Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

Activity Recognition in Human Robot Collaboration through Object Detection

Masterthesis

Frank Baustert

18.07.2018

Advisor:
Christian Bürckert

Examiners:
Prof. Dr. rer. nat. Dr. h.c. mult. Wolfgang Wahlster
Dr. Tim Schwartz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum / Date) (Unterschrift/ Signature)

Abstract

Human activity recognition is a widespread research field which aims to recognize the actions and intentions of humans. Over the last few years vision-based, deep learning approaches have become a popular way to tackle this problem by extracting information from videos, which can be seen as temporal sequences of frames. In a practical activity recognition scenario, it is important to recognize these activities with a low latency. Thus, one has to decide whether to spend more computational time on extracting information from single frames at the cost of processing them at a lower frequency, or to use information from more frames, which is possibly less accurate. This thesis investigates this tradeoff between the spatial and temporal dimension. For this purpose, a guideline on how this tradeoff can be determined is provided. The steps from this guideline are elaborated on the new dataset EgoBaxter created for this thesis, using convolutional neural networks to cover the spatial dimension and recurrent neural networks to cover the temporal dimension. The results show that focusing on the spatial dimension leads to better overall results.

Contents

1 Introduction

Many intelligent systems that assist or interact with humans require to understand human behaviour. One widespread and active research field investigating this topic is Human Activity Recognition (HAR). HAR is employed in many different applications such as assisted living [24] [92], video surveillance [96] [46], intelligent driving [8] or Human Robot Collaboration (HRC) [50].

The research on HAR can be divided into two main categories: vision-based HAR and sensor-based HAR [95]. Vision-based HAR uses images or videos from a camera as its main source of input, whereas sensor-based HAR uses motion information from sensors such as accelerometers or gyroscopes. Both of these categories can further be split into two types: handcrafted methods and (deep) learning-based methods, the latter having become more and more popular over the last few years. One of the reasons for this is the big success of a learning-based approach over handcrafted methods in the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2012 [71]. Furthermore, the advances in hardware, especially GPUs, have made learning-based approaches more feasible.

This thesis focuses on vision-based, deep learning HAR approaches. In this type of HAR, activities are usually recognized from videos, which can be seen as sequences of frames. Videos can encode information in two main dimensions: the spatial dimension, being the 2-D frames, encodes what is perceived and where it is located, and the temporal dimension, which encodes how these entities change over time. For each of the two dimensions there is a well suited deep neural network type. The spatial dimension is best covered by Convolutional Neural Networks (CNNs), whereas Recurrent Neural Networks (RNNs) are commonly used for the temporal dimension. Over the last few years, a lot of different CNN architectures have emerged. In general, the networks that achieve better results are more complex in structure, as they have for example more layers, which results in higher computational costs. This leads to a tradeoff when combining CNNs with RNNs for activity recognition from videos in practice. One could either devote more computational time on the spatial dimension by using a more accurate, but slower CNN, thus having less information about temporal changes. Or one could use a less precise, but faster CNN in order to invest more resources in the temporal dimension. For example, one CNN could evaluate a single frame with 90% accuracy, but can only do so at a frame rate of 2 frames per second (fps). Another CNN could extract spatial information at a frame rate of 10 fps, but can only extract this information at a precision of 50%. When combining these CNNs with a RNN which one will achieve the better overall performance? Using a slower, but more precise CNN (CNN 1 Figure 1.1) or by sacrificing spatial accuracy for more temporal information (CNN 3)? Maybe it is even a point in between which works best (CNN 2). This thesis aims to answer these questions, by determining the tradeoff between the spatial and temporal dimension in a HRC scenario.

The HRC scenario considered in this thesis is a Baxter robot which is working together with a human agent. From time to time, the human worker needs a tool and will request Baxter to hand it to him. If the tool is no longer needed he will give it back to the robot. Baxter will be taught to recognize these activities from its own point of view. Furthermore, in order to not only recognize that the worker is handing the robot a tool, but to also know where to grab it, object detection will be employed to detect and localize the tool at the right position.

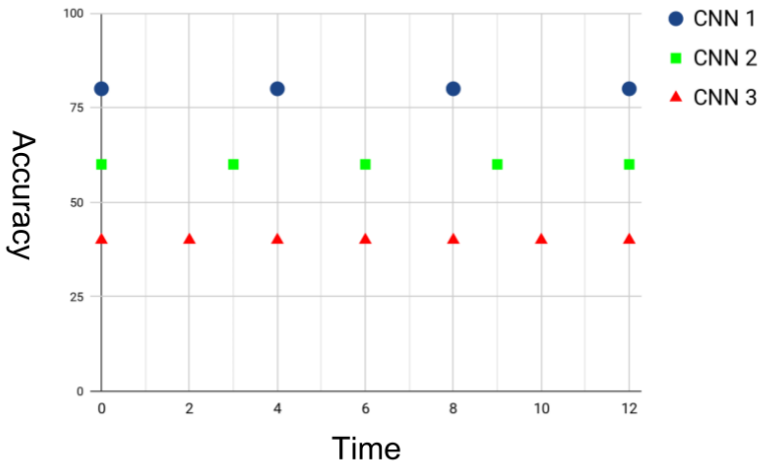


Figure 1.1: Different CNNs with different accuracies and computational speeds.

1.1 Motivation

Most practical HAR applications need to work in real-time for them to be useful. In case of a HRC scenario, this is relevant for two major reasons. First, when a robot and a human work together it is of utmost importance that the human agent is not endangered by the robot. For example, if a moving robot would recognize its interacting partner's intentions with a delay of 10 seconds, it might have caused an accident in the meantime. Second, in order to ensure a smooth workflow between a human and a robot, the robot needs to be reactive. If the robot needs a minute to answer to anything the human agent does then the point of collaboration becomes ineffective.

The term of *real-time* is a very vague one. Even though real-time describes a latency, it is often expressed in fps, e.g. a latency of 500ms is expressed by a system processing one frame at a frame rate of 2 fps. Some refer to real-time as a frame rate of at least 25 fps. Others claim it needs to be at least 30 or even 60 fps to be considered real-time. In order to have a common foundation, the term of *weak real-time* will be introduced here. An algorithm is considered to work in *weak real-time* if it works with a frame rate of at least 1 fps.

This restricition on the computational time limits the resources one can spend, and thus one needs to decide whether to invest more into the spatial dimension or the temporal dimension. This tradeoff between the two dimensions, hereafter referred to

as SpatioTemporal Tradeoff, is essential to know in order to achieve the best possible accuracy when recognizing activities.

The motivation for using object detection in the convolutional neural networks is two-fold. First, in order to be used in practice, it does not suffice for Baxter to recognize what is happening, the robot also needs to act according to the current activity. In case of a handover, it needs to know where to place or take the tool from. Second, the idea of not only knowing what is being perceived on a frame, but also knowing where it is located might potentially lead to an increase in performance in recognizing human activities.

1.2 Goals

Determining the SpatioTemporal Tradeoff

The main goal of this thesis is to determine the SpatioTemporal Tradeoff. When combining a CNN and a RNN for a weak real-time application, on which of both components should one focus more? Using higher temporal resolution at the cost of less accurate spatial information or vice versa? This thesis provides a guideline on how the SpatioTemporal Tradeoff can be determined, such that it can be reconstructed for any other dataset. Results are then provided for the newly introduced dataset EgoBaxter.

Other contributions of this thesis include:

- **The novel dataset EgoBaxter**
  To determine the SpatioTemporal Tradeoff a HAR dataset is needed. For this purpose the dataset EgoBaxter is created during the process of this thesis. This dataset contains scenes of a handover between a human agent and a Baxter robot from the robot's point of view.

- **A practical HAR application for a HRC scenario**
  As a side result of determining the tradeoff, a practical HAR application for the EgoBaxter dataset is designed. This application fulfills two important conditions in order to be considered practical: 1) it works in weak real-time, and 2) it works on ongoing sequences. The second condition is implicitly given by working in weak real-time, however, one issue with many approaches in the literature, which makes them unviable in practice, is that they use *after the fact* recognition. That is, they first record the complete video sequence before classifying it. There they also make use of information from e.g. frame *t+10* at frame t which results in a look ahead of information which is a priori unknown.

1.3 Outline

The outline of this thesis is as follows: Chapter 2 provides background information about the deep learning methods employed in this thesis. This includes convolutional neural networks, different object detection algorithms as well as recurrent neural networks. Chapter 3 gives an overview of existing works in the field of vision-based, deep learning HAR. Chapter 4 describes the deep learning architecture on which the SpatioTemporal Tradeoff is determined. It provides the reasons behind the architecturial choice and a comparison to the architectures presented in Chapter 3. Furthermore, the EgoBaxter dataset and a guideline on how to determine the SpatioTemporal Tradeoff are presented. Chapter 5 covers the implementational details from Chapter 4 and Chapter 6 presents the evaluations. Finally, the conclusion and ideas about future work are provided in Chapter 7 and Chapter 8.

2 Background - Artificial Neural Networks

Before diving into the related work and the thesis itself, it is necessary to understand certain concepts about neural networks, their structure, how they work, and what their strengths and limitations are. These points will be covered in this chapter.

Artificial neural networks (ANN) are inspired by the human nervous system. The main building block of the system consists of neurons which are connected to each other via synapses. A neural network is then formed by a group of interconnected neurons. One way such a network learns is by altering the strengths of connections between neurons, and by adding, respectively deleting such connections [59] [33]. Similar to the neural networks present in the nervous system, an ANN consists of an interconnected group of artificial neurons, also referred to as nodes. The connections between the nodes can be seen as the counterpart to the synapses, each having a weight assigned to it which can be adjusted during training.

Artificial neural networks are used for many different tasks and over the years a broad variety of networks has emerged. One particular characteristic of ANNs is that they learn from examples without being told any rules. Hereafter, the two main variants of networks appearing in this thesis will be presented, namely convolutional neural networks and recurrent neural networks.

2.1 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) [25] [49], also often referred to as convnets, are a type of feedforward neural networks and are commonly used in the visual domain. In feedforward networks the artificial neurons are grouped into layers. Information is then processed in a forward direction, from the first layer to the last layer, making the output from one layer the input to the next layer. The networks consist of one input and one ouput layer and one or more hidden layers. Neural networks are typically considered *deep* if they have more than one hidden layer. Consequently, the name deep learning comes from employing deep networks. An example is shown in Figure 2.1.

Even though CNNs have been around for quite a few years, they had to wait until 2012 to make their breakthrough. In the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2012 [71], Krizhevsky et al. [47] beat their competitors by more than 10%. They used a CNN whereas the other contestants relied on handcrafted features. Since then CNNs have become the primary method for image classification and related problems.

Generally, in a regular feedforward neural network, a neuron is connected to all the neurons in the previous layer. In case of an image as input, a neuron in the first hidden layer would be connected to all the pixels of each color channel. This would lead to

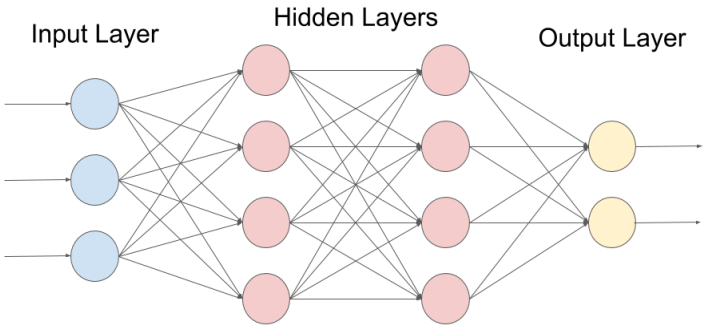


Figure 2.1: Simple example of a feedforward neural network with two hidden layers. It can be seen as a directed graph with no cycles.

$3 * hw$ weights for an h x w RGB image for a single neuron in the first hidden layer. For smaller images this is still feasible, but as the images become larger, the number of parameters explodes. CNNs overcome this problem by using weight sharing in the convolutional layers.

Convolutional Layer

As the name might suggest, convolutional layers are the main building block of convolutional networks. Instead of thinking of the convolutional layer as a set of neurons, it is easier to see them as a set of *filters*. Their role is to learn to detect visual features, which could be edges or corners in the early layers or more complex shapes in the later layers. These filters operate on an input volume. In case of the first convolutional layer, the input volume would be an image, where the depth is the number of color channels. A filter is connected to only a small spatial region, known as the receptive field, but uses the whole depth of its input. The filter is then slid over the whole volume, producing one output value at each position and thus a 2-D set of values, often referred to as *feature map*. As usually multiple filters are applied, the output of a convolutional layer is a 3-D volume as can be seen in Figure 2.2, where the depth is the number of filters.

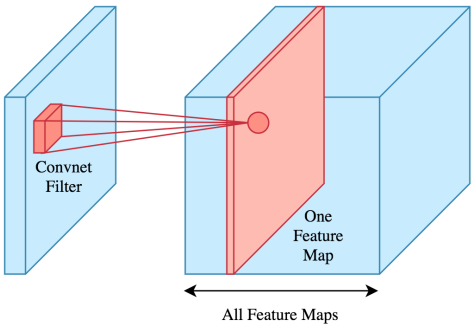


Figure 2.2: Computation of feature maps by a convolutional layer. Image from[1].

[1] `https://brilliant.org/wiki/convolutional-neural-network/` (last accessed on 16.06.2018)

In regular feedforward networks, each value in a 2-D feature map would require one neuron. For a m x n map this would result in $mn * filtersize$ weights. However, in CNNs, the same filter is used to calculate one feature map. In this way, the parameters are shared and only the weights of the filter are needed. Using the same filter at different spatial locations also allows for translation invariance, meaning a feature will be detected regardless of its position in an image. The convolutional layer has a few adjustable hyperparameters which are the number of filters, the size of the receptive field, its stride and its padding.

On top of these feature maps an element-wise non-linear function, often called *activation function*, is applied. In CNNS this is typically the Rectified Linear Unit (ReLU) [29] which is defined as $f(x) = max(0, x)$. Without the non-linearity, neural networks could only calculate linear functions and there would be no purpose in stacking multiple layers.

Pooling Layer

The second important type of layer in CNNs is the pooling layer. It is peridocially inserted between two convolutional layers. The pooling layer is applied to each feature map and its function is to reduce the spatial dimensionality, thus reducing the number of parameters. This also indirectly increases the size of the receptive field in the next convolutional layer. Furthermore, it adds non-linearity and stabilizes against noise. The most common pooling method is *max pooling* with a non-overlapping 2x2 filter size. An example can be seen in Figure 2.3. The hyperparameters of this layer are the size and the stride of the pooling window. CNNs without pooling layers have also been proposed [82].

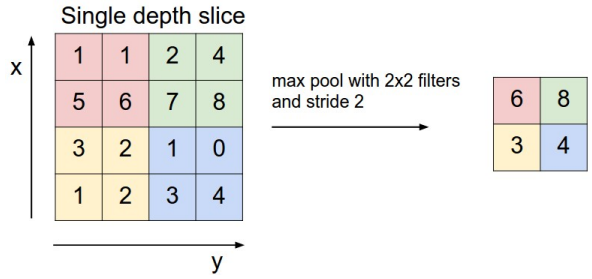


Figure 2.3: Example of non-overlapping max pooling with filter size 2x2. Graphic by Andrej Karpathy[2]

Fully-Connected Layer

Lastly, after multiple convolutional and pooling layers, one or more fully-connected layers are added. As in regular feedforward networks, neurons in a fully-connected layer take input from all the activations of the previous layer. For a classification problem the softmax function is typically applied in the output layer. This function outputs a probability distribution among the different classes to classify.

[2] `http://cs231n.github.io/convolutional-networks/` (last accessed on 17.06.2018)

In the fully-connected layer only the number of neurons is an adjustable parameter. In addition to all the hyperparameters, the number and ordering of the layers is another architectural choice which can be modified.

Learning

Image classification is a supervised learning problem where each image has a class label, also called ground truth, assigned to it. During the training, the inputs are first propagated through the network producing an output value. This value is then compared to the ground truth of the input, and a loss, indicating how close the prediction is to the ground truth, is calculated. Using backpropagation [70], this loss is then propagated backwards through the network and the weights are updated, usually by employing a form of gradient descent.

Applications

Over the last few years, convolutional neural networks have become the predominant method for visual recognition. They are used in many domains including image classification [47], image segmentation [45] [4] and video classification [44]. Another application is object detection which will be used in this thesis and is covered in the next section.

2.2 Object Detection

One application field of CNNs is object detection. In addition to classification, object detection also deals with localizing, possibly multiple, instances of objects in an image. This section gives a brief introduction to the object detection algorithms employed in this thesis.

2.2.1 Faster Region-CNN (Faster R-CNN)

Faster R-CNN [68] is the last part of the Region-CNN (R-CNN) trilogy, its predecessors being R-CNN [28] and Fast R-CNN [27]. It is easier to understand how Faster R-CNN works, when understanding how its ancestors work. Thus, starting in chronological order:

R-CNN

R-CNN [28] is based on three steps:

1. Extract region proposals where possible objects of interest could be located using Selective Search [90].

2. Compute features for each region using a CNN.

3. Classify each region by inputting the extracted features to class-specific Support Vector Machines (SVMs) [14].

One of the main problems of this approach is that it is very slow during testing time. The reason behind this is that for each proposed region, there is an entire forward pass through the CNN. This issue was tackled in the next work.

Fast R-CNN

The basic architecture behind Fast R-CNN [27] is the same as its predecessor's. First regions are proposed using Selective Search, then these regions are classified. However, there are two major changes which improve this model:

1. In a first step, the whole image is processed by the CNN, producing a feature map. Then, for each Region of Interest (RoI) the corresponding part is extracted from the feature map using a RoI pooling layer. This shares the computation of the CNN pass for all region proposals, instead of doing a seperate pass for each of them.

2. The second change is that the SVMs are replaced with fully-connected layers, allowing end-to-end training, contrary to the previous multi-step training (first training/fine-tuning the CNN, then training the classifiers). In the end, the output of these layers are a class and a bounding box for each region proposal.

Now the aquisition of the region proposals became the computational bottleneck of the whole process. This part was improved with the third work.

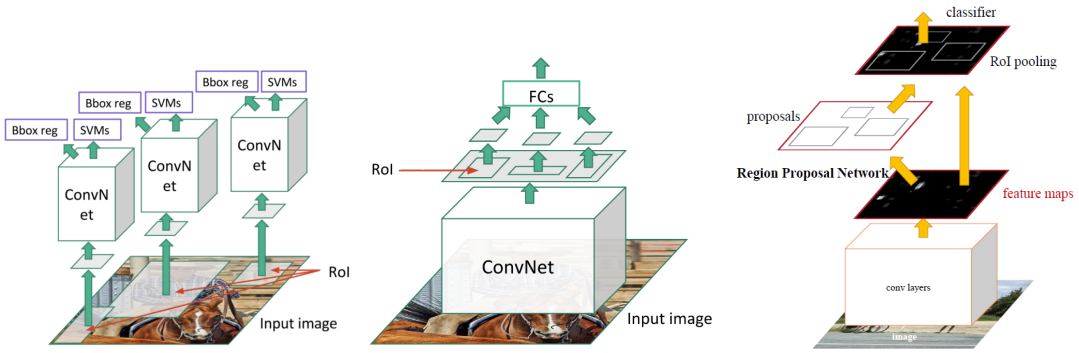
Faster R-CNN

The authors added a so called Region Proposal Network (RPN) to their approach to create region proposals instead of using Selective Search. As in Fast R-CNN, the whole image is processed by the CNN creating a feature map. The RPN makes use of these features as follows: first a small sliding window (3x3) is run over the feature map. At each location of the sliding window, several region proposals are made by so called *anchors*. These anchors are centered at the sliding window and have different scales and aspect ratios. For each of these anchors the likelihood whether it contains an object and its box coordinates are being output. Once the region proposals are obtained, the rest is basically as in Fast R-CNN.

How much the runtime improved with each new model can be found in Table 2.1. A graphical overview of the three methods can be found in Figure 2.4.

	Test Time per Image
R-CNN	50s
Fast R-CNN	2s
Faster R-CNN	0.2s

Table 2.1: Comparison of the different R-CNN models. The underlying CNN is the VGG16 network [79]. The results were produced using a Nvidia K40 GPU.



(a) R-CNN (b) Fast R-CNN (c) Faster R-CNN

Figure 2.4: Comparison of the R-CNN architectures. Credits to Ross Girshick for the diagrams.

2.2.2 Region-based Fully Convolutional Network (R-FCN)

Region-based Fully Convolutional Network (R-FCN) [15] is based on the Faster R-CNN architecture. The authors also employ the two-step architecture of first calculating region proposals and then classifying them. Just like from R-CNN to Fast R-CNN, where the runtime was improved by sharing computation, R-FCN aims to achieve the same by sharing even more computation. Specifically, they want to reduce the work needed for each RoI.

The authors employ Fully Convolutional Networks (FCN) [55], namely ResNet-101 [32]. Just like in Faster R-CNN, they first compute a feature map by running the whole image through the CNN. On this feature map they add one more convolutional layer which produces $k^2(C+1)$ *position-sensitive score maps*, where k^2 represents a k x k grid of relative positions. In their work, they set $k = 3$ leading to the relative positions *top-left, top-center, . . . , bottom-center, bottom-right*. For each of the relative positions one score map for each class, and one for the background (no-class object) are produced, leading to $C+1$ maps per position. The RoIs are calculated the same way as in Faster R-CNN, using an RPN. Each RoI is then divided into the same k^2 subregions as the produced score maps. Each of these RoI subregions is then pooled with its respective score map to check if it matches the corresponding relative position of an object. *"Does the top-left part of the RoI look like the top-left part of the object? Does the top-center part of the RoI look like the top-center part of the object?"* and so on. The k^2 position-sensitive scores are then averaged, producing a single output value per RoI and class. Thus, for each RoI a $C+1$ dimensional vector is output and the final class decision is obtained using the softmax function. An overview can be found in Figure 2.5.

Taking the example of a face-detector: a face can be split into different parts like the eyes, the nose, the mouth, ... These parts are usually positioned relative to each other; the eyes can be found in the *top-left and top-right* subregions of the face, the nose in the *center* and the mouth in the *bottom-center*. Thus, the *top-left score map* would look for the right eye, the *top-right score map* for the left eye and so on. If enough of these subregions match, then the total score for this class would be very high. Hence, if the eyes, nose and mouth are detected in their corresponding positions, then that RoI

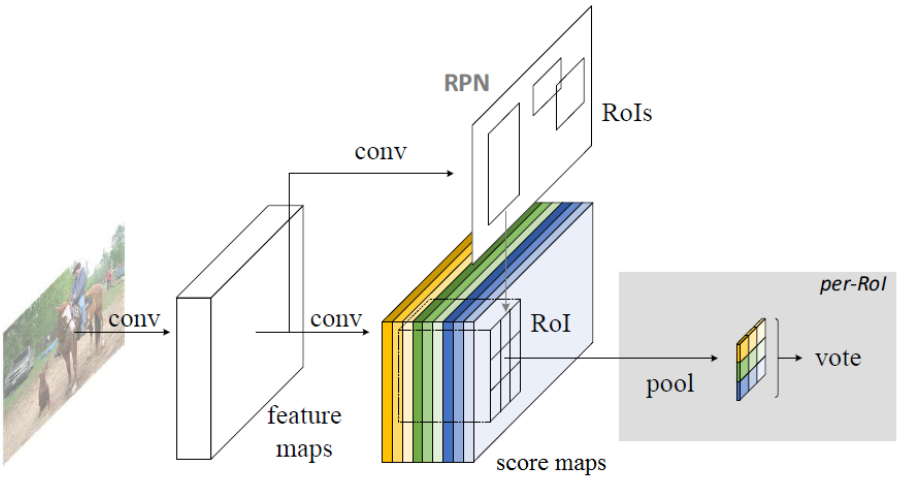


Figure 2.5: R-FCN architecture.

will most likely be classified as a face. A visualization using the example of a baby can be found in Figure 2.6.

A speed comparison between Faster-RCNN and R-FCN can be found in Table 2.2. The number of shared layers in each step is also included.

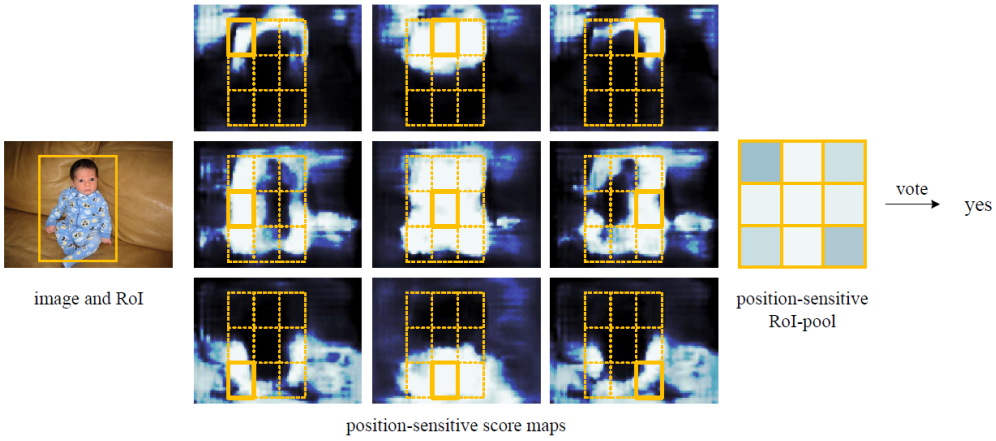
	Faster R-CNN	R-FCN
#shared layers in the conv subnet	91	101
#shared layers in the RoI-wise subnet	10	0
Test Time per Image	0.42s	0.17s

Table 2.2: Comparison between Faster R-CNN and R-FCN using ResNet-101 [32]. The timings are evaluated on a single Nvidia K40 GPU.

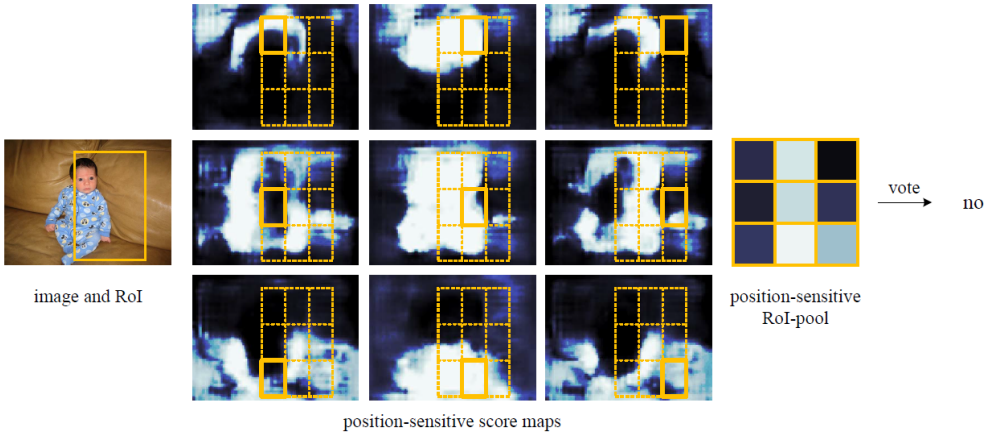
2.2.3 Single Shot MultiBox Detector (SSD)

Contrary to the two previous object detection methods, Single Shot MultiBox Detector (SSD) [54] is not a two-step model. Faster R-CNN and R-FCN are both first proposing RoIs and then classifying them. SSD on the other hand omits the region proposal step and does everything in a "single shot", hence the name. Instead of using region proposals, they use a set of pre-defined default bounding boxes on which they apply small filters to predict a class score and box offsets.

Similar to Faster R-CNN and R-FCN, they start by processing the input image through a CNN, in this case VGG16 [79]. They modify the base network by replacing the last few fully-connected layers with convolutional ones. Using these layers, they produce several feature maps of decreasing size (e.g. 10x10 then 5x5 then 3x3 and so on), allowing for detections at multiple scales. For each feature map, they iterate over all the positions, similar to how a convolutional filter is convolved over an image. Each position has a set of default bounding boxes with different aspect ratios assigned to it, similar to the anchors in Faster R-CNN. For each of these boxes the offset to the object and the per-class score are predicted using small 3x3 kernels. Thus, given a m



(a) Visualization of a positive RoI overlap.

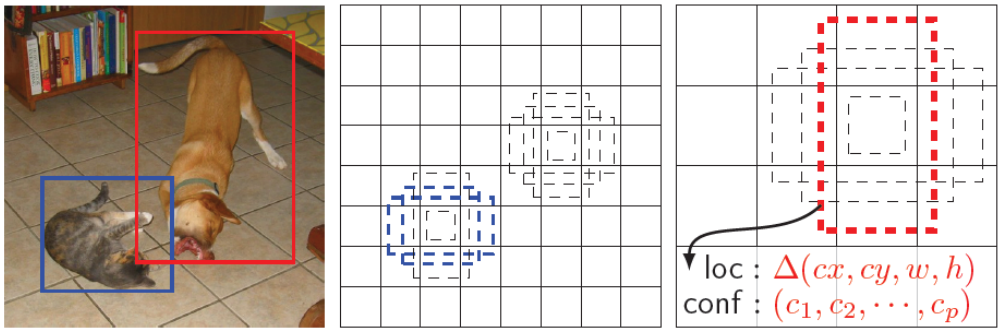


(b) Visualization of a negative RoI overlap.

Figure 2.6: Visualization of the R-FCN algorithm. Top: enough of the subregions are matching for the proposed RoI and it is classified as a baby. Bottom: The RoI does not accurately overlap with the baby and is thus not classified as such.

x n feature map, k default bounding boxes and c classes, there are $(c+4)kmn$ outputs for that map (4 as a bounding box consists of 4 parameters: x, y, width, height). In the end, SSD is not that different from the other methods. It simply skips the step of creating region proposals and instead considers all default bounding boxes at each location at different scales and aspect ratios.

An example can be seen in Figure 2.7. Although the authors use this figure to describe the training process in their original work, it can be used to demonstrate the idea of how the SSD detection works as well. Subfigure (c) shows a 4x4 feature map, each cell represents one position. The dotted lines represent the default bounding boxes of different aspect ratios, in this case 3. These are used in each cell, but for the sake of visibility only shown for one. The parameters at the bottom represent the output for that box: *loc :* $\Delta(c_x, c_y, w, h)$ are the box offset and *conf :* $(c_1, c_2, ..., c_p)$ the per-class score, given p classes. This box would correspond to the boundig box of the dog in subfigure (a). Similarly, one of the blue boxes from subfigure (b) would match the cat in subfigure (a). This example also displays the usage of different feature map scales: in larger feature maps each cell covers a smaller region, allowing to detect smaller objects.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.7: The SSD framework.

A comparison to Faster R-CNN can be found in Table 2.3. Unfortunately, there was no direct comparison between R-FCN and SSD. However, given the common comparison to Faster R-CNN, one can deduce that SSD is also faster than R-FCN, as R-FCN is around 2.5 times faster than Faster R-CNN and SSD is ∼3-8 times faster depending on the resolution of the input image. SSD does not only outperform its competitor in speed, but also in accuracy. One can also see that the higher the input resolution, the larger the feature maps. This implies better detection for smaller objects, but also the usage of more bounding boxes which slows down the speed of the model.

2.2.4 Common Practices in Object Detection

This subsection covers some more insight which is shared among the different object detection methods.

The first is how training data is handled. Even though the methods do not train in the exact same way, they share common concepts. The training data consists of images which are labeled with bounding boxes. These boxes represent the ground truth of an object, meaning its class and location. During training, the object detection algorithms

	mAP*	fps	#boxes	input resolution
Faster R-CNN	73.2	7	∼6000	∼1000x600
SSD300	74.3	59	8732	300x300
SSD512	76.8	22	24564	512x512

* mAP = mean Average Precision, the higher the better

Table 2.3: Comparison of SSD and Faster R-CNN on Pascal VOC 2007 test [19]. The base network is VGG16 and the experiments were conducted on a Nvidia Titan X. Note: the results for SSD here use a batch size of 8, whereas the batch size for Faster R-CNN is 1.

produce several boxes per image, e.g. using an RPN (Faster R-CNN, R-FCN) or by using default bounding boxes (SSD). Boxes with a certain overlap with the ground truth are considered positive examples. The overlap is typically measured by Intersection over Union (IoU), i.e. the ratio between the overlapping area (intersection) and the total area (union) of the box and the ground truth. If this ratio is above a certain threshold, e.g. 0.5, the box is considered a positive example, else it is treated as a negative one. As most of the boxes fall into the latter category, there is a high imbalance between positive and negative training examples. *Hard negative mining* [23] is a way of countering this imbalance: only the negative examples producing the highest loss are used.

After classifying an image and outputting several bounding boxes, it is possible that multiple detections are covering the same object of interest. To overcome this problem and to ensure that only one detection is output for an object *non-maximum suppression* is used. The general idea is to only use the bounding box with the highest score from such a cluster of boxes and discarding respectively suppressing the other ones. A visualization can be found in Figure 2.8. Note that the non-maximum suppression method used in the object detection algorithms differs from the one the authors of [37] employ in the visual example, however the key idea is the same.

Figure 2.8: Example of non-maximum suppression using a convnet [37].

2.3 Recurrent Neural Network (RNN)

Traditional feedforward networks treat each input as an isolated instance. They have no ”memories” of the past. Recurrent Neural Networks (RNNs) [36] overcome this shortcoming by managing a hidden state which depends on previous inputs, enabling them to work with sequences of inputs.

More formally, at time t the hidden state h_t is calculated as follows:

$$h_t = f(W_h h_{t-1} + W_x x_t) \tag{2.1}$$

where h_{t-1} is the previous hidden state, x_t is the input at time t and W_h and W_x are the weights. On top of that is a non-linear activation function f which is typically the hyperbolic tangent function $tanh(x)$ or the sigmoid function $sig(x) = 1/1 + e^{-1}$. The output y_t is then calculated with respect to the hidden state h_t:

$$y_t = f(W_y h_t) \tag{2.2}$$

A visualization can be seen in Figure 2.9.

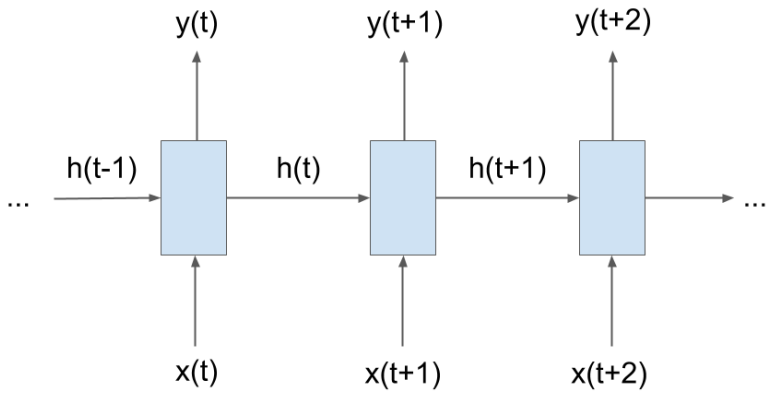


Figure 2.9: Visualization of a RNN.

In practice however, this type of RNN has problems with learning long-term dependencies when using gradient-based learning methods as was shown by Bengio et al. [7]. The problem when propagating the error backwards through many time steps is that the gradient either vanishes or explodes. These are known as the *vanishing* and *exploding gradient problems*.

2.3.1 Long Short-Term Memory (LSTM)

To overcome the aforementioned problems of vanishing and exploding gradients Hochreiter and Schmidhuber introduced Long Short-Term Memory (LSTM) networks [35]. Unlike standard RNNs, LSTMs have a *memory cell*, or *cell state*. Furthermore, several so called *gates* manage the information flow.

The first gate is the *forget gate*[3], which decides what information to forget from the cell state:

$$f_t = sig(W_{fh}h_{t-1} + W_{fx}x_t) \tag{2.3}$$

where h_{t-1} is the hidden state from the previous time step, x_t is the current input and W_{fh} respectively W_{fx} are weights.

The counterpart to the forget gate is the *input gate* which decides which information will be added to the cell state:

$$i_t = sig(W_{ih}h_{t-1} + W_{ix}x_t) \tag{2.4}$$

New memory content is then proposed by:

$$\tilde{c_t} = tanh(W_{ch}h_{t-1} + W_{cx}x_t) \tag{2.5}$$

And finally, cell state c_t is updated by forgetting parts of the previous cell state and adding parts of the newly proposed memory content:

$$c_t = f_t c_{t-1} + i_t \tilde{c_t} \tag{2.6}$$

Last but not least, the hidden state h_t is calculated with the help of the *output gate* and the updated cell state:

$$o_t = sig(W_{oh}h_{t-1} + W_{ox}x_t) \tag{2.7}$$

$$h_t = o_t tanh(c_t) \tag{2.8}$$

A visualization of the whole information flow, the cell state and the different gates can be found in Figure 2.10.

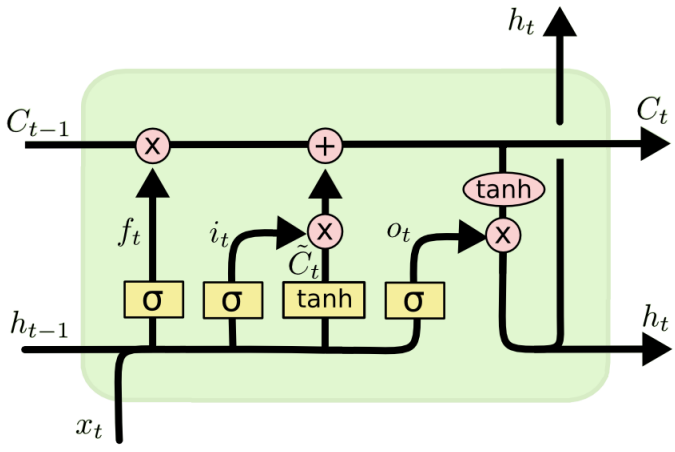


Figure 2.10: Visualization of an LSTM and its different gates. Graphic from [4]

[3]The forget gate was not present in the original work by Hochreiter and Schmidhuber [35], but was only later introduced by Gers et al. [26]

[4]`http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (last accessed on 20.06.2018)

2.3.2 Gated Recurrent Unit (GRU)

Similar to the LSTM the Gated Recurrent Unit (GRU) introduced by Cho et al. [11] makes use of gates. The hidden state h_t and the cell state c_t are merged into a single state h_t. Furthermore, instead of using three gates GRUs only use two, the *update gate* z and the *reset gate* r:

$$z_t = sig(W_{zh}h_{t-1} + W_{zx}x_t) \tag{2.9}$$

$$r_t = sig(W_{rh}h_{t-1} + W_{rx}x_t) \tag{2.10}$$

The reset gate can be seen as a counterpart to the forget gate and the update gate as one to the input gate. There is no output gate in a GRU. New candidate state values are computed by:

$$\tilde{h_t} = tanh(W_{hx}x_t + W_{hh}(r_t h_{t-1})) \tag{2.11}$$

And the state is finally updated by:

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h_t} \tag{2.12}$$

A visualization of the whole information flow can be found in Figure 2.11.

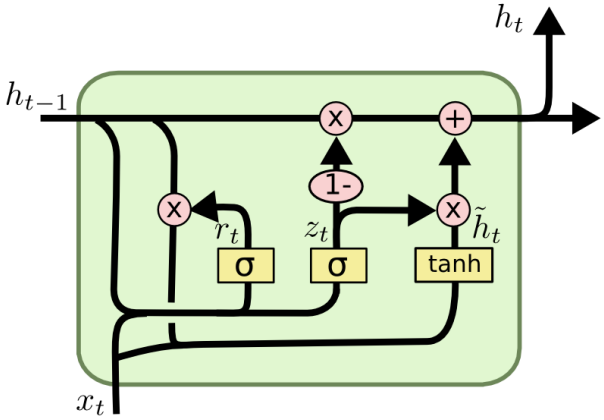


Figure 2.11: Visualization of a GRU. Graphic from [5]

In the work by Chung et al. [13], they concluded that both LSTM and GRU outperform traditional RNNs. However, when comparing the performance of LSTMs and GRUs they could not make a clear conclusion which one is better. In practice, GRUs are computationally more efficient as they have less parameters than the LSTMs.

Applications

RNNs are being used in various tasks where something can be modeled as a sequence. These sequences do not necessarily have to be temporal, e.g. a sentence can be regarded as a sequence of words. Furthermore, these sequences do not have to be in the input but can also be in the output. Examples include: 1) image captioning [43], where the input is a single image and the output is a sentence describing the image. 2) video classification [61], where the input is a sequence of frames and the output is a single class. 3) language translation [85], where both the input and output are sequences.

[5] `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (last accessed on 20.06.2018)

3 Related Work

As introduced in the first chapter, there are several approaches on how to recognize human activities. In this section, only vision-based, deep learning related work will be presented, as those are closest to the approach of this thesis. Moreover, as deep learning approaches became more and more popular over the last few years, lots of different methods have emerged. This section only covers the most common/important architectures along with some expansions of them.

3.1 Spatio-Temporal Features

A video can be regarded as a temporal sequence of images. For classifying human activities from videos it is thus an obvious choice to cover both dimensions, spatial and temporal, as both of them contain relevant information about what action a person is performing. The approaches presented in this section aim to extract spatio-temporal features from subsequences of the whole video. As the name suggests, these features contain information about both the spatial dimension and the temporal dimension simultaneously.

3D Convolutional Neural Networks for Human Action Recognition (2010)

This work by Ji et al. [42], being published in 2010, already emerged before the breakthrough of neural networks for image recognition tasks in 2012, making it one of the pioneer works of using neural networks for human activity recognition.

CNNs are deep networks that can extract features from raw, two-dimensional data, namely from raw images. This covers only the spatial dimension, however, for action recognition from videos it is also important to cover the temporal dimension. Thus, the authors of this paper developed a 3D CNN model, extending the old model by one more dimension, which will extract both spatial and temporal features.

The network structure can be seen in Figure 3.1. The input to the network is a block of 7 frames of size 60x40 consisting of the current frame, 3 preceding frames and 3 following frames. Before applying any convolutional layers to this input volume, they apply a set of hardwired kernels (H1). This step can be seen as a preprocessing step where information about the grayvalues, the horizontal and vertical gradients in each frame, and the horizontal and vertical optical flow of two consecutive frames is extracted from the 7 input frames. Afterwards a 3D filter of size 7x7x3 (7x7 in spatial and 3 in temporal dimension) is applied to each of the 5 categories listed above. Surprisingly, they only used two different filters for each category (indicated by the 23*2 in layer C2). This layer is followed by a subsampling layer (S3), which can be seen as a pooling layer, and the process is repeated with 3 different convolutional filters for each category (C4)

and a larger sampling rate (S5). The last convolutional layer (C6) is only performed in the spatial dimension as the temporal dimension already became relatively small, which is then followed by a fully-connected layer to determine the output.

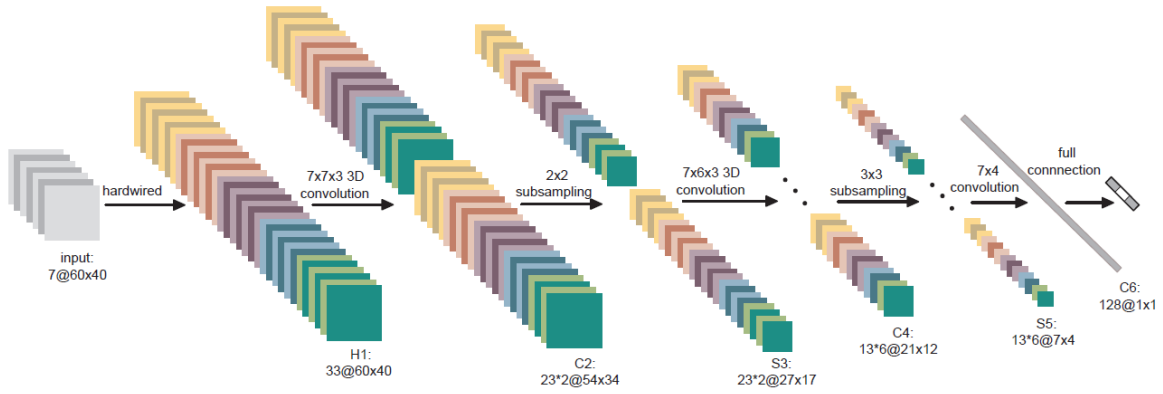


Figure 3.1: The complete 3D CNN structure. In the first layer (H1) the 7 input frames are preprocessed before the actual convolutions (C2, C4, C6) take place.

They evaluated their method on the TRECVID 2008 [80] and the KTH [73] dataset. On the TRECVID 2008 dataset, they compared their 3D CNN to a 2D CNN and two other baseline models; their model outperformed all the other models, hinting that the additional motion information indeed helps to classify an action. On the KTH dataset, they performed slightly worse (difference of 1.5%) than the handcrafted-based HMAX [75] [41] method.

Sequential Deep Learning for Human Action Recognition (2011)

Similar to Ji et al. [42], this work already emerged before the breakthrough of neural networks in image recognition. Baccouche et al. [3] use a two-step model where they combine a 3D CNN with a LSTM.

The goal of adding a LSTM sequentially to the 3D CNN is to be able to cover longer sequences. The 3D CNN takes as input a limited, usually small, amount of frames and can only classify actions which happen within this short sequence of frames. Thus, they can only cover a short motion while neglecting its evolution over time, as each block of frames is treated independently. The authors have shown in another work [2] that such information is beneficial to distinguish between actions. Therefore, they add a LSTM to their model which will cover the temporal changes of the whole video.

The first step of their two-step model consists of a 3D CNN similar to the one in the previously presented work. While the concept is the same, the structure is different. Instead of taking 7 60x40 frames as input, they take 9 34x54 frames. Also, they omit the preprocessing step of extracting information about gradients and optical flow. They are applying 3 3D convolutional layers creating 7/35/5 feature maps each. Between two convolutional layers, they inserted rectification (which simply returns the absolute value of its input) and sub-sampling layers. The 3D CNN is then trained on the KTH dataset [73] before the second step comes into place.

As already mentioned earlier, the second step of the two-step model consists of a LSTM. At each time step, this network takes as input the output of the third (which

is the last) convolutional layer of the 3D CNN. The structure of the LSTM itself is relatively simple, consisting of only one hidden layer with 50 nodes. The output is then a decision on the entire video sequence. The full model can be found in Figure 3.2.

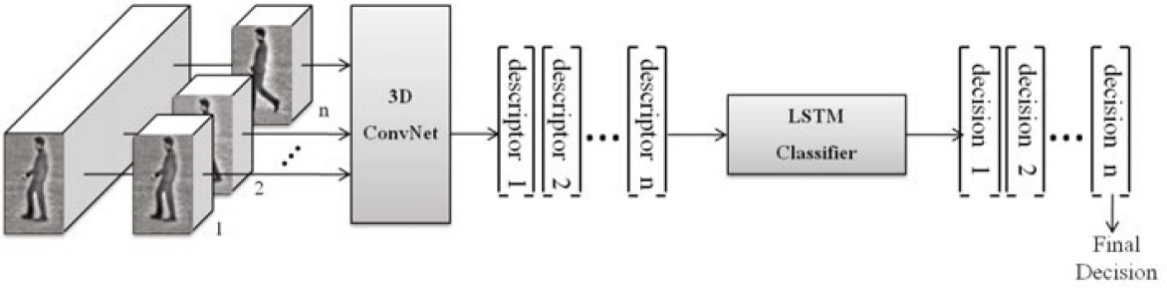


Figure 3.2: The two-step sequential architecture. A volume of 9 frames is input to the 3D CNN, where spatio-temporal features are extracted. These features are then input to the LSTM, which will classify the sequence.

They evaluated their model on the KTH dataset, which they further split into two: KTH1, is the original dataset and contains sequences where the same action is performed 3 to 4 times. For KTH2, they split these sequences into smaller ones, containing only one action per sequence. Furthermore, they did not only test their method against state-of-the-art approaches, but also against a baseline model of their approach which does not use an LSTM. Their results are as follows:

- Adding the **LSTM increases the performance** by around 3%, which is quite a lot considering that the overall performance is already at around 90%.
- The two-step model performs better on KTH1, which confirms the hypothesis that **LSTM are better suited for longer sequences**.
- Their model outperforms all other deep models on both KTH1 and KTH2 and achieves results close to the best state-of-the-art models.

3.2 Two-Stream Models

The approaches in this section differentiate from the ones in the previous section in the way they treat the spatial and temporal information. In the previous section, they extracted spatio-temporal features and handled them as a single unit. The approaches in this section separate the spatial and temporal information, i.e., treat them individually, and then make use of both of these channels to make a decision on a video sequence.

Two-Stream Convolutional Networks for Action Recognition in Videos (2014)

The architecture of this work by Simonyan and Zisserman [78] is based on the two-streams hypothesis which states that humans possess two distinct visual systems: the ventral stream ("what" stream) which is responsible for object identification, and the

dorsal stream ("where" stream) which recognizes motion [30]. They model these streams with two CNNs. The ventral stream is covered by what they call Spatial stream ConvNet and the dorsal stream is covered by the Temporal stream ConvNet.

For both networks they use the same architecture, the CNN-M-2048 from [10], which can be found in Figure 3.3. The networks consist of 5 convolutional layers and 3 fully-connected layers (full6, full7 and softmax). The activation function they use is the rectifier function (ReLU) and they apply 2x2 max pooling. The only difference between the two networks is that the Temporal ConvNet does not have a second normalisation layer.

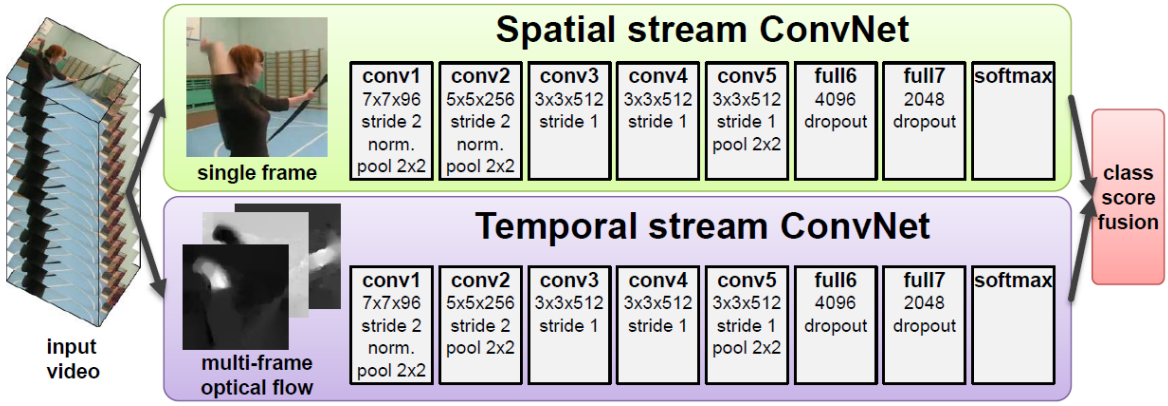


Figure 3.3: The Two-Stream Architecture: the Spatial ConvNet (top) extracts spatial information from a single frame, the Temporal ConvNet (bottom) extracts motion information from several frames of optical flow. Both channels are then combined to make a decision on which action takes place in a sequence.

They evaluated their model on the UCF-101 [81] and the HMDB-51 [48] datasets. However, before comparing their model to other state-of-the-art approaches, they first tested different configurations for their two-stream model and its components. Starting with the Spatial stream ConvNet, its role is to extract information from a single frame. As the architecture is already fixed, the only configurations they tested for this component were different training settings. They considered three setups, 1) training the network from scratch on the UCF-101 dataset, 2) pre-training the network on the ImageNet 2012 dataset [71] and then fine-tune it on UCF-101, 3) again pre-training the network on the ImageNet 2012 dataset and only retrain the last layer on UCF-101. The conclusions for this component were as follows:

- **Pre-trained networks** performed better than training from scratch.
- There is **no big difference** between fine-tuning several layers or only the last layer.

The Temporal stream ConvNet component is a bit more complex than the spatial one, giving room for more different configurations. The input to this network consists of the optical flow of several consecutive frames. The optical flow of two consecutive frames can be seen as a set of vectors which define the displacement which takes place in these two frames. For the input these vectors are further divided into their horizontal and

vertical component. Two main variations of the optical flow are considered 1) *optical flow stacking*, where the displacement vector is extracted at the same location each time, 2) *trajectory stacking*, where the displacement vector follows along the trajectory, thus not necessarily being at the same location every time. Furthermore, optical flow can be bi-directional, i.e. not only considering frames following the current one, but also the ones preceding it. Last but not least, they subtract the average displacement vector from each optical flow map to compensate for global motion such as camera movement. Their observations for the Temporal ConvNet are as follows:

- Considering the **optical flow of multiple consecutive frames** leads to better results as more motion is covered.
- **Subtracting the mean displacement vector** leads to better results as it compensates for global motion.
- **Optical flow stacking** performs slightly better than trajectory stacking, even though the difference is marginal.
- **Bi-directional optical flow** performs slightly better than only using forward-directional optical flow.
- The Temporal ConvNet performs better on its own than the Spatial ConvNet, leading to the conclusion that **motion information might be more important than spatial information** when it comes to action classification.

To combine the information of both components, they fused the output of the networks by either averaging them or using a linear SVM. Combining both spatial and temporal information leads to better results than when treated individually. Of the two fusion methods, SVM achieved the better results.

When compared to state-of-the-art approaches the two-stream model outperforms them on the UCF-101 dataset (even though it is only by 0.1%). On the HMDB-51 dataset it is outperformed by two other approaches.

Going Deeper into First-Person Activity Recognition (2016)

While the architecture and idea of Ma et al. [57] are similar to the ones described by Simonyan and Zisserman in the previous one, there are some fundamental differences and findings making this approach noteworthy.

Similar to the Spatial and Temporal network, they use two CNNs in parallel; one being responsible to detect objects of interest, the so called ObjectNet, and one being responsible to capture motion, the so called ActionNet. Both networks are depicted in Figure 3.4. As has been shown in [31] and [52], and also concluded by Simonyan and Zisserman [78], using both of these information sources together will lead to the best results, thus in this work they also combine the output of both networks to classify an activity. One important point to notice is that, contrarily to the works presented so far, they classify activities from first-person videos, more precisely, activities performed by the observer (i.e. the person wearing the camera).

ObjectNet is responsible to determine the object of interest in the activity taking place. Simply detecting all objects in an image is for one thing difficult to achieve,

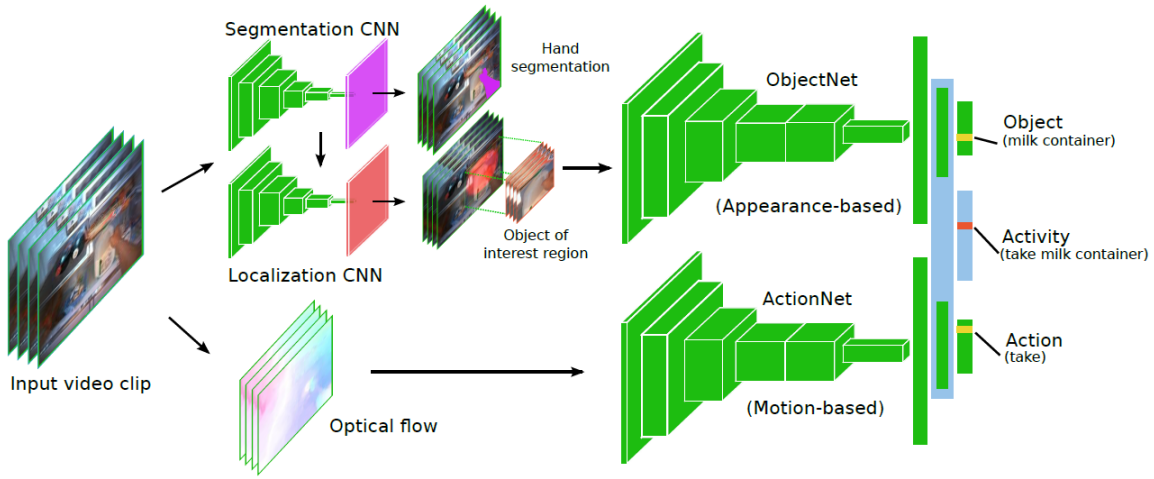


Figure 3.4: Top part: **ObjectNet** determines the object of interest using the appearance of the user's hand as a clue. Bottom part: **ActionNet** determines which action is taking place using motion information. Information of both networks combined leads to the classification of the activity taking place in the video sequence.

given that high accuracy is desired and for another thing one would need to determine the object of interest out of all the detected objects. The observation that the object of interest most often appears close to the hands lead to the idea of using the hand appearance as a clue to determining the desired object. For this purpose, they first train a hand segmentation network [56]. Afterwards, using the same network architecture as for the segmentation network, they train a localization network which will give them a probability map of where the object of interest occurs. Using said map, they crop a fixed-sized image of where they believe the object of interest is located and feed it to the actual object recognition network, which is the same model they used in the Spatial ConvNet [10]. Lastly, they take the object class scores from every frame in the action sequence and output the object with the highest mean value as the object of interest.

ActionNet follows the same idea as the Temporal ConvNet. They use forward-directional optical flow of multiple frames as their motion information to determine which action takes place in a frame. They then again average the scores of the classes over the whole sequence and output the one with highest value.

In a last step the information of both networks is fused to determine the activtiy from a video sequence. It is important to not just blindly combine the outputs of both networks, but to keep in mind the co-relation between the object of interest and the performed action. For example if the object of interest is a milk container then the probability of the action being *cut* or *crack* should be very low. Thus, the authors fuse both networks together by connecting their second last fully-connected layers and adding another fully-connected layer on top of it. When training the joint model, they do not only train this new fully-connected layer but also ObjectNet and ActionNet altogether.

They evaluated their model on the GTEA [21], GTEA gaze [22] and GTEA gaze+ [22] datasets. They evaluated ObjectNet and ActionNet individually, before reporting results of the combined model. Some of the most noteworthy findings are the following:

- **Hands are important for object recognition.** Even though the role of the ObjectNet is to determine the object of interest, the images input to the network contain big portions of hands on them. The network learned features of the appearance of the hand, e.g. how the hand is positioned, to help classifying the actual object.
- **Camera motion compensation is important for action recognition.** Similar observation to subtracting the mean displacement vector to compensate for global motion in the previous work.
- **Temporal motion patterns are important for action recognition**. That is, the temporal flow of an action is important, e.g. the actions *give* and *take* are only different in their temporal order ("reversing" *give* would lead to *take*).
- **Joint training is effective.** When training the model as a whole, the performance of the individual networks improved. Furthermore, fusing both CNNs with their joint method, they achieve better results than when using an SVM to combine them.
- **Object localization is crucial.**

Furthermore, they evaluated their model against a state-of-the-art method and against the two-stream model from Simonyan and Zisserman [78] (which can be seen as a predecessor model of this work) on the previousely mentioned datasets, outperforming both of them.

3.3 Pose-based Models

Another approach to human activity recognition are pose-based methods. Instead of treating each frame as a two-dimensional block of pixel-values, knowledge about the position and joints of the humans is extracted using depth information. Most of the spatial information is thus being discarded, as for example the background generally does not contain any hints on what action is taking place. The problem can thus be seen as recognizing an action from a temporal series of pose variations.

Hierarchical Recurrent Neural Network for Skeleton Based Action Recognition (2015)

Instead of only making use of two dimensions of an image, it would only be natural to make use of one more dimension: the depth. Using this extra information, one can extract the pose of the human subjects on said images. Actions can then be regarded as changes of the pose over time, i.e. a temporal sequence of movements performed by the human skeleton [77] [97].

Du et al. [18] make use of RNNs, which are strong on temporal series, and propose a hierarchical model for skeleton based action recognition. They divide the human body into five parts, the trunk, both arms, and both legs. Each of these parts is then fed into its own network. The outputs are then slowly fused together until the whole human skeleton is completed again. An important notice is that, unlike the models presented so far, they do not employ any CNNs in their model.

An overview of the hierarchical model can be found in Figure 3.5. They use bidirectional recurrent neural networks (BRNN) [74] at every stage in order to make use of past and future context for every timestep in the sequence. Layers 1, 3, 5 and 7 are BRNN layers, layers 2, 4 and 6 are fusion layers, layer 8 is a fully-connected layer, and layer 9 is a softmax layer for the action classification. Out of the 4 BRNN layers, only the last one has LSTM cells embedded, the other 3 are simply using the tanh activation function. The fusion layers are not trainable and their only role is to combine information from the previous layer to create the input for the next layer. The first fusion layer combines the trunk with each of the limbs, the second one creates the upper and lower body, and finally the last one builds the whole body.

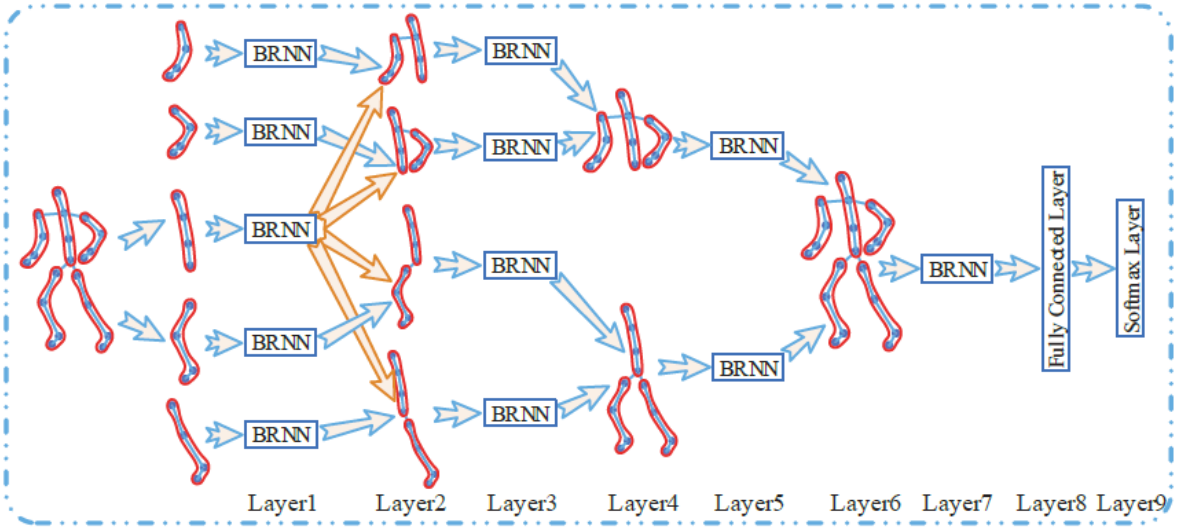


Figure 3.5: The human skeleton is split into five parts (torso, arms and legs) and each of these parts is fed into its own BRNN to get information about how the human pose changes over time. The deeper one advances in the network structure, the more parts are fused together (e.g. right arm and torso in Layer2), until the complete body is modeled (Layer6).

They evaluated their model on the MSR Action3D [51], the Berkeley MHAD [62], and the HDM05 [60] datasets. Not only did they compare their approach against other state-of-the-art ones, they also compared it to variations of their own one. They wanted to verify that certain architectural choices are indeed leading to better performance. Thus, what they tested was: 1) the importance of having bi-directional RNNs vs having uni-directional ones, 2) the choice of the structure, testing their hierarchical structure against a deep one which takes the whole skeleton as input, and 3) the advantage of having LSTM cells in their last BRNN layer.

Their findings and results are as follows:

- For the experiments on the different variations, the results are the same on all three datasets, namely:
  1. **Bi-directional** recurrent neural networks perform better than uni-directional ones.
  2. The **hierarchical architecture** achieves better performance than the deep architecture.
  3. **LSTM-cells** in the last layer lead to an increase in performance.
- Compared to the other approaches, they achieve state-of-the-art performance.
- Most of their errors occured through the **confusion of similar actions**. As they are only using the skeletal data of the human, actions with a similar spatial and temporal sequence, e.g. *grab* and *deposit*, look akin to the network.
- Their architecture is **very cost-efficient**. During testing, it only takes 52.46 ms on average to classify a whole sequence (234 frames on average) on a 3.2 GHz CPU. Using uni-directional RNNs it is even faster. It is important to note however, that the datasets already consisted of the skeletal data, thus factors like pose extraction from raw frames is not included in their calculation.

3.4 Hybrid Models

Recently, there also emerged approaches which combine the ideas from Section 3.2 and Section 3.3, thus creating two-stream models which also make use of the pose information.

Two-Stream RNN/CNN for Action Recognition in 3D Videos (2017)

Zhao et al. [101] propose a two-stream architecture combining a RNN and a CNN. Similar to Du et al. [18] from the previous section, they use skeletal information and feed it to an RNN and monitor its changes over time. For their CNN component they use the 3D CNN from [89] which can be seen as a progression of the one covered in Section 3.1. Both streams are then combined to classify an action from a video sequence.

Their RNN component takes as input the 3D coordinates of a number of joints representing the human skeletal. The first two layers of the network are two bidirectional [74] ones consisting of 300 GRU cells each. These layers are followed by a batch normalization layer [40] which is used to accelerate the training of the RNN. Next there is a dropout layer [83] with a keep probability of 75% to reduce overfitting, followed by a fully-connected layer with 600 neurons. Finally, a softmax layer is added to classify the actions.

The 3D CNN component takes as input 16 RGB frames and extracts spatio-temporal features from them. The network consists of a total of 8 convolutional layers, 5 max pooling layers, 2 fully-connected layers and a softmax layer. The model was pre-trained

on the Sports-1M dataset [44]. They propose two fusion models: decision fusion and feature fusion. Their full feature fusion model can be found in Figure 3.6 including details on the individual components.

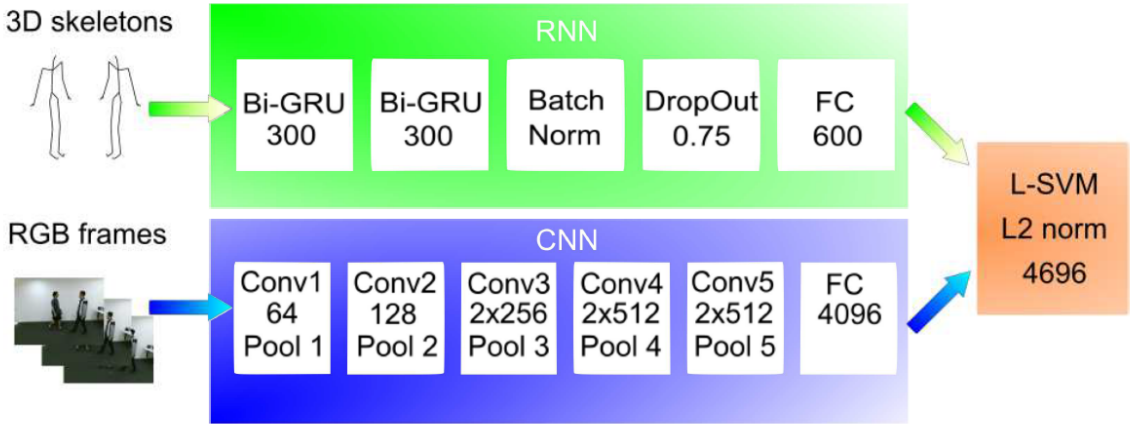


Figure 3.6: Top part: the RNN working on the skeletal data, consisting of two bidirectional GRU layers, a batch normalization layer, a dropout layer and a fully-connected layer. Bottom part: 3D CNN extracting spatio-temporal features from RGB frames. The first five layers are convolutional layers followed by max-pooling. Conv3-5 consist of two convolutions. Conv5 is then followed by a fully-connected layer. The features from the fully-connected layers from both networks are then concatented, L2 normalized and fed to an SVM classifier.

The decision fusion model is basically a voting method. They use trust weights w_r and w_c and the highest probabilities of the softmax layer of both networks to make their decision. The highest probabilty of the softmax layer from the RNN being p_r and the highest one from the CNN being p_c, their final decision is calculated as follows: if $w_r * p_r > w_c * p_c$ then return the output from the RNN, else the output from the CNN.

The feature fusion method is similar to the one used by Ma et al. [57] from Section 3.2. They first train both components indvidually. They then take the features from the first fully-connected layer from both networks, concatenate them, L2 normalize and finally feed them to an SVM classifier. In this case however, there is no joint training.

They trained, validated and evaluated their approach on the NTU RGB+D [76] dataset. The dataset consists of four major modalities out of which they used two: the 3D joint coordinates for their RNN and the RGB frames for their 3D CNN. They did not only compare their approach to other state-of-the-art approaches but also tested different settings for their RNN structure. Also, before evaluating their decision fusion method, they used the results on the validation set to determine the parameters w_r and w_c, resulting in $w_r = 1.00$ on both splits of the dataset and $w_c = 2.88/3.02$ respectively. These values hint that the 3D CNN component gives more hints on what activity is taking place than the RNN component. Their findings are as follows:

- Concerning the RNN structure, **batch normalization, dropout, 2 recurrent layers and adding a fully-connected layer** all help to improve the performance.

- LSTM and GRU have similar results in performance, however GRUs are **faster at training and testing.**
- Their final RNN alone already outperforms state-of-the-art approaches on the dataset.
- The **3D CNN component performs better** than the RNN, which is also indicated by the values of w_r and w_c.
- Using **combined information** achieves the best results, feature fusion outperforming decision fusion.
- **Features from RNN and CNN are highly complementary.** While the 3D CNN alone performs better than the RNN, the CNN can only extract spatiotemporal features from 16 frames (more are not possible due to GPU memory limitations). The RNN however, can learn motion patterns over the whole sequence. The fusion models performing better than the individual components supports this statement.
- Like Du et al. [18], their two-stream model **mostly confuses similar actions**, e.g. putting on a shoe vs taking off a shoe. The authors partly stated the noise in the skeletal data as an explanation for these errors.

Pose-conditioned Spatio-Temporal Attention for Human Action Recognition (2017)

Similar to the previous work, Baradel et al. [6] present a two-stream approach where one stream utilizes pose information and the other one uses RGB data. Even though the idea is pretty much identical, they handle and process the data in an entirely different way, as they use a CNN for the pose stream and an RNN for the RGB stream. Furthermore, they use so-called attention mechanisms to enhance the performance of the RGB stream by focusing on certain parts of a scene. Both streams are then fused to take a decision on the activity. A depiction of the approach can be found in Figure 3.7.

The goal of the pose CNN is to extract features about the temporal behaviour of the pose and correlations between different joints. The network takes as input a 20x300x3 volume. To extract information about the temporal behaviour, they need to look at multiple consecutive frames at once, which is represented by the first dimension. The second dimension represents the information about the different joints in each frame. They define a traversal order for the joints and store each joint's 3D coordinates. Some joints are even encoded multiple times in order to put them into relation with their neighboring parts. This results in 50*3=150 joint coordinates per person. As the activities they classify involve at most two persons, the second dimension equals 2*150=300. Note that if only one person is present, the values which would correspond to the second person are set to zero. The third dimension is the number of channels. They use the raw pose coordinates (first channel), their velocities (second channel) and their accelerations (third channel). The network structure itself is pretty compact, consisting of three convolutional layers, each followed by a max-pooling layer and employing ReLU

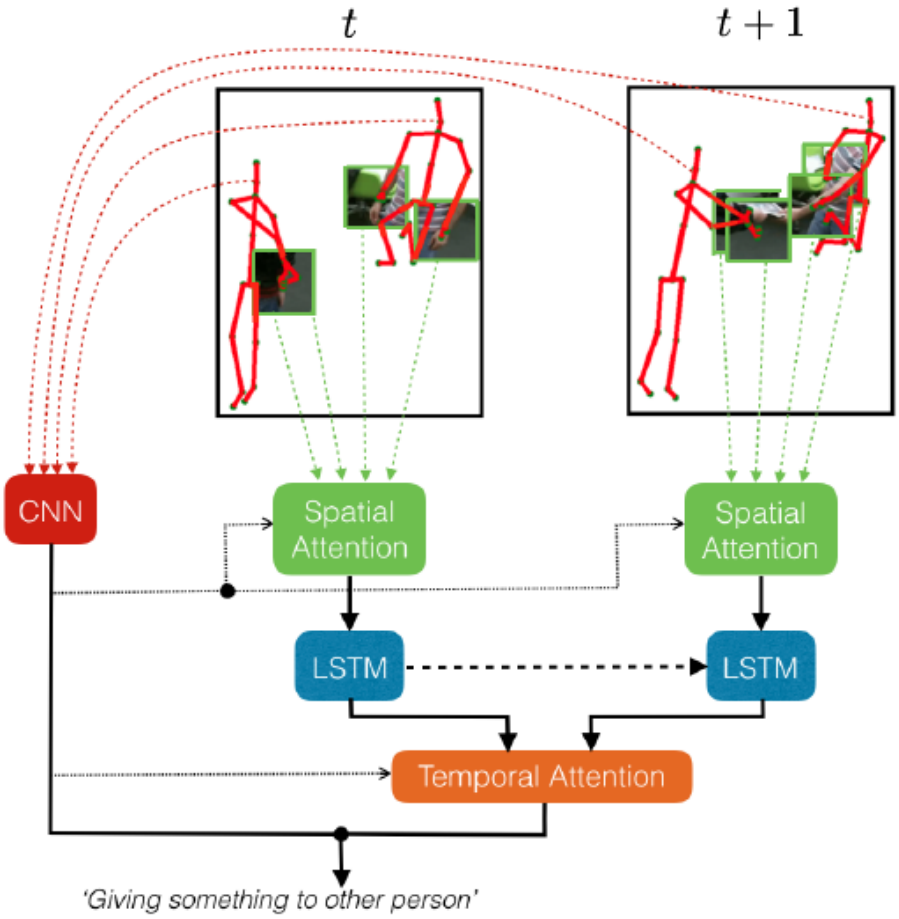


Figure 3.7: On the left hand side is the pose CNN which extracts features from poses from multiple frames at once. On the right hand side is the RGB-stream. With the help of a spatial and a temporal attention mechanism, its role is to extract the most important features from a RGB image. Finally, information from both streams are fused to classify an action sequence.

as its activation function. The pose features s which are ouput by the pose network are used at severeal steps of their model.

In order to overcome one of the most common problems of only using pose information, namely the confusion of actions which are similar in their motion, they take additional information from the RGB frames. However, instead of looking at each RGB frame as a whole, they only focus on specific parts of the image: the hands. Having information about the pose of the subjects, they crop fixed sized images around the hand joints and feed it to an Inception V3 convolutional network [86]. Furthermore, instead of just taking the features extracted by the network as is, they want to dynamically decide which features, respectively which hands play the most important role in a scene. A spatial attention mechanism is responsible for weighting said features. This mechanism is realized by a network which consists of a MultiLayer Perceptron (MLP) with one hidden layer with 256 units and uses the sigmoid activation function. It takes as input the pose features s and the hidden state h_{t-1} of a LSTM and outputs weights p_t. These weights are then linearly combined with the features extracted by the Inception V3 network to produce v_t. The aforementioned LSTM has a single layer with 1024 units and its hidden state h_t depends on its previous state h_{t-1} and v_t.

At each time step, they also calculate features using the hidden state of the LSTM. Information from each time step is then usually averaged to make a deduction about the whole sequence. However, similar to the spatial attention mechanism, it might be useful to assign a weight to the features of each time step. For their temporal attention mechanism they also employ another MLP, this time with 512 units. The input to this network are the pose features s and the weights p_t for every t output by the spatial attention mechanism. The weights output by the temporal attention mechanism are then linearly combined with the features calculated from the LSTM to output u. Finally, features from both stream, namely s from the pose-stream and u from the RGB-stream, are fused.

They evaluated their approach on the NTU RGB+D [76], MSR Daily Activity 3D [94] and the SBU Kinect Interaction [58] datasets. Their findings include:

- Their approach outperforms state-of-the-art methods on the NTU dataset (the two-stream model from the previous section was not included though).
- They use **transfer learning** from the NTU to the MSR and SBU datasets, and achieve state-of-the-art performance on SBU and a close performance on MSR.
- **Fusing** both streams achieves better results than the individual streams.
- **Attention Mechanisms** enhance the performance, especially if the RGB-stream is treated individually, but also in combination with the pose-stream.
- Adding **pose features** as an input to the attention mechanisms improves performance.

3.5 Unsupervised Learning

The works presented so far are all based on supervised learning: the models are being taught what they should learn. For this purpose they train on a labeled dataset, mean-

ing for each input its desired output is known. For example, an image classification model will be shown 1000 pictures. For each picture it will be provided with a label which will tell it whether there is a cat or a dog on the picture. After training on these pictures, the model should be able to map the correct label to a new image of a cat or a dog.

In unsupervised learning however, these labels are not provided. The model should learn to infer a function/structure of the underlying unlabeled dataset. One of the most common applications of unsupervised learning is clustering. Taking the example of the cat and dog pictures, if the unsupervised algorithm is being told to split the data into two, then it will probably result in a stack of cat pictures and a stack of dog pictures as they have distinguishing features. This section will cover an unsupervised learning work which is closest related to deep learning and human activity recognition. Other unsupervised methods such as deep belief networks (DBN) [34] will not be covered.

Learning Robot Activities from First-Person Human Videos Using Convolutional Future Regression (2017)

In their work, J. Lee and M. S. Ryoo [50] do not design a method to recognize human actions per se, as in classifying them. Rather, they design a method to make a robot learn them. Similar to how humans can learn activities by watching other persons perform them and then copying them, the authors of this work make the robot learn new activities by showing it unlabeled first-person videos of human-human interaction. The robot, who then "becomes" the first person agent of said videos, should learn to understand the temporal structure of the activity and learn how to execute it by itself.

The architecture of their approach consists of two components: 1) the perception component and 2) the manipulation component. The role of the perception component is to predict where the hands of the observer and the interacting person will be 1-2 seconds in the future. The manipulation component is then responsible to map this 2D hand information to the actual motoric execution. The details of the manipulation component will not be covered as the main point of interest consists of the perception component.

The perception component consists of two CNNs: 1) the hand representation network and 2) the future regression network. The perception component and its data flow during testing time (represented by the colored parts) can be seen in Figure 3.8. The hand representation network is used to extract hand information from a frame $\hat{X}_t$ at time t. For this purpose, they extend the SSD object detection framework [54] with a fully convolutional auto-encoder having 5 convolutional layers followed by 5 deconvolutional layers. In a first step features are extracted from the frame $g(\hat{X}_t) = \hat{F}_t$. Using these features the hand locations $h(\hat{F}_t) = \hat{Y}_t$ are then calculated. This network can detect hands at time t, the current frame. To further get the prediction for the hand locations at time $t + \Delta$ the second network, the future regression network, is needed.

The role of the future regression network is to infer the features at time $t + \Delta$: $r(\hat{F}_t) = \hat{F}_{t+\Delta}$. The network consists of 7 convolutional layers having 256 filters and kernel size 5x5, followed by one layer having 1024 filters and kernel size 13x13, followed by the last layer having 256 filters and 1x1 kernel size. The network is trained using the following loss function: $argmin \sum_t ||r(\hat{F}_t) - \hat{F}_{t+\Delta}||_2^2$. Furthermore, the network does

not only use the information at time t, but also makes use of the previous K frames to obtain $\hat{F}_{t+\Delta}$: $r(\hat{F}_t, ..., \hat{F}_{t-(K-1)}) = \hat{F}_{t+\Delta}$. The final pipeline then looks as follows: first, features $\hat{F}_t$ are extracted from frame $\hat{X}_t$. These are then fed into the future regression network to obtain $\hat{F}_{t+\Delta}$. From these features the hand locations $\hat{Y}_{t+\Delta} = h(\hat{F}_{t+\Delta})$ are calculated.

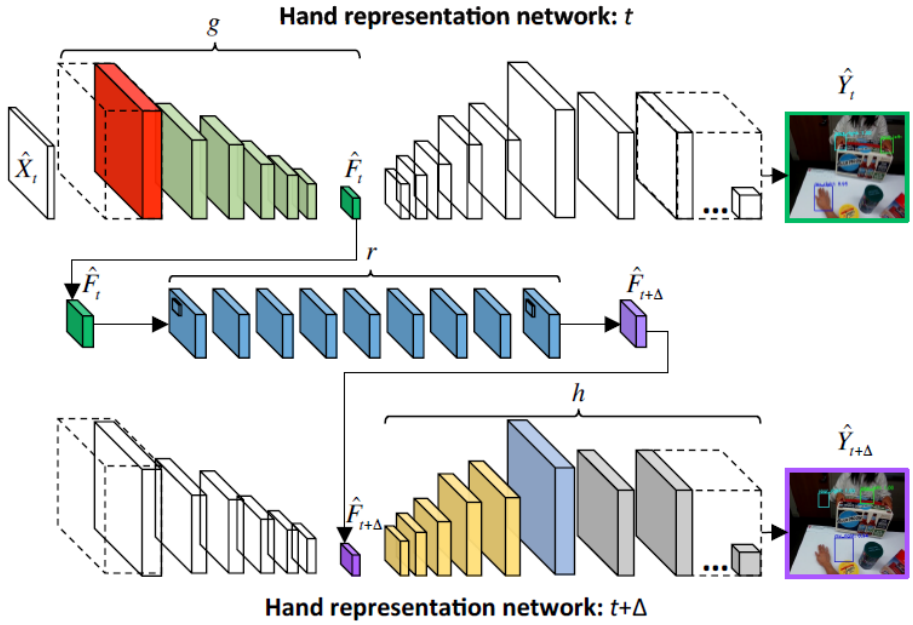


Figure 3.8: The data flow of the perception component at testing time. First, features get extracted from a frame $g(\hat{X}_t) = \hat{F}_t$ (top part). Next, using these features, the features at time $t + \Delta$ are infered by the future regression network $r(\hat{F}_t) = \hat{F}_{t+\Delta}$ (middle part). In the last step, the hand locations at time $t + \Delta$ are then calculated from the features output by the future regression network $\hat{Y}_{t+\Delta} = h(\hat{F}_{t+\Delta})$ (bottom part).

The hand representation network is trained on the EgoHands dataset [5], consisting of 4800 frames with hand labels. The authors added another 466 to train their network. Even though this part of the perception component is trained using supervised learning, the main part of learning activities does not require labeled data. The future regression network is trained on an unlabeled dataset created by the authors consisting of 47 first-person human-human interaction videos. The dataset consists of two different actions. Adding another action to the robot's repertoire would only require to record a few human-human interaction videos of the desired actions. The whole process of labeling the entire data can be omitted.

Evaluating unsupervised methods is difficult as, contrary to supervised learning, there is no ground truth or correct asnwer. They evaluated their perception component, which partly consists of supervised learning, against different baselines using other feature extractors or a different way to simulate the future regression network. Out of the contestants their model proved to be the best. They also concluded that using information of the past $K = 10$ frames increases the performance in predicting hand positions. To evaluate their whole model they conducted a user study where their model

scored an average of 3.29 (1:bad - 5:good). However, only 12 users participated in the study. Lastly, their model takes around 100ms per frame on a Nvidia Pascal Titan X GPU.

3.6 Recap and Runtime Analysis

To tackle the challenge of recognizing human activities from video with deep learning, quite a few methods have been proposed. One can extract spatio-temporal features from a video (Section 3.1), or one can seperate both dimensions and treat them seperately (Section 3.2). Abstracting an activity to a temporal sequence of the pose of a person (Section 3.3) or making use of RGB frame information in addition to the skeletal information have been researched (Section 3.4). Furthermore, there have also been works in the domain of unsupervised learning (Section 3.5).

In most of the cases, the main focus of the literature lies in the accuracy of their approaches. They measure the performance of their models based on how well they can classify actions in a sequence. However one factor which is often neglected, but plays an important role in most of the practical use cases, is how long it takes to classify such a sequence. For example when monitoring elderly people, it is of course great to recognize that a person has fallen down with an accuracy of 99.9%, however, it becomes quite useless if it takes an hour to classify the activity as it could already be too late. Even worse for autonomous cars where a fraction of a second can matter. Another point which is often used in the literature but cannot be applied in practical scenarios, is the usage of future information through after the fact recognition. Sequences are often considered as a whole where everything in the sequence is already known from the beginning. Some of the presented works use for example a time window centered around the current frame. In the said time window they use information from past frames as well as future frames. In practice however, explicit information of future frames is an unknown factor.

As some of the presented works do not expicitly state anything about the runtime of their models, a rough analysis on it will be provided for those works to see whether they can perform in weak real-time or not. A summary of the following analysis can be found in Table 3.1.

Both works from Section 3.1 (3D CNN [42] and Sequential Deep Learning [3]) make use of a 3D CNN which makes use of frames following the current frame. The authors don't state anything about the runtime of their models. The networks themselves are pretty compact, having only few layers and taking small images as input. However the hardware, especially the GPUs, at that time (2010/2011) was also a lot more limited than it is to date. The latter statement leads to believe that these models could probably not achieve weak real-time.

Simonyan and Zisserman (Two-Stream CNN [78]) also employ future information in their Temporal stream ConvNet. They do not explicitly mention anything about the runtime either. They do state though that the calculation of the optical flow takes 0.06s per pair of frames [9], and that they use 10 frames in their optical flow. This results in 9 pairs of frames for a total of 0.54s of calculation. Zhang et al. [100] who conducted further studies on Simonyan and Zisserman's work state a frame rate of 14.3 fps for this model. However, this is probably the frame rate where the optical flow is calculated in

a preprocessing step and is not added to the runtime. Given these facts, weak real-time seems to be feasible for this model.

Ma et al. (Going Deeper [57]) use forward-directional optical flow and thus make use of future information. Again, no statement about runtime has been made. The hand segmentation and localization are both based on [56], who state in their work that it takes around a tenth of a second to infer one image on a NVIDIA Titan X. The rest of the architecture is similar to Simonyan and Zisserman [78] and the calculation of the optical flow seems to be the bottleneck. Thus again, weak real-time seems to be feasible.

Du et al. (Hierarchical RNN [18]) employ BRNNs which make use of frames following the current one. As a runtime they state 52.46ms for a whole sequence, with around 234 frames per sequence, on a CPU. This results in roughly 0.2ms per frame. The compact structure of their network and the fact that they only use the pose without any image context allows them to achieve such a fast runtime. It should be noted, that the datasets they used in their work already contained the pose information and they did not have to extract them from each frame. Nonetheless, there are algorithms which can extract the joint information in weak real-time [77] [88].

Zhao et al. (Two-Stream RNN/CNN [101]) make use of future information in both of their components, by making use of 3D CNNs and BRNNs respectively. While they do make a statement about the training time, they do not give any information about testing time. They only mention that GRUs have a faster computational speed than LSTMs. The RNN component, being similar to the one from Du et al. [18], can probably easily achieve weak real-time. The CNN component is based on [89] which is a progressed version of the ones presented in Section 3.1. The authors of [89] aimed for a generic, compact, simple and efficient descriptor in their work and state a runtime of 313 fps, making this component also weak real-time.

Baradel et al. (Pose-Conditioned [6]) make their classification over sub-sequences of 20 frames. However, they do not state which frames they are using, thus it is unclear whether they make use of future information or not. For the runtime they state 1.4ms for a full prediction from features including pose feature extraction. RGB pre-processing, being the bottleneck, adds 1s on top of it, making the runtime roughly 1s for a sub-sequence of 20 frames.

J. Lee and M. S. Ryoo (Future Regression [50]) do make use of future information in their work. However, they calculate it themselves and do not take it as a given. As a runtime they state that it takes ∼100ms per frame on a Nvidia Pascal Titan X GPU.

Even though some works hint whether the spatial or temporal information works better for them, a real tradeoff between these dimensions has not been investigated. E.g. Simonyan and Zisserman [78] state that their Temporal ConvNet performs better on its own than the Spatial ConvNet, hinting that it might be better to invest in high frame rates. Karpathy et al. [44], who use 3D CNNs for the general case of video classification, however state that adding motion information only leads to a small increase in performance and spatial information alone can already give a good amount of information on what is taking place. This makes the idea of determining the SpatioTemporal Tradeoff worth investigating and a novel concept.

Approach	Runtime	Future Information
3D CNN [42]	Not stated[1]	Yes
Sequential Deep Learning [3]	Not stated[1]	Yes
Two-Stream CNN [78]	Not stated[2]	Yes
Going Deeper [57]	Not stated[2]	Yes
Hierarchical RNN [18]	52.46ms per sequence of 234 frames[3]	Yes
Two-Stream RNN/CNN [101]	Not stated[2]	Yes
Pose-Conditioned [6]	1s per sub-sequence of 20 frames[3]	Unclear
Future Regression [50]	∼100ms per frame	No

[1] most likely not achieving *weak real-time*
[2] most likely achieving *weak real-time*
[3] without the preprocessing step of extracting the pose

Table 3.1: Summary of the runtimes of the related works.

4 Concept

This chapter first introduces the architecture which is used for the HAR and the decisions behind it. A guideline on how to determine the SpatioTemporal Tradeoff and how to reproduce it on other datasets is presented. These steps are then elaborated for the HRC scenario and the newly generated dataset EgoBaxter of this thesis.

4.1 The Architecture - A Sequential Model

Chapter 3 covered several different architectures which are used in the literature to recognize human activities. Though they all achieve incredible results, some of the approaches cannot be used in practice. One of the main conditions a practical HAR application needs to fulfill is to not use future information. Some of the methods presented in the related work chapter treat sequences as a whole and make e.g. use of information of frame $t+10$ at frame t. Explicit information about the future is in practice however unknown. Furthermore, for most HAR applications to be useful in practice, they need to work with a frame rate of at least 1 fps. The architecture from this thesis should be applicable in practice and thus needs to fulfill both aforementioned conditions. An extension of Table 3.1 can be seen in Table 4.1, comparing the architecture of this thesis to the ones from Chapter 3.

Approach	Runtime	Future Information
3D CNN [42]	Not stated[1]	Yes
Sequential Deep Learning [3]	Not stated[1]	Yes
Two-Stream CNN [78]	Not stated[2]	Yes
Going Deeper [57]	Not stated[2]	Yes
Hierarchical RNN [18]	52.46ms per sequence of 234 frames[3]	Yes
Two-Stream RNN/CNN [101]	Not stated[2]	Yes
Pose-Conditioned [6]	1s per sub-sequence of 20 frames[3]	Unclear
Future Regression [50]	∼100ms per frame	No
This Thesis	Weak real-time	No

[1] most likely not achieving *weak real-time*
[2] most likely achieving *weak real-time*
[3] without the preprocessing step of extracting the pose

Table 4.1: Comparison of the architecture of this thesis to the ones presented in the related work chapter (Chapter 3).

Not making use of any future information already excludes employing methods like 3D CNNs or BRNNs. Pose information seems like a feasible option, however it was decided to stick with RGB frames. RGB images are typically easier to retrieve and the CNNs used in this thesis were pre-trained using RGB images. Parallel architectures

seem unsuitable to determine the tradeoff and unsupervised methods are also not the aim of this thesis. Thus, it was decided to use a sequential model, similar to Baccouche et al. [3], but using a regular CNN instead of a 3D CNN. Such an architecture of using a CNN in sequence with an RNN has also been employed in diverse video analysis tasks [61] [17] [98] [91], some including activity recognition. Optical flow is used in some of these works as well as in some of the related work. When using backward directional optical flow one can avoid using future information. However, the calculation of the optical flow adds up to the per frame calculations possibly reducing the achievable frame rate. It was decided not to use it in this thesis, however it is a point worth investigating in a future work.

Guideline

A general guideline for determining the SpatioTemporal Tradeoff for such a sequential architecture is as follows:

1. **Decide on the data**
   The first step is to decide on a dataset on which to train the sequential architecture. This could be one of the many publicly available datasets such as Sports-1M [44], or one could use a self-created dataset as is the case for this thesis.

2. **Train the spatial component**
   As the outputs of the spatial component are the inputs to the temporal component, the former has to be trained first. The role of this component is to extract information from the spatial dimension, in this case single frames. Furthermore, in order to determine the SpatioTemporal Tradeoff, multiple spatial components with various speeds and accuracies are required. In this thesis, convnets are used for the spatial component. These networks need to be evaluated before moving to the next step, as it would be ineffective to use CNNs which are too inaccurate.

3. **Train the temporal component**
   In order to classify the different activities, the temporal component is then trained on action sequences by using the ouputs of the spatial component as its input. As there are multiple spatial units, numerous temporal components have to be trained as well. The role of this component is to add the temporal dimension to the spatial information extracted from single frames. For this component RNNs are used in this thesis.

4. **Determine the SpatioTemporal Tradeoff**
   The SpatioTemporal Tradeoff can be determined from the evaluation of the temporal component, which represents the evaluation of the complete architecture. If less, but more accurate data provided by the spatial component achieves the highest accuracy, then the tradeoff is more on the spatial side. Vice versa, if the temporal component performs better with more, but less accurate data then the tradeoff is situated more on the temporal side.

A flowchart of the guideline can be found in Figure 4.1. Joint training, i.e. training the sequential architecture end-to-end is also possible (combining steps 2 and 3 to a single one), but is not covered in this thesis.

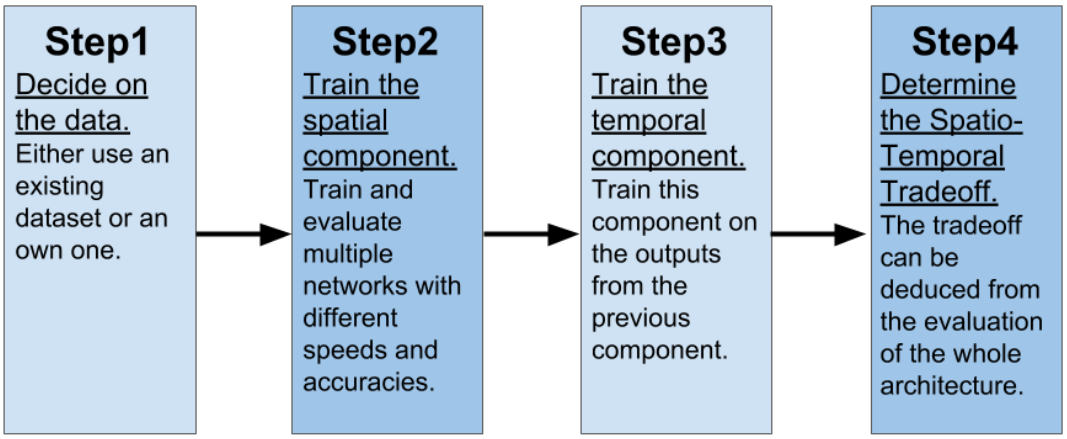


Figure 4.1: Flowchart of the guideline on how to determine the SpatioTemporal Tradeoff.

4.2 EgoBaxter - A Baxter PoV Dataset

The first step is to decide on the dataset to train and test the architecture on. There are quite a few existing datasets on human activity recognition, however there are a few issues with these as to why they are impractical for this thesis. One reason being that most of the datasets are either egocentric, e.g. GTEA [21] or GTEA gaze [22], or from a third-person point of view, e.g. KTH [73] or ActivityNet [20]. The egocentric datasets focus around recognizing the actions performed by the observer himself. In third-person videos the observer himself is not involved in the actions themselves. However, Baxter is supposed to interact and collaborate with surrounding humans and is thus required to recognize what is going on from its own point of view. This issue of first-person activity recognition was first tackled by Michael Ryoo and his colleagues [72]. However, the problem with their first-person dataset (JPL-Interaction dataset) is that it is not related to HRC, as it includes actions such as *hugging the observer* or *punching the observer*. Thus, it was decided to create a new dataset from Baxter's point of view: EgoBaxter.

As there is no need to create a gigantic dataset with a broad range of different actions for the purpose of determining the SpatioTemporal Tradeoff, it was decided to limit the dataset to the following two activities:

1. **Give**: handing a tool to the robot.

2. **Request**: asking the robot for a tool.

These two activities are similar in execution, in both cases a human reaches his arm/hand towards the robot. The difference lies in whether the person is holding a tool in his hand or not. For purely pose-based HAR approaches, as presented in the related work chapter, these actions might be difficult to distinguish.

The tool used in the handover is a yellow spirit level as depicted in Figure 4.2. In a practical scenario, a human worker might need more tools than just a spirit level, e.g. a

hammer or a screwdriver. However, these tools were not included for two reasons. First, the goal of the thesis is to determine the SpatioTemporal Tradeoff. There is no need to create a recognizer which can differentiate between several different tools. Second, in HRC scenarios the human's safety has top priority. Thus, one should refrain from equipping a robot with potentially dangerous objects.

Figure 4.2: The tool used for the handover.

EgoBaxter was further split into two parts: one part is used to train and test the CNN on, the other part is used for the RNN. The reason for dividing the data into two parts is that the RNN should not train/evaluate on outputs from data the CNN already trained on. The CNN part consists of 45 scenes recorded with 5 different individuals. Each individual performs the *Give* action 3 times, the *Request* action 3 times, as well as 3 scenes where they simply work with or without the tool. The RNN part consists of 160 scenes recorded with 8 different individuals. Each individual was asked to perform the *Give* and *Request* action 10 times each, 5 times with their left hand and 5 times with their right hand. There was only one action performed in each of these scenes. The scenario is located at the HRC lab of the DFKI Saarbrücken and is depicted in Figure 4.3.

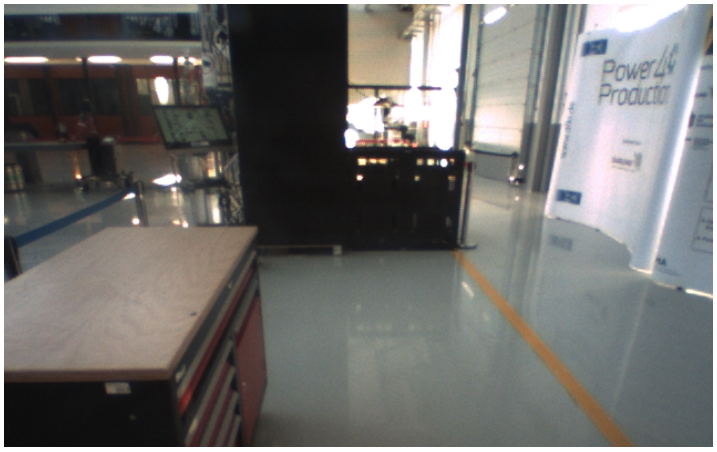


Figure 4.3: The scenario of EgoBaxter.

Additional Data

In addition to EgoBaxter, some more data was retrieved in order to be used for the CNN component. As hands play an important role in activity recognition, it was decided to make use of the EgoHands [5] dataset. This dataset contains images of two people interacting in different scenarios recorded from an egocentric point of view, and focuses on the hands. Furthermore, additional frames of hands and the tool have been recorded (see Section 4.4.1).

In order to train the networks with supervised learning, the next step is to label the aquired data. For the CNN data this means to draw bounding boxes around each object occurence in each frame. For the RNN data each frame needs to be assigned with an action label: *Give*, *Request* or *Dummy* (in case none of both actions is taking place). This process, and also more details on EgoBaxter (e.g. number of frames, resolution, etc.) and the additional data are covered in the implementation chapter (Chapter 5).

4.3 Training and Evaluating Neural Networks

It is common practice to split the whole dataset into two different parts: the training set and the testing or evaluation set. As the name suggests, the network trains on the training set which should make up around 80-90% of the whole data. The network is then frequently getting evaluated on the testing set, which is unseen data to the network and does not influence the training in any way. It is simply meant to check how well the network performs on new data at different stages of the training. There are a lot of values which can be measured during the training respectively the evaluation, the two most import ones being the *loss* and the *accuracy*. In supervised learning, the loss measures the difference between the prediction on a training sample and the actual ground truth of said sample. The goal of the training is to minimize the training loss by adjusting the weights of the network. The accuracy is typically defined as the number of correctly classified instances divided by the number of all instances, and expressed in percent. Loss and accuracy are related to each other in a way that when the loss decreases, the accuracy usually increases.

Overfitting and Underfitting

One common problem which tends to occur when training neural networks is *overfitting*. When a network is overfit it means that it learned the training data too well, in a sense that it performs badly when seeing new data. The opposite, but less common problem is called *underfitting* and occurs when the model is too simple to understand the underlying structure of the data. An ideal model lies in between, being neither underfit nor overfit. This is also often referred to as the *bias-variance tradeoff*. An example can be seen in Figure 4.4. The left model is too simple to capture the underlying structure, as too many points are classified incorrectly. The right model classifies all of the points correctly, which might seem good at first. However, some of these points are outliers and the model will probably have poor performance on new, unseen data. A balanced model is depicted in the middle. It classifies most of the data points correctly without paying too much attention to potential outliers.

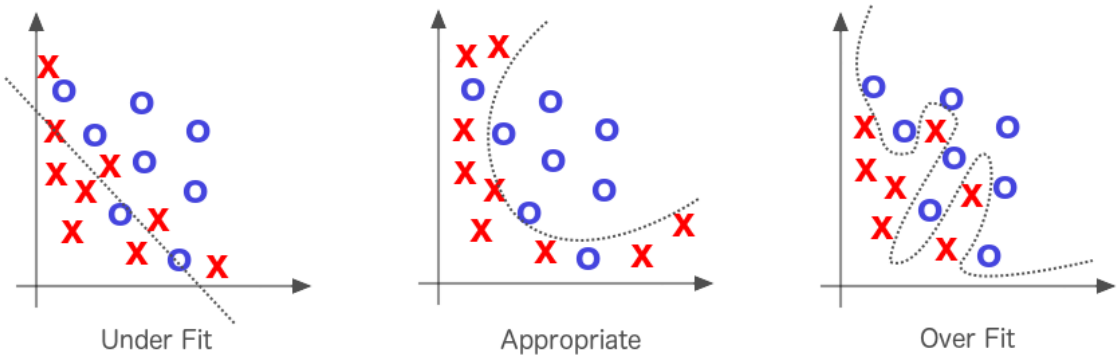


Figure 4.4: Example of an underfit, an ideal and an overfit model. Figure from [1].

Detecting and Preventing Overfitting

Overfitting can be detected from the training loss and the testing loss or testing accuracy. If the training loss decreases (Figure 4.5 red) but the testing loss (blue) starts increasing then the model starts to overfit. Analogously, one can notice overfitting when the training loss decreases and the testing accuracy (cyan) starts dropping. Small fluctuations in the testing loss or testing accuracy are possible and it should only be considered as overfitting if a general increase respectively decrease is notably visible.

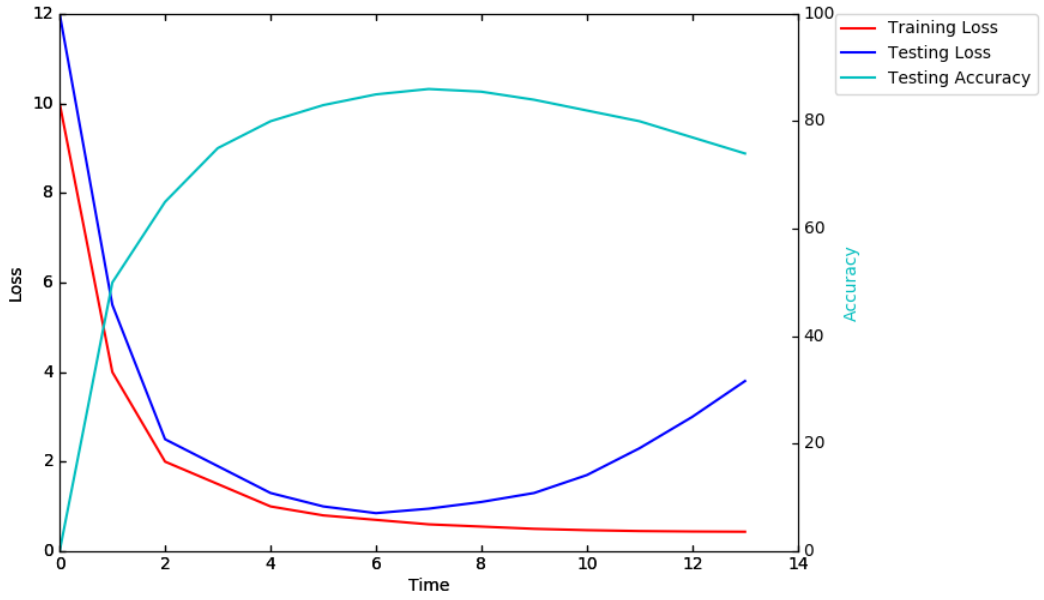


Figure 4.5: Detecting overfitting from the training loss (red) and the testing loss (blue) respectively the testing accuracy (cyan).

[1] The book *Deep Learning* by Adam Gibson and Josh Patterson

There are several ways to prevent overfitting including dropout [83], reducing the number of parameters in the network, increasing the number of data samples or simply stopping the training before the testing loss increases respectively the testing accuracy decreases.

Cross-Validation

In order to estimate the performance of a model in general, it is common practice to use cross-validation [84]. For cross-validation the data is divided into multiple parts, it is then trained on a subset of these parts and evaluated on the remaining ones. In this thesis, the so-called leave-one-out cross-validation is used, i.e. the models are trained on all of the parts except for one, on which they are evaluated. Cross-validation can be applied in an exhaustive manner, meaning that this process is repeated until each of the parts has been used for the evaluation exactly once. An example of exhaustive leave-one-out cross-validation can be seen in Figure 4.6.

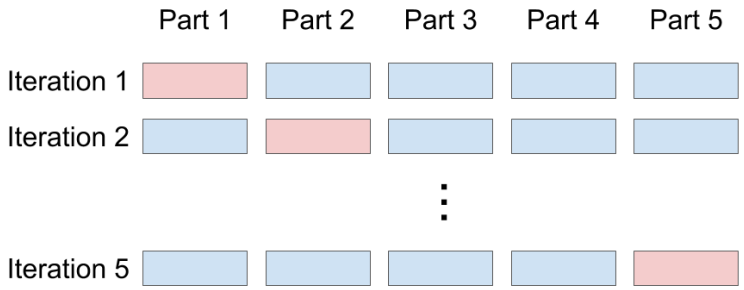


Figure 4.6: Example of exhaustive leave-one-out cross-validation when the data is split into 5 parts. The network is trained on the data in blue and evaluated on the data in red.

Mean Average Precision (mAP)

So far, the term of *accuracy* has been used to describe the performance of a network. However, the object detection component does not use the standard definition of accuracy (#correct/#all), but the so-called *Mean Average Precision (mAP)* as a metric to measure the performance.

There are different definitions of the mAP and different ways on how to calculate it. The calculation of the mAP presented here is based on the Pascal VOC detection metric [19] which is the one used for the object detection component in this thesis. The mAP is the mean of all the Average Precision (AP) values across the classes. The AP for a class is essentially defined as the *area under the precision-recall curve*. The *precision* is the ratio between correct predictions and total predictions, and the *recall* is ratio between correct predictions and the total number of correct instances. For example, an object detection algorithm should detect the cats in an image of 10 cats and 10 dogs. The algorithm outputs 8 bounding boxes, 6 which actually contain cats,

and 2 which were misclassified and contain dogs. The precision is then 6/8 and the recall is 6/10. In order to calculate the precision and the recall, true positives and false positives need to be identified. For this purpose, the Intersection over Union (IoU) of the predicted bounding box and the ground truth is calculated. If this value is bigger than a certain threshold the detection is considered a true positive, else a false one. In the Pascal VOC this threshold is set to 0.5. Furthermore, each detection outputs a confidence score along with the bouding box and the class label. This score is also accounted in the calculation of the precision-recall curve. The AP, and thus also the mAP, results in a value in the interval of [0:1] and can be seen as a metric which is similar to the standard accuracy. Both can be expressed in percent and the higher the better. Hereafter, mentioning the accuracy of the CNN refers to the mAP. The RNNs use the standard definition of accuracy for their evaluation.

4.4 Sequential Part 1 - The Spatial Component

The next step is to decide on the spatial component, namely the CNN. In order to determine the SpatioTemporal Tradeoff, several networks with different accuracies respectively speeds are needed. In this thesis, the CNNs are further used to localize objects of interest in the single frames, as for a practical application, Baxter also needs to know where to grab respectively put the tool. Though, in order to determine the tradeoff, CNNs without object detection could be used in this step as well.

4.4.1 Objects of Interest

The choices of the objects the CNN should detect are rather straightforward for the EgoBaxter dataset. The first obvious choice is to detect the *tool* as it is the main part of the handover. As suggested in [5] [6] [57], the hands are giving a lot of information of the action a person is performing. Thus, *left hand* and *right hand* are two further classes of the object detection component. It is important to distinguish both hands and not just use the general category *hand*. As information about the boxes will be given to the RNN component, the network might mix up the boxes of the hands which might in turn lead to a worse performance. The *face* was also considered as a class as it might give hints on a person's intention. However, in a lot of frames the face was only partially visible or not visible at all. Therefore, it was decided to keep it at the three classes *left hand*, *right hand* and *tool*.

4.4.2 Transfer Learning

Building CNNs from scratch is a tedious task, even more so when several networks with different speeds and accuracies are needed. Therefore, a common practice in deep learning is to make use of transfer learning [63] [99]. The idea is to reuse the knowledge aquired to solve one problem to solve another, similar problem. In case of CNNs, one typically takes a network which was pre-trained on a huge dataset, and retrains it on another, generally a lot smaller dataset. The learned features from the first layers are usually generic and are applicable to many datasets and tasks. Thus, they make a good starting point. A depiction of the idea of transfer learning can be found in Figure 4.7.

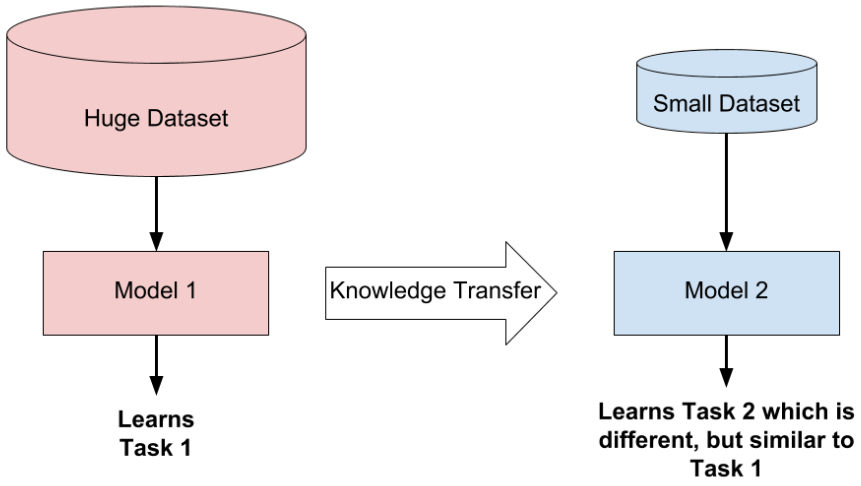


Figure 4.7: The idea of transfer learning for neural networks is to use a model which was pre-trained on a huge dataset as a starting point. In this case, the task of object detection is the same and the difference lies in the classes to detect, i.e. the dataset.

Fortunately, TensorFlow [1] provides a collection of CNNs with object detection [39] (Figure 4.8). The models used in this thesis are listed in Table 4.2 and the complete list can be found in their model zoo[2]. The models from Table 4.2 have been pre-trained on the Microsoft Common Objects in Context (MS COCO) dataset [53], and the mAP was calculated based on the corresponding evaluation protocol. The dataset contains 328k images, with 2.5 million labeled instances of 91 different object classes. The modelnames from Table 4.2 are made up of the object detection algorithm and the underlying CNN architecture. E.g. *ssd_mobilenet_v1_coco* uses the SSD algorithm with the Mobilenet CNN [38] to extract features. The Faster R-CNN models also have a *lowproposals* version. In the default version the RPN always outputs 300 box proposals, whereas the lowproposals models reduce this number, thus increasing the speed without a significant loss in accuracy. In their paper [39], they state that 50 proposals is probably a good sweetspot, but also showed examples with 100 proposals to compare Faster R-CNN to R-FCN. However, none of these numbers were explicitly confirmed to have been used in their model zoo.

These models are then retrained on EgoBaxter and the additional data in order to detect the three classes *left hand*, *right hand* and *tool*. The CNNs are evaluated using exhaustive leave-one-out cross-validation on the 5 individuals of the EgoBaxter_CNN data before moving on to the next step, as it would be inefficient to use inaccurate outputs as the input to the temporal component.

[2] `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md` last accessed on 26.06.2018

[3] `https://github.com/tensorflow/models/tree/master/research/object_detection` last accessed on 26.06.2018

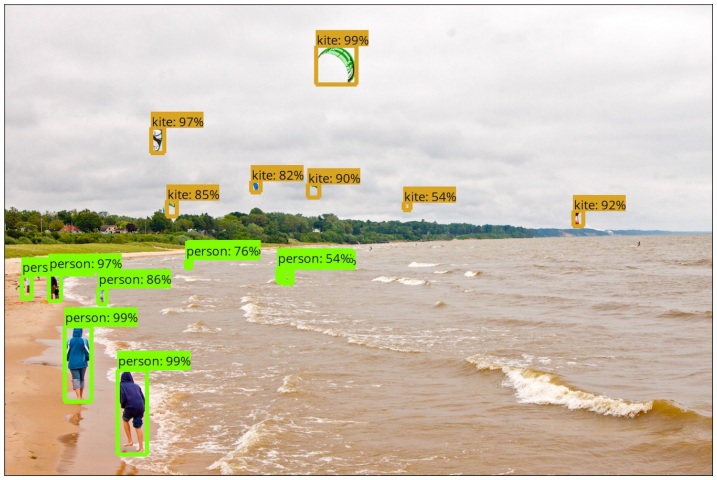


Figure 4.8: Example image of the TensorFlow object detection API[3].

Modelname	Speed (in ms)	mAP
ssd_mobilenet_v1_coco	30	21
ssd_inception_v2_coco	42	24
rfcn_resnet101_coco	92	30
faster_rcnn_resnet101_coco	106	32
faster_rcnn_resnet101_lowproposals_coco	82	
faster_rcnn_inception_resnet_v2_atrous_coco	620	37
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241	

Table 4.2: The TensorFlow models used in this thesis. The timings were performed using a Nvidia GeForce GTX Titan X. The mAP was calculated based on the MSCOCO evaluation protocol.

4.5 Sequential Part 2 - The Temporal Component

The third step is to add the temporal dimension by combining the information from several consecutive frames to form temporal sequences. For this purpose, recurrent neural networks are used in this thesis. The RNNs are implemented in Keras [12], a high-level neural network API which can use TensorFlow as its backend.

For this component transfer learning is unfortunately not an option, as a similar task from which knowledge could be transferred from was not found. Thus, the RNNs need to be built from scratch. For the CNNs the architecture of the networks was already given by the TensorFlow models which were used for transfer learning. As these architectures have proven to work well for the task of object detection, it seems to be a reasonable choice to adopt these architectures. This leads to some additional variables in the RNN such as the number of layers or the number of nodes per layer. Using an LSTM or a GRU, or what kind of input to give to the temporal component are further parameters.

As the RNNs train on the same CNN outputs for several iterations, it was decided to write these outputs to disk. This avoids that the computation of the CNN outputs becomes the bottleneck of the RNN training. To simulate the different frame rates of the object detection models, the RNN receives less inputs from the slower models in a given time interval than from the faster models. For example, on a time interval of 1 second, the RNN migth only receive information about 2 frames from the slower models, whereas it might obtain information about 5 frames from the faster ones. Moreover, the RNNs are compared to a baseline without memory using only the information of single frames as input instead of sequences.

In addition, it was decided to train several networks using the same settings. As the networks are built and trained from scratch, they are initialized with random weights. Thus, even using the same settings, the same network might achieve different accuracies. In order to gain a rough insight on how well they are able to perform, the accuracies of these multiple networks are then averaged. For the CNNs this step is not performed as they already have a solid starting point from the transfer learning and are not initialized with random weights.

The evaluation process is similar to that of the convnets using leave-one-out cross-validation on the 8 individuals from the EgoBaxter_RNN data. However, for this evaluation, cross-validation is used in a non-exhaustive way, only performing three iterations of training and testing.

4.6 Determining the SpatioTemporal Tradeoff

The SpatioTemporal Tradeoff can be determined from the end results of the RNN evaluation, as they represent the evaluation of the complete architecture. If the recurrent networks using the inputs from the slower, more accurate CNNs achieve the highest accuracy, then the tradeoff favors the spatial side. On the other hand, if the RNNs perform better on more inputs of a less precise CNN, then the tradeoff falls to the temporal side. A third outcome could be that a balance between both works best, using CNNs of intermediate speed and accuracy.

5 Implementation

This chapter covers the technical details of the previous chapter. This includes details on the EgoBaxter dataset and a description of the labeling process and the used tools, the pipeline on how to use this data to retrain the TensorFlow object detection models and details on their training settings, as well as the different RNN parameters and how they are configured and trained.

5.1 EgoBaxter - A Baxter PoV Dataset

EgoBaxter is a newly introduced dataset and contains short scenes of a human agent interacting with a Baxter from the robot's point of view. The activities include *Give* and *Request* using a yellow spirit level (Figure 4.2) as the tool of the handover.

5.1.1 Acquisition

The dataset was recorded using Baxter's head camera at a resolution of 640x400 and at a frame rate of 10 fps (see Table 5.2 for the choice behind this frame rate).

The data for the CNN consists of 45 scenes for a total of 3653 frames. Each of the 5 individuals performed the actions *Give* and *Request* 3 times each, and were recored for another 3 scenes where they were simply working with or without the tool.

Additional frames of both hands and the tool, hereafter referred to as *HandsTools*, were recorded in order to provide more data for the CNN. This includes 447 frames of both hands from two different persons, and 326 images of the tool, on which both hands are visible most of the time.

The data for the RNN consists of 160 scenes, being performed by 8 individuals for a total of 6955 frames. This results in an average of around 43 frames respectively 4.3s per scene. Each individual was asked to perform the *Give* and *Request* action 10 times each, 5 times with the left hand and 5 times with the right hand. Out of the 6955 frames, 4016 frames were labeled with *Dummy* (no-action) ($\sim$57.75%), 1556 with *Give* ($\sim$22.37%) and 1383 with *Request* ($\sim$19.88%), each handover action roughly lasting between 15 and 20 frames. During each scene exactly one action was performed.

Furthermore, the EgoHands dataset [5] is being used in addition to EgoBaxter in order to train the object detection component. EgoHands includes 4800 labeled frames of one-on-one human interactions of 4 different actions from an egocentric point of view. Both hands of the observer and the interactor are annotated with pixel-level ground truths. An overview of all the data used in this thesis can be seen in Table 5.1.

Data Name	Used For	#scenes	#frames	Labels
EgoBaxter_CNN	CNN	45	3653	*left hand, right hand, tool*
HandsTools	CNN	/	773	*left hand, right hand, tool*
EgoBaxter_RNN	RNN	160	6955	*Dummy, Give, Request*
EgoHands	CNN	48	4800	*left hand, right hand*

Table 5.1: Overview of all the data being used to train and evaluate the networks in this thesis.

5.1.2 Labeling

As the networks are trained with supervised learning, each input needs to have a ground truth assigned to it. For the CNNs the object instances in each frame need to be annotated with bounding boxes. For the RNNs the frames in each sequence need to be labeled with an action label.

Labeling the Frames

Tzutalin's LabelImg[1] (version 1.5.0 for Windows) was used to label the bounding boxes in each frame. The tool is straightforward to use as the bounding boxes can simply be drawn with the mouse on top of the image (Figure 5.1). After the box is drawn a label needs to be assigned to it, which can be chosen from a set of predefined labels (which the user can set by a simple .txt file) or by manually typing it in. A box was drawn if the object of interest was clearly recognizable, e.g. if only a part of a finger was visible then that part was not labeled as a left or right hand. The boxes are then saved in a .xml file where for each box the *class label, xmin, ymin, xmax* and *ymax* are stored.

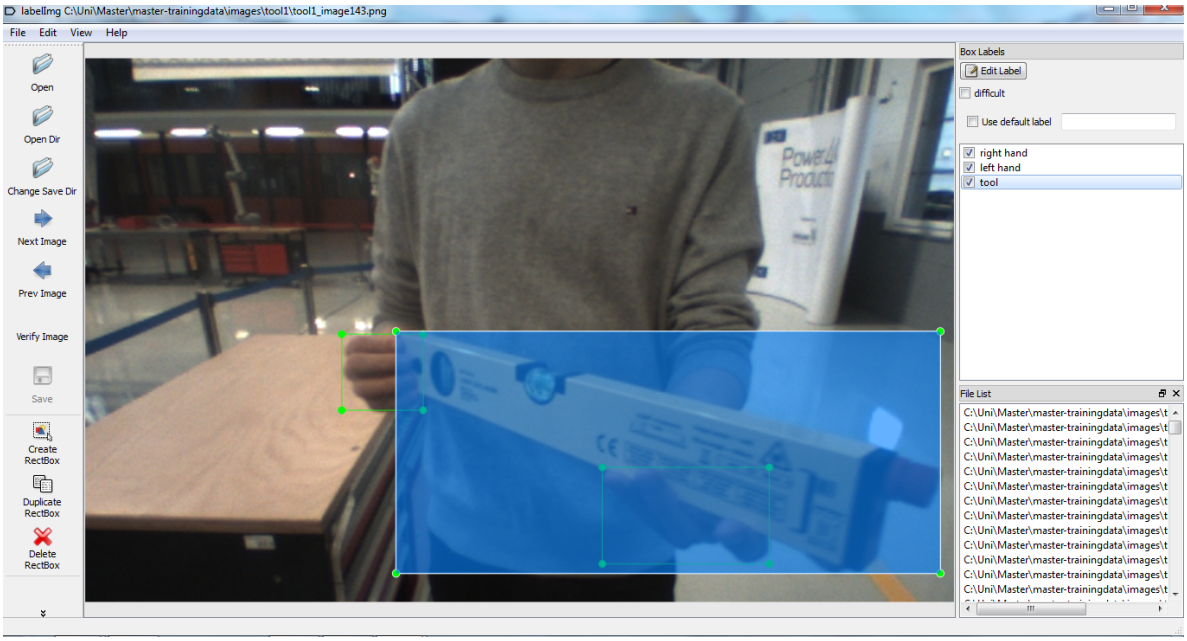


Figure 5.1: Interface of LabelImg[1].

[1] `https://github.com/tzutalin/labelImg` last accessed on 27.06.2018

For EgoHands, the 4800 frames are already labeled. However, the labels are pixel-level ground truths, where each pixel of the object is marked, instead of being surrounded by bounding boxes. Victor Dibia [93] kindly provides a script to generate the bounding boxes from the pixel-level annotations. This script runs over all the annotated pixels and retains the biggest and smallest x and y values, which are basically representing the bounding box (Figure 5.2). These are then written to a .csv file. Some minor adjustments were made to match the categories of this thesis.

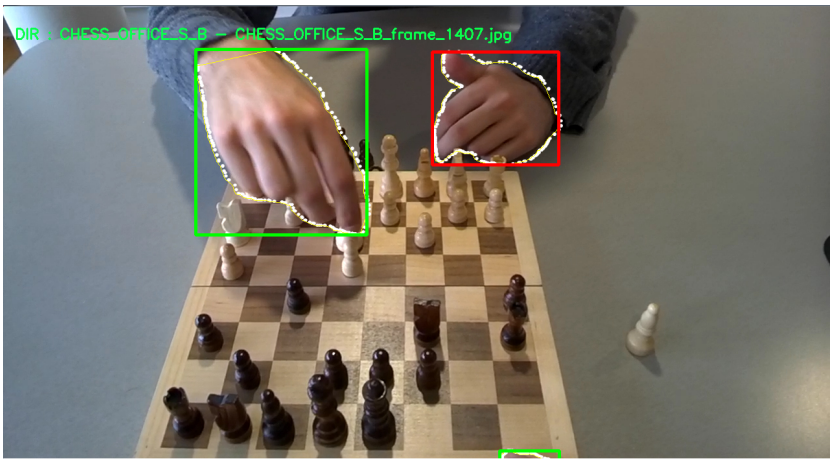


Figure 5.2: Example conversion from pixel-level ground truth to bounding box [93]. Only the outmost pixels of the pixel-level ground truths are displayed.

Create TFRecords

In order to train and evaluate the TensorFlow models, the data needs to be transformed into the TFRecord (.record) file format. This requires two steps: 1) transform the .xml files to .csv files 2) transform the .csv files to .record files. A flowchart of the steps can be found in Figure 5.3. The scripts `xml_to_csv.py` and `generate_tfrecord.py` from Dat Tran's racoon detector[2] were used for these two steps.

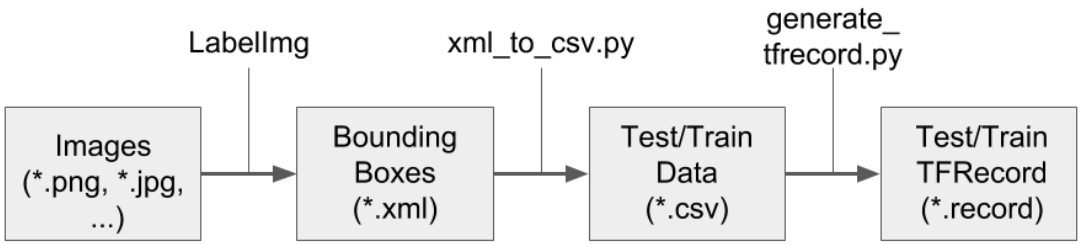


Figure 5.3: Flowchart of the steps to generate the TFRecords from raw images.

For the `xml_to_csv.py` script only the folder where the .xml files are stored needs to be adjusted. The .csv file stores the *image filename, image width, image height, class, xmin, ymin, xmax* and *ymax* values for each bounding box. To generate the TFRecords

[2] `https://github.com/datitran/raccoon_dataset` last accessed on 28.06.2018

with the `generate_tfrecord.py` script the folder where the image files are, which are listed in the .csv file, needs to be adjusted. Furthermore, each label has an ID assigned to it. In this case:

```
1 def class_text_to_int(row_label):
2     if row_label == 'left hand':
3         return 1
4     elif row_label == 'right hand':
5         return 2
6     elif row_label == 'tool':
7         return 3
8     else:
9         None
```

The .record files can then be used as input to train and test the TensorFlow models. Creating different training and testing setups is as simple as rearranging image and .xml files and rerunning the two above scripts after adjusting the target folder accordingly.

Labeling the Sequences

For the sake of assigning an action label to each frame in a sequence an own *Sequence-Labeler* was developed in Java (Figure 5.4). The SequenceLabeler displays the current frame in the center, the previous frame to the left and the next frame to the right. The labels one can assign to the frames are shown in the radio group in the upper-right corner. Pressing 'w' or hitting the 'Write' button will write the selection to a .txt file in the format of `filename : label`, e.g. `scene122_image047.png : Give`. Frames from the beginning to the end of a handover are labeled with *Give/Request*. All the other frames are labeled with *Dummy*.

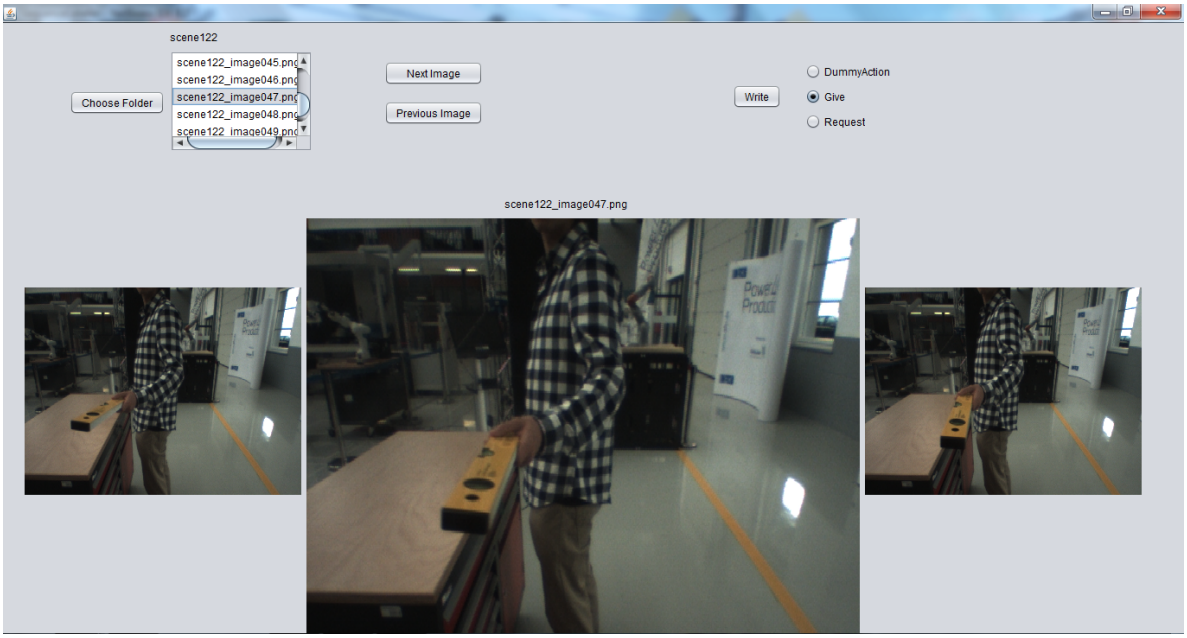


Figure 5.4: The interface of the SequenceLabeler developed in Java.

5.2 The Spatial Component

The spatial component was implemented in TensorFlow [1] (GPU version 1.3.0) using the August 11 2017 release of their object detection API[3]. The Nvidia CUDA Toolkit[4] (version 8.0) and Nvidia cuDNN[5] (version 6.0) were further installed in order to use the GPU version of TensorFlow. To shorten the long names of the models from Table 4.2, the version and pre-trained dataset are omitted, e.g. *ssd_mobilenet_v1_coco* will hereafter simply be called *ssd_mobilenet*. Furthermore, as the *lowproposal* models have a faster runtime without a significant loss in acccuracy, they will be used instead of the version where the RPN outputs 300 proposals. Thus, *faster_rcnn_resnet101* refers to the *faster_rcnn_resnet101_lowproposals_coco* model and *faster_rcnn_inception_resnet* refers to the *faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco* model. As it was not really clear from their paper [39] if *lowproposals* refers to 50 or 100 proposals, it was decided to set it to 100.

The speeds indicated in Table 4.2 were reported on a Nvidia GeForce GTX Titan X. The local computer on which the HAR application will later on be tested has a different GPU, namely a Nvidia GeForce GTX 1080. To determine the runtime on said machine, the different models were run on 320 images and the times were averaged. A comparison to the Titan X GPU can be found in Table 5.2.

Modelname	Titan X (ms)	GTX 1080 (ms)	FPS on GTX 1080
ssd_mobilenet	30	151	∼6.6
ssd_inception	42	158	∼6.3
rfcn_resnet101	92	226	∼4.4
faster_rcnn_resnet101	82	240	∼4.2
faster_rcnn_inception_resnet	241	486	∼2

Table 5.2: Comparison of the speed (in ms) on a Nvidia GeForce GTX Titan X (reported by the TensorFlow object detection API) and a Nvidia Geforce GTX 1080 (local machine).

The choice for the frame rate of 10 fps at which EgoBaxter was recorded is based on this table. As even the fastest model can only process around 6 frames per second, a recording framerate of 10 fps is sufficient for EgoBaxter.

5.2.1 Configuring the Detection Models

Each of the object detection models comes with a configuration file (.config) which is used when training respectively evaluating a model. There are a lot of parameters which can be set in such a file, and one could even change parts of the architecture. In the following these parameters will be listed before the config files are explored in more detail.

[3] `https://github.com/tensorflow/models/tree/master/research/object_detection` last accessed on 28.06.2018

[4] `https://developer.nvidia.com/cuda-toolkit` last accessed on 28.06.2018

[5] `https://developer.nvidia.com/cudnn` last accessed on 29.06.2018

Parameters

Changing parameters in a CNN involves training the network again. As the training is a computationally expensive step, it was decided to try to keep as many parameters as possible fixed and only change those which really need to be changed.

The CNNs are not built from scratch, but are based on existing models which are retrained on the EgoBaxter dataset using transfer learning. As the architectures have already proven to work well for the task of object detection there is no need to reinvent the wheel. Thus, it was decided not to change the structure, e.g. not changing the number of layers or the number of filters etc. This reduced the number of possible parameters, but there are still a few variables left, including:

- **Batch Size**
  The batch size is the number of samples which are propagated through the network before the weights are updated. For example, for 100 samples and a batch size of 5, the network is getting updated 20 times to process all the samples. The default values are 24 for ssd_mobilenet and ssd_inception and 1 for the other three models. This parameter was fixed at said default values.

- **Data Augmentation**
  Data Augmentation is one way to cope with overfitting. Common ways to augment the data is to add horizontally and/or vertically flipped versions of the samples to the training data. The `random_horizontal_flip` data augmentation, which was set for all the models, was disabled. Mirroring the image will lead to difficulties when distinguishing the left hand from the right one, the same issue was descriebd in [5]. Both SSD models further have the `ssd_random_crop` data augmentation option which was left enabled.

- **Dropout Rate**
  Dropout [83] is also one of the most commonly applied ways to help reduce overfitting. When using dropout a random portion of the inputs to a neuron is simply set to 0 at each update during training. Surprisingly, dropout was by default not enabled in the TensorFlow models. The `dropout_keep_probability` was set and fixed to 0.8 for all models except rfcn_resnet101 where dropout was not applicable. Note, the dropout rate is often expressed as the probability to drop inputs, here however, it expresses the probability at which inputs are kept.

- **Learning Rate & Learning Decay**
  To understand the meaning of the learning rate it is important to have an idea of how gradient descent works. A nice metaphor to gradient descent is climbing a hill where the goal is to reach the summit. The size of the steps one can take is fixed. If this size is large, then one can reach the top faster, however one also risks overshooting the summit. If this size is small, then one can make sure to reach the peak, however this comes at the cost of taking a long time to reach it.

  The learning rate can be seen as this step size and controls how much the weights are adjusted during each update. A high learning rate reduces the training time, but might also lead to the network not converging (overshooting the top). A low learning rate will make sure the network converges at the cost of a high training

time. In practice the learning rate is thus generally set to a high value in the beginning and reduced throughout the training process. As transfer learning is employed, the learning rate does not have to start as high as if it would be for a new model. For this parameter mostly the default values were adopted. The decay of the learning rate was set dependant on the step number.

- **Optimizer**
  The optimizer is used to update the weights in the network in order to minimize the loss. Most optimizers are based on some form of gradient descent. These were left at their defaults which is the RMSProp [87] for the SSD models and MomentumOptimizer [65] for the R-FCN and Faster R-CNN models.

- **Step Number**
  The number of steps defines the number of times the weights are updated during training. This is not to be confused with the term of *epoch* which is also often used to define how long a network is trained. Taking the example from the batch size parameter of 100 samples and a batch size of 5: the network is getting updated 20 times to process all the samples once. In this case it takes thus 20 steps to process all the samples. If batch size was 1, it would take 100 steps. This parameter highly depends on the training data and different values were tested for the different models and input data.

An overview of these parameters and their values can be seen in Table 5.3.

Modelname	BatchSize	DataAug.	Dropout	LearningRate	Optimizer	StepNum
ssd_mobilenet	24	random_crop	0.8	0.004	RMSProp	**TBD**
ssd_inception	24	random_crop	0.8	0.004	RMSProp	**TBD**
rfcn_resnet101	1	None	N/A	0.0003	MomOpt	**TBD**
faster_rcnn_resnet101	1	None	0.8	0.0003	MomOpt	**TBD**
faster_rcnn_inception_resnet	1	None	0.8	0.001	MomOpt	**TBD**

Table 5.3: Overview of the CNN parameters.

Config Files

The config files for the models consist of five parts:

1. `model`
   The `model` defines the architecture, i.e. which object detection algorithm and which feature extractor to use etc. These settings were left as is, except for two minor changes. First, the number of classes needs to be specified, which is 3 for EgoBaxter. Second, dropout was enabled with a `keep_probability` of 0.8 for all models except rfcn_resnet101 for which it was not applicable.

2. `train_config`
   In the `train_config` things like the `batch_size`, `optimizer`, `learning_rate`, `num_steps` and the path to the starting checkpoint, i.e. the pre-trained model, are specified. This part was tweaked the most for the different training settings, mainly using different values for the number of steps.

3. `train_input_reader`
   In the `train_input_reader` the path to the TFRecord of the training data as well as the path to the `label_map` (see below) need to be defined.

4. `eval_config`
   In the `eval_config` one can specify how often the network should be evaluated and the time in between two evaluations. Furthermore, one can set to visualize a number of samples to see how the network performance evolves over the training.

5. `eval_input_reader`
   Analogously to the `train_input_reader`, in the `eval_input_reader` the path to the TFRecord of the testing data as well as the path to the `label_map` need to be set.

The `label_map` is a .pbtxt file and is similar to what needed to be defined in the `generate_tfrecord.py` script: a map of labels and IDs. For EgoBaxter this map looks as follows:

```
item {
  id: 1
  name: 'left hand'
}
item {
  id: 2
  name: 'right hand'
}
item {
  id: 3
  name: 'tool'
}
```

5.2.2 Training the Models

Once the TFRecords, the label map, the config file, and the model are ready, the training can be launched with the `train.py` script from the object detection API:

```
python train.py --logtostderr
--train_dir=training/
--pipeline_config_path=path_to_config/configfile.config
```

It only needs the config file as input. The folder given by `train_dir`, in this example `training/`, specifies the location where the model will be saved. During the training process a checkpoint is saved every two minutes by default settings. If there are more than five checkpoints, the oldest one will be deleted in order to save disk space. Furthermore, an *event file* is created which is used by TensorBoard (see Section 6.1) to visualize, among other things, the training loss and how it varies during the training.

The networks are trained on a Nvidia DGX-1 which helped reduce the training time due to its strong performance. Nevertheless, training the CNNs was still a very time consuming step.

Training Settings

For the CNN training exhaustive leave-one-out cross-validation is performed. The CNN data of EgoBaxter is divided along the different individuals resulting in 5 parts. Hereafter the different cross-validation setups will be referred to as *personsplits*, resulting in *personsplit1* to *personsplit5*. Hyperparameters like the the number of steps are first identified on personsplit1 and are then taken over for the other personsplits. The models are then trained and evaluated on all the personsplits to check how well they perform in general. An accuracy of at least 50% would be desirable for all the models before moving to the temporal component, expecting a similar ranking as in Table 4.2. Some more experiments are conducted on personsplit1 using additional data as input, namely adding HandsTools, adding HandsTools and EgoHands, and training the models first on EgoHands and then on HandsTools and personsplit1. It is expected that similar observations can be made on the other personsplits, however the experiments are not explicitly conducted due to the time it would take to train all the networks.

5.2.3 Evaluating the Models

The networks have to be evaluated in parallel with the training in order to be able to observe the performance during different stages of the training. Analogously to the training, the evaluation can be launched with the `eval.py` script from the object detection API:

```
python eval.py --logtostderr
--pipeline_config_path=path_to_config/configfile.config
--checkpoint_dir=training/
--eval_dir=eval/
```

The config file is the same for the training and the evaluation, and the directory specified by `checkpoint_dir` points towards the directory where the training checkpoints are saved. The script produces event files which are then saved in the folder specified by `eval_dir`. Just like for the event files created during the training, they are used by TensorBoard in order to visualize certain metrics. In case of the evaluation, these include the testing accuracy (mAP) and box predictions on sample images. The evaluation of the spatial component takes place before moving on to the temporal component, as it would be inefficient to use outputs from CNNs which perform poorly. The evaluation results are presented in Chapter 6.

Once satisfied with the results, a so-called frozen inference graph which represents the network model can be generated from a training checkpoint:

```
python export_inference_graph.py
--input_type=image_tensor
--pipeline_config_path=path_to_config/configfile.config
--trained_checkpoint_prefix=training/model.ckpt-xxx
--output_directory=saved_model
```

This frozen inferenece graph can later on be loaded in applications using TensorFlow to detect the objects of interest from EgoBaxter. An overview flowchart of the training/evaluation of the spatial component can be found in Figure 5.5.

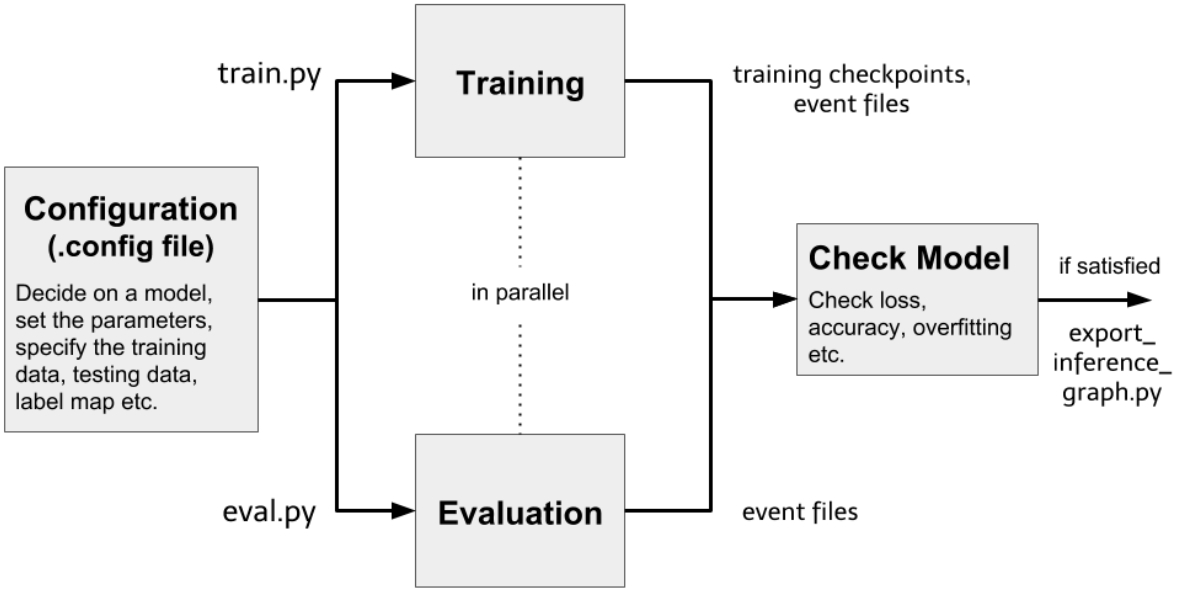


Figure 5.5: Overview flowchart of the training/evaluation of the spatial component. First all the settings in the configuration file need to be made. This config file is then used in the training and evaluation processes which are run in parallel. Using the outputs of these processes, the performance of the model can be checked, and if satisfied, an inference graph can be generated.

5.3 The Temporal Component

The temporal component was implemented in Keras [12] (version 2.1.5), a high-level neural network API which can be used on top of TensorFlow. Unlike for the spatial component, transfer learning is unfortunately not possible for this component as no similar task from which knowledge could be transferred was found. The RNNs are thus built from scratch.

5.3.1 Building the Recurrent Networks

As the networks are built from scratch, structural choices like the number of layers and the number of neurons per layer become additional variables. Analogous to the CNNs, it was tried to keep as many of the parameters as possible fixed. Some of these parameters also appeared in the CNN and are marked with a *, detailed explanations for these are omitted:

- **Batch Size***
  The batch size for the RNNs was fixed to 8.

- **Dropout Rate***
  Dropout rates of 0.2 and 0.4 were tested. Note that in this case they represent the probabilty at which inputs are dropped, contrary to the CNNs where they represented the probabilty at which the inputs are kept.

- **Learning Rate***
  The learning rate was fixed at 0.001, the default value for the corresponding optimizer.

- **Loss Function**
  The function which is used to calculate the loss during the training. This function was fixed to the *categorical_crossentropy* loss function.

- **Number of Epochs**
  The number of epochs defines the number of times the complete data is processed by the network for training. Just like the number of steps for the CNN, this parameter needs to be determined through running different experiments.

- **Number of Layers**
  This parameter defines the number of recurrent layers in the network, tests were conducted with 1 and 2 layers.

- **Number of Units**
  Keras uses the term of units, but this is interchangeable with cells or neurons. Different settings for this parameter were used. In case 2 recurrent layers are used, the same number of neurons is used in both.

- **Optimizer***
  The optimizer was fixed to the RMSProp [87].

- **RNN Type**
  Here the options are whether to use an LSTM or GRU, both were getting investigated.

- **Window Size**
  The window size determines the time interval from which sequences are input to the RNN. E.g. if said time interval were 2 seconds long, then the candidate frames are the current one and the 19 frames before it, as the sequences were recorded with a frame rate of 10 fps. As the CNNs operate at different frame rates, only a subset of these frames will be used as the input to the RNN. The window size is expressed in frames and was fixed to 11, the current frame and the 10 previous ones, which covers a time interval of a bit more than 1 second. More details on this parameter are clarified in the course of this section.

Additionally to the RNNs, networks with no memory units are used in the experiments. These networks take the outputs of the CNNs as input, just like the RNN, but they use fully-connected layers instead of the memory layers. The other parameters stay the same as for the recurrent networks. Networks using the fully-connected layers are hereafter referred to as *dense* networks.

The LSTM/GRU/Dense layers are followed by a batch normalization layer [40], a dropout layer [83] and a fully-connected layer to classify the action. A diagram can be found in Figure 5.6. An example LSTM network based on that diagram can easily be coded in Keras:

```
model = Sequential()
model.add(LSTM(num_cells, return_sequences=True,
          input_shape=(timesteps, timestep_dim)))
model.add(LSTM(num_cells))
model.add(BatchNormalization())
model.add(Dropout(dropout_rate))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop', metrics=['accuracy'])
```

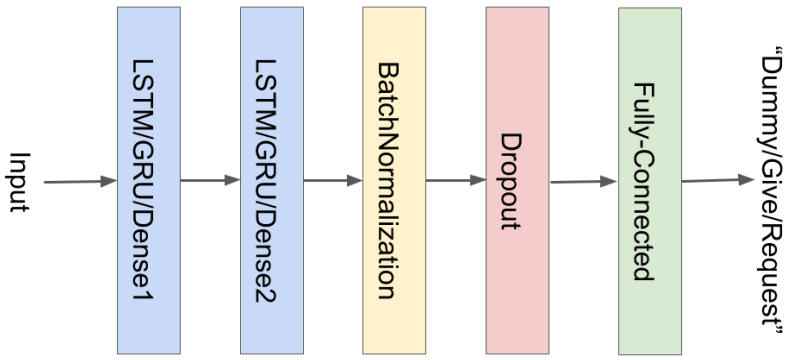


Figure 5.6: Structure of the temporal/dense component with 2 LSTM/GRU/Dense layers.

Modeling Sequences

For the first layer of the RNN the input shape needs to be specified. This shape is represented by a tuple consisting of the number of *timesteps*, i.e. the number of frames of the input sequence, and the *dimensionality* of each timestep. The sequences the recurrent network trains on should be the same as how they will be perceived in practice. The number of timesteps depends thus on the CNN, as they work at different frame rates, and the window size. The frame rates at which the CNNs work were investigated in Table 5.2. As the runtime of the RNNs is expected to be insignificant compared to the runtime of the object detection component, the frame rates were simply floored, resulting in 6 fps for both SSD models, 4 fps for the R-FCN model and faster_rcnn_resnet101 and 2 fps for faster_rcnn_inception_resnet. Which out of the 11 frames have been used to model the sequences is shown in Table 5.4. The sequences are made up of the current frame and the 10 previous frames.

	F_{t-10}	F_{t-9}	F_{t-8}	F_{t-7}	F_{t-6}	F_{t-5}	F_{t-4}	F_{t-3}	F_{t-2}	F_{t-1}	F_t
6 fps models	X		X		X	X		X		X	X
4 fps models	X		X			X		X			X
2 fps models	X					X					X

Table 5.4: Depiction of which frames from the time window are used for the different CNNs.

5.3.2 Training the Networks

Same as for the CNNs, the RNNs are trained on a Nvidia DGX-1. Before training the temporal component another evaluation of the convnets was conducted in order to verify that these are also able to achieve an accuracy of at least 50% on the EgoBaxter RNN data. For this purpose 320 frames from the RNN part of EgoBaxter were used as testing data. The CNNs were then trained on the whole CNN data, not on single personsplits. The outcomes of these tests are presented in Chapter 6.

After verifying that the convnets are suitable, the training data for the RNNs is created where the inputs consist of the outputs of the CNNs. As the same inputs are used for multiple epochs and do not change over the course of them, it was decided to write the outputs of the CNNs to disk in order to prevent the computation of the object detection component from becoming the bottleneck of the RNN training. Analogous to the labels of the recurrent networks, the inputs are written to a .txt file in a similar format of `filename : inputdata`. For the temporal component there was no need to transform the data into TFRecords, a simple parser for the input and label files already achieves the required results. An input sequence, consisting of the current frame and the 10 previous ones, is mapped to the label of the current frame.

Boxes

The bounding boxes output by the object detection component are the input to the RNNs. Each box consists of its four coordinates *xmin, ymin, xmax* and *ymax*, a *confidence score* and a *class label*. The spatial component always outputs exactly 100 boxes. Most of these have a very low confidence score as there are at most 3 objects of interest on one frame for EgoBaxter. An example can be seen in Figure 5.7, only the boxes where the confidence score is higher than 0.5 are displayed.

As it might be misleading for the network to input all 100 boxes, it was decided to only input one box for each object class. Specifically, the box with the highest confidence score over 0.5 for each class. If no such box is present, e.g. because the object is not visible or it is simply not detected accurately enough, the values for this box are set to 0. The input for the RNNs results thus in 18 values, 6 for each of the three boxes. These are further ordered by class, i.e. the first 6 values are represent the left hand, the next 6 the right hand and the last 6 the tool.

Training Settings

The networks are trained and evaluated on all the subsequences of length 11 from a scene. For example, if a scene is made up of 20 frames then a RNN using the outputs of a faster_rcnn_inception_resnet convnet trains on 10 subsequences of 3 frames (as only

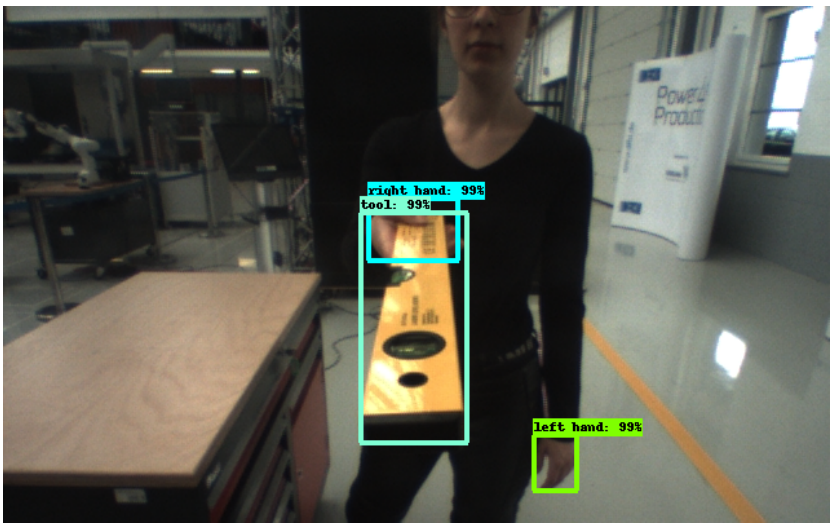


Figure 5.7: Example of bounding boxes on EgoBaxter. 100 boxes are output, but only those with a confidence score higher than 0.5 are displayed.

3 out of the 11 frames are used for faster_rcnn_inception_resnet, see Table 5.4). The dense networks are trained and evaluated on the outputs of single frames instead of sequences, and would thus train on all 20 frames from the example above.

As the recurrent networks are built from scratch they are initialized with random weights. Multiple instances of the same network trained on the same data with the same settings could produce different outcomes. Thus, it was decided to train several networks with the same settings to get a clearer idea of how they perform. Specifically, 5 networks are trained for each setting and they are then averaged over their accuracies. Training several networks with the same settings was not performed for the convnets for two reasons. First, they are not initialized with random weights but use a set of weights aquired from transfer learning. Second, even though the Nvidia DGX-1 allowed for a relatively fast training, it still took a considerable amount of time to train the CNNs using the different settings only once.

As for the convnets, leave-one-out cross-validation on the individuals is performed on the RNN data. In this case, in a non-exhaustive way, as 8 personsplits would take too much time. Thus, only 3 splits of different individuals were used. Again, analogously to the CNNs, hyperparameters like the number of units and the number of epochs are first identified on one personsplit. Afterwards, the networks are trained and evaluated on the remaining personsplits to get an idea of how well they perform in general.

5.3.3 Evaluating the Networks

The evaluation is again run in parallel with the training in order to measure the performance at different stages of the training process. However, there is no need to run two different scripts in Keras. It is sufficient to simlply call the *fit* function of the model:

```
1 tbCallBack = keras.callbacks.TensorBoard(log_dir='modellog')
2 model.fit(x=train_x, y=train_y, batch_size=batch_size,
3           epochs = num_epochs, callbacks=[tbCallBack],
4           validation_data=(test_x, test_y))
```

```
5 model.save('model.h5')
```

The TensorBoard `callback` writes an event file which logs the *training loss, training accuracy, testing loss* and *testing accuracy* to the folder specified by `log_dir`. As this is a classification problem the accuracy is simply measured by whether the correct label has been assigned to an input. The model itself is saved to the file specified by `model.save()` and can be reused with a simple `model = load_model()` command. A flowchart of the temporal component can be found in Figure 5.8.

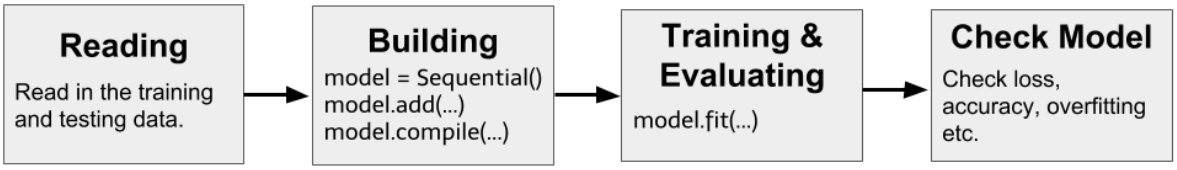


Figure 5.8: Flowchart of the temporal component. Unlike for the spatial component, everything can be done in one file in Keras.

From the results of this evaluation, which is representing the evaluation of the whole sequential architecture, not only the performance on EgoBaxter can be deduced, but also the SpatioTemporal Tradeoff can be determined. These results and the tradeoff are presented in the next chapter.

6 Evaluation

This chapter starts by giving an introduction to TensorBoard, a tool by TensorFlow to visualize different metrics, which was used in this thesis. Next, the evaluations of the spatial and temporal components are presented, before the SpatioTemporal Tradeoff on EgoBaxter is determined.

6.1 TensorBoard - A Visualization Tool

To make it easier to understand and optimize deep neural networks with TensorFlow, a visualization tool called TensorBoard is provided. This tool can display several metrics like the training loss or the testing accuracy, example images which are evaluated throughout the training process, or even a graph of the neural network model. TensorBoard can simply be called with:

```
tensorboard --logdir=dir_to_eventfile
```

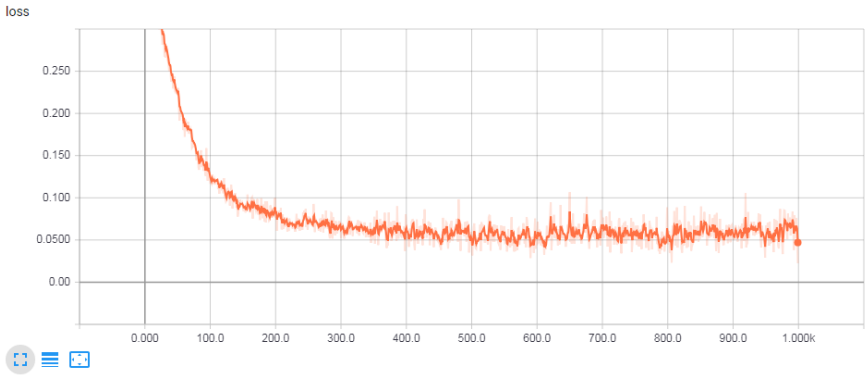
where `logdir` points to the directory where the event files, which are produced during the training and evaluation processes, are located.

Loss and Accuracy

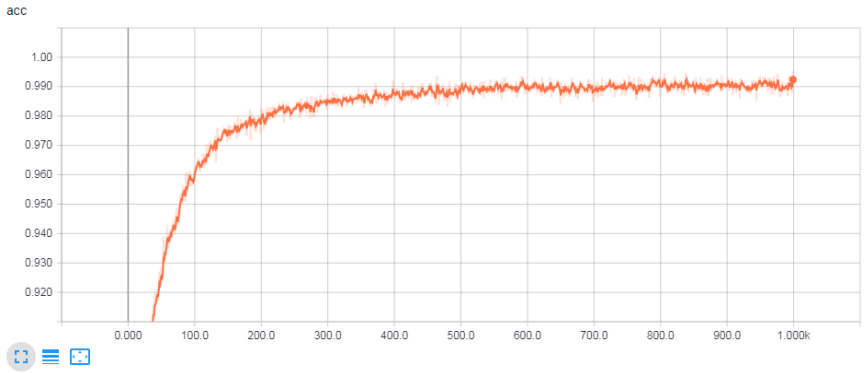
The two main metrics which are used to determine the performance of a network are the loss and the accuracy. The loss is usually inverse proportional to the accuracy, meaning that if the loss decreases the accuracy typically increases. An example can be seen in Figure 6.1. Both of these metrics can be measured for the training data as well as for the testing data.

The goal of the training is to adjust the weights of the network in order to minimize the training loss. The training accuracy is unsuited to determine how well the network performs in general as the model might have overfit on the training data, i.e. it has a very high accuracy on the training data, but performs badly on any other data (Section 4.3). For this purpose the network is frequently evaluated on a testing set. This data has no influence on the training, its sole purpose is to evaluate the network on new, unseen data in order to get an idea of how well it can perform in general. The testing accuracy can typically be used to represent the performance of a network. The TensorBoard interface displaying the 4 metrics of training loss, training accuracy, testing loss and testing accuracy can be seen in Figure 6.2.

On the left hand side of the interface the smoothing, which uses a simple moving average, of the curve can be controlled. The bigger the weight, the bigger the size of the moving average window. Smoothing allows to easier detect the general behaviour,



(a) Training Loss

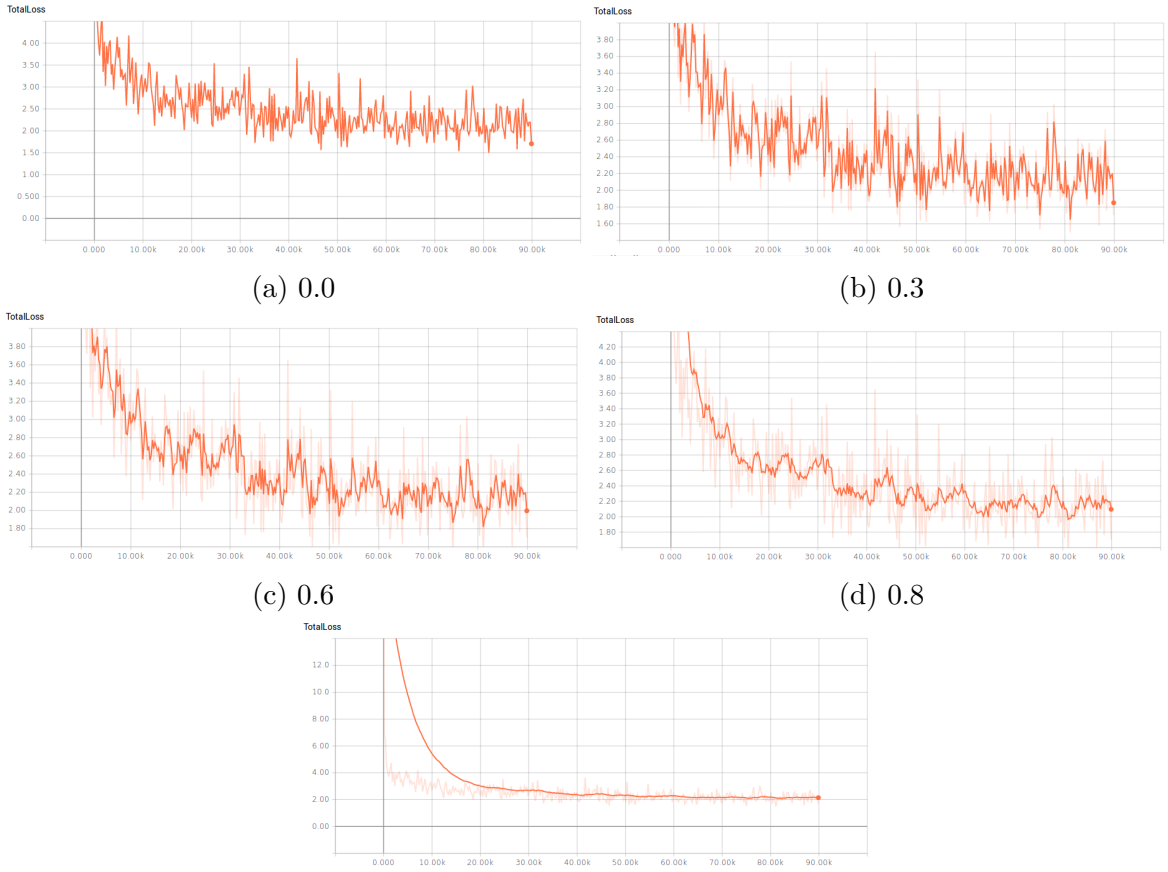


(b) Training Accuracy

Figure 6.1: Example of how the loss and accuracy typically progress throughout the training.

Figure 6.2: The interface of TensorBoard displaying the training loss (upper right graph), training accuracy (upper left), testing loss (bottom right) and testing accuracy (bottom left).

e.g. if a curve is decreasing or increasing, and is more robust to outliers. Smoothing with different weights can be seen in Figure 6.3.



(a) 0.0 (b) 0.3

(c) 0.6 (d) 0.8

(e) 0.95

Figure 6.3: Smoothing with different weights.

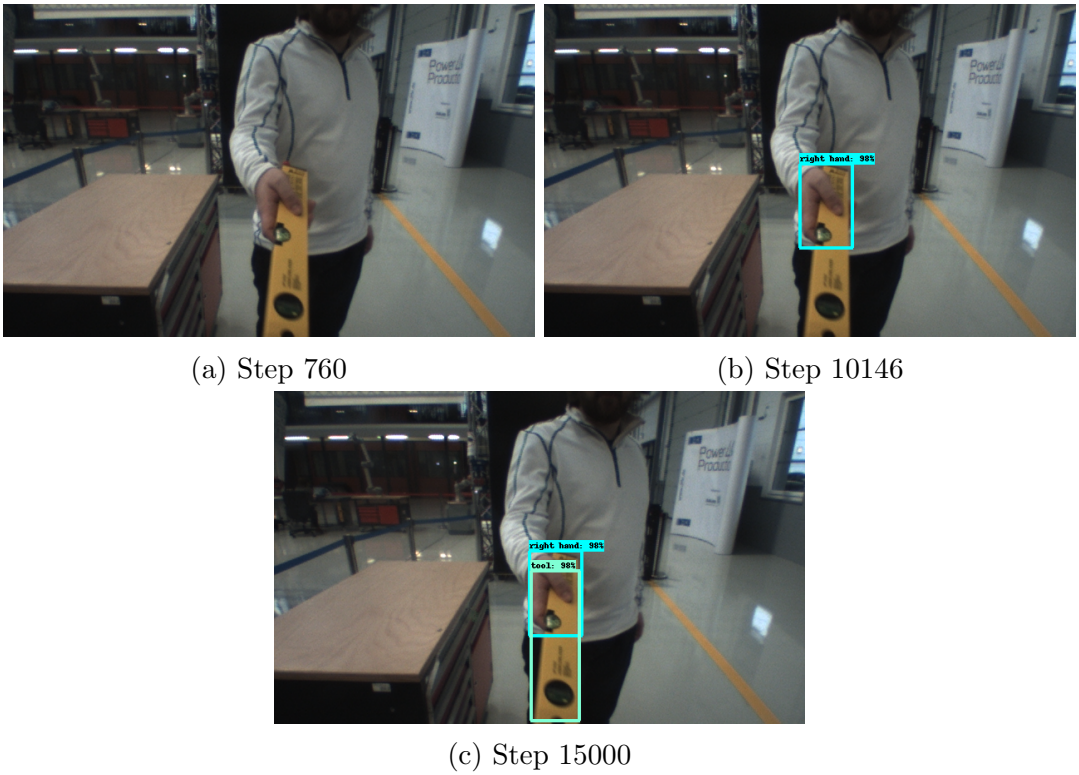
Example Visualizations

TensorBoard can also be used to visualize how the network performs at different steps of the training on example images from the test set. This feature was used during the training and evaluation of the CNNs as it helped to detect strengths and weaknesses of the network. An example can be seen in Figure 6.4. One can see how the convnet learns to detect the different object classes over the course of its training.

For the RNNs this feature could not be enabled as there are no images to visualize; the networks only train on the outputs of the CNNs and not the images themselves.

Graphs

TensorBoard also has a feature to display the model structure of the underlying network as a graph. This can be useful to understand the data flow, especially for the CNNs as they are not built from scratch, but based on existing architectures. Furthermore, the inputs and outputs of each node are displayed. This could be useful if one would for



(a) Step 760

(b) Step 10146

(c) Step 15000

Figure 6.4: Visualization of an example image from the testing set at different steps.

example like to know which features are used to make the predictions of a box. A part of such a graph can be seen in Figure 6.5.

6.2 Evaluation of the Spatial Component

The evaluation of the object detection component is based on the Pascal VOC detection metric [19] (see Section 4.3). For the CNNs, TensorBoard was used to log the training loss, the testing accuracy (mAP), and to display example visualizations on new data. The individual APs of the three object classes were also logged.

Hyperparameter - Number of Steps

As described in Section 5.2.1, it was decided to keep as many parameters as possible fixed, since the training of a CNN takes a considerable amount of time. The number of steps, which defines how long the training process lasts, depends a lot on the size of the dataset and needs to be identified empirically.

This value was identified by running the network for a high amount of steps and simply checking at which point of the training the testing accuracy converges, or, if the network overfits, checking at which point the accuracy reaches a peak before dropping. An example can be seen in Figure 6.6. The number of steps was identified on personsplit1 and was adopted for the other personsplits. The values for the different models can be found in Table 6.1.

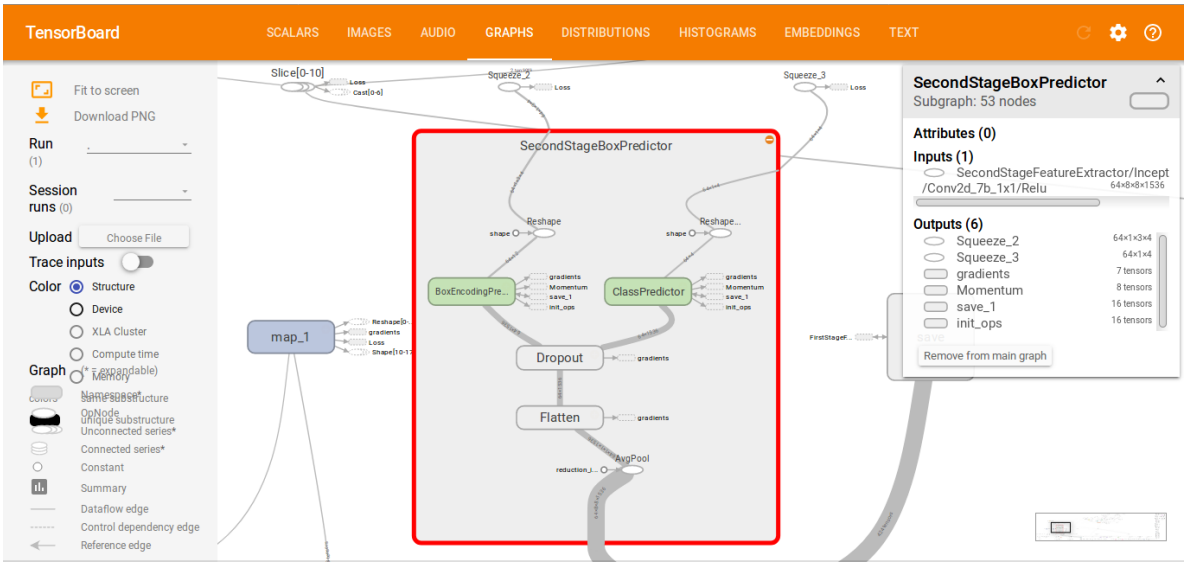


Figure 6.5: Part of the graph from the faster_rcnn_inception_resnet model.

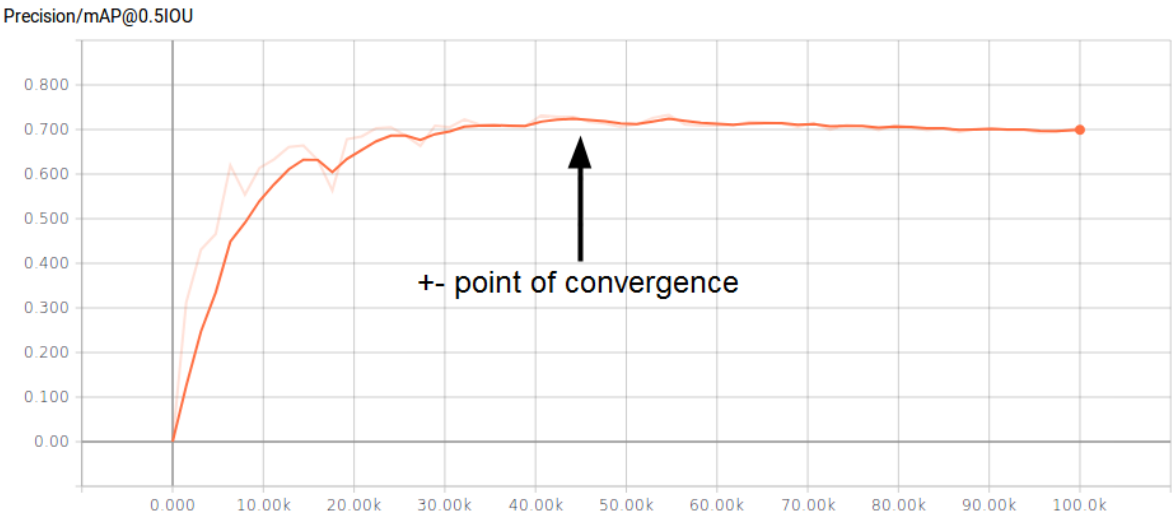


Figure 6.6: An example of how to identify the number of steps by checking when the model converges, respectively reaches its peak before overfitting. In this example, one can see a small drop in performance after the points of convergence. This might indicate overfitting, but could as well just be a small fluctuation as the decrease in performance is very small.

Modelname	Number of Steps
ssd_mobilenet	50k
ssd_inception	60k
rfcn_resnet101	30k
faster_rcnn_resnet101	40k
faster_rcnn_inception_resnet	15k

Table 6.1: The number of steps which was identified for the different models.

Performance of the Spatial Component

The performance of the spatial component can be deduced from the mAP metric, the higher the better. Furthermore, exhaustive leave-one-out cross-validation was applied in order to get a better general idea on how the models perform. The mAP of the different models on the 5 personsplits can be found in Table 6.2. The mAP averaged across the personsplits can be found in Table 6.3. The latter table essentially summarizes the evaluation of the spatial component

Modelname	Split1	Split2	Split3	Split4	Split5
ssd_mobilenet	0.7039	0.3302	0.6264	0.7644	0.7971
ssd_inception	0.7070	0.3408	0.5865	0.7115	0.8130
rfcn_resnet101	**0.8258**	**0.6253**	**0.8247**	0.9120	**0.9078**
faster_rcnn_resnet101	0.7975	0.5569	0.8234	**0.9181**	0.8919
faster_rcnn_inception_resnet	0.7514	0.5517	0.6972	0.8126	0.8200

Table 6.2: The mAPs of the different models on all the personsplits.

Modelname	Total mAP
ssd_mobilenet	0.6444
ssd_inception	0.6318
rfcn_resnet101	**0.8191**
faster_rcnn_resnet101	0.7976
faster_rcnn_inception_resnet	0.7266

Table 6.3: The mAPs of the different models averaged over all the personsplits.

From these tables a few observations can be made:

1. All the networks achieved the desired performance of over 50%, which means that their outputs can be used for the temporal component.

2. A similar ranking to the one in Table 4.2 was expected, however rfcn_resnet101 and faster_rcnn_resnet101 notably outperform faster_rcnn_inception_resnet, which was supposed to be the strongest model. Also rfcn_resnet101 performs better than faster_rcnn_resnet101 and ssd_mobilenet performs better than ssd_inception. However, the differences between these models are marginal (1-2%) in comparison. One reason for the different outcomes might be that distinct evaluation protocols were used. Table 4.2 uses the MS COCO detection metric, whereas Table 6.2 and Table 6.3 use the Pascal VOC detection metric. In the MS COCO evaluation, the APs are averaged over multiple IoU thresholds in the range of [0.5:0.05:0.95]. The high IoU values are rewarding for models which are very exact in their localization.
   Furthermore, for most of the parameters the networks adopted the default values which were already set in the config files. For EgoBaxter other values might achieve better results. However, due to the fact that the training takes a long amount of time, only the given subset of settings were used. The ranking from

Table 6.3 hints that rfcn_resnet101 and faster_rcnn_resnet101 will most probably perform better than faster_rcnn_inception_resnet in combination with the temporal component, as they both have the higher spatial accuracy as well as the higher processing speed.

3. All the models have a significant drop in performance on personsplit2. After further investigating this matter, it was observed that the networks performed relatively poor on both hands, especially the SSD models, as can be seen in Table 6.4. The individual, on which the networks were tested, was wearing short sleeves, whereas all of the participants from the training set were wearing long sleeves. The models could not cope well which this kind of new data and some of them even showed signs of overfitting. The problems that occured are visualized in Figure 6.7.

Modelname	Left Hand	Right Hand	Tool
ssd_mobilenet	0.2305	0.1050	0.6550
ssd_inception	0.2336	0.1062	0.6826
rfcn_resnet101	0.5049	0.5527	0.8183
faster_rcnn_resnet101	0.4951	0.4266	0.7490
faster_rcnn_inception_resnet	0.5886	0.4898	0.5767

Table 6.4: The APs of the three categories on personsplit2.

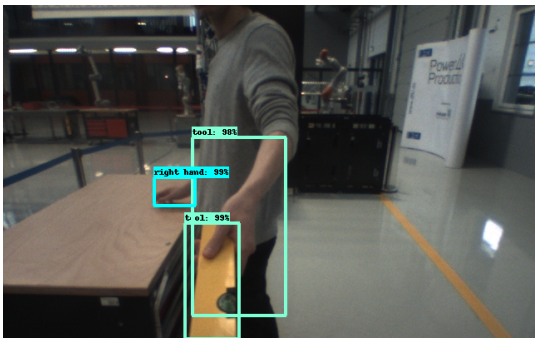
When taking a look at the example visualizations, further difficulties in detection seem to come from small object instances and motion blur (Figure 6.8). The latter causing difficulties is further supported by Dodge and Karam [16]. They state in their work that CNNs are susceptible to quality distortions in images, particularly to blur and noise. Small object instances were mainly an issue in the scenes were the human agent was not performing a handover, as the individual was occasionally standing further away from Baxter in these scenes than in the handover scenes.

Performance on the different Categories

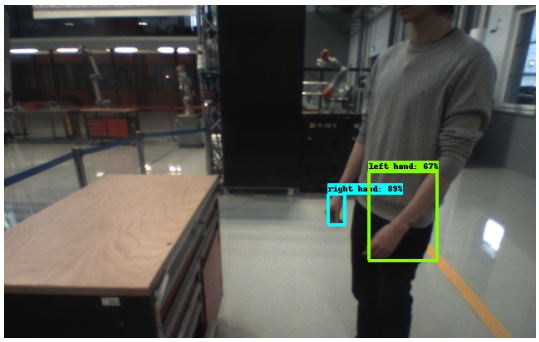
More observations could be made when taking a look at the AP for the three different object classes. The values for personsplit1 and personsplit3 are shown in Table 6.5, similar findings could be observed for personsplit4 and 5.

1. Concerning the hands, the left hand is getting detected better than the right hand. This might be due to the fact that the individuals were recorded from their left hand side. The left hand is thus better visible and larger than the right hand in most of the frames. Further, taking a look at the example visualizations, all the networks perform well in distinguishing both hands, although there are some occasional mix-ups.

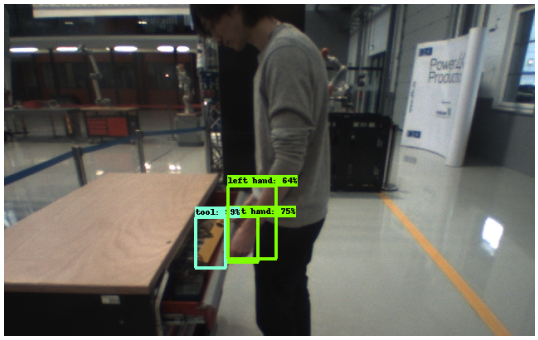
2. The left hand is detected the most accurately out of the three categories. The right hand and tool often have similar performances, e.g. the right hand has the higher AP on personsplit1 for most of the models, whereas the tool has the higher value on personsplit3.



(a) Detecting the forearm as the tool, as both are slim, longish and unicolor.

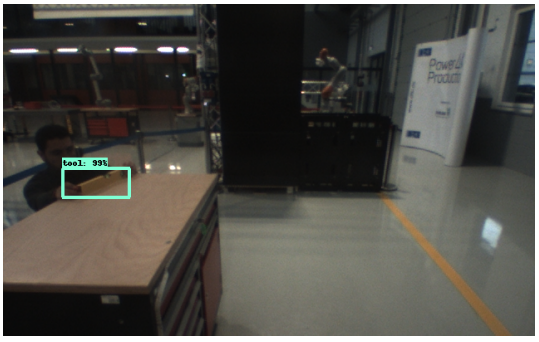


(b) Including the whole forearm in the bounding box of the hand.

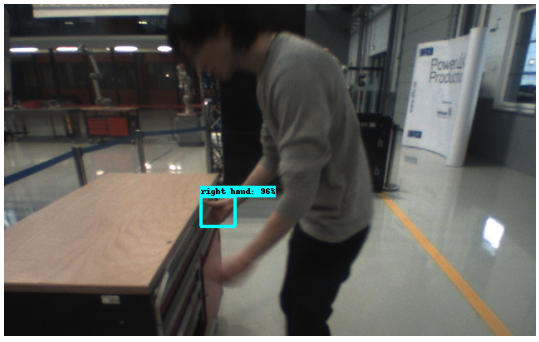


(c) Detecting the same hand several times.

Figure 6.7: Problems which occured on personsplit2.



(a) Small instances of both hands.



(b) Motion Blur.

Figure 6.8: Some cases where the networks have difficulties in detecting the object instances.

3. The faster_rcnn_inception_resnet model struggles with detecting the tool. Except for personsplit5, this model always has the lowest AP value for the tool out of all the models. This is rather surprising as it outperforms the SSD models when it comes to detecting the hands, and even achieves results close to rfcn_resnet101 and faster_rcnn_resnet101 on some splits. If not for the tool, the gap in performance to the latter two models might have been a lot smaller. A reason as to why this is the case was not found.

Split	Modelname	Left Hand	Right Hand	Tool
Personsplit1	ssd_mobilenet	0.8335	0.6803	0.5979
	ssd_inception	0.7912	0.6864	0.6435
	rfcn_resnet101	0.9015	0.8564	0.7196
	faster_rcnn_resnet101	0.8902	0.8181	0.6841
	faster_rcnn_inception_resnet	0.8814	0.7990	0.5500
Personsplit3	ssd_mobilenet	0.7218	0.5386	0.6188
	ssd_inception	0.7401	0.4674	0.5517
	rfcn_resnet101	0.8855	0.7514	0.8373
	faster_rcnn_resnet101	0.8981	0.7610	0.8111
	faster_rcnn_inception_resnet	0.8197	0.7335	0.5382

Table 6.5: The APs of the three categories on personsplit1 and personsplit3. Similar results could be observed for personsplit4 and 5.

Additional Experiments

Additional tests using more training data were conducted on personsplit1. Similar results are expected for the other personsplits, the explicit training runs were however omitted due to the time it would take to train all the networks. The additional training data consists of HandsTools and EgoHands (see Table 5.1) and was used in three different setups: 1) training on HandsTools and personsplit1, 2) training on EgoHands, HandsTools and personsplit1, 3) first training the model on EgoHands and then on HandsTools and personsplit1. The number of steps was increased accordingly to cope with the larger amount of data. For the third setup, the networks were first trained and evaluated on EgoHands, which was split into a training set of 4400 frames and a testing set of 400 frames. All of the models achieved a mAP of ~ 0.95 on EgoHands, and were then trained on the personsplit and HandsTools. For the second setup, all of the 4800 frames of EgoHands were included in the training. The evaluation results can be seen in Table 6.6.

The ssd_mobilenet performs best when using the additional data of HandsTools to train on. For faster_rcnn_inception_resnet adding EgoHands and HandsTools allowed for an improved performance. The other three models performed best without any additional data. In some of these cases, using additional data achieves close results (see rfcn_resnet101 split1 and setup1), in others the performance dropped notably (see ssd_inception split1 and setup3). None of the models performed best when training on EgoHands first and then on HandsTools and the personsplit. A definite conclusion in

Modelname	Split1	Setup1	Setup2	Setup3
ssd_mobilenet	0.7039	**0.7254**	0.7197	0.7176
ssd_inception	**0.7070**	0.6932	0.6918	0.6723
rfcn_resnet101	**0.8258**	0.8221	0.7989	0.8078
faster_rcnn_resnet101	**0.7975**	0.7780	0.7538	0.7727
faster_rcnn_inception_resnet	0.7514	0.7747	**0.7858**	0.7723

Table 6.6: The mAPs of the different models for the additional experiments.

the way of "the models need to train on this data to perform best" cannot be made. The cause behind these differences in performance has not been investigated.

Final Models

After confirming through cross-validation that the networks have a sufficient accuracy on EgoBaxter, the final models, whose outputs are used as the inputs for the RNNs, are trained. The training set for these models consists of all the scenes, i.e. using all of the EgoBaxer_CNN data without splits. As no clear conclusion could be made from Table 6.6, setup1 and setup2 were also run. The models were evaluated on a subset of frames from EgoBaxter_RNN, to confirm that the models also work well on this new data. In order to use these as evaluation data, they need to be labeled with ground truths. For this purpose, 320 frames, 2 from each of the 160 scenes, were labeled with bounding boxes. As these scenes consist only of handovers, small object instances are not an issue. Furthermore, it was avoided to choose frames with motion blur, which was noted to be not as present in EgoBaxter_RNN as it is in EgoBaxter_CNN. The results of the last training session can be seen in Table 6.7.

Modelname	EgoBaxter_CNN	Setup1	Setup2
ssd_mobilenet	**0.8785**	0.8654	0.8726
ssd_inception	0.8638	**0.8881**	0.8544
rfcn_resnet101	0.9069	**0.9238**	0.9088
faster_rcnn_resnet101	0.8963	0.8996	**0.9096**
faster_rcnn_inception_resnet	0.8162	0.8941	**0.9146**

Table 6.7: The mAPs of the final models which were evaluated on frames from the RNN dataset.

The numbers confirm that the models are suitable to provide input data to the RNNs. Due to the reasons stated above, the CNNs were expected to perform better on this data than on the one used in the cross-validation. The results from Table 6.3 give a better general idea on how the networks perform on EgoBaxter.

A conclusion of which data works best for the networks cannot be deduced from Table 6.7 either. The outcomes are even different than those from Table 6.6. Further investigations on this matter were however not conducted as the CNNs achieve the desired performance and can be used for the next step of training the temporal component. As the final models, those with the highest performance from Table 6.7 were chosen.

During the process of this thesis, over 60 CNNs have been trained and evaluated. The training of the networks took between 2 and 12 hours per network, depending on the network type and the amount of training data. Although there are a lot more different settings, some of which could potentially achieve better results, for this thesis, only the described subset of values for the parameters were used. Training more networks with different settings would take up a lot of time, and the models already reached the desired performance to be of use for the temporal component. Thus, it was decided to move on to the next step.

It should also be noted that even though the networks did not overfit, they can probably not be used for other datasets. This is due to the fact that they were trained and evaluated on EgoBaxter, which is a relatively small and limited dataset, e.g. as all the scenes were recorded at the same place. This also explains the relatively high accuracies. If general models were desired, a bigger dataset with a lot more variety in data would be required. However, the goal of this thesis is to provide a way on how to determine the SpatioTemporal Tradeoff and then demonstrate it on the example of EgoBaxter. Creating a huge dataset of several hundred-thousand images and building perfect object detection models for them is simply not needed for this thesis.

6.3 Evaluation of the Temporal Component

The evaluation of this component also represents the evaluation of the complete sequential architecture. The metrics which were logged for the temporal component are the training loss, training accuracy, testing loss and the testing accuracy. The accuracy is simply measured by the number of correctly classified subsequences divided by the number of all subsequences.

Hyperparameters

In a first step, the hyperparameters for the recurrent/dense networks need to be identified. These include the number of epochs, the number of layers and the number of units per layer. These paremeters are first determined on personsplit1, and are then adopted for the remaining two splits. For each of the following described settings, 5 networks were trained and the presented accuracies represent the average of these 5 runs.

First Iteration

For the very first iteration, the different parameters needed to be set to arbitrary values. The number of cells for the dense network was set to 2048, as it is a common number for fully-connected layers in CNNs, for the GRU and LSTM it was set to 32. The dropout rate was set to 0.2 (same as for the CNN), the number of layers to 2 and the networks were run for 1000 epochs.

During the first iteration, both the GRU and LSTM overfitted as can be seen in the example of Figure 6.9. The number of epochs was thus reduced to 100. The dense networks had high fluctuations in the testing loss and accuracy, however they showed no signs of overfitting. The fluctuations might be due to the high amount of units, which were reduced to 32 in the following test. The dropout rate was also increased to 0.4 in

all the following tests to prevent the overfitting and also reduce the high fluctuations in the dense networks.

Number of Epochs

For the number of epochs, the tested values differed from the dense network and the recurrent networks. For the dense network, values of 100 and 1000 were investigated, whereas for the recurrent networks values of 40 and 100 were tested. Some of the GRUs/LSTMs showed light signs of overfitting when using 100 epochs. Thus, the smaller value of 40 was further tested.

For the dense networks, 1000 epochs proved to be the better choice than 100 as can be seen in Table 6.8. Analogous results were observed for the recurrent networks where 100 turned out to be the better value.

Input from	#Units	Epochs	Accuracy
ssd_mobilenet	32	100	0.8998
		1000	**0.9106**
	16	100	0.8989
		1000	**0.9061**
faster_rcnn_inception_resnet	32	100	0.8924
		1000	**0.9315**
	16	100	0.9119
		1000	**0.9349**

Table 6.8: Comparison of the number of epochs for the dense networks. The number of layers was fixed to 2 for these tests.

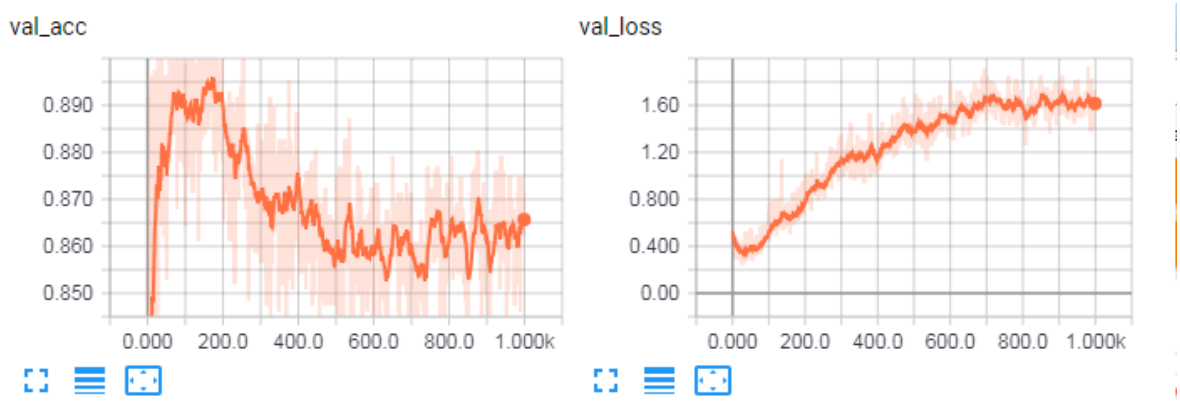


Figure 6.9: Testing accuracy (left) and testing loss (right) of an overfit LSTM from the first iteration. A high drop in the testing accuracy and a high increase in the testing loss are clearly visible. A smoothing weight of 0.9 was used.

Number of Layers

All the networks were tested with either 1 or 2 layers. The results are similar for the three types of networks: the accuracies between using 1 or 2 layers are close, but using 2 layers generally achieves the better results, as can be seen in Table 6.9.

Network Type	#Units	#Layers	Accuracy
Dense	32	1	0.9054
		2	**0.9106**
	16	1	0.9002
		2	**0.9061**
GRU	32	1	0.8866
		2	**0.8929**
	16	1	0.8863
		2	**0.8961**
LSTM	32	1	0.8898
		2	**0.8956**
	16	1	**0.8909**
		2	0.8816

Table 6.9: Comparison of the number of layers for the different types of networks. The number of epochs was fixed to 1000 for the dense and 100 for the recurrent networks. The inputs are from the ssd_mobilenet model.

Number of Cells

For the number of cells values of 16 and 32 were investigated. If the network had two layers then the number of cells was set to be the same in both of them. Some results can already be deduced from the comparison of the number of layers (Table 6.9), more explicit results can be found in Table 6.10. The number of cells does not have much influence on the outcomes as the accuracies are very close for both values (usually less than 0.5% difference).

Network Type	#Layers	#Units	Accuracy
Dense	1	16	0.9299
		32	**0.9312**
	2	16	**0.9349**
		32	0.9315
GRU	1	16	0.9171
		32	**0.9201**
LSTM	1	16	0.9128
		32	**0.9162**

Table 6.10: Comparison of the number of cells for the different types of networks. The number of epochs was fixed to 1000 for the dense and 100 for the recurrent networks. The inputs are from the faster_rcnn_inception_resnet model.

Final Settings

The final settings which were used on the different personsplits are summarized in Table 6.11. Analogously to the spatial component, more different settings for the different parameters could have been tested, some of which could potentially lead to better results. However, due to reasons of time, only the described subset of settings was used. For the temporal component, a total of around 400 networks have been trained and evaluated. Even though they do not need to train as long as the CNNs (training one network took at most 1h), the training and evaluation took a fair amount of time.

Network Type	#Epochs	#Layers	#Units
Dense	1000	2	32
GRU	100	2	32
LSTM	100	2	32

Table 6.11: Final settings of the networks which were used on the different personsplits to measure the performance of the temporal component.

Performance of the Temporal Component

The final results of the temporal component, and thus also the whole sequential architecture, can be found in Table 6.12 and Table 6.13. In the former the results across the three personsplits are shown, and in the latter the averages, which represent a summary of the whole evaluation, are displayed.

Network Type	Input from	Split1	Split2	Split3
Dense	ssd_mobilenet	0.9106	0.8199	0.8672
	ssd_inception	0.9108	0.7873	0.8678
	rfcn_resnet101	0.9404	**0.8320**	0.8893
	faster_rcnn_resnet101	**0.9468**	0.8155	0.8876
	faster_rcnn_inception_resnet	0.9315	0.8218	**0.9083**
GRU	ssd_mobilenet	0.8929	0.7744	0.8672
	ssd_inception	0.8982	0.7586	0.8725
	rfcn_resnet101	0.9246	0.7973	**0.8945**
	faster_rcnn_resnet101	**0.9337**	**0.8059**	0.8890
	faster_rcnn_inception_resnet	0.9271	0.7938	0.8811
LSTM	ssd_mobilenet	0.8956	0.7925	0.8357
	ssd_inception	0.8909	0.7650	0.8934
	rfcn_resnet101	**0.9317**	**0.8027**	0.8945
	faster_rcnn_resnet101	0.9250	0.7494	0.9067
	faster_rcnn_inception_resnet	0.9219	0.7876	**0.9095**

Table 6.12: Accuracies of the different network types across the personsplits.

Network Type	Input from	Total Accuracy
Dense	ssd_mobilenet	0.8659
	ssd_inception	0.8553
	rfcn_resnet101	**0.8872**
	faster_rcnn_resnet101	0.8833
	faster_rcnn_inception_resnet	**0.8872**
GRU	ssd_mobilenet	0.8448
	ssd_inception	0.8431
	rfcn_resnet101	0.8721
	faster_rcnn_resnet101	**0.8762**
	faster_rcnn_inception_resnet	0.8673
LSTM	ssd_mobilenet	0.8413
	ssd_inception	0.8498
	rfcn_resnet101	**0.8763**
	faster_rcnn_resnet101	0.8604
	faster_rcnn_inception_resnet	0.8730

Table 6.13: The averaged accuracies from the personsplits.

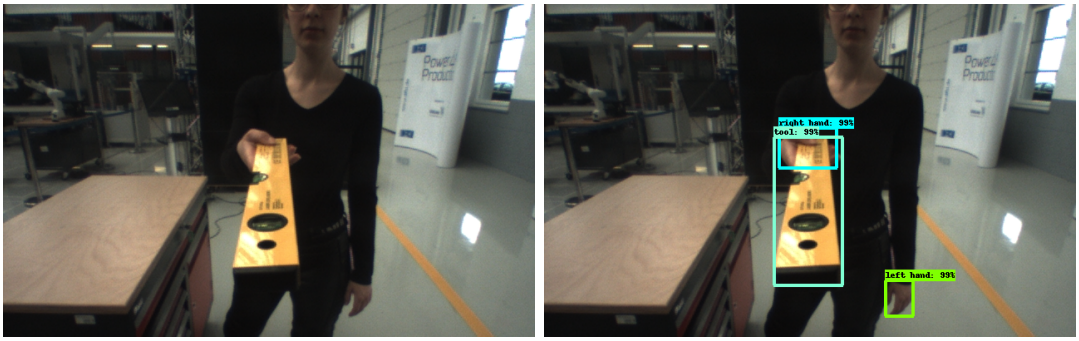
From these results, two major observations can be made:

1. Inputs from the rfcn_resnet101 and faster_rcnn_resnet101 CNNs achieve the best results, faster_rcnn_inception_resnet achieves comparable results. Both the SSD models perform notably worse. Thus, the models with the highest spatial accuracy achieved the best results. One should keep in mind though that both the rfcn_resnet101 and faster_rcnn_resnet101 CNNs are of medium speed. However, the fact that the faster_rcnn_inception_resnet of apparent intermediate spatial accuracy, but with the lowest frame rate achieves comparable results leads to the idea that spatial information is more important than temporal information. The second observation further confirms this statement.

2. The dense networks performed better than the recurrent networks in all combinations with the different CNNs. It is quite suprising that a network with no memory, which works on information from single frames alone, performs better than the RNNs which work on sequences. This further enhances the idea that spatial information is more important, as from this information alone the activities can be accurately classified.

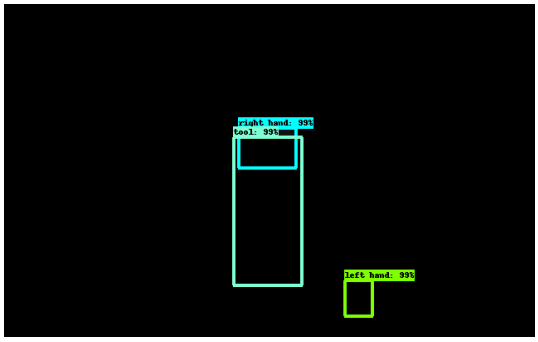
The total accuracies are remarkably high, considering that the inputs of the networks only consisted of the bounding boxes output by the CNNs (Figure 6.10). A simple classification rule such as, label an action as *Give* if the tool was present and as *Request* if it was absent is not enough, given the fact that in 57.75% of the frames no action took place, i.e. *Dummy*. The networks probably learned that the hand and tool bounding boxes need to be overlapping for a *Give* action, or that when a box of a hand is becoming bigger and moving towards the center it strongly signalizes the *Request* action.

Concerning both recurrent networks, there is no clear conclusion which of both performs better. For some settings the LSTM worked better, for others the GRU.



(a) Actual image.

(b) Actual image with boxes.

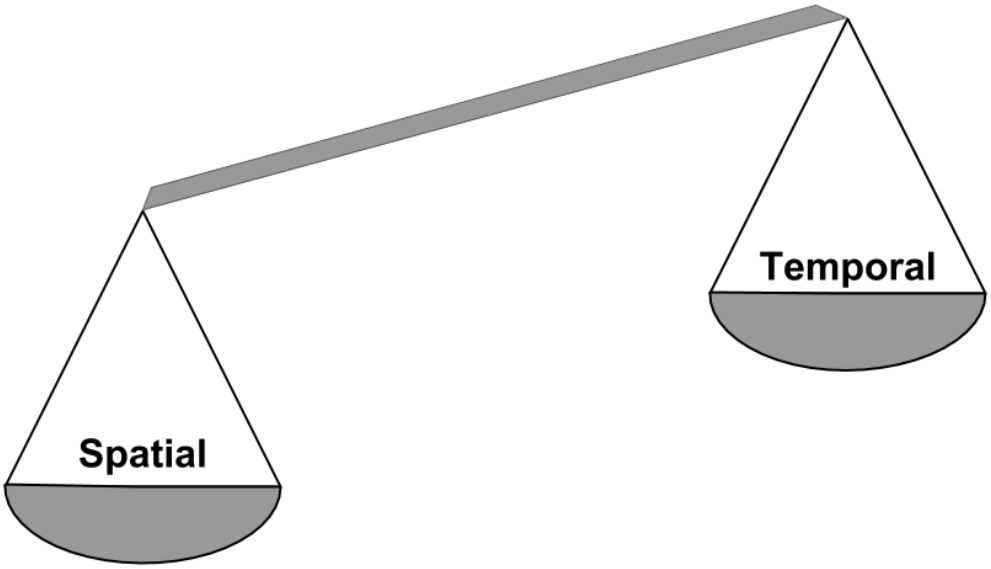


(c) What the network receives: only the boxes.

Figure 6.10: Visualization of which information from an image is passed to the recurrent/dense networks.

6.4 The SpatioTemporal Tradeoff

Based on the evaluation of the temporal component, which represents the evaluation of the complete sequential architecture, it is more important to focus on the single frames than on the temporal evolution of these. Thus, concerning the SpatioTemporal Tradeoff between the spatial and temporal dimension, reducing the temporal resolution for the sake of increased spatial accuracy achieves the better results. The tradeoff was determined for the EgoBaxter dataset in this thesis, in order to be able to make a more general conclusion the SpatioTemporal Tradeoff would need to be investigated on more different datasets.



7 Conclusion

In this thesis, a guideline on how to determine the SpatioTemporal Tradeoff has been provided. This process has then been elaborated on the newly created EgoBaxter dataset. For this purpose, more than 10,000 frames have been recorded and labeled with bounding boxes or action labels. The first component of the sequential architecture used in this thesis consists of CNNs with object detection. Five different network models with diverse accuracies and speeds have been trained and evaluated. The outputs of the CNNs are then input into the second component of the architecture: a RNN. For this component LSTMs and GRUs as well as dense networks with no memory have been investigated in combination with the different CNNs. Over the course of this thesis a total of more than 450 networks have been trained and evaluated.

The results from the evaluation indicate that focusing on the spatial information leads to the better overall performance. Thus, for EgoBaxter, the SpatioTemporal Tradeoff favors the spatial dimension. Although, in order to make a general statement on the SpatioTemporal Tradeoff for HAR scenarios, the tradeoff would need to be investigated on further datasets.

8 Future Work

The goal of this thesis was to provide a guideline on how to determine the SpatioTemporal Tradeoff and demonstrate it on the example of EgoBaxter. It would be interesting to explore this tradeoff on other datasets in order to be able to make a more general statement on the SpatioTemporal Tradeoff. Furthermore, in this thesis, only information about the bounding boxes has been used as input to the recurrent/dense networks. Using other kinds of input, e.g. features, would also be interesting to use.

Considering other models or approaches for the spatial and temporal component would also be worth investigating. For example, using CNNs with the You Only Look Once (YOLO) [66] object detection algorithm, especially since YOLO v3 [67] is apparently faster and more accurate than SSD. Using CNNs without object detection would also be an option, as there are HAR scenarios for which it is not needed.

Furthermore, information on the pose of the human agent could be used as additional spatial information. Recent approaches can even extract such information from RGB frames [64] [69] in weak real-time, so there would be no need for a depth camera on the robot.

Advances in hardware, especially GPU, will also allow to extract spatial information at higher frame rates.

Acknowledgements

I would like to thank Prof. Dr. Wahlster and the DFKI GmbH for the opportunity to write this thesis and for providing the required hardware and resources. I would also like to thank my colleagues from DFKI and my friends for all kinds of support throughout this thesis. Next, I would like to give my sincere thanks to my supervisor Christian Bürckert for guiding, supporting and encouraging me throughout this work. Finally, I would like to thank my father for supporting me.

Saarbrücken, 18th July 2018

Frank Baustert

List of Acronyms

ANN	Artificial Neural Network
AP	Average Precision
BRNN	Bidirectional Recurrent Neural Network
CNN	Convolutional Neural Network
DBN	Deep Belief Network
FCN	Fully Convolutional Network
GRU	Gated Recurrent Unit
HAR	Human Activity Recognition
HRC	Human Robot Collaboration
IoU	Intersection over Union
LSTM	Long Short-Term Memory
mAP	Mean Average Precision
MLP	MultiLayer Perceptron
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
R-FCN	Region-based Fully Convolutional Network
RNN	Recurrent Neural Network
RoI	Region of Interest
RPN	Region Proposal Network
SSD	Single Shot MultiBox Detector
SVM	Support Vector Machine
YOLO	You Only Look Once

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Action classification in soccer videos with long short-term memory recurrent neural networks. In Proceedings of the 20th International Conference on Artificial Neural Networks: Part II, ICANN'10, pages 154–159, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Sequential deep learning for human action recognition. In Proceedings of the Second International Conference on Human Behavior Unterstanding, HBU'11, pages 29–39, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017.

[5] Sven Bambach, Stefan Lee, David Crandall, and Chen Yu. Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions. In IEEE International Conference on Computer Vision (ICCV), 2015.

[6] Fabien Baradel, Christian Wolf, and Julien Mille. Pose-conditioned spatio-temporal attention for human action recognition. CoRR, abs/1703.10106, 2017.

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

[8] C. Braunagel, E. Kasneci, W. Stolzmann, and W. Rosenstiel. Driver-activity recognition in the context of conditionally autonomous driving. In 2015 IEEE 18th International Conference on Intelligent Transportation Systems, pages 1652–1657, Sept 2015.

[9] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. High accuracy optical flow estimation based on a theory for warping. In European Conference on Computer Vision (ECCV), volume 3024 of Lecture Notes in Computer Science, pages 25–36. Springer, May 2004.

[10] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In British Machine Vision Conference, 2014.

[11] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. CoRR, abs/1409.1259, 2014.

[12] François Chollet et al. Keras. `https://keras.io`, 2015.

[13] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR, abs/1412.3555, 2014.

[14] Corinna Cortes and Vladimir Vapnik. Support-vector networks. Mach. Learn., 20(3):273–297, September 1995.

[15] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. CoRR, abs/1605.06409, 2016.

[16] S. Dodge and L. Karam. Understanding how image quality affects deep neural networks. In 2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX), pages 1–6, June 2016.

[17] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In CVPR, 2015.

[18] Yong Du, Wei Wang, and Liang Wang. Hierarchical recurrent neural network for skeleton based action recognition. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015.

[19] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. International Journal of Computer Vision, 88(2):303–338, June 2010.

[20] Bernard Ghanem Fabian Caba Heilbron, Victor Escorcia and Juan Carlos Niebles. Activitynet: A large-scale video benchmark for human activity understanding. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 961–970, 2015.

[21] A. Fathi, X. Ren, and J. M. Rehg. Learning to recognize objects in egocentric activities. In CVPR 2011, pages 3281–3288, June 2011.

[22] Alireza Fathi, Yin Li, and James M. Rehg. Learning to recognize daily actions using gaze. In Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, Computer Vision – ECCV 2012, pages 314–327, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(9):1627–1645, Sept 2010.

[24] H. Foroughi, A. Naseri, A. Saberi, and H. Sadoghi Yazdi. An eigenspace-based approach for human fall detection using integrated time motion image and neural network. In 2008 9th International Conference on Signal Processing, pages 1499–1503, Oct 2008.

[25] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36:193–202, 1980.

[26] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with lstm. Neural Comput., 12(10):2451–2471, October 2000.

[27] Ross Girshick. Fast r-cnn. In Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15, pages 1440–1448, Washington, DC, USA, 2015. IEEE Computer Society.

[28] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR '14, pages 580–587, Washington, DC, USA, 2014. IEEE Computer Society.

[29] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, volume 15 of Proceedings of Machine Learning Research, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[30] Melvyn A. Goodale and A. David. Milner. Separate visual pathways for perception and action. Trends in Neurosciences, 15(1):20–25, 1992.

[31] A. Gupta, A. Kembhavi, and L. S. Davis. Observing human-object interactions: Using spatial and functional compatibility for recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(10):1775–1789, Oct 2009.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 770–778, June 2016.

[33] Donald O. Hebb. The organization of behavior: A neuropsychological theory. Wiley, New York, June 1949.

[34] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. Neural Comput., 18(7):1527–1554, July 2006.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural Comput., 9(8):1735–1780, November 1997.

[36] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences of the United States of America, 79(8):2554–2558, April 1982.

[37] J. Hosang, R. Benenson, and B. Schiele. Learning non-maximum suppression. In CVPR, 2017.

[38] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. CoRR, abs/1704.04861, 2017.

[39] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. CoRR, abs/1611.10012, 2016.

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

[41] H. Jhuang, T. Serre, L. Wolf, and T. Poggio. A biologically inspired system for action recognition. In International Conference on Computer Vision (ICCV), 2007.

[42] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. In Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10, pages 495–502, USA, 2010. Omnipress.

[43] Andrej Karpathy and Fei-Fei Li. Deep visual-semantic alignments for generating image descriptions. In CVPR, pages 3128–3137. IEEE Computer Society, 2015.

[44] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In CVPR, 2014.

[45] Alex Kendall, Vijay Badrinarayanan, , and Roberto Cipolla. Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding. arXiv preprint arXiv:1511.02680, 2015.

[46] L. Kratz and K. Nishino. Tracking pedestrians using local spatio-temporal motion patterns in extremely crowded scenes. IEEE Transactions on Pattern Analysis and Machine Intelligence, 34(5):987–1002, May 2012.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[48] H. Kuhne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre. Hmdb: A large video database for human motion recognition. In IEEE International Conference on Computer Vision (ICCV), 2011.

[49] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In Proceedings of the IEEE, pages 2278–2324, 1998.

[50] J. Lee and M. S. Ryoo. Learning robot activities from first-person human videos using convolutional future regression. In 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pages 472–473, July 2017.

[51] W. Li, Z. Zhang, and Z. Liu. Action recognition based on a bag of 3d points. In 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops, pages 9–14, June 2010.

[52] Y. Li, Zhefan Ye, and J. M. Rehg. Delving into egocentric actions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 287–295, June 2015.

[53] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. CoRR, abs/1405.0312, 2014.

[54] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector, pages 21–37. Springer International Publishing, Cham, 2016.

[55] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3431–3440, June 2015.

[56] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3431–3440, June 2015.

[57] M. Ma, H. Fan, and K. M. Kitani. Going deeper into first-person activity recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 1894–1903, June 2016.

[58] A. Manzi, L. Fiorini, R. Limosani, P. Dario, and F. Cavallo. Two-person activity recognition using skeleton data. IET Computer Vision, 12(1):27–35, 2018.

[59] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, Dec 1943.

[60] M. Müller, T. Röder, M. Clausen, B. Eberhardt, B. Krüger, and A. Weber. Documentation mocap database hdm05. Technical Report CG-2007-2, Universität Bonn, June 2007.

[61] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. In Computer Vision and Pattern Recognition, 2015.

[62] F. Ofli, R. Chaudhry, G. Kurillo, R. Vidal, and R. Bajcsy. Berkeley mhad: A comprehensive multimodal human action database. In 2013 IEEE Workshop on Applications of Computer Vision (WACV), pages 53–60, Jan 2013.

[63] S. J. Pan and Q. Yang. A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering, 22(10):1345–1359, Oct 2010.

[64] Paschalis Panteleris, Iason Oikonomidis, and Antonis A Argyros. Using a single rgb frame for real time 3d hand pose estimation in the wild. In IEEE Winter Conference on Applications of Computer Vision (WACV 2018), also available at arxiv., pages 436–445, lake Tahoe, NV, USA, March 2018. IEEE.

[65] Ning Qian. On the momentum term in gradient descent learning algorithms. Neural Netw., 12(1):145–151, January 1999.

[66] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. CoRR, abs/1506.02640, 2015.

[67] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv, 2018.

[68] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, NIPS'15, pages 91–99, Cambridge, MA, USA, 2015. MIT Press.

[69] Iasonas Kokkinos Rıza Alp Güler, Natalia Neverova. Densepose: Dense human pose estimation in the wild. arXiv, 2018.

[70] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. Nature, 323:533–, October 1986.

[71] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015.

[72] M. S. Ryoo and L. Matthies. First-person activity recognition: What are they doing to me? In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Portland, OR, June 2013.

[73] Christian Schüldt, Ivan Laptev, and Barbara Caputo. Recognizing human actions: A local svm approach. In Proceedings - International Conference on Pattern Recognition, volume 3, pages 32 – 36 Vol.3, 09 2004.

[74] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. Trans. Sig. Proc., 45(11):2673–2681, November 1997.

[75] Thomas Serre, Lior Wolf, and Tomaso Poggio. Object recognition with features inspired by visual cortex. In In CVPR, pages 994–1000, 2005.

[76] A. Shahroudy, J. Liu, T. T. Ng, and G. Wang. Ntu rgb+d: A large scale dataset for 3d human activity analysis. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 1010–1019, June 2016.

[77] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition, CVPR '11, pages 1297–1304, Washington, DC, USA, 2011. IEEE Computer Society.

[78] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1, NIPS'14, pages 568–576, Cambridge, MA, USA, 2014. MIT Press.

[79] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. CoRR, abs/1409.1556, 2014.

[80] Alan F. Smeaton, Paul Over, and Wessel Kraaij. Evaluation campaigns and trecvid. In MIR '06: Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval, pages 321–330, New York, NY, USA, 2006. ACM Press.

[81] Khurram Soomro, Amir Roshan Zamir, Mubarak Shah, Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. CoRR, page 2012.

[82] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. CoRR, abs/1412.6806, 2014.

[83] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014.

[84] M. Stone. Cross-validatory choice and assessment of statistical predictions. Journal of the Royal Statistical Society. Series B (Methodological), 36(2):111–147, 1974.

[85] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[86] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 2818–2826, June 2016.

[87] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[88] Jonathan Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1, NIPS'14, pages 1799–1807, Cambridge, MA, USA, 2014. MIT Press.

[89] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. Learning spatiotemporal features with 3d convolutional networks. In 2015 IEEE International Conference on Computer Vision (ICCV), pages 4489–4497, Dec 2015.

[90] J. R. Uijlings, K. E. Sande, T. Gevers, and A. W. Smeulders. Selective search for object recognition. Int. J. Comput. Vision, 104(2):154–171, September 2013.

[91] Subhashini Venugopalan, Marcus Rohrbach, Jeff Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence – video to text. In Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015.

[92] P. Vepakomma, D. De, S. K. Das, and S. Bhansali. A-wristocracy: Deep learning on wrist-worn sensing for recognition of user complex activities. In 2015 IEEE 12th International Conference on Wearable and Implantable Body Sensor Networks (BSN), pages 1–6, June 2015.

[93] Dibia Victor. Real-time hand tracking using ssd on tensorflow. `https://github.com/victordibia/handtracking`, 2017.

[94] J. Wang, Z. Liu, Y. Wu, and J. Yuan. Mining actionlet ensemble for action recognition with depth cameras. In 2012 IEEE Conference on Computer Vision and Pattern Recognition, pages 1290–1297, June 2012.

[95] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. Deep learning for sensor-based activity recognition: A survey. CoRR, abs/1707.03502, 2017.

[96] T. Xiang and S. Gong. Video behavior profiling for anomaly detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 30(5):893–908, May 2008.

[97] Mao Ye, Qing Zhang, Liang Wang, Jiejie Zhu, Ruigang Yang, and Juergen Gall. A Survey on Human Motion Analysis from Depth Data, pages 149–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[98] Serena Yeung, Olga Russakovsky, Ning Jin, Mykhaylo Andriluka, Greg Mori, and Li Fei-Fei. Every moment counts: Dense detailed labeling of actions in complex videos. arXiv preprint arXiv:1507.05738, 2015.

[99] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pages 3320–3328, Cambridge, MA, USA, 2014. MIT Press.

[100] Bowen Zhang, Limin Wang, Zhe Wang, Yu Qiao, and Hanli Wang. Real-time action recognition with enhanced motion vector cnns. CoRR, abs/1604.07669, 2016.

[101] R. Zhao, H. Ali, and P. van der Smagt. Two-stream rnn/cnn for action recognition in 3d videos. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 4260–4267, Sept 2017.