Title

Physics-Informed Attention Temporal Convolutional Network for EEG-Based Motor Imagery Classification

Abstract

Brain-computer interface (BCI) is a cutting-edge technology with the potential to change the world. Electroencephalogram (EEG) motor imagination (MI) signals have been widely used in many BCI applications to help people with disabilities control devices or environments or even enhance human abilities. ML methods that rely on manual feature extraction use feature extraction methods based on the time-frequency domain. Deep learning can learn unique underlying features from raw EEG data without the need for pre-processing or manual feature extraction. Based on the EEGNet feature extraction method, a high-performance ATCNet model is proposed, which utilizes the powerful functions of TCN, attention mechanism and convolution based sliding window.

Keywords: BCI sliding window TCN classification

1 Introduction

EEG based motor imagination activities have been used in a variety of medical applications, including stroke rehabilitation, wheelchair control, prosthetic limb control, exoskeleton control, cursor control, spellers, and mind-to-text conversion. MI-EEG signals are also used in non-medical applications such as vehicle control, drone control, environmental control, smart home, security, gaming, and virtual reality. Electroencephalography (EEG) is a non-invasive method of recording the electrical activity of the brain. EEG signals are captured on the scalp as a two-dimensional matrix (time and channel) of true values. EEG is widely used and superior to other technologies due to its ease of use, low cost, low risk, portability, and high temporal resolution, making it suitable for industrial applications. When it comes to the classification of brain-computer interfaces (BCIs), a common approach is to classify them according to how they interact with the brain and how they work. Brain-computer interfaces can be divided into invasive and non-invasive types based on contact methods. Invasive brain-computer interfaces, which require direct implantation into brain tissue and are typically achieved with microelectrode arrays or deep stimulation devices, have high signal quality and fine spatial resolution, but can come with surgical risks and long-term stability issues. Non-invasive brain-computer interfaces, which use sensors on the scalp (such as electroencephalography, functional magnetic resonance imaging, or functional near-infrared spectroscopy) to obtain information about brain activity, do not require surgical implantation, but have relatively low signal quality and spatial resolution. Another way to classify brain-computer interfaces is to classify them as active and passive based on how they work. Active BCI works by interpreting brain signals and translating them into instructions to control external devices or perform specific tasks, such as neurofeedback training and brain-controlled prosthetics. Passive brain-computer interfaces are used to monitor and record brain activity, such as EEG systems for diagnostic or research purposes.

2 Method

2.1 Overview

This model approach combines TCN, moving Windows, and attention mechanisms. First, the features are transformed through the convolution layer, and the sliding window is used for high-performance computing to accelerate the computing speed. Extract information from features through multiple modules containing attention and TCN.

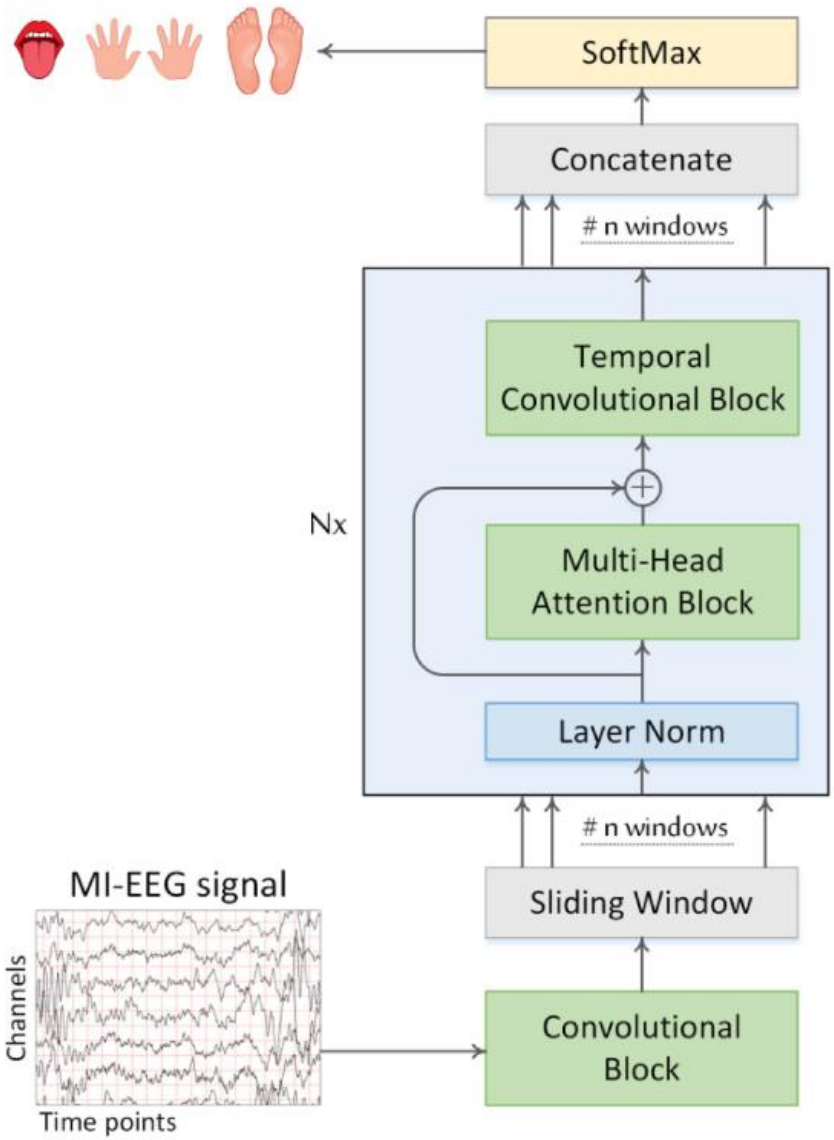

Figure 1. Overview of the method

2.2 Feature extraction

This module uses multiple convolution, mimics EEGNet architecture, extracts one dimensional feature from time and space dimension, and finally carries on feature mapping to map the feature dimension to the dimension we want.

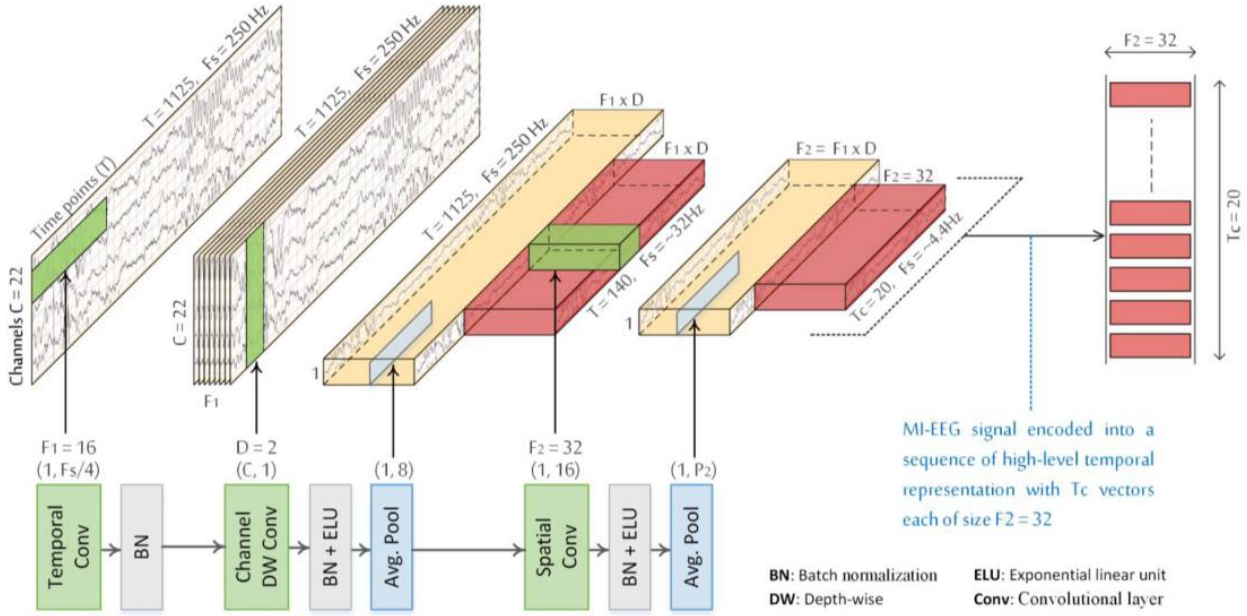

Figure 2. Feature extraction

2.3 TCN

The module consists of two parts: causal convolution and void convolution. Causal convolution ensures the timing of temporal information, so that the network only focuses on the input before the change. Hollow convolution accelerates the expansion of receptive field.

3 Implementation details

3.1 Reprocessing

The EEG data was preprocessed using MNE data packet. First, a band pass filter of 5 to 35 is passed. Secondly, the three ophthalmic electrical channels in the original data set were eliminated, and the classification was not carried out through them. Finally, the EEG data were segmented according to the marks made during the experiment, and the data segment of 4.5 seconds during the experiment was taken. This included 1,126 data points.

```
# Mention the file path to the dataset

filename = "E:/BCI_data/A01T.gdf"

raw = mne.io.read_raw_gdf(filename)

# Find the events time positions

events, _ = mne.events_from_annotations(raw)

# Pre-load the data

raw.load_data()

# Filter the raw signal with a band pass filter in 7-35 Hz

raw.filter(7., 35., fir_design='firwin')

# Remove the EOG channels and pick only desired EEG channels

raw.info['bads'] += ['EOG-left', 'EOG-central', 'EOG-right']

picks = mne.pick_types(raw.info, meg=False, eeg=True, eog=False, stim=False,
                       exclude='bads')

# Extracts epochs of 3s time period from the datset into 288 events for all 4 classes

tmin, tmax = 0, 4.5
# left_hand = 769,right_hand = 770,foot = 771,tongue = 772
event_id = dict({'769': 7,'770': 8,'771': 9,'772': 10})

epochs = mne.Epochs(raw, events, event_id, tmin, tmax, proj=True, picks=picks,
                    baseline=None, preload=True)

# Getting labels and changing labels from 7,8,9,10 -> 1,2,3,4
labels = epochs.events[:,-1] - 7 + 1

data = epochs.get_data()

print(data.shape)
print(labels.shape)
```

Figure 3. data processing

```
Extracting EDF parameters from E:\BCI_data\A01T.gdf...
GDF file detected
Setting channel info structure...
Could not determine channel type of the following channels, they will be set as EEG:
EEG-Fz, EEG, EEG, EEG, EEG, EEG, EEG-C3, EEG, EEG-Cz, EEG, EEG-C4, EEG, EEG, EEG, EEG, EEG, EEG, EEG-Pz, EEG, EEG, EOG-left,
EOG-central, EOG-right
Creating raw.info structure...
Used Annotations descriptions: ['1023', '1072', '276', '277', '32766', '768', '769', '770', '771', '772']
Reading 0 ... 672527  =      0.000 ...  2690.108 secs...

C:\Users\lenovo\anaconda3\Lib\contextlib.py:144: RuntimeWarning: Channel names are not unique, found duplicates for: {'EEG'}. Applying
running numbers for duplicates.
  next(self.gen)

Filtering raw data in 1 contiguous segment
Setting up band-pass filter from 7 - 35 Hz

FIR filter parameters
---------------------
Designing a one-pass, zero-phase, non-causal bandpass filter:
- Windowed time-domain design (firwin) method
- Hamming window with 0.0194 passband ripple and 53 dB stopband attenuation
- Lower passband edge: 7.00
- Lower transition bandwidth: 2.00 Hz (-6 dB cutoff frequency: 6.00 Hz)
- Upper passband edge: 35.00 Hz
- Upper transition bandwidth: 8.75 Hz (-6 dB cutoff frequency: 39.38 Hz)
- Filter length: 413 samples (1.652 s)

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s

Not setting metadata
288 matching events found
No baseline correction applied
0 projection items activated
Using data from preloaded Raw for 288 events and 1126 original time points ...
0 bad epochs dropped
(288, 22, 1126)
(288,)
```

Figure 4. EEG processing results

3.2 Data conversion

Change the data type, change the label to a unique thermal code, easy to use later training.

```
#data = torch.from_numpy(data)
data = torch.Tensor(data)
data = data.reshape(288, 1, 22, -1)
#data = data.double()
data = data.to(torch.float32)

labels = labels.astype(np.int64)
labels = torch.from_numpy(labels)
labels = nn.functional.one_hot(labels)
_, labels = torch.split(labels, [1,4], dim=1)
labels = labels.to(torch.float32)

print(data[0])
print(labels)
print(data.shape)
print(labels.shape)
```

Figure 5. Data conversion

3.3 Convolutional Blocks

```
class ConvBlock(nn.Module):
    def __init__(self, dropout, **kwargs):
        super(ConvBlock, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.Temporal_Conv = nn.Conv2d(1, 16, kernel_size=(1,64), padding="same")
        self.Channel_DW_Conv = nn.Conv2d(16, 32, kernel_size=(22,1))
        self.Spatial_Conv = nn.Conv2d(32, 32, kernel_size=(1,16), padding="same")
        self.window = nn.Conv2d(32, 32, kernel_size=(1,5))
        self.Convolutional_Block = nn.Sequential(self.Temporal_Conv, nn.BatchNorm2d(16),
                    self.Channel_DW_Conv, nn.BatchNorm2d(32), nn.ELU(),
                    nn.AvgPool2d(kernel_size=(1,8)),
                    self.Spatial_Conv, nn.BatchNorm2d(32), nn.ELU(),
                    nn.AvgPool2d(kernel_size=(1,7)),
                    self.window, nn.Flatten(2,3))
    def forward(self, X):
        return self.Convolutional_Block(X)
```

Figure 6. Convolutional Blocks

3.4 Attention

```python
class MultiHeadAttention(nn.Module):
    """多头注意力"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # queries，keys，values的形状:
        # (batch_size，查询或者“键－值”对的个数，num_hiddens)
        # valid_lens  的形状:
        # (batch_size，)或(batch_size，查询的个数)
        # 经过变换后，输出的queries，keys，values  的形状:
        # (batch_size*num_heads，查询或者“键－值”对的个数，
        # num_hiddens/num_heads)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            # 在轴0，将第一项（标量或者矢量）复制num_heads次，
            # 然后如此复制第二项，然后诸如此类。
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # output的形状:(batch_size*num_heads，查询的个数，
        # num_hiddens/num_heads)
        output = self.attention(queries, keys, values, valid_lens)

        # output_concat的形状:(batch_size，查询的个数，num_hiddens)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)

def transpose_qkv(X, num_heads):
    """为了多注意力头的并行计算而变换形状"""

    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    X = X.permute(0, 2, 1, 3)

    return X.reshape(-1, X.shape[2], X.shape[3])

def transpose_output(X, num_heads):
    """逆转transpose_qkv函数的操作"""

    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])

    X = X.permute(0, 2, 1, 3)

    return X.reshape(X.shape[0], X.shape[1], -1)
```

Figure 7. Attention

3.5 TCN Block

```
class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class TemporalBlock(nn.Module):
    def __init__(self, n_inputs, n_outputs, kernel_size, stride, dilation, padding, dropout=0.3):
        super(TemporalBlock, self).__init__()
        self.conv1 = weight_norm(nn.Conv1d(n_inputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp1 = Chomp1d(padding)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(dropout)

        self.conv2 = weight_norm(nn.Conv1d(n_outputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp2 = Chomp1d(padding)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)

        self.net = nn.Sequential(self.conv1, self.chomp1, self.relu1, self.dropout1,
                                 self.conv2, self.chomp2, self.relu2, self.dropout2)
        self.downsample = nn.Conv1d(n_inputs, n_outputs, 1) if n_inputs != n_outputs else None
        self.relu = nn.ReLU()
        self.init_weights()

    def init_weights(self):
        self.conv1.weight.data.normal_(0, 0.01)
        self.conv2.weight.data.normal_(0, 0.01)
        if self.downsample is not None:
            self.downsample.weight.data.normal_(0, 0.01)

    def forward(self, x):
        out = self.net(x)
        res = x if self.downsample is None else self.downsample(x)
        return self.relu(out + res)


class TemporalConvNet(nn.Module):
    def __init__(self, num_inputs, num_channels, kernel_size=2, dropout=0.2):
        super(TemporalConvNet, self).__init__()
        layers = []
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i-1]
            out_channels = num_channels[i]
            layers += [TemporalBlock(in_channels, out_channels, kernel_size, stride=1, dilation=dilation_size,
                                     padding=(kernel_size-1) * dilation_size, dropout=dropout)]

        self.network = nn.Sequential(*layers)

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

Figure 8. TCN Block

3.6 Classifier

```
class EncoderBlock(nn.Module):

    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.num_channels=[32]
        self.ln = nn.LayerNorm(16)
        self.attention = MultiHeadAttention(key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm = AddNorm(dropout)
        self.tcn = TemporalConvNet(32, self.num_channels, kernel_size=4, dropout=0.5)

    def forward(self, X, valid_lens):
        X = self.ln(X)
        Y = self.attention(X, X, X, valid_lens)
        Y = self.addnorm(X,Y)
        return  self.tcn(Y)
```

Figure 9. classifier

3.7 Residual module

```
class AddNorm(nn.Module):
    def __init__(self, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    def forward(self, X, Y):
        return self.dropout(Y) + X
```

Figure 10. Residual module

3.8 Overall model

```
class Encoder(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.ConvBlock = ConvBlock(dropout)
        self.blks = nn.Sequential()
        num_layers = 2
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                  num_heads, dropout))
        self.fl = nn.Flatten(1,2)
        self.sm = nn.Linear(32*16, 4, bias=False)

    def forward(self, X, valid_lens, *args):
        X = self.ConvBlock(X)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
        flx = self.fl(X)
        res = self.sm(flx)
        return res
```

Figure 11. overall model

4 Results and analysis

The experiment verifies that each component of ATCNet has a positive effect on the overall performance of the network. From the table, we can find that which module to remove has different degrees of loss of performance.

Removed block	Accuracy %	κ-score
None (ATCNet)	85.38	0.805
AT	83.84	0.784
SW	83.10	0.775
SW + AT	82.75	0.770
TC	79.44	0.726
SW + TC	80.48	0.740
AT + TC	82.60	0.768
SW + AT + TC	81.71	0.756

Figure 12. Results1

We compared our proposed model with other models that performed well in the same period, including EEGNet and EEG-TCNet. The experiment shows that our model is better than other models of the same period in accuracy and various data.

	Proposed (ATCNet)		EEGNet [12]		EEG-TCNet [22]	
Sub.	%	κ	%	κ	%	κ
1	88.5	0.85	88.5	0.85	84.0	0.79
2	70.5	0.61	66.0	0.55	66.3	0.55
3	97.6	0.97	95.1	0.94	94.1	0.92
4	81.0	0.75	73.6	0.65	72.6	0.63
5	83.0	0.77	75.4	0.67	76.0	0.68
6	73.6	0.65	64.2	0.52	62.9	0.50
7	93.1	0.91	90.3	0.87	89.9	0.87
8	90.3	0.87	85.8	0.81	84.7	0.80
9	91.0	0.88	86.5	0.82	85.4	0.81
Mean	**85.4**	**0.81**	80.6	0.74	79.6	0.73
St.D.	**9.1**	**0.12**	11.1	0.15	10.7	0.14

Figure 13. Results2

References

[1] H. Altaheri, G. Muhammad和M. Alsulaiman, 《Physics-Informed Attention Temporal Convolutional Network for EEG-Based Motor Imagery Classification》, *IEEE Trans. Ind. Inf.*, 2249–2258, 2023, doi: 10.1109/TII.2022.3197419.