Thuc hanh tren lop-answer

December 20, 2021

1 Thực hành Transformers

Trong bài này, ta sẽ thực hành cài đặt Transformer

1.0.1 1. Cài đặt và import thư viện

[ ]:
```
!which python3
```

[ ]:
```
!pip3 install spacy dill
!pip3 install torchtext
!pip3 install pandas
```

[ ]:
```
!python3 -m spacy download en && python3 -m spacy download fr
```

[ ]:
```
import torch.nn as nn
import torch
import torchtext
import copy
import math
import torch.nn.functional as F
```

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1.0.2 2. Cài đặt từng module của Transformer

[ ]:
```
class Embedder(nn.Module):
    def __init__(self, vocab_size, dim):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, dim)

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

Position Embedding Class:

[ ]:
```
# Positional encoding
class PositionalEncoder(nn.Module):
    def __init__(self, dim, max_seq_len=300):
        super().__init__()
        self.dim = dim

        # create a constant 'pe' matrix with values dependant on
        # pos and i
        pe = torch.zeros(max_seq_len, dim)
        for pos in range(max_seq_len):
            for i in range(0, dim, 2):
                pe[pos, i] = math.sin(pos/ (10000 ** ((2*i)/dim)))
                pe[pos, i+1] = math.cos(pos / (10000 ** ((2* (i+1))/dim)))

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # make embeddings relatively larger
        x = x *math.sqrt(self.dim)
        # add constant to embedding
        seq_len = x.size(1)
        x = x + Variable(self.pe[:, :seq_len], requires_grad=False).to(device)
        return x
```

Multi Head Attention: We first start with implementing attention function

Attention of q

[ ]:

```
def attention(q, k, v, d_k, mask=None, dropout=None):
    scores = torch.matmul(q, k.transpose(-2, -1)) /  math.sqrt(d_k)
    if mask is not None:
        mask = mask.unsqueeze(1)
        scores = scores.masked_fill(mask == 0, -1e9)

    scores = F.softmax(scores, dim=-1)

    if dropout is not None:
        scores = dropout(scores)

    output = torch.matmul(scores, v)
    return output
```

[ ]:

```
# Multi-headed attention
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, dim, dropout=0.1):
        super().__init__()
        self.dim = dim
        self.dim_head = dim//heads
        self.h = heads
        self.q_linear = nn.Linear(dim, dim)
        self.k_linear = nn.Linear(dim, dim)
        self.v_linear = nn.Linear(dim, dim)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(dim, dim)

    def forward(self, q, k, v, mask=None):
        bs = q.size(0)
        # perform linear operation and split into h heads
        k = self.k_linear(k).view(bs, -1, self.h, self.dim_head)
        q = self.q_linear(q).view(bs, -1, self.h, self.dim_head)
        v = self.v_linear(v).view(bs, -1, self.h, self.dim_head)
        # transpose to get dimensions bs * h * sl * dim
        k = k.transpose(1, 2)
        q = q.transpose(1, 2)
        v = v.transpose(1, 2)
        # calculate attention using the function we will define next
        scores = attention(q, k, v, self.dim, mask, self.dropout)
        # concatenate heads and put through final linear layer
        concat = scores.transpose(1,2).contiguous().view(bs, -1, self.dim)
        output = self.out(concat)
        return output
```

[ ]:

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout = 0.1):
        super().__init__()
        # We set d_ff as a default to 2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)
    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
        return x
```

[ ]:

```
class Norm(nn.Module):
    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model
        # create two learnable parameters to calibrate normalisation
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))
        self.eps = eps
    def forward(self, x):
        norm = self.alpha * (x - x.mean(dim=-1, keepdim=True)) \
        / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
        return norm
```

[ ]:

```python
# build an encoder layer with one multi-head attention layer and one
# feed-forward layer
class EncoderLayer(nn.Module):
    def __init__(self, d_model, heads, dropout = 0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model)
        self.ff = FeedForward(d_model)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x, mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn(x2,x2,x2,mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.ff(x2))
        return x
```

[ ]:

```python
# build a decoder layer with two multi-head attention layers and
# one feed-forward layer
class DecoderLayer(nn.Module):
    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.norm_3 = Norm(d_model)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
        self.dropout_3 = nn.Dropout(dropout)

        self.attn_1 = MultiHeadAttention(heads, d_model)
        self.attn_2 = MultiHeadAttention(heads, d_model)
        self.ff = FeedForward(d_model).cuda()

    def forward(self, x, e_outputs, src_mask, trg_mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn_1(x2, x2, x2, trg_mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.attn_2(x2, e_outputs, e_outputs,
        src_mask))
        x2 = self.norm_3(x)
        x = x + self.dropout_3(self.ff(x2))
        return x
```

```python
def get_clones(module, N):
    return nn.ModuleList([copy.deepcopy(module) for i in range(N)])
```

[ ]:

```python
class Encoder(nn.Module):
    def __init__(self, vocab_size, d_model, N, heads):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model)
        self.layers = get_clones(EncoderLayer(d_model, heads), N)
        self.norm = Norm(d_model)

    def forward(self, src, mask):
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, mask)
        return self.norm(x)

class Decoder(nn.Module):
    def __init__(self, vocab_size, d_model, N, heads):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model)
        self.layers = get_clones(DecoderLayer(d_model, heads), N)
        self.norm = Norm(d_model)
    def forward(self, trg, e_outputs, src_mask, trg_mask):
        x = self.embed(trg)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, e_outputs, src_mask, trg_mask)
        return self.norm(x)
```

[ ]:

```python
class Transformer(nn.Module):
    def __init__(self, src_vocab, trg_vocab, d_model, N, heads):
        super().__init__()
        self.encoder = Encoder(src_vocab, d_model, N, heads)
        self.decoder = Decoder(trg_vocab, d_model, N, heads)
        self.out = nn.Linear(d_model, trg_vocab)
    def forward(self, src, trg, src_mask, trg_mask):
        e_outputs = self.encoder(src, src_mask)
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output# we don't perform softmax on the output as this will be␣
 ↪handled
# automatically by our loss function
```

1.0.3 3. Chuẩn bị và tiền xử lý dữ liệu

[ ]:
```
import spacy
import re

# Tokenize

class tokenize(object):

    def __init__(self, lang):
        self.nlp = spacy.load(lang)

    def tokenizer(self, sentence):
        sentence = re.sub(
        r"[\*\"""\n\\…\+\-\/\=\(\)‘•:\[\]\|’\!;]", " ", str(sentence))
        sentence = re.sub(r"[ ]+", " ", sentence)
        sentence = re.sub(r"\!+", "!", sentence)
        sentence = re.sub(r"\,+", ",", sentence)
        sentence = re.sub(r"\?+", "?", sentence)
        sentence = sentence.lower()
        return [tok.text for tok in self.nlp.tokenizer(sentence) if tok.text !=␣
 ↪" "]
```

[ ]:
```
# Creating batch
from torchtext.legacy import data
import numpy as np
from torch.autograd import Variable


def nopeak_mask(size, opt):
    np_mask = np.triu(np.ones((1, size, size)),
    k=1).astype('uint8')
    np_mask =  Variable(torch.from_numpy(np_mask) == 0)
    np_mask = np_mask.to(device)
    return np_mask

def create_masks(src, trg, opt):

    src_mask = (src != opt.src_pad).unsqueeze(-2)

    if trg is not None:
        trg.to(device)
        trg_mask = (trg != opt.trg_pad).unsqueeze(-2).to(device)
        size = trg.size(1) # get seq_len for matrix
        np_mask = nopeak_mask(size, opt)
        trg_mask = trg_mask & np_mask
    else:
        trg_mask = None
    return src_mask, trg_mask

# patch on Torchtext's batching process that makes it more efficient
# from http://nlp.seas.harvard.edu/2018/04/03/attention.
 ↪html#position-wise-feed-forward-networks

class MyIterator(data.Iterator):
    def create_batches(self):
        if self.train:
            def pool(d, random_shuffler):
                for p in data.batch(d, self.batch_size * 100):
                    p_batch = data.batch(
                        sorted(p, key=self.sort_key),
                        self.batch_size, self.batch_size_fn)
                    for b in random_shuffler(list(p_batch)):
                        yield b
            self.batches = pool(self.data(), self.random_shuffler)

        else:
            self.batches = []
            for b in data.batch(self.data(), self.batch_size,
                                          self.batch_size_fn):
                self.batches.append(sorted(b, key=self.sort_key))

global max_src_in_batch, max_tgt_in_batch

def batch_size_fn(new, count, sofar):
    "Keep augmenting batch and calculate total number of tokens + padding."
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch,  len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch,  len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

[ ]:

```
import pandas as pd
import torchtext
from torchtext.legacy import data
import os
import dill as pickle

def read_data(opt):
    if opt.src_data is not None:
        try:
            opt.src_data = open(opt.src_data).read().strip().split('\n')
        except:
            print("error: '" + opt.src_data + "' file not found")
            quit()

    if opt.trg_data is not None:
        try:
            opt.trg_data = open(opt.trg_data).read().strip().split('\n')
        except:
            print("error: '" + opt.trg_data + "' file not found")
            quit()

def create_fields(opt):
    spacy_langs = ['en', 'fr', 'de', 'es', 'pt', 'it', 'nl']
    src_lang = opt.src_lang[0:2]
    trg_lang = opt.trg_lang[0:2]
    if src_lang not in spacy_langs:
        print('invalid src language: ' + opt.src_lang + 'supported languages : ' + spacy_langs)
    if trg_lang not in spacy_langs:
        print('invalid trg language: ' + opt.trg_lang + 'supported languages : ' + spacy_langs)

    print("loading spacy tokenizers...")

    t_src = tokenize(opt.src_lang)
    t_trg = tokenize(opt.trg_lang)
    TRG = data.Field(lower=True, tokenize=t_trg.tokenizer, init_token='<sos>', eos_token='<eos>')
    SRC = data.Field(lower=True, tokenize=t_src.tokenizer)

    return(SRC, TRG)

def create_dataset(opt, SRC, TRG):

    print("creating dataset and iterator... ")

    raw_data = {'src' : [line for line in opt.src_data], 'trg': [line for line in opt.trg_data]}
    df = pd.DataFrame(raw_data, columns=["src", "trg"])

    mask = (df['src'].str.count(' ') < opt.max_strlen) & (df['trg'].str.count(' ') < opt.max_strlen)
    df = df.loc[mask]
    df.to_csv("translate_transformer_temp.csv", index=False)

    data_fields = [('src', SRC), ('trg', TRG)]
    train = data.TabularDataset('./translate_transformer_temp.csv',␣
 ↪format='csv', fields=data_fields)

    train_iter = MyIterator(train, batch_size=opt.batchsize, device=device,
                        repeat=False, sort_key=lambda x: (len(x.src), len(x.
 ↪trg)),
                        batch_size_fn=batch_size_fn, train=True, shuffle=True)

    os.remove('translate_transformer_temp.csv')
    SRC.build_vocab(train)
    TRG.build_vocab(train)
    opt.src_pad = SRC.vocab.stoi['<pad>']
    opt.trg_pad = TRG.vocab.stoi['<pad>']

    opt.train_len = get_len(train_iter)

    return train_iter

def get_len(train):

    for i, b in enumerate(train):
        pass

    return i
```

1.0.4 4. Cài đặt giải thuật tối ưu và huấn luyện mô hình

[ ]:

```
# Optimizer
class CosineWithRestarts(torch.optim.lr_scheduler._LRScheduler):
    """
    Cosine annealing with restarts.
    Parameters
    ----------
    optimizer : torch.optim.Optimizer
    T_max : int
        The maximum number of iterations within the first cycle.
    eta_min : float, optional (default: 0)
        The minimum learning rate.
    last_epoch : int, optional (default: -1)
        The index of the last epoch.
    """

    def __init__(self,
                 optimizer: torch.optim.Optimizer,
                 T_max: int,
                 eta_min: float = 0.,
                 last_epoch: int = -1,
                 factor: float = 1.) -> None:
        # pylint: disable=invalid-name
        self.T_max = T_max
        self.eta_min = eta_min
        self.factor = factor
        self._last_restart: int = 0
        self._cycle_counter: int = 0
        self._cycle_factor: float = 1.
        self._updated_cycle_len: int = T_max
        self._initialized: bool = False
        super(CosineWithRestarts, self).__init__(optimizer, last_epoch)

    def get_lr(self):
        """Get updated learning rate."""
        # HACK: We need to check if this is the first time get_lr() was called, since
        # we want to start with step = 0, but _LRScheduler calls get_lr with
        # last_epoch + 1 when initialized.
        if not self._initialized:
            self._initialized = True
            return self.base_lrs

        step = self.last_epoch + 1
        self._cycle_counter = step - self._last_restart

        lrs = [
            (
                self.eta_min + ((lr - self.eta_min) / 2) *
                (
                    np.cos(
                        np.pi *
                        ((self._cycle_counter) % self._updated_cycle_len) /
                        self._updated_cycle_len
                    ) + 1
                )
            ) for lr in self.base_lrs
        ]

        if self._cycle_counter % self._updated_cycle_len == 0:
            # Adjust the cycle length.
            self._cycle_factor *= self.factor
            self._cycle_counter = 0
            self._updated_cycle_len = int(self._cycle_factor * self.T_max)
            self._last_restart = step
        return lrs
```

[ ]:

```
!mkdir data
!wget https://raw.githubusercontent.com/SamLynnEvans/Transformer/master/data/
 ↪english.txt
!mv english.txt data
!wget https://raw.githubusercontent.com/SamLynnEvans/Transformer/master/data/
 ↪french.txt data/french.txt
!mv french.txt data
```

[ ]:

```
def get_model(opt, src_vocab, trg_vocab):

    assert opt.d_model % opt.heads == 0
    assert opt.dropout < 1

    model = Transformer(src_vocab, trg_vocab, opt.d_model, opt.n_layers, opt.
 ↪heads)

    if opt.load_weights is not None:
        print("loading pretrained weights...")
        model.load_state_dict(torch.load(f'{opt.load_weights}/model_weights'))
    else:
        for p in model.parameters():
            if p.dim() > 1:
                nn.init.xavier_uniform_(p)

    if opt.device == 0:
        model = model.cuda()

    return model
```

[ ]:

```
import time

def train_model(model, opt):

    print("training model...")
    model.train()
    start = time.time()
    if opt.checkpoint > 0:
        cptime = time.time()

    for epoch in range(opt.epochs):

        total_loss = 0
        print("   %dm: epoch %d [%s]  %d%%  loss = %s" %\
            ((time.time() - start)//60, epoch + 1, "".join(' '*20), 0, '...'),␣
 ↪end='\r')

        if opt.checkpoint > 0:
            torch.save(model.state_dict(), 'weights/model_weights')

        for i, batch in enumerate(opt.train):

            src = batch.src.transpose(0,1).to(device)
            trg = batch.trg.transpose(0,1).to(device)
            trg_input = trg[:, :-1].to(device)
            src_mask, trg_mask = create_masks(src, trg_input, opt)
            preds = model(src, trg_input, src_mask, trg_mask)
            ys = trg[:, 1:].contiguous().view(-1)
            opt.optimizer.zero_grad()
            loss = F.cross_entropy(preds.view(-1, preds.size(-1)), ys,␣
 ↪ignore_index=opt.trg_pad)
            loss.backward()
            opt.optimizer.step()

            total_loss += loss.item()

            if (i + 1) % opt.printevery == 0:
                p = int(100 * (i + 1) / opt.train_len)
                avg_loss = total_loss/opt.printevery
                print("   %dm: epoch %d [%s%s]  %d%%  loss = %.3f" %\
                    ((time.time() - start)//60, epoch + 1, "".join('#'*(p//5)),␣
 ↪"".join(' '*(20-(p//5))), p, avg_loss))
                total_loss = 0

            if opt.checkpoint > 0 and ((time.time()-cptime)//60) // opt.
 ↪checkpoint >= 1:
                torch.save(model.state_dict(), 'weights/model_weights')
                cptime = time.time()


        print("%dm: epoch %d [%s%s]  %d%%  loss = %.3f\nepoch %d complete, loss␣
 ↪= %.03f" %\
        ((time.time() - start)//60, epoch + 1, "".join('#'*(100//5)), "".join('␣
 ↪'*(20-(100//5))), 100, avg_loss, epoch + 1, avg_loss))

class Opt(object):
    pass

def main():
    opt = Opt()
    opt.src_data = "data/english.txt"
    opt.trg_data = "data/french.txt"
    opt.src_lang = "en_core_web_sm"
    opt.trg_lang = 'fr_core_news_sm'
    opt.epochs = 2
    opt.d_model=512
    opt.n_layers=6
    opt.heads=8
    opt.dropout=0.1
    opt.batchsize=1500
    opt.printevery=100
    opt.lr=0.0001
    opt.max_strlen=80
    opt.checkpoint = 0
    opt.no_cuda = False
    opt.load_weights = None

    opt.device = 0
    if opt.device == 0:
        assert torch.cuda.is_available()

    read_data(opt)
    SRC, TRG = create_fields(opt)
    opt.train = create_dataset(opt, SRC, TRG)
    model = get_model(opt, len(SRC.vocab), len(TRG.vocab)).to(device)

    opt.optimizer = torch.optim.Adam(model.parameters(), lr=opt.lr, betas=(0.9,␣
 ↪0.98), eps=1e-9)

    if opt.checkpoint > 0:
        print("model weights will be saved every %d minutes and at end of epoch␣
 ↪to directory weights/"%(opt.checkpoint))

    train_model(model, opt)


    # for asking about further training use while true loop, and return
if __name__ == "__main__":
    main()
```