Оптимізація та оформлення власного DS пайплайну у вигляді алгоритму

Постановка задачі:

1 частина:

- Покращити пайплайн, видалити непотрібні операції, подивитися чи не використовуються лишні змінні;
- Загорнути пайплайн у функцію чи метод класу;
- Визначити O(n) алгоритму по часу і по пам’яті (лише алгоритмічна частина – без тренування моделі).

2 частина:

- Взяти датасет вашого колеги, прогнати на своєму пайплайні і спробувати отримати кращі результати.

Критерії валідації:

- Точність моделі нейронної мережі на моїх даних 95% і більше.
- Точність моделі нейронної мережі на даних іншого учасника 85% і більше.

Представлені граничні моменти:

Враховані граничні моменти:

1) Наявність шумів та викидів в показах акселерометра і гіроскопа;
2) Наявність пауз між виконанням вправ.

Невраховані граничні моменти:

1) Наявність інших видів фізичної активності, відмінних від досліджуваних;
2) Фіксування датчиків в іншому положенні, ніж те положення, в якому знаходився смартфон під час запису даних;
3) Відсутність даних для людей різних категорій (вік, стать, чи займається спортом і т.д.), оскільки алгоритм тестувався лише на моїх даних та даних іншого учасника школи.

План виконання першої частини

1) Розробити вебзастосунок для перегляду проміжних результатів виконання пайплайну, що дозволить покроково тестувати основні етапи не оптимізованого алгоритму під час його переписування із набору клітинок в Jupyter Lab в єдину функцію в PyCharm.
2) Оцінити час виконання та витрати пам’яті не оптимізованим алгоритмом для вебзастосунку.
3) Оцінити час виконання та витрати пам’яті не оптимізованим алгоритмом в Jupyter Lab та порівняти отримані результати.
4) Оптимізувати алгоритм та оцінити вищезгадані показники.

Етап 1: Розробка вебзастосунку

Оскільки часу на виконання завдання було не багато, для розробки вебзастосунку з метою демонстрації основних кроків мого DS-пайплайну було обрано бібліотеку Python – Streamlit, яка дозволяє доволі швидко створити веб-застосунок для відображення отриманих результатів (рис. 1).

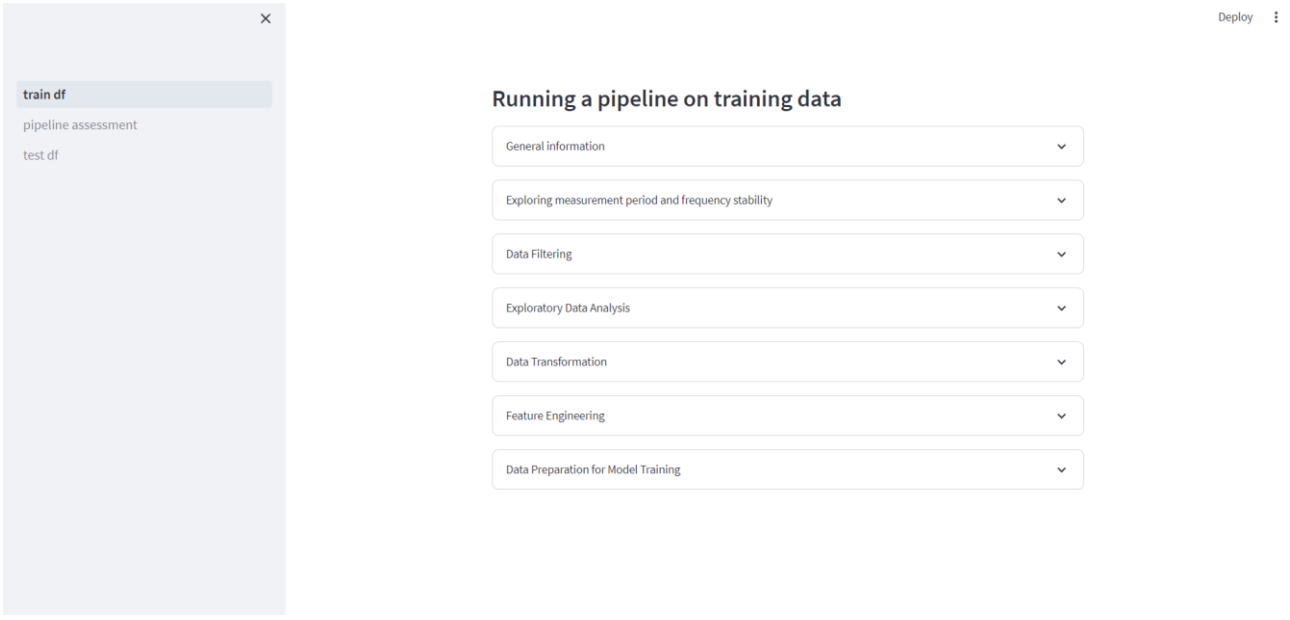

Рис. 1. Вебзастосунок для відображення основних кроків мого DS пайплайну

Отриманий вебзастосунок складається з трьох вебсторінок:

1) train df– відображає результати виконання різних кроків DS пайплайну для тренувального датасету;
2) pipeline assessment – відображає час роботи та споживання пам’яті алгоритму;
3) test df - відображатиме результати виконання різних кроків DS пайплайну для тестового датасету.

Наприклад, на вебсторінці "train df" можна буде переглянути вплив медіанного фільтра на різні осі акселерометра чи гіроскопа (рис. 2), або чи зберігається співвідношення класів фізичної активності внаслідок віконування (рис. 3).

Рис. 2. Вплив медіанного фільтра на вісь OX акселерометра

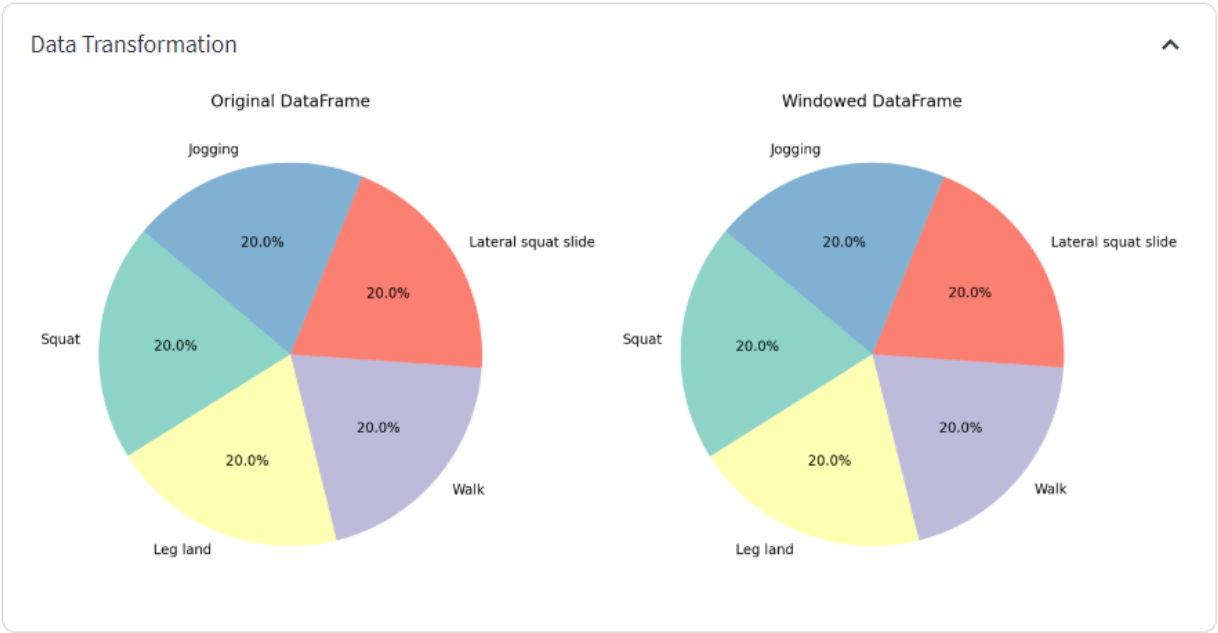

Рис. 3. Відображення результатів виконування (розподіл класів фізичної активності)

Етап 2: Оцінка часу виконання та витрат пам’яті не оптимізованого алгоритму для вебзастосунку

Алгоритм було оформлено у вигляді функції perform_pipeline(). Для оцінки згаданих параметрів під час роботи алгоритму у вебзастосунку, проводилося по 5 дослідів окремо для дослідження часу і пам’яті. Для дослідження часу виконання алгоритму було використано модуль time, а пам’яті – memory_profiler. Одночасне дослідження двох параметрів не дозволить правильно оцінити час виконання алгоритму. Це пояснюється тим, що модуль memory_profiler додає додаткові витрати для відстеження використання пам’яті для кожного рядка коду, який виконується. Чим більше у рядків коду та чим більше вимірювань пам’яті виконується, тим більший вплив на час виконання. Модуль memory_profiler може значно уповільнити виконання коду, особливо якщо він використовується для профілювання великих функцій або розділів коду. Це уповільнення очікуване, і це компроміс для отримання детальної інформації про використання пам’яті на детальному рівні.

Для експериментів використовувався мій тренувальний датасет, який містить 63529 записів для 5 різних видів фізичної активності (рис. 4)

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 63529 entries, 0 to 63528
Data columns (total 8 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   timestamp  63529 non-null  int64
 1   time       63529 non-null  float64
 2   accX       63529 non-null  float64
 3   accY       63529 non-null  float64
 4   accZ       63529 non-null  float64
 5   gyrX       63529 non-null  float64
 6   gyrY       63529 non-null  float64
 7   gyrZ       63529 non-null  float64
dtypes: float64(7), int64(1)
memory usage: 3.9 MB
```

Рис. 4. Інформація про тренувальний датасет

Результати вимірювань можна переглянути на вебсторінці "pipeline assessment" (рис. 5).

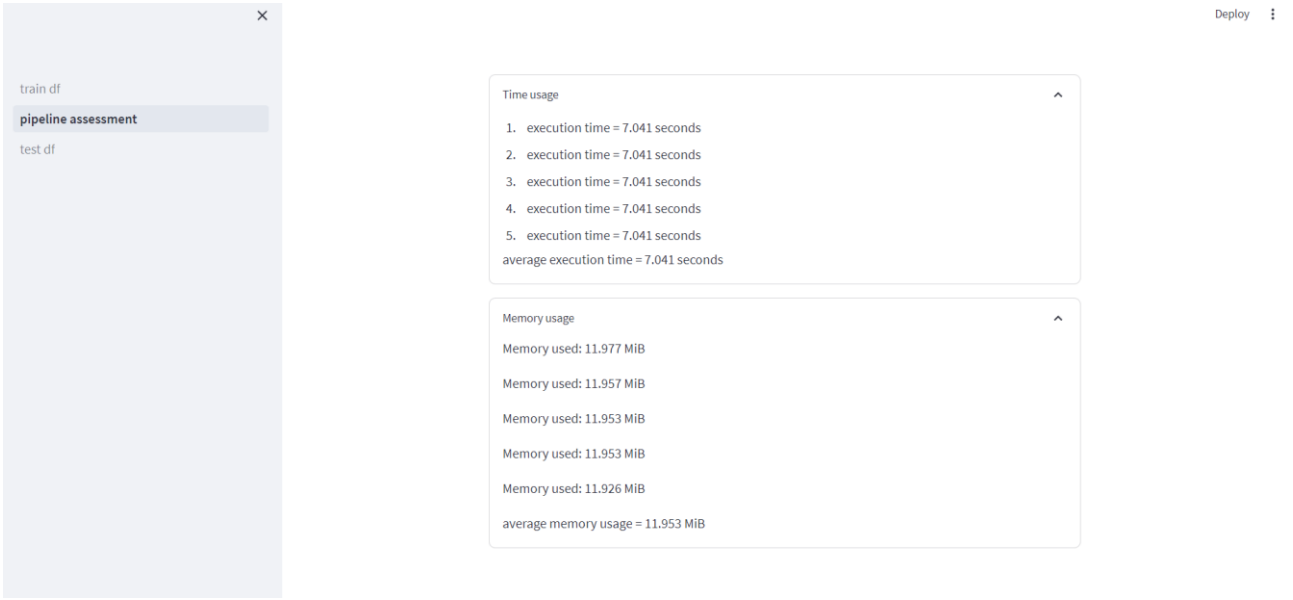

Рис. 5. Дослідження часу виконання і споживання пам’яті не оптимізованим алгоритмом у вебзастосунку

Як видно із рис. 5, середній час виконання не оптимізованого алгоритму становить приблизно 7 секунд, а середні витрати пам’яті – 11.953 MiB (1 Mebibyte = 2^20 bytes = 1 048 576 bytes). Бібліотека memory_profiler також виводить у консоль детальний звіт використання алгоритмом пам’яті (рис. 6).

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    50    328.1 MiB    328.1 MiB           1   @profile
    51                                         def perform_pipeline_memory(df_arg):
    52                                             # Exploring measurement period and frequency stability
    53    333.8 MiB      5.6 MiB           1       df_arg = df_arg[df_arg['time'].diff() <= frequency_stability.get_avg_period(df_arg, 'time') * 1.5]
    54
    55                                             # Data Filtering
    56    339.1 MiB      1.4 MiB           1       df_arg[['accX_filtered', 'accY_filtered', 'accZ_filtered', 'gyrX_filtered', 'gyrY_filtered', 'gyrZ_filtered']] \
    57    337.6 MiB      3.9 MiB           2           = data_filtering.median_filter_data(df_arg=df_arg,
    58    333.8 MiB      0.0 MiB           1                                               filter_columns_arg=['accX', 'accY', 'accZ', 'gyrX', 'gyrY', 'gyrZ'],
    59    333.8 MiB      0.0 MiB           1                                               window_size_arg=10)
    60
    61                                             # Exploratory Data Analysis
    62    341.7 MiB      2.6 MiB           1       df_arg = df_arg[df_arg['activity'] != 'No activity']
    63
    64                                             # Perform undersampling to get a balanced dataframe
    65    343.8 MiB      2.1 MiB           1       df_arg = exploratory_data_analysis.get_undersampled_df(df_arg=df_arg, column_name_arg='activity')
    66
    67                                             # Build a correlation matrix and remove certain axes of the accelerometer or gyroscope
    68    343.8 MiB      0.0 MiB           1       sel_columns = ['accX_filtered', 'accY_filtered', 'accZ_filtered', 'gyrX_filtered', 'gyrY_filtered',
    69                                                            'gyrZ_filtered']
    70
    71                                             # Calculate the correlation matrix for the selected columns
    72    343.8 MiB     -0.0 MiB           1       corr_matrix = df_arg[sel_columns].corr()
    73    343.8 MiB      0.0 MiB           1       important_columns = ['accX_filtered', 'accY_filtered', 'accZ_filtered']
    74    343.8 MiB      0.0 MiB           2       discard_columns = exploratory_data_analysis.get_discard_columns(corr_matrix_arg=corr_matrix,
    75    343.8 MiB      0.0 MiB           1                                                                       important_columns_arg=important_columns,
    76    343.8 MiB      0.0 MiB           1                                                                       df_arg=df_arg)
    77    343.8 MiB      0.0 MiB           9       sel_columns = [col for col in sel_columns if col not in discard_columns]
    78    343.8 MiB      0.0 MiB           1       sel_columns.append('activity')
    79    346.8 MiB      3.0 MiB           1       filtered_df = df_arg[['time'] + sel_columns].copy()
    80
    81                                             # Windowing
    82    346.8 MiB     -0.0 MiB           1       windowed_df = windowing.get_windowed_df(df_arg=filtered_df, window_duration_arg=2)
    83
    84                                             # Feature Engineering
    85    346.8 MiB      0.0 MiB           1       windowed_df = pipeline.perform_feature_engineering(df_arg=windowed_df)
    86
    87                                             # Model training
    88    347.8 MiB      1.0 MiB           1       return pipeline.model_training_data_preparation(df_arg=windowed_df)
```

Рис. 6. Детальний звіт з використання алгоритмом пам’яті

Проаналізувавши звіт з використання пам’яті, наведений на рис. 6, можна побачити, що найбільше використання пам’яті (Increment = 5.6 MiB) спостерігається для команди

```
df_arg = df_arg[df_arg['time'].diff() <= frequency_stability.get_avg_period(df_arg,
'time') * 1.5]
```

тобто під час вилучення записів із аномальним значенням періоду вимірювання.

Етап 3: Оцінка часу виконання та витрат пам’яті не оптимізованого алгоритму в Jupyter Lab

З метою ізолювати використання пам’яті і час виконання алгоритму від інших факторів, таких як кешування та інші оптимізації, які можуть виникнути під час запуску алгоритму в Streamlit-додатку, було вирішено здійснити оцінку алгоритму в Jupyter Lab.

Для оцінки алгоритму також використовувався той самий тренувальний датасет і також проводилися по 5 дослідів для часу виконання і використання пам’яті, результати яких наведені на рис. 7 і рис. 8.

```
1) time = 6.993 seconds
2) time = 7.139 seconds
3) time = 7.066 seconds
4) time = 6.929 seconds
5) time = 6.876 seconds
average execution time =  7.001 seconds
```

Рис. 7. Дослідження часу виконання не оптимізованого алгоритму у Jupyter Lab

```
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_8448\3798329931.py
1) memory = 0.035 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_8448\3798329931.py
2) memory = 0.023 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_8448\3798329931.py
3) memory = 9.969 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_8448\3798329931.py
4) memory = 0.008 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_8448\3798329931.py
5) memory = 9.961 MiB
average memory usage = 3.999 MiB
```

Рис. 8. Дослідження споживання пам’яті не оптимізованим алгоритмом у Jupyter Lab

Переглянувши результати, наведені на рис. 5 і рис. 7, можна зробити висновок, що не оптимізований алгоритм виконується приблизно 7 секунд, як під час запуску в PyCharm для за стосунку Streamlit, так і в Jupyter Lab.

Однак якщо розглянути результати споживання пам’яті в Jupyter Lab для 5-ти експериментів (рис. 8) можна спостерігати наявність значних коливань від

майже 0 MiB до 10 MiB, тому середнє значення використаної пам’яті становить приблизно 4 MiB.

Такі коливання результатів можна пояснити наявністю декількох факторів, які можуть призводити до різних вимірювань споживання пам’яті:

1) Кешування та поведінка системи: на використання пам’яті можуть впливати кешування системного рівня, збір сміття та інші фонові процеси. Ці фактори можуть призвести до різного використання пам’яті від одного запуску до іншого.
2) Тимчасові об’єкти: створення та видалення тимчасових об’єктів під час виконання коду може вплинути на використання пам’яті. Якщо код створює та скидає тимчасові структури даних, використання пам’яті може коливатися.
3) Інтерпретатор і бібліотеки Python: керування пам’яттю Python і поведінка зовнішніх бібліотек можуть впливати на споживання пам’яті. Бібліотеки можуть використовувати пам’ять по-різному залежно від того, як вони реалізовані.

Також як видно із рис. 8, через те, що системі не вдалося знайти певного тимчасового .py файлу, в Jupyter Lab не вийшло переглянути детальний звіт по споживанню пам’яті.

Оскільки під час вимірювання споживаної пам’яті для вебзастосунку в PyCharm покази пам’яті не коливалися так сильно, а були приблизно однакові, було прийнято рішення спробувати оцінити ефективність алгоритму в PyCharm без використання бібліотеки Streamlit (тобто було створено окремий файл "algorithm estimation"), в якому значення пам’яті виводяться в консоль (рис. 9).

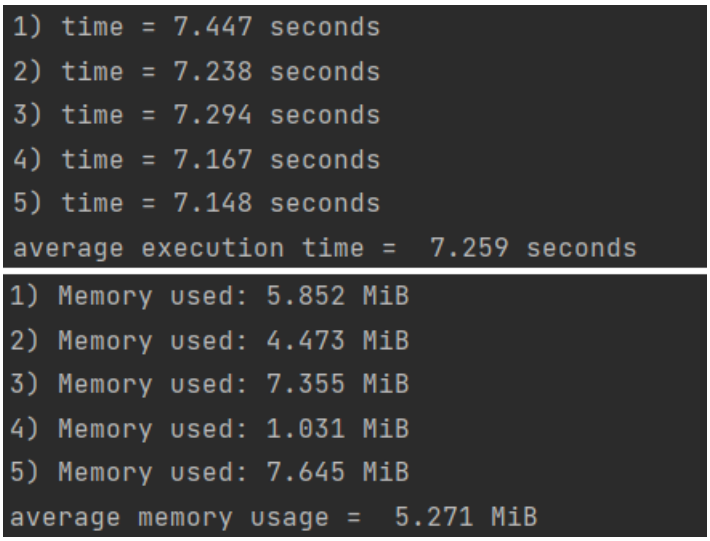

Рис. 9. Дослідження часу виконання і споживання пам’яті не оптимізованим алгоритмом у PyCharm

Із рис. 9 видно, що середній час майже збігається із попередніми показниками (рис. 5 і рис. 7), а для пам’яті знову спостерігаються коливання від 1 MiB до 7.5 MiB.

Наступним кроком було здійснено спробу використати замість бібліотеки memory_profiler бібліотеку psutil в Jupyter Lab, однак було отримані ще дивніші результати (рис. 10): серед результатів вимірювання пам’яті з допомогою psutil наявні навіть від’ємні показники, що не має ніякого сенсу.

```
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_7412\3798329931.py
1) memory = -26.438 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_7412\3798329931.py
2) memory = 6.047 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_7412\3798329931.py
3) memory = 0.262 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_7412\3798329931.py
4) memory = -0.215 MiB
ERROR: Could not find file C:\Users\ASUS\AppData\Local\Temp\ipykernel_7412\3798329931.py
5) memory = 7.934 MiB
Average memory usage = -2.482 MiB
```

Рис. 10. Дослідження споживання пам’яті не оптимізованим алгоритмом з допомогою бібліотеки psutil

Також я виконав спробу оцінити використовувану пам’ять з допомогою утиліти Windows – Process Explorer, зокрема використовуючи стовпці "Private Bytes" та "Working Set" (рис. 11). Стовпець "Private Bytes" надасть пам’ять, яка використовується виключно досліджуваним процесом, тоді як "Working Set" покаже фактичну фізичну пам’ять, яка використовується в цей момент. Однак відстежити зміни в пам’яті таким чином теж не вдалося, оскільки значення цих стовпців для обраного процесу постійно змінювалися і без запуску алгоритму.

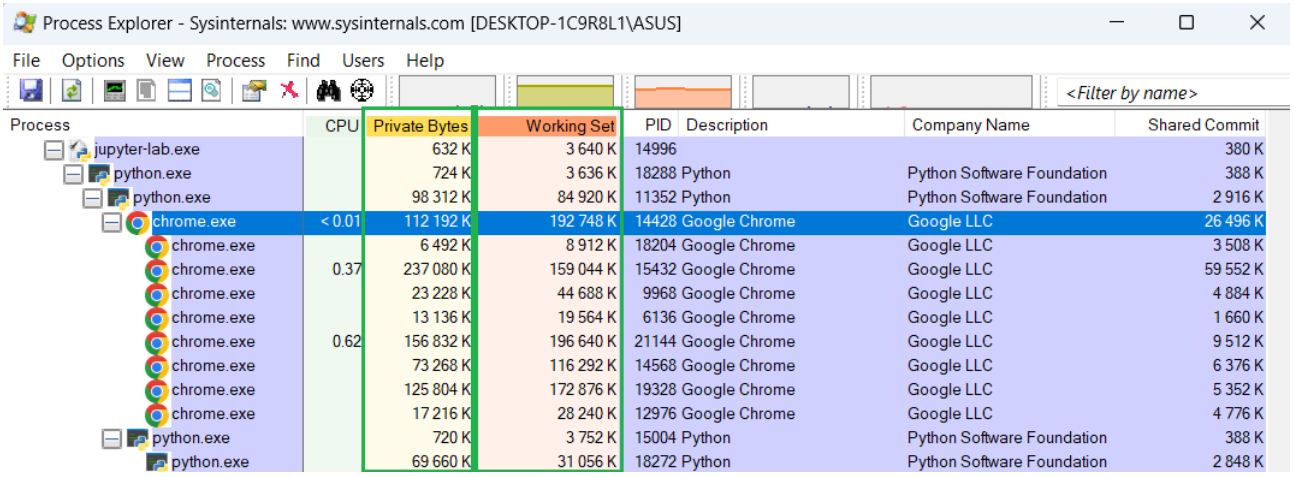

Рис. 11. Дослідження споживання пам’яті не оптимізованим алгоритмом з допомогою утиліти Process Explorer

Оскільки не вдалося знайти ефективний спосіб оцінки пам’яті, яку споживає лише алгоритм, для подальшої оцінки оптимізації алгоритму було використано лише час його виконання.

Етап 4: Оптимізація алгоритму

Перша версія алгоритму була не оптимізованою, оскільки для відображення результатів різних кроків мого DS пайплайну на вебсторінках Streamlit, потрібно було зберігати проміжні результати або в нових стовпцях оригінального датафрейму (наприклад, для відображення впливу медіанного фільтра) або в окремих датафреймах (наприклад, для відстеження розподілу класів фізичної активності в оригінальному та віконованому датафреймах) (рис. 12).

```
# Data Filtering
df[['accX_filtered', 'accY_filtered', 'accZ_filtered', 'gyrX_filtered', 'gyrY_filtered', 'gyrZ_filtered']] \
    = data_filtering.median_filter_data(df_arg=df,
                                        filter_columns_arg=['accX', 'accY', 'accZ', 'gyrX', 'gyrY', 'gyrZ'],
                                        window_size_arg=10)
```

```
filtered_df = df[['time'] + sel_columns].copy()

# Windowing
windowed_df = windowing.get_windowed_df(df_arg=filtered_df, window_duration_arg=2)

# Feature Engineering
windowed_df = pipeline.perform_feature_engineering(df_arg=windowed_df)

# Model Training Preparation
X_train, y_train, X_valid, y_valid = pipeline.model_training_data_preparation(df_arg=windowed_df)
```

Рис. 12. Зберігання проміжних результатів для подальшого відображення на вебсторінках Streamlit розробленого вебдодатку

Для оптимізації алгоритму, усі результати зберігалися в одному-єдиному датафреймі, тобто було вилучено всі лишні змінні. Окрім того, я постарався максимально мінімізувати кількість функцій, які викликатимуться під час виконання пайплайну і вилучити всі зайві операції (наприклад, побудову графіків). В результаті вдалося отримати значне покращення для швидкості виконання алгоритму від 7 секунд (рис. 7) до 4 секунд (рис. 13).

```
Optimized pipeline
1) time = 4.001 seconds
2) time = 3.791 seconds
3) time = 3.950 seconds
4) time = 3.892 seconds
5) time = 3.872 seconds
average execution time =  3.901 seconds
```

Рис. 13. Дослідження часу виконання оптимізованого алгоритму у Jupyter Lab

Етап 4: Використання алгоритму для тренування моделі нейронної мережі

Розглянутий раніше алгоритм працював лише із тренувальним датасетом і забезпечував його поділ на тренувальні та валідаційні дані.

Оскільки для оцінки моделі нейронної мережі потрібен і тестовий датасет, до отриманого алгоритму було додано функціональність, яка забезпечує підготовку ще й тренувальних даних. Тобто, на вхід алгоритму подаватиметься тренувальний і тестовий датасети, а на виході буде отримано X_train, y_train, X_valid, y_valid, X_test, y_test, підготовлені безпосередньо до тренування моделі.

Додавання нової функціональності в алгоритм безперечно збільшить час його виконання, що і спостерігається на рис. 14.

```
Optimized pipeline
1) time = 8.594 seconds
2) time = 8.469 seconds
3) time = 8.429 seconds
4) time = 8.560 seconds
5) time = 8.456 seconds
average execution time =  8.502 seconds
```

Рис. 14. Дослідження часу виконання повного оптимізованого алгоритму

Перед тим, як використовувати свій алгоритм-пайплайн на даних іншого учасника, протестуємо його на власних даних. Для початку переглянемо розміри тренувальних, валідаційних і тестових даних (рис. 15), після чого здійснимо 5 експериментів із тренування моделі нейронної мережі, архітектура якої наведена на рис. 16.

```
len(X_train) = 1440
len(y_train) = 1440
len(X_valid) = 361
len(y_valid) = 361
len(X_test) = 360
len(y_test) = 360
```

Рис. 15. Розміри тренувальних, валідаційних і тестових даних

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(60, activation='relu', input_shape=(len(X_train[0]), )))
model.add(tf.keras.layers.Dense(30, activation='relu'))
model.add(tf.keras.layers.Dense(15, activation='sigmoid'))
model.add(tf.keras.layers.Dense(5, activation='softmax'))

# compile the keras model
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.RMSprop(), metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=5, verbose=0, batch_size=10, validation_data=(X_valid, y_valid))

# evaluate the keras model
loss, accuracy = model.evaluate(X_test, y_test)
```

Рис. 16. Архітектура моделі нейронної мережі

```
12/12 [==============================] - 0s 1ms/step - loss: 0.0745 - accuracy: 0.9861
Accuracy: 98.611
12/12 [==============================] - 0s 1ms/step - loss: 0.1050 - accuracy: 0.9750
Accuracy: 97.500
12/12 [==============================] - 0s 2ms/step - loss: 0.0829 - accuracy: 0.9806
Accuracy: 98.056
12/12 [==============================] - 0s 2ms/step - loss: 0.0988 - accuracy: 0.9667
Accuracy: 96.667
12/12 [==============================] - 0s 2ms/step - loss: 0.0856 - accuracy: 0.9778
Accuracy: 97.778

Average accuracy = 97.722
```

Рис. 17. Результати тренування моделі нейронної мережі

Як видно із рис. 17, середня точність моделі становить понад 97%, а найкраща досягнута точність – 98.611%. Такі показники майже співпадають із значеннями точності, отриманими в попередньому домашньому завданні.

Також здійснено перевірку відсутності перенавчання на прикладі "найкращої" моделі, тобто моделі із найбільшою точністю (рис. 18).

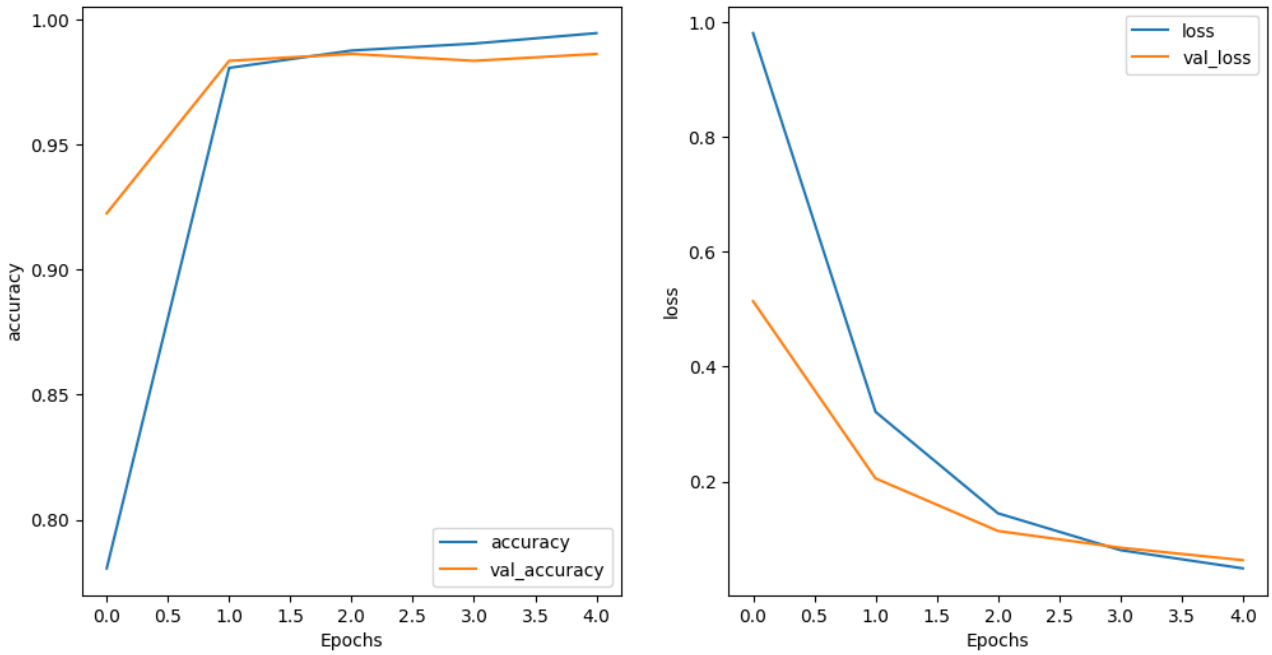

Рис. 18. Зміна тренувальних і валідаційних точності і втрат протягом 5 епох під час навчання моделі нейронної мережі

Із рис. 18 бачимо, що під час тренування моделі overfittig не проявляється.

Отже, отримані результати тренування моделі нейронної мережі підтверджують правильність роботи алгоритму мого DS пайплайну.

Етап 5: Використання алгоритму на даних іншого учасника

Перед безпосереднім використанням алгоритму на даних іншого учасника (https://github.com/TheXirex) та тренуванням моделі, розглянемо досліджуваний датасет.

В першу чергу варто відзначити, що на відміну від моїх даних, де були окремий тренувальний і тестовий датасети, у цьому випадку є лише один датасет (рис. 19).

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34500 entries, 0 to 34499
Data columns (total 10 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   ax           34500 non-null  float64
 1   ay           34500 non-null  float64
 2   az           34500 non-null  float64
 3   gx           34500 non-null  float64
 4   gy           34500 non-null  float64
 5   gz           34500 non-null  float64
 6   label        34500 non-null  object
 7   time         34500 non-null  float64
 8   timestamp    34500 non-null  float64
 9   dummy_label  34500 non-null  int64
dtypes: float64(8), int64(1), object(1)
memory usage: 2.6+ MB
```

Рис. 19. Загальна інформація про датасет іншого учасника

Отриманий датасет було трохи змінено (рис. 20):

1) Вилучено непотрібні для мого алгоритму стовпці: 'timestamp' та 'dummy_label';
2) Стовпець 'label' замінено на стовпець 'activity' для зручності перевірки результатів.

	ax	ay	az	gx	gy	gz	time	activity
0	-0.507871	-0.018787	0.080167	-0.044739	-1.061773	0.363105	5.073	Stay
1	0.477677	-0.282922	0.563089	0.564088	-2.054513	0.342565	5.116	Stay
2	-0.255176	0.007905	0.641888	0.024892	-0.437928	-0.129863	5.159	Stay
3	-0.336323	-0.292223	-0.120661	0.024892	-1.110912	-0.006621	5.203	Stay
4	-0.671813	-0.110583	0.200508	-0.576793	0.660209	-0.339904	5.247	Stay

Рис. 20. Результати зміни оригінального датафрейму

Наступним кроком було переглянуто співвідношення між класами фізичної активності (рис. 21).

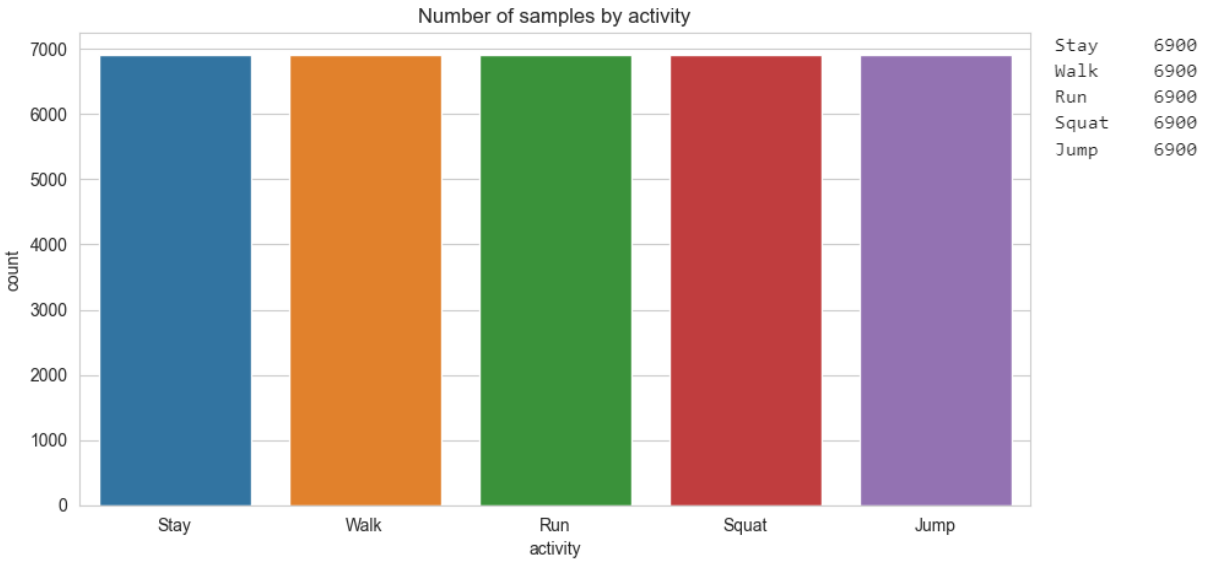

Рис. 21. Аналіз розподілу класів досліджуваного датасету

Із рис. 21 видно, що досліджуваний датасет є збалансованим, а кожен клас фізичної активності містить 6900 записів.

Оскільки мій пайплайн-алгоритм приймає на вхід тренувальний і тестовий датасети, досліджуваний датасет було поділено у співвідношенні 80/20, так щоб співвідношення класів в обох датасетах збереглося. В результаті поділу для тренування і валідації моделі було виділено 27600 записів, а для тестування – 6900 записів.

Після поділу здійснено перевірку збереження співвідношення між класами фізичної активності (рис. 22, рис. 23) і зроблено висновок, що тренувальний і тестовий датасети є збалансованими.

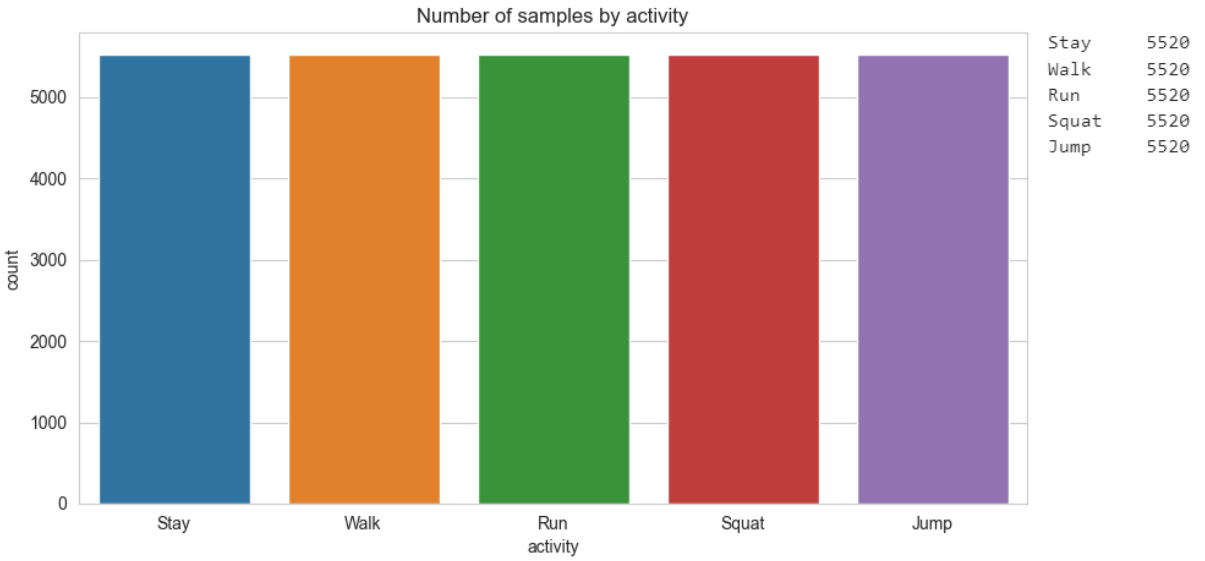

Рис. 22. Аналіз розподілу класів тренувального датасету

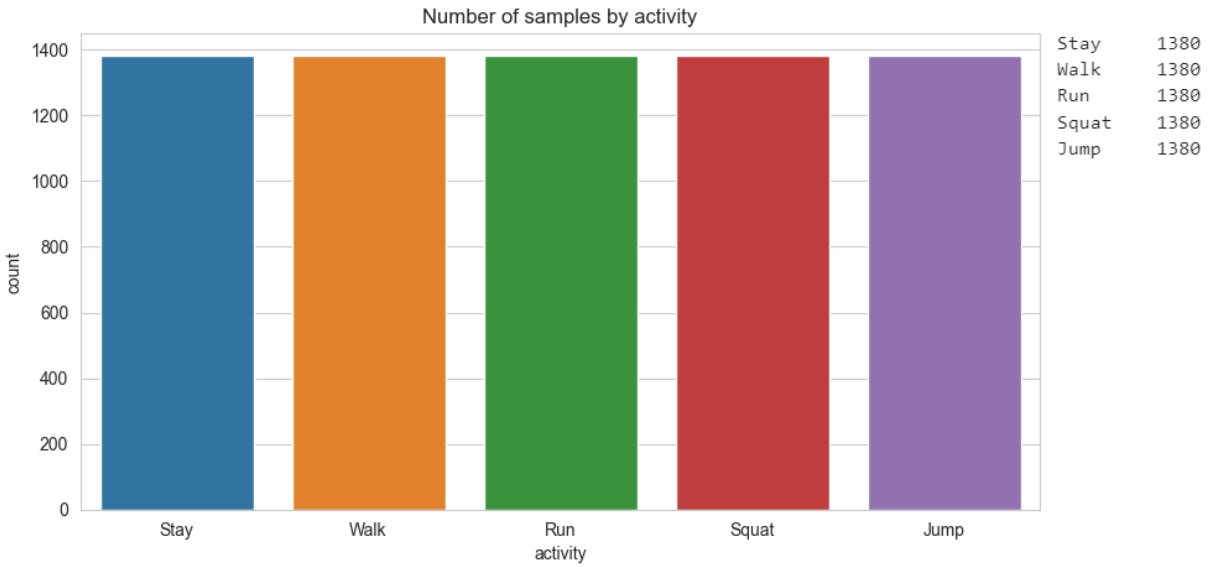

Рис. 23. Аналіз розподілу класів тестового датасету

Далі здійснено оцінку часу роботи алгоритму-пайплайну над тренувальним і тестовим датасетами (рис. 24).

```
Optimized pipeline (another participant's data)
1) time = 2.699 seconds
2) time = 2.684 seconds
3) time = 2.852 seconds
4) time = 2.720 seconds
5) time = 2.695 seconds
average execution time =  2.730 seconds
```

Рис. 24. Дослідження часу виконання алгоритму на даних іншого учасника

Як видно із рис. 24, середній час виконання пайплайну становить 2.730 с, що значно менше, ніж середній час виконання на моїх даних, який становив 8.502 с (рис. 14). Пояснити отримані результати дуже просто: зменшення часу виконання алгоритму-пайплайну для даних іншого учасника пов’язане із різними розмірами досліджуваних датасетів:

- Мій тренувальний датасет містив 63529 записів, а тренувальний датасет іншого учасника – 27600 записів;
- Мій тестовий датасет – 12557 записів, тестовий датасет іншого учасника – 6900 записів.

Перед тренуванням моделі нейронної мережі було розглянуто кількість записів для тренування, валідації та тестування моделі (рис. 25).

```
len(X_train) = 315
len(y_train) = 315
len(X_valid) = 80
len(y_valid) = 80
len(X_test) = 210
len(y_test) = 210
```

Рис. 25. Розміри тренувальних, валідаційних і тестових даних

Для тренування моделі на даних іншого учасника було використано ту саму архітектуру моделі, що й для тренування на моїх даних (рис. 16). Результати 5 експериментів наведені на рис. 17.

```
7/7 [==============================] - 0s 2ms/step - loss: 0.4577 - accuracy: 0.9476
Accuracy: 94.762
7/7 [==============================] - 0s 2ms/step - loss: 0.5056 - accuracy: 0.9286
Accuracy: 92.857
7/7 [==============================] - 0s 3ms/step - loss: 0.5389 - accuracy: 0.9286
Accuracy: 92.857
7/7 [==============================] - 0s 2ms/step - loss: 0.5051 - accuracy: 0.9524
Accuracy: 95.238
7/7 [==============================] - 0s 2ms/step - loss: 0.4404 - accuracy: 0.9667
Accuracy: 96.667

Average accuracy = 94.476
```

Рис. 26. Результати тренування моделі нейронної мережі на даних іншого учасника

Із рис. 26 видно, що середня точність моделі становила 94.476%, а найкраща досягнута точність – 96.667%. Така хороша точність, отримана внаслідок виконання пайплайн-алгоритму свідчить про дві речі:

1) Хороша якість даних досліджуваного датасету;
2) Висока ефективність мого алгоритму-пайплайну для задач класифікації видів фізичної активності.

Також було здійснено перевірку на перенавчання "найкращої" моделі (рис. 27), яка показала, що overfitting не спостерігався.

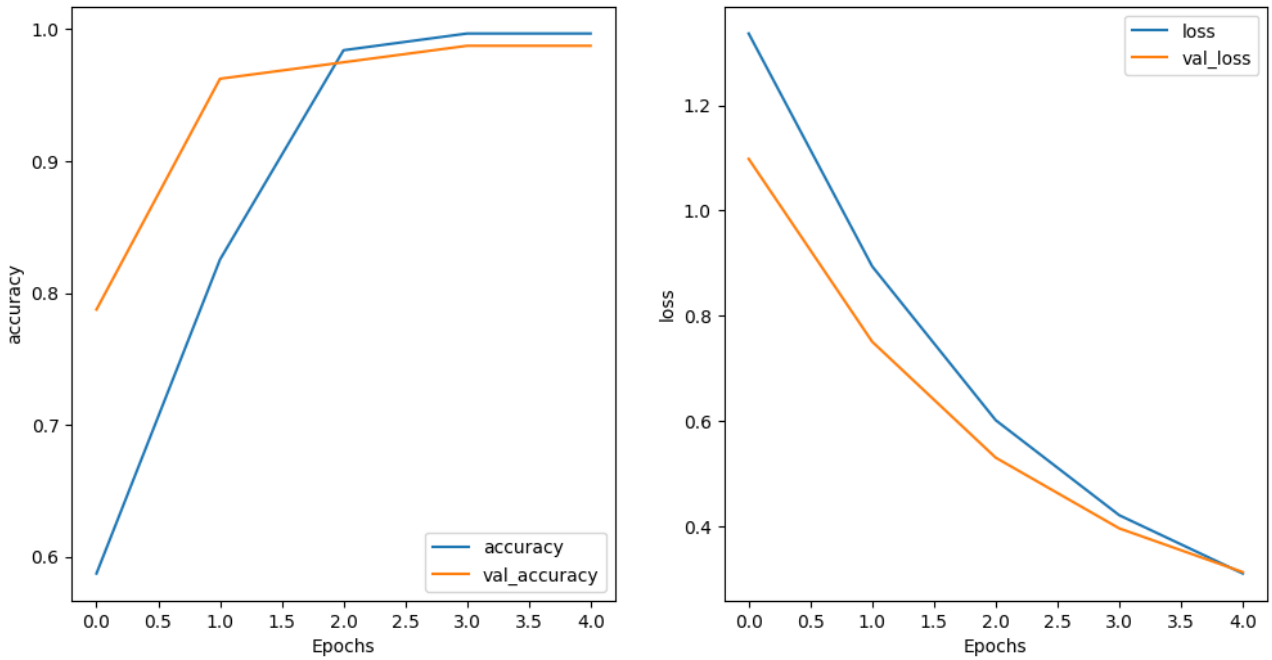

Рис. 27. Зміна тренувальних і валідаційних точності і втрат протягом 5 епох під час навчання моделі нейронної мережі на даних іншого учасника

Окрім того, для "найкращої" моделі було побудовано матрицю плутанини (рис. 28) і класифікаційний звіт (рис. 29).

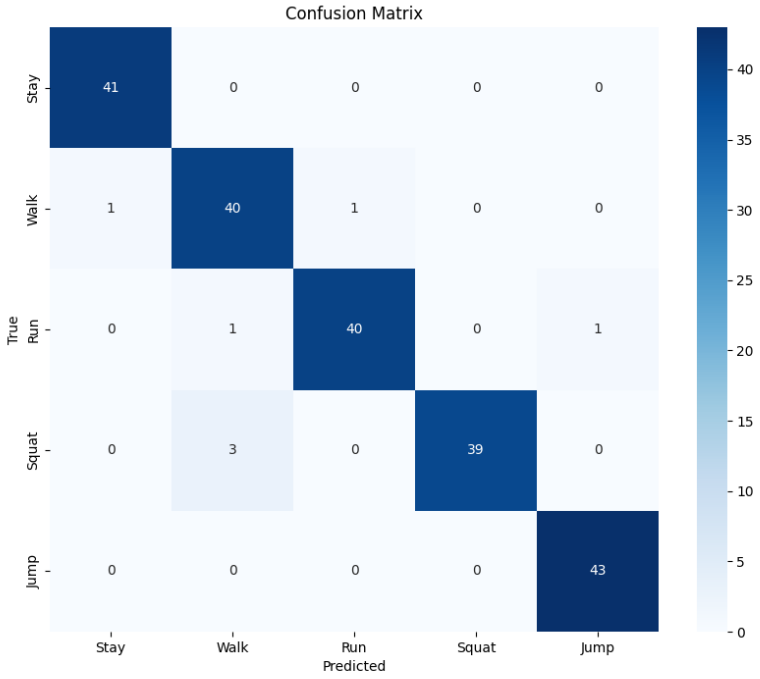

Рис. 28. Матриця плутанини "найкращої" моделі нейронної мережі

Classification Report:

	precision	recall	f1-score	support
Stay	0.976	1.000	0.988	41
Walk	0.909	0.952	0.930	42
Run	0.976	0.952	0.964	42
Squat	1.000	0.929	0.963	42
Jump	0.977	1.000	0.989	43
accuracy			0.967	210
macro avg	0.968	0.967	0.967	210
weighted avg	0.968	0.967	0.967	210

Рис. 29. Класифікаційний звіт "найкращої" моделі нейронної мережі

Розглянувши матрицю плутанини (рис. 28), можна побачити, що модель найчастіше плутала активність 'Walk' з іншими видами активності.

Висновки

Було розроблено алгоритм-пайплайн у вигляді функції perform_pipeline(), яка приймає на вхід тренувальний і тестовий датасети, а на виході видає X_train, y_train, X_valid, y_valid, X_test, y_test, які можна використовувати для тренування моделі.

Перша не оптимізована версія алгоритму, яка приймала як параметр лише тренувальний датасет, а повертала X_train, y_train, X_valid, y_valid була протестована у вебзастосунку, реалізованому з допомогою бібліотеки Streamlit. Середній час виконання цієї версії алгоритму для тренувального датасету розміром 63529 записів становив приблизно 7 с, а витрати пам’яті – приблизно 12 MiB (Mebibyte).

Внаслідок оптимізації вищезгаданої версії алгоритму-пайплайну, середній час зменшився приблизно до 4 с.

Однак після додавання до алгоритму-пайплайну ще й функціональності для опрацювання тестових даних, середній час роботи алгоритму для тренувального датасету розміром 63529 записів + тестового розміром 12557 збільшився до 8.5 с.

Алгоритм було протестовано спочатку на власних даних: максимальна досягнута точність моделі нейронної мережі становила 98.611%; а потім на датасеті іншого учасника школи: максимальна точність - 96.667%. В обидвох випадках overfitting моделі не спостерігався.

Отримані результати свідчать про доволі хорошу ефективність розробленого алгоритму-пайплайну для задач мультикласової класифікації видів фізичної активності.