Deep Learning with PyTorch Workshop

CHAPTER 2: INTRODUCTION TO PYTORCH

BY MUHAMMAD RAJABINASAB

BASED ON THE DEEP LEARNING WITH PYTORCH WORKSHOP

BY HYATT SALEH

Prerequisites

Python 3 and pip.

PyTorch Library, can be installed using pip:

```
pip install torch
```

Jupyter Lab, can be installed using pip:

```
pip install jupyterlab
```

Then you can run it using cmd/terminal:

```
jupyter-lab
```

About PyTorch

PyTorch was first released to the public in January 2017.

The library has a C++ backend, combined with the deep learning framework of Torch, which allows much faster computations than native Python libraries.

The frontend is in Python, which has helped it gain popularity, enabling data scientists new to the library to construct complex neural networks.

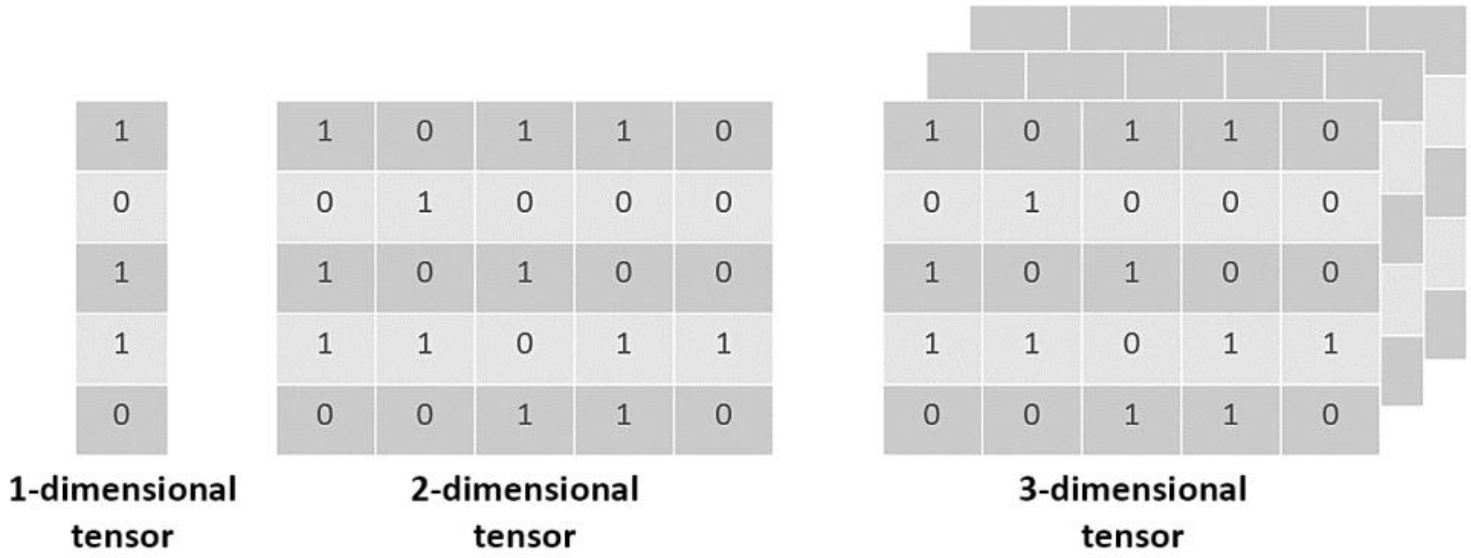
It is possible to use PyTorch alongside other popular Python packages.

What Are Tensors?

PyTorch uses tensors to represent data.

Tensors are matrix-like structures of n dimensions.

For tensors, dimensions are also known as ranks.



What Are Tensors?

Defining a one-dimensional tensor (tensor_1) and a two-dimensional tensor (tensor_2) in PyTorch can be achieved using the following code snippet:

```
tensor_1 = torch.tensor([1,1,0,2])
tensor_2 = torch.tensor([[0,0,2,1,2],[1,0,2,2,0]])
```

From the preceding snippet, the first tensor would have a size of 4 for one dimension, while the second one would have a size of 5 for each of the two dimensions, which can be verified by making use of the shape property over the tensor variables, as seen here:

```
tensor_1.shape
```
The output is **`torch.Size([4])`**.

```
tensor_2.shape
```
The output is **`torch.Size([2],[5])`**.

What Are Tensors?

When using a GPU-enabled machine, the following modification is implemented to define a tensor:

```
tensor = torch.tensor([1,1,0,2]).cuda()
```

Creating dummy data using PyTorch tensors is fairly simple, similar to what you would do in NumPy. For instance, torch.randn() returns a tensor filled with random numbers of the dimensions specified within the parentheses, while torch. randint() returns a tensor filled with integers (the minimum and maximum values can be defined) of the dimensions defined within the parentheses:

```
example_1 = torch.randn(3,3)
example_2 = torch.randint(low=0, high=2, \
size=(3,3)).type(torch.FloatTensor)
```

Exercise 1.01: Creating Tensors of Different Ranks Using PyTorch

In this exercise, we will use the PyTorch library to create tensors of ranks one, two, and three.

Advantages of Using PyTorch

- **Ease of use:** With respect to the API, PyTorch has a simple interface that makes it easy to develop and run models. Many early adopters consider it to be more intuitive than other libraries, such as TensorFlow.

- **Speed:** The use of GPUs enables the library to train faster than other deep learning libraries. This is especially useful when different approximations have to be tested in order to achieve the best possible model. Additionally, even though other libraries may also have the option to accelerate computations with GPUs, you can do this in PyTorch by typing just a couple of simple lines of code.

Advantages of Using PyTorch

- **Convenience:** PyTorch is flexible. It uses dynamic computational graphs that allow you to make changes to networks on the go. It also allows great flexibility when building the architecture as it is easy to make adjustments to conventional architectures.

- **Imperative:** PyTorch is also imperative. Each line of code is executed individually, allowing you to track the model in real time, as well as debug the model in a convenient way.

- **Pretrained models:** Finally, it contains many pretrained models that are easy to use and are a great starting point for some data problems.

Key Elements of PyTorch

Like any other library, PyTorch has a variety of modules, libraries, and packages for developing different functionalities.

In this section, the three most commonly used elements for building deep neural networks will be explained, along with a simple example of the syntax.

The PyTorch autograd Library

The autograd library consists of a technique called automatic differentiation.

Its purpose is to numerically calculate the derivative of a function.

his is crucial for a concept we will learn about in the next chapter called backward propagation, which is carried out while training a neural network.

The derivative (also known as the gradient) of an element refers to the rate of change of that element in a given time step.

In deep learning, gradients refer to the dimension and magnitude in which the parameters of the neural network must be updated in a training step in order to minimize the loss function.

The PyTorch autograd Library

To compute the gradients, simply call the backward() function, as shown here:

```
a = torch.tensor([5.0, 3.0], requires_grad=True)
b = torch.tensor([1.0, 4.0])
ab = ((a + b) ** 2).sum()
ab.backward()
```

By printing the gradients for both a and b, it is possible to confirm that they were only calculated for the first variable (a), while for the second one (b), it throws an error:

```
print(a.grad.data)
```

The output is **`tensor([12., 14.])`**.

```
print(b.grad.data)
```

The output is as follows:

```
AttributeError: 'NoneType' object has no attribute 'data'
```

The PyTorch nn Module

The nn module is a complete PyTorch module used to create and train neural networks, which, through the use of different elements, allows for simple and complex developments.

For instance, the Sequential() container allows for the easy creation of network architectures that follow a sequence of predefined modules (or layers) without the need for much knowledge of defining network architectures.

This module also has the capability to define the loss function to evaluate the model and many more advanced features that will be discussed.

The PyTorch nn Module

The process of building a neural network architecture as a sequence of predefined modules can be achieved in just a couple of lines, as shown here:

```
 import torch.nn as nn
 model = nn.Sequential(nn.Linear(input_units,
hidden_units), \
 nn.ReLU(), \
 nn.Linear(hidden_units, output_units), \
 nn.Sigmoid())
 loss_funct = nn.MSELoss()
```

Exercise 1.02: Defining a Single-Layer Architecture

In this exercise, we will use PyTorch's nn module to define a model for a singlelayer neural network, and also define the loss function to evaluate the model.

This will be the starting point so that you will be able to build more complex network architectures to solve real-life data problems.

The PyTorch optim Package

The optim package is used to define the optimizer that will be used to update the parameters in each iteration (which will be further explained in the following chapters) using the gradients calculated by the autograd module.

Here, it is possible to choose from different optimization algorithms that are available, such as Adam, Stochastic Gradient Descent (SGD), and Root Mean Square Propagation (RMSprop), among others.

To set the optimizer to be used, the following line of code shall suffice, after importing the package:

```
optimizer = torch.optim.SGD(model.parameters(),
lr=0.01)
```

The PyTorch optim Package

Next, the process of running the optimization for 100 iterations is shown here, which, as you can see, uses the model created by the nn module and the gradients calculated by the autograd library:

```
for i in range(100):
  # Call to the model to perform a prediction
  y_pred = model(x)
  # Calculation of loss function based on y_pred and y
  loss = loss_funct(y_pred, y)
  # Zero the gradients so that previous ones don't accumulate
  optimizer.zero_grad()
  # Calculate the gradients of the loss function
  loss.backward()
  """
  Call to the optimizer to perform an update
  of the parameters
  """
  optimizer.step()
```

Exercise 1.03: Training a Neural Network

In this exercise, we will learn how to train the single-layer network from the previous exercise, using PyTorch's optim package.

Considering that we will use dummy data as input, training the network won't solve a data problem, but it will be performed for learning purposes.

Activity 1.01: Creating a Single-Layer Neural Network

You work as an assistant of the mayor of Somerville and the HR department has asked you to build a model capable of predicting whether a person is happy with the current administration based on their satisfaction with the city's services. To do so, you have decided to build a single-layer neural network using PyTorch, using the response of previous surveys.

Summary

In this chapter, we learned about PyTorch Basics.

We worked with tensors and basic PyTorch modules.

We created and trained our first neural network.

Finally, we worked on a simple real life problem and managed to solve it using a simple neural network in PyTorch.

Thanks For Your Attention!

Feel free to ask any questions