# -\*- coding: utf-8 -\*-

"""data\_loader\_modified.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1JVzcC2bcgIkluafjag5\_R7pYRUWVcENh

"""

# -\*- coding: utf-8 -\*-

"""bert\_encoder\_modified.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1U\_2J9Idv57geXTBXT2lC9LpsB5mHQYzn

"""

import torch

import torch.nn as nn

from transformers import BertModel, BertTokenizer

# from .base\_encoder import BaseEncoder

import pickle as pk

class BERTEncoder(nn.Module):

def \_\_init\_\_(self, max\_length, pretrain\_path, blank\_padding=True):

"""

Args:

max\_length: max length of sentence

pretrain\_path: path of pretrain model

"""

super().\_\_init\_\_()

self.max\_length = max\_length

self.blank\_padding = blank\_padding

self.bert = BertModel.from\_pretrained(pretrain\_path)

self.hidden\_size = self.bert.config.hidden\_size

self.tokenizer = BertTokenizer.from\_pretrained(pretrain\_path)

def forward(self, token, att\_mask):

"""

Args:

token: (B, L), index of tokens

att\_mask: (B, L), attention mask (1 for contents and 0 for padding)

Return:

(B, H), representations for sentences

"""

result = self.bert(token, attention\_mask=att\_mask)

x = result[1]

return x

def tokenize(self, item):

"""

Args:

item: data instance containing 'text' / 'token', 'h' and 't'

Return:

Name of the relation of the sentence

"""

# Sentence -> token

if 'text' in item:

sentence = item['text']

is\_token = False

else:

sentence = item['token']

is\_token = True

pos\_head = item['h']['pos']

pos\_tail = item['t']['pos']

if not is\_token:

pos\_min = pos\_head

pos\_max = pos\_tail

if pos\_head[0] > pos\_tail[0]:

pos\_min = pos\_tail

pos\_max = pos\_head

rev = True

else:

rev = False

sent0 = self.tokenizer.tokenize(sentence[:pos\_min[0]])

ent0 = self.tokenizer.tokenize(sentence[pos\_min[0]:pos\_min[1]])

sent1 = self.tokenizer.tokenize(sentence[pos\_min[1]:pos\_max[0]])

ent1 = self.tokenizer.tokenize(sentence[pos\_max[0]:pos\_max[1]])

sent2 = self.tokenizer.tokenize(sentence[pos\_max[1]:])

pos\_head = [len(sent0), len(sent0) + len(ent0)]

pos\_tail = [

len(sent0) + len(ent0) + len(sent1),

len(sent0) + len(ent0) + len(sent1) + len(ent1)

]

if rev:

pos\_tail = [len(sent0), len(sent0) + len(ent0)]

pos\_head = [

len(sent0) + len(ent0) + len(sent1),

len(sent0) + len(ent0) + len(sent1) + len(ent1)

]

tokens = sent0 + ent0 + sent1 + ent1 + sent2

else:

tokens = sentence

# Token -> index

re\_tokens = ['[CLS]']

cur\_pos = 0

for token in tokens:

token = token.lower()

if cur\_pos == pos\_head[0]:

re\_tokens.append('[unused0]')

if cur\_pos == pos\_tail[0]:

re\_tokens.append('[unused1]')

re\_tokens += self.tokenizer.tokenize(token)

if cur\_pos == pos\_head[1] - 1:

re\_tokens.append('[unused2]')

if cur\_pos == pos\_tail[1] - 1:

re\_tokens.append('[unused3]')

cur\_pos += 1

re\_tokens.append('[SEP]')

indexed\_tokens = self.tokenizer.convert\_tokens\_to\_ids(re\_tokens)

avai\_len = len(indexed\_tokens)

# Padding

if self.blank\_padding:

while len(indexed\_tokens) < self.max\_length:

indexed\_tokens.append(0) # 0 is id for [PAD]

indexed\_tokens = indexed\_tokens[:self.max\_length]

indexed\_tokens = torch.tensor(indexed\_tokens).long().unsqueeze(

0) # (1, L)

# Attention mask

att\_mask = torch.zeros(indexed\_tokens.size()).long() # (1, L)

att\_mask[0, :avai\_len] = 1

return indexed\_tokens, att\_mask

class BERTHiddenStateEncoder(BERTEncoder):

def \_\_init\_\_(self, pretrain\_path, blank\_padding=True):

super().\_\_init\_\_(80, pretrain\_path, blank\_padding)

self.bert = BertModel.from\_pretrained(pretrain\_path, output\_hidden\_states=True,output\_attentions=True)

def forward(self, token, att\_mask):

"""

Args:

token: (B, L), index of tokens

att\_mask: (B, L), attention mask (1 for contents and 0 for padding)

Return:

(B, H), representations for sentences

"""

result = self.bert(token, attention\_mask=att\_mask)

x = result[1]

hs = result[2]

atts = result[3]

return x, hs, atts

def tokenize(self, item):

"""

Args:

item: data instance containing 'text' / 'token', 'h' and 't'

Return:

Name of the relation of the sentence

"""

# Sentence -> token

# if 'text' in item:

# sentence = item['text']

# is\_token = False

# else:

sentence = item['token']

is\_token = True

tokens = sentence

re\_tokens = ['[CLS]']

cur\_pos = 0

new\_index = [-1]

for idx, token in enumerate(tokens):

tokenized = self.tokenizer.tokenize(token)

re\_tokens += tokenized

# If BERT cannot recognize the token, assign the token as "."

if self.tokenizer.tokenize(token) == []:

re\_tokens += "."

num\_new\_tokens = 1

else:

num\_new\_tokens = len(re\_tokens) - len(new\_index)

# print(re\_tokens)

new\_index.extend([idx] \* num\_new\_tokens)

# print(new\_index)

cur\_pos += 1

re\_tokens.append('[SEP]')

new\_index.append(max(new\_index) + 1)

indexed\_tokens = self.tokenizer.convert\_tokens\_to\_ids(re\_tokens)

indexed\_tokens = torch.tensor(indexed\_tokens).long().unsqueeze(0) # (1, L)

# Attention mask

att\_mask = torch.ones(indexed\_tokens.size()).long() # (1, L)

return indexed\_tokens, att\_mask, new\_index

class BERTEntityEncoder(nn.Module):

def \_\_init\_\_(self, max\_length, pretrain\_path, blank\_padding=True):

"""

Args:

max\_length: max length of sentence

pretrain\_path: path of pretrain model

"""

super().\_\_init\_\_()

self.max\_length = max\_length

self.blank\_padding = blank\_padding

self.hidden\_size = 768 \* 2

self.bert = BertModel.from\_pretrained(pretrain\_path)

self.tokenizer = BertTokenizer.from\_pretrained(pretrain\_path)

self.linear = nn.Linear(self.hidden\_size, self.hidden\_size)

def forward(self, token, att\_mask, pos1, pos2):

"""

Args:

token: (B, L), index of tokens

att\_mask: (B, L), attention mask (1 for contents and 0 for padding)

pos1: (B, 1), position of the head entity starter

pos2: (B, 1), position of the tail entity starter

Return:

(B, 2H), representations for sentences

"""

result = self.bert(token, attention\_mask=att\_mask)

hidden = result[0]

hidden = hidden[-1] # the last hidden layer, (B, L, H)

# Get entity start hidden state

onehot = torch.zeros(hidden.size()[:2]).float() # (B, L)

if torch.cuda.is\_available():

onehot = onehot.cuda()

onehot\_head = onehot.scatter\_(1, pos1, 1)

onehot\_tail = onehot.scatter\_(1, pos2, 1)

head\_hidden = (onehot\_head.unsqueeze(2) \* hidden).sum(1) # (B, H)

tail\_hidden = (onehot\_tail.unsqueeze(2) \* hidden).sum(1) # (B, H)

x = torch.cat([head\_hidden, tail\_hidden], 1) # (B, 2H)

x = self.linear(x)

return x

def tokenize(self, item):

"""

Args:

item: data instance containing 'text' / 'token', 'h' and 't'

Return:

Name of the relation of the sentence

"""

# Sentence -> token

if 'text' in item:

sentence = item['text']

is\_token = False

else:

sentence = item['token']

is\_token = True

pos\_head = item['h']['pos']

pos\_tail = item['t']['pos']

if not is\_token:

pos\_min = pos\_head

pos\_max = pos\_tail

if pos\_head[0] > pos\_tail[0]:

pos\_min = pos\_tail

pos\_max = pos\_head

rev = True

else:

rev = False

sent0 = self.tokenizer.tokenize(sentence[:pos\_min[0]])

ent0 = self.tokenizer.tokenize(sentence[pos\_min[0]:pos\_min[1]])

sent1 = self.tokenizer.tokenize(sentence[pos\_min[1]:pos\_max[0]])

ent1 = self.tokenizer.tokenize(sentence[pos\_max[0]:pos\_max[1]])

sent2 = self.tokenizer.tokenize(sentence[pos\_max[1]:])

pos\_head = [len(sent0), len(sent0) + len(ent0)]

pos\_tail = [

len(sent0) + len(ent0) + len(sent1),

len(sent0) + len(ent0) + len(sent1) + len(ent1)

]

if rev:

pos\_tail = [len(sent0), len(sent0) + len(ent0)]

pos\_head = [

len(sent0) + len(ent0) + len(sent1),

len(sent0) + len(ent0) + len(sent1) + len(ent1)

]

tokens = sent0 + ent0 + sent1 + ent1 + sent2

else:

tokens = sentence

# Token -> index

re\_tokens = ['[CLS]']

cur\_pos = 0

pos1 = 0

pos2 = 0

for token in tokens:

token = token.lower()

if cur\_pos == pos\_head[0]:

pos1 = len(re\_tokens)

re\_tokens.append('[unused0]')

if cur\_pos == pos\_tail[0]:

pos2 = len(re\_tokens)

re\_tokens.append('[unused1]')

re\_tokens += self.tokenizer.tokenize(token)

if cur\_pos == pos\_head[1] - 1:

re\_tokens.append('[unused2]')

if cur\_pos == pos\_tail[1] - 1:

re\_tokens.append('[unused3]')

cur\_pos += 1

re\_tokens.append('[SEP]')

pos1 = min(self.max\_length - 1, pos1)

pos2 = min(self.max\_length - 1, pos2)

indexed\_tokens = self.tokenizer.convert\_tokens\_to\_ids(re\_tokens)

avai\_len = len(indexed\_tokens)

# Position

pos1 = torch.tensor([[pos1]]).long()

pos2 = torch.tensor([[pos2]]).long()

# Padding

# if self.blank\_padding:

# while len(indexed\_tokens) < self.max\_length:

# indexed\_tokens.append(0) # 0 is id for [PAD]

# indexed\_tokens = indexed\_tokens[:self.max\_length]

indexed\_tokens = torch.tensor(indexed\_tokens).long().unsqueeze(

0) # (1, L)

# Attention mask

att\_mask = torch.ones(indexed\_tokens.size()).long() # (1, L)

return indexed\_tokens, att\_mask, pos1, pos2