An open source machine learning framework that accelerates the path from research prototyping to production deployment.

Tensor - Pytorch's core data structure

In Python we can create lists, lists of lists, lists of lists and so on. In NumPy there is a `numpy.ndarray` which represents `n`- dimensional array. In math there is a special name for the generalization of vectors and matrices to a higher dimensional space - a tensor

Tensor is an entity with a defined number of dimensions called an order (rank).

Scalar can be considered as a rank-0-tensor.

Vector can be introduced as a rank-1-tensor.

Matrices can be considered as a rank-2-tensor.

Tensor Basics

Let's import the torch module first.

```
import numpy as np
import torch
```

Tensor Creation

Let's view examples of matrices and tensors generation

2-dimensional (rank-2) tensor of zeros:

```
torch.zeros(3, 4)
```

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

Random rank-3 tensor: *read the print below and convince yourself how this is a rank-3-tensor and learn what those 2, 3, 4 values are there for*

```
torch.rand(2, 3, 4)
```

```
tensor([[[0.0026, 0.4589, 0.8341, 0.8561],
         [0.5651, 0.2986, 0.4709, 0.7135],
         [0.9157, 0.8527, 0.5451, 0.4681]],

        [[0.6614, 0.4935, 0.1957, 0.4923],
         [0.3459, 0.1021, 0.6446, 0.3094],
         [0.9342, 0.0864, 0.9096, 0.7017]]])
```

I am hoping you have noticed 4-elements in a row, 3 rows making one block and there are 2 blocks.

Random rank-4-tensor:

```
torch.rand(2, 2, 2, 3)
```

```
tensor([[[[0.3538, 0.2852, 0.8185],
          [0.0380, 0.0413, 0.4717]],

         [[0.7822, 0.5089, 0.7497],
          [0.6415, 0.7038, 0.3637]]],
```

```
        [[[0.7178, 0.5782, 0.2475],
          [0.2320, 0.7770, 0.5674]],

         [[0.1770, 0.9911, 0.3948],
          [0.1702, 0.2399, 0.6749]]]])
```

Question 1:

How many dimensions are there in a tensor defined as below?

```
tensor_one = torch.rand(1, 1, 1, 1)
print(tensor_one)
tensor_one.dim()
```

```
tensor([[[[0.1591]]]])
```

```
4
```

.

There are many more ways to create tensor using some restrictions on values it should contatn - for the full reference, please follow the official docs.

•

Python / NumPy / Pytorch interoperability

You can create tensors from python as well as numpy arrays. You can also convert torch tensors to numpy arrays. So, the interoperability between torch and numpy is pretty good.

```
# Simple Python List
python_list = [1, 2]

# Create a numpy array from python list
numpy_array = np.array(python_list)

# Create a torch Tensor from python list
tensor_from_list = torch.tensor(python_list)

# Create a torch Tensor from Numpy array
tensor_from_array = torch.tensor(numpy_array)

# Another way to create a torch Tensor from Numpy array (share same storage)
tensor_from_array_v2 = torch.from_numpy(numpy_array)

# Convert torch tensor to numpy array
array_from_tensor = tensor_from_array.numpy()

print('List:   ', python_list)
print('Array:  ', numpy_array)
print('Tensor: ', tensor_from_list)
print('Tensor: ', tensor_from_array)
print('Tensor: ', tensor_from_array_v2)
print('Array:  ', array_from_tensor)
```

```
List:    [1, 2]
Array:   [1 2]
Tensor:  tensor([1, 2])
Tensor:  tensor([1, 2])
Tensor:  tensor([1, 2])
Array:   [1 2]
```

Difference between `torch.Tensor` and `torch.from_numpy`

Pytorch aims to be an effective library for computations. What does it mean? It means that pytorch avoids memory copying if it can.

```
numpy_array[0] = 10

print('Array:  ', numpy_array)
print('Tensor: ', tensor_from_array)
print('Tensor: ', tensor_from_array_v2)
```

```
Array:   [10  2]
Tensor:  tensor([1, 2])
Tensor:  tensor([10,  2])
```

Question 2:

Assume that we moved our complete (cats vs dogs) image dataset to numpy arrays. Then we use torch.from_numpy to convert these images to tensor. Then we apply a specific data augmentation strategy called "CutOut" which blocks a portion of the image directly on these tensors. What will happen to the accuracy of a model trained on this strategy compared to the one without this strategy? CutOut strategy is shown below:

Question 3:

Why do you think we are observing this behavior?

We have two different ways to create tensor from its NumPy counterpart - one copies memory and another one shares the same underlying storage. It works in the opposite way:

```
array_from_tensor = tensor_from_array.numpy()
print('Tensor: ', tensor_from_array)
print('Array: ', array_from_tensor)

tensor_from_array[0] = 11
print('Tensor: ', tensor_from_array)
print('Array: ', array_from_tensor)
```

```
Tensor:  tensor([1, 2])
Array:  [1 2]
Tensor:  tensor([11,  2])
Array:  [11  2]
```

Data types

The basic data type of all Deep Learning-related operations is float, but sometimes you may need something else. Pytorch support different number types for its tensors the same way NumPy does it - by specifying the data type on tensor creation or via casting. Ths full list of supported data types can be found here.

```
tensor = torch.zeros(2, 2)
print('Tensor with default type: ', tensor)
tensor = torch.zeros(2, 2, dtype=torch.float16)
print('Tensor with 16-bit float: ', tensor)
tensor = torch.zeros(2, 2, dtype=torch.int16)
print('Tensor with integers: ', tensor)
tensor = torch.zeros(2, 2, dtype=torch.bool)
print('Tensor with boolean data: ', tensor)
```

```
Tensor with default type:  tensor([[0., 0.],
        [0., 0.]])
Tensor with 16-bit float:  tensor([[0., 0.],
        [0., 0.]], dtype=torch.float16)
Tensor with integers:  tensor([[0, 0],
        [0, 0]], dtype=torch.int16)
Tensor with boolean data:  tensor([[False, False],
        [False, False]], dtype=torch.bool)
```

```
tensor = torch.ones(2, 2, dtype=torch.float16)
print(tensor)
num = tensor.numpy()
print(num)
num[:] = num * 0.5
print(num)
print(tensor)
```

```
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float16)
[[1. 1.]
 [1. 1.]]
[[0.5 0.5]
 [0.5 0.5]]
tensor([[0.5000, 0.5000],
        [0.5000, 0.5000]], dtype=torch.float16)
```

Question 4:

We saw above that some times numpy and tensors share same storage and changing one changes the other. If we define a rank-2-tensor with ones (dtype of f16), and then convert it into a numpy data type using tensor.numpy() and store it in a variable called "num", and then we perform this operation `num = num * 0.5`, will the original tensor have 1.0s or 0.5s as its element

values?

Question 5:

If the operation `num = num*5` is changed to `num[:] = num*5` will the original tensor have 1.0s or 0.5s as its element values?

Indexing

Tensor provides access to its elements via the same `[]` operation as a regular python list or NumPy array. However, as you may recall from NumPy usage, the full power of math libraries is accessible only via vectorized operations, i.e. operations without explicit looping over all vector elements in python and using implicit optimized loops in C/C++/CUDA/Fortran/etc. available via special function calls. Pytorch employs the same paradigm and provides a wide range of vectorized operations. Let's take a look at some examples.

Joining a list of tensors together with `torch.cat`

```
a = torch.zeros(3, 2)
b = torch.ones(3, 2)
print(torch.cat((a, b), dim=0))

c=torch.cat((a, b), dim=0)
d=torch.t(c)
print(d)
```

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

Question 6:

Is the transpose of concatenated a & b tensor on dimension 1, same as the contatenated tensor of a & b on dimension 0?

Indexing with another tenxor/array:

```
a = torch.arange(start=0, end=10)
indices = np.arange(0, 10) > 5
print(a)
print(indices)
print(a[indices])

indices = torch.arange(start=0, end=10) %5
print(indices)
print(a[indices])
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
[False False False False False False  True  True  True  True]
tensor([6, 7, 8, 9])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

```
a = torch.arange(start=0, end=10)
indices = np.arange(0, 10) % 5
print(a)
print(indices)
print(a[:-5])
```

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

```
a = torch.arange(start=0, end=10)
indices = np.arange(1, 11) % 5
print(a)
print(indices)
print(a[:-5])
```

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

Question 7:

`a` is defined as `torch.arange(start=0, end=10)`. We will create `b` using the two operations as below. In both cases do we get the same value?

1. indices variable created by the modulo operation on arange between 0 and 10. Then a new varialble `b` is created from `a` using the last 5 elements of indices.

2. indices variable created by the modulo operation on arange betwenn 1 and 11. Then a new varialble `b` is created from `a` using the last 5 elements of indices.

What should we do if we have, say, rank-2-tensor and want to select only some rows?

```
tensor = torch.rand((5, 3))
rows = torch.tensor([0, 2])
print(tensor)
tensor[rows]
```

```
tensor([[0.6657, 0.7041, 0.4038],
        [0.1543, 0.4606, 0.1103],
        [0.6704, 0.6282, 0.1762],
        [0.7504, 0.6413, 0.5980],
        [0.0452, 0.2649, 0.7961]])
```

```
tensor([[0.6657, 0.7041, 0.4038],
        [0.6704, 0.6282, 0.1762]])
```

```
tensor = torch.rand((6,5))
print(tensor)
print(tensor.size())
print("***********")
newTensor1 = tensor[0,2,4]
print(newTensor1)
```

```
tensor([[0.9094, 0.6931, 0.4230, 0.5800, 0.7722],
        [0.8639, 0.2755, 0.3595, 0.7562, 0.4170],
        [0.4675, 0.0073, 0.0222, 0.6935, 0.4776],
        [0.6998, 0.7483, 0.7621, 0.4270, 0.8196],
        [0.0417, 0.7587, 0.3187, 0.3180, 0.7327],
        [0.3420, 0.9875, 0.0532, 0.0131, 0.9042]])
torch.Size([6, 5])
***********
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-41-efbb8bb4dca9> in <module>()
      3 print(tensor.size())
      4 print("***********")
----> 5 newTensor1 = tensor[0,2,4]
      6 print(newTensor1)
```

```
IndexError: too many indices for tensor of dimension 2
```

Question 8:

Consider a tensor defined as `torch.rand((6, 5))`. Is the shape of the new tensor created by taking the 0th, 2nd and 4th row of the old tensor same as the shape of the a newer tensor created by taking the 0th, 2nd and 4th row of the old tensor after transposing it by operation `torch.transpose(tensor, 0, 1)` ?

Tensor Shapes

Reshaping a tensor is a frequently used operation. We can change the shape of a tensor without the memory copying overhead. There are two methods for that: `reshape` and `view`.

The difference is the following:

- view tries to return a tensor, and it shares the same memory with the original tensor. In case, if it cannot reuse the same memory due to some reason, it just fails.
- reshape always returns the tensor with the desired shape and tries to reuse the memory. If it cannot, it creates a copy

Let's see with the help of an example:

```
tensor = torch.rand(2, 3, 4)
print('Pointer to data: ', tensor.data_ptr())
print('Shape: ', tensor.shape)

reshaped = tensor.reshape(24)

view = tensor.view(3, 2, 4)
print('Reshaped tensor - pointer to data', reshaped.data_ptr())
print('Reshaped tensor shape ', reshaped.shape)

print('Viewed tensor - pointer to data', view.data_ptr())
print('Viewed tensor shape ', view.shape)

assert tensor.data_ptr() == view.data_ptr()
```

```
assert np.all(np.equal(tensor.numpy().flat, reshaped.numpy().flat))

print('Original stride: ', tensor.stride())
print('Reshaped stride: ', reshaped.stride())
print('Viewed stride: ', view.stride())
```

```
Pointer to data:  1922140530944
Shape:  torch.Size([2, 3, 4])
Reshaped tensor - pointer to data 1922140530944
Reshaped tensor shape  torch.Size([24])
Viewed tensor - pointer to data 1922140530944
Viewed tensor shape  torch.Size([3, 2, 4])
Original stride:  (12, 4, 1)
Reshaped stride:  (1,)
Viewed stride:  (8, 4, 1)
```

The basic rule about reshaping the tensor is definitely that you cannot change the total number of elements in it, so the product of all tensor's dimensions should always be the same. It gives us the ability to avoid specifying one dimension when reshaping the tensor - Pytorch can calculate it for us:

```
print(tensor.reshape(3, 2, 4).shape)
print(tensor.reshape(3, 2, -1).shape)
print(tensor.reshape(3, -1, 4).shape)
```

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

Question 9:

Consider a tensor `a` created with [1, 2, 3] and [1, 2, 3] of size (2, 3) is reshaped with operation `.reshape(-1, 2)`. Also consider a tensor `b` created with [[2, 1]] and of size (1, 2), later operated with `view(2, -1)` operation.

If we do a dot product of a and b (using `torch.mm`) and perform the sum of all the elements (using `torch.sum`) what do we get? (enter int value without any decimal point in the quiz)

```
a = torch.tensor([[1,2,3],[1,2,3]])
print(a)
print(a.size())
a = a.reshape(-1, 2)
print(a)
print(a.size())
```

```
print("****")

b = torch.tensor([[2, 1]])
print(b)
print(b.size())
b = b.view(2, -1)
print(b)
print(b.size())

print("****")
c = torch.mm(a,b)
print(c)
d = torch.sum(c)
print(d)
```

```
tensor([[1, 2, 3],
        [1, 2, 3]])
torch.Size([2, 3])
tensor([[1, 2],
        [3, 1],
        [2, 3]])
torch.Size([3, 2])
****
tensor([[2, 1]])
torch.Size([1, 2])
tensor([[2],
        [1]])
torch.Size([2, 1])
****
tensor([[4],
        [7],
        [7]])
tensor(18)
```

Alternative ways to view tensors - `expand` or `expand_as`.

- `expand` - requires the desired shape as an input
- `expand_as` - uses the shape of another tensor

These operations "repeat" tensor's values along the specified axes without actually copying the data.

As the documentation says, expand:

> returns a new view of the self tensor with singleton dimensions expanded to a larger size. Tensor can be also expanded to a larger number of dimensions, and the new ones will be appended at the front. For the new dimensions, the size cannot be set to -1.

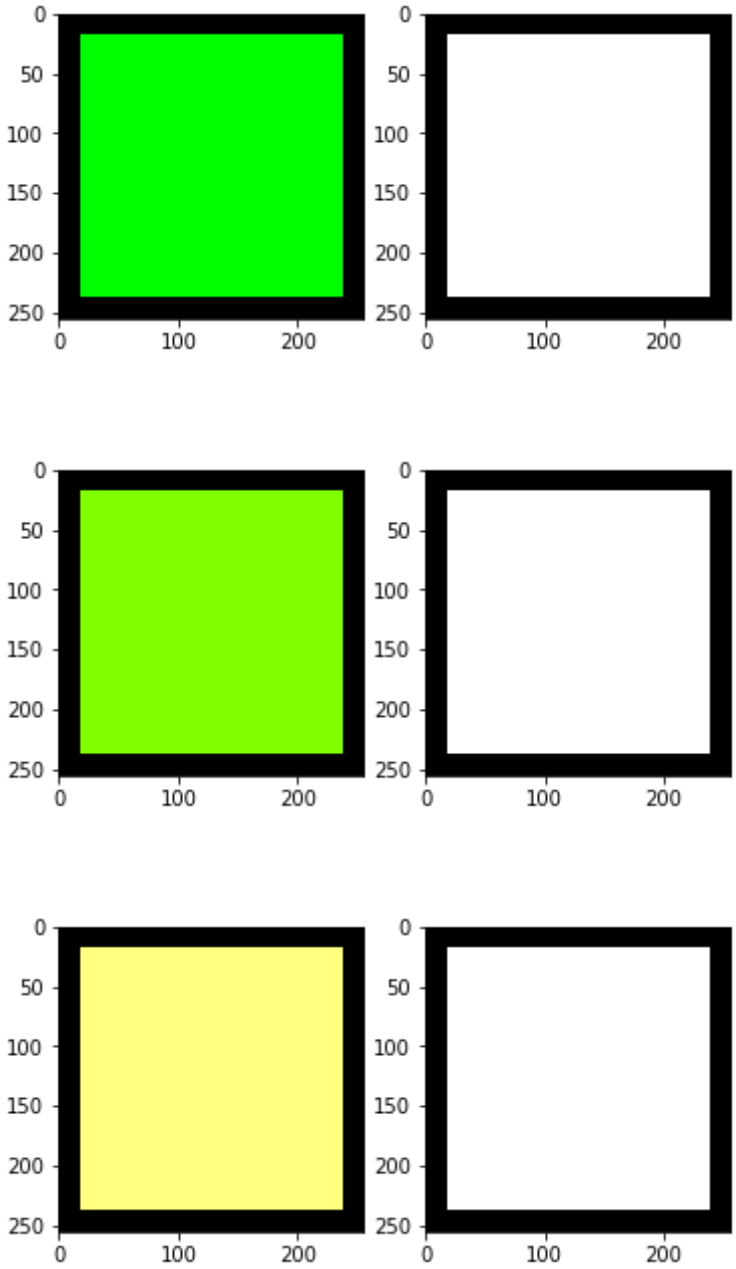
Use case:

- index multi-channel tensor with single-channel mask - imagine a color image with 3 channels (RGB) and binary mask for the area of interest on that image. We cannot index the image with this kind of mask directly since the dimensions are different, but we can use

`expand_as` operation to create a view of the mask that has the same dimensions as the image we want to apply it to, but has not copied the data.

```
%matplotlib inline
from matplotlib import pyplot as plt

# Create a black image
image = torch.zeros(size=(3, 256, 256), dtype=torch.int)

# Leave the borders and make the rest of the image Green
image[1, 18:256 - 18, 18:256 - 18] = 255

# Create a mask of the same size
mask = torch.zeros(size=(256, 256), dtype=torch.bool)

# Assuming the green region in the original image is the Region of interest,
change the mask to white for that area
mask[18:256 - 18, 18:256 - 18] = 1

# Create a view of the mask with the same dimensions as the original image
mask_expanded = mask.expand_as(image)
print(mask_expanded.shape)

mask_np = mask_expanded.numpy().transpose(1, 2, 0) * 255
image_np = image.numpy().transpose(1, 2, 0)

fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()

image[0, mask] += 128
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()

image[mask_expanded] += 128
image.clamp_(0, 255)
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()
```

```
torch.Size([3, 256, 256])
```



In the example above, one can also find a couple of useful tricks:

- `clamp` method and function is a Pytorch's analogue of NumPy's `clip` function
- many operations on tensors have in-place form, that does not return modified data, but change values in the tensor. The in-place version of the operation has trailing underscore according to Pytorch's naming convension - in the exmaple above it is `clamp_`
- tensors have the same indexing as Numpy's arrays - one can use `:` seperated range, negative indexes and so on.

.

Images and their representations

Now, let's discuss images, their representations and how different Python librarties work with them.

Probably, the most well-known library for image loading and simple processing is Pillow.

However, many people in deep learning area stick with OpenCV for image loading and processing with some usage of another libraries when it is justified by performance/functionality. This is because OpenCV is in general much faster than the other libraries. Here you can find a couple of benchmarks:

- https://www.kaggle.com/zfturbo/benchmark-2019-speed-of-image-reading
- https://github.com/albumentations-team/albumentations#benchmarking-results

To sum up the benchmarks above, there are two most common image formats, PNG and JPEGs. If your data is in PNG format - use OpenCV to read it. If it is in JPEG - use libturbojpeg. For image processing, use OpenCV if possible. *We will be using PIL a lot along with these.*

As you will read the code from others, you may find out that some of them use Pillow/something else to read data. You should know, that color image representations in OpenCV and other libraries are different - OpenCV uses "BGR" channel order, while others use "RGB" one.

To change "BRG" <-> "RGB" the only thing we need to do it to change channel order.

```
%matplotlib inline
from matplotlib import pyplot as plt
import cv2


bgr_image = cv2.imread('mars.jpg')
# remember to add your own image in case you run this block, if you want to use
the same image,
# download it from: https://encrypted-tbn0.gstatic.com/images?
q=tbn%3AANd9GcRCA4OftnscVzfV8ft8e7vIzQXfXeZdtco8nknJrfCUW6INI4OU
rgb_image = bgr_image[..., ::-1]
fig, ax = plt.subplots(1, 2)
ax[0].imshow(bgr_image)
ax[1].imshow(rgb_image)
plt.show()
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-64-72c49219a822> in <module>()
      7 # remember to add your own image in case you run this block, if you want
to use the same image,
      8 # download it from: https://encrypted-tbn0.gstatic.com/images?
q=tbn%3AANd9GcRCA4OftnscVzfV8ft8e7vIzQXfXeZdtco8nknJrfCUW6INI4OU
----> 9 rgb_image = bgr_image[..., ::-1]
     10 fig, ax = plt.subplots(1, 2)
     11 ax[0].imshow(bgr_image)
```

```
TypeError: 'NoneType' object is not subscriptable
```

Question 10:

Looking at the results above it can be said that the pixel values in the blue channels would be very small compared to red channel. True/False?

Autograd

Pytorch supports automatic differentiation. The module which implements this is called **AutoGrad**. It calculates the gradients and keeps track in forward and backward passes. For primitive tensors, you need to enable or disable it using the `required_grad` flag. But, for advanced tensors, it is enabled by default

```
a = torch.rand((3, 5), requires_grad = True)
print(a)
result = a * 5
print(result)

# grad can be implicitly created only for scalar outputs
# so let's calculate the sum here so that the output becomes a scalar and we can apply a backward pass
mean_result = result.sum()
print(mean_result)
# calculate gradient
mean_result.backward()
# print gradient of a
print(a.grad)
```

```
tensor([[0.1420, 0.0529, 0.6277, 0.6623, 0.4401],
        [0.0893, 0.8400, 0.5712, 0.4038, 0.1173],
        [0.3958, 0.8571, 0.1979, 0.5202, 0.4367]], requires_grad=True)
tensor([[0.7102, 0.2647, 3.1385, 3.3117, 2.2004],
        [0.4464, 4.2000, 2.8562, 2.0192, 0.5864],
        [1.9788, 4.2853, 0.9895, 2.6009, 2.1835]], grad_fn=<MulBackward0>)
tensor(31.7716, grad_fn=<SumBackward0>)
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]])
```

Question 11:

Why the gradient of a is all 5s above?

As we see, Pytorch automagically calculated the gradient value for us. It looks to be the correct value - we multiplied an input by 5, so the gradient of this operation equals to 5.

Disabling Autograd for tensors

We don't need to compute gradients for all the variables that are involved in the pipeline. The Pytorch API provides 2 ways to disable autograd.

`detach` - returns a copy of the tensor with autograd disabled. This

1. copy is built on the same memory as the original tensor, so in-place size / stride / storage changes (such as resize_ / resizeas / set / transpose) modifications are not allowed.
2. torch.no_grad() - It is a context manager that allows you to guard a series of operations from autograd without creating new tensors.

```
a = torch.rand((3, 5), requires_grad=True)
detached_a = a.detach()
detached_result = detached_a * 5
result = a * 10
# we cannot do backward pass that is required for autograd using
multideminsional output,
# so let's calculate the sum here
mean_result = result.sum()
mean_result.backward()
a.grad
```

```
tensor([[10., 10., 10., 10., 10.],
        [10., 10., 10., 10., 10.],
        [10., 10., 10., 10., 10.]])
```

```
a = torch.rand((3, 5), requires_grad=True)
with torch.no_grad():
    detached_result = a * 5
result = a * 10
# we cannot do backward pass that is required for autograd using
multideminsional output,
# so let's calculate the sum here
mean_result = result.sum()
mean_result.backward()
a.grad
```

```
tensor([[10., 10., 10., 10., 10.],
        [10., 10., 10., 10., 10.],
        [10., 10., 10., 10., 10.]])
```

Custom Network

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses PyTorch tensors to manually compute the forward pass, loss, and backward pass.

A PyTorch Tensor is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation.

The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch Tensor can run on either CPU or GPU. To run operations on the GPU, just cast the Tensor to a cuda datatype.

```
dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2*(y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

```
0 41697068.0
1 42741256.0
2 45341812.0
3 39751452.0
4 25930328.0
```

```
5 12725715.0
6 5614377.5
7 2761390.75
8 1680575.875
9 1209127.5
10 951487.9375
11 780990.3125
12 654308.125
13 554848.9375
14 474434.0625
15 408432.6875
16 353772.0625
17 307994.8125
18 269352.90625
19 236650.5625
20 208688.625
21 184659.71875
22 163873.390625
23 145837.0625
24 130119.34375
25 116374.53125
26 104308.7109375
27 93689.65625
28 84314.09375
29 76013.9765625
30 68648.3515625
31 62093.23046875
32 56248.9140625
33 51025.9296875
34 46352.0078125
35 42163.4921875
36 38400.97265625
37 35012.98046875
38 31960.2578125
39 29202.791015625
40 26708.99609375
41 24451.38671875
42 22404.189453125
43 20547.466796875
44 18860.30078125
45 17325.404296875
46 15927.8154296875
47 14653.16015625
48 13490.51953125
49 12431.3388671875
50 11466.0859375
51 10582.70703125
52 9773.423828125
53 9031.99609375
54 8352.2890625
55 7728.07421875
56 7154.50439453125
57 6626.9853515625
58 6141.50048828125
59 5694.49609375
60 5282.9736328125
61 4904.0185546875
62 4554.17822265625
```

```
63 4231.1953125
64 3933.441650390625
65 3658.220458984375
66 3403.770263671875
67 3168.24609375
68 2950.22802734375
69 2748.300048828125
70 2561.19091796875
71 2387.71044921875
72 2226.76513671875
73 2077.4560546875
74 1938.790283203125
75 1810.0057373046875
76 1690.32373046875
77 1579.087890625
78 1475.653564453125
79 1379.4122314453125
80 1289.7012939453125
81 1206.1904296875
82 1128.460205078125
83 1056.072021484375
84 988.5911254882812
85 925.7052612304688
86 867.09619140625
87 812.4027099609375
88 761.3564453125
89 713.7047729492188
90 669.213134765625
91 627.63720703125
92 588.7909545898438
93 552.4854125976562
94 518.5298461914062
95 486.77703857421875
96 457.07135009765625
97 429.2778625488281
98 403.2607727050781
99 378.90155029296875
100 356.08843994140625
101 334.7173156738281
102 314.69500732421875
103 295.92529296875
104 278.33172607421875
105 261.840576171875
106 246.3705291748047
107 231.86231994628906
108 218.24771118164062
109 205.46836853027344
110 193.47393798828125
111 182.2140655517578
112 171.64060974121094
113 161.7060546875
114 152.37240600585938
115 143.60488891601562
116 135.3736572265625
117 127.6404037475586
118 120.36878967285156
119 113.52889251708984
120 107.09272766113281
```

```
121 101.03926849365234
122 95.3431167602539
123 89.98001098632812
124 84.93232727050781
125 80.1795654296875
126 75.70399475097656
127 71.49104309082031
128 67.51925659179688
129 63.777496337890625
130 60.250972747802734
131 56.92778396606445
132 53.795379638671875
133 50.84176254272461
134 48.0562744140625
135 45.429996490478516
136 42.954071044921875
137 40.617462158203125
138 38.41382598876953
139 36.332862854003906
140 34.370635986328125
141 32.517269134521484
142 30.76898765563965
143 29.116533279418945
144 27.55792808532715
145 26.08561897277832
146 24.694355010986328
147 23.379676818847656
148 22.13813591003418
149 20.965492248535156
150 19.85602378845215
151 18.808303833007812
152 17.817197799682617
153 16.880889892578125
154 15.995311737060547
155 15.15793228149414
156 14.366055488586426
157 13.616883277893066
158 12.908146858215332
159 12.237586975097656
160 11.603570938110352
161 11.00322151184082
162 10.4354248046875
163 9.897371292114258
164 9.388818740844727
165 8.9064359664917
166 8.450139999389648
167 8.018111228942871
168 7.608608245849609
169 7.221286773681641
170 6.853899002075195
171 6.506208419799805
172 6.1767144203186035
173 5.864340305328369
174 5.568436622619629
175 5.288036346435547
176 5.022098541259766
177 4.769904136657715
178 4.531124114990234
```

```
179 4.304778575897217
180 4.089944839477539
181 3.8860511779785156
182 3.6927154064178467
183 3.5095672607421875
184 3.3354227542877197
185 3.1705892086029053
186 3.0140340328216553
187 2.8655760288238525
188 2.7244620323181152
189 2.5907294750213623
190 2.463684558868408
191 2.34313702583313
192 2.228688955307007
193 2.1201112270355225
194 2.017040967941284
195 1.9189649820327759
196 1.8258908987045288
197 1.737593650817871
198 1.6535447835922241
199 1.5737546682357788
200 1.497877836227417
201 1.4257644414901733
202 1.357348084449768
203 1.2924302816390991
204 1.2306777238845825
205 1.1719077825546265
206 1.1160553693771362
207 1.0629217624664307
208 1.0124152898788452
209 0.9644915461540222
210 0.9188042879104614
211 0.8754090666770935
212 0.8341062664985657
213 0.7947860956192017
214 0.757378339767456
215 0.7218834161758423
216 0.688075602054596
217 0.6558401584625244
218 0.6252689361572266
219 0.5960175395011902
220 0.5683121085166931
221 0.5418125987052917
222 0.5167039632797241
223 0.49281758069992065
224 0.4699476957321167
225 0.4482196867465973
226 0.42758142948150635
227 0.40780767798423767
228 0.38911840319633484
229 0.3712697923183441
230 0.354256272315979
231 0.3380529582500458
232 0.32258468866348267
233 0.30782800912857056
234 0.2938288152217865
235 0.28045469522476196
236 0.2677578032016754
```

```
237 0.25562939047813416
238 0.24404224753379822
239 0.23301008343696594
240 0.22251316905021667
241 0.21248207986354828
242 0.2029309868812561
243 0.19378213584423065
244 0.18509632349014282
245 0.176781564950943
246 0.16887733340263367
247 0.16132599115371704
248 0.1541213095188141
249 0.14725880324840546
250 0.14069634675979614
251 0.1344941109418869
252 0.1285015493631363
253 0.12282150983810425
254 0.11737260967493057
255 0.11218206584453583
256 0.10722695291042328
257 0.10252313315868378
258 0.09800814837217331
259 0.09369520097970963
260 0.08962604403495789
261 0.08565357327461243
262 0.08191042393445969
263 0.0783218964934349
264 0.0749213844537735
265 0.07165995240211487
266 0.06853197515010834
267 0.06556118279695511
268 0.06274613738059998
269 0.05999977886676788
270 0.05741201713681221
271 0.054932136088609695
272 0.05257082358002663
273 0.05031776428222656
274 0.048139605671167374
275 0.04608524590730667
276 0.04410548508167267
277 0.04222935810685158
278 0.04042355716228485
279 0.0386926494538784
280 0.03704044967889786
281 0.03546064347028732
282 0.033953797072172165
283 0.03250502049922943
284 0.03113105520606041
285 0.02981664426624775
286 0.028558827936649323
287 0.027360480278730392
288 0.026201019063591957
289 0.025094807147979736
290 0.024046752601861954
291 0.02303367853164673
292 0.022077186033129692
293 0.021150600165128708
294 0.020266614854335785
```

295 0.019426193088293076
296 0.018611058592796326
297 0.01785062626004219
298 0.017108455300331116
299 0.016407839953899384
300 0.0157299991697073
301 0.015083320438861847
302 0.014464815147221088
303 0.013861974701285362
304 0.013300239108502865
305 0.012763471342623234
306 0.012243134900927544
307 0.011749541386961937
308 0.011266397312283516
309 0.010812455788254738
310 0.010376501828432083
311 0.009960856288671494
312 0.009560356847941875
313 0.009170137345790863
314 0.008808694779872894
315 0.008454280905425549
316 0.008118793368339539
317 0.007794482633471489
318 0.007487921044230461
319 0.007187684066593647
320 0.006901375949382782
321 0.006632736884057522
322 0.006374903488904238
323 0.006123802158981562
324 0.005888651590794325
325 0.005660739727318287
326 0.005446350667625666
327 0.0052328892052173615
328 0.005025830585509539
329 0.004831657744944096
330 0.004644658882170916
331 0.0044679478742182255
332 0.0042971046641469
333 0.004133136477321386
334 0.003977714106440544
335 0.0038278165739029646
336 0.003684458788484335
337 0.0035473969765007496
338 0.0034140755888074636
339 0.0032898574136197567
340 0.0031679149251431227
341 0.00304879411123693
342 0.002935084281489253
343 0.0028290809132158756
344 0.002723771147429943
345 0.002623329870402813
346 0.0025318902917206287
347 0.002436097478494048
348 0.0023511317558586597
349 0.0022648824378848076
350 0.002186612260689735
351 0.002106821644887924
352 0.0020323770586401224

353 0.0019600491505116224
354 0.0018923120805993676
355 0.0018251631408929825
356 0.0017634676769375801
357 0.0017036865465342999
358 0.0016452072886750102
359 0.0015875922981649637
360 0.0015339134261012077
361 0.0014796849573031068
362 0.0014315239386633039
363 0.0013828749069944024
364 0.0013369454536587
365 0.0012926956405863166
366 0.001250681933015585
367 0.0012088485527783632
368 0.0011703247437253594
369 0.0011315990705043077
370 0.0010963284876197577
371 0.0010604908457025886
372 0.0010283330921083689
373 0.0009953598491847515
374 0.000963767699431628
375 0.00093246379401534 8
376 0.0009045046172104776
377 0.000877134851180017
378 0.0008499176474288106
379 0.0008233329863287508
380 0.0007980092195793986
381 0.000774208456277 8473
382 0.0007493104203604162
383 0.0007266938337124884
384 0.0007053439621813595
385 0.000684786937199533
386 0.0006640242063440382
387 0.0006444114842452109
388 0.000625890155788511
389 0.0006074241828173399
390 0.0005897730006836355
391 0.0005721691413782537
392 0.0005564333405345678
393 0.0005401982925832272
394 0.0005247755325399339
395 0.000510983110871166
396 0.0004960264777764678
397 0.00048262844211421907
398 0.0004690345958806 5743
399 0.000456277484772 7269
400 0.000443584722233 9362
401 0.000432513363193 7206
402 0.00042085189488716424
403 0.000410329521400854
404 0.00039858301170170307
405 0.000388080249198526
406 0.000378755445126 4441
407 0.00036880761035718024
408 0.000358167395461 3507
409 0.00034927541855722666
410 0.00034035247517749667

411 0.00033204955980181694
412 0.0003237562777940184
413 0.00031498822499997914
414 0.0003078297304455191
415 0.00030012454953975976
416 0.0002928829344455153
417 0.00028616582858376205
418 0.00027925442554987967
419 0.00027273231535218656
420 0.00026613505906425416
421 0.0002589935320429504
422 0.0002531931095290929
423 0.00024738258798606694
424 0.00024167619994841516
425 0.00023617155966348946
426 0.00023087780573405325
427 0.0002256411680718884
428 0.00021980957535561174
429 0.00021439349802676588
430 0.00021014249068684876
431 0.0002050718467216938 7
432 0.00020072999177500606
433 0.00019618167425505817
434 0.000192071616766043
435 0.00018807982269208878
436 0.00018373955390416086
437 0.00018011333304457366
438 0.00017592671792954206
439 0.00017252677935175598
440 0.0001688909687800333
441 0.00016549423162359744
442 0.0001617317902855575
443 0.00015873093798290938
444 0.00015536558930762112
445 0.0001526300038676709
446 0.000149371699080802 5
447 0.00014603826275561005
448 0.00014347078104037791
449 0.000140770091093145 3
450 0.00013777597632724792
451 0.000134774574 5710518211
452 0.00013245115405879915
453 0.00012971468095202
454 0.00012735446216538548
455 0.00012458441779017448
456 0.00012227818660903722
457 0.00012017651170026511
458 0.000118248281 0690999
459 0.00011597048433031887
460 0.00011374703171895817
461 0.00011132389772683382
462 0.00010925876267720014
463 0.00010745272447820753
464 0.00010559568909229711
465 0.00010388433292973787
466 0.00010192944318987429
467 0.00010013658902607858
468 9.839859558269382e-05

```
469 9.686136763775721e-05
470 9.52863265410997e-05
471 9.349161700811237e-05
472 9.17674697120674e-05
473 9.023297025123611e-05
474 8.901165710994974e-05
475 8.768695988692343e-05
476 8.629666263004765e-05
477 8.446068386547267e-05
478 8.306359086418524e-05
479 8.205755148082972e-05
480 8.055055513978004e-05
481 7.911526336101815e-05
482 7.770280353724957e-05
483 7.664189615752548e-05
484 7.5345320510678e-05
485 7.42390620871447e-05
486 7.316861592698842e-05
487 7.212166383396834e-05
488 7.111427839845419e-05
489 6.977125303819776e-05
490 6.901657616253942e-05
491 6.775659130653366e-05
492 6.652269803453237e-05
493 6.55873809591867e-05
494 6.454372487496585e-05
495 6.36925979051739e-05
496 6.274688348639756e-05
497 6.193131412146613e-05
498 6.104136991780251e-05
499 6.017859050189145e-05
```

Question 12

In the code above, why do we have 2 in '2.0*(y_pred - y)`?

Question 13

In the code above, what does `grad_h[h < 0] = 0` signify?

Question 14

In the code above, how many "epochs" have we trained the model for?

Question 15

In the code above, if we take the trained model, and run it on fresh inputs, the trained model will be able to predict fresh output with high accuracy.

Question 16

In the code above, if we dont use clone in `grad_h = grad_h_relu.clone()` the model will still train without any issues.