Introduction to Deep Learning Framework

1. Introduction

1.1. Commonly used frameworks
The most commonly used frameworks for deep learning include Pytorch, Tensorflow, Keras, caffe, Apache MXnet, etc.

PyTorch: open source machine learning library; developed by Facebook AI Research Lab; based on the Torch library; supports Python and C++ interfaces.

Tensorflow: open source software library dataflow and differentiable programming; developed by Google brain team; provides stable Python & C APIs.

Keras: an open-source neural-network library written in Python; conceived to be an interface; capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML.

Caffe: open source under BSD licence; developed at University of California, Berkeley; written in C++ with a Python interface.

Apache MXnet: an open-source deep learning software framework; supports a flexible programming model and multiple programming languages (including C++, Python, Java, Julia, Matlab, JavaScript, Go, R, Scala, Perl, and Wolfram Language.)

1.2. Pytorch

1.2.1 Data

- Tensor: the major computation unit in PyTorch. Tensor could be viewed as the extension of vector (one-dimensional) and matrix (two-dimensional), which could be defined with any dimension.
- Variable: a wrapper of tensor, which includes creator, value of variable (tensor), and gradient. This is the core of the automatic derivation in Pytorch, as it has the information of both the value and the creator, which is very important for current backward process.
- Parameter: a subset of variable

1.2.2. Functions:

- NNModules: NNModules (torch.nn) is a combination of parameters and functions, and could be interpreted as layers. There some common modules such as convolution layers, linear layers, pooling layers, dropout layers, etc.
- Optimizer: torch.optim which supports 9 Optimizer including SGD, Adagrad, Adam, RSMprop, etc
• Loss Function: torch.nn which supports 18 Loss functions including L1Loss, MSELoss, Cross Entropy, etc
• Multi-Processing

1.3. TensorFlow

1.3.1. Data Flow Graph

The computation in TensorFlow is based on Data Flow Graph, which is expressed with nodes and edges. Nodes, generally speaking, represent the operations, except when referred as where the data is fed in, pushed out, or read/wrote as persistent variable. Edges represent the flow between nodes. The direction of arrows in the graph below represents the direction of data flow.

1.3.2. Tensor

Since the framework is called TensorFlow, we could then infer that the data flows in the graph is in unit of tensor. As mentioned in 1.2.1., tensor could be viewed as the extension of vector (one-dimensional) and matrix (two-dimensional), which could be defined with any dimension.
There are two attributes for tensor: shape (rank for each dimension) and type (data type).

We operate the tensor to get outputs, and the input and output of each of the tensor is also always a tensor.

The type of Tensor includes tf.constant(), tf.Variable(), tf.placeholder(), and tf.SparseTensor() in TensorFlow.

We could get the value of the tensor through calling Session.run(tensor) and tensor.eval().

1.3.3. Session

After we successfully constructed a graph, we need to run it in a Session. Session acts as a bridge between graph and computing resource. The graph is a member of the Session class, together with a runner which responds for running the graph. The major task for Session is to distribute the computation of graph to CPU or GPU while the graph is running.

1.3.4. Useful tools

- TensorBoard: a tool for visualizing various aspects of training machine learning models. It’s one of the most useful features to come out of the TensorFlow project. With a few code snippets in a training script you can view training curves and validation results of any model. TensorBoard runs as a web service which is especially convenient for visualizing results stored on headless nodes.

- Keras: a higher-level API with a configurable back-end. At the moment TensorFlow, Theano and CNTK are supported. Keras is also distributed with TensorFlow as a part of tf.contrib. The API is especially easy to use. It’s one of the fastest ways to get running with many of the more commonly used deep neural network architectures. However, the API is not as flexible as PyTorch or core TensorFlow.

2. Comparison between PyTorch and Tensorflow
2.1. Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Tensorflow</th>
<th>PyTorch</th>
<th>Which is better?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramp-up time</td>
<td>TensorFlow takes longer time: TensorFlow needs to get compiled into Python graph and then run by the TensorFlow execution engine.</td>
<td>PyTorch takes shorter time: PyTorch is built to use the power of GPU for training, and is deeply get integrated with Python.</td>
<td>PyTorch</td>
</tr>
</tbody>
</table>
| Coverage         | TensorFlow has an extremely wide coverage. TensorFlow supports most of the functionality, and some of them are not supported by PyTorch. The contrib package has many more high level functions and operations that PyTorch doesn’t have. | PyTorch has a fairly wide coverage, but still less than TensorFlow. PyTorch lacks some functions compared with TensorFlow:  
  ● Flipping a tensor along a dimension (np.flip, np.fliplr)  
  ● Checking a tensor for NaN and infinity (np.is_nan, np.is_inf)  
  ● Fast Fourier transforms (np.fft) | TensorFlow        |
| Graph Creation and Debugging | Since the graph construction in TensorFlow is static, the graph is typically first compiled and then run. TensorFlow has control flow operations in constructing dynamic computations, but it’s more tedious than PyTorch. To debug TensorFlow code, there are two options:  
  1. Request variables that you want to | The graph construction is dynamic, which means it’s built and runs at the same time. Thus, PyTorch is easier to debug as well. Debugging PyTorch code is just like debugging Python code. | PyTorch          |
<table>
<thead>
<tr>
<th>Section</th>
<th>Comparison</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialization</td>
<td>Both TensorFlow and PyTorch are easy to save. TensorFlow, in some way, is more integrated. In serialization, TensorFlow can save the graph as protocol buffer, which turns out to be more efficient, supports more languages, and enables to change source code interrupting old models.</td>
<td>Both TensorFlow and PyTorch are easy to save. But PyTorch is less flexible compared with TensorFlow regarding the mentioned advantages of TensorFlow.</td>
</tr>
<tr>
<td>Deployment</td>
<td>Both TensorFlow and PyTorch are easy to wrap for small scale server-side deployment. TensorFlow supports both mobile and embedded deployment. For heavily used machine learning services, TensorFlow is better because of</td>
<td>Both TensorFlow and PyTorch are easy to wrap for small scale server-side deployment. For heavily used machine learning services, however, PyTorch is less efficient than TensorFlow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TensorFlow</td>
</tr>
<tr>
<td>TensorFlow Serving. Models can be hot-swapped easily without bringing the service down</td>
<td>The documentation enables users to search easily. However, there are a huge number of documentations for TensorFlow, and many of them are not well-organized enough.</td>
<td>The Python APIs are well documented and there are enough examples and tutorials to learn either framework. However, the C library is mostly undocumented, which might have impact on writing a custom C extension or contributing to the software.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Documentation</td>
<td>Data Loading</td>
<td>Device Management</td>
</tr>
<tr>
<td>Not very straightforward, especially for parallelizing data loading. Also, for TensorFlow, APIs are more verbose and harder to learn.</td>
<td>Well designed. While loading data in PyTorch, a data loader will take a dataset and a sampler and then produce iteration over the dataset with the sampler. Also, it’s pretty easy to conduct parallelizing data loading in PyTorch via passing the <code>num_workers</code> argument.</td>
<td>TensorFlow assumes the user want to run on GPU if one is available. By default, however, TensorFlow consumes the memory on all available GPUs. Thus, the user need to specify <code>CUDA_VISIBLE_DEVICES</code> argument.</td>
</tr>
<tr>
<td>PyTorch</td>
<td>TensorFlow</td>
<td>PyTorch</td>
</tr>
</tbody>
</table>
2.2. Summary

The implementation of TensorFlow is symbolic programming, while the implementation for PyTorch is imperative programming. Thus, PyTorch is easier to implement but less efficient compared with TensorFlow.

Also, though both of TensorFlow and PyTorch conduct computation on tensor, there are still differences. TensorFlow generates the graph statically, while PyTorch is dynamic. From this perspective, PyTorch seems more flexible, and easier to debug.

2.3. Key operations for Tensor

2.3.1. PyTorch

- torch.lerp(start, end, weight) : the return is out = start + (end - start) * weight
- torch.rsqrt(input) : return the reciprocal of the square root
- torch.mean(input) : return mean value
- torch.std(input) : return standard deviation
- torch.prod(input) : return the product of all elements
- torch.sum(input) : return the sum of all elements
- torch.var(input) : return the variance of all elements
- torch.equal(torch.Tensor(a), torch.Tensor(b)) : compare the two tensors, and then return true if the two tensors equal to each other, otherwise return false
- torch.ge(input, other, out = none) 
  - ge: input >= other  if a >= b, return true, otherwise return false
- torch.gt(input, other) :  if a > b, return true, otherwise return false
- torch.lt(input, other) :  if a < b, return true, otherwise return false
- torch.max(input) : return the maximum among elements
- torch.min(input) : return the minimum among elements
- element_size() : return bytes of a single element

2.3.2. TensorFlow

- tf.add (x, y, name = None)
- tf.sub (x, y, name = None)
- tf.mul (x, y, name = None)
- tf.div (x, y, name = None)
- tf.mod (x, y, name = None)
- tf.abs (x, name = None)
- tf.neg (x, name = None)
- tf.sign (x, name = None)
- tf.inv (x, name = None)
- tf.square (x, name = None)  
  - (y = x * x = x ^ 2)
- tf.round (x, name = None)  
  - # ‘a’ is [0.9, 2.5, 2.3, -4.4]
  - tf.round(a) ==> [ 1.0, 3.0, 2.0, -4.0 ]
- tf.sqrt (x, name = None)  
  - (y = sqrt{x} = x ^ {1 / 2}).
- tf.pow (x, y, name = None)  
  - # tensor ‘x’ is [[2, 2], [3, 3]]
  - # tensor ‘y’ is [[8, 16], [2, 3]]
  - tf.pow
(x, y) \rightarrow [[256, 65536], [9, 27]]

```
tf.exp (x, name = None)
tf.log (x, name = None)
tf.maximum (x, y, name = None)  (x > y ? x : y)
tf.minimum (x, y, name = None)  (x < y ? x : y)
tf.cos (x, name = None) cosine
tf.sin (x, name = None) sine
tf.tan (x, name = None) tan
tf.atan (x, name = None) ctan
```

3. Reference