Classification of radar microdoppler signatures of patient activity without time consequentiality

Marco Baroni[†]

Abstract—**Remote sensing for human activities detection is a relatively new field that could open up interesting scenarios in a variety of situations. The implementation of neural networks inside this topic is kind of obligated due to the complexity and range of movements that can be done by humans in various environments. We used a dataset of human activities in hospital environment and designed two different convolutional neural networks to classify two subsets of 5 general actions each, one of stand to perform and the other performed in bed. The main target of our work is to not take advantage of the consequentiality of different actions, but to classify them just according to the micro doppler signature given by two types of radars (77 and 60 GHz) posed into two different position of the room. We reach good results having a overall balanced accuracy score of almost 0.8 for the standing dataset and of 0.66 for the bed dataset. This work could be of some use to try to investigate further the classification of micro doppler signatures a priori from the context in which they are inserted.**

Index Terms—**Supervised Learning, Neural Networks, Convolutional Neural Networks, Radar Data.**

I. Introduction

Remote sensing and, in particular, radio wave sensing is an increasing important tool to monitor and map human activities and in general to obtain useful informations from a system without the need to interact directly with it trough cameras, direct investigations etc.
In particular, in medical sciences, it was found that, through the usage of neural networks, remote sensing with radio waves can help monitoring the safety of patients without the need of direct interaction with them (Bhavanasi et. al, 2022).
A radio wave signal can be used to monitor patient activity by analyzing how the waves are reflected back to the emitter from the various persons and objects inside the room. In case of the dataset we used (Bhavanasi et. al, 2022) it had only one person in the room, either a real hospital room or a fake hospital room named Homelab, and two radar sensors that are set in two different positions. One radar operates at 77GHz and the other one at 60GHz. The two different positions are used, both in the original paper (Bhavanasi et. al, 2022) and in this work, to differentiate and enlarge the samples.
This work concentrates on the investigation for the possibility to do good classification of patient activity without the need to take time consequentiality into account, thus treating data (images) as stand-alone and not immersed into a context.

[†]UniversitÃ degli Studi di Padova,
Dipartimento di Fisica e Astronomia Galileo Galilei
email: marco.baroni.1@studenti.unipd.it

We think this could be of some importance to strengthen micro Doppler features recognition, having just one micro-series (∼3.7 seconds) giving enough information to infer the type of movement performed.
In this work we used a portion of the provided dataset, in particular all the data from the subjects from number 00 to 19, with the exception of the first set of data of subject 09 that did not provided the micro Doppler signatures.
Having the dataset, we performed some preprocessing on the images and labels before using all of it and then, after we have obtained the data in the format we wanted, we begin the hyper parameter optimization, the training and then the final testing.
The preprocessing was done reducing the number of labels and splitting our original dataset into 2 separate datasets, one filled with just bed activities and the other one filled with "standing" activities plus all the bed activities grouped together. The detailed process can be found in section three with details about the number and type of data readable in tables 3 and 4.
Regarding the hyperparameter optimization phase we chose random search as our optimization method due to the fact that, for example, supposing that we have 100 solutions for our optimization problem will mean that the top 5 solutions will lie in the 5% of the solution space. Moreover we know that the probability of at least encountering one of these top 5 solutions in "n iterations" is given by:

$$p = 1 - \left(1 - \frac{5}{100}\right)^n$$

Thus, making p = 0.95 we can see that, inverting and calling 5/100 = q:

$$\log(1-p) = n \cdot \log(q) \implies n = \frac{\log(1-p)}{\log(q)}$$

That, given p = 0.95 and q = 0.05, will give a minimum value of n of:

$$n = 60 \tag{1}$$

This is true obviously only if we choose that the best models we choose really have the probability p to show up, or at least some similar value, we tried to make this true by having inside our hyperparameters space values similar and equal to the ones that are used in the paper (Bhavanasi et. al, 2022). Our grid of hyperparameters is readable in table 1. The hyperparameter optimization phase was done using half of the dataset (ensuring that the relative abundance of samples per class is conserved) to speed up computation.

The training function we used in either the final training and the random search is a classic training function for convolutional neural networks with some added components. Indeed we used a scheduler set to reduce the learning rate if a plateau of the chosen metric (balanced accuracy) is found and we implemented an early stop if the accuracy value is not growing after some epochs.
Finally, the testing was done on a separate part of the dataset for which we set a split of 80/20 (80% of the dataset goes into training and the other part goes into testing). To evaluate the model we used 4 metrics, the balanced accuracy, the precision, the recall and the f1-score.
For completeness, we remember that:

$$\begin{cases} \text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) \\ \text{Precision} = \frac{TP}{TP+FP} \\ \text{Recall} = \frac{TP}{TP+FN} \\ \text{F1-score} = 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}} \end{cases} \tag{2}$$

Where TP is the number of true positives, FP the number of false positive, TN the number of true negatives and FN the number of false negatives.
Regarding the choice of the activation function, we followed the approach of previous works (Bhavanasi et. al, 2022) using ELU (Exponential Linear Unit) as activation function. This was done because, unlike ReLU, ELU can produce negative output values, which allows for better exploration of the parameter space during training.

$$\text{ELU}(x) = \begin{cases} x & \text{if} \quad x > 0 \\ \alpha \cdot (e^x - 1) & \text{if} \quad x \leq 0 \end{cases} \tag{3}$$

Where α is a parameter comprehended between 0 and 1 and we kept it set to 1.
As last note we performed the optimization, the training and the testing on the online platform *Kaggle* using as accelerator the *GPU T4 x2* option. We have written our models using the Pyotorch module and the chosen Pytorch initial seed was manually set to 1.
The pre-processing processes were done with Jupyter Notebook.
Everything was done in Python.

II. Related Work

This work is based on the previous work of Bhavanasi, Werthen-Brabants, Dhaene and Couckuyt, from the Department of Information Technology (INTEC) at the Ghent University-imec, published on the journal Neural Computing and Applications in 2022 in which they used two radar sensors (one at 77 GHz and the other at 60 GHz) posed in two different positions in two different kind of environments (Homelab, a fake hospital room, and real hospital room) to monitor a variety of patient's activities (see table 2 for the entire list of activities) over 24 subjects.
They focused on the use of Deep Convolutional Neural Networks for the image recognition of two different data generated from the radar, the Range-Doppler data (3 dimensional) and the Micro-Doppler data (2 dimensional) and they compared the performance of their models to other known neural networks architectures (as LSTM, CNN-LSTM, etc) and also to well known machine learning algorithms from the Sklearn Python module (i.e. Random Forests, etc).
The activities the patients performed were grouped by them into subgroups and then they were related to each other by linking different activities in base of their consequentiality.
The model they used for the Range Doppler classification is a 4 convolutional-2 linear layers model with the addition of maxpooling and dropout layer after each convolutional layer and a dropout layer after the first linear layer all ended by a Softmax activation function. Their chosen loss function was the cross entropy loss and their chosen activation function is the ELU (see Equation 3).
They trained their model for 2000 epochs and with a batch size of 256 ensuring the same relative abundance of classes for each mini batch.
They organized their work differentiating the results in three cases: the first one being using the combination of different data types in the real hospital environment, the second one using the data coming from both environments and the third one using data considering the two different radars in the real hospital environments.
Their results are expressed only in term of accuracy over different subsets of the three cases.
We tried a different approach, using the data from either the 77 GHz and the 60 GHz radars and either the Homelab or hospital environments and we grouped the data into two separated datasets: the 'stand' dataset (see table 3) and the 'bed' dataset (see table 4) and then we used two different CNNs to try to classify those datasets not balancing the relative abundance of classes in the mini batches. Moreover we added to each 'convolutional block' (sequence of convolutional layer-activation function-maxpool layer-dropout layer) a batch normalization layer to ensure regularization and to prevent overfitting.
Moreover our datasets were shuffled to ensure that there was no possibility for the netowkr to relate the different types of action on the base of their consequentiality.
We adopted this method because we wanted to see how well a CNN, given a 'random micro doppler image', could infer the activity of the patient and we divided the dataset into the two types of subsets in the idea of further work based on consecutive classification, for example, if the network classifies a standing activity as bed activity, then the classification can be moved to the other model that will sub-classify the activity more in depth if necessary. Moreover we adopted this approach to try to be as general as possible being so, we think, more near to a 'real case' application of the network.

III. Processing Pipeline

After the preprocessing phase we developed three functions for model definition, training and for the random search.
The model definition function is defined in a Python class in which the inputs are the number of channels (1 in our case since we have black-and-white images), the number of classes, the number of convolutional and linear layers, two lists or numpy arrays in which are stored the number of filters for the convolutional layers and the linear layers, the dimension of the convolutional filters, two lists or numpy arrays with the values of the dropout for each layer, the x and y dimensions of the image and the weight initialization chosen for the model.
The training function takes as inputs the number of epochs, the dataloader Pytorch objects of the train dataset and the validation dataset, the optimizer, the criterion (cross entropy loss in our case), the model and the device. This function gives as outputs three lists containing the training loss, the validation loss and the balanced accuracy score.
Finally, the random search function takes many parameters as inputs as we tried to develop it as much general as possible, the main features are that it is able to use the number of convolutional and/or linear layers, the number of neurons in the linear layers, the number of filters in the convolutional layers, the dimensions of the filters , the values of the dropouts, the batch size, the epochs, the optimizer, the weight decay of the optimizer, the learning rate and the type of weight initialization as hyperparameters, for a total of 13. Some of these hyperparameters were set to a fixed value (optimizer and kernel sizes), this was done because in previous calculation the Adam optimizer was always overperforming other optimizers in the optimization (SGD and RMSprop) and the kernel sizes were fixed to have better control on the dimension of the output of one convolutional block (see table 5).
Moreover we implemented cross validation inside our random search function and we chose to do the search using 5 folds. The output is the model that gave out the larger balanced accuracy value.

IV. Signals and Features

Before effectively using the images for the optimization and training we preprocessed the raw data from the dataset with a Python script which functioning is depicted as following:

1) Opening of the .h5 file with h5py Python module to extract the micro doppler features named for each wavelength;
2) Following (Bhavanasi et. al, 2022) we deleted the central line and the marginal parts of the images, effectively to remove non useful noise and information;
3) The format of the micro doppler data is in 1-dim numpy arrays, so, to have the full 2-dim image we collected 40 lines to reconstruct the images using in the optimization and training stacking them in column to obtain a 2-d numpy array;
4) Then we split the labels dataset into the labels and time components and we removed the date part from the time column. We then one hot encoded the data (not for our work but for general purposes) and transformed the time component from h:m:s to seconds;
5) Lastly, knowing that 40 frames corresponds to 3.7 seconds (Bhavansi et. al, 2022), we checked for each image produced in step 3 if to that image was corresponding a label from the dataset, and if so we appended the image and the label into a corresponding Pytorch tensor of appropriate dimension ($n \times 98 \times 40$ for the images and $n \times 14$ for the labels) and saved the tensor into a *.t* format for after use.

Hyperparameter Grid value bed	**Grid values stand**
n of conv layers	[1, 2, 3, 4]
n of lin layers	[0, 1, 2]
n of conv filters	[8, 16, 32, 64, 128, 256, 512]
n of lin neurons	[8, 16, 32, 64, 128, 256, 512, 1024]
kernel size	Fixed
epochs	[50 , 100 , 250 , 500]
batch size	[32, 64, 128, 256, 512]
optimizer	Adam
Weight init.	Xavier/Kaiming Uniform/Normal
dropout	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
dropout2d	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
weight decay	[10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}]
learning rate	[10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}]

TABLE 1: Table of the hyperparameters grid for the random search. The kernel size is signed as fixed because we chose a priori values for the kernel size of each layers. Also the optimizer was fixed, as we always used Adam. For the bed dataset we made some changes to the grid, remove the possibility to have just one convolutional layer (which always came up as the best with overfitting), reducing the epochs number and setting it to [500, 1000] and reducing the possibly batch sizes, due to limited amount of data, to [32,64].

After the pre-processing phase we uploaded the obtained files on Kaggle and began a secondary small pre-processing phase to create the two stand and bed subsets:

1) We firstly transformed the label dataset from one hot encoded to ”number encoded” (for example having the action ’walk to room’ encoded as 0 etc.);
2) Then we grouped all the images relative to a single action (i.e. all the *walk* images) and we concatenated them into one tensor relabeling them according to table 3;
3) We grouped all the obtained tensors into one tensors and then we divided this last object into the training and testing datasets. We did so leaving the same the distribution of the labels, so that the training and test set contains the same relative number of images per labels. The division between training and test set was set to a ratio of 80:20.

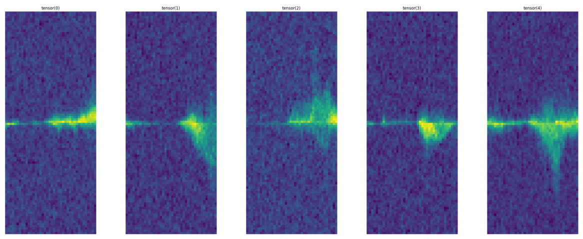

Fig. 1: Example of 1 image per class from the stand dataset. Each image has the corresponding coded label.

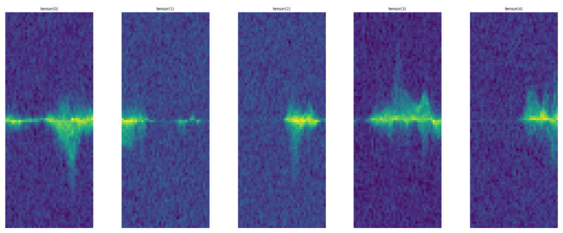

Fig. 2: Example of 1 image per class from the bed dataset. Each image has the corresponding coded label.

Label	**Code**	**n**
walk to room	0	1092
fall on floor	1	986
stand up form floor	2	974
walk to chair	3	1132
sit down on chair	4	976
stand up from chair	5	966
walk to bed	6	1252
sit down on bed	7	530
stand up from bed	8	524
get in bed	9	640
lie in bed	10	574
roll in bed	11	552
sit in bed	12	564
get out of bed	13	610

TABLE 2: Number of images per class. The column *n* gives the number of images per class.

Label	**Code**	**n**
Walk	0	3476
Fall	1	986
Stand up	2	2464
Sit down	3	1506
Bed activities	4	2940

TABLE 3: Number of images per class in the reduced dataset with 'standing' activities differently classified and bed activities all together.

V. Learning Framework

In general, as we said into section 3, we devised 2 main function other than the classical classification training function

Label	**Code**	**n**
Get in bed	0	640
Lie in bed	1	574
Roll in bed	2	552
Sit in bed	3	564
Get out of bed	4	610

TABLE 4: Number of images per class in the reduced dataset with only 'in bed' activities differently classified.

for Pytorch: the model class to obtain the model and the random search function with k-fold implemented.
The model class is designed to host at maximum 4 convolutional blocks and any number of linear blocks. The choice of maximum 4 convolutional blocks was done because, since each block has a maxpool layer, the dimension of the 4-th block output was too small to continue to append more convolutional blocks. The kernel and padding sizes were chosen to maintain the output dimension as an integer, knowing that stride and dilation were set to 1, following:

$$\begin{cases} W_{out} = 1 + W_{in} + 2p_x - k \\ H_{out} = 1 + H_{in} + 2p_y - k \end{cases} \quad (4)$$

The two dimension of the kernel size are the same as we chose to maintain the kernel squared. The chose padding mode was set to *zero padding*.
Other than the model class we designed an auxiliary function

n conv. layer	**kernel size**	**padding**	**flat dimension**
1	3	(1,2)	N·21·49
2	4	(3,3)	N·12·26
3	3	(1,0)	N·5·13
4	2	(1,1)	N·3·7

TABLE 5: Kernel, padding sizes and flattened dimension for each convolutional layer. The dilation and stride sizes are always 1 in each the x and y dimensions. The letter *N* represents the number of convolutional filters for that layer. The initial dimension for each image is 98×40 pixels.

to initialize the weights for the convolutional and linear layers as well as the batch normalization layers. The batch normalization layers were all initialized by filling the weights with ones and the bias of the convolutional and linear layers where initialized by filling them with zeros while, for the other two types of layers, we gave the choice between the following weight initializations:

- *Xavier uniform*: the weights are sampled from a uniform distribution in the interval (-a ; a), where a is given by:

$$a = \text{gain} \cdot \sqrt{\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}} \quad (5)$$

- *Xavier normal*: the weights are sampled from a normal distribution in the interval (0 ; std^2), where std is given by:

$$std = \text{gain} \cdot \sqrt{\frac{2}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}} \quad (6)$$

- *Kaiming uniform*: the weights are sampled from a uniform distribution in the interval (-a ; a), where a is given by:
$$a = \text{gain} \cdot \sqrt{\frac{3}{\text{fan}_{\text{mode}}}} \quad (7)$$
Where in our case fan_{mode} is set to preserve the magnitude of the variance of the weights in the forward pass;
- *Kaiming normal*: the weights are sampled from a normal distribution in the interval (0 ; std^2), where std is given by:
$$\text{std} = \frac{\text{gain}}{\sqrt{\text{fan}_{\text{mode}}}} \quad (8)$$
Where in our case fan_{mode} is set to preserve the magnitude of the variance of the weights in the forward pass.

For each initialization the *gain* parameter was set to 1. Fan_{in} and fan_{out} are the square root of the number of neurons (for convolutional networks, filters) in, respectively, the previous and next layer (X.Glorot Y.Bengio, 2010). Fan_{mode} in our case is equal to the previous number of neurons.

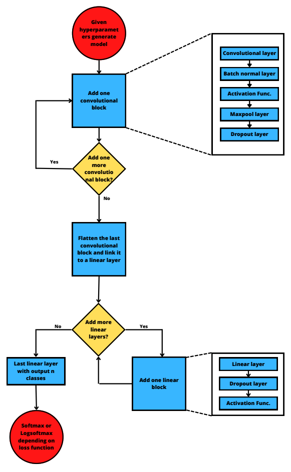
Given hyperparameters generate model
Add one convolutional block
Convolutional layer
Batch normal layer
Activation Func.
Maxpool layer
Dropout layer
Add one more convolutional block?
Yes
No
Flatten the last convolutional block and link it to a linear layer
Add more linear layers?
No
Yes
Last linear layer with output n classes
Add one linear block
Linear layer
Dropout layer
Activation Func.
Softmax or Logsoftmax depending on loss function

Fig. 3: Flow chart of the model generation class.

The training function used to train the CNN model takes the following arguments as inputs:

- *model*: the model to be trained;
- *train_ds*: the training dataset;
- *validation_ds*: the validation dataset;
- *num_epochs*: the number of training epochs;
- *patience*: the number of epochs without improvement in accuracy before early stopping;
- *weight_decay*: weight decay applied to the optimizer;
- *bs*: batch size for the data loaders;
- *device*: device (CPU or GPU) to use for training;
- *LR*: the learning rate;
- *class_weights*: the weight to be applied to the cross-entropy loss;
- *printer*: whether to or not to print the training progress.

The function first moves the model to the specified device and creates data loaders for the training and validation datasets. It then defines the criterion (cross-entropy loss with class weights) and optimizer (Adam) and a scheduler that reduces the learning rate when the validation loss plateaus.

The function then enters a loop for the number of epochs specified. In each iteration, it sets the model to train mode, and iterates over the training data, computing the loss and backpropagating the gradients. The average training loss for the epoch is stored.
Then the function sets the model to evaluation mode, and iterates over the validation data, computing the validation loss and the chosen metrics (accuracy, precision, recall, and f1-score). The scheduler steps are taken with validation loss. It also checks whether the current accuracy is the best seen so far. If it is it resets the number of epochs without improvement. If it is not, it increments the number of epochs without improvement. If the number of epochs without improvement exceeds the patience, the function returns the training loss, the validation loss and the metrics.
The output of the function are two lists with the losses and the final values of the metrics.
As last, the hyperparameters optimization is performed by randomly sampling the hyperparameters from a specified parameter space (saved as a Python dictionary) for a set number of trials. In each trial, the neural network is trained and evaluated using k-fold cross-validation, where the dataset is split into k folds, and the model is trained on k-1 folds and evaluated on the remaining fold.
The function uses the previously defined training function to train and evaluate the model employing early stopping and weight decay as regularization techniques to prevent overfitting. The training and evaluation process is repeated for each fold and the average balanced accuracy is calculated. The function keeps track of the best accuracy achieved during all trials and the corresponding set of hyperparameters.
The neural network architecture is specified in the Net class defined before. The function also sets the device used for training (CPU or GPU) and the random seed for reproducibility. The function returns the best balanced accuracy and the corresponding set of hyperparameters found during the search.

VI. Results

The final models generated from the hyperparameters optimization phase are visible in figure 4 and table 6. We obtained two model with a high number of convolutional blocks (similar to the one used in Bhavanasi et. al 2022). Being more specific, the model for the stand dataset had 3 convolutional layers and 2 linear layers, while the model for the bed dataset has 4 convolutional layers and 1 linear layer. The main differences between the two models are that for the standing model we have a normal-type of weight initialization, while for the other one we have a uniform initialization, the learning rate are different by a factor of 10 and the weight decay in the optimizer are really different, one being 1000 greater than the other (see table 6). Knowing the different properties between normal and uniform initializations, it seems fair that for the stand dataset, for which we obtained higher metrics values (see table 7 and figure 9), the choice was set onto a normal distribution, knowing that normal initialization is often used to break symmetry and prevent the model from getting stuck in a local minimum. While, for the bed dataset, a uniform initialization seems legit because, due to the lower metrics values (see table 8 and figure 10) it was probably chosen by the random search in order to encourage the model to explore different regions of the parameter space. From the loss plots (figures 3 and 4) we can see that the training went kind of good for both tasks, but with huge differences between the two: firstly, the standing dataset training completed all the steps without encountering an early stopping, while the other one stopped training after 175 epochs over one thousand, secondly, for the second dataset, the validation loss drops quicker than the other one, but reaches a higher plateau around 0.95, while for the standing dataset this plateau is kind of reached around 0.8, thirdly and lastly the training losses reaches two almost similar plateaus, but for the first one a higher epochs number could have made some difference and could have enhanced the performance.

Stand Model	
N conv layers	3
N lin layers	2
N conv neurons	128, 512, 256
N lin neurons	512, 8
Dropout 2d	0.5, 0, 0.1
Dropout	0.4
Weights init	Kaiming Normal
Optimizer	Adam
Learning rate	0.0001
Weight Decay	10^{-5}
batch size	128
N epochs	100
Bed Model	
N conv layers	4
N lin layers	1
N conv neurons	512, 64, 128, 32
N lin neurons	8
Dropout 2d	0.2, 0.9, 0, 0.4
Dropout	0.3
Weights init	Xavier Uniform
Optimizer	Adam
Learning rate	0.001
Weight Decay	0.01
batch size	32
N epochs	1000

TABLE 6: Summary table of the final models.

Regarding the results, we can see that, overall, the classification tasks went good (see tables 7 and 8 and figures 7 and 8), especially regarding the standing dataset. Looking at this one we can see that overall we have reached a balanced accuracy of almost 0.8 with a total f1 score of, again, almost 0.8.

Seeing more into deep with the single label classification

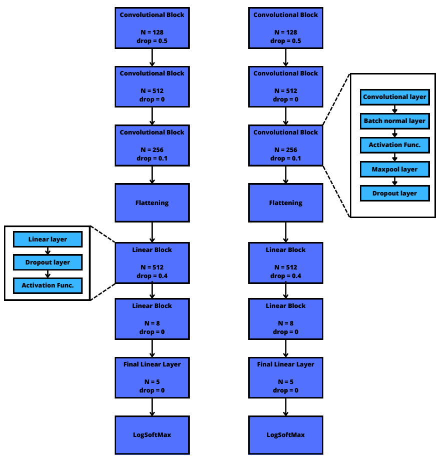
Convolutional Block N = 128 drop = 0.5
Convolutional Block N = 512 drop = 0
Convolutional Block N = 256 drop = 0.1
Flattening
Linear Block N = 512 drop = 0.4
Linear Block N = 8 drop = 0
Final Linear Layer N = 5 drop = 0
LogSoftMax
Linear layer
Dropout layer
Activation Func.
Convolutional Block N = 128 drop = 0.5
Convolutional Block N = 512 drop = 0
Convolutional Block N = 256 drop = 0.1
Flattening
Linear Block N = 512 drop = 0.4
Linear Block N = 8 drop = 0
Final Linear Layer N = 5 drop = 0
LogSoftMax
Convolutional layer
Batch normal layer
Activation Func.
Maxpool layer
Dropout layer

Fig. 4: Flow chart of the final models. The right one is the one used for the bed dataset, while the right one is the one used for the stand dataset. The highlighted pieces are the description of the blocks used, the only exception is that for the last linear layer the dropout is 0 and there is no activation function. We also remember that the activation function was the ELU and the loss function was the cross entropy loss.

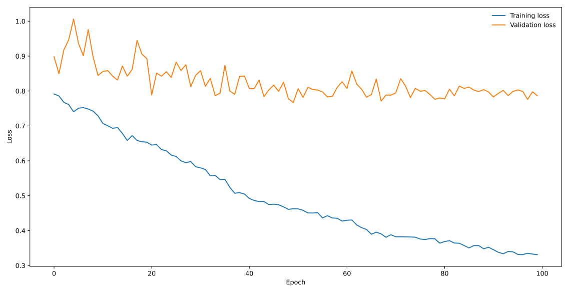

Fig. 5: Plot of the losses for the training of the ”standing” dataset.

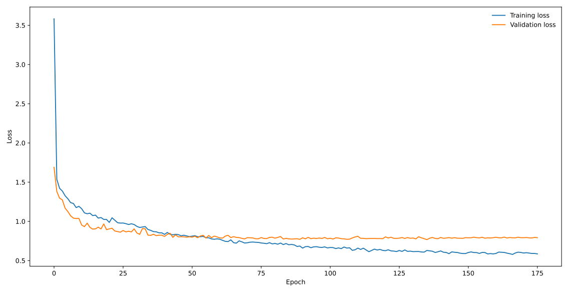

Fig. 6: Plot of the losses for the training of the bed dataset.

metrics we see that for each class we have a balanced accuracy over 0.8, with only the 'Sit' and 'Bed' classes not so well classified. This outcome could lie in the fact that the 'Sit' class is the second less represented class and, given that we gave the class weights to the cross entropy loss function and that we balanced the relative abundance in each subset we used (train, validation, test and random search), this one could have been kind of missed.
Moreover, regarding the 'Bed' class, the poor classification could lie int he fact that the we had 5 different classes classified as one, and, from figures 1 and 2, we can see some general patterns in the bed images that could make them misclassified in some standing images (i.e. the 'sit in bed' class have some general features in common with the 'stand up' class).
So, having this in mind, we can in any case say that the classification went pretty good also for this two classes for the standing dataset.
Regarding the bed dataset we see that in this case the overall classification gave a result that is more accurate than the flip of a coin, but in any case lower than some acceptable value. Going into deep for each class we especially see that regarding the 'lie in bed' and 'roll in bed' classes the precision and recall values are really low.
This last classification have for sure some problems, and maybe a search space of only 60 tries was not nearly sufficient to find a good enough model. The only class for which we can kind of say that the classification went well is the 'get in bed' class, for which we obtained a balanced accuracy and recall values above 0.8 and an f1 score of almost 0.8. The reason behind this bad outcome could also lie in the fact that the overall dataset dimension was not so high with the most represented class being the 'get in bed' class with 640 images (see table 4), so, even if this model is really one of the best models, it could not generalize well enough due to under representation of the classes.
The resulting classification are clearly visible in figures 9 and 10 in which the confusion matrices of the tasks are reported. We can see that, again, for the first task the classification went good with the notable exception of lots of misclassified samples from the 'bed' class (which we have previously discussed) and some misclassification that also happened for the 'sit down' class for which we have some incorrectly labeled especially, again, in the 'bed' class, and also for the 'stand up' class the same happened.
Regarding the other two classes we can see that the classification went good and it is notable the fact that for the 'fall' class, the most under represented, the classification was overall very nicely done.
Regarding the second task confusion matrix we can see that we have spread misclassications, almost all of them at the same order of magnitude of the true positives, again, as stated before, this could be related to the fact that the dataset was undersampled.

	B. Accuracy	**Precision**	**Recall**	**f1-score**
Dataset	0.7920	0.7961	0.7880	0.7897
Walk	0.9020	0.7871	0.9101	0.8441
Fall	0.9059	0.7932	0.8376	0.8148
Stand	0.8721	0.8682	0.7897	0.8271
Sit	0.8378	0.6928	0.7342	0.7129
Bed	0.8237	0.7330	0.7509	0.7418

TABLE 7: Summary table of the classification results for the standing dataset.

	B. Accuracy	**Precision**	**Recall**	**f1-score**
Dataset	0.6599	0.6644	0.6650	0.6631
Get in	0.8482	0.7464	0.8047	0.7744
Lie in	0.7812	0.7075	0.6522	0.6787
Roll in	0.7097	0.5728	0.5364	0.5540
Sit in	0.7490	0.6634	0.5929	0.6262
Get out	0.7823	0.6214	0.7131	0.6641

TABLE 8: Summary table of the classification results for the bed dataset.

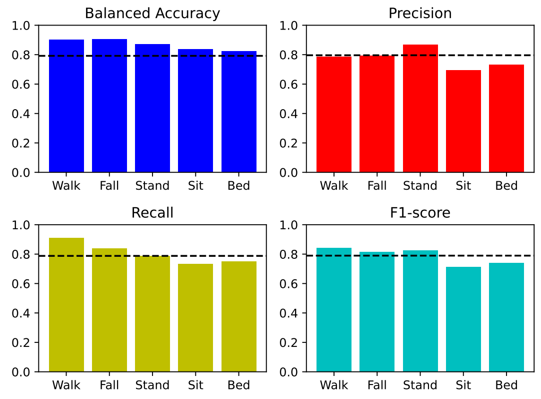

Fig. 7: Detail of the metrics used for each class for the stand dataset. The horizontal dotted line is the value for that metric parameter in the overall classification.

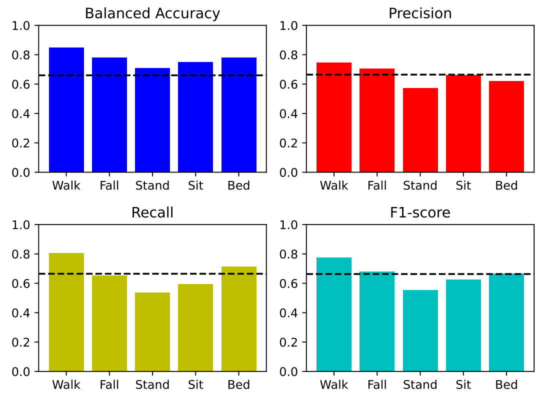

Fig. 8: Detail of the metrics used for each class for the bed dataset. The horizontal dotted line is the value for that metric parameter in the overall classification.

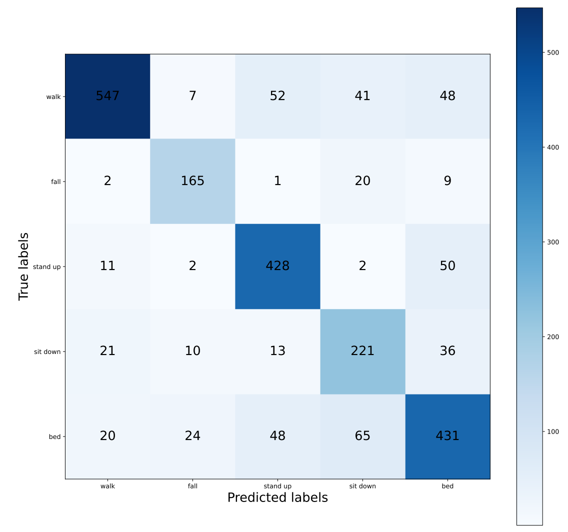

Fig. 9: Confusion matrix of the stand dataset.

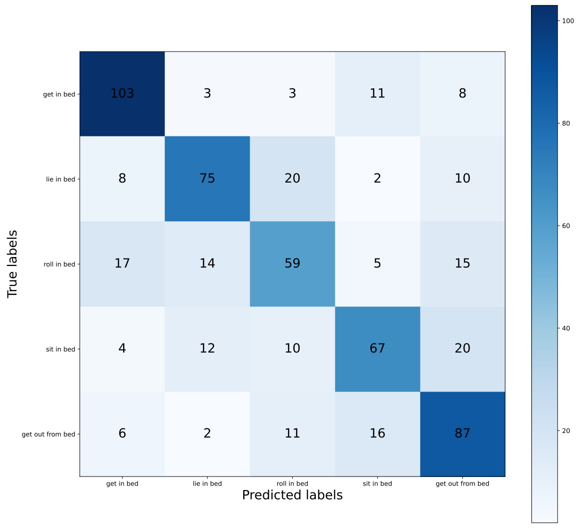

Fig. 10: Confusion matrix of the bed dataset.

VII. Concluding Remarks

In the end, in this work, we used the dataset generated by Bhavanasi et al., 2022 to perform a classification on their micro doppler maps dividing the dataset provided into two subsets and then finding a adapt model using random search hyperparameters optimization.
For the first dataset (stand) we reached good results as the overall balanced accuracy of 0.7920, while for the second dataset the results were worse with an overall accuracy of 0.6599. As stated before the poor performance on the second dataset could be related to a simple undersampling problem and this could be solved in future by implementing some data augmentation techniques (such as adding some noised images on the dataset).
In general we can see that our results are pretty in line with the ones obtained before (Bhavanasi et al., 2022) but for sure our performance could enhance using some techniques as data augmentation, or trying different optimizers or, moreover, deepening our convolutional models giving the option to have ore layers or even trying to combine different activation functions and/or weight initializations for each layer.
At the end, a future work could be to effectively combine the two networks by doing some concatenated classification (image in input into the 'stand' model and if the output is a bed activity the image is feeded to the 'bed' network) and, for sure, to try this new resulting model on new unseen data.
As appendix I would like to say that by doing this project I have learned to use deep convolutional networks in an effective manner on more complex data than the ones from the MISC datasets and, as an aspirant astrophysics researcher, this for sure will be useful for my future. A huge difficult I encountered was that by not having a personal GPU adapt to the cuda protocol it was kind of challenging to just use online resources as Kaggle to do all the computations, due to easy oberation of the given GPUs.