

实验三 机器翻译

一、实验任务

要求：

使用Seq2Seq方法，完成英语->德语的翻译任务，最终BLEU值不低于0.25

结果：

在测试集上的BLEU值达到0.41

使用的主要技术有：

文本预处理：调用torchtext模块、fasttext

模型：Seq2Seq模型、双向GRU、Embedding、Dropout、Attention机制

测试：bleu

二、实验环境

需要使用谷歌浏览器，登陆有效的谷歌账号和能够访问colab网站。

1.使用谷歌硬盘和colab pro结合的训练平台

在笔记本设置中选择GPU和高RAM运行时，保证训练时有充足的大规模并行算力和内存

笔记本设置

硬件加速器

GPU 

要充分利用 Colab Pro，请尽量避免使用 GPU，除非确实有需要。
[了解详情](#)

运行时规格

高 RAM 

☐ 保存此笔记本时忽略代码单元格输出项

取消

保存

2.设置依赖文件路径

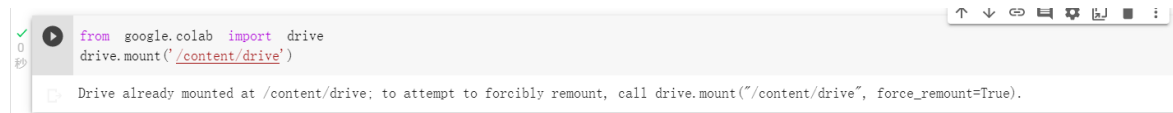
为了保证colab可以访问需要用到的外部文件（主要是训练集、验证集和测试集文件），需要挂载谷歌云里的文件夹。

用colab打开我们的.ipynb文件，运行下面的代码执行单元：

```
from google.colab import drive
drive.mount('/content/drive')
```

在输出栏会出现一个链接和一个输入框，进入链接复制授权码，将其粘贴回输入框，点回车键完成授权步骤。

文件挂载成功：



3.修改环境配置

colab提供某一版本的tensorflow、keras或Pytorch，正常情况下可以直接使用。如果想要使用不同的版本（会有库的更新），可以在.ipynb文件中输入命令。

比如在这次实验中我想要使用0.8.0版本的torchtext，可以运行代码：

```
!pip install torchtext==0.8.0
```

colab会卸载原有的0.11.0版本的torchtext，再安装0.8.0的torchtext



注意要先修改版本再在后面的代码中import，否则需要重新启动笔记本使修改生效。（会丢失已运行的内容）

三、实验过程

1.数据集预处理

训练集使用news-commentary-v11.de-en.en和news-commentary-v11.de-en.de的前10000条。

DOWNLOAD

- Parallel data:

File	Size	CS-EN	DE-EN	FI-EN	RO-EN	RU-EN	TR-EN	Notes
Europarl v7	628MB	✓	✓					same as previous year, corpus home page
Europarl v8	215MB			✓	✓			ro-en is new for this year, corpus home page
Common Crawl corpus	876MB	✓	✓			✓		Same as last year
News Commentary v11	72MB	✓	✓			✓		updated
CzEng 1.6pre	3.1GB	✓						New for 2016. Register and download CzEng 1.6pre.
Yandex Corpus	121MB					✓		ru-en
Wiki Headlines	9.1MB			✓		✓		Provided by CMU..
SETIMES2	?? MB				✓		✓	Distributed by OPUS

验证集使用newstest2010.en和newstest2010.de。

名称	修改日期	类型	大小
newstest2009-src.fr.sgm	2009/12/3 16:27	SGM 文件	449 KB
newstest2009-src.hu.sgm	2009/12/3 16:27	SGM 文件	437 KB
newstest2009-src.it.sgm	2009/12/3 16:27	SGM 文件	424 KB
newstest2009-src.xx.sgm	2009/12/3 16:27	SGM 文件	405 KB
newstest2010.cs	2010/3/5 18:00	CS 文件	329 KB
newstest2010.de	2010/3/5 18:00	DE 文件	375 KB
newstest2010.en	2010/3/5 18:00	EN 文件	321 KB
newstest2010.es	2010/3/5 18:00	ES 文件	356 KB
newstest2010.fr	2010/3/5 18:00	FR 文件	375 KB
newstest2010-ref.cs.sgm	2010/3/5 17:58	SGM 文件	398 KB
newstest2010-ref.de.sgm	2010/3/5 17:58	SGM 文件	443 KB
newstest2010-ref.en.sgm	2010/3/5 17:58	SGM 文件	390 KB
newstest2010-ref.es.sgm	2010/3/5 17:59	SGM 文件	425 KB
newstest2010-ref.fr.sgm	2010/3/5 17:59	SGM 文件	444 KB
newstest2010-src.cs.sgm	2010/3/5 17:53	SGM 文件	396 KB
newstest2010-src.de.sgm	2010/3/5 17:53	SGM 文件	442 KB
newstest2010-src.en.sgm	2010/3/5 17:53	SGM 文件	389 KB
newstest2010-src.es.sgm	2010/3/5 17:53	SGM 文件	423 KB
newstest2010-src.fr.sgm	2010/3/5 17:53	SGM 文件	442 KB
newstest2011.cs	2011/12/10 3:55	CS 文件	396 KB
newstest2011.de	2011/12/10 3:55	DE 文件	441 KB
newstest2011.en	2011/12/10 3:55	EN 文件	388 KB
newstest2011.es	2011/12/10 3:55	ES 文件	424 KB
newstest2011.fr	2011/12/10 3:55	FR 文件	446 KB

测试集使用newstest2016-ende-src.en.sgm和newstest2016-ende-ref.de.sgm。

这六个文件全部存放在谷歌硬盘的"/content/drive/MyDrive/AI/project3/dataset/"文件夹下，考虑到提交文件太大对助教来说可能比较麻烦，数据集没有和我的代码一同提交。

使用的训练集有二十多万条语句，在代码调试的过程中，我希望能首先使用规模较小的数据集，确定模型后再换回完整的数据集。

因此数据集预处理部分包括生成规模较小的训练集：

```

en_train = []
with open("/content/drive/MyDrive/AI/project3/dataset/news-commentary-v11.de-
en.en", "r") as f5:
    data4 = f5.read().split("\n")
    en_train = data4[0:10000]

f5.close()

with open("/content/drive/MyDrive/AI/project3/dataset/en_train.txt", "w") as f6:
    for line in en_train:
        f6.write(line+"\n")
f6.close()

```

```

de_train = []
with open("/content/drive/MyDrive/AI/project3/dataset/news-commentary-v11.de-
en.de", "r") as f7:
    data4 = f7.read().split("\n")
    de_train = data4[0:10000]

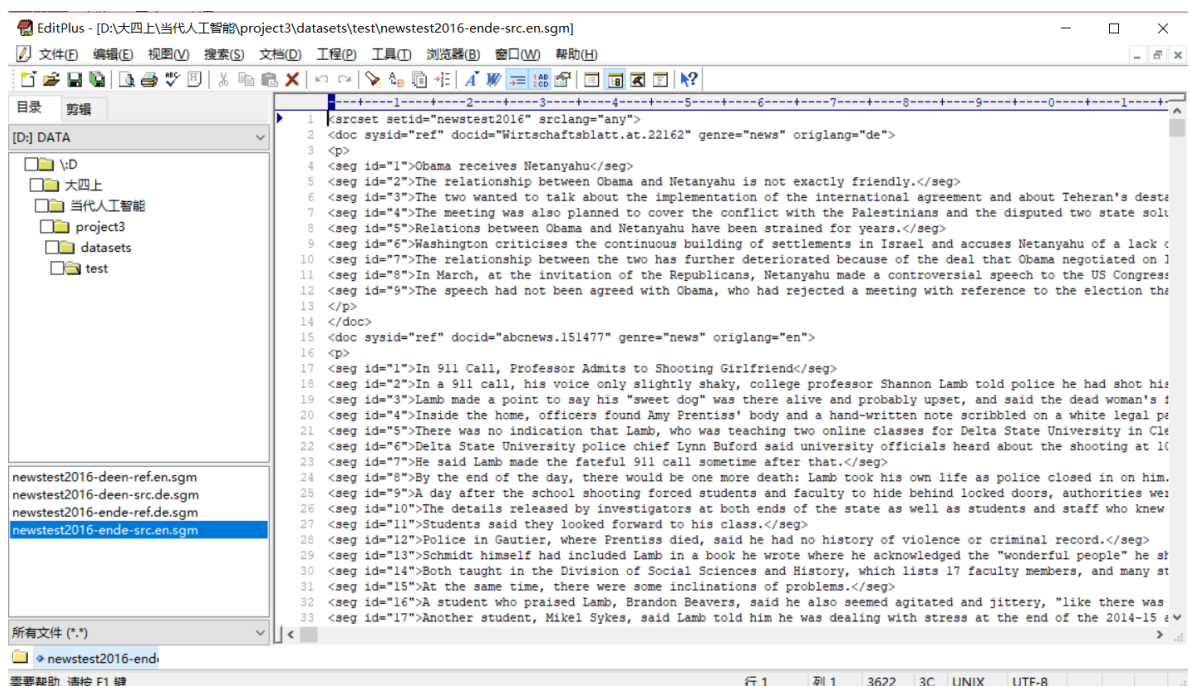
f7.close()

with open("/content/drive/MyDrive/AI/project3/dataset/de_train.txt", "w") as f8:
    for line in de_train:
        f8.write(line+"\n")
f8.close()

```

en_train.txt和de_train.txt中保存着原数据集的前10000条句子。

测试集中的英语和德语语料只有.sgm格式的文件，我尝试打开发现它的内容类似超文本标记语言，直接读取会读到标签。



所幸的是所有我们想要的内容都存放在标签中，因此我使用了正则表达式抽取这部分句子，保存在en_test.txt和de_test.txt文件中。

```

eng_test = []
with open("/content/drive/MyDrive/AI/project3/dataset/newstest2016-ende-
src.en.sgm", "r") as f1:
    data3 = f1.read().split("\n")
    for i in range(len(data3)):
        temp = re.findall(r'<seg.*?>(.*?)</seg>', data3[i], re.S|re.M)
        if not temp:
            continue
        else:
            eng_test.append(temp[0])
f1.close()

with open("/content/drive/MyDrive/AI/project3/dataset/en_test.txt", "w") as f2:
    for line in eng_test:
        f2.write(line+"\n")
f2.close()

```

```












de_test = []
with open("/content/drive/MyDrive/AI/project3/dataset/newstest2016-ende-
ref.de.sgm", "r") as f3:
    data4 = f3.read().split("\n")
    for i in range(len(data4)):
        temp = re.findall(r'<seg.*?>(.*?)</seg>', data4[i], re.S|re.M)
        if not temp:
            continue
        else:
            de_test.append(temp[0])
f3.close()

with open("/content/drive/MyDrive/AI/project3/dataset/de_test.txt", "w") as f4:
    for line in de_test:
        f4.write(line+"\n")
f4.close()

```

数据集预处理后生成的四个文件：

My Drive > AI > project3 > dataset ▾

Name ↑	Owner	Last modified	File size
 de_core_news_md-3.2.0.tar.gz	me	Nov 20, 2021	46.5 MB
 de_test.txt	me	Nov 20, 2021	371 KB
 de_train.txt	me	2:35 PM	1.6 MB
 en_core_web_md-3.2.0.tar.gz	me	Nov 20, 2021	43.6 MB
 en_test.txt	me	Nov 20, 2021	330 KB
 en_train.txt	me	2:35 PM	1.3 MB
 europarl-v7.en-de.de	me	Nov 18, 2021	1.6 MB
 europarl-v7.en-de.en	me	Nov 18, 2021	1.5 MB
 news-commentary-v11.de-en.de	me	Jan 6, 2016	39.4 MB
 news-commentary-v11.de-en.en	me	Jan 6, 2016	33.4 MB
 newstest2010.de	me	Mar 5, 2010	374 KB

2.输入数据预处理

主要使用了Torchtext

在主程序中运行：

```
data = DataPipeline(batch_size=16)
en_vocab = data.en_vocab
de_vocab = data.de_vocab
train_loader = data.train_loader
valid_loader = data.valid_loader
test_loader = data.test_loader
```

DataPipeline类专门用于输入数据预处理，返回后可以得到英文词典、德文词典，同时可以得到处理好的训练数据、验证数据和测试数据。

在定义DataPipeline类之前，下载spacy工具包的模型放在dataset文件夹下，英文和德文的模型压缩包分别是en_core_web_md-3.2.0.tar.gz和de_core_news_md-3.2.0.tar.gz。可以在官网下载<https://spacy.io/models>

spacy 提供了许多不同的模型，模型中包含了语言的信息词汇表，预训练的词向量，语法和实体。这里我们仅仅在torchtext中调用这一工具构建tokenizer。

下载压缩包后执行代码，加载模型：

```
!pip install '/content/drive/MyDrive/AI/project3/dataset/en_core_web_md-3.2.0.tar.gz'
!pip install '/content/drive/MyDrive/AI/project3/dataset/de_core_news_md-3.2.0.tar.gz'
```

运行下面的代码成功可以说明模型已成功安装：

```
nlp1 = spacy.load('en_core_web_md')
nlp2 = spacy.load('de_core_news_md')
```

下面进入DataPipeline类的构建，除了实例化该类使用的init函数外，还包括其他四个函数，主要负责构建英文词表、构建德文词表、从数据构建tensor以及将数据划分为batch。

init函数负责在实例化该类时做初始化工作，train_datasets、val_datasets、test_datasets分别存储着训练集、验证集、测试集的文件路径，接着调用get_tokenizer函数，利用spacy工具包获得英文和德文的分词器，将训练集文件路径和分词器传入该类的build_en_vocab函数和build_de_vocab函数，得到英文和德文的词表。接着对训练集、验证集、测试集分别用create_tensor做处理，再将返回值交给DataLoader处理。在训练模型时，我们通常希望以“小批量”的形式传递样本，在每个epoch重新打乱数据以减少模型过度拟合。

DataLoader 是一个迭代器，它为我们抽象了这种复杂性，调用该API即可。

```
def __init__(self, batch_size=64):
    train_datasets =
    ["/content/drive/MyDrive/AI/project3/dataset/en_train.txt",
     "/content/drive/MyDrive/AI/project3/dataset/de_train.txt"]
    val_datasets =
    ["/content/drive/MyDrive/AI/project3/dataset/newstest2010.en",
     "/content/drive/MyDrive/AI/project3/dataset/newstest2010.de"]
    test_datasets =
    ["/content/drive/MyDrive/AI/project3/dataset/en_test.txt",
     "/content/drive/MyDrive/AI/project3/dataset/de_test.txt"]
    self.en_tokenizer = get_tokenizer('spacy', language='en_core_web_md')
    self.de_tokenizer = get_tokenizer('spacy', language='de_core_news_md')
    self.en_vocab = self.build_en_vocab(train_datasets[0],
    self.en_tokenizer)
    self.de_vocab = self.build_de_vocab(train_datasets[1],
    self.de_tokenizer)

    train_data = self.create_tensor(train_datasets)
    val_data = self.create_tensor(val_datasets)
    test_data = self.create_tensor(test_datasets)

    self.train_loader = DataLoader(train_data, batch_size=batch_size,
    shuffle=True, collate_fn=self.create_batch)
    self.valid_loader = DataLoader(val_data, batch_size=batch_size,
    shuffle=True, collate_fn=self.create_batch)
    self.test_loader = DataLoader(test_data, batch_size=batch_size,
    shuffle=True, collate_fn=self.create_batch)
```

在build_en_vocab函数中，首先实例化一个collections模块的counter工具用于支持便捷和快速地计数。Torchtext的Vocab可以快速构建当前 corpus 的词典，注意这个API需要传入的参数即可。'unk'表示未在词典中的词，'pad'表示补全字符，'bos'是句子起始标识符，'eos'是句子结束标识符。使用FastText预训练的词向量，运行到这里时程序会自动下载约7个G的文件。torch.zeros用于返回全为0的向量，再对它进行维度扩充和拼接，最后返回处理好的英文词典。

```

def build_en_vocab(self, filepath, tokenizer):
    counter = Counter()
    with io.open(filepath, encoding="utf8") as f1:
        for temp in f1:
            counter.update(tokenizer(temp))
    vocab = Vocab(counter, specials=['<unk>', '<pad>', '<bos>', '<eos>'],
    vectors=FastText(language='en', max_vectors=1000_000))
    zero_vec = torch.zeros(vocab.vectors.size()[0])
    zero_vec = torch.unsqueeze(zero_vec, dim=1)
    vocab.vectors = torch.cat((zero_vec, zero_vec, zero_vec, vocab.vectors),
    dim=1)
    vocab.vectors[vocab["<pad>"]][0] = 1
    vocab.vectors[vocab["<bos>"]][1] = 1
    vocab.vectors[vocab["<eos>"]][2] = 1
    return vocab

```

在build_de_vocab函数中构建德文词典：

```

def build_de_vocab(self, filepath, tokenizer):
    counter = Counter()
    with io.open(filepath, encoding="utf8") as f:
        for temp in f:
            counter.update(tokenizer(temp))
    return Vocab(counter, specials=['<unk>', '<pad>', '<bos>', '<eos>'])

```

在create_tensor中将数据转化为张量类型。首先创建迭代器，在一对句子中，对每一个单词使用之前生成的词典和分词器做处理，最后转成对应的张量，并成对放在列表中返回。

```

def create_tensor(self, filepaths):
    raw_en_iter = iter(io.open(filepaths[0], encoding="utf8"))
    raw_de_iter = iter(io.open(filepaths[1], encoding="utf8"))
    data = []
    #cnt = 0
    for (raw_en, raw_de) in zip(raw_en_iter, raw_de_iter):
        #cnt += 1
        en_tensor_ = torch.tensor([self.en_vocab[token] for token in
        self.en_tokenizer(raw_en)[: -1]], dtype=torch.long)
        de_tensor_ = torch.tensor([self.de_vocab[token] for token in
        self.de_tokenizer(raw_de)[: -1]], dtype=torch.long)
        data.append((en_tensor_, de_tensor_))
        #if cnt >= 20000:
        #break
    return data

```

create_batch函数用于传入Dataloader，负责对数据集中的每一个句子添加开始标识符和结束标识符，并且用pad标识符补全，使大小一致。


```
def create_batch(self, data_batch):
    en_batch, de_batch, = [], []
    for (en_item, de_item) in data_batch:
        en_batch.append(torch.cat([torch.tensor([self.en_vocab['<bos>']]),
en_item, torch.tensor([self.en_vocab['<eos>']])], dim=0))
        de_batch.append(torch.cat([torch.tensor([self.de_vocab['<bos>']]),
de_item, torch.tensor([self.de_vocab['<eos>']])], dim=0))
    en_batch = pad_sequence(en_batch, padding_value=self.en_vocab['<pad>'])
    de_batch = pad_sequence(de_batch, padding_value=self.de_vocab['<pad>'])
    return en_batch, de_batch
```

3.构建模型

在主函数中运行：第一个部分设置参数，接着将参数传入，实例化Encoder类、Attention类、Decoder类。设置在有GPU时使用GPU，最后将encoder和decoder传入Seq2Seq构建模型。

优化器使用Adam，学习率设置为1e-3。

损失函数使用交叉熵损失。

```
dim_input = len(en_vocab)
dim_output = len(de_vocab)
dim_en_embedding = en_vocab.vectors.size()[1]
dim_de_embedding = 64
dim_en_hidden = 128
dim_de_hidden = 128
dim_attention = 32
dropout_en = 0.5
dropout_de = 0.5

enc = Encoder(dim_input, dim_en_embedding, dim_en_hidden, dim_de_hidden,
dropout_en)
attn = Attention(dim_en_hidden, dim_de_hidden, dim_attention)
dec = Decoder(dim_output, dim_de_embedding, dim_en_hidden, dim_de_hidden,
dropout_de, attn)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Seq2Seq(enc, dec, device).to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss(ignore_index=de_vocab.stoi['<pad>'])
```

Encoder类的实现：首先是第一层，使用嵌入层embedding，接着使用双向GRU，然后设置全连接层，最后设置Dropout，使神经元在每次迭代训练时会随机有 50% 的可能性被丢弃，不参与训练。

```
class Encoder(nn.Module):
    def __init__(self, dim_input: int, dim_en_embedding: int, dim_en_hidden:
int, dim_de_hidden: int, dropout: float):
        super().__init__()

        self.dim_input = dim_input
        self.dim_en_embedding = dim_en_embedding
        self.dim_en_hidden = dim_en_hidden
        self.dim_de_hidden = dim_de_hidden
```

```

        self.dropout = dropout

        self.embedding = nn.Embedding(dim_input,
dim_en_embedding).from_pretrained(en_vocab.vectors, freeze=True)
        self.rnn = nn.GRU(dim_en_embedding, dim_en_hidden, num_layers=1,
bidirectional=True)
        self.fc = nn.Linear(dim_en_hidden * 2, dim_de_hidden)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src: Tensor) -> Tuple[Tensor]:
        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

```

Decoder类:

首先设置Embedding, 接着加入GRU、全连接 (在这里使用Attention机制), 以及Dropout。

```

class Decoder(nn.Module):
    def __init__(self, dim_output: int, dim_de_embedding: int, dim_en_hidden:
int, dim_de_hidden: int, dropout: float, attention: nn.Module):
        super().__init__()

        self.dim_de_embedding = dim_de_embedding
        self.dim_en_hidden = dim_en_hidden
        self.dim_de_hidden = dim_de_hidden
        self.dim_output = dim_output
        self.dropout = dropout
        self.attention = attention

        self.embedding = nn.Embedding(dim_output, dim_de_embedding)
        self.rnn = nn.GRU((dim_en_hidden * 2) + dim_de_embedding, dim_de_hidden)
        self.out = nn.Linear(self.attention.attn_in + dim_de_embedding,
dim_output)
        self.dropout = nn.Dropout(dropout)

    def _weighted_encoder_rep(self, decoder_hidden: Tensor, encoder_outputs:
Tensor) -> Tensor:
        a = self.attention(decoder_hidden, encoder_outputs)
        a = a.unsqueeze(1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        weighted_encoder_rep = torch.bmm(a, encoder_outputs)
        weighted_encoder_rep = weighted_encoder_rep.permute(1, 0, 2)
        return weighted_encoder_rep

    def forward(self, input: Tensor, decoder_hidden: Tensor, encoder_outputs:
Tensor) -> Tuple[Tensor]:
        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))
        weighted_encoder_rep = self._weighted_encoder_rep(decoder_hidden,
encoder_outputs)

        rnn_input = torch.cat((embedded, weighted_encoder_rep), dim=2)

```

```

        output, decoder_hidden = self.rnn(rnn_input,
decoder_hidden.unsqueeze(0))

        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        weighted_encoder_rep = weighted_encoder_rep.squeeze(0)

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

        return output, decoder_hidden.squeeze(0)

```

往Encoder-Decoder模型中引入Attention，使模型的"注意力"集中在重点内容上，解决信息丢失问题（将Encoder的输出编码成一个向量的序列）：

```

class Attention(nn.Module):
    def __init__(self, dim_en_hidden: int, dim_de_hidden: int, dim_attention:
int):
        super().__init__()

        self.dim_en_hidden = dim_en_hidden
        self.dim_de_hidden = dim_de_hidden
        self.attn_in = (dim_en_hidden * 2) + dim_de_hidden
        self.attn = nn.Linear(self.attn_in, dim_attention)

    def forward(self, decoder_hidden: Tensor, encoder_outputs: Tensor) ->
Tensor:
        src_len = encoder_outputs.shape[0]
        repeated_decoder_hidden = decoder_hidden.unsqueeze(1).repeat(1, src_len,
1)

        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        energy = torch.tanh(self.attn(torch.cat((repeated_decoder_hidden,
encoder_outputs), dim=2)))

        attention = torch.sum(energy, dim=2)
        return F.softmax(attention, dim=1)

```

最后结合Encoder和Decoder，Seq2Seq框架就搭建完毕了。

4.训练模型和测试模型

定义train函数和evaluate函数，用于模型的训练和测试。evaluate函数调用bleu函数，计算结果的评分。

将epoch设置为20，调用train()训练模型，调用calculate_time()计算训练一个epoch所需的时间，同时在每一个epoch都调用evaluate()，使用验证集并输出验证集的Bleu。

```

N_EPOCHS = 20
CLIP = 1

print(f'The model has {sum(p.numel() for p in model.parameters() if
p.requires_grad):,} trainable parameters')

```

```

for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss = train(model, train_loader, optimizer, criterion, CLIP)
    valid_loss, valid_bleu = evaluate(model, valid_loader, criterion)

    end_time = time.time()
    epoch_mins, epoch_secs = calculate_time(start_time, end_time)

    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'\tVal. Loss: {valid_loss:.3f} | Val. PPL:
{math.exp(valid_loss):7.3f} | Val. Bleu: {round(100*valid_bleu, 2)}')

test_loss, test_bleu = evaluate(model, test_loader, criterion)

```

模型训练:

```

def train(model: nn.Module, iterator: torch.utils.data.DataLoader, optimizer:
optim.Optimizer, criterion: nn.Module, clip: float):
    model.train()
    epoch_loss = 0

    for _, (src, trg) in enumerate(iterator):
        src, trg = src.to(device), trg.to(device)
        optimizer.zero_grad()
        output = model(src, trg)
        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 calculate_time(start_time: float, end_time: float):
    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

```

模型评估:

```
def evaluate(model: nn.Module, iterator: torch.utils.data.DataLoader, criterion:
nn.Module):
    model.eval()
    epoch_loss, epoch_bleu = 0, 0
    with torch.no_grad():
        for _, (src, trg) in enumerate(iterator):
            src, trg = src.to(device), trg.to(device)
            output = model(src, trg, 0) # turn off teacher forcing
            epoch_bleu += bleu(output[1:, :, :], trg[1:, :], de_vocab, device)
            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), epoch_bleu / len(iterator)
```

```
def bleu(output, target, vocab, device):
    _, output_indices = torch.max(output, 2)
    eos_pad = torch.full((1, output.size()[1]), vocab.stoi["<eos>"],
device=device)
    output_indices = torch.cat((output_indices, eos_pad), 0)
    output_indices = output_indices.transpose(1, 0)
    target = target.transpose(1, 0)

    output_str = [[vocab.itos[x] for x in y] for y in output_indices]
    output_str = [x[:x.index("<eos>")] for x in output_str]
    target_str = [[vocab.itos[x] for x in y] for y in target]
    target_str = [x[:x.index("<eos>")] for x in target_str]
    return bleu_score(output_str, target_str)
```

训练时的输出:

```
➡ The model has 15,110,013 trainable parameters
Epoch: 01 | Time: 2m 10s
    Train Loss: 7.251 | Train PPL: 1409.779
    Val. Loss: 7.599 | Val. PPL: 1996.950 | Val. Bleu: 0.0
Epoch: 02 | Time: 2m 12s
    Train Loss: 6.629 | Train PPL: 756.478
    Val. Loss: 7.623 | Val. PPL: 2044.602 | Val. Bleu: 0.05
Epoch: 03 | Time: 2m 13s
    Train Loss: 6.144 | Train PPL: 465.779
    Val. Loss: 7.713 | Val. PPL: 2236.797 | Val. Bleu: 0.07
Epoch: 04 | Time: 2m 12s
    Train Loss: 5.706 | Train PPL: 300.748
    Val. Loss: 7.881 | Val. PPL: 2647.570 | Val. Bleu: 0.06
Epoch: 05 | Time: 2m 12s
    Train Loss: 5.299 | Train PPL: 200.039
    Val. Loss: 7.982 | Val. PPL: 2926.870 | Val. Bleu: 0.1
```

```

Epoch: 06 | Time: 2m 12s
    Train Loss: 4.945 | Train PPL: 140.478
    Val. Loss: 8.123 | Val. PPL: 3372.609 | Val. Bleu: 0.23
Epoch: 07 | Time: 2m 13s
    Train Loss: 4.649 | Train PPL: 104.526
    Val. Loss: 8.287 | Val. PPL: 3970.586 | Val. Bleu: 0.19
Epoch: 08 | Time: 2m 12s
    Train Loss: 4.410 | Train PPL: 82.256
    Val. Loss: 8.433 | Val. PPL: 4596.434 | Val. Bleu: 0.19
Epoch: 09 | Time: 2m 13s
    Train Loss: 4.218 | Train PPL: 67.876
    Val. Loss: 8.523 | Val. PPL: 5026.748 | Val. Bleu: 0.31
Epoch: 10 | Time: 2m 12s
    Train Loss: 4.079 | Train PPL: 59.069
    Val. Loss: 8.609 | Val. PPL: 5478.959 | Val. Bleu: 0.3

Epoch: 11 | Time: 2m 13s
    Train Loss: 3.960 | Train PPL: 52.461
    Val. Loss: 8.692 | Val. PPL: 5955.270 | Val. Bleu: 0.35
Epoch: 12 | Time: 2m 13s
    Train Loss: 3.851 | Train PPL: 47.026
    Val. Loss: 8.773 | Val. PPL: 6460.577 | Val. Bleu: 0.37
Epoch: 13 | Time: 2m 12s
    Train Loss: 3.757 | Train PPL: 42.807
    Val. Loss: 8.847 | Val. PPL: 6953.203 | Val. Bleu: 0.37
Epoch: 14 | Time: 2m 13s
    Train Loss: 3.674 | Train PPL: 39.423
    Val. Loss: 8.927 | Val. PPL: 7535.376 | Val. Bleu: 0.4
Epoch: 15 | Time: 2m 12s
    Train Loss: 3.607 | Train PPL: 36.841
    Val. Loss: 8.942 | Val. PPL: 7648.681 | Val. Bleu: 0.4

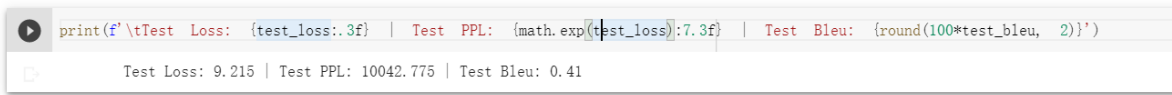
Epoch: 16 | Time: 2m 12s
    Train Loss: 3.534 | Train PPL: 34.245
    Val. Loss: 8.985 | Val. PPL: 7984.730 | Val. Bleu: 0.38
Epoch: 17 | Time: 2m 12s
    Train Loss: 3.453 | Train PPL: 31.598
    Val. Loss: 9.090 | Val. PPL: 8868.326 | Val. Bleu: 0.5
Epoch: 18 | Time: 2m 13s
    Train Loss: 3.413 | Train PPL: 30.361
    Val. Loss: 9.102 | Val. PPL: 8971.793 | Val. Bleu: 0.41
Epoch: 19 | Time: 2m 12s
    Train Loss: 3.352 | Train PPL: 28.565
    Val. Loss: 9.158 | Val. PPL: 9491.161 | Val. Bleu: 0.54
Epoch: 20 | Time: 2m 13s
    Train Loss: 3.291 | Train PPL: 26.876
    Val. Loss: 9.215 | Val. PPL: 10043.029 | Val. Bleu: 0.52

```

可以看到，在最好情况下，验证集上的Bleu值达到了0.54。

最后在测试集上的效果：bleu值为0.41

```
test_loss, test_bleu = evaluate(model, test_loader, criterion)
print(f'\tTest Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |
Test Bleu: {round(100*test_bleu, 2)}')
```



5.关于使用完整的数据集

完整数据集的句子数高达二十多万条，训练一个epoch需要数个小时，我尝试后发现colab在长时间运行时很容易断连。

目前采用的预防断连的方式是在页面的代码中添加自动点击命令。

打开控制台，使下列javascript代码运行：

```
function ClickConnect(){
    colab.config
    console.log("Connnect Clicked - Start");
    document.querySelector("#top-toolbar > colab-connect-
button").shadowRoot.querySelector("#connect").click();
    console.log("Connnect Clicked - End");
};
setInterval(ClickConnect, 60000)
```

这种方式是有效的，但我发现colab仍然会出现"验证您不是机器"，让我选出带红绿灯的图片之类的。这时我不在电脑前，colab会自动断连，之前代码运行的结果就全部消失了。

因此我觉得完整的数据集很难实现，不过这之后有尝试把训练数据集数量提升到两万条，训练4个epoch时的结果如下图所示：

```
[ ] N_EPOCHS = 4
    CLIP = 1

    for epoch in range(N_EPOCHS):
        start_time = time.time()
        train_loss = train(model, train_loader, optimizer, criterion, CLIP)
        valid_loss, valid_bleu = evaluate(model, valid_loader, criterion)

        end_time = time.time()
        epoch_mins, epoch_secs = epoch_time(start_time, end_time)

        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'\tVal. Loss: {valid_loss:.3f} | Val. PPL: {math.exp(valid_loss):7.3f} | Val. Bleu: {round(1

Epoch: 01 | Time: 7m 33s
    Train Loss: 5.609 | Train PPL: 272.882
    Val. Loss: 7.581 | Val. PPL: 1959.946 | Val. Bleu: 0.19
Epoch: 02 | Time: 7m 32s
    Train Loss: 5.187 | Train PPL: 179.009
    Val. Loss: 7.776 | Val. PPL: 2382.012 | Val. Bleu: 0.34
Epoch: 03 | Time: 7m 32s
    Train Loss: 4.833 | Train PPL: 125.596
    Val. Loss: 7.965 | Val. PPL: 2877.473 | Val. Bleu: 0.36
Epoch: 04 | Time: 7m 34s
    Train Loss: 4.558 | Train PPL: 95.368
    Val. Loss: 8.098 | Val. PPL: 3286.948 | Val. Bleu: 0.36
```

一轮的训练时间提升到7分钟左右，但是训练久了colab仍然容易断连，我也不太有时间整天守着这个项目训练，而且我的电脑本身不带GPU，几乎无法在本地跑（cpu太慢且占用太高内存电脑会很卡，有别的作业要写）。因此一万条数据集训练的结果就作为最终结果提交了。