PRML-期末作业

赵文轩-18307130104

Model-1

中文描述

训练一个文本分类器，用来判断目标文本的情感，数据集采用 IMDb。模型有 1 个 Embedding 层，3 个卷积模块，1 个 Flatten 层，1 个 Dropout 层，用 Sigmoid 激活函数的三层全连接层。每个卷积模块有一个不同大小卷积核的，采用 ReLU 激活函数的二维卷积层，一个最大化 Pooling 层。采用二值交叉熵损失函数，随机梯度下降优化器，最大100次迭代。

英文描述

The training code for classification task on IMDb dataset. The model is composed of one Embedding layer, three convolutional modules (each module has a ReLU activation Convolutional layer with different size of convolution kernel, and a maximize pooling layer), one Flatten layer, one Dropout layer, and three fully connected layer with Sigmoid activation.The training process uses Binary Cross Entropy Loss function and the SGD optimizer. We train 100 epochs in total.

代码

```
from typing import Counter
import torch
import random
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from  torch.utils.data import DataLoader
import torch.optim.lr_scheduler as lr_scheduler
import torchtext
import string
from collections import Counter
maxlen = 2000
PAD_TOK = '<pad>'
UNK_TOK = '<unk>'
class TextCNN(nn.Module):
    def __init__(self, vocab_size):
        filter_num = 100
        embedding_dim = 64
        kernel_list = [3, 4, 5]
        super(TextCNN, self).__init__()
        self.emed1 = nn.Embedding(vocab_size, embedding_dim)
        self.convs = nn.ModuleList([
```

```
            nn.Sequential(nn.Conv2d(1, filter_num, (kernel, embedding_dim)),
                          nn.ReLU(),
                          nn.MaxPool2d((maxlen - kernel + 1, 1)))
            for kernel in kernel_list
        ])
        self.flat = nn.Flatten()
        self.dropout = nn.Dropout(0.1)
        self.lin1 = nn.Linear(filter_num * len(kernel_list), 64)
        self.relu4 = nn.ReLU()
        self.lin2 = nn.Linear(64, 32)
        self.relu5 = nn.ReLU()
        self.lin3 = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # 1 * 317
        x = self.emed1(x)
        # print(x.shape)
        x = x.unsqueeze(1)
        # 1 * 64 * 317
        # print(x.shape)
        out = [conv(x) for conv in self.convs]
        x = torch.cat(out, dim = 1)
        # print(x.shape)
        # x = self.dropout1(x)
        # print(x.shape)
        x = self.flat(x)
        x = self.dropout(x)
        # print(x.shape)
        x = self.relu4(self.lin1(x))
        x = self.relu5(self.lin2(x))
        x = self.sigmoid(self.lin3(x))
        return x

def train(epoch, f):
    print("Training... Epoch = %d" % epoch)
    x_loader = []
    # fo = open("out.txt", "r+")
    for data, target in zip(traindata, trainlabel):
        # print(data, target)
        wd = [word2idx[i] for i in tokenizer(data)]
        if len(wd) <= maxlen:
            t = len(wd)
            for i in range(maxlen - t):
                wd.append(word2idx['<pad>'])
        else:
            wd = wd[0:maxlen]
        idata = torch.LongTensor([wd])
        # print(idata.shape)
        x = model(idata)
        tar = None
```

```
        if target == 'neg':
            tar = torch.Tensor([[0.]])
        else:
            tar = torch.Tensor([[1.]])
        # fo.write(str(x))
        # fo.write(str(tar))
        if f:
            print(x, tar)
        loss = nllloss(x, tar)
        if f:
            print(loss)


        optimizer4nn.zero_grad()
        loss.backward()
        optimizer4nn.step()
        # x_loader.append(x)

    # feat = torch.cat(x_loader, 0)
    # fo.write("finish")
    # fo.close()


train_iter = torchtext.datasets.IMDB(split='train', root='.data')
# train_loader = DataLoader(trainset, batch_size=128, num_workers=4)
tokenizer = torchtext.data.get_tokenizer('basic_english')
trainsetdata = list(train_iter)
# print(trainset)
traindata = []
trainlabel = []
for (label, data) in trainsetdata:
    traindata.append(data)
    trainlabel.append(label)
for i in range(len(traindata)):
    t = random.randint(i, len(traindata) - 1)
    tmp = traindata[i]
    traindata[i] = traindata[t]
    traindata[t] = tmp
    tmp = trainlabel[i]
    trainlabel[i] = trainlabel[t]
    trainlabel[t] = tmp
# traindata = traindata[0:1000]
# trainlabel = trainlabel[0:1000]
vocab = list(set(tokenizer(" ".join(traindata) + " " + UNK_TOK + " " +
PAD_TOK)))
# vocab = torchtext.vocab.build_vocab_from_iterator(word_list)
# vocab = list(set(word_list))
# vocab = torchtext.vocab.Vocab(Counter(specials+word_list))
print(len(vocab))
word2idx = {w: i for i, w in enumerate(vocab)}
model = TextCNN(len(vocab) + 1)
nllloss = nn.BCELoss()
```

```
optimizer4nn = optim.SGD(model.parameters(),lr=0.001)
sheduler = lr_scheduler.StepLR(optimizer4nn,20,gamma=0.8)
for epoch in range(1000):
    # print optimizer4nn.param_groups[0]['lr']
    train(epoch+1, 0)
    sheduler.step()
```

Model-2

中文描述

训练一个语言模型，数据集用 WikiText-2。模型有 1 个 Embedding 层，1 个 LSTM 网络，1 个线性层，1 个 Dropout 层，1 个 Flatten 层。使用交叉损失函数来训练，随机梯度下降优化器，最大100次迭代。

英文描述

The training code for training language model on WikiText-2 dataset. The model is composed of one Embedding layer, one LSTM network, one linear layer, one Dropout layer, one Flatten layer. The training process uses Cross Entropy Loss function and the SGD optimizer. We train 100 epochs in total.

代码

```
from typing import Counter
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from  torch.utils.data import DataLoader
import torch.optim.lr_scheduler as lr_scheduler
from torchtext import datasets, data

PAD_TOK = '<pad>'
UNK_TOK = '<unk>'
class RNNLM(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size,
num_layers):
        super(RNNLM, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,
batch_first=True)
        self.lin = nn.Linear(hidden_size, vocab_size)
        self.dropout = nn.Dropout()
        self.flat = nn.Flatten(0,1)

    def forward(self, x, h, c):
        x = self.embed(x)
        x = x.unsqueeze(0)
        print(x.shape)
```

```
        x, (hn, cn) = self.lstm(x, (h, c))
        x = x.view(x.size(0)*x.size(1), x.size(2))
        # x = x.reshape(x.shape[0] * x.shape[1], x.shape[2])
        x = self.lin(x)
        return x, (hn, cn)

def detach(states):
    return [state.detach() for state in states]
def train(epoch, f):
    print("Training... Epoch = %d" % epoch)
    h = torch.zeros(num_layers, 1, hidden_size)
    c = torch.zeros(num_layers, 1, hidden_size)
    for di in range(0, len(traindata)):
        data = traindata[di]
        # print(data)
        idata = [word2idx[i] for i in tokenizer(data)]
        # print(wd)
        print(len(idata))
        states = (h, c)
        for j in range(len(idata) - seq_len - 1):
            (h, c) = detach(states)
            pred, states = model.forward(torch.LongTensor(idata[j: j + seq_len]), h, c)
            # target = []
            # for d in idata[j + 1: j+ seq_len + 1]:
            #     print(d, vocab_size)
            #     target.append(F.one_hot(torch.LongTensor([d]), vocab_size))
            target = torch.LongTensor(idata[j + 1: j+ seq_len + 1])
            print(pred.shape, target.shape)
            print(pred, target)
            lss = criterion(pred, target)
            print(lss)
            optimizer4nn.zero_grad()
            lss.backward(retain_graph=True)
            optimizer4nn.step()

torch.autograd.set_detect_anomaly(True)
train_iter, validdata, testdata = datasets.WikiText103(root='.data', split=('train', 'valid', 'test'))
traindata = []
for d in list(train_iter):
    traindata.append(d)
tokenizer = data.get_tokenizer('basic_english')
vocab = list(set(tokenizer(" ".join(traindata) + " " + UNK_TOK + " " + PAD_TOK)))
word2idx = {w: i for i, w in enumerate(vocab)}
vocab_size = len(vocab)
embed_size = 128
hidden_size = 1024
num_layers = 2
seq_len = 30
```

```
print(vocab_size)
model = RNNLM(vocab_size, embed_size, hidden_size, num_layers)
criterion = nn.CrossEntropyLoss()
optimizer4nn = optim.SGD(model.parameters(),lr=0.001)
sheduler = lr_scheduler.StepLR(optimizer4nn,20,gamma=0.8)
for epoch in range(1000):
    # print optimizer4nn.param_groups[0]['lr']
    train(epoch+1, 0)
    sheduler.step()
```

model-3

中文描述

训练一个体态识别模型，数据集用 Kinetics400。模型有 1个卷积层，1个 ReLU 函数激活的 BatchNorm 层，1个最大化 Pooling 层， 4 个卷积模块，1个平均 Pooling 层，1个线性层。每个卷积模块包括多个下采样 ResNet 基本块。 ResNet 基本块包括 2 个二维卷积层，2 个 BatchNorm 层，并进行下采样，最后用 ReLU 函数激活。使用交叉损失函数来训练，随机梯度下降优化器，最大100次迭代。

英文描述

The training code for body recognition on Kinetics400 dataset. The model is composed of one Convolution layer，one BatchNorm layer with ReLU activation，one maximize Pooling layer, four convolutional module(each has certain number of ResNet basic blocks with downsample, ResNet basic block has two 2D-convolution layer, two BatchNorm layer, and ReLU activation after downsample operation), one average Pooling layer, one linear layer. The training process uses Cross Entropy Loss function and the SGD optimizer. We train 100 epochs in total.

代码

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from  torch.utils.data import DataLoader
import torch.optim.lr_scheduler as lr_scheduler
from torchvision import datasets
import torch.distributed as dist

def conv3x3(in_planes, out_planes, stride=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3,
stride=stride,
                     padding=1, bias=False)

num_classes = 4000

class Block(nn.Module):
```

```
    expansion = 1
    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Block, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.relu2 = nn.ReLU()
        self.downsample = downsample

    def forward(self, x):
        if self.downsample != None:
            residual = self.downsample(x)
        else:
            residual = x
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x += residual
        x = self.relu2(x)
        return x

class ResNet(nn.Module):
    def __init__(self, in_planes):
        super(ResNet, self).__init__()
        self.inplanes = 32
        self.conv1 = nn.Conv2d(in_planes, 32, kernel_size=7, stride=2,
padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)
        self.layer1 = self.make_layer(Block, 32, 3)
        self.layer2 = self.make_layer(Block, 64, 4, stride=2)
        self.layer3 = self.make_layer(Block, 128, 12, stride=2)
        self.layer4 = self.make_layer(Block, 256, 3, stride=2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(512 * Block.expansion, num_classes)

    def make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )
        layers = []
```

```
        layers.append(block(self.inplanes, planes, stride,
downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = x.float()
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        # print(x.shape)
        x = self.layer1(x)
        # print(x.shape)
        x = self.layer2(x)
        # print(x.shape)
        x = self.layer3(x)
        # print(x.shape)
        x = self.layer4(x)
        # print(x.shape)
        x = self.avgpool(x)
        # print(x.shape)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

def train(epoch):
    print("Training... Epoch = %d" % epoch)
    for video, s, target in data_loader:
        # print(video[0].shape, s.shape, target.shape)
        # print(video[0])
        idata = video[0].permute(0, 3, 1, 2)[:,:,0:224,0:256]
        # print(idata.shape)
        pred = model(idata)
        # print(pred.shape)
        loss = nllloss(pred, target)

        optimizer4nn.zero_grad()
        loss.backward()
        optimizer4nn.step()

kinetics_data = datasets.Kinetics400('.data',frames_per_clip=1,
step_between_clips=5,
                  extensions=('mp4',))
data_loader = DataLoader(kinetics_data, batch_size=1, shuffle=True)

# Model
model = ResNet(3)

# NLLLoss
```

```
nlllossloss = nn.NLLLoss() #CrossEntropyLoss = log_softmax + NLLLoss


# optimzer4nn
optimizer4nn = optim.SGD(model.parameters(),lr=0.001,momentum=0.9,
weight_decay=0.0005)
sheduler = lr_scheduler.StepLR(optimizer4nn,20,gamma=0.8)

for epoch in range(100):
    # print optimizer4nn.param_groups[0]['lr']
    train(epoch+1)
    sheduler.step()
```

Model-4

中文描述

训练一个机器翻译模型，数据集使用 IWSLT2017。模型包括 2 个 Embedding 层，2 个有隐藏层 RNN 网络，分别作为 Encoder 和 Decoder，以及一个线性层。Encoder 输出的隐藏状态会输入到 Decoder 中。使用交叉损失函数来训练，随机梯度下降优化器，最大100次迭代。

英文描述

The training code for machine translation on IWSLT2017 dataset. The model is composed of two Embedding layer, two RNN network with hidden layer used as Encoder and Decoder, a linear layer. The hidden status outputed by Encoder will pass to Decoder. The training process uses Cross Entropy Loss function and the SGD optimizer. We train 100 epochs in total.

代码

```
from typing import Counter
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from  torch.utils.data import DataLoader
import torch.optim.lr_scheduler as lr_scheduler
from torchtext import datasets, data

n_hidden = 128
n_step = 2000

class Seq2Seq(nn.Module):
    def __init__(self, n_class):
        super(Seq2Seq, self).__init__()
        self.embed1 =nn.Embedding(len(ivocab), n_class)
        self.embed2 =nn.Embedding(len(ivocab), n_class)
```

```
        self.encoder = nn.RNN(input_size=n_class, num_layers=3, hidden_size=n_hidden, dropout=1, batch_first=True) # encoder
        self.decoder = nn.RNN(input_size=n_class, num_layers=3, hidden_size=n_hidden, dropout=1, batch_first=True) # decoder
        self.fc = nn.Linear(n_hidden, n_class)

    def forward(self, enc_input, enc_hidden, dec_input):
        print(enc_input.shape, dec_input.shape)
        enc_input = self.embed1(enc_input)
        dec_input = self.embed2(dec_input)

        _, h_t = self.encoder(enc_input, enc_hidden)
        outputs, _ = self.decoder(dec_input, h_t)

        model = self.fc(outputs)
        return model

def train(epoch):
    print("Training... Epoch = %d" % epoch)
    for input, target in zip(iv, ov):
        idata = [iword2idx[i] for i in tokenizer1(input)]
        tmp = len(idata)
        for i in range(tmp, n_step - 1):
            idata.append(iword2idx[PAD_TOK])
        if tmp > n_step:
            print('bigger', tmp)
        ddata = [iword2idx[SOS_TOK]] + idata
        tmp = len(ddata)
        for i in range(tmp, n_step):
            ddata.append(iword2idx[PAD_TOK])
        ddata = torch.LongTensor([ddata])
        idata = torch.LongTensor([idata + [iword2idx[PAD_TOK]]])
        h_0 = torch.zeros(3, 1, n_hidden)
        # print(idata.shape, idata)
        pred = model(idata, h_0, ddata)
        target = [oword2idx[i] for i in tokenizer2(target)] + [oword2idx[EOS_TOK]]
        tmp = len(target)
        for i in range(tmp, n_step):
            target.append(oword2idx[PAD_TOK])
        target = torch.LongTensor(target)
        print(target.shape, target)
        loss = nllloss(pred[0], target)

        optimizer4nn.zero_grad()
        loss.backward()
        optimizer4nn.step()

iwslt, valid, test = datasets.IWSLT2017(root='.data', split=('train', 'valid', 'test'), language_pair=('de', 'en'))
data_loader = DataLoader(iwslt, batch_size=1)
iv = []
```

```
ov = []
for d in data_loader:
    iv.append(d[0][0])
    ov.append(d[1][0])
# iv = ['Er sagte: "Man möchte meinen, dass die Absicht zum Glücklichsein nicht Teil des Schöpfungsplans ist."']
# ov = ['He said, "One feels inclined to say that the intention that man should be happy is not included in the plan of creation."']

tokenizer2 = data.get_tokenizer('basic_english')
tokenizer1 = data.get_tokenizer('spacy', 'de_core_news_sm')
PAD_TOK = '<pad>'
UNK_TOK = '<unk>'
SOS_TOK = '<SOS>'
EOS_TOK = '<EOS>'
ivocab = list(set(tokenizer1(" ".join(iv)) + [UNK_TOK, PAD_TOK, SOS_TOK, EOS_TOK]))
ovocab = list(set(tokenizer2(" ".join(ov)) + [UNK_TOK, PAD_TOK, SOS_TOK, EOS_TOK]))
print(len(ivocab), len(ovocab))
iword2idx = {w: i for i, w in enumerate(ivocab)}
oword2idx = {w: i for i, w in enumerate(ovocab)}
nclass = len(ovocab)
# Model
model = Seq2Seq(nclass)

# NLLLoss
nlllloss = nn.NLLLoss()


# optimzer4nn
optimizer4nn = optim.SGD(model.parameters(),lr=0.001,momentum=0.9,weight_decay=0.0005)
sheduler = lr_scheduler.StepLR(optimizer4nn,20,gamma=0.8)

for epoch in range(100):
    # print optimizer4nn.param_groups[0]['lr']
    train(epoch+1)
    sheduler.step()
```

Model-5

中文描述

训练一个图片标注模型，数据集使用 COCO-Caption。模型包括 1 个 CNN 网络作为 Encoder，1 个 RNN 网络作为 Decoder，Encoder 的输出会作为参数输入到 Decoder 中。CNN-Encoder 网络包括多个 resnet 网络和 1 个线性层。RNN-Decoder 包括 1 个 Embedding 层，1 个 LSTM 网络，1 个线性层。使用交叉损失函数来训练，随机梯度下降优化器，最大100次迭代。

英文描述

The training code for image caption on COCO-Caption dataset. The model is composed of one CNN network for Encoder, one RNN network for Decoder. Output of Encoder will pass to Decoder. CNN-Encoder is composed of serval resnet network and one linear layer. RNN-Decoder is composed of one Embedding layer, one LSTM network, one linear layer. The training process uses Cross Entropy Loss function and the SGD optimizer. We train 100 epochs in total.

代码

```
import torch, torchtext
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from  torch.utils.data import DataLoader
import torch.optim.lr_scheduler as lr_scheduler
from torchvision import datasets, transforms
import torch.distributed as dist
import torchvision.models as models
from torch.nn.utils.rnn import pack_padded_sequence
import pickle

PAD_TOK = '<pad>'
UNK_TOK = '<unk>'
SOS_TOK = '<SOS>'
EOS_TOK = '<EOS>'
class Vocabulary(object):
    def __init__(self):
        self.word2idx = {}
        self.idx2word = {}
        self.idx = 0

    def add_word(self, word):
        if not word in self.word2idx:
            self.word2idx[word] = self.idx
            self.idx2word[self.idx] = word
            self.idx += 1

    def __call__(self, word):
        if not word in self.word2idx:
            return self.word2idx['<unk>']
        return self.word2idx[word]

    def __len__(self):
        return len(self.word2idx)

class EncoderCNN(nn.Module):
    def __init__(self, embed_size):
        super(EncoderCNN, self).__init__()
        resnet = models.resnet152(pretrained=True)
```

```
        modules = list(resnet.children())[:-1]
        self.resnet = nn.Sequential(*modules)
        self.linear = nn.Linear(resnet.fc.in_features, embed_size)
        # self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)

    def forward(self, images):
        with torch.no_grad():
            features = self.resnet(images)
        features = features.reshape(features.size(0), -1)
        features = self.linear(features)
        return features

class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size,
num_layers, max_seq_length=20):
        super(DecoderRNN, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,
batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.max_seg_length = max_seq_length

    def forward(self, features, captions, lengths):
        embeddings = self.embed(captions)
        # print(features.shape, embeddings.shape)
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
        packed = pack_padded_sequence(embeddings, lengths,
batch_first=True)
        hiddens, _ = self.lstm(packed)
        outputs = self.linear(hiddens[0])
        return outputs

class Net(nn.Module):
    def __init__(self, vocab_size):
        super(Net, self).__init__()
        self.encoder = EncoderCNN(256)
        self.decoder = DecoderRNN(256, 512, vocab_size, 1)

    def forward(self, x, captions, lengths):
        features = self.encoder(x)
        output = self.decoder(features, captions, lengths)
        return output


def train(epoch):
    print("Training... Epoch = %d" % epoch)
    for i, (images, captions) in enumerate(data_loader):
        # print(captions)
        idata = []
        for cap in captions:
            idata.append(vocab(SOS_TOK))
            for word in tokenizer(cap[0]):
```

```
                idata.append(vocab(word))
            idata.append(vocab(EOS_TOK))
        captions = torch.LongTensor([idata])
        lengths = torch.Tensor([len(idata)])
        target = pack_padded_sequence(captions, lengths,
batch_first=True)[0]
        # print(images.shape, captions.shape, lengths.shape)
        pred = model(images, captions, lengths)
        loss = nllloss(pred, target)

        optimizer4nn.zero_grad()
        loss.backward()
        optimizer4nn.step()

transform = transforms.Compose([
        transforms.RandomCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406),
                             (0.229, 0.224, 0.225))])

with open('vocab.pkl', 'rb') as f:
    vocab = pickle.load(f)
coco = datasets.CocoCaptions('.data', './.ann/captions_val2014.json',
transform)
data_loader = DataLoader(coco, batch_size=1, shuffle=True)
tokenizer = torchtext.data.get_tokenizer('basic_english')
# Model
model = Net(len(vocab))

# NLLLoss
nllloss = nn.NLLLoss()


# optimzer4nn
optimizer4nn = optim.SGD(model.parameters(),lr=0.001,momentum=0.9,
weight_decay=0.0005)
sheduler = lr_scheduler.StepLR(optimizer4nn,20,gamma=0.8)

for epoch in range(100):
    # print optimizer4nn.param_groups[0]['lr']
    train(epoch+1)
    sheduler.step()
```

vocab 构建

vocab.py

```
import nltk
import pickle
import argparse
```

```python
from collections import Counter
from pycocotools.coco import COCO


class Vocabulary(object):
    """Simple vocabulary wrapper."""
    def __init__(self):
        self.word2idx = {}
        self.idx2word = {}
        self.idx = 0

    def add_word(self, word):
        if not word in self.word2idx:
            self.word2idx[word] = self.idx
            self.idx2word[self.idx] = word
            self.idx += 1

    def __call__(self, word):
        if not word in self.word2idx:
            return self.word2idx['<unk>']
        return self.word2idx[word]

    def __len__(self):
        return len(self.word2idx)

PAD_TOK = '<pad>'
UNK_TOK = '<unk>'
SOS_TOK = '<SOS>'
EOS_TOK = '<EOS>'
def build_vocab(json, threshold):
    coco = COCO(json)
    counter = Counter()
    ids = coco.anns.keys()
    for i, id in enumerate(ids):
        caption = str(coco.anns[id]['caption'])
        tokens = nltk.tokenize.word_tokenize(caption.lower())
        counter.update(tokens)

        if (i+1) % 1000 == 0:
            print("[{}/{}] Tokenized the captions.".format(i+1,
len(ids)))

    # If the word frequency is less than 'threshold', then the word is
discarded.
    words = [word for word, cnt in counter.items() if cnt >= threshold]

    # Create a vocab wrapper and add some special tokens.
    vocab = Vocabulary()
    vocab.add_word(PAD_TOK)
    vocab.add_word(SOS_TOK)
    vocab.add_word(EOS_TOK)
    vocab.add_word(UNK_TOK)
```

```
54
55      # Add the words to the vocabulary.
56      for i, word in enumerate(words):
57          vocab.add_word(word)
58      return vocab
59
60  def main(args):
61      vocab = build_vocab(json=args.caption_path,
    threshold=args.threshold)
62      vocab_path = args.vocab_path
63      with open(vocab_path, 'wb') as f:
64          pickle.dump(vocab, f)
65      print("Total vocabulary size: {}".format(len(vocab)))
66      print("Saved the vocabulary wrapper to '{}'".format(vocab_path))
67
68
69  if __name__ == '__main__':
70      parser = argparse.ArgumentParser()
71      parser.add_argument('--caption_path', type=str,
72                          default='./model-5/.ann/captions_val2014.json',
73                          help='path for train annotation file')
74      parser.add_argument('--vocab_path', type=str, default='./model-
    5/vocab.pkl',
75                          help='path for saving vocabulary wrapper')
76      parser.add_argument('--threshold', type=int, default=4,
77                          help='minimum word count threshold')
78      args = parser.parse_args()
79      main(args)
```