Báo cáo tuần 8

Người làm: Trần Đức Thọ

Em gửi anh báo cáo em đã làm trong tuần này:

Em đã thử code bộ mã hóa và giải mã trong mạng transformer dựa trên sự tham khảo các nguồn trên mạng:

import torch

import torch.nn as nn

class SelfAttention(nn.Module):

    def \_\_init\_\_(self, embed\_size, heads):

        super(SelfAttention, self).\_\_init\_\_()

        self.embed\_size = embed\_size

        self.heads = heads

        self.head\_dim = embed\_size // heads

        assert (

            self.head\_dim \* heads == embed\_size

        ), "Embedding size needs to be divisible by heads"

        self.values = nn.Linear(self.head\_dim, self.head\_dim, bias=False)

        self.keys = nn.Linear(self.head\_dim, self.head\_dim, bias=False)

        self.queries = nn.Linear(self.head\_dim, self.head\_dim, bias=False)

        self.fc\_out = nn.Linear(heads \* self.head\_dim, embed\_size)

    def forward(self, values, keys, query, mask):

        # Get number of training examples

        N = query.shape[0]

        value\_len, key\_len, query\_len = values.shape[1], keys.shape[1], query.shape[1]

        # Split the embedding into self.heads different pieces

        values = values.reshape(N, value\_len, self.heads, self.head\_dim)

        keys = keys.reshape(N, key\_len, self.heads, self.head\_dim)

        query = query.reshape(N, query\_len, self.heads, self.head\_dim)

        values = self.values(values)  # (N, value\_len, heads, head\_dim)

        keys = self.keys(keys)  # (N, key\_len, heads, head\_dim)

        queries = self.queries(query)  # (N, query\_len, heads, heads\_dim)

        # Einsum does matrix mult. for query\*keys for each training example

        # with every other training example, don't be confused by einsum

        # it's just how I like doing matrix multiplication & bmm

        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # queries shape: (N, query\_len, heads, heads\_dim),

        # keys shape: (N, key\_len, heads, heads\_dim)

        # energy: (N, heads, query\_len, key\_len)

        # Mask padded indices so their weights become 0

        if mask is not None:

            energy = energy.masked\_fill(mask == 0, float("-1e20"))

        # Normalize energy values similarly to seq2seq + attention

        # so that they sum to 1. Also divide by scaling factor for

        # better stability

        attention = torch.softmax(energy / (self.embed\_size \*\* (1 / 2)), dim=3)

        # attention shape: (N, heads, query\_len, key\_len)

        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(

            N, query\_len, self.heads \* self.head\_dim

        )

        # attention shape: (N, heads, query\_len, key\_len)

        # values shape: (N, value\_len, heads, heads\_dim)

        # out after matrix multiply: (N, query\_len, heads, head\_dim), then

        # we reshape and flatten the last two dimensions.

        out = self.fc\_out(out)

        # Linear layer doesn't modify the shape, final shape will be

        # (N, query\_len, embed\_size)

        return out

class TransformerBlock(nn.Module):

    def \_\_init\_\_(self, embed\_size, heads, dropout, forward\_expansion):

        super(TransformerBlock, self).\_\_init\_\_()

        self.attention = SelfAttention(embed\_size, heads)

        self.norm1 = nn.LayerNorm(embed\_size)

        self.norm2 = nn.LayerNorm(embed\_size)

        self.feed\_forward = nn.Sequential(

            nn.Linear(embed\_size, forward\_expansion \* embed\_size),

            nn.ReLU(),

            nn.Linear(forward\_expansion \* embed\_size, embed\_size),

        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):

        attention = self.attention(value, key, query, mask)

        # Add skip connection, run through normalization and finally dropout

        x = self.dropout(self.norm1(attention + query))

        forward = self.feed\_forward(x)

        out = self.dropout(self.norm2(forward + x))

        return out

class Encoder(nn.Module):

    def \_\_init\_\_(

        self,

        src\_vocab\_size,

        embed\_size,

        num\_layers,

        heads,

        device,

        forward\_expansion,

        dropout,

        max\_length,

    ):

        super(Encoder, self).\_\_init\_\_()

        self.embed\_size = embed\_size

        self.device = device

        self.word\_embedding = nn.Embedding(src\_vocab\_size, embed\_size)

        self.position\_embedding = nn.Embedding(max\_length, embed\_size)

        self.layers = nn.ModuleList(

            [

                TransformerBlock(

                    embed\_size,

                    heads,

                    dropout=dropout,

                    forward\_expansion=forward\_expansion,

                )

                for \_ in range(num\_layers)

            ]

        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):

        N, seq\_length = x.shape

        positions = torch.arange(0, seq\_length).expand(N, seq\_length).to(self.device)

        out = self.dropout(

            (self.word\_embedding(x) + self.position\_embedding(positions))

        )

        # In the Encoder the query, key, value are all the same, it's in the

        # decoder this will change. This might look a bit odd in this case.

        for layer in self.layers:

            out = layer(out, out, out, mask)

        return out

class DecoderBlock(nn.Module):

    def \_\_init\_\_(self, embed\_size, heads, forward\_expansion, dropout, device):

        super(DecoderBlock, self).\_\_init\_\_()

        self.norm = nn.LayerNorm(embed\_size)

        self.attention = SelfAttention(embed\_size, heads=heads)

        self.transformer\_block = TransformerBlock(

            embed\_size, heads, dropout, forward\_expansion

        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src\_mask, trg\_mask):

        attention = self.attention(x, x, x, trg\_mask)

        query = self.dropout(self.norm(attention + x))

        out = self.transformer\_block(value, key, query, src\_mask)

        return out

class Decoder(nn.Module):

    def \_\_init\_\_(

        self,

        trg\_vocab\_size,

        embed\_size,

        num\_layers,

        heads,

        forward\_expansion,

        dropout,

        device,

        max\_length,

    ):

        super(Decoder, self).\_\_init\_\_()

        self.device = device

        self.word\_embedding = nn.Embedding(trg\_vocab\_size, embed\_size)

        self.position\_embedding = nn.Embedding(max\_length, embed\_size)

        self.layers = nn.ModuleList(

            [

                DecoderBlock(embed\_size, heads, forward\_expansion, dropout, device)

                for \_ in range(num\_layers)

            ]

        )

        self.fc\_out = nn.Linear(embed\_size, trg\_vocab\_size)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc\_out, src\_mask, trg\_mask):

        N, seq\_length = x.shape

        positions = torch.arange(0, seq\_length).expand(N, seq\_length).to(self.device)

        x = self.dropout((self.word\_embedding(x) + self.position\_embedding(positions)))

        for layer in self.layers:

            x = layer(x, enc\_out, enc\_out, src\_mask, trg\_mask)

        out = self.fc\_out(x)

        return out

class Transformer(nn.Module):

    def \_\_init\_\_(

        self,

        src\_vocab\_size,

        trg\_vocab\_size,

        src\_pad\_idx,

        trg\_pad\_idx,

        embed\_size=512,

        num\_layers=6,

        forward\_expansion=4,

        heads=8,

        dropout=0,

        device="cpu",

        max\_length=100,

    ):

        super(Transformer, self).\_\_init\_\_()

        self.encoder = Encoder(

            src\_vocab\_size,

            embed\_size,

            num\_layers,

            heads,

            device,

            forward\_expansion,

            dropout,

            max\_length,

        )

        self.decoder = Decoder(

            trg\_vocab\_size,

            embed\_size,

            num\_layers,

            heads,

            forward\_expansion,

            dropout,

            device,

            max\_length,

        )

        self.src\_pad\_idx = src\_pad\_idx

        self.trg\_pad\_idx = trg\_pad\_idx

        self.device = device

    def make\_src\_mask(self, src):

        src\_mask = (src != self.src\_pad\_idx).unsqueeze(1).unsqueeze(2)

        # (N, 1, 1, src\_len)

        return src\_mask.to(self.device)

    def make\_trg\_mask(self, trg):

        N, trg\_len = trg.shape

        trg\_mask = torch.tril(torch.ones((trg\_len, trg\_len))).expand(

            N, 1, trg\_len, trg\_len

        )

        return trg\_mask.to(self.device)

    def forward(self, src, trg):

        src\_mask = self.make\_src\_mask(src)

        trg\_mask = self.make\_trg\_mask(trg)

        enc\_src = self.encoder(src, src\_mask)

        out = self.decoder(trg, enc\_src, src\_mask, trg\_mask)

        return out

if \_\_name\_\_ == "\_\_main\_\_":

    device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

    print(device)

    x = torch.tensor([[1, 5, 6, 4, 3, 9, 5, 2, 0], [1, 8, 7, 3, 4, 5, 6, 7, 2]]).to(

        device

    )

    trg = torch.tensor([[1, 7, 4, 3, 5, 9, 2, 0], [1, 5, 6, 2, 4, 7, 6, 2]]).to(device)

    src\_pad\_idx = 0

    trg\_pad\_idx = 0

    src\_vocab\_size = 10

    trg\_vocab\_size = 10

    model = Transformer(src\_vocab\_size, trg\_vocab\_size, src\_pad\_idx, trg\_pad\_idx, device=device).to(

        device

    )

    out = model(x, trg[:, :-1])

    print(out.shape)

em cũng đã thử dùng model dịch máy của api Helsinki-NLP

https://colab.research.google.com/drive/1-lmlzozM17-R-KLfflQtldNKE4rTZtm\_?usp=sharing