A Study on Automatic License Plate Recognition

Joel Andersson

Master of Science (Technology) Thesis

Supervisor: Dragos Truscan

Åbo Akademi University

Faculty of Science and Engineering

2022

Abstract

Automatic license plate recognition, ALPR, is the process of detecting license plates, and recognizing the characters, present in any given image. ALPR can be applied in many situations to reduce the requirement for manned operation of various tasks. In this thesis, different methods for performing ALPR are tested and qualitatively evaluated. The main focus is on deep learning based object detectors, how they can be applied for a more modern version of ALPR and the challenges involved. A significant problem with deep learning is the amount of data needed, and this was also encountered by the author of this thesis. The data sets gathered did not have enough variation for training deep learning models with satisfactory accuracy, which was solved by generating images of license plates to increase the variation in the data sets.

Contents

Glossary

ALPR Automatic License Plate Recognition. 7–10, 40, 50, 54, 68, 72, 83, 84, 98, 110, 119, 123

ANN Artificial neural networks, often referred to as neural networks (NN), are machine learning models that attempt to mimic the function of biological neural networks. 11

GIoU Generalized Intersection over Union. 55

IoU Intersection over Union. 54, 57

ML Machine learning. Methods that can be performed by a computer to detect patterns in data, predict future data or make some type of decisions without explicit instructions. [11]. 74, 87

OpenCV Open source Computer Vision library. 42, 43, 50, 81, 86, 91

SSD Single Stage Detector, a type of machine learning object detection method. 45, 56

1 Introduction

Automatic License Plate Recognition (ALPR) is the process of taking an image and determining whether there are license plates in the images and then identifying the characters in those license plates, all without human intervention.

ALPR has many practical applications, such as traffic flow statistics collection [1]; parking management systems [2], travel time estimation [3], automated toll violation monitoring [4] and even aiding police by providing real-time data paired with location [5].

ALPR is a field which has been growing in popularity and especially so in recent years, as computing power is becoming more affordable and more readily available. Originally ALPR was performed with highly proprietary hardware but, in recent years, it has become possible to perform ALPR on both higher resolution images and using general purpose hardware. All in all, the progress of technology, open source software and machine learning has made it easier for anyone with an interest to implement their own ALPR systems.

There are several approaches to ALPR but, on the whole, the most important steps can be viewed as license plate detection followed by character recognition. The more conventional approaches to ALPR work best in controlled environments where the vehicle is always at the same angle and position in the image, and the license plate is brightly illuminated. These constraints come from the conventional approaches making assumptions about what to expect in the image. However, since the rise of machine learning, it has become more realistic to circumvent these tight constraints and instead make algorithms that work in more varied environments.

There are complete ALPR systems for both moving and stationary cameras, developed and sold by companies such as [6] that come with all the hardware etc., all built into one product. These kinds of systems have been successfully deployed by Police Departments in the US [5] but they are not cheap, since one pays for everything and the prices are often not even listed. At the more affordable end of the spectrum, there are online services to which one can forward images and receive the license plate recognitions in response. Such services are available from e.g. [7, 8]. These are usually more transparent regarding the pricing, but deploying thousands of systems using these kinds of services and running them every hour of the day will quickly become expensive as well. Not to mention the cost of the network bandwidth consumed. Having a reliable internet connection is also not necessarily a given and ALPR systems will often need to produce results immediately in order to be effective. Sending images of license plates to a third party also requires a high level of trust.

In this thesis, I go into the core problems surrounding ALPR, discuss and compare the performance of different possible solutions to each of these problems, as well as highlight the strengths and weaknesses of each solution. The goal is to identify good paths to approaching the problem of ALPR, depending on one's requirements. In this thesis, the context will be Finnish license plates, but the problems will be addressed on a general level.

1.1 Goal

There is a virtually limitless number of tasks that can be sped up remarkably by introducing an ALPR system. Most of these systems utilize static cameras, but some, such as the systems deployed on police vehicles [5], need to work even on moving vehicles. A stationary camera can use certain algorithms that are not feasible for a moving camera but, in the end, they all need to do the same thing: read pictures, extract license plate images and identify the characters in said license plate images. Thus, they all face the same problems to some extent. The main problems involved in each activity are that the license plate can be partially obscured, lighting conditions can vary, the resolution of the image can be too low if the vehicle is far away, and the plane of license plate may not be parallel to the plane of the camera, making it difficult to read the license plate characters correctly. The common factors between these problems can be boiled down to variation in the environment, variation in the license plates, and variation in technology used.

In the early days of ALPR, lights were often installed to try and ensure more consistency in the data [9, 10], but an algorithm that is capable of performing

ALPR regardless of lighting conditions would be preferable. A flexible system will handle different lighting conditions and perhaps difficult perspectives, but not partially obscured plates. In order to address that we also need to compare each plate with previous (recent) detections of the same license plate. For example, a strong light might reflect off of the plate in a way that obscures the characters in the area of the reflection, and that spot might move across the plate between images, thus always obscuring some part of the plate. In this case, a more advanced ALPR system should be able to piece the detections together and obtain a complete license plate. When more pictures of the same license plate are needed to form a complete recognition, the program needs to be fast enough to take more pictures while the license plate is still in view of the camera.

Some ALPR systems rely on an external trigger of some sort, which makes the camera take one picture and only then runs the ALPR system on that image. The nature of these kinds of systems is that the one image taken needs to be perfect, so a very high quality camera is needed (preferably with the focus automatically being adjusted by the estimated position of the vehicle). However, these systems do not usually require instantaneous results to be effective. An example of such a system is a speeding camera. The camera is triggered by a radar or some form of sensor underneath the drive path, a high resolution picture is taken using a camera with a very fast shutter, the license plate is identified, and the offender receives the bill in the mail a few days or weeks later. This kind of system would not necessarily even need to be automated; a police officer could take a look at the image and look up the owner of the license plate they see in the picture. Automating it would save man-hours spent, but the cost of false positives could potentially outweigh the benefits. Other applications, such as restaurant drive-throughs that employ loyalty programs or automatic gates for parking caves need to respond quickly to deliver the best possible service to the user. The faster an ALPR system can produce the recognitions, the more meta-analysis can be performed, such as tracking the position of a vehicle in the image and determining that all the license plate recognitions from within the bounds of that specific vehicle can be combined to create a recognition with a higher confidence. This meta-analysis touches on what was mentioned in the previous paragraph, namely being able to piece together incomplete recognitions. If the images are processed quickly enough, assumptions can be made regarding how far the vehicle could possibly have moved (from the perspective of a single camera). However, always collecting images (without a trigger) also makes it possible to track how fast the vehicle is moving on average, without expensive sensors such as radars. Continuous image collection permits comparing license plate recognitions between cameras in different locations. Live feed ALPR can also be used by city planners to determine how efficiently traffic is flowing and where traffic congestions emerge [1].

The main goal of this thesis is to investigate different approaches to implementing ALPR and to provide the reader with a better idea of how each problem can be tackled and what one can do to improve performance and reduce development time. The advantages and disadvantages to each approach will be discussed, and some of these approaches will be implemented and their performances compared.

2 Background - Machine Learning and Deep Learning

In this section, I review the necessary background theory surrounding the concepts and terms later used with regard to deep learning-based object detectors. The theory gradually becomes more complex and specific, until all of the concepts needed to discuss YOLOv5, which is the object detector that ultimately was tested, have been introduced.

2.1 Machine Learning

Machine learning constitutes any method that can be performed by a computer on a set of data in order to detect some type of pattern, predict future data or make some type of decision based on the data. The parameters of those methods are not explicitly written but, instead, determined through algorithms learning what the data means [11].

2.1.1 Neural Networks

An artificial neural network (ANN), sometimes multi-layer perceptron (MLP) or neural network (NN), is an interpretation of the biological neuron that can be expressed using mathematics and implemented in code. Neural networks were first introduced as a concept in 1943 by McCulloch & Pitts [12]. Later, in 1958, the perceptron was introduced by Rosenblatt as a model for the neuron [13]. Rosenblatt's theory of perceptrons is still applied in neural networks today [14]. Neural networks can be used to solve two types of problems: classification and regression. See Figure 1.

The perceptron/artificial neuron itself consists of weighted input connections, an activation function and a bias. The output of the neuron is calculated as shown in the following formula below, where j is the number of the neuron, f_j is the neuron's activation function, n is the number of connections, i is the connection number, w_i is the weight of the connection, x_i is the input value on that connection and b is the bias. See Figure 2.

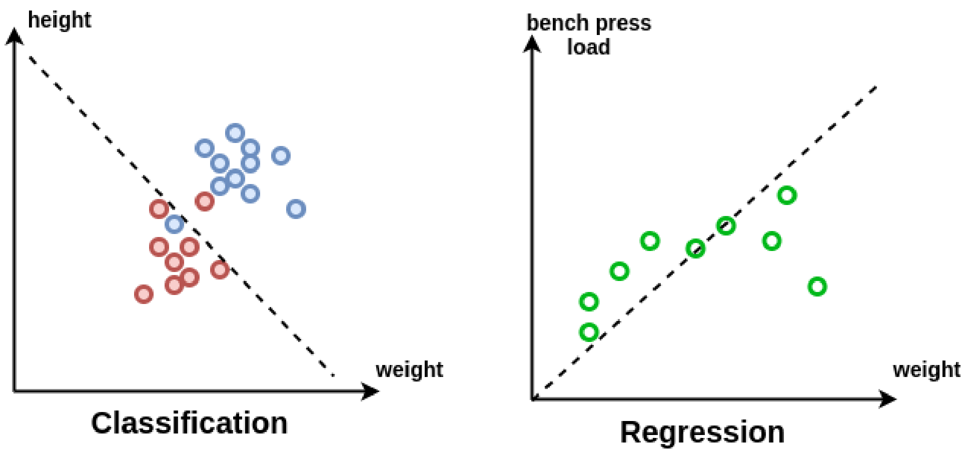


Figure 1: The two types of problems neural networks can solve: classification and regression. The classification example shows what a classification of male/female based on height and weight might look like. The regression example shows what the relationship between the maximum load in the bench press exercise and body weight might look like. Both examples are approximated using linear functions ($f(x) = ax + b$).

$$y_j = f_j(\sum_{i=0}^{n} w_i \cdot x_i) + b$$

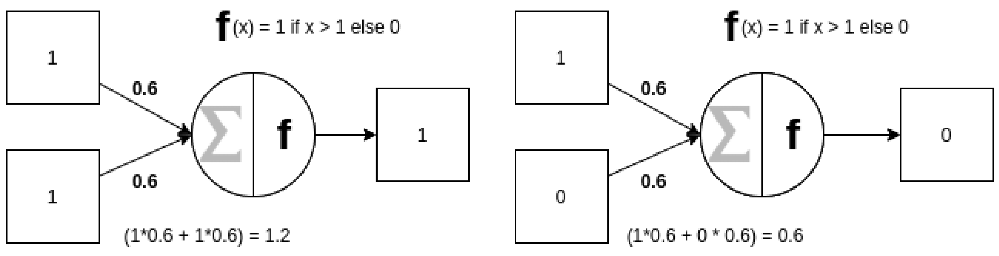


Figure 2: An example of what a perceptron might look like. Both weights have the value 0.6 and the activation function is a threshold set to 1. This perceptron calculates binary AND. Note that the weights and threshold value could change to a large degree, and it would still calculate AND correctly. The bias is left out of this example, but it would be added as described earlier.

There are many popular activation functions, some of which output a floating point value and some of which output a binary value; a few examples of popular activation functions [15] are shown in Figure 3. Because the activation function directly shapes the output, it ultimately determines what patterns the neural network will be able to recognize.

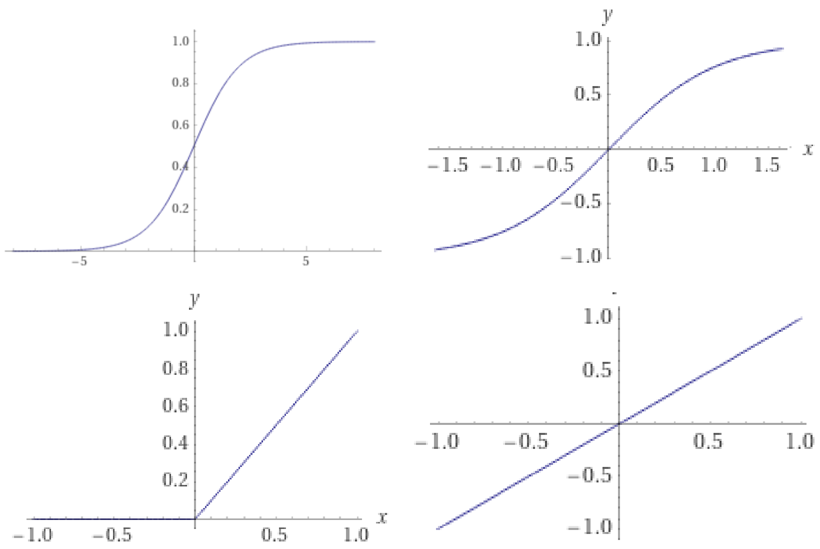


Figure 3: The sigmoid, tanh, rectified linear unit and linear functions. All commonly used as activation functions in neural networks.

Neural networks have three sections: the input layer, the hidden layers and the output layer. Each layer consists of a set of neurons which are connected to the next layer with weights. The final layer, the output layer, does the same thing but while some middle layers' activation functions might only produce either a 0 or a 1, the neurons in the final layer usually have an activation function that produces a floating point value between 0 and 1. This is because it can be interpreted as the neural network's confidence in its prediction [15] in a classifier. The output node with the largest value, thus, corresponds to the class that the neural network is most confident that the data represents.

When training neural networks one must ensure that there is enough variation in the training data so that the model does not become "overfitted". An overfitted model performs well on known data but not so well in general; see Figure 4. By training the network, we are essentially shaping the output of the activation functions and the combination of these activation functions [15]. Neural networks can be trained using two different methods: supervised and unsupervised learning. Supervised learning is comparable to a teacher giving a student some task and actively providing feedback as opposed to unsupervised learning which would be comparable to a teacher giving a student a bunch of blocks of different shapes and sizes and telling them to organize the blocks, allowing the student to decide what the best grouping is, on their own.

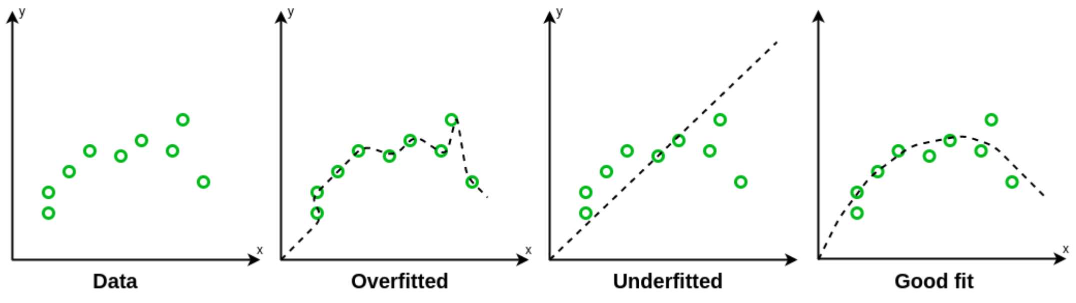


Figure 4: Non-linear regression fitting.

2.1.2 Supervised Learning

Supervised learning is the first type of machine learning, and it is the process of learning a mapping from some input data to some output data. In essence, supervised learning is done by comparing the network's prediction with what the expected answer was; and while we can simply say "true" or "false", it helps to be able to quantify to what degree the prediction was wrong. This is done by calculating the loss (or cost), using what is called a loss function (or cost function). If we have a neural network model that is being trained to predict the position of a license plate in a picture, and it predicts that the license plate is positioned at $x = 458, y = 50$, while we know that the license plate is actually at $x = 430, y = 56$ (the expected value is called "ground truth"), we can define a loss function like one of the following formulas. [11]

$$
\begin{aligned}
loss &= \left|x_{pred} - x_{ground\_truth}\right| + \left|y_{pred} - y_{ground\_truth}\right| \\
&= |458 - 430| + |50 - 56| \\
&= |28| + |-6| = 26 + 6 = 32 \\
loss\_squared &= \left(x_{pred} - x_{ground\_truth}\right)^2 + \left(y_{pred} - y_{ground\_truth}\right)^2 \\
&= (458 - 430)^2 + (50 - 56)^2 \\
&= (28)^2 + (-6)^2 = 784 + 36 = 820
\end{aligned} \tag{1}
$$

The squared loss is useful because it is naturally disproportional, so a larger error will be disproportionately larger than a smaller error.

When we know the loss, we can look at which weights contributed to the loss and adjust those in proportion to how much they each contributed to the error. We want to do this with large enough steps that the loss function is pushed down towards zero (or as low as it goes); but with small enough steps so as not to overshoot and end up jumping back and forth over the minimum. The most widely used learning algorithm for supervised feed-forward neural networks is called back-propagation.

Back-propagation was first introduced by Linnainmaa in his master's thesis [16]. The context of Linnainmaa's master's thesis was not one of neural networks, but the ideas he covered can be applied to neural networks. The idea of back-propagation is to minimize the error of nested functions using the chain rule. Back-propagation is done by calculating the error (loss) of the output (final layer) using a loss function, determining the gradient of the loss function given the current parameters (weights and biases) for each neuron for each layer, and propagating the error backwards as one is traversing the weights backwards to the start of the network. Ultimately, the gradient (the derivative) of the loss function with respect to the parameter (the weight

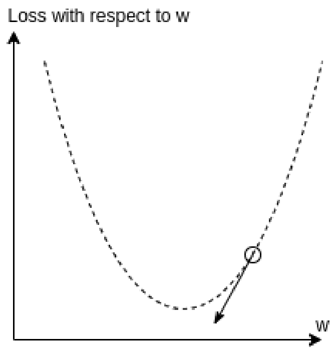


Figure 5: The gradient of a loss function, with respect to the weight w, is minimized using back-propagation.

or the bias) in question is multiplied by a learning rate (to slow it down, for accuracy) and added to the parameter. This pushes the loss function to the local (ideally global) minimum. When inching closer to the local minimum the gradient will approach zero, and the training will converge (slow down/reach a minimum).

2.1.3 Unsupervised Learning

Unsupervised learning is the second type of machine learning and, unlike in supervised learning, the machine learning method is never told what the answer is supposed to be; rather, it is simply given some data to process. The goal with unsupervised learning is to discover patterns in the data. For example, the classification graph in Figure 1 shows linear function classifying datum as male/female based on which side of the line they land. This is a problem which could be solved using clustering, which is an unsupervised learning method, as well [11].

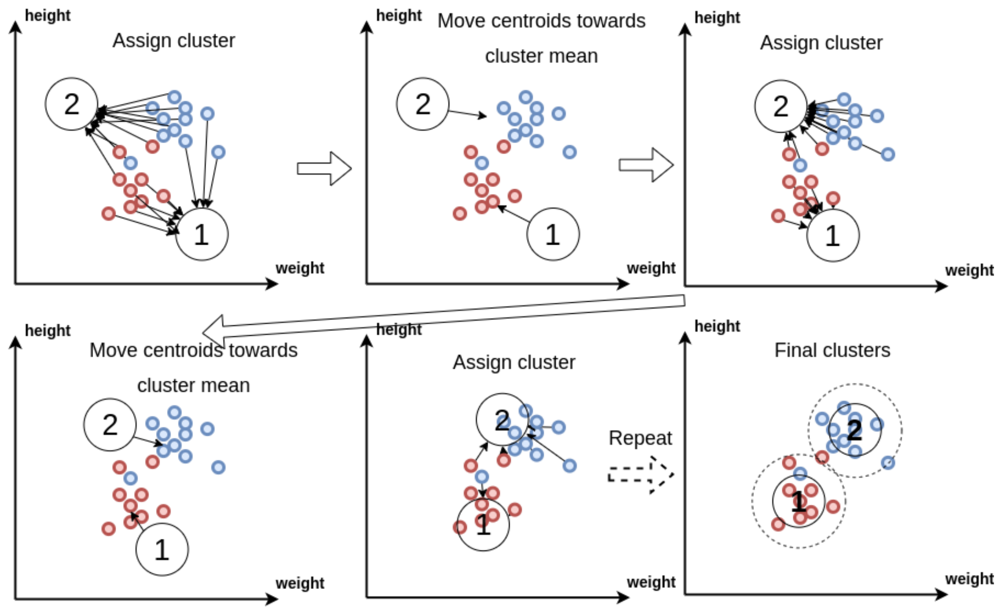


Figure 6: K-means clustering solving the weight/height to gender classification problem.

One popular type of clustering is the K-means algorithm (or K-means clustering), which has one objective, to find K (an arbitrary number) groups (clusters) of similar datum in a data set; see Figure 6. K-means clustering first creates K "centroids" at random, assigns each datum to its closest centroid and then shifts the centroids towards the mean of its cluster. The two latter steps are repeated until the centroids have stabilized or until the maximum number of iterations have been performed. Given the height and weight to gender relationship, we could give the K-means a K of 2 and expect it to identify the centers of each cluster (but it would not know which is which) [11].

2.1.4 Fully Connected Neural Networks

A fully connected neural network is one where every neuron is connected to every neuron in the next layer. This is quite many connections and can lead to what is often referred to as "the curse of-dimensionality" [17]. If one takes an image of size 28×28 such as those in the MNIST data set of hand-drawn digits [18] and passes them to a fully connected first hidden layer, it means that each *neuron* will have $28 \times 28 \times 1 = 784$ weights. If there were three color channels instead of only one, it would mean $28 \times 28 \times 3 = 2352$ weights. If one takes an image of size 640×640 such as those used in YOLOv5 (which will be discussed later), this means $640 \times 640 \times 3 = 1'288'800$ weights per neuron. This very quickly runs away from the realm of computability. It is easy to see that fully connected neural networks are better suited for tasks with fewer input variables, but with complex relationships between these variables.

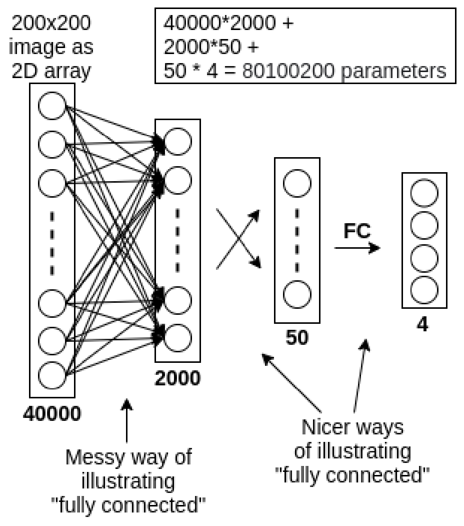


Figure 7: A fully connected neural network.

Another problem with using fully connected neural networks for visual tasks is that they are not necessarily very good at generalizing, as they tend to end up classifying on very specific things in the image, instead of recognizing and combining features. This means that they tend to overfit more easily, by factors such as the exact position of an object, instead of only focusing on the features of that object regardless of its position. Looking at the examples in Figure 1 they both only have two dimensions, but feeding a neural network an image means that the solution space has one dimension per pixel. Finding the global/local minimum of such a solution space is very complex. This is where convolutional neural networks come into the picture, as they try to recognize the features of the image and generally reduce the amount of variability for each layer while being better at generalizing. [15]

One method that has been shown to reduce overfitting in neural networks is using a dropout layer [19]. A dropout layer is simply a layer that during training has some defined probability of setting the activation function of a neuron in the previous layer to 0. The effect of this is that all neurons need to spread their weights more evenly between all the incoming connections in order to still be able to perform correctly, making it generalize slightly better. One disadvantage of using dropout is that the loss function will not necessarily point the gradient in the correct direction when suddenly dropping some neurons. [19]

2.1.5 Convolutional Neural Networks

The convolutional neural network was first introduced as the "neocognitron" by Fukushima [20]. His work was inspired by the theory of how the visual nervous system of a cat works, proposed by Hubel and Wiesel [21]. The architecture he proposed (see Figure 8) is one where the network recognizes different features in an S-layer (simple layer), reducing the size of the input and then feeds those features into a C-layer (complex layer) which makes sense of the combinations of features, which then is fed into another S-layer and so on. The cells in the S-layers only feed their output forward into one C-cell in the next C-layer, but each C-cell feeds its output to all the S-cells in the next layer [20]. This specifically, means that each C-cell will be responsible for detecting a certain feature and, naturally, the deeper one goes the more complex the features become. The convolutional networks of today have evolved since this point but the idea of combining simpler features to recognize more complex features is still the same.

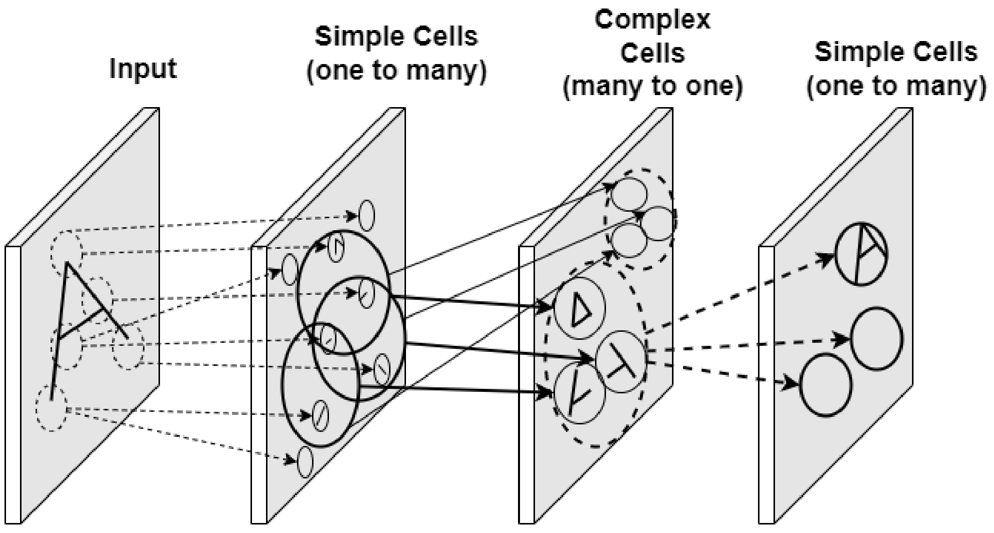


Figure 8: The neocognitron.

Convolutional neural networks are built from convolutional layers, which in turn consist of a series of weight matrices, known as "kernels". The output of a kernel is called a "convolution" (or a "channel"). The kernels are used in matrix multiplications together with the input image (or convolutions from previous layers). More specifically, a convolution is the scalar product of a weight matrix with every neighborhood in the input. In the convolutional layer, the kernel is slid over the image to compute all the scalar products as shown in Figure 9. As we can see in the figure, the output is the size of the freedom of movement that the kernel has in the image. The kernel in the example has a "stride" of one, meaning that the kernel moves over one pixel at a time, but the stride can be larger than one. A larger stride means less freedom of movement, which means that the output will be smaller. With any kernel larger than 1×1 the output will be smaller than the input. Sometimes it can be useful to have the output be of the same size as the input, and then "zero padding" can be used (if nothing else then because it is easier to calculate of what dimensions the output will be). Zero padding means that zeros are added around the image/feature-map so that the kernel can compute a value for each pixel.

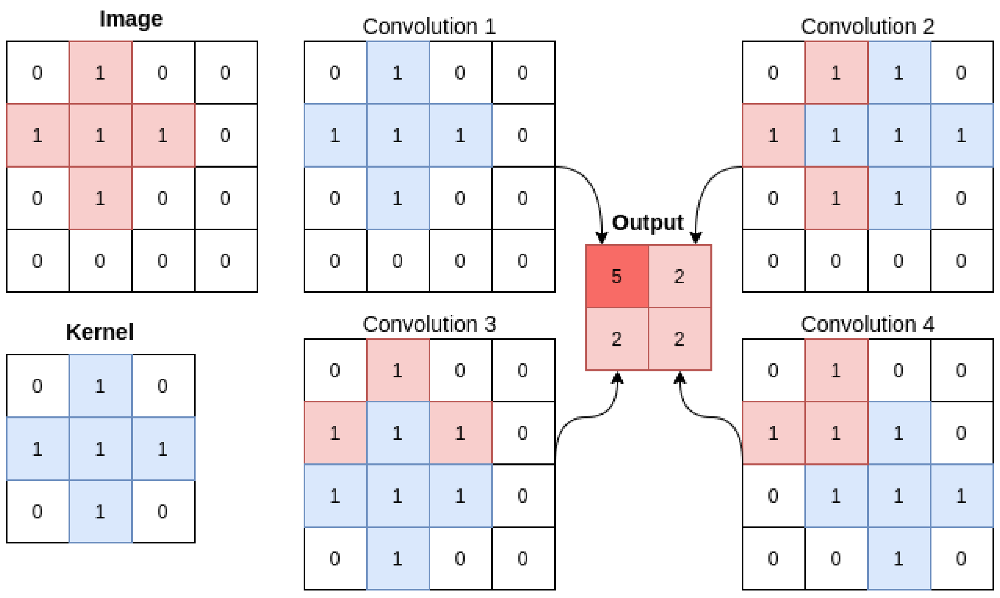


Figure 9: An illustration of the process of a convolutional kernel.

In a multichannel image (RGB has three color channels), there is one kernel per channel per filter. Using the cross-shaped kernel filter example in Figure 9 there would be one such kernel for each channel, each doing the same things, as illustrated in Figure 10. It is important to note, however, that the kernels can end up with entirely different weights, depending on the color (channel) they correspond to, so it might not look like a cross for the other channels. Then, the values of all kernels are summed up and a scalar (1×1) bias is added to the sum.

In order to detect larger structures the convolution output of one layer can be

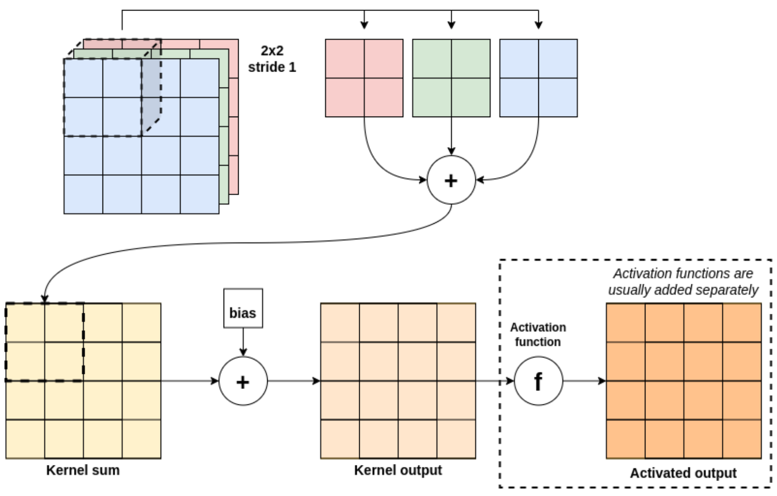


Figure 10: The workings of a multichannel convolutional filter.

downsampled before being passed to the next. For example, given an image of size 32×32 and a 4×4 kernel; using zero padding, the output will be a 32×32 sized output. If we downsample that by half to 16×16 we can now use the same size kernel to detect the same features that are twice as large.

Max pooling is a downsampling filter often used in convolutional neural networks. It preserves the strongest features detected by simply preserving the largest value in an area of interest, rather than performing some kind of averaging and making the activation of the convolution diluted. See Figure 11.

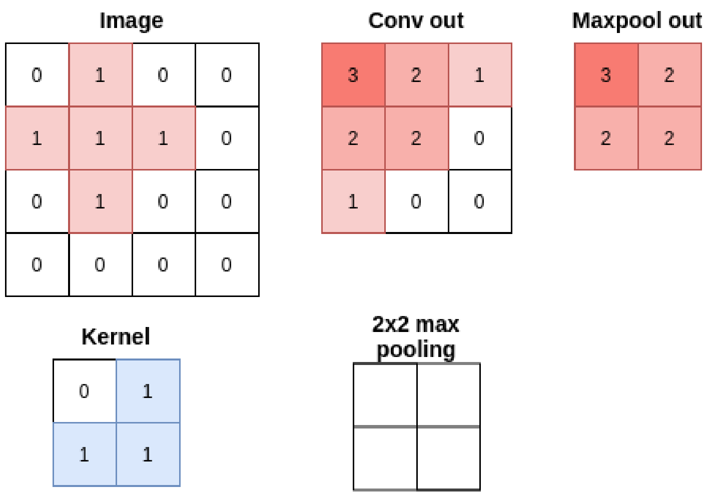


Figure 11: Max-pooling example. The result shows that the corner shape of the kernel is most present in the top-left corner of the image.

The weights in a kernel are adjusted by training [15], just like any other neural network, and it is usually done with back-propagation as shown in Figure 12.

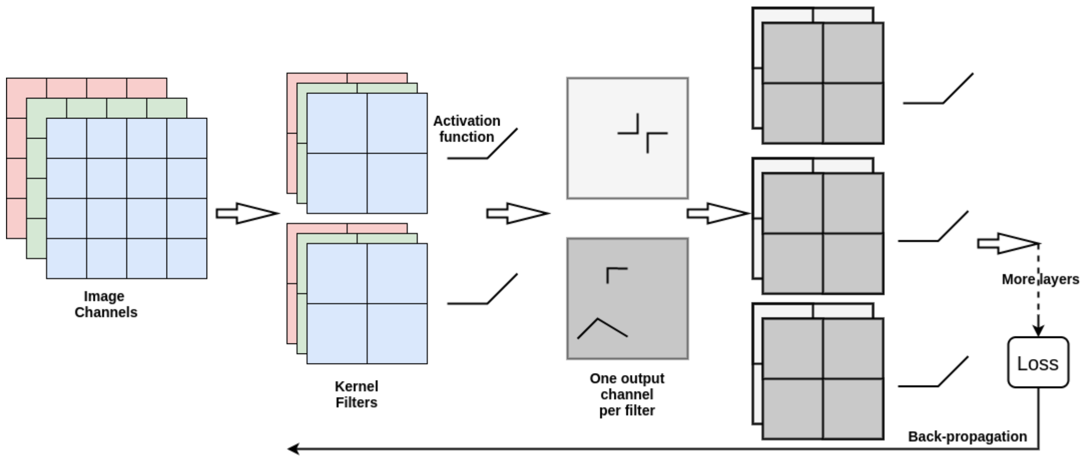


Figure 12: Convolutional neural networks can also be trained using back-propagation.

Because the kernels in convolutional neural networks, usually, are two-dimensional and there are multiple kernel filters per convolutional layer that each produce their own two-dimensional output; the result is, naturally, three-dimensional. These three-dimensional output volumes are usually represented by cubes, like in Figure 13. The number of convolutions produced is the same as the number of kernel filters per layer. In order to get some form of classification out of a CNN, we can look at the example Figure 13 where, as we go further down in the network, the depth dimension is growing, and finally the convolutions are fed into two fully connected perceptron layers that are responsible for the classification.

However, as discussed, fully connected perceptrons are prone to overfitting and, thus, Lin et al. [22] proposed using a 1×1 kernel filter instead of perceptrons, which they call "Global Average Pooling". The advantage of this is that it makes the classifier look at the same pixel in each channel (refer to Figure 9), as opposed to the neurons in the fully connected perceptron which could compare any combination of pixels in all the channels. This should naturally force different areas, through the whole chain of convolution kernels, to specialize on specific features. This should make training faster, and Lin et al. were indeed able to show that it does. They were also able to show that this works better than using only dropout. [22]

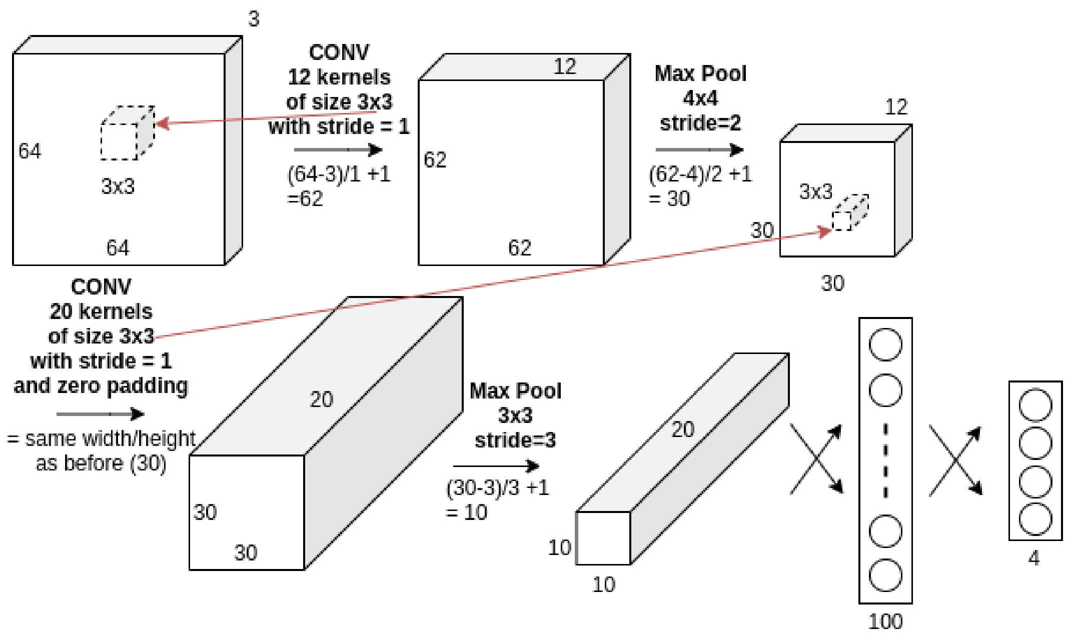


Figure 13: Illustration of a convolutional neural network and the calculations determining the dimensions of the channels. This example is a classifier with four classes.

2.1.6 Revisiting activation functions

Now that we have a general understanding of how neural networks work, it is useful to revisit activation functions (see Figure 3 for reference) to understand why one would be chosen over another. What we need out of an activation function is non-linearity, because our input data will not be linear, so we need a way to adapt to that. It does not matter how many times one chains the function $f(x) = w * x + b$ or what value one gives either of the parameters; the resulting shape will always be linear, which means that it could be modeled using a single neuron, so that is not very useful. It also helps if the activation function has some range where it is more sensitive to change.

The sigmoid function has been quite popular, it is very much non-linear, and the rate of change is lower the further away from $x = 0$ ones goes. It also does not change much at all below -5 or above +5 (with the default parameters, though these may be adjusted), so no matter how strong the input is, the rate of change (gradient) is not going to be very large outside that range. This can be desirable e.g. if one wants a confidence value, because the model should not very easily reach 100% confidence. Thus, the sigmoid function is often used for the final layer. However, when we are talking about neural networks with multiple hidden layers, we want to optimize them using exactly the gradient and if many neurons end up outside the areas with strong gradients, training will be slow. This is both desirable and undesirable for a neural network. Desirable because we want to be able to find a good combination of parameters and stay there and undesirable because the training can stagnate if too many neurons end up outside the active range. The sigmoid function also takes a few computations to calculate compared to simpler activation functions.

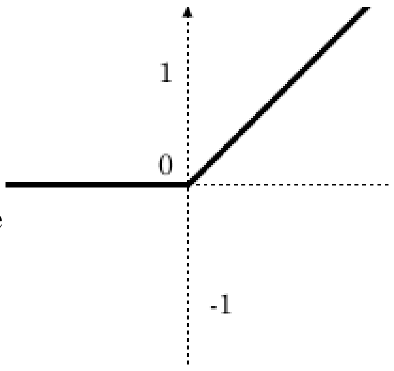
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The hyperbolic tangent (tanh) function is similar to the sigmoid function, but it is centered around 0 instead, which can be advantageous if the inputs are normalized around 0. The derivative of the tanh function also has a larger peak (larger gradient = faster training). The tanh function, however, has the same disadvantage as the sigmoid function, that is to say that tanh also becomes saturated towards the extremes, and it is slightly more complex than the sigmoid function.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The rectified linear unit (ReLU) is another alternative, and it is computed as follows:

$$relu(x) = \begin{cases} x & \text{if x > 0} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$



Mathematically speaking, the derivative is undefined at the point where the ReLU function bends, but in a computer program one can simply define that point to be 0 and that solves that problem. ReLU is non-linear, even though it locally looks linear, because of the bend where the direction of the function suddenly changes. The first reaction might be that this will be as useless as $y = x$, but since non-linearity is what we need to be able to model complex functions, this is also possible with ReLU, as shown in Figure 14.

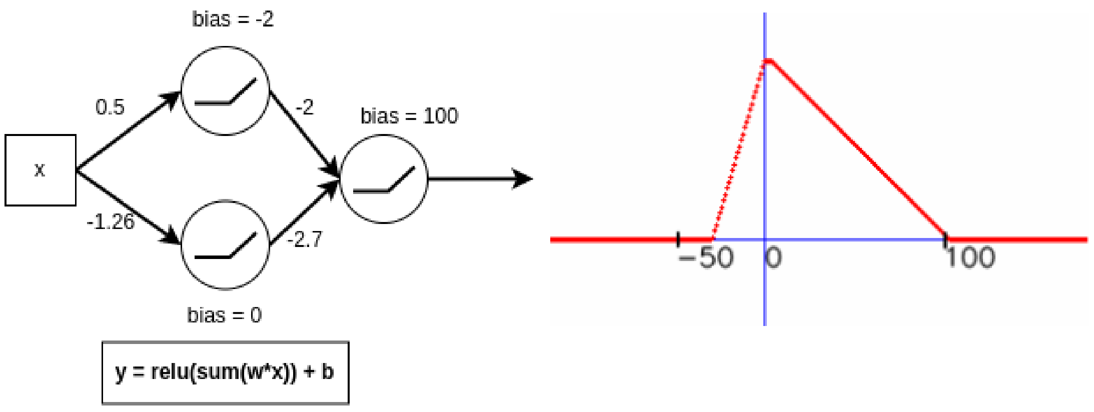


Figure 14: A combination of ReLU neurons creating a wedge shaped activation.

ReLU has become quite popular for deep CNNs and there are several variations of it. A "leaky" ReLU is one where y is allowed to go below zero, but slowly. The reason for using leaky ReLU is to avoid "dead neurons" that simply output 0 for any (reasonable) input [23]. A very wrong answer will be punished more than a slightly wrong answer when using a leaky ReLU.

$$leaky(x) = \begin{cases} x & \text{if x} > 0 \\ 0.1x & \text{otherwise} \end{cases}$$

A modern take on ReLU called "Funnel ReLU" or "FReLU", specifically meant for convolutional neural networks, was introduced by Ma et al. [24]. They claimed an improvement in accuracy of up to 1.5% by using FReLU over ReLU. The FReLU formula is $frelu(x_{ij}) = max(x_{ij}, T(x_{ij}))$, where x_{ij}, is the pixel and i and j are its coordinates in the image x. $T(x_{ij})$ is the value of the "parametric pooling window" (essentially a convolutional kernel that looks at each channel separately), centered on x_{ij} of the input x.

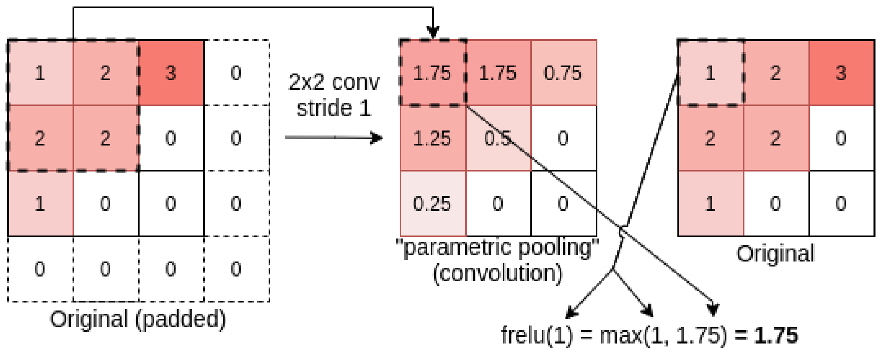


Figure 15: Funnel rectified linear unit.

2.2 Deep Learning and Recent Improvements

Deep learning is a subclass of machine learning specifically referring to deep neural networks. Neural networks tend to be able to recognize more complicated features and/or relationships in data the deeper they are, so deep learning simply refers to the topic of very deep neural networks [15]. The term deep learning was likely coined by Rina Dechter in 1986 [25] and first used in the context of neural networks in the year 2000 by Aizenberg et al. [26]. However, as early as 1962, Hubel and Wiesel [21] suggested that the feline visual cortex consists of layers of neurons where the early layers detect simple edges and the deeper in the visual cortex the neurons are the more complicated the detected features are. In 1980, Fukushima, inspired by the work of Hubel and Wiesel, introduced the concept of neocognitrons, today known as convolutional neural networks [20]. The concept of convolutional neural networks is the very foundation of most state-of-the-art object detectors [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40] and, per implication, the topic of deep learning itself, to a large degree.

2.2.1 Batch Normalization

For any neural network, it is useful to normalize and/or standardize the data before passing it to the model. Normalization means to adjust the size of the input values to a more usable scale (usually 0-1). For example, if we are trying to predict the number of years a person will live (from today) by the variables *age*, *daily caloric intake* and *kilometers driven per day* (in a car); then these values are likely to all be of different magnitudes. The age is in all likelihood going to be 0-100, the daily caloric intake 1000-10000 and kilometers driven 0-1000. These variables would all be mapped, respectively, such that the lowest value present is 0 and the highest is 1, and the rest of the values are extrapolated linearly between 0 and 1. If we do not normalize this, the extreme of a thousand kilometers driven per day will have a much stronger impact on the prediction than an age of 100. Standardization is the process of calculating the mean and standard deviation of the input parameters, subtracting the mean and dividing by the standard deviation. This will make the new mean 0 and the new standard deviation 1.

Another problem with neural networks is that the covariates of the data in a data set may be shifted in some direction away from the true mean and variation values. This is called "covariate shift" [41]. In rough terms, if we are trying to train a classifier to classify cats, but during training we are only giving it images of black cats, the covariates of these images will be slightly different from the covariates in a set of images of gray or orange cats. Meaning that our classifier might no longer be convinced that the images are of cats. By standardizing the data this would be solved for the first layer, but the output of one layer is not guaranteed to be standardized even though its input was.

This is where batch normalization (BN) comes into the picture. It was first introduced by Ioffe and Szegedy in 2015 [42] when they realized that the current standard was to normalize/standardize the input, while the internal layers also experience the same problem of covariate shift, which they call "internal covariate shift". Batch normalization looks to address internal covariate shift by looking at a whole batch at a time, and for each layer determining the mean and standard deviation and standardizing the output. While training, if we are updating the weights after every single evaluation, this can cause the weights to be updated in the wrong direction, or the loss function to become stuck in a local minimum. If we instead consider multiple evaluations at once, as a batch, we are much more likely to be updating the weights in the correct direction. Because modern GPUs are good at parallel processing, running multiple evaluations at the same time will also be faster than running them one at a time.

The first step is to, for every neuron in the network, calculate the mean μ_B

and variance V_B of the weighted input sum x_i (the paper [42] does not specify if this is done before or after the activation function, but let us assume before) for the entire batch B and normalize that ($\rightarrow \hat{x_i}$).

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$V_B = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{V_B + \epsilon}}$$

Where ϵ is a small number to prevent division by zero. They also introduced two new parameters, γ and β, that are trained together with the network.

$$y_i = \gamma\hat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i)$$

This might seem somewhat counter-intuitive, that we first normalize the input and then modify it again, but the idea behind it is that it might not actually be ideal to have $\mu = 0$ and $V = 1$ but scaled and shifted slightly. Then the γ and β parameters can learn what the appropriate shift is and adjust it. They also presented a second algorithm, but understanding the first one is enough for us and batch normalization is already available in modern machine learning frameworks (e.g. PyTorch).

Ioffe et al. were able to show that, when using batch normalization, they only needed 7% of the training time to achieve the same performance compared to without batch normalization (given the specific network they tested on). Batch normalization also effectively eliminated the need to use dropout layers, which was the previous gold-standard for reducing over-fitting and improving training performance, because BN also helps with the problems of becoming stuck in local minima and vanishing/exploding gradients.

2.2.2 Residual Learning

In accordance with the theory presented thus far: increasing the number of layers in a neural network means that it will be able to detect more patterns. However, when training very deep neural networks, He et al. [36] noticed that a deeper neural network showed a higher error rate after the same number of iterations and using the same data set, than a shallower one. This is strange, because if we have a neural network that is 20 layers deep and add another 36 layers to those (these are the numbers they tested with in the paper); then the 56 layer network should be able to perform at least as well as its shallower counterpart, given that neural networks are nothing but nested functions. They had already addressed the potential problem of vanishing gradients by using batch normalization and ruled out overfitting because that would mean the error rate would go up after going down, which it did not. Since simply adding more layers *increased* the error rate, there was clearly something else at play. [36]

If the layers in neural network A can learn a function, and if the layers C that are added after A can be initialized as the identity function (the function that given x returns x), there should be no difference in learning rate. [36]

Thus, if some set of layers can learn to approximate a function $H(x)$, they should also be able to learn to approximate the residual function $F(x) = H(x) - x$, and we can write the original function as $H(x) = F(x) + x$. If it is optimal to have the residual be zero, the weights can adapt to this to force it down. While both functions, $F(x)$ and $F(x) + x$, should be equally capable of mapping the desired function they might affect learning differently. This residual connection is achieved by adding a shortcut connection between every N layers, and adding the output x of layer L_i to the output of layer L_{i+N} (see Figure 16). In their paper they use the term "building blocks" for the blocks (see Figure 16) between the connections but as that is quite general, they are usually referred to as "residual blocks". The shortcut connection is only implied and does not rely on any extra parameters being introduced. The only difference is a few element-wise additions, so this modification comes very cheap. They hypothesized that, if implemented, each layer would actually need to alter the

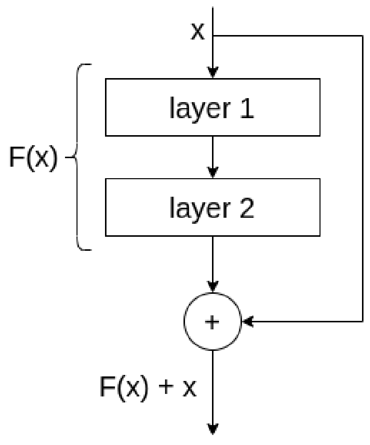


Figure 16: The residual block.

incoming data less (compared to non-residual networks), because the identity function already is a quite good answer. [36]

One problem that comes up with residual blocks is that the dimensions of the layers to be added need to match. They proposed, and tested, three different options for addressing this problem. The first is to simply zero pad x to be of the same size (e.g. concatenate channels with only zeros until $F(x)$ and x are of the same dimensions). The second is to increase the number of channels with a projection shortcut (using 1×1 convolutions) whenever there is a difference in dimensionality. The third option is to always use the projection shortcut instead of the addition, but this introduces many extra parameters and does not improve performance significantly enough to justify the cost over using the second option. The second option yields better performance than the first. [36]

For an architectural comparison they compared a 34-layer model ResNet with a previous state-of-the-art model, VGG-19. The VGG-19 model (19 layers) requires 19.6 billion FLOPs (floating point operations) for one inference and the ResNet model requires 3.6 billion FLOPs. In the performance comparisons they compared with VGG-16 (15.3 billion FLOPs), but nonetheless the ResNet model performed significantly better with only 23% of the floating point operations (and some models with more layers performed even better). [36]

Because the networks were easier to train they were able to reduce the size of the first convolutions faster than VGG. In the beginning of the network VGG-19 has two (3×3, 64 filters, stride 1) layers working on 224×224 images/channels, pooling with a stride of two to halve the size (in both x and y) and doubling the channels, followed by two (3×3, 128 filters, stride 1) layer followed by another pooling to halve/double and arrive at 128 channels of size 56×56. Whereas the 34-layer ResNet model starts off with a (7×7, 64 filters, stride 2) layer and then pooling that with a stride of two, to halve, and arrive at the same 56×56 channel dimensions, but with 64 channels instead of 128. The VGG-19 model also increased the channel size much faster and had a total of 8 (3×3, 512 filters, stride 1) on channels with dimensions 28×28 and 4 on channels with dimensions 4×14. The ResNet model only has one layer doing (3×3, 512 filters, stride 2) on 14×14 channels to bring it down to 7×7 channels, and then 5 layers of

(3×3, 512 filters, stride 1). Finally, the VGG-19 model adds two fully connected perceptron layers, each with 4096 neurons to the 512 7×7 channels and then a 1000 layer fully connected to that, whereas the 34-layer ResNet model only fully connects a 1000 neuron perceptron layer to the 512 7×7 channels. These changes meant that they were able to train deeper models with fewer parameters. [36]

As the entire point of the paper was training very deep networks they were becoming concerned with how computationally heavy the models were becoming and also proposed a "bottleneck" building block, which first projects the incoming convolutions down to a smaller number of channels before computing the convolution, and then projecting it back up again, as shown in Figure 17. In the example bottleneck, 256 channels are projected down to 64 using a 1×1 convolutional layer. This is 9 times cheaper than a 3×3 convolution, and now the 3×3 convolution can be performed on the 64 channels instead of 256 channels, which is 4 times cheaper. [36]

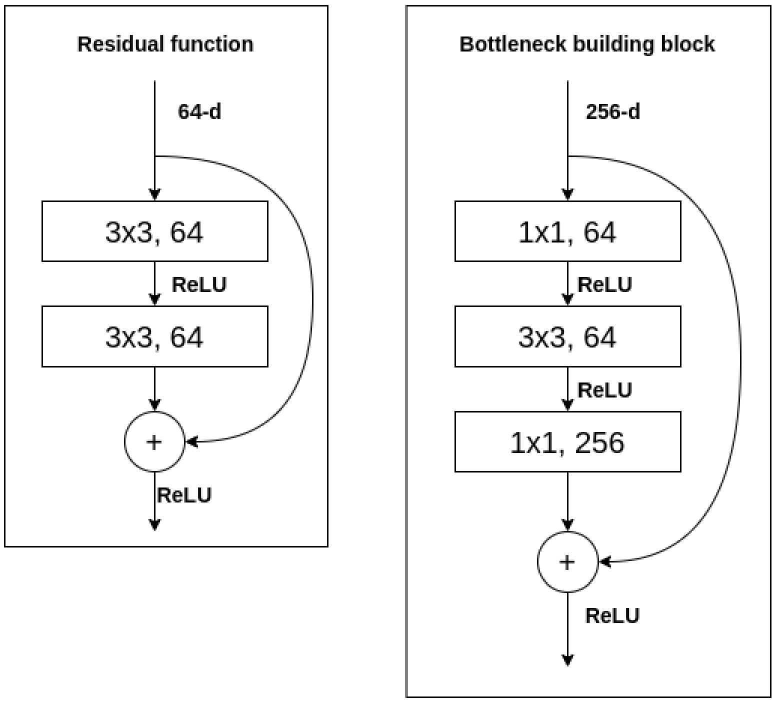


Figure 17: A bottleneck building block compared to a residual function.

Finally, they looked at the magnitude of the responses of the layers in the ResNet models and found that their hypothesis about the individual layers not needing to change the input significantly stood true. The models they tested without residual connections were exhibiting much stronger activations (more changing of the input data) than those with the residual connections. Afterwards, when comparing the training errors between a 34-layer network and an 18-layer network they were able to see what they expected to find at first. By adding more layers the training error goes down. Unlike models without the residual connections, when going from 18 layers to 34 layers increased the training error. [36]

Thanks to these ideas they were able to train deeper neural networks and obtain a relative performance increase of 28% on the COCO object detection data set and won the 2015 ImageNet competition. Their paper has also been one of the most influential papers in deep learning with over 36000 citations to date. [36]

As a side-note, Huang and Sun et al. [43] were able to improve the training efficiency by re-introducing "dropout", in a new form that they call "Stochastic Depth". Instead of dropping individual neurons like in multi-layer perceptrons, they proposed dropping individual residual blocks (randomly, during training), such that only the identity function was passed forward. This makes the gradient less diluted during training and, thus, makes training faster, while also being inherently faster to train due to having fewer parameters. They saw training time reductions of about 25% while using stochastic depth; and while He et al. [36] concluded that their 1202-layer model was overfitting (7.93% error rate on CIFAR-10 [44], whereas their 110 layer model had a 6.43% error rate), Huang and Sun et al. [43] were able to successfully train a 1202-layer model without overfitting (a 110-layer model showed a 5.25% error rate and a 1202-layer model showed 4.91% error rate).

2.2.3 DenseNet

Inspired by the recent progress in the ability to train ever deeper neural networks, Huang and Liu et al. brought yet another improvement (partly the same authors that brought stochastic depth [43]). This time they proposed "Densely Connected Convolutional Networks", or DenseNet [37]. As they were able to show in their paper on stochastic depth, some layers can be dropped while training and still improve the performance of the network. This aligns with what He et al. pointed out in their ResNet paper [36] (about most layers only needing to manipulate the input very slightly). This led them to a more explicit way of preserving the previous information than ResNet does with the residual. Instead of adding the identity to the output, they concatenate the previous convolutions and reduce the number of filters per convolutional layer; see Figure 18. This means that there will be $\frac{L(L+1)}{2}$ connections in an L-layer network instead of just L like there usually would have been. However, to prevent the networks from growing too large they use a low, constant, growth rate (each convolutional layer adds K new channels) as opposed to the previously popular doubling of channels whenever the dimensions are halved.

When the size of the feature maps change, concatenation is no longer possible. They then use a transition layer consisting of a batch normalization layer, a 1×1 convolutional layer and a 2×2 average pooling layer.

The main advantages of DenseNet are that there is less redundancy by not producing more convolutions than really are needed, and the information flow is better as each layer has explicit access to the same information as the previous layers (this leads to stronger gradients and more efficient training). They also found that these dense connections reduce overfitting with smaller training set sizes. DenseNet achieved 3.46% error rate on CIFAR-10 [44].

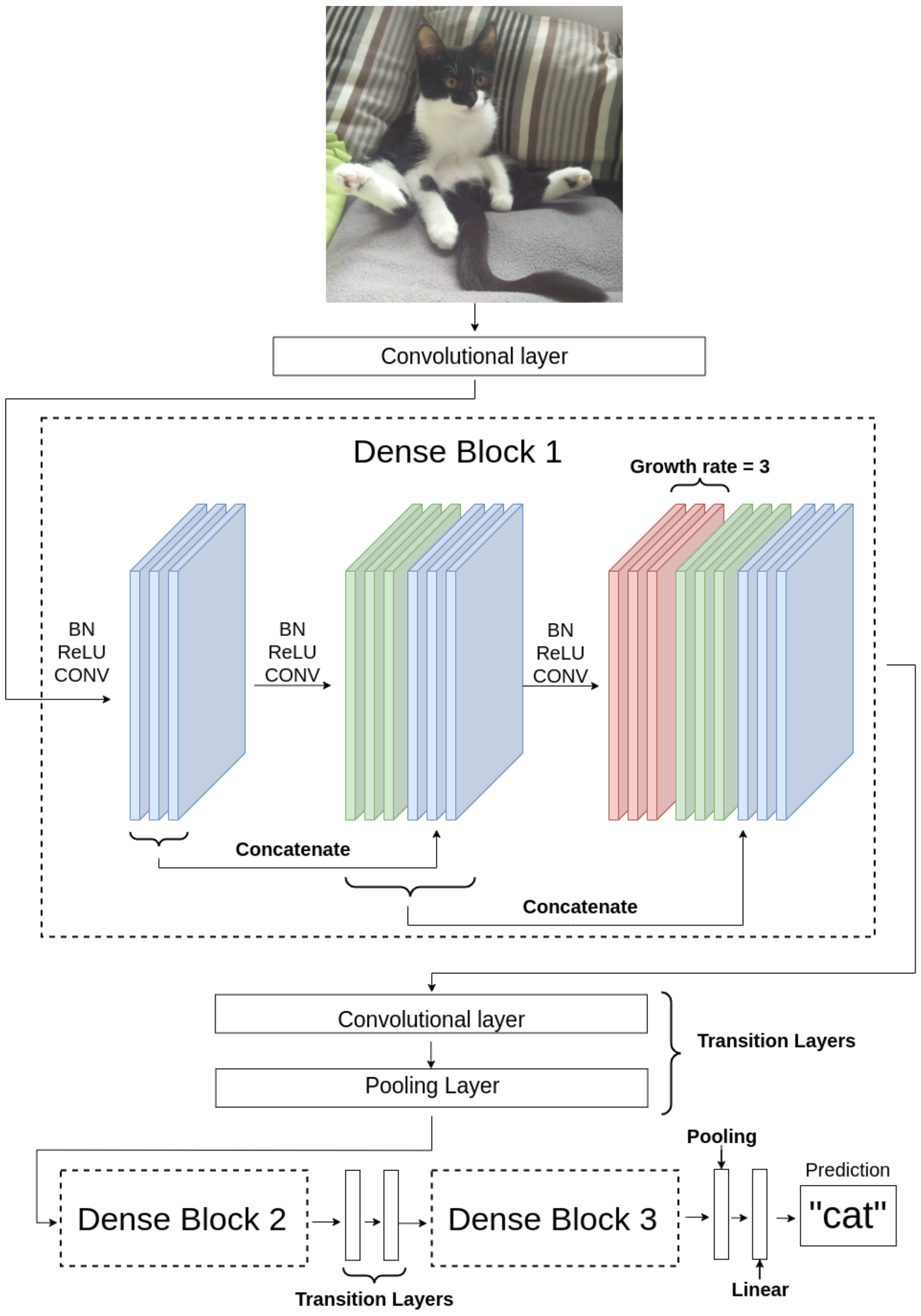


Figure 18: A DenseNet with three dense blocks

Further improving on DenseNet, Wang et al. [45], brought about the idea of splitting the given input channels passed to the dense block into two parts, such that one half is passed into the dense block and the other half is concatenated to the output of the dense block, in the transition layer; see Figure 19. They call this "Cross Stage Partial Networks" (CSPNet), or when applied to DenseNet, "Cross Stage Partial DenseNet". By adding a shortcut between the input and output this method provides another path for the gradient to flow, meaning that the training efficiency should be improved. With this change, applied to any model (at least those they tried in the paper), one can reduce computational load by at least 10% (the largest reduction observed in the paper was 22%) while maintaining the same level of accuracy (or increasing it). They mention that part of the performance gains come from removing computational bottlenecks by distributing the amount of computation of each CNN-layer more evenly.

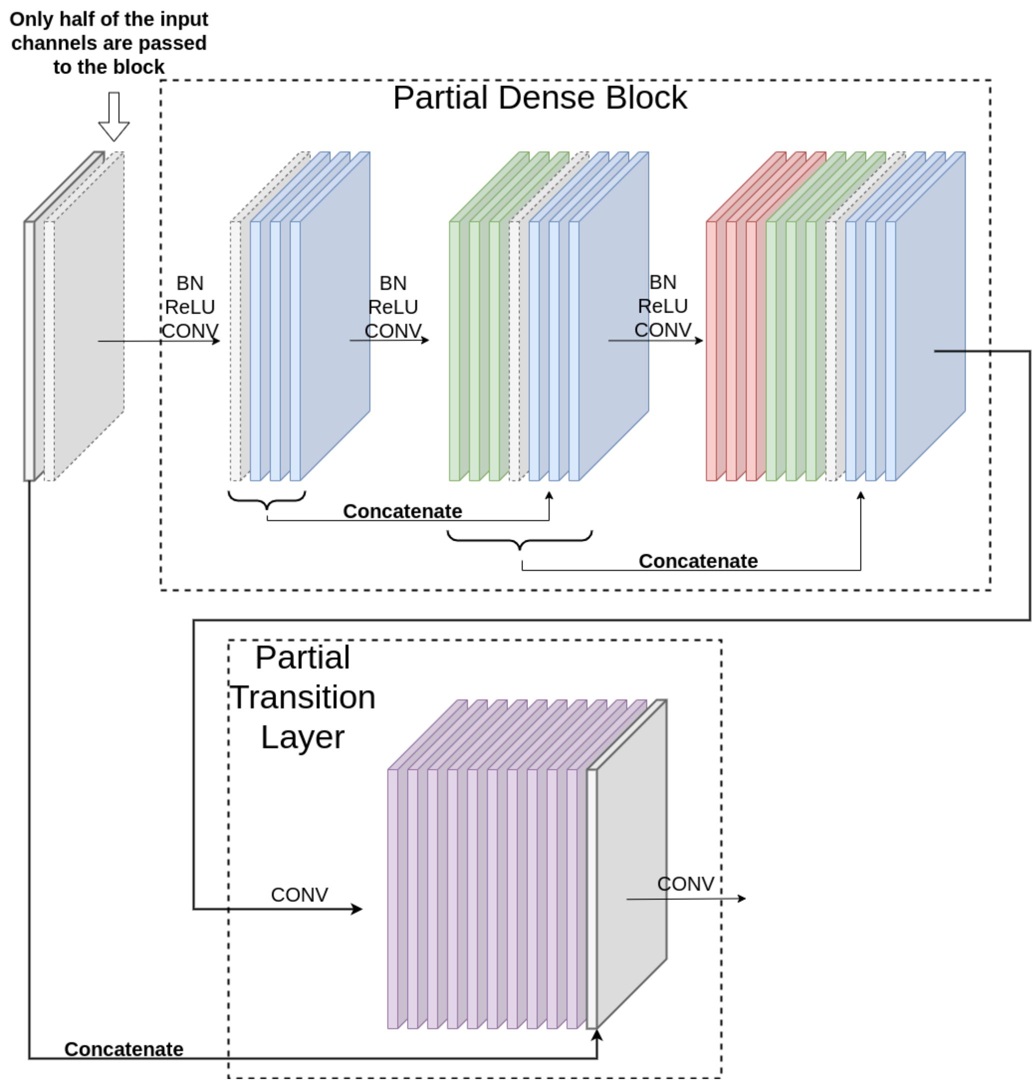


Figure 19: A Cross Stage Partial DenseNet block. The input channels are split into two parts, one is passed to the dense block and the other is concatenated to the output of the dense block they are all processed by the partial transition layer.

2.2.4 Spatial Pyramid Pooling

Spatial Pyramid Pooling (SPP) is the method of taking a set of feature maps or an image of any dimension, and pooling said feature maps with kernels of different sizes, in order to preserve information of different scales while transforming the data into an input-independent shape. This was introduced by He et al. [46], because the standard at the time was to crop and/or warp the input image before passing it to the network to achieve a constant shape, and those methods imply loss of information. They concluded that the only part of the network that really needs a specific size is the fully connected layers at the end of the classifier/detector. Convolutional kernels can operate on any feature map dimensions. Thus, they introduced SPP, which takes the convolutions and converts them to an expected shape by pooling on the feature maps, on different scales, but using kernel filters of dimensions relative to the input dimensions. Figure 20 shows an example of SPP turning a feature volume of shape $W \times H \times C$ into the expected shape $21 \times C$. Where W and H could vary from image to image but C would be determined by the architecture of the model or, for the first layer, equal to the number of color channels.

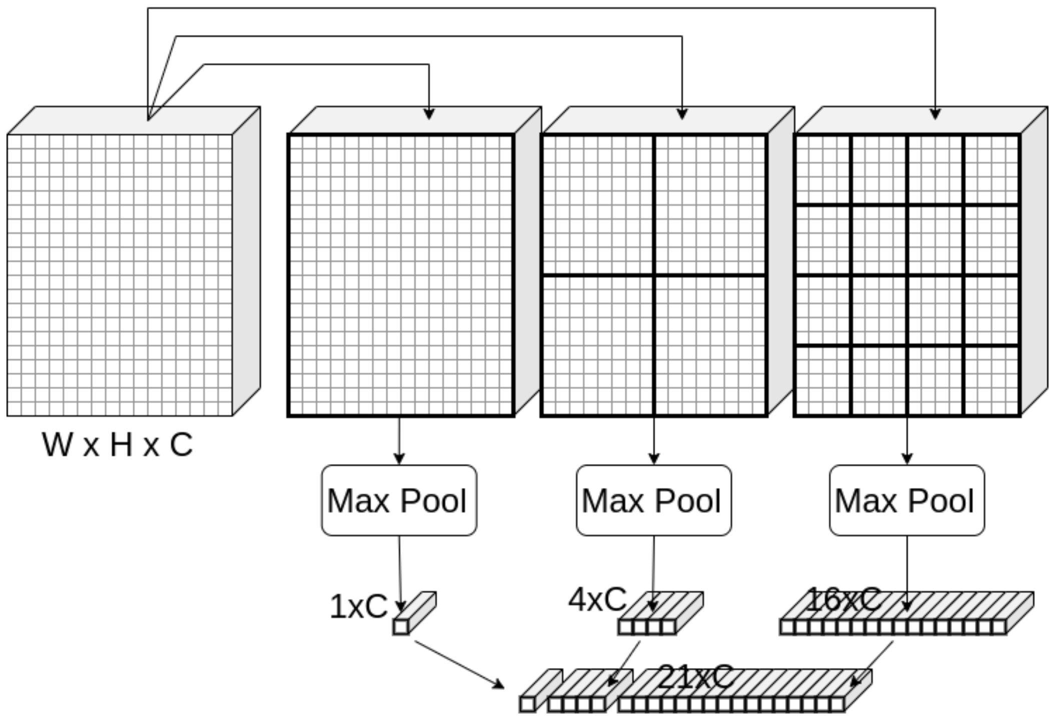


Figure 20: Spatial Pyramid Pooling as described in the original paper.

2.2.5 Feature Pyramid Networks

CNNs are, as discussed, inherently capable of detecting features independent of scale and locality in an image; but, the deeper a network becomes, the more the representations of each scale of the objects start to interfere with each other. Feature Pyramid Networks as introduced by Lin et al. [38] help make detection of objects at multiple scales more feasible by removing the implicit aspect and instead explicitly concatenating some earlier feature maps, of the same dimensions, to later feature maps to preserve the difference; see Figure 21. Applying this to Faster R-CNN they achieved state-of-the-art accuracy on the COCO object detection benchmark.

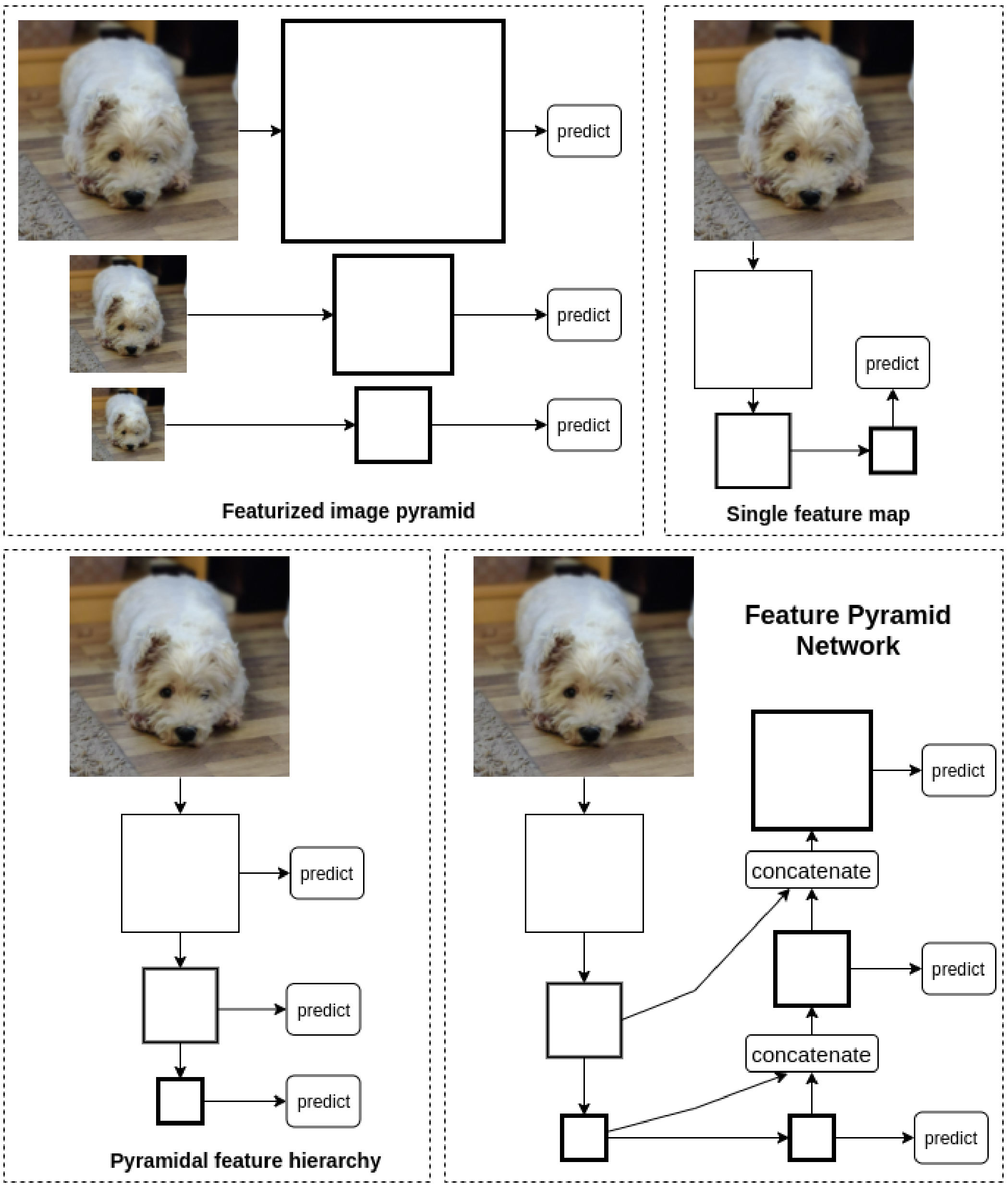


Figure 21: Feature Pyramid Network compared to previous feature pyramid attempts. Darker feature maps imply "stronger" features.

3 Background - Object Detection, Classification, Recognition and Localization

A paper from 1969 mentions several types of Automatic Vehicle Identification (AVI) systems designed and tested, but most of these required additional technology to be installed on the vehicle [47]. These systems ranged from relying on induction antennas buried in the road to microwave and acoustic transmitters with an identification code being encoded into the vehicle somehow. In this paper the authors also mention scanning the license plate using a "television camera", but without further elaboration [47]. This seems to be one of the earliest mentions of ALPR. In another paper from the 70s the cost of a single unit of these types of AVI systems were listed and ranged from between 2500 USD and 16000 USD (not adjusted for inflation) [48]. Most of these early systems were simply AVI however, and not ALPR, i.e. they do not rely on optical recognition of the license plate characters itself.

ALPR was allegedly first implemented in 1976 by the Police Scientific Development Branch in the UK, where its first purpose seems to have been to monitor traffic in and around London to help the police fight crime [49]. However, the interest in these kinds of systems started picking up in the late 80s/early 90s [50], and since then there has been an ever-increasing number of papers being produced on the subject. Many of these systems are slower and rely on some kind of trigger to start the system, but one of the first papers to mention real-time processing is one by Kanayama et al. [10] and they quoted a respectable 150 ms processing time. It was achieved using proprietary equipment and lights to illuminate the license plates, yet the performance of the system suffered noticeably during nighttime (90% recognition accuracy daytime vs 60% recognition accuracy nighttime).

3.1 Classification, Detection, Recognition and Localization

It is necessary to clarify the distinction between classification, detection, recognition and localization as they are entirely different things, and they are often referred to differently in different texts. In this thesis we are doing object detection, image classification and object localization. License plate recognition, is what we have ultimately achieved after determining exactly which license plate we are looking at.

Image Classification is the process of classifying the pattern in some data. For example, determining whether there is a cat in an image, without necessarily knowing where it is in the image, only that the image is one of a cat.

Object Detection is the process of detecting different objects in an image, creating a bounding box around each of them and classifying them.

Object Recognition is the process of recognizing a subtype of a type of object. Such as recognizing a specific face instead of just any face.

In our case, detecting any license plate in a picture corresponds to object detection, while detecting a string of literals (e.g. AZT-882) in the license plate object corresponds to object recognition; see Figure 22. Similarly, detecting a car is detection, but recognizing that a car specifically is a Honda is recognition. By this same logic, we are doing character recognition and not character detection. Confusingly, object recognition is often used synonymously with object detection, and vice versa.

Localization is the process of determining where some objects are in an image, an example of this is the motion detection algorithms discussed later in the thesis. Localization means that something is detected at some position, but nothing is necessarily known about what it is.

In this thesis we are not doing "Recognition" of specific license plates in one step, but we are performing several individual steps of detection and localization, which in the end yields a license plate recognition. All the detections in this thesis carry location (pixel coordinates) information, explicitly or implicitly, and all them are referred to as detections. Localizations will be referred to as such until they have been classified, at which point they will be referred to as detections.

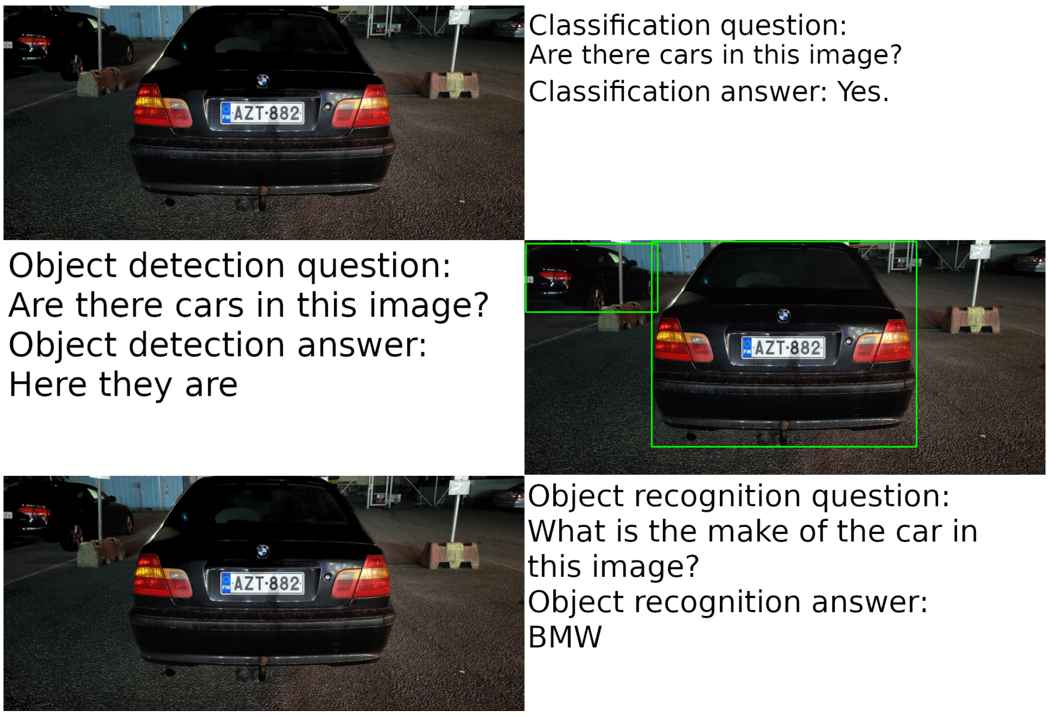

Classification question:
Are there cars in this image?
Classification answer: Yes.
Object detection question:
Are there cars in this image?
Object detection answer:
Here they are
Object recognition question:
What is the make of the car in this image?
Object recognition answer:
BMW

Figure 22: The differences between classification, object detection and object recognition.

3.2 Libraries for ALPR

3.2.1 OpenCV

OpenCV is an open source computer vision and machine learning library, and perhaps the most popular computer vision library as of writing this thesis [51]. OpenCV versions before 4.5.0 are licensed under the 3-clause BSD license and OpenCV versions greater than or equal to 4.5.0 are licensed under the Apache 2 license [52]. All the methods mentioned in this thesis will utilize OpenCV to some extent; some methods do more processing and some only load, crop, resize and show the images.

3.2.2 Tesseract OCR

Tesseract OCR is an open source optical character recognition software originally developed by Hewlett-Packard Laboratories Bristol [53]. It has been developed by google since 2006. Tesseract OCR works best on perfectly formatted, clean text, with correct perspective and no skew. As such, one can see that it will take some work to make this program to work nicely in our real world scenarios where the license plate is not always in the same location, the same angle to the camera or even in the same lighting conditions. Nevertheless, this kind of cleaning can be completed using OpenCV as described in later sections, but the cleaning results may vary significantly based on said conditions and, as mentioned, Tesseract OCR requires quite clean images to perform optimally. It should be mentioned that Tesseract OCR also incorporates machine learning as of Tesseract 4 as opposed to the more conventional methods previously used.

3.2.3 OpenALPR

OpenALPR is a License Plate Recognition Software. It was initially available only as a library that one could use in one's own programs but they later released a paid cloud service. Today, the benefit of using the cloud service over doing it oneself is that the models they provide through the paid service perform better than the models they provide for free. As of the writing of this thesis, the paid service is available through four different products: OpenALPR Watchman Agent, On-Premises SDK, Cloud API and Forensic Plate Finder [1].

The open source library The open source library is available on GitHub at `https://github.com/openalpr/openalpr` and it is the only freely available product by OpenALPR. The library is available for C/C++ but also has bindings for C#, Java, and Python. Using the libraries one can pass in single images at a time and obtain the result as single plate readings. Using the library one will have to implement the algorithm for grouping the plates oneself. It is not necessary to perform grouping for all purposes, but there are benefits to doing so. For example if one has a parking garage that needs to let people in based on their license plate it may be preferable to only send the request to the server when one is quite certain what the license plate is, instead of drowning the server with 30 license plate predictions per second.

As of May 2022 there has only been maintenance commits since 2018, so

[1]As I was finishing up this thesis I noticed that OpenALPR has now been bought or re-branded as Rekor, and the selection of products and services they offer seem to be different, but likely close enough to the ones mentioned in this thesis.

it seems quite safe to assume that they are not going to be improving on the open source version anymore. When installing the program according to the instructions (at least on Ubuntu 18.04), and running the test they provide, the program crashes due to a segmentation fault. This issue can be worked around by specifying the country code but that their own tests fail does not exactly instill confidence.

The open source version of OpenALPR can be improved upon by training one's own models, but this will require providing one's own labeled images. Both the character recognition and plate detection models can be retrained. This does, however, entail large amounts of manual work for something that may or may not improve one's results. Of course, the project is open source so changing it to your own liking is free to do, as long as one abides by the current license.

OpenALPR Watchman Agent The OpenALPR Watchman Agent (paid service) is configurable to take input from multiple IP cameras and turns them into what they call plate groups, that is, "data objects" containing readings of similar plates, based on their Levenshtein distance [54], so that "ABC123" and "ABC128" would be placed in the same group if they were spotted close enough in time. These plate groups essentially provide information about how long the vehicle was visible and they provide some extra confidence by determining which license plate has appeared most frequently during a specific interval of time.

On-Premises SDK The On-Premises SDK (Software Development Kit) is available for C/C++ and can be configured in much the same way as the Watchman Agent, but one will need to perform the plate grouping oneself. This On-Premises SDK differs from the open source version in that it is using the better models, so the performance should be better.

3.3 Methods for Object Detection

Object detection can be done through two different primary approaches: deep learning-based methods and conventional computer vision-based methods. The latter potentially combined with more basic machine learning such as simple classifying multi-layer perceptrons.

Deep learning means neural network models that learn to identify features and patterns, automatically, as opposed to being given a set of hand-crafted features and simply identifying those [15]. Features learned by a deep neural network are often better than hand-crafted features [15]. A popular deep-learning approach in modern object detection is to only have one model for generating bounding boxes around the objects that are automatically classified, all in one step [27, 33]. Such a model is called a single stage detector (SSD) which is the type of model this thesis mainly will be focusing on, with regard to deep learning.

The conventional computer vision methods rely on applying various image manipulation techniques in order to make the object(s) of interest stand out from the background, and then isolating the most visible objects that match some criteria for e.g. shape or size.

Depending on how one looks at this, either of these approaches might be the difficult way or the easy way. If one has large amounts of ready data and a powerful GPU, training a deep learning model to detect license plates and another to recognize characters might be the easier approach. If one does not, it might be easier to combine the more conventional computer vision methods of image manipulation with template matching, a classifying neural network or some existing optical character recognition library such as Tesseract OCR.

Lastly, the different methods have their own strengths and weaknesses. The deep learning methods will probably require more work to achieve good results, but when it works well it will most likely be better suited as a general solution than the conventional methods. Implicitly, the conventional methods are probably faster to deploy, but with less flexibility, and most likely would have to be tuned specifically for each install to zoom the camera in on the, exact, correct spot etc. One of the main differences between the deep learning methods versus the conventional methods is that it is nearly impossible to do vehicle detection using the conventional methods as it simply becomes too complex. License plate detection is feasible provided that the image is zoomed in closely enough such that the license plate is guaranteed to be the most prominent feature after the image manipulation steps.

3.3.1 Sliding Window Object Detection

A common approach to object detection is to perform a sliding window search over the image. A window is slid across the image and each sub-image is classified using some preferred classification method. Classification methods include, but are not limited to, template matching and classifying neural networks.

The time it takes to detect objects using the sliding window approach depends on how many total windows the classifier ends up processing. No matter how the sliding window algorithm is approached, it inherently relies on running one or more classifiers multiple time for each image, which can turn out quite costly. This is later shown with the character classifier in section 5.3.3 and template matching algorithm in section 5.3.2.

3.3.2 ML-based Object Detection

Machine learning-based object detection comes in many flavours but, generally speaking, they all require training some algorithm or model on a set of images combined with labels corresponding to the positions and classes of the objects present in the image. Such a set of images and labels is called a *data set*. Training an ML-based object detector is done by iteratively processing the data set, letting the object detector predict what the correct labels are, and adjusting its parameters based on its predictions. The goal with training is to improve the predictions of the detector. This process is more or less the same no matter what type of object one is attempting to detect in an image.

There are many types of machine learning-based object detectors. For example, the Viola-Jones (discussed below) cascading classifier is indeed such a detector. However, as of 2021, the most accurate ML-based object detectors are all deep learning-based.

Modern deep learning-based object detection models are usually made up of two parts, a "backbone" for detecting features and a "head" for predicting the class and bounding box of the objects detected in the backbone. Furthermore, there are one-stage object detectors [27, 28, 29, 30, 31, 33, 34, 55, 56] and two-stage object detectors [35, 57, 58, 59, 60]. The two-stage object detectors separately generate region proposals and then, in the second step, classify those areas; whereas the one-stage detectors produce both in the same stage (in one neural network). Generally speaking, the one-stage detectors tend to be faster while the two-stage detectors tend to be more accurate.

The Viola-Jones Cascading Classifier (sometimes referred to as a HAAR cascade classifier), is an object detection algorithm as described by P. Viola and M. Jones, in 2001 [61].

The Viola-Jones cascading classifier uses a sliding window approach to move across the image and compares that sub-image with all the classifiers, starting with the weakest classifier and ending with the strongest classifier. The idea being that it is best to quickly discard areas that have a low probability of being the sought object. The window sizes to be tested are determined beforehand (through training or hand-picking), but they are set in stone, so there needs to be a window size that matches the size of the sought object closely enough to pass through the whole classifier chain successfully. The features used in each classifier are selected through training as described in their paper [61].

The main drawback of HAAR cascade classifiers is that they are prone to producing false positives [62, 63, 64]. They can be very fast, however. Even back when their paper was released, this algorithm was able to detect faces at 15 frames per second for images of size 384×288; and this was on a single core 700 MHz processor. Admittedly, this is very low resolution and HAAR cascade classifiers do not scale well, with regard to resolution; but, in cases that allow a lower resolution this can be a fast method of doing object detection.

The algorithm is based on first computing what they refer to as an "integral image" (see Figure 23), and then comparing that image with a set of HAAR-like features (see Figure 24). In a cascading manner, any suggestion that does not pass the next classifier is rejected.

The integral image (see Figure 23) has one value for each pixel in the original image, and this value is the sum of all the pixel values above and to the left of the pixel plus the value of the current pixel. Because the next sum is based on the previously calculated sums, this is relatively fast to calculate. On larger images

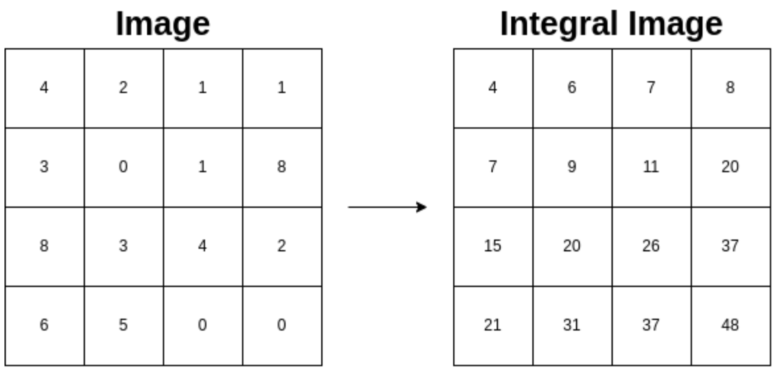


Figure 23: *The integral image calculated from the pixel values in an image.*

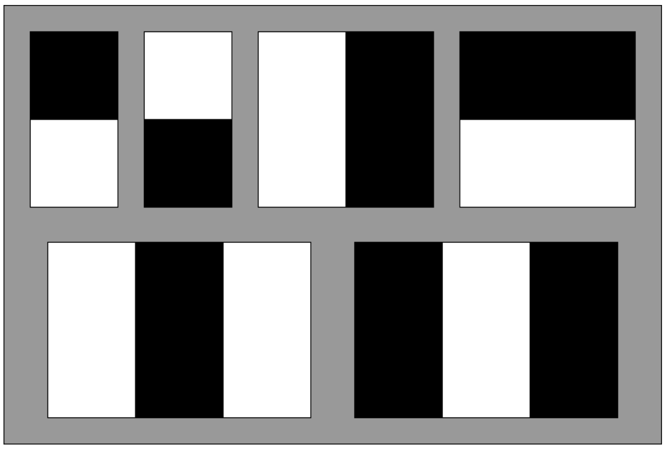


Figure 24: *Some examples of HAAR-like features.*

this will still take a very long time (e.g. a python for-loop of size 1920×1080 takes around 78ms on the author's computer). The beautiful thing with an integral image is that one can pick any rectangle in the image and quickly calculate the total pixel intensity for that whole rectangle at a constant time-cost, regardless of the actual size of the rectangle.

The features can be compared with the integral image by doing comparisons as described in Figure 25; one can, e.g., compare the total brightness between area B and D by simply calculating the sum of pixel intensity of both areas in just a few steps. Thus, by accessing four values in the integral image matrix one can calculate the sum of any area; and in other words, the complexity of any comparison is constant, regardless of the size of the area.

Features are combined to form classifiers, which denote some part of the face (in the case of the Viola-Jones paper), e.g. an eye, a nose, or a mouth. A

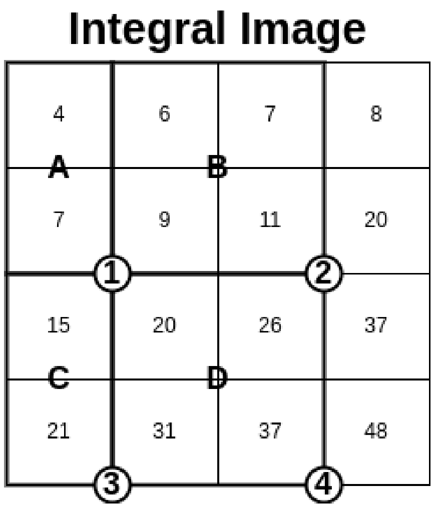


Figure 25: The value at location 1 is A, the value of location 2 is A + B, the value at location 3 is A + C and the value at location 4 is A + B + C + D. The sum of area D is 4 + 1 - (2+3) (the values at each point, not the integer values).

classifier is successful if more features are detected than the threshold value for the classifier. A weak classifier has fewer features and a lower threshold than a strong classifier. By combining multiple classifiers like a chain, one can move on to the next window if the first classifier in the chain fails (e.g. a face requires one nose, two eyes and one mouth). Viola & Jones quoted that only ten features, on average, were compared per window in their tests, meaning that negatives were quickly rejected.

3.3.3 Character Segmentation

Character segmentation is the process of isolating each character in an image into their own images. Usually using quite simple and intuitive algorithms.

One method of doing character segmentation is by calculating the intensity of each row and/or column of the image. Essentially, this means scanning through each pixel in the image (or a scaled down version of the image for faster processing) once and summing the values by row and/or column so that one receives a list of the pixel intensity for each row and/or column. When this is done one can use an intensity threshold to create boxes around each character. From left to right for column intensity: the location at which the intensity value goes above the threshold is the left edge of the box and the location at which the intensity value goes below the threshold is the right edge of the box. This works best if the image's perspective has been corrected. If the image has not been perspective corrected there might be some characters that overlap on the X-axis. Overlapping characters is a difficult problem to solve, but one approach is to combine the pixel intensity sums with the pixel projection as discussed in e.g. [65].

3.3.4 Perspective Transformation and Skew Correction

Perspective transformation and skew correction are two different methods of transforming an image to prepare it for OCR. They are both important parts of the ALPR pipeline, because regardless the approach, it will be easier to correctly identify characters if the shapes of the characters are consistent.

Perspective transformation is the process of identifying the perspective of the object in relation to the plane of the picture and transforming the image so that the object is facing in the desired direction. Perspective transform is a three-dimensional operation, whereas skew correction is two-dimensional.

OpenCV has built-in functions for performing perspective transformation. For perspective transformation one needs to know the original position of the four corners (top left, top right, bottom left, bottom right) and the new positions of these corners, which one would place squarely in the image. In order to locate the corners OpenCV also provides a function, "cornerHarris", for detecting corners in an image [66], this is a very cheap operation (0.2ms on an image of resolution 150×76) and using these corners we can try to determine the actual corners of the objects in question.

Another approach to this, is if one knows what the angle of the camera is going to be and the camera is placed above the vehicle path. Characters in such an image would be compressed along the vertical axis so by dividing the characters into horizontal sections and inserting a sort of extrapolated middle, one thereby corrects the aspect ratio of the characters detected, making them look less compressed. This approach was taken in one of the earliest attempts at ALPR (or Automatic Vehicle Identification, AVI as it is referred to in the paper) by [10]. This is by no means a perfect method because it will not enlarge the characters uniformly, but it is at least an improvement over a distorted image, and it is an alternative that could more easily be utilized on embedded systems because of its simplicity.

Skew Correction is the process of rotating a text that is facing straight towards oneself, but is rotated along its central point, so that the text is right-side up. It is easy to assume that this is unnecessary because for most scenarios involving license plates the camera will be right-side up, and the plates will always be right-side up, but as can be seen in Figure 63, even minimal skew can have a profound effect on less flexible text recognition algorithms.

A more brute-force approach for determining the correct rotation angle is to calculate histograms of the sums of the pixel intensity values in each column of an image and comparing the histograms for each rotation degree respectively. The premise being, that histograms with a larger difference between the peaks and valleys have a better rotation. In Figure 26, a license plate was correctly rotated with this method. The problem with this method is that the more rotation one allows for the longer the processing will take as it has to rotate the image and calculate the histogram additional times.

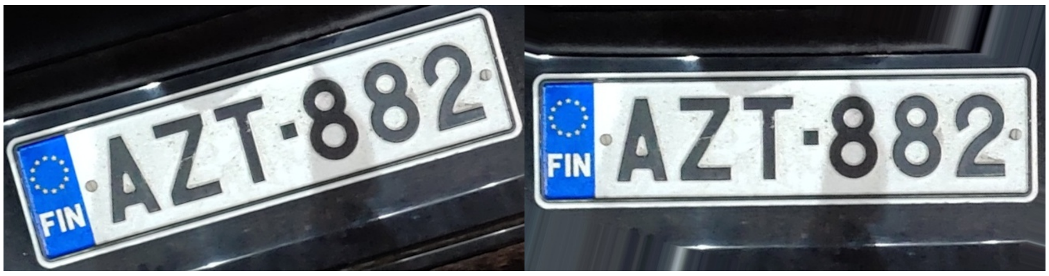


Figure 26: Simple skew correction of license plate. Processing time 16ms. Testing angles -15 to +15.

3.3.5 Template Matching

Template matching is a method for locating a template image in a larger image [67]. This can be useful for detecting characters in images, provided that the image is clean enough. The template image slides across the target image and the difference is calculated at each point. See the formula below for an example of such a difference calculation.

$$R(x, y) = \sum_{x',y'}(T'(x', y') \cdot I'(x + x', y + y'))$$

Where R(x,y) is the total difference for the point (x,y) in the image, T is the template image, I is the target image, x' and y' are the positions for x and y in the template image ($x' = 0...w - 1, y' = 0...h - 1$). By doing this sliding comparison one obtains a map of the difference where the highest value corresponds to the position at which the image best matches the template. This means that the coordinate one is comparing is not the center of the template but the corner that is the closest to the origin (x=0, y=0). The size of the resulting image is: $(W - w + 1) \times (H - h + 1)$, where W,H are the dimensions of the image and w,h are the dimensions of the template. This means that an image and template of the same size results in a 1×1 ($\times 1$, if gray scale color) matrix.

One can then create a template for each character one wants to detect, do this comparison for each template over the license plate image and then select the template which matches the best.

3.3.6 Neural Network Classifier

A neural network classifier is a feed-forward neural network that has been trained to recognize certain patterns when given data of a specific shape. For an image this means that there would be one input neuron for each pixel in the image and one output neuron for each possible class. The neural network classifier can be used to classify images of characters that have been cropped to bounds of the characters.

3.4 Metrics for Object Detectors

To monitor the progress when training an object detector or classifier it is common to use the metrics "Precision" and "Recall". In order to understand precision and recall we must first discuss the possible outcomes of a classification. Imagine a classifier that simply gives one single positive or negative response. If the instance is classified as positive and the instance is positive, that is a true positive. If the instance was actually negative, that is a false positive. If the instance is classified as negative and the instance is negative, that is a true negative. If the instance was classified as negative, but it was positive, that is a false negative. Figure 27 illustrates these relationships using confusion matrices. Using this knowledge, the precision and recall metrics can be calculated. [15]

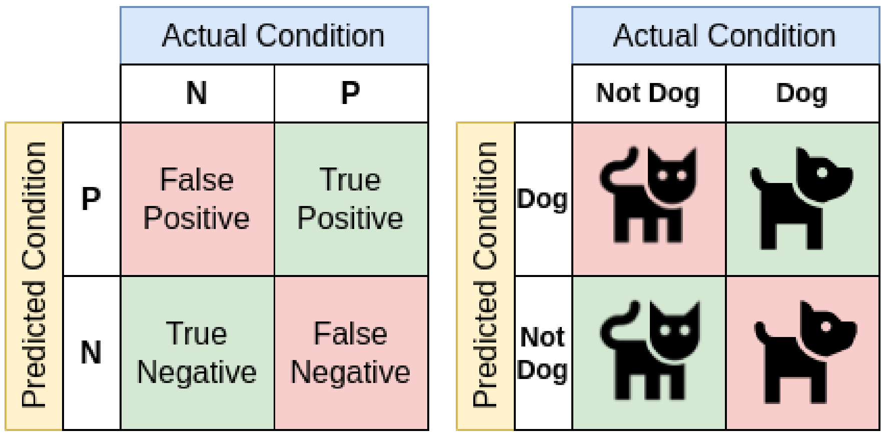


Figure 27: The relationship between true/false positive/negative illustrated with two confusion matrices.

Precision denotes the number of correct predictions out of the total number of predictions in an example, and recall denotes the number of correct predictions out of the total number of positives in an example; see Figure 28. Naturally, it is best that both values are as high as possible, as a precision of 1 and a recall of 1 would mean that there are no false positives and all objects were detected. If the recall is 1 and the precision is 0.5, that means we detected every object successfully, but we also got many false positives; which likely means that our detector has become very sensitive and is now reporting way too many positives. However, if we have a precision of 1 and a recall of 0.5, that means we produced no false positives, but we only detected half of the objects, meaning that the detector has become too "picky", possibly because of too little variation in the data or by overfitting. [15]

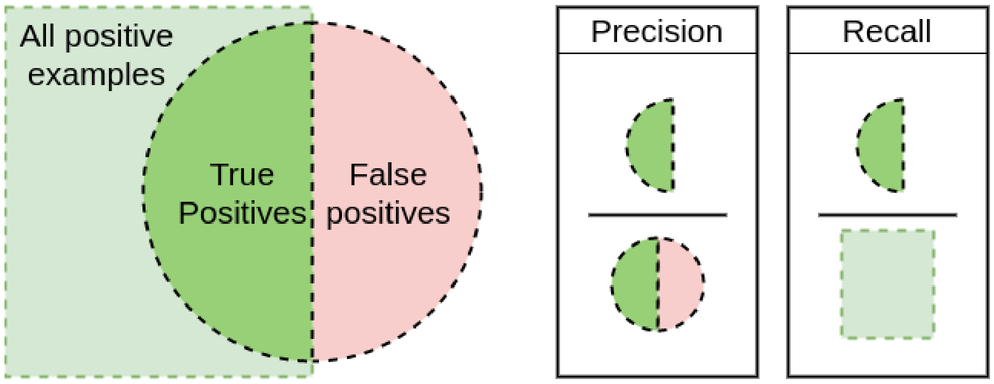


Figure 28: The precision and recall metrics.

Looking at the precision and recall metrics from the ALPR perspective, it is not too bad if the recall is much higher than the precision for our vehicle detector, because that just means that we should not actually be missing any vehicles. For the license plate detector it is more important that the recall and precision are at similar (and high) levels and for the character detector it is the even more important. If our character detector has a low recall it means that it will not detect all the characters and if the precision is low it means that it will produce many false positives. False positives, in this case, are better than false negatives, as the false positives most likely would show up as noise and be something we can filter out based on how often each character appears (assuming that the same license plate is captured multiple times). A false negative, however, is lost data; thus, when training, it is likely a good idea to utilize a vehicle/license plate detector that is quite sensitive. Whereas the character detector really should have as high precision and recall as possible.

Object detectors produce bounding boxes, i.e. values that represent rectangles defining the bounds of the detected object. To determine the accuracy of a bounding box, one must compare the "ground truth" bounding box (the bounding box that has been defined by the person conducting the test) and the predicted bounding box. This is often done using "Intersection over Union", IoU. This IoU is calculated by dividing the area of the *intersection* of the bounding boxes by the area of the *union* of the bounding boxes; see Figure 29. In research papers, the standard metric for accuracy of object detection algorithms is Average Precision (AP) or mean Average Precision (mAP). Average precision is the average (or mean) precision (for different thresholds of IoU) between the predicted boxes and the ground truth. One alternative (which is mentioned later in this thesis) to IoU is Generalized Intersection over Union, GIoU. According to the original paper on GIoU [68], object detection models perform better when trained using GIoU rather than IoU.

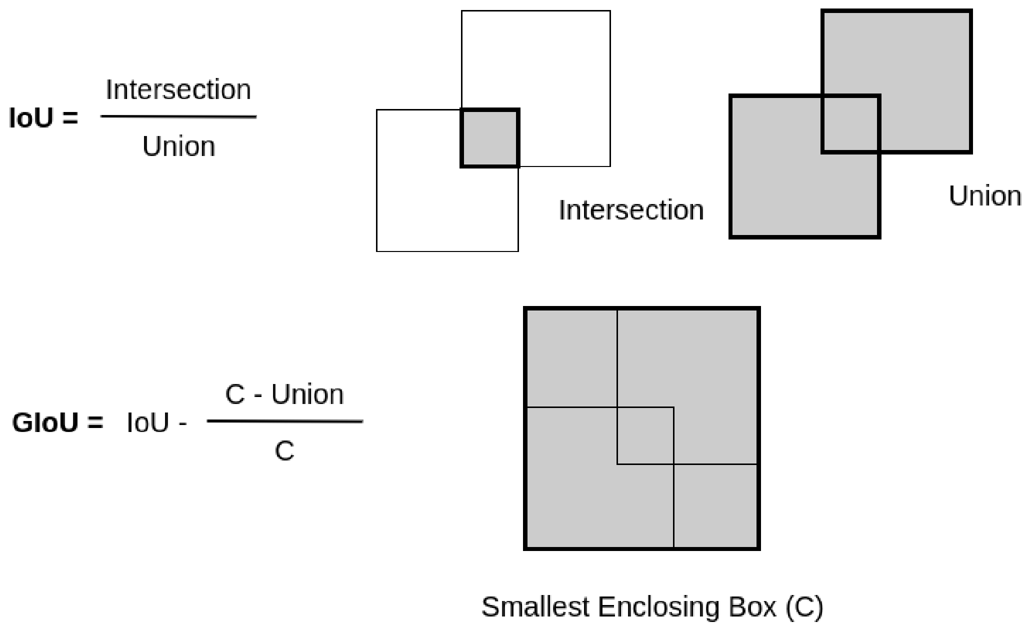

IoU = Intersection / Union
Intersection
Union
GIoU = IoU - (C - Union) / C
Smallest Enclosing Box (C)

Figure 29: How IoU and GIoU are calculated.

When comparing object detection models the authors often test their models on some commonly used data sets such as the PASCAL VOC 2007 [69] or COCO [70] and calculate the AP to measure their performances. These data sets were created for the purpose of advancing the field of machine learning. The AP is not necessarily calculated the same way by different data set providers (the AP's for PASCAL VOC 2007 and COCO are calculated differently) but for object detection it is usually calculated by looking at the number of true/false positive detections and false negatives (determined by the IoU between the prediction and the ground truth) at different thresholds for IoU. APs with different thresholds are used as different metrics, e.g. YOLOv3 refers to AP_{50} which means the IoU threshold for a true positive is 0.5 and AP_{75} which means the threshold is 0.75. Exactly how the mAP is calculated is not, however, very relevant here as we are only interested in the direct comparison; for more information refer to the pages of the specific data sets.

3.5 You Only Look Once (YOLO)

You Only Look Once, YOLO, is a deep learning-based object detection model that was developed with the idea of putting the whole object detection pipeline in a single neural network in mind. This is contrary to the norm at the time, which was to split the algorithm up into several steps. YOLO is one of the most popular modern Single Stage Detectors (SSD, sometimes "one-stage detector"). It was developed by Redmond et al. [27].

3.5.1 YOLOv1

YOLOv1 is the first of three versions published by the original authors. The first thing one needs to know to understand YOLO is that it is a model that is divided into a grid of $S \times S$ cells. Each cell in turn, has B bounding box predictors and each of those predict one bounding box, for a total of five values per bounding box. Finally, each cell also produces C class predictions.

The bounding boxes consist of five prediction values between 0 and 1, the prediction confidence, the center x coordinate, the center y coordinate and the width and height of the bounding box. The bounding box predictions are treated as regressions instead of classifications. The predicted width and height of the bounding box correspond to the relative width in comparison to the image. The predicted center coordinate is relative to the bounds of the cell.

Since YOLO can produce multiple detections for the same object, non-max suppression is used. This simply means that the detections are ordered by confidence and added to a list one by one; unless the detection has an IoU larger than 0.5 with another detection, in which case they are discarded (only one of the overlapping detections is kept).

YOLO works by first performing a set of convolutions and pooling, sequentially detecting more and more complex features. To make sense of the features, a perceptron layer is fully connected to the last convolution layer, which in turn is fully connected to the, three-dimensional, output layer of size $S \times S \times (B \cdot 5 + C)$. In the original YOLO paper the parameters were defined as $S = 7$, $B = 2$ and $C = 20$, which comes out to be $7 \times 7 \times (5 \cdot 2 + 20) = 7 \times 7 \times 30$.

The YOLOv1 loss function is defined by the formula below.

$$
\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

In the summations, S is the number of cells and B is the number of bounding boxes predicted per cell, as described in previous paragraphs. The i is the index of the cell and j is assigned based on which predictor's prediction has the highest IoU with the ground truth bounding box. This means that the predictor that is best at detecting one object will become better at detecting that object, and it means that the individual predictors will become good at predicting objects of certain sizes, aspect ratios and classes.

The x and y parameters are the center coordinate values of the given training label and $\hat{x_i}$, $\hat{y_i}$ are the predicted center coordinates for the predicted bounding box. The w_i and h_i parameters are the width and height of the given training label and $\hat{w}$, $\hat{h}$ are the squared widths and heights of the predicted bounding box. They are squared because a small deviation in a large box should be punished less than a small deviation in a small box.

C_i and $\hat{C}_i$ are the expected confidence (should be 1 if an object is present), and the predicted confidence (objectness).

Finally, the $\mathbb{1}_{i}^{obj}$ denotes whether there is an object in cell i and $\mathbb{1}_{ij}^{obj}$ is equal to 1 (thus enabling the loss) for the predictor j that is responsible for the bounding box prediction. This means that the loss function only penalizes the classification error if there is an object present, and it only penalizes the location, dimension and "objectness" (object presence confidence) errors for the bounding box responsible for the prediction. The λ_{coord} and λ_{noobj} parameters are used to increase the loss for specific parts of the loss function. In the original YOLO paper they used $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$.

YOLOv1 has 24 convolutional layers followed by 2 fully connected layers, consistently reducing the x and y dimensions and increasing the number of channels, until reaching a volume of size $7 \times 7 \times 1024$. Figure 30 shows the general idea without showing all the convolutional layers. Some convolutional layers use filters with 1×1 kernels as described in Section 2.1.5. The $7 \times 7 \times 1024$ parameters of the final channels are fully connected to a perceptron layer with 4096 neurons, which in turn is connected to the output layer. The output layer uses a linear activation function and all the other layers use a leaky rectified linear activation function. When training, they used a dropout layer with a rate of 0.5 between the last convolutional layer and the first perceptron layer (in the paper they said it is after the first perceptron layer but according to the code available on GitHub it comes before [71]).

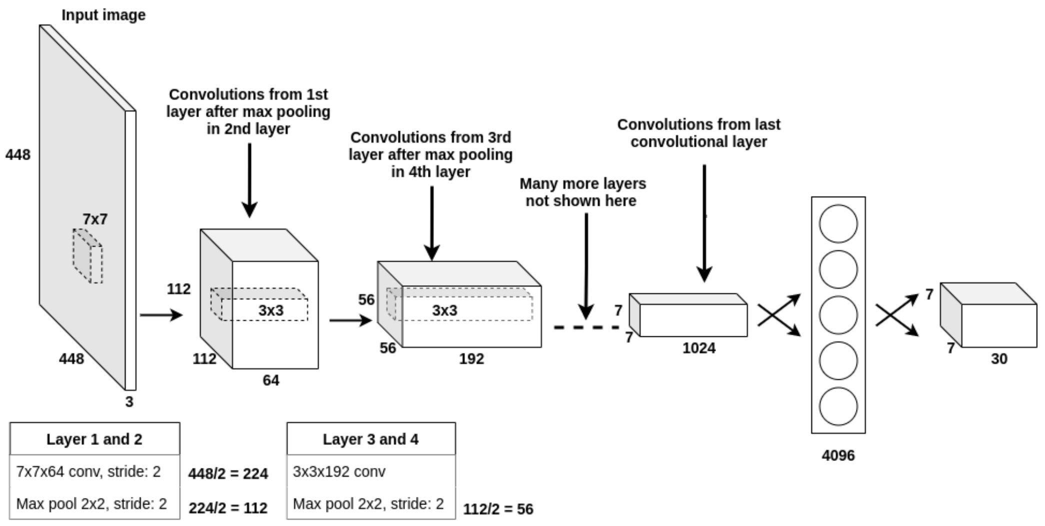


Figure 30: Compressed image of the YOLOv1 model structure.

When YOLOv1 was released it had the best speed per mAP (PASCAL VOC 2007) by far, based on the tests made in the paper [27]. The YOLO network ran at 45 FPS with an accuracy of 63.4 mAP while the second-fastest object detector, the 100Hz DPM [72] (running at 100 FPS), had an accuracy of 16.0 mAP. The most accurate network at the time was MR_CNN_MORE_DATA, at 73.9 mAP, but its frame rate was not listed. Faster R-CNN VGG-16 [59], however, performed at 73.2 mAP, running at 18 FPS. This new level of speed at such a high accuracy is what made YOLO popular.

3.5.2 YOLOv2

The second iteration, YOLOv2, runs faster and with a higher accuracy (PASCAL VOC 2007) at 78.6 mAP, than YOLOv1 at 63.4 mAP. These improvements come from a slew of changes, but the most notable ones are perhaps, training the classifier part of the network in higher resolution and increasing the final channel dimensions to $13 \times 13(\times 1024)$. They also replaced the fully connected perceptron classifier with a convolutional approach where a kernel filter is responsible for the prediction of each variable, concatenate convolutions like in section 2.2.3, employ *anchor* boxes and random image rescaling during training. Normally, anchor boxes refer to fixed size boxes, but in YOLOv2 they use a modified version. In the modified version, the predicted bounding box values modify the anchor values instead of only predicting a center point and using the anchors as the bounding boxes. In the YOLOv2 paper they refer to these as dimension clusters, but for simplicity we will refer to them as anchors as well. In YOLOv2 the output also looks slightly different: each predicted box now has its own class predictions, whereas in YOLOv1 there was only one set of class predictions per cell. The concatenation was done from a previous $26 \times 26 \times 512$ layer, which had to be resized to $13 \times 13(\times 2048)$ for it to be possible, for a total of 3072 channels; see Figure 31. This helps the network detect smaller features.

The dimensions of the anchors are decided before training, using K-means clustering [28]. The anchor-based bounding boxes in YOLOv2 are constrained to have their center fall within the bounds of the predicting cell, unlike in YOLOv1 where the center point could fall anywhere in the image, this makes YOLOv2 much easier to train and the bounding box width and height is restricted by the sizes of the available anchors. An anchor is simply a bounding box of predecided dimensions (based on the training data set). The predicted bounding box values describe the translation of the bounding box position relative to the cell, a scale value for the width and a scale value for the height to be multiplied by the dimensions of the anchor (see the formula following this paragraph). Like in YOLOv1, the confidence (IoU during training). Each cell also predicts 5 boxes instead of 2, in YOLOv2.

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w \cdot e^{t_w} \\
b_h &= p_h \cdot e^{t_h} \\
Pr(object) &\times IoU(b, object) = \sigma(t_o)
\end{aligned}
$$

Where b refers to the predicted bounding box, (c_x, c_y) is the top-left corner coordinates of the cell, t_x is the predicted value for the x coordinate modifier, t_y is the predicted value for the y coordinate modifier, t_w is the predicted value for the width modifier, t_h: is the predicted value for the height modifier, t_o is the predicted objectness, $Pr(object)$ is 1 when there is an object in the cell, $IoU(b, object)$ is the IoU between the predicted box and the ground truth, σ is the activation function and $Pr(object) \times IoU(b, object)$ is the objectness during training.

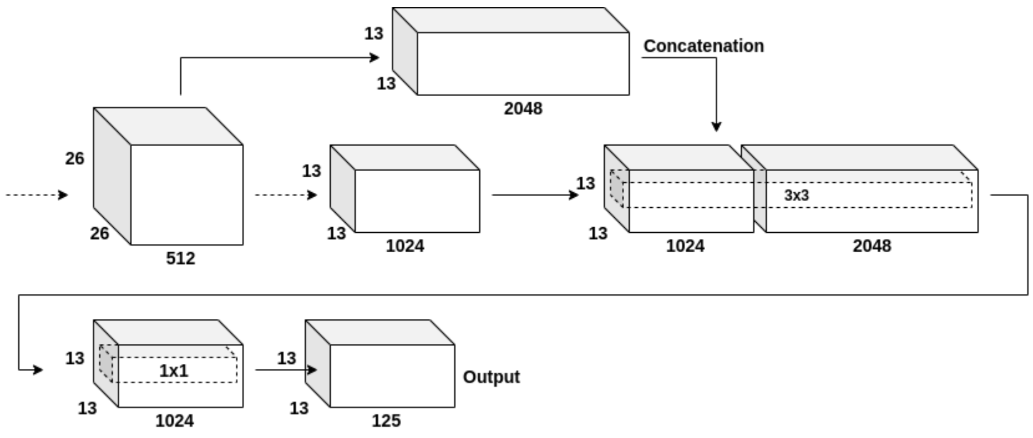


Figure 31: Simplified image of the YOLOv2 model structure.

3.5.3 YOLOv3

In the third version, YOLOv3, there has been significant changes architecture-wise, but not so much training-wise. To quote the authors of the paper *"We mostly took good ideas from other people. We also trained a new classifier network that's better than the other ones."* [29]. Foremost, the feature extraction part of the network has increased in size from 19 to 53 layers. In addition to this, YOLOv3 predicts boxes on three scales using an FPN (section 2.2.5) style approach by concatenating feature maps from previous layers with later ones. YOLOv3 also comes with residual blocks as described in section 2.2.2. Each cell predicts $B \times (5 + C)$ values, like YOLOv2. In the tests made for the YOLOv3 paper $B = 3$ and $C = 80$.

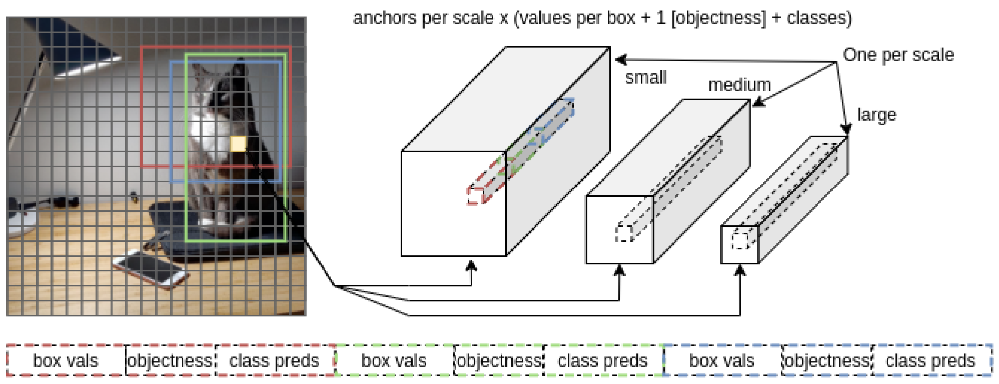


Figure 32: The YOLOv3 output. The yellow grid cell produces three bounding box predictions on three different scales.

As a final note about YOLOv1-3, the authors first trained the feature detector parts of the networks to classify, and then trained the full network, object detection layers and all. The training methods differ between YOLO versions but for most end-users this is not important as one can start with the weights from the pre-trained models and then train with one's own data.

3.5.4 YOLOv5

YOLOv5 was not developed by the same authors as the three original YOLO versions, but by Glenn Jocher, and there is no research paper describing it. However, it builds upon the same model architecture used in YOLOv3 for the backbone and the entire project is available on GitHub under the GPL-3.0 license [31]. YOLOv4 [30] was not developed by the same authors as YOLOv1-3 either, but it is forked from the original project (i.e. it uses the same framework) on GitHub and there is a research paper backing it up, so that is something to consider. There has been some confusion and debate about which one of YOLOv4 and YOLOv5 is better/faster/more accurate but there are not many papers available on the topic and since the author of YOLOv5 has not published a paper it makes things slightly more difficult. Regardless, YOLOv5 is built using PyTorch instead of Darknet, which all the other YOLO versions are built with. Because the author of this thesis has a stronger familiarity with python and PyTorch, YOLOv5 was chosen. The main drawback of using YOLOv5 as the example in this thesis is that there is no official research paper about it. Thus, the implementation must be interpreted by reading the code, which could potentially change at any time.

Towards the end of the process of writing this thesis, another paper improving on the YOLO architecture was published [32]. They also compared the differences between YOLOv4 and YOLOv5 and showed that YOLOv5 is indeed faster and more accurate.

There are several architectures of YOLOv5 available in the GitHub repository, some of which are surely employing more modern methods than the original models posted there. Nonetheless, one of the original models "yolov5s" was used in this thesis. Figure 33 shows the architecture of the "yolov5s" model configuration (or at least what it looked like when it was downloaded for this thesis).

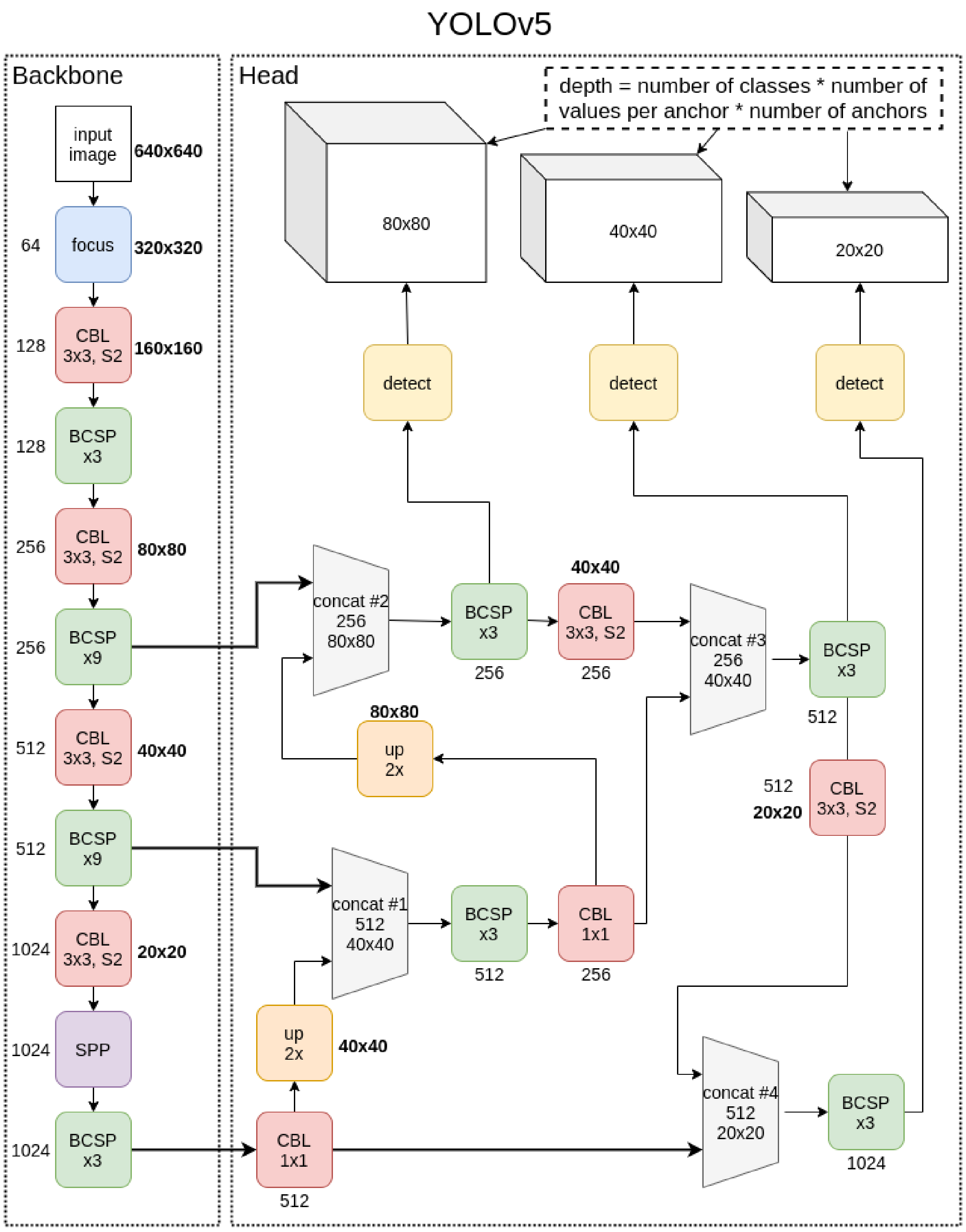
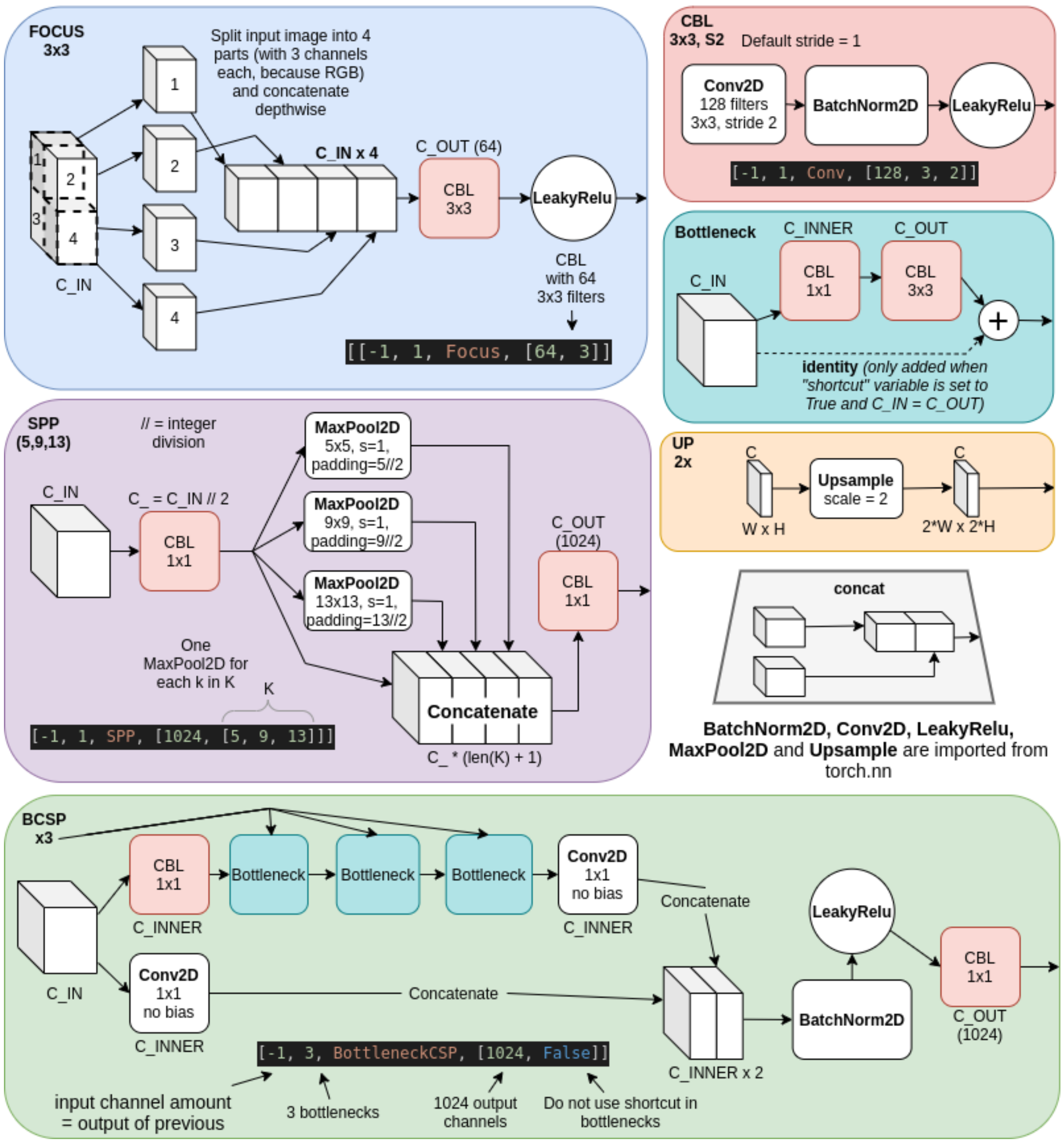
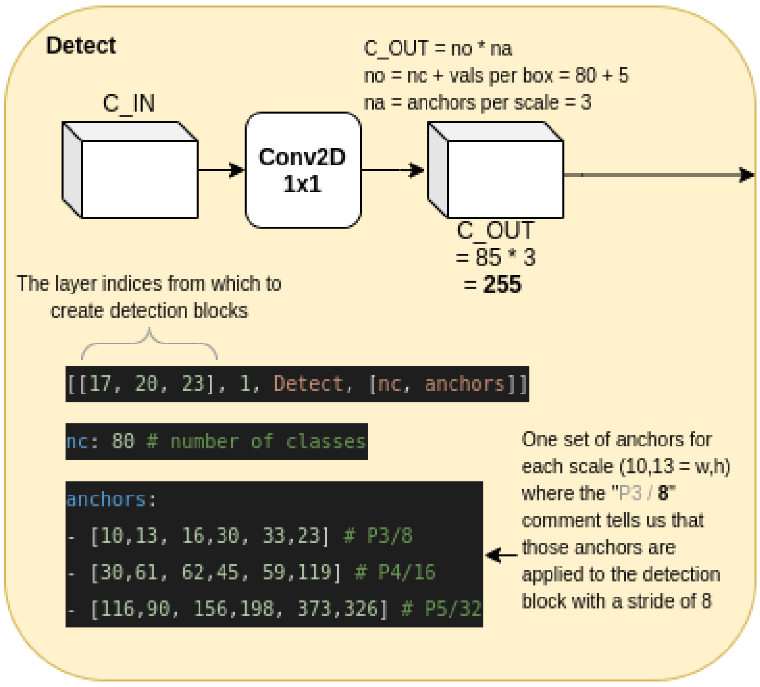


Figure 33: The YOLOv5s model architecture. The blocks are explained in Figure 34.



```
[[-1, 1, Focus, [64, 3]]
[-1, 1, Conv, [128, 3, 2]]
[-1, 1, SPP, [1024, [5, 9, 13]]]
[-1, 3, BottleneckCSP, [1024, False]]
```

Figure 34: The explanations of the blocks used in Figure 33. The "detect" block is shown in Figure 35.


Detect
C_IN
Conv2D 1x1
C_OUT = no * na
no = nc + vals per box = 80 + 5
na = anchors per scale = 3
C_OUT = 85 * 3 = 255
The layer indices from which to create detection blocks
One set of anchors for each scale (10,13 = w,h) where the "P3 / 8" comment tells us that those anchors are applied to the detection block with a stride of 8

```
[[17, 20, 23], 1, Detect, [nc, anchors]]
```

```
nc: 80 # number of classes
```

```
anchors:
- [10,13, 16,30, 33,23] # P3/8
- [30,61, 62,45, 59,119] # P4/16
- [116,90, 156,198, 373,326] # P5/32
```

Figure 35: Figure 33 shows three detect blocks, each in essence only consisting of a 1×1 *convolutional layer. This figure shows how the parameters defined for the models are connected graphically.*

As we can see in Figure 33, the YOLOv5s model was implemented similarly to YOLOv3 using a FPN style architecture for multiscale detection.

The first difference is the introduction of a "Focus" block, often referred to as a "space-to-depth layer" [73, 74]. The purpose of this block is to quickly reduce the resolution of the image by dividing the image and concatenating the parts depth-wise. This means that the entire resolution of the image is still available in the feature maps, but with one fourth the area and four times the depth. After reducing the resolution the volume is passed to a convolutional layer with 64 3×3 kernel filters, and then that output is passed through a leaky ReLU.

YOLOv5 also employs CSP, but not quite implemented the way suggested in the CSPNet paper [45]. In the CSPNet paper they suggest splitting the input feature maps, passing half into the dense block and concatenating the other half to the output of the block in the transition layer; see Figure 19. In the YOLOv5 implementation of CSP the input feature maps are not split up, instead, all feature maps are passed through the dense block for the first path and through a single 1×1 convolutional layer for the second path before the outputs of each path are concatenated. Naturally, this means more computation and *potentially* higher performance (my tests indicate that it does perform better). The dense block first reduces the depth to C_{INNER} (it is variable but in this model it is always half that of C_{IN}), after which that volume is passed through a chain of bottleneck blocks, similar to the bottleneck blocks from ResNet [36]; see Figure 17. The number of bottleneck blocks is determined by the second parameter in the configuration array for the layer, as shown in Figure 34.

The SPP-layer is also not quite the same as the one suggested in the original paper [46], as the output is not two-dimensional, but it follows the same principles of pooling at different scales, which essentially means that the different feature maps of the output volume represent features/objects at different scales.

The output is of the same shape as the output in YOLOv3 (compare figures 32 and 35), but the bounding box variables are calculated differently. After performing the 1×1 convolution, the output volumes are processed one at a time. First, they are split into 4-dimensional volumes, or hyper-volumes, such that each predicted bounding box has their own coordinate in the new dimension (it might help to think of this as a list of volumes). The values are all passed through a sigmoid function and output a volume where each cell corresponds to the one detection this cell produced, containing all parameters depth-wise, like in Figure 32. Then the bounding box values are calculated from the predicted values t_x, t_y, t_w and t_h using the following formulas.

$$b_x = 2\sigma(t_x) - 0.5 + c_x$$

$$b_y = 2\sigma(t_y) - 0.5 + c_y$$

$$b_w = 2\sigma(t_w)^2 \cdot p_w$$

$$b_h = 2\sigma(t_h)^2 \cdot p_h$$

Jocher identified a problem with regard to how the width and height values were calculated in YOLOv2 and YOLOv3 (and YOLOv4). Because the width and height are calculated by $b_w = p_w \cdot e^{t_w}$ there is no upper limit, $b_w(t_w) \in [0, \infty[$. To stabilize the training YOLOv5 also passes the width and height predictions through the sigmoid function.

Finally, the loss function is also different in YOLOv5. YOLOv5 uses GIoU instead of IoU (see Figure 29), which is combined with the "binary cross-entropy"-loss (BCE) of the class predictions and the BCE-loss of the objectness. The binary cross-entropy loss is defined as follows [75].

$$BCE = -\sum_{i}^{C} t_i log(y_i) + (1 - t_i) log(1 - y_i)$$

Where C is the number of classes, i is the class in question, y_i is the predicted value and t_i is the ground truth value. The loss of each class is calculated, summed up and inverted. To be precise, YOLOv5 uses the BCEWithLogitsLoss function from PyTorch, which applies the sigmoid function to the predicted values before calculating the logarithm [75].

$$BCEWithLogitsLoss = -\sum_{i}^{C} t_i log(\sigma(y_i)) + (1 - t_i) log(1 - \sigma(y_i))$$

3.5.5 Training YOLOv5

When training an object detection model, such as YOLO, it is important to be as consistent as possible when preparing the bounding box labels for the training data set. This is because the loss function of object detectors need to take the size and location of the box into account in order to improve. It not only takes into account whether the predicted class is correct but also how well the predicted bounding box matches the ground truth bounding box. Recall how the YOLO loss function (and probably most other object detection loss functions) tries to minimize the bounding box location error, the bounding box dimension error and the classification error. If the model in reality produces the best possible bounding box, but the label it has been given is not perfect, it will be penalized even though the prediction might have been closer to the truth than the label. Thus, consistency is important.

YOLOv5 comes with a slew of configuration options for training. These hyperparameters, as Jocher refers to them, are configurable, manually or through the project's built in hyperparameter evolution. This means that by simply running the training program as if one were to actually train a model, but with the

"–evolve" parameter set, a better set of hyperparameters will be produced. Do note that those hyperparameters are not necessarily going to be very good for a different model and/or data set. Hyperparameters are parameters related to training but not the network itself. In other words, hyperparameters are not the weights of the network, but the parameters used in training, such as learning rate and multipliers used to amplify/reduce the effect of a specific part of the loss function, as well as training data-modification options. The hyperparameters will continue evolving until the program is stopped, after which one can use them to train new models.

Another very useful thing that comes "out of the box" with YOLOv5 is the generation of anchors for the data sets. New anchors are generated by assigning an integer value instead of the anchor lists to the "anchors" variable, in the model configuration file. This integer will represent the number of anchors to generate per detection layer (3 anchors and 3 detection layer scales by default means it generates 9 anchors). The anchors are printed to the console and need to be copied into the configuration file in order to be reused. Using anchors generated specifically for one's data set can improve performance tremendously as the pre-generated ones are for the COCO data set and do not necessarily suit one's own data.

During training, YOLOv5 will automatically modify the input data (can be disabled) so that the same example is never shown twice, despite using the same example image/label multiple times. The data can be modified by rotation, translation, flipping, altering the colors, "mosaic" augmentation or cutting out positive examples of images at random, etc. Mosaic augmentation is a simple data augmentation method, first introduced in the YOLOv4 paper [30], which combines parts from four images into one training image, arranged randomly.

3.6 Vehicle Detection

Vehicle detection can be useful when doing ALPR for a few different reasons. First, one can be more confident that a detected license plate is actually a real license plate (and not e.g. a traffic sign), if it is within the boundaries of a detected vehicle. This works the other way around too: if one detects a vehicle in the image one can be fairly certain that there is also going to be a license plate present within that section of the image. Second, one might be able to reduce processing time if it is faster to detect vehicles than license plates. Third, it might be useful to know the make and model of the vehicle as this means more information to compare with if trying to group the license plate recognitions. Adding the make and model of the vehicle to the recognition can help with the confidence of any license plate recognition, This is especially true in countries such as The United States, where it could even be necessary, as the same license plate number can exist as long as they are from different states. In the United States the license plates look different for each state, which can be used as a way to differentiate vehicles with the same license plate number. As a side note, the make and model could also be used to create a persona of a customer, e.g. "ABC-123; Honda; Sedan; vegan/vegetarian" and "ABC-123; Ford; Pickup truck; gluten-free".

3.6.1 Conventional Methods

Vehicle detection is not easy to implement using conventional methods. The main limitation of the conventional methods is that one specifically needs to define the process for detection oneself. One method that avoids complex conventional algorithms is motion detection.

Motion detection can be used as implicit vehicle detection. Motion detection can be done in many ways, but the perhaps most obvious approach is to simply calculate the difference between a new image and a reference image, binarize the difference by thresholding and perform contour detection on that. This works best when the new image and the reference image were taken relatively closely in time, because the longer the algorithm is running the larger the difference between the images will be (meaning noisier output). One way to fix this is to simply take a new reference image every so often, but if the image is taken at the wrong time (i.e. when something that is not actually part of the background is visible) all comparisons with that reference image will produce bad detections. A more resilient approach, presented by Stauffer and Grimson, is to model each pixel as a mixture of gaussians and updating them live [76]. This model is better at determining, e.g. if a swaying tree branch is part of the background or the foreground.

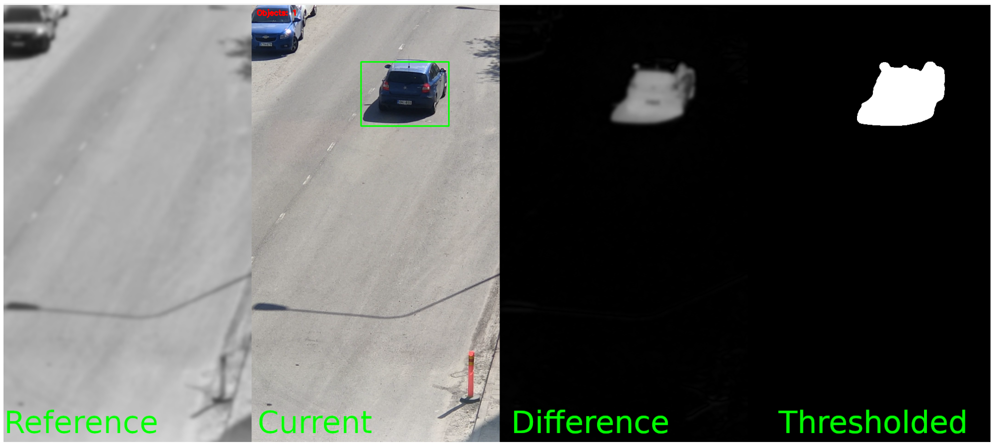


Figure 36: Naive motion detection using OpenCV. Short video clip.

3.6.2 ML-Based Methods

Specifically for vehicle detection one does not have to look far and wide to find a working solution. For example, YOLOv5 comes with pre-trained models on a data set that contains several types of vehicles, and is very easy to deploy. Meaning that one could deploy a powerful vehicle detection algorithm with minimal effort. These models are trained on other classes of objects as well, but those can be ignored if one does not wish to train one's own model. Generally speaking, ML-based vehicle detection will be much the same as for any other type of ML-based object detection, acquire a data set, design a model and start training.

There are also HAAR-cascades available, for free, for vehicle detection, so vehicle detection can quite easily deployed with the Viola Jones cascading classifier.

3.7 License Plate Detection

License plate detection is the process of finding license plates in an image, or "simpler" versions such as looking for rectangles and specific colour patterns.

3.7.1 Conventional Methods

By the heuristic/conventional method we mean that we specifically try to break down what it means to see a license plate. A license plate is usually quite oblong, rectangular, with a more or less uniform background color and a number of shapes (the characters) with a color contrasting the background color. A heuristics-based approach would be one that manipulates the image in ways that make the features that fit our definition of what a license plate looks like more prominent and the features that do not fit less prominent. One example of that could be to first detect contours in the image, detect rectangular shapes based on those contours and then using some criteria about the shapes and sizes determine which of those rectangles are likely to be license plates.

One approach published in 1991 by Kanayama et al. [10] is based on the premise that cars have mostly horizontal lines in the front, except for the license plate itself. Thus, they divided the image in horizontal sections and starting at the bottom right check for contours in each image section, and use all the sections between which the two sections the number of lines exceed a pre-determined threshold. The first vertical contour detected in this area will be determined to be the right edge of the plate.

3.7.2 ML-Based Methods

Just like with the vehicle plate detection, we can employ YOLOv5 or similar object detectors, and just like for the vehicles, there are license plate HAAR-cascades available to download for free. However, these might not necessarily be available for the correct region (e.g. I have not found any for Finnish license plates).

4 Methodology

The flow of most ALPR systems can be described by something like Figure 37, in which one can see that some skip the step of first detecting the vehicle, since it might not actually be relevant in all cases. Nevertheless, this will be the baseline for the processes we will try to implement and optimize through all the steps. Since the ALPR algorithm can be considered quite modular (with some exception), and the output it produces is very quantifiable, it is rather easy to measure the performance gained, or lost, by modifying code in one "module" of the algorithm. In the steps involved it often comes down to comparing if performing an object detection approach is more efficient than the more conventional approaches based on cleaning the image and performing various edge/contour detections and similar computer vision-/heuristic-based steps.

4.1 Benchmarking Performance

In order to be able to determine which ALPR systems or combination of algorithms evaluated work the best the different methods were tested individually and in part as a whole system. The conventional methods were evaluated by simply implementing them and noting how quickly they performed the relevant task and determining whether that execution time is technically feasible for a real-time application with similar results as the YOLO-alternative. The different YOLO-models were all tested using the same benchmark-program which ran the model on a short video file which has a few license plates showing and the number of correct (and almost correct) detections were noted. This way, the performance of each individual model could be compared. Due to time constraints the conventional models were not combined into one whole algorithm.

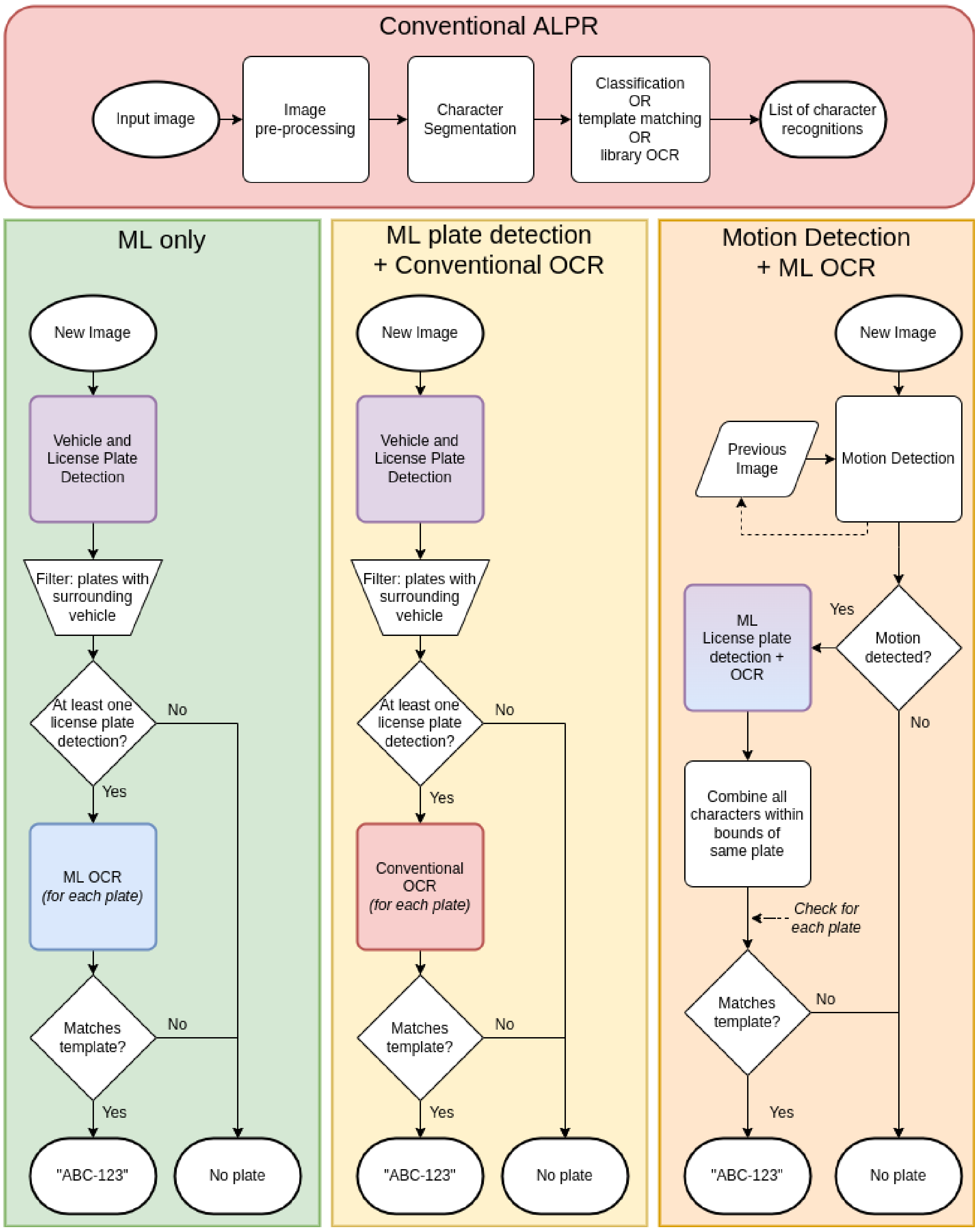

Conventional ALPR
Input image
Image pre-processing
Character Segmentation
Classification OR template matching OR library OCR
List of character recognitions
ML only
New Image
Vehicle and License Plate Detection
Filter: plates with surrounding vehicle
At least one license plate detection?
No
Yes
ML OCR (for each plate)
Matches template?
No
Yes
"ABC-123"
No plate
ML plate detection + Conventional OCR
New Image
Vehicle and License Plate Detection
Filter: plates with surrounding vehicle
At least one license plate detection?
No
Yes
Conventional OCR (for each plate)
Matches template?
No
Yes
"ABC-123"
No plate
Motion Detection + ML OCR
New Image
Previous Image
Motion Detection
Motion detected?
Yes
No
ML License plate detection + OCR
Combine all characters within bounds of same plate
Check for each plate
Matches template?
No
Yes
"ABC-123"
No plate

Figure 37: The different arrangements of ALPR evaluated in this thesis.

4.2 Data Used for Training

To train machine learning models one needs a data set, and it needs to be labeled according to the specific shape the algorithm uses. Freely available data sets are rarely labeled according to the format one needs but converting them will in most cases be a trivial task that can be automated.

Most of the data used was for the ML-based object detection models (the YOLOv5 models). The YOLOv5 models used for training were pre-trained by the author of YOLOv5. This means they have already learned to recognize specific features, which for us means that fewer data are needed for training to learn to recognize a new combination of features compared to beginning with entirely random weights.

For the vehicle and license plate data a combination of data sets was used. First, a license plate/car data set (of Croatian license plates), freely available to use, was downloaded from [77]. This was used to train the first version of a license plate detector but because it did not seem to be quite enough to make the algorithm work well on Finnish license plates this was complemented by collecting a data set of Finnish license plate images. This data set of around 700 pictures of Finnish license plates were used to train a vehicle and license plate detector. Numerous of Finnish license plate pictures were also computer generated in hopes that this could be used to prepare the network before fine-tuning with the real data.

According to the documentation available for YOLOv5 [31] the following matters should be considered with the data set (copied directly from the GitHub page).

- *Images per class. >= 1500 images per class recommended*
- *Instances per class. >= 10000 instances (labeled objects) per class recommended*
- *Image variety. Must be representative of deployed environment. For real-world use cases we recommend images from different times of day, different seasons, different weather, different lighting, different angles, different sources (scraped online, collected locally, different cameras) etc.*
- *Label consistency. All instances of all classes in all images must be labelled. Partial labelling will not work.*
- *Label accuracy. Labels must closely enclose each object. No space should exist between an object and its bounding box. No objects should be missing a label.*
- *Background images. Background images are images with no objects that are added to a data set to reduce False Positives (FP). We recommend about 0-10% background images to help reduce FPs (COCO has 1000 background images for reference, 1% of the total). No labels are required for background images.*

Obviously, neither the Finnish data set, nor the Croatian data set, nor the combination of the two, amount to 1500 images. Acceptable results were nevertheless observed for the vehicle/license plate detector, perhaps because it only has two classes (vehicle and license plate). Using the plate generator program we can generate as many images as we want, so for the character detector this is less of an issue.

All the real images were hand labeled and as such the characters in that part of the data set are not as perfectly boxed as the characters in the generated images are. It is important to note, however, that even though the characters in the generated data might be perfectly boxed at first, that is not necessarily the case after applying perspective transformation; see Figure 38. Thus, our "perfectly"

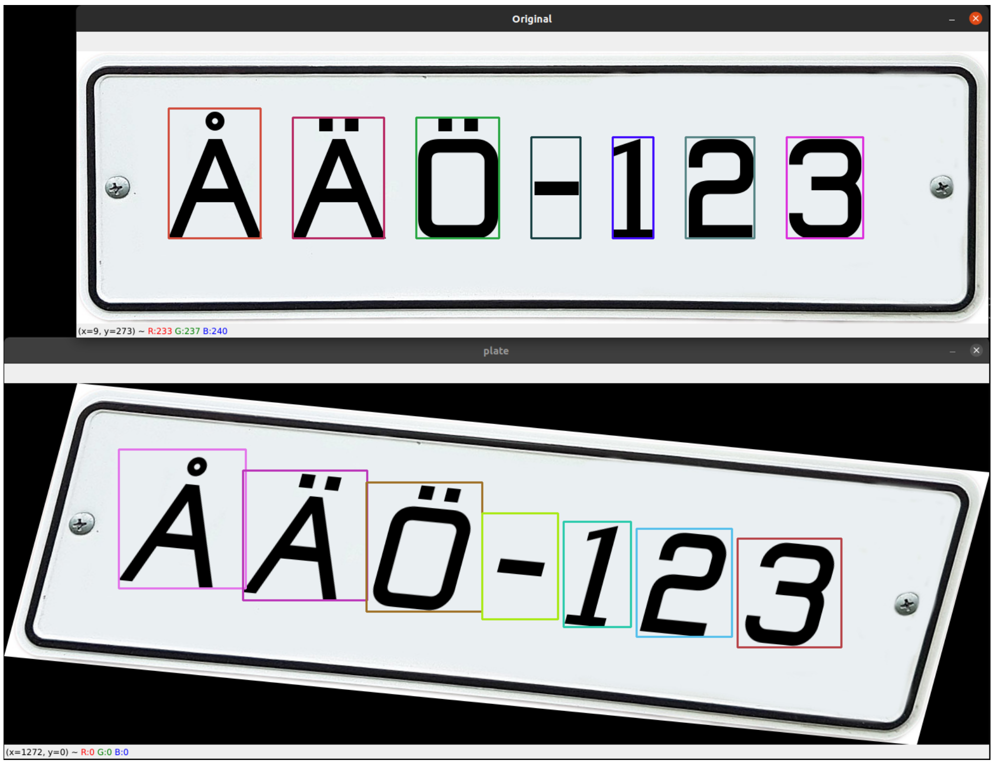


Figure 38: The bounding boxes of a license plate need to be adjusted after perspective transform.

machine-labeled generated plates may also cause some localization loss.

4.2.1 Finnish Data Set

The Finnish license plate images were taken, by the author of this thesis, with a OnePlus 6T phone using the default settings and most of them from the inside of a moving car. Many of these images were so blurry that they were unusable for character recognition, but were usable for vehicle and license plate detection. Thus, the data had to be separated into those suitable for character recognition and those that were not. The images were not originally of resolution 1920×1080, and since that is, perhaps, the most common resolution as of 2021, the images were cropped so that a 1920×1080 sub-image was collected, centered on the most relevant part of the image. This was done manually. For training the vehicle and license plate detector these images were then resized to 640×640, which naturally distorts the proportions of the image. This is not an issue if all the input images have the same resolution (1920×1080). For the character detector, resizing the images to 640×640 like that is not recommended. Because the license plate character images' dimensions depend on the original resolution and the angle and distance to the camera, it would be difficult to generate a good set of bounding boxes, and it would be difficult to make the models to generalize well. The better approach here is to resize the larger dimension to 640 and pad the other dimension with black pixels (zeros), such that the image is centered in the padding, this is sometimes referred to as "letterboxing". By padding the image we are not utilizing the entire 640×640 resolution, which essentially means that we are wasting compute time. This could be remedied by inserting multiple license plates into the same 640×640 image, provided that they would fit. This would be immediately useful if there are two license plates visible at the same time. Since most license plates are well above the minimum width of twice the height, it would be feasible. Doing this would imply a significant performance improvement. This was not tested in this thesis, but the run-time for a detection using YOLOv5 will be more or less consistent, regardless of whether the entire image contains relevant information or not.

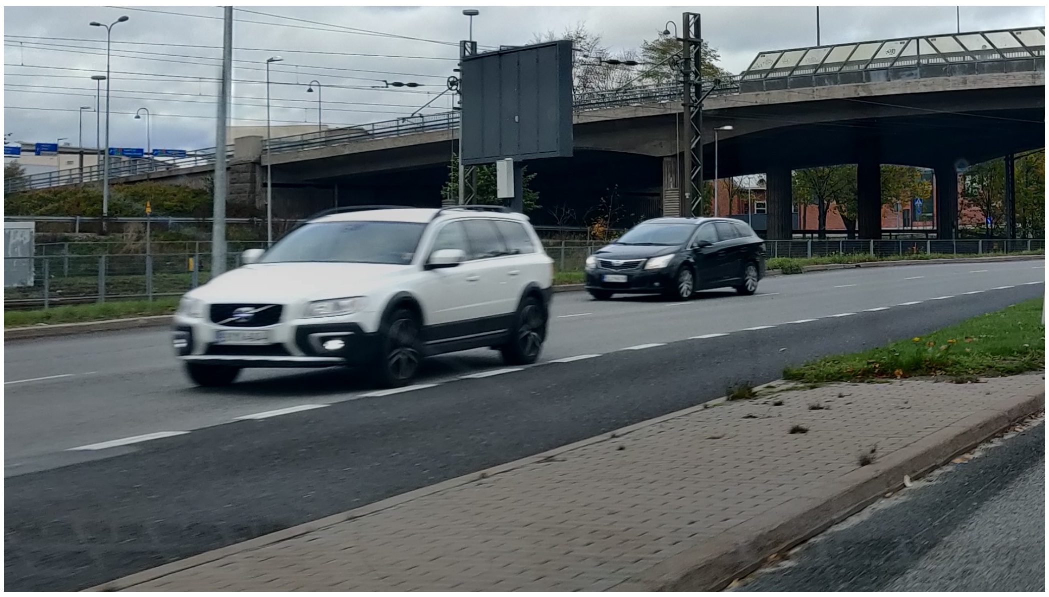


Figure 39: An example of one of the blurrier images (only suitable for vehicle detection) in the Finnish data set.

4.2.2 The Croatian Data Set

The data set of rear views of Croatian vehicles and their license plates has around 500 images [77]. It was gathered for a student project by students at The University of Zagreb in 2003. The data set is not labeled so labeling it oneself is necessary. The images are of resolution 640 × 480 pixels, which is smaller than our 640 × 640 input size expected for YOLOv5, so a padding of black pixels is applied to the smaller dimension.

4.2.3 Generated Data

A large data set of Finnish license plates was automatically generated using a program created by the author [78]. This program has many options for adding noise, picture transformation, using different fonts and license plate formats as well as different license plate base images. It proved quite useful for adding variation to the training data. A few examples of generated license plates can be seen in Figure 41.

As mentioned earlier, in section 4.2, the bounding boxes of the generated license plates would be larger than the characters after applying perspective transformation. To improve the bounding boxes again, the plate generator program

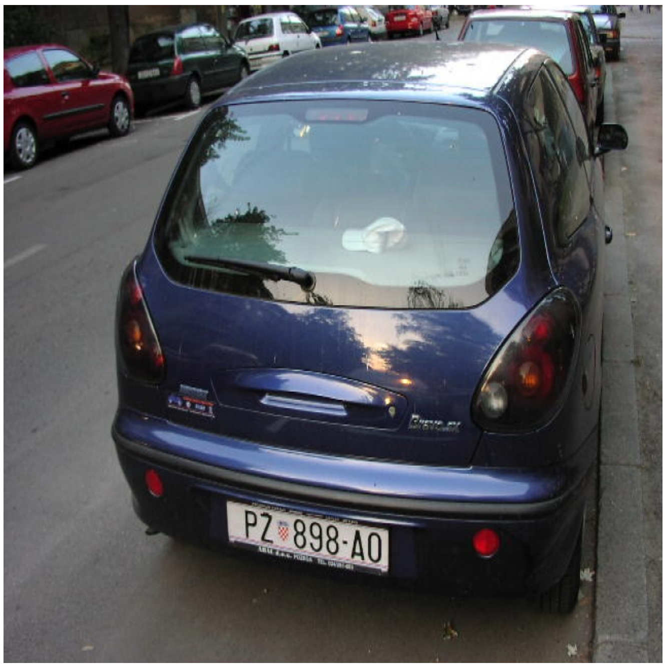


Figure 40: An example of the images in the Croatian data set.

attempts to detect (before noise is applied) which contour within the bounds of the bounding box that actually is the character and adjust the bounding box according to that. This is more difficult with the Å, Ä and Ö characters as they have disconnected parts, so for those the bounding box is simply adjusted to the bottom of the best contour and the left, top and right edges are set to the first coordinate where a dark pixel is (starting from the corresponding edge). The hyphen character is also treated slightly differently, as the box we produce for it is set to be the same height as the other characters, so the bounding box adjusting process does not change the y coordinates of it, but only the x coordinates. These factors mean that, in general, any model trained on this generated data will not be able to produce as good bounding boxes for these edge case characters as for the other ones. Figure 41 demonstrates the results of the adjustment process. Improving the method for adjusting boxes would likely make the data easier to train on.

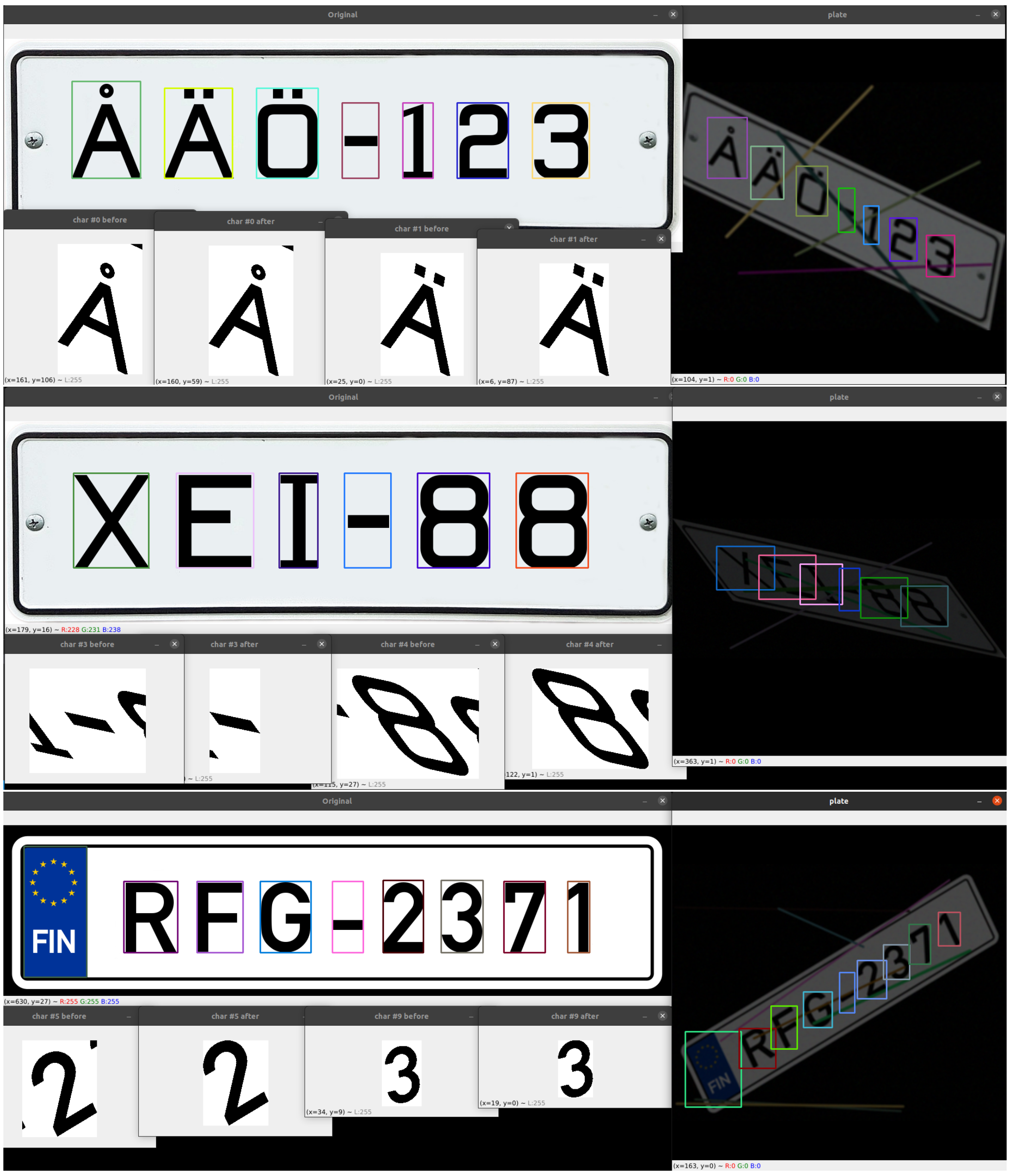


Figure 41: A few examples of generated plates, with and without noise/transformation and some pictures of the characters demonstrating the box-widening effect mitigation. The images labeled "original" are the generated images without any effects applied, and those labeled "plate" are the final images (where the bounding boxes are only visible for demonstration).

A smaller number of data was also generated for a character classifier, some of these were hand-drawn using a small helper program written in java and some were generated using a simple OpenCV-based python program.

4.2.4 Data format

After acquiring a data set one first needs to label it according to the format used by the algorithm one intends to use. For YOLO, this means that the label files have a separate row for each object with each data point separated by a space. The values are "class x_center y_center width height" and the coordinates and dimensions are relative to the size of the image (floating point values between 0 and 1). There are open source programs to help with labeling e.g. [79] (it does not write the labels in the YOLO format, but the conversion is trivial).

4.2.5 Ways to improve the data

A simple program, which allows the user to adjust each label bounding box to more closely represent the ground truth, was created (using the same principles mentioned in section 4.2.3). After using this program to improve a license plate character data set, a twin data set with slightly randomly altered boxes was generated from the newly fixed boxes, and one model was trained with each data set. The training performance difference can be seen in figures 42 and 43.

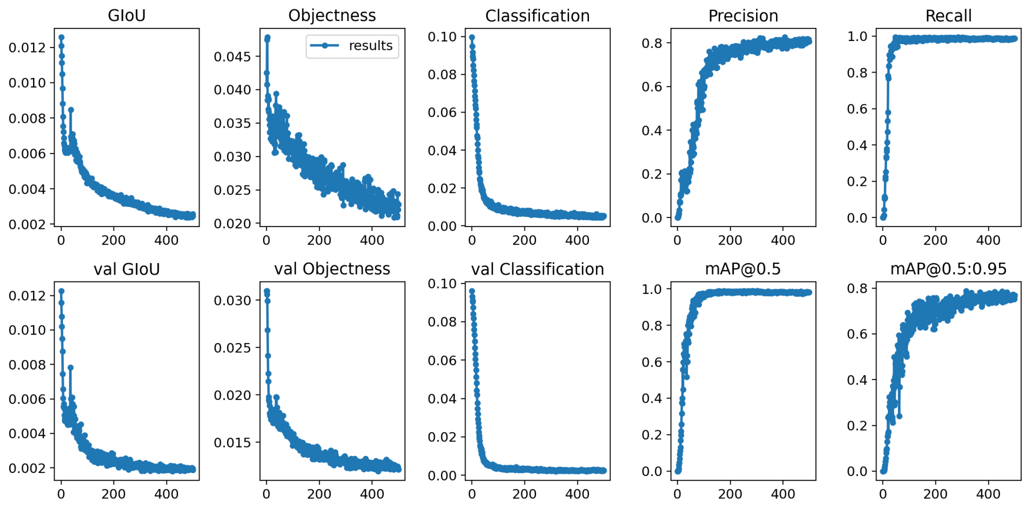


Figure 42: Training output of training with well boxed labels using YOLOv5. The x-axis is the epoch number.

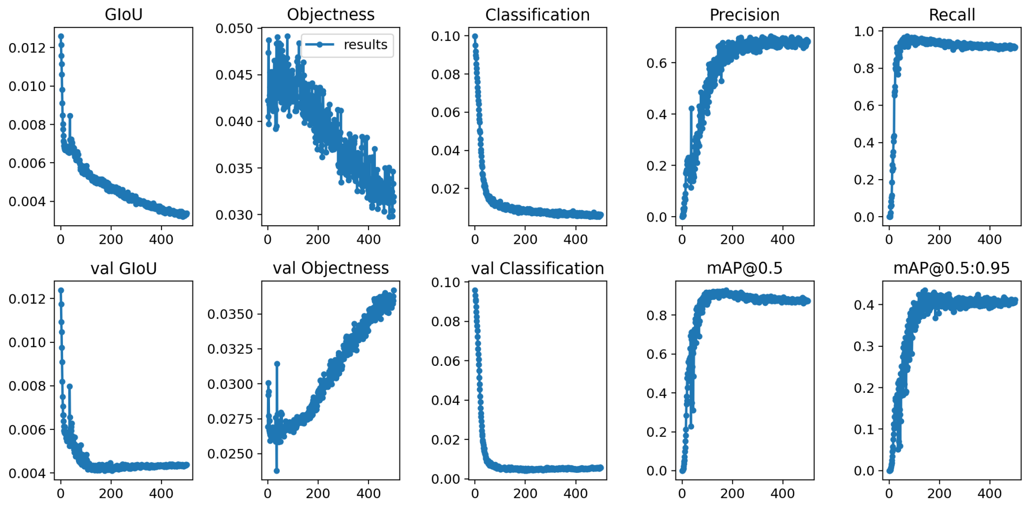


Figure 43: Training output of training with badly boxed labels using YOLOv5. Same images and labels as in the previous figure, but with small random mutations to the boxes coordinates and size. The x-axis is the epoch number.

4.3 Challenges

4.3.1 Variability

The variability in an ALPR system comes in two separate forms, the variability of the input data and the variability of the tool chain.

The variability of the input data stems from many problems. One such problem is the installation/environment, i.e. how the camera will be positioned in space in relation to the vehicles it will be observing, the lighting conditions, if the camera continuously processes frames or if a tandem system triggers the camera to take a picture and even if the camera is moving or stationary. A different camera will also produce a different color shade given the same set of circumstances and will not necessarily have the same resolution or aspect ratio as another. All of these are factors one must take into consideration, and they will have a major impact on performance. It is easy to produce decent results in a lab environment in controlled lighting and conditions, whereas the real world is a different matter. These factors might place constraints that one needs to follow when installing the camera, such as ensuring that one does not exceed the maximum distance from the area of interest (to ensure high enough image resolution of the license plates). However, a robust system should be able to handle all of this variability.

The variability of the tool chain comes from using different algorithms that are sensitive to different kinds of problems. One algorithm might perform amazingly well, at best, but only at 15 o'clock on a sunny day in June because that is the only time when the vehicles' license plate would be in daylight, while another algorithm might produce results that are more consistent, while less impressive at best, because it is less dependent on lighting conditions. This is similar to the variability of the input data to some degree but the aspect we are interested in is how well the algorithm handles it, not how to stabilize the environment.

Area of interest, that is to say the area in which we expect to find license plates, and outside which we should discard license plate detections or, even better, not perform the detection at all. If the developer of the ALPR system is not in control of the installation of the physical camera, it will most likely be necessary to restrict the area of interest, if possible. In an ALPR installation with a stationary camera, the area of interest can be restricted at install time by creating an image mask which is referenced at run-time. Such a mask can simply be a black/white image of the same dimensions as the processed images that has one color for the area of interest and another for the "ignore zone". Any object/detection with a center point outside the area of interest is then discarded, or an image mask with a transparent area of interest and opaque ignore zone can be overlaid on top of the input image to block the unwanted areas before detection.

Pixels on area of interest is something that is important to consider for any object detection model. The resolution at which the images are processed, at what distance from the camera the objects are likely to appear and how large a part of the field of view they are going to occupy will all affect the resolution of the area of interest. The size of a pixel (in real, physical distance) can be calculated by dividing the arc length by the size of the image along that axis (in pixels). The arc length can be calculated based on the distance of the object and the field of view (degrees); see Figure 44. The formula to calculate the arc length is

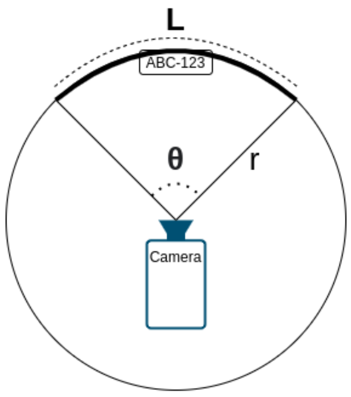


Figure 44: The relationship between physical size per pixel, field of view and distance from camera to object.

$$L = \theta/360^\circ \times 2 \times \pi \times r$$

where theta is the field of view and r is the distance from the camera to the object. Using this formula we can determine the pixels per Finnish license plate at certain distances, as shown in Table 1. While the input resolution is something one can configure, training a network of a size like 1440×1440 (same pixel amount as 1920×1080) or similar would be many times slower.

Resolution	Distance	Pixel Width	Pixels / Plate (40 cm)
640	1	0.16	244
640	2	0.33	122
640	3	0.49	81
640	5	0.82	49
640	10	1.64	24
640	15	2.45	16
1080	1	0.1	413
1080	2	0.19	206
1080	3	0.29	138
1080	5	0.48	83
1080	10	0.97	41
1080	15	1.45	28
1440	1	0.07	550
1440	2	0.15	275
1440	3	0.22	183
1440	5	0.36	110
1440	10	0.73	55
1440	15	1.09	37

Table 1: Pixels per distance for pictures 640, 1080 and 1440 pixels wide assuming the camera has field of view of 60 degrees.

5 Character Recognition

Once we have a picture of a license plate image we are ready to start processing what characters are present in the image.

There are many approaches to this, but in this thesis we will qualitatively evaluate a couple of free to use open source libraries, YOLOv5 and some more conventional heuristics-based approaches. In general what they all have in common is that based on a given image they will produce a set of character recognitions (character and location in image).

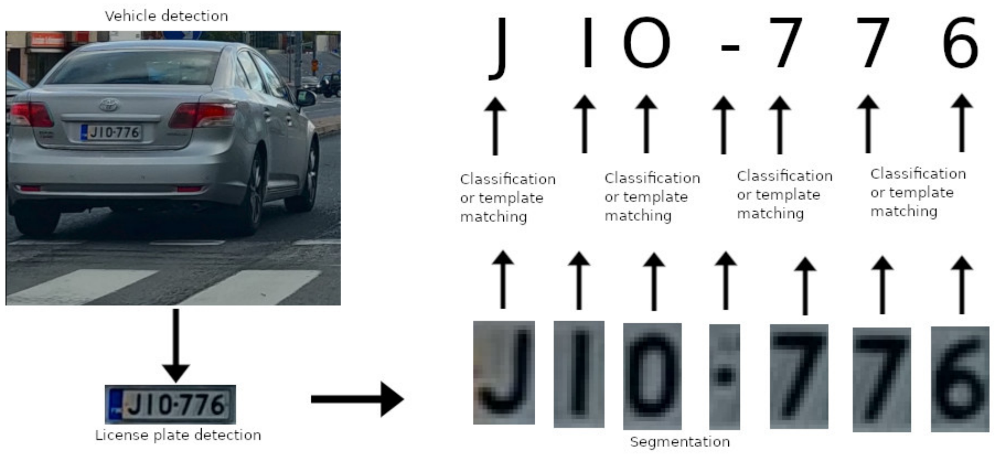

Vehicle detection
J I O - 7 7 6
Classification or template matching
Classification or template matching
Classification or template matching
Classification or template matching
License plate detection
Segmentation

Figure 45: ALPR algorithm with the segmentation step.

There are libraries for this such as Tesseract OCR, OpenCV's built-in text recognition, and EasyOCR. The problem with all of these is that they tend to work well for images with very good visibility and picture quality. EasyOCR perhaps being the exception to this rule, but it does not perform flawlessly either and, according to my tests, it can take minutes to process even just one image and that is when running on the GPU.

5.1 Using Tesseract OCR

To test Tesseract OCR, a minimal program was written (Figure 47) and fed some variations of the same images. In Figure 48 only the middle left image was recognized. Tesseract OCR found no text in the other images. In Figure 50 Tesseract OCR detected no text in either image, even though the text is as perfect

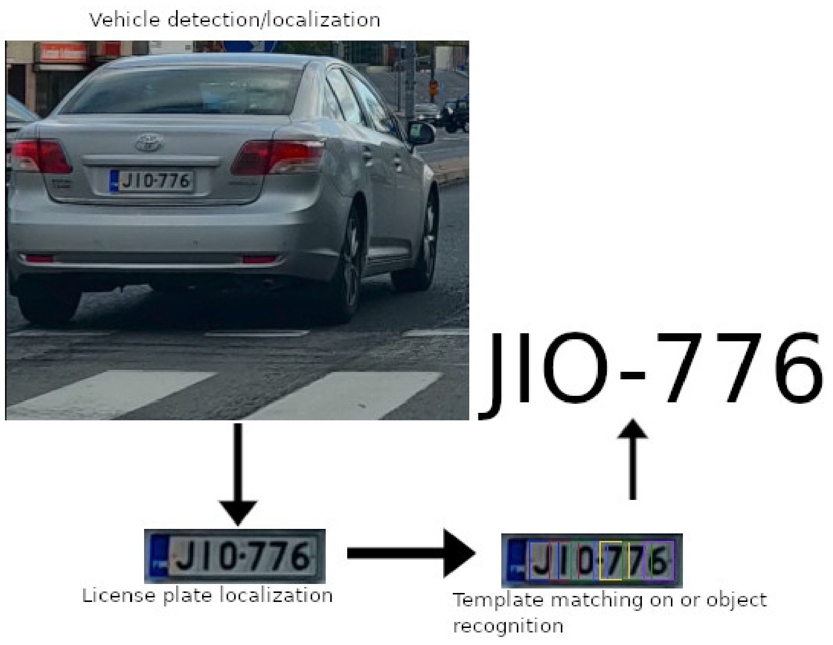
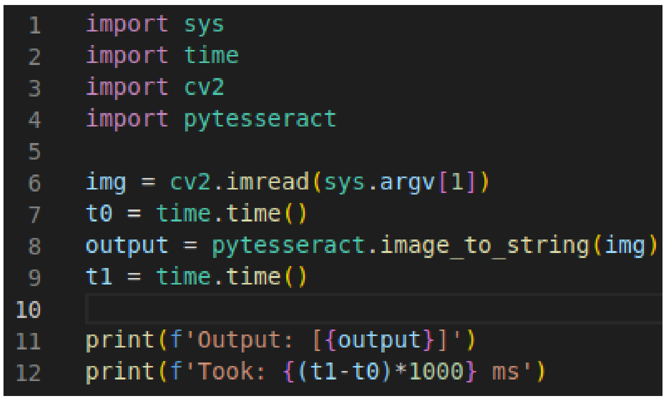

Vehicle detection/localization
JIO-776
License plate localization
Template matching on or object recognition

Figure 46: ALPR algorithm without segmentation step using sliding template matching or ML object detection for recognizing characters and their positions in an image. Cleaning the image might be necessary before template matching but not before ML object detection.

as could possibly be imagined, being computer generated and noise free.



```python
import sys
import time
import cv2
import pytesseract

img = cv2.imread(sys.argv[1])
t0 = time.time()
output = pytesseract.image_to_string(img)
t1 = time.time()

print(f'Output: [{output}]')
print(f'Took: {(t1-t0)*1000} ms')
```

Figure 47: Minimal Tesseract OCR test program

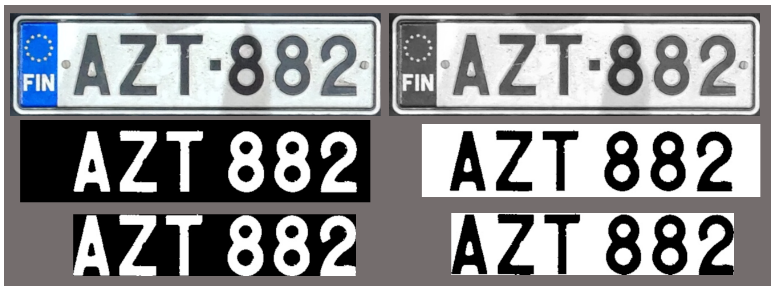


Figure 48: Image variants of AZT-882 license plate that Tesseract OCR was tested on. The original image was of size 650×180 and the others were the same or smaller. The processing time varied between 100ms and 140ms.

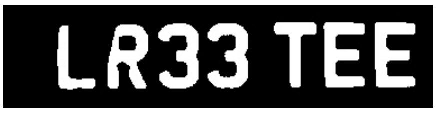


Figure 49: Recognized text: "Beceem is". Processing time: 117ms.

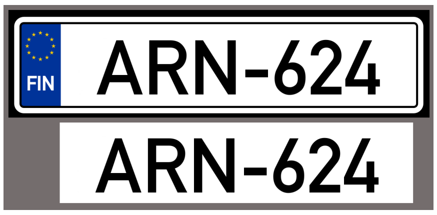


Figure 50: Generated license plate, raw and cleaned. Image sizes: 1280×326 and 1074×244. Processing times: 148ms and 124ms.

One caveat with Tesseract OCR (arguably the best open source OCR engine) is the performance, because even if one achieves good cleaning of one's images prior to passing them through, it sometimes detects nothing at all, and it always takes a long time. The, very minimal, python program in Figure 47 was used to obtain an idea of the performance of Tesseract OCR. A set of license plate images were cropped and passed to the program (images cropped to contain ONLY the license plate) and the fastest processing time was around 100ms, which may be fast enough depending on one's specific use case and requirements. Cleaning said images can be quite fast, however. When tested using six different images two of them gave the correct result, two of them were one and two letters off respectively and two gave an empty string as output (because they only found one single contour in the image). The cleaning process of said images took 32 milliseconds at the most and 3 milliseconds at the least. The resolution of the largest image was 735×481 and the resolution of the smallest image was 131×77. The tests completed seemed to suggest that adding some padding around the characters (so that the edges of the characters do not touch the edges of the images) of the images processed by Tesseract OCR increases the accuracy.

5.2 Using Object Detection

Instead of trying to ensure perfect conditions and cleaning the input images to perfection, one could train an object detection model for this. One simply needs to create or acquire a data set of images with characters in them, and their corresponding labels, preferably of license plates, because that is what we are looking for, and train the network on that. We know that it will not be too many classes because the COCO data set has 80 classes, whereas in this thesis we will end up training on 41 classes (29 characters + one class for license plate symbols + hyphen + the digits, zero through nine). Since we know that the YOLOv5 network is quite fast when trained on the COCO data set and with 80 classes (more complex than our 41 classes), we can make the assumption that this will also be the case for simple characters, and since we have fewer classes we may be able to decrease the size of the network to make it run faster. This approach means that we do not necessarily have to perform any perspective transform to create perfect conditions, because we can simply train our model to recognize license plate characters at an angle. It might, however, still be advantageous to perform perspective transformation.

As far as license plate recognition is concerned, what we consider the most important is simply which characters are present in the image, and in what order. However, exactly how large each character is, e.g. 48 pixels or 50 pixels, does not matter as much. This means that we can reduce the effect of the localization part of the loss function and increase the effect of the class part. This way, when training the network it learn how to more effectively differentiate the characters rather than exactly how big each one is expected to be. Since part of the data set *might be* (*is* in this case) hand-labeled, the size may vary by a few pixels. Because the characters of the training/test data images are often relatively small, e.g. 30 pixels high, but they might be 40 pixels high according to the label, the localization loss might be quite large. This can affect the training performance as a whole, even if the correct class was assigned. Thus, because the training algorithm does not know which labels are good and which labels are not (it assumes they all are perfect), it is simply better to reduce the "punishment" of the part of the loss function that tries to improve localization. Thus, the "giou" hyperparameter (localization loss) was reduced to 0.005 from the default of 0.01 and the "cls" hyperparameter (class loss) was increased to 0.5 from the default of 0.4.

5.3 Using Heuristics

We can use some basic heuristics in order to perform character recognition. To begin with, we know that (at least in Finland) the license plate usually has black letters and a white background. We also know that the characters do not touch each other, and we know that the letters come first, followed by a hyphen, followed by the numbers. License plates for motorcycles, however, follow slightly different rules as the letters and numbers are stacked horizontally instead. Based on these rules, we can create an algorithm that segments all black shapes and performs classification/template matching on the cropped images.

5.3.1 Character Segmentation

For the conventional/heuristics-based approaches it helps to clean the image first, so that one is better able to isolate the important parts of the image, the characters themselves. The closer one can come to the original character shape the better. In the best case scenario, we want to end up with a picture where the background is entirely black and there is nothing except for the characters that is white, or vice versa. The method chosen by the author of this thesis is as follows. First, we convert the image to gray scale and blur it to reduce noise but still preserve the largest most important shapes. Now we have an image with much less noise, which is ready for binarization by thresholding.

After thresholding, there are a few different popular methods. One of these is to look for peaks and valleys in the pixel intensity in each axis and another is to look for contours. The pixel intensity sums of a rotated image and its skew corrected counter-part are shown in figures 51 and 52. Based on the histogram of the rows in the skew corrected image one can clearly tell where the top and bottom lines are, and in the histogram of the columns it is also easy to tell where the left and right of each character is. The histograms of the original image are not as clear.

When going for the contour detection approach, it was noted that the "findContours" function in OpenCV sometimes did not find anything except one large contour in the image. To remedy this, simply removing 10% off each side of the image sometimes helped. This extra cropping step will obviously not always work, but it is, relatively speaking, not a costly operation. When this extra cropping step was needed, "findContours" was run again. If at this point the number of contours is larger than some threshold value (e.g. three in Finland, because Finnish license plates need to have at least two letters and one digit [80]), we can sort the contours by area and keep the largest N (maximum number of expected characters plus margin) contours and turn them into bounding boxes using the "boundingRect" function in OpenCV. Now the contour boxes can be analyzed and filtered based on their shapes, e.g. any contour box taking up more than half

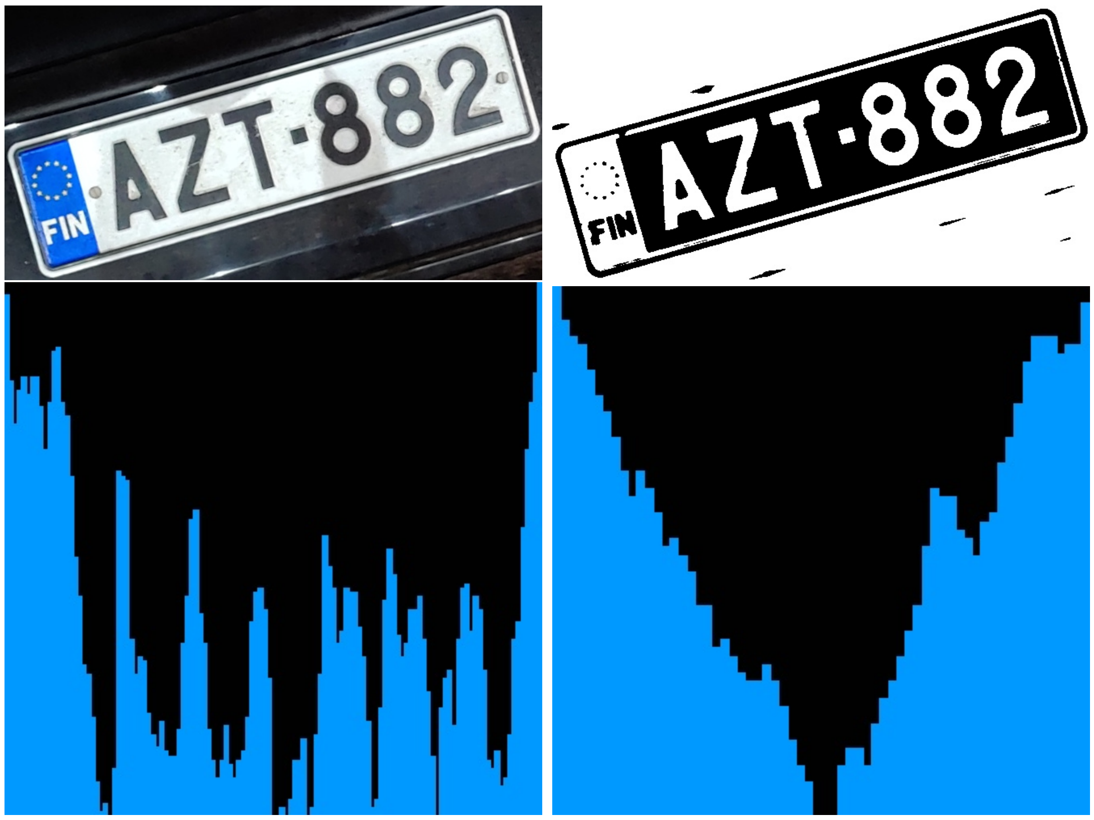


Figure 51: The original image, the cleaned and thresholded image, the pixel intensity by column and the pixel intensity by row.

of the width and contours with too much overlap on the x-axis (except if the relevant plate formats allow multiple rows). For Tesseract OCR and sliding template matching, the final step is to isolate the characters by cropping them out and placing them on a clean black background. For the single template matching and classifier methods, we simply crop the individual characters and pass the images to the template matcher or classifier.

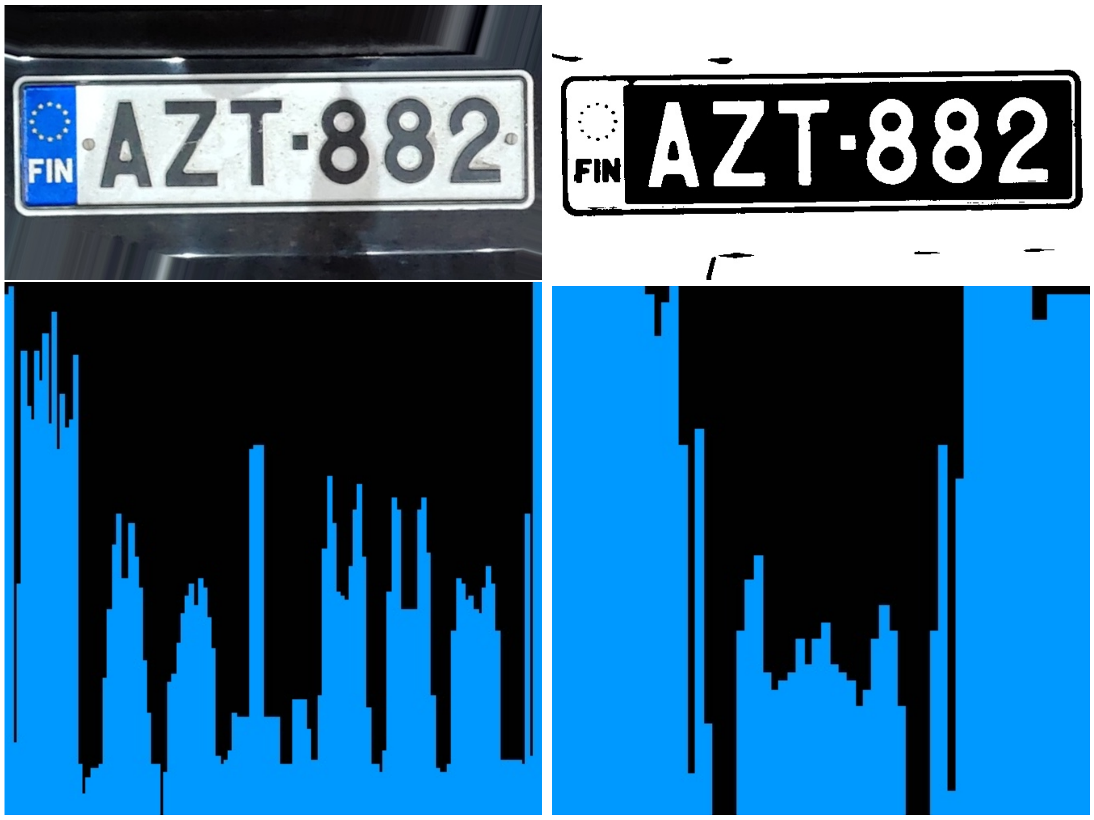


Figure 52: The original image, the cleaned and thresholded image, the pixel intensity by column, the pixel intensity by row.

The following is a description, of how one can clean images for character recognition, using a combination of pseudocode and actual python code utilizing the OpenCV library. Figure 53 shows the results of the first steps of this process.

```
-p: Gray scale conversion
    - convert the image to grayscale
        - gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
-p: Blur the image
    - perform blurring on the image to reduce noise
        - blur = cv2.bilateralFilter(gray, int(image_width*0.02), int(
          image_width*0.1), int(image_width*0.17))
          - These values were chosen simply by trial and error
-p: Thresholding
    - binarize the image by thresholding
        - thresholded = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV
           | cv2.THRESH_OTSU)[1]
-p Contour detection & reduction
    - detect contours in the image
        - cnts = cv2.findContours(img.copy(), cv2.REzR_EXTERNAL, cv2.
          CHAIN_APPROX_SIMPLE)
          cnts = imutils.grab_contours(cnts)
        - if too few boxes are detected, try the two above steps once
          more after removing 10% from each side of the image.
          cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
          boxes = [cv2.boundingRect(cnt) for cnt in cnts]
-p Contour rejection & character isolation
    - filter the contour boxes according to some pre-specified criteria
    - determine background color and create a new image of the same size
    - insert character crops into the same positions as in the original
      image
-p Perspective transformation / skew correction
    - determine perspective transformation or rotation of image
    - perform perspective transform or skew correction
```

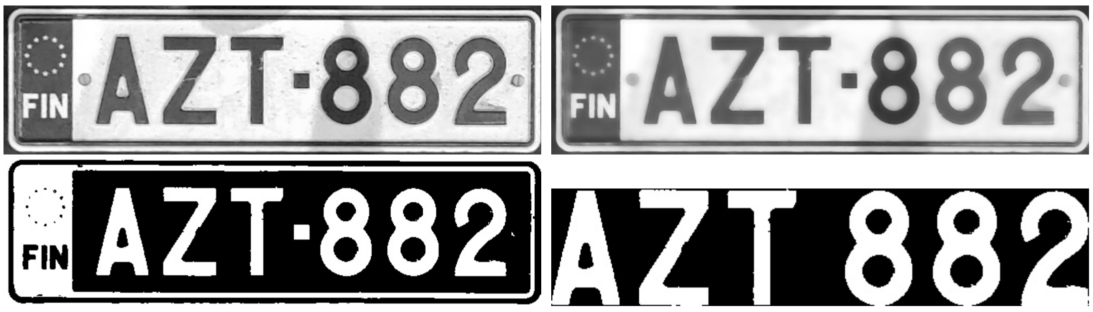


Figure 53: ***Successful image cleaning. 17ms total processing time.***

- Top left: grayed, 0.1 ms processing time

- Top right: blurred, 15 ms processing time

- Bottom left: binarized, 0.1 ms processing time

- Bottom right: cropped smaller, contours filtered and isolated, 1.4 ms processing time

5.3.2 Template matching

There are three main problems with template matching and in our case also a fourth one. The first is that the template image needs to be the correct size or one will be looping through (almost) every pixel, doing the template matching once for the area around each one. The second is that one needs to do the template checking for each template to find the strongest match (provided that there are multiple options). The third is that it becomes difficult to determine which template is correct as soon as the perspective in the target image is different from the perspective in the template image. When comparing characters, like in our case, the characters in the image need to use the same font as the characters in the templates. Unlike a neural network that can learn to recognize a combination of patterns, template matching algorithms only look at how similar two images are.

The first problem of the image size can, in some cases, be solved by simply cropping the image perfectly around the character. That is to say that after cropping the character which we are trying to recognize it will have the same kind of margins as our template does. The second problem is, in a sense, solved at the same time as the first one. By making sure that the template is the same size as the image we are comparing, we only have to run it once for each template, making it drastically less expensive to have multiple templates.

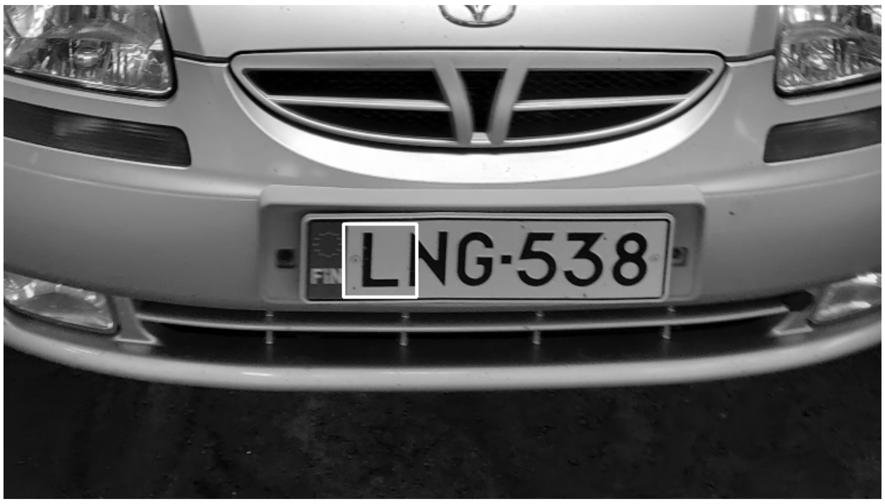


Figure 54: The template matching algorithm was used to search for the letter L using a template created from using the din1451mittelschrift font (seems to be one of the closest, free alternatives, to the Finnish license plate font). Each available method was run 100 times and the average execution time was calculated: around 5ms. The image used is of the dimensions 768×432. "TM_ CCOEFF" was the fastest of the ones that was positioned correctly.

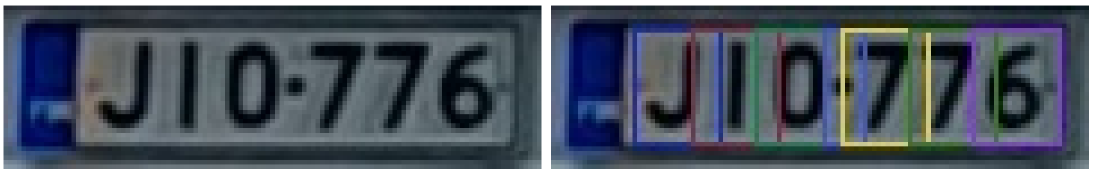


Figure 55: Matches on a plate. All characters, except for the letter O which was recognized as the number 0, were correctly recognized. Resolution: 119×36 and processing time: 9.5ms.

As shown in Figure 54 it quickly becomes infeasible to run template matching using multiple templates if the image is not cropped very close to the license plate. It took around 5ms to run sliding template matching using one template. Running it on all possible characters will multiply that processing time by a factor of around 40, leaving us with a processing time of 200ms for one frame. In Figure 56 it took 113 and 87 milliseconds respectively, but these images are still very big, even though cropped close to the license plate. Sliding template matching can be relatively fast however, if the resolution is low enough, as seen in 55. There it took around 9.5ms to detect all the characters in the license plate (except for the O which was read as a 0). Bigger images will always take longer to run but if they are well cropped they can always be scaled down to the same minimum resolution, given the same plate format.

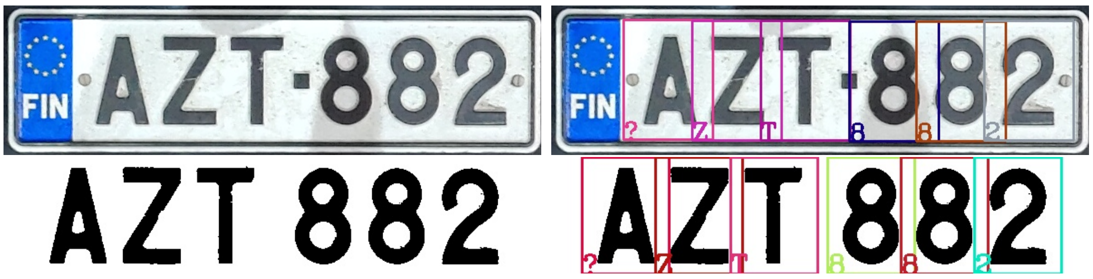


Figure 56: Template matching with and without cleaning, the "?" character recognized was the Swedish letter "Å", in both cases, otherwise all matches were correct. Original image: 650×180 and 113 ms processing time. Cleaned image: 594×128 and 87 ms processing time.

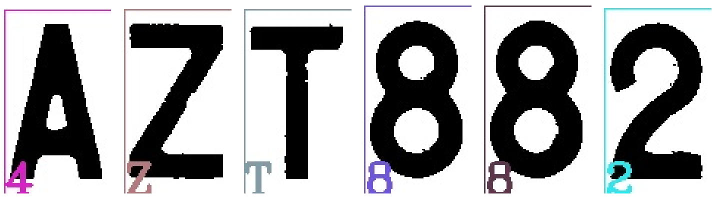


Figure 57: Template matching on individual character crops. Total processing time: 6.4ms. Templates: 40. Template size: 32×24. Image size: 32×24. Looking at the "A" character crop compared to the "A" template, the bottom right "intersection" matches that in the "4" template better than the "A" template.

Running the template matching on an individual image is the fastest way, because it means one run per template, per image.

In order to make sure that the template and character image sizes match, one can calculate a histogram (0.1ms for a 32×24 image) of the pixel intensity for each axis of the image. Based on the histogram one can determine the edges of the characters in the image crops and resize the image so that it matches the template. A binary image with black background will only have zeros until the first row/column that has some piece of the character (or unfiltered noise) in it.

The false detection can be remedied using the information we have about the image itself (width, height), and about what we expect the plate to look like. We

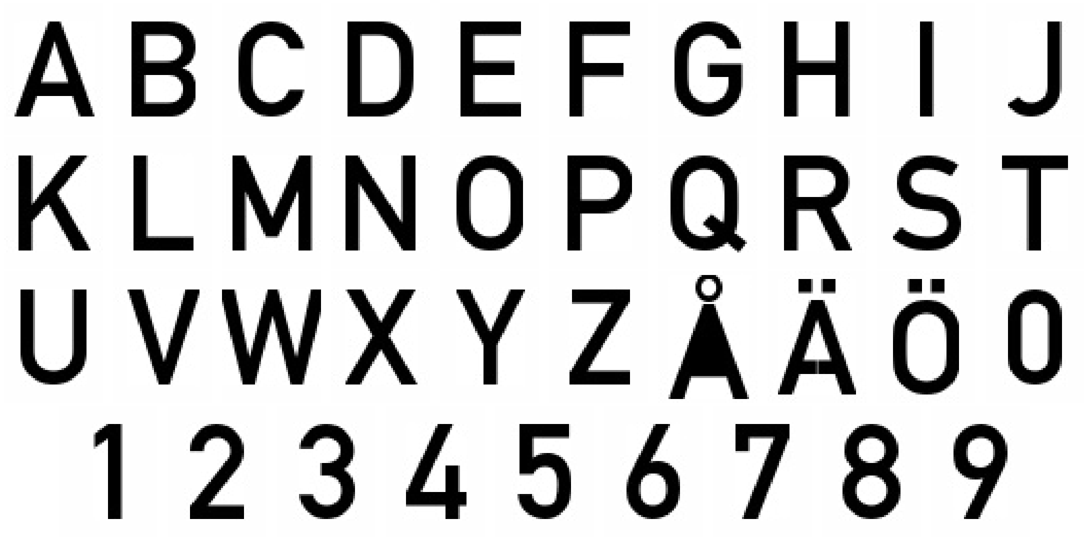


Figure 58: The relevant templates.

can make the assumption that the 0 should be an O, because it is on the left side of the hyphen the license plate would not fit any of the Finnish license plate formats otherwise.

The third problem, of the perspective, is especially important to address when performing template matching, as having an image of inconsistent character sizes will have a large effect on template matching, as seen in figures 59 and 60. Even though a 1 to 1 ratio for the transformed image (Figure 59) yields a perfect result, we are not necessarily able to crop the image perfectly around the license plate every time. Thus, the problem of the template size remains, even if performing sliding window template matching. If performing perspective transform followed by character segmentation we can utilize the histogram approach mentioned above to adjust the image and/or template size, to address the size problem further. Perspective transform can be addressed as described in section 3.3.4.

The last problem of different fonts can be addressed by using separate templates for each font. This comes with the disadvantage of a larger processing cost, but if one's ALPR system is only supposed to run in a specific country one should be able to make some assumptions about the font used and, thus, only use one set of templates. One option is to supply a set of templates for each country and let the end user specify the country and load the template images based on that.

Figures 59 and 60 show the importance of transforming the image to the correct perspective and using the correct template size.

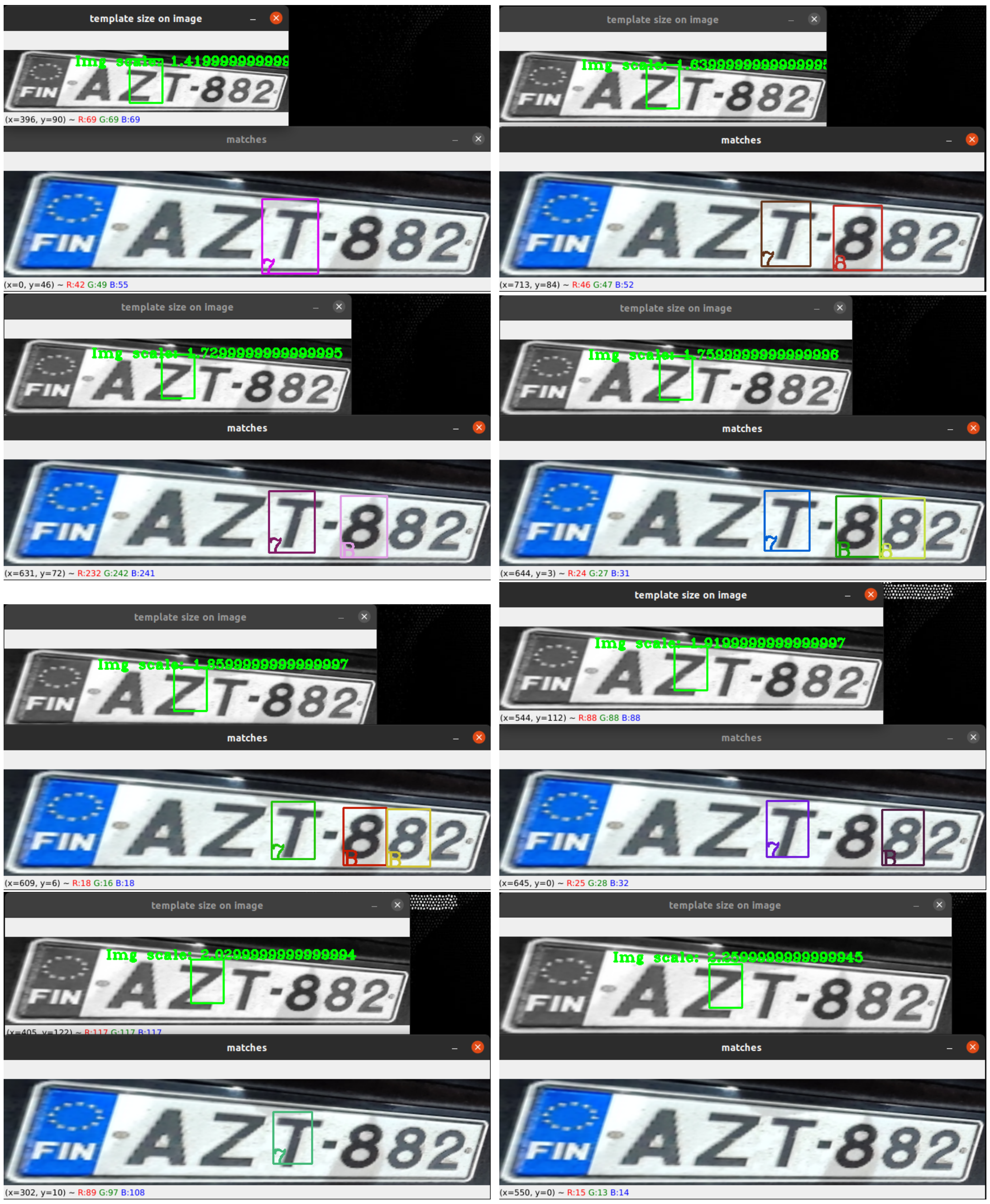

template size on image
Img scale: 1.419999999999
(x=396, y=90) ~ R:69 G:69 B:69
matches
(x=0, y=46) ~ R:42 G:49 B:55
template size on image
Img scale: 1.6399999999999995
matches
(x=713, y=84) ~ R:46 G:47 B:52
template size on image
Img scale: 1.7299999999999995
matches
(x=631, y=72) ~ R:232 G:242 B:241
template size on image
Img scale: 1.7599999999999996
matches
(x=644, y=3) ~ R:24 G:27 B:31
template size on image
Img scale: 1.8599999999999997
matches
(x=609, y=6) ~ R:18 G:16 B:18
template size on image
Img scale: 1.9199999999999997
(x=544, y=112) ~ R:88 G:88 B:88
matches
(x=645, y=0) ~ R:25 G:28 B:32
template size on image
Img scale: 2.0299999999999994
(x=405, y=122) ~ R:117 G:117 B:117
matches
(x=302, y=10) ~ R:89 G:97 B:108
template size on image
matches
(x=550, y=0) ~ R:15 G:13 B:14

Figure 59: Comparison of matches found depending on the size of the template relative to the license plate image. The first match was found when the image height was 1.42 times the template height, and no matches were found starting from 2.6 times the template height.

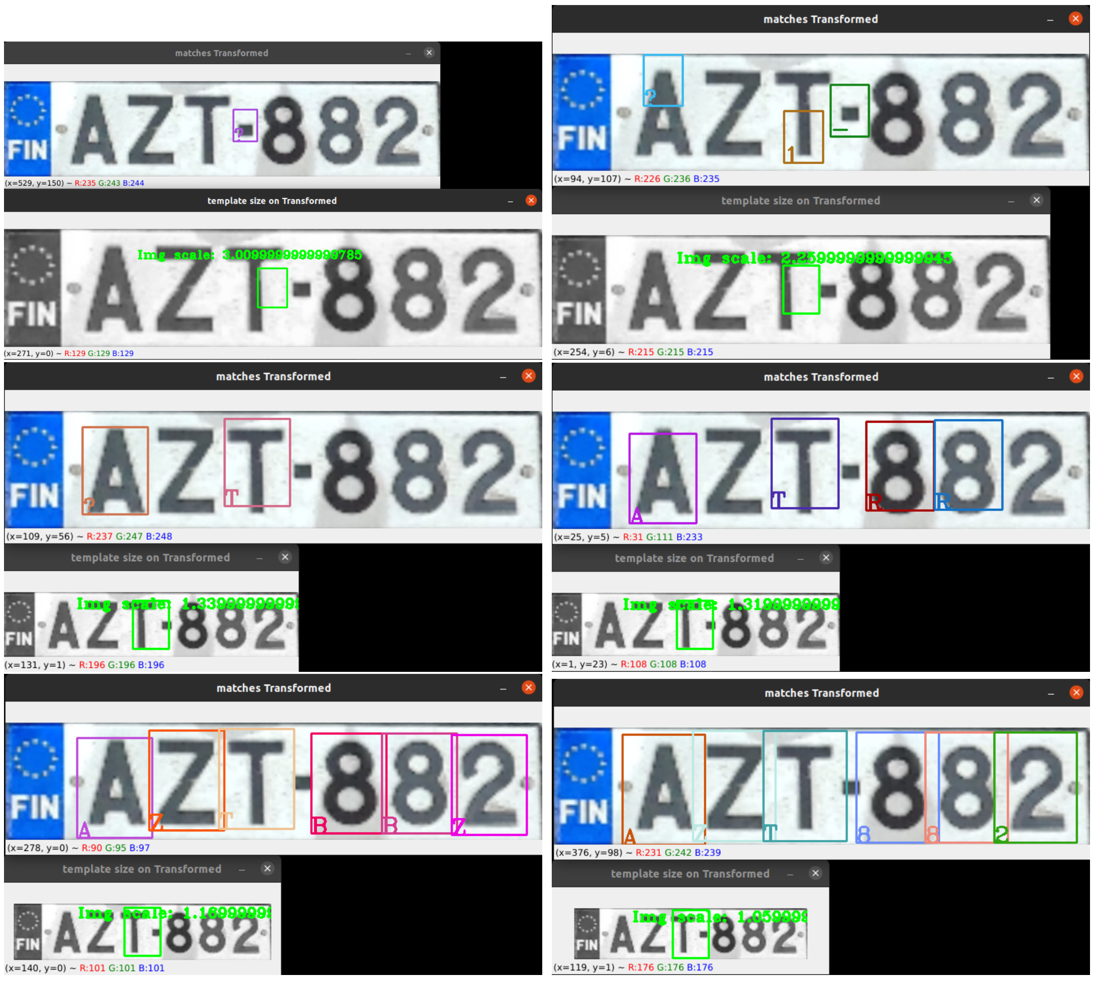
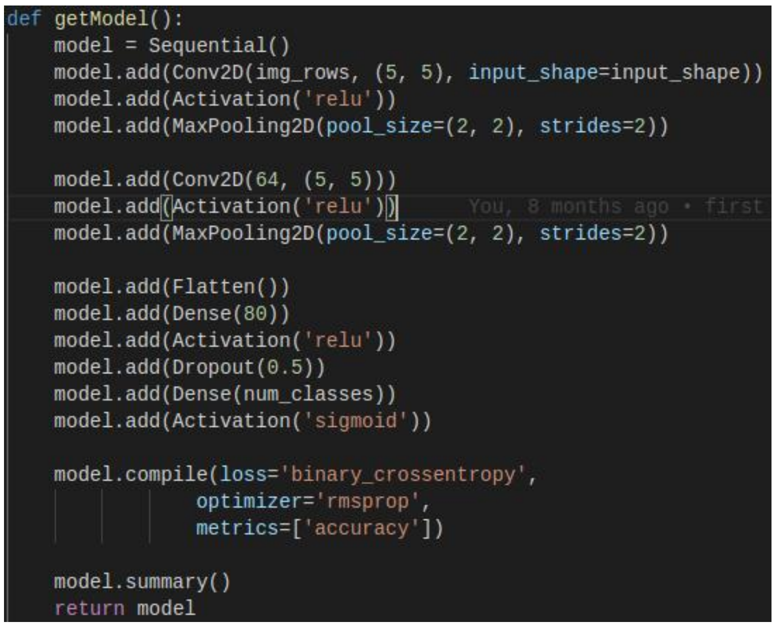

matches Transformed
template size on Transformed
Img scale: 3.0099999999999785
Img scale: 2.2599999999999945
Img scale: 1.3399999999999
Img scale: 1.3199999999
Img scale: 1.1699999
Img scale: 1.059999

Figure 60: Comparison of matches found depending on the size of the template relative to the license plate image on a perspective corrected image at a few different values of scale. Going down to a 1 to 1 plate-height to template image-height yields the best (perfect) results, which makes sense, as the template images were generated with the same padding as is used in the Finnish license plates and the transformed image license plate is cropped (nearly) perfectly.

Performance gains can be had by using what we know about the license plate format. For example, if we are looking at the first 2 characters (from the left) in a Finnish license there is no point in comparing with digit templates, and vice versa if 3 letters have already been recognized. Thus, for a Finnish license plate we can potentially reduce the number of template checks by up to $9 \times 2 + 28 \times 3$.

5.3.3 Classification

Using the same cleaning process described in the section about Tesseract OCR, but instead of passing the whole license plate crop as one image to the Tesseract OCR program we instead pass the individually cropped character images, one at a time, to our own character classifier. The classifier has been trained on character data automatically generated as described in section 4.2.3 and hand-drawn images produced using a simple tool that lets one draw a 28×28 sized image in black and white. The model was implemented in Keras, the Python library, and follows a variation of a very common structure that is shown in various articles about classifiers in python and Keras. The model implementation can be seen in Figure 61. Using the same test-video as for the YOLOv5 tests, the classifier correctly classified 28% of the characters that were passed to it. Needless to say, with some effort the classifier should surely be able to perform better than this, but these were the results I observed.



```python
def getModel():
    model = Sequential()
    model.add(Conv2D(img_rows, (5, 5), input_shape=input_shape))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=2))

    model.add(Conv2D(64, (5, 5)))
    model.add(Activation('relu'))      You, 8 months ago • first
    model.add(MaxPooling2D(pool_size=(2, 2), strides=2))

    model.add(Flatten())
    model.add(Dense(80))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes))
    model.add(Activation('sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer='rmsprop',
                  metrics=['accuracy'])

    model.summary()
    return model
```

Figure 61: The function that returns the character classifier model made with Keras. Documentation of the Keras library at `https://keras.io/api/`

5.3.4 Character Correction

If our plate does not quite match one of the known formats, but it is close, we might be able to perform a correction. Character correction does not work for any character. For example if a T is detected on the right side of the hyphen (or if no hyphen is detected splitting the plate in the middle can be an option) we can not really assume that it should be a 1 or a 7 because it could be either or. However, if we detect an O (letter) to the right of a hyphen we can with quite good confidence correct that. A zero to the left of a hyphen could be an O, a Q, an Ö or even a G, C or D, so we can not necessarily make the same assumptions vice versa. Depending on the font used in the relevant country, and the character recognition model used, one can utilize more or fewer of these corrections.

Character correction depends on having enough information about the plate. First, we need to know *where* we expect to find *what*. Most Finnish plates have the format AAA-111, that is to say three letters and three numbers that are separated by a hyphen. If we detect a plate that matches this format it is probably best to not change anything. If we detect a plate with very many characters it is most likely not a plate, or it might be a plate with some extra text around it (e.g. some car dealerships put their business' name on the license plate holder).

6 Results and discussion

During the process of training the object detectors for this thesis it became obvious that using the generated data improved the performance of the character detectors. I did not do a quantitative analysis of which ALPR method or model had the best performance. Regardless, the evaluation of the different methods and models seemed to indicate that the best alternative for an ALPR system, when the ALPR scenario is unknown, is the combination of a vehicle and license plate detector combined with a character detector. This conclusion was reached because the vehicle and license plate detector is quite flexible with the data given. The character detectors can be trained, at least partially, on generated data, meaning that the amount of work can be reduced significantly. In contrast, the more conventional computer vision-based methods can yield good results in controlled setups but are very sensitive to changes in the environment and therefore not as suitable for license plate recognition.

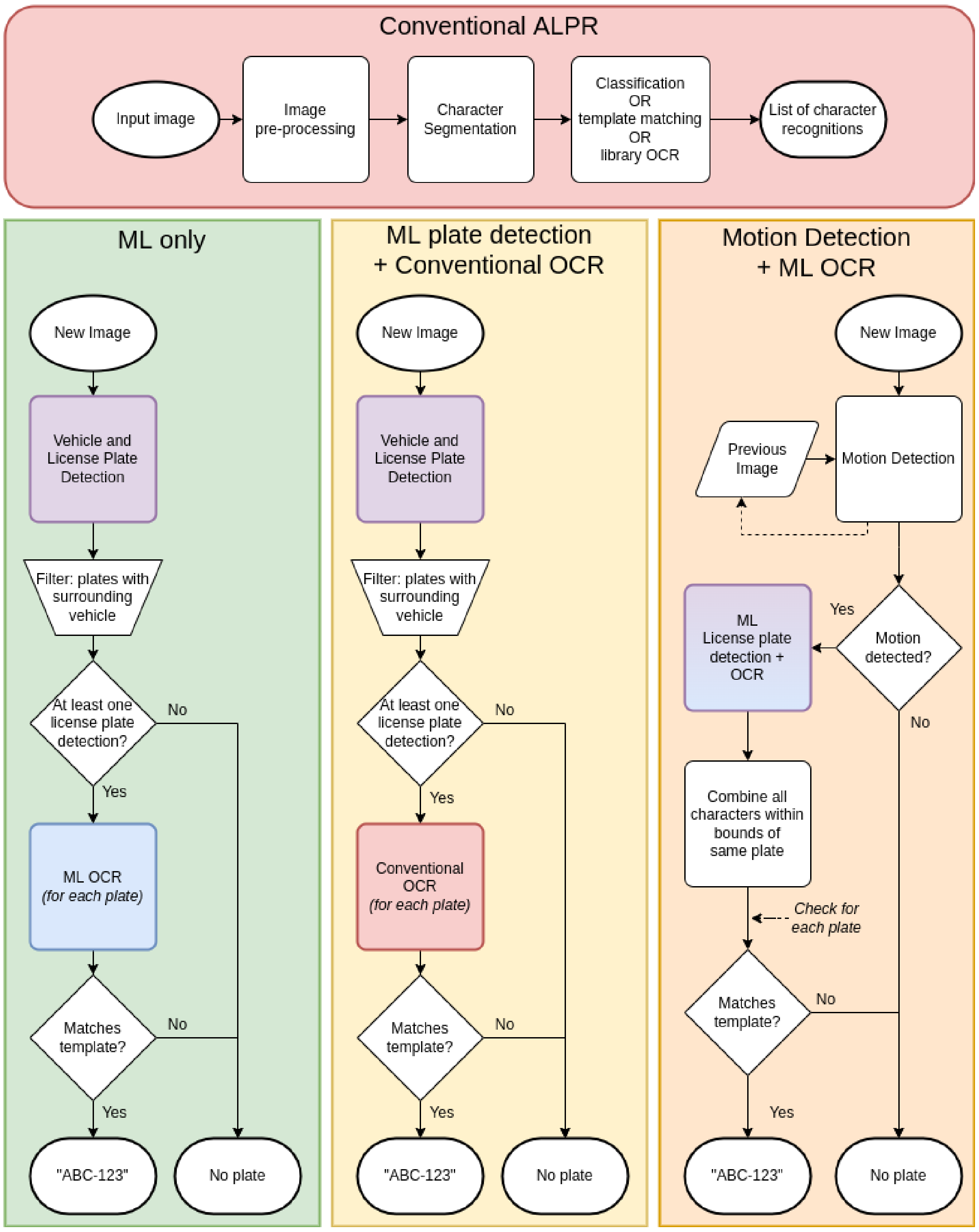
The environment in which an ALPR system will be installed to a large part dictates what sort of solution would constitute the best option. In the following paragraphs I discuss the areas in which I believe the different methods in Figure 62 can be applied.

Conventional ALPR could be used successfully in ALPR systems where the camera can be focused on a very precise area of interest. When this is possible, implementing the conventional methods may require significantly less effort, compared to the machine learning counterparts. In these cases these methods can be reliable, especially if two such methods are run in parallel, although some tweaking might be needed. Using multiple conventional methods could potentially also reduce some of the inherent inflexibility. I would not, however, recommend using only conventional methods-based ALPR in any other scenarios.

ALPR systems, using only machine learning methods (e.g. YOLOv5), can be utilized in any ALPR scenario and should be considered if the developer of the system does not actually have strict control of the installation of the cameras and the setup of the systems themselves or if the conditions and visibility are expected to vary to a large degree. The robustness that the ML-based object detectors provide, granted that they have been trained on data with enough variation, ensures that they will be a good choice regardless of the quality of the camera installation. One important matter to note is that a system based entirely on object detectors could run into issues with detecting license plates of different types or layouts than those on which they have been trained. The character detectors should do fine, as long as they are of similar enough fonts.

Machine learning-based license plate detection combined with conventional OCR can be used in any case where ML-only ALPR systems could be used. However, the conventional OCR methods are, as mentioned, subject to difficulties in varied environments. As such, I would not necessarily recommend this combination for any ALPR case, as it still requires gathering and labeling a license plate data set for the region in which the ALPR-system will be run. That being said, one reason to choose this combination is that the ML-based character detectors require more training than the license plate detector.

Finally, the combination of motion detection and machine learning-based OCR can be employed successfully in ALPR-systems where camera is not moving, but one does not wish to run very powerful object detectors for each camera. To clarify, this combination could be used in instances where a less powerful computer runs the relatively simple process of motion detection, and sends the images to a server that is capable of performing machine learning-based OCR. One significant drawback with this combination is that the machine learning OCR would be run on an image containing not only the license plate, meaning one would have fewer pixels per character. There would also be a large degree of variation in the size of the characters in the images presented to the character detector, compared to a scenario where the character detector receives images cropped exactly to the bounds of the license plates.



Conventional ALPR
Input image
Image pre-processing
Character Segmentation
Classification OR template matching OR library OCR
List of character recognitions
ML only
ML plate detection + Conventional OCR
Motion Detection + ML OCR
New Image
Vehicle and License Plate Detection
Filter: plates with surrounding vehicle
At least one license plate detection?
No
Yes
ML OCR (for each plate)
Conventional OCR (for each plate)
Matches template?
"ABC-123"
No plate
Previous Image
Motion Detection
Motion detected?
ML License plate detection + OCR
Combine all characters within bounds of same plate
Check for each plate

Figure 62: The different arrangements of ALPR evaluated in this thesis.

7 Conclusion

The deep learning methods require large amounts of data to train the model on, which brings us to one of the bigger issues with object detection problems in general, good data sets are difficult to find and labor-intensive to procure. In addition to this, the more flexible one needs the model to be the more variation the data set needs to have, making it even larger and, thus, requiring more manual labor. More variation in the data set also means that the model will need more parameters to learn all the different features, meaning the model will require more compute power to run. License plate data sets are not widely available, since they contain personally identifiable information (unless properly cleaned, the images' metadata may contain timestamps and GPS coordinates, making the data quite sensitive) and those who possess good data sets are not likely to share.

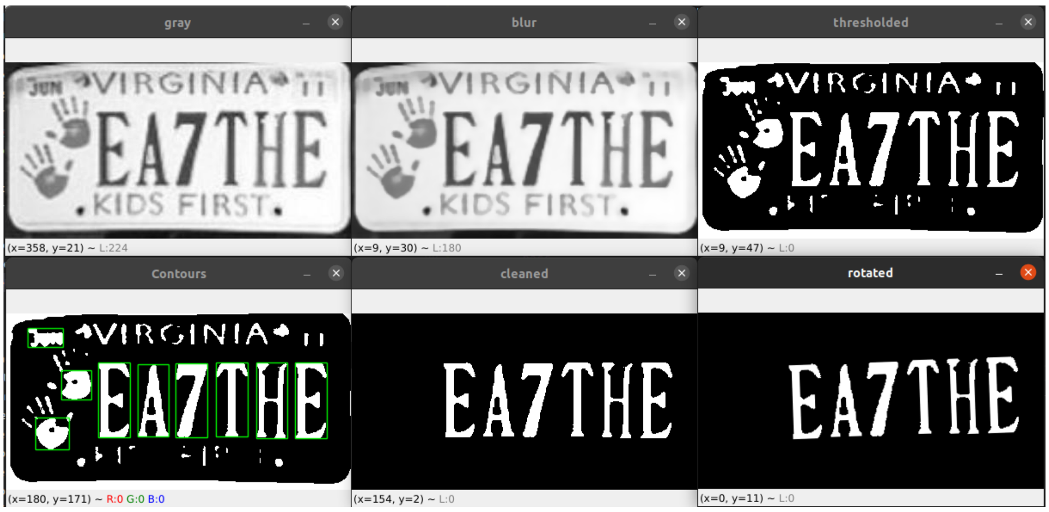


Figure 63: The cleaning process of an image before passing it to Tesseract OCR. The "cleaned" image was the final result, in which Tesseract OCR correctly found the text "EA7THE", however, the "rotated" image was intentionally rotated by 2 degrees and after which Tesseract OCR recognized "Vail" instead.

It should be noted that while license plate data sets are not readily available, pre-trained models capable of vehicle detection are. In addition to this, we have seen that generated data can be used (at least partially) to train license plate character detectors. There are also existing frameworks for text recognition (e.g. Tesseract OCR) but such frameworks can be quite demanding with the quality of the image, the font, skew and perspective etc. As can be seen in Figure 63, Tesseract OCR does not respond well to skew.

In the end, we have learned that it is possible to complement an otherwise lacking license plate data set with generated data, how important it is to make sure that the label boxes are as close to the truth as possible and that it is worth the effort of creating a program to automatically adjust the character labels, even if they may seem perfectly labeled.

7.1 Further Comparisons

The Viola Jones classifier for license plate detection on scaled-down images could potentially be used to rapidly detect license plates before running OCR on the cropped license plate (not scaled down). The drawback of this approach, as discussed, is that the Viola Jones cascading classifier is prone to false positives, leading to excessive time spent by the OCR program trying to identify characters in images that do not contain license plates. Regardless, this could be a viable option that was not tested within the scope of this thesis.

7.2 Further Improvements

Tests of the meta data analyses mentioned, namely the idea of combining previous recognitions based on their locality in the image to complete a license plate that is partially obscured, e.g. by a tow hook or a bright reflection, were regrettably not done in this thesis. These meta data analyses are likely going one of the best ways to improve the performance of an ALPR system that is not always producing consistent recognitions.

As mentioned in section 4.2, the character detector could utilize the fact that the license plates are not filling up the entire 640 $\times$ 640 size image, such that multiple license plates could be processed at once. This would imply half the processing time for detecting characters for an image with two license plates in it. This makes sense, because the object detector will take just as long to process the image regardless of whether only half the image contains a license plate or whether it is entirely covered with license plate images. The limiting factor here is of course whether the license plates will fit into the 640×640 size image without scaling them down. A small amount of scaling could be acceptable in order to increase performance.

Once a license plate has been successfully detected, the ALPR program could skip the license plate detection step for a few frames and simply crop the expected position of the license plate based on the direction and rate of change of the detected characters in the plate.

References

[1] X. Luo, D. Ma, S. Jin, Y. Gong and D. Wang, "Queue Length Estimation for Signalized Intersections Using License Plate Recognition Data," in IEEE Intelligent Transportation Systems Magazine, vol. 11, no. 3, pp. 209-220, Fall 2019, doi: 10.1109/MITS.2019.2919541.

[2] H. Chandra, Michael, K. R. Hadisaputra, H. Santoso and E. Anggadjaja, "Smart Parking Management System: An integration of RFID, ALPR, and WSN," 2017 IEEE 3rd International Conference on Engineering Technologies and Social Sciences (ICETSS), 2017, pp. 1-6, doi: 10.1109/ICETSS.2017.8324174.

[3] Y. Tanaka, "Travel-time data provision system using vehicle license number recognition devices," Proceedings of the Intelligent Vehicles '92 Symposium, 1992, pp. 353-358, doi: 10.1109/IVS.1992.252285.

[4] P. Davies, N. Emmott and N. Ayland, "License plate recognition technology for toll violation enforcement," IEE Colloquium on Image Analysis for Transport Applications, 1990, pp. 7/1-7/5.

[5] ALPR deployment case study by Motorola. Accessed 29th of August 2021. `https://www.motorolasolutions.com/content/dam/msi/docs/global-software/glendale-az-case-study/casestudy_le_glendale_final.pdf`

[6] Motorola/Vigilant Solutions `https://www.motorolasolutions.com/en_us/video-security-access-control/license-plate-recognition-camera-systems.html?utm_source=vigilantsolutions.com&utm_medium=referral&utm_campaign=vigilantsolutions_redirect` Leonardo Company `https://www.leonardocompany-us.com/lpr`

[7] The OpenALPR project's official page. Accessed 29th of August 2021. `https://www.openalpr.com/`

[8] Accessed 29th of August 2021. `https://platerecognizer.com/`

[9] K. Miyamoto, K. Nagano, M. Tamagawa, I. Fujita and M. Yamamoto, "Vehicle license-plate recognition by image analysis," Proceedings IECON '91: 1991 International Conference on Industrial Electronics, Control and Instrumentation, 1991, pp. 1734-1738 vol.3, doi: 10.1109/IECON.1991.239253.

[10] K. Kanayama, Y. Fujikawa, K. Fujimoto and M. Horino, "Development of vehicle-license number recognition system using real-time image processing and its application to travel-time measurement," [1991 Proceedings] 41st IEEE Vehicular Technology Conference, 1991, pp. 798-804, doi: 10.1109/VETEC.1991.140605.

[11] Book by K. P. Murphy, "Machine Learning A Probabilistic Perspective"

[12] McCulloch, W.S., Pitts, W. "A logical calculus of the ideas immanent in nervous activity". Bulletin of Mathematical Biophysics 5, 115–133 (1943). https://doi.org/10.1007/BF02478259

[13] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65 6 (1958): 386-408 .

[14] Gurney, Kevin. "An introduction to Neural Networks" University of Sheffield.

[15] Book by E. Stevens, L. Antiga, T. Viehmann. "Deep Learning with PyTorch"

[16] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[17] Bellman, Richard Ernest Rand Corporation (1957). Dynamic programming. Princeton University Press. p. ix. ISBN 978-0-691-07951-6.

[18] Accessed 1st of September 2021 http://yann.lecun.com/exdb/mnist/

[19] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors", 2012, arXiv:1207.0580

[20] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biol. Cybernetics 36, 193–202 (1980). https://doi.org/10.1007/BF00344251

[21] Hubel, D.H., Wiesel, T.N. : Receptive fields, binocular interaction and functional architecture in cat's visual cortex. J. Physiol. (London) 160, 106-154 (1962)

[22] M. Lin, Q. Chen and S. Yan, "Network In Network", 2013, arXiv:1312.4400

[23] L. Lu, Y. Shin, Y. Su and G. Karniadakis, "Dying ReLU and Initialization: Theory and Numerical Examples", 2019, arXiv:1903.06733

[24] N. Ma, X. Zhang and J. Sun, "Funnel Activation for Visual Recognition", 2020, arXiv:2007.11824

[25] Dechter, Rina. (1986). Learning While Searching in Constraint-Satisfaction-Problems.. AAAI. 178-185.

[26] Book by Igor Aizenberg, Joos P.L. Vandewalle, and Naum N. Aizenberg, "Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications"

[27] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection", 2015, arXiv:1506.02640

[28] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger", 2016, arXiv:1612.08242

[29] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement", 2018, arXiv:1804.02767

[30] A. Bochkovskiy, C. Wang and H. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection", 2020, arXiv:2004.10943

[31] Accessed 2nd of September 2021 `https://github.com/ultralytics/yolov5/wiki/Tips-for-Best-Training-Results`

[32] Z. Ge, S. Liu, F. Wang, Z. Li and J. Sun, "YOLOX: Exceeding YOLO Series in 2021", 2021, arXiv:2107.08430

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu and A. C. Berg "SSD: Single Shot MultiBox Detector", 2016, arXiv:1512.02325

[34] T. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal Loss for Dense Object Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 318-327, 1 Feb. 2020, doi: 10.1109/TPAMI.2018.2858826.

[35] K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 386-397, 1 Feb. 2020, doi: 10.1109/TPAMI.2018.2844175.

[36] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", 2015, arXiv:1512.03385

[37] G. Huang, Z. Liu, L. Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks", 2018, arXiv:1608.06993

[38] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature Pyramid Networks for Object Detection", 2016, arXiv:1612.03144

[39] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", 2014, arXiv:1409.1556

[40] R. K. Srivastava, K. Greff and J. Schmidhuber, "Training Very Deep Networks", 2015, arXiv:1507.06228

[41] Shimodaira, Hidetoshi. (2000). Improving predictive inference under covariate shift by weighting the log-likelihood function. Journal of Statistical Planning and Inference. 90. 227-244. 10.1016/S0378-3758(00)00115-4.

[42] Sergey Ioffe and Christian Szegedy. 2015. "Batch normalization: accelerating deep network training by reducing internal covariate shift". In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 448–456.

[43] G. Huang, Y. Sun, Z. Liu, D. Sedra and K. Weinberger, "Deep Networks with Stochastic Depth", 2016, arXiv:1603.09382

[44] The CIFAR-10 data set. Accessed 16th of October 2021 `https://www.cs.toronto.edu/~kriz/cifar.html`

[45] C. Wang, H.M. Liao, I. Yeh, Y. Wu, P. Chen and J. Hsieh, "CSPNet: A New Backbone that can Enhance Learning Capability of CNN", 2019, arXiv:1911.11929

[46] K. He, X. Zhang, S. Ren and J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition", 2014, arXiv:1406.4729

[47] A. S. Palatnick and H. R. Inhelder, "Automatic vehicle identification systems - Methods of approach," 20th IEEE Vehicular Technology Conference, 1969, pp. 30-31, doi: 10.1109/VTC.1969.1621966.

[48] R. S. Foote, "Automatic vehicle identification: Tests and applications in the late 1970's," in IEEE Transactions on Vehicular Technology, vol. 29, no. 2, pp. 226-229, May 1980, doi: 10.1109/T-VT.1980.23844.

[49] The history of ANPR by ANPR international. Accessed 2nd of May 2020. `http://www.anpr-international.com/history-of-anpr/`

[50] A. S. Johnson and B. M. Bird, "Number-plate matching for automatic vehicle identification," IEE Colloquium on Electronic Images and Image Processing in Security and Forensic Science, 1990, pp. 4/1-4/8.

[51] The OpenCV page. Accessed April 3rd 2021. www.opencv.org

[52] The licenses used by OpenCV Accessed April 3rd 2021. `https://opencv.org/license/`

[53] The Tesseract OCR github page. Accessed 25th of September 2020. `https://github.com/tesseract-ocr/tesseract`

[54] A description of the levenshtein algorithm. `http://levenshtein.net/`

[55] T. Wang, R. M. Anwer, H. Cholakkal, F. S. Khan, Y. Pang and L. Shao, "Learning Rich Features at High-Speed for Single-Shot Object Detection," 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 1971-1980, doi: 10.1109/ICCV.2019.00206.

[56] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks", 2014, arXiv:1312.6229

[57] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[58] R. Girshick, "Fast R-CNN", 2015, arXiv:1504.08083

[59] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, 1 June 2017, doi: 10.1109/TPAMI.2016.2577031.

[60] J. Dai, Y. Li, and J. Sun, "R-FCN: Object Detection via Region-based Fully Convolutional Networks", 2016, arXiv:1605.06409

[61] P.Viola and M.Jones: Rapid Object Detection using a Boosted Cascade of Simple Features Accessed 10th of September 2021 https://www.cs.cmu.edu/ efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

[62] L. Cuimei, Q. Zhiliang, J. Nan and W. Jianhua, "Human face detection algorithm via Haar cascade classifier combined with three additional classifiers," 2017 13th IEEE International Conference on Electronic Measurement & Instruments (ICEMI), 2017, pp. 483-487, doi: 10.1109/ICEMI.2017.8265863.

[63] B. Tej Chinimilli, A. T., A. Kotturi, V. Reddy Kaipu and J. Varma Mandapati, "Face Recognition based Attendance System using Haar Cascade and Local Binary Pattern Histogram Algorithm," 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184), 2020, pp. 701-704, doi: 10.1109/ICOEI48184.2020.9143046.

[64] Kuan Zheng, Yuanxing Zhao, Jing Gu and Qingmao Hu, "License plate detection using Haar-like features and histogram of oriented gradients," 2012 IEEE International Symposium on Industrial Electronics, 2012, pp. 1502-1505, doi: 10.1109/ISIE.2012.6237313.

[65] S. Liang, M. Shridhar and M. Ahmadi, "Efficient algorithms for segmentation and recognition of printed characters in document processing," Proceedings of IEEE Pacific Rim Conference on Communications Computers and Signal Processing, 1993, pp. 240-243 vol.1, doi: 10.1109/PACRIM.1993.407179.

[66] Harris Corner detection with OpenCV Accessed 15th of September 2021 `https://docs.opencv.org/master/dc/d0d/tutorial_py_features_harris.html`

[67] OpenCV template matching usage example, showing all comparison methods. Accessed 7th of May 2021. `https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html`

[68] Generalized Intersection Over Union Accessed 14th of September 2021 https://arxiv.org/pdf/1902.09630.pdf

[69] The PASCAL VOC 2007 data set. Accessed October 8th 2021. `https://paperswithcode.com/dataset/pascal-voc-2007`

[70] T. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. Zitnick and P. Dollár, "Microsoft COCO: Common Objects in Context", 2014, arXiv:1405.0312 And the website: `https://cocodataset.org/` Accessed October 8th 2021.

[71] The YOLOv1 model config file. Accessed October 9th 2021. `https://github.com/pjreddie/darknet/blob/master/cfg/yolov1.cfg`

[72] Sadeghi M.A., Forsyth D. (2014) 30Hz Object Detection with DPM V5. In: Fleet D., Pajdla T., Schiele B., Tuytelaars T. (eds) Computer Vision – ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol 8689. Springer, Cham. `https://doi.org/10.1007/978-3-319-10590-1_5`

[73] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network", 2016, arXiv:1609.05158

[74] T. Ridnik, H. Lawen, A. Noy, E. B. Baruch, G. Sharir and I. Friedman, "TResNet: High Performance GPU-Dedicated Architecture", 2020, arXiv:2003.13630

[75] PyTorch documentation. `https://pytorch.org/docs/stable/index.html`

[76] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149), 1999, pp. 246-252 Vol. 2, doi: 10.1109/CVPR.1999.784637.

[77] The image data set containing over 500 images of the rear views of various vehicles (cars, trucks, busses), taken under various lighting conditions (sunny, cloudy, rainy, twilight, night light). Croatian license plates. Accessed 16th of August 2021 http://www.zemris.fer.hr/projects/LicensePlates/english/results.shtml

[78] The license plate generator program created by the author of this thesis. `https://www.github.com/joelcma/plate-generator`

[79] Accessed 1st of September 2021 https://github.com/puzzledqs/BBox-Label-Tool

[80] The application page for special Finnish license plates (the form on the page contains the rules). Accessed 13th of September 2021 https://www.traficom.fi/fi/asioi-kanssamme/hae-ajoneuvolle-erityiskilpea

Svensk sammanfattning

En studie om automatisk nummerplåtsigenkänning

Introduktion

Automatisk nummerplåtsigenkänning, ALPR [2], är en automatisk process som går ut på att extrahera och känna igen tecknen i en bild som innehåller nummerplåtar.

Målet med avhandlingen är att beskriva hur man kan implementera de olika delsystem som behövs för ALPR och att kvalitativt beskriva hur moderna respektive äldre tekniker presterar och hur de skiljer sig från varandra.

År 1969 ordnades en konferens om fordonsigenkänning, och redan då nämndes automatisk nummerplåtsigenkänning. Sedan dess har forskningen gått framåt och idag finns ett stort urval av metoder att välja mellan. I min avhandling skriver jag om hur utvecklingen sett ut och om de system för nummerplåtsigenkänning vi har idag. Jag har valt att fokusera på YOLO-algoritmen och jämföra den med konventionella metoder.

YOLO-algoritmen

För att täcka den teoretiska bakgrund som behövs för att förstå hur YOLO-algoritmen implementeras tar jag i min avhandling avstamp i teori om maskininlärning och neurala nätverk. Jag beskriver hur respektive uppstått och utvecklats, och går stegvis igenom hur utvecklingen lett till det nyaste vi ser och använder inom datorsyn (computer vision) idag, för att slutligen komma fram till objektsdetekteringsalgoritmen YOLO. Eftersom YOLO-algoritmen är en snabb och modern objektsdetekteringsalgoritm lämpar den sig väl för objektsdetektering i realtid. En del andra algoritmer kan producera detektioner med högre träffsäkerhet än YOLO-algoritmen, men de gör det ofta i en långsammare takt och medför därför inte samma potential för statistisk meta-analys av detektionerna.

[2] ALPR står för Automatic License Plate Recognition.

Detta är en av YOLO-algoritmens fördelar i jämförelse med andra alternativ, och samtidigt även orsaken till att jag valt att i min avhandling fokusera på just YOLO-algoritmen. Då YOLO-algoritmen används blir det möjligt att göra en statistisk meta-analys av detektionerna, vilket i praktiken innebär att man jämför alla detektioner och analyserar vilka tecken som mest sannolikt förekommer i nummerplåten och i vilken ordning. På detta vis höjs reliabiliteten, och det går att med större säkerhet lita på resultatet än om man bara hade litat på en eller ett fåtal detektioner. YOLO-algoritmen är i jämförelse med andra alternativ en så pass snabb objektsdetektor att det kompenserar för den något lägre träffsäkerhetsnivån som YOLO medför jämfört med andra moderna objektsdetektorer.

Teoretisk bakgrund och centrala begrepp

Klassificering, igenkänning, lokalisering och objektsdetektering är centrala begrepp inom nummerplåtsigenkänning.

Klassificering är att baserat på en datapunkts egenskaper tilldela den en klass. Ett datorsynsexempel på klassificering är att ett datorprogram konstaterar att en bild föreställer en katt. Med igenkänning avses processen att känna igen ett specifikt exemplar inom en klass, dvs. att katten i fråga konstateras vara den specifika katten Maui och inte vilken katt som helst. Lokalisering är den process som gör det möjligt att upptäcka någonting i en bild och att kunna definiera var i bilden objektet i fråga befinner sig, men utan att definiera vilken klass objektet tillhör.

Objektsdetektering är i princip en kombination av klassificering och lokalisering, dvs. att känna igen vad och var objekten i en bild är. Moderna maskininlärningsbaserade objektsdetektorer gör detektioner på en gång, medan mer primitiva metoder skapar implicita detektioner genom att kombinera separata steg av lokalisering och klassificering.

Innan maskininlärning med djupa neurala nätverk blev genomförbart i realtid på lättillgänglig hårdvara använde man sig av mer primitiva datorsynsbaserade metoder. Dessa metoder baserar sig på slutsatser som en människa kan dra då hon tittar på en bild. Exempel på sådana slutsatser är att nummerplåtar har 4 hörn och är rektangulära, de har oftast vit bakgrund (åtminstone i Finland) och tecknen är svarta. Jag tar i min avhandling upp dylika metoder och beskriver hur de fungerar. Man kan kombinera flera sådana metoder för att känna igen nummerplåtar och man kan använda dem för att förbereda bilder för att lättare producera bra nummerplåtsigenkänningar.

Metodologi

Jag tränade och testade YOLO-modeller [3] med hjälp av olika dataset. I huvudsak tränades två sorters modeller; den första sorten känner igen bilar och nummerplåtar i bilder föreställande trafiksituationer. Den andra känner igen i bilder på nummerplåtar. De två olika sorternas modeller behövdes eftersom objektsdetektorn, YOLOv5, använder bilder med upplösningen 640×640 pixlar, vilket inte räcker för att känna igen tecknen på en nummerplåt på flera meters avstånd från kameran. Den första modellen gavs alltså bilder som urpsrungligen var 1920×1080 pixlar stora omskalade till 640×640, vilket duger för att känna igen bilar och nummerplåtar. Efter att ha detekterat en nummerplåt inom samma ramar som man detekterat en bil konverteras koordinaterna till den ursprungliga bildens koordinatsystem. Nummerplåten klipps sedan ut i full upplösning och skickas vidare till teckendetektorn.

Nedan beskriver jag de dataset (samlingar av bilder) som användes för att skapa de dataset (samlingar av bilder och motsvarande annoteringar [4]) som jag sedan använde för att träna YOLO-modellerna som testades.

För det första datasetet annoterade jag bilar och nummerplåtar. Det första datasetet beskriver alltså var det finns bilar och nummerplåtar i bilderna.

För det andra datasetet klippte ett program automatiskt ut nummerplåtarna enligt annoteringarna i det första datasetet. Efter att programmet klippt ur nummerplåtarna annoterade jag manuellt de enskilda tecknen.

[3]YOLO-algoritmen används för att träna djupa neurala nätverk. Då man använder olika dataset för att träna ett neuralt nätverk uppstår olika modeller, även om strukturen på själva nätverket är densamma.

[4]med annotering menar jag här att jag markerat objektets position och dess klass i en separat fil som YOLO-algoritmen sedan läser under träningsprocessen.

Jag använde mig av ett eget finskt dataset och ett kroatiskt dataset för att skapa dataseten och sedan träna YOLO-modellerna. Det första datasetet med annoterade bilar och nummerplåtar användes för att träna bil- och nummerplåtsdetektorer medan det andra datasetet med annoterade tecken användes för att träna teckendetektorer.

Ursprungligen använde jag mig av det kroatiska datasetet eftersom det var tillgängligt utan licens för studerande. Jag märkte att variationen i tecknen på de kroatiska nummerplåtarna dessvärre inte är så stor och att upplägget på plåtarna i sig inte är detsamma som på finländska nummerplåtar. På grund av detta valde jag att skapa ett eget finskt dataset.

Variationen i dataseten var fortfarande liten och därför skapade jag ett program som genererar annoterade bilder på nummerplåtar och dessutom förvränger bilderna på olika sätt för att skapa mer variation. Nummerplåtsdetektorerna presterade tillräckligt bra utan genererade data, men sådana data behövdes för att träna teckendetektorerna. Kombinationen av det kroatiska, det finska och mitt eget genererade dataset fungerade bra för att träna teckendetektorerna till applicerbar träffsäkerhet.

Under träningen av modellerna lade jag även märke till att det ibland uppstod stor variation i resultatet, dvs ibland presterade modellen bra och ibland sämre. Vid närmare efterforskning märkte jag att modellen presterade enligt förväntan trots att de finska och det kroatiska datapunkterna i dataseten var de samma, men bara de genererade nummerplåtarna var olika från test till test. Min slutsats var att annoteringarna av tecknen i de genererade nummerplåtarna inte längre motsvarade tecknens egentliga positioner i de förvrängda bilderna. På grund av denna variation klarade inte YOLO-algoritmen av att träna modellerna till lika hög precision som den kunde då annoteringarna var mer korrekta.

Jag åtgärdade detta genom att skapa ett program som automatiskt korrigerar annoteringarna genom att processera de genererade bilderna. Detta program användes sedan för att förbättra annoteringarna i såväl de manuellt annoterade dataseten som de automatiskt genererade dataseten.

För att sedan jämföra skillnaden i dataset med bra och dåliga annoteringar jämfördes ett dataset med korrigerade annoteringar och en kopia av det där annoteringarna justerades godtyckligt, men till en minimal grad, för att simulera manuella annoteringar. Skillnaden var tillräckligt stor för att vara värt den investerade tiden eftersom resultaten blev bättre.

Resultat

Efter att ha tränat YOLO-modellerna och testat konstaterade jag att YOLO-modellerna är en bra metod för nummerplåtsdetektering.

De mer konventionella, datorsynsbaserade metoderna kan ge goda resultat för ALPR om omgivningen, eller indata, inte har stora förändringar. De medför dock aldrig samma flexibilitet som en objektsdetektor baserad på neurala nätverk, t.ex. YOLOv5, gör. Eftersom neurala nätverk imiterar den biologiska hjärnan är de på samma sätt flexibla med hur de tolkar data. Konventionella metoder kräver däremot ofta mycket justering och petande med detaljer för att uppnå resultat, och påverkas därför mer av t.ex. varierande väderförhållanden.

Diskussion

Trots att jag skapade ett eget finskt dataset visade det sig att inte heller det datasetet hade tillräckligt stor variation i de tecken som ingick. Detta berodde på att bilderna togs från insidan av en bil i rörelse, och samma nummerplåtar har därför fastnat på flera bilder. Eftersom vissa tecken förekom oftare än andra blev modellerna mer benägna att föreslå dem än andra. I min avhandling kunde jag motverka denna utmaning genom att skapa ett genererat dataset.

Om någon annan skulle göra ett motsvarande arbete, vill jag alltså påpeka att det är möjligt att komplettera ett annars otillräckligt dataset med genererad data eftersom det sparar tid och resurser jämfört med att samla in och annotera ett tillräckligt stort och varierande dataset. Oberoende av vilken typ av dataset man använder sig av är det viktigt att noggrant kontrollera att annoteringarna är så goda som möjligt.