FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

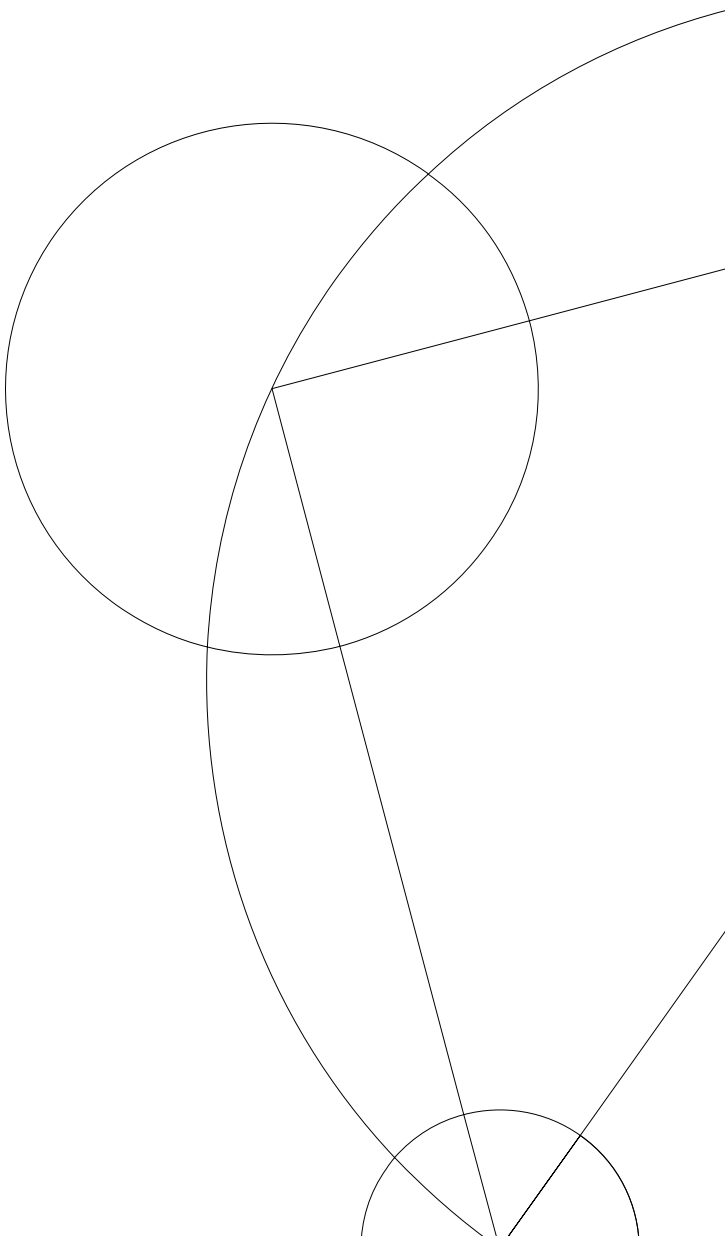
Master Thesis

2D Tracking in Climbing

Using Temporal Smoothing

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

2023



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

Abstract

Preface

Acknowledgement

Contents

Notation

$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A matrix
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

1 Introduction

1.1 Related Work

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

Image-based methods were initially based on the geometry between the joints of the taget image [16, 19, 24]. Following this, were the convolutional-based methods, that used convolutional neural networks [11] to perform the pose estimation [21, 14, 4, 7]. More recent methods use transformers [20] to deliver state-of-the-art results [22, 23].

Early video-based methods used 3-dimensional convolutions to capture the temporal information between neighboring frames [15, 5]. Other methods use LSTM's [8] to capture this temporal information [12, 2]. Like in the case of image-based methods, transformers [20] have recently been introduced to the video-based methods to capture the temporal information and deliver state-of-the-art-results [25].

1.2 Problem Definition

1.3 Reading Guide

2 Deep Learning Theory

The following section covers the most important background theory for the experiments in Section 5.3.3. This includes an introduction to various types of neural networks, as well as an introduction to the optimization of such networks.

2.1 Feedforward Neural Networks

Feedforward neural networks are the most basic type of neural networks. The aim of a feedforward neural network is to approximate some function f^*, by defining a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learning the parameters $\boldsymbol{\theta}$, that results in the best approximation of f^*. These models are called **feedforward** because there are no **feedback** connections in which the outputs of the model are fed back into itself. Instead, information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. Feedforward neural networks generally consists of multiple **layers**, arranged in a chain structure, with each layer being a function of the layer that preceded it [6].

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 a **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 [6]

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

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)** [6].

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 $k \times k$ [17]. 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

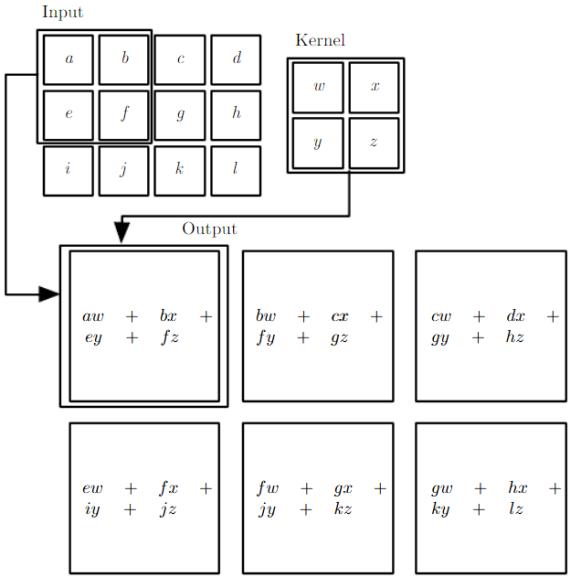


Figure 1: An example of applying a 2d kernel on an input [6].

compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure 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 [6].

2.2 Recurrent Neural Networks

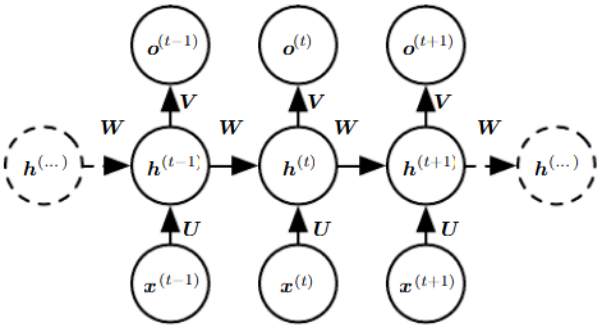


Figure 2: An illustration of an RNN [6].

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 of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. Generally, a RNN consists of three parts: (1) the input ($\boldsymbol{x}^{(i)}$), (2) the hidden state ($\boldsymbol{h}^{(i)}$), and (3) the output ($\boldsymbol{o}^{(i)}$). During inference, the model maps each input value to an 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 [6].

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)** for sequences of images. Both LSTMs work by potentially stacking multiple **cells** together, such that each LSTM cell is taking in outputs of the preceding LSTM cell as its input.

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 [6, 18].

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 image, 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** unit F_t (at time step 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** unit O_t, which controls whether or not the latest cell output will be proagated 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 images. 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 [18].

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 [20].

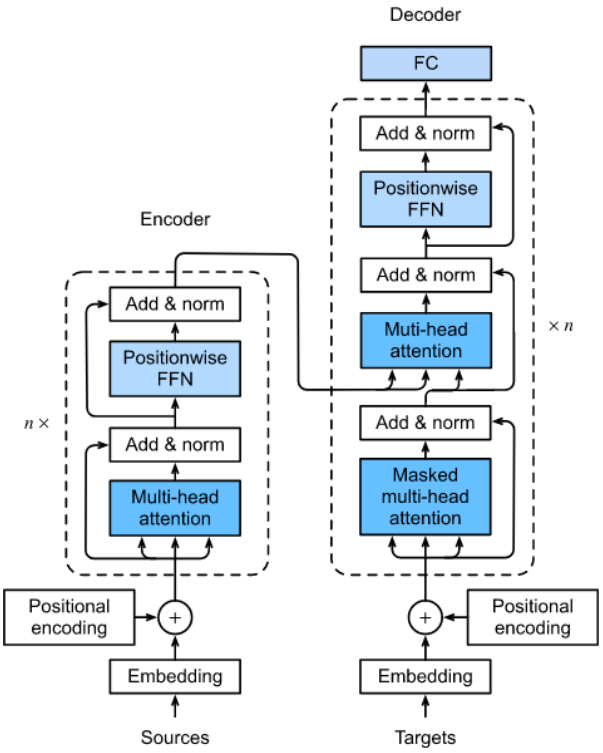


Figure 3: An illustration of the Transformer architecture [26].

2.3.1 Attention

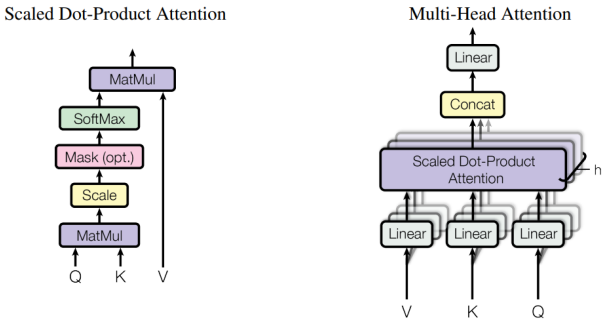


Figure 4: Illustration of (lef) the Scaled Dot-Product Attention, and (right) the Multi-Head Attention, which consists of several attention layers running in parallel [20].

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 [20]. 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 [20].

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 [20].

2.3.2 The Architecture

The Transformer follow an encoder-decoder structure, where the encoder maps an input sequence $\boldsymbol{x}$ to a sequence of continuous representations $\boldsymbol{z}$. The decoder then uses $\boldsymbol{z}$ to generate an output sequence $\boldsymbol{y}$, one element at a time. At each step the model consumes the previously generated output element as additional input when generating the next output [20]. Figure 3 illustrates the overall architecture of the Transformer.

Encoder: The encoder consists of 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 position-wise 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 [20].

Decoder: The decoder also consists of 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** [10]). 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 [20].

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 [20]

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

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 [20].

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 [9].

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 of a function 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, 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. [6].

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 samples of the dataset to comptute $\nabla L(f(\boldsymbol{x}, \boldsymbol{y}))$. 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 $|B|$ 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 [6].

Algorithm 1 Adam [6]

Require: Learning rate η
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$
Require: Small constant δ used for numerical stabilization
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m 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.

Thus, the summed inputs $\boldsymbol{w}_i^{(l)^\top} \boldsymbol{h}^{(l-1)}$ is 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 [3].
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** is used, which uses 20% of the torso diameter as its threshold [2].

Dropout: 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 technies called **regularization** to avoid the overfitting of the network. One common regularization technique is **dropout**, where the each parameter of the machine learning model is randomly disabled with probability p during training. By doing so the parameters are being trained on the dataset fewer times, leading to less of a probability of the model overfitting.

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 consecutive epochs, the training is terminated.

3 Models

The following section covers the architectures of the various models that will be used in Section 5.3.3 when we will be performing our experiments.

3.1 Baseline

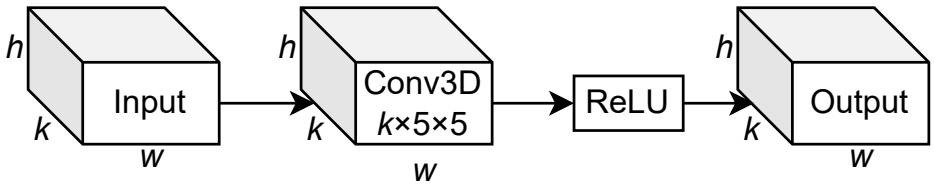


Figure 5: Illustration of the implemented baseline model.

The first model we will be using in Section 5.3.3 is a very simple baseline model based on a 3-dimensional convolution. Figure 5 illustrates the architecture of the model.

The model takes a sequence of $T \in \mathbb{R}$ 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 $K \in \mathbb{R}$ is the amount of keypoints in each estimated pose, $h \in \mathbb{R}$ is the height of each heatmap and $w \in \mathbb{R}$ 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 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

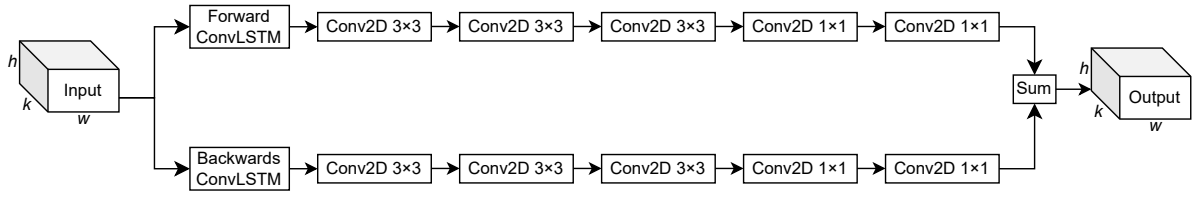


Figure 6: Illustration of the implemented bidirectional convolutional LSTM, where the two sequence orders are summed together.

Our second and third model are based on the LSTM-extension of Unipose-LSTM by Artacho and Savakis [2]. 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 $T \in \mathbb{R}$ 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 $K \in \mathbb{R}$ is the amount of keypoints, $h \in \mathbb{R}$ is the height of each heatmap and $w \in \mathbb{R}$ is the width of each heatmap.

3.2.1 Summing the Sequence Directions

Figure 6 illustrates the architecture of the second model. 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 $t \in \mathbb{R}$, 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 Concatenating the Sequence Directions

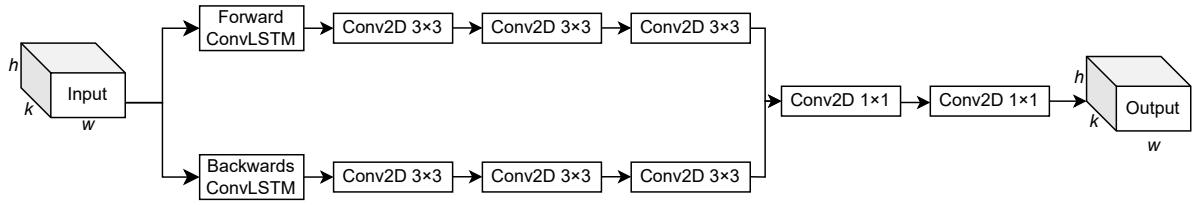


Figure 7: Illustration of the implemented bidirectional convolutional LSTM, where the two sequence orders are concatenated together and processed by two convolutional layers.

Looking at our second model, we see two problems with the architecture behind it: (1) the model does not have any opportunity of priorizing one processing order over the other, as the two branches are just summed together, and (2) by summing the two branches together it can occur, that some of the elements of the two branches are evened out, removing the prediction of the branches. Our third model aims at solving these two issues.

Figure 7 illustrates the architecture of our third model. This model is very similar to the previous model. It also starts of 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 of 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 convolutions, 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

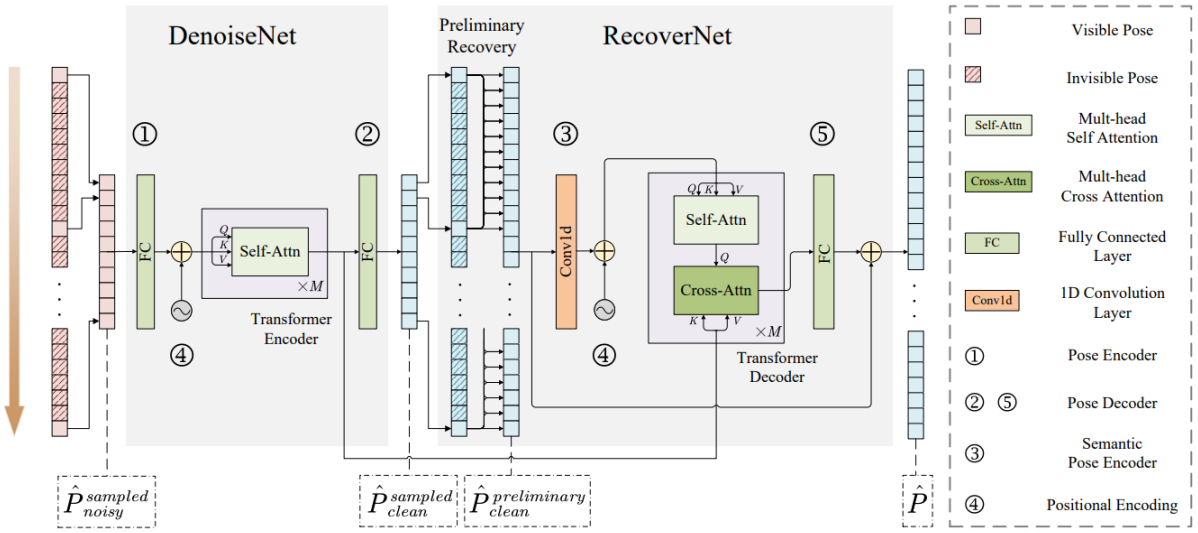


Figure 8: Illustration of the DeciWatch [25]

The last model we implement is a transformer-based model named *DeciWatch*, introduced by Zeng *Et al.*, 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 interpotation of the official paper behind DeciWatch [25].

More specifically, the model takes a sequences of $T \in \mathbb{R}$ estimated poses $\hat{\mathcal{P}} = \{\hat{\boldsymbol{P}}^t\}_{t=1}^{T}$ as input, where $\hat{\boldsymbol{P}}^t$ is represented by 2-dimensional keypoint position. Due to redundancy in consecutaive frames and continuity of human poses, 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 $K \in \mathcal{R}$ 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 $M \in \mathbb{R}$ 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 $C \in \mathbb{R}$ is the embedding dimensions. Lastly, another linear projection matrix $W_{DD} \in \mathbb{R}^{C \times 2K}$ is used to obtain the denoised sparse poses

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

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.

4 Dataset

To perform the pose estimation in Section 5.3.3, we need some data on which to train, validate and test our models. The following section describes the datasets that will be used.

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 ClimbAlong Dataset

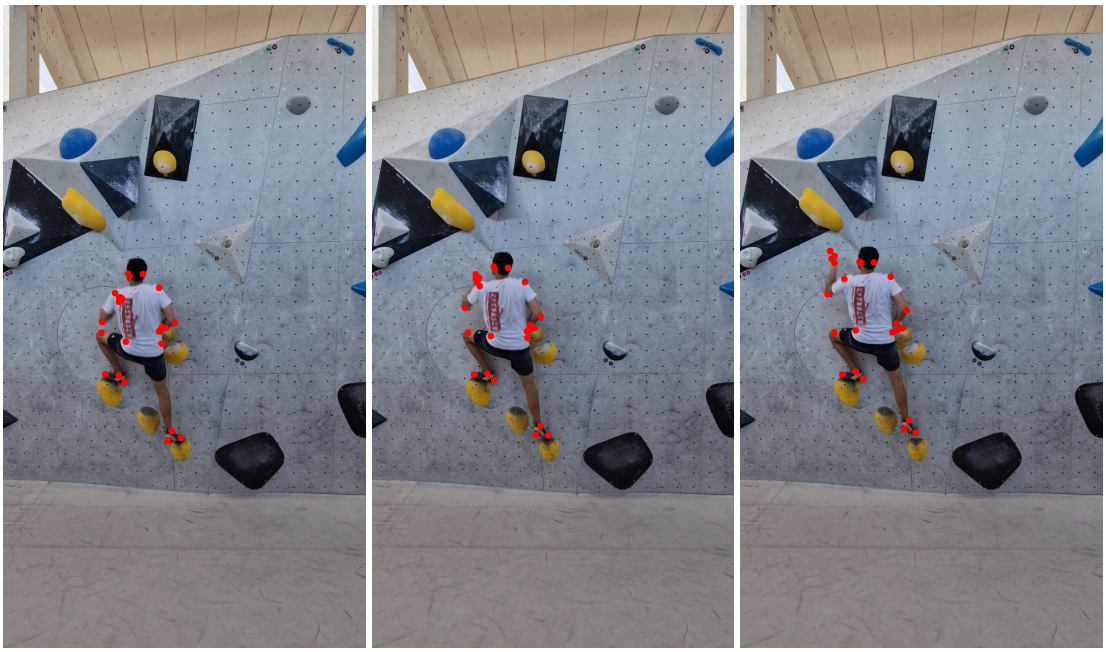
As the aim of our models is to perform well on climbers, we will be using some annotated data of climbers. For this, ClimbAlong ApS has developed a dataset that we will be using. The dataset consists of videos of various climbers on bouldering walls, where each video contains just a single climber. Figure 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 positions, where the climber holds a position for a while, as well as quick movements.

The dataset consists of 29 fully annotated videos and a total of $9,535$ 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

(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.

1080×1920 and 30 frames per second.

4.2 The BRACE Dataset

The second dataset we will be using is the *BRACE* dataset [13]. 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 [13].

We chose to use this dataset as breakdancers tend to swap between static and quick poses, as well as containing some acrobatic poses, similarly to the ones seen in the ClimbAlong dataset. Generally, the movements of the BRACE dataset are quicker than the movements 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. However, as both the actors of both datasets swap between static and quick poses, as well as both datasets containing acrobatic poses, we found the BRACE dataset relevant for our experiments in Section 5.3.3. 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 [13].

4.3 The Penn Action Dataset

The final dataset we will be using is the *Penn Action* dataset [27]. This dataset consists of $2,326$ video sequences of 15 different action-types. Table 2 lists these 15 action-types [27].

Each sequence has been manually annotated with human joint annotation, consisting of 13 joints as well as a corresponding binary visibility-flag for each joint. The frames have a resolution within the size of 640×480 [27].

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 sequences that have `baseball_pitch`, `bench_press` or `sit_ups` as their corresponding action-type [27]. Further, the movements of the Penn Action dataset tend to be of a more similar pace to the ClimbAlong dataset than the BRACE dataset, making the Penn Action dataset relevant for our task.

In total, we use 307 video sequences from the Penn Action dataset, consisting of a total of

(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.

(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.

(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.

$26,036$ frames. Figure 13 illustrates five consecutive frames with its groundtruth annotations for one of these video sequences.

5 Experiments

The following section describes the training of our models. This includes the the various experiments we perform, the data preprocessing of the data, the configuration details we use, as well as the obtained results.

5.1 Approach

As the 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 performancing well on the ClimbAlong dataset.

5.2 Pretraining

In the pretraining stage we will not be using the already developed pose-estimator, but instead only use our temporal-inclusive models by adding noise to the data such that it simulates the output of the already developed pose-estimator 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 already developed pose-estiamtor 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.

5.2.1 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 start by extracting the bounding-box of the annotated pose in each frame by using the annotated keypoints. Further, we expand each side by 10%, such that no keypoint lies on any of the boundaries of the bounding-box. To ensure that the aspect ratio of the pose is kept later on, we transform the bounding-box into a square by extending the shorter sides, such that they have the same length as the longer sides. Next, we discard everything outside the bounding-box and rescale the bounding-box to have a sidelength of 56, such that it has the same size as the output of the already developed pose-estimator.

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 $c \in \mathbb{R}$ 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 annotation 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 already developed pose-estimator, we will also need some data as input. We acquire this data by adding some noise to the data, such that they become similar to the output of the already developed pose-estimator, essentially simulating the output of the already developed pose-estimator. The noise is introduced by randomly shifting each keypoint of each sample and by smearing our each keypoint of each sample by using a Gaussian filter with mean $\mu_{in} = 1$ and some standard deviation $\sigma_{in} \in \mathbb{R}_{>0}$. For the shift-value we use $x \cdot k$, where the **shifting-scalar** $k \in \mathbb{R}_{>0}$ is some fixed positive number and $x \in \mathbb{R}$, is equal to 20% of the mean torso-diameter. We clip the position of the shifted keypoints between 0 and 55, such that they cannot be outside of their corresponding heatmaps.

5.2.2 Training Details

Data Configuration For the data we use a window-size of $k = 5$ frames, as Artacho *et al.* found this to be the optimal number of frames to use [2], 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. We insert the fixed value $c = 255$ in each heatmap at the position of the corresponding keypoint.

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 the 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.

Setups For each of the four models we use three different setups. In the first setup 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 already developed pose-estimator does 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 of the first setup makes, we fix this standard deviation in our second setup, 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 $k = 5$, 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 our last setup we still use a window-size of 5, however, with half the frame rate.

We further test the denoise capability of the models by running each setup twice for each model. In the first run we use a shifting-scalar of $k = 1$, whereas for the second run we increase this value to $k = 2$, making the data a lot more noisy.

Training Configuration For optimizing the parameters of the models, we use the ADAM optimizer with a batch size of 16, an initial learning rate of 10^{-3} and the MSE loss-function. During training, we keep track of the lowest reached validation loss of an epoch. If this lowest val-

idation loss has not been beaten for five consecutive epochs, we reduce the learning rate by a factor of 0.1. Further, if the lowest validation loss has not been beaten for ten consecutive epochs, we terminate the training of the corresponding model. In case this never happens, we terminate the training of the corresponding model once it has been trained for fifty epochs.

5.2.3 Results

5.3 Finetuning

In the finetuning stage we will be using the already developed pose-estiamtor 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 for the following three reasons: (1) the training of the models will be quicker, as we just need to train the tempoeral-inclusive models and not the already developed pose-estimator, (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 tempoeral-inclusive models, and (3) we lower the probability of overfitting, as we have less tuneable parameters.

5.3.1 Data Preprocessing

For the ClimbAlong dataset we perform only minor preprocessing. First, the preprocessing of each video is done by having the already developed pose-estimator 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.

5.3.2 Training Details

Data Configuration Generally, 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 $9,419$ windows. Also here are we using $c = 255$ as a representation of the placement of each keypoint. We also split the dataset into a non-overlapping and non-repeating training, validation and test set, consisting of 60%, 20% and 20% of the data, respectively. However, we note that one incorrect frame can have a huge impact on the evaluation results, as this frame is used five times during evaluation, due to the small dataset size. For that reason, for the validation and test set we make sure that none of the windows of the same set.

Setups As the finetuning dataset is so small, the fitting of the models is very quick, making us fit all of the developed models from the pretraining stage. 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 ADAM optimzer with a batch size of 16 and the MSE loss-function. 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 initial learning rate of 10^{-4}, as the weights only need to be fineadjusted, making us believe that greater learning rate would skew the weights too much.

5.3.3 Results

Technical Details

6 Discussion

- Ved ikke PA's fps
- BRACE er landscape
- Der er noget usikkerhed i BRACE da den er annoteret automatisk
- BRACE of Penn Action kan semantisk være meget forskellig
- Jeg shifter efter jeg har udvidet min data med 10%. Dette kan give noget bias, da der er større sandsynlighed for, at dataen ligger på billedets kant
- Vanishing gradient for unipose - den fungerer bedre på range [0, 255] end på range [0, 1]
- Configuration details (se Camilla's)
- Future work

7 Conclusion

8 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] 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.

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

[4] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. *OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. 2018. DOI: 10.48550/ARXIV.1812.08008. URL: https://arxiv.org/abs/1812.08008.

[5] 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.

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

[7] 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.

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

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

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

[11] 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.

[12] 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.

[13] 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).

[14] Alejandro Newell, Kaiyu Yang, and Jia Deng. *Stacked Hourglass Networks for Human Pose Estimation*. 2016. DOI: 10.48550/ARXIV.1603.06937. URL: https://arxiv.org/abs/1603.06937.

[15] 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.

[16] Leonid Pishchulin, Mykhaylo Andriluka, Peter Gehler, and Bernt Schiele. "Poselet Conditioned Pictorial Structures". In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2013, pp. 588–595. DOI: 10.1109/CVPR.2013.82.

[17] 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.

[18] 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].

[19] Yuandong Tian, C. Lawrence Zitnick, and Srinivasa G. Narasimhan. "Exploring the Spatial Hierarchy of Mixture Models for Human Pose Estimation". In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 256–269. ISBN: 978-3-642-33715-4.

[20] 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.

[21] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. *Convolutional Pose Machines*. 2016. DOI: 10.48550/ARXIV.1602.00134. URL: https://arxiv.org/abs/1602.00134.

[22] 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.

[23] 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.

[24] Yi Yang and Deva Ramanan. "Articulated Human Detection with Flexible Mixtures of Parts". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.12 (2013), pp. 2878–2890. DOI: 10.1109/TPAMI.2012.261.

[25] 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.

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

[27] 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.