PyTorch

# Introduction To PyTorch Part 1


> Hamza Fatnaoui

> Hicham Ibn Issaghyr

> Ikram Arif

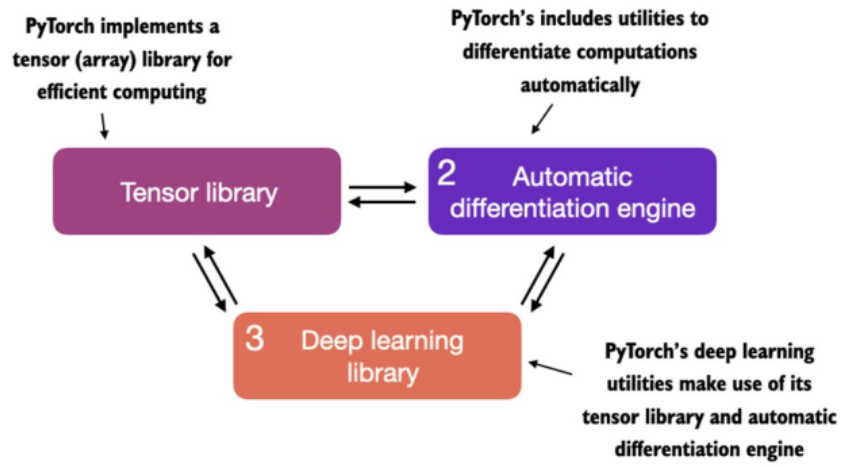

## A.1 What is PyTorch

PyTorch is an open-source Python-based deep learning library that has become the most widely used tool for research in deep learning since 2019, according to data from Papers With Code. Its popularity is also reflected in the Kaggle Data Science and Machine Learning Survey 2022, where approximately 40% of respondents reported using PyTorch, a number that continues to grow each year.

PyTorch is favored for its user-friendly interface, making it accessible to beginners while also offering the flexibility to advanced users to customize and optimize their models at a lower level. This balance between ease of use and advanced functionality makes PyTorch a preferred choice for both practitioners and researchers in the field of machine learning and deep learning.

### A.1.1 The three core components of PyTorch

PyTorch is a relatively comprehensive library, and one way to approach it is to focus on its three broad components




- Firstly, PyTorch is a tensor library that extends the concept of array-oriented programming library NumPy with the additional feature of accelerated computation on GPUs, thus providing a seamless switch between CPUs and GPUs.
- Secondly, PyTorch is an automatic differentiation engine, also known as autograd, which enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization.
- Finally, PyTorch is a deep learning library, meaning that it offers modular, flexible, and efficient building blocks (including pre-trained models, loss functions, and optimizers) for designing and training a wide range of deep learning models, catering to both researchers and developers.

## A.1.2 Installing PyTorch

PyTorch can be installed just like any other Python library or package. However, since PyTorch is a comprehensive library featuring CPU- and GPU-compatible codes, the installation may require additional explanation.

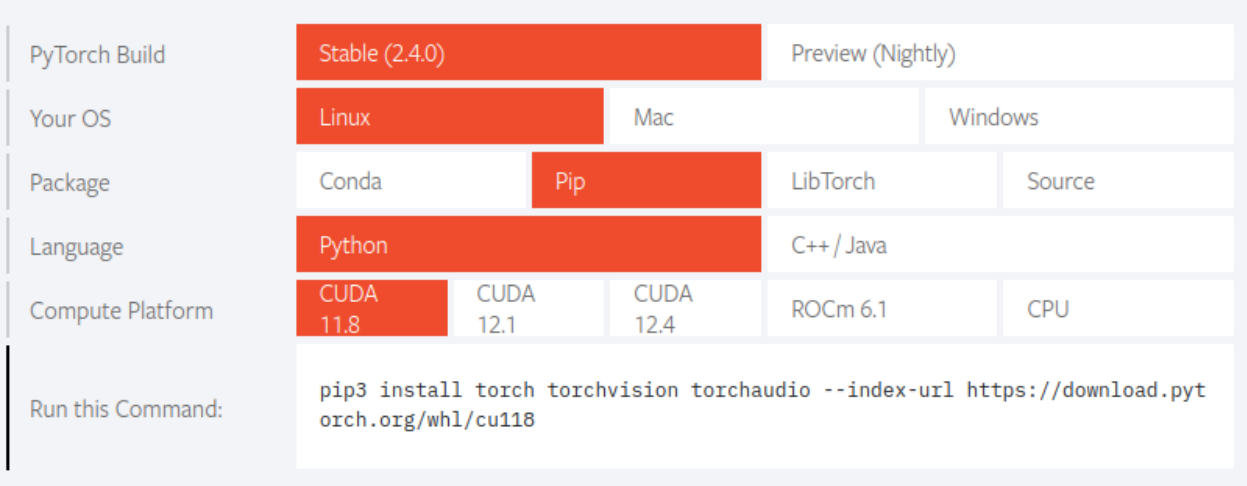
There are two versions of PyTorch: a leaner version that only supports CPU computing and a version that supports both CPU and GPU computing. If your machine has a CUDA-compatible GPU that can be used for deep learning (ideally

an NVIDIA T4, RTX 2080 Ti, or newer), I recommend installing the GPU version. Regardless, the default command for installing PyTorch is as follows

```
[1] pip install -q torch  # q is used to reduce the output during the installation steps.
```

Suppose your computer supports a CUDA-compatible GPU. In that case, this will automatically install the PyTorch version that supports GPU acceleration via CUDA, given that the Python environment you're working on has the necessary dependencies (like pip) installed.

However, to explicitly install the CUDA-compatible version of PyTorch, it's often better to specify the CUDA you want PyTorch to be compatible with. PyTorch's official website (https://pytorch.org) provides commands to install PyTorch with CUDA support for different operating systems as shown




To specify the version of your PyTorch you can just replace torch by torch==2.0.1 (example)

After importing torch 'import torch' you can check the version

```
# check the version
torch.__version__
```

After installing PyTorch, you can check whether your installation recognizes your built-in NVIDIA GPU by running the following code in Python:

```
import torch
torch.cuda.is_available()
```

If the command returns True, you are all set. If the command returns False,your computer may not have a compatible GPU, or PyTorch does not recognize it.

If you don't have access to a GPU, there are several cloud computing providers where users can run GPU computations against an hourly cost. A popular Jupyter-notebook-like environment is Google Colab (https://colab.research.google.com), which provides time-limited access to GPUs. Or LightningAi, a platfrom that provide 22 hrs GPU monthly.

# A.1.3 Install and set up PyTorch

## A.1.3.1 Download and install Miniforge

- Download miniforge from the GitHub repository here.
- Depending on your operating system, this should download either an `.sh` (macOS, Linux) or `.exe` file (Windows). For the `.sh` file, open your command line terminal and execute the following command

```
(base) fatnaoui@hp-probook-640-g2:~$ sh ~/Desktop/Miniforge3-MacOSX-arm64.sh
```

Next, step through the download instructions, confirming with "Enter".

If you work with many packages, Conda can be slow because of its thorough but complex dependency resolution process and the handling of large package indexes and metadata. To speed up Conda, you can use the following setting, which switches to a more efficient Rust reimplementation for solving dependencies:

```
(base) fatnaoui@hp-probook-640-g2:~$ conda config --set solver libmamba
```

- Create a new virtual environment

```
(base) fatnaoui@hp-probook-640-g2:~$ conda create -n LLMs python=3.8
```

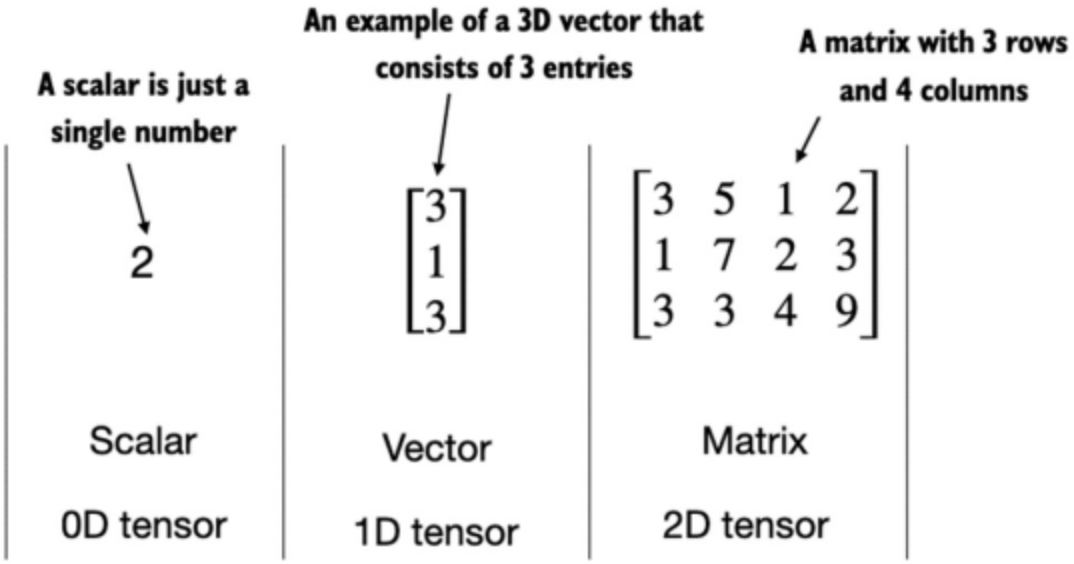
Next, activate your new virtual environment (you have to do it every time you open a new terminal window or tab):

```
(base) fatnaoui@hp-probook-640-g2:~$ conda activate LLMs
(LLMs) fatnaoui@hp-probook-640-g2:~$
```

You can notice that from base enivronment i am now in LLMs environment. You can now install PyTorch with pip or conda in this environment

# A.2 Understanding tensors

Tensors represent a mathematical concept that generalizes vectors and matrices to potentially higher dimensions. In other words, tensors are mathematical objects that can be characterized by their order (or rank), which provides the number of dimensions. For example, a scalar (just a number) is a tensor of rank 0, a vector is a tensor of rank 1, and a matrix is a tensor of rank 2.




From a computational perspective, tensors serve as data containers. For instance, they hold multi dimensional data, where each dimension represents a different

feature. Tensor libraries, such as PyTorch, can create, manipulate, and compute with these multi-dimensional arrays efficiently. In this context, a tensor library functions as an array library.

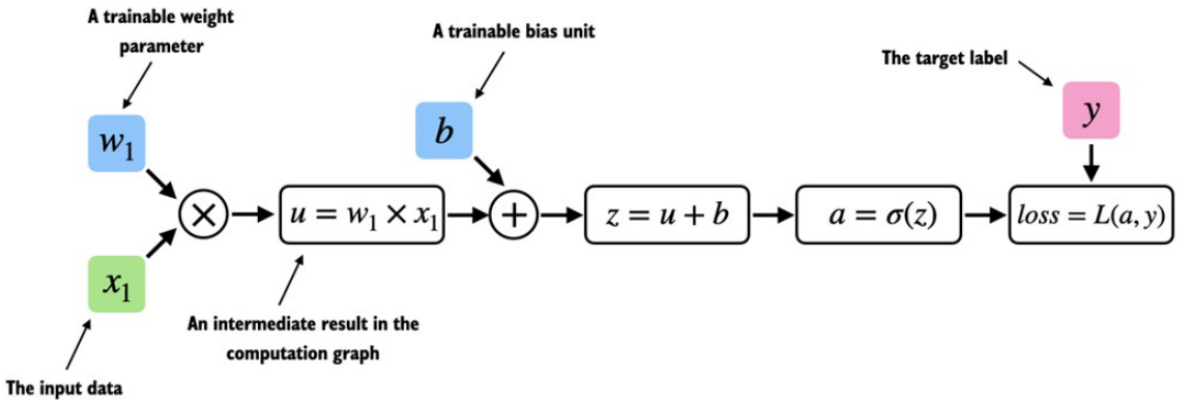
PyTorch tensors are similar to NumPy arrays but have several additional features important for deep learning. For example, PyTorch adds an automatic differentiation engine, simplifying computing gradients. PyTorch tensors also support GPU computations to speed up deep neural network training

### A.2.1 Working with tensors

The following titles will be put into practice in the file linked below, which demonstrates these concepts: Link.

## A.3 Seeing models as computation graphs

A computational graph (or computation graph in short) is a directed graph that allows us to express and visualize mathematical expressions. In the context of deep learning, a computation graph lays out the sequence of calculations needed to compute the output of a neural network -- we use it to compute the required gradients for backpropagation.

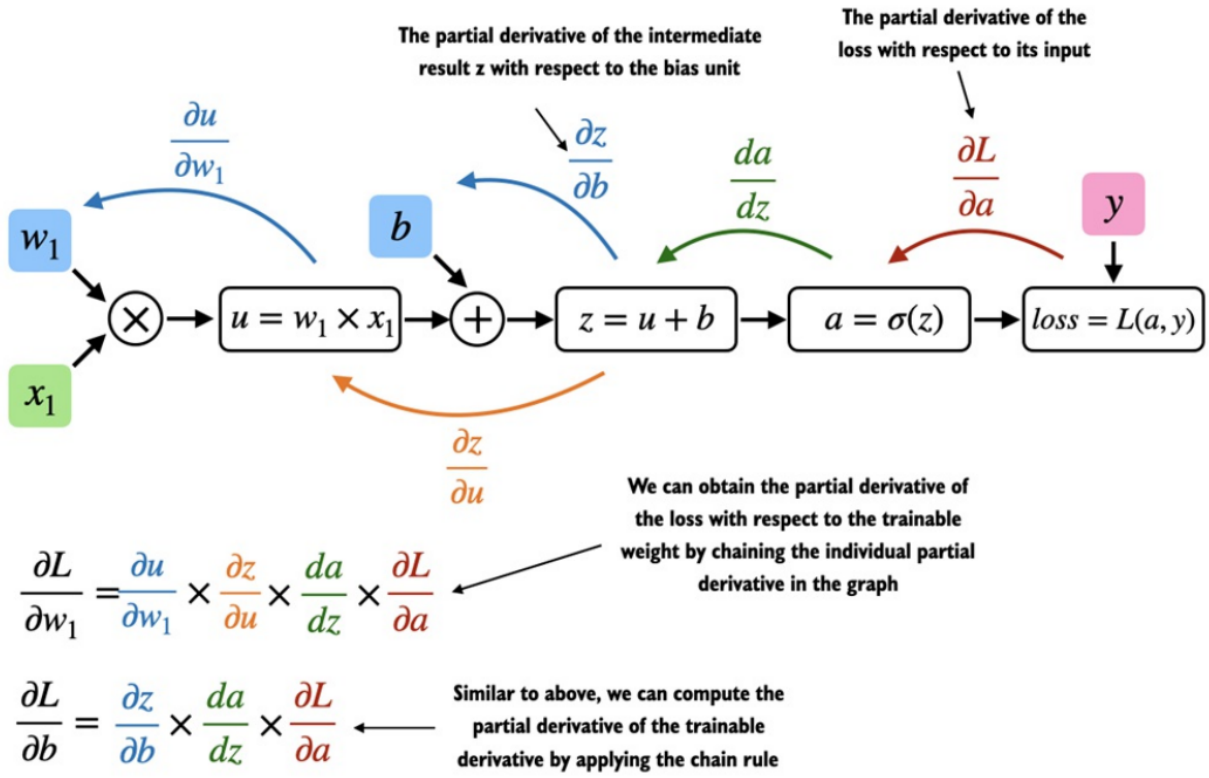


```
import torch.nn.functional as F

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2])
b = torch.tensor([0.0])
z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)
```

PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here w1 and b) to train the model

# A.4 Automatic differentiation made easy

we introduced the concept of computation graphs. If we carry out computations in PyTorch, it will build such a graph internally by default if one of its terminal nodes has the requires_grad attribute set to True. This is useful if we want to compute gradients. Gradients are required when training neural networks via the popular backpropagation algorithm, which can be thought of as an implementation of the chain rule from calculus for neural networks.




Now, how is this all related to the second component of the PyTorch library we mentioned earlier, the automatic differentiation (autograd) engine? By tracking every operation performed on tensors, PyTorch's autograd engine constructs a computational graph in the background. Then, calling the grad function, we can compute the gradient of the loss with respect to model parameter w1 as follows:

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)
z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

Above, we have been using the grad function "manually," which can be useful for experimentation, debugging, and demonstrating concepts. But in practice, PyTorch provides even more high-level tools to automate this process. For instance, we can call .backward on the loss, and PyTorch will compute the gradients of all the leaf nodes in the graph, which will be stored via the tensors' .grad attributes:

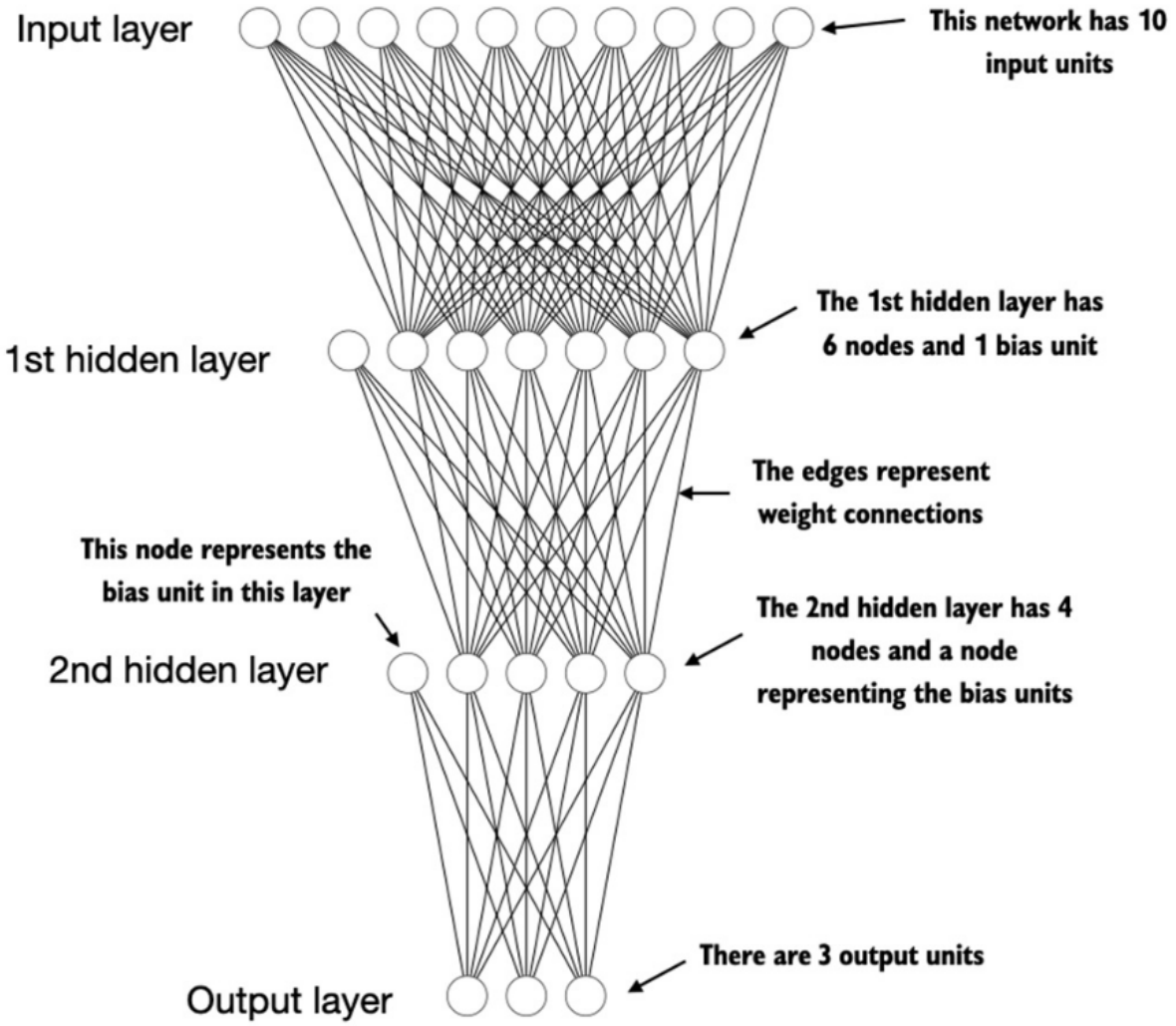
```
loss.backward()
print(w1.grad)
print(b.grad)
```

```
tensor([-0.0898])
tensor([-0.0817])
```

# A.5 Implementing multilayer neural networks

This section focuses on PyTorch as a library for implementing deep neural networks.

To provide a concrete example, we focus on a multilayer perceptron, which is a fully connected neural network




When implementing a neural network in PyTorch, we typically subclass the torch.nn.Module class to define our own custom network architecture. This Module base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.

Within this subclass, we define the network layers in the **init** constructor and specify how they interact in the forward method. The forward method describes how the input data passes through the network and comes together as a computation graph.

In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function with

respect to the model parameters.

The following code implements a classic multilayer perceptron with two hidden layers to illustrate a typical usage of the Module class:

```
class NeuralNetwork(torch.nn.Module):
  def __init__(self, num_inputs, num_outputs):
    super().__init__()
    self.layers = torch.nn.Sequential(

    # 1st hidden layer
    torch.nn.Linear(num_inputs, 30),
    torch.nn.ReLU(),

    # 2nd hidden layer
    torch.nn.Linear(30, 20),
    torch.nn.ReLU(),

    # output layer
    torch.nn.Linear(20, num_outputs),
    )

  def forward(self, x):
    logits = self.layers(x)
    return logits
```

We can then instantiate a new neural network object as follows:

```
model = NeuralNetwork(50, 3)
print(model)
```

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Note that we used the Sequential class when we implemented theNeuralNetwork class. Using Sequential is not required, but it can make our life easier if we have a series of layers that we want to execute in a specific order, as is the case here.

This way, after instantiating self.layers = Sequential(...) in the **init** constructor, we just have to call the self.layers instead of calling each layer individually in the NeuralNetwork's forward method.

Next, let's check the total number of trainable parameters of this model:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

```
Total number of trainable model parameters: 2213
```

Note that each parameter for which requires_grad=True counts as a trainable parameter and will be updated during training

In the case of our neural network model with the two hidden layers above, these trainable parameters are contained in the torch.nn.Linear layers. A linear layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes also referred to as a feedforward or fully connected layer.

Based on the print(model) call we executed above, we can see that the first Linear layer is at index position 0 in the layers attribute. We can access the corresponding weight parameter matrix as follows:

```
print(model.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.1122,  0.1349, -0.0907,  ...,  0.0397, -0.0934, -0.0593],
        [-0.0862,  0.0581, -0.0431,  ..., -0.0396, -0.0551,  0.0548],
        [ 0.0764,  0.0251, -0.1336,  ...,  0.0357,  0.0371, -0.0254],
        ...,
        [ 0.0280,  0.0466,  0.1067,  ...,  0.1157,  0.0712,  0.0148],
        [-0.1183, -0.0376, -0.0912,  ...,  0.0758, -0.0466,  0.0117],
        [ 0.0732,  0.1200,  0.0651,  ...,  0.0393, -0.0432, -0.0534]],
       requires_grad=True)
```

Note that if you execute the code above on your computer, the numbers in the weight matrix will likely differ from those shown above. This is because the model weights are initialized with small random numbers, which are different each time we instantiate the network. In deep learning, initializing model weights with small random numbers is desired to break symmetry during training  - otherwise, the

nodes would be just performing the same operations and updates during backpropagation, which would not allow the network to learn complex mappings from inputs to outputs.

However, while we want to keep using small random numbers as initial values for our layer weights, we can make the random number initialization reproducible by seeding PyTorch's random number generator via manual_seed:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

Now, after we spent some time inspecting the NeuraNetwork instance, let's briefly see how it's used via the forward pass:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

```
tensor([[-0.0879,  0.1729,  0.1534]], grad_fn=<AddmmBackward0>)
```

These three numbers returned above correspond to a score assigned to each of the three output nodes. Notice that the output tensor also includes a grad_fn value.

Here, grad_fn=<AddmmBackward0> represents the last-used function to compute a variable in the computational graph. In particular, grad_fn=<AddmmBackward0> means that the tensor we are inspecting was created via a matrix multiplication and addition operation. PyTorch will use this information when it computes gradients during backpropagation. The <AddmmBackward0> part of grad_fn=<AddmmBackward0> specifies the operation that was performed. In this case, it is an Addmm operation. Addmm stands for matrix multiplication (mm) followed by an addition (Add).

If we just want to use a network without training or backpropagation, forexample, if we use it for prediction after training, constructing this computational graph for

backpropagation can be wasteful as it performs unnecessary computations and consumes additional memory. So, when we use a model for inference (for instance, making predictions) rather than training, it is a best practice to use the torch.no_grad() context manager, as shown below. This tells PyTorch that it doesn't need to keep track of the gradients, which can result in significant savings in memory and computation.

```
with torch.no_grad():
  out = model(X)
print(out)
```

```
tensor([[-0.0879,  0.1729,  0.1534]])
```

In PyTorch, it's common practice to code models such that they return the outputs of the last layer (logits) without passing them to a nonlinear activation function. That's because PyTorch's commonly used loss functions combine the softmax (or sigmoid for binary classification) operation with the negative log-likelihood loss in a single class. The reason for this is numerical efficiency and stability. So, if we want to compute class-membership probabilities for our predictions, we have to call the softmax function explicitly:

```
with torch.no_grad():
  out = torch.softmax(model(X), dim=1)
print(out)
```

```
tensor([[0.2801, 0.3635, 0.3565]])
```