MASTER'S THESIS | LUND UNIVERSITY 2018

Vehicle Counting using Video Metadata

Mattias Gustafsson, Sebastian Hjelm

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-13

Vehicle Counting using Video Metadata

(Looking beyond the image)

Mattias Gustafsson, Sebastian Hjelm
dat13mgu@student.lu.se dat13shj@student.lu.se

August 20, 2018

Master's thesis work carried out at Axis communications AB.

Supervisors: Yuan Song, yuan.song@axis.com
Jörn Janneck, jorn.janneck@cs.lth.se

Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

Abstract

The current field of object detection and image recognition is huge but not without complications. Processing large amounts of high resolution videos needs powerful hardware and also risks breaching the privacy of those who are recorded. In times of increasing demand for decentralized solutions and stricter privacy protection regulations being put in place a new approach is needed.

We present an alternative to traditional object detection in video where we analyze changes to its metadata over time rather than the content of the video frames. This approach has several benefits over traditional object detection: it is incredibly fast, lightweight and protects the privacy of its subjects.

We have trained and evaluated several neural network models tasked with detecting and counting vehicles in various scenes and have achieved accuracies above 90%. Finally, we take the first steps toward a decentralized solution running entirely on embedded devices.

Keywords: Machine learning, Neural networks, Vehicle detection, CNN, Metadata, Bitrate, QP, Fast, Privacy

Acknowledgements

We would like to thank our supervisors Yuan Song and Jörn Janneck for their valuable feedback and help throughout the thesis project. Serving as our third supervisor we also want to thank Anton Jakobsson for his feedback and help with TensorFlow and machine learning.

Additionally we would like to thank Mikael Pendse, Mattias Petterson, Peter Wilhelmsson, Malte Johansson and others at Axis Communications for their technical support and expertise. Extra thanks to Gustaf Oxenstierna and Johan Rodhe who peer reviewed an early draft of our report and to Mattias Ohlsson for taking time to discuss the possible use of confidence intervals.

Lastly we also want to thank our examiner Falvius Gruian and our opponents Gustaf Oxenstierna, Emir Husic and Aleksandar Simeunovic for their feedback.

Contents

Chapter 1
Introduction

The availability and utility of machine learning and data analysis has made it increasingly more common to extract information from all kinds of data, to the point where it is done everywhere. In the field of object detection in images we see the rise of self-driving cars, more extensive object tracking and face recognition. Already there exist several powerful neural networks designed for detecting objects, such as YOLO [37] and Faster-RCNN [39]. It is very much possible to locate arbitrary objects with high accuracy and at relatively high speeds, given enough training.

However, despite being powerful, today's neural networks are simply too large and complex to run on smaller, embedded systems or too slow to keep up with real time video. In order to process video of increasingly high resolution, in real time, a new approach is required. In this thesis we have found one such approach that solves this problem.

We have focused on the task of counting vehicles in video using machine learning and neural networks. Unlike traditional systems our networks differ in that they do not actually look at the video image. Instead we look at the video's so called *metadata*, information about how it was encoded. Specifically we track each frame's *bitrate* and *QP-values* (discussed in detail in Section 3.2). By looking at the changes over time our networks can estimate how many vehicles have passed.

Depending on the model pipeline and network structure this method can be both very small in size and extremely fast. This is because the metadata is much smaller, more compact and has much fewer features compared to raw video, which drastically reduces complexity. The downside is that with less information available it is not as accurate as dedicated state-of-the-art image networks, particularly because there exist no pre-annotated datasets to train the networks on. Our goal was to examine this approach and evaluate how well it works in practice; both how small the models can be and how well they work.

We have performed this thesis project at Axis Communications AB, a world leading company specialized in network surveillance cameras. In order to present vehicle counting as a feature in their cameras, Axis would currently be required to upload the video into the cloud to do the object detection. This does not only result in an extra cost but also presents a potential privacy issue as data is sent from the cameras to third party servers. By only using the metadata values our models make the videos anonymized, keeping the subject's privacy intact. Finally, by deploying our networks on Axis' surveillance cameras and automating the training of the networks they can be tailored to each particular scene where they are installed, improving their accuracy compared to a single global model.

Chapter 2

Background

This chapter is aimed at providing a basic level of knowledge of the different types of neural networks that we have used in our thesis project. The following sections start by explaining the fundamental concepts in brief, followed by some sections on various architectures. Finally we devote a section to network training.

2.1 A brief introduction to neural networks

Artificial Neural Networks (ANN) have been around for a long time. You can trace the original idea to the McCulloch-Pitts model in 1943 [27]. Since their conception neural networks have provided a mathematical model that tries to mimic the structure of biological brains. Each network consists of a number of nodes (or neurons) in some structure, usually in layers, with connections to other nodes. See Figure 2.1 for an illustration.

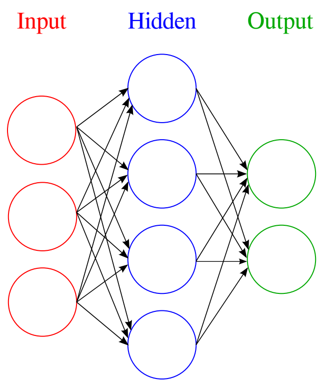


Figure 2.1: A simple ANN with three layers. The first layer represents the input variables, the second layer performs internal (hidden) computations and the final layer represents output nodes.

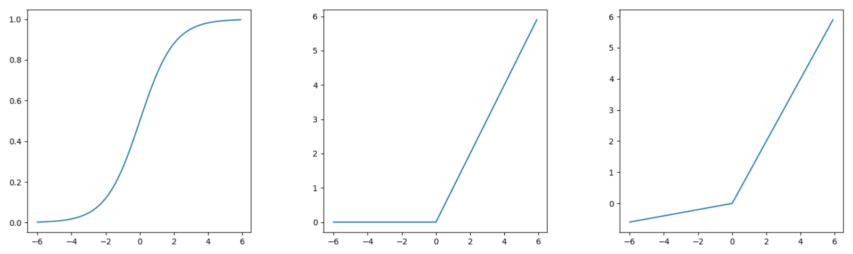


Figure 2.2: Some common activation functions: logistic sigmoid (left), ReLU (middle) and leaky ReLU (right).

Each node in the network performs a simple calculation using some of its neighbors and propagates its result to other nodes. This computation usually consists of a linear combination of its inputs using a set of weights followed by the application of an activation function. The following equation describes this procedure:

$$y_j = \phi\left(\sum_{i=0}^{N} x_i\theta_{ij}\right) \tag{2.1}$$

where x_i are the inputs, θ_{ij} are the weights, $\phi(\cdot)$ is the activation function and y_j is the output of the jth node. There is usually constant terms (biases) in the sum as well, using our notation we have included them by defining $x_0 = 1$ and letting θ_{0j} represent the biases.

The activation function is chosen when building the network and may be more or less arbitrary. There are two main constraints; it must be non-linear, at least for the hidden layers, and it must be differentiable. In Figure 2.2 you can see three common activation functions. The first is the logistic sigmoid which is commonly used for classification, the second is called the rectified linear units (ReLU) [31] which has proven useful as a way to deal with vanishing gradients, and the third one is called leaky ReLU which extends the ReLU to have non-zero output for negative inputs.

The weights determine how the network behaves and to give them reasonable values a training procedure is usually applied. We will discuss this more in Section 2.3.1.

When the node output is propagated to other nodes they will use it as input to compute their own result and so on. To make this useful you define some nodes as inputs to the network and some nodes as outputs; by feeding in a set of inputs you will eventually get a set of outputs after propagating it through the network.

During the time between 1943 and 2018 there have been several periods were the interest in neural networks has peeked. In 2012 the paper on AlexNet [21] was published which marked the start of the current boom. During all this time many new algorithms have been invented and the computer hardware has improved significantly, making the use of neural networks feasible. Since our goal is to use neural networks as a machine learning tool the following sections will focus on that.

2.2 Neural network types

There are a multitude of different types of neural networks that you can use depending on what type of problem you are trying to solve. We have selected a small subset of these during our thesis project and the following sections aim at explaining what they are and how they work.

When we talk about different types of networks what we really mean is networks with different structures and interpretations. In most cases the basic building blocks are still the same as we discussed in Section 2.1.

2.2.1 Multi-layer Perceptron

One of the most basic types of neural networks is the Multi-layer Perceptron (MLP) [24]. In MLPs all nodes are split into layers of varying size and every node in each layer is connected to every node in the next layer; this is a dense network structure. An example of this type of network can be seen in Figure 2.1. When evaluated the input is fed into the first layer whereafter the result is propagated layer by layer until the end is reached.

The MLP is a simple type of feed-forward neural network. There are a lot of specialized types of networks with other structures (see the succeeding sections). This basic structure is still commonly used, both by itself and in conjunction with other techniques.

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are also feed-forward networks like the MLPs but designed in a different way. They assume that the data is some sort of signal, like an image (in 2D) or a sensor reading over time (in 1D), thus containing local structures that can be detected. The overarching idea is to use many layers, each containing filters, which will gradually build a more abstract understanding of the input [22]. For example, given an input image and the task to detect a car the network will detect simple shapes in the first layer, like edges, while the succeeding layers use the edges to detect more complex shapes, like rectangles or circles. The complexity of shapes gradually increases through the layers until a single filter could principally become a car detector.

In order to detect features in each layer convolution filters are used. The idea is to define a small matrix with scalars and sweep it over the input, calculating the scalar product at each position. Whenever the input contains a feature that corresponds to what the filter detects its output will have a large value at that position. To make this more powerful you define multiple filters for each layer, thus generating a series of feature maps that can be used as the input for the next layer.

The filter sweep allows for local structures to appear anywhere in the input signal and still being detected by the same filter. For instance, consider the problem were the input is an image that contains a dog and we would like to determine where it is located. A filter in a CNN can detect the dog anywhere since it is swept over the whole input signal. By contrast, if you instead used an MLP to perform the same task you would essentially need one node for each image position to indicate whether or not it contains a dog. This causes the knowledge of what a dog is to be duplicated across the weights of all nodes. In practice this means that you would need a lot more training data, with dogs in all possible positions, or the accuracy of the model might suffer.

In order for the CNN to get a high level understanding of the input many layers of convolutions may be needed. For fully connected networks a large number of layers could become a problem since you get a huge number of weights between each pair of layers,

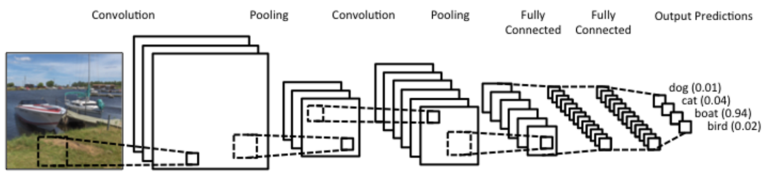


Figure 2.3: An example of a CNN architecture. Note its structure with several convolution-pooling pairs followed by dense layers. The figure was made by Long [26].

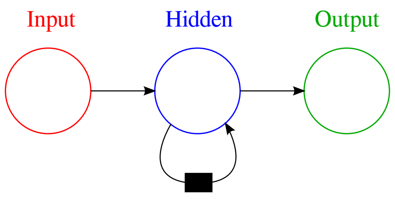


Figure 2.4: A simple RNN with a single hidden node. The output depends on both the input and on the previous state of the hidden node.

however thanks to the CNN structure with the filters the model has much fewer weights. The reason for this is that each filter uses the same weights for the whole input space which results in a lot of weight sharing.

CNNs are usually split into two different types of layers called Convolutional layers and Maxpooling layers. Both use the filter-structure we have described above but in different ways. See Figure 2.3 for an illustration of how a CNN network might be constructed using these layers. Also note that the figure contains fully connected layers at the end, this is commonly added to compose all the features that the CNN has derived.

The Convolutional layers have a set of filters with unique weights, as we discussed above, but also includes activation functions that are computed after summing the results of the filter application.

The Maxpooling layers, on the other hand, consist of only a single filter and uses no weights at all. Instead it finds the maximum value in the input each time it is applied. Usually maxpooling layers have a stride set to the size of the filter which means that every input will only be processed once. Maxpooling is used to strongly reduce the size of the data, however information is also lost, particularly the location of features. If the task is to classify images maxpooling can be a good way to reduce the complexity, however if the task is to locate objects in images it may instead ruin the precision.

2.2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a more general type of neural networks than the feed-forward networks we have discussed previously. The RNNs allow for feed-back connections as well as feed-forward connections, i.e. there may be circular dependencies between nodes in the network. See Figure 2.4 for a simple example.

This type of network is often used when the input is a sequence of data. Since the output of the hidden node depends both on the input and on its previous state it can be viewed as a simple memory. The network is updated in discrete time steps, each step you introduce the next input from the sequence and calculate a new hidden state [22]. It is easy to realize that this procedure can be performed for sequences of arbitrary length which is one of the big advantages that RNNs have over feed-forward networks.

There are many different types of RNNs that each have their advantages and disadvantages. In our thesis we have focused on using the LSTM structure.

Long Short Term Memory

The Long Short Term Memory (LSTM) is not a type of RNN but rather an improvement to existing network structures. The simple RNN model in Figure 2.4 has well documented problems that makes it very hard to train for long input sequences when it uses the logistic sigmoid or the hyperbolic tangent as its activation function [14][4]. The weight updates have an exponential dependency on the length of the input sequence which causes them to either vanish or explode.

Thanks to the structure and mathematical properties of the LSTM the issue with vanishing gradients can be solved and the likeliness of exploding gradients can be decreased [15]. To use the LSTM you simply replace the hidden nodes with the much more complex LSTM node, thus this can be done for both simple and complex networks.

Another much more recent approach to solve the same problem is to use Gated Recurrent Units (GRU). The GRU was devised by Cho et al. [7] and is based on the same idea as the LSTM but with the goal of being simpler and easier to implement. The performance of both these types of units is roughly the same if you compare models with the same amount of parameters [8].

2.3 Training neural networks

When constructing a neural network model the desired end result is that it should perform some operation on your data. However, the building blocks of the model itself do not contain any information about your particular task. In order to make it do what you want you will need to assign its weights to appropriate values. Since neural networks in general are very complex and non-convex models there exists no closed form solution for the optimal weights. Instead a learning procedure is applied to derive an approximation of the their values iteratively.

2.3.1 The backpropagation algorithm

The most common procedure to do this derivation is to use the backpropagation algorithm [22] together with some version of gradient descent. This is done by first defining a loss function which quantifies how far off the network is from the expected output. By computing the derivative of the loss function with respect to every single weight in the model we can determine in which direction we need to change the weight values to minimize the loss.

This procedure might seem very difficult at first but it turns out that it is quite simple. By exploiting the layered structure of the model it is possible to compute the derivatives using only the output from the previous layer and the derivatives from the next. To do this we compute two sets of derivatives for each layer, firstly with respect to the weights and secondly with respect to the output from the previous layer. When the expressions for the derivatives have been derived we can just stack them on top of one another to match the shape of the network we are using. It is even possible to use this approach for RNNs by unfolding them in time [22].

To use the backpropagation algorithm we first propagate the input through the network to get the outputs of each layer and to compute the loss. We then compute the derivatives layer by layer, starting with the last one, backpropagating the loss towards the first layer. When the derivatives are computed we ask an optimizer to decide how they should be used to update the weights.

In order for the backpropagation algorithm to work effectively it is important to choose the correct types of loss function and optimizer. The following sections will discuss this briefly.

Choice of loss function

Perhaps the most important part of the backpropagation algorithm is the loss function. If chosen correctly it will cause the model to converge faster and get a better result during training. How the function is defined depends very much on the model and the problem you are solving. Commonly the mean squared error is used for regression tasks and the cross entropy error is used for classification tasks, but in theory it could be defined arbitrarily as long as it is a good, continuous estimate of the model error.

The mean squared error was the most important loss function in our thesis project, it is defined as follows:

$$L(\theta) = \frac{1}{N}\sum_{n=1}^{N}(y(n,\theta) - t(n))^2 \tag{2.2}$$

where θ are the weights, $y(n,\theta)$ is the network output, $t(n)$ is the target value for input sample n and N is the total amount of samples. Minimizing this type of loss is intuitive since it penalizes outputs that diverge a lot from the correct values, while still allowing for some leeway close to the targets.

The cross entropy error for binary classification was also used and is defined as:

$$L(\theta) = -\sum_{n=1}^{N}\Big(t(n)\log y(n,\theta) + (1 - t(n))\log(1 - y(n,\theta))\Big) \tag{2.3}$$

where $\log(\cdot)$ is the base 2 logarithm, θ are the weights, $y(n,\theta) \in [0,1]$ is the network output, $t(n) \in [0,1]$ is the target value for input sample n and N is the total amount of samples. This function heavily penalizes misclassification and is more apt to result in quick convergence for classification tasks compared to using the mean squared error.

The difference is illustrated by the following example. Let $y(0) = 0.95$ and $t(0) = 0$. Now the cross entropy loss is $L_{cross} = 4.32$ while the squared loss is $L_{square} = 0.9$. Furthermore we can see that $L_{cross} \to \infty$ as $y(0) \to 1$, while $L_{square} \to 1$. Clearly the cross entropy error penalizes the classification errors more than the mean squared error.

Choice of optimizer

After computing the loss and all the gradients of the weights they need to be applied in some sort of update step. There are many methods to do this, ranging from simple Stochastic Gradient Descent (SGD), to RMSprop [13], Adam [19] and AdaGrad [10]. The different methods experiment with a variety of heuristics to achieve a good convergence speed. In this thesis we have used the Adam optimizer so this section will focus on that method.

The Adam optimizer tries to combine the benefits of both RMSprop and AdaGrad while also having small memory requirements. The method tries to estimate the first and second moments of the gradients and use them to calculate individual, adaptive learning rates for each weight.

Mathematically the Adam optimizer is defined as follows:

$$m_t = \gamma_1 m_{t-1} + (1 - \gamma_1) \nabla_\theta L(\theta_{t-1}) \tag{2.4}$$

$$v_t = \gamma_2 v_{t-1} + (1 - \gamma_2) \nabla_\theta L(\theta_{t-1})^2 \tag{2.5}$$

$$\hat{m}_t = \frac{m_t}{1 + \gamma_1^t} \tag{2.6}$$

$$\hat{v}_t = \frac{v_t}{1 + \gamma_2^t} \tag{2.7}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.8}$$

where t is the iteration number, θ are the weights and $L(\cdot)$ is the loss function. Note that γ_1^t denotes exponentiation to the power of t rather than a time index. The equations 2.4 and 2.5, are the estimates of the first and second order moments of the gradients. These estimates are moving averages that are both controlled by one decay factor each: γ_1 and γ_2 respectively. The initial values for these estimates are zero which makes them biased. To counter this equations 2.6 and 2.7 compute bias-corrected estimates. Finally the weight update is done in 2.8 by combining the bias-corrected estimates. The weight update is controlled by two additional hyperparameters, ϵ and α, which serve as protection from division by zero and the maximum learning rate, respectively.

The learning rate is a scale factor that controls the step size when updating the weights in a model. In Adam the learning rates are individual which means that each weight will get its own learning rate assigned. In the equations above the learning rate is computed using both the bias-corrected estimates as well as the global scale factor α. This causes the optimizer to target the weight updates towards the weights that are in the biggest need of an update.

As you can see above, this optimizer contains four hyperparameters that needs to be assigned. Determining the best values can be tricky, however Kingma and Ba gave some recommendations in their paper that can serve as a good starting point[19]. Their values are summarized in Table 2.1.

Adam has emerged as a very popular optimization algorithm in recent years, in part due to its speed of convergence. However, recently the convergence proof of the method has been brought to question [35]. Still, the method has strong empirical evidence of working for many different optimization problems, it is just important to be vigilant when using it.

Table 2.1: The recommended initial values for the hyperparameters of the Adam optimizer.

Parameter	**Initial value**
γ_1	0.9
γ_2	0.999
ϵ	10^{-8}
α	0.001

2.3.2 Training procedure

The backpropagation algorithm is not a single step method, it needs to be applied iteratively to arrive at a good solution. To put it into the context of a larger system we formulate the following general training procedure:

1. Split input into training and validation sets, T and V
2. Split T into a set batches B
3. Perform training for E epochs
    (a) For each batch in B
        i. Compute network output and loss
        ii. Apply backpropagation
    (b) Compute loss for V
4. Evaluate result

Here we assume that we have an input dataset available to use for training.

The first step is very important. In order to get an unbiased measure of the performance of the model the available training data is split into two disjoint sets: a training set and a validation set. During training only the training set will be used for backpropagation while the validation set will be used to measure the performance.

Secondly the training set is split into a set of batches. By applying the backpropagation algorithm to mini-batches rather than the entire dataset we make the process stochastic which improves the convergence speed. Since only a small amount of samples is considered at each update step their gradients will have a much larger effect on the weights compared to what they would have had if all of the data was used at once. This will cause the optimization steps to be both bigger and jerkier which leads to quicker convergence and helps avoid small local minima in the loss surface. However, the batch size must not be too small or the model risks becoming unstable.

The third step involves the training itself. Like we have stated several times, the training is an iterative procedure. For how many iterations the training should continue can be decided in many different ways but the easiest method is to just select a value and use the evaluation step to see if it was large enough. The iterations are commonly called epochs in machine learning. A single epoch is equivalent to a single pass through the entire training set.

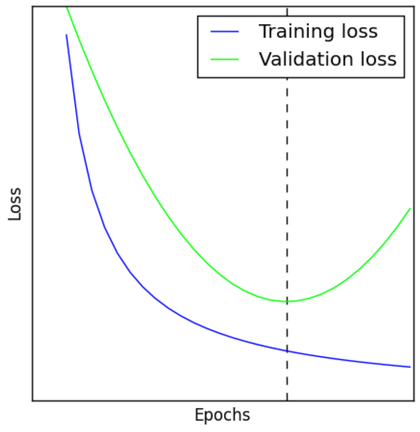


Figure 2.5: An illustration of how the training and validation error behaves over time. The optimal model is marked with a dashed line.

For each epoch every mini-batch is selected and used to compute the network outputs and the corresponding loss. This can then be used to apply backpropagation and update the model weights. Typically the losses you get for each mini-batch are also accumulated to acquire a total training loss, this can be an important metric when analyzing how well the training is going. After training on all the batches the model performance is estimated using the validation set. The resulting loss values can be saved for the evaluation step.

The fourth and final step is to examine the result of the training procedure and decide if any changes should be made. Typically this is done by plotting the training and validation losses versus time to see how they behave. This is discussed in more detail in the following section.

2.3.3 Overfitting and Underfitting

To get the optimal model with the best ability to generalize it is important to know when to stop training. Generally the performance improves the longer you train until you reach a certain point. This is illustrated, somewhat idealistically, in Figure 2.5.

The best model is marked with a dashed line and is located at the minimum of the validation loss. This is the best model since the validation loss is an indicator of how well the model will generalize to new data. If you stop training before reaching the optimum your model is underfitting and if you stop after passing it your model is overfitting.

Underfitting is when the model has not become complicated enough to describe the underlying distribution of the dataset. In this case the model has not yet extracted all useful information from the training data. It could also be the case that the optimum would have been even better if the training had been done on a more advanced model to begin with.

Overfitting is when the model becomes too complicated and starts describing the peculiarities of the training data rather than the true underlying distribution. Overfitting is a common problem for complex machine learning tasks and can be mitigated by choosing a simpler model or by using regularization (see the next section).

While real loss curves probably will not look like the example in the figure the same principle applies: if you train too little (or the model is too simple) you will underfit, if you train too much (without regularization) you will overfit.

2.3.4 Regularization

Regularization is a group of techniques that try to prevent overfitting by keeping the model simple. There are three main ways to do this which are outlined below, the last two were the most relevant ones for our thesis.

The first idea is to simply add some norm of the weights as a term in the loss function, thereby causing the training to keep the weights small when finding the optimal solution. There exist many variants of this approach including ridge and lasso regression [47] among others.

The second idea is to use a technique called Early stopping [34]. It works by aborting the training when the validation error starts to increase. This could easily be done by running two training passes, using the first to determine when to stop and the second to stop at the chosen time. Another way could be to implement a mechanism that saves every model that achieves a smaller error than previously encountered, then when you have trained for some time you just keep the last one.

The third idea is to use a technique called Dropout [41]. This works by randomly removing nodes from parts of the neural network model during training. For each training sample that is processed by the network a set of nodes is randomly removed from every layer that uses dropout. The idea behind this is to stop adjacent nodes from becoming too dependent on each others. When the network models are deployed on real data the dropout is disabled.

2.3.5 Normalization

Apart from the previously discussed topics there is one more important thing to consider when training neural networks and that is whether to use normalization. This technique is very common in machine learning and the reasons behind this are twofold: firstly to avoid differences in scale between various input features and secondly to help the network converge faster.

For instance, if a model has two inputs, one in the range $[0, 1]$ and one in the range $[0, 1000]$ the model will have to adapt to this which might take a long time during training. When using neural networks this is particularly true since the input weights would need to iteratively increase until they differ with a factor of 1000 to let both inputs have equal contributions. This also holds if you have a model were all the variables are contained in the same interval but the scale is huge. You will still need to train a lot to reduce the input weights enough to not oversaturate the model.

There are multiple techniques to perform normalization, two of the common ones are min-max normalization and standard deviation normalization (also known as Z-score normalization) [17]. The latter is the one we focused on during our thesis project.

The standard deviation normalization is computed using the following formula:

$$data_{norm}(i) = \frac{data(i) - mean_{data}}{std_{data}} \tag{2.9}$$

for each input sample i, where the normalization parameters $mean_{data}$ and std_{data} are the mean and standard deviations of the dataset, respectively. It is important to use the same normalization parameters for both training, validation and when deploying to real data, otherwise the behavior of the model becomes undefined.

Chapter 3

Approach

Neural networks are very flexible and are commonly used for problems where a lot of data has to be analyzed. Being a state-of-the-art approach they are also interesting for Axis to explore, which was a key reason why we chose to use them over more traditional machine learning techniques or simpler algorithms.

Constructing neural networks for vehicle recognition and vehicle counting involves several steps which are documented in this chapter. There are two major parts to creating a powerful network: designing it and training it. But before we can do either of those parts we need to have a clear idea of what the network is meant to do.

We want our networks to count vehicles passing through a scene and we want them to do so by looking at the metadata we can extract rather than the video streams themselves. In order to create such networks we have to consider what data to train on and how to get hold of it.

To get realistic training data we recorded videos from three cameras at two different locations, presented in Section 3.1. From these videos we extract metadata to train the network on. In Section 3.2 we introduce the various metadata we have looked at, what they are and how they relate to detecting vehicles.

Before the metadata can be used to train the network it must be properly annotated so that we know if there is a vehicle or not in the data we show the network. This is discussed in Section 3.3 were we consider data annotation and how it can be automated. Finally, Section 3.4 details the construction of our neural networks and how the design differs for each type of metadata. Here we also explain our training process and how we evaluated the networks. This is followed by Section 3.5 where we discuss the process of porting the network models so that they can be deployed on cameras. The complete system can be seen in Figure 3.1 which shows how the various parts relate to each other.

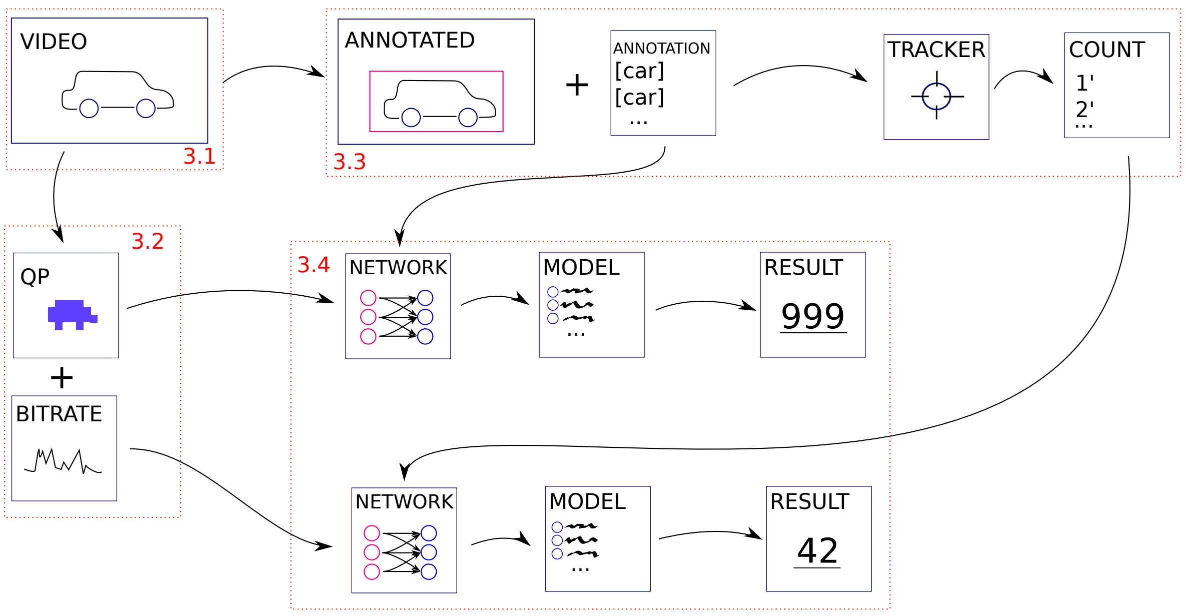


Figure 3.1: An overview of the system as a whole and how its pieces connect to each other. Metadata and annotations are produced from the videos and are then combined to train neural networks. This results in a model that can be deployed on real data, giving a final vehicle count. The dotted regions indicate subsystems described in the various subsections.

3.1 Data collection

Initially we had intended to train the networks on three different scenes in order to cover a variety of cases. This turned out to be difficult given that the laws and regulations on where and how recordings are allowed to be made are very strict. In the end we had to settle for three cameras in two scenes. The first camera filmed a moderately trafficked road, seen from a distance with a slightly elevated perspective, and the other two filmed the gates to an indoor parking garage, viewed from the inside. Figure 3.2 through 3.4 show these scenes.

We added black privacy masks to the road scene when recording in order to avoid filming various parking areas. We feared that the large number of stationary cars would ruin our observations if they were detected as false positives. Adding them to the annotations would ruin the network training since we only want to detect moving vehicles. Ideally the camera should be set up in such a way that this step would not be necessary or so that the undesired detections could easily be pruned away (more on this in Section 3.3.4).

The recordings were done using Axis's cameras. We recorded the videos in clips of 15 minutes. Seeing how we might have to record several days worth of footage we felt this struck a good balance between having continuous videos while still being small enough chunks that we could easily pick out and work with them individually.

We recorded around 2000 videos from the road scene which covered various weather conditions, such as snow, rain and fog, and different times of day. From these videos we would later select a smaller subset to use since there were many more than we needed (see

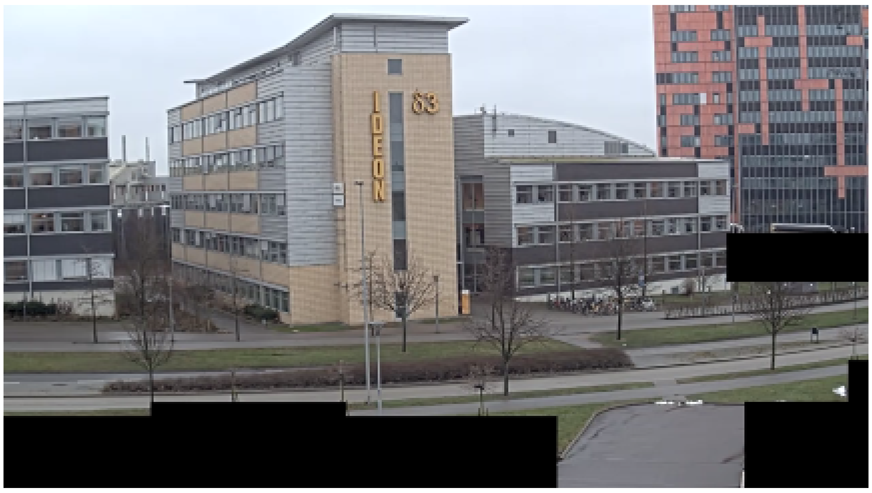


Figure 3.2: An image taken from the road scene. Parking areas with stationary cars have been masked out.

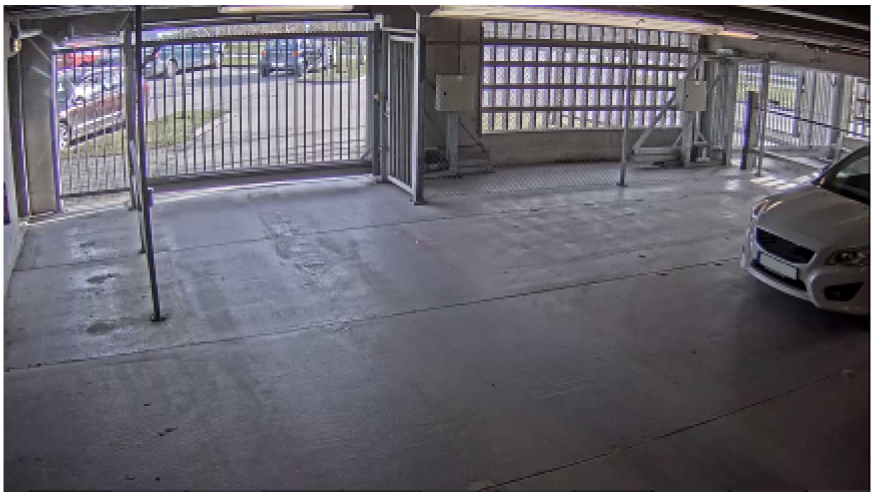


Figure 3.3: An image taken from the first garage scene.

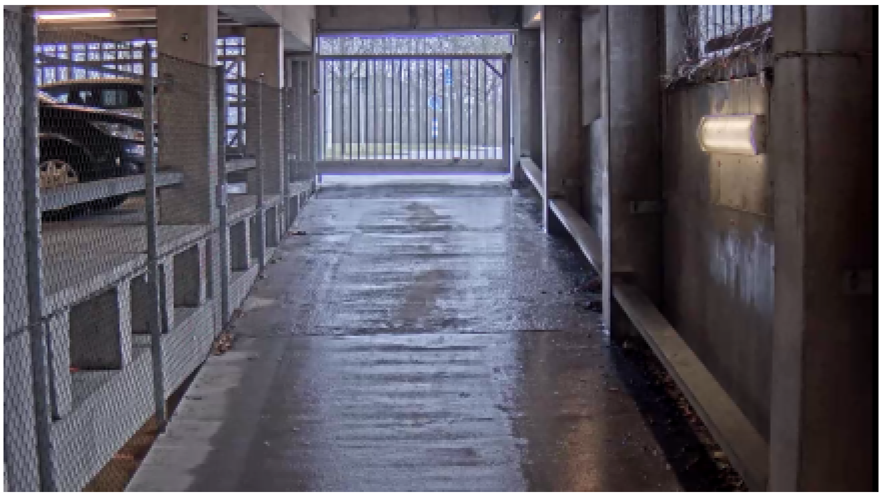


Figure 3.4: An image taken from the second garage scene.

Section 3.4.2). Certain scene conditions were also more prevalent than others, making the dataset biased towards them, which could have a negative impact on performance. From the two garage scenes we recorded 1000 and 1500 videos respectively. Because we could only access one garage camera at a time these numbers differ slightly. Furthermore, it is worth noting that an average road video contains around a hundred vehicles, including a few buses and trucks, while a garage video average at only a single car.

3.2 Feature selection

Metadata is used to describe information about the video rather than its content. Thanks to the dynamic compression algorithms that the cameras used many of these metrics vary between frames in the video depending on what currently is in the scene. Before we discuss the metadata we used we will briefly explain the basics of the H.264 compression standard (also known as MPEG-4 AVC) [12] which has been used for all our recordings. We will then continue to the various metrics we considered, such as bitrate and QP-values, and describe them in detail.

3.2.1 H.264 Compression

Raw, uncompressed video takes a lot of space, even more so if it is recorded in both high resolution and high frame rate. For example, recording a video with a resolution of 1920x1080 pixels (HD) in 30 frames per second and using 8 bits per color channel per pixel gives approximately a gigabit (10^9 bits) of data per second. In order to reduce the amount of data to a manageable quantity (especially if that data is also streamed over a network) different compression algorithms are used.

H.264 is one of the most common video coding standards thanks to its high compression performance and versatility [11]. It was created in a collaboration between ITU-T (international Telecommunication Union) and ISO/IEC (International Organisation for Standardisation / International Electrotechnical Commission) to be efficient in a wide range of applications [12] [40] [48]. The standard defines how the compressed form of the video data should look and the algorithms used to decode it. Only the decoding algorithms are specified in the H.264 standard, the encoding algorithms are left for the implementor to decide on. With a good implementation of the encoder it is possible to reduce the amount of data by a factor of 100 depending on the video, reducing the load in the example above to just 10 megabit (10^7 bits) per second.

The encoding algorithm

While there may be no specification for the H.264 encoding algorithm there is an expected procedure, which is the decoding done backwards. H.264 encoding yields a lossy compression. For each frame it starts by dividing the frame into macroblocks, small groups of pixels, for which it then makes so called predictions. These predictions are done by either comparing the blocks to other blocks within the frame (called intra prediction) or to blocks in previously encoded frames using motion vectors to represent temporal dependencies (inter prediction). The idea is to "predict" the current image using as little new information as possible.

By subtracting the prediction from the original image a residual is created. Each macroblock in the residual is transformed to a set of quantized coefficients to reduce their size. A set of quantization parameters (QP), is used to regulate how aggressive the quantization should be. A large QP value results in less detail as more of the coefficients are set to zero. Conversely, a small QP will keep more information. The final step of the compression is to combine the predictions with the quantized transform coefficients and encode them, using variable length coding or arithmetic coding.

Because there are many ways to create the encoder, existing encoders vary in their performance and can often be customized to better suit specific needs. Deciding which blocks are best to create a prediction from is by itself a difficult problem that can be solved with varying success, especially when more complex techniques are introduced such as varying frame types. As a result, a trade off has to be made between encoding speed, quality and complexity.

Frame types

Many compression formats, including H.264, use a concept called frame types, or picture types. There are three common types of video frames: I-frames, P-frames and B-frames [50]. These frame types define if the video frame depends on other frames in order to be decoded. I-frames use intra prediction only and can therefore be decoded directly while P-frames depend on one or several previous frames. B-frames are bi-directional and may have inter prediction dependencies to both earlier and later frames.

Because P- and B-frames can rely on other frames for parts of the video that are unchanged they can be significantly cheaper to encode in terms of bit count compared to I-frames, however due to the many options available when choosing an earlier frame the encoding process is usually slower. Both P- and B-frames commonly point back to the previous frame, as seen in Figure 3.5, because there are likely fewer differences there than in earlier frames. This is especially true if there is a lot of motion in the video, which usually increases the number of bits required to encode it as the similarities between frames decrease. For the algorithm to work there must still be at least some I-frames or the video would be impossible to decode; without them there would be no anchor point for the other frames to refer to.

There is also the issue of memory usage: with many frames to sample from more frames must be stored in memory while encoding and decoding. In the videos we recorded and used in this thesis most frames are P-frames but I-frames appear once every two seconds.

Including both I- and P-frames has an interesting effect on the metadata where some frames suddenly differ from the rest because they are I-frames and are encoded differently. This will be especially prominent when looking at the bitrate (later in Section 3.2.2) where special care must be taken to avoid these frames becoming outliers.

Zipstream

Axis Zipstream [2] is a specialized H.264 encoder that improves the video compression further. It has three main areas of focus: finding regions of interest (ROI), dynamically

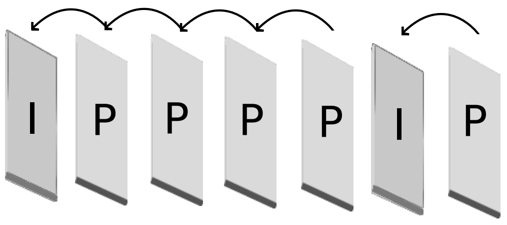


Figure 3.5: An example of a video sequence with frames encoded using inter prediction. P-frames are referring to the previous frame continuously until a new I-frame breaks the pattern.

changing the frequency of I-frames and reducing the frame rate of static scenes. Of these techniques the ROI is the most significant for our work as it means that foreground objects and areas with motion will have a lowered QP-value while the static background remains aggressively compressed at a higher value. The dynamic frame rate and I-frame frequencies were kept turned off. When we started recording we were unsure of what effect they would have on the results and therefore we decided to not introduce additional parameters. There is a possibility that having a lower frame rate when there is no movement could help reduce the noise in the data which might help but would not necessarily improve the results. It would, however, require additional preprocessing of the data before we could use it, making it harder for us to work with the videos. For example, with varying frame rate we could no longer estimate a frame count from how long a video has played and would have to account for that when writing our tools for manual annotation.

3.2.2 Bitrate

The bitrate of the compressed video is simply the rate at which data is transferred, measured in bits per second or a similar unit. A heavily compressed video frame will require fewer bits to be sent which lowers the bitrate while a feebly compressed frame results in a higher bitrate.

By using `ffprobe`, a tool provided by FFmpeg [1] we could extract the size in bytes of each frame in the recorded video. For our purposes we define this as the bitrate, that is the rate of bytes per frame. We also extracted the timestamps and frame types for each data point which we needed for preprocessing of the data. Figure 3.6 shows an example of bitrate over time. The large, periodical spikes in bitrate correspond to I-frames where significantly more data is being sent. These spikes would be a problem for a neural network as they are extremely large outliers compared to the values of the P-frames. When the values are normalized the interesting variations found in the P-frames would be practically invisible and the networks would most likely have a harder time following them.

In order to deal with this we preprocessed the recorded bitrate to smooth out the spikes. We used the frame types we had extracted from `ffprobe` and replaced all I-frames with

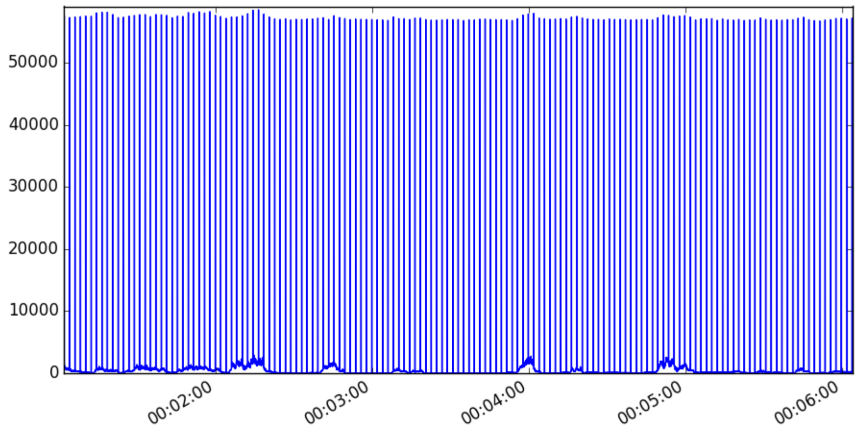


Figure 3.6: An example of bitrate where the I-frames produce massive spikes.

an average of the preceding and succeeding frames through the following equation:

$$b_i = \begin{cases} b_{i+1}, & i = 1 \\ b_{i-1}, & i = N \\ \frac{b_{i-1}+b_{i+1}}{2}, & otherwise \end{cases} \tag{3.1}$$

where b_i is the bitrate of the ith frame out of N frames. The result of this smoothing can be seen in Figure 3.7.

3.2.3 QP-values

As described in Section 3.2.1, QP-values are the rate of compression for blocks of pixels within the image (16x16 pixels in our case). Normally these quantization values are relatively large but they are lowered to enhance details. Thanks to the H.264 encoder areas with lots of changes, a moving car for instance, are given a low QP while the static background is kept at high values. Figure 3.8 shows an example scene where blocks with low QP form the silhouette of moving vehicles.

To further reduce the bitrate only QP-values that have changed between frames are actually sent. This means we lack QP-values for most blocks but given that these blocks likely were not interesting (or they would have been encoded with a low QP and not skipped in the first place) we decided to ignore the missing blocks and replace them with a maximized QP-value. This seemed to be a good approximation as we could not spot any incorrect mappings when comparing a video converted to QP-blocks with the original video.

We used the reference implementation of the H.264 decoder [44] to extract the QP-values from our recorded video clips. We modified it slightly to be able to go through each frame of the videos and write the frame's QP matrix directly to a file on a format we could later read. It was not possible to extract QP-values from every video since some of them had frames missing due to network latency and the use of the UDP-protocol. This meant that we lost up to 10% of data for QP compared to bitrate. Because of how many

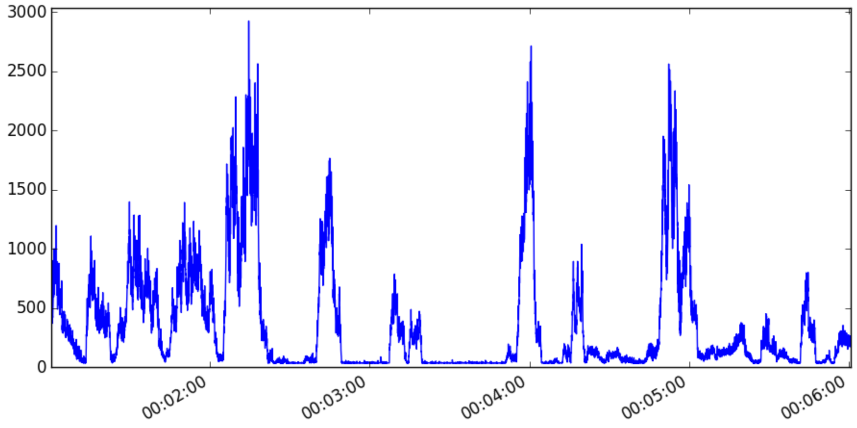


Figure 3.7: With I-frames removed it gets a lot clearer. Now the spikes come from vehicles passing through the scene.

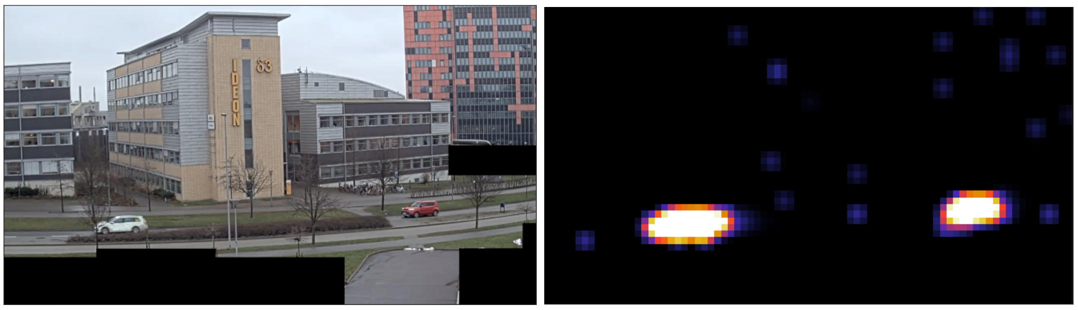


Figure 3.8: A frame from the road scene and its corresponding QP-values. The QPs go from bright (low) to dark (high). The bright blobs represent the two cars while the darker dots are noise. Note that some temporal and gaussian smoothing has been applied to the QP-values to make the image more readable.

videos we had, this was not a big issue but it caused our QP network to have slightly fewer videos in its evaluation set than the bitrate network had.

3.2.4 Motion vectors

During encoding when the predictions are made for a new frame, each macroblock is compared to blocks in the earlier frames to find the one that is the most similar. The macroblock is then assigned a motion vector that points to that block. The vector is two-dimensional and simply indicates the relative location between the two blocks. By extracting these vectors we would get data similar to the QP-values but instead of just a level of compression we would see both how much the block has moved and in which direction.

These values could potentially be very powerful features but we decided not to use them in our research. The reason for this is twofold. Firstly, being an internal part of the encoding algorithm we thought the motion vectors would be harder for us to extract when running the network models on the camera compared to the bitrate and QP-values. Secondly, having two-dimensional values as input would require a larger network compared to using QP-values. Because we did not yet know what limitations we were working with we decided to focus on the QP-values instead. In Section 4.2.8 we will discuss how using motion vectors could change our results.

3.2.5 Signal to noise

The signal to noise ratio (SNR) can be extracted per frame and gives an indication of the noise level of the scene. Noise is introduced as the light levels decrease which makes the SNR a way to estimate the time of day. While very limited as information source on its own it is possible that combining SNR with another input, such as bitrate which varies greatly depending on the time of day, would help the network balance changes to the average amplitude of bitrate, thus making it more robust. Because of time constraints this feature was never implemented into the networks and remains as a theoretical possibility.

3.3 Data annotation

In order to train our own neural networks we needed data to train on. As we saw in Section 3.1 we collected a lot of video material from which we extracted both bitrate and QP-values. However all this data is useless unless it is annotated.

The point of the annotation is to mark where in the video sequences all vehicles are located, both in time (which frames) and in space (where in the frame). Together with labels of what kind of object it is this can then be used to train neural networks. To avoid having to go through and label thousands of frames manually (which would have been both tedious and impossible to scale) we decided to use an existing neural network to annotate the video footage for us.

In the following sections we will describe the network we used to detect vehicles in the videos, how we modified it to do a count of the vehicles and how we improved it to fit our scenes and raise its performance.

3.3.1 You Only Look Once

Object detection in images is nothing new and there already exist several powerful neural networks that can do it. For our project we needed a model that could recognize cars and preferably also buses and trucks so that our networks could learn the differences between these vehicles (or at least recognize them all as vehicles). We also needed the network to be fast enough to be able to annotate all videos in a reasonable timeframe. Most importantly it had to be accurate. The annotating network's performance would put an upper bound on the performance we can achieve with our own networks and if it could not detect vehicles in the scene it would mean we would be using inaccurate labeling and poorly annotated data for training.

We decided to use a TensorFlow implementation of YOLOv2 (You Only Look Once [37]) called Darkflow [46]. YOLO is a state of the art object detector designed to be fast while still being on par with other powerful networks such as Faster R-CNN [39]. It also comes with pre-trained weights for the COCO object detection datataset [23], which conveniently for us contains labels for cars, buses and trucks among others. While the original implementation of YOLO is written for Darknet [36] we decided against using it since our coworkers at Axis have a lot of experience with TensorFlow. Furthermore, it would also be more consistent with the rest of the project since we were planning to build our own neural networks using TensorFlow as well (see Section 3.4).

By default YOLO outputs bounding boxes around objects it found in the input image. However, Darkflow also provides functionality which enables annotation of videos by extracting their individual frames and sending them to YOLO. Using this system we could annotate our videos at 33 frames per second on our Nvidia GeForce GTX 1080 Ti graphics cards. We modified the source code to output the bounding boxes on a JSON format for videos as the original implementation only provided JSON output for still images.

3.3.2 Counting cars

YOLO can tell where in the image objects are located but every frame of a video is being considered individually, i.e. there is no continuity. Lacking the notion of tracking, the system cannot by itself be used to count vehicles; it does not know if two bounding boxes in subsequent frames depict the same vehicle or not.

This issue applies to our networks as well. As Figure 3.8 showed, a QP-frame is very similar to a video frame. Thus, if we were to use a network similar to YOLO on the QP data it too would need some kind of post processing in order to count the vehicles. If we instead were to use the bitrate the network would return the total amount of appearing vehicles in a sequence. In the latter case no post processing is necessary for the network, however a tracker will still be needed to generate the true vehicle counts that are used during training. To solve this we created a tracker that takes YOLO's output and converts it into a list of timestamps, one for when each new vehicle appears. This lets us get a total vehicle count from YOLO (to train the bitrate networks) and later from the QP-network as well.

There are a few important things we considered when building this tracker. Firstly, because it will be used during the annotation procedure its performance will have an immediate effect on the performance of our neural networks. Much like YOLO its performance will put an upper bound on how high accuracy our networks can achieve. With that said, if the tracker is really good it might compensate for YOLO's misclassifications, thereby not causing any further reduction in accuracy and possibly even an increase.

Secondly, we had multiple options for how to implement the tracker. We settled for writing a simple program that checks if two bounding boxes in subsequent frames are close enough in Euclidean distance. This solution proved to be very fast and with some tweaking and fine-tuning (see Section 3.3.4 for details) it ended up being fairly accurate.

Other possible solutions could be to use more machine learning, for example another neural network that does tracking, or to use existing implementations. Both of these options were considered too time consuming to try unless our own tracker proved too unreliable.

Table 3.1: The initial set of manually annotated videos for the road scene. All videos were 15 minutes long and recorded at 30 frames per second. The asterisks mark the videos that were in the set of most common scenarios.

Location	Description	Vehicle count
Road*	Sunny day	109
Road*	Sunny day, snow covered ground	99
Road*	Sunny day, dense traffic	203
Road*	Clear evening	177
Road	Thick fog	179
Road*	Clear night	68
Road*	Rainy day	79
Road*	Rainy day, alternate	100
Road	Snowy day	139

3.3.3 Evaluating performance

To get an idea of how accurate YOLO and our tracker were we annotated some videos manually. Because of the work required to manually place bounding boxes in the videos we decided to only specify when new vehicles entered the scene. While this is not enough to estimate the performance of YOLO by itself, it would tell us how well YOLO performed in combination with our tracker.

In Table 3.1 we present our initial set of videos that were annotated manually and used for evaluation. More videos were added to these at a later point, including videos from other scenes to get more accurate measurements when evaluating the networks we built ourselves. For a complete list, see Appendix A.

It is worth noting that all our initial videos were from the road scene because it took us some time before we could start recording from the garage. This means that our annotation and network construction were based primarily on this scene and only modified to perform better at the garage scenes.

The manual annotation was performed by letting a person watch a video sequence and press a button to record a timestamp whenever a vehicle entered the scene. We could then decide on how lenient we would be when mapping them to detections generated by the tracker. The person then re-watched the videos and added classes to the timestamps specifying what kind of vehicle it was. We counted only cars, buses and trucks: while bicycles could have been included as well we noted that they were too small to be reliably classified by YOLO over the large distances present in the road scene.

There were some inconsistencies as to when a vehicle should be considered inside the scene. We decided not to count a vehicle unless it was entirely within the camera's field of view even though YOLO would still count these half-occluded vehicles on occasion. We were also unsure how to clearly separate cars from trucks in several cases since their size and shape could fall under either of the two categories. In the end we decided, subjectively, that larger vans should be considered as trucks and smaller should be cars. If our manual classifications would turn out to introduce a lot of errors we could ignore the

Table 3.2: Accuracy of the original YOLO model trained on the COCO dataset + tracker without modifications, compared to manual annotations. Road scenes only.

Scene	**Vehicle count**	**True count**	**Matches**	**Precision**	**Recall**	**F_1 score**
Road, sunny	147	109	87	0.5918	0.7982	0.6797
Road, traffic	272	203	174	0.6397	0.8571	0.7326
Road, night	118	68	30	0.2542	0.4412	0.3226
Road, evening	228	177	167	0.7325	0.9435	0.8247
Road, fog	294	179	109	0.3707	0.6089	0.4608
Road, rain	130	79	54	0.4154	0.6835	0.5167
Road, rain 2	157	100	65	0.4140	0.6500	0.5058
Road, snow	204	139	94	0.4608	0.6763	0.5481
Road, white	159	99	56	0.3522	0.5657	0.4341
Average	—	—	—	0.4701	0.6916	0.5597

classes altogether and ask YOLO to recognize trucks and buses as cars, with the obvious drawback that our neural networks would no longer be able to tell them apart.

Using the manual annotations we computed the precision and recall of our YOLO + tracker combination. At this point we also consider a wrongfully labeled vehicle as a miss (i.e. the vehicle types are significant). The precision and recall metrics are measurements of accuracy, with precision being the fraction of counted vehicles that were correctly classified (i.e. matched a real vehicle) and recall being the fraction of real vehicles that were counted. Formally they are defined as:

$$precision = \frac{TP}{TP + FP} = \frac{matches}{vehicles\ counted} \tag{3.2}$$

$$recall = \frac{TP}{TP + FN} = \frac{matches}{true\ count} \tag{3.3}$$

Here TP, FP and FN are true positives, false positives and false negatives respectively. This means that a low precision corresponds to a large number of false positives while a low recall is the result of missing several vehicles. The metrics have an inverse relationship such that an increase of recall often reduces precision and vice versa. This is not a one-to-one relationship but rather a side effect of their definitions. For example, if only a single one of several passing vehicles was counted the recall would be near zero but precision would be one. Conversely, if many objects in the scene were incorrectly counted as vehicles then the recall would be one and precision zero. Ideally, both precision and recall should be close to one. We have also calculated the harmonic F score (known as the F_1 score). This is a weighted measurement that describes accuracy and is defined using precision and recall as follows:

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \tag{3.4}$$

The results we got when evaluating the YOLO + tracker combination are presented in Table 3.2. Evidently the performance was mediocre: although our system managed to spot a majority of vehicles, it has a large number of false positives. We also see the performance go down significantly at night and when weather conditions apply, likely because YOLO was not trained on those scenarios so it provides worse data for the tracker. We believe the main reason for the poor results still lies with the tracker. By watching the videos and comparing the annotated bounding boxes with the vehicles found by the tracker we saw that many of the false positives came from re-emerging vehicles that were occluded by obstacles in the scene. The tracker also struggled when multiple vehicles passed near each other. In order to raise performance we decided to improve the tracker but also to fine-tune YOLO to focus on vehicles only, hoping that this would raise its detection rate.

3.3.4 Fine-tuning

The initial version of our annotation framework had poor accuracy (see Table 3.2) and we identified several problems which could be amended. We decided to fine-tune our system to increase its performance and in the following sections we discuss improvements to both the YOLO network as well as our tracker.

Specializing YOLO

The version of YOLO that we used was initially trained on the COCO object detection dataset as we stated earlier. The dataset consists of 80 different classes, ranging from vehicles to animals, furniture, accessories, etc... By targeting this wide range of object classes the network needs to store knowledge of many different types of shapes. In contrast, if you would train the same model on fewer categories, say three or four, there will be a lot fewer features it needs to adapt to, which means that a larger part of the network can specialize on each class.

Following this line of reasoning we decided to fine-tune the YOLO model by using a restricted dataset which only contained a few of the original classes. This was done by loading the pre-trained model with 80 classes and performing additional training on the restricted dataset. While we could have trained the network from scratch it would take much longer than we were willing to spend, especially as the original 80 class YOLO model had already been trained for a week to reach its current state.

In order to get a diverse set of models we decided to fine-tune on both one, three and four classes, see Table 3.3 for a short summary. For each of these cases we created a dataset by taking all images from COCO that contained the corresponding classes. The goal was to create all of these models and then perform comparisons to determine which one gave the best performance. When fine-tuning we used Darkflow with the Adam optimizer, a learning rate of 0.00001 and a batch size of eight.

In order to perform fine-tuning successfully we needed to figure out at which point during training we had found the optimal model. We did this by running a validation pass after each iteration of fine-tuning. We computed the error on a subsample of the validation data in the COCO dataset and summarized the result in a few plots. By default Darkflow does not include any method for model validation so we had to implement one of our own to get this to work. Figure 3.9 shows how the training and validation errors behaved over a large number of training steps for the model with four classes — the plots for the other models looked similar. As you can see the error initially dropped only to

Table 3.3: The different sets of classes we fine-tuned YOLO on, including the amount of training images we used for each of these networks.

#	Classes	Images
1	Car	8606
3	Car, Truck, Bus	11432
4	Car, Truck, Bus, Bicycle	12667

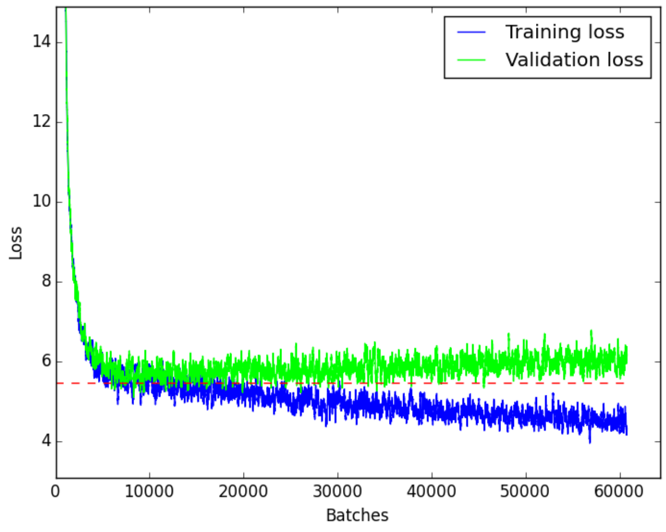


Figure 3.9: The validation and training errors versus training steps when fine-tuning the YOLO model on 4 classes. Each step corresponds to the computation of one batch. A moving average of size 100 was applied to smooth the curves. Note the increase in validation error after the initial drop.

start increasing again after a while. Using the information in the graph we arrived at three different approaches to finding the best model.

The first approach was to just train the model for a long time and keep the last result. The rationale behind this idea was that even if it has overfit slightly to the training data it would still have a solid representation of the vehicles. When we watched the annotated videos that YOLO produced using this model the bounding boxes tended to be smaller and slightly more stable than with the other models.

The second approach was to stop training at around 10000 steps when the validation error is at its smallest, using early stopping. Adapting this strategy we should be able to capture the model before it starts overfitting, however the drawback could be that the result is less stable since the model might not have been trained enough. When we watched the resulting videos using this model we observed that the bounding boxes indeed were more prone to flicker and vary in size compared to the first approach.

The third approach was to train for a long time but only keep the model that got the

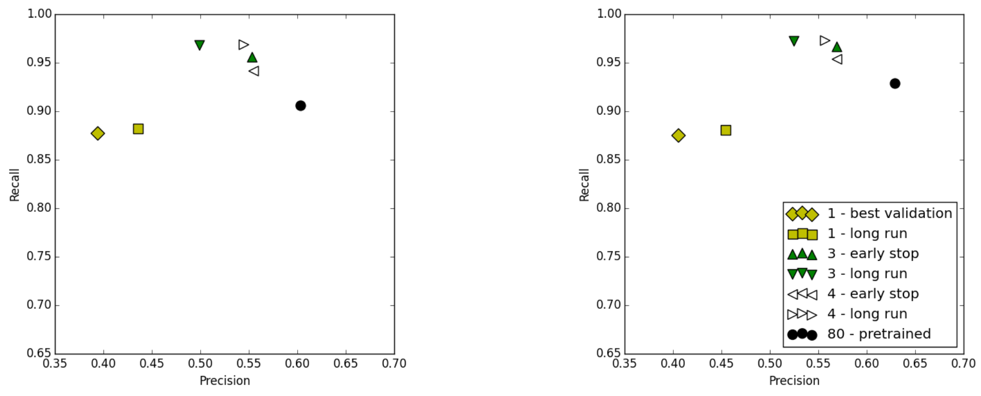


Figure 3.10: The average recall versus precision for models with different amounts of classes and using different approaches for model selection. The left graph represents an average over all weathers and times of day (see Table 3.1), while the right represents an average of the most common scenarios (entries with asterisks in the table). These graphs were generated using the initial version of the tracker.

smallest validation error (i.e. the lowest point in the graph). Using this approach we thought we could capture a point in time when the model momentarily performed really well. This turned out to work poorly in practice however. When we watched the result it seemed to overfit at least as much as the first approach if not more. This method was probably incorrect since the validation only considers a sample from the validation set at each time step, this means that a small error could be due to chance rather than the model being good.

To determine which of the above approaches was the best one we performed some manual validation by observing the output YOLO produced. Since the third approach appeared to perform poorly at best and could be downright incorrect at worst we decided to focus on the other two. To make a robust comparison we constructed a number of different combinations of the one, three and four class models with the different approaches. For each of the models we computed the average recall and precision over all the different scenarios in Table 3.1. We also made the same computations and averaged over the most common scenarios for comparison (marked with asterisks in the same table). The results of this can be seen in Figure 3.10.

The figure is constructed such that the bottom-left contains the worst results and the top-right contains the optimal values. Determining the best model from these graphs will be done differently depending on whether a high recall or precision is the most desirable. We decided to prioritize a balanced trade-off between the two, using the precision to break ties. Since the output of the model will be used for training neural networks later we want to keep both the precision and recall high to avoid false positives/negatives as far as possible.

With that in mind we can analyze the contents of the graph. The original model with 80 classes performs the best, followed by the three and four class models which have approximately the same performance. Lastly we have the single class models which performed

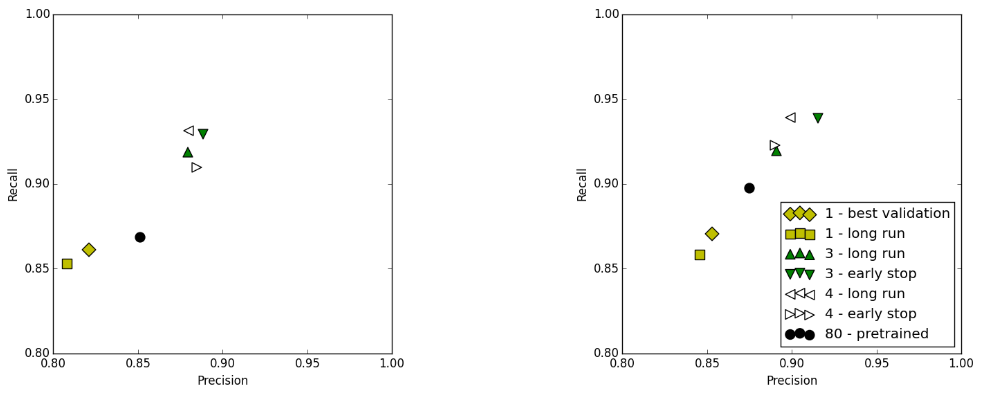


Figure 3.11: The average recall versus precision for models with different amounts of classes and using different approaches for model selection. The left graph represents an average over all weathers and times of day (see Table 3.1), while the right represents an average of the most common scenarios (entries with asterisks in the table). These graphs were generated using the improved version of the tracker. Note the different axis limits compared to the previous figure.

poorly.

Since the single class models only looked for cars while the manual annotations also contained buses and trucks there is a natural degradation in accuracy for those models. We can also see that the version that kept the lowest validation error had the worst result which confirms that this approach did not work.

The reasons the 80 class model outclassed all of our fine-tuned models may be several. Firstly, the initial version of the tracker was constructed on top of this model which may have caused overfitting. Secondly, by watching the resulting videos from all our models it seems that the 80 class model supplies a more stable output than the others which makes it easier for the tracker to work correctly. The original version of YOLO was trained with other values for the hyperparameters, such as a higher batch size, which might have resulted in a more stable model.

Both the three and four class models performed similarly with the early stop approach having slightly higher precision and lower recall than the long run approach. These models also had a higher recall compared to the 80 class model which might be an effect of the less stable output (e.g. flickering bounding boxes can count as several vehicles).

Even though the above discussion seems to indicate that we should keep the 80 class model and discard our fine-tuning we must also consider the final result after the tracker was improved. Figure 3.11 displays the same graphs as before using the improved tracker (see the next section). This figure shows a completely different result compared to the previous one. While the single class models are still performing the worst the three and four class models are now outperforming the 80 class model. In particular the three class early stopping model seems to be the best one. Consequently it is the model we used during the rest of the thesis project.

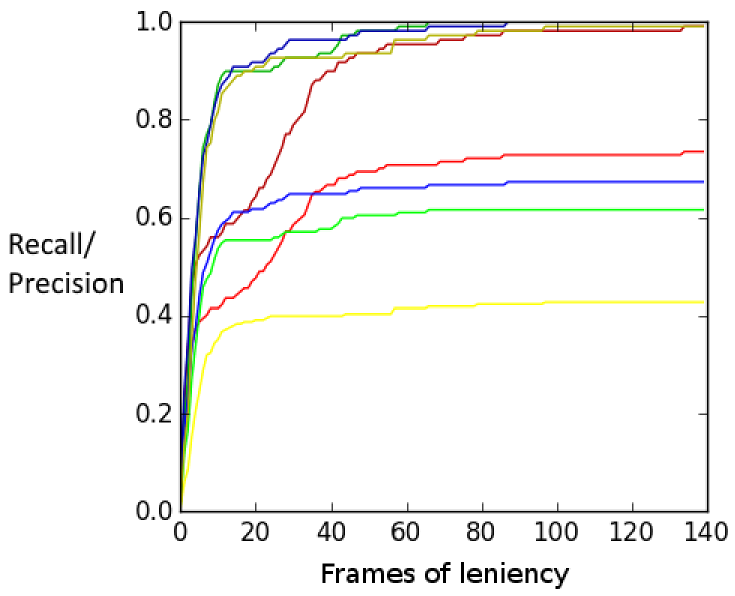


Figure 3.12: Plots showing how recall and precision for some of the networks increase as the allowed detection deviation increases. The key point to take from this graph is how the accuracy always rises quickly followed by a slower increase as the window grows wider and more unlikely matches are let through.

Improved tracking

As mentioned earlier we had to decide how many frames a vehicle would be allowed to deviate from our manual timestamps before we considered it a mismatch. If the window was set to too small a value the inherent inaccuracies of our measuring approach would make us lose a lot of otherwise correct matches. Conversely, if we were too lenient we would potentially match false positives with our manual timestamps, giving us a misleading boost in precision and recall. By plotting the precision and recall against how many frames of leniency we used, illustrated in Figure 3.12, we could see that there was a large increase in accuracy early on from very low levels. From this we decided that an interval of plus minus two seconds (60 frames) was a reasonable time window, seeing how it encapsulates the steep rise of accuracy but little more, and limited ourselves to that. If other changes to the tracker would speed up the increase in accuracy, or drastically improve it, we could then lower this threshold and still get good results.

The next step was to enhance the system's ability to keep track of vehicles. This was done through many iterations of trial and error where we compared the new precision and recall on the manually annotated data with previous results to see if we had improved. We did this by plotting precision and recall for all different YOLO versions we had for each video, then we compared the graphs by overlapping them and keeping the optimal ones.

To deal with bounding boxes disappearing between frames (either because YOLO fluctuated or because of temporary occlusion) we allowed them to be missing for up to 40 consecutive frames before we flagged them as gone. We added an estimation of velocity, calculated as the difference in horizontal and vertical position compared to their last known position and then dividing by the time they had traveled. This let us predict their movement over time if they disappeared. Because the road scene is (almost) limited to horizontal movement, we decided it was enough to only predict the horizontal velocity as large vertical movements were more likely to be the result of errors. This also reduced the risk of confusing bounding boxes of vehicles that passed each others in opposite directions. To further limit accidental mis-predictions caused by vertical proximity to other vehicles we changed the range inside which movement was accepted to a rectangle of 100 by 50 pixels (in width and height) rather than our original range of 100 by 100 pixels. Finally we averaged the velocity predictions using the 40 last occurrences for each bounding box to smooth out large sudden changes in movement. This helped when the size of the bounding boxes was varying a lot between frames.

We also tried to remove bounding boxes that were completely wrong. We added requirements that they would have to move a minimum of 100 pixels from their initial position and appear in at least five frames throughout their lifetime to be considered real (these did not have to be consecutive). We experimented with requiring a higher level of confidence from YOLO's annotation, discarding all bounding boxes with a confidence of less than 0.63 (the default is 0.6). This turned out to work well for some networks but after we had implemented more of the constraints mentioned above we found that reverting back to the default gave better results.

Our last step was to disable classes. By accepting any vehicle class as just a vehicle we could get a significant boost in both precision and recall because we could avoid misclassifications where YOLO knew the position of a vehicle but failed to predict its type. Here we argued that it was better for us to start with a simpler problem with more accurate training data and perhaps later add support for classes again if our networks performed good enough.

Compared to earlier (Table 3.2) the precision has increased drastically across the board with around 30 percentage points. The recall has been reduced somewhat which is to be expected given the inverse relationship described earlier. Furthermore, the precision and recall are now very close to each other, which means that the number of vehicles counted by the tracker should be fairly accurate because there are about as many false positive as false negatives in each video. In Table 3.4 we have taken the three-class, early stopping version of YOLO together with the improved tracker. Here the performance increase is even larger and both precision and recall lie around the 90% mark.

It would be possible to further tune the tracker to increase its performance, for example by introducing uncertainty to the model or forbidding sudden changes to the direction of motion to reduce the risk of bounding boxes getting mixed up. We could also try to determine when bounding boxes are occluding each others (e.g. a car passing behind a bus) and treat it as a special case. Despite all options for improvements we chose to stop here in order to limit the scope of the thesis. The accuracy we got is good enough for our needs.

Tracking in the garage scenes

Until now the tracker has only been adapted to the road scene with no consideration taken to other locations. Unfortunately, the improved tracker still performs poorly on videos from the first garage scene. Studying the videos annotated by YOLO we noticed that YOLO has

Table 3.4: Accuracy of YOLO, fine tuned on three classes and stopped early + tracker with all modifications, compared to manual annotations. Using the road scene only.

Scene	Vehicle count	True count	Matches	Precision	Recall	F_1 score
Road, sunny	108	109	100	0.9259	0.9174	0.9216
Road, traffic	204	203	184	0.9020	0.9064	0.9042
Road, evening	181	177	174	0.9613	0.9831	0.9721
Road, night	69	68	63	0.9130	0.9265	0.9197
Road, fog	201	179	149	0.7413	0.8324	0.7842
Road, rain	86	79	74	0.8605	0.9367	0.8670
Road, rain 2	103	100	95	0.9223	0.9500	0.9359
Road, snow	146	139	135	0.9247	0.9712	0.9474
Road, white	112	99	91	0.8125	0.9192	0.8626
Average	—	—	—	0.8848	0.9270	0.9054

trouble detecting cars viewed from the front or rear. It also lost track of cars if the driver opened the door to access the gate terminal. Moreover, the garage had several cars passing outside the gate that YOLO detected but we did not want to count.

By significantly increasing the number of frames we allowed a vehicle to be missing and returning the tracking range to a 100 by 100 pixel square we could improve tracking somewhat. The increased leniency meant that we could find the vehicle again when the driver was done with the terminal and had closed the car door. The increase in vertical tracking range (100 compared to 50 pixels) compensated for the road no longer being strictly horizontal. We could ignore the irrelevant passing vehicles by requiring YOLO's bounding boxes to have a center coordinate lower than a certain threshold. This has an effect similar to the black privacy masks in the road scene, with the difference that it is added after the video is recorded. However, bitrate and qp-values will still be affected by objects passing in the background which might complicate training.

These changes improved the precision and recall slightly but they were still not near the levels of the road scene. We attributed most of the poor performance to YOLO's inability to track the cars from the camera angle in this scene since it produced too few frames for the tracker to work with. We lowered YOLO's confidence threshold from 0.60 to 0.25 in order to get more detections. By requiring less certainty from YOLO, frames that would otherwise have been discarded would now be approved but this naturally decreased the accuracy of the predictions as well. The result was a major boost in both precision and recall for the tracker. We concluded that the lower quality detections were not an issue in this scene as the tracker could either use the additional detections or discard them as incorrect, given the constraints put on it.

Worth noting is that the second garage scene, which showed only a straight lane to the gate, did not benefit as much from these tweaks to the tracker. We did keep most of the changes but let YOLO's confidence threshold stay at its original, higher level. This may not have been optimal as some vehicles were still hard to detect but we lacked time to experiment and find the best value.

3.4 Network construction

Once the data annotation and fine-tuning was done it resulted in a lot of preprocessed annotated data that could be used by neural networks to perform vehicle counting. The next step was to construct the networks. However, how the networks should be structured and how they function depends on what input is used and how the input is interpreted. Since the bitrate and QP data are of very different formats their models will need to be constructed in different ways. Moreover, we also implemented the bitrate network using three unique ways of interpreting the bitrate data, resulting in four networks in total. We decided to count every type of vehicle as a single class to make the models less complex.

This section starts by detailing our training setup and the common components that are shared across the different bitrate networks, then we dive into the specifics for all the networks.

3.4.1 TensorFlow environment

We decided to build our networks using TensorFlow [3] since it is widely known to perform well and since Axis has a lot of experience using it. To get a clean environment for TensorFlow to run in we created a Docker [16] image using nvidia-docker [32]. Apart from TensorFlow the docker environment also contained the Jupyter notebook system [18] which we used as our primary IDE when building the networks. The networks themselves were trained using a computer with the following specifications:

- Intel®Core^{TM} i7-7700K CPU @ 4.20GHz
- 16 GB RAM
- Nvidia GeForce GTX 1080 Ti GPU/Nvidia GeForce GTX Titan X GPU

3.4.2 Dataset selection

Because we recorded the videos during the course of the thesis project we only had very limited data early on when creating our networks. This, combined with the wish to iterate quickly, led us to use smaller datasets when iterating on and evaluating the designs of our networks. These datasets varied in size but ranged between 30 to 300 videos, with the latter being most common. They often included both day and night scenes and cloudy but clear weather, creating a baseline to work from.

When evaluating the final network designs we used several more videos. Table 3.5 shows the exact count for the different scenes and the distribution of time and weather between them. When deciding on which videos to include we wanted a natural mix of possible scenarios. We focused on clear weather and daytime videos as those were the most likely videos to contain many vehicles. We then made sure to include special weather conditions at least once, picking videos from days with rain, fog and snow. For the garage scenes weather had a very small effect, which was fortunate for us as we had less of those videos to choose from.

Table 3.5: The final datasets we trained the networks on. The weather distributions are somewhat approximative, especially for the garage scenes where it was harder to discern. In general the garage scenes have slightly more snow and sun and less rain and fog.

Dataset	**Video count**
Road scene	560
Garage 1	571
Garage 2	579
	Daytime distribution (Day / Night)
Road scene	300 / 260
Garage 1	255 / 316
Garage 2	395 / 184
	Weather distribution (Sun / Overcast / Rain / Snow / Fog)
Road scene	25 / 395 / 60 / 50 / 30
Garage 1	Similar
Garage 2	Similar

3.4.3 Bitrate training setup

Even though the network models differ a lot in their implementation there are several components and techniques that were the same for most of them. Our training pipeline, including input pruning, parsing, preprocessing and normalization; our choice of loss function and optimizer; and our use of regularization were the same across all networks. The only major exception to this setup is the QP network which instead is detailed in Section 3.4.8.

Pruning and parsing

Before loading all the data we decided to apply some pruning in order to remove bad entries. Some of the videos ended prematurely or missed large chunks of bitrate statistics which could cause problems during training. All videos that were missing more than ten seconds of footage or bitrate statistics were pruned during this procedure.

After the pruning was done the remaining data was loaded and parsed in preparation for the next steps. The data contained both the bitrate statistics and the corresponding annotations that were generated by YOLO and our tracker (see Section 3.3).

Preprocessing

The first step in the preprocessing was to remove all the I-frames, like we discussed in Section 3.2.2. Since the I-frames are infrequent and their magnitude is much larger than the P-frames they are simply a huge source of noise and outliers.

The second step was to handle fluctuations in the recording speed. Even though the videos we used as input were recorded at a fixed framerate there were still fluctuations

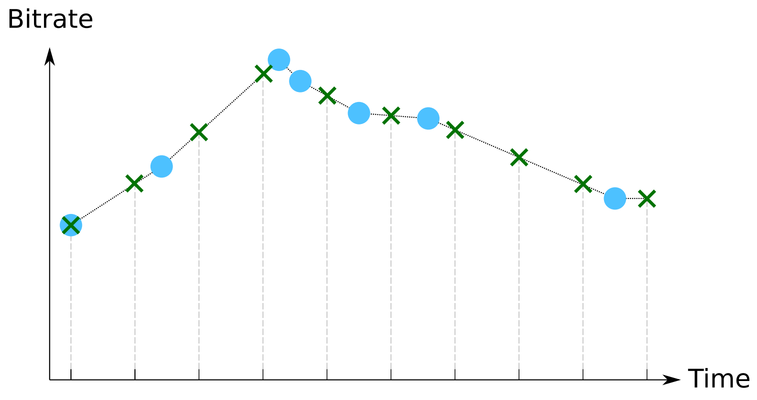


Figure 3.13: An illustration of our bitrate post processing. The circles represent the uneven input and the crosses represent our evenly interpolated output. Note how the last cross is handled.

during the recording due to network delays. This causes both the amount of frames in each video as well as their distribution to differ slightly. To make the input to our networks deterministic and predictable we ideally want a fixed amount of equally spaced frames for each input.

We solved this problem by placing the input on a time line and interpolating between adjacent points to get an approximation of the true input at a fixed frame rate. See Figure 3.13 for an illustration. After applying this preprocessing we always get the same amount of frames from each video and they will always be equally spaced.

Theoretically, removing the fluctuations is important to get optimal performance out of our networks because it removes the "noise" a varied input spacing would cause (a slope could be seen as either steep or gentle depending on the input width). If we were to train them on input with some distribution of frames but the input it receives when running live has another distribution (due to other network delays for instance) its output could become undefined. The problem would become even worse if dynamic frame rate is used since that could make the frame distribution very uneven. Interpolating also makes sure the videos have an amount of bitrate samples that is divisible by the network input size. Otherwise we might have to throw away the end if a few frames are missing.

The final preprocessing step was to randomize the order of the input data. This step was important to give every batch a more even distribution of different types of input. Without randomization we risk "temporal overfitting" because the videos would be in the order they were recorded. This happens since every contiguous chunk of videos would have approximately the same light conditions, weather and traffic intensity. For the same reason randomization is also important when using cross-validation (see Section 3.4.4).

Normalization

Normalization is important to make the model behave as predictably as possible, especially when the scale differs a lot among the input. Our dataset consists of bitrate curves which, depending on the weather or time of day, could stay both close to zero or in the range of thousands, see Table 3.6 for some examples. The curve can also change quickly when there is movement in the scene. Both the magnitude and the fluctuations would have caused

Table 3.6: Examples of typical ranges of bitrate values for different conditions (for the road scene).

Weather	Typical bitrate range
Sun	50 - 4000
Cloudy	100 - 6000
Rain	300 - 14 000
Fog	30 - 2000
Snow	1000 - 70 000
Night	50 - 3000

problems for our networks without normalization.

When implementing the normalization we decided to try two different approaches. The first approach normalized over all the training data and stored the normalization parameters for later use on the test data or when running live. The second approach normalized each input sample by itself without reusing the normalization parameters, this meant that no values were stored for later use. The normalization procedure itself was a simple standard deviation normalization as defined in Section 2.3.5 and it was applied according to the two approaches above.

Loss function and optimizer

Apart from the preprocessing of the input the main part our system is the network models. While the network structures differed between our models the loss function and optimizer were kept the same.

Since the bitrate models are all regression models with continuous output we decided to use the mean squared error as our loss function. We also decided to use the Adam optimizer since it is known to work well for many problems. The hyperparameters of Adam were left to their recommended default values during training, except for the learning rate which varied depending on the model. We chose learning rates large enough that the networks converged quickly but small enough that the convergence is stable. An example can be seen in Figure 3.14 where the smaller learning rate improves the network performance.

Regularization

Like we have mentioned before the validation error starts to increase as a result of overfitting when you train for too long. To avoid this we decided to use a combination of early stopping and dropout rather than weight penalties. In order to do early stopping effectively we followed the following procedure:

1. Build model/pipeline

2. Train for many epochs (≥ 500)

3. Inspect loss curves and do early stopping

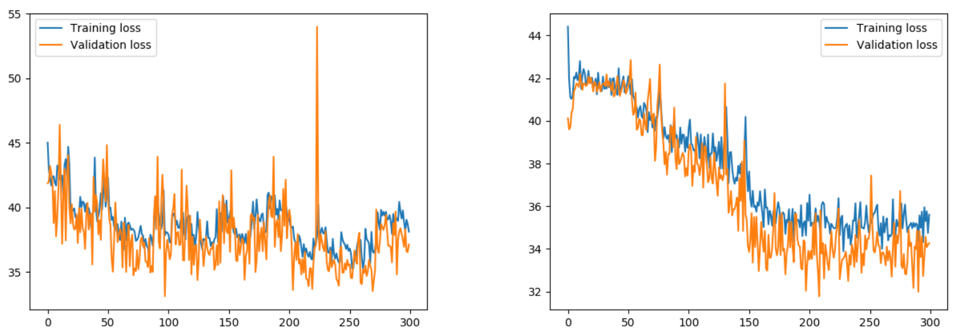


Figure 3.14: The convergence of a network with a large and small learning rate. The smaller rate makes the graph much more stable and improves convergence.

We applied this procedure for each configuration of preprocessing, network setup and hyper parameters that we used. Since we did early stopping independently for every network setup they ended up running for a varying number of epochs. The intermediary results we obtained from Step 2 are omitted from the report, instead we only present the final results we got using early stopping.

During the procedure we used dropout with a probability of 40% for each node in every dense layer. We never used dropout on the output nodes. In their paper Srivastava et al. recommended a probability of 50% which achieves maximum regularization [41] but we decided to stay slightly beneath that since our models only overfit a little.

3.4.4 Performance evaluation

There are many ways to construct a network model and in order to select the best one you need some measure of performance. Previously when fine-tuning YOLO we used the validation error for this, however using the result of a single training run for each network you are comparing is not really sufficient to get a reliable result. It becomes hard to determine which model works the best, especially if the difference in performance is small.

In order to make the comparison of our own models more accurate we applied k-fold cross-validation [20] to get a better estimate of the performance. The idea behind cross-validation is to split the dataset into k parts and then train k different networks using every subset of $k - 1$ parts for training and the remaining part for validation, see Figure 3.15. This results in k networks that were all trained on a slightly different dataset and were all validated on different data. Finally you take the average of all these networks to get an estimate of the performance.

Typical values of k are 5 or 10, however which value you use depends on how much data you have and how long it takes to train the models. We decided to set $k = 5$ to make training faster, after all we are constructing many different models. We used the cross-validation approach when constructing each of the networks in the upcoming sections, starting out with an initial version of the models and then iteratively changing them using the validation results to compare the performance.

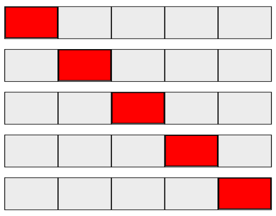


Figure 3.15: An illustration of the data partitioning when using cross validation. Every row corresponds to the dataset for a single network: the white parts are training data and the highlighted parts are validation data.

This method was also valuable when making the final comparison between all our different network models. By using the models that were created during training and evaluating them on our manually annotated test data (see Section 3.3) we could get accurate estimates of their performance. Both means and standard deviations were computed for all the models using the networks from the cross-validation folds.

The main drawback of using cross-validation is that it can be very slow to train multiple models. For instance, when fine-tuning YOLO it could take several days to train a single model which made it infeasible for us to perform cross-validation. It becomes a trade-off between accuracy and time. All our models used cross-validation except for the QP-model.

3.4.5 Bitrate CNN - 1 Dimensional

The bitrate is a time series of some length, consisting of measurements. By splitting it into segments of a fixed size you can view it as a sequence of short signals, each of which might be interpreted by a CNN. In this case the task for the network would be to count the amount of vehicles that appear for the first time in each segment. You could then let it run for all segments of bitrate that you have and sum the result to get a vehicle count.

Selecting a length of the segments is a trade-off between several factors. Longer sequences contain more information, however when running live it also means that you will need to wait longer between each output. Moreover, we should select a length that every video is easily divided into. In the end we decided to use segments of 30 seconds which we thought was a good trade-off.

We constructed a simple CNN model as an initial guess for what a good model could look like. The setup we used corresponds to the one described in Section 3.4.3, with the only exception being an added pass that performs the segmentation of the input before the normalization. We decided to set the learning rate to 0.0001 for this network model since it gave us a good trade-off in convergence speed and performance.

Structure

To create our initial model we studied some examples of bitrate curves to identify its basic features. The three main features were uphills, downhills and plateaus. If you disregard differences in slope steepness you would need at least three filters and one convolution layer in the model to recognize these features. To make the network more robust to tilting we added some more filters to this layer. By adding a second convolution layer it becomes

Table 3.7: The network specifications for the bitrate 1D CNN: the initial version (left) and the final version (right). All the convolutions and max pooling were computed in 1D. The activation functions were set to leaky ReLU for the convolution and dense layers, and to linear for the output layer.

Node	Details
Input	30s per sample
Convolution	8 filters, 5x1 kernel size
Maxpooling	2x1 pool size
Convolution	5 filters, 3x1 kernel size
Maxpooling	2x1 pool size
Dense	10 nodes
Output	1 per sample
Parameters	11 444 weights
FLOPS	22 847

Node	Details
Input	30s per sample
Convolution	8 filters, 5x1 kernel size
Maxpooling	2x1 pool size
Convolution	8 filters, 5x1 kernel size
Maxpooling	4x1 pool size
Convolution	8 filters, 5x1 kernel size
Maxpooling	4x1 pool size
Dense	10 nodes
Output	1 per sample
Parameters	2965 weights
FLOPS	5868

possible for the network to compound the basic features into a notion of hills. This layer should also include several filters to handle multiple hill shapes. Finally we added a dense layer of nodes which served to combine the feature maps from the CNN to accumulate an output for the vehicle count. An overview of this model can be seen to the left in Table 3.7.

We used the leaky ReLU as the activation function for both the convolution layers and the dense layer. The main reason for this was to speed up the training by avoiding problems with vanishing gradients. The ReLU is also cheaper to compute compared to alternatives like the logistic sigmoid, which is ideal since we want to run the model on the cameras. We decided to use the leaky version of the ReLU to make sure the nodes always have an output, otherwise they may become "dead" and always output zero.Lastly, the output node used a linear activation function since we are just interested in accumulating the dense layer into a single output.

The initial model performed quite well on our tests so the focus when improving the network was mostly geared towards reducing the number of weights without loosing performance. This was done by trial and error, through trial and error, by increasing the max pooling and changing the structure a little bit. The final structure of this network is outlined to the right in Table 3.7. Note that both the tables include the number of weights and float operations (FLOPS) that the models use. The FLOPS is the amount of 32bit float operations the processor needs to perform to evaluate the network.

Training

We first trained the model using the initial guess and then iteratively made changes and compared them using cross-validation. In the end we arrived at the final design.

When we had found the final network structure we trained it using both the normalization approaches we discussed earlier as well as varying segment overlap. Segment overlap

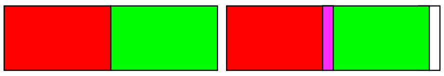


Figure 3.16: An illustration of segment overlap. To the left is a video with two segments and no overlap, and to the right is the same video with 10% overlap per segment. Note that the last 5% of the video becomes unused since it is too small to fit another segment.

means that instead of splitting the video into disjoint segments we let them overlap by some percentage. The rationale behind this was to try to catch vehicles that appeared at the start or end of the videos better. For instance, if we miss a car in the end of a segment we would likely catch it in the beginning of the next. When doing this we have to scale the final vehicle count to account for vehicles that are being counted twice. The scale factor is calculated as:

$$Scale = N_{\text{no overlap}}/N_{\text{overlap}} \tag{3.5}$$

where N_{overlap} and $N_{\text{no overlap}}$ are the amount of segments a video results in with and without overlap, respectively. The reason we use this scaling rather than the more intuitive:

$$Scale = 1 - \text{overlap percent} \tag{3.6}$$

is that the overlap will cause parts of the video to go unused. For instance, using 10% overlap in a video of two segments we would still have two segments, but lose the last 5% of the video since it is too small to turn into a third segment (see Figure 3.16 for an illustration). The two segments have an overlap of 10% of their lengths which corresponds to 5% of the entire video since it contains two segments. We still process the same amount of video, although with overlap, which means that our scale should have a value of one. In contrast, the more intuitive method would have given us a scale of 0.9.

Note that this technique only improves the result if there are enough vehicles at the start and end of the videos to make the average result better. For instance, consider a video with 10 segments and 30% of overlap, this would result in $N_{\text{overlap}} = 13$ and a scale of 0.77. Now, if every vehicle is in the middle of the segments and none are in the beginnings or ends we will never count any vehicles twice since they never appear in the overlapping sections. After scaling this results in an output that is 23% too low. In contrast, if instead every vehicle is in the beginnings and ends of the segments we would count all of them twice but only scale down the result by 23%. The output only becomes more accurate when the vehicles are evenly distributed over the segments so that we can catch the vehicles on the edges. We think that assuming a uniform distribution of vehicles is a realistic approximation given the huge amount of videos we are using.

By trial and error we concluded that using a segment overlap at around 10% gave the best results. We decided to train four different models using combinations of the two normalization approaches we discussed earlier and segmentation overlap at 0% and 10%, see Table 3.8 for an overview.

Table 3.8: The different model variations we used for the 1D CNN and RNN. Normalization was either done over the full dataset or per input sample and segment overlap was either disabled or set at 10%.

Model	Normalization	Overlap
Base	Full	0%
Truncated	Sample	0%
Overlap	Full	10%
Both	Sample	10%

3.4.6 Bitrate RNN

Since the bitrate is a time series the natural approach would be to model it as one. By using an RNN it would be possible to feed the bitrate to the network time step by time step and let the network predict the total amount of vehicles it has encountered thus far. While RNNs in theory can handle time series of any length we settled for using the same segmentation approach as in the previous section and only let the RNN work with time series of a fixed length.

The main reason for using segmentation was to handle inputs that were longer than the training samples; as soon as the input surpasses all our training data in length the RNN would have undefined behavior. There could also be a benefit in keeping the sequence length short because it decreases the problem of vanishing and exploding gradients during training.

Apart from using segmentation the network pipeline was the same as in Section 3.4.3. The loss function was also the same but it was changed to include terms for every time step in the input sample. Previously we have only computed the loss on the final network output but in this case we get an output for every time step, which in turn allows us to compute the loss more accurately.

The Adam optimizer was used but we changed the learning rate to 0.0005 to try to smooth out the loss curves and avoid sudden spikes in magnitude. Figure 3.14 above shows the convergence of the RNN before and after this change. To smooth the curve even further we added exponential learning rate decay that decreased the learning rate by 25% every 10 epochs during training.

However, even though these changes improved the loss curves we still occasionally had spikes that caused them to become undefined. To overcome this issue we implemented gradient norm clipping [33] which simply puts an upper bound on the norm of the gradients before each update step. If the norm is too large the gradients are scaled down to be within our upper bound.

Structure

When we created the initial model for the RNN we first decided to use LSTM nodes instead of simple RNN nodes since they perform better and remedy the problem of vanishing gradients. Using the same line of reasoning as we did in the previous section we decided

Table 3.9: The network specifications for the bitrate RNN: the initial version (left) and the final version (right). The activation function were set to leaky ReLU for the dense layer and to linear for the output layer.

Node	**Details**
Input	1 per frame
LSTM	3 nodes
Dense	10 nodes
Output	1 per frame
Parameters	111 weights
FLOPS	716

Node	**Details**
Input	1 per frame
LSTM	10 nodes
LSTM	10 nodes
LSTM	10 nodes
Dense	10 nodes
Output	1 per frame
Parameters	2281 weights
FLOPS	15 622

to start out with a small model with a single layer of three LSTM nodes. Each of them should allow us to capture uphills, downhills and plateaus, respectively. We also added a dense layer afterwards to combine the LSTM outputs into a vehicle count that could be presented as output. An overview of this model can be found in Table 3.9.

We decided to use leaky ReLU as the activation function in the dense layer in this model as well, for the same reasons as for the 1D CNN. We also used a linear activation for the output node to sum everything up to a single result.

The initial model performed quite poorly on our tests and showed tendencies of slow convergence. In order to get a better result we decided to increase the size of the existing layers and to add several extra layers of LSTM nodes. One way to improve the convergence speed could have been to implement truncated back propagation through time [43] or something similar but we decided to skip this and just let the training run for longer. After some experimenting we decided on the final structure of the model which can be found in Table 3.9.

Training

We used the same approach to training as with the 1D CNN, iteratively making changes and using cross-validation for comparison. When evaluating the final model structure we trained it using both the normalization approaches and segment overlap at 0% and 10% to create four different network pipelines, see Table 3.8 for an overview.

3.4.7 Bitrate CNN - 2 Dimensional

In the beginning of the project we played with the idea of using a 2D CNN but we quickly realized that it does not make much sense when we only have 1D data. In the two previous network models we have split each input segment into chunks of 30 seconds each and processed them as either a signal or as a time series. Just doing the same thing but somehow artificially expanding it to two dimensions should not give us better results, instead we opted for doing some more preprocessing.

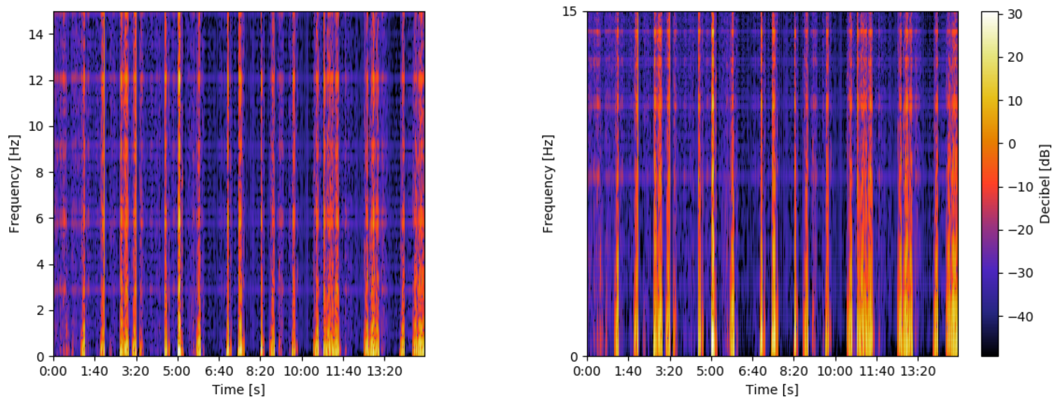


Figure 3.17: Spectrogram (left) and Mel spectrogram (right) of a bitrate segment. Note that the y-axis of the mel spectrogram is non-linear.

We decided to do a frequency analysis and create spectrograms out of the bitrate data. By using this approach we can expand the 1D data to two dimensions in a way that might expose some useful information. While one of the key features of neural networks is their ability to transform the input, like converting it to a spectrogram if that is the best representation, it requires that the network is large enough since it may take several layers of nodes. By instead doing the transformation ourselves we can have a smaller network that only has to focus on interpreting the input. If the spectrogram indeed makes the information easier to parse, then this new network model should perform better than the previous ones.

We made the spectrograms using a combination of the Signal processing package of SciPy [9] and the LibROSA [28] library. The implementation added some steps to the common input pipeline from Section 3.4.3 just before the normalization:

1. Split the videos into chunks of 15 minutes
2. Create spectrogram using `scipy.signal`
3. Create mel spectrogram using `librosa`

The first step was trivial since our videos were recorded with 15 minutes of length by default, however the second and third step needed some changes to the code. There were a lot of parameters to assign when doing this, to give an overview the relevant code is included in Appendix B.

The reason why we chose to transform the curves into spectrograms was as follows. If you study a bitrate curve (e.g. Figure 3.7) you can see both the high frequency noise as well as the low frequency rises and falls which represent moving cars and other interesting things. By creating a spectrogram out of this we can separate the frequencies into a spectrum of bins that the neural network can use to its advantage. For instance it could learn to ignore the high frequency components and just focus on the low frequencies. An example of a spectrogram can be seen to the left in Figure 3.17.

The example shows a nice separation between high and low frequencies but the drawback is that they are linearly distributed. We are more interested in lower frequencies than higher since the amount of new vehicles at each time step is small. To facilitate this we would like to make the distribution non-linear to give more room for the low frequencies, and one way of doing that is by using mel spectrograms. A mel spectrogram is basically a transformation of a normal spectrogram using the mel scale, originally defined by Stevens and Wolkmann [42]. While this method is primarily intended to be used when analyzing sound we still thought it might improve our results so we decided to try it out. The mel spectrogram corresponding to the previous spectrogram can be seen to the right in Figure 3.17.

As you can see the lower frequencies are attenuated in a way reminiscent of a logarithmic scale. This is especially apparent if you compare the horizontal bands between the figures. The resolution is also lower in the new figure since a lot of information about the higher frequencies has been discarded. During the training and cross-validation there turned out to be no difference in accuracy between using spectrograms or mel spectrograms, however the latter were faster to compute due to their lower resolution.

Apart from adding the above steps to the pipeline we kept it the same as the common setup. We left the learning rate at 0.001 for these networks.

Structure

For the previous models we created the initial guess by analyzing the bitrate curve and trying to identify features and corresponding building blocks for the networks. In this case we found that to be much harder since the spectrograms have no obvious features. Instead we opted for doing an educated guess and used two convolution layers with some filters and a dense layer on top. This structure is outlined to the left in Table 3.10. The input size was chosen to fit the size of the mel spectrograms. All the activation functions were set to be the same as for the 1D CNN.

As with the 1D CNN the initial guess performed quite well but used a huge amount of parameters. When improving it we tried to achieve a balance between performance and fewer parameters. As before the changes were done empirically and resulted in one additional convolution layer as well as increased max pooling. The final structure can be seen to the right in Table 3.10, especially note the huge reduction in parameters.

Training

We used the same approach to training as with the 1D CNN; iteratively making changes and using cross-validation for comparison. When evaluating the final model structure we trained it using both the normalization approaches which resulted in two different network pipelines, these correspond to the first two rows in Table 3.8.

3.4.8 QP CNN

The QP network differs significantly from the bitrate networks in that its goal is to locate vehicles rather than counting them, which is a much more complex task. There are many advanced models for localizing objects in images, including YOLO amongst others, but we decided to settle for a much simpler solution. We let the model output a grid which contains zeros and ones, representing whether or not a vehicle is centered on that position

Table 3.10: The network specifications for the bitrate 2D CNN: the initial version (left) and the final version (right). The activation functions were set to leaky ReLU for the convolution and dense layers, and to linear for the output layer.

Node	Details
Input	420x210 per sample
Convolution	8 filters, 7x7 kernel size
Maxpooling	2x2 pool size
Convolution	8 filters, 5x5 kernel size
Maxpooling	2x2 pool size
Dense	10 nodes
Output	1 per sample
Parameters	438 829 weights
FLOPS	877 612

Node	Details
Input	420x210 per sample
Convolution	8 filters, 5x5 kernel size
Maxpooling	2x2 pool size
Convolution	8 filters, 5x5 kernel size
Maxpooling	4x4 pool size
Convolution	8 filters, 5x5 kernel size
Maxpooling	4x4 pool size
Dense	10 nodes
Output	1 per sample
Parameters	9685 weights
FLOPS	19 309

in the input. The output grid was 30x30 cells in size, mapping to the input matrix of size 80x45.

The structure of the input pipeline was quite different from the bitrate networks but still shared some similarities. One of the big differences between the bitrate networks and the QP network is that the former uses a sequence of frames as a time series while the latter looks at each frame in isolation. This means that the QP model had no need for the time line interpolation or I-frame pruning. The input values were also limited to the range $[0, 51]$ which was small enough that normalization was not really necessary.

However, input pruning was still necessary for the QP network and it was even extended to remove frames where YOLO was uncertain in its annotations. This was needed since the QP network was trained directly against them and we wanted to avoid incorrect detections in the training data as much as possible. We implemented this through an extension of the tracker that could flag individual frames as bad if they contained annotations for a vehicle that appeared or disappeared within a span of ± 10 frames. We could then run the modified tracker on the annotations to prune them before we trained the QP model. We hoped that this would reduce the risk of training on incorrectly classified data.

The parsing step was also similar to the previous models, but with some added complications. Since the QP values are a matrix of values for each frame in every video the dataset quickly grew very large. It turned out that we could only load a fraction of a single video at a time to avoid running out of memory. To make this work we had to change the pipeline to continuously load and unload video segments during training. The drawback became a massive increase in training time, sometimes several hours per epoch. We could remedy this slightly by increasing the batch size to around 1000 frames at a time, however it quickly became a balancing act between increasing the batch size and increasing the model complexity with limited memory.

We still needed to randomize the order of the input data to avoid temporal overfitting but this was slightly complicated by our inability to keep all the data in the memory at once. We therefore implemented a two-fold randomization by both randomizing the order of the video segments, before loading them, and the order of the segment frames, after loading them. The resulting randomness was not as evenly distributed as for the bitrate networks but it came close.

Regarding the optimizer we decided to keep using Adam but with a learning rate set to 0.0005. We exchanged the mean squared loss with the cross-entropy loss because this model was performing classification rather than regression.

To use the QP network to actually count the vehicles rather than locating them we added our tracker as a post-processing pass. The tracker does not care about the size of the vehicles, only where their centers are located, which made it easy for us to send our network output to it.

Structure

We did not have a clearly defined initial guess for the QP network model, instead we started out with the same network structure as the 1D CNN used (but widened into 2D). The model structure was developed side by side with other improvements such as input randomization and the exchange of loss functions. Like the previous models we used leaky ReLU for the convolution layers, however, since the model does a classification for each cell in the output grid (car or no car) we used the logistic sigmoid as the activation for the output nodes.

After a while we reached the upper limit for how large the model could be without running out of memory during training. Like we said earlier it was a balancing act between an increased batch size and an increased model complexity. While it is not uncommon to have several hundred filters in a CNN (YOLO for instance has 1024 filters for some of its layers) we had to limit ourselves to using fewer. We could have used heavy max pooling to reduce the model size but that would have had a significant cost in accuracy, especially when the size of the feature maps becomes smaller than the output.

In the end we arrived at a huge model that we trained on around ten million images, see its structure in Table 3.11. Even though the model was so large it still had significant performance problems; likely another approach to processing the QP data would have been needed to get a really good result.

Training

One big difference with the QP network was that we decided against using cross-validation as a means to compare different models. A single pass through the entire input dataset took roughly seven hours for the final model which means that it took over a week per network model to train for just a few epochs. If we wanted to perform cross-validation on top of the training we would have been looking at months before we had any decent results to compare. The decreased training time came at the cost of reliability however, but we did not have much choice in this case.

Due to the long training durations we never reached the point where overfitting would have started to become a problem. In principle you could say that every round of training we did for the QP network was made using early stopping but without following the procedure we outlined in Section 3.4.3.

Table 3.11: The network specification for the final version of the QP CNN. The activation functions were set to leaky ReLU for the convolution layers and to the logistic sigmoid for the output layer.

Node	**Details**
Input	80x45 per sample
Convolution	8 filters, 1x1 kernel size
Convolution	8 filters, 3x3 kernel size
Maxpooling	2x2 pool size
Convolution	16 filters, 1x1 kernel size
Convolution	32 filters, 3x3 kernel size
Convolution	32 filters, 3x3 kernel size
Output	900 per sample
Parameters	50 704 432 weights
FLOPS	101 405 082

3.5 On-device - network on a camera

With the networks constructed our final task was to port them to run directly on the cameras. This involved the following steps:

1. Export the network models
2. Convert to C code
3. Write wrapper program

The first step saves a copy of the models without all the metadata that TensorFlow uses during training. The procedure is regarded as "freezing" the models in TensorFlow and can easily be done by invoking a built-in function. Once the models were exported they could be converted to C code using a tool Axis provided us with. The generated code only contained the networks themselves which meant that we needed to write a wrapper that could initialize the model and pass input to it. It turned out that Axis' tool did not have support for the leaky ReLU activation function so we had to replace it with the ordinary ReLU before exporting. This also unfortunately meant that we had to retrain the models since the new activation function behaves differently.

We decided to only apply this procedure to the 1D bitrate CNN because it was the model that was best suited for the task. We could not use the bitrate RNN since the tool lacked support for RNNs. The 2D bitrate CNN required preprocessing of its input which would have forced us to find or create an implementation of the spectrogram generation that could be deployed to the camera together with the network. Even if we could have found a library for this it would have needed to be exactly equivalent to Scipy and LibROSA to work correctly. Finally, the QP CNN simply was not good enough to be worth the effort, given how it was both large, slow and performed poorly.

In the end we managed to convert the 1D bitrate CNN to C code and write the wrapper for it, however we could not actually run it on the cameras due to firmware bugs that stopped us from uploading any networks to them. The issues will be resolved in the future but they were not fixed in time for the publication of this report. This situation was unfortunate since we wanted to evaluate the performance impact of our models and we could not do that properly without running them. We did try to keep the models small during the network construction but we cannot know for sure if they were small enough. However, by looking at the amount of float operations our networks used (see Tables 3.7 through 3.11) we can see that all of them are well below the limit of 9.6 giga-FLOPS per second that the embedded GPU can handle.

Chapter 4
Evaluation

Previously we discussed how we compared and evaluated our different network models (see Section 3.4.4). In this section we first present the final results of these evaluations followed by a discussion.

4.1 Results

The performance of our bitrate networks is displayed in Figures 4.1 through 4.3, each representing a separate error measure computed on every scene. One standard deviation has been highlighted around the mean errors, signifying how stable they are. For each scene we used a diverse set of 15 manually annotated videos with varying weather conditions, vehicle densities and times of day. By only testing on "common" scenarios (excluding snow and fog) the performance improves by up to five percentage points compared to these results.

There are many ways to quantize performance. We have looked at the total error, the absolute error and the weighted absolute error, following the example of McGowen and Sanderson [29]. Each graph shows the results using a different error measure. They were calculated as follows:

$$total\ error = \frac{\sum(C_i - T_i)}{\sum T_i} \tag{4.1}$$

$$absolute\ error = \frac{\sum \frac{|C_i - T_i|}{T_i}}{N} \tag{4.2}$$

$$weighted\ absolute\ error = \frac{\sum |C_i - T_i|}{\sum T_i} \tag{4.3}$$

where C_i is the number of cars counted in the ith video, T_i is the true number of cars, N is the number of videos and $\sum$ is a sum over all videos. In general the total error is the most lenient of the three as positive and negative mismatches negate each other. The absolute

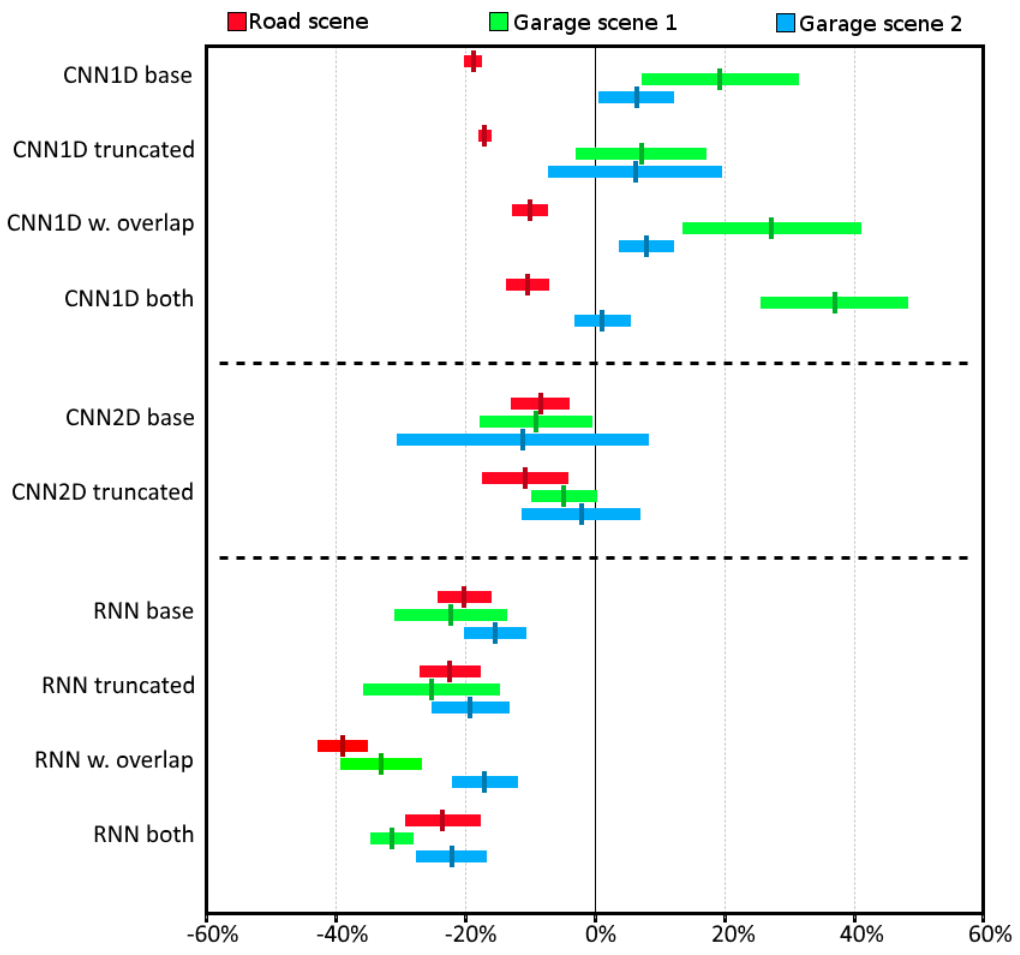


Figure 4.1: The total error for the various bitrate networks. The bars show one standard deviation around the mean error.

error combines positive and negative misses, creating a larger error, and the weighted absolute error takes into account the total number of cars, reducing the impact of large errors in videos with few cars. This becomes evident if we consider an example: with just two videos, one with 10 cars and one with 100 cars, counting one car too many would result in an absolute error of either $\frac{1}{20}$ or $\frac{1}{200}$, depending on if we counted 11 cars in the first video or 101 in the second. The weighted absolute error would give an error of $\frac{1}{110}$ in both cases.

As for the QP network, its performance is summarized in Table 4.1. Because we did not use cross-validation we instead opted to take an average of the last five epochs' results for each model to get a more stable measure of their performance. The variance between individual videos was very high for the QP network, with the best videos having a surprisingly small error of near zero while others reached over 500 percent in all scenes.

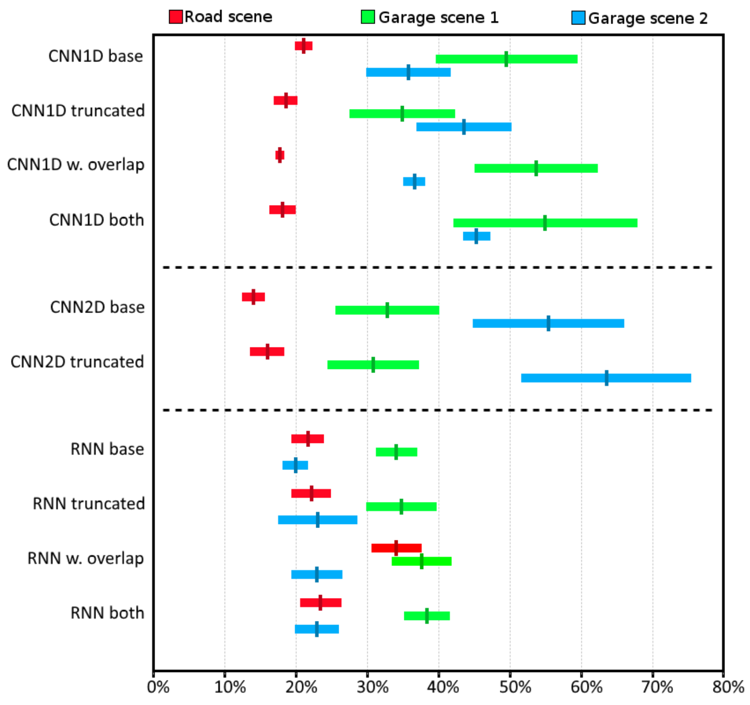


Figure 4.2: The absolute error for the various bitrate networks. The bars show one standard deviation around the mean error.

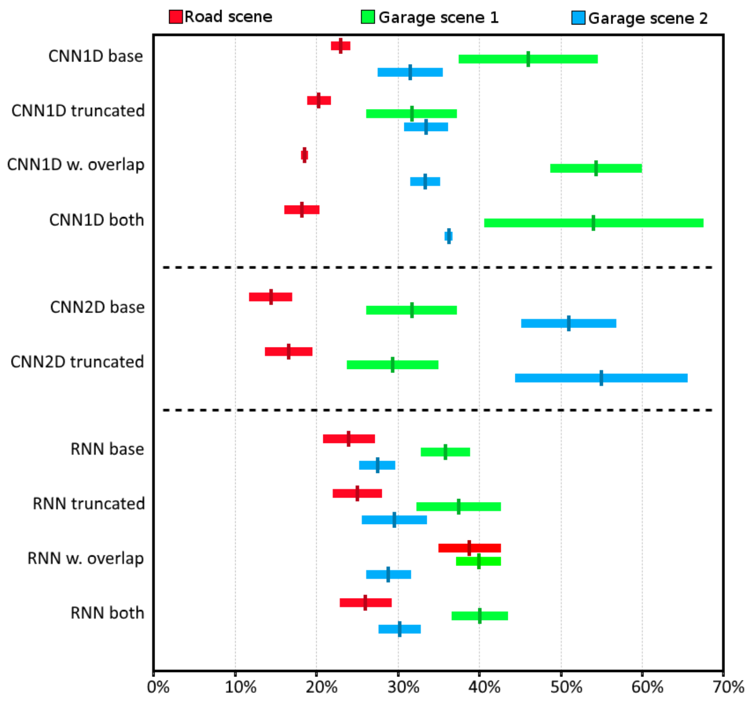


Figure 4.3: The weighted absolute error for the various bitrate networks. The bars show one standard deviation around the mean error.

Table 4.1: The error of the QP network for the different scenes taken from the last five epochs.

Dataset	Total mean	Abs. mean	Weighted mean
Road scene	375.70	413.07	386.87
Garage 1	47.36	240.17	245.00
Garage 2	64.74	82.10	78.42

4.2 Discussion

The following subsections will focus on analyzing our results, evaluating our process and listing future work. We also make comparisons to a few similar methods and note how their techniques could be used to improve our results.

4.2.1 Interpreting the results

Apart from the mean values the Figures 4.1 through 4.3 also contain a single standard deviation as horizontal bars. Since the deviations are computed over the errors the different cross validation folds achieved on the test set they give us a measure of how well the different models generalized from the training data. When the bars are small it means that all the network folds performs equally well despite having trained on slightly different training data. When they are large the network folds vary a lot in performance which might indicate that the model is poorly suited for the task.

Generally we can see that the bars are smaller for the road scene compared to the garage scenes which is a sign of overfitting. This suspicion is further substantiated by the fact that the garage scenes have worse absolute and weighted absolute errors for the 1D and 2D CNN models. The overfitting probably stems from the fact that the models were developed mainly using the road scene, mainly because we had too few recordings from the other scenes at the time. Perhaps the models are too simple to handle all the complex noise that exists in the garage, like people walking in and out.

Looking at the mean values in figures we see that the 2D CNNs have low total errors in all three scenes but in the absolute error graph it looks a lot worse, especially for the second garage scene. This means the network makes many errors but they negate one another so that the total error remains small. In contrast, the 1D CNNs have similar absolute errors but larger total errors, meaning that they tend to overshoot (in the case of the garage scenes) or undershoot (the road scene) the true vehicle count.

Looking at the graphs we can see that the RNN networks tend to have the same performance across all scenes, regardless of error measure. This means that they are flexible enough to handle all our different scenarios with similar results. Even though the variability among the RNN networks is small they are all ending up with vehicle counts that are too low. Obviously the models have ended up being non-optimal which could be due to several reasons.

One reason could be that the bitrate curve tends to be noisy. When the signal suddenly peaks it could both be due to noise but also due to vehicles in the image. The difference is easy to spot when you study the entire sequence but the RNN is only exposed to one frame at a time. Furthermore, the LSTM memories in the network will likely accumulate errors as the time goes by, causing further drift from the true count the longer the sequence is. In contrast the CNN networks are memoryless and only need to count the features it sees in the image. They also process the entire sequence at once which helps them avoid classifying temporary peaks as noise. This probably helps them achieve better results.

Another reason for its poor performance could be that the loss function is too penalizing. The loss is computed by taking the mean squared error at each time step between the network output and the true vehicle count. The problem with this is that the true function is a discontinuous step function. The vehicle count is only increased when the tracker detects a vehicle, which might be delayed compared to when it really appeared. The RNN is unable to increase its output in discrete steps since the output is a continuous function which will cause it to be penalized during the transition. This problem is probably worsened by the inexact annotations that YOLO and the tracker generates, causing heavy penalization even when the RNN behaves correctly.

Since the total errors of the RNNs are all less than zero in the figure we can draw the conclusion that the network detects too few vehicles compared to what the videos contain. The issues we mentioned could act as a kind of regularization that makes the network more "careful" in its prediction, thus causing this result. However, in order to be really sure what is wrong with the model we would need to make a deeper analysis. One simple way of doing this could be to plot the network output over time together with the expected output to see what the correlation looks like.

The QP network is in a league of its own: it performs very poorly and often overshoots the target by several hundred percent. Considering that you can easily locate vehicles by looking at the QP-values manually a neural network *should* be able to perform the same task with decent performance. That this is not the case we think is a result of the network design being unsuited for the task. Because the output is in relatively large discreet steps it becomes harder for the tracker to find movement than if the output had been continuous or scaled up to individual pixels in resolution. Changing the shape of the output layer to have the same aspect ratio as the input (rather than a square) could potentially also improve the results. Finally, it is worth considering changing the output entirely: if the network currently splits a vehicle into several clustered outputs it is possible that the accuracy would improve if the network were asked to output bounding boxes, more in vein with how YOLO works, rather than just the center of the found vehicle.

In general the truncated 2D CNN appears to be the top performing network, having very little error on all three scenes, but if the absolute error is prioritized it is instead the truncated 1D CNN that performs the best. Given that these networks are intended to run for longer periods of time (enough to train/fine-tune the network and then start counting) the absolute error is less interesting than if we were to consider shorter time spans. Should the camera be mounted for long the total error will likely be negligible, although the accuracy would be less reliable if a subset of the data is requested, such as the load during peak hours.

Averaging at 89% accuracy on the road scene and 95% and 98% on the garage scenes, our truncated 2D CNN is certainly not bad but it fails to reach the level of accuracy of more traditional methods. Pneumatic road tubes, for instance, have an accuracy of 99% (although that number is likely averaged over a larger time frame [29]). Using recent machine learning techniques on video others have reached a level of 97% accuracy [6], outperforming our network on the road scene but being closer to our results in the garage.

4.2.2 Scenes and generalization

As described earlier, our networks were originally based and evaluated on the road scene. It would not be very surprising then if they generally performed better on that scene than on the garages. This also seems to be the case, especially when comparing it with the first garage scene, which is the more complex of the two. We also noted that YOLO performed worse in the garage, giving us less accurate training data which, combined with larger noise from people entering and exiting the garage, could account for these scenes being harder for the networks.

Another major factor is the difference in vehicle density. No 15-minute video in the garage had more than ten cars entering or leaving (more than half of the videos were actually empty of cars). This means that missing or counting just one extra vehicle has a larger effect on the error, which could explain why they have such large absolute errors and why the standard deviations are so large. By pruning away video segments with no events in them the accuracy may be improved but care must be taken not to skew the input too far. The pruning could be done by discarding all videos where YOLO finds no vehicles. It could be taken even further by pruning away individual video segments where the count was unchanged.

The biggest difference between the scenes is how the tracker, and to an extent YOLO, has been modified to fit that specific scene. Unlike when training the networks this part was done manually and as such does not scale well to systems with many cameras or with dissimilar traffic situations. In order to generalize the system and make it fully automated this must be solved and there are several ways to to that. One way would be to find a good balance where YOLO and the tracker performs decently well in most scenarios. Another is to have a few default configurations and choose one that fits the current scene upon installation. The last option is to replace the tracker with something more general and powerful. This might be useful if the scenes are more complex as well, such as corners or intersections where the direction of travel changes.

A different approach entirely would be to limit the camera placement. In all three scenarios the camera is placed in such a way that a lot of noise and unwanted features are included in the videos. In the road scene we masked out the parking areas but left several roads next to the main one. While the one furthest away rarely disturbed the scene as vehicles on it were too small to be counted most of the time the nearest one was a larger issue, both for the automatic and manual annotation, and it created difficult scenarios where some cars were half-occluded by the edge of the image. By placing the camera in an optimal position, for example pointing down directly over the road, and fine-tuning YOLO on this angle, the accuracy could potentially be significantly improved while noise would be reduced to a minimum. This could also limit the disturbances caused by harsh weather conditions, such a heavy snowfall or rain on the camera, and make sabotage harder. Finally, a top-down view would avoid the problem with vehicles occluding each other which could cause issues for both the tracker and the networks.

Setting the tracker aside we believe the network models should generalize fairly well, especially the bitrate RNN since it has a small difference in performance between scenes. We could then record from different locations and once enough training material has been gathered all that is needed is to train the model and upload it to the camera. While the scene may be different from our testing scenes, the network model should be able to adapt as its task is still the same and the training was done for the new location. It is, of course, possible that the networks do not generalize as well as we believe, making our results less significant. Because we have only examined video from two unique locations (the second garage scene is very similar to the first) we cannot know if other scenes, such as intersections or roads with heavy pedestrian traffic, would perform differently. However, because of how different these two locations are we are fairly confident that new scenes would share similarities with at least one of them and perform similarly.

4.2.3 Adapting to variations in data

According to Medina et al. [30], adverse weather conditions can severely damage the accuracy of video based systems, with snow in particular causing up to 90% false positives in their tests. This was something we also noted on when evaluating YOLO but it did not seem to affect our own networks' performance to any large degree. While it usually varied some between videos, these variations were seemingly random: one video with rain could perform poorly while the other performed really well, then for the next network it was the other way around. There is one large exception to this observation however: the networks generally performed worse during foggy weather and in the evenings, with the worst case being 80% missed vehicles in a fog test. We believe the reason for this is that the bitrate is drastically lower in these scenarios which makes cars stand out much less from the background noise, especially when the input normalization is applied. It should be relatively easy to see if this is indeed the case. By applying a defogging algorithm to the image before compression the image should become sharper which in turn raises the bitrate and, if we are correct in our assumption, improves the result.

Varying training conditions

When we trained our networks we were fortunate enough to get a wide variety of recordings, covering most possible weather combinations. This allowed us to create a robust training dataset, better than what would likely be used for a real camera. This, then, leads to the question of how important a varied dataset is.

To get an idea of its importance we can alter which videos are included in the training set and then redo the training. In Table 4.2 we have listed a few scenarios and the results from training the basic version of 1D CNN on them. Because of the time-span of this project we have a limited number of videos with clear, sunny weather and rain so we have focused on day/night, clear versus snowy weather and a large versus small dataset.

There are a few interesting things to take from these results. First, a note on the datasets themselves: they are somewhat smaller than the actual training sets (about 300-500 videos) with the minimal sets being exceptional as they have exactly nine overcast videos each. We see that the results are similar to those of the original training set. Apart from the minimal set, training only on overcast videos yields the largest difference from the baseline, possibly because this is the "base level" weather condition, but it is questionable whether or not the difference is significant, especially given the small difference in absolute error. We see that training on a very small dataset has severe detrimental effects on the performance. In

Table 4.2: Error of the base 1D CNN given different training datasets.

Dataset	**Total mean**	**Std.**	**Abs. mean**	**Std.**	**W. mean**	**Std.**
CNN1D base	-18.70	1.36	21.22	1.24	23.08	1.18
Road, day only	-17.95	2.44	20.54	0.92	22.54	1.07
Road, overcast	-14.42	2.16	21.12	1.29	22.16	1.02
Road, snow	-17.39	1.99	24.05	1.33	23.63	1.2
Road, minimal	621.36	100.7	795.74	124.92	639.18	101.75
Garage, minimal	137.48	25.46	156.01	29.65	139.73	23.61

this case the network models had a very hard time to adapt: while the road scene network had an error of below 50% (which is still very large) on most of our test videos, the error on our two snow videos were over 3800%, drastically raising the mean error. The garage network did not have a single large error but was simply worse across the board, likely because the scene has less variation between videos.

While these experiments are too small in scope for us to draw any solid conclusions, they suggest that the exact composition of the training set is less important than we had originally thought, given that there are enough samples in it. This in turn means that the dataset we picked for training was likely a good choice and should not skew our results too far in any direction.

The outlier QP test

Curiously, the QP network actually performed much better in snowy weather than in any other scenario among our manual test data, achieving an error generally between five to ten times lower than the other scenes (and reaching only 1.4% in the best case). Given how snowfall affects the QP-values, as seen in Figure 4.4, we thought snow would have a severe negative effect on the results, not the other way around. It is hard to say if this is a coincidence or not without more test data. We did annotate additional snow videos previously not used for training but those videos, with medium snowfall, had an error on par with or just slightly lower than the other test videos. This could be reasonable as comparing the images in Figure 4.4 we see that the video with medium snowfall looks like a mix between the other two and does not have quite as extreme features as the heavy snowfall video.

One idea as to why the network suddenly improves is that the vehicles are filled in with a solid color during heavy snowfall because there is more movement in the scene. The reasoning behind this idea is the assumption that solid areas are easier to locate and recognize for the network than areas where parts of the vehicles are discarded, as seen during clear weather. Another reason might be that the contrast is too great during clear weather and the brighter background works to normalize the input. It might also be a result of the QP-values generally being smaller during heavy snowfall, which might push them over a threshold limit inside the network. If either of the latter two were the case it would mean our assumption that normalization was unnecessary was wrong but they also indicate that medium snowfall should affect the accuracy more than it does. Lastly, it is

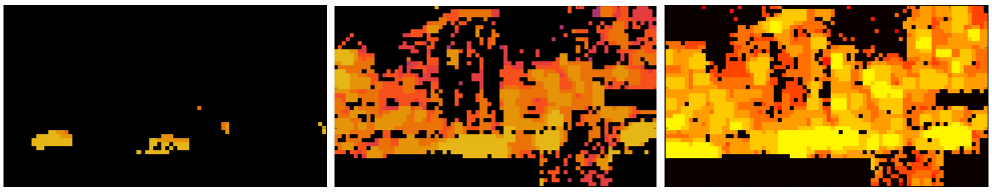


Figure 4.4: QP-values during clear weather, medium snowfall and heavy snowfall. The left and middle images have two cars each while among the brightest areas in the right image there are three cars hidden.

also possible that the network just fails to find as many vehicles as it usually does. The QP network has a tendency to drastically overshoot the true count so if the snow causes it to miss the majority of vehicles (that are not necessarily real) that it would normally find the final result could end up being more realistic, although it is hard to imagine it having *such* good results if it was just missing vehicles at random.

To fully understand what sets the heavy snowfall video apart we would need to further examine similar videos. Another alternative would be to examine the network's output closely and compare it to YOLO's. This would be easier with a decent visualization tool. It is possible that the QP network could perform much better if the input was modified slightly, or the better performance is merely a coincidence.

4.2.4 Improving our system

After training our networks and generating the results we reviewed the system we had constructed to try to find things to improve. In the end we recognized a number of changes that could be made to potentially make the system more robust.

The first change relates to the data collection. We have stated several times that our dataset included corrupt videos that we could not use for training, either because a lot of frames were missing or because QP vales could not be extracted. In order to deal with this we had to try to filter them out when we were setting up the training pipeline. Another solution could have been to add a filtering pass directly after the recording step. Doing so would have spared us a lot of work later, not only could we have removed our filtering from the network pipeline but we would also have saved all the time we spent annotating broken videos. As it turned out, even some of the videos we annotated manually were broken. Potentially our networks could have performed better as well since they would have had cleaner data to train on.

Secondly, a major part of our pipeline was the YOLO network and the tracker. Calculating YOLO and the tracker's mean error the same way we did for the bitrate networks reveals that it had a total error of roughly 2% and an absolute error of just below 5% on the road scene. It should be possible to take this into account when calculating the network's loss to improve its performance but since these numbers are unlikely to be consistent across different scenes that might be unrealistic. To increase their accuracy we could have replaced YOLO with its newest version, YOLOv3 [38], which was released halfway through the thesis project, or we could have replaced our tracker with a more powerful existing model. Since every error introduced this early in the pipeline will accumulate

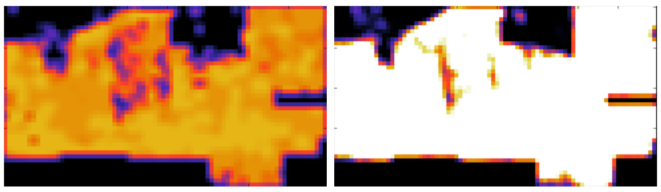


Figure 4.5: QP-values during heavy snowfall after applying gaussian smoothing (left) and both gaussian and temporal smoothing (right). It is completely impossible to discern any vehicles.

through the rest of the system we may have seen a large improvement in our final results.

Another way of improving YOLO could have been to fine-tune it on a dataset of our own. Like we have mentioned previously. YOLO had problems identifying vehicles in the garage scenes due to the camera angles. If we had constructed a dataset containing images of vehicles taken from many directions before fine-tuning we could have increased the accuracy, particularly for the garage scenes. However, this would have taken a lot of time.

Thirdly, there were several decisions during the network construction that could have been explored more or done differently. All networks used video segments of either 30 seconds or 15 minutes in length. These numbers were more or less arbitrarily chosen and it is possible that other settings could have yielded better results. We also normalized the input over a single sample when using the second normalization approach, but we could have chosen to normalize over two or more samples instead. Both of these decisions were made by educated guesswork since we did not have time to test all combinations.

The design of the networks themselves could also have been more thorough. We put a lot of effort on implementing all the preprocessing algorithms to tune the results at the cost of experimenting with network structures. To keep the scope of the thesis limited we did not have time to delve deeply on the designs, there probably exists better ones.

Fourthly, there are some small details in the network pipelines for the different models that we probably should have done differently. To begin with, the spectrogram generation for the 2D CNN used a lot of parameters that we had to assign and these were also set using educated guesswork. There was simply too much work involved in trying out different settings and running the entire network pipeline to evaluate the results. However, we did some tests to see how much impact the size of the spectrograms had on the performance before deciding which to use.

To improve the QP network we could both have used normalization, like we have said previously, and applied some more preprocessing. In Section 3.2.3 we showed Figure 3.8 where we had applied temporal and gaussian smoothing to the QP values. One idea is to use the same type of filters when training the network instead of just feeding it the raw data, this would make the vehicles more solid which might help the QP network. The drawback would be that the data from the snow videos would end up as a solid blob of color, making them completely useless. Figure 4.5 shows the effect of the smoothing on the video with heavy snowfall (from Figure 4.4).

To improve the RNN network there are several things we could have done differently. First off, we might improve the results by suppressing the noise using some sort of input averaging. When the input is smoothed it is easier for the network model to infer knowledge

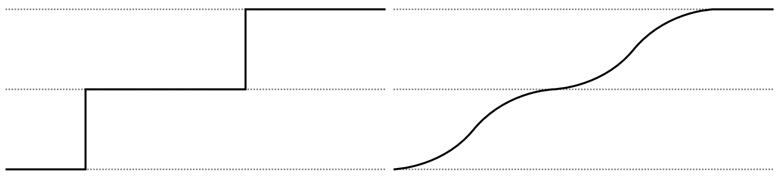


Figure 4.6: The true vehicle count versus time (left) and the smoothed equivalent (right). The dashed lines represent different vehicle counts.

from the slope of the curve. We could also preprocess the true vehicle count to make the loss function less penalizing, for instance by using a smooth step function instead of the current step function, see Figure 4.6 for an illustration. With the smoother curve we get rid of the discrete behavior, which the RNN cannot match, and also allow for some leeway in when the vehicles are detected, which will make inaccuracies in the annotations less penalizing.

Lastly, we could expand our set of manually annotated test data. Ideally we would have liked to have multiple videos for each weather condition to get a deeper insight into how our models handle them. A larger test set would also make our measures of the average accuracies better overall. One way of generating a lot of annotations could be to outsource it to someone else, however as soon as we hand the videos over we would breach of the privacy of those who were recorded and it would be harder to maintain control over quality.

4.2.5 Sources of error

There are several factors that potentially could jeopardize the validity of our results and our analysis, they are summarized in the following list:

- Overfitting in the tracker
- Lack of network visualization
- Lack of sufficient testing data
- Lack of confidence intervals

The first problem is related to how the tracker was fine-tuned. We realized that the same dataset was used both to validate the tracker improvements and to estimate the generalization performance of the networks. This could cause overfitting and make our results too optimistic, giving biased estimates of the true performance. The issue would be more significant for the road scene since the tracker was built using its test set as the benchmark. It also applies to the garage scenes, as the tracker was tweaked for those scenes, but to a lesser extent as fewer modifications were made. Since we became aware of this problem quite late during the thesis project we did not have time to create entirely new test sets, however many new videos had already been added to the existing ones since the fine-tuning was done. We came to the conclusion that if there is an optimistic bias in our results it is likely quite small, in particular since the test sets have been extended with new videos.

The second problem lies in our lack of visualization of the networks, or more specifically of their behavior. Much of our discussion is based in educated guesses and assumptions from the observations we have made, however we could still be wrong on several counts, most notably regarding the reasons behind the performance of the models. A way to provide some more evidence for the analysis could have been to visualize the network behaviors. While this could be done quite easily for the QP CNN and bitrate RNN networks it would have been much harder for the 1D and 2D CNN networks. The former two have outputs that can be displayed as bounding boxes or a time series while the latter two just provide a single number each, making them a lot harder to visualize. We could have ventured deeper into visualization but we decided against it to limit the scope of the thesis project.

The third problem regards our limited amount of manually annotated test data. Like we have said earlier we would ideally have liked to have multiple videos for every weather condition to make comparisons more accurate. The smaller the test sets are the likelier it is that the results are due to their peculiarities rather than due to the true generalization performance. We have tried to make this problem less likely by continuously adding new videos to the test sets but it might still have an effect on our results.

The final problem is related to how we present our results. While we do present both the means and standard deviations (see Figure 4.1 through 4.3) it would have been much more interesting to present them in the form of confidence intervals. Then we could have applied basic statistical analysis to conclude if the performance of our various models was statistically significantly different. The reasons why we decided not to use confidence intervals were several but there were two main problems.

Firstly, in order to compute confidence intervals you usually collect data from several independently and identically distributed random variables. In our case the random variables were the different cross-validation folds which are heavily dependent in both their training data and test inputs. Moreover we could not be certain that they truly are identically distributed, after all they are the result of different training runs.

Secondly, since we were trying to construct the confidence intervals for our different error measures we ran into the problem of truncated distributions. Both the absolute and the weighted absolute errors have a lower bound at zero percent and this must be taken into account to not get incorrect confidence intervals.

During our research we only came across one good way of computing the confidence intervals that avoids these problems. By using a technique called bootstrapping [49] we can generate new datasets from our training data by randomly selecting samples (with replacement). These new datasets would become more independent than our cross-validation folds due to the stochastic nature of bootstrapping. When we have generated a lot of new datasets, say 100 for instance, we can apply statistical theory to all of these models to finally arrive at confidence intervals.

However, generating 100 network variants per model would be prohibitively expensive, we neither had the time nor the resources to do that. Since we found no other way around this issue we decided not to use confidence intervals.

When we searched for related work we have realized that there are few papers around that use confidence intervals or similar statistically sound methods. It could be an indication that doing so is a hard problem that takes a lot of effort to solve.

4.2.6 Automating and decentralizing

In order to make it possible to deploy our models to any camera, regardless of where it is positioned, we would need to make our system general. There are two approaches to do this, the first is to make a single model and deploy it everywhere while the second is to train the model for every single camera. The second approach should generally be the best since it makes the model tailored to its scene. In order to make it work every camera must have a training phase where data is first collected and then used to fine-tune the model. A drawback is that the training data cannot be recorded in advance so it might take a long time to deploy the model after the camera has been installed.

When recording the training data it will be hard to get a sufficient distribution of weather conditions, day and night as well as varying traffic intensities. This might be the biggest issue when deploying the model to different cameras. In our case we had to record for quite a while before we could start training the networks due to the lack of variety in the scenes. In a real scenario it would mean that the camera has to be mounted and recording for some time: hours, days or months, depending on how varied the scene is and how frequent the vehicles are. Looking at our scenes we see that the garage would take weeks to get the same amount of vehicles as the road scene does in a day. On the other hand the road scene changes a lot more as the weather and time of day varies, resulting in many different scenarios. If there is not sufficient data to cover all these scenarios the network models may not perform well at all. There is also a risk that the distribution of the data becomes skewed, causing overfitting to certain conditions. A smart algorithm would be needed to create a varied dataset from a distorted one.

To make the system viable when you have a large number of cameras it needs to be automated. This makes it a lot harder to tweak the settings of each network model to their scenes. For instance, our road scene had many uninteresting areas with a lot of action which we wanted to hide and the garage scenes had vehicles driving past outside the gates which caused problems. Our approach to handling this were to either mask out the areas completely or to ignore YOLO's detections outside some area of interest. Both of these solutions were applied manually which works well for a few cameras but becomes problematic for large systems. Instead the training procedure should ideally be able to detect a region of interest on its own, possibly by sampling which parts of the scene contains the largest or most frequent changes. In the worst case it could be left as a manual setup step but in that case it must be very easy to do. Because YOLO's detection rate seems to worsen if there is a very large number of vehicles in the scene some masking might be required in order to get good results.

In order to train the network it could be connected to a centralized unit for the duration of the training period. During that time it would send the video to this unit for annotation and training, something that is too heavy for the camera to perform itself. When this is done, the central unit sends the trained weights to the network which can then work autonomously. That way the time while it has to send potentially sensitive data over the Internet is minimized.

4.2.7 Related work

There exist a plethora of vehicle counting networks, many of which have an accuracy in the 90%-100% range. Earlier we compared our results to those of Biswas et al. [6]. In their paper they used two methods for counting vehicles in video, a background subtraction method (BSM) and an OverFeat framework, and tested their systems on various conditions, much like we did. While the BSM struggled, the OverFeat framework performed very well. They implemented it using a CNN for feature extraction and logistic regression for training. Rather than tracking vehicles throughout the scene they defined regions of interest (ROI) where, if a vehicle was detected, the count would be increased.

The use of ROI for detecting vehicles is something that could improve our networks as well. By defining areas where only a single vehicle can appear at a time there is no need for an advanced tracker system which could greatly improve the accuracy of both our QP network and the training data for all our networks.

A similar approach was made by Liu et al. [25] who used a variant of BSM where they had separate algorithms for different scenarios, such as daytime, nighttime or very dense traffic, to increase the overall robustness of the system. With this approach they achieved an accuracy of around 99% in various conditions at a speed of 67.33ms on average. While this is far from real-time they argue that only every third of the frames needed to be analyzed, allowing the network to take more time processing each frame. However, this speed was achieved when running the network on a personal computer, not a camera, and as such it is very unlikely that it could compare to our networks: a modern high-end graphics card has roughly a thousand times more computing power than our cameras[45].

We have suggested using an alternative to our tracker. One such alternative is SORT [5] or DeepSORT [51], the latter being an extension of SORT which uses a CNN for re-identification. While both of these networks were built to detect people this can changed by training them on vehicles instead, though it requires a suitable training dataset. (which was our primary reason for not using them). It would also be possible to use the system devised by Liu et al. above to annotate our training data, removing both YOLO and tracker entirely while keeping the speed and size benefits of our networks.

4.2.8 Future work

We have already discussed a wide range of future work in the previous sections; both changes to our existing pipeline such as replacing YOLO or the tracker, fine-tuning more or doing a wider search of the parameter space (Section 4.2.4); model visualization (Section 4.2.5) and new features such as using ROI (Section 4.2.7). In this section we discuss some additional future work that did not fit in the earlier sections.

We have focused on using the bitrate and the QP-values as the basis for training our models, however, there is a variety of other metadata that could have been used instead. Previously we have mentioned SNR and motion vectors as two other options but there are countless others that both stems from H.264 and the Axis Zipstream system. While QP and bitrate are relatively easy to extract from videos and have an intuitive interpretation we cannot be sure that they result in the highest performance possible. Motion vectors in particular contain information that might be very useful for the QP network. When a vehicle passes through a scene the motion vectors will be heavily affected by its movement.

They would not only make it possible to determine where a vehicle is but also in which direction it travels. This could be very helpful to a network model since it can be used to rule out movement that goes along the wrong direction (such as a mass of people crossing the road).

Another important change that should be researched is how well our models work with multiclass problems. To limit the scope in our thesis we counted every type of vehicle as a single class. The next step is to split the vehicle count into a spectrum of vehicles. We expect the performance to worsen for multiple classes since it is a much harder problem but it would be very interesting to research. When looking at multiple classes it would also be possible to abandon vehicle counting altogether and start counting something else instead, such as people or animals. Our algorithms would more or less function in the same way for other types of classes.

Finally, we also think that using unsupervised learning would be an interesting approach to counting vehicles. Unsupervised learning is when you perform machine learning without any labels (annotations in our case). In contrast we have performed supervised learning throughout our thesis since we had annotations for all our training data. The idea is to let the machine learning algorithm do clustering instead of regression and classification. For instance, we could just feed in the bitrate graphs and the algorithm would split it into pieces and group together those that are similar to each others. If you let it run on the entire dataset you would end up with a set of clusters, each representing a type of phenomena. Ideally you would get one cluster for cars, one for trucks, etcetera... When running it live you could use the same clusters to identify vehicles in the metadata and count them. An added advantage to unsupervised learning is that it can also be used to detect anomalies. If it encounters some data that is very far from all existing clusters it could be flagged so that camera operators could take a look at them manually.

Chapter 5 Conclusions

We have presented a new method of analyzing video data that is both fast and lightweight. We conclude that it is possible to accurately detect and count vehicles by looking at video metadata using simple neural network models. While the accuracy is lower than for other network models, the massive gain in performance is worth the trade off for smaller systems that are unable to run the larger networks. We have seen that bitrate works especially well as a metric and note the potential of using QP-values as well, although the latter needs more research to evaluate its feasibility. The potential for future works is high as there are several kinds of metadata we have yet to try, with motion vectors being a prime candidate. There is also the possibility to expand upon the bitrate and QP networks to account for multiple vehicle classes, or to use unsupervised learning for anomaly detection in bitrate. Finally, our method has great potential to improve through various optimizations, such as using better annotation tools or by tweaking the learning process.

Bibliography

[1] Ffmpeg. `https://ffmpeg.org/`. Accessed 2018-02-09.

[2] Axis Communications AB. Zipstream. `https://www.axis.com/se/sv/technologies/zipstream`. Accessed 2018-03-01.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Y. Bengio, P. Y. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994.

[5] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3464–3468, 2016.

[6] D. Biswas, H. Su, C. Wang, J. Blankenship, and A. Stevanovic. An automatic car counting system using overfeat framework. *Sensors*, 17(7):1535, 2017.

[7] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[8] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[9] SciPy Community. Scipy. `http://www.scipy.org`. Accessed 2018-04-26.

[10] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[11] Encoding.com. 2017 global media formats report. `https://www.encoding.com/resources/#white-papers`. Accessed 2018-03-05.

[12] International Organization for Standardization. Iso/iec 14496-10:2003. `https://www.iso.org/standard/37729.html`. Accessed 2018-03-02.

[13] G. Hinton, N. Srivastava, and K. Swersky. Lecture 6d: a separate, adaptive learning rate for each connection. 2012.

[14] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91:1, 1991.

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[16] Docker Inc and Docker Project. Docker. `https://www.docker.com`. Accessed 2018-04-16.

[17] T. Jayalakshmi and A. Santhakumaran. Statistical normalization and back propagation for classification. *International Journal of Computer Theory and Engineering*, 3(1):89, 2011.

[18] Project Jupyter. Jupyter. `http://jupyter.org`. Accessed 2018-04-16.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[20] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1137–1145, 1995.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114, 2012.

[22] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[23] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[24] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2):4–22, Apr 1987.

[25] F Liu, Z. Zeng, and R. Jiang. A video-based real-time adaptive vehicle-counting system for urban roads. *PLOS ONE*, 12, 11 2017.

[26] Long. Convolutional neural network. `https://medium.com/@Aj.Cheng/convolutional-neural-network-d9f69e473feb`. Accessed 2018-05-03.

[27] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[28] B. McFee, C. Raffel, D. Liang, D. PW. Ellis, M. McVicar, E. Battenberg, and O. Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, pages 18–25, 2015.

[29] P. McGowen and M. Sanderson. Accuracy of pneumatic road tube counters. `https://pdfs.semanticscholar.org`. Accessed 2018-04-26.

[30] J. Medina, M. Chitturi, and R. Benekohal. Effects of fog, snow, and rain on video detection systems at intersections. *Transportation Letters*, 2010.

[31] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814, 2010.

[32] NVIDIA. nvidia-docker. `https://github.com/NVIDIA/nvidia-docker`. Accessed 2018-04-16.

[33] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1310–1318, 2013.

[34] L. Prechelt. *Early Stopping - But When?*, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[35] J. R. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations, ICLR 2018, Vancouver, Canada.*, 2018.

[36] J. Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[37] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[38] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[39] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6):1137–1149, 2017.

[40] I. E. Richardson. *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons, Chichester, West Sussex PO19 8SQ, England, 2004.

[41] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[42] S. S. Stevens, J. Volkmann, and E. B. Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3):185–190, 1937.

[43] I. Sutskever. Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*, 2013.

[44] K. Sühring. H.264/avc reference software. `http://iphome.hhi.de/suehring/tml/`. Accessed 2018-04-27.

[45] Techpowerup. Nvidia geforce gtx 1080. `https://www.techpowerup.com/gpudb/2839/geforce-gtx-1080`. Accessed 2018-06-06.

[46] thtrieu. Darkflow. `https://github.com/thtrieu/darkflow`. Accessed 2018-02-22.

[47] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[48] Vcodex. An overview of h.264 advanced video coding. `https://www.vcodex.com/an-overview-of-h264-advanced-video-coding/`. Accessed 2018-03-05.

[49] R. Wehrens, H. Putter, and L. M.C. Buydens. The bootstrap: a tutorial. *Chemometrics and intelligent laboratory systems*, 54(1):35–52, 2000.

[50] Wikipedia. Inter frame. `https://en.wikipedia.org/wiki/Inter_frame`. Accessed 2018-03-05.

[51] N. Wojke, A. Bewley, and D. Paulus. Simple online and realtime tracking with a deep association metric. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3645–3649, 2017.

Appendices

Appendix A
A complete list of annotated videos

Table A.1: All manually annotated videos used for testing. All videos were 15 minutes long but frame rate varies with location.

Location	**description**	**vehicle count**
Road (30fps)	Sunny day	109
Road	Sunny day, snow covered ground	99
Road	Sunny day, dense traffic	203
Road	Clear evening	177
Road	Thick fog	179
Road	Clear night	68
Road	Rainy day	79
Road	Rainy day, alternate	100
Road	Snowy day	139
Road	Overcast day	112
Road	Overcast day, snow covered ground	185
Road	Overcast day, dense traffic	204
Road	Overcast day, dense traffic 2	208
Road	Thick fog evening	93
Road	Snowy day 2	93

Table A.2: All manually annotated videos. All videos were 15 minutes long but frame rate varies with location.

Location	**description**	**vehicle count**
Garage (25fps)	Morning	4
Garage	Night	5
Garage	Sunny day	5
Garage	Evening	4
Garage	Late evening	4
Garage	Night 2	4
Garage	Night 3	4
Garage	Day 2	4
Garage	Day 3	3
Garage	Snowy day	2
Garage	Snowy evening	4
Garage	Night 4	1
Garage	Night 5	0
Garage	Rainy night	2
Garage	Day 4	3

Table A.3: All manually annotated videos. All videos were 15 minutes long but frame rate varies with location.

Location	**description**	**vehicle count**
Garage 2 (25fps)	Snowy day	4
Garage 2	Late evening	2
Garage 2	Night	1
Garage 2	Day	2
Garage 2	Misty day	4
Garage 2	Evening	1
Garage 2	Day 2	0
Garage 2	Night 2	5
Garage 2	Sunny day	4
Garage 2	Mixed day	1
Garage 2	Day 3	1
Garage 2	Day 4	9
Garage 2	Night 3	2
Garage 2	Snowy evening	3
Garage 2	Sunny day 2	3

Appendix B

Source code for spectrograms

The following code was used to generate the spectrograms. The first chunk consists of various settings that must to be defined in order to generate the spectrograms, while the last chunk creates the spectrograms.

```
# Define sample frequency (frame rate of our recordings)
sample_frequency = FRAME_RATE

# Define spectrum window size (how many input points are used at a time)
window_size = 128

# Define spectrum window type
window = signal.get_window('hann', window_size)

# Define how much the window computations overlap
overlap_fraction = 0.5

# Define FFT size
fft_length = 1024

# Set output units to be V^2 where V is the input units
scaling = 'spectrum'

# Set desired amount of mel bins
mel_bins = 128


# Create spectrogram
f, t, spectrogram = signal.spectrogram(
        x=np.array(input),
        fs=sample_frequency,
        window=window,
        noverlap=overlap_percent * window_size,
        nfft=fft_length,
        scaling=scaling)
```

```
# Create mel spectrogram
mel_spectrogram = librosa.feature.melspectrogram(
        S=np.abs(spectrogram),
        n_fft=fft_length,
        hop_length=window_size * (1 - overlap_percent),
        n_mels=mel_bins)

# Compute output in dB scale
output = librosa.power_to_db(mel_spectrogram, ref=np.mean)
```

Appendix C
Table of results

Table C.1: Mean and standard deviations for total, average and weighted average error, listed for all bitrate networks on the road scene.

Dataset	**Total mean**	**Std.**	**Abs. mean**	**Std.**	**Weighted mean**	**Std.**
CNN1D base	-18.70	1.36	21.22	1.24	23.08	1.18
CNN1D truncated	-16.87	1.06	18.65	1.64	20.42	1.49
CNN1D w. overlap	-9.90	2.84	17.86	0.63	18.65	0.42
CNN1D both	-10.28	3.27	18.21	1.79	18.36	2.16
CNN2D base	-8.32	4.58	14.12	1.62	14.50	2.63
CNN2D truncated	-10.63	6.71	16.07	2.40	16.71	2.89
RNN base	-20.05	4.13	21.74	2.27	24.06	3.17
RNN truncated	-22.27	4.76	22.23	2.74	25.12	2.98
RNN w. overlap	-38.79	3.9	34.17	3.5	38.88	3.81
RNN both	-23.37	5.83	23.55	2.92	26.13	3.22

Table C.2: Mean and standard deviations for total, average and weighted average error, listed for all bitrate networks on the first garage scene.

Dataset	Total mean	Std.	Abs. mean	Std.	Weighted mean	Std.
CNN1D base	19.40	12.14	49.62	9.91	46.06	8.57
CNN1D truncated	7.25	10.04	34.96	7.39	31.77	5.55
CNN1D w. overlap	27.35	13.77	53.82	8.64	54.37	5.61
CNN1D both	37.05	11.43	55.05	12.91	54.11	13.44
CNN2D base	-9.07	8.70	32.89	7.25	31.77	5.59
CNN2D truncated	-4.64	5.07	30.88	6.43	29.46	5.62
RNN base	-22.14	8.63	34.19	2.86	35.91	3.02
RNN truncated	-25.07	10.56	34.85	4.93	37.53	5.15
RNN w. overlap	-32.87	6.30	37.72	4.21	40.01	2.77
RNN both	-31.23	3.39	38.41	3.20	40.13	3.44

Table C.3: Mean and standard deviations for total, average and weighted average error, listed for all bitrate networks on the second garage scene.

Dataset	Total mean	Std.	Abs. mean	Std.	Weighted mean	Std.
CNN1D base	6.52	5.80	35.86	5.96	31.57	4.01
CNN1D truncated	6.31	13.43	43.62	6.66	33.55	2.70
CNN1D w. overlap	8.03	4.23	36.72	1.54	33.43	1.85
CNN1D both	1.28	4.38	45.40	1.89	36.32	0.49
CNN2D base	-10.99	19.50	55.50	10.67	51.05	5.83
CNN2D truncated	-2.03	9.21	63.61	11.90	55.04	10.54
RNN base	-15.31	4.82	20.02	1.80	27.57	2.23
RNN truncated	-19.12	6.05	23.09	5.56	29.63	3.99
RNN w. overlap	-16.86	5.07	23.02	3.60	28.92	2.77
RNN both	-21.99	5.46	23.05	3.07	30.34	2.60

Appendix D
Work distribution

The work done during the course of this project has been evenly split. Of this paper, Sebastian has written the background chapter, the sections on fine-tuning YOLO, our network designs and the training of said networks. Mattias has written the introduction, the Approach chapter (excluding the above mentioned parts) as well as the results section. Both authors wrote the discussion together.

Sebastian created the bitrate networks, our extraction and visualization tools for bitrate and QP-values, wrote our recording script and fine-tuned yolo while Mattias set up and integrated YOLO into our pipeline, created the tracker and its evaluation tools, made the QP network and the compilation/visualization of our results.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-05-31

EXAMENSARBETE Vehicle Counting using Video Metadata
STUDENT Mattias Gustafsson, Sebastian Hjelm
HANDLEDARE Yuan Song, Jörn Janneck
EXAMINATOR Flavius Gruian

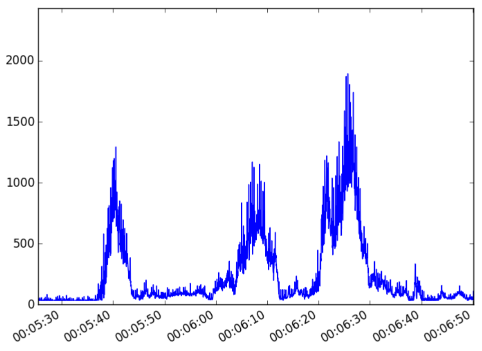
Vi räknar bilar - utan att titta!

POPULÄRVETENSKAPLIG SAMMANFATTNING **Mattias Gustafsson, Sebastian Hjelm**

Att hitta och följa fordon i video kräver mycket datorkraft vilket är svårt att få plats med i små, integrerade system, så som vägkameror. Genom att istället mäta storleken på förändringarna i videons bilder kan vi avgöra när bilar passerar till en mycket lägre beräkningskostnad.

En modern övervakningskamera kan spela in HD-video med 30 bilder per sekund. Det betyder att det varje sekund finns 100 miljoner nya bildpunkter som måste tolkas av kameran för att avgöra om där finns några bilar. Även om man bara tittar på en bråkdel av punkterna så har en dator mycket svårare att förstå vad den ser än vad en människa har vilket gör sökprogram både stora och långsamma.

Vi har löst detta genom att titta på videons metadata, information om själva videon, istället för dess innehåll. Eftersom metadatan är miljoner gånger mindre än bilden så går det extremt snabbt att få ut ett resultat. I figuren till höger ser vi en graf över hur mycket videons bilder förändras över tid. När scenen innehåller rörelse ger det mycket förändringar och vi får en kulle i grafen. Med hjälp av maskininlärning tränade vi en dator på att känna igen mönster i förändringen och kunde med en träffsäkerhet på över 95% avgöra hur många bilar som passerat. Resultaten påverkades inte av att andra saker, som cyklar och människor, också fanns med.



Figur: *Bilderna förändras mer när fordon passerar framför kameran. Vi kan enkelt se när nya bilar dyker upp men utan en dator är det svårt att veta att spikarna innehåller en, två, respektive tre bilar.*

Det finns ytterligare en fördel med vår metod: den anonymiserar videon. Eftersom vi inte behöver titta på videons innehåll undviker vi att känslig information läcker ut. Detta gör det lättare att använda kameror för datainsamling utan att det kränker folks integritet.

I framtiden kan tekniken utökas för att klassificera och räkna olika fordonstyper. Den kan också användas för att hitta speciella händelser i videor genom att leta efter avvikande mönster i graferna. Det skulle till exempel göra det möjligt för kameran att varna om en bil kör för fort eller om djur har tagit sig ut på vägen. Slutligen så skulle vår teknik kunna användas för att övervaka andra scener än vägar, exempelvis för att räkna människor som rör sig i ett varuhus.