Introduction to PyTorch

L. Palzer

Machine Learning for Communications – TUM LNT

November 12, 2019


TUM Uhrenturm

Overview

Deep Learning Frameworks

Why use a deep learning framework?

- Implementing own models from scratch does not scale
- Huge open source community is there to help

How to choose one?

- Ease of use
- Computation speed
- Active community
- Applications (images, NLP, etc.)

Deep Learning Frameworks

Most prominent deep learning frameworks for python:

- TensorFlow
- PyTorch

Other frameworks

- Microsoft Cognitive Toolkit
- Caffe2 (merged into Pytorch in March 2018)
- Theano (discontinued by original developers in May 2018)
- ...

TensorFlow

- Starting in 2011, Google Brain built DistBelief as a proprietary machine learning framework
- TensorFlow emerged from this as an open source library, first release in November 2015
- Current stable release: 2.0.0 (October 1, 2019)
- Native Python API, but with bindings to many other languages
- Tensorflow Lite: release for embedded devices such as Android based smartphones

`https://www.tensorflow.org`

PyTorch

- Based on the Torch[1] library for Lua, first converted to Python by Adam Paszke
- Initial release in October 2016
- Now primarily developed by Facebook AI team
- Current stable release: 1.3.0 (October 10, 2019)
- PyTorch Mobile: experimental release for iOS and Android

[1] `http://torch.ch`

`https://pytorch.org`

TensorFlow vs. Pytorch

Before 2.0 / 1.3

TensorFlow:

- \+ Better performance
- \+ Support for embedded devices
- \+ Larger community
- \+ Visualization with Tensorboard
- \- Difficult to learn/ adapt models/ debug

$\rightarrow$ Production

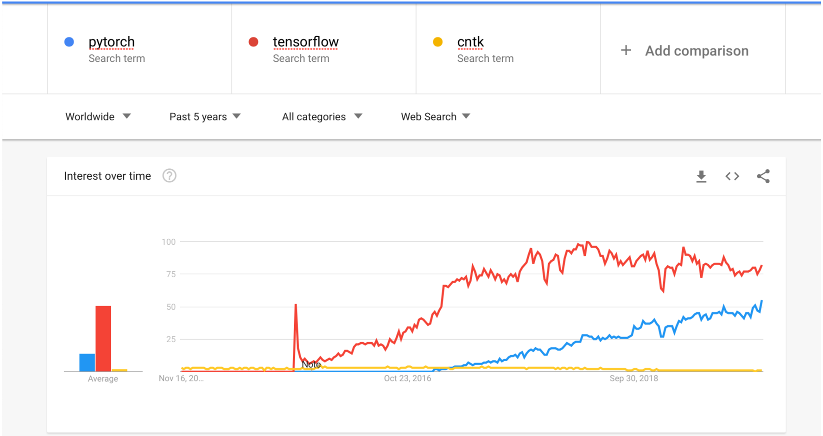
Pytorch:

- \+ More ”Pythonic”
- \+ More flexible
- \+ Easier to debug
- \+ Easier to parallelize
- \- No proper visualization

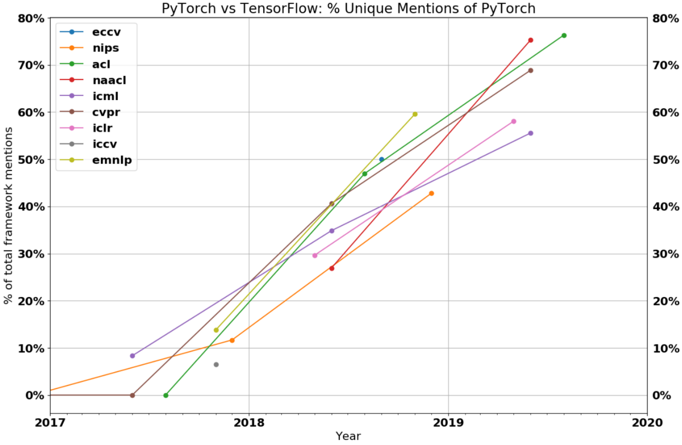
$\rightarrow$ Research

With 2.0 / 1.3, increasingly a ”matter of taste”

Google Trend Search



Popularity in Academia



https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

High Level APIs

Keras[2]:

- Developed by François Chollet at Google
- Works on top of TensorFlow, CNTK, Theano
- Recommended starting point for learning TensorFlow

fastai[3][4]

- Works on top of PyTorch
- Project lead by Jeremy Howard
- Used in fast.ai online courses

---

[2] https://keras.io

[3] https://docs.fast.ai

[4] https://www.fast.ai

Example with fastai

Train a world class model with five lines of code:

```
from fastai.vision import *
path = untar_data(MNIST_PATH)
data = image_data_from_folder(path)
learn = cnn_learner(data, models.resnet18, metrics=accuracy)
learn.fit(5)
```

PyTorch Basics

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Getting PyTorch

- Install PyTorch locally:
  `pip3 install torch torchvision`
- Already available in Google Colab
- Import module:

```
>>> import torch
```

Tensors

All of you have encountered special classes of tensors before:

- Scalars: Zero order tensor
- Vectors: First order tensor
- Matrices: Second order tensor
- $\vdots$
- Multidimensional arrays: higer order tensor

Tensors in PyTorch are similar to NumPy's ndarrays, but tensors can be used on a GPU to accelerate computing

Basics: Tensors

- Create a tensor of zeros:

```
>>> a = torch.zeros(2,3)
>>> print(a)
tensor([[0 0 0],
        [0 0 0]])
```

- Create a tensor from data:

```
>>> b = torch.tensor([1.25,4.3])
>>> print(b)
tensor([1.25,4.3])
```

- Important methods such as `.empty`, `.randn`, `.float` etc.

Tensors from NumPy

- Pass by reference

```
>>> a = np.array([[1,2],[3,4]])
>>> print(a)
[[1 2]
 [3 4]]
>>> b = torch.from_numpy(a[:,0])
>>> print(b)
tensor([1, 3])
>> b[:] = 8
>>> print(a)
[[8 2]
 [8 4]]
```

Tensors from NumPy

- Copy from NumPy:

```
>>> a = np.array([[1,2],[3,4]])
>>> b = torch.tensor(a)
>>> c = torch.tensor(a, dtype=torch.float32) #for float
>>> print(c.numpy())
[[1 2]
 [3 4]]
```

- Indexing works the same as in NumPy
- For reshaping, use `.view` on a tensor similar to `.reshape` in NumPy

PyTorch Automatic Differentiation

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Autograd

- Central package in PyTorch: `autograd`
- Provides automatic differentiation for all operations on tensors
- Workflow:
  - Set tensor attribute `.requires_grad` to `True` to start tracking operations
  - Perform computations as usual
  - Each involved tensor has now a function `.grad_fn` referencing the gradient
  - Backprop the gradients using `.backward()`
  - Call the gradient via `.grad`

Autograd Example

- Consider $y = (2 \cdot x_1 + x_2)^2$ with $x_1 = 3$ and $x_2 = 4$

```
>>> x1 = torch.tensor(3.0,requires_grad=True)
>>> x2 = torch.tensor(4.0,requires_grad=True)
>>> y = (2*x1 + x2)
>>> print(y)
tensor(10., grad_fn=<AddBackward0>)
>>> y = y**2
tensor(100., grad_fn=<PowBackward0>)
>>> y2.backward()
>>> print(x1.grad,x2.grad)
tensor(40.) tensor(20.)
```

PyTorch Neural Networks

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

Workflow

- Define neural network
- Set up data set
- Choose loss function and optimization algorithm
- Train!

Defining the Network

- Network is `child class` inheriting predefined properties and methods
- Define learnable parameters and forward propagation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.module):
    def __init__(self):
        super(Net, self).__init__()
        # define learnable parameters
    def forward(self, x):
        # define forward propagation
        return x
```

Learnable Parameters

- Example: network with two hidden layers

```
    def __init__(self):
        super(Net, self).__init__()
        self.h1  = nn.Linear(1,8, bias=True)
        self.h2  = nn.Linear(8,8, bias=True)
        self.out = nn.Linear(8,1, bias=True)
```

- Weights and biases are initialized with uniform distribution
- Many other layers[5], e.g., `nn.conv1d`, `nn.conv2d`, etc.

[5] https://pytorch.org/docs/stable/nn.html#convolution-layers

Forward Propagation

- Example: network with two hidden layers

```
def forward(self, x):
    x = F.relu(self.h1(x)) # First hidden layer
    x = F.relu(self.h2(x)) # Second hidden layer
    x = self.out(x) # Output layer
    return x
```

- Can perform many other operations such as reshaping

Example

```
>>> net = Net() # Initialize network
>>> print(net)
Net(
  (h1): Linear(in_features=1, out_features=8, bias=True)
  (h2): Linear(in_features=8, out_features=8, bias=True)
  (out): Linear(in_features=8, out_features=1, bias=True)
)
>>> params = list(net.parameters())
>>> print(params[0].T) # weights in the first layer
tensor([[-0.3591,  0.7028, -0.0415, -0.7531,  0.3992, -0.6650,
-0.4175,  0.5087]], grad_fn=<PermuteBackward>)
```

Dataset

- Create dataloader object to manage dataset

```
dataset = torch.utils.data.TensorDataset(x_tensor,y_tensor)
dataloader = torch.utils.data.DataLoader(dataset,
    batch_size = 128)
```

- Example: download MNIST dataset

```
train_set = torchvision.datasets.MNIST('./files/',
    train=True, download=True)
train_loader = torch.utils.data.DataLoader(train_set,
    batch_size=128, shuffle=True)
```

Optimization

- Define loss function and optimization algorithm
- Examples `.BCEloss`, `.CrossEntropyloss`

```
criterion = nn.MSELoss()
```

- Choose optimization algorithm[6]
- Can specify learning rate, momentum, weight decay, etc.

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
```

[6] See this paper: https://arxiv.org/pdf/1609.04747.pdf

Learning

- Run the learning procedure

```
for epoch in range(num_epoch)
    for idx, (inputs, labels) in enumerate(train_loader):
        # reset optimizer
        optimizer.zero_grad()
        # Forward pass
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        # Backward and optimize
        loss.backward()
        optimizer.step()
```

Outlook

Outlook

- Using GPUs
- Other networks, e.g., convolutional
- Different optimization algorithms