

**课 程 报 告**

**课程名称： 自然语言处理**

**专业班级： 计算机科学与技术202001**

**学 号： U202015533**

**姓 名： 徐瑞达**

**指导教师： 魏巍**

**报告日期： 2023年 6 月 14 日**

**计算机科学与技术学院**

目 录

[1 依存句法分析 2](#_Toc137673323)

[1.1 问题描述 2](#_Toc137673324)

[1.1.1 短语结构树 2](#_Toc137673325)

[1.1.2 依存句法树 2](#_Toc137673326)

[1.1.3 依存句法分析 2](#_Toc137673327)

[1.2 基础模块 3](#_Toc137673328)

[1.3 系统实现 3](#_Toc137673329)

[1.3.1 arc-standard算法 3](#_Toc137673334)

[1.3.2 创建Oracle 4](#_Toc137673335)

[1.3.3 算法优化 4](#_Toc137673336)

[1.3.4 具体实现 4](#_Toc137673337)

[1.3.5 效果评估 5](#_Toc137673338)

[1.4 实验小结 6](#_Toc137673339)

[参考文献 7](#_Toc137673340)

[附录A 依存句法分析实现的源程序 8](#_Toc137673341)

# 1 依存句法分析

## 1.1 问题描述

语法分析(syntactic parsing)是NLP中的重要任务，其目标是分析句子的语法结构并将其表示为树形结构。

## 短语结构树

短语结构语法描述了如何自顶而下地生成句子，反过来，句子也可以用短语结构语法来递归的分解，其基于生成观点描述句法结构。常见的短语结构树库如宾州树库、CTB树库，然而由于短语结构语法较复杂，相应句法分析器的准确率不高，因此研究者大部分转向研究另一种语法形式。

## 依存句法树

依存句法树关注的是句子中词语之间的语法联系，并且将其约束为树形结构。依存语法理论认为词与词之间存在主从关系，即在句子中，如果一个词修饰另一个词，则称修饰词为从属词(dependent)，被修饰的词语称为支配词(head),两者之间的语法关系称为依存关系(dependency relation)。将一个句子中所有词语的依存关系以有向边的形式表示出来，就会得到一棵树，称为依存句法树(dependency parse tree)。常见的依存句法树库有UD树库、CTB树库（需要进行转换），可以使用Dependency Viewer（南京大学汤光超老师开发）可视化依存句法树。

## 依存句法分析

依存句法分析的输入通常是词语和词性，输出则是一棵依存句法树。

作为最一般的形式，依存句法树可以表示为一个有向图G=(V, A)。V代表节点，句子中的每一个词都对于一个节点。A表示有向边，表示词之间有依存关系，边上有一个标签表示具体的依存关系(如nsub)。通常假设依存分析的结果是一棵树，其满足如下条件：

* 有一个特殊的树根节点，没有入边
* 其它节点有且仅有一条入边(可以有零个或者多个出边)
* 对于每一个叶子节点(词)，存在且仅存在一条从根到它的路径

依存句法分析有两种实现方法，即基于图的依存句法分析和基于转移的依存句法分析，本实验使用基于转移的依存句法分析实现。

## 1.2 基础模块

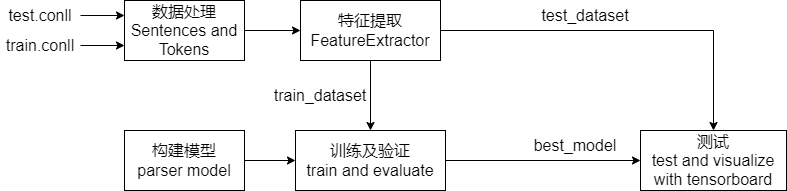


图1-1 依存句法分析算法模块

## 1.3 系统实现



## arc-standard算法

与编译原理中的移进归约算法类似，基于转换的算法有一个栈和一个词队列，以及一个已经parse的依存关系，这三个数据结构组合在一起称为一种配置。arc-standard算法包含三种操作：

* LEFT-ARC：栈顶和它下面的词构成依存关系，并且中心词是栈顶元素，把这两个词从栈中弹出，把这个依存关系加入到已parse的数据结构里，最后把中心词再加到栈中
* RIGHT-ARC：栈顶和它下面的词构成依存关系，中心词是下面的元素，把这两个词从栈中弹出，把这个依存关系加入到已parse的数据结构里，最后把中心词再加到栈中
* SHIFT：把队列中的一个词加入到栈顶

基于上述三种操作，算法伪代码描述如下：

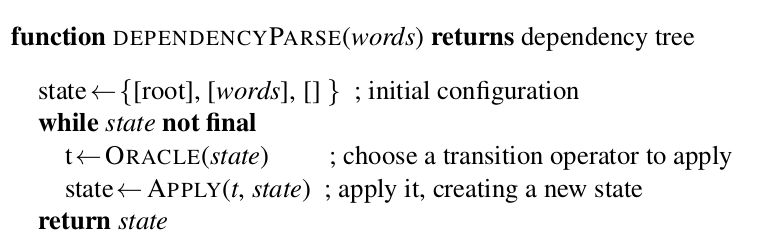


图1-2 arc-standard算法伪代码

初始状态栈里只有一个根节点root，队列中包含所有的词，然后循环算法直到栈中只有root且队列也为空时结束。每次循环时，根据当前状态使用Oracle函数选择合适的操作并执行。如果Oracle函数不合适，那么最终的parse结果可能错误，因此需要根据训练数据学习在不同的状态下采取什么操作才能得到正确的依存语法树。

## 创建Oracle

由1.3.1可知，基于转换的依存句法分析算法核心是训练一个分类器，其输入是一个状态，输出是一个操作（及其预测的label）。假设当前的栈是S，当前得到的关系列表是Rc，正确的依存树的点集合是V，边集合是Rp，则选择操作的伪代码如下：

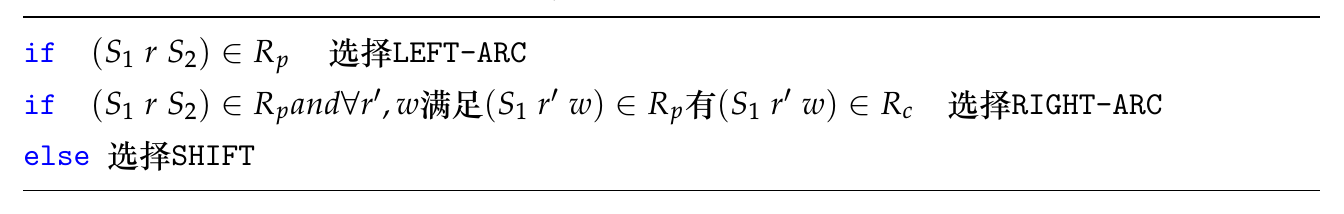


图1-3 oracle创建算法

## 算法优化

在图1-2中，算法在输出操作序列时使用的是贪心算法，当产生第一个错误后无法回溯，容易产生错误累积，解决方法如下：

* 使用复杂的分类器，如Bi-LSTM
* Beam Search：在每个时刻选择多个操作
* Dynamic Oracle：在选择操作时，动态调整Oracle函数

## 具体实现

本项目基于Manning等人的工作[2]和Eliyahu Kiperwasser等人的工作[3]实现基于Bi-LSTM的解析器。

首先需要读取训练集，训练集格式为.conll文件，用类Sentence表示一个句子，Token表示一个词（一个Sentence包含多个Token）。

然后使用FeatureExtractor根据Sentences提取特征，包括栈、队列和已经得到的依存关系。得到提取的特征集后，即获取各个状态下应进行的操作（获取Oracle函数）。

构建模型时，使用立方函数作为激活函数，使用交叉熵作为损失函数。同时，使用Bi-LSTM改进解析器的效果。最后对模型进行训练，并根据在验证集上的效果更新epoch、batch\_size等超参数。

## 效果评估

由于依存分析就是为每个词指定head，因此只需要计算其准确率。计算准确率时可以使用UAS和LAS两种方法，这里使用了UAS方法。

实验选择训练20个epoch，使用TensorBoard可视化训练过程及结果。

训练过程中的batch\_loss下降过程如下：

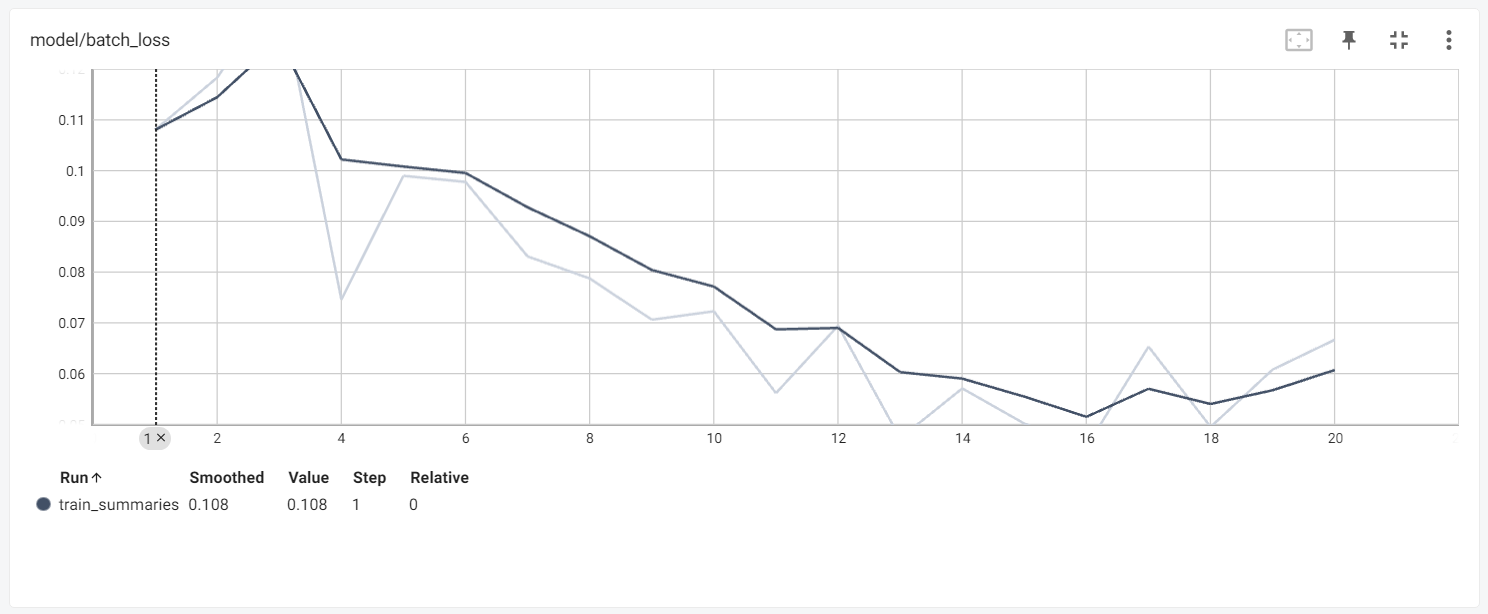


图1-4 batch\_loss图

最后一轮的UAS效果如下：

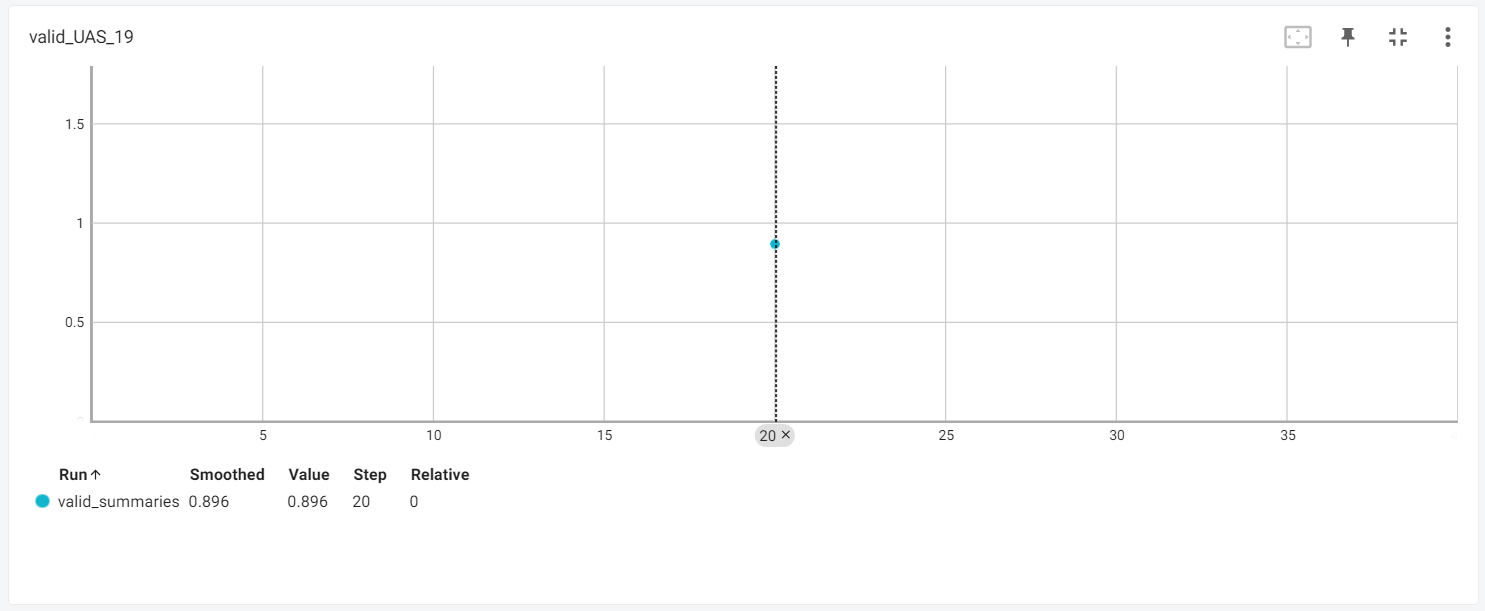


图1-5 UAS结果图

由上图可知，在测试集上的UAS为0.896，效果较好。

使用DependencyViewer工具将测试生成的树文件可视化展示如下：

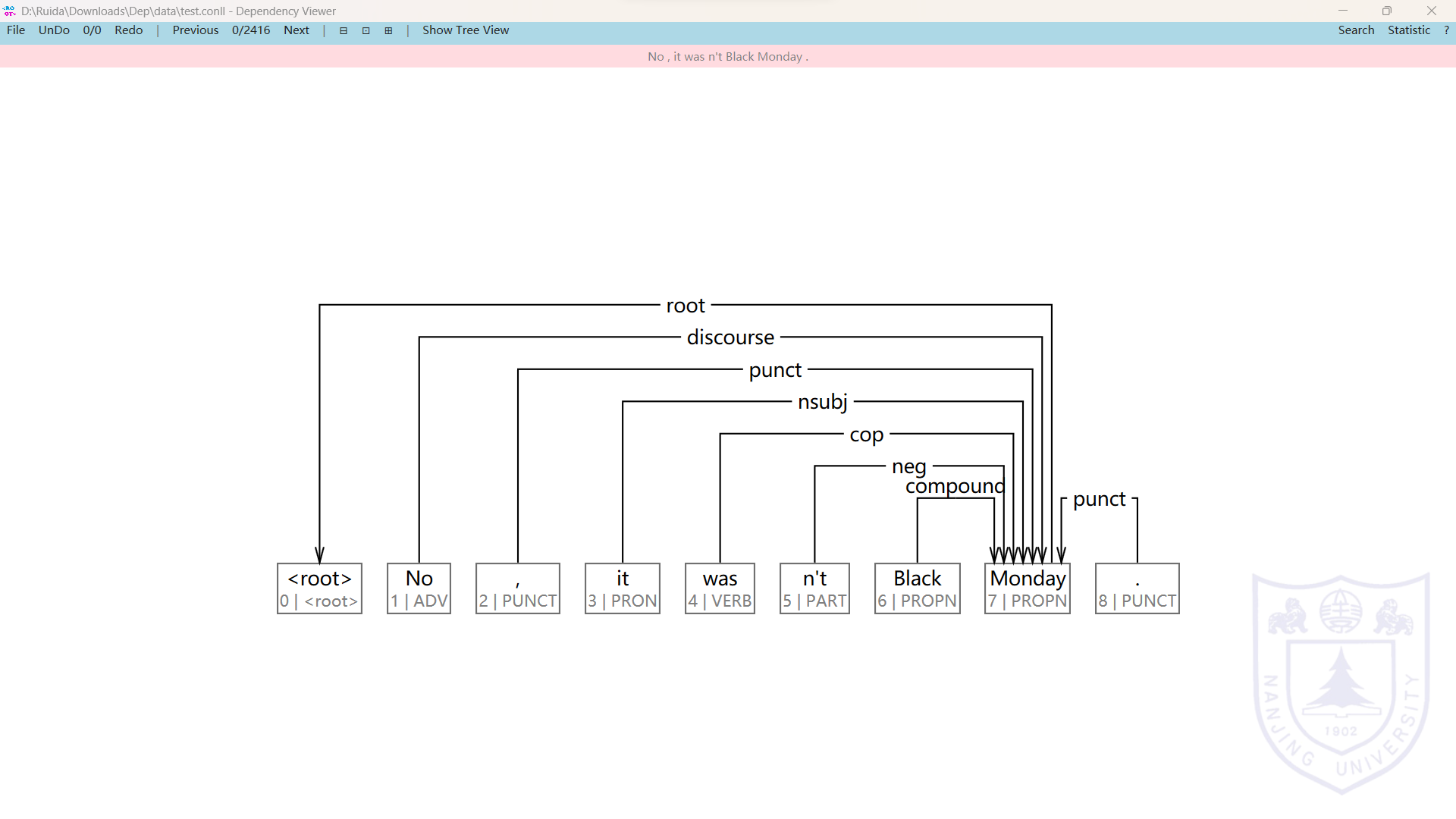


图1-6 依存语法树可视化

## 1.4 实验小结

通过处理依存句法分析任务，我对自然语言处理产生了更加浓厚的兴趣。同时，通过使用Bi-LSTM优化解析器的性能，我也更加认识到深度学习的强大。

在这门自然语言处理课程中，从n-gram到序列标注任务，从基本的概率模型到深奥的贝叶斯网络和马尔可夫随机场，从前馈神经网络、卷积神经网络、循环神经网络再到如今强大的Transfom模型，我系统地学习了NLP领域的基本理论知识和深度学习实践。在实验中，通过使用Bert优化Bi-LSTM+CRF模型在序列标注任务上的性能，以及使用Bi-LSTM提高依存句法分析的效果，大大提高了我对深度学习的掌握程度。

最后，感谢魏老师的授课和助教们的帮助！

# 参考文献

[1] 郑捷著. NLP汉语自然语言处理---原理与实践. 电子工业出版社

[2] Danqi Chen, Christopher Manning. A Fast and Accurate Dependency Parser using Neural Networks.

[3] Eliyahu Kiperwasser, Yoav Goldberg. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations.

# 附录A 依存句法分析实现的源程序

### parser\_model.py

import os

import time

import tensorflow as tf

import numpy as np

from base\_model import Model

from params\_init import random\_uniform\_initializer, random\_normal\_initializer, xavier\_initializer

from general\_utils import Progbar

from general\_utils import get\_minibatches

from feature\_extraction import load\_datasets, DataConfig, Flags, punc\_pos, pos\_prefix

from tf\_utils import visualize\_sample\_embeddings

class ParserModel(Model):

def \_\_init\_\_(self, config, word\_embeddings, pos\_embeddings, dep\_embeddings):

self.word\_embeddings = word\_embeddings

self.pos\_embeddings = pos\_embeddings

self.dep\_embeddings = dep\_embeddings

self.config = config

self.build()

def add\_placeholders(self):

with tf.compat.v1.variable\_scope("input\_placeholders"):

self.word\_input\_placeholder = tf.compat.v1.placeholder(shape=[None, self.config.word\_features\_types],

dtype=tf.int32, name="batch\_word\_indices")

self.pos\_input\_placeholder = tf.compat.v1.placeholder(shape=[None, self.config.pos\_features\_types],

dtype=tf.int32, name="batch\_pos\_indices")

self.dep\_input\_placeholder = tf.compat.v1.placeholder(shape=[None, self.config.dep\_features\_types],

dtype=tf.int32, name="batch\_dep\_indices")

with tf.compat.v1.variable\_scope("label\_placeholders"):

self.labels\_placeholder = tf.compat.v1.placeholder(shape=[None, self.config.num\_classes],

dtype=tf.float32, name="batch\_one\_hot\_targets")

with tf.compat.v1.variable\_scope("regularization"):

self.dropout\_placeholder = tf.compat.v1.placeholder(

shape=(), dtype=tf.float32, name="dropout")

def create\_feed\_dict(self, inputs\_batch, labels\_batch=None, keep\_prob=1):

feed\_dict = {

self.word\_input\_placeholder: inputs\_batch[0],

self.pos\_input\_placeholder: inputs\_batch[1],

self.dep\_input\_placeholder: inputs\_batch[2],

self.dropout\_placeholder: keep\_prob

}

if labels\_batch is not None:

feed\_dict[self.labels\_placeholder] = labels\_batch

return feed\_dict

def write\_gradient\_summaries(self, grad\_tvars):

with tf.name\_scope("gradient\_summaries"):

for (grad, tvar) in grad\_tvars:

mean = tf.reduce\_mean(grad)

stddev = tf.sqrt(tf.reduce\_mean(tf.square(grad - mean)))

tf.compat.v1.summary.histogram("{}/hist".format(tvar.name), grad)

tf.compat.v1.summary.scalar("{}/mean".format(tvar.name), mean)

tf.compat.v1.summary.scalar("{}/stddev".format(tvar.name), stddev)

tf.compat.v1.summary.scalar(

"{}/sparsity".format(tvar.name), tf.nn.zero\_fraction(grad))

def add\_embedding(self):

with tf.compat.v1.variable\_scope("feature\_lookup"):

self.word\_embedding\_matrix = random\_uniform\_initializer(self.word\_embeddings.shape, "word\_embedding\_matrix",

0.01, trainable=True)

self.pos\_embedding\_matrix = random\_uniform\_initializer(self.pos\_embeddings.shape, "pos\_embedding\_matrix",

0.01, trainable=True)

self.dep\_embedding\_matrix = random\_uniform\_initializer(self.dep\_embeddings.shape, "dep\_embedding\_matrix",

0.01, trainable=True)

word\_context\_embeddings = tf.nn.embedding\_lookup(

self.word\_embedding\_matrix, self.word\_input\_placeholder)

pos\_context\_embeddings = tf.nn.embedding\_lookup(

self.pos\_embedding\_matrix, self.pos\_input\_placeholder)

dep\_context\_embeddings = tf.nn.embedding\_lookup(

self.dep\_embedding\_matrix, self.dep\_input\_placeholder)

word\_embeddings = tf.reshape(word\_context\_embeddings,

[-1, self.config.word\_features\_types \*

self.config.embedding\_dim],

name="word\_context\_embeddings")

pos\_embeddings = tf.reshape(pos\_context\_embeddings,

[-1, self.config.pos\_features\_types \*

self.config.embedding\_dim],

name="pos\_context\_embeddings")

dep\_embeddings = tf.reshape(dep\_context\_embeddings,

[-1, self.config.dep\_features\_types \*

self.config.embedding\_dim],

name="dep\_context\_embeddings")

with tf.compat.v1.variable\_scope("batch\_inputs"):

embeddings = tf.concat(

[word\_embeddings, pos\_embeddings, dep\_embeddings], 1, name="batch\_feature\_matrix")

return embeddings, word\_embeddings, pos\_embeddings, dep\_embeddings

def add\_cube\_prediction\_op(self):

print("\*\*\*Building network with CUBE activation\*\*\*")

\_, word\_embeddings, pos\_embeddings, dep\_embeddings = self.add\_embedding()

with tf.compat.v1.variable\_scope("layer\_connections"):

with tf.compat.v1.variable\_scope("layer\_1"):

w11 = random\_uniform\_initializer((self.config.word\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w11",

0.01, trainable=True)

w12 = random\_uniform\_initializer((self.config.pos\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w12",

0.01, trainable=True)

w13 = random\_uniform\_initializer((self.config.dep\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w13",

0.01, trainable=True)

b1 = random\_uniform\_initializer((self.config.l1\_hidden\_size,), "bias1",

0.01, trainable=True)

"""

w11 = xavier\_initializer((self.config.word\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w11")

w12 = xavier\_initializer((self.config.pos\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w12")

w13 = xavier\_initializer((self.config.dep\_features\_types \* self.config.embedding\_dim,

self.config.l1\_hidden\_size), "w13")

b1 = xavier\_initializer((self.config.l1\_hidden\_size,), "bias1")

"""

# for visualization

preactivations = tf.pow(tf.add\_n([tf.matmul(word\_embeddings, w11),

tf.matmul(

pos\_embeddings, w12),

tf.matmul(dep\_embeddings, w13)]) + b1, 3, name="preactivations")

tf.compat.v1.summary.histogram("preactivations", preactivations)

# non\_positive\_activation\_fraction = tf.reduce\_mean(tf.cast(tf.less\_equal(preactivations, 0),

# tf.float32))

# tf.compat.v1.summary.scalar("non\_positive\_activations\_fraction", non\_positive\_activation\_fraction)

h1 = tf.nn.dropout(preactivations,

rate=self.dropout\_placeholder,

name="output\_activations")

with tf.compat.v1.variable\_scope("layer\_2"):

"""

w2 = xavier\_initializer((self.config.l1\_hidden\_size, self.config.l2\_hidden\_size), "w2")

b2 = xavier\_initializer((self.config.l2\_hidden\_size,), "bias2")

"""

w2 = random\_uniform\_initializer((self.config.l1\_hidden\_size, self.config.l2\_hidden\_size), "w2",

0.01, trainable=True)

b2 = random\_uniform\_initializer((self.config.l2\_hidden\_size,), "bias2",

0.01, trainable=True)

h2 = tf.nn.relu(tf.add(tf.matmul(h1, w2), b2),

name="activations")

with tf.compat.v1.variable\_scope("layer\_3"):

"""

w3 = xavier\_initializer((self.config.l2\_hidden\_size, self.config.num\_classes), "w3")

b3 = xavier\_initializer((self.config.num\_classes,), "bias3")

"""

w3 = random\_uniform\_initializer((self.config.l2\_hidden\_size, self.config.num\_classes), "w3",

0.01, trainable=True)

b3 = random\_uniform\_initializer(

(self.config.num\_classes,), "bias3", 0.01, trainable=True)

with tf.compat.v1.variable\_scope("predictions"):

predictions = tf.add(tf.matmul(h2, w3), b3,

name="prediction\_logits")

return predictions

def add\_prediction\_op(self):

print("\*\*\*Building network with ReLU activation\*\*\*")

x = self.add\_embedding()

with tf.compat.v1.variable\_scope("layer\_connections"):

with tf.compat.v1.variable\_scope("layer\_1"):

w1 = xavier\_initializer((self.config.num\_features\_types \* self.config.embedding\_dim,

self.config.hidden\_size), "w1")

b1 = xavier\_initializer((self.config.hidden\_size,), "bias1")

# for visualization

preactivations = tf.add(

tf.matmul(x, w1), b1, name="preactivations")

tf.compat.v1.summary.histogram("preactivations", preactivations)

non\_positive\_activation\_fraction = tf.reduce\_mean(tf.cast(tf.less\_equal(preactivations, 0),

tf.float32))

tf.compat.v1.summary.scalar(

"non\_negative\_activations\_fraction", non\_positive\_activation\_fraction)

h1 = tf.nn.dropout(tf.nn.relu(preactivations),

rate=self.dropout\_placeholder,

name="output\_activations")

with tf.compat.v1.variable\_scope("layer\_2"):

w2 = xavier\_initializer(

(self.config.hidden\_size, self.config.num\_classes), "w2")

b2 = xavier\_initializer((self.config.num\_classes,), "bias2")

with tf.compat.v1.variable\_scope("predictions"):

predictions = tf.add(tf.matmul(h1, w2), b2,

name="prediction\_logits")

return predictions

def l2\_loss\_sum(self, tvars):

return tf.add\_n([tf.nn.l2\_loss(t) for t in tvars], "l2\_norms\_sum")

def add\_loss\_op(self, pred):

tvars = tf.compat.v1.trainable\_variables()

without\_bias\_tvars = [

tvar for tvar in tvars if 'bias' not in tvar.name]

with tf.compat.v1.variable\_scope("loss"):

cross\_entropy\_loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(

labels=self.labels\_placeholder, logits=pred), name="batch\_xentropy\_loss")

l2\_loss = tf.multiply(self.config.reg\_val, self.l2\_loss\_sum(

without\_bias\_tvars), name="l2\_loss")

loss = tf.add(cross\_entropy\_loss, l2\_loss, name="total\_batch\_loss")

tf.compat.v1.summary.scalar("batch\_loss", loss)

return loss

def add\_accuracy\_op(self, pred):

with tf.compat.v1.variable\_scope("accuracy"):

accuracy = tf.reduce\_mean(tf.cast(tf.equal(tf.argmax(pred, axis=1),

tf.argmax(self.labels\_placeholder, axis=1)), dtype=tf.float32),

name="curr\_batch\_accuracy")

return accuracy

def add\_training\_op(self, loss):

with tf.compat.v1.variable\_scope("optimizer"):

optimizer = tf.compat.v1.train.AdamOptimizer(

learning\_rate=self.config.lr, name="adam\_optimizer")

tvars = tf.compat.v1.trainable\_variables()

grad\_tvars = optimizer.compute\_gradients(loss, tvars)

self.write\_gradient\_summaries(grad\_tvars)

train\_op = optimizer.apply\_gradients(grad\_tvars)

return train\_op

# inputs\_batch : list([list(word\_id), list(pos\_id)])

def get\_word\_pos\_inputs(self, inputs\_batch):

# inputs\_batch: [ [[1,2], [3,4], [5,6]], [[7,8], [9,10],[11,12]] ]

inputs\_batch = np.asarray(inputs\_batch)

word\_inputs\_batch, pos\_inputs\_batch, dep\_inputs\_batch = np.split(

inputs\_batch, 3, 1)

# removes extra dimenstion -> convert 3-d to 2-d matrix

word\_inputs\_batch = np.squeeze(word\_inputs\_batch)

pos\_inputs\_batch = np.squeeze(pos\_inputs\_batch)

dep\_inputs\_batch = np.squeeze(dep\_inputs\_batch)

return word\_inputs\_batch, pos\_inputs\_batch, dep\_inputs\_batch

def train\_on\_batch(self, sess, inputs\_batch, labels\_batch, merged):

word\_inputs\_batch, pos\_inputs\_batch, dep\_inputs\_batch = inputs\_batch

feed = self.create\_feed\_dict([word\_inputs\_batch, pos\_inputs\_batch, dep\_inputs\_batch], labels\_batch=labels\_batch,

keep\_prob=self.config.keep\_prob)

\_, summary, loss = sess.run(

[self.train\_op, merged, self.loss], feed\_dict=feed)

return summary, loss

def compute\_dependencies(self, sess, data, dataset):

sentences = data

rem\_sentences = [sentence for sentence in sentences]

[sentence.clear\_prediction\_dependencies() for sentence in sentences]

[sentence.clear\_children\_info() for sentence in sentences]

while len(rem\_sentences) != 0:

curr\_batch\_size = min(

dataset.model\_config.batch\_size, len(rem\_sentences))

batch\_sentences = rem\_sentences[:curr\_batch\_size]

enable\_features = [0 if len(sentence.stack) == 1 and len(sentence.buff) == 0 else 1 for sentence in

batch\_sentences]

enable\_count = np.count\_nonzero(enable\_features)

while enable\_count > 0:

curr\_sentences = [sentence for i, sentence in enumerate(

batch\_sentences) if enable\_features[i] == 1]

# get feature for each sentence

# call predictions -> argmax

# store dependency and left/right child

# update state

# repeat

curr\_inputs = [

dataset.feature\_extractor.extract\_for\_current\_state(sentence, dataset.word2idx, dataset.pos2idx,

dataset.dep2idx) for sentence in curr\_sentences]

word\_inputs\_batch = [curr\_inputs[i][0]

for i in range(len(curr\_inputs))]

pos\_inputs\_batch = [curr\_inputs[i][1]

for i in range(len(curr\_inputs))]

dep\_inputs\_batch = [curr\_inputs[i][2]

for i in range(len(curr\_inputs))]

predictions = sess.run(self.pred,

feed\_dict=self.create\_feed\_dict([word\_inputs\_batch, pos\_inputs\_batch,

dep\_inputs\_batch]))

legal\_labels = np.asarray([sentence.get\_legal\_labels() for sentence in curr\_sentences],

dtype=np.float32)

legal\_transitions = np.argmax(

predictions + 1000 \* legal\_labels, axis=1)

# update left/right children so can be used for next feature vector

[sentence.update\_child\_dependencies(transition) for (sentence, transition) in

zip(curr\_sentences, legal\_transitions) if transition != 2]

# update state

[sentence.update\_state\_by\_transition(legal\_transition, gold=False) for (sentence, legal\_transition) in

zip(curr\_sentences, legal\_transitions)]

enable\_features = [0 if len(sentence.stack) == 1 and len(sentence.buff) == 0 else 1 for sentence in

batch\_sentences]

enable\_count = np.count\_nonzero(enable\_features)

# Reset stack and buffer

[sentence.reset\_to\_initial\_state() for sentence in batch\_sentences]

rem\_sentences = rem\_sentences[curr\_batch\_size:]

def get\_UAS(self, data):

correct\_tokens = 0

all\_tokens = 0

punc\_token\_pos = [pos\_prefix + each for each in punc\_pos]

for sentence in data:

# reset each predicted head before evaluation

[token.reset\_predicted\_head\_id() for token in sentence.tokens]

head = [-2] \* len(sentence.tokens)

# assert len(sentence.dependencies) == len(sentence.predicted\_dependencies)

for h, t, in sentence.predicted\_dependencies:

head[t.token\_id] = h.token\_id

non\_punc\_tokens = [

token for token in sentence.tokens if token.pos not in punc\_token\_pos]

correct\_tokens += sum([1 if token.head\_id == head[token.token\_id] else 0 for (\_, token) in enumerate(

non\_punc\_tokens)])

# all\_tokens += len(sentence.tokens)

all\_tokens += len(non\_punc\_tokens)

UAS = correct\_tokens / float(all\_tokens)

return UAS

def run\_epoch(self, sess, config, dataset, train\_writer, merged):

prog = Progbar(

target=1 + len(dataset.train\_inputs[0]) / config.batch\_size)

for i, (train\_x, train\_y) in enumerate(get\_minibatches([dataset.train\_inputs, dataset.train\_targets],

config.batch\_size, is\_multi\_feature\_input=True)):

# print "input, outout: {}, {}".format(np.array(train\_x).shape, np.array(train\_y).shape)

summary, loss = self.train\_on\_batch(sess, train\_x, train\_y, merged)

prog.update(i + 1, [("train loss", loss)])

# train\_writer.add\_summary(summary, global\_step=i)

return summary, loss # Last batch

def run\_valid\_epoch(self, sess, dataset):

print("Evaluating on dev set",)

self.compute\_dependencies(sess, dataset.valid\_data, dataset)

valid\_UAS = self.get\_UAS(dataset.valid\_data)

print("- dev UAS: {:.2f}".format(valid\_UAS \* 100.0))

return valid\_UAS

def fit(self, sess, saver, config, dataset, train\_writer, valid\_writer, merged):

best\_valid\_UAS = 0

for epoch in range(config.n\_epochs):

print("Epoch {:} out of {:}".format(

epoch + 1, self.config.n\_epochs))

summary, loss = self.run\_epoch(

sess, config, dataset, train\_writer, merged)

if (epoch + 1) % dataset.model\_config.run\_valid\_after\_epochs == 0:

valid\_UAS = self.run\_valid\_epoch(sess, dataset)

valid\_UAS\_summary = tf.compat.v1.summary.scalar(

"valid\_UAS", tf.constant(valid\_UAS, dtype=tf.float32))

valid\_writer.add\_summary(

sess.run(valid\_UAS\_summary), epoch + 1)

if valid\_UAS > best\_valid\_UAS:

best\_valid\_UAS = valid\_UAS

if saver:

print("New best dev UAS! Saving model..")

saver.save(sess, os.path.join(DataConfig.data\_dir\_path, DataConfig.model\_dir,

DataConfig.model\_name))

# trainable variables summary -> only for training

if (epoch + 1) % dataset.model\_config.write\_summary\_after\_epochs == 0:

train\_writer.add\_summary(summary, global\_step=epoch + 1)

print

def highlight\_string(temp):

print(80 \* "=")

print(temp)

print(80 \* "=")

def main(flag, load\_existing\_dump=False):

highlight\_string("INITIALIZING")

print("loading data..")

dataset = load\_datasets(load\_existing\_dump)

config = dataset.model\_config

print("word vocab Size: {}".format(len(dataset.word2idx)))

print("pos vocab Size: {}".format(len(dataset.pos2idx)))

print("dep vocab Size: {}".format(len(dataset.dep2idx)))

print("Training Size: {}".format(len(dataset.train\_inputs[0])))

print("valid data Size: {}".format(len(dataset.valid\_data)))

print("test data Size: {}".format(len(dataset.test\_data)))

print(len(dataset.word2idx), len(dataset.word\_embedding\_matrix))

print(len(dataset.pos2idx), len(dataset.pos\_embedding\_matrix))

print(len(dataset.dep2idx), len(dataset.dep\_embedding\_matrix))

if not os.path.exists(os.path.join(DataConfig.data\_dir\_path, DataConfig.model\_dir)):

os.makedirs(os.path.join(

DataConfig.data\_dir\_path, DataConfig.model\_dir))

with tf.Graph().as\_default(), tf.compat.v1.Session() as sess:

print("Building network...",)

start = time.time()

with tf.compat.v1.variable\_scope("model") as model\_scope:

model = ParserModel(config, dataset.word\_embedding\_matrix, dataset.pos\_embedding\_matrix,

dataset.dep\_embedding\_matrix)

saver = tf.compat.v1.train.Saver()

"""

model\_scope.reuse\_variables()

-> no need to call tf.compat.v1.variable\_scope(model\_scope, reuse = True) again

-> directly access variables & call functions inside this block itself.

-> ref: https://www.tensorflow.org/versions/r1.2/api\_docs/python/tf/variable\_scope

-> https://stackoverflow.com/questions/35919020/whats-the-difference-of-name-scope-and-a-variable-scope-in-tensorflow

"""

print("took {:.2f} seconds\n".format(time.time() - start))

merged = tf.compat.v1.summary.merge\_all()

train\_writer = tf.compat.v1.summary.FileWriter(os.path.join(DataConfig.data\_dir\_path, DataConfig.summary\_dir,

DataConfig.train\_summ\_dir), sess.graph)

valid\_writer = tf.compat.v1.summary.FileWriter(os.path.join(DataConfig.data\_dir\_path, DataConfig.summary\_dir,

DataConfig.test\_summ\_dir))

if flag == Flags.TRAIN:

# Variable initialization -> not needed for .restore()

""" The variables to restore do not have to have been initialized,

as restoring is itself a way to initialize variables. """

sess.run(tf.compat.v1.global\_variables\_initializer())

""" call 'assignment' after 'init' only, else 'assignment' will get reset by 'init' """

sess.run(tf.compat.v1.assign(model.word\_embedding\_matrix, model.word\_embeddings))

sess.run(tf.compat.v1.assign(model.pos\_embedding\_matrix, model.pos\_embeddings))

sess.run(tf.compat.v1.assign(model.dep\_embedding\_matrix, model.dep\_embeddings))

highlight\_string("TRAINING")

model.print\_trainable\_varibles()

model.fit(sess, saver, config, dataset,

train\_writer, valid\_writer, merged)

# Testing

highlight\_string("Testing")

print("Restoring best found parameters on dev set")

saver.restore(sess, os.path.join(DataConfig.data\_dir\_path, DataConfig.model\_dir,

DataConfig.model\_name))

model.compute\_dependencies(sess, dataset.test\_data, dataset)

test\_UAS = model.get\_UAS(dataset.test\_data)

print("test UAS: {}".format(test\_UAS \* 100))

train\_writer.close()

valid\_writer.close()

# visualize trained embeddings after complete training (not after each epoch)

with tf.compat.v1.variable\_scope(model\_scope, reuse=True):

pos\_emb = tf.compat.v1.get\_variable("feature\_lookup/pos\_embedding\_matrix",

[len(dataset.pos2idx.keys()), dataset.model\_config.embedding\_dim])

visualize\_sample\_embeddings(sess, os.path.join(DataConfig.data\_dir\_path, DataConfig.model\_dir),

dataset.pos2idx.keys(), dataset.pos2idx, pos\_emb)

print("to Visualize Embeddings, run in terminal:")

print("tensorboard --logdir=" + os.path.abspath(os.path.join(DataConfig.data\_dir\_path,

DataConfig.model\_dir)))

else:

ckpt\_path = tf.compat.v1.train.latest\_checkpoint(os.path.join(DataConfig.data\_dir\_path,

DataConfig.model\_dir))

if ckpt\_path is not None:

print("Found checkpoint! Restoring variables..")

saver.restore(sess, ckpt\_path)

highlight\_string("Testing")

model.compute\_dependencies(sess, dataset.test\_data, dataset)

test\_UAS = model.get\_UAS(dataset.test\_data)

print("test UAS: {}".format(test\_UAS \* 100))

# model.run\_valid\_epoch(sess, dataset.valid\_data, dataset)

# valid\_UAS = model.get\_UAS(dataset.valid\_data)

# print "valid UAS: {}".format(valid\_UAS \* 100)

highlight\_string("Embedding Visualization")

with tf.compat.v1.variable\_scope(model\_scope, reuse=True):

pos\_emb = tf.compat.v1.get\_variable("feature\_lookup/pos\_embedding\_matrix",

[len(dataset.pos2idx.keys()), dataset.model\_config.embedding\_dim])

visualize\_sample\_embeddings(sess, os.path.join(DataConfig.data\_dir\_path, DataConfig.model\_dir),

dataset.pos2idx.keys(), dataset.pos2idx, pos\_emb)

print("to Visualize Embeddings, run in terminal:")

print("tensorboard --logdir=" + os.path.abspath(os.path.join(DataConfig.data\_dir\_path,

DataConfig.model\_dir)))

else:

print("No checkpoint found!")

if \_\_name\_\_ == '\_\_main\_\_':

main(Flags.TEST, load\_existing\_dump=True)

### feature\_extraction.py

import os

import numpy as np

import datetime

from enum import Enum

from general\_utils import get\_pickle, dump\_pickle, get\_vocab\_dict

NULL = "<null>"

UNK = "<unk>"

ROOT = "<root>"

pos\_prefix = "<p>:"

dep\_prefix = "<d>:"

punc\_pos = ["''", "``", ":", ".", ","]

today\_date = str(datetime.datetime.now().date())

class DataConfig: # data, embedding, model path etc.

# Data Paths

data\_dir\_path = "./data"

train\_path = "train.conll"

valid\_path = "dev.conll"

test\_path = "test.conll"

# embedding

embedding\_file = "en-cw.txt"

# model saver

model\_dir = "params\_" + today\_date

model\_name = "parser.weights"

# summary

summary\_dir = "params\_" + today\_date

train\_summ\_dir = "train\_summaries"

test\_summ\_dir = "valid\_summaries"

# dump - vocab

dump\_dir = "./data/dump"

word\_vocab\_file = "word2idx.pkl"

pos\_vocab\_file = "pos2idx.pkl"

dep\_vocab\_file = "dep2idx.pkl"

# dump - embedding

word\_emb\_file = "word\_emb.pkl" # 2d array

pos\_emb\_file = "pos\_emb.pkl" # 2d array

dep\_emb\_file = "dep\_emb.pkl" # 2d array

class ModelConfig(object): # Takes care of shape, dimensions used for tf model

# Input

word\_features\_types = None

pos\_features\_types = None

dep\_features\_types = None

num\_features\_types = None

embedding\_dim = 50

# hidden\_size

l1\_hidden\_size = 200

l2\_hidden\_size = 15

# output

num\_classes = 3

# Vocab

word\_vocab\_size = None

pos\_vocab\_size = None

dep\_vocab\_size = None

# num\_epochs

n\_epochs = 0

# batch\_size

batch\_size = 2048

# dropout

keep\_prob = 0.5

reg\_val = 1e-8

# learning\_rate

lr = 0.001

# load existing vocab

load\_existing\_vocab = False

# summary

write\_summary\_after\_epochs = 1

# valid run

run\_valid\_after\_epochs = 1

class SettingsConfig: # enabling and disabling features, feature types

# Features

use\_word = True

use\_pos = True

use\_dep = True

is\_lower = True

class Flags(Enum):

TRAIN = 1

VALID = 2

TEST = 3

class Token(object):

def \_\_init\_\_(self, token\_id, word, pos, dep, head\_id):

self.token\_id = token\_id # token index

self.word = word.lower() if SettingsConfig.is\_lower else word

self.pos = pos\_prefix + pos

self.dep = dep\_prefix + dep

self.head\_id = head\_id # head token index

self.predicted\_head\_id = None

self.left\_children = list()

self.right\_children = list()

def is\_root\_token(self):

if self.word == ROOT:

return True

return False

def is\_null\_token(self):

if self.word == NULL:

return True

return False

def is\_unk\_token(self):

if self.word == UNK:

return True

return False

def reset\_predicted\_head\_id(self):

self.predicted\_head\_id = None

NULL\_TOKEN = Token(-1, NULL, NULL, NULL, -1)

ROOT\_TOKEN = Token(-1, ROOT, ROOT, ROOT, -1)

UNK\_TOKEN = Token(-1, UNK, UNK, UNK, -1)

class Sentence(object):

def \_\_init\_\_(self, tokens):

self.Root = Token(-1, ROOT, ROOT, ROOT, -1)

self.tokens = tokens

self.buff = [token for token in self.tokens]

self.stack = [self.Root]

self.dependencies = []

self.predicted\_dependencies = []

def load\_gold\_dependency\_mapping(self):

for token in self.tokens:

if token.head\_id != -1:

token.parent = self.tokens[token.head\_id]

if token.head\_id > token.token\_id:

token.parent.left\_children.append(token.token\_id)

else:

token.parent.right\_children.append(token.token\_id)

else:

token.parent = self.Root

for token in self.tokens:

token.left\_children.sort()

token.right\_children.sort()

def update\_child\_dependencies(self, curr\_transition):

if curr\_transition == 0:

head = self.stack[-1]

dependent = self.stack[-2]

elif curr\_transition == 1:

head = self.stack[-2]

dependent = self.stack[-1]

if head.token\_id > dependent.token\_id:

head.left\_children.append(dependent.token\_id)

head.left\_children.sort()

else:

head.right\_children.append(dependent.token\_id)

head.right\_children.sort()

# dependent.head\_id = head.token\_id

def get\_child\_by\_index\_and\_depth(self, token, index, direction, depth): # Get child token

if depth == 0:

return token

if direction == "left":

if len(token.left\_children) > index:

return self.get\_child\_by\_index\_and\_depth(

self.tokens[token.left\_children[index]], index, direction, depth - 1)

return NULL\_TOKEN

else:

if len(token.right\_children) > index:

return self.get\_child\_by\_index\_and\_depth(

self.tokens[token.right\_children[::-1][index]], index, direction, depth - 1)

return NULL\_TOKEN

def get\_legal\_labels(self):

labels = ([1] if len(self.stack) > 2 else [0])

labels += ([1] if len(self.stack) >= 2 else [0])

labels += [1] if len(self.buff) > 0 else [0]

return labels

def get\_transition\_from\_current\_state(self): # logic to get next transition

if len(self.stack) < 2:

return 2 # shift

stack\_token\_0 = self.stack[-1]

stack\_token\_1 = self.stack[-2]

if stack\_token\_1.token\_id >= 0 and stack\_token\_1.head\_id == stack\_token\_0.token\_id: # left arc

return 0

elif stack\_token\_1.token\_id >= -1 and stack\_token\_0.head\_id == stack\_token\_1.token\_id \

and stack\_token\_0.token\_id not in map(lambda x: x.head\_id, self.buff):

return 1 # right arc

else:

return 2 if len(self.buff) != 0 else None

def update\_state\_by\_transition(self, transition, gold=True): # updates stack, buffer and dependencies

if transition is not None:

if transition == 2: # shift

self.stack.append(self.buff[0])

self.buff = self.buff[1:] if len(self.buff) > 1 else []

elif transition == 0: # left arc

self.dependencies.append(

(self.stack[-1], self.stack[-2])) if gold else self.predicted\_dependencies.append(

(self.stack[-1], self.stack[-2]))

self.stack = self.stack[:-2] + self.stack[-1:]

elif transition == 1: # right arc

self.dependencies.append(

(self.stack[-2], self.stack[-1])) if gold else self.predicted\_dependencies.append(

(self.stack[-2], self.stack[-1]))

self.stack = self.stack[:-1]

def reset\_to\_initial\_state(self):

self.buff = [token for token in self.tokens]

self.stack = [self.Root]

def clear\_prediction\_dependencies(self):

self.predicted\_dependencies = []

def clear\_children\_info(self):

for token in self.tokens:

token.left\_children = []

token.right\_children = []

class Dataset(object):

def \_\_init\_\_(self, model\_config, train\_data, valid\_data, test\_data, feature\_extractor):

self.model\_config = model\_config

self.train\_data = train\_data

self.valid\_data = valid\_data

self.test\_data = test\_data

self.feature\_extractor = feature\_extractor

# Vocab

self.word2idx = None

self.idx2word = None

self.pos2idx = None

self.idx2pos = None

self.dep2idx = None

self.idx2dep = None

# Embedding Matrix

self.word\_embedding\_matrix = None

self.pos\_embedding\_matrix = None

self.dep\_embedding\_matrix = None

# input & outputs

self.train\_inputs, self.train\_targets = None, None

self.valid\_inputs, self.valid\_targets = None, None

self.test\_inputs, self.test\_targets = None, None

def build\_vocab(self):

all\_words = set()

all\_pos = set()

all\_dep = set()

for sentence in self.train\_data:

all\_words.update(set(map(lambda x: x.word, sentence.tokens)))

all\_pos.update(set(map(lambda x: x.pos, sentence.tokens)))

all\_dep.update(set(map(lambda x: x.dep, sentence.tokens)))

all\_words.add(ROOT\_TOKEN.word)

all\_words.add(NULL\_TOKEN.word)

all\_words.add(UNK\_TOKEN.word)

all\_pos.add(ROOT\_TOKEN.pos)

all\_pos.add(NULL\_TOKEN.pos)

all\_pos.add(UNK\_TOKEN.pos)

all\_dep.add(ROOT\_TOKEN.dep)

all\_dep.add(NULL\_TOKEN.dep)

all\_dep.add(UNK\_TOKEN.dep)

word\_vocab = list(all\_words)

pos\_vocab = list(all\_pos)

dep\_vocab = list(all\_dep)

word2idx = get\_vocab\_dict(word\_vocab)

idx2word = {idx: word for (word, idx) in word2idx.items()}

pos2idx = get\_vocab\_dict(pos\_vocab)

idx2pos = {idx: pos for (pos, idx) in pos2idx.items()}

dep2idx = get\_vocab\_dict(dep\_vocab)

idx2dep = {idx: dep for (dep, idx) in dep2idx.items()}

self.word2idx = word2idx

self.idx2word = idx2word

self.pos2idx = pos2idx

self.idx2pos = idx2pos

self.dep2idx = dep2idx

self.idx2dep = idx2dep

def build\_embedding\_matrix(self):

# load word vectors

word\_vectors = {}

embedding\_lines = open(os.path.join(DataConfig.data\_dir\_path, DataConfig.embedding\_file), "r").readlines()

for line in embedding\_lines:

sp = line.strip().split()

word\_vectors[sp[0]] = [float(x) for x in sp[1:]]

# word embedding

self.model\_config.word\_vocab\_size = len(self.word2idx)

word\_embedding\_matrix = np.asarray(

np.random.normal(0, 0.9, size=(self.model\_config.word\_vocab\_size, self.model\_config.embedding\_dim)),

dtype=np.float32)

for (word, idx) in self.word2idx.items():

if word in word\_vectors:

word\_embedding\_matrix[idx] = word\_vectors[word]

elif word.lower() in word\_vectors:

word\_embedding\_matrix[idx] = word\_vectors[word.lower()]

self.word\_embedding\_matrix = word\_embedding\_matrix

# pos embedding

self.model\_config.pos\_vocab\_size = len(self.pos2idx)

pos\_embedding\_matrix = np.asarray(

np.random.normal(0, 0.9, size=(self.model\_config.pos\_vocab\_size, self.model\_config.embedding\_dim)),

dtype=np.float32)

self.pos\_embedding\_matrix = pos\_embedding\_matrix

# dep embedding

self.model\_config.dep\_vocab\_size = len(self.dep2idx)

dep\_embedding\_matrix = np.asarray(

np.random.normal(0, 0.9, size=(self.model\_config.dep\_vocab\_size, self.model\_config.embedding\_dim)),

dtype=np.float32)

self.dep\_embedding\_matrix = dep\_embedding\_matrix

def convert\_data\_to\_ids(self):

self.train\_inputs, self.train\_targets = self.feature\_extractor. \

create\_instances\_for\_data(self.train\_data, self.word2idx, self.pos2idx, self.dep2idx)

# self.valid\_inputs, self.valid\_targets = self.feature\_extractor.\

# create\_instances\_for\_data(self.valid\_data, self.word2idx)

# self.test\_inputs, self.test\_targets = self.feature\_extractor.\

# create\_instances\_for\_data(self.test\_data, self.word2idx)

def add\_to\_vocab(self, words, prefix=""):

idx = len(self.word2idx)

for token in words:

if prefix + token not in self.word2idx:

self.word2idx[prefix + token] = idx

self.idx2word[idx] = prefix + token

idx += 1

class FeatureExtractor(object):

def \_\_init\_\_(self, model\_config):

self.model\_config = model\_config

def extract\_from\_stack\_and\_buffer(self, sentence, num\_words=3):

tokens = []

tokens.extend([NULL\_TOKEN for \_ in range(num\_words - len(sentence.stack))])

tokens.extend(sentence.stack[-num\_words:])

tokens.extend(sentence.buff[:num\_words])

tokens.extend([NULL\_TOKEN for \_ in range(num\_words - len(sentence.buff))])

return tokens # 6 features

def extract\_children\_from\_stack(self, sentence, num\_stack\_words=2):

children\_tokens = []

for i in range(num\_stack\_words):

if len(sentence.stack) > i:

lc0 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 0, "left", 1)

rc0 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 0, "right", 1)

lc1 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 1, "left",

1) if lc0 != NULL\_TOKEN else NULL\_TOKEN

rc1 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 1, "right",

1) if rc0 != NULL\_TOKEN else NULL\_TOKEN

llc0 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 0, "left",

2) if lc0 != NULL\_TOKEN else NULL\_TOKEN

rrc0 = sentence.get\_child\_by\_index\_and\_depth(sentence.stack[-i - 1], 0, "right",

2) if rc0 != NULL\_TOKEN else NULL\_TOKEN

children\_tokens.extend([lc0, rc0, lc1, rc1, llc0, rrc0])

else:

[children\_tokens.append(NULL\_TOKEN) for \_ in range(6)]

return children\_tokens # 12 features

def extract\_for\_current\_state(self, sentence, word2idx, pos2idx, dep2idx):

direct\_tokens = self.extract\_from\_stack\_and\_buffer(sentence, num\_words=3)

children\_tokens = self.extract\_children\_from\_stack(sentence, num\_stack\_words=2)

word\_features = []

pos\_features = []

dep\_features = []

# Word features -> 18

word\_features.extend(map(lambda x: x.word, direct\_tokens))

word\_features.extend(map(lambda x: x.word, children\_tokens))

# pos features -> 18

pos\_features.extend(map(lambda x: x.pos, direct\_tokens))

pos\_features.extend(map(lambda x: x.pos, children\_tokens))

# dep features -> 12 (only children)

dep\_features.extend(map(lambda x: x.dep, children\_tokens))

word\_input\_ids = [word2idx[word] if word in word2idx else word2idx[UNK\_TOKEN.word] for word in word\_features]

pos\_input\_ids = [pos2idx[pos] if pos in pos2idx else pos2idx[UNK\_TOKEN.pos] for pos in pos\_features]

dep\_input\_ids = [dep2idx[dep] if dep in dep2idx else dep2idx[UNK\_TOKEN.dep] for dep in dep\_features]

return [word\_input\_ids, pos\_input\_ids, dep\_input\_ids] # 48 features

def create\_instances\_for\_data(self, data, word2idx, pos2idx, dep2idx):

lables = []

word\_inputs = []

pos\_inputs = []

dep\_inputs = []

for i, sentence in enumerate(data):

num\_words = len(sentence.tokens)

for \_ in range(num\_words \* 2):

word\_input, pos\_input, dep\_input = self.extract\_for\_current\_state(sentence, word2idx, pos2idx, dep2idx)

legal\_labels = sentence.get\_legal\_labels()

curr\_transition = sentence.get\_transition\_from\_current\_state()

if curr\_transition is None:

break

assert legal\_labels[curr\_transition] == 1

# Update left/right children

if curr\_transition != 2:

sentence.update\_child\_dependencies(curr\_transition)

sentence.update\_state\_by\_transition(curr\_transition)

lables.append(curr\_transition)

word\_inputs.append(word\_input)

pos\_inputs.append(pos\_input)

dep\_inputs.append(dep\_input)

else:

sentence.reset\_to\_initial\_state()

# reset stack and buffer to default state

sentence.reset\_to\_initial\_state()

targets = np.zeros((len(lables), self.model\_config.num\_classes), dtype=np.int32)

targets[np.arange(len(targets)), lables] = 1

return [word\_inputs, pos\_inputs, dep\_inputs], targets

class DataReader(object):

def \_\_init\_\_(self):

print("A")

def read\_conll(self, token\_lines):

tokens = []

for each in token\_lines:

fields = each.strip().split("\t")

token\_index = int(fields[0]) - 1

word = fields[1]

pos = fields[4]

dep = fields[7]

head\_index = int(fields[6]) - 1

token = Token(token\_index, word, pos, dep, head\_index)

tokens.append(token)

sentence = Sentence(tokens)

# sentence.load\_gold\_dependency\_mapping()

return sentence

def read\_data(self, data\_lines):

data\_objects = []

token\_lines = []

for token\_conll in data\_lines:

token\_conll = token\_conll.strip()

if len(token\_conll) > 0:

token\_lines.append(token\_conll)

else:

data\_objects.append(self.read\_conll(token\_lines))

token\_lines = []

if len(token\_lines) > 0:

data\_objects.append(self.read\_conll(token\_lines))

return data\_objects

def load\_datasets(load\_existing\_dump=False):

model\_config = ModelConfig()

data\_reader = DataReader()

train\_lines = open(os.path.join(DataConfig.data\_dir\_path, DataConfig.train\_path), "r").readlines()

valid\_lines = open(os.path.join(DataConfig.data\_dir\_path, DataConfig.valid\_path), "r").readlines()

test\_lines = open(os.path.join(DataConfig.data\_dir\_path, DataConfig.test\_path), "r").readlines()

# Load data

train\_data = data\_reader.read\_data(train\_lines)

print ("Loaded Train data")

valid\_data = data\_reader.read\_data(valid\_lines)

print ("Loaded Dev data")

test\_data = data\_reader.read\_data(test\_lines)

print ("Loaded Test data")

feature\_extractor = FeatureExtractor(model\_config)

dataset = Dataset(model\_config, train\_data, valid\_data, test\_data, feature\_extractor)

# Vocab processing

if load\_existing\_dump:

dataset.word2idx = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.word\_vocab\_file))

dataset.idx2word = {idx: word for (word, idx) in dataset.word2idx.items()}

dataset.pos2idx = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.pos\_vocab\_file))

dataset.idx2pos = {idx: pos for (pos, idx) in dataset.pos2idx.items()}

dataset.dep2idx = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.dep\_vocab\_file))

dataset.idx2dep = {idx: dep for (dep, idx) in dataset.dep2idx.items()}

dataset.model\_config.load\_existing\_vocab = True

print("loaded existing Vocab!")

dataset.word\_embedding\_matrix = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.word\_emb\_file))

dataset.pos\_embedding\_matrix = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.pos\_emb\_file))

dataset.dep\_embedding\_matrix = get\_pickle(os.path.join(DataConfig.dump\_dir, DataConfig.dep\_emb\_file))

print( "loaded existing embedding matrix!")

else:

dataset.build\_vocab()

dump\_pickle(dataset.word2idx, os.path.join(DataConfig.dump\_dir, DataConfig.word\_vocab\_file))

dump\_pickle(dataset.pos2idx, os.path.join(DataConfig.dump\_dir, DataConfig.pos\_vocab\_file))

dump\_pickle(dataset.dep2idx, os.path.join(DataConfig.dump\_dir, DataConfig.dep\_vocab\_file))

dataset.model\_config.load\_existing\_vocab = True

print ("Vocab Build Done!")

dataset.build\_embedding\_matrix()

print ("embedding matrix Build Done")

dump\_pickle(dataset.word\_embedding\_matrix, os.path.join(DataConfig.dump\_dir, DataConfig.word\_emb\_file))

dump\_pickle(dataset.pos\_embedding\_matrix, os.path.join(DataConfig.dump\_dir, DataConfig.pos\_emb\_file))

dump\_pickle(dataset.dep\_embedding\_matrix, os.path.join(DataConfig.dump\_dir, DataConfig.dep\_emb\_file))

print ("converting data into ids..")

dataset.convert\_data\_to\_ids()

print ("Done!")

dataset.model\_config.word\_features\_types = len(dataset.train\_inputs[0][0])

dataset.model\_config.pos\_features\_types = len(dataset.train\_inputs[1][0])

dataset.model\_config.dep\_features\_types = len(dataset.train\_inputs[2][0])

dataset.model\_config.num\_features\_types = dataset.model\_config.word\_features\_types + \

dataset.model\_config.pos\_features\_types + dataset.model\_config.dep\_features\_types

dataset.model\_config.num\_classes = len(dataset.train\_targets[0])

return dataset