Bài 7 - Pytorch - Buổi 2 - Seq2seq model correct spelling

19 Aug 2019 - phamdinhkhanh

Menu

1. Giới thiệu chung

Cùng với sự phát triển của deep learning nói chung. Ngày nay lớp các mô hình seq2seq càng tỏ ra hiệu quả trong nhiều tác vụ khác nhau như dịch máy, sửa lỗi chính tả, image captioning, recommendation, dự báo chuỗi thời gian,…. Nhờ sự phát triển của các kiến trúc mạng RNN hiện đại kèm theo các kĩ thuật learning hiệu quả như sử dụng thêm kiến trúc `attention layer`, các phương pháp cải thiện accuracy như ` teach_forcing, beam search` mà mô hình dịch máy ngày càng đạt độ chính xác cao. Các tài liệu về các phương pháp trên đã có nhiều tuy nhiên đa phần là lý thuyết và khá khó cho người mới bắt đầu tiếp cận. Bài viết này nhằm mục đích tạo ra một bản diễn giải kèm thực hành về các bước xây dựng mô hình seq2seq ứng dụng trong sửa lỗi chính tả. Để hiểu được các nội dung trong bài yêu cầu bạn đọc có kiến thức nền tảng về pytorch (https://phamdinhkhanh.github.io/2019/08/10/PytorchTurtorial1.html), nắm vững lý thuyết về mạng LSTM (https://phamdinhkhanh.github.io/2019/04/22/L%C3%BD_thuy%E1%BA%BFt_v%E1%BB%81_m%E1%BA%A1ng_LSTM.html). Ngoài ra bạn đọc cần có sẵn máy tính cài pytorch (https://pytorch.org/) hoặc các VM hỗ trợ pytorch. Để tiện cho thực hành tôi khuyến nghị bạn đọc sử dụng google colab (https://colab.research.google.com) miễn phí và cài sẵn các deep learning framework cơ bản như tensorflow, pytorch, keras,….

Ngoài ra bài viết được xây dựng và tổng hợp dựa trên nhiều nguồn tài liệu khác nhau. Đặc biệt là từ trang hướng dẫn thực hành pytorch (https://pytorch.org/tutorials/beginner/chatbot_tutorial.html), từ ý tưởng được chia sẻ nằm trong top của cuộc thi thêm dấu cho tiếng việt - aivivn 1st (https://forum.machinelearningcoban.com/t/aivivn-3-vietnamese-tone-prediction-1st-place-solution/5721), thêm dấu cho tiếng việt - aivivn 2nd (https://forum.machinelearningcoban.com/t/aivivn-3-vietnamese-tone-prediction-2nd-place-solution/5759).

Để huấn luyện mô hình trên google colab chúng ta cần mount folder lưu trữ dữ liệu trên google drive để có thể access dữ liệu dễ dàng. Bên dưới là câu lệnh thực hiện mount dữ liệu.

```
from google.colab import drive
import os
drive.mount('/content/gdrive')
path = os.path.join('gdrive/My Drive/your_data_folder_link')
os.chdir(path)
!ls
```

Top

Trong đó `your_data_folder_link` là đường link tới folder chứa dữ liệu. Lưu ý link `gdrive/My Drive` là đường trỏ mặc định để đi vào thư mục `My Drive` của bạn.

2. Chuẩn bị dữ liệu

2.1. Giải nén dữ liệu

Ở bước này chúng ta sẽ sử dụng đầu vào là dữ liệu của cuộc thi thêm dấu từ tiếng việt tại aivivn (https://www.aivivn.com/contests/3). Bạn đọc có thể tải về bộ dữ liệu tại google drive (https://drive.google.com/file/d/1m_5CDQQSavev5zWb8JUq97_zUTnOcVvS/view) dưới dạng zip file. Để giải nén dữ liệu ta cần extract bằng hàm extract bên dưới. Nhớ cài đặt package zipfile trước khi chạy lệnh.

```
# Extract zip file
import zipfile
import os

def _extract_zip_file(fn_zip, fn):
  with zipfile.ZipFile(fn, 'r') as zip_ref:
      zip_ref.extractall(fn)

_extract_zip_file(fn_zip = 'vietnamese_tone_prediction.zip', fn = 'vietnamse_tone_predic

print(os.path.exists('vietnamse_tone_prediction'))
```

Cùng lấy ra 500000 dòng đầu tiên của file `train.txt` trong folder giải nén làm tập huấn luyện.

```
def _data_train(fn):
  with open(fn, 'r') as fn:
    train = fn.readlines()
  train = [item[:-1] for item in train[:500000]]
  return train

train = _data_train(fn = 'vietnamse_tone_prediction/train.txt')
print('length of train: {}'.format(len(train)))
train[:5]
```

```
length of train: 500000

['Bộ phim lần đầu được công chiếu tại liên hoan phim Rome 2007 và sau đó được chiếu ở Fa
 'Những kiểu áo sơ mi may theo chất liệu cotton, KT, hay có chút co giãn năm nay cũng đu
 'Đương kim tổng thống là Andrés Manuel López Obrador, người nhậm chức vào ngày 1 tháng
 'Centaurea gloriosa là một loài thực vật có hoa trong họ Cúc.',
 'Sau này mới thấy người ta nói, đó là con rắn đực đi tìm người ăn thịt để trả thù cho r
```

Bên dưới là một số hàm tiện ích để chuyển các từ có dấu của tiếng việt sang không dấu. Mục đích là để tạo ra các cặp (câu không dấu, câu có dấu) từ câu có dấu.

Top

```
# encoding=utf8
import codecs
import csv
import re
import sys

# sys.setdefaultencoding('utf8')

def remove_tone_line(utf8_str):
    intab_l = "ạảãàáâậầấẩẫăắằặẳẵóòọõỏôộổỗồốơờớợởỡéèẻẹẽêếềệểễúùụủũưựữửừứíìịỉĩýỳỷỵỹđ"
    intab_u = "ẠẢÃÀÁÂẬẦẤẨẪĂẮẰẶẲẴÓÒỌÕỎÔỘỔỖỒỐƠỜỚỢỞỠÉÈẺẸẼÊẾỀỆỂỄÚÙỤỦŨƯỰỮỬỪỨÍÌỊỈĨÝỲỶỴỸĐ"
    intab = list(intab_l+intab_u)

    outtab_l = "a"*17 + "o"*17 + "e"*11 + "u"*11 + "i"*5 + "y"*5 + "d"
    outtab_u = "A"*17 + "O"*17 + "E"*11 + "U"*11 + "I"*5 + "Y"*5 + "D"
    outtab = outtab_l + outtab_u

    r = re.compile("|".join(intab))
    replaces_dict = dict(zip(intab, outtab))
    return r.sub(lambda m: replaces_dict[m.group(0)], utf8_str)

remove_tone_line('Đi một ngày đàng học 1 sàng khôn')
```

```
'Di mot ngay dang hoc 1 sang khon'
```

Để ý thấy hầu hết các dấu câu sẽ dính liền với câu. Chẳng hạn câu `tại khanh blog, tôi lưu lại các bài viết như một quyển nhật kí` . Thì từ `blog` và dấu phảy bị dính liền. Vì vậy ta cần sử dụng hàm `normalizeString()` để tách các dấu ra khỏi từ.

```
# Tách dấu ra khỏi từ
def normalizeString(s):
    # Tách dấu câu nếu kí tự liền nhau
    marks = '[.!?,-${}()]'
    r = "(["+"\\".join(marks)+"])"
    s = re.sub(r, r" \1 ", s)
    # Thay thế nhiều spaces bằng 1 space.
    s = re.sub(r"\s+", r" ", s).strip()
    return s

normalizeString('vui vẻ, hòa đồng, hoạt bát')
```

```
'vui vẻ , hòa đồng , hoạt bát'
```

Sử dụng 2 hàm số trên để tạo ra các list `train` gồm các câu có dấu và `train_rev_accent` gồm các câu không dấu.

```
import itertools
train = [normalizeString(item) for item in train]
train_rev_accent = [remove_tone_line(item) for item in train]

print('train top 5:', train[:5])
print('train_rev_accent top 5:', train_rev_accent[:5])
```

```
train top 5: ['Bộ phim lần đầu được công chiếu tại liên hoan phim Rome 2007 và sau đó đượ
train_rev_accent top 5: ['Bo phim lan dau duoc cong chieu tai lien hoan phim Rome 2007 va
```

1.2. Load and Trim data

Bên dưới ta sẽ xây dựng class Voc để xây dựng từ điển cho tập dữ liệu. Các hàm trong Voc có tác dụng:

- `addWord()` : Kiểm tra xem từ đã xuất hiện trong từ điển chưa. Nếu chưa sẽ thêm từ mới đó vào từ điển.
- `addSentence()` : Truyền vào 1 câu và thêm các từ trong câu vào từ điển.
- `trim()` : Loại bỏ các từ hiếm trong từ điển nếu nó có ngưỡng số tần xuất xuất hiện trong toàn bộ tập dữ liệu ít hơn `min_count` .

Top

```
# Default word tokens
PAD_token = 0  # Used for padding short sentences
SOS_token = 1  # Start-of-sentence token
EOS_token = 2  # End-of-sentence token


class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3  # Count SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []

        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) / len(self.word2index
        ))

        # Reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # Count default tokens

        for word in keep_words:
            self.addWord(word)
```

1.2. Chuẩn hóa dữ liệu

1.2.1 Tạo ngram

Trong dữ liệu có những câu rất dài làm giảm hiệu quả dự báo của mô hình. Do đó chúng ta sẽ tìm cách tạo ra các ngram với độ dài đo bằng số lượng từ cố định để dự báo trong một khoảng cách ngắn. Ở bài này ta sẽ lựa chọn ngram = 4.

Top

```
def _ngram(text, length = 4):
  words = text.split()
  grams = []
  if len(words) <= length:
    words = words + ["PAD"]*(length-len(words))
    return [' '.join(words)]
  else:
    for i in range(len(words)-length+1):
      grams.append(' '.join(words[i:(i+length)]))
    return grams

print(_ngram('mùa đông năm nay không còn lạnh nữa. Vì đã có gấu 37 độ ấm'))
print(_ngram('mùa đông'))
```

```
['mùa đông năm nay', 'đông năm nay không', 'năm nay không còn', 'nay không còn lạnh', 'kh
['mùa đông PAD PAD']
```

```
import itertools

train_grams = list(itertools.chain.from_iterable([_ngram(item) for item in train]))
train_rev_acc_grams = list(itertools.chain.from_iterable([_ngram(item) for item in train_
```

```
corpus = list(zip(train_rev_acc_grams, train_grams))
corpus[:5]
```

```
[('Bo phim lan dau', 'Bộ phim lần đầu'),
 ('phim lan dau duoc', 'phim lần đầu được'),
 ('lan dau duoc cong', 'lần đầu được công'),
 ('dau duoc cong chieu', 'đầu được công chiếu'),
 ('duoc cong chieu tai', 'được công chiếu tại')]
```

Top

```
import unicodedata

MAX_LENGTH = 4  # Maximum sentence length to consider

# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters
def normalizeString(s):
    # Tách dấu câu nếu kí tự liền nhau
    s = re.sub(r"([.!?,\-\&\(\)\[\]])", r" \1 ", s)
    # Thay thế nhiều spaces bằng 1 space.
    s = re.sub(r"\s+", r" ", s).strip()
    return s

# Read query/response pairs and return a voc object
def readVocs(lines, corpus_name = 'corpus'):
    # Split every line into pairs and normalize
    pairs = [[normalizeString(str(s)) for s in l] for l in lines]
    voc = Voc(corpus_name)
    return voc, pairs

voc, pairs = readVocs(corpus)

# Returns True iff both sentences in a pair 'p' are under the MAX_LENGTH threshold
def filterPair(p):
    # Input sequences need to preserve the last word for EOS token
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# Filter pairs using filterPair condition
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# # Using the functions defined above, return a populated voc object and pairs list
def loadPrepareData(voc, pairs):
    print("Read {!s} sentence pairs".format(len(pairs)))
    # pairs = filterPairs(pairs)
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs

# Load/Assemble voc and pairs
save_dir = os.path.join("data", "save")
voc, pairs = loadPrepareData(voc, pairs)
# Print some pairs to validate
print("\npairs:")
for pair in pairs[:10]:
    print(pair)
```

Top

```
Read 9771412 sentence pairs
Trimmed to 9771412 sentence pairs
Counting words...
Counted words: 202153

pairs:
['Bo phim lan dau', 'Bộ phim lần đầu']
['phim lan dau duoc', 'phim lần đầu được']
['lan dau duoc cong', 'lần đầu được công']
['dau duoc cong chieu', 'đầu được công chiếu']
['duoc cong chieu tai', 'được công chiếu tại']
['cong chieu tai lien', 'công chiếu tại liên']
['chieu tai lien hoan', 'chiếu tại liên hoan']
['tai lien hoan phim', 'tại liên hoan phim']
['lien hoan phim Rome', 'liên hoan phim Rome']
['hoan phim Rome 2007', 'hoan phim Rome 2007']
```

Model sẽ huấn luyện nhanh hơn nếu:

- Giảm bớt số lượng các token không quá phổ biến.
- Loại bỏ bớt các câu không phổ biến.

Bên dưới ta sẽ xây dựng hàm số loại bỏ các token có tần suất nhỏ hơn ngưỡng MIN_COUNT và loại bỏ các câu có chứa token vừa bị loại bỏ.

```
MIN_COUNT = 3    # Minimum word count threshold for trimming

def trimRareWords(voc, pairs, MIN_COUNT):
    # Trim words used under the MIN_COUNT from the voc
    voc.trim(MIN_COUNT)
    # Filter out pairs with trimmed words
    keep_pairs = []
    for pair in pairs:
        input_sentence = pair[0]
        output_sentence = pair[1]
        keep_input = True
        keep_output = True
        # Check input sentence
        for word in input_sentence.split(' '):
            if word not in voc.word2index:
                keep_input = False
                break
        # Check output sentence
        for word in output_sentence.split(' '):
            if word not in voc.word2index:
                keep_output = False
                break

        # Only keep pairs that do not contain trimmed word(s) in their input or output s
        if keep_input and keep_output:
            keep_pairs.append(pair)

    print("Trimmed from {} pairs to {}, {:.4f} of total".format(len(pairs), len(keep_pai
    return keep_pairs


# Trim voc and pairs
pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

```
keep_words 171772 / 202150 = 0.8497
Trimmed from 9771412 pairs to 9741582, 0.9969 of total
```

Như vậy sau khi loại bỏ các từ hiếm với tần xuất xuất hiện <= 3 thì dữ liệu còn lại 87% số lượng các token và 99.81% các câu được dữ lại.

1.3. Tạo batch huấn luyện

Top

```
print('EOS_token: ', EOS_token)
print('SOS_token: ', SOS_token)
print('PAD_token: ', PAD_token)
```

```
EOS_token:  2
SOS_token:  1
PAD_token:  0
```

Bên dưới ta sẽ xây dựng các hàm chức năng, trong đó:

- `indexesFromSentence()` : Mã hóa câu văn thành chuỗi index của các token theo giá trị của cặp word2index trong từ điển và đính thêm token EOS ở cuối để đánh dấu kết thúc câu. Chẳng hạn trong từ điển từ các từ tương ứng với index như sau: {'học':5, 'sinh':7, 'đi':9, 'EOS':2}, khi đó giá trị mã hóa index của câu 'học sinh đi học' sẽ là [5, 7, 9, 5,2].
- `zeroPadding()` : Nhận đầu vào là 1 list các chuỗi index đại diện cho câu. Hàm này sẽ xác định câu có độ dài lớn nhất trong list. Sau đó padding thêm 0 vào cuối mỗi chuỗi index các giá trị 0 về cuối để các câu có độ dài bằng nhau và bằng độ dài lớn nhất.
- `binaryMatrix()` : Một ma trận sẽ biểu diễn một batch của các câu truyền vào. Mỗi dòng của ma trận sẽ đại diện cho 1 câu. Trong ma trận này sẽ tồn tại những index tương ứng với vị trí padding. binaryMatrix sẽ đánh dấu các vị trí mà tương ứng với padding bằng 0 và tương ứng với từ bằng 1.
- `inputVar()` : Nhận giá trị truyền vào là 1 list các câu input và từ điển. Hàm sẽ trả về ma trận được padding thêm 0 đại diện cho list các câu input và list độ dài tương ứng thực tế của các câu.
- `outputVar()` : Nhận giá trị truyền vào là 1 list các câu output và từ điển. Về cơ bản cũng giống như `inputVar()` nhưng ngoài trả về ma trận padding và độ dài lớn nhất của các câu còn trả về thêm ma trận mask có kích thước bằng ma trận padding đánh dấu các vị trí là padding (=0) và từ (=1).
- `batch2TrainData()` : Nhận giá trị truyền vào là các cặp câu (input, output) và từ điển. Hàm số sẽ khởi tạo batch cho huấn luyện mô hình bao gồm: ma trận batch input, ma trận batch output, ma trận mask đánh đấu padding của output. Ngoài ra còn trả thêm list độ dài thực tế các câu trong input và độ dài lớn nhất của các câu trong output.

Top

```
import random
import torch

def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split(' ')] + [EOS_token]

# Padding thêm 0 vào list nào có độ dài nhỏ hơn về phía bên phải
def zeroPadding(l, fillvalue=PAD_token):
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))

# Tạo ma trận binary có kích thước như ma trận truyền vào l nhưng giá trị của mỗi phần t
def binaryMatrix(l, value=PAD_token):
    m = []
    for i, seq in enumerate(l):
        m.append([])
        for token in seq:
            if token == PAD_token:
                m[i].append(0)
            else:
                m[i].append(1)
    return m

# Returns padded input sequence tensor and lengths
def inputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    padVar = torch.LongTensor(padList)
    return padVar, lengths

# Returns padded target sequence tensor, padding mask, and max target length
def outputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    max_target_len = max([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    mask = binaryMatrix(padList)
    mask = torch.ByteTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len

# Returns all items for a given batch of pairs
def batch2TrainData(voc, pair_batch):
    pair_batch.sort(key=lambda x: len(x[0].split(" ")), reverse=True)
    input_batch, output_batch = [], []
    for pair in pair_batch:
        input_batch.append(pair[0])
        output_batch.append(pair[1])
    inp, lengths = inputVar(input_batch, voc)
    output, mask, max_target_len = outputVar(output_batch, voc)
    return inp, lengths, output, mask, max_target_len


# Example for validation
small_batch_size = 4
batches = batch2TrainData(voc, [random.choice(pairs) for _ in range(small_batch_size)])
input_variable, lengths, target_variable, mask, max_target_len = batches
```

Cuối cùng ta sẽ thử nghiệm giá trị trả ra của hàm `batch2TrainData()` khi lựa chọn ra 4 cặp câu bất kì trong list các cặp (input, output).

```
print("input_variable: \n", input_variable)
print("lengths: \n", lengths)
print("target_variable: \n", target_variable)
print("mask: \n", mask)
print("max_target_len: \n", max_target_len)
```

Top

```
input_variable:
 tensor([[  66,  369,   66, 1272],
        [ 567,  183,   28,  616],
        [ 392, 1558, 1143,  175],
        [ 394,   31,   31, 5558],
        [   2,    2,    2,    2]])
lengths:
 tensor([5, 5, 5, 5])
target_variable:
 tensor([[  66,  370,  124, 1275],
        [ 568,  184,   29,  617],
        [ 393, 1560, 1144, 6240],
        [ 395,   31,   31, 5559],
        [   2,    2,    2,    2]])
mask:
 tensor([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]], dtype=torch.uint8)
max_target_len:
 5
```

2. Mô hình seq2seq sửa lỗi chính tả

Model seq2seq sẽ nhận đầu vào là 1 chuỗi và trả ra kết quả output cũng là 1 chuỗi. Chính vì thế tên gọi của mô hình là sequence to sequence (từ câu đến câu).

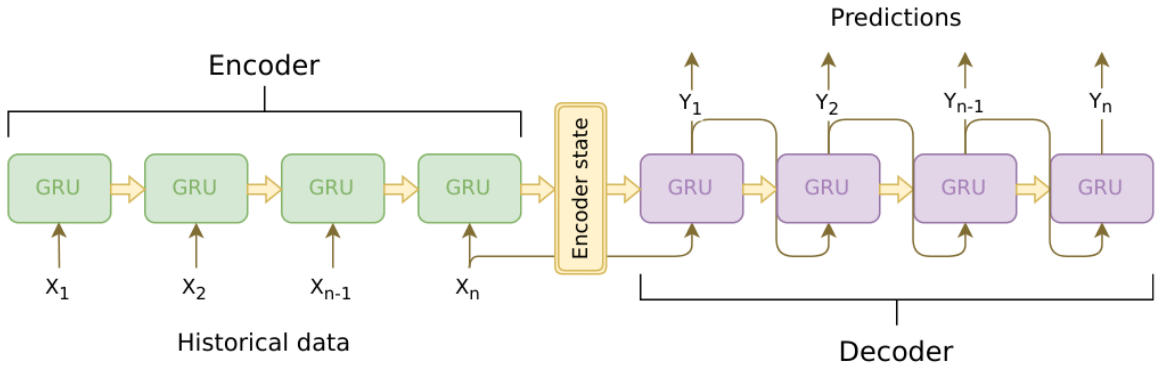
Trong kiến trúc của model seq2seq sẽ gồm 2 phrases: Encoder và decoder.

- **Encoder**: Nhúng các từ thành những véc tơ embedding với kích thước tùy ý. Encoder sẽ xây dựng một chuỗi các xử lý liên hoàn sao cho output của bước liền trước là input của bước liền sau. Khi đó tại mỗi time step sẽ truyền đầu vào là các véc tơ đã được mã hóa ứng với mỗi từ $\mathbf{x_i}$. Encoder sẽ trả ra 2 kết quả ở đầu ra gồm: encoder outputs đại diện cho toàn bộ câu input trong đó mỗi véc tơ của encoder outputs đại diện cho 1 từ trong câu và hidden state của GRU cuối cùng. hidden state sẽ được sử dụng để làm giá trị hidden khởi tạo cho quá trình Decoder (chi tiết ở hình 1 bên dưới). ma trận encoder outputs được sử dụng để tính attention weight tại mỗi time step trong phrase decoder. Để dễ hình dung output của một mạng RNN chúng ta có thể xem thêm lý thuyết về mạng LSTM (https://phamdinhkhanh.github.io/2019/04/22/L%C3%BD_thuy%E1%BA%BFt_v%E1%BB%81_m%E1%BA%A1ng_LSTM.htm
- Decoder: Sau phrase encoder ta sẽ thu được một hidden state của GRU cuối cùng và ma trận encoder outputs đại điện cho toàn bộ câu input. Phrase decoder có tác dụng giải mã thông tin đầu ra ở encoder thành các từ. Do đó tại mỗi time step đều trả ra các véc tơ phân phối xác xuất của từ tại bước đó. Từ đó ta có thể xác định được từ có khả năng xảy ra nhất tại mỗi time step. Tại time step t, mô hình sẽ kết hợp giữa decoder embedding véc tơ h_t đại diện cho token $word_t$ và ma trận encoder outputs theo cơ chế global attention (được đề xuất bởi anh Lương Mạnh Thắng) để tính ra trọng số attention weight phân bố cho vị trí từ ở câu input lên $word_t$. véc tơ context đại diện cho toàn bộ câu input sẽ được tính bằng tích trọng số của attention weight với từng encoder véc tơ của ma trận encoder outputs (cụ thể hình 3). Tiếp theo để dự báo cho từ kế tiếp $word_{t+1}$ ta cần kết hợp véc tơ decoder hidden state h_{t+1} và véc tơ context như hình 5. Qúa trình này sẽ được lặp lại liên tục cho đến khi gặp token cuối cùng là `<EOS>` đánh dấu vị trí cuối của câu. Như vậy ta sẽ trải qua các bước:
    - Tính decoder input chính là embedding véc tơ h_t của từ đã biết $word_t$ dựa vào embedding layer.
    - Tính attention weight đánh giá mức độ tập trung của các từ input vào từ được dự báo dựa vào ma trận encoder outputs và h_t.
    - Tính context véc tơ c_t chính là tích có trọng số của attention weights với các véc tơ đại diện cho mỗi từ input trong ma trận encoder outputs.
    - Truyền decoder input và last hidden state ở bước t vào mô hình decoder GRU để tính ra được decoder hidden state h_{t+1}.
    - Kết hợp decoder hidden state h_{t+1} và context véc tơ c_t để dự báo từ tại time step $t+1$.

Quá trình tiếp tục cho đến khi gặp token `<EOS>` đánh dấu kết thúc câu.

Kết quả đầu ra là chuỗi các indexes (lấy theo vocabulary) đại diện cho từng từ tại mỗi vị trí của câu.

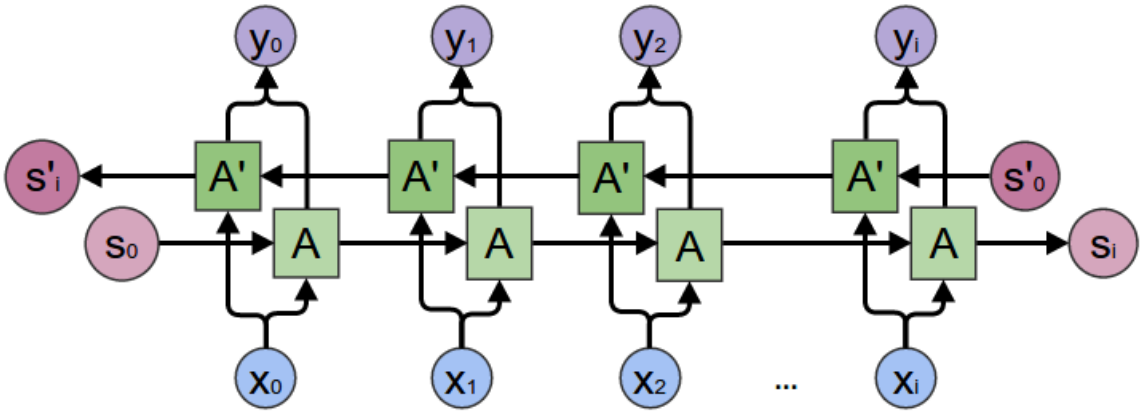
Top



Hình 1: Sơ đồ Encoder và Decoder sử dụng mạng GRU trong mô hình seq2seq.

2.1. Encoder

Tại phrase encoder chúng ta sẽ sử dụng kiến trúc mạng bidirectional GRU (2 chiều) như bên dưới để mã hóa thông tin.



Hình 2: Kiến trúc mạng bidirectional GRU. Khác với mạng unidirectional GRU (1 chiều) chỉ có chiều từ trái qua phải. Mạng GRU 2 chiều sẽ đánh giá thêm chiều từ phải qua trái. Điều này giúp cho việc học được đầy đủ hơn khi sự phụ thuộc của các từ trong câu luôn tuân theo cả 2 chiều.

Lưu ý một embedding layer được áp dụng để mã hóa các từ về một véc tơ với kích thước được khai báo là `hidden_size` . Khi huấn luyện xong model, embedding layer sẽ có kết quả sao cho những từ gần nghĩa sẽ được đại diện bởi những vec tơ sao cho độ tương quan về nghĩa càng lớn. Độ tương quan này được đo bằng cosin similarity của các embedding véc tơ của mỗi từ.

Mô hình RNN sẽ nhận đầu vào là một batch. Để padding một batch vào model RNN thì chúng ta phải pack dữ liệu bằng hàm `nn.utils.rnn.pack_padded_sequence` để tự động padding thêm 0 vào các véc tơ từ. Và ở bước decoder chúng ta phải unpack phần zero padding bao quanh đầu ra bằng hàm `nn.utils.rnn.pad_packed_sequence` .

Bên dưới ta sẽ xây dựng Module encoder.

Quá trình tính toán của đồ thị:

1. Mã hóa các từ về index và embedding các index thành các véc tơ.
2. Xác định batch cho mô hình. Padding câu về chung 1 độ dài và đóng gói thành batch bằng hàm `nn.utils.rnn.pack_padded_sequence` .
3. Thực hiện quá trình feed forward.
4. unpack các zero padding bằng hàm `nn.utils.rnn.pad_packed_sequence`
5. Tính tổng của bidirectional GRU outputs theo hai chiều trái và phải.
6. Trả về encoder output của layer GRU cuối cùng và hidden state tại layer cuối cùng.

Các tham số của model:

Inputs:

Top

- `input_seq` : Batch của các câu đầu vào dưới dạng tensor (như tham số input được trả ra của hàm `batch2trainData()`). `shape = max_length x batch_size` . Trong đó `batch_size` là kích thước batch (tương ứng với số câu được đưa vào batch) và `max_length` là kích thước của câu văn đã được pad hoặc trim về độ dài chuẩn.
- `input_lengths` : list của độ dài thực tế (không tính pad hoặc trim) các câu tương ứng trong batch.

Outputs:

- `outputs` : hidden layer cuối cùng của GRU (tổng của bidirectional outputs). Trên hình chính là GRU tại vị trí $\mathbf{x_n}$; `shape = (max_length, batch_size, hidden_size)` . Trong đó `max_length` là độ dài của câu. `batch_size` là kích thước batch. `hidden_size` là số chiều mã hóa của embedding layer.
- `hidden` : là ma trận concatenate của các hidden state được cập nhật từ mỗi layer của GRU; `shape = (n_layers x num_directions, batch_size, hidden_size)` . Trong đó:
  - `n_layers` là số layers GRU được áp dụng. Số lượng layers này sẽ bằng chính số lượng hidden state của mạng GRU.
  - `hidden_size` chính là kích thước của embedding véc tơ ứng với mỗi từ.
  - `num_bidirections` là số chiều của mạng GRU. Trong trường hợp này là 2 chiều.

Trước tiên ta cần khai báo sử dụng pytorch cuda bằng đoạn code bên dưới. hàm `torch.cuda.is_available()` sẽ tự động kiểm tra xem máy có GPU hỗ trợ CUDA hay không. Nếu có sẽ khai báo `device` là `cuda` và trái lại sẽ sử dụng `cpu` để huấn luyện mô hình. Và tất nhiên là tốc độ huấn luyện trên `cuda` nhanh hơn rất nhiều.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import torch
from torch.jit import script, trace
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import csv
import random
import re
import os
import unicodedata
import codecs
from io import open
import itertools
import math


USE_CUDA = torch.cuda.is_available()
device = torch.device("cuda" if USE_CUDA else "cpu")
```

Top

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # Initialize GRU; the input_size and hidden_size params are both set to 'hidden_
        # set bidirectional = True for bidirectional
        # https://pytorch.org/docs/stable/nn.html?highlight=gru#torch.nn.GRU to get more
        self.gru = nn.GRU(input_size = hidden_size, # number of expected feature of inpu
                          hidden_size = hidden_size, # number of expected feature of hic
                          num_layers = n_layers, # number of GRU layers
                          dropout=(0 if n_layers == 1 else dropout), # dropout probabili
                          bidirectional=True # one or two directions.
                         )

    def forward(self, input_seq, input_lengths, hidden=None):
        # Step 1: Convert word indexes to embeddings
        # shape: (max_length , batch_size , hidden_size)
        embedded = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module. Padding zero when length less t
        # shape: (max_length , batch_size , hidden_size)
        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)
        # Step 2: Forward packed through GRU
        # outputs is output of final GRU layer
        # hidden is concatenate of all hidden states corresponding with each time step.
        # outputs shape: (max_length , batch_size , hidden_size x num_directions)
        # hidden shape: (n_layers x num_directions , batch_size , hidden_size)
        outputs, hidden = self.gru(packed, hidden)
        # Unpack padding. Revert of pack_padded_sequence
        # outputs shape: (max_length , batch_size , hidden_size x num_directions)
        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
        # Sum bidirectional GRU outputs to reshape shape into (max_length , batch_size ,
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size:]
        # outputs shape:(max_length , batch_size , hidden_size)
        # hidden shape: (n_layers x num_directions , batch_size , hidden_size)
        return outputs, hidden
```

Để kiểm nghiệm kết quả trả ra của EncoderRNN bên dưới ta sẽ thực hiện 1 ví dụ giả lập encoder với `hidden_size` = 3 và `n_layers = 7`.

```
# Thử nghiệm phrase Encoder bằng cách giả lập 1 mạng Encoder với:
from torch import nn

hidden_size = 3
n_layers = 7
embedding = nn.Embedding(voc.num_words, hidden_size)
print('input_seq: \n', input_variable)
print('input_lengths: \n', lengths)
encoder = EncoderRNN(hidden_size = hidden_size, embedding = embedding, n_layers = n_laye

print('encoder phrase: \n', encoder)

output, hidden = encoder.forward(input_seq = input_variable, input_lengths = lengths)
```

Top

```
input_seq:
 tensor([[  66,  369,   66, 1272],
        [ 567,  183,   28,  616],
        [ 392, 1558, 1143,  175],
        [ 394,   31,   31, 5558],
        [   2,    2,    2,    2]])
input_lengths:
 tensor([5, 5, 5, 5])
encoder:
 EncoderRNN(
  (embedding): Embedding(171775, 3)
  (gru): GRU(3, 3, num_layers=7, bidirectional=True)
 )
```

```
print('output size: ', output.size())
print('hidden size: ', hidden.size())
```

```
output size:  torch.Size([5, 4, 3])
hidden size:  torch.Size([14, 4, 3])
```

Ta nhận thấy đầu ra của output là `(max_length, batch_size, hidden_size)` và của hidden là `(n_layers x num_directions, batch_size, hidden_size)`. Thông qua ví dụ này bạn đọc đã hình dung được quá trình Encoder rồi chứ.

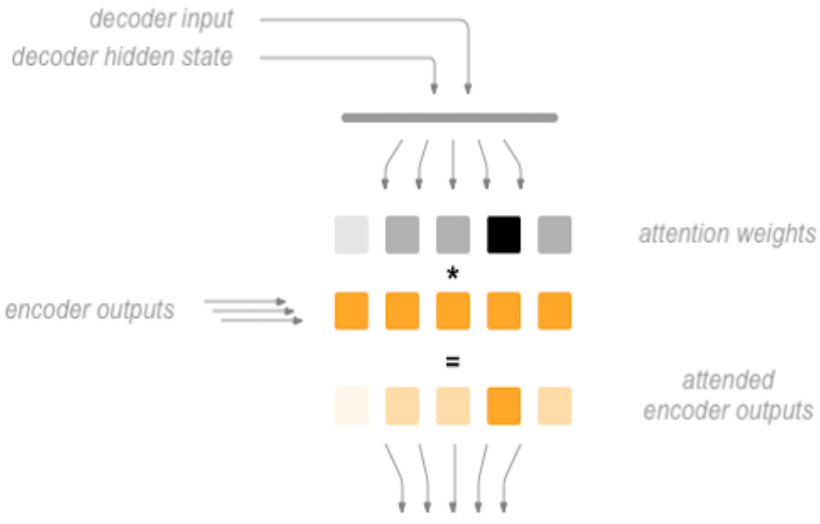
2.2. Decoder

Như vậy sau bước encoder ta thu được output là layer GRU cuối cùng (gọi là encoder outputs) và các hidden state của layer GRU cuối cùng. Bước tiếp theo chúng ta cần giải mã các kết quả thu được từ encoder thành câu hoàn chỉnh.

2.2.1. Áp dụng Attention layer

Ở phrase này chúng ta sẽ sử dụng thêm 1 layer attention để tính phân phối trọng số attention weights cho các véc tơ từ (hidden state) của ma trận encoder outputs. attention weight sẽ có kích thước bằng đúng độ dài của câu. Sau khi tính tổng theo attention weights ta sẽ thu được context véc tơ đại diện cho toàn bộ câu. Quá hình này được thể hiện như hình 5. context véc tơ sẽ kết hợp với các decoder hidden state (các thẻ h_t màu đỏ trong hình 5) tại mỗi time step để dự đoán từ tiếp theo. decoder hidden state chính là outptut trả ra của model Decoder sau mỗi time step. Quá trình này lặp lại truy hồi (output layer trước làm input cho layer sau) cho đến khi kết quả trả về là `<EOS>` (end of sequence).

Hình bên dưới sẽ minh họa cho quá trình tính toán attention weigths dựa trên sự kết hợp của decoder input (véc tơ embedding của từ được dự báo ở bước trước), decoder hidden state (ma trận của các hidden state sau cùng) và encoder outputs.



Hình 3: Kết hợp giữa decoder input, decoder hidden state và encoder outputs để tạo ra một attended encoder outputs.

Top

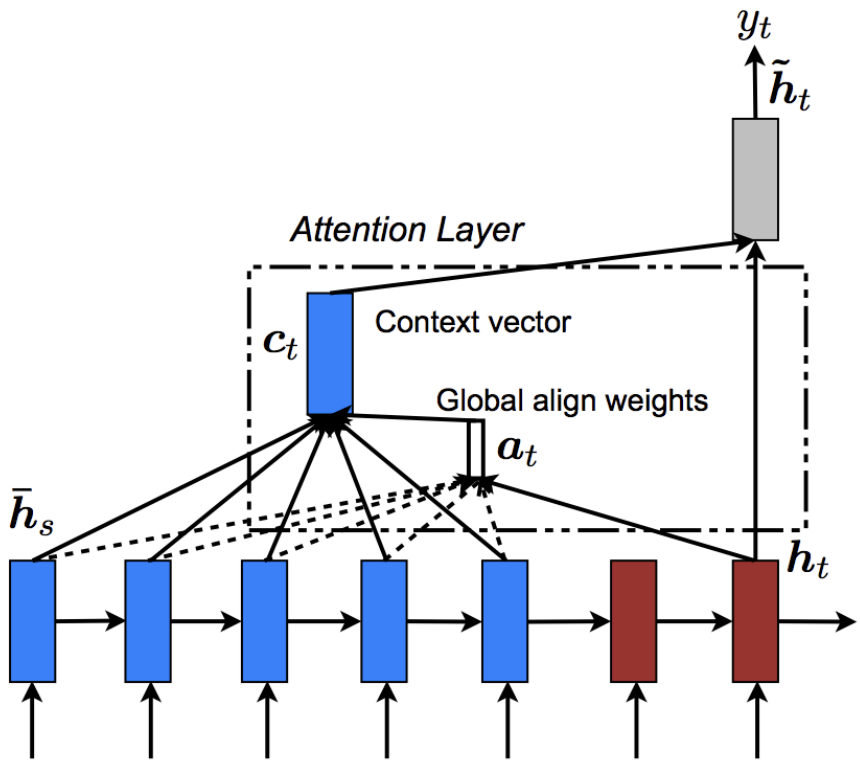
Các bạn đã hình dung được quá trình kết hợp attention vào decoder rồi chứ? Điểm mấu chốt là chúng ta phải tính ra được các trọng số attention tại mỗi time step. Việc tính toán attention weight đã được đề xuất theo rất nhiều cách khác nhau bởi anh Lương Mạnh Thắng (https://arxiv.org/abs/1508.04025). Trong đó điểm cải tiến so với cha đẻ của attention layer Bahdanau chính là một `global attention` được tính toán trên toàn bộ encoder hidden state thay vì `local attention` được tính toán dựa trên chỉ encoder hidden state của time step hiện tại. Điểm khác biệt thứ 2 là global attention tính attention weight chỉ dựa trên decoder hidden state tại time step hiện tại thay vì `local attention` được đề xuất bởi Bahdanau yêu cầu thêm các decoder hidden state trước đó. Theo đó điểm của các attention ở time step hiện tại được anh Thắng Lương đề xuất dưới nhiều công thức khác nhau như hình bên dưới.

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Hình 4: Tính điểm tại time step t dựa trên decoder hidden state (h_t) của thời điểm t và toàn bộ các encoder hidden states ($\bar{h_s}$).

Các bước tiếp theo để tính context véc tơ ta có thể tham khảo ở bài diễn giải về attention is all you need (https://phamdinhkhanh.github.io/2019/06/18/AttentionLayer.html).

Cơ chế tính context véc tơ có thể khái quát hóa như hình bên dưới. Lưu ý chúng ta sẽ triển khai `Attention Layer` như một `nn.Module` tách biệt và được gọi là `Attn` . Kết quả của attention layer là `attention weights` là một phân phối xác xuất hàm softmax dưới dạng tensor có kích thước `(batch_size, 1, max_length)` .



Hình 5: Cơ chế global attention.

Bên dưới là code của class `Attention` .

Top

```
# Luong attention layer
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        # encoder_output shape:(max_length , batch_size , hidden_size)
        # hidden shape: (1 , batch_size , hidden_size)
        # return shape: (max_length, batch_size)
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        # encoder_output shape:(max_length , batch_size , hidden_size)
        # hidden shape: (batch_size , hidden_size)
        # energy shape: (max_length , batch_size , hidden_size)
        # return shape: (max_length , batch_size)
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        # encoder_output shape:(max_length , batch_size , hidden_size)
        # hidden shape: (batch_size , hidden_size)
        # energy shape: (max_length , batch_size , 2*hidden_size)
        # self.v shape: (hidden_size)
        # return shape: (max_length , batch_size)
        energy = self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1, -1), enc
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # Calculate the attention weights (energies) based on the given method
        # attn_energies.shape: (max_length , batch_size)
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)

        # Transpose max_length and batch_size dimensions
        # attn_energies.shape: (batch_size , max_length)
        attn_energies = attn_energies.t()
        # Return the softmax normalized probability scores (with added dimension)
        attn_weights = F.softmax(attn_energies, dim=1).unsqueeze(1)
        # attn_weights shape: (batch_size , 1 , max_length)
        return attn_weights
```

Để hiểu rõ hơn attention hoạt động như thế nào ta sẽ cùng thử nghiệm truyền vào decoder hidden là các hidden state véc tơ của decoder và encoder outputs là ma trận mã hóa đầu ra của toàn bộ các từ trong câu input. Ví dụ bên dưới ta sẽ xét ở time step 0 nên decoder hidden khi đó sẽ là véc tơ hidden state cuối cùng của encoder (tham số last_hidden_encoder).

Top

```
method = 'dot'
# Take last hidden encoder vector as a initialize of decoder
last_hidden_encoder = hidden[encoder.n_layers, :, :].unsqueeze(0)
print('last_hidden_encoder.size: ', last_hidden_encoder.size())
print('encoder_outputs.size: ', output.size())
print('hidden_size: ', hidden_size)
decoder_hidden_size = 7
print('decoder_hidden_size: ', decoder_hidden_size)

attn = Attn(method = method, hidden_size = decoder_hidden_size)
attn_weights = attn.forward(hidden = last_hidden_encoder, encoder_outputs = output)
print('attn_weights shape: ', attn_weights.size())
```

```
last_hidden_encoder.size:  torch.Size([1, 4, 3])
encoder_outputs.size:  torch.Size([5, 4, 3])
hidden_size:  3
decoder_hidden_size:  7
attn_weights shape:  torch.Size([4, 1, 5])
```

Như vậy tại mỗi time step để tính attention weights chúng ta sẽ truyền vào một decoder hidden véc tơ (chính là output của GRU tại time step đó) và ma trận encoder outputs. Thông qua các phương pháp dot_score, general_score hoặc concat_score ta sẽ tính được attention weights đại diện cho mức độ attention của từng vị trí của encoder outputs lên từ được dự báo.

Qua diễn giải trên chúng ta mới chỉ hiểu cách thức mà attention hoạt động. Vậy thì attention được áp dụng trong toàn bộ quá trình decoder như thế nào tại mỗi time step. Để hiểu rõ chúng ta tìm hiểu qua phrase decoder.

2.2.2. Quá trình Decoder

Trong bước decoder chúng ta sẽ truyền một từ một lần tại mỗi time step. Do đó các véc tơ embedding của từ và GRU output phải có chung shape là `(1, batch_size, hidden_size)`.

Quá trình tính toán đồ thị

1. Lấy embedding của input word hiện tại.
2. thực hiện lan truyền thuận (feed forward) qua một GRU 1 chiều (unidirectional GRU) để thu được decoder hidden state.
3. Tính attention weights từ decoder hidden state của GRU hiện tại ở bước 2 kết hợp với encoder outputs của encoder.
4. Nhân attention weights với encoder outputs để thu được một tổng có trọng số là context véc tơ.
5. Concatenate context véc tơ và GRU output.
6. Dự báo từ tiếp theo (không sử dụng softmax).
7. Trả về output và hidden state cuối cùng.

Inputs

- `input_step` : một bước thời gian tương ứng với 1 từ của input sequence; shape = (1, batch_size).
- `last_hidden` : layer GRU cuối cùng; shape=(n_layers x num_directions, batch_size, hidden_size).
- `encoder_outputs` : đầu ra của bước encoder; shape = (max_length, batch_size, hidden_size).

Outputs

- `output` : softmax normalized tensor trả về xác xuất tương ứng với mỗi từ là từ tại vị trí tương ứng của câu; shape=(batch_size, voc.num_words)
- `hidden` : hidden state cuối cùng của GRU; shape=(n_layers x num_directions, batch_size, hidden_size). Do ở phrase decoder ta chỉ áp dụng unidirectional GRU nên shape = (n_layers , batch_size, hidden_size).

Top

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, drop
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers ==
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        '''
        input_step: list time step index of batch. shape (1 x batch_size)
        last_hidden: last hidden output of hidden layer (we can take in right direction
        encoder_outputs: output of encoder
        '''
        #============================================
        # Step 1: Embedding current sequence index
        # Note: we run this one step (word) at a time
        # Get embedding of current input word
        # embedded shape: 1 x batch_size x hidden_size
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)

        #============================================
        # Step 2: pass embedded and last hidden into decoder
        # Forward through unidirectional GRU
        # rnn_output shape: 1 x batch_size x hidden_size
        # hidden shape: n_layers x batch_size x hidden_size
        rnn_output, hidden = self.gru(embedded, last_hidden)
        # Calculate attention weights from the current GRU output
        # attn_weights shape: batch_size x 1 x max_length
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # Multiply attention weights to encoder outputs to get new "weighted sum" contex
        # encoder_outputs shape: max_length x batch_size x hidden_size
        # context shape: batch_size x 1 x hidden_size
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # Concatenate weighted context vector and GRU output using Luong eq. 5
        # rnn_output shape: batch_size x hidden_size
        rnn_output = rnn_output.squeeze(0)
        # context shape: batch_size x hidden_size
        context = context.squeeze(1)

        #============================================
        # Step 3: calculate output probability distribution
        # concat_input shape: batch_size x (2*hidden_size)
        concat_input = torch.cat((rnn_output, context), 1)
        # concat_output shape: batch_size x hidden_size
        concat_output = torch.tanh(self.concat(concat_input))
        # Predict next word using Luong eq. 6
        # output shape: output_size
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        # Return output and final hidden state
        return output, hidden
```

Bên dưới ta sẽ giải thích quá trình dự đoán các từ tiếp theo tại time step đầu tiên. Lưu ý quá trình được thực hiện trên batch nên các đầu vào sẽ có kích thước scale theo batch_size.

Top

```
time_step = 0
# Take all index of batch at time step 0. All words are <SOS> mark for start of sentence
input_step = torch.tensor([SOS_token] * small_batch_size).unsqueeze(0)
n_layers = 7
# take last hidden vector of encoder
last_hidden = hidden[:n_layers]
print('batch_size: ', small_batch_size)
print('input_step.size at time_step 0: ', input_step.size())
print('last_hidden.size: ', last_hidden.size())
attn_model = 'dot'
hidden_size = 3
# Output size of decoder model is size of vocabulary
output_size = len(voc.word2index)


luongAttnDecoderRNN = LuongAttnDecoderRNN(attn_model = attn_model,
                                          embedding = embedding,
                                          hidden_size = hidden_size,
                                          output_size = output_size,
                                          n_layers = n_layers)

print('luongAttnDecoderRNN phrase: \n', luongAttnDecoderRNN)
dec_output, dec_hidden = luongAttnDecoderRNN.forward(input_step = input_step,
                                                     last_hidden = last_hidden,
                                                     encoder_outputs = output)
```

```
batch_size:  4
input_step.size at time_step 0:  torch.Size([1, 4])
last_hidden.size:  torch.Size([7, 4, 3])
luongAttnDecoderRNN phrase:
 LuongAttnDecoderRNN(
  (embedding): Embedding(171775, 3)
  (embedding_dropout): Dropout(p=0.1)
  (gru): GRU(3, 3, num_layers=7, dropout=0.1)
  (concat): Linear(in_features=6, out_features=3, bias=True)
  (out): Linear(in_features=3, out_features=171772, bias=True)
  (attn): Attn()
)
```

```
print('dec_output.size: ', dec_output.size())
print('dec_hidden.size: ', dec_hidden.size())
```

```
dec_output.size:  torch.Size([4, 171772])
dec_hidden.size:  torch.Size([7, 4, 3])
```

Diễn giải hàm forward:

Như vậy sau khi truyền vào decoder embedding véc tơ đại diện cho từ liền trước kết hợp với last hidden output của chính time step trước (có kích thước = `n_layers x hidden_size`) ta sẽ thu được decoder hidden state véc tơ (chính là các rnn_output trong hàm `luongAttnDecoderRNN.forward()`). Đây chính là một quá trình lan truyền thuật thông thường của một mạng RNN.

Tiếp theo kết hợp decoder hidden state véc tơ với encoder outputs thu được ở phrase encoding ta sẽ tính được attention weight và từ đó suy ra context véc tơ. concatenate context véc tơ và decoder hidden state véc tơ tạo thành feature learning. Truyền feature learning véc tơ này qua hàm softmax ta sẽ tìm được véc tơ phân phối của từ tại vị trí time step.

3. Huấn luyện model.

3.1 Loss function

Bởi vì chúng ta huấn luyện mô hình dựa trên các batch đã được padding nên không đơn thuần là đưa toàn bộ các phần tử của output vào để tính toán giá trị của loss function mà cần có thêm một binary mask tensor để xác định các vị trí padding của target tensor. Khi đó loss function được tính toán dựa trên decoder's output tensor, target tensor và binary

Top

mask tensor. Hàm loss function chính là trung bình âm của log likelihood của toàn bộ các phần tử tương ứng với 1 trong 1 trong binary mask tensor.

```
def maskNLLLoss(inp, target, mask):
    nTotal = mask.sum()
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)).squeeze(1))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

3.2. Huấn luyện vòng lặp đơn

Hàm huấn luyện `train()` bên dưới sẽ được xây dựng để huấn luyện cho 1 batch đơn lẻ.

Các kĩ thuật sử dụng

Chúng ta sẽ sử dụng 2 kĩ thuật để hỗ trợ hội tụ:

- **teacher forcing**: Tại một ngưỡng xác xuất nào đó được thiết lập thông qua tham số `teacher_forcing_ratio`, chúng ta sẽ sử dụng luôn current target word như là đầu vào cho decoder tại bước tiếp theo hơn là sử dụng dự báo từ mô hình tại bước hiện tại. Kĩ thuật này hỗ trợ cho quá trình huấn luyện hiệu quả hơn. Tuy nhiên điểm hạn chế là có thể khiến cho mô hình thiếu ổn định trong quá trình suy diễn, khi decoder có thể không có cơ hội học đủ các khả năng để tạo ra output sequence trong quá trình huấn luyện. Do đó phải hết sức cẩn thận khi sử dụng `teacher_forcing_ratio`.
- **gradient clipping**: Hay còn gọi là kĩ thuật kẹp gradient để tránh hiện tượng bùng nổ gradient (exploding gradient - gradient lớn quá nhanh) bằng các thiết lập giá trị max cho gradient descent.

Chuỗi các bước triển khai

1. Truyền vào toàn bộ các batch của câu qua encoder.
2. Khởi tạo decoder input bằng token `<SOS>` (start of sequence) và encoder's hidden state cuối cùng.
3. Truyền input batch sequence vào decoder một lần tại một time step.
4. Nếu thực hiện teaching forcing: Thiết lập decoder input tiếp theo chính là giá trị target hiện tại; trái lại: thiết lập decoder input tiếp theo như là đầu ra của decoder hiện tại.
5. Tính loss function.
6. Thực hiện lan truyền ngược.
7. Kẹp gradient tránh hiện tượng bùng nổ tham số.
8. Cập nhật các tham số của encoder và decoder cho mô hình.

Top

```
def train(input_variable, lengths, target_variable, mask, max_target_len, encoder, decoc
          encoder_optimizer, decoder_optimizer, batch_size, clip, max_length=MAX_LENGTH)

    # Zero gradients
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Set device options
    input_variable = input_variable.to(device)
    lengths = lengths.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)

    # Initialize variables
    loss = 0
    print_losses = []
    n_totals = 0

    # Forward pass through encoder
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # Create initial decoder input (start with SOS tokens for each sentence)
    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # Set initial decoder hidden state to the encoder's final hidden state
    decoder_hidden = encoder_hidden[:decoder.n_layers]

    # Determine if we are using teacher forcing this iteration
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    # Forward batch of sequences through decoder one time step at a time
    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # Teacher forcing: next input is current target
            decoder_input = target_variable[t].view(1, -1)
            # Calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal
    else:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # No teacher forcing: next input is decoder's own current output
            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in range(batch_size)]])
            decoder_input = decoder_input.to(device)
            # Calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal

    # Perform backpropatation
    loss.backward()

    # Clip gradients: gradients are modified in place
    _ = nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    _ = nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # Adjust model weights
    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals
```

Top

3.3. Huấn luyện vòng lặp

Cuối cùng đã đến lúc gắn kết toàn bộ quy trình huấn luyện với dữ liệu. Hàm `trainIters` chịu trách nhiệm chạy các `n_interations` quá trình huấn luyện từ các đầu vào gồm các model, hàm tối ưu optimizer, dữ liệu, v.v.

Một lưu ý là khi chúng ta lưu mô hình, chúng ta lưu vào một tarball (một dạng folder) chứa encoder và decoder state_dicts (chính là các tham số), optimizers' state_dicts, the loss, iteration, …. Lưu mô hình theo cách này sẽ mang lại sự linh hoạt cho mô hình nhờ checkpoint. Sau khi load một checkpoint, chúng ta sẽ có thể sử dụng các tham số mô hình để chạy suy diễn hoặc có thể tiếp tục huấn luyện ngay tại trạng thái rời đi trước đó.

Top

```
def trainIters(model_name, voc, pairs, encoder, decoder,
               encoder_optimizer, decoder_optimizer,
               embedding, encoder_n_layers, decoder_n_layers,
               save_dir, n_iteration, batch_size, print_every,
               save_every, clip, corpus_name, loadFilename):
    '''
    model_name: Tên model
    voc: bộ từ vựng cho mô hình
    pairs: list các cặp (input, output)
    encoder: phrase encoder
    decoder: phrase decoder
    encoder_optimizer: phương pháp tối ưu hóa encoder
    decoder_optimizer: phương pháp tối ưu hóa decoder
    embedding: layer embedding
    encoder_n_layers: số lượng layers ở encoder
    decoder_n_layers: số lượng layers ở decoder
    save_dir: link save model
    n_iteration: số iteration tổng cộng được chọn ngẫu nhiên từ pairs. Mỗi iteration = 1
    batch_size: kích thước của batch.
    print_every: số iteration của batch để in kết quả
    save_every: số iteration của batch để save model
    clip: trimming giá trị của tensor về 1 range
    corpus_name: tên bộ từ vựng
    loadFilename: link load file
    '''
    # Load batches for each iteration
    training_batches = [batch2TrainData(voc, [random.choice(pairs) for _ in range(batch_
                      for _ in range(n_iteration)]

    # Initializations
    print('Initializing ...')
    start_iteration = 1
    print_loss = 0
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1

    # Training loop
    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        training_batch = training_batches[iteration - 1]
        # Extract fields from batch
        input_variable, lengths, target_variable, mask, max_target_len = training_batch

        # Run a training iteration with batch
        loss = train(input_variable, lengths, target_variable, mask, max_target_len, enc
                     decoder, embedding, encoder_optimizer, decoder_optimizer, batch_siz
        print_loss += loss

        # Print progress
        if iteration % print_every == 0:
            print_loss_avg = print_loss / print_every
            print("Iteration: {}; Percent complete: {:.1f}%; Average loss: {:.4f}".forma
            print_loss = 0

        # Save checkpoint
        if (iteration % save_every == 0):
            directory = os.path.join(save_dir, model_name, corpus_name, '{}-{}_{}'.forma
            if not os.path.exists(directory):
                os.makedirs(directory)
            torch.save({
                'iteration': iteration,
                'en': encoder.state_dict(),
                'de': decoder.state_dict(),
                'en_opt': encoder_optimizer.state_dict(),
                'de_opt': decoder_optimizer.state_dict(),
                'loss': loss,
                'voc_dict': voc.__dict__,
                'embedding': embedding.state_dict()
            }, os.path.join(directory, '{}_{}.tar'.format(iteration, 'checkpoint')))
```

Top

4. Xác định phương pháp đánh giá

Sau khi huấn luyện mô hình, chúng ta đã có khả năng giao tiếp với bot. Muốn như vậy cần phải xác định xem chúng ta muốn mô hình mã hóa đầu vào như thế nào. Chúng ta có những cách giải mã sau:

Mã hóa tham lam (greedy decoding) Các mã hóa này được sử dụng trong tính huống không sử dụng có teacher forcing. Đầu ra của mô hình tại mỗi time step đơn giản là từ với xác xuất cao nhất.

Bên dưới chúng ta tạo ra class `GreedySearchDecoder` nhận đầu vào là một câu input với shape (input_seq length, 1), một scalar độ dài đầu vào (input_length) tensor, và một max_length để giới hạn độ dài lớn nhất của câu trả về. Câu đầu vào sẽ được đánh giá qua các bước như đồ thị bên dưới.

Đồ thị tính toán:

1. Truyền input qua model encoder.
2. Chuẩn bị hidden layer cuối cùng của encoder làm hidden input đầu tiên của decoder.
3. Khởi tạo input đầu tiên của decoder là `<SOS>`
4. Khởi tạo tensor để append vào các từ được giải mã vào.
5. Lặp lại quá trình giải mã một word token 1 lần:
6. Truyền vào decoder.
7. Thu được word token dự báo dựa trên xác xuất lớn nhất.
8. Lưu lại token và score.
9. Coi token hiện tại như là đầu vào tiếp theo của decoder.
10. Trả về một tợp hợp các từ và điểm số.

```
class GreedySearchDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input_seq, input_length, max_length):
        # Forward input through encoder model
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)
        # Prepare encoder's final hidden layer to be first hidden input to the decoder
        decoder_hidden = encoder_hidden[:decoder.n_layers]
        # Initialize decoder input with SOS_token
        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token
        # Initialize tensors to append decoded words to
        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)
        # Iteratively decode one word token at a time
        for _ in range(max_length):
            # Forward pass through decoder
            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden,
            # Obtain most likely word token and its softmax score
            decoder_scores, decoder_input = torch.max(decoder_output, dim=1)
            # Record token and score
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)
            # Prepare current token to be next decoder input (add a dimension)
            decoder_input = torch.unsqueeze(decoder_input, 0)
        # Return collections of word tokens and scores
        return all_tokens, all_scores
```

Đánh giá câu dự báo Qua bước trên chúng ta đã có hàm decoder xác định câu đầu ra dựa trên câu đầu vào. Hàm đánh giá quản lý quá trình xử lý câu đầu vào mức độ thấp. Trước tiên, chúng ta định dạng câu dưới dạng một batch đầu vào của các word index với batch_size = 1. Chúng ta thực hiện điều này bằng cách chuyển đổi các từ của câu thành các indexes tương ứng và transpose để chuẩn bị tensor cho các mô hình. Chúng ta cũng tạo ra một tensor độ dài chứa độ dài của câu đầu vào. Trong trường hợp này, độ dài là scalar vì chúng ta chỉ đánh giá một câu tại một thời điểm (batch_size = 1). Tiếp theo, chúng ta thu được một tensor các câu trả về được giải mã bằng cách sử dụng object `GreedySearchDecoder` (trình tìm kiếm). Cuối cùng, chúng tôi chuyển đổi các indexes trả về thành các từ và trả về danh sách các từ được giải mã.

class `evaluateInput` hoạt động như giao diện người dùng cho chatbot. Khi được gọi, một input text field sẽ sinh ra trong đó chúng ta có thể nhập câu hỏi của mình. Sau khi nhập câu đầu vào và nhấn Enter, text sẽ được chuẩn hóa giống như dữ liệu huấn luyện và cuối cùng được đưa vào hàm đánh giá để thu được câu đầu ra được giải mã. Chúng ta lặp lại quy trình này cho liên tục cho đến khi nhấn `q` hoặc `quit` để quit.

Top

Cuối cùng, nếu một câu được nhập có chứa một từ không có trong từ vựng, chúng ta sẽ xử lý việc này một cách tinh tế bằng cách in một thông báo lỗi và nhắc người dùng nhập một câu khác.

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    ### Format input sentence as a batch
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # Create lengths tensor
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # Transpose dimensions of batch to match models' expectations
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # Use appropriate device
    input_batch = input_batch.to(device)
    lengths = lengths.to(device)
    # Decode sentence with searcher
    tokens, scores = searcher(input_batch, lengths, max_length)
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # Get input sentence
            input_sentence = input('> ')
            # Check if it is quit case
            if input_sentence == 'q' or input_sentence == 'quit': break
            # Normalize sentence
            input_sentence = normalizeString(input_sentence)
            # Evaluate sentence
            output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
            # Format and print response sentence
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")
```

5. Huấn luyện model

Bên dưới ta sẽ thực hiện huấn luyện mô hình bằng cách thiết lập các tham số. Chúng ta có thể thử nghiệm nhiều tham số cấu hình khác nhau để lựa chọn được 1 mô hình tối ưu. Chúng ta cũng có thể xây dựng mô hình từ đầu hoặc load từ checkpoint một mô hình sẵn có.

Load model

Top

```
# Configure models
model_name = 'correct_spelling_model'
corpus_name = 'corpus_aivivn'
attn_model = 'dot'
#attn_model = 'general'
#attn_model = 'concat'
hidden_size = 500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
batch_size = 100

# Set checkpoint to load from; set to None if starting from scratch
loadFilename = None
checkpoint_iter = 5000

# Load model if a loadFilename is provided
if loadFilename:
    # If loading on same machine the model was trained on
    checkpoint = torch.load(loadFilename)
    # If loading a model trained on GPU to CPU
    #checkpoint = torch.load(loadFilename, map_location=torch.device('cpu'))
    encoder_sd = checkpoint['en']
    decoder_sd = checkpoint['de']
    encoder_optimizer_sd = checkpoint['en_opt']
    decoder_optimizer_sd = checkpoint['de_opt']
    embedding_sd = checkpoint['embedding']
    voc.__dict__ = checkpoint['voc_dict']


print('Building encoder and decoder ...')
# Initialize word embeddings
embedding = nn.Embedding(voc.num_words, hidden_size)
# if loadFilename:
#     embedding.load_state_dict(embedding_sd)
# Initialize encoder & decoder models
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words, decoder
if loadFilename:
    encoder.load_state_dict(encoder_sd)
    decoder.load_state_dict(decoder_sd)
# Use appropriate device
encoder = encoder.to(device)
decoder = decoder.to(device)
print('Models built and ready to go!')
```

```
Building encoder and decoder ...
Models built and ready to go!
```

Khi muốn load một model từ check point chúng ta chỉ cần thay đổi loadFilename như bên dưới.

```
# Khi muốn load model thì enable đoạn dưới để tạo file lưu địa model training.
loadFilename = os.path.join(save_dir, model_name, corpus_name,
                            '{}-{}_{}'.format(encoder_n_layers, decoder_n_layers, hidden_s
                            '{}_checkpoint.tar'.format(checkpoint_iter))
```

5.1. Huấn luyện model

Để huấn luyện mô hình trước tiên ta cần thiết lập các tham số. Gọi vào hàm `trainIters` để huấn luyện qua các vòng lặp.

Top

```
# Configure training/optimization
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 5000
print_every = 100
save_every = 1000

# Ensure dropout layers are in train mode
encoder.train()
decoder.train()

# Initialize optimizers
print('Building optimizers ...')
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate * decoder_learning
if loadFilename:
    encoder_optimizer.load_state_dict(encoder_optimizer_sd)
    decoder_optimizer.load_state_dict(decoder_optimizer_sd)

# Run training iterations
print("Starting Training!")
trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer, decoder_optimize
           embedding, encoder_n_layers, decoder_n_layers, save_dir, n_iteration, batch_s
           print_every, save_every, clip, corpus_name, loadFilename)
```

```
Building optimizers ...
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 2.0%; Average loss: 6.7370
Iteration: 200; Percent complete: 4.0%; Average loss: 5.0068
Iteration: 300; Percent complete: 6.0%; Average loss: 3.5866
Iteration: 400; Percent complete: 8.0%; Average loss: 2.3804
Iteration: 500; Percent complete: 10.0%; Average loss: 1.8249
Iteration: 600; Percent complete: 12.0%; Average loss: 1.5482
Iteration: 700; Percent complete: 14.0%; Average loss: 1.3622
Iteration: 800; Percent complete: 16.0%; Average loss: 1.2253
Iteration: 900; Percent complete: 18.0%; Average loss: 1.1341
Iteration: 1000; Percent complete: 20.0%; Average loss: 1.0986
Iteration: 1100; Percent complete: 22.0%; Average loss: 1.0536
Iteration: 1200; Percent complete: 24.0%; Average loss: 1.0119
Iteration: 1300; Percent complete: 26.0%; Average loss: 0.9666
Iteration: 1400; Percent complete: 28.0%; Average loss: 0.9291
Iteration: 1500; Percent complete: 30.0%; Average loss: 0.9200
```

5.2 Đánh giá model

Bên dưới chúng ta cùng đánh giá mô hình thông qua việc dự đoán một số từ không dấu. Lưu ý rằng do dữ liệu được huấn luyện chỉ là các ngram có 4 từ nên chúng ta sẽ truyền vào các cụm 4 từ. Để thêm dấu cho 1 câu văn với độ dài tùy ý bạn đọc có thể chia câu thành các ngram với kích thước 4 và ghép các kết quả dự báo trên từng ngram đơn lẻ. Hàm này tôi sẽ không viết ở đây và xin dành cho bạn đọc.

```
# Set dropout layers to eval mode
encoder.eval()
decoder.eval()

# Initialize search module
searcher = GreedySearchDecoder(encoder, decoder)

# Begin chatting (uncomment and run the following line to begin)
evaluateInput(encoder, decoder, searcher, voc)
```

Top

```
> tai nguyen thien nhien
Bot: tài nguyên thiên nhiên
> thuong mai dien tu
Bot: thương mại điện tử
> hoc sinh cap 1
Bot: học sinh cấp 1
```

6. Hạn chế của mô hình

Mô hình seq2seq luôn tồn tại một số hạn chế nhất định mà chúng ta sẽ nhận ra trong quá trình huấn luyện đó là:

- Mô hình rất tốn tài nguyên và thời gian huấn luyện lâu. Để huấn luyện mô hình dịch máy, google đã huy động một hệ thống máy chủ mà diện tích có thể trải trên 1 km2.
- Mô hình sẽ không thêm dấu chính xác đối với những cụm từ mà chúng chưa từng được học. Do đó để nâng cao mức độ chuẩn xác ngoài cần một kiến trúc mô hình mạnh thì một tập dữ liệu đủ lớn.
- Trong bài tôi sử dụng mô hình theo word level. Chính vì thế kích thước của vocabolary là rất lớn và sẽ ảnh hưởng đến số lượng parameter của model.
- Huấn luyện mô hình theo character level sẽ có lợi thế hơn khi các từ thay đổi chỉ nằm trong tập các chữ cái `ueoaidy` . Do đó chúng ta sẽ teacher forcing để chỉ thay đổi các từ nằm trong tập `ueoaidy` và không thay đổi các từ còn lại.
- Mô hình mới chỉ xây dựng cho các cụm từ ngram với kích thước bằng 4. Để dự báo cho câu với độ dài tùy ý dựa trên ngram = 4 xin dành cho bạn đọc.
- Lớp model transformer cũng là một trong những mô hình cân nhắc thay thế cho seq2seq trong bài toán này vì tốc độ tính toán nhanh, có thể xử lý trên nhiều GPU. Bạn đọc có thể tham khảo ý tưởng của đội xếp 2nd của cuộc thi thêm dấu tiếng việt.

7. Tài liệu tham khảo

1. hướng dẫn pytorch (https://phamdinhkhanh.github.io/2019/08/10/PytorchTurtorial1.html)
2. lý thuyết về mạng LSTM (https://phamdinhkhanh.github.io/2019/04/22/L%C3%BD_thuy%E1%BA%BFt_v%E1%BB%81_m%E1%BA%A1ng_LSTM.htm
3. seq2seq pytorch (https://pytorch.org/tutorials/beginner/chatbot_tutorial.html)
4. tensor2tensor project (https://github.com/tensorflow/tensor2tensor)
5. attention là tất cả bạn cần (https://phamdinhkhanh.github.io/2019/06/18/AttentionLayer.html)
6. thêm dấu cho tiếng việt - aivivn 1st (https://forum.machinelearningcoban.com/t/aivivn-3-vietnamese-tone-prediction-1st-place-solution/5721)
7. thêm dấu cho tiếng việt - aivivn 2nd (https://forum.machinelearningcoban.com/t/aivivn-3-vietnamese-tone-prediction-2nd-place-solution/5759)

Top