Representation Learning
Assignment 1 - Report

Student Name: **Maryam Alipourhajiagha**

Q1: Implementing and Testing a MLP on HAR Time Series

1.1- MLP Model Implementation :

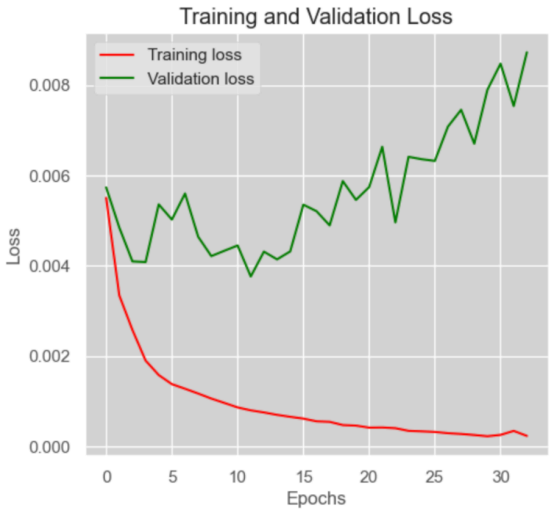
Here I implemented the MyMLP based on the default configuration of the question.

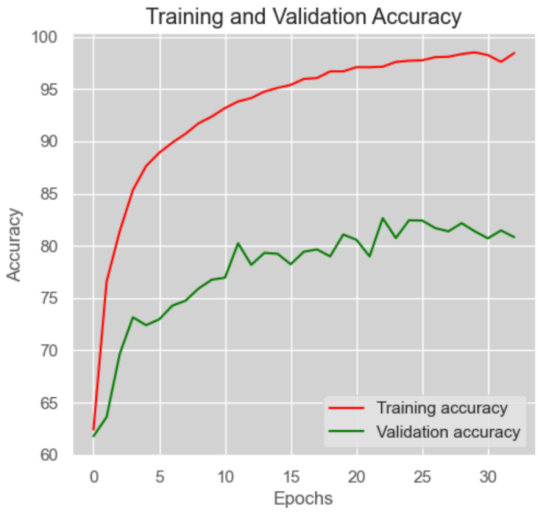
```
MyMLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (fc4): Linear(in_features=100, out_features=6, bias=True)
)
```

Hyperparameters of my models follow as below:
Batch Size = 200, Epochs = 500, Optimizer = Adam, Learning Rate=0.001,
Patience (Early Stopping) =10, Delta (Early Stopping) = 0.001

By looking at the below charts, it's apparent that while the training loss decreases consistently, indicating good learning within the training dataset, the validation loss starts to plateau and even slightly increase after a certain number of epochs. This is a classic indication of the model beginning to overfit to the training data. Implementing early stopping helps prevent this overfitting by terminating the training process when the validation loss has ceased to decrease and starts to rise, ensuring the model maintains its generalizability. The use of early stopping also saves on computational resources by avoiding unnecessary training epochs that do not contribute to model improvement.

```
Accuracy on dev set:  80.8261233603126
Loss on dev set:  0.008732044527812524
```





1.2 - Hyperparameter Tuning :

First Configuration:

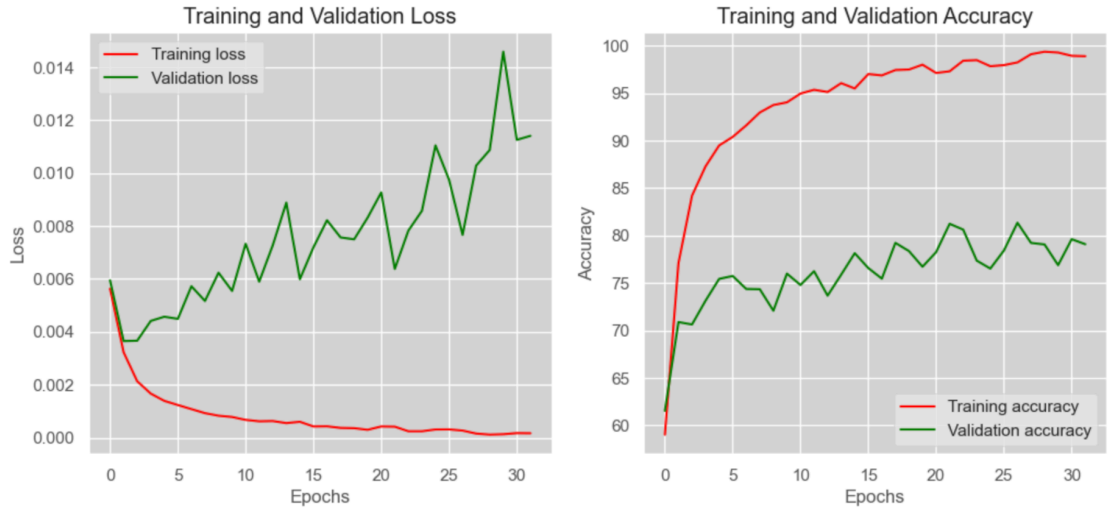
I added two more layers and neurons to MyMLP models without changing other hyperparameters, which has below architecture:

```
MyMLP_Configuration_1(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=64, bias=True)
  (fc5): Linear(in_features=64, out_features=32, bias=True)
  (fc6): Linear(in_features=32, out_features=6, bias=True)
)
```

Hyperparameters of my models follow as below:
Batch Size = 200, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) =10, Delta (Early Stopping) = 0.001

As we can see, even though we might expect that by adding more layers and neurons the capacity of the model should increase and therefore perform better, it did not show improved performance on the validation set. This outcome could be attributed to several probable reasons: 1) Overfitting, where the model learns the training data too well but fails to generalize to unseen data; 2) Vanishing gradients, which can occur in deep networks and hinder the learning process; 3) Optimization difficulty, as deeper networks are indeed harder to optimize; 4) The need for parameter adjustment, since with additional layers, it's often necessary to tune other hyperparameters, such as learning rate, batch size, and regularization terms.

```
Accuracy on dev set:  79.09572983533351
Loss on dev set:  0.011410940559424926
```



Second Configuration:

I kept the base MLP model (MyMLP) architecture and changed the activation function to Leaky-Relu and Patience parameter to 20. Below is the Architecture and hyperparameters:

```
MyMLP_Configuration_2(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (fc4): Linear(in_features=100, out_features=6, bias=True)
  (leaky_relu): LeakyReLU(negative_slope=0.01)
)
```

Hyperparameters of my models follow as below:
Batch Size = 200, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 20, Delta (Early Stopping) = 0.001

Changing the activation function to LeakyReLU and setting a higher patience for early stopping seem to have impacted the model's performance. The use of LeakyReLU can help mitigate the issue of dying ReLU units—a problem where neurons can become inactive and stop learning entirely—as it allows a small gradient when the unit is not active. This change may have enabled the network to continue learning where it might have stopped with the ReLU activation function.
Increasing the patience in early stopping allows the model more time to find and possibly recover from local minima or plateaus in performance. This extended opportunity to learn could lead to the model capturing more nuanced patterns in the data, reflected in the steadier improvement in validation accuracy.

```
Accuracy on val data:  82.08205414457159
Loss on val data:  0.011295359585201692
```

It's worth noting that while the validation accuracy is not perfect and there are fluctuations, the overall trend is positive. The graph suggests that the model could benefit from further hyperparameter tuning.

Third Configuration:

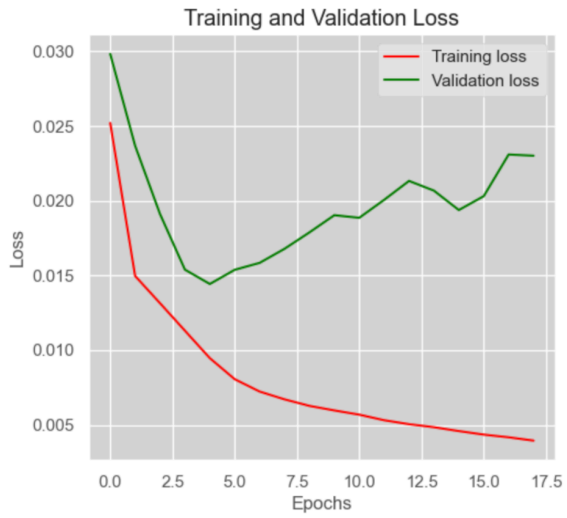
I used the more complex model with more layers and neurons in first configuration and a learning rate of 0.001.

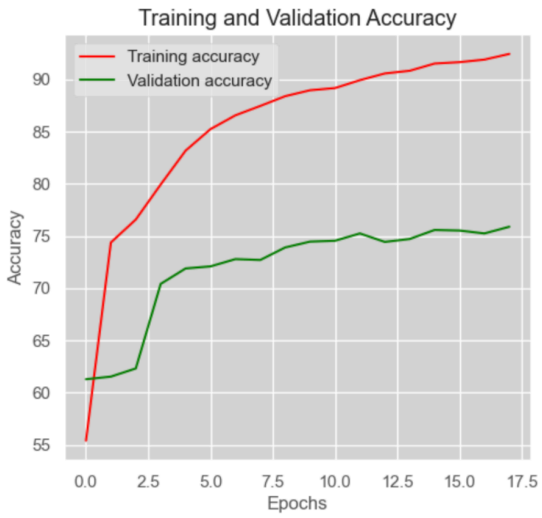
```
MyMLP_Configuration_3(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=64, bias=True)
  (fc5): Linear(in_features=64, out_features=32, bias=True)
  (fc6): Linear(in_features=32, out_features=6, bias=True)
)
```

Hyperparameters of my models follow as below:
Batch Size = 50, Epochs = 500, Optimizer = Adam, Learning Rate=0.0001
Patience (Early Stopping) = 10, Delta (Early Stopping) = 0.001

We see that in comparison to the base model the accuracy on validation set decreased. However by reducing the batch size to 50 we helped to prevent overfitting (it's still overfit) to some extent because they provide more noise during gradient estimation. This noise can act as a form of regularization, making the model potentially generalize better but also making the learning curves more erratic. Smaller learning rates can lead to more stable training and avoid drastic updates that might cause the model to diverge or overshoot minima. However if the learning rate is too small, the model might get stuck in a plateau or a local minimum and might not be able to escape due to insufficiently large weight updates.

```
Accuracy on val data:  75.88612894222719
Loss on val data:  0.023012284307226 36
```





Fourth Configuration:

I added more hidden layers and neurons and used tanh activation function and using Delta = 0.01.

MyMLP_Configuration_4(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=128, bias=True)
  (fc5): Linear(in_features=128, out_features=64, bias=True)
  (fc6): Linear(in_features=64, out_features=32, bias=True)
  (fc7): Linear(in_features=32, out_features=6, bias=True)
)

Hyperparameters of my models follow as below:
Batch Size = 200, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 10, Delta (Early Stopping) = 0.01

As we can see more complexity of the model in this configuration was more helpful. And using the tanh activation function model since it centers the output around zero, can lead to faster convergence during training due to better data distribution and gradients flow. Additionally, the tanh function's negative values can also help the model learn more complex patterns by providing a stronger gradient signal for negative inputs

compared to ReLU. And the model went through all 500 epochs without triggering early stopping because the validation loss was decreasing by more than 0.01 at least once within the patience period throughout the training process.

```
Accuracy on val data:  79.23527770025119
Loss on val data:  0.0077274887298293335
```

1.3 - Model Performance and Hyperparameter Analysis :

In the previous section I analyzed different model architecture and altered some of the hyperparameters. Here is the confusion matrix and the evaluation of the best performance model (Second Confguration).

Each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class. Here's an interpretation based on the image:

1) Downstairs: The model has moderate difficulty distinguishing the 'Downstairs' activity, with a notable number of false positives where 'Downstairs' is confused with 'Walking' (120 instances) and 'Jogging' (50 instances).

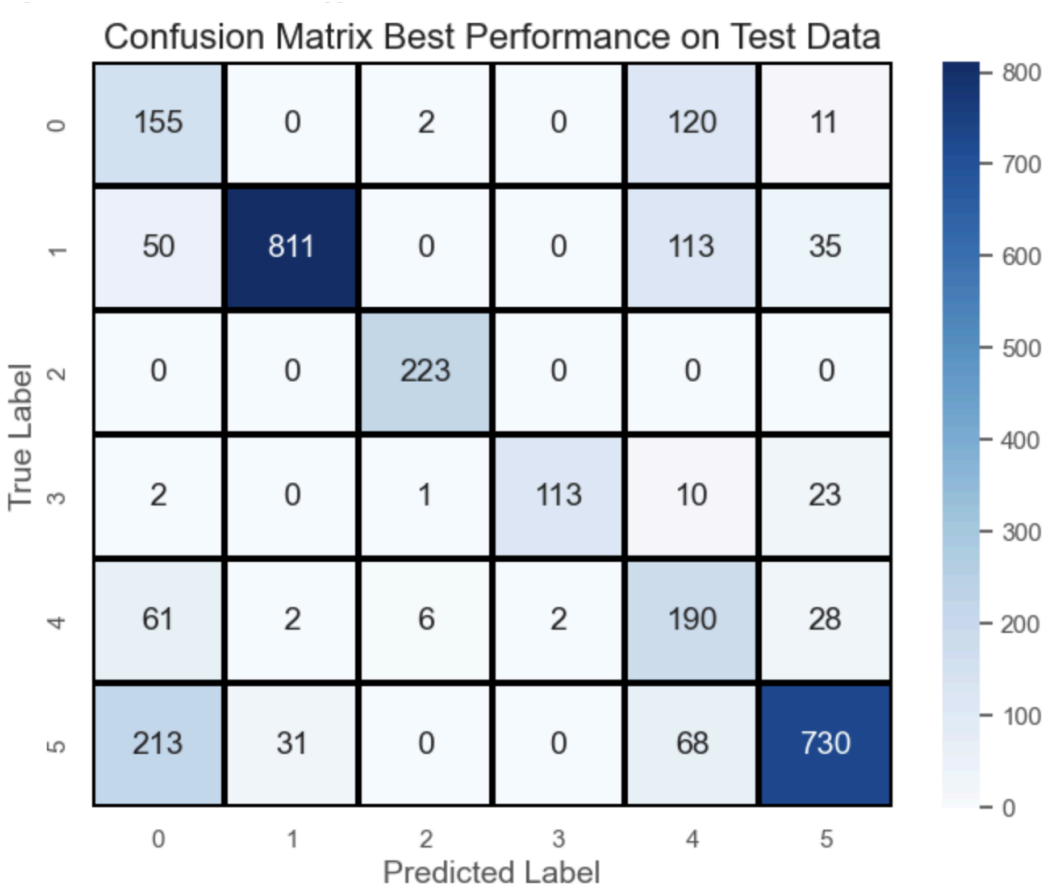
2) Jogging: The model performs very well in classifying 'Jogging', with a high recall of 0.80. However, there are still 35 instances where 'Jogging' was mistaken as 'Walking'.

3) Sitting: This activity is perfectly classified with no confusion with other activities.

4) Standing: 'Standing' is occasionally confused with 'Walking' and 'Upstairs', but overall, the model performs well on this class.

5) Upstairs: The model has some difficulty with 'Upstairs', often confusing it with 'Walking' (213 instances), which is the largest number of off-diagonal entries in the matrix.

6) Walking: While 'Walking' has a relatively high precision of 0.88, suggesting that



	precision	recall	f1-score	support
0	0.32	0.54	0.40	288
1	0.96	0.80	0.88	1009
2	0.96	1.00	0.98	223
3	0.98	0.76	0.86	149
4	0.38	0.66	0.48	289
5	0.88	0.70	0.78	1042
accuracy			0.74	3000
macro avg	0.75	0.74	0.73	3000
weighted avg	0.82	0.74	0.77	3000

when the model predicts 'Walking', it is correct most of the time, it also has a significant number of false negatives, particularly with 'Upstairs' and 'Downstairs'.

> The recall of 0.70 shows that 'Walking' is often mislabeled as another activity. This is mostly because of that this activity consists most of the dataset.

In summary, while the model performs well for 'Jogging' and 'Sitting', it struggles to differentiate between 'Walking' and other activities involving movement, such as 'Upstairs' and 'Downstairs'. To improve the model, we could focus on gathering more discriminative features for these activities or explore more complex model architectures. Additionally, considering class-specific weights or sampling strategies could help address the imbalances in model performance across different activities.

1.4 - Confusion Matrix Evaluation :

A confusion matrix is key for pinpointing exactly where a classification model excels or falls short, offering a detailed view of each class's performance. It highlights the model's specific errors, such as frequent misclassifications between classes, which is vital for refining the model. Precision and recall, derived from the matrix, inform you about the reliability of predictions and the model's ability to capture all relevant cases. This detailed breakdown is essential for targeted improvements and understanding the potential real-world implications of the model's predictions.

Q2: Implementing and Testing a 1D CNN on HAR Time Series

2.1 - 1D CNN Model Implementation :

I implemented MyConvModel as requested architecture with early stopping.
As MLP models all the configurations are included early stopping so that I can understand other hyperparameters impact on models' performance.

```
MyConvModel(
  (conv1): Conv1d(3, 100, kernel_size=(10,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(10,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(10,), stride=(1,))
  (conv4): Conv1d(160, 160, kernel_size=(10,), stride=(1,))
  (pool): MaxPool1d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.5, inplace=False)
```

```
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (fc): Linear(in_features=160, out_features=6, bias=True)
)
```

Hyperparameters of my models follow as below:
Batch Size = 400, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 10, Delta (Early Stopping) = 0.01
Number of Convolutional Layers = 4, Number of Input Channels for each Layer, Number of Output Channels for each Layer, Kernel Size = 10, Stride Size for each Layer, Max Pooling: Size = 3 and Stride = 3, Drop Out Rate = 0.5, Adaptive Pooling = Output Size 1

The switch to a CNN model has led to better results, as evidenced by the training and validation curves. The training loss has decreased and stabilized at a lower level, suggesting the model is learning effectively, as the validation loss follows a similar trend with less volatility. The training accuracy is nearly perfect, and the validation accuracy is significantly higher and more stable than with the previous MLP model, indicating strong generalization and the CNN's ability to capture spatial hierarchies and features in the data, which are essential for activity recognition tasks.

```
Accuracy on val data:  81.88668713368685
Loss on val data:  0.005447206837303173
```

Even though overfitting would typically be indicated by a significant divergence between the training and validation curves, especially if the validation loss increases while the training loss continues to decrease. However, by looking at the curves, both the training and validation loss exhibit a downward trend, although the validation loss does show some volatility with occasional spikes.

2.2 - Hyperparameter Tuning :

Here I change and modify hyperparameters in relative to the base model to interpret the effects of each alternation.

First Configuration:

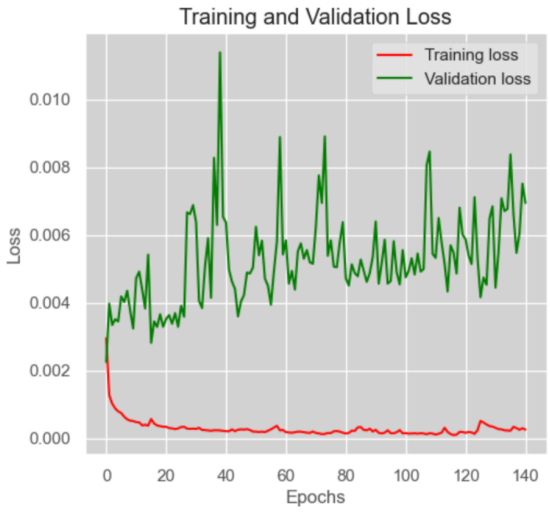
I removed the 4th conv layer and decreased the kernels size to 15. I expect fewer layers reduce overfitting and larger filters could improve the ability to capture more spatial context, potentially improving generalization. The modifications follow as below:

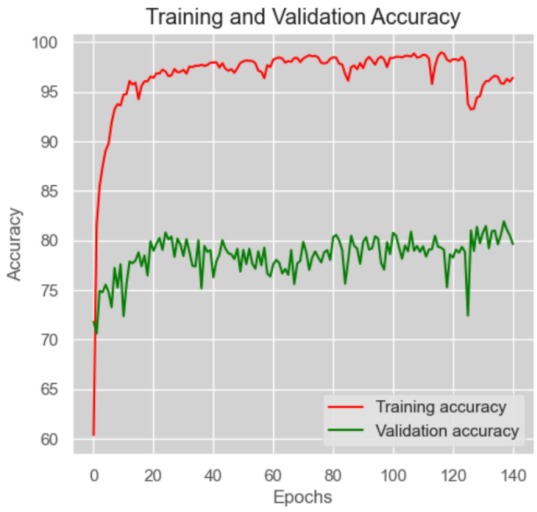
```
MyConvModel_Configuration1(
  (conv1): Conv1d(3, 100, kernel_size=(15,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(15,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(15,), stride=(1,))
  (pool): MaxPool1d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.5, inplace=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (fc): Linear(in_features=160, out_features=6, bias=True)
)
```

Hyperparameters of my models follow as below:
Batch Size = 400, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 10, Delta (Early Stopping) = 0.01
Number of Convolutional Layers = 3, Number of Input Channels for each Layer, Number of Output Channels for each Layer Kernel Size = 15, Stride Size for each Layer, Max Pooling: Size = 3 and Stride = 3, Drop Out Rate = 0.5, Adaptive Pooling = Output Size 1

By looking at the curves we see that it did not go as we expected maybe the influence of other hyperparameters were stronger preventing the model to perform well enough.

```
Accuracy on val data:  79.62601172202065
Loss on val data:  0.0069470142164823065
```





Second Configuration:

I reduced the size of kernels from 10 to 3 since Smaller kernels capture finer, more localized features in the input data. This can be beneficial when the important features are small or fine-grained. Smaller kernels require more layers to achieve a broader view, increasing depth and feature hierarchy. So I added one more convolution layer which has 160 channels and output 240 channels. Adding a layer with more output channels can help the network learn a richer representation of the data.

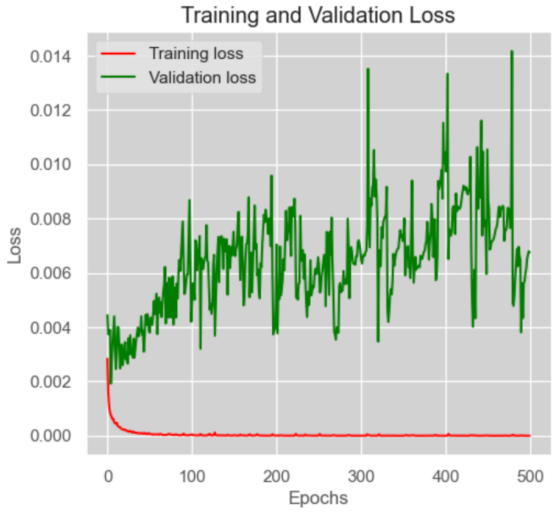
MyConvModel_Configuration2(
  (conv1): Conv1d(3, 100, kernel_size=(3,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(3,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(3,), stride=(1,))
  (conv4): Conv1d(160, 160, kernel_size=(3,), stride=(1,))
  (conv5): Conv1d(160, 240, kernel_size=(3,), stride=(1,))
  (pool): MaxPool1d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.5, inplace=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (fc): Linear(in_features=240, out_features=6, bias=True)

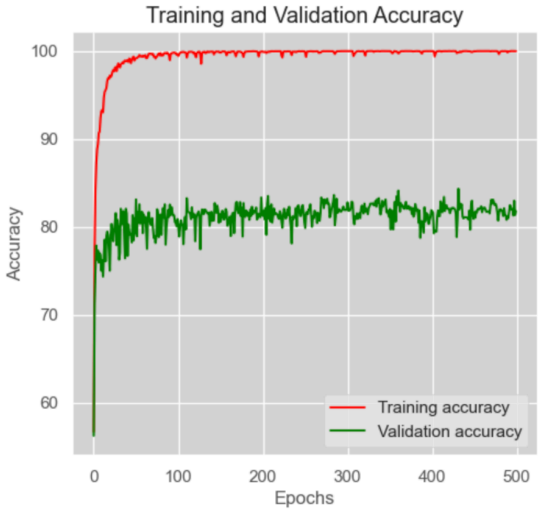
Hyperparameters of my models follow as below:
Batch Size = 400, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 10, Delta (Early Stopping) = 0.01

Number of Convolutional Layers = 5, Number of Input Channels for each Layer, Number of Output Channels for each Layer , Kernel Size = 3, Stride Size for each Layer, Max Pooling: Size = 3 and Stride = 3, Drop Out Rate = 0.5, Adaptive Pooling = Output Size 1

By looking at the curves we see that it did not go as we expected maybe the influence of other hyperparameters were stronger preventing the model to perform well enough. Also the early stopping has not been triggered. Early stopping may not trigger if the model consistently shows even slight improvements, staying within the set 'delta' threshold, or if the 'patience' parameter is too lenient, allowing the model to continue training despite minimal gains.

```
Accuracy on val data:  81.77504884175272
Loss on val data:  0.006760053441371376
```





Third Configuration:

I modified three hyperparameters simultaneously this time. First removed one conv layer, and then used leaky_relu activation function in the forward method while passing the conv layers and switched the max pooling with average pooling.

```
MyConvModel_Configuration3(
  (conv1): Conv1d(3, 100, kernel_size=(10,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(10,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(10,), stride=(1,))
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.5, inplace=False)
```

(fc): Linear(in_features=160, out_features=6, bias=True)
)

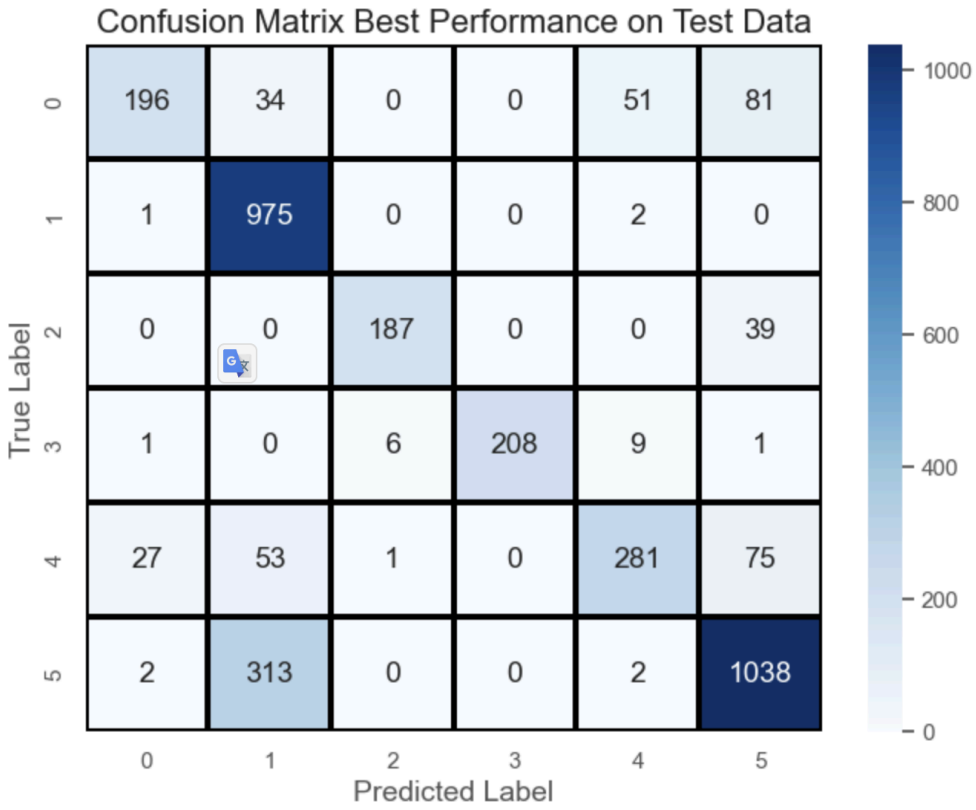
Hyperparameters of my models follow as below:
Batch Size = 400, Epochs = 500, Optimizer = Adam, Learning Rate=0.001
Patience (Early Stopping) = 20, Delta (Early Stopping) = 0.01
Number of Convolutional Layers = 3, Number of Input Channels for each Layer, Number of Output Channels for each Layer , Kernel Size = 10, Stride Size for each Layer, Drop Out Rate = 0.5, Adaptive Average Pooling = Output Size 1

The improvement of performance can be justified. Removing a convolutional layer can streamline the model, potentially reducing overfitting and computational cost, while the switch to LeakyReLU may prevent dead neurons and ensure a consistent gradient flow, aiding the training of deeper models. Replacing max pooling with adaptive average pooling smooths out the feature maps, which could lead to more stable and generalizable features by focusing on the average presence of a feature instead of its strongest signal.

2.3 - Confusion Matrix Evaluation :

The advantage of using CNN is its ability to capture time-dependent features is highlighted in the precision and recall of specific activities, such as 'Sitting' and

'Jogging', which are sequential and have distinct motion patterns over time. Also by looking at the confusion matrix we can interpret that the optimal configuration we"ve arrived at, with fewer convolutional layers and the use of adaptive average pooling, suggests that simplifying the model has helped it focus on the most salient features in the data, which are crucial for time-series classification. Adaptive average pooling is particularly effective here, as it ensures that the most important signals are captured regardless of their position in the time series, and it helps the model to be more robust to variations in the input data. The use of LeakyReLU likely helped maintain gradient flow, which is important for learning complex patterns in time-dependent data.



	precision	recall	f1-score	support
0	0.86	0.54	0.67	362
1	0.71	1.00	0.83	978
2	0.96	0.83	0.89	226
3	1.00	0.92	0.96	225
4	0.81	0.64	0.72	437
5	0.84	0.77	0.80	1355
accuracy			0.81	3583
macro avg	0.87	0.78	0.81	3583
weighted avg	0.82	0.81	0.80	3583

Here is the analysis of confusion matrix and classification report for each activity:

1) Downstairs: The model predicted 196 correctly as class 0, but there were significant misclassifications as class 5 (81 instances) and class 1 (34 instances). The precision is 0.86, but the recall is only 0.54, suggesting the model is missing a good portion of the actual class 0 instances.
2) Jogging: The model is very accurate with class 1, with 975 out of 978 instances correctly identified, giving it perfect precision and a recall of 1.00. This indicates excellent model performance for class 1.
3) Sitting: This class also sees a good level of correct predictions (187 out of 226), with a precision of 0.96 and a recall of 0.83.
4) Standing: The model correctly predicted 208 out of 225 instances, with both high precision and recall, indicating reliable performance in class 3.
5) Upstairs: There were 281 correct predictions out of 437 instances. The model has lower precision and recall for this class (0.81 and 0.64, respectively), suggesting it often confuses class 4 with other classes, particularly class 1 and class 5.
6) Walking: The model is highly accurate in predicting class 5, with 1038 out of 1355 instances correctly identified. However, there's a notable number of instances where class 5 is mistaken for class 1 (313 instances).

Summary:

The model performs exceptionally well in distinguishing class Jogging and fairly well for classes Sitting and Standing. However, there is some confusion between class Downstairs and classes Upstairs and Walking, as well as between class Upstairs and class Jogging, which suggests that these classes have features that the model finds hard to distinguish. The model may benefit from further training or refinement to better differentiate these classes. The high misclassification rate of class Downstairs instances as class Walking and class Jogging instances as class Walking suggests a potential similarity in the feature space that the model is sensitive to.

Q3: Reflections of the Implementation and Experimentation

3.1- Data Augmentation :

3.1.a: Noise Addition

I added Guassian noise to my data and trained the best performance model which is third configuration of CNN models on this data.

If the real-world data is likely to contain noise, training with noisy data can make our model more robust to such variations. This is particularly useful in our scenario where the data is collected from sensors.

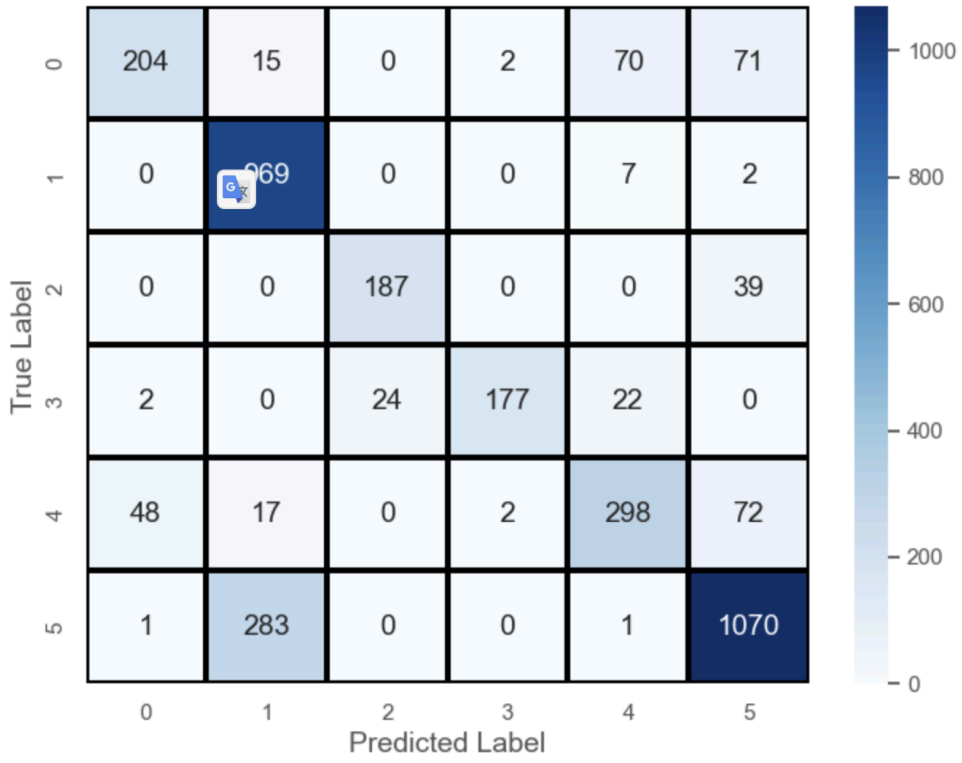
This is how I made my new data containing gaussian noise with noise_level = 0.05:

```
def add_noise(data, noise_level):

    noise = np.random.normal(0, noise_level, data.shape)

    return data + noise
```

```
Accuracy on dev set:  91.76666666666667
Loss on dev set:  0.0010326663007338842
```

The observed enhancement in performance after adding Gaussian noise was modest, which may not be as substantial as expected. This could be attributed to the possibility that the dataset already contained a sufficient level of noise. Consequently, introducing additional noise at the chosen level did not significantly impact the model's performance.

As we can see the validation loss has started lower than the training loss. Here is the possible reasons: Different Data Distribution (which in our case is not the reason), Smaller Batch Size (they both have the same batch size in my mode), Initial Weights and Biases.



	precision	recall	f1-score	support
0	0.80	0.56	0.66	362
1	0.75	0.99	0.86	978
2	0.89	0.83	0.86	226
3	0.98	0.79	0.87	225
4	0.75	0.68	0.71	437
5	0.85	0.79	0.82	1355
accuracy			0.81	3583
macro avg	0.84	0.77	0.80	3583
weighted avg	0.82	0.81	0.81	3583

3.1.b: Time Shifting :

I augmented my best-performing model with time-shifted data, reasoning that this approach could further enhance the model's performance. Introducing such variations in the training set is thought to be beneficial because it encourages the model to learn more generalized and invariant features of the data, rather than memorizing sequences and timings that are too specific. By training the model to recognize patterns despite temporal shifts, it's expected to become better at making accurate predictions across a range of conditions, thus boosting its overall performance.

```
def time_shift(data, shift_steps = 30):
    shifted_data = np.roll(data, shift=shift_steps, axis=1)
    return shifted_data
```

```
Accuracy on dev set:  91.43333333333334
Loss on dev set:  0.005195095945706536
```

After adding time-shifted data to the model, the improvement in accuracy wasn't as much as we hoped. This might be because of different settings in the model, like how much we shifted the data in time. It went up by just one percent. This hints that we might need to tweak the model settings more to really see the benefits of adding the shifted data.

	precision	recall	f1-score	support
0	0.48	0.41	0.44	362
1	0.85	0.96	0.90	978
2	0.76	0.83	0.79	226
3	0.99	0.89	0.94	225
4	0.66	0.72	0.69	437
5	0.85	0.78	0.82	1355
accuracy			0.79	3583
macro avg	0.76	0.76	0.76	3583
weighted avg	0.79	0.79	0.79	3583

After we added data that was shifted in time, the updated confusion matrix shows the model got better at recognizing 'walking' and 'Upstairs' activities. But it wasn't as good at spotting 'Downstairs' as the earlier version of the model that had Gaussian noise. This means that changing the timing in the data helped in some cases but made it harder for the model to tell when someone was Downstairs.This could be due to various reasons specific to the nature of the activities and the data distribution:

1) Walking and Upnstairs activities have distinctive patterns that are not significantly affected by the time shift, allowing the model to still recognize the essential features despite the time lag.

2) Downstairs might be more sensitive to time shifts, possibly because the data for this activity depends more on the precise timing of the features.
3) It's also possible that time shifting has introduced overlaps with patterns from other activities, making it harder for the model to distinguish Downstairs from other similar activities like going upstairs.

3.1.c:

In the context of image data, like the datasets used for ImageNet challenges, several augmentation techniques are widely employed to improve the robustness and generalization of the models. Some of these techniques could potentially be adapted for time-series data, such as sensor data, audio signals, or any other form of sequential data. Here's how they translate:

1) Random Cropping and Scaling (ImageNet) to Random Subsampling and Stretching (Time-Series):
    - In images, random cropping can help the model focus on different parts of an image, while scaling changes the size of the objects in the image.
    - For time-series, similar effects can be achieved by randomly selecting subsequences (subsampling) and stretching or compressing these sequences in time (similar to changing the sampling rate).
2) Rotation and Flipping (ImageNet) to Shifting and Inverting (Time-Series):
    - Rotations and flips are used to train the model to recognize objects from different orientations.
    - In time-series, a similar concept could involve circular shifting (rotating the data points) or inverting signal values, especially for sensor data where direction might change due to orientation.
3) Color Jittering (ImageNet) to Amplitude Transformation (Time-Series):
    - Color jittering changes the colors in an image to various shades, making the model less sensitive to color variations.
    - Amplitude transformation varies the magnitude of time-series data, helping models to learn from different signal strengths or sensor sensitivities.
4) Random Erasing or Cutout (ImageNet) to Random Occlusion (Time-Series):
    - Removing parts of an image randomly forces the model to focus on less prominent features.

- Randomly occluding parts of a time-series (setting some values to zero or a mean value) can encourage the model to not rely on specific segments of data.

5) Perspective Transformations (ImageNet) to Time Warping (Time-Series):
   - Perspective changes in images can alter the apparent shape of objects.
   - Time warping in time-series data, where the time axis is distorted, can simulate different rates of activities or processes.
6) Mixup (ImageNet) to Data Blending (Time-Series):
   - Mixup creates new images by blending two or more images and their labels, leading to a smoother estimation of the model's decision boundary.
   - A similar strategy for time-series could involve blending two or more sequences together, potentially mixing different activities or sensor signals.

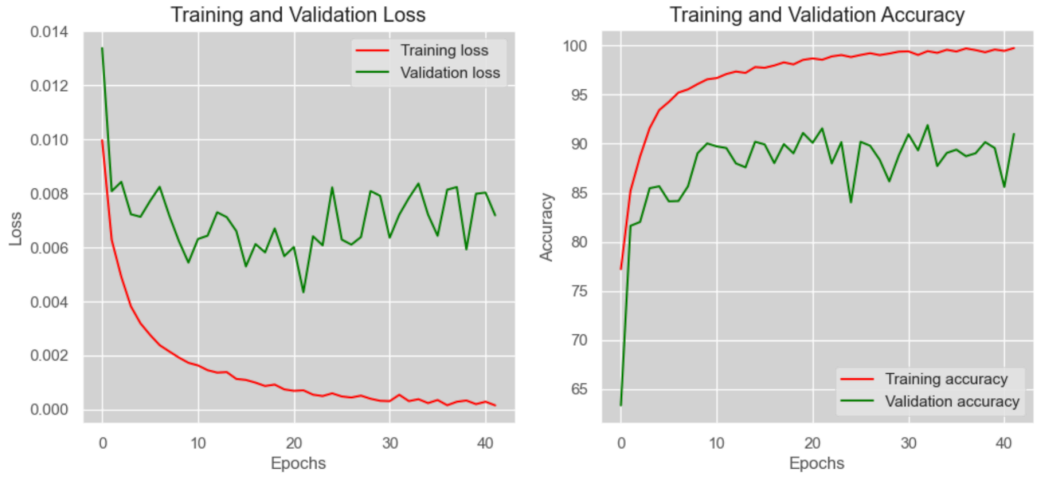
3.1.d:

Another augmentation technique we can consider is jittering, which involves adding small random perturbations to the data. This can improve the robustness of the model by simulating small variations that could occur in the real-world data due to sensor imperfections or environmental fluctuations.

```
def apply_jittering(data, jitter_strength=0.01):

    jitter = jitter_strength * (np.max(data, axis=0) - np.min(data, axis=0))

    jittered_data = data + np.random.normal(0, jitter, data.shape)

    return jittered_data
```

In this code **jitter_strength** is a parameter that determines how strong the jittering effect should be. It's multiplied by the range of the data to ensure that the added noise is proportional to the scale of the data.

```
Accuracy on dev set:  91.0
Loss on dev set:  0.007187242465826916
```



Adding jitter to the dataset didn't significantly change how well the top-performing CNN model worked — the accuracy only went up by one percent. However, by adjusting the jitter strength and other model settings, there's potential to see better improvements. The important advantage of using this approach of data augmentation is that it prevents having abrupt bumps in the loss and accuracy.

3.2 - Baseline Comparison :

3.2.a:

For classification tasks, a common baseline is random guessing. This means predicting class labels randomly according to the class distribution in the dataset.

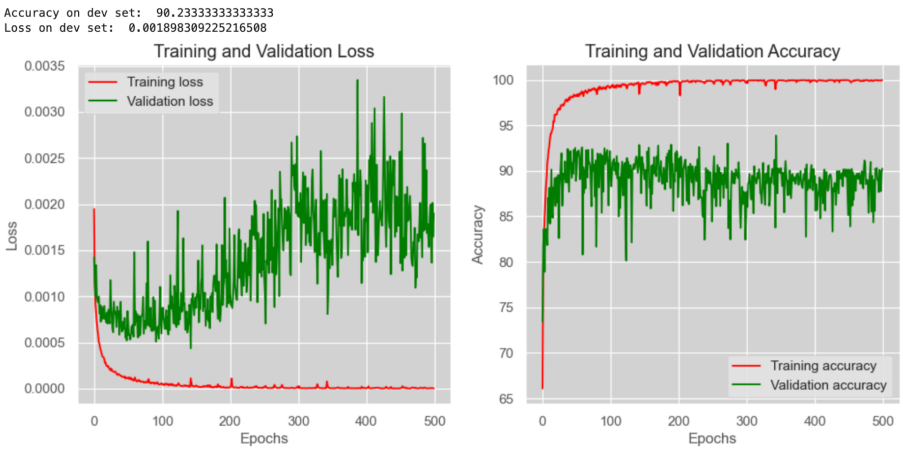
If the class distribution is imbalanced, the random baseline should be calculated based on the proportion of the majority class. For instance, if 50% of your data is labeled as 'Walking', then a naive classifier that always predicts 'Walking' would be correct 50% of the time. This would be your random baseline. So, we'll calculate the expected accuracy of random guesses weighted by the class distribution. In our imbalance dataset the **Random Baseline Accuracy: 27.21%**

3.2.b:

When measured against a random baseline accuracy of 27.21%, my models showed superior performance on the validation set. The top CNN model, which I refer to as Configuration 3, achieved an impressive 90% accuracy. Moreover, models that were enhanced with data augmentation techniques demonstrated even greater accuracy, surpassing the already high benchmark set by Configuration 3.

3.2.c:

I trained the best-performance CNN for 500 epochs. The validation accuracy quickly rises and then begins to fluctuate within a range, never reaching the training accuracy. While it doesn't exhibit a clear downward trend, the fluctuations suggest the model might be struggling to generalize to new data as effectively as it fits the training data.

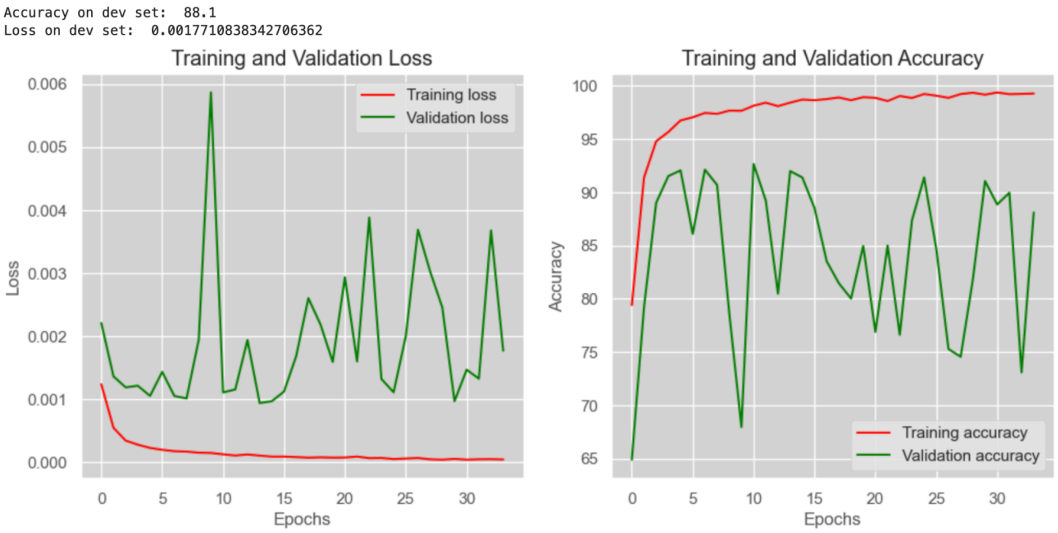


The validation loss starts lower than the training loss, possibly due to the reasons discussed previously (like regularization effects only being present during training). It then exhibits fluctuation but doesn't show a clear upward trend. While there is noise in the validation loss, the lack of a sustained increase suggests that significant overfitting may not be occurring.

In summary, while there **isn't a definitive sign of overfitting**, the model's performance on the validation set compared to the training set indicates there could be a **mild overfitting issue**.

3.3 - Batch Normalization and Layer Normalization :

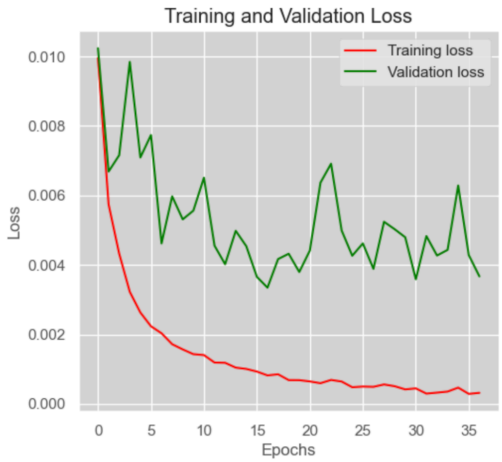
I applied batch normalization **after each convolutional layer** but before the activation function. It works by normalizing the output of the previous layer using the mean and variance computed from the current mini-batch. This can help in reducing internal covariate shift.
I applied layer normalization **after convolutional layers**. It normalizes the activations across the features instead of the batch dimension.
Results: My optimal CNN model batch size is 400. Since BatchNorm relies on the mini-batch statistics, it tends to work well with sizes of mini-batches where these statistics are reliable. So applying normalization had negative impacts on the model's performance. All these modifications are applied to the best-performance CNN

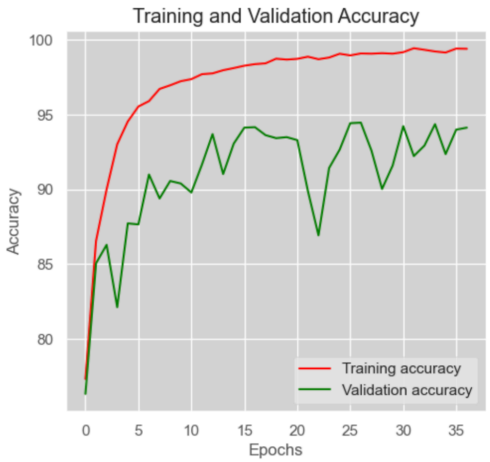


3.4 - Experimentation with Optimizers :

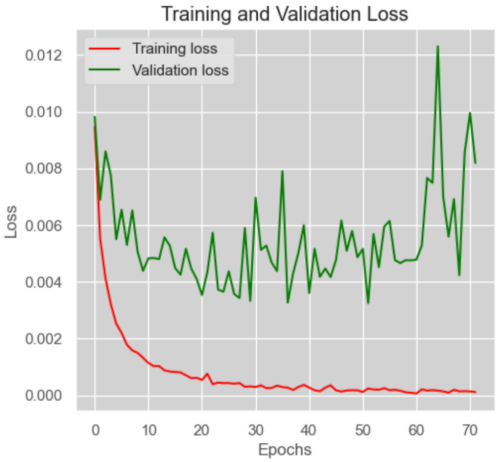
	lr = 0.1	lr = 0.01	lr = 0.001
Adam	Accuracy on dev set: 33.63333333333333	Accuracy on dev set: 79.13333333333334	Accuracy on dev set: 92.43333333333334
AdamW	34.733333333333334	Accuracy on dev set: 85.2	Accuracy on dev set: 94.13333333333334

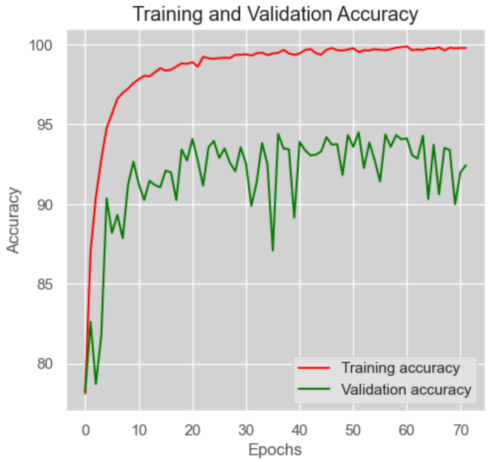
AdamW with Learning Rate 0.001:





Adam with Learning Rate 0.001:





AdamW, with its decoupled weight decay, inherently provides

some form of regularization by preventing the weights from growing too large, which can lead to a more robust model less prone to overfitting.

- A higher learning rate (0.1) caused both optimizers to overshoot minima.
- At moderate learning rates (0.01), you may see more stable and efficient convergence in both Adam and AdamW, but AdamW handle this better due to improved generalization from the decoupled weight decay.
- A lower learning rate (0.001) require more epochs to converge but could lead to finer convergence in both optimizers. The impact of weight decay becomes more pronounced with AdamW, potentially leading to better generalization.

3.5 - Model Scaling :

3.5.a) Reduce the Number of Filters in Convolutional Layers :

Since my optimal CNN model doesn't have four layers I assume the purpose of this part is the base CNN model.

Reducing the number of filters in your CNN's convolutional layers decreases the model's complexity, potentially reducing overfitting and improving generalization if the original model is too large. In our case it helped the performance to enhance.

3.5.b) Simplify or Remove Regularization :

Again I applied requested alternations on base CNN model.

With Dropout at 0.3: By reducing the dropout rate from 0.5 to 0.3, we're allowing more neurons to remain active during training. This can help in retaining more information

and may counteract potential underfitting caused by the reduced model complexity. It lead to better performance, as the model can learn a richer representation of the data.

```
Accuracy on dev set:  90.0
Loss on dev set:  0.0017515484636339048
```

Without Dropout: Removing dropout entirely will mean that all neurons contribute to the learning process during every forward and backward pass. This is beneficial if the model's capacity has been significantly reduced since you need all available parameters to capture the nuances in the data. However, it could also increase the risk of overfitting since dropout is a regularization technique.

```
Accuracy on dev set:  87.53333333333333
Loss on dev set:  0.0017515484636339048
```

3.5.c) Analyzing:

In the graph where dropout was adjusted to 0.3, the training loss remains steadily low, while the validation loss shows variability but does not trend upwards, indicating that

the model is not overfitting significantly. The training accuracy plateaus near 100%, and validation accuracy shows variability, which is typical due to dropout providing regularization.

Without dropout, both training and validation accuracy remaining high, which could mean better learning from the full capacity of the model. However, this configuration carry a higher risk of overfitting, as dropout is not present to randomly disable connections during training.

After reducing the number of filters, the training loss decreases smoothly, suggesting efficient learning, and the validation loss shows fluctuations within a range without a clear increasing trend, hinting that overfitting is not severe. The training and validation accuracy are both high, indicating good model performance.

Downscaling the model likely made it more computationally efficient, as indicated by the smooth and quick convergence of the training loss. Less computational resources would be required due to fewer parameters.

3.5.d) Data Scaling:

Downsampling to 10Hz: We originally sample at 20Hz and select every 2nd sample, we're reducing the dataset frequency by half. This means the number of time points in each sequence will be halved as well, effectively reducing the dataset size by 50%.

Downsampling to 4Hz: Similarly, we're starting at 20Hz and take every 5th sample to achieve a 4Hz rate, we're reducing the dataset frequency by a factor of five. In this case, we're left with 20% of the original data points in each sequence, cutting the dataset size to one-fifth.

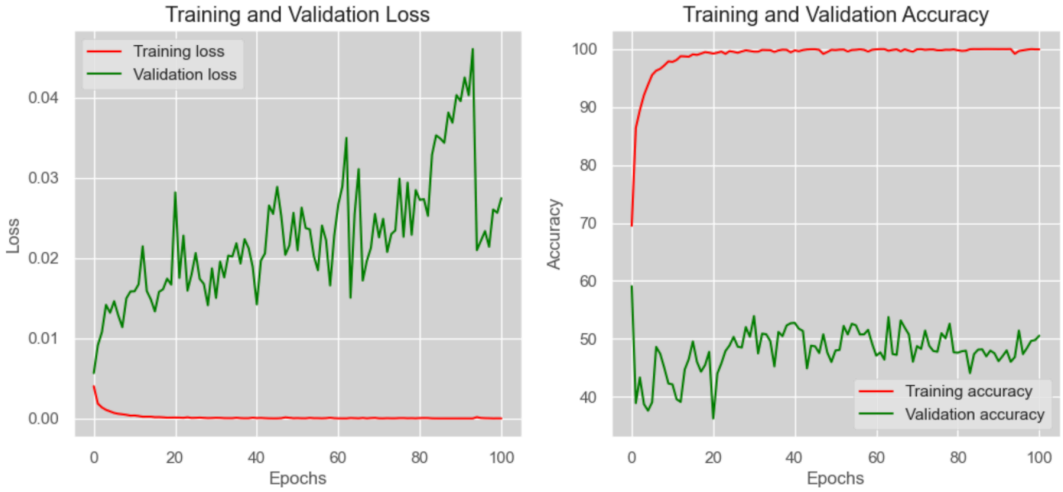
When we reduce the sampling rate of your time-series data by selecting every 2nd sample or every 5th sample, we're effectively reducing the dataset's temporal resolution. This means that each input sequence to our CNN will have fewer data points. If the sequence becomes too short, it can no longer accommodate the convolution operation with the specified kernel size.

To resolve this issue, we would need to do one or more of the following:

- Reduce Kernel Size
- Use Padding
- Change Stride

10Hz:
I reduced the kernel size to 5:

```
Accuracy on dev set:  50.558240293284456
Loss on dev set:  0.02745708188811344
```



4Hz:
I reduced the kernel size to 2:

```
Accuracy on dev set:  46.184605131622796
Loss on dev set:  0.11860444705592796
```

Observing the performance at both downsampling rates, there was a significant drop. When you reduce the sampling rate of your dataset, there are several reasons why the performance of your model might worsen:

- Loss of Information: Certain temporal patterns or features that are crucial for classification might no longer be captured at lower sampling rates.

- Temporal Resolution: A lower sampling rate can lead to a lower temporal resolution.

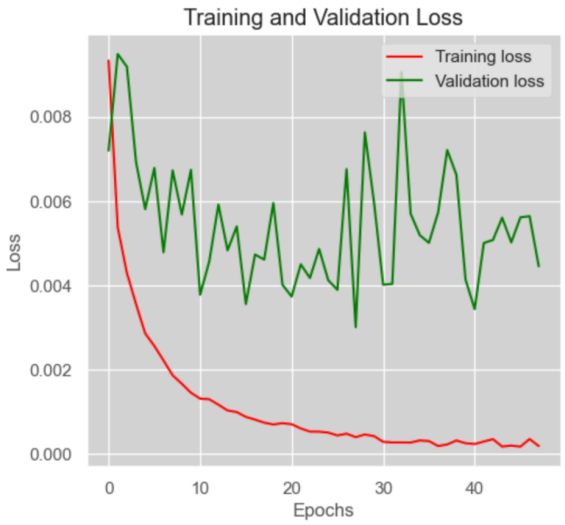
3.6 Sampling Strategies :

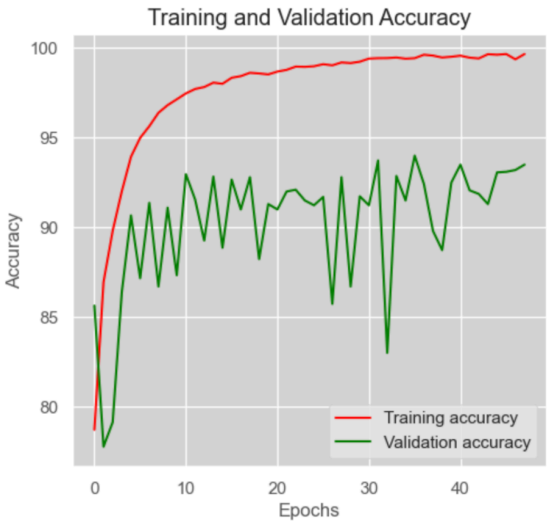
3.6.a) Random Sampling:

I applied these modifications to my CNN best performance model.
Setting the “shuffle” argument to “True” in “DataLoader” is a way to implement random sampling. When “shuffle=True”, the data points in dataset are randomly shuffled before being split into batches. This means that each time we iterate over our train_loader, the data points in each batch will be different.

With Random Sampling On:

```
Accuracy on dev set:  93.5
Loss on dev set:  0.004456826882512428
```



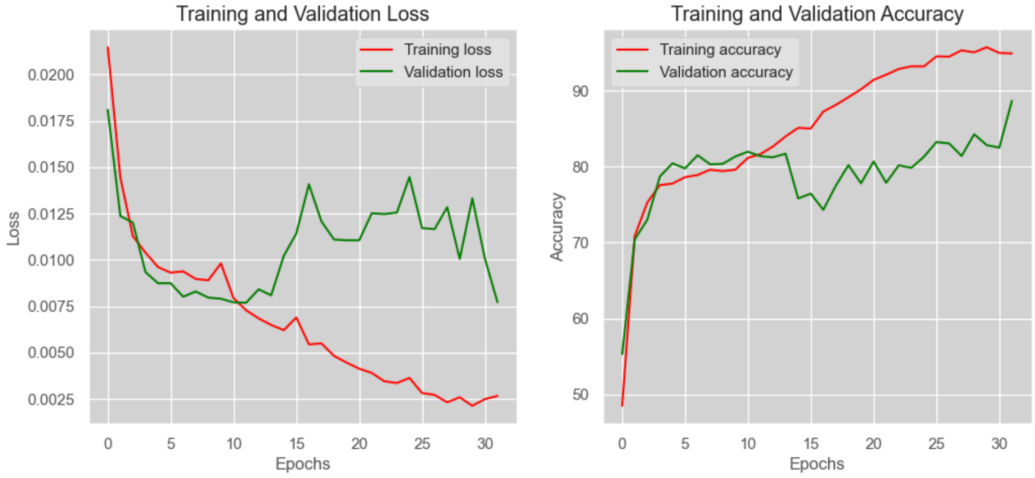


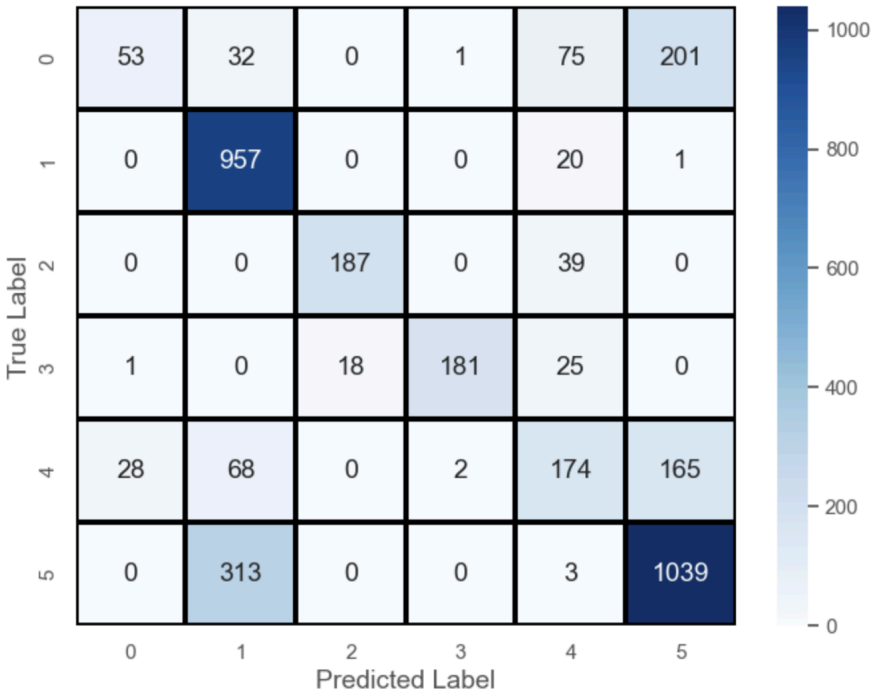
	precision	recall	f1-score	support
0	0.86	0.57	0.68	362
1	0.71	0.99	0.83	978
2	0.85	0.83	0.84	226
3	0.99	0.84	0.91	225
4	0.78	0.60	0.68	437
5	0.84	0.77	0.80	1355
accuracy			0.80	3583
macro avg	0.84	0.77	0.79	3583
weighted avg	0.81	0.80	0.79	3583

Without Random Sampling:

I Set the “shuffle” argument to “False” in “DataLoader”

```
Accuracy on dev set:  88.7
Loss on dev set:  0.007721815628734523
```





	precision	recall	f1-score	support
0	0.65	0.15	0.24	362
1	0.70	0.98	0.82	978
2	0.91	0.83	0.87	226
3	0.98	0.80	0.89	225
4	0.52	0.40	0.45	437
5	0.74	0.77	0.75	1355
accuracy			0.72	3583
macro avg	0.75	0.65	0.67	3583
weighted avg	0.72	0.72	0.70	3583

We can observe that the model's ability to correctly identify the 'Downstairs' and 'Upstairs' activities declined in the version without random sampling.

Honestly, I cannot justify why this happened. This can be because of the imbalance dataset. And since 'Downstairs' and 'Upstairs' are underrepresented in the training data, the model may not learn to recognize these activities as effectively as others.

3.6.b) Oversampling Minor Classes

I addressed the imbalanced dataset issue by implementing oversampling to ensure that all classes have an equal number of samples. After applying oversampling, here are the updated counts for each activity in the dataset:

- Walking: 424,399 samples

- Jogging: 424,399 samples
- Upstairs: 424,399 samples
- Downstairs: 424,399 samples
- Sitting: 424,399 samples
- Standing: 424,399 samples

This equal cardinality across classes should help in training a more balanced and fair model.

```
Accuracy on dev set:  62.621490280777536
Loss on dev set:  0.06074075650129543
```

	precision	recall	f1-score	support
0	0.40	0.02	0.03	1278
1	0.54	0.95	0.69	835
2	1.00	1.00	1.00	1076
3	1.00	0.00	0.00	1596
4	0.32	0.85	0.46	1185
5	0.18	0.22	0.20	1043
accuracy			0.45	7013
macro avg	0.57	0.51	0.40	7013
weighted avg	0.60	0.45	0.35	7013

However, after implementing oversampling to equalize the class distribution, a decrease in model accuracy is observed. It would have occured due to several reasons:

- oversampling was done by simply replicating the minority class samples until they match the majority class, the model might overfit to the replicated samples. This could make the model less able to generalize to new, unseen data.

- With the increase in data, the model complexity might need to be adjusted. A model that was well-tuned for a smaller dataset might not perform optimally on the larger, oversampled dataset.
- Oversampling could make the class boundaries less distinct. The model may become less certain about these boundaries, leading to poor performance on validation or test sets.

3.6.c)

In addition to the previously mentioned sampling techniques, I will further explore and analyze alternative methods for handling imbalanced datasets:

Synthetic Sample Generation (SMOTE or ADASYN):

- Creates synthetic samples that are interpolations of the minority class data points, adding more diversity than simple replication.
- The synthetic nature of the data can introduce noise or artificial patterns that do not exist in the real data distribution.

Cluster-Based Over-Sampling:

- Involves creating clusters within the minority class and generating samples within those clusters to preserve data structures.
- Balances the data while potentially maintaining the integrity of minority class characteristics.