grammaropt Documentation

Release 0.1

Mehdi Cherti

Nov 01, 2017

Contents

CHAPTER 1

API Documentation

1.1 Grammar

1.2 Walkers

class grammaropt.grammar.**Walker**(*grammar*, *min_depth=None*, *max_depth=None*, *strict_depth_limit=False*)

Walkers are objects that do a random walk (deterministic walks are a special case of random walks) on the production rules space of a given grammar : the responsability of a Walker is to choose a production rule whenever it has to, and to choose *Type* values whenever it has to. After the end of the walk, a trace is left describing the traversal of the space. The trace differs depending on the kind of Walker.

Parameters **grammar** : Grammar

grammar where to walk

min_depth : int

minimum depth of the parse tree.

max_depth : int

maximum depth of the parse tree. Note that it could exceed *max_depth* because when it reaches *max_depth* there is no garanthee that there would always be a terminal production rule to choose. The solution to this problem is that when *max_depth* is reached, non-terminal production rules stop from being candidates to be chosen, but when only what we can choose are non-terminal production rules, we just choose one of them, even if *max_depth* is exceeded, otherwise the obtained string will not be a valid one according to the grammar.

strict_depth_limit : bool

if True, when *max_depth* is reached, forbid any further production rules when a choice should be made. If False, even when *max_depth* is reached, choose terminals when terminals are available, otherwise keep applying production rules.

Methods

next_rule
next_value
walk

next_rule(*rules*)
Given a set of production *rules*, choose the next one. Implemented by specific Walkers. *depth* is an *int* provides information about the current depth of the parsetree.

next_value(*rule*)
Given a *Type rule*, choose its value. Implemented by specific Walkers.

walk(*start=None*)
Do a random walk on the grammar.
Parameters **start** : str
str that provides the starting rule name if *start* is not provides it uses the default rule of the grammar (which is the first rule).

class grammaropt.grammar.**DeterministicWalker**(*grammar*, *expr*)
a very specific Walker that uses a given str expression *expr*, parse it usign the grammar *grammar*, then use the parse tree to force the next rule to choose and the next value to choose to correspond exactly to the expression *expr*. This very specific walker trace is used by the RNN walker to compute the loss for a given groundtruth expressions. Unfortunately I had to patch the uncached_match method of *parsimonious OneOf* and *Type* rules of the grammar to get some missing information required in the trace, the missing information needed by *DeterministicWalker* was the parent rule that is used to create a *Node*.

Methods

next_rule
next_value
walk

1.3 Adapters

1.4 Models

1.5 Optimizers

1.6 Types

CHAPTER 2

General examples

Introductory examples.

2.1 Vectorize

Example of using Vectorize to transform a set of string expressions into a list of list of integers.

```
from grammaropt.grammar import build_grammar
from grammaropt.grammar import Vectorizer

rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / ("exp" po S pc) / "x
↪" / int
    po = "("
    pc = ")"
    int = "1" / "2" / "3"
"""

grammar = build_grammar(rules)
v = Vectorizer(grammar)
corpus = [
    "1+2",
]
X = v.transform(corpus)
corpus_ = v.inverse_transform(X)
print(corpus_)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.2 Simple

Simple example of using random uniform generation.

```
import sys

from grammaropt.grammar import build_grammar
from grammaropt.grammar import as_str

from grammaropt.terminal_types import Int
from grammaropt.random import RandomWalker


rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / ("exp" po S pc) / "x
↪" / int
    po = "("
    pc = ")"
"""

types = {'int': Int(1, 10)}
grammar = build_grammar(rules, types=types)

wl = RandomWalker(grammar=grammar, min_depth=1, max_depth=10)

for _ in range(10):
    wl.walk()
    expr = as_str(wl.terminals)
    print(expr)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.3 Simple RNN

Simple example of using RNNs.

```
import pandas as pd
import numpy as np

import torch

from grammaropt.grammar import build_grammar
from grammaropt.grammar import extract_rules_from_grammar
from grammaropt.grammar import as_str
from grammaropt.random import RandomWalker
from grammaropt.terminal_types import Int, Float
from grammaropt.rnn import RnnModel
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnWalker


X = np.random.uniform(-3, 3, size=(1000,))
y = X**2
```

```
def evaluate(code):
    from numpy import exp, cos, sin
    x = X
    y_pred = eval(code)
    score = (np.abs(y_pred - y) <= 0.1).mean()
    score = float(score)
    return score

rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / "x" / int
    po = "("
    pc = ")"
"""

types = {'int': Int(1, 10)}
grammar = build_grammar(rules, types=types)

rules = extract_rules_from_grammar(grammar)
tok_to_id = {r: i for i, r in enumerate(rules)}

# set hyper-parameters and build RNN model
nb_iter = 1000
vocab_size = len(rules)
emb_size = 128
hidden_size = 256
nb_features = 2
lr = 1e-3
gamma = 0.9

model = RnnModel(
    vocab_size=vocab_size,
    emb_size=emb_size,
    hidden_size=hidden_size,
    nb_features=nb_features)

optim = torch.optim.Adam(model.parameters(), lr=lr)
rnn = RnnAdapter(model, tok_to_id)

# optimization loop
acc_rnn = []
R_avg, R_max = 0., 0.
wl = RnnWalker(grammar=grammar, rnn=rnn, min_depth=1, max_depth=10, strict_
↪depth_limit=False)
out = []
for it in range(nb_iter):
    wl.walk()
    code = as_str(wl.terminals)
    R = evaluate(code)
    R_avg = R_avg * gamma + R * (1 - gamma)
    model.zero_grad()
    loss = (R - R_avg) * wl.compute_loss()
    loss.backward()
    optim.step()
    R_max = max(R, R_max)
    acc_rnn.append(R_max)
    out.append({"code": code, "R": R})
    print(code, R, R_max)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.4 Fit

Example of fitting a set of string expressions into an RNN model.

```
import sys

import torch

from grammaropt.grammar import build_grammar
from grammaropt.grammar import DeterministicWalker
from grammaropt.grammar import extract_rules_from_grammar
from grammaropt.grammar import as_str
from grammaropt.terminal_types import Int
from grammaropt.rnn import RnnModel
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnWalker
from grammaropt.rnn import RnnDeterministicWalker
from grammaropt.random import RandomWalker


nb_iter = 1000
lr = 1e-4

# rules are a simple symbolic expression
rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = ("(" S ")") / ("sin(" S ")") / ("exp(" S ")") / "x" / int
"""
types = {'int': Int(1, 5)}

# build grammar
grammar = build_grammar(rules, types=types)
rules = extract_rules_from_grammar(grammar)
tok_to_id = {r: i for i, r in enumerate(rules)}

# build model
model = RnnModel(vocab_size=len(rules))
optim = torch.optim.Adam(model.parameters(), lr=lr)
rnn = RnnAdapter(model, tok_to_id)

# generate uniformly at random an expression from the grammar
wl = RandomWalker(grammar=grammar, min_depth=1, max_depth=5, random_state=42)
wl.walk()
expr = as_str(wl.terminals)

# collect the sequence of decisions (production rules and values) needed to␣
↪produce
# the generated expression `expr`
wl = DeterministicWalker(grammar=grammar, expr=expr)
wl.walk()
gt = wl.decisions

for _ in range(1000):
    # fit the RNN model to make it more likely to generate `expr
    model.zero_grad()
```

```
    wl = RnnDeterministicWalker(grammar=grammar, rnn=rnn, decisions=gt)
    wl.walk()
    loss = wl.compute_loss()
    loss.backward()
    optim.step()
    # check if the generation works by generating from the RNN model
    wl = RnnWalker(grammar=grammar, rnn=rnn)
    wl.walk()
    expr_rnn = as_str(wl.terminals)
    print('Loss : {:.5f}, Generated : {}, Groundtruth : {}'.format(loss.
↪data[0], expr_rnn, expr))
```

Total running time of the script: (0 minutes 0.000 seconds)

2.5 Torch model

Example of using a Torch RNN model along with Vectorize to fit a set of string expressions.

```
import numpy as np
from itertools import product

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.optim import Adam

from grammaropt.grammar import build_grammar
from grammaropt.grammar import Vectorizer
from grammaropt.grammar import as_str
from grammaropt.grammar import NULL_SYMBOL
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnModel
from grammaropt.rnn import RnnWalker

def acc(pred, true_classes):
    _, pred_classes = pred.max(1)
    acc = (pred_classes == true_classes).float().mean()
    return acc


# Grammar and corpus
rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / ("exp" po S pc) / "x
↪" / int
    po = "("
    pc = ")"
    int = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
"""
grammar = build_grammar(rules)
corpus = [
    'x*{}+{}'.format(i, j)
    for i, j in product(range(10), range(10))
]
vect = Vectorizer(grammar, pad=True)
X = vect.transform(corpus)
```

```
X = [[0] + x for x in X]
X = np.array(X).astype('int32')

# Model
max_length = max(map(len, X))
vocab_size = len(vect.tok_to_id)
emb_size = 32
batch_size = 32
hidden_size = 32
epochs = 1000
model = RnnModel(vocab_size=vocab_size, emb_size=emb_size, hidden_
↪size=hidden_size)
optim = Adam(model.parameters(), lr=1e-3)
adp = RnnAdapter(model, tok_to_id=vect.tok_to_id, begin_tok=NULL_SYMBOL)
wl = RnnWalker(grammar, adp, temperature=1.0, min_depth=1, max_depth=5)

# Training
I = X[:, 0:-1]
O = X[:, 1:]
crit = nn.CrossEntropyLoss()
avg_loss = 0.
avg_precision = 0.
for i in range(epochs):
    for j in range(0, len(I), batch_size):
        inp = I[j:j+batch_size]
        out = O[j:j+batch_size]
        out = out.flatten()
        inp = torch.from_numpy(inp).long()
        inp = Variable(inp)
        out = torch.from_numpy(out).long()
        out = Variable(out)

        model.zero_grad()
        y = model(inp)
        loss = crit(y, out)
        precision = acc(y, out)
        loss.backward()
        optim.step()

        avg_loss = avg_loss * 0.9 + loss.data[0] * 0.1
        avg_precision = avg_precision * 0.9 + precision.data[0] * 0.1
        if i % 10 == 0:
            print('Epoch : {:05d} Avg loss : {:.6f} Avg Precision : {:.6f}'.
↪format(i, avg_loss, avg_precision))
            print('Generated :')
            wl.walk()
            expr = as_str(wl.terminals)
            print(expr)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.6 Keras model

Example of using a keras RNN model along with Vectorize to fit a set of string expressions.

```
from itertools import product
import numpy as np
from keras.layers import LSTM, Input, TimeDistributed, Activation, Embedding,
↪ Dense
import keras.backend as K
from keras.models import Model
from keras.optimizers import Adam
from grammaropt.grammar import build_grammar
from grammaropt.grammar import Vectorizer


def precision(y_true, y_pred):
    return (y_true.argmax(axis=-1) == y_pred.argmax(axis=-1)).mean()


def categorical_crossentropy(y_true, y_pred):
    yt = y_true.flatten()
    ypr = y_pred.reshape((y_pred.shape[0]*y_pred.shape[1], y_pred.shape[2]))
    return K.categorical_crossentropy(yt, ypr)


def onehot(X, D=10):
    X = X.astype('int32')
    nb = np.prod(X.shape)
    x = X.flatten()
    m = np.zeros((nb, D))
    m[np.arange(nb), x] = 1.
    m = m.reshape(X.shape + (D,))
    return m.astype('float32')

def generate(model, length, nb=1):
    x = np.zeros((nb, length + 1,)).astype('int32')
    for i in range(length):
        y = model.predict(x[:, 0:length])
        for e in range(nb):
            symbol = np.random.choice(np.arange(y.shape[2]), p=y[e, i])
            x[e, i + 1] = symbol
    return x


# Grammar and corpus
rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / ("exp" po S pc) / "x
↪" / int
    po = "("
    pc = ")"
    int = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
"""
grammar = build_grammar(rules)
corpus = [
    'x*{}+{}'.format(i, j)
    for i, j in product(range(10), range(10))
]

vect = Vectorizer(grammar, pad=True)
X = vect.transform(corpus)
X = [[0] + x for x in X]
```

```
X = np.array(X).astype('int32')

# Model
max_length = max(map(len, X))
vocab_size = len(vect.tok_to_id)
emb_size = 32
batch_size = 32
epochs = 800
inp = Input(shape=(max_length - 1,))
x = Embedding(vocab_size, emb_size)(inp)
x = LSTM(32, return_sequences=True)(x)
x = TimeDistributed(Dense(vocab_size))(x)
out = Activation('softmax')(x)
model = Model(inputs=inp, outputs=out)
model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-3))
I = X[:, 0:-1]
O = X[:, 1:]

# Training
avg_precision = 0.0
avg_loss = 0.0
for i in range(epochs):
    for j in range(0, len(I), batch_size):
        inp = I[j:j+batch_size]
        out = O[j:j+batch_size]
        out = onehot(out, D=vocab_size)
        loss = model.train_on_batch(inp, out)
        p = np.mean(precision(out, model.predict(inp)))
        avg_precision = avg_precision * 0.9 + p * 0.1
        avg_loss = avg_loss * 0.9 + loss * 0.1
        if i % 10 == 0:
            print('Epoch : {:05d} Avg loss : {:.6f} Avg Precision : {:.6f}'.
↪format(i, avg_loss, avg_precision))
    y = generate(model, max_length - 1, nb=1)
    y = [expr[1:] for expr in y]
    try:
        y = vect.inverse_transform(y)
    except Exception:
        # happens because the `generate` function does not take into
        # account the forbidden rules by the grammar. So the exception
        # occurs because of a syntax error.
        continue
    print('Generated:')
    for expr in y:
        print(expr)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.7 Pipeline

Simple example of optimizing a scikit-learn pipeline

```
import warnings

import pandas as pd
import numpy as np
```

```
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.model_selection import cross_val_score

import torch

from grammaropt.grammar import build_grammar
from grammaropt.grammar import extract_rules_from_grammar
from grammaropt.grammar import as_str
from grammaropt.random import RandomWalker
from grammaropt.terminal_types import Int, Float
from grammaropt.rnn import RnnModel
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnWalker

warnings.filterwarnings("ignore")

digits = datasets.load_digits()
dataset = digits


def evaluate(code):
    X = dataset.data
    y = dataset.target
    clf = _build_estimator(code)
    try:
        scores = cross_val_score(clf, X, y, cv=5)
    except Exception:
        return 0.
    else:
        return float(np.mean(scores))


def _build_estimator(code):
    clf = eval(code)
    return clf


def main():
    # grammar is simple scikit-learn pipeline
    rules = """
        pipeline = "make_pipeline" "(" elements "," estimator ")"
        elements = (element "," elements) / element
        element = pca / rf
        pca = "PCA" "(" "n_components" "=" int ")"
        estimator = rf / logistic
        logistic = "LogisticRegression" "(" ")"
        rf = "RandomForestClassifier" "(" "max_depth" "=" int "," "max_
↪features" "=" float ")"
    """
    # build grammar
    types = {'int': Int(1, 10), 'float': Float(0., 1.)}
    grammar = build_grammar(rules, types=types)
    rules = extract_rules_from_grammar(grammar)
```

```
    tok_to_id = {r: i for i, r in enumerate(rules)}

    # set hyper-parameters and build RNN model
    nb_iter = 100
    vocab_size = len(rules)
    emb_size = 128
    hidden_size = 128
    nb_features = 2
    lr = 1e-4
    gamma = 0.9

    model = RnnModel(
        vocab_size=vocab_size,
        emb_size=emb_size,
        hidden_size=hidden_size,
        nb_features=nb_features)

    optim = torch.optim.Adam(model.parameters(), lr=lr)
    rnn = RnnAdapter(model, tok_to_id)

    # optimization loop
    acc_rnn = []
    R_avg, R_max = 0., 0.
    wl = RnnWalker(grammar=grammar, rnn=rnn)
    out = []
    for it in range(nb_iter):
        # walk to generate a parse tree
        wl.walk()
        # get the obtained terminals from the walker
        # and convert to a python code as str
        code = as_str(wl.terminals)
        # evaluate the code with python interpreter
        # and return the validation accuracy, which is the
        # reward
        R = evaluate(code)
        R_avg = R_avg * gamma + R * (1 - gamma)
        # update the model
        model.zero_grad()
        # loss is policy gradient : generate using the
        # model, observe reward R, then maximize the probability
        # of the generated decisions propoertionally to the reward.
        # `R_avg is the policy gradient baseline to make the gradient
        # have less variance.
        loss = (R - R_avg) * wl.compute_loss()
        loss.backward()
        optim.step()
        R_max = max(R, R_max)
        acc_rnn.append(R_max)
        out.append({"code": code, "R": R})
        print(code, R)
    df = pd.DataFrame(out)
    df = df.sort_values(by="R", ascending=False)
    df.to_csv('pipeline.csv')

if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

2.8 SMILES

Example of fitting SMILES, a representation of molecular graphs.

```
import numpy as np
import pandas as pd

from molecules.molecule import is_valid

import torch

from grammaropt.grammar import build_grammar
from grammaropt.grammar import as_str
from grammaropt.random import RandomWalker
from grammaropt.grammar import extract_rules_from_grammar
from grammaropt.rnn import RnnModel
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnWalker
from grammaropt.rnn import RnnDeterministicWalker
from grammaropt.grammar import DeterministicWalker


def _get_max_depth(node):
    if len(node.children) == 0:
        return 0
    return max(1 + _get_max_depth(c) for c in node.children)


rules = r"""
    smiles = chain
    atom = bracket_atom / aliphatic_organic / aromatic_organic
    aliphatic_organic = "Cl" / "Br" / "B" / "N" / "O" / "S" / "P" / "F" / "I
↪" / "C"
    aromatic_organic = "c" / "n" / "o" / "s"
    bracket_atom = "[" BAI "]"
    BAI = (isotope symbol BAC) / (isotope symbol) / (symbol BAC) / symbol
    BAC = (chiral BAH) / BAH / chiral
    BAH = (hcount BACH) / BACH / hcount
    BACH = (charge class) / charge / class
    symbol = aliphatic_organic / aromatic_organic
    isotope = (DIGIT DIGIT DIGIT) / (DIGIT DIGIT) / DIGIT
    DIGIT = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8"
    chiral = "@@" / "@"
    hcount = ("H" DIGIT) / "H"
    charge = ("-" DIGIT DIGIT) / ("+" DIGIT DIGIT) / ("-" DIGIT) / ("+"␣
↪DIGIT) / "-" / "+"
    bond = "-" / "=" / "#" / "/" / "\\"
    ringbond = (bond DIGIT) / DIGIT
    branched_atom = (atom RB BB) / (atom RB) / (atom BB) / atom
    RB = (ringbond RB) / ringbond
    BB = (branch BB) / branch
    branch = ( "(" bond chain ")" ) / ("(" "." chain ")" ) / ("(" chain ")
↪" )
    chain = (bond branched_atom chain) / (branched_atom chain) / (branched_
↪atom "." chain) / (bond branched_atom) / branched_atom
    class = ":" digits
    digits = (DIGIT digits) / DIGIT
"""
```

```
# build grammar
grammar = build_grammar(rules)
rules = extract_rules_from_grammar(grammar)
tok_to_id = {r: i for i, r in enumerate(rules)}
# build model
lr = 1e-3
gamma = 0.99

model = RnnModel(vocab_size=len(rules), num_layers=2)
optim = torch.optim.Adam(model.parameters(), lr=lr)
rnn = RnnAdapter(model, tok_to_id)

smiles = np.load('zinc_250k_subset.npz')['X']
#for s in smiles:
#    grammar.parse(s)
max_depth = 100
print('Size of training : {}'.format(len(smiles)))
#max_depth = max(map(_get_max_depth, map(grammar.parse, smiles)))
nb_updates = 0
avg_loss = 0.
print('Start training...')
for epoch in range(100):
    np.random.shuffle(smiles)
    for i, s in enumerate(smiles):
        wl = RnnDeterministicWalker.from_str(grammar, rnn, s)
        wl.walk()
        model.zero_grad()
        loss = wl.compute_loss() / len(wl.decisions)
        loss.backward()
        optim.step()
        avg_loss = avg_loss * gamma + loss.data[0] * (1 - gamma)
        print('Example {:06d}/{:06d} avg loss : {:.4f}, loss : {:4f}'.
↪format(i, len(smiles), avg_loss, loss.data[0]))
        if nb_updates % 100 == 0 and nb_updates > 0:
            print('Generating...')
            wl = RnnWalker(
                grammar=grammar,
                rnn=rnn,
                min_depth=1, max_depth=max_depth,
                strict_depth_limit=False
            )
            nb_valid = 0
            nb = 100
            for _ in range(nb):
                # check if the generation works by generating from the RNN␣
↪model
                wl.walk()
                expr = as_str(wl.terminals)
                print(expr)
                nb_valid += is_valid(expr)
            print('nb valid : {}/{}'.format(nb_valid, nb))
            torch.save(model, 'model.th')
        nb_updates += 1
```

Total running time of the script: (0 minutes 0.000 seconds)

2.9 Autoencoder model with Torch

Example of using a Torch RNN autoencoder model to learn continuous representations of strings.

```
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
from itertools import product

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.optim import Adam

from grammaropt.grammar import build_grammar
from grammaropt.grammar import Vectorizer
from grammaropt.grammar import NULL_SYMBOL
from grammaropt.grammar import as_str
from grammaropt.rnn import RnnAdapter
from grammaropt.rnn import RnnWalker

class Model(nn.Module):
    def __init__(self, vocab_size=10, emb_size=128, hidden_size=128, num_
↪layers=1, use_cuda=False):
        super().__init__()
        self.vocab_size = vocab_size
        self.emb_size = emb_size
        self.hidden_size = hidden_size
        self.use_cuda = use_cuda
        self.X = None
        # convolutional autoencoder
        self.features = nn.Sequential(
            nn.Conv1d(emb_size, 32, 3),
            nn.ReLU(True),
            nn.Conv1d(32, hidden_size, 3),
        )
        self.emb = nn.Embedding(vocab_size, emb_size)
        # LSTM decoder
        self.lstm = nn.LSTM(hidden_size, 128, batch_first=True, num_
↪layers=num_layers)
        self.out_token  = nn.Linear(128, vocab_size)

    def forward(self, inp):
        T = inp.size(1)
        x = self.encode(inp)
        x = x.view(x.size(0), 1, -1)
        x = x.repeat(1, T, 1)
        o, _ = self.lstm(x)
        o = o.contiguous()
        o = o.view(o.size(0) * o.size(1), o.size(2))
        o = self.out_token(o)
        return o

    def encode(self, inp):
        x = self.emb(inp)
        x = x.transpose(1, 2)
        x = self.features(x)
```

```
        x = x.mean(2)
        return x

    def given(self, inp):
        x = self.encode(inp)
        self.X = x.view(x.size(0), 1, -1)

    def next_token(self, inp, state):
        if self.use_cuda:
            inp = inp.cuda()
        _, state = self.lstm(self.X, state)
        h, c = state
        h = h[-1] # last layer
        o = self.out_token(h)
        return o, state


def acc(pred, true_classes):
    _, pred_classes = pred.max(1)
    acc = (pred_classes == true_classes).float().mean()
    return acc


# Grammar and corpus
rules = r"""
    S = (T "+" S) / (T "*" S) / (T "/" S) / T
    T = (po S pc) / ("sin" po S pc) / ("cos" po S pc) / ("exp" po S pc) / "x
↪" / int
    po = "("
    pc = ")"
    int = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
"""
grammar = build_grammar(rules)
corpus = [
    'x*{}+{}'.format(i, j)
    for i, j in product(range(10), range(10))
]
vect = Vectorizer(grammar, pad=True)
X = vect.transform(corpus)
X = [[0] + x for x in X]
X = np.array(X).astype('int32')

# Model
max_length = max(map(len, X))
vocab_size = len(vect.tok_to_id)
emb_size = 32
batch_size = 64
hidden_size = 2
epochs = 1000
model = Model(vocab_size=vocab_size, emb_size=emb_size, hidden_size=hidden_
↪size)
optim = Adam(model.parameters(), lr=1e-3)
adp = RnnAdapter(model, tok_to_id=vect.tok_to_id, begin_tok=NULL_SYMBOL)
wl = RnnWalker(grammar, adp, temperature=1.0, min_depth=1, max_depth=5)

# Training
I = X
O = X
```

```
crit = nn.CrossEntropyLoss()
avg_loss = 0.
avg_precision = 0.
for i in range(epochs):
    for j in range(0, len(I), batch_size):
        inp = I[j:j+batch_size]
        out = O[j:j+batch_size]
        out = out.flatten()
        inp = torch.from_numpy(inp).long()
        inp = Variable(inp)
        out = torch.from_numpy(out).long()
        out = Variable(out)

        model.zero_grad()
        y = model(inp)
        loss = crit(y, out)
        precision = acc(y, out)
        loss.backward()
        optim.step()

        avg_loss = avg_loss * 0.9 + loss.data[0] * 0.1
        avg_precision = avg_precision * 0.9 + precision.data[0] * 0.1
        if i % 10 == 0:
            print('Epoch : {:05d} Avg loss : {:.6f} Avg Precision : {:.6f}'.
↪format(i, avg_loss, avg_precision))
            print('Generated :')
            model.given(inp)
            wl.walk()
            expr = as_str(wl.terminals)
            print(expr)
            inp = I
            inp = torch.from_numpy(inp).long()
            inp = Variable(inp)
            h = model.encode(inp)
            h = h.data.numpy()
            fig = plt.figure(figsize=(30, 10))
            plt.scatter(h[:, 0], h[:, 1])
            # from https://stackoverflow.com/questions/5147112/matplotlib-
↪how-to-put-individual-tags-for-a-scatter-plot
            for label, x, y in zip(corpus, h[:, 0], h[:, 1]):
                plt.annotate(
                    label,
                    xy=(x, y), xytext=(-20, 20),
                    textcoords='offset points', ha='right', va='bottom',
                    bbox=dict(boxstyle='round,pad=0.5', fc='yellow', alpha=0.
↪5),
                    arrowprops=dict(arrowstyle = '->', connectionstyle='arc3,
↪rad=0'))
            plt.savefig('latent_space.png')
            plt.close(fig)
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index