by hand (>30 min) and only take 1/10s with the model



<--- MODEL BENCHMARK ON 50 IMAGES FOR EACH CHECKPOINT --->

50 epochs	Grayscale model --> 22.21% accuracy on mean dice score RGB model --> 20.69% accuracy on mean dice score
100 epochs	Grayscale model --> 23.58% accuracy on mean dice score RGB model --> 24.54% accuracy on mean dice score
150 epochs	Grayscale model --> 25.65% accuracy on mean dice score RGB model --> 19.78% accuracy on mean dice score
200 epochs	Grayscale model --> 27.34% accuracy on mean dice score RGB model --> 21.42% accuracy on mean dice score
250 epochs	Grayscale model --> 29.11% accuracy on mean dice score RGB model --> 23.05% accuracy on mean dice score
300 epochs	Grayscale model --> 28.73% accuracy on mean dice score RGB model --> 25.53% accuracy on mean dice score
350 epochs	Grayscale model --> 33.48% accuracy on mean dice score RGB model --> 22.81% accuracy on mean dice score
400 epochs	Grayscale model --> 26.16% accuracy on mean dice score RGB model --> 24.33% accuracy on mean dice score
450 epochs	Grayscale model --> 25.92% accuracy on mean dice score RGB model --> 27.21% accuracy on mean dice score
500 epochs	Grayscale model --> 27.60% accuracy on mean dice score RGB model --> 25.09% accuracy on mean dice score
550 epochs	Grayscale model --> 28.81% accuracy on mean dice score RGB model --> 25.86% accuracy on mean dice score
600 epochs	Grayscale model --> 29.46% accuracy on mean dice score RGB model --> 26.64% accuracy on mean dice score

---> Best grayscale mean dice score is 33.5 % after 350 epochs
---> Best RGB mean score dice is 27.2 % after 450 epochs

Reflection

The project may benefit from pretrained model using transfer learning or fine-tuning, different architectures, data augmentation. Now that the solution as proved to work, a good idea to make the model generalize would merely be to expand at least by ten the dataset's size. The model evaluation and validation part above provided interesting insight on what were the model's bottlenecks and is a good starting point to try improving it.

Improvement

The next step would be to increase the accuracy of the model, train it on other types of disorders and make it tractable on video to make it inference on high rate video.

Coupling this solution with lidars, radar and IMU it would even be possible to detect disorders and locate them in a 3D environment precisely and in real-time.