import argparse

import itertools

import os.path

import time

import uuid

import torch

import torch.optim.lr\_scheduler

import numpy as np

import math

import evaluate

import trees

import vocabulary

import makehp

import Zparser

from dep\_reader import CoNLLXReader

import dep\_eval

import utils

import json

import csv

tokens = Zparser

uid = uuid.uuid4().hex[:6]

def torch\_load(load\_path):

if Zparser.use\_cuda:

return torch.load(load\_path)

else:

return torch.load(load\_path, map\_location=lambda storage, location: storage)

def format\_elapsed(start\_time):

elapsed\_time = int(time.time() - start\_time)

minutes, seconds = divmod(elapsed\_time, 60)

hours, minutes = divmod(minutes, 60)

days, hours = divmod(hours, 24)

elapsed\_string = "{}h{:02}m{:02}s".format(hours, minutes, seconds)

if days > 0:

elapsed\_string = "{}d{}".format(days, elapsed\_string)

return elapsed\_string

def make\_hparams():

return makehp.HParams(

max\_len\_train=0, # no length limit

max\_len\_dev=0, # no length limit

sentence\_max\_len=300,

learning\_rate=0.0008,

learning\_rate\_warmup\_steps=40,

clip\_grad\_norm=0., #no clipping

step\_decay=True, # note that disabling step decay is not implemented

step\_decay\_factor=0.5,

step\_decay\_patience=5,

partitioned=True,

use\_cat = False,

const\_lada = 0.5,

num\_layers=12,

d\_model=1024,

num\_heads=8,

d\_kv=64,

d\_ff=2048,

d\_label\_hidden=250,

d\_biaffine = 1024,

attention\_dropout=0.2,

embedding\_dropout=0.2,

relu\_dropout=0.2,

residual\_dropout=0.2,

use\_tags=False,

use\_words=False,

use\_elmo = False,

use\_bert=False,

use\_xlnet = False,

use\_bert\_only=False,

use\_chars\_lstm=False,

dataset = 'ptb',

model\_name = "joint",

embedding\_type = 'random',

#['glove','sskip','random']

embedding\_path = "/data/glove.gz",

punctuation=['.', '`', "'" ,':' ,',', '"' ,'...', '/', ';', '-'],

d\_char\_emb = 64, # A larger value may be better for use\_chars\_lstm

tag\_emb\_dropout=0.2,

word\_emb\_dropout=0.4,

morpho\_emb\_dropout=0.2,

timing\_dropout=0.0,

char\_lstm\_input\_dropout=0.2,

elmo\_dropout=0.5, # Note that this semi-stacks with morpho\_emb\_dropout!

bert\_model="bert-large-uncased",

bert\_do\_lower\_case=True,

bert\_transliterate="",

xlnet\_model="xlnet-large-cased",

xlnet\_do\_lower\_case=False,

pad\_left=False,

)

def count\_wh(str, data, heads, types):

cun\_w = 0

for i, c\_tree in enumerate(data):

nodes = [c\_tree]

while nodes:

node = nodes.pop()

if isinstance(node, trees.InternalParseNode):

cun\_w += node.cun\_w

nodes.extend(reversed(node.children))

print("total wrong head of :", str, "is", cun\_w)

def run\_train(args, hparams):

if args.numpy\_seed is not None:

print("Setting numpy random seed to {}...".format(args.numpy\_seed))

np.random.seed(args.numpy\_seed)

# Make sure that pytorch is actually being initialized randomly.

# On my cluster I was getting highly correlated results from multiple

# runs, but calling reset\_parameters() changed that. A brief look at the

# pytorch source code revealed that pytorch initializes its RNG by

# calling std::random\_device, which according to the C++ spec is allowed

# to be deterministic.

seed\_from\_numpy = np.random.randint(2147483648)

print("Manual seed for pytorch:", seed\_from\_numpy)

torch.manual\_seed(seed\_from\_numpy)

hparams.set\_from\_args(args)

print("Hyperparameters:")

hparams.print()

train\_path = args.train\_ptb\_path

dev\_path = args.dev\_ptb\_path

dep\_train\_path = args.dep\_train\_ptb\_path

dep\_dev\_path = args.dep\_dev\_ptb\_path

if hparams.dataset == 'ctb':

train\_path = args.train\_ctb\_path

dev\_path = args.dev\_ctb\_path

dep\_train\_path = args.dep\_train\_ctb\_path

dep\_dev\_path = args.dep\_dev\_ctb\_path

dep\_reader = CoNLLXReader(dep\_train\_path)

print('Reading dependency parsing data from %s' % dep\_train\_path)

dep\_dev\_reader = CoNLLXReader(dep\_dev\_path)

print('Reading dependency parsing data from %s' % dep\_dev\_path)

counter = 0

dep\_sentences = []

dep\_data = []

dep\_heads = []

dep\_types = []

inst = dep\_reader.getNext()

while inst is not None:

inst\_size = inst.length()

if hparams.max\_len\_train > 0 and inst\_size - 1 > hparams.max\_len\_train:

inst = dep\_reader.getNext()

continue

counter += 1

if counter % 10000 == 0:

print("reading data: %d" % counter)

sent = inst.sentence

dep\_data.append((sent.words, inst.postags, inst.heads, inst.types))

#dep\_sentences.append([(tag, word) for i, (word, tag) in enumerate(zip(sent.words, sent.postags))])

dep\_sentences.append(sent.words)

dep\_heads.append(inst.heads)

dep\_types.append(inst.types)

inst = dep\_reader.getNext()

dep\_reader.close()

print("Total number of data: %d" % counter)

dep\_dev\_data = []

dev\_inst = dep\_dev\_reader.getNext()

dep\_dev\_headid = np.zeros([3000,300],dtype=int)

dep\_dev\_type = []

dep\_dev\_word = []

dep\_dev\_pos = []

dep\_dev\_lengs = np.zeros(3000, dtype=int)

cun = 0

while dev\_inst is not None:

inst\_size = dev\_inst.length()

if hparams.max\_len\_dev > 0 and inst\_size - 1> hparams.max\_len\_dev:

dev\_inst = dep\_dev\_reader.getNext()

continue

dep\_dev\_lengs[cun] = inst\_size

sent = dev\_inst.sentence

dep\_dev\_data.append((sent.words, dev\_inst.postags, dev\_inst.heads, dev\_inst.types))

for i in range(inst\_size):

dep\_dev\_headid[cun][i] = dev\_inst.heads[i]

dep\_dev\_type.append(dev\_inst.types)

dep\_dev\_word.append(sent.words)

dep\_dev\_pos.append(sent.postags)

#dep\_sentences.append([(tag, word) for i, (word, tag) in enumerate(zip(sent.words, sent.postags))])

dev\_inst = dep\_dev\_reader.getNext()

cun = cun + 1

dep\_dev\_reader.close()

print("Loading training trees from {}...".format(train\_path))

#delete later

with open('data/VnData/VnData3/dep/dep\_types.txt', 'w') as f:

for i in range(0, len(dep\_types)):

f.write(str(dep\_types[i]) + '\n')

with open('data/VnData/VnData3/dep/dep\_heads.txt', 'w') as f:

for i in range(0, len(dep\_heads)):

f.write(str(dep\_heads[i]) + '\n')

#

train\_treebank = trees.load\_trees(train\_path, dep\_heads, dep\_types, dep\_sentences)

if hparams.max\_len\_train > 0:

train\_treebank = [tree for tree in train\_treebank if len(list(tree.leaves())) <= hparams.max\_len\_train]

print("Loaded {:,} training examples.".format(len(train\_treebank)))

print("Loading development trees from {}...".format(dev\_path))

dev\_treebank = trees.load\_trees(dev\_path, dep\_dev\_headid, dep\_dev\_type, dep\_dev\_word)

if hparams.max\_len\_dev > 0:

dev\_treebank = [tree for tree in dev\_treebank if len(list(tree.leaves())) <= hparams.max\_len\_dev]

print("Loaded {:,} development examples.".format(len(dev\_treebank)))

print("Processing trees for training...")

train\_parse = [tree.convert() for tree in train\_treebank]

dev\_parse = [tree.convert() for tree in dev\_treebank]

count\_wh("train data:", train\_parse, dep\_heads, dep\_types)

count\_wh("dev data:", dev\_parse, dep\_dev\_headid, dep\_dev\_type)

print("Constructing vocabularies...")

tag\_vocab = vocabulary.Vocabulary()

tag\_vocab.index(Zparser.START)

tag\_vocab.index(Zparser.STOP)

tag\_vocab.index(Zparser.TAG\_UNK)

word\_vocab = vocabulary.Vocabulary()

word\_vocab.index(Zparser.START)

word\_vocab.index(Zparser.STOP)

word\_vocab.index(Zparser.UNK)

label\_vocab = vocabulary.Vocabulary()

label\_vocab.index(())

sublabels = [Zparser.Sub\_Head]

label\_vocab.index(tuple(sublabels))

type\_vocab = vocabulary.Vocabulary()

char\_set = set()

for i, tree in enumerate(train\_parse):

const\_sentences = [leaf.word for leaf in tree.leaves()]

if len(const\_sentences) != len(dep\_sentences[i]):

continue

nodes = [tree]

while nodes:

node = nodes.pop()

if isinstance(node, trees.InternalParseNode):

label\_vocab.index(node.label)

if node.type is not Zparser.ROOT:#not include root type

type\_vocab.index(node.type)

nodes.extend(reversed(node.children))

else:

tag\_vocab.index(node.tag)

word\_vocab.index(node.word)

type\_vocab.index(node.type)

char\_set |= set(node.word)

char\_vocab = vocabulary.Vocabulary()

#char\_vocab.index(tokens.CHAR\_PAD)

# If codepoints are small (e.g. Latin alphabet), index by codepoint directly

highest\_codepoint = max(ord(char) for char in char\_set)

if highest\_codepoint < 512:

if highest\_codepoint < 256:

highest\_codepoint = 256

else:

highest\_codepoint = 512

# This also takes care of constants like tokens.CHAR\_PAD

for codepoint in range(highest\_codepoint):

char\_index = char\_vocab.index(chr(codepoint))

assert char\_index == codepoint

else:

char\_vocab.index(tokens.CHAR\_UNK)

char\_vocab.index(tokens.CHAR\_START\_SENTENCE)

char\_vocab.index(tokens.CHAR\_START\_WORD)

char\_vocab.index(tokens.CHAR\_STOP\_WORD)

char\_vocab.index(tokens.CHAR\_STOP\_SENTENCE)

for char in sorted(char\_set):

char\_vocab.index(char)

tag\_vocab.freeze()

word\_vocab.freeze()

label\_vocab.freeze()

char\_vocab.freeze()

type\_vocab.freeze()

punctuation = hparams.punctuation

punct\_set = punctuation

def print\_vocabulary(name, vocab):

special = {tokens.START, tokens.STOP, tokens.UNK}

print("{} ({:,}): {}".format(

name, vocab.size,

sorted(value for value in vocab.values if value in special) +

sorted(value for value in vocab.values if value not in special)))

if args.print\_vocabs:

print\_vocabulary("Tag", tag\_vocab)

print\_vocabulary("Word", word\_vocab)

print\_vocabulary("Label", label\_vocab)

print\_vocabulary("Char", char\_vocab)

print\_vocabulary("Type", type\_vocab)

print("Initializing model...")

#load path: put path to best trained model

load\_path = None

#load\_path = 'models/run26th/joint\_single\_best\_dev=79.29\_devuas=87.27\_devlas=79.63.pt'

if load\_path is not None:

print(f"Loading parameters from {load\_path}")

info = torch\_load(load\_path)

parser = Zparser.ChartParser.from\_spec(info['spec'], info['state\_dict'])

else:

parser = Zparser.ChartParser(

tag\_vocab,

word\_vocab,

label\_vocab,

char\_vocab,

type\_vocab,

hparams,

)

print("Initializing optimizer...")

trainable\_parameters = [param for param in parser.parameters() if param.requires\_grad]

trainer = torch.optim.Adam(trainable\_parameters, lr=1., betas=(0.9, 0.98), eps=1e-9)

if load\_path is not None:

trainer.load\_state\_dict(info['trainer'])

def set\_lr(new\_lr):

for param\_group in trainer.param\_groups:

param\_group['lr'] = new\_lr

assert hparams.step\_decay, "Only step\_decay schedule is supported"

warmup\_coeff = hparams.learning\_rate / hparams.learning\_rate\_warmup\_steps

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(

trainer, 'max',

factor=hparams.step\_decay\_factor,

patience=hparams.step\_decay\_patience,

verbose=True,

)

def schedule\_lr(iteration):

iteration = iteration + 1

if iteration <= hparams.learning\_rate\_warmup\_steps:

set\_lr(iteration \* warmup\_coeff)

clippable\_parameters = trainable\_parameters

grad\_clip\_threshold = np.inf if hparams.clip\_grad\_norm == 0 else hparams.clip\_grad\_norm

print("Training...")

total\_processed = 0

current\_processed = 0

check\_every = len(train\_parse) / args.checks\_per\_epoch

best\_dev\_score = -np.inf

best\_model\_path = 'models/joint\_model'

model\_name = hparams.model\_name

best\_uas = -1

best\_las = -1

best\_fscore =-1

best\_epoch\_num = -1

print("This is ", model\_name)

start\_time = time.time()

def check\_dev(epoch\_num):

nonlocal best\_dev\_score

nonlocal best\_model\_path

nonlocal best\_uas

nonlocal best\_las

nonlocal best\_fscore

nonlocal best\_epoch\_num

dev\_start\_time = time.time()

parser.eval()

dev\_predicted = []

for dev\_start\_index in range(0, len(dev\_treebank), args.eval\_batch\_size):

subbatch\_trees = dev\_treebank[dev\_start\_index:dev\_start\_index+args.eval\_batch\_size]

subbatch\_sentences = [[(leaf.tag, leaf.word) for leaf in tree.leaves()] for tree in subbatch\_trees]

predicted, \_,= parser.parse\_batch(subbatch\_sentences)

del \_

dev\_predicted.extend([p.convert() for p in predicted])

dev\_fscore = evaluate.evalb(args.evalb\_dir, dev\_treebank, dev\_predicted)

print(

"dev-fscore {} "

"dev-elapsed {} "

"total-elapsed {}".format(

dev\_fscore,

format\_elapsed(dev\_start\_time),

format\_elapsed(start\_time),

)

)

dev\_pred\_head = [[leaf.father for leaf in tree.leaves()] for tree in dev\_predicted]

dev\_pred\_type = [[leaf.type for leaf in tree.leaves()] for tree in dev\_predicted]

assert len(dev\_pred\_head) == len(dev\_pred\_type)

assert len(dev\_pred\_type) == len(dep\_dev\_type)

stats, stats\_nopunc, stats\_root, num\_inst = dep\_eval.eval(len(dev\_pred\_head), dep\_dev\_word, dep\_dev\_pos,

dev\_pred\_head, dev\_pred\_type,

dep\_dev\_headid, dep\_dev\_type,

dep\_dev\_lengs, punct\_set=punct\_set,

symbolic\_root=False)

dev\_ucorr, dev\_lcorr, dev\_total, dev\_ucomlpete, dev\_lcomplete = stats

dev\_ucorr\_nopunc, dev\_lcorr\_nopunc, dev\_total\_nopunc, dev\_ucomlpete\_nopunc, dev\_lcomplete\_nopunc = stats\_nopunc

dev\_root\_corr, dev\_total\_root = stats\_root

dev\_total\_inst = num\_inst

print(

'W. Punct: ucorr: %d, lcorr: %d, total: %d, uas: %.2f%%, las: %.2f%%, ucm: %.2f%%, lcm: %.2f%%' % (

dev\_ucorr, dev\_lcorr, dev\_total, dev\_ucorr \* 100 / dev\_total, dev\_lcorr \* 100 / dev\_total,

dev\_ucomlpete \* 100 / dev\_total\_inst, dev\_lcomplete \* 100 / dev\_total\_inst))

print(

'Wo Punct: ucorr: %d, lcorr: %d, total: %d, uas: %.2f%%, las: %.2f%%, ucm: %.2f%%, lcm: %.2f%%' % (

dev\_ucorr\_nopunc, dev\_lcorr\_nopunc, dev\_total\_nopunc,

dev\_ucorr\_nopunc \* 100 / dev\_total\_nopunc,

dev\_lcorr\_nopunc \* 100 / dev\_total\_nopunc,

dev\_ucomlpete\_nopunc \* 100 / dev\_total\_inst, dev\_lcomplete\_nopunc \* 100 / dev\_total\_inst))

print('Root: corr: %d, total: %d, acc: %.2f%%' % (

dev\_root\_corr, dev\_total\_root, dev\_root\_corr \* 100 / dev\_total\_root))

dev\_uas = dev\_ucorr\_nopunc \* 100 / dev\_total\_nopunc

dev\_las = dev\_lcorr\_nopunc \* 100 / dev\_total\_nopunc

#las

if dev\_uas > best\_dev\_score:

best\_model\_path = 'models/joint\_model'

if best\_model\_path is not None:

extensions = [".pt"]

for ext in extensions:

path = best\_model\_path + ext

if os.path.exists(path):

print("Removing previous model file {}...".format(path))

os.remove(path)

best\_dev\_score = dev\_uas

best\_model\_path = "{}\_best\_dev={:.2f}\_devuas={:.2f}\_devlas={:.2f}".format(

args.model\_path\_base, dev\_fscore.fscore, dev\_uas,dev\_las)

print("Saving new best model to {}...".format(best\_model\_path))

torch.save({

'spec': parser.spec,

'state\_dict': parser.state\_dict(),

'trainer' : trainer.state\_dict(),

}, best\_model\_path + ".pt")

best\_uas = dev\_uas

best\_las = dev\_las

best\_fscore = dev\_fscore.fscore

best\_epoch\_num = epoch\_num

for epoch in itertools.count(start=1):

if args.epochs is not None and epoch > args.epochs:

break

#check\_dev(epoch)

np.random.shuffle(train\_parse)

epoch\_start\_time = time.time()

for start\_index in range(0, len(train\_parse), args.batch\_size):

trainer.zero\_grad()

schedule\_lr(total\_processed // args.batch\_size)

parser.train()

batch\_loss\_value = 0.0

batch\_trees = train\_parse[start\_index:start\_index + args.batch\_size]

batch\_sentences = [[(leaf.tag, leaf.word) for leaf in tree.leaves()] for tree in batch\_trees]

for subbatch\_sentences, subbatch\_trees in parser.split\_batch(batch\_sentences, batch\_trees, args.subbatch\_max\_tokens):

\_, loss = parser.parse\_batch(subbatch\_sentences, subbatch\_trees)

loss = loss / len(batch\_trees)

loss\_value = float(loss.data.cpu().numpy())

batch\_loss\_value += loss\_value

if loss\_value > 0:

loss.backward()

del loss

total\_processed += len(subbatch\_trees)

current\_processed += len(subbatch\_trees)

grad\_norm = torch.nn.utils.clip\_grad\_norm\_(clippable\_parameters, grad\_clip\_threshold)

trainer.step()

print(

"epoch {:,} "

"batch {:,}/{:,} "

"processed {:,} "

"batch-loss {:.4f} "

"grad-norm {:.4f} "

"epoch-elapsed {} "

"total-elapsed {}".format(

epoch,

start\_index // args.batch\_size + 1,

int(np.ceil(len(train\_parse) / args.batch\_size)),

total\_processed,

batch\_loss\_value,

grad\_norm,

format\_elapsed(epoch\_start\_time),

format\_elapsed(start\_time),

)

)

if current\_processed >= check\_every:

current\_processed -= check\_every

check\_dev(epoch)

# adjust learning rate at the end of an epoch

if hparams.step\_decay:

if (total\_processed // args.batch\_size + 1) > hparams.learning\_rate\_warmup\_steps:

scheduler.step(best\_dev\_score)

best\_model\_path = "{}\_best\_dev={:.2f}\_devuas={:.2f}\_devlas={:.2f}".format(args.model\_path\_base, best\_fscore, best\_uas,best\_las)

print("Saved best model is {}.pt in epoch {}.".format(best\_model\_path, best\_epoch\_num))

def run\_test(args):

const\_test\_path = args.consttest\_ptb\_path

dep\_test\_path = args.deptest\_ptb\_path

if args.dataset == 'ctb':

const\_test\_path = args.consttest\_ctb\_path

dep\_test\_path = args.deptest\_ctb\_path

print("Loading model from {}...".format(args.model\_path\_base))

assert args.model\_path\_base.endswith(".pt"), "Only pytorch savefiles supported"

info = torch\_load(args.model\_path\_base)

assert 'hparams' in info['spec'], "Older savefiles not supported"

parser = Zparser.ChartParser.from\_spec(info['spec'], info['state\_dict'])

parser.eval()

dep\_test\_reader = CoNLLXReader(dep\_test\_path, parser.type\_vocab)

print('Reading dependency parsing data from %s' % dep\_test\_path)

dep\_test\_data = []

test\_inst = dep\_test\_reader.getNext()

dep\_test\_headid = np.zeros([40000, 300], dtype=int)

dep\_test\_type = []

dep\_test\_word = []

dep\_test\_pos = []

dep\_test\_lengs = np.zeros(40000, dtype=int)

cun = 0

while test\_inst is not None:

inst\_size = test\_inst.length()

dep\_test\_lengs[cun] = inst\_size

sent = test\_inst.sentence

dep\_test\_data.append((sent.words, test\_inst.postags, test\_inst.heads, test\_inst.types))

for i in range(inst\_size):

dep\_test\_headid[cun][i] = test\_inst.heads[i]

dep\_test\_type.append(test\_inst.types)

dep\_test\_word.append(sent.words)

dep\_test\_pos.append(sent.postags)

# dep\_sentences.append([(tag, word) for i, (word, tag) in enumerate(zip(sent.words, sent.postags))])

test\_inst = dep\_test\_reader.getNext()

cun = cun + 1

dep\_test\_reader.close()

print("Loading test trees from {}...".format(const\_test\_path))

test\_treebank = trees.load\_trees(const\_test\_path, dep\_test\_headid, dep\_test\_type, dep\_test\_word)

print("Loaded {:,} test examples.".format(len(test\_treebank)))

print("Parsing test sentences...")

start\_time = time.time()

punct\_set = ['.', '`', "'" ,':' ,',', '"' ,'...', '/', ';', '-']

parser.eval()

test\_predicted = []

for start\_index in range(0, len(test\_treebank), args.eval\_batch\_size):

subbatch\_trees = test\_treebank[start\_index:start\_index + args.eval\_batch\_size]

subbatch\_sentences = [[(leaf.tag, leaf.word) for leaf in tree.leaves()] for tree in subbatch\_trees]

predicted, \_, = parser.parse\_batch(subbatch\_sentences)

del \_

test\_predicted.extend([p.convert() for p in predicted])

test\_fscore = evaluate.evalb(args.evalb\_dir, test\_treebank, test\_predicted)

print(

"test-fscore {} "

"test-elapsed {}".format(

test\_fscore,

format\_elapsed(start\_time),

)

)

test\_pred\_head = [[leaf.father for leaf in tree.leaves()] for tree in test\_predicted]

test\_pred\_type = [[leaf.type for leaf in tree.leaves()] for tree in test\_predicted]

#print(test\_pred\_head)

#print(test\_pred\_type)

with open('predicted\_result.csv', 'w', encoding='utf-8') as f:

header = ['Stt', 'Word', 'Predicted head', 'Predicted type', 'True head', 'True type']

writer = csv.writer(f, delimiter='\t')

writer.writerow(header)

for i in range(0, len(test\_pred\_head)):

stt = list(range(1, len(test\_pred\_head[i]) + 1))

writer.writerows(zip(stt, dep\_test\_word[i], test\_pred\_head[i], test\_pred\_type[i], dep\_test\_headid[i], dep\_test\_type[i]))

writer.writerows('\n')

assert len(test\_pred\_head) == len(test\_pred\_type)

assert len(test\_pred\_type) == len(dep\_test\_type)

stats, stats\_nopunc, stats\_root, test\_total\_inst = dep\_eval.eval(len(test\_pred\_head), dep\_test\_word, dep\_test\_pos,

test\_pred\_head,

test\_pred\_type, dep\_test\_headid, dep\_test\_type,

dep\_test\_lengs, punct\_set=punct\_set,

symbolic\_root=False)

test\_ucorrect, test\_lcorrect, test\_total, test\_ucomlpete\_match, test\_lcomplete\_match = stats

test\_ucorrect\_nopunc, test\_lcorrect\_nopunc, test\_total\_nopunc, test\_ucomlpete\_match\_nopunc, test\_lcomplete\_match\_nopunc = stats\_nopunc

test\_root\_correct, test\_total\_root = stats\_root

print(

'best test W. Punct: ucorr: %d, lcorr: %d, total: %d, uas: %.2f%%, las: %.2f%%, ucm: %.2f%%, lcm: %.2f%%' % (

test\_ucorrect, test\_lcorrect, test\_total, test\_ucorrect \* 100 / test\_total,

test\_lcorrect \* 100 / test\_total,

test\_ucomlpete\_match \* 100 / test\_total\_inst, test\_lcomplete\_match \* 100 / test\_total\_inst

))

print(

'best test Wo Punct: ucorr: %d, lcorr: %d, total: %d, uas: %.2f%%, las: %.2f%%, ucm: %.2f%%, lcm: %.2f%% ' % (

test\_ucorrect\_nopunc, test\_lcorrect\_nopunc, test\_total\_nopunc,

test\_ucorrect\_nopunc \* 100 / test\_total\_nopunc,

test\_lcorrect\_nopunc \* 100 / test\_total\_nopunc,

test\_ucomlpete\_match\_nopunc \* 100 / test\_total\_inst,

test\_lcomplete\_match\_nopunc \* 100 / test\_total\_inst))

print('best test Root: corr: %d, total: %d, acc: %.2f%%' % (

test\_root\_correct, test\_total\_root, test\_root\_correct \* 100 / test\_total\_root))

print(

'============================================================================================================================')

def run\_parse(args):

print("Loading model from {}...".format(args.model\_path\_base))

assert args.model\_path\_base.endswith(".pt"), "Only pytorch savefiles supported"

info = torch\_load(args.model\_path\_base)

assert 'hparams' in info['spec'], "Older savefiles not supported"

parser = Zparser.ChartParser.from\_spec(info['spec'], info['state\_dict'])

parser.eval()

print("Parsing sentences...")

with open(args.input\_path) as input\_file:

sentences = input\_file.readlines()

sentences = [sentence.split() for sentence in sentences]

print(sentences)

# Parser does not do tagging, so use a dummy tag when parsing from raw text

if 'UNK' in parser.tag\_vocab.indices:

dummy\_tag = 'UNK'

else:

dummy\_tag = parser.tag\_vocab.value(0)

start\_time = time.time()

def save\_data(syntree\_pred, cun):

pred\_head = [[leaf.father for leaf in tree.leaves()] for tree in syntree\_pred]

pred\_type = [[leaf.type for leaf in tree.leaves()] for tree in syntree\_pred]

appent\_string = "\_" + str(cun) + ".txt"

if args.output\_path\_synconst != '-':

with open(args.output\_path\_synconst + appent\_string, 'w') as output\_file:

for tree in syntree\_pred:

output\_file.write("{}\n".format(tree.convert().linearize()))

print("Output written to:", args.output\_path\_synconst)

if args.output\_path\_syndep != '-':

with open(args.output\_path\_syndep + appent\_string, 'w') as output\_file:

for heads in pred\_head:

output\_file.write("{}\n".format(heads))

print("Output written to:", args.output\_path\_syndep)

if args.output\_path\_synlabel != '-':

with open(args.output\_path\_synlabel + appent\_string, 'w') as output\_file:

for labels in pred\_type:

output\_file.write("{}\n".format(labels))

print("Output written to:", args.output\_path\_synlabel)

syntree\_pred = []

cun = 0

for start\_index in range(0, len(sentences), args.eval\_batch\_size):

subbatch\_sentences = sentences[start\_index:start\_index+args.eval\_batch\_size]

subbatch\_sentences = [[(dummy\_tag, word) for word in sentence] for sentence in subbatch\_sentences]

syntree, \_ = parser.parse\_batch(subbatch\_sentences)

syntree\_pred.extend(syntree)

if args.save\_per\_sentences <= len(syntree\_pred) and args.save\_per\_sentences > 0:

save\_data(syntree\_pred, cun)

syntree\_pred = []

cun += 1

if 0 < len(syntree\_pred):

save\_data(syntree\_pred, cun)

def main():

parser = argparse.ArgumentParser()

subparsers = parser.add\_subparsers()

hparams = make\_hparams()

subparser = subparsers.add\_parser("train")

subparser.set\_defaults(callback=lambda args: run\_train(args, hparams))

hparams.populate\_arguments(subparser)

subparser.add\_argument("--numpy-seed", type=int)

subparser.add\_argument("--model-path-base", required=True)

subparser.add\_argument("--embedding-path", required=True)

subparser.add\_argument("--embedding-type", default="random")

subparser.add\_argument("--model-name", default="test")

subparser.add\_argument("--evalb-dir", default="EVALB/")

subparser.add\_argument("--dataset", default="ptb")

subparser.add\_argument("--train-ptb-path", default="data/02-21.10way.clean")

subparser.add\_argument("--dev-ptb-path", default="data/22.auto.clean")

subparser.add\_argument("--dep-train-ptb-path", default="data/ptb\_train\_3.3.0.sd")

subparser.add\_argument("--dep-dev-ptb-path", default="data/ptb\_dev\_3.3.0.sd")

subparser.add\_argument("--train-ctb-path", default="data/train\_ctb.txt")

subparser.add\_argument("--dev-ctb-path", default="data/dev\_ctb.txt")

subparser.add\_argument("--dep-train-ctb-path", default="data/train\_ctb.conll")

subparser.add\_argument("--dep-dev-ctb-path", default="data/dev\_ctb.conll")

subparser.add\_argument("--batch-size", type=int, default=250)

subparser.add\_argument("--subbatch-max-tokens", type=int, default=2000)

subparser.add\_argument("--eval-batch-size", type=int, default=30)

subparser.add\_argument("--epochs", type=int, default=150)

subparser.add\_argument("--checks-per-epoch", type=int, default=4)

subparser.add\_argument("--print-vocabs", action="store\_true")

subparser = subparsers.add\_parser("test")

subparser.set\_defaults(callback=run\_test)

subparser.add\_argument("--model-path-base", required=True)

subparser.add\_argument("--evalb-dir", default="EVALB/")

subparser.add\_argument("--embedding-path", default="data/glove.6B.100d.txt.gz")

subparser.add\_argument("--dataset", default="ptb")

subparser.add\_argument("--consttest-ptb-path", default="data/23.auto.clean")

subparser.add\_argument("--deptest-ptb-path", default="data/ptb\_test\_3.3.0.sd")

subparser.add\_argument("--consttest-ctb-path", default="data/test\_ctb.txt")

subparser.add\_argument("--deptest-ctb-path", default="data/test\_ctb.conll")

subparser.add\_argument("--eval-batch-size", type=int, default=100)

subparser = subparsers.add\_parser("parse")

subparser.set\_defaults(callback=run\_parse)

subparser.add\_argument("--model-path-base", required=True)

subparser.add\_argument("--embedding-path", default="data/glove.6B.100d.txt.gz")

subparser.add\_argument("--dataset", default="ptb")

subparser.add\_argument("--save-per-sentences", type=int, default=-1)

subparser.add\_argument("--input-path", type=str, required=True)

subparser.add\_argument("--output-path-synconst", type=str, default="-")

subparser.add\_argument("--output-path-syndep", type=str, default="-")

subparser.add\_argument("--output-path-synlabel", type=str, default="-")

subparser.add\_argument("--eval-batch-size", type=int, default=50)

args = parser.parse\_args()

args.callback(args)

# %%

if \_\_name\_\_ == "\_\_main\_\_":

main()