Introduction

For many years I have been using PyTorch to construct and train deep learning models. Even though I have learned its syntax and rules, something has always aroused my curiosity: what is happening internally during these operations? How does all of this work?

If you have gotten here, you probably have the same questions. If I ask you how to create and train a model in PyTorch, you will probably come up with something similar to the code below:

```
import torch
import torch.nn as nn
import torch.optim as optim

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(1, 10)
        self.sigmoid = nn.Sigmoid()
        self.fc2 = nn.Linear(10, 1)

    def forward(self, x):
        out = self.fc1(x)
        out = self.sigmoid(out)
        out = self.fc2(out)

        return out

...

model = MyModel().to(device)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

for epoch in range(epochs):
    for x, y in ...

        x = x.to(device)
```

```
        y = y.to(device)

        outputs = model(x)
        loss = criterion(outputs, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

But what if I ask you how does this backward step works? Or, for instance, what happens when you reshape a tensor? Is the data rearranged internally? How does that happens? Why is PyTorch so fast? How does PyTorch handle GPU operations? These are the types of questions that have always intrigued me, and I imagine they also intrigue you. Thus, in order to better understand these concepts, what is better than building your own tensor library ***from scratch***? And that is what you will learn in this article!

#1 — Tensor

In order to construct a *tensor library*, the first concept you need to learn obviously is: what is a tensor?

You may have an intuitive idea that a tensor is a mathematical concept of a n-dimensional data structure that contains some numbers. But here we need to understand how to model this data structure from a computational perspective. We can think of a tensor as consisting of the data itself and also some metadata describing aspects of the tensor such as its shape or the device it lives in (i.e. CPU memory, GPU memory...).

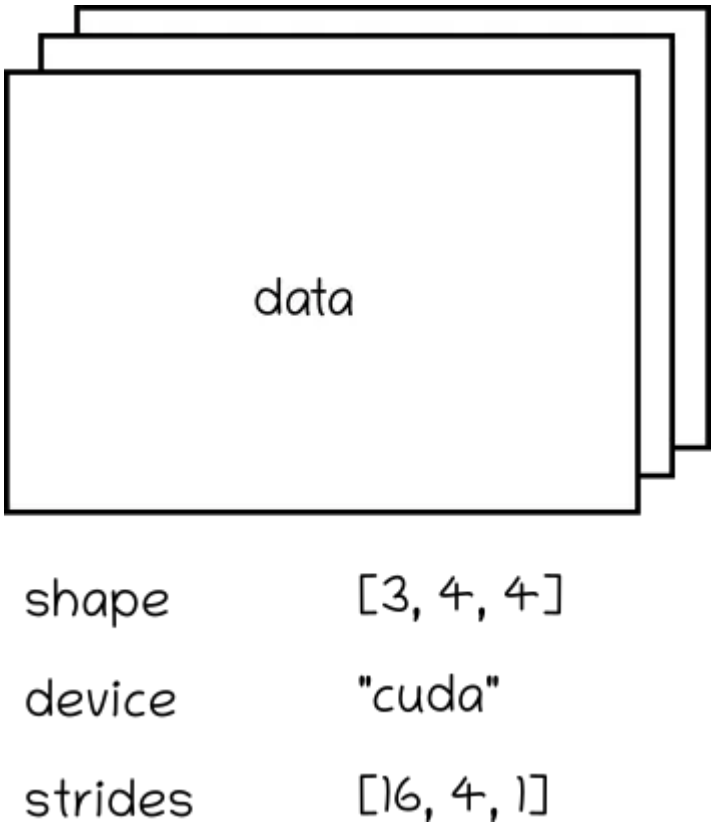
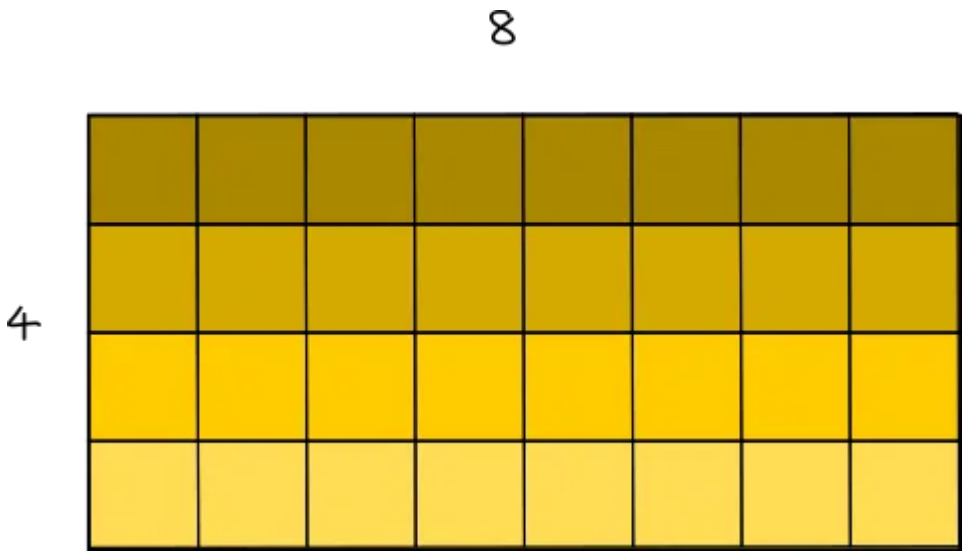


Image by Author

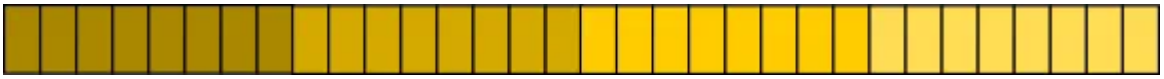
There is also a less popular metadata that you may have never heard of, called *stride*. This concept is very important to understand the internals of tensor data rearrangement, so we need to discuss it a little more.

Imagine a 2-D tensor with shape [4, 8], illustrated below.



4x8 Tensor (Image by Author)

The data (i.e. float numbers) of a tensor is actually stored as a 1-dimensional array on memory:



1-D dimensional data array of the tensor (Image by Author)

So, in order to represent this 1-dimensional array as a N-dimensional tensor, we use strides. Basically the idea is the following:

We have a matrix with 4 rows and 8 columns. Considering that all of its elements are organized by rows on the 1-dimensional array, if we want to access the value at position [2, 3], we need to traverse 2 rows (of 8 elements each) plus 3 positions. In mathematical terms, we need to traverse 3 + 2 * 8 elements on the 1-dimensional array:

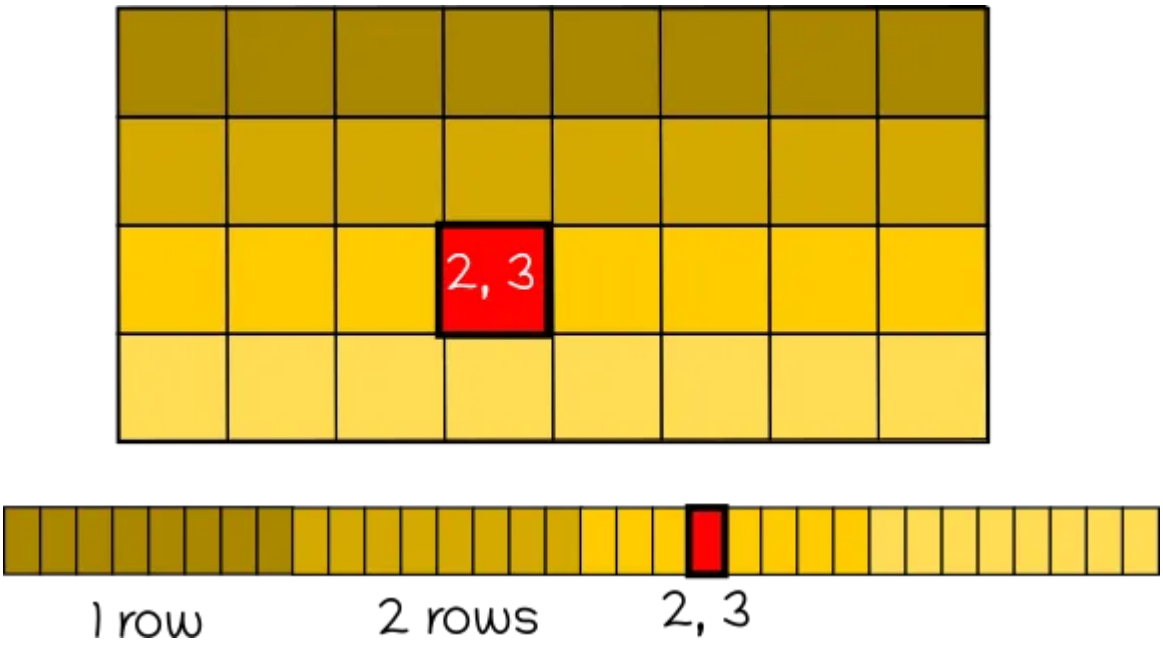
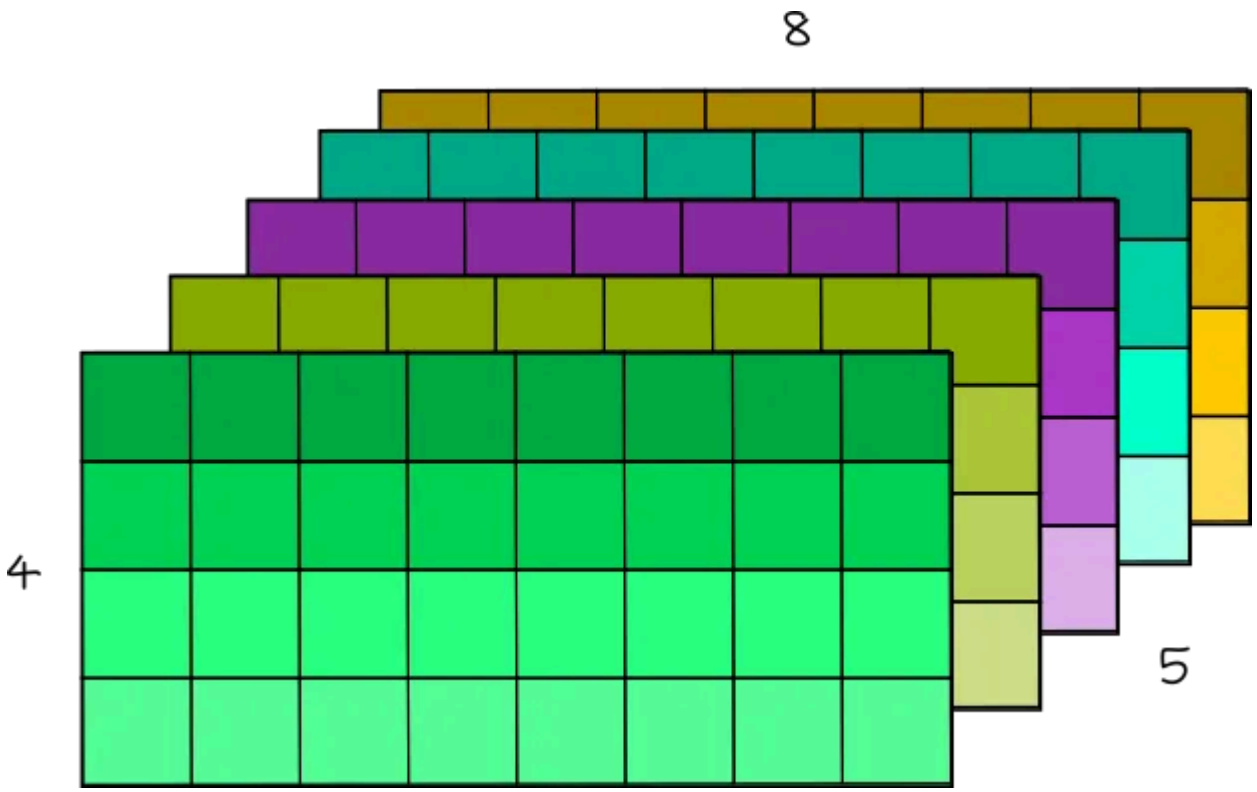


Image by Author

So this ‘8’ is the *stride* of the *second* dimension. In this case, it is the information of how many elements I need to traverse on the array to “jump” to other positions on the *second dimension*.

Thus, for accessing the element [i, j] of a 2-dimensional tensor with shape [shape_0, shape_1], we basically need to access the element at position j + i * shape_1

Now, let us imagine a 3-dimensional tensor:



5x4x8 Tensor (Image by Author)

You can think of this 3-dimensional tensor as a sequence of matrices. For example, you can think of this [5, 4, 8] tensor as 5 matrices of shape [4, 8].

Now, in order to access the element at position [1, 2, 7], you need to traverse 1 complete matrix of shape [4,8], 2 rows of shape [8] and 7 columns of shape [1]. So,

you need to traverse (1 * 4 * 8) + (2 * 8) + (7 * 1) positions on the 1-dimensional array.

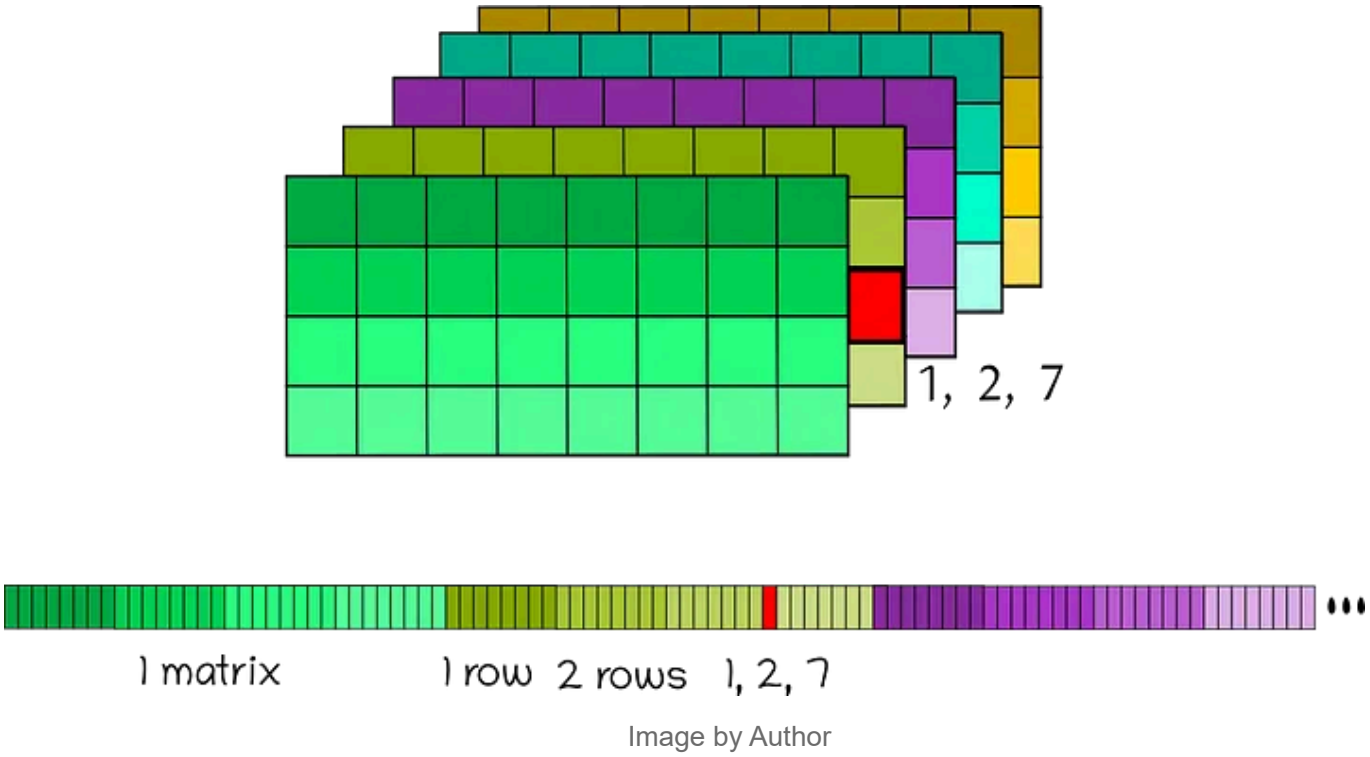


Image by Author

Thus, to access the element [i][j][k] of a 3-D tensor with [shape_0, shape_1, shape_2] on the 1-D data array, you do:

$$T[i][j][k] = \text{array}[i * (\text{shape}[1] * \text{shape}[2]) + j * (shape[2]) + k * (1)]$$

This shape_1 * shape_2 is the *stride* of the first dimension, the shape_2 is the *stride* of the second dimension and 1 is the stride of the third dimension.

Then, in order to generalize:

$$T[i][j][k]...[z] = \text{array}[i * \text{stride}[0] + j * \text{stride}[1] + k * \text{stride}[2] + ... + z * \text{stride}[n-1]]$$

Where the *strides* of each dimension can be calculated using the product of the next dimension tensor shapes:

$$stride[k] = \prod_{i=k+1}^{N-1} shape[i]$$

Then we set stride[n-1] = 1.

On our tensor example of shape [5, 4, 8] we would have strides = [4*8, 8, 1] = [32, 8, 1]

You can test on your own:

```
import torch

torch.rand([5, 4, 8]).stride()
#(32, 8, 1)
```

Ok, but why do we need shapes and strides? Beyond accessing elements of N-dimensional tensors stored as 1-dimensional arrays, this concept can be used to manipulate tensor arrangements very easily.

For example, to reshape a tensor, you only need to set the new shape and calculate the new strides based on it! (since the new shape guarantees the same number of elements)

```
import torch

t = torch.rand([5, 4, 8])

print(t.shape)
# [5, 4, 8]

print(t.stride())
# [32, 8, 1]

new_t = t.reshape([4, 5, 2, 2, 2])

print(new_t.shape)
# [4, 5, 2, 2, 2]

print(new_t.stride())
# [40, 8, 4, 2, 1]
```

Internally, the tensor is still stored as the same 1-dimensional array. The reshape method did not change the order of the elements within the array! That's amazing, isn't? 😁

You can verify on your own using the following function that accesses the internal 1-dimensional array on PyTorch:

```
import ctypes

def print_internal(t: torch.Tensor):
    print(
        torch.frombuffer(
            ctypes.string_at(t.data_ptr(), t.storage().nbytes()), dtype=t.dtype
        )
    )

print_internal(t)
# [0.0752, 0.5898, 0.3930, 0.9577, 0.2276, 0.9786, 0.1009, 0.138, ...
```

```
print_internal(new_t)
# [0.0752, 0.5898, 0.3930, 0.9577, 0.2276, 0.9786, 0.1009, 0.138, ...
```

Or for instance, you want to transpose two axes. Internally, you just need to swap the respective strides!

```
t = torch.arange(0, 24).reshape(2, 3, 4)
print(t)
# [[[ 0,  1,  2,  3],
#   [ 4,  5,  6,  7],
#   [ 8,  9, 10, 11]],

#  [[12, 13, 14, 15],
#   [16, 17, 18, 19],
#   [20, 21, 22, 23]]]

print(t.shape)
# [2, 3, 4]

print(t.stride())
# [12, 4, 1]

new_t = t.transpose(0, 1)
print(new_t)
# [[[ 0,  1,  2,  3],
#   [12, 13, 14, 15]],

#  [[ 4,  5,  6,  7],
#   [16, 17, 18, 19]],

#  [[ 8,  9, 10, 11],
#   [20, 21, 22, 23]]]

print(new_t.shape)
# [3, 2, 4]

print(new_t.stride())
# [4, 12, 1]
```

If you print the internal array, both have the same values:

```
print_internal(t)
# [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

print_internal(new_t)
# [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
```

However, the stride of `new_t` now does not match with the equation I showed above. This is due to the fact that the tensor is now not contiguous. That means that although the internal array remains the same, the order of its values in memory does not match with the actual order of the tensor.

```
t.is_contiguous()
# True

new_t.is_contiguous()
# False
```

This means the accessing the non-contiguous elements in sequence is less efficient (as the real tensor elements is not ordered in sequence on memory). In order to fix that, we can do:

```
new_t_contiguous = new_t.contiguous()

print(new_t_contiguous.is_contiguous())
# True
```

If we analyze the internal array, its order now matches with the actual tensor order now, which can provide better memory access efficiency:

```
print(new_t)
# [[[ 0,  1,  2,  3],
#   [12, 13, 14, 15]],

#  [[ 4,  5,  6,  7],
#   [16, 17, 18, 19]],

#  [[ 8,  9, 10, 11],
#   [20, 21, 22, 23]]]

print_internal(new_t)
# [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

print_internal(new_t_contiguous)
# [ 0,  1,  2,  3, 12, 13, 14, 15,  4,  5,  6,  7, 16, 17, 18, 19,  8,  9, 10, 11, 20,
```

Now that we comprehend how a tensor is modeled, let us start creating our library!

I will call it *Norch*, which stands for NOT PyTorch, and also makes an allusion to my last name, Nogueira 😁

The first thing to know is that although PyTorch is used through Python, internally it runs C/C++. So we will first create our internals C/C++ functions.

We can first define a tensor as a struct to store its data and metadata, and create a function to instantiate it:

```c
//norch/csrc/tensor.cpp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct {
    float* data;
    int* strides;
    int* shape;
    int ndim;
    int size;
    char* device;
} Tensor;

Tensor* create_tensor(float* data, int* shape, int ndim) {

    Tensor* tensor = (Tensor*)malloc(sizeof(Tensor));
    if (tensor == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    tensor->data = data;
    tensor->shape = shape;
    tensor->ndim = ndim;

    tensor->size = 1;
    for (int i = 0; i < ndim; i++) {
        tensor->size *= shape[i];
    }

    tensor->strides = (int*)malloc(ndim * sizeof(int));
    if (tensor->strides == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    int stride = 1;
    for (int i = ndim - 1; i >= 0; i--) {
        tensor->strides[i] = stride;
        stride *= shape[i];
    }

    return tensor;
}
```

In order to access some element, we can take advantage of strides, as we learned before:

```
//norch/csrc/tensor.cpp

float get_item(Tensor* tensor, int* indices) {
    int index = 0;
    for (int i = 0; i < tensor->ndim; i++) {
        index += indices[i] * tensor->strides[i];
    }

    float result;
    result = tensor->data[index];

    return result;
}
```

Now, we can create the tensor operations. I will show some examples and you can find the complete version in the repository linked at the end of this article.

```
//norch/csrc/cpu.cpp

void add_tensor_cpu(Tensor* tensor1, Tensor* tensor2, float* result_data) {

    for (int i = 0; i < tensor1->size; i++) {
        result_data[i] = tensor1->data[i] + tensor2->data[i];
    }
}

void sub_tensor_cpu(Tensor* tensor1, Tensor* tensor2, float* result_data) {

    for (int i = 0; i < tensor1->size; i++) {
        result_data[i] = tensor1->data[i] - tensor2->data[i];
    }
}

void elementwise_mul_tensor_cpu(Tensor* tensor1, Tensor* tensor2, float* result_data)
```

```
    for (int i = 0; i < tensor1->size; i++) {
        result_data[i] = tensor1->data[i] * tensor2->data[i];
    }
}

void assign_tensor_cpu(Tensor* tensor, float* result_data) {

    for (int i = 0; i < tensor->size; i++) {
        result_data[i] = tensor->data[i];
    }
}

...
```

After that we are able to create our other tensor functions that will call these operations:

```
//norch/csrc/tensor.cpp

Tensor* add_tensor(Tensor* tensor1, Tensor* tensor2) {
    if (tensor1->ndim != tensor2->ndim) {
        fprintf(stderr, "Tensors must have the same number of dimensions %d and %d fo
        exit(1);
    }

    int ndim = tensor1->ndim;
    int* shape = (int*)malloc(ndim * sizeof(int));
    if (shape == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }

    for (int i = 0; i < ndim; i++) {
        if (tensor1->shape[i] != tensor2->shape[i]) {
            fprintf(stderr, "Tensors must have the same shape %d and %d at index %d fo
            exit(1);
        }
        shape[i] = tensor1->shape[i];
    }
    float* result_data = (float*)malloc(tensor1->size * sizeof(float));
    if (result_data == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
```

```
        exit(1);
    }
    add_tensor_cpu(tensor1, tensor2, result_data);

    return create_tensor(result_data, shape, ndim, device);
}
```

As mentioned before, the tensor reshaping does not modify the internal data array:

```
//norch/csrc/tensor.cpp

Tensor* reshape_tensor(Tensor* tensor, int* new_shape, int new_ndim) {

    int ndim = new_ndim;
    int* shape = (int*)malloc(ndim * sizeof(int));
    if (shape == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }

    for (int i = 0; i < ndim; i++) {
        shape[i] = new_shape[i];
    }

    // Calculate the total number of elements in the new shape
    int size = 1;
    for (int i = 0; i < new_ndim; i++) {
        size *= shape[i];
    }

    // Check if the total number of elements matches the current tensor's size
    if (size != tensor->size) {
        fprintf(stderr, "Cannot reshape tensor. Total number of elements in new shape
        exit(1);
    }

    float* result_data = (float*)malloc(tensor->size * sizeof(float));
    if (result_data == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
```

```
    assign_tensor_cpu(tensor, result_data);
    return create_tensor(result_data, shape, ndim, device);
}
```

Although we can now do some tensor operations, nobody deserves to run it using C/C++, right? Let us start building our Python wrapper!

There are a lot of options to run C/C++ code using Python, such as *Pybind11* and *Cython*. For our example I will use *ctypes*.

The basically structure of *ctypes* is shown below:

```
//C code
#include <stdio.h>

float add_floats(float a, float b) {
    return a + b;
}
```

```
# Compile
gcc -shared -o add_floats.so -fPIC add_floats.c
```

```
# Python code
import ctypes

# Load the shared library
lib = ctypes.CDLL('./add_floats.so')

# Define the argument and return types for the function
lib.add_floats.argtypes = [ctypes.c_float, ctypes.c_float]
lib.add_floats.restype = ctypes.c_float

# Convert python float to c_float type
```

```
a = ctypes.c_float(3.5)
b = ctypes.c_float(2.2)

# Call the C function
result = lib.add_floats(a, b)
print(result)
# 5.7
```

As you can see, it is very intuitive. After you compile the C/C++ code, you can use it on Python with *ctypes* very easily. You only need to define the arguments & return c_types of the function, convert the variable to its respective c_types and call the function. For more complex types such as arrays (float lists) you can use pointers.

```
data = [1.0, 2.0, 3.0]
data_ctype = (ctypes.c_float * len(data))(*data)

lib.some_array_func.argstypes = [ctypes.POINTER(ctypes.c_float)]

...

lib.some_array_func(data)
```

And for struct types we can create our own c_type:

```
class CustomType(ctypes.Structure):
    _fields_ = [
        ('field1', ctypes.POINTER(ctypes.c_float)),
        ('field2', ctypes.POINTER(ctypes.c_int)),
        ('field3', ctypes.c_int),
    ]
```

```
# Can be used as ctypes.POINTER(CustomType)
```

After this brief explanation, let us construct the Python wrapper for our tensor C/C++ library!

```
# norch/tensor.py

import ctypes

class CTensor(ctypes.Structure):
    _fields_ = [
        ('data', ctypes.POINTER(ctypes.c_float)),
        ('strides', ctypes.POINTER(ctypes.c_int)),
        ('shape', ctypes.POINTER(ctypes.c_int)),
        ('ndim', ctypes.c_int),
        ('size', ctypes.c_int),
    ]

class Tensor:
    os.path.abspath(os.curdir)
    _C = ctypes.CDLL("COMPILED_LIB.so"))

    def __init__(self):

        data, shape = self.flatten(data)
        self.data_ctype = (ctypes.c_float * len(data))(*data)
        self.shape_ctype = (ctypes.c_int * len(shape))(*shape)
        self.ndim_ctype = ctypes.c_int(len(shape))

        self.shape = shape
        self.ndim = len(shape)

        Tensor._C.create_tensor.argtypes = [ctypes.POINTER(ctypes.c_float), ctypes.PO
        Tensor._C.create_tensor.restype = ctypes.POINTER(CTensor)

        self.tensor = Tensor._C.create_tensor(
            self.data_ctype,
            self.shape_ctype,
            self.ndim_ctype,
        )
```

```
    def flatten(self, nested_list):
        """
        This method simply convert a list type tensor to a flatten tensor with its sha

        Example:

        Arguments:
            nested_list: [[1, 2, 3], [-5, 2, 0]]
        Return:
            flat_data: [1, 2, 3, -5, 2, 0]
            shape: [2, 3]
        """
        def flatten_recursively(nested_list):
            flat_data = []
            shape = []
            if isinstance(nested_list, list):
                for sublist in nested_list:
                    inner_data, inner_shape = flatten_recursively(sublist)
                    flat_data.extend(inner_data)
                shape.append(len(nested_list))
                shape.extend(inner_shape)
            else:
                flat_data.append(nested_list)
            return flat_data, shape

        flat_data, shape = flatten_recursively(nested_list)
        return flat_data, shape
```

Now we include the Python tensor operations to call the C/C++ operations.

```
# norch/tensor.py

def __getitem__(self, indices):
    """
    Access tensor by index tensor[i, j, k...]
    """

    if len(indices) != self.ndim:
        raise ValueError("Number of indices must match the number of dimensions")

    Tensor._C.get_item.argtypes = [ctypes.POINTER(CTensor), ctypes.POINTER(ctypes.c_in
    Tensor._C.get_item.restype = ctypes.c_float
```

```
        indices = (ctypes.c_int * len(indices))(*indices)
        value = Tensor._C.get_item(self.tensor, indices)

        return value

    def reshape(self, new_shape):
        """
        Reshape tensor
        result = tensor.reshape([1,2])
        """
        new_shape_ctype = (ctypes.c_int * len(new_shape))(*new_shape)
        new_ndim_ctype = ctypes.c_int(len(new_shape))

        Tensor._C.reshape_tensor.argtypes = [ctypes.POINTER(CTensor), ctypes.POINTER(ctype
        Tensor._C.reshape_tensor.restype = ctypes.POINTER(CTensor)
        result_tensor_ptr = Tensor._C.reshape_tensor(self.tensor, new_shape_ctype, new_nd

        result_data = Tensor()
        result_data.tensor = result_tensor_ptr
        result_data.shape = new_shape.copy()
        result_data.ndim = len(new_shape)
        result_data.device = self.device

        return result_data

    def __add__(self, other):
        """
        Add tensors
        result = tensor1 + tensor2
        """

        if self.shape != other.shape:
            raise ValueError("Tensors must have the same shape for addition")

        Tensor._C.add_tensor.argtypes = [ctypes.POINTER(CTensor), ctypes.POINTER(CTensor)]
        Tensor._C.add_tensor.restype = ctypes.POINTER(CTensor)

        result_tensor_ptr = Tensor._C.add_tensor(self.tensor, other.tensor)

        result_data = Tensor()
        result_data.tensor = result_tensor_ptr
        result_data.shape = self.shape.copy()
        result_data.ndim = self.ndim
        result_data.device = self.device

        return result_data

    # Include the other operations:
    # __str__
```

```
# __sub__ (-)
# __mul__ (*)
# __matmul__ (@)
# __pow__ (**)
# __truediv__ (/)
# log
# ...
```

If you got here, you are now capable to run the code and start doing some tensor operations!

```
import norch

tensor1 = norch.Tensor([[1, 2, 3], [3, 2, 1]])
tensor2 = norch.Tensor([[3, 2, 1], [1, 2, 3]])

result = tensor1 + tensor2
print(result[0, 0])
# 4
```

#2 — GPU Support

After creating the basic structure of our library, now we will take it to a new level. It is well-known that you can call `.to("cuda")` to send data to GPU and run math operations faster. I will assume that you have basic knowledge on how CUDA works, but if you do not, you can read my other article: CUDA tutorial. I will wait here. 😊

...

For those in a hurry, a simple introduction here:

Basically, all of our code until here is running on CPU memory. Altough for single operations CPUs are faster, the advantage of GPUs relies on its parallelization capabilities. While CPU design aims to execute a sequence of operations (threads) fast (but can only execute dozens of them), the GPU design aims to execute millions of operations in parallel (by sacrificing individual threads performance).

Thus, we can take advantage of this capability to perform operations in parallel. For example, in a million-sized tensor addition, instead of adding the elements of each index sequentially inside a loop, using a GPU we can add all of them in parallel at once. In order to do that, we can use CUDA, which is a platform developed by NVIDIA to enable developers to integrate GPU support to their software applications.

In order to do that, you can use CUDA C/C++, which is a simple C/C++ based interface designed to run specific GPU operations (such as copy data from CPU memory to GPU memory).

The code below basically uses some CUDA C/C++ functions to copy data from CPU to GPU, and run the AddTwoArrays function (also called kernel) on a total of N GPU threads in parallel, each responsible to add a different element of the array.

```
#include <stdio.h>

// CPU version for comparison
void AddTwoArrays_CPU(flaot A[], float B[], float C[]) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}

// Kernel definition
__global__ void AddTwoArrays_GPU(float A[], float B[], float C[]) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
```

```
}

int main() {

    int N = 1000; // Size of the arrays
    float A[N], B[N], C[N]; // Arrays A, B, and C

    ...

    float *d_A, *d_B, *d_C; // Device pointers for arrays A, B, and C

    // Allocate memory on the device for arrays A, B, and C
    cudaMalloc((void **)&d_A, N * sizeof(float));
    cudaMalloc((void **)&d_B, N * sizeof(float));
    cudaMalloc((void **)&d_C, N * sizeof(float));

    // Copy arrays A and B from host to device
    cudaMemcpy(d_A, A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, N * sizeof(float), cudaMemcpyHostToDevice);

    // Kernel invocation with N threads
    AddTwoArrays_GPU<<<1, N>>>(d_A, d_B, d_C);

    // Copy vector C from device to host
    cudaMemcpy(C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

}
```

As you can notice, instead of adding each element pair per operation, we run all of the adding operations in parallel, getting rid of the loop instruction.

After this brief introduction, we can go back to our tensor library.

The first step is to create a function to send tensor data from CPU to GPU and vice versa.

```
//norch/csrc/tensor.cpp
```

```
void to_device(Tensor* tensor, char* target_device) {
    if ((strcmp(target_device, "cuda") == 0) && (strcmp(tensor->device, "cpu") == 0))
        cpu_to_cuda(tensor);
    }

    else if ((strcmp(target_device, "cpu") == 0) && (strcmp(tensor->device, "cuda") ==
        cuda_to_cpu(tensor);
    }
}
```

```
//norch/csrc/cuda.cu

__host__ void cpu_to_cuda(Tensor* tensor) {

    float* data_tmp;
    cudaMalloc((void **)&data_tmp, tensor->size * sizeof(float));
    cudaMemcpy(data_tmp, tensor->data, tensor->size * sizeof(float), cudaMemcpyHostToI

    tensor->data = data_tmp;

    const char* device_str = "cuda";
    tensor->device = (char*)malloc(strlen(device_str) + 1);
    strcpy(tensor->device, device_str);

    printf("Successfully sent tensor to: %s\n", tensor->device);
}

__host__ void cuda_to_cpu(Tensor* tensor) {
    float* data_tmp = (float*)malloc(tensor->size * sizeof(float));

    cudaMemcpy(data_tmp, tensor->data, tensor->size * sizeof(float), cudaMemcpyDevice
    cudaFree(tensor->data);

    tensor->data = data_tmp;

    const char* device_str = "cpu";
    tensor->device = (char*)malloc(strlen(device_str) + 1);
    strcpy(tensor->device, device_str);

    printf("Successfully sent tensor to: %s\n", tensor->device);
}
```

The Python wrapper:

```
# norch/tensor.py

def to(self, device):
    self.device = device
    self.device_ctype = self.device.encode('utf-8')

    Tensor._C.to_device.argtypes = [ctypes.POINTER(CTensor), ctypes.c_char_p]
    Tensor._C.to_device.restype = None
    Tensor._C.to_device(self.tensor, self.device_ctype)

    return self
```

Then, we create GPU versions for all of our tensor operations. I will write examples for addition and subtraction:

```
//norch/csrc/cuda.cu

#define THREADS_PER_BLOCK 128

__global__ void add_tensor_cuda_kernel(float* data1, float* data2, float* result_data,

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        result_data[i] = data1[i] + data2[i];
    }
}

__host__ void add_tensor_cuda(Tensor* tensor1, Tensor* tensor2, float* result_data) {

    int number_of_blocks = (tensor1->size + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK
    add_tensor_cuda_kernel<<<number_of_blocks, THREADS_PER_BLOCK>>>(tensor1->data, ten

    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }
```

```
    cudaDeviceSynchronize();
}

__global__ void sub_tensor_cuda_kernel(float* data1, float* data2, float* result_data,

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        result_data[i] = data1[i] - data2[i];
    }
}

__host__ void sub_tensor_cuda(Tensor* tensor1, Tensor* tensor2, float* result_data) {

    int number_of_blocks = (tensor1->size + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK
    sub_tensor_cuda_kernel<<<number_of_blocks, THREADS_PER_BLOCK>>>(tensor1->data, ten

    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }

    cudaDeviceSynchronize();
}

...
```

Subsequently, we include a new tensor attribute `char* device` on the `tensor.cpp` and we can use it to select where the operations will be run (CPU or GPU):

```
//norch/csrc/tensor.cpp

Tensor* add_tensor(Tensor* tensor1, Tensor* tensor2) {
    if (tensor1->ndim != tensor2->ndim) {
        fprintf(stderr, "Tensors must have the same number of dimensions %d and %d for
        exit(1);
    }

    if (strcmp(tensor1->device, tensor2->device) != 0) {
```

```
        fprintf(stderr, "Tensors must be on the same device: %s and %s\n", tensor1->de
        exit(1);
    }

    char* device = (char*)malloc(strlen(tensor1->device) + 1);
    if (device != NULL) {
        strcpy(device, tensor1->device);
    } else {
        fprintf(stderr, "Memory allocation failed\n");
        exit(-1);
    }
    int ndim = tensor1->ndim;
    int* shape = (int*)malloc(ndim * sizeof(int));
    if (shape == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }

    for (int i = 0; i < ndim; i++) {
        if (tensor1->shape[i] != tensor2->shape[i]) {
            fprintf(stderr, "Tensors must have the same shape %d and %d at index %d fo
            exit(1);
        }
        shape[i] = tensor1->shape[i];
    }

    if (strcmp(tensor1->device, "cuda") == 0) {

        float* result_data;
        cudaMalloc((void **)&result_data, tensor1->size * sizeof(float));
        add_tensor_cuda(tensor1, tensor2, result_data);
        return create_tensor(result_data, shape, ndim, device);
    }
    else {
        float* result_data = (float*)malloc(tensor1->size * sizeof(float));
        if (result_data == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(1);
        }
        add_tensor_cpu(tensor1, tensor2, result_data);
        return create_tensor(result_data, shape, ndim, device);
    }
}
```

Now our library has GPU support!

```
import norch

tensor1 = norch.Tensor([[1, 2, 3], [3, 2, 1]]).to("cuda")
tensor2 = norch.Tensor([[3, 2, 1], [1, 2, 3]]).to("cuda")

result = tensor1 + tensor2
```

#3 — Automatic Differentation (Autograd)

One of the main reasons why PyTorch got so popular is due to its Autograd module. It is a core component that allows automatic differentiation in order to compute gradients (crucial for training models with optimization algorithms such as gradient descent). By calling a single method `.backward()`, it computes all of the gradients from previous tensor operations:

```
x = torch.tensor([[1., 2, 3], [3., 2, 1]], requires_grad=True)
# [[1,  2,  3],
#  [3,  2., 1]]

y = torch.tensor([[3., 2, 1], [1., 2, 3]], requires_grad=True)
# [[3,  2, 1],
#  [1,  2, 3]]

L = ((x - y) ** 3).sum()

L.backward()

# You can access gradients of x and y
print(x.grad)
# [[12, 0, 12],
#  [12, 0, 12]]

print(y.grad)
# [[-12, 0, -12],
#  [-12, 0, -12]]

# In order to minimize z, you can use that for gradient descent:
```

```
# x = x - learning_rate * x.grad
# y = y - learning_rate * y.grad
```

In order to understand what is happening, let's try to replicate the same procedure manually:

$$x = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_3 & x_4 & x_5 \end{bmatrix} y = \begin{bmatrix} y_0 & y_1 & y_2 \\ y_3 & y_4 & y_5 \end{bmatrix}$$

$$L = \sum_{i=0}^{5} (x_i - y_i)^3$$

Let's first calculate:

$$\frac{\partial L}{\partial x}$$

Note that *x* is a matrix, thus we need to calculate the derivative of L with respect to each element individually. Additionally, L is a summation over all elements, but it is important to remember that for each elements, the other elements does not interfere on its derivative. Therefore, we obtain the following term:

$$\frac{\partial L}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x_0}(x_0 - y_0)^3 & \frac{\partial}{\partial x_1}(x_1 - y_1)^3 & \frac{\partial}{\partial x_2}(x_2 - y_2)^3 \\ \frac{\partial}{\partial x_3}(x_3 - y_3)^3 & \frac{\partial}{\partial x_4}(x_4 - y_4)^3 & \frac{\partial}{\partial x_5}(x_5 - y_5)^3 \end{bmatrix}$$

By applying the chain rule for each term, we differentiate the outer function and multiply by the derivative of the inner function:

$$\frac{\partial}{\partial x_i}(x_i - y_i)^3 = 3(x_i - y_i)^2 . \frac{\partial}{\partial x_i}(x_i - y_i)$$

Where:

$$\frac{\partial}{\partial x_i}(x_i - y_i) = \frac{\partial}{\partial x_i} x_i$$

Finally:

$$\frac{\partial}{\partial x_i} x_i = 1$$

Thus, we have the following final equation to calculate the derivative of L with respect to *x:*

$$\frac{\partial L}{\partial x} = \begin{bmatrix} 3(x_0 - y_0)^2 & 3(x_1 - y_1)^2 & 3(x_2 - y_2)^2 \\ 3(x_3 - y_3)^2 & 3(x_4 - y_4)^2 & 3(x_5 - y_5)^2 \end{bmatrix}$$

Substituting the values into the equation:

$$\frac{\partial L}{\partial x} = \begin{bmatrix} 3(1-3)^2 & 3(2-2)^2 & 3(3-1)^2 \\ 3(3-1)^2 & 3(2-2)^2 & 3(1-3)^2 \end{bmatrix}$$

Calculating the result, we get the same values we obtained with PyTorch:

$$\frac{\partial L}{\partial x} = \begin{bmatrix} 12 & 0 & 12 \\ 12 & 0 & 12 \end{bmatrix}$$

Now, let's analyze what we just did:

Basically, we observed all the operations involved in reserved order: a summation, a power of 3 and a subtraction. Then, we applied chain of rule, calculating the derivative of each operation and recursively calculated the derivative for the next operation. So, first we need an implementation of the derivative for different math operations:

For addition:

$$\frac{\partial(f(x)+g(y))}{\partial x} = 1 \cdot \frac{df}{dx}$$
$$\frac{\partial(f(x)+g(y))}{\partial y} = 1 \cdot \frac{dg}{dy}$$

```
# norch/autograd/functions.py

class AddBackward:
```

```
    def __init__(self, x, y):
        self.input = [x, y]

    def backward(self, gradient):
        return [gradient, gradient]
```

For sin:

$$\frac{d(sin(f(x)))}{dx} = cos(x)\frac{df}{dx}$$

```
# norch/autograd/functions.py

class SinBackward:
    def __init__(self, x):
        self.input = [x]

    def backward(self, gradient):
        x = self.input[0]
        return [x.cos() * gradient]
```

For cosine:

$$\frac{d(cos(f(x)))}{dx} = -sin(x)\frac{df}{dx}$$

```
# norch/autograd/functions.py
```

```
class CosBackward:
    def __init__(self, x):
        self.input = [x]

    def backward(self, gradient):
        x = self.input[0]
        return [- x.sin() * gradient]
```

For element-wise multiplication:

$$\frac{\partial(f(x)*g(y))}{\partial x} = g(y) \cdot \frac{df}{dx}$$
$$\frac{\partial(f(x)*g(y))}{\partial y} = f(x) \cdot \frac{dg}{dy}$$

```
# norch/autograd/functions.py

class ElementwiseMulBackward:
    def __init__(self, x, y):
        self.input = [x, y]

    def backward(self, gradient):
        x = self.input[0]
        y = self.input[1]
        return [y * gradient, x * gradient]
```

For summation:

```
# norch/autograd/functions.py

class SumBackward:
    def __init__(self, x):
```

```
            self.input = [x]

        def backward(self, gradient):
            # Since sum reduces a tensor to a scalar, gradient is broadcasted to match the
            return [float(gradient.tensor.contents.data[0]) * self.input[0].ones_like()]
```

You can access the GitHub repository link at the end of the article to explore other operations.

Now that we have derivative expressions for each operation, we can proceed with implementing the recursive backward chain rule. We can set a `requires_grad` argument for our tensor to indicate that we want to store the gradients of this tensor. If true, we will store the gradients for each tensor operation. For instance:

```
# norch/tensor.py

def __add__(self, other):

  if self.shape != other.shape:
      raise ValueError("Tensors must have the same shape for addition")

  Tensor._C.add_tensor.argtypes = [ctypes.POINTER(CTensor), ctypes.POINTER(CTensor)]
  Tensor._C.add_tensor.restype = ctypes.POINTER(CTensor)

  result_tensor_ptr = Tensor._C.add_tensor(self.tensor, other.tensor)

  result_data = Tensor()
  result_data.tensor = result_tensor_ptr
  result_data.shape = self.shape.copy()
  result_data.ndim = self.ndim
  result_data.device = self.device

  result_data.requires_grad = self.requires_grad or other.requires_grad
```

```
    if result_data.requires_grad:
        result_data.grad_fn = AddBackward(self, other)
```

Then, implement the `.backward()` method:

```
# norch/tensor.py

def backward(self, gradient=None):
    if not self.requires_grad:
        return

    if gradient is None:
        if self.shape == [1]:
            gradient = Tensor([1]) # dx/dx = 1 case
        else:
            raise RuntimeError("Gradient argument must be specified for non-scalar ten

    if self.grad is None:
        self.grad = gradient

    else:
        self.grad += gradient

    if self.grad_fn is not None: # not a leaf
        grads = self.grad_fn.backward(gradient) # call the operation backward
        for tensor, grad in zip(self.grad_fn.input, grads):
            if isinstance(tensor, Tensor):
                tensor.backward(grad) # recursively call the backward again for the gr
```

Finally, just implement `.zero_grad()` to zero the gradient of a tensor, and `.detach()` to remove the tensor autograd history:

```
# norch/tensor.py
```

```
    def zero_grad(self):
        self.grad = None

    def detach(self):
        self.grad = None
        self.grad_fn = None
```

Congratulations! You just created a complete tensor library with GPU support and automatic differentiation! Now we can create the `nn` and `optim` modules to train some deep learning models more easily.

#4 — nn and optim modules

The `nn` is a module for building neural networks and deep learning models and `optim` is related to optimization algorithms to train these models. In order to recreate them, the first thing to do is implementing a Parameter, which simply is a trainable tensor, with the same operations, but with `requires_grad` set always as `True` and with some random initialization technique.

```
# norch/nn/parameter.py

from norch.tensor import Tensor
from norch.utils import utils
import random

class Parameter(Tensor):
    """
    A parameter is a trainable tensor.
    """
    def __init__(self, shape):
        data = utils.generate_random_list(shape=shape)
        super().__init__(data, requires_grad=True)
```

```
# norch/utisl/utils.py

def generate_random_list(shape):
    """
    Generate a list with random numbers and shape 'shape'
    [4, 2] --> [[rand1, rand2], [rand3, rand4], [rand5, rand6], [rand7, rand8]]
    """
    if len(shape) == 0:
        return []
    else:
        inner_shape = shape[1:]
        if len(inner_shape) == 0:
            return [random.uniform(-1, 1) for _ in range(shape[0])]
        else:
            return [generate_random_list(inner_shape) for _ in range(shape[0])]
```

By using parameters, we can start contructing modules:

```
# norch/nn/module.py

from .parameter import Parameter
from collections import OrderedDict
from abc import ABC
import inspect

class Module(ABC):
    """
    Abstract class for modules
    """
    def __init__(self):
        self._modules = OrderedDict()
        self._params = OrderedDict()
        self._grads = OrderedDict()
        self.training = True

    def forward(self, *inputs, **kwargs):
        raise NotImplementedError

    def __call__(self, *inputs, **kwargs):
        return self.forward(*inputs, **kwargs)

    def train(self):
```

```
        self.training = True
        for param in self.parameters():
            param.requires_grad = True

    def eval(self):
        self.training = False
        for param in self.parameters():
            param.requires_grad = False

    def parameters(self):
        for name, value in inspect.getmembers(self):
            if isinstance(value, Parameter):
                yield self, name, value
            elif isinstance(value, Module):
                yield from value.parameters()

    def modules(self):
        yield from self._modules.values()

    def gradients(self):
        for module in self.modules():
            yield module._grads

    def zero_grad(self):
        for _, _, parameter in self.parameters():
            parameter.zero_grad()

    def to(self, device):
        for _, _, parameter in self.parameters():
            parameter.to(device)

        return self

    def inner_repr(self):
        return ""

    def __repr__(self):
        string = f"{self.get_name()}("
        tab = "    "
        modules = self._modules
        if modules == {}:
            string += f'\n{tab}(parameters): {self.inner_repr()}'
        else:
            for key, module in modules.items():
                string += f"\n{tab}({key}): {module.get_name()}({module.inner_repr()})
        return f'{string}\n)'

    def get_name(self):
        return self.__class__.__name__
```

```
    def __setattr__(self, key, value):
        self.__dict__[key] = value

        if isinstance(value, Module):
            self._modules[key] = value
        elif isinstance(value, Parameter):
            self._params[key] = value
```

For example, we can construct our custom modules by inheriting from `nn.Module`, or we can use some previously created modules, such as the `linear`, which implements the $\boldsymbol{y} = W\boldsymbol{x} + b$ operation.

```
# norch/nn/modules/linear.py

from ..module import Module
from ..parameter import Parameter

class Linear(Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.weight = Parameter(shape=[self.output_dim, self.input_dim])
        self.bias = Parameter(shape=[self.output_dim, 1])

    def forward(self, x):
        z = self.weight @ x + self.bias
        return z

    def inner_repr(self):
        return f"input_dim={self.input_dim}, output_dim={self.output_dim}, " \
               f"bias={True if self.bias is not None else False}"
```

Now we can implement some loss and activation functions. For instance, a mean squared error loss and a sigmoid function:

```
# norch/nn/loss.py

from .module import Module

class MSELoss(Module):
    def __init__(self):
      pass

    def forward(self, predictions, labels):
        assert labels.shape == predictions.shape, \
            "Labels and predictions shape does not match: {} and {}".format(labels.sha

        return ((predictions - labels) ** 2).sum() / predictions.numel

    def __call__(self, *inputs):
        return self.forward(*inputs)
```

```
# norch/nn/activation.py

from .module import Module
import math

class Sigmoid(Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 1.0 / (1.0 + (math.e) ** (-x))
```

Finally, create the optimizers. On our example I will implement the Stochastic Gradient Descent algorithm:

```
# norch/optim/optimizer.py

from abc import ABC
from norch.tensor import Tensor
```

```
class Optimizer(ABC):
    """
    Abstract class for optimizers
    """

    def __init__(self, parameters):
        if isinstance(parameters, Tensor):
            raise TypeError("parameters should be an iterable but got {}".format(type
        elif isinstance(parameters, dict):
            parameters = parameters.values()

        self.parameters = list(parameters)

    def step(self):
        raise NotImplementedError

    def zero_grad(self):
        for module, name, parameter in self.parameters:
            parameter.zero_grad()


class SGD(Optimizer):
    def __init__(self, parameters, lr=1e-1, momentum=0):
        super().__init__(parameters)
        self.lr = lr
        self.momentum = momentum
        self._cache = {'velocity': [p.zeros_like() for (_, _, p) in self.parameters]}

    def step(self):
        for i, (module, name, _) in enumerate(self.parameters):
            parameter = getattr(module, name)

            velocity = self._cache['velocity'][i]

            velocity = self.momentum * velocity - self.lr * parameter.grad

            updated_parameter = parameter + velocity

            setattr(module, name, updated_parameter)

            self._cache['velocity'][i] = velocity

            parameter.detach()
            velocity.detach()
```

And, that's it! We just created our own deep learning framework! 🥳

Let's do some training:

```
import norch
import norch.nn as nn
import norch.optim as optim
import random
import math

random.seed(1)

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(1, 10)
        self.sigmoid = nn.Sigmoid()
        self.fc2 = nn.Linear(10, 1)

    def forward(self, x):
        out = self.fc1(x)
        out = self.sigmoid(out)
        out = self.fc2(out)

        return out

device = "cuda"
epochs = 10

model = MyModel().to(device)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
loss_list = []

x_values = [0. ,  0.4,  0.8,  1.2,  1.6,  2. ,  2.4,  2.8,  3.2,  3.6,  4. ,
        4.4,  4.8,  5.2,  5.6,  6. ,  6.4,  6.8,  7.2,  7.6,  8. ,  8.4,
        8.8,  9.2,  9.6, 10. , 10.4, 10.8, 11.2, 11.6, 12. , 12.4, 12.8,
       13.2, 13.6, 14. , 14.4, 14.8, 15.2, 15.6, 16. , 16.4, 16.8, 17.2,
       17.6, 18. , 18.4, 18.8, 19.2, 19.6, 20.]

y_true = []
for x in x_values:
    y_true.append(math.pow(math.sin(x), 2))
```

```python
for epoch in range(epochs):
    for x, target in zip(x_values, y_true):
        x = norch.Tensor([[x]]).T
        target = norch.Tensor([[target]]).T

        x = x.to(device)
        target = target.to(device)

        outputs = model(x)
        loss = criterion(outputs, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss[0]:.4f}')
    loss_list.append(loss[0])

# Epoch [1/10], Loss: 1.7035
# Epoch [2/10], Loss: 0.7193
# Epoch [3/10], Loss: 0.3068
# Epoch [4/10], Loss: 0.1742
# Epoch [5/10], Loss: 0.1342
# Epoch [6/10], Loss: 0.1232
# Epoch [7/10], Loss: 0.1220
# Epoch [8/10], Loss: 0.1241
# Epoch [9/10], Loss: 0.1270
# Epoch [10/10], Loss: 0.1297
```

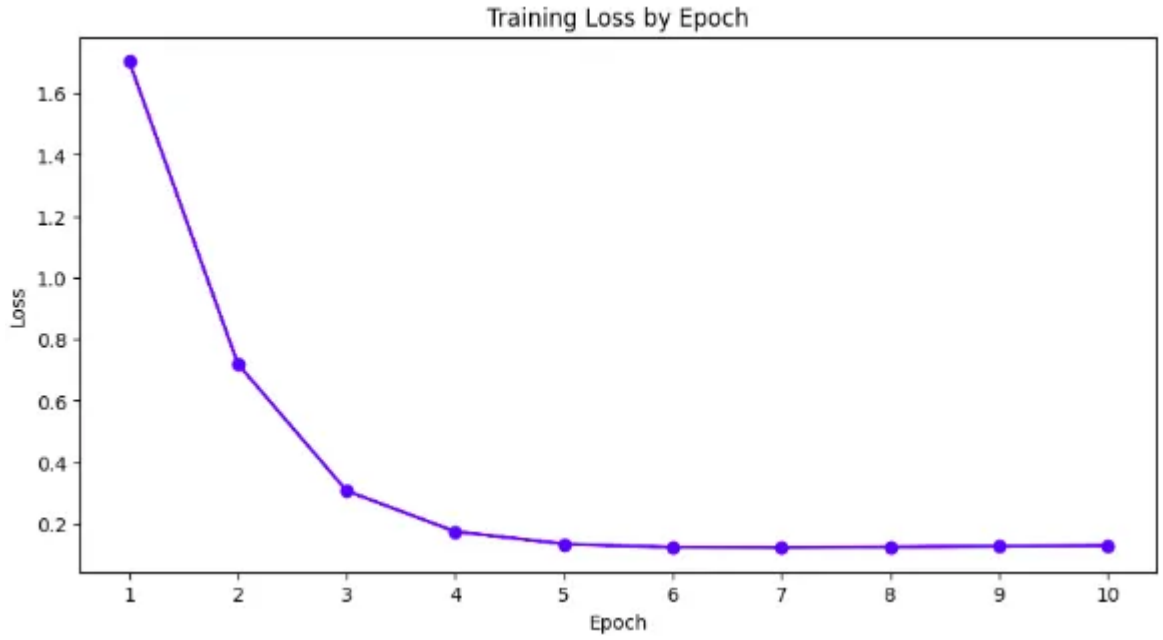


Image by Author

The model was successfully created and trained using our custom deep learning framework!

You can check the complete code here.

Conclusion

In this post we covered the basic concepts of what is a tensor, how it is modeled and more advanced topics such as CUDA and Autograd. We successfully created a deep learning framework with GPU support and automatic differentiation. I hope this post helped you to briefly understand how PyTorch works under the hood.

In future posts, I will try to cover more advanced topics such as distributed training (multinode / multi GPU) and memory management. Please let me know what you think or what you would like me to write about next in the comments! Thanks so much for reading! 😊