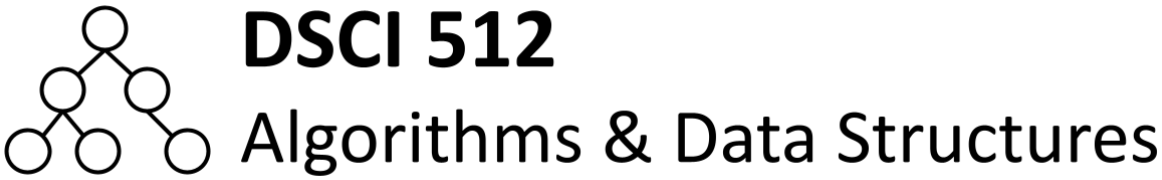
Lecture 4 Sparse matrices

Contents



Topics

- Dask and parallelized computation on CPUs
- Pytorch and GPU computation
- Data types
- Sparse matrices
- Bag-of-word matrices

Learning objectives

- Identify situations where matrix parallelization is beneficial or necessary.
- Apply array operations using GPUs.
- Understand the advantages and disadvantages of different floating-point precision levels.
- Understand sparse matrix representations.
- Select the appropriate sparse matrix representation for different types of operations.

Block matrix operations

A block matrix is a matrix that is divided into smaller rectangular or square submatrices, which are called blocks.

When doing matrix multiplication, you can split a large matrix into smaller blocks and perform operations on these blocks independently. You can get the exact same result as doing multiplication with the whole matrix.

```
import numpy as np

# Create a large matrix A
A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12],
              [13, 14, 15, 16]])

# Partition A into four blocks
A11 = A[:2, :2]
A12 = A[:2, 2:]
A21 = A[2:, :2]
A22 = A[2:, 2:]

# Compute A^2 using block matrix operations
A_squared_block = np.block([[np.dot(A11, A11) + np.dot(A12, A21), np.dot(A11,
                            [np.dot(A21, A11) + np.dot(A22, A21), np.dot(A21,

# Compute A^2 using standard matrix multiplication
A_squared = np.dot(A, A)

# Check if the two results are equivalent
equivalent = np.array_equal(A_squared, A_squared_block)

print("A^2 computed using block operations:\n", A_squared_block)
print("\nA^2 computed using standard matrix multiplication:\n", A_squared)
print("\nAre the results equivalent?", equivalent)
```

```
A^2 computed using block operations:
 [[ 90 100 110 120]
 [202 228 254 280]
 [314 356 398 440]
 [426 484 542 600]]

A^2 computed using standard matrix multiplication:
 [[ 90 100 110 120]
 [202 228 254 280]
 [314 356 398 440]
 [426 484 542 600]]

Are the results equivalent? True
```

We can utilize this nice property to speed up matrix computation. We can do operations on individual blocls means we can parallelize the operations, that is, distribute the task to individual processes.

Dask is a flexible parallel computing library for Python that is designed to scale computations across multiple cores or even distributed systems.

To use dask, we first need to start the client.

We need to Start the Client. This is just for setting up dask. You need to do it once at the start of your program and ignore it afterwards.

```
import dask
from dask.distributed import Client
# set a temporary folder to store some intermediate results generated by dask.
dask.config.set(temporary_directory='../dask_tmp')
client = Client(n_workers=2)
# set how many processes you want. this depends on how many cores your compute
client # get client information
```

Client

Client-1e71d5ec-92fc-11ef-977c-ca2a41bcfd8c

Connection method: Cluster object **Cluster type:** distributed.LocalCluster

Dashboard: http://127.0.0.1:8787/status

Cluster Info

Here is an example of numpy vs. dask.

```
import numpy as np
import dask.array as da
import time

# Create a large NumPy array
numpy_array = np.random.rand(10**8)

# Create a large Dask array
# here we split the numpy array into chunks of 10**6
dask_array = da.from_array(numpy_array, chunks=10**6).persist()

# Perform a computationally intensive operation using NumPy
def complex_operation(arr):
    return np.sin(arr) + np.cos(arr) * np.tan(arr)

start_time = time.time()
result_numpy = complex_operation(numpy_array)
print("NumPy time:", time.time() - start_time)

# Perform the same operation using Dask with parallelism
# Dask will automatically manage the parallization on individual chunks for yo
# its syntax is almost the same as numpy
def complex_operation_dask(arr):
    return da.sin(arr) + da.cos(arr) * da.tan(arr)

start_time = time.time()
result_dask = complex_operation_dask(dask_array)
result_dask = result_dask.compute()  # Compute the result in parallel. If you
print("Dask time:", time.time() - start_time)

# Compare the results
print(np.allclose(result_numpy, result_dask))
```

```
/Users/hedayatzarkoob/miniforge3/lib/python3.11/site-packages/distributed/clie
This may cause some slowdown.
Consider loading the data with Dask directly
 or using futures or delayed objects to embed the data into the graph without
See also https://docs.dask.org/en/stable/best-practices.html#load-data-with-da
  warnings.warn(
```

```
NumPy time: 2.5162911415100098
Dask time: 1.0489699840545654
True
```

Let's choose a different chunk size.

```
# Create a large NumPy array
numpy_array = np.random.rand(10**8)

# Create a large Dask array
dask_array = da.from_array(numpy_array, chunks=10**5).persist()

# Perform the same operation using Dask with parallelism
# Dask will automatically manage the parallization on individual chunks for yo
# its syntax is almost the same as numpy
def complex_operation_dask(arr):
    return da.sin(arr) + da.cos(arr) * da.tan(arr)

start_time = time.time()
result_dask = complex_operation_dask(dask_array)
result_dask = result_dask.compute()  # Compute the result in parallel. If you
print("Dask time:", time.time() - start_time)

# Compare the results
print(np.allclose(result_numpy, result_dask))
```

```
/Users/hedayatzarkoob/miniforge3/lib/python3.11/site-packages/distributed/clie
This may cause some slowdown.
Consider loading the data with Dask directly
 or using futures or delayed objects to embed the data into the graph without
See also https://docs.dask.org/en/stable/best-practices.html#load-data-with-da
  warnings.warn(
```

```
Dask time: 3.376739025115967
False
```

It's now taking a much longer time! Why?

There is communication overhead associated with parallelized computation. If the chunksize is small, the processing time might be overwhelmed by the time for communication between different processes, bringing no speed-up at all!

Generally speaking, the chunks should be more than the number of cores you allocated (why? if you have 4 cores and split the matrix into 2 chunks, what will happen?). The chunks should not be too small. In this situation, the time needed to process each chunk is much less than the communication overhead across processes, such that we are spending most of our time waiting for the communication between processes.

Paralleling matrices across CPUs can speed up the computation. We can get even much more speed-up by using GPUs!

Massively parallelized matrix operations

PyTorch is a machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is free and open-source software released under the modified BSD license.

Numpy arrays are mainly used in typical machine learning algorithms (such as k-means or Decision Tree in scikit-learn) whereas pytorch tensors are mainly used in deep learning which requires heavy matrix computation.

Example:

Cosine similarity measures the cosine of the angle between two non-zero vectors a and b. It quantifies how similar the two vectors are, regardless of their magnitude (length). Because we normalize them to be length 1.

$$sim(a, b) = \frac{a}{||a||} \cdot \frac{b}{||b||} = \frac{ab}{||a|| \cdot ||b||}$$

The value of cosine similarity ranges from -1 to 1:

- 1 means the vectors are identical (pointing in the same direction).
- 0 means the vectors are orthogonal (perpendicular), indicating no similarity.
- -1 means the vectors are diametrically opposed (pointing in opposite directions).

```
import numpy as np
import torch
import time

# Define the matrix sizes
size = 500000
num_iterations = 10

# NumPy Matrix Multiplication
a = np.random.rand(size)
b = np.random.rand(size)
```

```
%%timeit

for _ in range(num_iterations):
    np_result = np.dot(a, b)/(np.linalg.norm(a)*np.linalg.norm(b))
```

```
1.99 ms ± 5.27 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
# PyTorch Matrix Multiplication
a = torch.rand(size)
b = torch.rand(size)
```

```
%%timeit
for _ in range(num_iterations):
    torch_result = torch.dot(a, b)/(torch.norm(a)*torch.norm(b))
```

```
1.05 ms ± 100 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Pytorch isn't that fast on CPU. However, it is optimized for GPU computation.

Tips: if you don't have a GPU on your computer, you can open this notebook on Google Colab and select T4 as your runtime. Then you can run you jobs on GPU.

```
device = 'mps' #change to CUDA is your are using an Nvidia GPU
# PyTorch Matrix Multiplication
a = torch.rand(size).to(device)
b = torch.rand(size).to(device)
```

```
%%timeit
for _ in range(num_iterations):
    torch_result = torch.dot(a, b)/(torch.norm(a)*torch.norm(b))
```

```
734 µs ± 9.61 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

What is a GPU?

A GPU, or Graphics Processing Unit, is a special type of computer chip originally designed to handle complex tasks related to graphics and images. However, GPU's ability to handle

complex matrix operations makes it possible to train neural networks like AlexNet. Hence begins the neural network revolution that you and I are currently experiencing.

PyTorch's significant speed advantage when using GPUs compared to CPUs is due to several key factors:

- **Parallelism**: Modern GPUs are designed with thousands of cores, which allow them to perform many operations in parallel. PyTorch is capable of harnessing this parallelism, making it highly efficient for tasks like deep learning where many matrix multiplications occur simultaneously.
- **Optimized Libraries**: PyTorch leverages highly optimized GPU libraries, such as NVIDIA cuBLAS and cuDNN, to accelerate mathematical operations. These libraries are finely tuned for specific GPU architectures and provide substantial speed improvements over CPU-based implementations.
- **Data Transfer**: When using PyTorch on a GPU, data transfer between the CPU and GPU is typically minimized. This is achieved by loading data directly onto the GPU and keeping it there as much as possible, avoiding costly data transfers that can occur when using CPUs.

It's important to note that not all tasks benefit equally from GPU acceleration. Simple, small-scale operations may not see a substantial speed improvement on GPUs compared to CPUs. However, for computationally intensive tasks such as deep learning, scientific simulations, and large-scale linear algebra operations, GPUs can provide orders of magnitude in performance improvements.

Pytorch has similar operations in Numpy, but with a slightly different syntax. For example, an array in numpy is called tensor in pytorch. Below are some simple examples.

```
# Create a tensor
x = torch.Tensor(2, 3, 4)
print(x)

# Create a tensor from a (nested) list
x = torch.Tensor([[1, 2], [3, 4]])
print(x)

# Create a tensor with random values between 0 and 1 with the shape [2, 3, 4]
x = torch.rand(2, 3, 4)
print(x)
```

```
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.]]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[[0.1047, 0.0723, 0.9731, 0.2648],
         [0.9728, 0.1515, 0.4555, 0.0548],
         [0.1664, 0.4687, 0.0307, 0.4851]],

        [[0.1981, 0.0235, 0.7364, 0.1675],
         [0.4887, 0.9179, 0.9276, 0.7995],
         [0.3948, 0.0469, 0.2908, 0.3455]]])
```

```
shape = x.shape
print("Shape:", x.shape)

size = x.size()
print("Size:", size)

dim1, dim2, dim3 = x.size()
print("Size:", dim1, dim2, dim3)
```

```
Shape: torch.Size([2, 3, 4])
Size: torch.Size([2, 3, 4])
Size: 2 3 4
```

```
x = torch.arange(12).view(3, 4)
print("X", x)

print(x[:, 1])   # Second column
print(x[0])      # First row
print(x[:2, -1]) # First two rows, last column
print(x[1:3, :]) # Middle two rows
```

```
X tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
tensor([1, 5, 9])
tensor([0, 1, 2, 3])
tensor([3, 7])
tensor([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

```
x = torch.arange(6)
x = x.view(2, 3) # view is reshape in numpy
print("X", x)
```

```
X tensor([[0, 1, 2],
        [3, 4, 5]])
```

```
W = torch.arange(9).view(3, 3) # We can also stack multiple operations in a si
print("W", W)
```

```
W tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

```
h = torch.matmul(x, W) # Verify the result by calculating it by hand too!
print("h", h)
```

```
h tensor([[15, 18, 21],
        [42, 54, 66]])
```

Speeding up matrix multiplication with half-precision

Note. A GPU is needed to run this experiment.

In computer engineering, decimal numbers like 1.0151 or 566132.8 are traditionally represented as floating point numbers. Since we can have infinitely precise numbers (think π), but limited space in which to store them, we have to make a compromise between precision (the number of decimals we can include in a number before we have to start rounding it) and size (how many bits we use to store the number).

The technical standard for floating point numbers, IEEE 754 (for a deep dive I recommend the PyCon 2019 talk Floats are Friends: making the most of IEEE754.0000000000000002), sets the following standards:

- `fp64`, aka double-precision or “double”, max rounding error of ~2^-52

- `fp32`, aka single-precision or "single", max rounding error of ~2^-23
- `fp16`, aka half-precision or "half", max rounding error of ~2^-10

Python uses fp64 for the float type. PyTorch, which is much more memory-sensitive, uses fp32 as its default dtype instead.

If we halve the precision (fp32 → fp16), we halve the time and space at the cost of precision (only if this is supported by specific hardware, i.e. GPU).

It's rare that we need to do highly accurate matrix computations at scale. In fact, often we're doing some kind of machine learning, and less accurate approaches can prevent overfitting.

If we accept some decrease in accuracy, then we can often increase speed by orders of magnitude (and/or decrease memory use).

Let's look at an example with cosine similarity.

```
# Define the matrix sizes
size = 500000
num_iterations = 10000

# NumPy Matrix Multiplication
a = np.random.rand(size)
b = np.random.rand(size)
print(a.dtype)
```

```
float64
```

```
%%timeit
for _ in range(num_iterations):
    np_result = np.dot(a, b)/(np.linalg.norm(a)*np.linalg.norm(b)) # this line
```

```
2.18 s ± 160 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# NumPy Matrix Multiplication
a = np.random.rand(size).astype(np.float32)
b = np.random.rand(size).astype(np.float32)
print(a.dtype)
```

```
float32
```

```
%%timeit

for _ in range(num_iterations):
    np_result = np.dot(a, b)/(np.linalg.norm(a)*np.linalg.norm(b))
```

```
1.04 s ± 65.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# NumPy Matrix Multiplication
a = np.random.rand(size).astype(np.float16)
b = np.random.rand(size).astype(np.float16)
```

```
# %%timeit

# for _ in range(num_iterations):
#     np_result = np.dot(a, b)/(np.linalg.norm(a)*np.linalg.norm(b))
```

This example above probably gives you a numerical error. Because low precision computation can easily lead to underflow or overflow, triggering errors.

- Overflow: When numbers get too big for the computer to handle, they wrap around or behave unexpectedly.
- Underflow: When numbers get too small, they are rounded down to zero, losing important information.

Let's look at how `pytorch` does it.

```
# Set the size of the vectors to be created
size = 500000

# Set the number of iterations
num_iterations = 10000

# Specify the device for computation (GPU). You need this to enable GPU comput
device = 'mps'

# Create two random vectors (arrays) using PyTorch
a = torch.rand(size).to(device)
b = torch.rand(size).to(device)


print(a.dtype)
```

```
torch.float32
```

```
%%timeit

for _ in range(num_iterations):
    # Calculate the cosine similarity between vectors 'a' and 'b'
    torch_result = torch.dot(a, b) / (torch.norm(a) * torch.norm(b))
```

```
759 ms ± 4.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# PyTorch Matrix Multiplication
a = torch.rand(size).half().to(device)
b = torch.rand(size).half().to(device)
print(a.dtype)
```

```
torch.float16
```

```
%%timeit
for _ in range(num_iterations):
    torch_result = torch.dot(a, b)/(torch.norm(a)*torch.norm(b))
```

```
751 ms ± 1.54 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Using `float16`, aslo known as half precision, can reduce the computation time/memory by half! You might not see the speed up in this example. We need a problem of bigger scale to see the actual improvement. If you increase the size or the iterations by 100 times, you should see the speed up. Of course, a price we must pay is the reduced precision, which makes numerical underflow and overflow more frequent.

Intro to sparse matrices

```
import scipy.sparse
import networkx as nx
```

Sparse matrices are a class of matrices with most of its elements being 0. Let's initialize a sparse matrix.
(Here we are initializing the graph adjcency matrix, which we will cover in week 3 on graph data structure. Here you just need to know that it is a matrix with a lot of zeros in it.)

```
G = nx.ladder_graph(5)
# Return the Ladder graph of length n.
# This is two rows of n nodes, with each pair connected by a single edge.
am_ladder = nx.adjacency_matrix(G)
```

```
type(am_ladder) # Compressed Sparse Row
```

```
scipy.sparse._csr.csr_array
```

- Sparse matrices are a conceptual data structure like a list, dictionary, set, etc.
- `scipy.sparse` matrices are the standard Python implementation of this conceptual data structure, like `list`, `dict`, `set`, etc.
- Going to that link, we can see there are many types of scipy sparse matrix.
  - This one is a `csr_matrix`
  - More later on these types.
- You can convert them to numpy arrays with `toarray()`, but this is often a bad idea.

```
print(am_ladder.toarray())
```

```
[[0 1 0 0 0 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 0]
 [0 1 0 1 0 0 0 1 0 0]
 [0 0 1 0 1 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 1 0 1 0 0]
 [0 0 1 0 0 0 1 0 1 0]
 [0 0 0 1 0 0 0 1 0 1]
 [0 0 0 0 1 0 0 0 1 0]]
```

```
A = am_ladder.toarray() # undirect graph = symmetric matrix;

print(A)
print("---")
print(A.T)
```

```
[[0 1 0 0 0 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 0]
 [0 1 0 1 0 0 0 1 0 0]
 [0 0 1 0 1 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 1 0 1 0 0]
 [0 0 1 0 0 0 1 0 1 0]
 [0 0 0 1 0 0 0 1 0 1]
 [0 0 0 0 1 0 0 0 1 0]]
---
[[0 1 0 0 0 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 0]
 [0 1 0 1 0 0 0 1 0 0]
 [0 0 1 0 1 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 1 0 1 0 0]
 [0 0 1 0 0 0 1 0 1 0]
 [0 0 0 1 0 0 0 1 0 1]
 [0 0 0 0 1 0 0 0 1 0]]
```

```
# make a bigger sparse matrix
G = nx.fast_gnp_random_graph(100_000,1e-4)
am = nx.adjacency_matrix(G)
type(am)
```

```
scipy.sparse._csr.csr_array
```

```
am.shape # 10 billion
```

```
(100000, 100000)
```

```
am.nnz # non-zero elements of the matrix
```

```
998890
```

Stored in full form, the matrix would take up:

```
import numpy as np
```

```
%timeit np.sum(am, axis = 0) # column-wise sum
```

```
1.39 ms ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
# https://numpy.org/doc/stable/reference/generated/numpy.sum.html
print(np.sum([[0, 1], [2, 5]], axis=None))  # default

print(np.sum([[0, 1], [2, 5]], axis=1))     # row-wise sum
print(np.sum([[0, 1], [2, 5]], axis=0))     # column-wise sum

print(np.sum([[0, 1], [2, 5]], axis=-1))     # last axis ->  row
print(np.sum([[0, 1], [2, 5]], axis=-2))     # second last axis ->  column
```

```
8
[1 7]
[2 6]
[1 7]
[2 6]
```

```
import numpy as np

full_size = int(np.prod(am.shape))*8/(1e9)
print("The full matrix would take up %d GB" % full_size)
```

```
The full matrix would take up 80 GB
```

```
x = am.toarray()

print("Size of the array: ",
      x.size)

print("Memory size of one array element in bytes: ",
      x.itemsize)

print("Memory size of numpy array in bytes:",
      x.size * x.itemsize, "Bytes")

print("Memory size of numpy array in gigabytes:",
      (x.size * x.itemsize) / (1e9), "GB")                          # 1 byte = 1
```

```
Size of the array:  10000000000
Memory size of one array element in bytes:  8
Memory size of numpy array in bytes: 80000000000 Bytes
Memory size of numpy array in gigabytes: 80.0 GB
```

That's a lot! How big is the sparse matrix?

```
sparse_size = am.data.nbytes/1e6
print("The sparse matrix takes up %d NB" % sparse_size)
```

```
The sparse matrix takes up 7 NB
```

So, the fraction of space saved is:

```
frac_nz = am.nnz / np.prod(am.shape)
print("The sparse matrix is %dx smaller" % (1/frac_nz))
```

```
The sparse matrix is 10011x smaller
```

- Right, so we definitely don't want to store the full matrix.
- Regular numpy functions work on sparse matrices, although they might be fast/slow depending on various factors.

- You definitely do not want to iterate through the rows - make sure you use built-in numpy functions.

Sparse datasets

Sparse matrices come up *a lot* in practice. For example:

- Word counts: we might represent a document by which words are in it, but only a small fraction of all words would appear in a given document.
- Ratings: we might represent an Amazon item by the user ratings, but only a small fraction of all users have rated a given item.
- Physical processes: in a 2019 Capstone project, MDS students examined images from a particle physics dataset, in which most of the sensors got zero signal.
- etc.

Building sparse matrices

To create an empty sparse martrix, just provide the shape as the argument to the constructor

```
from scipy.sparse import csr_matrix,csc_matrix,lil_matrix
# csr: Compressed Sparse Row array
# csc: Compressed Sparse Column array
# lil: Row-based list of lists sparse array

shape = (10,10)
#my code here
x = csr_matrix(shape)
#my code here
x.shape
```

```
(10, 10)
```

However, this is a bad way to build a sparse matrix of any significant size. In fact, scipy will warn you that it's a very bad idea to try to populate a CSR sparse matrix directly

```
x[5,5] = 1
```

```
/Users/hedayatzarkoob/miniforge3/lib/python3.11/site-packages/scipy/sparse/_in
  self._set_intXint(row, col, x.flat[0])
```

A better way? Build the matrix in another sparse format, and then convert. Let build a diagonal matrix using the lil (linked list, i.e. the treasure boxes) format, which doesn't support efficient row or column operations like the csr and csc matrices, but does support efficient assignment.

```
#my code here
shape = (10,10)

x = lil_matrix(shape)
for i in range(shape[0]):
    x[i,i] = 1

x = csr_matrix(x)
print(x)
#my code here
```

```
<Compressed Sparse Row sparse matrix of dtype 'float64'
        with 10 stored elements and shape (10, 10)>
  Coords        Values
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
  (4, 4)        1.0
  (5, 5)        1.0
  (6, 6)        1.0
  (7, 7)        1.0
  (8, 8)        1.0
  (9, 9)        1.0
```

The Scikit learn package allows you to create csr sparse matrices from Python dictionaries (using DictVectorizer) and even directly from raw text strings (using CountVectorizer;).

```

```

```
from sklearn.feature_extraction import DictVectorizer

dv = DictVectorizer(sparse=False)
D1 = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
X1 = dv.fit_transform(D1)
print(dv.get_feature_names_out())
print(X1)
print(type(X1))
# print(dv.vocabulary_)
print("----")
```

```
['bar' 'baz' 'foo']
[[2. 0. 1.]
 [0. 1. 3.]]
<class 'numpy.ndarray'>
----
```

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer()
D2 = ['foo bar baz', 'bar foo', 'foo', 'bar', 'bar']
X2 = cv.fit_transform(D2)
print(cv.get_feature_names_out())
print(X2)
print(type(X2))
print(X2.toarray())
# print(cv.vocabulary_)
```

```
['bar' 'baz' 'foo']
<Compressed Sparse Row sparse matrix of dtype 'int64'
        with 8 stored elements and shape (5, 3)>
  Coords        Values
  (0, 2)        1
  (0, 0)        1
  (0, 1)        1
  (1, 2)        1
  (1, 0)        1
  (2, 2)        1
  (3, 0)        1
  (4, 0)        1
<class 'scipy.sparse._csr.csr_matrix'>
[[1 1 1]
 [1 0 1]
 [0 0 1]
 [1 0 0]
 [1 0 0]]
```

Optional example

```
# #pip install nltk
import nltk
nltk.download("brown")
```

```
[nltk_data] Downloading package brown to
[nltk_data]     /Users/hedayatzarkoob/nltk_data...
[nltk_data]   Package brown is already up-to-date!
```

```
True
```

```
from nltk.corpus import brown
```

Then, we apply the vectorizer to convert this to a csr matrix. First, initialize the vectorizer, then use its `fit_transform` method

`CountVectorizer()`

Convert a collection of text documents to a matrix of token counts.

This implementation produces a sparse representation of the counts using scipy.sparse.csr_matrix.

```
from sklearn.feature_extraction.text import CountVectorizer

count_vec = CountVectorizer()
count_vec_output = count_vec.fit_transform(brown.words('ca01'))

# print the identified unique words along with their indices
print(count_vec.vocabulary_['fulton'])
```

```
284
```

```
#print(count_vec_output)
print(type(count_vec_output))
print(count_vec_output.toarray())
```

```
<class 'scipy.sparse._csr.csr_matrix'>
[[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]]
```

Matrix vs. Array

- Note that when you do an operation which on sparse matrix that removes one of the dimensions (as we did above), you actually don't have a sparse matrix anymore.
- Surprisingly, you also don't have a numpy array!
- What you have a numpy matrix, which is distinct from an array.
  - See here for a discussion of the differences between arrays and matrices.
- Rule of thumb: (non-sparse) matrices will cause you headaches and you should stay away from them.
- You can access an array version of the matrix (without copying it) using `np.asarray`, and you should probably do that right away

```
import numpy as np
wpt_matrix = np.matrix(np.random.rand(15,15))
```

```
type(wpt_matrix)
```

```
numpy.matrix
```

```
wpt_array = np.asarray(wpt_matrix)
wpt_array[:10]
```

```
array([[0.06923231, 0.91851702, 0.37591892, 0.48076971, 0.7422663 ,
        0.96699858, 0.24414196, 0.32340762, 0.24895789, 0.30296409,
        0.7675246 , 0.89395855, 0.37799328, 0.74979643, 0.13844749],
       [0.3422048 , 0.49872977, 0.21432752, 0.51772241, 0.68127839,
        0.40091081, 0.70660313, 0.26479628, 0.30425981, 0.95524655,
        0.88995323, 0.26461024, 0.8264857 , 0.34009014, 0.34517586],
       [0.97020152, 0.24784675, 0.41440296, 0.24177035, 0.13898654,
        0.58526424, 0.95071219, 0.16084719, 0.62763114, 0.24894151,
        0.98697413, 0.64055135, 0.21057308, 0.05069404, 0.62251933],
       [0.06421394, 0.55711578, 0.47713244, 0.72443364, 0.43935553,
        0.10002106, 0.32347853, 0.39863408, 0.87219439, 0.93852426,
        0.34434528, 0.79452557, 0.26949563, 0.89001203, 0.69222248],
       [0.07298253, 0.93544381, 0.81526366, 0.93671325, 0.3410356 ,
        0.40413537, 0.25987851, 0.54887055, 0.86350517, 0.62090323,
        0.51656315, 0.21397415, 0.46642738, 0.81898199, 0.18945519],
       [0.37020914, 0.18655233, 0.75641224, 0.54300189, 0.53678969,
        0.19369649, 0.76889943, 0.54183373, 0.2905696 , 0.05602051,
        0.9707123 , 0.0311946 , 0.05936341, 0.15316907, 0.19877285],
       [0.77942794, 0.96530404, 0.21881053, 0.50758767, 0.18766242,
        0.16562541, 0.25018867, 0.95143982, 0.60329342, 0.00956355,
        0.33572899, 0.91329823, 0.35888087, 0.03780412, 0.46525093],
       [0.18600424, 0.42156861, 0.15573973, 0.95937243, 0.69205964,
        0.19180813, 0.15949234, 0.90511197, 0.86248775, 0.28810767,
        0.94011304, 0.065426  , 0.33331434, 0.71084201, 0.06827596],
       [0.79601779, 0.72970918, 0.74948884, 0.6168198 , 0.3520926 ,
        0.43609863, 0.1357824 , 0.42683319, 0.18589867, 0.00361196,
        0.78111018, 0.62216343, 0.46151955, 0.01479854, 0.56409479],
       [0.33375956, 0.75712608, 0.67368419, 0.46248264, 0.89842673,
        0.72687339, 0.05634948, 0.32491033, 0.43296749, 0.60256294,
        0.69605872, 0.86279083, 0.05119381, 0.21428457, 0.6969406 ]])
```

An example of where a numpy matrix will cause you trouble: operations like `flatten` (which is suppose to reduce a dataset to 1 dimension) won't work properly with a matrix, which is *always* 2-d

```
wpt_matrix.flatten()
```

```
matrix([[0.06923231, 0.91851702, 0.37591892, 0.48076971, 0.7422663 ,
         0.96699858, 0.24414196, 0.32340762, 0.24895789, 0.30296409,
         0.7675246 , 0.89395855, 0.37799328, 0.74979643, 0.13844749,
         0.3422048 , 0.49872977, 0.21432752, 0.51772241, 0.68127839,
         0.40091081, 0.70660313, 0.26479628, 0.30425981, 0.95524655,
         0.88995323, 0.26461024, 0.8264857 , 0.34009014, 0.34517586,
         0.97020152, 0.24784675, 0.41440296, 0.24177035, 0.13898654,
         0.58526424, 0.95071219, 0.16084719, 0.62763114, 0.24894151,
         0.98697413, 0.64055135, 0.21057308, 0.05069404, 0.62251933,
         0.06421394, 0.55711578, 0.47713244, 0.72443364, 0.43935553,
         0.10002106, 0.32347853, 0.39863408, 0.87219439, 0.93852426,
         0.34434528, 0.79452557, 0.26949563, 0.89001203, 0.69222248,
         0.07298253, 0.93544381, 0.81526366, 0.93671325, 0.3410356 ,
         0.40413537, 0.25987851, 0.54887055, 0.86350517, 0.62090323,
         0.51656315, 0.21397415, 0.46642738, 0.81898199, 0.18945519,
         0.37020914, 0.18655233, 0.75641224, 0.54300189, 0.53678969,
         0.19369649, 0.76889943, 0.54183373, 0.2905696 , 0.05602051,
         0.9707123 , 0.0311946 , 0.05936341, 0.15316907, 0.19877285,
         0.77942794, 0.96530404, 0.21881053, 0.50758767, 0.18766242,
         0.16562541, 0.25018867, 0.95143982, 0.60329342, 0.00956355,
         0.33572899, 0.91329823, 0.35888087, 0.03780412, 0.46525093,
         0.18600424, 0.42156861, 0.15573973, 0.95937243, 0.69205964,
         0.19180813, 0.15949234, 0.90511197, 0.86248775, 0.28810767,
         0.94011304, 0.065426  , 0.33331434, 0.71084201, 0.06827596,
         0.79601779, 0.72970918, 0.74948884, 0.6168198 , 0.3520926 ,
         0.43609863, 0.1357824 , 0.42683319, 0.18589867, 0.00361196,
         0.78111018, 0.62216343, 0.46151955, 0.01479854, 0.56409479,
         0.33375956, 0.75712608, 0.67368419, 0.46248264, 0.89842673,
         0.72687339, 0.05634948, 0.32491033, 0.43296749, 0.60256294,
         0.69605872, 0.86279083, 0.05119381, 0.21428457, 0.6969406 ,
         0.24305659, 0.8527706 , 0.11678966, 0.49563005, 0.88045105,
         0.86963136, 0.13846179, 0.78259563, 0.85878384, 0.0070805 ,
         0.3388426 , 0.54501452, 0.12454609, 0.49316088, 0.59059898,
         0.85840092, 0.49081242, 0.7259451 , 0.45315222, 0.56129137,
         0.48062699, 0.78491361, 0.26271677, 0.82937967, 0.71151765,
         0.0625318 , 0.90609356, 0.3307048 , 0.5001077 , 0.35565968,
         0.12086803, 0.68655967, 0.52317783, 0.16621209, 0.99701509,
         0.51957086, 0.7145493 , 0.97785203, 0.82931251, 0.30526969,
         0.15485402, 0.99227111, 0.8296711 , 0.01587383, 0.38745313,
         0.70700646, 0.03833806, 0.72663521, 0.67028374, 0.91222049,
         0.24956731, 0.79505933, 0.9189898 , 0.0361318 , 0.42807731,
         0.10285553, 0.76520361, 0.58620779, 0.07822476, 0.859025  ,
         0.76728033, 0.53862148, 0.13706987, 0.46528487, 0.88195847,
         0.35157164, 0.19156709, 0.23071585, 0.99137353, 0.73604717,
         0.11326871, 0.13739764, 0.70277018, 0.76862078, 0.577464  ]])
```

But this will work fine with the array version

```
wpt_array.flatten()
```

```
array([0.06923231, 0.91851702, 0.37591892, 0.48076971, 0.7422663 ,
       0.96699858, 0.24414196, 0.32340762, 0.24895789, 0.30296409,
       0.7675246 , 0.89395855, 0.37799328, 0.74979643, 0.13844749,
       0.3422048 , 0.49872977, 0.21432752, 0.51772241, 0.68127839,
       0.40091081, 0.70660313, 0.26479628, 0.30425981, 0.95524655,
       0.88995323, 0.26461024, 0.8264857 , 0.34009014, 0.34517586,
       0.97020152, 0.24784675, 0.41440296, 0.24177035, 0.13898654,
       0.58526424, 0.95071219, 0.16084719, 0.62763114, 0.24894151,
       0.98697413, 0.64055135, 0.21057308, 0.05069404, 0.62251933,
       0.06421394, 0.55711578, 0.47713244, 0.72443364, 0.43935553,
       0.10002106, 0.32347853, 0.39863408, 0.87219439, 0.93852426,
       0.34434528, 0.79452557, 0.26949563, 0.89001203, 0.69222248,
       0.07298253, 0.93544381, 0.81526366, 0.93671325, 0.3410356 ,
       0.40413537, 0.25987851, 0.54887055, 0.86350517, 0.62090323,
       0.51656315, 0.21397415, 0.46642738, 0.81898199, 0.18945519,
       0.37020914, 0.18655233, 0.75641224, 0.54300189, 0.53678969,
       0.19369649, 0.76889943, 0.54183373, 0.2905696 , 0.05602051,
       0.9707123 , 0.0311946 , 0.05936341, 0.15316907, 0.19877285,
       0.77942794, 0.96530404, 0.21881053, 0.50758767, 0.18766242,
       0.16562541, 0.25018867, 0.95143982, 0.60329342, 0.00956355,
       0.33572899, 0.91329823, 0.35888087, 0.03780412, 0.46525093,
       0.18600424, 0.42156861, 0.15573973, 0.95937243, 0.69205964,
       0.19180813, 0.15949234, 0.90511197, 0.86248775, 0.28810767,
       0.94011304, 0.065426  , 0.33331434, 0.71084201, 0.06827596,
       0.79601779, 0.72970918, 0.74948884, 0.6168198 , 0.3520926 ,
       0.43609863, 0.1357824 , 0.42683319, 0.18589867, 0.00361196,
       0.78111018, 0.62216343, 0.46151955, 0.01479854, 0.56409479,
       0.33375956, 0.75712608, 0.67368419, 0.46248264, 0.89842673,
       0.72687339, 0.05634948, 0.32491033, 0.43296749, 0.60256294,
       0.69605872, 0.86279083, 0.05119381, 0.21428457, 0.6969406 ,
       0.24305659, 0.8527706 , 0.11678966, 0.49563005, 0.88045105,
       0.86963136, 0.13846179, 0.78259563, 0.85878384, 0.0070805 ,
       0.3388426 , 0.54501452, 0.12454609, 0.49316088, 0.59059898,
       0.85840092, 0.49081242, 0.7259451 , 0.45315222, 0.56129137,
       0.48062699, 0.78491361, 0.26271677, 0.82937967, 0.71151765,
       0.0625318 , 0.90609356, 0.3307048 , 0.5001077 , 0.35565968,
       0.12086803, 0.68655967, 0.52317783, 0.16621209, 0.99701509,
       0.51957086, 0.7145493 , 0.97785203, 0.82931251, 0.30526969,
       0.15485402, 0.99227111, 0.8296711 , 0.01587383, 0.38745313,
       0.70700646, 0.03833806, 0.72663521, 0.67028374, 0.91222049,
       0.24956731, 0.79505933, 0.9189898 , 0.0361318 , 0.42807731,
       0.10285553, 0.76520361, 0.58620779, 0.07822476, 0.859025  ,
       0.76728033, 0.53862148, 0.13706987, 0.46528487, 0.88195847,
       0.35157164, 0.19156709, 0.23071585, 0.99137353, 0.73604717,
       0.11326871, 0.13739764, 0.70277018, 0.76862078, 0.577464  ])
```

Indexing

Another issue with sparse matrices is indexing. With a regular numpy array, `x[i,j]` and `x[i][j]` are equivalent:

```
x = np.random.rand(10,10)
```

```
x[1,2]
```

```
0.41118731521740703
```

```
x[1][2]
```

```
0.41118731521740703
```

This is because `x[1]` returns the first row, and then the `[2]` indexes into that row:

```
x[1]
```

```
array([0.48802204, 0.8551031 , 0.41118732, 0.12974941, 0.84005109,
       0.4861311 , 0.56041248, 0.62394123, 0.62505716, 0.33407994])
```

```
row_1 = x[1]
row_1[2]
```

```
0.41118731521740703
```

However, with `scipy.sparse` matrices, things are a bit different:

```
x_sparse = csr_matrix(x)
```

```
x_sparse[1,2]
```

```
0.41118731521740703
```

```
```

```
x_sparse[1][2]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[63], line 1
----> 1 x_sparse[1][2]

File ~/miniforge3/lib/python3.11/site-packages/scipy/sparse/_csr.py:24, in _cs
     22 def __getitem__(self, key):
     23     if self.ndim == 2:
---> 24         return super().__getitem__(key)
     26     if isinstance(key, tuple) and len(key) == 1:
     27         key = key[0]

File ~/miniforge3/lib/python3.11/site-packages/scipy/sparse/_index.py:52, in In
     51 def __getitem__(self, key):
---> 52     row, col = self._validate_indices(key)
     54     # Dispatch to specialized methods.
     55     if isinstance(row, INT_TYPES):

File ~/miniforge3/lib/python3.11/site-packages/scipy/sparse/_index.py:180, in I
    178 row = int(row)
    179 if row < -M or row >= M:
--> 180     raise IndexError('row index (%d) out of range' % row)
    181 if row < 0:
    182     row += M

IndexError: row index (2) out of range
```

Why?

```
row_1_sparse = x_sparse[1]
```

```
row_1_sparse.shape
```

```
(1, 10)
```

- The sparse matrix returns a different shape.
- This is because sparse matrices must always be 2d (same problem as non-sparse matrices)
- In general, use the `x[1,2]` notation when possible because chaining the `[]` can be problematic in several places (e.g., also pandas).

- However, **this is only for numpy**, not, say, a list of lists:

```
lst = [[1,2,3],[4,5,6],[7,9]]
```

```
lst[0][1]
```

```
2
```

```
lst[0,1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[68], line 1
----> 1 lst[0,1]

TypeError: list indices must be integers or slices, not tuple
```

Rows versus columns operations

Generally, you should avoid looping directly through sparse matrices.

But if you do need to, you need to be very sensitive to the difference between Row (CSR) and Column (CSC) sparse matricies.

Note. CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) are two common formats for storing sparse matrices. Here's a breakdown of the differences:

Feature	CSR	CSC
Storage Arrays	Values, Column Indices, Row Pointers	Values, Row Indices, Column Pointers
Optimized For	Row operations	Column operations
Access Pattern	Fast row slicing	Fast column slicing
Use Cases	Matrix-vector multiplication, row-wise operations	Column-wise computations, matrix factorization

You should not use CSR matrices for operations which require looping over columns. Observe:

```
am = scipy.sparse.random(10000, 10000, density=0.2, format="csr", random_state
```

```
num_rows, num_cols = am.shape
```

```
am
```

```
<Compressed Sparse Row sparse matrix of dtype 'float64'
        with 20000000 stored elements and shape (10000, 10000)>
```

```
type(am)
```

```
scipy.sparse._csr.csr_matrix
```

```
%%timeit
(np.sum(am, axis = 0))
```

```
8.48 ms ± 59.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
(np.sum(am, axis = 1))
```

```
2.8 ms ± 61 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit -r1 -n1

# iterate through the rows - this is much slower than numpy functions
for i in range(num_rows):
    am[i,:]
```

```
222 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
# %%timeit -r1 -n1

# # iterate through the columns - this is much MUCH slower
# for i in range(num_cols):
#     am[:,i]
```

Above:

- Looping is slower, as per usual.
- But at least looping through the rows of a `csr_matrix` isn't that bad.
- However, looping through the columns of a `csr_matrix` is a disaster - it took several min on my laptop!!
  - Because it is stored *row by row*.
  - To grab a single column, it needs to loop through each row and look for items in that column.
- If you want to loop through columns you should first convert the matrix type.

```
am_csc = am.tocsc()
```

```
%%timeit -r1 -n1

for i in range(num_cols):
    am_csc[:,i]
```

```
214 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Optional: Einstein Summation (einsum)

The Einstein summation convention is the ultimate generalization of products such as matrix multiplication to multiple dimensions. It offers a compact and elegant way of specifying almost any product of scalars/vectors/matrices/tensors. Despite its generality, it can reduce the number of errors made by computer scientists and reduce the time they spend reasoning about linear algebra. It does so by being simultaneously clearer, more explicit, more self-documenting, more declarative in style and less cognitively burdensome to use. Its advantages over such things as matrix multiplication are that it liberates its user from having to think about:

- The correct order in which to supply the argument tensors
- The correct transpositions to apply to the argument tensors
- nsuring that the correct tensor dimensions are lined up with one another
- The correct transposition to apply to the resulting tensor

The only things it requires are knowledge of:

- Along which dimensions to compute (inner/element-wise/outer) products.
- The desired output shape.

EINSUM IS ALL YOU NEED - EINSTEIN SUMMATION IN DEEP LEARNING