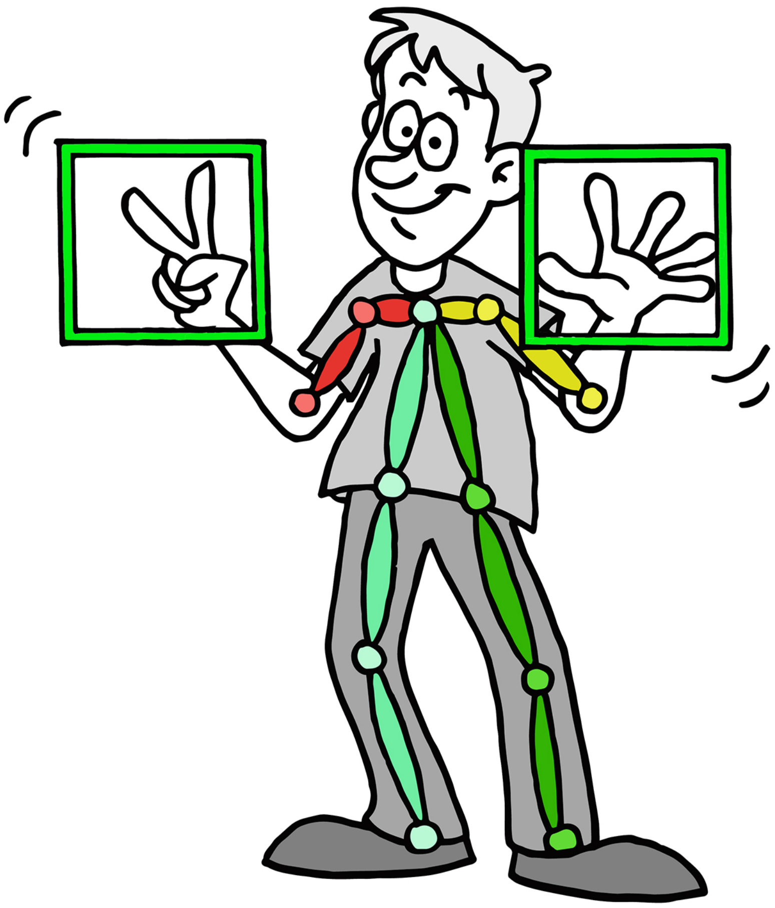
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Deep Learning based tracking framework for passenger monitoring

Master's thesis in Computer Science – algorithms, languages & logic and Computer Systems & Networks

Filip Granqvist
Oskar Holmberg

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis EX058/2018

A Deep Learning based tracking framework for passenger monitoring

Filip Granqvist
Oskar Holmberg

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

A Deep Learning based tracking framework for passenger monitoring

FILIP GRANQVIST
OSKAR HOLMBERG

Advisor: Niclas Wennerdal, Smart Eye AB
Supervisor: Lennart Svensson, Electrical Engineering
Examiner: Tomas McKelvey, Electrical Engineering

Master's Thesis EX058/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2018

A Deep Learning based tracking framework for passenger monitoring

Filip Granqvist
Oskar Holmberg

Department of Electrical Engineering
Chalmers University of Technology

Abstract

A substantial amount of traffic accidents are caused by distracted driving and drowsy behaviour each year. Two important areas when tackling these inadequacies is human-car interaction and the car's awareness of the passengers. Tracking passenger movements over time can result in a more intelligent vehicle that is able to react to passenger actions, and thereby provide a more secure driver experience.

In this thesis, we present a Deep Learning based tracking framework for passenger monitoring. The framework is divided into two separate parts: a human body pose estimator and an object detector. Additionally, we demonstrate the use case of the framework by developing a hand gesture classifier and a body action recognizer upon the framework. The framework and additional modules are optimized for efficiency, and we achieve real-time performance on a Nvidia Jetson TX2 embedded system.

Keywords: Computer vision, Deep learning, Body pose estimation, Action recognition, Object detection, Hand gesture classification.

Acknowledgements

We would like to thank Henrik Lind and Smart Eye for hosting this thesis and being supportive during the course of this project. Additionally we would like to thank Niclas Wennerdal, our onsite supervisor, and the AI team at Smart Eye for an enjoyable collaboration and a nice atmosphere. A special thanks to Lennart Svensson who acted as our academic supervisor for this thesis and provided superb insight.

Filip Granqvist & Oskar Holmberg, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1 Introduction

One important factor for traffic safety is the interaction between driver and car. For example, you want the driver to be able to interact with the car while still focusing on the road. Currently, the top cause of traffic accidents is distracted driving, which includes operating the vehicle, adjusting audio and cellphone usage [2].

Another security issue is the car's awareness of the passengers. An intelligent vehicle needs to be aware of every passenger's current position and their behaviour to be able to provide a driver experience with maximum security. For example, certain positions and rotations of a passenger's body will potentially make the use of airbags dangerous.

According to National Highway Traffic Safety Administration drowsy driving is the cause of over two percent of the fatal car crashes [3]. One indication of a person with strong drowsiness is unnecessary body movements. This includes movements such as touching face, adjusting sitting position and shrugging of shoulders [4]. By tracking passenger's movement over time, change in behaviour can be detected and thus prevent any accidents caused by human error before occurring.

With the aforementioned aspects in mind, Smart Eye, a technology company based in Gothenburg, has commissioned the investigation of how to expand their product segment through deep learning to provide increased passenger safety and comfort with the particular application of hand gesture recognition and tracking body key-points in mind. This is an open-ended request that enables this thesis to explore a broad range of deep learning techniques and allowed the goals to evolve to best suit the applicability of deep learning techniques in embedded environments.

1.1 Background

Deep Learning techniques in general, and Convolutional Neural Networks in particular, has dominated the area of Computer Vision in recent years. Along with the successes of Deep Learning, the number of applications for monitoring passengers inside vehicles has increased significantly. This can for example be used to detect drowsiness of the driver [5] and monitor stress levels [6]. Therefore, the solutions proposed in this thesis will be heavily influenced by Deep Learning techniques and state-of-the-art neural network architectures.

A way to track passenger motions over time is to extract their body poses from sensor data. The change of pose over time can be mapped to certain activities or behaviour within a car. An abnormal amount of shoulder shrugging indicates a drowsy behaviour, which can let the infotainment advise the driver to take a break [4]. Previous work by Bruce Xiaohan Nie et al. and Guilhem Chéron et al. have shown that estimating a human body pose as a pre-processing step for recognizing actions and behavior improves the performance and severely reduces the complexity of the problem [7] [8].

A method that may increase passenger safety is human-car interaction through gesture recognition. Instead of directly targeting the dashboard to perform commands to the infotainment system of the car, it can be done through the use of hand gestures. According to a market study performed by IHS Automotive, the global market for gesture recognition within cars expects to grow from 700,000 in 2013 to more than 38 million units in 2023 [9]. The use case for hand gestures also expects to grow in 2018 due to the new law being put into use in Sweden, which prohibits motorists from using their mobile phone while driving [10].

An existing passenger monitoring system currently in production is *Depthsense Car Library* by *SoftKinetic*, wich was recently acquired by Sony. The library can be divided into two categories: in-cabin monitoring and in-vehicle infotainment control [11]. For in-cabin monitoring the library can extract body pose information from the passengers. This can be used for automated adjustment of seats and mirrors, and biometric recognition among others. For in-vehicle infotainment control, the library supports hand tracking which can be used to classify a variety of hand gestures which can be mapped to actions such as *answer call*.

The Depthsense Car Library is the most well-known deep learning based implementation for passenger monitoring. However, they are gaining information through the use of multiple sensors, such as depth cameras, which significantly reduces difficulty of the tracking problem because of the rich multimodal data available. Smart Eye's product segment as of today is focused on the use of a single near infrared camera. As a result, this increases the difficulty of the problem, but with the benefit of only requiring a single camera to implement the solution in a vehicle.

1.2 System overview

Smart Eye has mandated this thesis work to investigate how state-of-the-art deep learning techniques can be used in order to increase passenger safety with the subject of human-car interaction in mind. This thesis answers this open-ended request in the form of a deep learning based tracking framework for passenger monitoring. The tracking framework is developed in a generic way to be freely integrated into more explicit solutions for passenger safety and comfort such as models for automated adjustment of seats and mirrors, and observing passengers in a variety of ways. In particular, Smart Eye has commissioned two standalone prototypes that puts the framework to use in practice to support *action recognition* and *hand gesture classification*.

As input to the system Smart Eye will provide one vision based sensor. Desirably, the proposed solution should support near infrared cameras as it is the most common sensor in their product segment as of today. The aforementioned vision based sensor is going to be placed above the dashboard inside a car. The exclusive use of vision based sensors restricts this thesis to only explore computer vision based solutions. As a result, our architecture will make a substantial use of Convolutional Neural Networks (CNNs) in combination with other deep learning techniques that are found useful.

The tracking framework. The intent of the tracking framework is to extract useful passenger information from a given image into a more intelligible feature representation, which can be used to solve a multiple of problems within a car. The tracking framework should at least be able to extract the information that is found useful from the driver, but desirably all the passengers. The framework consists of two modules with separate tasks, an *object detector* and a *human body pose estimator*. As a starting point, the object detector is specifically trained to locate hands, but can be extended to find arbitrary objects. The body pose estimator is responsible for detecting a number of upper body keypoints and matching them into human poses.

Action recognition. A use case for the body pose estimator from the framework is to map a given pose to a specific action and/or their position in relation to the car in question. For example, a body pose can be mapped to actions such as *both of the driver's hands on the steering wheel.* Having a pose estimation stage as a preprocessing step for action recognition can significantly reduce the complexity of the problem. The body pose estimator can potentially be used for several other tasks, such as detecting drowsiness of the driver and detecting if the position of passengers makes the use of airbag dangerous.

Hand gesture classification. To reduce the amount of distractions for a driver, such as directly targeting the infotainment system of the car, hand gestures can be used. By recognizing hand gestures, they can be translated into commands such as *increase volume*, *change song* and *answer call.* A minimum requirement is to be able to successfully develop an architecture that supports static hand gestures, while support for dynamic hand gestures is a bonus.

Real-time integration. The solutions presented in this thesis are implemented with the goal of working real-time in an embedded environment with limited resources. As a result, this thesis will experiment suitable trade-offs between performance and efficiency. All models are to be deployed, and able to operate, on a Nvidia Jetson TX2 embedded platform. In Figure 1.1 the entire pipeline of the described system can be seen.

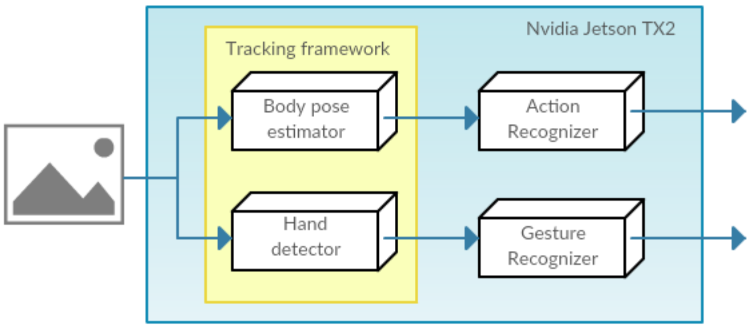


Figure 1.1: Overview of the system to be implemented.

1.3 Objectives

To evaluate the overall system as described in the previous section, a set of measurable objectives were composed. All implementations will be assessed with relevant metrics on the most popular data sets and compared with pre-existing state-of-the-art implementations both in terms of performance as well as efficiency. The objectives are divided into a group of minimal requirements and a group of nice-to-have features. The minimal requirements are divided into the following objectives:

- A tracking framework with the support for tracking arbitrary objects and body keypoints.
- The tracking framework should be able to extract sequences of hands in a video.
- The tracking framework should be able to extract human body poses in a video for at least the driver.
- There should be support for classifying static hand gestures from the tracking framework for at least the driver.
- There should be support for recognizing human body actions from the tracking framework for at least the driver.
- The entire system should be able to operate real-time on a Nvidia Jetson TX2 embedded platform.
- The real-time implementation should not waste computing power performing deep inference even when its not needed.

In addition, depending on time resources and applicability, this thesis will investigate and potentially implement the following nice-to-have features:

- Dynamic hand gestures should be supported, i.e. gestures that are performed as a movement over time.
- Multi-person pose tracking should be supported to enable tracking of every passenger present inside the car.
- The object detection module and pose estimation module should be merged into a single multi-task neural network for increased efficiency.
- Models should be trained on gray-scale images to enable support for near infrared camera input.

1.4 Delimitations

Due to the requirement of our solution being integrated in a car with limited computer power available, a few delimitations has to be made regarding our implementation. We will not be able to utilize the power of the very deep neural networks, but instead focus on making shallower networks more efficient. The execution time of the algorithm is also of high importance to be suitable for a real-time setup, which restricts our vocabulary of deep learning algorithms available.

A majority of the state-of-the-art computer vision algorithms are nowadays influenced heavily by the field of deep learning. This doesn't make other algorithms obsolete in any way, but in this thesis, we will mainly focus on investigating the potential of deep learning algorithms for our purpose.

The implementation part of this thesis should result in a prototype, which means that there are no demands for the implementation to be production ready.

Another delimitation of our implementation is to restrict the input to a single RGB camera. One can argue for including other sensor data and even multiple cameras that might be able to improve the results, but our implementation is restricted to be cost-effective using a single camera.

One of the most important parts of any Deep Learning project is obtaining data with labels of high quality. Gathering data and labeling will, if not completely necessary, not be a part of this thesis. Neither is any of Smart Eye's existing data sets relevant to the problems this thesis will face. As a result, only data sets found online will be considered. Therefore, there may be a disparity between general data sets and the environment inside a car. However, we will still reason around how the models to be implement may be used in such environments.

For a true measurement of efficiency on the Jetson TX2 platform, the models should be implemented in C++ using Nvidia's real-time framework for deep learning called TensorRT. Unfortunately, the Python API for TensorRT has not yet been released for the Jetson series. Therefore, we exclude any TensorRT implementation from our scope because of time constraints, and instead focusing on efficiency measurements using GPU accelerated Tensorflow implementations.

1.5 Thesis Outline

The remainder of this report is divided into seven chapters: Deep Learning for computer vision, Hand detection, Hand gesture classification, Action recognition, Real-time system and Conclusions. In Deep Learning for computer vision we go through general background theory. The models with their respective previous work, implementation and result is divided into three standalone chapters: Hand detection, Hand gesture classification and Action Recognition. Finally, this report will wrap up with a real-time system implementation of our proposed tracking framework along with the gesture classifier and action recognizer, and a chapter summarizing our conclusions that are connected to our initial analysis of the problem and our objectives of the thesis project.

2

Deep Learning for Computer Vision

This thesis will only consider the use of vision based sensors. Thus, the solutions presented will be based on Computer Vision techniques. In the wake of the success from Machine Learning in general and Convolutional Neural Networks(CNNs) in particular, Deep Learning has been a dominating part in recent year's state-of-the-art solutions for Computer Vision. The following sections will go through the general Deep Learning techniques and theories used in this thesis to solve said problems.

2.1 Supervised Learning

Machine Learning tasks can roughly be divided into three categories: *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning.* This thesis will almost exclusively make use of Supervised Learning for our models. Put in the simplest of terms, Supervised Learning can be described as learning a function $f : X \to Y$, i.e. mapping a given input X into a predicted output Y. By considering examples, a model predicts an output $\hat{Y}$ and thereafter adjust its parameters for the prediction becoming more close to the ground truth Y. By iterating through examples, the model performs this correction until it has fit the data.

Supervised Learning is extensively used to learn models to deal with *classification* tasks as well as *regression* tasks. In the case of a classification task, the purpose is to train a model to output a discrete categorical representation of the given input. An example could be to predict if a given image represents a cat or a dog. In contrast, the output from a regression task is continuous, for example predicting a housing price given its size and location.

2.2 Deep Learning fundamentals

Deep learning is a subset in the field of Machine Learning. A definition by Microsoft goes as following: “A class of Machine Learning techniques that exploit many layers of non-linear information processing for Supervised or Unsupervised feature extraction and transformation, and for pattern analysis and classification” [12]. This includes deep neural networks with several hidden layers between the input and output layer. The following sections will go through the basic building blocks for implementing a deep neural network architecture.

2.2.1 Artificial Neural Networks

Inspired by biological neural networks, Artificial Neural Networks (ANN), consists of a set of artificial neurons. By considering examples, as in Supervised Learning, an ANN is able to learn underlying attributes and features of the very problem it is trying to solve.

The most common Neural Network architecture, a Feed-forward Neural Network, can be seen in Figure 2.1. The input, for example an image, is fed through a multiple of hidden fully connected layers, which are made up of a set of *neurons*. The data is forward-propagated through the layers, until output layer, which are used to predict either classification or regression tasks. The parameters that is used to approximate the function $f: X \to Y$ are called *weights* and *biases*. Each edge in Figure 2.1 represents a learnable weight which is multiplied with the output from the previous layer.

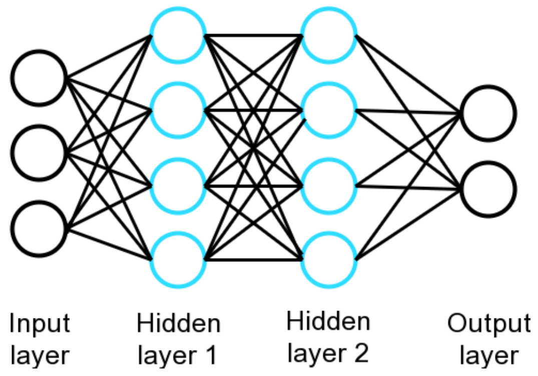


Figure 2.1: A Feed-forward Neural Network with two hidden layers. All layers are fully connected.

The output of a neuron of the hidden layers all follow the *perceptron* algorithm presented by Frank Rosenblatt in 1957 [13], which mathematically can be seen in Equation 2.1 and visualized in Figure 2.2. In a fully connected layer, each output from the previous layer is multiplied with a weight and then summed together with a bias, a constant that moves the activation function right or left. The sum is then passed on through an activation function, making a non-linear transformation of the input.

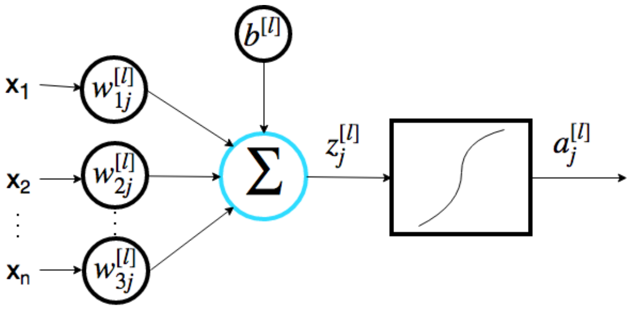


Figure 2.2: Neuron j at layer l. The input neurons are multiplied with their respective weights and them summed together with the bias. The output from the sum $z_j^{[l]}$ is then propagated through an activation function, resulting in the output $a_j^{[l]}$

The mathematical representation of the perception algorithm is defined as

$$\mathbf{a}^{[l]} = g(\mathbf{z}^{[l]}) = g(\mathbf{w}^{[l]T}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}), \tag{2.1}$$

which introduces the notation that will be used throughout this thesis. $a^{[l]}$ is the output from layer l, which comes as a result of the weights $w^{[l]}$ multiplied with the output from the previous layer $a^{[l-1]}$, added with the bias $b^{[l]}$ and propagated through an activation function g.

2.2.2 Activation functions

Activation functions are the part of the network that perform non-linear transformations. Without them a neural network would work as a linear regression model, and not able to solve non-linear problems. The activation functions are applied at each neuron of a layer, performing a non-linear transformation over the input signal, i.e. weights, inputs and bias.

The most commonly used activation functions in a Deep Learning perspective is the sigmoidal function, tanh and the ReLU function, which will be described in detail in the following subsections.

Sigmoid
The sigmoid function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

and squashes its input between 0 and 1. In the early stages of Machine Learning, this was a common choice of an activation function. It behaves as a saturating neuron in real life [14]. However, the sigmoid function had some unattractive characteristics that were absent in it's successors. Saturating neurons were actually a problem since it makes the gradient vanish for large and small values of the input. In addition, the exponential value in the sigmoid function is expensive to compute. The sigmoid outputs are neither zero-centered, making back-propagation less efficient because the gradients are either going to be all positive or all negative. [14].

Tanh
Similiar to the sigmoid function, the tanh function squashes it's inputs, but in between -1 and 1:

$$f(x) = tanh(x). \tag{2.3}$$

This solved the issue with the outputs from neurons not being zero-centered when using the sigmoid function. Nevertheless, the tanh function still makes gradients vanish when saturated.

ReLU
The Rectified Linear Unit (ReLU) function outputs the maximum of either its input or zero, i.e. outputs the positive part of its arguments:

$$f(x) = max(0, x). \tag{2.4}$$

It was proved to be much more efficient than its predecessors tanh and sigmoid [14]. ReLU has several desirable characteristics, such as not saturating (in the positive region), computationally efficient, converge 6 times faster than tanh and sigmoid and is more biologically plausible [14].

In addition to the original ReLU function there exists several variants. Notable mentions is the Noisy ReLU which adds some Gaussian noise, Leaky ReLU, which multiplies the input with a small number if less than zero, and ELU, which tries to make the mean activation closer to zero. The purpose of the substitutes were originally to improve the performance, but in practice the gain is often negligible in relation to the extra computation.

Softmax
For classification tasks, the output from a neural network represents a score for each class. In order to calculate the probability of a given class in relation to the other possible classes the softmax function can be used, creating a categorical probability distribution over the set of classes. The probability for a given class i is defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}, \tag{2.5}$$

which like the sigmoid function, squashes the input score between zero and one. This is done by exponentiating the scores and normalizing them through division with the sum off all exponents, making all outputs together add up to 1.

2.2.3 Loss functions

When dealing with classification and regression tasks within Machine Learning we need to define a way to measure how good the predicted output is in relation to the ground truth. This is done using a *loss function*, or *cost function* as it also often is called, that measures the error of the prediction.

Consider the scenario where we have three categories; A, B and C. Then given a sample from category A, it would have a ground truth probability vector $\mathbf{y} = [1, 0, 0]$, and a network predicts $\hat{\mathbf{y}} = [0.2, 0.4, 0.4]$. The loss functions purpose is to measure the cost of the deviation, $\hat{\mathbf{y}}$ in comparison to $\mathbf{y}$, which is $L = \frac{1}{3}((1-0.2)^2 + (0 - 0.4)^2 + (0-0.4)^2) = 0.43$ in the case of mean squared error.

Cross-entropy
One way to measure the deviation between the network prediction and the ground truth is the loss function cross-entropy, or log loss as it is also called, which is defined as

$$L(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i). \tag{2.6}$$

The error output by the cross-entropy increases logarithmically with the deviation from its ground truth label. Cross-entropy loss is often used in classification tasks as it offers a way to compute the divergence of two probability distributions, the true distribution y and the predicted distribution $\hat{y}$.

Mean squared error
A common way of measuring the cost of regression tasks is the *mean squared error* loss function, defined as

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{2.7}$$

As the name indicates, the function will measure the average of the squares of the deviations between every network output $\hat{y}_i$ and the ground truth y_i.

2.2.4 Mean Average Precision

Mean Average Precision (mAP) is a common evaluation metric in information retrieval. To understand mAP, one must first have an understanding of the precision and recall metrics. Precision is defined as

$$precision = \frac{\#\ true\ positives}{\#\ true\ positives + \#\ false\ positives} \tag{2.8}$$

and can be interpreted as the percentage of how many of your predictions that are correct. Recall is defined as

$$recall = \frac{\#\ true\ positives}{true\ positives + \#\ false\ negatives} \tag{2.9}$$

and can be interpreted as the percentage of total positive samples that were recognized as positive by the classifier.

Using mAP as an evaluation metric is a common choice for computer vision regression tasks, but calculating precision and recall uses statistics from a confusion matrix, which requires a classification task. For example when converting an object detection problem into a classification task, a threshold on the Intersection-over-Union can be used [15], and for converting a pose estimation problem into a classification task, a threshold on the Object Keypoint Similarity (OKS) can be used [1].

The process of calculating mAP is to first calculate the confidence of the predictions for every sample in the dataset. Thereafter, the predictions are sorted by the confidence levels in descending order, and precision and recall is thereafter calculated for the top k predictions, ranging from $k = 1$ to $k = K$, where K is some number less than the number of predictions. The result is a table of precision and recall values for different top k predictions. Thereafter, AP is calculated by

$$AP = \frac{1}{11} \sum_{r \in 0.0, 0.1, \ldots, 1.0} \max_{\tilde{r} \leq r} precision(\tilde{r}), \tag{2.10}$$

where *precision(r)* is the maximum precision for any recall values exceeding r, which is retrieved by looking at the table of precision and recall values from the top k evaluations. In other words, to calculate the average precision as seen in Equation 2.10, you iterate over different minimum recall values, $r \in \{0.0, 0.1, ..., 1.0\}$, and find the maximum precision value present in the table given the recall constraint, and then averages the precisions. AP is calculated for each class separately, and to calculate mAP, AP is simply averaged across the available classes [16].

2.2.5 Backpropagation

When training any artificial neural network, the gradients of each trainable weight with respect to the loss function needs to be calculated. The gradient of a weight with respect to the loss function will tell in which direction the weight vector should move towards to output a more favourable prediction. The procedure of computing the gradients are known as the backpropagation algorithm.

By utilizing the chain rule of derivatives, the gradients are calculated recursively through each layer, starting at the loss function. The gradients to calculate are $\frac{\partial L}{\partial \mathbf{w}^{[l]}}$ and $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$ for each layer l. To first calculate the gradients that belong to layer l, the gradient first has to be propagated from layer $l + 1$ down through the activation function of layer l, called $\boldsymbol{\delta}^{[l]}$. This gradient can be implemented using the recursive formula

$$\boldsymbol{\delta}^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l+1]}} \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \boldsymbol{\delta}^{[l+1]} \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \sum_j \delta_j^{[l+1]} w_{ij}^{[l]} g'(\mathbf{z}^{[l]}), \tag{2.11}$$

where **z**, **a** and g has the same representation as in Section 2.2.1 about feed-forward networks.

The formula for the gradients $d\mathbf{w}^{[l]}$ and $d\mathbf{b}^{[l]}$ at layer l can now be expressed in terms of the gradient that flows through from the previous layer:

$$d\mathbf{w}^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{w}^{[l]}} = \boldsymbol{\delta}^{[l]} \mathbf{a}^{[l-1]}, \tag{2.12}$$

$$d\mathbf{b}^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{b}^{[l]}} = \boldsymbol{\delta}^{[l]} \cdot 1. \tag{2.13}$$

What remains is the special case of calculating the gradient from the loss to the last layer L,

$$\boldsymbol{\delta}^{[L]} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[L]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}} \frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}. \tag{2.14}$$

Here, $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}}$ is the partial derivative of the loss function, and will vary depending on what loss function is used, for example cross-entropy or MSE. Lastly, $\frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}$ is the partial derivative of the output activation function, for example softmax or linear.

When the gradients has been calculated, they can be used in the optimization procedure to change the weights in a way that moves the predicted output closer to the ground truth.

2.2.6 Optimization

The optimization, also called training for Artificial Neural Networks, performs the weight updates using the gradients received from backpropagation. The standard training loop is described in Algorithm 1. The gradients are calculated by measuring the loss of a batch of predictions using the backpropagation algorithm. The optimizer is thereafter responsible for updating each weight.

```
1 $X, Y$ ← Get next batch of training data;
2 $Y'$ ← Predict using a forward pass of the batch $X$;
3 $\nabla$ ← Calculate the gradients using backpropagation with respect to the loss
    $L(Y', Y)$;
4 foreach Weight $w_i$ and Bias $b_i$ in the network do
5     $w_i$ ← update the weight using an optimizer $OPT(w_i, \nabla)$
6     $b_i$ ← update the bias using an optimizer $OPT(b_i, \nabla)$
7 end
```

Algorithm 1: The training procedure for updating the weights to result in predictions closer to the ground truth.

The ancestor of all neural network optimization strategies is called Stochastic Gradient Descent (SGD), and its update formula for a weight **w** is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\partial L}{\partial \boldsymbol{w}}. \tag{2.15}$$

The weight **w** is updated by a fraction η of the computed gradient, which is called the learning rate [17].

The stochastic part of SGD comes from the fact that only a single data sample is used to measure the loss and compute the gradients. A more common approach nowadays is calculating the gradients using the average of a mini-batch of samples from the dataset at once (usually a collection of 2-64 samples). This modification is called Mini-batch Gradient Descent, but it is common to use the words mini-batch and batch interchangeable in the context of deep learning.

Even when averaging the gradients from a batch of samples, Mini-batch Gradient Descent is still pretty stochastic in nature. Momentum is a method for helping the gradient vectors to accelerate in the right direction by instead using a rolling mean of gradients to update the weights, which changes the update rule into

$$\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} + \eta \frac{\partial L}{\partial \boldsymbol{w}}, \tag{2.16}$$

$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v}_t. \tag{2.17}$$

This means that γ percent of gradients from previous steps will be taken into consideration before making the update. A simple explanation of the impact of momentum is to think of v_t as the velocity of parameter change at time t and the gradient acts as a small change in velocity. A common value of γ is 0.9.

The most popular optimizer strategy as of the writing of this thesis is called Adam. In this optimization strategy, a learning rate is maintained for each network weight and separately adapted as learning unfolds. With this feature, the learning procedure is much more effective and it has been shown that using Adam can often lead the network to the state of convergence much faster than other optimizers [18].

The formulas for implementing Adam are shown in Equations 2.18-2.22, where $\otimes$ is element-wise multiplication and $\oslash$ is element-wise division. $\mathbf{m}_{d\boldsymbol{\theta}}$ is a rolling mean of the first moment of the gradients,

$$\mathbf{m}_{d\boldsymbol{\theta}} = \beta_1 \mathbf{m}_{d\boldsymbol{\theta}} + (1 - \beta_1) \frac{\partial L}{\partial \boldsymbol{\theta}}, \tag{2.18}$$

very much like Equation 2.16 for momentum. $\mathbf{s}_{d\boldsymbol{\theta}}$ is calculated in a similar way,

$$\mathbf{s}_{d\boldsymbol{\theta}} = \beta_2 \mathbf{s}_{d\boldsymbol{\theta}} + (1 - \beta_2) \frac{\partial L}{\partial \boldsymbol{\theta}} \otimes \frac{\partial L}{\partial \boldsymbol{\theta}}, \tag{2.19}$$

but has the responsibility for storing a rolling mean of the second moment of the gradients, i.e. the squared gradients.

$$\mathbf{m}_{d\boldsymbol{\theta}} = \frac{\mathbf{m}_{d\boldsymbol{\theta}}}{1-\beta_1} \tag{2.20}$$

and

$$\mathbf{s}_{d\boldsymbol{\theta}} = \frac{\mathbf{s}_{d\boldsymbol{\theta}}}{1-\beta_2} \tag{2.21}$$

perform bias corrections on $\mathbf{m}_{d\boldsymbol{\theta}}$, and lastly

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \mathbf{m}_{d\boldsymbol{\theta}} \oslash \sqrt{\mathbf{s}_{d\boldsymbol{\theta}} + \epsilon} \tag{2.22}$$

performs the weight update, where η is the base learning rate and β_1, β_2 and ϵ are hyper-parameters specific to Adam. Common default values are: $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$. In summary, there exist a velocity $\mathbf{m}_{d\boldsymbol{\theta}}$ and a oscillation dampener $\mathbf{s}_{d\boldsymbol{\theta}}$ for each trainable weight $\boldsymbol{\theta}$. Keeping a separate instance of these properties for every weight makes Adam into the highly adaptive optimizer it is.

2.2.7 Weight decay

Weight decay is a common regularization technique to improve generalization. To implement weight decay, an additional term is added to the loss function,

$$L = L_0 + \frac{1}{2}\lambda \sum_i w_i^2. \tag{2.23}$$

The term includes the squared value of every weight in the network and is sometimes called L_2 regularization. The higher in magnitude of a weight, the higher the loss is, which results in the optimization procedure preferring weights with low magnitudes. λ is a constant used for tuning the strength of the weight decay.

The effect of penalizing large weight magnitudes are twofold. Firstly, it minimizes any irrelevant components of the weights by choosing the smallest weight that solves the problem. Secondly, a good choice of λ results in a model more robust to noise due to increased bias in the network, hence the regularization [19]. There exists other types of weight decays, such as penalizing the absolute value of every weight, called L_1 regularization, but it is not as popular as L_2 regularization.

2.2.8 Dropout

Fully connected layers within a neural network often hold a huge number of parameters and is therefore prone to overfitting, where the network fails to perform well on data points that has not previously been seen. To overcome this issue, Hinton et al. developed a new regularization technique called *Dropout* [20]. Randomly “turning off” a percentage of all neurons within a fully-connected layer at each training iteration have shown to make a network less prone to overfitting, and thus making it more robust and better at generalizing. This is because at every iteration, a different subset of neurons are turned off, and therefore the entire layer can not rely on a small set of neurons to forward propagate the information and instead every neuron is forced to learn valuable features.

2.2.9 Batch Normalization

Batch Normalization is another technique that provides regularization, but it is actually not its main purpose. Instead, the purpose of Batch Normalization is to make the optimization procedure more stable. It lowers the importance of careful initialization of weights, enables the use of higher learning rates, and can significantly reduce the amount of training steps needed to converge. The idea of Batch Normalization is to normalize the output of a neural network layer before passing through the activation function. Normalizing the inputs of a neural network to make the learning easier is a well known fact, and Batch Normalization sees any output of a neural network layer as the input to a smaller subsequent network [21].

The formulas for implementing batch normalization are presented in Equations 2.24 through 2.27. Firstly, the mean μ_B is computed by

$$\mu_B = \frac{1}{N}\sum_i z^{(i)} \tag{2.24}$$

and variance σ_B^2 is computed from only the current batch output z by

$$\sigma_B^2 = \frac{1}{N}\sum_i (z^{(i)} - \mu_B)^2, \tag{2.25}$$

where $z^{(i)}$ is a particular sample from the batch. Secondly, the normalized version of the original output $z^{(i)}$ is calculated in by

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \tag{2.26}$$

where ϵ is a small number to avoid division by zero. Lastly, $z_{norm}^{(i)}$ is scaled into a more appropriate representation

$$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta. \tag{2.27}$$

What is a more appropriate scale is determined during training, i.e. γ and β are trainable parameters. Because μ_B and σ_B^2 are computed using only the current batch and not the entire dataset, the variables will include a little bit of noise, and this causes Batch Normalization to also have a regularization effect.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN), or ConvNet, is a subclass in the field of Neural Networks that is specifically designed to perform well on inputs with spatial dependencies, such as images. Just like Feed-forward Neural Networks, ConvNets consists of neurons, with their respective weights and biases. However, using the fact that the input to the network has spatial dependencies, opens up for several architectural choices to make the network much more efficient. An issue with Feed-forward Neural Networks when working with image classification is that the network size grows tremendously as the image grows in size. For example an image size of $200 \times 200 \times 3$ would result in 120,000 weights for the first layer.

Inspired by the connectivity arrangement of neurons in animals visual cortex, ConvNet operations are divided into receptive fields (filters), only responsible for a small region of its input. In contrast to FC layers, a ConvNet will, as the name indicates, convolve with the input using *filters* or *kernels*. Compared to traditional approaches such as Viola-Jones [22], the filters to find patterns such as edges, will not be engineered by hand. Instead the network itself will figure out the relevant filters during training.

ConvNets are composed of an input layer (commonly a 2D image), a number of hidden layers and finally the output layer. In addition to the Fully Connected layers described in Section 2.2.1, the hidden layers for ConvNets are made up of Convolutional layers as well as Pooling layers. An example architecture can be seen in Figure 2.3.

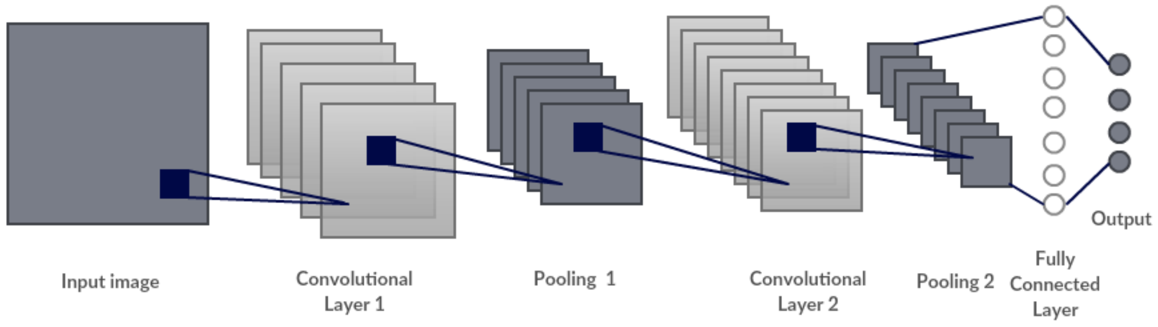


Figure 2.3: A common ConvNet structure. Input is the raw image, which is then followed up by a number of convolutional and pooling layers stacked upon each others. The network then ends up with a fully connected layer into the output classes.

2.3.1 Convolution

The core part of the ConvNet architecture, the convolutional layer, consists of several filters of trainable weights. The filters are convoluted across the input image, resulting in a new set of images called *feature maps*. What those feature maps represents is up to the network to learn through backpropagation. For instance this could be vertical edges and horizontal edges in the earlier layers, and more abstract objects in the later layers.

Mathematically, the output from one convolution with filter size $K_x \times K_y \times D$, where D is the depth of the input (for example color channels at the first layer and features maps thereafter) can be described as

$$a^{(l)}_{k,n} = \phi(z_{k,n}) = \phi(b^{(l)} + \sum_{i=0}^{K_x} \sum_{j=0}^{K_y} \sum_{d=0}^{D} w^{(l)}_{i,j,d} a^{(l-1)}_{k+i-\frac{K_x}{2}, n+j-\frac{K_y}{2}, d}). \tag{2.28}$$

An example of one convolution operation can be seen in Figure 2.4. A $2 \times 2 \times 1$ filter is convolved with a $4 \times 4 \times 1$ input. The same filter is applied at all locations of the input, implying a *stride* of size 1, i.e. the filter is just slided one step at the time over the input volume.

As seen in Figure 2.4, the $2 \times 2 \times 1$ filter will just slide three times in the horizontal and vertical direction, resulting in an output size of $3 \times 3 \times 1$. If this is done consecutively, the initial image will be down-sampled significantly, especially if a large filter size is used. In order to preserve the spatial size *padding* can be used. The most common way to do this is through zero-padding, which increases the input image with zeros around the border, which will help preserving the input size.

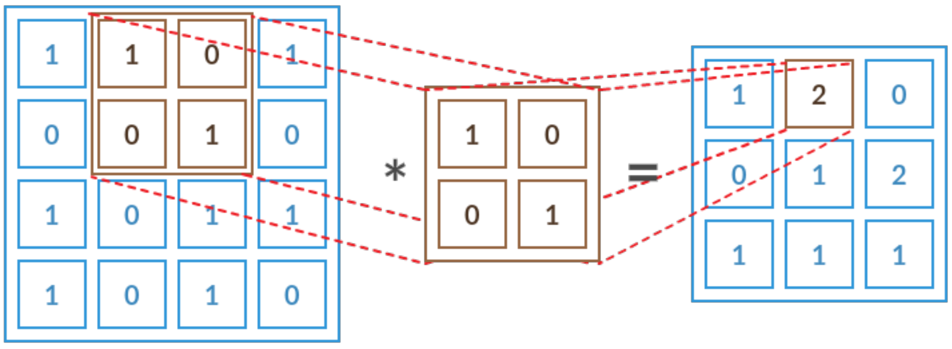


Figure 2.4: A convolutional operation performed on an input size of 4×4 with filter size 2×2. Stride is set to 1, thus only moving 1 step at a time, resulting in a output size of 3×3. No padding is used, which implies a down-sampling.

Another example with the same input and filter size, but with stride 2 can be seen in Figure 2.5. The filter will slide over the input 2 steps at a time, resulting in a larger down-sampling of the input.

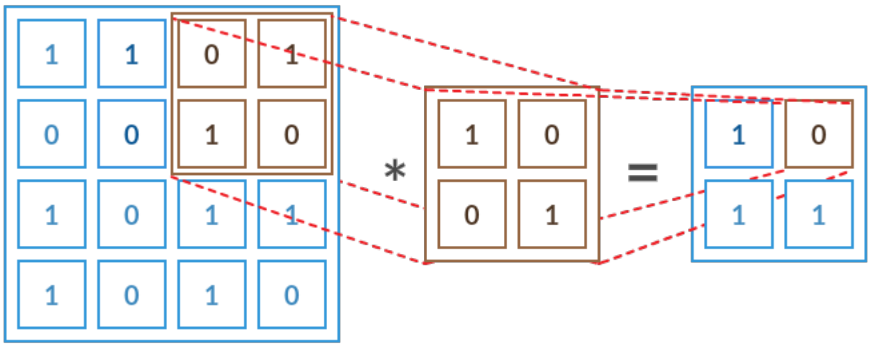


Figure 2.5: A convolutional operation performed on an input size of 4×4 with filter size 2×2. Stride is set to 2, thus sliding 2 steps at a time, resulting in a output size of 2×2. No padding is used.

3D convolution
Instead of 2D inputs, CNNs can also handle 3D volume inputs by increasing the dimension of the filters to 3D as well. In contrast to a 2D CNN, which strides the filter over input width and height on all feature maps from the previous layer and thus generating a 2D output, the 3D convolution strides in depth as well, creating another 3D dimensional output from one filter.

The 3D convolution is illustrated in Figure 2.6. As aforementioned, the depth of the filter d is smaller than the entire depth D. The filter will as a result stride over all input dimensions, width, height and depth respectively.

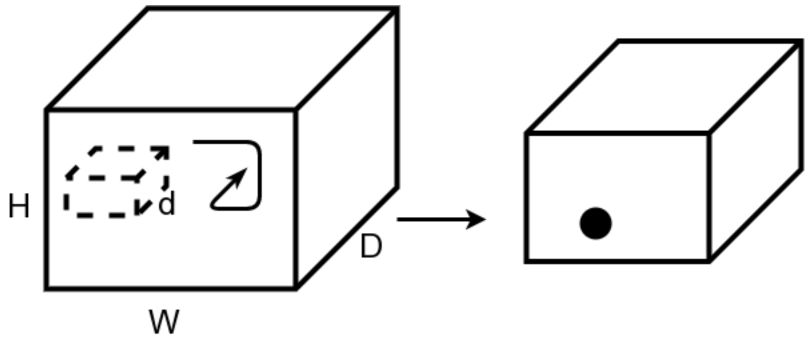


Figure 2.6: A 3D convolution is performed over the input volume. It strides over the input width, height and depth.

2.3.2 Pooling

The Pooling Layer is a building block in the ConvNet architecture used to truncate the spatial size of its input into a more dense representation [23]. The pooling operation will merge a number of neurons from the previous layer using a predetermined operation into a single neuron in the next layer [24]. The most common type of pooling is *max pooling*, which extracts the maximum value from a set of neurons and passes it on to next layer. An example of the max Pooling operation can be seen in Figure 2.7. Just like when convolving a filter, the pooling operation will convolve with the input.

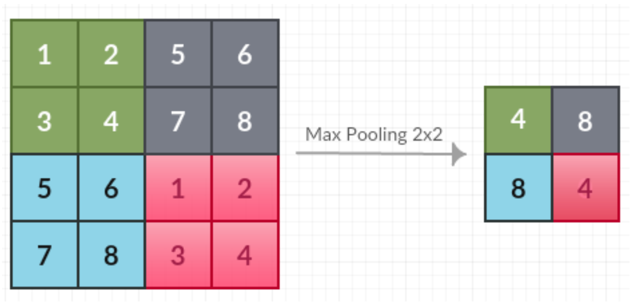


Figure 2.7: A max pooling operation applied with stride 2. Resulting in an output size of 2×2, reduced from its input size 4×4.

2.4 Recurrent Neural Networks

Recurrent Neural Networks (RNN), is a type of Artificial Neural Networks that is primarily used in order to extract features within the temporal domain in order to be able to handle problems involving sequences of data. In contrast to Feed-forward networks, RNNs also allow for a feedback loop with the use of an *internal state*, a memory that works as a link between forward-propagations of different time steps, thus allowing to save sequential information. An example of a recurrent connection can be seen in Figure 2.8, where the hidden state $\mathbf{h_t}$ is determined as a function of the previous hidden state $\mathbf{h}_{t-1}$ and the new input $\mathbf{x_t}$. RNNs are commonly used in order to solve problems such as speech recogniton, language translation and video classification.

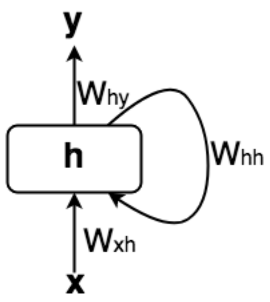


Figure 2.8: A vanilla RNN architecture. The input from the previous layer at the current iteration is combined with the internal state, generating an output to the next layer and updating the internal state.

An unrolled version of the RNN can be seen in Figure 2.9 with the entire sequence of length n.

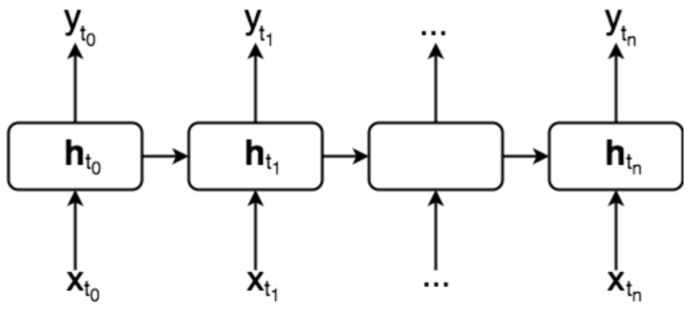


Figure 2.9: An unrolled RNN network, where the hidden state is passed on to its successor of the sequence.

2.4.1 Long Short Term Memory

While the standard versions of RNNs are good for capturing short term dependencies, they become less effective as the relation gap between the input and the output grows in time and tends to suffer from exploding- or vanishing gradients [25] [26]. In order to handle long term dependencies, Hochreiter and Schmidhuber developed a new RNN architecture, *Long Short Term Memory* [27].

In contrast to Vanilla RNNs, a basic LSTM implementation consists of two feedback connections instead of one: the hidden state h, and the cell state c. The hidden state represents the output from the LSTM from the previous iteration, while as the cell state represents the memory that is kept from the earlier stage. In addition to the feedback connections the output from the previous layer, x, is fed into the LSTM. The hidden state and the cell state will be manipulated and updated through different types of *gates* as seen in Figure 2.10.

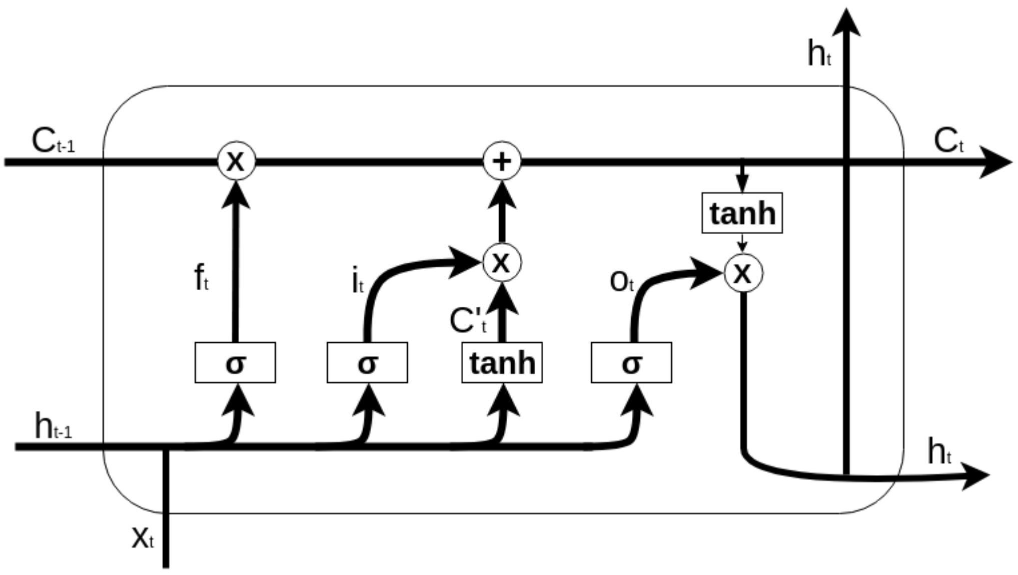


Figure 2.10: Internal representation of an LSTM cell. The input to the LSTM cell is the previous cell state, hidden state and the input from the previous layer at the current iteration.

Forget Gate
The first operation within the LSTM cell is the *Forget Gate*,

$$f_t = \sigma(\mathbf{W}_f \cdot [h_{t-1}, x_t] + b_f), \tag{2.29}$$

which decides what information from each element that are to be passed on from the previous cell state.

As mentioned earlier, the sigmoid function outputs a value between zero and one. A value of zero would result in completely forgetting the element in the cell state from the previous iteration and a value of one to keep it altogether. As input to the sigmoid a weight is applied on the previous hidden state and the new input, together with the bias [28].

Input Gate
After deciding what information to keep from the previous cell state, the LSTM decides which values that are to be updated. This is done through the *Input Gate*,

$$i_t = \sigma(\mathbf{W}_i \cdot [h_{t-1}, x_t] + b_i), \tag{2.30}$$

which applies a sigmoid activation on the new input and the previous hidden state [28].

Candidate Values
After the Input Gate decides which values to update, new *Candidate Values*, C'_t, will be proposed to be added to the new cell state. A tanh activation function is applied on the input and the previous hidden state:

$$C'_t = \tanh(\mathbf{W}_c \cdot [h_{t-1}, x_t] + b_c). \tag{2.31}$$

Cell State
After the candidate values have been proposed, the new updated *Cell State* will then be defined as the of the old state multiplied with the forget gate added together with the new candidate values:

$$C_t = f_t * C_{t-1} + i_t * C'_t. \tag{2.32}$$

Hidden State
Finally the output of the LSTM cell, the *hidden state*, is set by

$$o_t = \sigma(\mathbf{W}_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.33}$$

and

$$h_t = o_t * \tanh(C_t). \tag{2.34}$$

A sigmoid gate, o_t, will be used to determine which information from the cell state that are going to be passed on as output from the cell. A tanh activation function is then applied on the cell state, resulting in the output from LSTM cell.

2.5 Transfer Learning

Transfer Learning is a method that is commonly applied in Deep Learning projects to ease the training procedure. Several Deep Learning tasks faces deficiencies such as limited data available, which often lead to overfitting. Instead of initializing weights of a neural network randomly, a common approach is to copy a well-known network architecture with pre-trained weights on a closely related problem.

For image classification tasks, there exists several CNN architectures with their respective weights that have been trained on the large dataset ImageNet with 1.2 million images for classification [29]. The weights at the first layers are often frozen and not updates during transfer learning as they are already trained for extracting common features from images. A common case is to only make the last layers of the network trainable to fine-tune them for the new task.

2.6 CNN Architectures

A fact in visual recognition problems is that the human brain is naturally good at it while machines have historically been particularly bad at it. For example, detecting and classifying objects comes very easily for humans. In the recent couple of years this fact has however become less of an issue because of the developments of Convolutional Neural Networks. Over the years there have been several architectures implemented with both their pros and cons. Many architectures are developed solely to score a high accuracy on data sets such as the ImageNet, which result in a large model size, and thereby not always applicable in a real-time scenarios [30][31].

2.6.1 LeNet

The first practical implementation of a Convolutional Neural Network was presented by LeCun et. al in 1998. Inspired by the work presented by Hubel and Wiesel in the 50s which showed that visual cortexes consists of neurons that are individually responsible for small regions of the visual field [32]. With LeCun's invention, simple recognition tasks could be automated, such as recognizing hand-written numbers. The network architecture consisted of CONV-POOL layer combinations followed up with two Fully Connected layers. The convolutional layers used 5×5 filters with stride 1, whilst the pooling layers used 2×2 applied at stride 2 [33]. The activation function of choice at this time was the sigmoid function.

2.6.2 AlexNet

Inspired by the ideas of LeNet, AlexNet was the first real breakthrough of ConvNets, designed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton [24]. It was the first large scale neural network that was able to achieve state-of-the-art performance on the ImageNet classification challenge. For the yearly ImageNet classification challenge, it severely outperformed all non Deep Learning based approaches. AlexNet performed the record breaking top 5 test error rate of 15.4%, in contrast to the second best, non deep-learning based approach, of 26.2%.

The network architecture followed the same principles as LeNet - simply with more layers [24]. In total, the network architecture consists of 5 convolutional layers, max-pooling layers, dropout layers and 3 fully connected layers. In contrast to LeNet, AlexNet introduced the ReLU activation function, to the disadvantage of the sigmoid/tanh functions.

As AlexNet was the first substantial network of this kind it is often used as a comparison in benchmarks in terms of performance, number of parameters and model size.

2.6.3 VGGNet

VGGNet was the runner up at the yearly ImageNet classification challenge in 2014. Even though it failed to deliver state-of-the-art in terms of accuracy, it had other characteristics that historically have showed to make the architecture noteworthy anyway. In the original paper, *Very Deep Convolutional Networks for Large-Scale Image Recognition* [34], the authors demonstrate that the VGGNet performs well on other data sets outside of ImageNet, i.e. the feature representations generalize well. This is an indication that this architecture will be exceptional for transfer learning tasks, which is also established by the community as it is the most favored choice for extracting features from images.

The architecture follow its predecessors AlexNet and LeNet, with exclusive use of convolutional- and pooling layers stacked upon each other. It was architectured with simplicity and depth in mind. All convolutional layers are applied with a 3×3 filter with stride 1, and 2×2 pooling layers with stride 2. The big jump from AlexNet's 11×11 filter size to 3×3 drastically reduced the number of parameters per layer, and model size was instead increased by focusing on building a deeper architecture.

2.6.4 GoogLeNet

For the winner of the ImageNet classification challenge in 2014, GoogLeNet, things drastically started to change from the conventional CONV-POOL layer combinations. The earlier, fully sequential process, where either a convolution or a pooling operation was applied at each step, was replaced with the *Inception* module. Different convolutional filter sizes as well as max pooling operations was now being applied at all steps in parallel. The results from all operations are then concatenated together [35] and passed on to the next layer.

The architecture proposed in the paper stacked the *Inception* module together with the occasional conventional convolution layer or pooling layer. In total it resulted in a 22 layers deep network(counting only depth), with over 100 layers in total. Fully connected layers was not part of the GoogLeNet, instead an average pooling layers was used at the end. Together with the 1×1 filters used within the *Inception* module GoogLeNet actually reduced the number of parameters in comparison with AlexNet by a number of 12.

2.6.5 ResNet

In 2015, the creative nature continued with ResNet, short for Residual Network - the winner of the ImageNet classification challenge the same year. A common issue with previous conventional ConvNet implementations was the fact that adding more layers actually made the network perform worse. The hypothesis by the Microsoft team designing ResNet was that larger networks were harder to optimize. If optimized correctly, deeper networks should perform *atleast* as good or better than shallow network architectures.

ResNet was designed with a different approach than typical ConvNets. Rather than trying to learn features, it tries to learn residuals. The residual innovation, the *Residual block*, consists of a conv-relu-conv series. Given an input x to the Residual block, it propagates through the conv series and constitutes a result F(x). The output from the last convolutional layer is then added together to the original input x, the residual connection. The resulting output of a residual block if therefore $H(x) = F(x) + x$, and instead of learning $H(x)$ directly, the residual $H(x) - x$

is learned instead, allowing for the signal to easier flow through the layers during backpropagation [36].

This new approached made it possible to stack many additional layers on top of each other without causing vanishing gradients. The winning architecture of the ImageNet challenge used 152 layers in depth, with an error rate of 3.6% - better than human performance.

2.6.6 SqueezeNet

The development of new neural network architectures has mainly focused on improving the accuracy for different challenges. The purpose of SqueezeNet is to instead maintain the accuracy of previous architectures, while focusing on reducing the network size instead. Reducing the network size has many benefits, for example smaller models implies less memory references and by extension requires less energy. In addition, smaller models often runs faster, especially if they fit into SRAM [37].

The authors of SqueezeNet had two strategies in mind when designing a neural network architecture that should have competitive performance and minimal number of parameters. Firstly, limit the number of 3×3 filters and prefer 1×1 filters instead because it simply has the fewest parameters (9 times fewer than a 3×3 filter). Secondly, decrease number of input channels of 3×3 filters. This resulted in the fire module [38].

A fire module, see Figure 2.11, consists of a squeeze layer and an expand layer. Using 1×1 filters in both the squeeze layer and the expand layer is a result of the first strategy mentioned, and the squeeze layer is a result of the first strategy mentioned, limiting the number of input layers to the expand layer. A fire module can be stacked like any other convolutional layer to form a deep architecture. The original SqueezeNet is shown in Table 2.1, consisting of one convolutional layer, followed by eight fire modules, and lastly another convolutional layer. To keep the number of parameters small, there are no fully connected layers at the top of the network. The resulting SqueezeNet architecture has a x50 reduced size compared to AlexNet, but performs just as well.

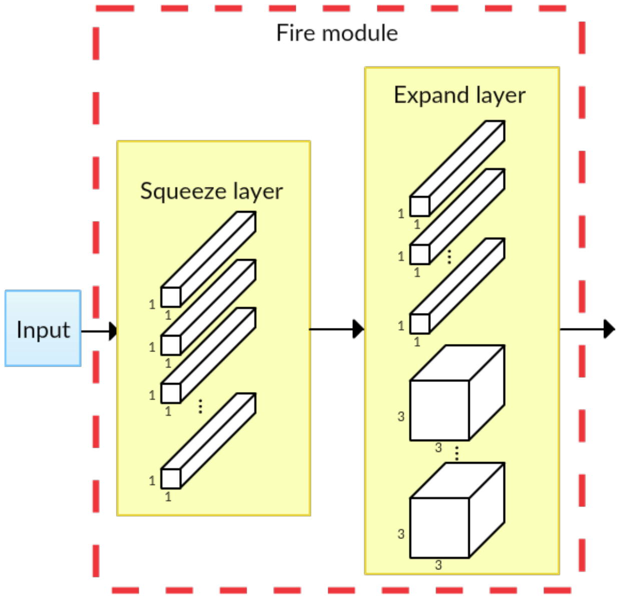


Figure 2.11: The architecture of a fire module

Type	Filter size / Stride	# Filters
Conv	7×7 /2	96
Max Pooling	3×3 /2	-
Fire module	-	$s_{1\times1} = 16$, $e_{1\times1} = 64$, $e_{3\times3} = 64$
Fire module	-	$s_{1\times1} = 16$, $e_{1\times1} = 64$, $e_{3\times3} = 64$
Fire module	-	$s_{1\times1} = 32$, $e_{1\times1} = 128$, $e_{3\times3} = 128$
Max Pooling	3×3 /2	-
Fire module	-	$s_{1\times1} = 32$, $e_{1\times1} = 128$, $e_{3\times3} = 128$
Fire module	-	$s_{1\times1} = 48$, $e_{1\times1} = 192$, $e_{3\times3} = 192$
Fire module	-	$s_{1\times1} = 48$, $e_{1\times1} = 192$, $e_{3\times3} = 192$
Fire module	-	$s_{1\times1} = 64$, $e_{1\times1} = 256$, $e_{3\times3} = 256$
Max Pooling	3×3 /2	-
Fire module	-	$s_{1\times1} = 64$, $e_{1\times1} = 256$, $e_{3\times3} = 256$
Conv	1×1	1000
Global Avg pooling	Pool 13×13 /1	-

Table 2.1: Overview of the SqueezeNet architecture. Global average pooling averages over every input channel.

2.6.7 MobileNet

Like SqueezeNet, MobileNet is a CNN architecture aiming for efficiency [39]. It builds upon the theory of separable convolutions. Separable convolutions is characterized by factorizing a convolution with a regular filter up into two stages, illustrated in Figure 2.12. The first stage, called the depthwise convolution, uses a $k \times k \times 1$.

Instead of the filter also including the depth, a depth of 1 is used and the filter is instead applied once for each input channel. The second stage, called the pointwise convolution, uses a $1 \times 1 \times M$ filter to combine the outputs of the depthwise convolutions. In theory, the depthwise convolution should be able to learn spatial patterns within an input map, and the pointwise convolution should be able to learn any patterns between these maps, enabling similar learning capacities as a standard convolution filter but with many fewer parameters.

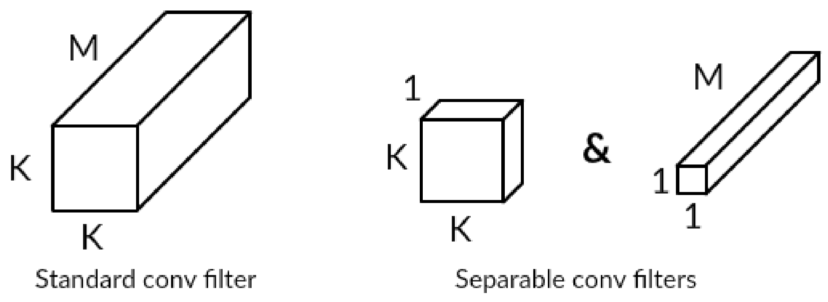


Figure 2.12: The depthwise and pointwise filter of a separable convolution

Assume $D_K \times D_k$ is the dimensions of an input channel, M is the input channel depth, D_F is the filter size and N is the number of filters. The computational complexity of a standard convolution will therefore be $D_K \times D_K \times M \times N \times D_F \times D_F$. Because of the factorization that a separable convolution brings, its computational complexity is only $D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F$. In the case of a 3×3 filter, the separable convolution alternative will use between 8 to 9 times less computations.

The components of a separable convolution module that MobileNet implements is illustrated in Figure 2.13. Batch normalization and ReLU both follows after the depthwise convolution layer and the pointwise convolutions layer. The batch normalization also enables the removal of biases of the filters for fewer computations. The full MobileNet architecture is shown in Table 2.2.

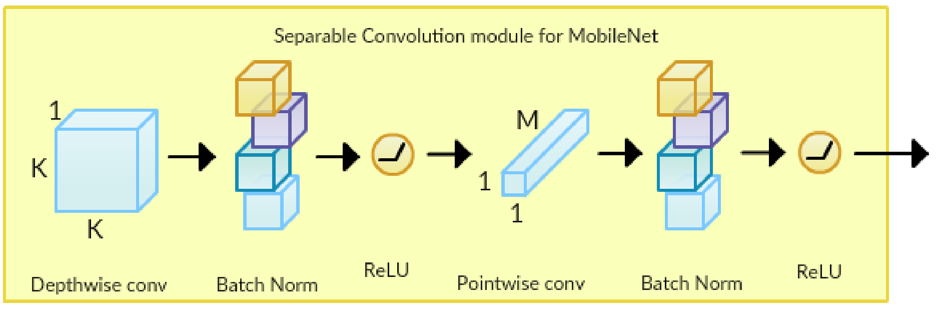


Figure 2.13: The components of a Separable Convolution module in the MobileNet architecture

Type	Filter size	# Filters / Output
Conv	3×3	32
Separable Conv module	3×3	64
Separable Conv module	3×3	128
Separable Conv module	3×3	128
Separable Conv module	3×3	256
Separable Conv module	3×3	256
Separable Conv module $\times 6$	3×3	512
Separable Conv module $\times 2$	3×3	1024
Avg pooling	Pool 7×7	-
FC	-	1024 Neurons
Softmax	-	# Classes

Table 2.2: Overview of the MobileNet architecture. Filter size equals to K and number of filters equals to M for separable convolutions.

According to the original paper, the version *0.50 MobileNet-160* has about the same amount of trainable parameters as SqueezeNet, but has higher accuracy on ImageNet classification (60.2% versus 57.7%) [39].

2.7 Data Augmentation

Data augmentation is one of the easier and most common method for preventing overfitting of a Convolutional Neural Network. The concept of augmentation is to artificially enlarge the data set by performing a collection of label-preserving image transformations and hopefully result in a more generalized neural network when trained on. As a result, a single augmentation of flipping every image horizontally will double the amount of data available. Performing a variable augmentation, for example translate image x pixels, will create even more new samples out of one original image. Performing multiple augmentations in a sequence will have an exponential effect on the augmented data available, but may result in too aggressive augmentation if one is not monitoring the augmented images and adjusting the sequence of augmentations properly.

Some of the most common augmentations include image transformations like flip, warp, rotate, scale and translate, but also image enhancement techniques like contrast normalization and gamma correction.

2.8 Human performance

An important metric when evaluating new models is how they compare to human performance. Until very recently, humans have been way ahead of Machine Learning algorithms for classifying images. The use cases for applying Deep Learning for certain tasks is thus limited by how well it measures up against humans.

Andrej Karpathy, director of AI at Tesla, argues that human performance is not a point in terms of accuracy. Human performance should rather be seen as a trade-off curve [40]. In order to achieve a better accuracy from a human perspective effort and expertise is required. After one week of training on the ImageNet, Andrej Karpathy was able to achieve an error rate of 5.1%. ImageNet consists of 1000 different classes, including 120 species of dogs. [40] estimates that 37% of human errors fall into the category of failing on such fine-grained decisions, whilst for GoogLeNet, only 7% of its errors do.

2.9 Deep Learning Work Flow

This thesis follows a particular method for structuring Machine Learning projects, teached by Andrew Ng, 2017, which is integrated into the work flow of Google Brain and Baidu AI research projects [41]. This structured work flow prevents undercomplicated and overcomplicated results and continuously provides a clear direction on how to make further progress with the project.

A flow chart of the concept of the work flow to use in this thesis project is demonstrated in Figure 2.14. The initial steps are to set up an evaluation metric, and split data into train and test set. Thereafter, the goal is to implement the initial model quickly, without over-complicating things. The initial models define a starting point that will help to confirm or reject our hypotheses about which parts should be prioritized to spend time on. The prioritization is decided by the second step, which is performing bias-variance analysis and error analysis, explained below in the next subsections. The earlier steps are thereafter revisited with the current model as starting point, and an improvement is made to the models according to the results of the analyses of what change can be most beneficial with respect to time required. A bias-variance and (but not necessarily every time) an error analysis is performed on the improved model, which hints about any new directions in how to improve the model further. Hereon, the research and development becomes a continuous loop that iterates between improving the model and analyzing what to improve next.

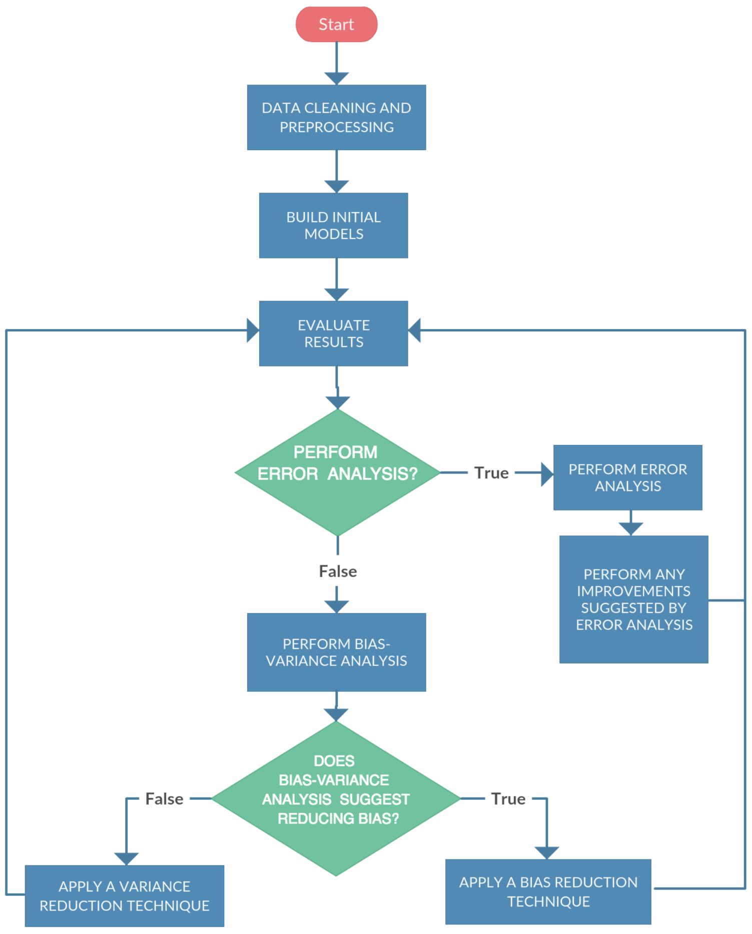

Start
DATA CLEANING AND PREPROCESSING
BUILD INITIAL MODELS
EVALUATE RESULTS
PERFORM ERROR ANALYSIS?
True
PERFORM ERROR ANALYSIS
PERFORM ANY IMPROVEMENTS SUGGESTED BY ERROR ANALYSIS
False
PERFORM BIAS-VARIANCE ANALYSIS
DOES BIAS-VARIANCE ANALYSIS SUGGEST REDUCING BIAS?
False
True
APPLY A VARIANCE REDUCTION TECHNIQUE
APPLY A BIAS REDUCTION TECHNIQUE

Figure 2.14: Flowchart of our workflow to follow throughout the thesis project.

2.9.1 Bias-variance analysis

The purpose of bias-variance analysis is to gain insight into whether a method for reducing the bias or a method for reducing the variance has the most potential for performance gain on the validation set.

The analysis makes use of the Bayes error to pinpoint the theoretical performance limit of a classifier to analyze. In most cases, the Bayes error is unknown and human-level performance is used as a landmark of what is at least possible in terms of performance. When deciding whether the next step is to reduce bias or to reduce variance, the gaps between the human-level performance, training set performance and validation set performance is examined. An example is shown in Figure 2.15 where the gap between training set performance and validation set performance is larger than the gap between human-level performance and training set performance, which indicates that the most potential gain lies in trying to reduce the validation set error by reducing the variance.

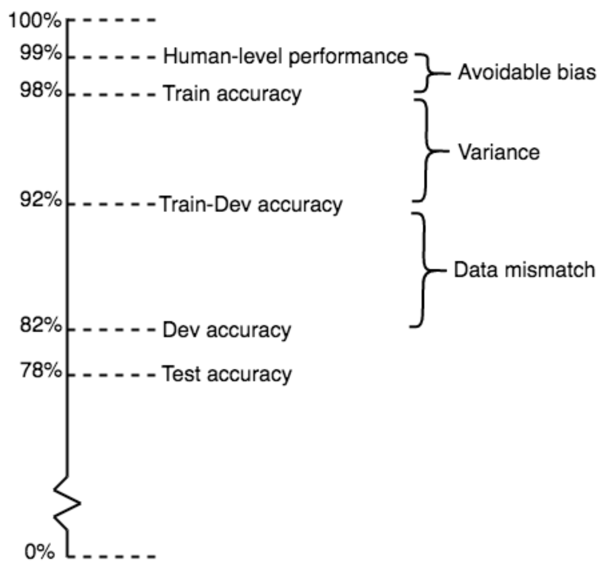


Figure 2.15: An example of how the performance of the different data set partitions can be distributed and what gaps between the performances indicates.

Methods for reducing the bias includes increasing capacity of the model by using a larger network, train longer, try other optimization algorithms, try different network architectures and hyperparameter search. Methods for reducing the variance includes decreasing the capacity of the model, gathering more data and regularization techniques.

2.9.2 Error analysis

Error analysis involves investigating the misclassified samples to gain insight into which classes are hard to classify. This will determine which classes are worth improving the accuracy for to make the highest impact on the overall accuracy. Techniques for improving the accuracy of a particular class includes gathering more data samples and further data augmentation of that class.

When examining the misclassified samples, one can also approximate the percentage of incorrectly labeled data to decide whether its a problem that is worthwhile fixing.

3

Hand Detection

The human mind is extremely accurate at locating and recognizing objects in a scene. An object detection algorithm comparable to human performance would enable computers to perform tasks that right now requires a human's visual recognition system, which is therefore a research area of high importance. The process of object detection is defined as localization and recognition of objects in an image. Namely, first locate the position of an object in the image, and then classify what type of object it is. An image can contain a variable number of objects of different sizes, which makes it a much more complex problem than simple recognition.

We include object detection as a part of the thesis by addressing locating hands that needs to be input to the gesture classifier as an object detection problem. A general object detector implementation is scalable when there is need of recognizing additional objects other than hands. But for the sole purpose of gesture classification by the driver only, a hand detector only able to detect one hand at maximum inside a bounded area near the dashboard is also acceptable. This thesis will implement and evaluate both alternatives

3.1 Previous work

The classical approach towards solving object detection problems involves the Viola-Jones method and using HOG features [22] [42]. Deep learning has significantly overcome these classical approaches and are now the main ingredient in state-of-the-art object detection algorithms.

One of the first breakthroughs of deep learning based object detection algorithms was called R-CNN, published by Ross Girshick et al. in 2013 [43]. The concept was to first use a method for proposing regions inside the image that could contain an object. Every region is thereafter put into a CNN image recognizer to classify whether the proposed region contains any of the object classes available. In the original paper, the region proposal method used was selective search, and have since then been replaced by more efficient region proposal methods, evolving its name to Fast R-CNN and later Faster R-CNN [44] [45]. Even though the name is Faster R-CNN, its only efficient enough to be able to operate in real-time (5 fps) on a high-end Nvidia Tesla K40 GPU. Faster-RCNN introduces a module called Region Proposal Network (RPN) to replace the selective search algorithm, which was the bottleneck in the previous approaches. Using a RPN, the entire model also becomes trainable end-to-end. Some of the most accurate object detectors as of the writing of this thesis are Mask-RCNN [46], Deformable Convolutional Networks [47] and Path Aggregation Networks [48], but they all build upon the theory of Faster R-CNN and thereby can only perform multiple inferences a second using a high-end GPU.

There exists another family of object detection algorithms using a technique called You-Only-Look-Once (YOLO) [49]. The original publication can reach up to 155 fps on a Nvidia TitanX GPU, but not quite exceeding the performance of Faster R-CNN. The improved method however, YOLOv2, still retains its real-time inference speed while exceeding the performance of Faster R-CNN ([46], [47] and [48] have since exceeded YOLOv2). M.J. Shafie et al. have improved inference time of YOLOv2 even further for video sequences by implementing logic that only allows deep inference of the object detector if there are enough pixels in the image that has changed compared to a reference image [50]. B. Wu et al. have also improved the inference time of YOLOv2 by replacing the original feature extractor with a SqueezeNet [51]. Worth mentioning is also the Single Shot MultiBox Detector (SSD), which outperforms the accuracy and inference time of YOLOv1, but not YOLOv2 [52].

T.H.N. Le et al. has tackled the problem of detecting hands within a car multiple times [53] [54] [55]. Inspired by the theory of Faster R-CNN, their different publications reveal a inference speed of 0.06 fps, 0.234 fps and 4.65 fps on a Nvidia TitanX GPU for respective publication and their latest version currently holds state-of-the-art performance for the VIVA Challenge dataset [56].

3.2 You-Only-Look-Once

YOLO, shorthand for You-Only-Look-Once, is an object detection technique published in 2015, that was invented with the purpose of being able to perform real-time inference. YOLO introduces the ability to detect objects in an image using only a single forward pass of the image through the network, which was a shortcoming of previous methods [49].

The network first divides the input image into a grid (13 by 13 cells are used in the paper). Each cell is responsible for predicting a fixed amount of bounding boxes (5 bounding boxes per cell is used in the paper). Along with every bounding box prediction, each box is associated with a confidence score prediction and a class prediction. The confidence score is a value of how certain the network is that the predicted bounding box encloses an object of any kind. An example is shown in Figure 3.1, where panel a) demonstrates the grid and panel b) demonstrates predicted bounding boxes where a thicker box border represents a higher confidence score. The class prediction output of every predicted bounding box is similar to a

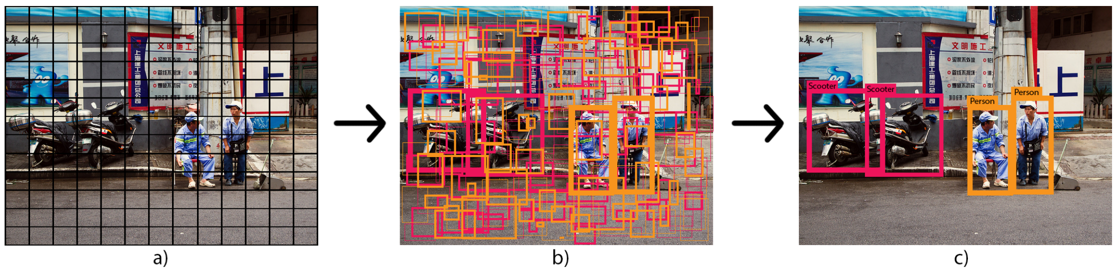


Figure 3.1: Example of how the coordinate labels are calculated using an anchor box and the ground truth coordinates.

regular image recognizer, i.e. outputs a distribution of class probabilities. The class probabilities are combined with the confidence score into a final value that represents the probability of the bounding box containing a specific type of object. If two bounding boxes has a high overlap, the bounding box with the highest confidence score is kept, while the other one is discarded. This procedure is called non-max suppression and it will remove redundant predicted bounding boxes. A last step is to remove any bounding boxes with a confidence score below a certain threshold. Panel c) of Figure 3.1 shows how only the bounding boxes above a certain confidence score are kept using these steps.

Its successor, YOLOv2, adopts the RPN idea from Faster R-CNN. Instead of predicting the actual dimensions of the bounding box, an anchor box is instead used and an offset from that anchor box is predicted. Predicting offsets instead of coordinates simplifies the features to learn and by extension makes the learning process easier. If each cell in the image grid is responsible for predicting 5 bounding boxes, there will be 5 anchor boxes that an offset is predicted from. The width and height of the anchor boxes are predetermined before training. Suitable dimensions of the anchor boxes can be calculated by performing k-means clustering on the set of ground truth bounding boxes before training [52].

When implementing YOLOv2, a general feature extractor is used as a base. J. Redmon et al. uses a custom architecture that they call darknet [52]. Thereafter, the prediction layer is put on top that makes up for the grid previously mentioned. For each grid cell, there are $K \times (4 + 1 + C)$ outputs. K is the number of anchor boxes to use, and each anchor box will have 4 coordinate outputs, one confidence score output and C class probability outputs. If the grid has I columns, J rows and K anchor boxes, then there will be $I \times J \times K$ bounding box coordinate outputs (δx_{ijk},δy_{ijk},δw_{ijk},δh_{ijk}) that are relative to the corresponding anchor box coordinate ($\hat{x}_i$,$\hat{y}_j$,$\hat{w}_k$,$\hat{h}_k$).

How the absolute bounding box prediction coordinates (x_i^p, y_j^p, w_k^p, h_k^p) are predicted from the relative coordinate predictions and the anchor boxes are calculated by:

$$\begin{aligned} x_i^p &= \hat{x}_i + \hat{w}_k \delta x_{ijk}, \\ y_j^p &= \hat{y}_j + \hat{h}_k \delta y_{ijk}, \\ w_k^p &= \hat{w}_k e^{\delta w_{ijk}}, \\ h_k^p &= \hat{h}_k e^{\delta h_{ijk}}. \end{aligned} \tag{3.1}$$

Since object detection is a multitask problem, both localization and recognition, the loss function consists of multiple terms. The loss function is

$$\begin{aligned} &\frac{\lambda_{bbox}}{N_{obj}} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} I_{ijk} \Big[(\delta x_{ijk} - \delta x_{ijk}^G)^2 + (\delta y_{ijk} - \delta y_{ijk}^G)^2 \\ &\qquad + (\delta w_{ijk} - \delta w_{ijk}^G)^2 + (\delta h_{ijk} - \delta h_{ijk}^G)^2 \Big] \\ &+ \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \frac{\lambda_{conf}^+}{N_{obj}} I_{ijk} (\gamma_{ijk} - \gamma_{ijk}^G)^2 + \frac{\lambda_{conf}^-}{WHK - N_{obj}} \bar{I}_{ijk} \gamma_{ijk}^2 \\ &\qquad + \frac{1}{N_{obj}} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \sum_{c=1}^{C} I_{ijk} l_c^G \log(p_c). \end{aligned} \tag{3.2}$$

It consists of mean squared error terms for each relative coordinate output, a mean squared error term for the confidence score, a regularizing term for confidence scores and a cross-entropy term for class predictions. G symbolizes ground truth and the ground truth bounding box $(\delta x_{ijk}^G, \delta y_{ijk}^G, \delta w_{ijk}^G, \delta h_{ijk}^G)$ is calculated by

$$\begin{aligned} \delta x_{ijk}^G &= (x^G - \hat{x}_i)/\hat{w}_k, \\ \delta y_{ijk}^G &= (y^G - \hat{y}_i)/\hat{h}_k, \\ \delta w_{ijk}^G &= \log(w^G/\hat{w}_k), \\ \delta h_{ijk}^G &= \log(h^G/\hat{h}_k), \end{aligned} \tag{3.3}$$

where (x^G, y^G, w^G, h^G) is the ground truth bounding box coordinates. The hyperparameters of the loss function are λ_{bbox}, λ_{conf}^+ and λ_{conf}^-, which determine the strengths of the different loss terms. I_{ijk} is an indicator function, equating to 1 if the kth anchor at position (i,j) has the largest overlap with a ground truth bounding box, otherwise 0. I_{ijk} makes sure to penalize the squared difference between the ground truth confidence score γ_{ijk}^G and the prediction γ_{ijk} only if anchor box k of grid cell (i,j) has a ground truth bounding box assigned to it. $\bar{I}_{ijk}$ is its counterpart ($\bar{I}_{ijk} = 1 - I_{ijk}$) and it makes sure to penalize the confidences of bounding box predictions that does not have any ground truth bounding box assigned to it with a squared error of the confidence score. The class prediction cross-entropy loss term is also masked by I_{ijk} because there should be no class prediction loss for bounding box predictions that does not have a ground truth object assigned to it.

3.3 Implementation

The detection of a hand as pre-processing step before gesture classification can be either very flexible by using an object detector to detect hands at any location inside the vehicle, or can be very restricted by limiting the use to one gesture at a time, only performed at the area in front of the infotainment system. We consider both alternatives, implementing both a deep learning object detector and a simple fixed window hand detector.

3.3.1 Fixed window hand detector

We decided to first implement a simple solution that would only be able to recognize gestures of the driver when the hand is located close to the infotainment system. The complete algorithm for implementing our approach is shown in Algorithm 2. First, the area of the input that captured the space in front of the infotainment system was cropped out (line 1). Our theory was that a good enough prediction of extracting the hand from this cropped area was to extract the largest moving object. Since the hand needed to be moved into the space near the infotainment system before performing the gesture, we assumed that the hand would be the largest moving object in this area. Since only moving objects would be considered, an exponential moving average of the cropped images was kept in memory (lines 3-5 and line 17). To extract moving pixels from the current image, the absolute difference of the moving average image and the current image was calculated (line 6). To only filter out the more significant movements of the image, a binarization threshold was applied (lines 7-8). Thereafter, two hand crafted convolutional filters were convolved with the binary image of moving pixels to extract vertical and horizontal edges. The resulting image of vertical edges was used to create a histogram along the horizontal axis of the image, and the same was done to create a histogram of horizontal edges along the vertical axis of the image (lines 9-12). Lastly, the bounding box coordinates are calculated from respective histogram. The starting point of an axis is set at the pth percentile and the end point is set at the $(1-p)$th percentile, i.e. the bounding box captures $100(1-2p)$ percent of the moving edge pixels at any axis. Lines 11-16 can better be understood by illustrating the resulting histograms from the edge maps, shown in in Figure 3.3 in the Results section.

```
input : $I$ as the input image from camera, $\theta$ as the binarization threshold, $\gamma$ as
        the moving average period, $p$ as the percentile cutoff
output: $(x_1, y_1, x_2, y_2)$ as a predicted bounding box
$I_c \leftarrow cropImage(I)$;
if $I_{ma} == null$ then
    $I_{ma} \leftarrow I_c$;
    return $null$;
end
$I_{diff} \leftarrow |I_{ma} - I_c|$;
$I_{bin} \leftarrow I_{diff}[I_{diff} <= \theta] = 0$;
$I_{bin} \leftarrow I_{bin}[I_{bin} > \theta] = 255$;
$f_h \leftarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 \end{bmatrix}$;
$f_v \leftarrow \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}$;
$hist_h \leftarrow sum(I_{diff} \circledast f_h)_{axis=vertical}$;
$hist_v \leftarrow sum(I_{diff} \circledast f_v)_{axis=horizontal}$;
$x_1 \leftarrow getIndexAtPercentile(hist_h, p)$;
$x_2 \leftarrow getIndexAtPercentile(hist_h, 1 - p)$;
$y_1 \leftarrow getIndexAtPercentile(hist_v, p)$;
$y_2 \leftarrow getIndexAtPercentile(hist_v, 1 - p)$;
$I_{ma} \leftarrow \gamma I_{ma} + (1 - \gamma) I_c$;
return $(x_1, y_1, x_2, y_2)$
```

Algorithm 2: The fixed window hand detector algorithm. The input is an image from the camera, and the output is a bounding box prediction.

3.3.2 Deep learning hand detector

After reviewing the inference time and hardware requirements of the techniques discussed in Section 3.1, we concluded that SSD and YOLOv2 was the only techniques efficient enough for our purpose. Of the two alternatives, we chose YOLOv2 because of its superior performance. The original architecture uses Darknet-19 as feature extractor, which is similar to VGG-16 [52], but we decided to implement YOLOv2 using a more efficient feature extractor. We adopted the idea of SqueezeDet [51], implementing YOLOv2 with a stub of SqueezeNet as feature extractor. We also implemented an alternative architecture using a stub of MobileNet as feature extractor.

The SqueezeNet architecture can be seen in Table 3.1 and the MobileNet architecture in Table 3.2. The label pre-processing was adopted from Equation 3.3 and the loss function was constructed according to Equation 3.2. Originally, there are nine anchor boxes for each grid cell, but since there is a low chance that many hands are cluttered together for the application of hand detection in cars, we decided to set the number of anchor boxes to 5. As for the anchor box dimensions, they were calculated by performing k-means clustering on the bounding box dimensions of the labels of the dataset. k was initialized to 5, which was the number of anchor boxes to use, and the clusters were fit on the labels using bounding box width and height as features.

Type	Filter size / Stride	# Filters
Conv(Frozen) 1	3×3 /2	64
Max Pooling 1	3×3 /2	-
Fire module 2	-	$s_{1\times1} = 16$, $e_{1\times1} = 64$, $e_{3\times3} = 64$
Fire module 3	-	$s_{1\times1} = 16$, $e_{1\times1} = 64$, $e_{3\times3} = 64$
Max Pooling 3	3×3 /2	-
Fire module 4	-	$s_{1\times1} = 32$, $e_{1\times1} = 128$, $e_{3\times3} = 128$
Fire module 5	-	$s_{1\times1} = 32$, $e_{1\times1} = 128$, $e_{3\times3} = 128$
Max Pooling 5	3×3 /2	-
Fire module 6	-	$s_{1\times1} = 48$, $e_{1\times1} = 192$, $e_{3\times3} = 192$
Fire module 7	-	$s_{1\times1} = 48$, $e_{1\times1} = 192$, $e_{3\times3} = 192$
Fire module 8	-	$s_{1\times1} = 64$, $e_{1\times1} = 256$, $e_{3\times3} = 256$
Fire module 9	-	$s_{1\times1} = 64$, $e_{1\times1} = 256$, $e_{3\times3} = 256$
Fire module 10	-	$s_{1\times1} = 96$, $e_{1\times1} = 384$, $e_{3\times3} = 384$
Fire module 11	-	$s_{1\times1} = 96$, $e_{1\times1} = 384$, $e_{3\times3} = 384$
Dropout layer 11	-	-
Conv	3×3 /1	anchors per grid *(classes+1+4)
Softmax	-	# Classes

Table 3.1: Overview of the SqueezeNet architecture. Global average pooling averages over every input channel.

Type	Filter size	# Filters / Output
Conv	3×3	32
Separable Conv module	3×3	64
Separable Conv module	3×3	128
Separable Conv module	3×3	128
Separable Conv module	3×3	256
Separable Conv module	3×3	256
Separable Conv module $\times 5$	3×3	512
Separable Conv module	3×3	1024
Avg pooling	2×2	-
Conv	3×3	anchors per grid *(classes+1+4)
Softmax	-	# Classes

Table 3.2: Overview of the MobileNet architecture. A Separable Conv module follows the architecture presented in Figure 2.13.

The datasets used for experiments were VIVA Challenge dataset [56] and EgoHands [57], and data simple augmentation techniques were performed online while training.

3.4 Results

A visualization of how the fixed window hand detector creates its bounding box prediction is shown in Figure 3.2 and Figure 3.3. The image of exponentially moving averaged vertical edges is shown in panel a) of Figure 3.2, the image of exponentially moving averaged horizontal edges is shown in panel b) of Figure 3.2 and the original image is shown in Figure 3.3. The resulting histogram for each edge image is also shown. Figure 3.3 also shows how the bounding box coordinates are calculated by aligning the histograms and calculating the specified percentiles (shown in red).

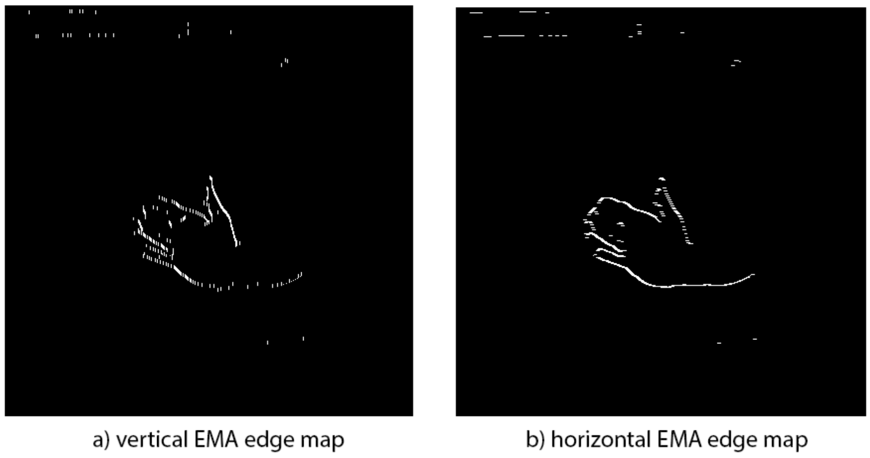


Figure 3.2: Panel a) shows the result of $I_{diff} \circledast f_v$ from line 12 of Algorithm 2, which is a feature map of exponentially moving averaged vertical edges. Panel b) shows the result of $I_{diff} \circledast haar_h$ from line 11 of Algorithm 2, which is a feature map of exponentially moving averaged horizontal edges.

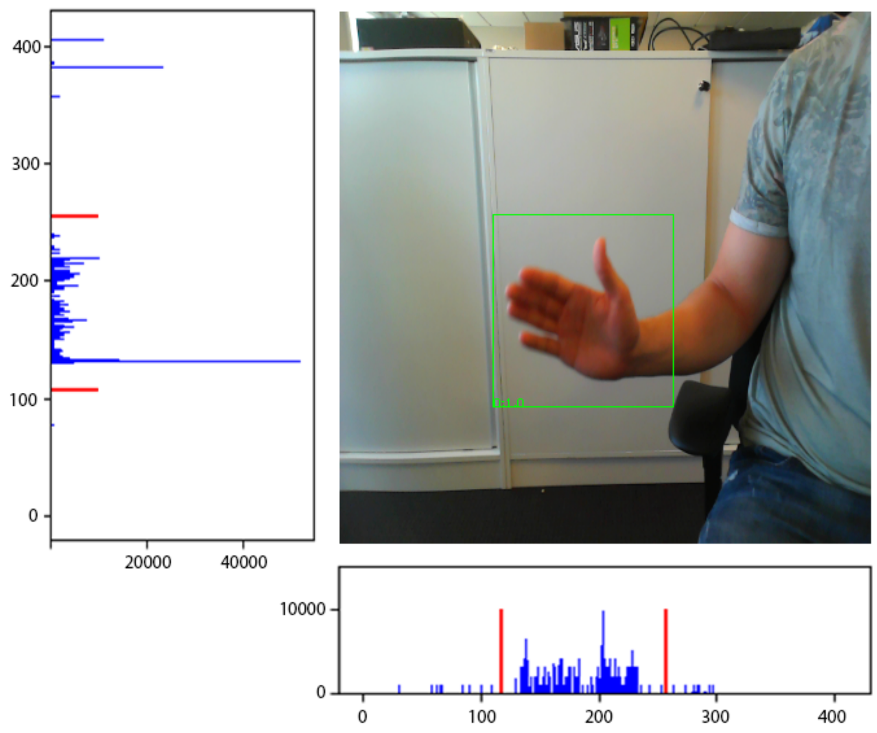


Figure 3.3: Visualization of how Algorithm 2 predicts a bounding box. The percentile cutoffs of each histogram is shown in red, and its corresponding bounding box is shown in green.

Figure 3.4 shows the result of performing k-means ($k = 5$) clustering on the bounding boxes of the VIVA challenge dataset. The dimensions of these clusters are saved to disk and thereafter used as priors (anchor boxes) for the YOLOv2 algorithm.

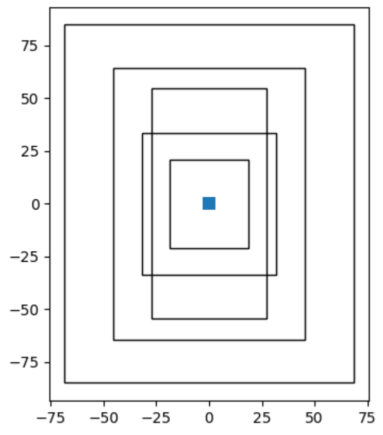


Figure 3.4: The resulting anchor box dimensions calculated using k-means clustering on VIVA Challenge dataset

The performance over time during training of the object detector using different feature extractors on the EgoHands dataset is shown in Figure 3.5. Comparing the train and test accuracy reveals that both architectures manage to generalize well on the test data.

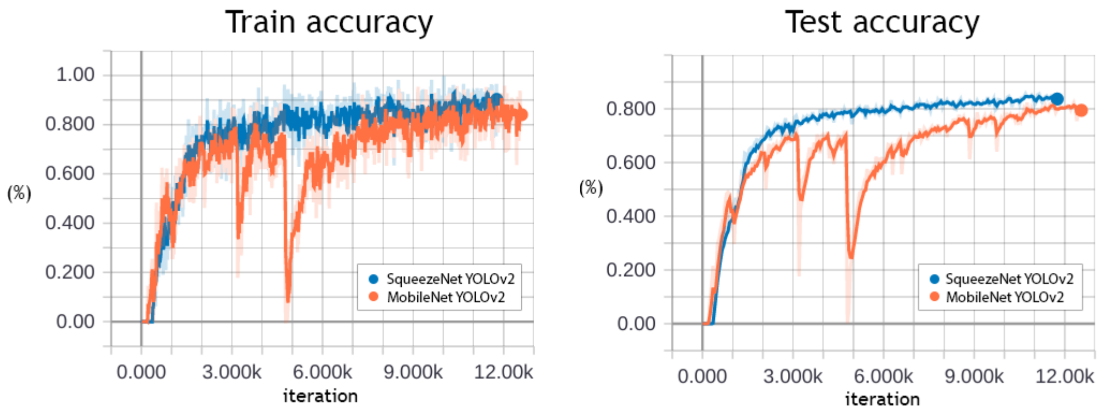


Figure 3.5: Train set and test set performance during training of object detector with different feature extractors as base. Orange represents MobileNet as feature extractor (82mAP) and Blue represents SqueezeNet as feature extractor (85mAP).

Table 3.3 shows the final performance and inference time of respective architecture. The model using a stub of SqueezeNet is superior in this particular case, both in terms of performance and inference time.

Method	mAP	FPS
YOLOv2 + SqueezeNet	85	16.7
YOLOv2 + MobileNet	82	12.8

Table 3.3: Comparison of performance and inference time for our different object detector architectures.

Table 3.4 summarizes the performance and inference time of different methods for the VIVA Challenge dataset. Our method is superior to many of the VIVA Challenge contestants in terms of mAP, but a significant performance gap remains when comparing to the state-of-the-art hand detector MS-RFCN. Nevertheless, the inference time of our model is by far the best when considering our inference time is measured on a Jetson TX2 embedded platform.

Methods	mAP	FPS
MS-RFCN [55]	86.9	4.65 (TitanX)
MS-FRCNN [54]	77.6	0.234 (TitanX)
YOLOv1 [49]	69.5	35 (TitanX)
ACF_Depth4 [58]	60.1	-
CNN with Spatial Region Sampling [59]	57.8	0.78 (Tesla K20)
Ours (YOLOv2 + SqueezeNet)	78.9	16.7 (Jetson TX2)

Table 3.4: Comparison of our hand detector to other methods in terms of performance (mAP) and inference time (FPS).

3.5 Discussion

The simple hand crafted solution worked amazingly well for creating a bounding box around the largest moving object within the specified area. The drawback is obvious though, any eventual gestures are restricted to the area in front of the infotainment system and any large object other than a hand will result in a false positive detection. If the hand gesture classifier to use upon the detections are trained with any category resembling "no gesture", false positives from the object detector will be acceptable.

DepthSense CARlib by Sony is a car infotainment gesture control system currently deployed in a few BMW car models, and it also uses the restriction of only being able to perform the gestures in front of the infotainment system because their depth camera only monitors that area. Smart Eye on the other hand, use their camera to monitor a much bigger area, and therefore this additional cropping or detection step is necessary.

Additional to the simple hand crafted solution, we also implemented a deep learning solution. The argument for also implementing a deep learning solution was that it is scalable with respect to expanding the categories of objects to track, and would result in a more general framework to build upon for Smart Eye. The inference times of our two YOLO architectures are extremely fast compared to the R-CNN family of implementations, see Table 3.3. This comes with the drawback of worse performance, but when the inference time is critical, this performance loss can be acceptable and our object detector implementations are suitable for the task. With the high train mAP of almost 100 in Figure 3.5, we believe that there are plenty of capacity left for adding additional object classes to track, but time constraints and lack of labels has left this topic for future work.

Regarding differences between the SqueezeNet YOLOv2 implementation and the MobileNet YOLOv2 implementation, the SqueezeNet variant was superior, see Table 3.3. Even though the SqueezeNet variant also has a faster inference time, MobileNet still can't be excluded. It seems that the Tensorflow implementation of depthwise convolution is slower than normal convolution [60] and therefore it is misleading to compare its inference time in Tensorflow with other methods. Also, the Applied Solutions department of Smart Eye states that a separable convolution implementation in C++ can be much faster. With respect to the performance on the EgoHands dataset and the inference times of our Tensorflow Python implementation, SqueezeNet with YOLOv2 is superior, but the same may not be the case for other datasets and implementations in other frameworks. Whether we decide to implement our object detector with a SqueezeNet stub or MobileNet stub, one thing is for certain, that there will be even better substitutes in the near future. During the time of this thesis, multiple substitutes for efficient feature extraction has already been published, such as MobileNet V2 [61] and SqueezeNext [62].

4
Hand Gesture Classification

Human hand gestures can be divided into two categories: dynamic or static [63]. To recognize a static hand gesture from recorded sensor data you only need to consider spatial features such as the posture of the hand. In addition, dynamic hand gestures also involves tracking the hand motion over time, and therefore temporal features need to be extracted as well.

This thesis will only address solving hand gesture classification from vision based sensors with the explicit purpose to communicate directly with a computer, i.e. the infotainment inside a car. With inspiration from how humans interact with each other, the use of hand gestures have become one of the most applicable ways for Human Computer Interaction (HCI) [64]. The use of vision based sensors is desirable because it requires no physical contact with the user, which is especially beneficial for a motorist whom can remain focused on driving.

In contrast to the entire body, a hand has many degrees of freedom, which makes it a complex task to solve with a RGB camera alone. In addition, classifying hand gestures within environments with changing lightning conditions, such as the inside of a car, further increases the complexity of the classification task [65].

4.1 Previous work

The classical approach for recognizing dynamic hand gestures is to use Hidden Markov Models (HMM). With the rise of deep learning in general and Convolutional Neural Networks in particular, there have been a transition phase with hybrid solutions with HMMs and Deep Learning methods used in combination. However, this thesis will only consider the most recent state-of-the-art techniques, which are purely Deep Learning based.

In 2015, Tran et al. introduced the 3D ConvNet which showed to be a good fit for learning spatiotemporal features from video sequences [66]. The 3D ConvNet architecture was quickly recognized and applied for video classification in general [67], as well as on classifying hand gestures. On *ChaLearn*, a dynamic hand gesture dataset, (which contains color images and depth data) the 3D ConvNet showed state-of-the-art performance when applied by the FLiXT team in 2016 [68].

Time dependencies that are learned from applying 3D convolutions only go as far as the size of the receptive field, the size of the convolutional filters in the depth dimension. As a result, longer sequences would require a severe amount of layers with the use of 3D convolutions alone. Recent publications tackle this deficiency in different ways. Bolei Zhou et al. introduces the Temporal Relation Network (TRN), which makes a fusion of frames at a multiple of time scales from a given sequence [69].

Another approach for classifying longer video sequences is to add one or several LSTM layers on top of the 3D convolutions. However, an issue with fully connected LSTMs (the conventional version) is that it removes the spatial structure from the feature maps because the hidden state is a one-dimensional vector. Xingjian Shi et al. extended the LSTM with convolutional operations to solve this deficiency, introducing the Convolutional LSTM [70] described in Section 4.2. In addition, to further increase the performance from Convolutional LSTM, work such as [71] by Guangming Zhu et al. also employs Spatial Pyramid Pooling on the output. Spatial Pyramid Pooling performs pooling operations of different filter size, which are then concatenated together and passed on to the next layer [72].

The most well-known hand gesture implementation is Sony's *Depth Sense CARlib* which have been successfully integrated into BMW's Gesture Control for their luxury 7 series [11]. With a depth camera placed above the gear stick the Gesture Control supports five dynamic hand gestures which are all used to perform actions with the infotainment inside a car. It support one time actions such as *Swipe Right* and *Point*. Furthermore the system also support continuous actions with the infotainment such as raising the volume through circling a finger.

4.2 Convolutional LSTM

Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting by Xingjian Shi et al. extended the traditional FC LSTM into a ConvLSTM, to keep the spatial relations, similar to a CNN [70]. They use it for the purpose of precipitation nowcasting where it consequently outperforms fully connected LSTMs as well as the previous state-of-the-art *ROVER* algorithm. Work such as [71] by Guangming Zhu et al. have applied the ConvLSTM architecture on hand gesture classification and perform 98.89% validation accuracy on the multi-modal SKIG dataset, achieving state-of-the-art performance.

Xinjian Shi et al. claim that the original fully connected LSTM is good for handling temporal correlation, but is heavily redundant for dealing with spatial data [70]. Traditionally, when dealing with sequential vision-based data the output from the convolutional layers are flattened and passed on to an LSTM as a one dimensional vector and thereby "losing" the spatial information. In contrast, the convolutional

LSTM uses convolutional structures in both input-to-state and state-to-state transitions. The gate equations introduced in Section 2.4.1 are extended with convolutional operations resulting in the following ConvLSTM equations [70]:

$$\begin{aligned} i_t &= \sigma(\mathbf{W}_{xi} \circledast \mathbf{x}_t + \mathbf{W}_{hi} \circledast \mathbf{h}_{t-1} + \mathbf{W}_{ci} \circ \mathbf{C}_{t-1} + b_i), \\ f_t &= \sigma(\mathbf{W}_{xf} \circledast \mathbf{x}_t + \mathbf{W}_{hf} \circledast \mathbf{h}_{t-1} + \mathbf{W}_{cf} \circ \mathbf{C}_{t-1} + b_f), \\ \mathbf{C}_t &= f_t \circ \mathbf{C}_{t-1} + i_t \circ \tanh(\mathbf{W}_{xc} \circledast \mathbf{x}_t + \mathbf{W}_{hc} \circledast \mathbf{h}_{t-1} + b_c), \\ o_t &= \sigma(\mathbf{W}_{xo} \circledast x_t + \mathbf{W}_{ho} \circledast \mathbf{h}_{t-1} + \mathbf{W}_{co} \circ \mathbf{C}_t + b_o), \\ \mathbf{h}_t &= o_t \tanh(\mathbf{C}_t). \end{aligned} \tag{4.1}$$

In Figure 4.1 we can see a simplified visualization of the equations. A 2×2 filter is applied on the previous cell state and hidden state as well as the new input, which are then added together, creating the new output and cell state. Instead of doing vector multiplication we stride over a 3D tensor, $M \times N \times P$, where N and M could represent the input image, and P the number of feature maps. A larger filter implies a larger receptive field and will thus also be able to capture faster motions, while a smaller filter will be able to capture slower motions.

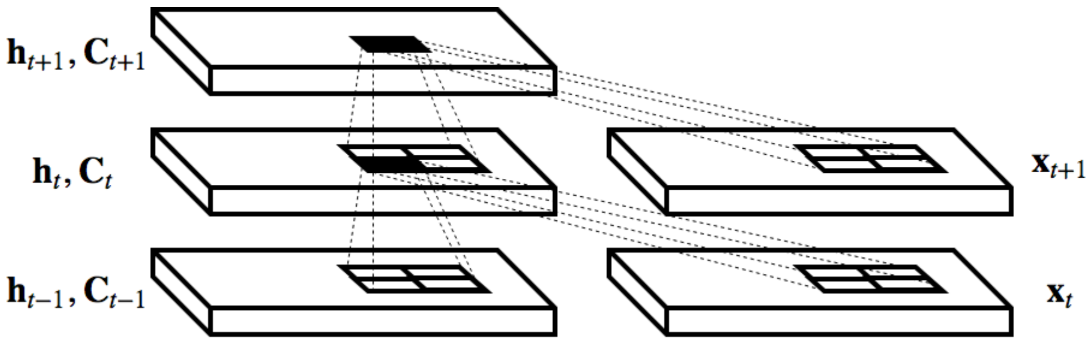

$\mathbf{h}_{t+1}, \mathbf{C}_{t+1}$
$\mathbf{h}_t, \mathbf{C}_t$
$\mathbf{h}_{t-1}, \mathbf{C}_{t-1}$
$\mathbf{x}_{t+1}$
$\mathbf{x}_t$

Figure 4.1: An example of how the performance of the different data set partitions can be distributed and what gaps between the performances indicates.

Furthermore Xingjian Shi et al. concludes that a ConvLSTM may very well be applied after a number of vanilla convolutional layers instead of on the raw image. This was quickly recognized by work such as [71], which even go so far as applying 3D convolutions before. Thereby the 3D convolution performs well on extracting short term dependencies, while as the ConvLSTM will cover the long term.

4.3 Implementation

The goal was to implement a hand gesture classifier working in an embedded environment with the purpose to reduce distractions for motorists. The use of dynamic gestures was desirable because it opens up for making incremental changes to the car's infotainment. Interviews with stakeholders at Smart Eye directed this thesis into not considering depth, as it not commonly used as a part of their core product segment.

Performed hand gestures will be directly mapped to perform actions with the car's infotainment. False positives would result in annoyance for the passengers and counteract the purpose of what was to be implemented. Because of that, a low false-positive rate as well as a high accuracy is highly beneficial. In addition, response times over 100ms have also been showed to be annoying [73], which creates a window where a solution is viable to work within a car. Initial experiments with dynamic hand gestures forced the final solution to be divided into two different models; a prototype of dynamic hand gesture classification that requires additional hardware and a static hand gesture solution working in real time within an embedded environment.

4.3.1 Dynamic hand gestures

Smaller models for classifying dynamic hand gestures were quickly discarded as they failed to predict gestures better than random. We decided to create a larger model which will work as a prototype to demonstrate what is achievable with more expensive hardware. The implemented model however will not apply deep, state-of-the-art, solutions that is far from applicable in real-time scenario even with non-embedded hardware.

An almost unanimous part of state-of-the-art solutions for dynamic hand gestures is the use of the 3D convolutions mentioned in Section 4.1 by Tran et al. [66]. They excel especially on learning spatiotemporal features throughout a sequence. However, the time dependencies that are learned only go as far as the size of the receptive field, the size of the convolutional filters in the time dimension. To solve this issue we choose to add the 3D Convolutional LSTM layers introduced by [70] in general and the implementation of [71] in particular at the end of our network. The architecture of [71] also consists of spatial pyramid pooling after the 3D convolutional LSTM layers as explained in Section 4.1 to further increase the performance of the model.

The final architecture can be seen in Figure 4.2. It can be divided into three separate components which are used to extract features from the video sequence; a 3DCNN feature extractor, two stacked convolutional LSTM's and Spatial Pyramid Pooling. Due to the static nature of 3DCNN's, each input sequence is down- or upsampled to 30 frames before being feed as input to the network.

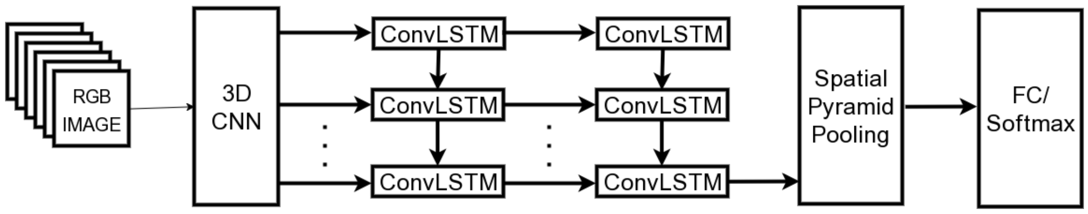


Figure 4.2: Architecture overview of the dynamic hand gesture classifier. A video sequence is down-sampled to 30 frames and fed into a 3DCNN feature extractor. The spatiotemporal features are then passed into two 3DConvLSTM layers, applied with a spatial pyramid pooling and finally a fully connected layer for classification.

Dataset The dataset of choice was going to be *THE 20BN-JESTER DATASET V1*, or simply *Jester*, the world's largest RGB based dynamic hand gesture dataset [74]. The dataset contains 27 different gestures performed by crowd workers in front of a web camera in different environments. The dataset exists of 148,092 videos which are split among the training set, validation set and test set with 118562, 14787 and 14743 videos respectively. The test set is as of this thesis unlabeled due to the ongoing contest. Noteworthy gestures within the dataset is *No gesture* and *Doing other things*, which are both a necessity in real applications to avoid false positives.

In order to reduce the likelihood of overfitting extensive augmentation was done. During offline augmentation each gesture is flipped horizontally, doubling the dataset. Gestures such as *Drumming fingers* remained the same class, while as *Swipe Left* changed label to *Swipe Right*. Additionally, classes such as *Stop Sign* and *Thumb Up* was also played backwards. Finally, some gestures were both horizontally flipped and mirrored in the time domain. For example *Swipe Right*, was double negated and thus remained the same class. Online augmentation was done with the following operations: random cropping, padding, add and multiplication on input channels, dropout, rotation and Gaussian noise.

3DCNN feature extractor
The first part of the final model is a 3D Convolutional Neural Network, which is used to extract spatiotemporal features. The input to the 3DCNN is 30 frames from a given dynamic hand gesture reshaped to $176 \times 100 \times 3$ size. The output from the 3DCNN feature extractor has been down-sampled to 8 frames in the time dimension. The final implementation consists of three consecutive convolutional and pooling layers on top of each other, and an additional convolutional layer at the end, as seen in Figure 4.3. The output is then passed on to the LSTM component to learn the long term dependencies.

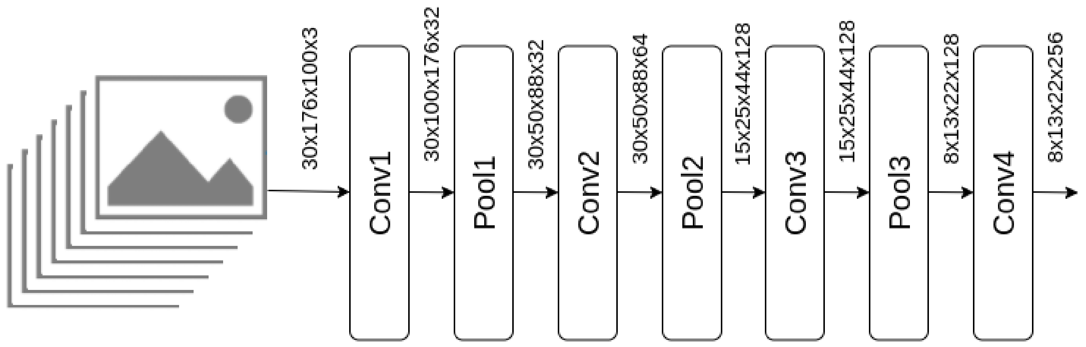


Figure 4.3: The 3D Convolutional Neural Network architecture for dynamic hand gestures.

3DConvolutional LSTM
Through the convolutional and pooling layers from the 3DCNN feature extractor, the initial input is reduced to 8 × 13 × 22 × 256. Each of the output channels from the previous stage were sent into separate ConvLSTM cells, together forming a 3DConvLSTM cell. As aforementioned in Section 4.2, convolutional LSTMs keep the spatial structure in input-to-state as well as state-to-state transitions [71]. "SAME"-padding is used, which implies that the same number of time frames will be passed on to the second 3D convolutional LSTM layer. The first 3DConvLSTM layer uses 256 filters, passing on 256 feature maps to each respective cell in the next layer.

As seen in Figure 4.2, only the last ConvLSTM cell in the second layer passes on an output, i.e. the time frames have been reduced to a size of 1. This means that all temporal features have been extracted. 384 filters are used for the second layer, passing on a 384 two dimensional feature maps.

Spatial Temporal Pooling
From the input spatial size of 176 × 100 to the output of the ConvLSTM layers the spatial size have been reduced to 13×22. By the very nature of fully connected layers, they often lead to a huge amount of parameters. To reduce the spatial size further before passing it on to the FC layer, spatial temporal pooling was performed.

Four type of pooling operations was performed with filter sizes of 28 × 28, 14 × 14 7 × 7, and 4 × 4 respectively. The results was concatenated together, flattened and then fully connected to the last layer for predicting hand gesture class.

4.3.2 Static hand gestures

For static hand gesture classification, we gathered ideas and inspiration from general deep neural network image classification architectures. However, for static hand gesture classification, datasets were limited, small in size and outdated.

Without aid from specific deep neural network solutions within static hand gesture classification the architecture of choice was going to be SqueezeNet, introduced in Section 2.1, specifically the first five layers are used in our implementation [30]. It has an appropriate trade off between accuracy and inference time suitable for embedded implementations.

Dataset. This thesis started of by trying out the Marcel dataset [75] with the following sign language based gestures; A, B, C, Five, Point and Peace. The dataset was small and quickly lead to overfitting for a relatively deep architecture like SqueezeNet. Instead, a custom dataset was created with the same gestures in mind, which can be seen in Figure 4.4. In addition the *No gesture* class was added, to avoid false positives which is a necessity in real scenarios. In total the train set consists of over 18000 images from two persons with $\sim$ 3000 images from each respective gesture. The test set contains 3000 images distributed among classes from a yet unseen person by the network.

Due to the still relatively small dataset, the training procedure lead to overfitting. According to the bias-variance analysis described in Section 2.9.1, a direction to reduce the likelihood of overfitting, and by extension the variance problem is increasing the data size. As a result, the custom dataset was extended with offline and online augmentation. Offline augmentation involved simply flipping all images and saving them to disk, which doubled the dataset. In addition, online augmentation was performed in RAM during training with the following operations; random cropping, padding, add and multiplication on input channels, dropout, rotation and Gaussian noise.

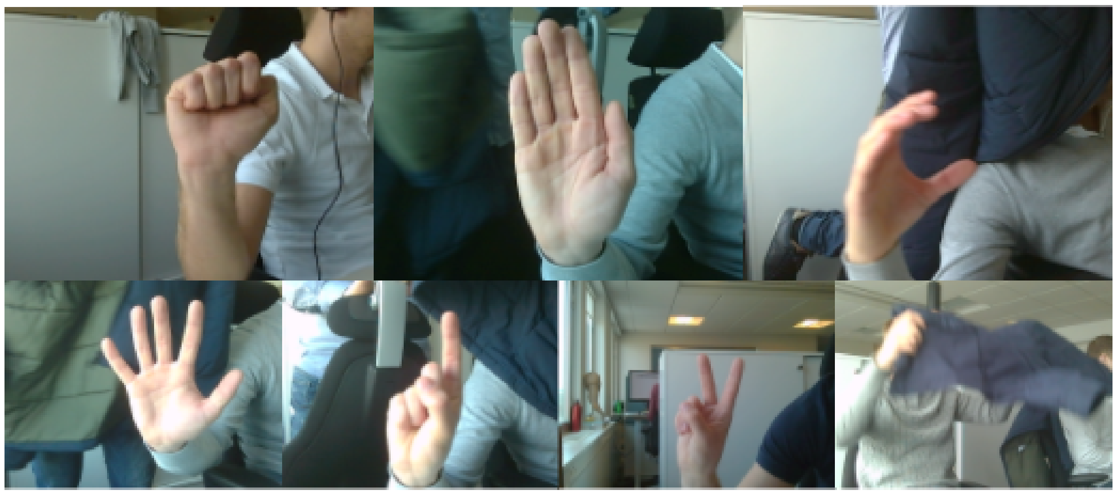


Figure 4.4: The seven gestures available in our custom dataset. In addition to the ones that previously existed we have added the *No gesture*- gesture.

4.4 Result

4.4.1 Dynamic hand gesture classifier

In Table 4.1 the result of our dynamic hand gesture classifier can be seen in comparison to other submitted result in terms of accuracy. Notably among the top results is the Bidirectional 3DConvLSTM, an extension of our proposed solution, however with an increased forward-propagation time.

Model	Top 1 acc(%)
DRX3D	96.6%
Ford's Gesture Recognition System	94.11%
Our Model	**89.97%**
Twenty Billion Neuron's Jester System	82.35%
Our 3DCNN Model	73.00%

Table 4.1: Our implementation in comparison to other submissions.

The normalized confusion matrix on the validation set can be seen in Figure 4.5. The last 1000 sequences is used, with the model that is trained with all augmentation techniques. *Drumming Fingers*, *No gesture* and *Pushing Hand Away* are all correctly classified correctly above 97% of the time. The worst classes are *Turning Hand Clockwise*, *Turning Hand Counterclockwise* and *Zooming Out With Full hand*, which are correctly classified 42%, 63% and 56% of the times respectively.

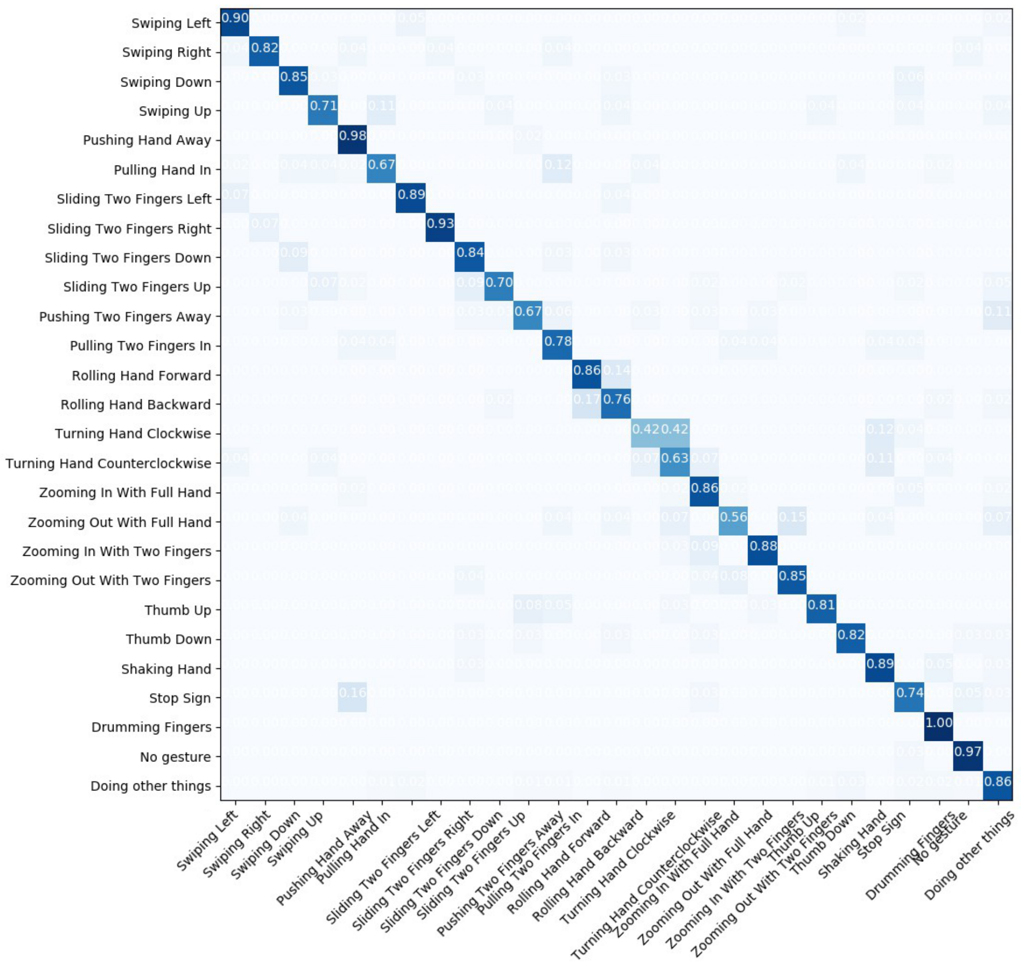


Figure 4.5: Normalized confusion matrix on the Jester validation set. The model used for predictions was with all augmentation techniques.

4.4.2 Static hand gesture classifier

In Figure 4.6 the accuracy during the training procedure can be seen. After 2000 iterations the train accuracy started to consequently perform at 100%. After 6000 iterations the test accuracy converged at 98% without augmentation. An iteration represents one batch of size 100. The final architecture runs at 48 fps on a Jetson TX2 platform.

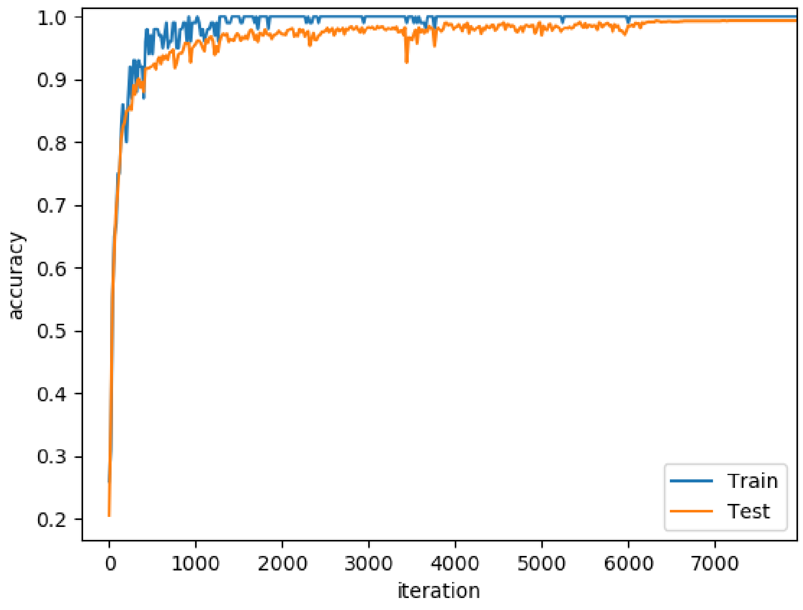


Figure 4.6: Train and test accuracy over time during the training procedure. After 2000 iteration the training accuracy consequently performed 100% and after 6000 iterations the test accuracy converged at 98%.

In Figure 4.7 we can see the normalized confusion matrix when making predictions on the test set. Apart from the *Point*-gesture, which is classified as the *Peace*-gesture six percent of the times, the actual gestures all perform well. The *No gesture* class are correctly classified 81% of the times, often for the benefit of the *A* or *B* gestures.

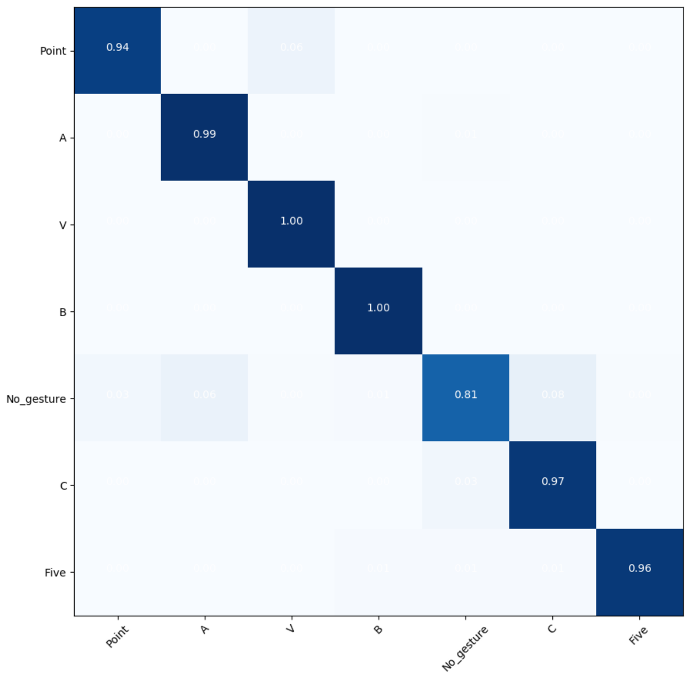


Figure 4.7: Normalized confusion matrix for the custom static hand gesture dataset.

4.5 Discussion

The goal was to implement a hand gesture classifier working in a real time environment. The use of dynamic gestures was desirable because it opens up for making continuous changes to the car's infotainment. The final result for dynamic gestures required a huge network design in order to provide a good-enough accuracy on the large Jester dataset. Because of this, it is not suitable for an embedded real-time scenario. However, for a solution that is purely built for working within an automobile environment, with a static background we believe the complexity of the classification would be reduced tremendously. In addition a specific camera location opens up for solutions in which the gesture can be performed within a hard-coded area of the image, such as above the gear stick. Nevertheless, for a general dataset as Jester, where gestures can be performed at all areas in the picture and with several different backgrounds, we were not able to scale down our architecture. Smaller solutions often failed to learn anything at all when including all classes. As a result, our solution for dynamic gesture classification fail to run on the Jetson TX2 due to memory issues. The final presented model performs 89.97% on the large Jester dataset, which is a bit off from state-of-the-art solutions at 96% but is reasonable because of the still relatively small architecture.

The final dynamic hand gesture implementation failed to operate in real time in an embedded environment. If dynamic hand gestures are very desirable to use, we believe an almost necessary step is going towards the use of depth cameras. It allows to remove all background noise, severely reducing the complexity of the classification task. Another direction could be too use connectionist temporal classification loss, which requires a more sophisticated labeling of sequences. Currently, on the Jester dataset a dynamic hand gesture includes frames that are not necessarily a part of the gesture, such as the pre-gesture step when the capture just have been started. If you label these situations as *No gesture*, sequences can be divided into smaller chunks and fed one at a time through a much smaller 3D CNN architecture.

As a result of the large, dynamic model, a static hand gesture model was implemented on a Jetson TX2. The model is using the first five layers of the SqueezeNet architecture together with a FC layer for classification. Static hand gestures do not rely on the motion in time, and therefore our implemented hand detector was used as a pre-processing step in order to remove unnecessary information from the complete image in our real time implementation. In addition this made it possible to classify a multiple of hand gestures in parallel from multiple agents.

The final model performs 100% and 98% for the training set and test set respectively on our custom dataset. The dataset is still rather small, which made some degree of overfitting inescapable. The small dataset in combination with a rather simple underlying problem, similar to digit recognition, made deeper architectures redundant. In addition, even shallower architectures could be examined in the future. However, the final static hand gesture model runs at 48 fps on a Jetson TX2, which is more than enough to make it applicable in real scenario.

For live implementations a reduced amount of gestures might be a good choice. By examining the confusion matrix, similarities between gestures can be seen. Such gestures are hard to separate and often lead to a high number of false positives, to the benefit of the related gesture. For example, the dynamic gestures of the Jester dataset *Rolling Hand Backward* and *Rolling Hand Forward* are naturally similar and just using one of them live could be beneficial. For static hand gestures, we can by investigating the confusion matrix (Figure 4.7), see that the *No gesture* class is often mis-classified as another hand gesture. This is most likely because the *No gesture* class contains random backgrounds as well as hand gestures which are not included as an actual gesture in the dataset. Therefore, some images in the *No gesture* class are probably more related to the other gestures in the data set than its own class. As a consequence it might be a good idea to split the *No gesture* class into two separate classes: *Background* and *Other gestures*. Additionally in the confusion matrix we can see that the *Point* gesture is classified as the *Peace* sign six percent of the time. This makes sense as they are naturally closely related, and therefore it would be beneficial to only use one of them in a live implementation.

5

Action Recognition

Action recognition can be explained as the classification of any activity by one or multiple agents from recorded sensor data. The word action and activity are used interchangeably through this chapter. The activity can require multiple agents, such as playing chess, but this thesis will only address single person activities, such as hand-waving. Some practical applications of action recognition include assisted living applications for smart homes, health care monitoring applications and security and surveillance applications. This thesis will consider recognizing actions of passengers within a car.

5.1 Previous work

The latest advances in human action recognition all involve deep neural networks. A popular approach for capturing the temporal dependencies is clever use of 3D convolutions [76] [77]. Another possibility is leveraging the region proposals of an object detector, since in theory there will be particular regions of the image that are of high importance when classifying which activity is occurring in an image [78]. A third popular approach for classifying an action is utilizing the dense representation of humans from body pose estimation networks, which is claimed to be of high importance when classifying the action of humans [7] [8].

Body pose estimation can be divided into two categories: single-person and multi-person. The technique to solve single-person pose estimation is dominated by stacked hourglass networks among the state-of-the-art architectures [79] [80] [81] [82]. A. Newell el al. discloses that their small stacked hourglass architecture has an inference time of 75 ms on a Nvidia TitanX GPU.

A pose estimation architecture that excels in inference time (5ms on a Nvidia GTX 1080) is *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields* by Z. Cao et al. [83]. This technique is not only multi-pose and has an inference time that has the potential to enable real-time inference on embedded platforms, but also was accurate enough to win *COCO 2016 keypoints challenge*. Since then, a few publications has exceeded the performance of the part affinity fields architecture, like Mask R-CNN [46], RMPE [84] and Cascaded Pyramid Networks. Unfortunately, these new architectures does not exceed the affinity fields technique in efficiency because they either make use of a 101-layer deep ResNet or the highly inefficient stacked hourglass technique. The authors of Mask R-CNN shows an inference time of 200 ms, but this is achieved using the high-end Nvidia Tesla M40 GPU.

5.2 Pose estimation using Part Affinity Fields

Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields by Z. Cao et al. published in 2016 was the first body pose estimation technique both exceeding state-of-the-art performance (winner of COCO 2016 keypoints challenge) and claiming to have real-time inference capabilities. A common side effect of multi-person pose estimation is that inference time tends to increase with with the number of people present in the image, but this technique enables simultaneous keypoint detection and association of all bodies present, which result in reasonable inference time even for images involving many people.

Pose estimation using Part Affinity Fields is best explained by referencing to the original network architecture, shown in Figure 5.2. Like many deep learning solutions to vision tasks, the first few layers of the network consists of a general feature extractor CNN. This component can be any of the architectures presented in Section 2.6 for easy access to pre-trained weights, but the original paper uses the first 10 layers of VGG-19. The output of the feature extractor is thereafter branched into two isolated components; one responsible for learning to detect body keypoints in the form of returning heat maps $\mathbf{S}^1$, and one responsible for learning associations between the keypoints $\mathbf{L}^1$, called the party affinity fields. There exist one heat map in $\mathbf{S}^1$ for each unique keypoint to track. For example, one heat map could be responsible for detecting every occurrence of a left shoulder in the input image. An association represented by an affinity field can be thought of as a directed limb between two different keypoint categories.

Illustrations of the ground truth heat maps and affinity fields are shown in Figure 5.1. Panel a) is the original image, panel b) represents every heat map concatenated into one image, panel c) represents the x-values of the vectors of every part affinity field and panel d) represents the y-values. In this example, we can clearly see that the part affinity fields act as relations between keypoints where there exists an association in form of a limb. By inspecting the Figure more carefully, we can see that limbs aligned vertically with the image has a brighter y-value of the part affinity field, and limbs aligned more horizontally along the original image has a brighter x-value in of the part affinity field, hence the joint vector is directed from one limb to another. Since a part affinity field is represented by a feature map of x-values and a feature map of y-values, $\mathbf{L}^i$ will consist of twice as many feature maps as there are keypoint associations.

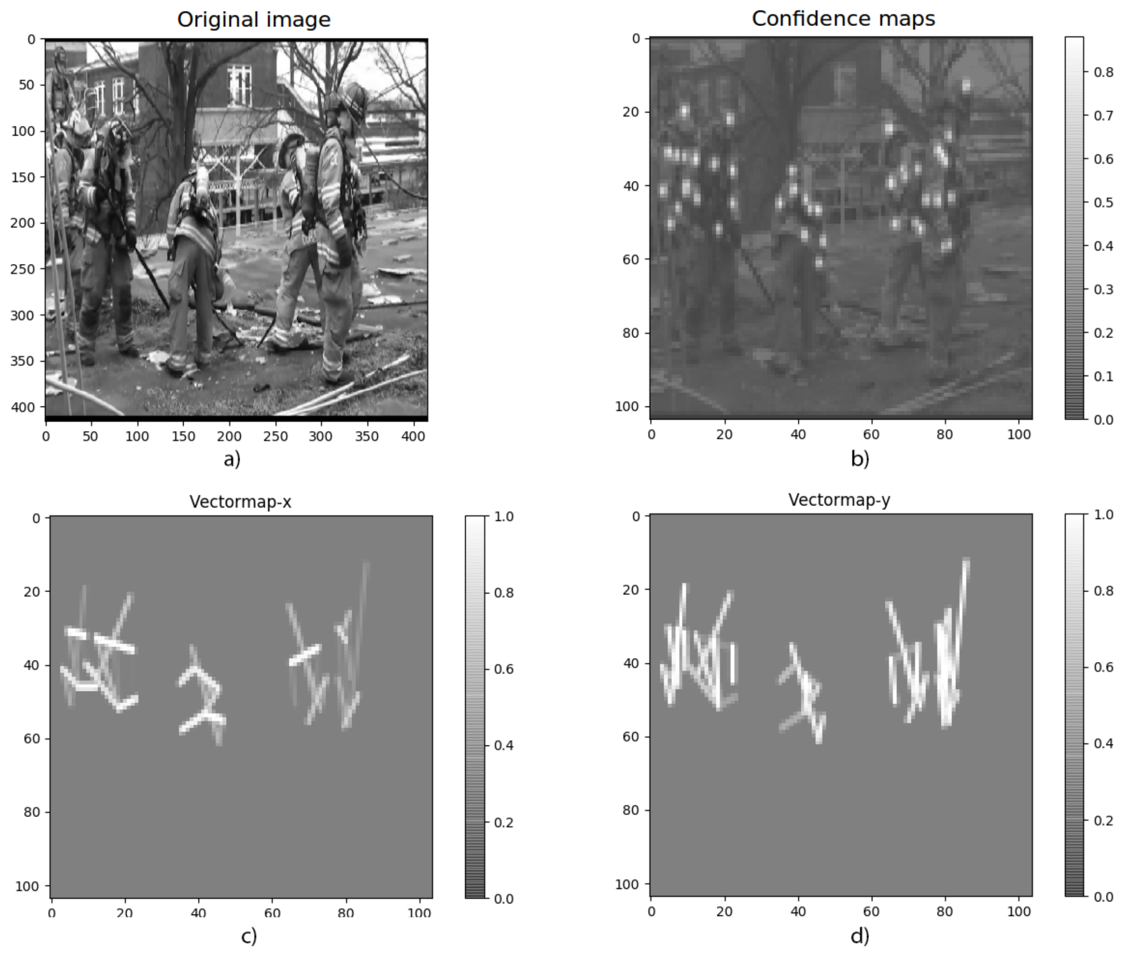


Figure 5.1: The ground truth labels of a sample. a) is the original image, b) are all heat maps concatenated into an image, c) are the x-values of the part affinity fields and d) are the y-values of the part affinity fields.

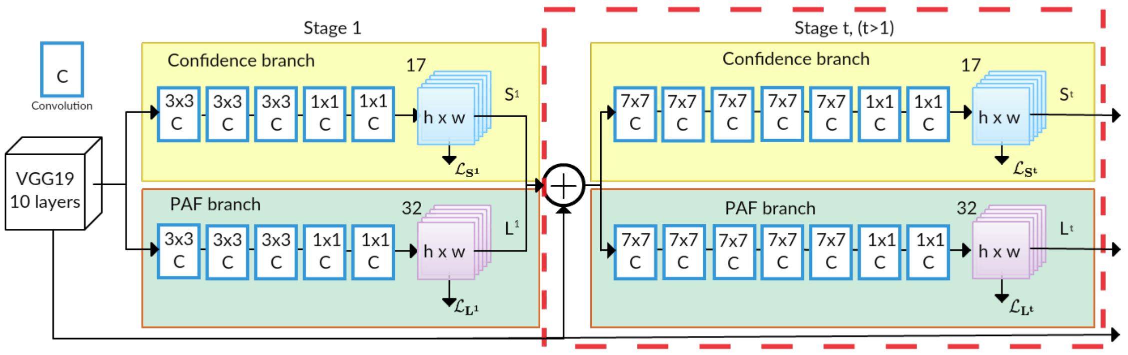


Figure 5.2: Architecture overview of Pose estimation using Part Affinity Fields

The remaining part of the network follows the principle of refinement stages, first introduced in *Convolutional Pose Machines* by Wei et al. [85]. $\mathbf{S^1}$, $\mathbf{L^1}$, along with the output of the feature extractor **F** from stage 1 is concatenated and used as input to stage 2, shown within the dashed lines in Figure 5.2. Every refinement stage consists of five layers with 7×7 filters and two layers with 1×1 filters for both the keypoints branch and the affinity fields branch. What characterizes the refinement stages are that they all learn to produce the same output, each stage an improvement of the previous stage. This is possible due to intermediate supervision at each stage by the loss f_1^t and f_2^t. The original paper show results from using four refinement stages, but there is no set rule for how many refinement stages one should implement. It becomes a performance/efficiency trade-off.

The formula for calculating the output of keypoint heat maps are

$$\mathbf{S}^t = \rho^t(\mathbf{F}, \mathbf{S}^{t-1}, \mathbf{L}^{t-1}), \forall t \geq 2 \tag{5.1}$$

and affinity fields for a refinement stage t are

$$\mathbf{L}^t = \phi^t(\mathbf{F}, \mathbf{S}^{t-1}, \mathbf{L}^{t-1}), \forall t \geq 2. \tag{5.2}$$

The functions ρ^t and ϕ^t represents forward propagation through refinement stage t for the keypoints branch and affinity fields branch respectively.

The loss function for each refinement stage t is calculated by

$$f_{\mathbf{S}}^t = \sum_{j=1}^{J} \sum_{\mathbf{p}} \mathbf{W}(\mathbf{p}) \left\| \mathbf{S}_j^t(\mathbf{p}) - \mathbf{S}_j^*(\mathbf{p}) \right\|_2^2, \tag{5.3}$$

$$f_{\mathbf{L}}^t = \sum_{c=1}^{C} \sum_{\mathbf{p}} \mathbf{W}(\mathbf{p}) \left\| \mathbf{L}_c^t(\mathbf{p}) - \mathbf{L}_c^*(\mathbf{p}) \right\|_2^2, \tag{5.4}$$

and

$$f = \sum_{t=1}^{T} (f_{\mathbf{S}}^t + f_{\mathbf{L}}^t), \tag{5.5}$$

which is regular L_2 loss conditioned on whether an annotation at point $\mathbf{p}$ is present or not. $\mathbf{W}(\mathbf{p})$ equals 0 when image location p is missing an annotation and 1 otherwise. S_j^* is the ground truth heat map of the jth keypoint and L_c^* is the ground truth part affinity field of the cth limb/association. Equation 5.5 is the total loss which makes up for the objective function to minimize when training.

The procedure of creating body pose predictions using heat maps of keypoints and affinity fields from the last refinement stage is approached as a graph matching problem. With K unique keypoint categories, the matching problem is K-dimensional and usually NP-Hard. But with the help of the affinity fields, the matching problem can be divided into smaller subproblems of dimension 2, called maximum bipartite matching.

X. Zhu et al. has proposed some small improvements upon the original work of pose estimation using Part Affinity Fields [86]. One of the suggestions being redundant Part Affinity Fields, i.e. each keypoint will have more than one association to another keypoint. They conclude that this makes it easier for the optimization problem to pair the keypoints into humans correctly.

5.3 Implementation

A few recent publications on action recognition presented in Section 5.1 pointed out the importance of estimating a body pose to use for classifying an action. By having a pose estimation stage and an action recognition stage isolated from each other, the complexity of the time dependency needed to be captured will be severely reduced. This is because the body pose estimation will be evaluated on a frame-by-frame basis and the time dependencies are learned only from the body pose keypoint detections available. Because of the above benefits, we chose to divide up our action recognizer into a body pose estimation stage and classification stage.

5.3.1 Pose Estimation component

Due to the reasons all of the recent state-of-the-art single person pose estimation architectures mentioned in Section 5.1 using the highly inefficient stacked hourglass technique, we had to look elsewhere for inspiration. Among the more efficient but less accurate alternative solutions is *Convolutional Pose Machines*. Thereby we decided to transition into developing a multi-person pose estimator straight away because the multi-person variant of *Convolutional Pose Machines* is Pose estimation using Part Affinity Fields described in section 5.2. This choice will also scale well with the number of passengers present in the vehicle.

We adopted the idea and architecture of Z. Cao et al. [83] for efficient and accurate multi-person pose estimation. The original paper uses a pre-trained stub of VGG19 as feature extractor and 7×7 filters within the refinement stages, so there were room for further experimentation of reducing the number of parameters. These are the efficiency changes we decided to implement:

- Replaced all convolutional layers with its separable convolution equivalent (see Section 2.6.7).
- Replaced the VGG19 feature extractor with the first nine layers of a pre-trained MobileNet.
- Replaced every 7×7 filter with a smaller 3×3 separable convolution filter
- Reduced the number of layers in each refinement stage from 7 to 5
- Removed any tracked keypoints for the face and below the waist

The argument for removing some of the available keypoints is that the legs of passengers will not be visible while sitting properly in the car and the face keypoints are redundant because of Smart Eye's existing facial tracking algorithms. Also, the face and legs keypoints are not of interest for the specific actions to learn in the later stage.

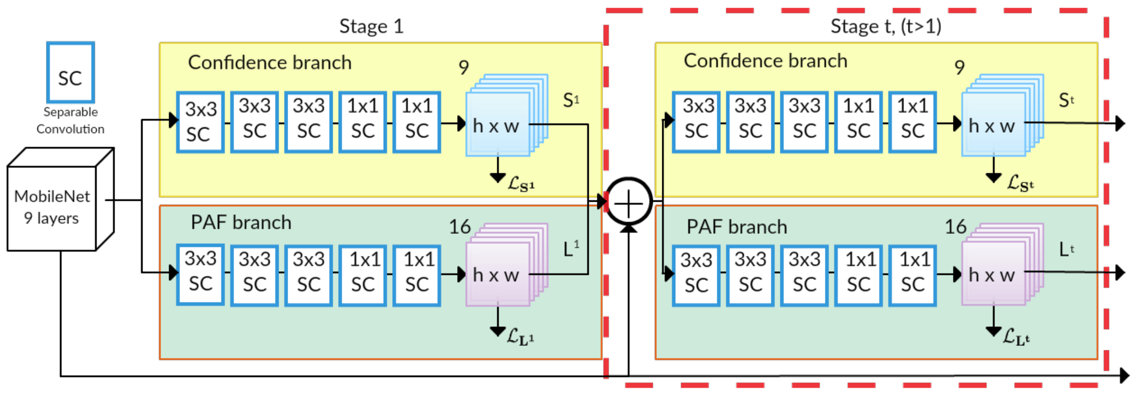


Figure 5.3: Architecture overview of an improved model with respect to efficiency of Pose estimation using Part Affinity Fields

The modified architecture for improved efficiency is shown in Figure 5.3. F consists of the first nine layers of a pre-trained MobileNet, branching out to a sub network for learning keypoint heat maps and another sub network for learning the affinity fields. Both the confidence branch and the affinity branch have identical architecture design in every stage. A branch of the first stage consists of five layers. the first three are using 3×3 separable convolution filters and the last two are using 1×1 separable convolution filters. Unlike the refinement stages of the original architecture, Figure 5.2, our refinement stages have identical architecture as stage 1. There is no underlying thought of having the same architecture at every stage in our architecture, but only a coincidence after converting the 7×7 filters to 3×3 and removing two layers. Non-max suppression is performed on the final output keypoint heat maps $\mathbf{S}^t$ to obtain a small discrete set of keypoint predictions, and thereafter fed into a maximum matching optimization problem along with the affinity fields to simplify the procedure, just like in the original approach [83]. The output of the maximum matching optimization is a list of humans. If a total of k keypoints are tracked, then a human is represented by a vector of size $2k$, containing the x and y coordinates of every keypoint.

The keypoints to be tracked are determined by the keypoint labels available in the dataset. We used the *COCO keypoints 2017* dataset for training. This dataset was used because it is the largest pose estimation dataset available (120,000 labeled images) and it is backed by Microsoft. A total of 17 unique keypoint categories are available in this dataset, and which ones we decided to exclude due to the unique application of monitoring passengers of cars are shown in Table 5.1

Keypoint category	Z. Cao et al.	our implementation
nose		
left_eye	✓	
right_eye	✓	
left_eat	✓	
right_ear	✓	
left_shoulder	✓	✓
right_shoulder	✓	✓
left_elbow	✓	✓
right_elbow	✓	✓
left_wrist	✓	✓
right_wrist	✓	✓
left_hip	✓	✓
right_hip	✓	✓
left_knee	✓	
right_knee	✓	
left_ankle	✓	
right_ankle	✓	

Table 5.1: The keypoints available in the dataset that were used when training

5.3.2 Action recognition component

Our thesis about the action recognition is that estimating a body pose as a preprocessing step will not only improve the results (concluded by [7] and [8]), but also significantly reduce the complexity of the problem if only the predicted body pose is used as input. If the type of actions involve interacting with the environment, such as cooking food, that information will of course be lost by only considering the body pose. We thereby make sure to use a dataset that is predictable using only information from the body pose. This restriction will also be enough for the type of actions Smart Eye is interested in recognizing.

The spatial information was lost by extracting the body pose, but by storing sequences of body pose estimations, the problem of capturing temporal dependencies was introduced. Therefore, when designing the architecture, a combination of fully connected layers and recurrent layers were considered. By following the work flow principles of Section 2.9, an initial simple model was implemented using only one LSTM cell and a softmax function, see Figure 5.4. As the unrolling of the LSTM cell shows in the Figure, the model follows the many-to-one principle, i.e. given a sequence of inputs, predict an output only at the last time step.

The dataset used for experiments while Smart Eye's dataset was under development was *KTH action database*, containing 2391 videos of dynamic actions performed by one human without the need of interacting with the environment. The actions available are: walking, jogging, running, boxing, hand waving, and hand clapping.

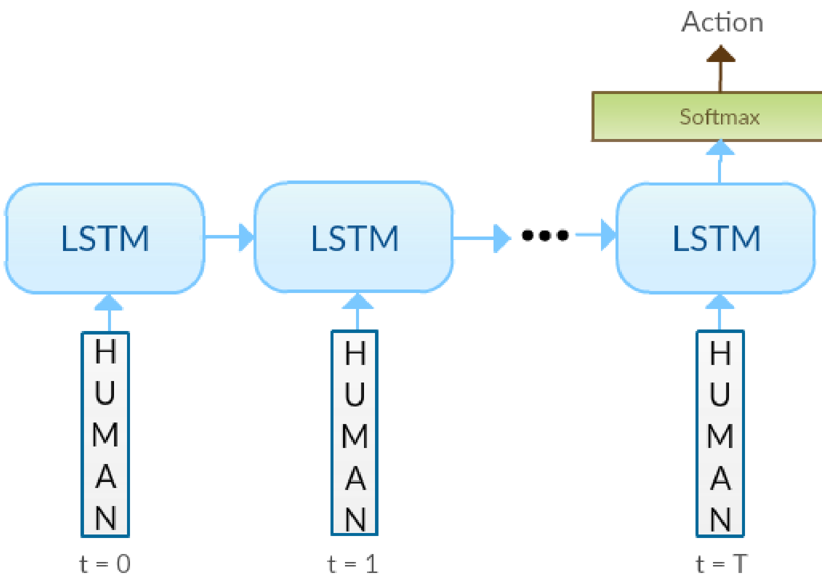


Figure 5.4: Architecture overview of the action recognizer. An LSTM cell unrolled over time shows how the sequence of human vectors are used as input for one prediction.

To have as reliable pose estimation inputs as possible when training the action recognition component, we used a pre-trained model of the original part-affinity fields body pose estimator. This would not limit the performance of the action recognition component by the slightly worse performance of our light implementation of the part-affinity fields pose estimator. Also, if the body pose estimator was not able to detect a pose for at least 50% of the images of a particular video sequence, then we removed that video sequence from the dataset.

5.4 Results

The mean average precision of our final body pose estimator model is 8.7 mAP and the inference time on Jetson TX2 is 3.6 FPS. The mAP over time during training can be seen in Figure 5.5. Compared to the best methods published as of today, shown in Table 5.2, we perform the worst, but with the trade-off of being able to perform real-time inference on the Jetson TX2 platform.

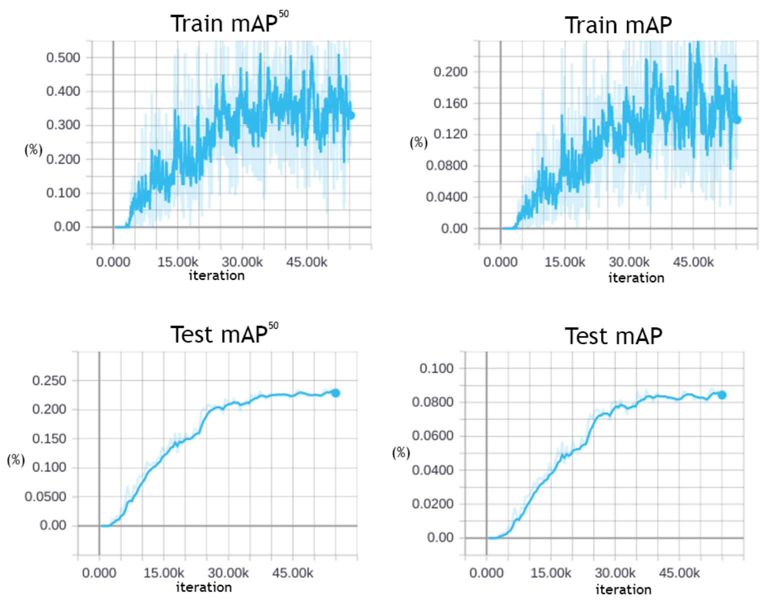
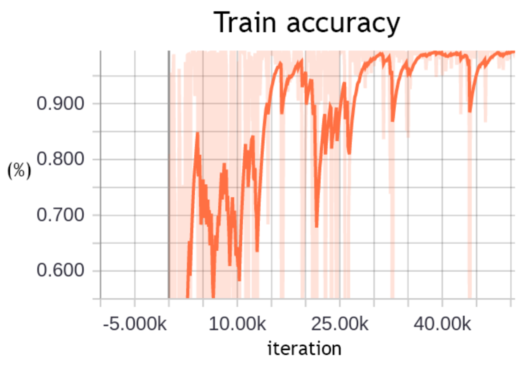


Figure 5.5: Mean average precision of the body pose estimator during training. see [1] for more information about the metrics.

Methods	mAP	mAP^{50}	FPS
Mask R-CNN	69.2	90.4	2.5 (Tesla P100)
RMPE [84]	68.8	87.5	0.5
Part Affinity Fields original [83]	60.5	83.4	10 (GTX 1080)
Stacked Hourglass Network [79]	46	75	12 (TitanX)
Ours (MobileNet + light PAF)	8.7	23	3.6 (Jetson TX2)

Table 5.2: Comparison of our body pose estimator to other methods in terms of performance (mAP) and inference time (FPS).

Figure 5.6 shows the train accuracy and test accuracy during training of the action recognition module. The dataset used was *KTH action database*. The training procedure was very fast and 100% accuracy on a training batch were achieved already at iteration 500. The test accuracy after training is 100%, but keep in mind that some video sequences were excluded from the dataset.



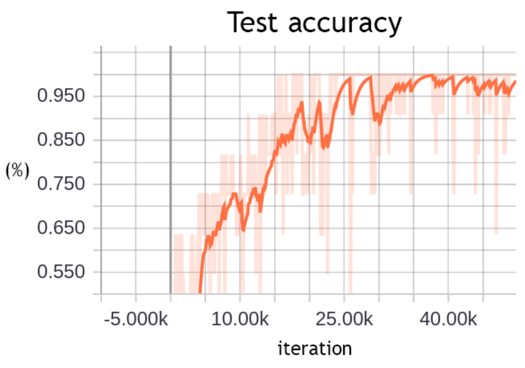


Figure 5.6: The train and test accuracy of the action recognizer during the training procedure. The train and test accuracy consequently performed 100% after convergence.

5.5 Discussion

Our light implementation of multi-person pose estimation using affinity fields has successfully proven to be able to show real-time performance on a Nvidia Jetson TX2 embedded platform. The inference time should be about three times faster than the original affinity fields architecture but with the cost of a significant reduction in robustness. This becomes a trade-off where the number of layers and filter sizes of the refinement stages can be increased to utilize any available computing power that is left while still satisfying the inference time requirements.

The use of a body pose estimator as a preprocessing step for the action recognition component has enabled a feature otherwise not available using an end-to-end network like many other publications mentioned in Section 5.1. This feature being the support of recognizing different actions from multiple persons in one video. The body pose estimator also introduces a limitation that can be critical depending on the type of actions to learn, namely the environment has been stripped away. This will remove the ability to learn any action that involves interaction with an object, such as using the mobile phone in different ways.

A practical benefit of separating the problem into a pose estimation step and an action recognition step is modularity. Smart Eye is a tracking company and by separating the problem into a body pose estimation stage, this output can act as a general body joint tracking framework that any future component can use in addition to the action recognition component.

As for the action recognizer module, its highly likely that removing some of the video sequences because of too few detections from the body pose estimator made the result biased. The current state-of-the-art result on the KTH action database is 96.8% accuracy [87], and because of our introduced bias, we can’t compare our result with other architectures. Apart from not being able to compare performance, we can still conclude that the body pose estimator of the tracking framework provides a very good abstract representation of human bodies and can be used as a base for implementing action recognition modules.

6

Real-time system

As the performance of Deep Learning solutions has become accurate enough for practical applications during the last few years, a new central question has been addressed, namely how to deploy these models in resource constrained environments like embedded systems and mobile devices. The hardware industry has answered with inventions like the Neural Engine from Apple [88] and the Jetson embedded system series from Nvidia [89]. In this chapter, we will investigate how applicable our most efficient and accurate proposed models are in a GPU accelerated embedded environment.

One of the end goals of this thesis is to develop a prototype that consists of the modules presented in previous chapters that is able to operate real-time on a Nvidia Jetson TX2 embedded platform. More specifically, the on-board camera of Jetson TX2 should continuously capture images as input to the tracking framework, consisting of the body pose estimator and the object detector, and thereafter use the extracted features as input to a hand gesture classifier and an action recognizer.

The prototype will not be implemented inside a car for testing, but only on the Jetson TX2 standalone platform for measurements to see if our models are efficient enough to be jointly implemented together in an embedded environment.

6.1 Implementation

The live inference software was implemented to run on a Nvidia Jetson TX2 platform using the GPU-accelerated Tensorflow framework. Recall the initial idea of the architecture from Figure 1.1. A more detailed visualization of the final real-time inference architecture is shown in Figure 6.1. There is a separate pipeline for handling hand gestures and a separate pipeline for handling body pose actions. Also, we implemented a guard that is responsible for deciding when a deep inference should be performed and when its not worth it, to save computing power.

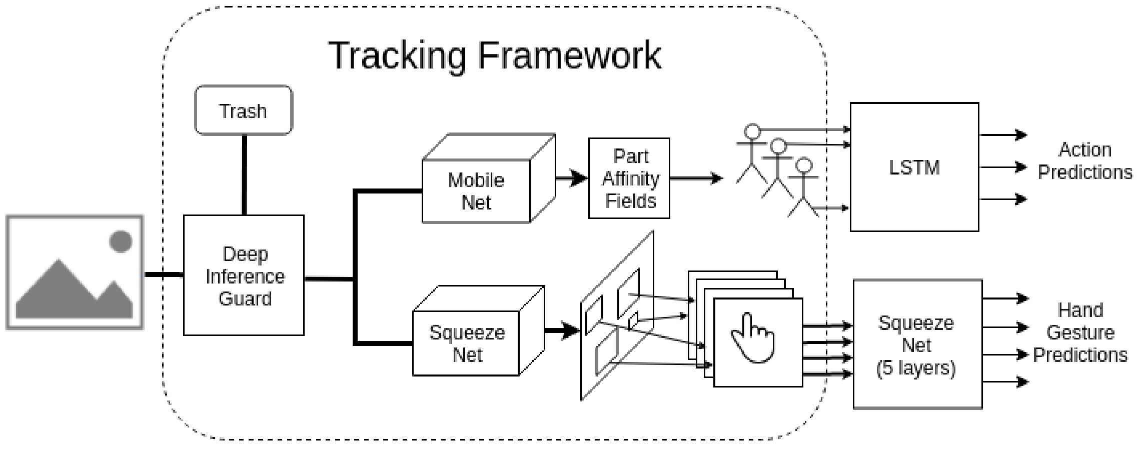


Figure 6.1: The final architecture of our prototype

6.1.1 Deep inference guard

We call the first component of our real-time system a deep inference guard. Its mission is to determine when movement in the input image is high enough for a deep inference to be worth it. Our argument for implementing this component is that there can be up to multiple seconds in a driving scenario where neither of the passengers' hands nor body parts are moving, and therefore the predictions of the previous frame can be used instead.

The implementation follows mostly the same steps as the fixed window hand detector implementation in Section 3.3.1. More specifically, lines 2-12 of Algorithm 2 are implemented in the deep inference guard and performed on the whole image to extract $hist_v$ and $hist_h$. Thereafter, a simple comparison between the sum of visible pixels and the total number of pixels are performed to determine how many percent of the original image is captured by the exponentially averaged moving edges, see Algorithm 3. If the percent of visible moving edges are above θ, then deep inference is allowed. θ can be adjusted to meet sensitivity requirements. A value of 0 will always allow deep inference and a value of 1 will always block deep inference.

```
input  : $I$ as the input image from camera, $kwargs$ as the parameters needed for
         Algorithm 2, $\theta$ as the deep inference threshold ( $\theta \in [0,1]$ )
output: $g$ as a boolean (True == allow, False == block)
1 $pixels_{tot} \leftarrow 2 \cdot product(I.shape)$;
2 $hist_h, hist_v \leftarrow Algorithm2(I, kwargs)$;
3 $pixels_{vis} \leftarrow sum([hist_h, hist_v])$;
4 $s \leftarrow \frac{pixels_{vis}}{pixels_{tot}}$;
5 $g \leftarrow s > \theta$ return $g$
```

Algorithm 3: The deep inference guard algorithm. Returns true or false whether to perform a deep inference or not.

6.1.2 Gesture recognition pipeline

The gesture recognition module utilizes the object detection capabilities of the tracking framework. Every detected hand is extracted from the input image at every time step. When storing a sequence of hand detections over time, a detected hand at time step t_n has to be paired together with the same hand detected at time steps t_i where $i < n$. To match a detected hand with any hand at a previous time step, their intersection-over-union (IoU) was calculated [15]. If the IoU between a detected hand at time t_{i-1} and a detected hand at time t_i was above a certain threshold θ, it was considered to be the same hands. If the IoU was too low when comparing to any previously detected hand at time t_i, it was considered to be a new hand not seen before in the sequence of images.

These sequences of detected hands are stored in separate buffers and the hand gesture classifier performs an inference on a particular buffer whenever it becomes large enough (15-30 frames for our dynamic hand classifier and 1 frame for our static hand classifier). In the case of multiple detected hands that need to be classified in the same image, we were able to use the batch dimension of the gesture classifier to perform classification on all detections simultaneously.

6.1.3 Action recognition pipeline

The action recognizer module utilizes the body pose estimation capabilities of the tracking framework. Just like the input for the dynamic hand gesture classifier, the action recognizer requires a detected body pose as a sequence over time. An approach similar to the hand detections for matching and storing the body pose detection was implemented, but with a difference in the matching algorithm. Instead of measuring IoU, the euclidean distance between the mean keypoint for every detected body pose was compared. If the distance between two body poses of different time steps were within a certian distance threshold θ, they were considered to originate from the same body and thereby stored in the same buffer.

Since the action recognizer is dynamic and only requires one input pose at each time step (unlike the gesture classifier that requires more because of 3D convolutions), it enables a continuous flow of body pose inputs and an action prediction for every time step. By performing an inference at each time step, the buffer size was able to be kept at a size of one, with the addition that a separate hidden state of the LSTM for every unique body pose sequence needed to be stored.

6.1.4 Merged feature extractor

Since the hand detector and the pose estimator both use a general feature extractor CNN at the beginning of each network, we experimented with merging these layers into a common feature extractor, see Figure 6.2. This resulted in a multitask learning problem, where the common feature extractor needs to learn good features for both object detection and keypoint estimation. Training was performed by using a pre-trained SqueezeNet as common layers and then freezing them. The idea was that no task should dominate in which direction to update the weights and a pre-trained SqueezeNet on ImageNet classification should already have learned valuable features. Freezing the majority of layers and only training the upper layers has been shown to work well for transfer learning scenarios when the end goal is also image classification [90]. We try this approach for transfer learning for object detection and pose estimation.

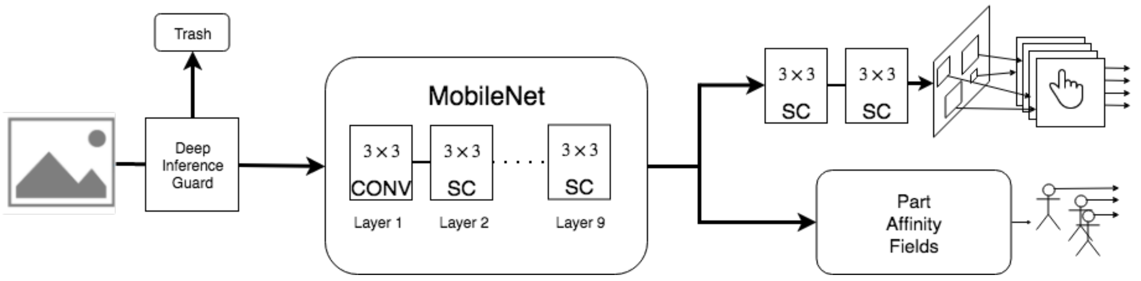


Figure 6.2: The tracking framework with the merged feature extractor. After an image has passed the deep inference guard it is first sent through the first nine layers of a MobileNet before being divided into the object detector and body pose estimator.

6.2 Results

Inference times for the hand gesture classifiers and action recognizer were tested separately to examine how much overhead they would cause when appending on top of the tracking framework. Unfortunately, the dynamic hand gesture classifier were not able to run on Jetson TX2 due to limited RAM. However, inference times for the static Hand Gesture Classifier are shown in Table 6.1 for different amount of input hands. Inference times for the action recognizer were also tested with different amount of input body poses at once, presented in Table 6.2.

# Hands	inference (ms)	FPS
1	21	48
2	29	34
3	40	25
4	47	21

Table 6.1: Inference times for the static hand gesture classifier with different number of hands to classify at once.

# Body poses	inference (ms)	FPS
1	7	143
2	8	125
3	14	71
4	17	59
5	19	52

Table 6.2: Inference times for the Action Recognizer with different number of hands to classify at once.

Inference times for our implemented modules are shown in Table 6.3. Inference times for each module tested separately, the final tracking framework and the framework with additional modules are shown.

Module	inference (ms)	FPS
Object Detector (SqueezeNet)	62	16.7
Action Recognizer	7	142
Hand Gesture Classifier (Static)	21	48
Hand Gesture Classifier (Dynamic)	-	-
Body Pose Estimator	280	3.6
Tracking Framework	390	2.6
Tracking Framework & Hand Gesture Classifier (Static)	410	2.4
Tracking Framework & Action Recognizer	397	2.5
Tracking Framework & Gestures & Actions	417	2.4

Table 6.3: Inference times of separate modules and combined modules of the real-time inference implementation.

6.3 Discussion

Empirical results of the inference guard were promising. It successfully blocked any deep inferences whenever the changes over time were insignificant. The guard also includes a threshold θ that enables easy tuning of its sensitivity. We believe that saving computing power by avoiding to perform inference on similar images is critical when implemented in an embedded environment. A problem that may occur when deployed inside a car is that major sections of the background may be moving because of transparent windows, and we suspect that further development of this guard component is needed to emphasize the moving pixels of the areas where passengers are located.

The framework can successfully operate real-time on the Jetson TX2 platform. Unfortunately, the dynamic hand gesture classifier was not able to fit into RAM. Testing the live inference capabilities of the dynamic hand gesture classifier on a machine with bigger RAM revealed additional setbacks. When the framework is cropping a detected hand from the input image, the information of where in space the hand is moving is lost. For example, in the case of the "swipe right" gesture, the hand is moving from left to right in space over time, but the hand detector removes this information because the bounding box prediction is moving along the hand every time step. This resulted in the dynamic hand gesture classifier in combination with the object detector to be unusable. Instead, we recommend having a fixed window to be input to a dynamic hand gesture classifier without cropping out the hand. This will preserve the movement in space over time of the hand.

The dynamic hand gesture model is heavy with an unattractive inference time. By jumping more than one frame at a time allows the system to a deep inference less frequent. However you do not want to slide too heavy, making the model ignore the first movements of the gesture. Additionally, making less inferences also potentially increases the response time to detect a gesture from our model. After experiments the real time implementation strides five frames between each deep inference, constituting the best trade off between inferences per second and information retained.

The static hand gesture classifier was able to operate real-time together with the tracking framework on Jetson TX2. Empirical results also revealed that the performance of the classifier was moderate with the object detector. This is due to the classifier not being dependant on movements over time, and only a snapshot of the current pose of the hand is necessary to classify a gesture.

Since the action recognition module only consisted of one LSTM layer and a softmax, it was able to operate real-time on top of the tracking framework with only 7ms overhead. According to Table 6.2, increasing the number of body pose inputs to 5 for a single image added only an additional 12ms overhead, which means that this implementation will be able to handle an in-car scenario with maximum number of passengers present.

Freezing the first 6-8 layers of the SqueezeNet YOLO object detector before training resulted in the model not learning to detect anything at all. This means that the merged feature extractor CNN implementation was not trainable, and was therefore not included in the final prototype. We suspect SqueezeNet having not very rich transfer learning capabilities, since it has much fewer parameters than other architectures that are common for transfer learning, such as VGG19.

7 Conclusions

The proposed system was implemented with the objectives that was established in the beginning of this thesis in mind. Overall, we believe that the thesis was successful carrying out the set objectives. However, all implementations currently assumes an RGB input, which was not desirable with the prerequisites set from Smart Eye. Furthermore three bonus objectives were explored; dynamic hand gestures, multi-person pose, and a merge of the first layers of feature extractors of the object detector and body pose estimator. As a consequence of the embedded environment constraint, all our models have been experimented with to find a suitable trade-off between performance and efficiency.

7.1 Future work

Given additional time, there exists several additional adaptions of our system to make it more valuable and by extension integrable into Smart Eye's product segment. In the field of Deep Learning in general and Computer Vision in particular, new state-of-the-art solutions are continuously published in a rapid pace. Therefore, we anticipate that new improved ideas of solving similar problems to the ones this thesis tackles will be published on an on-going basis, becoming the solutions to ideally research for future work.

Our system shall be able to work under various lightning conditions, day and night. During night time, color information is redundant, thus there is no need for RGB input, which also complies with Smart Eye's main sensor of choice - near infrared cameras. A requirement to train the models used throughout this thesis is the use of pre-trained weights. Architectures such as SqueezeNet is extensively used, but currently there are no pre-trained weights available in gray-scale format. Therefore retraining SqueezeNet on ImageNet and converting the images and the architecture to one channel input would be a necessity for an implementation usable by Smart Eye given their current constraints. Another direction would be to convert gray-scale input to a RGB-format, using the current implementation, however this may result in a to significant performance loss.

Embedded environments do not only require a tractable inference time, but also small model sizes. Smaller models make less memory references and thus require less energy. In addition, it also makes over air updates on limited bandwidth easier, and may result in the entire model being fit into SRAM. For this purpose it is our belief that all real-time implementations should be coded in C/C++ with frameworks such as Nvidia's TensorRT when deploying live on a Jetson TX2. TensorRT performs automated compression of model size through methods such as quantization of weights and removing unnecessary nodes. Additionally, you can cluster weights from hidden layers to a discrete representation along with aforementioned methods, as proposed in *Deep Compression* by Song Han et al. Deep compression reduces VGG-19 model size 49 times and speed-up off up to 3 times [37].

The proposed framework is currently built for the purpose of helping our action recognizer and hand gesture classifier. However, ideally it should be able to support the classification of arbitrary tasks. The object detector is only trained on detecting hands. If found useful, additional objects such as mobile phones may be added to the detector without interfering with the network architecture. In addition, the human body pose estimator is also architectured in a way to be able to add more joints.

An issue with recognizing actions solely on human body pose estimations is that it completely removes environmental information. Therefore the potential to recognize actions that interacts with the the environment, e.g. talking in a phone or interacting with the infotainment, may be diminished. Extending our framework, the object detector, to detect additional objects such as mobile phones and integrating it into the action recognizer may increase its applicability, while keeping the inference time close to as before.

7.2 Conclusion

The purpose of this thesis was to investigate how Deep Learning based methods can be used in order to increase passenger safety. This thesis response to the open-ended question comes in the form of a Deep Learning based framework for monitoring passengers inside a car. The framework provide tools with the possibility to be integrated into solutions such as automated adjustment of seats and mirrors, and observing passengers in various ways. Additionally, this thesis demonstrates the applicabilities of said framework in practice with an action recognizer and a hand gesture classifier.

The body pose estimator is trained to track keypoints from the upper body and the object detector to keep track of hands. However, their general architecture enables extending the capabilities by tracking additional objects and body keypoints if the required labels are available. Adding extra tracking capabilities will only increase the last layer of the architectures, hence our framework is scalable.

The applicability of the object detection module of the framework has been demonstrated by training on hand detection datasets and using detected hands as input to a gesture classifier. We conclude that this approach works well for detecting and recognizing static hand gestures from any hands located anywhere within the receptive field of the camera. However, this approach is unusable for the case of dynamic hand gestures, because cropping the hand results in loss of location in space.

We have also demonstrated the applicability of detected keypoints from our tracking framework. Our results show that the keypoints represent good features for learning actions involving body part movements. This implementation assumes that the keypoints will include the information needed to classify an action, i.e. no external objects will be taken into account. Expanding actions to include external objects has been left to future work.

Lastly, we have successfully implemented the proposed tracking framework onto a Nvidia Jetson TX2 embedded platform, with a run time of 2.6 fps. Unfortunately, our dynamic hand gesture classifier has too many parameters to fit onto the embedded system. However, we can include the static hand gesture recognition module and the action recognition module on top of the framework. It reduces the run time to 2.4 fps, but still considered a successful real-time implementation.

Bibliography

[1] Microsoft COCO. (2014) Keypoint Evaluation - Object Keypoint Similarity. [Online]. Available: http://cocodataset.org/#keypoints-eval

[2] National Highway Traffic Safety Administration. (2015) Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey. [Accessed: 2017-11-22]. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115

[3] National Highway Traffic Safety Administration . (2011) Drowsy Driving. [Accessed: 2017-11-22]. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/811449

[4] Y. Hatakeyema, "Feasibility Study of Drowsy Driving Prediction based on Eye Opening Time," in *SAE Technical Paper*. SAE International, 03 2017. [Online]. Available: https://doi.org/10.4271/2017-01-1398

[5] B. Reddy, Y. H. Kim, S. Yun, C. Seo, and J. Jang, "Real-Time Driver Drowsiness Detection for Embedded System Using Model Compression of Deep Neural Networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, July 2017, pp. 438–445.

[6] V. Corcoba Magaña, M. Muñoz Organero, J. Arias Fisteus, and L. Sánchez Fernández, "Estimating the stress for drivers and passengers using deep learning," *CEUR-WS. org*, 2017.

[7] B. X. Nie, C. Xiong, and S.-C. Zhu, "Joint action recognition and pose estimation from video," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1293–1301, 2015.

[8] G. Ch, I. Laptev, C. Schmid, G. Ch, and G. Ch, "P-CNN : Pose-based CNN Features for Action Recognition," *International Conference on Computer Vision (ICCV)*, pp. 3218–3226, 2015.

[9] IHS Markit. (2013) Sales of Automotive Proximity and Gesture Recognition Systems Shift into High Gear. [Accessed: 2018-01-22]. [Online]. Avail-

able: http://news.ihsmarkit.com/press-release/design-supply-chain-media/sales-automotive-proximity-and-gesture-recognition-systems-s

[10] Regeringen. (2017) Händerna på ratten – inte på mobilen. [Accessed: 2018-01-22]. [Online]. Available: http://www.regeringen.se/pressmeddelanden/2017/11/handerna-pa-ratten--inte-pa-mobilen/

[11] Sony Depthsensing Solutions. (2018) Sony DepthSense CARlib. [Online]. Available: https://www.sony-depthsensing.com/DepthSense/Markets/Automotive

[12] L. Deng and D. Yu, "Deep Learning: Methods and Applications," Tech. Rep., 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/

[13] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain," *Psychological Review*, pp. 65–386, 1958.

[14] S. Yeung, "Lecture 6 | Training Neural Networks I," Stanford University, 2017. [Online]. Available: https://www.youtube.com/watch?v=wEoyxE0GP2M&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=6

[15] A. Ng. (2017, August) Convolutional Neural Networks - Intersection Over Union. Coursera. [Online]. Available: https://www.coursera.org/specializations/deep-learning

[16] J. Hui. (2018) mAP (mean Average Precision) for Object Detection. [Online]. Available: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

[17] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.0, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[18] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, vol. abs/1412.6, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[19] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems*, 1992, pp. 950–957.

[20] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

[21] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *CoRR*, vol. abs/1502.0, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[22] M. J. Jones and P. Viola, "Robust real-time object detection," in *Workshop on Statistical and Computational Theories of Vision*, vol. 266, 2001, p. 56.

[23] A. Karpathy. (2017, May) CS231n Convolutional Neural Networks for Visual Recognition. Stanford University. [Accessed: 2018-08-26]. [Online]. Available: http://cs231n.github.io/convolutional-networks/

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR*, vol. abs/1211.5063, 2012. [Online]. Available: http://arxiv.org/abs/1211.5063

[26] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-term Dependencies with Gradient Descent is Difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[27] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[28] C. Olah. (2017) Understanding LSTM Networks. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[29] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "ImageNet Large Scale Visual Recognition Challenge," *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[30] V. Golkov, A. Dosovitskiy, J. I. Sperl, M. I. Menzel, M. Czisch, P. Sämann, T. Brox, and D. Cremers, "q-Space Deep Learning: Twelve-Fold Shorter and Model-Free Diffusion MRI Scans," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1344–1351, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[31] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *CoRR*, vol. abs/1605.0, pp. 1–7, 2016. [Online]. Available: http://arxiv.org/abs/1605.07678

[32] D. H. Hubel and T. N. Wiesel, "Receptive Fields and Functional Architecture of Monkey Striate Cortex," *Journal of Physiology (London)*, vol. 195, pp. 215–243, 1968.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning ap-

plied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, nov 1998.

[34] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *CoRR*, vol. abs/1409.4, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR*, vol. abs/1512.0, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[37] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," pp. 1–14, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149https://github.com/songhan

[38] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," pp. 1–13, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR*, vol. abs/1704.0, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[40] A. Karpathy, "What I learned from competing against a ConvNet on ImageNet," 2014. [Online]. Available: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/

[41] A. Ng. (2017, June) Structuring Machine Learning Projects (Course 3 of the Deep Learning Specialization). Coursera. [Online]. Available: https://www.coursera.org/specializations/deep-learning

[42] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[43] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2, 2013. [Online]. Available: http://arxiv.org/abs/1311.2524

[44] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: http://arxiv.org/abs/1504.08083

[45] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *CoRR*, vol. abs/1506.0, no. 6, pp. 1137–1149, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[46] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, “Mask R-CNN,” *CoRR*, vol. abs/1703.0, 2017. [Online]. Available: http://arxiv.org/abs/1703.06870

[47] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, “Deformable Convolutional Networks,” *CoRR*, vol. abs/1703.0, 2017. [Online]. Available: http://arxiv.org/abs/1703.06211

[48] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path Aggregation Network for Instance Segmentation,” *CoRR*, vol. abs/1803.0, 2018. [Online]. Available: http://arxiv.org/abs/1803.01534

[49] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *CoRR*, vol. abs/1506.0, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[50] M. J. Shafiee, B. Chywl, F. Li, and A. Wong, “Fast YOLO: A Fast You Only Look Once System for Real-time Embedded Object Detection in Video,” *CoRR*, vol. abs/1709.0, 2017. [Online]. Available: http://arxiv.org/abs/1709.05943

[51] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, “SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving,” *CoRR*, vol. abs/1612.0, 2016. [Online]. Available: http://arxiv.org/abs/1612.01051

[52] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” *CoRR*, vol. abs/1612.0, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[53] T. H. N. Le, Y. Zheng, C. Zhu, K. Luu, and M. Savvides, “Multiple Scale Faster-RCNN Approach to Driver’s Cell-Phone Usage and Hands on Steering Wheel Detection,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 46–53, 2016.

[54] T. Hoang, N. Le, C. Zhu, Y. Zheng, K. Luu, M. Savvides, T. H. N. Le, C. Zhu, Y. Zheng, K. Luu, and M. Savvides, “Robust hand detection in Vehicles,” in *2016 23rd International Conference on Pattern Recognition (ICPR)*, dec 2016, pp. 573–578.

[55] T. H. N. Le, K. G. Quach, C. Zhu, C. N. Duong, K. Luu, and M. Savvides, “Robust Hand Detection and Classification in Vehicles and in the Wild,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2017-July, pp. 1203–1210, 2017.

[56] N. Das, E. Ohn-Bar, and M. M. Trivedi, “On Performance Evaluation of Driver

Hand Detection Algorithms: Challenges, Dataset, and Metrics," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, sep 2015, pp. 2953–2958.

[57] S. Bambach, S. Lee, D. J. Crandall, and C. Yu, "Lending A Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions," in *The IEEE International Conference on Computer Vision (ICCV)*, dec 2015.

[58] N. Das, E. Ohn-Bar, and M. M. Trivedi, "On performance evaluation of driver hand detection algorithms: Challenges, dataset, and metrics," in *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on.* IEEE, 2015, pp. 2953–2958.

[59] S. Bambach, S. Lee, D. J. Crandall, and C. Yu, "Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions," in *Computer Vision (ICCV), 2015 IEEE International Conference on.* IEEE, 2015, pp. 1949–1957.

[60] BKZero. (2017) Tensorflow issue - slim.separable_conv2d is too slow. GitHub. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/12132

[61] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[62] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. H. Jin, S. Zhao, and K. Keutzer, "SqueezeNext: Hardware-Aware Neural Network Design," *CoRR*, vol. abs/1803.10615, 2018. [Online]. Available: http://arxiv.org/abs/1803.10615

[63] J. Guo and S. Li, "Hand gesture recognition and interaction with 3D stereo camera," 2011, The Project Report of Australian National University.

[64] H. Cheng, Z. Dai, Z. Liu, and Y. Zhao, "An image-to-class dynamic time warping approach for both 3D static and trajectory hand gesture recognition," *Pattern Recognition*, vol. 55, pp. 137 – 147, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320316000157

[65] A. Camurri, G. Volpe, S. A. (e-book collection), and S. (e-book collection), *Gesture-based communication in human-computer interaction: 5th International Gesture Workshop, GW 2003 : Genova, Italy, April 2003 : selected revised papers.* New York: Springer, 2004, vol. 2915.

[66] D. Tran, L. D. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "C3D: Generic Features for Video Analysis," *CoRR*, vol. abs/1412.0767, 2014. [Online]. Available: http://arxiv.org/abs/1412.0767

[67] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, “Large-Scale Video Classification with Convolutional Neural Networks,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 1725–1732.

[68] H. J. Escalante, V. Ponce-López, J. Wan, M. A. Riegler, B. Chen, A. Clapés, S. Escalera, I. Guyon, X. Baró, P. Halvorsen, H. Müller, and M. Larson, “ChaLearn Joint Contest on Multimedia Challenges Beyond Visual Analysis: An overview,” in *2016 23rd International Conference on Pattern Recognition (ICPR)*, Dec 2016, pp. 67–73.

[69] B. Zhou, A. Andonian, and A. Torralba, “Temporal Relational Reasoning in Videos,” *CoRR*, vol. abs/1711.08496, 2017. [Online]. Available: http://arxiv.org/abs/1711.08496

[70] X. Shi, Z. Chen, H. Wang, D. Yeung, W. Wong, and W. Woo, “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting,” *CoRR*, vol. abs/1506.04214, 2015. [Online]. Available: http://arxiv.org/abs/1506.04214

[71] G. Zhu, L. Zhang, P. Shen, and J. Song, “Multimodal Gesture Recognition Using 3-D Convolution and Convolutional LSTM,” *IEEE Access*, vol. 5, pp. 4517–4524, 2017.

[72] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,” *CoRR*, vol. abs/1406.4729, 2014. [Online]. Available: http://arxiv.org/abs/1406.4729

[73] R. B. Miller, “Response Time in Man-computer Conversational Transactions,” in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS ’68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476628

[74] Twenty Billion Neurons GmbH. (2018) the 20bn-jester dataset v1. [Accessed: 2018-06-08]. [Online]. Available: https://20bn.com/datasets/jester

[75] S. Marcel. (1999) Hand Posture and Gesture Datasets static hand posture database. [Online]. Available: http://www-prima.inrialpes.fr/FGnet/data/10-Gesture/gestures/main.html

[76] L. Wang, Y. Qiao, and X. Tang, “Action recognition with trajectory-pooled deep-convolutional descriptors,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June, pp. 4305–4314, 2015. [Online]. Available: http://arxiv.org/abs/1505.04868

[77] C. Feichtenhofer, A. Pinz, and A. Zisserman, “Convolutional Two-Stream Network Fusion for Video Action Recognition,” *CoRR*, vol. abs/1604.0, 2016. [Online]. Available: http://arxiv.org/abs/1604.06573

[78] G. Gkioxari, R. B. Girshick, and J. Malik, “Contextual Action Recognition with R*CNN,” *CoRR*, vol. abs/1505.0, 2015. [Online]. Available: http://arxiv.org/abs/1505.01197

[79] A. Newell, K. Yang, and J. Deng, “Stacked Hourglass Networks for Human Pose Estimation,” *CoRR*, vol. abs/1603.0, 2016. [Online]. Available: http://arxiv.org/abs/1603.06937

[80] C.-J. Chou, J.-T. Chien, and H.-T. Chen, “Self Adversarial Training for Human Pose Estimation,” *CoRR*, vol. abs/1707.0, 2017. [Online]. Available: http://arxiv.org/abs/1707.02439

[81] W. Yang, S. Li, W. Ouyang, H. Li, and X. Wang, “Learning Feature Pyramids for Human Pose Estimation,” *CoRR*, vol. abs/1708.0, 2017. [Online]. Available: http://arxiv.org/abs/1708.01101

[82] L. Ke, M.-C. Chang, H. Qi, and S. Lyu, “Multi-Scale Structure-Aware Network for Human Pose Estimation,” *CoRR*, vol. abs/1803.0, 2018. [Online]. Available: http://arxiv.org/abs/1803.09894

[83] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, “Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields,” *CoRR*, vol. abs/1611.0, 2016. [Online]. Available: http://arxiv.org/abs/1611.08050

[84] H. Fang, S. Xie, and C. Lu, “RMPE: Regional Multi-person Pose Estimation,” *CoRR*, vol. abs/1612.00137, 2016. [Online]. Available: http://arxiv.org/abs/1612.00137

[85] S. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, “Convolutional Pose Machines,” *CoRR*, vol. abs/1602.00134, 2016. [Online]. Available: http://arxiv.org/abs/1602.00134

[86] Xiangyu Zhu, Yingying Jiang and Z. Luo, “Multi-Person Pose Estimation for PoseTrack with Enhanced Part Affinity Fields,” 2017. [Online]. Available: https://doi.org/10.1109/CVPR.2017.143

[87] Y. Shi, Y. Tian, Y. Wang, and T. Huang, “Sequential Deep Trajectory Descriptor for Action Recognition with Three-stream CNN,” *CoRR*, vol. abs/1609.03056, 2016. [Online]. Available: http://arxiv.org/abs/1609.03056

[88] Apple. (2017, sep) The future is here: iPhone X. [Online]. Available: https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/

[89] NVIDIA Corporation. (c2018) Hardware For Every Situation. [Accessed: 2018-06-07]. [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware

[90] Y. Katariya. (2017) Transfer Learning. [Online]. Available: https://yashk2810.github.io/Transfer-Learning/

A

Appendix 1

A.1 Hand detection

A.1.1 VIVA dataset predictions

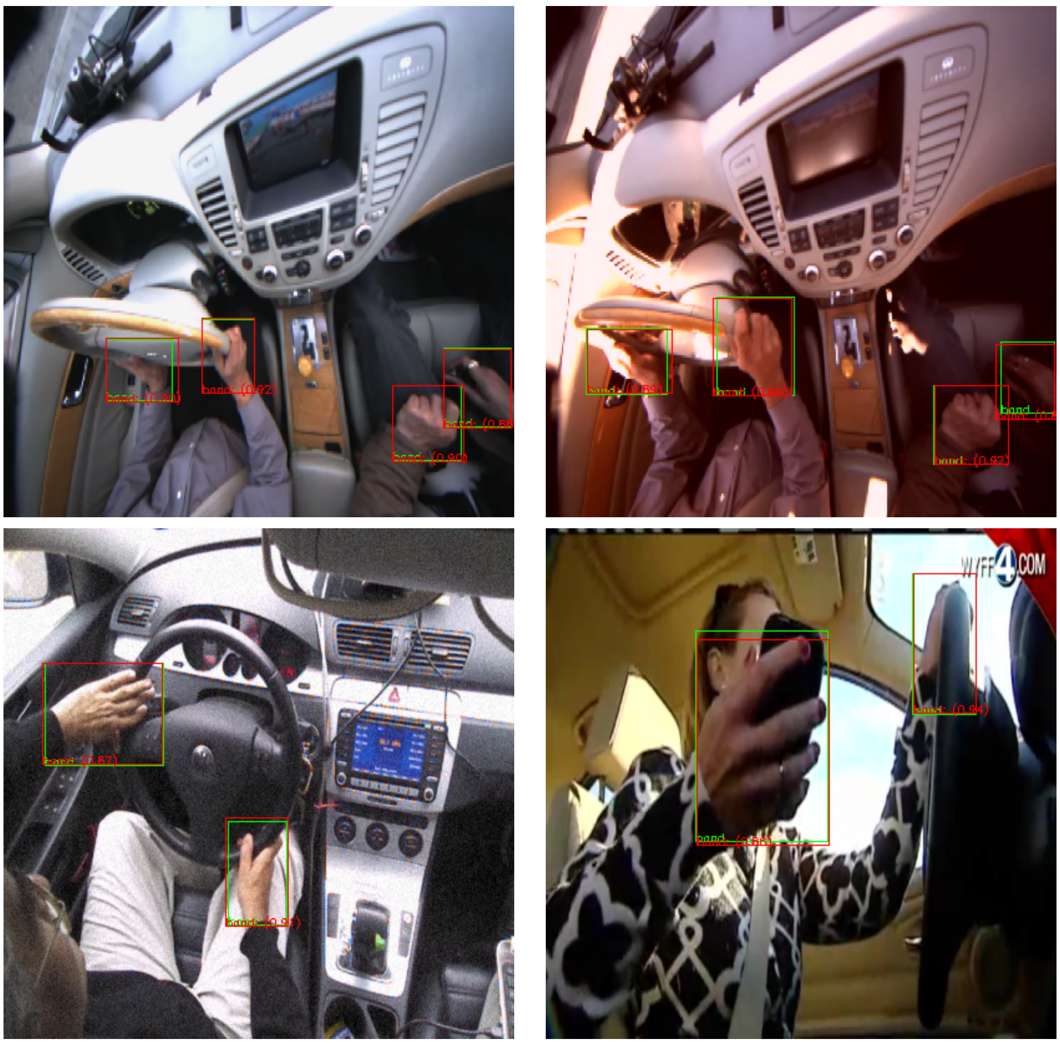


Figure A.1: Some successful predictions for hand detection on the VIVA Challenge dataset.

A.1.2 EgoHands dataset predictions

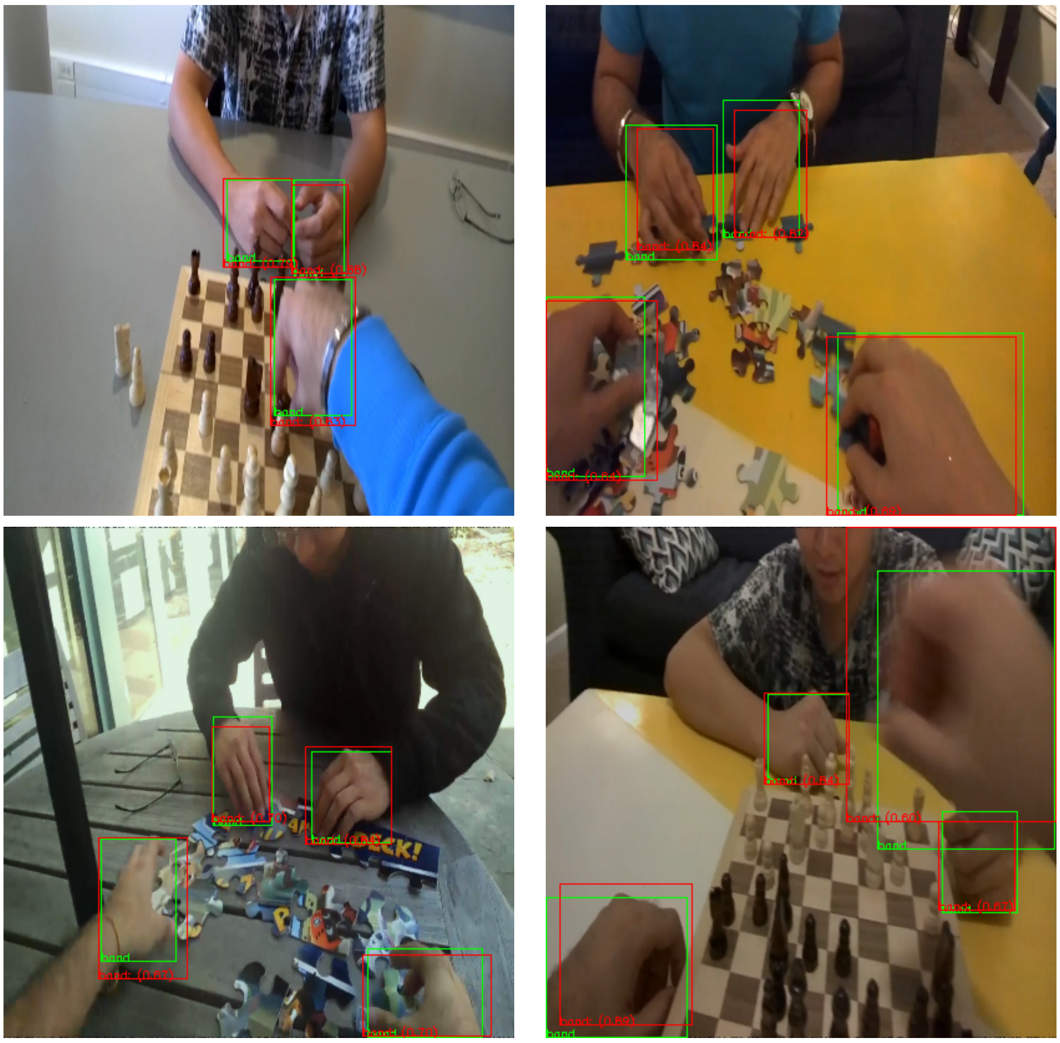


Figure A.2: Some successful predictions for hand detection on the EgoHands dataset.

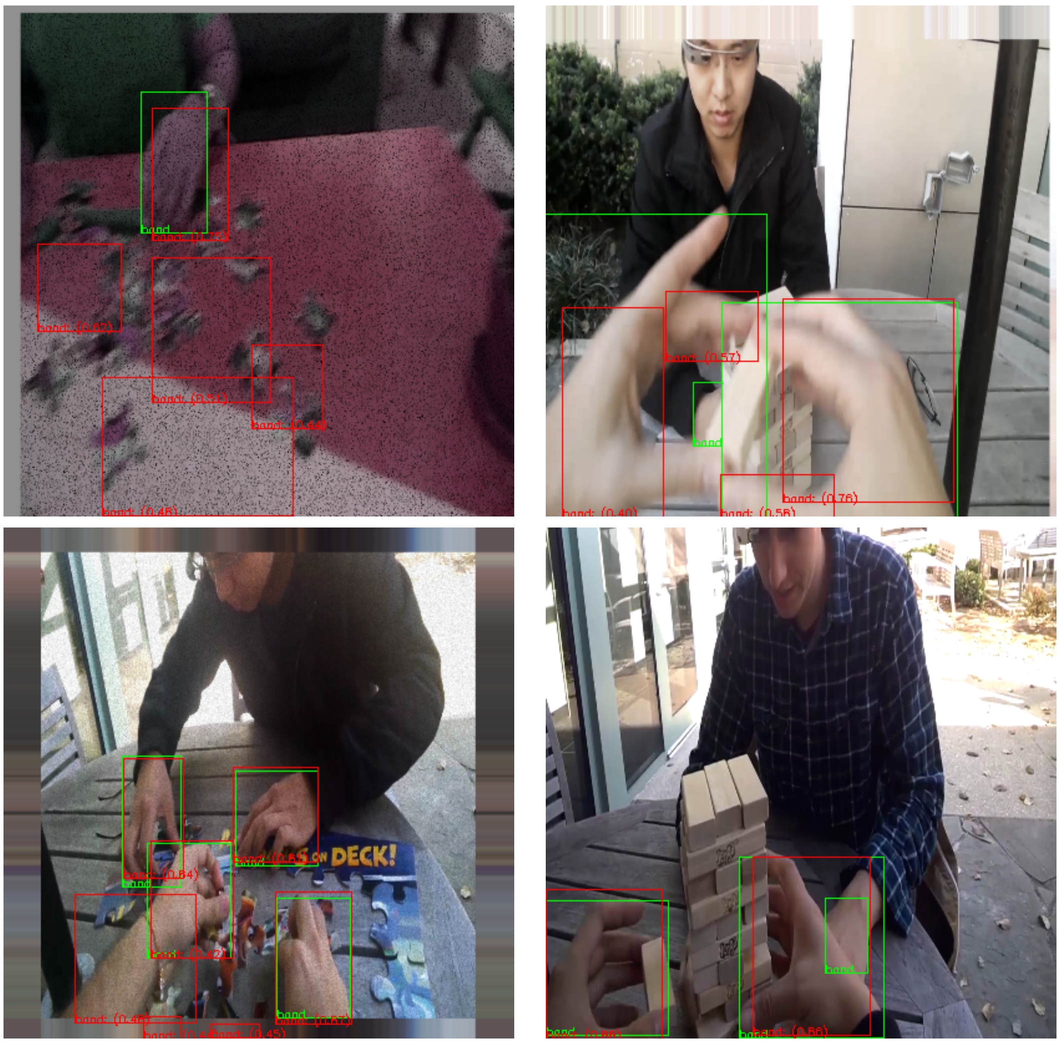


Figure A.3: Some unsuccessful predictions for hand detection on the EgoHands dataset.

A.2 Human pose predictions

A.2.1 COCO dataset predictions

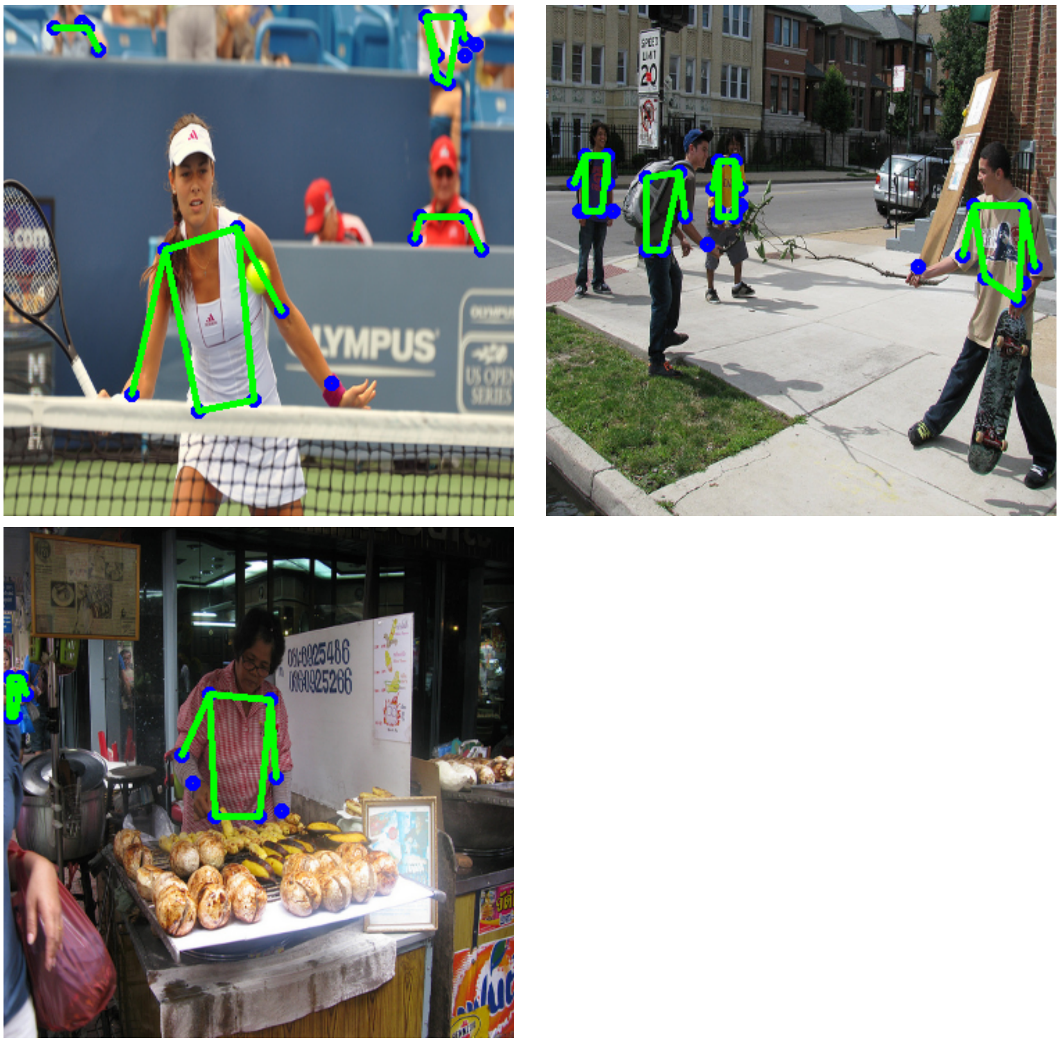


Figure A.4: Some successful predictions for body pose estimation on COCO dataset.

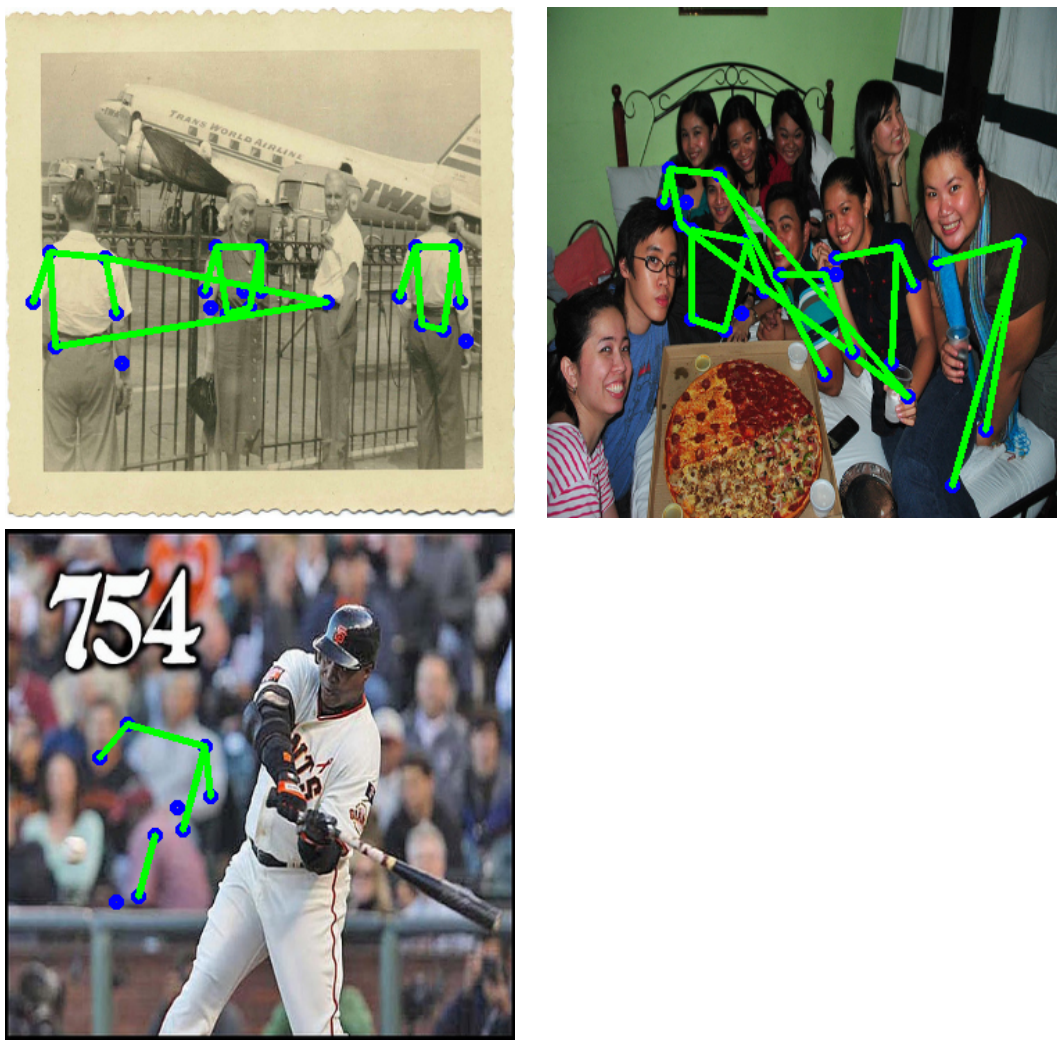
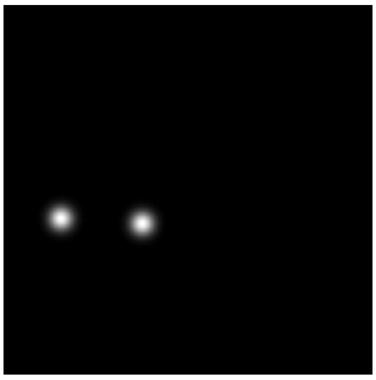


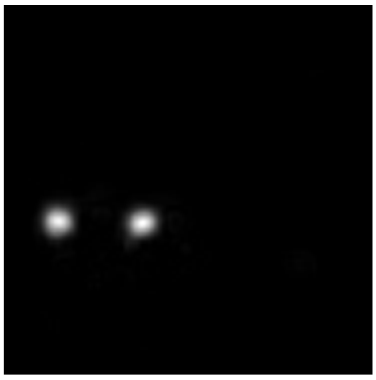
Figure A.5: Some unsuccessful predictions for body pose estimation on COCO dataset.

A.2.2 Keypoint heatmaps and affinity fields predictions

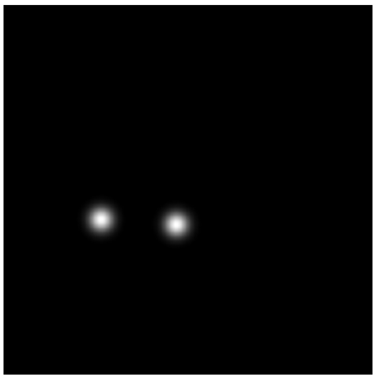
This section shows a sample of keypoint heat map and part-affinity fields predictions.

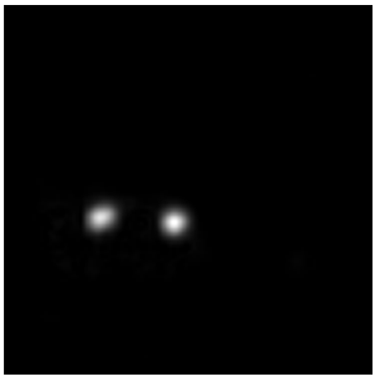
a) GT image


b) Left shoulder GT heatmap

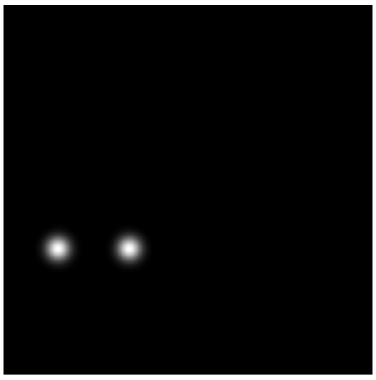

c) Left shoulder predicted heatmap

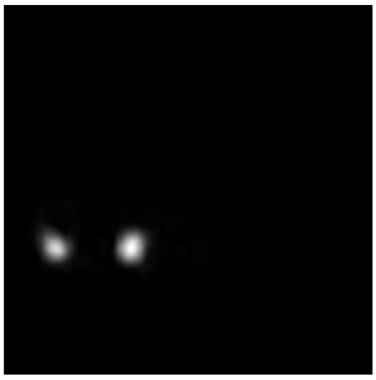
a) GT image


b) Right shoulder GT heatmap

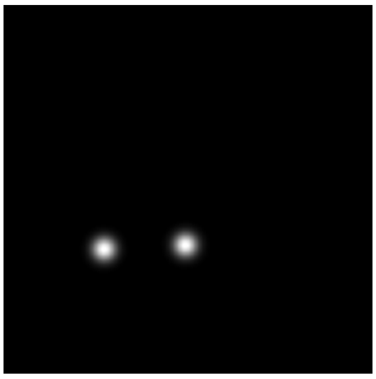

c) Right shoulder predicted heatmap

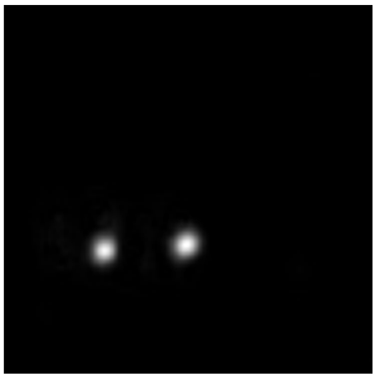
a) GT image

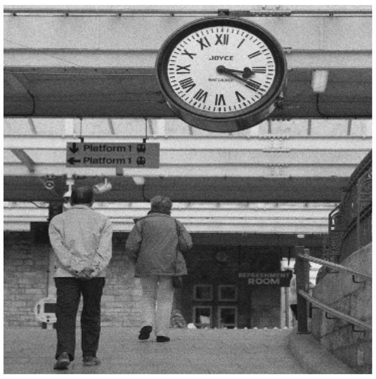

b) Left elbow GT heatmap


c) Left elbow predicted heatmap

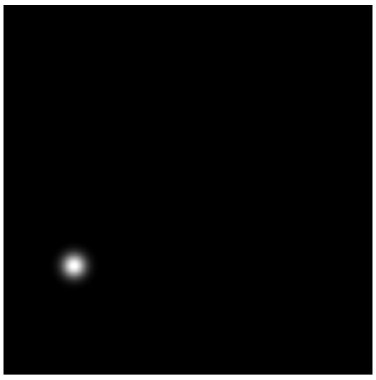
a) GT image


b) Right elbow GT heatmap

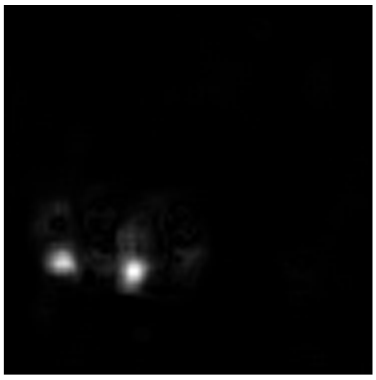

c) Right elbow predicted heatmap



a) GT image

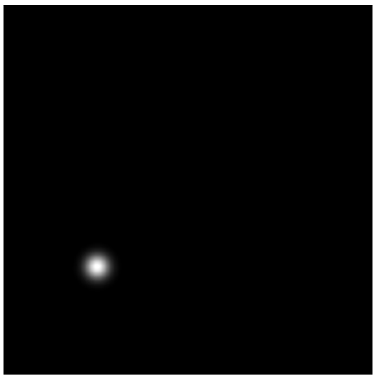


b) Left wrist GT heatmap

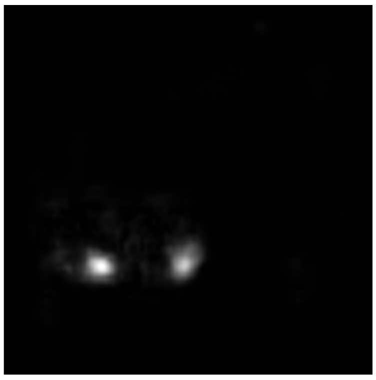


c) Left wrist predicted heatmap

a) GT image

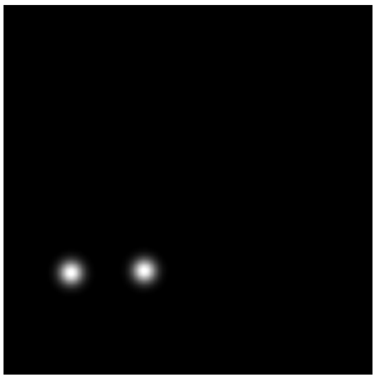


b) Right wrist GT heatmap

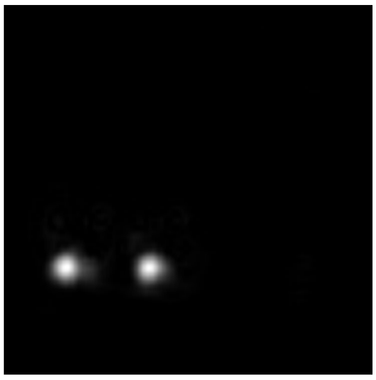


c) Right wrist predicted heatmap

a) GT image

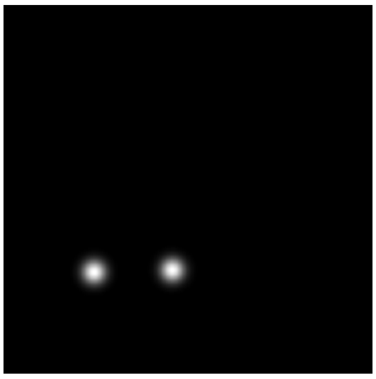


b) Left hip GT heatmap

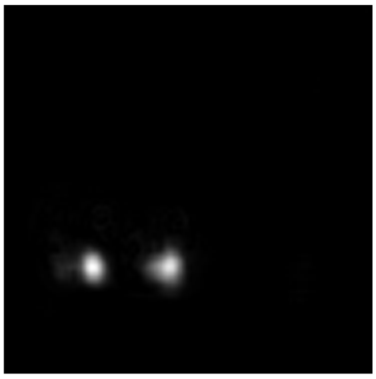


c) Left hip predicted heatmap

a) GT image

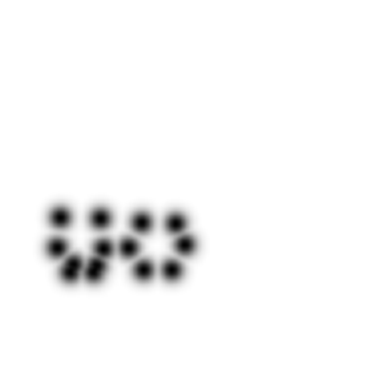


b) Right hip GT heatmap



c) Right hip predicted heatmap

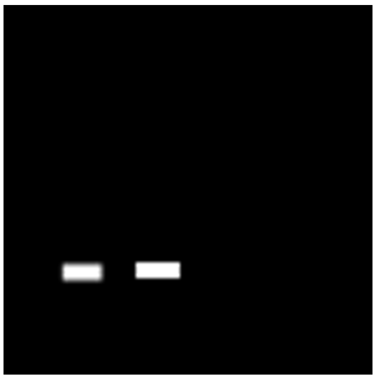
a) GT image



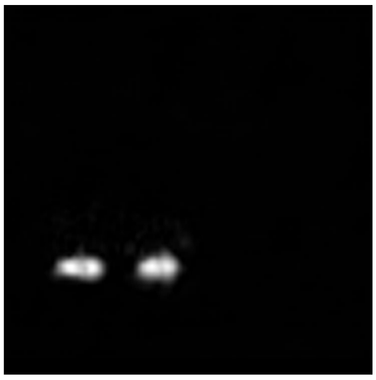
b) Background GT heatmap

c) Background predicted heatmap

a) GT image

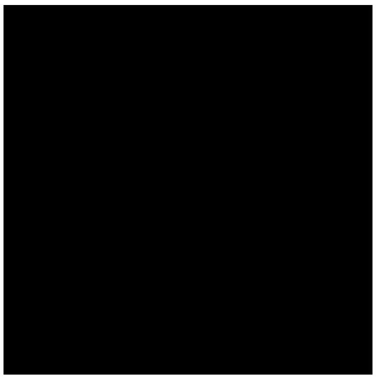


b) 1st GT affinity field x-vector

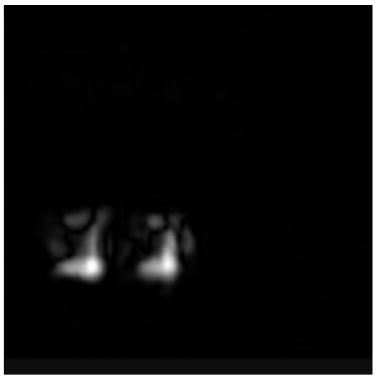


c) 1st predicted affinity field x-vector

a) GT image

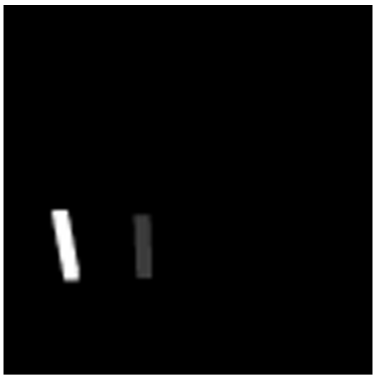


b) 1st GT affinity field y-vector

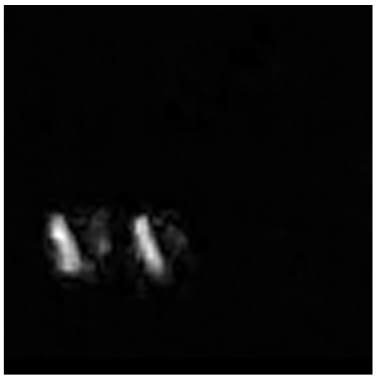


c) 1st predicted affinity field y-vector

a) GT image

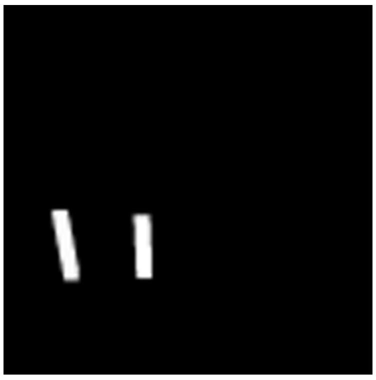


b) 2nd GT affinity field x-vector

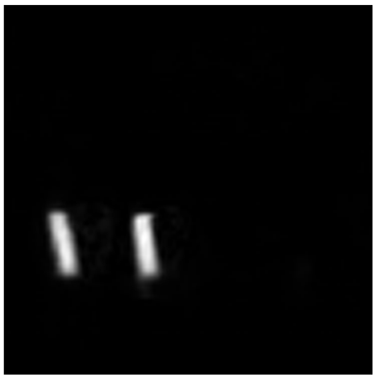


c) 2nd predicted affinity field x-vector

a) GT image

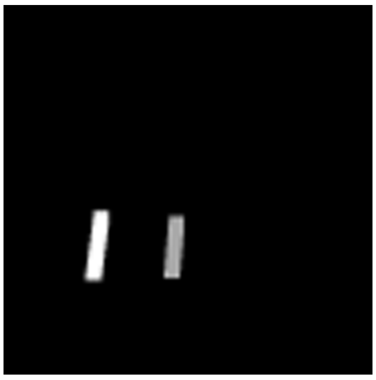


b) 2nd GT affinity field y-vector

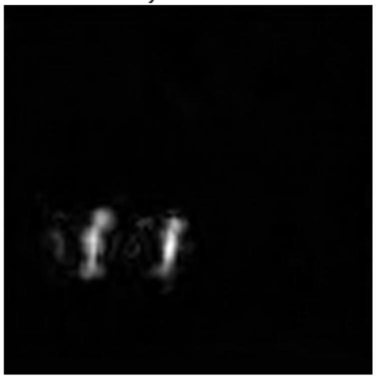


c) 2nd predicted affinity field y-vector

a) GT image

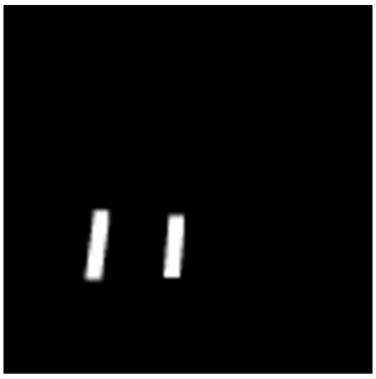


b) 3rd GT affinity field x-vector

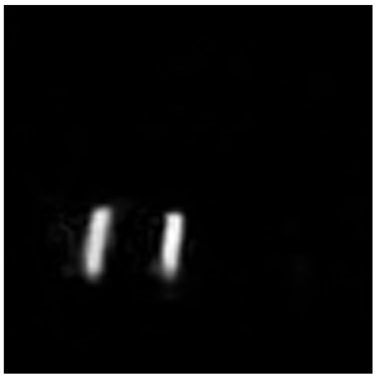


c) 3rd predicted affinity field x-vector

a) GT image

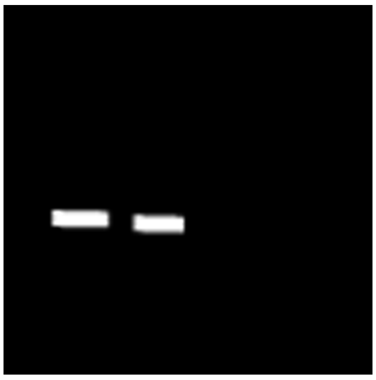


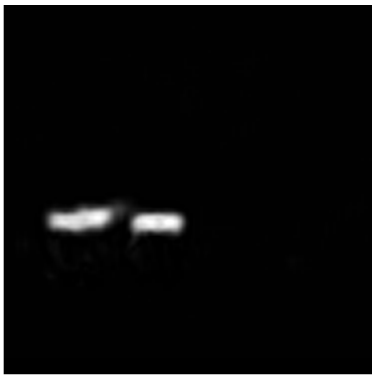
b) 3rd GT affinity field y-vector



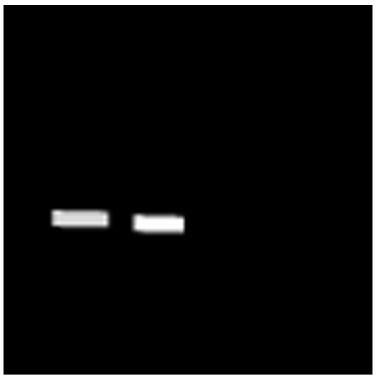
c) 3rd predicted affinity field y-vector

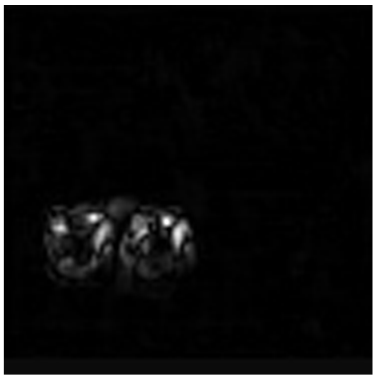
a) GT image


b) 4th GT affinity field x-vector

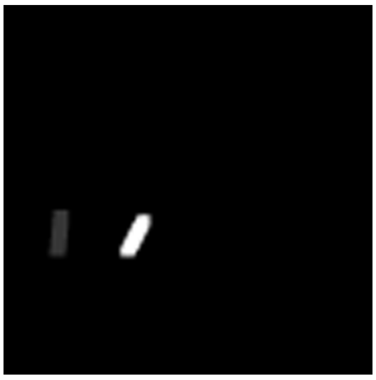

c) 4th predicted affinity field x-vector

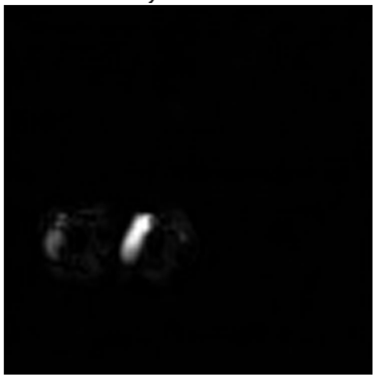
a) GT image


b) 4th GT affinity field y-vector

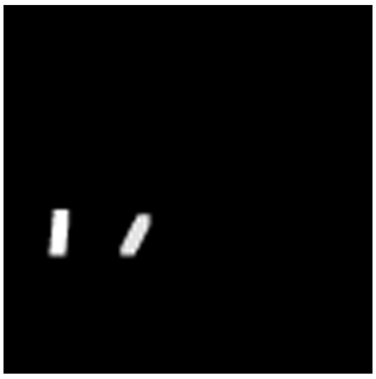

c) 4th predicted affinity field y-vector

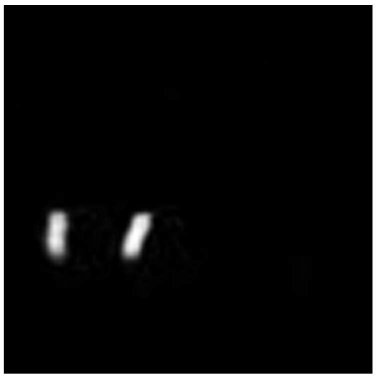
a) GT image


b) 5th GT affinity field x-vector

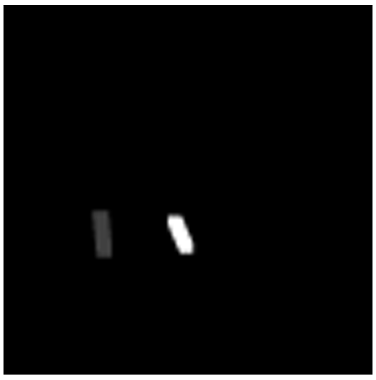

c) 5th predicted affinity field x-vector

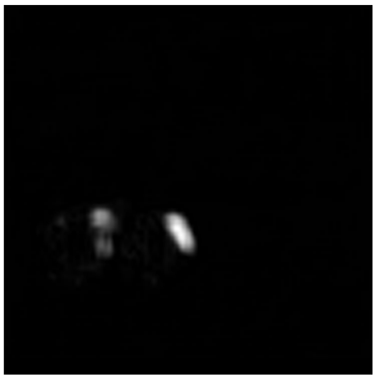
a) GT image


b) 5th GT affinity field y-vector

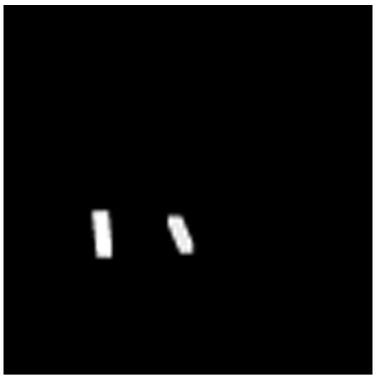

c) 5th predicted affinity field y-vector

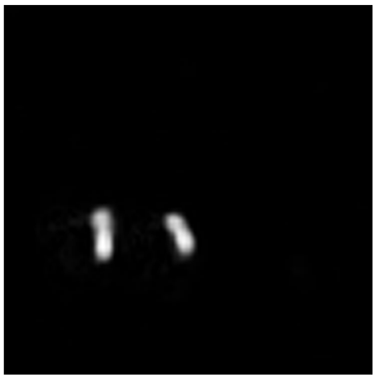
a) GT image


b) 6th GT affinity field x-vector

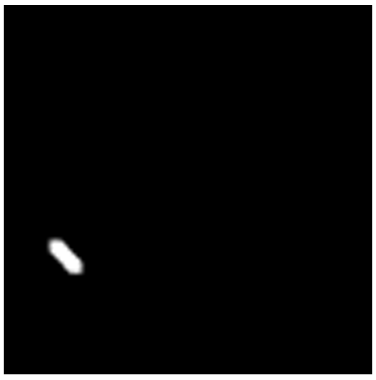

c) 6th predicted affinity field x-vector

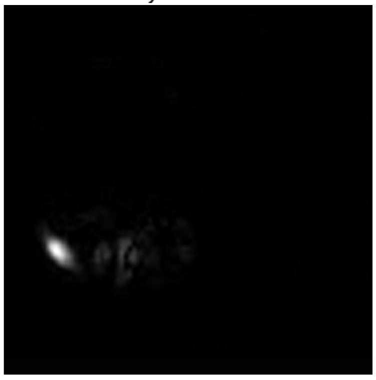
a) GT image


b) 6th GT affinity field y-vector

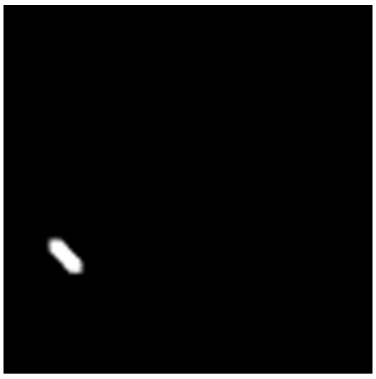

c) 6th predicted affinity field y-vector

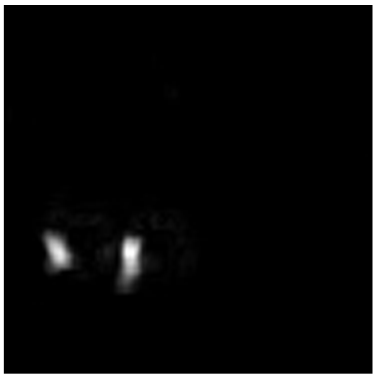
a) GT image


b) 7th GT affinity field x-vector

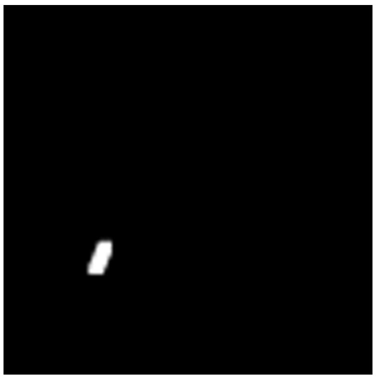

c) 7th predicted affinity field x-vector

a) GT image

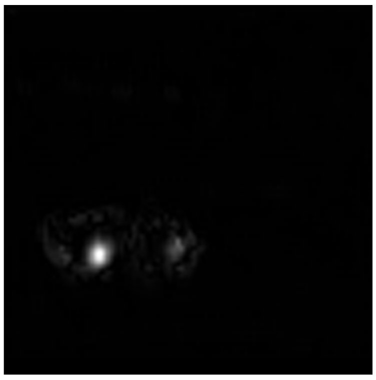

b) 7th GT affinity field y-vector


c) 7th predicted affinity field y-vector

a) GT image

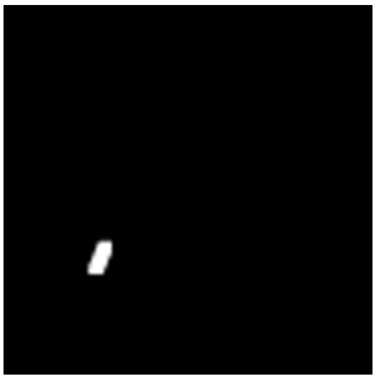


b) 8th GT affinity field x-vector

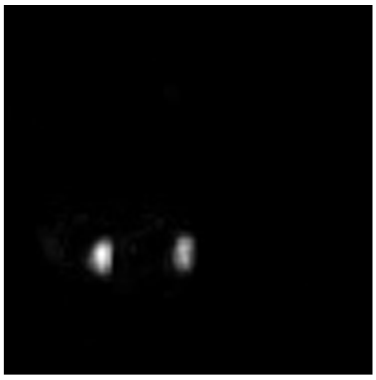


c) 8th predicted affinity field x-vector

a) GT image



b) 8th GT affinity field y-vector



c) 8th predicted affinity field y-vector