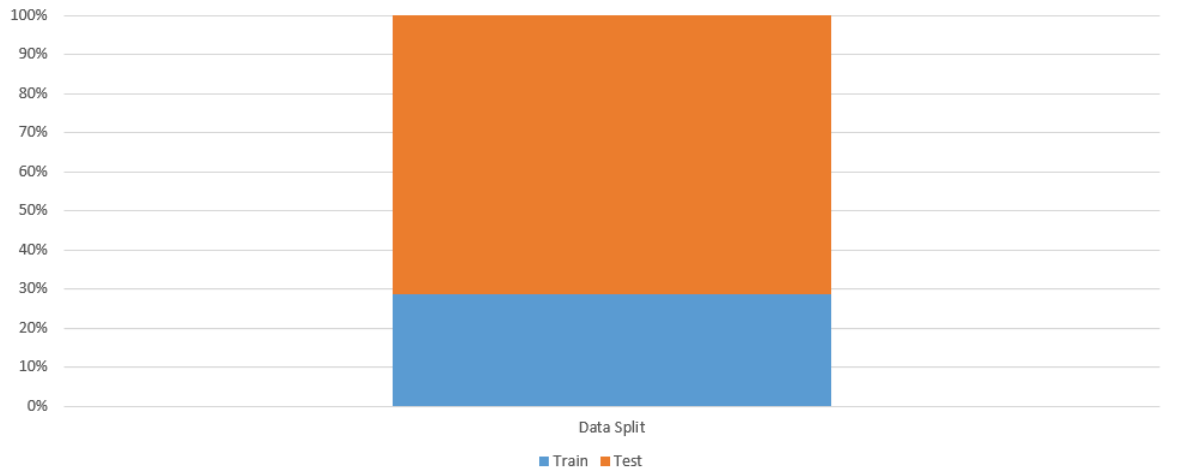
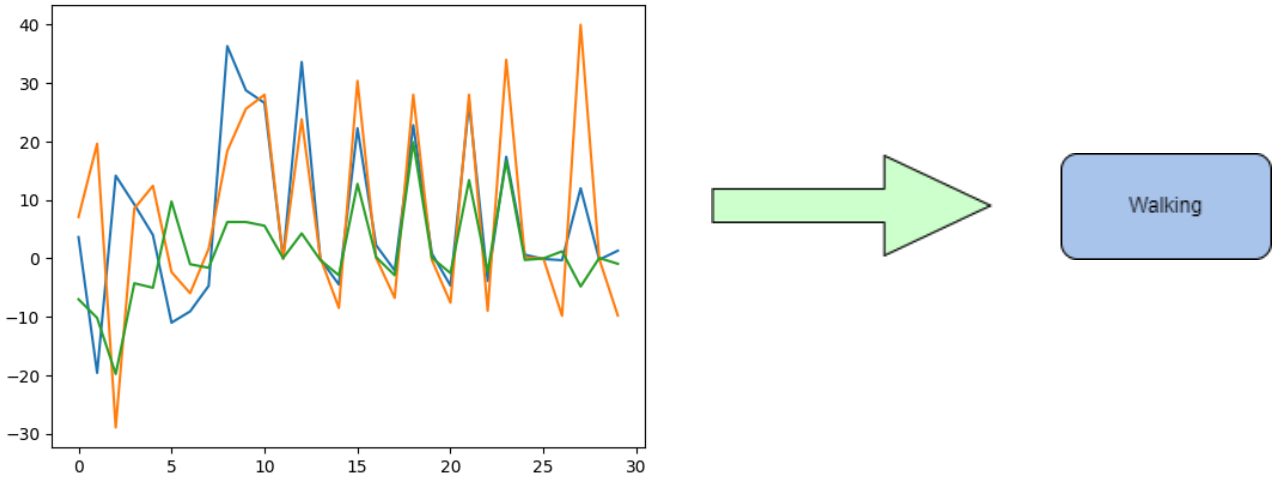
Data Split Amount: We count the data amount by the amount of csv files in the given data folder. There are a total of 124992 csv files, and 50248 of them are for the train set. Which means training (including evaluation) percentage is
40%. Here is a plot to clarify the splitting proportion:



B. The task given is a classification task, given the x,y,z acceleration sequences, we have to predict what activity the user was doing (a.i. In the past). Here is a plot to show some sample:

Given the following sequence of x(blue), y(green), z(orange), the given activity is “walking”



So given another sequence of x,y,z we have to predict the activity the user were doing

C:

1. **LSTM Auto Encoders**

it is a self-supervised learning approach that learns useful representations by predicting the future in latent space. We think this method would be good for sequence data, where the goal is to learn representations that capture underlying factors of the data.

How it works:

Encoder: First, an encoder network based on LSTM, transforms the input sequences into a latent space representation.
Decoder: Take the new space representation from the encoder and predict the real sequence (also using LSTM).
Auto Encoder: a model that contains both encoder and decoder, its objective loss is by comparing the predicted sequence from the decoder with the real sequence.

2. **Predicting the Next Sequence (Temporal Prediction)**

Given the time series sequences from sensors, one self-supervised task could involve predicting the next sequence in a time series given the previous sequences. This type of task is useful for learning features from the sequences.

How it works:
Input: Use a sequence or a series of sequences as input to the model.
Target: The subsequent sequence in the series becomes the target output.
Model Pretraining: Train a model (e.g., an RNN or LSTM) to predict the target sequence based on the input. The model learns to understand the progression and patterns in the activities.

1.

2) Model Build

a. The given data is partitioned into 40% labelled and 60% unlabelled. We Splitted the data into train, test and validation sets
If we had enough time and computing power we would have chosen k-fold-cross validation since there are low amount of the data, this method lets us see the model generalization level.
For the Loss: we chose cross entropy loss, which is a good measure for Multiclass classification task.

Set	Percentage	Count
Train	0.72	36k
validation	0.08	4k
Test	0.2	10k

c) We chose the models Naive Bayes and XGBoost. Two reasons to choose these models:

1. These models support batch-training.
2. Check the performance of a simple ML model versus more complex one.

For our experiments we tried multiple approaches of extracting features from the sequences, each time we tried more complex approach to know how much the extracted features would help the models, and we may pass these features for the neural networks as additional features. Here are the approaches:

1. Decrease the length of the sequence by picking one point (x,y,z) every N points , and then extract 6 features: mean of x,y,z (each axis separated, making it total 3 features), and std of x,y,z (3 features too).

2. Same method for decreasing the sequences, but now split the sequences to segments and get the mean and std of x,y,z. So the total number of features is (num_of_segments * 6). We think that splitting the sequence into segments will give important information to the models.

3. Now Decrease the length of the sequence using moving-avg smoothing (less noisy data), same features as approach 1. Total num of features: 6

4. moving-avg smoothing, and same feature segmenting as approach 2

Results Table:

	Naive Bayes					XGBoost				
Approach	**Train acc**	**Train loss**	**Test Acc**	**Test loss**	**Train Time**	**Train Acc**	**Train loss**	**Test Acc**	**Test loss**	**Train Time**
1	0.34	14.11	0.375	3.269	**18s**	0.8052	0.6882	0.551	3.728	**35 m**
2	0.34	14.11	0.32	14.65	**64 s**	0.766	1.098	0.4508	6.025	**4 m**
3	0.3818	3.291	0.3779	3.111	**26 s**	0.765	0.888	0.575	0.182	**4 m**
4	0.355	13.14	0.35	13.673	**67 s**	0.784	1.007	0.486	5.487	**6 m**

Notes and conclusion:

1. XGBoost is clearly overfitted in the first two approaches. We tried to add noise using different distributions, and tried to change loss functions, but the results didnt change significantly.
2. Naive Bayes classifier is more stable when using applying the feature extraction on the whole sequence and not on segments.
3. For our features naive bayes performed better (more simple and stable), but we think picking new features (like skewness would change the results), but we didnt have enough time to try all the options
4. According to the results table we assume that XGBoost would perform better when we use segmenting.

For now we can consider accuracy of 0.48 is the baseline for our neural networks models.

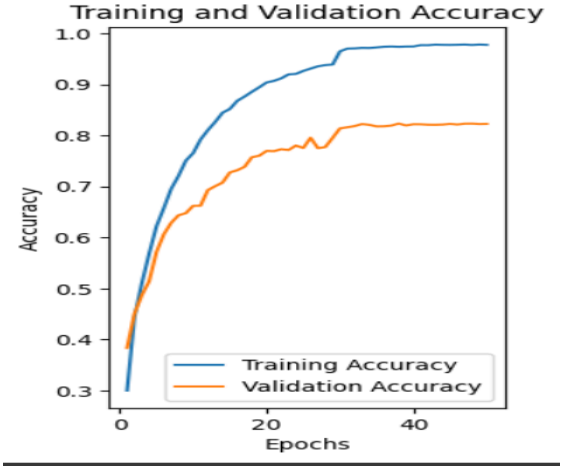
D)

1DCNN model architecture :

```
Conv1DNet(
  (conv1): Conv1d(3, 64, kernel_size=(1,), stride=(1,))
  (pool1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv1d(64, 128, kernel_size=(1,), stride=(1,))
  (pool2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv1d(128, 256, kernel_size=(1,), stride=(1,))
  (pool3): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.4, inplace=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (fc1): Linear(in_features=256, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=18, bias=True)
)
```

The model consists of three **convolutional layers** (conv1, conv2, conv3) each followed by a **max pooling layer** (pool1, pool2, pool3), which are used for feature extraction. The number of filters in the convolutional layers increases with depth, starting at 64 and doubling with each subsequent layer to 256, capturing more complex features at each level. A **dropout layer** with a probability of 0.4 is included to prevent overfitting by randomly setting a portion of the input units to 0 at each update during training. An **adaptive average pooling layer** follows, which outputs a fixed size tensor, making the network adaptable to inputs of varying sizes. Finally, two **fully connected layers** (fc1 and fc2) act as a classifier, with the last layer outputting 18 features, which likely correspond to the number of classes in the classification task. The use of 'bias=True' in the fully connected layers indicates that bias terms are added to the learning process, providing additional flexibility to the model.

After training the model for 50 epochs we got the following results on validation and training sets:



From the accuracy graph, it's evident that the model's training accuracy increases consistently and significantly over epochs, indicating that the model is effectively learning from the training data. **However**, the validation accuracy also improves but plateaus earlier and displays more variability, suggesting that the model might be overfitting to the training data and not generalizing as effectively to the unseen validation data.

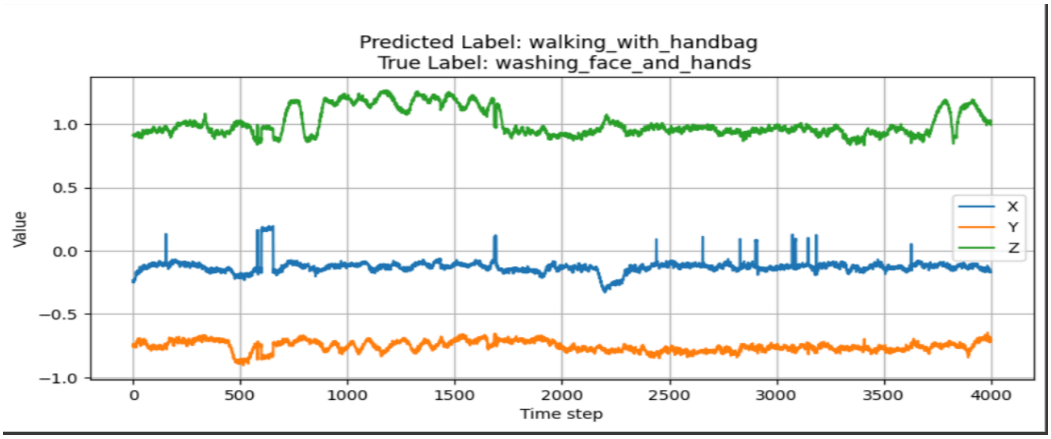
The loss graphs further support this interpretation. The training loss decreases sharply and then levels off, which is expected during the training process. The validation loss, however, decreases initially but then starts to fluctuate, indicating that the model's predictions are less consistent on the validation data. This fluctuation and the gap between training and validation accuracy suggest that the model could benefit from regularization techniques, more training data, or hyperparameter tuning to better generalize and improve validation performance.

	Train	**Validation**	**Test**
accuracy	**0.9802**	**0.7756**	**0.7612**

loss	**0.115**	**0.9778**	**0.9855**

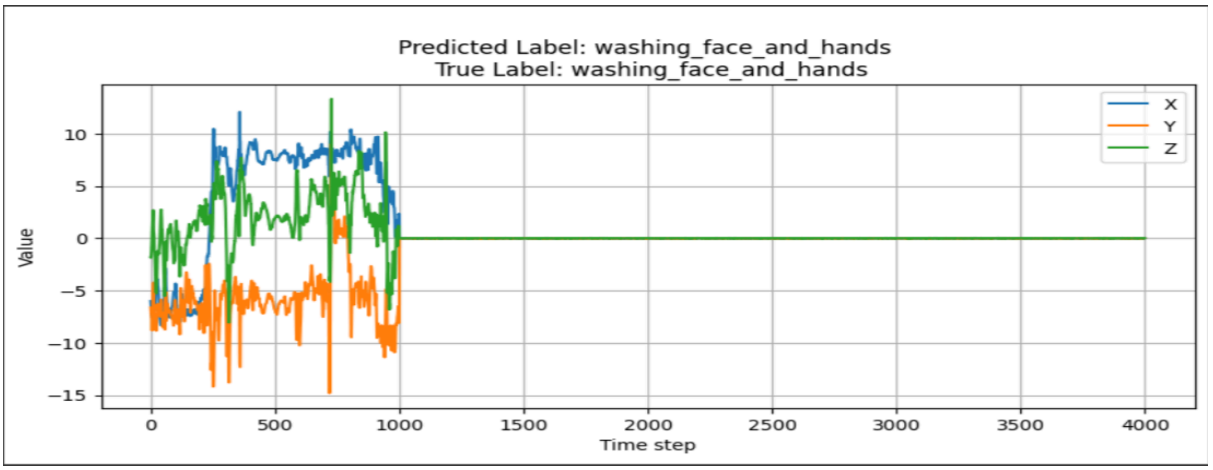
Model classification and misclassification

Misclassifaction :



the model predicted the label "walking_with_handbag" instead of the true label "washing_face_and_hands." The plot reveals that the sequence's values are relatively stable across all three dimensions (X, Y, Z) with no significant peaks or erratic movements that would typically be associated with an activity involving walking with a handbag. The relatively steady pattern might be more characteristic of a stationary activity, such as washing one's face and hands, which involves less overall body movement. The misclassification could be due to the model not learning the subtle differences between the two activities or due to similarities in the dataset where the stationary phases of "walking_with_handbag" might resemble "washing_face_and_hands." This could indicate that the model might be over-generalizing from the data, or the features extracted are not discriminative enough to distinguish between these activities accurately.

Correct classification:



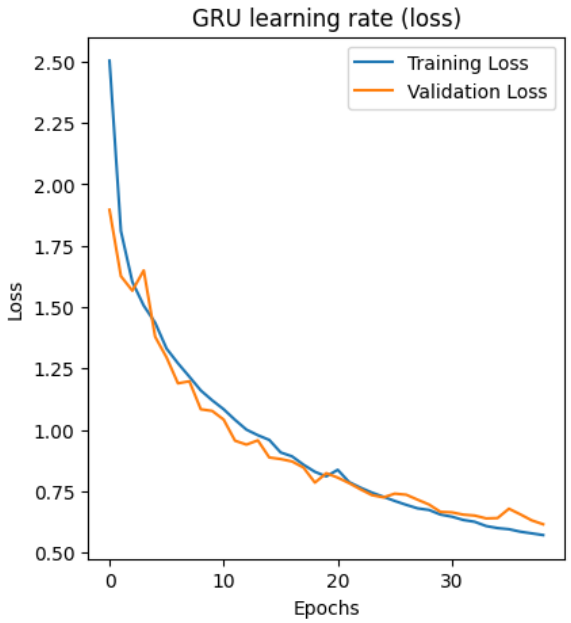
the predicted label "washing_face_and_hands" matches the true label. The sequence here exhibits more variation and some transient spikes in the X and Y dimensions, which might correspond to the natural movements of a person washing their face and hands, such as reaching out for soap or splashing water, movements that would be detected by motion sensors. The correct prediction here suggests that the model has effectively learned the characteristic patterns for this activity from the training data. The presence of distinctive, possibly more pronounced movements in the sequence may have provided clear signals that the model could use to differentiate this activity from others in the dataset. It implies that for this particular instance, the model's feature representation was robust enough to capture the essence of the activity, leading to a correct classification.

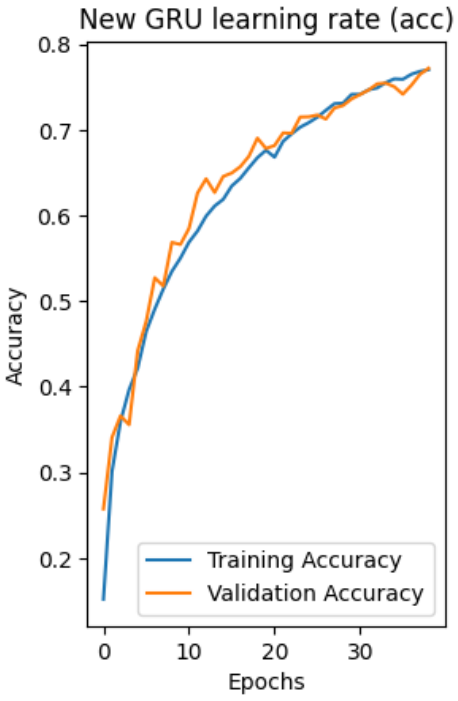
Gru model architecture :

```
GRUNet(
  (gru): GRU(3, 100, num_layers=2, batch_first=True)
  (fc_a): Sequential(
    (0): Linear(in_features=100, out_features=1024, bias=True)
    (1): ReLU()
    (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=1024, out_features=512, bias=True)
    (5): Sigmoid()
    (6): Dropout(p=0.5, inplace=False)
  )
  (fc_b): Linear(in_features=512, out_features=18, bias=True)
```

The model consists of a GRU layer with 3 input features and **100 hidden units**, stacked into **2 recurrent layers** where batch processing is prioritized. Following the recurrent layers, the network has a **fully connected** section organized sequentially: it first linearly transforms the GRU output to a 1024-dimensional space, applies a **ReLU** activation function for non-linearity, then normalizes the batch using **BatchNorm** with momentum of 0.1 to stabilize learning. A **dropout layer** with a probability of 0.5 is used to mitigate overfitting by randomly zeroing out some of the features. The next linear layer reduces the dimensionality to 512, followed by a **sigmoid** activation function for binary classification tasks. Another **dropout layer** with the same probability precedes the final linear layer which maps the 512 features to 18 output features, likely representing the classes for the final classification task. The consistent use of dropout indicates a focus on reducing overfitting, and the **sigmoid** function suggests a binary classification for each of the 18 outputs

	Train	Val	Test
Acc	0.7704	0.7721	0.7713
Loss	0.5710	0.6151	0.5876

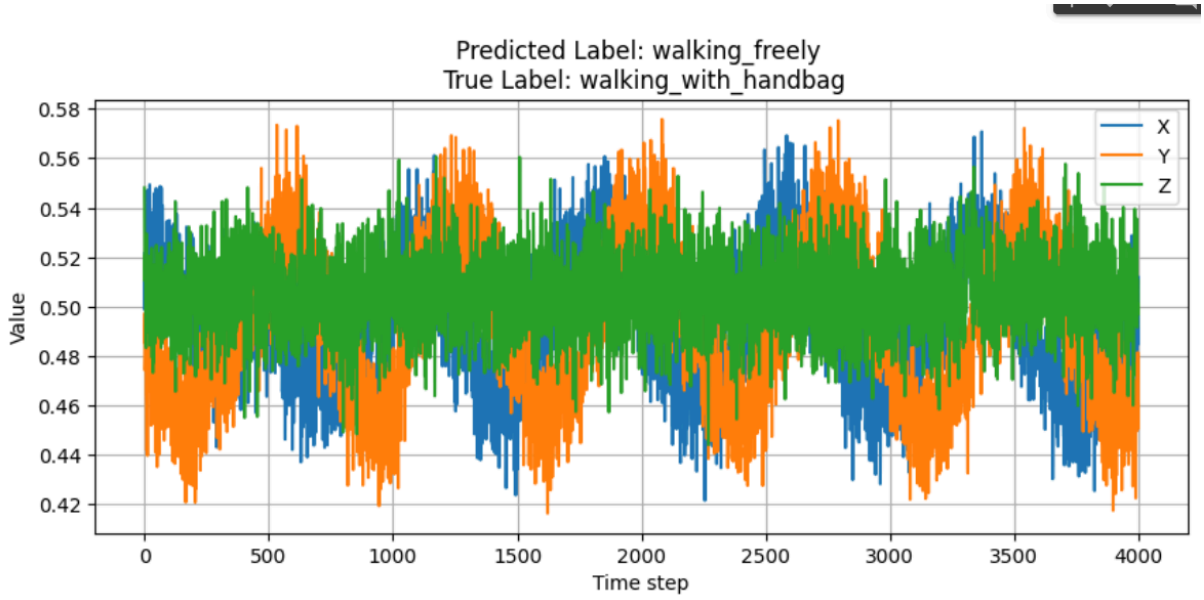




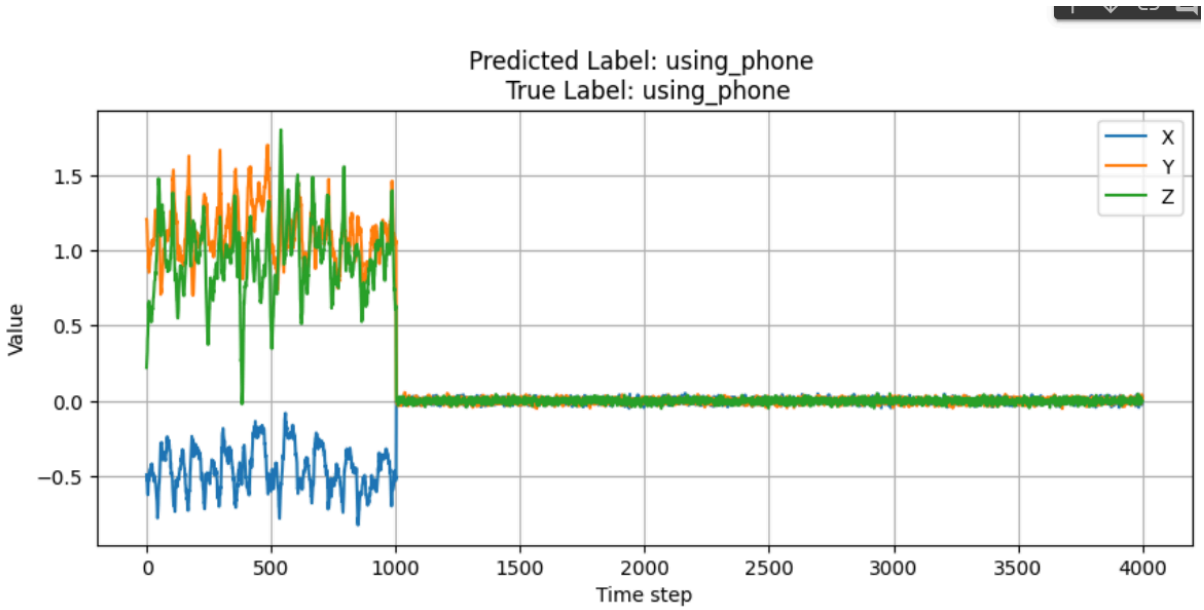
Additional metrics for each one of the classes:

	precision	recall	f1-score	support
0	0.96	0.84	0.89	219
1	0.69	0.81	0.75	398
2	0.60	0.85	0.71	361
3	0.85	0.71	0.77	336
4	0.98	0.98	0.98	162
5	0.98	0.99	0.98	203
6	0.70	0.81	0.75	165
7	0.75	0.76	0.75	671
8	0.76	0.69	0.72	370
9	0.70	0.71	0.70	710
10	0.82	0.83	0.83	688
11	0.69	0.76	0.72	668
12	0.74	0.88	0.80	604
13	0.89	0.62	0.73	671
14	0.86	0.72	0.78	388
15	0.82	0.76	0.79	374
16	0.83	0.76	0.79	354
17	0.81	0.75	0.78	196
accuracy			0.77	7538
macro avg	0.80	0.79	0.79	7538
weighted avg	0.78	0.77	0.77	7538

Missclassification:



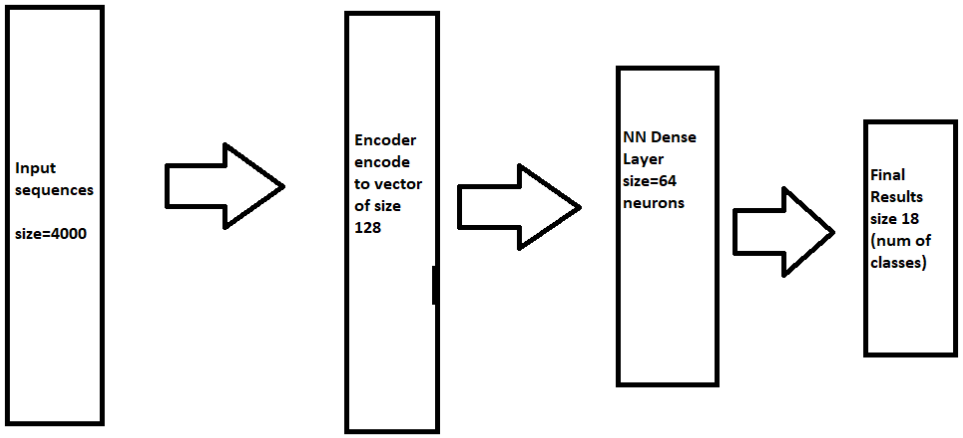
Correct Classification:



E)

We chose the self supervised task 1: LSTM Auto encoders. We trained auto encoders on the whole unlabelled data, a final loss of 1.1.

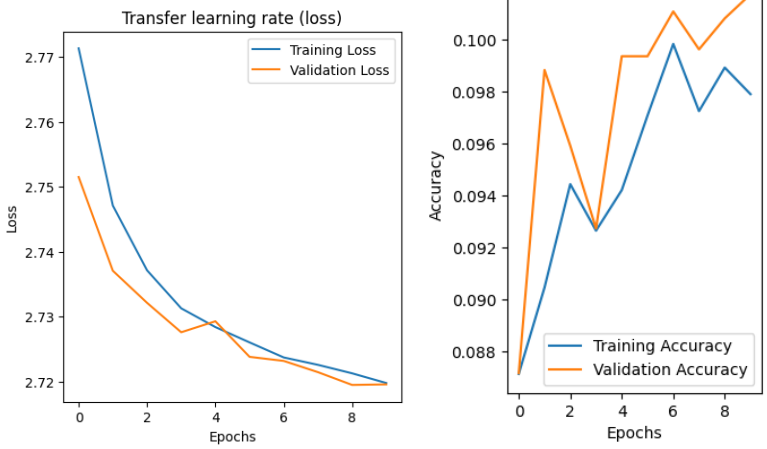
Then used its embedding representation(output of encoders), of size 128, as input to a neural network. The following diagram represents the network archieticture



Results:

	Train	Val	Test
Acc	0.0979	0.1018	
Loss	2.7198	2.7196	

Learning rate



Looks like that the model is training but the score is not good, we think that changing the embedding size and the complexity of the network would enhance the model

f)

Suggested improvements:

1. **Normalize the data:** We think that normalizing the data would make the model more stable. We can either standardize or use min-max normalization. Normalizing the data would oragnize the data into one scale, this is important because our data (the acceleration sequences) recorded using different units, devices, body parts, and different users.
2. **Augmentation:** Augmentation and insetting noise to the train data would definitely let the model generalize better. Since the data is recorded from different users. In addition sensors are not always accurate and might mis-record some data. Applying noise to the data would let the model focus more on detecting patterns and anomalies rather than recognizing the data itself.
3. **Ensemble:** One of our future experiments is to make Ensemble learning model that is combined by the different models we trained. Since each model extracts its own features, we can combine them to build more complex models.

   We suggest 2 approaches for this method:

   1. Build a neural network (or single perceptron) such that its input is the output class of each of the trained models.
   2. Remove the last layer of each trained model to get theri last layer features vector (except for ML take only the features we extracted). Contacetane all the features and send them as an input to a new model. We have many options for the type of this new model, it can be classical ML model or neural network.

G.

We choose suggestion 1 and 2 to implement. For **normalization** we first tried standardization, but it resulted very high loss (about 200), it may be a technical issue but we directly moved to Min-Max scaler. For **Augmentation** we added 3 layers of noise on each training sequence:

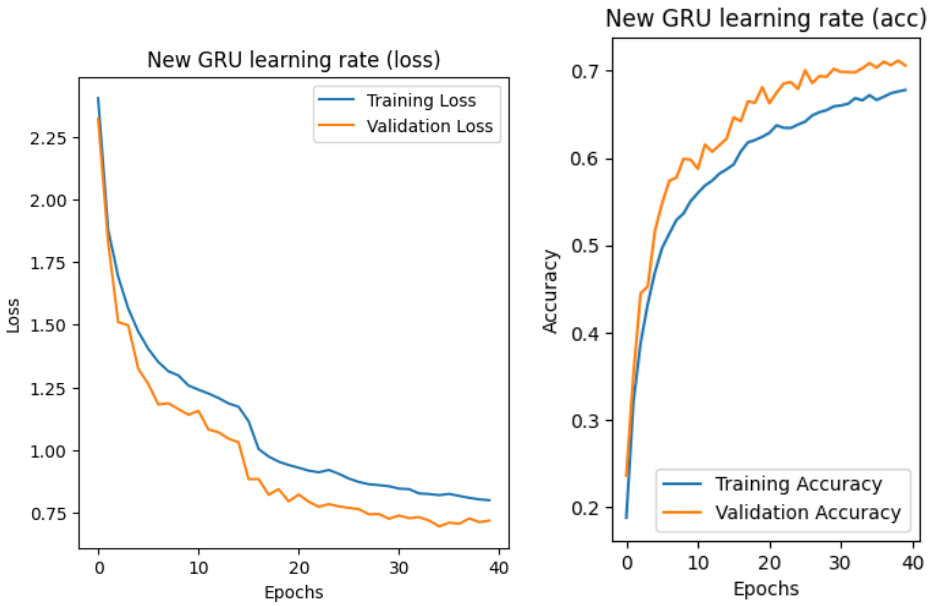
1. **Gaussian noise**, mean=0, std = 0.05.
2. **Uniform distribution** noise, values range from -0.03 to 0.1
3. **Random cropping:** We cropped random part of the sequence (cropped part is displayed as zeroes in the sequence)

The above Augmentation methods did not perform well from the beginning, so we have done kind of parameter tuning to get these parameters (like std = 0.05)

After applying these two improvements we noticed that our model is generalizing better, because at each epoch the model validation score was better than training score. One disadvantage of these modifications is that the model now is learning slower.

	Train	Val	Test
Acc	0.6778	0.7059	0.7265
Loss	0.7997	0.7183	0.6740

Here is a learning rate plot of the new model:



Additional metrics for each one of the classes:

	precision	recall	f1-score	support
0	0.96	0.82	0.88	219
1	0.41	0.86	0.56	398
2	0.71	0.73	0.72	361
3	0.98	0.63	0.76	336
4	1.00	1.00	1.00	162
5	0.99	1.00	0.99	203
6	0.97	0.65	0.78	165
7	0.92	0.65	0.76	671
8	0.83	0.49	0.62	370
9	0.38	0.94	0.54	710
10	0.98	0.67	0.79	688
11	0.95	0.61	0.74	668
12	1.00	0.70	0.82	604
13	0.96	0.65	0.78	671
14	0.65	0.84	0.73	388
15	0.95	0.72	0.82	374
16	0.94	0.76	0.84	354
17	0.83	0.68	0.75	196
accuracy			0.73	7538
macro avg	0.86	0.74	0.77	7538
weighted avg	0.84	0.73	0.75	7538

Performance on kaggle:
Although the final loss on the given dataset were higher than the previous model (without normalization and augmentation) this model kept stable and its score was better on the kaggle competition. The following table shows the difference between the two models

Model	Loss on dataset	Score in competition
Old	~ 0.458	2.272
New	0.799	1.778

Conclusion
According to the results we can consider the augmentation and normalization as a game changer. It has contributed significantly and improved the score with a huge gap. We know that we can improve our models much more, we have more experiments to try and learn, but unfortunately we did not have enough time to try all of our ideas. One of our future work would be utilizing the self supervised LSTM Auto Encoders trained embeddings. We also think that self supervision can be powerful for tasks similar to this one. Note that we implemented a self supervised solution after the kaggle competition ended.