transformer_for_machine_translation

July 15, 2024

[ ]:

```
import math

import torch
import torch.nn as nn
from torch.utils.data import dataset
import torch.nn.functional as F

import numpy as np
import matplotlib.pyplot as plt
```

[ ]:

```
class MultiHeadAttention(nn.Module):
  def __init__(self, d_k, d_model, n_heads, max_len, causal = False):
    super().__init__()

    self.d_k = d_k
    self.n_heads = n_heads

    self.key = nn.Linear(d_model, d_k*n_heads)
    self.query = nn.Linear(d_model, d_k*n_heads)
    self.value = nn.Linear(d_model, d_k*n_heads)

    self.fc = nn.Linear(d_k*n_heads, d_model)

    self.causal = causal
    if causal:
      cm = torch.tril(torch.ones(max_len, max_len))
      self.register_buffer(
        "causal_mask",
        cm.view(1,1,max_len,max_len)
    )

  def forward(self, q, k,v,pad_mask = None):
    q = self.query(q)
    k = self.key(k)
    v = self.value(v)

    N = q.shape[0]
    T_output = q.shape[1]
    T_input = k.shape[1]

    q = q.view(N,T_output,self.n_heads, self.d_k).transpose(1,2)
    k = k.view(N,T_input,self.n_heads, self.d_k).transpose(1,2)
    v = v.view(N,T_input,self.n_heads, self.d_k).transpose(1,2)

    attn_scores = q@k.transpose(-2,-1)/math.sqrt(self.d_k)
    if pad_mask is not None:
      attn_scores = attn_scores.masked_fill(
          pad_mask[:,None,None, :] == 0, float('-inf')
      )
    if self.causal:
      attn_scores = attn_scores.masked_fill(
        self.causal_mask[:,:,:T_output,:T_input] == 0, float('-inf')
    )
    attn_weights = F.softmax(attn_scores, dim = -1)

    A = attn_weights@v

    A = A.transpose(1,2)
    A = A.contiguous().view(N, T_output, self.d_k * self.n_heads)

    return self.fc(A)
```

[ ]:

```
class EncoderBlock(nn.Module):
  def __init__(self, d_k,d_model, n_heads, max_len, dropout_prob = 0.1):
    super().__init__()

    self.ln1 = nn.LayerNorm(d_model)
    self.ln2 = nn.LayerNorm(d_model)
    self.mha = MultiHeadAttention(d_k, d_model, n_heads, max_len, causal = False)
    self.ann = nn.Sequential(
        nn.Linear(d_model, d_model *4),
        nn.GELU(),
        nn.Linear(d_model*4, d_model),
        nn.Dropout(dropout_prob)
    )
    self.dropout = nn.Dropout(p=dropout_prob)

  def forward(self, x, mask = None):
    x = self.ln1(x+self.mha(x,x,x,mask))
    x = self.ln2(x+self.ann(x))
    x = self.dropout(x)
    return x
```

[ ]:

```
class DecoderBlock(nn.Module):
  def __init__(self, d_k,d_model, n_heads,max_len, dropout_prob = 0.1):
    super().__init__()

    self.ln1 = nn.LayerNorm(d_model)
    self.ln2 = nn.LayerNorm(d_model)
    self.ln3 = nn.LayerNorm(d_model)
    self.mha1 = MultiHeadAttention(d_k, d_model, n_heads, max_len, causal = True)
    self.mha2 = MultiHeadAttention(d_k, d_model, n_heads, max_len, causal =␣
↪False)
    self.ann = nn.Sequential(
        nn.Linear(d_model, d_model *4),
        nn.GELU(),
        nn.Linear(d_model*4, d_model),
        nn.Dropout(dropout_prob)
    )
    self.dropout = nn.Dropout(p=dropout_prob)

  def forward(self, enc_output, dec_input, enc_mask = None, dec_mask = None):
    x = self.ln1(dec_input+self.mha1(dec_input,dec_input,dec_input,dec_mask))
    x = self.ln2(x+self.mha2(x,enc_output,enc_output,enc_mask))
    x = self.ln3(x+self.ann(x))
    x = self.dropout(x)
    return x
```

[ ]:

```
class PositionalEncoding(nn.Module):
  def __init__(self, d_model, max_len = 2048, dropout_prob = 0.1):
    super().__init__()

    self.dropout = nn.Dropout(p=dropout_prob)

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

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

[ ]:

```
class Encoder(nn.Module):
  def __init__(self,
               vocab_size,
               max_len,
               d_k,
               d_model,
               n_heads,
               n_layers,

               dropout_prob):
    super().__init__()
    self.embedding = nn.Embedding(vocab_size, d_model)
    self.pos_encoding = PositionalEncoding(d_model, max_len, dropout_prob)
    transformer_blocks = [
        EncoderBlock(
            d_k,
            d_model,
            n_heads,
            dropout_prob) for _ in range(n_layers)]
    self.transfomer_blocks = nn.Sequential(*transformer_blocks)
    self.ln = nn.LayerNorm(d_model)

  def forward(self, x, pad_mask = None):
    x = self.embedding(x)
    x = self.pos_encoding(x)
    for block in self.transfomer_blocks:
      x = block(x,pad_mask)


    x = self.ln(x)


    return x
```

[ ]:

```
class Decoder(nn.Module):
  def __init__(self,
               vocab_size,
               max_len,
               d_k,
               d_model,
               n_heads,
               n_layers,
               dropout_prob):
    super().__init__()
    self.embedding = nn.Embedding(vocab_size, d_model)
    self.pos_encoding = PositionalEncoding(d_model, max_len, dropout_prob)
    transformer_blocks = [
        DecoderBlock(
            d_k,
            d_model,
            n_heads,
            max_len,
            dropout_prob) for _ in range(n_layers)]
    self.transfomer_blocks = nn.Sequential(*transformer_blocks)
    self.ln = nn.LayerNorm(d_model)
    self.fc = nn.Linear(d_model, vocab_size)

  def forward(self, enc_output, dec_input, enc_mask = None, dec_mask = None):
    x = self.embedding(dec_input)
    x = self.pos_encoding(x)
    for block in self.transfomer_blocks:
      x = block(enc_output,x,enc_mask, dec_mask)

    x = self.ln(x)
    x = self.fc(x)

    return x
```

[ ]:

```
class Transformer(nn.Module):
  def __init__(self,encoder,decoder):
    super().__init__()
    self.encoder = encoder
    self.decoder = decoder

  def forward(self, enc_input, dec_input, enc_mask, dec_mask):
    enc_output = self.encoder(enc_input, enc_mask)
    dec_output = self.decoder(enc_output, dec_input, enc_mask, dec_mask)
    return dec_output
```

[ ]:

```
#test_it
encoder = Encoder(
    vocab_size = 20_000,
    max_len = 512,
    d_k = 16,
    d_model = 64,
    n_heads = 4,
    n_layers = 2,
    dropout_prob = 0.1
)
decoder = Decoder(
    vocab_size = 10_000,
    max_len = 512,
    d_k = 16,
    d_model = 64,
    n_heads = 4,
    n_layers = 2,
    dropout_prob = 0.1
)
transformer = Transformer(encoder, decoder)
```

[ ]:

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

[ ]:

```
xe = np.random.randint(0,20_000, size = (8,512))
xe_t = torch.tensor(xe).to(device)

xd = np.random.randint(0,10_000, size = (8,256))
xd_t = torch.tensor(xd).to(device)

maske = np.ones((8,512))
maske[:,256:] = 0
maske_t = torch.tensor(maske).to(device)

maskd = np.ones((8,256))
maskd[:,128:] = 0
maskd_t = torch.tensor(maskd).to(device)

out = transformer(xe_t,xd_t,maske_t,maskd_t)
out.shape
```

[ ]:

```
torch.Size([8, 256, 10000])
```

[ ]:

```
!head spa.txt
```

[ ]:

```
import pandas as pd
df = pd.read_csv('spa.txt', sep = "\t", header = None)
df.head()
```

[ ]:

```
df.shape
```

[ ]:

```
(115245, 2)
```

[ ]:

```
df = df.iloc[:30_000]
```

[ ]:

```
df.columns = ['en','es']
df.to_csv('spa.csv', index = None)
```

[ ]:

```
!head spa.csv
```

[ ]:

```
!pip install transformers datasets sentencepiece sacremoses
```

[ ]:

```
from datasets import load_dataset
raw_dataset = load_dataset('csv', data_files = 'spa.csv')
```

[ ]:

```
raw_dataset
```

[ ]:

```
split = raw_dataset['train'].train_test_split(test_size = 0.3, seed = 42)
split
```

[ ]:

```
from transformers import AutoTokenizer

model_checkpoint = "Helsinki-NLP/opus-mt-en-es"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
```

[ ]:

```
en_sentence = split['train'][0]['en']
es_sentence = split['train'][0]['es']

inputs = tokenizer(en_sentence)
targets = tokenizer(text_target = es_sentence)

tokenizer.convert_ids_to_tokens(targets['input_ids'])
```

[ ]:

```
es_sentence
```

[ ]:

```
max_input_length = 128
max_target_length = 128

def preprocess_function(batch):
  model_inputs = tokenizer(
      batch['en'], max_length = max_input_length, truncation = True
  )

  labels = tokenizer(
      text_target = batch['es'],max_length = max_target_length, truncation = True
  )

  model_inputs["labels"] = labels["input_ids"]
  return model_inputs
```

[ ]:

```
tokenized_datasets = split.map(
    preprocess_function,
    batched = True,
    remove_columns = split['train'].column_names

)
```

[ ]:

```
tokenized_datasets
```

[ ]:

```
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(tokenizer)
```

[ ]:
```
batch = data_collator([tokenized_datasets["train"][i] for i in range(0,5)])
batch.keys()
```

[ ]:
```
batch['input_ids']
```

[ ]:
```
batch['attention_mask']
```

[ ]:
```
batch['labels']
```

[ ]:
```
tokenizer.all_special_ids
```

[ ]:
```
[0, 1, 65000]
```

[ ]:
```
tokenizer.all_special_tokens
```

[ ]:
```
tokenizer('<pad>')
```

[ ]:
```
from torch.utils.data import DataLoader

train_loader = DataLoader(
    tokenized_datasets["train"],
    shuffle = True,
    batch_size = 32,
    collate_fn = data_collator
)

valid_loader = DataLoader(
    tokenized_datasets["test"],
    batch_size = 32,
    collate_fn = data_collator
)
```

[ ]:
```
for batch in train_loader:
  for k, v in batch.items():
    print("k:", k, "v.shape", v.shape)
  break
```

[ ]:
```
tokenizer.vocab_size
```

[ ]:
```
tokenizer.decode([60000])
```

[ ]:
```
tokenizer.add_special_tokens({"cls_token":"<s>"})
```

[ ]:
```
tokenizer("<s>")
```

[ ]:
```
tokenizer.vocab_size
```

[ ]:

```
encoder = Encoder(vocab_size = tokenizer.vocab_size +1,
                  max_len = 512,
                  d_k = 16,
                  d_model = 64,
                  n_heads = 4,
                  n_layers = 2,
                  dropout_prob = 0.1)
decoder = Decoder(vocab_size = tokenizer.vocab_size +1,
                  max_len = 512,
                  d_k = 16,
                  d_model = 64,
                  n_heads = 4,
                  n_layers = 2,
                  dropout_prob = 0.1)

tranformer = Transformer(encoder, decoder)
```

[ ]:

```
encoder.to(device)
decoder.to(device)
```

[ ]:

```
criterion = nn.CrossEntropyLoss(ignore_index = -100)
optimizer = torch.optim.Adam(transformer.parameters())
```

[ ]:

```
from datetime import datetime
def train(model, criterion, optimizer, train_loader, valid_loader, epochs):
  train_losses = np.zeros(epochs)
  test_losses = np.zeros(epochs)

  for it in range(epochs):
    model.train()
    t0 = datetime.now()
    train_loss = []

    for batch in train_loader:
      batch = {k:v.to(device) for k,v in batch.items()}

      optimizer.zero_grad()

      enc_input = batch['input_ids']
      enc_mask = batch['attention_mask']
      targets = batch['labels']

      dec_input = targets.clone().detach()
      dec_input = torch.roll(dec_input, shifts =1, dims =1)
      dec_input[:,0] = 65_001

      dec_input = dec_input.masked_fill(
          dec_input == -100, tokenizer.pad_token_id
      )

      dec_mask = torch.ones_like(dec_input)
      dec_mask = dec_mask.masked_fill(dec_input == tokenizer.pad_token_id, 0)

      outputs = model(enc_input, dec_input, enc_mask, dec_mask)
      loss = criterion(outputs.transpose(2,1), targets)

      loss.backward()
      optimizer.step()

      train_loss.append(loss.item())


    train_loss = np.mean(train_loss)
    model.eval()
    test_loss = []

    for batch in valid_loader:
      batch = {k:v.to(device) for k, v in batch.items()}

      enc_input = batch['input_ids']
      enc_mask = batch['attention_mask']
      targets = batch['labels']

      dec_input = targets.clone().detach()
      dec_input = torch.roll(dec_input, shifts =1, dims =1)
      dec_input[:,0] = 65_001

      dec_input = dec_input.masked_fill(
          dec_input == -100, tokenizer.pad_token_id
      )

      dec_mask = torch.ones_like(dec_input)
      dec_mask = dec_mask.masked_fill(dec_input == tokenizer.pad_token_id, 0)

      outputs = model(enc_input, dec_input, enc_mask, dec_mask)
      loss = criterion(outputs.transpose(2,1), targets)

      test_loss.append(loss.item())

    test_loss = np.mean(test_loss)

    train_losses[it] = train_loss
    test_losses[it] = test_loss
```

```
    dt = datetime.now() - t0
    print(f'Epoch {it+1}/{epochs}, Train Loss: {train_loss:.4f}, Test Loss:␣
 ↪{test_loss: .4f}, Duration:{dt}')
  return train_losses, test_losses
```

[ ]:

```
train_losses, test_losses = train(
    transformer, criterion, optimizer, train_loader, valid_loader, epochs = 15
)
```

[ ]:

```
input_sentence = split['test'][10]['en']
input_sentence
```

[ ]:

```
enc_input = tokenizer(input_sentence, return_tensors = 'pt')
enc_input
```

[ ]:

```
dec_input_str = '<s>'

dec_input = tokenizer(text_target = dec_input_str, return_tensors = 'pt')
dec_input
```

[ ]:

```
enc_input.to(device)
dec_input.to(device)
output = transformer(
    enc_input['input_ids'],
    dec_input['input_ids'][:,:-1],
    enc_input['attention_mask'],
    dec_input['attention_mask'][:,:-1]
)
output
```

[ ]:

```
outptu.shape
```

[ ]:

```
enc_output = encoder(enc_input['input_ids'], enc_input['attention_mask'])
enc_output.shape
```

[ ]:

```
dec_output = decoder(
    enc_output,
    dec_input['input_ids'][:,:-1],
    enc_input['attention_mask'],
    dec_input['attention_mask'][:,:-1]
)

dec_output.shape
```

[ ]:

```
torch.allclose(output, dec_output)
```

[ ]:

```
dec_input_ids = dec_input['input_ids'][:,:-1]
dec_attn_mask = dec_input['attention_mask'][:,:-1]

for _ in range(32):
  dec_output = decoder(
      enc_output,
      dec_input_ids,
      enc_input['attention_mask'],
      dec_attn_mask
  )

  prediction_id = (torch.argmax(dec_output[:,:-1,_], axis = -1))

  dec_input_ids = torch.hstack((dec_input_ids, prediction_id.view(1,1)))

  dec_attn_mask = torch.ones_like(dec_input_ids)

  if prediction_id == 0:
    break
```

[ ]:

```
tokenizer.decode(dec_input_ids[0])
```

[ ]:

```
split['test'][10]['es']
```

[ ]:

```
def translate(input_sentence):
  enc_input = tokenizer(input_sentence, return_tensors = 'pt').to(device)
  enc_output = encoder(enc_input['input_ids'], enc_input['attention_mask'])

  dec_input_ids = torch.tensor([[65_001]], device = device)
  dec_attn_mask = torch.ones_like(dec_input_ids, device = device)

  for _ in range(32):
    dec_output = decoder(
        enc_output,
        dec_input_ids,
        enc_input['attention_mask'],
        dec_attn_mask
    )

    prediction_id = (torch.argmax(dec_output[:,-1,_], axis = -1))

    dec_input_ids = torch.hstack((dec_input_ids, prediction_id.view(1,1)))

    dec_attn_mask = torch.ones_like(dec_input_ids)

    if prediction_id == 0:
      break
  translation = tokenizer.decode(dec_input_ids[0,1:])
  print(translation)
```

```
[ ]: translate("How are you?")
```