基于SGNS的词向量学习算法

刘帅 2020212267

实验简介

本次实验利用paddle的skipgram模型，在text8数据集进行文本相似度训练，并于wordsim353数据集进行推理。

本次实验的收获及贡献点如下：

- 了解了skipgram的算法和embedding流程
- 熟悉了paddle框架，并自主实现了dataloader的训练过程
- 规范了data pipeline，以pytorch的格式进行dataloader的读取（即利用paddle.dataset类进行实现）
- 由于数据量较为庞大，因此在数据读取、shuffle、训练loss的过程中都进行了可视化监控。
- 实现了skipgram算法的分布式训练（单机多卡），进一步加快了数据存储和训练效率。
- 设定了训练和模型保存机制，从而防止训练过拟合。

目录组织形式

```
|-- 2020212267.txt #wordsim353_agreed的测试结果
|-- comparison.jpg #测试集结果比较
|-- ddp_skip-gram.py #gpu并行代码#更改训练流程
|-- log    #tensorboard的可视化代码
|-- skipgram.py #原始的sg算法
|-- testsg.py #模型加载和测试过程
|-- train.sh  #并行训练的脚本
|-- visualize.py #可视化代码
```

一、实验环境

由于后续有分布式训练的部署，且为了方便调试，更加熟悉paddle框架，本次实验在自己的集群上进行：

```
Sun Apr 30 14:08:44 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.129.06   Driver Version: 470.129.06   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  On   | 00000000:65:01.0 Off |                    0 |
| N/A   39C    P0    95W / 400W |  80333MiB / 81251MiB |     25%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-SXM...  On   | 00000000:67:01.0 Off |                    0 |
| N/A   58C    P0   161W / 400W |  52199MiB / 81251MiB |     85%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
```

paddle环境部署

根据paddle文档内容配置paddle环境

```
python -m pip install paddlepaddle-gpu==2.4.2.post112 -f
https://www.paddlepaddle.org.cn/whl/linux/mkl/avx/stable.html
```

二、代码分析

2.1 导入相关package

由于个人对paddle内置的tensorboard方法不熟悉，因此更加偏向利用torch当中的tensorboard，由于paddle中某些组件的初始化版本与torch不同，可能会产生冲突，因此我们先引入torch，从而利于后面tensorboard的初始化。

```
import io
import os
import sys
import torch #引入torch只是为了利于后面tensorboard的初始化,由于torch和paddle内置的初始化版本不同，因此后调入torch会报错
import requests
from collections import OrderedDict
import math
import random
import numpy as np
import paddle
import paddle.fluid as fluid
import tqdm
from paddle.fluid.dygraph.nn import Embedding
from torch.utils.tensorboard import SummaryWriter #用tensorboard对训练过程进行记录
writer=SummaryWriter()
```

2.2 数据前处理

2.2.1 下载数据

```
def download():
    #可以从百度云服务器下载一些开源数据集（dataset.bj.bcebos.com）
    corpus_url = "https://dataset.bj.bcebos.com/word2vec/text8.txt"
    #使用python的requests包下载数据集到本地
    web_request = requests.get(corpus_url)
    corpus = web_request.content
    #把下载后的文件存储在当前目录的text8.txt文件内
    with open("./text8.txt", "wb") as f:
        f.write(corpus)
    f.close()
download()
```

2.2.2 读取数据

```
def load_text8():
    with open("./text8.txt", "r") as f:
        corpus = f.read().strip("\n")
    f.close()
    return corpus
corpus = load_text8()
```

2.2.3 语料预处理

将数据全部转换为小写，并且以每个单词的形式进行切分

```
def data_preprocess(corpus):
    corpus = corpus.strip().lower()
    corpus = corpus.split(" ")
    return corpus
```

2.2.4 构造字典

对于词典中的词，根据其出现次序重新排列，并且构造三个映射，分别为word-id，word-freq，id-word的映射。

```
def build_dict(corpus):
    #首先统计每个不同词的频率（出现的次数），使用一个词典记录
    word_freq_dict = dict()
    for word in corpus:
        if word not in word_freq_dict:
            word_freq_dict[word] = 0
        word_freq_dict[word] += 1
    #将这个词典中的词，按照出现次数排序，出现次数越高，排序越靠前
    word_freq_dict = sorted(word_freq_dict.items(), key = lambda x:x[1], reverse =
True)
    word2id_dict = dict()
    word2id_freq = dict()
    id2word_dict = dict()
    for word, freq in word_freq_dict:
        curr_id = len(word2id_dict)
        word2id_dict[word] = curr_id
        word2id_freq[curr_id] = freq #以id为标识，把标识加到freq中
        id2word_dict[curr_id] = word #
    return word2id_freq, word2id_dict, id2word_dict
word2id_freq, word2id_dict, id2word_dict = build_dict(corpus)
```

2.2.5 转换为id序列

将每个位置的词用id来表示，作为这个词的token，便于后续送入网络训练。

```
def convert_corpus_to_id(corpus, word2id_dict):
  corpus = [word2id_dict[word] for word in corpus] #一个text中每个位置的词用id来表示
  return corpus
corpus = convert_corpus_to_id(corpus, word2id_dict)
```

2.2.6 二次采样过滤数据

随机清除其中的一些词，如果频率很大，则该数据很可能被遗弃，这样的操作防止了数据分布的long-tail。由于某些常见词（且通常这些常见词是没有意义的，例如the a 等，他们对于数据的关联有着不利作用）

```
def subsampling(corpus, word2id_freq):
    discard=lambda word_id:random.uniform(0, 1) < 1 - math.sqrt(1e-4 /
word2id_freq[word_id] * len(corpus))
    corpus = [word for word in corpus if not discard(word)] #随机清除其中的一些词，如果频率很大，则很可能被遗弃
    return corpus
corpus = subsampling(corpus, word2id_freq)
```

2.3 数据集

2.3.1 构建数据集

```
def build_data(corpus, word2id_dict, word2id_freq, max_window_size = 3,
negative_sample_num = 4):
    #使用一个list存储处理好的数据
    dataset = []
    #从左到右，开始枚举每个中心点的位置
    for center_word_idx in tqdm.tqdm(range(len(corpus))): #可视化一下
        #以max_window_size为上限，随机采样一个window_size，这样会使得训练更加稳定
        window_size = random.randint(1, max_window_size)
        #当前的中心词就是center_word_idx所指向的词
        center_word = corpus[center_word_idx] #第i个词就是此时的centerword
        #以当前中心词为中心，左右两侧在window_size内的词都可以看成是正样本
        positive_word_range = (max(0, center_word_idx - window_size),
min(len(corpus) - 1, center_word_idx + window_size))
        positive_word_candidates = [corpus[idx] for idx in
range(positive_word_range[0], positive_word_range[1]+1) if idx != center_word_idx]
        #中心词附近的正样本
        #对于每个正样本来说，随机采样negative_sample_num个负样本，用于训练
        for positive_word in positive_word_candidates:
            #首先把（中心词，正样本，label=1）的三元组数据放入dataset中，
            #这里label=1表示这个样本是个正样本
            dataset.append((center_word, positive_word, 1))
            #开始负采样
            i = 0
            while i < negative_sample_num:
                negative_word_candidate = random.randint(0, vocab_size-1)

                if negative_word_candidate not in positive_word_candidates:
                    #把（中心词，正样本，label=0）的三元组数据放入dataset中，
                    #这里label=0表示这个样本是个负样本
```

```
                    dataset.append((center_word, negative_word_candidate, 0))
                    i += 1
    return dataset
dataset = build_data(corpus, word2id_dict, word2id_freq)
```

关于窗口大小和负样本的采样数量的分析：

窗口大小：如果文本单词之间的语义关系较为稠密，则需要较小的较小的window_size，反之，则需要较大的窗口大小。我们可以考虑动态调整窗口大小的方法进一步防止数据陷于局部最优。如果模型的训练损失已经趋于稳定，但是模型的性能还没有达到最优，可以尝试增加窗口大小和负样本数量，以提高模型的性能。但由于shuffle操作较为耗时，可以选择在训练到一半以后重新构建batch。

负样本的采样数量：如果文本中的单词之间的语义关系比较稀疏，则选择较大的负样本数量。较小的负样本数量可以提高模型的训练速度，而较大的负样本数量可以提高模型的准确度。

2.3.2 构建batch

```
def build_batch(dataset, batch_size, epoch_num):

    #center_word_batch缓存batch_size个中心词
    center_word_batch = []
    #target_word_batch缓存batch_size个目标词（可以是正样本或者负样本）
    target_word_batch = []
    #label_batch缓存了batch_size个0或1的标签，用于模型训练
    label_batch = []

    for epoch in range(epoch_num):
        #每次开启一个新epoch之前，都对数据进行一次随机打乱，提高训练效果
        print(f"start shuffling at {epoch} epoch...")
        shuf=visualRandom()
        shuf.shuffle(dataset) #shuffle中加入tqdm可视化计算时间
        for center_word, target_word, label in dataset:
            #遍历dataset中的每个样本，并将这些数据送到不同的tensor里
            center_word_batch.append([center_word])
            target_word_batch.append([target_word])
            label_batch.append(label)

            #当样本积攒到一个batch_size后，我们把数据都返回回来
            #在这里我们使用numpy的array函数把list封装成tensor
            #并使用python的迭代器机制，将数据yield出来
            #使用迭代器的好处是可以节省内存
            if len(center_word_batch) == batch_size:
                yield np.array(center_word_batch).astype("int64"),
np.array(target_word_batch).astype("int64"),np.array(label_batch).astype("float32")
                #每一个yield，把一个batch信息搞到迭代器的container里，惰性计算
                center_word_batch = []
                target_word_batch = []
                label_batch = []

    if len(center_word_batch) > 0:
```

```
        yield
np.array(center_word_batch).astype("int64"),np.array(target_word_batch).astype("int6
4"),np.array(label_batch).astype("float32")
```

2.3.3 重写shuffle方法

由于构建batch过程中的shuffle操作过于费时，因此笔者继承了shuffle类并重写了shuffle方法从而对shuffle进行可视化监控

```
class visualRandom(random.Random):
    def __init__(self):
        super(visualRandom,self).__init__()

    def shuffle(self, x, random=None):
        import tqdm
        if random is None:
            randbelow = self._randbelow
            for i in tqdm.tqdm(reversed(range(1, len(x)))):
                # pick an element in x[:i+1] with which to exchange x[i]
                j = randbelow(i+1)
                x[i], x[j] = x[j], x[i]
        else:
            _int = int
            for i in reversed(range(1, len(x))):
                # pick an element in x[:i+1] with which to exchange x[i]
                j = _int(random() * (i+1))
                x[i], x[j] = x[j], x[i]
```

2.4 skipgram 模型

```
class SkipGram(fluid.dygraph.Layer):
    def __init__(self, vocab_size, embedding_size, init_scale=0.1):
        #vocab_size定义了这个skipgram这个模型的词表大小
        #embedding_size定义了词向量的维度是多少
        #init_scale定义了词向量初始化的范围，一般来说，比较小的初始化范围有助于模型训练
        super(SkipGram, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        #使用paddle.fluid.dygraph提供的Embedding函数，构造一个词向量参数
        #这个参数的大小为：[self.vocab_size, self.embedding_size]
        #数据类型为：float32
        #这个参数的名称为：embedding_para
        #这个参数的初始化方式为在[-init_scale, init_scale]区间进行均匀采样
        self.embedding = Embedding(
            size=[self.vocab_size, self.embedding_size], #253854,200
            dtype='float32',
            param_attr=fluid.ParamAttr(
                name='embedding_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))
        #使用paddle.fluid.dygraph提供的Embedding函数，构造另外一个词向量参数
```

```
        #这个参数的大小为：[self.vocab_size, self.embedding_size]
        #数据类型为：float32
        #这个参数的名称为：embedding_para_out
        #这个参数的初始化方式为在[-init_scale, init_scale]区间进行均匀采样
        #跟上面不同的是，这个参数的名称跟上面不同，因此，
        #embedding_para_out和embedding_para虽然有相同的shape，但是权重不共享
        self.embedding_out = Embedding(
            size=[self.vocab_size, self.embedding_size],
            dtype='float32',
            param_attr=fluid.ParamAttr(
                name='embedding_out_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))
    def forward(self, center_words, target_words, label):
        center_words_emb = self.embedding(center_words)
        target_words_emb = self.embedding_out(target_words)
        word_sim = fluid.layers.elementwise_mul(center_words_emb, target_words_emb)
        word_sim = fluid.layers.reduce_sum(word_sim, dim = -1)
        word_sim = fluid.layers.reshape(word_sim, shape=[-1])
        pred = fluid.layers.sigmoid(word_sim)
        #通过估计的输出概率定义损失函数，注意我们使用的是sigmoid_cross_entropy_with_logits函
数
        #将sigmoid计算和cross entropy合并成一步计算可以更好的优化，所以输入的是word_sim，而不
是pred
        loss = fluid.layers.sigmoid_cross_entropy_with_logits(word_sim, label)
        loss = fluid.layers.reduce_mean(loss)
        #返回前向计算的结果，飞桨会通过backward函数自动计算出反向结果。
        return pred, loss
```

针对中心词和目标此分别维护一个embedding矩阵进行学习。

我们设置超参数大小为：

```
batch_size = 512
epoch_num = 3
embedding_size = 200
step = 0
learning_rate = 0.001
```

参数对训练的影响

embedding_size:

较小的词嵌入维度可能会导致信息丢失，而较大的词嵌入维度可能会导致模型过拟合。同时，embedding_size会影响初始化内容，由于

```
param_attr=fluid.ParamAttr(
                name='embedding_out_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))
```

均匀分布的上下界由embedding大小决定。

2.5 训练过程

```
with fluid.dygraph.guard(fluid.CUDAPlace(0)):
    #通过我们定义的SkipGram类，来构造一个Skip-gram模型网络
    skip_gram_model = SkipGram(vocab_size, embedding_size)
    #构造训练这个网络的优化器
    adam = fluid.optimizer.AdamOptimizer(learning_rate=learning_rate, parameter_list
= skip_gram_model.parameters())
    #使用build_batch函数，以mini-batch为单位，遍历训练数据，并训练网络
    for center_words, target_words, label in tqdm.tqdm(build_batch(dataset,
batch_size, epoch_num)):
        center_words_var = fluid.dygraph.to_variable(center_words)
        target_words_var = fluid.dygraph.to_variable(target_words)
        label_var = fluid.dygraph.to_variable(label)
        pred, loss = skip_gram_model(
            center_words_var, target_words_var, label_var)
        loss.backward()
        #通过minimize函数，让程序根据loss，完成一步对参数的优化更新
        adam.minimize(loss)
        #使用clear_gradients函数清空模型中的梯度，以便于下一个mini-batch进行更新
        skip_gram_model.clear_gradients()
        best_loss=100.
        no_optim=0
        step += 1
        #每经过100个mini-batch，打印一次当前的loss，看看loss是否在稳定下降
        if step % 100 == 0:
            print("step %d, loss %.3f" % (step, loss.numpy()[0]))
        if step % 1000 == 0:
            if loss>=best_loss:
                no_optim+=1
            else: #此时的loss较小
                no_optim=0
                best_loss=loss
            if no_optim>6:
                fluid.save_dygraph(skip_gram_model.state_dict(),
f'./weight/skip_gram_model{step}')
                print(f'early stop at {step} epoch')
                break
            writer.add_scalars('trainloss',{'trainloss':loss.numpy()[0]},step)
            writer.flush()
            writer.close()#利用summery writer记录out文件，便于后续在tensorboard中进行可视
化
    fluid.save_dygraph(skip_gram_model.state_dict(),
f'./weight/skip_gram_model{step}')
    print('finish!')
```

同时，在训练过程中设置了梯度监控机制，如果loss长期处于最优（6次），则我们认为此时loss已经达到最优，针对这种情况有两种处理办法。

1. 如前文提到，此时模型学了部分数据分布，已经陷入局部最优，则可以动态调整windowsize，并增加负样本数量，从而对模型进一步训练。（由于训练时间问题，并未尝试此训练方法）
2. 直接early stop(代码展示做法)，并将模型保存

三、算法优化

3.1 dataloader构建的讨论

根据上面代码，我们在build_batch的过程中采用了yield，通过yield，将一个batch信息添加到迭代器的container里，实现了惰性计算。

```
if len(center_word_batch) == batch_size:
                yield np.array(center_word_batch).astype("int64"),
np.array(target_word_batch).astype("int64"),np.array(label_batch).astype("float32")
```

尽管yield定义的生成器可以节省内存并具有灵活性，但生成器数据需要逐个生成，无法利用多核CPU或GPU的并行计算能力。因此我们考虑torch和paddle框架的dataset类，利用dataset构建的dataloader同样利用了迭代器惰性计算的机制，然而他在gpu并行计算以及shuffle，sampler等数据预处理上具有更高的扩展性。

3.2 构建dataset

数据在initialize的过程和上述大致相同，取数据并加入batch的操作与上述代码相同，并减少了篇幅。

```
from paddle.io import Dataset #继承dataset的父类
class text8data(Dataset):
    def __init__(self, data_dir):
        super(text8data, self).__init__()
        self.data_list = []
        #load data from text
        with open(data_dir, "r") as f:
            corpus= f.read().strip("\n")
        f.close()
        #data formating...
        corpus = corpus.strip().lower()
        corpus = corpus.split(" ")
        self.word2id_freq, self.word2id_dict, self.id2word_dict = build_dict(corpus)
        corpus = convert_corpus_to_id(corpus, self.word2id_dict)
        corpus = subsampling(corpus, self.word2id_freq)#subsampling
        self.data_list=build_data(corpus, self.word2id_dict, self.word2id_freq)
        # 传入定义好的数据处理方法，作为自定义数据集类的一个属性
    def __getitem__(self, index):
        # 根据索引，从列表中取出一个图像
        center_word,target_word,label = self.data_list[index]
        return center_word, target_word,label
    def __len__(self):
        return len(self.data_list)
```

3.3 构建dataloader

```
train_loader = paddle.io.DataLoader(text8dataset,batch_size=batch_size,
num_workers=4,drop_last=False) #不丢弃最后一个batch
```

3.4 分布式训练初始化

首先初始化fleet.init(is_collective=True) 保证启用 Paddle的分布式训练模式，从而实现分布式训练。在使用PaddlePaddle 进行分布式训练时，需要启用分布式训练模式，以便在多个进程之间共享模型参数和数据，并实现数据的分布式加载、分布式采样和分布式优化等功能。

```
from paddle.distributed import fleet  # 导入分布式训练库 fleet
from paddle.io import DataLoader, DistributedBatchSampler  # 导入数据加载和采样工具
paddle.set_device('gpu')  # 设置使用 GPU 进行训练
fleet.init(is_collective=True)  # 初始化 fleet，启用分布式训练
skip_gram_model = SkipGram(len(text8dataset.word2id_freq), embedding_size)  # 创建
SkipGram 模型
skip_gram_model = fleet.distributed_model(skip_gram_model)  # 对模型进行分布式包装
adam = fluid.optimizer.AdamOptimizer(learning_rate=learning_rate,
parameter_list=skip_gram_model.parameters())  # 创建 Adam 优化器
adam = fleet.distributed_optimizer(adam)  # 对优化器进行分布式包装
train_sampler = DistributedBatchSampler(text8dataset, batch_size, shuffle=True,
drop_last=false)  # 创建分布式采样器
train_loader = DataLoader(text8dataset, batch_sampler=train_sampler, num_workers=4)
 # 创建数据加载器
```

sampler对一个batch的数据进行切分和同步，从而实现分布式训练

3.5 运行

由于paddle的并行计算需要指定cuda_id以及启动paddle.distributed.launch，执行shell

```
python -m paddle.distributed.launch --gpus=0,1 ddp_skip-gram.py
```

进行训练

四、测试和结果分析

4.1 模型加载

```
import sys
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Embedding
import numpy as np
class SkipGram(fluid.dygraph.Layer):
    def __init__(self, vocab_size, embedding_size, init_scale=0.1):
        self.embedding = Embedding(
            size=[self.vocab_size, self.embedding_size],
            dtype='float32',
```

```
            param_attr=fluid.ParamAttr(
                name='embedding_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))
        self.embedding_out = Embedding(
            size=[self.vocab_size, self.embedding_size],
            dtype='float32',
            param_attr=fluid.ParamAttr(
                name='embedding_out_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))
    def forward(self, center_words, target_words, label):
        center_words_emb = self.embedding(center_words)
        target_words_emb = self.embedding_out(target_words)
        word_sim = fluid.layers.elementwise_mul(center_words_emb, target_words_emb)
        word_sim = fluid.layers.reduce_sum(word_sim, dim = -1)
        word_sim = fluid.layers.reshape(word_sim, shape=[-1])
        pred = fluid.layers.sigmoid(word_sim)
        loss = fluid.layers.sigmoid_cross_entropy_with_logits(word_sim, label)
        loss = fluid.layers.reduce_mean(loss)
        return pred, loss
def load_text8():
    with open("./text8.txt", "r") as f:
        corpus = f.read().strip("\n")
    f.close()
    corpus = corpus.strip().lower()
    corpus = corpus.split(" ")
    return corpus
corpus = load_text8()

def build_dict(corpus):
    word_freq_dict = dict()
    for word in corpus:
        if word not in word_freq_dict:
            word_freq_dict[word] = 0
        word_freq_dict[word] += 1
    word_freq_dict = sorted(word_freq_dict.items(), key = lambda x:x[1], reverse =
True)
    word2id_dict = dict()
    word2id_freq = dict()
    id2word_dict = dict()
    for word, freq in word_freq_dict:
        curr_id = len(word2id_dict)
        word2id_dict[word] = curr_id
        word2id_freq[curr_id] = freq #以id为标识，把标识加到freq中
        id2word_dict[curr_id] = word #
    return word2id_freq, word2id_dict, id2word_dict
def get_similar_tokens(token1, token2, embed):
    W = embed.numpy()
    x = W[word2id_dict[token1]]
    y = W[word2id_dict[token2]]
    cos_sim = np.dot(x, y) / (np.linalg.norm(x) * np.linalg.norm(y))
```

```
    min_val = -1.0
    max_val = 1.0
    cos_sim_mapped = (cos_sim - min_val) / (max_val - min_val) * 10
    #将结果从(-1,1)映射到(0,10)的范围
    return cos_sim_mapped
word2id_freq, word2id_dict, id2word_dict = build_dict(corpus)
vocab_size,embedding_size=len(word2id_dict),200
skip_gram_model = SkipGram(vocab_size, embedding_size)
model_state_dict = paddle.load('/mnt/ve_share/liushuai/skip-
gram/weight/skip_gram_model1024789.pdparams')
skip_gram_model.set_state_dict(model_state_dict)
```

在get_similar_tokens方法中求得余弦相似度，并将结果从(-1,1)映射到(0,10)的范围

4.2 数据写入

```
with open("./output.txt","w")as outf:
    pass
with open ("./wordsim353_agreed.txt")as f:
    datas=f.readlines()
    tmplist=[]
    origindata=[]
    scoredata=[]
    for data in datas:
        data=data.strip().split('\t')
        token1=data[-3]
        token2=data[-2]
        origindata.append((data[-1]))

        with open("./output.txt","a+")as outf:
            if token1 not in word2id_dict or token2 not in word2id_dict:
                score=0
            else:

 score=get_similar_tokens(token1,token2,skip_gram_model.embedding.weight)
                score="{:.2f}".format(score)
            scoredata.append(score)
            new_data = "\t".join(data) + "\t" + str(score) + "\n"
            outf.write(new_data)
```

4.3 结果分析

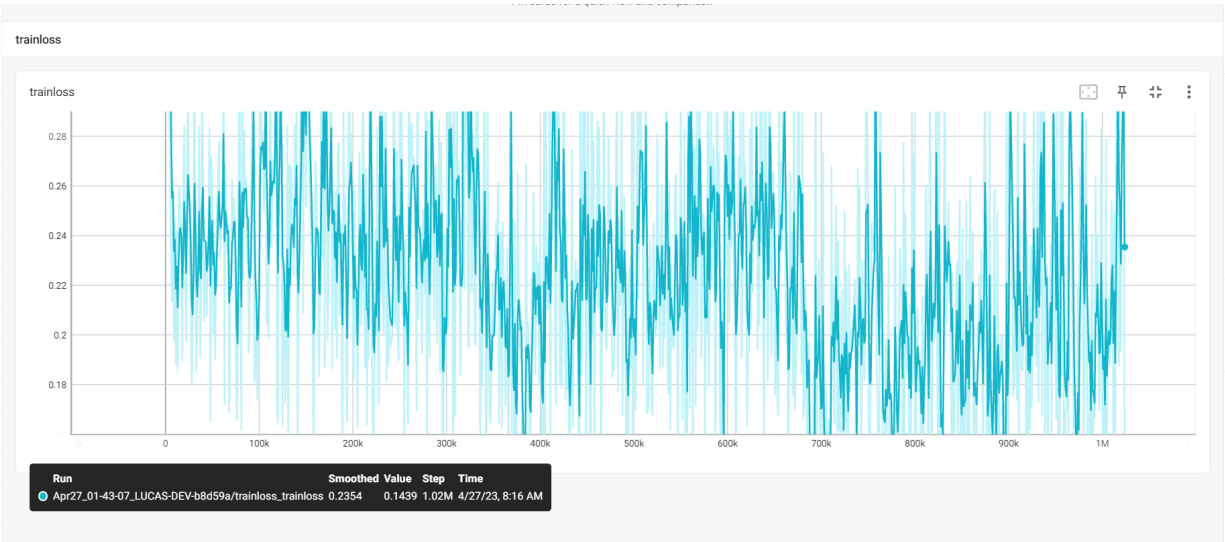
4.3.1loss 可视化

执行

```
tensorboard --logdir=/mnt/ve_share/liushuai/skip-gram/runs/Apr27_01-43-07_LUCAS-DEV-
b8d59a
```

并将服务器端口映射到本地，在相应端口进行查看。

可见loss在400k iter时下降较为明显，此时可能达到局部最优，在700k-900k iter时又处于较低水平，此时可能已经收敛（然而波动较大，应该一直未满足连续六轮loss不上升的要求）



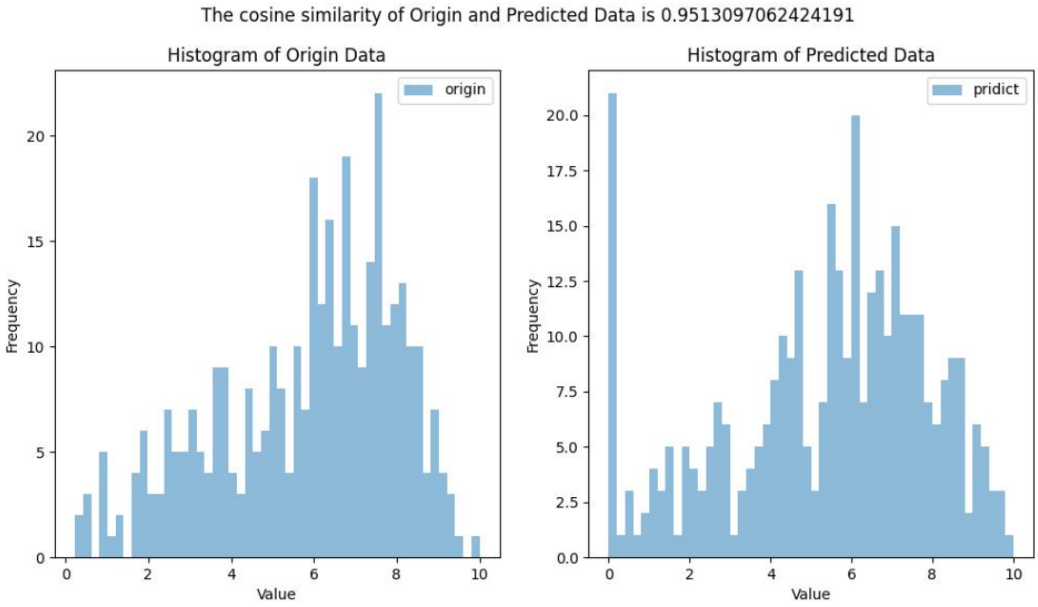
4.2测试结果可视化

```
import numpy as np
with open("./output1.txt")as f :
    datas=f.readlines()
    originlist=[]
    pridict=[]
    for data in datas:
        data=data.strip().split('\t')
        originlist.append(float(data[-2]))
        pridict.append(float(data[-1]))
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.hist(originlist, bins=50, alpha=0.5, label='origin')
ax1.set_xlabel('Value')
ax1.set_ylabel('Frequency')
ax1.set_title('Histogram of Origin Data')
ax1.legend()

ax2.hist(pridict, bins=50, alpha=0.5, label='pridict')
ax2.set_xlabel('Value')
ax2.set_ylabel('Frequency')
ax2.set_title('Histogram of Predicted Data')
ax2.legend()
cosine_similarity = np.dot(originlist, pridict) / (np.linalg.norm(originlist) *
np.linalg.norm(pridict))
plt.suptitle(f'The cosine similarity of Origin and Predicted Data is
{cosine_similarity}')
# 计算两个列表的相似度
plt.savefig("./trash.jpg")
```

测试集数据分布结果



图表显示表示了数据分布的情况的全局信息，而余弦相似度表示了两个每一组数据相似度的局部信息。