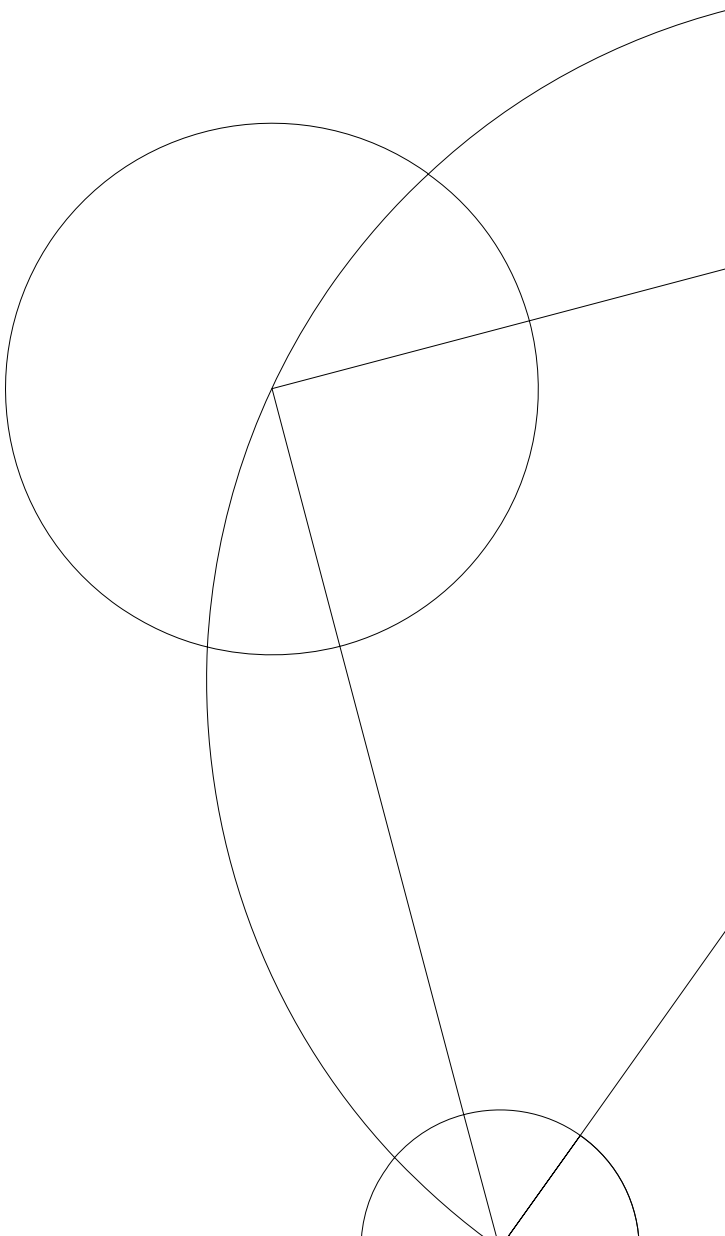
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Temporal Smoothing in 2D Human Pose Estimation for Bouldering

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

2023



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

Abstract

In this thesis we implement four architectures for extending an already developed keypoint detector for bouldering. The three architectures consist of (1) a single 3-dimensional convolutional layer followed by the ReLU activation function, (2) DeciWatch by Zeng *Et al.* [31], and (3) two kinds of bidirectional convolutional LSTMs inspired by Unipise-LSTM by Artacho and Savakis [3], where the difference between the two architectures lies in how they combine the two processing directions. The models are pretrained on the BRACE and Penn Action datasets and finetuned on a dataset for bouldering. The keypoint detector and the finetuning dataset are both provided by ClimbAlong at NorthTech ApS. We perform various experiments to find the optimal setting of the four models. Finally, we conclude, that DeciWatch by Zeng *Et al.* [31] yielding the most accurate results, one of the bidirectional convolutional LSTMs yielded the best rough estimations, as well as the simple 3-dimensional convolutional layer yielded the best results when also considering the size and prediction time of the model.

Preface

The following report is a thesis focusing on incorporating temporal smoothing in 2D human pose estimation for bouldering. The thesis was done in collaboration with ClimbAlong at NorthTech ApS and supervised by Kim Steenstrup Pedersen. It is written as part of the completion of my masters in computer science offered at the University of Copenhagen ending summer 2023 and covers the used methods, datasets and results.

Acknowledgement

First and foremost, I would like to thank my supervisor Kim Steenstrup Pedersen for both suggesting this exciting project and for supporting me throughout the project.

Secondly, I would like to thank Andrea Luongo and the rest of ClimbAlong at NorthTech ApS for providing me with the necessary data and models. Without them this project would not have been possible.

Last but not least, I would like to thank my parents, Stig and Elizabeth, as well as my sister, Isabella, for their never ending support and encouragement.

Contents

Notation

x	A scalar
$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A matrix
X	A tensor
$\boldsymbol{X}_{ij}$	Element located at row i column j in matrix $\boldsymbol{X}$
$*$	The convolution operator
$\circ$	The Hadamard product
σ	The sigmoid activation function
$\nabla_{\boldsymbol{x}} f$	Gradient of f with respect to $\boldsymbol{x}$
$f(x;\theta)$	Function f with input x and parameter θ
$\mathcal{U}(a,b)$	THe uniform distribution with support $[a,b]$
$\mathbb{R}$	The set of real numbers
$\mathbb{R}_{>0}$	The set of positive real numbers
$\mathbb{N}$	The set of natural numbers

1 Introduction

Video analysis in sports have throughout the last decade become more and more common, as these recordings contain a lot of important information. By analyzing such a video recording of people engaging in sports, we can for instance help a referee make the correct calls of help the people engaged in the sport improve their technique. However, most of these analyses requires the system to know where the relevant people are in the video recordings. The models that perform such a task have already been developed for the most popular sports, such as football or basketball, and tend to deliver very accurate results. On the orher hand, for the less popular sports, such as bouldering, such models are not as common

To perform video analysis, machine learning methods are often used, where they can for instance be used for estimating the pose of the people in the video. Often however, these models process the frames of an input video independently of each other, leading to suboptimal results. As the individual frames of a video contain some temporal information that correlates across the frames, one may incorporate this temporal information to improve the performance of the model that otherwise processes the frames independently of each other. Incorporating this temporal information is however not always straight forward and can sometimes require a lot of data, which does not align with the less popular sports, such as bouldering, where annotated data does not come in large quantities. Further, the poses and movements of the people in some of these sports are often very unlike most large public datasets, making these datasets unapplicable for these less popular sports.

Thus, the aim of this thesis is to implement various methods for extending an already developed keypoints detector for bouldering, such that it makes use of temporal smoothing for infering the position of the keypoints. This will be done by developing and testing various machine learning methods through multiple different experiments, such that we end up with the most optimal results.

The thesis is done in collaboration with ClimbAlong at NorthTech ApS. ClimbAlong provides an annotated dataset of video recordings of people climbing a bouldering wall. This dataset is not a synthetic dataset but instead a real dataset, making the developed results with this dataset be more realistic. Secondly, ClimbAlong provides an already developed and trained machine learning model for detecting the keypoints of a human in 2 dimensions of a given video recording. This model does not make use of temporal smoothing, but instead just process the frames independently of each other.

1.1 Related Work

2-dimensional articulated human pose estimation can be divided into two groups. The first group focuses on still images, where the processing of multiple images is done independently of each other. The second group focuses on video sequences, where the temporal information across the frames of a video sequence may be used for performing the pose estimation. These pose estimation methods for video sequences may use the pose estimation methods for still images for getting an initial pose estimation of the frames of a video sequence. Then later use temporal-inclusive methods for smoothing out these still image pose estimation [5].

Whether a human pose estimation method is for still images or a video sequence, the method can be categorized in various ways. First off, they can be categorized as being a generative method, which is model based, or a discriminative method, which is not model based. Secondly, they can be categorized as being a top-down method or a bottom-up method, based on which level they start the processing. Thirdly, they can be categorized as being regression-

based, where they directly map from input image to the body joint position, or as being detection-based, where they generate intermediate image patches or heatmaps of joint location. Lastly, they can be categorized as being one-stage, where end-to-end training is used, or multi-stage, where the human pose is estimated in multiple stages and are accompanied by intermediate supervision [5].

The early methods for articulated human pose estimation for still images focused on using the pictorial structures model [7], which consists of unary terms that model body part appearances and pairwise terms between adjacent body parts and/or joints, capturing their preferred spatial arrangement [21, 2, 14, 30].

More recent methods for articulated human pose estimation for still images are based on deep learning. Especially convolutional neural networks [17] tend to be very popular. AlexNet [16] was one of the earlies deep learning based methods for human pose estimation. In 2014 Toshev and Szegedy introduced DeepPose, which was an AlexNet-like deep neural network that learned joint coordinates from full images in a very straightforward manner without using any body model or part detectors [26, 5]. In 2015 Tompson *Et al.* intrdouced an architecture that included a 'pose refinement' model that estimates the joint offset location within a small region of an image, which was trained in combination with a convolutional neural network [25]. In 2017 He *Et al.* introduced the Mask R-CNN, which could perform both articulated pose estimation and segmentation simultaneously [11].

With the recent introduction of transformers [27] and vision transformers [6], recent still image methods have started to incorporate these techniques. For instance, in 2020 TransPose was introduced which used multiple transformer encoder layers to perform the pose estimation [29]. In 2022 ViTPose was introduced, which instead used a vision transformer for the pose estimation [28].

The incorporation of temporal information video sequences was firstly done by Simonyan *Et al.* who used a convolutional neural network for capturing this temporal information for action recognition [24]. This was rather quickly adapted for human pose estimation by Jain *Et al.* who also used a convolutional neural network for capturing the temporal information across the frames of a video sequence [20, 12]. Girdhar *Et al.* expanded on this idea by using a 3-dimensional convolutional layers to capture the temporal information [8].

Further, with the introduction of convolutional LSTM networks [23] in 2015, these types of networks were also experimented on articulated human pose estimation by Luo *Et al.*, who in 2018 achieved state-of-the-art results with a unidirectional convolutional LSTM [18]. This idea was expanded by Artacho and Savakis in 2020, who showed that their pose estimator for still images could deliver state-of-the-art results in video sequences by extending it with a unidirectional convolutional LSTM [3].

Like in the case of the still images, transformer-based networks have also started to been used to capture the temporal information for human pose estimation. One example of this is DeciWatch, introduced by Zeng *Et al.* in 2022, which efficiently delivers state-of-the-art results [31].

1.2 Choice of Models

As seen in section 1.1, there are generally three different approaches for incorporating the temporal information of the video into the pose estimator: (1) a 3-dimensional convolutional layer

based approach, (2) an approach based on a convolutional LSTM, and (3) a transformer-based approach. As we find all three approaches interesting and see potential in all of them, we will for each of the approaches be implementing a machine learning method.

For the 3-dimensional convolutional layer based approach, we will be implementing a very simple model, consisting of only a 3-dimensional convolutional layer followed by an activation function. We picked this model, as we needed something simple to be our baseline model and as we find it interesting whether a more complex model is even needed.

For the convolutional LSTM we let us inspire by the work of Artacho *Et al.* as they had a similar problem, where they aimed at extending a pose estimator, such that that it included temporal information, which ended up yielding state-of-the-art results [3]. However, we will be modifying their convolutional LSTM, as we see some potential shortcomings of their model.

Finally, for the transformer-based approach we will be implementing DeciWatch by Zeng *Et al.* [31]. We picked this model, as this model provides state-of-the-art results as well as it being very fitting for our project, as it is being described as being a model build on top of a keypoint detector for human pose estimators, such that it includes temporal information.

1.3 Reading Guide

The following section, section 2, covers the most relevant machine learning theory for this project. This is then followed by section 3, where we go into details with the various models that we will be developing. Section 4 then covers the used datasets. In section 5 and section 6 we describe the experiments we perform, as well as the preprocessing of the datasets. Finally, in section 7 we discuss the obtain results, as well as some of our shortcomings. Lastly, we conclude our results in section 8.

2 Background Theory

The following section covers the most important background theory for the experiments in section 5 and section 6. This includes an introduction to various types of neural networks, as well as an introduction to the optimization of such networks. In section 2.1 we go into the details of feedfoward neural networks, including the math behind fully-connected layers and convolutional layers. This is then followed by section 2.2, where we cover the general idea behind recurrent neural networks, as well as go into details with convolutional LSTMs. Next, in section 2.3 we cover the Transformer-architecture, as well as the attention mechanism. Lastly, in section 2.4 we cover the most central machine learning theory.

2.1 Feedforward Neural Networks

Feedforward neural networks are the most basic types 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 consist of multiple **layers**, arranged in a chain structure, with each layer being a function of the layer that preceded it [10].

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 and learnable parameter vector, as well as an **activation function**, which is a non-linear function, that makes it possible for the network to model non-linearity. Three of the most common activation functions are the **Rectified Linear Unit (ReLU)**, **Sigmoid** and **Hyperbolic Tangent (Tanh)** activation functions, defined as

$$\text{ReLU}(x) = \max\{0, x\} \tag{1}$$

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + exp(x)} \tag{2}$$

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{3}$$

Given the learnable parameter matrix $\boldsymbol{W}$, learnable parameter vector $\boldsymbol{b}$ and activation function g, the i'th fully-connected layer is defined as

$$\boldsymbol{h}^{(i)} = \begin{cases} g^{(i)}\left(\boldsymbol{W}^{(i)\top}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}\right) & \text{if } i > 1 \\ g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) & \text{if } i = 1 \end{cases}. \tag{4}$$

Thus, for a neural network with n layers, we have the mapping [10]

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{h}^{(n)}. \tag{5}$$

2.1.2 Convolutional Layer

A **convolutional layer** is a specialized kind of feedforward layer, usually used in analysis of time-series or image data. If a network has at least one convolutional layer, it is called a **Convolutional neural network (CNN)** [10].

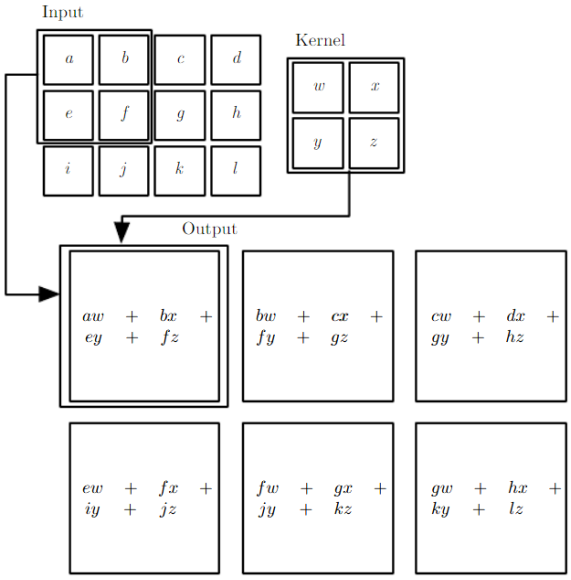


Figure 1: An example of applying a 2d kernel on an input [10].

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 $k \times k$ [22]. 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, as visualized in Figure 1. [1]. 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 1, 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 it was replaced by a fully-connected layer [10].

2.2 Recurrent Neural Networks

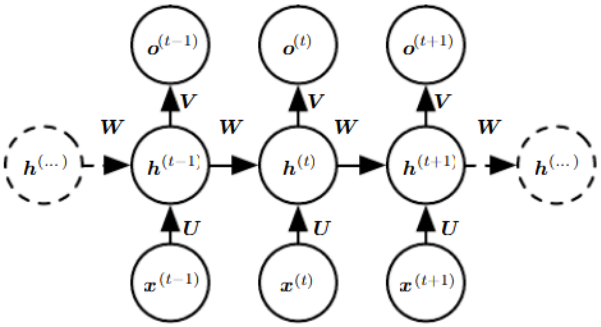


Figure 2: An illustration of an RNN [10].

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data. Figure 2 illustrates the general setup of such a network, which maps an input sequence to a corresponding output sequence. Generally, an RNN consists of three parts: (1) the input ($\boldsymbol{x}^{(t)}$), (2) the hidden state ($\boldsymbol{h}^{(t)}$), and (3) the output ($\boldsymbol{o}^{(t)}$). During inference, the model maps

each input value to an output 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. This kind of RNN is **unidirectional**, as it only considers the data in one sequence-direction. To utilize both directions of the sequence one can use two RNNs, where each RNN processes the data in opposite direction. This type of RNN is called being a **bidirectional** RNN (**bi-RNN**) [10].

2.2.1 Convolutional Long Short-Term Memory

One common recurrent neural network unit is the **convolutional long short-term memory (ConvLSTM)**, which is an adaptation of the standard **long short-term memory (LSTM)**, such that it implements convolutional layers.

The idea of both LSTM cells is to create paths through time that have derivatives that neither vanish nor explode. This is done by introducing a memory cell C_t, which accumulates state information over a long duration. This cell is accessed, written and cleared by several controlling gates. The model learns during training, when to access, write and clear this memory cell. By using the memory cell and gates to control the flow of information, the gradient will be trapped in the cell and thus be prevented from vanishing too quickly [10, 23].

The ConvLSTM cell consists of three gates. The first gate is the input gate I_t, which controls whether or not to accumulate the information of a new input to the cell. This gate is defined as

$$\mathsf{I}_t = \sigma\left(\mathsf{W}_{xi} * \mathsf{X}_t + \mathsf{W}_{hi} * \mathsf{H}_{t-1} + \mathsf{W}_{ci} \circ \mathsf{C}_{t-1} + \mathsf{B}_i\right) \tag{6}$$

where X_t is the current input matrix, H_{t-1} is the hidden state of the previous time step, C_{t-1} is the cell output of the previous time step, and W_{xi}, W_{hi}, W_{ci} and B_i are learnable parameters

The second gate is the **forget gate** F_t, which controls whether or not to "forget" the status of the cell output of the previous time step. The forget gate is defined as

$$\mathsf{F}_t = \sigma\left(\mathsf{W}_{xf} * \mathsf{X}_t + \mathsf{W}_{hf} * \mathsf{H}_{t-1} + \mathsf{W}_{cf} \circ \mathsf{C}_{t-1} + \mathsf{B}_f\right) \tag{7}$$

where W_{xf}, W_{hf}, W_{cf} and B_f are learnable parameters.

The last gate is the **output gate** O_t, which controls whether or not the latest cell output will be propagated to the final state H_t. This cell is defined as

$$\mathsf{O}_t = \sigma\left(\mathsf{W}_{xo} * \mathsf{X}_t + \mathsf{W}_{ho} * \mathsf{H}_{t-1} + \mathsf{W}_{co} \circ \mathsf{C}_t + \mathsf{B}_o\right) \tag{8}$$

where W_{xo}, W_{ho}, W_{co} and B_o are learnable parameters.

By combining the three gates, we get the following definition of the ConvLSTM Cell. Let $\mathsf{X}_1, \mathsf{X}_2, ..., \mathsf{X}_t$ be a sequence of input matrices. Then, at each time step t, we compute

$$\mathsf{I}_t = \sigma\left(\mathsf{W}_{xi} * \mathsf{X}_t + \mathsf{W}_{hi} * \mathsf{H}_{t-1} + \mathsf{W}_{ci} \circ \mathsf{C}_{t-1} + \mathsf{B}_i\right) \tag{9}$$

$$\mathsf{F}_t = \sigma\left(\mathsf{W}_{xf} * \mathsf{X}_t + \mathsf{W}_{hf} * \mathsf{H}_{t-1} + \mathsf{W}_{cf} \circ \mathsf{C}_{t-1} + \mathsf{B}_f\right) \tag{10}$$

$$\mathsf{C}_t = \mathsf{F}_t \circ \mathsf{C}_{t-1} + \mathsf{I}_t \circ \tanh\left(\mathsf{W}_{xc} * \mathsf{X}_t + \mathsf{W}_{hc} * \mathsf{H}_{t-1} + \mathsf{B}_c\right) \tag{11}$$

$$\mathsf{O}_t = \sigma\left(\mathsf{W}_{xo} * \mathsf{X}_t + \mathsf{W}_{ho} * \mathsf{H}_{t-1} + \mathsf{W}_{co} \circ \mathsf{C}_t + \mathsf{B}_o\right) \tag{12}$$

$$\mathsf{H}_t = \mathsf{O}_t \circ \tanh(\mathsf{C}_t) \tag{13}$$

For $t = 1$, both H_{t-1} and C_{t-1} have to be initialized [23].

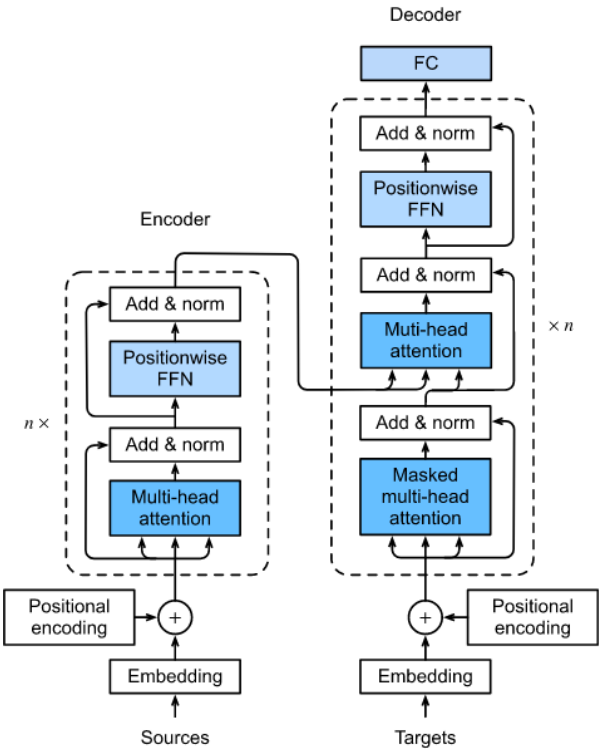


Figure 3: An illustration of the Transformer architecture [32].

2.3 Transformer

The sequential nature of RNNs precludes parallelization withing training examples, heavily slowing down the training of these models. Another type of model for sequential data is the **Transformer**, which eschews recurrence and instead relies on an **attention** mechanism to draw dependencies between input and output. The Transformer allows for more parallelization and can reach state of the art results [27].

2.3.1 Attention

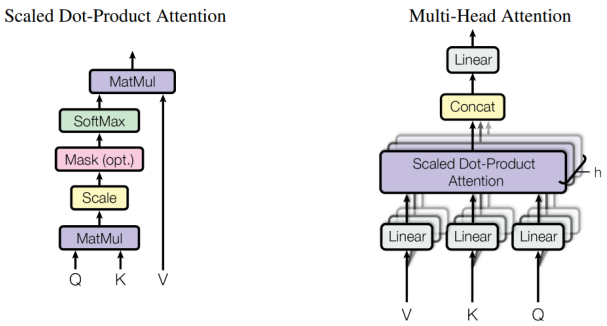


Figure 4: Illustration of (lef) the Scaled Dot-Product Attention, and (right) the Multi-Head Attention, which consists of several attention layers running in parallel [27].

An attention function is a function that maps a given query and a given set of key-value pairs to an output. In this function the query, keys, values, and output are all vectors, and the output is a weighted sum of the values, where the weight of each value if computed by a compatibility function of the query with the corresponding key [27]. Figure 4 illustrates the two types of

attention functions used in the Transformer.

Scaled Dot-Product Attention: The scaled dot-product attention is the core attention function of the transform. The function takes queries and keys of dimensions d_k as input, and values of dimension d_v. In practice, the attention function is computed on a set of queries, keys and values simultaneously, by packing them into matrices $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$, respectively. Thus, the scaled dot-product attention is computed as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{14}$$

where the scaling of $\boldsymbol{Q}\boldsymbol{K}^T$ is done to counteract the cases with a large d_k, which would result in the softmax-function having an extremely small gradient [27].

Multi-Head Attention: The **multi-head attention** is the used attention function by the Transformer and is an extention to the scaled dot-product attention. This attention function linearly projects the queries, keys and values h times with different, learned linear projections to d_k, d_k, and d_v dimensions, respectively. For each of the projected queries, keys and values, the scaled dot-product attention is applied, yielding d_v-dimensional output values. These output values are finally concatenated and further projected, resulting in the final values. By using multi-head attention, the model is allowed to jointly attend to information from different representation subspaces at different positions [27].

2.3.2 The Architecture

The Transformer follow an encoder-decoder structure, where the encoder maps an input sequence $\boldsymbol{x} \in \mathbb{R}^n$ to a sequence of continuous representations $\boldsymbol{z} \in \mathbb{R}^n$. The decoder then uses $\boldsymbol{z}$ to generate an output sequence $\boldsymbol{y} \in \mathbb{R}^m$, one element at a time. At each step the model consumes the previously generated output element as additional input when generating the next output [27]. Figure 3 illustrates the overall architecture of the Transformer.

Encoder: The encoder consists of $N \in \mathbb{N}$ identical layers, where each layer consists of two sub-layers. The first sub-layer is a multi-head **self-attention** layer, and the second sub-layer is a fully-connected feedforward network. Around each sub-layer is a **residual connection**, where the the input of the sub-layer is added to the output of the sub-layer. This residual connection is then followed by a round of **layer normalization**. In this self-attention layer the keys, values and queries come from the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder [27].

Decoder: The decoder consists of $M \in \mathbb{N}$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder also consists of a third sub-layer, which performs multi-head attention over the output of the encoder stack (called **cross-attention** [15]). Also here, is residual connections used around each of the sub-layers, followed by layer normalization. To ensure, that the predicitons for position i only depends on the known outputs at positions less than i, the self-attention sub-layer in the decoder stack is modified. This self-attention sub-layer, allows each position in the decoder to attend to all positions in the decoder up to and including that position. The self-attention sub-layer is modified by masking out all values in the input of the softmax which correspond to illegal connections [27].

Feedforward Networks: Each of the layers in the encoder and decoder contains a fully-connected feedforward layer, consisting of two linear transformations with the ReLU activation-function

applied in between, as described below [27]

$$FFN(\boldsymbol{x}) = \max\left(0, \boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1\right)\boldsymbol{W}_2 + \boldsymbol{b}_2. \tag{15}$$

Positional Encoding: As the model does not contain any recurrences nor any convolutions, it has to carry some other type of information to know about the relative or absolute position of the elements in the input sequence. This is done by adding **positional encodings** to the input of the encoder and decoder stacks. The positional encoding is a vector of size d_{model} and is defined as the following

$$\boldsymbol{PE}_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{16}$$

$$\boldsymbol{PE}_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{17}$$

$$\tag{18}$$

where *pos* is the position and i is the dimension [27].

2.4 General Machine Learning Terminology

The following section covers the general terminology and some algorithms, that will be used through this paper.

Loss Function: Training a machine learning model is generally done by minimizing a predefined **loss function**, which is used for measuring the error of the model. One common loss function for regression is the **Mean Squared error (MSE)**. Let $f^*(\boldsymbol{x})$ be a vector of ground truth observations and $f(\boldsymbol{x};\boldsymbol{\theta})$ be an estimation of $f^*(\boldsymbol{x})$. Then, MSE is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(f^*\left(\boldsymbol{x}^{(i)}\right) - f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right)\right)^2. \tag{19}$$

Thus, MSE measures the average squared difference between the ground truth observations and the estimated observations [13].

Optimizer: The minimization of the loss function is usually done by an **optimizer**, which is a type of algorithm, that is typically some type of variation of **gradient descent**. Gradient descent is a process that iteratively updates the parameters $\boldsymbol{\theta}$ of a function f by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}} L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right), f^*\left(\boldsymbol{x}^{(i)}\right)\right) \tag{20}$$

where η is called the **learning rate**, which controls the stepsize of each update, and L is a loss function such as MSE.

One of the most used optimizers is **ADAM**, which is an optimizer that uses **momentum** and **adaptive learning rate** to accelerate its learning. Momentum works by accumulating an exponentially decaying moving average of past gradients and continuing to move in their direction, which especially accelerates learning in the face of high curvature, small but consistent gradients, or noisy gradients. Adaptive learning rate is a mechanism for adapting individual learning rates for the model parameters. The idea is as follows; if the partial derivative of the

loss function, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If that partial derivate changes sign, then the learning rate should decrease. The algorithm uses a hyperparameter ρ for controling the length scale of the moving average The pseudocode of the algorithm has been illustrated in Algorithm 1. [10].

Online, mini-batch and batch gradient methods: Unlike gradient descent, ADAM bases its computations on a **mini-batch** of samples instead of basing it on a single sample. Thus, there are three ways of sampling data for the optimizer. The first way is done by sampling a single sample at each iteration. This class of methods is called **online gradient methods**. The advantage of these methods is, that each iteration is very quick, as we only have to compute the gradient of a single sample, however, the main disadvantage is, that the algorithm uses a lot of iterations. On the other hand we have **batch gradient methods** which uses the average gradient of all of the $n \in \mathbb{N}$ samples of the dataset to compute $\nabla L(f(\boldsymbol{x}, \boldsymbol{y}))$. By doing so the algorithm only requires one iteration, however, this iteration is much slower than each iteration of the online gradient methods. The last class of methods is **mini-batch gradient methods** which uses the average gradient of $B \in \mathbb{N}$ samples of the dataset for computing $\nabla L(f(\boldsymbol{x}, \boldsymbol{y}))$, where B is a predefined hyperparameter called the **mini-batch size**. This method lies in between the online gradient methods and batch gradient methods, as it uses fewer iterations than the online gradient methods, however, it uses more iterations than the batch gradient methods. Likewise, each iteration is faster to compute than it is for the batch gradient methods, but slower than it is for the online gradient methods. Further, online gradient methods can lead to a very noisy learning process due to the high variance of the computed gradients of the samples, which can make it more difficult for the optimizer to reach the minimum of the loss function. Mini-batch gradient methods and batch gradient methods removes this noise by averaging the computed gradient at each iteration, essentially leading to a quicker learning process [10].

Algorithm 1 Adam [10]

Require: Learning rate η
Require: Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$
Require: Small constant δ used for numerical stabilization
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}, \boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**

Sample a minibatch of $m \in \mathbb{N}$ random observations from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$
Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

Layer Normalization: One problem when optimizing a neural network is, that the gradients with respect to the weights of one layer depends on the outputs of the neurons of the previous

previous layer. If these outputs change in a highly correlated way, the model could experience some undesired "covariate shift", which would make the training take longer. One way to reduce these "covariate shifts" is to make use of **layer normalization**, where the layer normalization statistics over all the hidden units in the same layer l is computed as the following

$$\mu^{(l)} = \frac{1}{H}\sum_{i=1}^{H} {\boldsymbol{w}_i^{(l)}}^\top \boldsymbol{h}^{(l-1)}, \quad \sigma^{(l)} = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left({\boldsymbol{w}_i^{(l)}}^\top \boldsymbol{h}^{(l-1)} - \mu^{(l)}\right)^2} \tag{21}$$

where H is the amount of hidden units of the lth layer, $\boldsymbol{w}_i^{(l)}$ is the incoming learnable weight vector to the ith hidden unit of the lth layer, and $\boldsymbol{h}^{(l-1)}$ is the output of the preceding layer. Then, the summed inputs ${\boldsymbol{w}_i^{(l)}}^\top \boldsymbol{h}^{(l-1)}$ are normalized and rescaled by

$$\bar{a}_i = \frac{{\boldsymbol{w}_i^{(l)}}^\top \boldsymbol{h}^{(l-1)} - \mu^{(l)}}{\sigma^{(l)}}. \tag{22}$$

Further, the input to the current layer l also learns an adaptive bias b and gain g for each neuron after the normalization. Thus, the ith input to layer l is defined as the following

$$h_i^{(l)} = f\left(\frac{g_i(a_i - \mu^{(l)}) + b_i}{\sigma^{(l)}}\right) \tag{23}$$

where f is a non-linear activation function [4].

Evaluation: We often want to know how well a machine learning model performs on an unseen sample. This is commonly done by splitting the dataset into three non-overlapping subsets to avoid a biased evaluation. The first dataset is the **training** dataset which is used for training the machine learning model. The second dataset is the **validation** dataset which is used during training for evaluating various settings of the the used (hyper)parameters. The last dataset is the **testing** dataset, which is then used for evaluating the final machine learning model.

The evaluation of a machine learning model is done by making use of an **evaluation metric**, which depends on the task of the machine learning model. For pose estimation, the **Percentage of Correct Keypoints (PCK)** is commonly used. This metric measures the ratio of predicted keypoints that are closer to their corresponding ground-truth keypoints than some threshold. Often **PCK@0.2**, **PCK@0.1**, **PCK@0.05** ared used, which uses 20%, 10% and 5% of the torso diameter as its threshold, respectively [3].

Regularization Techniques: A machine learning model is said to be **overfitting** if the loss of the training dataset is decreasing, while the loss of the validation dataset is increasing. One can make use of a group of techniques called **regularization techniques** to avoid the overfitting of the network. One common regularization technique is **dropout**, where the hidden units of a neural network is randomly zeroed out with probability $p \in [0, 1]$ during training, making the network only consist of a subset of the original weights. By doing so the parameters are being trained on the dataset fewer times, leading to less of a probability of the model overfitting [32]. Another common regularization technique is **weight decay**, which aims at driving the weights of the model closer to the origin. This is done by adding a regularization term $\lambda\Omega(\theta) = \frac{\lambda}{2}||\boldsymbol{w}||_2^2$ for $\lambda \in \mathbb{R}_{>0}$ to the loss function. By doing so, the large weights are penalizied, resulting in the model learning simpler functions that are less likely to overfit [10].

Epoch: Often, the optimizer needs multiple "rounds" of using the whole training data for

reaching its minimum. Each of these rounds is called an **epoch**. One can pick the number of epochs to iterate through prior to training a model and just stop the training once the optimizer has used all of its epochs. Another methods is to use **early-stopping**, where one keeps track of the loss of the model on the validation-set for each epoch. Once the validation-loss has not decreased for $n \in \mathbb{N}$ consecutive epochs, the training is terminated.

3 Models

The following section goes into details with the models from section 1.2, that we will also be experimenting with in section 5 and section 6. The section starts off with section 3.1, where we describe the architecture behind the most simple models we will be implementing. Throughout section 3.2 we describe the architecture behind the two different bidirectional convolutional LSTMs that we will be experimenting with. This is then followed by section 3.3, where we cover the architecture of our Transformer-based model. Lastly, in section 3.4, we give a brief overview of the already developed keypoint detector.

3.1 3DConv

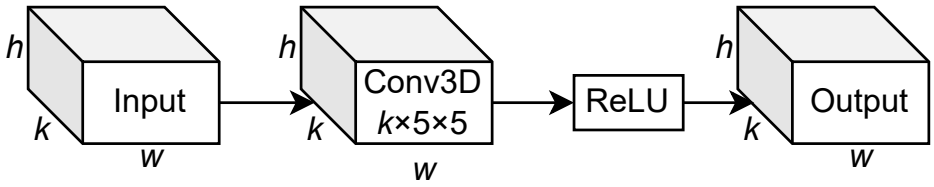


Figure 5: Illustration of 3DConv.

The first model we will be using is a very simple baseline model based on a 3-dimensional convolution. Throughout this project, we will be refering to this model as **3DConv**. Figure 5 illustrates the architecture of the model.

The model takes a sequence of $T \in \mathbb{N}$ estimated poses $\hat{\mathcal{P}} = \{\hat{\boldsymbol{P}}^t\}_{t=1}^{T}$ as input, where each estimated pose $\hat{\boldsymbol{P}}^t \in \mathbb{R}^{K \times h \times w}$ is represented using heatmaps, such that $K \in \mathbb{N}$ is the amount of keypoints in each estimated pose, $h \in \mathbb{N}$ is the height of each heatmap and $w \in \mathbb{N}$ is the width of each heatmap.

Once the data has been passed to the model, the processing of the data is very simple. As illustrated in Figure 5, the model starts by applying a 3-dimensional convolutional layer to the input data. The convolutional layer consists of $K \in \mathbb{N}$ filters, each with a kernel-size of $T \times 5 \times 5$. To ensure the input and output of the convolutional layer has the same shape, we pad the input with zeros.

Once the convolutional layer has processed the data, the ReLU activation-function is applied element-wise to the data, resulting in the final prediction of the model.

3.2 Bidirectional Convolutional LSTM (bi-ConvLSTM)

Our second and third model are based on the LSTM-extension of Unipose-LSTM by Artacho and Savakis [3]. Artacho and Savakis' model was based on a unidirectional convolutional LSTM. However, as we do not require our model to work in real-time, and as we believe a bidirectional convolutional LSTM would be benifical, our models will be based on a bidirectional convolutional LSTM. The major difference between our second and third model is how they combine the information of the two sequence directions.

Both models take a sequence of $T \in \mathbb{N}$ estimated poses $\hat{\mathcal{P}} = \{\hat{\boldsymbol{P}}^t\}_{t=1}^{T}$ as input, where each estimated pose $\hat{\boldsymbol{P}}^t \in \mathbb{R}^{K \times h \times w}$ is represented as a set of heatmaps, where $K \in \mathbb{N}$ is the amount of keypoints, $h \in \mathbb{N}$ is the height of each heatmap and $w \in \mathbb{N}$ is the width of each heatmap.

3.2.1 Model S

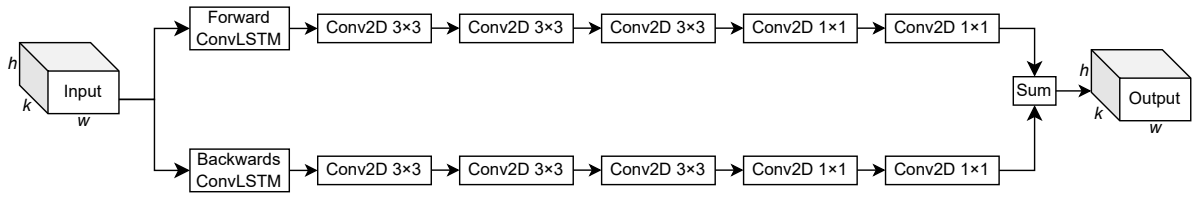


Figure 6: Illustration of the implemented bidirectional convolutional LSTM, where the two sequence orders are summed together.

Figure 6 illustrates the architecture of the second model. We will throughout this project refer to this model as **bi-ConvLSTM Model S**. The model starts by branching into two seperate branches, that processes the estimated poses in opposite sequence order. Each branch processes the estimated poses one frame at a time. This is done by first applying a convolutional LSTM to the input frame at time step $t \in \mathbb{N}$, using the preceding output of the convolutional LSTM as the hidden state. Each convolutional LSTM is followed by five 2-dimensional convolutional layers, each applying 128 filters, except for the last convolutional layer of each branch, which aplies K filters. The first three convolutional layers use a kernel size of 3×3, whereas the following two convolutional layers use a kernel size of 1×1. The outputs of the two branches are then summed together element-wise.

All convolutional layers use a stride of one and zero-padding on the input, such that the output of each convolutional layer has the same dimensions as the input to the convolutional layer.

3.2.2 Model C

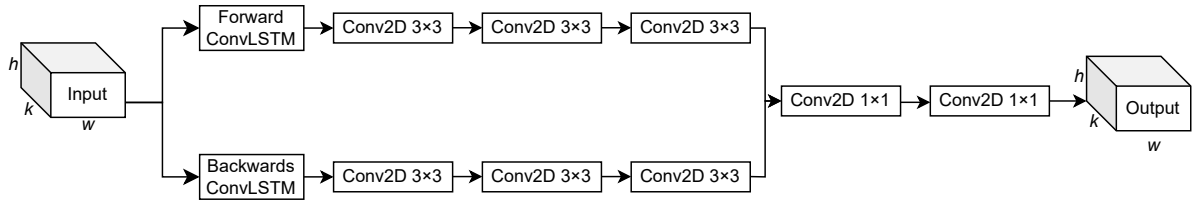


Figure 7: Illustration of the implemented bidirectional convolutional LSTM, where the two sequence orders are concatenated together and processed by two convolutional layers.

Looking at our second model, we see one major problem with the architecture behind it: the model does not have any option of prioritizing one processing order over the other, as the two branches are just summed together. Our third model aims at solving this problem.

Figure 7 illustrates the architecture of our third model. We will through this project refer to this model as **bi-ConvLSTM Model C**. This model is very similar to the previous model. It also starts off by branching into two seperate branches, that processes the estiamted poses in opposite sequence orders, where each branch processes the estimated poses one frame at a time. Similarly to the previous model, this model also starts off by applying a convolutional LSTM to the input frame, by using the preceding output of the convolutional LSTM as the hidden state. Further, each convolutional LSTM is also fillowed by three 2-dimensional convolutional layers, each applying 128 filters with a kernel size of 3×3.

Different from the previous model, this model concatenates the output of each branch to form

a $256 \times h \times w$ tensor, which is then processed by two 1×1 2-dimensional convolutional layers with 128 and K filters, respectively.

Similarly to the second model, all convolutional layers also use a stride of one and zero-padding on the input, such that the output of each convolutional layer has the same dimensions as the input to the convolutional layer.

3.3 DeciWatch

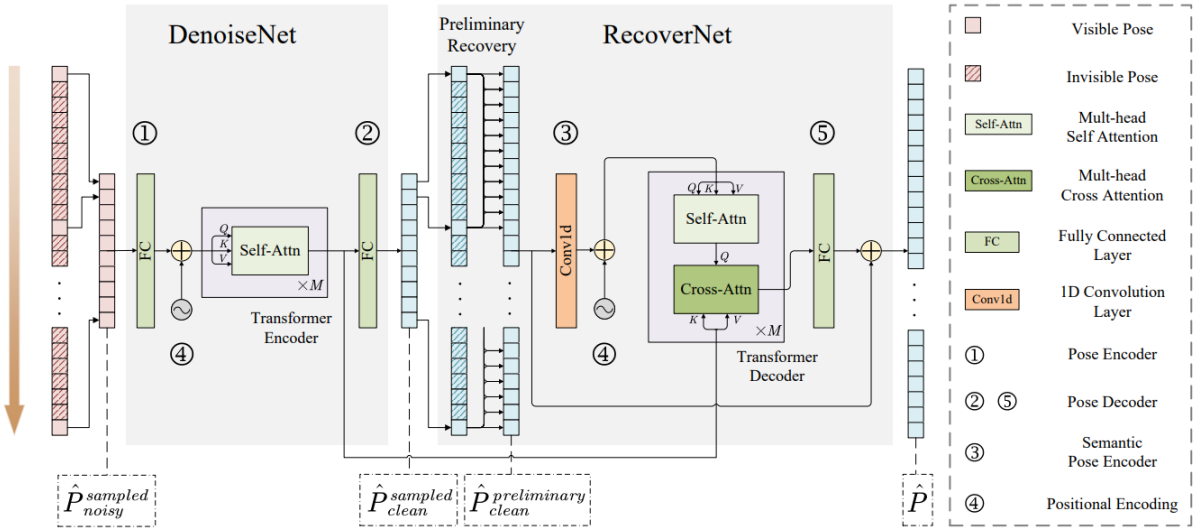


Figure 8: Illustration of the DeciWatch [31]

The last model we implement is a transformer-based model named *DeciWatch*, introduced by Zeng *Et al.* [31], which is illustrated in Figure 8. The model works by only processing some of the input-frames. It consists of two parts: the *DenoiseNet* and the *RecoverNet*. The aim of DenoiseNet is to denoise the estimated poses given as input to the model, whereas the aim of RecoverNet is to recover the poses of the missing frames. The following description of the model is based on an interpretation of the official paper behind DeciWatch [31].

More specifically, the model takes a sequences of $T \in \mathbb{N}$ estimated poses $\hat{\mathcal{P}} = \{\hat{\boldsymbol{P}}^t\}_{t=1}^{T}$ as input, where $\hat{\boldsymbol{P}}^t$ is represented by a 2-dimensional keypoint position. Due to redundancy in consecutaive frames and continuity of human poses, the model actually do not need all of the frames. Thus, the model starts by sampling every nth frame to select sparse poses $\hat{\boldsymbol{P}}_{noisy}^{sampled} \in \mathbb{R}^{\frac{T}{n} \times 2K}$, where $K \in \mathbb{N}$ is the number of keypoints. These sampled poses are then passed to DenoiseNet.

The goal of DenoiseNet is to denoise the sparse poses, that were estimated by a single-frame pose estimator. The denoise process can be formulated as

$$\hat{\boldsymbol{F}}_{clean}^{sampled} = \textbf{TransformerEncoder}\left(\hat{\boldsymbol{P}}_{noisy}^{sampled}\boldsymbol{W}_{DE} + \boldsymbol{E}_{pos}\right). \tag{24}$$

That is, $\hat{\boldsymbol{P}}_{noisy}^{sampled}$ is first encoded through a linear projection matrix $\boldsymbol{W}_{DE} \in \mathbb{R}^{2K \times C}$ and summed with a positional embedding $\boldsymbol{E}_{pos} \in \mathbb{R}^{\frac{T}{n} \times C}$. This is then passed to a transformer-encoder consisting of $M \in \mathbb{N}$ multi-head Self-Attention blocks, resulting in the noisy poses being embedded into a clean feature $\hat{\boldsymbol{F}}_{clean}^{sampled} \in \mathbb{R}^{\frac{T}{n} \times C}$, where $C \in \mathbb{N}$ is the embedding dimensions. Lastly,

another linear projection matrix $\boldsymbol{W}_{DD} \in \mathbb{R}^{C \times 2K}$ is used to obtain the denoised sparse poses

$$\hat{\boldsymbol{P}}_{clean}^{sampled} = \hat{F}_{clean}^{sampled} \boldsymbol{W}_{DD}. \tag{25}$$

After the sparse poses has been denoised as $\hat{\boldsymbol{P}}_{clean}^{sampled} \in \mathbb{R}^{\frac{T}{n} \times 2K}$, the data is passed to the RecoverNet, whose goal is to recover the absent poses. First, a linear transformation $\boldsymbol{W}_{PR} \in \mathbb{R}^{T \times \frac{T}{n}}$ is applied to perform preliminary sequence recovery to get $\hat{\boldsymbol{P}}_{clean}^{preliminary} \in \mathbb{R}^{T \times 2K}$ by

$$\hat{\boldsymbol{P}}_{clean}^{preliminary} = \boldsymbol{W}_{PR} \hat{\boldsymbol{P}}_{clean}^{sampled}. \tag{26}$$

To improve the recovery of the absent poses a transformer-decoder and positional embedding is used together with a 1D convolutional layer to bring temporal semantics into pose encoding to encode the neighboring $D \in \mathbb{R}$ frames' poses into pose tokens. Thus, RecoverNet, and the final prediction of DeciWatch, can be summarized by

$$\hat{\boldsymbol{P}} = \textbf{TransformerDecoder}\left(Conv1d\left(\hat{\boldsymbol{P}}_{clean}^{preliminary}\right) + \boldsymbol{E}_pos, \hat{\boldsymbol{F}}_{clean}^{sampled}\right) \boldsymbol{W}_{RD} + \hat{\boldsymbol{P}}_{clean}^{preliminary} \tag{27}$$

where $\boldsymbol{W}_{RD} \in \mathbb{R}^{C \times 2K}$ is another linear transformation layer. Further, as illustrated by Figure 8, key information is drawn in the the Cross-Attention block by leveraging the denoised features $\hat{\boldsymbol{F}}_{clean}^{sampled}$.

To avoid overfitting, dropout is applied to the input of each sub-layer and sums of the embeddings of the transformer-encoder and transformer-decoder, as well as to the positional encodings.

3.4 Mask R-CNN for Human Pose Estimation

The last model we will be using is the *Mask R-CNN for human pose estimation* as introduced by He *Et al.* in 2017 [11]. Unlike the previously described model, we will not be implementing this model, but instead use an already developed model on the ClimbAlong dataset, which has been provided by ClimbAlong A/S. As we will not be implementing this model, we will here not go into the details of how the model works, but instead just give a brief overview of the model. We will throughout this project be refering to this already developed model as the **keypoint detector**.

The model takes a single rgb-image with height $h_{in} \in \mathbb{N}$ and width $w_{in} \in \mathbb{N}$ as input and returns $K \in \mathbb{N}$ heatmaps of height $h_{out} \in \mathbb{N}$ and width $w_{out} \in \mathbb{N}$ such that each heatmap corresponds to a single keypoint where the position of the element with the greatest value corresponds is the location of the predicted keypoint. The Mask R-CNN by ClimbAlong A/S uses $h_{out} = w_{out} = 56$ and $K = 25$.

Each heatmap corresponds to the predicted bounding-box of the person that the keypoints corresponds to. Further, this bounding-box has been resized such that it is squared, discarding the aspect ratio of the original bounding-box.

4 Dataset

To perform the pose estimation in section 5 and section 6, we need some data on which to train, validate and test our models. The following section describes the datasets that will be used. This starts off in section 4.1, where we describe the dataset provided by ClimbAlong. This is then followed by section 4.2, where we describe the datasets that we will be using during pretraining of our models.

Keypoint label	ClimbAlong	BRACE	Penn Action
Head	No	No	Yes
Nose	Yes	Yes	No
Left ear	Yes	Yes	No
Right ear	Yes	Yes	No
Left eye	No	Yes	No
Right eye	No	Yes	No
Left shoulder	Yes	Yes	Yes
Right shoulder	Yes	Yes	Yes
Left elbow	Yes	Yes	Yes
Right elbow	Yes	Yes	Yes
Left wrist	Yes	Yes	Yes
Right wrist	Yes	Yes	Yes
Left pinky	Yes	No	No
Right pinky	Yes	No	No
Left index	Yes	No	No
Right index	Yes	No	No
Left thumb	Yes	No	No
Right thumb	Yes	No	No
Left hip	Yes	Yes	Yes
Right hip	Yes	Yes	Yes
Left knee	Yes	Yes	Yes
Right knee	Yes	Yes	Yes
Left ankle	Yes	Yes	Yes
Right ankle	Yes	Yes	Yes
Left heel	Yes	No	No
Right heel	Yes	No	No
Left toes	Yes	No	No
Right toes	Yes	No	No

Table 1: Overview of the annotated keypoints of the three used datasets

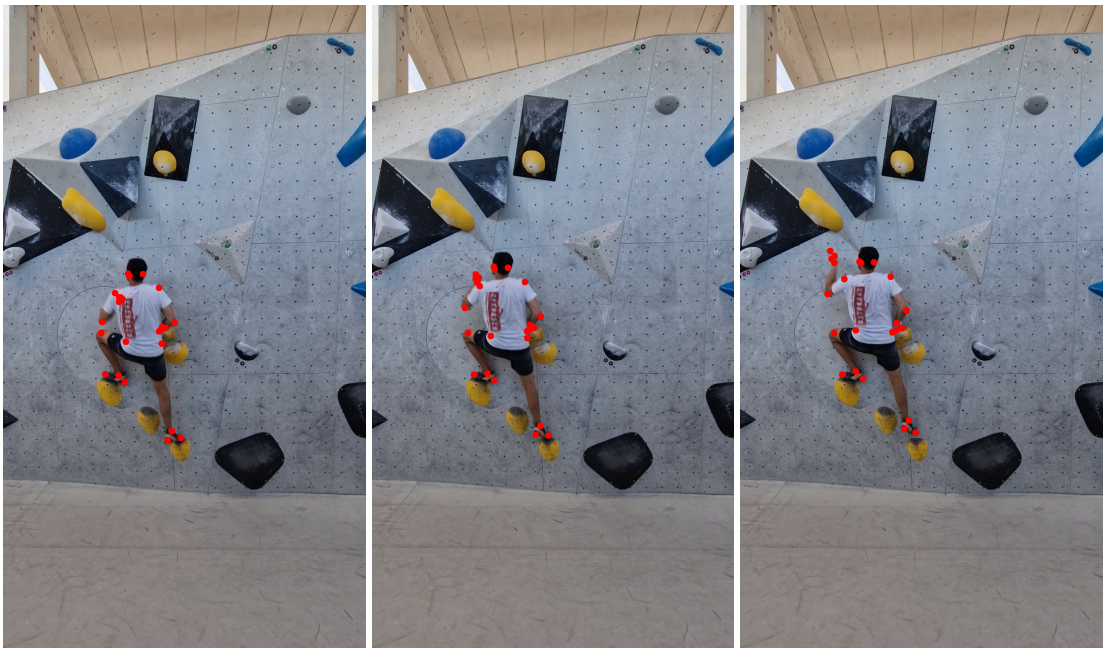
4.1 The Finetuning Dataset

As the aim of our models is to perform well on climbers, we will be using some annotated data of climbers. For this, ClimbAlong has developed a dataset that we will be using. The dataset consists of videos of various climbers on bouldering walls, where each video contains just a single climber. Figure 9 and 10 illustrates two windows of five consecutive frames of a single video from the ClimbAlong dataset. As shown in the figures, the videos in the dataset contains both static movements, where the climber holds a position for a while, as well as quick movements.

(a) Frame 17 (b) Frame 18 (c) Frame 19

(d) Frame 20 (e) Frame 21

Figure 9: Example of five consecutive frames of a video from the ClimbAlong dataset with the corresponding groundtruth keypoints, where the actor holds his position for a while.



(a) Frame 31 (b) Frame 32 (c) Frame 33

(d) Frame 34 (e) Frame 35

Figure 10: Example of five consecutive frames of a video from the ClimbAlong dataset with the corresponding groundtruth keypoints, where the actor performs a quick movement.

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 2: The original 15 action-types in the Penn Action dataset [33].

The dataset consists of 30 fully annotated videos and a total of $10,293$ fully annotated frames, where each annotation consists of 25 keypoints. Table 1 gives an overview of which keypoints are annotated in the dataset. Each videos is filmed in portrait mode with a resolution of 1080×1920 and 30 frames per second. We will throughout this project be referring to this dataset as either the **ClimbAlong dataset** or the **finetuning dataset**.

4.2 The Pretraining Dataset

As we will in section 5 be pretraining our models before specializing our models on bouldering, we will be requiring some pretraining data. For this we use the BRACE dataset [19] and parts of the Penn Action dataset [33]. We chose these datasets, as they are rather similar to the ClimbAlong dataset, as they also mostly consist of unusual movements and only very few usual movements, such as walking. Genrally, the movements of BRACE tend to be quicker than the movements of ClimbAlong, whereas the movements of Penn Action tend to be of a more similar pace. Thus, by incorporating both BRACE and Penn Action, we should capture both the fast and slow movements of the ClimbAlong dataset. The following section introduces these datasets for pretraining.

4.2.1 The BRACE Dataset

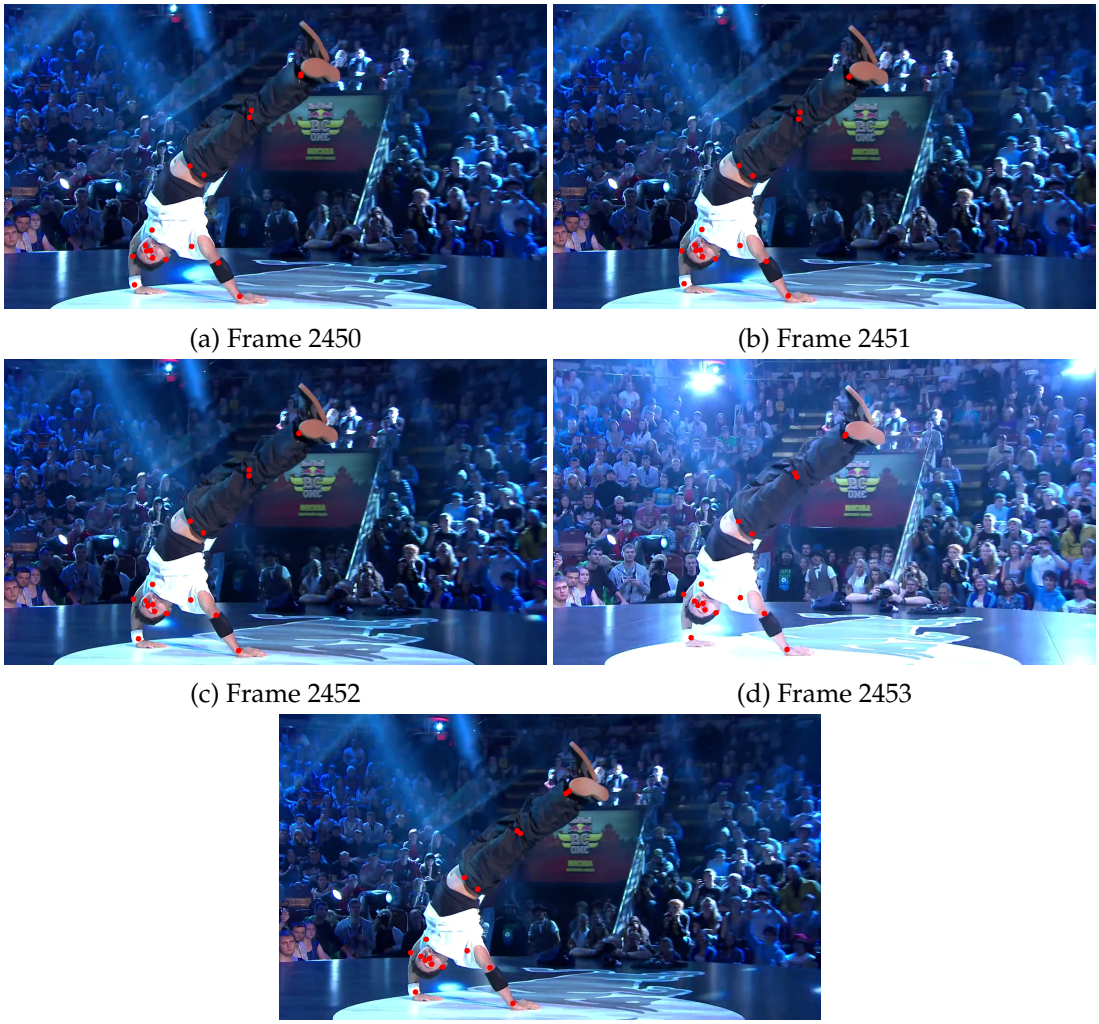
The first pretraining dataset we will be using is the BRACE dataset [19]. This dataset consists of $1,352$ video sequences and a total of $334,538$ frames with keypoints annotations of breakdancers. The frames of the video sequences have a resolution of 1920×1080 [19].

Generally, the movements of the BRACE dataset are quicker than the movements of the ClimbAlong dataset. Similarly, the annotated person of BRACE tend to swap between static and quick movements just like the annotated person of the ClimbAlong dataset. The static poses of the BRACE dataset tend to occur less frequently than the static poses of the the ClimbAlong dataset, as well as the quick movements tend to be quicker than the quick movements of the ClimbAlong dataset. Figure 11 and 12 contains two consecutive sequences, each of five frames, that illustrates these two cases.

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 by using interpolating. Each frame-annotation consists of 17 keypoints, following the COCO-format, as illustrated in Table 1 [19].

4.2.2 The Penn Action Dataset

The second pretraining dataset we will be using is the Penn Action dataset [33]. This dataset consists of $2,326$ video sequences of 15 different action-types. Table 2 lists these 15 action-types [33]. Each frame have a resolution within the size of 640×480 [33].



(a) Frame 2450 (b) Frame 2451

(c) Frame 2452 (d) Frame 2453

(e) Frame 2454

Figure 11: Example of five consecutive frames of a video from the BRACE dataset with the corresponding groundtruth keypoints, where the actor holds his position for a while [19].

(a) Frame 1148 (b) Frame 1149

(c) Frame 1150 (d) Frame 1151

(e) Frame 1152

Figure 12: Example of five consecutive frames of a video from the BRACE dataset with the corresponding groundtruth keypoints, where the actor performs a quick movement [19].

(a) Frame 1148 (b) Frame 1149

(c) Frame 1150 (d) Frame 1151

(e) Frame 1152

Figure 13: Example of five consecutive frames of a video from the Penn Action dataset with the corresponding groundtruth keypoints [33].

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 handling of the non-visible keypoints is very inconsistent, as the position of some of the keypoints have been estimated, whereas for others it have been placed near one of the corners of the frame.

Unlike the BRACE dataset, most of the poses in the Penn Action dataset are not very unusual 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 unusual poses. Thus, we only keep the keypoints that have `baseball_pitch`, `bench_press` or `sit_ups` as their corresponding action-type [33].

In total, we use 307 video sequences from the Penn Action dataset, consisting of a total of $26,036$ frames. Figure 13 illustrates five consecutive frames with its groundtruth annotations for one of these video sequences.

5 Pretraining

As we expect our models to require more data than what is in the ClimbAlong dataset to reach their optimal performace, we have decided to pretrain our models on the BRACE and Penn Action datasets, followed by finetuning them on the ClimbAlong dataset. By doing so we expect our models to yield better results than if we were only using the ClimbAlong dataset, as the pretraining data will be used for adjusting the randomly initialized weights, whereas the finetuning-data will just be used for specializing the model in performing well on the ClimbAlong dataset. The following section describes the pretraining stage, including the data preprocessing, the used configuration details, as well as the obtained results.

In the pretraining stage we will not be using the keypoint detector, but instead only use our temporal-inclusive models by adding noise to the data such that it simulates the output of the keypoint detector on the ClimbAlong dataset. We do this, as the images of pretraining data is very different from the ClimbAlong data, making us believe that the keypoint detector will yield some very inaccurate results, as well as some predictions that will be very different from the predictions of the model on the ClimbAlong dataset.

Throughout section 5.1 we describe the preprocessing of the pretraining dataset. This is then followed by section 5.2, where we describe the used configuration that we use. Then, in section 5.3 we cover the obtain training and validation results, which is followed by section 5.4, where we cover the testing results. Finally, in section 5.5 we cover the technical details of our pretraining.

5.1 Data Preprocessing

As our models take a sequence of estimated poses as input, we will not be using the images of the frames, hence why we discard the images of all frames from BRACE and Penn Action, such that we only keep the annotated poses. We further preprocess these annotations, such that they simluate the output of the keypoint detector as closely as possible.

We start by extracting the bounding-box of the annotated pose in each frame by using the annotated keypoints. As the placement of the invincible keypoints are very inconsistently placed, we have decided to just discard these keypoints, as we expect them to result in sub-optimal results. For the extracted bounding-boxes, we expand each side by 10%, such that no keypoint lies on any of the boundaries of the bounding-box. This is followed by discarding everything outside the bounding-box and rescale the bounding-box to have a sidelength of 56, such that it has the same dimensions as the output of the keypoint detector.

Next, we transform each frame into twenty five heatmaps. This is done by creating twenty five 56×56 zero-matrices for each frame, such that each zero-matrix represents a single keypoint of a single frame. Further, for each keypoint we insert a fixed value $c = 255$ at the position of the keypoint in its corresponding zero-matrix and apply a Gaussian filter with mean $\mu_{out} = 0$ and standard deviation $\sigma_{out} = 1$ to smear out each heatmap. For missing keypoints, we do not place the value c in the corresponding heatmap, making the heatmap consist of only zeros. Further, as Penn Action is the only dataset with the position of the head annotated, as well as the only dataset missing a annotation for the nose, we treat the head-annotation of Penn Action as if it was a nose-annotation, as the position of the two annotations would be very close to each other.

The heatmaps that we produce by following the above description will be used as the groundtruth data. However, as we will be pretraining our models detached from the keypoint detector, we

will also need some data as input. We acquire this data by adding some noise to the data, such that it becomes similar to the output of the keypoint detector, essentially simulating the output of it. The noise is introduced by randomly shifting and smearing out each keypoint of each frame. For the shift-value we sample from a Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = 3k$, where the **shifting-scalar** $k \in \mathbb{R}_{>0}$ is some fixed positive number based on what experiment is run. We clip the position of the shifted keypoints between 0 and 55, such that they cannot be outside of their corresponding heatmap. The smearing is done by using a Gaussian filter with mean $\mu_{in} = 1$ and some standard deviation $\sigma_{in} \in \mathbb{R}_{>0}$

5.2 Training Details

5.2.1 Data Configuration

For the data we use a window-size of $s = 5$ frames, as Artacho *et al.* found this to be the optimal number of frames to use [3], making our dataset consist of $345,120$ windows. Further, we randomly split our dataset into a training, validation and test set, consisting of 60%, 20% and 20% of the data, respectively, without any overlapping or repeating windows among eachother, as this results in minimal evaluation-bias. Lastly, to help the models learn the patterns of the data, we shuffle the three subsets, such that the order of which the data is given to the models does not influence the learning. This should helpt the models learn, as the mini-batches have a very small likelihood of just containing windows from a single video, which would have resulted in a very noisy learning path of the model.

Instead of splitting each video sequence into windows of 5 frames, we could have instead just used the whole video sequence at once. However, we see a couple of problems with this. First off, this would require a lot of memory, as we have to store all of these frames at once. Secondly, for models like 3DConv which has a parameter based on the length of the input sequence, this would be a problem, as the video sequences have different lengths and the only solution would be to either pad the shorter video sequences or trim the longer video sequences, such that all video sequences have the same length.

As the datasets for the pretraining stage are missing some keypoints of the ClimbAlong dataset, we have to cancel out the training of these missing keypoints, as this would otherwise result in the models learning to never predict the presence of these keypoints. There are multiple ways to do this. We handled it during training by setting the groundtruth heatmaps of the missing keypoints equal to the corresponding predicted heatmaps, making the loss of these missing heatmaps be zero and thus the weights of the model of these heatmaps would not be adjusted.

5.2.2 Experiments

For each of the four models we run three different experiments. In experiment 1 we uniformly at random sample the standard deviation used by the Gaussian filter at the input data σ_{in} from the set $\{1, 1.5, 2, 2.5, 3\}$. We do this, as the output heatmaps of the keypoint detector do not use a fixed standard deviation, making the data a better representation of the pose estimator output.

As we find it interesting how big of a difference the randomness experiment 1 makes, we fix this standard deviation in experiment 2, such that we have $\sigma_{in} = 1$, thus, essentially, the models only have to learn to translate the input heatmaps.

Finally, with 30 frames per second and a window-size of $s = 5$, we suspect that the models might be given too little context to actually be able to effectively smooth out the input data.

We could fix this by increasing the window-size, however, we instead chose to make the models work at a lower frame rate, as the increased window-size would also increase the memory usage. Thus, for experiment 3 we still use a window-size of 5, however, with half the frame rate. For this experiment we also sample σ_{in} from the set $\{1, 1.5, 2, 2.5, 3\}$.

As we find it difficult to set the optimal value of the shifting-scalar k, such that the data becomes as close as possible to the finetuning data, we run each experiment twice for each model. In the first run we use a shifting-scalar of $s = 1$, whereas for the second run we increase the value of the shifting-scalar to $s = 2$, making the data a lot more noisy.

5.2.3 Configuration Choices

For optimizing the weights of the models, we use the MSE loss-function, a batch size of 16, and the ADAM optimizer with an initial learning rate of 10^{-3}, and $\rho_1 = 0.9$ and $\rho_2 = 0.999$ as these three values were suggested by [10]. During training we keep track of the lowest reached validation loss of an epoch. If this lowest validation loss has not been beaten for five consecutive epochs, we reduce the learning rate by a factor of 0.1, essentialy speeding up the training. Further, if the lowest validation loss has not been beaten for ten consecutive epochs, we terminate the training of the corresponding model, as it seems to be overfitting. In case this never happens, we terminate the training of the corresponding model once it has been trained for fifty epochs.

To help the models learn, we initialize the weights of all models by sampling from the Glorot uniform distribution, defined by

$$\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right), \tag{28}$$

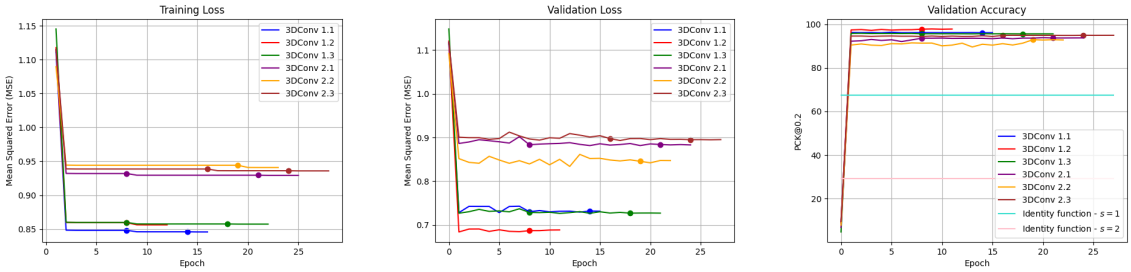
where n_j is the size of the previous layer and n_{j+1} is the size of the layer which is having its weights initialized, as this should decrease the likelihood of **vanishing** or **exploding gradients**, where the steps of gradient are either too short or too long [9].

Further, Zeng *Et al.* [31] have not described in their report their configuration choices for the various parameters of DeciWatch. However, by inspecting the official source code of the paper, we can actually find the configuration of the model, which we will be following [1]. Thus, our implementation of DeciWatch will be using $c = 128$ embedding dimensions, $M = 4$ multi-head Self-Attention blocks, a dropout-rate of 0.1 and five encoder and decoder layers. Zeng *Et al.* [31] actually only sample every 10th frame. However, as the datasets we are using are a lot more fast-paced than the one used by Zeng *Et al.* [31], we have decided to change this sample rate, such that it instead samples every 5th frame, making the model process more frames of the input video.

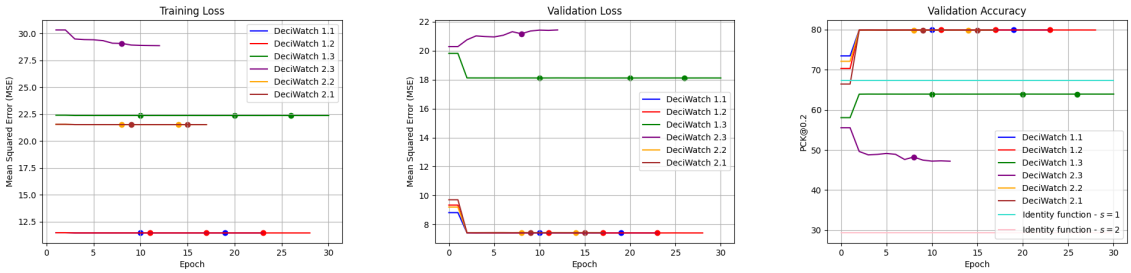
5.3 Training and Validation Results

We have in Figure 14 visualized the evolution of the training loss, validation loss and validation PCK@0.2 accuracy for our 24 models. Each model is delt two numbers seperated by a dot such that it follows a $x.y$-format (for instance, '2.3'), indicating what type of shifing-scalar and experiment it refers to. x indicates what shifting-scalar was used when shifting the input keypoints, and y indicates which of the three experiments from section 5.2 the run belongs to. Each plot has at least one dot, which indicates the reduction of the learning rate due to five

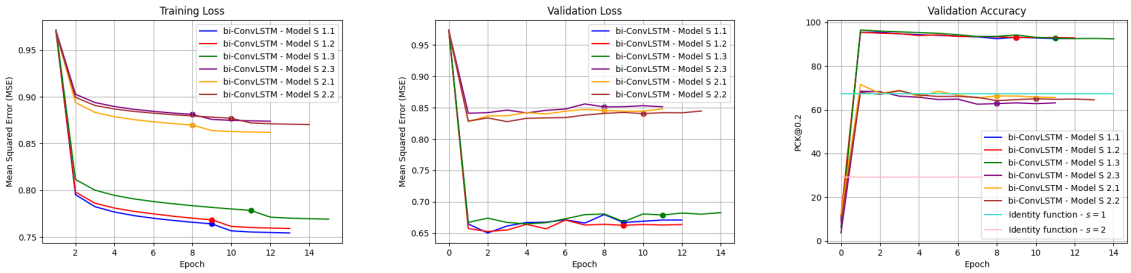
[1] https://github.com/cure-lab/DeciWatch/blob/main/configs/config_jhmdb_simplepose_2D.yaml



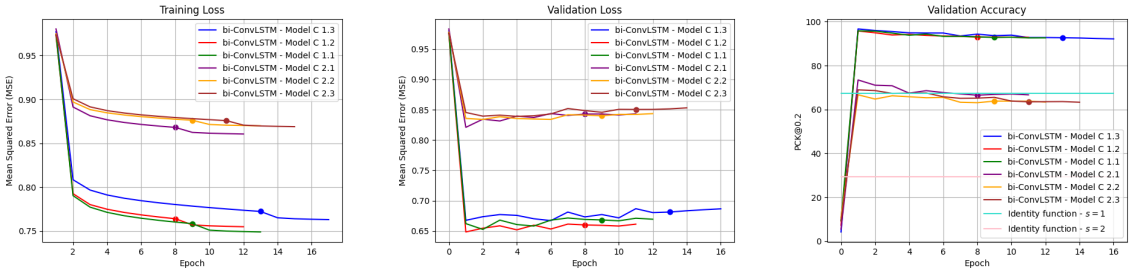
(a) Pretraining results of 3DConv.



(b) Pretraining results of DeciWatch.



(c) Pretraining results of the bi-ConvLSTM Model S.



(d) Pretraining results of the bi-ConvLSTM Model C.

Figure 14: Evolution of the training loss, validation loss and validation PCK@0.2 accuracy of the 24 models during training, as well as the validation PCK@0.2 accuracy of the identity function of the two datasets. The dots indicates a reduction of learning rate. First row: 3DConv. Second row: DeciWatch. Third row: bi-ConvLSTM Model S. Fourth row: bi-ConvLSTM Model C.

consecutive epochs without beating the minimum validation loss.

By comparing the validation accuracies of the various models against the identity function we see, that all of the models, except for DeciWatch 1.3, do learn to somewhat denoise the input data. The simple 3DConv seems to be the architecure that generally delivers the greatest results, whereas DeciWatch is generally the architecure that delivers the worst results. Further, the two architectures that are based on bidirectional convolutional LSTMs have close to no difference in their performance, indicating that our raised concern about one of the architectures not being able to prioritize one LSTM-branch over the other is not as great as we hypothesized it to be.

From the figure we can see, that generally the reduction of the learning rate has an effect on the training loss, as generally the training loss is immediately decreased after a reduction of the learning rate. However, the same cannot be said about the validation loss and accuracy, as these do not immediately decrease after the learning rate has been reduced. Often, the training even terminates before a second reduction of the learning rate takes place, indicating the missing effect of the learning rate reduction on the validation loss.

Further, we can clearly see the effects of the shifting-scalar on the training of the models, as all of the models performs better on the dataset with a low shifting-scalar, than on their corresponding counterpart with a higher shifting-sclar. DeciWatch 1.3 and DeciWatch 2.3 do however seems to be struggling quite a bit, as the validation accuracies of these models are much lower than the other DeciWatch models with the same noise-scalar.

5.4 Test Results

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance (px)*	1.11			2.23			4.46		
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3
Identity function	6.95	6.95	6.95	25.7	25.7	25.7	67.4	67.4	67.4
3DConv	1.84	20.5	0.67	30.6	58.5	23.8	**96.6**	**98.0**	96.3
DeciWatch	**51.4**	**51.4**	**44.5**	64.6	64.6	51.5	80.2	80.2	64.2
bi-ConvLSTM Model S	27.8	31.1	33.8	68.4	71.0	**72.5**	95.7	95.8	96.7
bi-ConvLSTM Model C	29.5	31.7	31.8	**69.5**	**71.3**	71.8	96.1	96.1	**96.9**

Table 3: Testing accuracies of the various developed models for shifting-scalar $s = 1$. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, measured in the heatmap coordinate system.

We have in Table 3 and Table 4 illustrated the results of testing the epoch of each model that yielded the best validation PCK@0.2 accuracy.

For shifting-scalar $s = 1$ we see, that DeciWatch is the architecure that yields the best results for PCK@0.05, however, for PCK@0.2 DeciWatch actually tends to be the worst performing model. Further, as stated in section 5.3 DeciWatch 1.3 does perform worse than the identity function when considering PCK@0.2, however, for both PCK@0.05 and PCK@0.1 DeciWatch 1.3 completely outperforms the identity function, hence why it actually does learn to denoise the data, unlike what we stated in section 5.3 where we just looked at the validation PCK@0.2 accuracy. Generally, most of the predicitons of DeciWatch that are considered correct, are actually very close to the groundtruth, however the predictions that are incorrect are pretty far

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance (px)*	1.11			2.23			4.46		
Experiment	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3
Identity function	1.84	1.84	1.84	7.75	7.75	7.75	29.3	29.3	29.3
3DConv	0.80	2.40	0.11	21.6	27.6	16.8	**94.2**	**91.8**	**95.5**
DeciWatch	**51.2**	**51.2**	**10.3**	**64.4**	**64.4**	24.4	80.2	80.2	50.3
bi-ConvLSTM Model S	10.5	11.6	10.1	31.1	33.4	29.5	68.5	67.8	69.6
bi-ConvLSTM Model C	12.5	12.0	9.63	36.4	32.5	**29.8**	74.6	65.5	70.0

Table 4: Testing accuracies of the various developed models for shifting-scalar $s = 2$. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, measured in the heatmap coordinate system.

away from their corresponding groundtruth location.

On the other hand, the simple 3DConv seems to be the architecture that generally performs the worst for PCK@0.05 and PCK@0.1, and actually the architecture that performs the best for PCK@0.2, except for run 1.3, where the two bidirectional convolutional LSTM based models actually outperforms it. Thus, 3DConv is very good at inferencing a rough estimate of the position of the keypoints, however, the exact position of these predictions are generally incorrect.

Generally, the bidirectional convolutional LSTMs yield some decent results, as the predicitons of these models are always pretty far away from the worst-performing model for any of the three used accuracy metrics, and for PCK@0.1 they even seem to yield the best results of the four architectures. Similarly to the validation results, the test results do only indicate a minor performance differences between the two different types of bidirectional convolutional LSTMs in favor of the bi-ConvLSTM Model C. Likewise, similarly to the validation results, the test results indicate a major effect of a low frame rate for DeciWatch.

If we compare the performances of the models on experiment 2 against their performances on experiment 1 we see, that all arhcitectures, except for DeciWatch, performs better on experiment 2 than they do on experiment 1. This is especially true for 3DConv, whose run on experiment 2 completely outperforms its run on experiment 1. On the other hand, if we compare the results of experiment 3 against the results of experiment 1 we see, that both 3DConv and DeciWatch perform worse on experiment 3 than they do on experiment 1. However, the two archtectures that are based on a bidirectional convolution LSTM actually continuously performs better in experiment 3 than they do in experiment 1, making us believe that the added context when decreasing the frame rate, actually help these models.

Generally, the testing results of shifting-scalar $s = 2$ follow the same pattern with some minor differences. If we consider the results of PCK@0.2, 3DConv is now the architecture that always yields the best results - and by a large margin, but it is on the other hand completely outperformed by the other models when considering PCK@0.05. This indicates, that the architecture can still yield decent rough estimations of the keypoints when a lot of noise is added, however, it does also indicate, that the exact position of these keypoints are very much effected by the level of noise. Further, its results in experiment 2 do not any longer outperform its results in experiment 1 when considering PCK@0.2, as it was the case for shifting-scalar $s = 1$.
We have further in Table 5 and Table 6 illustrated the keypoint-specific testing PCK@0.2 of these models.

	3DConv			DeciWatch			bi-ConvLSTM sum.			bi-ConvLSTM concat.			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	96.2	97.3	96.3	83.0	83.0	68.0	97.3	94.2	97.8	96.2	96.4	95.9	91.8
Ear	96.3	97.0	96.3	85.1	85.1	70.0	96.8	96.3	96.9	96.3	96.4	96.1	92.4
Shoulder	96.7	98.6	96.7	83.8	83.8	68.6	97.0	96.9	96.0	96.5	97.0	97.2	92.4
Elbow	96.7	98.7	96.4	78.3	78.3	61.8	96.1	97.6	97.6	96.1	96.5	98.5	91.1
Wrist	96.5	97.8	96.2	74.0	74.0	57.7	95.8	95.0	96.2	95.9	95.5	97.0	89.3
Hip	96.8	99.1	96.3	80.3	80.3	64.0	93.5	95.7	96.6	96.8	95.3	96.3	90.1
Knee	96.8	98.5	96.4	75.9	75.9	60.0	95.2	96.0	96.7	95.4	96.4	97.6	90.1
Ankle	96.2	96.9	95.8	72.8	72.8	57.9	95.1	94.1	96.7	95.7	95.6	96.4	88.8
Total	96.6	98.0	96.3	80.2	80.2	64.2	95.7	95.8	96.7	96.1	96.1	96.9	

Table 5: Keypoint-specific testing PCK@0.2-accuracies of the various models for shiting-scalar $s = 1$. All the accuracies are in percentage.

	3DConv			DeciWatch			bi-ConvLSTM sum.			bi-ConvLSTM concat.			Total
Experiment	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3	
Nose	92.5	84.5	94.4	82.9	82.9	53.3	75.8	70.6	73.5	73.9	72.2	73.7	77.5
Ear	92.0	91.1	94.4	85.0	85.0	51.0	80.3	77.7	77.7	80.8	79.2	79.9	81.2
Shoulder	93.7	91.9	95.6	83.8	83.8	40.8	74.7	68.2	68.9	70.4	68.1	68.3	75.7
Elbow	94.8	92.3	95.8	78.3	78.3	39.9	58.2	66.7	64.7	67.0	61.6	63.3	71.7
Wrist	94.8	92.6	96.0	73.9	74.0	56.0	69.4	66.3	69.4	75.6	65.4	69.4	75.2
Hip	94.5	91.6	95.7	80.3	80.3	46.0	64.7	61.9	67.5	80.7	57.3	74.8	74.6
Knee	95.0	92.7	95.7	75.9	75.9	53.3	60.1	65.9	64.7	68.6	57.0	54.5	71.6
Ankle	95.2	94.1	95.6	72.7	72.7	57.7	69.4	67.3	72.5	78.1	67.5	77.5	76.7
Total	94.2	91.8	95.5	80.2	80.2	50.3	68.5	67.8	69.6	74.6	65.5	70.0	

Table 6: Keypoint-specific testing PCK@0.2-accuracies of the various models for shiting-scalar $s = 2$. All the accuracies are in percentage.

Looking at Table 5 we see, that the models with shifting-scalar $s = 1$ generally perform the best on the ears and shoulders, whereas they perform the worst the wrists and ankles. The models in Table 6 tells another story, as instead of performing the worst on the ankles and wrists, the elbows and knees are now the most difficult keypoints, and the ears, nose and ankles are now the least difficult keypoints.

5.5 Technical Details

All models were trained and evaluated on a shared GPU cluster using a 24GB NVIDIA Titan RTX and an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz. All models were implemented in Python version 3.9.9 using PyTorch 2.0.0. 3DConv took about 60 minutes per epoch, DeciWatch about 86 minutes per epoch, the bi-ConvLSTM Model S about 75 minutes per epoch, and the bi-ConvLSTM Model C about 88 minutes per epoch.

6 Finetuning

As we now have finetuned our models, we need to finetune the models, such that they are specialized to yield optimial results on the ClimbAlong dataset. The following section describes the finetuning of these models. This includes the preprocessing of the data, the configuration details we use, as well as the obtained results.

In the finetuning stage we will be using the keypoint detector to train our temporal-inclusive models. However, we will be freezing the pose estimator, such that the weights of the model will not change during the training and we will thus only train our temporal-inclusive models. We do this as (1) the training of the models will be quicker, as we just need to train the temporal-inclusive models and not the keypoint detector, and (2) we get an greater understanding of the effects of our models when combined with the pose estimator, as we can clearly see how big of a difference it makes by adding our temporal-inclusive models.

In section 6.1 we cover the preprocessing of the finetune dataset. Then, in section 6.2 we cover the used configuration for the finetuning. Then, in section 6.3 and section 6.4 we cover the training and validation results, as well as the testing results, respectively. Lastly, in section 6.5 we cover the technical details of the finetuning.

6.1 Data Preprocessing

For the ClimbAlong dataset we perform only minor preprocessing. First, the preprocessing of each video is done by having the keypoint detector process the video, such that we have the output heatmaps of the pose estimator, containing all of the pose-estimations of each video. Next, we preprocess the heatmaps by setting all negative values to 0 and normalizing each heatmap, such that each heatmap sums up to the fixed value $c = 255$ that we used when preprocessing the BRACE and Penn Action datasets, essentially making the heatmaps more similar to the preprocessed heatmaps of BRACE and Penn Action. These heatmaps will then be used as the input for our models.

For the groundtruth heatmaps we create twenty five heatmaps of each frame, similarly to how we did it for the BRACE and Penn Action datasets, however, in this case we use the predicted bounding-box of the pose estimator as our bounding-box. In cases where the groundtruth keypoint is placed outside of the bounding-box, we place the groundtruth keypoint at the closest border of the bounding-box.

6.2 Training Details

Data Configuration Generally, for the data configuration we follow a similar approach to how we did in the pretraining stage. We again use a window-size of $k = 5$ frames, resulting in a total of $10,173$ windows. Also here are we using $c = 255$ as a representation of the ground truth placement of each keypoint.

As this dataset is much smaller the the dataset used during the pretraining-stage, we can much more easily introduce some evaluation-bias, hence why we also take much more careful steps. Thus, the splitting of the dataset is different than how we performed it in the pretraining-stage. First, we extract the longest video, such that none of its windows are on the dataset, which we will be used to unsure an unbias evaluation of our models. Next, we use the first 60% of the frame-windows and use those as the training dataset. For the remaining 40% of the frame-windows, we make sure that they have no overlapping frames with training dataset - if they do, the whole window is moved into the training dataset. As we still have to split the

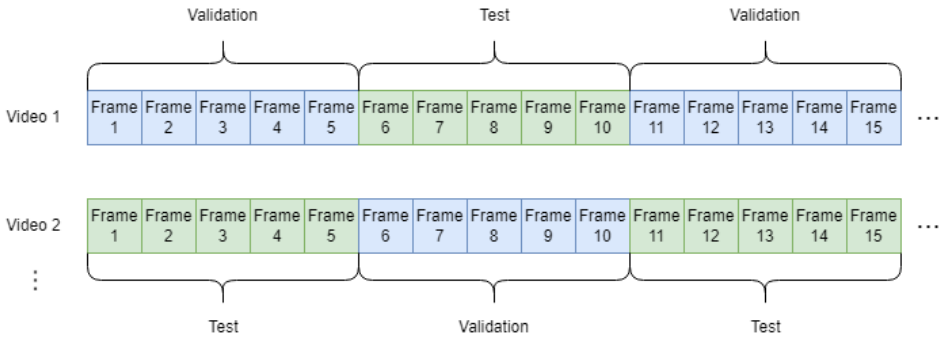


Figure 15: Illustration of how the ClimbAlong data has been split into a validation dataset and a test dataset.

remaining 40% of the frame-windows into a validation dataset and a test dataset, we have to make sure, that (1) both datasets use as many video sequences as possible, which is especially important in this case due to the small dataset size, (2) that none of the windows are overlapping among the two datasets, and (3) both datasets sample the windows uniformly distributed throughout the video sequences, as there can be some sepcial movements in certain parts of the video sequences. All of these constraints will make sure that we minimize the evaluation-bias of our models. The splitting of the remaining 40% of the window into two datasets is then done by sorting the non-overlapping windows by their video and by their time of occurence in their corresponding video. Then, if the ith window-frame is divisible by two, it is moved into the validation dataset, otherwise it is moved into the test dataset. By doing so we meet all three of our conditions. The overall idea has been illustrated in Figure 15.

Experiments As the finetuning dataset is so small, the fitting of the models are very quick, making us fit all of the 26 developed models from the pretraining stage, instead of us picking which models we should finetune. For each model we pick the epoch from the pretraining stage, that yielded the highest validation accuracy and use that for finetuning.

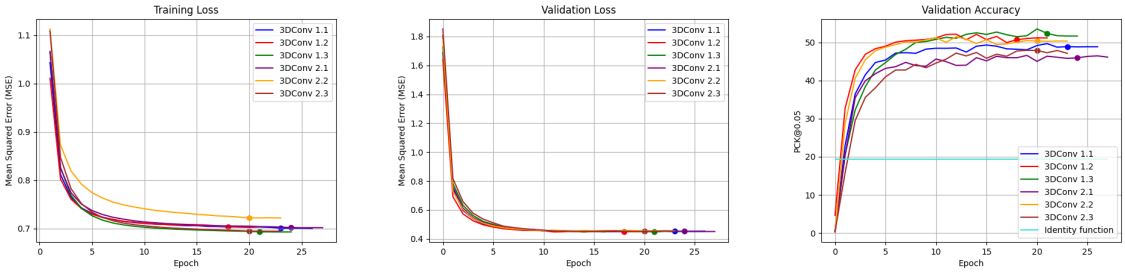
Training Configuration The optimization parameters are very similar to the ones from the pretraining stage. We again use the MSE loss-function, a batch size of 16, and the ADAM optimizer with an initial learning rate of 10^{-3}, and $\rho_1 = 0.9$ and $\rho_2 = 0.999$ as these three values were suggested by [10]. During training, we again keep track of the lowest reached validation loss of an epoch and use learning rate reduction and early-stopping in a similar manner to how we did in the pretraining stage. However, unlike the pretraining stage, we here use a smaller iniitial learning rate of 10^{-4}, as the weights only need to be fineadjusted, making us believe that a greater learning rate would skew the weights too much.

6.3 Training and Validation Results

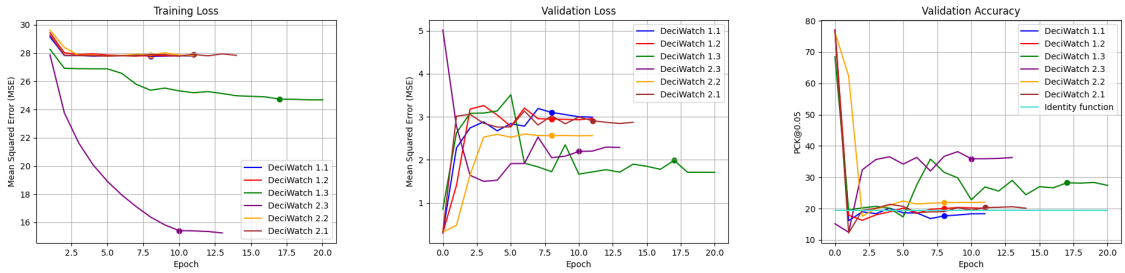
We have in Figure 16 illustrated the evaluation of the training loss, validation loss, and validation PCK@0.05 accuracy of the various models during the finetuning.

If we compare the models against the identity function we clearly see, how all models at some point beats the identity function, indicating the positive effects of incorporating temporal information into pose estimation.

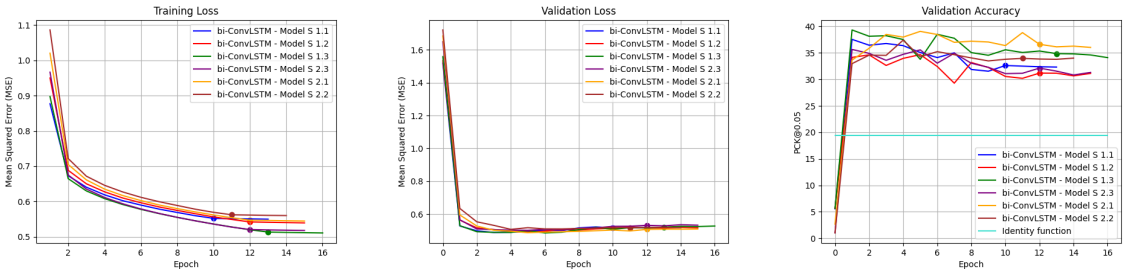
Generally, 3DConv seems to be converging towards the greatest results, as these models tend to converge towards the highest validation PCK@0.05 accuracy. However, some DeciWatch



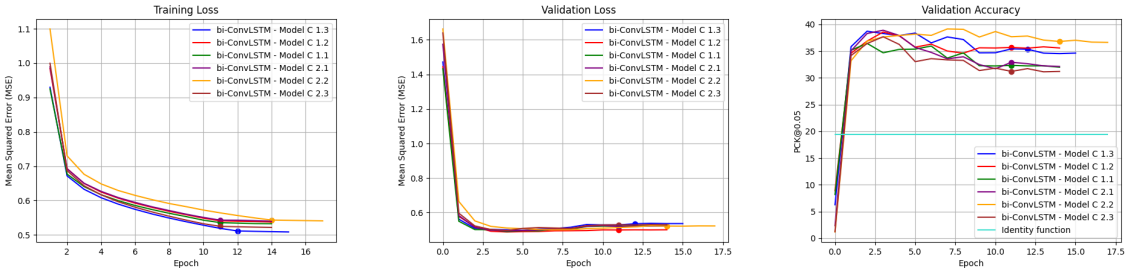
(a) Finetuning results of 3DConv.



(b) Finetuning results of DeciWatch.



(c) Finetuning results of the bidirectional convolutional LSTM with summing.



(d) Finetuning results of the bidirectional convolutional LSTM with concatenation.

Figure 16: Evolution of the training loss, validation loss and validation PCK@0.05 accuracy of the 24 models during training, as well as the validation PCK@0.05 accuracy of the identity function of the two datasets. The dots indicates a reduction of learning rate. First row: 3DConv. Second row: DeciWatch. Third row: Bidirectional Convolutional LSTM with summing. Fourth row: Bidirectional Convolutional LSTM with concatenation.

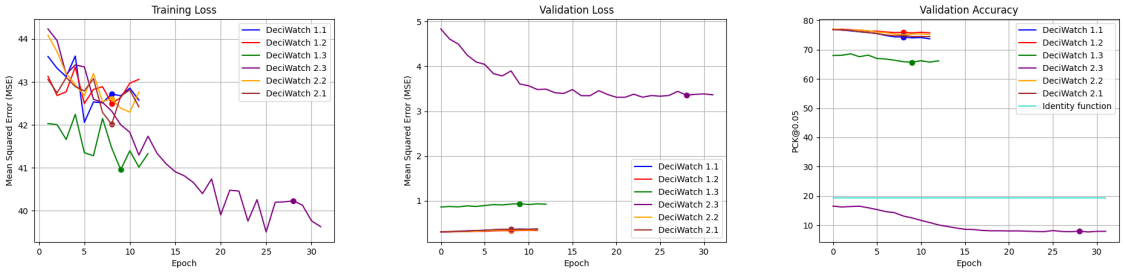


Figure 17: Finetuning results of DeciWatch with regularization techniques.

runs actually starts off with an even higher validation PCK@0.05 accuracy, which then decreases to a much lower value quickly. The two architectures that are based on a bidirectional convolutional LSTM tend to yield some decent results, however, their training tend to plateau rather early, hence why they also terminate very quickly. Generally however, both DeciWatch and the two architectures based on a bidirectional convolutional LSTM do seem to overfit. For DeciWatch this is mostly immediately, whereas for the bidirectional convolutional LSTMs this tend to happen a few epochs later.

Further, the shifting-scalar seems to only have a minor effect on the models during the finetuning, as all six runs of each model tend converge towards the same result.

6.3.1 Additional Experiment Results

As seen in Figure 16, most of the DeciWatch methods would immediately overfit, as the validation loss increased while the training loss decreased, leading to the validation accuracy decreasing overtime. We performed further experiments, where we implemented various regularization techniques in hopes of this would lower the likelihood of the models overfitting. This was done by (1) freezing the whole network except for the very last fully-connected layer, which we reinitialized randomly, (2) added variance to the data by rotating each window throughout the training with a random degree in the range $[-45, 45]$, and (3) by applying weight decay with $\lambda = 0.001$.

By incorporating these regualrization techniques we obtain the results visualized in Figure 17. While the training loss doss generally decrease much slower than previously, the validation accuracy is still decreasing, making the best setting of the models without these regualrization methods perform either at the same level or an even better level than the models with regualrization.

6.4 Test Results

We have in Table 7 and Table 8 illustrated the testing results of the epoch of each model that yielded the highest validation PCK@0.05 accuracy.

by comparing the two tables against each other we see, that the shifting-scalar only have a minor effect on the results of the models. Generally however, the models do perform best with the noise-scalar $s = 1$.

Similarly to the pretraining stage, we also see here how 3DConv from experiment 2 performs better than 3DConv from experiment 1, however, this performance difference is now smaller than it was in the pretraining stage.

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance (px)*	0.80			1.60			3.21		
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3
Identity function	19.4	19.4	19.4	66.1	66.1	66.1	85.2	85.2	85.2
3DConv	49.7	52.3	53.1	**95.7**	**95.7**	**95.8**	99.2	99.3	99.3
DeciWatch	**76.6**	**76.7**	**68.1**	94.4	94.3	87.3	99.2	99.2	96.1
bi-ConvLSTM - Model S	37.8	34.9	39.0	91.8	92.1	92.2	99.4	**99.7**	99.2
bi-ConvLSTM - Model C	35.9	39.0	38.5	93.1	93.6	92.6	**99.8**	**99.7**	**99.7**

Table 7: Testing accuracies of the various developed models for shifting-scalar $s = 1$. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, measured in the heatmap coordinate system.

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance (px)*	0.80			1.60			3.21		
Experiment	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3
Identity function	19.4	19.4	19.4	66.1	66.1	66.1	85.2	85.2	85.2
3DConv	46.5	51.6	**47.3**	**95.5**	**95.5**	**95.8**	99.2	99.3	99.2
DeciWatch	**76.0**	**75.9**	36.8	94.2	94.2	74.9	99.2	99.2	92.8
bi-ConvLSTM - Model S	38.8	37.4	35.9	92.7	92.1	91.2	99.4	**99.5**	99.3
bi-ConvLSTM - Model C	39.2	39.5	37.1	92.5	92.9	92.6	**99.6**	99.3	**99.6**

Table 8: Testing accuracies of the various developed models for shifting-scalar $s = 2$. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, measured in the heatmap coordinate system.

Further, for experiment 3 DeciWatch is not delivering great results, similarly to its results in the pretraining stage. For the other architectures, there do not seem to be any consistent pattern, as some of them experiment 3 yields the best results, whereas it for others yield the worst results.

Like in the pretraining stage, there does not seem to be any major performance differences between the two architectures that are based on the bidirectional convolutional LSTM, making us further believe that our concern about the missing opportunity of biConv-LSTM Model S to prioritize a processing direction is not that important.

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance*	0.87			1.77			3.55		
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3
Identity function	21.2	21.2	21.2	65.5	65.5	65.5	84.7	84.7	84.7
3DConv	58.4	61.4	61.7	**98.7**	**98.9**	**99.0**	**99.6**	**99.8**	**99.7**
DeciWatch	**82.6**	**82.4**	**74.6**	96.2	96.1	92.3	99.1	99.1	97.4
bi-ConvLSTM - Model S	45.7	45.0	47.6	97.3	96.9	97.0	**99.6**	**99.6**	99.1
bi-ConvLSTM - Model C	44.5	46.1	48.5	97.4	97.9	97.9	99.6	99.5	99.6

Table 9: Testing accuracies of the various developed models for shifting-scalar $s = 1$ on the additional test video. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, using the units of the heatmap coordinates.

Accuracy metric	PCK@0.05			PCK@0.1			PCK@0.2		
Mean threshold distance*	0.87			1.77			3.55		
Experiment	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3
Identity function	21.2	21.2	21.2	65.5	65.5	65.5	84.7	84.7	84.7
3DConv	56.2	60.0	**56.6**	**98.9**	**98.8**	**98.8**	**99.7**	**99.7**	**99.7**
DeciWatch	**81.6**	**81.8**	37.5	96.0	96.0	73.3	99.1	99.1	90.7
bi-ConvLSTM - Model S	44.8	46.2	45.0	96.9	95.9	97.1	99.5	99.6	99.5
bi-ConvLSTM - Model C	45.9	47.9	46.7	96.7	97.1	98.1	99.6	99.4	99.6

Table 10: Testing accuracies of the various developed models for shifting-scalar $s = 2$ on the additional test video. All the accuracies are in percentage. *: The mean maximum distance between the predicted keypoint and corresponding groundtruth keypoint for the prediction to count as being correct, using the units of the heatmap coordinates.

One might argue, that our way of splitting the finetuning dataset into a validation and test dataset might introduce some bias, as the two datasets have overlapping videos and thus videos of the same person that might have some unique movement. To further minimize the likelihood of a biased evaluation of the models, we have in Table 9 and Table 10 tested the models on the longest video of the dataset, which was not used at all in any of the three data-subsets and consists of 754 overlapping windows. By comparing these tables against Table 7 and 8 we can clearly see, that the models are performing similarly, if not actually better, than how they did previously, hence why we would argue, that there are no indication of the first test evaluation being biased. However, we have to note, that the evaluation in Table 9 and Table 10 do contain some bias, as the evaluation is of a single person who might have some easily predictable movements.
We have in Table 11 and Table 12 illustrated the keypoint specific PCK0.1 testing accuracies of the models. For similar tables for PCK@0.05 and PCK@0.2 see Table 15-18 in the appendix.

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	95.5	97.6	95.7	94.9	94.9	88.5	93.5	89.7	95.4	93.4	88.8	90.9	93.2
Ear	96.5	96.7	96.8	95.5	95.5	88.6	94.3	91.9	94.2	96.3	95.6	96.1	94.8
Shoulder	95.6	97.4	97.1	95.4	95.4	88.4	91.2	90.6	87.2	91.9	95.1	92.9	93.2
Elbow	97.6	97.2	97.5	94.5	94.6	86.5	93.2	93.7	94.9	95.1	96.7	95.2	94.7
Wrist	96.4	96.4	96.3	93.9	93.9	85.8	94.2	92.7	94.1	94.5	93.2	94.3	93.8
Pinky	86.6	85.4	85.4	92.5	92.2	84.0	84.5	86.5	86.6	83.4	88.4	87.0	86.9
Index finger	92.7	92.5	93.1	92.2	92.1	84.3	90.3	89.1	88.2	89.9	92.3	92.1	90.7
Thumb	91.6	91.0	91.4	92.8	92.7	84.5	81.0	89.3	84.9	89.6	85.5	86.9	88.4
Hip	99.4	99.0	99.8	96.4	96.4	89.9	92.5	96.5	98.0	94.6	97.7	96.1	96.4
Knee	97.7	98.1	98.4	95.0	95.0	88.4	96.1	92.5	96.5	96.0	93.9	93.6	95.1
Ankle	98.6	98.4	99.1	95.0	95.0	87.6	94.3	93.6	96.0	97.2	98.1	94.3	95.6
Heel	98.6	98.1	98.5	95.2	95.1	86.2	94.2	93.4	94.9	92.5	92.6	93.0	94.4
Toes	97.2	97.5	98.0	94.2	94.0	86.8	94.9	95.3	95.5	93.8	95.7	94.4	94.8
Total	95.7	95.7	95.8	94.4	94.3	87.3	91.8	92.1	92.2	93.1	93.6	92.6	

Table 11: Keypoint-specific testing PCK@0.1-accuracies of the various models for shiting-scalar $s = 1$. All the accuracies are in percentage.

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	95.6	97.0	95.6	94.1	94.3	80.8	90.9	89.6	93.2	88.1	91.1	90.8	91.76
Ear	96.8	95.7	96.9	93.3	93.4	80.8	94.2	92.7	92.6	91.9	93.2	94.9	93.0
Shoulder	95.1	97.1	95.7	95.3	95.4	77.5	93.5	93.4	90.1	90.0	97.7	95.6	93.0
Elbow	97.2	97.1	97.5	94.4	94.5	72.0	96.0	96.7	90.9	97.1	97.0	94.7	93.8
Wrist	96.4	96.5	96.6	93.8	93.8	77.4	89.2	93.8	91.9	93.2	94.3	9.33	85.5
Pinky	85.0	85.3	87.0	92.3	92.4	66.1	84.5	82.6	84.0	87.4	83.6	89.7	85.0
Index finger	93.1	92.8	93.0	92.5	92.5	65.3	92.7	89.9	92.4	92.1	88.0	91.4	89.6
Thumb	91.1	90.8	91.6	93.0	92.9	71.4	85.3	83.5	81.1	82.6	84.6	88.9	86.4
Hip	99.3	98.8	99.6	96.3	96.4	78.6	97.1	96.2	96.8	97.8	97.4	84.5	94.9
Knee	97.6	97.8	97.9	95.1	95.0	77.9	94.9	93.3	91.3	95.9	98.4	97.5	94.4
Ankle	98.6	98.5	99.3	94.5	94.6	79.3	96.2	96.0	93.8	98.5	96.8	96.2	95.2
Heel	98.2	97.8	98.8	95.1	95.0	78.5	93.7	93.0	93.2	93.0	91.3	92.8	93.4
Toes	97.4	97.4	97.8	94.0	94.0	73.1	95.1	96.8	93.5	92.7	93.7	93.7	93.3
Total	95.5	95.5	95.8	94.2	94.2	74.9	92.7	92.1	91.2	92.5	92.9	92.6	

Table 12: Keypoint-specific testing PCK@0.1-accuracies of the various models for shiting-scalar $s = 2$. All the accuracies are in percentage.

By comparing these two tables to the equivalent tables from section 5.3 we see, that the most difficult keypoints are no longer the wrists and ankles, but generally are instead the pinkies, the index fingers and the thumbs, on which the models generally performs much worse than on the remaining keypoints.

For some illustrations of the predictions of the best setup of each model based on Table **??** and Table **??**, see Figure **??** in the appendix.

6.5 Technical Details

All models were trained and evaluated using a 8GB NVIDIA GTX 1070 and an Intel Core i7-4790K @ 4.00GHz. All models were implemented in Python version 3.9.9 using PyTorch 2.0.0. 3DConv took about 1.5 minutes per epoch, DeciWatch about 2 minutes per epoch, and the two bidirectional convolutional LSTMs took about 2.5 minutes per epoch each.

7 Discussion

The following section discusses our obtained results from the pretraining and finetuning stages, some concerns regarding possible suboptimal choices we have made, as well as suggests some possible future work. The section begins in section 7.1, where we discuss the obtain results. Then, in section 7.2, where we discuss why the models performed much better during the finetuning stage, than during the pretraining stage. Section 7.3 then discusses which model is the optimal choice. In section 7.4 we then discuss some pitfalls of our approach and how we could have made some better choices. Lastly, in section 7.5 we present some possible future work possibilities.

7.1 Results

In section 5 and section 6 we successfully implemented and tested the four architectures in various experiments. We will in the following be analysing and discussing the potential reasons for these obtained results.

7.1.1 Pretraining

We saw in section 5.4 how 3DConv tend to perform much better in experiment 2, than it does in experiment 1 (with the exception of run 2.1 and 2.2). As the goal of experiment 1 is for the models to both learn translation and scaling, whereas the goal of experiment 2 is just for the models to learn translation, we can clearly see here how big of a difference the task of learning to also scale the data has on the results for 3DConv. Of course, a tiny bit of randomness does also play in here, as the models for the two experiments were initialized with different weights. However, as the weights were sampled from the same distribution, as well as 3DConv having very few weights, we find it hard to believe that this randomness has such a big impact on the results. Thus we claim that the performance differences between the two experiments is due to the task of scaling the data. For the two architectures that are based on a bidirectional convolutional LSTM, a similar pattern is observed for the shifting scalar, however, this performance difference is much smaller than it was in the case for 3DConv. We simply believe, that this improved performance is due to the bidirectional convolutional LSTMs predicting the position of the keypoints in a more sophisticated manner. For DeciWatch there are no performance differences between experiment 1 and experiment 2 when considering shifting-scalar. However, this is expected, as unlike the other architectures, DeciWatch works directly on keypoint-coordinates instead of the heatmaps, making it scaling invariant and thus making it the two experiments equivalent to each other.

We further saw in section 5.4 the effects on the models of halfing the frame rate. For both the 3-dimensional convolutional LSTM and DeciWatch this had a negative effect, whereas it actually had a positive effect for the two models that are based on a bidirectional LSTM. For DeciWatch we have to keep in mind, that it is already only considering every fifth frame, so by halfing the frame rate it is actually only considering every tenth frame, removing a lot of the context. For 3DConv and the two models based on a bidirectional convolutional LSTM, we again suggest that the performance difference is due to the later models yielding their predictions in a more sophisticated manner than the simple 3DConv. In the case of when the shifting-scalar is $s = 2$ all models generally perform worse in experiment 3 than they do in experiment 1, which we simply believe is due to the data being so noisy, that the models need the finer details to infer their predictions. As the distance-threshold for PCK is increased however, 3DConv actually perform better in experiment 3 than it does in experiment 1, contradicting this belief. However, we have to note here, that the performance difference is rather small and

that it is in the case where the the distance-threshold is at its greatest. Thus, we believe the performance difference is due to the random initialization of the models or the models learning a rough estimate of the keypoints over learning a very fine estimation as this is very difficult when the data is very noisy. We further strengthen our last believe when considering, that the model is the lowest performing model when considering PCK@0.05 but the highest performing model when considering PCK@0.2 and thus generally learns to yield a rough estiamte of the position of the keypoints.

We finally saw in 5.4 how the models for shifting-scalar $s = 1$ had the most difficulty with the wrists and ankles, whereas they performed the best on the ears and shoulders. However, we find this very expected, as the wrists and ankles tend to be the joints that have the highest amount of movements, especially in sports which our pretraining dataset is centered around, and thus are much more difficult to denoise than more static joints such as the ears and shoulders. On the other hand, in the case of using the shifting-scalar $s = 2$ we saw, that the models struggled the most with the elbows and knees, and performed the best on the ears, nose and the ankles. At first glance, one would find this rather unexpecting as the wrists and ankles are no longer the most difficult keypoints. However, we suggest that this could be caused by these keypoints being too close to the heatmap borders, such that when they are shifted a lot towards any of these borders, they will end up at the border, as they cannot exceed the border. Thus, the wrists and ankles are actually not shifted as far as the other keypoints, such as the knees and elbows which would then be the most difficult keypoints, as these keypoints are both shifted a lot and the corresponding joints contain a decent amount of movement.

7.1.2 Finetuning

In section 6.3 we saw how 3DConv generally converged towards the highest validation accuracy, whereas the DeciWatch-based models and the models based on a bidirectional convolutional LSTM tended to overfit and thus yielding worse validation results as the training continues.The ClimbAlong-dataset is very small and these models are much bigger than 3DConv (in terms of tuneable parameters), which means that they have a greater likelihood of "remembering" the ClimbAlong-dataset instead of learning its patterns and thus overfitting. We do however find it odd, that the DeciWatch models from experiment 3 are not overfitting. We are not too sure why these two models are not overfitting.

In section 6.4 we saw how the models from the experiments with shifting-scalar $s = 1$ tend to yield better results than the models trained on data with shifting-scalar $s = 2$. We further saw, how the models from experiment 2 generally performs better than the models from experiment 1. Generally however, these performance differences are very minor, which makes sense as the finetuning-data is the same for the two groups. We do note the minor performance differences however, which probably means, that (1) the noise in the finetuning data is more similar to the data with shifting-scalar $s = 1$ than the data with shifting-scalar $s = 2$, and (2) the standard deviation of the peaks of the finetuning data is not changing as much as they do in our experiment 2.

One could look at Figure 16 and claim that the pretraining-stage has had no effect, as the validation accuracy starts off very low. However, we disagree for two reasons. First off, we just noted that the choice of shifting-scalar from the pretraining-stage had an effect on the final results, so the pretraining-stage must have had an effect. Secondly, most of the models actually reach a very high validation accuracy after just a couple of epochs, which we believe is due to the pretraining. One could argue, that a similar pattern was observed during the pretraining-stage and is thus just a general pattern. However, we have to remember, that the

pretraining-dataset is much bigger than the finetuning-dataset and thus just after one epoch in the pretraining-stage, the models have already been trained on a lot of samples, which explains this major performance-jump during the pretraining-stage.

Lastly, one saw in section 6.4 how the models tend to perform the worst on keypoints related to the thumbs, pinkies and index fingers. We see two reasons for this observation. First off, these keypoints were not included in the pretraining dataset, meaning the models have learned to infer the position of these keypoints purely from the finetuning dataset. We believe this reason to only have a minor effect, as if we look at the performance of the models on the keypoints related to the heels and the toes, which were also not included in the pretraining dataset, we see, that the models generally performs only a tiny bit worse on these keypoints, that on the remaining keypoints. The second reason is, that the finger keypoints just have a lot of movement, as the related joints are the joints that are mostly used for bouldering and ara thus the joints that moves the most. And as previously stated, keypoints that move a lot are more difficult to predict the position of.

7.2 Why did the Models Perform Better during Finetuning than during Pretraining?

All of the finetuned models outperforms their pretrained counterpart. We see a couple of reasons for this.

First off, the models have been trained om more data, as they have been trained on both the pretraining data and the finetuning data. Secondly, the various videos of the ClimbAlong dataset are very semantically similar. One the other hand, the pretraining dataset consisted of people breakdancing, throwing a baseball, bench pressing and performing sit ups, which are all very semantically different from each other, which could make it hard for the models to learn to generalize. Similarly, the samples from the pretraining dataset were all shot from very similar camera ankles, whereas the samples from the pretraining data were filmed from all kinds of camera ankles, resulting in the same keypoints being placed at different locations

Further, the BRACE dataset was not fully manually annotated, but instead annotated using a pose estimator and only certain incorrectly predicted poses were manually annotated, where incorrectly predicted poses were detected by a machine learning model. However, if either the machine learning model for detecting incorrectly predicted poses or the pose estimator delivered suboptimal results, then the annotations would also be suboptimal and thus contain some noise, which is very difficult for our models to replicate. The finetuning dataset on the other hand was completely fully annotated by humans, making the annotations much more consistent and thus easier for the models to replicate.

We also see a problem with the Penn Action dataset, which can have an effect on the pretraining results. For the BRACE dataset, it is clearly documented, that the video sequences is filmed using 30 frames per second. For Penn Action, on the other hand, a similar information is not stated anywhere and the video sequences come as individual frames without the duration of the video sequences noted, hence why we cannot either compute the frame rate of these video sequences. This could cause some problems, as two sample windows, one from each dataset, could span two different durations and thus capture two different duration of context, which can confuse the developed models. On the other hand, all of the video sequences of the finetuning dataset are all filmed using 30 frames per second, making the dataset much more consistent, which again makes the fitting easier.

Lastly, one major reason behind the performance differences behind the models on the pretraining dataset and the finetuning dataset is the perforamnce of the identity functions. If we compare the performance of the identity function on the pretraining dataset in Table 3 and Table 4 against the performance of the identity function on the finetuning dataset in Table 7 and Table 8, we clearly see, that the identity function performs much better during the finetuning than during the pretraining. Thus, a lot less temporal smoothing has to be done by our temporal-inclusive models, making the learning a lot easier.

7.3 Which Model is the Best for Temporal Smoothing?

	1.1	1.2	1.3	2.1	2.2	2.3	Total
3DConv	81.5	82.4	82.7	80.4	82.1	**80.8**	81.7
DeciWatch	**90.1**	**90.1**	**83.8**	**89.8**	**89.8**	68.2	**85.3**
bi-Conv LSTM Model S	76.3	75.6	76.8	77.0	76.0	75.5	76.2
bi-Conv LSTM Model C	76.3	77.4	77.3	77.1	77.2	76.4	77.0

Table 13: Mean of testing PCK@0.05, PCK@0.1, and PCK@0.2 of each finetuning run.

	Mean prediction time (ms)	Standard deviation of prediction time (ms)	Number of parameters
3DConv	0.39	9.29×10^{-2}	78,150
DeciWatch	14.2	0.11	1,708,388
bi-Conv LSTM Model S	10.6	1.68×10^{-2}	1,875,666
bi-Conv LSTM Model C	20.1	6.93×10^{-2}	1,872313

Table 14: Mean prediction time of each architecture on a single window from the whole finetune dataset, as well as the total number of parameters of each architecture. The time measurements are based on five runs and were measure on the machine described in section 6.5.

As we have now fuly developed and tested all of our models on all three experiments, we can decide which models yield the best results for temporal smoothing in bouldering.

We have in Figure **??** written the mean testing PCK@0.05, PCK@0.1, and PCK@0.2 of each finetuning run. Based on these numbers it seems like DeciWatch 1.1/1.2 would be the models to pick, as these yield the highest testing accuracy, based on the average PCK@0.05, PCK@0.1, and PCK@0.2 accuracies. However, as we know from section **??**, DeciWatch 1.1/1.2 delivers some of the least optimal results when considering PCK@0.2. On the other hand, if we instead consider PCK@0.05, then DeciWatch is generally the model that delivers the greatest results and by quite a big margin.

Thus, if one is looking for the run that is on average delivering the greatest results, then DeciWatch 1.1 or 1.2 would be the ideal choice. However, if one is just looking for a rough estimation of the position of the keypoints, then bi-ConvLSTM Model C 1.1 would be the ideal choice, as we know from section 6.4, that this run delivers the highest PCK@0.2 testing accuracy, at the cost of one of the lowest PCK@0.05 testing accuracies.

One could also consider the prediciton time and the size of each model. We have in Table **??** written the mean prediction time of each model on a single window, as well as the size of each model. From the table we can clearly see, that 3DConv is by far the fastest and smallest architecture. This, combined with the general decent results of 3DConv in Table **??**, could

mean, that 3DConv could be the optimal choice if one is concerned about the prediction time or size of the model.

7.4 General Reflections

Generally, we believe that we made the correct choices throughout the execution of the project. However, looking back throughout the project, we do find some bad decision made by us, that could have been avoided or made in another way.

7.4.1 Pretraining

For the preprocessing of the pretraining dataset, the goal was to preprocess the data, such that it simulated the output of the keypoint detector. This was done through multiple steps, where we used multiple values. For instance, when we expanded the bounding-boxes created from the keypoints by 10% in each direction, we used a standard deviation in the range $\{1, 1.5, 2, 2.5, 3\}$ for the Gaussian filter on the input data, and we shifted the input by sampling from a normal distribution with mean $\mu_{in} = 1$ and standard deviation 3 or 6, depending on the experiment. These values were all some values that we came up with. We could instead have picked these values in a more sophisticated manner such that the data would be more similar to the ClimbAlong data. This could for instance have been done by approximating them from the ClimbAlong dataset. We could further have made this standard deviation keypoint-dependent, such that it would have modelled how much movement each keypoint makes. However, we do not too sure how beneficial this would have been, as (1) we generally already receive some great results which are difficult to beat, and (2) for the experiments where we increased the added noise, we did not see an improvement on of the keypoints that carry a lot of movement, for instance the pinky, hence why do not believe that more moving keypoints require being shifted more than less moving keypoints.

For the last part of the preprocessing of the pretraining dataset, we split the data into a training, validation and test set, consisting of 60%, 20% and 20% of the data respectively. This was done by making sure that none of the windows were repeated and that none of the windows were overlapping between the three datasets. By doing so we ensured that the evaluation carried minimal bias, as the models had not seen any of the frames in the validation and test set during its training. However, different frames of the same video sequence could appear across the three sets, which could potentially carry some bias, as the same person appear accross the multiple sets. Instead, it would most likely have been better if we made sure that the same sets had no overlapping video sequences, as this would lower the likelihood of introducing any evaluation bias.

7.4.2 Finetuning

When creating the target heatmaps for the finetuning data, we occasionally met a problem, where the groundtruth keypoints was placed outside of the predicted bounding-box, which was used as target bounding-box. As the goal of our models is not to correct the predicted bounding-box, but instead just to correct the position of the predicted keypoints, we had to make a choice of what to do for keypoints that were supposed to be placed outside of the bounding-box. We generally saw two solutions to the, as we could either (1) completely remove the keypoint and just leave the heatmap empty, or we could (2) move the keypoint such that it was somewhere inside the bounding-box. As we expected the first solution to have a negative impact, as the models would just learn that the keypoint was missing, we simply went with the second solution, however, one could argue that the first solution would be the optimal choice, as the models are essentially learning the incorrect position of the keypoints.

7.5 Future Work

If we were to work further with this project, we first find it interesting to experiment with other machine learning methods. We would for instance find it very interesting to test the effects of letting DeciWatch process all frames, instead of only processing every fifth frame. Further, we find it interesting whether or not it would improve the results if DeciWatch was adapted, such that it made use of Vision Transformers as introduced by Dosovitskiy *Et al*. [6].

Further, we see some potential work in trying to avoid the overfitting during the finetuning-stage that we experienced in section 6 and find it interesting how much this would improve the final results. We did attempt some of this in section 6.3.1, however, without any luck. One could for instance further try to avoid the overfitting by (1) increasing the variance of the dataset by incorporating even more data augmentation, for instance by adding some noise to the data, or (2) decrease the complexity of the models, such that they have less tuneable parameters

Lastly, we suggest that the models could be retrained multiple times. In section 7.1 we blaimed some of the results on the random initialization of the weights of the models. By retraining the models multiple times, the likelihood of this randomness being the reasoning for these results decreases and thus we will be sure whether or not our results are due to the randomness of the weights.

8 Conclusion

Throughout this thesis we have succesfully developed and testign various machine learning models for extending an already developed keypoint detector for bouldering, such that it makes use of temporal smoothing. This was done by pretraining these models on the BRACE and Penn Action datasets and further finetune them on a dataset for bouldering, provded by ClimbAlong at Northtech ApS. Further, three experiments with two different setups were run for each model, such that we would end up with the optimal setting of each model. Lastly, we discussed our approach, including our results as well as mistakes we have made, as well as argued that the optimal choice of model depended on ones needs. Generally however, we find that DeciWatch by Zeng *Et al.* [31] yielding the most accurate results, the bidirectional convolutional LSTM that uses a convolutional layer for combining the two processing directions yielded the best rough estimations, as well as the simple 3-dimensional convolutional layer yielded the best results when also considering the size and prediction time of the model.

9 References

[1] André Oskar Andersen. "2D Articulated Human Pose Estimation, Using Explainable Artifical Intelligence". Bachelor's Thesis. University of Copenhagen, Department of Computer Science, 2020.

[2] Mykhaylo Andriluka, Stefan Roth, and Bernt Schiele. "Discriminative appearance models for pictorial structures". In: *International journal of computer vision* 99 (2012), pp. 259–280.

[3] Bruno Artacho and Andreas Savakis. *UniPose: Unified Human Pose Estimation in Single Images and Videos*. 2020. DOI: 10.48550/ARXIV.2001.08095. URL: https://arxiv.org/abs/2001.08095.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[5] Yucheng Chen, Yingli Tian, and Mingyi He. "Monocular human pose estimation: A survey of deep learning-based methods". In: *Computer Vision and Image Understanding* 192 (Mar. 2020), p. 102897. DOI: 10.1016/j.cviu.2019.102897. URL: https://doi.org/10.1016%2Fj.cviu.2019.102897.

[6] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV].

[7] Martin A Fischler and Robert A Elschlager. "The representation and matching of pictorial structures". In: *IEEE Transactions on computers* 100.1 (1973), pp. 67–92.

[8] Rohit Girdhar, Georgia Gkioxari, Lorenzo Torresani, Manohar Paluri, and Du Tran. *Detect-and-Track: Efficient Pose Estimation in Videos*. 2017. DOI: 10.48550/ARXIV.1712.09184. URL: https://arxiv.org/abs/1712.09184.

[9] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. *Mask R-CNN*. 2017. DOI: 10.48550/ARXIV.1703.06870. URL: https://arxiv.org/abs/1703.06870.

[12] Arjun Jain, Jonathan Tompson, Yann LeCun, and Christoph Bregler. *MoDeep: A Deep Learning Framework Using Motion Features for Human Pose Estimation*. 2014. arXiv: 1409.7963 [cs.CV].

[13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R. First Edition*. Springer, 2017.

[14] Sam Johnson and Mark Everingham. "Learning effective human pose estimation from inaccurate annotation". In: *CVPR 2011*. IEEE. 2011, pp. 1465–1472.

[15] Vaclav Kosar. *Cross-Attention in Transformer Architeacture*. https://vaclavkosar.com/ml/cross-attention-in-transformer-architecture.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[17] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[18] Yue Luo, Jimmy Ren, Zhouxia Wang, Wenxiu Sun, Jinshan Pan, Jianbo Liu, Jiahao Pang, and Liang Lin. *LSTM Pose Machines*. 2017. DOI: 10.48550/ARXIV.1712.06316. URL: https://arxiv.org/abs/1712.06316.

[19] Davide Moltisanti, Jinyi Wu, Bo Dai, and Chen Change Loy. "BRACE: The Breakdancing Competition Dataset for Dance Motion Synthesis". In: *European Conference on Computer Vision (ECCV)* (2022).

[20] Tomas Pfister, James Charles, and Andrew Zisserman. *Flowing ConvNets for Human Pose Estimation in Videos*. 2015. DOI: 10.48550/ARXIV.1506.02897. URL: https://arxiv.org/abs/1506.02897.

[21] Leonid Pishchulin, Mykhaylo Andriluka, Peter Gehler, and Bernt Schiele. "Poselet Conditioned Pictorial Structures". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2013.

[22] Moacir Antonelli Ponti, Leonardo Sampaio Ferraz Ribeiro, Tiago Santana Nazare, Tu Bui, and John Collomosse. "Everything You Wanted to Know about Deep Learning for Computer Vision but Were Afraid to Ask". In: *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*. 2017, pp. 17–41. DOI: 10.1109/SIBGRAPI-T.2017.12.

[23] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun Woo. *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. 2015. arXiv: 1506.04214 [cs.CV].

[24] Karen Simonyan and Andrew Zisserman. *Two-Stream Convolutional Networks for Action Recognition in Videos*. 2014. arXiv: 1406.2199 [cs.CV].

[25] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christopher Bregler. *Efficient Object Localization Using Convolutional Networks*. 2015. arXiv: 1411.4280 [cs.CV].

[26] Alexander Toshev and Christian Szegedy. "DeepPose: Human Pose Estimation via Deep Neural Networks". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2014. DOI: 10.1109/cvpr.2014.214. URL: https://doi.org/10.1109%2Fcvpr.2014.214.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[28] Yufei Xu, Jing Zhang, Qiming Zhang, and Dacheng Tao. *ViTPose: Simple Vision Transformer Baselines for Human Pose Estimation*. 2022. DOI: 10.48550/ARXIV.2204.12484. URL: https://arxiv.org/abs/2204.12484.

[29] Sen Yang, Zhibin Quan, Mu Nie, and Wankou Yang. *TransPose: Keypoint Localization via Transformer*. 2020. DOI: 10.48550/ARXIV.2012.14214. URL: https://arxiv.org/abs/2012.14214.

[30] Yi Yang and Deva Ramanan. "Articulated pose estimation with flexible mixtures-of-parts". In: *CVPR 2011*. IEEE. 2011, pp. 1385–1392.

[31] Ailing Zeng, Xuan Ju, Lei Yang, Ruiyuan Gao, Xizhou Zhu, Bo Dai, and Qiang Xu. *DeciWatch: A Simple Baseline for 10x Efficient 2D and 3D Pose Estimation*. 2022. DOI: 10.48550/ARXIV.2203.08713. URL: https://arxiv.org/abs/2203.08713.

[32] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. "Dive into Deep Learning". In: *arXiv preprint arXiv:2106.11342* (2021).

[33] Weiyu Zhang, Menglong Zhu, and Konstantinos G. Derpanis. "From Actemes to Action: A Strongly-Supervised Representation for Detailed Action Understanding". In: *2013 IEEE International Conference on Computer Vision*. 2013, pp. 2248–2255. DOI: 10.1109/ICCV.2013.280.

Appendix

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	38.6	50.1	44.5	76.8	76.7	68.3	51.2	23.2	34.7	31.0	23.1	26.7	45.4
Ear	33.4	40.3	37.4	77.3	77.2	69.1	40.0	28.5	35.9	44.3	31.1	55.3	47.5
Shoulder	51.4	62.1	59.9	77.5	77.7	69.7	42.0	34.8	28.6	34.6	45.6	33.0	51.4
Elbow	61.5	60.8	62.3	76.7	76.8	67.4	39.5	24.2	36.0	38.9	47.1	50.7	53.5
Wrist	44.0	46.5	44.7	75.9	75.9	67.0	30.4	34.1	40.0	34.9	39.5	52.4	48.8
Pinky	30.5	29.1	27.4	73.9	74.1	65.7	19.6	28.9	32.0	15.8	20.5	24.7	36.9
Index finger	39.1	38.9	40.4	74.6	75.0	66.4	32.4	33.0	40.8	34.0	40.3	34.3	45.8
Thumb	35.7	35.8	36.7	74.8	74.7	66.1	21.6	36.3	33.3	29.5	25.0	24.8	41.2
Hip	79.3	76.9	87.2	79.9	80.2	71.3	45.4	50.8	51.9	42.9	50.4	40.8	63.1
Knee	54.3	61.2	53.6	77.8	78.0	69.0	35.0	33.8	47.4	31.6	30.8	26.8	49.9
Ankle	51.2	57.6	55.5	78.7	78.9	67.8	49.9	43.7	43.5	50.2	57.2	40.8	56.3
Heel	64.4	62.8	69.0	77.9	77.5	67.0	34.3	33.3	38.0	37.8	36.5	36.2	52.9
Toes	57.9	57.0	61.2	76.4	76.4	68.6	57.2	43.8	44.0	40.7	51.6	53.0	57.3
Total	49.7	52.3	53.1	76.6	76.7	68.1	37.8	34.9	39.0	35.9	39.0	38.5	

Table 15: Keypoint-specific testing PCK@0.05-accuracies of the various models for shiting-scalar $s = 1$. All the accuracies are in percentage.

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	33.0	49.3	37.7	74.3	74.5	22.4	33.1	32.3	36.0	22.6	27.4	30.7	39.4
Ear	30.8	35.1	30.3	72.9	72.9	43.8	36.4	33.2	40.2	29.7	37.1	45.8	42.4
Shoulder	48.7	56.6	51.5	77.0	77.0	31.0	49.9	45.3	27.0	40.2	41.5	37.9	48.6
Elbow	56.8	58.8	54.4	76.6	76.6	30.2	34.2	47.8	37.1	54.8	45.6	32.6	50.5
Wrist	43.5	46.0	41.2	74.7	74.7	36.3	25.6	35.2	40.0	34.1	40.7	39.6	44.3
Pinky	28.6	29.5	28.2	73.6	74.0	30.6	27.5	20.6	28.0	21.2	24.8	31.3	34.8
Index finger	39.3	40.7	40.7	74.8	74.5	30.6	42.2	33.4	44.7	37.7	30.3	38.2	43.9
Thumb	36.0	35.3	36.6	74.9	74.4	28.1	28.1	28.5	16.1	24.2	24.3	24.6	35.9
Hip	78.2	75.7	82.6	80.0	80.1	42.7	55.3	27.6	53.9	53.9	59.9	8.22	58.2
Knee	39.7	57.5	37.5	78.3	78.1	24.8	44.8	26.1	17.4	47.1	42.8	50.7	45.4
Ankle	37.2	62.9	39.4	75.8	76.0	53.9	46.3	46.1	34.9	51.2	49.2	45.7	51.6
Heel	65.3	60.3	68.8	78.2	77.8	55.1	24.3	32.8	40.1	42.5	29.4	37.7	51.0
Toes	61.4	62.4	60.0	75.8	75.9	32.8	54.1	72.1	42.1	39.1	53.0	56.0	57.1
Total	46.5	51.6	47.3	76.0	75.9	36.8	38.8	37.4	35.9	39.2	39.5	37.1	

Table 16: Keypoint-specific testing PCK@0.05-accuracies of the various models for shiting-scalar $s = 2$. All the accuracies are in percentage.

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	1.1	1.2	1.3	
Nose	100	100	100	99.8	99.8	99.8	100	100	99.9	100	99.7	99.9	99.9
Ear	100	100	100	99.8	99.8	99.8	97.7	99.9	100	100	100	99.9	99.7
Shoulder	99.9	100	99.9	99.8	99.8	99.8	100	100	99.9	100	100	100	99.9
Elbow	99.9	99.9	99.9	99.4	99.4	99.4	100	100	100	100	99.9	100	99.8
Wrist	99.8	99.9	99.9	99.1	99.2	99.1	100	99.9	99.8	100	99.9	100	99.7
Pinky	93.4	93.1	94.4	98.3	98.4	98.3	97.2	98.8	97.0	98.0	99.0	98.6	97.0
Index finger	99.0	98.8	98.8	98.2	98.2	98.2	99.5	98.7	97.0	99.6	99.4	99.4	98.7
Thumb	98.9	98.8	98.9	98.3	98.3	98.3	96.8	99.6	97.8	99.7	98.6	99.6	98.6
Hip	99.9	100	100	99.7	99.7	99.7	100	100	100	100	100	100	99.9
Knee	100	100	99.9	99.7	99.7	99.7	100	100	100	100	100	100	99.9
Ankle	100	100	100	99.5	99.5	99.5	100	100	99.9	100	100	99.9	99.9
Heel	100	100	100	99.2	99.2	99.2	99.3	99.9	99.9	99.9	100	99.8	99.7
Toes	99.9	100	100	99.1	99.0	99.1	99.6	99.8	99.4	99.8	100	99.8	99.6
Total	99.2	99.3	99.3	99.2	99.2	96.1	99.4	99.7	99.2	99.8	99.7	99.7	

Table 17: Keypoint-specific testing PCK@0.2-accuracies of the various models for shiting-scalar $s = 1$. All the accuracies are in percentage.

	3DConv			DeciWatch			bi-ConvLSTM Model S			bi-ConvLSTM Model C			Total
Experiment	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3	2.1	2.2	2.3	
Nose	100	100	100	99.8	99.8	97.2	100	99.9	99.7	99.7	99.9	100	99.7
Ear	100	99.8	100	99.7	99.7	97.1	99.8	99.8	100	99.9	99.9	99.9	99.6
Shoulder	99.9	99.7	99.9	99.8	99.9	95.8	99.9	99.8	100	99.9	100	100	99.6
Elbow	99.8	99.9	99.9	99.5	99.4	90.8	100	99.5	100	100	100	100	99.1
Wrist	99.8	100	99.9	99.1	99.1	94.4	99.8	99.7	99.8	99.8	100	99.7	99.3
Pinky	93.1	93.7	93.9	98.4	98.4	86.5	97.7	98.0	97.9	99.1	96.0	98.2	95.9
Index finger	98.9	99.0	98.8	98.2	98.2	88.9	99.4	99.1	99.2	99.6	98.0	97.5	97.9
Thumb	98.6	98.6	98.6	98.3	98.2	90.1	95.7	96.6	98.6	97.5	98.3	99.6	97.4
Hip	100	99.9	100	99.7	99.8	95.1	99.9	99.8	99.9	99.8	100	99.9	99.5
Knee	100	100	99.9	99.6	99.7	95.1	100	99.9	99.9	99.9	99.9	100	99.5
Ankle	100	99.9	100	99.5	99.5	93.9	100	100	100	100	100	100	99.4
Heel	100	100	100	99.3	99.3	91.0	99.9	99.9	99.8	99.9	99.6	99.9	99.1
Toes	99.9	100	100	99.0	99.1	91.9	99.9	99.4	98.4	99.8	99.6	99.8	98.9
Total	99.2	99.3	99.2	99.2	99.2	92.8	99.4	99.5	99.3	99.6	99.3	99.6	

Table 18: Keypoint-specific testing PCK@0.2-accuracies of the various models for shiting-scalar $s = 2$. All the accuracies are in percentage.

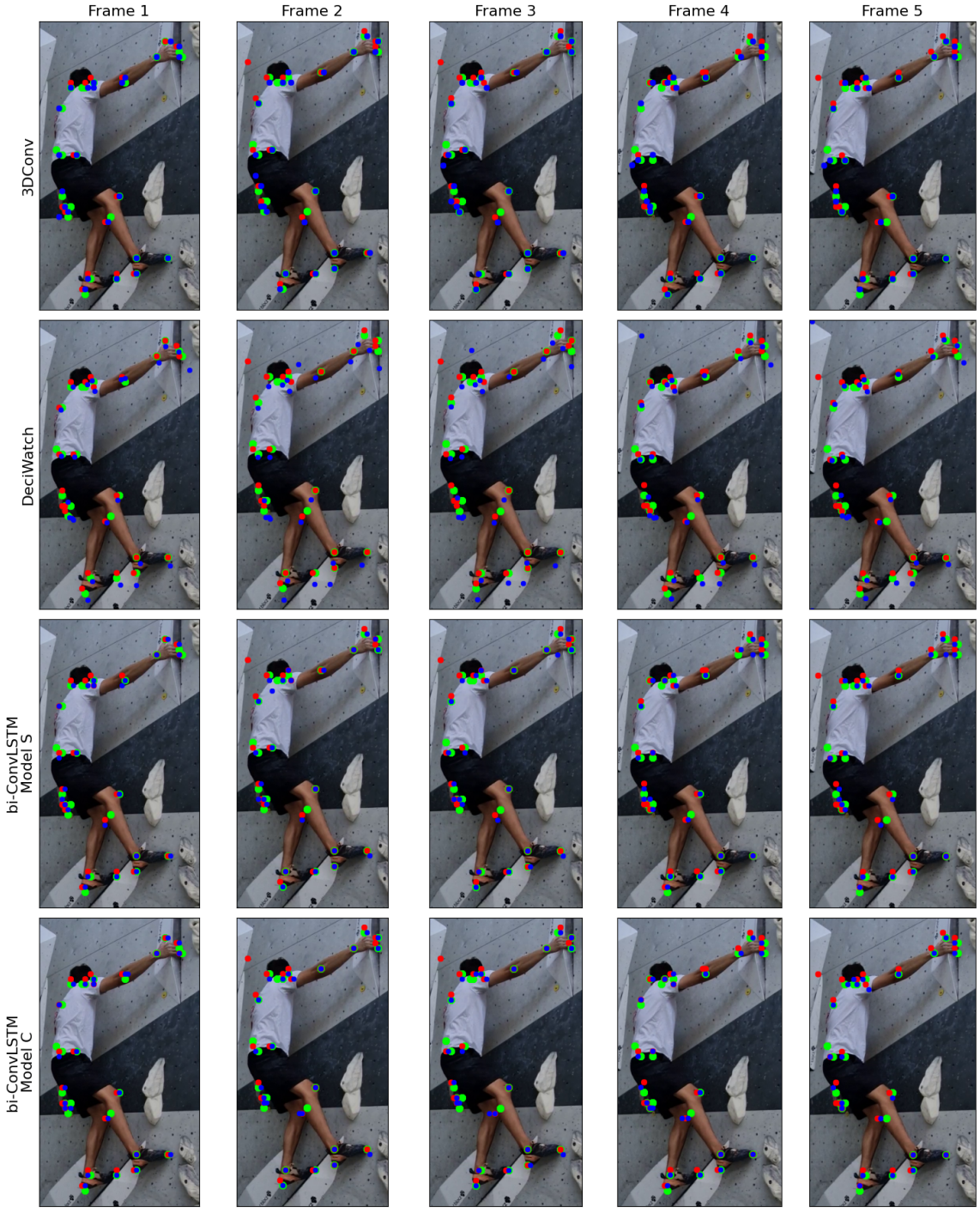


Figure 18: Ground truth keypoints (green), Mask R-CNN predictions (red) and predictions of the best setting of each model from section 6.4 on five consecutive frames from the single hold-out video.