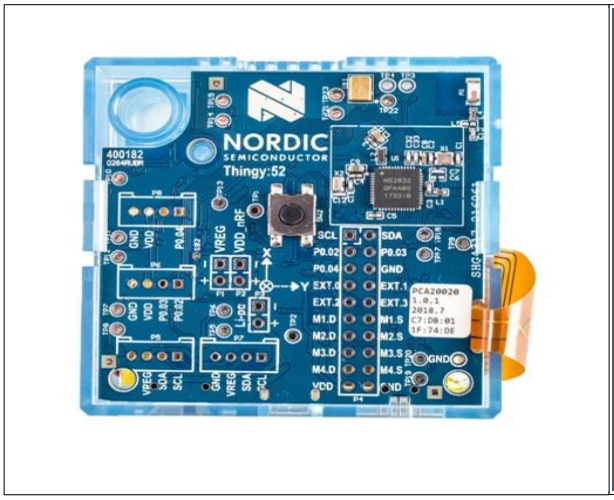
RELAZIONE PROJECT WORK DI RSDI

CONSEGNA

Il progetto consisteva nella collezione di dati provenienti da due diverse attività e, in base a queste, addestrare il programma a riconoscerle in maniera autonoma.

STRUMENTI UTILIZZATI

La collezione dei dati è stata effettuata tramite il Nordic Thingy 52, un dispositivo che supporta la connettività wireless tramite Bluetooth Low Energy (BLE) ed è a basso consumo energetico. Il Thingy è stato programmato tramite l'utilizzo di PyCharm, un IDE installato nel computer e progettato principalmente per lo sviluppo di Python.



INTRODUZIONE

Ho sviluppato un sistema per raccogliere dati provenienti da due attività: **DORMIRE** e **SCRIVERE**. La raccolta dei dati è avvenuta tramite il dispositivo **Thingy 52**, che ho fissato sul dorso della mia mano destra durante entrambe le attività. Ogni sessione di raccolta dati è durata circa **7 minuti** per ciascuna attività. In particolare, sono stati acquisiti i dati di **accelerazione** (utilizzando l'accelerometro) e di **rotazione** (tramite il giroscopio) del dispositivo, registrando le variazioni nelle **coordinate spaziali** (x, y, z) in base alla posizione del dispositivo nello spazio. Sulla base di questi dati ho addestrato il programma a riconoscere le due attività in autonomia.

DESCRIZIONE DEL CODICE

Per facilitare la comprensione del codice ho deciso di spiegarlo e dividerlo in fasi:

FASE 1: Riconoscimento del dispositivo e connessione.

Le funzioni che definiscono questa fase sono:

- **MAIN:** funzione che abilita la scansione dei vari dispositivi, la connessione del computer al Thingy e la collezione dei dati inerziali. Il Thingy è definito da un proprio indirizzo e da una classe per salvare i dati inerziali in una cartella.

```python
import asyncio
from utils.utility import scan, find
from classes import Thingy52Client


async def main():  1 usage
    my_thingy_addresses = ["FB:7B:DF:44:C3:1A"]

    discovered_devices = await scan()
    my_devices = find(discovered_devices, my_thingy_addresses)

    thingy52 = Thingy52Client.Thingy52Client(my_devices[0])
    await thingy52.connect()
    thingy52.save_to(str(input("Enter recording name: ")))
    await thingy52.receive_inertial_data()

if __name__ == '__main__':
    asyncio.run(main())
```

```python
class Thingy52Client(BleakClient):  1 usage

    def __init__(self, device: BLEDevice):
        super().__init__(get_uuid(device))
        self.path = "training/data"  # Define the directory path where
        self.mac_address = device.address

        self.buffer_size = 60
        self.data_buffer = []

        self.recording_name = None
        self.file = None

        # Ensure that the directory exists
        if not os.path.exists(self.path):
            os.makedirs(self.path)
```

- **SCAN:** qui avviene la scansione di tutti i dispositivi BLE circostanti.

```python
async def scan():  2 usages
    """
    Scan for BLE devices
    :return: A list of BLE devices found
    """
    # Scan and save the list of devices
    print(f"{datetime.datetime.now()} | Scanning...")
    devices = await BleakScanner.discover(timeout=10)

    print(f"{datetime.datetime.now()} | End scan...")

    return devices
```

- **FIND:** cerca tra tutti i dispositivi l'indirizzo del Thingy definito nel main.

```
def find(discovered_devices, addresses):  2 usages
    my_devices = []
    for i in range(len(discovered_devices)):
        if discovered_devices[i].address in addresses:
            my_devices.append(discovered_devices[i])
            # Avoid useless checks
            if len(my_devices) == len(addresses):
                break

    return my_devices
```

- **CONNECT:** qui avviene la connessione con il dispositivo. Se la connessione è avvenuta con successo allora comparirà il messaggio di connessione avvenuta. Se la connessione è fallita comparirà il messaggio di errore nella connessione.

```
async def connect(self, **kwargs) -> bool:  1 usage
    try:
        await super().connect(**kwargs)
        print(f"Successfully connected to {self.mac_address}")
        print(f"Device {self.mac_address} is now connected.")
        return True
    except Exception as e:
        print(f"Failed to connect to {self.address}: {e}")
        return False
```

I risultati che si ottengono nel terminale, se avviene tutto in maniera corretta, sono i seguenti che si trovano nella seguente immagine.

```
2025-01-12 12:03:58.836179 | Scanning...
2025-01-12 12:04:09.107072 | End scan...
Successfully connected to FB:7B:DF:44:C3:1A
Device FB:7B:DF:44:C3:1A is now connected.
Enter recording name:
```

FASE 2: Raccolta dei dati.

Questa fase si attiva quando, da terminale, compare “Enter recording name”. A questo punto attraverso il tasto “Invio” iniziamo la raccolta dei dati:

- **RECEIVE INERTIAL DATA:** gestisce la comunicazione asincrona del dispositivo, avviando la raccolta dei dati e registrandoli su un file. La funzione rimane in esecuzione fino a quando non viene annullata. In caso di annullamento si occupa di fermare la comunicazione e chiudere correttamente le risorse.
- **SAVE TO:** mi permette il salvataggio dei dati prelevati dal dispositivo nella cartella “data”.

```python
async def receive_inertial_data(self, sampling_frequency: int = 60):  1 usage
    payload = motion_characteristics(motion_processing_unit_freq=sampling_frequency)
    await self.write_gatt_char(TMS_CONF_UUID, payload)

    # Open file in the data directory
    if self.recording_name:
        self.file = open(self.recording_name, "a+")
    else:
        print("No recording name provided!")
        return

    await self.start_notify(TMS_RAW_DATA_UUID, self.raw_data_callback)

    await change_status(self,  status: "recording")

    try:
        while True:
            await asyncio.sleep(0.1)

    except asyncio.CancelledError:
        await self.stop_notify(TMS_RAW_DATA_UUID)
        print("Stopped notification")

def save_to(self, file_name):  1 usage
    # Ensure the file is being saved in the correct directory
    self.recording_name = os.path.join(self.path, f"{self.mac_address.replace( _old: ':',  _new: '-')}{file_name}.csv")
```

- **MOTION CHARACTERISTICS:** riceve una serie di parametri che definiscono le caratteristiche di funzionamento del sensore di movimento. Questi parametri vengono impacchettati in un formato binario ben definito.

```python
def motion_characteristics(  3 usages
        step_counter_interval=100,
        temperature_comp_interval=100,
        magnetometer_comp_interval=100,
        motion_processing_unit_freq=60,
        wake_on_motion=1
):
    format_str = "<4H B"

    return struct.pack(format_str,
                       *v: step_counter_interval,
                       temperature_comp_interval,
                       magnetometer_comp_interval,
                       motion_processing_unit_freq,
                       wake_on_motion)
```

- **CHANGE STATUS:** Cambia il colore LED del dispositivo. In particolare in stato di “connected” sarà verde, mentre durante la collezione dati sarà rosso.

```python
async def change_status(client, status: str):  4 usages
    """
    Change the color of the LED on the Thingy52
    :param client: The client object
    :param status: The status to change to
    :return: None
    """
    payload = None

    if status == "connected":
        format_str = "<4B"
        constant_light = 1
        green = (0, 255, 0)
        payload = struct.pack(format_str,  *v: constant_light, *green)

    if status == "recording":
        format_str = "<4B"
        constant_light = 1
        red = (255, 0, 0)
        payload = struct.pack(format_str,  *v: constant_light, *red)

    if payload is not None:
        # Write the color data to the LED characteristic
        await client.write_gatt_char(UIS_LED_UUID, payload)
```

- **RAW DATA CALLBACK:** raccoglie, registra e monitora i dati di movimento e orientamento del dispositivo in tempo reale.

```python
def raw_data_callback(self, sender, data):  1 usage
    receive_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")

    # Accelerometer
    acc_x = (struct.unpack( format: 'h', data[0:2])[0] * 1.0) / 2 ** 10
    acc_y = (struct.unpack( format: 'h', data[2:4])[0] * 1.0) / 2 ** 10
    acc_z = (struct.unpack( format: 'h', data[4:6])[0] * 1.0) / 2 ** 10

    # Gyroscope
    gyro_x = (struct.unpack( format: 'h', data[6:8])[0] * 1.0) / 2 ** 5
    gyro_y = (struct.unpack( format: 'h', data[8:10])[0] * 1.0) / 2 ** 5
    gyro_z = (struct.unpack( format: 'h', data[10:12])[0] * 1.0) / 2 ** 5

    # Compass
    comp_x = (struct.unpack( format: 'h', data[12:14])[0] * 1.0) / 2 ** 4
    comp_y = (struct.unpack( format: 'h', data[14:16])[0] * 1.0) / 2 ** 4
    comp_z = (struct.unpack( format: 'h', data[16:18])[0] * 1.0) / 2 ** 4
    # Write data to file
    if self.file:
        self.file.write(f"{receive_time},{acc_x},{acc_y},{acc_z},{gyro_x},{gyro_y},{gyro_z}\n")
    # Manage buffer size and add new data
    self.data_buffer = {"x": [], "y": [], "z": []}
    self.data_buffer["x"].append(acc_x)
    self.data_buffer["y"].append(acc_y)
    self.data_buffer["z"].append(acc_z)
    print(f"\r{self.mac_address} | {receive_time} - Accelerometer: X={acc_x: 2.3f}, Y={acc_y: 2.3f}, Z={acc_z: 2.3f}",
          end="", flush=True)
```

I risultati che si ottengono nel terminale, se avviene tutto in maniera corretta, sono i seguenti che si trovano nella seguente immagine.

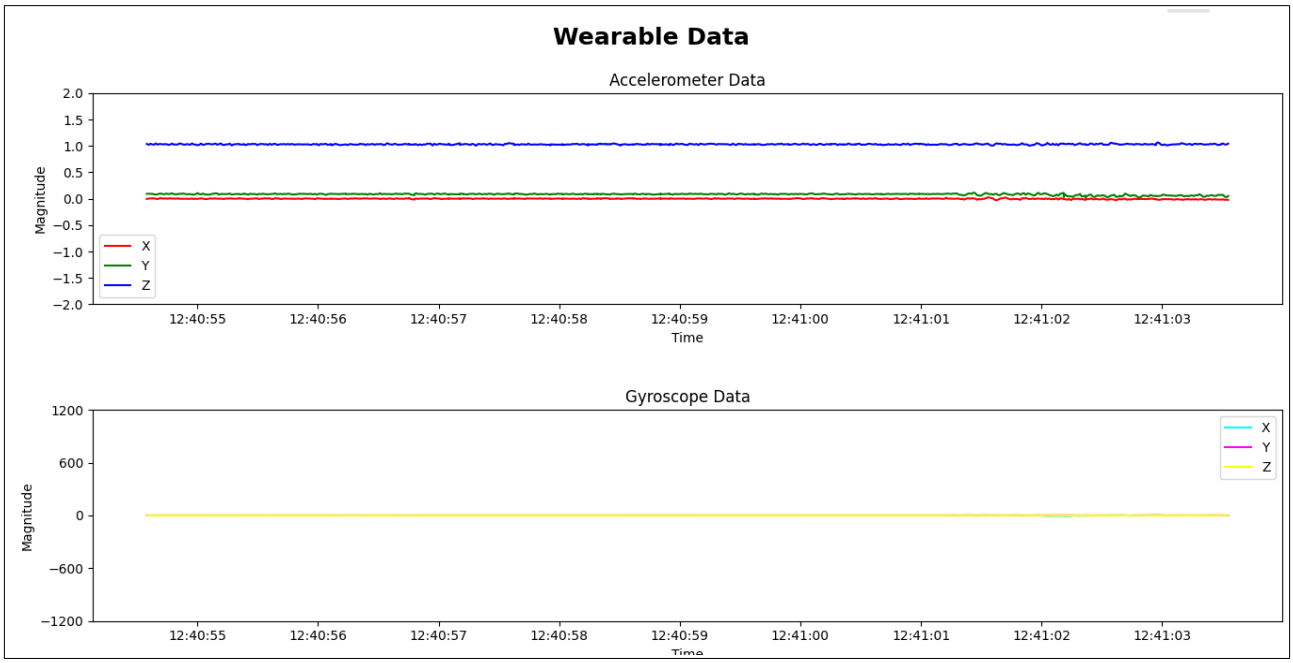
FASE 3: Analisi grafica dei dati.

- **ANIMATE:** aggiorna continuamente i grafici in base ai dati raccolti dai sensori (accelerometro e giroscopio). I dati vengono letti da un file CSV, elaborati e visualizzati in tempo reale, consentendo di tracciare le variazioni nel tempo delle letture sui vari assi dei sensori.
- **LIVE PLOTTING:** avvia animazione dei grafici, aggiornando i dati ad intervalli regolari di 60 millisecondi

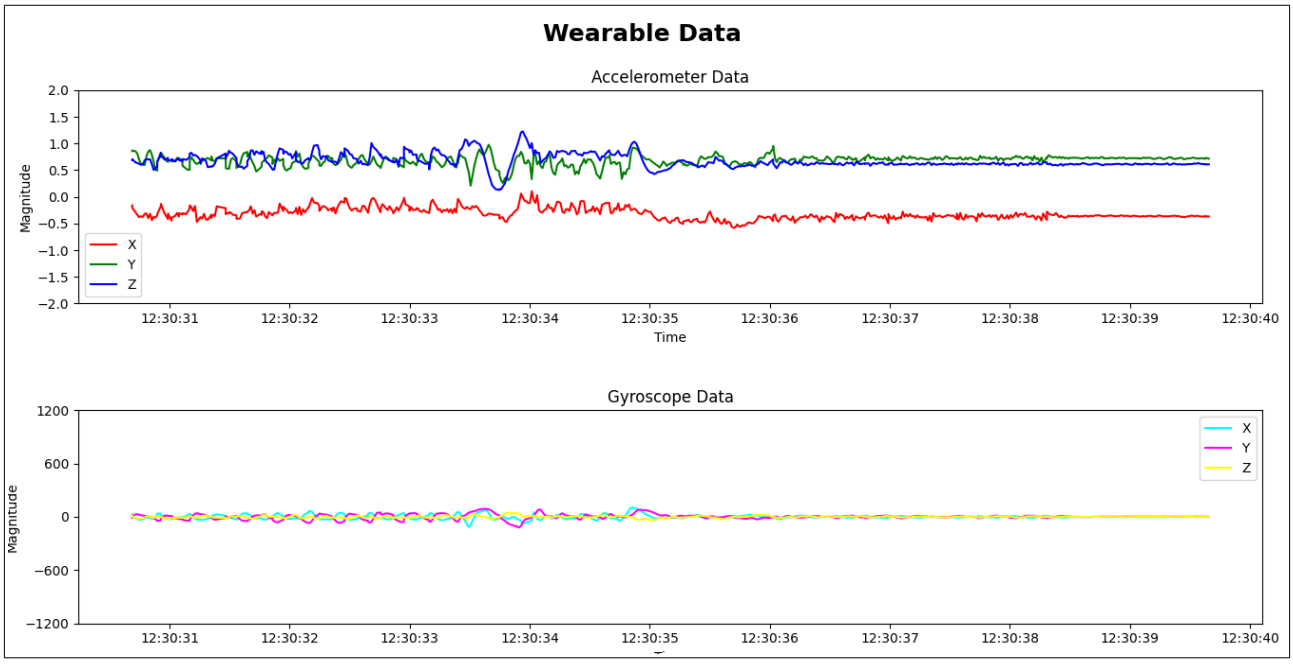
```python
data_directory = "training/data"  # Path to the 'data' directory within 'training'
recording_name = os.path.join(data_directory, "FB-7B-DF-44-C3-1A.csv")  # Full file path
skip_rows = 0

# Create the figure for plotting
fig, (accelerometer_fig, gyroscope_fig) = plt.subplots( nrows: 2,  ncols: 1, figsize=(16, 10))

def animate(i):  1 usage
    global skip_rows, recording_name
    if not os.path.exists(recording_name):
        print(f"File not found: {recording_name}")
        return
     # Read the CSV file into a DataFrame
    df = pd.read_csv(
        recording_name,
        header=None,
        names=["time", "acc_x", "acc_y", "acc_z", "gyro_x", "gyro_y", "gyro_z"],
        index_col=0,
        parse_dates=True,
        skiprows=skip_rows
    )
     # Adjust skip_rows to only keep the last 600 rows
    if len(df) >= 600:
        skip_rows += len(df) - 600
     # Clear the figures for updating
    accelerometer_fig.clear()
    gyroscope_fig.clear()
     # Plot Accelerometer Data
    accelerometer_fig.plot(df["acc_x"], color="red", label="X")
    accelerometer_fig.plot(df["acc_y"], color="green", label="Y")
    accelerometer_fig.plot(df["acc_z"], color="blue", label="Z")
    accelerometer_fig.set_yticks(np.arange(-2, 2.5, 0.5))
    accelerometer_fig.set_ylabel("Magnitude")
    accelerometer_fig.set_xlabel("Time")
    accelerometer_fig.set_title("Accelerometer Data")
    accelerometer_fig.legend()
    # Plot Gyroscope Data
    gyroscope_fig.plot(df["gyro_x"], color="cyan", label="X")
    gyroscope_fig.plot(df["gyro_y"], color="magenta", label="Y")
    gyroscope_fig.plot(df["gyro_z"], color="yellow", label="Z")
    gyroscope_fig.set_xlabel("Time")
    gyroscope_fig.set_yticks(np.arange(-1200, 1800, 600))
    gyroscope_fig.set_ylabel("Magnitude")
    gyroscope_fig.set_title("Gyroscope Data")
    gyroscope_fig.legend()
     # Update the title and adjust layout
    fig.suptitle( t: f"Wearable Data", fontsize=18, fontweight="bold")
    plt.tight_layout()
    plt.subplots_adjust(hspace=0.5, top=0.88)

def live_plotting():  2 usages
    # Set up animation to continuously update the plots
    ani = FuncAnimation(fig, animate, interval=60, cache_frame_data=False)
    plt.show()
```

Queste funzioni sono chiamate da un main e questo funziona separato rispetto al programma di collezione dei dati. Ha prodotto i seguenti risultati:

SLEEPING



WRITING



FASE 4: Training del programma e creazione matrici di confusione.

Questa fase comprende due classi:

- **CSV DATA MODULE:** è un modulo utile per lavorare con dati temporali provenienti dal Thingy, con la possibilità di eseguire una valutazione tramite k-fold cross-validation. Nel dettaglio, questo modulo:

  - Carica dati da file CSV contenenti i dati temporali di accelerometro e giroscopio attraverso la funzione **CREATE DATASET**;

```
def create_dataset(self):  1 usage
    x, y = [], []
    for file in os.listdir(self.root_dir):
        if file.endswith(".csv"):
            df = pd.read_csv(
                os.path.join(self.root_dir, file),
                index_col=0,
                header=None,
                names=["time", "acc_x", "acc_y", "acc_z", "gyro_x", "gyro_y", "gyro_z"]
            )
            df["label"] = file.split("_")[1].split(".")[0]
            x_i, y_i = self.windowing(df)
            x.extend(x_i)
            y.extend(y_i)

    x, y = np.array(x), np.array(y)

    # Label encoding
    self.legend, y = self.labels_encoding(y)

    return x, y
```

  - Esegue il **WINDOWING** dei dati per creare sequenze temporali etichettate;

```
def windowing(self, data):  1 usage
    windows, labels = [], []

    for i in range(0, len(data) - self.window_size * self.sample_rate, int(self.overlap * self.sample_rate)):
        window = data.iloc[i:i + self.window_size * self.sample_rate]
        label = window["label"].iloc[0]
        window = window.drop(columns=["label"])
        windows.append(window.values)
        labels.append(label)

    return np.array(windows), np.array(labels)
```

- Esegue la **CROSS-VALIDATION** suddividendo i dati in **K-FOLD** attraversola funzione **SETUP**;

```
class CSVDataModule(pl.LightningDataModule):  2 usages
    def __init__(self, root_dir, batch_size=32, num_workers=4, k_folds=3, window_size=1, overlap=1, sample_rate=60):
        super().__init__()
        self.root_dir = root_dir
        self.batch_size = batch_size
        self.num_workers = num_workers
        self.k = k_folds

        self.window_size = window_size
        self.overlap = overlap
        self.sample_rate = sample_rate

        self.legend = None
        self.x, self.y = None, None

        self.train_indices = []
        self.val_indices = []


    def setup(self, stage=None):  1 usage
        self.x, self.y = self.create_dataset()
        k_fold = KFold(n_splits=self.k, shuffle=True, random_state=42)

        for train_idx, val_idx in k_fold.split(self.x):
            self.train_indices.append(train_idx)
            self.val_indices.append(val_idx)
```

- Codifica le etichette come **ONE-HOT** attraverso la funzione **LABELS ENCODING**;

```
@staticmethod  1 usage
def labels_encoding(y):
    categories, inverse = np.unique(y, return_inverse=True)

    # Create the one-hot encoded matrix
    one_hot = np.zeros((y.size, categories.size))
    one_hot[np.arange(y.size), inverse] = 1

    return categories, one_hot.astype(np.float64)

def prepare_dataset(self, indexes, fold):  2 usages
    x = torch.tensor(self.x[indexes[fold]], dtype=torch.float32)
    y = torch.tensor(self.y[indexes[fold]], dtype=torch.float32)
    return torch.utils.data.TensorDataset( *tensors: x, y)
```

- Fornisce i dati tramite **DATALOADER** per addestramento e validazione.

```
def train_dataloader(self, fold=0):  1 usage
    return DataLoader(
        self.prepare_dataset(self.train_indices, fold, ),
        batch_size=self.batch_size,
        num_workers=self.num_workers,
        shuffle=True,
        persistent_workers=True
    )

def val_dataloader(self, fold=0):  2 usages
    return DataLoader(
        self.prepare_dataset(self.val_indices, fold, ),
        batch_size=self.batch_size,
        num_workers=self.num_workers,
        shuffle=False,
        persistent_workers=True
    )
```

- **CNN:** è un modello di rete neurale. Gestisce l'addestramento, la validazione e il test, calcolando la perdita e le metriche come la precisione e il f1-score. Inoltre, include il salvataggio dei modelli migliori tramite checkpoint e la generazione di matrici di confusione alla fine del test.

```
class CNN(pl.LightningModule):  3 usages

    def __init__(self, input_dim, fold, classes_names, output_dim=2, learning_rate=1e-3):
        super(CNN, self).__init__()
        self.name = "CNN"
        self.classes_labels = classes_names
        self.fold = fold
        self.classes = output_dim

        self.val_predictions = []
        self.val_targets = []
        self.test_predictions = []
        self.test_targets = []

        self.learning_rate = learning_rate
        self.loss_function = nn.CrossEntropyLoss()

        self.input_dim = 6
        self.dim = 64
        self.filter_size = 3
        self.window_size = input_dim
```

```
        # Convolution Branch
        self.conv1 = nn.Sequential(
            nn.Conv1d(self.window_size, self.dim, self.filter_size),
            nn.BatchNorm1d(self.dim),
            nn.PReLU(),
            nn.MaxPool1d(2),
            nn.Dropout(p=0.1)
        )  # (_, 64, 2)

        self.fc1 = nn.Sequential(
            nn.Linear(self.dim * 2,  out_features: 128),
            nn.PReLU(),
            nn.Dropout(p=0.3),
            nn.Linear( in_features: 128, output_dim),
            nn.Softmax(dim=-1)
        )

    def id(self):  2 usages
        return f"{self.name}_{self.window_size}"
```

```
def forward(self, x):
    # input: (batch size, d_model, length)
    # x = x.view(-1, self.input_dim, self.window_size)
    x = self.conv1(x)

    x = x.view(x.size(0), -1)
    x = self.fc1(x)

    return x

def training_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(x)
    loss = self.compute_loss(y_hat, y)
    # Log training loss
    self.log( name: "train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
    return loss

def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(x)
    val_loss = self.compute_loss(y_hat, y)
    self.log( name: "val_loss", val_loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
     # Collect predictions and targets
    self.val_predictions.append(np.argmax(y_hat.cpu().numpy(),  axis: 1))
    self.val_targets.append(np.argmax(y.cpu().numpy(),  axis: 1))
    return val_loss
```

```python
def on_validation_epoch_end(self):
    # Concatenate all predictions and targets from this epoch
    val_predictions = np.concatenate(self.val_predictions)
    val_targets = np.concatenate(self.val_targets)

    # Log or print confusion matrix and classification report
    precision = precision_score(val_targets, val_predictions, average='macro', zero_division=0)
    f1 = f1_score(val_targets, val_predictions, average='macro', zero_division=0)
    self.log( name: "prec_macro", precision, on_step=False, on_epoch=True, prog_bar=True, logger=True)
    self.log( name: "f1_score", f1, on_step=False, on_epoch=True, prog_bar=True, logger=True)

    # Clear stored values for the next epoch
    self.val_predictions.clear()
    self.val_targets.clear()

def test_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(x)
    test_loss = self.compute_loss(y_hat, y)
    self.log( name: "test_loss", test_loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)

    # Collect predictions and targets
    self.test_predictions.append(np.argmax(y_hat.cpu().numpy(), axis: 1))
    self.test_targets.append(np.argmax(y.cpu().numpy(), axis: 1))

    return test_loss
def on_test_end(self):
    output_path = f"output/{self.id()}"
    Path(output_path).mkdir(parents=True, exist_ok=True)

    test_predictions = np.concatenate(self.test_predictions)
    test_target = np.concatenate(self.test_targets)
    cm_analysis(
        test_target,
        test_predictions,
        filename: f"{output_path}/confusion_matrix_segments_fold_{self.fold}",
        range(self.classes),
        self.classes_labels,
        specific_title=f"Segments: {self.id()} fold {self.fold}"
    )
    self.fold += 1

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate, weight_decay=1e-6)
    return optimizer

def compute_loss(self, y_hat, y):  3 usages
    return self.loss_function.forward(y_hat, y)
```

Queste classi sono richiamate opportunamente da una funzione **MAIN** che gestisce il processo di k-fold cross-validation per addestrare e testare modelli CNN su dati letti da file CSV. Per ogni fold:

- viene creato un modello CNN con i dati del fold attuale;

```python
def main():  1 usage
    path = "data"
    k = 3
    window_size = 1
    data_module = CSVDataModule(
        root_dir=path,
        batch_size=32,
        k_folds=k,
        window_size=window_size, # seconds
    )

    data_module.setup()
```

- viene impostato un callback di checkpoint per salvare il miglior modello in base alla perdita di validazione;

```python
# Perform k-fold cross-validation
for fold in range(data_module.k):
    print(f'Fold {fold + 1}/{data_module.k + 1}')

    model = CNN(window_size * 60, fold + 1, classes_names=data_module.legend)

    # Define checkpoint callback to save the best model for each fold
    checkpoint_callback = ModelCheckpoint(
        monitor='val_loss',
        dirpath=f'checkpoints/fold_{fold}',
        filename='best-checkpoint',
        save_top_k=1,
        mode='min'
    )

    trainer = Trainer(
        max_epochs=20,
        accelerator="cpu",
        devices=1,
        logger=True,
        log_every_n_steps=10,
        callbacks=[checkpoint_callback]
    )
```

- il modello viene addestrato con il trainer di PyTorch Lightning;

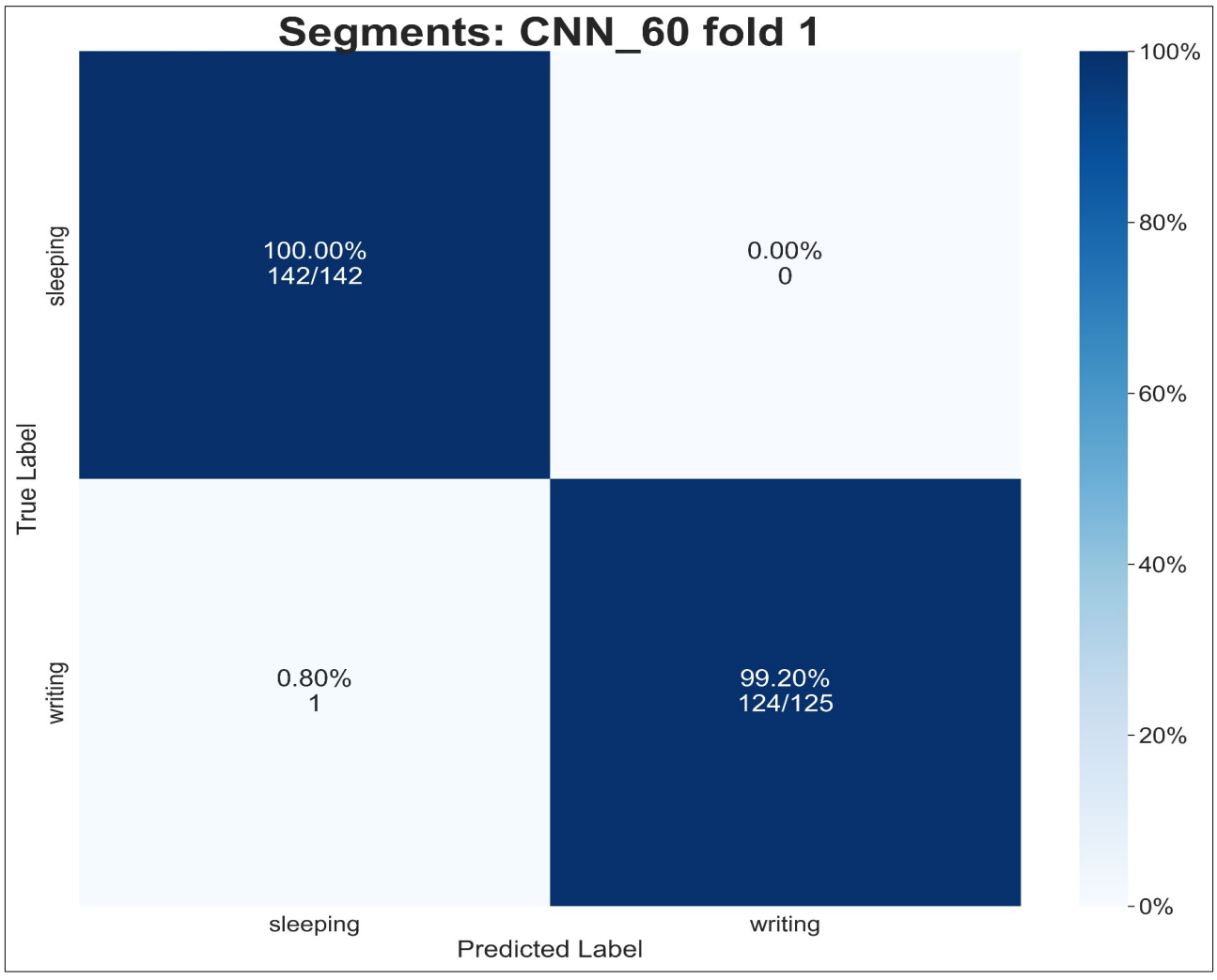
```
# Train the model
trainer.fit(model, data_module.train_dataloader(fold=fold), data_module.val_dataloader(fold=fold))
```

- al termine dell'addestramento, il modello viene testato sui dati di validazione.

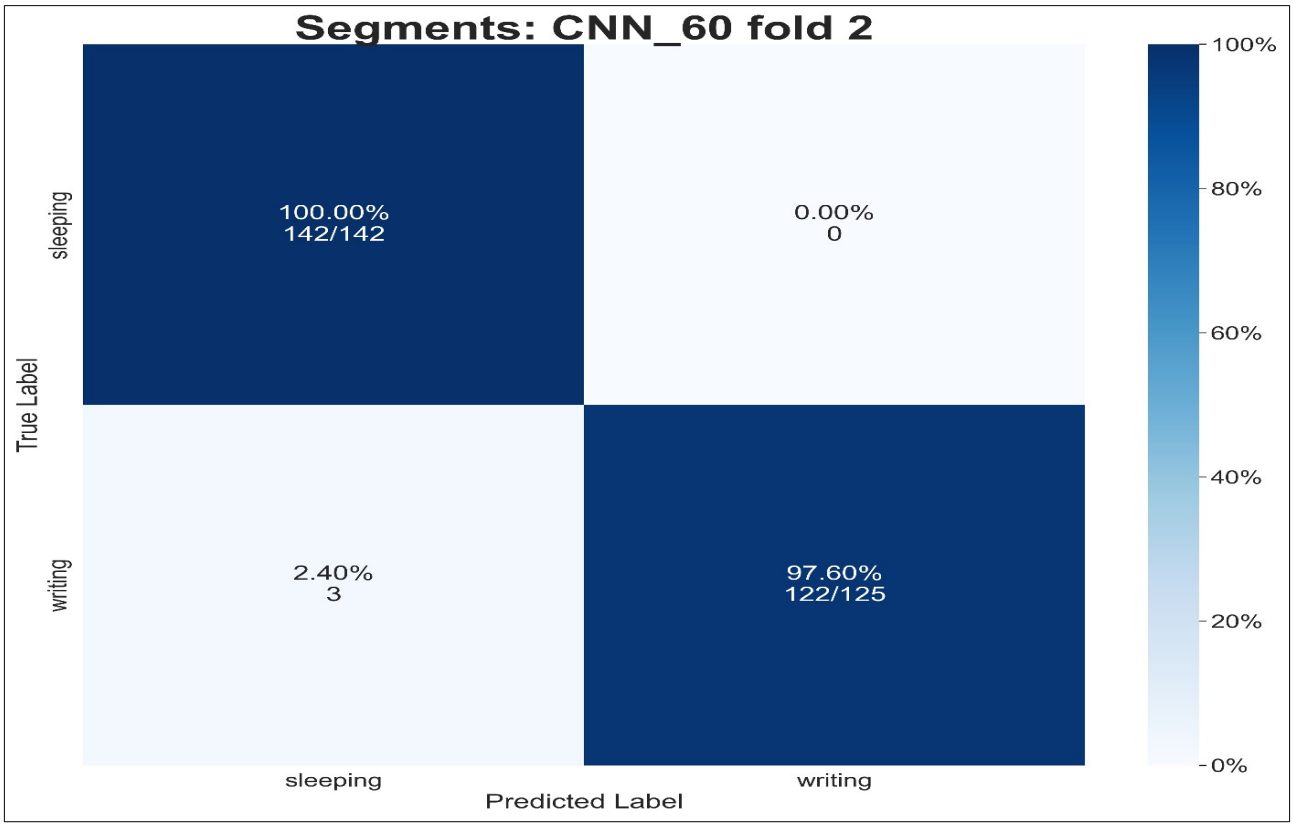
```
# Test the model
trainer.test(model, data_module.val_dataloader(fold=fold))
```

I risultati che si ottengono da questo programma sono la creazione di matrici di confusione aventi tre differenti risultati sul riconoscimento delle due attività.

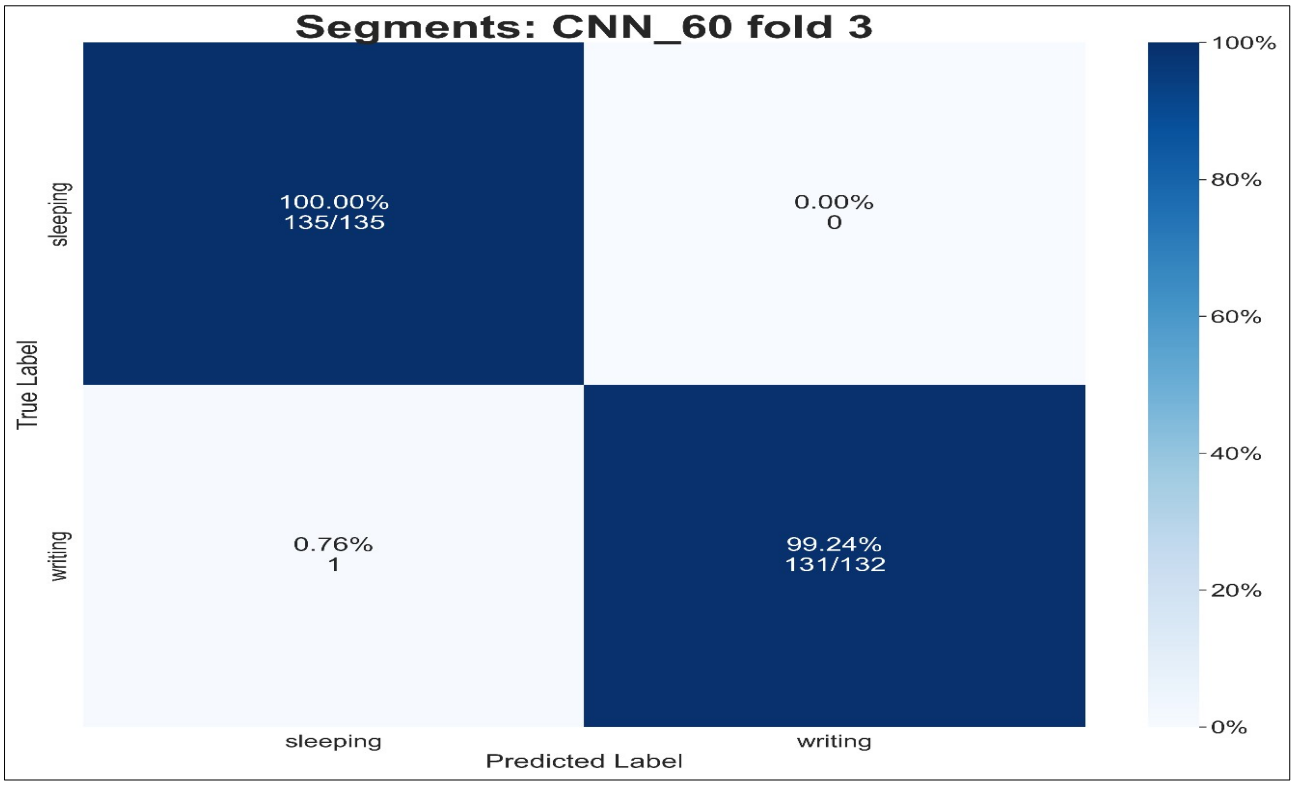
RISULTATO 1:



RISULTATO 2:



RISULTATO 3:



FASE 5: Conversione modello ckpt in onnx e riconoscimento delle attività.

La conversione viene fatta attraverso due funzioni principali:

- **LOAD MODEL ():** questa funzione carica un modello pre-addestrato dal file checkpoint, che contiene i pesi e la configurazione del modello.

```
def load_model(fold):  1 usage
    model_path = f"checkpoints/{fold}/best-checkpoint.ckpt"
    model = CNN.load_from_checkpoint(
        cls: f"{model_path}",
        input_dim=60,
        fold=fold,
        classes_names=["sleeping", "writing"]
    ).to("cpu")
    model.eval()
    return model
```

- **TORCH TO ONNX ():** converte il modello Pytorch in formato ONNX.

```
def torch_to_onnx(model, dummy_input):
    torch.onnx.export(
        model,
        dummy_input,
        f: f"{model.id()}.onnx",
        export_params=True,
        verbose=True,
        opset_version=10
    )
```

L’esecuzione del programma viene effettuata attraverso la funzione main().

```
def main():  1 usage
    info = "fold_1"
    dummy_input = torch.randn(1, 60, 6)
    model = load_model(info)
    torch_to_onnx(model, dummy_input)

if __name__ == '__main__':
    main()
```

Il riconoscimento delle attività viene fatto sulla base del modello ONNX, attraverso la fase di connessione e raccolta dei dati del dispositivo. Il risultato a terminale è il seguente:

```
2025-02-05 09:22:53.333752 | Scanning...
2025-02-05 09:23:03.378036 | End scan...
Successfully connected to FB:7B:DF:44:C3:1A
Device FB:7B:DF:44:C3:1A is now connected.
Enter recording name:
FB:7B:DF:44:C3:1A | 2025-02-05 09:24:08.039656 - Prediction: sleeping
```

LIBRERIE UTILIZZATE

Le librerie installate per l'implementazione del codice sono le seguenti:

- **ASYNCIO:** usata per la gestione asincrona delle operazioni come la comunicazione con dispositivi BLE.
- **NUMPY:** Usata per la manipolazione di array e operazioni numeriche.
- **BLEAK:** Gestisce la comunicazione con dispositivi BLE.
- **PANDAS:** Usata per la manipolazione dei dati in Python. Utilizzata per caricare, pulire, esplorare e trasformare i dati in formato CSV, Excel e altro.
- **SCIKIT-LEARN:** Utilizzata per l'apprendimento automatico. Include funzioni di pre-processing dei dati, come normalizzazione, encoding delle etichette e divisione di dati in set di allenamento e test.
- **TORCH:** Utilizzata per facilitare la costruzione, l'addestramento e l'inferenza di modelli di deep learning.
- **LIGHTNING:** Utilizzata per la gestione di training, validazione e test in modo più organizzato e più facilmente scalabile.
- **SEABORN e MATPLOTLIB:** librerie per la visualizzazione dei dati. Permette di creare grafici 2D o 3D e utile per la visualizzazione di dati statistici

- **ONNX:** Mi permette di convertire i modelli e lavorare con modelli pre-allenati.
- **ONNX RUNTIME:** motore di esecuzione che esegue modelli ONNX in modo ottimizzato e ad alte prestazioni.

RISULTATI OTTENUTI E CONSIDERAZIONI

Ho addestrato il programma su due attività che sono ben distinte poiché una è statica (dormire) e una è più dinamica (scrivere). I risultati ottenuti dalle matrici di confusione sembrano andare bene poiché le due attività sono abbastanza riconosciute dal programma. La predizione delle due attività, durante la raccolta dei dati, invece, presenta qualche problema nel riconoscimento delle attività. Ad esempio mentre si è seduti e fermi rileva che sto dormendo oppure mentre muovo il polso rileva che sto scrivendo. Quindi il programma è migliorabile su diversi aspetti:

- Nella raccolta dati si potrebbe raccogliere un altro set di dati chiamato “altre attività” in modo da raccogliere dati diversi rispetto alle due attività da analizzare.
- Si potrebbe così determinare una matrice di confusione 3 x 3 con le due attività + l’altro set di dati di altre attività.
- Nel riconoscere le attività durante la raccolta dati se si sta facendo qualcosa di diverso rispetto alle altre attività il programma dovrebbe riuscire a predire “altre attività”.