FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

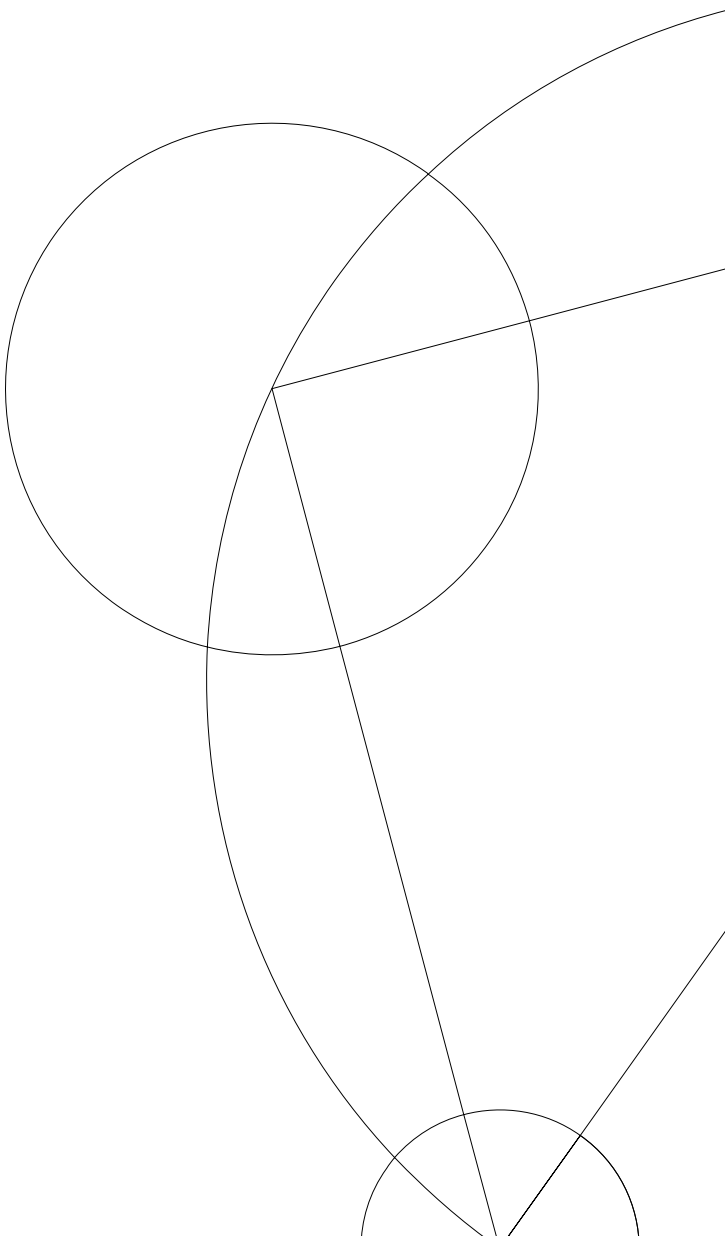
Master Thesis

2D Tracking in Climbing

Using Temporal Smoothing

André Oskar Andersen (wpr684)
wpr684@alumni.ku.dk

2023



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

Abstract

Preface

Acknowledgement

Contents

Notation

$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A matrix
$*$	The convolution operator
$\circ$	The Hadamard product

1 Introduction

1.1 Related Work

2-dimensional pose estimation can be divided into either being image-based or video-based, where the methods in the latter case use the tempoeral information of the video to perform the pose estimation.

Image-based methods were initially based on the geometry between the joints of the taget image [**6618926**, **10.1007/978-3-642-33715-4_19**, **6380498**]. Following this, were the convolutional-based methods, that used convolutional neural networks [**lecun1995convolutional**] to perform the pose estimation [**https://doi.org/10.48550/arxiv.1602.00134**, **https://doi.org/10.48550/arxiv.1603.06937**, **https://doi.org/10.48550/arxiv.1812.08008**, **https://doi.org/10.48550/arxiv.1703.06870**]. More recent methods use transformers [**https://doi.org/10.48550/arxiv.1706.03762**] to deliver state-of-the-art results [**https://doi.org/10.48550/arxiv.2204.12484**, **https://doi.org/10.48550/arxiv.2012.14214**].

Early video-based methods used 3-dimensional convolutions to capture the temporal information between neighboring frames [**https://doi.org/10.48550/arxiv.1506.02897**, **https://doi.org/10.48550/arxiv.171**
Other methods use LSTM's [**HochSchm97**] to capture this temporal information [**https://doi.org/10.48550/arxi**
https://doi.org/10.48550/arxiv.2001.08095]. Like in the case of image-based methods, transformers [**https://doi.org/10.48550/arxiv.1706.03762**] have recently been introduced to the video-based methods to capture the temporal information and deliver state-of-the-art-results [**https://doi.org/10.4855**

1.2 Problem Definition

1.3 Reading Guide

2 Deep Learning Theory

The following section covers the most important background theory for the experiments in Section **??**. This includes an introduction to various types of neural networks, as well as an introduction to the optimization of such networks.

2.1 Feedforward Neural Networks

Feedforward neural networks are the most basic type of neural networks. The aim of a feedforward neural network is to approximate some function f^*, by defining a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learning the parameters $\boldsymbol{\theta}$, that results in the best approximation of f^*. These models are called **feedforward** because there are no **feedback** connections in which the outputs of the model are fed back into itself. Instead, information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. Feedforward neural networks generally consists of multiple **layers**, arranged in a chain structure, with each layer being a function of the layer that preceded it [**DL_book**].

2.1.1 Fully-connected Layers

The most simple type of layer found in a feeforward neural network is the **fully-connected layer**. The fully-connected layer usually consists of some learnable parameter matrix $\boldsymbol{W}$ and learnable parameter vector $\boldsymbol{b}$, as well as a non-linear **activation function** g (which will be covered further in Section **??**). In this case, the i'th layer is defined as [**DL_book**]

$$\boldsymbol{h}^{(i)} = \begin{cases} g^{(i)}\left(\boldsymbol{W}^{(i)\top}\boldsymbol{h}^{(i)} + \boldsymbol{b}^{(i)}\right) & \text{if } i > 1 \\ g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) & \text{if } i = 1 \end{cases}. \tag{1}$$

2.1.2 Convolutional Layer

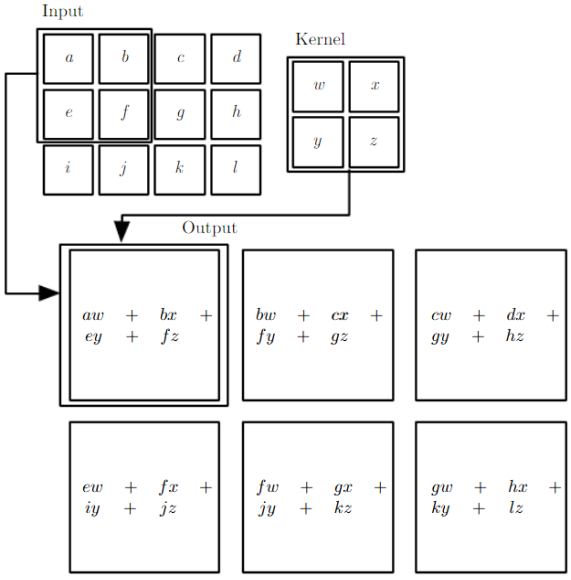


Figure 1: An example of applying a 2d kernel on an input [**DL_book**].

A **convolutional layer** is a specialized kind of feedforward layer, usually used in analysis of time-series or image data [**DL_book**]. If a network has at least one convolutional layer, it is called a **Convolutional neural network (CNN)**.

The convolutional layer consists of a set of **kernels**, each to be applied to the entire input vector, where each kernel is a learnable parameter matrix $k \times k$ [**everything**]. Each kernel is applied on the input to produce a **feature map**. The kernels are applied to the input by "sliding" over the input (where the step size is called **stride**). Each $k \times k$ grid of the input is then used to compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure **??**. [**bsc_thesis**]. To control the dimensions of the output, one might **pad** the sides with a constant value. Commonly, zero is used as the padding-value.

As seen in Figure **??**, each kernel produces a linear combination of all pixel values in a neighbourhood defined by the size of the kernel. Thus, unlike a fully-connected layer, a convolutional layer captures the high correlation between a pixel and its neighbours. Further, by limiting the size of the kernel, the network will use much fewer parameters, than if a fully-connected layer would be used instead [**DL_book**].

2.2 Recurrent Neural Networks

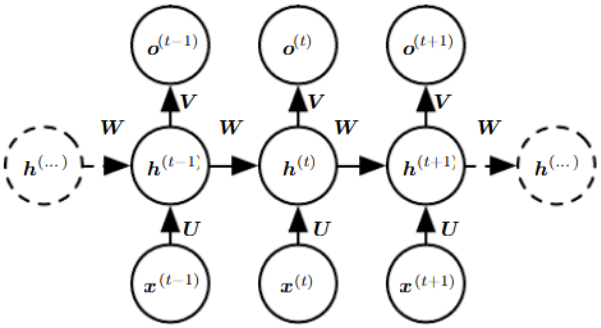


Figure 2: An illustration of an RNN [**DL_book**].

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data. Figure **??** illustrates the general setup of such a network, which maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. Generally, a RNN consists of three parts: (1) the input ($\boldsymbol{x}^{(i)}$), (2) the hidden state ($\boldsymbol{h}^{(i)}$), and (3) the output ($\boldsymbol{o}^{(i)}$). During inference, the model maps each input value to an out value, in a sequential matter, where it first maps the first input value, then the second, then the third, and so forth. The network maps each input value to an output value by making use of the hidden state from the preceding step, where the first hidden state has to be initiallized [**DL_book**].

2.2.1 Convolutional Long Short-Term Memory

One common recurrent neural network unit is the **convolutional long short-term memory (ConvLSTM) cell**, which is an adaptation of the standard **long short-term memory (LSTM) cell** for sequences of images.

The idea of the ConvLSTM cell is to create paths through time that have derivatives that neither vanish nor explode. This is done by accumulating information over a long duration. Once that information has been used, the cell *forgets* the old state, which the ConvLSTM cell learns to decide when to clear. Each cell has the same inputs and outputs as an ordinary reccurent network, but also has more parameters and a system of gating units that controls the flow of information [**DL_book**].

The ConvLSTM cell is defined as the following. Let $\boldsymbol{X}_1, \boldsymbol{X}_2, ..., \boldsymbol{X}_t$ be a sequence of input images, $\boldsymbol{C}_1, \boldsymbol{C}_2, ..., \boldsymbol{C}_t$ be a sequence of cell outputs, and $\boldsymbol{H}_1, \boldsymbol{H}_2, ..., \boldsymbol{H}_t$ be a sequence of hidden states. Then, at each input step t, we compute

$$\boldsymbol{i_t} = \sigma\left(\boldsymbol{W}_{xi} * \boldsymbol{X}_t + \boldsymbol{W}_{hi} * \boldsymbol{H}_{t-1} + \boldsymbol{W}_{ci} \circ \boldsymbol{C}_{t-1} + \boldsymbol{b}_i\right) \boldsymbol{f_t} = \sigma\left(\boldsymbol{W}_{xf} * \boldsymbol{X}_t + \boldsymbol{W}_{hf} * \boldsymbol{H}_{t-1} + \boldsymbol{W}_{cf} \circ \boldsymbol{C}_{t-1} + \boldsymbol{b}_f\right) \boldsymbol{C}_t = \tag{2}$$

2.2.2 Gated Recurrent Unit

2.3 Transformer

2.3.1 Encoder

2.3.2 Decoder

2.4 Training a Neural Network

2.4.1 Activation function

3 Models

The following section covers the theory behind the various models that will be introduced in Section **??**.

3.1 Mask R-CNN

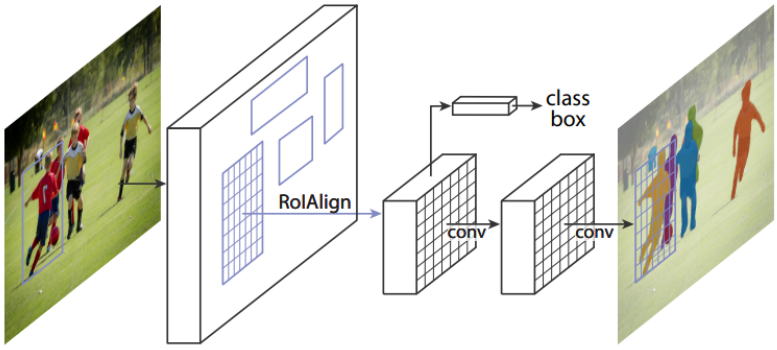


Figure 3: The Mask R-CNN framework for instance segmentation [**https://doi.org/10.48550/arxiv.1703.06870**].

NOTE: MANGLER NOK AT SKRIVE NOGET MED, AT INPUT ER ET BILLEDE OG LIGNENDE NOGET OM HVAD OUTPUT ER.
When we will be performing the pose estimation in Section **??**, our developed methods will be a variation of the *Mask R-CNN*, introduced by He *et al.* in 2018 [**https://doi.org/10.48550/arxiv.1703.06870**]. The following subsection explains the architecture of the Mask R-CNN and is based on an interpretation of He *et al.* [**https://doi.org/10.48550/arxiv.1703.06870**] and Zhang [**mask_rcnn_explained**].

The Mask R-CNN can generally be split into various components, which we will explain in further details in the following subsections.

3.1.1 The Backbone

The first major component of the Mask R-CNN is the *Backbone*, which is a network used for extracting the features of the input image. Commonly, a pretrained variation of the *residual network* (*ResNet*) is used [**https://doi.org/10.48550/arxiv.1512.03385**]. The backbone takes an image as input and returns a feature map.

3.1.2 Region Proposal Network (RPN)

The next major component of the Mask R-CNN is the *Region Proposal Network* (*RPN*). The RPN takes the feature map from the backbone as input, processes the feature map and proposes regions that may contain an object (the so-called *Region of Interests* or *RoI*) in the form of a feature map.

The RPN works by first processing the feature map with a convolutional layer that outputs a tensor with c channels, where each spacial vector (also with c channels) is associated with an anchor center. For each of these anchor centers a set of anchor boxes are generated. This convolutional layer is then followed by two 1×1 convolutional layers that independently processes this tensor. One of these 1×1 convolutional layers is a binary classifier that predicts whether each anchor box has an object. This is done by mapping each c-channel vector to a k-channel

vector. The other 1×1 convolutional layer is an object bounding-box regressor, which predicts the offsets between the true object bounding-box and the anchor box. This is done by makking each c-channel vector to a $4k$-channel vector. For the overlapping bounding-boxes of the same object, we keep the one with the highest objectness score and discard the rest.

3.1.3 Region of Interest Alignment (RoIAlign)

The third major components of the Mask R-CNN is the *Region of Interest Alignment* (*RoIAlign*). This components takes the proposed RoIs from the previous components as input and finds where each RoI is in the feature map. This is done by extracting feature vectors from the output feature map from the RPN and transform them into a fix-sized tensor.

3.1.4 Object Detection Branch

3.1.5 Mask Generation Branch

3.2 UniPose-LSTM

3.3 DeciWatch

baseball_pitch	baseball_swing	bench_press
bowling	clean_and_jerk	golf_swing
jumping_jacks	jump_rope	pull_ups
push_ups	sit_ups	squats
strumming_guitar	tennis_forehand	tennis_serve

Table 1: The original 15 action-types in the Penn Action dataset.

4 Dataset

LAV EN LISTE OF HVILKE JOINTS ER ANNOTERET I DE FORSKELLIGE DATASÆT

4.1 The BRACE Dataset

The second dataset we will be using is the *BRACE* dataset [**BRACE**]. We chose to use this dataset, as breakdancers tend be in acrobatic poses, similar to the ones that climbers tend to be in, making the poses relevant for our experiments in Section **??**.

This dataset consists of $1,352$ video sequences and a total of $334,538$ frames with keypoints annotations of breakdancers. The frames of the video sequences are in RGB and have a resolution of 1920×1080 [**BRACE**].

The frames of the video sequences have been annotated by initially using state-of-the-art human pose estimators to extract automatic poses. This was then followed by manually annotating bad keypoints, corresponding to difficult poses, as well as pose outliers. Finally, the automatic and manual annotations were merged, interpolating the keypoint seequence with Bézier curves. The keypoints is a list of 17-elements, following the COCO-format [**BRACE**].

4.2 The Penn Action Dataset

One of the dataset we will be using is the *Penn Action* dataset [**penn_action**]. This dataset consists of 2326 video sequences of 15 different action-types. Table **??** lists these 15 action-types [**penn_action**].

Each sequence has been manually annotated with human joint annotation, consisting of 13 joints as well as a corresponding binary visibility-flag for each joint. The frames of each sequence are in RGB and has a resolution within the size of 640×480 [**penn_action**].

Unlike the BRACE dataset, most of the poses in the Penn Action dataset are not very acrobatic and thus are not very relevant for the poses of climbers. For that reason, we have decided to focus on the action-types that may contain more acrobatic poses. Thus, we only keep the sequences that have `baseball_pitch`, `bench_press` or `sit_ups` as their corresponding action-type [**penn_action**].

4.3 The ClimbAlong Dataset

5 Experiments

6 Discussion

7 Conclusion

8 References