In [1]:

```
import torch as t
import numpy as np
import torchtext
from torchtext.datasets import AG_NEWS
from torchtext.data.utils import get_tokenizer
import collections

!pip install 'portalocker>=2.8.2'
```

```
Requirement already satisfied: portalocker>=2.8.2 in /usr/local/lib/python3.10/dist-packages (2.8.2)
```

1) Torch text datasets

In [2]:

```
''' According to this link, https://pytorch.org/text/stable/datasets.html pretty much every
    single torchtext (tt) dataset (ds) has root and split arguments.

    1) root - ??? Directory where ds are saved ???
    2) split - train or test for ag_news, but for a ds like SST2 it has train, test and DEV. '''
train, test = AG_NEWS()
```

In [3]:

```
''' Iter gives ability to of course loop (technically) over the object. As of now its type
    is "ShardingFilterIterDataPipe". After iter? It's
    "<generator object ShardingFilterIterDataPipe.__iter__ at 0x7d2b405bc660>" '''
x = iter(train)

x
```

Out[3]:

```
<generator object ShardingFilterIterDataPipe.__iter__ at 0x7dd03da399a0>
```

In [4]:

```
''' Just keeps getting the next value, can also use a default value if end is reached.
    https://stackoverflow.com/questions/76302971/question-in-pytorch-transformer-tutorial-about-nonetype-object-has-no-attribut ''
'
next(iter(train))
```

Out[4]:

```
(3,
 "Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street's dwindling\\band of ultra-cynics, are se
eing green again.")
```

In [5]:

```
# Zip is good for a certain range, 2, 5, etc. n in general. Very useful.
# for i, x in zip(range(2), train):
#   print(f'Index {i}. x:\n{x}\n\n')

# Enumerate works fine as well.
for i, x in enumerate(train):
  print(f'Index {i}:\nx: {x}')
  break
```

```
Index 0:
x: (3, "Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street's dwindling\\band of ultra-cynics,
are seeing green again.")
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/datapipes/iter/combining.py:333: UserWarning: Some child DataPipes are no
t exhausted when __iter__ is called. We are resetting the buffer and each child DataPipe will read from the start again.
  warnings.warn("Some child DataPipes are not exhausted when __iter__ is called. We are resetting "
```

2) Tokenization

In [6]:

```
# Get list of sentences
s = []
for i, x in zip(range(3), train):
  s.append(x[1])

print(f'List of sentences to tokenize:\n\n{s}\nTotal sentences: {len(s)}')
```

```
List of sentences to tokenize:

["Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street's dwindling\\band of ultra-cynics, are se
eing green again.", 'Carlyle Looks Toward Commercial Aerospace (Reuters) Reuters - Private investment firm Carlyle Group,\\which h
as a reputation for making well-timed and occasionally\\controversial plays in the defense industry, has quietly placed\\its bets
on another part of the market.', "Oil and Economy Cloud Stocks' Outlook (Reuters) Reuters - Soaring crude prices plus worries\\abou
t the economy and the outlook for earnings are expected to\\hang over the stock market next week during the depth of the\\summer do
ldrums."]
Total sentences: 3
```

In [7]:

```
tk = get_tokenizer('basic_english')

# ts = tokenized sentences. !!! do "tk(s[0])" for individual sentences !!!
ts = [tk(sent) for sent in s]
```

```
ts
```

Out[7]:

```
[['wall',
  'st',
  '.',
  'bears',
  'claw',
  'back',
  'into',
  'the',
  'black',
  '(',
  'reuters',
  ')',
  'reuters',
  '-',
  'short-sellers',
  ',',
  'wall',
  'street',
  "'",
  's',
  'dwindling\\band',
  'of',
  'ultra-cynics',
  ',',
  'are',
  'seeing',
  'green',
  'again',
  '.'],
 ['carlyle',
  'looks',
  'toward',
  'commercial',
  'aerospace',
  '(',
  'reuters',
  ')',
  'reuters',
  '-',
  'private',
  'investment',
  'firm',
  'carlyle',
  'group',
  ',',
  '\\which',
  'has',
  'a',
  'reputation',
  'for',
  'making',
  'well-timed',
  'and',
  'occasionally\\controversial',
  'plays',
  'in',
  'the',
  'defense',
  'industry',
  ',',
  'has',
  'quietly',
  'placed\\its',
  'bets',
  'on',
  'another',
  'part',
  'of',
  'the',
  'market',
  '.'],
 ['oil',
  'and',
  'economy',
  'cloud',
  'stocks',
  "'",
  'outlook',
  '(',
  'reuters',
  ')',
  'reuters',
  '-',
  'soaring',
  'crude',
  'prices',
  'plus',
  'worries\\about',
  'the',
  'economy',
  'and',
  'the',
  'outlook',
  'for',
  'earnings',
  'are',
  'expected',
  'to\\hang',
```

```
  'over',
  'the',
  'stock',
  'market',
  'next',
  'week',
  'during',
  'the',
  'depth',
  'of',
  'the\\summer',
  'doldrums',
  '.']]
```

In [8]:

```
# counter just counts how many times its seen something.

# 1) Give first tokenized str
# counter = collections.Counter(ts[0])
# counter

# 2) Init default and update it with tokenized str
# counter = collections.Counter()
# counter.update(ts[0])
# counter

# 3) Use whole ds with it (part of it here)
# counter = collections.Counter()
# for i, x in zip(range(3), train):
#   counter.update(tk(x[1]))
# counter

# 4) The 3rd option used train directly, can also use the tokenized sentences in the list "ts".
counter = collections.Counter()
for sent in ts:
  counter.update(sent)
counter
```

Out[8]:

```
Counter({'wall': 2,
         'st': 1,
         '.': 4,
         'bears': 1,
         'claw': 1,
         'back': 1,
         'into': 1,
         'the': 7,
         'black': 1,
         '(': 3,
         'reuters': 6,
         ')': 3,
         '-': 3,
         'short-sellers': 1,
         ',': 4,
         'street': 1,
         "'": 2,
         's': 1,
         'dwindling\\band': 1,
         'of': 3,
         'ultra-cynics': 1,
         'are': 2,
         'seeing': 1,
         'green': 1,
         'again': 1,
         'carlyle': 2,
         'looks': 1,
         'toward': 1,
         'commercial': 1,
         'aerospace': 1,
         'private': 1,
         'investment': 1,
         'firm': 1,
         'group': 1,
         '\\which': 1,
         'has': 2,
         'a': 1,
         'reputation': 1,
         'for': 2,
         'making': 1,
         'well-timed': 1,
         'and': 3,
         'occasionally\\controversial': 1,
         'plays': 1,
         'in': 1,
         'defense': 1,
         'industry': 1,
         'quietly': 1,
         'placed\\its': 1,
         'bets': 1,
         'on': 1,
         'another': 1,
         'part': 1,
         'market': 2,
         'oil': 1,
         'economy': 2,
         'cloud': 1,
         'stocks': 1,
         'outlook': 2,
         'soaring': 1,
         'crude': 1,
         'prices': 1,
```

```
 'prices': 1,
 'plus': 1,
 'worries\\about': 1,
 'earnings': 1,
 'expected': 1,
 'to\\hang': 1,
 'over': 1,
 'stock': 1,
 'next': 1,
 'week': 1,
 'during': 1,
 'depth': 1,
 'the\\summer': 1,
 'doldrums': 1})
```

In [9]:

```
# vocab is a dictionary
v = torchtext.vocab.vocab(counter, min_freq=1)

print(v)
print(v.vocab)
print(len(v.vocab))
```

```
Vocab()
<torchtext._torchtext.Vocab object at 0x7dd0387828f0>
75
```

In [10]:

```
# Returns dictionary of word/assigned number in key/value pairs.
v.get_stoi()
```

Out[10]:

```
{'depth': 72,
 'during': 71,
 'next': 69,
 'over': 67,
 'to\\hang': 66,
 'earnings': 64,
 'worries\\about': 63,
 'prices': 61,
 'crude': 60,
 'dwindling\\band': 18,
 'green': 23,
 'group': 33,
 'has': 35,
 'private': 30,
 'of': 19,
 'carlyle': 25,
 'looks': 26,
 ',': 14,
 'street': 15,
 'plus': 62,
 'black': 8,
 'toward': 27,
 'bears': 3,
 'the\\summer': 73,
 'week': 70,
 'reuters': 10,
 'again': 24,
 ')': 11,
 '.': 2,
 '\\which': 34,
 'st': 1,
 'market': 53,
 'the': 7,
 'in': 44,
 'expected': 65,
 '(': 9,
 'are': 21,
 'part': 52,
 "'": 16,
 'claw': 4,
 'into': 6,
 'short-sellers': 13,
 '-': 12,
 'aerospace': 29,
 'commercial': 28,
 'investment': 31,
 'firm': 32,
 'seeing': 22,
 'making': 39,
 'for': 38,
 'another': 51,
 'cloud': 56,
 'back': 5,
 'outlook': 58,
 'a': 36,
 'on': 50,
 'reputation': 37,
 's': 17,
 'well-timed': 40,
 'and': 41,
 'wall': 0,
 'industry': 46,
 'soaring': 59,
 'stocks': 57,
 'occasionally\\controversial': 42,
 'stock': 68,
 'bets': 49,
 'doldrums': 74,
```

```
 'ultra-cynics': 20,
 'plays': 43,
 'defense': 45,
 'quietly': 47,
 'placed\\its': 48,
 'oil': 54,
 'economy': 55}
```

In [11]:

```
#  Returns list of all words. showing first 10
v.get_itos()[:10]
```

Out[11]:

```
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(']
```

In [12]:

```
# zip is cool.
for i, x in zip(range(5), v.get_itos()):
  print(f'Numerical value in dictionary: {i}. Assigned word: {x}')
```

```
Numerical value in dictionary: 0. Assigned word: wall
Numerical value in dictionary: 1. Assigned word: st
Numerical value in dictionary: 2. Assigned word: .
Numerical value in dictionary: 3. Assigned word: bears
Numerical value in dictionary: 4. Assigned word: claw
```

In [13]:

```
''' Convert sentence into nums. es = example sentence. Get the tokenized sentence.
    Loop through it, get_stoi() remember returns dictionary of str/num key/value pairs.
    So give it str (or token) to get num back. '''
es = ts[0]

numerical_sentence = [v.get_stoi()[cur_token] for cur_token in es]

print(f'Example sentence:\n{es}\n\nNumerical version of same sentence:\n{numerical_sentence}')
```

```
Example sentence:
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(', 'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wa
ll', 'street', "'", 's', 'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing', 'green', 'again', '.']

Numerical version of same sentence:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, 13, 14, 0, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24, 2]
```

In [14]:

```
# Create lists with tokens and their numerical conversions.
pairs = []
for cur_sent in ts:
  pairs.append([list((v.get_stoi()[cur_token], cur_token)) for cur_token in cur_sent])
pairs
```

Out[14]:

```
[[[0, 'wall'],
  [1, 'st'],
  [2, '.'],
  [3, 'bears'],
  [4, 'claw'],
  [5, 'back'],
  [6, 'into'],
  [7, 'the'],
  [8, 'black'],
  [9, '('],
  [10, 'reuters'],
  [11, ')'],
  [10, 'reuters'],
  [12, '-'],
  [13, 'short-sellers'],
  [14, ','],
  [0, 'wall'],
  [15, 'street'],
  [16, "'"],
  [17, 's'],
  [18, 'dwindling\\band'],
  [19, 'of'],
  [20, 'ultra-cynics'],
  [14, ','],
  [21, 'are'],
  [22, 'seeing'],
  [23, 'green'],
  [24, 'again'],
  [2, '.']],
 [[25, 'carlyle'],
  [26, 'looks'],
  [27, 'toward'],
  [28, 'commercial'],
  [29, 'aerospace'],
  [9, '('],
  [10, 'reuters'],
  [11, ')'],
  [10, 'reuters'],
  [12, '-'],
  [30, 'private'],
  [31, 'investment'],
  [32, 'firm'],
  [25, 'carlyle'],
  [33, 'group'],
  [14, ','],
  [34, '\\which'],
```

```
  [35, 'has'],
  [36, 'a'],
  [37, 'reputation'],
  [38, 'for'],
  [39, 'making'],
  [40, 'well-timed'],
  [41, 'and'],
  [42, 'occasionally\\controversial'],
  [43, 'plays'],
  [44, 'in'],
  [7, 'the'],
  [45, 'defense'],
  [46, 'industry'],
  [14, ','],
  [35, 'has'],
  [47, 'quietly'],
  [48, 'placed\\its'],
  [49, 'bets'],
  [50, 'on'],
  [51, 'another'],
  [52, 'part'],
  [19, 'of'],
  [7, 'the'],
  [53, 'market'],
  [2, '.']],
 [[54, 'oil'],
  [41, 'and'],
  [55, 'economy'],
  [56, 'cloud'],
  [57, 'stocks'],
  [16, "'"],
  [58, 'outlook'],
  [9, '('],
  [10, 'reuters'],
  [11, ')'],
  [10, 'reuters'],
  [12, '-'],
  [59, 'soaring'],
  [60, 'crude'],
  [61, 'prices'],
  [62, 'plus'],
  [63, 'worries\\about'],
  [7, 'the'],
  [55, 'economy'],
  [41, 'and'],
  [7, 'the'],
  [58, 'outlook'],
  [38, 'for'],
  [64, 'earnings'],
  [21, 'are'],
  [65, 'expected'],
  [66, 'to\\hang'],
  [67, 'over'],
  [7, 'the'],
  [68, 'stock'],
  [53, 'market'],
  [69, 'next'],
  [70, 'week'],
  [71, 'during'],
  [7, 'the'],
  [72, 'depth'],
  [19, 'of'],
  [73, 'the\\summer'],
  [74, 'doldrums'],
  [2, '.']]]
```

In [15]:

```
v.get_itos()[21]
```

Out[15]:

```
'are'
```

In [16]:

```
def encode(str_to_encode):
  return [v.get_stoi()[token] for token in str_to_encode]

''' get_itos works for decoding because the vocab object already assigned numbers
    to words. So as of right now, v.get_stoi()[0] gets the word in the dictionary
    with assigned num 0. '''
def decode(nums_to_decode):
  return [v.get_itos()[num] for num in nums_to_decode]

# es = example str declared in earlier cell. already tokenized.
encoded_str = encode(es)
print(f'Example str:\n{es}\n\nEncoded str:\n{encoded_str}\n\n')

decoded_str = decode(encoded_str)
print(f'Decoded str:\n{decoded_str}')
```

```
Example str:
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(', 'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wa
ll', 'street', "'", 's', 'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing', 'green', 'again', '.']

Encoded str:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, 13, 14, 0, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24, 2]


Decoded str:
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(', 'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wa
ll', 'street', "'", 's', 'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing', 'green', 'again', '.']
```

3) N grams

In [17]:

```
from torchtext.data.utils import ngrams_iterator

''' Ngrams will help solve multiword expression issues like "hamburger" because "ham" and
    "burger" can be 2 separate words. Can't always represent both of those words with the same
    vector. The update func gets pairs because ngrams=2. If the goal was to be able to understand
    3 sets of words, ngrams will be 3. Downside is this WILL example the counter object
    by a lot. Specifically len(v) * 2/3/etc. '''

# nc = ngrams counter
nc = collections.Counter()
nc.update(ngrams_iterator(es, ngrams=2))
nc
```

Out[17]:

```
Counter({'wall': 2,
         'st': 1,
         '.': 2,
         'bears': 1,
         'claw': 1,
         'back': 1,
         'into': 1,
         'the': 1,
         'black': 1,
         '(': 1,
         'reuters': 2,
         ')': 1,
         '-': 1,
         'short-sellers': 1,
         ',': 2,
         'street': 1,
         "'": 1,
         's': 1,
         'dwindling\\band': 1,
         'of': 1,
         'ultra-cynics': 1,
         'are': 1,
         'seeing': 1,
         'green': 1,
         'again': 1,
         'wall st': 1,
         'st .': 1,
         '. bears': 1,
         'bears claw': 1,
         'claw back': 1,
         'back into': 1,
         'into the': 1,
         'the black': 1,
         'black (': 1,
         '( reuters': 1,
         'reuters )': 1,
         ') reuters': 1,
         'reuters -': 1,
         '- short-sellers': 1,
         'short-sellers ,': 1,
         ', wall': 1,
         'wall street': 1,
         "street '": 1,
         "' s": 1,
         's dwindling\\band': 1,
         'dwindling\\band of': 1,
         'of ultra-cynics': 1,
         'ultra-cynics ,': 1,
         ', are': 1,
         'are seeing': 1,
         'seeing green': 1,
         'green again': 1,
         'again .': 1})
```

In [18]:

```
for i, a in zip(range(3), nc):
  print(a)
```

```
wall
st
.
```

4) Bag of words (bow) & more

In [19]:

```
# A bow is simply seeing how many times a word occurs. Get the first 3 sentences num representations here
num_sents = [encode(tk(a[1])) for i, a in zip(range(3), train)]

for x in num_sents:
  print(x)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, 13, 14, 0, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24, 2]
[25, 26, 27, 28, 29, 9, 10, 11, 10, 12, 30, 31, 32, 25, 33, 14, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 7, 45, 46, 14, 35, 47,
48, 49, 50, 51, 52, 19, 7, 53, 2]
[54, 41, 55, 56, 57, 16, 58, 9, 10, 11, 10, 12, 59, 60, 61, 62, 63, 7, 55, 41, 7, 58, 38, 64, 21, 65, 66, 67, 7, 68, 53, 69, 70, 71
, 7, 72, 19, 73, 74, 2]
```

In [20]:

```
# Get tensor of length of vocab. With first 3 sentences in ag_news USED for vocab, this will be 75.
a = t.zeros(len(v))

# Loop over first num sentence seen in previous cell.
for num in num_sents[0]:
  ''' Ex: v.get_stoi() has "wall" paired with num 0. this will increase index 0 by 1. Keep in mind,
      we can access index 0 in this "a" tensor, and give that index to v.get_stoi()[] and it'll
      return the proper word '''
  a[num] = a[num] + 1

a
```

Out[20]:

```
tensor([2., 1., 2., 1., 1., 1., 1., 1., 1., 1., 2., 1., 1., 1., 2., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0.])
```

In [21]:

```
''' Want to combine tensors? Stack helps. MUST be same dimensions. dim=1 gives vertical
    alignment, dim=0 gives horizontal.  '''
first_tensor = t.tensor([1,2,3])
second_tensor = t.tensor([5,5,5])

v_tensor = t.stack([first_tensor, second_tensor],dim=1)
h_tensor = t.stack([first_tensor, second_tensor],dim=0)

print(f'Vertical tensor:\n{v_tensor}\n\nHorizontal tensor:\n{h_tensor}\n')
print(f'Vertical tensor shape: {v_tensor.shape}\nHorizontal tensor shape: {h_tensor.shape}')
```

```
Vertical tensor:
tensor([[1, 5],
        [2, 5],
        [3, 5]])

Horizontal tensor:
tensor([[1, 2, 3],
        [5, 5, 5]])

Vertical tensor shape: torch.Size([3, 2])
Horizontal tensor shape: torch.Size([2, 3])
```

In [22]:

```
''' num_sents aren't the same size which is required for models like embedding, so padding will be
    demonstrated. map wil take the len func and apply it to every num sentence and list will get
    all the lengths. Then max just gets the biggest one. Quick and easy. '''
max_length = max(list(map(len, num_sents)))

''' Convert sent to tensor first, and 0, max length - len of current sentence just makes sure we get
    the CORRECT amount of 0s. '''
padded_num_sents = [t.nn.functional.pad(t.tensor(sent), (0, max_length - len(sent))) for sent in num_sents]

print(padded_num_sents)

for x in padded_num_sents:
  print(len(x))
```

```
[tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
        16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0]), tensor([25, 26, 27, 28, 29,  9, 10, 11, 10, 12, 30, 31, 32, 25, 33, 14, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 44,  7, 45, 46, 14, 35, 47, 48, 49, 50,
        51, 52, 19,  7, 53,  2]), tensor([54, 41, 55, 56, 57, 16, 58,  9, 10, 11, 10, 12, 59, 60, 61, 62, 63,  7,
        55, 41,  7, 58, 38, 64, 21, 65, 66, 67,  7, 68, 53, 69, 70, 71,  7, 72,
        19, 73, 74,  2,  0,  0])]
42
42
42
```

5) Dataset & Data Loader

In [23]:

```
from torch.utils.data import Dataset, DataLoader

''' Datasets are good because they can potentially load data only when necessary instead of all at once
    like with image processing tasks. Also a way to keep things organized. Must override len and getitem. '''
class TestDataset(Dataset):
  def __init__(self, data_to_use):
    self.data = data_to_use

  def __getitem__(self, index):
    return self.data[index]

  def __len__(self):
    return len(self.data)

# The collate_fn arg in DataLoader which extra things to the dataset before it's saved in data loader.
def sub(num):
  return t.tensor([cur_tensor_num.item() - 1 for cur_tensor_num in num])

# Get some generic data. "tensor([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])"
x = t.tensor([x * 2 for x in range(10)])
tds = TestDataset(x)
```

```
''' tdl = test data loader (dl) Very efficient way of loading data. Also when using the collate_fn arg, it
    seems like the DataLoader GIVES the data to whatever func as a list. Strange.
    Uncomment/comment whichever below DataLoader to see results.  '''
# tdl = DataLoader(tds, 1, shuffle=False, collate_fn=sub)
tdl = DataLoader(tds, 1, shuffle=False)

for x in tdl:
  print(x)
```

```
tensor([0])
tensor([2])
tensor([4])
tensor([6])
tensor([8])
tensor([10])
tensor([12])
tensor([14])
tensor([16])
tensor([18])
```

6) Basic layer, activation function, etc

In [24]:

```
# Seeing inputs of a linear layer and what it'll output.
t.nn.Linear(len(v), 3)
```

Out[24]:

```
Linear(in_features=75, out_features=3, bias=True)
```

In [25]:

```
x = t.tensor([[2,8,9],
              [5,1,3],
              [4,8,7]], dtype=t.float64)

''' ??? 1 is likely vertical (row by row) and 0 horizontal (column by column). ??? Softmax
    converts things to probabilities, so log softmax does the same but with log applied.
    Also logsoftmax is for classification
    https://www.baeldung.com/cs/softmax-vs-log-softmax '''
ls = t.nn.LogSoftmax(dim=1)

print(x)
print(ls(x))
```

```
tensor([[2., 8., 9.],
        [5., 1., 3.],
        [4., 8., 7.]], dtype=torch.float64)
tensor([[-7.3139, -1.3139, -0.3139],
        [-0.1429, -4.1429, -2.1429],
        [-4.3266, -0.3266, -1.3266]], dtype=torch.float64)
```

In [26]:

```
# Sequential is just layers in an order.
network = t.nn.Sequential(t.nn.Linear(len(v), 3),
                          t.nn.LogSoftmax(dim=1))

network
```

Out[26]:

```
Sequential(
  (0): Linear(in_features=75, out_features=3, bias=True)
  (1): LogSoftmax(dim=1)
)
```

In [27]:

```
''' len(v) because model input is equal to vocab length.
    1) "TypeError: linear(): argument 'input' (position 1) must be Tensor, not list"
          I converted it to a tensor because of this.

    2) "RuntimeError: mat1 and mat2 must have the same dtype, but got Long and Float"
          I checked the input_test dtype and it was torch.int64. Apparently that
          qualifies as a long? A QUICK fix would be to change the input_test tensor
          to dtype=t.float32, but I also wanted to check models input.

          2.2) Checking model input.
                Help from link:
                https://discuss.pytorch.org/t/get-appropriate-model-in-output-type-programmatically/53742

                Doing:
                "z = t.nn.Linear(len(v), 3)
                list(z.parameters())" returns a list of tensors. First tensor is 2d
                because it has 3 inner tensors of 75 values, the vocab size.

                Ex:
                "tensor([[-0.0353, -0.0916,  0.0258,  0.0546, -0.0598,  0.0273, -0.0447, -0.0129,
                           0.0199, -0.0067,  0.0418,  0.1102, -0.0363, -0.0615, -0.1136,  0.0627],

                         [0.0922, -0.0013, -0.0645, -0.0640, -0.0426,  0.0972,  0.0951, -0.0859,
                           0.0891,  0.1113, -0.0431, -0.0607,  0.0699, -0.0706,  0.0240, -0.0716],

                         [-0.0369,  0.0732,  0.0284, -0.0249,  0.0937, -0.0938,  0.0555,  0.0908,
                          -0.0923, -0.0790,  0.0530,  0.0607,  0.1147,  0.0963, -0.0195, -0.1021]]"

                It's in 2d. Strange. I guess input must be in 2d as well.
```

```
        But to the point of this section, which is 2, the error can be fixed by changing the
        type of the input to dtype=t.float32.


    3) "IndexError: Dimension out of range (expected to be in range of [-1, 0], but got 1)"
        Then this one came up because I BELIEVE the input tensor was not 2d. I used
        "t.unsqueeze(input_test, dim=0)" which ADDS a dimension horizontally so now its definitely
        2d. Shape is "torch.Size([1, 75])"

     '''
input_test = t.tensor([i * 2 for i in range(len(v))], dtype=t.float32)
# uit = updated input test
uit = t.unsqueeze(input_test, dim=0)
print(f'Input for model:\n{input_test}\n\nType: {input_test.dtype}\n\nNEW input for model:\n{uit}\n\n')

# Show model result.
network(uit)
```

```
Input for model:
tensor([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.,  20.,  22.,
         24.,  26.,  28.,  30.,  32.,  34.,  36.,  38.,  40.,  42.,  44.,  46.,
         48.,  50.,  52.,  54.,  56.,  58.,  60.,  62.,  64.,  66.,  68.,  70.,
         72.,  74.,  76.,  78.,  80.,  82.,  84.,  86.,  88.,  90.,  92.,  94.,
         96.,  98., 100., 102., 104., 106., 108., 110., 112., 114., 116., 118.,
        120., 122., 124., 126., 128., 130., 132., 134., 136., 138., 140., 142.,
        144., 146., 148.])

Type: torch.float32

NEW input for model:
tensor([[  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.,  20.,  22.,
          24.,  26.,  28.,  30.,  32.,  34.,  36.,  38.,  40.,  42.,  44.,  46.,
          48.,  50.,  52.,  54.,  56.,  58.,  60.,  62.,  64.,  66.,  68.,  70.,
          72.,  74.,  76.,  78.,  80.,  82.,  84.,  86.,  88.,  90.,  92.,  94.,
          96.,  98., 100., 102., 104., 106., 108., 110., 112., 114., 116., 118.,
         120., 122., 124., 126., 128., 130., 132., 134., 136., 138., 140., 142.,
         144., 146., 148.]])
```

Out[27]:

```
tensor([[  0.0000, -26.0792, -37.4430]], grad_fn=<LogSoftmaxBackward0>)
```

In [28]:

```
''' Most basic model ever. Embedding takes in v size and outputs dimension size of whatever requested.
    And that'll be input to the Linear layer of course.

    Errors:
    1) "RuntimeError: Expected tensor for argument #1 'indices' to have one of the following scalar
        types: Long, Int; but got torch.FloatTensor instead (while checking arguments for embedding)"

        Solution: "uit = uit.type(t.long)" type conversion


    2) "IndexError: index out of range in self"

        Understanding the error: https://rollbar.com/blog/how-to-handle-index-out-of-range-in-self-pytorch/#
        The code at the beginning is simple and I made my own test code
        "r = t.nn.Embedding(10, 5)
        m = t.tensor([9])
        r(m)"
        The thing is, if the tensor value is 10, it breaks and gives same error. But if its 9 or less, it
        works fine. Which tells me that the MAX range it'll accept is 10. Anything greater and it breaks.
        Of course it indexes from 0. The above is just for a 1d dimensional tensor.

        Solution: I was given numerical input to the model when the VOCAB didn't match it. The vocab
          went to num 75. Yet I tried giving the model:
        "tensor([[  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.,  20.,  22.,
                   24.,  26.,  28.,  30.,  32.,  34.,  36.,  38.,  40.,  42.,  44.,  46.,
                   48.,  50.,  52.,  54.,  56.,  58.,  60.,  62.,  64.,  66.,  68.,  70.,
                   72.,  74.,  76.,  78.,  80.,  82.,  84.,  86.,  88.,  90.,  92.,  94.,
                   96.,  98., 100., 102., 104., 106., 108., 110., 112., 114., 116., 118.,
                   120., 122., 124., 126., 128., 130., 132., 134., 136., 138., 140., 142.,
                   144., 146., 148.]])"
        The aforementioned link helped me understand this with t.all(). Initially I thought
        t.nn.Embedding(vocab size, etc) meant the LENGTH of the input tensor couldn't be bigger
        than vocab size, but it was checking for INDIVIDUAL numbers instead. Example code below:

        " r = t.nn.Embedding(len(v), 5) # vocab len 75
          m = t.tensor([9,4,2,4,1,4,5,8,3,6,5,2,3,2,4,6,2,4,6,5,5,5,55,31,2]) # Will pass!
          # m = t.tensor([9,4,2,4,1,4,5,8,74,10,75]) # Will fail!
          # m = t.tensor([9,4,2,4,1,4,5,8,74,10,74]) # Will pass!

          # All literally checks ALL values in tensor.
          if t.all(m >= 0):
            print(f'Passed. Tensor is: {m.shape}')
          if t.all(m < r.num_embeddings):
            print(f'Passed. Tensor is: {m.shape}')
          else:
            print(f'!!! Failed. Tensor shape: {uit.shape} and num embeddings: {embedding.num_embeddings} !!!')"



    3) "RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x29 and 5x3)"
        Why did this happen? 1x29 is the size of the num input tenor which is:
        "tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
                 16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2])"
         Real tokenized sentence of above numerical sentence is:
            "['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(',
               'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wall', 'street', "'", 's',
```

```
        'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing', 'green', 'again', '.']"


Solution 1: I commented out:
  "# x = t.mean(x,dim=1)
   # print(f'x shape after t.mean: {x.shape}\nx after t.mean:\n{x}')"
   And it worked perfectly. Why?

   Answer: Because if input is:
     "tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
              16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2])", that's length 29. If
              embedding dimension arg for model (which is the output of the embedding
              layer and input to the linear layer) is say, 5, then the embedding
              layer will output a 2d matrix of size 29,5 because there's 29 values in
              the original tensor so theres a nested tensor for each one. Remember the
              embedding dimension is INPUT to the linear layer, it has the same value
              of 5. Since the embedding layer output shape is 29,5 , it works.


  Solution 2: I used t.mean(x,dim=0) which get the mean in a horizontal fashion. Why did
    it work? BEFORE matrix x goes into the embedding layer, the tensor is:
    "tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
             16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2])" More importantly the shape/size
    is 29. AFTER the matrix x goes through the embedding layer, it's shape/size is (29,5).
    It's now 2d. Why? Well because the output of the embedding layer (called embed_dim)
    is 5. So it gets 29 matrices of length 5 each. Then the linear layers INPUT is embed_dim
    as well which is of course 5. So the linear layer can take a 2d matrix as long as the
    columns of the 2d matrix (the "5" in 29,5) matches the linear layers INPUT (input is
    also 5).
    Ex Code:
    "td = t.tensor([[1,4,2,8],
                    [6,9,3,7]], dtype=t.float32)
      x = t.nn.Linear(4, 1)
      x(td)"

    Will print something like:
    "tensor([[2.5558],
             [4.6453]], grad_fn=<AddmmBackward0>)"

    Ex code 2 (1d):
    " # 1d linear layer test. Tensor is "tensor([5., 5., 5.])", shape is torch.Size([3])
      e = t.tensor([5] * 3, dtype=t.float32)
      x = t.nn.Linear(3, 1)
      # Will print something like "tensor([0.6625], grad_fn=<ViewBackward0>)" is linear output is 1.
      print(x(e))"



Conclusion: Solving the issue WITHOUT messing with t.mean(x, dim=1) is impossible. Simply
  because of the dimensions of both x (after it goes through t.mean) and the input of the
  linear layer.
  1) Dimension of x after t.mean(x, dim=1) - torch.Size([29])
  2) Dimensions of linear layer - 5,3. 5 input, 3 output.

  I initially THOUGHT the REAL size was 30, due to programming indexing from 0. If that
  was the case I had an idea to resize the x matrix into shape (6,5) because that'll be
  accepted by the linear linear and it'd work. As a matter of fact I wrote some test
  code for just that issue, resizing a tensor that was initially 30 in length but the
  wrong input size for a linear layer that had input as 5. See code below:
  "
    # Get same values.
    t.manual_seed(0)

    # nt = new tensor. Make tensor of size 30, (0-29)
    nt = t.rand(30)
    print(f'New tensor:\n{nt}\nNew tensor SIZE: {nt.shape}\n\n')

    # Create a linear layer which only takes 1d tensors of 5, and 2d tensors of (n, 5).
    ll = t.nn.Linear(5, 1)

    # Try reshaping the new tensor to be in form (n, 5). First get size and check if its divisible by embed_dim (ed) 5
    ed = 5
    cur_mat_size = nt.size()[0]

    # Can 30 be divided by 5? If so, we can make a new tensor evenly.
    if cur_mat_size % ed == 0:
      rows_for_reshape = int(cur_mat_size / ed)

      # nm = new matrix.
      nm = t.reshape(nt, (rows_for_reshape, ed))
      print(f'New reshaped matrix is:\n{nm}\nNew reshaped matrix shape: {nm.shape}\n\n')

      print(ll(nm))
  "

  Feel free to copy and past in a different cell. The point of the code was to demonstrate how
  it would be POSSIBLE to reshape a tensor so it can be passed to a linear layer, which I initially
  thought was possible with the tensor([ 0,  1,  2,  3,  4,  5,  6,  7, etc]) input tensor used
  below. But I remembered it was size 29 and not 30. '''

class EmbedClassifier(t.nn.Module):
  def __init__(self, vocab_size, embed_dim, num_class):
    super().__init__()
    self.embedding = t.nn.Embedding(num_embeddings=vocab_size, embedding_dim=embed_dim)
    self.fc = t.nn.Linear(embed_dim, num_class)

  def forward(self, x):
    print(f'x shape: {x.shape}\nx BEFORE embedding:\n{x}\n\n')
    x = self.embedding(x)
    print(f'x shape: {x.shape}\nx AFTER embedding is: {x}\n\n')
    x = t.mean(x,dim=0)
    print(f'x shape after t.mean: {x.shape}\nx after t.mean:\n{x}\n\n')
```

```
    return self.fc(x)

ec = EmbedClassifier(len(v), 5, 3)

''' Tokenized text is:
    "['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(',
      'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wall', 'street',
      "'", 's', 'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing',
      'green', 'again', '.']

    Must get appropriate input for the model. Length is 29. Actual tensor is:
    "tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
             16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2])"  '''
t_test = t.tensor([v.get_stoi()[cur_token] for cur_token in es], dtype=t.long)

x = ec(t_test)
print(f'Returned tensor x:\n{x}\nReturned tensor x shape: {x.shape}')
```

```
x shape: torch.Size([29])
x BEFORE embedding:
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
        16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2])


x shape: torch.Size([29, 5])
x AFTER embedding is: tensor([[-1.1908e+00, -2.1081e-01, -2.9817e-01, -3.5324e-01,  4.2111e-01],
        [-2.9621e-01, -1.1288e+00, -1.8752e+00, -1.2576e+00, -9.7593e-01],
        [-3.3415e-01, -6.8127e-01,  1.0206e+00,  9.0301e-01, -6.6421e-01],
        [ 1.1420e+00,  1.3447e+00,  1.7629e+00, -9.2219e-01,  1.0127e+00],
        [ 2.5227e-01, -3.3943e-01, -1.5154e-01, -5.7956e-01, -6.1363e-01],
        [ 8.0078e-01, -8.4058e-03, -2.6809e-01,  3.6670e-01, -8.7147e-01],
        [ 3.5158e-01,  9.7037e-01, -6.1841e-01, -1.7150e+00, -9.4477e-01],
        [ 5.6200e-02, -1.3585e+00, -1.5107e+00,  1.8437e-01, -1.1318e+00],
        [-4.4572e-01,  1.7038e-01,  9.6803e-01, -4.9766e-01,  2.4211e+00],
        [ 8.4083e-02,  1.3674e-01, -1.1200e-01, -1.2794e+00,  4.8112e-01],
        [-3.2511e-01,  1.2637e+00,  3.0319e+00, -1.7106e-01, -1.5574e+00],
        [-1.3395e-01, -1.2261e-02,  5.7549e-01, -4.3395e-01, -1.7950e-01],
        [-3.2511e-01,  1.2637e+00,  3.0319e+00, -1.7106e-01, -1.5574e+00],
        [ 3.9644e-01,  3.0643e-02, -3.0668e-01, -8.6101e-01, -9.6468e-01],
        [-5.4013e-01, -8.8454e-01,  5.9310e-01,  4.7474e-01, -8.8687e-01],
        [-7.1413e-01,  9.3311e-01, -1.5028e+00,  2.2423e-01,  1.3635e+00],
        [-1.1908e+00, -2.1081e-01, -2.9817e-01, -3.5324e-01,  4.2111e-01],
        [-1.7817e+00, -1.3880e+00, -5.0155e-01, -4.2099e-01,  5.2547e-01],
        [ 9.2339e-01,  1.5365e+00, -1.2359e+00, -9.8602e-01,  5.3560e-01],
        [-9.5344e-01,  1.5216e+00, -2.5985e-03,  3.0238e-01,  8.4563e-01],
        [-1.1308e+00, -5.9252e-01, -1.0334e+00,  1.0114e+00,  4.9759e-01],
        [-1.1516e-02, -8.5977e-01,  5.9038e-02, -9.7705e-01,  1.1238e+00],
        [ 3.5628e-01, -3.6851e-01, -6.1963e-01, -4.2227e-01, -2.8116e+00],
        [-7.1413e-01,  9.3311e-01, -1.5028e+00,  2.2423e-01,  1.3635e+00],
        [-3.0770e-01, -4.8818e-01,  1.5327e+00, -1.5249e+00, -4.9166e-01],
        [-2.8156e-01, -2.0995e-01, -2.3335e-01, -1.1894e+00, -3.4772e-01],
        [-1.9881e-01, -5.2293e-01, -3.6105e-01, -4.3697e-01, -8.2985e-02],
        [-2.2953e-01, -1.4091e+00,  1.4600e+00,  1.7599e+00, -8.4887e-01],
        [-3.3415e-01, -6.8127e-01,  1.0206e+00,  9.0301e-01, -6.6421e-01]],
       grad_fn=<EmbeddingBackward0>)


x shape after t.mean: torch.Size([5])
x after t.mean:
tensor([-0.2440, -0.0431,  0.0905, -0.2827, -0.1580], grad_fn=<MeanBackward1>)


Returned tensor x:
tensor([-0.2359,  0.2123,  0.0163], grad_fn=<ViewBackward0>)
Returned tensor x shape: torch.Size([3])
```

In [29]:

```
''' This code documents why solving the 3rd error: "RuntimeError: mat1 and mat2 shapes cannot
    be multiplied (1x29 and 5x3)" in the above cell, was impossible. '''

# Get same values.
t.manual_seed(0)

# nt = new tensor. Make tensor of size 30, (0-29)
nt = t.rand(30)
print(f'New tensor:\n{nt}\nNew tensor SIZE: {nt.shape}\n\n')

# Create a linear layer which only takes 1d tensors of 5, and 2d tensors of (n, 5).
ll = t.nn.Linear(5, 1)

# Try reshaping the new tensor to be in form (n, 5). First get size and check if its divisible by embed_dim (ed) 5
ed = 5
cur_mat_size = nt.size()[0]

# Can 30 be divided by 5? If so, we can make a new tensor evenly.
if cur_mat_size % ed == 0:
  rows_for_reshape = int(cur_mat_size / ed)

  # nm = new matrix.
  nm = t.reshape(nt, (rows_for_reshape, ed))
  print(f'New reshaped matrix is:\n{nm}\nNew reshaped matrix shape: {nm.shape}\n\n')

  print(ll(nm))
```

```
New tensor:
tensor([0.4963, 0.7682, 0.0885, 0.1320, 0.3074, 0.6341, 0.4901, 0.8964, 0.4556,
        0.6323, 0.3489, 0.4017, 0.0223, 0.1689, 0.2939, 0.5185, 0.6977, 0.8000,
        0.1610, 0.2823, 0.6816, 0.9152, 0.3971, 0.8742, 0.4194, 0.5529, 0.9527,
        0.0362, 0.1852, 0.3734])
New tensor SIZE: torch.Size([30])
```

```
New reshaped matrix is:
tensor([[0.4963, 0.7682, 0.0885, 0.1320, 0.3074],
        [0.6341, 0.4901, 0.8964, 0.4556, 0.6323],
        [0.3489, 0.4017, 0.0223, 0.1689, 0.2939],
        [0.5185, 0.6977, 0.8000, 0.1610, 0.2823],
        [0.6816, 0.9152, 0.3971, 0.8742, 0.4194],
        [0.5529, 0.9527, 0.0362, 0.1852, 0.3734]])
New reshaped matrix shape: torch.Size([6, 5])


tensor([[-0.3574],
        [-0.8912],
        [-0.4575],
        [-0.5929],
        [-0.6101],
        [-0.3124]], grad_fn=<AddmmBackward0>)
```

7) Gensim

In [30]:

```
import gensim.downloader as api

'''

Commenting this out because it takes forever to download

'''

# # w2v has word embeddings and pytorch neural nets work with them.
# w2v = api.load('word2vec-google-news-300')

# print(type(w2v['taco'])) # Returns numpy.ndarray
# print(w2v['taco'][:10]) # Get first 10 values of 300 embedded vector.

# ''' [('Jackson', 0.5326348543167114),
#      ('Prince', 0.5306329727172852),
#      ('Tupou_V.', 0.5292826294898987),
#      ('KIng', 0.5227501392364502),
#      ('e_mail_robert.king_@', 0.5173623561859131)] '''
# print(w2v.similar_by_word('King', topn=5)) # This got the values in below comment.

# '''
# w is vehicle and x is 0.7821096181869507
# w is cars and x is 0.7423831224441528
# w is SUV and x is 0.7160962224006653
# w is minivan and x is 0.6907036900520325
# w is truck and x is 0.6735789775848389
# w is Car and x is 0.6677608489990234
# w is Ford_Focus and x is 0.667320191860199
# w is Honda_Civic and x is 0.6626849174499512
# w is Jeep and x is 0.651133120059967
# w is pickup_truck and x is 0.6441438794136047 '''

# # This gets comment above.
# for w, x in w2v.most_similar('car'):
#   print(f'w is {w} and x is {x}')
```

Out[30]:

```
'\n\nCommenting this out because it takes forever to download\n\n'
```

In [31]:

```
# A bit quicker to download.
te = api.load("glove-twitter-25")
```

In [32]:

```
''' .vocab can't be used with this set of embeddings, so key_to_index returns a dictionary. Also
    index_to_key returns a list which is much better.'''
len(te.key_to_index)
```

Out[32]:

```
1193514
```

In [33]:

```
all_words = te.index_to_key
selected_word = all_words[0]
emb = te[selected_word]

print(f'Word: {selected_word}\nEmbedding FOR {selected_word}:\n{emb}')
```

```
Word: <user>
Embedding FOR <user>:
[ 0.62415   0.62476  -0.082335  0.20101  -0.13741  -0.11431   0.77909
  2.6356   -0.46351   0.57465  -0.024888 -0.015466 -2.9696   -0.49876
  0.095034 -0.94879  -0.017336 -0.86349  -1.3348    0.046811  0.36999
 -0.57663  -0.48469   0.40078   0.75345 ]
```

8) Updating embedding weights with pre trained embeddings

In [34]:

```
''' The concept of this is literally one embedding layer borrowing/taking embedding values.
```

```
    In a real model, there might be certain words that don't have proper embedding values at
    all. So why not plug those holes up? First I'll use an example embedding layer. It'll
    take a vocab size and output a embed_dim size tensor.

    The issue that also happened in the EmbedClassifier model should be talked about here.
    The second error that happened was "IndexError: index out of range in self". Basically,
    to sum the error, if there's a tensor like t.tensor([15]) and the embedding layer is:
    t.nn.Embedding(num_embeddings=15, embedding_dim=5), we can't pass the tensor to it. The
    embedding first arg is the MAX value for ALL elements in a given tensor. So I need to be
    careful about the size of the embedding layer.

    Errors:
    1) "RuntimeError: a view of a leaf Variable that requires grad is being used in an
        in-place operation."

        Solution: el.weight.requires_grad = False '''

# Get max length of vocab dictionary.
el = t.nn.Embedding(num_embeddings=len(te.key_to_index), embedding_dim=25)
el.weight.requires_grad = False

num_words = 5

# Loop for first 5 words.
for i, x in zip(range(num_words), all_words):
  print(f'-----Index {i}-----\nWord: {x}\nWord embedding in glove embeddings:\n{te[x]}\n\n')
  print(f'--- Previous model embedding:\n{el.weight[i]}')
  # Update the embedding with embedding in downloaded word embeddings (currently "glove-twitter-25")
  el.weight[i] = t.tensor(te[x], dtype=t.float32)
  print(f'--- UPDATE model embedding:\n{el.weight[i]}\n\n')

# Get all new weights and display them.
updated_weights = el.weight[:num_words]
print(f'New updated first {num_words} word embeddings in embedding layer:\n{updated_weights}')
```

```
-----Index 0-----
Word: <user>
Word embedding in glove embeddings:
[ 0.62415   0.62476  -0.082335  0.20101  -0.13741  -0.11431   0.77909
  2.6356   -0.46351   0.57465  -0.024888 -0.015466 -2.9696   -0.49876
  0.095034 -0.94879  -0.017336 -0.86349  -1.3348    0.046811  0.36999
 -0.57663  -0.48469   0.40078   0.75345 ]


--- Previous model embedding:
tensor([ 0.4397,  0.1124,  0.6408,  0.4412, -0.2159, -0.7425,  0.5627,  0.2596,
         0.5229,  2.3022, -1.4689, -1.5867,  1.2032,  0.0845, -1.2001, -0.0048,
        -0.2303, -0.3918,  0.5433, -0.3952,  0.2055, -0.4503, -0.5731, -0.5554,
        -1.5312])
--- UPDATE model embedding:
tensor([ 0.6241,  0.6248, -0.0823,  0.2010, -0.1374, -0.1143,  0.7791,  2.6356,
        -0.4635,  0.5746, -0.0249, -0.0155, -2.9696, -0.4988,  0.0950, -0.9488,
        -0.0173, -0.8635, -1.3348,  0.0468,  0.3700, -0.5766, -0.4847,  0.4008,
         0.7534])


-----Index 1-----
Word: .
Word embedding in glove embeddings:
[ 0.69586  -1.1469   -0.41797  -0.022311 -0.023801  0.82358   1.2228
  1.741    -0.90979   1.3725    0.1153   -0.63906  -3.2252    0.61269
  0.33544  -0.57058  -0.50861  -0.16575  -0.98153  -0.8213    0.24333
 -0.14482  -0.67877   0.7061    0.40833 ]


--- Previous model embedding:
tensor([-1.2341,  1.8197, -0.5515, -1.3253,  0.1886, -0.0691, -0.4949, -1.4782,
         2.5672, -0.4731,  0.3356,  1.5091,  2.0820,  1.7067,  2.3804, -1.0670,
         1.1149, -0.1407,  0.8058,  0.3276, -0.7607, -1.5991,  0.0185,  0.8419,
        -0.4000])
--- UPDATE model embedding:
tensor([ 0.6959, -1.1469, -0.4180, -0.0223, -0.0238,  0.8236,  1.2228,  1.7410,
        -0.9098,  1.3725,  0.1153, -0.6391, -3.2252,  0.6127,  0.3354, -0.5706,
        -0.5086, -0.1657, -0.9815, -0.8213,  0.2433, -0.1448, -0.6788,  0.7061,
         0.4083])


-----Index 2-----
Word: :
Word embedding in glove embeddings:
[ 1.1242    0.054519 -0.037362  0.10046   0.11923  -0.30009   1.0938
  2.537    -0.072802  1.0491    1.0931    0.066084 -2.7036   -0.14391
 -0.22031  -0.99347  -0.65072  -0.030948 -1.0817   -0.64701   0.32341
 -0.41612  -0.5268   -0.047166  0.71549 ]


--- Previous model embedding:
tensor([ 1.0395,  0.3582, -0.0033, -0.5344,  1.1687,  0.3945, -1.0450, -0.9565,
         0.0335,  0.7101, -1.5353, -0.4127,  0.9663,  1.6248, -2.6133, -1.6965,
        -0.2282,  0.2800,  0.0732,  1.1133,  0.2823,  0.4342,  0.4569, -0.8654,
         0.7813])
--- UPDATE model embedding:
tensor([ 1.1242,  0.0545, -0.0374,  0.1005,  0.1192, -0.3001,  1.0938,  2.5370,
        -0.0728,  1.0491,  1.0931,  0.0661, -2.7036, -0.1439, -0.2203, -0.9935,
        -0.6507, -0.0309, -1.0817, -0.6470,  0.3234, -0.4161, -0.5268, -0.0472,
         0.7155])


-----Index 3-----
Word: rt
Word embedding in glove embeddings:
[ 0.74056  0.9155  -0.16352  0.35843  0.05266  0.1456   1.0421   2.8073
  0.12865  1.0492   0.13033  0.20508 -2.6686  -0.50551 -0.29574 -0.91433
```

```
 -0.40456 -1.0988  -1.0333  -0.17875  0.37979 -0.25922 -0.74854  0.36001
  0.61206]


--- Previous model embedding:
tensor([-0.9268,  0.2064, -0.3334, -0.4288,  0.2329,  0.9625,  0.3492, -0.9215,
        -0.0562, -0.7015,  1.0367, -0.6037, -1.2788,  0.1239,  1.1648,  0.9234,
         1.3873,  1.3750,  0.6596,  0.4766, -1.0163,  0.6104,  0.4669,  1.9507,
        -1.0631])
--- UPDATE model embedding:
tensor([ 0.7406,  0.9155, -0.1635,  0.3584,  0.0527,  0.1456,  1.0421,  2.8073,
         0.1286,  1.0492,  0.1303,  0.2051, -2.6686, -0.5055, -0.2957, -0.9143,
        -0.4046, -1.0988, -1.0333, -0.1787,  0.3798, -0.2592, -0.7485,  0.3600,
         0.6121])


-----Index 4-----
Word: ,
Word embedding in glove embeddings:
[ 0.84705  -1.0349   -0.050419  0.27164  -0.58659   0.99514   0.25267
  1.6963    0.10313   0.80073   0.74655  -1.2667   -4.036    -0.22557
  0.16322  -0.67015  -0.64812   0.010373 -0.71889  -0.74997   0.24862
  0.10319  -1.1732    0.58196   0.33846 ]


--- Previous model embedding:
tensor([ 1.1404, -0.0899,  0.7298, -1.8453, -0.1021, -1.0335, -0.3126,  0.2458,
         0.3772,  1.1012, -1.1428,  0.0376,  0.2886,  0.3866, -0.2011, -0.1179,
        -0.8294, -1.4073,  1.6268,  0.1723, -0.7043,  0.3147,  0.1574,  0.3854,
         0.5737])
--- UPDATE model embedding:
tensor([ 0.8471, -1.0349, -0.0504,  0.2716, -0.5866,  0.9951,  0.2527,  1.6963,
         0.1031,  0.8007,  0.7466, -1.2667, -4.0360, -0.2256,  0.1632, -0.6701,
        -0.6481,  0.0104, -0.7189, -0.7500,  0.2486,  0.1032, -1.1732,  0.5820,
         0.3385])


New updated first 5 word embeddings in embedding layer:
tensor([[ 0.6241,  0.6248, -0.0823,  0.2010, -0.1374, -0.1143,  0.7791,  2.6356,
         -0.4635,  0.5746, -0.0249, -0.0155, -2.9696, -0.4988,  0.0950, -0.9488,
         -0.0173, -0.8635, -1.3348,  0.0468,  0.3700, -0.5766, -0.4847,  0.4008,
          0.7534],
        [ 0.6959, -1.1469, -0.4180, -0.0223, -0.0238,  0.8236,  1.2228,  1.7410,
         -0.9098,  1.3725,  0.1153, -0.6391, -3.2252,  0.6127,  0.3354, -0.5706,
         -0.5086, -0.1657, -0.9815, -0.8213,  0.2433, -0.1448, -0.6788,  0.7061,
          0.4083],
        [ 1.1242,  0.0545, -0.0374,  0.1005,  0.1192, -0.3001,  1.0938,  2.5370,
         -0.0728,  1.0491,  1.0931,  0.0661, -2.7036, -0.1439, -0.2203, -0.9935,
         -0.6507, -0.0309, -1.0817, -0.6470,  0.3234, -0.4161, -0.5268, -0.0472,
          0.7155],
        [ 0.7406,  0.9155, -0.1635,  0.3584,  0.0527,  0.1456,  1.0421,  2.8073,
          0.1286,  1.0492,  0.1303,  0.2051, -2.6686, -0.5055, -0.2957, -0.9143,
         -0.4046, -1.0988, -1.0333, -0.1787,  0.3798, -0.2592, -0.7485,  0.3600,
          0.6121],
        [ 0.8471, -1.0349, -0.0504,  0.2716, -0.5866,  0.9951,  0.2527,  1.6963,
          0.1031,  0.8007,  0.7466, -1.2667, -4.0360, -0.2256,  0.1632, -0.6701,
         -0.6481,  0.0104, -0.7189, -0.7500,  0.2486,  0.1032, -1.1732,  0.5820,
          0.3385]])
```

9) Glove embeddings & more

In [35]:

```
# ttv = torchtext vocab
ttv = torchtext.vocab.GloVe(name='6B', dim=50)
```

In [36]:

```
# Very similar to the vocab created earlier since this has stoi and itos as well
ttv["Car"]
```

Out[36]:

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.])
```

In [37]:

```
ttv.stoi
```

Out[37]:

```
{'the': 0,
 ',': 1,
 '.': 2,
 'of': 3,
 'to': 4,
 'and': 5,
 'in': 6,
 'a': 7,
 '"': 8,
 "'s": 9,
 'for': 10,
 '-': 11,
 'that': 12,
 'on': 13,
 'is': 14,
 'was': 15,
 'said': 16,
 'with': 17
```

```
 'with': 17,
 'he': 18,
 'as': 19,
 'it': 20,
 'by': 21,
 'at': 22,
 '(': 23,
 ')': 24,
 'from': 25,
 'his': 26,
 "''": 27,
 '``': 28,
 'an': 29,
 'be': 30,
 'has': 31,
 'are': 32,
 'have': 33,
 'but': 34,
 'were': 35,
 'not': 36,
 'this': 37,
 'who': 38,
 'they': 39,
 'had': 40,
 'i': 41,
 'which': 42,
 'will': 43,
 'their': 44,
 ':': 45,
 'or': 46,
 'its': 47,
 'one': 48,
 'after': 49,
 'new': 50,
 'been': 51,
 'also': 52,
 'we': 53,
 'would': 54,
 'two': 55,
 'more': 56,
 "'": 57,
 'first': 58,
 'about': 59,
 'up': 60,
 'when': 61,
 'year': 62,
 'there': 63,
 'all': 64,
 '--': 65,
 'out': 66,
 'she': 67,
 'other': 68,
 'people': 69,
 "n't": 70,
 'her': 71,
 'percent': 72,
 'than': 73,
 'over': 74,
 'into': 75,
 'last': 76,
 'some': 77,
 'government': 78,
 'time': 79,
 '$': 80,
 'you': 81,
 'years': 82,
 'if': 83,
 'no': 84,
 'world': 85,
 'can': 86,
 'three': 87,
 'do': 88,
 ';': 89,
 'president': 90,
 'only': 91,
 'state': 92,
 'million': 93,
 'could': 94,
 'us': 95,
 'most': 96,
 '_': 97,
 'against': 98,
 'u.s.': 99,
 'so': 100,
 'them': 101,
 'what': 102,
 'him': 103,
 'united': 104,
 'during': 105,
 'before': 106,
 'may': 107,
 'since': 108,
 'many': 109,
 'while': 110,
 'where': 111,
 'states': 112,
 'because': 113,
 'now': 114,
 'city': 115,
 'made': 116,
 'like': 117,
 'between': 118,
 'did': 119,
```

```
'just': 120,
'national': 121,
'day': 122,
'country': 123,
'under': 124,
'such': 125,
'second': 126,
'then': 127,
'company': 128,
'group': 129,
'any': 130,
'through': 131,
'china': 132,
'four': 133,
'being': 134,
'down': 135,
'war': 136,
'back': 137,
'off': 138,
'south': 139,
'american': 140,
'minister': 141,
'police': 142,
'well': 143,
'including': 144,
'team': 145,
'international': 146,
'week': 147,
'officials': 148,
'still': 149,
'both': 150,
'even': 151,
'high': 152,
'part': 153,
'told': 154,
'those': 155,
'end': 156,
'former': 157,
'these': 158,
'make': 159,
'billion': 160,
'work': 161,
'our': 162,
'home': 163,
'school': 164,
'party': 165,
'house': 166,
'old': 167,
'later': 168,
'get': 169,
'another': 170,
'tuesday': 171,
'news': 172,
'long': 173,
'five': 174,
'called': 175,
'1': 176,
'wednesday': 177,
'military': 178,
'way': 179,
'used': 180,
'much': 181,
'next': 182,
'monday': 183,
'thursday': 184,
'friday': 185,
'game': 186,
'here': 187,
'?': 188,
'should': 189,
'take': 190,
'very': 191,
'my': 192,
'north': 193,
'security': 194,
'season': 195,
'york': 196,
'how': 197,
'public': 198,
'early': 199,
'according': 200,
'several': 201,
'court': 202,
'say': 203,
'around': 204,
'foreign': 205,
'10': 206,
'until': 207,
'set': 208,
'political': 209,
'says': 210,
'market': 211,
'however': 212,
'family': 213,
'life': 214,
'same': 215,
'general': 216,
'-': 217,
'left': 218,
'good': 219,
'top': 220,
'university': 221,
```

```
'going': 222,
'number': 223,
'major': 224,
'known': 225,
'points': 226,
'won': 227,
'six': 228,
'month': 229,
'dollars': 230,
'bank': 231,
'2': 232,
'iraq': 233,
'use': 234,
'members': 235,
'each': 236,
'area': 237,
'found': 238,
'official': 239,
'sunday': 240,
'place': 241,
'go': 242,
'based': 243,
'among': 244,
'third': 245,
'times': 246,
'took': 247,
'right': 248,
'days': 249,
'local': 250,
'economic': 251,
'countries': 252,
'see': 253,
'best': 254,
'report': 255,
'killed': 256,
'held': 257,
'business': 258,
'west': 259,
'does': 260,
'own': 261,
'%': 262,
'came': 263,
'law': 264,
'months': 265,
'women': 266,
"'re": 267,
'power': 268,
'think': 269,
'service': 270,
'children': 271,
'bush': 272,
'show': 273,
'/': 274,
'help': 275,
'chief': 276,
'saturday': 277,
'system': 278,
'john': 279,
'support': 280,
'series': 281,
'play': 282,
'office': 283,
'following': 284,
'me': 285,
'meeting': 286,
'expected': 287,
'late': 288,
'washington': 289,
'games': 290,
'european': 291,
'league': 292,
'reported': 293,
'final': 294,
'added': 295,
'without': 296,
'british': 297,
'white': 298,
'history': 299,
'man': 300,
'men': 301,
'became': 302,
'want': 303,
'march': 304,
'case': 305,
'few': 306,
'run': 307,
'money': 308,
'began': 309,
'open': 310,
'name': 311,
'trade': 312,
'center': 313,
'3': 314,
'israel': 315,
'oil': 316,
'too': 317,
'al': 318,
'film': 319,
'win': 320,
'led': 321,
'east': 322,
'central': 323,
'20': 324,
```

```
'20': 324,
'air': 325,
'come': 326,
'chinese': 327,
'town': 328,
'leader': 329,
'army': 330,
'line': 331,
'never': 332,
'little': 333,
'played': 334,
'prime': 335,
'death': 336,
'companies': 337,
'least': 338,
'put': 339,
'forces': 340,
'past': 341,
'de': 342,
'half': 343,
'june': 344,
'saying': 345,
'know': 346,
'federal': 347,
'french': 348,
'peace': 349,
'earlier': 350,
'capital': 351,
'force': 352,
'great': 353,
'union': 354,
'near': 355,
'released': 356,
'small': 357,
'department': 358,
'every': 359,
'health': 360,
'japan': 361,
'head': 362,
'ago': 363,
'night': 364,
'big': 365,
'cup': 366,
'election': 367,
'region': 368,
'director': 369,
'talks': 370,
'program': 371,
'far': 372,
'today': 373,
'statement': 374,
'july': 375,
'although': 376,
'district': 377,
'again': 378,
'born': 379,
'development': 380,
'leaders': 381,
'council': 382,
'close': 383,
'record': 384,
'along': 385,
'county': 386,
'france': 387,
'went': 388,
'point': 389,
'must': 390,
'spokesman': 391,
'your': 392,
'member': 393,
'plan': 394,
'financial': 395,
'april': 396,
'recent': 397,
'campaign': 398,
'become': 399,
'troops': 400,
'whether': 401,
'lost': 402,
'music': 403,
'15': 404,
'got': 405,
'israeli': 406,
'30': 407,
'need': 408,
'4': 409,
'lead': 410,
'already': 411,
'russia': 412,
'though': 413,
'might': 414,
'free': 415,
'hit': 416,
'rights': 417,
'11': 418,
'information': 419,
'away': 420,
'12': 421,
'5': 422,
'others': 423,
'control': 424,
'within': 425,
'large': 426,
```

```
'economy': 427,
'press': 428,
'agency': 429,
'water': 430,
'died': 431,
'career': 432,
'making': 433,
'...': 434,
'deal': 435,
'attack': 436,
'side': 437,
'seven': 438,
'better': 439,
'less': 440,
'september': 441,
'once': 442,
'clinton': 443,
'main': 444,
'due': 445,
'committee': 446,
'building': 447,
'conference': 448,
'club': 449,
'january': 450,
'decision': 451,
'stock': 452,
'america': 453,
'given': 454,
'give': 455,
'often': 456,
'announced': 457,
'television': 458,
'industry': 459,
'order': 460,
'young': 461,
"'ve": 462,
'palestinian': 463,
'age': 464,
'start': 465,
'administration': 466,
'russian': 467,
'prices': 468,
'round': 469,
'december': 470,
'nations': 471,
"'m": 472,
'human': 473,
'india': 474,
'defense': 475,
'asked': 476,
'total': 477,
'october': 478,
'players': 479,
'bill': 480,
'important': 481,
'southern': 482,
'move': 483,
'fire': 484,
'population': 485,
'rose': 486,
'november': 487,
'include': 488,
'further': 489,
'nuclear': 490,
'street': 491,
'taken': 492,
'media': 493,
'different': 494,
'issue': 495,
'received': 496,
'secretary': 497,
'return': 498,
'college': 499,
'working': 500,
'community': 501,
'eight': 502,
'groups': 503,
'despite': 504,
'level': 505,
'largest': 506,
'whose': 507,
'attacks': 508,
'germany': 509,
'august': 510,
'change': 511,
'church': 512,
'nation': 513,
'german': 514,
'station': 515,
'london': 516,
'weeks': 517,
'having': 518,
'18': 519,
'research': 520,
'black': 521,
'services': 522,
'story': 523,
'6': 524,
'europe': 525,
'sales': 526,
'policy': 527,
'visit': 528,
```

```
'northern': 529,
'lot': 530,
'across': 531,
'per': 532,
'current': 533,
'board': 534,
'football': 535,
'ministry': 536,
'workers': 537,
'vote': 538,
'book': 539,
'fell': 540,
'seen': 541,
'role': 542,
'students': 543,
'shares': 544,
'iran': 545,
'process': 546,
'agreement': 547,
'quarter': 548,
'full': 549,
'match': 550,
'started': 551,
'growth': 552,
'yet': 553,
'moved': 554,
'possible': 555,
'western': 556,
'special': 557,
'100': 558,
'plans': 559,
'interest': 560,
'behind': 561,
'strong': 562,
'england': 563,
'named': 564,
'food': 565,
'period': 566,
'real': 567,
'authorities': 568,
'car': 569,
'term': 570,
'rate': 571,
'race': 572,
'nearly': 573,
'korea': 574,
'enough': 575,
'site': 576,
'opposition': 577,
'keep': 578,
'25': 579,
'call': 580,
'future': 581,
'taking': 582,
'island': 583,
'2008': 584,
'2006': 585,
'road': 586,
'outside': 587,
'really': 588,
'century': 589,
'democratic': 590,
'almost': 591,
'single': 592,
'share': 593,
'leading': 594,
'trying': 595,
'find': 596,
'album': 597,
'senior': 598,
'minutes': 599,
'together': 600,
'congress': 601,
'index': 602,
'australia': 603,
'results': 604,
'hard': 605,
'hours': 606,
'land': 607,
'action': 608,
'higher': 609,
'field': 610,
'cut': 611,
'coach': 612,
'elections': 613,
'san': 614,
'issues': 615,
'executive': 616,
'february': 617,
'production': 618,
'areas': 619,
'river': 620,
'face': 621,
'using': 622,
'japanese': 623,
'province': 624,
'park': 625,
'price': 626,
'commission': 627,
'california': 628,
'father': 629,
'son': 630,
'education': 631,
```

```
'education': 631,
'7': 632,
'village': 633,
'energy': 634,
'shot': 635,
'short': 636,
'africa': 637,
'key': 638,
'red': 639,
'association': 640,
'average': 641,
'pay': 642,
'exchange': 643,
'eu': 644,
'something': 645,
'gave': 646,
'likely': 647,
'player': 648,
'george': 649,
'2007': 650,
'victory': 651,
'8': 652,
'low': 653,
'things': 654,
'2010': 655,
'pakistan': 656,
'14': 657,
'post': 658,
'social': 659,
'continue': 660,
'ever': 661,
'look': 662,
'chairman': 663,
'job': 664,
'2000': 665,
'soldiers': 666,
'able': 667,
'parliament': 668,
'front': 669,
'himself': 670,
'problems': 671,
'private': 672,
'lower': 673,
'list': 674,
'built': 675,
'13': 676,
'efforts': 677,
'dollar': 678,
'miles': 679,
'included': 680,
'radio': 681,
'live': 682,
'form': 683,
'david': 684,
'african': 685,
'increase': 686,
'reports': 687,
'sent': 688,
'fourth': 689,
'always': 690,
'king': 691,
'50': 692,
'tax': 693,
'taiwan': 694,
'britain': 695,
'16': 696,
'playing': 697,
'title': 698,
'middle': 699,
'meet': 700,
'global': 701,
'wife': 702,
'2009': 703,
'position': 704,
'located': 705,
'clear': 706,
'ahead': 707,
'2004': 708,
'2005': 709,
'iraqi': 710,
'english': 711,
'result': 712,
'release': 713,
'violence': 714,
'goal': 715,
'project': 716,
'closed': 717,
'border': 718,
'body': 719,
'soon': 720,
'crisis': 721,
'division': 722,
'&amp;': 723,
'served': 724,
'tour': 725,
'hospital': 726,
'kong': 727,
'test': 728,
'hong': 729,
'u.n.': 730,
'inc.': 731,
'technology': 732,
'believe': 733,
```

```
'organization': 734,
'published': 735,
'weapons': 736,
'agreed': 737,
'why': 738,
'nine': 739,
'summer': 740,
'wanted': 741,
'republican': 742,
'act': 743,
'recently': 744,
'texas': 745,
'course': 746,
'problem': 747,
'senate': 748,
'medical': 749,
'un': 750,
'done': 751,
'reached': 752,
'star': 753,
'continued': 754,
'investors': 755,
'living': 756,
'care': 757,
'signed': 758,
'17': 759,
'art': 760,
'provide': 761,
'worked': 762,
'presidential': 763,
'gold': 764,
'obama': 765,
'morning': 766,
'dead': 767,
'opened': 768,
"'ll": 769,
'event': 770,
'previous': 771,
'cost': 772,
'instead': 773,
'canada': 774,
'band': 775,
'teams': 776,
'daily': 777,
'2001': 778,
'available': 779,
'drug': 780,
'coming': 781,
'2003': 782,
'investment': 783,
'’s': 784,
'michael': 785,
'civil': 786,
'woman': 787,
'training': 788,
'appeared': 789,
'9': 790,
'involved': 791,
'indian': 792,
'similar': 793,
'situation': 794,
'24': 795,
'los': 796,
'running': 797,
'fighting': 798,
'mark': 799,
'40': 800,
'trial': 801,
'hold': 802,
'australian': 803,
'thought': 804,
'!': 805,
'study': 806,
'fall': 807,
'mother': 808,
'met': 809,
'relations': 810,
'anti': 811,
'2002': 812,
'song': 813,
'popular': 814,
'base': 815,
'tv': 816,
'ground': 817,
'markets': 818,
'ii': 819,
'newspaper': 820,
'staff': 821,
'saw': 822,
'hand': 823,
'hope': 824,
'operations': 825,
'pressure': 826,
'americans': 827,
'eastern': 828,
'st.': 829,
'legal': 830,
'asia': 831,
'budget': 832,
'returned': 833,
'considered': 834,
'love': 835,
```

```
'wrote': 836,
'stop': 837,
'fight': 838,
'currently': 839,
'charges': 840,
'try': 841,
'aid': 842,
'ended': 843,
'management': 844,
'brought': 845,
'cases': 846,
'decided': 847,
'failed': 848,
'network': 849,
'works': 850,
'gas': 851,
'turned': 852,
'fact': 853,
'vice': 854,
'ca': 855,
'mexico': 856,
'trading': 857,
'especially': 858,
'reporters': 859,
'afghanistan': 860,
'common': 861,
'looking': 862,
'space': 863,
'rates': 864,
'manager': 865,
'loss': 866,
'2011': 867,
'justice': 868,
'thousands': 869,
'james': 870,
'rather': 871,
'fund': 872,
'thing': 873,
'republic': 874,
'opening': 875,
'accused': 876,
'winning': 877,
'scored': 878,
'championship': 879,
'example': 880,
'getting': 881,
'biggest': 882,
'performance': 883,
'sports': 884,
'1998': 885,
'let': 886,
'allowed': 887,
'schools': 888,
'means': 889,
'turn': 890,
'leave': 891,
'no.': 892,
'robert': 893,
'personal': 894,
'stocks': 895,
'showed': 896,
'light': 897,
'arrested': 898,
'person': 899,
'either': 900,
'offer': 901,
'majority': 902,
'battle': 903,
'19': 904,
'class': 905,
'evidence': 906,
'makes': 907,
'society': 908,
'products': 909,
'regional': 910,
'needed': 911,
'stage': 912,
'am': 913,
'doing': 914,
'families': 915,
'construction': 916,
'various': 917,
'1996': 918,
'sold': 919,
'independent': 920,
'kind': 921,
'airport': 922,
'paul': 923,
'judge': 924,
'internet': 925,
'movement': 926,
'room': 927,
'followed': 928,
'original': 929,
'angeles': 930,
'italy': 931,
'`': 932,
'data': 933,
'comes': 934,
'parties': 935,
'nothing': 936,
'sea': 937,
'bring': 938,
```

```
 'bring': 938,
 '2012': 939,
 'annual': 940,
 'officer': 941,
 'beijing': 942,
 'present': 943,
 'remain': 944,
 'nato': 945,
 '1999': 946,
 '22': 947,
 'remains': 948,
 'allow': 949,
 'florida': 950,
 'computer': 951,
 '21': 952,
 'contract': 953,
 'coast': 954,
 'created': 955,
 'demand': 956,
 'operation': 957,
 'events': 958,
 'islamic': 959,
 'beat': 960,
 'analysts': 961,
 'interview': 962,
 'helped': 963,
 'child': 964,
 'probably': 965,
 'spent': 966,
 'asian': 967,
 'effort': 968,
 'cooperation': 969,
 'shows': 970,
 'calls': 971,
 'investigation': 972,
 'lives': 973,
 'video': 974,
 'yen': 975,
 'runs': 976,
 'tried': 977,
 'bad': 978,
 'described': 979,
 '1994': 980,
 'toward': 981,
 'written': 982,
 'throughout': 983,
 'established': 984,
 'mission': 985,
 'associated': 986,
 'buy': 987,
 'growing': 988,
 'green': 989,
 'forward': 990,
 'competition': 991,
 'poor': 992,
 'latest': 993,
 'banks': 994,
 'question': 995,
 '1997': 996,
 'prison': 997,
 'feel': 998,
 'attention': 999,
 ...}
```

In [38]:

```
ttv.itos[379]
```

Out[38]:

```
'born'
```

In [39]:

```
# ft = first tensor, st = second tensor
ft = t.tensor([1,2,3,4,5])
st = t.tensor([9,8,7,6,-1])

tt = t.tensor([[3,2,4],
               [1,5,7]])

# 1) add up a 1d tensor. Both below gets 15.
# print(ft.sum())
# print(t.sum(ft))

# 2) Add 2d tensor
# print(tt.sum()) # Total 22
# print(tt.sum(0)) # 0 = columns, so it adds by column.
# print(tt.sum(1)) # 1 = rows, so it adds by row

''' 3) Predictons and labels
       In code projects something like "torch.sum(predictions == labels).item()" so
       why not mess around with that here.

       x = ground truths
       y = predictions
'''

x = t.tensor([1,0,1,1,1,1,0,1]) # ground truths
y = t.tensor([1,1,0,1,1,1,0,1]) # predictions

''' == will get boolean vec on which columns MATCH. Ex in x and y above, index 0 is
    both 1, so that's true, they equal eachother. Then the sum just how many Trues and
```

```
    adds them since in a numerical fashion True is 1 and False is 0. So:
    t.sum(tensor([ True, False, False,  True,  True,  True,  True,  True])) will be 6.
    And of course .item just gets the number itself, out of the tensor. '''
# print(x == y)
# print(t.sum(x == y))
print(t.sum(x == y).item())
```

```
6
```

In [40]:

```
''' Argmax is cool, simply gets the indices of highest values.
    1 - Vertical/Row by row.
    0 - Horizontal/Column by column.

    So:
    1 - Indices 1,0,1,2 and the numbers are 7,10,5,22.
    0 - Indices 1,3,3 and the numbers are 10,20,22.

    Argmin is opposite. '''

z = t.tensor([[3, 7, 1],
              [10, 3, 5],
              [4, 5, 1],
              [4, 20, 22]])

print(f'---Argmax---')
print(t.argmax(z, 1))
print(t.argmax(z, 0))
print('\n\n')

print(f'---Argmin---')
print(t.argmin(z, 1))
print(t.argmin(z, 0))
```

```
---Argmax---
tensor([1, 0, 1, 2])
tensor([1, 3, 3])



---Argmin---
tensor([2, 1, 2, 0])
tensor([0, 1, 0])
```

10) Packed Sequences

In [41]:

```
''' What are packed sequences?
    Mainly for RNN.

    [[1,2,3,4,5],
     [6,7,8,0,0],
     [9,0,0,0,0]]"
    Len is 5,3,1. RNN cells train with 1,6,9. Then 2,7. Then 3,8 Etc. So packed sequence is one
    vec "[1,6,9,2,7,3,8,4,5]" Len of vec still 5,3,1.

    First we need sentences of different lengths. "ts" is from section 1, tokenized sentences. '''

for tks in ts:
  print(tks)
```

```
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(', 'reuters', ')', 'reuters', '-', 'short-sellers', ',', 'wa
ll', 'street', "'", 's', 'dwindling\\band', 'of', 'ultra-cynics', ',', 'are', 'seeing', 'green', 'again', '.']
['carlyle', 'looks', 'toward', 'commercial', 'aerospace', '(', 'reuters', ')', 'reuters', '-', 'private', 'investment', 'firm', 'c
arlyle', 'group', ',', '\\which', 'has', 'a', 'reputation', 'for', 'making', 'well-timed', 'and', 'occasionally\\controversial', '
plays', 'in', 'the', 'defense', 'industry', ',', 'has', 'quietly', 'placed\\its', 'bets', 'on', 'another', 'part', 'of', 'the', 'm
arket', '.']
['oil', 'and', 'economy', 'cloud', 'stocks', "'", 'outlook', '(', 'reuters', ')', 'reuters', '-', 'soaring', 'crude', 'prices', 'p
lus', 'worries\\about', 'the', 'economy', 'and', 'the', 'outlook', 'for', 'earnings', 'are', 'expected', 'to\\hang', 'over', 'the'
, 'stock', 'market', 'next', 'week', 'during', 'the', 'depth', 'of', 'the\\summer', 'doldrums', '.']
```

In [42]:

```
# encode them, function in section 1.
num_sents = [encode(tks) for tks in ts]

for ns in num_sents:
  print(ns)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, 13, 14, 0, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24, 2]
[25, 26, 27, 28, 29, 9, 10, 11, 10, 12, 30, 31, 32, 25, 33, 14, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 7, 45, 46, 14, 35, 47,
48, 49, 50, 51, 52, 19, 7, 53, 2]
[54, 41, 55, 56, 57, 16, 58, 9, 10, 11, 10, 12, 59, 60, 61, 62, 63, 7, 55, 41, 7, 58, 38, 64, 21, 65, 66, 67, 7, 68, 53, 69, 70, 71
, 7, 72, 19, 73, 74, 2]
```

In [43]:

```
# Now pad. This code was written already in section 1 as well.
padded_sents = [t.nn.functional.pad(t.tensor(sent), (0, max_length - len(sent))) for sent in num_sents]

padded_sents
```

Out[43]:

```
[tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 10, 12, 13, 14,  0, 15,
         16, 17, 18, 19, 20, 14, 21, 22, 23, 24,  2,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0]),
 tensor([25, 26, 27, 28, 29,  9, 10, 11, 10, 12, 30, 31, 32, 25, 33, 14, 34, 35,
         36, 37, 38, 39, 40, 41, 42, 43, 44,  7, 45, 46, 14, 35, 47, 48, 49, 50,
```

```
        51, 52, 19,  7, 53,  2]),
 tensor([54, 41, 55, 56, 57, 16, 58,  9, 10, 11, 10, 12, 59, 60, 61, 62, 63,  7,
        55, 41,  7, 58, 38, 64, 21, 65, 66, 67,  7, 68, 53, 69, 70, 71,  7, 72,
        19, 73, 74,  2,  0,  0])]
```

In [44]:

```
# tens_len = t.tensor(len(padded_sents[0]), dtype=t.int64, device='cpu')
# tens_len = tens_len.unsqueeze(0)

# t.nn.utils.rnn.pack_padded_sequence(padded_num_sents[0], lengths=tens_len)

# Create 2d tensor of 0s to be used and have the num sents added to them soon.
a = t.zeros((len(padded_sents), len(padded_sents[0])))

# Get lengths of each numerical sentence
num_sent_lens = [len(s) for s in num_sents]
print(f'Num sent lengths: {num_sent_lens}')

# Get longest one, in this case it's 42.
max_len = max(num_sent_lens)
print(f'Max length: {max_len}\n')

''' Loop over the numerical sentences ALONG with the specific lengths. But do so with index,
    that's why enumerate is used with "i". Remember "a" is a 2d tensor and I can access each
    index with "a[i, etc]". num_sent will be current numerical sentence, and same concept for
    sent_len, which will be the current length.

    Ex numerical sentence: " 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.,
    10., 12., 13., 14.,  0., 15., 16., 17., 18., 19., 20., 14., 21., 22., 23., 24.,
    2."
    Ex length: 29

    So if i is 0, then in the tensor of 0's, which is "a", go to the tensor in index 0
    which would be "[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0.]" and UP UNTIL the numerical sentence length, which is ":sent_len",
    apply the real numerical sentence. '''
for i, (num_sent, sent_len) in enumerate(zip(num_sents, num_sent_lens)):
  a[i, :sent_len] = t.FloatTensor(num_sent)

a
```

```
Num sent lengths: [29, 42, 40]
Max length: 42
```

Out[44]:

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 10., 12.,
         13., 14.,  0., 15., 16., 17., 18., 19., 20., 14., 21., 22., 23., 24.,
          2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [25., 26., 27., 28., 29.,  9., 10., 11., 10., 12., 30., 31., 32., 25.,
         33., 14., 34., 35., 36., 37., 38., 39., 40., 41., 42., 43., 44.,  7.,
         45., 46., 14., 35., 47., 48., 49., 50., 51., 52., 19.,  7., 53.,  2.],
        [54., 41., 55., 56., 57., 16., 58.,  9., 10., 11., 10., 12., 59., 60.,
         61., 62., 63.,  7., 55., 41.,  7., 58., 38., 64., 21., 65., 66., 67.,
          7., 68., 53., 69., 70., 71.,  7., 72., 19., 73., 74.,  2.,  0.,  0.]])
```

In [45]:

```
t.LongTensor(num_sent_lens)
```

Out[45]:

```
tensor([29, 42, 40])
```

In [46]:

```
x, y = t.LongTensor(num_sent_lens).sort(0, descending=True)

print(x)
print(y)
```

```
tensor([42, 40, 29])
tensor([1, 2, 0])
```