sx2261_NLP_HW3

April 4, 2020

0.1 Analytical Component

Paper: Chen, D., & Manning, C. (2014). A fast and accurate dependency parser using neural networks.

0.2 Programming Component - Neural Network Dependency Parsing

[1]:
```
import keras
import tensorflow as tf
```

```
Using TensorFlow backend.
```

conll_reader.py

[2]:
```
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))
```

0.2.1 Part 1 Obtaining the Vocabulary

get_vocab.py

[4]:
```
import sys
from conll_reader import conll_reader
from collections import defaultdict

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
```

[5]:
```
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))
```

```
Writing word indices…
Writing POS indices…
```

[ ]:

0.2.2 Part 2 - Extracting Input/Output matrices for training

0.2.3 extract_tranning_data.py

[6]:
```
from conll_reader import DependencyStructure, conll_reader
from collections import defaultdict
import copy
import sys
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', 'acomp',␣
↪'number', 'csubj', 'root', 'auxpass', 'prep', 'mark', 'expl', 'cc',␣
↪'npadvmod', 'prt', 'nsubj', 'advmod', 'conj', 'advcl', 'punct', 'aux',␣
↪'pcomp', 'discourse', 'nsubjpass', 'predet', 'cop', 'possessive', 'nn',␣
↪'xcomp', 'preconj', 'num', 'amod', 'dobj', 'neg','dt','det']


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):
        # TODO: Write this method for Part 2
        # return a single vector of 6
        # the idea
        # 1: when state.stack[position] is 0, the current word is considered␣
↪as"<Root>"
        # 2: "<NULL>": padding context window
        # 3: if pos[state.stack[position]] == "CD", consider the current word␣
↪as "<CD>"
        # 4: if pos[state.stack[position]] == "NNP", consider the current word␣
↪as "<NNP>"
        # 5: otherwise: consider the current word as "<UNK>"

        # result list
        input_list = []
```

```
        length_stack = len(state.stack)
        position = -1
        while position >= -3:
            if length_stack > 0:
                word_pos = state.stack[position]
                if word_pos == 0:
                    input_list.append(self.word_vocab["<ROOT>"])
                elif pos[word_pos] == "CD":
                    input_list.append(self.word_vocab["<CD>"])
                elif pos[word_pos] == "NNP":
                    input_list.append(self.word_vocab["<NNP>"])
                else:
                    if words[word_pos].lower() in self.word_vocab:
                        input_list.append(self.word_vocab[words[word_pos].
↪lower()])
                    else:
                        input_list.append(self.word_vocab["<UNK>"])
            else:
                input_list.append(self.word_vocab["<NULL>"])
            length_stack = length_stack - 1
            position = position - 1


        length_buffer = len(state.buffer)
        position = -1
        while position >= -3:
            if length_buffer > 0:
                word_pos = state.buffer[position]
                if word_pos == 0:
                    input_list.append(self.word_vocab["<ROOT>"])
                elif pos[word_pos] == "CD":
                    input_list.append(self.word_vocab["<CD>"])
                elif pos[word_pos] == "NNP":
                    input_list.append(self.word_vocab["<NNP>"])
                else:
                    if words[word_pos].lower() in self.word_vocab:
                        input_list.append(self.word_vocab[words[word_pos].
↪lower()])
                    else:
                        input_list.append(self.word_vocab["<UNK>"])

            else:
                input_list.append(self.word_vocab["<NULL>"])
            length_buffer = length_buffer - 1
            position = position - 1

        return np.asarray(input_list, dtype=np.int)
```

```
    def get_output_representation(self, output_pair):
        # TODO: Write this method for Part 2
        return keras.utils.to_categorical(self.output_labels[output_pair],␣
 ↪num_classes=len(self.output_labels), dtype=int)


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)
```

0.2.4 testing get_training_matrices() for training set

[7]:
```
WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

argv1 = "data/train.conll"
argv2 = "data/input_train.npy"
argv3 = "data/target_train.npy"

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(argv1,'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(argv2, inputs)
    np.save(argv3, outputs)
```

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

[ ]:

0.2.5 testing get_training_matrices() for development data set

[9]:
```
WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

argv1 = "data/dev.conll"
argv2 = "data/input_dev.npy"
argv3 = "data/target_dev.npy"

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(argv1,'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(argv2, inputs)
    np.save(argv3, outputs)
```

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

[ ]:

0.2.6 Part 3 Designing and Training the network

0.2.7 train_model.py

[10]:
```
from extract_training_data import FeatureExtractor
import sys
import numpy as np
import keras
from keras import Sequential
from keras.layers import Flatten, Embedding, Dense
# add by Shusen Xu
from keras.layers import Activation

def build_model(word_types, pos_types, outputs):
    # TODO: Write this function for part 3
    model = Sequential()
    # add Embedding layer
    model.add(Embedding(input_dim=word_types, output_dim=32, input_length=6))
    # flatten
    model.add(Flatten())
    # add hidden layers
    model.add(Dense(100, activation='relu'))
    model.add(Dense(10, activation='relu'))

    # add outputlayers
    model.add(Dense(outputs, activation='softmax'))

    model.compile(keras.optimizers.Adam(lr=0.01),␣
 ↪loss="categorical_crossentropy")
    return model
```

0.2.8 test for trainning models

[11]:
```
WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'
argv1 = "data/input_train.npy"
argv2 = "data/target_train.npy"
argv3 = "data/model.h5"


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(argv1)
outputs = np.load(argv2)
print("Done loading data.")

# Now train the model
model.fit(inputs, outputs, epochs=5, batch_size=100)

model.save(argv3)
```

```
Compiling model.
Done loading data.
```

```
//anaconda3/lib/python3.7/site-
packages/tensorflow_core/python/framework/indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

```
Epoch 1/5
1899519/1899519 [==============================] - 80s 42us/step - loss: 0.4914
Epoch 2/5
1899519/1899519 [==============================] - 80s 42us/step - loss: 0.4147
Epoch 3/5
1899519/1899519 [==============================] - 82s 43us/step - loss: 0.3997
Epoch 4/5
1899519/1899519 [==============================] - 83s 44us/step - loss: 0.3915
Epoch 5/5
1899519/1899519 [==============================] - 83s 44us/step - loss: 0.3850
```

0.2.9 Part 4 Greedy Parsing Algorithm - Building and Evaluating the Parser

0.2.10 decoder.py

[13]:
```
from conll_reader import DependencyStructure, DependencyEdge, conll_reader
from collections import defaultdict
import copy
import sys

import numpy as np
import keras
```

```
from extract_training_data import FeatureExtractor, State

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)

        while state.buffer:
            # pass
            # TODO: Write the body of this loop for part 4
            # step 1:
            features = self.extractor.get_input_representation(words, pos,␣
↪state)
            possible_actions = self.model.predict(features.reshape([1,6]))
            possible_actions = possible_actions.reshape(91)
            # step 2: select the highest scoring permitted transition

            # create a possible action indices list sorted by their␣
↪possibility(largest one comes first)
            # sorted_actions_indices = np.flipud(np.argsort(possible_actions))
            sorted_actions_indices = np.flipud(np.argsort(possible_actions))

            # going through and find the highest scoring permitted trasition
            for i in sorted_actions_indices:
                flag = False
                # check the current transition whether permitted or not
                if self.output_labels[i][0] == "shift":
                    if state.stack and len(state.buffer) == 1:
                        flag = False
                    else:
                        flag = True

                elif self.output_labels[i][0] == "left_arc":
                    if not state.stack:
                        flag = False
                    elif state.stack[-1] == 0:
                        flag = False
```

```
                    else:
                        flag = True

                elif self.output_labels[i][0] == "right_arc":
                    if not state.stack:
                        flag = False
                    else: flag = True

                # when flag == True, it states that the cuurent transition is␣
↪permitted
                if flag == True:
                    transition = self.output_labels[i]
                    # update the state accordingly
                    if transition[0] == "shift":
                        state.shift()
                    elif transition[0] == "left_arc":
                        state.left_arc(transition[1])
                    elif transition[0] == "right_arc":
                        state.right_arc(transition[1])
                    break

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

[ ]:

0.2.11 test for decoder.py

[14]:
```
WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

argv1 = "data/model.h5"
argv2 = "data/dev.conll"


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, argv1)

with open(argv2,'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()
```

```
//anaconda3/lib/python3.7/site-
packages/tensorflow_core/python/framework/indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

[ ]:

0.2.12 evaluate.py

[15]:
```
from decoder import Parser
from extract_training_data import FeatureExtractor
from conll_reader import conll_reader
import sys


def compare_parser(target, predict):
    target_unlabeled = set((d.id,d.head) for d in target.deprels.values())
    target_labeled = set((d.id,d.head,d.deprel) for d in target.deprels.
 ↪values())
    predict_unlabeled = set((d.id,d.head) for d in predict.deprels.values())
    predict_labeled = set((d.id,d.head,d.deprel) for d in predict.deprels.
 ↪values())

    labeled_correct = len(predict_labeled.intersection(target_labeled))
    unlabeled_correct = len(predict_unlabeled.intersection(target_unlabeled))
    num_words = len(predict_labeled)
    return labeled_correct, unlabeled_correct, num_words
```

0.2.13 test for evaluate.py

[16]:
```
WORD_VOCAB_FILE = 'data/words.vocab'
POS_VOCAB_FILE = 'data/pos.vocab'

argv1 = "data/model.h5"
```

```
argv2 = "data/dev.conll"


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, argv1)


total_labeled_correct = 0
total_unlabeled_correct = 0
total_words = 0

las_list = []
uas_list = []

count = 0


with open(argv2,'r') as in_file:
    print("Evaluating. (Each . represents 100 test dependency trees)")
    for dtree in conll_reader(in_file):
        words = dtree.words()
        pos = dtree.pos()
        predict = parser.parse_sentence(words, pos)
        labeled_correct, unlabeled_correct, num_words = compare_parser(dtree,␣
 ↪predict)
        las_s = labeled_correct / float(num_words)
        uas_s = unlabeled_correct / float(num_words)
        las_list.append(las_s)
        uas_list.append(uas_s)
        total_labeled_correct += labeled_correct
        total_unlabeled_correct += unlabeled_correct
        total_words += num_words
        count +=1
        if count % 100 == 0:
            print(".",end="")
            sys.stdout.flush()
print()

las_micro = total_labeled_correct / float(total_words)
```

```
uas_micro = total_unlabeled_correct / float(total_words)

las_macro = sum(las_list) / len(las_list)
uas_macro = sum(uas_list) / len(uas_list)

print("{} sentence.\n".format(len(las_list)))
print("Micro Avg. Labeled Attachment Score: {}".format(las_micro))
print("Micro Avg. Unlabeled Attachment Score: {}\n".format(uas_micro))
print("Macro Avg. Labeled Attachment Score: {}".format(las_macro))
print("Macro Avg. Unlabeled Attachment Score: {}".format(uas_macro))
```

```
//anaconda3/lib/python3.7/site-
packages/tensorflow_core/python/framework/indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "

Evaluating. (Each . represents 100 test dependency trees)
…
5039 sentence.

Micro Avg. Labeled Attachment Score: 0.695314378580964
Micro Avg. Unlabeled Attachment Score: 0.7512997189124648

Macro Avg. Labeled Attachment Score: 0.7049957030377082
Macro Avg. Unlabeled Attachment Score: 0.7612958100753767
```

[ ]: