Manual Solution of EEG Transformer

Yingqi Ding & Ruyue Hong

May 31, 2019

Contents

1 EncoderDecoder

```
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        return self.generator(self.decode(self.encode(src, src_mask), src_mask,
                                          tgt, tgt_mask))

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

This is a standard architecture for encoder and decoder, which is the base of the model. In addition, specially for the transformer, the inputs of the forward function are masked source and target sequences, then the function will process these inputs. The function of the mask will be described later in the document.

2 Generator

```
class Generator(nn.Module):
    def __init__(self, d_model,output_d):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, output_d)

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

We modify the original code by eliminating some operations specific for the language processing. Here the generator is simply a standard linear one.

3 Clone

```
def clones(module, N):
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

The function that clones N identical layers. The original model uses a stack of six layers.

4 Encoder

```
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

The core encoder is a stack of N layers, which is followed by the normalization. We employ a residual connection around each of the two sub-layers, followed by layer normalization. The forward function passes the inputs through each layer.

5 LayerNorm

```
class LayerNorm(nn.Module):
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

The output of each sub-layer is LayerNorm(x+Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized.

6 SublayerConnection

```
class SublayerConnection(nn.Module):
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

This applies a residual connection to any sublayer with the same size, followed by a layer normalization. For code simplicity, the norm is first as opposed to last.

7 EncoderLayer

```
class EncoderLayer(nn.Module):
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Encoder is made up of self-attention layer and a feed-forward layer, so each encoder layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed- forward network.

8 Decoder

```
class Decoder(nn.Module):
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

Similar to the encoder, the decoder is also composed of a stack of $N = 6$ identical layers. Specially for the transformer, the inputs of the forward function are masked source and target sequences, then the function will process these inputs.

9 DecoderLayer

```python
class DecoderLayer(nn.Module):
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by the layer normalization.

10 attention

```python
def attention(query, key, value, mask=None, dropout=None):
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

The function implements $\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$, which computes the scaled dot product attention.

11 MultiHeadedAttention

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

h is the number of heads. The function is adopted and remain unchanged. The following equations are implemented:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O \text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The projections are parameter matrices where $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

12 PositionwiseFeedForward

```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

This is just a very simple feedforward module, which implements the equation $\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$.

13 Embeddings

```
class Embeddings(nn.Module):
    def __init__(self, d_model, input_d):
        super(Embeddings, self).__init__()
        self.proj = nn.Linear(input_d, d_model)

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

Since EEG itself has been numerical time-dependent sequence, the embedding is unnecessary. So we modify the original embedding to a standard linear layer. We did not delete this function because we want to adopt the architecture of the original code.

14 PositionalEncoding

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0.0, max_len).unsqueeze(1) # [max_len,1]
        div_term = torch.exp(torch.arange(0.0, d_model, 2)
                                 -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + torch.tensor(self.pe[:, :x.size(1)], requires_grad=False)
        return self.dropout(x)
```

Here we implement the positional encoding function and compute the positional encodings once in log space. Unlike the original version, we use torch.tenser() instead of Variable(), as the later is deprecated. Since the transformer contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed.

15 make_model

```
def make_model(src_d,tgt_d,N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn),
                             c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_d),c(position)),
        nn.Sequential(Embeddings(d_model, tgt_d),c(position)),
        Generator(d_model,tgt_d))

    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform(p)
    return model
```

This function specify hyperparameters and construct a model. To use the function, simply call *make_model*() by plugging in dimensions and according parameters.

16 NoamOpt

```
class NoamOpt:
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))
```

This part is very important, since we need to train with this setup of the model. The mathematical formular employed here is $lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$. Please consult the paper or another report for details and numerical parameters.

17 SimpleLossCompute

```
class SimpleLossCompute:
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt

    def __call__(self, x, y,norm):
        loss = self.criterion(x.contiguous(), y.contiguous())
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.item()
```

A simple loss compute and train function. Here we use Adam as the optimizer and the mean squared error as the loss function.

18 Mask

```
def subsequent_mask(size):
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

The subsequent mask is used to mask the future EEGs in a sequence. In other words, the mask indicates which entries should not be used.

This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. This means that to predict the third word, only the first and second word will be used. Similarly to predict the fourth word, only the first, second and the third word will be used and so on.

19 Batch

```
class Batch:
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = torch.ones(src.size(0), 1, src.size(1))
        if trg is not None:
            self.trg = trg[:, :-1,:]
            self.trg_y = trg[:, 1:,:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = self.trg_y.size(1)
    @staticmethod
    def make_std_mask(tgt, pad):
        tgt_mask = torch.ones(tgt.size(0), 1, tgt.size(1),dtype = torch.long)
        tgt_mask = tgt_mask.type_as(tgt_mask.data) &
        subsequent_mask(tgt.size(1)).type_as(tgt_mask.data)
        return tgt_mask
```

Here we use Batch Class to contain a sequence of inputs and targets along with their corresponding masks. How masks are created are described in Mask section. Specially, we have $self.trg\_y$ which is used as 'teacher forcing' in training process.

20 Data Generator

```
def eeg_data_gen(dataloader,start_symbol = 1):
    for idx, (data_x,data_y) in enumerate(dataloader):
        data_x[:,0,:] = start_symbol
        src_ = torch.tensor(data_x.float(), requires_grad=False)
        data_y[:,0, :] = start_symbol
        tgt_ = torch.tensor(data_y.float(), requires_grad=False)
        yield Batch(src_, tgt_, 0)
```

This part is to pre-process datasets, we force the first time-step to be start_symbol which indicates the start of a sequence.

21 training process part1

```
def run_epoch(data_iter, model,loss_compute):
    total_loss = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y,batch.ntokens)
        total_loss += loss
    return total_loss / (i+1)
```

This function specifies the basic operations in the training process. It iterates all the training data and compute the losses given the model and the loss compute function.

22 training process part2

```
model = make_model(opt['src_d'],opt['tgt_d'],
    opt['Transformer-layers'],opt['Model-dimensions'],
    opt['feedford-size'],opt['headers'],opt['dropout'])
model_opt = NoamOpt(model_size=opt['Model-dimensions'],
    factor=1, warmup=400,optimizer =
    torch.optim.Adam(model.parameters(), lr=0.015, betas=(0.9, 0.98), eps=1e-9))

total_epoch = 2000
train_losses=np.zeros(total_epoch)
test_losses=np.zeros(total_epoch)
for epoch in range(total_epoch):
    model.train()
    train_loss = run_epoch(data_gen(16,50,ts_train,ls_train),
        model,SimpleLossCompute(model.generator, criterion, model_opt))
    train_losses[epoch]=train_loss

    if (epoch+1)%100 == 0:
        torch.save({
                    'epoch': epoch,
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': model_opt.optimizer.state_dict(),
                    'loss': loss1,
                    }, 'model_toydata_checkpoint/'+str(epoch)+'.pth')

        torch.save(model, 'model_toydata_save/model_toydata%d.pth'%(epoch))

    model.eval()
    test_loss = run_epoch(data_gen(5,40,ts_test,ls_test),
        model, SimpleLossCompute(model.generator, criterion, None))
    test_losses[epoch]=test_loss
    print('Epoch[{}/{}],train_loss{:.6f},test_loss: {:.6f}'
                  .format(epoch+1, total_epoch,train_loss,test_loss))
```

Firstly, we use the pre-defined model options to create a new model architecture. Then we define hyper-parameters for Noam Optimization which is specifically described in *NoamOpt* section. Then we iterate the training process for total_epoch times, and record the training loss and test loss at each iteration. We also save the trained model weights in a certain frequency.

23 prediction process

```
def output_prediction(model,ts_test,ls_test, max_len, start_symbol,output_d):
    input_data = torch.from_numpy(ts_test).unsqueeze(0).float()
    input_data[:,0, :] = 1
    true_output = torch.from_numpy(ls_test).unsqueeze(0).float()
    true_output[:,0, :] = 1
    src = torch.tensor(input_data, requires_grad=False)
    tgt = torch.tensor(true_output, requires_grad=False)

    test_batch = Batch(src, tgt, 0)
    src = test_batch.src
    src_mask = test_batch.src_mask

    model.eval()
    memory = model.encode(src.float(), src_mask)
    ys = torch.ones(1, 1, output_d).fill_(start_symbol)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask, ys,
            subsequent_mask(ys.size(1)).type_as(src.data))
        out = model.generator(out)
        ys = torch.cat([ys, out[:,[-1],:]], dim=1)
    outs = [out for out in ys]
    return torch.stack(outs,0).detach().numpy(), true_output
```

In this prediction part, we first pre-process the data like in *data generation*, then we feed input to encoder to get memory output which is one of the inputs to decoder. Later, we apply a loop to generate output sequence one by one. This means to generate the fourth output we feed the first three generated output along with memory output from encoder to decoder. Then we return predicted sequence and the true output sequence.

References