TensorPose: Real-time pose estimation for interactive applications

Luiz José Schirmer Silva¹, Djalma Lúcio Soares da Silva¹, Alberto Raposo¹, Luiz Velho¹, Hélio Lopes¹

¹PUC-Rio - Pontifícia Universidade Católica do Rio de Janeiro
²IMPA - Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, Brazil

A R T I C L E   I N F O

Article history:
Received August 23, 2019

Keywords: Convolutional Neural Networks, Pose estimation, Real time applications

A B S T R A C T

The state of the art has outstanding results for 2D multi-person pose estimation using multi-stage Deep Neural Networks in images with high accuracy. However, the use of these models on real-time applications may be impractical not just because they are computationally intensive, but also because they suffer from flickering, from the inability for capturing temporal correlations among video frames, as well as from image degradation. To tackle these problems, we expand the use of pose estimation to motion capture in interactive applications. To do so, we propose a novel deep neural network with streamlined architecture and tensor decomposition for pose estimation with improved processing time, named TensorPose. We introduce an architecture for markerless motion capture using Convolutional Neural Networks combined with sparse optical flow and Kalman Filters. We also apply this architecture in a multi-user environment, based on the Holojam framework, where it is possible to create simultaneous collaborative experiences.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Pose estimation is a challenging problem in computer vision with many real applications in areas including augmented reality, virtual reality, computer animation and 3D scene reconstruction. Usually, the problem to be addressed involves estimating the 2D human pose, i.e., the anatomical keypoints or body parts of persons in images or videos. Recent works have presented great results for multi-person pose estimation in images, but still, have some issues when considering videos. These approaches can achieve high accuracy using architectures based on conventional convolution neural networks; however, mistakes caused by occlusion and motion blur are not uncommon, and those models are computationally very intensive for real-time applications. Also, despite some related work to increase performance, few papers explore the use of temporal coherence and the possibility of using such machine learning models for real-time applications involving animation.

OpenPose is considered the state-of-art approach on multi-person pose estimation, but it does not achieve the desired performance in terms of frames per second, which make it difficult to use in interactive applications that require frame rates close to or above 30 FPS. To increase the OpenPose performance, we propose a new pose estimation model, called TensorPose. In our model, we replace conventional convolution operations by successive pointwise and regular convolutions in a reduced space. As we will show, the proposed modifications is analogous to applying a tensor decomposition, more specifically a high order singular value decomposition (HOSVD). Another issue that we study is the temporal coherence, which is not present in previous works since they do not consider the relationship between the processed frames. To track the detected persons in videos and 3D captures, we use sparse optical flow and a Kalman filter to smooth the movement.

In this work, we also propose the creation of an architec-
ture that uses markless motion-capture for real-time applications. Such an architecture brings us the possibility of creating environments for the development of shared and distributed applications. With the collected information, our architecture easily composes an environment for the development of applications that can involve motion tracking using WebGL or a game engine. This software infrastructure can be used to implement multi-user interaction, positional tracking of people and objects, to create a complete sense of immersion in a tangible space, where it is possible to bridge the gap between the physical and virtual spaces. In other words, users in different physical environments and using various devices can share and explore a unique and integrated virtual environment.

In summary, the main contributions of this work are:

- a novel deep neural network with streamlined architecture and tensor decomposition for pose estimation with improved processing time;
- the extension of our model with Sparse Optical Flow and Kalman Filters for motion tracking; and
- an architecture for multi-user applications, where is possible to create simultaneous collaborative experiences based on the Holojam Framework.

The paper is structured as follows. Section 2 presents the related works considering pose estimation and motion capture. Section 3 describes an overview of pose estimation task using a neural network. Section 4 presents the main properties and operations with tensor, as well the decomposition that we use in our approach. Section 5 introduces the main aspects of our deep learning model and the software architecture for interactive applications. Section 6 shows the mains aspects of virtual environment communication based on the Holojam Framework. Section 7 shows the results and finally, Section 8 concludes the work by discussing our contributions and proposing future works.

2. Related Work

2.1. Pose Machines and Tracking

Ramakrishna et al. proposed [9] an inference machine framework and presented a method for articulated human pose estimation, called Pose Machines. A pose machine consists of a sequence of multi-class predictors, that is trained to predict the location of each part in each level of the hierarchy. Their method has the benefits of implicitly learning long-range dependencies between image and multi-part cues, tight integration between learning and inference, and a modular sequential design. The model was built on the inference machine framework so that it could learn reliable interconnections between body parts.

Wei et al. [10] introduced the Convolutional Pose Machines (CPMs) for the task of pose estimation. CPMs expand the main idea of pose machine architectures and combine them with the advantages of convolutional neural networks. Their contributions were the ability to learn feature representations for both image and spatial context directly from data, and the ability to handle large training datasets efficiently. CPMs consists of a sequence of convolutional networks that repeatedly produce 2D confidence maps for the location of each body part. At each stage in a CPM, image features and the confidence maps produced by the previous stage are used as input. As said by Wei et al. [10], the confidence maps provide the subsequent stage an expressive non-parametric encoding of the spatial uncertainty of location for each part, allowing the CPM to learn rich image-dependent spatial models of the relationships between elements.

Cao et al. [6] proposed an efficient method for 2D multi-person pose estimation with high accuracy on multiple public datasets, extending the original Convolutional Pose Machines. They created a bottom-up representation of association scores via Part Affinity Fields (PAFs), a set of 2D vector fields that encode the location and orientation of limbs over the image domain. They show that their approach encodes global context sufficiently well to allow for a combinatorial optimization algorithm that solves an assignment problem to detect a person’s skeleton. However, they did not consider temporal coherence in the original work. When considering a video, there is no relation between persons detected in multiple frames. Also, this approach is computationally intensive, demanding a lot of processing power. Following their algorithm and considering their benchmarks, we achieved approximately only 10 FPS using an Nvidia Titan Xp GPU.

In regards to the tracking problem, a solution is to use dense optical flow to adjust the predicted positions to generate a smooth movement across frames. The Thin-Slicing Network [11] achieved excellent results. However, this system is computationally very intensive and is slower when compared with the proposed by other papers.

Luo et al. [7] achieved some improvements in terms of both accuracy and efficiency. They propose a recurrent CNN model with LSTM for video pose estimation. Although considering occlusion, some incorrect predictions can be detected when the joint is not visible for an extended period. Also, their tests do not consider multi-person pose estimation.

2.2. Motion Capture and Mesh Recovery

Concerning motion capture, recent papers have presented promising solutions. Shafaei et al. [12] showed an efficient and inexpensive approach to markerless motion capture using a set of sensors, like the Microsoft Kinect. They split the multiview pose estimation task into three subproblems of dense classification, view aggregation, and pose estimation. Their approach also runs in real-time. They do not assume a shape model or a specific application, such as home entertainment, being applicable to a wider range of scenarios. However, their method considers only one person and not a multi-user system. Also, they use depth sensors like Kinect, and not regular cameras or stereo cameras as our architecture does.

Tome et al. [13] proposed a CNN-based approach for multi-camera markerless motion capture of the human body. Their technique makes use of 3D reasoning throughout a multi-stage procedure. They extended existing works on single view reconstruction to a multi-camera setting and showed how such single view methods could be enhanced by training on the multiview
based annotation of unlabelled data. However, this technique
seems to be intense, considering the computational cost and, for
multi-user detection, cannot be used in real time applications.

Mehta et al. [8] introduced the first real-time method to cap-
ture the full global 3D skeletal pose of a human in a stable, tem-
porally consistent manner using a single RGB camera. How-
ever, their approach suffers when there are significant move-
ment shifts between frames as well as occlusion. Also, the
method is only capable of tracking a single person.

Kanazawa et al. [14] show an end-to-end framework for re-
constructing a full 3D mesh of a human body from a single
RGB image. They used the generative human body model,
SMPL [15], which parameterizes the mesh by 3D joint angles
and low-dimensional linear shape space. Also, an adversarial
model is presented to adjust the mesh, determining if a pose can
be considered acceptable. An image is passed through a convo-
lutional encoder and sent to a 3D regression module that infers
the 3D representation of the human. The 3D parameters are
also sent to a discriminator, whose goal is to tell if these param-
eters come from a real human shape and pose. This mesh can
be useful to generate humanoid animations, which can immedi-
ately be used by animators, modified, measured, manipulated,
and retargeted. Again, this approach has a high computational
cost and only considers single person detection.

3. Pose estimation model

This section describes the main aspects of the architecture
proposed by Cao et al. [6]. The network architecture consists
of a feedforward neural network that predicts a set $S$ of 2D
"confidence maps", from which the body parts are located in
an image, and also a set $L$ of 2D vector fields that help iden-
tify the connection between two body parts to generate a skele-
ton. The set $S = \{S_1, S_2, \ldots, S_n\}$ has $n$ "confidence maps" one
for each part of the body (hand, elbow, head, etc) and the set
$L = \{L_1, L_2, \ldots, L_C\}$ has $C$ 2D vector fields, each used to con-
struct the skeleton, identifying some member of it: an arm or a leg
for example. $N$ candidates for each body part are generated,
creating for each pair a bipartite graph. Finally, the results are
analyzed following a greedy strategy, using the Hungarian algo-
rithm with some modifications to create a connection between
two parts. This process is performed in each frame.

The neural network is divided into two branches: one re-
ponsible for generating the “confidence maps”; and the other,
the affinity fields. Also, these two branches are divided into $t$
stages, where the authors of the original paper set $t = 6$ stages.
At first, the images are processed by the first ten layers of a
VGG-19 convolution network, generating a set of $F$ features
that will be used in the subsequent stages of the OpenPose.
These features are used as input for the other layers of the net-
work and, as result, they have a set $S$ of confidence maps and a
set $L$ of Affinity Fields (vector fields). Let $\rho$ and $\Phi$ represent
the convolution layers for the confidence maps and affinity fields.
At the end of this step, the results are concatenated with the ini-
tial set $F$ to produce refined results. This process is performed
for each subsequent stage.

To guide the network in its predictions, two objective func-
tions are given at the end of each stage, one for each branch.
The objective functions for both branches are based on the dis-
tance between the network output data and the ground truth gen-
erated for the confidence maps and vector fields, according to
Equations 1 and 2:

$$
J_f^s = \sum_{j=1}^{J} \sum_{p} W(p) || S_j^s(p) - S_j^t(p) ||^2_2;
$$

$$
J_f^l = \sum_{c=1}^{C} \sum_{p} W(p) || L_c^s(p) - L_c^t(p) ||^2_2,
$$

where $S_j^s$ and $L_c^s$ represent the ground truth of the annotated
data; $J$ and $C$ are, respectively, the confidence maps for each
part of the body, and the vector fields. The $W(p)$ is a binary
mask with $W(p) = 0$ when there is no data annotated at the
point $p$ of image. The mask is used because there are images
in the training dataset that are not entirely annotated, so it per-
tends true positive data from being penalized. The overall loss
function is defined by the sum of Equations 1 and 2.

The training images, as well as their annotations, were
obtained through the COCO (Common Objects in COntext)
dataset [16], using more than 100000 images for the task.

Following the architecture model, they first analyze the set
$S$ to identify the possible candidates for a part of the body.
In total, 18 confidence maps are generated by branch 1 defined
by the convolution network $\rho$. Each confidence map of the set $S$
can be viewed as a heatmap. For this, they used a technique
called Non-maximum suppression to identify the position of
potential candidates by getting the peaks in the image. With
all possible candidates (points) obtained, it is necessary to find
the real connections between them. Therefore, for each con-
nection, is built a complete bipartite graph where each node is
a part of the body, and each edge is a connection representing
an arm or a leg, for example. To find the right relationship,
they face a well-known problem from graph theory that is find-
ing the best match between the vertices of a bipartite graph: an
assignment problem. For this task, the model uses the vector
fields that were generated by the output of $\Phi$ of the network.
An integral line is computed along the segment described by
each pair of candidates, and since the integral line shows the in-
fluence of a given field along a line. Equation 3 shows how they
compute the score of each segment represented by each pair of
body parts. Given two points, $d_1$ and $d_2$, as candidates for two
parts of a skeleton forming a connection, the score of an edge
connecting these two points is:

$$
E = \int_{u=0}^{u=1} L_c(p(u)) \cdot \frac{d_{j2}^2 - d_{j1}^2}{||d_{j2} - d_{j1}||_2} du
$$

where $L_c$ represents the vector field and $p(u)$ interpolates the
position between two body parts $d_{j1}$ and $d_{j2}$.

$$
p(u) = (1 - u) d_{j1} + u d_{j2}.
$$

Afterward, to obtain the optimal solution, they used a classi-
cal method called Hungarian Algorithm. Regarding the detec-
tion of multiple people, determining the pose of each becomes
an n-dimensional matching problem. Since this is a considered
NP-Hard problem, some restrictions can be made. First, a minimum number of vertices for each body segment is chosen to generate a spanning tree instead of having the complete graph. Afterward, the problem is decomposed into a set of bipartite graphs to determine the matching between adjacent nodes independently. With all the connections defined, they make the union of all the connections that share common vertices, together with each other, to create the skeleton as a whole. Figure [43x587 to 295x729] illustrates the final result of the pose estimation.

### 4. Tensor Decompositions and Convolutional Neural Networks

This section presents Tensor analysis and its relation with Convolutional Neural Networks. This can lead us to better understand the connection between the HOSVD theory and the factorized convolutions. There is a rising interest in exploring different architectures for Convolutional Neural Networks based on factorized convolutions [17][18][19], and according to our opinion, the hidden theory behind these applications is related to tensor algebra and high order decompositions; although few papers discuss these fundamentals.

A 2D convolutional layer of a neural network can be described as a 4-dimensional tensor, where its dimensions are defined by the number of columns and rows, the number of channels of an input image, i.e., the RGB channels, and the number of output channels. In this work, we are interested in improving the pose estimation network for fast applications. To do so, we decompose convolutional layers into smaller tensors, i.e., we made approximations of these layers. This approximation is essential since it reduces the number of operations performed by each layer and, in this way, it accelerates our neural network model. The following subsections will discuss general aspects of tensors as well the Tucker decomposition [20][21][22][23].

#### 4.1. Tensor Properties

A tensor can be seen as a high-dimensional matrix, i.e., with three or more dimensions. The order of a tensor $\mathcal{J}$ is the number of its dimensions, also known as ways or modes [22]. In a manner analogous to matrices’ rows and columns, tensors have fibers. A matrix column is a mode-1 fiber and a matrix row is a mode-2 fiber [22][23]. Third-order tensors have column, row, and depth fibers for example.

In tensor analysis, there are operators that create subarrays by fixing some of the given tensors indices. In this way, when we fix all indexes leaving only one free, we create a fiber. In the same way, we create slices if we fix all indexes but leaving two free. For a third order tensor, the frontal, lateral and horizontal slices are obtained by fixing the third, the second and the first indexes, respectively.

Unfolding is similar to flattening a matrix, where we stack rows into a vector, and the columns into another vector. A mode-$n$ unfolding of the tensor 

\[
\mathcal{T} = \mathcal{T}(n) = \mathcal{T}(n, 1) \mathcal{T}(2) \cdots \mathcal{T}(n) \mathcal{T}(n+1)
\]

is the smallest number of rank-one tensors that generate $\mathcal{T}$ by computing their sum [22]. A rank-one tensor is a mode-$N$ tensor where it can be seen as the outer product of $N$ vectors [22][23] as we can see in Equation 5. In other words, each element of $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is the product of the corresponding vector elements defined by Equation 6.

\[
\mathcal{T} = \mathbf{v}^{(1)} \circ \mathbf{v}^{(2)} \circ \ldots \mathbf{v}^{(N)}
\]

The product of a mode-$n$ tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ by a matrix $M \in \mathbb{R}^{I_n \times K}$ is a tensor $\mathcal{X}$ with dimensions $(I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N)$ given by Equation 7.

\[
\mathcal{X} = (\mathcal{T} \times M)(I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N) = \sum_{k_n} {t_{i_1,...,i_{n-1}} m_{j_n,i_n}}
\]
4.2. Tucker Decomposition

A Singular Value Decomposition of a 2D matrix, i.e., a 2-order tensor, can be defined as follows in Equation (8):

\[ M = U \Sigma V^T \]  

where U is a unitary matrix \( m \times m \), \( \Sigma \) is a rectangular diagonal matrix \( m \times n \) and \( V^T \) is the transpose conjugate \( n \times n \). This equation can be rewritten as follows, in Equation (9) [26]:

\[ M = C \times U^1 \times U^2 \]  

where \( U^1 \) is defined as \((u^1_1 \times u^1_2 \times \ldots \times u^1_k)\) and \( U^2 \), defined as \((u^2_1 \times u^2_2 \times \ldots \times u^2_k)\), where both are unitary matrices, and C is a core matrix \((I_1 \times I_2)\).

The SVD decomposition can be generalized to High Order Tensors, where this approach is also known as High Order SVD or Tucker decomposition [26]. Such mathematical tool considers the orthonormal spaces associated with the different modes of a tensor [21]. For example, Equation (9) can be extended for 3D tensors as follows:

\[ M = C \times U^1 \times U^2 \times U^3 \]  

where \( U^1, U^2, U^3 \) contains the 1-mode, 2-mode, and 3-mode singular vectors, respectively, related to the column space of \( M_{mode=1}, M_{mode=2}, \) and \( M_{mode=3} \) matrix unfoldings. C is a core tensor with orthogonality property [26].

We could now apply these Tucker decomposition concepts to our domain [21]. Consider a mode-4 kernel tensor \( J \), which can be seen as a kernel in a convolutional layer. All operations for its decomposition can be written in the form described in Equation (11) [21]:

\[ J_{t,y,z,k} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} C_{r_1,r_2,r_3} U^1_{k,r_1} U^2_{y,r_2} U^3_{z,r_3} U^4_{t,r_4}. \]  

where C is a core tensor of size \((R_1 \times R_2 \times R_3 \times R_4)\) and the \( U_s \) matrices are factor matrices of sizes \( X \times R_1, Y \times R_2, Z \times R_3, \) and \( K \times R_4, \) respectively. Each dimension, in its domain, is associated with the dimensions defined by the number of columns and rows, RGB channels and the number of output channels in a convolutional layer of a neural network. Although this is true considering as input an RGB image, in successive layers the dimensions and number of channels vary. In this way, in intermediate layers, our tensors can be defined by the spatial dimensions \( X \) and \( Y \) of the input, the number of filters of the input data, i.e., the “depth”, and the number of filters to the output data of the layer. Considering this, the decomposition is performed over the weights of each layer of convolution in our Neural Network and, as we can see, the convolution kernels are 4D tensors. We can boost the speed performance of a neural network by creating several factorized approximations of regular convolutions with only a few reductions in accuracy. The next section presents the details of our model as well its relation with tensor decompositions.

5. TensorPose Network and Architecture

In this section, we describe the main aspects of our deep convolutional neural network model and the architecture for interactive applications. At first, we focus on the new CNN model. Secondly, we present the main aspects of our architecture for interactive applications. We describe the modules for inference of 2D and 3D data and for distributed applications.

5.1. Network Model and Skeleton Tracking

In real-time applications, efficiency and approaches for reducing the computational cost are essential. Concerning the model of Cao et al. [6], we propose a new deep neural network based on a streamlined architecture and Tensor decompositions.

Initially, we modified the generation of the first features using Depthwise Separable Convolution of MobileNet [17], which is typically used for embedded devices. We replace the original layers of VGG network because its performance in this step represents the first bottleneck. Here, we decrease the number of operations required in the convolution network process, making an approximation through successive convolutions. The MobileNet model is based on a form of factorized convolutions which transforms a standard convolution into a depthwise and a pointwise convolution [17]. Depthwise Separable Convolution deals not only with the spatial characteristics of the image but also with depth, i.e., the RGB channels. In this process, a kernel is separated into other two to perform two convolutions: a depthwise and a pointwise. Essentially, the depthwise convolution applies a single filter to each input channel and the pointwise applies a 1x1 kernel convolution to combine the outputs.

A Depthwise Separable Convolution requires approximately 8 to 9 times less computation than a standard convolution, with only a small reduction in accuracy [17]. Based on the architecture of the MobilenetV2 [27], we create 12 layers of Depthwise separable convolutions instead of regular ones for feature extraction.

Following this same idea, we create another lightweight structure for intermediate layers of the network intending to increase FPS performance. In the first three stages, we perform approximations using tensors to speedup the runtime performance without having significant losses in the network accuracy. Each intermediate convolution stage was replaced by a block consisting of a pointwise, a standard convolution, but with reduced space, and another pointwise convolution. For example, in the intermediate layers of our network, convolutions with input channels and output channels of \( n = 128, \) kernel \( 3 \times 3 \) and stride \( 1 \times 1 \) were replaced by a block containing a pointwise convolution with 128 input channels and 28 output channels, a regular convolution with 28 and 32, for input and output size respectively, with a kernel \( 3 \times 3 \) and another pointwise convolution with 32 channels for input and 128 channels for output. We use this approach to change the stages 1 to 3 of our network. This can be seen in an analogous manner as an approximation of a regular convolution described by Kim et al. [21], were they used a High Order SVD for this task. Similar as in the depthwise separable convolutions, we split a regular convolution into other 3, where it drastically reduce the computation.
and model size. Considering the tensor decomposition’s theory stated in the previous sections, let us relate it with convolutional layers. A regular convolution maps an input tensor into another with different size by successive operations as we can see in the Equation\(^\text{12}\):

\[
\text{conv}(x, y, z) = \sum_{i} \sum_{j} \sum_{k} \Theta_{i,j,k,z} W_{i,j,y-j,k} \quad (12)
\]

where \(\Theta\) is a kernel of size \(IJKZ\) (for example, a kernel with size \(I \times J = 3 \times 3\), with \(K = 3\) channels (RGB), and \(Z = 128\) convolution filters) and \(W\) an input tensor with size \(XYK\), as we refer typically as an image, for example, with \(X\) and \(Y\) the image dimensions and \(K\) the number of channels. Replacing \(\Theta\) by our approximation, this approach can be seen as kernel tensor decomposition [21] as we can see in the Equation\(^\text{13}\):

\[
\text{conv}(x, y, z) = \sum_{i} \sum_{j} \sum_{k} \text{approx} W_{i,j,y-j,k} \quad (13)
\]

Regarding the equation\(^\text{11}\) in our previous section, the equation\(^\text{14}\) defines the tensor named \text{approx} computed by the HOSVD decomposition \([20]\):

\[
\text{approx} = \sum_{i} \sum_{j} \sum_{k} U_{i,j,z,k}^1 U_{i,j,k}^2 W_{i,j,z,k} \quad (14)
\]

where \(U_{i,j,z,k}^1\) and \(U_{i,j,k}^2\) are the factor matrices of sizes \(KxR_1\) and \(ZxR_4\), respectively, and \(C\) is a core tensor of size \((I, J, R_1, R_4)\). Notice that this equation uses only the \(R_3\) and \(R_4\) ranks. In our application, we suppress mode-1 \((R_1)\) and mode-2 \((R_2)\), which are associated with spatial dimensions of an image, because they are quite small [21].

When replacing \(\Theta\) in the equation\(^\text{12}\) by the equation\(^\text{14}\) we can rearrange the operations obtaining 3 convolutional operations used to approximate the original: 2 pointwise convolutions and 1 regular convolution with reduced space. This leads us to the convolutional block previously described. Here the first pointwise convolution reduces the number of channels from \(K\) to \(R_3\), the regular convolution of \(K\) input and \(Z\) output channels has now \(R_3\) input channels and \(R_4\) output channels. Finally, the last pointwise convolution is used to get back the original output size. This process not only makes a speedup in our application, but also reduces the number of parameters needed.

A regular convolution requires \(D^2ZKXY\) multiplication-addition operations, where \(D^2\) refer to the dimensions of our kernel \(3 \times 3\) and \(X\) and \(Y\) refer to the spatial dimensions of our data. Considering the decomposition, our approach has a speed-up ratio \(S\) defined by the equation\(^\text{15}\):

\[
S = \frac{D^2ZKXY}{KR_3XY + D^2R_3R_4XY + ZR_4XY} \quad (15)
\]

Regarding our model, each decomposed layer requires approximately 9.3 times less operations than a regular convolution. Figure\(^\text{2}\) represents the architecture model of our network. Here, we have more layers of convolution, but the number of operations and weights is smaller, since each regular convolution is substituted by other 3 factorized. As we can see in figure\(^\text{2}\) for stage 1, the overall number of convolutions, considering all blocks, is \(3 \times 3 + 2\) and for stages 2 and 3 is \(5 \times 3 + 2\). We defined this model for the first three stages, after the feature extraction, in an empirical way, where we did not observe significant differences in the accuracy of our network when comparing to the OpenPose. If we extend this approach to the other stages, for example, to stage 4, we verified a more significant difference in the accuracy, more than 25%, leading to wrong results in inference. Also, if we apply to the whole 6 stages, we face an instability problem [21, 28], where it is difficult to find a good learning rate as a well how to initialize the weights. We believe that is an effect of applying stacked decomposed layers to the model, and for this reason, we limited the number of decomposed stages to the first three. Figure\(^\text{7}\) shows some results of the inference using our network. As we can see, in major cases, we can achieve a good performance in the detection despite some small errors considering high movement shifts and blurred images.

Despite frame rate issues, another problem identified in the OpenPose was the lack of temporal coherence in the original paper. In other words, there is no relation between the objects defined in one frame to the ones defined in the subsequent frames. Therefore, the reference is lost for each identified person. For the context of our applications, it is necessary to create a module for this task. Temporal coherence enables us to actually implement real motion tracking, where the temporal correspondence between each pair of the frame needs to be preserved, considering motion coherence between spatial neighbors and across the temporal dimension [29]. We also can use this to fix the tracking in a single person, removing problems involving people interaction, for example.

To solve this problem, in a first experiment, we create a module for temporal coherence using Optical Flow. Optical Flow is a pattern of apparent motion of objects in images between two consecutive frames caused by the movement of an object or the camera. In other words, it is an approximation of image motion based upon local derivatives in a given sequence of images. Some patterns can affect the sequence of images, causing temporal variations in the brightness. In sum, it specifies how image pixels moves between subsequent frames.

Firstly, we tested the Kanade-Lucas-Tomasi (KLT) algorithm [30], and our preliminary results not only presented an increase in the performance of the technique considering the frame rate, but also smoothness in the motion capture. For the KLT algorithm, we pass the points of skeletons detected by the network. At each interval of \(t\) frames, these points are iteratively tracked through the optical flow, where the previous frame, the position of the previous points, and the next frame are passed to the detection function. With each new frame, the location of each point of the tracked skeletons is updated. If any position or point are lost in the process, the tracking is stopped. Then, the network is executed again and the skeletons recalculated. The same is done at the end of the interval of \(t\) frames to guarantee the consistency of the tracking. Normally we use an interval of 10 frames, where we do not where we did not detect a greater error in the tracking when compared with the technique frame to frame. Here, the sparse optical flow deals with much less...
parameters than the use of the CNN and the calculation of line integral at each frame, where it tracks just the previous point detected by the CNN. The optical flow assumes that the intensity of the pixels of an object does not change with time and neighboring pixels have the same pattern of movement. Classical approaches, such as optical flow fail when it comes to environments with illumination changes and long-range motions. To solve these issues, we change the algorithm to use the Robust Local Optical Flow [31], following the framework proposed by Senst et al., while taking into account an illumination model to deal with varying illumination. Here the computational cost of this variation, used by the KLT method, is given by the upper bound of $O(nN_{\text{large}}l)$, where $n$ is the number of computed motion vectors and $N$ is the number of pixels of the larger image region $i$ [31]. Also, as support to the tracking process, we use the Kalman filter and Multiple Instance Learning (MIL). The Kalman filter [32] is used to predict the position of the skeleton in the frame after the current one, and to ensure that the references for each skeleton detected are maintained. Given the distance between the points calculated by the network and those predicted by the Kalman filter, a threshold is given to ensure that the reference points are maintained. We fix this threshold considering a radius of 20 pixels of error. Combining Optical Flow and Kalman filter gives us the possibility to interpolate the data over subsequent frames, which enable us to smooth the captured movement. We also use the MIL [33] just to ensure that the detected person’s reference is maintained. We use it as an additional feature to identify the bounding box of detected head keypoints, and to maintain the reference of a person, even when the network is called. Although their use is optional.

5.2. TensorPose Architecture

TensorPose was developed with the aim of being a platform for the development of 2D and 3D applications that involve motion capture. In this section, we will describe the general aspects of its architecture, as well as the framework for network communication of different applications. Such applications can be computer animation, games and also shared virtual experiences. One of our goals is to provide developers with a way to create environments for shared 2D or 3D experiences where users share a virtual environment, but not necessarily at the same physical environment. The motion capture data can be used to track the activities of users in this kind of application. In our architecture, each module is independent in terms of video processing, the inference of captured skeletons, temporal coherence, and information transmission. Figure 4 represents the architecture model.

As one can see in Figure 4, the architecture model is composed by several modules. Next we will describe the functionality of each one.

Pose Client. This module is responsible for processing video data from different cameras. Here, each frame is prepared using the OpenCV library [34], where we also do an equalization of the histogram to improve the contrast in the images and send the data to the inference module or the temporal coherence module. In this case, here, the resolution of the image is 1280x960, and the network resolution is 656x386 to fit in GPU memory.

Pose Inference. This neural network module receives data from the pose client and performs the inference of skeletons in an image. Also, a post-processing is performed to solve the assignment problem. For each set of key points detected and for each person, a dictionary is created, which contains the 2D position of each body part and the connections between them. For each recognized person is given an Id and their corresponding dictionary is added to a list of tracked persons who are sent to the module responsible by the instantiating objects, which will represent the capture of those people.

Zed SDK. In this stage, we use the ZED stereo camera SDK [35] [36] for 3D data capture. This module was developed us-
ing the Python API, where data from a 2D image is sent to the Pose Client to be processed and the 3D data is sent to the depth module.

**Depth Front.** This module is responsible for processing the depth data of one or more ZED cameras and receiving intrinsic camera data. ZED has two cameras separated by 12 cm, which capture a high-resolution 3D video of the scene and estimate depth and motion by comparing the displacement of pixels between the left and right images. This module stores a distance value \( Z \) for each pixel in the image.

**2D/3D Positions.** This part of the architecture receives data from the positions inferred by the network and creates objects identifying each detected person. In case of 2D, keeps the positions received in coordinates of the image. In the case of 3D, it receives data from Front-Depth and performs transformations in 2D coordinates for positions in the 3D world. It is also responsible for passing information to the temporal coherence module, also processing the Kalman filter, identifying the points to be tracked and ensuring the reference of the detected skeletons.

**Temporal Coherence.** This module tracks the points using the Optical Flow algorithm. It receives the initial data detected by the network, and every \( t \) frames perform the tracking of the points.

**Holojam Emitter.** Uses the Holojam platform to send the captured data via the network. It is a Python client that communicates with the Holojam Server. The Holojam platform is described in the following section.
6. Virtual environment communication by Holojam

Holojam is a virtual space sharing platform developed by the Future Reality Lab of the New York University [37]. This platform consists of the Holojam Node library and the Holojam SDK project. It enables content creators to build complex location-based multiplayer VR experiences in a simple and unified Unity project. The development framework provides an extensible and clean interface, allowing rapid prototyping. Additionally, it abstracts away specific VR hardware, promoting a flexible and customizable creation of virtual reality experiences. We use the Holojam protocol framework in our architecture to communicate applications with very low latency, and to send motion capture information through the network, expanding the use to not only VR applications.

6.1. Holojam Node

We use the Holojam Node, which is a client-server library developed in Node.js and targeted to applications running on a local network or over the web. One of its main characteristics is to perform low-latency communication between the various clients and the server. It consists of the following components: relay, emitters, sinks, and clients.

The relay component is responsible for routing the messages (updates and events) in a Holojam network. It acts both as a central server, which collects data received through a preconfigured (upstream) address, as well as performs a broadcast of this data through a multicast (downstream) address.

In addition to the relay in a Holojam network, there are also several nodes, called endpoints. Endpoints are either emitters, which only send upstream data; sinks, which only receive data through the downstream; or clients that receive downstream data and send upstream data. Holojam Node also provides a WebSocket interface where you can receive and transmit data over the web.

6.2. Holojam Protocol

Packets routed through a network are either hosted or updated. An update is essentially an array of flakes (generic Holojam objects). An event has an array containing only one flake. There is also a notification that is an abstraction of an event that includes just the label.

6.3. Holojam Objects

Holojam provides two types of objects: Nuggets and Flakes. These objects are defined through Google’s FlatBuffers Interface Description Language (IDL). We use Flatbuffers to serialize data, as well as streaming data, because it offers very low processing overhead. As follows, we present the structure of the objects used.

Nugget Composition:
- It is event type or update. Default is update
- It is mandatory to have an array of flakes
enum NuggetType : byte { UPDATE, EVENT }

// Message (update or event)
table Nugget {
    scope : string; // Namespace
    origin : string; // Source
    type : NuggetType = UPDATE;
    flakes : [Flake] (required); // Data array
}

Flake Composition:
- A label is mandatory

```c
table Flake { // Data container
    label : string (required);
    // Optional data
    vector3s : [Vector3];
    vector4s : [Vector4];
    floats : [float];
    ints : [int];
    bytes : [ubyte];
    text : string;
}
```

6.4. Holojam SDK

The Holojam SDK project is a Unity3D project containing all the elements needed to create a multiplayer virtual reality application. This project provides a C# API that allows the application created to use or extend this API allowing for rapid prototyping. Also, it abstracts the configuration of the VR hardware used in the project. One of its main components is the implementation of the Holojam client in C#. Thus, all Holojam objects can be integrated easily into the Unity project.

6.5. Holojam and TensorPose applications

All components of the Holojam platform were used in the TensorPose project applications, both in 2D applications as well as 3D and VR applications. Since TensorPose is implemented in Python, the components needed to use Holojam in TensorPose were also deployed in Python, including a serializer for the keypoints and the emitter. The serialization of the obtained keypoints was implemented from the IDL FlatBuffers of Holojam. With this, it is possible to send the keypoints to the relay, giving access to data to any sink/client/websocket that is connected to this relay.

Figure 5 shows the components for various applications created in the TensorPose project and to which other component they are related to.

7. Experiments

Our proposal was tested and compared with the results obtained by OpenPose 1.3. The TensorPose code was implemented in Python, including the post-processing steps, such as the line integral calculation and the assignment problem. In training, 110000 images were used as well as 5000 images in validation, with a maximum of 200 iterations per epoch. In COCO dataset, approximately 150000 people and 1.7 million labeled keypoints are available. Considering the inference, we use the COCO validation dataset to compare our modified model and the OpenPose. COCO uses a metric called Object Keypoint Similarity (OKS), which measures of how close the predicted Keypoint is to the ground truth annotation. Also, COCO uses the mean average precision (AP) and average recall (AR) over 10 OKS thresholds as the main competition metrics. In our initial tests, we consider Precision and Recall at 50% and 75% OKS (AP-50, AP-75, AR-50, AR-75) which are usually used for benchmark. We test both models with the evaluation data from COCO and submit the results to the COCO evaluation server. Equation 16 shows the formula used to calculate the OKS.

\[
OKS = \frac{\sum_i e^{-\frac{d_i^2}{\sigma^2}} \delta(v_i > 0)}{\sum_i \delta(v_i > 0)}
\]

where \(d_i\) are the Euclidean distances between each detected keypoint and the ground truth, \(v_i\) are the visible flags for each annotated keypoint, and \(s \times k\) (scale \times keypointconstant) is related to a standard deviation of a gaussian related to concentric circles where their radius varies by keypoint type. The metrics AP-50, AP-75, AR-50, AR-75 are related to how close a prediction is to annotated data considering the Gaussian standard deviation. The \(s \times k\) is computed by evaluating this Gaussian function, centered on the ground-truth position of a keypoint and the standard deviation is specific to the keypoint type, which is scaled by the area of the instances, measured in pixels. Table 1 shows our results in contrast to OpenPose and Figure 6 shows a visual comparison between the 2 techniques. Also, varying the OKS threshold from 0.5 to 0.95, we measure...
both precision and recall for the COCO validation dataset, as we can see in the figure where highest values mean how close our predictions are to the ground truth annotations considering each threshold.

<table>
<thead>
<tr>
<th>Model</th>
<th>AP 0.50</th>
<th>AP 0.75</th>
<th>AR 0.50</th>
<th>AR 0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenPose</td>
<td>0.780</td>
<td>0.593</td>
<td>0.807</td>
<td>0.647</td>
</tr>
<tr>
<td>TensorPose</td>
<td>0.750</td>
<td>0.536</td>
<td>0.772</td>
<td>0.591</td>
</tr>
</tbody>
</table>

Table 1. Precision and Recall for OpenPose and TensorPose Models.

Here, higher OKS means higher overlap between predicted keypoints and the ground truth. As we can see, our model has about 6.5% lower accuracy than OpenPose, which is mainly caused by our proposed simplification on the intermediary layers. Just to remember that this simplification promotes an advantage of having 9.3 times less operations in these layers than the original OpenPose. We have focus on real-time application, so this accuracy loss does not represent significant problems, since some prediction errors are acceptable.

Following the benchmarking and error diagnosis for pose estimation proposed by Ruggero et al. [38], we analyze 4 types of localization errors using the annotations of COCO dataset:

- Jitter: small position error around the correct keypoint location;
- Miss: large localization error when the detected key point is far away of any body part;
- Inversion: confusion between semantically similar parts of a detected person, i.e., wrong body part associations for the same person due problems like self occlusion, for example;
- Swap: confusion between semantically similar parts of different persons, i.e., problems related to the association of body parts of a detected person to another.

Table 2 shows the total of correct predictions and errors and Table 3 presents the estimate error associated to each detected joint considering our model. As we can see, most of the errors are related to the small difference between the predict keypoint and the annotated data and to significant localization error, which we believe this is due to false positives detected. We can also see that most localization errors affect keypoints more sensitive to errors related to occlusion or that usually involves some interaction between people such as the arms.

<table>
<thead>
<tr>
<th>Keypoints</th>
<th>Jitter %</th>
<th>Inversion %</th>
<th>Swap %</th>
<th>Miss %</th>
</tr>
</thead>
<tbody>
<tr>
<td>nose</td>
<td>9.6</td>
<td>0</td>
<td>2.4</td>
<td>5</td>
</tr>
<tr>
<td>eyes</td>
<td>16</td>
<td>4.8</td>
<td>4.2</td>
<td>7.4</td>
</tr>
<tr>
<td>ears</td>
<td>13.1</td>
<td>0.3</td>
<td>3.6</td>
<td>6.1</td>
</tr>
<tr>
<td>shoulders</td>
<td>12.8</td>
<td>12</td>
<td>21.9</td>
<td>7.1</td>
</tr>
<tr>
<td>elbows</td>
<td>12.6</td>
<td>4.6</td>
<td>17.5</td>
<td>14.3</td>
</tr>
<tr>
<td>wrists</td>
<td>9.7</td>
<td>10.8</td>
<td>15.2</td>
<td>21.8</td>
</tr>
<tr>
<td>hips</td>
<td>13.5</td>
<td>24.2</td>
<td>14</td>
<td>11.7</td>
</tr>
<tr>
<td>knees</td>
<td>7.1</td>
<td>22.7</td>
<td>11.1</td>
<td>13.4</td>
</tr>
<tr>
<td>ankles</td>
<td>5.5</td>
<td>20.6</td>
<td>10.1</td>
<td>13.2</td>
</tr>
</tbody>
</table>

Table 3. The frequency of localization errors over all the predicted keypoints. We have 62790 joints detected in 5000 images with this test.

In terms of runtime performance comparisons, we begin with the test of the CNN processing. Considering just one image with 23 people, we compared our approach with OpenPose. The tests were performed in a GPU Nvidia RTX 2060 with 6GB of memory, where we vary the scale of the image tested by a factor of 0.5 and repeat each test 1000 times. As we can see in Table 4, in average, our approach has a better performance when compared with the OpenPose.

<table>
<thead>
<tr>
<th>Image Resolution</th>
<th>OpenPose</th>
<th>TensorPose</th>
</tr>
</thead>
<tbody>
<tr>
<td>328 x 193</td>
<td>~55.08 ms</td>
<td>~54.86 ms</td>
</tr>
<tr>
<td>656 x 386</td>
<td>~120.22 ms</td>
<td>~84.25 ms</td>
</tr>
<tr>
<td>984 x 579</td>
<td>~174.75 ms</td>
<td>~116.66 ms</td>
</tr>
<tr>
<td>1312 x 772</td>
<td>~320.18 ms</td>
<td>~210.41 ms</td>
</tr>
</tbody>
</table>

Table 4. CNN processing time for OpenPose 1.3 and TensorPose. We vary the scale of the image tested by a factor of 0.5 considering 4 image scales.

When processing videos, to avoid processing the CNN and calculating the line integral and the execution of the Hungarian algorithm at each step, we use sparse optical flow. We get not only performance improvements, with a frame rate above 30 FPS, but also smoothness in motion tracking, also due to the Kalman filter. In the first test, we defined a fixed interval of 10 frames, where the optical flow is processed and, afterward, its parameters are updated by a new processing step in the neural network. Such range was defined empirically, where we saw that there was not a significant difference in the accuracy of the tracking. Our strategy, considering both changes in the network and the use of optical flow during frames, reduces the overall time for tracking and increases the frame rate, as we can see in Table 5.

Similar to the OpenPose, we analyzed some limitations when our approach fails. We face the same problems with highly crowded images where people are overlapping, and the approach tends to merge annotations from different people while missing others. Also, our application has less precision when compared to previous work due to the factorized layers. It has more difficulty to detect joints considering occlusion having a slight noise in the inferred data. As we can see in Table 6, this is not a significant problem as the results are comparable and we still have a good performance gain.

Subsequently, with the trained network, we defined an architecture for the creation of applications for motion capture, connecting different cameras and applications, such as scenes in the Unity3D engine and web applications. The initial model...
Fig. 6. Comparison of between the OpenPose model and ours. The first figure is generated by the OpenPose and the second by our model. As we can see, our model is slightly less accurate than the original work and fails to find the exactly position of a body joint in particular cases. Although due to the nature of our application, this issue can be considered acceptable. Image from COCO dataset.

Table 5. Frame rate comparison between the OpenPose 1.3 and TensorPose. As we can see, our model surpasses in approximately 3x the performance of the OpenPose model used as base.

<table>
<thead>
<tr>
<th>GPU</th>
<th>CPU</th>
<th>OpenPose</th>
<th>TensorPose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia Titan RTX</td>
<td>Intel(R) Core i9(R) CPU @ 3.3GHz</td>
<td>~11. FPS</td>
<td>~36.8 FPS</td>
</tr>
<tr>
<td>Nvidia Tesla P40</td>
<td>Intel(R) Xeon(R) CPU E5-2630v3 @ 2.40GHz</td>
<td>~10 FPS</td>
<td>~34.43 FPS</td>
</tr>
<tr>
<td>Nvidia TITAN Xp</td>
<td>Intel(R) Core i7(R) CPU @ 3.3GHz</td>
<td>~8.28 FPS</td>
<td>~27 FPS</td>
</tr>
<tr>
<td>Nvidia RTX 2060</td>
<td>AMD Ryzen 7 1700 @ 3.2GHz</td>
<td>~6.121 FPS</td>
<td>~18.27 FPS</td>
</tr>
<tr>
<td>Nvidia GTX 960</td>
<td>Intel(R) Xeon(R) CPU E5-2630v3 @ 2.40GHz</td>
<td>~3.73 FPS</td>
<td>~12 FPS</td>
</tr>
</tbody>
</table>

Fig. 7. Precision and Recall for all OKS thresholds. AreaRng is related to the area of the objects annotated with size bigger than $32^2$ pixels.

In addition to the 2D capture, tests involving stereo cameras were also conducted. We used the ZED camera to map three-dimensional coordinates of the world from two-dimensional coordinates of images. The skeletal position is initially processed in the same way, but, in a later step, it is transformed into 3D space using intrinsic camera data and depth information. As in the previous process, the capture and processing modules are independent, with Holojam also being used to send information.

Some applications were developed as a proof of concept using the Unity 3D engine as well as WebGL. Regarding applications in Unity3D, the data is sent via the network by the Holojam Server, using multicast. This application was developed for the use of multiple devices, but it is also possible to use unicast connections. Regarding the 3D scene in the Unity engine, “Holojam Unity” objects implement support for multiplayer, which includes communication with the Holojam Server, components with position recognition, and Avatar presence. These objects have a script component called “Holojam Network”, which contains some fields indicating how to communicate with the server.

The “Multicast Address” field indicates the IP address used to receive data from the server. This field is not relevant if there is already a unicast connection between this client and the server since, in this case, the client will receive the data directly from the server. The Server Address field can be used to indicate the IP of the server directly. This address is used to send data to the server, but if the indicated address is not
local (not starting with 192), Holojam will ping this address, which creates a connection between the server and this client. All data that is sent and received uses an update data structure. When Holojam sends an object, it uses the label “SendData” for such an update. For the virtual characters, they are created from the tracked data of the real person using inverse kinematics. A manager implemented in the application reconstructs the entire body of the Actor from pairable elements corresponding to the connections between hands, elbows, shoulders, neck, head, and so on. The reconstructed data are used as targets for a 3D model in the scene. Figure 8 shows the Unity3D application.

The OpenPose also has a plugin for Unity 3D, although it is limited to run the inference locally and do not achieve the necessary performance. Their plugin often encounters problems when considering low cost GPUs, where it consumes a lot of memory and frequently is necessary to run in CPU mode. Also, unlike our application, their plugin does not deals with possibility to create shared virtual experiences where the physical presence of users in the same environment is not necessary. In our proposal, we can have a server dedicated to inference, without the need to make intensive use of the local machine. The entire project was developed considering network communication using the low latency model of Holojam. All modules in the figure were developed to be independent.

Similarly, a web application with a unicast connection was developed, as we can see in Figure 9. The entire application was designed in javascript using WebGL. A script called “Manager” was implemented, responsible for receiving the data sent by the server, considering each object that identifies each person tracked and the position of their skeleton. The Manager instantiates objects called characters for the identification of actors, while still being in charge of updating their positions to each frame. It also manages the removal of actors from the scene, if they are no longer in it, or if the track has been lost. Subsequently, the characters are rendered as a ragdoll using the position of each part of their skeleton directly, not using inverse kinematics, in this case.

8. Conclusion

In this work we proposed a novel deep neural network with streamlined architecture and tensor decomposition for pose estimation with improved processing time, named TensorPose. We adopted it on a real-time motion capture multi-user interactive application. Considering our results, we show an efficient optimization in the CNN model, since we achieved performance above 30 FPS, where in major cases represents 3x the performance of the original work. The accuracy of the detection of keypoints shows to be slightly smaller. However, this fact does not represent great problems in our applications, since our primary objective is the performance gain, when considering FPS.

As we showed in Figure 3, some failure cases were detected, but do not represent major problems in our applications. Also, we present a statistic to each kind of error detected. As a proof of concept, we present two main applications using Unity3D and WebGL, integrated with Holojam platform. We show that it is possible to create shared virtual experiences using motion capture with the TensorPose. As future work, we expect to adapt our CNN architecture to be used in low power devices.

To do so, we have to still continue to explore computational cost reduction. For this, we intend to explore strategies to create a complete form of the factored network, trying to solve the stability error, where all the layers will be in the factored form, using different types of initialization of the weights.

Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research. The authors also thanks CAPES and CNPq for the partial support of this research.

References

Fig. 8. Unity3D application with TensorPose. One of our use cases is the development of applications that make use of shared virtual environments.
Fig. 9. TensorPose web test.


