A Visual Inventory System for Bakeries Trained with Synthetic Data

Hannes von Essen[1] **and Roman Naeem**[1]

[1]Department of Electrical Engineering, Chalmers University of Technology

November 2020

Göteborg, Sweden

CHALMERS
UNIVERSITY OF TECHNOLOGY

A Visual Inventory System for Bakeries Trained with Synthetic Data

Hannes von Essen, Roman Naeem

Department of Physics
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Department of Physics
Göteborg, Sweden 2020

Abstract

Deep learning systems have great potential in automating retail stores. However, collecting and annotating large amounts of training data for such systems can require substantial manual effort. Synthetical generation of such data could therefore save time and effort if it is similar enough to the real data to be useful. We explore this by designing an automatic inventory system and training it on synthetic datasets generated with path-traced rendering of 3d-scanned models. The object detection model trained on only synthetic data achieves 0.885 mAP50 score on our test set, reaching relatively close to the score of 0.907 for a model trained on just the real data, however, the model trained on both the real and synthetic data surpasses both of these scores with a score of 0.919. Finally, we also develop an approach to cluster visually similar objects into clusters using a novel similarity classifier that outperforms the present Siamese networks with a 17% performance improvement in our setting. The combined detection and clustering system allows us to count different types of breads in an image without predetermined classes.

Keywords: deep learning, neural networks, cascade mask r-cnn, synthetic data, domain randomization

Contents

1 Introduction

With the advances of deep learning in the last few years, more and more sophisticated computer vision models are becoming available, with possible applications for object detection, tracking and segmentation across a wide range of industries.

One industry where this type of system could be applied is in supermarkets, where it could be used for automatic shelf inventory or other types of product tracking. Some actors such as Amazon with their cashier-less supermarkets (*Amazon Go*) are completely embracing this technology in all ways possible while most supermarkets have yet to make use of any such systems at all.

One common barrier to adopting machine learning models is the amount of work required to collect and annotate the data for training. In some domains, the use of computer-generated data has shown promise [1]. For instance, [2] trained a road segmentation system using images from the computer game *GTA V*.

In this work, we design a computer vision system for automatic shelf inventory in supermarkets, and explore the application of computer-rendered data on this system. Specifically we focus on the bread section of supermarkets, as bread is a product with low shelf-life, where tracking the stock can help in planning how much to bake so as to reduce waste.

We further include a comprehensive study of object detection models in the Region Based Convolutional Neural Networks (R-CNN) and You Only Look Once (YOLO) families. Cascade Mask R-CNN [3], the most accurate model among the studied models is used to investigate our various datasets.

The object detection model is paired with our novel clustering system, that can cluster similar bread objects into clusters without predetermined classes, providing us with a generalized system that can count the number of each kind of bread in any given image. The similarity classifier, which is the main component of the clustering system, uses a simple approach to predict a similarity score for the two input objects, and significantly surpasses other systems like Siamese networks on our dataset.

1.1 Aim

The aim of the project is to develop a computer vision system for automatic inventory of products in a supermarket, and to investigate the performance of using a synthetically generated dataset to train our system. In this thesis, we limit the scope of the project by focusing only on bread shelves.

The project has the following two main objectives.

- Design a system that can take an image as an input, detect all the bread

objects, and give a count for each type of bread object by clustering similar objects as in Figure 1.

Figure 1: Input (top) and output (bottom) of the bread inventory system.

In our case, we don't know exactly the types of bread that can appear in our input images. Therefore, to get a count for each type of bread, we cannot just train an object detector with classes equal to the number of bread types that appear in the input images and count the detections. To get a count for each kind of bread without predefined classes, we must also implement a similarity classifier that can predict whether the two input bread objects are of the same kind or not, and use these predictions to cluster breads belonging to a single type.

- Design a system for generation of synthetic training images as shown in Figure 2 using modern 3d rendering techniques, and investigate to what

extent this type of generated data can replace or complement real training data. Also collect and annotate a real dataset as shown in Figure 3 to compare its performance with the synthetic dataset.

Figure 2: Sample image of synthetic dataset.

Figure 3: Sample image of real dataset.

1.2 Scope and Limitations

The original plan was to design a system of visual inventory of shelves in a supermarket, however, due to presence of thousands of different kind of products, and huge test setting, the time required for development and testing of such a system is out of the scope of this project. Therefore, the use case of our inventory system has been limited to only bakery products, to be able to conduct comprehensive experiments with our system.

The evaluation of our system is performed using 204 real test images that are manually annotated and put into different classes. The object detection is evaluated using the metric mAP50 and the clustering is evaluated using the metric Adjusted Rand Index (ARI).

In addition to evaluating the performance of the complete system, the effect of various parameters of the synthetic data are evaluated, specifically:

- The number of different light scenarios
- The number of different bread models

1.3 Related Work

Deep learning-based systems for inventory have been explored in many previous works. The general challenges in these works are the large number of product classes typically prevalent in supermarkets, the often very small details that make up the difference between different products, and the lack of more than a few studio images of each product as training data. Many of the previous works therefore present methods for classification using a single example per class ([4], [5]). In our approach we essentially use zero examples per class. Instead, we cluster the products based on their similarity to each other, and report the count for each cluster.

Using synthetic data to augment learning has been explored previously, for example in [6], and there are many synthetically generated datasets available for different computer vision tasks [1]. While most previous works have relied on real-time simulations such as Unreal Engine [6], we choose to go with a more realistic physics based rendering, path-tracing, which can achieve higher fidelity at the cost of longer rendering times.

1.4 New Contributions

Our contributions include

- A new similarity classifier where two images are tiled side by side and fed into a single network rather than two different networks which in our case gives a higher performance than Siamese networks.

- The application of example-based texture synthesis [7] on the texture level rather than on the rendered image, resulting in style-transferred 3d models that can be reused in multiple camera angles.

2 Theory

A brief overview of some of the concepts and a description of the models and algorithms used by our system is provided in this section.

2.1 Object Detection

The major deep learning breakthroughs in computer vision seen in the last years stem from the introduction of convolutional neural networks (CNN's) [8], in combination with the evolution of large enough computing resources. By sharing weights across different parts of the image, CNN's made it possible to reduce the number of parameters and made the training of deep neural networks for **image classification** possible on modern computers. In image classification, a single image is taken as input and classified into belonging to one of several classes (often thousands).

Much of the work since this success has focused on how to make more nuanced and complete predictions. Rather than doing one prediction per image, **object detection** models try to find all the objects in an image and return their bounding boxes and class predictions, and **instance segmentation** models additionally predict the pixel-precise mask of each found object. There are a number of proposed architectures for these, and most of them build upon the original image classification networks by including them as part of the model. They are then usually referred to as the *backbone* of the network, and are used to extract feature maps as a base for further computations.

In this theory section we will briefly go through convolutional neural networks and then go through some of the most popular image classification, object detection and instance segmentation methods.

2.1.1 Neural Networks

A neural network is a mathematical model that represents a series of computations of interconnected units called neurons. A neuron is a function with one or more than one inputs and a single output. In a neural network, each input to a neuron is the output of another neuron, except for the neurons at the beginning of the network – these take as input the values provided by the user, such as pixel intensities in an image, which we want the network to reason about.

Neurons in a neural network are conceptually similar to biological neurons, which similarly take in the signals from multiple other neurons (via dendrites), while giving off a single signal (via the axon). This single output value can then be delivered as input to multiple other neurons.

While in a biological neural network the neurons can seem to be connected in a chaotic way with for example cyclical paths, neural networks in machine learning are a bit different in that they typically have a clear layered structure.

Each neuron of the first layer of neurons take in all the values of the input. In the second layer of neurons, each neuron takes in all the values of the first layer of neurons, and so on. The number of connections is thus limited by the fact that a layer only takes input from the previous layer.

The function that each neuron performs on its input values to produce its single output is also analogical to the biological counterpart. Each incoming connection has a weight, so that the neuron is influenced more by some neurons than by others. This weight can be both positive and negative, so a incoming neuron can both contribute to the output and suppress the positive contributions of other neurons. The weighted incoming neuron activations are summed up into a value z. Instead of simply using this sum as the output, a non-linear function is applied to it, called the activation function. A typical activation function is the ReLU function shown in Figure 4, and mathematically given as

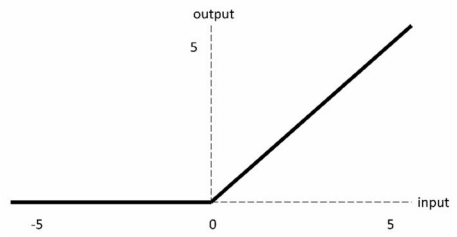

Figure 4: The ReLU function. Image reference: [9]

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The activation function makes the model more interesting by introducing a non-linearity, and has a clear analogy to biological neurons here as well: In a biological neuron, a neuron will only fire if it is stimulated enough to reach a certain threshold, otherwise it does nothing. Similarly, the neuron in machine learning only gives a non-zero output if the weighted sum of inputs passes a certain threshold, which in this case is 0. In order to allow other thresholds than zero, there is an additional bias parameter for each neuron which is added as a constant to the weighted input before the activation function.

The layered structure of the network allows initial layers to build up more refined information for the next layer to process, and by repeatedly building on earlier concepts and processing them further, the layers can represent more and more complex concepts in a hierarchical manner. By providing examples of desired input and output pairs to the network (when the network is given the values x as input to the first layer, the last layer should output the values y), the parameters of the network (weights and biases) can be mathematically optimized using

gradient-based techniques to produce the desired behaviour. This is possible because the entire network can be viewed as one large mathematical function, which is differentiable because all its components are differentiable.

2.1.2 Convolutional Neural Networks

The goal of convolutional neural networks (CNN's) is to extract image features useful for tasks such as classification and segmentation. While regular neural networks could also be used for this, the large number of input pixels from an image often makes it infeasible to train such networks. CNN's therefore introduced a few tricks to reduce the number of parameters such as having multiple neurons in the same layer share weights, making training possible on modern computers. As in regular neural networks there is a hierarchical layered structure, and CNN's have been inspired by how the visual cortex in a human brain works, where the raw light input is first processed to detect simple edges and colors in the initial layers, patterns and textures in intermediate layers and parts of objects or entire objects in the last layers.

In practice, a CNN pipeline consists of an input layer, then network layers, and finally the output. An input in computer vision is usually an image with a certain width, height and number of channels. The image will have a single channel if it's a black and white image, 3 channels (red, blue, green) if it's a colored image, or more than 3 channels if more information such as depth is included. The input is in the form of a matrix with dimensions width x height x channels, and values ranging from 0 to 255 (or 0.0 to 1.0). Such inputs are usually normalized by subtracting the mean from the original values and then dividing by standard deviation of the pixels. This makes the mean of all pixels 0 and values that are one standard deviation away from the mean 1. These normalization processes help the network in learning the features using a single learning rate as the distribution of feature values across different images becomes more similar after normalization [10]. The input obtained after performing these processes is referred to as the preprocessed input.

This preprocessed input is then fed to the network layers. There are different kinds of layers that perform different operations in a CNN. The most important type of layer is the convolutional layer. It consists of a filter/kernel that slides over the input to that layer, and at every position performs element-wise multiplication between the input and the filter followed by summation as shown in Figure 5. This is how weights are shared in a CNN – the same grid of kernel weights are used for each placement of the kernel on the image. The number of weights are also greatly reduced by the fact that each neuron in the next layer is only connected to the neurons inside the corresponding kernel window of the previous image rather than to all neurons of the previous image. There are various kinds of convolutional layers that can result in output of size less, greater than or equal to the input depending on for example how many pixels the sliding window moves in each step, which is called the *stride* of the convolutional layer.

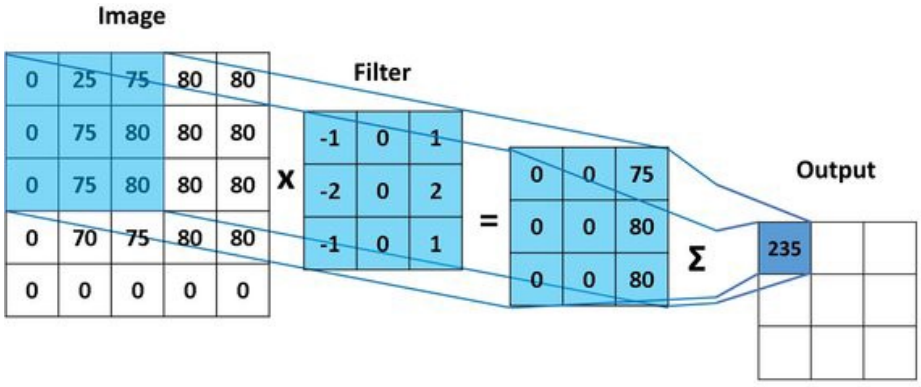

Figure 5: Example of convolution operation on an input. Image reference: [11]

The max pooling layer is another important type of layer, which picks the the max value from a given window as shown in Figure 6. It is used to downscale the input and reduce the number of parameters.

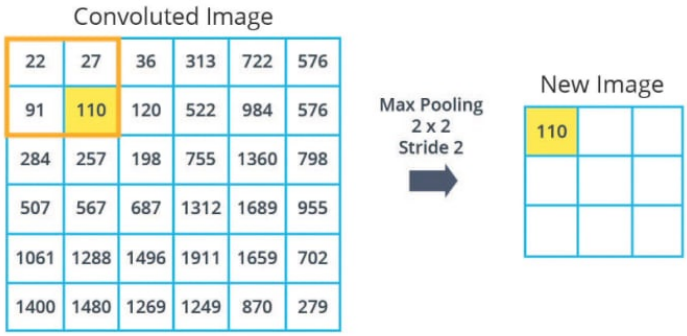

Figure 6: Example of max pooling operation on a an input. Image reference: [12]

The input after getting passed through a sequence of convolutional and max pooling layers is finally fed to a single or more fully connected layers (FCNs), i.e. regular neural network layers. An FCN flattens the input before processing it as shown in Figure 7. In image classification, the output layer has one neuron for each possible class, where the values can typically be interpreted as the network's certainty that the image belongs to the given class. If for instance the second output neuron represents the correct class, then the ideal output would

be 01000.... Other outputs are also possible, such as the positions of bounding box corners.

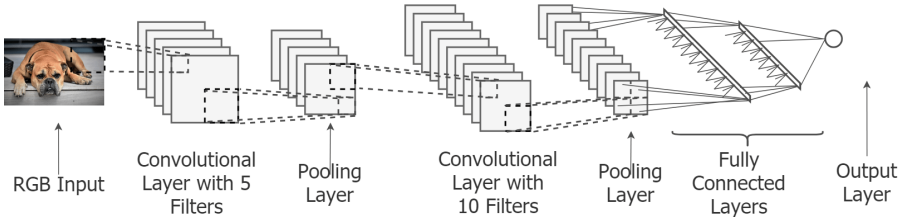

Figure 7: Example of a convolutional neural network pipeline. Image reference: [13]

2.1.3 ResNet

Intuitively, in a vanilla CNN, adding more layers should always give either better or at least the same result as the lower layered counterpart, since in the worst case scenario the additional layers can just act as identity layers. However, experiments shows saturation and degradation of training accuracy when the number of layers is increased [14, 15]. The ResNet [16] paper shows that the problem arises from the fact that in a deep network, even learning the identity function becomes an extremely difficult task and the network requires a lot of time to converge to that solution.

ResNet solves this problem by introducing Identity Shortcut Connections (IDCs) shown in Figure 8, between the input and output of a layer or a block of layers. Such shortcuts have been used before by the Highway network [15], which has gated shortcuts through which direct flow of information across layers can be controlled, however they have been refined in ResNet resulting in better performance.

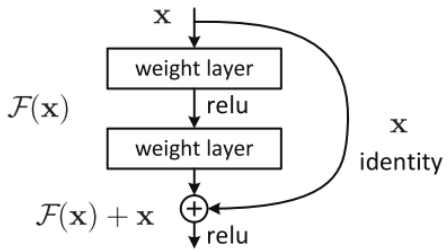

Figure 8: An identity shortcut connection. Image reference: [16].

IDCs take advantage of the fact that it is much easier to learn a zero function than an identity function by non linear layers and since using these shortcuts the initial problem of solving $f(x)$ becomes $g(x) = f(x) + x$, learning $f(x) = 0$ gives

us $g(x) = x$, the identity function. The network ResNet got its name from the fact that we can also look at the problem as an attempt to learn $f(x) = g(x) - x$, where $f(x)$ is the residual. The full ResNet architecture is show in Figure 9. The ResNet is used for parts of networks that will be discussed in the following sections.

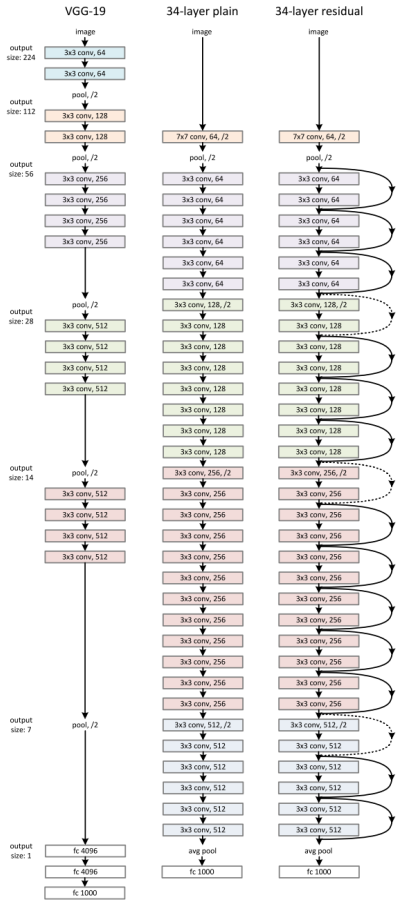

Figure 9: ResNet model architeture in comparison with VGG-19 model [17] and a plain network. Image reference: [16].

2.1.4 R-CNN

In 2013, a big leap in object detection came with R-CNN (Regions with CNN features) [18]. This model performs object detection in three steps. The first step is region proposal, followed by feature extraction for proposed regions, and finally a Support Vector Machine (SVM)-based [19] object classification using these features. Optionally, regression for adjusting the proposal size and location can also be performed which improves the accuracy by 4% but it is

not a part of the original R-CNN design. R-CNN is also an example of a two stage object detector where the first stage constitutes of the region proposal and the object classifier forms the second stage. Training in R-CNN is a two stage process (three stage if bounding box regression is present), where the first stage is fine-tuning the network using a log loss, followed by training the SVM classifier and lastly training the bounding box regressor, if applicable.

During region proposal, a manageable number (2000 in R-CNN) of proposed bounding boxes are generated by Selective Search (SS) [20]. At its base, the SS algorithm uses the concept of super-pixel straddling, which is grouping of regions of pixels with similar properties. In order to make the algorithm more generalized, multiple strategies are used to group pixels together in SS. The main strategies include grouping together based on the color, texture or composition. In Figure 10, pixels of the image of cats can be grouped based on colors, however in the image of chameleon grouping based on just color will be difficult, so in that case the texture of the pixels can be used instead. In the image of the car, the tire and the car body can not be grouped based on either color or texture – in such cases composition of pixels is used. For example, the tire and the windshield are contained in the car body, so they become a part of the car. This grouping is done in a hierarchical fashion. For example, in the image of the table, the table is the highest in hierarchy, the bowl is lower than the table and the spoon is the lowest. This proposal generation is fast and works for objects of all scales, and can produce 3574 regions in 3.8 seconds [20].

Figure 10: Example of images in which using different strategies in selective search aids in producing better object proposals. Image reference: [20].

R-CNN takes each of these image proposals, warps them to the size 227x227,

normalizes them and passes them through the CNN described in [8], which outputs a feature vector of constant size 4096. This feature vector is used by class-specific support vector machines to classify the object in the proposed region. The complete architecture of R-CNN is shown in Figure 11.

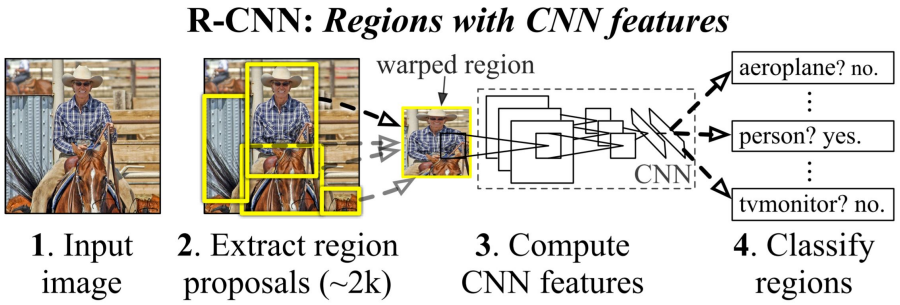

Figure 11: R-CNN model architecture. Image reference: [18].

2.1.5 Fast R-CNN

R-CNN was a breakthrough in terms of accuracy, but the fact that it had to classify 2000 regions for each image, which takes 47 seconds per image (excluding proposal generation) makes it unfeasible to be used in many real life applications. Furthermore, the three-stage training is very time consuming. Fast R-CNN [21] improves the architecture of R-CNN to overcome some of R-CNN's problems.

The first major change that improves the inference time significantly, is to feed the whole image to the feature extractor convolutional network once instead of feeding 2000 crops of the proposals from the image, while Selective Search (SS) [20] generates the region proposals for the image separately in parallel. These region proposals are projected to the output feature map of the feature extractor, a process called ROI (region of interest) projection. During ROI projection, the subsampling ratio (ratio of output size to input size for a CNN) is used to translate the coordinates of the ROI in the image to the ROI in the feature map. Fast R-CNN also introduced ROI pooling, replacing the last max pooling layer of CNN in R-CNN. In ROI pooling, the feature output of the CNN for each proposal is divided into a 7x7 grid and the max pooling operation is applied to each cell in the grid, giving us a constant-sized output. ROI pooling allows Fast R-CNN to take images of different sizes and aspect ratios, which increases the flexibility and accuracy of the model as the images are not required to be distorted if they are not square. R-CNN has a bounding box regressor as an optional component, while Fast R-CNN has it as a necessary part. Huber loss (smooth L1 loss) is used for the regressor in Fast R-CNN instead of L2 loss which improves and stabilizes the regression. Unlike R-CNN, where a separate log loss function is used for fine-tuning the network and an SVM as a classifier,

Fast R-CNN uses a single softmax classifier for both tasks which achieves same accuracy. Fast R-CNN also combines the classifier and regressor losses to make training of the network a one-stage process instead of a three-stage process as in R-CNN. The complete architecture of Fast R-CNN is shown in Figure 12.

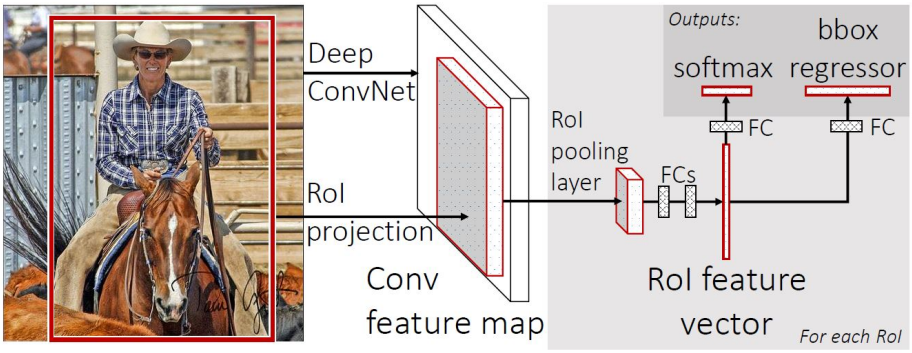

Figure 12: Fast R-CNN model architecture. Image reference: [21].

Fast R-CNN improves the accuracy over R-CNN by only a small margin from 66.0% to 66.9%. However, the significant improvement comes in inference time, which is 146 times faster than R-CNN at 0.32 seconds per image (excluding proposal generation). Fast R-CNN brought object detection much closer to real-time and opened up doors to many new applications.

2.1.6 Faster R-CNN

Selective Search (SS) [20] being a fixed algorithm does not require any training, so in some cases it can produce bad candidates for region proposals, and since it doesn't provide a measure of how good each proposal is, it is difficult to optimally reduce the number of proposal without affecting the recall of objects. We also cannot improve the time required by SS to generate these proposals. Faster R-CNN [22] solves these problems using a Region Proposal Network (RPN), which relies on a convolutional network to output a feature map and then perform dense sampling (using a sliding window) at each position of the feature map. Furthermore, to propose objects of different aspect ratios and sizes, and even overlapping objects in the same position, at each sliding window position nine different anchor boxes are used. These anchor boxes are the reference boxes to be used by the network to perform relative bounding box regression, and they have three different aspect ratios of 1:1, 2:1, 1:2 and three different scales of 128 pixels^2, 256 pixels^2, 512 pixels^2, giving us a total of nine anchor boxes. However, with just bounding box regression we will end up with too many proposals. For example, if an image of size 800x600 is fed to Alexnet [8], we get approximately 20,000 proposal, which is 10 times more than that of SS. To solve this problem RPN also includes a softmax classifier, that gives each proposal an objectness score, which is used to pick top 300 ranked proposal and

discard the rest. RPN can also handle images of different sizes and aspect ratios because the last fully connected layers have been replaced with convolutional layers that can handle variable sizes. The time required by RPN to produce proposals for a single image is 10 ms compared to 1510 ms by SS, which makes it 151 times faster [22]. Thus RPN can provide a lower number of high quality region proposals, while still being much faster than SS. An illustration of RPN is shown in Figure 13.

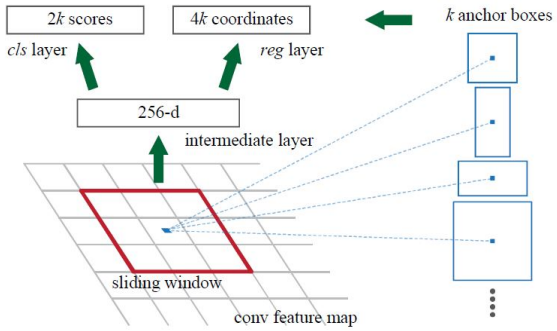

Figure 13: Region Proposal Network. Image reference: [22].

To improve the efficiency of the entire network, Faster R-CNN shares the convolutional network, also known as the backbone between the RPN and the rest of the network. This makes training the network a complicated four stage process, however the inference rate is 5 fps compared to 0.5 fps of Fast R-CNN. Since the paper [22] was published, a joint single stage training method has also been suggested for Faster R-CNN making the training process much simpler. The complete architecture of Faster R-CNN is shown in Figure 14.

2.1.7 Mask R-CNN

Mask R-CNN [23] is an extension of Faster R-CNN to output instance segmentation masks for input images in addition to object detection. There are three main differences between Mask R-CNN and Faster R-CNN, including replacement of ResNet by Feature Pyramid Network (FPN) as the feature extraction backbone, replacement of RoI pooling with the more accurate RoIAlign as the pooling operator and lastly adding an extra fully convolutional branch for generating instance segmentation masks in parallel with classification and regression heads as shown in Figure 15.

The Feature Pyramid Network is a powerful feature extraction network that has improved performance of many object detection networks. FPN utilizes a top-down approach to compute its output features. As shown in Figure 16, first, the input goes through a series of convolutional and down-scaling operations, where every layer reduces the size of the input by two times and doubles the

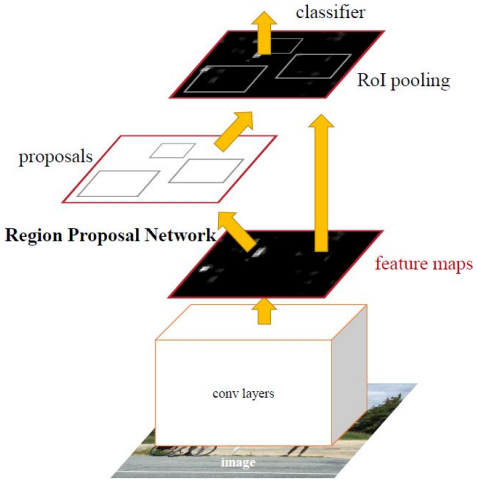

Figure 14: Faster R-CNN model architecture. Image reference: [22].

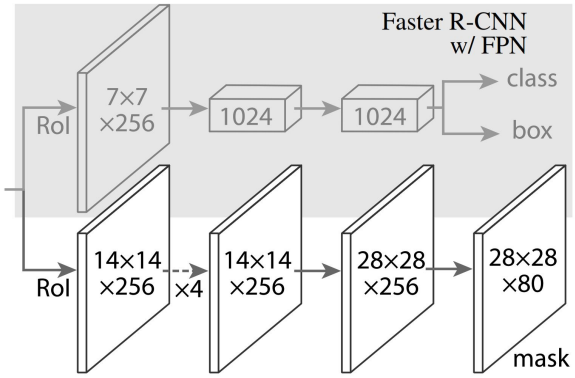

Figure 15: Head architecture of Mask R-CNN, which extends the Faster R-CNN with FPN heads with a fully convolutional branch to predict instance segmentation mask for each object. Image reference: [23].

number of channels. Then starting from the smallest feature map i.e. the output of the last layer, the number of channels is reduced to a smaller fixed number (256 in Mask R-CNN) using a 1x1 convolutional layer. This operation is shown as a lateral connection from the top left layer output to the top right layer output, and is performed on the output of each layer on the left side. The top lateral connection is followed by an up-scaling operation using a transposed convolutional layer (the opposite or reverse of a convolutional layer) increasing the output size by two. This output goes through element-wise addition with second-top lateral connection's output and produces the first final output of

the feature extractor. The up-scaling and element-wise addition with lateral connection outputs continues till the bottom layer to produce the final output feature maps for each layer. By using FPN as a backbone, in RPN we only use a single-scale proposal with three aspect ratios on each output of feature map, and objects of different sizes are taken care of by the five different sized outputs of FPN.

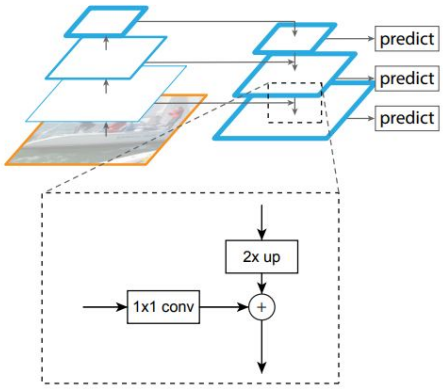

Figure 16: Feature Pyramid Network model architecture. Image reference: [24].

RoI pooling involve quantization steps that can provide good results in case of object detection but perform poorly for instance segmentation, since information about position is lost during the pooling operation, resulting in misaligned RoI's and extracted features. RoIAlign overcomes this misalignment by avoiding the quantization steps in RoI pooling by using a bi-linear interpolation. As shown in the Figure 17, the RoI can be placed anywhere on the feature and does not require quantization to match the grid of the feature map. The RoI is divided in 2x2 bins in the Figure (7x7 in Mask R-CNN) and each bin has 4 sampling points. The value of each sampling point is calculated using bi-linear interpolation from the values of the 4 closest boxes. This process keeps the information about the position of RoI in the feature map resulting in aligned RoI's and extracted features, making instance segmentation results significantly better.

2.1.8 Cascade Mask R-CNN

Past object detection models suffer from a degradation in performance when increasing the threshold for Intersection over Union (IoU) between the ground truth bounding boxes and the network predicted bounding boxes due to two main reasons. The first reason is the lack of positive samples during training, as most of the proposed samples are classified as negative due to the high IoU threshold, leading to over-fitting during training. The second reason is the mismatch between the IoU threshold used for training and the IoU threshold used for inference. This mismatch degrades performance because the network gives optimal performance only for the IoU threshold used during training. The

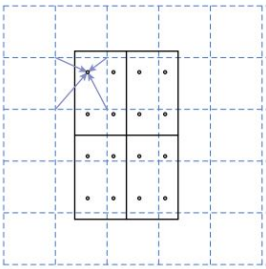

Figure 17: Illustration of RoIAlign operation, where the feature map is represented by dashed lines and the RoI by the solid lines with four bins and four sampling points in each bin. Image reference: [23].

architecture of Cascade R-CNN [3], a proposed network that tackles the above mentioned challenges, is shown in Figure 18.

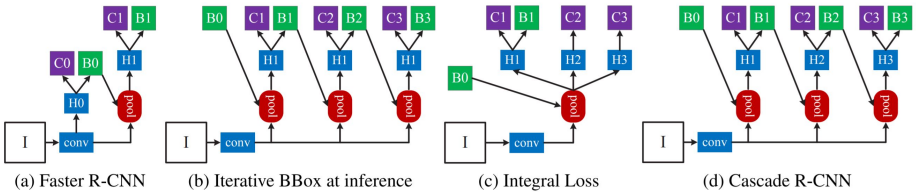

Figure 18: The architectures of four different networks. Here, 'I' represents the input image, 'conv' represents the backbone convolutional layers, 'H' represent the network heads, 'C' represents the classification output, 'B' represents the bounding box output. 'H0' is the RPN and 'B0' are the object proposals in each architecture. Image reference: [3].

Cascade R-CNN extends the Faster R-CNN architecture by adding additional network heads to perform classification and regression. These network heads work in series such that the output bounding boxes from the RPN are fed to the first head, and the output bounding boxes from the first head are fed to the second head and so on. Each head is trained on a single IoU threshold to perform optimally for that IoU threshold. In Cascade R-CNN [3], the first, second and third heads are trained stage by stage for IoU threshold of 0.5, 0.6 and 0.7 respectively. This stage by stage process removes the lack of positive samples since the proposal fed to heads with higher IoU thresholds are refined by the previous heads, solving the over-fitting problem. The problem of non-optimal results due to mismatch between train time and inference time IoU threshold is also solved since we have three network heads that are trained on three increasing IoU thresholds.

This proposed architecture significantly improved the performance of Faster

R-CNN and can also be used with Mask R-CNN to get even better instance segmentation results. During this project, we chose to use Cascade Mask R-CNN with ResNet-101 as the feature extractor as the main object detector for our experiments on different versions of our dataset.

2.1.9 YOLO

YOLO (You only look once) [25] treats the problem of object detection as a regression problem rather than a classification problem and it uses a single convolutional network to perform all the tasks involved in object detection making it a single stage detector unlike the R-CNN family. Using a single network provides much greater speed as convolution is done only once on the entire image. YOLO also makes less mistakes regarding the background, as convolution is done on the whole image, which allows it to encode contextual information about objects and their appearances.

YOLO works by dividing an input image of size 448x448 into an SxS grid, 7x7 grid in the YOLO paper [25]. Each grid cell is known as an anchor, which represents a classifier and is responsible for generating k bounding boxes around potential objects whose ground truth center falls in that grid cell and classify it as an object class. Given an image YOLO outputs width, height, coordinates of the center and confidence of the each object, along with probabilities of each class.

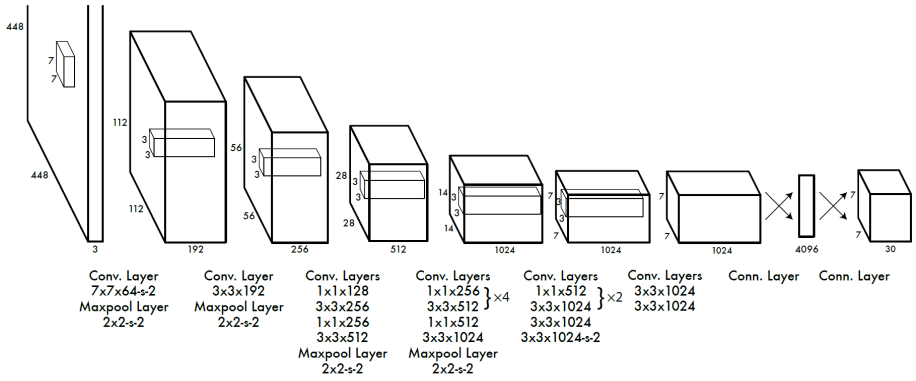

Figure 19: YOLO model architecture. The output is a 7x7x30 tensor, which is the result of SxSx(B*5+20), where S = 7 is the grid size, B = 2 is the number of bounding boxes per grid cell, and each bounding box has an associated width, height, center coordinates and confidence. The model also outputs single set of 20 class probabilities for each grid cell, which represents 20 classes of Pascal VOC dataset. At test time, each bounding box confidence is multiplied with corresponding grid cell class probabilities to get class probabilities for each bounding box. Image reference: [25].

Architecture of YOLO consist of CNN feature extractor that uses the Darknet framework [25]. It is pretrained on ImageNet and modified for detection by adding 4 convolutional layers and two fully connected layers on top as shown in Figure 19. The architecture is relatively simple compared to the two stage architecture of R-CNN family. Overall the network has 24 convolutional and 2 fully connected layers.

YOLO has some limitations that allows it to be fast but the resulting accuracy is low. YOLO struggles with detecting flocks of multiple small objects since each grid cell of input image can only predict two boxes. The model downsamples the input and uses coarse features to predict the bounding boxes. It also has difficulty dealing with objects with aspect ratios and configurations different than that of objects in the training data, making it harder to generalize. The loss function treats the errors in large objects and small objects equally, which means that a small object with less accurate bounding box will have a similar loss compared to more accurate bounding box on a large object, making the network focus more on large objects.

2.1.10 YOLOv2

YOLOv2 [26] is an improved version of YOLO by the same authors. YOLOv2 uses a variety of techniques to improve the performance of YOLO as shown in Figure 20. YOLOv2 uses anchor boxes to make the problem of localization easier by training the network to output only offsets to the anchor box location, and not the complete location, this reduces the mAP slightly but significantly increases the recall. The anchor box dimensions are usually picked by hand and the network can learn to adjust them to output bounding boxes, however, better anchor boxes dimension makes it more easier for the network to adjust the anchors. In YOLOv2, the process of finding anchor box dimension is automated by clustering the ground truth bounding boxes of training set using k-means and then picking the centroids' dimensions as the anchor boxes' dimensions. The clustering is done based on IoU instead of the euclidean distance. Using this dimension clustering, they pick k = 5 centroids' dimensions to be used.

YOLOv2 architecture uses DarkNet-19 as the feature detector, detailed in Figure 21. It has 19 convolutional layers and 5 max pooling layers. On top of these layers is a softmax layer for classification. It achieves a top-5 accuracy of 91.2% on ImageNet classification challenge improving on YOLO's 88% . It also uses multi-scale training to better detect images of different sizes. It uses fine-grained features which improves the average precision for small objects, making it comparable to SSD-300 [27] but still worse than Faster R-CNN. On Pascal VOC 2007 dataset YOLOv2 mean average precision (mAP) increases to 78.6 compared to 63.4 of YOLO.

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?							✓	✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figure 20: Improvements in YOLOv2 compared to YOLOv1 that lead to a significant performance gain and increased efficiency. Image reference: [26].

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 21: Details of DarkNet-19 architecture. Image reference: [26].

2.1.11 YOLOv3

After YOLOv2, the same authors released another version YOLOv3 [28] that includes further incremental improvements. In YOLOv3, for each bounding box

an objectness score is predicted using a logisitic regressor, which is 1 if an anchor box overlaps the ground truth bounding box the most compared to other anchor boxes. Unlike YOLOv2, YOLOv3 uses multiple separate logistic classifiers to classify the bounding boxes instead of a softmax classifier to get multi-class classification. To support detection at various scales YOLOv3 predicts boxes at three different scales allowing it to better predict objects of different scales. The feature Detector of YOLOv3 has also been improved and it uses a network called DarkNet-53, detailed in Figure 22.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 22: Details of DarkNet-53 architecture. Image reference: [28].

DarkNet-53 has 53 convolution layers with residual connections. It is more accurate than DarkNet-19 and more efficient than other bigger feature extractors such as ResNet-101 or ResNet-151. On top of this feature extractor three blocks of covolutional layers are added to predict at different scales as shown in Figure 23. The increased complexity of YOLOv3 makes it slower than YOLOv2.

The mentioned changes improved the mean average precision (mAP) for small objects and with increased mAP localization errors have been reduced. Detection of objects at different scales improved and the overall mAP has been significantly improved.

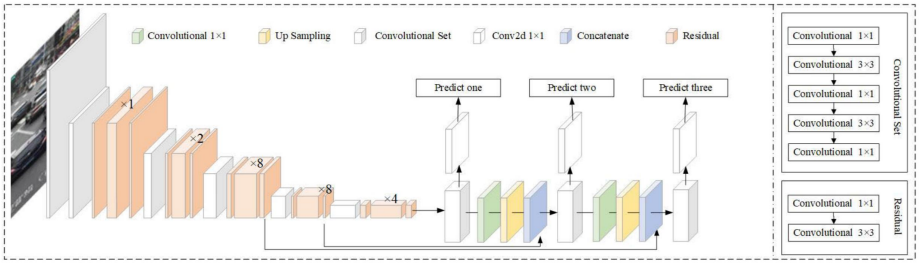

Figure 23: YOLOv3 architecture. Image reference: [29].

At the time of research and testing three version of YOLO (You only look once) [25] network had been published, and we used YOLOv3 for our testing. A fourth version [30] has been published recently, which further improves the accuracy of previous YOLO models, however it will not be described here.

2.1.12 EfficientNet

EfficientNet [31] is a recent convolutional neural network (CNN) that achieved state of the art performance on standard image classification benchmarks. The motivation behind EfficientNet as evident from the name is make CNN's more efficient. To achieve this goal, the authors approached the problem from two different directions. First, systematically scale up a small network by scaling up the width (number of channels), depth (number of layers) and input resolution (size) simultaneously in a way that gives better performance gain than if the these parameters were scaled independently. This method of scaling is called compound scaling where a single compound coefficient is used to scale all three hyper-parameters uniformly. The intuition behind this approach is that, for example if the resolution of the input image is increased, then width of the network should also be increased as there are more fine-grained features to be captured, and the depth of the network should be increased as well to get a larger receptive field to capture more of the larger input. Different types of model scaling methods are shown in Figure 24. Second, come up with a small efficient network that can be then scaled up using the above mentioned approach. For this purpose, Neural Architecture Search is used to design a new small baseline network called EfficientNet. This network is then scaled to 8 levels of complexities using compound scaling from EfficientNet-B0 to EfficientNet-B7.

The objective of scaling the network here is to maximize the accuracy of a baseline network with constraint of limited computing resources. To scale the network, as mentioned before, the width (w), depth (d) and input resolution (r) can be changed. One solution for finding the optimal values for w, h and r is to perform a grid search, that provides accuracy value for different combinations of these parameters. However, since for each combination of the parameters, we have to train and evaluate a network, performing an exhaustive search becomes

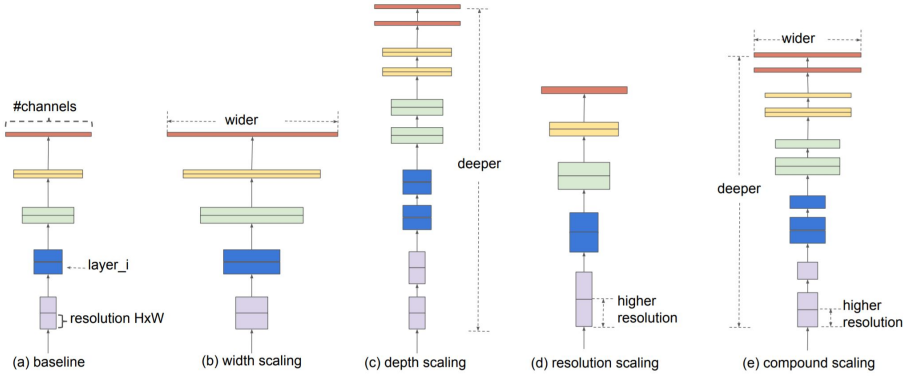

Figure 24: Illustration of different types of model scaling methods. (a) shows the baseline small network, (b) is scaling the width, (c) is scaling the depth and (d) is scaling the resolution of the baseline model. (e) is scaling the depth, width and resolution simultaneously using compound scaling. Image reference: [31].

very expensive, so we have to constraint our search.

As mentioned before, compound scaling is used to scale three hyper-parameters simultaneously, which is done by equating w, d an r to variables with a common exponent ϕ as in equation 2. ϕ is then changed to scale the network uniformly. However, we still have to find the values for the new variables α, β and γ. The lower bound of these variables are given as $\alpha \geq 1$, $\beta \geq 1$ and $\gamma \geq 1$. The upper bound is determined by taking into account the complexity of the network, which represents the limited computing resources. The complexity of a network is determined by computing the number of floating point operations (FLOPS). The complexity of the scaled network is related to the complexity of the baseline network as the ratio shown in equation 3, from which we can determine an upper bound on the α, β and γ by stating how much of complexity increase we can afford. The authors of the paper, chose $\alpha\beta^2\gamma^2 = 2$, which for $\phi = 1$ gives a network with twice the complexity of the base network. This choice provides a tighter upper bound for α, β and γ making the grid-search faster. Once optimal values for α, β and γ are found, then ϕ can be used to scale the network uniformly using

$$\begin{aligned} d &= \alpha^{\phi}, \\ w &= \beta^{\phi}, \\ r &= \gamma^{\phi} \end{aligned} \tag{2}$$

and

$$\frac{FLOPS(Network(\alpha,\beta,\gamma))}{FLOPS(Network(1,1,1))} = (\alpha\beta^2\gamma^2)^{\phi}. \tag{3}$$

Since the compound scaling does not affect the basic structure of the layers of the baseline network, it is important to use an efficient and accurate network as a base. For this purpose, the authors used their previous work MnasNet [32] as an inspiration, where the cost function optimizes for both Accuracy and Efficiency. The loss function is shown in equation 4 below,

$$loss = ACC(Network) \cdot [FLOPS(Network]/T]^{w}, \tag{4}$$

where ACC(Network) is the accuracy of the network and FLOPs(Network) is the complexity, T is the target complexity and $w = -0.07$ represents the trade off between complexity and accuracy. In contrast, to their previous work [32] they optimize complexity as opposed to latency. The result of the search is an efficient network called EfficientNet-B0. The main components of this network are mobile inverted bottleneck MBConv [33, 32] and squeeze-and-excitation optimizations [34].

2.2 Object Similarity

In order to decide whether two objects are of the same class there are different possible approaches. If the classes are few and well-defined, training a classifier is the standard option. However, for cases where the number of classes is very large or less well-defined, it may make sense to remove the notion of classes and instead focus on learning an embedding space such that object of the same type lie close to each other in the embedding space. While such an embedding is often a natural result of training a classifier CNN, there are methods that specifically try to encourage such an embedding space through the loss function.

These methods typically reframe the learning problem as follows: different inputs are fed to the same network (which can also be viewed as feeding the inputs to networks with shared weights, as shown in Figure 25. Their different embeddings output from the network are then compared to each other through a loss function. Two such loss functions are contrastive loss and triplet loss, which will be discussed below.

2.2.1 Contrastive Loss

In contrastive loss, there is a different loss function depending on whether we have a positive input pair (i.e. they are of the same type) or a negative input pair (they are of different types). The loss function is designed to pull positive pairs closer to each other and push negative pairs away from each other. Therefore the distance in embedding space is used as the loss for the positive pairs, while

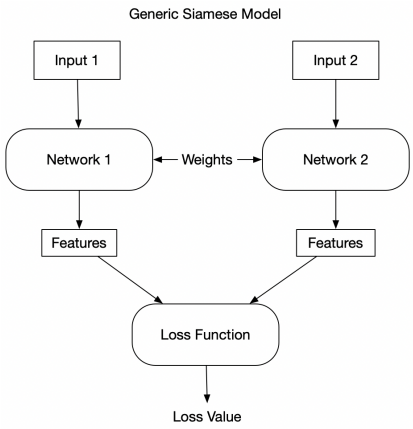

Figure 25: A siamese architecture, the basis of many *metric (distance function) learning* loss functions. Image reference: [35]

for negative pairs the negative of this distance is used. Additionally, a margin m is defined such that we only care about increasing the distance if distance is small enough; after negative pairs have moved enough apart (to a distance of m) we do not force them to continue further. Without this it would be harder for the system to reach an equilibrium as the two opposite goals could never be fully satisfied. Mathematically contrastive loss fucntion is given as

$$L(r_0, r_1) = \begin{cases} d(r_0, r_1) & \text{if } (r_0, r_1) \text{ is a positive pair} \\ \max(0, m - d(r_0, r_1)) & \text{if } (r_0, r_1) \text{ is a negative pair.} \end{cases} \tag{5}$$

2.2.2 Triplet Loss

In triplet loss, there are instead three different inputs: an anchor input a, a positive input p and a negative input n. The anchor can be viewed as a baseline to which the other two inputs are compared; the positive input is of the same type as the anchor, and the negative input is of a different type. The distance from the positive input to the anchor is minimized and the distance from the negative input to the anchor is maximized. In contrastive loss we stopped caring about the distance in negative pairs after a minimum margin m had been achieved. Here m has a similar function, although slightly different. It tells us how much larger we require the distance to be between the anchor and the negative, than between the anchor and the positive input before we stop caring (i.e. before the loss contribution becomes 0). If m was 0, then we would be happy as soon as the distance from the anchor to the negative input surpassed the distance to the positive input, while if it's higher we are not that easily satisfied and require the distance to the negative input to be larger by a certain margin. Mathematically, the triplet loss is given as

$$L(a, p, n) = \max\left(0,\ d(a, p) - (d(a, n) + m)\right). \tag{6}$$

2.3 Synthetic Data

One important concept in synthetic data generation is the technique of *domain randomization* [6]. The idea is that since it will be difficult to generate synthetic data that is perfectly similar to the real data domain, it is easier to instead make sure to include a lot of variation in the synthetic data so that the network will have to learn the essential features of the objects. By non-realistically varying the parameters of the simulator such as lighting, pose and textures it is hoped that the robustness to this variation carries over to the real data domain.

2.4 Example-based Texture Synthesis

There are many proposed methods for style transfer in images, many of which are neural network-based [36]. An interesting non-neural approach is proposed by [7]. By creating a patchwork using patches from the original style exemplar, they achieve a crisp high-resolution result compared to the often blurry results associated with neurally generated images. Another benefit of their method is that it does not require any training, and requires only a single style exemplar and two sets of guide channels as input.

In their paper, they have computer-rendered 3d models of two different objects. An artist has recreated the first object (a simple sphere), but in an artistic medium such as paint or charcoal. The goal of the algorithm is then to create an image of the second 3d model in this same artistic style. The algorithm thus has three parts of information available (see Figure 26): A (the source guide image), A' (the source stylized image) and B (the target guide image), and has to use these to find the fourth missing part B' (the target stylized image) such that the following analogy holds:

$$A : A' :: B : B'$$

i.e. such that A' is to A as B' is to B. The goal is then roughly to find similar patches between the guide images and then copy between these patch locations patches from the source style to the target style image. At the same time, we should also take into account the patches we have already drawn on the B' image so that the new patches fit into our current drawing.

In order to formalize this, let us first define a way to compare patches so that we can compute their dissimilarity. Let $I(p)$ be the square patch in image I with its center pixel at position p and a fixed size, where I is one of the images A, A', B, B'. We can represent $I(p)$ as a single vector of values by putting all the pixels of the patch in a single column in scan-line order, and flattening out the channels of each pixel, so that for example each RGB pixel contributes with 3 values to the vector.

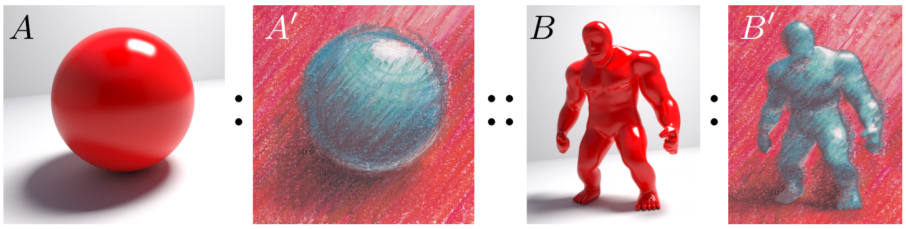

Figure 26: Illustration of image analogies. Image reference: [7].

The problem of finding these similar patches while taking both the aforementioned goals into account can be are expressed as follows: For each patch location $q \in B$, find the best matching patch location p in the source A such that the following is minimized:

$$E(A, B, A', B', p, q, \mu) = \mu \, ||A(p) - B(q)||^2 + ||A'(p) - B'(q)||^2$$

where μ is a weight that controls the balance between the first term, the fit of the patch positions in the guide images A/B, and the second term, the fit of the same patch positions in the stylized images A'/B'. Note that B' is the image that we are constructing, so as we continue to add patches to it the second term of E will change. The guide images are not limited to RGB images but can have any number of channels, in fact, in [7], multiple extra channels are used giving information about where specific types of light appear such as specular, diffuse and indirect light, see Figure 27. Also, instead of minimizing E directly, [7] adopt an expectation maximization-like iteration, which is executed multiple times from coarse to fine image resolution to improve the results.

2.5 Clustering Techniques

Our proposed system clusters bread objects detected in an image based on their similarity to each other, to get a count for each type of bread in the image. In our case, since we do not have a specific number of different types of bread, we needed an algorithm that does not require 'number of clusters' as an input and clusters objects into suitable number of clusters. For this purpose, three algorithms were researched and tested using their Scikit-learn [37] implementations.

2.5.1 Affinity Propagation

Affinity propagation [38] is a clustering algorithm based on a clustering scheme that uses messages passed between data points to establish which of the data points can act as the exemplars (most representative) for the rest of the data points, and then clustering the data points with the same exemplar into one

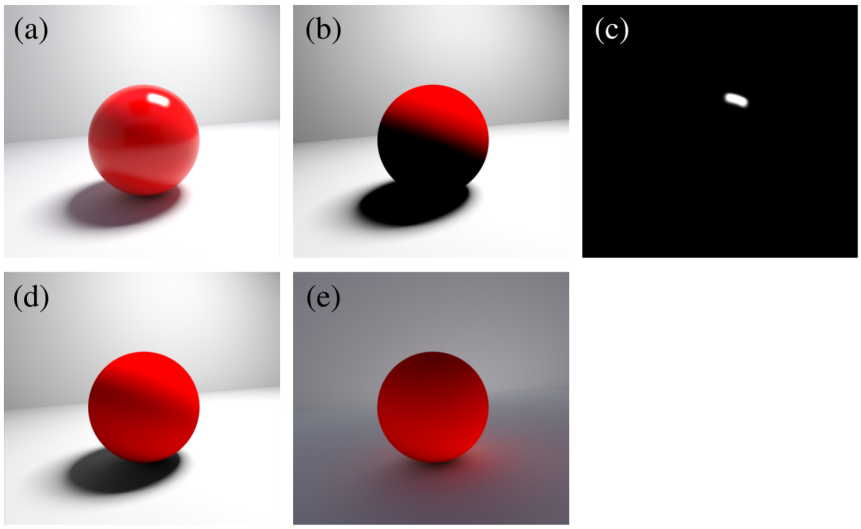

Figure 27: The different guide image channels used by [7]: (a) full global illumination render, (b) direct diffuse (LDE), (c) direct specular (LSE), (d) first two diffuse bounces (LD1,2E), (e) diffuse interreflection (L.*DDE) Image reference: [7].

cluster. Affinity propagation, unlike K-means or K-medoid, does not require the number of clusters (k) as input and determines the suitable number of clusters based on the data provided. This property of the algorithm makes it a suitable choice for our application.

There are four matrices that are used to compute the clusters in this algorithm. The similarity matrix (s) is an nxn matrix where n is the number of objects and $s(i, j)$ is the similarity between the instances i and j. Given the features of two objects x_i and x_j, their similarity can be computed as $s(i, j) = -||x_i - x_j||^2$. In our case, the values of this similarity matrix are computed by a similarity classifier network that assigns a value ranging from 0 (not similar) to 1 (similar) to the input object pairs based on their similarity. One thing to note about similarity matrix is to make the diagonal elements (similarity of an object to itself), which will originally have the highest values in the matrix, equal to the smallest value 0. This is done in order to get a lower number of clusters. For instance, in the similarity matrix, if we are looking in the ith row for the most similar jth object, we would most probably end up with $i = j$, and due to this mapping, the object i will map to itself, forming a cluster. This can change later as the algorithm performs more iterations but ultimately it would lead to a higher number of clusters.

The second matrix is called the responsibility matrix (r), where the value of

$r(i, j)$ indicates how qualified the object j is to be object i's exemplar, while taking into consideration the nearest competitor j'. The responsibility matrix is initialized as zero and it is computed as

$$r(i, j) \leftarrow s(i, j) - \max_{j \neq j'}(a(i, j') - s(i, j')). \tag{7}$$

Here a is the availability matrix that will be described next. In equation (7), the responsibility value $r(i, j)$ is computed as the similarity between the object i and object j, however, also taking into account the similarity of i to some other object j' and availability of j' to be i's exemplar, such that if i is more similar to j' and j' is more available to be i's exemplar then the responsibility of j to i is decreased.

The third matrix is the above mentioned availability matrix (a). $a(i, j)$ indicates how suitable it is for the object i to choose object j as its exemplar, considering its responsibility to itself and other objects. It is computed as

$$a(i, j) \leftarrow \min\Big(0, r(j, j) + \sum_{i' \notin \{i,j\}} max(0, r(i', j))\Big). \tag{8}$$

In equation (8), $a(i, j)$ is defined as the sum of j's responsibility to itself and j's responsibility to all the other elements excluding the element i, while ignoring negative responsibility values. The negative responsibility values are ignored as we are only interested in the suitability of j being an exemplar for some points and not in its unsuitability for other points. If j's responsibility to itself is negative, that indicates that j is not a suitable exemplar, and should belong to some other exemplar instead. $a(i, j)$ has the maximum value of 0. Self-availability $a(j, j)$ is computed differently as

$$a(j, j) \leftarrow \sum_{i' \neq j} max(0, r(i', j)). \tag{9}$$

The self-availability $a(j, j)$ quantifies the total evidence of j being an exemplar which is the sum of its positive responsibilities for all objects excluding itself.

The responsibility and availability matrices are updated in iterations. The updates are stopped when a certain threshold for changes during each update is reached, or when the values of the matrices become constant. This is the point of consensus.

The last matrix is called the criterion matrix (c). $c(i, j)$ represents the criterion used to determine whether the object j is used as an exemplar for object i. The criterion matrix is computed as

$$c(i,j) \leftarrow a(i,j) + r(i,j). \tag{10}$$

In each row of the criterion matrix, the object with highest value is selected as an exemplar for the object represented by that row and the objects that get the same exemplar are assigned to a single cluster. From equation 10, we see that for j to become an exemplar for i it should have both high responsibility and high availability for i. The process of affinity-propagation is illustrated in Figure 28.

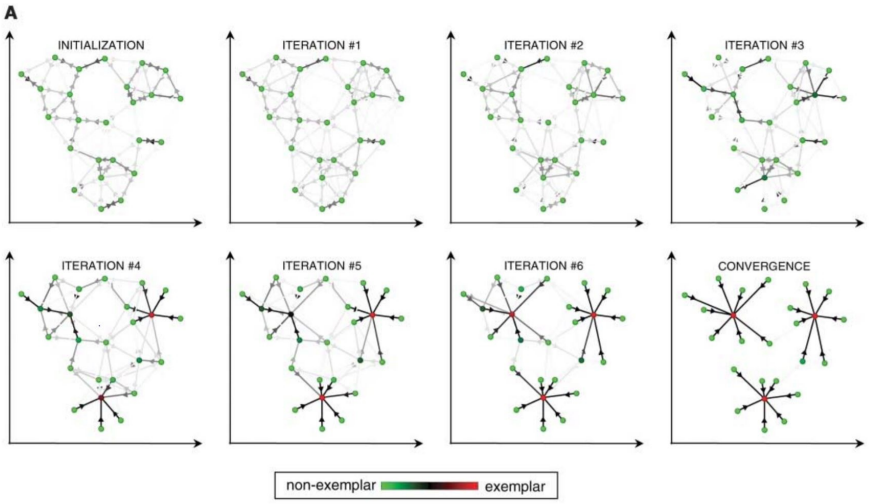

Figure 28: Illustration of the process of affinity propagation, where the darkness of the arrows from i to j represents how strongly i belongs to exemplar j, while the color of each point indicates how much that point is an exemplar. Image reference: [38]

2.5.2 Density Based Spatial Clustering of Applications with Noise (DBSCAN)

Clustering algorithms can be mainly classified into three categories namely, partition-based clustering, hierarchical clustering and density-based clustering. DBSCAN [39] is a type of density-based algorithm. Density-based algorithms are good at clustering data with arbitrary underlying structure, and are better at dealing with outliers compared to other types of clustering algorithms. DBSCAN proposed a cluster model as shown in Figure 29, in which a cluster is formed on the basis of the minimum number of points (minPts) in a radius ϵ of a certain point. If this criteria is achieved for a certain point then that point is assigned as a core point and the neighbors of the core point are assigned to the same cluster as the core point. The neighboring points are called direct density

reachable and if any of these points also becomes a core point, then its neighbors are also assigned the same cluster and are called density reachable. This process is repeated for all the points in a dataset, resulting in clusters of data points in areas of high density separated by areas of low density. This clustering scheme enables DBSCAN to capture arbitrary shaped clusters, and ignore outliers.

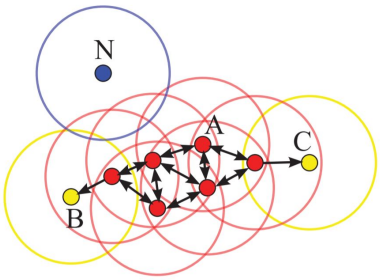

Figure 29: Illustration of the DBSCAN cluster model. In this model the parameter minPts is 4 and ϵ is indicated by the circles around the data points. The red data points such as A represents the core points, point B and C are the border points, and N is an outlier. Image reference: [40]

The algorithm to form clusters in the DBSCAN cluster model consists of following steps. First, minPts and ϵ are initialized to desired values. A random data point is selected and if there are not minPts points in its radius ϵ, the point is marked as an outlier. However, if the criteria is fulfilled, the point becomes a core point. In the next step, all the neighboring points are checked and marked as either core points or border points, based on whether the core point criteria is fulfilled. Once this process is finished, another point is randomly picked from the data points not yet tested, and the same process is repeated till each point is reached. A point initially assigned as an outlier, might become part of a cluster as the algorithm progress.

2.5.3 Mean Shift

Mean shift [41] is a centroid-based clustering algorithm and can be classified under the category of hierarchical clustering algorithms. Mean shift builds upon and uses the idea of kernel density estimation (KDE) [42, 43]. Given a dataset represented in a mathematical manner (as points), KDE can be used to estimate the probability distribution from which the dataset is sampled. This is done by putting a kernel (a weighting function) on each data point. The kernel in most cases is a Gaussian distribution and its width is called the bandwidth of the kernel. After putting the kernel on each data point, the kernels are summed up to get a probability distribution. Mean shift uses this probability distribution to form clusters, by pushing all the data points to the respective closest peaks of the estimated probability distribution. The bandwidth of the kernel affects the size and shape of clusters. The cluster formation is illustrated in Figure 30.

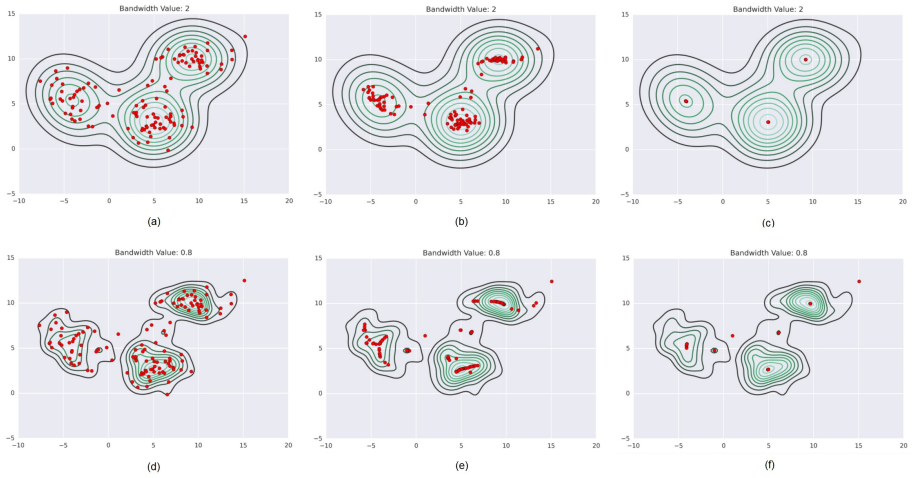

Figure 30: Illustration of Mean shift clustering. Panels (a), (b) and (c) shows the formation of clusters for bandwidth value 2, and (d), (e) and (f) for bandwidth value 0.8.. Image reference: [44]

The Mean shift algorithm to compute the clusters starts by making each data point a centroid. This followed by updating each data point x_i at time $t+1$ using the the equation

$$x_i^{t+1} = m(x_i^t). \tag{11}$$

Where m is a vector representing the direction of maximum increase in the probability distribution of the dataset. It is computed as

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}. \tag{12}$$

Here $N(x_i)$ is the neighborhood of data points around x_i, and K is the kernel. After multiple updates the centroids converge to the peaks of the probability distribution. At this point, nearly duplicate centroids are removed to get the final centroids.

2.6 Clustering Evaluation Measures

2.6.1 Homogeneity, Completeness and V-measure

An intuitive way of quantifying the quality of clustering of a given dataset can be achieved by analysing the conditional entropy of the clusters provided the

ground truth assignments are known. Homogeneity is the objective of each cluster consisting of only the samples that belongs to the same class, while completeness, in contrast is the objective that all the samples that belongs to the same class are clustered together [45]. Both homogeneity and completeness scores have values ranging from 0 to 1, with 1 being the best score. Mathematically, homogeneity (h) is given as

$$h = 1 - \frac{H(C|K)}{H(C)} \tag{13}$$

and completeness (c) is given as

$$h = 1 - \frac{H(C|K)}{H(C)}. \tag{14}$$

Here $H(C|K)$ represents the conditional entropy of all the classes in C, the set of all classes, provided all the cluster assignments in K, the set of all cluster assignments, and $H(C)$ is the entropy of all the classes. Mathematically, they are given as

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{N}\cdot\log\left(\frac{n_{c,k}}{n_k}\right) \tag{15}$$

and

$$H(C) = -\sum_{c=1}^{|C|}\frac{n_c}{n}\cdot\log\left(\frac{n_c}{N}\right). \tag{16}$$

Here, N represents the total number of samples, while n_c and n_k are the number of samples in class c and cluster k respectively. $n_{c,k}$ is the number of samples that are in both class c and cluster k.

For a good clustering, both the homogeneity and completeness should be as high as possible simultaneously, as it is easy to achieve perfect homogeneity and completeness independently by assigning each sample to a cluster containing only itself, and by assigning all the samples to the same cluster respectively. A high V-measure [45] score ensures both high homogeneity and completeness, and is computed by taking harmonic mean of both. Mathematically it is given as

$$v = 2 \cdot \frac{h \cdot c}{h + c}. \tag{17}$$

V-measure is one of the measures we used in our experiments to evaluate different clustering algorithms. However, V-measure has a drawback, especially in case of small number of samples, which is true in our case with only 12 bread objects per image. Homogeneity, completeness and consequently V-measure are not normalized with respect to random labeling, due to which for completely random cluster assignments we might obtain different scores. Therefore, in case of smaller number of samples, it is advised to use Adjusted Rand Index.

2.6.2 Adjusted Rand Index

Given the ground truth classes in set C and the cluster assignments in set K, obtained by using some clustering algorithm, for a dataset, we can use the Rand Index (RI) [46] to quantify the similarity of C and K. Mathematically, RI can be computed as

$$\mathrm{RI} = \frac{a + b}{C_2^{n_{samples}}}. \tag{18}$$

Here, a represents the number of pairs of the samples that belong to the same class in C and the same cluster in K, while b is the number of pairs of samples that belong to different classes in C and different clusters in K. $C_2^{n_{samples}}$ is the total number of possible pair combinations of all the samples in the dataset.

The original Rand Index also faces the same drawback mentioned for Homogeneity, completeness and V-measure. However, Adjusted Rand Index (ARI) [47] is a version of Rand Index that provides fix for random labeling such that any random clustering result gets a score close to 0. This is fix is generally known as corrected-for-chance. Mathematically, ARI is computed by disregarding the expected value of RI, $E[\mathrm{RI}]$, for random labeling from the RI as

$$\mathrm{ARI} = \frac{\mathrm{RI} - E[\mathrm{RI}]}{\max(\mathrm{RI}) - E[\mathrm{RI}]}. \tag{19}$$

3 Methodology

3.1 Generating Synthetic Training and Evaluation Data

The goal with the synthetic data was to make it realistic enough so that the knowledge learned from it could transfer to real images and to create a large enough variation in the data so that the model would not overfit.

We chose to use photo-scanned bread models to make the objects similar to real breads, and we chose to use physics based path-tracing rendering to make the light and shadows as realistic as possible. For other aspects such as the shelves themselves, we went more for creating a large variation than for realism.

3.1.1 Base 3d Models

A set of 31 3d models, based on scans of real breads, were purchased[1]. They can be further grouped into 17 bread types. Each model is built up from the following components, also shown in Figure 31 and 32:

- A *3d mesh* for the basic geometry of the bread. This also includes a mapping from a two-dimensional *texture space* or *UV space* to the 3d mesh; we will refer to this mapping as the *UV mapping*.
- A *color texture*. This is a 2d image defining the base color (albedo) of each point of the 3d model, through the *UV mapping*.
- A *normal map*. This is a 2d image defining small-scale geometrical details in the model through the *UV mapping*. Instead of defining a color for each point of the 3d model, it encodes the normal direction of each point. This affects how the simulated light rays bounce off the surface of the object, and creates the appearance of more geometrical details than what is defined in the 3d mesh itself. Each such normal direction is encoded as a pixel color in the image by treating XYZ coordinates as RGB channels.
- A *roughness map*. This is a 2d single-channel image defining the shininess of each point of the bread. For example, regions of pure crust are shinier than regions covered in flour. This affects the light paths in the light simulation of the renderer.
- An *ambient-occlusion map*. This encodes information about where occlusion shadows appear, and can be used to fake more realistic light simulations. We ignored this as we are using a physically based realistic renderer.

[1] https://www.cgtrader.com/3d-models/food/miscellaneous/baked-pastries-3d-pack

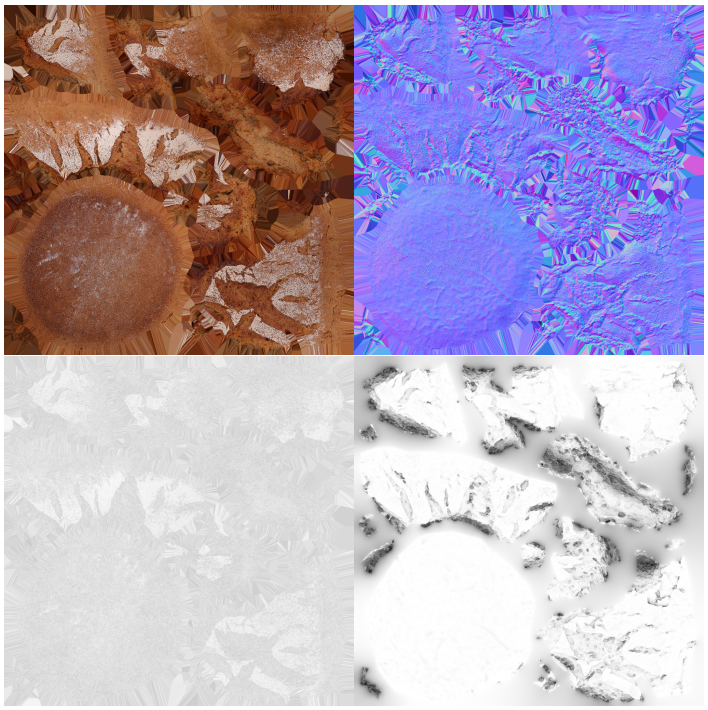

Figure 31: The different textures that describe the model surface. Top left: base color, top right: normal, bottom left: roughness, bottom right: ambient occlusion

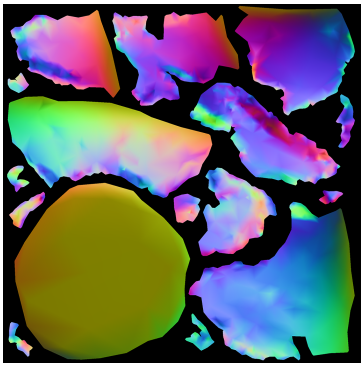

Figure 32: Additional normal map generated for the guided texture synthesis. Instead of specifying the normal relative to the original surface, the normal is specified as a global 3-dimensional vector. It is generated from the simplest version of the model to make it more smooth, as the details are already provided by the original normal map.

3.1.2 Data Augmentation - Shape

In order to add more variation to the models and make it harder for the network to simply memorize them, the vertices of the models were perturbed with noise in 3d space. We chose to use a sum of Perlin noises [48] with different frequencies to add smooth deformations on both large and small scales.

3.1.3 Data augmentation - Texture

As a further attempt at increasing the variation of the relatively few bread models available, a style transfer-like technique was employed. The idea was to be able to take two breads and create a new bread by combining aspects from both. Specifically, by taking the shape of one bread and the texture of the other. If one would naively use the texture image of the second bread for the first bread, the result would be poor as the UV mapping would not match the texture, see Figure 33. Rather we would like to synthesize a new texture with a similar layout to the original texture, where for example large cracks in the surface of the bread stay in the same place on the texture, but with a style similar to the other texture.

Figure 33: Notice how naively replacing the texture (left) introduces visible seams and a poor match between the geometry and color of the bread, while the synthesized texture (right) gives a more well-adapted look.

In order to do this we utilize the fact that we have textures not only for the color of the breads but also textures that provide geometrical hints: the normal map (showing precise geometrical details) and the ambient occlusion map (which is darker in for example cracks of the bread). We use the example-based texture synthesis of [7] with the color texture of the source bread as *style* channel and as *guide* channels we use the ambient occlusion map, the standard normal map (in local coordinates) and an additional normal map in global coordinates (we use this to make it easier to learn for example that some types of texture are only found on the top of the bread), see Figure 32. The roughness map was ignored as there was too much variation in it across different bread types. Figure 34 and 35 show the resulting breads.

Figure 34: Five breads and all breads synthesized by applying the style of one bread onto the shape of another. The original breads are found on the diagonal, each row is a shape and each column is a style.

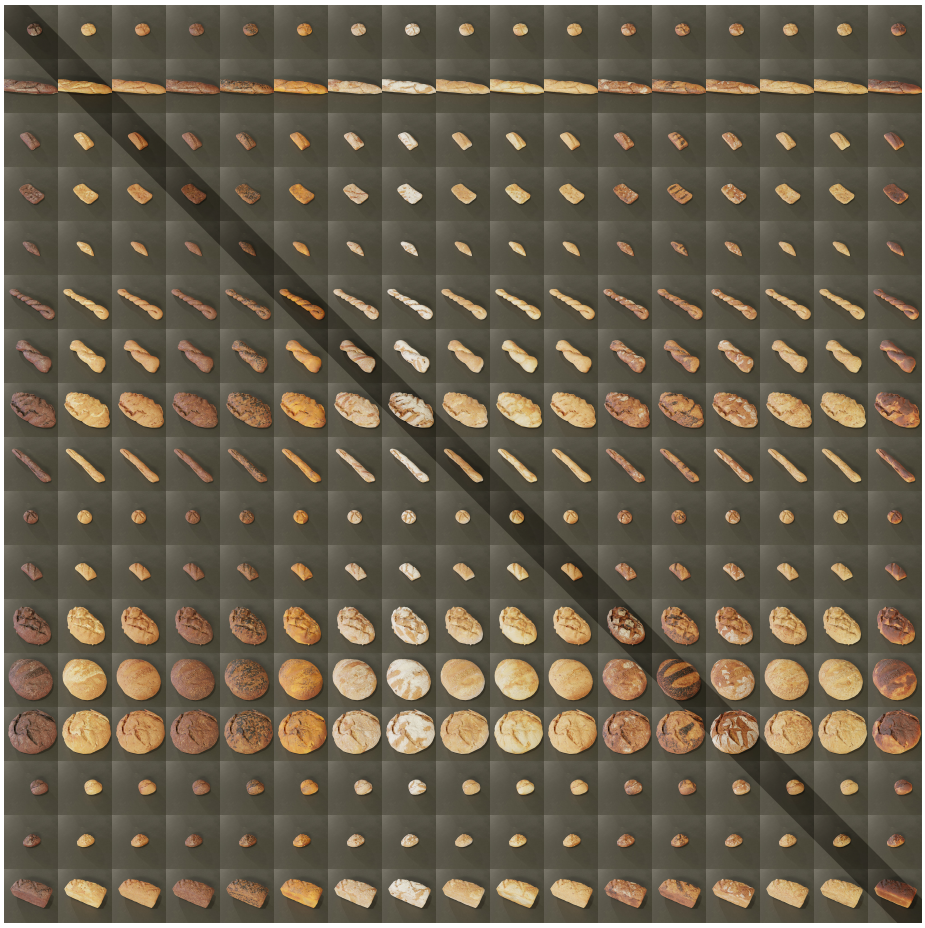

Figure 35: All bread types synthesized from the original 17 distinct bread types. Note that some bread types have multiple bread models, so that in total there are actually $31 \times 31 = 961$ breads. Here only one bread is shown for each pair of bread types.

3.1.4 Physics Simulation

Blender's built-in rigid-body physics engine was used to ensure a realistic placement of the breads.

The breads were first randomly positioned in a volume above the shelf, with random rotation around the up-axis, and gravity was then simulated until the breads had landed and stopped moving.

3.1.5 Light Setups

Rather than trying to recreate the exact light environments found in stores, we aimed at including a large variation of light setups as shown in Figure 36 to make the model robust to many types of lighting.

Figure 36: Four of the 104 HDRIs used for realistic lighting.

For each image, we randomly selected one out of 104 different light scenarios. Each light scenario is recorded from a real-life location and stored in an HDRI (High Dynamic Range Image). An HDRI is a 360° image that contains nuanced information about the intensity of the light across a wide range, and is typically created by combining information from multiple photos taken with different exposure settings. The HDRIs were taken from the indoor section of the open HDRI library *HDRI Haven*[2].

For extra variation, the HDRI intensity was multiplied by a random value between 0.7 and 1.3, and the light scenario was rotated randomly around the up-axis of the 3d scene.

3.1.6 Camera Setup

The camera was randomly positioned and oriented towards the center of the scene. A resolution of 640 by 640 was selected as a compromise between high resolution and fast rendering time.

3.1.7 Plane Materials

The initial shelf consisted of a single ground plane. The material of the plane was randomly chosen out of 74 images of shelf-like materials such as wood and

[2]https://hdrihaven.com/

metal, retrieved from the open texture collection *Texture Ninja*[3] shown in Figure 37.

Figure 37: Four of the 74 shelf textures.

The roughness property of the material (indicated how shiny it is) was randomly selected and the texture itself was used as a bump map (meaning the differences in brightness of different parts of the texture was used to make different parts of the surface rise or sink) to make it more realistic and less flat.

3.1.8 Bread Groups

Instead of selecting the breads completely at random, we tried to model the fact that breads in a shelf are often of the same type, or of a mix of similar types of breads (such as small breakfast breads). Thus, for each image we randomly selected from one of the following types of bread groups:

- Single type of bread
- Category of similar breads
- Completely random breads

[3] https://texture.ninja/

3.1.9 Shelf Walls

In the second version of the dataset, walls were randomly placed to add more variation to the environment and to encourage the model to ignore shelf walls in the real data. They were randomly assigned a material (from the same selection as for the floor planes), a thickness, height, length, position, and orientation. In order to make them a bit more realistic, a simple algorithm was used to make the walls more likely to go well together by making them more similar:

For the first wall placed:

- With 80% probability, use the same material as the ground plane, otherwise randomize the material
- Randomly select the thickness, height, length and rotation

For the remaining walls:

- With 70% probability, use the same material as one of the already placed walls, otherwise randomize the material. Use the same method to choose the value for the thickness, height and length.
- For the rotation:
  - With 44% probability, use the same rotation as a previous wall
  - With 44% probability, use a rotation perpendicular to a previous wall
  - With the remaining 12% probability, use a random rotation

Additionally, a transparent glass material was added to train the model to ignore the reflections sometimes seen in the real images. A glass wall was added with 1/3 probability.

3.1.10 Variants of the Synthetic Dataset

One dataset was created without walls (NOWALLS, Figure 38), one was created with walls (WALLS, Figure 39) and one highly simplified dataset was created by taking the first dataset and replacing each bread with its average color and replacing the background with its average color in each image (SIMPLE, Figure 40).

Figure 38: Sample images from Dataset NOWALLS.

Figure 39: Sample images from Dataset WALLS.

Figure 40: Sample images from Dataset SIMPLE.

3.2 Gathering Real Data

The bakery shelves in 17 supermarkets were photographed with cell phone cameras. The images were cropped into smaller parts, yielding 631 images in total. Samples of the dataset are shown in Figure 41.

Figure 41: Real dataset samples.

3.2.1 Object Detection

Instead of annotating the images from scratch, a semi-supervised approach was used, where annotations were first generated by the model trained only on synthetic data, and then used as a starting point as we carefully corrected mistakes in the masks, added missed breads, and removed false positives.

A subset of the supermarkets were selected for testing, giving a split of 427 training images and 204 test images. The test images were selected from store

images not included in the training images.

3.2.2 Similarity Classifier

To train the similarity classifier, we picked 360 images from the real dataset that had bread objects of more than a single type. These images were further divided into 330 for the training set and 30 for the test set.

3.3 Automatic Inventory System

The automatic inventory system consists of three main components: the object detection models to detect bread objects in an image, the similarity classifier to determine a similarity matrix of the pairwise similarity of all detected breads in the image, and the clustering algorithm to cluster the bread objects based on the similarity matrix. The input to the network is an image of a bread shelf and the output of the network is the clustered breads as shown in Figure 42.

Figure 42: Input of the automatic inventory system (top); Output of the automatic inventory system (bottom).

The architecture of the system is shown in Figure 43. The input is a variable sized image that is fed to an object detector. We tested various object detection models, however in the final version we use an instance segmentation model, Cascade Mask R-CNN. The object detector outputs bounding boxes and binary masks for each detected object in the input image.

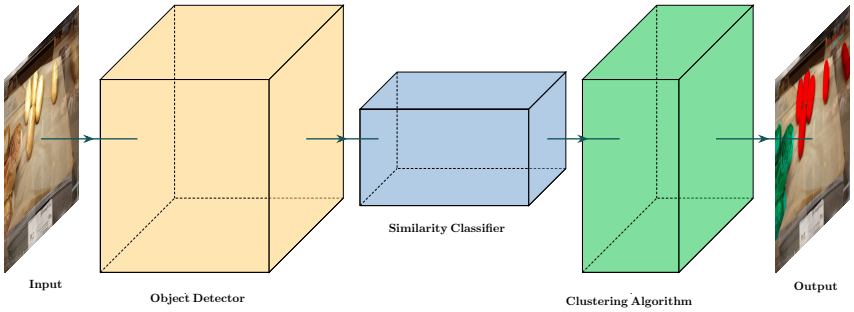

Figure 43: The architecture of the automatic bread inventory system.

The bounding box coordinates obtained from the object detector are used to crop all the detected objects from the input image. The surrounding area of an object aids in the similarity classification since in our case it is likely that similar breads are grouped together. We take advantage of this fact by adding 25% more area along the width of the bounding box. The height of the bounding box is set to twice the width, which is helpful in getting a square image when two cropped images are tiled together. Each object is paired with all other objects including itself to generate N^2 images, where N is the number of objects detected. An example of the final image is shown in Figure 44. These images are fed to the similarity classifier to get a similarity score for each image ranging from 0 to 1.

Figure 44: Example of input to the similarity classifier.

The scores from the similarity classifier are arranged in an NxN similarity ma-

trix, where element $x_{i,j}$ gives the similarity of object i to object j. This similarity matrix is used by the clustering algorithm to clustering similar objects together.

Training details of the object detector, similarity classifier and clustering algorithm are provided in the following sections. The training is a disjoint process for the object detector and the similarity classifier.

3.3.1 Object Detector

We tested different versions of R-CNN models, and YOLOv3 model for object detection.

- **R-CNN**: The github repository for TensorPack's implementation of Faster R-CNN [49] is used to test different version of R-CNN models. The test versions are listed below.
  - Faster R-CNN without Feature Pyramid Network (FPN).
  - Faster R-CNN with FPN.
  - Mask R-CNN, which is a Faster R-CNN with FPN and an additional mask branch.
  - Cascade Mask R-CNN, which is a Mask R-CNN with additional network heads for cascading.

  Each network used pretrained models trained on COCO Dataset [50] provided in the repository and fine-tuned on the synthetic WALLS dataset for 50 epochs with 500 steps per epoch. The first 2 epochs are the warm up epochs where the learning rate is 0.0001. The learning rate for the rest of the epochs is 0.001 with a decay of 0.0001. The feature extractor used is ResNet-101. The initial layers and the first block of ResNet-101 (c2) is frozen during fine-tuning, this corresponds to `BACKBONE.FREEZE_AT = 2` in the training configuration. The input image size is set to 640 pixels on the short edge and the long is scaled accordingly preserving the aspect ratio, however if the long edge length exceeds 1280 pixels, then it is set to 1280 and short edge is scaled accordingly preserving the aspect ratio. The default parameters are use for the rest of the model parameters.

- **YOLOv3**: We use David's YOLOv3 Github repository [51] to test the model on on the synthetic WALLS dataset. The feature extractor backbone is DarkNet-53, for which we use the provided weights pretrained on the ImageNet Dataset [52] for initialization. We use our dataset to perform transfer learning by freezing the backbone for the first 20 epochs, followed by 100 epochs of fine-tuning for which all the layer are unfrozen. This model requires approximately eight times more training time to reach comparable performance to R-CNN models, since the pretrained weights

are trained on an image classification dataset, instead of objection detection dataset. The Adam optimizer is used with a learning rate of 0.001. The input image size is 416x416 pixels. For the rest of the parameters the default values are used.

Some modifications are made to the above mentioned repositories to adapt them to our training environments and for ease of use.

3.3.2 Similarity Classifier

We test three different models for similarity classification, which are described in the following sections.

- **Object Similarity using Tiled Inputs (OSTI)**: The model that we use for the final version of our automatic bread inventory system is a novel EfficientNet based model OSTI that performs object similarity by using crops of an object pair tiled together as a single input. The motivation behind using a single input is to give access to the earlier convolutional layers of the classifier to both objects simultaneously, in contrast to Siamese models where the feature extractor has access to only one object at a time. We hypothesize that this earlier access may give the network more opportunities for comparing the two images at a lower level, which could improve the result compared to a normal siamese network. The Github repository for Qubvel's implementation of EfficientNet in Keras [53] is used for the EfficientNet base. We use Efficient-B0 without the classification head as the feature extractor, with a custom head that consists of two fully connected layers with batch normalization, dropout and swish activation layers. The output is obtained through a logistic Classifier. The network architecture is shown in Figure 45. The total number of model parameters for this model is 4,778,909.

  Training is performed using the binary cross-entropy loss with Adam optimizer using a learning rate of 0.0001. The model is trained on the similarity classifier dataset for 15 epochs. The input image size used in the final model is 380x380 pixels.

  For the training dataset, we take the 360 images from the similarity classifier dataset. For each bread object a rectangular crop is made with 25% extra width around the object and with the height equal to twice the width. If the object height is greater than twice the width then we instead take the height equal to the object height and make the width half of the height. Every object crop is then tiled with all the other object crops of the same image including itself to obtain a square image for every pair. Multiple datasets are created using different configurations. The main parameters involved in creating these datasets include image size `ims`, object area threshold `th`, preserving aspect ratio `par` and included pixels `ip`. The threshold `th` is used to filter objects that have area less than `th` times of

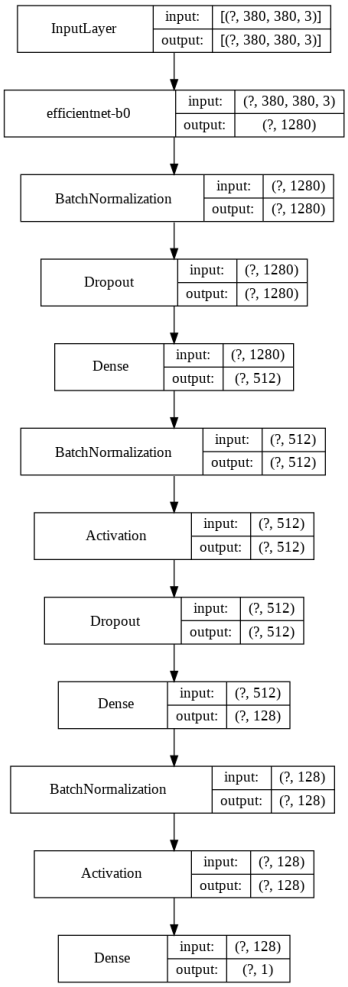

Figure 45: Architecture of the OSTI model.

the average area of objects in an image, while `par` is used to either pad the image with black bars if it doesn't fill the canvas or stretch it. We have a binary segmentation mask for each object, `ip` indicates whether we use the segmentation mask to include only the area covered by the object or also include the surrounding of the object. The datasets created are listed below.

– Dataset with `ip = mask`, including only the mask region. The following dataset all have `ip = surround`, which includes the surrounding pixels of the object.

– Datasets with `par = stretch` and `par = pad`.

– Datasets with `th = 0.0, 0.1, 0.2` and `0.3`.

– Datasets with `ims = 224, 240, 260, 300, 380, 400, 456, 528` and `600`.

We test our model with all these datasets to find the best configuration. Samples of some of the datasets are shown in Figure 46.

Figure 46: Samples from similarity classifier training datasets, (top left) Dataset with `ip = mask`, (top right) Dataset with `ip = surround` and `par = stretch`, (bottom left and right) Dataset with `ip = surround` and `par = pad`.

- **Siamese Model with Contrastive Loss**

  The second model we test is a Siamese network with contrastive loss. We try to keep the network as similar as possible to OSTI to perform an accurate comparison of the different approaches used. This model uses the same EfficientNet-B0 as the base model with the same top as mentioned above. The difference is the additional input, and the extra multiplication and subtraction layers that forms the custom L2-distance

layer. The architecture design is shown in Figure 47. Total number of parameters is 5,439,389 for this model.

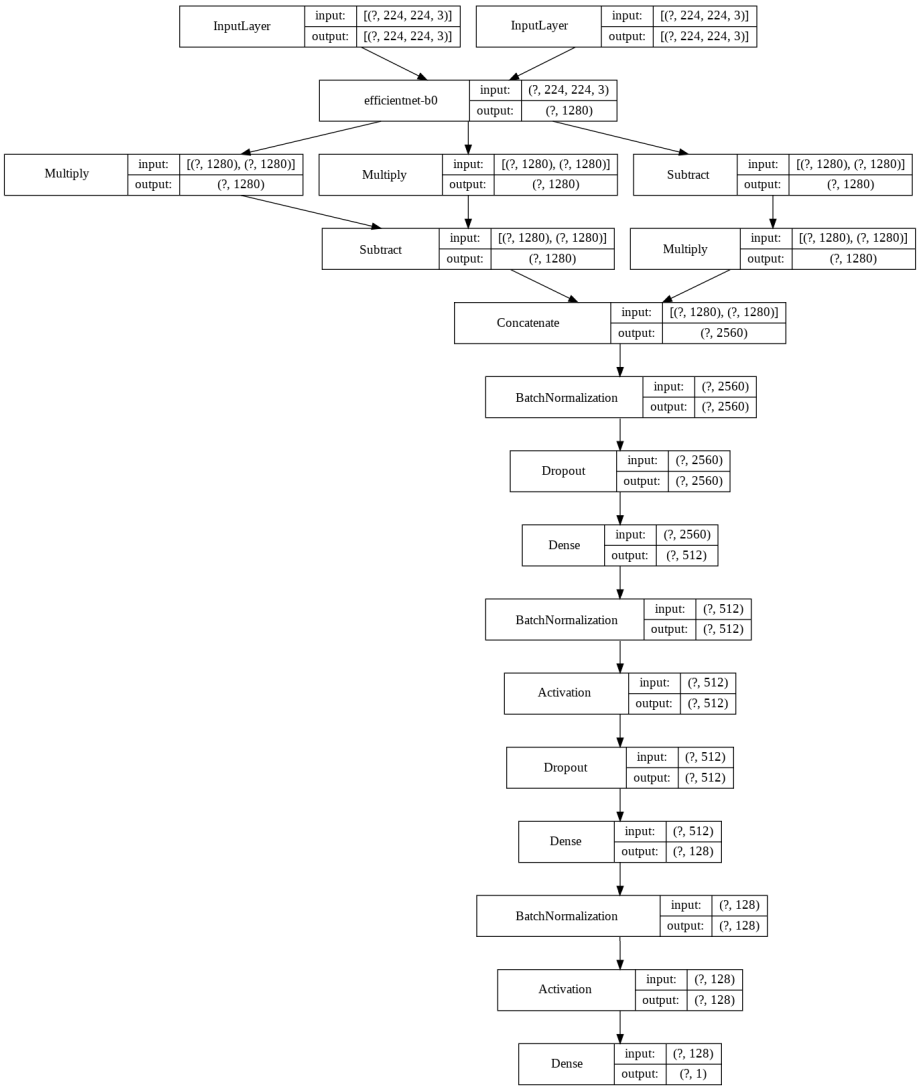

Figure 47: Architecture of the Siamese model with contrastive loss.

We train the model using the same binary cross-entropy loss and Adam optimizer with a learning rate of 0.0001. The model is trained on the similarity classifier dataset for 15 epochs. The input image size used in the final model is 224x224 pixels for each input.

For the training dataset, we again take the 360 images from the similarity

classifier real dataset. From these images, we crop the objects with 25% additional width and and height equal to width, and if the object height is greater than the assigned height than we set the height to object height and the width equal to height. This gives us a square image for each object crop. Here we don't need to tile the object crops to form pairs as the Siamese model takes two inputs.

- **Siamese Model with Triplet Loss**

  The third model tested is also a Siamese model but with the triplet loss. Again we try to keep the network as similar as possible to OSTI. To use the triplet loss, first we train the EfficientNet-B0 base separately with model architecture as shown in Figure 48.

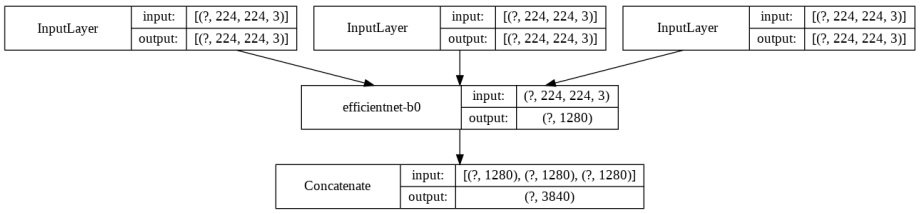

Figure 48: Architecture of the Siamese model with triplet loss.

For training, we make the same crops as for the Siamese model with contrastive loss, and then make all possible triplet combinations for each object. This results in a 10 times bigger dataset, and takes equally long to train for the same number of epochs.

The architecture of the final model capable of inference is the same as that of Siamese model with contrastive loss, but the EfficientNet-B0 base is replaced with the one trained with triplet loss. The base is then frozen while the top is trained. Since, the architecture of the inference model is the same as that of the one with contrastive loss, the number of total parameters is also same.

We use the same binary cross-entropy loss and Adam optimizer with a learning rate of 0.0001 for both the base and the final model. The models are trained for 15 epochs each. The input image size used in the final model is 224x224 pixels for each input as in the previous model.

3.3.3 Clustering Algorithms

We use Scikit-learn implementation of Affinity Propagation, DBSCAN and Mean Shift clustering algorithms to cluster the objects using similarity matrix as input.

3.4 Evaluation

3.4.1 Object Detector

The object detector is evaluated through the mAP50 score. We chose mAP50 instead of mAP COCO since we are more interested in recall of the detected objects to count every object possible, rather than the precise accuracy of each detection. All the models are tested on 204 validation images of the real dataset.

3.4.2 Similarity Classifier

Here the evaluation is performed on the basis of prediction accuracy. The 30 validation images of the similarity classifier dataset are used for to get evaluation results. For training, we use datasets with different object area thresholds, however in real scenarios, since we cannot exclude object with low object area, we set the object area thresholds to 0 for all evaluations.

3.4.3 Clustering Algorithm

The clustering algorithm is evaluated based on the V-measure and Rand Index scores. The model is evaluated on the 30 validation images of the similarity classifier dataset, and we use ground truth classes to compute the evaluation scores.

4 Results

4.1 Object Detection

To evaluate the object detection models and the added benefit of the synthetic dataset we perform various experiments with different object detection models and different datasets.

4.1.1 Results for Full Synthetic Datasets

The results for various object detection models on the full synthetic dataset WALLS are shown in Table 1. The model with best accuracy among the tested models is Cascade Mask R-CNN which is used for the rest of the tests and the final automatic bread inventory system.

Training data	mAP50
Faster R-CNN w/o FPN	0.851
Faster R-CNN	0.872
Cascade Faster R-CNN	0.882
Cascade Mask R-CNN	0.885
YOLOv3	0.863

Table 1: Performance result of various object detection models.

4.1.2 Results for Full Synthetic Datasets

Table 2 shows the results for Cascade Mask R-CNN trained on the three variations of the full synthetic dataset. As expected, we get the best result for the model with WALLS dataset, which resembles the test scenario the most. At the same time, the SIMPLE dataset gives a surprisingly high performance, showing a strong generalization ability of the network even when realistic details are not present.

Training data	mAP50
NOWALLS	0.842
WALLS	0.885
SIMPLE	0.736

Table 2: Full synthetic dataset result

4.1.3 Performance vs. Amount of Data

We can see in Figure 49 and 50 that both datasets follow a similar curve, where the performance gains of adding more data are greatest in the beginning, and that they converge toward different final values. While the synthetic data reaches a lower performance than the real data, it is still relatively high, showing that the model trained on synthetic data generalizes to real data to some extent.

The gap between the performance of the real and synthetic dataset can also be attributed to the fact that our test dataset include images with objects that are not only breads but also other bakery products such as pastries. These images were included in the real dataset to test the ability of the model to generalize to other kinds of bakery products, which can be realistically assumed to taken care of by a capable automatic bread inventory system.

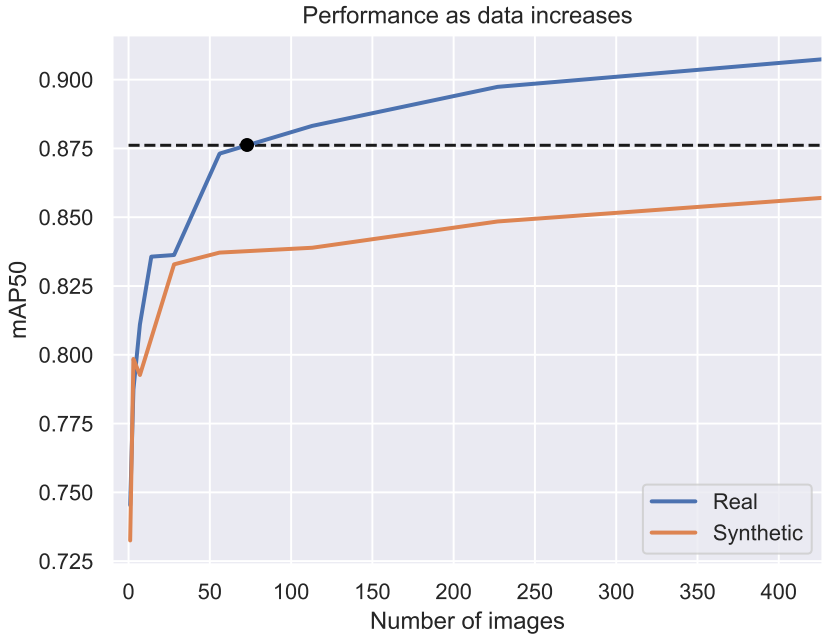

Figure 49: Performance of the model when trained on varying amounts of data. The dashed line shows the maximum performance of the synthetic dataset (0.876), when trained on all 14 570 images. The intersection with the blue curve shows that the same performance is reached by training on 73 real images.

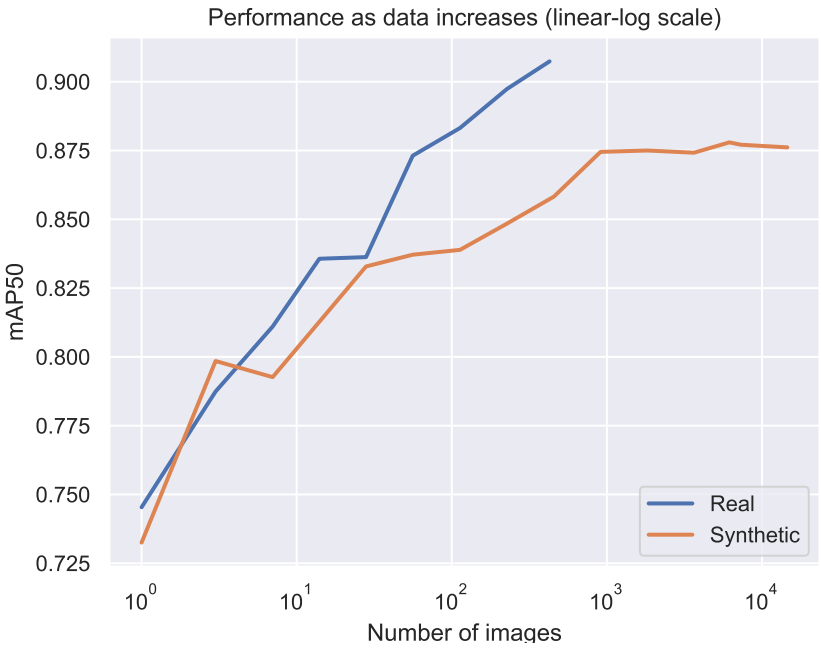

Figure 50: Linear-log plot of the same data, including the remaining part of the synthetic curve. We can see that the synthetic performance increase comes to a halt at around 1000 images, after which even an order of magnitude more data doesn't seem to make much of a difference. The curve for real data on the other hand does not yet seem to have converged.

4.1.4 Performance vs. Fraction of Real Data

Figure 51 shows a plot of the performance of the model when trained on the same number of images but using different mixes of synthetic and real data. Note how there is a relatively large performance gain when going from 0 to 10 percent of real data, after which the performance continues to improve but a little slower. This can be explained by the fact that model is able to learn to detect other types of bakery products mentioned in the previous section, with only a few examples, and therefore, improves dramatically, with only a few added training examples.

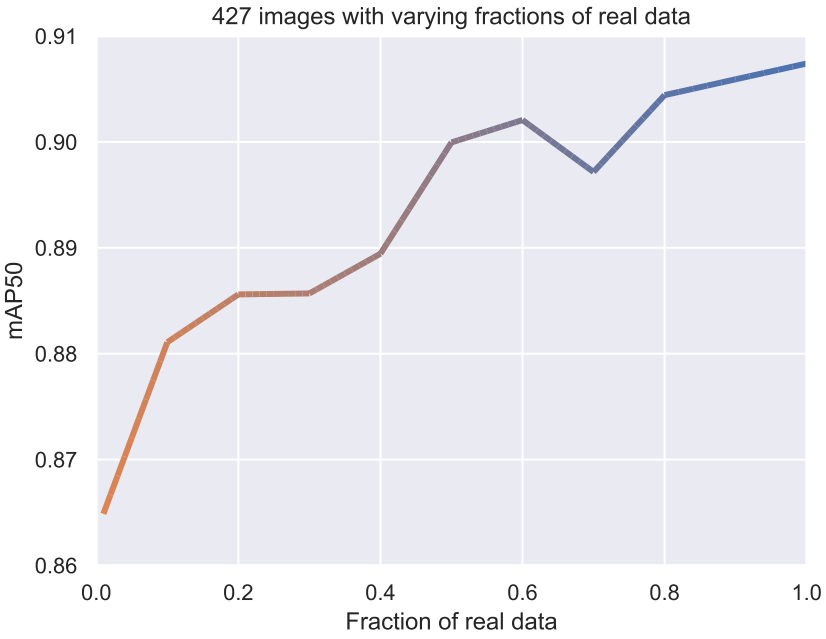

Figure 51: Results for varying fractions of synthetic vs real data.

4.1.5 Performance vs. Amount of Light Variation

Increasing the number of different light scenarios surprisingly gives a slightly worse performance than when using fewer scenarios as can be seen in Figure 52. It is possible that some light conditions have a negative effect on the training by creating images that are too difficult, for example too dark or too unnatural.

Figure 52: Results for increasing number of different light environments

4.1.6 Performance vs. Number of Different Bread Models

Increasing the number of breads generally improves the performance, but the extra addition of the style-transferred breads gives no further improvement, as can be seen in Figure 53. This indicates that making the dataset too complex proves to be detrimental for learning better object detection instead of providing the expected better generalization.

Figure 53: Results for increasing number of different bread models

4.1.7 Effects of Adding Real Negative Examples to Synthetic Data

The Cascade Mask-RCNN models trained entirely on the synthetic dataset tend to detect objects that are not bread as bread, for example the edge of the bread basket or the bread shelf can be identified as bread because it has a similar shape to a baguette. This is an indication that the synthetic dataset does not include enough negative examples of the kinds of background objects that one can find in a bread shelf. To mitigate this shortcoming, we added some negative samples to the synthetic dataset, which came from real images of places inside the supermarkets containing no breads as in Figure 54, and crops of real images of bread shelves that containing no bread as shown in Figure 55.

Figure 54: Images of places inside the supermarkets containing no breads added to the synthetic dataset as negative examples.

Figure 55: Images of bread shelves inside the supermarkets containing no breads added to the synthetic dataset as negative examples.

After addition of these negative examples to the dataset, the model learns to not detect every bread shape as an object, but that also leads to a decreased recall. This is due to the fact that the synthetic dataset has limited kinds of bread models, which don't generalize too well. The total number of negative example images is 250, and to see the effect of adding more negative images, we add copies of these images to increase the total number of negative example images, giving us datasets with 250, 500, 1000 and 2000 negative example images, and 14570 synthetic images. The validation accuracy for these datasets is shown in Table 3.

Number of Negative Examples	mAP50 Score
0	0.880
250	0.880
500	0.876
1000	0.873
2000	0.860

Table 3: Increasing negative examples

4.1.8 Training Schemes

We train the Cascade Mask R-CNN to get the best possible accuracy using both the real and synthetic dataset. For this purpose try different training schemes, which are listed below.

- Fine-tuning with only synthetic dataset.
- Fine-tuning with only real dataset.
- Fine-tuning with synthetic dataset and followed by fine-tuning with real dataset.
- Fine-tuning with combined dataset of both real and synthetic dataset.

The model accuracy for the above mentioned schemes are shown in Table 4. The result shows that training on only the synthetic data is quite close to training on only the real data in terms of accuracy. Combining real and synthetic datasets into a single dataset is better than just synthetic data but worse that only real data, indicating that mixing these dataset makes it harder for the model to learn useful features. Fine-tuning on only synthetic followed by fine-tuning on only real dataset proves to be the best approach yielding the best result.

Training Scheme	mAP50 Score
Synthetic Only	0.885
Real Only	0.907
Real + Synthetic	0.905
Synthentic followed by Real	0.919

Table 4: Combinations of real and synthetic data

4.2 Object Clustering

4.2.1 Siamese Network with Contrastive Loss

The clustering metrics result obtained for the clustering validation set are shown in Table 5. Different configurations of the model are tested, however we only present the results obtained for the best configuration, and present more results for the best approach OSTI in the corresponding section ahead.

We obtain decent results with this approach, but there is room for improvement. Our dataset include many difficult examples where very similar looking breads can belong to different class, this can be one reason that stops the model from achieving better results.

Algorithm	V-measure	Adjusted Rand Index
DBSCAN	0.666	0.442
Mean Shift	0.717	0.534
Affinity Propagation	0.669	0.557

Table 5: Clustering evaluation for Siamese network with the contrastive loss

4.2.2 Siamese Network with Triplet Loss

Siamese network with triplet loss gives us the clustering evaluation result shown in Table 6. Using the triplet loss gives worse results for our dataset compared to its contrastive loss counterpart. Again only the results for best configuration are presented.

A possible reason for this result is that triplet loss performs better with datasets containing many classes, however in our case, we have only few different kinds of breads per image, and since the objects are compared with other objects within the same image, the number of classes is low. Another reason is that for the triplet loss to perform good, it is better have the anchor and the negative of a sample carefully picked such that they are similar but still from different classes, however in our case, it was not feasible to do so due to size of the dataset.

Algorithm	V-measure	Adjusted Rand Index
DBSCAN	0.564	0.395
Mean Shift	0.658	0.414
Affinity Propagation	0.631	0.491

Table 6: Clustering evaluation for Siamese network with the triplet loss

4.2.3 Object Similarity using Tiled Inputs (OSTI)

We used our EfficientNet-B0 based similarity classifier on various datasets to find the best similarity classifier. The performance difference between dataset

`ip = mask` (included pixels), including only the mask region and dataset `ip = surround`, which also includes the surrounding pixels of the object is shown in Table 7.

Dataset	Validation Accuracy
mask only	0.5974
surrounding	0.7961

Table 7: Different included object pixels

As expected, including the surrounding pixels of the objects in the training data significantly improves the performance of the classifier. In Table 8, we see the validation accuracy for datasets with `par = stretch` (preserve aspect ratio) and `par = pad` with `ip = surround` for both.

Dataset	Validation Accuracy
stretch	0.6973
pad	0.7961

Table 8: Different canvas filling approach

Here we see that padding the image gives a higher accuracy, which can be expected since padding preserves the aspect ratio and avoids distortion of the training data, making the side-by-side breads more comparable to each other. Next we show the comparison of datasets with `th = 0.0, 0.1, 0.2` and `0.3` (object area threshold) in Table 9.

Threshold	Validation Accuracy
0.0	0.9142
0.1	0.9180
0.2	0.9153
0.3	0.9154

Table 9: Increasing object area threshold

In Table 9, we see that the object area threshold value of 0.1 gives the best result. Table 10 shows the comparison of datasets using different input image sizes.

Image Size	Validation Accuracy
224x224	0.9269
240x240	0.9259
260x260	0.9312
300x300	0.9187
380x380	0.9321
400x400	0.9190
456x456	0.9257
528x528	0.9251
600x600	0.9142

Table 10: Increasing input image size

From Table 10 we can see that there are some slight changes for different sized inputs. The best accuracy is given by input size of 380x380, which what we use for the final system. Further experiments include increasing the dataset size, which is shown in the table below.

Dataset Size	Validation Accuracy
22656	0.7961
53032	0.8172
106242	0.9142

Table 11: Increasing dataset size

We can see that the performance of the models significantly increases with increase in the dataset size. We also tried increasing the complexity of the model by using a EfficientNet-B2 as the base model. The results are shown in Table 12.

Base Model	Validation Accuracy
EfficientNet-B0	0.8172
EfficientNet-B2	0.7634

Table 12: Increasing model complexity

Increasing the model complexity decreased the performance, as the model starts to overfit the training data. We see high training accuracy for the classifiers with EfficientNet-B0 as the base, which suggest that B0 is capable of learning the required features. The last experiment deals with freezing majority of layer of EfficientNet base. The results of this experiment are shown in Table 13.

Freezing part of the EfficientNet base leads to lower accuracy, therefore, we trained our best models with all layer unfrozen.

Frozen level	Validation Accuracy
Frozen till block B6	0.7349
All layers unfrozen	0.7961

Table 13: Freezing model layers

Results of clustering metrics when used with our proposed OSTI approach yeilds results shown in Table 14.

Algorithm	V-measure	Adjusted Rand Index
DBSCAN	0.798	0.627
Mean Shift	0.753	0.567
Affinity Propagation	0.732	0.652

Table 14: Clustering evaluation for OSTI

As can be observed the results are significantly better than those of Siamese networks with the contrastive and triplet losses.

4.2.4 Failure Cases

From the results, we see that DBSCAN tends to give the highest number of clusters, and Mean Shift tends to give a lower number of clusters than DBSCAN, while Affinity Propagation gives the lowest number of clusters. This behavior can lead to DBSCAN performing better in images with more different kinds of bread, while Affinity Propagation performing better in images with less different kinds of bread. Mean Shift gives a balance between these two extremes. For our test set, Affinity Propagation gives the best average performance. This behavior is shown in Figures 56 to 61.

Figure 56: Results for clustering algorithms example 1, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result.

Figure 57: Results for clustering algorithms example 2, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result.

Figure 58: Results for clustering algorithms example 3, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result.

Figure 59: Results for clustering algorithms example 4, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result..

Figure 60: Results for clustering algorithms example 5, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result.

Figure 61: Results for clustering algorithms example 6, (top left) original image, (top right) DBSCAN result, (bottom left) Mean Shift result, (bottom right) Affinity Propagation result.

5 Discussion

The discussion is divided into two main sections, relating to the design of the automatic bread inventory system and generation of synthetic training data for training the inventory system.

5.1 Automatic Bread Inventory System

5.1.1 Object Detection

The choice of the object detection model for our system was mainly guided by the accuracy of the model, without taking into consideration the inference time for the tested model. That is the reason we chose Cascade Mask R-CNN over YOLOv3, which actually turns out to be much more efficient and quite close to our model in performance. Other reasons for choosing Cascade Mask R-CNN includes faster transfer learning time and being able to predict a mask for each object.

The object detection training scheme that gave the best accuracy was to first fine-tune the Cascade Mask R-CNN on only the synthetic dataset and then fine-tune it further using only the real dataset. This shows that the real and synthetic datasets might be able to complement each other, but the effect in our case is very small.

5.1.2 Object Similarity Classification

The design for Object Similarity with Tiled inputs (OSTI) model, was a product of an effort to come up with the simplest model to perform object similarity between two objects. It was also motivated by the fact that the more complicated models such as the Siamese networks with the contrastive and triplet loss, didn't provide satisfactory results.

We did not train OSTI or the other object similarity classifiers using the synthetic data. While synthetic data could be used to increase the quantity of the data, it would be interesting to see in a future work whether a similarity model based on synthetic data could generalize to real data.

5.1.3 Clustering

The clustering algorithm used in the system does not require the number of clusters to be known, making it difficult to cluster objects, since, in the same image we can have different objects that are visually quite similar and also different objects that are significantly different.

5.2 Generation of Synthetic Training Data

5.2.1 Performance for Only the Synthetic Data

The performance when using only the synthetic dataset is relatively impressive (0.88 mAP50), and not that far from the result when trained on real data. However, it's important to remember that we are fine-tuning a network that is already pretrained on large amounts of real images. Training on the synthetic data from scratch would probably lead to much more overfitting.

5.2.2 Stripping Down the Synthetic Data

When we set out to design the synthetic dataset, our strategy was to make it as varied as possible in order to avoid overfitting and improve the generalization abilities to the real dataset. While this view is supported by some of our results (having more images leads to better performance), other results surprisingly speak against it. For instance, increasing the number of bread types gave a slight improvement at first when adding more of the original models, but when adding the style transfer-generated models the performance stayed unchanged (Figure 53). And when increasing the number of light types, the performance even went down slightly, with the top performance reached at the lowest number of light types 3 (Figure 52).

One conclusion in our case is thus that a large amount of variation in these aspects were not as important as expected, and that having less variation in them still gives good generalization. One explanation for this could be that the network we are fine-tuning is highly invariant to the lighting of the scene. Through its extensive pretraining on classifying images, it has learned to recognize objects in any light situation and to care more about the geometry and 3d form of the object than its lighting, which makes having a large light variation superfluous. The large variation would probably prove more useful if the network was trained from scratch.

Finally, the relatively high result on the SIMPLE dataset shows that using simple blob-shapes instead of path-traced renders can give high performance. The realistic appearance of the scene accounts only for approximately the final 0.15 points of the achieved mAP50 score. The performance reached with the SIMPLE dataset is also similar to the performance reached when training on only a few real images, which both demonstrate the high generalization ability of the network used.

5.2.3 Effects of Using Real Data

For both synthetic and real datasets, more data is better until a point of diminishing returns is reached. However, the real dataset ends up at a higher accuracy than the synthetic dataset. There is a gap between the accuracy curves, which can be explained by the fact that some aspects of the real data are not fully

captured by the synthetic data, e.g. the existence of price tags and towels, signs with mini breads, pastries etc.

We can see a benefit of adding even a relatively small amount of real data to the synthetic data to adapt the model to the different real-world domain.

Synthetic data could be an alternative if somewhat lower performance is acceptable. Depending on the amount of work required to set up the synthetic data generation system and obtaining the 3d models, it may be more worthwhile to just collect the real data.

6 Conclusion

We have designed and implemented a computer vision system for automatic bread inventory in supermarkets, achieving satisfactory results, and we have proposed and used a simple novel approach to classify object similarity. The shortcomings of our system have also been detailed which can be further improved upon.

We have shown that synthetic data generation using path-tracing and photo-scanned models can be a viable method for fine-tuning a modern object detection network at the cost of some reduction in accuracy compared to real data. Whether the method is suitable depends on the the required accuracy and the difficulty of collecting real data vs. obtaining photo-scanned models.

References

[1] A. Handa, V. Patraucean, V. Badrinarayanan, S. Stent, and R. Cipolla, "Scenenet: Understanding real world indoor scenes with synthetic data," *CoRR*, vol. abs/1511.07041, 2015. [Online]. Available: http://arxiv.org/abs/1511.07041

[2] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *European Conference on Computer Vision (ECCV)*, ser. LNCS, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., vol. 9906. Springer International Publishing, 2016, pp. 102–118.

[3] Z. Cai and N. Vasconcelos, "Cascade r-cnn: Delving into high quality object detection," 2017.

[4] A. Tonioni, E. Serra, and L. Di Stefano, "A deep learning pipeline for product recognition on store shelves," 12 2018, pp. 25–31.

[5] L. Karlinsky, J. Shtok, Y. Tzur, and A. Tzadok, "Fine-grained recognition of thousands of object categories with single-example training," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[6] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, and S. Birchfield, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[7] J. Fišer, O. Jamriška, M. Lukáč, E. Shechtman, P. Asente, J. Lu, and D. Sýkora, "StyLit: Illumination-guided example-based stylization of 3d renderings," *ACM Transactions on Graphics*, vol. 35, no. 4, 2016.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[9] H. H. Sultan, N. Salem, and W. Al-Atabany, "Multi-classification of brain tumor images using deep neural network," *IEEE Access*, vol. PP, pp. 1–1, 05 2019.

[10] G. Bonaccorso, *Machine learning algorithms: popular algorithms for data science and machine learning*. Packt, 2018.

[11] S. Kini, "Getting started with cnn," Apr 2020. [Online]. Available: https://medium.com/@kinisanketh/getting-started-with-cnn-18c03efc7d06

[12] S. Balachandran, "Machine learning - max average pooling," Mar 2020. [Online]. Available: https://dev.to/sandeepbalachandran/machine-learning-max-average-pooling-1366

[13] "Light on math machine learning: Intuitive guide to convolution neural networks," May 2018. [Online]. Available: http://www.thushv.com/computer_vision/light-on-math-machine-learning-intuitive-guide-to-convolution-neural-networks/

[14] K. He and J. Sun, "Convolutional neural networks at constrained time cost," 2014.

[15] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," 2015.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2013.

[19] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[20] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, "Selective search for object recognition," 2013. [Online]. Available: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf

[21] R. Girshick, "Fast r-cnn," 2015.

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2015.

[23] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," 2017.

[24] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," 2016.

[25] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[26] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016.

[27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2

[28] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[29] Q.-C. Mao, H.-M. Sun, Y.-B. Liu, and R.-S. Jia, "Mini-yolov3: Real-time object detector for embedded applications," *IEEE Access*, vol. PP, pp. 1–1, 09 2019.

[30] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[31] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2019.

[32] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," 2018.

[33] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Inverted residuals and linear bottlenecks: Mobile networks forclassification, detection and segmentation," 01 2018.

[34] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-excitation networks," 2017.

[35] B. Willims, "Contrastive loss explained," March 2020. [Online]. Available: https://towardsdatascience.com/contrastive-loss-explaned-159f2d4a87ec

[36] Y. Jing, Y. Yang, Z. Feng, J. Ye, and M. Song, "Neural style transfer: A review," *CoRR*, vol. abs/1705.04058, 2017. [Online]. Available: http://arxiv.org/abs/1705.04058

[37] "Clustering." [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html

[38] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007. [Online]. Available: https://science.sciencemag.org/content/315/5814/972

[39] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, E. Simoudis, J. Han, and U. M. Fayyad, Eds. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[40] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: Why and how you should (still) use dbscan," *ACM Trans. Database Syst.*, vol. 42, no. 3, Jul. 2017. [Online]. Available: https://doi.org/10.1145/3068335

[41] Yizong Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[42] M. Rosenblatt, "Remarks on some nonparametric estimates of a density function," *Ann. Math. Statist.*, vol. 27, no. 3, pp. 832–837, 09 1956. [Online]. Available: https://doi.org/10.1214/aoms/1177728190

[43] E. Parzen, "On estimation of a probability density function and mode," *Ann. Math. Statist.*, vol. 33, no. 3, pp. 1065–1076, 09 1962. [Online]. Available: https://doi.org/10.1214/aoms/1177704472

[44] M. Nedrich, "Mean shift clustering overview," May 2015. [Online]. Available: https://spin.atomicobject.com/2015/05/26/mean-shift-clustering/

[45] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure." 01 2007, pp. 410–420.

[46] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical Association*, vol. 66, no. 336, pp. 846–850, 1971. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482356

[47] L. Hubert and P. Arabie, "Comparing partitions," *Journal of classification*, vol. 2, pp. 193–218, 1985. [Online]. Available: https://doi.org/10.1007/BF01908075

[48] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, Jul. 1985. [Online]. Available: https://doi.org/10.1145/325165.325247

[49] Y. Wu, "Tensorpack faster r-cnn." [Online]. Available: https://github.com/tensorpack/tensorpack/tree/master/examples/FasterRCNN

[50] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015.

[51] david8862, "david8862/keras-yolov3-model-set." [Online]. Available: https://github.com/david8862/keras-YOLOv3-model-set.git

[52] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[53] Qubvel, "Efficientnet." [Online]. Available: https://github.com/qubvel/efficientnet.git