# Understanding the Object Oriented design - PyTorch

Sravanth Yajamanam

Divya James Athoopallil

## What is PyTorch?

- **PyTorch** is an open source Deep Learning Framework and a scientific computing package.
- Scientific computing aspect of PyTorch is because of PyTorch's Tensor library and the associated Tensor operations.
- PyTorch is mainly used for building neural networks and accelerated computations using GPUs(Graphics Processing Units).

**What is a Tensor?**

- The fundamental unit of PyTorch is the Tensor.
- Tensors can be considered as multi-dimensional arrays.
- Tensors in PyTorch are similar to NumPy arrays.
- Tensors can be used on a GPU that supports **CUDA** (Compute Unified Device Architecture).

## Brief History of PyTorch:

- PyTorch was initially released in October 2016.
- Before, PyTorch was created, there was a Machine Learning Framework called **Torch** which is based on **Lua** programming language.
- **Soumith Chintala** is the creator of PyTorch framework. It was created at FAIR (Facebook AI Research).
- Reason for building PyTorch was that the existing Torch framework was fading out and the need for newer version written in Python.
- Most of the internals that are performance bottlenecks are written in C++
- Source code of PyTorch is written mostly in Python.

## Components of PyTorch:

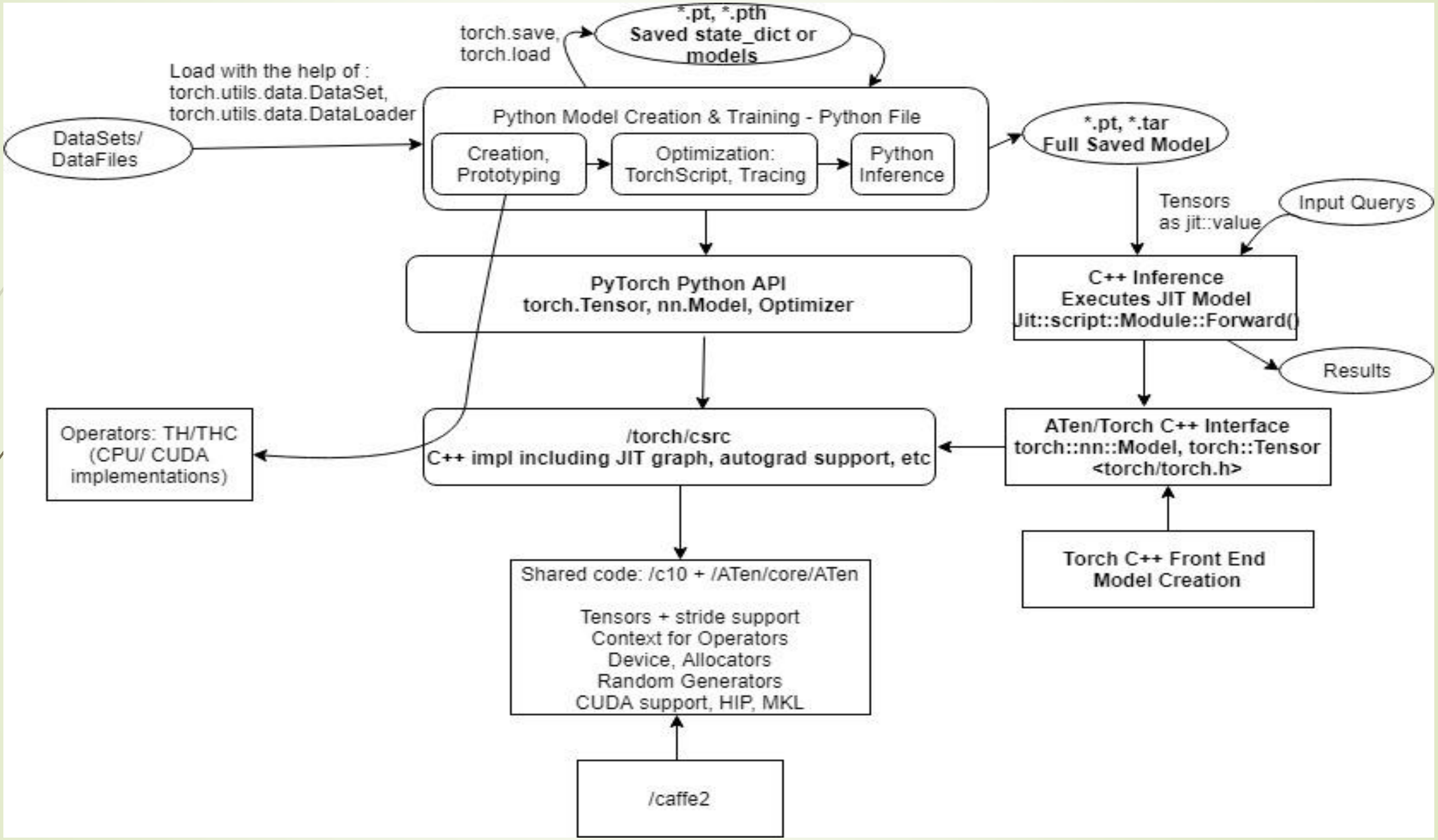
| Package | Description |
|---|---|
| torch | Top-level PyTorch package & tensor library with GPU support |
| torch.nn | A sub package that contains modules & extensible classes for building neural networks. |
| torch.autograd | A sub package that supports all the differentiable Tensor operations in PyTorch |
| torch.nn.functional | A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations. |
| torch.optim | A sub package that contains standard optimization operations like SGD and Adam. |
| torch.utils | A sub package that contains utility classes like data sets and data loaders that make data pre-processing easier. |
| torchvision | A package that provides access to popular datasets, model architectures, and image transformations for computer vision. |

## PyTorch Philosophy:

- Stay out of the way
- Cater to the impatient
- Promote linear code-flow
- Full interop with the Python ecosystem
- Be as fast as anything else

- Writing in PyTorch is nothing but extending standard Python classes.
- To debug PyTorch code, we make use of the normal Python debugger.
- Source code is readable as it is written in Python and only changed to C++ and CUDA code if it is related to performance issues.

# Data Flow & Interface Diagram of PyTorch:



## Data Flow & Interface Diagram of PyTorch:

- In **PyTorch**, model development can be done in a single Python file which can be used for prototype building, and later extended with TorchScript tags or traced for optimization.
- Inference can be done directly on the trained data or specifically in C++ under **“C++ Inference”.**
- **Data Files/ DataSet:** Data can come from variety of formats such as .csv or custom data bases. This data can be loaded directly into PyTorch/Caffe or it can be pre-processed before loading to PyTorch.

# Data Flow & Interface Diagram of PyTorch:

**DataSet Class:**

- **torch.utils.data.Dataset** is an **abstract class** representing a dataset. Our custom dataset can inherit Dataset and override the given below methods
  - __len__ so that len(dataset) returns the size of the dataset.
  - __getitem__ to support the indexing such that dataset[i] can be used to retrieve  a sample i

- Data is loaded directly into Python and accessed with the help of DataSet abstraction. **torch.utils.data.DataLoader** can be used to support batching, shuffling and loading the data in parallel using multiprocessing workers.

## Data Flow & Interface Diagram of PyTorch:

- **PyTorch Model Creation & Training File** -
  - Main Python file which can be developed by user with help of PyTorch APIs.
  - This file defines the model, loads data and runs the training process.
  - The file can also make the inference directly, or it can export an optimized model through the use of tracing or TorchScript for execution in later stages.

- **Model Creation / Prototyping** -
  - Network is created through torch.nn.Model derived class, defining layers and a forward() function that connects them in the class.
  - The model can then be trained or exported.

## Data Flow & Interface Diagram of PyTorch:

- **Loading / Saving Model State** :
  - 1) Internal state of PyTorch model is represented by state_dict and it can be saved to a file with help of **torch.save** and loaded with **model.load_state_dict**.
  - 2) Loading models allows them to be used for inference without running training again.
  - 3) Common file extensions are .pt or .pth and Python's pickle format is used for serialization whereas the full models have the extension as .tar

## Saving & Loading Models

- **torch.save**: Saves a serialized object to disk. This function uses Python's pickle utility for serialization. Models, tensors, and dictionaries of all kinds of objects can be saved using this function.
- **torch.load**: Uses pickle's unpickling facilities to deserialize pickled object files to memory. This function also facilitates the device to load the data into CPUs, GPUs.
- **torch.nn.Module.load_state_dict**: Loads a model's parameter dictionary using a deserialized state_dict.
- **Tracing & TorchScript** -
  - An existing model can be converted to a serializable graph with help of tracing, which requires deriving model from torch.jit.ScriptModule and using torch.jit.trace. Alternatively the  methods can be labeled with TorchScript, which allows them to include control flow logic.
  - In either case, the model can be saved to a .tar checkpoint file so as to be later loaded from C++.

# Inference:

- **Inference**:
  - Inference refers to using the trained model to make predictions on the dataset.
  - Basically it involves fetching the input values and getting outputs by running the trained model.
  - This can be done directly in PyTorch, or in a C++ file in further stages.
- **Torch C++ Inference** :
  - The saved PyTorch model can be loaded and executed from C++.
  - This may offer many performance/memory benefits for inference since we don't have to execute Python as it is slow compared to C++ in terms of faster execution.
- **Torch C++ Front End Model Creation and Tensors** :
  - Models can be directly created through C++ front end, by including torch.h and using its classes such as torch.nn.Module, torch.optim.SGD .
  - The tensor library can be directly used by including <ATen/Aten.h>

# PyTorch Internals

**Extending the Python Interpreter:**

- **PyTorch** defines a new package **torch.**
- Extension modules such as ._C module (a Python module written in C) allow to define new built-in object types like Tensor and to call C/C++ functions.
- The init_C()/PyInit__C() function creates the module and adds the method definitions as appropriate.
- These __init() functions add the Tensor object for each type to the ._C module so that they can be used in the module.

# PyTorch Internals

**The THPTensor Type**

- A generic Tensor is defined in PyTorch that can take different data types.
- Python runtime sees all Python objects as variables of type PyObject *, which serves as a “base type” for all Python objects.
- The formula for defining a new type is as follows:
  - The new object is defined by a  struct.
  - The type of the object is defined.
- Type of  object is defined as a set of fields that holds its properties.
- There is a pre set list of fields that can be view in th object header file in the CPython backend

## PyTorch Internals

**PyTorch cwrap:**

- In order for the Python backend to utilize the TH Tensor method it requires a cwrap which is implemented by PyTorch

- The cwrap generates an array of PyMethodDefs that can be stored or appended to the THPTensor's tp_methods field in order to interface with the CPython backend.

- Steps to executed in the wrapper function:
  - YAML "declaration" is parsed and processed
  - Source code is generated:
    - for instance, defining the method header, the actual call to the underlying library such as TH
  - The cwrap tool allows for processing the entire file at a time.

## PyTorch Internals

**PyTorch codebase**

- The PyTorch codebase has a variety of components:
  - The core Torch libraries:
    - TH, THC, THNN, THCUNN
  - Vendor libraries:
    - CuDNN, NCCL
  - Python Extension libraries
  - Additional third-party libraries:
    - NumPy, MKL, LAPACK

# PyTorch Internals

**PyTorch- Build Design**

- Setuptools and PyTorch's setup( ) function
  - The information required to build the project is contained in setup.py file present in Setuptools

- NN(Neural Network) Wrappers contains the generate_nn_wrappers() function:
  - It  parses the header files which generats cwrap YAML declarations. This is then written to output .cwrap files
  - Calls cwrap with the appropriate plugins on the output .cwrap files to generate source code.
  - The header files are parsed a second to generate THNN_generic.h.
  - THNN_generic.h. finally calls into the appropriate THNN/THCUNN library function based on the dynamic type of the Tensor

# PyTorch Internals

**PyTorch- Build Design:**

- Building" the Pure Python Modules
  - Once the dependencies are in place  Setuptools runs the build.py file. This builds the Pure Python Modules in the Library.

- Building the Extension Modules
  - The PyTorch Modules in C++ the CPython backend are the Extension Modules.
  - We take the YAML declarations in TensorMethods.cwrap, and use them to generate output C++ source files that contain implementations that work within PyTorch's C++ Ecosystem

## Neural Networks & OOPs:

- By using an object-oriented approach to the design of software systems, we can achieve advantages in terms of flexibility, extensibility, and portability.
- We can design object-oriented neural network simulation systems to achieve advantages mentioned above.
- **Why Object oriented Programming?**
  - For example, consider a backpropagation network which is a specialized learning algorithm for specific application domains. A consequence of this is lack of flexibility.
  - If a recurrent network architecture is required for an application domain, then the system has to be redesigned to accommodate the changes.
- The main motivation for an object-oriented approach to a neural network simulation system is to design reusable components that facilitate a flexible configuration of an application.

# Neural Networks & OOPs:

- **What are objects of a Neural Network?**
  - From a computational point of view, neural network objects consist of matrices, vectors and mathematical operations.
  - This view concentrates on a functional approach to neural networks, which is not what the object-oriented paradigm encourages.
  - Object-oriented methodology focuses on modelling concepts in terms of collaborating objects, rather than functions. In a neural network system, concepts can be represented by the objects such as model, neuron, weight, and pattern.

- Using object-oriented methodology in the development of the Neural Network system, a unified view of the problem domain can be achieved in all development phases from analysis to maintenance. The requirements for the system are :

  a) Abstraction

  b) Flexibility

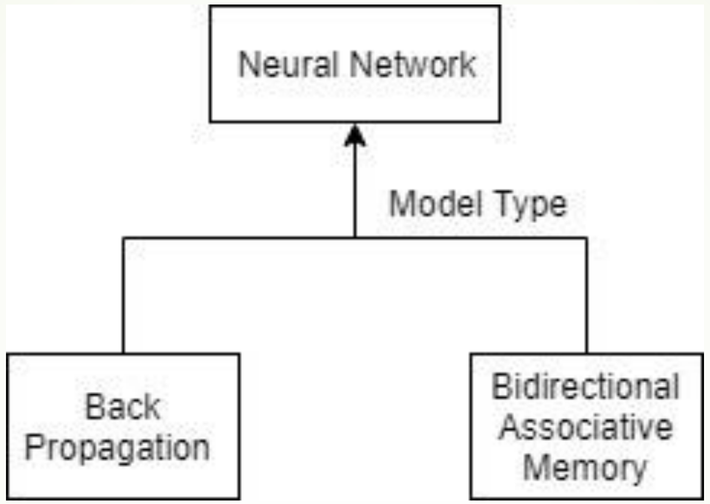
  c) Extensibility

## Neural Networks & OOPs:

- An important requirement of Neural Network system is to provide different levels of **abstraction.**
- The neural network is conceptually separated in general and specific properties.
  - General properties : Neuron Layers and weights among them
  - Specific properties : Different learning algorithms. (broadly classified to supervised, unsupervised)
- A flexible solution is required to obtain a comprehensible and powerful user interface so that any modification of the network model should be done by changing the existing network definition, not the code itself.
- A major achievement of Modular Neural Network design approach is extensibility.
- For instance, if a new learning algorithm is added to the system, only learning specific behaviour is needed and the remaining general properties are provided from the Neural Network system enabling a high degree of reuse.

## Neural Network & OOPs:

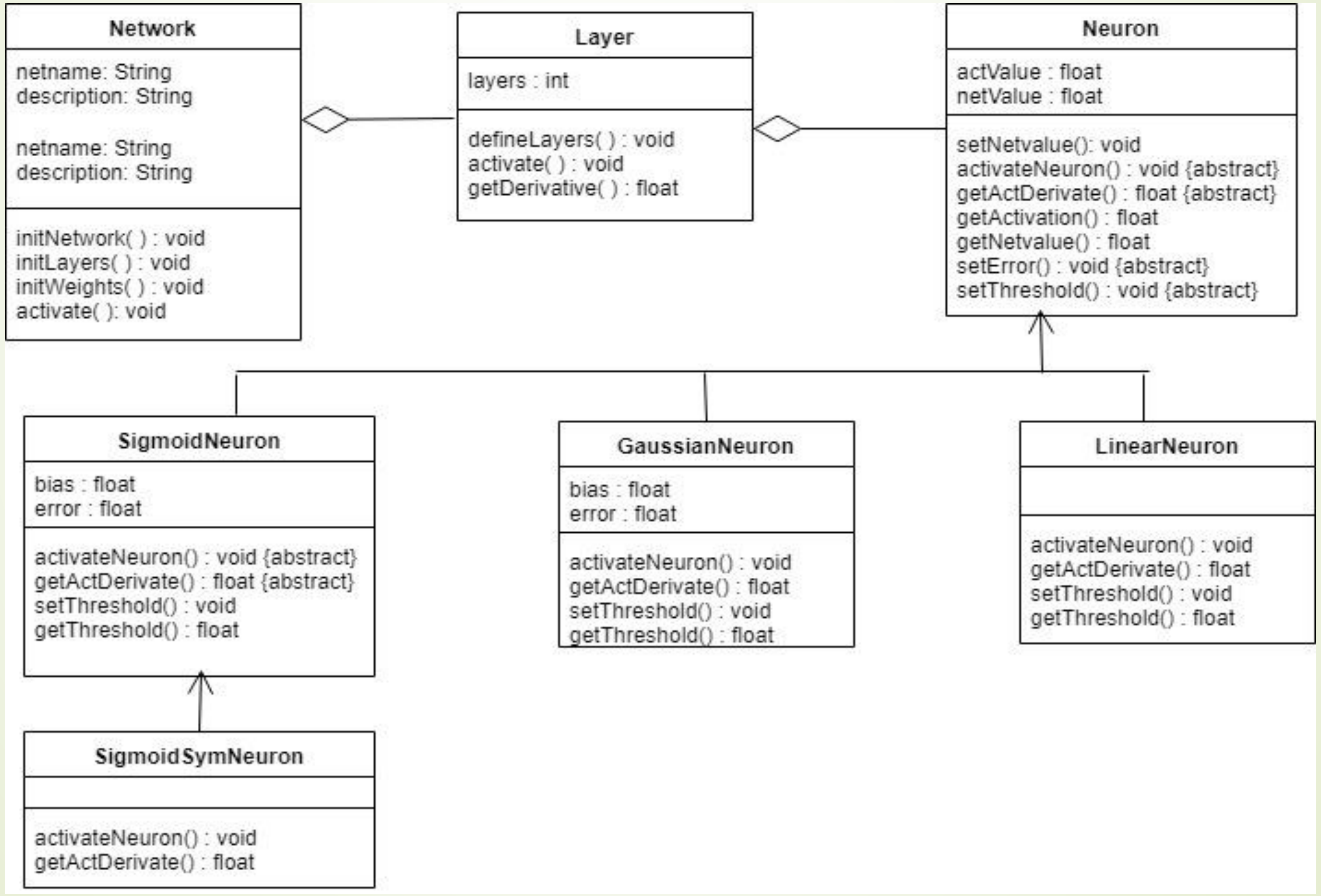
- Specific properties are different specializations of a neural network. For instance, the below diagram has two levels of abstractions.



- The general properties are partitioned in modules for logical grouping of classes, associations, and generalizations. There are basically four modules in this design: layer, weight, pattern, and parameters.

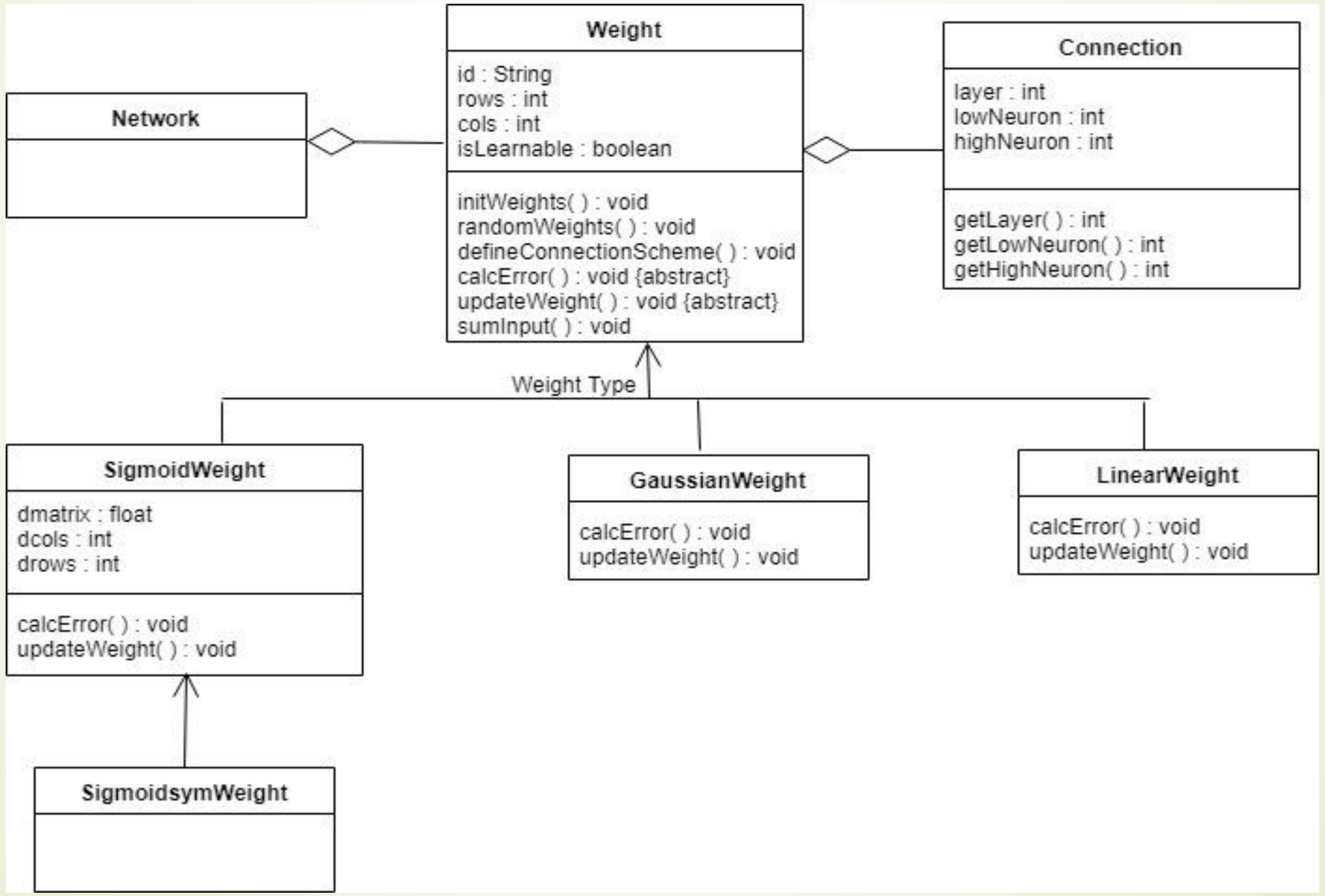
# Neural Networks & OOPs:

- Different neurons have specific behaviour. There are 3 different types of neurons and the sigmoid symmetric neuron which is a derived neuron uses different activation method instead of standard activation function.



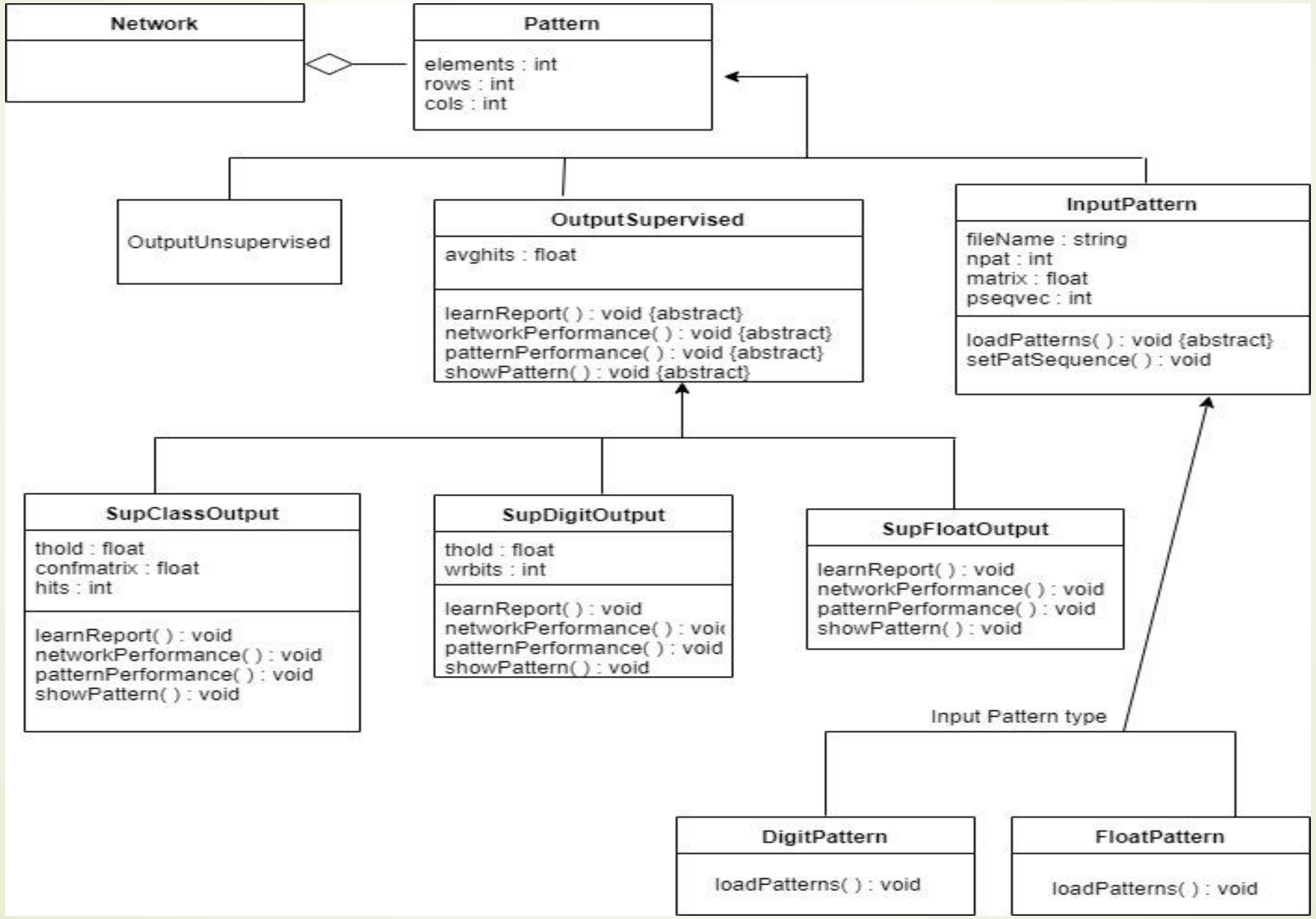
# Neural Networks & OOPs:

- Weight class is dependent on the type of neurons it connects to. A connection class is provided to make connections among neurons in the layers. Different weight classes have different behaviours associated with them. (All different weights inherit from weight class).



# Neural Networks & OOPs:

- Pattern interface is used for Unsupervised, Supervised and input patterns.(pattern input files)
- Digit output is a binary output, Float output is a continuous value output.



## PyTorch Implementation Example

## PyTorch: Tensor

- Tensor is the fundamental data structure used in PyTorch
- We transform the data into tensors in order to pass it through the neural network.
- Instances of the **torch.Tensor/ torch.tensor** class
  - Tensor attributes
    - torch.dtype: Tensors contain uniform (of the same type) numerical data for example: torch.float32
    - torch.device: specifies the device (CPU or GPU) where the tensor's data is allocated. This determines where tensor computations for the given tensor will be performed
    - torch.layout: specifies how the tensor is stored in memory

## PyTorch: Tensor

- **torch.Tensor** is the constructor of the torch.Tensor class
  - torch.Tensor() constructor lacks configuration options, hence we are unable to pass dtypes

- **torch.tensor** a factory function gets called constructs torch.Tensor objects
  - dtype is selected based on the incoming data i.e. the dtype is inferred based on the incoming data

## Building the model

- PyTorch's neural network library contains all of the typical components needed to build neural networks. The primary component to build a neural network is a layer.
- Each layer in a neural network has two primary components:
  - A transformation (code)
  - A collection of weights (data)
- A class Module within the neural network package is the base class for all neural network modules with layers.
  - This is an instance of inheritance in action.
- Neural networks and layers in PyTorch extend the nn.Module class. This means that we must extend the nn.Module class when building a new layer or neural network in PyTorch.

# Building a neural network in PyTorch

In order to build the network:

- We first design a neural network class that extends the nn.Module base class.
- As we instantiate a neural network we pass the network's layers as class attributes using pre-built layers from torch.nn through the class constructor
- We use operations from the nn.functional API the network's layer attributes as well as to define the network's forward pass.

- **Code Sample**

```
class Network:
    def __init__(self):
        self.layer = None

    def forward(self, inp):
        inp = self.layer(inp)
        return inp
```

## Building a neural network in PyTorch

- To make our Network class extend nn.Module, we must do two additional things:
- Specify the nn.Module class in parentheses on line 1.
- Insert a call to the super class constructor inside the Network constructor.

- **Code Sample**

```
class Network(nn.Module): # line 1
    def __init__(self):
        super(Network, self).__init__()
        self.layer = None

    def forward(self, t):
        t = self.layer(t)
        return t
```

## Define the network’s layers as class attributes

- In the previous slide the Network class has a single dummy layer as an attribute.
- We are now building some real layers that come pre-built for us from PyTorch's nn library.
- We’re building a CNN(Convolutional Neural Network), so the two types of layers we'll use are linear layers and convolutional layers.
- **Code Sample**

```
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)

        self.fc1 = nn.Linear(in_features=12 * 4 * 4, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=60)
        self.out = nn.Linear(in_features=60, out_features=10)

    def forward(self, t):
        # implement the forward pass
        return t
```

## Citations

- Ellinger, Barry Kristian; An Object-Oriented Approach to Neural Network; https://pdfs.semanticscholar.org/e582/181b2b722e00f6db92f2f64a1f5b7713dc37.pdf
- Vasilev Ivan et.al. ; Python Deep Learning
- Tianqi Chen et.al.; MXNet: A Flexible and Efficient Machine LearningLibrary for Heterogeneous Distributed Systems; https://www.cs.cmu.edu/~muli/file/mxnet-learning-sys.pdf
- Abadi, Martín,; TensorFlow: A System for Large-Scale Machine Learning; https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
- Lerer Adam et.al., Pytorch-Biggraph: A large scale graph embedding system https://www.sysml.cc/doc/2019/71.pdf
- Liao, Qianli ; Object Oriented Deep Learning; https://dspace.mit.edu/bitstream/handle/1721.1/112103/CBMM-Memo-070.pdf?sequence=1
- Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch

# Citations

- Bradbury, James et,al. ;Automatic Batching as a Compiler Pass in PyTorch; http://learningsys.org/nips18/assets/papers/107CameraReadySubmissionMatchbox__LearningSys_Abstract_%20(2).pdf
- https://pytorch.org/features
- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/blog/a-tour-of-pytorch-internals-1/
- https://pytorch.org/blog/a-tour-of-pytorch-internals-2/
- https://github.com/PyTorch/PyTorch/wiki
- https://github.com/PyTorch/PyTorch/wiki/PyTorch-Data-Flow-and-Interface-Diagram
- https://github.com/PyTorch/PyTorch/wiki/Code-review-values
- https://docs.python.org/3.7/extending/index.html

THANK YOU