544Project

December 2, 2021

[ ]:
```
import torch
import pandas as pd
import numpy as np
```

1 Prepare inputs

1.1 Need to do

1.2 Prepare data using RAKE

[ ]:
```
df = pd.read_csv("ROCStories_winter2017 - ROCStories_winter2017.csv")
# Need to prepare data format
```

1.3 Using Provided Data for testing

in https://github.com/rishishian/plan_write/tree/master/data

[ ]:
```
from tqdm import tqdm

# split src data to (title+line) and (line+story)
src_data_path_list = ['train_title_line_story.txt', 'valid_title_line_story.
 ↪txt', 'test_title_line_story.txt']
t2l_data_path_list = ['train_title_line.tsv', 'valid_title_line.tsv',␣
 ↪'test_title_line.tsv']
l2s_data_path_list = ['train_line_story.tsv', 'valid_line_story.tsv',␣
 ↪'test_line_story.tsv']
```

[ ]:
```
def parse_line(line):
    title, rest = line.split('<EOT>')
    story_line, story = rest.split('<EOL>')
    return title.strip(), story_line.strip(), story.strip()


for src_data_path, t2l_data_path, l2s_data_path in zip(src_data_path_list,␣
 ↪t2l_data_path_list, l2s_data_path_list):
    with open(src_data_path, 'r') as src_file:
        with open(t2l_data_path, 'w') as t2l_file:
```

```
            with open(l2s_data_path, 'w') as l2s_file:
                print(f'Processing {src_data_path}')
                src_lines = src_file.readlines()
                for line in tqdm(src_lines):
                    title, story_line, story = parse_line(line)
                    t2l_file.write(title + '\t' + story_line + '\n')
                    l2s_file.write(story_line + '\t' + story + '\n')

# ground-truth story for testset
gt_testset_path = 'test_story.txt'
with open(src_data_path_list[-1], 'r') as src_file:
    with open(gt_testset_path, 'w') as gt_file:
        src_lines = src_file.readlines()
        for line in tqdm(src_lines):
            title, story_line, story = parse_line(line)
            gt_file.write(story + '\n')
```

```
Processing train_title_line_story.txt

100%|| 80186/80186 [00:00<00:00, 351126.92it/s]

Processing valid_title_line_story.txt

100%|| 9816/9816 [00:00<00:00, 301990.63it/s]

Processing test_title_line_story.txt

100%|| 8159/8159 [00:00<00:00, 333135.33it/s]
100%|| 8159/8159 [00:00<00:00, 268398.35it/s]
```

[ ]:
```
import nltk
import numpy as np
import os
from torchtext.legacy.data import Field, TabularDataset


class TitleLine(TabularDataset):
    @staticmethod
    def sort_key(ex):
        return len(ex.story_line)


class LineStory(TabularDataset):
    @staticmethod
    def sort_key(ex):
        return len(ex.story_line)
```

```
VOCAB = Field(init_token='<sos>', eos_token='<eos>', lower=True)


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

2 Define all models used

[ ]:
```
import torch
import torch.nn as nn
import random
import torch.nn.functional as F


class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout,␣
↪n_layer=1):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, enc_hid_dim, bidirectional=True,␣
↪num_layers=n_layer)
        #nn.LSTM(self.embedding_size, self.hidden_size, num_layers=num_layers,␣
↪batch_first=True, bidirectional=True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        embedded = self.dropout(self.embedding(src))

        outputs, hidden = self.rnn(embedded)

        hidden = torch.tanh(self.fc(torch.cat((hidden[-2, :, :], hidden[-1, :, :
↪]), dim=1)))
```

```
        return outputs, hidden
```

[ ]:
```
class SelfAttention(torch.nn.Module):
    def __init__(self, qkv_dimensions, hidden_size=256, n_heads=4,␣
↪output_dim=None, dropout=0.1, normaliza_qk=False):
        super(SelfAttention, self).__init__()
        self.hidden_size = hidden_size
        self.n_heads = n_heads
        if self.hidden_size % self.n_heads != 0:
            raise ValueError("Hidden size must be evenly divisible by the␣
↪number of heads.")
        self.dropout = dropout
        self.dropout = nn.Dropout(dropout) if 0 < dropout < 1 else None
        self.normalize_qk = normaliza_qk

        q_dim, k_dim, v_dim = qkv_dimensions
        self.q_proj = nn.Linear(q_dim, self.hidden_size, bias=False)
        self.k_proj = nn.Linear(k_dim, self.hidden_size, bias=False)
        self.v_proj = nn.Linear(v_dim, self.hidden_size, bias=False)

        if output_dim is None:
            self.output_transform = None
        else:
            self.output_transform = nn.Linear(self.hidden_size, output_dim,␣
↪bias=False)

    @property
    def depth(self):
        return self.hidden_size // self.n_heads

    def forward(self, q, k, v):
        k_equal_q = k is None
        if self.q_proj is not None:
            q = self.q_proj(q)
        if k_equal_q:
            k = q
        elif self.k_proj is not None:
            k = self.k_proj(k)
        if self.v_proj is not None:
            v = self.v_proj(v)
        if self.n_heads > 1:
            q = self._split_heads(q)
            if not k_equal_q:
                k = self._split_heads(k)
            v = self._split_heads(v)
        if self.normalize_qk:
            q = q / torch.norm(q, dim=-1).unsqueeze(-1)
```

```
            if not k_equal_q:
                k = k / torch.norm(k, dim=-1).unsqueeze(-1)
        if k_equal_q:
            k = q
        q = q * self.depth ** -0.5

        # q, k, v  : [num_heads x B, T, depth]
        logits = torch.bmm(q, k.transpose(1, 2))
        weights = F.softmax(logits, dim=-1)
        if self.dropout is not None:
            weights = self.dropout(weights)
        attention_output = torch.bmm(weights, v)
        attention_output = self._combine_heads(attention_output)
        if self.output_transform is not None:
            attention_output = self.output_transform(attention_output)
        return attention_output

    def _split_heads(self, x):
        time_step = x.shape[1]
        return (
            x.view(-1, time_step, self.n_heads, self.depth)
                .transpose(1, 2).contiguous()
                .view(-1, time_step, self.depth)
        )

    def _combine_heads(self, x):
        time_step = x.shape[1]
        return (
            x.view(-1, self.n_heads, time_step, self.depth)
                .transpose(1, 2).contiguous()
                .view(-1, time_step, self.hidden_size)
        )
```

[ ]:
```
class Encoder_with_SelfAttn(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout,␣
↪n_layer=2):
        assert n_layer > 1
        print('Enter Encoder with Self Attention')
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, enc_hid_dim, bidirectional=True,␣
↪num_layers=n_layer)

        qkv_dimensions = [enc_hid_dim, enc_hid_dim, enc_hid_dim + emb_dim]
        self.self_attentions = torch.nn.ModuleList([
```

```
            SelfAttention(qkv_dimensions, enc_hid_dim, n_heads=4)
            for _ in range(n_layer - 1)
        ])

        input_dimensions = [emb_dim] + [enc_hid_dim] * (n_layer - 1)
        self.rnns = torch.nn.ModuleList([nn.GRU(
            dim, enc_hid_dim, 1,
            batch_first=True, bidirectional=True  # batch first
        ) for dim in input_dimensions])

        self.bidirectional_projections = torch.nn.ModuleList(
            [nn.Linear(enc_hid_dim * 2, enc_hid_dim, bias=False)
             for _ in range(n_layer)])

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)
        self.embedding_dropout = nn.Dropout(dropout)

    def forward(self, src):
        # src = [src sent len, batch size]
        src = src.transpose(0, 1)

        embedded = self.embedding_dropout(self.embedding(src))

        # embedded = [src sent len, batch size, emb dim]

        net = embedded
        for i, rnn in enumerate(self.rnns):
            net, final_state = rnn(net, None)
            if self.bidirectional_projections is not None and i < len(self.
↪rnns) - 1:
                net = self.bidirectional_projections[i](net)
            if self.self_attentions is not None and i < len(self.rnns) - 1:
                net = self.self_attentions[i](net, net, torch.cat([embedded,␣
↪net], dim=2))
        # net = [bs, len, hid_dim * 2]
        outputs = net.transpose(0, 1)

        # outputs = [src sent len, batch size, hid dim * num directions]

        # initial decoder hidden is final hidden state of the forwards and␣
↪backwards
        #  encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(net[:, -1, :]))

        # outputs = [src sent len, batch size, enc hid dim * 2]
        # hidden = [batch size, dec hid dim]
        return outputs, hidden
```

[ ]:
```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout,␣
↪attention):
        super().__init__()

        self.output_dim = output_dim
        self.attention = attention

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.LSTM((enc_hid_dim * 2) + emb_dim, dec_hid_dim)

        self.out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim,␣
↪output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):

        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))
        a = self.attention(hidden, encoder_outputs)
        a = a.unsqueeze(1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        weighted = torch.bmm(a, encoder_outputs)
        weighted = weighted.permute(1, 0, 2)
        rnn_input = torch.cat((embedded, weighted), dim=2)
        output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))
        assert (output == hidden).all()
        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        weighted = weighted.squeeze(0)

        output = self.out(torch.cat((output, weighted, embedded), dim=1))

        return output, hidden.squeeze(0)
```

[ ]:
```
class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim)
        self.v = nn.Parameter(torch.rand(dec_hid_dim))

    def forward(self, hidden, encoder_outputs):

        batch_size = encoder_outputs.shape[1]
```

```
        src_len = encoder_outputs.shape[0]

        # repeat encoder hidden state src_len times
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)

        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs),␣
↪dim=2)))

        energy = energy.permute(0, 2, 1)

        v = self.v.repeat(batch_size, 1).unsqueeze(1)

        attention = torch.bmm(v, energy).squeeze(1)

        return F.softmax(attention, dim=1)
```

[ ]:
```
class Seq2Seq(nn.Module):
    MAX_DECODE_LEN = 100

    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg=None, teacher_forcing_ratio=1.0):
        batch_size = src.shape[1]
        if trg is not None:
            max_len = trg.shape[0]
        else:
            assert teacher_forcing_ratio == 0
            max_len = Seq2Seq.MAX_DECODE_LEN

        trg_vocab_size = self.decoder.output_dim

        # tensor to store decoder outputs
        outputs = torch.zeros(max_len, batch_size, trg_vocab_size).to(self.
↪device)

        encoder_outputs, hidden = self.encoder(src)

        # first input to the decoder is the <sos> tokens
        input = src[0, :]
```

```
        for t in range(1, max_len):
            # insert input token embedding, previous hidden state and all␣
 ↪encoder hidden states
            # receive output tensor (predictions) and new hidden state
            output, hidden = self.decoder(input, hidden, encoder_outputs)

            # place predictions in a tensor holding predictions for each token
            outputs[t] = output
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = trg[t] if teacher_force else top1

        return outputs
```

[ ]:
```
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import random
import math
import time
from tqdm import tqdm

def train(model, src_field, trg_field, iterator, optimizer, criterion, clip,␣
 ↪teacher_force):
    model.train()
    epoch_loss = 0
    for i, batch in enumerate(iterator):
        src = getattr(batch, src_field)
        trg = getattr(batch, trg_field)
        optimizer.zero_grad()
        output = model(src, trg, teacher_force)
        output = output[1:].view(-1, output.shape[-1])
        trg = trg[1:].view(-1)
        loss = criterion(output, trg)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()
        epoch_loss += loss.item()
    return epoch_loss / len(iterator)


def evaluate(model, src_field, trg_field, iterator, criterion):
    model.eval()
    epoch_loss = 0
    with torch.no_grad():
        for i, batch in enumerate(iterator):
```

```
            src = getattr(batch, src_field)
            trg = getattr(batch, trg_field)
            output = model(src, trg, 0)  # no teacher forcing
            output = output[1:].view(-1, output.shape[-1])
            trg = trg[1:].view(-1)

            loss = criterion(output, trg)
            epoch_loss += loss.item()
    return epoch_loss / len(iterator)


def decode_story(output, vocab, join=' '):
    id_tensor = output.argmax(2)
    ids = id_tensor.transpose(0, 1)
    sent_batch = []
    for i in range(ids.shape[0]):
        sent = []
        for j in range(ids.shape[1]):
            w = vocab.itos[ids[i][j]]
            sent.append(w)
        sent = join.join(sent)
        sent_batch.append(sent)
    return sent_batch


def decode_story_line(output, vocab, join=' '):
    output = output.squeeze(1)
    sent = []
    values, indices = torch.topk(output, k=7, dim=1)
    decoded_indices = []
    # forbid any word to appear twice in storyline
    for i in range(output.shape[0]):  # for the i-th word
        for idx in indices[i]:  # for top-k candidate
            if idx not in decoded_indices:
                # a new word
                w = vocab.itos[idx]
                decoded_indices.append(idx)
                if w == '<unk>':  #
                    continue
                sent.append(w)
                break
    sent = join.join(sent)
    return [sent]


def test_generate(model, src_field, trg_field, iterator, criterion,␣
 ↪result_path, decode_func, compute_loss):
```

```
    model.eval()
    epoch_loss = 0
    generated_sentence = []
    with torch.no_grad():
        for i, batch in tqdm(enumerate(iterator)):
            src = getattr(batch, src_field)
            trg = getattr(batch, trg_field) if compute_loss else None
            output = model(src, trg, 0)
            generated_sentence.extend(decode_func(output, VOCAB.vocab, join=' ␣
 ↪'))
            output = output[1:].view(-1, output.shape[-1])
            if compute_loss:
                trg = trg[1:].view(-1)
                loss = criterion(output, trg)
                epoch_loss += loss.item()
    test_loss = epoch_loss / len(iterator) if epoch_loss else 'Test Loss Not␣
 ↪Computed.'
    with open(result_path, 'w') as f:
        for sent in generated_sentence:
            f.write(sent + '\n')
    return test_loss, generated_sentence
```

[ ]:
```
trainset_path = 'train_title_line.tsv'
validset_path = 'valid_title_line.tsv'
testset_path = 'test_title_line.tsv'
print(f'train/valid/test dataset path:{trainset_path}/{validset_path}/
 ↪{testset_path}')
```

```
train/valid/test dataset
path:train_title_line.tsv/valid_title_line.tsv/test_title_line.tsv
```

[ ]:
```
src_field, trg_field = 'title', 'story_line'
named_fields = [(src_field, VOCAB), (trg_field, VOCAB)]
print(f'src_filed:{src_field}, trg_field:{trg_field}')
```

```
src_filed:title, trg_field:story_line
```

[ ]:
```
class TitleLine(TabularDataset):
    @staticmethod
    def sort_key(ex):
        return len(ex.story_line)


class LineStory(TabularDataset):
    @staticmethod
    def sort_key(ex):
        return len(ex.story_line)
```

[ ]:
```
DataSet = TitleLine if trg_field == 'story_line' else LineStory
train_data = DataSet(path=trainset_path, format='tsv', fields=named_fields)
valid_data = DataSet(path=validset_path, format='tsv', fields=named_fields)
test_data = DataSet(path=testset_path, format='tsv', fields=named_fields)
```

[ ]:
```
print(f"Number of training examples: {len(train_data.examples)}")
print(f"Number of validation examples: {len(valid_data.examples)}")
print(f"Number of testing examples: {len(test_data.examples)}")
print('Show data example')
print(vars(train_data.examples[0]), vars(valid_data.examples[0]),␣
 ↪vars(test_data.examples[0]))
```

```
Number of training examples: 80186
Number of validation examples: 9816
Number of testing examples: 8159
Show data example
{'title': ['overweight', 'kid'], 'story_line': ['dan', 'overweight',
'unhealthy', 'make', 'decided']} {'title': ['the', 'pet', 'bug'], 'story_line':
['oliver', 'spotted', 'jar', 'hoped', 'safe']} {'title': ['literature', 'vs',
'math'], 'story_line': ['literature', 'choose', 'indecisive', 'make', 'deny']}
```

[ ]:
```
VOCAB.build_vocab(train_data, min_freq=25)
print(f"Unique tokens in vocabulary(min frequency:{25}): {len(VOCAB.vocab)}")
torch.cuda.is_available()
```

```
Unique tokens in vocabulary(min frequency:25): 3372
```

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

```
Using device: cuda
```

[ ]:
```
from torchtext.legacy.data import Iterator, BucketIterator
train_iterator, valid_iterator = BucketIterator.splits((train_data, valid_data),
                                                       batch_size=128,␣
 ↪device=device)
test_iterator = Iterator(test_data, batch_size=1, device=device, shuffle=False)
```

3 Title to Storyline

[ ]:
```
# model
INPUT_DIM = len(VOCAB.vocab)
OUTPUT_DIM = len(VOCAB.vocab)
ENC_EMB_DIM = 256
```

```
DEC_EMB_DIM = 256
ENC_HID_DIM = 256
DEC_HID_DIM = 256
N_LAYERS = 1
ENC_DROPOUT = 0
DEC_DROPOUT = 0
attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
# enc = Encoder_with_SelfAttn(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM,␣
 ↪ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT,␣
 ↪attn)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Device:{device}')
model = Seq2Seq(enc, dec, device).to(device)
print(f'The model has {count_parameters(model)} trainable parameters')

optimizer = optim.Adam(model.parameters())
PAD_IDX = VOCAB.vocab.stoi['<pad>']
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
N_EPOCHS = 5
CLIP = 1
TEACHER_FORCE = 0.5
MODEL_PATH = 'title2line.pt'
```

```
Device:cuda
The model has 7614508 trainable parameters
```

[8]:
```
print(f'Training {src_field} to {trg_field} model')
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss = train(model, src_field, trg_field, train_iterator, optimizer,␣
 ↪criterion, CLIP, TEACHER_FORCE)
    valid_loss = evaluate(model, src_field, trg_field, valid_iterator,␣
 ↪criterion)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), MODEL_PATH)
    print(f'Epoch: {epoch + 1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.
 ↪3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.
 ↪3f}')
```

```
Training story_line to story model
 Epoch: 01 | Time: 11m 57s
 Train Loss: 4.807 | Train PPL: 122.375
 Val. Loss: 5.520 |  Val. PPL: 249.582
 Epoch: 02 | Time: 11m 57s
 Train Loss: 4.163 | Train PPL:  64.270
 Val. Loss: 5.427 |  Val. PPL: 227.458
 Epoch: 03 | Time: 11m 58s
 Train Loss: 3.928 | Train PPL:  50.791
 Val. Loss: 5.516 |  Val. PPL: 248.587
 Epoch: 04 | Time: 11m 58s
  Train Loss: 3.793 | Train PPL:  44.407
 Val. Loss: 5.459 |  Val. PPL: 234.898
 Epoch: 05 | Time: 11m 58s
 Train Loss: 3.688 | Train PPL:  39.977
 Val. Loss: 5.460 |  Val. PPL: 235.056
 8159it [15:41,  8.66it/s])
```

[ ]:
```python
def test_bleu(result_path):
    GT_PATH = 'test_story.txt'
    with open(GT_PATH, encoding="utf-8") as f:
        refs = f.readlines()
        refs = [''.join(l.split('</s>')) for l in refs]

    with open(result_path, encoding='utf-8') as f:
        raw_sents = f.readlines()
        cans = []
        for l in raw_sents:
            for sep in ['</s>', '<unk>', '<eos>']:
                l = l.replace(sep, '')
            cans.append(l.strip())

    assert len(cans) == len(refs), print(len(cans), len(refs))
    score_list = []
    for ref, can in zip(refs, cans):
        score_list.append(bleu_score(ref, can))
    sentence_bleu = np.mean(score_list)
    print(f'Sentence bleu score:{sentence_bleu}')


def create_testfile(generated_line_path='title2line.txt'):
    base, ext = os.path.splitext(generated_line_path)
    ext = '_fortest.tsv'
    dest_file = base + ext
    print(f'Refining Test File(from:{generated_line_path}, to:{dest_file})')
    with open(generated_line_path, 'r') as f:
```

```
        lines = f.readlines()
        new_lines = [l.strip() + '\t' + f'STORY {i} TO BE GENERATED\n' for i, l␣
 ↪in enumerate(lines)]
        with open(dest_file, 'w') as wf:
            for line in new_lines:
                wf.write(line)
```

[ ]:
```
model.load_state_dict(torch.load(MODEL_PATH))
RESULT_PATH = 'title2line.txt'
decode_func = decode_story_line if trg_field == 'story_line' else decode_story
compute_loss = True if trg_field == 'story_line' else False
test_loss, result = test_generate(model, src_field, trg_field, test_iterator,␣
 ↪criterion, RESULT_PATH,
                                  decode_func, compute_loss)
if trg_field == 'story_line':
    print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss)} |')
    create_testfile(RESULT_PATH)
elif trg_field == 'story':
    test_bleu(RESULT_PATH)
```

4 Storyline to Story

[ ]:
```
trainset_path = 'train_line_story.tsv'
validset_path = 'valid_line_story.tsv'
testset_path = 'test_line_story.tsv'
print(f'train/valid/test dataset path:{trainset_path}/{validset_path}/
 ↪{testset_path}')
src_field, trg_field = 'story_line', 'story'
named_fields = [(src_field, VOCAB), (trg_field, VOCAB)]
print(f'src_filed:{src_field}, trg_field:{trg_field}')
DataSet = TitleLine if trg_field == 'story_line' else LineStory
train_data = DataSet(path=trainset_path, format='tsv', fields=named_fields)
valid_data = DataSet(path=validset_path, format='tsv', fields=named_fields)
test_data = DataSet(path=testset_path, format='tsv', fields=named_fields)
print(f"Number of training examples: {len(train_data.examples)}")
print(f"Number of validation examples: {len(valid_data.examples)}")
print(f"Number of testing examples: {len(test_data.examples)}")
print('Show data example')
print(vars(train_data.examples[0]), vars(valid_data.examples[0]),␣
 ↪vars(test_data.examples[0]))
```

[ ]:
```
VOCAB.build_vocab(train_data, min_freq=25)
print(f"Unique tokens in vocabulary(min frequency:{25}): {len(VOCAB.vocab)}")
train_iterator, valid_iterator = BucketIterator.splits((train_data, valid_data),
                                                       batch_size=128,␣
 ↪device=device)
```

```
test_iterator = Iterator(test_data, batch_size=1, device=device, shuffle=False)
```

[ ]:
```
# model
INPUT_DIM = len(VOCAB.vocab)
OUTPUT_DIM = len(VOCAB.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 256
DEC_HID_DIM = 256
N_LAYERS = 1
ENC_DROPOUT = 0
DEC_DROPOUT = 0
attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
# enc = Encoder_with_SelfAttn(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM,␣
 ↪ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT,␣
 ↪attn)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Device:{device}')
model = Seq2Seq(enc, dec, device).to(device)
print(f'The model has {count_parameters(model)} trainable parameters')

optimizer = optim.Adam(model.parameters())
PAD_IDX = VOCAB.vocab.stoi['<pad>']
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
N_EPOCHS = 5
CLIP = 1
TEACHER_FORCE = 0.5
MODEL_PATH = 'line2story.pt'
```

[ ]:
```
print(f'Training {src_field} to {trg_field} model')
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss = train(model, src_field, trg_field, train_iterator, optimizer,␣
 ↪criterion, CLIP, TEACHER_FORCE)
    valid_loss = evaluate(model, src_field, trg_field, valid_iterator,␣
 ↪criterion)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), MODEL_PATH)
    print(f'Epoch: {epoch + 1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.
 ↪3f}')
```