# PyTorch Basics: Tensors & Gradients

## Part 1 of "Deep Learning with Pytorch: Zero to GANs"

This tutorial series is a hands-on beginner-friendly introduction to deep learning using PyTorch, an open-source neural networks library. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

1. PyTorch Basics: Tensors & Gradients
2. Gradient Descent & Linear Regression
3. Working with Images & Logistic Regression
4. Training Deep Neural Networks on a GPU
5. Image Classification using Convolutional Neural Networks
6. Data Augmentation, Regularization and ResNets
7. Generating Images using Generative Adversarial Networks

If you're just getting started with data science and deep learning, then this tutorial series is for you. All you need to know is a bit of Python programming (functions, loops, classes, etc.) and some high school math (vectors, matrices, derivatives, and probability). We'll cover all the mathematical and theoretical concepts we need as we go along.

This tutorial covers the following topics:

- Introductions to PyTorch tensors
- Tensor operations and gradients
- Interoperability between PyTorch and Numpy
- How to use the PyTorch documentation site

# How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian (don't worry if these terms seem unfamiliar; we'll learn more about them soon). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

### Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. You can also select "Run on Binder" or "Run on Kaggle" if you face issues running the notebook on Google Colab.

### Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](), download the notebook and install the required libraries. We recommend using the [Conda]() distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This tutorial is a [Jupyter notebook]() - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc. instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

We begin by installing and importing the required libraries.

```
!pip install torch numpy --quiet
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 100.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 76.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 45.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 5.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 12.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 7.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 5.9 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 68.6 MB/s eta 0:00:00
```

```
import torch
```

## Tensors

At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix, or any n-dimensional array. Let's create a tensor with a single number.

```
# Number
t1 = torch.tensor(4.)
t1
```

```
tensor(4.)
```

`4.` is a shorthand for `4.0`. It is used to indicate to Python (and PyTorch) that you want to create a floating-point number. We can verify this by checking the `dtype` attribute of our tensor.

```
t1.dtype
```

```
torch.float32
```

Let's try creating more complex tensors.

```
# Vector
t2 = torch.tensor([1., 2, 3, 4])
t2
```

```
tensor([1., 2., 3., 4.])
```

```
# Matrix
t3 = torch.tensor([[5., 6],
                   [7, 8],
                   [9, 10]])
t3
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
```

```
# 3-dimensional array
t4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])
t4
```

```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

Tensors can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of a tensor.

```
print(t1)
t1.shape
```

```
tensor(4.)
```

```
torch.Size([])
```

```
print(t2)
t2.shape
```

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

```
print(t3)
t3.shape
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
torch.Size([3, 2])
```

```
print(t4)
t4.shape
```

```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

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

## Tensor operations and gradients

We can combine tensors with the usual arithmetic operations. Let's look at an example:

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

```
(tensor(3.), tensor(4., requires_grad=True), tensor(5., requires_grad=True))
```

We've created three tensors: `x`, `w`, and `b`, all numbers. `w` and `b` have an additional parameter `requires_grad` set to `True`. We'll see what it does in just a moment.

Let's create a new tensor `y` by combining these tensors.

```
# Arithmetic operations
y = w * x + b
y
```

```
tensor(17., grad_fn=<AddBackward0>)
```

As expected, `y` is a tensor with the value `3 * 4 + 5 = 17`. What makes PyTorch unique is that we can automatically compute the derivative of `y` w.r.t. the tensors that have `requires_grad` set to `True` i.e. w and b. This feature of PyTorch is called *autograd* (automatic gradients).

To compute the derivatives, we can invoke the `.backward` method on our result `y`.

```
# Compute derivatives
y.backward()
```

The derivatives of `y` with respect to the input tensors are stored in the `.grad` property of the respective tensors.

```
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

As expected, `dy/dw` has the same value as `x`, i.e., `3`, and `dy/db` has the value `1`. Note that `x.grad` is `None` because `x` doesn't have `requires_grad` set to `True`.

The "grad" in `w.grad` is short for *gradient*, which is another term for derivative. The term *gradient* is primarily used while dealing with vectors and matrices.

## Interoperability with Numpy

Numpy is a popular open-source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays and has a vast ecosystem of supporting libraries, including:

- Pandas for file I/O and data analysis
- Matplotlib for plotting and visualization
- OpenCV for image and video processing

If you're interested in learning more about Numpy and other data science libraries in Python, check out this tutorial series: https://jovian.ai/aakashns/python-numerical-computing-with-numpy .

Instead of reinventing the wheel, PyTorch interoperates well with Numpy to leverage its existing ecosystem of tools and libraries.

Here's how we create an array in Numpy:

```
import numpy as np

x = np.array([[1, 2], [3, 4.]])
x
```

```
array([[1., 2.],
       [3., 4.]])
```

We can convert a Numpy array to a PyTorch tensor using `torch.from_numpy`.

```
# Convert the numpy array to a torch tensor.
y = torch.from_numpy(x)
y
```

```
tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

Let's verify that the numpy array and torch tensor have similar data types.

```
x.dtype, y.dtype
```

```
(dtype('float64'), torch.float64)
```

We can convert a PyTorch tensor to a Numpy array using the `.numpy` method of a tensor.

```
# Convert a torch tensor to a numpy array
z = y.numpy()
z
```

```
array([[1., 2.],
       [3., 4.]])
```

The interoperability between PyTorch and Numpy is essential because most datasets you'll work with will likely be read and preprocessed as Numpy arrays.

You might wonder why we need a library like PyTorch at all since Numpy already provides data structures and utilities for working with multi-dimensional numeric data. There are two main reasons:

1. **Autograd**: The ability to automatically compute gradients for tensor operations is essential for training deep learning models.
2. **GPU support**: While working with massive datasets and large models, PyTorch tensor operations can be performed efficiently using a Graphics Processing Unit (GPU). Computations that might typically take hours can be completed within minutes using GPUs.

We'll leverage both these features of PyTorch extensively in this tutorial series.

## Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. Jovian offers an easy way of saving and sharing your Jupyter notebooks online.

First, you need to install the Jovian python library if it isn't already installed.

```
!pip install jovian --upgrade --quiet
```

```
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.6/68.6 kB 1.9 MB/s eta 0:00:00
  Building wheel for uuid (setup.py) ... done
```

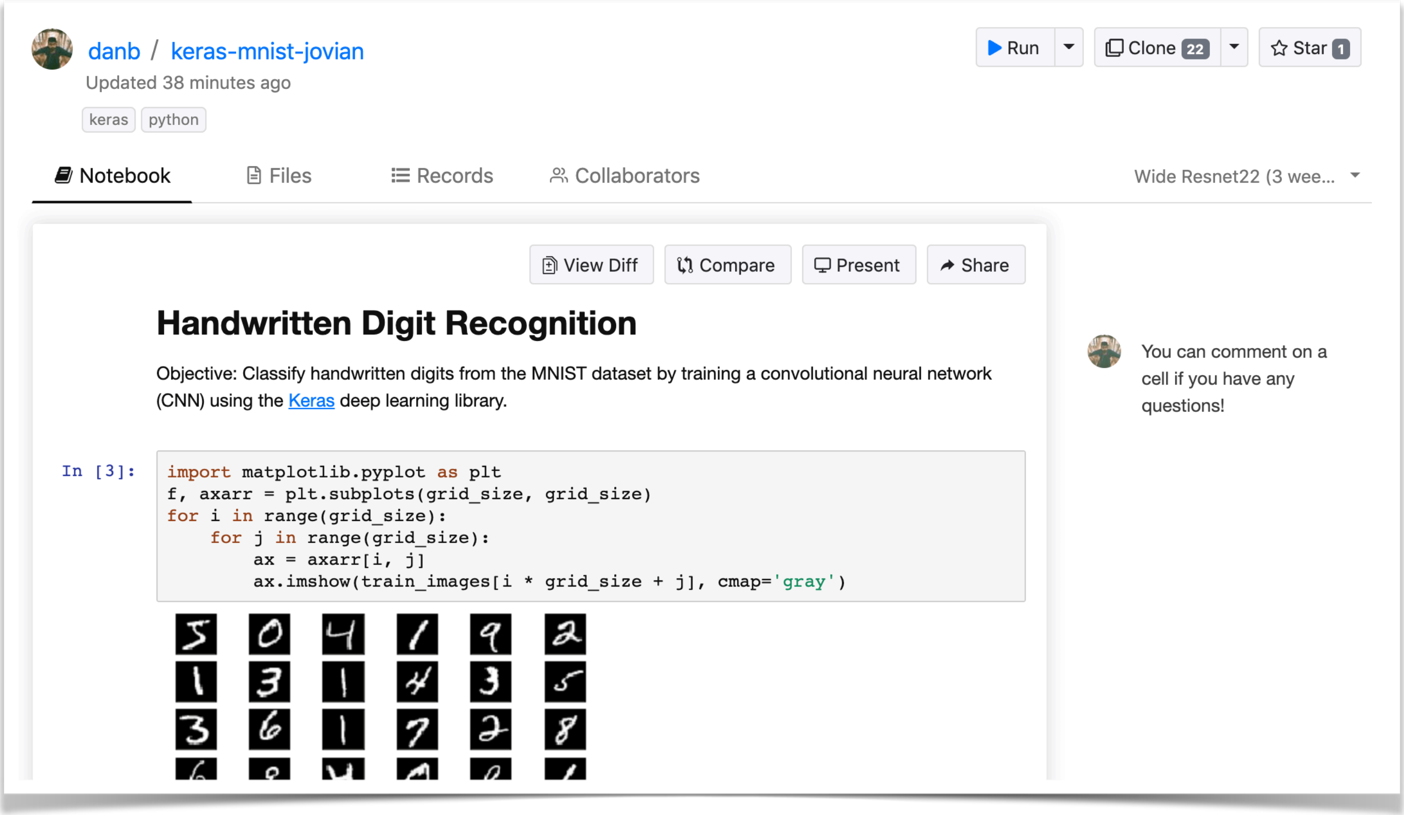
```
import jovian
```

```
jovian.commit(project='01-pytorch-basics')
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

> The first time you run `jovian.commit`, you may be asked to provide an *API Key* to securely upload the notebook to your Jovian account. You can get the API key from your Jovian profile page after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work. Jovian also includes a powerful commenting interface, so you can discuss & comment on specific parts of your notebook:




You can do a lot more with the `jovian` Python library. Visit the documentation site to learn more: https://jovian.ai/docs/index.html

## Summary and Further Reading

This tutorial covers the following topics:

- Introductions to PyTorch tensors
- Tensor operations and gradients
- Interoperability between PyTorch and Numpy

Tensors in PyTorch support various operations, and what we've covered here is by no means exhaustive. You can learn more about tensors and tensor operations here: https://pytorch.org/docs/stable/tensors.html.

If you're interested, you can learn more about matrix derivatives on Wikipedia (although it's not necessary for following along with this series of tutorials): https://en.wikipedia.org/wiki/Matrix_calculus#Derivatives_with_matrices .

The material in this series is inspired by PyTorch Tutorial for Deep Learning Researchers by Yunjey Choi and FastAI development notebooks by Jeremy Howard.

With this, we complete our discussion of tensors and gradients in PyTorch, and we're ready to move on to the next topic: Gradient Descent & Linear Regression.