Object Recognition in Augmented Reality

Jonatan Atles och Jan Svensson

Master's thesis
2018:E78

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Lund University
Lunds Tekniska Högskola
Centre for Mathematical Sciences

OBJECT DETECTION IN AUGMENTED REALITY PROJECT REPORT

Hand-in date: December 22, 2018

Jan Svensson, elt12jsv@student.lu.se
Jonatan Atles, elt13jat@student.lu.se

Abstract

Augmented reality is a field which is getting increased funding every year, as more businesses are realizing the potential of rendering virtual objects in the real world. As the equipment gets more commercialized, the costs will get lowered while performance also goes up. As of now, augmented reality mostly makes use of *plane detection* and *marker detection* to find and locate objects. We want to incorporate *machine learning* using *deep neural networks* to be able to find objects in an augmented reality scene.

In this report we go through the process of developing an *iOS* application, which incorporates use of *object detection*, *object recognition* and augmented reality. Our goal is to identify if the combination of these fields is both feasible and desirable with modern tools available. The application is supposed to work as an alternative to conventional assembly manuals for furniture.

The project is about showing a user how to, step by step, put together a furniture using an *iPhone X* and give instructions in an augmented reality experience. The machine learning model is taught to recognize all the different parts of a furniture and where they are in the image. Different methods were tried to make this possible, such as *image segmentation*, *edge detection* and *object detection*. One of the object detection methods used is called *YOLO*.

The final product makes use of a toolkit useful for developing augmented reality application, called *ARKit*. It is developed by *Apple*, and is used to render the augmented reality scene. It also uses *Turi Create* to train a neural network for object detection and classifier.

We then evaluate the application with user tests. We further conclude that modern technology available makes our idea possible and the user tests show that the concept has great potential and is desirable.

Acknowledgements

We would like to give our thanks to our supervisor from LTH, Karl Åström. We would also like to give our thanks to Jayway for giving us the opportunity to do this project, letting us use their equipment and facility, as well as inviting us to several conferences and events where we could improve our competence in AR and machine learning. Thanks to Andreas Kristiansson and Andreas Back from Jayway for their feedback as well as letting us pitch ideas with them. Thanks to Tomas Nilsson for giving us the idea and showing interest in the project . Thanks to all the people that helped us out by performing user tests.

Contents

Chapter 1

Introduction

To first get an understanding of why this project was done, it is important for the reader to learn, not only the background of the fields studied, but also about the prospects of the technologies involved. At 1.1 we go through the current and the potential use of these technologies, as well as what prospects the consumers and the industry have. Later on, at 1.2 we go through some of the previous research that has been done in the fields that are studied in the report. In section 1.3 we explain what we hoped to achieve with this project and a brief rundown of how. Section 1.4 tells about the company that ordered the project and why they were interested in it. This chapter is rounded off with section 1.5, which illustrates a mock up of how we had hoped the final product would have ended up looking like.

The full repository for this project can be downloaded at `https://github.com/iSadist/master-thesis`.

1.1 Background

The idea of Augmented Reality, often referred to as just AR, is to render virtual objects in the real world. This usually requires hardware in the form of a camera and display, a processing unit and software. Common devices capable of AR today are the HoloLens [1], Google Glass [2] and a vast amount of mobile devices, such as Apple's iPhone X [3].

The general interest for AR has undeniably grown in recent years, with mobile games and applications such as Niantic's Pokémon Go [4] and IKEA Place [5] the AR industry has peaked a general interest and sources speculate it could be a $90 billion dollar industry by 2022 [6]. But Augmented Reality has not only reached the end users; the technology has also peaked an interest in several industries. One such project is Fieldbit Hero [7], a platform enabling technicians to get instant AR annotations to their AR devices from an engineer, allowing the technician to get visual instructions while simultaneously being able to work hands-free.

However, as of now, most localization and identification of objects within AR are done using marker detection [8]. An alternative to this would be to use *Convolutional Neural Networks*, as they have been proven to be very useful for

object detection in images. The hype around neural networks has been particularly strong since AlexNet scored high in the ImageNet LSVRC-2010 challenge and the ImageNet ILSVRC-2012 challenge [9], which are are image classification contests. They accomplished these feats by making use of new methods, e.g. *dropout* (explained in section 4.1.6). Since then several different types of network models have been created. Now we have network models that can detect both where an object is in an image, and what the object is in one go. One such network is *YOLO*[10]. By adopting such a network, the need for having to mark objects would drop significantly and it could open up for many more possible use cases for AR.

1.2 Previous studies

D. Chatzopoulos et al.(2017) describes the basics of MAR, or Mobile Augmented Reality, its advancements and it's flaws. They estimate that MAR is the most promising field of mobile applications and that it will have a massive impact on how we interact with the real world. However, they also go through some of the current hiccoughs with the technology, such as bandwith limitation and the computing power required [11].

S.Gould et al.(2009) writes about a hierarchical model for joint object detection and image segmentation, where they basically tried to segment all objects in an image and classifies every pixel [12].

Y LeCun and M.A. Ranzato goes in-depth on the history and progress of deep learning, from the first machine learning model, the Perceptron, to future challenges with deep learning. They write about the use cases for machine learning and how models generally work. They also classify different kinds of models into separate categories. Also written about is areas such as what makes a good feature and how convolutional neural networks are constructed [13].

1.3 The goal of this project

Our goal is to try to combine Augmented Reality with Object detection and Object recognition to find out if it is feasible and if there is an actual desirable use case for this technique.

This will be limited to the use of ARKit, a toolkit for developing AR applications, developed by *Apple* for iOS devices. An augmented reality application for iPhone X incorporating the use of modern machine learning models for object detection and recognition will be developed in order to test our thesis. The application will be an assembly manual for furniture. A Deep neural network will be trained to recognize the different available parts and configurations of said parts. The model should then predict which parts it is seeing in the camera feed, where they are and show how they should be assembled in an AR scene.

Hence, we will not research what is possible with other hardware, although, we will mention them. We will also restrict the amount of furniture we train on, as

well as the environments they will be built in. However, we hypothesize that if it works for one furniture and one set of environments, it should be possible to scale for larger sets.

1.4 Jayway

This project is conducted partly on Jayway's request and much of the work is done in their office in Malmö. Jayway is a software studio with offices in Malmö, Halmstad, Stockholm, Copenhagen and Palo Alto. They aim to provide costumers with digital solutions within fields such as mobile, web, backend, cloud as well as UX & design. They also have a focus on AR & VR, where they hope to give businesses a new perspective on the technologies.

The purpose of this project for Jayway is mainly to explore Object Recognition and ARKit, and how they could be combined together to create value. The main question is to know, if possible, how easy it is to implement a solution. Jayway is continuously looking to explore new technologies and Augmented Reality is a hot new trend emerging in the markets in the time of writing this report. A second objective is for Jayway to take the finished implementation to one of their customers and present an example of what they are able to do for their customers.

1.5 Mock-up of the planned product

To be able to design the layout and flow of the application, a mock-up of the application was first created, illustrating the information and use flow throughout the usage. In this section the general concept will be described and why it was designed in this way. The mock up was created using a free online tool called MockFlow[14].

When the application is opened up, one is taken to a furniture selection screen, see figure 1.1, which allows the user to select which furniture they want to assemble. A user can select the furniture by either scrolling through the list, using the search bar, or using the built in barcode scanner. It was important for a first time user to understand what the purpose of the application was when they first open up the app. Therefore, first time users are greeted with an informative pop-up notification explaining the general purpose.

After selecting a furniture the user is taken to a detail view, figure 1.2. This view is there for the user to see bigger images of the furniture and its unique article number. This is useful since some furniture models look very similar. Here the user can also read other specific information about the piece of furniture as well as preview a 3D model of the furniture, showing how it will look like when it's completed. Once the user is ready, they are supposed to press the "Assemble" button to progress.

In the Assemble view the user is greeted with a live feed camera view as well as an information field containing instructions, see figure 1.3. The first step for the

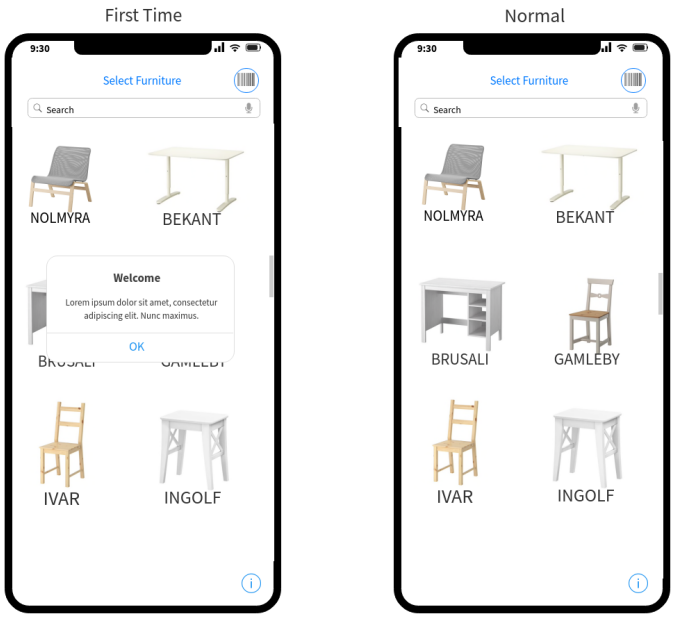

Figure 1.1: Furniture selection screen which the user is greeted with when starting the application. The left image shows how it looks for first time users, and right side how it looks for known users.

Figure 1.2: .

user is to scan the pieces on the floor, allowing the machine learning model to find the pieces that are supposed to be assembled. Once the first parts are automatically found, they are highlighted using bounding boxes to alert the user of their current importance. The user can then press the "Next" button to continue. Afterwards, an animation showing the 3D parts in question being assembled properly and informatively. When the user has put together the parts, they press "Next". This process will then be fed with the next set of instructions and will continue like this until the piece of furniture is fully assembled.

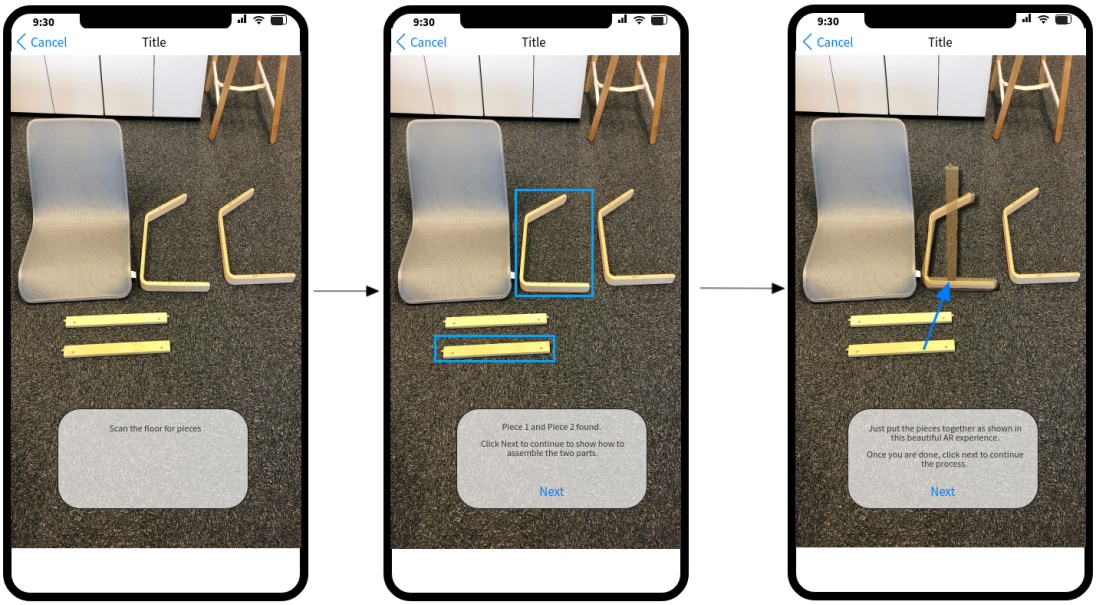

Figure 1.3: Shows from left to right the user flow of putting two pieces together.

Chapter 2

Project Methodology

When working in a project it is generally a good idea to work according to a predetermined project methodology model. This tends to make the work more effective and produce better quality. That is, more is produced during the same amount of time, the final product is more thought out, better teamwork etc.

Simply, if you have a plan at the start it is easier to stick to that plan and stay on the same path as intended, rather than shifting to another one. That is not to say that the plan cannot change, but if it does it does so controllably.

A historically good model has been the waterfall model. However, in the software industry this has changed lately with the agile methodology being more popular and it has proven to be very effective.

2.1 Agile

We have used the scrum method, which is an agile method. Basically, the scrum model says that rather than planning a workload for 6 months forward or so it is better to work in short iterations. These iterations should be between 1 - 4 weeks depending on the team. Instead of trying to estimate the time it will take to complete an entire project, the team is given a finite time frame and tries to complete as much as possible during that time. During this time, the team works very closely to the customer and project owner to make sure that they will get what they want and ask for. For selecting items to work with for every sprint (explained further in section 2.2) the team keeps a backlog of items it wishes to complete during the project. Every sprint, these items or stories are picked out and included into the sprint and estimated in size. The stories themselves should be collected from customers, project owner, users and people connected to the product. This way, the team knows what the purpose of developing a certain thing is. If there is any doubt about a specific feature it should be easier to ask than to assume.

This is of course a simplified version of how the agile method works. 'Extreme Programming Pocket Guide' is a good book for anyone who wants to read more about agile methodology [15] .

2.2 How we used the agile methodology

Sprints
For our planned work we have decided to work in two week period sprints. At each start of a sprint we pick out stories to focus on for the coming two weeks and try to estimate how long each of them will take. On weekdays we start with a quick discussion about yesterdays work and what we are planning to do today. This is for everyone to be up to speed about the other person's work. At the end of the sprint everything is reviewed and analyzed so to do even better the next sprint by correcting possible faults.

Story board
For organizing our sprints we used a story board which contained an area for our backlog items and four rows for our stories during a sprint. The stories were broken into tasks which were placed on either one of the columns (Started, In Progress, Waiting, Completed). Each story was estimated with a size which was a number in the fibonacci sequence. Each story was also divided into 5 parts (Started, Halfway done, Completed, Reviewed, Verified). These two numbers were multiplied and summed up with all the other stories to get how many points the sprints had. As we worked these points were subtracted and ultimately hit zero when we were done with everything. We plotted the progress on a chart as well for graphic representation. This is called a burndown.

Figure 2.1: Story board used during the project

Close relationship with project owner
We worked very close to the project owner by sitting on desks right across him. Whenever we had a questing we could simply raise our head and ask it.

Chapter 3

Augmented Reality

To help better the understanding of augmented reality, this chapter will describe the historical background as well as how it works. In section 3.1 there are a few different types of AR explained. Then a brief history of AR is presented in section 3.2. Afterwards, a method about mapping the environment called Simultaneous Localization and Mapping, *S.L.A.M*, is described in section 3.3. The chapter is rounded in section 3.4 with some information about how the the toolkit used in this project, *ARKit*, functions.

3.1 The different types of Augmented Reality

There are different types of augmented reality. Some of those are marker-based, location-based, superimposition-based and projection-based.

Marker-based AR is when markers, in the form of images have to be placed in the real world and detected by the application. Virtual object are rendered on top of these markers. An example would be a picture in a magazine which, when pointing a camera at it, the application renders a 3D object on top. An example is shown in figure 3.1.

Location-based AR is when the content on the users screen differs depending on the location of the user. This type is highly dependent on the GPS signal. An example of where this could be useful is in a museum where different information could be given to the user depending on which room he or she is in.

Superimposition-based AR uses object recognition in order to enhance that object with some sort of visual information. It replaces the real object with an enhanced virtual one. It could be used in retail to display different patterns on a piece of clothing.

Projection-based AR is when virtual object can be placed in a room to make it appear as if they were there. A popular example of this would be the IKEA

Figure 3.1: A virtual object in the form of a car being rendered on top of a marker in a magazine.

Place app that was mentioned in the introduction. An example is shown in figure 3.2.

In this paper we will mainly be using projection based AR with a form of recognition. The projections will be in the form of way-pointers used as instructions for the user to perform the next step in putting together a furniture piece. These instructions can visualize how the pieces will look like after the step is completed, and to show arrow pointers between the pieces that are supposed to be put together.

3.2 History of AR

The idea of augmented reality has existed a long time, the phrase has only been used for about 30 years but it is not until recently that the technology has become mainstream. This is mainly due to it becoming good enough to be used by the average person at home. Today we can just download an app on our smart phones to enjoy the technology. Below follows a brief history of how augmented reality has progressed throughout the years.

1968 The Sword of Damocles - The first mounted headset. This device was mounted on the head and could display a cube wireframe floating in the air. It was invented by Ivan Sutherland.

1975 Myron Krueger - Videoplace. Using cameras to interact with a digital world with shadows. This application could be used to draw things or play simple video games with the shadow of your hand [16].

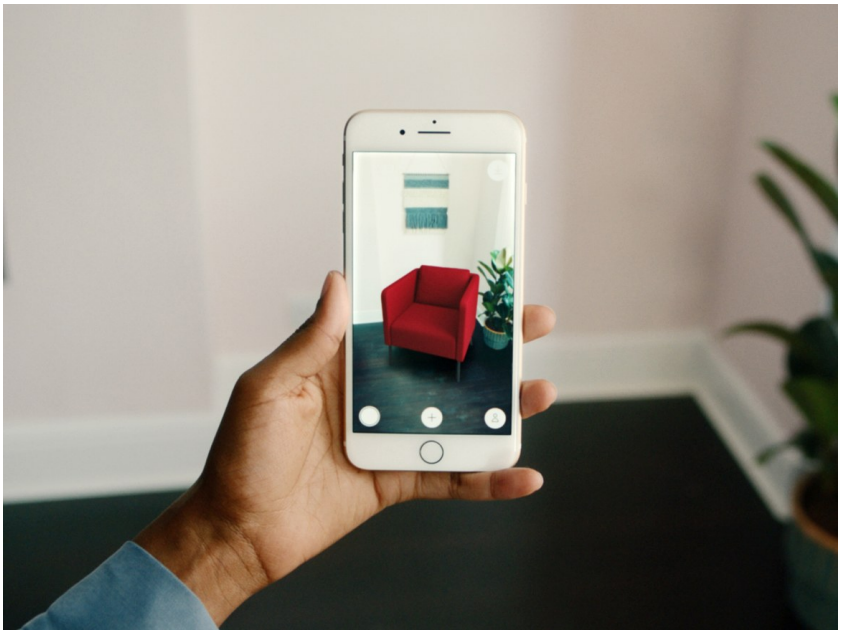

Figure 3.2: A virtual sofa chair being added to a room within the IKEA Place app.

1990 The first time the term "Augmented Reality" was used by the Boeing researcher Tom Caudell.

2009 AR comes to the web in the form of an open source toolkit called ARToolKit.

2017 Apple launches AR Kit and Google launches AR Core.

3.3 S.L.A.M.

S.L.A.M is a way for a machine to get to know the environment that it is in. It registers features and maps them to its surroundings. S.L.A.M is about having the map of the environment and knowing where the robot is in that map. The problem with this is that a map is needed for knowing where you are, and you have to know where you are to be able to create a map. That is why S.L.A.M is doing this at the same time, hence 'Simultaneous'. The system is used in autonomous robots, but also valuable in Augmented Reality [17]. In figure 3.3 a robot is driving around in a room collecting data. The robots camera can be seen in the lower left corner. On the right, a 3D model with recognized feature points has been created.

iPhone X does this by tracking multiple reference points in space and from them building a 3D model of the surroundings using a form of S.L.A.M technique. This is accomplished by keeping a map of the features while keeping track of the

path the observer is taking. A number of hardware components make this task possible, including gyroscope, accelerometer and a compass. [46]

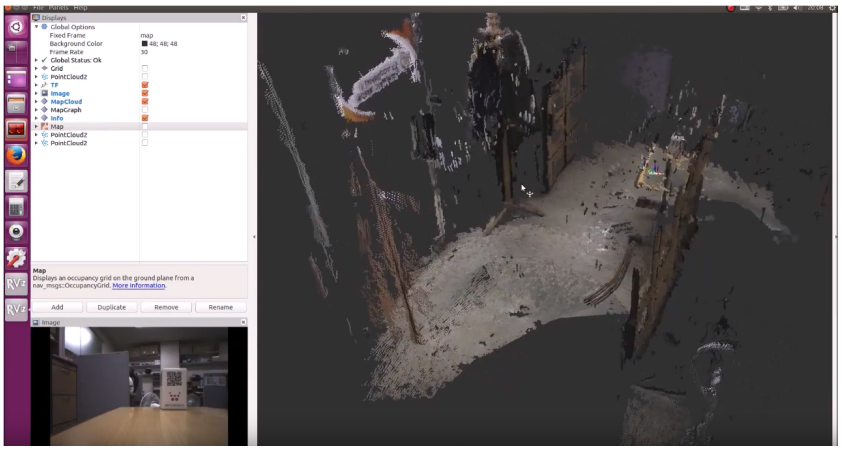

Figure 3.3: A robot performing S.L.A.M in an environment and the 3D model created.

3.4 How ARKit works

The easiest way to use ARKit is to use it through Xcode, which can be downloaded from the Mac App Store. The coding language can be either Objective-C or Swift. We will be using Swift for all examples.

ARKit works similar to SceneKit where a scene, which is basically a 3D environment in which nodes containing 3D models, known as geometries, can be rendered, is loaded at start up and interacted with. An ARScene is contained inside an ARSCNView (AR Scene View) which also has an ARSession that that manages the motion tracking and camera image processing. For an ARScene to work, it must have a running ARSession. The session is started with configuration (ARConfiguration). This configuration can be of many kinds, the most common ones being ARWorldTrackingConfiguration and ARFaceTrackingConfigurations. For this project, ARWorldTrackingConfiguration will be used since the face tracking one uses the front camera.

An example of how to setup a configuration and running a session using Swift is given in appendix A.1.

But before a session can be started, an ARScene must exist and be loaded. The ARScene is a regular .scn file that has the starting nodes that together form the starting environment the user will interact with. An example of how a scene like that looks like is shown in figure 3.4.

To load a scene, a new .scn file must be created in the art.scnassets folder and fetched, as seen in the code snippet in appendix A.2.

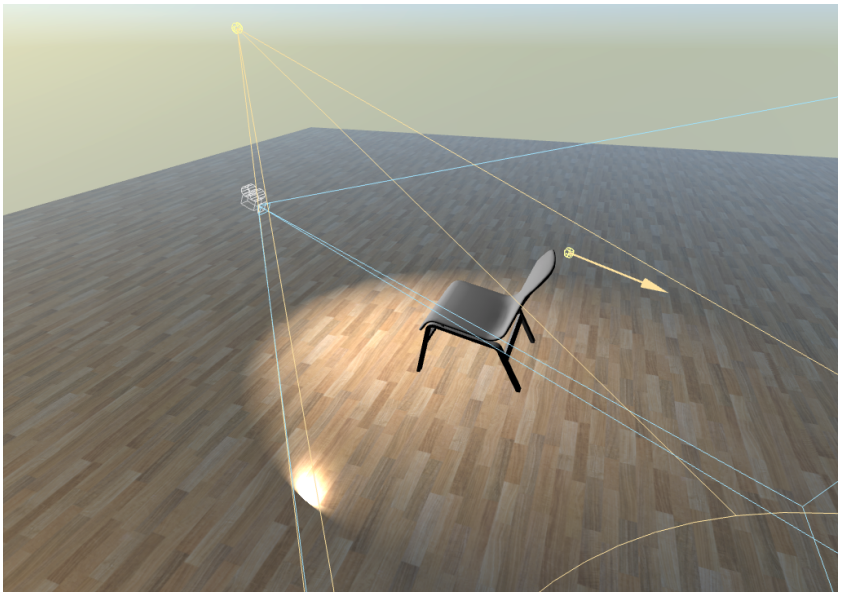

Figure 3.4: An example of a 3D scene created in XCode with a camera, a plane, directional light and three geometry nodes.

When ARScene detects objects, images, planes etc. it calls the renderer function. This function can be implemented by setting the ViewController to conform to the ARSCNViewDelegate. ARScene adds an anchor (ARAnchor) and a node (SCNNode) for the place where it detected it, and this can be used inside the function to render new nodes or other logic. The code we used to do this is seen in appendix A.3.

Chapter 4

Neural Networks

This chapter will focus on neural networks and how they were utilized for parts of this report. First, some theoretical background surrounding neural networks in general, and then more specifically, convolutional neural networks, is given in sections 4.1 and 4.2. Afterwards, the chapter goes more into how neural networks were tried and used during this project. Data collection and augmentation is described in sections 4.3 and 4.4. Further sections, 4.5 & 4.6, describe how this data was used to train neural networks, with the latter section explaining an alternative to training a model from scratch.

4.1 How neural networks work

There exists many different machine learning techniques out there today. Due to its high effectiveness and relevance, for this report we are going to focus on the highly popular method of artificial neural networks. A variant, convolutional neural networks, is a proven method for working well with images and is therefore highly relevant for this project.

4.1.1 Artificial neuron

An artificial neuron is the simplest form of the neural network. It has a set of inputs and an output. The artificial neuron first sums up all the input values, x, multiplied with the weight value, w. After that it passes that sum through an activation function. This activation function can be everything from a simple $f(x) = x$ to the more complex sigmoid function, depending on the need. More on this in section 4.1.2.

A bias also exists in every node which is not based on any input. The bias function in the artificial neuron is similar to what the m in $y = kx+m$ does. It gives the function the ability to move up and down in the graph for more possibilities of splitting the data set. The bias is usually disregarded when illustrating the artificial neuron.

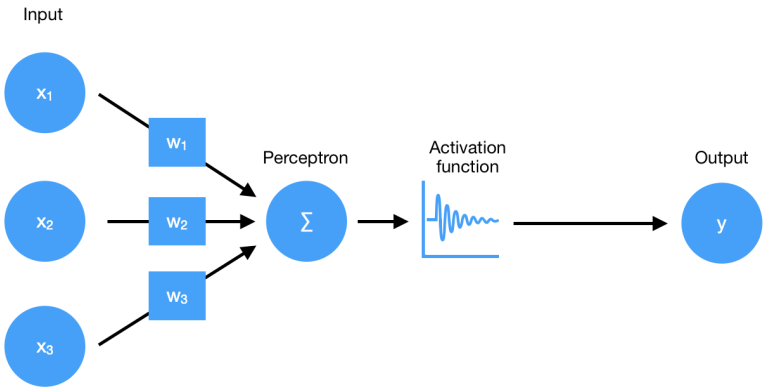

Figure 4.1: An illustration of an artificial neuron, the simplest version of a neural network.

The artificial neuron only has the ability to draw a single line and thus is only able to split simple data sets.

The output of an artificial neuron is described by the following formula:

$$y = b + \sum_{i=1}^{n} x_i \cdot w_i$$

$$y = f(b + \sum_{i=1}^{n} x_i \cdot w_i)$$

where b is the bias value, x is the input, w is the weight for that input, n is the number of inputs and $f(*)$ is the activation function.

4.1.2 Activation functions

The activation function, $\phi(v_i)$, takes the sum of all the inputs from a node as input and passes them through a function before giving an output. This is beneficial when for instance the output should be kept in a range between 0 and 1, or perhaps when negative values don't make sense. Usually, activation functions are attributed to layers instead of individual nodes.

A few commonly used functions are **ReLu**, **Tanh**, **Sigmoid** and **Softmax**.

ReLu is described as

$$f(x) = max(0, x)$$

and is a good option when negative values should be ignored or don't make sense. It is also a good choice avoiding the net to become computationally heavy, since it only outputs the same input value but sets all negative values to zero. That is

usually a good choice in larger networks with many neurons which can become slow.

Tanh is the tangens hyperbolicus function and is used to output values either as -1 or 1 like a binary operator. Unlike a binary operator, tanh's derivative is always defined which makes back propagation possible. Back propagation is described in section 4.1.4

Sigmoid is described as

$$\frac{e^x}{1+e^x}$$

and works like the Tanh function but keeps the values between 0 and 1 and sets y(0) = 0.5 instead.

Softmax is a little bit more complex. It takes a vector v of dimension n and turns it into a vector $\sigma(v)$ of the same dimension where $\sum_{i=1}^{n} \sigma_i(v) = 1$ and each element value is between zero and one.

Each element in the array is described as below

$$\sigma_i(v) = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_k}}$$

This output is good for describing the probability for each element to be correct and is therefore commonly used in the output layer.

4.1.3 Calculating the loss

When training the model, it will at first make a guess to what the right answer or value is based on the random initial weigh values. In the beginning, the model is usually mostly wrong and it is then important to know how wrong it was and in what direction it should go.

For this, the model uses a loss function to calculate that error. For different applications the loss function can be different. One way to calculate the error, E, is to simply take the predicted value, y_p, and subtract it with the correct value, y_c.

$$E = y_p - y_c$$

Sometimes the sign of the error is not important. Then the absolute value can be calculated instead. $E = |y_p - y_c|$

Another approach is the **mean squared error** which squares the difference and takes the mean value over a few predictions. n is the number of predictions.

$$E(n) = \frac{1}{n} \cdot \sum^{n} (y_p - y_c)^2$$

4.1.4 Optimizers

Once the error has been calculated, the network will make changes to itself to improve the performance by decreasing the error value. This is called backpropagation. The principle behind backpropagation is to go backwards in the network from the output node and changing the weight values, ω. The optimizers decide how the weights should change.

A popular method for minimizing the error is to use **Gradient Decent** which works by, step by step, move in the negative direction that minimize the error. The direction is determined by taking the derivative of the error function. The new weight value is calculated accordingly. η is a chosen parameter called the learning rate.

$$\Delta\omega_{ik} = -\eta\frac{\delta E}{\delta\omega_{ik}}$$

where the error function is

$$E = \frac{1}{N}\sum_{i=1}^{N} E(n)$$

where N is the total number of neurons. This leads to

$$\Delta\hat{\omega}_{ik} = \frac{1}{N}\sum_{i=1}^{N} \Delta\omega_{ik}(n)$$

$E(n)$ can be, for example, mean squared error mentioned above. The most common form of gradient decent is called **Stochastic gradient decent**, SGD. The difference is that SGD only evaluates on a small number of nodes/patterns, P and updates all the weights from that observation.

$$\Delta\hat{\omega}_{ik} = \frac{1}{P}\sum_{i=1}^{P} \Delta\omega_{ik}(p)$$

Adam[18], short for Adaptive moment estimation, is another optimizer which is quite popular. The reason is because it has been proven to be a very effective algorithm in many different use cases.

Adam is a kind of combination of using **RMSPROP**[19] and SGD with momentum. Without going into too much detail, SGD with momentum keeps track of which direction the the improvement is in and keeps the improvement going in that same way. The optimisation goes faster when the previous direction is the same and slower when they are different. Almost like a rolling ball that gains speed as it is rolling down hill. RMSPROP does something similar and uses a running average of each weight value and the previous values importance can be controlled

with a parameter. It also only uses the sign of the direction and not its value. Also RMSPROP has individual weights.

Adam keeps a running average of both the past gradients and the squared past gradients. The full mathematical description of Adam is given by the formula below where t is a time iteration index.

$$\omega_i(t+1) = \omega_i(t) - \eta \frac{m_i}{\sqrt{v_i} + \epsilon}$$

where

$$m_i(t+1) = \beta_1 m_i(t) + (1 - \beta_1) \frac{\delta E(t)}{\delta \omega_i}$$

$$v_i(t+1) = \beta_2 v_i(t) + (1 - \beta_2) (\frac{\delta E(t)}{\delta \omega_i})^2$$

and η, β_1 and β_2 are adjustable parameters and ϵ a small value to keep the equation from dividing by zero.

There are a lot of other optimizers such as **Adagrad**, **Adadelta**, **Nadam** [20]. They all have their own advantage and use-cases. We won't go into further details about them.

4.1.5 Layers

With more layers and more neurons the networks parameters and complexity begins to grow. So does also the training time, size of the model and the cost for doing predictions. Thus, these networks are capable of describing much more complex data sets. But as a result of that, the risk of overfitting, the act of describing the training data set too well so that new data sets will not get recognized, becomes much greater. That is why a complex network is not always wanted. In figure 4.2 an example is given of how a network with many layers might look.

4.1.6 Overfitting

When training a neural net, one needs to be careful not to train the model too much or overfitting is likely to happen. Overfitting is when a model gets really good at predicting the data that it is training on but fails to predict accurately on new data. This is because it starts to pick up too much on detail or in some instances even noise. For this reason it fails to pick up the general trends, which is more valuable. An illustration is given i figure 4.3.

There are a few methods that can help handle the problem of overfitting. They usually involve trying to limit the size of a small amount of weights. The hypothesis behind this is that if a weight is much larger than the rest it also has much more

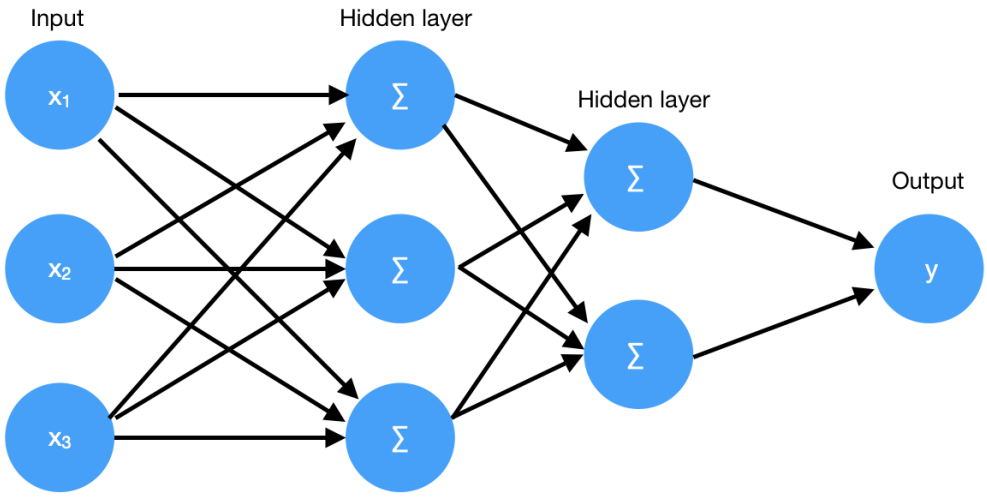

Figure 4.2: Two fully connected layers. One with 3 neurons and one with 2 neurons.

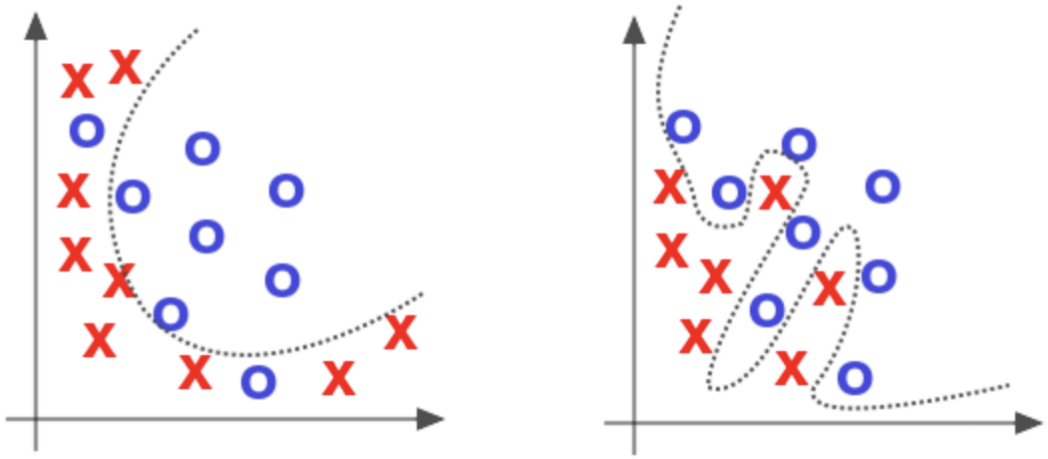

Figure 4.3: Overfitting of a dataset. On the left is a generalized trained model and on the right is an overtrained model. Image taken from OReilly.com [21].

influence over the final prediction. That way, a small detail in the data can have much more influence than the general trend.

One way to do this is to cut random connections between layers during epochs, usually by specifying a certain amount that is going to be cut. This way, the model is not relying on a small number of nodes to make the correct predictions. This method is called **dropout** [22].

Another way is to introduce random noise on a layer during training. This works because if a node with a large weight receives noise, it will be heavily amplified and probably give a false prediction. When doing this, Gaussian noise is usually implemented, which is basically random noise with a gaussian distribution.

To force the model to keep weights small, one way is to add a penalty to the loss for every weight based on its size. This is called weight regularisation and is widely used and exists in two forms, L1 and L2. L1 simply adds the weights size multiplied with an L1 term that is chosen by the designer. The other, L2, is to do the same but in this case square that value. This has the effect of making values over 1 even bigger and values less than 1 smaller. In that way, it doesn't affect the loss as much as L1 when not being overtrained. L2 is also called weight decay and is the more common method of the two.

Another way to keep the values small is to normalize the input to every layer and set the mean to 0 and variance to 1. In Tensorflow, this can be done with the BatchNormalization layer [23].

One important thing to point out is that all these methods mentioned so far are only active during training and is not doing anything when making real predictions.

If lots of training data exists, the ensemble technique could be a good way to go. This technique divides the training data into smaller sets and trains a model for every data set. The final prediction is then an average of the predictions of all the models. This technique works because if the model is highly overtrained in a certain direction, chances are that the other networks will drown out this result by the many other models.

It is kind of like when someone has an off pitch in a big choir. If there are many other singers, this off-pitch will not be noticed very much.

The final technique is called early stopping and is based on validating the model after every epoch and stopping the training when the validation loss, or some other criteria, is not improving anymore. A graph showing when to do early stopping is given in figure 4.4.

4.2 Convolutional Neural Networks

Convolutional neural networks, CNN, have been used for quite some time when it comes to deep learning and their capabilities are rather astounding, as proved by Krizhevsky et al [9]. They use layers which perform convolutions, hence it's name. The input to a convolutional network are either 2D or 3D tensors, where the 3D alternative usually has color channels along the third dimension. Use of convolutional neural networks in image recognition can be practical for several

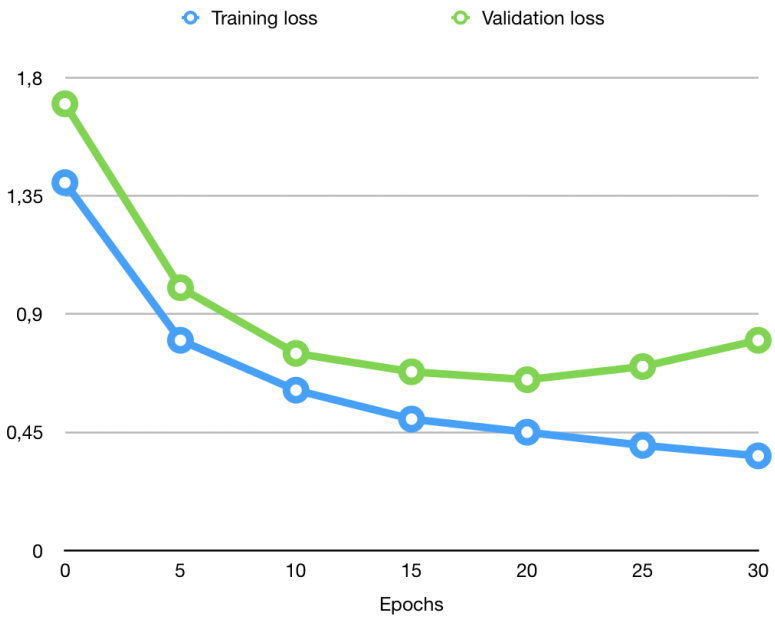

Figure 4.4: A graph showing when to do an early stop. The graph shows the total loss over trained epochs. Notice that the validation starts to increase again somewhere after epoch 20.

reasons, e.g., it can tell of spatial relationships in an image. This section has been written with reference to V. Dumolin et al [24]. Figure 4.5 shows an example of how a CNN may look.

A convolutional layer performs the mathematical operation *convolution* on the input tensor, which can be presented as an image. A convolutional filter, or kernel, of size $k \cdot l$ slides across the input. The kernel consist of weights in each element. These weights are found during the training process of the network. As the kernel is slides across the image, the dot product of the weights of the kernel and the pixels the the kernel is above, is calculated as the output. Afterwards, a new image containing all of the produced dot products is formed.

Padding on the input can be used to account for values when the kernel is outside the input. Use of padding have an impact on the output size after the convolutional layer. Stride can also be used, which tells how much the kernel translates along an axis; increased stride leads to subsampling. Since changes of parameters in one axis do not affect outcome in another axis it simplifies explaining CNNs by having the parameter values being the same along both axes. The more convolutional layers used in a network, the more complex shapes are detected. First layers may detect edges and corners, whereas, later layers may find features representing, for example, a car or a dog. A common addition to the layers is an activation function, commonly ReLU (which is described in section 4.1.2). This gives faster training by only allowing for positive weights to pass the layer.

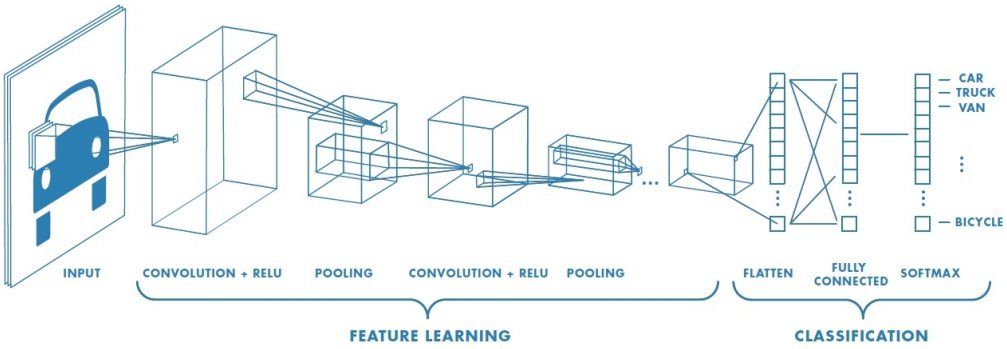

Figure 4.5: Image shows the flow of a convolutional neural network from left to right. Several convolutions as well as pooling functions are done in order on the input image. The last feature map is then flattened to fit into the hidden layers, to then be classified with a softmax function. Image taken from [25].

The common types of padding is *no zero*, *same* and *full*; examples seen in figure 4.6. No Zero padding involves having no padding outside of the input. This means that the kernel never goes outside of the actual image, once a side of the kernel hits the side of the input it jumps down to the next line (if stride is 1).

Same padding is used when one desires the output size of the layer to be the same as the input size. This is achieved by having a padding p be $p = \left\lfloor \frac{k}{2} \right\rfloor$ for any odd kernel size k.

With full padding one actually makes the output size larger than the input, by fully utilizing every possible combination of the kernel and the input image. This is accomplished by having the padding p be $p = k - 1$ for any kernel size k.

Another useful feature used in a lot of convolutional neural networks is pooling. Pooling layers are somewhat similar to convolutional layers in that they use a sliding window which performs an operation on the contents of the window and outputs a new value. However, the difference is that they use other functions instead of linear addition. Two common pooling functions are *max pooling*, where the output is the largest value within the window, and *average pooling*, where the output is the average of the components within the window. Pooling is commonly done do reduce the size of the input. Figure 4.7 shows example of pooling being used.

When a classification is to be done in the CNN, the shape needs to be shifted to an array. This is done after the last pooling layer or convolutional layer. This array can then be passed into a hidden layer for classification.

4.3 Collecting data

When working with machine learning, big sets of data is often required for a good resulting model. A problem with this is that it can be tricky to obtain such a large data set because it usually also requires labels with that data to be manually put

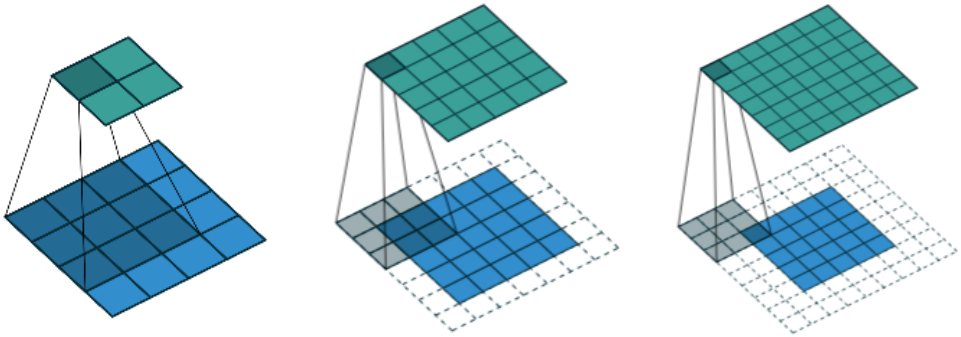

Figure 4.6: Different examples of padding. Grid in green is the output, grid in blue is the input, and area in shadow is the current area where the kernel is currently at. Leftmost image shows a no zero padding being used. Here the kernel size is $k = 3$, input size is $i = 4$ and the output size is $o = 2$, i.e., smaller than input. In the middle image same padding is used; output size is the same as input size $o = i = 5$. In the rightmost image full padding is used. Here a bigger output size than the input size is produced, $o = 7, i = 5$. Images taken from [24]

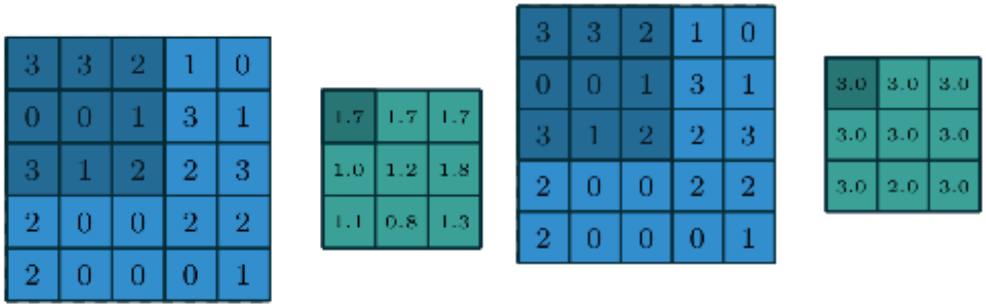

Figure 4.7: Examples of two types of pooling. Blue grid represents input, green grid represents the output, and the shaded area is where the sliding window is located. The left image shows the use of average pooling, while the right one shows the use of max pooling. Images taken from [24]

in. These labels will be used by the neural network to, during training, check if the predicted value is correct or not.

In this project the data is sets of images of different furniture parts. These images did not exist anywhere online, so they had to be collected by taking lots of photos. The photos contained the part in different background and from different angles. Table 4.1 shows how many images of each class we had. The backgrounds were mostly of typical office surroundings, although there were exceptions (example is shown in figure 4.10). All the photos were then resized to 256x256 pixels. The reason for choosing a square size is because when rotating them (this will be useful later on when augmenting data in section 4.4) the images will not have any black bars on the sides nor be stretched.

Class	Number of images
Leg piece	203
Bridge piece	210
Seat	216

Table 4.1: Table showing how many images of each class we had.

For the first furniture Nolmyra, there were only 3 unique parts (excluding screws and similar parts). Therefore the model was trained to recognize 4 things; the different parts and an unknown object. The unknown label was because the object detection algorithm could detect other objects and it is then unwanted that those objects be mistaken for furniture parts. This could of course also be done by setting a threshold on the confidence value, i.e., if the model gives a confidence value under 0.8 it discards it as being an unknown object. The problem with this type is that it is much harder to distinguish parts from unknown objects. This is due to that the best model is the one that can give a close to 100% confidence value on every prediction.

For this reason, photos of random objects were also added to the data set. Most of these photos were taken by ourselves, but some were also collected from the web. The images were placed in folders with a specific item which meant labelling them became much simpler. Just put all the images containing a certain part in the same folder.

Training a model usually requires hundreds of thousands or even millions of photos. That much data is hard to get and would take a lot of time to obtain Going around the office to snap that many photos is almost unthinkable. However, there are other options which will be explained in the next section.

4.4 Augmenting data

Instead of training on only original images, some images can be created from other images by for example rotating, flipping, changing brightness, saturation, contrast etc. This will essentially be an image of the same object, but the data will look

Figure 4.8: Images showing different samples of training data used for training the model. From left to right: seat, bridge piece, leg piece.

different, thus giving the model more relevant data to train on, see figure 4.9. When doing this it is important to keep in mind that the augmented data should be relevant to real situations. Creating data with only the blue color band when the real situations are only in daylight makes no sense.

Furthermore, images of objects of interest can be cut out and pasted into random environments to create even more data, see figure 4.10. The idea of this is to try and make the model understand that the focus should be on the furniture part and the background environment is irrelevant. While doing this, even though the parts could be cut into totally random environments we tried to focus on pasting them into relevant spaces like office floors or carpets. In our project, this gave a good result as it increased the test accuracy by 8%.

Doing this by hand can still be very time consuming, so automating this process as far as possible is recommended.

4.5 Image classification

Designing a neural network for the image classification is not the easiest task and much work consists of trying things and making intelligent guesses. As stated in the section about data collection we had 4 different classes to classify, which means that a test accuracy of over 25% would be significant. We were aiming for somewhere above or around 90% with a relatively small amount of data.

We started out with many convolutional layers layered with pooling layers and a few dense layers at the end. The final layer had four nodes and softmax activation functions to give a classification probability between four classes. The activation functions of the other layers were set to ReLu since they are the fastest and preferred for larger nets. Tanh was also tried but with no success.

SGD with momentum as an optimizer was tested but was changed to Adam quite early on since it had better performance. Only linear optimizers were considered because non-linear optimizers take a lot of computing power and are usually only recommended for small networks.

Figure 4.9: Images showing some resulting images from augmentation. Top left image shows the original image. Top middle image shows the result after rotating 180 degrees. Top right image shows result after rotating −90 degrees. Bottom left image show result after reducing the contrast with by 30%. Bottom middle image shows result after reducing color balance by 70%. Bottom right image shows result after increasing color balance by 60%.

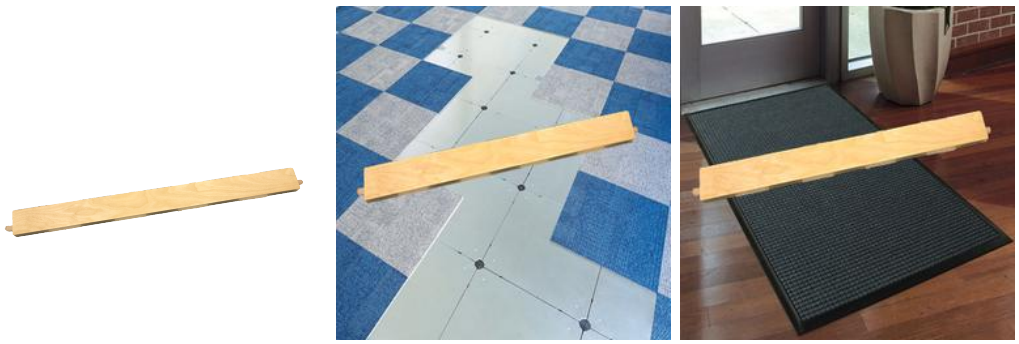

Figure 4.10: Images showing how cutouts of images were utilized. Leftmost image shows a cutout of one of the furniture parts. The other two images show the result after they were pasted in to different environments to artificially increase training data.

We went back and forth with trying different mixes of kernel sizes, amounts of convolutional layers, insertion of pooling layers on different places, stride sizes, amount of dense layers and how many nodes they had.

When training the models, overfitting was a big problem from the start. The tested methods for combating this was dropouts with different values, L2 weight regularisation, batch normalization and reducing the number of weights.

It was hard to reduce the number of weights as the evaluation usually converged around 25% whenever it was tried. When using the dropout method a value of 50% with a minimum of 64 nodes in the layer reduced overfitting most effectively. That, with a combination of weight regularisation and batch normalization gave good results. Figure 4.12 shows how overfitting was avoided.

Gaussian Noise gave very good results, but was unfortunately not possible to be converted from the model format generated in keras, to the mlmodel format that is required for Xcode.

The most important thing to mention is the difference the amount of photos we had and how they were taken. Adding more photos made a huge difference to the test evaluation score. After having added 50 more images per class with a library of 600 images the test score increased from **80.55%** to **83.45%** in accuracy.

When finally testing the model we made a mistake by not separating the training data and test data. Since we shuffled our data before splitting it up into two parts, some augmented data got into the test set. The model had already trained on the original image and it was similar enough to give unrealistic good results of 99.52%. This was finally fixed by splitting up the test and train data into separate folders. Figure 4.11 shows the printed graph from one of those training sessions.

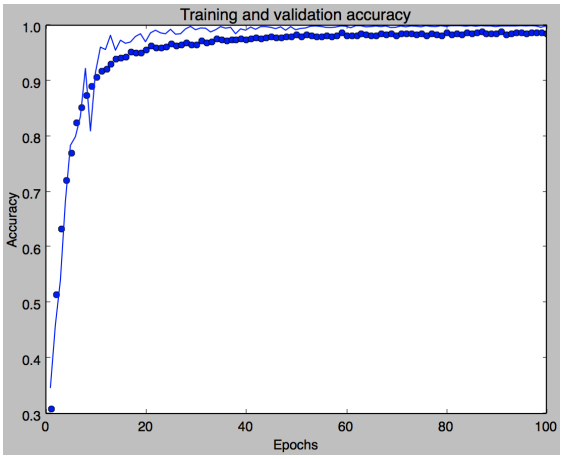

Figure 4.11: Test and train accuracy over a 100 epochs. The performance seems much better than it actually is.

As shown in the code in appendix B.1., early stopping was also finally implemented with a callback to save the highest performing model at the end of the

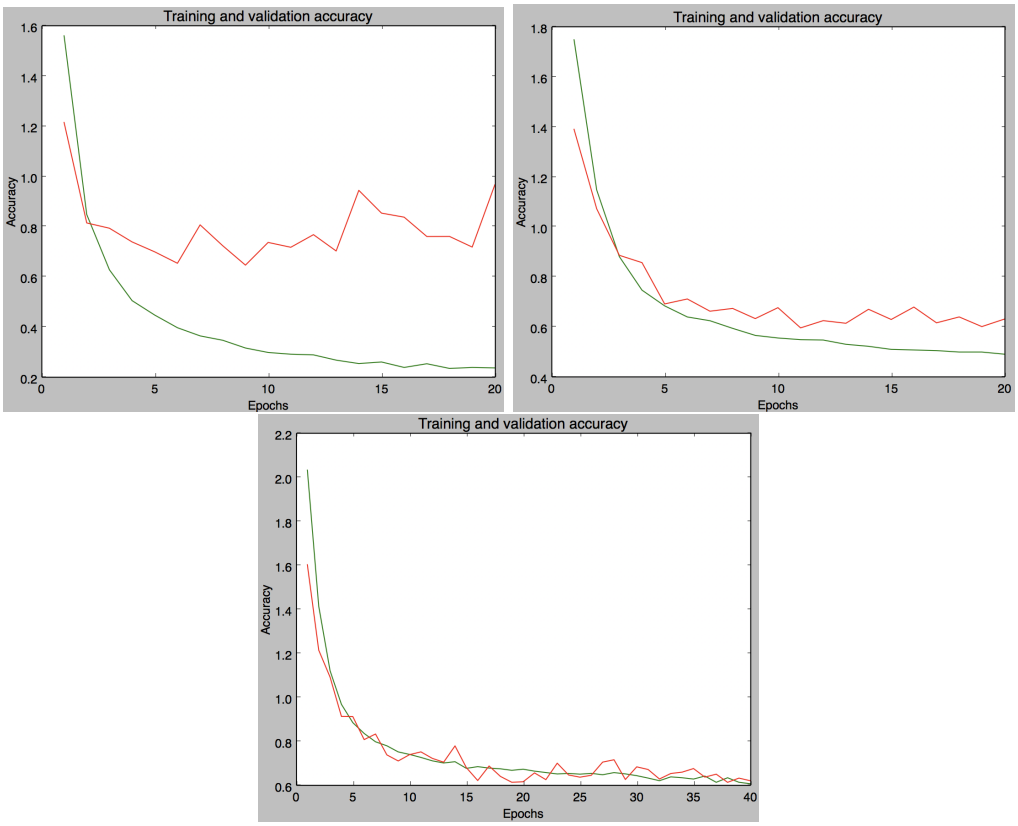

Figure 4.12: Images shows the result from introducing weight regularisation and dropout to a model. The graphs show the training loss (green) and evaluation loss (red). The upper left image shows the training without any dropout or weight regularisation. The upper right image shows the training with dropout implemented. The bottom image shows the training with dropout and weight regularisation implemented.

process.

The network that proved to be the best performing network (**92.25%** accuracy) had the configuration shown in appendix B.2.

A model with many convolutional layers and a few dense layers at the end gave the best results. A larger network could have been implemented and tests as well but since the model size would grow more and training time would increase drastically, we didn't increase it further.

4.6 Transfer Learning

After trial of designing the network from scratch, it was decided to try and make use of pre-trained networks and then retrain them for another purpose. Models trained on imageNet [26] were chosen since that source domain is similar to this target domain. The models with their respective weights were loaded, without their fully connected layer. The top layer's weights were then frozen at different stages, and new fully connected layers were added on top, and new classifiers were trained. Several different models were tried, including ResNet50, InceptionV3 and VGG16, which where all integrated in Keras to start with.

The code shown in appendix B.3. shows how to load the VGG16 network which has weights that has been pretrained on imageNet. The fully connected layer is then removed and instead a custom top layer is added to be trained.

The best result that was achieved for each of the models using this method is listed in table 4.2. It shows that using InceptionV3 model with pretrained weights gave the best result. However, a few factors, such as, where the models were frozen, how adding and removing layers, has most likely affected the results. As can be seen from using ResNet, the best model achieved 25% accuracy, which is the same as making a guess, seeing as there are only four possible classes. These models could, and would most like have been, improved if it was not decided to take a new approach to the problem, which is described further in section 5.4.

Pretrained model used	Best achieved accuracy
VGG16	70.8%.
ResNet	25.0%
InceptionV3	72.9%

Table 4.2: Results from doing transfer learning .

Chapter 5

Object Detection

At this point of the project process, a machine learning model capable of decently classifying which of the furniture parts were in the image existed. However, when using the planned application, it is very likely that there are several parts within the same frame. Hence, a method of localizing where an object was located was highly sought after. The idea was to find such a method, segmenting out the areas containing an object, and then use these areas as input images for to the machine learning model.

In this chapter, a few of the methods for object detection that were tried is explained, even though none of them ended up being viable in the end.

5.1 Testing Object Scanning with ARKit2

Apple's ARKit has a feature where it can detect scanned 3D objects. For scanning, they have developed an app that can be downloaded from their website [27]. After the scanning is complete the app lets you export the model to then include it in your ARKit project. Once in the project it is simply imported into the ARWorldTrackingConfiguration in a way show in appendix C.1. (The models are kept inside the Assets.xcassets catalogue in a folder called 'Objects').

Once imported into our project we were able to test the performance of recognizing and tracking three pieces of our furniture. Sadly, this method came up short for our purpose. When testing live in our app the time for detection were much longer (over 1 second) which made tracking them difficult. The tracking wasn't smooth but rather choppy. Many times, the object wasn't detected at all. This was mainly due to our objects being a little big to fit the screen while being close with the camera but also that the object had a lot of empty space within its bounding box.

The detection worked best while being in the same environment, static with the same kind of lightning conditions, but fell short when the object was moving or being held. Therefore, since the user is going to hold the pieces by hand and moving them around, this method cannot be used for our purpose.

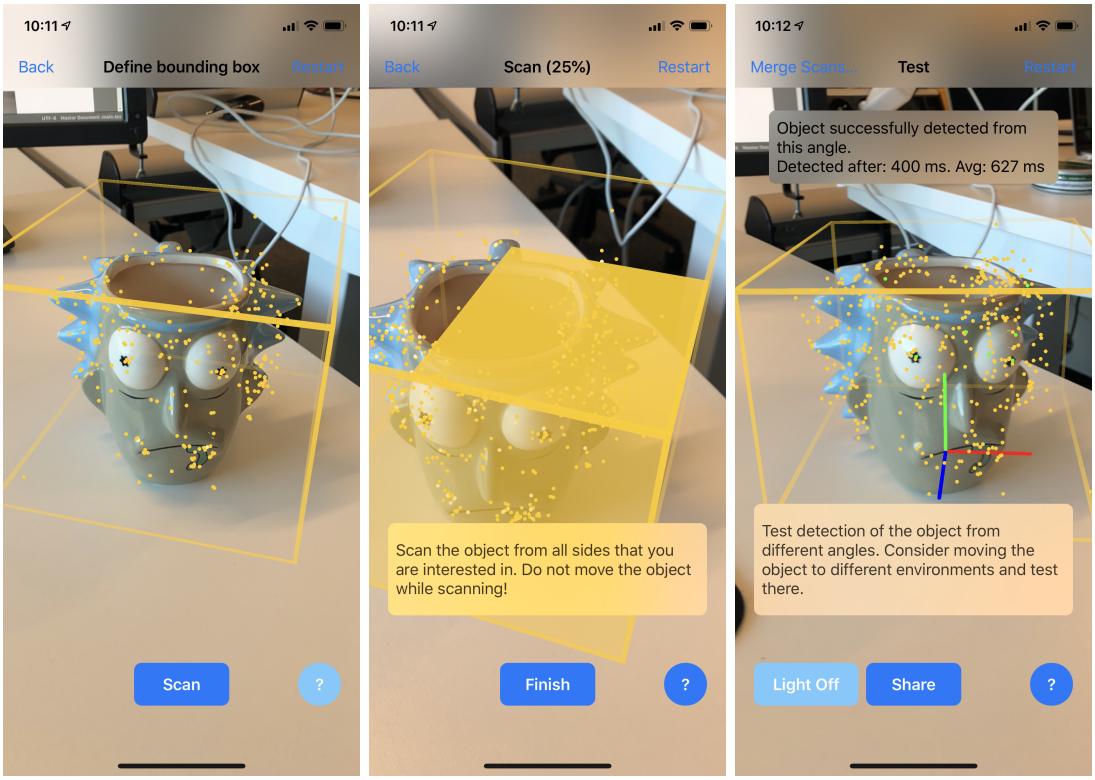

Figure 5.1: 3D Scanning using Apple's app. On the left, the bounding box is defined so that no reference points from other objects are included. In the middle, the object is scanned by aiming the camera around the object at all angles. On the right the created model is tested. In this case, the object is detected after 0.4 seconds.

5.2 Object Detection with traditional machine learning

When trying to detect objects in a still image we have looked into two main methods. One typical way is to try and look for patterns or features in the image. Either a specific image can be matched within the larger image or a series of features can be found. An example of the latter is Haar features which is used in the Viola-Jones for face detection[28].

Using the image integral (which is the summation of pixel values in a specific region) different features can be obtained.

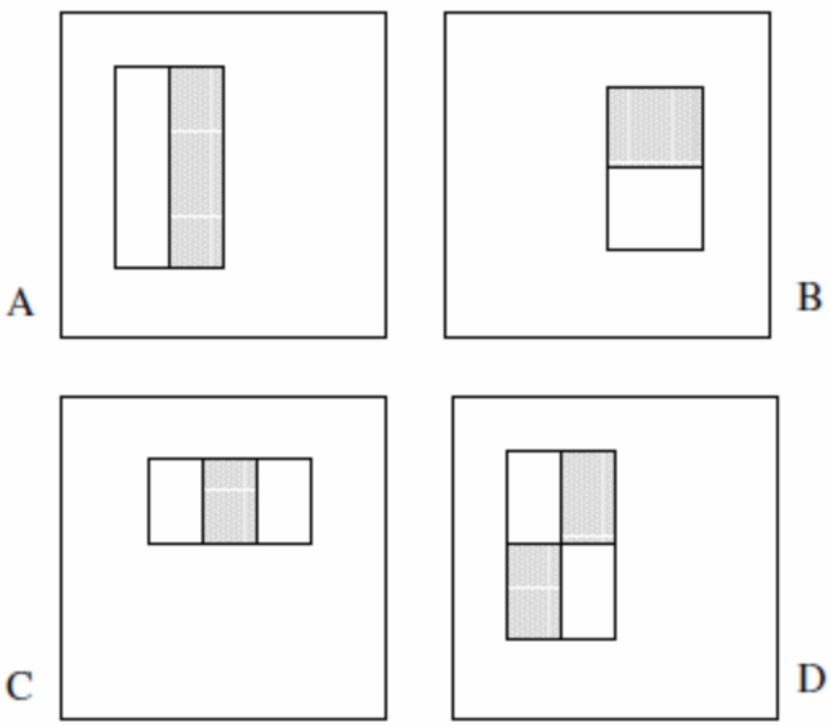

Figure 5.2: A set of Haar features used to detect faces in the Viola-Jones method. The pixels in the white regions are summed and subtracted with the pixels in the black region. The algorithm will later decide if a specific feature has been found or not, depending on obtained value.

This method is based on the fact that every face shares some basic similarities. Even objects of the same type can share some similarities.

Another way to find objects in an image is to try to classify each pixel as either background or foreground, usually referred to as Foreground/Background-segmentation. This usually requires some background knowledge of how the background usually looks like, for example, if the background is grass or a concrete floor.

One of the most basic functions for separating background and foreground is the flood fill method. Anyone that has used Paint for Windows knows exactly what this one is. It fills a segment of similar pixels with the same color. This method works best on one backgrounds with only one color. Online there are many different variations but they all accomplish the same goal.[29]

One possible way of detecting objects in an image is to use depth data, or RGB-D which is depth data embedded in an RGB photo. A popular device that uses depth data for object detection is the Kinect camera for Xbox.

On the iPhone X, depth data can be captured by either True Depth on the front camera or with the dual cameras on the back. The True Depth camera works by having a dot projector emit light dots (mainly on a face which is why this feature is found on the front camera) and picking those dots up with an infrared camera.

The dual camera on the back works by taking two photos and comparing those to find the pixel shifts of the same objects. The distance is calculated by

$$\frac{pixelShift}{pixelFocalLength \cdot baselineInMeters}$$

and gives the unit in $1/meter$. It is the same principal to how we humans see distance by having two eyes pointing the same direction. [30]

However, obtaining the depth data from the dual cameras in real time is not possible since it requires too much computational power. This unfortunately makes it impossible to use in an ARScene and thus not possible for this project.

Despite all the available methods above, object detection without machine learning is still very tricky. These methods work best when the images are in an controlled environment, typically industrial, like finding screws on a white background (as they do in an article posted by combine.se). [31] When the environment is a more casual place though, e.g., recognizing furniture indoors, the task becomes much more difficult. For this reason, object detection with pure algorithms is not very common in household applications. Instead object detection with machine learning methods such as R-CNN's (Regional-Convolutional Neural Network) are much more common nowadays.

5.3 MATLAB prototype for Object Detection

A purely feature based object segmentation method was implemented as a prototype in MATLAB. This was done to see if a machine learning model for detecting objects could be dismissed. The idea was that if one sent an image into the system, the resulting output would be bounding box coordinates for each respective object within the image. These smaller sub-images would later be separately classified using a deep neural networks. The bounding boxes would also be useful for user interface in the application.

First, an edge detection method was run on the entire image, to find the edges, which would serve as a good variable for the segmentation. Then, Bradley's

threshold algorithm [32] would binarize the image. Bradley's method is locally adaptive and computes the threshold value for each pixel by looking at the mean intensity of a neighborhood of pixels surrounding it. From this, one can find enclosed blobs, and by finding enclosed blobs containing more than a certain amount of pixels, to remove noisy errors, one can find the objects. The bounding boxes are then found by selecting the minimal and maximal height and weight values of the blobs.

5.4 Results from doing object detection in MATLAB

A MATLAB script was tried in order to find possible ways of segmenting out regions containing objects in an image. The output segments would then each be classified separately. This way we would have both object locations and classifications on each object. Figure 5.3 shows an example of when this method worked. However, results from this were highly unpredictable and the parameters were too dependent on lighting, shadows, the background among other features. Figure 5.4 shows an example of when applying the same method on a different input image. Because of these unreliable results, this method was scrapped.

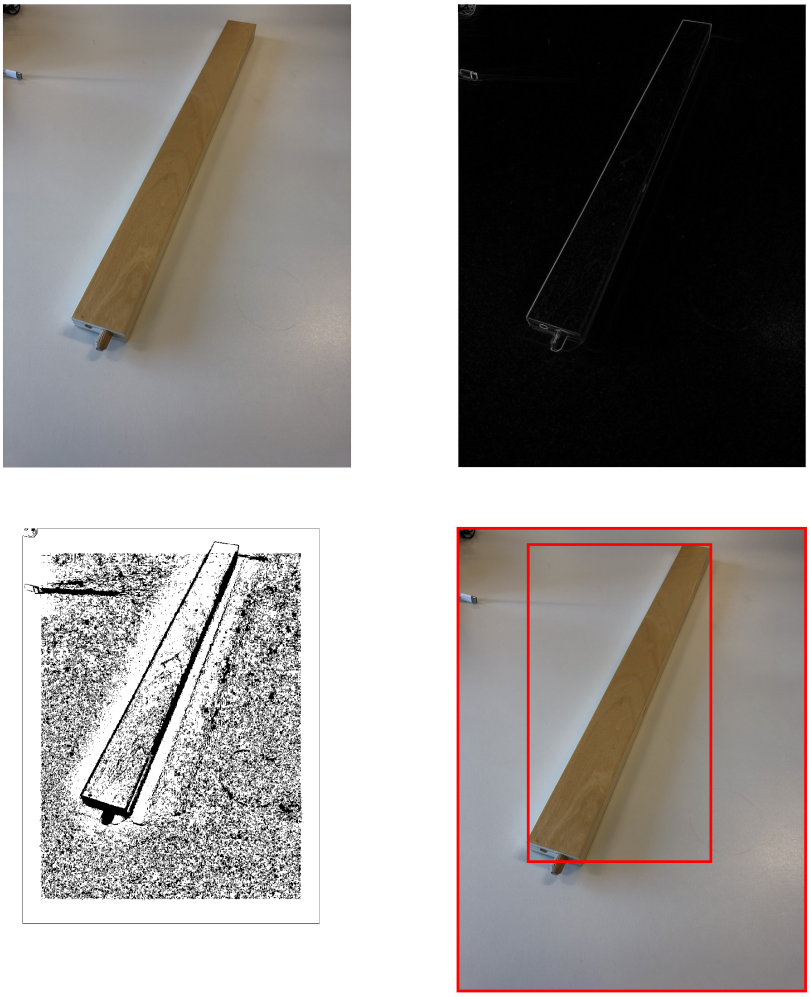

Figure 5.3: Images showing the results from doing object detection in MATLAB. Top left image shows the original image. The top right image shows the result from running edge detection on the original image. Bottom left image shows the the result after binarizing the top right image. the bottom right image shows the generated bounding box superimposed on the original image

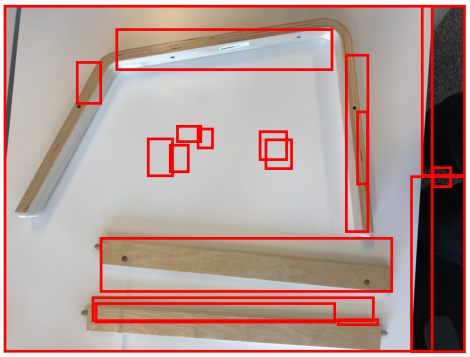

Figure 5.4: Example image of when this method did not work as well.

Chapter 6

One Stage Detector

After getting stuck for trying to develop a solution that would first do object detection, and then afterward classify the segmented areas separately, it was decided to try a different approach. The one stage detector *You Only Look Once* method was tried to great success. This chapter will describe how it works, sections 6.1 6.2, how the models were created and trained, section 6.3, and then an evaluation of the resulting models in section 6.4.

There are other methods of doing object detection and classification, such as *Mask R-CNN* and *Faster R-CNN*, however, they are outside of the scope of this project.

6.1 YOLO

YOLO, or You Only Look Once, is a one stage detector approach to object detection and recognition [10]. It takes an image and predicts both bounding boxes and the probabilities of the classes being within these bounding boxes in one run, hence its name. It was designed to be fast and usable in real-time scenarios. Since YOLO sees the entire image during training and testing, it receives contextual information about the classes and reduces error with matching background patches for objects.

The architecture for *YOLO* consist mainly as a convolutional neural network, with 24 convolutional layers and two fully connected layers. There was also a smaller neural network trained called *Fast YOLO* trained which were only 9 layers which was designed to create an even faster system for object detection.

The system first divides the input image into an $S \times S$ grid, where each cell predicts B amount of bounding boxes respectively confidence scores for the boxes. Each bounding box prediction consists of 5 separate predictions: the x, y coordinates, represented as their position relative to the the grid cell, width and height relative to the entire image and a confidence value. The confidence values are there to reflect how certain the model is that there exists an object within the box, i.e. ideally, confidence should be zero when no object, and the intersection over union between ground truth and the predicted box if there is. Each grid cell also predicts C probabilities for each class, conditioned that there's an object

within the boxes.

Intersection over union, also known as the Jaccard index, is a way of measuring similarity between sets. It can be written as

$$IOU(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}, \quad 0 \leq IOU(A,B) \leq 1$$

if A, B are two finite sets. If the two sets are equal, then

$$IOU(A,A) = \frac{|A \cap A|}{|A \cup A|} = \frac{|A \cap A|}{|A| + |A| - |A \cap A|} = \frac{|A|}{|A|} = 1$$

When testing, these scores are then combined to give the class specific confidence values for each of the boxes, thus it receives both the confidence of the class being in the box, and how well the box fits the object. Figure 6.1 summarizes the flow in YOLO.

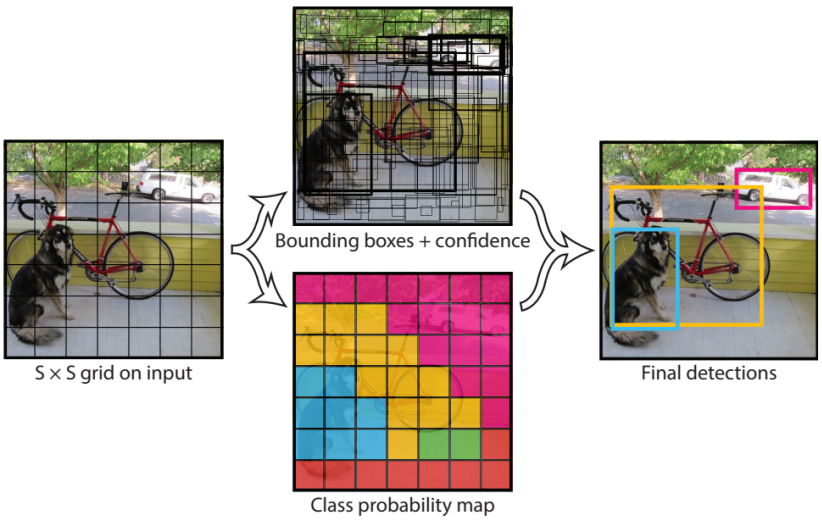

Figure 6.1: The stages of YOLO. First dives into $S \times S$ grid. Separately predicts bounding boxes with respective confidence, as well as class probability. Then, combines the two to form the final predictions. Image taken from [10].

Comparisons done by the team working on *YOLO* found that, compared to other real-time systems at the time, both *Fast YOLO* and *YOLO* outperformed them all with *Fast YOLO* being the fastest, but *YOLO* being more accurate on the *PASCAL VOC* data sets [35].

Throughout the years Redmon et al. has been working on updating the design pattern of the YOLO network. First in 2016 when they introduced *YOLOv2* and then April 2018 with *YOLOv3* [33] [34]. *YOLOv2* added a few concepts to the system to make it even more fast and accurate than the earlier iteration. It removed the fully connected layers and it was now possible to train on several different input

image resolutions and it could now also predict many more bounding boxes than its predecessor. *YOLOv3* added some changes which improved its ability to detect small objects, with the trade off of having a bit worse performance when finding larger sized objects.

6.2 Mean Average Precision

There is quite a difference when it comes to evaluating a model that does object detection, compared to normal image classification. When doing the latter, accuracy tells you how often the model makes the correct prediction. However, with object detection it is done differently. The common metric is *mean Average Precision* (mAP). This metric makes use of the previously discussed *IoU* (Intersection over Union). The IoU and the classification determines if the prediction is deemed correct. The classification obviously has to be the same as the ground truth, and the IoU has to be over a set threshold, these are known as true positives. A predicted box that does not pass the threshold is known as a false positive. If there are more than one prediction that passes the threshold, only one will get seen as a true positive, and the other as false positive. When a prediction was not made for a ground truth box, it is called a false negative [38].

From the false positives, true positives, and the false negatives, one can calculate recall as well as precision scores [36]. Precision is how good the model is to identify only the relevant data, and can be written as

$$\texttt{precision} = \frac{\texttt{True Positives}}{\texttt{True Positives + False positives}}$$

Recall is how good the model is to find all the relevant data, and is written as

$$\texttt{recall} = \frac{\texttt{True Positives}}{\texttt{True Positives +False Negatives}}$$

When a classification is made you also get a confidence value. One only calls it a prediction if the predictions confidence passes a set threshold. However, if one calculates the recall and precision for all possible thresholds, a Recall x Precision curve can be made. From this curve, one gets the mAP by taking the area under the curve.

There are different standards when calculating the mAP. The PASCAL VOC Challenge uses a mean average precision with a static threshold of 50% [35]. This is commonly called *mean_average_precision_50*. Another common standard is the one used by COCO [37]. This one calculates the mAP at IoU threshold from 50% to 95%, with increments of 5% every iteration, and averages them all together. This is quite strangely known as just *mean_average_precision*. The latter mentioned method is the one this report will use for evaluation, since it puts more value on localization than the PASCAL VOC method.

6.3 Turi Create

Apple has a licenced toolkit for development of custom machine learning models called Turi Create [39]. This was created to help developers easily implement their own ideas into an app. It includes a methods object detection.

In this project, the object detection method included in Turi Create was tested. First, one has to create ground truth data for every image that was trained and tested on. This was done using Simple Image Annotator [40], where one has to draw bounding boxes for the objects in the images and label them; the data output is in the *.csv* format. Turi Create uses a different format for annotations called *.sframe*, however, they provide a simple Python script which automatically does the conversion.

Turi then trains a model using a re-implementation of the TinyYOLO network. It also utilizes transfer learning by starting with an image classifier that has been trained on the imageNet dataset and then does *end-to-end finetuning* to change the network to a *one-stage detector*.

Several models were trained, with different amount of training data, to evaluate how many were needed to get a decent result. The Turi website states that at least 30 samples of each class is needed to generate a good enough model. Hence, we tested for even lower samples, as well as much more. Lowest amount of training images were around 50, and highest were around 1000. The *mean average precision* was measured for each model using 200 testing images, and then plotted to create a graph showing how it changed, depending on the amount of training data was used. The results can be seen in section 6.4. The best performing model was then to be used in the application.

The code in appendix D.1. shows how the model was created using Turi in python.

The results could also be further inspected by drawing the newly predicted bounding boxes on top of the original image, from this one can visualize how the model actually performed. This was done using code in appendix D.2. Some of the resulting images from different models can be seen in section 6.4.

6.3.1 Importing model to application

After training and evaluating that the model lives up to a standard worth including in this project, it was converted to the *.mlmodel* format used in Swift. When inputting a frame from the application into the model, the model outputs a multi array containing an array for each *object* found. These predictions have not been *non-max suppressed*, since that functionality is lost during the conversion from the model used in python, to the .mlmodel format. Thus, it has to be reimplemented in the application itself. A non-max supression threshold of 0.5 was used as it is considered a traditional standard [41]. The code in appendix D.3. shows how non-max suppression was implemented.

6.4 Results from doing object detection with Turi

Because of the vast amount of possible use cases it was decided to scale the objective down. The model was trained only on a few floor backgrounds.

The mean average precision from the different models that were trained were observed and can be read in table 6.1. These values were plotted, resulting in the graph shown in figure 6.2.

Amount of training images	Percentage of total amount	mean_average_precision
≈ 50	5%	0.17064
≈ 100	10%	0.30188
≈ 140	15%	0.30342
≈ 185	20%	0.39269
≈ 280	30%	0.41588
≈ 370	40%	0.40595
≈ 415	45%	0.49917
≈ 450	50%	0.47397
≈ 570	60%	0.50454
≈ 700	75%	0.54433
926	100%	0.57618

Table 6.1: Mean average precision depending on the amount of training data used.

When training on different amount of data it can appear as if more data gives a worse result sometimes. That can be explained in two ways. The first explanation is that when training the models the data was randomly split from the full training data set. That means that sometimes there can be more "good" data and sometimes less in the set. Some images have lots of objects in them and others have fewer. Thus, if you have a data set with images with lots of objects in them it will likely give better results. An important thing to mention however is that for the model to perform well it also needs areas in the image of no object.

The other explanation is that when training, the most optimal model for that data set is not always, if ever, returned. Take into account early stopping for example. Sometimes the early stopping can happen on a good place and sometimes it could have given a better result if the training session would have kept going for a little bit. We did not have a way to use an evaluation set during training, and as such, could not receive training and validation curves. Hence, we can not know how if we trained for too long or too little.

The curve from the tests, fig. 6.2, shows no sign of saturation yet, thus even more data would surely improve the performance. The reason for not testing with more data is because of the difficulty of gathering more data (taking photos in different environments and adding ground truth data) and the time it took for training. As of now, the time for training with 926 images on a MacBook Pro 15 inch 2017 with a 2,8 GHz Intel Core i7 processor is over 24 hours. With half the amount it takes about 17 hours. We would have trained with a GPU, however, it

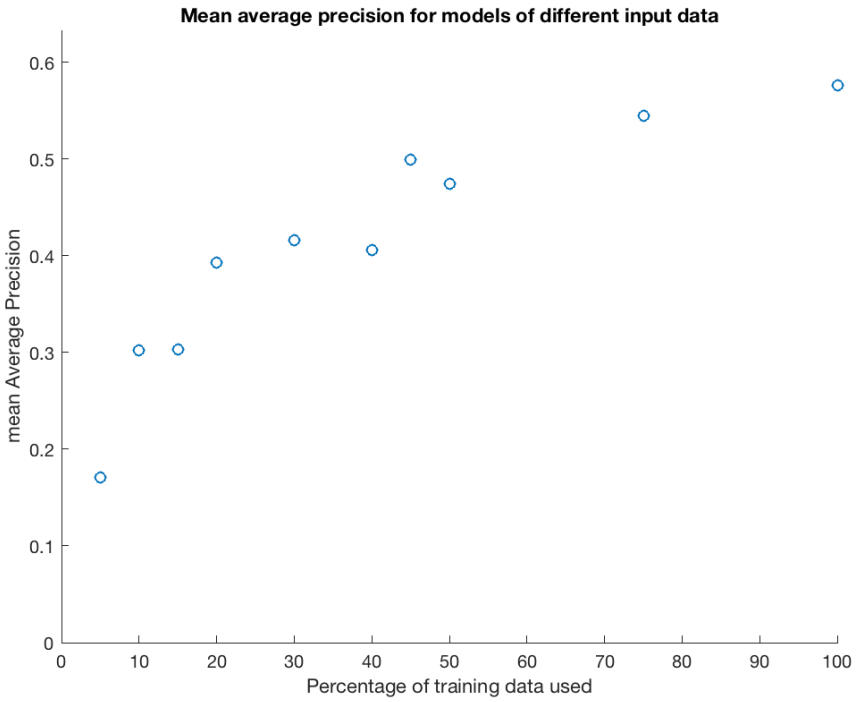

Figure 6.2: Mean Average Precision from table 6.1 plotted against the amount of training data used. 100% corresponds to 926 images.

was at the time of training not supported on our hardware.

From the graph there is a small plateau between 185 to 370 images. Around that performance (0.4 mAP) is a sort of minimum for the model to perform well. For our 6 classes that would correlate to needing around 30-60 images per class.

From the plot in figure 6.2 one can see a sharp ascension up until around 20% of the training set, or 185 images, were used to train the model. This corresponds to roughly 30 images per class. Afterwards, it starts to even out. Hence, the claim that at least 30 images are need to train a somewhat decent model is correct. However, as seen, more data usually generates even better result.

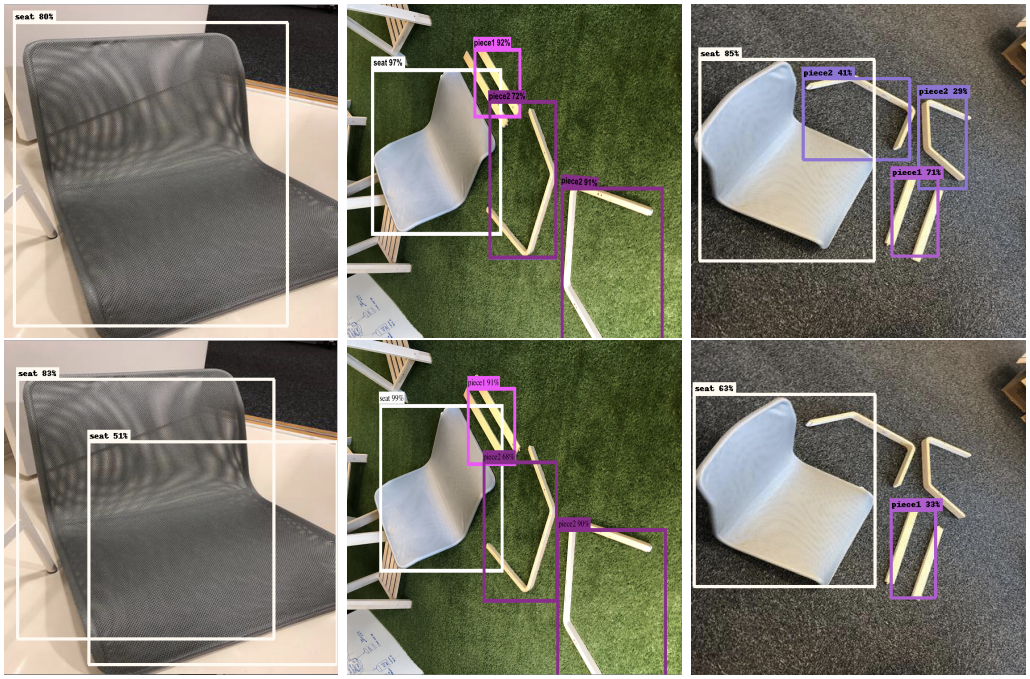

Figure 6.3: Images showing how the same three images were classified differently using different models. The top row is classified using the model that was trained on all of the training set. The bottom row shows the images that were trained on only 15% of the training set.

Chapter 7

Object Tracking

To improve the user experience, object tracking was tried. This infers that the application would keep track on where certain objects are on the screen. The thought behind this idea was that the application would not have to constantly run the current frame through YOLO model in order to keep track of where objects were, which could lead to a choppy user experience.

This chapter will describe the concept of object tracking, how it was implemented and tested, as well as our reasoning for not keeping this functionality in the final application.

7.1 Testing Object Tracking with Vision package

Vision is a package from Apple which contains a lot of different methods for images and video. It contains still image analysis, image sequence analysis, object tracking, face detection etc.

On their website, Apple has a project that lets any user try out the object tracking on a video [42]. When trying this on one of the furniture we are going to assemble, the result was very promising. While the parts were laying on the floor and simultaneously moving the camera around, the objects were tracked fairly well. It was only when the camera was moved in such a way that made the pieces rotate in the picture that it started having a hard time tracking it.

For solving this, object detection can be performed in a reasonable time interval and be tracked until object detection is performed once again etc.

The difference for this project and Apple's test project is that object tracking is going to be performed in real-time. This puts a limit on how many frames per second we can perform object tracking, since in a video playback you can just choose how fast you want to feed the new images. In real-time, the world doesn't stop moving.

After the system was implemented into our application different frame rates were tested. The optimal value was somewhere in-between 10-30 fps. If you went higher than that the application would become very choppy and eventually shut down.

Going lower than 10 fps the user will start to experience that the objects are hard to track in rapid movements.

For this application, however, the user is not going to encounter any scenarios where objects are flying around rapidly. Therefore we will settle for 20 fps as the optimal value for performance since any higher amounts don't really contribute to a better experience.

The code presented in appendix E.1. show how the object tracking was set up. One important thing to realize when setting this up is that the heavy calculations are run on a different thread than the main thread (in this case the work thread).

That is why they can be executed in a while true loop. Later though, when drawing on the GUI wants to be made, they must be done on the main thread.

7.2 Combining Object detection with Object Tracking

The purpose of having the object tracker is to be able to avoid performing object detection and object recognition 30 frames per second or so. Also, doing this with YOLO nets have shown choppy results where an object can be identified in one frame, not identified in the next and then again identified. There is no real issue with this since both solutions perform really well overall. There is however concerns for the end user to have a good experience.

One way to solve this is to perform the detection and recognition once, to later continue to track the position of those items with a cheaper algorithm. Object detection and recognition will then be repeated, but only update the state of the rectangles if the correct objects are identified. In this application, an arrow was drawn between the objects with an instructional text for the user. Doing this created a rather big issue though.

In this project, ARKit by Apple has been utilized. As previously stated, when working with ARKit, a frame from the camera is fetched by either calling the function snapshot() on the ARSCNView object or getting the currentFrame attribute from the ARSession. However, both of these images contain both the image from the camera as well as all the virtual items rendered in the ARScene. We did not find a way to capture just the image from the camera.

This whole scenario created a kind of positive feedback loop because the virtual arrow that was rendered between the objects was usually contained within either one of the tracking rectangles. Thus, the arrow's position was determined by the tracking rectangle and the tracking rectangle was tracking the position of the arrow. Just the tiniest change in the picture made the arrow jump up and down until it finally got out of frame.

The takeaway from this was to skip object tracking and solve the problem in another way.

An assumption was made that just as talked about before, the furniture parts are laying still on the ground while the user is using the application. That means

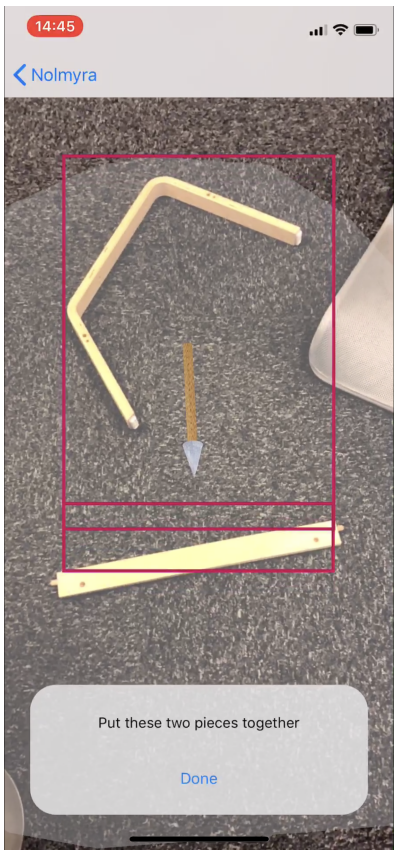

Figure 7.1: The image shows an early stage of the application were two pieces have been recognized by the network and an outline around them has been drawn. Between them, an arrow is rendered on the floor, showing how to put together the two pieces. The objects are continuously being tracked within the frame and the arrow is also updated.

that we can rely on ARKit to keep track of where the object is for us instead.

Switching gear to that solution made the whole application much better and easier to code. A takeaway from that is that adding more technology to a project does not always equal a better product. Sometimes it is better to use the components the way they are meant to be used and try not to make it more complicated than it needs to be.

Chapter 8

The Finished Application

This chapter will round off the work process of the report. It goes through how the final application was designed and implemented in 8.1. The chapter then ends with section 8.2, detailing the evaluation process of the application, which was the by having users test the finished application.

8.1 The iOS Application

The application is build for iOS using Xcode and Swift 4.0. The app has been developed specifically for iPhone X. It has also only been tested on iPhone X. However, the app should work on other iPhone models that support ARKit 2.0 as well.

8.1.1 Class diagrams

Below are some class diagrams for the reader to better understand what different parts there are, how they fit together and how the application works as a whole. The diagrams are divided into three parts to be readable in the paper.

The application follows the Model View Controller design pattern as well as the Delegate design pattern. This decision was made since many tools and features in iOS and Swift are implemented that way, so we are just following the standard procedure.

AssemblerViewController is the controller for the view that holds the ARSCNView. It is responsible for what happens when the user taps on a button in that view and displaying the correct information to the user at the right times. It also holds the **Instruction Executioner** which holds a set of instructions that it gets at initialisation. The instruction executioner executes the current instruction and therefore decides what will happen when it executes. Every instruction is executed on a worker queue thread to avoid the video feedback to freeze during execution.

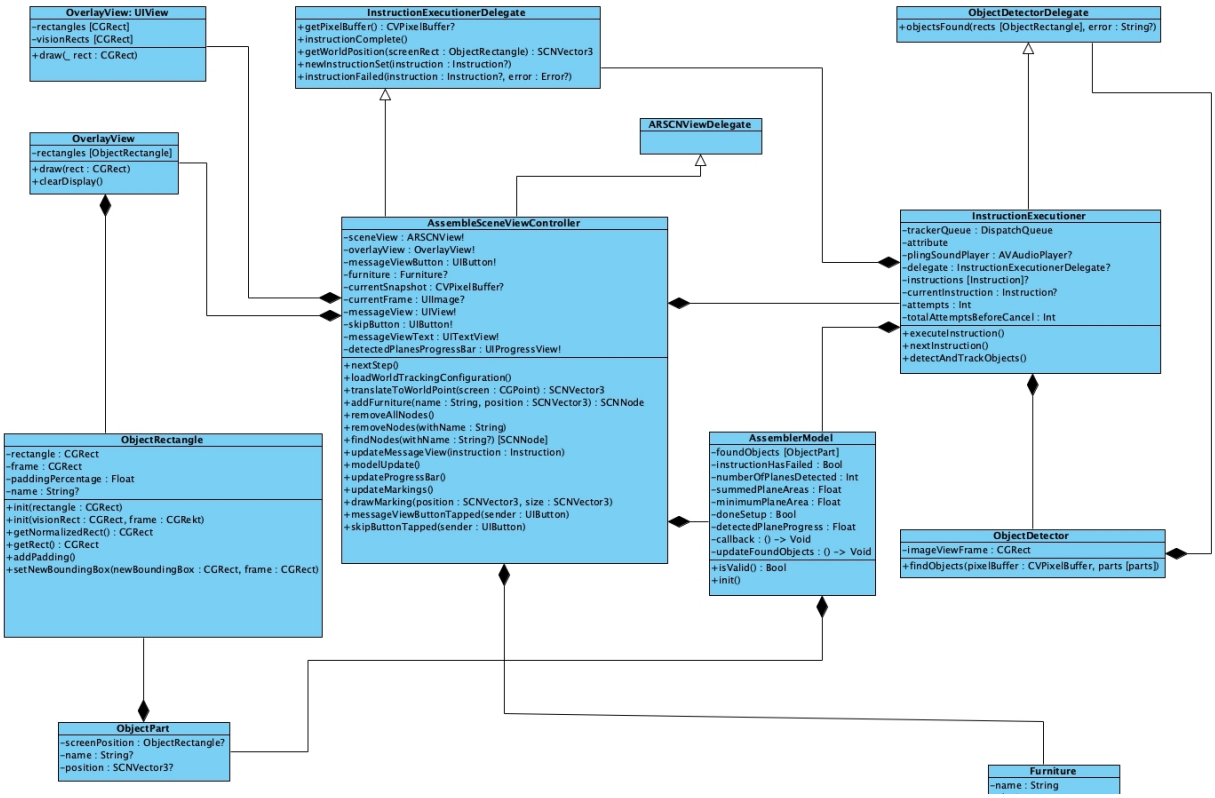

Figure 8.1: A class diagram of the assembler view part of the application. The Furniture class in the lower right is incomplete and is shown in the next diagram.

The InstructionExecutioner holds the **ObjectDetector** which finds specific objects in a trained model from a pixel buffer. The found objects are passed back as **ObjectRectangles** through the delegate. The object rectangles are bounding boxes of the objects on screen. They are represented as CGRect's and can be stored and fetched as either regular or normalized rectangles. The normalized variants are needed in all kinds of machine learning purposes and the regular form are used in all GUI purposes, such as in the **OverlayView**. (Normalized rectangles have values of 0-1 for widths, heights, x-, and y-position. The origin is in the lower left corner.)

The **AssemblerModel** is the model in the AssemblerViewController, but is also used in the InstructionExecutioner since they are highly dependent on each other. The most important attributes it holds are the **ObjectParts** that have been spotted by the app during the execution of the current instruction. That way, all the parts needed for the current instruction do not need to be in the same image together but instead can be spotted separately.

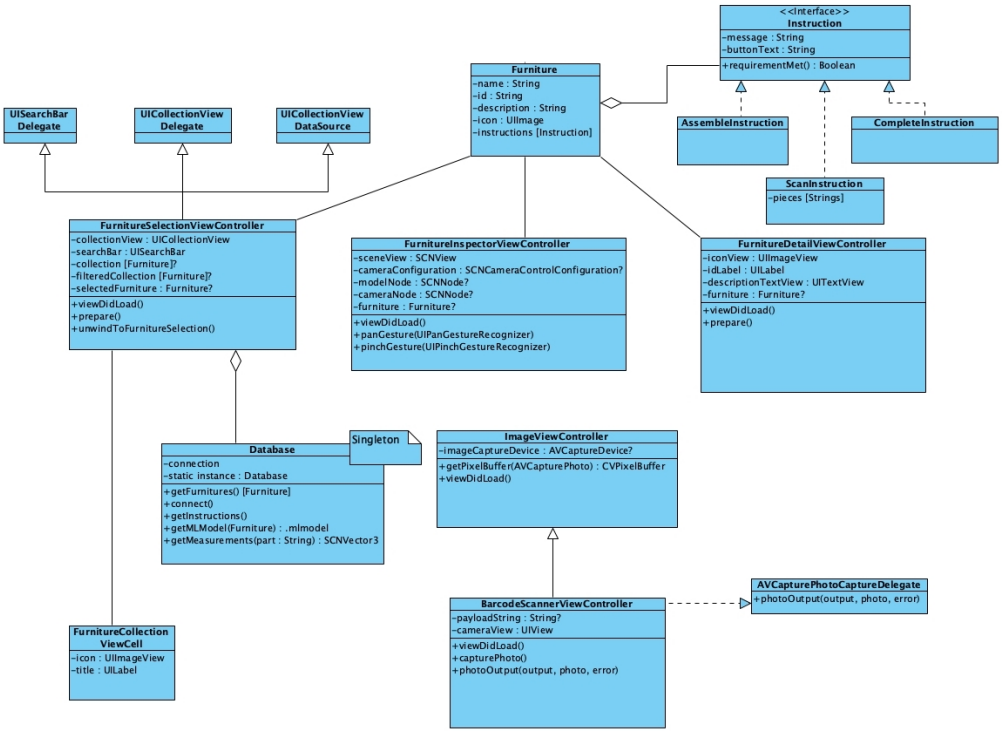

Figure 8.2: A class diagram of the furniture selection views part of the application. The Furniture class.

The furniture that the user wants to put together is selected in a collection view **FurnitureSelectionViewController** which conforms to the UICollectionViewDelegate- and UICollectionViewDataSource protocol for the collection view functionality.

The data to be viewed in the controller is fetched from the **Database** class. The return data from the class are hardcoded from the start but can easily be changed to fetch data from a server.

The **Furniture** holds all the information about a specific furniture as well as the instruction set of how to put it together. The instruction set consists of **Instruction**'s that can be of the kind **ScanInstruction**, **AssembleInstruction** or **CompleteInstruction**. A regular Instruction is usually just text instructions to the user, while the scan instructions tell the instruction executioner to look for specific parts during that step. The assemble instruction is run when the user is in the process of assembling two parts. Finally the complete instruction is given which tells the executioner that the furniture has been fully assembled.

For scanning barcodes, the **BarcodeScannerViewController** is used, which inherits all the photo capture functionality from the **ImageViewController**.

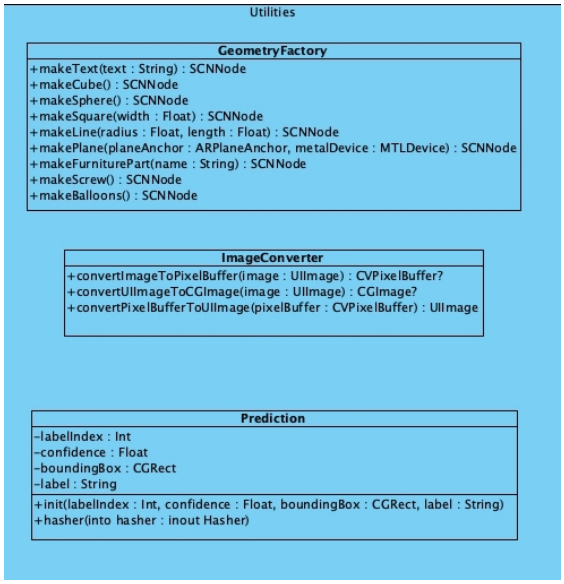

Figure 8.3: A class diagram of the utilities part of the application. These are some classes that make the rest of the code in the application easier to understand and use.

In the utilities folder there are some classes that simplifies the rest of the application by holding code that removes a lot of redundancy.

The **GeometryFactory** creates all the 3D geometry for the ARScene and return the nodes containing this geometry. An example is that it can create all the furniture parts as virtual object nodes to be placed in the scene.

A lot of conversion between pixel buffers and UIImage's are made and the **ImageConverter** eases the pain of having to do that in many places.

Finally the **Prediction** class helps the object detector when getting results from the VNCoreMLRequest.

8.1.2 Connecting two pieces in AR

After finding the specific pieces that are supposed to be connected they are created as virtual objects in the scene on their respective locations. The location is determined using hit tests of the screen positions to the detected plane from the ARScene.

The virtual objects are rendered in the scene using SCNNode's as talked about in 3.4 and the node position is the origin of the 3D model. The origin has been chosen to be at a location that is touching the floor when it is standing. That way they can easily be placed to look like they are standing right on the floor. When both pieces are placed in the scene they are to be moved with an animation in a way so that they become connected.

Each virtual object node can embed another node called the "anchor point". The position of this node within the parent node is where the other object is to be connected. To connect them, either the object without the anchor point moves to the anchor point position, or the two object move so that each of their anchor points are at the same position. This is accomplished using the algorithm descibed in code in appendix F.1.

Afterwards, the items in 'action' are performed on the respective node.

8.1.3 The finished GUI

The finished GUI is similar to the prototype but some details are different. The resulting screen shots can be seen in figure 8.4 and 8.5.

8.2 User Testing

Evaluating the AR experience is possible on a technical level, to research if the combination of AR with object recognition is feasible. This can be done by, for instance, making sure the application is running at an acceptable frame rate. However, as this report also wants to research into the desirability of this combination of technology, the same approach is not as appropriate. What was done was to test the application ourselves, as well as get outside evaluations through user tests.

When setting up the user testing we decided to use mainly observation and task demonstration as elicitation techniques, as well as a questionnaire. These methods are recommended in Soren Lauesen's book 'Software requirements, Styles and techniques' [45].

The user tests were set up by having people from the office sign up online to come test our application. This was done since lots of users have problems describing why they do a certain thing. The users were also asked to answer a questionnaire afterwards where they answered several questions. The questionnaire was for the

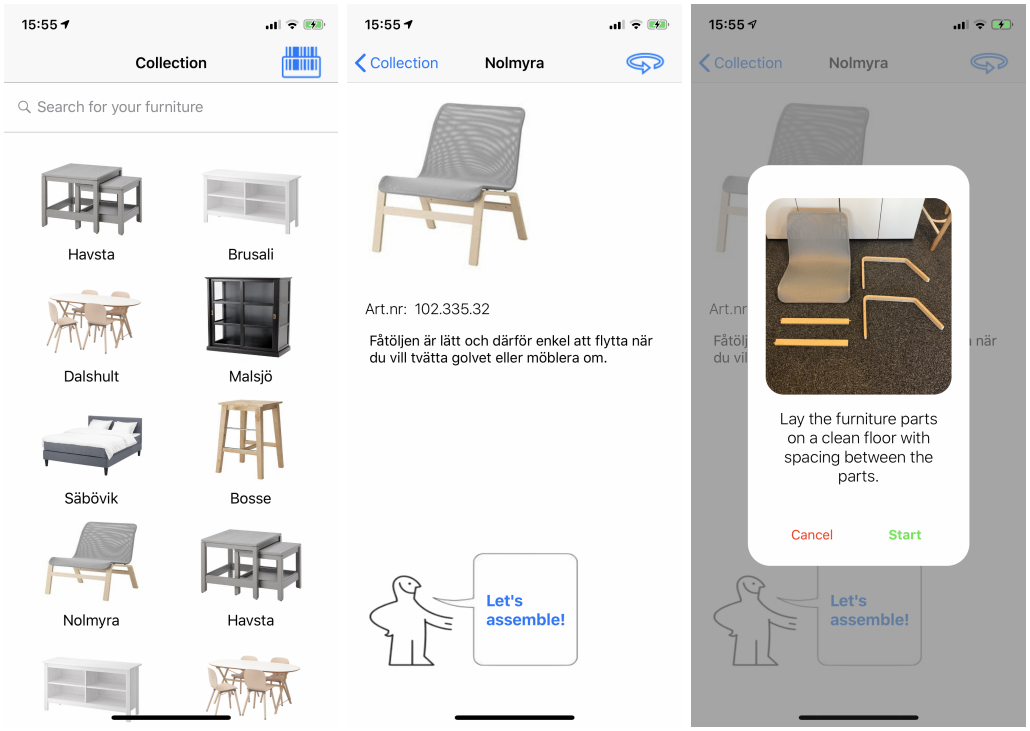

Figure 8.4: Images from the use of the application when selecting a furniture and right before starting the assembler view.

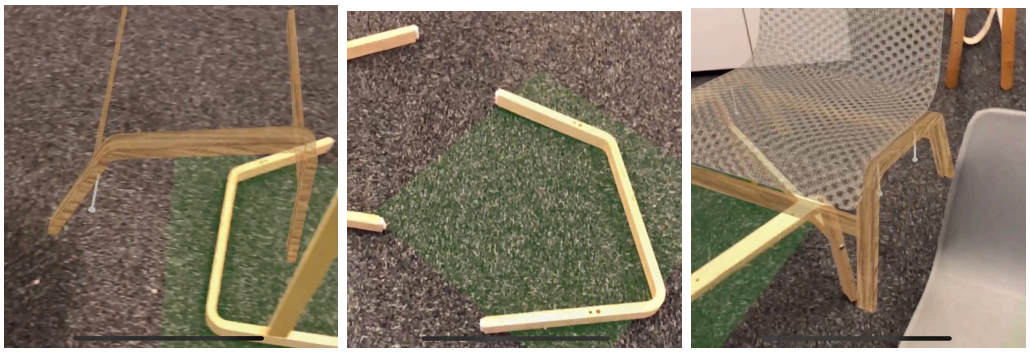

Figure 8.5: Images from the application when having started the ARScene, finding objects and showing animations when putting them together. The green rectangle on the floor is an indication that the part has been recognized by the application. We call these markings.

user to give opinions and suggestions about the app. While the users were doing the tests, they were observed and comments were written down, reporting on how they behaved. We also used screen recording on the phone, so to be able to replay the scenario afterwards.

The form asked the participants the following questions:

- Did you know that you could skip instructions?
- How easy was it to understand how to get to the next step?
- How easy was it to understand how the pieces fit together?
- How easy did you feel the app was to use?
- Compared to using paper instructions, did the app make it easier to understand how to put together the furniture? ? If not, then why?
- What potential do you see in this app?
- General feedback
- Suggestions of improvement
- What is your role at Jayway?

A user survey was done in order to evaluate the performance of the application. By having a multitude of interested people booking time slots for a ten minute testing period, a decent amount of sample data could be gathered and evaluated. Figures 8.6 to 8.11 show bar charts and pie charts representing the answers gathered from the participants to some of the questions they were asked.

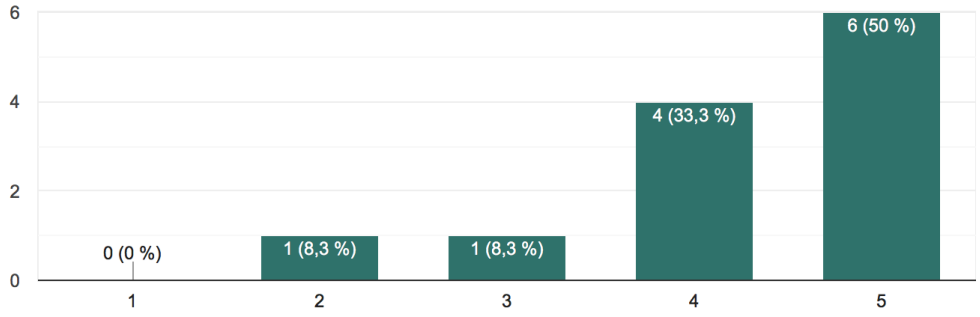

Figure 8.6: Results from when users were asked "How easy was it to understand how to get to the next step?". X axis is their rating from 1-5 and y axis is how many.

By collecting the data from the forms, the recorded videos and our own observations we could find some general trends and reach a few conclusions.

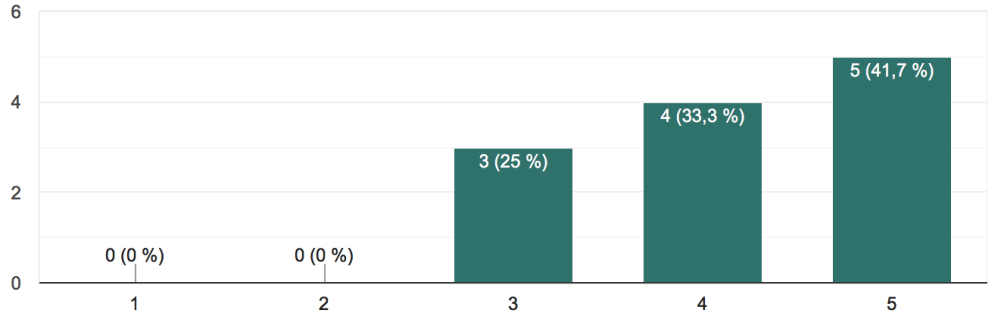

Figure 8.7: Results from when users were asked "How easy was it to understand how the pieces fit together?". X axis is their rating from 1-5 and y axis is how many

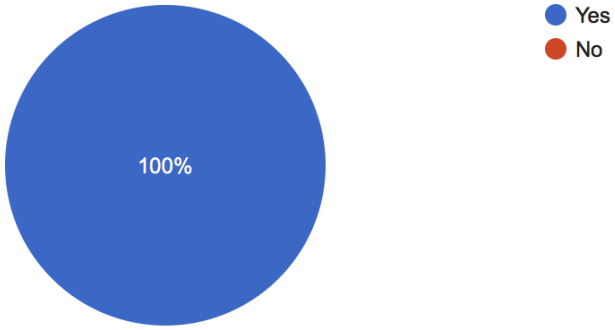

Figure 8.8: Results from when users were asked "Did you know that you could skip instructions?"

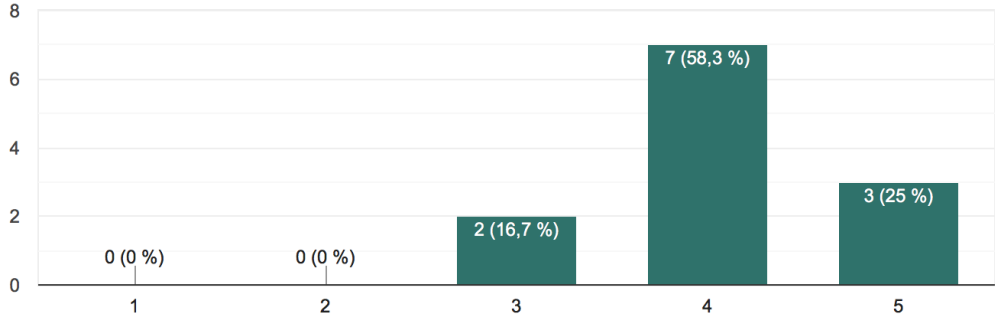

Figure 8.9: Results from when users were asked "How easy was it to understand how to get to the next step? X axis is their rating from 1-5 and y axis is how many"

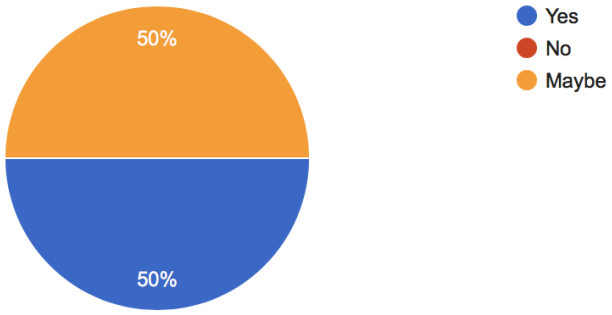

Figure 8.10: Results from when users were asked "Compared to using paper instructions, did the app make it easier to understand how to put together the furniture?"

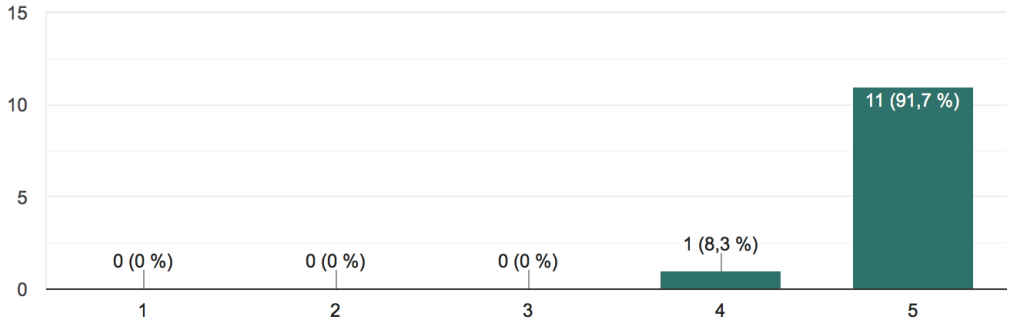

Figure 8.11: Results from when users were asked "What potential do you see in this app? X axis is their rating from 1-5 and y axis is how many"

What we found was that half of the participants thought it was easier to use our application to assemble the furniture than using a traditional paper instruction, and no-one thought it was definitively more difficult, as seen in figure 8.10. This could be due to the fact that the piece of furniture we chose to train on were quite simple to assemble, relative to other possible furniture. It would be interesting to perform the same task but on a more complicated piece of furniture.

Every participant thought that the application had good or great potential, see figure 8.11, if it were to be improved. This result shows that the use of this technology is mostly desirable, if done in a more proper way.

The answers show that most people could understand how to use the application and could intuitively start up the application and start the assembly process without outside help, shown in figure 8.6, 8.7 & 8.9. In some cases we noticed that people were a bit confused with the augmented reality interface. Several participants had a hard time understanding the purpose of the green rectangles that were rendered around a found object in the scene. Mostly because the rectangles were significantly larger than the object itself, hence, it often overlaps with other objects on the floor.

The hassle of having to put the phone down in between looking at the animation and actually assembling the furniture parts was also brought up. Some of the participants had to pick up the phone several times during certain steps to make sure they were executing the task in the correct manner.

At times, some participants were getting confused because the application could not find the correct parts. This happened because of several reasons. One being that they had accidentally skipped an instruction, thus the application were looking for model parts that had not yet been assembled. Another reason being that the machine learning model were inconsistent at times and could not identify certain in the orientation it was currently in. It also happened when the users did not perform the first task given by the application, being that they had to make sure that the parts were laying on the floor and not overlapping each other. A solution to these problems could be to give the users a more clarified and more intuitive set of instructions such that the user don't accidentally skip an important instruction. Another solution is to retrain the machine learning model to be able to identify the parts in more orientations.

The user test was performed in the Jayway offices on people with background in technology, where most participants were developers or designers. Due to this, the result could be skewed, since people with similar background have a tendency to understand each others intent. The same test would have to be performed on actual end users with a more varied background to see if it would match the results we've gathered.

Chapter 9

Conclusion

The main goal for this project was to investigate whether it was possible to use object detection and object recognition together with augmented reality, both with respect to if it is technically possible and if it creates value for the customer.

From a technical perspective it is surely possible to combine the two in a single product. In the report we have shown that a neural network like YOLO can be used for object detection, while at the same time having an AR session active to keep awareness of the surrounding space and where the device is located in it. The main challenge is to collect large amounts of relevant data and training a neural network to give a good generalisation of that data. This can partly be eased by using transfer learning, but for a full scale product there are still much data needed. Apart from that, there are a lot of frameworks (for AR, maching learning, object detection etc.) to be used to avoid developing from scratch.

From a customer value perspective there is also potential but not as much interest with the current hardware. From our user tests we got some interesting results. Although the users liked our solution to a paperless assemble manual in AR, they were not so keen to holding a phone while watching the instructions to then put the phone down when putting together a piece of furniture. Apart from that they all saw big potential with using this type of technology together, but a pair of glasses would be really great. However, using other hardware was outside the scope of this project.

A fully working prototype for combining object detection with augmented reality was finally developed for iPhone X.

9.1 Future Work

Because of the rapid development in the fields discussed in this report, much of what we've gone through could be improved in the future. With new *Augmented Reality* and *Mixed Reality* hardware being developed by big franchises, such as Facebook [43], the performance will increase ever so rapidly.

Underneath, we describe a few ways not only our application could be improved, but also the combination of used technologies in general.

9.1.1 Wearable

One improvement of our application would be to port it to, not only hand-held devices, but also the headgear, thus allowing the user to building with both hands and never have to pause during the assembly process.

9.1.2 Detecting smaller objects

Another issue with our model is that we are unable to detect screws and bolts because of their comparatively small size, instead, we just inform the user that screws should be used during a particular step using animations. This issue could possibly be fixed by adapting our model into taking account to smaller objects. Some future work could be to investigate whether this would be possible.

9.1.3 Mask R-CNN

Since there was a lot of confusion around the green markings from the user tests, it would be an improvement to only mark the exact position of the part and not the surrounding floor. For this application only the bounding box of the 2D image is known, not the floor where it lays.

A possible solution to this in the future could be to implement a Mask R-CNN network. However, the problem with this type right now is that it does not work in real time applications (definitely not on mobile devices). If this would get better it could be implemented in the app for a much nicer user experience.

9.1.4 Starting the app in the ARScene

When comparing to other AR Applications in the App Store, a lot of them starts right in the AR scene. This seems to be somewhat of a convention in the AR community.

We chose to have the user select the furniture before entering the AR scene to make the GUI implementation easier. However, it could be changed so that the furniture could be selected either from an overlay or a bottom card design over the AR scene view.

9.1.5 Shared AR experience

Ever since ARKit 2.0 was released, it is possible to share an AR experience with another user. Since the app already utilizes ARKit 2.0 it would be relatively simple to implement a shared experience. The shared experience could make it easier for multiple people to work together to assemble a furniture. Big furnitures are rarely assembled by just one person.

9.1.6 Voiced instructions

Reading instructions on the screen while viewing a video preview is not the most intuitive in an AR environment. The eyes are mostly focused on what is going on in the environment and text instructions given on an overlay can easily be missed if they are subtle.

Furthermore, people with hard of seeing could have a hard time with both reading the instructions and seeing the 3D models. This problem can be solved by introducing voiced instructions in the app.

9.1.7 Finding the anchor points with object detection

Since screws were to small to detect, so are the anchor points on the furniture parts that connect each other. An improvement in the future could be to try to detect the exact anchor points on the real furniture parts and rendering the virtual object right on that location. This would require more work on training the neural net to detect those small parts and probably higher resolution on the images.

9.1.8 Make it scalable

The application only supports one furniture. For a full scale application it would need to support many more furnitures. For a complete catalogue there could exist hundreds of furnitures.

To add support for a new furniture in the app the following steps are needed:

1. Take at least 30 pictures of every furniture part, however, more is recommended for more complex objects and more backgrounds
2. Train a neural net to recognize those parts and import the model into Xcode
3. Add an instruction set on how to put the furniture together
4. Add 3D models of all the parts and the fully assembled model
5. Add anchor points and screw points to the models

The problem of making the app scalable is the size of the ML-model. For many furnitures it could easily be the size of 1 GB or more. A way to solve this is to have an downloadable ML-model for each furniture. These could be fetched from a database before starting the assembly session.

All the other instructions, geometry, anchor- and screw points could also be fetched from a database.

Figure 9.1: Book shelf rendered in a corner. To the left no objects in front so it looks realistic.On the right, the book shelf has a lamp in front of it.

9.1.9 Occlusion problem in AR

One of the current problems with Augmented Reality for us is that the models are rendered on top of the real world. When there are no other object in the scene and we just have a simple plane to render one, the result can look decently realistic. However, when other objects are in the scene, the illusion of realism is easily lost. Example of this can be seen in figure 9.1.

A way to solve this problem would be to create a 3D model of the real world, to be able to find foreground objects and thus, add a transparency mask on the object we wish to render in the real world to increase the effect of illusion that it is actually there.

As of now it is not possible with ARKit to do this in a real time fashion. However, there are scholars working on solutions for the occlusion problem in augmented reality, such as Shah, Niyati [44].

Bibliography

[1] Microsoft HoloLens, Software Asset Management – Microsoft SAM, Microsoft https://www.microsoft.com/en-us/hololens

[2] Glass Explorer Edition, Google Developers, Google https://developers.google.com/glass/

[3] Augmented Reality, Apple https://www.apple.com/lae/ios/augmented-reality/

[4] Pokémon GO, NIANTIC https://www.pokemongo.com/en-us/

[5] IKEA Place, App Store, IKEA Systems B.V, (2017) https://itunes.apple.com/us/app/ikea-place/id1279244498?mt=8

[6] Digi-Capital. *Ubiquitous $90 billion AR to dominate focused $15 billion VR by 2022* (January 26, 2018) https://www.digi-capital.com/news/2018/01/ubiquitous-90-billion-ar-to-dominate-focused-15-billion-vr-by-2022/

[7] Fieldbit Hero, FieldBit https://www.fieldbit.net/products/fieldbit-hero/

[8] Sanni Siltanen, *Theory and applications of marker-based augmented reality*, Copyright © VTT 2012 (ISBN:978-951-38-7449-0) https://www.vtt.fi/inf/pdf/science/2012/S3.pdf

[9] Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E, *ImageNet Classification with Deep Convolutional Neural Networks* in *Advances in Neural Information Processing Systems 25* ed. F. Pereira and C. J. C. Burges and L. Bottou and K. Q. Weinberger, pp 1097-1105 (2012) http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[10] Joseph Redmo, Santosh Divvala, Ross Girshick, Ali Farhadi (2016), University of Washington , Allen Institute for AI , Facebook AI Research You Only Look Once: Unified, Real-Time Object Detection https://arxiv.org/pdf/1506.02640.pdf

[11] D. Chatzopoulos, C. Bermejo, Z. Huang and P. Hui, "Mobile Augmented Reality Survey: From Where We Are to Where We Go," in *IEEE Access*, vol. 5, pp. 6917-6950, 2017. doi: 10.1109/ACCESS.2017.2698164 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7912316&isnumber=7859429

[12] Region-based Segmentation and Object Detection S. Gould, T. Gao and D. Koller, in *Advances in Neural Information Processing Systems 22*, ed. Y. Bengio and D. Schuurmans and J. D. Lafferty and C. K. I. Williams and A. Culotta, pp 655–663 (2009) http://papers.nips.cc/paper/3766-region-based-segmentation-and-object-detection.pdf

[13] Yann LeCunn, Marc'Aurelio Ranzato, Deep Learning Tutorial, ICML, Atlanta, (2013), https://cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf

[14] MockFlow, A Produle Systems Pvt Ltd https://mockflow.com/

[15] Extreme Programming O'Reilly Media ISBN-13: 978-0596004859

[16] Myron Krueger - Videoplace, Responsive Environment https://www.youtube.com/watch?v=dmmxVA5xhuo

[17] SLAM: Simultaneous Localization and Mapping - Wolfram Burgard, Cyrill Stachniss, Kai Arras, Maren Bennewitz http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf

[18] Adam: A Method for Stochastic Optimization, 22 Dec 2014 Diederik P. Kingma, Jimmy Ba https://arxiv.org/abs/1412.6980

[19] Neural Networks for Machine Learning, Lecture 6a Geoffrey Hinton http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[20] An overview of gradient descent optimization algorithms Sebastian Ruder http://ruder.io/optimizing-gradient-descent/index.html#fn16

[21] Beware of overfitting and underfitting, O'Reilly https://www.oreilly.com/library/view/scala-and-spark/9781785280849/3c1c7845-811d-47b9-a54f-c2584fe930b3.xhtml

[22] Dropout: A Simple Way to Prevent Neural Networks from Overfitting Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov http://jmlr.org/papers/v15/srivastava14a.html

[23] Batch Normalization layer, Tensorflow https://www.tensorflow.org/api_docs/python/tf/nn/batch_normalization

[24] Vincent Dumoulin, Francesco Visin, *A guide to convolution arithmetic for deep learning* MILA, Université de Montréal, AIRLab, Politecnico di Milano (2018) (arXiv:1603.07285v2) https://arxiv.org/pdf/1603.07285.pdf

[25] Convolutional Neural Network 3 things you need to know, MathWorks https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html

[26] ImageNet 2016 Stanford Vision Lab, Stanford University, Princeton University http://www.image-net.org

[27] Scanning and Detecting 3D Objects, Apple https://developer.apple.com/documentation/arkit/scanning_and_detecting_3d_objects

[28] Face Detection System Based on Viola - Jones Algorithm, Mehul K Dabhi1, Bhavna K Pancholi2 https://pdfs.semanticscholar.org/f63c/fcdd63bd34bcd6ec028169e6fe144e9cc83c.pdf

[29] Lode's Computer Graphics Tutorial - Flood Fill, Lode Vandevenne https://lodev.org/cgtutor/floodfill.html

[30] AVDepthData, Apple Documentation https://developer.apple.com/documentation/avfoundation/avdepthdata

[31] Object recognition without deep-learning, Combine http://combine.se/object-recognition-without-deep-learning/

[32] Derek Bradley, Gerhard Roth, *Adaptive Thresholding Using the Integral Image*, Carleton University, Canada & National Research Council of Canada (2005), http://people.scs.carleton.ca/~roth/iit-publications-iti/docs/gerh-50002.pdf

[33] Joseph Redmon, Ali Farhadi, *YOLO9000: Better, Faster, Stronger*, University of Washington & Allen Institute for AI (2016) https://arxiv.org/pdf/1612.08242.pdf

[34] Joseph Redmon, Ali Farhad *YOLOv3: An Incremental Improvement* University of Washington (2018) https://arxiv.org/pdf/1804.02767.pdf

[35] The PASCAL Visual Object Classes http://host.robots.ox.ac.uk/pascal/VOC/

[36] Rafael Padilla, Metrics for object detection https://github.com/rafaelpadilla/Object-Detection-Metrics

[37] Detection Evaluation, Common Objects in Context (COCO) http://cocodataset.org/#detection-eval

[38] Advanced Usage, Turi Create, Apple https://apple.github.io/turicreate/docs/userguide/object_detection/advanced-usage.html

[39] Turi Create, Apple https://github.com/apple/turicreate

[40] Simple Image Annotator, Sebastian G. Perez (@sgp715) https://github.com/sgp715/simple_image_annotator

[41] Jan Hosang, Rodrigo Benenson, Bernt Schiele *Learning non-maximum suppression*, Max Planck Institut für Informatik, Saarbrücken, Germany (9 May 2017) https://arxiv.org/pdf/1705.02950.pdf

[42] Tracking Multiple Objects or Rectangles in Video, Apple https://developer.apple.com/documentation/vision/tracking_multiple_objects_or_rectangles_in_video

[43] TechCrunch, *Facebook's Head Of Augmented reality on its plans for AR glasses* Published on Oct 24, 2018 https://youtu.be/JEGqc9wzC0o?t=1041

[44] Shah, Niyati. *Realtime Object Occlusion In Augmented Reality Environments.* (2018) , B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology, https://pdfs.semanticscholar.org/0329/d772efe01b5d818dfc22e4a357d03324a01e.pdf

[45] Software requirements, Styles and techniques Soren Lauesen ISBN: 0-201-74570-4

[46] Seene SLAM Technology - culturengine https://www.youtube.com/watch?v=434SsV9nGHc

[47] Keras: The Python Deep Learning library, Keras https://keras.io

Appendices

Appendix A

Code from chapter Augmented Reality

A.1

```
1  func loadWorldTrackingConfiguration()
2      {
3          let configuration = ARWorldTrackingConfiguration()
4          configuration.planeDetection = [.horizontal]
5
6          // All the objects that are tracked is contained in the
       Objects folder
7          guard let detectingObjects =
       ARReferenceObject.referenceObjects(inGroupNamed: "Objects",
       bundle: nil) else { return }
8          configuration.detectionObjects = detectingObjects
9
10         // Setting up tracking of images
11         for imageURL in trackingImageURLs
12         {
13             guard let image: CGImage = UIImage(named:
       imageURL)?.cgImage else { return }
14             let referenceImage = ARReferenceImage(image,
       orientation: CGImagePropertyOrientation.up, physicalWidth: 0.3)
15             configuration.detectionImages.insert(referenceImage)
16         }
17
18         configuration.maximumNumberOfTrackedImages =
       trackingImageURLs.count
19
20         // Running the sessoin with the configuration
21         sceneView.session.run(configuration)
22     }
```

A.2

```
// Load the scene
let scene = SCNScene(named: "art.scnassets/world.scn")!
sceneView.scene = scene
```

A.3

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode,
    for anchor: ARAnchor)
     {
              // If detected an object
          if let objectAnchor = anchor as? ARObjectAnchor
          {
                       // Add some 3D text to the scene
              let objectName = objectAnchor.referenceObject.name!
              let textNode = GeometryFactory.makeText(text:
    objectName)
              node.addChildNode(textNode)
          }
          // If detected a plane
          else if let planeAnchor = anchor as? ARPlaneAnchor
          {
          // Add a plane geometry of the detected floor to the scene

    node.addChildNode(GeometryFactory.createPlane(planeAnchor:
    planeAnchor, metalDevice: metalDevice!))
              model.numberOfPlanesDetected += 1
          }
     }
```

If nodes need to be rendered outside of this function it can be done by accessing the scenes root node.

```
sceneView.scene.rootNode.addChildNode(node)
```

Appendix B

Code from chapter Neural Networks

B.1

```
#Python script from training
#Project path:
        master-thesis/Training/trainer.py
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.utils import shuffle

train_images = []
train_labels = []
loadImages(train_images, train_labels, "Train", 200)
train_images = reshapeArray(train_images)

test_images = []
test_labels = []
loadImages(test_images, test_labels, "Test", 39)
test_images = reshapeArray(test_images)

# Create the neural network
model = keras.Sequential([
        keras.layers.Conv2D(4, kernel_size=(5, 5),
    strides=(2, 2), input_shape=(image_height, image_width,
    number_of_color_channels)),

        ["The code for the hidden layers"]

        keras.layers.Dense(4, activation=tf.nn.softmax)
```

```
28 ])
29
30 model.compile(optimizer=keras.optimizers.Adam(),
31               loss='sparse_categorical_crossentropy',
32               metrics=['accuracy'])
33
34 train_data, train_labels = shuffle(train_data, train_labels)
35
36 early_stopping =
      keras.callbacks.EarlyStopping(monitor='val_acc',
      patience=5, verbose=1)
37 checkpoint =
      keras.callbacks.ModelCheckpoint("./Models/Nolmyra.h5",
      monitor='val_acc',
38 verbose=1, save_best_only=True, save_weights_only=False,
      mode='auto', period=1)
39
40 history = model.fit(train_data, train_labels, epochs=40,
      batch_size=10, validation_data=(test_images,
      test_labels), callbacks=[early_stopping, checkpoint] ,
      verbose=1)
41
42 model.save("./Models/recognizer.h5")
```

B.2

```
1  Conv2D(4, kernel_size=(5, 5), strides=(2, 2),
       input_shape=(image_height, image_width,
       number_of_color_channels)),
2  Conv2D(4, kernel_size=(3, 3), strides=(1, 1),
       input_shape=(image_height, image_width,
       number_of_color_channels)),
3  MaxPool2D(pool_size=(2, 2), padding="valid"),
4  BatchNormalization(),
5  LeakyReLU(),
6  Conv2D(8, kernel_size=(3, 3), strides=(1, 1)),
7  Conv2D(8, kernel_size=(3, 3), strides=(1, 1)),
8  Conv2D(8, kernel_size=(3, 3), strides=(1, 1)),
9  MaxPool2D(pool_size=(2, 2), padding="valid"),
10 BatchNormalization(),
11 LeakyReLU(),
12 Conv2D(16, kernel_size=(3, 3), strides=(1, 1)),
13 Conv2D(16, kernel_size=(3, 3), strides=(1, 1)),
14 Conv2D(16, kernel_size=(3, 3), strides=(1, 1)),
15 MaxPool2D(pool_size=(2, 2), padding="valid"),
16 BatchNormalization(),
```

```
17 LeakyReLU(),
18 Flatten(),
19 Dropout(0.5),
20 Dense(64, kernel_regularizer=keras.regularizers.l2(0.003),
      activation=tf.nn.relu),
21 GaussianNoise(0.2),
22 Dense(64, kernel_regularizer=keras.regularizers.l2(0.003),
      activation=tf.nn.relu),
23 Dropout(0.25),
24 Dense(4, activation=tf.nn.softmax)
```

B.3

```
1  #Project path:
      master-thesis/FeatureTraining/transferLearning.py
2  from keras import applications
3  from keras.preprocessing.image import ImageDataGenerator
4  from keras import optimizers
5  from keras.models import Sequential, Model
6  from keras.layers import Dropout, Flatten, Dense,
      GlobalAveragePooling2D, Input, Conv2D, MaxPool2D
7  from keras import backend as k
8  from keras.callbacks import ModelCheckpoint,
      LearningRateScheduler, TensorBoard, EarlyStopping
9
10 #Load data and pretrained model
11 img_width, img_height = 256, 256
12 train_data_dir = "data/train"
13 validation_data_dir = "data/val"
14 nb_train_samples = 129
15 nb_validation_samples = 21
16 batch_size = 16
17 epochs = 50
18 input_layer = Input(shape=(256,256,3))
19 model = applications.VGG16(include_top=False,
      weights='imagenet', input_tensor=input_layer,
      pooling=None)
20
21 #Cut network and add own layers
22 x = model.get_layer('block5_pool').output
23 x = Flatten()(x)
24 x = Dense(512, activation="relu")(x)
25 predictions = Dense(4, activation="softmax")(x)
26
27 # creating the composed model
```

```
28 model_final = Model(inputs = model.input, outputs =
      predictions)
29 for layer in model_final.layers[:-2]:
30     layer.trainable = False
31
32 # compile the model
33 model_final.compile(loss = "categorical_crossentropy",
      optimizer = optimizers.SGD(lr = 0.0001, momentum = 0.9),
      metrics=["accuracy"])
34
35 #Hidden lines of code
36 .........
37
38 #Train the model
39 hist = model_final.fit_generator(
40 train_generator,
41 epochs = epochs,
42 validation_data = validation_generator,
43 callbacks = [checkpoint, early])
```

Appendix C

Code from chapter Object Detection

C.1

```
1 let configuration = ARWorldTrackingConfiguration()
2 guard let detectingObjects =
      ARReferenceObject.referenceObjects(inGroupNamed: "Objects",
      bundle: nil) else { return }
3 configuration.detectionObjects = detectingObjects
```

Appendix D

Code from chapter One Stage Detector

D.1

```
1  import turicreate as tc
2  tc.config.set_num_gpus(0)
3
4  # Load the data
5  train_data = tc.SFrame("Train_Data.sframe")
6
7  #Random split train data to get specific training size
8  train_data, unused_data = train_data.random_split(0.3)
9
10 test_data = tc.SFrame("Test_Data.sframe")
11
12 # Create a model
13 model = tc.object_detector.create(train_data)
14
15 # Evaluate the model and save the results into a dictionary
16 metrics =
       model.evaluate(test_data,metric='mean_average_precision')
17 print(metrics)
18
19 # Save the model for later use in Turi Create
20 model.save('Nolmyra030.model')
21
22 # Export for use in Core ML
23 model.export_coreml('Nolmyra030.mlmodel',
       include_non_maximum_suppression=False)
```

D.2

```
#Load test data
test_data = tc.SFrame("Test_Data.sframe")

#Load created model
model = tc.load_model('Nolmyra.model')

# Save predictions to an SArray and draw predicted bounding boxes
predictions = model.predict(test_data)
predictions_stacked = tc.object_detector.util.stack_annotations(predictions)
image_prediction = tc.object_detector.util.draw_bounding_boxes(test_data['image'], predictions)

#Look through the predicted bounding boxes
image_prediction.explore()
```

D.3

```
    private func filterOverlappingPredictions(unorderedPredictions: [Prediction], nmsThreshold: Float) -> [Prediction] {
        var predictions = [Prediction]()
        let orderedPredictions = unorderedPredictions.sorted { $0.confidence > $1.confidence }
        var keep = [Bool](repeating: true, count: orderedPredictions.count)
        for i in 0..<orderedPredictions.count {
            if keep[i] {
                predictions.append(orderedPredictions[i])
                let bbox1 = orderedPredictions[i].boundingBox
                for j in (i+1)..<orderedPredictions.count {
                    if keep[j] {
                        let bbox2 = orderedPredictions[j].boundingBox
                        if bbox1.IoU(other: bbox2) > nmsThreshold {
                            keep[j] = false
                        }
                    }
                }
            }
        }
        return predictions
    }

    extension CGRect
```

```
24 {
25     func IoU(other: CGRect) -> Float
26     {
27         let intersection = self.intersection(other)
28         let union = self.union(other)
29         return Float((intersection.width * intersection.height) /
       (union.width * union.height))
30     }
31 }
```

Appendix E

Code from chapter Object Tracking

E.1

```
1  // Project path:
2  //
       master-thesis/Application/ObjectDetectionInAR/Assembler/ObjectTracker.swift
3
4  func track()
5      {
6      // Init all variables
7          cancelTracking = false
8          var trackingObservations = [UUID:
       VNDetectedObjectObservation]()
9          var trackedObjects = [UUID: ObjectRectangle]()
10         let requestHandler = VNSequenceRequestHandler()
11         let boundingFrame = delegate?.getBoundingFrame()
12
13         // Add the objects to track to the created lists above
14         for object in objectsToTrack
15         {
16             let observation =
       VNDetectedObjectObservation(boundingBox:
       object.getNormalizedRect(frame: viewFrame))
17             trackingObservations[observation.uuid] = observation
18             trackedObjects[observation.uuid] = object
19         }
20
21         // Loop over until a cancel tracking request is made
22         while true
23         {
24             if cancelTracking { break }
25
26             var rects = [ObjectRectangle]()
27             var trackingRequests = [VNRequest]()
28
```

```
29              guard let frame = delegate?.getFrame() else {
30                  usleep(useconds_t(millisecondsPerFrame * 1000))
31                  continue
32              }
33
34              for trackingObservation in trackingObservations
35              {
36                  // Create the requests
37                  let request =
    VNTrackObjectRequest(detectedObjectObservation:
    trackingObservation.value)
38                  request.trackingLevel = .fast
39                  trackingRequests.append(request)
40              }
41
42                      // Perform the requests
43              try? requestHandler.perform(trackingRequests, on:
    frame, orientation: CGImagePropertyOrientation.up)
44
45              for processedRequest in trackingRequests
46              {
47                  // Handle the results from the requests
48                  guard let observation =
    processedRequest.results?.first as?
    VNDetectedObjectObservation else { continue }
49
50                  if observation.confidence > confidenceThreshold
51                  {
52                      guard let object =
    trackedObjects[observation.uuid] else { continue }
53                      // Set new bounding box
54                      object.setNewBoundingBox(newBoundingBox:
    observation.boundingBox, frame: boundingFrame)
55                      trackedObjects[observation.uuid] = object
56                      trackingObservations[observation.uuid] =
    observation
57
58                      rects.append(object)
59                  }
60              }
61
62              DispatchQueue.main.async {
63                  rects = rects.sorted { $0.name! < $1.name! }
64                  self.delegate?.trackedRects(rects: rects)
65              }
66
67              // The tracking will stop if no observation has a
    high confidence value
68              if rects.isEmpty
69              {
70                  DispatchQueue.main.async {
71                      self.requestCancelTracking()
72                      self.delegate?.trackingLost()
```

```
73                  }
74              }
75
76              usleep(useconds_t(millisecondsPerFrame * 1000))
77          }
78
79          DispatchQueue.main.async {
80              self.delegate?.trackingDidStop()
81          }
82      }
```

Appendix F

Code from chapter The Finished Application

F.1

```
1       var furniturePartNodes = [SCNNode]()
2
3           for object in model.foundObjects
4           {
5               guard object.name != nil else { return }
6               guard object.position != nil else { return }
7
8               let furnitureNode = addFurniture(part: object.name!,
    position: object.position!)
9               furniturePartNodes.append(furnitureNode)
10          }
11
12          var previousNode: SCNNode? = nil
13          var previousAnchorPoint: SCNNode? = nil
14
15          var nodeActions = [(SCNNode, [SCNAction])]() // A list
    for storing animations to run on a node later
16
17          for node in furniturePartNodes
18          {
19              var actions: [SCNAction] = []
20              actions.append(SCNAction.rotate(by: -CGFloat.pi / 2,
    around: SCNVector3(0, 0, 1), duration: 1))
21
22              let anchorPoint = node.childNode(withName:
    ANCHOR_POINT, recursively: true)
23
24              if anchorPoint == nil
25              {
26                  if previousAnchorPoint != nil
27                  {
```

```
28                      actions.append(SCNAction.move(to:
    previousAnchorPoint!.worldPosition, duration: 2))
29                  }
30
31                  previousNode = node
32              }
33              else
34              {
35                  previousNode?.runAction(SCNAction.move(to:
    anchorPoint!.worldPosition, duration: 2))
36                  if previousAnchorPoint != nil
37                  {
38                      actions.append(SCNAction.move(to:
    previousAnchorPoint!.worldPosition, duration: 2))
39                      actions.append(SCNAction.move(by:
    node.worldPosition.substract(other:
    anchorPoint!.worldPosition), duration: 2))
40                  }
41
42                  previousAnchorPoint = anchorPoint
43              }
44
45              // HACK: Adds an extra action with no content at the
    end to make completion handler wait until the last action is
    done
46              actions.append(SCNAction.move(by: SCNVector3Zero,
    duration: 1))
47
48              nodeActions.append((node, actions))
49          }
50  }
```

Master's Theses in Mathematical Sciences 2018:E78
ISSN 1404-6342

LUTFMA-3371-2018

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/