Every value that is 0 or below is giving value 0 if we use the standard *rectifier* function. This can cause a problem for networks that have negative values for weights (vanishing gradient). This situation is not happening in nature, of course, but in this world it's common. This means that a lot of values will be 0 and our network will not update during the learning process. So, we use the *Leaky ReLU* function, which will never come across this problem. Mathematically, it is expressed like this: $f(x)=1(x<0)(\alpha x)+1(x>=0)(x)f(x)=1(x<0)(\alpha x)+1(x>=0)(x)$.

6.2 Intro to Pytorch

Elon Musk made it clear that he doesn't like *Facebook*, however, *Tesla* still uses one piece of **technology** that comes from Zukenberg's laboratory. We can bet that it made its way into *SpaceX* as well and that it was an important part of the recent launch too. This technology has a big advocate in the AI world and that is Yan LeCunn, father of modern Convolutional Neural Networks. That is correct, we are talking about **PyTorch**. This deep learning framework is extremely popular among the research community and we can witness that in the past couple of years, it gained popularity in the industry as well. In this chapter, we explore the basic principles of *PyTorch* and how they can be used for simple tasks.

Generally speaking, *PyTorch* as a tool has two big **goals**. The first one is to be *NumPy* for **GPUs**. This doesn't mean that *NumPy* is a bad tool; it just means that it doesn't utilize the power of GPUs. The second goal of *PyTorch* is to be a **deep learning** framework that provides speed and flexibility. That is why it is so popular in the research community; it provides a platform in which users can quickly perform experiments. Apart from that, building models in *PyTorch* is very easy.

6.2.1 Installation

Installing *PyTorch* is quite easy and you can create a command that you need to run for the installation on **this page**. All you need to do is enter your environment details and run the command. Here is how I've set it up for my machine:

To verify that you have installed it correctly, run *Jupyter Notebook* and try to import *torch*:

```
import torch

torch.__version__
'1.5.0'
```

6.2.2 Tensors and Gradients

In general, a lot of concepts in machine learning and deep learning can be abstracted using multi-dimensional matrices – **tensors**. Mathematically speaking, tensors are described as geometric objects that describe linear relationships between other geometric objects. For the simplification of all concepts, tensors can be observed as **multi-dimensional** arrays of data. When we observe them like n-dimensional arrays, we can apply matrix operations easily and effectively. That is what *PyTorch* is actually doing. In this framework, a tensor is a **primitive unit** used to model scalars, vectors, and matrices located in the central class of the package *torch.Tensor*. We can do various operations with tensors, but first, let's see how we can create and initialize them.

6.2.3 Creating and Initializing Tensors

Here is how we can define scalar, vector, and matrix:

```
# Scalar
tensor_1 = torch.tensor(11.)
print('------ Scalar ------')
print(tensor_1)
print(tensor_1.dtype)
print(tensor_1.shape)
print('\n')

# Vector
tensor_2 = torch.tensor([11, 23, 9, 33])
print('------ Vector ------')
print(tensor_2)
print(tensor_2.dtype)
print(tensor_2.shape)
print('\n')

# Matrix
tensor_3 = torch.tensor([[5., 6],
```

```
                  [7, 4],
                  [9, 11]])

print('------ Matrix ------')
print(tensor_3)
print(tensor_3.dtype)
print(tensor_3.shape)
print('\n')

# 3D Array
tensor_4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])

print('------ 3D Array ------')
print(tensor_4)
print(tensor_4.dtype)
print(tensor_4.shape)
print('\n')
```

In the code snippet above, we created a scalar, vector, matrix and 3D array. Note that for vectors, we use integers. For matrix and scalar we use floats. Also, we printed out their data types and shapes. Here is the output:

```
------ Scalar ------
tensor(11.)
torch.float32
torch.Size([])


------ Vector ------
tensor([11, 23,  9, 33])
torch.int64
torch.Size([4])


------ Matrix ------
tensor([[ 5.,  6.],
        [ 7.,  4.],
        [ 9., 11.]])
torch.float32
```

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


------ 3D Array ------
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
torch.float32
torch.Size([2, 2, 3])
```

As you can see, it is quite easy to create all the types of data that we need quickly. There are various tensor initialization functions. For example, you can create an uninitialized tensor, or a tensor initialized with zeros, ones or with a random value:

```
# Uninitialized (does not contain definite known values before it is used.)
tensor = torch.empty(2, 3)
print(tensor)

# Tensor with all zeroes
tensor = torch.zeros(2, 3)
print(tensor)

# Tensor with all ones
tensor = torch.ones(2, 3)
print(tensor)

# Tensor with randomly initialized values
tensor = torch.rand(2, 3)
print(tensor)
```

```
tensor([[1.0286e-38, 9.0919e-39, 9.3674e-39],
        [9.2755e-39, 1.4013e-43, 0.0000e+00]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0.1093, 0.3990, 0.5312],
        [0.2732, 0.6488, 0.4613]])
```