jerrybai1995 Added missing files; updated documentations

Latest commit 029afeb on Jun 6, 2019 History

2 contributors

146 lines (126 sloc) | 6.48 KB Raw Blame

```python
import torch
from torch import nn
import torch.nn.functional as F

from modules.transformer import TransformerEncoder


class MULTModel(nn.Module):
    def __init__(self, hyp_params):
        """
        Construct a MulT model.
        """
        super(MULTModel, self).__init__()
        self.orig_d_l, self.orig_d_a, self.orig_d_v = hyp_params.orig_d_l, hyp_params.orig_d_a, hyp_params.orig_d_v
        self.d_l, self.d_a, self.d_v = 30, 30, 30
        self.vonly = hyp_params.vonly
        self.aonly = hyp_params.aonly
        self.lonly = hyp_params.lonly
        self.num_heads = hyp_params.num_heads
        self.layers = hyp_params.layers
        self.attn_dropout = hyp_params.attn_dropout
        self.attn_dropout_a = hyp_params.attn_dropout_a
        self.attn_dropout_v = hyp_params.attn_dropout_v
        self.relu_dropout = hyp_params.relu_dropout
        self.res_dropout = hyp_params.res_dropout
        self.out_dropout = hyp_params.out_dropout
        self.embed_dropout = hyp_params.embed_dropout
        self.attn_mask = hyp_params.attn_mask

        combined_dim = self.d_l + self.d_a + self.d_v

        self.partial_mode = self.lonly + self.aonly + self.vonly
        if self.partial_mode == 1:
            combined_dim = 2 * self.d_l   # assuming d_l == d_a == d_v
        else:
            combined_dim = 2 * (self.d_l + self.d_a + self.d_v)

        output_dim = hyp_params.output_dim        # This is actually not a hyperparameter :-)

        # 1. Temporal convolutional layers
        self.proj_l = nn.Conv1d(self.orig_d_l, self.d_l, kernel_size=1, padding=0, bias=False)
        self.proj_a = nn.Conv1d(self.orig_d_a, self.d_a, kernel_size=1, padding=0, bias=False)
        self.proj_v = nn.Conv1d(self.orig_d_v, self.d_v, kernel_size=1, padding=0, bias=False)

        # 2. Crossmodal Attentions
        if self.lonly:
            self.trans_l_with_a = self.get_network(self_type='la')
            self.trans_l_with_v = self.get_network(self_type='lv')
        if self.aonly:
            self.trans_a_with_l = self.get_network(self_type='al')
            self.trans_a_with_v = self.get_network(self_type='av')
        if self.vonly:
            self.trans_v_with_l = self.get_network(self_type='vl')
            self.trans_v_with_a = self.get_network(self_type='va')

        # 3. Self Attentions (Could be replaced by LSTMs, GRUs, etc.)
        #    [e.g., self.trans_x_mem = nn.LSTM(self.d_x, self.d_x, 1)
        self.trans_l_mem = self.get_network(self_type='l_mem', layers=3)
        self.trans_a_mem = self.get_network(self_type='a_mem', layers=3)
        self.trans_v_mem = self.get_network(self_type='v_mem', layers=3)

        # Projection layers
        self.proj1 = nn.Linear(combined_dim, combined_dim)
        self.proj2 = nn.Linear(combined_dim, combined_dim)
        self.out_layer = nn.Linear(combined_dim, output_dim)

    def get_network(self, self_type='l', layers=-1):
        if self_type in ['l', 'al', 'vl']:
            embed_dim, attn_dropout = self.d_l, self.attn_dropout
        elif self_type in ['a', 'la', 'va']:
            embed_dim, attn_dropout = self.d_a, self.attn_dropout_a
        elif self_type in ['v', 'lv', 'av']:
            embed_dim, attn_dropout = self.d_v, self.attn_dropout_v
        elif self_type == 'l_mem':
            embed_dim, attn_dropout = 2*self.d_l, self.attn_dropout
        elif self_type == 'a_mem':
            embed_dim, attn_dropout = 2*self.d_a, self.attn_dropout
        elif self_type == 'v_mem':
            embed_dim, attn_dropout = 2*self.d_v, self.attn_dropout
        else:
            raise ValueError("Unknown network type")

        return TransformerEncoder(embed_dim=embed_dim,
                                  num_heads=self.num_heads,
                                  layers=max(self.layers, layers),
                                  attn_dropout=attn_dropout,
                                  relu_dropout=self.relu_dropout,
                                  res_dropout=self.res_dropout,
                                  embed_dropout=self.embed_dropout,
                                  attn_mask=self.attn_mask)

    def forward(self, x_l, x_a, x_v):
        """
        text, audio, and vision should have dimension [batch_size, seq_len, n_features]
        """
        x_l = F.dropout(x_l.transpose(1, 2), p=self.embed_dropout, training=self.training)
        x_a = x_a.transpose(1, 2)
        x_v = x_v.transpose(1, 2)

        # Project the textual/visual/audio features
        proj_x_l = x_l if self.orig_d_l == self.d_l else self.proj_l(x_l)
        proj_x_a = x_a if self.orig_d_a == self.d_a else self.proj_a(x_a)
        proj_x_v = x_v if self.orig_d_v == self.d_v else self.proj_v(x_v)
        proj_x_a = proj_x_a.permute(2, 0, 1)
        proj_x_v = proj_x_v.permute(2, 0, 1)
        proj_x_l = proj_x_l.permute(2, 0, 1)

        if self.lonly:
            # (V,A) --> L
            h_l_with_as = self.trans_l_with_a(proj_x_l, proj_x_a, proj_x_a)    # Dimension (L, N, d_l)
            h_l_with_vs = self.trans_l_with_v(proj_x_l, proj_x_v, proj_x_v)    # Dimension (L, N, d_l)
            h_ls = torch.cat([h_l_with_as, h_l_with_vs], dim=2)
            h_ls = self.trans_l_mem(h_ls)
            if type(h_ls) == tuple:
                h_ls = h_ls[0]
            last_h_l = last_hs = h_ls[-1]   # Take the last output for prediction

        if self.aonly:
            # (L,V) --> A
            h_a_with_ls = self.trans_a_with_l(proj_x_a, proj_x_l, proj_x_l)
            h_a_with_vs = self.trans_a_with_v(proj_x_a, proj_x_v, proj_x_v)
            h_as = torch.cat([h_a_with_ls, h_a_with_vs], dim=2)
            h_as = self.trans_a_mem(h_as)
            if type(h_as) == tuple:
                h_as = h_as[0]
            last_h_a = last_hs = h_as[-1]

        if self.vonly:
            # (L,A) --> V
            h_v_with_ls = self.trans_v_with_l(proj_x_v, proj_x_l, proj_x_l)
            h_v_with_as = self.trans_v_with_a(proj_x_v, proj_x_a, proj_x_a)
            h_vs = torch.cat([h_v_with_ls, h_v_with_as], dim=2)
            h_vs = self.trans_v_mem(h_vs)
            if type(h_vs) == tuple:
                h_vs = h_vs[0]
            last_h_v = last_hs = h_vs[-1]

        if self.partial_mode == 3:
            last_hs = torch.cat([last_h_l, last_h_a, last_h_v], dim=1)

        # A residual block
        last_hs_proj = self.proj2(F.dropout(F.relu(self.proj1(last_hs)), p=self.out_dropout, training=self.training))
        last_hs_proj += last_hs

        output = self.out_layer(last_hs_proj)
        return output, last_hs
```