Tensors
Thursday, 23 October 2025 5:24 AM

"Tom strongly! Stay strong with Sigmoid"

Topics

→ What are tensors?
→ How they are diff. from lists etc.
→ Indexing, Broadcasting.
→ Storage, Clone → Make the tensor.
→ Strides, offsets, views.
→ Tensor data types.
→ Contiguous tensors.

Next:
→ Tensor API.
→ Various types of tensors.
→ Pytorch, Distribution.

Optional tensors
→ Sparse, etc.
→ Interoperability of tensor with Numpy.
→ Serialization.
→ NJIT → (Numpy)

Q:
1. Why create tensor tensors? (Not-me, Numpy's array.)

NOTES:
① Functions that contain _ (underscore), modify storage in place for tensor.
→ Example: = relate output tensor.
→ Example: .T → in place.

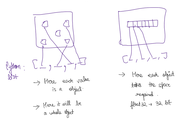
More for Side of Curiosity.
② The interoperability interoperability with Numpy array is possible because the storage system works with the python buffer protocol.

1 Tensors

→ Tensors are fundamental data structure used in Pytorch and its core representation.

import torch
points = torch.tensor([4.0, 1.0, 5.0])
print(points)
[1.] → torch.Size([3,])

→ The stored data of a single type in contiguous memory locations



→ Here, each value is a object
→ Here, it will be a single object

1.1 Other methods to create tensors
① torch.ones(3,2) → all ones
② torch.zeros(3,2) → " " " 0
③ torch.from_numpy() → from a numpy array
④ torch.arange() → from a list?

. rand →
. random →

Other Important
① dtype
② device

1.2 Tensor dtypes
① float32 or float
② int64 → short
③ float16 or half precision
④ bfloat16 → float16 but more stable training
⑤ Int → int8, 32, 64

2 Indexing & Broadcasting

2.1 Indexing

→ This is basically accessing the tensor as part of tensor.

p = [[1, 2, 3],
[4, 5, 6],
[7, 8, 9]]

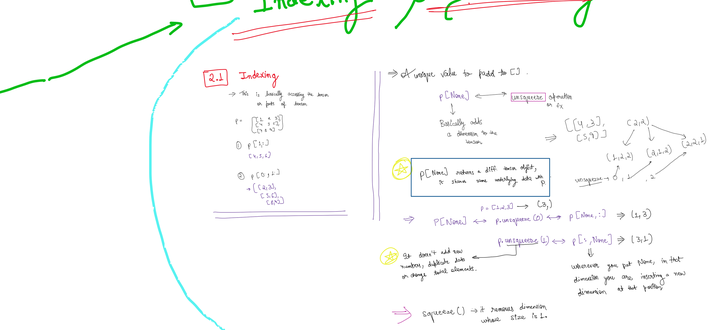
① p[1,1]
[5,]

② p[0:2, 1:]
= [[2,3],
[5,6]]

→ Of unique value to feed to [ ].

p[None] ⟷ unsqueeze of numpy on 0th
↓
Basically adds a dimension to the tensor.

⇒ [[4, 5, 6],
[7, 8, 9]]



p[None] makes a diff. tensor object, it stores same underlying data as p.

p = [1,2,3] → [3,]
⇒ p[None] ⟷ p.unsqueeze(0) ⟷ p[None,:] ⇒ [1,3]
p.unsqueeze(1) ⟷ p[:,None] ⇒ [3,1]

Its doesn't add new numbers, difference lies in storage, based elements.

unsqueeze, you put none, in that dimension you are inserting a new dimension, at that position.

⇒ squeeze() → it removes dimension without size 1.

2.2 Broadcasting

→ Why Broadcasting in order to perform efficient ops, tensor's dimension can be extended to match ops.

Rules
(1) Two dimensions are same.
(2) One of the dimension is 1.

Example:
w = torch.tensor([10.0, 20.0, 30.0])
→ w.shape → [3]
p = torch.tensor([[1.0, 2.0, 3.0],
[4.0, 5.0, 6.0]])
→ p.shape → [2,3]

Q. Can we multiply add w to p?
w.shape, p.shape
↓
Yes → because
w [10, 20, 30] → [[10, 20, 30], [10, 20, 30]]
(3,) (2,3)

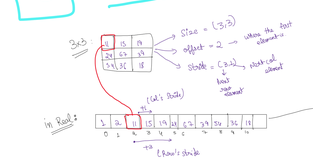
ans = p × w[None,:] = [[10, 20, 30], [40, 50, 60]]
(2,3) × (1,3) (2,3)

It will multiply row wise for all rows

optional: Because pytorch can broadcast

⇒ [[10, 20, 30], [40, 50, 60]] × [0.1, 0.2, 0.3] = [[10×0.1, 20×0.2, 30×0.3], [40×0.1, 50×0.2, 60×0.3]]

3 Under the Hood



3.1 Storage Class & Other concepts

⇒ Tensors are just a view of storage class. Values are allocated in contiguous chunks of memory. (Managed by torch.Storage instance)
→ Values in tensor are stored in a 1-D contiguous array in memory.
data type: (numpy.dtype)

p = torch.tensor([[4, 1],
[5, 3],
[6, 13]])

p.storage()
↓
[4,
1,
5,
3,
6,
13]
[torch.storage.TypedStorage(dtype=__, device=__) of size 6]

⇒ We can access the storage that can be indexed as well and changing it will obviously change the tensor.
⇒ x = p.storage()
x[0] ⇒ 5

why?? → We can create different views without equally responsible of the memory allocation done once.

Now, How we index 2D arrays?
↓
By using trace → ① size → tuple indicating the no. of element across each dimension of tensor.
② offset → Index in storage corresponding to 1st element of tensor.
③ stride → A tuple that tells no. of elements to be skipped over to obtain next one in each dimension. index is increased by one in each dimension.

⇒ For a 2D tensor,
A[i,j] = t[position]
position = offset + (stride[0]*i) + (stride[1]*j)
Formula: stride[0] × index[0] + offset

3.2 Contiguous Tensors.

⇒ A tensor values whose values are both out storage starting from neighbour dimension around is defined as contiguous tensor.

ex → Simply [[4,1], [5,3]] moving along row 2 ⇒ [4,1] [5,5]

⇒ Some operations can only work on contiguous tensor. → To check use .is_contiguous()
↓
If it is not, a exception will set to use .contiguous()

In our example,
p (original) → contiguous ✓
p.t (transpose) → ✗
p.t = [[4,5], [1,3]] & storage = [4, 5, 1, 3]
p.t.contiguous() → [4, 5, 1, 3]

3.3 Typed & Untyped Storage

(later)

Note:
Sets p = [[1,2], [3,5], [6,8]] → same storage.
p.t = p[:2] ⇒ [[1,3]]
p.t[:2] → p[[1,2], [3,5], [6,8]]
↓ to avoid
p.2 = p[:2].clone()

Extra Concepts

1. View
→ So basically this underlying data is same, just one different view.
→ changing one affects storage. (Smallest copy, no copy or different)

2. Copy.
→ Independent underlying data.
→ Torch copy → .clone()

⇒ These 2 concepts affects:
1. Memory Usage.
2. Automatic differentiation.

→ In pytorch, op will either return a new tensor or modify the tensor in place.

How we know if it is copy or view?
One way is to check with code, let's say t = one tensor.
⇒ p.untyped_storage().data_ptr() == v.
If True, then they have same underlying data.

Example:
id(p.storage()) == id(v.storage())

Example: Transpose op^n

2x3:
p = [[1, 3], [5, 7]] → Storage view → [1, 3, 5, 7]
size = (2,2)
stride = (2,1)
offset = 0

⇒ Transpose
t(t) → Trace for transposing 2d.
p.t = p.t()
↓
[[1, 5]
[3, 7]]
Now for different op^n, we don't change tensor in storage or memory.
→ size is same
⇒ stride is now → (1,2)