Gesture recognition using convolutional neural networks on skeletal data

A.Thomas[a], G.Devineau[b]

a. MINES ParisTech, alexandre.thomas@mines-paristech.fr
a. MINES ParisTech CAOR, guillaume.devineau@mines-paristech.fr

Key words : hand gesture recognition ; convolutional networks ; skeletal data ; time sequences ;

Abstract : In this paper, we study different options to improve the performance of a deep learning model for 3D hand gesture recognition.

We use a convolutional network to process the temporal sequences of skeletal joints positions, without using depth images. We compare a classical convolution network architecture with Temporal Convolution Network (TCN) architecture. We compare different activation functions. We also study the impact of hyperparameters such as the batch size, the learning rate, the dropout, as well as the impact of changing the size and the depth of the network, or processing the channels separately.

Experimental results on the DHG dataset from the SHREC17 Shape Retrieval Contest show that it is possible to achieve high accuracy with a relatively small and shallow convolution network (91.5% in the 14 gesture classes case, 84.0% in the 28 gesture classes case).

1 Introduction

1.1 What kind of gesture recognition ?

Gesture recognition is a way for humans to interact with machines without physical contact or voice command. Be it for device controlling without a remote (e.g. during driving or a medical operation) or for intelligent systems to understand humans intentions (e.g. pedestrians for autonomous driving [5] or for robot-human collaboration in a factory [2]), reliable and efficient algorithms for gesture recognition have an utility.

There are multiple ways to do gesture recognition. One can simply use 2D data from a standard camera and extract features from the image. One can also use 3D-model based algorithms, which require more resources but have more potential. 3D data can be acquired with sensors (e.g. wired gloves track the motion of the hand, its position and the bending of the fingers), with depth-aware cameras or with stereo cameras.

In order to deal with less parameters while keeping the essential information, skeletal representations of the body or the hand can be computed from 3D models. Even real-time pose estimation to detect keypoints of a body from images is possible (using special cameras such as Kinect [18], or even with regular cameras [19]). For these reasons, our work is focused on gesture recognition using only 3D skeletal data sequences.

1.2 Existing approaches, ours

Multiple approaches currently exist. For example, non-deep-learning methods with hand crafted features [3]. However we choose to focus on deep learning methods, as the field has evolved a lot recently and shows promising results on various tasks, including gesture recognition with skeletal data [14].

Recurrent Neural Networks (RNNs) are usually the preferred approach to temporal sequences processing (as suggested by the title of the sequence modeling chapter in the reference textbook on deep learning : "Sequence Modeling : Recurrent and Recursive Nets" [8]). However, other architectures such as Convolutional Neural Networks (CNNs) have also been successfully used for gesture recognition [6]

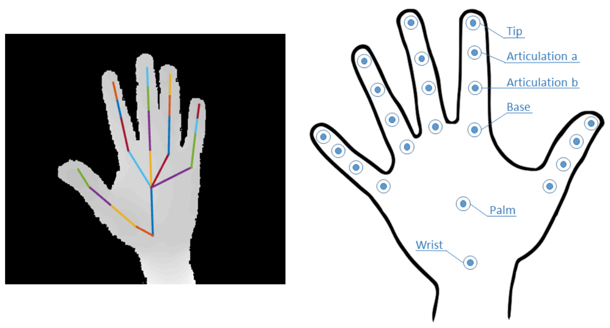


Figure 1 – *The full skeleton returned by an Intel Real Sense Depth camera*

or sequence modeling [1]. As CNNs have potential advantages over RNNs, such as being more easily optimized in parallel or less sensitive to the first examples seen, we choose to explore this method and focus on convolutional networks.

As a starting point, we use the model proposed by Devineau et al [6], for gesture recognition on the DHG skeletal dataset [3]. This network had state of the art results at the time (although now outperformed, cf 3.6), but had a relatively high total parameters count.

Our aim is to improve the model architecture by improving its accuracy or reducing its total parameters count while keeping high accuracy. We explore and compare multiple options :

— adding a preprocessing module (using a linear layer) on the temporal sequences before applying the convolutions, hoping to extract some meaning out of the sequences

— tweaking the architecture of the network (number of layers, number of separate channels, size of the layers, size of the fully-connected layers at the end, convolutions type)

— comparing different activation functions

We also explore a model based on Temporal Convolution Networks.

2 Work

In order to try new approaches, we end up with several possibilities at each layer, be it for the type of module used or the hyperparameters we choose. Let's detail, layer by layer, our options.

2.1 Network architecture

2.1.1 Input data

When working with 3D skeletal data, we are working with the spatial coordinates of m skeletal joints $(j_i)_{i\in[1,m]}$. For each joint j_i, we have 3 temporal sequences of coordinates

$$(x^{(i)}(t), y^{(i)}(t), z^{(i)}(t))_{t\in\mathbb{R}}$$

As such, we define a 3D skeletal data sequence as a vector

$$s = (s_1 s_2 \dots s_n)^T$$

where $n = 3m$ and where the variables s_i are temporal sequences corresponding to the x, y or z coordinate of a joint. In a skeletal representation of a 3D subject, the joints j_i represent physical points of the subject body, such as articulations. Figure 1 gives an illustration of a 3D skeleton of the hand.

Our experiments were done on the Dynamical Hand Gesture 14/28 (DHG) dataset [3]. The dataset consists of 2800 labeled sequences, each corresponding to a hand gesture. The SHREC17 Challenge [4] introduced an evaluation protocol by separating the 2800 sequences dataset into a train set (70%) and a test set (30%). A total of 14 gestures are possible (*Grab (G), Tap (T), Expand (E), Pinch (P), Rotation clockwise (RC), Rotation counter-clockwise (RCC), Swipe right (SR), Swipe left (SL), Swipe up (SU),*

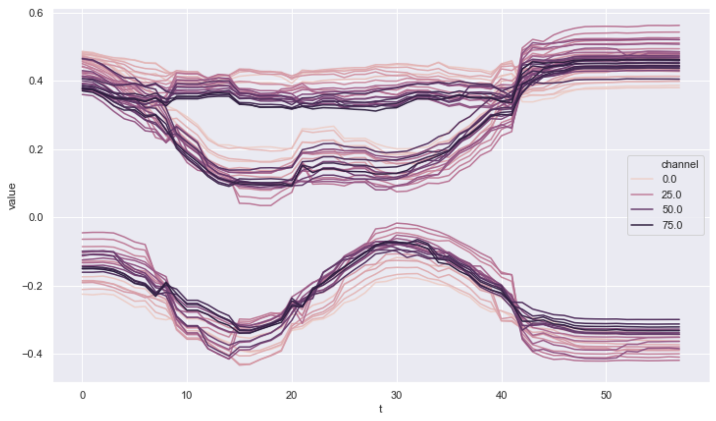


Figure 2 – *Evolution of the data sequences s_i for a Swipe X gesture, before resizing*

Swipe down (SD), Swipe X (SX), Swipe + (S+), Swipe V (SV), Shake (Sh)) and each gesture can be performed either using one finger or the whole hand. This makes a total of 14 or 28 label classes, depending on whether or not we consider the the 2 types of gestures as different label classes. The hand skeleton contains $m = 22$ joints, so each gesture consists of $n = 66$ temporal sequences.

The dataset sequences have different lengths, ranging mostly between 20 and 120 time steps (at 95%). As our model requires fixed-sized inputs, we choose to resize all the sequences to a fixed length L. Different values of L have been tried, in practice we chose $L = 100$, which gave good results. In the end, our input data consists, for each test/train example, of n data sequences $(s_1, s_2, \dots s_n)$ with

$$s_i = (s_i(t_0), s_i(t_1), \dots s_i(t_{L-1})), \forall i \in [1, n]$$

Figure 2 gives an example of such sequences.

2.1.2 Preprocessing module

The preprocessing module aimed at facilitating the work of the convolutional layers (described in 2.1.3), maybe by extracting some meaning out of the input data. The raw data distribution already had coherent, nearly normalized values (the data values range mostly from -0,6 to +0,8, with a mean of 0,18), so we did not feel the need to add a normalization layer during preprocessing.

Let $x^{(i)}$ be a sequence example (i.e. an input tensor of shape $(L = 100, C = 66)$ for the DHG dataset), and h_p the output of the preprocessing module. We tried the following options :

A linear module applying spatial combinations The idea here is to obtain linear combinations of the sequences $(s_i(t_k))_{k \in [0, L-1]}$. We hope that these linear combinations (allowing not only absolute joints coordinates, but also relative distances between the joints for instance, that could have higher semantic value) can be more easily exploited by the next layers.

This layer is simply a linear transformation

$$h_p = Ax^{(i)} + b$$

and we can choose the number of output channels $C_{p,out}$, so that output of the preprocessing layer h_p has $(L, C_{p,out})$ shape.

No preprocessing module The input sequences were directly processed by the convolutional module.

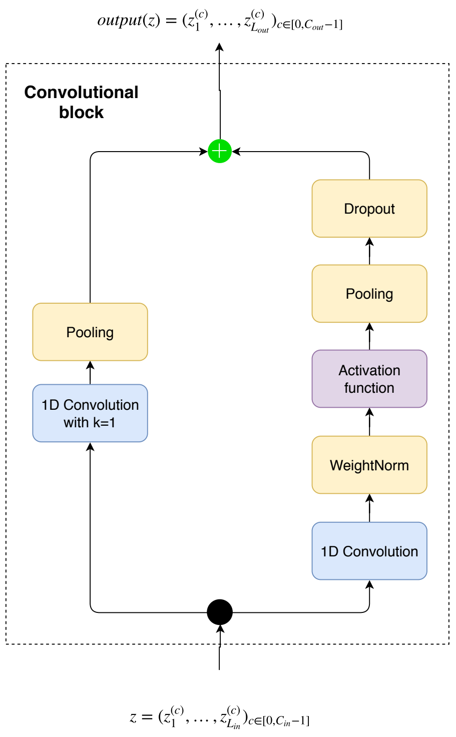

$output(z) = (z_1^{(c)}, \dots, z_{L_{out}}^{(c)})_{c \in [0, C_{out}-1]}$
Convolutional block
Dropout
Pooling
Activation function
WeightNorm
1D Convolution
Pooling
1D Convolution with k=1
$z = (z_1^{(c)}, \dots, z_{L_{in}}^{(c)})_{c \in [0, C_{in}-1]}$

FIGURE 3 – *Architecture of a regular convolution block. Kernel size can be $k = 3$ or $k = 7$ depending on the branch, and the* groups *parameter allows the 1D Conv block to process inputs independantly*

2.1.3 Convolutional module

The convolutional module does most of the work by extracting important features out of the sequences, before the final classification module. We explore here 2 options :

A regular 1D Convolutional network based on the multi-channel convolutional neural network proposed by Devineau et al. [6].

The initial network (explained with more details in [6]) was based on multiple parallel and independant modules : for each channel $i \in [1, n]$, the sequence s_i would be processed separately by a channel-specific module before being concatenated to the other channel-specific outputs and processed by the classification module. Each channel-specific module consisted of 3 parallel branches :

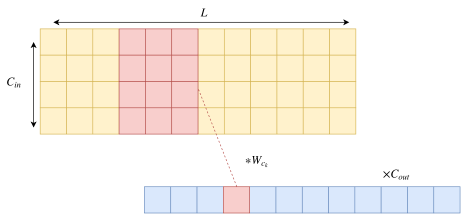
— A high-resolution convolutional branch, consisting of several 1D convolutions with pooling, with a "high" kernel size ($k = 7$ in the paper).
— A low-resolution convolutional branch, identical to the high resolution except for the kernel size ($k = 3$ in the paper).
— A residual branch that is almost an identity function with some pooling. It has been found that residual branches can help optimizing deep networks and improving accuracy [10], and it seemed to have a small but favorable impact on our initial network [6], especially in the 28-gestures case.

In our regular convolutional module, we keep these ideas (high and low-resolution branches, residual branches) but we want to study a more flexible architecture.

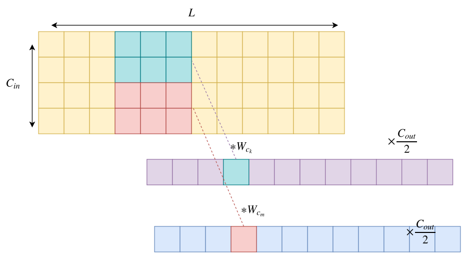
Our convolutional module is made of 2 parallel branches, each one working at a different resolution (kernel size $k = 3$ for the small branch, $k = 7$ for the high branch). Their outputs are finally concatenated into a single vector, for the classification module.

Both branches are made of multiple convolutional blocks, one after the other. The architecture of a convolutional block is illustrated in figure 3.

Each block is made of :



(a) *A regular 1D convolution layer*



(b) *A regular 1D convolution layer, with groups = 2 (instead of 1, by default). Half of the inputs are convolved to produce half of the outputs, independantly*

FIGURE 4 – *Illustration of regular 1D convolutions*

1. *A classical 1D convolution transformation*

   Let C_{in} be the number of input channels, C_{out} the number of output channels, $(z^{(c)})_{c\in[0,C_{in}-1]}$ the input sequences. The outputs of a 1D convolution layer are calculated as :

   $$(output(z))^{(c_o)} = \sum_{c_i=0}^{C_{in}-1} W_{c_o}^{(c_i)} * z^{(c_i)} + b_{c_o}$$

   Where $c_o \in [0, C_{out}-1]$ is the index of the output sequence, W_{c_o} the convolution filter of output channel c_o and b_{c_o} the bias. Figure 4a illustrates this.

   Kernel size is $k = 3$ or $k = 7$ depending on the resolution, and zero-padding is applied on both ends of the input to obtain same-length sequences. For each convolutional block (which can be seen as a layer), the number of outputs C_{out} can be chosen as a hyperparameter.

   The other major hyperparameter here is the *groups* parameter, which controls the connection between inputs and outputs, and allows to treat channels independantly. For instance, having $groups = 2$ is equivalent to having two 1D convolution layers side by side, each one seeing half of the inputs and producing half of the outputs. Having $groups = C_{in}$ is equivalent to having C_{in} separate convolution layers, each one seeing only one input channel, just like our initial network architecture. Figure 4b illustrates this mechanism.

   Having independant convolution weights for each input channel allows the initial network to perform well, however it may require a high number of parameters. It is likely that some input channels share similar information and therefore we could possibly share convolution weights between the channels while keeping comparable accuracy. This would greatly reduce the total number of model parameters.

   Finally, we add a weight normalization layer to accelerate training [17], an activation function for non-linearity, a pooling layer and dropout regularization.

2. *A residual connection*

   As stated earlier, we add a residual connection to the output of the convolution as it tends to improve accuracy and facilitate the training of deep networks. However, after the convolution transformation, the shape of tensor $output(z)$ is usually modified, both regarding the number of channels (if we choose $C_{out} \neq C_{in}$) and the length of the sequence (because of pooling).

   In order to still be able to add z and $output(z)$, we use a 1D convolution with kernel size $k = 1$ to change the number of channels without transforming the sequences in the temporal space, and we use average pooling to change the length of the sequences accordingly.

A Temporal Convolutional network (*TCN*) based on a slightly different architecture proposed by [1] for sequence modeling tasks (given a sequence of elements, the goal is to predict the next element of the sequence).

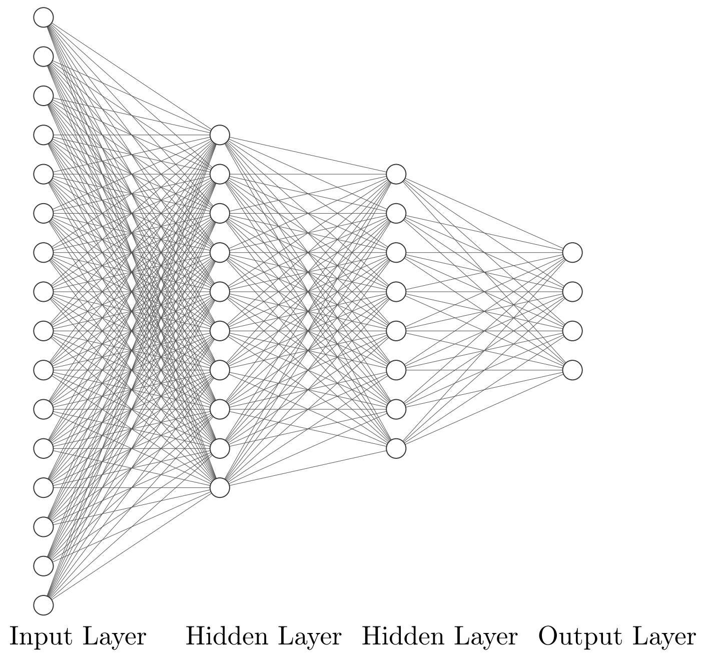


Figure 5 – Schematic illustration of the MLP used for classification. There are $L_{out} \times C_{out} \times 2$ inputs and as many outputs as the number of classes (14 or 28 for the DHG dataset). There can be one or more hidden layer, each of size $n_{hidden}^{(i)}$.

Both architectures (our classical 1D convolution and TCN) use convolutional layers to process the sequence and produce multiple output channels. Our classical architecture detailed above takes a number of fixed size sequences as input and outputs several sequences that are usually shorter ($L_{out} < L_{in}$), the goal being to synthetize information for the classification layer. The full output sequences are then processed by the classification layer. The fixed-size input sequences can be either the full gestures (offline gesture recognition), or the last timesteps recorded up to time T (online, real-time gesture recognition).

The TCN architecture is made to work on online data. The inputs are still fixed size sequences and the network outputs multiple sequences $(y^{(c)})_{c \in [0, C_{out}-1]}$ of same size ($L_{out} = L_{in} = L$). However only the last timestep $(y^{(c)}(t = L))_{c \in [0, C_{out}-1]}$ is processed by the classification layer and used for training and/or evaluating. The network uses dilated convolutions so that even a single output sequence element is represents widespread input data, and therefore keeps a long memory (more details in [1]).

The results looked promising on various sequence modeling tasks, achieving or surpassing recurrent networks. Although our goal is to classify temporal sequences and not predict the next element, the task still requires the network to gain an understanding of the input sequences and have a memory of the past time steps. Furthermore, even though we did not have the time to really explore real-time gesture classification, we initially wanted to and this kind of architecture seemed interesting.

2.1.4 Classification module

Finally, we use a multi layer perceptron (MLP) for classification. The outputs of low and high resolution convolution branches are concatenated, and flattened into a 1D tensor, of size $L_{out} \times C_{out} \times 2$. Figure 5 illustrates the architecture. The choice of the number of hidden layers (one or more) and their size is left as an hyperparameter.

2.2 Hyperparameters

For the cost function, we choose the commonly used negative log likelihood loss. For the optimization algorithm, we use the Adam optimizer, a stochastic gradient descent algorithm with adaptative learning rate [13] known to perform well in most situations [16]. Weight initialization can help a lot to converge, so we use Xavier initialization [7] instead of random initilization.

In addition to the hyperparameters detailed in previous sections, that will define our model architecture, we also have to choose a few other classical hyperparameters in deep learning. Here they are, with the set of values we would like to try.

— *Number of epochs*
  In order to prevent overfitting on the training set, it is important to stop the training once the validation score stops improving. Therefore, we use early stopping to automatically stop training if validation loss did not improve during the last 50 steps by at least 0,01%.
— *Batch size*
  32, 64, ..., 512 are frequently used batch sizes. Higher batch sizes are quicker to process during training on GPU, however it has been found that it can lead to poorer generalization [12]. In stochastic gradient descent algorithms, the gradient is averaged over a batch and larger batches seem to converge more sharply to local optimas, in comparison to smaller batches which are noisier. Therefore, we try a range of values : $batch\_size \in [8, 16, 32, 64, 128, 256, 512]$
— *Learning rate*
  By default it is suggested to use 0,001 with the Adam optimizer, but we found that decreasing the initial learning rate lead to more stable training. Therefore, we try a few different values.
— *Activation function*
  The ReLU function (Rectified Linear Unit) $ReLU(x) = max(0, x)$ is the standard activation function. However we try two other promising options :
  1. The Parametric Rectified Linear Unit (PReLU), a variant of the ReLU function : $PReLU(x) = max(0, x) + \lambda min(0, x)$, where λ is a parameter learned during training. This function, introduced by [9], aims to improve ReLU by avoiding zero gradients, while adapting the negative slope for every layer.
  2. The Swish function : $Swish(x) = x \cdot sigmoid(\beta x)$, where β is a constant or trainable parameter. This function, introduced by [15], seemed to work better than ReLU on a number of datasets. In order to avoid adding a hyperparameter, we make β a trainable parameter, just like the λ parameter of PReLU.
— *Dropout*
  We use dropout as a regularizer, and we try different values for the drop rate $p \in [0, 1]$.

3 Results

3.1 Implementation

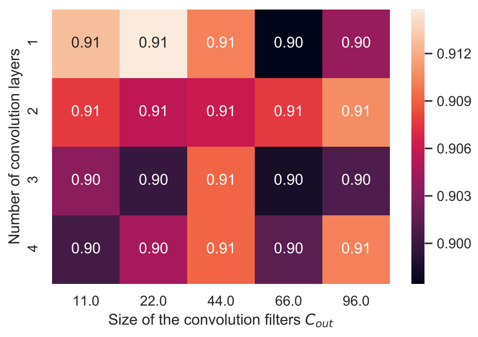
The network was implemented with PyTorch [1], and we used GPU acceleration when possible to speed up training : with a high enough batch size (≥ 64), we were able to train most models in a few minutes with a few hundreds to a thousand epochs.

The DHG dataset is already splitted in a train and a test dataset. However, as we want to evaluate multiple different architectures and hyperparameters, we must avoid using the test dataset to select these hyperparameters, otherwise we might overfit. Therefore, we use stratified 3-fold cross-validation to evaluate and compare our different model architectures. Most of the following scores are mean accuracy of the model across the 3 validation sets. The test dataset is only used for final evaluation, once the architecture and the hyperparameters are chosen.

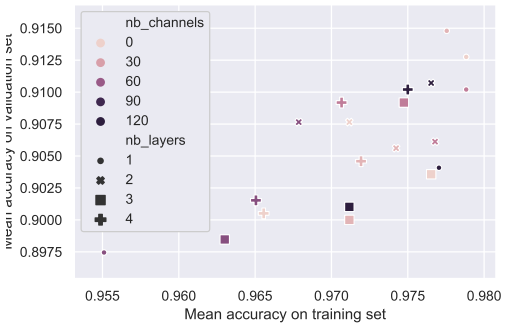
Many experiments (using both grid searches and randomized searches) were conducted to find good-performing architectures. We found that the following values worked well on the DHG dataset, so we take them as default in the next sections, and only change one or two hyperparameters at a time :

$$\begin{gathered} batch\_size = 128; learning\_rate = 10^{-4}; dropout = 0,4; \\ activation\_function = PReLU; preprocess = None; conv\_type = regular \end{gathered} \tag{1}$$

1. Code can be found at `https://github.com/alexandrethm/S3R`



(a) Mean accuracy of different convolution architectures on the validation set (using 3-fold cross-validation)



(b) Mean accuracy on the validation set (y-axis) and the training set (x-axis). Models in the bottom right part of the graph are more likely to overfit on the training set.

Figure 6 – Influence of the number and the size of convolution layers on accuracy.

with 1 convolution layer ($C_{out} = 22, groups = 1$) and two hidden fully-connected layers of size ($N_{out,1} = 1024, N_{out,2} = 128$).

3.2 Convolutional module

How to size the network ? We want to find out how deep our network should be (number of convolution layers), and how much channels should there be per layer. Randomized searches gave us some good performing architectures, but in order to have more insight on the role of these parameters, we try a more extensive grid search with $n \in \{1, 2, 3, 4\}$ convolution layers and $C_{out} \in \{11, 22, 44, 66, 96\}$ output channels per layer. Results are presented in figure 6.

It seems that small and shallow architectures perform better, meaning :

- A deep network is not necessary. When the convolutions are small ($C_{out} \in \{11, 22\}$), adding more layers generally decreases validation accuracy. When $C_{out} \geq 44$, no trend seems visible.
- A large network (with many convolution channels and parameters) is not necessary. Good performance is achieved with various C_{out}, with $C_{out} = 22$ having the best overall score.

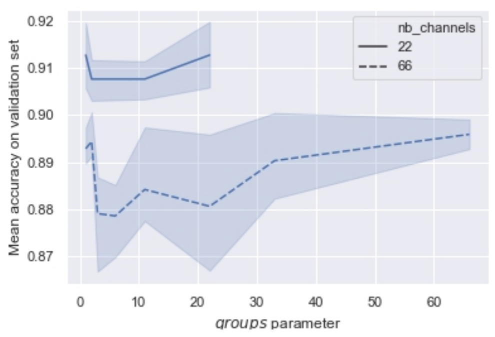
Figure 6b shows that small and shallow architectures ($C_{out} \in \{11, 22\}$ and $n = 1$) have better scores on both validation and training set. This suggests that they generalize better (higher validation accuracy) but also converge more easily (higher training accuracy). It should be noted however that we are working on a relatively small dataset, and that deeper architectures may be necessary for datasets with more examples and classes.

The *groups* parameter The initial network [6] was using 66 parallel convolution networks, each one having 3 layers of size $C_{out} = (8, 8, 4)$ and having 1 input sequence. This is the same as having the following 3 layers

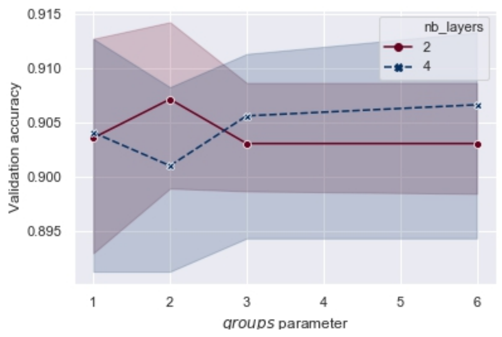
$$[(C_{out} = 66 \times 8, groups = 66), (C_{out} = 66 \times 8, groups = 66), (C_{out} = 66 \times 4, groups = 66)]$$

working on all 66 input sequences. Section 3.2 suggests that having this many channels and this many layers is not necessary, at least on the DHG dataset. We still want to see how this *groups* parameter affects our model, and if some sequences separately improves the performance.

Figure 7 presents some results. The role of processing input sequences separately rather does not seem obvious here, at least for relatively small architectures such as ours. Nonetheless, it does not seem to improve accuracy, so we choose to stick with classical convolution layers ($groups = 1$).



(a) Here we have 1-layer architectures, with either 22 or 66 output channels



(b) Here we have 2 or 4-layer architectures, with 96 output channels for each layer. The values *groups* can take are limited because it must divide 66 and 96

Figure 7 – Influence of the *groups* parameter on accuracy. Each one of the C_{out} convolutions sees only $\frac{66}{groups}$ input sequences. $groups = 1$ is a classical convolution layer, $groups = 66$ means that input sequences are processed independantly.
The average score on the 3 splits of the cross-validation is taken, and the blue bars represent the 95% confidence interval.

What about TCNs ? Here, the architecture is slightly different from our classical convolution network, as described in 2.1.3. Furthermore, we do not have 2 separate branches working with different filter sizes k, but only one branch. We explore 3 essential hyper-parameters :
— The depth d of the network, i.e. the number of successive convolution layers
— The number of convolution channels per layer C_{out}
— The filter size k of the convolutions [2]

Experimental results (figure 8) suggest that :
— TCNs require more depth than our classical CNN to be accurate (at least 5 layers). Cf figure 8a.
— Having large networks with a high $C_{out}(\geq 200)$ enables good performance (cf figure 8b). Other experiments with lower C_{out} (not shown here) did not perform that well.
— A minimum kernel size is required to perform well, presumably for the model to have a long enough memory. We see that, with $d = 5$ layers, kernel size $k = 2$ (giving a receptive field of $k^d = 2^5 = 32$ timesteps to the network) is not sufficient, whereas $k = 3$ (giving a receptive field of $k^d = 3^5 = 243$) is enough (cf figure 8c).

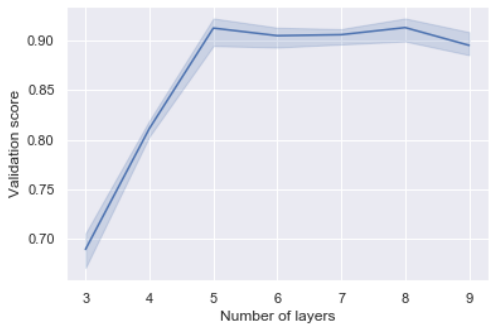
Note that, in the TCN case, the classification module's input is only the last timestep output $(y^{(c)}(t = L))_{c\in[0,C_{out}-1]}$, of size C_{out} (number of channels output of the last convolution layer), whereas the size is (C_{out}, L_{out}) for our regular CNN model. Even though we choose a larger C_{out} than in the regular CNN architecture, the output of the convolution module is much smaller. Therefore we use a MLP with only one hidden layer (of size 128) for classification.

Another remark we can make is that, although results are comparable to the regular CNN model, the TCN model has considerably more parameters and is much slower to train (20 times slower than a small and shallow CNN architecture), as most of the output is not used for classification.

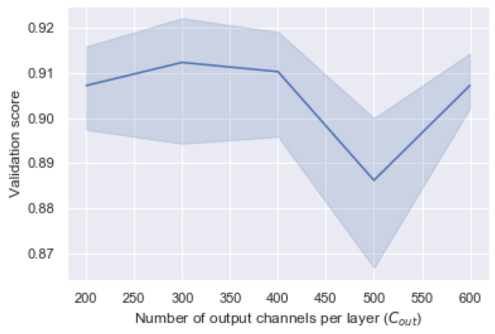
3.3 Preprocessing module

Here we try to add linear spatial combinations of the 66 input sequences before the convolution layers (as detailed in 2.1.2). We keep the reference architecture (1 convolution layer with $C_{out} = 22$ and $groups = 1$) and we try various sizes of the linear layer, with 22 to 132 channels out.

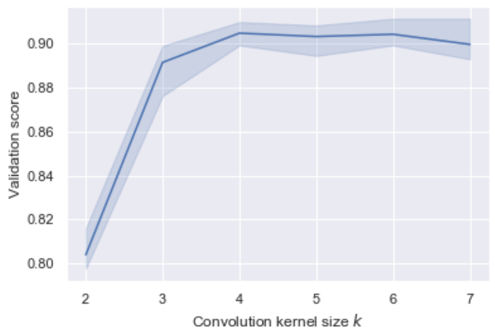
2. This is new compared to the previous paragraph with regular CNNs, where our model had 2 branches with fixed filter size $k = 3$ and $k = 7$



(a) Here the number of channels per convolution layer is set to $C_{out} = 300$ and we change the number of layers



(b) Here the number of layers is set to $d = 5$ and we change the number of output channels per layer



(c) Here we have 5 layers with $C_{out} = 300$ output channels, and we change the kernel size (previously set to 3)

Figure 8 – Experiments on TCNs architectures

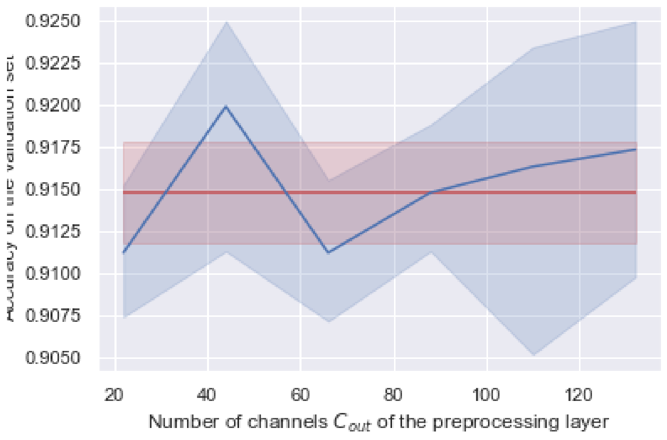


FIGURE 9 – Accuracy obtained with various sizes of the linear preprocessing module (blue curve). The red line is the reference architecture, without a preprocessing module.

The results (figure 9) seem to indicate that adding this layer has no particular effect on the accuracy.

3.4 Classification module

After the convolution network, we use a MLP (described in 2.1.4) with 1 or 2 fully connected hidden layers, of various sizes (the number of hidden units in a layer). Figure 10 shows the results of the experiments.

It seems that :

— having 2 layers gives slightly better results

— the 2 layers should not have too many parameters : the biggest architectures ($fc\_layer\_0 = 2048$ and $fc\_layer\_1 = 512$ for instance) perform slightly worse than the others.

Apart from that, there does not seem to be a clear optimum : our choice of $fc\_layer\_0 = 1024$ and $fc\_layer\_1 = 128$ seems acceptable.

3.5 Other hyperparameters

Batch size Here we try $batch\_size \in [8, 16, 32, 64, 128, 256, 512]$, and figure 11 shows the results. Although a bit chaotic, we find that :

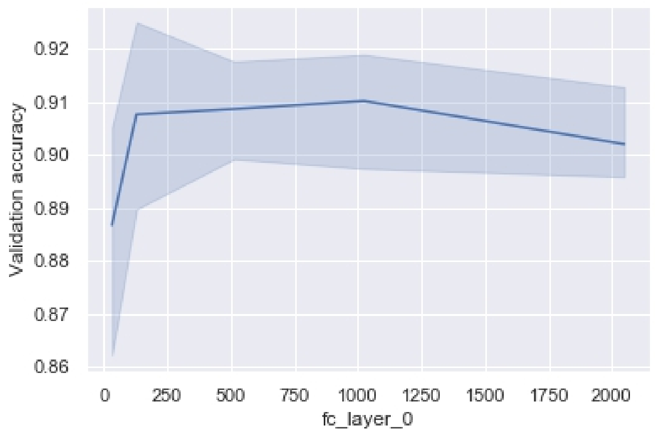
— Usual batch sizes (64 to 512) work generally fine

— Accuracy indeed starts decreasing when batch sizes get too big (after 1024)

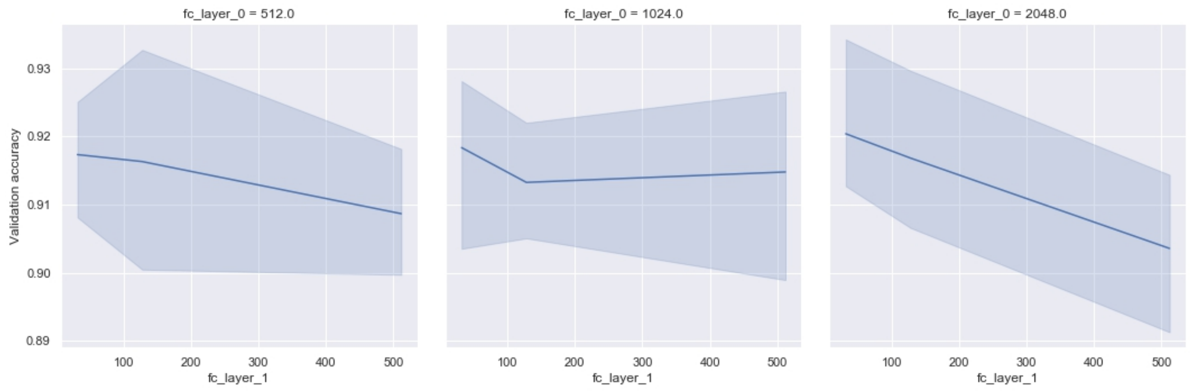
Learning rate Results are shown in figure 12. The optimal value seems to differ a bit from the default value proposed for Adam optimizer [13], being around $lr = 5 \times 10^{-5}$ in our case. Models with learning rates too low have difficulty converging, while models with learning rates too high have difficulty finding the optimum (we found that accuracy fell sharply to 0.6 when $lr \geq 5 \times 10^{-3}$ — not shown in the figure).

Activation function Trying the different activation functions detailed in 2.2, we find that the *PReLU* [9] and the *Swish* [15] activation functions perform better than the classical *ReLU* function (results in figure 13).

Dropout Here we try different values of dropout, from 0 to 0.8 (figure 14), and we find that dropout is an effective regularization technique and improves our model's accuracy (up to +2.5%).



(a) Here we have only 1 hidden layer, and we change its size $fc\_layer\_0 \in \{32, 128, 512, 1024, 2048\}$.



(b) Here we have 2 hidden layers and we try different combinations of their sizes fc_layer_0 and fc_layer_1.

Figure 10 – Experiments on the size and the depth of the MLP we use for classification

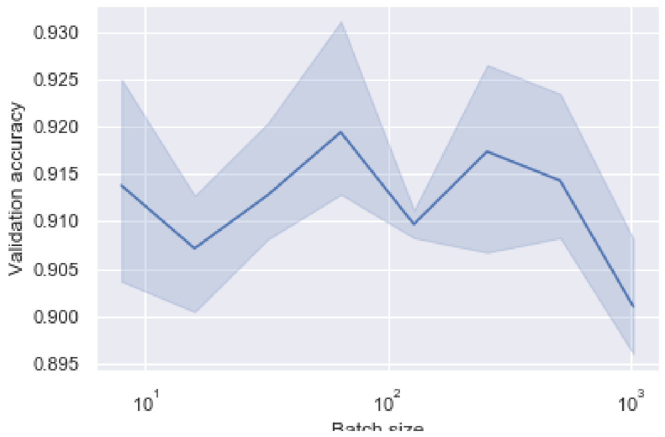


Figure 11 – Experiments on the batch size

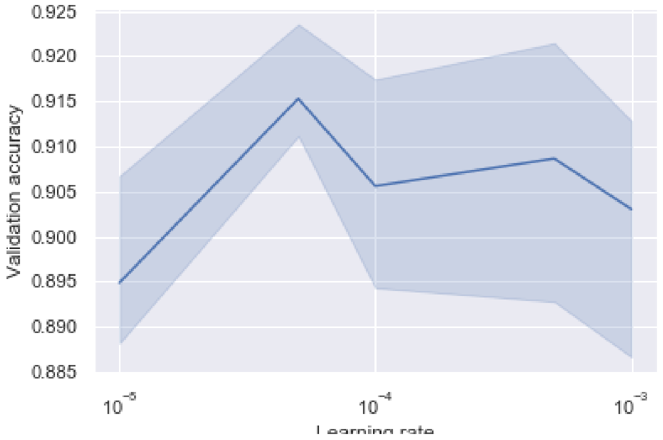


Figure 12 – Experiments on the learning rate

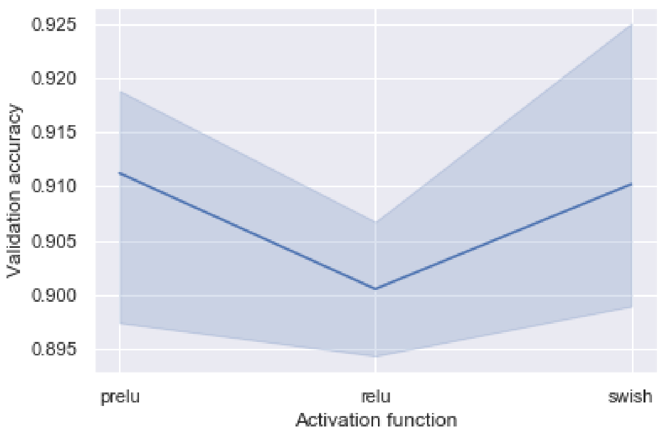


Figure 13 – Comparison of different activation functions

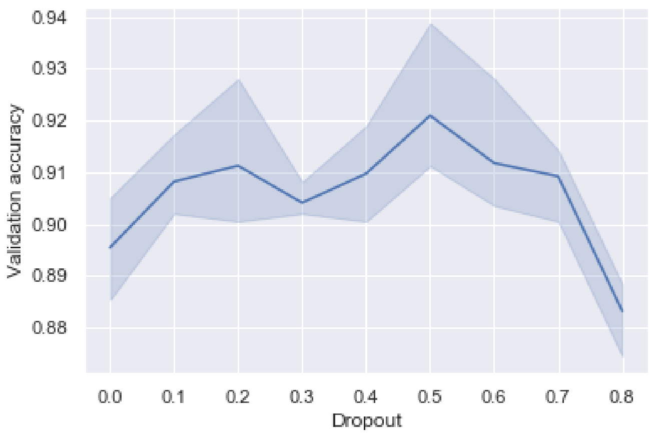


Figure 14 – Experiments on the dropout

Method	14 gestures	28 gestures
De Smedt et al. [4]	88.2	81.9
Devineau et al. [6]	91.3	84.3
STA-Res-TCN [11]	93.6	90.7
Deep-GRU [14]	**94.5**	**91.4**
Ours (regular CNN)	*91.5*	*84.0*
Ours (TCN)	*91.3*	*84.8*

Table 1 – Comparaisons of accuracy (%) on the SHREC17 DHG dataset

3.6 Results on the DHG dataset

In the end, after adjusting our initial architecture according to the above sections, we find — in the 14-gestures case and with the regular CNN architecture — a mean accuracy of

$$valid\_acc = 0.918 \pm 0.006$$

on our validation datasets, when doing 3-fold cross-validation (95% confidence interval).

Now that the architecture is chosen, we can test our models against the test dataset (both the regular CNN architecture and the TCN architecture). Results are shown in the table 1. We can see that our models are outperformed by other recent approaches :

— The *Deep GRU* model [14], which uses several stacked Gated Recurrent Units (GRU) with an attention mechanism

— The *STA-Res-TCN* model [11], which uses a spatial-temporal attention mechanism combined with Temporal Convolution Network (TCN)

Regular CNN Figure 15 shows more details on the accuracy on each class, with the confusion matrix. Our results are very similar to those of the initial model (Devineau et al [6]), while having considerably less parameters, going from 3 convolution layers with respectively 66×8, 66×8 and 66×4 channels to 1 convolution layer with 22 channels. This makes the network faster to train and to run, which can be desirable for real-time use.

TCN Even though we did not have time to fine tune the TCN architecture as well as the regular CNN one, the accuracy is very similar, and could probably surpass the regular CNN architecture. However, the model is considerably larger and slower to train.

4 Discussion

This work indicates that relatively small and shallow convolution network architectures can achieve good performance on skeletal data for gesture recognition. It would interesting however to push the study further, notably by :

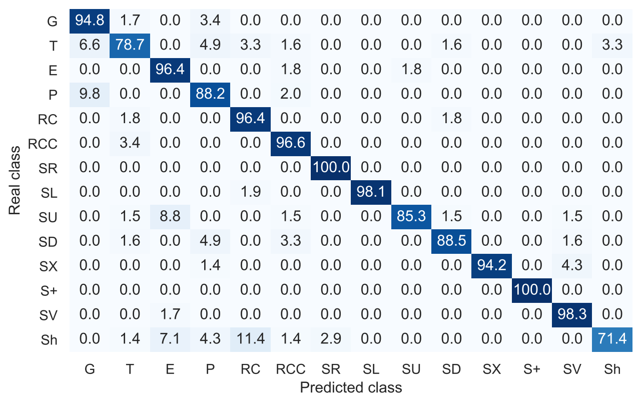
— Adding a spatial and/or temporal attention mechanism, which seems to improve performance [11] by giving more importance to the more interesting parts of the sequences. Our first experiments in that direction did not work well, however it could interesting to investigate further.

— Testing our model on other datasets, to see how well it generalizes.

— Using the model for real-time, step-by-step classification.

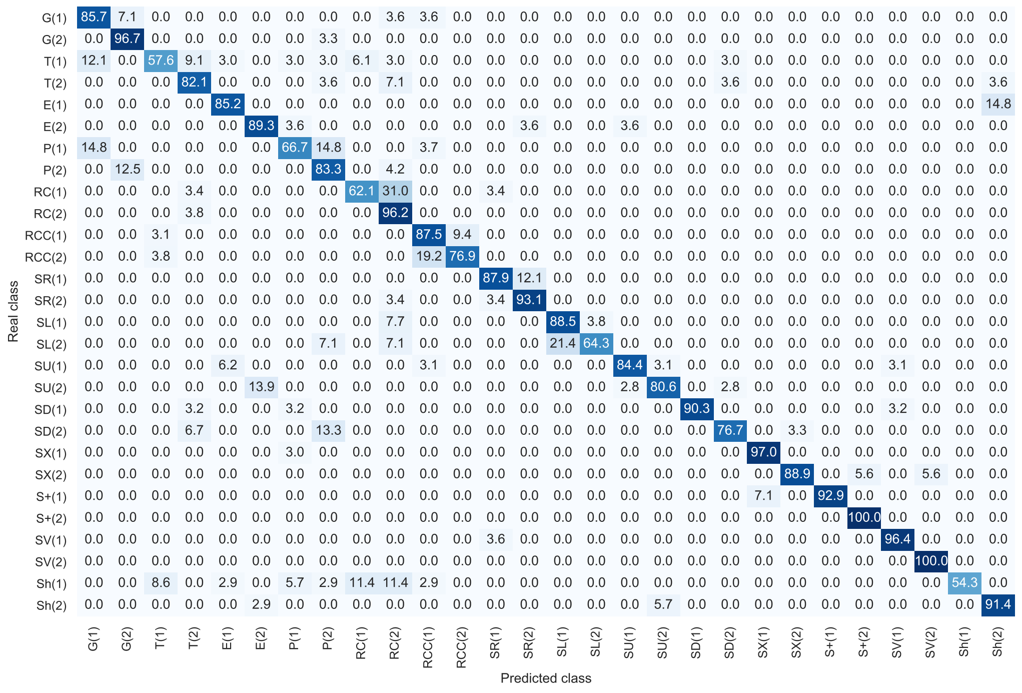
Références

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. 2018.



	G	T	E	P	RC	RCC	SR	SL	SU	SD	SX	S+	SV	Sh
G	94.8	1.7	0.0	3.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
T	6.6	78.7	0.0	4.9	3.3	1.6	0.0	0.0	0.0	1.6	0.0	0.0	0.0	3.3
E	0.0	0.0	96.4	0.0	0.0	1.8	0.0	0.0	1.8	0.0	0.0	0.0	0.0	0.0
P	9.8	0.0	0.0	88.2	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RC	0.0	1.8	0.0	0.0	96.4	0.0	0.0	0.0	0.0	1.8	0.0	0.0	0.0	0.0
RCC	0.0	3.4	0.0	0.0	0.0	96.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SR	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SL	0.0	0.0	0.0	0.0	1.9	0.0	0.0	98.1	0.0	0.0	0.0	0.0	0.0	0.0
SU	0.0	1.5	8.8	0.0	0.0	1.5	0.0	0.0	85.3	1.5	0.0	0.0	1.5	0.0
SD	0.0	1.6	0.0	4.9	0.0	3.3	0.0	0.0	0.0	88.5	0.0	0.0	1.6	0.0
SX	0.0	0.0	0.0	1.4	0.0	0.0	0.0	0.0	0.0	0.0	94.2	0.0	4.3	0.0
S+	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0
SV	0.0	0.0	1.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	98.3	0.0
Sh	0.0	1.4	7.1	4.3	11.4	1.4	2.9	0.0	0.0	0.0	0.0	0.0	0.0	71.4

(a) 14 gesture classes case



	G(1)	G(2)	T(1)	T(2)	E(1)	E(2)	P(1)	P(2)	RC(1)	RC(2)	RCC(1)	RCC(2)	SR(1)	SR(2)	SL(1)	SL(2)	SU(1)	SU(2)	SD(1)	SD(2)	SX(1)	SX(2)	S+(1)	S+(2)	SV(1)	SV(2)	Sh(1)	Sh(2)
G(1)	85.7	7.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.6	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
G(2)	0.0	96.7	0.0	0.0	0.0	0.0	0.0	3.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
T(1)	12.1	0.0	57.6	9.1	3.0	0.0	3.0	3.0	6.1	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
T(2)	0.0	0.0	0.0	82.1	0.0	0.0	0.0	3.6	0.0	7.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.6
E(1)	0.0	0.0	0.0	0.0	85.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	14.8
E(2)	0.0	0.0	0.0	0.0	0.0	89.3	3.6	0.0	0.0	0.0	0.0	0.0	0.0	3.6	0.0	0.0	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
P(1)	14.8	0.0	0.0	0.0	0.0	0.0	66.7	14.8	0.0	0.0	3.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
P(2)	0.0	12.5	0.0	0.0	0.0	0.0	0.0	83.3	0.0	4.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RC(1)	0.0	0.0	0.0	3.4	0.0	0.0	0.0	0.0	62.1	31.0	0.0	0.0	3.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RC(2)	0.0	0.0	0.0	3.8	0.0	0.0	0.0	0.0	0.0	96.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RCC(1)	0.0	0.0	3.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	87.5	9.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RCC(2)	0.0	0.0	3.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	19.2	76.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SR(1)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	87.9	12.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SR(2)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.4	0.0	0.0	3.4	93.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SL(1)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.7	0.0	0.0	0.0	0.0	88.5	3.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SL(2)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.1	0.0	7.1	0.0	0.0	0.0	0.0	21.4	64.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SU(1)	0.0	0.0	0.0	0.0	6.2	0.0	0.0	0.0	0.0	0.0	3.1	0.0	0.0	0.0	0.0	0.0	84.4	3.1	0.0	0.0	0.0	0.0	0.0	0.0	3.1	0.0	0.0	0.0
SU(2)	0.0	0.0	0.0	0.0	0.0	13.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.8	80.6	0.0	2.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SD(1)	0.0	0.0	0.0	3.2	0.0	0.0	3.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	90.3	0.0	0.0	0.0	0.0	0.0	3.2	0.0	0.0	0.0
SD(2)	0.0	0.0	0.0	6.7	0.0	0.0	0.0	13.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	76.7	0.0	3.3	0.0	0.0	0.0	0.0	0.0	0.0
SX(1)	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	97.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SX(2)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	88.9	0.0	5.6	0.0	5.6	0.0	0.0
S+(1)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.1	0.0	92.9	0.0	0.0	0.0	0.0	0.0
S+(2)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0
SV(1)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	96.4	0.0	0.0	0.0
SV(2)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0
Sh(1)	0.0	0.0	8.6	0.0	2.9	0.0	5.7	2.9	11.4	11.4	2.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	54.3	0.0
Sh(2)	0.0	0.0	0.0	0.0	0.0	2.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	91.4

(b) 28 gesture classes case.

Figure 15 – Confusion matrix of our regular CNN model on the DHG-14/28 test dataset. In the 14 gesture classes case we do not make a difference between using 1 finger and the whole hand, whereas we do in the other case.

[2] Eva Coupeté, Fabien Moutarde, Sotiris Manitsaris, and Olivier Hugues. Recognition of Technical Gestures for Human-Robot Collaboration in Factories. In *The Ninth International Conference on Advances in Computer-Human Interactions*, Venise, Italy, April 2016.

[3] Q. De Smedt, H. Wannous, and J. Vandeborre. Skeleton-based dynamic hand gesture recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1206–1214, June 2016.

[4] Quentin De Smedt, Hazem Wannous, Jean-Philippe Vandeborre, Joris Guerry, Bertrand Le Saux, and David Filliat. SHREC'17 Track : 3D Hand Gesture Recognition Using a Depth and Skeletal Dataset. In I. Pratikakis, F. Dupont, and M. Ovsjanikov, editors, *3DOR - 10th Eurographics Workshop on 3D Object Retrieval*, pages 1–6, Lyon, France, April 2017.

[5] Qiwen Deng, Renran Tian, Yaobin Chen, and Kang Li. Skeleton model based behavior recognition for pedestrians and cyclists from vehicle sce ne camera. pages 1293–1298, 06 2018.

[6] G Devineau, W Xi, F Moutarde, and J Yang. Convolutional Neural Networks for Multivariate Time Series Classification using both Inter-& Intra-Channel Parallel Convolutions. Technical report.

[7] Xavier Glorot and Y Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9 :249–256, 01 2010.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers : Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[11] Jingxuan Hou, Guijin Wang, Xinghao Chen, Jing-Hao Xue, Rui Zhu, and Huazhong Yang. *Spatial-Temporal Attention Res-TCN for Skeleton-Based Dynamic Hand Gesture Recognition : Munich, Germany, September 8-14, 2018, Proceedings, Part VI*, pages 273–286. 01 2019.

[12] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning : Generalization gap and sharp minima, 2016.

[13] Diederik P. Kingma and Jimmy Ba. Adam : A method for stochastic optimization, 2014.

[14] Mehran Maghoumi and Joseph J Laviola. DeepGRU : Deep Gesture Recognition Utility. Technical report.

[15] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[17] Tim Salimans and Diederik P. Kingma. Weight normalization : A simple reparameterization to accelerate training of deep neural networks, 2016.

[18] Jamie Shotton, Andrew Fitzgibbon, , Alex Kipman, Mark Finocchio, , and Toby Sharp. Real-time human pose recognition in parts from a single depth image. IEEE, June 2011. Best Paper Award.

[19] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*, 2017.