UiT
THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Engineering, Science and Techology
Department of Computer Science and Computational Engineering

Automatic verification of UI tests

—

Christopher Kragebøl Hagerup
Master Thesis in Applied Computer Science

Abstract

The purpose of this thesis is to investigate whether or not it is possible to perform automatic verification of UI Tests using Neural Networks.

The problem being looked at is variance tied to the operating system, graphics card, or other hardware. This can cause false positives during UI testing, and thus we wanted to find a solution that could learn to ignore this, while still verifying the result.

The main technique used was Convolutional Neural Networks, since this task was tied to verifying images of results. The neural network model used was based on the VGG16-model. The models were trained on recognizing 3D-rendered objects in a geological modelling program, with varying translation, rotation and zoom to simulate various different valid UI-test results.

The results part of the thesis takes the form of the classification reports generated after training. In addition to this, the models were verified on an additional image, taken from outside the datasets they were trained on.

With the one model having an accuracy of 89%, and the two others having around 100% accuracy, we concluded that it is possible to perform automatic verification of UI tests with neural networks.

Acknowledgements

I thank my parents for support, both during my studies and this thesis. Furthermore, I would like to thank my dear Charite for being extra supportive during the last few weeks.

My sincere gratitude goes to my supervisors Rune Dalmo (The Arctic University of Norway) and Lucas Provensi (Roxar) for excellent support and feedback while writing the thesis.

I would also like to thank Roxar and Andreas Solem Kjær for suggesting this thesis, in addition to providing access to their software, RMS. Without their support, I simply wouldn't have had enough data to complete the experiments I did.

I am also grateful to my fellow students Daniel Aaron Salwerowicz, Hans Victor Andersson Lindbäck, Kent Arne Larsen and Olav Kjartan Larseng for coming together to create a fantastic classroom environment during the master's degree. They were excellent discussion partners during both the thesis, and the degree as a whole.

Contents

List of Figures

List of Tables

1 Introduction

UI-testing is a type of testing where button presses and mouse clicks/movement are simulated on the user interface of a program. As opposed to unit testing (which tests the individual methods/classes) and integration testing (which tests the application's integration with external frameworks and Application programming interfaces (APIs)), UI testing usually tests the user interface and the program as a whole when subjected to user input.

As of now, most automatic verification of UI-testing is done by reading the internal state of the UI elements. When this is not possible, image comparison is used. However, this may not always be a completely optimal solution, especially in cases where there can be variance in the picture, such as unexpected rotation or translation. Examples are related to platform differences and hardware.

In some cases these variances may be large enough for tests to be unpredictable. For instance, a test that succeeds on one platform could fail on another. Therefore, it could be useful to create a system or tool that can verify UI tests, even when the platforms used have noticable differences.

The question that is asked in this thesis is as follows: Can neural networks be used to verify images from UI Tests? Our hypothesis is that it is possible to use neural networks to verify UI tests, in a way where this variance doesn't have a huge impact.

The next sections will detail some existing tools for UI-testing and verification. Afterwards, these tools will be compared. The remainder of the chapter will further discuss the comparisons between the tools. Finally, the roadmap for the rest of the thesis will be laid out.

1.1 Existing products and technologies

The next sections will look at the current tools present in UI testing.

1.1.1 UI-testing methods

According to Aho et al. (2013), there are two types of UI testing methods: Model-based and Capture-Replay.

A model based UI-testing tool works by extracting the GUI into a model that is easier to interact with (Aho et al., 2013). The most important thing in this case is the model itself, as the tests may fail if the model is not correct.

Meanwhile, a capture-replay tool works by capturing user interaction with the program or application (Aho et al., 2013). A script or captured interaction can then be played back to see if some condition still holds or if the behaviour of the program has changed.

In this thesis, we will focus on Capture-replay-based methods, as all of the existing tools that we looked at is of this type.

1.2 Existing UI Test Products

Below is a selection of popular products and libraries for UI-testing. They have been selected because they are, in our opinion, fairly representative of the market today. In addition, Qt Test and Protractor are the official test tools for Qt and Angular, respectively.

1.2.1 Squish

Squish is a capture-replay testing tool for desktop applications (Froglogic, 2019). Squish works by capturing user input, which is used to create a script that repeats the interaction (Burkimsher et al., 2011). The capture saves a reference to the object being clicked (object name, text, etc.). As a result, the script still works if the location of the object changes, as long as the object itself is still recognizable.

Squish also has support for image-based interactions and conditions. In this case, multiple steps of the replay can be substituted for one or more images. Squish can then click on areas of the screen that matches the image(s). This also extends to the results, where the user can verify if an image matches one or more areas on screen, and fail the test if such an image cannot be found.

1.2.2 Protractor

Protractor is a script-based UI testing tool for AngularJS (Protractor, 2019). It allows the user to write tests in javascript, which interacts with elements on a website (found through either CSS or a model). Tests are usually verified by expecting certain conditions to be true, and checking that this is the case.

Additionally, protractor supports capturing images of the page or application being tested, and supports comparison between these through a plugin called Pix-diff. This plugin is based on Blink-diff (Yahoo, 2016), which is a screenshot-comparison tool developed by Yahoo. See Subsection 1.3.1

1.2.3 Puppeteer

Puppeteer is a test tool maintained by Google (Google, 2019). It is built on a headless Google Chrome-client, which can interact with websites, including single page web applications. This allows the user to perform UI-testing on webpages. Puppeteer works by executing a node-js script, which instructs the headless Chrome client to interact with certain parts of the webpage. This client can also read the properties of elements on the webpage, and use them to verify that the style or functionality of the page is as expected. In addition, puppeteer can also capture the website to both images and pdf-files, enabling visual verification of the result.

1.2.4 Qt Test

Qt Test is a unit-testing framework developed by the Qt Company (The Qt Company, 2019). It is primarily built for unit testing and benchmarking Qt-applications, but features support for performing simple UI-tests. This is done by emulating key- and mouse-events, which are inserted into the Qt event queue. These events are then sent to the widget, which responds to the actions as if they were genuine user input. As with Protractor, verification is usually done with asserts.

Qt also has built-in functionality for capturing images of the UI. However, it possesses no built-in way to verify these results.

1.2.5 Roxar Internal UI Test Tool

Emerson Roxar uses an in-house script-based UI-test tool called PyUITest. This tool is based on Qt Test. It works by running python-scripts. These in turn call methods in the program that retrieves the location of elements on screen. The library can then simulate input on them through Qt Test. This means that there is support for mouse and keyboard interaction. As a whole, PyUiTest can be thought of as an extension of Qt Test with additional methods. It also supports entering and running scripts during runtime as opposed to compile time.

PyUiTest also has support for saving screenshots of the UI, which is done by calling the underlying screenshot methods in Qt.

1.3 Tools for image verification

A subset of verifying UI tests might include testing both computer graphics (3D and 2D) and verifying that 3D-views render correctly. For the purpose of this paper, a 3D-view is a viewport or window on the screen where computer graphics are rendered. One challenge is that the content of such a view is usually handled by a rendering tool, such as OpenGL. This complicates things, since rendered ouput (or 2D-graphics for that matter) cannot be analysed by reading the internal state of the program.

The solution to this problem is to capture and analyse images of the viewport-rendered result, either through comparing with an already existing image, either directly or by hashing. In this section, a few tools for performing image comparison will be looked at.

1.3.1 Blink-diff

Blink-diff is a lightweight image-comparison tool developed by Yahoo (2016). Blink-diff has three different comparison-modes. One is pixel-by-pixel, where the images are compared by raw pixel values alone. The second is by comparing the images perceptually (giving less or no weight to details that look indistinguishable to the human eye). The last method of comparison is contextual, where the tool detects missing or distorted parts of the image, while ignoring smaller differences.

1.3.2 Perceptual Image Diff

Perceptual Image Diff (Yee, 2019) is an image comparison utility that uses Yee (2004)'s perceptual metrics to perceptually compare two images. These metrics are further explained in Subsection 1.4.1.

1.4 Techniques for Image Verification

Section 1.3 detailed various tools that can be used to verify images from UI tests. This section will delve into potential techniques that can be used to verify images from UI tests. Afterwards, the techniques and technologies will be compared in Section 1.5.

1.4.1 Perceptual Metrics

(Yee, 2004) described a process for testing if two images are perceptually different. In this case, perceptually different means that the images are different enough to be spotted by the human eye. This means that even though two images may be very different when using pixel-by-pixel comparison, the differences are negligible to an observer (this could be the result of differences in the Operating System or architecture that something runs on). An example of this is a picture where all the pixels have been shifted slightly to one direction.

In (Yee, 2004)'s example, the images were first converted into an alternate colour space, where the euclidean distance between colours are equal to the perceptual distance between them. He then used the laplacian pyramid of Burt and Adelson (Burt and Adelson, 1983) to compute spatial frequencies in the image and how sensitivity to contrast changes through a Contrast Sensitivity function.

1.4.2 Perceptual Image Hashing

Another way to compare images is through perceptual image hashing. Perceptual image hashing is a process where an image is hashed based its visible features. In theory, this means that two images of different formats should still have the same hash, as long as they are perceptually similar. This property has made Perceptual image hashing popular for digital forensics and copyright enforcement, since some hashing algorithms can detect cases of tampering. Such "attacks" include rescaling/resizing, rotation and translation. Examples of such hashing can be found in (Saikia and Bora, 2014) and (Zhen-Kun et al., 2010).

1.4.3 Neural Networks

Neural networks can also be used to verify the results of tests, but there are a few caveats. First of all, a neural network needs to be trained on a data set. This dataset usually has to be quite large, especially when compared to other methods of image comparison. In addition, the neural network itself is only as good as the data it is trained on. In other words, if there is some bias in the training set, then this bias will also affect the neural network.

A neural network is a machine learning technique where layers of "neurons" try to best map an input to a certain output. The technique has been used for a variety of different uses, including counting (e.g. cells (Xue et al., 2016) and crowds (Yang et al., 2018)) and detecting text in images (Chen et al., 2004).

A neural network would be good in the case where the users would have a good training set, which would include different results. If the variance in this training set is large enough, then the neural network should perform well at classifying and verifying the results. A neural network is also quite flexible, and can with perform a variety of tasks depending on the training it receives.

1.5 Comparison of tools

Table 1.1 compares the previously discussed UI testing tools. Replay means that a script can be generated by recording UI interactions, where 'script' means that the scripts have to be written by the programmer. Compiled is similar, though this means that the tests cannot be changed during runtime. Image is whether the library/tool supports verifying image based results, either by default or through a plug-in. All of the tools supports capturing images, but only the ones with Y supports analysing directly with the tool.

	Script/Re-play	Supported Language(s)	Test/script Language	Image
Squish	Replay/ Script	Multiple[1]	Multiple languages[2]	Y
Protractor	Script	Angular JS	JavaScript	Y[3]
Puppeteer	Script	Html, Javascript[4]	Javascript (Node.js)	N
Qt Test	Compiled	C++ (Qt)	C++	N
PyUiTest	Script	C++	Python	N

Table 1.1: Type and languages supported by each library, as well as whether they natively support image-comparison or interaction

It should also be noted that each of these tools have different use cases. These are shown in Table 1.2, along with the comparison methods that each framework uses.

Framework	**Use case**	**Comparison Method**	**Image comparison**
Squish	UI-testing of desktop-, web- and mobile-based programs and applications	Assertions, Image detection	Direct with error margin
Protractor	UI-testing of Angular-JS based websites and web-applications	Assertions, Image comparison	Direct, Perceptual and Contextual
Puppeteer	UI-testing websites and web-applications	Assertions[5]	N/A
Qt Test	Unit tests and simple UI tests of Qt-based programs	Assertions	N/A
PyUiTest	UI-test of Roxar Emerson's in-house developed software	Assertions	N/A

Table 1.2: Use-cases for each program

1. Squish has multiple editions, supporting C++, Java, C#, and web applications
2. Squish supports scripts in Python, Javascript, Ruby, etc.
3. Protractor has support for capturing images by default, and comparing images through a plugin
4. Puppeteer is primarily designed for testing webpages, and thus works on html, Javascript and CSS

As seen in Table 1.2, each tool is specialized for it's own group of software.

The next step is to look at the different techniques for comparing or verifying images.

	Comparison	Required images	Notes
Direct Comparison	Comparing pixel-by-pixel	2	Compares the images based on pixel values. Will detect differences not visible to the human eye, including noise
Perceptual Image Diff	Perceptual	2	Only does perceptual comparison
Image Hashing	Perceptual Hash	2[6]	Effectiveness depends on hash used
Neural Networks	Convolution/ Categorization[7]	1[8]	Very dependent on training set. With the correct training, this could better handle variation within the image

Table 1.3: Comparison between the different image comparison/classification tools.

1.6 Discussion

Machine learning is a compelling technology for UI testing, because of its flexibility and tolerance for variance. However, none of the frameworks looked at has made use of this technology. Therefore, in this thesis, we intend to

5. Puppeteer doesn't have any way to verify test results on its own, but node.js already supports this out of the box
6. Hash can be stored, reducing the need for storage space
7. A neural network trained for this task would classify the images based on training, instead of comparing to another image
8. Requires prior training with several images, rather than a single other image for comparison

investigate the possibility of improving UI testing by applying machine learning techniques.

One example would be to employ a neural network to compare or analyse pictures. By doing this, we hope that the tools and frameworks could be improved in e.g. adaptability. The source of improvement comes from the fact that a neural network doesn't necessarily compare images, but rather analyses and categorizes them based on earlier training. An example could be when an image from a test has evidence of failure, but is sufficiently similar to the expected image. However, whether or not machine learning is an improvement in such a situation is left as future work.

1.7 Neural Network Techniques

Overall, we think a convolutional neural network would be a good fit for using machine learning in this context. This is because verifying these results is based around image recognition, which is the field that these networks are based around. In addition, the convolutions allow the neural network to look for details, and to find correlations between features in the picture and the classifications.

A convolutional neural network is a neural network that applies one or more image transformations to extract features from an image. Examples of this includes edge detection, sharpening up the image, and so on. In addition to the convolution, an convolution-layer also contains an activator function and pooling, which scales down an image based on a filter (e.g. taking the pixel with the highest value, or averaging the pixels). See Section 2.4 for more details about these kinds of networks.

Below is a list of Neural Network technologies that could be used as inspiration during the thesis itself. They have primarily been chosen because they are quite new, and that they seem relevant for the purpose of achieving better performance with the neural network itself.

1.7.1 Mixed-scale Dense Convolutional Neural Networks

(Pelt and Sethian, 2018) looked into a new approach to Convolutional Neural Networks. This approach uses dilated convolutions and dense connections. The results of these two techniques was called the MS-D architecture.

(Pelt and Sethian, 2018) explains dilated convolutions as follows:

> A dilated convolution $D_{h,s}$ with dilation $s \in \mathbb{Z}^+$ uses a dilated filter h that is nonzero only at distances that are a multiple of s pixels from the center.

According to the authors, this enables the networks to capture additional features compared to the traditional scaling assumed by regular CNNs. The information at one scale could then affect choices made at other scales without having to pass through additional layers. The benefits thus enables smaller networks that are more trainable.

Meanwhile, the mixed-scale approach means that the output of each of the Convolutional layers has the same size or form as the output of the final layer. This enables us to not only use the result of the final layer as output, but every layer. Pelt and Sethian called this "densely connecting a network" (Pelt and Sethian, 2018).

However, this method was ultimately not used. This is because the dilated convolutions output an image that is just as large as its input. Usually when we want to classify a picture, we use convolutions and pooling to scale down the image. This is primarily because every pixel of the image is given as an input to every neuron. Thus, when we have a large image, we need many weights, which leads to the model becoming unnecessarily large.

1.7.2 Adversarial Networks

For a neural network, the training set is very important. If the training set is somehow biased or not covering all possible inputs, then there is a chance that the neural network will not give a desirable output, or recognize a completely different object or variable all-together. Thus, some degree of adversarial networks could be useful, depending on the training data.

In 2018, (Moore et al., 2018) looked into the use of adversarial networks to identify and remove bias from a training set. An adversarial neural network is based on the idea of trying to optimize two loss functions simultaneously. One of the two functions is a generator, which creates data. The other is a discriminator, which seeks to predict the data created by the first neural network. In the case of (Moore et al., 2018) however, they based their solution on a technique called "adversarial optimization". With this type of approach, the "generator" part tries to create a result that doesn't predict a nuisance variable, which is a variable that should not be correlated with the output. The second network tries to predict the nuisance variable.

This kind of bias removal was not used due to time constraints.

1.8 Conclusion

In this chapter, a few different tools for UI testing were looked at. However, when they were compared, it was discovered that none of tools were using any functionality related to machine learning. As a result, convolutional neural networks were looked at, and it was decided that these would be the best fit, due to the visual nature of an UI or viewport.

The next steps are as follows: First, images are to be collected for a training set for our neural network. Afterwards, this set will be used to create a simple test, like detection of a background colour. Afterwards, we will move on to more complex tasks, like detection of objects within a viewport.

2 Methods

This chapter will detail the methods that were used in the thesis. Some of the explainations here are based on lectures by Prof. Bernt Bremdal (Bremdal, 2018).

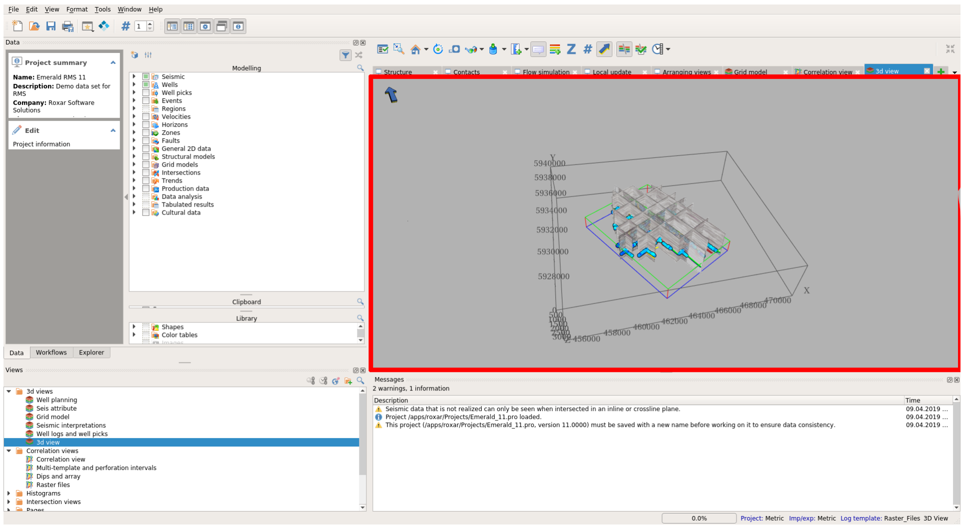
2.1 Software and Tools used

For the machine learning parts of the thesis, Keras (Chollet et al., 2015) and Tensorflow (Tensorflow, 2019) was used together with Python (Python Software Foundation, 2019). Tensorflow is an open-source machine-learning platform that was originally developed by Google for internal use. The library has stable APIs for both C and Python. It also supports running on both CPUs and GPUs.

Keras is a high-level API for neural networks, written in Python, and is capable of running on top of TensorFlow. Keras was created with fast experimentation in mind, and models can quickly be modified without extensive work. It also has functionality for loading training-images directly from disk into the neural networks, which is helpful when working on large image sets.

The training images were generated by running tests in Roxar's software, Reservoir Management Software (RMS) (Roxar Software Solutions, 2019), using their in-house test tool. Elements in the viewport of RMS (shown in

Figure 2.1) were manipulated either through the testing tool, or manually by an user. Afterwards, the test tool saved an image of the viewport. This was repeated to build up a set of images that contained different objects at various angles and scales. This was later given as input to our neural network.



Name: Emerald RMS 11
Description: Demo data set for RMS
Company: Roxar Software Solutions

2 warnings, 1 information

Description	Time
Seismic data that is not realized can only be seen when intersected in an inline or crossline plane.	09.04.2019 ...
Project /apps/roxar/Projects/Emerald_11.pro loaded.	09.04.2019 ...
This project (/apps/roxar/Projects/Emerald_11.pro, version 11.0000) must be saved with a new name before working on it to ensure data consistency.	09.04.2019 ...

0.0% Project: Metric Imp/exp: Metric Log template: Raster_Files 3D View

Figure 2.1: Image of RMS (Roxar Software Solutions, 2019), which was used to produce our test images. The red square contains the viewport, which is the primary source of data.

2.2 Activation Functions

An activation function is the definition of an output for a node in a neural network, when given an input or a set of inputs. An activation function is usually accompanied by a set of trainable weights, which represents the learning of the node. Whenever a result produced by a neural network doesn't match the expected result, the difference is then taken and propagated back through the network, adjusting the weights of the neuron. This causes the neural network to learn.

2.2.1 Rectifiers

A ReLU (rectified linear unit) is an unit that employs a rectifier. The rectifier is an activation function that returns the positive part of its input. This is useful for modeling computer vision, since it is closer to how biological neurons work compared to other activation functions (Glorot et al., 2011). As an example, if a convolution detects edges where 0 means no edge, having a negative value wouldn't make sense. The formula for a rectifier is as follows:

$$\text{rectifier}(x) = \max(0, x) \tag{2.1}$$

There are different subcategories of ReLUs. One of them is the leaky ReLU, which gives a very small value instead of 0 when below the threshold as seen in Equation 2.2. This is done to alleviate the "Dying ReLU problem", where the weights of a node are never changed. This happens because the weight of the node is only adjusted when it is activated in the neural network (Maas et al., 2013). However, with the leaky ReLU, the weights are updated even when the input is below 0.

$$\text{Leaky rectifier}(x) = \max(0.01x, x) \tag{2.2}$$

2.2.2 Softmax

At the end of the neural network, we use a densely connected layer with the softmax activation function to provide classifications. This activation receives real numbers from the preceding layer, and uses it to produce K probabilities that sum up to 1. In our classification task, K is the number of classes that we have. In other words, each output of the neural network is the probability of the input being each class (Habibi Aghdam, 2017, p. 49-50).

2.3 Dropout

Dropout is the act of randomly ignoring certain units or neurons during training. This means that the ignored neurons will not be adjusted, and will not learn during that generation. The reason for using dropout is very simple: to prevent overfitting (Srivastava et al., 2014).

Overfitting is when neurons start to develop co-dependencies during training, which results in the neurons only adapting to best recognize the training data. This leads to the network becoming less accurate on the test data, and in a real-world environment afterwards. Dropout helps the network develop redundancy, thus reducing these co-dependencies.

In convolutional neural networks, dropout is usually employed in the dense (i.e. feed-forward) parts that compute the output value (e.g. classification).

2.4 Convolutional Neural Networks

The first network used was a Convolutional Neural Network. This is a type of neural network where convolutions are applied to the input (Habibi Aghdam, 2017, p. 85-95). This allows the neural network to work with a simplified or downscaled version of the image. A convolutional neural network usually features multiple convolutions, with pooling to downscale the input.

2.4.1 Convolution

A convolution is to apply a filter to the matrix representing the image (Habibi Aghdam, 2017, p. 85-95). This filter is usually represented by a 3-dimensional matrix, where the third dimension is equal to the amount of channels in the image. Each element of the filter and corresponding section of the image matrix are multiplied piecewise, before they are summed up. The sum of a convolution is a single scalar. These scalars are inserted into the corresponding spot on a new matrix, called a feature map. The purpose of convolutions is to simplify the image so that it's easier for a dense layer to work on it.

Convolutions work in strides, which are offsets at which the convolution is applied. Sometimes, the strides may be two or higher. This is done when we want to convolute alternate pixels. In addition to the choice of strides, sometimes the image is padded out. This may be done in order to preserve the size of the image (i.e. prevent downscaling or to apply a larger filter). In the case of this thesis, single strides were used, and the image was padded out with zeroes to preserve the size.

In our experiments, we chose a kernel size of 3×3. This is in part due to tradition within the field. Another reason is that our network architecture has a lot of trainable parameters. A 5×5 kernel would increase this amount to a point where the GPU of the machine we used would run out of memory.

2.4.2 Convolutions and Activation functions

In the case of a convolutional layer, each cell in the kernel is an activation function (Section 2.2). As a result, the kernel itself learns with the rest of the network. In our case, Leaky ReLUs (Subsection 2.2.1) were used as the activation function for the convolutional layers.

2.4.3 Pooling

Pooling may also be used to further simplify the data. Pooling is a technique where the matrix is downscaled according to a method. This works similarly to a convolution that you have a kernel that is applied to the image matrix. Unlike convolutions, pooling does not have any learning or gradients (Habibi Aghdam, 2017, p. 95-98). Three common types of pooling are min-pooling, max-pooling and average-pooling.

- Min-pooling is a pooling technique where the smallest value is chosen.
- Average pooling takes the average of all elements in the kernel.
- Max pooling takes the highest value of the kernel.

In the case of our network, we used max pooling. This is because we want to detect the most pronounced features.

2.5 Choice of optimizer

Neural networks are mostly trained with an optimization algorithm. These algorithms compute gradients of the network during training. These are in turn used to update the weights that our network uses.

In keras, we used Adaptive Movement Estimation (Adam) (Kingma and Ba, 2014) as the optimization algorithm for our network. The reason why we chose Adam over other optimizers is due to it's low memory usage, in addition to its robustness.

2.6 Measures to prevent exploding gradients

During testing, the network started suffering from exploding gradients. This is a problem where the gradients grow exponentially as they are sent back though the network to update the weights (Philipp et al., 2017). In the case of the network that we used, this lead to the following:

1. Slower learning
2. The network would sometimes not convergence to any learning, to a point where the neural network could get "stuck" on the starting validation

accuracy.

3. The gradients would occasionally reach NaN, which would set the weights of the neural network to NaN.

Of the three symptoms, the third one was the most problematic. After the weights reached NaN, the network would just give the same classification regardless of the input. The subsections below detail the steps that were taken in order to identify and reduce the effects that the exploding gradients had on the model used.

2.6.1 Testing on CIFAR-10 dataset

After the exploding gradients were first encountered, the neural network was tested on the CIFAR-10 image set (Krizhevsky, 2009). This was done in order to make sure that the issue was not related to the training set we used.

CIFAR-10 is a dataset consisting of 60000 tiny images (Krizhevsky, 2009). These are grouped into 10 classes, with 6000 images per class. Some of the classes include "airplane, "automobile", "bird" and "dog". It has also, among other things, been used to train neural networks for image completion (Swofford, 2018), and image classification for embedded systems (Calik and Demirci, 2018).

There were a couple of reasons why the CIFAR-10 dataset was used for this troubleshooting. First, the dataset itself is much larger than the image set collected from RMS. This eliminated any possibility that the issues originated from a lack of data. Second, there is much greater variance within CIFAR-10. This would identify that the problem was not due to invariance. Finally, CIFAR-10 is already scaled to a resolution of 64×64, which eliminates any issues with the scaling of our input.

2.6.2 Gradient Clipping

Gradient clipping was the first measure used to address exploding gradients in our model. Gradient clipping is done by scaling the gradient down by a certain amount if it exceeds a threshold. Gradient clipping has been utilized both in Convolutional (Zhang et al., 2018a) and recurrent (Aharoni et al., 2017) neural networks.

In our case, we set clipnorm = 1., which means that the gradients are clipped to a maximum norm of 1.

2.6.3 L2-normalization

The final step that we used to solve exploding gradients were to use L2 normalization. This technique projects all the input values of the layer into the same range (Wu and Li, 2018). After this method was applied, exploding gradients were no longer observed.

2.7 Learning Rate

The learning rate controls how quickly the neural network learns during training. In order to reduce overfitting, we reduced the learning rate from 0.01 (keras default) to 0.00001.

2.8 Implementation

The model of our neural network is based on VGG16 (Simonyan and Zisserman, 2014), which is a deep convolutional model that uses multiple convolutions with small filters (3×3) between each pooling layer. This is opposed to other convolutional models like AlexNet (Krizhevsky et al., 2017), which instead relies on fewer convolutions with larger kernels. Our VGG16-based model was used due to a couple of different reasons. First, the thesis author was already experienced with a similiar model. Second, its simplicity makes it a compelling tool for prototyping.

We make use of 5 "blocks" where each block has two convolutional layers. At the end of each block (with the exception of the last one), we use pooling to scale down the image. Furthermore, we make use of batch normalization in between the convolutional layers. After the convolutional part of the network, the image is flattened to a 1D-array and sent into two densely connected layers. These output to a softmax layer, which provides the classification.

We used 214×214 images as the input due to the features of the images in question. These include thin lines and other features which may disappear when the image is sufficiently downscaled. The choice of this input shape lead us to use 5 blocks.

The model of the network is shown in Figure 2.2. Here, the topology of the network is shown at the top, while the bottom shows the layout of each block. The filters shown are based on each layer's interpretation of the input image.

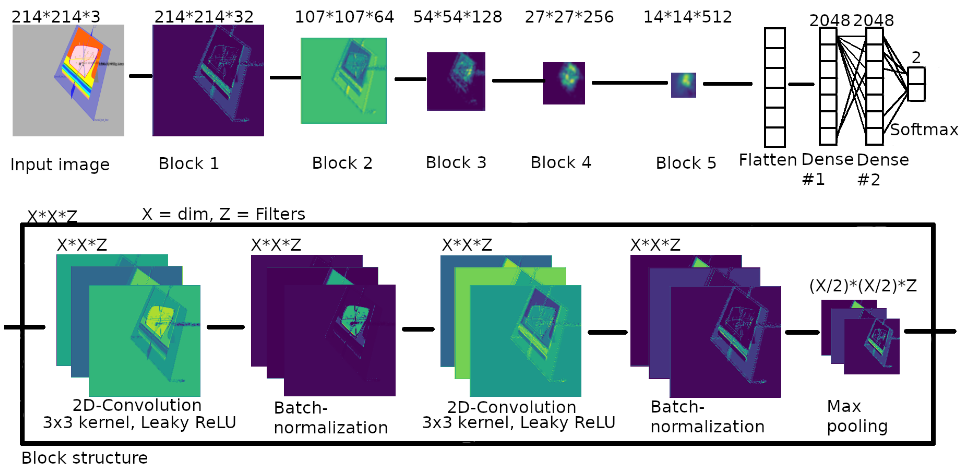


Figure 2.2: Illustration of the model used. The numbers above the images are the dimensions of each block, while the numbers above the dense layers show the amount of neurons.

2.9 Measurement

During training, performance of the neural networks in question is measured through the metrics listed in Table Table 2.1.

Metric	**Explaination**
Accuracy	Amount of correct predictions
Loss	Cumulative loss for current epoch
Precision	For each class, how many of the picked elements were correct
Recall	How many of the relevant objects of a class were picked
F1-Score	The harmonic average between precision and recall
Epoch time	How long it takes to run one epoch of training. An epoch consists of doing one iteration on the training set, and then measuring accuracy on the validation set

Table 2.1: The different metrics which will be used for classifying the performance of our networks

2.10 Procedure

The objective of this thesis was to investigate whether or not verification of UI testing was possible with convolutional neural networks. One way to create a proof of concept for this would be to check if the convolutional neural network was able to detect certain features in a viewport. First, a simple scenario was tried, which was to detect whether the background was a single colour. After this a more complex task was attempted, which was to detect objects wihin the viewport.

2.10.1 Recognizing background colors

The first test consisted of trying to categorize the background color of a 2D view. For this purpose, two categories were used: "White" and "Not white". The training data was generated by running a test where the background was set to a random color. The 2D view was occationally also manually changed by manipulating the objects inside. This allowed us to create many different test pictures quickly, but at additional cost: the training set contained duplicate pictures with the same color, and there were considerably fewer white than non-white pictures.

2.10.2 Exploding gradients

We encountered a problem when we first started training our neural network on the data set: The network was not learning anything. This behaviour only occured when using leaky ReLUs in the densely connected layers at the end of our model. Changing these activations to a tanh- or sigmoid-shaped function temporarily solved this issue. Our theory was that this was because the ReLU is linear (i.e. does not converge to a number) as shown in Figure 2.3.

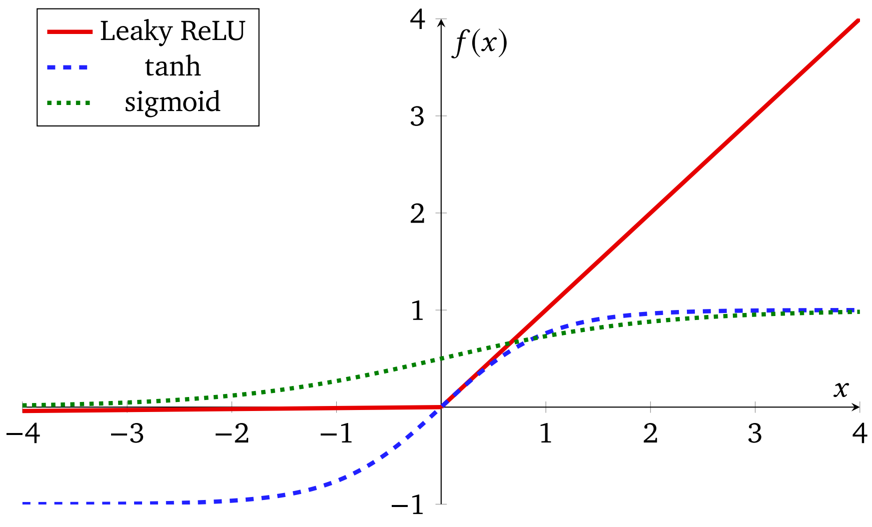


Figure 2.3: Plot of the leaky ReLU, tanh and sigmoid functions for $x \in (-4, 4)$

However, we still had no definite answer as to why these layers acted this way. The activations used in these layers were leaky ReLUs, which means that the "Dying ReLU"-problem should not have occured. This is because even negative values would still adjust the gradient of a leaky ReLU.

In addition, since this also occurred only on ReLUs, it meant that one or more of the layers were outputting large values. This, in turn could lead to the conclusion that the weights in one or more layers may have become excessively large.

The source of the problem was discovered when we temporarily switched out the dataset we used for CIFAR-10 (see Subsection 2.6.1).

Since CIFAR-10 is already adjusted and scaled, we could thus rule out any problems related to the dataset itself. When we trained our network for the new data, we discovered the same symptoms we had seen until now: The neural network would suddenly start classifying all images as the same category. In addition, sometimes the network simply wouldn't converge to any higher accuracy than the starting one. These two issues, combined with the high loss, were symptoms of the "Exploding Gradient Problem". For more information about this, see Section 2.6.

Figure 2.5 and Figure 2.4 shows the loss and accuracy when we tested the model on CIFAR-10. Figure 2.4 displays training and validation loss on the CIFAR10-dataset, while Figure 2.5 displays the corresponding accuracy. Here, the x-axis shows the epochs while the y-axis displays the loss and accuracy, respectively. The sudden increase in loss on the right hand side of Figure 2.4 is when the gradients become NaN, causing the Neural Network to stop learning. This can also be seen in the corresponding drop in accuracy in Figure 2.5.

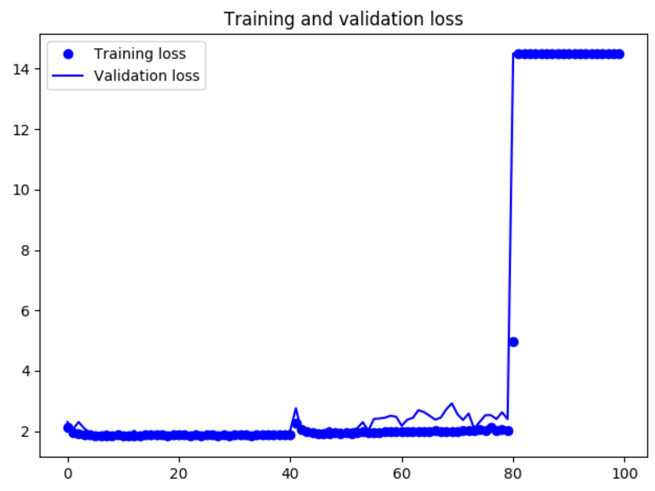


Figure 2.4: Loss plot for training and test data on CIFAR10 dataset, prior to adressing exploding gradients. X-axis is epoch number, while the y-axis represents loss (Lower is better).

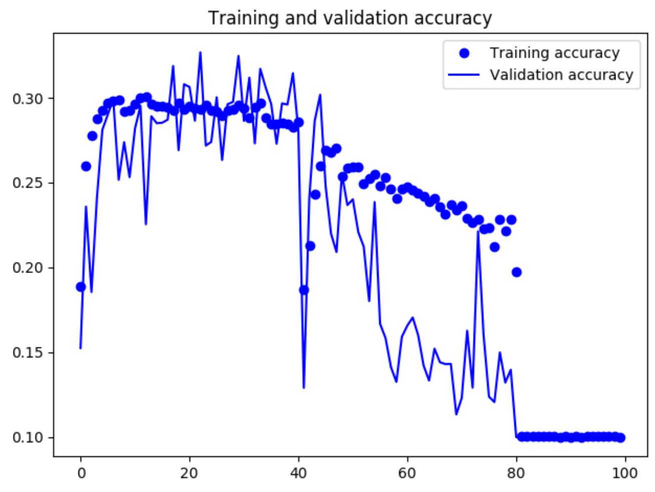


Figure 2.5: Accuracy plot when trained on CIFAR10 prior to addressing Exploding Gradients. x-axis is epoch number while the y-axis is accuracy

2.10.3 Measures taken

We were able to prevent this from affecting the neural network through a few different techniques. First, we enabled gradient clipping on the learning algorithm. This reduced the chance of exploding gradients occurring, but didn't remove the problem entirely. Because of this, we also enabled l2 normalization on each layer. After this, exploding gradients were no longer observed.

2.10.4 Recognizing Horizons and Wells from RMS

After being successfully able to detect the background colours in the viewport, the focus went towards detecting two objects: horizons (Figure 2.6) and wells (Figure 2.7). In RMS, a horizon is a representation of a geological sub-surface. The wells in RMS are represented as points, with corresponding curves and labels representing geological data.

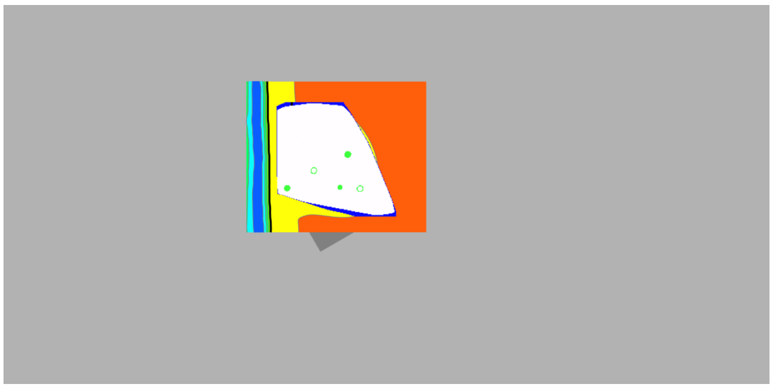


Figure 2.6: Screenshot of viewport from RMS showing a horizon represented as a two-dimensional surface in $\mathbb{R}^3$

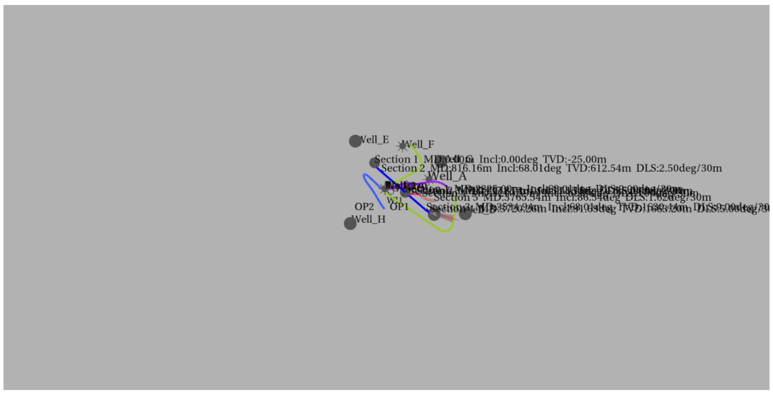

Well_E
Well_F
Well_G
Section 1 MD:0.00m Incl:0.00deg TVD:-25.00m
Section 2 MD:816.16m Incl:68.01deg TVD:612.54m DLS:2.50deg/30m
Well_A
Section 5 MD:3765.54m Incl:86.54deg DLS:1.62deg/30m
WI1
OP2
OP1
Section 3 MD:3534.94m Incl:68.01deg TVD:1630.44m DLS:0.00deg/30
Section 4 MD:3720.26m Incl:91.69deg TVD:1683.20m DLS:5.00deg/30
Well_H

Figure 2.7: Zoomed-out view of the wells present in RMS' demo project, Emerald

The work on this part was started by generating the dataset. As with Subsection 2.10.1, images were captured by interacting with the 3D-view in RMS. This time, around 1600 images were created. After the images were generated, they were copied into two folders: one for the horizon classifier and one for the wells. From here, the images were sorted into two subfolders: True if the image contained the object in question and false otherwise.

The horizon detection task was worked on first, since this was the most difficult. After the horizon detection was optimized, the same model was used for the well detection.

During testing, the network started overfitting quickly. Since we already employed a dropout of 0.7 during training, we had to look at alternative methods to counteract the overfitting. Thus, we lowered the learning rate (first to 1/10th of the initial value, then 1/100th).

After the training and validation of the models were done, we tested them on Figure 2.8. This image was used for a couple of reasons:

1. It was not part of the training set for either model.
2. It contains both a horizon and wells
3. The horizon and wells overlap, making the detection task more difficult

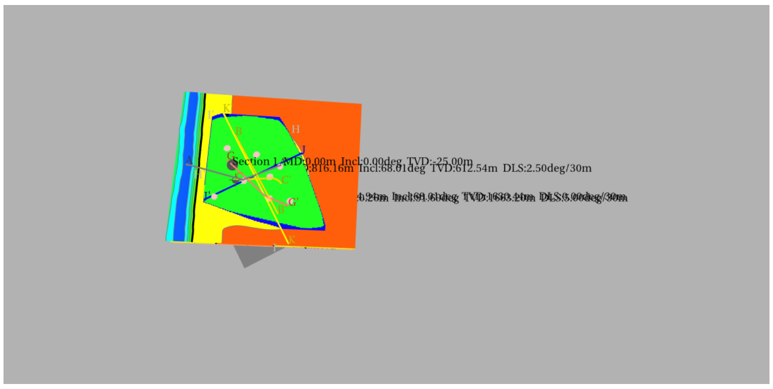


Figure 2.8: Image used to test the models after the training was complete. Both a horizon (surface) and well (point on top of horizon) are visible here.

2.11 Conclusion

In this chapter, we looked at the various methods used. The most important choices were the choice of activation functions (Leaky ReLUs and SoftMax) (Section 2.2) and choice of network model (VGG16-based) (Section 2.8). The activation functions were used due to tradition within the field of convolutional neural networks. Meanwhile the network model itself was used both based on the thesis author's familiarity, in addition to its simplicity making it more compelling for prototyping.

In addition to this, we covered the measures that were taken to eliminate exploding gradients (Section 2.6). Here, gradient clipping diminished the problem, while l2-normalization eliminated it in our case. Finally, we described the procedure that was followed during the thesis (Section 2.10).

The methods described in this chapter are used to generate the results shown in Chapter 3. Furthermore, ways to further improve the model is dicussed in Chapter 4.

3 Results

Over the course of the thesis, we trained the model for three purposes: Background-color-, Horizon- and well detection. In all three cases, the snapshot from training with the highest combined training- and validation-accuracy was used for the classifications below. The training/validation split is 80/20.

In all graphs, the x-axis shows epoch number (starting from epoch 1 at $x = 0$). In the loss-graphs, the y-axis represents loss (lower is better). For accuracy-graphs, the y-axis is accuracy (from 0 to 1, where 1.0 equals 100% accuracy).

The tables represent the validation performance of the individually trained models. The metrics here are precision, recall and f1-score. For more information about these metrics, see Table 2.1 in Section 2.9. The analysis of the results will be presented in Chapter 4.

3.1 Background Color

Figure 3.1 displays the accuracy during training on the background-colour-detection task (see Subsection 2.10.1). As with the previous accuracy graph, Figure 3.1 displays the training accuracy over 40 epochs (x-axis). Table 3.1 shows the validation performance of the snapshot with the highest combined training and validation accuracy in Figure 3.1 (epoch 30).

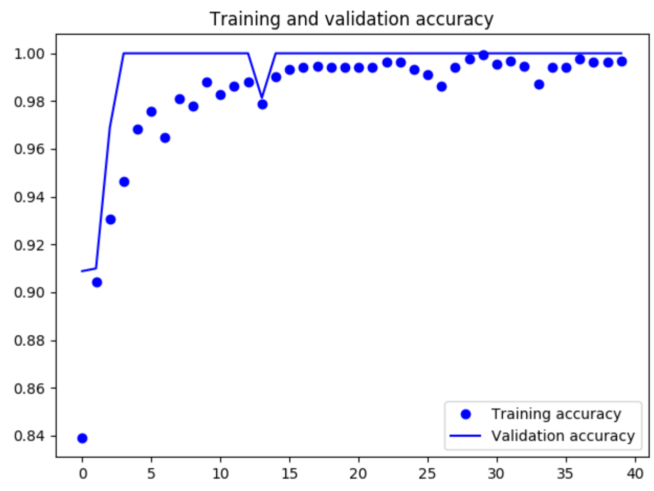


Figure 3.1: Accuracy plot for background color task. x-axis is epoch number and y-axis is accuracy

Background colour	precision	recall	f1-score	support
Class 0 (non-white)	1.00	1.00	1.00	147
Class 1 (white)	1.00	1.00	1.00	22
micro avg	1.00	1.00	1.00	169
macro avg	1.00	1.00	1.00	169
weighted avg	1.00	1.00	1.00	169

Table 3.1: Classification report for validation of background-colour detection model. The first three columns are decimal representations (from 0 (0%) to 1.00 (100%)), while the last column is the amount of images in each class/-dataset)

3.2 Horizon

Figure 3.2 shows accuracy for the horizon detection model during training. The x- and y-axis are epochs and accuracy, respectively. Figure 3.3 displays the loss over the course of the training. We used the model saved after epoch 33 for the validation, the results of which are displayed in Table 3.2.

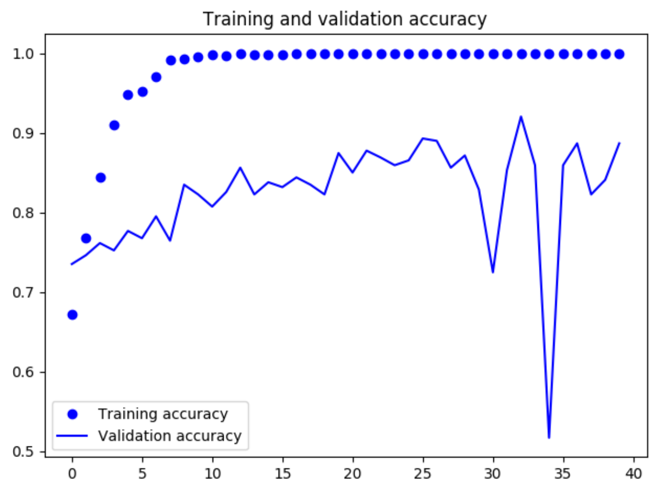


Figure 3.2: Accuracy plot for horizon detection training. X-axis is epoch number and y-axis is accuracy (higher is better)

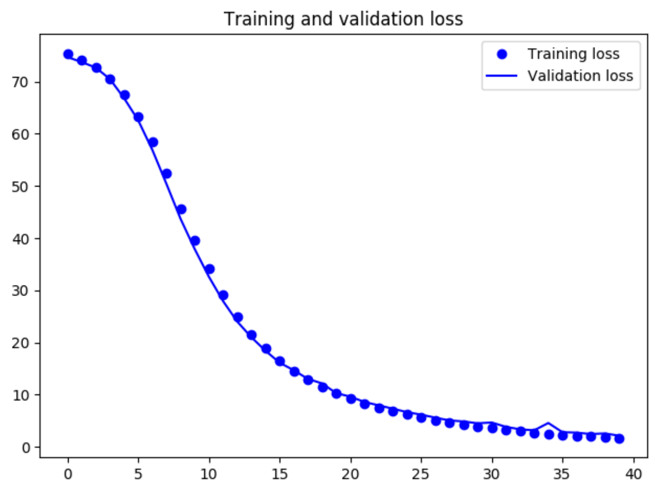


Figure 3.3: Loss plot for horizon detection training. x-axis is the epochs, while the y-axis is loss (lower is better)

Horizon	precision	recall	f1-score	support
Class 0 (true)	0.91	0.94	0.93	247
Class 1 (false)	0.84	0.77	0.80	96
micro avg	0.90	0.90	0.90	343
macro avg	0.88	0.86	0.87	343
weighted avg	0.89	0.90	0.89	343

Table 3.2: Classification report for validation of horizon detection model. The first three columns are decimal representations (from 0 (0%) to 1.00 (100%)), while the last column is the amount of images in each class/dataset)

3.3 Wells

Like the previous sections, Figure 3.4 displays the accuracy during training of the well-detection model, while Figure 3.5 shows loss during the same period. Here, the model saved after epoch 25 was used for verification. The classification performance for this model is shown in Table 3.3.

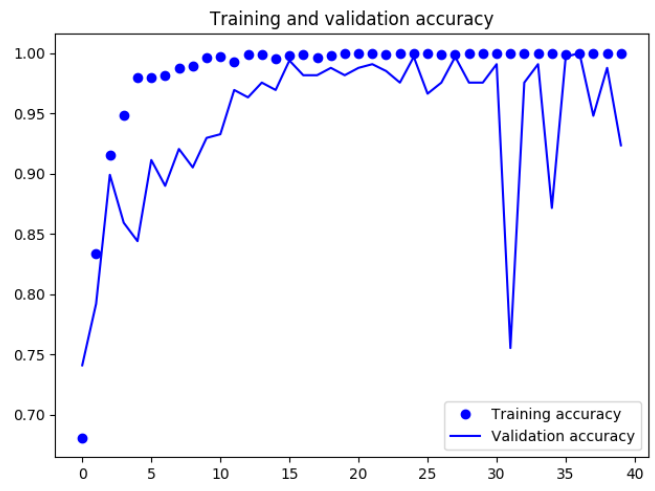


Figure 3.4: Accuracy plot for well detection training. X-axis is epoch number and y-axis is accuracy (higher is better)

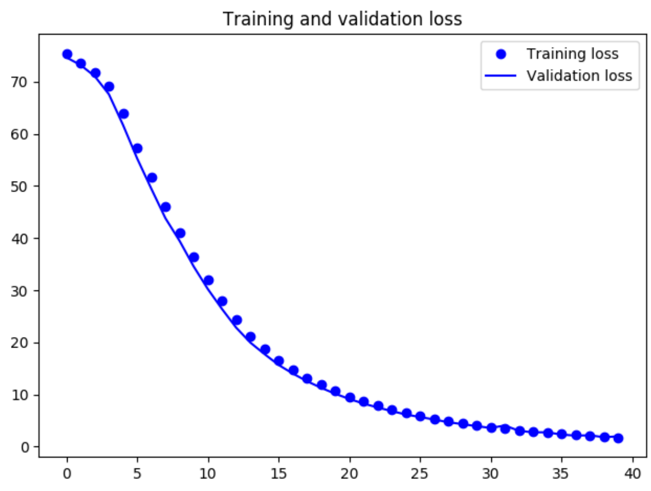


Figure 3.5: Loss plot for well detection training. X-axis is epoch number and y-axis is loss (lower is better)

	precision	recall	f1-score	support
Class 0	0.99	1.00	1.00	151
Class 1	1.00	0.99	1.00	192
micro avg	1.00	1.00	1.00	343
macro avg	1.00	1.00	1.00	343
weighted avg	1.00	1.00	1.00	343

Table 3.3: Classification report for validation of well detection model. The first three columns are decimal representations (from 0 (0%) to 1.00 (100%)), while the last column is the amount of images in each class/dataset)

3.4 Comparison

Table 3.4 displays the prediction made by the horizon and well models on Figure 2.8. This image was not included in the training or validation data for neither model.

Image	Horizon Model		Well Model		Expected	
	true	false	true	false	Horizon	Wells
Figure 2.8	0.9996	0.0004	0.9859	0.0140	true	true

Table 3.4: Comparison between the trained horizon and well models on classifying Figure 2.8. The first four colums show the predicted probability for each model, while the expected column shows the expected classifications.

4 Discussion

In this section, we will first analyse the results from Chapter 3. This extends to analysing all three of the models and comparing them to existing techniques. Afterwards, this chapter will discuss potential methods to collect training data for test-verification applications. Finally we will discuss future work that can be derived from this thesis.

4.1 Analysis

In all three cases, the compiled model has a very good accuracy on the validation data, as shown in Table 3.1, Table 3.2 and Table 3.3. In this section, the results of the three models will be analyzed.

4.1.1 Background Colour

The background colour trained network is able to classify white backgrounds with 100% efficiency (see Table 3.1). This may be due to the simplicity of the task, as in all cases the background compromised about 50% of the image. In addition, there were only two classifications: *white* and *non-white*. Since colour detection is a fairly simple task, this could be potentially extended to classifying colours into more categories than white and non-white. For the future, it could also be interesting to see how the background-colour detection would handle backgrounds with colour gradients.

4.1.2 Horizon

As we can see in Section 3.2, horizon detection had the lowest accuracy of the three tests, with a 89% weighted average. This may be because the horizon is harder to recognize with rotation, since it is a surface (with surfaces being somewhat more suceptible to rotation than other objects, like lines or points).

Another thing to note is the 77% recall for images in the *false* class, which is low compared to the rest of the metrics. This may be a corncern since this may lead to a high amount of false positives, which is not ideal when verifying tests. This is because a false positive may lead to a UI test passing when it is supposed to fail.

One explaination for the low recall rate may be that there are other planes in the dataset other than the horizon being recognized, and that the network may classify them incorrectly. This may for instance happen at a sharp angle, where the differences in features are harder to distinguish.

4.1.3 Wells

When compared to the horizon detection (Section 3.2), the model trained on detecting wells performs quite well, as shown in Section 3.3. One possible reason for this high performance is that the wells have text labels next to them. These labels stay static regardless of rotation and zoom level. This makes the well detection tasks easier for the model, but it also raises a valid concern that the model might return a false negative on a test if the well labels disappear.

4.1.4 Direct comparison with existing techniques

While the results here show that the model we trained is able to recognize objects, it still remains to be seen how it performs compared to other verification tools, such as Perceptual Image Diff (Subsection 1.3.2) or Blink-diff (Subsection 1.3.1). If the test is dependent on correctly verifying that an object is present in the view, we believe that the CNN should be able to have a lower error rate. However, we have not conducted experiments to verify it, and we will leave comparison with other tools as a future work.

That being said, neural networks could cover a fields of image verification that isn't directly related to comparisons, such as classification and feature detection.

4.2 Collection of training data

Over the course of the thesis, a valid question was raised: How would someone collect the data to train a neural network to verify UI-tests? There are a couple of ways this could be done.

4.2.1 Manually

The first way to generate the training set would be to manually interact with the UI and manipulate viewports or other parts of the UI to generate the data. This was the method that we used over the course of this thesis, together with a degree of semi-automation. The drawback to this approach is that it is very repetitive for the person collecting the data, so we would not recommend it if the data set is going to be large.

4.2.2 Scripting or automation

Another way to generate data would be to instead offload the interaction to an UI test script or to use another form of automation. In this case, as long as the manipulation required to generate data can be scripted in some sense, this is a possible way to get a test set. However, if the interaction cannot be scripted, it will instead have to be done through other means. In addition to this, the data will still have to be verified. This is both to classify the data, and to make sure that it is suitable/relevant for the training.

It is also worth noting that semi-automation is a possibility. For example, the user could do manual manipulation of the zoom level and then use scripting to rotate the viewport at random. Another approach to semi-automation would be to manipulate the viewport manually, and using scipts to automate saving images of the viewport. This allows the user generating the data to perform operations that cannot be automated while still avoiding repetitive tasks.

4.2.3 Adversarial Neural Networks

Adversarial neural networks could be used to generate additional data samples. Generative Adversarial Networks use an adversarial process to generate new data based on an existing dataset (Goodfellow et al., 2014). Zhang et al. (2018b) explored a self-paced learning approach, where both real and artificial data are given to a neural network. One drawback to this approach is that the data may not be authentic data from the viewport itself, but it should hopefully still be able to generate valid data for training.

4.2.4 Crowdsourcing

The final suggestion we have is to crowdsource the generation of test data. One way to do this could be to collect the dataset when performing other tests, e.g. when a new commit is being tested for regressions.

The data itself can also be classified through crowdsourcinPeceptualSectiong. One way would be to have a popup open in the program and to ask the user to verify the type of object. On the other hand, this may be intrusive for the user, depending on who receives these popups and how often they appear.

For example, development builds for RMS are constantly generated and tested by testers. If these queries were only included in the development builds, then only the developers and testers would be exposed to them. Thus, using this approach would ensure that data could be collected in this way without affecting the end user.

4.3 Future work

While the neural network was able to detect the objects in a 3d-view, there are still topics that could have been explored.

4.3.1 Detection of Multiple objects

This thesis focused primarily on detecting singlar objects, but it would also be interesting to see how well a neural network trained on detecting multiple objects would perform.

One example would be to train a single neural network to detect if you have multiple instances of an object in the view, and if so, how many. This could be an useful extension of e.g. the well-detection task in this thesis. Thus, the neural network could count the amount of objects in the view and fail the test if there is a discrepancy, e.g. in the well count. If this is explored further, the metods described in Yang et al. (2018) and Xue et al. (2016) may be useful as a starting point.

4.3.2 Classifications of subtypes

Another interesting case would be to have multiple different classifications. For example, in the horizon detection we focused on detecting a specific type. Meanwhile, RMS has several different horizons present in the demo project. Thus, a neural network could be trained to detect the different types of horizons. This would require a larger dataset, since transformation of the new classes would have to be accounted for.

4.3.3 Consolidation detection to a single network

It could also be interesting to look at consolidating the different neural networks into a single neural network to save disk space. This neural network could then seperate between multiple different objects, e.g. wells and horizons.

The main hurdle that we can see as of now would be to handle cases when you have multiple types of objects in the view, for instance both the wells and a horizon. This is because the probabilities given out by the softmax activation has to sum up to 1. Thus, a different activation function will have to be used in this case. A bayesian network could be a potential solution here.

Additionally, instead of a single softmax layer, we could use multiple layers in parallel. The drawbacks to this is that it would make structuring the training data harder, since an image would have multiple classifications at once, but it would provide simultaneous classification for multiple types of objects.

4.3.4 Non-object scenarios

While the detection of objects is one way to verify an UI test, another approach could be to supply a neural network with test data that does not necessarily contain a specific object. It would be interesting to see what would happen if the neural network was supplied with regular results on 'true' and known regressions on 'false'. This way, the neural network would learn to look for features that were tied to the specific regressions.

4.3.5 Integration into RMS

Our final proposal for future work is to integrate the convolutional neural network into the RMS suite. This would allow the verification of UI tests detecting objects of interest. As an example, a test could be run on a job that creates a Well. The neural network would then verify that the job actually results in a 3D-view containing the well. This approach would also be valid for other programs and applications, but then the training set would need to be generated with that program or application in mind.

5 Conclusions

In this thesis, we have investigated the possibility of applying Convolutional Neural Networks for UI test verification. After conductiong extensive experimentation using data obtained from a real 3D application, we have reached the following conclusion: it is possible to use Convolutional Neural Networks to verify a class of UI tests. However, it is currently uncertain if classification using this method is superior to image comparison for ordinary test cases. This is especially true with the neural network requiring a training set. This training set needs to include at least two classes. For the purpose of automatically verifying UI tests, these classes can be *true* and *false*

The benefit of the neural network over a traditional tool is that a the neural network can look for features and details in the picture, while a traditional image comparison can only compare the images.

Thus, we can conclude that with enough data it is possible to use neural networks for automatic verification of UI tests.

The claims that we make is further supported by empirical data. The experiments to obtain this data can be replicated by using the source code available in Appendix B.

5.1 Future Work

As discussed in Section 4.3, there are a several areas that could be explored as future work. One would be detection of multiple objects, where the amount of objects in view could be counted. The second direction would be to see if subtypes could be successfully classified. It would also be interesting to see if the specialised networks could be consolidated into a single network. The final field for future work could be to see how the convolutional neural network would learn on a task that isn't directly related to detecting specific objects.

Bibliography

Aharoni, Z., Rattner, G., and Permuter, H. (2017). Gradual learning of recurrent neural networks.

Aho, P., Suarez, M., Kanstrén, T., and Memon, A. (2013). Industrial adoption of automatically extracted gui models for testing. volume 1078, pages 49–54. CEUR-WS.

Bremdal, B. A. (2018). Artificial intelligence and intelligent agents lectures. Personal communication.

Burkimsher, P., Gonzales-Berges, M., and Klikovits, S. (2011). Multi-platform scada gui regression testing at cern. pages 1201–1204.

Burt, P. and Adelson, E. (1983). The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4):532–540.

Calik, R. C. and Demirci, M. F. (2018). Cifar-10 image classification with convolutional neural networks for embedded systems. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–2. IEEE.

Chen, D., Odobez, J.-M., and Bourlard, H. (2004). Text detection and recognition in images and video frames. *Pattern Recognition*, 37(3):595–608.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Froglogic (2019). Squish. `https://www.froglogic.com/squish/`.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. volume 15, pages 315–323.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks.

Google (2019). Puppeteer. https://developers.google.com/web/tools/puppeteer/.

Habibi Aghdam, H. (2017). Guide to convolutional neural networks : A practical application to traffic-sign detection and classification.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. *University of Toronto*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3.

Moore, J., Pfeiffer, J., Wei, K., Iyer, R., Charles, D., Gilad-Bachrach, R., Boyles, L., and Manavoglu, E. (2018). Modeling and simultaneously removing bias via adversarial neural networks.

Pelt, D. M. and Sethian, J. A. (2018). A mixed-scale dense convolutional neural network for image analysis. *Proceedings of the National Academy of Sciences of the United States of America*, 115(2):254–259.

Philipp, G., Song, D., and Carbonell, J. G. (2017). The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions.

Protractor (2019). Angularjs. https://www.protractortest.org.

Python Software Foundation (2019). Python language reference.

Roxar Software Solutions (2019). Rms.

Saikia, N. and Bora, P. (2014). Perceptual hash function for scalable video. *International Journal of Information Security*, 13(1):81–93.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting.

Journal of Machine Learning Research, 15:1929–1958.

Swofford, M. (2018). Image completion on cifar-10.

Tensorflow (2019). Tensorflow. https://www.tensorflow.org/.

The Qt Company (2019). Qt test overview. https://doc.qt.io/qt-5/qtest-overview.html.

Wu, Q. and Li, Z. (2018). A fast and general method for partial face recognition. volume 10735, pages 215–224. Springer Verlag.

Xue, Y., Ray, N., Hugh, J., and Bigras, G. (2016). Cell counting by regression using convolutional neural network. volume 9913, pages 274–290. Springer Verlag.

Yahoo (2016). Blink-diff. https://github.com/yahoo/blink-diff.

Yang, B., Cao, J., Wang, N., Zhang, Y., and Zou, L. (2018). Counting challenging crowds robustly using a multi-column multi-task convolutional neural network. *Signal Processing: Image Communication*, 64:118–129.

Yee, H. (2004). Perceptual metric for production testing. *Journal of Graphics Tools*, 9(4):33–40.

Yee, H. (2019). Perceptual image diff. http://pdiff.sourceforge.net/.

Zhang, F., Cai, N., Cen, G., Li, F., Wang, H., and Chen, X. (2018a). Image super-resolution via a novel cascaded convolutional neural network framework. *Signal Processing: Image Communication*, 63:9–18.

Zhang, X.-Y., Wang, S., Lv, Y., Li, P., and Wang, H. (2018b). Augmented self-paced learning with generative adversarial networks. volume 10862, pages 450–456. Springer Verlag.

Zhen-Kun, W., Wei-Zong, Z., Jie, O., Peng-Fei, L., Yi-Hua, D., Meng, Z., and Jin-Hua, G. (2010). A robust and discriminative image perceptual hash algorithm. pages 709–712. IEEE.

A
List of Abbreviations

Adam	Adaptive Movement Estimation
API	Application Programming Interface
CIFAR	Canadian Institute For Advanced Research
CNN	Convolutional Neural Network
CPU	Central Processing Unit
GPU	Graphics Processing Unit
NaN	Not a Number
ReLU	Rectified Linear Unit
RMS	Reservoir Modelling Software
UI	User Interface
UiT	The Arctic University of Norway (Formerly University of Tromsø)

/B

Code and Dataset

Attached in a zip-archive with the rest of the thesis is the code that was used to train the models. The code takes the form of a project written in Python 3, with dependencies on Keras and Tensorflow. The requirements.txt file contains the list of packages required to run the project. All python files have the .py extension.

Additionally, the image set used to train the horizon and well models is also contained in this zip-archive. These images are saved in png-format.

/C Problem Description

Included here is the problem description. It details the aim of the thesis. The initial survey has been incorporated into Chapter 1 and is thus not included.

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering
UiT - The Arctic University of Norway

Automatic verification of UI tests

Christopher Kragebøl Hagerup

Thesis for Master of Science in Computer Science

Problem description

Automated UI scripting is a valuable tool for testing of interactive software. However, its applicability is limited because verifying the test results can be challenging. On one hand, manual verification by testers is time consuming, expensive and may not be available to all projects. On the other hand, existing automated verification methods could lead to false positives, making the test results unreliable.

Objectives:
Determine to what extent automated verification of test results is beneficial. First of all, is it possible to completely automate the verification? If not, what are the benefits and/or drawbacks of partial automation?

Implement a prototype to verify the hypothesis and show the benefits it will bring to test verification.

<u>Structure the problem</u>.
- What kind of typical issues / mismatch do we expect?
- How can we recognize these issues?
- How much, and which, information do we need?

<u>Recognition part</u>.
- Obtain an overview of possible technologies which could be considered, e.g.
    - Object recognition
    - Image based, text, et cetera
    - ...
- Develop a proof of concept based on one or several methods of choice.

<u>Verification part</u>.
- Consider a few alternative approaches, e.g.
    - AI based methods
    - Computer Vision / Image analysis and comparison
    - Text recognition
- Implement a prototype for analysis of one or more spesific types of recognision
- Analysis and comparison
- Errors and tolerances
- Presentation of the results

<u>Implementation</u>.
The choice of programming language, third party libraries, operating system etc. may be restricted per agreement with the supervisors. The same applies for solution architecture and mathematical models.

Information, provided by Roxar [https://www.emerson.com/no-no/automation/operations-business-management/reservoir-management-software].

- Documentation. Characteristics and measurements.
- Experimental results.
- Known behavior of the automated tests under varying conditions, including ones that fail.
- Assumptions.

Dates

Date of distributing the task:	<07.01.2019>
Date for submission (deadline):	<11.06.2019>

Contact information

Candidate	Christopher Kragebøl Hagerup cha113@post.uit.no
Supervisor at UiT-IVT	Rune Dalmo rune.dalmo@uit.no
Supervisor at UiT-IVT	Hans Olofsen hans.olofsen@uit.no
Supervisor at Roxar / Emerson	Lucas Provensi lucas.provensi@emerson.com

General information

This master thesis should include:

❄ Preliminary work/literature study related to actual topic
  - A state-of-the-art investigation
  - An analysis of requirement specifications, definitions, design requirements, given standards or norms, guidelines and practical experience etc.
  - Description concerning limitations and size of the task/project
  - Estimated time schedule for the project/ thesis

❄ Selection & investigation of actual materials
❄ Development (creating a model or model concept)
❄ Experimental work (planned in the preliminary work/literature study part)
❄ Suggestion for future work/development

Preliminary work/literature study

After the task description has been distributed to the candidate a preliminary study should be completed within 3 weeks. It should include bullet points 1 and 2 in “The work shall include”, and a plan of the progress. The preliminary study may be submitted as a separate report or “natural” incorporated in the main thesis report. A plan of progress and a deviation report (gap report) can be added as an appendix to the thesis.

In any case the preliminary study report/part must be accepted by the supervisor before the student can continue with the rest of the master thesis. In the evaluation of this thesis, emphasis will be placed on the thorough documentation of the work performed.

Reporting requirements

The thesis should be submitted as a research report and could include the following parts; Abstract, Introduction, Material & Methods, Results & Discussion, Conclusions, Acknowledgements, Bibliography, References and Appendices. Choices should be well documented with evidence, references, or logical arguments.

The candidate should in this thesis strive to make the report survey-able, testable, accessible, well written, and documented.

Materials which are developed during the project (thesis) such as software / source code or physical equipment are considered to be a part of this paper (thesis). Documentation for correct use of such information should be added, as far as possible, to this paper (thesis).

The text for this task should be added as an appendix to the report (thesis).

General project requirements

If the tasks or the problems are performed in close cooperation with an external company, the candidate should follow the guidelines or other directives given by the management of the company.

The candidate does not have the authority to enter or access external companies’ information system, production equipment or likewise. If such should be necessary for solving the task in a satisfactory way a detailed permission should be given by the management in the company before any action are made.

Any travel cost, printing and phone cost must be covered by the candidate themselves, if and only if, this is not covered by an agreement between the candidate and the management in the enterprises.

If the candidate enters some unexpected problems or challenges during the work with the tasks and these will cause changes to the work plan, it should be addressed to the supervisor at the UiT or the person which is responsible, without any delay in time.

Submission requirements

This thesis should result in a final report with an electronic copy of the report including appendices and necessary software, source code, simulations and calculations. The final report with its appendices will be the basis for the evaluation and grading of the thesis. The report with all materials should be delivered according to the current faculty regulation.

If there is an external company that needs a copy of the thesis, the candidate must arrange this. A standard front page, which can be found on the UiT internet site, should be used. Otherwise, refer to the “General guidelines for thesis” and the subject description for master thesis.

The supervisor(s) should receive a copy of the the thesis prior to submission of the final report. The final report with its appendices should be submitted no later than the decided final date.