## 实验概述

## 实验步骤

### 步骤1：项目入口

本实验通过使用torch构建一个BiLSTM的实战，通常再项目中，为方便进行模型的测试，对项目进行模块化，然后在主函数或者运行函数中进行相关模块的组合，能够大大的提高编译的可观性与可调整性。

在本项目中，run.py是整个项目的入口，它包含两部分，一是使用argparse工具，配置相关参数，二是整个项目的流程框架，各个模块/函数的调用。

在run.py中，首先是相关依赖的导入。

|  |
| --- |
| # coding: UTF-8  import time  import torch  import numpy as np  from train\_eval import train, init\_network  from importlib import import\_module  import argparse |

然后定义第一部分，即配置参数，参数设置与解析如下

|  |
| --- |
| #声明argparse对象 可附加说明  parser = argparse.ArgumentParser(description='Chinese Text Classification')  #添加参数  #模型是必须设置的参数(required=True) 类型是字符串  parser.add\_argument('--model', type=str, required=True, help='choose a model: TextCNN, TextRNN, FastText, TextRCNN, TextRNN\_Att, DPCNN, Transformer')  #embedding随机初始化或使用预训练词或字向量 默认使用预训练  parser.add\_argument('--embedding', default='pre\_trained', type=str, help='random or pre\_trained')  #基于词还是基于字 默认基于字  parser.add\_argument('--word', default=False, type=bool, help='True for word, False for char')  #解析参数  args = parser.parse\_args() |

介绍完参数后，我们进入主函数，对模块进行组建，主要流程为：模型参数导入—数据预处理—构建模型对象---最后调用训练、验证及测试模块。

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  dataset = 'THUCNews' # 数据集  # 搜狗新闻:embedding\_SougouNews.npz, 腾讯:embedding\_Tencent.npz, 随机初始化:random  embedding = 'embedding\_SougouNews.npz'  if args.embedding == 'random':  embedding = 'random'  #获取选择的模型名字  model\_name = args.model # 'TextRCNN' # TextCNN, TextRNN,TextRCNN, TextRNN\_Att  from utils import build\_dataset, build\_iterator, get\_time\_dif  # 根据所选模型名字在models包下 获取相应模块(.py)  x = import\_module('models.' + model\_name)  # 每一个模块(.py)中都有一个模型定义类和与该模型相关的配置类(定义该模型的超参数) 初始化配置类的对象  config = x.Config(dataset, embedding)  #设置随机种子 确保每次运行的条件(模型参数初始化、数据集的切分或打乱等)是一样的  np.random.seed(1)  torch.manual\_seed(1)  torch.cuda.manual\_seed\_all(1)  torch.backends.cudnn.deterministic = True # 保证每次结果一样  # 数据预处理  start\_time = time.time()  print("Loading data...")  vocab, train\_data, dev\_data, test\_data = build\_dataset(config, args.word)  # 构建训练集、验证集、测试集迭代器/生成器（节约内存、避免溢出）  train\_iter = build\_iterator(train\_data, config)  dev\_iter = build\_iterator(dev\_data, config)  test\_iter = build\_iterator(test\_data, config)  time\_dif = get\_time\_dif(start\_time)  print("Time usage:", time\_dif)  # train  # 构造模型对象  config.n\_vocab = len(vocab) #词典大小可能不确定，在运行时赋值  # 构建模型对象 并to\_device  model = x.Model(config).to(config.device)  #使用自定义的参数初始化方式  init\_network(model)  print(model.parameters)  #训练、验证和测试  train(config, model, train\_iter, dev\_iter, test\_iter) |

### 步骤2：训练模块

接下来对run.py函数中的各个模块进行细致的讲解，首先介绍训练及验证模块，文件描述在train\_eval.py中。该模块主要包含4个函数：init\_network，train，test，evaluate。

首先是导入相关依赖

|  |
| --- |
| # coding: UTF-8  import numpy as np  import torch  import torch.nn as nn  import torch.nn.functional as F  from sklearn import metrics  import time  from utils import get\_time\_dif  from tensorboardX import SummaryWriter |

在init\_network函数中，对权重和偏执进行初始化，Xavier initialization是 Glorot 等人为了解决随机初始化的问题提出来的另一种初始化方法，他们的思想倒也简单，就是尽可能的让输入和输出服从相同的分布，这样就能够避免后面层的激活函数的输出值趋向于0。

|  |
| --- |
| # 权重初始化，默认xavier  def init\_network(model, method='xavier', exclude='embedding', seed=123):  for name, w in model.named\_parameters():  if exclude not in name:  if 'weight' in name:  if method == 'xavier':  nn.init.xavier\_normal\_(w)  elif method == 'kaiming':  nn.init.kaiming\_normal\_(w)  else:  nn.init.normal\_(w)  elif 'bias' in name:  nn.init.constant\_(w, 0)  else:  pass |

在train函数中，首先定义模型的训练模式，然后定义优化器，这里我们采用常用的Adam优化器，然后再每个批次的数据中，定义模型训练过程：前向传播-计算损失-计算梯度-更新参数-验证模型，并保存训练集（当前batch）、验证集的损失和准确率信息方便可视化。

|  |
| --- |
| def train(config, model, train\_iter, dev\_iter, test\_iter):  start\_time = time.time()  # 训练模式  model.train()  # 学习率指数衰减，每过一个epoch：学习率 = gamma \* 学习率 也可以采用达观杯中的方式，当模型在验证集上指标不再提高时，学习率减半  # scheduler = torch.optim.lr\_scheduler.ExponentialLR(optimizer, gamma=0.9)  optimizer = torch.optim.Adam(model.parameters(), lr=config.learning\_rate)  total\_batch = 0 # 记录进行到多少batch  dev\_best\_loss = float('inf')  last\_improve = 0 # 记录上次验证集loss下降的batch数  flag = False # 记录是否很久没有效果提升  writer = SummaryWriter(log\_dir=config.log\_path + '/' + time.strftime('%m-%d\_%H.%M', time.localtime()))  for epoch in range(config.num\_epochs):  print('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))  # scheduler.step() # 学习率衰减  for i, (trains, labels) in enumerate(train\_iter):  outputs = model(trains)#前向传播,获取输出  model.zero\_grad()#清空梯度  # 计算交叉熵损失（内部包含softmax log等操作） 可以用nn里面的函数 也可以用F中的函数 labels为整数索引 内部会自动转换为one-hot  loss = F.cross\_entropy(outputs, labels)  loss.backward()#计算梯度  optimizer.step()#更新参数  # 每100个batch 计算一下在验证集上的指标  if total\_batch % 100 == 0:  # 每多少轮输出在训练集和验证集上的效果  true = labels.data.cpu()  predic = torch.max(outputs.data, 1)[1].cpu()  # 当前batch上训练集的准确率 因为是类别均衡数据集，所以可以直接用准确率作评估指标  train\_acc = metrics.accuracy\_score(true, predic)  # 计算此时模型在验证集上的损失和准确率  dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)  # 保存在验证集上损失最小的参数  if dev\_loss < dev\_best\_loss:  dev\_best\_loss = dev\_loss  torch.save(model.state\_dict(), config.save\_path)  improve = '\*'  last\_improve = total\_batch  else:  improve = ''  time\_dif = get\_time\_dif(start\_time)  msg = 'Iter: {0:>6}, Train Loss: {1:>5.2}, Train Acc: {2:>6.2%}, Val Loss: {3:>5.2}, Val Acc: {4:>6.2%}, Time: {5} {6}'  print(msg.format(total\_batch, loss.item(), train\_acc, dev\_loss, dev\_acc, time\_dif, improve))  # 保存 训练集（当前batch）、验证集的损失和准确率信息 方便可视化 以batch为单位  writer.add\_scalar("loss/train", loss.item(), total\_batch)  writer.add\_scalar("loss/dev", dev\_loss, total\_batch)  writer.add\_scalar("acc/train", train\_acc, total\_batch)  writer.add\_scalar("acc/dev", dev\_acc, total\_batch)  model.train()  total\_batch += 1  if total\_batch - last\_improve > config.require\_improvement:  # 验证集loss超过1000batch没下降，结束训练  print("No optimization for a long time, auto-stopping...")  flag = True  break  if flag:  break  writer.close()  # 模型训练结束后 进行测试  test(config, model, test\_iter) |