Transformer

March 1, 2021

[15]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

To complete the exercise focus on the parts that are mentioned in questions. You don't need to understand everything to its fullest here. Even if it's nice to know. The following file loads the english/german sentence pairs and prints out an example of what a pair looks like. Note that for fast training we cut the sentences down to pairs where the english text starts with a personal pronoun and the correspoding form of "to be".

[16]:
```
%run /content/gdrive/My\ Drive/Colab\ Notebooks/data/load_languages.py
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
Reading lines…
Read 195847 sentence pairs
Trimmed to 11727 sentence pairs
Counting words…
Counted words:
eng 7046
ger 4484
['Das tut mir leid für dich.', 'I am sorry for you.']
```

[17]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import math, copy, time
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[18]:
```
class EncoderDecoder(nn.Module):
    """
```

```
    A standard Encoder-Decoder architecture.
    """
    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):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src), tgt)

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

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

class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

[19]:
```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([module for _ in range(N)])
```

[20]:
```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x):
        "Pass the input through each layer in turn."
        for layer in self.layers:
            x = layer(x)
        return self.norm(x)
```

[21]:
```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features)).to(device)
        self.b_2 = nn.Parameter(torch.zeros(features)).to(device)
        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
```

[22]:
```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

[23]:
```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    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 = [SublayerConnection(size, dropout) for i in range(2)]
        self.size = size

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

[24]:
```
class Decoder(nn.Module):
    "Generic N layer decoder"
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)
```

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

[25]:

```python
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    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):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x
        , mask=subsequent_mask(x.shape[-2])))
        x = self.sublayer[1](x, lambda x: self.src_attn(m, m, x))
        return self.sublayer[2](x, self.feed_forward)
```

Exercise 1a) Complete the attention method

[57]:

```python
def attention(value, key, query, mask = None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    # fill in the gaps
    d_k = key.shape[3]
    # what is inside the softmax
    try:
      scores = query @ torch.transpose(key, 2, 3) / (d_k ** .5)
      #scores = query * key / (d_k ** .5)
    except:
      print(value.shape, key.shape, query.shape, mask)
      raise

    if mask is not None:
        # change all the values where the mask equals 0 to minus infinity (-1e9␣
 ↪is enough)
        try:
          scores = scores.masked_fill((mask == 0).to(device), -1e9)
        except:
          print(scores.shape, mask.shape)
          raise
    # apply softmax to the right dimension
```

```
    p_attn = torch.softmax(scores, dim=3)
    if dropout is not None:
        p_attn = dropout(p_attn)

    # multiply with value
    try:
      attention = value * p_attn.sum(dim=2).unsqueeze(3).repeat(1, 1, 1, value.
↪shape[3])
    except:
      print(value.shape, p_attn.shape, p_attn.sum(dim=2).shape)
      raise

    return attention, p_attn
```

[27]:
```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        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, value, key, query, mask = None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        value, key, query = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (value, key, query))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(value, key, query, mask=mask,
                                 dropout=self.dropout)
        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Exercise 1d) Visualize the mask and explain what it does

[28]:
```python
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

[29]:
```python
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

[30]:
```python
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    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))))


class PositionalEncoding(nn.Module):
    "Implements the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0., max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(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 + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

[31]:
```
def make_model(src_vocab, tgt_vocab, N=6,
               d_model=512, d_ff=2048, h=8, dropout=0.1):
    "Helper: Construct a model from hyperparameters."
    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_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab))
    print(tgt_vocab)
    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform(p)
    return model
```

[32]:
```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]


def tensorFromSentence(lang, sentence):
    # converts a string into a tensor
    # the options for lang are either input_lang or output_lang
    indexes = indexesFromSentence(lang, sentence)
    sen_len = len(indexes)
    # fill the tensor with EOS_tokens at the end, s.t. it has length 10
    indexes.extend([EOS_token for _ in range(10-sen_len)])
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)


def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    input_tensor = input_tensor.permute(1,0)
    target_tensor = target_tensor.permute(1,0)
    return (input_tensor, target_tensor)

def outputTensorFromTGT(tgt):
    # converts the tgt to an output tensor, with which we can compute the loss
    # think about what happens here. 4484 =  output_lang.n_words is the number␣
↪of german words in our dictionary
```

```
    tgt_list = []
    bs = tgt.shape[0]
    for i in range(bs):
        sentdist_tensor = torch.cat([torch.from_numpy(np.eye(1,output_lang.
 ↪n_words,index.item())).unsqueeze(0)
                              for index in tgt[i,:]], dim = 1)
        tgt_list.append(sentdist_tensor)
    return torch.cat(tgt_list, dim = 0).float()




def train(pair, batch_size, sample = False):
    src, tgt = pair
    src.to(device)
    tgt.to(device)
    tgt_tensor = outputTensorFromTGT(tgt).to(device)
    decoder_input = torch.tensor([[SOS_token] for _ in range(batch_size)],␣
 ↪device=device)
    if not sample:
        decoder_input = torch.cat([decoder_input, tgt], dim = 1)
    else:
        cat_list = [decoder_input]
        gen_list = []
    memory = transformer.encode(src)
    if sample:
        for i in range(MAX_LENGTH):
          decoder_output = transformer.decode(memory, decoder_input)
          gen = transformer.generator(decoder_output[:,-1])
          gen_list.append(gen.unsqueeze(1))
          pred = (gen.argmax(dim = 1)).unsqueeze(1)
          cat_list.append(pred)
          decoder_input = torch.cat(cat_list, dim = 1)
        output = torch.cat(gen_list, dim = 1)
    else:
        decoder_output = transformer.decode(memory, decoder_input)
        gen = transformer.generator(decoder_output[:,:10,:])
        output = gen
    loss = criterion(output, tgt_tensor)
    return loss, decoder_input
```

[33]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
transformer = make_model(input_lang.n_words, output_lang.n_words, 2 , d_model =
↪512, d_ff = 512, h = 4)
transformer.to(device)

optimizer = optim.Adam(transformer.parameters())
criterion = nn.KLDivLoss(reduction = 'batchmean')

n_pairs = 64000 # number of training pairs
batch_size = 512

#--------------
# Create Batch
#--------------


training_pairs = []
for i in range(n_pairs // batch_size):
    batch_pairs = [tensorsFromPair(random.choice(pairs)) for _ in
↪range(batch_size)]
    batch_input  = torch.cat([pair[0] for pair in batch_pairs], dim = 0)
    batch_tgt = torch.cat([pair[1] for pair in batch_pairs], dim = 0)
    training_pairs.append((batch_input, batch_tgt))
```

```
4484
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:20: UserWarning:
nn.init.xavier_uniform is now deprecated in favor of nn.init.xavier_uniform_.
```

[34]:

```
def evaluate(pair, showTgt = False, print_ = True):
  transformer.eval()      # disable dropout
  inputs = tensorsFromPair(pair)
  if print_:
      print("Example: Input -> Output:")
      print(pair[0])
      output_sent ='> '
      if showTgt:
            print('- '+pair[1])
  else:
      output_sent = ''
  _ , output = train(inputs, batch_size = 1, sample = True)

  for word in output.squeeze(0):
    if(word.item() == EOS_token):
        # don't print EOS at the end
        break
    if(word.item() != SOS_token):
        # don't print SOS at the beginning
```

```
        output_sent += output_lang.index2word[word.item()]+' '
  return (output_sent)
```

[58]:
```
import time


start_time = time.time()
def timer(progress, iterations):
    now = time.time()-start_time
    now = now*iterations/progress
    std = now // 3600
    min = (now // 60) % 60
    sec = now % 60
    return int(std), int(min), int(sec)


total_loss = 0
log = 25        # when to show training status
epochs = 5      # how many epochs to train
losses = []
for epoch in range(epochs):
    for i,pair in enumerate(training_pairs):
        transformer.train()
        optimizer.zero_grad()
        loss, _ = train(pair, batch_size = batch_size)
        loss.backward()
        optimizer.step()
        total_loss += loss
        losses.append(loss)
        if i% log == log-1:
            print("[Epoch: {}][{}/{}][Loss: {}][Time per epoch: {:02d}:{:02d}:{:
 ↪02d}]".format(
                epoch+1, i+1, int(n_pairs/batch_size), total_loss/log,␣
 ↪*timer(i+1,int(n_pairs/batch_size))))
            total_loss = 0
            pair = random.choice(pairs)
            print(evaluate(pair))
    start_time = time.time()
print('Loss plot:')
plt.plot(losses)
plt.show()
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
RuntimeError                              Traceback (most recent call
↪last)

<ipython-input-58-78663b4b1549> in <module>()
     20         transformer.train()
     21         optimizer.zero_grad()
---> 22         loss, _ = train(pair, batch_size = batch_size)
     23         loss.backward()
     24         optimizer.step()


<ipython-input-32-a2f2b40074c9> in train(pair, batch_size, sample)
     57         output = torch.cat(gen_list, dim = 1)
     58     else:
---> 59         decoder_output = transformer.decode(memory, decoder_input)
     60         gen = transformer.generator(decoder_output[:,:10,:])
     61         output = gen


<ipython-input-18-afa1d14a8edb> in decode(self, memory, tgt)
     19
     20     def decode(self, memory, tgt):
---> 21         return self.decoder(self.tgt_embed(tgt), memory)
     22
     23 class Generator(nn.Module):


/usr/local/lib/python3.7/dist-packages/torch/nn/modules/module.py in
↪_call_impl(self, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),


<ipython-input-24-038c84cbfe3a> in forward(self, x, memory)
      8     def forward(self, x, memory):
      9         for layer in self.layers:
---> 10             x = layer(x, memory)
     11         return self.norm(x)


/usr/local/lib/python3.7/dist-packages/torch/nn/modules/module.py in
↪_call_impl(self, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
```

```
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),


    <ipython-input-25-0bc009261609> in forward(self, x, memory)
     14         x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x
     15         , mask=subsequent_mask(x.shape[-2])))
---> 16         x = self.sublayer[1](x, lambda x: self.src_attn(m, m, x))
     17         return self.sublayer[2](x, self.feed_forward)


    /usr/local/lib/python3.7/dist-packages/torch/nn/modules/module.py in␣
↪_call_impl(self, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),


    <ipython-input-22-b1fea0696ed6> in forward(self, x, sublayer)
     11     def forward(self, x, sublayer):
     12         "Apply residual connection to any sublayer with the same␣
↪size."
---> 13         return x + self.dropout(sublayer(self.norm(x)))


    RuntimeError: The size of tensor a (11) must match the size of tensor b␣
↪(10) at non-singleton dimension 1
```

[ ]:
```
pair = random.choice(pairs)
print(evaluate(pair, showTgt = True))
```

Exercise 1c) Translate your own german sentence. You can make use of the evaluate function to achive that.

[ ]:
```

```

Exercise 2b) Print the distance from your word to sister in embedding space

[ ]:
```
word1 = "woman"
word2 = "man"
word3 = "brother"
word4 = "sister"
```

Exercise 2a) Complete the following code. Be aware that there will be 10 values for each word even if the input sentence is shorter than 10 words. Thats because the sentences get filled with EOS_tokens at the end such that they all are the same size. Set the tick_label of the bar plot to be the input sentences words (use string.split() to convert strings to lists).

[ ]:
```
output = evaluate(pair, showTgt = True, print_ = False)
sentence = pair[0].split()
which_word = 3
print('Output sentence: {}'.format(output))
print(output.split()[which_word],':')
for i in range(4):




plt.show()
```