Deep learning enables sleep staging from photoplethysmogram for patients with suspected sleep apnea

Korkalainen H[1,2], Aakko J[3], Duce B[4,5], Kainulainen S[1,2], Leino A[1,2], Nikkonen S[1,2], Afara I O[1,6], Myllymaa S[1,2], Töyräs J[1,2,6], Leppänen T[1,2]

[1] *Department of Applied Physics, University of Eastern Finland, Kuopio, Finland*

[2] *Diagnostic Imaging Center, Kuopio University Hospital, Kuopio, Finland*

[3] *CGI Suomi Oy, Helsinki, Finland*

[4] *Sleep Disorders Centre, Princess Alexandra Hospital, Brisbane, Australia*

[5] *Institute of Health and Biomedical Innovation, Queensland University of Technology, Brisbane, Australia*

[6] *School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia*

1 Supplementary material

In this study, we developed a combined convolutional and recurrent neural network to automatically identify sleep stages from a photoplethysmogram signal obtained with a finger pulse oximeter. The complete neural network structure includes a time distributed layer of the convolutional neural network (CNN) which is then followed by a dropout layer and a bidirectional gated recurrent unit (GRU) layer. The neural network was trained using sequences of hundred 30-second epochs of the PPG signals downsampled to 64 Hz and overlap of 75% was used between consecutive sequences in the training set. The sequence forming protocol was repeated until the complete PPG signals were utilized.

The following sections include the Python implementation of the models using Keras and an example of how the model can be formed and compiled with the functions.

1.1 Python functions of the neural network

Below, the functions used to form the neural network are presented.

```
# The combined convolutional and recurrent neural network developed in the study:
# H Korkalainen, J Aakko et al. "Deep learning enables sleep staging from
# photoplethysmogram for patients with suspected sleep apnea"

# Contact information: henri.korkalainen@uef.fi

from keras.layers import Input, Conv1D, BatchNormalization
```

```
from keras.layers import Activation, MaxPooling1D, GlobalAveragePooling1D
from keras.layers import Dense, GaussianDropout, TimeDistributed
from keras.layers import Bidirectional, GRU
from keras.models import Model

def build_base_cnn_model(input_size = 1920, input_channels = 1,
                         init_conv_kernel_size = 21, ksize = 5,
                         conv_stride_first = 5,
                         conv_stride_rest = 1,
                         maxpool_size = 2, maxpool_stride = 2,
                         act_function = "relu", fs =64):
    # input_size = length of a single PPG-epoch
    # input_channels = number of PPG channels
    # init_conv_kernel_size = kernel size for the first convolution layer
    # ksize = kernel size for the rest convolutional layers
    # conv_stride_first = stride for the first convolution layer
    # conv_stride_rest = stride for the remaining convolution layers
    # maxpool_size = pool size for the max-pooling
    # maxpool_stride = stride size for the max-pooling
    # act_function = activation function
    # fs = the sampling frequency

    input1 = Input(shape = (input_size, input_channels), name = "input")
    x = Conv1D(fs, kernel_size = init_conv_kernel_size,
                         strides = conv_stride_first)(input1)
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    x = Conv1D(fs, kernel_size = init_conv_kernel_size,
                          strides = conv_stride_rest)(x)
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    x = MaxPooling1D(pool_size = maxpool_size, strides = maxpool_stride)(x)

    x = Conv1D(2*fs, kernel_size = ksize, strides = conv_stride_rest)(x)
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    x = Conv1D(2*fs, kernel_size = ksize, strides = conv_stride_rest)(x)
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    x = MaxPooling1D(pool_size = maxpool_size, strides = maxpool_stride)(x)

    x = Conv1D(4*fs, kernel_size = ksize, strides = conv_stride_rest)(x)
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    x = Conv1D(4*fs, kernel_size = ksize, strides = conv_stride_rest)(x)
```

```
    x = BatchNormalization()(x)
    x = Activation(act_function)(x)

    out = GlobalAveragePooling1D()(x)
    model = Model(inputs = input1, outputs = out)
    return model

def build_cnn_to_rnn_model(input_size = 1920, input_channels = 1, n_categories = 3,
                           seq_length = None,init_conv_kernel_size = 21, ksize = 5,
                           conv_stride_first = 5, conv_stride_rest = 1,
                           maxpool_size = 2, maxpool_stride = 2,
                           gru_units_multiplier = 4,rdo = 0.5, do = 0.3, gdo = 0.3,
                           act_function = "relu", fs = 64):
    # input_size = length of a single PPG-epoch
    # input_channels = number of PPG channels
    # n_categories = number of different sleep stages
    # seq_length = number of epochs in the input.
    # With seq_length = 'None' the model accepts any sequence length
    # init_conv_kernel_size = kernel size for the first convolution layer
    # ksize = kernel size for the rest convolutional layers
    # conv_stride_first = stride for the first convolution layer
    # conv_stride_rest = stride for the remaining convolution layers
    # maxpool_size = pool size for the max-pooling
    # maxpool_stride = stride size for the max-pooling
    # gru_units_multiplier * fs = number of GRU units
    # rdo = recurrent dropout size
    # do = (forward) drop out size
    # gdo = Gaussian dropout size
    # act_function = activation function
    # fs = the sampling frequency

    seq_input = Input(shape=(seq_length, input_size, input_channels))

    base_model = build_base_cnn_model(input_size = input_size,
                              input_channels = input_channels,
                              init_conv_kernel_size = init_conv_kernel_size,
                              ksize = ksize,
                              conv_stride_first = conv_stride_first,
                              conv_stride_rest = conv_stride_rest,
                              maxpool_size = maxpool_size,
                              maxpool_stride = maxpool_stride,
                              act_function=act_function,
                              fs = fs)

    encoded_sequence = TimeDistributed(base_model)(seq_input)

    encoded_sequence = GaussianDropout(gdo)(encoded_sequence)

    encoded_sequence = Bidirectional(GRU(gru_units_multiplier*fs,
                                         return_sequences = True,
```

```
                        recurrent_dropout = rdo,
                        dropout = do))(encoded_sequence)

    out = TimeDistributed(Dense(n_categories, activation = "softmax"))(encoded_sequence)

    model = Model(inputs = seq_input, outputs = out)

    return model
```

1.2 Compiling the neural network

Below, an example of how the model can be formed and compiled is presented. The input must be a 4D tensor with shape (number of sequences, length of a single sequence, sampling frequency * 30 s, number of channels).

```
cnnrnn = build_cnn_to_rnn_model(input_channels = n_channels,
        n_categories = n_stages, act_function = "relu",
        do = 0.3, rdo = 0.5, gdo = 0.3)

cnnrnn.compile(loss = 'categorical_crossentropy', optimizer =
        'adam', metrics = ['accuracy'])
```