Students:

Vincent Roduit
Caspar Henking
Fabio Palmisano
Bastien Marconato

ecg_rhythm_classification

December 12, 2024

1 ECG Rhythm Classification

The goal of this exercise is to train a neural network model to classify different cardiac rhythm from single-lead ECG signals. The ECG signals we will use are a subset of the large scale 12-lead electrocardiogram database for arrhythmia study (https://physionet.org/content/ecg-arrhythmia/1.0.0/).

The subset includes the following cardiac rhythms:

- Atrial fibrillation
- Atrial flutter
- Normal sinus rhythm
- Sinus bradycardia
- Sinus tachycardia

There are 1500 single-lead ECG signals (lead II) for each rhythm.

First, we import all required packages, define global constants, and seed the random number generators to obtain reproducible results.

[1]:
```
%matplotlib widget


import collections
import itertools
import logging
import operator
import pathlib
import warnings

import IPython.display
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pytorch_lightning as pl
import sklearn.metrics
import sklearn.model_selection
import sklearn.preprocessing
import torch
```

```
DATA_FILE = pathlib.Path('../data/ecg_rhythms.npz')
LOG_DIRECTORY = pathlib.Path('../logs/ecg_rhythm_classification')


# Disable logging for PyTorch Lightning to avoid too long outputs.
logging.getLogger('pytorch_lightning').setLevel(logging.ERROR)

# Seed random number generators for reproducible results.
pl.seed_everything(42)
```

```
Seed set to 42
```

[1]:
```
42
```

Then, we load the ECG signals and the corresponding rhythm annotations.

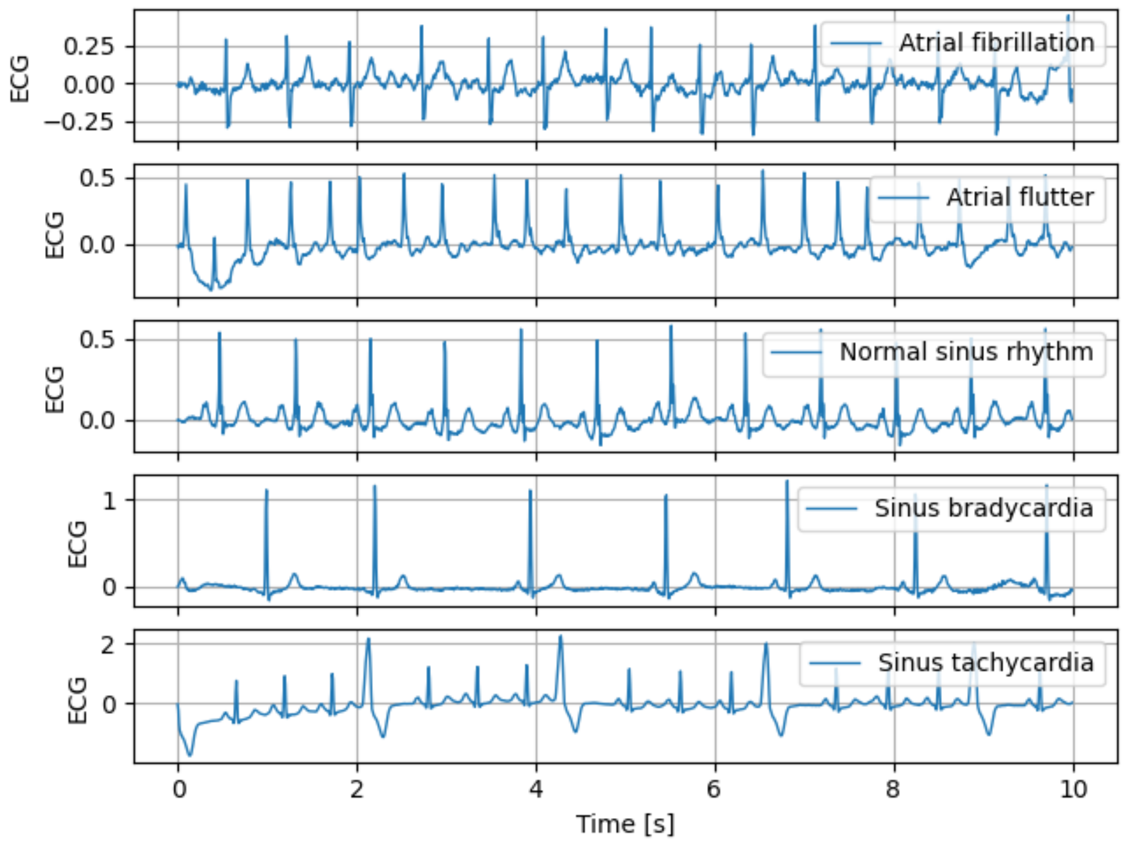
[2]:
```
def load_data():
    with np.load(DATA_FILE) as data:
        signals = data['signals']
        rhythms = data['rhythms']
        fs = data['fs'].item()
    return signals, rhythms, fs


signals, rhythms, fs = load_data()

IPython.display.display(pd.DataFrame(sorted(collections.Counter(rhythms).
 ↪items()), columns=['rhythm', 'count']))
```

	rhythm	count
0	atrial_fibrillation	1500
1	atrial_flutter	1500
2	normal_sinus_rhythm	1500
3	sinus_bradycardia	1500
4	sinus_tachycardia	1500

Here are a few examples of ECG signals.

[3]:
```
def plot_ecg_examples(signals, rhythms, fs):
    time = np.arange(signals.shape[-1]) / fs
    labels = np.unique(rhythms)
    indices = [np.random.choice(np.flatnonzero(rhythms == label)) for label in␣
 ↪labels]

    fig, axes = plt.subplots(len(labels), 1, sharex='all',␣
 ↪constrained_layout=True)
    for ax, index in zip(axes.flat, indices):
        label = rhythms[index].replace('_', ' ').capitalize()
        plt.sca(ax)
```

```
        plt.plot(time, signals[index].T, linewidth=1, label=label)
        plt.grid()
        plt.ylabel('ECG')
        plt.legend(loc='upper right')
    plt.xlabel('Time [s]')


plot_ecg_examples(signals, rhythms, fs)
```



Question 1

Visually, what are the differences between the different rhythms?

Answer

Atrial fibrillation, as it has been shown many times in the other lab sessions, shows a lot of irregularities in its PR-intervals and a shift of R and S points : instead of being respectively around 0.5 and -0.1, their values are around 0.25 and -0.25. There is also an increase in heartbeat frequency.

The *atrial flutter* signal shows a huge increase in heartbeat frequency (much bigger than the one in atrial fibrillation, nearly twice the one of the normal signal).

Sinus bradycardia is the opposite and shows a decrease in frequency. There is also an increase in amplitude of the R-peaks.

Sinus tachycardia has a signal frequency slightly lower than the *atrial flutter* one, with one every four PQRST complexes showing really extreme behaviours (extreme amplitudes of R and S points).

Then, we split that data into subsets for training, validation, and testing stratified by rhythms.

[4]:
```
def split_data(rhythms):
    n = rhythms.size
    splitter = sklearn.model_selection.StratifiedKFold(n_splits=5)
    indices = list(map(operator.itemgetter(1), splitter.split(np.zeros((n, 1)),␣
↪rhythms)))
    i_train = np.hstack(indices[:-2])
    i_val = indices[-2]
    i_test = indices[-1]

    assert np.intersect1d(i_train, i_val).size == 0
    assert np.intersect1d(i_train, i_test).size == 0
    assert np.intersect1d(i_val, i_test).size == 0
    assert np.all(np.sort(np.hstack((i_train, i_val, i_test))) == np.arange(n))

    return i_train, i_val, i_test


i_train, i_val, i_test = split_data(rhythms)


def build_summary(rhythms, indices):
    labels = np.unique(rhythms)
    data = []
    for subset, i in indices:
        y = rhythms[i]
        data.append({'subset': subset, 'total_count': y.size})
        for label in labels:
            data[-1][f'{label}_count'] = np.sum(y == label)
    return pd.DataFrame(data)


IPython.display.display(build_summary(rhythms, (('train', i_train), ('val',␣
↪i_val), ('test', i_test))))
```

```
  subset  total_count  atrial_fibrillation_count  atrial_flutter_count  \
0  train         4500                        900                   900
1    val         1500                        300                   300
2   test         1500                        300                   300

   normal_sinus_rhythm_count  sinus_bradycardia_count  sinus_tachycardia_count
0                        900                      900                      900
1                        300                      300                      300
```

2	300	300	300

The final preprocessing steps are to scale the ECG signals so that they have approximiately unit variance and to encode the rhythm labels with one-hot encoding.

[5]:
```
def compute_scaling(signals):
    sigma = np.std(signals)
    return 1.0 / sigma


alpha = compute_scaling(signals[i_train])
signals *= alpha


def encode_rhythms(rhythms):
    categories = [np.unique(rhythms)]
    encoder = sklearn.preprocessing.OneHotEncoder(categories=categories,␣
 ↪sparse_output=False)
    return encoder.fit_transform(rhythms[:, None])


encoded_rhythms = encode_rhythms(rhythms)


def print_encoded_rhythms(rhythms, encoded_rhythms, n=10):
    df = pd.DataFrame(encoded_rhythms, columns=np.unique(rhythms))
    df.insert(0, 'rhythm', rhythms)
    IPython.display.display(df.head(n))


print_encoded_rhythms(rhythms, encoded_rhythms)
```

```
                 rhythm  atrial_fibrillation  atrial_flutter  \
0  atrial_fibrillation                  1.0             0.0
1    sinus_bradycardia                  0.0             0.0
2    sinus_bradycardia                  0.0             0.0
3       atrial_flutter                  0.0             1.0
4    sinus_bradycardia                  0.0             0.0
5  atrial_fibrillation                  1.0             0.0
6  normal_sinus_rhythm                  0.0             0.0
7    sinus_bradycardia                  0.0             0.0
8    sinus_bradycardia                  0.0             0.0
9    sinus_bradycardia                  0.0             0.0

   normal_sinus_rhythm  sinus_bradycardia  sinus_tachycardia
0                  0.0                0.0                0.0
1                  0.0                1.0                0.0
2                  0.0                1.0                0.0
```

```
3                   0.0                 0.0                 0.0
4                   0.0                 1.0                 0.0
5                   0.0                 0.0                 0.0
6                   1.0                 0.0                 0.0
7                   0.0                 1.0                 0.0
8                   0.0                 1.0                 0.0
9                   0.0                 1.0                 0.0
```

We define a class and few utility functions for training and evaluating models.

[6]:
```python
class Classifier(pl.LightningModule):

    def __init__(self, model, learning_rate=0.001):
        super().__init__()
        self.save_hyperparameters(ignore=['model'])
        self.model = model
        self.learning_rate = learning_rate
        self.example_input_array = torch.zeros((1,) + self.model.input_shape)

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=self.learning_rate)

    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        return self._run_step(batch, 'train')

    def validation_step(self, batch, batch_idx):
        self._run_step(batch, 'val')

    def test_step(self, batch, batch_idx):
        self._run_step(batch, 'test')

    def predict_step(self, batch, batch_idx, dataloader_idx=0):
        x, y = batch
        return self.model(x)

    def _run_step(self, batch, subset):
        x, y = batch
        logits = self.model(x)
        loss = torch.nn.functional.cross_entropy(logits, y)
        acc = (torch.argmax(y, 1) == torch.argmax(logits, 1)).float().mean()
        self.log_dict({
            f'{subset}_loss': loss,
            f'{subset}_acc': acc,
        }, on_step=False, on_epoch=True, prog_bar=True)
        return loss
```

```python
def build_loader(*tensors, batch_size=100, shuffle=False, n_workers=0):
    dataset = torch.utils.data.TensorDataset(*map(torch.Tensor, tensors))
    return torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        num_workers=n_workers,
    )


def train_model(name, model, x, y, i_train, i_val, learning_rate=0.001, batch_size=100, n_epochs=10):
    train_loader = build_loader(x[i_train], y[i_train], batch_size=batch_size, shuffle=True)
    val_loader = build_loader(x[i_val], y[i_val], batch_size=batch_size)
    classifier = Classifier(model, learning_rate)
    print(pl.utilities.model_summary.ModelSummary(classifier, max_depth=-1))

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        trainer = pl.Trainer(
            default_root_dir=LOG_DIRECTORY,
            logger=pl.loggers.TensorBoardLogger(LOG_DIRECTORY, name),
            enable_model_summary=False,
            max_epochs=n_epochs,
        )
        trainer.fit(classifier, train_loader, val_loader)

    return classifier


def evaluate_model(model, x, y, i_train, i_val, i_test, batch_size=100):
    loader = build_loader(x, y, batch_size=batch_size)

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        trainer = pl.Trainer(
            default_root_dir=LOG_DIRECTORY,
            logger=False,
            enable_progress_bar=False,
            enable_model_summary=False,
        )
        z = trainer.predict(model, loader)
    z = np.vstack([u.numpy() for u in z])
```

```python
    references = np.argmax(y, axis=1)
    predictions = np.argmax(z, axis=1)
    matrices = {}
    for subset, indices in (('train', i_train), ('val', i_val), ('test', i_test)):
        matrices[subset] = sklearn.metrics.confusion_matrix(
            references[indices],
            predictions[indices],
        )

    return matrices
```

We start TensorBoard to visualize performance metrics during training.

If you prefer to view TensorBoard in a separate window, you can open http://localhost:6006/ in your web browser.

[7]:
```python
%reload_ext tensorboard
%tensorboard --logdir ../logs/ecg_rhythm_classification --port 6006
```

```
<IPython.core.display.HTML object>
```

We define a convolutional neural network.

[8]:
```python
class CnnModel(torch.nn.Module):

    def __init__(self, input_shape, output_shape, kernel_size=5):
        super().__init__()
        self.input_shape = input_shape
        self.output_shape = output_shape
        self.layers = torch.nn.Sequential(
            torch.nn.Conv1d(self.input_shape[0], 8, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(8),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(8, 16, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(16, 32, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(32),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(32, 64, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(64),
```

```
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),

            torch.nn.Flatten(),
            torch.nn.Linear(64, self.output_shape[0]),
        )

    def forward(self, x):
        return self.layers(x)
```

Then, we train and evaluate this model.

[9]:
```
input_shape = signals.shape[1:]
output_shape = encoded_rhythms.shape[1:]
n_epochs = 50
batch_size = 100

cnn = train_model(
    name='cnn',
    model=CnnModel(input_shape, output_shape),
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    learning_rate=0.0001,
    batch_size=batch_size,
    n_epochs=n_epochs,
)

cnn_matrices = evaluate_model(
    model=cnn,
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    i_test=i_test,
    batch_size=batch_size,
)
```

```
   | Name            | Type              | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model           | CnnModel          | 14.2 K | train | [1, 1, 1280] | [1,
5]
1  | model.layers    | Sequential        | 14.2 K | train | [1, 1, 1280] | [1,
5]
2  | model.layers.0  | Conv1d            | 48     | train | [1, 1, 1280] | [1,
```

```
8, 1280]
3  | model.layers.1  | BatchNorm1d       | 16     | train | [1, 8, 1280] | [1,
8, 1280]
4  | model.layers.2  | ReLU              | 0      | train | [1, 8, 1280] | [1,
8, 1280]
5  | model.layers.3  | MaxPool1d         | 0      | train | [1, 8, 1280] | [1,
8, 640]
6  | model.layers.4  | Conv1d            | 656    | train | [1, 8, 640]  | [1,
16, 640]
7  | model.layers.5  | BatchNorm1d       | 32     | train | [1, 16, 640] | [1,
16, 640]
8  | model.layers.6  | ReLU              | 0      | train | [1, 16, 640] | [1,
16, 640]
9  | model.layers.7  | MaxPool1d         | 0      | train | [1, 16, 640] | [1,
16, 320]
10 | model.layers.8  | Conv1d            | 2.6 K  | train | [1, 16, 320] | [1,
32, 320]
11 | model.layers.9  | BatchNorm1d       | 64     | train | [1, 32, 320] | [1,
32, 320]
12 | model.layers.10 | ReLU              | 0      | train | [1, 32, 320] | [1,
32, 320]
13 | model.layers.11 | MaxPool1d         | 0      | train | [1, 32, 320] | [1,
32, 160]
14 | model.layers.12 | Conv1d            | 10.3 K | train | [1, 32, 160] | [1,
64, 160]
15 | model.layers.13 | BatchNorm1d       | 128    | train | [1, 64, 160] | [1,
64, 160]
16 | model.layers.14 | ReLU              | 0      | train | [1, 64, 160] | [1,
64, 160]
17 | model.layers.15 | AdaptiveAvgPool1d | 0      | train | [1, 64, 160] | [1,
64, 1]
18 | model.layers.16 | Flatten           | 0      | train | [1, 64, 1]   | [1,
64]
19 | model.layers.17 | Linear            | 325    | train | [1, 64]      | [1,
5]
--------------------------------------------------------------------------------
-------------
14.2 K    Trainable params
0         Non-trainable params
14.2 K    Total params
0.057     Total estimated model params size (MB)
20        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

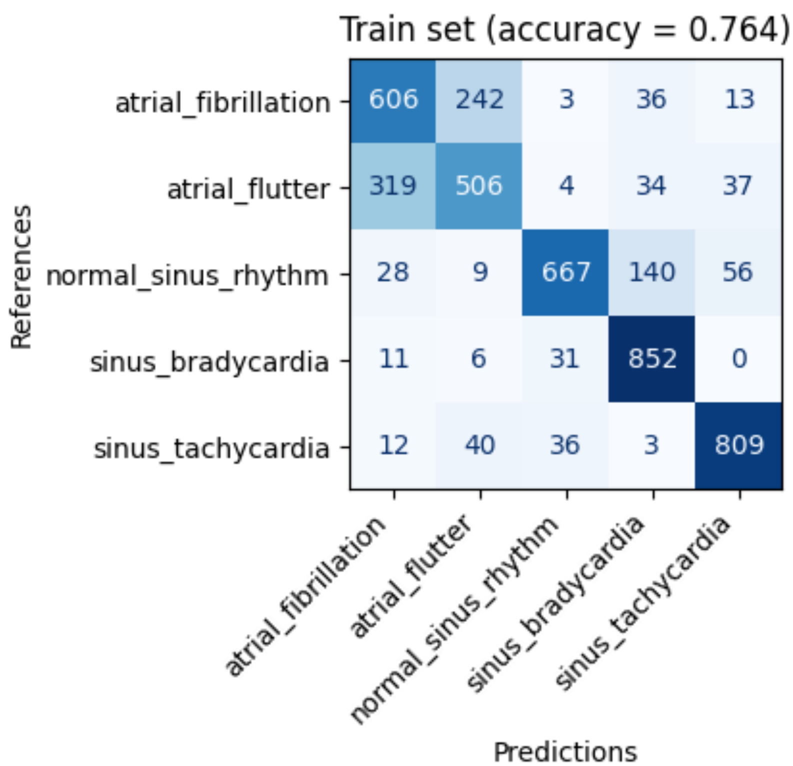
```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```

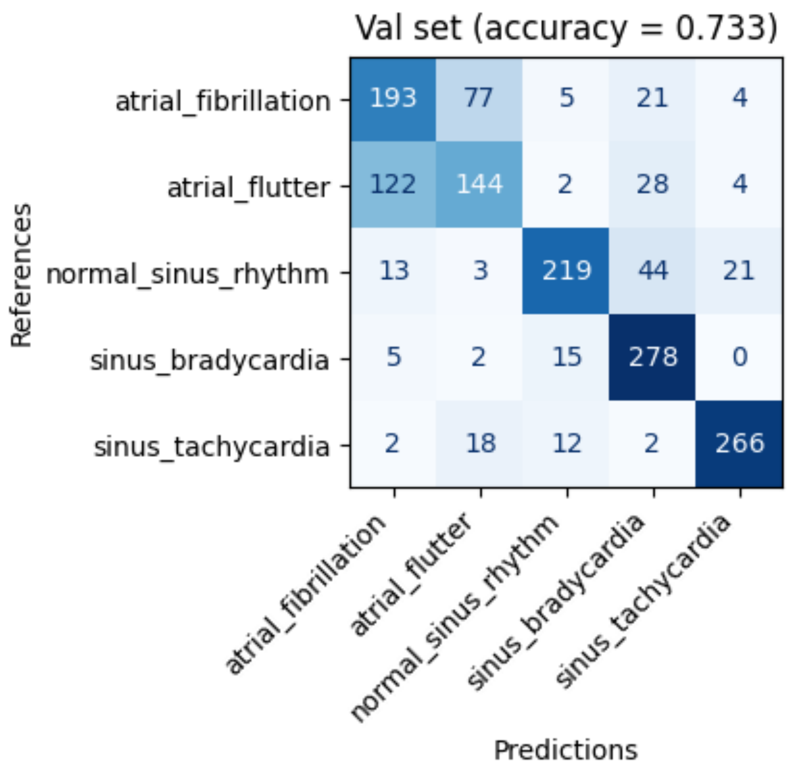
```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```

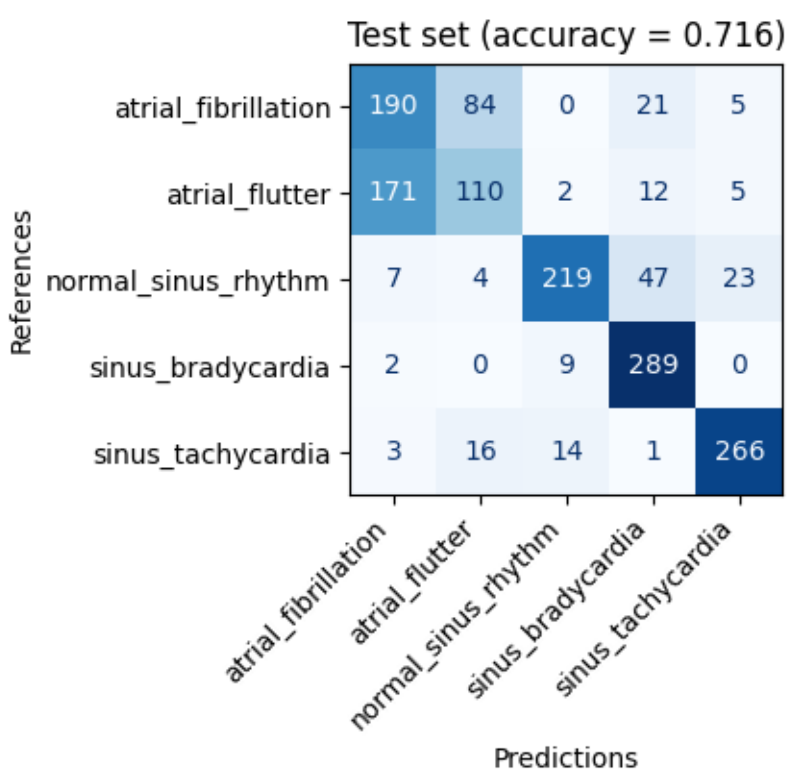
After the evaluation is finished, we can plot the confusion matrices for the training, validatoin, and test sets.

[10]:
```
def plot_confusion_matrix(c, labels=None, title=None):
    c = np.asarray(c)

    fig = plt.figure(figsize=(5, 4), constrained_layout=True)
    image = plt.imshow(c, cmap='Blues', interpolation='nearest')

    threshold = (c.min() + c.max()) / 2
    for i, j in itertools.product(range(c.shape[0]), repeat=2):
        if c[i, j] < threshold:
            color = image.cmap(image.cmap.N)
        else:
            color = image.cmap(0)
        text = format(c[i, j], '.2g')
        if c.dtype.kind != 'f':
            integer_text = format(c[i, j], 'd')
            if len(integer_text) < len(text):
                text = integer_text
        plt.text(j, i, text, color=color, ha='center', va='center')
```

```
    if labels is not None:
        plt.xticks(np.arange(c.shape[-1]), labels, rotation=45, ha='right')
        plt.yticks(np.arange(c.shape[-1]), labels)
    plt.xlabel('Predictions')
    plt.ylabel('References')
    if title is not None:
        plt.title(title)



def plot_confusion_matrices(matrices, labels):
    for subset in ('train', 'val', 'test'):
        c = matrices[subset]
        accuracy = np.trace(c) / c.sum()
        title = f'{subset.capitalize()} set (accuracy = {accuracy:.3f})'
        plot_confusion_matrix(c, labels=labels, title=title)


plot_confusion_matrices(cnn_matrices, np.unique(rhythms))
```




Val set (accuracy = 0.733)
References
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
193 77 5 21 4
122 144 2 28 4
13 3 219 44 21
5 2 15 278 0
2 18 12 2 266
Predictions


Test set (accuracy = 0.716)
References
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
190 84 0 21 5
171 110 2 12 5
7 4 219 47 23
2 0 9 289 0
3 16 14 1 266
Predictions

Question 2

Comment the metrics shown in TensorBoard and the confusion matrices. Does the model overfit? Are there some rhythms that are difficult to classify?

Answer The model does not overfit (0.71 for test set against 0.76 for validation). Furthermore, by inspecting the confusion matrices, we can observe that there is a cluster with atrial fibrillation and atrial flutter. In fact, these two rhythms are difficult to classify and are often confused with one another in the model.

Question 3

Define two custom models to classify cardiac rhythms from ECG signals.

You can directly define the layers of the custom models in the following classes.

[ ]:
```
class CustomModel1(torch.nn.Module):

    def __init__(self, input_shape, output_shape):
        super().__init__()
        self.input_shape = input_shape
        self.output_shape = output_shape

        kernel_size = 5
        # Implement you own model here.
        # define two twins CNNs
        self.cnn1 = torch.nn.Sequential(
            torch.nn.Conv1d(self.input_shape[0], 8, kernel_size, 
 ↪padding='same'),
            torch.nn.BatchNorm1d(8),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(8, 16, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(16, 32, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(32),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(32, 64, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(64),
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),
        )
```

```
        self.cnn2 = torch.nn.Sequential(
            torch.nn.Conv1d(self.input_shape[0], 8, kernel_size,␣
↪padding='same'),
            torch.nn.BatchNorm1d(8),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(8, 16, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(16, 32, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(32),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(32, 64, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(64),
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),
        )

        self.classif = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(2*64, self.output_shape[0]),
        )

    def forward(self, x):
        tmp = torch.concat([self.cnn1(x), self.cnn2(x)], dim=1)
        return self.classif(tmp)


class CustomModel2(torch.nn.Module):

    def __init__(self, input_shape, output_shape):
        super().__init__()
        self.input_shape = input_shape
        self.output_shape = output_shape

        # Implement you own model here.
        kernel_size = 3
        self.layers = torch.nn.Sequential(
            torch.nn.Conv1d(self.input_shape[0], 8, kernel_size,␣
↪padding='same'),
            torch.nn.BatchNorm1d(8),
            torch.nn.ReLU(),
```

```python
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(8, 16, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(16, 32, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(32),
            torch.nn.ReLU(),
            torch.nn.MaxPool1d(2),

            torch.nn.Conv1d(32, 64, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(64),
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),

            torch.nn.Conv1d(64, 32, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(32),
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),

            torch.nn.Conv1d(32, 16, kernel_size, padding='same'),
            torch.nn.BatchNorm1d(16),
            torch.nn.ReLU(),
            torch.nn.AdaptiveAvgPool1d(1),

            torch.nn.Flatten(),
            torch.nn.Linear(16, self.output_shape[0]),
        )

    def forward(self, x):
        return self.layers(x)
```

You can train and evaluate the first custom model.

[15]:
```python
custom1 = train_model(
    name='custom1',
    model=CustomModel1(input_shape, output_shape),
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    learning_rate=0.0001,
    batch_size=batch_size,
    n_epochs=n_epochs,
)
```

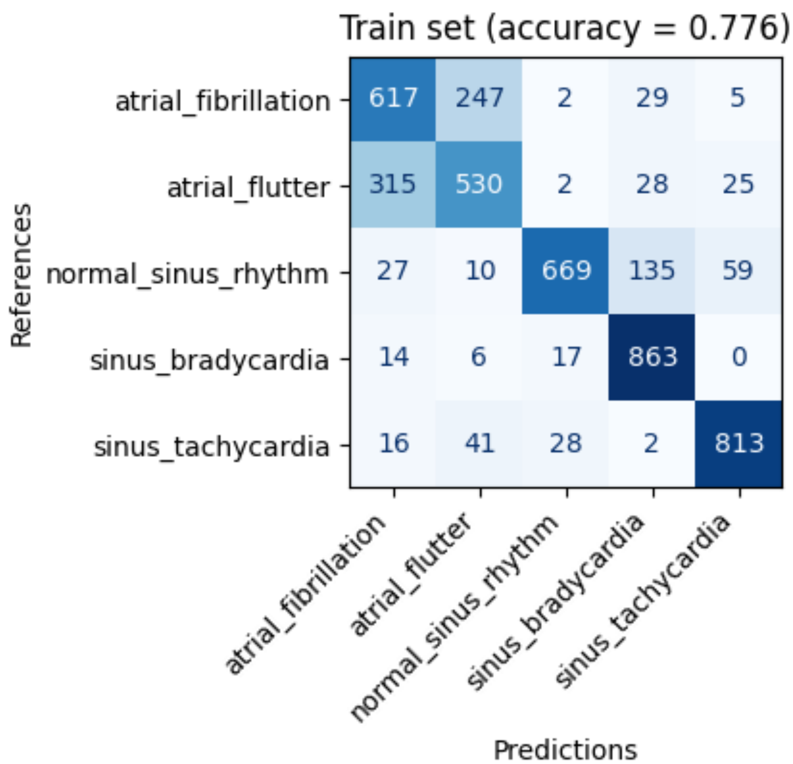
```
custom1_matrices = evaluate_model(
    model=custom1,
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    i_test=i_test,
    batch_size=batch_size,
)

plot_confusion_matrices(custom1_matrices, np.unique(rhythms))
```

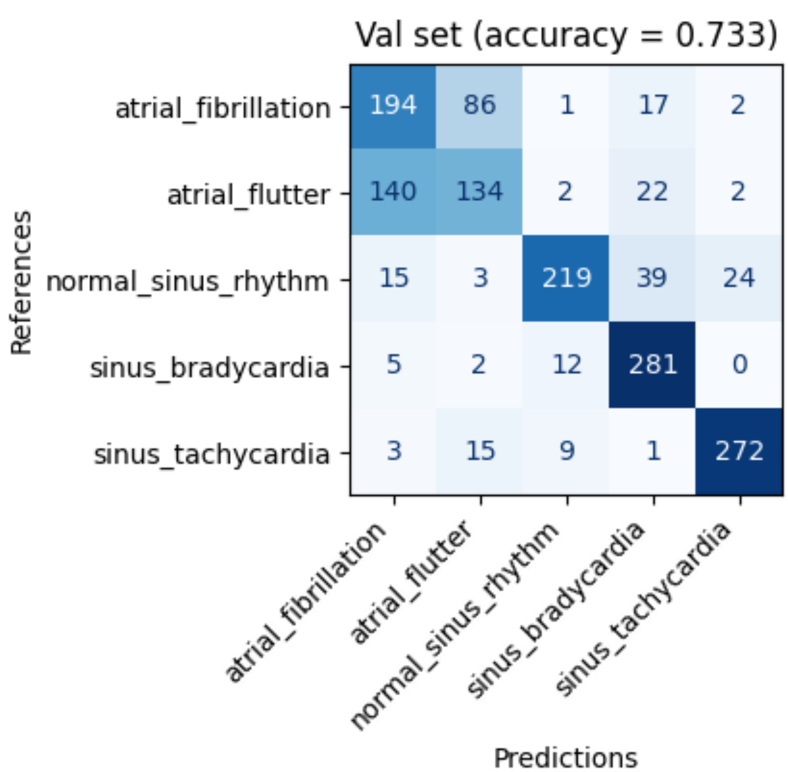
```
   | Name              | Type              | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model             | CustomModel1      | 28.3 K | train | [1, 1, 1280] | [1,
5]
1  | model.cnn1        | Sequential        | 13.8 K | train | [1, 1, 1280] | [1,
64, 1]
2  | model.cnn1.0      | Conv1d            | 48     | train | [1, 1, 1280] | [1,
8, 1280]
3  | model.cnn1.1      | BatchNorm1d       | 16     | train | [1, 8, 1280] | [1,
8, 1280]
4  | model.cnn1.2      | ReLU              | 0      | train | [1, 8, 1280] | [1,
8, 1280]
5  | model.cnn1.3      | MaxPool1d         | 0      | train | [1, 8, 1280] | [1,
8, 640]
6  | model.cnn1.4      | Conv1d            | 656    | train | [1, 8, 640]  | [1,
16, 640]
7  | model.cnn1.5      | BatchNorm1d       | 32     | train | [1, 16, 640] | [1,
16, 640]
8  | model.cnn1.6      | ReLU              | 0      | train | [1, 16, 640] | [1,
16, 640]
9  | model.cnn1.7      | MaxPool1d         | 0      | train | [1, 16, 640] | [1,
16, 320]
10 | model.cnn1.8      | Conv1d            | 2.6 K  | train | [1, 16, 320] | [1,
32, 320]
11 | model.cnn1.9      | BatchNorm1d       | 64     | train | [1, 32, 320] | [1,
32, 320]
12 | model.cnn1.10     | ReLU              | 0      | train | [1, 32, 320] | [1,
32, 320]
13 | model.cnn1.11     | MaxPool1d         | 0      | train | [1, 32, 320] | [1,
32, 160]
14 | model.cnn1.12     | Conv1d            | 10.3 K | train | [1, 32, 160] | [1,
64, 160]
15 | model.cnn1.13     | BatchNorm1d       | 128    | train | [1, 64, 160] | [1,
```

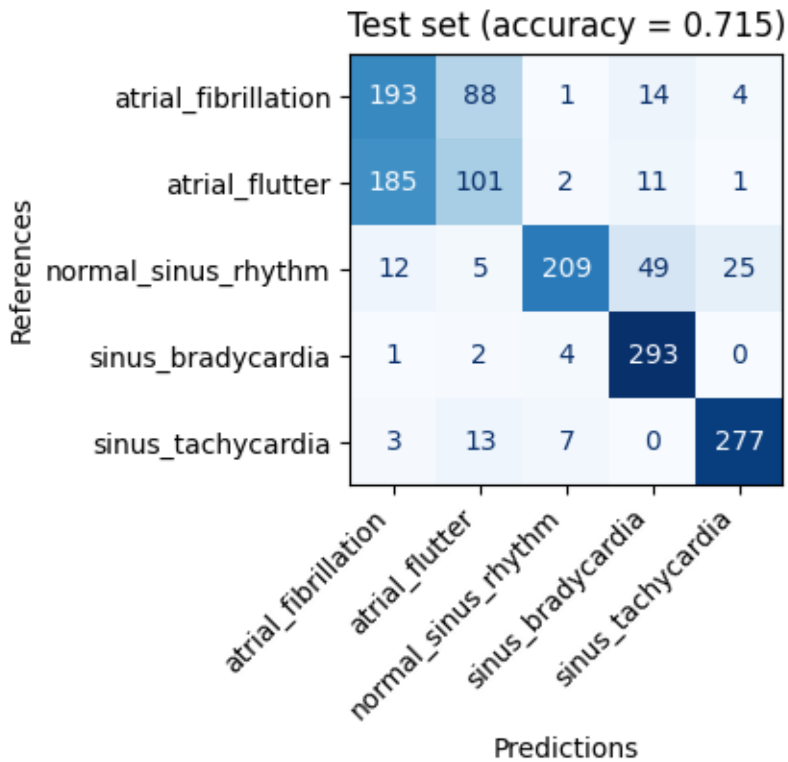
```
64, 160]
16 | model.cnn1.14   | ReLU              | 0      | train | [1, 64, 160] | [1,
64, 160]
17 | model.cnn1.15   | AdaptiveAvgPool1d | 0      | train | [1, 64, 160] | [1,
64, 1]
18 | model.cnn2      | Sequential        | 13.8 K | train | [1, 1, 1280] | [1,
64, 1]
19 | model.cnn2.0    | Conv1d            | 48     | train | [1, 1, 1280] | [1,
8, 1280]
20 | model.cnn2.1    | BatchNorm1d       | 16     | train | [1, 8, 1280] | [1,
8, 1280]
21 | model.cnn2.2    | ReLU              | 0      | train | [1, 8, 1280] | [1,
8, 1280]
22 | model.cnn2.3    | MaxPool1d         | 0      | train | [1, 8, 1280] | [1,
8, 640]
23 | model.cnn2.4    | Conv1d            | 656    | train | [1, 8, 640]  | [1,
16, 640]
24 | model.cnn2.5    | BatchNorm1d       | 32     | train | [1, 16, 640] | [1,
16, 640]
25 | model.cnn2.6    | ReLU              | 0      | train | [1, 16, 640] | [1,
16, 640]
26 | model.cnn2.7    | MaxPool1d         | 0      | train | [1, 16, 640] | [1,
16, 320]
27 | model.cnn2.8    | Conv1d            | 2.6 K  | train | [1, 16, 320] | [1,
32, 320]
28 | model.cnn2.9    | BatchNorm1d       | 64     | train | [1, 32, 320] | [1,
32, 320]
29 | model.cnn2.10   | ReLU              | 0      | train | [1, 32, 320] | [1,
32, 320]
30 | model.cnn2.11   | MaxPool1d         | 0      | train | [1, 32, 320] | [1,
32, 160]
31 | model.cnn2.12   | Conv1d            | 10.3 K | train | [1, 32, 160] | [1,
64, 160]
32 | model.cnn2.13   | BatchNorm1d       | 128    | train | [1, 64, 160] | [1,
64, 160]
33 | model.cnn2.14   | ReLU              | 0      | train | [1, 64, 160] | [1,
64, 160]
34 | model.cnn2.15   | AdaptiveAvgPool1d | 0      | train | [1, 64, 160] | [1,
64, 1]
35 | model.classif   | Sequential        | 645    | train | [1, 128, 1]  | [1,
5]
36 | model.classif.0 | Flatten           | 0      | train | [1, 128, 1]  | [1,
128]
37 | model.classif.1 | Linear            | 645    | train | [1, 128]     | [1,
5]
--------------------------------------------------------------------------------
-------------
28.3 K    Trainable params
```

```
0         Non-trainable params
28.3 K    Total params
0.113     Total estimated model params size (MB)
38        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```


Train set (accuracy = 0.776)
References
Predictions
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
617
247
2
29
5
315
530
2
28
25
27
10
669
135
59
14
6
17
863
0
16
41
28
2
813


Val set (accuracy = 0.733)
References
Predictions
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
194
86
1
17
2
140
134
2
22
2
15
3
219
39
24
5
2
12
281
0
3
15
9
1
272



And then the second custom model.

[16]:
```
custom2 = train_model(
    name='custom2',
    model=CustomModel2(input_shape, output_shape),
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    learning_rate=0.0001,
    batch_size=batch_size,
    n_epochs=n_epochs,
)

custom2_matrices = evaluate_model(
    model=custom2,
    x=signals,
    y=encoded_rhythms,
    i_train=i_train,
    i_val=i_val,
    i_test=i_test,
    batch_size=batch_size,
```

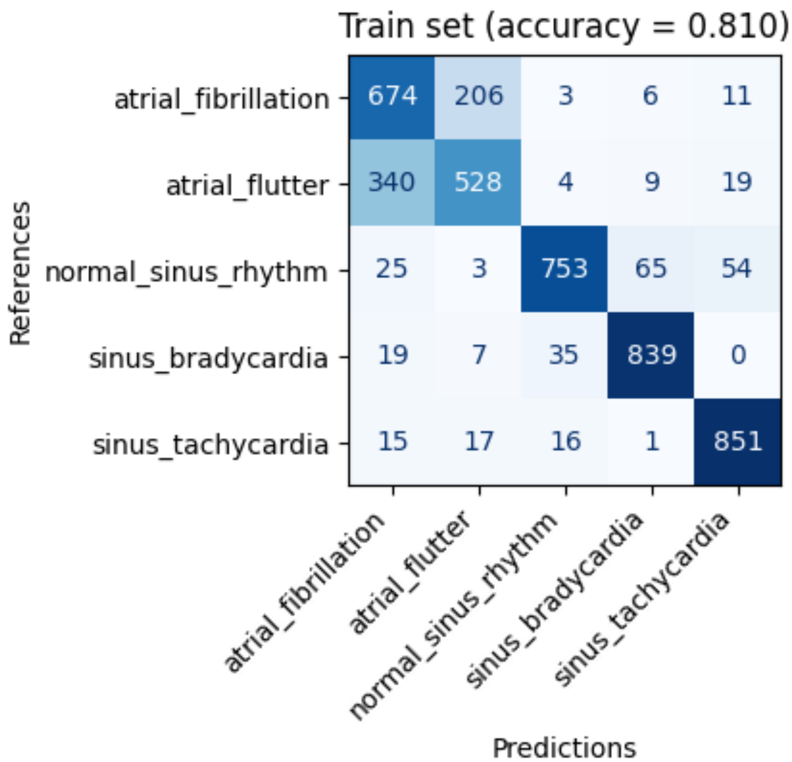
```
)

plot_confusion_matrices(custom2_matrices, np.unique(rhythms))
```

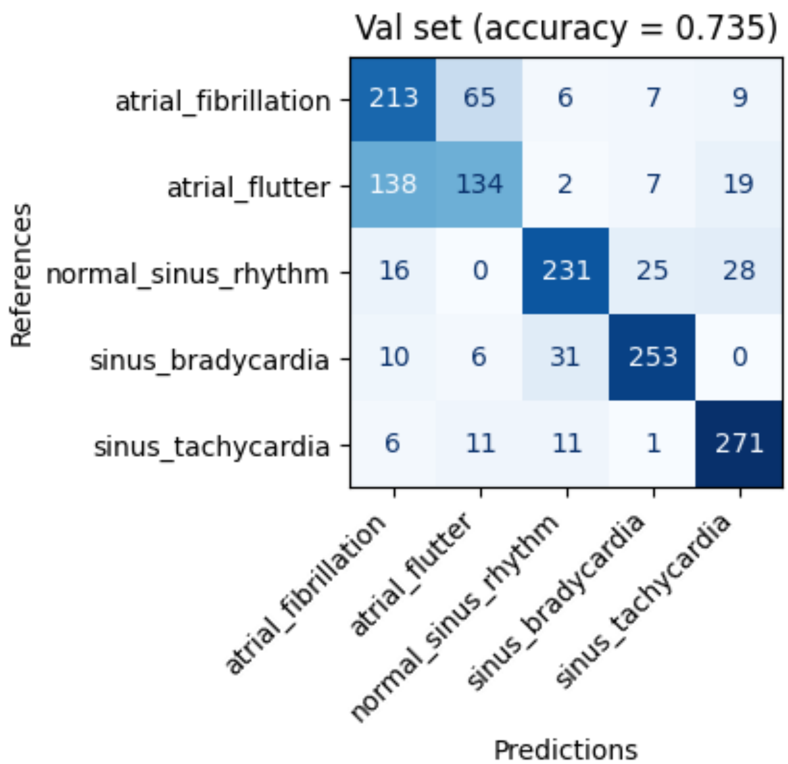
```
   | Name             | Type              | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model            | CustomModel2      | 16.4 K | train | [1, 1, 1280] | [1,
5]
1  | model.layers     | Sequential        | 16.4 K | train | [1, 1, 1280] | [1,
5]
2  | model.layers.0   | Conv1d            | 32     | train | [1, 1, 1280] | [1,
8, 1280]
3  | model.layers.1   | BatchNorm1d       | 16     | train | [1, 8, 1280] | [1,
8, 1280]
4  | model.layers.2   | ReLU              | 0      | train | [1, 8, 1280] | [1,
8, 1280]
5  | model.layers.3   | MaxPool1d         | 0      | train | [1, 8, 1280] | [1,
8, 640]
6  | model.layers.4   | Conv1d            | 400    | train | [1, 8, 640]  | [1,
16, 640]
7  | model.layers.5   | BatchNorm1d       | 32     | train | [1, 16, 640] | [1,
16, 640]
8  | model.layers.6   | ReLU              | 0      | train | [1, 16, 640] | [1,
16, 640]
9  | model.layers.7   | MaxPool1d         | 0      | train | [1, 16, 640] | [1,
16, 320]
10 | model.layers.8   | Conv1d            | 1.6 K  | train | [1, 16, 320] | [1,
32, 320]
11 | model.layers.9   | BatchNorm1d       | 64     | train | [1, 32, 320] | [1,
32, 320]
12 | model.layers.10  | ReLU              | 0      | train | [1, 32, 320] | [1,
32, 320]
13 | model.layers.11  | MaxPool1d         | 0      | train | [1, 32, 320] | [1,
32, 160]
14 | model.layers.12  | Conv1d            | 6.2 K  | train | [1, 32, 160] | [1,
64, 160]
15 | model.layers.13  | BatchNorm1d       | 128    | train | [1, 64, 160] | [1,
64, 160]
16 | model.layers.14  | ReLU              | 0      | train | [1, 64, 160] | [1,
64, 160]
17 | model.layers.15  | AdaptiveAvgPool1d | 0      | train | [1, 64, 160] | [1,
64, 1]
18 | model.layers.16  | Conv1d            | 6.2 K  | train | [1, 64, 1]   | [1,
32, 1]
19 | model.layers.17  | BatchNorm1d       | 64     | train | [1, 32, 1]   | [1,
32, 1]
```

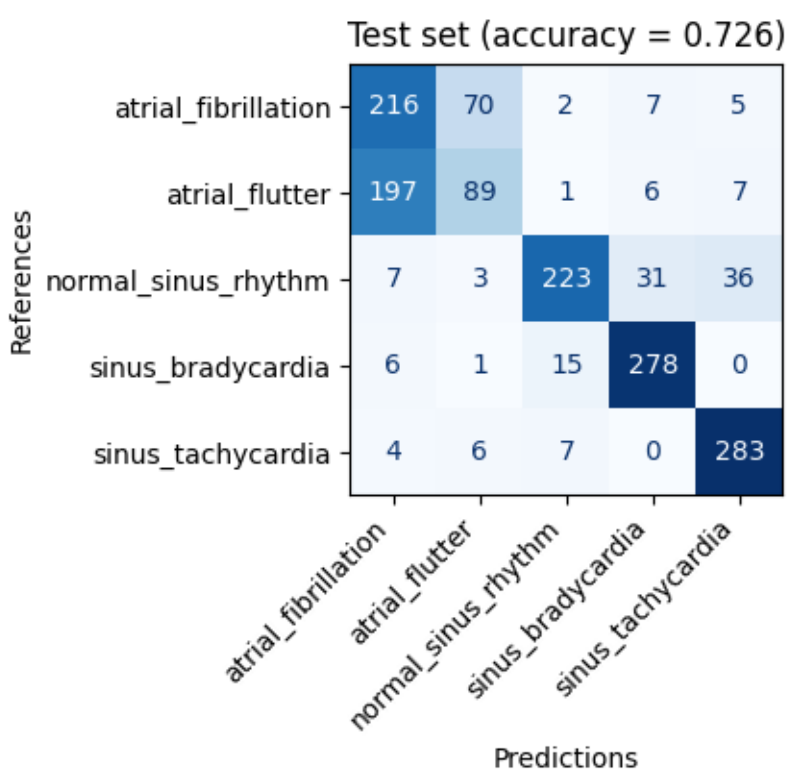
```
20 | model.layers.18 | ReLU              | 0      | train | [1, 32, 1]   | [1,
32, 1]
21 | model.layers.19 | AdaptiveAvgPool1d | 0      | train | [1, 32, 1]   | [1,
32, 1]
22 | model.layers.20 | Conv1d            | 1.6 K  | train | [1, 32, 1]   | [1,
16, 1]
23 | model.layers.21 | BatchNorm1d       | 32     | train | [1, 16, 1]   | [1,
16, 1]
24 | model.layers.22 | ReLU              | 0      | train | [1, 16, 1]   | [1,
16, 1]
25 | model.layers.23 | AdaptiveAvgPool1d | 0      | train | [1, 16, 1]   | [1,
16, 1]
26 | model.layers.24 | Flatten           | 0      | train | [1, 16, 1]   | [1,
16]
27 | model.layers.25 | Linear            | 85     | train | [1, 16]      | [1,
5]
--------------------------------------------------------------------------------
-------------
16.4 K    Trainable params
0         Non-trainable params
16.4 K    Total params
0.065     Total estimated model params size (MB)
28        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```




Val set (accuracy = 0.735)
References
Predictions
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
213 65 6 7 9
138 134 2 7 19
16 0 231 25 28
10 6 31 253 0
6 11 11 1 271


Test set (accuracy = 0.726)
References
Predictions
atrial_fibrillation
atrial_flutter
normal_sinus_rhythm
sinus_bradycardia
sinus_tachycardia
216 70 2 7 5
197 89 1 6 7
7 3 223 31 36
6 1 15 278 0
4 6 7 0 283

Question 4

How do the two custom models perform? Do they overfit? Do they outpeform the first model?

Answer Both of our custom models slightly outperform the first model in terms of accuracy, but seem to struggle with `atrial_flutter` and `atrial_fibrillation` signals just like the proposed model.

hr_estimation

December 12, 2024

1 Heart Rate Estimation

The goal of this exercise is to estimate heart rate from PPG and acceleration signals. We signals from the PPG-DaLiA dataset (https://archive.ics.uci.edu/ml/datasets/PPG-DaLiA). It includes PPG and acceleration signals as well as reference heart rate computed from an ECG signal.

These signals were collected during various activities but we focus on two of them: sitting and walking.

First, we import all the packages we will need, define some global variables, and seed the random number generators.

[1]:
```
%matplotlib widget


import copy
import functools
import itertools
import logging
import operator
import pathlib
import warnings

import IPython.display
import joblib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pytorch_lightning as pl
import torch


DATA_FILE = pathlib.Path('../data/ppg_dalia.pkl')
LOG_DIRECTORY = pathlib.Path('../logs/hr_estimation')


# Disable logging for PyTorch Lightning to avoid too long outputs.
logging.getLogger('pytorch_lightning').setLevel(logging.ERROR)
```

```
# Seed random number generators for reproducible results.
pl.seed_everything(42)
```

```
Seed set to 42
```

[1]: 42

Then, we load the PPG and acceleration signals as well as the reference heart rate. The signals are already pre-processed with the following steps:

- Band-pass filtering between 0.4 and 4.0 Hz (24 - 240 bpm).
- Resampling to 25 Hz.

We also define the window length and shift length used to compute the reference heart rate.

[2]:
```
FS = 25.0  # Sampling frequency of the PPG and acceleration signals in Hertz.
WINDOW_LENGTH = 8.0  # Window duration in seconds used to compute the reference␣
 ↪heart rate.
SHIFT_LENGTH = 2.0  # Shift between successive windows in seconds.

WINDOW_SIZE = round(FS * WINDOW_LENGTH)
SHIFT_SIZE = round(FS * SHIFT_LENGTH)

records = joblib.load(DATA_FILE)
subjects = set(record['subject'] for record in records)

print(f'Window length: {WINDOW_LENGTH} s (n = {WINDOW_SIZE})')
print(f'Shift length: {SHIFT_LENGTH} s (n = {SHIFT_SIZE})')
print(f'Number of records: {len(records)}')
print(f'Number of subjects: {len(subjects)}')
```

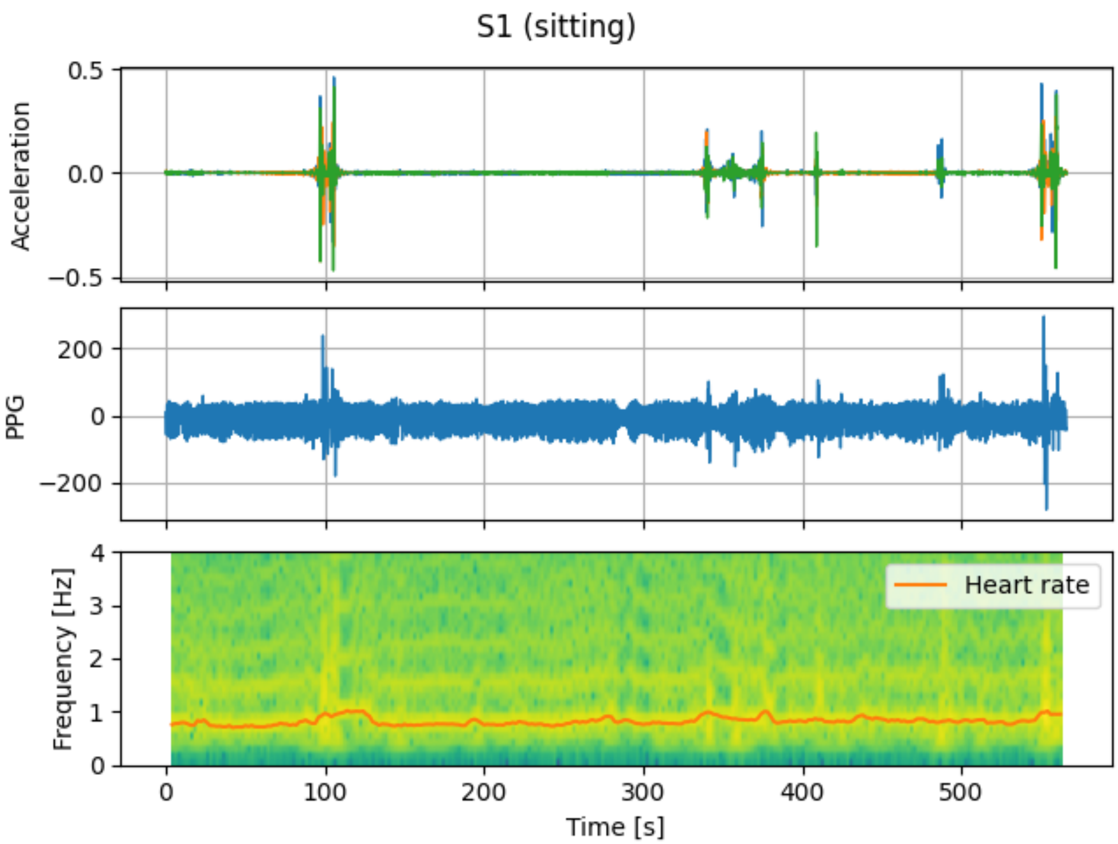
```
Window length: 8.0 s (n = 200)
Shift length: 2.0 s (n = 50)
Number of records: 29
Number of subjects: 15
```

Here are two examples of PPG and acceleration signals. One recorded when the subject is sitting and one recorded when the subject is walking.

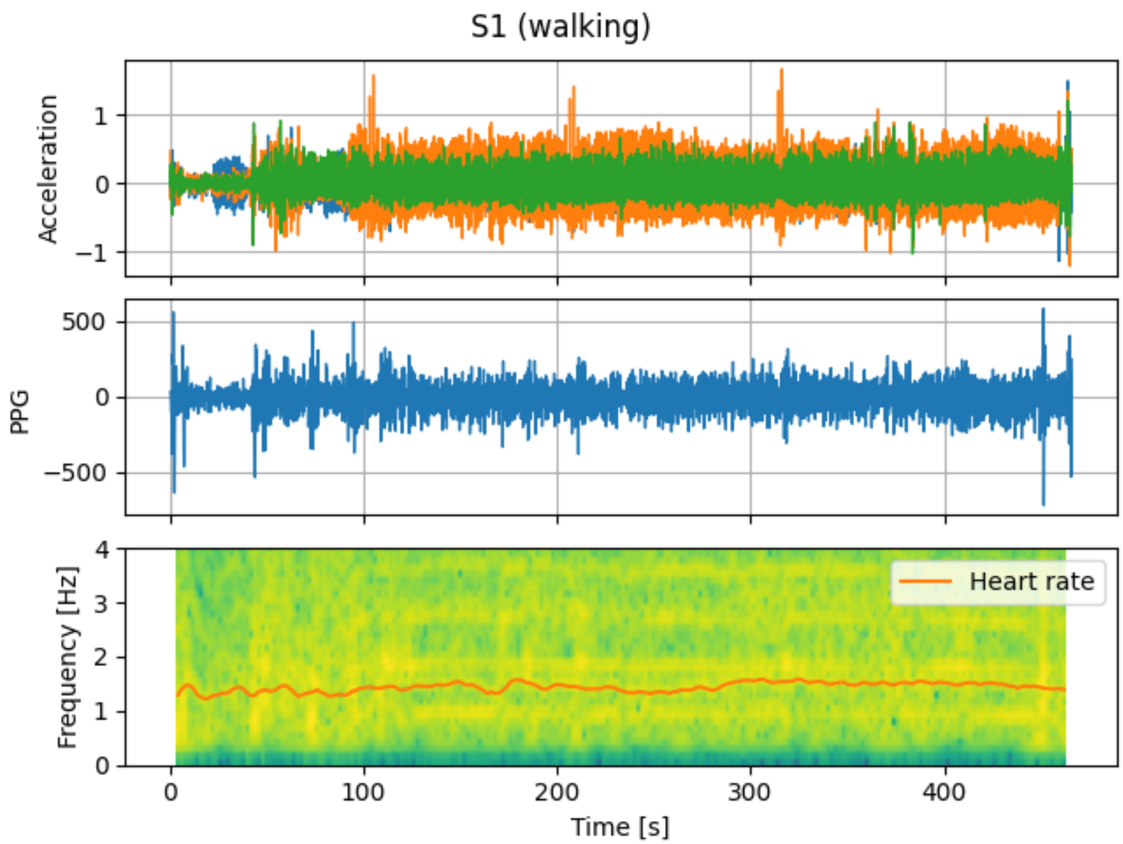
Each figure includes three plots: the tri-axis acceleration signals, the PPG signal, and a spectrogram of the PPG signal with the reference heart rate on top.

[3]:
```
def plot_signals(record):
    signals = record['signals']
    hr = record['hr']

    fig, axes = plt.subplots(3, 1, sharex='all', constrained_layout=True)
    plt.suptitle(f'{record["subject"]} ({record["activity"]})')

    plt.sca(axes.flat[0])
    plt.plot(signals['time'].to_numpy(),
```

```
            signals[['acc_x', 'acc_y', 'acc_z']].to_numpy(),
            linewidth=1)
    plt.grid()
    plt.ylabel('Acceleration')

    plt.sca(axes.flat[1])
    plt.plot(signals['time'].to_numpy(), signals['ppg'].to_numpy(),
            linewidth=1)
    plt.grid()
    plt.ylabel('PPG')

    plt.sca(axes.flat[2])
    plt.specgram(signals['ppg'].to_numpy(), Fs=FS, NFFT=WINDOW_SIZE,
                noverlap=WINDOW_SIZE - SHIFT_SIZE)
    plt.plot(hr['time'].to_numpy(), hr['hr'].to_numpy() / 60.0,
            color='tab:orange', label='Heart rate')
    plt.ylim(0.0, 4.0)
    plt.xlabel('Time [s]')
    plt.ylabel('Frequency [Hz]')
    plt.legend(loc='upper right')


plot_signals(records[0])
plot_signals(records[1])
```



S1 (sitting)
0.5
0.0
−0.5
Acceleration
200
0
−200
PPG
4
3
2
1
0
Frequency [Hz]
Heart rate
0
100
200
300
400
500
Time [s]



By zooming on the PPG signal, it is clear that walking cause a degradation in signal quality.

We will try to estimate the heart rate on sliding windows of the PPG and acceleration signals. To make things easier, we use the same window length and shift between windows as the reference heart rate.

So the next step is to extract sliding windows from all the records. We also extract the corresponding subject identifier for splitting the windows into subsets for training, validation, and testing.

In addition, we also prepare windows that include only the PPG signal (first channel).

[4]:
```
def extract_windows(record):
    x = record['signals'][['ppg', 'acc_x', 'acc_y', 'acc_z']].to_numpy()
    n = x.shape[0]

    windows = []
    for start in range(0, n - WINDOW_SIZE + 1, SHIFT_SIZE):
        end = start + WINDOW_SIZE
        windows.append(x[start:end].T)
    windows = np.stack(windows)
    targets = record['hr']['hr'].to_numpy()
```

```
    return windows, targets


def extract_all_windows(records):
    windows = []
    targets = []
    subjects = []
    activities = []
    for record in records:
        x, y = extract_windows(record)
        windows.append(x)
        targets.append(y)
        subjects.extend(itertools.repeat(record['subject'], x.shape[0]))
        activities.extend(itertools.repeat(record['activity'], x.shape[0]))

    windows = np.concatenate(windows, axis=0)
    targets = np.concatenate(targets)[:, None]
    subjects = np.array(subjects)
    activities = np.array(activities)

    return windows, targets, subjects, activities


ppg_acc_windows, targets, subjects, activities = extract_all_windows(records)
ppg_windows = ppg_acc_windows[:, :1, :]

print(f'Shape of PPG and accleration windows: {ppg_acc_windows.shape}')
print(f'Shape of PPG windows: {ppg_windows.shape}')
```

```
Shape of PPG and accleration windows: (7420, 4, 200)
Shape of PPG windows: (7420, 1, 200)
```

We have 7420 windows with 1 or 4 channels and that each window includes 200 samples (8 seconds at 25 Hz).

Next, we split the windows for training, validation, and testing by subjects. The test set includes 9 subjects, the validation set 3 subjects, and the test set 3 subjects.

[5]:
```
def split_subjects(subjects):
    val_subjects = ('S10', 'S11', 'S12')
    test_subjects = ('S13', 'S14', 'S15')

    i_val = np.flatnonzero(np.isin(subjects, val_subjects))
    i_test = np.flatnonzero(np.isin(subjects, test_subjects))
    i_train = np.setdiff1d(np.arange(subjects.size), np.union1d(i_val, i_test))

    assert not (set(subjects[i_train]) & set(subjects[i_val]))
    assert not (set(subjects[i_train]) & set(subjects[i_test]))
```

```
    assert not (set(subjects[i_val]) & set(subjects[i_test]))
    assert (set(subjects[i_train]) | set(subjects[i_val]) |␣
 ↪set(subjects[i_test])) == set(subjects)

    return i_train, i_val, i_test


i_train, i_val, i_test = split_subjects(subjects)

print(f'Subject used for training   : {pd.unique(subjects[i_train])}')
print(f'Subject used for validation : {pd.unique(subjects[i_val])}')
print(f'Subject used for testing    : {pd.unique(subjects[i_test])}')
```

```
Subject used for training   : ['S1' 'S2' 'S3' 'S4' 'S5' 'S6' 'S7' 'S8' 'S9']
Subject used for validation : ['S10' 'S11' 'S12']
Subject used for testing    : ['S13' 'S14' 'S15']
```

To make training more stable, we scale the windows such that they have approximately unit variance.

[6]:
```
def compute_scaling(windows):
    sigma = np.mean(np.std(windows, axis=-1, keepdims=True), axis=0)
    return 1.0 / sigma


ppg_acc_alpha = compute_scaling(ppg_acc_windows[i_train])
ppg_acc_windows *= ppg_acc_alpha
ppg_alpha = compute_scaling(ppg_windows[i_train])
ppg_windows *= ppg_alpha
```

Question 1

The windows are scaled but not centered. Why?

Answer

We can see in the cell below that the mean of the signal is already zero. Therefore there is no need to center the signal. When filtering (Band-Pass) the signal, the offset (zero frequency) is removed and the signal is centered.

[24]:
```
ppg_windows.mean(axis=(0, 2))
```

[24]:
```
array([-6.8200391e-05])
```

Now we define the convolutional neural network (CNN) we will use to estimate heart rate. It is composed of convolutional layers to extract features and dense layers to estimate heart rate. The convolutional layers can optionally include batch normalization and the dense layers dropout.

[7]:
```
class CnnModel(torch.nn.Module):

    def __init__(self,
```

```python
                 input_shape,
                 output_shape,
                 n_convolutional_layers=1,
                 kernel_size=5,
                 n_initial_channels=16,
                 use_normalization=False,
                 n_dense_layers=1,
                 n_units=128,
                 dropout=0.0):
        super().__init__()
        self.input_shape = input_shape
        self.output_shape = output_shape
        self.n_convolutional_layers = n_convolutional_layers
        self.kernel_size = kernel_size
        self.n_initial_channels = n_initial_channels
        self.use_normalization = use_normalization
        self.n_dense_layers = n_dense_layers
        self.n_units = n_units
        self.dropout = dropout
        self.layers = self._build_layers()

    @property
    def input_size(self):
        return functools.reduce(operator.mul, self.input_shape)

    @property
    def output_size(self):
        return functools.reduce(operator.mul, self.output_shape)

    def _build_layers(self):
        layers = self._build_convolutional_layers()
        layers.extend(self._build_dense_layers())
        return torch.nn.Sequential(*layers)

    def _build_convolutional_layers(self):
        layers = []

        n_output_channels = self.input_shape[0]
        for i in range(self.n_convolutional_layers):
            n_input_channels = n_output_channels
            n_output_channels = self.n_initial_channels * 2 ** i
            layers.append(torch.nn.Conv1d(
                in_channels=n_input_channels,
                out_channels=n_output_channels,
                kernel_size=self.kernel_size,
                padding='same',
            ))
```

```
            if self.use_normalization:
                layers.append(torch.nn.BatchNorm1d(n_output_channels))
            layers.append(torch.nn.ReLU())
            if i < self.n_convolutional_layers - 1:
                layers.append(torch.nn.MaxPool1d(kernel_size=2))
            else:
                layers.append(torch.nn.AdaptiveAvgPool1d(1))
        layers.append(torch.nn.Flatten())

        return layers

    def _build_dense_layers(self):
        sizes = [self.n_initial_channels
                 * 2 ** (self.n_convolutional_layers - 1)]
        sizes.extend(itertools.repeat(self.n_units, self.n_dense_layers - 1))
        sizes.append(self.output_size)

        layers = []
        for i in range(self.n_dense_layers - 1):
            layers.append(torch.nn.Linear(sizes[i], sizes[i + 1]))
            layers.append(torch.nn.ReLU())
            if 0.0 < self.dropout < 1.0:
                layers.append(torch.nn.Dropout(self.dropout))
        layers.append(torch.nn.Linear(sizes[-2], sizes[-1]))

        return layers

    def forward(self, x):
        return self.layers(x)
```

We also define a class to specify how the model should be trained and evaluated and a few utility functions. It is also here that we select the mean squared error (MSE) as the loss function to optimize the parameters. We also compute the mean absolute error (MAE) as an additional metric to monitor training.

[8]:
```
class Regressor(pl.LightningModule):

    def __init__(self, config):
        super().__init__()
        self.save_hyperparameters()
        self.config = config
        self.model = CnnModel(**self.config['model'])
        self.example_input_array = torch.zeros((1,) + self.model.input_shape)

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), **self.config['optimizer'])

```

```
    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        return self._run_step(batch, 'train')

    def validation_step(self, batch, batch_idx):
        self._run_step(batch, 'val')

    def test_step(self, batch, batch_idx):
        self._run_step(batch, 'test')

    def predict_step(self, batch, batch_idx, dataloader_idx=0):
        x, y = batch
        return self.model(x)

    def _run_step(self, batch, subset):
        x, y = batch
        z = self.model(x)
        mse = torch.nn.functional.mse_loss(z, y)
        mae = torch.nn.functional.l1_loss(z, y)
        self.log_dict({
            f'{subset}_mse': mse,
            f'{subset}_mae': mae,
        }, on_step=False, on_epoch=True, prog_bar=True)
        return mse


def build_loader(*tensors, batch_size=100, shuffle=False, n_workers=0):
    dataset = torch.utils.data.TensorDataset(*map(torch.Tensor, tensors))
    return torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        num_workers=n_workers,
    )


def train_model(config, windows, targets, i_train, i_val, n_epochs, name):
    train_loader = build_loader(windows[i_train], targets[i_train],␣
 ↪shuffle=True)
    val_loader = build_loader(windows[i_val], targets[i_val])
    regressor = Regressor(config)
    print(pl.utilities.model_summary.ModelSummary(regressor, max_depth=-1))

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
```

```
        trainer = pl.Trainer(
            default_root_dir=LOG_DIRECTORY,
            logger=pl.loggers.TensorBoardLogger(LOG_DIRECTORY, name),
            enable_model_summary=False,
            max_epochs=n_epochs,
        )
        trainer.fit(regressor, train_loader, val_loader)

    return regressor


def compute_metrics(targets, predictions):
    targets = targets.ravel()
    predictions = predictions.ravel()
    return {
        'count': targets.size,
        'mse': np.mean((targets - predictions) ** 2),
        'mae': np.mean(np.abs(targets - predictions)),
    }


def evaluate_model(model, windows, targets, i_train, i_val, i_test):
    loader = build_loader(windows, targets)

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        trainer = pl.Trainer(
            default_root_dir=LOG_DIRECTORY,
            logger=False,
            enable_progress_bar=False,
            enable_model_summary=False,
        )
        predictions = trainer.predict(model, loader)
    predictions = np.vstack([p.numpy() for p in predictions])

    metrics = []
    for subset, indices in (('train', i_train), ('val', i_val), ('test',␣
↪i_test)):
        metrics.append({
            'subset': subset,
            **compute_metrics(targets[indices], predictions[indices]),
        })
    return pd.DataFrame(metrics)
```

We also start TensorBoard to monitor training.

If you prefer to view TensorBoard in a separate window, you can open http://localhost:6006/ in your web browser.

[28]:
```
%reload_ext tensorboard
%tensorboard --logdir ../logs/hr_estimation --port 6007
```

```
<IPython.core.display.HTML object>
```

We are finally to define the first model configuration we will train. It includes the following layers:

- Input windows (input_size = 200, input channels = 1 or 4)
- Convolutional layer (kernel size = 5, output size = 200, output channels = 16)
- ReLU activation
- Max pooling (output size = 100, output channels = 16)
- Convolutional layer (kernel size = 5, output size = 100, output channels = 32)
- ReLU activation
- Max pooling (output size = 50, output channels = 32)
- Convolutional layer (kernel size = 5, output size = 50, output channels = 64)
- ReLU activation
- Max pooling (output size = 25, output channels = 64)
- Convolutional layer (kernel size = 5, output size = 25, output channels = 128)
- ReLU activation
- Global averaging pooling (output size = 128)
- Dense layer (output size = 128)
- ReLU activation
- Dense layer (output size = 128)
- ReLU activation
- Dense layer (output size = 1)

We use the same configuration for PPG only and PPG and acceleration windows (only the input shape changes).

We also define the number of epochs to use for training.

[29]:
```
ppg_cnn_config = {
    'model': {
        'input_shape': ppg_windows.shape[1:],
        'output_shape': targets.shape[1:],
        'n_convolutional_layers': 4,
        'kernel_size': 5,
        'n_initial_channels': 16,
        'use_normalization': False,
        'n_dense_layers': 3,
        'n_units': 128,
        'dropout': 0.0,
    },
    'optimizer': {
        'lr': 0.0001,
    },
}
ppg_acc_cnn_config = copy.deepcopy(ppg_cnn_config)
ppg_acc_cnn_config['model']['input_shape'] = ppg_acc_windows.shape[1:]
```

```
n_epochs = 30

print('PPG CNN config')
IPython.display.display(ppg_cnn_config)
print()
print('PPG ACC CNN config')
IPython.display.display(ppg_acc_cnn_config)
```

```
PPG CNN config

{'model': {'input_shape': (1, 200),
  'output_shape': (1,),
  'n_convolutional_layers': 4,
  'kernel_size': 5,
  'n_initial_channels': 16,
  'use_normalization': False,
  'n_dense_layers': 3,
  'n_units': 128,
  'dropout': 0.0},
 'optimizer': {'lr': 0.0001}}


PPG ACC CNN config

{'model': {'input_shape': (4, 200),
  'output_shape': (1,),
  'n_convolutional_layers': 4,
  'kernel_size': 5,
  'n_initial_channels': 16,
  'use_normalization': False,
  'n_dense_layers': 3,
  'n_units': 128,
  'dropout': 0.0},
 'optimizer': {'lr': 0.0001}}
```

We are ready to train the first two models with these configurations.

[30]:
```
ppg_cnn = train_model(
    config=ppg_cnn_config,
    windows=ppg_windows,
    targets=targets,
    i_train=i_train,
    i_val=i_val,
    n_epochs=n_epochs,
    name='ppg_cnn',
)

ppg_acc_cnn = train_model(
    config=ppg_acc_cnn_config,
```

```
    windows=ppg_acc_windows,
    targets=targets,
    i_train=i_train,
    i_val=i_val,
    n_epochs=n_epochs,
    name='ppg_acc_cnn',
)
```

```
   | Name             | Type              | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model            | CnnModel          | 87.2 K | train | [1, 1, 200]   | [1,
1]
1  | model.layers     | Sequential        | 87.2 K | train | [1, 1, 200]   | [1,
1]
2  | model.layers.0   | Conv1d            | 96     | train | [1, 1, 200]   | [1,
16, 200]
3  | model.layers.1   | ReLU              | 0      | train | [1, 16, 200]  | [1,
16, 200]
4  | model.layers.2   | MaxPool1d         | 0      | train | [1, 16, 200]  | [1,
16, 100]
5  | model.layers.3   | Conv1d            | 2.6 K  | train | [1, 16, 100]  | [1,
32, 100]
6  | model.layers.4   | ReLU              | 0      | train | [1, 32, 100]  | [1,
32, 100]
7  | model.layers.5   | MaxPool1d         | 0      | train | [1, 32, 100]  | [1,
32, 50]
8  | model.layers.6   | Conv1d            | 10.3 K | train | [1, 32, 50]   | [1,
64, 50]
9  | model.layers.7   | ReLU              | 0      | train | [1, 64, 50]   | [1,
64, 50]
10 | model.layers.8   | MaxPool1d         | 0      | train | [1, 64, 50]   | [1,
64, 25]
11 | model.layers.9   | Conv1d            | 41.1 K | train | [1, 64, 25]   | [1,
128, 25]
12 | model.layers.10  | ReLU              | 0      | train | [1, 128, 25]  | [1,
128, 25]
13 | model.layers.11  | AdaptiveAvgPool1d | 0      | train | [1, 128, 25]  | [1,
128, 1]
14 | model.layers.12  | Flatten           | 0      | train | [1, 128, 1]   | [1,
128]
15 | model.layers.13  | Linear            | 16.5 K | train | [1, 128]      | [1,
128]
16 | model.layers.14  | ReLU              | 0      | train | [1, 128]      | [1,
128]
17 | model.layers.15  | Linear            | 16.5 K | train | [1, 128]      | [1,
128]
```

```
18 | model.layers.16 | ReLU              | 0      | train | [1, 128]     | [1,
128]
19 | model.layers.17 | Linear            | 129    | train | [1, 128]     | [1,
1]
--------------------------------------------------------------------------------
-------------
87.2 K    Trainable params
0         Non-trainable params
87.2 K    Total params
0.349     Total estimated model params size (MB)
20        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

   | Name             | Type              | Params | Mode  | In sizes     | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model            | CnnModel          | 87.5 K | train | [1, 4, 200]  | [1,
1]
1  | model.layers     | Sequential        | 87.5 K | train | [1, 4, 200]  | [1,
1]
2  | model.layers.0   | Conv1d            | 336    | train | [1, 4, 200]  | [1,
16, 200]
3  | model.layers.1   | ReLU              | 0      | train | [1, 16, 200] | [1,
16, 200]
4  | model.layers.2   | MaxPool1d         | 0      | train | [1, 16, 200] | [1,
16, 100]
5  | model.layers.3   | Conv1d            | 2.6 K  | train | [1, 16, 100] | [1,
32, 100]
6  | model.layers.4   | ReLU              | 0      | train | [1, 32, 100] | [1,
32, 100]
7  | model.layers.5   | MaxPool1d         | 0      | train | [1, 32, 100] | [1,
32, 50]
8  | model.layers.6   | Conv1d            | 10.3 K | train | [1, 32, 50]  | [1,
64, 50]
9  | model.layers.7   | ReLU              | 0      | train | [1, 64, 50]  | [1,
64, 50]
10 | model.layers.8   | MaxPool1d         | 0      | train | [1, 64, 50]  | [1,
64, 25]
11 | model.layers.9   | Conv1d            | 41.1 K | train | [1, 64, 25]  | [1,
128, 25]
12 | model.layers.10  | ReLU              | 0      | train | [1, 128, 25] | [1,
128, 25]
13 | model.layers.11  | AdaptiveAvgPool1d | 0      | train | [1, 128, 25] | [1,
128, 1]
14 | model.layers.12  | Flatten           | 0      | train | [1, 128, 1]  | [1,
128]
15 | model.layers.13  | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
```

```
16 | model.layers.14 | ReLU              | 0      | train | [1, 128]     | [1,
128]
17 | model.layers.15 | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
18 | model.layers.16 | ReLU              | 0      | train | [1, 128]     | [1,
128]
19 | model.layers.17 | Linear            | 129    | train | [1, 128]     | [1,
1]
--------------------------------------------------------------------------------
-------------
87.5 K    Trainable params
0         Non-trainable params
87.5 K    Total params
0.350     Total estimated model params size (MB)
20        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

Question 2

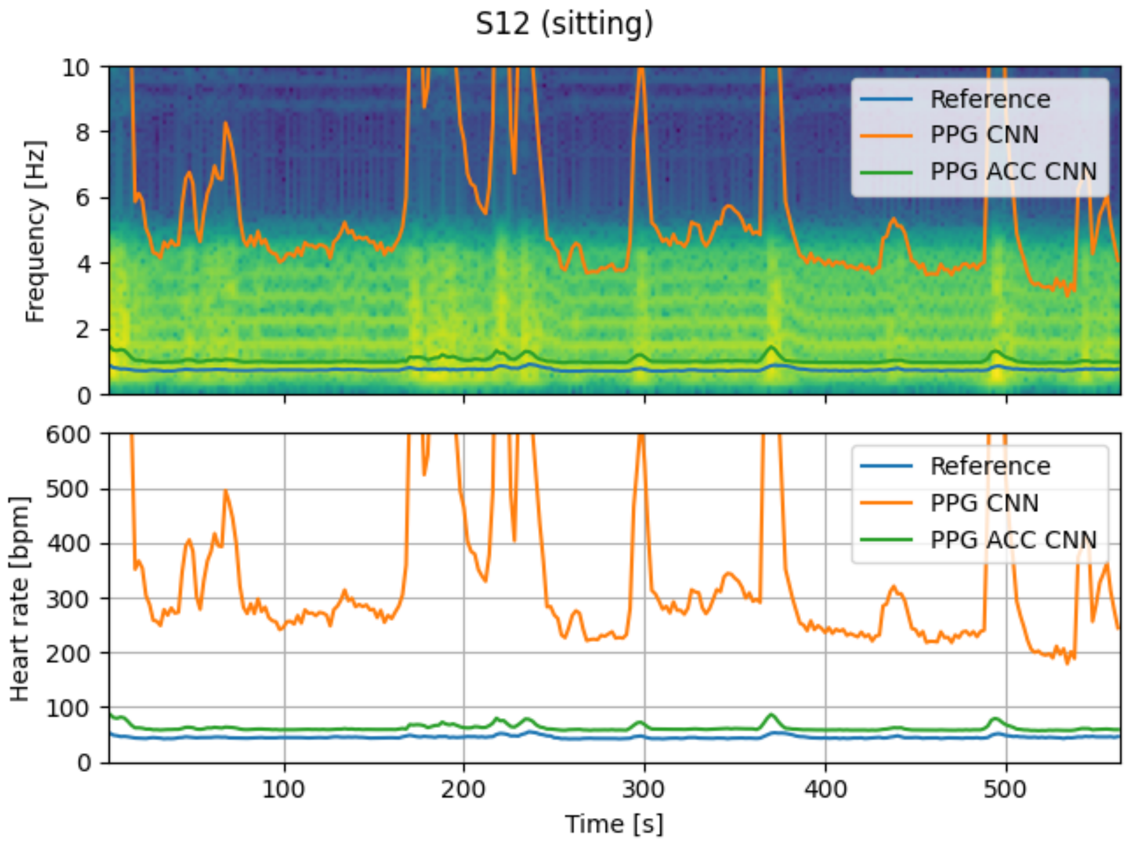
Based on metrics shown in TensorBoard, does using the acceleration signals in addition to the PPG signal help to improve performance?

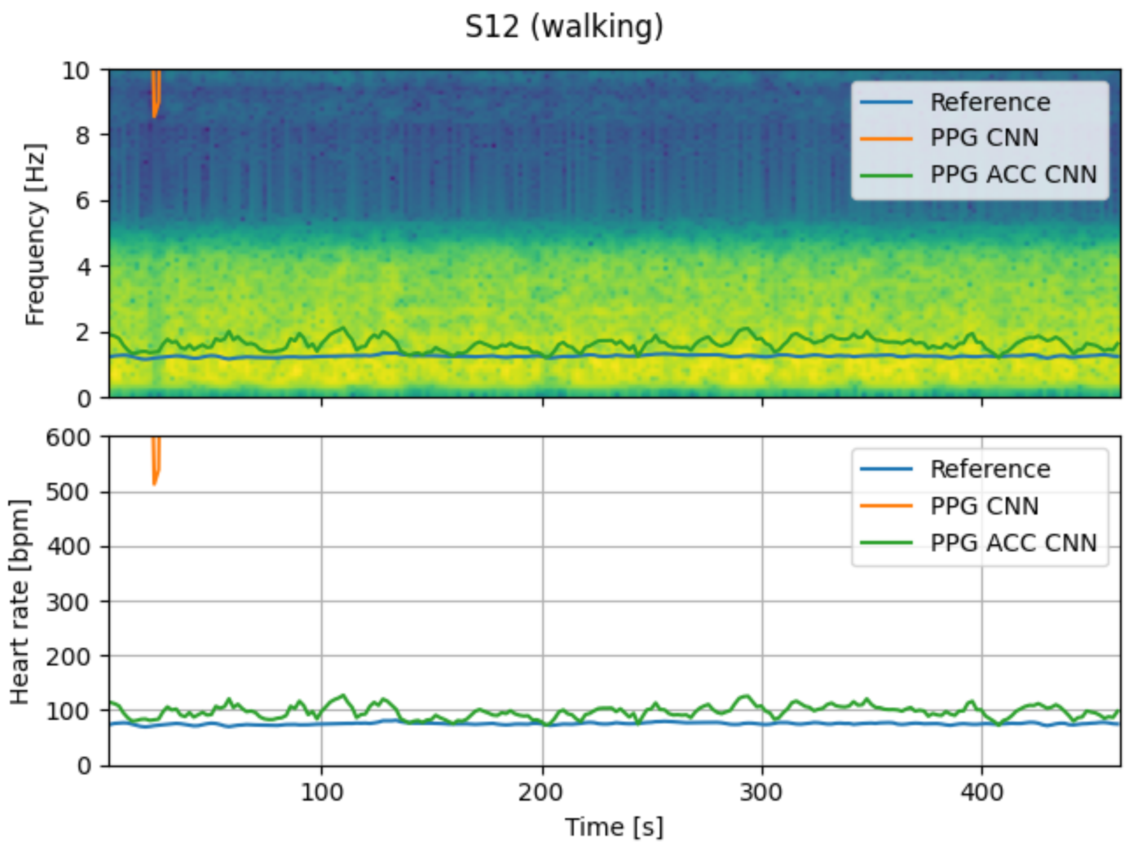
Answer Yes, for the acceleration model, we found a RMSE of 278.6604 for the validation set, while 534.7772 for the normal one. This shows that using the acceleration signals indeed improve the model performances. It can also be seen in the plot below that the estimation is more accurate for the predictions with acceleration signals.

We plot the heart rate predicted by these two models for two records from the validation set: one where the subject is sitting and on where the subject is walking.

[12]:
```python
def apply_model(model, record, alpha):
    windows, targets = extract_windows(record)
    if alpha.shape[0] == 1:
        windows = windows[:, :1, :]
    windows *= alpha
    loader = build_loader(windows, targets)

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        trainer = pl.Trainer(
            default_root_dir=LOG_DIRECTORY,
            logger=False,
            enable_progress_bar=False,
            enable_model_summary=False,
        )
        predictions = trainer.predict(model, loader)
    predictions = np.vstack([p.numpy() for p in predictions])

    return targets.ravel(), predictions.ravel()
```

```python
def plot_results(record, models, limits=(0.0, 600.0)):
    predictions = {}
    for name, (model, alpha) in models.items():
        _, predictions[name] = apply_model(model, record, alpha)

    signals = record['signals']
    hr = record['hr']

    limits = np.array(limits)
    fig, axes = plt.subplots(2, 1, sharex='all', constrained_layout=True)
    plt.suptitle(f'{record["subject"]} ({record["activity"]})')
    plt.sca(axes.flat[0])
    plt.specgram(signals['ppg'].to_numpy(), Fs=FS, NFFT=WINDOW_SIZE,
                 noverlap=WINDOW_SIZE - SHIFT_SIZE)
    plt.plot(hr['time'].to_numpy(), hr['hr'].to_numpy() / 60.0,
             label='Reference')
    for name, prediction in predictions.items():
        plt.plot(hr['time'].to_numpy(), prediction / 60.0, label=name)
    plt.ylim(limits / 60.0)
    plt.ylabel('Frequency [Hz]')
    plt.legend(loc='upper right')
    plt.sca(axes.flat[1])
    plt.plot(hr['time'].to_numpy(), hr['hr'].to_numpy(), label='Reference')
    for name, prediction in predictions.items():
        plt.plot(hr['time'].to_numpy(), prediction, label=name)
    plt.ylim(limits)
    plt.grid()
    plt.xlabel('Time [s]')
    plt.ylabel('Heart rate [bpm]')
    plt.legend(loc='upper right')


models = {
    'PPG CNN': (ppg_cnn, ppg_alpha),
    'PPG ACC CNN': (ppg_acc_cnn, ppg_acc_alpha),
}

plot_results(records[21], models)
plot_results(records[22], models)
```


S12 (sitting)
Frequency [Hz]
Reference
PPG CNN
PPG ACC CNN
Heart rate [bpm]
Time [s]



Question 3

What can you say about the predictions computed with the two models?

Answer As already mentioned in Question 2, the prediction with the model including acceleration signals is way more accurate. We can see in Figure 3 that this prediction sticks more to the reality, while the PPG without acceleration does not make sense at all.

Next, we modify the configuration of the model using both PPG and acceleration to add batch normalization after each convolution layer. We do not do the same for the model using only PPG since it does not work at all.

[31]:
```
ppg_acc_norm_cnn_config = copy.deepcopy(ppg_acc_cnn_config)
ppg_acc_norm_cnn_config['model']['use_normalization'] = True

print('PPG ACC norm CNN config')
IPython.display.display(ppg_acc_norm_cnn_config)
```

```
PPG ACC norm CNN config
```

```
{'model': {'input_shape': (4, 200),
  'output_shape': (1,),
  'n_convolutional_layers': 4,
  'kernel_size': 5,
```

```
 'n_initial_channels': 16,
 'use_normalization': True,
 'n_dense_layers': 3,
 'n_units': 128,
 'dropout': 0.0},
'optimizer': {'lr': 0.0001}}
```

We train a model with this new configuration.

[32]:
```
ppg_acc_norm_cnn = train_model(
    config=ppg_acc_norm_cnn_config,
    windows=ppg_acc_windows,
    targets=targets,
    i_train=i_train,
    i_val=i_val,
    n_epochs=n_epochs,
    name='ppg_acc_norm_cnn',
)
```

```
   | Name             | Type                 | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model            | CnnModel             | 88.0 K | train | [1, 4, 200]   | [1,
1]
1  | model.layers     | Sequential           | 88.0 K | train | [1, 4, 200]   | [1,
1]
2  | model.layers.0   | Conv1d               | 336    | train | [1, 4, 200]   | [1,
16, 200]
3  | model.layers.1   | BatchNorm1d          | 32     | train | [1, 16, 200] | [1,
16, 200]
4  | model.layers.2   | ReLU                 | 0      | train | [1, 16, 200] | [1,
16, 200]
5  | model.layers.3   | MaxPool1d            | 0      | train | [1, 16, 200] | [1,
16, 100]
6  | model.layers.4   | Conv1d               | 2.6 K  | train | [1, 16, 100] | [1,
32, 100]
7  | model.layers.5   | BatchNorm1d          | 64     | train | [1, 32, 100] | [1,
32, 100]
8  | model.layers.6   | ReLU                 | 0      | train | [1, 32, 100] | [1,
32, 100]
9  | model.layers.7   | MaxPool1d            | 0      | train | [1, 32, 100] | [1,
32, 50]
10 | model.layers.8   | Conv1d               | 10.3 K | train | [1, 32, 50]   | [1,
64, 50]
11 | model.layers.9   | BatchNorm1d          | 128    | train | [1, 64, 50]   | [1,
64, 50]
12 | model.layers.10  | ReLU                 | 0      | train | [1, 64, 50]   | [1,
64, 50]
```

```
13 | model.layers.11 | MaxPool1d         | 0      | train | [1, 64, 50]  | [1,
64, 25]
14 | model.layers.12 | Conv1d            | 41.1 K | train | [1, 64, 25]  | [1,
128, 25]
15 | model.layers.13 | BatchNorm1d       | 256    | train | [1, 128, 25] | [1,
128, 25]
16 | model.layers.14 | ReLU              | 0      | train | [1, 128, 25] | [1,
128, 25]
17 | model.layers.15 | AdaptiveAvgPool1d | 0      | train | [1, 128, 25] | [1,
128, 1]
18 | model.layers.16 | Flatten           | 0      | train | [1, 128, 1]  | [1,
128]
19 | model.layers.17 | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
20 | model.layers.18 | ReLU              | 0      | train | [1, 128]     | [1,
128]
21 | model.layers.19 | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
22 | model.layers.20 | ReLU              | 0      | train | [1, 128]     | [1,
128]
23 | model.layers.21 | Linear            | 129    | train | [1, 128]     | [1,
1]
--------------------------------------------------------------------------------
-------------
88.0 K    Trainable params
0         Non-trainable params
88.0 K    Total params
0.352     Total estimated model params size (MB)
24        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

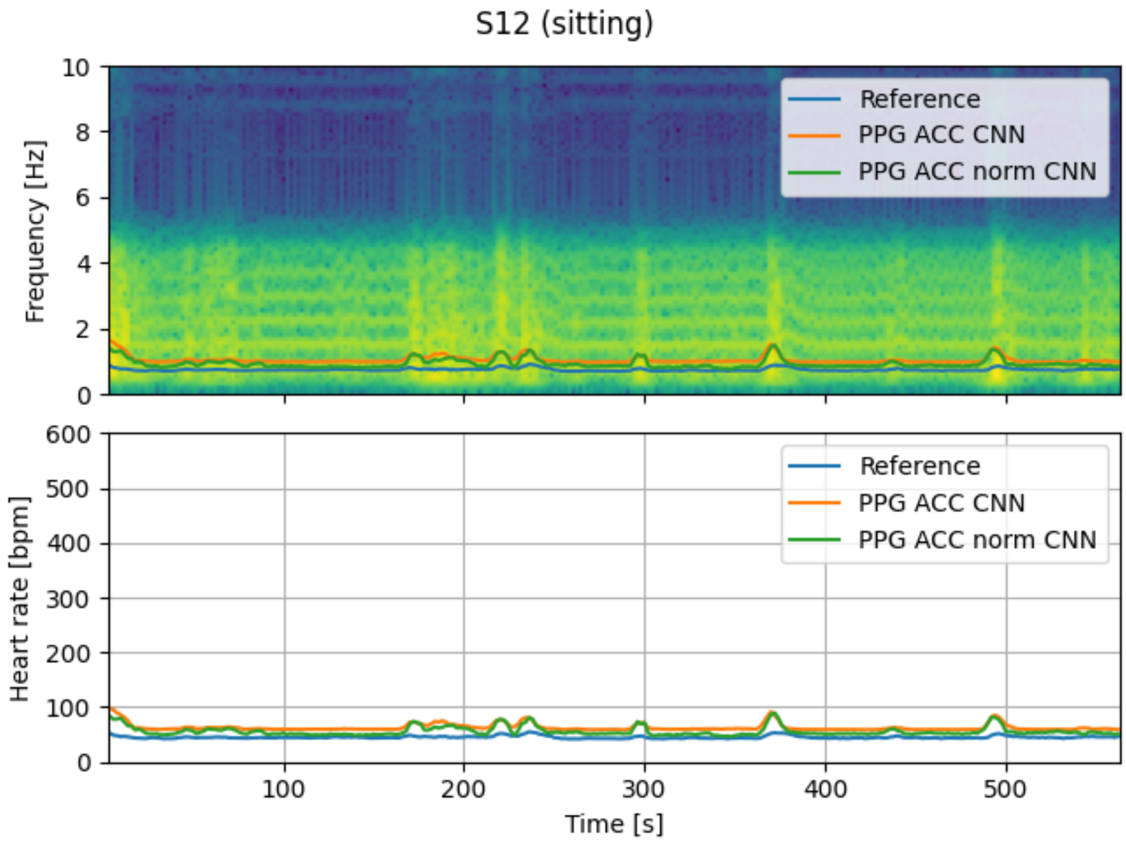
Question 4

What is the effect of batch normalization on the training procedure and on the performance metrics?

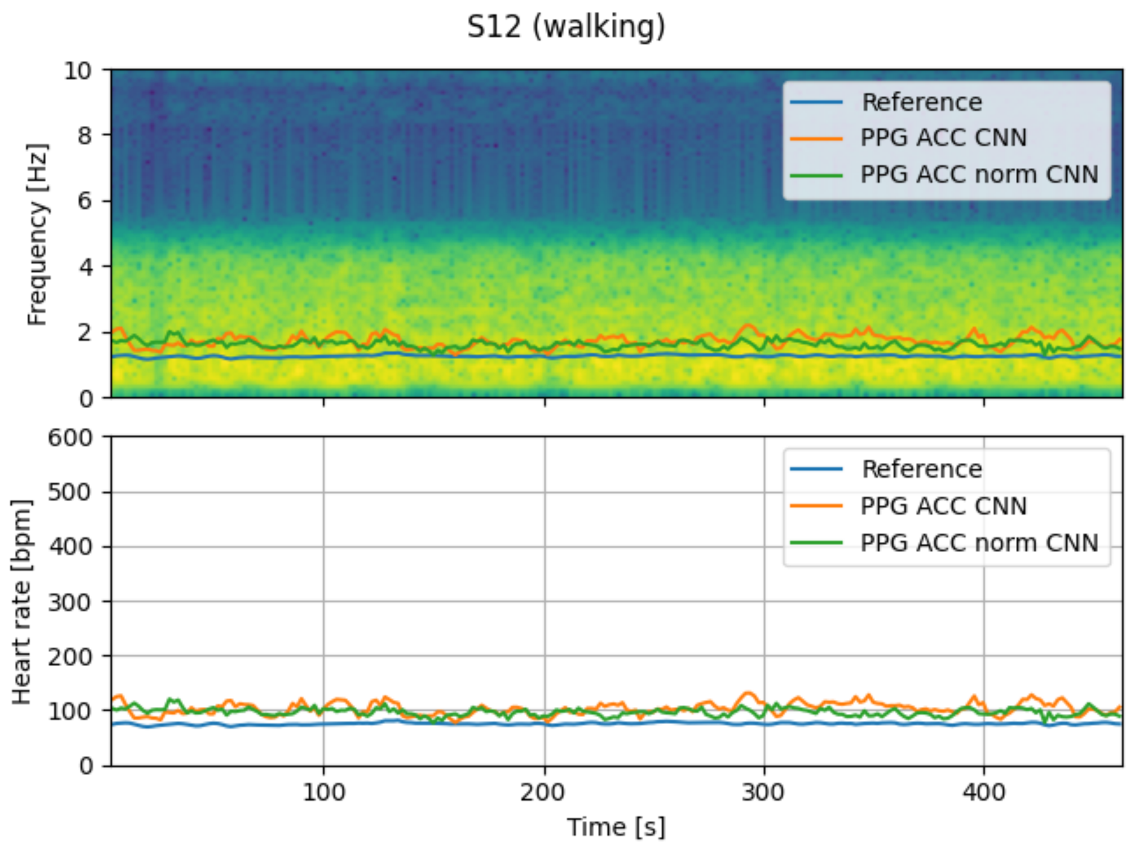
Answer Using batch normalization helps the training loss to converge faster as it reduces the variance inside each batch. Furthermore, inspecting training and validation RMSE show that BN helps achieving lower values.

We plot the predicted heart rate with respect to the reference for these new models (and we drop the one that did not work at all).

[33]:
```
models = {
    'PPG ACC CNN': (ppg_acc_cnn, ppg_acc_alpha),
    'PPG ACC norm CNN': (ppg_acc_norm_cnn, ppg_acc_alpha),
}

plot_results(records[21], models)
plot_results(records[22], models)
```



S12 (sitting)
Frequency [Hz]
Heart rate [bpm]
Time [s]
Reference
PPG ACC CNN
PPG ACC norm CNN
0
2
4
6
8
10
0
100
200
300
400
500
600
100
200
300
400
500



Question 5

Visually, is there a large difference between the CNN models with and without batch normalization on these examples?

Anser No, visually, no clear difference can be observed between the two predictions.

Finally, we try another configuration where we add dropout after each dense layer (except the last one which is the output layer).

[35]:
```
ppg_acc_norm_dropout_cnn_config = copy.deepcopy(ppg_acc_norm_cnn_config)
ppg_acc_norm_dropout_cnn_config['model']['dropout'] = 0.5

print('PPG ACC norm dropout CNN config')
IPython.display.display(ppg_acc_norm_dropout_cnn_config)
```

```
PPG ACC norm dropout CNN config
```

```
{'model': {'input_shape': (4, 200),
  'output_shape': (1,),
  'n_convolutional_layers': 4,
  'kernel_size': 5,
  'n_initial_channels': 16,
  'use_normalization': True,
```

```
  'n_dense_layers': 3,
  'n_units': 128,
  'dropout': 0.5},
 'optimizer': {'lr': 0.0001}}
```

And we train a new model with this configuration.

[36]:
```python
ppg_acc_norm_dropout_cnn = train_model(
    config=ppg_acc_norm_dropout_cnn_config,
    windows=ppg_acc_windows,
    targets=targets,
    i_train=i_train,
    i_val=i_val,
    n_epochs=n_epochs,
    name='ppg_acc_norm_dropout_cnn',
)
```

```
   | Name             | Type              | Params | Mode  | In sizes      | Out
sizes
--------------------------------------------------------------------------------
-------------
0  | model            | CnnModel          | 88.0 K | train | [1, 4, 200]   | [1,
1]
1  | model.layers     | Sequential        | 88.0 K | train | [1, 4, 200]   | [1,
1]
2  | model.layers.0   | Conv1d            | 336    | train | [1, 4, 200]   | [1,
16, 200]
3  | model.layers.1   | BatchNorm1d       | 32     | train | [1, 16, 200]  | [1,
16, 200]
4  | model.layers.2   | ReLU              | 0      | train | [1, 16, 200]  | [1,
16, 200]
5  | model.layers.3   | MaxPool1d         | 0      | train | [1, 16, 200]  | [1,
16, 100]
6  | model.layers.4   | Conv1d            | 2.6 K  | train | [1, 16, 100]  | [1,
32, 100]
7  | model.layers.5   | BatchNorm1d       | 64     | train | [1, 32, 100]  | [1,
32, 100]
8  | model.layers.6   | ReLU              | 0      | train | [1, 32, 100]  | [1,
32, 100]
9  | model.layers.7   | MaxPool1d         | 0      | train | [1, 32, 100]  | [1,
32, 50]
10 | model.layers.8   | Conv1d            | 10.3 K | train | [1, 32, 50]   | [1,
64, 50]
11 | model.layers.9   | BatchNorm1d       | 128    | train | [1, 64, 50]   | [1,
64, 50]
12 | model.layers.10  | ReLU              | 0      | train | [1, 64, 50]   | [1,
64, 50]
13 | model.layers.11  | MaxPool1d         | 0      | train | [1, 64, 50]   | [1,
64, 25]
```

```
14 | model.layers.12 | Conv1d            | 41.1 K | train | [1, 64, 25]  | [1,
128, 25]
15 | model.layers.13 | BatchNorm1d       | 256    | train | [1, 128, 25] | [1,
128, 25]
16 | model.layers.14 | ReLU              | 0      | train | [1, 128, 25] | [1,
128, 25]
17 | model.layers.15 | AdaptiveAvgPool1d | 0      | train | [1, 128, 25] | [1,
128, 1]
18 | model.layers.16 | Flatten           | 0      | train | [1, 128, 1]  | [1,
128]
19 | model.layers.17 | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
20 | model.layers.18 | ReLU              | 0      | train | [1, 128]     | [1,
128]
21 | model.layers.19 | Dropout           | 0      | train | [1, 128]     | [1,
128]
22 | model.layers.20 | Linear            | 16.5 K | train | [1, 128]     | [1,
128]
23 | model.layers.21 | ReLU              | 0      | train | [1, 128]     | [1,
128]
24 | model.layers.22 | Dropout           | 0      | train | [1, 128]     | [1,
128]
25 | model.layers.23 | Linear            | 129    | train | [1, 128]     | [1,
1]
--------------------------------------------------------------------------------
-------------
88.0 K    Trainable params
0         Non-trainable params
88.0 K    Total params
0.352     Total estimated model params size (MB)
26        Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
```

Question 6

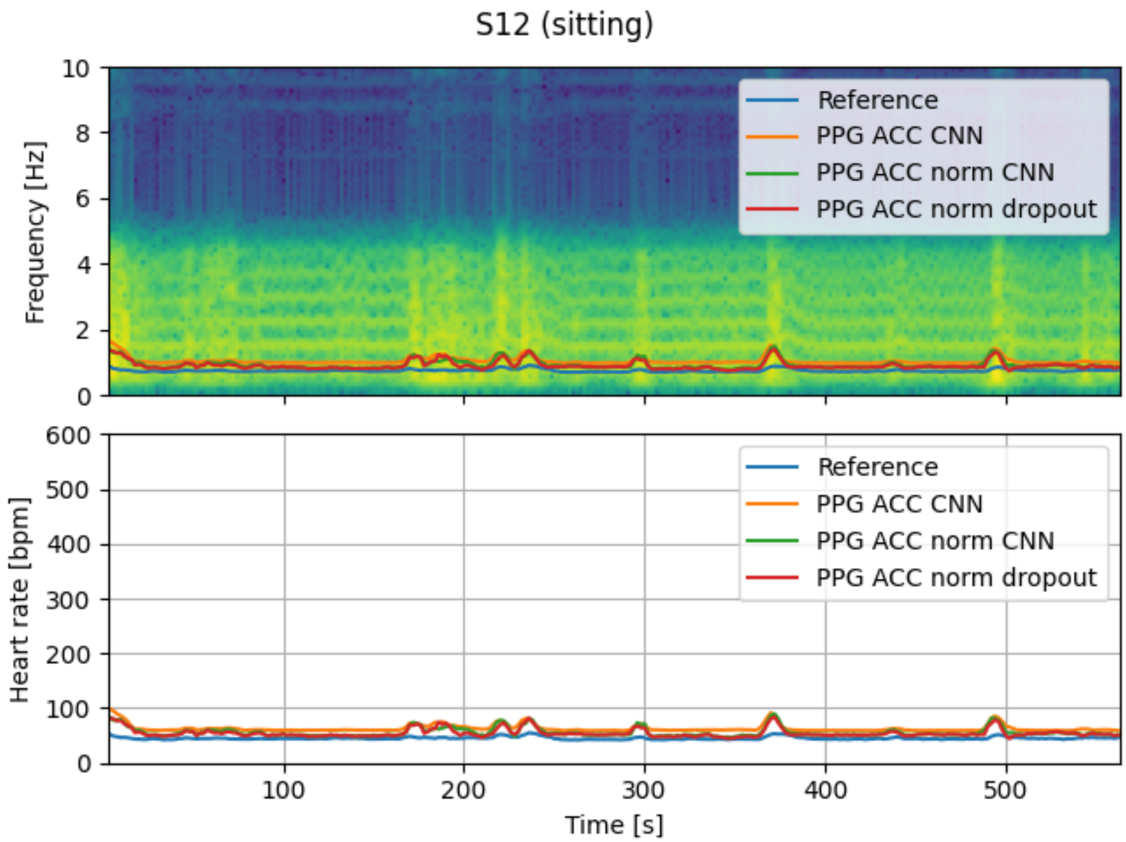
Does using dropout help to reduce overfitting compare to the same model without dropout?

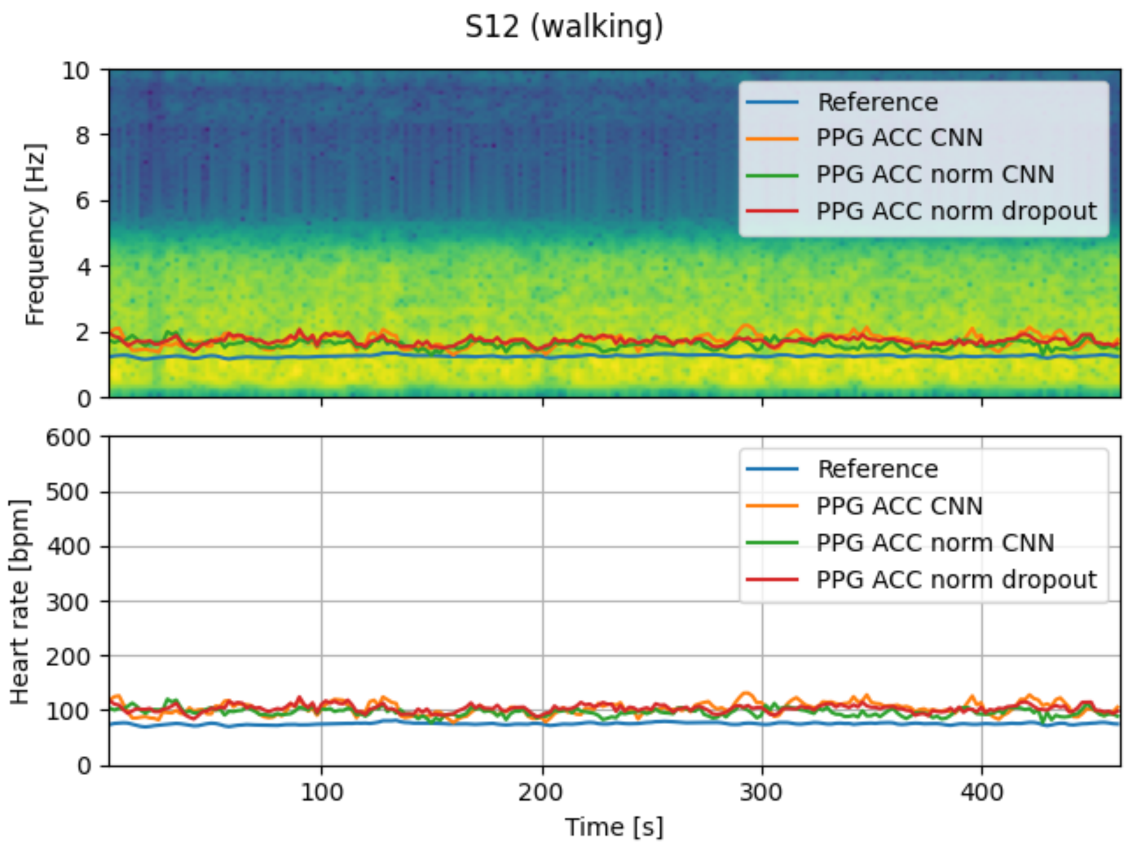
Answer Yes, applying dropout reduces overfitting. This can be observed since MSE and MAE are slightly higher for the training set when applying Dropout. This can be explained since dropout randomly selects nodes at each steps, preventing nodes to be too specific to the data. Nevertheless, this difference remains small.

Again, we plot the predicted heart rate for two records from the validation set.

[37]:
```
models = {
    'PPG ACC CNN': (ppg_acc_cnn, ppg_acc_alpha),
    'PPG ACC norm CNN': (ppg_acc_norm_cnn, ppg_acc_alpha),
    'PPG ACC norm dropout': (ppg_acc_norm_dropout_cnn, ppg_acc_alpha),
}
```

```
plot_results(records[21], models)
plot_results(records[22], models)
```





Finally, we compute performance metrics (MSE and MAE) for all models on the subsets for training, validation, and testing.

[19]:
```
models = {
    'PPG CNN': ppg_cnn,
    'PPG ACC CNN': ppg_acc_cnn,
    'PPG ACC norm CNN': ppg_acc_norm_cnn,
    'PPG ACC norm dropout': ppg_acc_norm_dropout_cnn,
}

metrics = []
for name, model in models.items():
    if 'ACC' in name:
        windows = ppg_acc_windows
    else:
        windows = ppg_windows
    df = evaluate_model(model, windows, targets, i_train, i_val, i_test)
    df.insert(0, 'model', name)
    metrics.append(df)
metrics = pd.concat(metrics, axis=0, ignore_index=True)
metrics = metrics.set_index(['model', 'subset'])
```

```
index = metrics.index.get_level_values(0).unique()
columns = pd.MultiIndex.from_product([metrics.columns, metrics.index.
  ↪get_level_values(1).unique()])
metrics = metrics.unstack().reindex(index=index, columns=columns)
IPython.display.display(metrics)
```

```
                     count                   mse                             \
subset               train   val  test       train         val         test
model
PPG CNN               4360  1530  1530  253.105415  519.009044  199.510891
PPG ACC CNN           4360  1530  1530  127.893587  263.449593  198.234142
PPG ACC norm CNN      4360  1530  1530   27.097020  230.597111   71.887220
PPG ACC norm dropout  4360  1530  1530   30.607524  240.611002   60.324805

                            mae
subset                    train        val       test
model
PPG CNN               12.369010  19.012989  12.443980
PPG ACC CNN            8.779760  13.099361  12.045710
PPG ACC norm CNN       3.844666  11.438116   6.177527
PPG ACC norm dropout   4.044759  11.505437   5.498473
```

Question 7

What is the best model in terms of MSE and MAE? What can you say about batch normalizaton and dropout?

Answer Based on the table, the best mse, as well as the mae on the test set is achieved for the model with batch normalization and dropout. This highlights the theoretical aspects of both methods, effectively preventing overfitting. But regarding the validation set, the model without dropout seems to perform better. With this difference, it is therefore difficult to ensure that dropout really helps in this case.