8^{η} εργασία Deep Learning

Χρήση transformer για μετάφραση από αγγλικά στη γλώσσα των ξωτικών από το Lord of the rings (Sindarin).

Sources:

https://www.elfdict.com

https://www.tecendil.com

Reddit r/lotr r/Sindarin

Κώδικας:

```
1.
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import time
import pandas as pd
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
from elvish_dict import elvish
from collections import Counter
from torch.nn.utils.rnn import pad_sequence
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)
```

```python
    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self.W_o(self.combine_heads(attn_output))
        return output

class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

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

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

class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x

class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
```

```
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x

class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff,
max_seq_length, dropout):
        super(Transformer, self).__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)
        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff, dropout) for _
in range(num_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _
in range(num_layers)])

        self.fc = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def generate_mask(self, src, tgt):
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
        tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)
        seq_length = tgt.size(1)
        nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length),
diagonal=1)).bool()#.to(self.device)
        tgt_mask = tgt_mask & nopeak_mask
        return src_mask, tgt_mask

    def forward(self, src, tgt):
        src_mask, tgt_mask = self.generate_mask(src, tgt)
        src_embedded = self.dropout(self.positional_encoding(self.encoder_embedding(src)))
        tgt_embedded = self.dropout(self.positional_encoding(self.decoder_embedding(tgt)))

        enc_output = src_embedded
        for enc_layer in self.encoder_layers:
            enc_output = enc_layer(enc_output, src_mask)

        dec_output = tgt_embedded
        for dec_layer in self.decoder_layers:
            dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

        output = self.fc(dec_output)
        return output

class CustomDataset(Dataset):

    def __init__(self,data,src_vocab,trg_vocab,transform = None):
        self.src_vocab = src_vocab
        self.trg_vocab = trg_vocab
        self.translator = data
        self.transform = transform
        self.english = data['en']
        self.elvish = data['elf']

    def __len__(self):
        return len(self.translator)

    def __getitem__(self, index):
```

```
        en = self.translator['en'].iloc[index]
        en_caption = []
        elf = self.translator['elf'].iloc[index]
        elf_caption = []
        en_caption.append(self.src_vocab["<sos>"])
        en_caption += numericalize(en,self.src_vocab)
        en_caption.append(self.src_vocab["<eos>"])
        elf_caption.append(self.trg_vocab["<sos>"])
        elf_caption += numericalize(elf,self.trg_vocab)
        elf_caption.append(self.trg_vocab["<eos>"])

        return torch.tensor(en_caption, dtype=torch.long),torch.tensor(elf_caption,
dtype=torch.long)

#Encode text based on this vocabulary instance
def numericalize(text,vocab):
    tokens = tokenize(text)
    numericalized = []
    for token in tokens:
        if token in vocab:
            numericalized.append(vocab[token])
        else:
            numericalized.append(vocab["<unk>"])
    return numericalized

def tokenize(text):
    return text.lower().split()

def build_vocab(sentences, min_freq=1):
    counter = Counter()
    for sentence in sentences:
        tokens = tokenize(sentence)
        counter.update(tokens)
    vocab = {"<pad>": 0, "<sos>": 1, "<eos>": 2, "<unk>": 3}
    index = 4
    for token, freq in counter.items():
        if freq >= min_freq:
            vocab[token] = index
            index += 1
    return vocab

def pad_batch(batch,pad_token_src=0,pad_token_trg=0):

    src_seqs, tgt_seqs = zip(*batch)

    # Pad the sequences
    padded_src = pad_sequence(src_seqs, batch_first=True, padding_value=pad_token_src)
    padded_trg = pad_sequence(tgt_seqs, batch_first=True, padding_value=pad_token_trg)

    return padded_src, padded_trg

def translate(model, src_sentence, english_vocab, elf_vocab, max_length=50):
    model.eval()
    src_encoded = numericalize(src_sentence, english_vocab)
    src_tensor = torch.tensor(src_encoded).unsqueeze(0)

    generated = [english_vocab["<sos>"]]

    for _ in range(max_length):
        tgt_tensor = torch.tensor([generated])
        with torch.no_grad():
            output = model(src_tensor, tgt_tensor)

        logits = output[0, -1]
        logits[english_vocab["<unk>"]] = -float('inf')  # Block UNK.
```

```
        logits[english_vocab["<pad>"]] = -float('inf')  # Block PAD.

        probs = torch.softmax(logits / 0.7, dim=-1)
        next_token = torch.multinomial(probs, num_samples=1).item()

        generated.append(next_token)
        if next_token == english_vocab["<eos>"]:
            break

    # Converting to text and filtering special tokens.
    return ' '.join(elf_vocab[token] for token in generated if token not in
[english_vocab["<unk>"],english_vocab["<sos>"],english_vocab["<eos>"],english_vocab["<pad>"]])

df = pd.DataFrame(elvish.keys(),columns=['en'])
df['elf'] = pd.Series(elvish.values())

elf_vocab = build_vocab(df['elf'])
en_vocab = build_vocab(df['en'])
elf_ivocab = {v:k for k,v in elf_vocab.items()}

data = CustomDataset(df,en_vocab,elf_vocab)

data_train,data_test = torch.utils.data.random_split(data,[0.8,0.2])

TRAIN = False
BATCH_SIZE = 16
src_vocab_size = len(en_vocab)
tgt_vocab_size = len(elf_vocab)
d_model = 512
num_heads = 4
num_layers = 2
d_ff = 2048
max_seq_length = 100
dropout = 0.5

train_loader = DataLoader(
  data_train,
  batch_size=BATCH_SIZE,
  shuffle=True,
  collate_fn=pad_batch,
  drop_last=False,
  num_workers=0
)
test_loader = DataLoader(
  data_test,
  batch_size=BATCH_SIZE,
  shuffle=False,
  collate_fn=pad_batch,
  drop_last=False,
  num_workers=0
)

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff,
max_seq_length, dropout)

criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

transformer.train()

if TRAIN:
    for epoch in range(100):
        progress_bar = tqdm(train_loader,desc = f"Epoch {epoch+1}", unit="batch")
        for en,elf in progress_bar:
            #en,elf = en.to(device),elf.to(device)
```

```
            optimizer.zero_grad()
            output = transformer(en, elf[:, :-1])
            loss = criterion(output.contiguous().view(-1, tgt_vocab_size), elf[:,
1:].contiguous().view(-1))
            loss.backward()
            optimizer.step()
            print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
    torch.save(transformer.state_dict(), "transformer.pth")
else:
    transformer.load_state_dict(torch.load("transformer.pth",weights_only=True))
    sentence = "fight for your life."
    print("English:",sentence)
    print('Elvish:',translate(transformer,sentence,en_vocab,elf_ivocab))
```

Αποτελέσματα:

```
English: My life has meaning.
Elvish: cuil nín.
```

Cuil nin = my life

```
English: I eat meat.
Elvish: madon basgorn a aegor.
```

Eat meat and cooked

```
English: hello
Elvish: mae govannen
```

Hello/well met

```
English: war brings freedom.
Elvish: hîr leithian.
```

Captain freedom

```
English: Elves sleep.
Elvish: lôrol nu tîr hên
```

~I sleep watching student

```
English: fight for your life.
Elvish: dagrol an cuil lín lín.
```

Fight for your life (kind of accurate)

```
English: heal my wounds.
Elvish: nestol hon dor nín. nín. nín. nín. nín.
```

Heal the land me. Me. Me. Me. Me. Me.

```
English: he smells the future.
Elvish: glîrol hon athra.
```

He smells straight/across/future

```
English: Death came to his children.
Elvish: beriol an nîdh laer.
```

Protect a sad/mournful poet/song. (what?)