Assignment 1 - Programming
Human Activity Recognition using MLP and CNN

IFT6135-H2024, Section B
Prof : Aaron Courville

Student: Mohammadhossein Malekpour

Table of Content:

Question 1

I ran all code and experiments on my local machine. PyTorch uses the new Metal Performance Shaders (MPS) backend for GPU training acceleration on Mac with Apple silicon; so I changed the backend from CPU/CUDA to **MPS** to enable GPU training.
Separating the validation set from the train and test sets helps in fine-tuning model hyperparameters, avoiding overfitting, and ensuring unbiased evaluation. It provides a basis for model selection and enables early stopping to prevent overfitting. This separation is crucial for building reliable and generalizable machine learning models; so I create a validation set as follows:

```
df_train = df[df['user'] <= 28]
df_validate = df[df['user'].isin([31,30,29])]
df_test = df[df['user'] > 32]
```

Question 1.1

Base MLP model architecture:

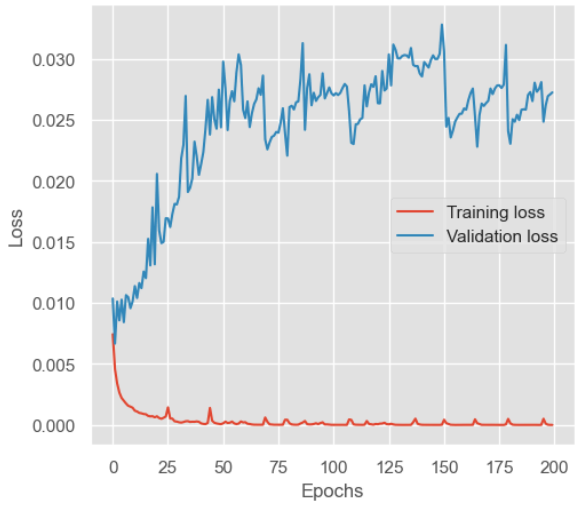
```
MLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (fc4): Linear(in_features=100, out_features=6, bias=True)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

Config:

BATCH_SIZE = 64
EPOCHS = 200
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 80.03
Loss on val data: 0.02



The plot illustrates the training dynamics of a multi-layer perceptron neural network over 200 epochs. On the left, the training loss decreases sharply and then plateaus, indicating the model is learning and then stabilizing. The validation loss increases and starts to fluctuate after around 10 epochs, which might suggest the beginning of overfitting as the model learns noise from the training data. The right plot shows the training accuracy rapidly increasing and plateauing near 100%, which also indicates potential overfitting since the model performs almost perfectly on the training data. In contrast, the validation accuracy increases but plateaus around 80%, which is substantiated by the reported 80.03% accuracy on validation data. Despite the simplicity, the model achieves a reasonable accuracy on the validation data but might benefit from regularization or other techniques to reduce overfitting as suggested by the divergence between training and validation performance.

Question 1.2

Configuration 1: Increase the Number of Hidden Layers and Use Sigmoid Activation

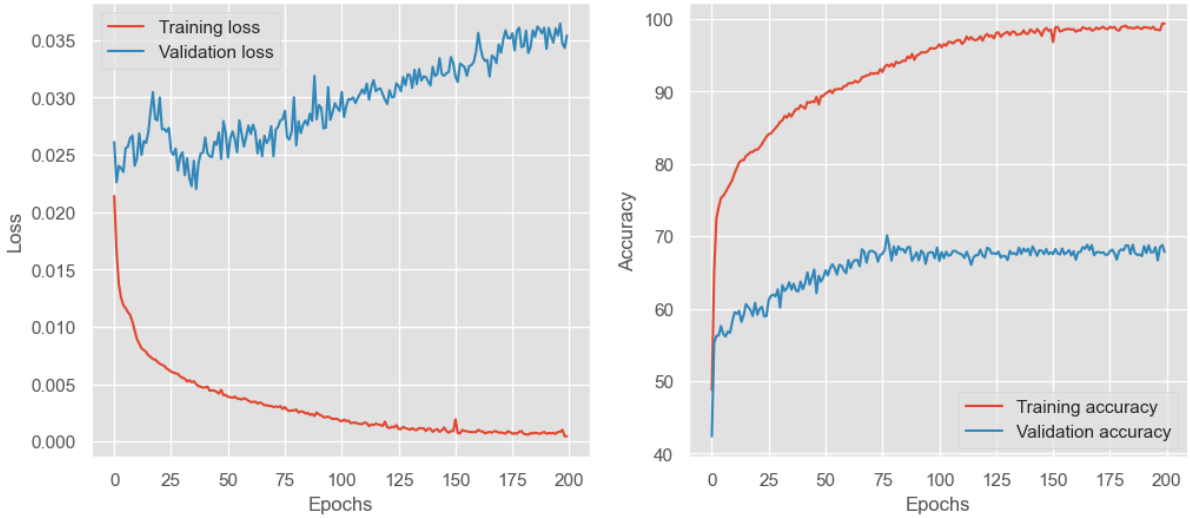
```
MLP_Config1(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=150, bias=True)
  (fc2): Linear(in_features=150, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (fc4): Linear(in_features=100, out_features=100, bias=True)
  (fc5): Linear(in_features=100, out_features=100, bias=True)
  (fc6): Linear(in_features=100, out_features=6, bias=True)
)
```

Config:

BATCH_SIZE = 64
EPOCHS = 200
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 67.829
Loss on val data: 0.035



The updated MLP model with additional hidden layers and sigmoid activation functions demonstrates a decrease in performance compared to the base model. The validation loss shows more fluctuation, suggesting the model is overfitting and not generalizing well. Validation accuracy settles around 67-70%, a significant drop from the previous 80.03%. The use of sigmoid activations and a deeper network likely contributes to this reduced performance, as indicated by the final validation accuracy of 67.829% and a higher validation loss of 0.035. This implies that the modifications in the network architecture did not improve the model and might be counterproductive for this specific dataset and task.

Configuration 2: Reduce the Number of Hidden Layers, Use Tanh Activation, Increase Units in Hidden Layers, and Adjust Batch Size

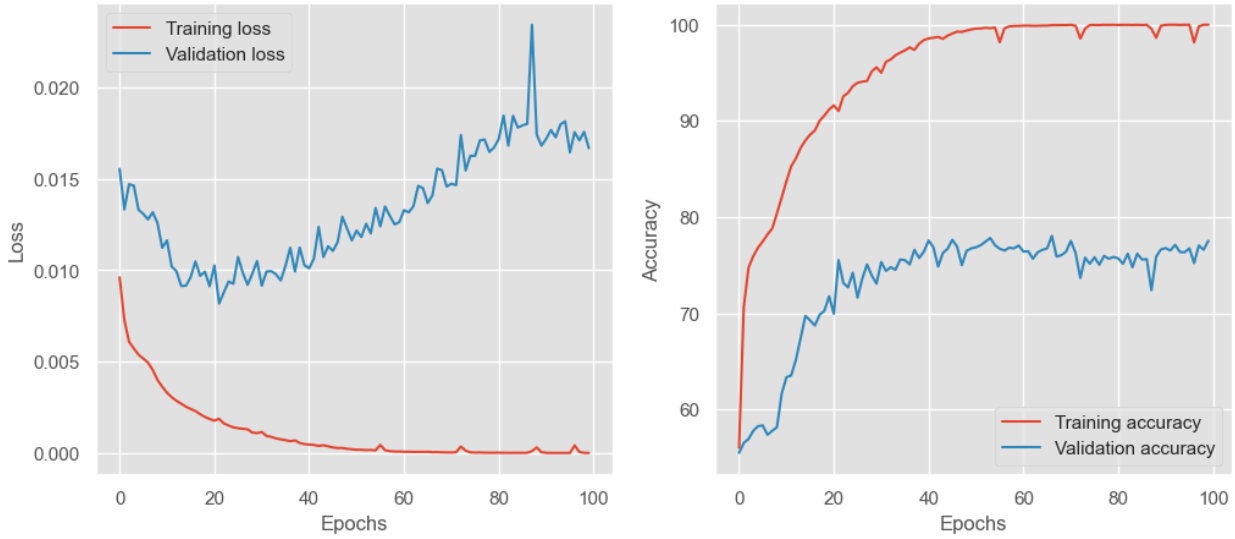
```
MLP_Config2(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=200, bias=True)
  (fc2): Linear(in_features=200, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=6, bias=True))
```

Config:

BATCH_SIZE = 128
EPOCHS = 100
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 77.51
Loss on val data: 0.016



The updated MLP model with fewer hidden layers, tanh activation, and more neurons shows better validation performance than Configuration 1 but still underperforms the base model. Training loss remains low, suggesting good fit to training data, while validation loss begins to rise after 20 epochs, hinting at some overfitting. Validation accuracy plateaus at approximately 77.5%, higher than Configuration 1's 67.829% but lower than the base model's 80.03%. This suggests tanh activations helped model's validation loss curve and it decreases

Configuration 3: Use ReLU Activation، Increase Batch Size and Increase Learning Rate

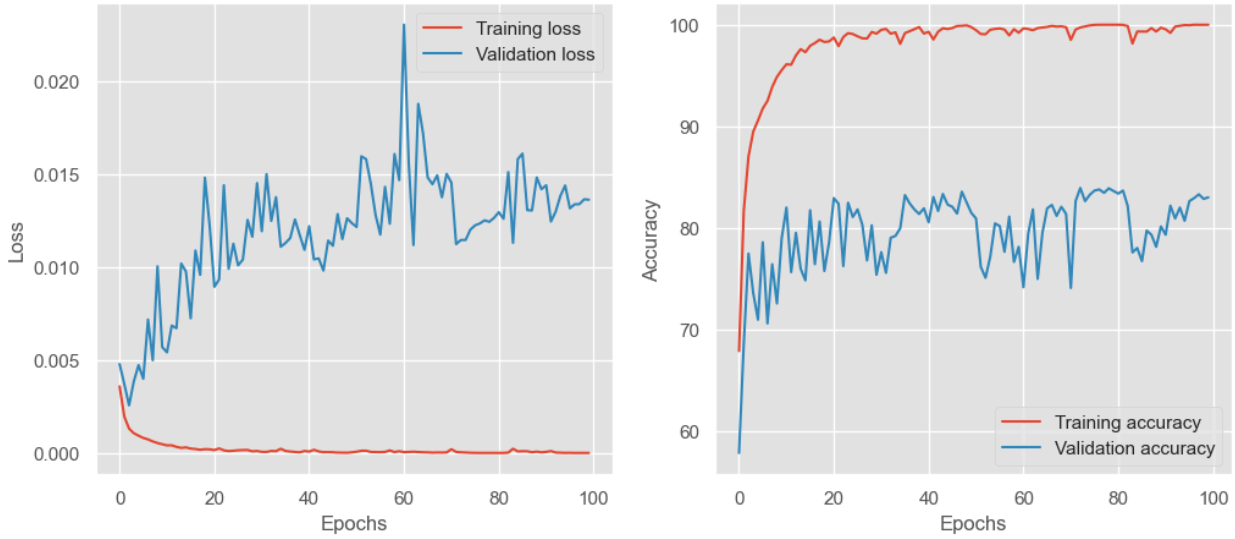
```
MLP_Config3(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=240, out_features=150, bias=True)
  (fc2): Linear(in_features=150, out_features=150, bias=True)
  (fc3): Linear(in_features=150, out_features=100, bias=True)
  (fc4): Linear(in_features=100, out_features=6, bias=True)
)
```

Config:

BATCH_SIZE = 256
EPOCHS = 100
Optimizer = Adam
Learning Rate = 0.003

Result:

Accuracy on val data: 83.005
Loss on val data: 0.0136



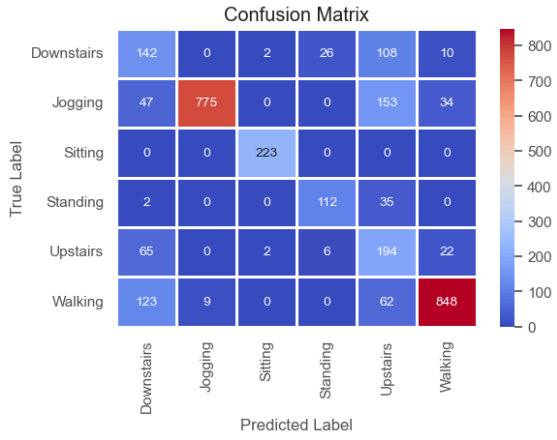
The updated Configuration 3 of the MLP model, incorporating ReLU activations and a higher number of units in hidden layers, demonstrates enhanced performance. The training loss declines steadily and remains low, indicative of effective learning, while the validation loss, despite fluctuations, suggests a better generalization compared to earlier configurations. The training accuracy quickly reaches a plateau near 100%, a characteristic benefit of ReLU activations in preventing vanishing gradients. The validation accuracy shows a significant improvement, stabilizing around 83%, which is a marked increase from previous configurations. The combination of larger batch sizes, increased learning rate, and ReLU activation contributes to this improved performance, culminating in a final validation accuracy of 83.005% and a lower loss of 0.0136, making this the most successful model iteration among those presented.

Question 1.3 & 1.4

classification report and confusion matrix for the best-performing configuration (Config 3) on the test set:

<Figure size 600x400 with 2 Axes>

	precision	recall	f1-score	support
0	0.37	0.49	0.43	288
1	0.99	0.77	0.86	1009
2	0.98	1.00	0.99	223
3	0.78	0.75	0.76	149
4	0.35	0.67	0.46	289
5	0.93	0.81	0.87	1042
accuracy			0.76	3000
macro avg	0.73	0.75	0.73	3000
weighted avg	0.84	0.76	0.79	3000



From the confusion matrix, we can make the following observations:
1. **Sitting** has the highest true positive rate, with 223 out of 223 (a perfect recall score), indicating that the model has no difficulty distinguishing this activity.
2. **Jogging** also shows strong performance with 775 out of 1009 correctly predicted, but there seems to be a notable number of instances (153 + 34) where Jogging is confused with Walking.
3. **Walking** has a relatively high number of false negatives, where 123 instances are predicted as Downstairs, and 62 as Upstairs, which suggests the model struggles to differentiate between these activities.
4. **Upstairs** and **Downstairs** activities have considerable confusion with each other and with Walking, as indicated by the high off-diagonal numbers in these rows and columns.
5. **Standing** shows a decent recall, but there are some confusions with Upstairs.

- The **precision** for Sitting and Jogging is particularly high, indicating that when the model predicts these classes, it is very often correct.
- The **recall** for Sitting is perfect, and for Jogging and Walking, it's quite good, but for Downstairs and Upstairs, the recall is much lower, indicating many false negatives.
- The **f1-score**, which is a harmonic mean of precision and recall, is also high for Sitting, Jogging, and Walking, reflecting the balanced precision and recall for these classes. However, it's lower for Downstairs and Upstairs due to lower recall.
- The **support** indicates the number of true instances for each class in the dataset. The class distribution is imbalanced, with more samples for Jogging and Walking, which might be influencing the model's performance.
- The overall **accuracy** of the model is 76%, which is quite good but suggests there is room for improvement, especially in distinguishing between similar activities like Walking, Upstairs, and Downstairs.

Given this information, the model appears to perform best on activities that have distinct movement patterns, such as Sitting, and it struggles with activities involving walking or steps, which can be similar in accelerometer data. The model may benefit from additional feature engineering, data augmentation techniques to balance the class distribution, or perhaps a more sophisticated model architecture that can better capture the nuances between these similar activities.

Question 2

Question 2.1

Here's the implementation of CNN class:

```
class CNN(nn.Module):
  def __init__(self, time_periods, n_sensors, n_classes):
    super(CNN, self).__init__()
    self.n_sensors = n_sensors
    self.time_periods = time_periods
    self.n_classes = n_classes
    self.conv1 = nn.Conv1d(n_sensors, 100, kernel_size=10)
    self.conv2 = nn.Conv1d(100, 100, kernel_size=10)
    self.conv3 = nn.Conv1d(100, 160, kernel_size=10)
    self.conv4 = nn.Conv1d(160, 160, kernel_size=10)
    self.maxpool = nn.MaxPool1d(3)
    self.adaptive_pool = nn.AdaptiveAvgPool1d(1)
    self.dropout = nn.Dropout(0.5)
    self.fc = nn.Linear(160, n_classes)

  def forward(self, x):
    x = x.view(-1, self.n_sensors, self.time_periods)
    x = F.relu(self.conv1(x))
    x = F.relu(self.conv2(x))
    x = self.maxpool(x)
    x = F.relu(self.conv3(x))
    x = F.relu(self.conv4(x))
    x = self.adaptive_pool(x)
    x = self.dropout(x)
    x = x.view(-1, 160)
    x = self.fc(x)
    return F.log_softmax(x, dim=1)

  def predict(self, x):
    self.eval()  # Set the model to evaluation mode
    with torch.no_grad():
      logits = self.forward(x)
      predictions = torch.argmax(logits, dim=1)
    return predictions
```

Also I used early stopping as a regularization technique to prevent overfitting. This approach allows the training to halt when the validation loss stops decreasing, ensuring that the model generalizes well to unseen data while saving computational resources.

Base MLP model architecture:

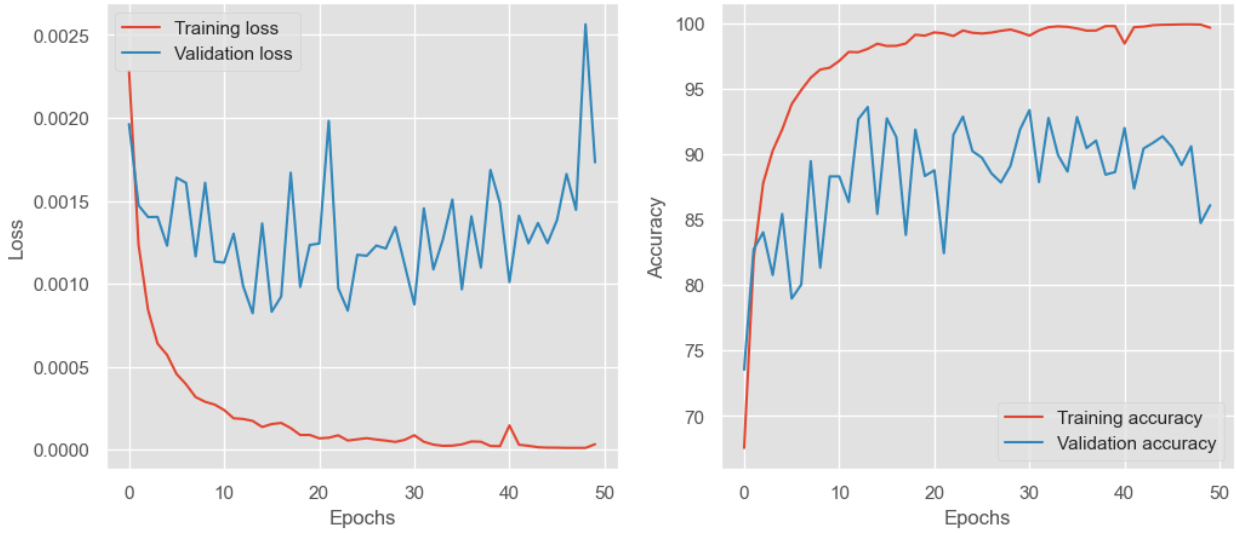
```
CNN(
  (conv1): Conv1d(3, 100, kernel_size=(10,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(10,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(10,), stride=(1,))
  (conv4): Conv1d(160, 160, kernel_size=(10,), stride=(1,))
  (maxpool): MaxPool1d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=160, out_features=6, bias=True)
)
```

Config:

BATCH_SIZE = 64
EPOCHS = 500
patience = 100 (belongs to early stopping)
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 86.10 (Early Stopping occurred on the 110th epoch)
Loss on val data: 0.0017



The Base CNN model's training chart shows it learned well from the data, with the training loss getting very low and training accuracy getting very high. The validation loss goes up and down, which means the model isn't as steady when guessing on new data it hasn't seen before. Despite this, the model does better than the previous MLP model, with a final validation accuracy of 86.10%, which is pretty good. Early stopping kicked in at the 110th epoch to prevent the model from learning the training data by heart, which

can make it worse at predicting new data. The final loss on validation data was really low, at 0.0017, showing the model's predictions were quite close to the actual results.

Question 2.2

Configuration 1: Reduced Number of Convolutional Layers and Adjust Channels, Kernel Size

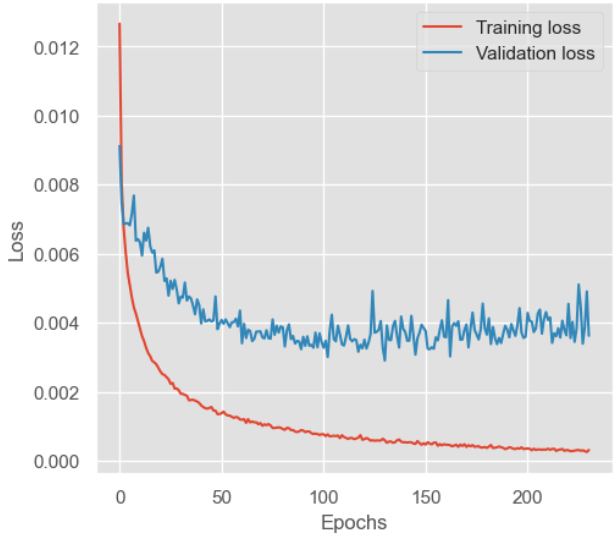
```
CNN_Config1(
  (conv1): Conv1d(3, 64, kernel_size=(5,), stride=(1,))
  (conv2): Conv1d(64, 128, kernel_size=(5,), stride=(1,))
  (maxpool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=128, out_features=6, bias=True)
)
```

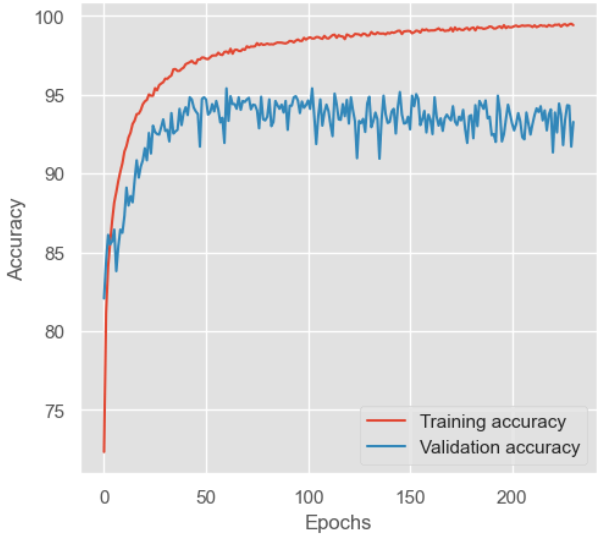
Config:

BATCH_SIZE = 64
EPOCHS = 500
patience = 100 (belongs to early stopping)
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 93.26 (Early Stopping occurred on the 231st epoch)
Loss on val data: 0.0036





The new CNN model with fewer convolutional layers shows a smoother learning curve, indicating a better generalization performance. The training loss drops quickly and stabilizes at a low level, suggesting the model is efficiently learning from the training data. Similarly, the validation loss decreases and maintains a relatively steady trend with less fluctuation than before, which is a good sign that the model isn't overfitting as much. The accuracy plot also reflects this, with the training accuracy swiftly reaching a high plateau and the validation accuracy rising to a high of over 90% and remaining fairly stable. This

stability is an improvement over the previous model and suggests that reducing the number of convolutional layers has made the model less complex and more capable of generalizing to new data. The model's final validation accuracy of 93.26% and loss of 0.0036, achieved at the 231st epoch where early stopping was triggered, demonstrating a significant improvement in performance compared to the previous architecture.

Configuration 2: Used three Convolutional Layers, Average Pooling and Leaky Relu Activation Functions

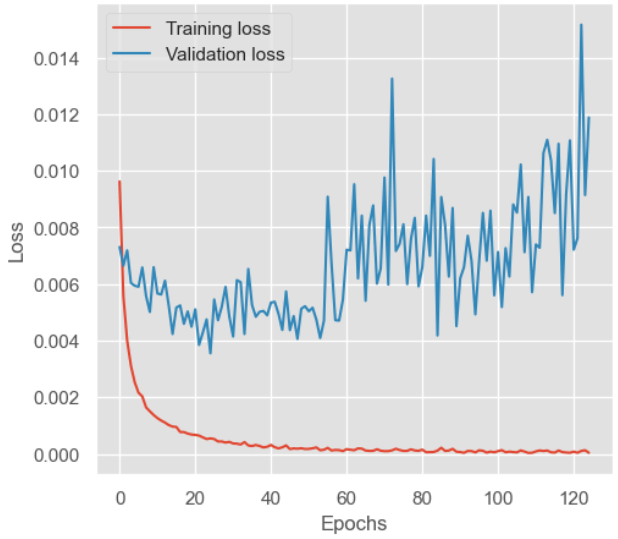
```
CNN_Config2(
  (conv1): Conv1d(3, 100, kernel_size=(6,), stride=(1,))
  (conv2): Conv1d(100, 100, kernel_size=(6,), stride=(1,))
  (conv3): Conv1d(100, 160, kernel_size=(6,), stride=(1,))
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=160, out_features=6, bias=True)
)
```

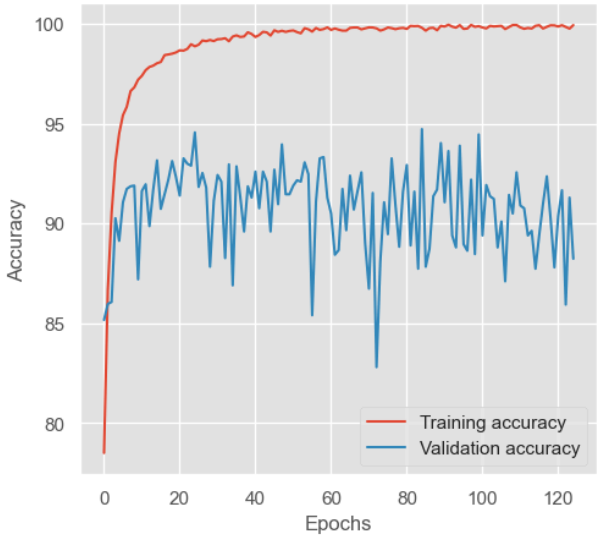
Config:

BATCH_SIZE = 64
EPOCHS = 500
patience = 100 (belongs to early stopping)
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 88.23 (Early Stopping occurred on the 125th epoch)
Loss on val data: 0.011





CNN Configuration 2 shows a model that learns well initially, as indicated by the low training loss. However, the validation loss is quite erratic, suggesting the model might be overfitting and not performing as consistently on data it hasn't seen before. The training accuracy is very high, but the validation accuracy varies a lot, which again points to potential overfitting issues. With a final validation accuracy of 88.23%, this model performs worse than the previous Configuration 1, which had a higher

validation accuracy of 93.26%. The use of average pooling and Leaky ReLU didn't lead to an improved model in this case. It seems these changes made the model less stable, resulting in lower performance on the validation set.

Configuration 3: Different Pooling, Increased Dropout, Increased Batch Size

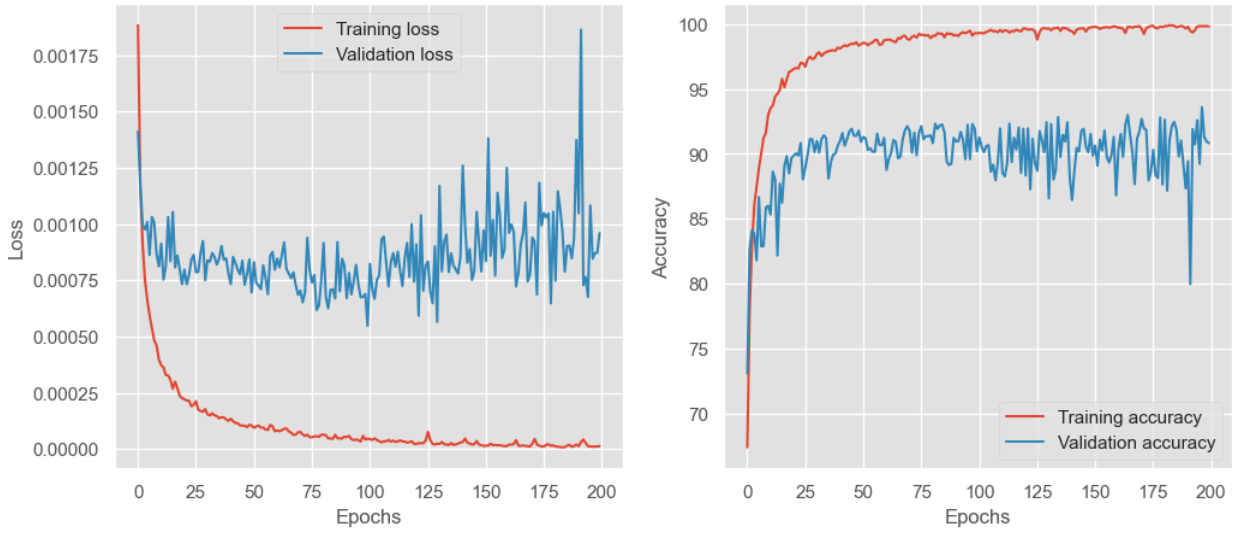
```
CNN_Config3(
  (conv1): Conv1d(3, 200, kernel_size=(10,), stride=(1,))
  (conv2): Conv1d(200, 120, kernel_size=(10,), stride=(1,))
  (conv3): Conv1d(120, 160, kernel_size=(10,), stride=(1,))
  (avgpool): AvgPool1d(kernel_size=(3,), stride=(3,), padding=(0,))
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.6, inplace=False)
  (fc): Linear(in_features=160, out_features=6, bias=True)
)
```

Config:

BATCH_SIZE = 512
EPOCHS = 500
patience = 100 (belongs to early stopping)
Optimizer = Adam
Learning Rate = 0.001

Result:

Accuracy on val data: 88.23 (Early Stopping occurred on the 200th epoch)
Loss on val data: 0.011

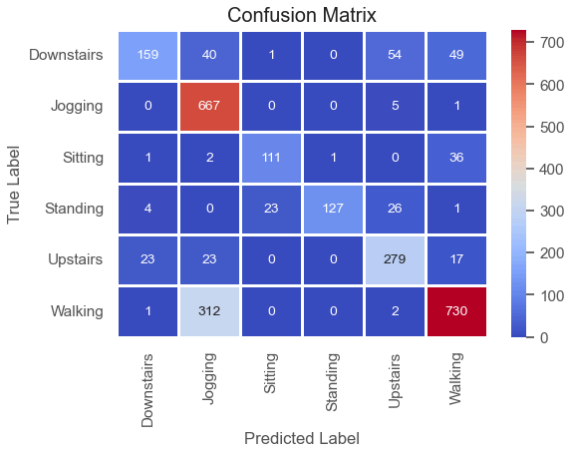


Configuration 3's adjustments to the CNN architecture result in a stable training loss and high training accuracy, but validation accuracy and loss show more variation and do not significantly improve, with a validation accuracy of 88.23%. These changes, including altered pooling and increased dropout, do not markedly enhance performance compared to previous settings.

Question 2.3

classification report and confusion matrix for the best-performing CNN (Config 1) on the test set:

	precision	recall	f1-score	support
0	0.85	0.52	0.65	303
1	0.64	0.99	0.78	673
2	0.82	0.74	0.78	151
3	0.99	0.70	0.82	181
4	0.76	0.82	0.79	342
5	0.88	0.70	0.78	1045
accuracy			0.77	2695
macro avg	0.82	0.74	0.76	2695
weighted avg	0.80	0.77	0.77	2695



The confusion matrix for the classification model shows
better performance, with high values on the diagonal, indicating accurate predictions across all activity classes. The precision and recall across all activities are almost good, leading to good f1-scores (except "Downstairs"). This reflects the model's robustness in correctly identifying each activity type, resulting in an overall accuracy of 77%. CNNs performed over MLPs due to their ability to capture local temporal patterns through convolutional filters and shared weights. They learn hierarchical features and offer robustness to variations with pooling layers, and their translation invariance allows them to recognize patterns regardless of their position in time. This makes CNNs particularly adept at identifying the distinct rhythms and motions characteristic of different activities, as reflected in the loss curve and classification report.

Question 3

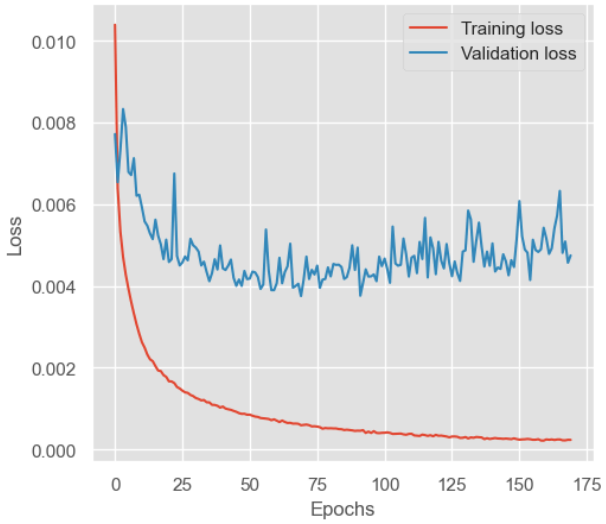
Question 3.1

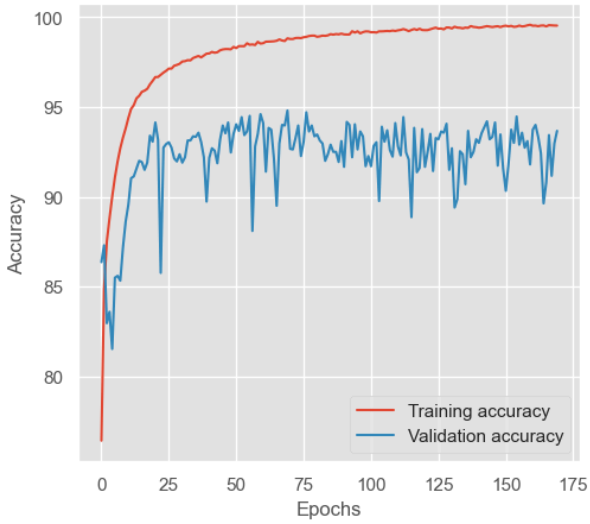
Question 3.1.a

```
def add_noise(data, noise_level=0.01):
    noise = np.random.normal(0, noise_level, data.shape)
    return data + noise
```

How it works: Gaussian noise (also known as white noise) is commonly added to the data. This noise follows a normal distribution with a mean of zero and a specified standard deviation (noise level). By adding this noise to your original data, you create slightly perturbed versions of your samples.

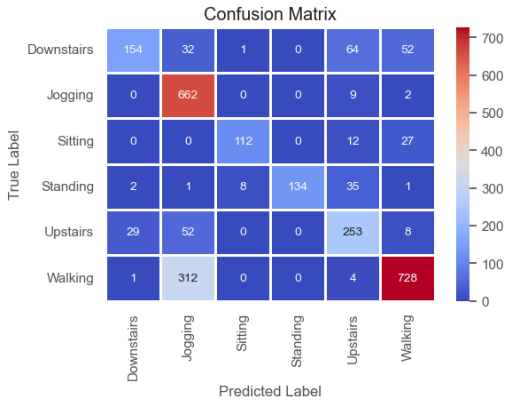
Performance of best model (CNN Config 1) after applying noise addition to training data:
Validation Set:
Accuracy on val data: 92.9
Loss on val data: 0.0051





Test Set:

	precision	recall	f1-score	support
0	0.83	0.51	0.63	303
1	0.63	0.98	0.76	673
2	0.93	0.74	0.82	151
3	1.00	0.74	0.85	181
4	0.67	0.74	0.70	342
5	0.89	0.70	0.78	1045
accuracy			0.76	2695
macro avg	0.82	0.74	0.76	2695
weighted avg	0.80	0.76	0.76	2695



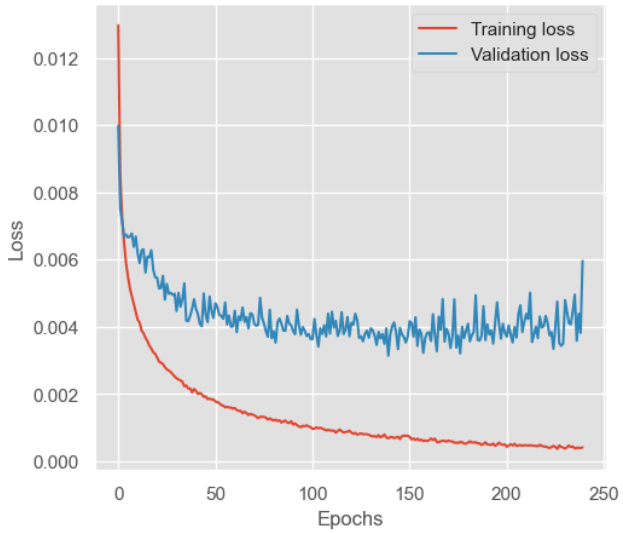
Adding Gaussian noise slightly decreased the overall accuracy and the recall for several classes, notably for the 'Downstairs' and 'Walking' activities. This decrease can occur because noise can make the activity patterns less distinct, thus harder for the model to classify accurately. Generally, adding noise is expected to test the robustness of the model against real-world data imperfections, with a potential slight decline in performance due to increased difficulty in identifying the underlying patterns in the data.

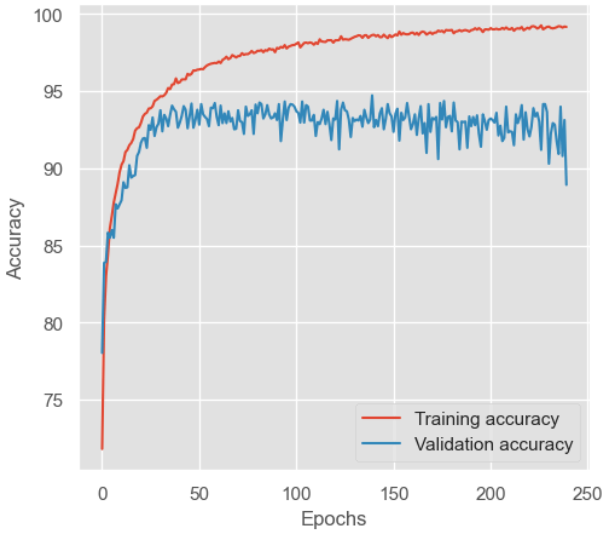
Question 3.1.b

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

Time shifting involves moving the data points forward or backward in time. This can be done by rolling the data array along the time axis. For example, a positive shift moves the data points forward in time, while a negative shift moves them backward.

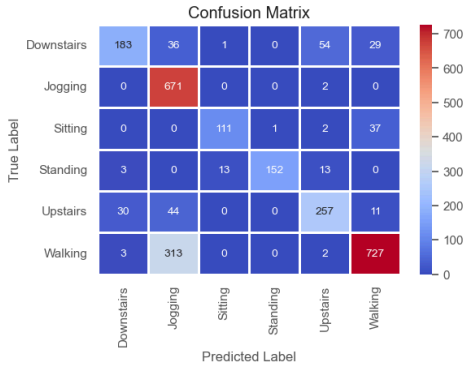
Performance of best model (CNN Config 1) after applying time shifting to training data:
Validation Set:
Accuracy on val data: 88.93
Loss on val data: 0.0059





Test Set:

	precision	recall	f1-score	support
0	0.84	0.60	0.70	303
1	0.63	1.00	0.77	673
2	0.89	0.74	0.80	151
3	0.99	0.84	0.91	181
4	0.78	0.75	0.76	342
5	0.90	0.70	0.79	1045
accuracy			0.78	2695
macro avg	0.84	0.77	0.79	2695
weighted avg	0.82	0.78	0.78	2695



Time shifting improved precision in several classes but had mixed effects on recall, with notable improvements in 'Downstairs' but reductions in 'Walking'. This suggests that while the model may have become better at confidently predicting some activities, the shifting made it harder to capture all instances of others. Time shifting tests the model's resilience to temporal misalignments in the data, and typically, you would expect some robustness against minor shifts, but a decrease in performance as shifts become larger and patterns get distorted.

Question 3.1.c

Here are some augmentation techniques used in image contexts and their potential applicability to time-series data:
1. Rotation and Flipping (ImageNet): In image data, rotation and flipping are common augmentation techniques. For time-series data, an analogous technique could be to invert the signal (similar to flipping) or to rotate the axes if the data is multidimensional (e.g., 3D accelerometer data).

2. Scaling and Zooming (ImageNet): In image data, scaling and zooming can change the size of objects within the image. For time-series data, this could translate to amplitude scaling (changing the magnitude of the signal) or time scaling (stretching or compressing the signal in time).

3. Cropping (ImageNet): Cropping parts of images is a common augmentation technique. For time-series data, this could mean segmenting the data into shorter subsequences, which can be particularly useful if the events of interest are localized in time.

4. Noise Injection (ImageNet): Adding random noise to images is a way to make models more robust. In time-series data, adding random noise or jitter to the signal can help the model learn to ignore minor fluctuations and focus on the overall pattern.

5. Time Warping: While not directly from ImageNet, time warping is a technique specific to time-series data where the time axis is distorted, simulating variations in the speed of the underlying events.

6. Window Slicing: Similar to cropping in images, window slicing involves taking different time windows of the data. This can help the model learn from different parts of the signal and improve its ability to generalize.

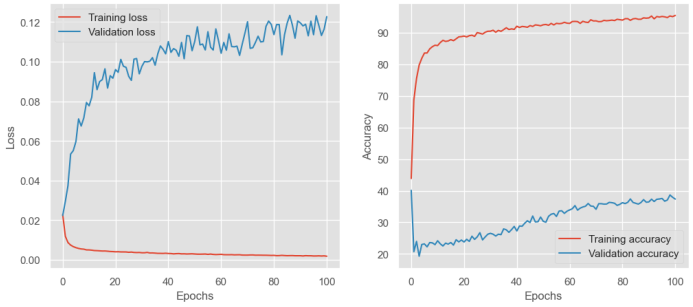
7. Channel Shuffling: For multivariate time-series data (e.g., data from multiple sensors), shuffling the channels can help the model learn to focus on the relationships between channels rather than their specific order.

When applying these techniques to time-series data, it's important to ensure that the augmentations are meaningful for the specific context and do not distort the underlying patterns in a way that would mislead the model. For example, time scaling should be used cautiously if the timing of events is critical to the classification task.

Question 3.1.d

I've used an **undersampling by randomly eliminating samples from overrepresented classes**. After applying it, the model's validation accuracy dropped significantly from 93.26% to 37.36%, and the loss increased to 0.1227. Typically, random under-sampling can help mitigate overfitting by balancing classes, but if done excessively, it may lead to the loss of important data, resulting in poor model performance and generalization, as observed with the decreased accuracy.

CNN with undersampling:
Accuracy on val data: 37.36
Loss on val data: 0.1227



Question 3.2

Question 3.2.a

```
def random_baseline_predict(class_counts, num_predictions):

    labels, counts = zip(*class_counts.items())
    total_count = sum(counts)
    probabilities = [count / total_count for count in counts]

    # Use np.random.choice to generate random predictions based on the probabilities
    predictions = np.random.choice(labels, size=num_predictions, p=probabilities)

    return predictions
```

For an imbalanced dataset, the random baseline accuracy is not as straightforward as in a balanced dataset. In a balanced dataset with six classes, a random classifier would have an accuracy of about 1/6 or approximately 16.67%. For the Human Activity Recognition task, a random baseline is particularly informative due to the class imbalance present in the dataset. This baseline isn't just a guess; it's a calculated guess based on the frequency of each activity in the data. The random_baseline_predict function in the code reflects this approach. It uses the distribution of the six activities—Walking, Jogging, Sitting, Standing, Upstairs, and Downstairs—to generate random predictions that mimic what might happen if one were to guess the activity based on how common each activity is in the dataset. This method ensures that the baseline predictions are proportionally as likely as the actual occurrences of activities, providing a fair and relevant baseline against which the performance of more sophisticated models can be compared.

Question 3.2.b

Random Baseline Accuracy:

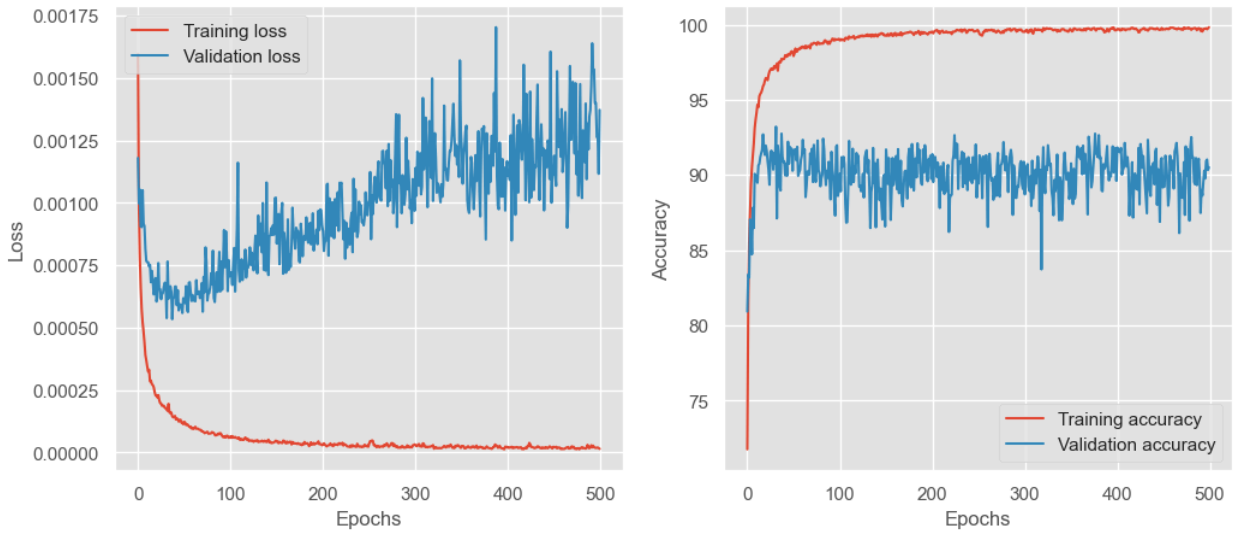
```
def calculate_accuracy(true_labels, predictions):
    correct_predictions = sum(1 for true, pred in zip(true_labels, predictions) if true == pred)
    accuracy = correct_predictions / len(true_labels)
    return accuracy

true_labels = df.activity.values
num_predictions = len(true_labels)  # Number of predictions should match the number of true labels
random_predictions = random_baseline_predict(class_counts, num_predictions)

random_baseline_accuracy = calculate_accuracy(true_labels, random_predictions)
print(f"Random Baseline Accuracy: {random_baseline_accuracy * 100:.2f}%")
```

OUTPUT: Random Baseline Accuracy: 27.19%

The random baseline accuracy is **27.19%**, which is based on the distribution of the activity classes in the dataset. This means if we guessed the activity randomly, we'd be correct about 27.19% of the time. On the other hand, the CNN model (Config 1) achieves a much higher accuracy of 93% on the validation set,

which is significantly above the random baseline. This substantial increase in accuracy demonstrates that the CNN model is not only learning patterns in the data but is also effectively differentiating between the activities. In conclusion, The CNN model significantly outperforms the random baseline.

Question 3.2.c



I've trained the model for over 500 epochs; The plots indicate that the model is indeed overfitting: the training loss continues to decrease while the validation loss increases after around 30 epochs. The training accuracy is near-perfect, but the validation accuracy fluctuates and doesn't improve much, reinforcing the overfitting diagnosis.

Question 3.3

In my modified CNN_Config1 model (my best model in previous experiments), I've added both Batch Normalization after each convolutional layer (bn1 and bn2) and Layer Normalization before the fully connected layer (ln). The choice of layers for applying BN and LN is based on optimizing training stability and performance:

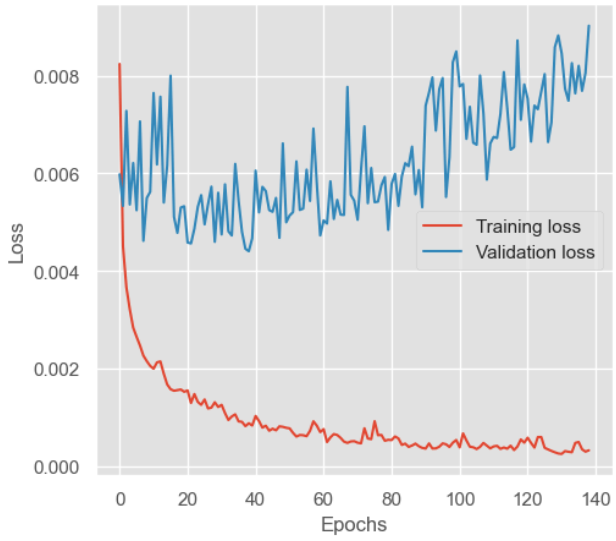
1. Batch Normalization after Convolutional Layers:
   - Helps to keep the input distribution to activation functions more stable, reducing internal covariate shift.
   - Ensures better flow of gradients through the network, preventing vanishing or exploding gradients.

2. Layer Normalization before the Fully Connected Layer:
   - Effective even with small or varying batch sizes, making it suitable for the final layers of the network.
   - Provides consistent normalization for training and inference, which is crucial for the layers close to the output.

Model Architecture:

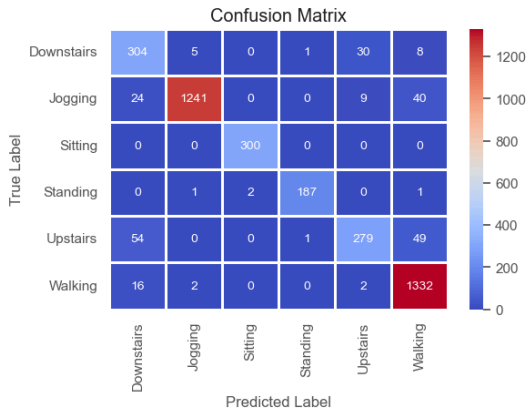
```
CNN_Config1(
  (conv1): Conv1d(3, 64, kernel_size=(5,), stride=(1,))
  (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv1d(64, 128, kernel_size=(5,), stride=(1,))
  (bn2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (adaptive_pool): AdaptiveAvgPool1d(output_size=1)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=128, out_features=6, bias=True)
  (ln): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
)
```

Performance on validation set:
Accuracy on val data: 93.69
Loss on val data: 0.009



<Figure size 600x400 with 2 Axes>

	precision	recall	f1-score	support
0	0.76	0.87	0.82	348
1	0.99	0.94	0.97	1314
2	0.99	1.00	1.00	300
3	0.99	0.98	0.98	191
4	0.87	0.73	0.79	383
5	0.93	0.99	0.96	1352
accuracy			0.94	3888
macro avg	0.92	0.92	0.92	3888
weighted avg	0.94	0.94	0.94	3888



After applying Batch Normalization (BN) and Layer Normalization (LN), the overall accuracy of the model improved from 93% to 93.7%. The precision, recall, and f1-scores for most classes also saw improvements, indicating better model performance. Notably, Class 2's recall reached 100%, and Class 1's

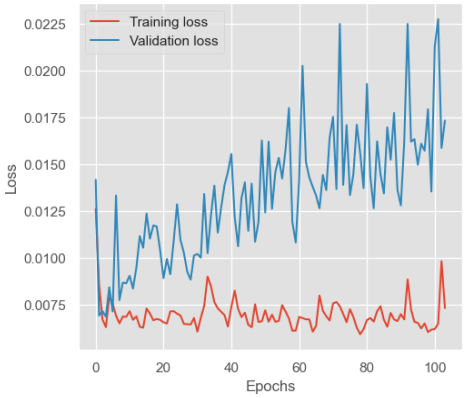
precision increased to 99%. These changes suggest that the normalization techniques helped stabilize the training process and enhanced the model's ability to generalize across different classes.

Question 3.4

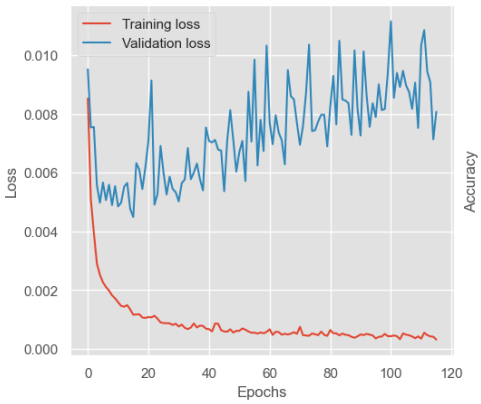
- Adam: Adam is an adaptive learning rate optimization algorithm that combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.
- AdamW: AdamW is a variant of Adam that decouples the weight decay from the optimization steps. This modification helps in stabilizing the training process and often leads to better generalization performance.

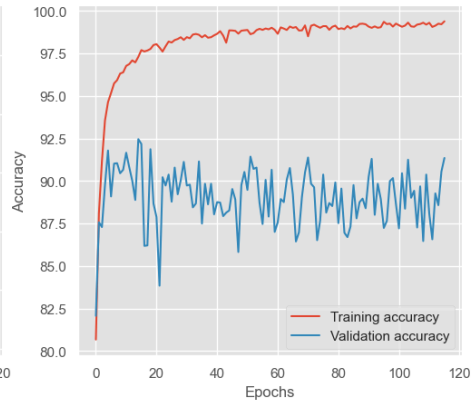
My Experiments:

Adam with Learning rate 0.1:
Accuracy on val data: 87.93
Loss on val data: 0.0173



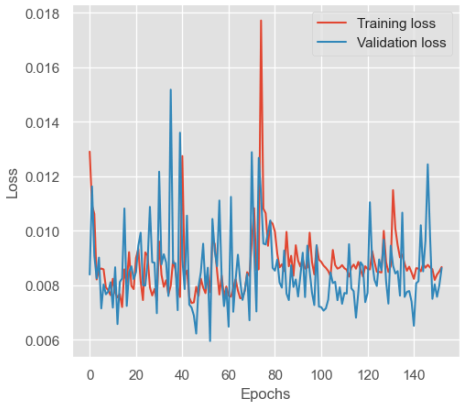
Adam with Learning rate 0.01:
Accuracy on val data: 91.35
Loss on val data: 0.0080



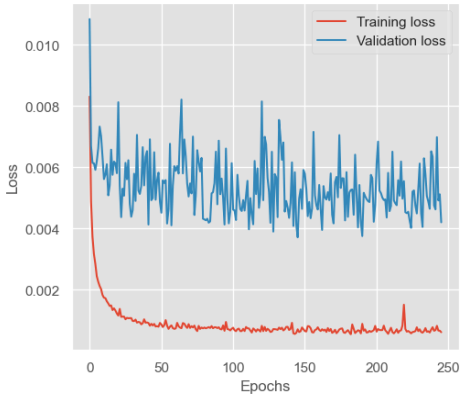


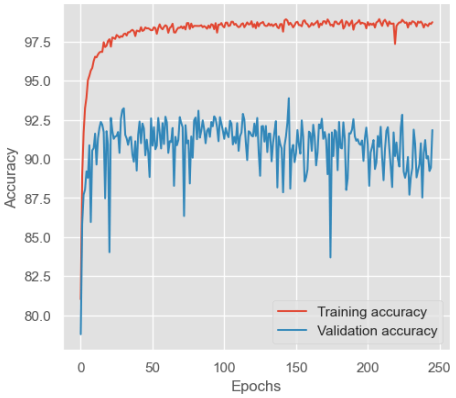
Adam with Learning rate 0.001:
Accuracy on val data: 92.56
Loss on val data: 0.0050

AdamW with Learning rate 0.1:
Accuracy on val data: 88.76
Loss on val data: 0.0086

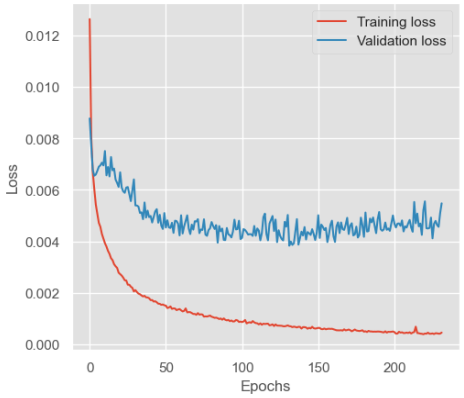


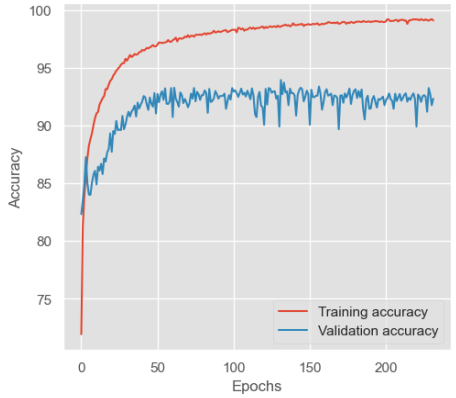
AdamW with Learning rate 0.01:
Accuracy on val data: 91.84
Loss on val data: 0.0042





AdamW with Learning rate 0.001:
Accuracy on val data: 92.33
Loss on val data: 0.0054





Performance Comparison:

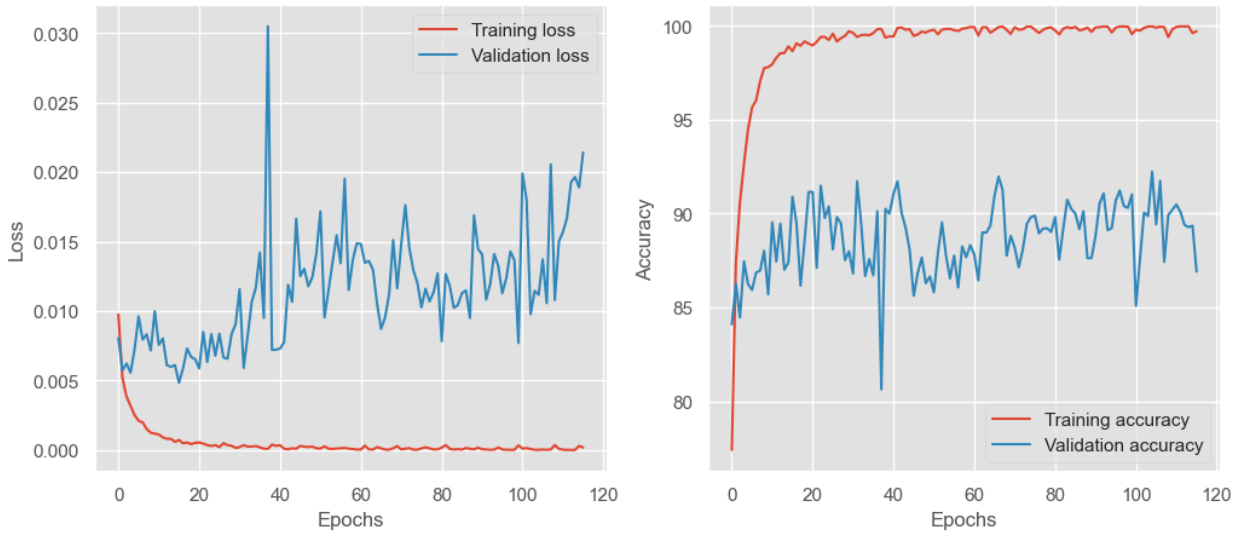
- At a high learning rate (0.1), both optimizers perform similarly, with AdamW having slightly higher accuracy and lower loss.
- At a learning rate of 0.01, AdamW outperforms Adam in terms of both accuracy and loss.
- At a lower learning rate (0.001), Adam achieves slightly higher accuracy than AdamW, but AdamW has a lower loss.

Overall, AdamW tends to have a slight edge over Adam, especially at lower learning rates, in terms of generalization performance as indicated by the validation loss.

Question 3.5

Question 3.5.a & c

Reduce the Number of Filters in Convolutional Layers:
Accuracy on val data: 86.90
Loss on val data: 0.0214

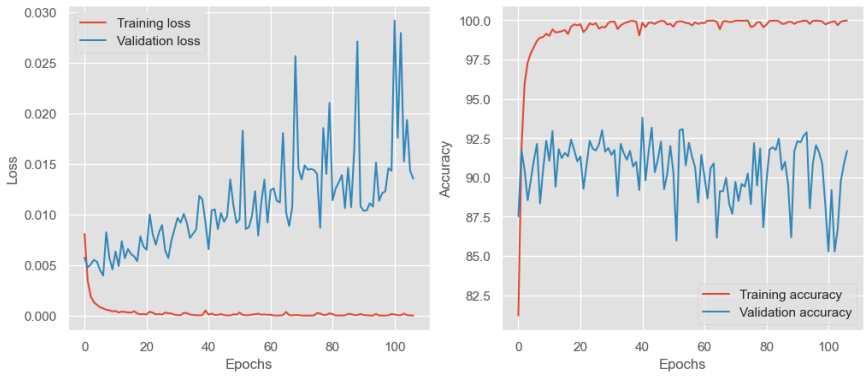


Reducing the number of filters in the convolutional layers led to a slight increase in validation accuracy from 86.10% (base CNN) to 86.90%, but with a higher validation loss, rising from 0.0017 to 0.0214. The plots show more fluctuation in validation loss, indicating potential overfitting with the simplified model.

Question 3.5.b & c

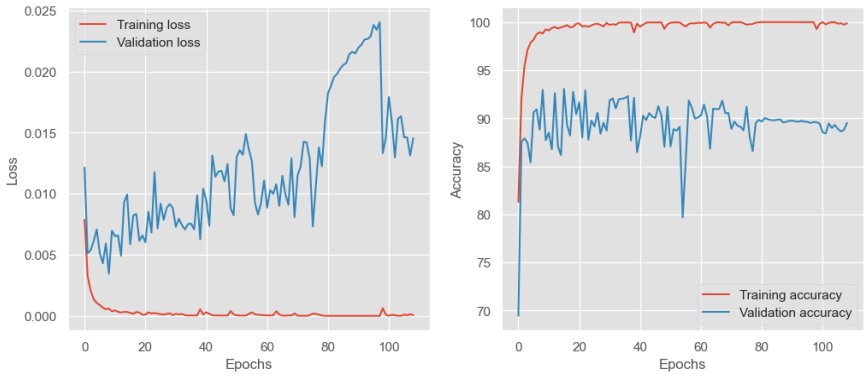
With Dropout Rate of 0.3:

Accuracy on val data: 91.69
Loss on val data: 0.0135



Without Dropout

Accuracy on val data: 89.50
Loss on val data: 0.0145

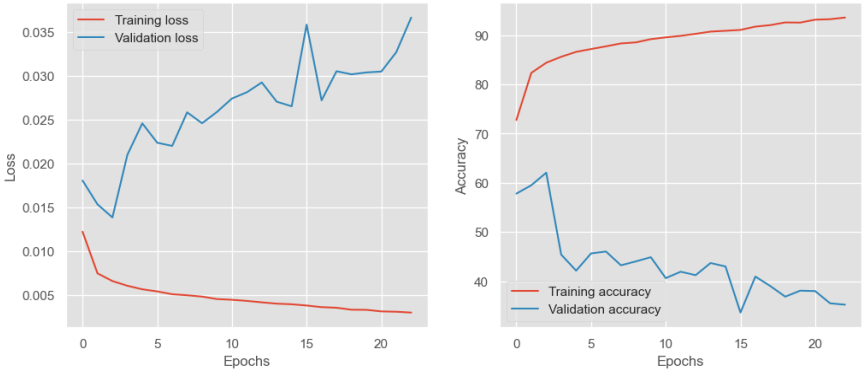


Reducing the dropout rate to 0.3 improved validation

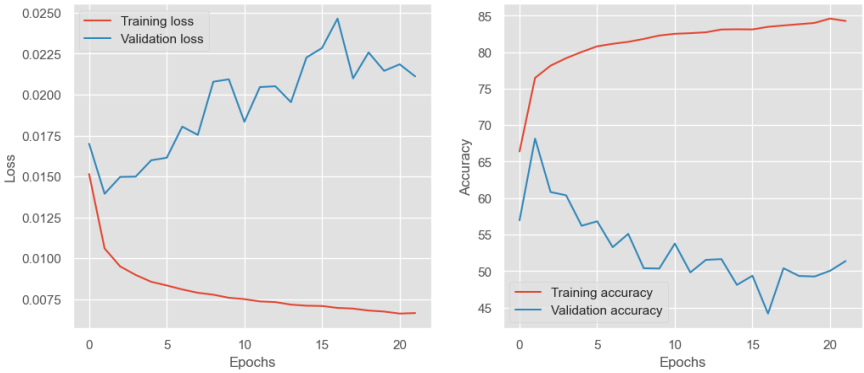
accuracy substantially to 91.69% while increasing the validation loss slightly to 0.0135. Removing dropout altogether resulted in a lower accuracy of 89.51% and a similar loss of 0.0145 compared to the 0.3 dropout rate.
Usually, reducing the dropout rate can lead to better learning if the initial rate is too restrictive, as the accuracy improves with a lower dropout rate. However, completely removing dropouts often increases the risk of overfitting, which might be why the accuracy decreased without any dropout despite the model being less complex. The increased validation loss in both cases compared to the original model also points towards a slight overfitting as the model becomes less regularized.

Question 3.5.d

selecting every 2nd sample for a new rate of 10Hz:
x_trian.shape = (20868, 120)
Accuracy on val data: 35.24
Loss on val data: 0.0366



selecting every 5th sample for a new rate of 4Hz:
x_trian.shape = (20868, 48)
Accuracy on val data: 51.37
Loss on val data: 0.0211



Typically, reducing the temporal resolution is expected to degrade model performance if the downsampling discards critical time-dependent features necessary for accurate activity recognition, which seems to be the case here. Downsampling the data to 10Hz and 4Hz significantly reduced the dataset size, which would improve computational efficiency, but it adversely affected model performance. The accuracy dropped drastically to 35.24% for the 10Hz rate and improved slightly to 51.37% for the 4Hz rate, but both were much lower than the original 93.26%. The loss also increased with downsampling.

Question 3.6

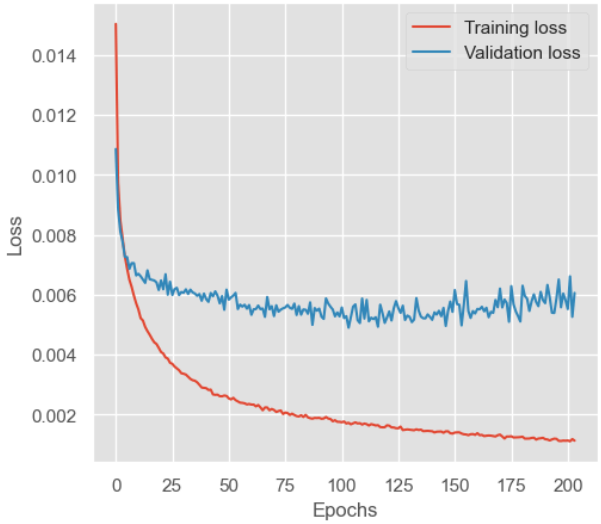
Question 3.6.a & c

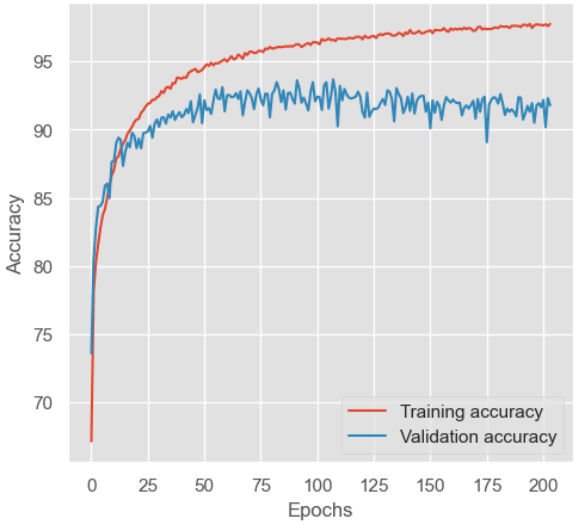
I previously used shuffle=True in my training data for experiments. For this question, I turned it to False, and here are the results.

with Random Sampling:

```
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

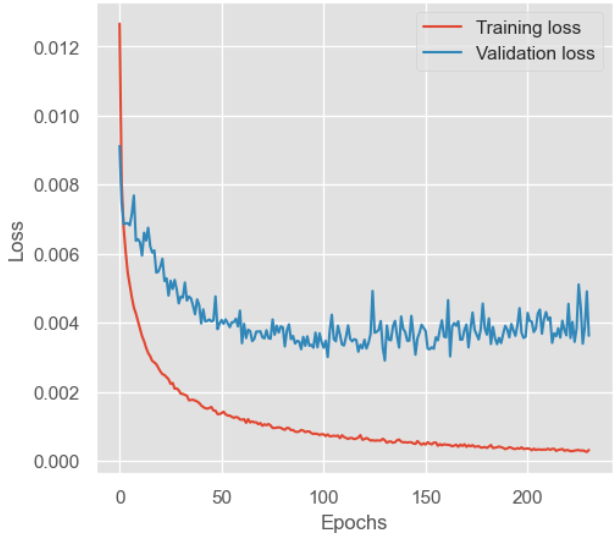
Accuracy on val data: 93.26
Loss on val data: 0.0036

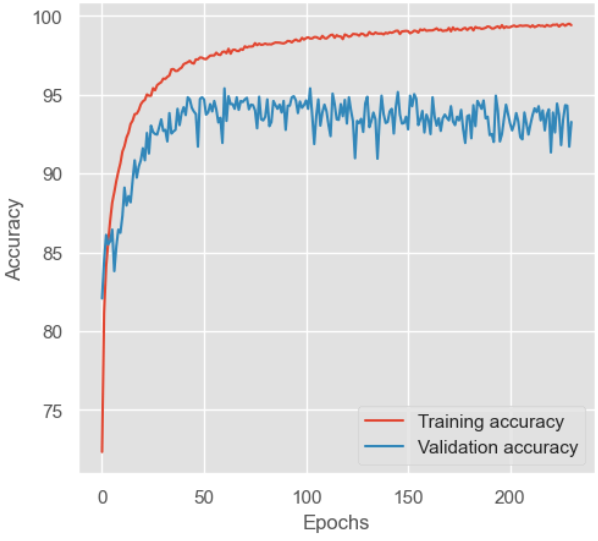




without Random Sampling:

Accuracy on val data: 91.76
Loss on val data: 0.0060





Implementing random sampling in an unbalanced dataset led to an improved validation accuracy of 93.26% compared to 91.76% without it. The loss also decreased from 0.0060 to 0.0036 with random sampling. Usually, random sampling is expected to even out the representation of classes during training, preventing the model from overfitting to more frequent activities and enhancing its ability to generalize across all activities, which is reflected in the improved performance metrics.

Question 3.6.b & c

Over Sampling Implementation:

```
for activity, count in activity_counts.items():
   df_activity = df[df['activity'] == activity]
   # Resample the minority classes
   df_activity_resampled = resample(df_activity, replace=True, n_samples=max_count, random_state=42)
   df_balanced = pd.concat([df_balanced, df_activity_resampled])
```

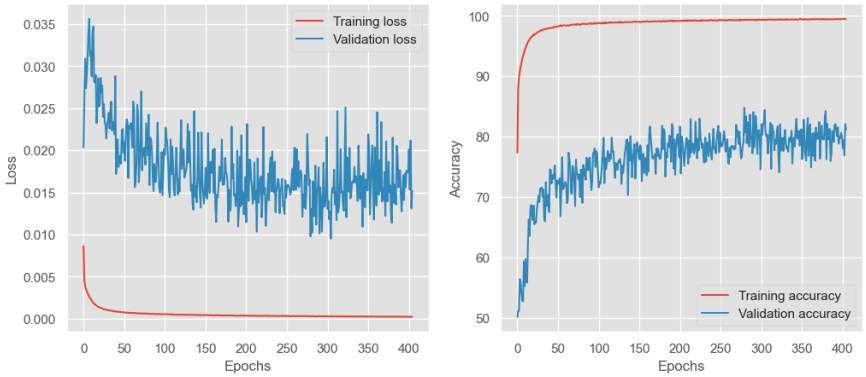
Before Over Sampling:

```
activity
Walking       424399
Jogging       342179
Upstairs      122869
Downstairs    100427
Sitting        59939
Standing       48395
Name: count, dtype: int64
```

After Over Sampling:

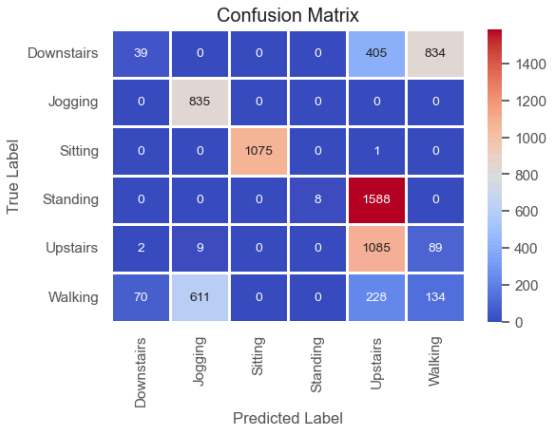
```
activity
Walking       424399
Jogging       424399
Upstairs      424399
Downstairs    424399
Sitting       424399
Standing      424399
Name: count, dtype: int64

x_train.shape (47063, 240)
y_train.shape (47063,)
```

Accuracy on val data: 81.22
Loss on val data: 0.0151



Performance on test data:

	precision	recall	f1-score	support
0	0.35	0.03	0.06	1278
1	0.57	1.00	0.73	835
2	1.00	1.00	1.00	1076
3	1.00	0.01	0.01	1596
4	0.33	0.92	0.48	1185
5	0.13	0.13	0.13	1043
accuracy			0.45	7013
macro avg	0.56	0.51	0.40	7013
weighted avg	0.59	0.45	0.35	7013



After oversampling, the model's validation accuracy decreased and the loss increased, suggesting a potential overfitting to the overrepresented classes. The precision and recall on the test set became more unbalanced; some classes saw a dramatic drop in performance, like Class 0 and Class 3, indicating the model struggled to generalize the overrepresented features to the test data. Oversampling is expected to improve recall for minority classes by giving the model more examples to learn from, but it can also lead to overfitting if not complemented with proper regularization and a balanced validation approach.