華中科技大學

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

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。

1.1.1 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。

1.1.2 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。

1.1.3 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

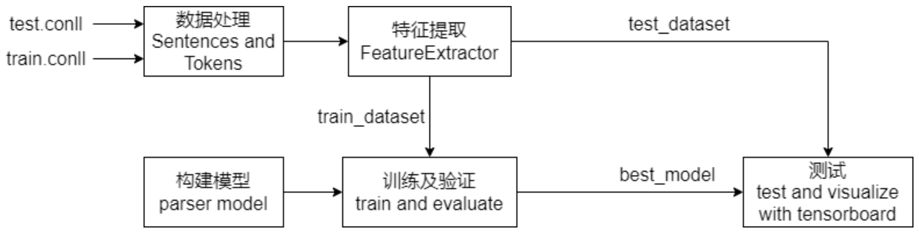


图 1-1 依存句法分析算法模块

1.3 系统实现

1.3.1 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：

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)          ; choose a transition operator to apply
    state ← APPLY(t, state)  ; apply it, creating a new state
  return state
```

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。

1.3.2 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：

```
if   $(S_1\ r\ S_2) \in R_p$   选择LEFT-ARC
if   $(S_1\ r\ S_2) \in R_p and \forall r', w$满足$(S_1\ r'\ w) \in R_p$有$(S_1\ r'\ w) \in R_c$   选择RIGHT-ARC
else 选择SHIFT
```

图 1-3 oracle 创建算法

1.3.3 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

1.3.4 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）。

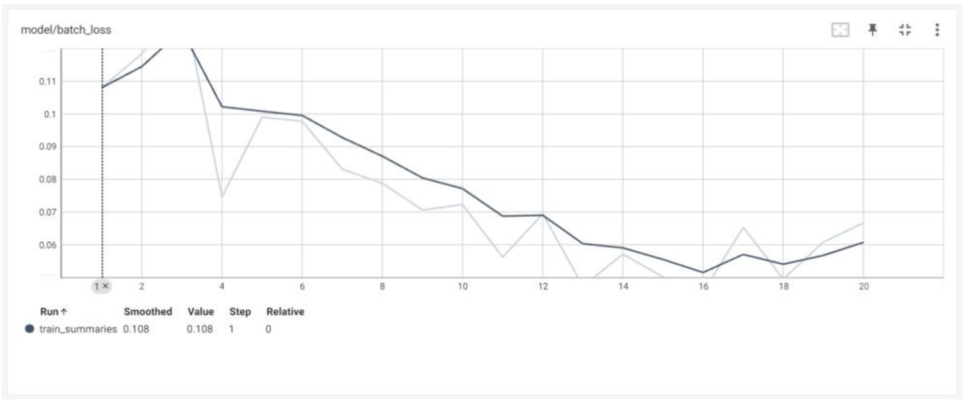
构建模型时，使用立方函数作为激活函数，使用交叉熵作为损失函数。同时，使用 Bi-LSTM 改进解析器的效果。最后对模型进行训练，并根据在验证集上的效果更新 epoch、batch_size 等超参数。

1.3.5 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。

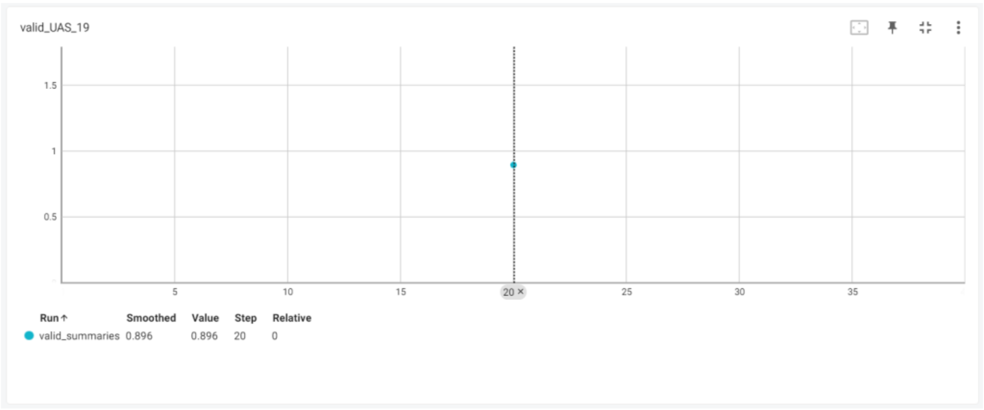
训练过程中的 batch_loss 下降过程如下：



Run	Smoothed	Value	Step	Relative
● train_summaries	0.108	0.108	1	0

图 1-4 batch_loss 图

最后一轮的 UAS 效果如下：



Run	Smoothed	Value	Step	Relative
● valid_summaries	0.896	0.896	20	0

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：

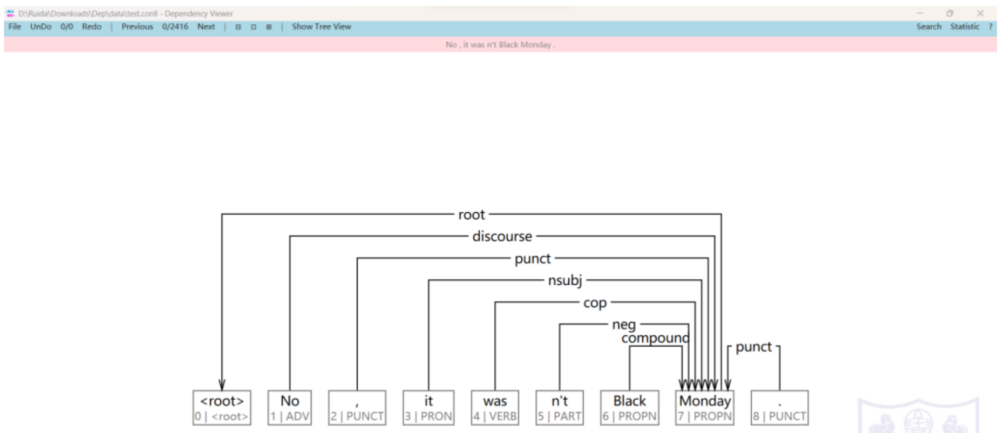


图 1-6 依存语法树可视化

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-variabl
e-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
```