This diagram illustrates potential dataflows of an AI application written in PyTorch, highlighting the data sources and artifacts produced, as well as APIs used to interact with the underlying system. User-written Python files are highlighted as gray, while interfaces/APIs they interact with are blue. Details of the diagram are described by bullets in the text below.

[[images/pytorch_wiki_dataflow_interface_diagram.png]]

Since PyTorch codebase also includes the Caffe2 library, interactions with this library are illustrated as well. There is a merge ongoing between the two libraries, with the goal of sharing code and exposing benefits of both through PyTorch. We recommend that new users rely only on the PyTorch interfaces (near the top of the diagram), as these will be the supported API going forward.

PyTorch

In PyTorch, model development can be done in one Python file that is first used for prototyping, and later augmented with TorchScript tags or traced for optimization. Inference can also be done directly on trained data, or later in C++ as described under "C++ Inference".

Here is the description of related steps and components as seen on the diagram: * **Data Files / DataSet** - Data for AI training can come from variety of formats, such as .csv, image files in a directory, or custom databases. This data can either be preprocessed, or loaded directly into PyTorch/Caffe2. * **Loading data** - In PyTorch, data is often loaded directly into Python and accessed with help of DataSet abstraction (a class that define **getitem** accessor length). torch.utils.data.DataLoader can be used to support batching, shuffling and background loading; plus extra helpers are available such as torchvision.datasets.ImageFolder to access and process images. Many of these are described in (https://pytorch.org/tutorials/beginner/data_loading_tutorial.html). * You can also use torchvision.datasets helpers, such as torchvision.datasets.CIFAR10. * https://pytorch.org/docs/stable/torchvision/datasets.html. Look here for example of implementing datasets.

- **PyTorch Model Creation & Training File** - Main Python file developed by user with help of PyTorch APIs. This file defines the model, loads data and runs training. The file can also do inference directly, or export optimized model through the use of tracing or TorchScript for execution later on.
    - **Model Creation / Prototyping** - Net is created by creating torch.nn.Model derived class and defining layers and a forward() function that connects them, in the class. The model can then be trained or exported.
        - **Loading / Saving Model State** - 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, as

described here: https://pytorch.org/tutorials/beginner/saving_loading_models.html. Loading models allows them to be used for inference without re-running the expensive training. Common file extension are .pt or .pth; internally, they use Python's pickle format is used for serialization. Full models have extension .tar.

  - **Tracing & TorchScript** - An existing model can be converted to a serializable graph with the help of tracing, which requires deriving model from torch.jit.ScriptModule and using torch.jit.trace. Alternatively, its methods can be labeled with with TorchScript, which allows them to include control flow logic. This is described here: https://pytorch.org/tutorials/beginner/deploy_seq2seq_hybrid_frontend_tutorial.html. In either case, the model can be saved to a .tar checkpoint file to be loaded later from C++.
    * Tracing is also used internally to export PyTorch models to ONNX, for execution with Caffe2 (https://pytorch.org/tutorials/advanced/super_resolution_wit
  - **Inference** - Inference refers to using trained model to make predictions; it generally involves getting some input queries and getting output result from running the model. This can be done directly in PyTorch, or in a C++ file later on.

- **Torch C++ Inference** - The saved PyTorch model can be loaded and executed from C++. This may offer some performance/memory benefits for inference since we don't have to include or execute Python. This is described in https://pytorch.org/tutorials/advanced/cpp_export.html.
- **Torch C++ Front End Model Creation and Tensors** - You can create models directly through the C++ Front End, by including torch.h and using its classes such as torch::nn::Module, torch::optim::SGD, etc; you can also use the tensor library directly by including <ATen/Aten.h>. This is described in https://pytorch.org/cppdocs/frontend.html and https://pytorch.org/cppdocs/.

Caffe2

Caffe2 (https://caffe2.ai/) is a separate ML library that work by building an op graph. Some of its functionality is currently being merged into PyTorch, so we are describing it here in case you use it to interface with older code.

- **Loading Data** in Caffe 2 is usually done through operators.
  - **Data Files / DataSet** - Similar to PyTorch, training data can come from variety of formats, such as .csv or custom databases. This data is often reprocessed, but can also be loaded directly into Caffe2.
    * Samples of popular datasets are present here (https://caffe2.ai/docs/datasets.html).
  - **Script Preprocessing** - It is sometimes useful to preprocess data into the desired structure as a separate step, either with Python or a binary tool. In Caffe2 examples MNIST example, we use 'make_mnist_db'

binary to generate LevelDB files, further loaded during training (https://github.com/caffe2/tutorials/blob/master/MNIST_Dataset_and_Databases.ipynb).

  - **LevelDB, LMDB, etc.** - Some Caffe2 examples use the popular Key/Value pair databases. These are convenient for accessing data. The library also contains operators to read them. More info about these formats: https://github.com/google/leveldb. LMDB: https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database.
  - **Database Access Operators** - A key feature of Caffe2 is that it can load data directly into the workspace of the NetDef graphs through dedicated operators. These are usually set up during model creation and are executed with the net. There are operators to test that database "Exists", and allow it to "Load" or "Save" tensor data to it - these are some of the main way to get data in. There are also additional custom operators such as ImageInput, VideoInput, etc. to pull in image or video frames.

- **Creation & Training** - Caffe2 Python files build graph based models, where you first create a Net with operators, and then execute it within a workspace. Data is fed into workspace through named NumPy tensors (workspace.FeedBlob/FetchBlob); and used when you run the net workspace.RunNet/RunNetOnce(net). Backward propagation is supported with AddGradiendOperators. Several Python helper modules are used for making models such as ModelHelper or brew, creating the same underlying graph structure that is passed to the underlying Caffe2 C++ code in protobuffer format for execution. Intro explanation: https://caffe2.ai/docs/intro-tutorial.html.
  - Two saved formats: protobuffers with *.pb extension
    * https://caffe2.ai/docs/tutorial-loading-pre-trained-models.html
- **Caffe2 API** - Caffe2 has a couple of tiers of Python APIs: Lower level NetDefs, and higher level helper modules such as 'brew' and 'ModelHelper'. You can use helpers such a 'brew' (https://caffe2.ai/docs/brew.html) to simplify the creation of the models; in either case they create the same NetDef graph which is passed down into the runtime as a protobuffer.
- **DataBase File** - Trained Caffe2 nets are often saved as a database file. The database includes serialized MetaNetDef in protobuffer format (describing init and predict nets) and weight/value blobs accessible based on parameter names.
- **Caffe2 C++ Inference** - Inference runs the net, typically getting a tensor map of inputs (labeled by name) and producing a tensor output. The net is often split into init_net, which is ran once, and predict_net, ran on every request. caffe2::Predictor or similar helper classes can be used to manager this process.

ONNX

ONNX stands for “Open neural network exchange format” (https://onnx.ai/) used for saving models; it is developed in collaboration with Microsoft, AWS and other partners. The .onnx format supports a subset of operators from PyTorch / Caffe2, and can be used to export PyTorch models for execution in Caffe2.

Github location: https://github.com/onnx/onnx

- **ONNX Model -** Usually stored as .onnx file, can be saved from PyTorch by calling torch.onnx.export.
- **Running ONNX on Caffe 2 -** Exported model can be loaded into Caffe2 with help of caffe2.python.onnx.backend and ran from Python as described in TRANSFERING A MODEL FROM PYTORCH TO CAFFE2 AND MOBILE USING ONNX (https://pytorch.org/tutorials/advanced/super_resolution_with_caffe2.html).
- **onnx -** Open source Python module, “import onnx”, used to load/work with onnx files. You read more about it here: https://github.com/onnx/tutorials/blob/master/tutorials/Py