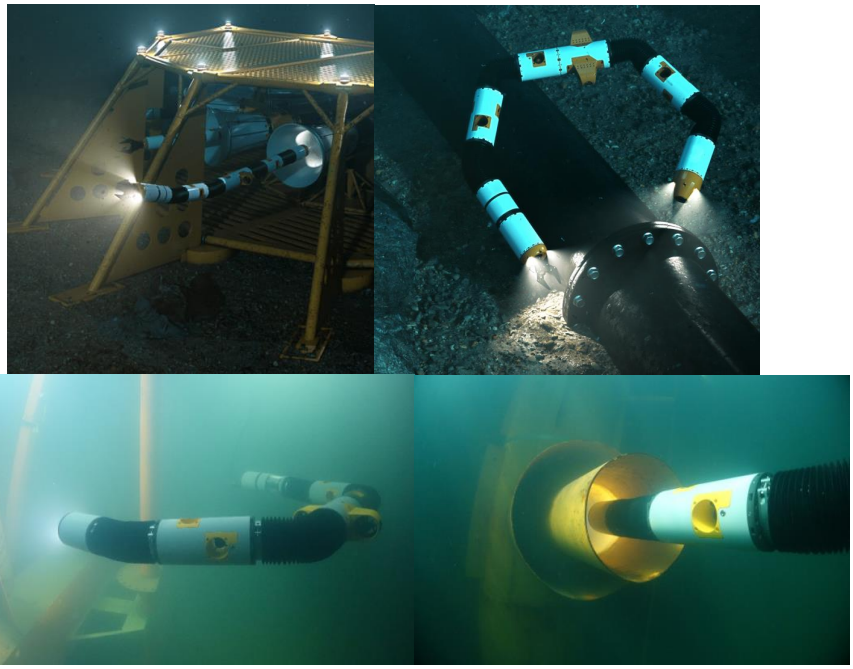
NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, og elektroteknikk
Institutt for teknisk kybernetikk

PROSJEKTOPPGAVE

Kandidatens navn:	**NN**
Fag:	**Kybernetikk og robotikk**
Oppgavens tittel (norsk):	Autonom navigasjon og mapping for undervanns slangeroboter
Oppgavens tittel (engelsk):	Autonomous navigation and mapping for underwater snake robots



In all major subsea industries, such as oil & gas and aquaculture, there is currently a large and increasing demand for autonomous underwater vehicles (AUVs) to carry out various types of inspection and intervention tasks. Very often, the need for autonomy is based on a need for reducing the use of costly surface vessels required to carry out underwater operations with remotely operated vehicles (ROVs).

To achieve high levels of autonomy in underwater robots, it is essential that these robots can use machine vision, such as optical cameras or sonars, to map their environment and navigate based on this map. This capability is the focus of this project and requires that the robot can carry out SLAM (Simultaneous Localization and Mapping), as well as autonomous motion planning and navigation in the mapped environment.

The project will be carried out in close collaboration with Eelume AS, a spin-off company from NTNU developing underwater snake robots in cooperation with Statoil and Kongsberg

Maritime. The Eelume vehicles are designed to live in docking stations on the seabed over extended periods in order to carry out operations on subsea infrastructure (visual inspection, valve operations, etc) without the need for surface vessel support. Autonomous navigation and mapping is an essential capability for the upcoming autonomy of these vehicles.

The objective of this project is to develop underwater SLAM algorithms based on optical cameras, where an Eelume robot is considered as the target platform. The Eelume vehicle is a particularly relevant platform for SLAM since its long body allows for a good spatial distribution of the cameras and lights used for mapping, thereby enabling cameras and lights from different angles. The overall objective is to enable the robot to develop a 3D map of its environment (such as the seabed, structures on the seabed, or floating structures) and also to enable the robot to continuously determine its own location as it moves in the mapped environment.

An additional objective is to develop algorithms where the vehicle autonomously plans and executes motion with respect to the mapped environment, such as:

- station keeping based on visual odometry (by fixating on features in the camera image).
- scanning around an object at a fixed distance.
- following a pipeline on the seabed.

An Eelume vehicle, as well as the lab basin of Eelume, will be made available to this project work for experimental testing. The use of a simulator currently being developed by NTNU and Eelume should also be considered for early testing.

As this task is intented to be worked on by three students it has been further devided into three main elements, SLAM, control and detection. The detection aspect aims to provide context for the map produced by the SLAM system. The student will focus on the detection and will work on the following:

1. Set up a framework to evaluate the performance of the pipeline detection algorithm.

2. Consider how the task of detecting an underwater pipeline can be done using computer vision and/or machine learning (deep learning).

3. Develop algorithms that will solve the detection of said pipeline.

4. Investigate problems that may arise from a realistic environment.

5. Propose improvements to the algorithm and if any alternative approaches may be beneficial.

The report shall be written in English and edited as a research report including Abstract, Introduction with motivation, literature survey, contributions of the project work, and the outline of the report. This is followed by the chapters describing the results of the project work, simulation results and corresponding discussion, and a conclusion including a proposal for further work. It is assumed that the Department of Engineering Cybernetics, NTNU, can use the results freely in its research work, unless otherwise agreed upon, by referring to the student's work.

Oppgaven gitt: 20.08.2018

Besvarelsen leveres: 21.12.2018

Utført ved Institutt for teknisk kybernetikk

Faglærere:

Professor Kristin Y. Pettersen, NTNU ITK

Professor Annette Stahl, NTNU ITK

Edmund Førland Brekke, Førsteamanuensis NTNU ITK

Medveiledere:

Pål Liljebäck, CTO Eelume AS

Abstract

Autonomy is becoming ever more present in industry and the field of underwater robotics is not exempt from this. This development may also be done best in increments, thus assisting the manual operations and gradually making the robot more autonomous. This report will look at the problem of underwater pipe detection with the goal of being of useful in an autonomous operation. The problem will be looked at from two perspective, the first utilizing traditional computer vision techniques. The second aims at solving some of the challenges in the real sea environment by basing itself on machine learning techniques. These techniques will be implemented and their performance evaluated.

Sammendrag

Autonomi vert ein stadig større del av industrien, dette gjeld og undervassrobotikk. Det kan vera hensiktsmessig å gjera denne utviklinga stegvis og ikkje nødvendigvis alt på ein gang. Slik kan roboten gradvis verta meir autonom, medan teknologien utvikla for dette vil vera med å gjera manualle operasjoner enklare. Rapporten ønskjer å undersøkja problemet å detektera røyrledningar under havet. Denne deteksjonen er tenkt å bidra til autonome undervassoperasjonar. Problemet vil verta undersøkt frå to ulike innfallsvinklar. Den første ser på tradisjonelle datasynteknikkar, medan den andre ønskjer å løyse ulike utfordringar ved reelle opperasjonsforhald ved å basere seg på maskinlæring. Begge innfallsvinklane vert implementert og deretter evaluert.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The project have two main aspects. The first and main focus is on the traditional computer vision techniques and how to use the information assist an autonomous robot in its operation. The second focus is on using machine learning techniques to gather the necessary information. The goal behind the traditional computer vision techniques is as stated to provide information about any pipes found in the image. While the goal behind the machine learning is to see if this can be used to handle challenges found in the underwater environment.

1.1 Motivation

Autonomy is gradually becoming an increasing part of society, though this does not mean that everything needs to be fully autonomous before implementation. The goal of this project is to have operator assisting autonomy, which might be a step on the bridge between manual and autonomous operations.

This report is part of a larger project in collaboration with Andreas Våge and Marcus Aleksander Engebretsen. The objective of this project is to develop underwater SLAM algorithms based on optical cameras, where an Eelume robot is considered as the target platform. The Eelume vehicle is a particularly relevant platform for SLAM

since its long body allows for a good spatial distribution of the cameras and lights used for mapping, thereby enabling cameras and lights from different angles. The overall objective is to enable the robot to develop a 3D map of its environment (such as the seabed, structures on the seabed, or floating structures) and also to enable the robot to continuously determine its own location as it moves in the mapped environment. An additional objective is to develop algorithms where the vehicle autonomously plans and executes motion with respect to the mapped environment, such as:

- station keeping based on visual odometry (by fixating on features in the camera image)
- scanning around an object at a fixed distance
- following a pipeline on the seabed

A workflow has been created that that represents the complete system which are the basis for the subsequent master thesis. This can be shown in figure 1.1. The following

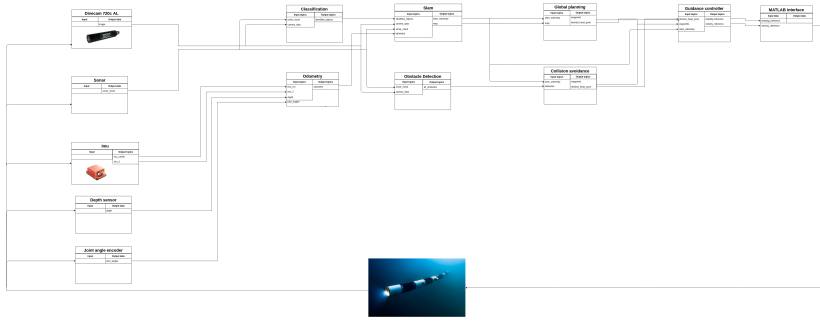


Figure 1.1: System work flow.

report will focus on the classification aspect of the workflow, seen in figure 1.2. More specifically the processing of the data that comes from the camera system. The goal behind the classification is to provide the SLAM system with context and thus giving the robot an understanding of not just where it is on the map, but also where objects of interest are in this map. Such objects could be a pipe or the docking station. Labels

in the map may be beneficial to the planning and control system in operations aiming to inspect a pipeline and return to the docking station. For the inspections aspect the real strength of the labeled map would be for routine inspections where, after the first inspection, the robot will have the location of the pipe in its map, which can then be utilized to plan the operation. In tandem with this report Andreas Våge is working

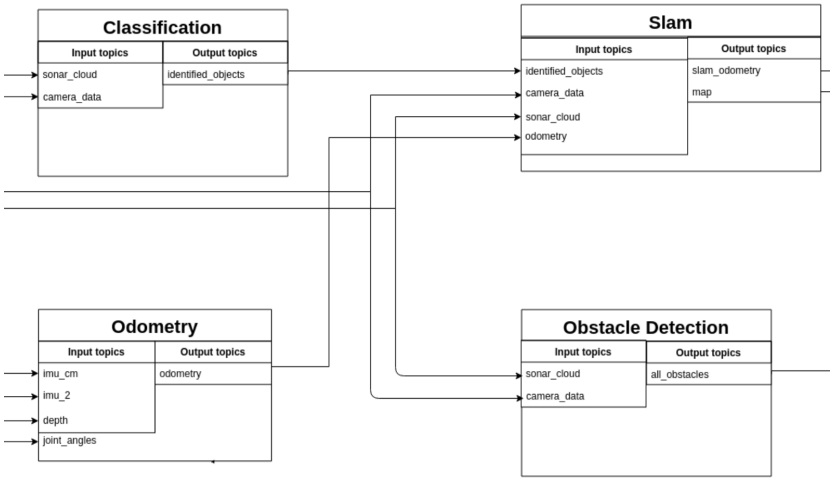


Figure 1.2: Classification system in larger context.

on the SLAM system, while Marcus A. Engebretsen is working on the control system of the robot. It has been decided to implement the complete system in ROS (Robotic Operating System) since there are many seperate modules that need to communicate and work together to complete the system.

1.2 Problem description

The project aims to navigate in an underwater environment and thus needs information about this environment before making decisions. The project will look at the part of identifying an underwater pipe and thus providing the robot with information to enable it to track the pipe.

The project aims at two distinct scenarios, the first being a controlled pool environment, while the other is a real pipeline in the sea. The pool environment represent the simplified case where we aim to find the pipe under very controlled conditions. An example of this can be seen in figure 1.3. The sea environment needs to tackle a

Figure 1.3: Pipe in pool environment.

variety of added challenges. An example can be found in figure 1.4. The most notable

Figure 1.4: Pipe in real environment.

is arguably the biofouling of the pipe which makes it so that it is hard to estimate the pipe as two large edges. A marine environment with natural water have a large

number of macro-molecules that attach to surfaces and facilitate marine growth (Melo and Bott; 1997). Melo and Bott (1997) present a figure indicating the idealized biofilm development where there is a slow formation period, followed by rapid growth up to a plateau. As pipes under water are expected to stay submerged for longer periods of time, it seems to be a reasonable assumption to expect the biofouling to has reached the plateau state. Therefore, it is important to rely on a detection scheme that does not expect an ideal pipe, completely exempt from the biofouling.

1.3 Literature review

Before building any program it is important to have a clear understanding of what it should do. In addition it is valuable to see what have already been done in the field. This approach ensures that the scientific field may work together towards an accumulative bank of knowledge and not unnecessarily duplicate solutions to known problems. As this report propose techniques that are to be implemented, it is important to see if there are any libraries that can be beneficial to use as a foundation and what should be implemented from scratch. As mentioned in section 1.1, the complete system should be implemented in ROS and it is therefore important to develop the proposed algorithms so that they are compatible with ROS. There are currently three programming languages that are fully compatible with ROS; Python, C++ and Lisp (Dattalo; 2018). As the contributors, Engebretsen, Utbjoe and Våge, to the complete system where most familiar with C++ and Python it was decided to implement the system in C++. Programs that are not intended to be part of the complete system may be developed in Python.

With the criteria on programming language and framework decided it seemed a natural choice to implement the system utilizing the open source computer vision library OpenCV. There are a few different reasons for this. First and foremost it is a computer vision library that is compatible with ROS (Allevato; 2017). Secondly the OpenCV library are compatible with both C++ and Python (Bradski and Kaehler; 2008).

The natural next step is to explore which computer vision methods are available through OpenCV and which of these might be applicable to the problem the algorithm

should solve.

1.3.1 Image processing

As an image often can have noise in it, it is beneficial to perform a smoothing or blurring operation on the image prior to any further processing. OpenCV have five varieties proposed for performing the smoothing operation (Bradski and Kaehler; 2008); simple blur, simple blur with no scaling, median blur, gaussian blur and bilateral filter. According to Bradski and Kaehler gaussian blur is probably the most useful, though it is not the fastest (Bradski and Kaehler; 2008). This will then be a good starting technique, though should it be necessary with a faster processing time it might need to be swapped for a different technique. OpenCV also has implemented higher performance optimization for kernels of size 3x3, 5x5 and 7x7 (Bradski and Kaehler; 2008).

There are also morphological transformations available, such as dilation and erosion. Both dilation and erosion applies a kernel to the image and evaluates the values in the kernel to decide what to do with the pixel in question. Dilation will set the pixel to the maximum value in the kernel, while the erosion will choose the minimum value (Bradski and Kaehler; 2008). Erosion is often used to remove small specks in the image, while preserving the more significant features (Bradski and Kaehler; 2008). For instance if one applied a dilation operation after an erosion the more significant features will be restored to their original sizes. The dilation operation can be used to merge together areas with small gaps dividing them (Bradski and Kaehler; 2008). For instances where there are white areas in the image with dark speckles in it, a dilation operation followed by an erosion will fill in these specks while returning the areas to their proper size again with the erosion operation.

Thresholding is another technique that is available through OpenCV. In thresholding the values are sorted based on whether or not they are within the specified bounds. Bradski and Kaehler (Bradski and Kaehler; 2008) present two main techniques for doing thresholding, binary and adaptive binary thresholding. The adaptive binary thresholding differs from the binary thresholding by evaluating the surrounding area

when deciding the bounds. Thresholding is used to make a final decision about the pixels (Bradski and Kaehler; 2008), for instance deciding whether the pixel is of a certain colour.

1.3.2 Image transformation

Convolution is when all areas in the image are filtered through an operation specified by a kernel of a certain size and with the values of the kernel deciding what the result of the convolution is. Convolution is the basis of many of the subsequent transformations to be discussed (Bradski and Kaehler; 2008).

When looking at the image it is often useful to observe where in the image changes occur. When talking about changes over time or distance it is natural to look to derivatives of the said expression. The sobel derivative is an approximation it tries to fit a parabolic function over a local area of pixels (Bradski and Kaehler; 2008). The sobel derivatives can then produce a sort of gradient image in the x and y direction. Since the sobel is an approximation it is less accurate for small kernels and one can then use the Scharr filter to preserve the speed and accuracy when using a small kernel (Bradski and Kaehler; 2008).

When detecting features, such as blobs, using derivatives one can use a method such as the Laplacian operator (Bradski and Kaehler; 2008). The Laplace operator can also be used for edge detection, though this technique was later refined into the Canny edge detector (Bradski and Kaehler; 2008). The canny algorithm computes the first derivatives in the x and y direction and then combine them into four directional derivatives, which are then used to locate local maxima that are candidates to be assembled into edges (Bradski and Kaehler; 2008). The edge detection seems to be a good candidate for finding and classifying pipes.

Once the edges of the image is found it is necessary to find any lines present as it is a fair assumption that the pipe has some lines pointing along the direction of the pipe. A technique for finding these lines is the Hough line transform. The Hough line transform bases itself on the principle that any point in the binary image can be a part of some set of possible lines (Bradski and Kaehler; 2008). By evaluating all points

in the binary image one can see which lines are present by finding the local maxima among the possible lines (Bradski and Kaehler; 2008). OpenCV has implemented two variations on the Hough line transform, the standard Hough transform and the progressive probabilistic Hough transform (Bradski and Kaehler; 2008). The difference is that the latter only accumulate a fraction of the possible points and relies on the heuristic that any significant lines will still be present when finding the local maxima and thus saving computation time (Bradski and Kaehler; 2008).

1.3.3 Machine learning

Machine learning is a large field containing multiple sub-fields. One of these fields are deep learning. Traditional machine learning methods had problems when processing data in its raw format and thus it was necessary to construct a feature extractor to process the data prior to the network into a suitable representation, such as feature vectors (LeCun et al.; 2015). Deep learning methods learn these representations themselvs based on the raw input (LeCun et al.; 2015).

1.3.3.1 Deep learning

Deep learning methods generates the data representation and classification through multiple layers that, for feed foreward networks, process on each other generating this representation into higher and more abstract levels (LeCun et al.; 2015). An analogy for the low, medium and high level feature may be viewed as follows. Say that the network wants to classify if something is a face or not, then the low level features may identify where in the images there are found lines, curves, circles etc. The medium level features may learn that a combination of lines, curves and circles may constitute eyes, mouths, ears, etc. The high level feature may then learn that say two eyes with a small distance between them followed by one or two ears on the sides and a mouth below will constitute a face. This analogy is only meant to illustrate how a network may learn it, it is not by any means certain that this is exactly how it would occur in a real network.

There are two main approaches to the training process, unsupervised and super-

vised. The former aim to find patterns in the data, while not necessarily give context to the data, while the latter provides a solution to the network in the training process to enable it to give context to the classifications. In supervised learning we would typically try to train a classifier to classify a certain number of classes and the network may then be shown an input image and outputting a vector with the probability of the presence of each class in the image. This will then be compared with our ground truth for that image and used to calculate an error to be used in backpropagation (LeCun et al.; 2015). This error is then backpropogated to adjust the weights in the network and thus train the network (LeCun et al.; 2015). This adjustment of the weights is done using a gradient vector based on some objective function in order to ensure that the weights find a local minima (LeCun et al.; 2015). A type of deep learning networks are convolutional neural networks, CNNs, which are composed of two main type of layers, convolutional and pooling (LeCun et al.; 2015). The convolutional layer takes the input and subjects it to a set of filters in order to produce the output. This produce multiple outputs based on the input, so in order to reduce the spatial complexity the output is subjected to a pooling layer which aims to preserve the information while keeping the most important information. A common way of pooling is the max pooling which keeps the highest value in the pooling kernel and thus preserving the highest values. A typical structure of a CNN can be multiple layers of convolutions and pooling layers which performs the feature representation that are then presented to a fully connected network to classify the feature representation.

As the low level features are more general it can be generalized for various applications and may be applicable for for problems that the layers is not necessarily trained for. This can be done by utilizing pre-trained networks, as have been done successfully by Lee, D. and Lee, S. (Lee and Lee; 2019).

1.4 Assumptions

In order to reduce the scope of the project there are several assumptions made. On the control section it is assumed that the vessel using this technique has a separate control system that can control the vessel to achieve the desired direction given from

the vision system. It is also assumed that the pipe in question does not split into several sections, causing there to be more than two pipe segments. This may be a realistic assumption for long range pipe tracking. However, extending this technique to handle pipes that are more complex than a simple bend will be a natural extension of this work.

This project is also part of a larger collaboration and therefore several aspects that may have been natural to include in this report will not be explored much if they are covered by the other collaborators. An instance of this is the control aspect as Marcus Aleksander Engebretsen work on that area. The report will therefore assume that if the system can provide the spatial location of the pipe in the image, then the robot will be able to be controlled adequately by separate modules.

Unlike monocular vision, stereo vision has the ability to extract depth information about the surrounding and may therefore be useful when the information provided by the vision system are going to be used for control purposes. In order for the results to be of any value we need to calibrate the cameras. Andreas Våge has worked mainly on the SLAM section using stereo camera. This report will therefore assume that depth information may be provided from this SLAM system. This project will not look further into stereo vision and camera calibration as this is a larger focus in the work of Andreas Våge.

1.5 Background and Contributions

The project are using pre-existing libraries such as OpenCV and ROS. ROS is used to be able to communicate between the different modules of the system work flow. OpenCV is used as a computer vision library to process the frames. The deep learning part of the project builds part of its foundation on the project "Autogenerated training data using CV techniques" from TDT-4265 computer vision and deep learning conducted by Ole S. Otterholm, Runar A. Olsen and Øystein B. Utbjoe.

1.6 Outline

The report is organized as follows. In Chapter 2 the systems proposed by this report is presented and the rational behind them explained. There will be one presentation for the machine learning and one for the traditional computer vision. In Chapter 3 the results of the two systems will be presented as well as discussed. In Chapter 4 the conclusion will be presented as well as some suggestion for future improvements.

Chapter 2

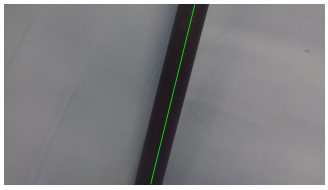
System description

2.1 Traditional computer vision

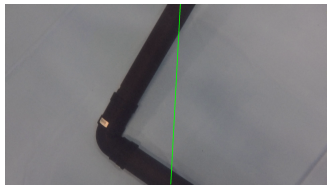
In the traditional computer vision system it is necessary to use feature engineering to determine what is classified as a pipe. The project will focus on monocular vision when designing the system.

2.1.1 Modelling a pipe

In order to detect a pipe it is necessary to have some sort of model of what constitutes a pipe. The subsequent algorithm is based on the assumption that the pipe can be modelled as a line. This can be seen in figure 2.1a. The first and more obvious weakness of this is how to handle bends. This is a weakness that we see from the results, but is also to be expected. For the ground truth it is assumed that the lines are straight from where it enters the image to where it leaves the image. Bends are treated as part of a triangle where the resulting ground truth is the third side. This can be seen in figure 2.1b. With the pipe modelled as a line it is needed to have some sort of mathematical representation of a line.



(a) Straight pipe.



(b) Pipe with bend.

Figure 2.1: Ground truth modelled.

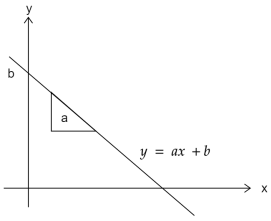


Figure 2.2: Line with $y = a \cdot x + b$.

Maybe the most obvious would be to represent it using $y = a * x + b$ where x and y denote the pixel axis. A representation of this sort is shown in figure 2.2. This is dependent on the slope of the line and if we align the pipe vertically the slope approaches ∞, which is not desired. A more preferable representation can therefore be to model it as an angle, θ and the tangential distance from the optical centre, r. An illustration of this can be seen in figure 2.3.

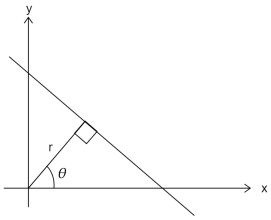


Figure 2.3: Line represented by θ and r.

For the first round of the algorithm it is only focused on finding the angle of the pipe. The tangential distance, indicating the spatial location, is reserved for future improvements.

2.1.2 Detection algorithm

One of the main facets of monocular vision is that it will struggle to extract depth information. There are methods that can overcome this issue, such as a pair of lasers with known relative distance, though in this project we do not have any such technique available.

In the case of detecting an underwater pipe there needs to be some feature common to pipes that can be used to find them in the image. One of the first techniques that comes to mind is colour segmentation, however upon further consideration colour segmentation seem to be a poor solution. Such a scheme would require prior knowledge about the pipes in question and will be troublesome due to the rapid dissipation of colors in water. This dissipation would render the system with a rather short operating range. With color segmentation deemed unsuitable there would need to be some kind of evaluation of the contrast in the image which is unrelated to the colors.

When considering how pipes will appear when examining contrasts in the image two considerations come to mind. The first is based on edge detection where one would then search for the two lines which would represent the outside of the pipe. The second is based on thresholding where the pipe is assumed to have different coloration from the surrounding area. The first method seems to have the most basis as this would be least dependent on assumptions about the image.

The technique will utilize edge detection, and it is therefore important to filter out noise from the image. This will make it so the edge detection only register larger trends in the image. This noise reduction is done using a Gaussian blur filter. After the image has been blurred it is then subjected to an edge detection using the Canny method. The output of the edge detection is a binary image, where edges are denoted with a value of 1.

The next step of the algorithm is to identify the lines present in this binary image. However, in order to improve robustness by connecting, the image is first subjected to a dilation followed by an erosion. The purpose behind this is to join together lines that are interrupted by sporadic pixels with 0 values which break up the line, but are still part of the pipe edge. After the dilation and erosion the algorithm endeavors to identify the lines in the binary image. The method chosen for doing this is the probabilistic Hough lines transform.

Once the lines in the image are gathered they need to be sorted and determined how their information can be utilized to their full extent. The report considered two ways to classify the lines as a pipe. One where the longest two lines are kept and used as representations of the pipe. The other where all the lines perform a vote over which

directions are the most prevalent. Both of these techniques have their advantages as well as challenges that need to be utilized and solved. The simplest case which checks the two longest lines assumes a near ideal pipe, which is not realistic in a realistic environment. The second method is therefore preferred.

A fairly obvious challenge with the voting scheme is that there is a possibility that small lines can outvote the pipe direction due to their shear number. In order to combat this it is introduced a weighting on the lines to give more emphasis on longer line segments. There is also an issue that the outliers distort the proposed pipe. These outliers should ideally not be considered when deciding where the pipe is.

There was an issue when the pipe was oriented vertically in the image, shown in figure 2.4. The reason why this issue arose is that lines in the image have two alternative representations when examining the non directed lines. Initially the angle was calculated between $-\pi$ and π using atan2. When the pipe was oriented vertically the atan2 method provided a mix of these two representations resulting that the average was oriented somewhere in between. This was solved by adding π to any values found between $-\pi$ and 0, meaning that the algorithm only looks at the representation between 0 and π, where the lines have a unique representation.

(a) Original (b) Analyzed

Figure 2.4: Issue with vertically oriented pipe.

The first proposed method used the average of all the detected lines and then used this average as the proposed pipe angle. This method had a weakness that it struggled to correctly evaluate bends and was very susceptible to outliers. To combat these issues an alternative scheme is proposed.

This scheme prepares an array of which each element represent a section of angles. When the algorithm finds a line it will add the magnitude of this line to the correct element of the array. As it iterates through the detected lines the angle section containing the largest value will be the direction of the lines. For the first iteration it divided the section of between 0 and π into 36 sections, but after some consideration this was increased to 180 sections which slightly improved the performance. With this sorting scheme the algorithm proves much more robust against outliers, though it still struggles with bends.

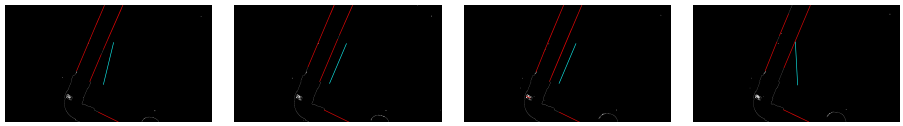
In order to identify the bend it is necessary to have some understanding on how to categorize the bend. The algorithm considers that the pipe may have one main and one secondary pipe segment. It assumes that secondary pipe segment is of sufficient size in relation to the main pipe section. For the first case this was set as one fifth of the main pipe. This heuristic proved to be quite strict and there was a performance increase of around 1 percent when decreasing it to one tenth.

The algorithm performed better now, though it did not consider the relative difference in magnitude between the main and secondary pipe section. To solve this relative difference the algorithm calculated an $\alpha_w = \frac{\beta_{main}}{\beta_{main}+\beta_{secondary}}$, α_w being the weighting factor and β being the magnitude in the angle section of the main and secondary pipe section. The average between the main and secondary angle is then computed with $\theta_{average} = \theta_{main} \cdot \alpha_w + \theta_{secondary} * (1 - \alpha_w)$. This weighted evaluation between the main and secondary pipe section proved to be a significantly improved representation of the pipe bend.

There where still some frames that was expected to be correct, but still fooled the algorithm. When examining which of the frames that failed the test, these frames where found in between frames corrected with the weighted two pipe section scheme. Figure 2.5 shows four consecutive images that illustrates the issue. From 2.5b and 2.5c it seems that the algorithm only use the main pipe section, while 2.5a and 2.5d use the secondary pipe section as well. This indicates that the heuristic of one tenth is too strict. When further lowering the heuristic, the performance dropped and the same issue persisted. This means that the heuristic was not the problem. The next theory was that the largest pipe section is located right on the border of the prepared angle

sections and thus constitutes the largest and second largest values. This essentially hides the smaller pipe segment. When implementing this improvement the issues from 2.5b and 2.5c vanished and the performance rose.

Even with this improvement there where still some frames that proved elusive. In order to improve the robustness against such rogue frames it was implemented a sliding window processing. The algorithm essentially remembers the results of the n_{th} previous frames and evaluates the pipe angle based on the information gathered over all these frames. This should suppress any outliers. This sliding window approach is based on the assumption that the pipe does not move significantly between a small set of consecutive frames. This proves to be a reasonable assumption in the relative slow moving robot compared to the frame rate of the camera. This implementation resulted in an improvement of several percentage points as well as a smoother behaviour of the estimated line.



(a) Image 206 (b) Image 207 (c) Image 208 (d) Image 209

Figure 2.5: Bend problem.

After implementing the sliding window there was a definite improvement. The new performance can be seen in figure 2.6, which shows the same sequence of frames with the sliding window implemented. Though there was still the question of how many frames this sliding window should contain. The fear was that too few would loose the value provided from the sliding window approach, rendering it useless, while too many might introduce a sort of inertia that was too large on the pipe. To determine what the optimal number of frames where, the values where started at 10 and increased until the performance started to drop and then a few more frames to see that it was a larger trend. From figure 2.7 it seems that the performance increase when increasing the sliding window scope from 10 until it reach an optimum around 15 to 16, before decreasing as it reaches 20. This is in line with the qualitative observations where the

line seemed to drag due to this increased pseudo inertia when increasing the frames of the sliding window scope too much. A sliding window scope of 15 frames was therefore chosen for further implementation.

(a) Image 206 (b) Image 207 (c) Image 208 (d) Image 209

Figure 2.6: Bend with sliding window.

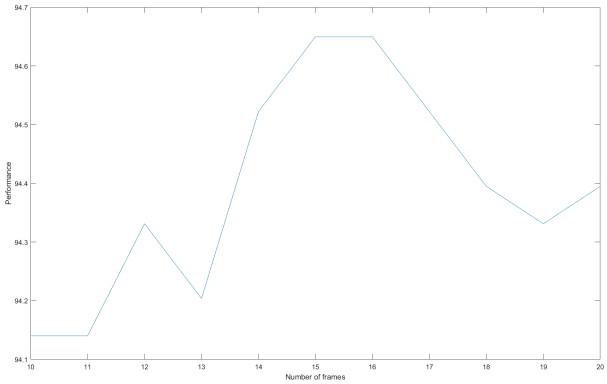


Figure 2.7: Sliding window performance.

Figure 2.8 shows the resulting system proposed by this report. In short it reads a frame from the camera, which is then passed through a variety of image processes until it has found all the lines in the image. These lines are then passed to the line sorting scheme described above before passing the proposed pipe to the sliding window comparison. The sliding window comparison compare the proposed pipe with the proposed pipes from the 15 last frames before selecting a detected pipe.

With the scheme working as intended it will be interesting to see how the scheme

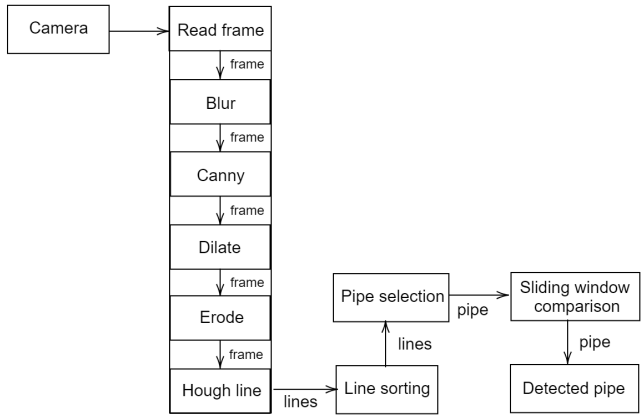


Figure 2.8: Flow chart pipe detection.

perform when exposed to a variety of different tuning parameters to see if it is sensitive to its tuning and which parameters have the most impact on its performance. After finishing the tuning it will be tried with the real video. As this does not have a ground truth it will be evaluated based on observation. The main goal behind this is to see if it works at all or not. Should it work well it will be developed a ground truth to see how well it performs, however, should it fail miserably this will not be necessary.

2.2 Machine learning

In the project "Autogenerated training data using CV techniques" it is proposed a technique where realistic training data is obtained and labeled from a simulation before being used to train a neural network. Then the network tried to categorize images from a real operation. The network performed well on the training data, but struggled to handle the real world images. This project propose that it might be advantageous to train the network on a mixture of real and simulated data. The goal

behind this is to make the network perform better than expected if it was only exposed to a limited number of training data. Figure 2.9 shows the architecture of the proposed network. This network utilize transfer learning and is built upon the output of the existing network NasNetLarge, a well performing network for image classification (Zoph et al.; 2018). It then receives the output and takes it through a pooling layer before giving it to a fully connected layer with 64 nodes that utilize the ReLu activation function. The last step is a layer with one node that utilize the sigmoid activation function to make the binary prediction. The project will use this architecture for as long as it proves to be sufficiently complex.

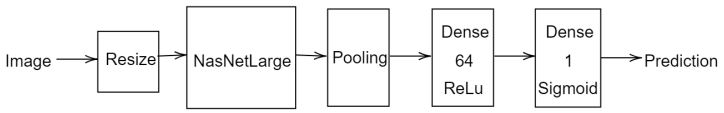


Figure 2.9: Network architecture.

Earlier, the project has only used the network for binary image classification, which is not the most useful for navigation. Should the simulated data be able to improve the performance of the network it can in theory be expanded to other techniques. Figure 2.10 shows how the simplified binary image of the simulation looks before it is classified by whether or not the image contains a pipe. This binary image is in essence a pixel by pixel ground truth on what constitutes a pipe and should therefore be very well suited for other applications, such as object detection or even semantic segmentation.

The thing of interest for this report is not necessarily how well the network perform, but rather if the mixture between real and simulated data can improve the result when testing on real data. The main contribution of this report is therefore the sorting and labeling of the data set combined with looking at how this can be augmented using different ratios of simulated data. The data set is based on a video provided from Marco Leonardi. This video was split into separate images using the developed script split_video.py in the attachment. This resulted in more than 86 000 images and after

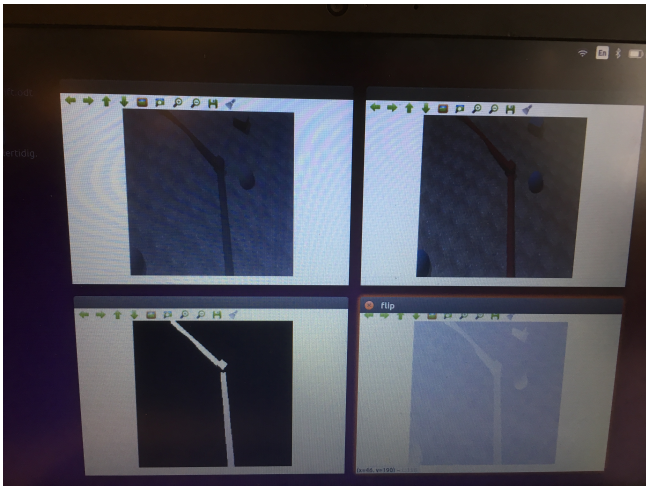


Figure 2.10: Simulated training data, from the project "Autogenerated training data using CV techniques".

sorting through around 6 000 it was decided that this was too many to be suitable for this purpose. The script was then altered to only save every tenth image and drastically reducing the images to be sorted, while giving a representation from the entire video. The resulting images were then sorted manually into separate folders based on whether or not the image contained a pipe. In this data set there where 5 710 images containing pipes and 1 469 images not containing any pipes. This data set is used as the test to see how well the network performs after training.

The original video had several overlay features, as seen in figure 2.11, which would not be ideal to keep for the training of the network. The image was therefore cropped to remove the undesirable overlay, an example of this can be shown in figure 2.12.

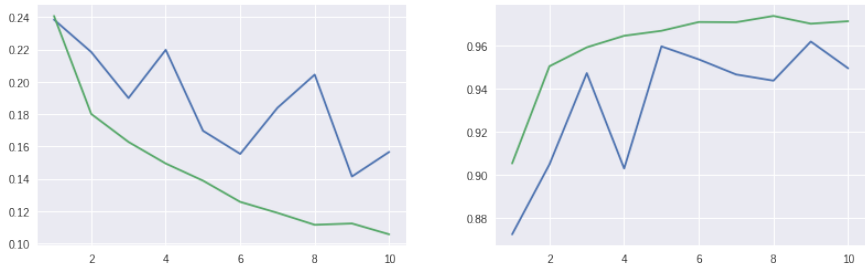
With the entire data set labeled a natural next step was to split the data set into training, validation and test data. This was done using the second script sort_training.py in the attachments, which reads through the labeled data set and sorts the image randomly into either training, validation or testing. The sorting was done so that one tenth would be reserved for testing. Of the resulting images not reserved for testing a

Figure 2.11: Overlay on video.

Figure 2.12: Overlay removed.

quarter was reserved for validation and the rest for training.

With the data set completed it was necessary to get a baseline of how well the network proposed in the project "Autogenerated training data using CV techniques" would perform when subjected to a large amount of real training data. The network was therefore subjected to the data set sorted into training, validation and testing. The data set contained 4874 training images, 1601 validation images and 704 testing images. The network was set up with a batch size of 32 and set to train over 10 epochs with 200 steps per epoch. After the final epoch it achieved the test accuracy of 96%, which indicate that the network has the capacity to classify the images in the data set. From figure 2.13 it seems that the network trains correctly. As the network trains correctly and the goal of the project is only to see how a mix of simulated data may effect the performance there is no need to alter the architecture of the network.



(a) Loss (b) Accuracy

Figure 2.13: Real data results.

Now that the network performs as intended it is interesting to see how it behaves when trying to classify the real data, but only being trained on the simulated data. Then it will be subjected to a blend of simulated and real data to see if that may affect the performance in any meaningful manner. Intuitively it may be reasonable to assume that the network will perform best if given a large set of real data. Thus it is interesting to see how the network perform when subjected to very small amounts of real training data.

As previously stated this part of the project is based on the project "Autogenerated

training data using CV techniques" and the fourth data set proposed in this project will be the one used when supplementing the reduced set of real data. The real data set was then reduced so that each of the folders contained roughly five percent of the amount of images in each of the folders for the simulated data set. The images that are not kept for training or validation will be the data used for testing to evaluate the overall performance after the training is complete. This reduced set of real data will serve as the base line. The simulated data will then be added so that the real data constitutes close to five percent of the training and validation data. This percentage will then be increased by reducing the amount of simulated data added to see how the ratio between simulated and real data will effect performance of the network. The various scripts developed for the machine learning as well as the various data sets are located in the attachments.

Chapter 3

Results

3.1 Traditional computer vision

In order to have an ordered testing regime it is beneficial to have some sort of ground truth. This was done by manually indicating the direction on the pipe for each image and store them in the file, which can be found as ground_truth_finished.py in the attachments. In order to streamline this process it was developed a python script, create_ground_truth.py in the attachments, that displays the image waiting for left mouse clicks as input. The first input is accepted and used as an anchor for the line that is drawn to the mouse position at all times until the next second mouse click is received. This line is to aid in labeling the ground truth. On the second mouse click the image number, line angle and click position are saved to a text file for later comparison. There is also implemented a cancel button which allows the first mouse click to be provided again in the case that this was put in the wrong location. A python script was also developed, compare_result.py in the attachments, comparing the results of the pipe detection algorithm with the ground truth. In addition, a python script that splits the video into images named based on the number of frames that preceded the image in question, was developed. The subsequent tuning parameters are therefore compared with how well they perform against the human created baseline. This can

Kernel	Canny	Threshold	Line min size	Line max gap	Performance
3 x 3	10 - 50	120	100	20	94.65
5 x 5	10 - 50	120	100	20	91.02
7 x 7	10 - 50	120	100	20	91.53
9 x 9	10 - 50	120	100	20	91.08
11 x 11	10 - 50	120	100	20	90.89

Table 3.1: Tuning kernel size.

be shown in figure 3.1. With the ground truth established the next step is to see how the different tunable parameters will effect the performance of the algorithm.

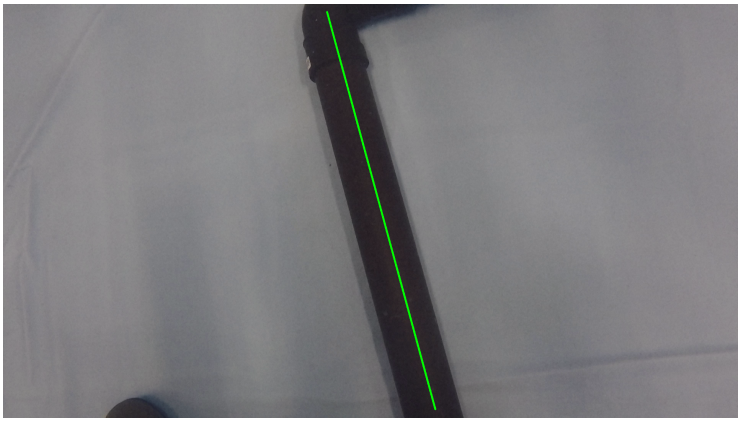


Figure 3.1: Ground truth image.

From table 3.1 it becomes apparent that enlarging the kernel size of the dilation and erosion kernel has a negative impact on performance. This makes sense as this process would bundle together outliers so that they have a higher impact on the performance. As increasing the kernel size did not have a positive impact on the performance it was decided to continue using a kernel of size 3 by 3.

For the tuning of the edge detection it was decided to start with the values of 10 - 50 and then trying to change both by increasing and then decreasing the values to see

Kernel	Canny	Threshold	Line min size	Line max gap	Performance
3 x 3	10 - 50	120	100	20	94.65
3 x 3	10 -30	120	100	20	70.13
3 x 3	30 - 50	120	100	20	94.20
3 x 3	20 - 50	120	100	20	93.63
3 x 3	10 - 40	120	100	20	82.29
3 x 3	10 - 60	120	100	20	94.46
3 x 3	20 - 60	120	100	20	94.46
3 x 3	30 -60	120	100	20	95.10
3 x 3	40 - 60	120	100	20	94.84
3 x 3	30 - 70	120	100	20	94.52
3 x 3	35 - 60	120	100	20	93.57
3 x 3	25 - 60	120	100	20	95.10
3 x 3	25 - 55	120	100	20	95.29
3 x 3	25 - 50	120	100	20	93.82
3 x 3	24 - 55	120	100	20	95.29
3 x 3	23 - 55	120	100	20	95.41
3 x 3	22 - 55	120	100	20	95.41
3 x 3	21 - 55	120	100	20	94.90
3 x 3	26 - 55	120	100	20	94.78
3 x 3	23 - 56	120	100	20	94.46
3 x 3	23 - 54	120	100	20	95.41
3 x 3	23 - 53	120	100	20	93.63

Table 3.2: Tuning canny edge detection values.

how this impacted the performance. In table 3.2 we can see that lowering these values significantly will have a large negative impact on the performance while increasing them above certain values will similarly impact the performance negatively. When the values was too low, as shown in figure 3.2, there were too many edges accepted by the edge detection making the line detection struggle. In the other case where the values where too large, as shown in figure 3.3, there are too few edges accepted making the line detection struggle, not provided with a sufficient amount of the edges present in the image. After much tuning the values 23 - 55 had the best performance with 95.41% accuracy. Therefore, these were selected as the values to continue using for further tuning.

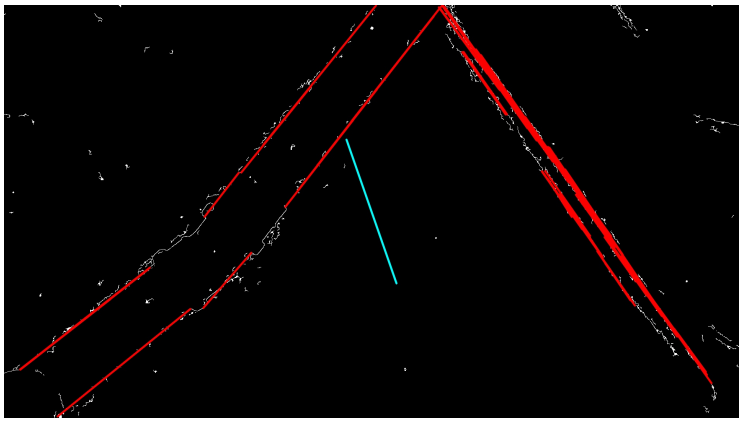


Figure 3.2: Too low canny values.

The last tunable parameters were for the probabilistic Hough line transform. From table 3.3 it seemed that changing the threshold value had no positive effect, causing it to be left at 120. Lowering the minimum line length had a positive effect causing it to be set to 10. Lowering the allowed line gap had little to none effect.

With the tuning complete the algorithm was then subjected to a section of the real video. From figure 3.4 it seem to be working as intended, though if we look at figure 3.5 it seems that when the camera approaches close to the seabed it starts to pick up outliers and thus effecting the performance. This indicate that the canny should be

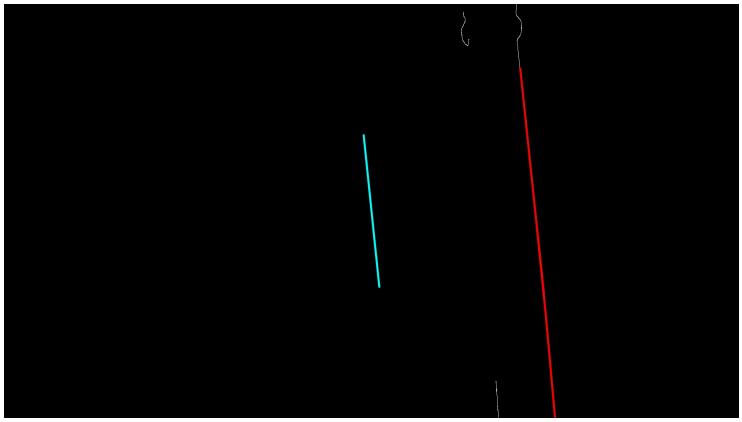
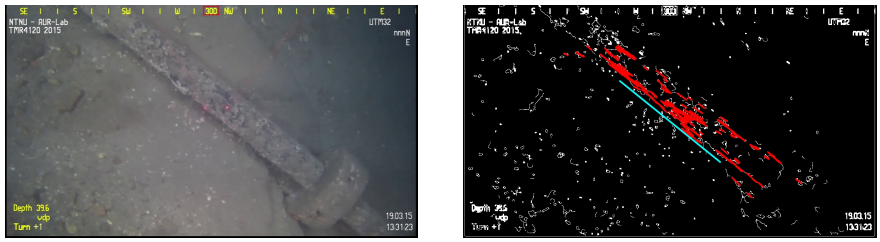


Figure 3.3: Too high canny values.

Kernel	Canny	Threshold	Line min size	Line max gap	Performance
3 x 3	23 - 55	120	100	20	95.41
3 x 3	23 - 55	100	100	20	94.71
3 x 3	23 - 55	130	100	20	94.97
3 x 3	23 - 55	110	100	20	93.50
3 x 3	23 - 55	120	50	20	95.48
3 x 3	23 - 55	120	10	20	95.86
3 x 3	23 - 55	120	10	10	95.73

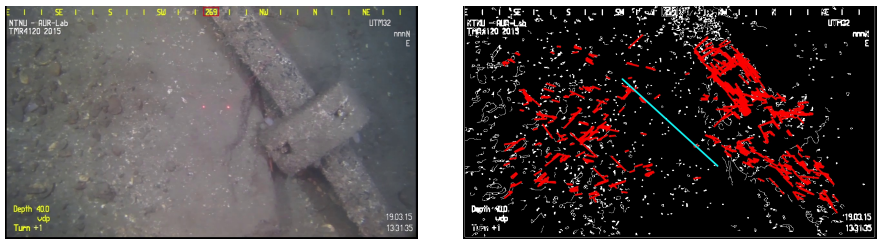
Table 3.3: Tuning probabilistic Hough line transform values.

stricter with what is accepted as edges, however, when looking at figure 3.6 the seabed is far away and no lines are detected. For this case the canny should be more accepting. Finally if one looks at figure 3.7 the camera are very close and almost everything is accepted as lines. To be fair, there where longer sections that it worked rather well, though it struggled when the camera moved closer or further away. This indicates that the system is very dependant on the correct tuning of the canny parameters. An improvement on this could be to do this tuning dynamically, but as the algorithm stands, it is not robust enough to be used on a real operation.



(a) Original (b) Analysed

Figure 3.4: Working on real video.

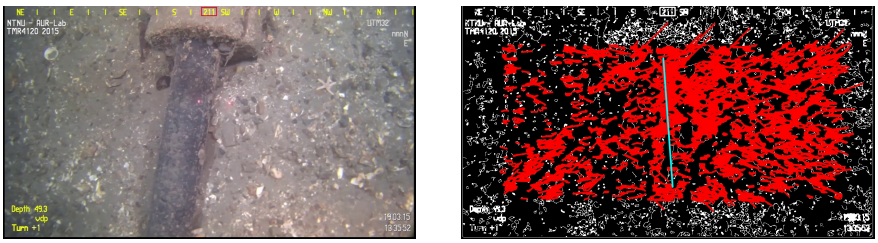


(a) Original (b) Analysed

Figure 3.5: Some outliers.

(a) Original (b) Analysed

Figure 3.6: Camera far away.



(a) Original (b) Analysed

Figure 3.7: Camera very close.

3.2 Machine learning

In the deep learning section there was a small data set of real data that was augmented with differing degrees of simulated data. The mixture is based on percentage of real data present and has been tested all the way from 100% to 0% real data, with steps of 5% between each step. The network was at each step trained for 10 epochs, with a batch size of 32 and with 200 steps per epoch. The goal is to see if any specific blending ratio affect the performance in a significant manner.

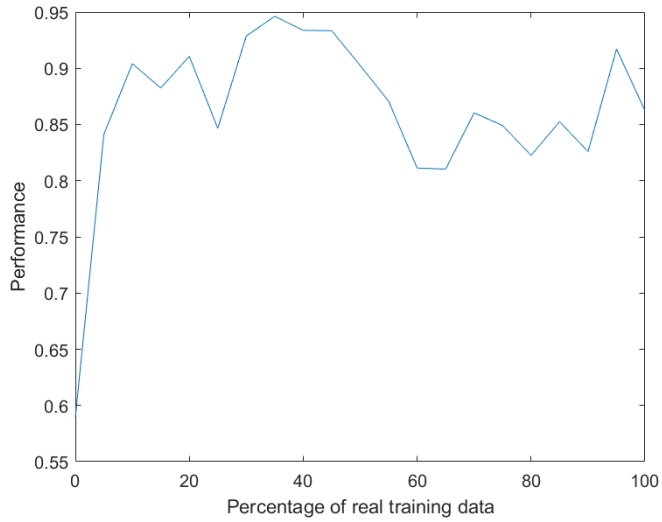


Figure 3.8: Result from different ratios of real and simulated data, for complete results see appendix A.

From figure 3.8 it seems to be a sharp decrease in performance when there are no real data present in the training data, as the performance drops to below 60%. This may be explained as we find that the simulation does not necessarily incorporate the same degree of complexity as the real data. In figure 3.9 it is shown an example from the simulated training data set. When comparing the simulation to the image from the real data in figure 3.10 it becomes apparent that though the simulated data is clearly a

pipe, it lacks elements of the real data, such as biofouling. It is reasonable to assume that the network would be confused if it has not been exposed to such complexities throughout its training process.

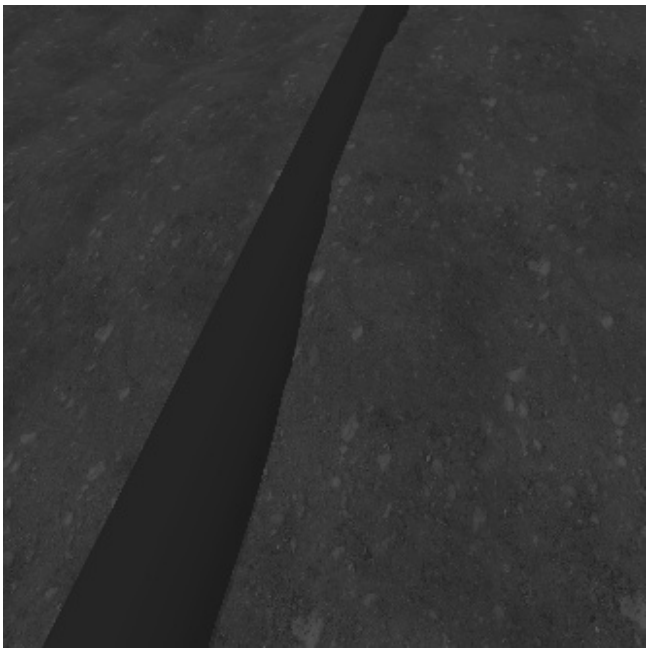
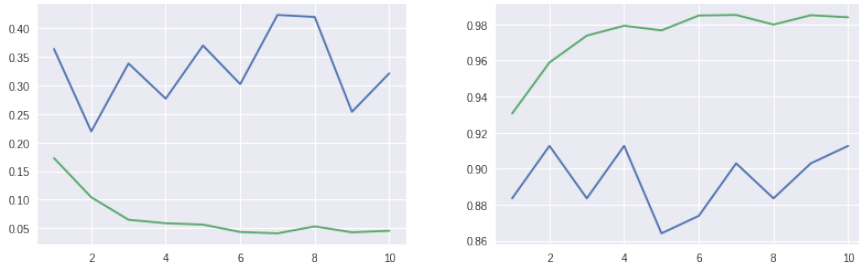


Figure 3.9: Example of simulated data.

When looking at the different ratios in figure 3.8 there is a trend that as we increase the percentage of real data in the data set the performance drops, with the exception of 95%, which seems to be an outlier. This may be explained by the fact that the network has the same number of real images to work with for each of the training instances. The poor performance is simply due to the restricted amount of training data. From figure 3.11 it seems that when training only on the limited real data set the validation loss and accuracy seem to oscillate and the validation loss does not improve, indicating that the network overfit as the data set is too limited in size. The addition of simulated data was originally to combat issues arising from a limited number of

Figure 3.10: Example of real data.

data available. From figure 3.8 it seems that a ratio of 40% real data is optimal as the simulated data may help the system, while not being so abundant that they make the real data negligible as far as the network is concerned. Though it should be mentioned that as there are outliers present it would be interesting to run the experiment over a longer training sequence to see if they are actually outliers or part of a larger trend.



(a) Loss

(b) Accuracy

Figure 3.11: 100% real data.

To investigate the potential outlier at 95% real data, the network was retrained and yielded the following result found in figure 3.12. From this we can see that 95% mix not necessarily is an ideal mix.

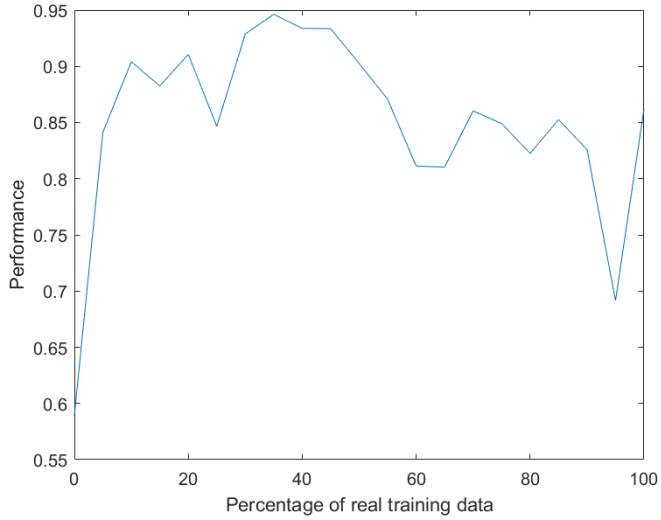


Figure 3.12: Result with retrained 95%.

Chapter 4

Conclusions and future work

The proposed pipe detection scheme based on traditional computer vision techniques performed well in the controlled pool environment, though it lacks the robustness needed to be used in a real ocean environment with high levels of biofouling. Tuning the different parameters showed that the technique is quite sensitive to tuning which may be part of causing its lack of robustness. A natural extension of this work is to add spatial awareness to the algorithm and not only limit it to what angle the pipe have in the image. The ground truth should also be augmented to incorporate this. With the spatial awareness of where the pipe is in the image the system is ready to be integrated into the larger project. The euclidean location of the pipe is currently unknown, but if this system is integrated with the SLAM module it can provide context to the map and thus the euclidean location may be extracted from the map.

The report also looked into machine learning as a viable option to detect the pipe in more complex and realistic environments. The network architecture that have been tested worked well on the complex environment. The augmentation of the data set using simulated data had positive effect. This report limited the problem to look at binary image classification. A natural extension of this work is to extend the technique to detection and even semantic segmentation. To do this it would be necessary to label the data set accordingly as well as to use another network architecture.

Appendix A

Training results

Training with 0% real data

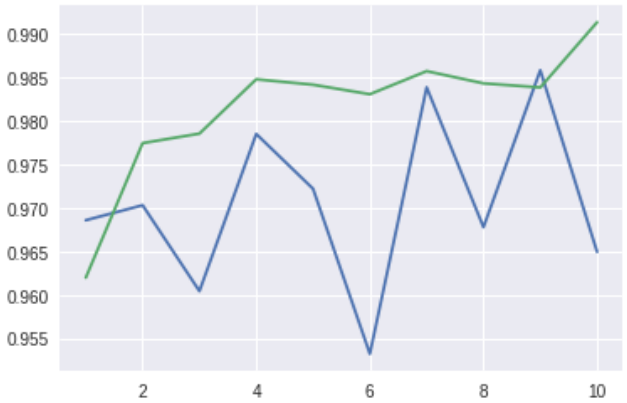
Using TensorFlow backend.

Found 5537 images belonging to 2 classes.
Found 1832 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 13s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 737s 4s/step - loss: 0.0930 - acc: 0.9666 - val_loss: 0.0509 - val_acc: 0.9814
Epoch 2/10
200/200 [==============================] - 711s 4s/step - loss: 0.0737 - acc: 0.9725 - val_loss: 0.0626 - val_acc: 0.9749
Epoch 3/10
200/200 [==============================] - 711s 4s/step - loss: 0.0447 - acc: 0.9845 - val_loss: 0.0538 - val_acc: 0.9797
Epoch 4/10
200/200 [==============================] - 712s 4s/step - loss: 0.0440 - acc: 0.9825 - val_loss: 0.0700 - val_acc: 0.9758
Epoch 5/10
200/200 [==============================] - 711s 4s/step - loss: 0.0283 - acc: 0.9900 - val_loss: 0.0473 - val_acc: 0.9816
Epoch 6/10
200/200 [==============================] - 711s 4s/step - loss: 0.0767 - acc: 0.9731 - val_loss: 0.0447 - val_acc: 0.9854
Epoch 7/10
200/200 [==============================] - 708s 4s/step - loss: 0.0296 - acc: 0.9909 - val_loss: 0.0470 - val_acc: 0.9829
Epoch 8/10
200/200 [==============================] - 712s 4s/step - loss: 0.0218 - acc: 0.9923 - val_loss: 0.0613 - val_acc: 0.9761
Epoch 9/10
200/200 [==============================] - 711s 4s/step - loss: 0.0232 - acc: 0.9906 - val_loss: 0.0541 - val_acc: 0.9845
Epoch 10/10
200/200 [==============================] - 711s 4s/step - loss: 0.0262 - acc: 0.9911 - val_loss: 0.0476 - val_acc: 0.9829
100/100 [==============================] - 246s 2s/step
Final Loss and Score of correct classification: [2.2253368052840234, 0.589375]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])


```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2449s 2s/step
Final Loss and Score of correct classification:  [2.1935397285065954,
0.5929422835633626]
```

Training with 5% real data

Found 5846 images belonging to 2 classes.
Found 1935 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 750s 4s/step - loss: 0.1082 - acc: 0.9620 - val_loss: 0.0839 - val_acc: 0.9686
Epoch 2/10
200/200 [==============================] - 702s 4s/step - loss: 0.0625 - acc: 0.9775 - val_loss: 0.0822 - val_acc: 0.9703
Epoch 3/10
200/200 [==============================] - 703s 4s/step - loss: 0.0644 - acc: 0.9786 - val_loss: 0.1282 - val_acc: 0.9604
Epoch 4/10
200/200 [==============================] - 702s 4s/step - loss: 0.0413 - acc: 0.9848 - val_loss: 0.0548 - val_acc: 0.9785
Epoch 5/10
200/200 [==============================] - 702s 4s/step - loss: 0.0436 - acc: 0.9841 - val_loss: 0.0652 - val_acc: 0.9722
Epoch 6/10
200/200 [==============================] - 703s 4s/step - loss: 0.0415 - acc: 0.9831 - val_loss: 0.1511 - val_acc: 0.9532
Epoch 7/10
200/200 [==============================] - 702s 4s/step - loss: 0.0379 - acc: 0.9857 - val_loss: 0.0449 - val_acc: 0.9839
Epoch 8/10
200/200 [==============================] - 702s 4s/step - loss: 0.0433 - acc: 0.9843 - val_loss: 0.0868 - val_acc: 0.9678
Epoch 9/10
200/200 [==============================] - 703s 4s/step - loss: 0.0443 - acc: 0.9839 - val_loss: 0.0410 - val_acc: 0.9859
Epoch 10/10
200/200 [==============================] - 702s 4s/step - loss: 0.0256 - acc: 0.9913 - val_loss: 0.0892 - val_acc: 0.9649
100/100 [==============================] - 242s 2s/step
Final Loss and Score of correct classification: [0.4094449247419834, 0.84125]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])



Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2404s 2s/step
Final Loss and Score of correct classification: [0.40552627407899533, 0.841844416562108]

Training with 10% real data

Using TensorFlow backend.

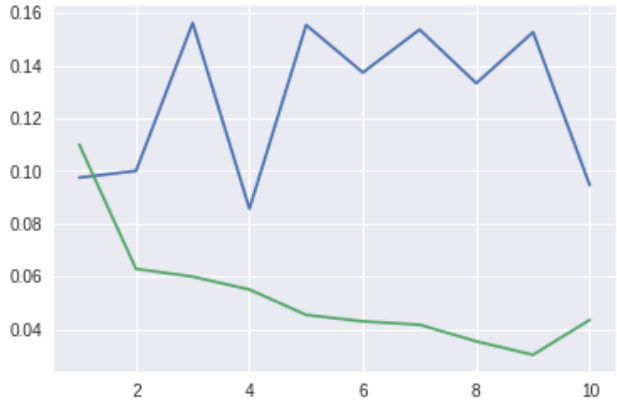
Found 3077 images belonging to 2 classes.
Found 1020 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 10s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 681s 3s/step - loss: 0.1150 - acc: 0.9550 - val_loss: 0.1579 - val_acc: 0.9391
Epoch 2/10
200/200 [==============================] - 656s 3s/step - loss: 0.0668 - acc: 0.9757 - val_loss: 0.0904 - val_acc: 0.9620
Epoch 3/10
200/200 [==============================] - 656s 3s/step - loss: 0.0597 - acc: 0.9791 - val_loss: 0.0890 - val_acc: 0.9680
Epoch 4/10
200/200 [==============================] - 656s 3s/step - loss: 0.0662 - acc: 0.9786 - val_loss: 0.0854 - val_acc: 0.9668
Epoch 5/10
200/200 [==============================] - 656s 3s/step - loss: 0.0550 - acc: 0.9787 - val_loss: 0.0734 - val_acc: 0.9730
Epoch 6/10
200/200 [==============================] - 655s 3s/step - loss: 0.0673 - acc: 0.9755 - val_loss: 0.0833 - val_acc: 0.9630
Epoch 7/10
200/200 [==============================] - 655s 3s/step - loss: 0.0470 - acc: 0.9827 - val_loss: 0.0869 - val_acc: 0.9633
Epoch 8/10
200/200 [==============================] - 655s 3s/step - loss: 0.0490 - acc: 0.9801 - val_loss: 0.0956 - val_acc: 0.9604
Epoch 9/10
200/200 [==============================] - 655s 3s/step - loss: 0.0417 - acc: 0.9845 - val_loss: 0.0771 - val_acc: 0.9702
Epoch 10/10
200/200 [==============================] - 655s 3s/step - loss: 0.0397 - acc: 0.9856 - val_loss: 0.0558 - val_acc: 0.9787
100/100 [==============================] - 227s 2s/step
Final Loss and Score of correct classification: [0.2471233455836773, 0.9040625]
Results

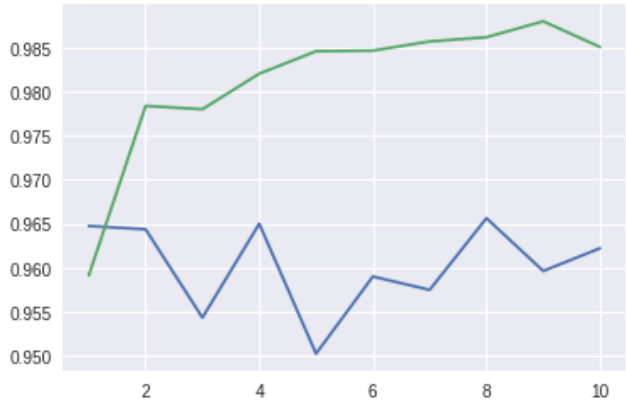
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2259s 2s/step
Final Loss and Score of correct classification: [0.25460785695234367, 0.9021643663739022]

Training with 15% real data

Found 1731 images belonging to 2 classes.
Found 571 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 684s 3s/step - loss: 0.1093 - acc: 0.9597 - val_loss: 0.0975 - val_acc: 0.9647
Epoch 2/10
200/200 [==============================] - 652s 3s/step - loss: 0.0628 - acc: 0.9787 - val_loss: 0.1000 - val_acc: 0.9644
Epoch 3/10
200/200 [==============================] - 654s 3s/step - loss: 0.0623 - acc: 0.9768 - val_loss: 0.1561 - val_acc: 0.9543
Epoch 4/10
200/200 [==============================] - 652s 3s/step - loss: 0.0544 - acc: 0.9823 - val_loss: 0.0857 - val_acc: 0.9650
Epoch 5/10
200/200 [==============================] - 652s 3s/step - loss: 0.0449 - acc: 0.9848 - val_loss: 0.1553 - val_acc: 0.9502
Epoch 6/10
200/200 [==============================] - 653s 3s/step - loss: 0.0426 - acc: 0.9848 - val_loss: 0.1373 - val_acc: 0.9590
Epoch 7/10
200/200 [==============================] - 652s 3s/step - loss: 0.0413 - acc: 0.9859 - val_loss: 0.1536 - val_acc: 0.9575
Epoch 8/10
200/200 [==============================] - 651s 3s/step - loss: 0.0350 - acc: 0.9864 - val_loss: 0.1333 - val_acc: 0.9656
Epoch 9/10
200/200 [==============================] - 653s 3s/step - loss: 0.0309 - acc: 0.9881 - val_loss: 0.1526 - val_acc: 0.9596
Epoch 10/10
200/200 [==============================] - 651s 3s/step - loss: 0.0478 - acc: 0.9838 - val_loss: 0.0946 - val_acc: 0.9622
100/100 [==============================] - 227s 2s/step
Final Loss and Score of correct classification: [0.3413085944205523, 0.8825]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])


0.16
0.14
0.12
0.10
0.08
0.06
0.04
2
4
6
8
10


0.985
0.980
0.975
0.970
0.965
0.960
0.955
0.950
2
4
6
8
10

Training with 20% real data

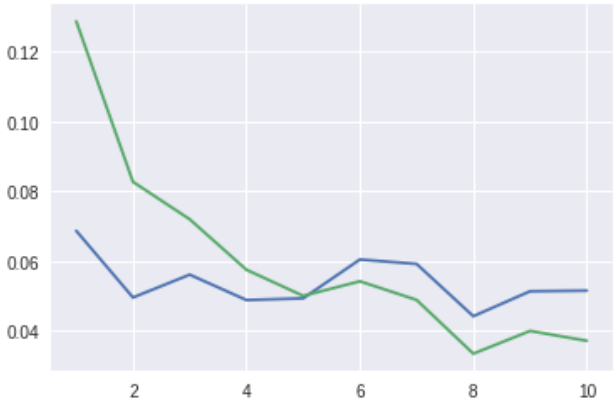
Using TensorFlow backend.

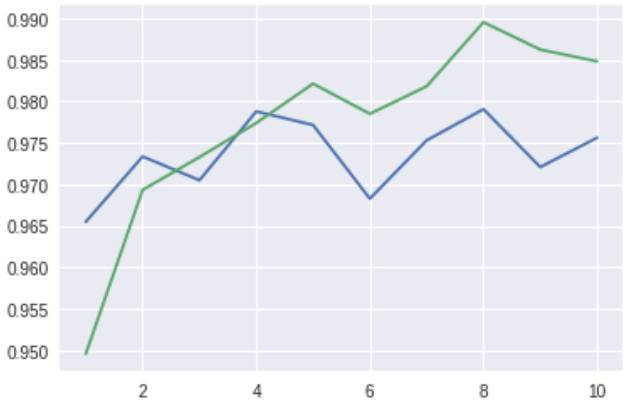
Found 925 images belonging to 2 classes.
Found 308 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 11s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 742s 4s/step - loss: 0.1298 - acc: 0.9495 - val_loss: 0.1998 - val_acc: 0.9156
Epoch 2/10
200/200 [==============================] - 712s 4s/step - loss: 0.0932 - acc: 0.9655 - val_loss: 0.0590 - val_acc: 0.9805
Epoch 3/10
200/200 [==============================] - 712s 4s/step - loss: 0.0686 - acc: 0.9760 - val_loss: 0.0576 - val_acc: 0.9773
Epoch 4/10
200/200 [==============================] - 712s 4s/step - loss: 0.0812 - acc: 0.9699 - val_loss: 0.0651 - val_acc: 0.9708
Epoch 5/10
200/200 [==============================] - 714s 4s/step - loss: 0.0523 - acc: 0.9790 - val_loss: 0.0830 - val_acc: 0.9643
Epoch 6/10
200/200 [==============================] - 720s 4s/step - loss: 0.0505 - acc: 0.9823 - val_loss: 0.1137 - val_acc: 0.9481
Epoch 7/10
200/200 [==============================] - 720s 4s/step - loss: 0.0405 - acc: 0.9862 - val_loss: 0.0581 - val_acc: 0.9773
Epoch 8/10
200/200 [==============================] - 720s 4s/step - loss: 0.0385 - acc: 0.9850 - val_loss: 0.0786 - val_acc: 0.9675
Epoch 9/10
200/200 [==============================] - 720s 4s/step - loss: 0.0426 - acc: 0.9836 - val_loss: 0.0606 - val_acc: 0.9708
Epoch 10/10
200/200 [==============================] - 720s 4s/step - loss: 0.0409 - acc: 0.9850 - val_loss: 0.0699 - val_acc: 0.9643
100/100 [==============================] - 251s 3s/step
Final Loss and Score of correct classification: [0.24989594276994467, 0.910625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])


```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2492s 2s/step
Final Loss and Score of correct classification:  [0.25199398870161854,
0.9135194479297365]
```

Training with 25% real data

```
Found 1233 images belonging to 2 classes.
Found 410 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 748s 4s/step - loss: 0.1287
- acc: 0.9497 - val_loss: 0.0687 - val_acc: 0.9655
Epoch 2/10
200/200 [==============================] - 714s 4s/step - loss: 0.0820
- acc: 0.9697 - val_loss: 0.0496 - val_acc: 0.9734
Epoch 3/10
200/200 [==============================] - 714s 4s/step - loss: 0.0720
- acc: 0.9732 - val_loss: 0.0561 - val_acc: 0.9705
Epoch 4/10
200/200 [==============================] - 715s 4s/step - loss: 0.0577
- acc: 0.9771 - val_loss: 0.0488 - val_acc: 0.9788
Epoch 5/10
200/200 [==============================] - 714s 4s/step - loss: 0.0496
- acc: 0.9823 - val_loss: 0.0493 - val_acc: 0.9772
Epoch 6/10
200/200 [==============================] - 714s 4s/step - loss: 0.0548
- acc: 0.9783 - val_loss: 0.0605 - val_acc: 0.9683
Epoch 7/10
200/200 [==============================] - 715s 4s/step - loss: 0.0488
- acc: 0.9818 - val_loss: 0.0592 - val_acc: 0.9753
Epoch 8/10
200/200 [==============================] - 713s 4s/step - loss: 0.0338
- acc: 0.9894 - val_loss: 0.0442 - val_acc: 0.9791
Epoch 9/10
200/200 [==============================] - 714s 4s/step - loss: 0.0405
- acc: 0.9860 - val_loss: 0.0514 - val_acc: 0.9721
Epoch 10/10
200/200 [==============================] - 715s 4s/step - loss: 0.0379
- acc: 0.9845 - val_loss: 0.0516 - val_acc: 0.9756
100/100 [==============================] - 249s 2s/step
Final Loss and Score of correct classification:  [0.40791284330189226,
0.8465625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```






```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2472s 2s/step
Final Loss and Score of correct classification:  [0.40142193726861025,
0.8491844416562108]
```

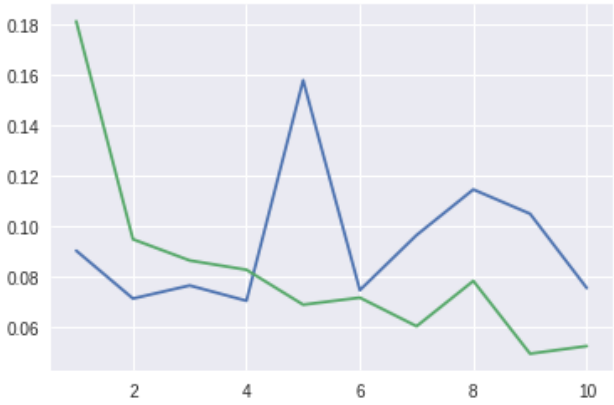
Training with 30% real data

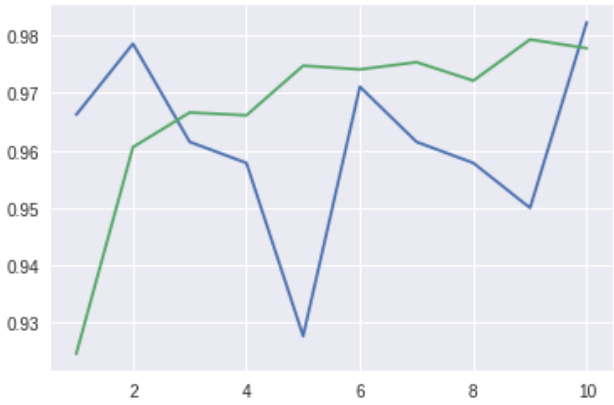
```
Found 809 images belonging to 2 classes.
Found 270 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 746s 4s/step - loss: 0.1466
- acc: 0.9422 - val_loss: 0.0884 - val_acc: 0.9670
Epoch 2/10
200/200 [==============================] - 696s 3s/step - loss: 0.0937
- acc: 0.9659 - val_loss: 0.1222 - val_acc: 0.9547
Epoch 3/10
200/200 [==============================] - 701s 4s/step - loss: 0.0799
- acc: 0.9722 - val_loss: 0.0769 - val_acc: 0.9710
Epoch 4/10
200/200 [==============================] - 705s 4s/step - loss: 0.0634
- acc: 0.9787 - val_loss: 0.0836 - val_acc: 0.9773
Epoch 5/10
200/200 [==============================] - 703s 4s/step - loss: 0.0706
- acc: 0.9739 - val_loss: 0.1044 - val_acc: 0.9624
Epoch 6/10
200/200 [==============================] - 703s 4s/step - loss: 0.0618
- acc: 0.9773 - val_loss: 0.0914 - val_acc: 0.9744
Epoch 7/10
200/200 [==============================] - 705s 4s/step - loss: 0.0485
- acc: 0.9815 - val_loss: 0.0928 - val_acc: 0.9514
Epoch 8/10
200/200 [==============================] - 704s 4s/step - loss: 0.0442
- acc: 0.9836 - val_loss: 0.1105 - val_acc: 0.9340
Epoch 9/10
200/200 [==============================] - 702s 4s/step - loss: 0.0413
- acc: 0.9846 - val_loss: 0.1313 - val_acc: 0.9370
Epoch 10/10
200/200 [==============================] - 705s 4s/step - loss: 0.0402
- acc: 0.9862 - val_loss: 0.0926 - val_acc: 0.9740
100/100 [==============================] - 251s 3s/step
Final Loss and Score of correct classification:  [0.20442844314035027,
0.92875]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```


```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2488s 2s/step
Final Loss and Score of correct classification:  [0.18115035741101515,
0.9335633626097867]
```

Training with 35% real data

```
Found 707 images belonging to 2 classes.
Found 236 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 759s 4s/step - loss: 0.1884
- acc: 0.9243 - val_loss: 0.0902 - val_acc: 0.9662
Epoch 2/10
200/200 [==============================] - 691s 3s/step - loss: 0.0945
- acc: 0.9607 - val_loss: 0.0712 - val_acc: 0.9786
Epoch 3/10
200/200 [==============================] - 692s 3s/step - loss: 0.0943
- acc: 0.9635 - val_loss: 0.0764 - val_acc: 0.9615
Epoch 4/10
200/200 [==============================] - 693s 3s/step - loss: 0.0853
- acc: 0.9643 - val_loss: 0.0703 - val_acc: 0.9578
Epoch 5/10
200/200 [==============================] - 692s 3s/step - loss: 0.0730
- acc: 0.9728 - val_loss: 0.1580 - val_acc: 0.9277
Epoch 6/10
200/200 [==============================] - 691s 3s/step - loss: 0.0742
- acc: 0.9736 - val_loss: 0.0745 - val_acc: 0.9711
Epoch 7/10
200/200 [==============================] - 695s 3s/step - loss: 0.0620
- acc: 0.9747 - val_loss: 0.0964 - val_acc: 0.9615
Epoch 8/10
200/200 [==============================] - 691s 3s/step - loss: 0.0881
- acc: 0.9673 - val_loss: 0.1146 - val_acc: 0.9578
Epoch 9/10
200/200 [==============================] - 693s 3s/step - loss: 0.0516
- acc: 0.9786 - val_loss: 0.1049 - val_acc: 0.9500
Epoch 10/10
200/200 [==============================] - 693s 3s/step - loss: 0.0568
- acc: 0.9756 - val_loss: 0.0754 - val_acc: 0.9823
100/100 [==============================] - 251s 3s/step
Final Loss and Score of correct classification:  [0.1516513626044616,
0.94625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```


0.18
0.16
0.14
0.12
0.10
0.08
0.06
2
4
6
8
10


0.98
0.97
0.96
0.95
0.94
0.93
2
4
6
8
10

Training with 40% real data

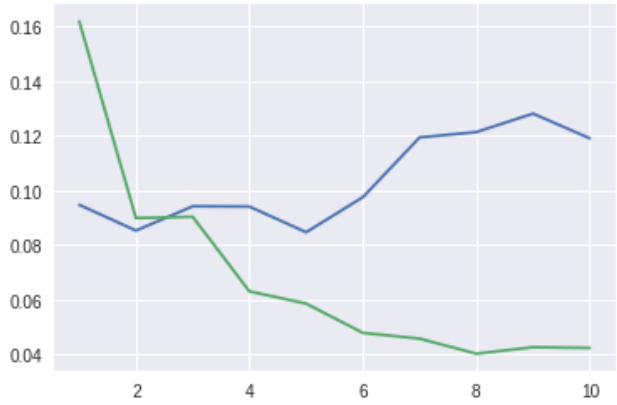
```
Found 543 images belonging to 2 classes.
Found 181 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 28s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 706s 4s/step - loss: 0.1595
- acc: 0.9346 - val_loss: 0.1141 - val_acc: 0.9511
Epoch 2/10
200/200 [==============================] - 679s 3s/step - loss: 0.0939
- acc: 0.9634 - val_loss: 0.1133 - val_acc: 0.9499
Epoch 3/10
200/200 [==============================] - 679s 3s/step - loss: 0.0973
- acc: 0.9646 - val_loss: 0.1378 - val_acc: 0.9389
Epoch 4/10
200/200 [==============================] - 680s 3s/step - loss: 0.0675
- acc: 0.9764 - val_loss: 0.1347 - val_acc: 0.9170
Epoch 5/10
200/200 [==============================] - 679s 3s/step - loss: 0.0534
- acc: 0.9806 - val_loss: 0.1442 - val_acc: 0.9336
Epoch 6/10
200/200 [==============================] - 679s 3s/step - loss: 0.0563
- acc: 0.9773 - val_loss: 0.1130 - val_acc: 0.9562
Epoch 7/10
200/200 [==============================] - 680s 3s/step - loss: 0.0488
- acc: 0.9837 - val_loss: 0.1012 - val_acc: 0.9554
Epoch 8/10
200/200 [==============================] - 679s 3s/step - loss: 0.0399
- acc: 0.9865 - val_loss: 0.1510 - val_acc: 0.9233
Epoch 9/10
200/200 [==============================] - 679s 3s/step - loss: 0.0429
- acc: 0.9820 - val_loss: 0.1259 - val_acc: 0.9671
Epoch 10/10
200/200 [==============================] - 680s 3s/step - loss: 0.0381
- acc: 0.9853 - val_loss: 0.1296 - val_acc: 0.9504
100/100 [==============================] - 238s 2s/step
Final Loss and Score of correct classification:  [0.2117665709927678,
0.93375]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```


```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2370s 2s/step
Final Loss and Score of correct classification:  [0.2083301443621806,
0.9304579673776663]
```

Training with 45% real data

Using TensorFlow backend.

Found 573 images belonging to 2 classes.
Found 231 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 9s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 707s 4s/step - loss: 0.1616 - acc: 0.9344 - val_loss: 0.0947 - val_acc: 0.9576
Epoch 2/10
200/200 [==============================] - 676s 3s/step - loss: 0.0899 - acc: 0.9672 - val_loss: 0.0853 - val_acc: 0.9510
Epoch 3/10
200/200 [==============================] - 676s 3s/step - loss: 0.0904 - acc: 0.9658 - val_loss: 0.0942 - val_acc: 0.9531
Epoch 4/10
200/200 [==============================] - 671s 3s/step - loss: 0.0629 - acc: 0.9768 - val_loss: 0.0941 - val_acc: 0.9565
Epoch 5/10
200/200 [==============================] - 672s 3s/step - loss: 0.0587 - acc: 0.9789 - val_loss: 0.0847 - val_acc: 0.9476
Epoch 6/10
200/200 [==============================] - 670s 3s/step - loss: 0.0478 - acc: 0.9813 - val_loss: 0.0975 - val_acc: 0.9482
Epoch 7/10
200/200 [==============================] - 673s 3s/step - loss: 0.0459 - acc: 0.9832 - val_loss: 0.1193 - val_acc: 0.9393
Epoch 8/10
200/200 [==============================] - 670s 3s/step - loss: 0.0405 - acc: 0.9847 - val_loss: 0.1213 - val_acc: 0.9437
Epoch 9/10
200/200 [==============================] - 672s 3s/step - loss: 0.0427 - acc: 0.9852 - val_loss: 0.1280 - val_acc: 0.9476
Epoch 10/10
200/200 [==============================] - 671s 3s/step - loss: 0.0423 - acc: 0.9846 - val_loss: 0.1190 - val_acc: 0.9527
100/100 [==============================] - 240s 2s/step
Final Loss and Score of correct classification: [0.19655508413910866, 0.9334375]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

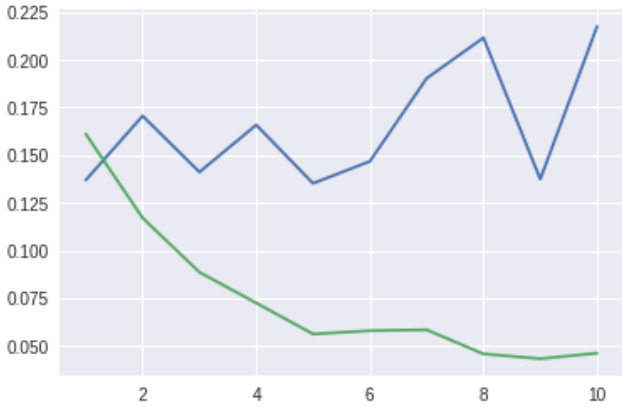



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2381s 2s/step
Final Loss and Score of correct classification:  [0.19593033162829346,
0.9346925972396487]
```

Training with 50% real data

Using TensorFlow backend.

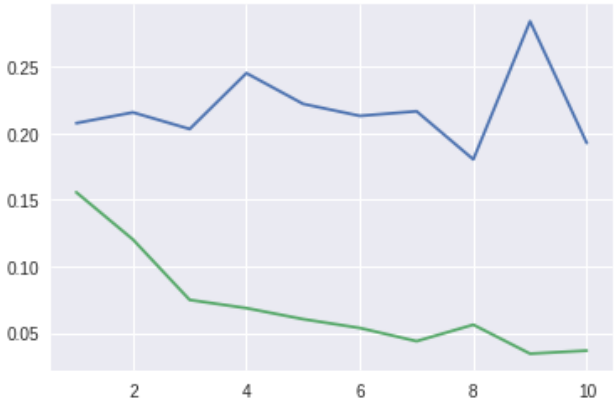
Found 465 images belonging to 2 classes.
Found 155 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 5s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 726s 4s/step - loss: 0.1622 - acc: 0.9347 - val_loss: 0.1369 - val_acc: 0.9355
Epoch 2/10
200/200 [==============================] - 695s 3s/step - loss: 0.1169 - acc: 0.9570 - val_loss: 0.1705 - val_acc: 0.9355
Epoch 3/10
200/200 [==============================] - 694s 3s/step - loss: 0.0882 - acc: 0.9670 - val_loss: 0.1411 - val_acc: 0.9548
Epoch 4/10
200/200 [==============================] - 695s 3s/step - loss: 0.0715 - acc: 0.9723 - val_loss: 0.1658 - val_acc: 0.9290
Epoch 5/10
200/200 [==============================] - 695s 3s/step - loss: 0.0565 - acc: 0.9772 - val_loss: 0.1352 - val_acc: 0.9355
Epoch 6/10
200/200 [==============================] - 694s 3s/step - loss: 0.0577 - acc: 0.9773 - val_loss: 0.1467 - val_acc: 0.9548
Epoch 7/10
200/200 [==============================] - 695s 3s/step - loss: 0.0590 - acc: 0.9799 - val_loss: 0.1902 - val_acc: 0.9355
Epoch 8/10
200/200 [==============================] - 696s 3s/step - loss: 0.0458 - acc: 0.9837 - val_loss: 0.2115 - val_acc: 0.9419
Epoch 9/10
200/200 [==============================] - 695s 3s/step - loss: 0.0433 - acc: 0.9848 - val_loss: 0.1374 - val_acc: 0.9548
Epoch 10/10
200/200 [==============================] - 696s 3s/step - loss: 0.0469 - acc: 0.9831 - val_loss: 0.2175 - val_acc: 0.9290
100/100 [==============================] - 246s 2s/step
Final Loss and Score of correct classification: [0.28214028500020505, 0.9025]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

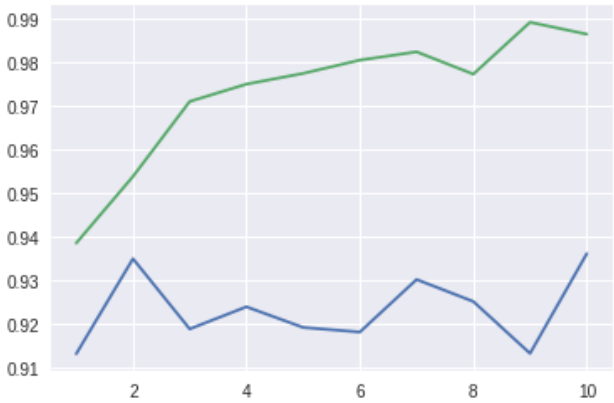



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2445s 2s/step
Final Loss and Score of correct classification:  [0.27086368228700825,
0.9071204516938519]
```

Training with 55% real data

```
Found 436 images belonging to 2 classes.
Found 172 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 719s 4s/step - loss: 0.1559
- acc: 0.9386 - val_loss: 0.2076 - val_acc: 0.9132
Epoch 2/10
200/200 [==============================] - 680s 3s/step - loss: 0.1201
- acc: 0.9535 - val_loss: 0.2157 - val_acc: 0.9350
Epoch 3/10
200/200 [==============================] - 680s 3s/step - loss: 0.0753
- acc: 0.9708 - val_loss: 0.2032 - val_acc: 0.9189
Epoch 4/10
200/200 [==============================] - 681s 3s/step - loss: 0.0696
- acc: 0.9748 - val_loss: 0.2452 - val_acc: 0.9240
Epoch 5/10
200/200 [==============================] - 680s 3s/step - loss: 0.0601
- acc: 0.9773 - val_loss: 0.2220 - val_acc: 0.9192
Epoch 6/10
200/200 [==============================] - 680s 3s/step - loss: 0.0538
- acc: 0.9803 - val_loss: 0.2130 - val_acc: 0.9182
Epoch 7/10
200/200 [==============================] - 681s 3s/step - loss: 0.0439
- acc: 0.9823 - val_loss: 0.2165 - val_acc: 0.9302
Epoch 8/10
200/200 [==============================] - 681s 3s/step - loss: 0.0569
- acc: 0.9768 - val_loss: 0.1803 - val_acc: 0.9252
Epoch 9/10
200/200 [==============================] - 681s 3s/step - loss: 0.0345
- acc: 0.9889 - val_loss: 0.2841 - val_acc: 0.9133
Epoch 10/10
200/200 [==============================] - 683s 3s/step - loss: 0.0367
- acc: 0.9863 - val_loss: 0.1927 - val_acc: 0.9361
100/100 [==============================] - 247s 2s/step
Final Loss and Score of correct classification:  [0.4318155372142792,
0.870625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```



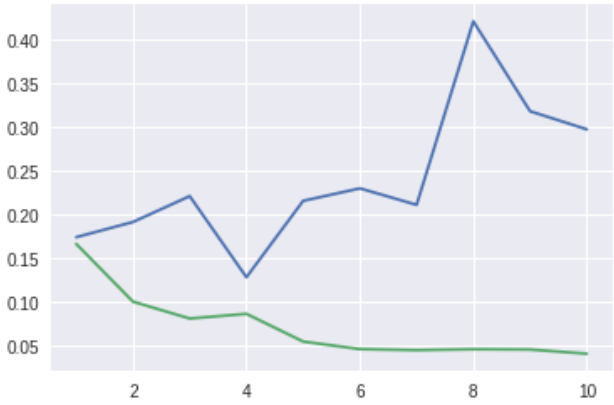



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2452s 2s/step
Final Loss and Score of correct classification:  [0.40599683155448557,
0.8780112923462986]
```

Training with 60% real data

Using TensorFlow backend.

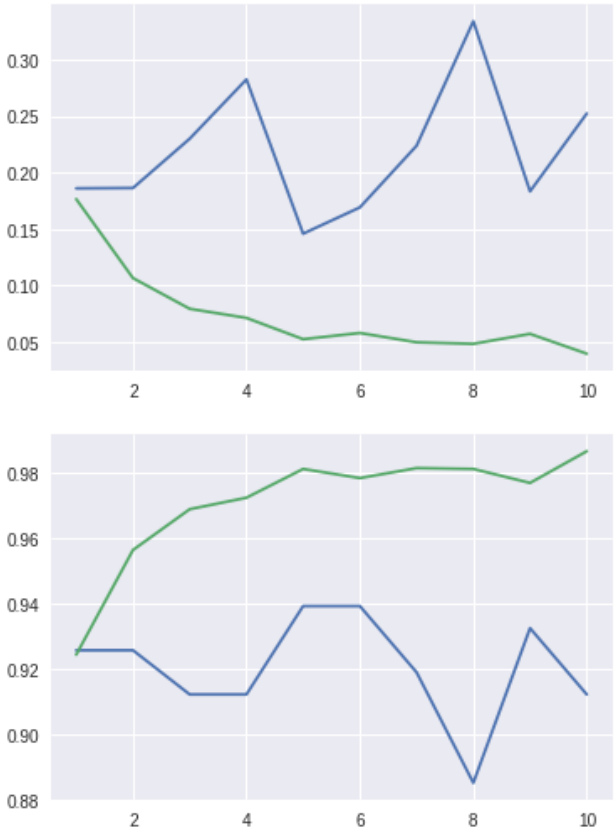
Found 413 images belonging to 2 classes.
Found 159 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 9s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 683s 3s/step - loss: 0.1660 - acc: 0.9377 - val_loss: 0.1739 - val_acc: 0.9245
Epoch 2/10
200/200 [==============================] - 652s 3s/step - loss: 0.1001 - acc: 0.9600 - val_loss: 0.1912 - val_acc: 0.9057
Epoch 3/10
200/200 [==============================] - 653s 3s/step - loss: 0.0805 - acc: 0.9716 - val_loss: 0.2209 - val_acc: 0.9308
Epoch 4/10
200/200 [==============================] - 654s 3s/step - loss: 0.0863 - acc: 0.9676 - val_loss: 0.1280 - val_acc: 0.9371
Epoch 5/10
200/200 [==============================] - 653s 3s/step - loss: 0.0543 - acc: 0.9781 - val_loss: 0.2155 - val_acc: 0.9119
Epoch 6/10
200/200 [==============================] - 653s 3s/step - loss: 0.0458 - acc: 0.9826 - val_loss: 0.2298 - val_acc: 0.9119
Epoch 7/10
200/200 [==============================] - 654s 3s/step - loss: 0.0445 - acc: 0.9830 - val_loss: 0.2109 - val_acc: 0.9182
Epoch 8/10
200/200 [==============================] - 653s 3s/step - loss: 0.0453 - acc: 0.9854 - val_loss: 0.4212 - val_acc: 0.8868
Epoch 9/10
200/200 [==============================] - 653s 3s/step - loss: 0.0450 - acc: 0.9826 - val_loss: 0.3182 - val_acc: 0.9182
Epoch 10/10
200/200 [==============================] - 653s 3s/step - loss: 0.0405 - acc: 0.9847 - val_loss: 0.2974 - val_acc: 0.9182
100/100 [==============================] - 226s 2s/step
Final Loss and Score of correct classification: [0.6711165000498295, 0.81125]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])




```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2245s 2s/step
Final Loss and Score of correct classification:  [0.646444324737618,
0.8169385194479297]
```

Training with 65% real data

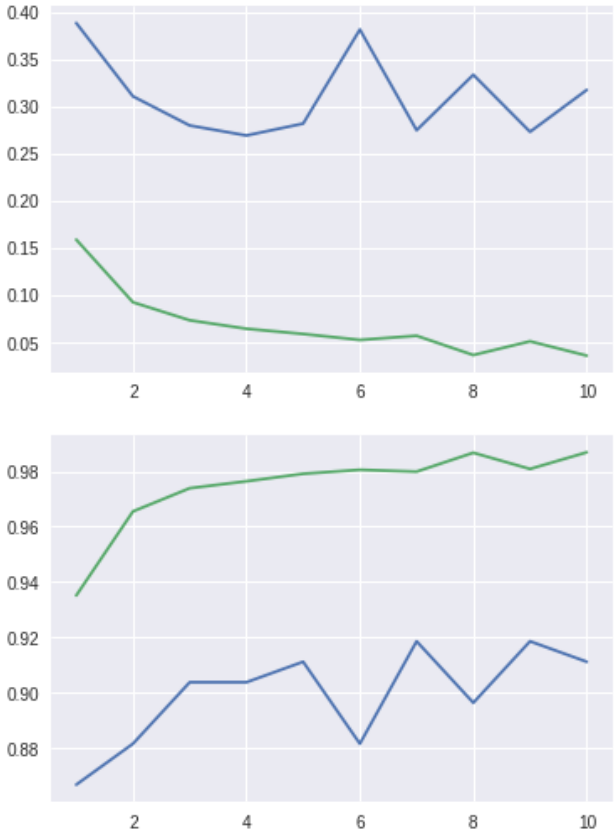
```
Found 426 images belonging to 2 classes.
Found 148 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 658s 3s/step - loss: 0.1778
- acc: 0.9242 - val_loss: 0.1859 - val_acc: 0.9257
Epoch 2/10
200/200 [==============================] - 620s 3s/step - loss: 0.1065
- acc: 0.9564 - val_loss: 0.1862 - val_acc: 0.9257
Epoch 3/10
200/200 [==============================] - 620s 3s/step - loss: 0.0802
- acc: 0.9689 - val_loss: 0.2299 - val_acc: 0.9122
Epoch 4/10
200/200 [==============================] - 619s 3s/step - loss: 0.0733
- acc: 0.9717 - val_loss: 0.2824 - val_acc: 0.9122
Epoch 5/10
200/200 [==============================] - 620s 3s/step - loss: 0.0539
- acc: 0.9803 - val_loss: 0.1457 - val_acc: 0.9392
Epoch 6/10
200/200 [==============================] - 620s 3s/step - loss: 0.0590
- acc: 0.9777 - val_loss: 0.1690 - val_acc: 0.9392
Epoch 7/10
200/200 [==============================] - 619s 3s/step - loss: 0.0519
- acc: 0.9806 - val_loss: 0.2236 - val_acc: 0.9189
Epoch 8/10
200/200 [==============================] - 620s 3s/step - loss: 0.0508
- acc: 0.9803 - val_loss: 0.3338 - val_acc: 0.8851
Epoch 9/10
200/200 [==============================] - 621s 3s/step - loss: 0.0578
- acc: 0.9766 - val_loss: 0.1832 - val_acc: 0.9324
Epoch 10/10
200/200 [==============================] - 621s 3s/step - loss: 0.0423
- acc: 0.9848 - val_loss: 0.2522 - val_acc: 0.9122
100/100 [==============================] - 226s 2s/step
Final Loss and Score of correct classification:  [0.5284045244008303,
0.8103125]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```




```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2248s 2s/step
Final Loss and Score of correct classification:  [0.5181151632600915,
0.8197302383939774]
```

Training with 70% real data

```
Found 442 images belonging to 2 classes.
Found 135 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 670s 3s/step - loss: 0.1584
- acc: 0.9351 - val_loss: 0.3880 - val_acc: 0.8667
Epoch 2/10
200/200 [==============================] - 619s 3s/step - loss: 0.0919
- acc: 0.9656 - val_loss: 0.3103 - val_acc: 0.8815
Epoch 3/10
200/200 [==============================] - 619s 3s/step - loss: 0.0732
- acc: 0.9739 - val_loss: 0.2794 - val_acc: 0.9037
Epoch 4/10
200/200 [==============================] - 619s 3s/step - loss: 0.0643
- acc: 0.9763 - val_loss: 0.2688 - val_acc: 0.9037
Epoch 5/10
200/200 [==============================] - 620s 3s/step - loss: 0.0589
- acc: 0.9789 - val_loss: 0.2814 - val_acc: 0.9111
Epoch 6/10
200/200 [==============================] - 620s 3s/step - loss: 0.0520
- acc: 0.9806 - val_loss: 0.3811 - val_acc: 0.8815
Epoch 7/10
200/200 [==============================] - 619s 3s/step - loss: 0.0571
- acc: 0.9798 - val_loss: 0.2744 - val_acc: 0.9185
Epoch 8/10
200/200 [==============================] - 619s 3s/step - loss: 0.0359
- acc: 0.9868 - val_loss: 0.3331 - val_acc: 0.8963
Epoch 9/10
200/200 [==============================] - 619s 3s/step - loss: 0.0504
- acc: 0.9809 - val_loss: 0.2727 - val_acc: 0.9185
Epoch 10/10
200/200 [==============================] - 619s 3s/step - loss: 0.0356
- acc: 0.9868 - val_loss: 0.3171 - val_acc: 0.9111
100/100 [==============================] - 227s 2s/step
Final Loss and Score of correct classification:  [0.45401140160858633,
0.8603125]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```

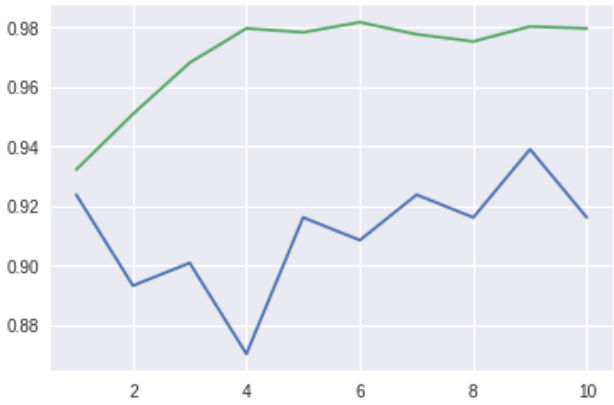



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2251s 2s/step
Final Loss and Score of correct classification:  [0.4726368737182684,
0.8507528230865746]
```

Training with 75% real data

Using TensorFlow backend.

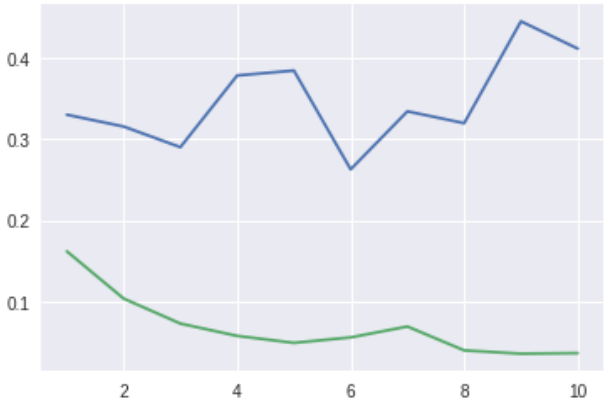
Found 393 images belonging to 2 classes.
Found 131 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 5s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 642s 3s/step - loss: 0.1731 - acc: 0.9318 - val_loss: 0.2170 - val_acc: 0.9237
Epoch 2/10
200/200 [==============================] - 604s 3s/step - loss: 0.1235 - acc: 0.9514 - val_loss: 0.1931 - val_acc: 0.8931
Epoch 3/10
200/200 [==============================] - 616s 3s/step - loss: 0.0938 - acc: 0.9667 - val_loss: 0.3450 - val_acc: 0.9008
Epoch 4/10
200/200 [==============================] - 605s 3s/step - loss: 0.0611 - acc: 0.9798 - val_loss: 0.3618 - val_acc: 0.8702
Epoch 5/10
200/200 [==============================] - 609s 3s/step - loss: 0.0596 - acc: 0.9790 - val_loss: 0.3099 - val_acc: 0.9160
Epoch 6/10
200/200 [==============================] - 603s 3s/step - loss: 0.0495 - acc: 0.9819 - val_loss: 0.2051 - val_acc: 0.9084
Epoch 7/10
200/200 [==============================] - 603s 3s/step - loss: 0.0641 - acc: 0.9768 - val_loss: 0.1651 - val_acc: 0.9237
Epoch 8/10
200/200 [==============================] - 602s 3s/step - loss: 0.0769 - acc: 0.9730 - val_loss: 0.1871 - val_acc: 0.9160
Epoch 9/10
200/200 [==============================] - 604s 3s/step - loss: 0.0539 - acc: 0.9801 - val_loss: 0.2122 - val_acc: 0.9389
Epoch 10/10
200/200 [==============================] - 604s 3s/step - loss: 0.0605 - acc: 0.9782 - val_loss: 0.2047 - val_acc: 0.9160
100/100 [==============================] - 230s 2s/step
Final Loss and Score of correct classification: [0.43964236676692964, 0.8490625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

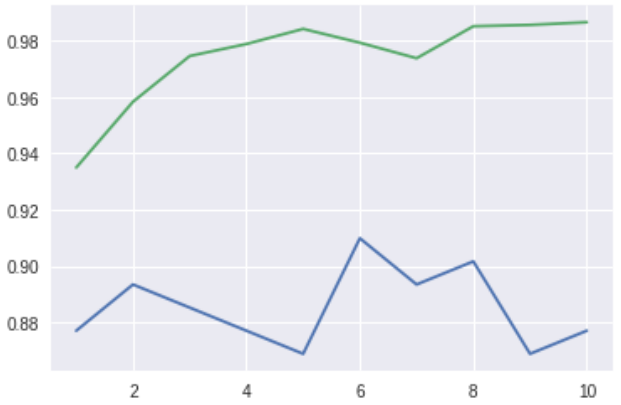



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2286s 2s/step
Final Loss and Score of correct classification:  [0.4354002942335736,
0.8492158092848181]
```

Training with 80% real data

```
Found 349 images belonging to 2 classes.
Found 122 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 693s 3s/step - loss: 0.1618
- acc: 0.9350 - val_loss: 0.3301 - val_acc: 0.8770
Epoch 2/10
200/200 [==============================] - 655s 3s/step - loss: 0.1040
- acc: 0.9581 - val_loss: 0.3155 - val_acc: 0.8934
Epoch 3/10
200/200 [==============================] - 655s 3s/step - loss: 0.0730
- acc: 0.9744 - val_loss: 0.2901 - val_acc: 0.8852
Epoch 4/10
200/200 [==============================] - 655s 3s/step - loss: 0.0579
- acc: 0.9787 - val_loss: 0.3783 - val_acc: 0.8770
Epoch 5/10
200/200 [==============================] - 654s 3s/step - loss: 0.0493
- acc: 0.9840 - val_loss: 0.3844 - val_acc: 0.8689
Epoch 6/10
200/200 [==============================] - 655s 3s/step - loss: 0.0560
- acc: 0.9791 - val_loss: 0.2628 - val_acc: 0.9098
Epoch 7/10
200/200 [==============================] - 655s 3s/step - loss: 0.0692
- acc: 0.9738 - val_loss: 0.3342 - val_acc: 0.8934
Epoch 8/10
200/200 [==============================] - 654s 3s/step - loss: 0.0401
- acc: 0.9850 - val_loss: 0.3196 - val_acc: 0.9016
Epoch 9/10
200/200 [==============================] - 655s 3s/step - loss: 0.0359
- acc: 0.9855 - val_loss: 0.4450 - val_acc: 0.8689
Epoch 10/10
200/200 [==============================] - 655s 3s/step - loss: 0.0366
- acc: 0.9865 - val_loss: 0.4114 - val_acc: 0.8770
100/100 [==============================] - 230s 2s/step
Final Loss and Score of correct classification:  [0.6128289913386107,
0.8225]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```

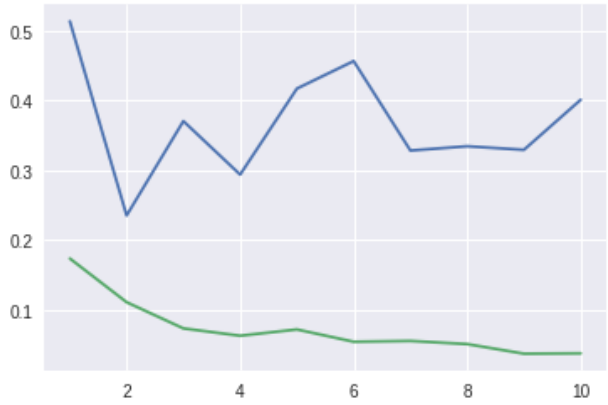


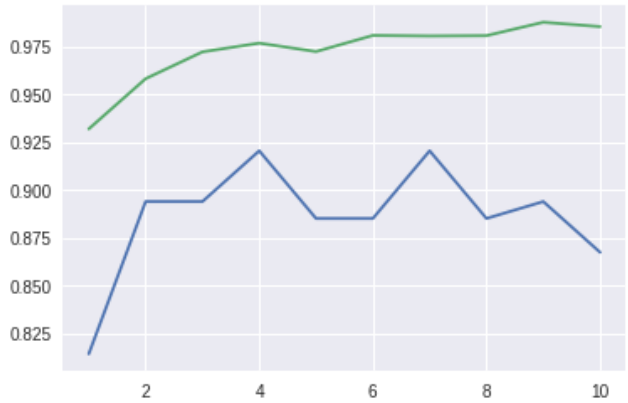



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2289s 2s/step
Final Loss and Score of correct classification:  [0.6231214438328739,
0.8239962358845672]
```

Training with 85% real data

```
Found 337 images belonging to 2 classes.
Found 113 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 680s 3s/step - loss: 0.1710
- acc: 0.9325 - val_loss: 0.5141 - val_acc: 0.8142
Epoch 2/10
200/200 [==============================] - 626s 3s/step - loss: 0.1123
- acc: 0.9576 - val_loss: 0.2352 - val_acc: 0.8938
Epoch 3/10
200/200 [==============================] - 627s 3s/step - loss: 0.0750
- acc: 0.9714 - val_loss: 0.3707 - val_acc: 0.8938
Epoch 4/10
200/200 [==============================] - 626s 3s/step - loss: 0.0621
- acc: 0.9770 - val_loss: 0.2939 - val_acc: 0.9204
Epoch 5/10
200/200 [==============================] - 627s 3s/step - loss: 0.0742
- acc: 0.9714 - val_loss: 0.4175 - val_acc: 0.8850
Epoch 6/10
200/200 [==============================] - 625s 3s/step - loss: 0.0535
- acc: 0.9806 - val_loss: 0.4570 - val_acc: 0.8850
Epoch 7/10
200/200 [==============================] - 627s 3s/step - loss: 0.0562
- acc: 0.9803 - val_loss: 0.3285 - val_acc: 0.9204
Epoch 8/10
200/200 [==============================] - 627s 3s/step - loss: 0.0505
- acc: 0.9806 - val_loss: 0.3346 - val_acc: 0.8850
Epoch 9/10
200/200 [==============================] - 625s 3s/step - loss: 0.0370
- acc: 0.9874 - val_loss: 0.3295 - val_acc: 0.8938
Epoch 10/10
200/200 [==============================] - 627s 3s/step - loss: 0.0376
- acc: 0.9851 - val_loss: 0.4014 - val_acc: 0.8673
100/100 [==============================] - 231s 2s/step
Final Loss and Score of correct classification:  [0.4642712503671646,
0.8525]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```



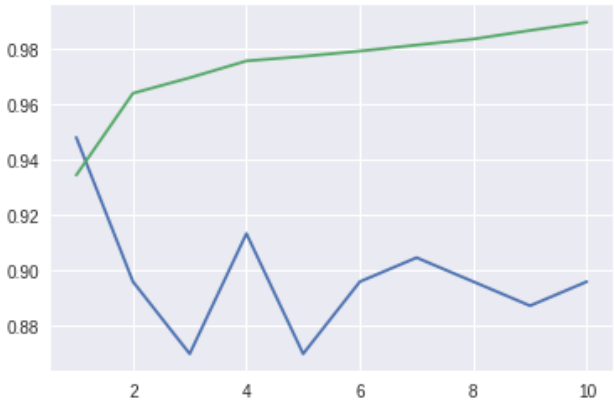



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2291s 2s/step
Final Loss and Score of correct classification:  [0.45364171095023215,
0.8569322459222083]
```

Training with 90% real data

Using TensorFlow backend.

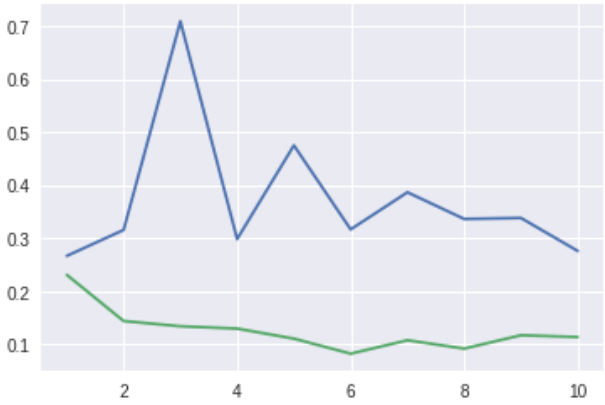
Found 334 images belonging to 2 classes.
Found 115 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 11s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 661s 3s/step - loss: 0.1648 - acc: 0.9335 - val_loss: 0.1658 - val_acc: 0.9478
Epoch 2/10
200/200 [==============================] - 630s 3s/step - loss: 0.0932 - acc: 0.9638 - val_loss: 0.2822 - val_acc: 0.8957
Epoch 3/10
200/200 [==============================] - 630s 3s/step - loss: 0.0813 - acc: 0.9689 - val_loss: 0.4183 - val_acc: 0.8696
Epoch 4/10
200/200 [==============================] - 630s 3s/step - loss: 0.0623 - acc: 0.9755 - val_loss: 0.2500 - val_acc: 0.9130
Epoch 5/10
200/200 [==============================] - 630s 3s/step - loss: 0.0678 - acc: 0.9763 - val_loss: 0.3432 - val_acc: 0.8696
Epoch 6/10
200/200 [==============================] - 629s 3s/step - loss: 0.0531 - acc: 0.9780 - val_loss: 0.2488 - val_acc: 0.8957
Epoch 7/10
200/200 [==============================] - 630s 3s/step - loss: 0.0484 - acc: 0.9816 - val_loss: 0.2452 - val_acc: 0.9043
Epoch 8/10
200/200 [==============================] - 630s 3s/step - loss: 0.0449 - acc: 0.9824 - val_loss: 0.2674 - val_acc: 0.8957
Epoch 9/10
200/200 [==============================] - 630s 3s/step - loss: 0.0409 - acc: 0.9854 - val_loss: 0.2949 - val_acc: 0.8870
Epoch 10/10
200/200 [==============================] - 630s 3s/step - loss: 0.0305 - acc: 0.9894 - val_loss: 0.3098 - val_acc: 0.8957
100/100 [==============================] - 232s 2s/step
Final Loss and Score of correct classification: [0.5278738474845887, 0.8259375]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

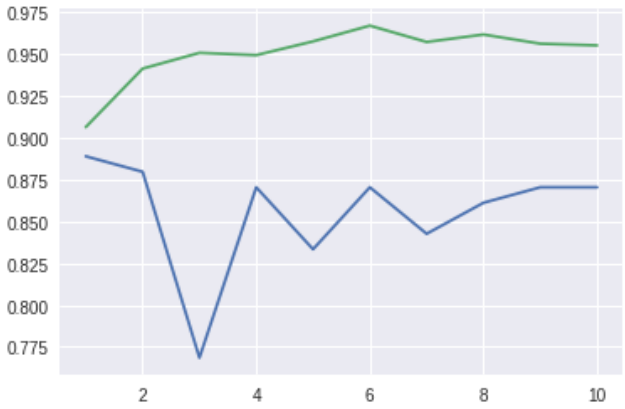



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2304s 2s/step
Final Loss and Score of correct classification:  [0.496252160974673,
0.832465495608532]
```

Training with 95% real data

```
Found 321 images belonging to 2 classes.
Found 108 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 641s 3s/step - loss: 0.2508
- acc: 0.8952 - val_loss: 0.2664 - val_acc: 0.8889
Epoch 2/10
200/200 [==============================] - 603s 3s/step - loss: 0.1541
- acc: 0.9319 - val_loss: 0.3154 - val_acc: 0.8796
Epoch 3/10
200/200 [==============================] - 602s 3s/step - loss: 0.1523
- acc: 0.9502 - val_loss: 0.7087 - val_acc: 0.7685
Epoch 4/10
200/200 [==============================] - 602s 3s/step - loss: 0.1489
- acc: 0.9441 - val_loss: 0.2982 - val_acc: 0.8704
Epoch 5/10
200/200 [==============================] - 602s 3s/step - loss: 0.1305
- acc: 0.9468 - val_loss: 0.4748 - val_acc: 0.8333
Epoch 6/10
200/200 [==============================] - 600s 3s/step - loss: 0.0863
- acc: 0.9700 - val_loss: 0.3164 - val_acc: 0.8704
Epoch 7/10
200/200 [==============================] - 602s 3s/step - loss: 0.1222
- acc: 0.9513 - val_loss: 0.3863 - val_acc: 0.8426
Epoch 8/10
200/200 [==============================] - 602s 3s/step - loss: 0.1087
- acc: 0.9602 - val_loss: 0.3361 - val_acc: 0.8611
Epoch 9/10
200/200 [==============================] - 602s 3s/step - loss: 0.1433
- acc: 0.9455 - val_loss: 0.3379 - val_acc: 0.8704
Epoch 10/10
200/200 [==============================] - 603s 3s/step - loss: 0.1396
- acc: 0.9446 - val_loss: 0.2758 - val_acc: 0.8704
100/100 [==============================] - 232s 2s/step
Final Loss and Score of correct classification:  [0.22877684962004424,
0.9171875]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss',
'acc'])
```



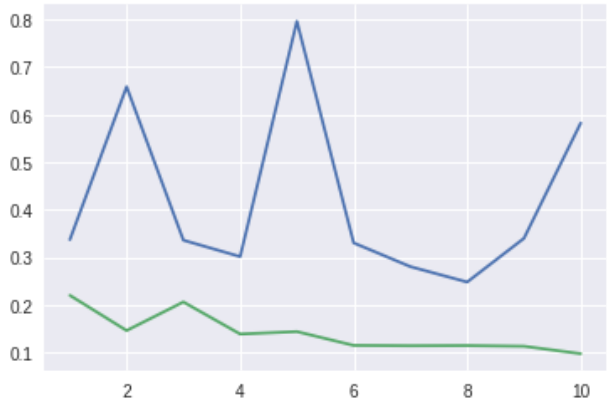



```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2307s 2s/step
Final Loss and Score of correct classification:  [0.2404934093542806,
0.9121392722710163]
```

Training 95 v2

Using TensorFlow backend.

Found 321 images belonging to 2 classes.
Found 108 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Downloading data from
https://github.com/titu1994/Keras-NASNet/releases/download/v1.2/NASNet-large-no-top.h5
343613440/343610240 [==============================] - 4s 0us/step
Training model
Epoch 1/10
200/200 [==============================] - 659s 3s/step - loss: 0.2329 - acc: 0.9057 - val_loss: 0.3369 - val_acc: 0.8611
Epoch 2/10
200/200 [==============================] - 626s 3s/step - loss: 0.1519 - acc: 0.9456 - val_loss: 0.6584 - val_acc: 0.7963
Epoch 3/10
200/200 [==============================] - 626s 3s/step - loss: 0.2425 - acc: 0.8978 - val_loss: 0.3358 - val_acc: 0.8796
Epoch 4/10
200/200 [==============================] - 626s 3s/step - loss: 0.1657 - acc: 0.9256 - val_loss: 0.3013 - val_acc: 0.8889
Epoch 5/10
200/200 [==============================] - 626s 3s/step - loss: 0.2105 - acc: 0.9141 - val_loss: 0.7959 - val_acc: 0.7315
Epoch 6/10
200/200 [==============================] - 624s 3s/step - loss: 0.1599 - acc: 0.9365 - val_loss: 0.3301 - val_acc: 0.8981
Epoch 7/10
200/200 [==============================] - 626s 3s/step - loss: 0.1678 - acc: 0.9332 - val_loss: 0.2802 - val_acc: 0.8981
Epoch 8/10
200/200 [==============================] - 626s 3s/step - loss: 0.1582 - acc: 0.9426 - val_loss: 0.2478 - val_acc: 0.8981
Epoch 9/10
200/200 [==============================] - 626s 3s/step - loss: 0.1413 - acc: 0.9334 - val_loss: 0.3397 - val_acc: 0.8611
Epoch 10/10
200/200 [==============================] - 626s 3s/step - loss: 0.1304 - acc: 0.9463 - val_loss: 0.5820 - val_acc: 0.8148
100/100 [==============================] - 241s 2s/step
Final Loss and Score of correct classification: [0.8226512435078621, 0.6915625]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])



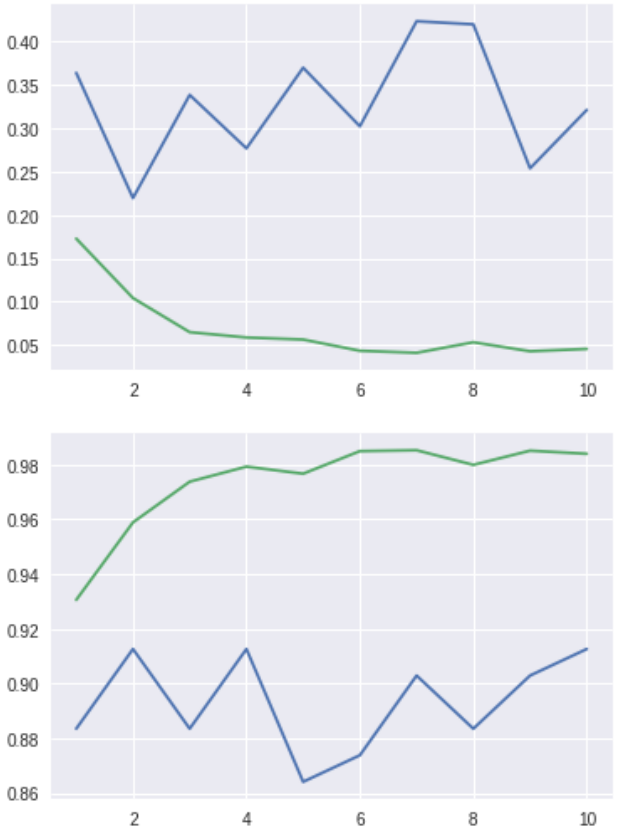

```
Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2395s 2s/step
Final Loss and Score of correct classification:  [0.7989528774107377,
0.6925972396486826]
```

Training with only real data 100%

batch_size = 32
epochs = 10
steps_per_epoch=200
validation_steps=100
test_steps = 100

Found 309 images belonging to 2 classes.
Found 103 images belonging to 2 classes.
Found 6818 images belonging to 2 classes.
Creating model
Training model
Epoch 1/10
200/200 [==============================] - 681s 3s/step - loss: 0.1723 - acc: 0.9310 - val_loss: 0.3639 - val_acc: 0.8835
Epoch 2/10
200/200 [==============================] - 643s 3s/step - loss: 0.1048 - acc: 0.9586 - val_loss: 0.2196 - val_acc: 0.9126
Epoch 3/10
200/200 [==============================] - 643s 3s/step - loss: 0.0655 - acc: 0.9740 - val_loss: 0.3385 - val_acc: 0.8835
Epoch 4/10
200/200 [==============================] - 643s 3s/step - loss: 0.0584 - acc: 0.9794 - val_loss: 0.2769 - val_acc: 0.9126
Epoch 5/10
200/200 [==============================] - 643s 3s/step - loss: 0.0570 - acc: 0.9763 - val_loss: 0.3699 - val_acc: 0.8641
Epoch 6/10
200/200 [==============================] - 643s 3s/step - loss: 0.0436 - acc: 0.9847 - val_loss: 0.3024 - val_acc: 0.8738
Epoch 7/10
200/200 [==============================] - 643s 3s/step - loss: 0.0410 - acc: 0.9855 - val_loss: 0.4235 - val_acc: 0.9029
Epoch 8/10
200/200 [==============================] - 643s 3s/step - loss: 0.0534 - acc: 0.9798 - val_loss: 0.4200 - val_acc: 0.8835
Epoch 9/10
200/200 [==============================] - 643s 3s/step - loss: 0.0436 - acc: 0.9849 - val_loss: 0.2539 - val_acc: 0.9029
Epoch 10/10
200/200 [==============================] - 643s 3s/step - loss: 0.0462 - acc: 0.9838 - val_loss: 0.3211 - val_acc: 0.9126
100/100 [==============================] - 242s 2s/step

Final Loss and Score of correct classification: [0.4408679356426001, 0.8621875]
Results
Availible variables to plot: dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])



Found 6818 images belonging to 2 classes.
100/100 [==============================] - 242s 2s/step
Final Loss and Score of correct classification: [0.40048708651214837, 0.8715625]

Found 6818 images belonging to 2 classes.
100/100 [==============================] - 242s 2s/step
Final Loss and Score of correct classification: [0.43197784580290316, 0.865625]

Found 6818 images belonging to 2 classes.
1000/1000 [==============================] - 2403s 2s/step
Final Loss and Score of correct classification: [0.4318133778586152,
0.8662170639899623]

References

Allevato, A. (2017). Converting between ros images and opencv images (c++). [Online; last edit 2017-04-20].
URL: *http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVIm*

Bradski, G. and Kaehler, A. (2008). *Learning OpenCV*, O'Reilly Media, Inc.

Dattalo, A. (2018). Rosintroduction. [Online; last edit 2018-08-08].
URL: *http://wiki.ros.org/ROS/Introduction*

LeCun, Y., Bengio, Y. and Hinton, G. (2015). Deep learning, *Nature* **521**(1): 436–444.

Lee, D. and Lee, S. (2019). Prediction of partially observed human activity based on pre-trained deep representation, *Pattern Recognition* **85**(NN): 198–206.

Melo, L. and Bott, T. (1997). Biofouling in water systems, *Experimental Thermal and Fluid Science* **14**(1): 375–381.

Zoph, B., Vasudevan, V., Shlens, J. and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition, *CVPR* .