```
import sys
from collections import defaultdict


class DependencyEdge(object):
    """
    Represent a single dependency edge:
    """

    def __init__(self, ident, word, pos, head, deprel):
        self.id = ident
        self.word = word
        self.pos = pos
        self.head = head
        self.deprel = deprel

    def print_conll(self):
        return "{d.id}\t{d.word}\t_\t_\t{d.pos}\t_\t{d.head}\t{d.deprel}\t_\t_".format(d=self)


def parse_conll_relation(s):
    fields = s.split('\t')
    ident_s, word, lemma, upos, pos, feats, head_s, deprel, deps, misc = fields
    ident = int(ident_s)
    head = int(head_s)
    return DependencyEdge(ident, word, pos, head, deprel)


class DependencyStructure(object):

    def __init__(self):
        self.deprels = {}
        self.root = None
        self.parent_to_children = defaultdict(list)

    def add_deprel(self, deprel):
        self.deprels[deprel.id] = deprel
        self.parent_to_children[deprel.head].append(deprel.id)
        if deprel.head == 0:
            self.root = deprel.id

    def __str__(self):
        for k,v in self.deprels.items():
            print(v)

    def print_tree(self, parent = None):
        if not parent:
            return self.print_tree(parent = self.root)

        if self.deprels[parent].head == parent:
            return self.deprels[parent].word

        children = [self.print_tree(child) for child in self.parent_to_children[parent]]
        child_str = " ".join(children)
        return("({} {})".format(self.deprels[parent].word, child_str))

    def words(self):
        return [None]+[x.word for (i,x) in self.deprels.items()]

    def pos(self):
        return [None]+[x.pos for (i,x) in self.deprels.items()]

    def print_conll(self):
        deprels = [v for (k,v) in  sorted(self.deprels.items())]
        return "\n".join(deprel.print_conll() for deprel in deprels)


def conll_reader(input_file):
    current_deps = DependencyStructure()
    while True:
        line = input_file.readline().strip()
        if not line and current_deps:
            yield current_deps
            current_deps = DependencyStructure()
            line = input_file.readline().strip()
            if not line:
                break
```

```
            current_deps.add_deprel(parse_conll_relation(line))


    def get_vocabularies(conll_reader):
        word_set = defaultdict(int)
        pos_set = set()
        for dtree in conll_reader:
            for ident, node in dtree.deprels.items():
                if node.pos != "CD" and node.pos!="NNP":
                    word_set[node.word.lower()] += 1
                pos_set.add(node.pos)

        word_set = set(x for x in word_set if word_set[x] > 1)

        word_list = ["<CD>","<NNP>","<UNK>","<ROOT>","<NULL>"] + list(word_set)
        pos_list =  ["<UNK>","<ROOT>","<NULL>"] + list(pos_set)

        return word_list, pos_list


    with open('data/train.conll','r') as in_file, open('data/words.vocab','w') as word_file, open('data/pos.vocab','w') as pos_file:
        word_list, pos_list = get_vocabularies(conll_reader(in_file))
        print("Writing word indices...")
        for index, word in enumerate(word_list):
            word_file.write("{}\t{}\n".format(word, index))
        print("Writing POS indices...")
        for index, pos in enumerate(pos_list):
            pos_file.write("{}\t{}\n".format(pos, index))


        word_list, pos_list = get_vocabularies(conll_reader(in_file))
        print("Writing word indices...")
        for index, word in enumerate(word_list):
            word_file.write("{}\t{}\n".format(word, index))
        print("Writing POS indices...")
        for index, pos in enumerate(pos_list):
            pos_file.write("{}\t{}\n".format(pos, index))
```

```
 Writing word indices...
 Writing POS indices...
 Writing word indices...
 Writing POS indices...
```

```
import copy
import keras
import numpy as np

class State(object):
    def __init__(self, sentence = []):
        self.stack = []
        self.buffer = []
        if sentence:
            self.buffer = list(reversed(sentence))
        self.deps = set()

    def shift(self):
        self.stack.append(self.buffer.pop())

    def left_arc(self, label):
        self.deps.add( (self.buffer[-1], self.stack.pop(),label) )

    def right_arc(self, label):
        parent = self.stack.pop()
        self.deps.add( (parent, self.buffer.pop(), label) )
        self.buffer.append(parent)

    def __repr__(self):
        return "{},{},{}".format(self.stack, self.buffer, self.deps)



def apply_sequence(seq, sentence):
    state = State(sentence)
    for rel, label in seq:
        if rel == "shift":
            state.shift()
        elif rel == "left_arc":
            state.left_arc(label)
        elif rel == "right_arc":
            state.right_arc(label)

    return state.deps

class RootDummy(object):
    def __init__(self):
        self.head = None
        self.id = 0
        self.deprel = None
    def __repr__(self):
        return "<ROOT>"


def get_training_instances(dep_structure):

    deprels = dep_structure.deprels

    sorted_nodes = [k for k,v in sorted(deprels.items())]
    state = State(sorted_nodes)
    state.stack.append(0)

    childcount = defaultdict(int)
    for ident,node in deprels.items():
        childcount[node.head] += 1

    seq = []
    while state.buffer:
        if not state.stack:
            seq.append((copy.deepcopy(state),("shift",None)))
            state.shift()
            continue
        if state.stack[-1] == 0:
            stackword = RootDummy()
        else:
            stackword = deprels[state.stack[-1]]
        bufferword = deprels[state.buffer[-1]]
        if stackword.head == bufferword.id:
            childcount[bufferword.id]-=1
            seq.append((copy.deepcopy(state),("left_arc",stackword.deprel)))
            state.left_arc(stackword.deprel)
        elif bufferword.head == stackword.id and childcount[bufferword.id] == 0:
```

```
            childcount[stackword.id]-=1
            seq.append((copy.deepcopy(state),("right_arc",bufferword.deprel)))
            state.right_arc(bufferword.deprel)
        else:
            seq.append((copy.deepcopy(state),("shift",None)))
            state.shift()
    return seq


dep_relations = ['tmod', 'vmod', 'csubjpass', 'rcmod', 'ccomp', 'poss', 'parataxis', 'appos', 'dep', 'iobj', 'pobj', 'mwe', 'quantmod', 'acc


class FeatureExtractor(object):

    def __init__(self, word_vocab_file, pos_vocab_file):
        self.word_vocab = self.read_vocab(word_vocab_file)
        self.pos_vocab = self.read_vocab(pos_vocab_file)
        self.output_labels = self.make_output_labels()

    def make_output_labels(self):
        labels = []
        labels.append(('shift',None))

        for rel in dep_relations:
            labels.append(("left_arc",rel))
            labels.append(("right_arc",rel))
        return dict((label, index) for (index,label) in enumerate(labels))

    def read_vocab(self,vocab_file):
        vocab = {}
        for line in vocab_file:
            word, index_s = line.strip().split()
            index = int(index_s)
            vocab[word] = index
        return vocab

    def get_input_representation(self, words, pos, state):
        size_s = len(state.stack)
        size_b = len(state.buffer)
        stack = []
        buffer = []

        if size_b >= 3:
            indcs = state.buffer[::-1][0:3]
            for i in indcs:
                if words[i] not in self.word_vocab.keys() and words[i] != None:
                    buffer.append(self.pos_vocab[pos[i]])
                elif words[i] == None:
                    buffer.append(3)
                else:
                    buffer.append(self.word_vocab[words[i]])
        else:
            indcs = state.buffer[::-1][0:size_b]
            for i in indcs:
                if words[i] not in self.word_vocab.keys() and words[i] != None:
                    buffer.append(self.pos_vocab[pos[i]])
                elif words[i] == None:
                    buffer.append(3)
                else:
                    buffer.append(self.word_vocab[words[i]])
            for i in range(3-size_b):
                buffer.append(4)

        if size_s >= 3:
            indcs = state.stack[::-1][0:3]
            for i in indcs:
                if words[i] not in self.word_vocab.keys() and words[i] != None:
                    stack.append(self.pos_vocab[pos[i]])
                elif words[i] == None:
                    stack.append(3)
                else:
                    stack.append(self.word_vocab[words[i]])
        else:
            indcs = state.stack[::-1][0:size_s]
            for i in indcs:
                if words[i] not in self.word_vocab.keys() and words[i] != None:
                    stack.append(self.pos_vocab[pos[i]])
```

```
                elif words[i] == None:
                    stack.append(3)
                else:
                    stack.append(self.word_vocab[words[i]])
            for i in range(3-size_s):
                stack.append(4)

        input = np.array(stack+buffer)

        return input


    def get_output_representation(self, output_pair):
        out = np.zeros(91)
        if  output_pair[0] == "shift":
            out[90] = 1
        elif output_pair[0] == 'left_arc':
            out[dep_relations.index(output_pair[1])] = 1
        else:
            out[dep_relations.index(output_pair[1])+45] = 1
        return out



def get_training_matrices(extractor, in_file):
    inputs = []
    outputs = []
    count = 0
    for dtree in conll_reader(in_file):
        words = dtree.words()
        pos = dtree.pos()
        for state, output_pair in get_training_instances(dtree):
            inputs.append(extractor.get_input_representation(words, pos, state))
            outputs.append(extractor.get_output_representation(output_pair))
        if count%100 == 0:
            sys.stdout.write(".")
            sys.stdout.flush()
        count += 1
    sys.stdout.write("\n")
    return np.vstack(inputs),np.vstack(outputs)



WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

try:
    word_vocab_f = open(WORD_VOCAB_FILE,'r')
    pos_vocab_f = open(POS_VOCAB_FILE,'r')
except FileNotFoundError:
    print("Could not find vocabulary files {} and {}".format(WORD_VOCAB_FILE, POS_VOCAB_FILE))
    sys.exit(1)


with open('data/train.conll','r') as in_file:

    extractor = FeatureExtractor(word_vocab_f, pos_vocab_f)
    print("Starting feature extraction... (each . represents 100 sentences)")
    inputs, outputs = get_training_matrices(extractor,in_file)
    print("Writing output...")
    np.save('data/input_train.npy', inputs)
    np.save('data/target_train.npy', outputs)
```

```
Starting feature extraction... (each . represents 100 sentences)
.....................................................................................................................................
Writing output...
```

```
from keras import Sequential
from keras.layers import Flatten, Embedding, Dense, Bidirectional, LSTM

def build_model(word_types, pos_types, outputs):
    model = Sequential()
    model.add(Embedding(input_dim=word_types, input_length=6, output_dim=32))
    model.add(Flatten())
    model.add(Dense(units=10, activation='relu'))
    model.add(Dense(units=100, activation='relu'))
    model.add(Dense(outputs, activation=keras.activations.softmax))
    model.compile(keras.optimizers.legacy.Adam(learning_rate=0.01), loss="categorical_crossentropy")
    return model



WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

try:
    word_vocab_f = open(WORD_VOCAB_FILE,'r')
    pos_vocab_f = open(POS_VOCAB_FILE,'r')
except FileNotFoundError:
    print("Could not find vocabulary files {} and {}".format(WORD_VOCAB_FILE, POS_VOCAB_FILE))
    sys.exit(1)

extractor = FeatureExtractor(word_vocab_f, pos_vocab_f)
print("Compiling model.")
model = build_model(len(extractor.word_vocab), len(extractor.pos_vocab), len(extractor.output_labels))
inputs = np.load('data/input_train.npy')
outputs = np.load('data/target_train.npy')
print("Done loading data.")

# Now train the model
model.fit(inputs, outputs, epochs=10, batch_size=128)
model.save('data/model.h5')
```

```
Compiling model.
Done loading data.
Train on 1899519 samples
Epoch 1/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.5209
Epoch 2/10
1899519/1899519 [==============================] - 43s 23us/sample - loss: 0.4469
Epoch 3/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4351
Epoch 4/10
1899519/1899519 [==============================] - 43s 23us/sample - loss: 0.4287
Epoch 5/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4260
Epoch 6/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4244
Epoch 7/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4270
Epoch 8/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4242
Epoch 9/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4295
Epoch 10/10
1899519/1899519 [==============================] - 44s 23us/sample - loss: 0.4286
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning: You are saving your model as an HDF5 file via `m
  saving_api.save_model(
```

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()

class Parser(object):

    def __init__(self, extractor, modelfile):
        self.model = keras.models.load_model(modelfile)
        self.extractor = extractor

        # The following dictionary from indices to output actions will be useful
        self.output_labels = dict([(index, action) for (action, index) in extractor.output_labels.items()])

    def parse_sentence(self, words, pos):
        state = State(range(1,len(words)))
        state.stack.append(0)
```

```
        dep_relations = ['tmod', 'vmod', 'csubjpass', 'rcmod', 'ccomp', 'poss', 'parataxis', 'appos', 'dep', 'iobj', 'pobj', 'mwe', 'quantmod


        while state.buffer:
            inRep = np.asarray([self.extractor.get_input_representation(words, pos, state)])
            out = self.model.predict(inRep)
            actions = {}
            for i in range(len(out)):
                actions[i] = out[i]
            sorted(actions.items(), key=lambda x: x[1], reverse=True)
            for k,v in actions.items():
                mv = (k/45, k%45)
                if len(state.stack)==0:
                    state.shift()
                    break
                elif len(state.buffer)==1 and mv[0]==2:
                    if len(state.stack)!=0:
                        continue
                    else:
                        state.shift()
                        break
                elif mv[0] == 0 and mv[1] == dep_relations.index('root'):
                    continue
                else:
                    actIdx = mv[0]
                    depIdx = mv[1]
                    dep = dep_relations[depIdx]
                    if actIdx == 0:
                        state.left_arc(dep)
                    if actIdx == 1:
                        state.right_arc(dep)
                    else:
                        state.shift
                    break


        result = DependencyStructure()
        for p,c,r in state.deps:
            result.add_deprel(DependencyEdge(c,words[c],pos[c],p, r))
        return result



WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

try:
    word_vocab_f = open(WORD_VOCAB_FILE,'r')
    pos_vocab_f = open(POS_VOCAB_FILE,'r')
except FileNotFoundError:
    print("Could not find vocabulary files {} and {}".format(WORD_VOCAB_FILE, POS_VOCAB_FILE))
    sys.exit(1)

extractor = FeatureExtractor(word_vocab_f, pos_vocab_f)
parser = Parser(extractor, 'data/model.h5')

with open('data/dev.conll','r') as in_file:
    for dtree in conll_reader(in_file):
        words = dtree.words()
        pos = dtree.pos()
        deps = parser.parse_sentence(words, pos)
        print(deps.print_conll())
        print()
```

```
5       to      _       _       TO      _       6       tmod    _       _
6       quality _       _       NN      _       7       tmod    _       _
7       ''      _       _       ''      _       8       tmod    _       _
8       that    _       _       WDT     _       9       tmod    _       _
9       triggered       _       _       VBD     _       10      tmod    _       _
10      Friday  _       _       NNP     _       11      tmod    _       _
11      's      _       _       POS     _       12      tmod    _       _
12      explosive       _       _       JJ      _       13      tmod    _       _
13      bond-market     _       _       JJ      _       14      tmod    _       _
14      rally   _       _       NN      _       15      tmod    _       _
15      was     _       _       VBD     _       16      tmod    _       _
16      reversed        _       _       VBN     _       17      tmod    _       _
17      yesterday       _       _       NN      _       18      tmod    _       _
18      in      _       _       IN      _       19      tmod    _       _
19      a       _       _       DT      _       20      tmod    _       _
20      ``      _       _       ``      _       21      tmod    _       _
21      flight  _       _       NN      _       22      tmod    _       _
22      from    _       _       IN      _       23      tmod    _       _
23      quality _       _       NN      _       24      tmod    _       _
24      ''      _       _       ''      _       25      tmod    _       _
25      rout    _       _       NN      _       26      tmod    _       _

0       None    _       _       None    _       1       tmod    _       _
1       The     _       _       DT      _       2       tmod    _       _
2       setback _       _       NN      _       3       tmod    _       _
3       ,       _       _       ,       _       4       tmod    _       _
4       in      _       _       IN      _       5       tmod    _       _
5       which   _       _       WDT     _       6       tmod    _       _
6       Treasury        _       _       NNP     _       7       tmod    _       _
```