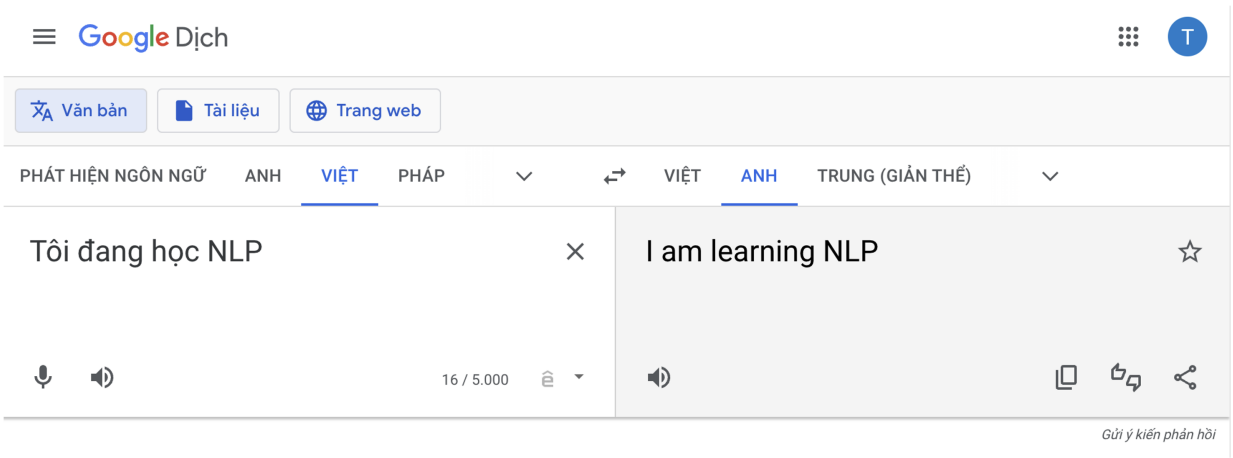
AI VIET NAM – AIO COURSE 2023

Project: Neural Machine Translation

Quoc-Thai Nguyen và Quang-Vinh Dinh
PR-Team: Minh-Châu Phạm, Hoàng-Nguyên Vũ và Đăng-Nhã Nguyễn

Ngày 19 tháng 2 năm 2024

Phần I. Giới thiệu



Hình 1: Hệ thống dịch máy Google.

Dịch Máy (Machine Translation) với mục đích tự động dịch văn bản hoặc lời nói từ ngôn ngữ tự nhiên này sang ngôn ngữ tự nhiên khác. Dịch máy sử dụng kết hợp nhiều ý tưởng và các kỹ thuật với nhau từ ngôn ngữ học, khoa học máy tính, xác suất thống kê và trí tuệ nhân tạo. Với mục tiêu của dịch máy là phát triển một hệ thống cho phép tạo ra bản dịch chính xác giữa các ngôn ngữ tự nhiên của con người.

Các hệ thống dịch máy điển hình hiện này như: Google Translate, Bing Translator,... đã đạt được chất lượng bản dịch tốt và được tích hợp trong nhiều nền tảng ứng dụng khác nhau và có thể dịch tốt giữa hơn 100 các ngôn ngữ tự nhiên.

Như vậy, Input/Output của bài toán là:

- **Input:** Văn bản đầu vào của ngôn ngữ nguồn.
  VD: Câu đầu vào tiếng việt: "Tôi đang học NLP"
- **Output:** Văn bản được dịch sang ngôn ngữ đích.
  VD: Câu được dịch sang tiếng anh: " I am learning NLP"

Mô hình hoá bài toán dịch máy, với mục tiêu là huấn luyện và tối ưu các tham số mô hình θ với đầu vào văn bản từ ngôn ngữ nguồn $w^{(s)}$ và đầu ra văn bản từ ngôn ngữ đích tương ứng $w^{(s)}$:

$$\hat{w}^{(t)} = argmax_{(w^{(t)})}\theta(w^{(s)}), w^{(t)}))$$

Vì vậy, để giải quyết tốt bài toán dịch máy, chúng ta cần quan tâm tối ưu:

- **Phần 1:** Thuật toán học tối ưu bộ tham số θ
- **Phần 2:** Thuật toán giải mã (decoding) để sinh ra bản dịch tốt nhất cho văn bản đầu vào

Hiện nay, có ba hướng tiếp cận chính cho bài toán này:

- **Hướng 1:** Dịch máy dựa vào luật (Rule-based Machine Translation - RBMT)
- **Hướng 2:** Dịch máy dựa vào thống kê (Statistical Machine Translation - SMT)
- **Hướng 3:** Dịch máy dựa vào mạng nơ-ron (Neural Machine Translation - NMT)

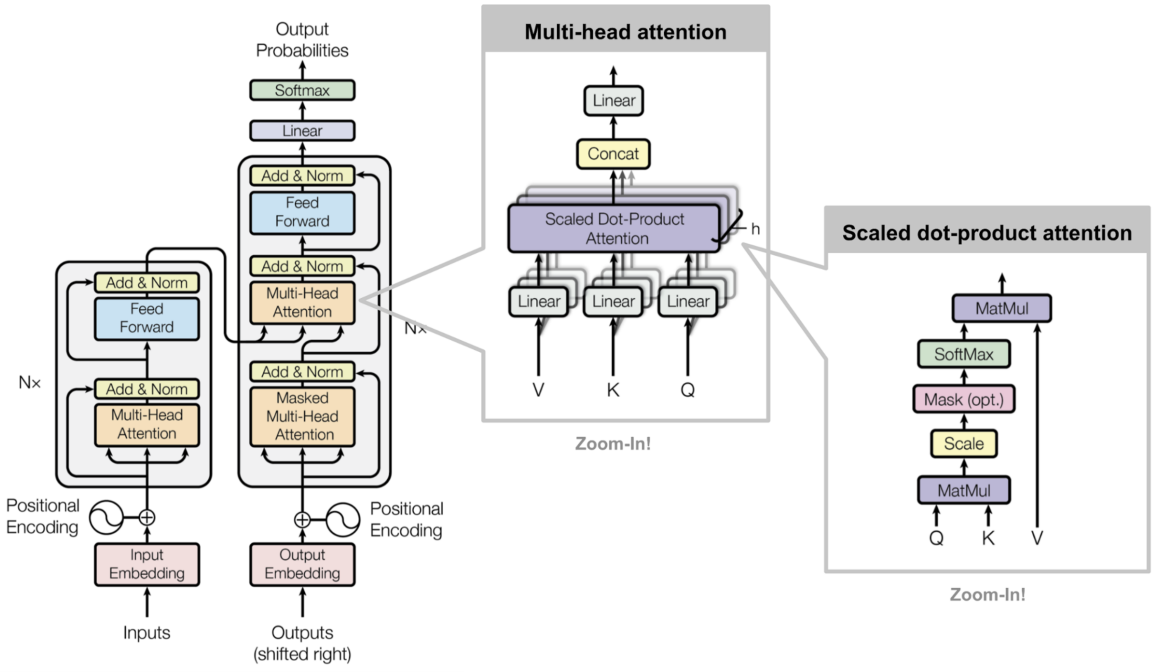
Trong các hướng tiếp cận này NMT đang ngày càng phát triển và cho chất lượng bản dịch tốt vượt trội. Vì vậy, phạm vi project tập trung vào các phương pháp dựa trên mạng nơ-ron gồm 2 nội dung chính:

- **Phương pháp 1:** Xây dựng mô hình dịch máy sử dụng kiến trúc Transformer
- **Phương pháp 2:** Xây dựng mô hình dịch máy sử dụng Pre-trained Language Model như BERT và GPT

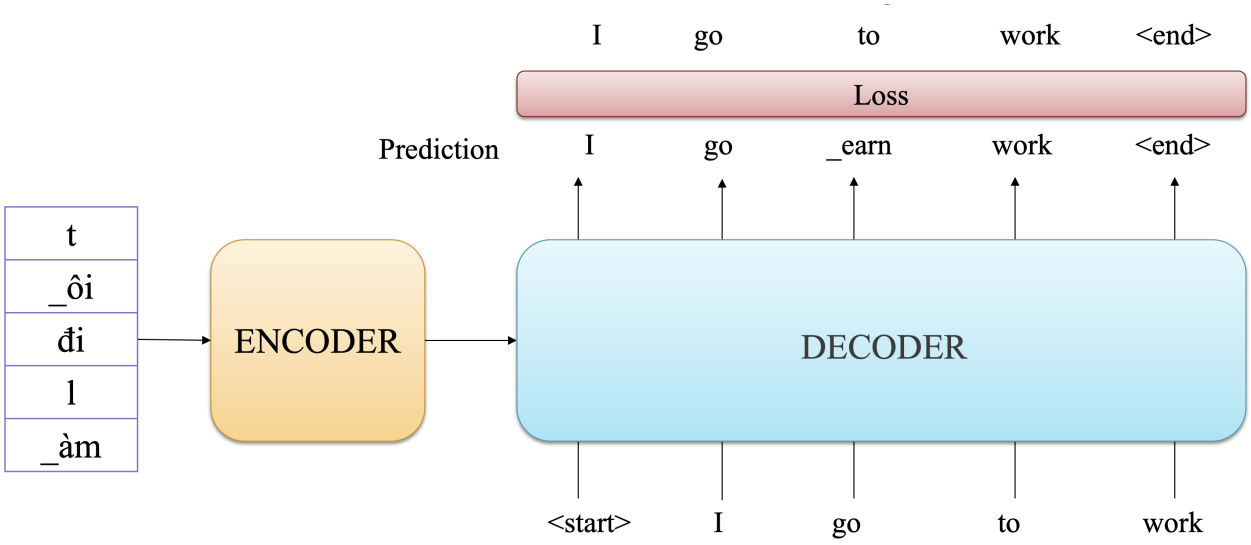
Phần II. Machine Translation using Transformer Model

Để xây dựng, đánh giá hiệu suất các mô hình chúng ta sử dụng bộ dữ liệu dịch **IWSLT'15 English -Vietnamese** với số lượng mẫu cho training: 133,317 cặp câu song ngữ, tập validation: 1,553 cặp câu song ngữ và tập test: 1,269 cặp câu song ngữ. Chiều dịch sẽ từ tiếng anh sang tiếng việt.

Metric để đánh giá chúng ta sử dụng: ScacreBleu (BLEU)



Hình 2: Mô hình Transformer.



Hình 3: Dịch máy sử dụng mô hình Transformer.

1. Build Dataset

```
# import libs
!pip install -q datasets sacrebleu

# download dataset
from datasets import load_dataset

data = load_dataset(
    "mt_eng_vietnamese",
    "iwslt2015-en-vi"
)

# tokenization
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator

SRC_LANGUAGE = 'en'
TGT_LANGUAGE = 'vi'

token_transform = {}
vocab_transform = {}

token_transform[SRC_LANGUAGE] = get_tokenizer('basic_english')
token_transform[TGT_LANGUAGE] = get_tokenizer('basic_english')

UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX = 0, 1, 2, 3
special_symbols = ['<unk>', '<pad>', '<bos>', '<eos>']

def yield_tokens(data_iter, lang):
    for data_sample in data_iter['translation']:
        yield token_transform[lang](data_sample[lang])


for lang in [SRC_LANGUAGE, TGT_LANGUAGE]:
    train_iter = data['train']

    # Create torchtext's Vocab object
    vocab_transform[lang] = build_vocab_from_iterator(
        yield_tokens(train_iter, lang),
        min_freq=1,
        specials=special_symbols,
        special_first=True
    )

    vocab_transform[lang].set_default_index(UNK_IDX)

# dataloader
import torch
from torch.nn.utils.rnn import pad_sequence

# helper function to club together sequential operations
def sequential_transforms(*transforms):
    def func(txt_input):
        for transform in transforms:
            txt_input = transform(txt_input)
        return txt_input
    return func

# function to add BOS/EOS and create tensor for input sequence indices
```

```
def tensor_transform(token_ids):
    return torch.cat((torch.tensor([BOS_IDX]),
                      torch.tensor(token_ids),
                      torch.tensor([EOS_IDX])))

# ``src`` and ``tgt`` language text transforms to convert raw strings into tensors indices
text_transform = {}
for lang in [SRC_LANGUAGE, TGT_LANGUAGE]:
    text_transform[lang] = sequential_transforms(
        token_transform[lang], # Tokenization
        vocab_transform[lang], # Numericalization
        tensor_transform # Add BOS/EOS and create tensor
    )

# function to collate data samples into batch tensors
def collate_fn(batch):
    src_batch, tgt_batch = [], []
    for sample in batch:
        src_sample, tgt_sample = sample[SRC_LANGUAGE], sample[TGT_LANGUAGE]
        src_batch.append(text_transform[SRC_LANGUAGE](src_sample).to(dtype=torch.int64))
        tgt_batch.append(text_transform[TGT_LANGUAGE](tgt_sample).to(dtype=torch.int64))

    src_batch = pad_sequence(src_batch, padding_value=PAD_IDX, batch_first=True)
    tgt_batch = pad_sequence(tgt_batch, padding_value=PAD_IDX, batch_first=True)
    return src_batch, tgt_batch

from torch.utils.data import DataLoader

BATCH_SIZE = 8

train_dataloader = DataLoader(
    data['train']['translation'],
    batch_size=BATCH_SIZE,
    collate_fn=collate_fn
)

valid_dataloader = DataLoader(
    data['validation']['translation'],
    batch_size=BATCH_SIZE,
    collate_fn=collate_fn
)

test_dataloader = DataLoader(
    data['test']['translation'],
    batch_size=BATCH_SIZE,
    collate_fn=collate_fn
)
```

2. Modeling

```
from torch import Tensor
import torch
import torch.nn as nn
from torch.nn import Transformer
import math
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

```

```python
# helper Module that adds positional encoding to the token embedding to introduce a notion of word order.
class PositionalEncoding(nn.Module):
    def __init__(self,
                 emb_size: int,
                 dropout: float,
                 maxlen: int = 5000):
        super(PositionalEncoding, self).__init__()
        den = torch.exp(- torch.arange(0, emb_size, 2)* math.log(10000) / emb_size)
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        pos_embedding = torch.zeros((maxlen, emb_size))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding: Tensor):
        return self.dropout(token_embedding + self.pos_embedding[:token_embedding.size(0), :])

# helper Module to convert tensor of input indices into corresponding tensor of token embeddings
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size

    def forward(self, tokens: Tensor):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)

# Seq2Seq Network
class Seq2SeqTransformer(nn.Module):
    def __init__(self,
                 num_encoder_layers: int,
                 num_decoder_layers: int,
                 emb_size: int,
                 nhead: int,
                 src_vocab_size: int,
                 tgt_vocab_size: int,
                 dim_feedforward: int = 512,
                 dropout: float = 0.1):
        super(Seq2SeqTransformer, self).__init__()
        self.transformer = Transformer(d_model=emb_size,
                                       nhead=nhead,
                                       num_encoder_layers=num_encoder_layers,
                                       num_decoder_layers=num_decoder_layers,
                                       dim_feedforward=dim_feedforward,
                                       dropout=dropout,
                                       batch_first=True)
        self.generator = nn.Linear(emb_size, tgt_vocab_size)
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(
            emb_size, dropout=dropout)

    def forward(self,
                src: Tensor,
```

```
                trg: Tensor,
                src_mask: Tensor,
                tgt_mask: Tensor,
                src_padding_mask: Tensor,
                tgt_padding_mask: Tensor,
                memory_key_padding_mask: Tensor):
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        outs = self.transformer(src_emb, tgt_emb, src_mask, tgt_mask, None,
                                src_padding_mask, tgt_padding_mask,
    memory_key_padding_mask)
        return self.generator(outs)

    def encode(self, src: Tensor, src_mask: Tensor):
        return self.transformer.encoder(self.positional_encoding(
                            self.src_tok_emb(src)), src_mask)

    def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):
        return self.transformer.decoder(self.positional_encoding(
                          self.tgt_tok_emb(tgt)), memory,
                          tgt_mask)

def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1,
    float(0.0))
    return mask


def create_mask(src, tgt):
    src_seq_len = src.shape[1]
    tgt_seq_len = tgt.shape[1]

    tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
    src_mask = torch.zeros((src_seq_len, src_seq_len),device=DEVICE).type(torch.bool)

    src_padding_mask = (src == PAD_IDX)
    tgt_padding_mask = (tgt == PAD_IDX)
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

3. Trainer

```
import time

def train_epoch(model, optimizer, criterion, train_dataloader, device):
    model.train()
    losses = []

    for src_ids, tgt_ids in train_dataloader:
        src_ids = src_ids.to(device)
        tgt_ids = tgt_ids.to(device)

        tgt_input = tgt_ids[:, :-1]
        tgt_output = tgt_ids[:, 1:]

        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src_ids,
    tgt_input)
        try:
            output = model(
                src_ids, tgt_input, src_mask, tgt_mask,src_padding_mask,
```

```
    tgt_padding_mask, src_padding_mask
            )
        except:
            print(src_ids.shape, tgt_input.shape)

        optimizer.zero_grad()

        loss = criterion(
            output.reshape(-1, output.shape[-1]),
            tgt_output.reshape(-1))
        loss.backward()

        optimizer.step()
        losses.append(loss.item())

    return sum(losses) / len(losses)

def evaluate(model, data_loader, criterion, device):
    model.eval()
    losses = []
    with torch.no_grad():
        for src_ids, tgt_ids in data_loader:
            src_ids = src_ids.to(device)
            tgt_ids = tgt_ids.to(device)

            tgt_input = tgt_ids[:, :-1]
            tgt_output = tgt_ids[:, 1:]

            src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(
    src_ids, tgt_input)
            output = model(
                src_ids, tgt_input, src_mask, tgt_mask,src_padding_mask,
    tgt_padding_mask, src_padding_mask
            )
            loss = criterion(
                output.reshape(-1, output.shape[-1]),
                tgt_output.reshape(-1)
            )
            losses.append(loss.item())
    return sum(losses) / len(losses)

def train(model, train_dataloader, valid_dataloader, optimizer, criterion, device,
    epochs):
    for epoch in range(1, epochs+1):
        start_time = time.time()
        train_loss = train_epoch(model, optimizer, criterion, train_dataloader, device
    )
        valid_loss = evaluate(model, valid_dataloader, criterion, device)
        end_time = time.time()
        print((f"Epoch: {epoch}, Train loss: {train_loss:.3f}, Val loss: {valid_loss
    :.3f}, "f"Epoch time = {(end_time - start_time):.3f}s"))
```

4. Training

```
SRC_VOCAB_SIZE = len(vocab_transform[SRC_LANGUAGE])
TGT_VOCAB_SIZE = len(vocab_transform[TGT_LANGUAGE])
EMB_SIZE = 512
NHEAD = 8
FFN_HID_DIM = 512
BATCH_SIZE = 128
```

```
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3

transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS, EMB_SIZE,
                                 NHEAD, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, FFN_HID_DIM)
transformer = transformer.to(DEVICE)

criterion = torch.nn.CrossEntropyLoss(ignore_index=PAD_IDX)

optimizer = torch.optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98),
    eps=1e-9)

epochs = 5
train(transformer, train_dataloader, valid_dataloader, optimizer, criterion, DEVICE,
    epochs)
```

5. Evaluation

```
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    src = src.to(DEVICE)
    src_mask = src_mask.to(DEVICE)

    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type(torch.long).to(DEVICE)
    for i in range(max_len-1):
        memory = memory.to(DEVICE)
        tgt_mask = (generate_square_subsequent_mask(ys.size(1))
                    .type(torch.bool)).to(DEVICE)
        out = model.decode(ys, memory, tgt_mask)
        out = out.transpose(0, 1)
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word[-1].item()

        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
        if next_word == EOS_IDX:
            break
    return ys


# actual function to translate input sentence into target language
def translate(model: torch.nn.Module, src_sentence: str):
    model.eval()
    src = text_transform[SRC_LANGUAGE](src_sentence).view(1, -1)
    num_tokens = src.shape[1]
    src_mask = (torch.zeros(num_tokens, num_tokens)).type(torch.bool)
    tgt_tokens = greedy_decode(
        model,  src, src_mask, max_len=num_tokens + 5, start_symbol=BOS_IDX).flatten()
    return " ".join(
        vocab_transform[TGT_LANGUAGE].lookup_tokens(list(tgt_tokens.cpu().numpy()))).
    replace("<bos>", "").replace("<eos>", "")
translate(transformer, "i go to school") # => toi den truong

# evaluate on test set
from tqdm import tqdm
import sacrebleu

pred_sentences, tgt_sentences = [], []
for sample in tqdm(data['test']['translation']):
```

```
    src_sentence = sample[SRC_LANGUAGE]
    tgt_sentence = sample[TGT_LANGUAGE]

    pred_sentence = translate(transformer, src_sentence)
    pred_sentences.append(pred_sentence)

    tgt_sentences.append(tgt_sentence)

bleu_score = sacrebleu.corpus_bleu(pred_sentences, [tgt_sentences], force=True)
bleu_score => 46.8/16.8/6.4/2.4
```

Phần III. Machine Translation using Pre-trained LMs

Các pre-trained language model dựa vào kiến trúc transformer được huấn luyện trên tập dữ liệu lớn. Vì vậy các mô hình ngôn ngữ đạt hiệu quả tốt cho các tác vụ NLU (Natural Language Understanding). Trong đó điển hình là BERT cốt lõi là các khối Transformer-Encder và GPT (Cụ thể trong project này sẽ sử dụng GPT2) cốt lõi là các khối Transformer-Decoder. Dựa trên kiến trúc đó, chúng ta hoàn toàn có thể sửu dụng trong số các mô hình như BERT và GPT2 để khởi tạo trọng số cho mô hình Transformer.

Ở trong phần này, chúng ta sẽ sử dụng các pre-trained LMs model là BERT và GPT2 để khởi tạo cho Transformer thông qua thư viện "transformers" của huggingface. Gồm có 2 cách chính:

- **Cách 1:** BERT cho Transformer-Encoder và BERT cho Transformer-Decoder
- **Cách 2:** BERT cho Transformer-Encoder và GPT2 cho Transformer-Decoder

1. Build Dataset

```
# install libs
!pip install -q datasets sacrebleu accelerate>=0.20.1

# import libs
import os
import numpy as np
import sacrebleu
import torch
from torch.utils.data import Dataset
from datasets import load_dataset, load_metric
from transformers import *

# load dataset
class NMTDataset(Dataset):
    def __init__(self, cfg, data_type="train"):
        super().__init__()
        self.cfg = cfg

        self.src_texts, self.tgt_texts = self.read_data(data_type)

        self.src_input_ids, self.src_attention_mask = self.texts_to_sequences(self.
    src_texts)
        self.tgt_input_ids, self.tgt_attention_mask, self.labels = self.
    texts_to_sequences(
            self.tgt_texts,
            is_src=False
        )

    def read_data(self, data_type):
        data = load_dataset(
            "mt_eng_vietnamese",
            "iwslt2015-en-vi",
            split=data_type
        )
        src_texts = [sample["translation"][self.cfg.src_lang] for sample in data]
        tgt_texts = [sample["translation"][self.cfg.tgt_lang] for sample in data]
        return src_texts, tgt_texts
```

```
36
37      def texts_to_sequences(self, texts, is_src=True):
38          if is_src:
39              src_inputs = self.cfg.src_tokenizer(
40                  texts,
41                  padding='max_length',
42                  truncation=True,
43                  max_length=self.cfg.src_max_len,
44                  return_tensors='pt'
45              )
46              return (
47                  src_inputs.input_ids,
48                  src_inputs.attention_mask
49              )
50
51          else:
52              if self.cfg.add_special_tokens:
53                  texts = [
54                      ' '.join([
55                          self.cfg.tgt_tokenizer.bos_token,
56                          text,
57                          self.cfg.tgt_tokenizer.eos_token
58                          ])
59                      for text in texts
60                  ]
61              tgt_inputs = self.cfg.tgt_tokenizer(
62                  texts,
63                  padding='max_length',
64                  truncation=True,
65                  max_length=self.cfg.tgt_max_len,
66                  return_tensors='pt'
67              )
68
69              labels = tgt_inputs.input_ids.numpy().tolist()
70              labels = [
71                  [
72                      -100 if token_id == self.cfg.tgt_tokenizer.pad_token_id else
    token_id
73                      for token_id in label
74                  ]
75                  for label in labels
76              ]
77
78              labels = torch.LongTensor(labels)
79
80              return (
81                  tgt_inputs.input_ids,
82                  tgt_inputs.attention_mask,
83                  labels
84              )
85
86      def __getitem__(self, idx):
87          return {
88              "input_ids": self.src_input_ids[idx],
89              "attention_mask": self.src_attention_mask[idx],
90              "decoder_input_ids": self.tgt_input_ids[idx],
91              "decoder_attention_mask": self.tgt_attention_mask[idx],
92              "labels": self.labels[idx]
93          }
94
```

```
    def __len__(self):
        return np.shape(self.src_input_ids)[0]
```

2. Tokenizer

```
def postprocess_text(preds, labels):
    preds = [pred.strip() for pred in preds]
    labels = [[label.strip()] for label in labels]
    return preds, labels

def load_tokenizer(model_name_or_path):
    if 'bert' in model_name_or_path.split('-'):
        return BertTokenizerFast.from_pretrained(model_name_or_path)
    elif 'gpt2' in model_name_or_path.split('-'):
        return GPT2TokenizerFast.from_pretrained(model_name_or_path)
    else:
        return AutoTokenizer.from_pretrained(model_name_or_path)
```

3. Trainer

```
class Manager():
    def __init__(self, cfg, is_train=True):
        self.cfg = cfg

        print("Loading Tokenizer...")
        self.get_tokenizer()

        print("Loading Model...")
        self.get_model()

        print("Loading Metric...")
        self.bleu_metric = load_metric("sacrebleu")

        print("Check Save Model Path")
        if not os.path.exists(self.cfg.ckpt_dir):
            os.mkdir(self.cfg.ckpt_dir)

        if is_train:
            # Load dataloaders
            print("Loading Dataset...")
            self.train_dataset = NMTDataset(self.cfg, data_type="train")
            self.valid_dataset = NMTDataset(self.cfg, data_type="validation")

        print("Setting finished.")

    def get_tokenizer(self):
        if self.cfg.load_model_from_path:
            self.cfg.src_tokenizer = load_tokenizer(self.cfg.ckpt_dir)
            self.cfg.tgt_tokenizer = load_tokenizer(self.cfg.ckpt_dir)
        else:
            self.cfg.src_tokenizer = load_tokenizer(self.cfg.src_model_name)
            self.cfg.tgt_tokenizer = load_tokenizer(self.cfg.tgt_model_name)
            if "bert" in self.cfg.tgt_model_name.split('-'):
                self.cfg.add_special_tokens = False
                self.cfg.bos_token_id = self.cfg.tgt_tokenizer.cls_token_id
                self.cfg.eos_token_id = self.cfg.tgt_tokenizer.sep_token_id
                self.cfg.pad_token_id = self.cfg.tgt_tokenizer.pad_token_id
            else:
                self.cfg.add_special_tokens = True
                self.cfg.tgt_tokenizer.add_special_tokens(
```

```
                    {
                        "bos_token": "[BOS]",
                        "eos_token": "[EOS]",
                        "pad_token": "[PAD]"
                    }
                )
                self.cfg.bos_token_id = self.cfg.tgt_tokenizer.bos_token_id
                self.cfg.eos_token_id = self.cfg.tgt_tokenizer.eos_token_id
                self.cfg.pad_token_id = self.cfg.tgt_tokenizer.pad_token_id
        self.cfg.src_tokenizer.save_pretrained(
                os.path.join(self.cfg.ckpt_dir, f"{self.cfg.src_lang}_tokenizer_{cfg.src_model_name}")
            )

        self.cfg.tgt_tokenizer.save_pretrained(
                os.path.join(self.cfg.ckpt_dir, f"{self.cfg.tgt_lang}_tokenizer_{cfg.tgt_model_name}")
            )

    def get_model(self):
        if self.cfg.load_model_from_path:
            save_model_path = os.path.join(self.cfg.ckpt_dir, self.cfg.ckpt_name)
            self.model = EncoderDecoderModel.from_pretrained(save_model_path)
        else:
            self.model = EncoderDecoderModel.from_encoder_decoder_pretrained(
                self.cfg.src_model_name,
                self.cfg.tgt_model_name
            )
            self.model.decoder.resize_token_embeddings(len(self.cfg.tgt_tokenizer))
            self.model.config.decoder_start_token_id = self.cfg.bos_token_id
            self.model.config.eos_token_id = self.cfg.eos_token_id
            self.model.config.pad_token_id = self.cfg.pad_token_id
            self.model.config.vocab_size = len(self.cfg.tgt_tokenizer)
            self.model.config.max_length = self.cfg.max_length_decoder
            self.model.config.min_length = self.cfg.min_length_decoder
            self.model.config.no_repeat_ngram_size = 3
            self.model.config.early_stopping = True
            self.model.config.length_penalty = 2.0
            self.model.config.num_beams = self.cfg.beam_size

    def train(self):
        print("Training...")
        if self.cfg.use_eval_steps:
            training_args = Seq2SeqTrainingArguments(
                predict_with_generate=True,
                evaluation_strategy="steps",
                save_strategy='steps',
                save_steps=self.cfg.eval_steps,
                eval_steps=self.cfg.eval_steps,
                output_dir=self.cfg.ckpt_dir,
                per_device_train_batch_size=self.cfg.train_batch_size,
                per_device_eval_batch_size=self.cfg.eval_batch_size,
                learning_rate=self.cfg.learning_rate,
                weight_decay=0.005,
                num_train_epochs=self.cfg.num_train_epochs
            )
        else:
            training_args = Seq2SeqTrainingArguments(
                predict_with_generate=True,
                evaluation_strategy="epoch",
```

```
                save_strategy='epoch',
                output_dir=self.cfg.ckpt_dir,
                per_device_train_batch_size=self.cfg.train_batch_size,
                per_device_eval_batch_size=self.cfg.eval_batch_size,
                learning_rate=self.cfg.learning_rate,
                weight_decay=0.005,
                num_train_epochs=self.cfg.num_train_epochs
            )

        data_collator = DataCollatorForSeq2Seq(
            self.cfg.tgt_tokenizer,
            model=self.model
        )

        trainer = Seq2SeqTrainer(
            self.model,
            training_args,
            train_dataset=self.train_dataset,
            eval_dataset=self.valid_dataset,
            data_collator=data_collator,
            tokenizer=self.cfg.tgt_tokenizer,
            compute_metrics=self.compute_metrics
        )

        trainer.train()

    def compute_metrics(self, eval_preds):
        preds, labels = eval_preds
        if isinstance(preds, tuple):
            preds = preds[0]
        decoded_preds = self.cfg.tgt_tokenizer.batch_decode(preds, skip_special_tokens=True)

        labels = np.where(labels != -100, labels, self.cfg.tgt_tokenizer.pad_token_id)
        decoded_labels = self.cfg.tgt_tokenizer.batch_decode(labels, skip_special_tokens=True)

        decoded_preds, decoded_labels = postprocess_text(decoded_preds, decoded_labels)

        result = self.bleu_metric.compute(
            predictions=decoded_preds,
            references=decoded_labels
        )

        result = {"bleu_score": result["score"]}

        prediction_lens = [np.count_nonzero(pred != self.cfg.tgt_tokenizer.pad_token_id) for pred in preds]
        result["gen_len"] = np.mean(prediction_lens)
        result = {k: round(v, 4) for k, v in result.items()}

        return result
```

4. Training

```
class BaseConfig:
    """ base Encoder Decoder config """

    def __init__(self, **kwargs):
```

```
        for k, v in kwargs.items():
            setattr(self, k, v)

class NMTConfig(BaseConfig):
    # Data
    src_lang = 'en'
    tgt_lang = 'vi'
    src_max_len = 75
    tgt_max_len = 75

    # Model
    src_model_name = "bert-base-multilingual-cased"
    tgt_model_name = "bert-base-multilingual-cased"

    # Training
    load_model_from_path = False
    device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
    learning_rate = 3e-5
    train_batch_size = 16
    eval_batch_size = 8
    num_train_epochs = 5
    ckpt_dir = src_model_name + '_to_' + tgt_model_name
    use_eval_steps = False
    eval_steps = 2000

    # Inference
    max_length_decoder = 75
    min_length_decoder = 25
    beam_size = 1

cfg = NMTConfig()

manager = Manager(cfg, is_train=True)

manager.train()
```

5. Prediction

```
# load model
def load_model(cfg, checkpoint_name):
    # Load Tokenizer
    src_tokenizer_save_path = f"{cfg.ckpt_dir}/{cfg.src_lang}_tokenizer_{cfg.src_model_name}"
    src_tokenizer = BertTokenizerFast.from_pretrained(src_tokenizer_save_path)

    tgt_tokenizer_save_path = f"{cfg.ckpt_dir}/{cfg.tgt_lang}_tokenizer_{cfg.tgt_model_name}"
    tgt_tokenizer = GPT2TokenizerFast.from_pretrained(tgt_tokenizer_save_path)

    # Load Model
    model_save_path = f"{cfg.ckpt_dir}/{checkpoint_name}"
    model = EncoderDecoderModel.from_pretrained(model_save_path)

    # Inference Param
    device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

    return src_tokenizer, tgt_tokenizer, model, device

```

```python
from tqdm import tqdm
def inference(
    text,
    src_tokenizer,
    tgt_tokenizer,
    model,
    device="cpu",
    max_length=75,
    beam_size=5
    ):
    inputs = src_tokenizer(
        text,
        padding="max_length",
        truncation=True,
        max_length=max_length,
        return_tensors="pt"
        )
    input_ids = inputs.input_ids.to(device)
    attention_mask = inputs.attention_mask.to(device)
    model.to(device)

    outputs = model.generate(
        input_ids,
        attention_mask=attention_mask,
        max_length=max_length,
        early_stopping=True,
        num_beams=beam_size,
        length_penalty=2.0
    )

    output_str = tgt_tokenizer.batch_decode(outputs, skip_special_tokens=True)

    return output_str

def inference_bath(
    texts,
    src_tokenizer,
    tgt_tokenizer,
    model,
    device="cpu",
    max_length=75,
    beam_size=5,
    batch_size=32
    ):

    pred_texts = []

    if len(texts) < batch_size:
        batch_size = len(texts)

    for x in tqdm(range(0, len(texts), batch_size)):
        text = texts[x:x+batch_size]

        inputs = src_tokenizer(
            text,
            padding="max_length",
            truncation=True,
            max_length=max_length,
            return_tensors="pt"
            )
```

```
79
80          input_ids = inputs.input_ids.to(device)
81          attention_mask = inputs.attention_mask.to(device)
82          model.to(device)
83
84          outputs = model.generate(
85              input_ids,
86              attention_mask=attention_mask,
87              max_length=max_length,
88              early_stopping=True,
89              num_beams=beam_size,
90              length_penalty=2.0
91          )
92
93          output_str = tgt_tokenizer.batch_decode(outputs, skip_special_tokens=True)
94          pred_texts.extend(output_str)
95          torch.cuda.empty_cache()
96
97      return pred_texts
98
99  class BaseConfig:
100     """ base Encoder Decoder config """
101
102     def __init__(self, **kwargs):
103         for k, v in kwargs.items():
104             setattr(self, k, v)
105
106 class NMTConfig(BaseConfig):
107     # Data
108     src_lang = 'en'
109     tgt_lang = 'vi'
110     src_max_len = 75
111     tgt_max_len = 75
112
113     # Model
114     src_model_name = "bert-base-multilingual-cased"
115     tgt_model_name = "bert-base-multilingual-cased"
116
117     # Training
118     load_model_from_path = False
119     device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu'
        )
120     learning_rate = 3e-5
121     train_batch_size = 16
122     eval_batch_size = 8
123     num_train_epochs =15
124     ckpt_dir = src_model_name + '_to_' + tgt_model_name
125     use_eval_steps = False
126     eval_steps = 2000
127
128     # Inference
129     max_length_decoder = 75
130     min_length_decoder = 25
131     beam_size = 5
132
133 cfg = NMTConfig()
134
135 # load data
136 data = load_dataset("mt_eng_vietnamese", "iwslt2015-en-vi", split="test")
137 src_texts = [sample["translation"]["en"] for sample in data]
```

```
tgt_texts = [sample["translation"]["vi"] for sample in data]

src_tokenizer, tgt_tokenizer, model, device = load_model(cfg, checkpoint_name="
    checkpoint-41665")

pred_texts = inference_bath(src_texts, src_tokenizer, tgt_tokenizer, model, device,
    beam_size=1)

sacrebleu.corpus_bleu(pred_texts, [tgt_texts]) # => 25.41 53.8/31.8/19.8/12.3
```

Phần 4. Câu hỏi trắc nghiệm

Câu hỏi 1 Mục tiêu của bài toán dịch máy là gì?
a) Xây dựng mô hình dịch từ ngôn ngữ nguồn sang ngôn ngữ đích
b) Xây dựng mô hình phân loại văn bản
c) Xây dựng mô hình tóm tắt văn bản
d) Xây dựng mô hình phân tích cảm xúc

Câu hỏi 2 Số lượng sample tập test của bộ dữ lịch sử dụng trong thực nghiệm là?
a) 1267
b) 1268
c) 1269
d) 1270

Câu hỏi 3 Hệ thống nào sau đây không phải là hệ thống dịch?
a) Bing Translator
b) Google Translate
c) ChatGPT
d) Dall-E

Câu hỏi 4 Thư viện sentencepiece dùng để làm gì?
a) Xây dựng Transformer-Encoder
b) Xây dựng Transformer-Decoder
c) Xây dựng Subword-Based Tokenization
d) Huấn luyện mô hình Transformer

Câu hỏi 5 BERT sử dụng kiến trúc nào sau đây?
a) Bi-LSTM
b) RNN
c) Transformer-Encoder
d) Transformer-Decoder

Câu hỏi 6 BERT-Large sử dụng bao nhiêu khối encoder?
a) 6
b) 12
c) 18
d) 24

Câu hỏi 7 Trong số các mask tokens trong quá trình huấn luyện BERT. Tỷ lệ mask nào là tối ưu nhất cho BERT?
a) 80% thay bởi token [MASK] : 20% thay bởi token bất kỳ : 0% giữ nguyên không đổi

b) 80% thay bởi token [MASK] : 0% thay bởi token bất kỳ : 20% giữ nguyên không đổi
c) 80% thay bởi token [MASK] : 10% thay bởi token bất kỳ : 10% giữ nguyên không đổi
d) 80% thay bởi token [MASK] : 15% thay bởi token bất kỳ : 5% giữ nguyên không đổi

Câu hỏi 8 GPT sử dụng kiến trúc nào sau đây?
a) Bi-LSTM
b) RNN
c) Transformer-Encoder
d) Transformer-Decoder

Câu hỏi 9 GPT-Small sử dụng bao nhiêu khối decoder?
a) 6
b) 12
c) 18
d) 24

Câu hỏi 10 Độ đo thường sử dụng cho mô hình dịch máy là?
a) F1
b) BLEU
c) Accuracy
d) ROUGE

- Hết -