University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Computer Science - Data Science	Spring semester, 2021 Open / ~~Restricted access~~
Writer: Bernt Andreas Eide	Bernt Andreas Eide …………………………………………… (Writer's signature)
Faculty supervisor: Erlend Tøssebro External supervisor(s): Karl Skretting	
Thesis title: Detecting and classifying vehicles entering and exiting a tunnel	
Credits (ECTS): 30	
Key words: Deep Learning Tensorflow Machine Learning Convolutional Neural Networks Region Proposal Systems Background Subtraction Computer Vision	Pages: 69 + enclosure: 12 Stavanger, 15.06.2021 Date/year

University of Stavanger

Master thesis (MSc)

Detecting and Classifying Vehicles entering and exiting a tunnel

Supervisor:
Prof. Erlend Tøssebro

Authors:
Bernt Andreas Eide

External Supervisor:
Prof. Karl Skretting

Preface

This is a thesis in Data Science at the University of Stavanger. The main topic is detection and classification of vehicles entering and exiting a tunnel. Two main methods have been developed, both utilizing transfer learning. A fully fledged region proposal system, and a simpler system utilizing a vanilla Convolutional Neural Network (CNN) and background subtraction have been created.

I would like to thank my two supervisors, Erlend Tøssebro and Karl Skretting for their support and input throughout the thesis work, and of course my dear friend Heloise Fonseca, who took the time to give feedback along the way.

Abstract

Tunnel safety is an increasing concern for the road administrative authorities. In this thesis, a vision based surveillance system is developed as a safety measure. The main purpose of this surveillance system is to detect and classify incoming and outgoing vehicles, thus rescue workers will have an idea about which vehicles reside inside the tunnel at any given time.

The thesis builds its foundation on two previous theses from previous students at the University of Stavanger. In the previous work, the main bottleneck has been detection and classification during challenging lighting/illumination and/or weather conditions. To counter these challenges, transfer learning has been used to create a more solid model, and with explicit data that has been created for challenging lighting conditions. New techniques for detection has also been tested, such as region proposal networks.

For the region proposal system, the Single Shot Detector (SSD) MobileNet v2 pre-trained model has been used. This model has been trained on the popular Common Objects in Context (COCO) dataset, which consist of at least 200,000 annotated images, and span across 90 different classes.

As for the vanilla CNN, MobileNet v3 has been used. This model has been trained on the ImageNet dataset, which consist of 14,197,122 annotated images.

Both models use transfer learning, thus, less data is required to train and build a solid model. 5560 images have been collected and annotated for the training of these models, the images contain roughly 21,325 objects spanning across five classes (car, truck, person, bike, bus).

Classification rates converge fast with almost every configuration, however, there are some challenges separating some of the minority classes. Practical considerations have also been taken into account, a system like this would most suitably be deployed on a low-cost micro-controller, such as an Internet of Things (IoT) device.

Contents

1 Introduction

1.1 Background

In Norway, there are more than a thousand road tunnels, some in varying conditions. The national road authorities have tunnel maintenance as a very high priority. A major problem in long road tunnels is ventilation, to direct the fumes out of the tunnel. In the case of a fire, it would be challenging to ventilate the tunnel, which poses a significant danger to the people inside it [1]. To aid rescue workers, it would be necessary to have a tunnel surveillance system that can keep track of how many vehicles reside in the tunnel at any given time. Additional information like the vehicles estimated speed, position, and if they carry dangerous material would also be useful for the authorities. This registry will allow the authorities to plan ahead, and efficiently rescue any people stuck inside the tunnel.

For the surveillance, a vision based system will be proposed in this thesis. In this system, the main task is to detect and classify vehicles that enter and exit an arbitrary tunnel. This system will naturally have to be quite resilient, due to radical weather conditions and illumination changes. With said system in mind, the road authorities will have an estimate of how many vehicles reside in the tunnel at a given time, thus this information will allow rescue workers to plan ahead, and potentially save lives. The longer the tunnel, the more important it would be to be able to plan ahead.

1.2 Object Recognition

Big Data and advancements in the field of Artificial Intelligence (AI) has made traditional image processing tasks like object detection and classification more reliable [2]. With the use of Artificial Neural Networks (ANN) and Convolution, it is possible to do very accurate image classification. Furthermore, this can be combined with a region proposal network to extract and classify regions of an image. Such network could make it unnecessary to perform traditional background subtraction and do complex pre and post processing to extract the objects of interest.

Traditionally, the detection part has been done by performing background subtraction to extract the objects in motion (foreground) and then applying additional post-processing to remove shadows and noise. A bounding box is then fitted around each foreground object and, finally, classification is done for each object using a classifier of some sort. The classifier can be a CNN, or even a Support Vector Machine (SVM), in either case, feature maps are fed to the fully connected layer in the CNN and otherwise directly in the SVM classifier. If sufficient data has been used to train the classifier, it will be possible to predict the class of a given vehicle with high probability of success (accuracy).

1.3 Previous Work

Two previous students at the University of Stavanger (UiS) have published similar work in their theses. Their work have given the foundation for this thesis, and the main goal is to improve and/or find more efficient ways to do detection and classification under different weather and lighting conditions.

Eirik Atlekt Thomessen introduced the use of [3]

- CNN for image classification (using specific collected data, with 3 classes: person, car and truck).
- Gaussian Mixture Model (GMM) together with background subtraction for object detection.
- Kalman filter for tracking a linear model.
- Predict traffic behaviour based on occlusion or detection rate.
- Created a surveillance system app in PyQt for real time simulation.

And Erik Sudland, who proposed the initial work, based itself solely on classic image processing techniques like [4]

- Detect object by background subtraction, using a GMM.
- Uses the color space YCbCr and Hue, Saturation, Value (HSV) to make the detection less sensitive to lighting variations, such as vehicle headlights, floodlights, etc.
- Detect vehicle front based on the initial detection of the registration plate.

- Run Histogram of Oriented Gradients (HoG) plus Trace Transform on the detected vehicle front, used as a unique structure together with the color to recognize the vehicle on exit.
- Unable to detect vehicle fronts of trucks, classifies vehicle as light or heavy.
- Unable to detect vehicle at all in poor lighting conditions.

The main issue in both of these theses is detection and classification under various lighting, weather, and vehicle occlusion conditions.

1.4 Problem Statement

Training a large convolutional neural network can be challenging because of the large amounts of data it would require to generalize well to unseen data. Transfer learning eliminates this challenge to a certain extent, as it allows the user to use the weights and structure of a model which has been trained on a large dataset. It is still necessary to do model training, but the data amount needed to train an efficient model is significantly reduced. In this thesis, there was no existing data available from the road administrative authorities, so manually collecting and annotating data was necessary. The amount collected would not have been enough to build a solid model, which is why transfer learning is highly leveraged in this thesis.

2 Theory

This chapter will focus on explaining the theory behind the techniques used in the thesis.

2.1 Artificial Neural Networks

In 1943 Warren McCulloch and Walter Pitts modeled a simple neural network using electrical circuits, this was done in order to describe how neurons in the brain might act [5]. Hebbian learning was introduced in the late 1940s, a theory which pointed out that neural pathways are strengthened each time they are used, which essentially allows for learning [6].

In 1958 Frank Rosenblatt proposed the perceptron [7], which is essentially the simplest form of a neural network. There are no hidden layers, a single or multiple inputs, a neuron and an output. The neuron is activated with a non-linear activation function. It was proven that for single-layer neural nets you could learn any task that its parameters could embody, in a finite amount of training cycles [8].

In 1969 Minsky-Papert published a book which revolves around Frank Rosenblatt's work on the perceptron. The book focuses on providing proofs, linear separation problems, and thoughts on simple and multilayer perceptrons [9].

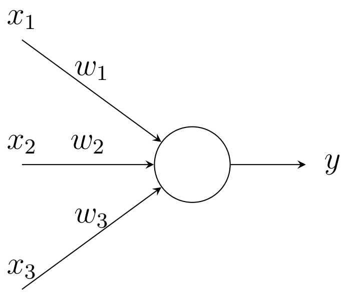


Figure 1: Perceptron.

In 1994 the Multi Layer Perceptron was proposed by Simon Haykin [10], which involves an input layer, a hidden layer and output layer. This type of network is also known as a feedforward network because connections between the nodes do not form a cycle (either forward or backward). The inputs are fed through the network and are activated at each neuron in every hidden layer before being propagated to the next layer. And for training the network, backpropagation is done to minimize the error of the network. Backpropagation is only necessary if the predicted values are far off from the target values, in such a case the weights in the network will be updated to minimize the loss. These properties allow the network to distinguish data that is not linearly separable [11].

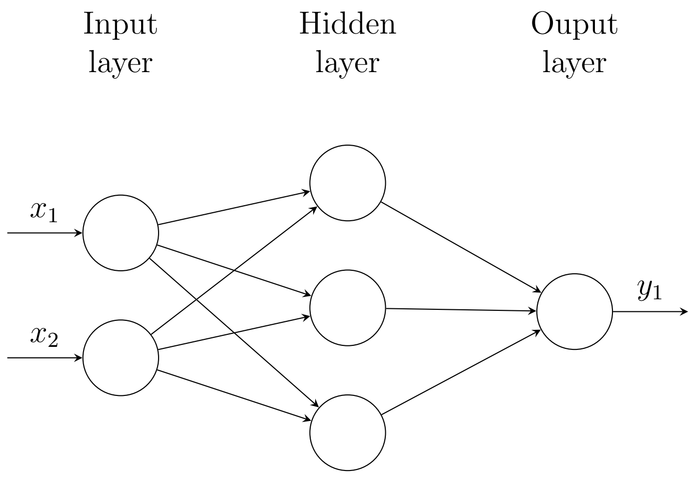


Figure 2: Multi-Layer Perceptron.

A neural network is designed to replicate the biology in our brain, a neuron that is fully connected to other neurons. A fully connected layer in the case of neural networks. What makes neural networks powerful is that they can generalize anything with enough data, an input X is fed into the network, and is propagated through the layers where each layer applies an activation function to the input times its weights plus a bias.

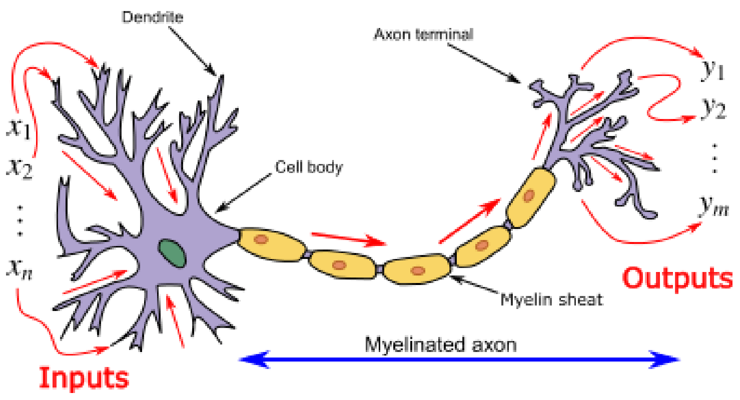


Figure 3: Neuron [12].

The activation function is used to apply non-linearity throughout the network. Linear activation functions can also be used, but are generally avoided because back propagation will not be able to find any relation between the previous input and the output of the activation function when the weights are being updated (because the derivative is constant). Some common activation functions are:

- tanh, ranges from -1 to 1. $\frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
- sigmoid, ranges from 0 to 1. $\frac{1}{1+e^{-x}}$
- Rectified Linear Unit (ReLu), ranges from 0 and up. $max(0, x)$
- Softmax, $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ where K is the number of classes. Generally used for the output layer in a multiclass network. The range is between 0 and 1, for each class (returns a vector of probabilities, where each probability represents the probability for the input X to belong to the class K_i, this vector sums to 1).

To compute the output for a neural network in the case of figure 1,

$$\hat{y} = f_x(Wx + bias) \tag{1}$$

where $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$, $W = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$ and f_x is some activation function (bias is 0 in this example).

This is known as forward propagation, the input X is fed into the network and propagated through the layers until it reaches the output layer. In a supervised setting, which normally is the case, there will be target labels. After the forward propagation, a loss function will be used to tell the network how it is doing. If the predicted values are deviating far off from the target labels, backward propagation will have to be initiated. This procedure will compensate for the deviation by updating the weights in a way which will minimize the loss. The process of actually minimizing the loss is known as gradient descent. This algorithm will try to minimize the target versus predicted loss, and will do so by finding the local minima in every scenario. The gradients are leveraged during this procedure, thus the activation function should be differentiable.

If the network is large (many nodes to train and respective weights to adjust) and the gradients have a very rough terrain, it will pose a challenge, which is why learning rates are used to speed up or slow down the rate of the descent. To prevent the gradient descent algorithm from getting stuck, adaptive algorithms may be used. Adaptive learning rates are used with algorithms such as Root Mean Square Propagation (RMSProp) and Adaptive Moment Optimization (Adam), these usually provide good convergence in comparison to fixed learning rates. Adaptive algorithms will adjust their learn rate depending on various conditions. In addition, initializing the weights randomly before training may also help prevent correlation.

RMSProp was discovered by Geoffrey Hinton [13], the algorithm keeps a moving average of the squared gradient for each weight. Adam leverages first order gradients, and tries to compute individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients [14]. There are plenty of other adaptive algorithms to choose from, their effectiveness may vary depending on the problem at hand. Neural Networks can be tuned in many ways, and certain *configurations* can be very problem specific.

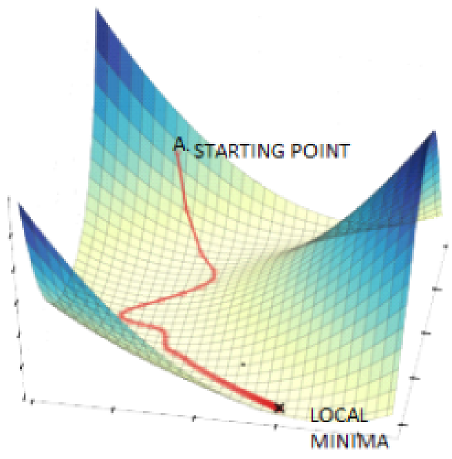


Figure 4: Gradient Descent in action [15].

In figure 4 we see how the gradient descent traverses further down the terrain. This scenario is not always the case, if the terrain is not as smooth it may get stuck, in which case the algorithm may find many local minimas rather than a global minima.

In the case of figure 2, without bias the computations would be:

$$y = f(W^2 f(W^1 x)) = f(\begin{bmatrix} w^2_{11} & w^2_{21} & w^2_{31} \end{bmatrix} f(\begin{bmatrix} w^1_{11} & w^1_{21} \\ w^1_{12} & w^1_{22} \\ w^1_{13} & w^1_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix})) \tag{2}$$

for the forward propagation, then the loss is computed using the Mean Squared Error (MSE):

$$J(\theta) = \frac{1}{2} \sum (y - \hat{y})^2 \tag{3}$$

If the error is greater than a threshold, do backwards propagation to update the weights and in turn hopefully reduce the error (depends on gradient descent efficiency). The derivative of the function f is used ($f^{'}$) and the *sensitivity* is computed for each layer as we go backwards.

$$\delta^2 = diag(f^{'}(z^2))(y - y^2)^T \tag{4}$$

Because the network only has one output, the above computation will be a scalar (*diag* will diagonalize the result, if not a scalar):

$$\delta^1 = diag(f^{'}(z^1))(W^{2^T} \delta^{2^T}) \tag{5}$$

$$\delta^1 = \begin{bmatrix} f^{'}(w^1_{11}x_1 + w^1_{21}x_2) & 0 & 0 \\ 0 & f^{'}(w^1_{12}x_1 + w^1_{22}x_2) & 0 \\ 0 & 0 & f^{'}(w^1_{13}x_1 + w^1_{23}x_2) \end{bmatrix} \begin{bmatrix} w^2_{11} \\ w^2_{21} \\ w^2_{31} \end{bmatrix} \delta^{2^T} \tag{6}$$

the deltas can be computed, μ is the learning rate:

$$\Delta W^2 = \mu \delta^2 y^{1^T} = \mu \delta^2 \begin{bmatrix} f(w^1_{11}x_1 + w^1_{21}x_2) & f(w^1_{12}x_1 + w^1_{22}x_2) & f(w^1_{13}x_1 + w^1_{23}x_2) \end{bmatrix} \tag{7}$$

$$\Delta W^1 = \mu\delta^1 x^T = \mu\delta^1 \begin{bmatrix} x_1 & x_2 \end{bmatrix} \tag{8}$$

Finally, the weights can be updated by simply adding the deltas to the old weight matrices, $W^1 = W^1 + \Delta W^1$, and $W^2 = W^2 + \Delta W^2$. This completes one round of updating the weights, the network would continue this procedure until the loss is at a satisfactory low value. In the real world, however, it would be preferable to use different activation functions, dropout, and possibly an adaptive learn rate.

It is also important to note that due to the nature of these computations, it is possible to run into two problems known as the vanishing and exploding gradient problem. A vanishing gradient converges to zero because it is multiplied with too many small numbers, while an exploding gradient is converging towards infinity for the opposite reason. These problems are normally solved by choosing an appropriate activation function for the desired layers, and randomly initializing the weights. These issues are usually detected by noticing that the loss function is sky high or never changes at all.

As for over- and under-fitting a neural network, with too little data the model will not be able to generalize. And if the model is trained to rely on certain nodes it may overfit. Dropout is used to prevent the network from being too dependent on a single node, dropout makes sure that an arbitrary node in an arbitrary layer will be reset with a given probability. This will also naturally help prevent certain correlation tendencies between certain nodes.

Underfitting can be addressed by resampling with replacement (bootstrapping), augmenting existing data to add additional data. Augmenting can be in the form of simple rotations, horizontal/vertical flip, shear, zoom, tilt, etc. Each feature vector generated for these various modifications would be different, thus the model would be given more diverse samples of the same objects. This can be beneficial not only in the case of underfitting, it generally helps increase the models ability to generalize.

Training will be done over several epochs, an epoch is the process of training the neural network with all the training data. Training over several epochs will continue to adjust the weights further. Each epoch may introduce a batch of samples at a time, this is done to speed up the training process, this will leverage the Graphical Processing Unit (GPU). This training procedure naturally leads to a high accuracy on the train set (the network tries to fit the data perfectly over time), in which case it is important to have an additional set for testing. The test set should contain samples that are not present in your training set, to simply verify whether or not your model is able to generalize. At the end of each epoch the model will apply the test set, if the test accuracy deteriorates over time, early stopping should be initiated. Early stopping is a simple measure to prevent overfitting, when the test accuracy worsens and the train accuracy converges towards 100%, stop at the earliest stage (epoch) where the test accuracy is at a feasible point. This data is usually used to construct a learning curve, for each epoch record the train and test accuracy, plot and analyze the results when training is finished. To further validate the model's ability to generalize, a validation set can be used in the end but this set is only used once.

Another approach which can be used to prevent overfitting is regularization, in neural networks the regularization will be applied to the cost function (equation (3)), this function would be expanded to:

$$J(\theta) = \frac{1}{2}\sum(y-\hat{y})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}||w_j||_1$$

for L1 regularization, and:

$$J(\theta) = \frac{1}{2}\sum(y-\hat{y})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}||w_j||_2^2$$

for L2 regularization. Where $||w_j||_1$ and $||w_j||_2^2$ is the L1 (max) and L2 (frobenius) norm. This is helpful because λ can be used to adjust the model fit complexity (large weights will be penalized). Smaller λ values will provide little change in the model fit, large values of λ will provide a very simple fit (underfit), ideally the λ value should be discovered through hyperparameter tuning. Try a selection of λ values and see which works best for the problem at hand.

Label smoothing is another useful technique which can increase the networks generalization by making it less confident in its predictions. Normally, the network will be taught that it should predict the target labels such that the class label in question was predicted with 100% accuracy. Label smoothing changes this behavior, rather than assuming that the target is at a confidence of 100% it will assign a $\frac{value}{K}$ to every class, and subtract $\frac{K-1}{K} * value$ from the target. This implies that every class has a small probability to be predicted as the target, rather than only one class, in other words, the ”stubbornness” of the network is reduced.

2.2 Convolutional Neural Networks

Convolutional Neural Networks combine convolution, pooling and a fully connected layer. A CNN is normally used for image classification tasks, such as vehicle classification, facial recognition, detecting cancer in computed tomography (CT) scan images, etc.

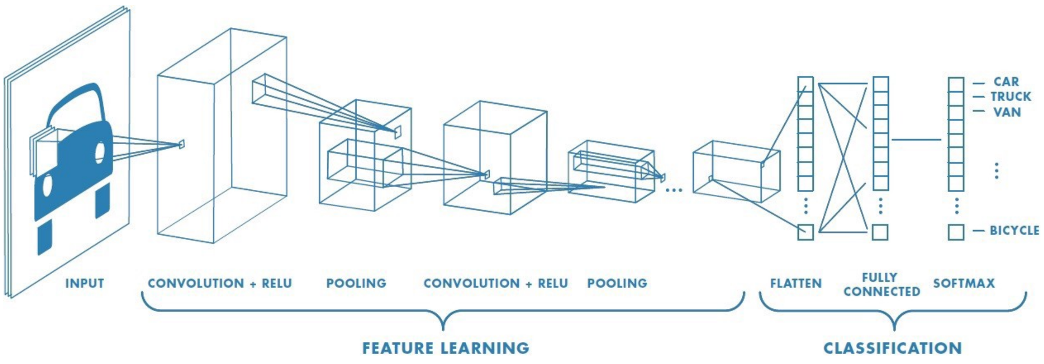


Figure 5: CNN Architecture [16].

Figure 5 shows an in-depth overview of a CNN, prior to supplying an image to the CNN the image will be resized for consistency. The size varies, smaller images may lose important structure and information, but in most cases it is not a major problem. Numerous filters will be convolved with the input image, these filters try to extract features from the image (features such as edges, for example). Pooling can then be applied to the result in order to only extract the higher, lower or mean values, thus the image will be *downsampled*. This is convenient because there will not necessarily be useful features extracted for every region in the input image. Additionally, supplying a large feature vector to the fully connected layer can pose several challenges. When pooling has been applied, the result is flattened into a row vector, and will be supplied as an input to a neural network. Certain activation functions may be used for applying non-linearity, these functions can be applied to the results gained from sliding the filters over the input image.

Convolution is defined as

$$(f * x)(x) = \int f(z)g(x-z)dz$$

overlap between f and g is measured when one of the functions have been flipped and shifted by x. But in the case of CNNs, these operations are discrete, thus a sum is adequate

$$(f * x)(i) = \sum_a f(a)g(i-a)$$

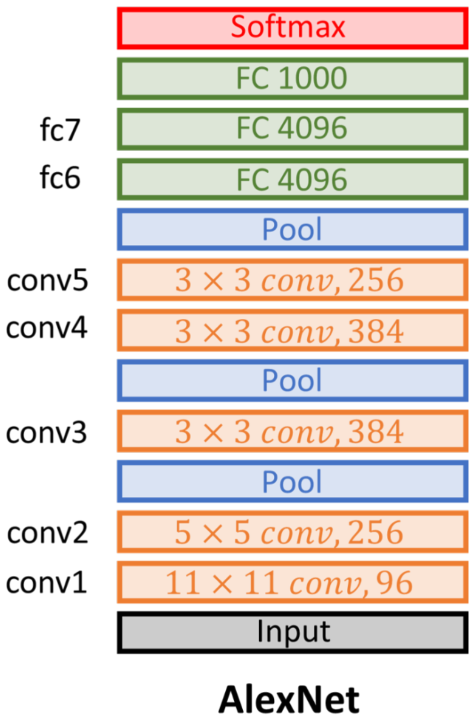


Figure 6: AlexNet Architecture [17].

Using figure 6 as an example, the input size of the image is $224 \times 224 \times 3$, channels Red, Green, Blue (RGB). Computing the size of the output from applying the first filter (11×11) with a stride of 4 and zero padding:

$$\frac{W-K+2P}{S}+1 = \frac{224-11+0}{4}+1 = 54 = (54 \times 54) \times 96 \text{ filters} \tag{9}$$

Input $\begin{bmatrix} 15 & 25 & 10 & 10 \\ 25 & 15 & 10 & 35 \\ 15 & 60 & 10 & 50 \\ 45 & 15 & 45 & 85 \end{bmatrix}$, filter $\begin{bmatrix} 2 & 4 \\ 3 & 7 \end{bmatrix}$, stride 1 and 0 padding. This will result in a 3×3 matrix

$$\begin{bmatrix} (15*2+25*4+25*3+15*7) & \cdots & (10*2+10*4+10*3+35*7) \\ \vdots & \ddots & \vdots \\ (15*2+60*4+45*3+15*7) & \cdots & (10*2+50*4+45*3+85*7) \end{bmatrix}$$

$$= \begin{bmatrix} 310 & 205 & 335 \\ 575 & 320 & 540 \\ 510 & 520 & 950 \end{bmatrix}$$

The 2×2 filter is multiplied with each respective 2×2 region in the input matrix. With a stride of 1, the filter will move with an increment of 1 along the columns and rows until it reaches the end. Thus, a 3×3 matrix will be the output. If the stride was 2, the filter would slide along the columns by an increment of 2 and equivalently by an increment of 2 along the rows, which would create a 2×2 output.

After a few convolutions have been applied, it is common to do pooling, alas keep interesting values and discard the rest. This downsampling procedure can be done by, for example, sliding a 2×2 filter on to the previous 3×3 output. Although this filter will not contain any values, it will capture the wanted information from the segment it is slided across.

Max pooling with a 2×2 filter

$$\begin{bmatrix} 575 & 540 \\ 575 & 950 \end{bmatrix}$$

Equivalently with min pooling

$$\begin{bmatrix} 205 & 205 \\ 320 & 320 \end{bmatrix}$$

Additionally, using average pooling will compute the average value in the 2×2 blocks

Table 1: A 4×4 matrix, applied 2×2 max pooling with stride $= 2$.

2	3	1	9
4	7	3	5
8	2	2	2
1	3	4	5

2×2 *max pooling* $=$

7	9
8	5

$$\begin{bmatrix} 352 & 350 \\ 481 & 582 \end{bmatrix}$$

Table 1 visualizes the procedure of pooling a bit more clearly. Lastly, padding can be used if the dimensions do not match (or if you want the output dimension to match the input dimension), zeros will be padded along the rows and columns.

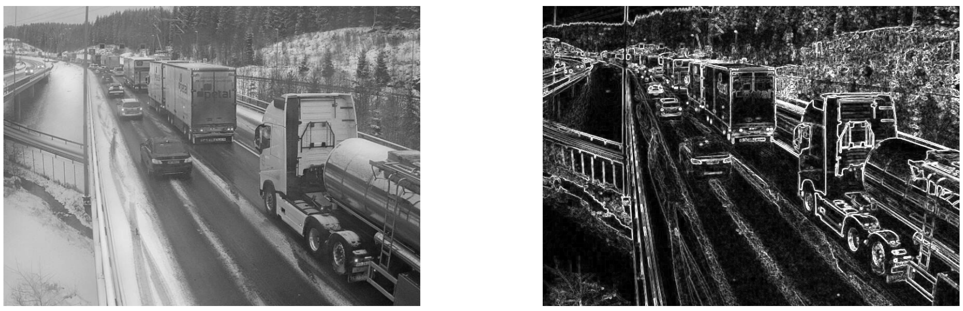

(a) E6, Nøstvet (b) Sobel result

Figure 7: Convolving an image with sobel filter in x and y direction, and computing the magnitude of the gradient.

In figure 7 the image has been convolved with the filter $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ in x direction, and $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$ in y direction. This will capture the gradients of the image, areas with a large difference in pixel values will have a high intensity value. Thus, the edges will be detected and the gradient magnitude can be computed as follows, $|\nabla I| = \sqrt{I_x^2(m,n) + I_y^2(m,n)}$. This is an example of extracting features from an image, in this case the edges are extracted. In a CNN there will be multiple filters applied in various regions of the image, these filters will capture higher and lower level features. Pooling is done

Table 2: Confusion Matrix Example.

		True Label	
		Positive	Negative
Predicted Label	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

to keep the interesting segments, and simultaneously downsample the result, this will yield a more compact feature vector after flattening.

2.3 Evaluating a Neural Network

Measuring the performance of the network may depend on whether or not the classification setting is binary or multi-class, in either case, the formulas are based on the confusion matrix. A confusion matrix contains correct classifications along the diagonal, and false positives/negatives in the other columns (see table 2 for a binary classification example).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{\sum_{i=0}^{N-1} A(i,i)}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} A(i,j)} \tag{10}$$

$$Precision = \frac{TP}{TP + FP} = \frac{A(0,0)}{\sum_{i=0}^{N-1} A(0,i)} \tag{11}$$

$$Recall = \frac{TP}{TP + FN} = \frac{A(0,0)}{\sum_{i=0}^{N-1} A(i,0)} \tag{12}$$

A is the confusion matrix (table 2), indexed from zero, and N is the number of classes. These metrics can be used to evaluate the performance of the model, but in some cases these metrics can be misleading, especially if the data is imbalanced. The model may do very well on the majority classes, but still fail on the minority classes. This situation can still report fairly high accuracy, depending on how imbalanced the data is. A metric which incorporates better weighting of minority classes is the metric known as the F1 score, which is a harmonic mean of precision and recall. This metric will take the false negatives and positives into greater account.

$$F1_{Score} = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{13}$$

This measure will give a more reasonable accuracy in the case of imbalanced data if the confusion matrix reports a high number of false positives and false negatives.

Additionally, there is the Receiver Operating Characteristic (ROC) curve. This curve is plotted using various thresholds of true positive rates versus false positive rates. The area under the curve (AUC) is calculated to give an indication of how well the model is predicting. A value of 50% indicates that the model is guessing, and a lower value indicates that the model can not generalize at all, while a high value indicates that the model is good at generalizing to unseen data.

For a multi-class setting, the ROC curve is not ideal, but the other metrics can be applied by performing a *one VS rest* classification technique. This reduces a multi-class setting down to a binary-class setting, thus the metrics can be computed as is. This is done for each class, and micro-averaging or macro-averaging can be performed to indicate how well the model is doing.

In the case of three classes:

$$Micro - Avg_{Precision} = \frac{TP1 + TP2 + TP3}{TP1 + TP2 + TP3 + FP1 + FP2 + FP3} \tag{14}$$

$$Micro - Avg_{Recall} = \frac{TP1 + TP2 + TP3}{TP1 + TP2 + TP3 + FN1 + FN2 + FN3} \tag{15}$$

$$Macro - Avg_{Precision} = \frac{P_1 + P_2 + P_3}{3} \tag{16}$$

$$Macro - Avg_{Recall} = \frac{R_1 + R_2 + R_3}{3} \tag{17}$$

Micro-averaging is preferred because it is more sensitive to false positives and negatives than macro-averaging.

Table 3: Various pre-trained models, trained on ImageNet [2].

Model	Top 10 Acc	Weights
EfficientNet-B7	97.1%	66 M
EfficientNet-B5	96.7%	30 M
NoisyStudent	96.3%	9.2 M
MobileNetV3	75.2%	5.4 M

2.4 Transfer Learning

Training a deep neural network from scratch can be a tedious task, especially if there is not sufficient data. Neural networks are generally data demanding, but with a concept known as transfer learning it is possible to use an already trained, or parts of an already trained model to your advantage. As the network is trained, the weights are continuously adjusted. These weights can be reused and frozen, which means that they will not update during training. In the case of a CNN, the convolutional layers are frozen, thus, it will act as a feature extractor for an arbitrary image of a given size. The resulting feature vector is then supplied to a fully connected layer, and the fully connected layer will not have its weights frozen, these will be trained.

Various pre-trained models available have been used as part of object recognition challenges, and these models are usually trained on extremely large datasets, with many classes. However, the data used for the pre-trained model should match your field of interest. Using a pre-trained model which has been trained on a large database of flowers will not be very useful when training a vehicle classifier, for example.

Nevertheless, leveraging transfer learning can provide various benefits, such as having a matured feature extractor, in the case of CNN, and lower training time. Choosing the right pre-trained model should be based on the data it was trained with, the amount of parameters (total amount of weights) in the network, image size constraints (due to memory usage), and prediction speed. If the model is exported and used for inference on some IoT device, it would be preferable to use a model which can predict with low latency.

Table 3 shows a few models that have been trained on ImageNet. ImageNet is a dataset containing $14,197,122$ annotated images, and is regularly used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [18]. Other datasets like Microsoft's COCO, Pascal Visual Object Classes, CiFar, among others are also suitable.

2.5 Regions with CNN Features (R-CNN)

R-CNN is a powerful object detection and semantic segmentation method developed by researchers at UC Berkeley [19]. This algorithm combines a CNN for feature extraction with region proposals in an image, the algorithm runs selective search which tries to separate and segment the image into various regions. Many regions are extracted and fed into a CNN, in this approach a CNN and region extractor is trained (anchor points).

This method can be superior to traditional techniques that only involve background subtraction and a vanilla CNN. This approach does not rely on a foreground mask, and does not suffer from potential noisy masks that lead to false detections. However, training such a network is considerably more demanding.

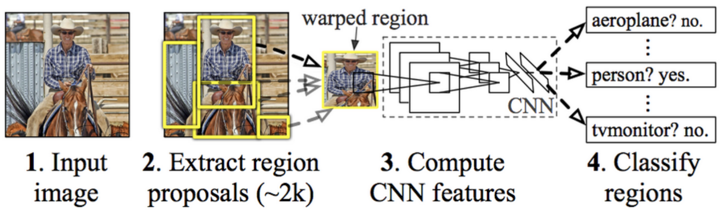


Figure 8: R-CNN Overview [19].

Overview: figure 8 depicts the overview of the method, the first step is to extract regions from an input image. R-CNN uses selective search to extract the regions, and will extract ≈ 2000 regions from an input image. A feature map will be calculated for each region and will be classified accordingly.

Selective Search combines an exhaustive search and segmentation algorithm. *Efficient Graph-Based Image Segmentation* is used to create initial regions, the regions are then grouped together iteratively using a greedy algorithm. A similarity measure between all the regions are computed and the two most similar regions are grouped together. New similarity measures are calculated between the resulting region and its neighbors, this being repeated until the whole image becomes a single region.

A hierarchical algorithm (bottom-up grouping) is used to allow all object scales to be taken into account. Segmentation may be done based on separating objects based on texture (pattern), color, or lighting (shading and lighting color). Diversification of the sampling is done by using a variety of color spaces with different invariance properties, by using different similarity measures s_{ij}, and by varying the starting regions. Four complementary similarity measures are used, and they all range from $[0, 1]$.

Color similarities are measured by $s_{color}(r_i, r_j)$, for each region a one-dimensional color histogram is obtained for all color channels using 25 bins respectively. These histograms are normalized using their L_1 norm.

$$s_{color}(r_i, r_j) = \sum_{k=1}^{n} min(c_i^k, c_j^k) \tag{18}$$

$$C_t = \frac{size(r_i)xC_i + size(r_j)xC_j}{size(r_i) + size(r_j)} \tag{19}$$

Similarity is measured using the histogram intersection.

Texture similarities are measured by $s_{texture}(r_i, r_j)$. Scale Invariant Feature Transform (SIFT) measurements are used to represent textures. Gaussian derivatives are taken in eight orientations using $\sigma = 1$ for each color channel. For each orientation for each color channel, a histogram of bin size 10 is extracted. This leads to a histogram T_i for each region r_i with dimensionality $n = 8 * 10 * 3 = 240$ when three color channels are used. L_1 norm is used here as well.

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} min(t_i^k, t_j^k) \tag{20}$$

$s_{size}(r_i, r_j)$ encourages small regions to merge early, thus keeping unmerged regions rather similar in size throughout the algorithm.

$s_{fill}(r_i, r_j)$ measures how well region r_i and r_j fit into each other, which is used to fill gaps.

Finally, using the various similarity measures, regions are grouped together, forming the respective bounding boxes. Mean Average Best Overlap (MABO) is used for evaluating the results.

Extracting Features: for each region, a 4096-dimensional feature vector is extracted using UC Berkeley's in-house *Caffe* [20] CNN deep learning framework. Additional image preprocessing steps are required prior to the feature extraction, for the architecture used the region must be resized to 227×227 pixels.

Training: the CNN used for this algorithm has been explicitly pre-trained on a larger dataset (ImageNet [18]), but for classification purposes only. Additional tuning is required to adapt the CNN to do detection and deal with warped proposal windows, Stochastic Gradient Descent (SGD) is continued to further update the weights of the CNN, but only using the warped region proposals.

Testing involves extracting the ≈ 2000 regions for each test image, warp each region proposal and propagate it through the CNN for feature extraction. For each extracted feature vector (per class), score it with an SVM classifier trained for the class in question. Lastly, for each scored region, apply a greedy non-maximum suppression which rejects a region based on Intersection over Union (IoU) score larger than a learned threshold.

Fast R-CNN: due to the impractical performance of R-CNN, a new faster variant has been developed by the same researchers. Fast R-CNN [21] emphasizes on reducing proposal generation bottlenecks by introducing sparse object proposals.

Faster R-CNN: further changes to Fast R-CNN were made to make it even more robust. Faster R-CNN [22] improves detection speed by sharing convolutional features with the down-stream detection network.

2.6 Single Shot Detector

A Single Shot Detector, or SSD for short, is an object detection technique that tries to detect objects in images by utilizing a single neural network [23]. A fixed size collection of bounding boxes are produced through a feed-forward convolutional network, object class instances in the boxes are scored, and then non maximum suppression is applied to produce the final detection. These detections include key features such as:

- **Multi-Scale Feature Maps**
  Convolutional feature layers are added to the end of the truncated base network, allowing for prediction of detections at variable scales.
- **Convolutional predictors for detection**

Each feature layer can produce a fixed set of detection predictions using a set of convolutional filters.

- **Default boxes and aspect ratios**
  Associate a set of default bounding boxes with each feature map cell.

2.7 Spatial Transformer Network

Rather than using simplistic data augmentation that only rotates, shifts, flips, etc. A Spatial Transformer Network (STN) [24] can be deployed, this network will work towards making the data fully invariant of transformations, scaling, and rotation. Before a sample is fed into the fully connected layer, it will be fed through the STN. This will alter the feature map extracted from the convolutional steps, thus more diverse data can be generated and fed to the fully connected layer. A STN has not been used for this thesis, but it is worth mentioning as a potential further work technique.

2.8 GPU Acceleration for training

A deep neural network can become quite difficult for a Central Processing Unit (CPU) to handle on its own, due to low cache memory, speed and low parallelism, it is often preferred to use a GPU for training the network. The forward and backwards propagation involves a lot of elementary linear algebra operations, such as matrix multiplications. For a large network it will be expensive and slow to run these types of operations on a CPU. A GPU is designed to handle such operations explicitly, hence why games rely more on a GPU than a CPU due to the nature of its computations to render and handle a 3D world.

At UiS the Tesla P100 and V100 GPUs have been used for GPU acceleration, however, consumer grade GPUs are also very effective. Some brief steps regarding how to setup GPU acceleration with Tensorflow will be presented in the implementation section.

2.9 Background Subtraction

Background Subtraction, also known as foreground detection, is an algorithm which tries to separate the moving parts in an image from its background.

Frame differencing is the simplest method, compare consecutive frames with a background frame and only keep the pixels which changed by a certain amount defined by a threshold λ. If no explicit background frame is available, pick the first frame as the background and compare consecutive frames with it.

If a background is present, $I'(x, y) = |I(x, y) - B(x, y)| > \lambda$. $I'(x, y) = |I(x, y)(t + 1) - I(x, y)(1)| > \lambda$ otherwise. This will extract the foreground mask.

Averaging N consecutive frames is also an option, as N grows, the foreground will disappear, leaving a background mask which can be used to extract the foreground from future frames. At frame t the background is computed, $B(x, y, t) = \frac{1}{N}\sum_{i=1}^{N} I(x, y, t - i)$. Frame differencing is used to extract the foreground mask.

Background Mixture Models is a technique that tries to model the background as a series of probability density functions. The gaussian distribution is commonly used to model the background, and pixels that do not follow the various distributions are assumed to be part of the foreground.

2.10 Morphological Operations

Mathematical Morphology was developed by Georges Matheron and Jean Serra. Initially, this method was used for quantification of mineral characteristics from thin cross sections. This was part of Serra's PhD thesis, the work also contributed to theoretical advancements in integral geometry and topology. Mathematical Morphology is useful for pre- and post- processing of binary images, as it provides as set of tools for expanding and shrinking regions in an image [25].

Morphological operations are non-linear operations that can be applied to binary images or grayscale images. These operations are mainly used to alter the shape, form or structure of the concentrated pixels in an image. For example when applying background subtraction, a binary foreground mask is extracted and this mask may have some noise which should be removed. Morphological operations in this context are mainly used for removing noise, and potentially separating occluded segments. A structuring element (matrix) is used, similarly to convolution, and is applied over all regions in the binary image.

Dilation, denoted by $\oplus$ is the dilation of an object A with the structuring element B. Defined as $A \oplus B = \{x : B_x \cap A \neq \emptyset\}$. This implies that a pixel under the anchor point of B is set to 1 if at least one pixel in B is inside A.

Erosion, denoted by $\ominus$ is the erosion of an object A with the structuring element B. Defined as $A \ominus B = \{x : B_x \subseteq A\}$. This will set any pixel in the anchor point B to 1 if B intersects entirely with A.

Dilation and Erosion can be thought of as expansion and shrinking, mixing these two operators can fill, expand (separate) holes, etc.

Opening, denoted by $\circ$ is the erosion, and subsequent dilation of the object A with the structuring element B. Defined as $A \circ B = (A \ominus B) \oplus B$, this will shrink and expand the result, further separating it.

Closing, denoted by $\bullet$, is the dilation and subsequent erosion of the object A with the structuring element B. Defined as $A \bullet B = (A \oplus B) \ominus B$, this will expand and shrink fill the expanded segments (fill holes).

3 Implementation

This chapter will emphasize on the methodology used throughout the thesis work.

3.1 Data Collection

Data has been collected from different sources, some data has been collected for consecutive days using publicly available on-demand data. Specifically, the webcams administered by the road authority have been used, since those cover Norwegian terrain and traffic. Certain busses, trucks and vans can differ from country to country, in both shape and/or color. In Norway most of the public busses are Volvo's, but they may differ in design from region to region. Capturing these differences is important to ensure correct classifications if the system is deployed in a different region. Additional data has been collected from online datasets to add diversity.

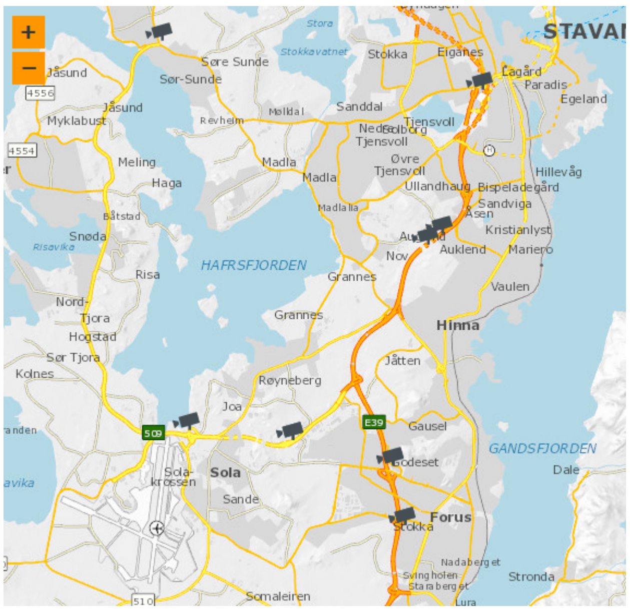


Figure 9: Norway's Road Administrative Authority's publicly available webcams in the Stavanger region.

Other data collection sources include the exclusive dark dataset [26] and KITTI Vision Benchmark [27]. The exclusive dark dataset has been created for the sole purpose of training classifiers to identify objects in varying illumination. The project also includes some matlab code which can be used on images to artificially change the illumination (data augmentation). In addition, the dataset contains around 7,363 annotated images, spanning across 12 classes (bicycle, bottle, boat, bus, car, cat, chair, cup, dog, motorbike, people, table), although for this project only four of these classes are relevant.

Lastly, the KITTI dataset is a large dataset used primarily for autonomous driving tasks/challenges, and contains roughly 15,000 annotated images spanning across 8 classes (car, van, truck, pedestrian, person-sitting, cyclist, tram, misc). Only four classes were relevant here as well, and due to the images in this dataset being considerably larger, only a subset of this dataset was extracted. A fair sample of each relevant class was extracted by parsing all the images and their respective annotation data, images accumulating up a certain amount of objects (annotations) for each class would be kept.

For the data collected from publicly available webcams, annotation had to be done manually. LabelImg [28] was used for this purpose, bounding boxes would be placed over objects of interest and labeled accordingly. Each image file would have a respective Extensible Markup Language (XML) file which contained various properties for each annotated object, properties such as bounding box coordinates, truncation, label, etc. Bounding boxes would be marked as truncated if the bounding box intersected with the boundaries of the source image. A bounding box could also be marked as difficult, which can be helpful to identify highly occluded segments. As for the other data sources, these had different annotation formats. A conversion was needed for these, everything was converted to match LabelImg's annotation format.

Finally, the annotated data would be compiled into a train and test file of filetype *TF record* (Tensorflow's own binary format). Data augmentations would be done in memory, as specified in the pipeline configuration file. This file format is not necessary for training and testing a classifier, but for the purpose of training and testing a streamlined object detection system it was beneficial. Due to the fact that there are more files to keep track of, it will lead to unnecessary disk I/O.

A 90-10 train-test split was used for both methods, for the region proposal based system there would be

- 5089 images for training

  - RAW: 694
  - KITTI: 2348
  - ExDark: 2047

- 570 images for testing
  - RAW: 78
  - KITTI: 263
  - ExDark: 229

Similarly for the regular CNN based approach there would be

- 18288 images for training
  - RAW: 4719
  - KITTI: 6240
  - ExDark: 7329
- 2014 images for testing
  - RAW: 496
  - KITTI: 728
  - ExDark: 790

Both methods are technically trained on the same data, the classifier is trained on each actual object in every image in the data used to train the region proposal system. In the region proposal system, larger images are used, and these images contain multiple objects that have been annotated accordingly.

Listing 1: TF Record Format for Object Detection - One annotated object.

```
'image/height': dataset_util.int64_feature(height)
'image/width': dataset_util.int64_feature(width)
'image/filename': dataset_util.bytes_feature(filename)
'image/source_id': dataset_util.bytes_feature(filename)
'image/key/sha256': dataset_util.bytes_feature(key)
'image/encoded': dataset_util.bytes_feature(encoded_jpg)
'image/format': dataset_util.bytes_feature(image_format)
'image/object/bbox/xmin': dataset_util.float_list_feature(xmins)
'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs)
'image/object/bbox/ymin': dataset_util.float_list_feature(ymins)
```

```
'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs)
'image/object/class/text': dataset_util.bytes_list_feature(classes_name)
'image/object/class/label': dataset_util.int64_list_feature(classes_id)
'image/object/difficult': dataset_util.int64_list_feature(difficult)
'image/object/truncated': dataset_util.int64_list_feature(truncated)
'image/object/view': dataset_util.bytes_list_feature(poses)
```

Listing 1 is an example of one annotated object in an image, the fields are extracted by Tensorflow's object detection API during training to fetch the annotated objects.

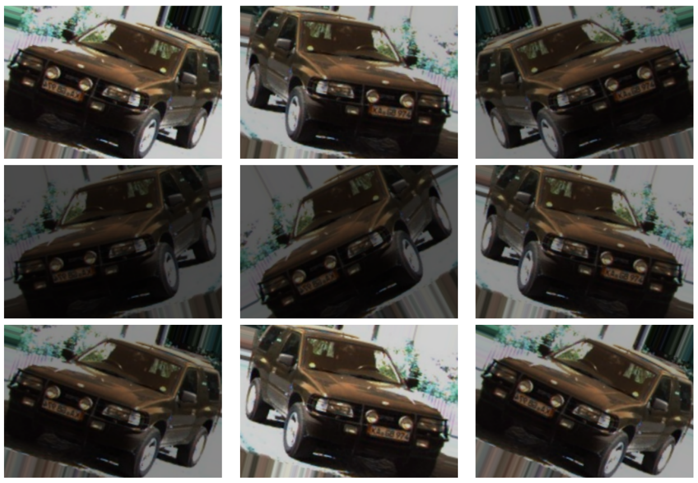


Figure 10: Augmentations applied to a random image.

Figure 10 showcases various image augmentations applied to an image of a car. In tensorflow, ImageDataGenerator is used to generate these augmentations, as it supports a vast set of options, such as width-shift, height-shift, rotation, vertical and horizontal flip, shear, zoom, brightness adjustments, etc. One thing to keep in mind is that this generator will replace the original data. Intuitively, it should add to the original data, but it does not. This can, however, be altered by using a custom data generator.

Table 4: Vehicle Distribution.

Source	**Bike**	**Bus**	**Car**	**Person**	**Truck**
Webcam	1	77	4815	0	322
ExDark	1000	685	2360	4074	0
KITTI	0	0	5623	1956	412
SUM	1001	762	12798	6030	734

3.2 Imbalanced Dataset

Table 4 showcases the vehicle distribution of the data, the data collected is heavily biased towards general vehicles. And this bias is easily identified in the confusion matrices (as shown in the results chapter), thus it is considerably easy for the model to correctly identify cars. But other vehicle types may be generalized to cars as well, because there is simply too few of these other types in comparison. To overcome this, more data can be added for the classes that lack enough samples, re-sampling techniques which emphasize on augmentation, augmenting only the scarce classes, etc.

A technique which has been used to combat the imbalanced data is a custom data generator. The default data generator shuffles the data and extracts the first X images for a batch. Due to the imbalance, there is a high probability that most of the images in each batch will be cars or pedestrians, which could lead to a network that will have high loss whenever different classes are introduced in a batch. A custom data generator can be tailored to take this issue into account. Rather than using one data generator which samples from all the classes, one generator is used for each class to sample from a shuffled set of the respective images belonging to that class. The sampling is probabilistic, thus we can control to a certain extent the probability of a minority class being present in a batch. A *Bernoulli* distribution is used to determine whether or not to extract an image from data generator X with probability p. For a batch, each class is considered with a probability p, this probability is higher for the minority classes, and lower for the majority classes. The algorithm sorts the probabilities from low to high, and evaluates each class with its probability. Defined as $P(x = 1) = p$ and $P(x = 0) = 1-p$. This way the network is guaranteed to be trained on a sufficient amount of samples from the minority classes. If no class has been decided for an image in the batch, it will revert to a random selection. Results related to the effect of this generator can be observed in the results section.

3.3 Tensorflow Library

Tensorflow is a popular machine learning library, it is originally written in C++ but offers Python bindings that allow the user to make use of the library through Python. This makes the library very fast, because it directly interfaces with the lower level bindings. The framework also encourages GPU acceleration, and is fairly easy to configure if you have compatible hardware.

A Compute Unified Device Architecture (CUDA) compatible NVIDIA GPU is required, and a CPU with Advanced Vector Extensions (AVX) instruction support is recommended. CPUs launched around 2011 and later should have AVX support. AVX support is not strictly required to run GPU acceleration, the default binaries shipped with Tensorflow (via pip) have been compiled with AVX. If your hardware does not have AVX support, you either recompile the source code or you can download pre-compiled binaries from the *tensorflow-windows-wheel* GitHub repository (for Windows OS) [29].

3.4 Tensorflow Object Detection API

An explicit Object Detection Application Programming Interface (API) [30] has been widely used for the R-CNN approach in this thesis, as it offers a wide selection of pre-trained models. Some of the pre-trained model choices are ResNet, EfficientDet, SSD MobileNet, InceptionNet, among others [31]. The various choices have their pros and cons, such as increased performance vs decreased accuracy, and vice versa. Since this would be deployed on rather limiting hardware where latency is important, better performance is preferred.

Listing 2: Installing Tensorflow Object Detection API

```
git clone https://github.com/tensorflow/models.git
cd models/research
protoc object_detection/protos/*.proto --python_out=. # Compile protos.
cp object_detection/packages/tf2/setup.py . # Install TF Obj Det API.
python -m pip install --use-feature=2020-resolver .
```

Listing 2 showcases how to download and install the API, the steps involved in using the API are:

- Configure a pipeline configuration file, define how deep the network will be, preprocessing steps including data augmentation, optimizer, define pre-trained checkpoint path, train and test data source, etc.
- Convert train and test data into single binary files (TF record) for streamlined use (assuming data is already annotated and ready).
- Train and evaluate the model using specified evaluation metrics in the pipeline configuration.

- Export the model for inference usage.

These steps are generally followed if a new model structure from scratch is not necessary. If a new model needs to be defined, more configurations have to be made rather than relying on the pre-trained model's configuration.

Configuring: when a pre-trained model has been downloaded, the pipeline configuration file used for the model in question is copied and edited to fit the needs of the data you have available. The amount of classes used, path to train set, test set and pre-trained checkpoint should be updated prior to training and, additionally, some evaluation metric should be set for the test set. Tensorflow tensorboard files are generated throughout training and evaluation, these files can be used to get an in-depth overview of the train and evaluation process. It is possible to detect overfitting, figure out where to do early stopping, see if the gradient descent is getting stuck, etc (the files generated during training are very large though). Metrics available include the default COCO detection evaluation metrics, and COCO mask evaluation metrics. Mean Average Precision (MAP) is used as the evaluation measure for these models.

A wide range of data augmentation options are also available. These options help improve the model by adding additional data, steps like rotating segments, flipping, brightness adjustments, crop random regions, etc. will give the model extra insight. One challenge in this thesis is varying illumination, especially poor illumination. To account for this, the model is trained with augmentation configurations that focus on altering the brightness, hue, contrast, color and saturation of the training images (to "sharpen" the model's "vision"). But, naturally, this additional data is also helpful in other means, such as improving the overall accuracy due to more diverse data.

Train & Test data: as mentioned in section 3.1, data is merged into a binary format so that it can be fed directly to the API.

Training & Evaluation: a file *research/object_detection/packages/tf2/setup.py* was copied from the API into the workspace to initiate the training. The path to the pipeline configuration and model directory has to be supplied to this script. The same script is used for evaluation by supplying *–checkpoint_dir* and the path to the generated checkpoints from the training procedure. This eval procedure will generate an eval folder with event files (within workspace/models/my_model_here), and these files can then be analyzed in Tensorboard itself. Additionally, the train procedure can generate event files if *–record_summaries* is set to true.

Run with parameters, where *–checkpoint_dir* is only needed for evaluation.

- –model_dir
- –pipeline_config_path
- –checkpoint_dir

Listing 3: Exporting the necessary environment variables for GPU acceleration support.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib/cudnn/lib64
export CUDA_VISIBLE_DEVICES=<ID>
```

To use GPU acceleration on a Unix based system, the *LD_LIBRARY_PATH* environment variable has to be extended so it can reach the CUDA binaries and the cuDNN bindings. See listing 3 for an example. The *CUDA_VISIBLE_DEVICES* environment variable is used to let Tensorflow know which GPU to use, the ID of the GPU, ranging from 0 and up is used. Multiple IDs can be inserted as well (cuDNN is the Python binding which allows interaction with CUDA when using Python).

Listing 4: Enable Dynamic Memory Allocation for GPU.

```
cfg = tf.compat.v1.ConfigProto()
cfg.gpu_options.allow_growth = True
sess = tf.compat.v1.Session(config=cfg)
```

Lastly, if GPU acceleration is used, configuring the train script to not consume the entire capacity of the GPU will allow for more flexibility, such as training multiple models on the same GPU, if necessary (see listing 4).

Exporting the model: this is done by copying *research/object_detection/packages/exporter_main_v2.py* into the workspace folder and running it with the parameters

- –input_type image_tensor
- –pipeline_config_path
- –trained_checkpoint_dir
- –output_directory

This will generate a subfolder with assets and variables in the exported-models folder in the workspace, and this model can then be loaded for inference.

Inference uses standard Tensorflow object detection API functions, however there is also a threshold value which can be adjusted. This threshold value determines the minimum required probability for a detection to be valid.

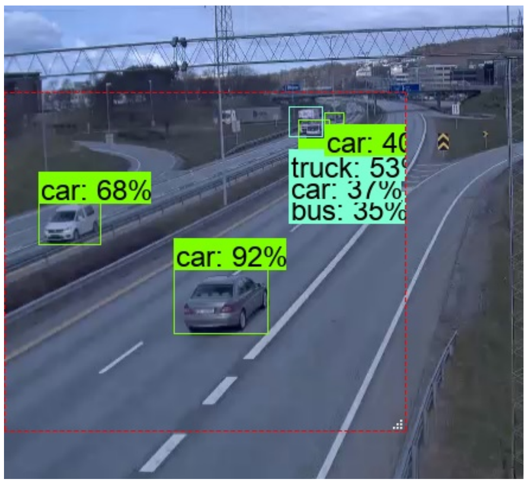


Figure 11: Region Proposal - Detections within the red stapled box, threshold of 30%.

Figure 11 shows a real time object detection system, detections are registered inside the Region of Interest (RoI). The real time object detection Graphical User Interface (GUI) application was written in Python and it uses PyQt5 for the GUI framework [32].

3.5 CNNs with Tensorflow

In addition to the R-CNN approach with the object detection API from Tensorflow, a technique using a vanilla CNN and background subtraction has been developed. A CNN is trained on the annotated data, each image may have multiple annotations since the data is collected and optimized for region proposal, meaning that these annotated segments have to be extracted and exported as new individual images.

Tensorflow recommends a file structure like this for CNN models:

- *workspace/images*
  - train

    - car
    - truck
    - ...
  - test
    - car
    - truck
    - ...

The images for the respective classes are then loaded through Tensorflow's ImageDataGenerator class by calling *flow_from_directory*. ImageDataGenerator also supports data augmentation parameters, such as random horizontal flip, vertical flip, shear, zoom, cropping, brightness adjustments, shifting, rescaling, etc, this way the framework augments your data further, leading to further train and test samples. This is good for overcoming overfitting, especially if there is a lack of data overall, and it may naturally help your model generalize better.

Hyperparameter tuning is initiated to determine the most suitable parameters for the train procedure, the model with the highest accuracy on the test set is selected, and its parameters are noted. There are endless possibilities in this tuning procedure, which makes it very important to choose reasonable parameters. Simply because it will take too long time to check too many combinations, working with image data is considerably more costly. Parameters that have been prioritized in this thesis are:

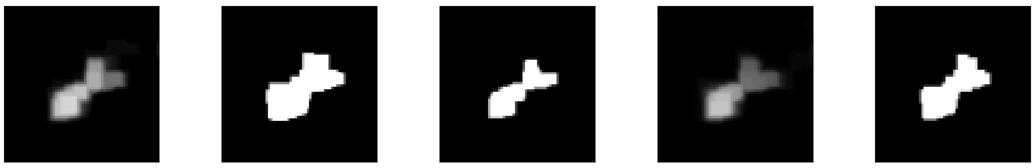
- Optimizers
- Drop Out
- Learning Rates
- Label Smoothing
- Regularization

Optimizers include: RMSProp, Adam and SGD. Drop out is useful to add because it helps the network to not rely entirely on single nodes. A drop out of 1% implies that the network has 1% chance to reset the weights on a random node. As for label smoothing, a value of 0.1 will change how the model determines if a class belongs to one of the five classes used. For example, if you have an image with the label car, it would normally imply that this is 100% a car, but with a label smoothing of 0.1 we say that every class has a $\frac{0.1}{5} = 2\%$ chance. This will reduce our model's confidence in a way which could boost its way to generalize for unseen data, and the association would now be $1 - 0.02 * 4 = 0.92 = 92\%$ car and 2% for any other class.

For the CNN itself, transfer learning is used, specifically the MobileNet v3 model [33]. This model has been trained on the ImageNet dataset, as mentioned in the theory section. Training has been done with both freezing the weights and unfreezing them (on the convolutional layers). Dropout has been added to the last convolutional layer (input layer in this case), furthermore, the fully connected layer consists of a dense layer with 1024 nodes, dropout, and an output layer with 5 nodes (5 classes). Softmax is used for the output layer, and relu is used for the hidden layers. *Categorical Crossentropy* is used for the loss function, as this is a multi-class setting. L1 regularization is used on the output layer to penalize large weights (provide slightly simpler fit), a $\lambda = 0.001$ was found to be satisfactory. A fully connected layer with two hidden layers has also been trained and tested, but this did not provide any significant change in accuracy. More details about the results of various parameters will be presented in the next chapter.

3.6 Background Subtraction

Background subtraction is an alternative method to the R-CNN detection approach, and is used together with a vanilla CNN. The background subtraction approach is used to extract objects in motion from a series of images extracted from a stationary camera. This procedure is prone to generate various random noise and requires denoising before separating and extracting the objects of interest. When a frame has been denoised, a set of morphological operations (non-linear filters for binary images) will be applied to further separate objects from one another. Objects are extracted using OpenCV's contour line function, which separates objects that are surrounded by different pixels (pixel values of 0 and 1 in this case). Finally, the extracted objects are classified using a CNN.



(a) Mean Blur (b) Box Mean Blur No norm. (c) Median Blur (d) Gaussian Blur (e) Bilateral Filter

Figure 12: Various filters applied before using any morphological operations.

Figure 12 visualizes how the different filters applied to the very first frame have turned out. The median filter seems to be the strongest filter overall, followed by the bilateral filter.

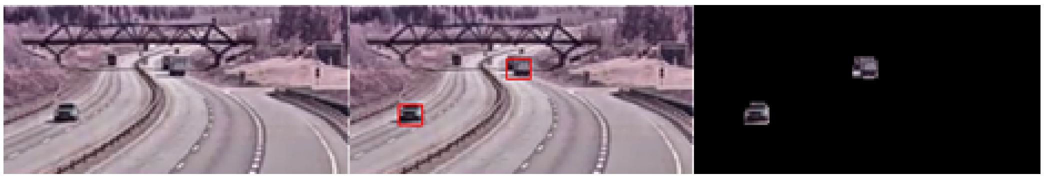


Figure 13: Capturing an image, detecting the objects in motion, and extracting them.

In figure 13, a frame from a webcam has been captured and the foreground objects have been extracted, and are now separable. The separated objects can be fed to a classifier, in this case, a CNN.

OpenCV's K Nearest Neighbour (KNN) background subtractor is used and it holds a history of 50 images at a time, which it uses to adapt to the foreground changes. When a foreground object is extracted, various smoothing thresholds are applied from low to high. The increased smoothing removes more noise and the detection function tries to minimize this noise by finding the most suitable smoothing value. A median filter is used immediately on the extracted foreground image, and is followed by OpenCV's find contours function. This will extract the objects in the binary mask then, finally, the position and size of these objects will be used to extract the RGB counterpart from the source image. At this point, the objects are ready for classification.

A threshold is given to determine the minimum amount of % reasonable objects required for a smoothing value to be deemed useful, all smoothing values are evaluated, and the highest score (% of reasonable objects) is selected. Reasonable objects are objects that are at least greater than a given minimum size, so, if there is a lot of tiny noise segments, these will be excluded, thus if there is too much noise the reasonable objects fraction will be small, thereby trying a higher smoothing value for "better luck": $R_{Frac} = \frac{R_{Count}}{Num\ Objects}$ where R_{Frac} is a value between 0 to 1 and R_{Count} is the amount of reasonable objects. Num objects is the total number of objects extracted from the find contours function. Smoothing values $\{1, 3, 5, 7, 9\}$ were used for this algorithm, more values can be used (odd values), higher values blur more, thus potentially removing important details in the image.

After applying smoothing, various morphological operations conforming to a research paper from Javadzadeh, Banihashemi and Hamidzadeh is applied [34].

- Dilation with a 5×5 structuring element.
- Closing with a 3×3 structuring element.
- Erosion with a 6×6 structuring element.
- Opening with a 7×7 structuring element.

The size of the structuring elements can determine if smaller patches get filled, larger regions get merged, etc.

4 Result

This chapter will emphasize on presenting the findings of various experiments conducted throughout the thesis work.

4.1 Data Collection

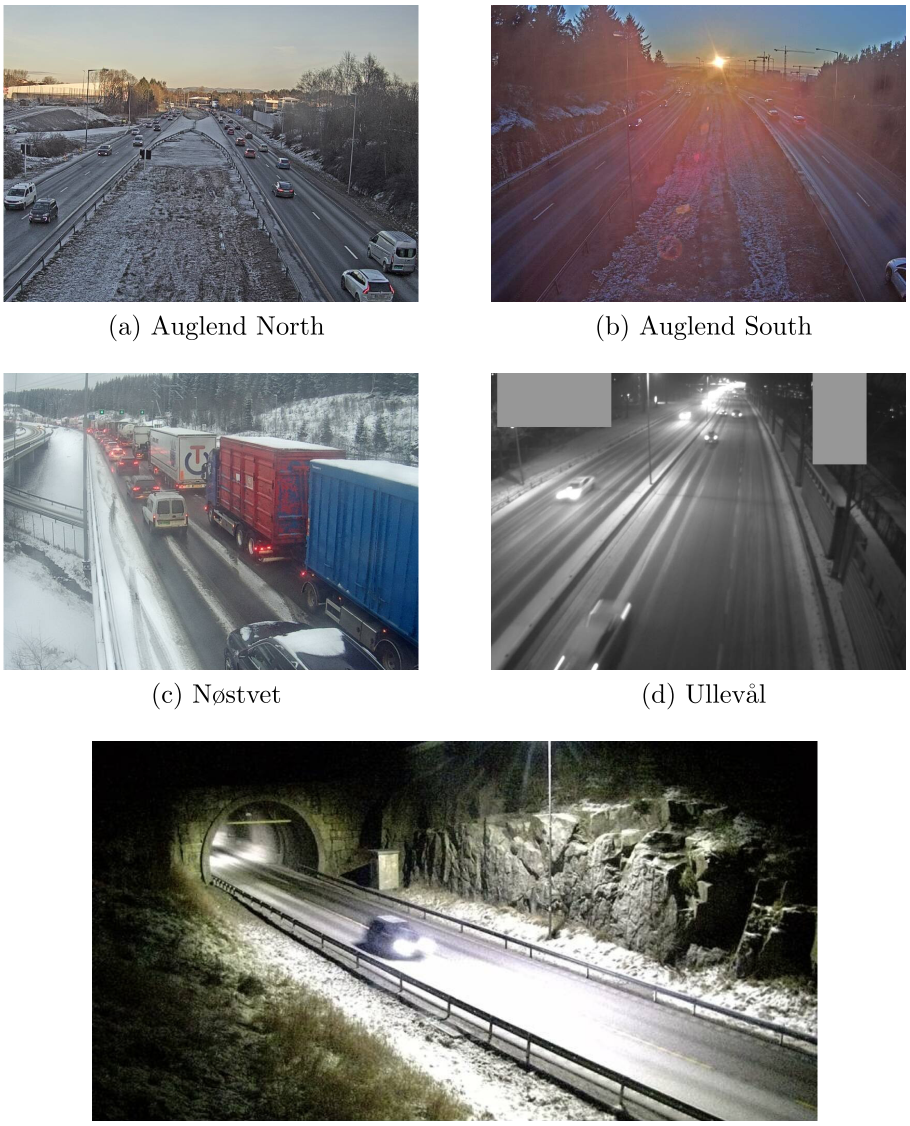
As mentioned in the implementation section, some data was collected both manually and online. The images collected via the online webcams hosted by the road administrative authority have rather varying quality. Qualities such as resolution, frames per second (FPS), positioning, and surrounding illumination from headlights varies a lot from region to region. Most of the images collected were 800×600 pixels (RGB).

Figure 14 displays some samples from the various locations used for collecting data from publicly available webcams. Busier roads have a better coverage, such as positioning, illumination along the road, etc. New image frames could be acquired from their respective sources every 4 to 8 minutes. There are also live webcams (streams) available, but these are only stationed in busier roads near the major cities. The frames per second on these live streams were between 1 to 2, with various hiccups every 10 seconds, short video streams were retrieved from their server and would only consist of a few frames recorded within a short interval.

These video streams were used to test the real time region proposal system, but those were more challenging to rely on for the background subtraction based method, due to the low frequency of images it became infeasible to properly separate the foreground from the background (too much difference between successive frames produces too much noise).

4.2 Region Proposal & Object Detection

As seen in figure 11, detections and classifications are made in real time solely based on the input frame. Evaluation is done through tensorboard event files which are generated during train and evaluation, as stated earlier. These files need to have a distinct prefix *events.out.tfevents.** to be recognized by tensorboard.

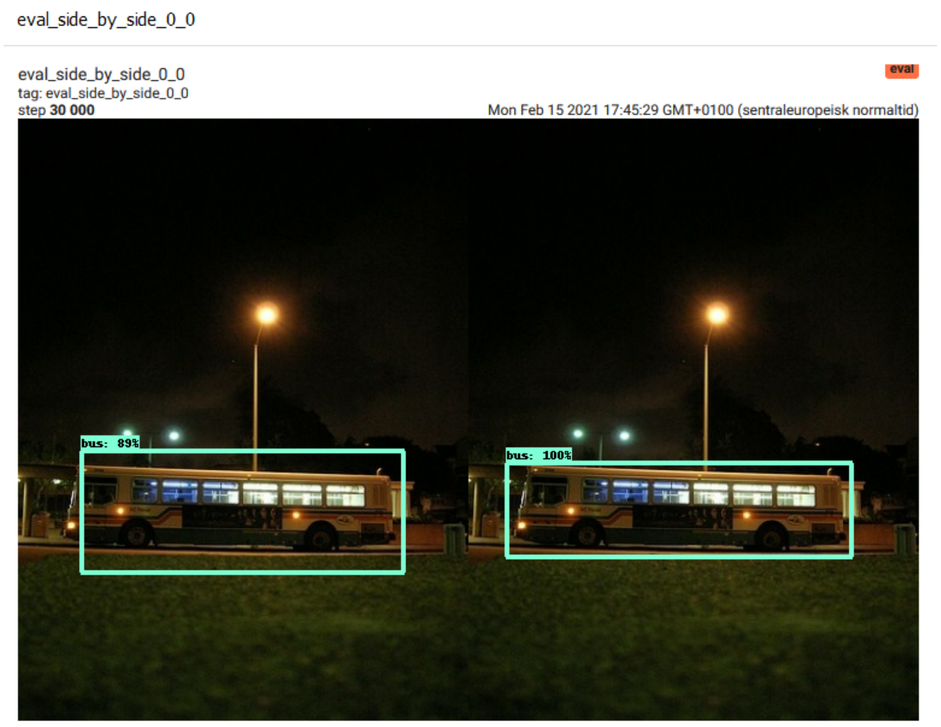


(a) Auglend North (b) Auglend South

(c) Nøstvet (d) Ullevål

(e) Vatland

Figure 14: Some samples collected from the public webcams.

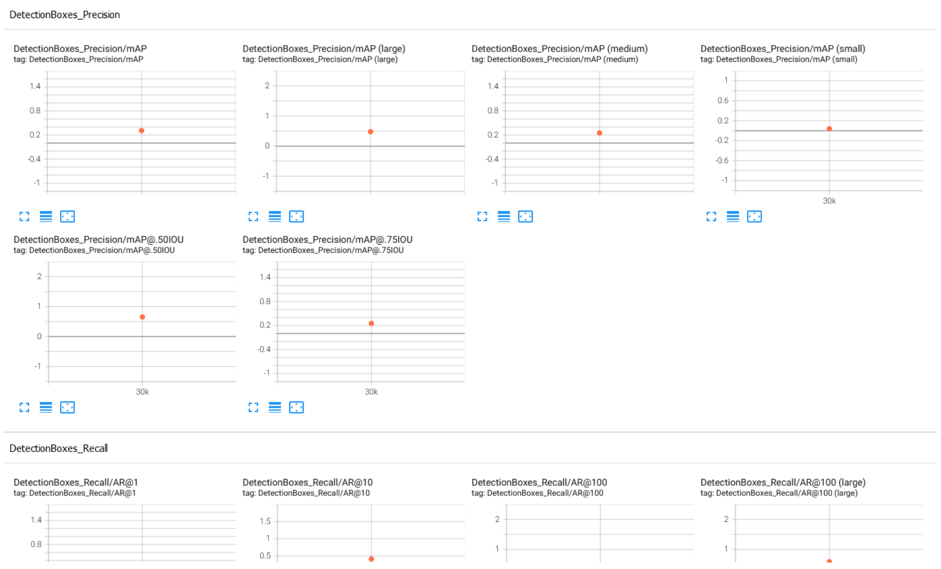


eval_side_by_side_0_0

eval_side_by_side_0_0
tag: eval_side_by_side_0_0
step 30 000
Mon Feb 15 2021 17:45:29 GMT+0100 (sentraleuropeisk normaltid)

Figure 15: Evaluating the EfficientNet detection model in tensorboard.

In figure 15 we can decipher how well the predictions are, the left image will show the predicted detection anchor points and the classification, and the right image is the ground truth.



DetectionBoxes_Precision
DetectionBoxes_Precision/mAP
tag: DetectionBoxes_Precision/mAP
DetectionBoxes_Precision/mAP (large)
tag: DetectionBoxes_Precision/mAP (large)
DetectionBoxes_Precision/mAP (medium)
tag: DetectionBoxes_Precision/mAP (medium)
DetectionBoxes_Precision/mAP (small)
tag: DetectionBoxes_Precision/mAP (small)
DetectionBoxes_Precision/mAP@.50IOU
tag: DetectionBoxes_Precision/mAP@.50IOU
DetectionBoxes_Precision/mAP@.75IOU
tag: DetectionBoxes_Precision/mAP@.75IOU
30k
DetectionBoxes_Recall
DetectionBoxes_Recall/AR@1
tag: DetectionBoxes_Recall/AR@1
DetectionBoxes_Recall/AR@10
tag: DetectionBoxes_Recall/AR@10
DetectionBoxes_Recall/AR@100
tag: DetectionBoxes_Recall/AR@100
DetectionBoxes_Recall/AR@100 (large)
tag: DetectionBoxes_Recall/AR@100 (large)

Figure 16: Evaluating the EfficientNet detection model in tensorboard.

Table 5: Top 5 Accuracy, round 1 (3 epochs).

Loss	**Acc**	**Batch Size**	**Optimizer**	**Learn Rate**	**Drop Out**	**Lbl Smooth**	**Reg Type**	**Reg Value**
1.03656	0.68918	16	SGD	0.001	0	0	L1	0.001
1.09448	0.66683	16	SGD	0.001	0.2	0	L1	0.001
0.86586	0.65392	16	SGD	0.001	0.2	0	L2	0.0001
1.06023	0.64945	16	SGD	0.001	0.1	0	L1	0.001
0.86956	0.64499	16	SGD	0.001	0	0	L2	0.001

Table 6: Hyperparameter tuning parameters overview (1296 combinations).

Batch Size	**Optimizer**	**Learn Rate**	**Drop Out**	**Lbl Smooth**	**Reg Type**	**Reg Value**	**Tuning**
8	RMSprop	0.001	0	0	L1	0.0001	No
16	Adam	0.01	0.1	0.1	L2	0.001	Yes
	SGD	0.1	0.2	0.2			

Furthermore, in figure 16, the performance metrics can be analyzed. These metrics reflect how well the model is able to predict correct anchor points for bounding box proposals, and whether or not the model is able to correctly predict the class label.

4.3 Convolutional Neural Networks

In most of the experiments the MobileNet v3 model has been used. This model requires images of size 224×224, and normalized pixel values between $0 - 1$. This model is optimized for IoT devices, and has a significantly lower amount of parameters that have to be trained (roughly 5.4 M). Training is fast (≈ 5 min per epoch) and convergence can be reached within roughly 15 epochs.

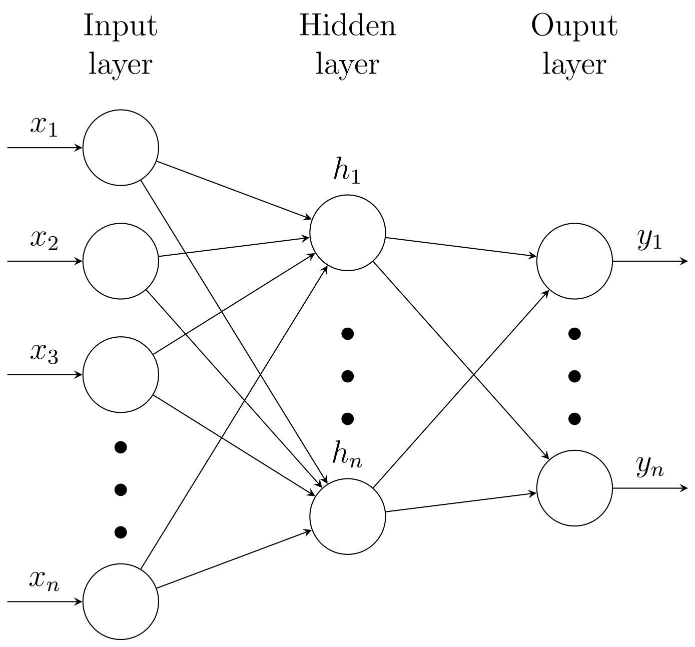


Figure 17: Fully Connected Layer.

Figure 17 showcases the fully connected layer used.

- $x_n = 1280$
- $h_n = 1024$
- $y_n = 5$

x is the resulting flattened feature vector from the convolutional filters and pooling. The input for the convolutional network is $224 \times 224 \times 3 = 50176 \times 3$, and from this a size 1280×1 feature vector is extracted.

Naive approach: a naive model was initiated at first for testing purposes, with parameters chosen based on intuition.

- 16 Batch Size.
- 0.0001 learning rate, RMSProp.
- 0.2 dropout on the input layer.
- 0.1 label smoothing.

- 0.0001 L2 regularization on the output layer.
- No hidden layer.

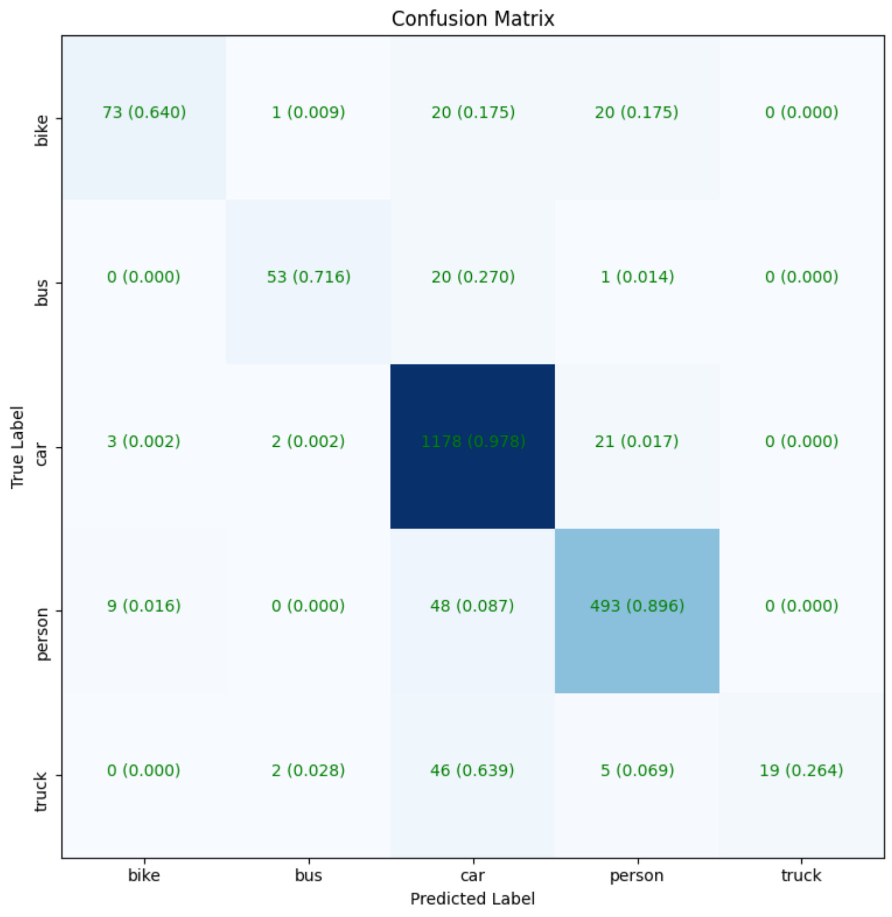


Figure 18: Naive model confusion matrix.

Training was only done for 4 epochs. In figure 18, we can see that the car & person classes have almost entirely converged, while the other minority classes are not performing particularly well, especially the truck class seems to generalize most trucks to cars.

Hyperparameter tuning was initiated with the various parameters listed in table 6. These combinations of parameters were used for 3 epochs each. As seen in table 5, the top five results have been listed. This process was very time consuming, each epoch took roughly 5 minutes each. Firstly, tuning was set to false so that it would take less than a week to generate the results (tuning decides whether or not to unfreeze the weights on the pre-trained model). The major drawback, however, is that 3 epochs is a far too low number. Initially the network should mature for several epochs, and learning curves would be analyzed to see if early stopping would be useful (in case of overfitting). Due to the greediness of the grid search, a lower epoch count was selected.

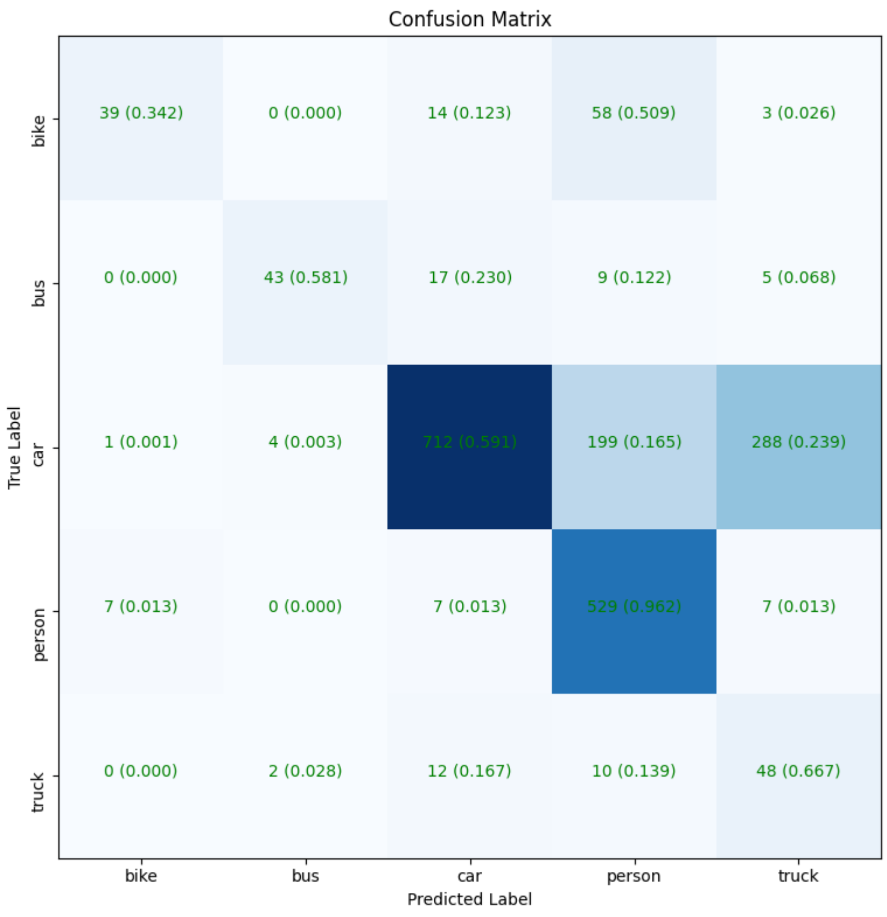
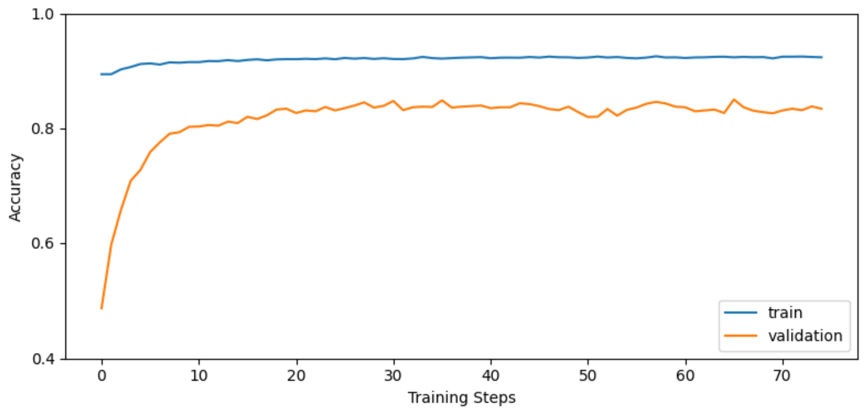


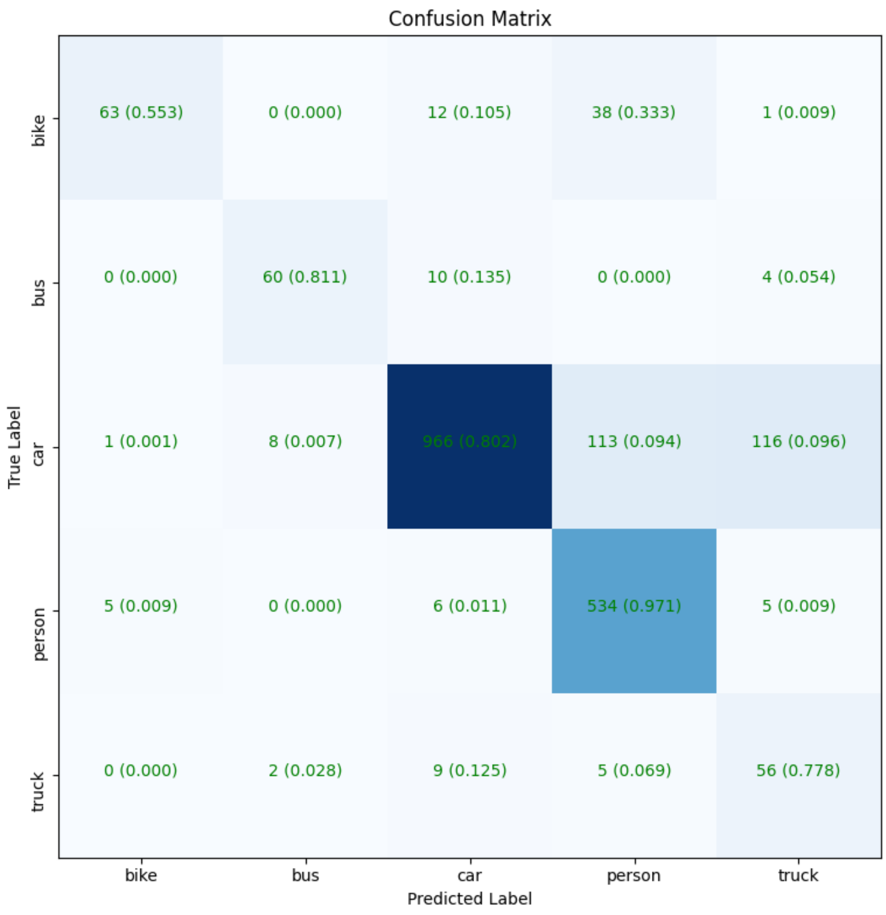
Figure 19: Tuned model confusion matrix, after 3 epochs.

As seen in figure 19, the tuned model performs worse on some classes compared to the naive approach, but when training with these parameters further for 80 epochs we get a slightly better result.

As seen in figure 20, the accuracy has improved, but not significantly. No hidden layers were used for the architecture here either.



(a) Learning Curve



	bike	bus	car	person	truck
bike	63 (0.553)	0 (0.000)	12 (0.105)	38 (0.333)	1 (0.009)
bus	0 (0.000)	60 (0.811)	10 (0.135)	0 (0.000)	4 (0.054)
car	1 (0.001)	8 (0.007)	966 (0.802)	113 (0.094)	116 (0.096)
person	5 (0.009)	0 (0.000)	6 (0.011)	534 (0.971)	5 (0.009)
truck	0 (0.000)	2 (0.028)	9 (0.125)	5 (0.069)	56 (0.778)

(b) Confusion Matrix

Figure 20: Training with the optimal parameters for 80 epochs.

Class imbalance challenges: to deal with data imbalance, two methods were tested. The first involves getting rid of data from the ExDark and KITTI set. The car class is removed from both, and the person class is removed from ExDark. Data was removed because some of the data collected online did not have entirely consistent annotation.

- 16 Batch Size.
- 0.001 learning rate, SGD.
- 0.1 dropout on the input layer.
- 0.1 label smoothing.
- 0.001 L1 regularization on the output layer.
- No hidden layer.

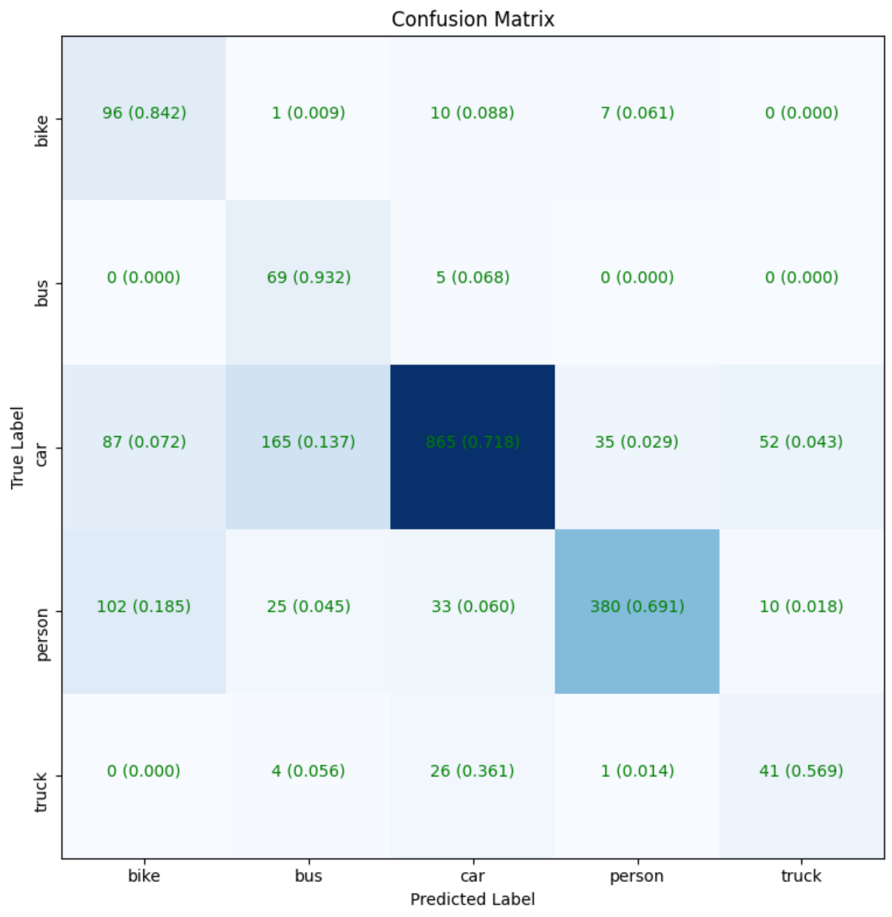


Figure 21: Smaller Dataset Confusion Matrix (100 epochs).

The smaller dataset, as shown in figure 21, seems to only separate bikes and buses better than the previous model. The second method to get around the imbalanced data is to use a custom data generator which weighs the smaller classes higher than the majority classes, implying that classes with low occurrence will have a higher probability to be present in a batch (as explained in the previous chapter).

The following approach, and approaches, used the network in figure 17.

- 32 Batch Size.
- 0.001 learning rate, RMSProp.
- 0.2 dropout on the input layer.

- 0.2 dropout on the hidden layer.
- 0.15 label smoothing.
- 0.001 L1 regularization on the output layer.
- 1 hidden layer.

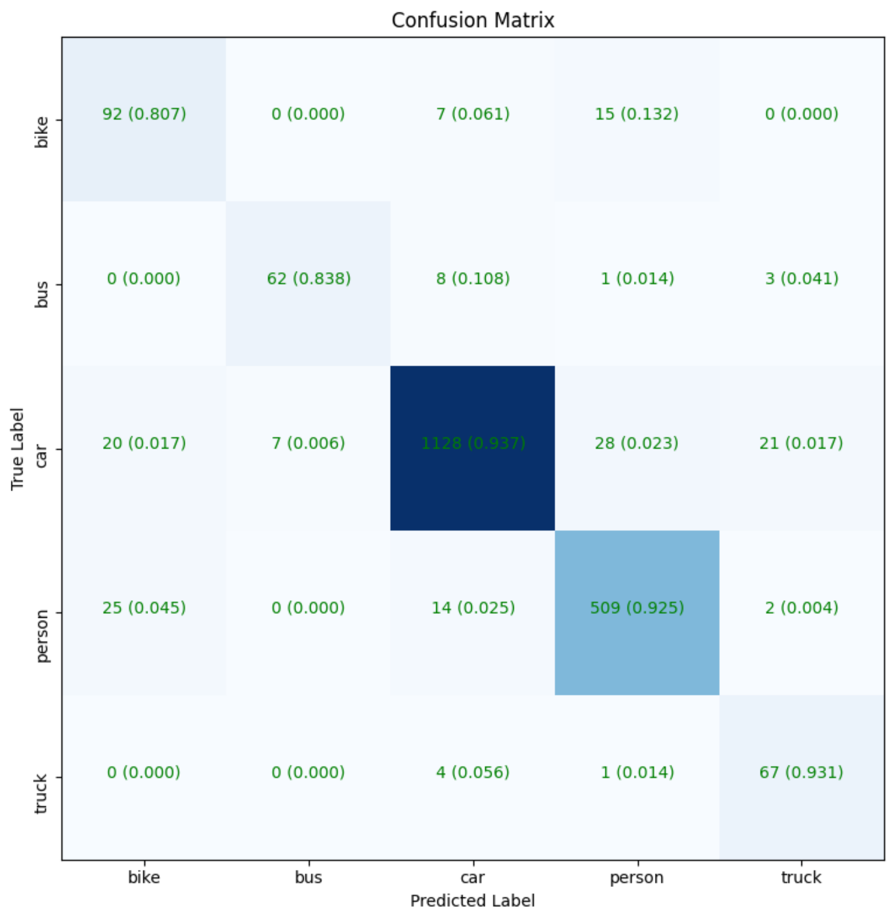


Figure 22: Training with a custom data generator, confusion matrix (65 epochs).

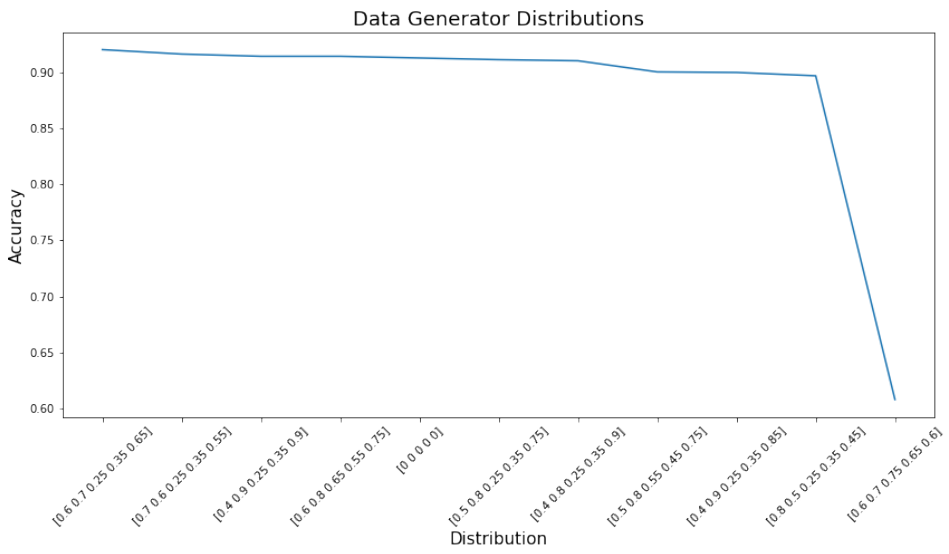
These results are slightly more acceptable, as in figure 22 all of the classes are separated by at least 80%. The model generally struggles to separate bikes and people, because the images with bikes normally have a person on it.

Evaluating Custom Data Generator: to verify whether or not the custom data generator actually makes a positive contribution, it has been tested with various probability distributions for the different classes. The distributions in table 7 have been tested.

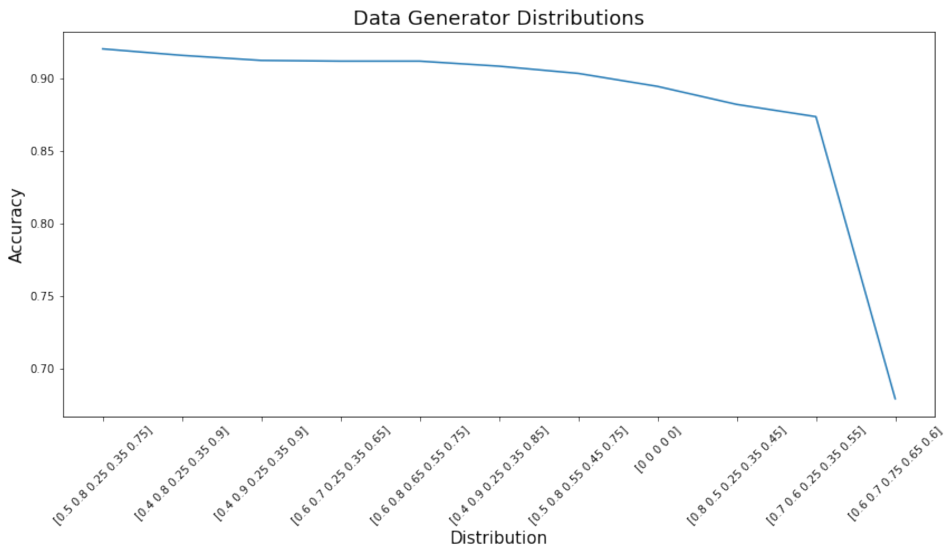
- 64 Batch Size, 20 epochs for each distribution.
- 0.001 learning rate, RMSProp.
- 0.25 dropout on the input layer.
- 0.25 dropout on the hidden layer(s).
- 0.15 label smoothing.
- 0.001 L1 regularization on the output layer.
- 1 hidden layer in the first attempt, 2 in the other.

Table 7: Distributions.

bike	**bus**	**car**	**person**	**truck**
0	0	0	0	0
0.50	0.80	0.25	0.35	0.75
0.60	0.70	0.25	0.35	0.65
0.70	0.60	0.25	0.35	0.55
0.80	0.50	0.25	0.35	0.45
0.40	0.90	0.25	0.35	0.85
0.40	0.80	0.25	0.35	0.90
0.40	0.90	0.25	0.35	0.90
0.50	0.80	0.55	0.45	0.75
0.60	0.80	0.65	0.55	0.75
0.60	0.70	0.75	0.65	0.60



(a) One Hidden Layer



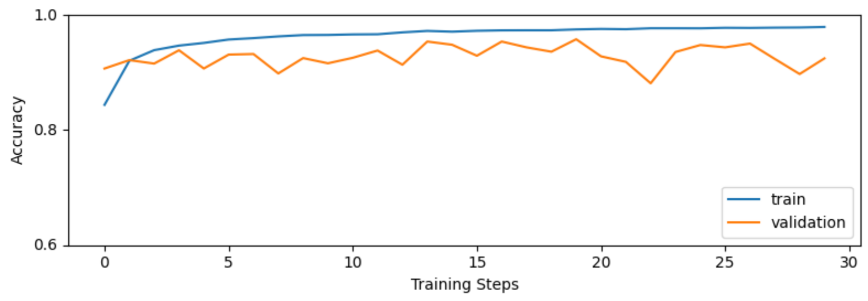
(b) Two Hidden Layers

Figure 23: High to Low Accuracy of various custom data generator distributions.

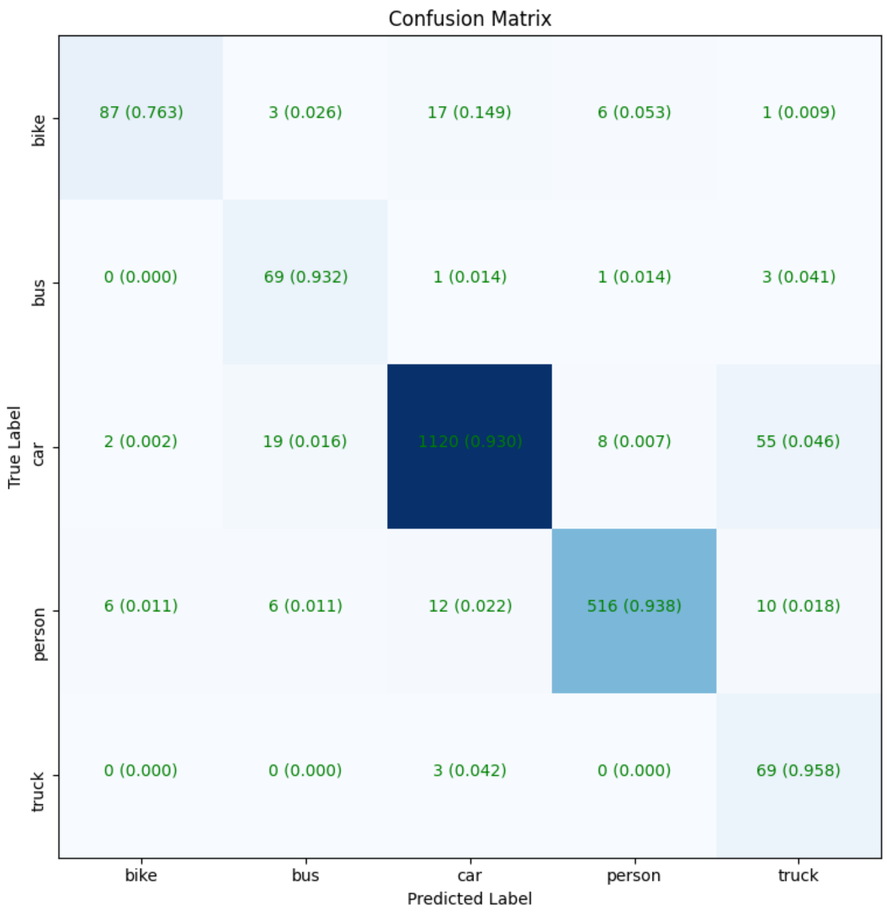
Figure 23 confirms that certain distributions matter, and that it generally is most sufficient when the probability for cars and persons are lower than the minority classes.

Unfreezing the weights: using the most suitable data generator distribution from the previous result, a new model was trained with this distribution $\begin{bmatrix}0.60 & 0.70 & 0.25 & 0.35 & 0.65\end{bmatrix}$ while unfreezing the weights. This will update the weights on the convolutional layers, such as the filters, during training, thus changing how features are extracted as well. This will help the model further adapt to the type of data used.

- 64 Batch Size.
- 0.001 learning rate, RMSProp.
- 0.1 dropout on the input layer.
- 0.1 dropout on the hidden layer.
- 0.15 label smoothing.
- 0.001 L1 regularization on the output layer.
- 1 hidden layer.



(a) Learning Curve



	bike	bus	car	person	truck
bike	87 (0.763)	3 (0.026)	17 (0.149)	6 (0.053)	1 (0.009)
bus	0 (0.000)	69 (0.932)	1 (0.014)	1 (0.014)	3 (0.041)
car	2 (0.002)	19 (0.016)	1120 (0.930)	8 (0.007)	55 (0.046)
person	6 (0.011)	6 (0.011)	12 (0.022)	516 (0.938)	10 (0.018)
truck	0 (0.000)	0 (0.000)	3 (0.042)	0 (0.000)	69 (0.958)

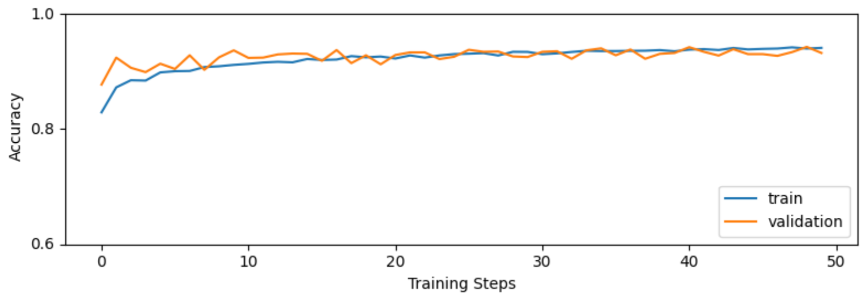
(b) Confusion Matrix

Figure 24: Training the model with tuning set to true and an optimal distribution (30 epochs).

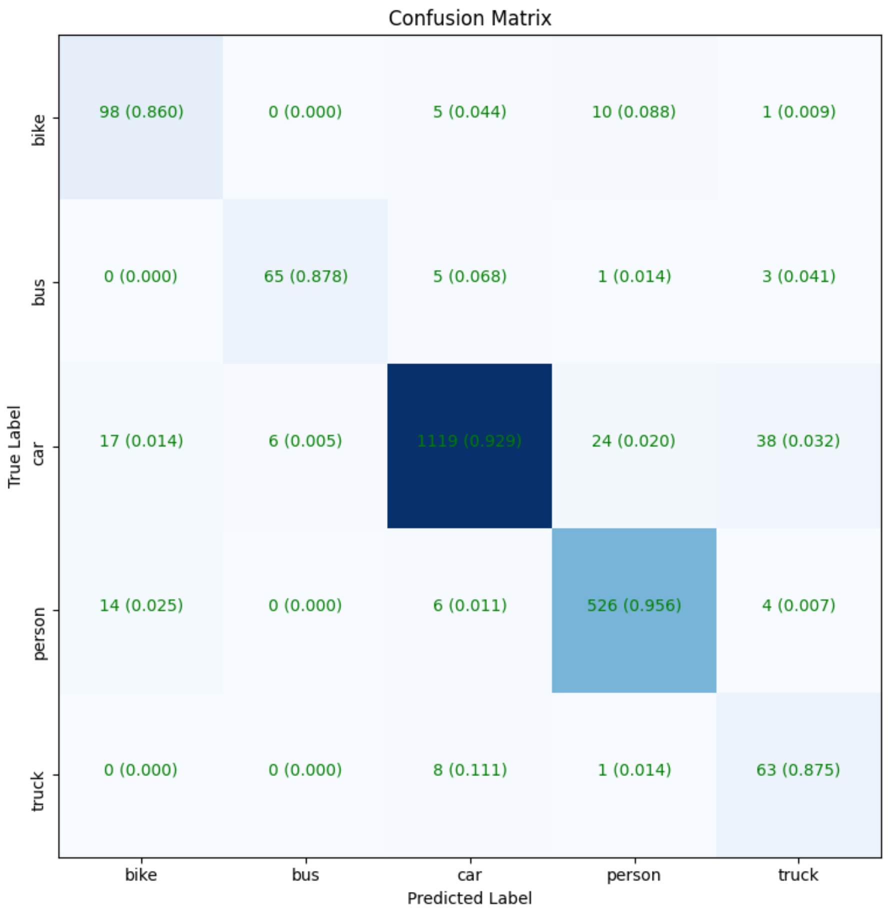
Training using a larger model: EfficientNet b7 [35] has also been tested.

This model contains roughly 66 M params (number of weights in the network), consequently increasing the overall training time. Additionally, this model expects images of size 600×600, which is much higher than the MobileNet v3 counterpart, making this model consume considerably more memory not only because the inputs are larger, but because the network is larger as well.

- $x_n = 2560$, more features are extracted here.
- 32 Batch Size.
- 0.001 learning rate, RMSProp.
- 0.25 dropout on the input layer.
- 0.25 dropout on the hidden layer.
- 0.15 label smoothing.
- 0.001 L1 regularization on the output layer.
- 1 hidden layer.



(a) Learning Curve



	bike	bus	car	person	truck
bike	98 (0.860)	0 (0.000)	5 (0.044)	10 (0.088)	1 (0.009)
bus	0 (0.000)	65 (0.878)	5 (0.068)	1 (0.014)	3 (0.041)
car	17 (0.014)	6 (0.005)	1119 (0.929)	24 (0.020)	38 (0.032)
person	14 (0.025)	0 (0.000)	6 (0.011)	526 (0.956)	4 (0.007)
truck	0 (0.000)	0 (0.000)	8 (0.111)	1 (0.014)	63 (0.875)

(b) Confusion Matrix

Figure 25: Training a model using EfficientNet b7 for 50 epochs with the optimal distribution.

The EfficientNet model did not increase performance considerably, barely 1%

difference can be noticed when computing the accuracy and comparing it to the previous model(s).

$$Acc_{EfficientNet} = \frac{98 + 65 + 1119 + 526 + 63}{2014} = 0.929 \approx 0.93$$
$$Acc_{MobileNet\ Tuned} = \frac{87 + 69 + 1120 + 516 + 69}{2014} = 0.924$$

4.4 Profiling

Multiple parameters have been measured to determine practical aspects of the methods developed throughout this thesis. Profiling has been done with and without a GPU. Predictions were made on 2014 successive images, with a batch size of 32 ($\approx$ 63 predictions per batch).

Parameters considered for the models, including both region proposal system and regular convolutional neural network, were:

- Loading Time
- Prediction Time
- Average Prediction Time
- Process Memory Consumption
- GPU Memory Consumption

All profiling was done with an Intel i5 750 CPU, Tesla P100 GPU (for general CNN) and Tesla V100 GPU (for region proposal system).

4.5 Background Subtraction

Under ideal conditions, such as having a sufficient FPS on the video stream, and a stabilized perspective, the algorithm is able to extract the objects of interest, as seen in figure 13.

Table 8: Profiling CNN, time in seconds, AVG predictions measure the time it took to predict each sample in a batch.

Type	**Model**	**Load Time**	**Pred Time**	**AVG Pred Time**	**Process Mem**	**GPU Mem**
CPU	MobileNet	19.314	94.046	1.517	281 MiB	0
	MobileNet Tuned	23.311	92.262	1.488	304 MiB	0
	EfficientNet	288.29	660+	???	???	0
GPU	MobileNet	4.45	24.866	0.401	1559 MiB	6422 MiB
	MobileNet Tuned	5.472	6.094	0.098	210 MiB	6422 MiB
	EfficientNet	54.347	79.309	1.279	2486 MiB	10518 MiB

Table 9: Profiling Region Proposal System, time in seconds, AVG predictions measure prediction time per image.

Type	**Model**	**Load Time**	**Pred Time**	**AVG Pred Time**	**Process Mem**	**GPU Mem**
CPU	MobileNet	55.683	73.867	0.130	797 MiB	0
	EfficientNet	132.883	300.550	0.527	1605 MiB	0
GPU	MobileNet	13.940	32.522	0.057	2124 MiB	2236 MiB
	EfficientNet	32.454	32.936	0.058	1212 MiB	2236 MiB

5 Discussion

Various models have been built and tested, all of them leverage transfer learning. One disadvantage can be that the feature maps generated are not entirely sufficient for the target data, but this can be dealt with by tuning the model further (unfreeze weights). Although if the transfer learning models have been trained on irrelevant data, it may not be sufficient regardless. In this case, all transfer learning models leverage relevant data, but there is also considerable amounts of irrelevant data due to the size of the data used (ImageNet / COCO).

5.1 Data Collection

Finding reliable sources for the data can be a challenge, especially when you have limited time for collecting and annotating data manually. The data collected online have some samples that can be misleading, such as annotations that cover multiple objects of the same class, or multiple objects of different classes. A dedicated camera for recording brief segments of traffic would also be useful, relying on the webcam stream did not provide a sufficient data rate, which in some cases resulted in poor performance of the background subtraction algorithm. The region proposal system was, however, not affected by this issue, mainly because this algorithm does not rely on successive frames to detect objects in motion.

5.2 Region Proposal & Object Detection

Training the region proposal network was considerably more challenging due to memory restrictions, seen as even the most lightweight models would exceed the capacity of a 32 gigabyte (GB) Video Random Access Memory (VRAM) Tesla V100 GPU. Using multiple GPUs would have been more feasible, but not very practical on a system where most of the resources are constrained.

More options could have been tested here, such as editing the pipeline configuration to use different optimizers, learn rates, etc. However, due to the constraints mentioned, the simpler method utilizing a CNN and background subtraction was favored.

5.3 Convolutional Neural Networks

Various models have been built and tested, relatively high accuracy has been achieved in the final models. There is, however, some challenges in increasing the accuracy further, due to data constraints. Some classes simply have too little data, and some data samples are not entirely consistent. There's a few samples in the data collected manually which are far too small, and some samples in the data collected online which have inconsistent annotation, such as larger portions of the image annotated as a single class rather than being split up (large occluded segments). These issues can potentially confuse the neural network. As seen in the confusion matrices in the previous chapter, there is a tendency for most classes to be generalized to cars (imbalance challenge).

Naive approach: in figure 18, the majority classes have been fairly well separated. However, the minority classes have a large portion generalized to the majority classes rather than their own respective classes.

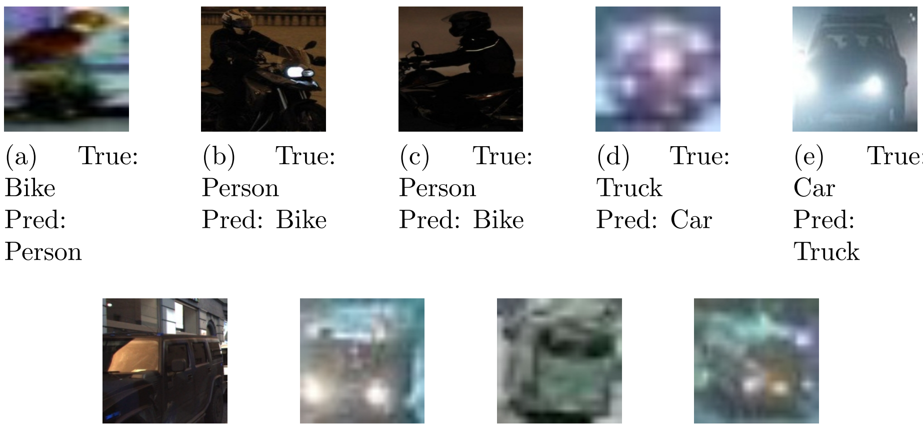
Hyperparameter tuning: this tuning procedure is not particularly helpful because the models have not been trained for a sufficient amount of epochs. Working with image data and deep neural networks make it challenging to train models within a sufficient time span.

Class imbalance challenges: lowering the amount of data worsens the performance, even when being trained for 100 epochs there is still severe misclassification amongst some of the classes (as seen in figure 21). Utilizing a custom data generator to directly influence the sampling procedure in a way which guarantees more minority classes to be present in a batch, did, however, improve the accuracy slightly. The increase is not considerably large, but it is not negligible.

Evaluating Custom Data Generator: a suitable distribution, one which maximizes the accuracy over at least 20 epochs, had to be found. This distribution was actively used in the newer models, and it did provide further improvements to the separation of the various classes. This is true because the generator increases the probability of minority classes being present in a batch, thus the network will converge early by taking all classes into account (as seen in figure 24).

Unfreezing the weights: this provided further accuracy increase, since unfreezing the weights of the transfer learning model allows the convolutional filters's weights to adapt further to the data being used. This will allow the network to capture further details, which could help separate the classes from one another.

Training using a larger model: in this case, the larger model does not provide considerable increase in accuracy. This could be related to the fact that there was too little data available (see figure 25). Nor does a larger model imply better accuracy, though a deeper network could capture details that are more significant, thus separating the classes with a higher accuracy. This model does, however, increase the accuracy of correctly classifying pedestrians, which is crucial.



(a) True: Bike Pred: Person

(b) True: Person Pred: Bike

(c) True: Person Pred: Bike

(d) True: Truck Pred: Car

(e) True: Car Pred: Truck

(f) True: Car Pred: Truck

(g) True: Bus Pred: Car

(h) True: Bus Pred: Person

(i) True: Bus Pred: Truck

Figure 26: Some random samples with misclassifications, from the test set (using the tuned MobileNet model).

Class Separation Challenges: In figure 26, various misclassifications are listed, and many of these images are either small, cropped to some extent, or have heavy illuminated headlights. Separation between bike and person can be tough, samples like these will confuse the network. Either bike and person should be merged to one class (person), or bike and person must be fully separate images (objects). Naturally, larger cars may accidentally be classified as trucks, or even busses, this is however not necessarily a big problem. More problematic is classifying vehicles as persons, or persons as vehicles, we see that in figure 24, there is only 1 bus classified as a person, however there are 6 persons classified as busses, 12 as cars and 10 as trucks. False positives and false negatives of this magnitude can lead to safety concerns. To improve on this, the data should be overhauled. Sufficient data collection for every class, with proper annotation, and possibly removing the bike class would be the next few steps to deal with this challenge.

5.4 Profiling

Table 8 and 9 showcases roughly how costly the two methods are. These results may vary depending on the hardware used, but it establishes a fair baseline regardless. The results indicate that the regular CNN method is faster, something which is expected (no greedy segmentation and region extraction needed, solely relying on convolutional filters and pooling to extract features). However, the region proposal approach is not particularly bad, as this approach predicts multiple objects per image at a decent rate. Even when using CPU only, it performs fairly well with 130 milliseconds per prediction. CNN is naturally faster in every regard because it only works with a single smaller image, the prediction time per image is $\frac{AVG\ Pred\ Time}{\frac{2014}{32}} = \frac{1.488}{63} = 0.0236\ sec$ for the tuned model.

As for the loading times, loading a region proposal model is taking longer due to more weights and general variables in these models. The EfficientNet CNN model used also suffers from this, so the loading time is significantly larger than the MobileNet counterparts. Again, this is due to the difference in model architecture, seen as EfficientNet has a huge amount of weights in comparison. The implications of high loading times could be problematic in a real time setting, as if for whatever reason the system crashes, it will take at least $\approx$ 20 seconds for the system to restart, and in this time multiple vehicles could enter the tunnel without being registered.

Lastly, the implications of minimal hardware is quite relevant in simple IoT devices. The results presented would require at least 300 megabyte (MB) Random Access Memory (RAM) and a decent single core CPU. However, using a GPU will provide much faster predictions, but will require at least 3 GB VRAM (region proposal system) or 7 GB (general CNN). These VRAM values may vary depending on how much VRAM Tensorflow decides to cache.

Using a tuned (unfrozen weights) MobileNet model seems to perform very well when used with a GPU. The process uses significantly less memory, most likely because the GPU is used more effectively on the tuned model than the regular model. A significant difference in prediction times can also be observed between the tuned and regular model, when using a GPU. It is, however, important to note that the GPUs used in these experiments are incredibly powerful in comparison to consumer grade GPUs.

5.5 Background Subtraction

Figure 13 showcases the detection algorithm used, however, due to limiting factors regarding the online video streams, it has been challenging to fine tune this algorithm. Most of the effort has been put into tuning the CNN, although the background subtraction algorithm developed is still capable, it is quite prone to noise, which typically is excessively generated when the video stream has a low throughput.

6 Further Work

Diving deeper into object detection frameworks like Detectron [36] from Facebook, and You only look once (YoLo) [37] would be an appropriate step towards building a more solid foundation. Relying on traditional methods like background subtraction is not recommended, as it is not particularly reliable due to weather, occlusion and lighting anomalies. Additionally, the bike and pedestrian class could possibly be merged into a single class, pedestrian. Also, naturally, collecting more data for the various classes and preferably having a separate test set with entirely unique data for better validation of the different models.

If, however, a classical approach is still preferred, having a dedicated camera available for real time testing would be beneficial. This would make it easier to fine tune the background subtraction algorithm used.

Additionally, collecting an explicit dataset with challenging samples, such as vehicles in poor illumination, floodlight blending, etc, would be useful to test the resilience of the system. Such data should preferably be collected during the autumn, because otherwise it will be challenging to find samples with poor illumination, due to how the sun is projected over Norway.

7 Conclusion

With the data collected, a system has been built and tested for detecting and classifying vehicles. Transfer learning has been heavily used in this work, and it did provide reasonable results. There are however some challenges related to further improving the performance of the system, mostly due to the collected data not being fully sufficient. The system has been trained on five classes, although the amount of classes can potentially be further generalized down to four. Due to the nature of bikes and pedestrians, there are challenges separating the two, especially when some samples have persons riding on bikes.

Other potential methods for further improving on the system as a whole has been presented as further work.

References

[1] F. J. Helland, "Nye råd etter tunnelbrann: Må bli lettare for folk å komme seg ut," *NrK.*

[2] paperswithcode, "Image classification on imagenet." `https://paperswithcode.com/sota/image-classification-on-imagenet`. Accessed: June 15, 2021.

[3] E. A. Thomessen, "Advanced vision based vehicle classification for traffic surveillance system using neural networks," pp. 1–118, 2017.

[4] E. Sudland, "Gjenkjenning av kjøretøy ved inn- og ut- kjøring av tunneler," pp. 1–122, 2016.

[5] S. C. Kleene, "Representations of events in nerve nets and finite automata," pp. 1–101, 1951.

[6] D. Hebb, *The Organization of Behavior.* Psychology Press, 1949.

[7] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," pp. 386–408, 1958.

[8] M. Olazaran, "A sociological study of the official history of the perceptrons controversy," p. 611–659, 1996.

[9] M. Minsky and S. Papert, "Perceptrons: An introduction to computational geometry," pp. 1–308, 1988.

[10] S. Haykin, *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[11] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989.

[12] P. L. Vu-Quoc, "Artificial neural network." `https://en.wikipedia.org/wiki/Artificial_neural_network`. Accessed: June 15, 2021.

[13] G. Hinton, "Overview of mini-batch gradient descent." `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: June 15, 2021.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[15] R. Jain, “3 types of gradient descent algorithms for small large data sets.” https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/. Accessed: June 15, 2021.

[16] S. Saha, “A comprehensive guide to convolutional neural networks — the eli5 way.” https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b116 Accessed: June 15, 2021.

[17] data hacker, “Cnn alexnet.” http://datahacker.rs/deep-learning-alexnet-architecture/. Accessed: June 15, 2021.

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[19] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013.

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[21] R. Girshick, “Fast r-cnn,” 2015.

[22] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” 2016.

[23] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” *Lecture Notes in Computer Science*, p. 21–37, 2016.

[24] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, “Spatial transformer networks,” 2016.

[25] I. Young, “Image analysis and mathematical morphology, by j. serra. academic press, london, 1982, xviii + 610 p.,” *Cytometry*, vol. 4, pp. 184–185, 09 1983.

[26] Y. P. Loh and C. S. Chan, “Getting to know low-light images with the exclusively dark dataset,” *Computer Vision and Image Understanding*, vol. 178, pp. 30–42, 2019.

[27] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *International Journal of Robotics Research (IJRR)*, 2013.

[28] Tzutalin, “Labelimg.” `https://github.com/tzutalin/labelImg`, 2015.

[29] fo40225, “Tensorflow prebuilt binary for windows.” `https://github.com/fo40225/tensorflow-windows-wheel`, 2021.

[30] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, “Speed/accuracy trade-offs for modern convolutional object detectors,” *CoRR*, vol. abs/1611.10012, 2016.

[31] Google, “Tensorflow object detection model zoo.” `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md`. Accessed: June 15, 2021.

[32] PyQt5. https://www.qt.io/. Accessed: June 15, 2021.

[33] Google, “Tensorflow hub, mobilenet v3, feature vector.” `https://tfhub.dev/google/imagenet/mobilenet_v3_large_100_224/feature_vector/5`. Accessed: June 15, 2021.

[34] R. Javadzadeh, E. Banihashemi, and J. Hamidzadeh, “T fast vehicle detection and counting using background subtraction technique and prewitt edge detection,” 2015.

[35] Google, “Tensorflow hub, efficientnet b7, feature vector.” `https://tfhub.dev/google/efficientnet/b7/feature-vector/1`. Accessed: June 15, 2021.

[36] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He, “Detectron.” `https://github.com/facebookresearch/detectron`, 2018.

[37] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv*, 2018.

List of Figures

List of Tables

List of Program Code

Abbreviations

Adam	Adaptive Moment Optimization
AI	Artificial Intelligence
ANN	Artificial Neural Networks
API	Application Programming Interface
AUC	area under the curve
AVG	average
AVX	Advanced Vector Extensions
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CT	computed tomography
CUDA	Compute Unified Device Architecture
FPS	frames per second
GB	gigabyte
GD	Gradient Descent
GiB	gibibyte
GMM	Gaussian Mixture Model
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HoG	Histogram of Oriented Gradients
HSV	Hue, Saturation, Value
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoT	Internet of Things

IoU Intersection over Union

KNN K Nearest Neighbour

MABO Mean Average Best Overlap

MAP Mean Average Precision

MB megabyte

MiB mebibyte

MSE Mean Squared Error

NN Neural Network

RAM Random Access Memory

R-CNN Regions with CNN Features

ReLu Rectified Linear Unit

RGB Red, Green, Blue

RMSProp Root Mean Square Propagation

ROC Receiver Operating Characteristic

RoI Region of Interest

SGD Stochastic Gradient Descent

SIFT Scale Invariant Feature Transform

SSD Single Shot Detector

STN Spatial Transformer Network

SVM Support Vector Machine

VRAM Video Random Access Memory

XML Extensible Markup Language

YoLo You only look once

Terms

convolved A filter (square matrix) is convolved with a target image, the filter slides across the image and transforms intersecting pixels

disk I/O Writing and reading to/from a disk (operations)

Max Norm (L1) $||w||_1 = max(\sum_{c=1}^{m} |a_c|)$, sum each column and pick the max

ReLu Rectified Linear Unit, $max(0, x)$

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

softmax $\frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

tanh Hyperbolic Tangent, $\frac{(e^x - e^{-x})}{(e^x + e^{-x})}$

greedy An algorithm is considered greedy if it performs CPU and memory intensive operations in a brute-force manner (naive or inefficient, but necessary)

gridsearch Try lots of parameter combinations, measure accuracy to see which combinations were best (hyperparameter tuning)

params Defines the complexity of a neural network, the number of params is the total amount of weights in the network

Frobenius Norm (L2) $||w||_2^2 = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$

vanilla Classical technique / regular technique

Appendix

.1 Code Hierarchy

- **preprocessing.ipynb** - Used for splitting the data into train and test, creates a file which holds the filenames rather than copying over the same files. Additionally it includes some histogram plots of the various vehicle distributions, and an algorithm for extracting the objects in the annotated images, such that they can be used in a regular CNN.
- **gen-tf-record.ipynb** - Generates the TF Record files from the train and test image folders. These folders contain the mapping files which hold images and their respective annotation file paths.
- **convert-data-labels.ipynb** - Is used to convert other annotation formats, and image formats to the correct formats. Various parsers and translation stuff logic is found here.
- **webcam-scraper** [**.ipynb, .py**] - Is used for fetching images from certain webcams.
- **utils.py** - Utility functions for plotting and parsing XML.
- **stream.py** - Realtime R-CNN / SSD based object detection system, utilizes PyQt5 for the GUI.
- **object-detection.ipynb** - Used for testing the R-CNN / SSD approach on single images.
- **object-detection-bgsub.ipynb** - Used for testing vanilla CNN with BG subtraction logic.
- **detection.py** - Contains object detection utility functions for the R-CNN / SSD approach.
- **eval-stat.py** - Used for creating plots for the evaluation of the custom data generator.
- **transfer-learning.py** - Used for training vanilla CNN, doing hyper-param tuning, testing distributions, etc.
- **testing.ipynb** - Misc testing, and various plotting.
- **profiling.py** - Used for profiling the CNN and region proposal method.

.2 Training a regular CNN

Navigate to **transfer-learning.py**, in the function *createModel*, alter the default parameters to fit your needs, then run the file in the usual way. To run tuning logic, supply an additional command line argument when running the file. Update the call to *gridSearchOptimize* with a different tuning method if necessary.

.3 Training a region proposal network

This procedure has been covered in the implementation section, download pre-trained models at the Tensorflow Object Detection Model Zoo, alter the pipeline configuration file, run the scripts mentioned in the implementation section.

.4 Resources

- Source Code + Data on GitHub
- Tensorflow Hub
- Tensorflow Object Detection API
- Tensorflow Object Detection Model Zoo
- Tensorflow Object Detection API Documentation
- Real Time Obj. Det. Demo