PyTorch 1.2 Tutorial

Presenter: Hao-Ting Li (李皓庭)

What/Why is PyTorch?

- Wiki:
  - PyTorch is an open-source machine learning library for Python
- Advantage:
  - Python-based
  - Dynamic Computational Graph
  - Clear API
  - Update frequently

比較其他框架

- Caffe (2014~2017)
  - bad documents
  - hard to use
- Tensorflow (2016~)
  - hard to install
  - hard to use
- Keras (2015~)
  - high-level API only

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

Installation

- Recommend using Conda
  - Anaconda
  - Miniconda (Good)
- Prerequisite
  - NVIDIA graphics card
  - CUDA
    - Linux 上尋找 CUDA 版本的指令：
      `find /usr/local -maxdepth 1 -type d -name 'cuda*'`
    - e.g.

```
maniac@gslave01[03:53:37]~$ find /usr/local -maxdepth 1 -type d -name 'cuda*'
/usr/local/cuda-9.0
/usr/local/cuda-8.0
```

Installation

PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	Stable (1.2)	Preview (Nightly)			
Your OS	Linux	Mac	Windows		
Package	Conda	Pip	LibTorch	Source	
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7	C++
CUDA	9.2	10.0	None		

Run this Command:

```
conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
```

Previous versions of PyTorch

Verify the PyTorch is Using the GPU

- `import torch`
  `torch.cuda.is_available()`
- e.g.

```
(pytorch-1.0) maniac@kurisu[04:01:01]~$ python
Python 3.7.2 (default, Dec 29 2018, 06:19:36)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
>>>
```

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

Basic Data Type: **Tensor**

- Tensor: multi-dimensional array
- Shape: the dimension
  - Order: (batch, channels, width, height)

torch.Tensor

- Equal to `np.ndarray` (NumPy)
  - support GPU computing
  - support gradient computing
  - can use built-in list to construct
- e.g.

```
>>> import torch
>>> x = torch.tensor([[5, 4], [8, 7]])
>>> x.shape
torch.Size([2, 2])
```

torch.Tensor

- $f(x, y) = x^2 + 2y$

```
>>> def f(x, y):
...     return x.pow(2) + 2*y
...
```

- set $x = 8, y = 7$

```
>>> x = torch.tensor([8.])
>>> y = torch.tensor([7.])
>>> f(x, y)
tensor([78.])
```

Autograd: Automatic Differentiation

- $f(x, y) = x^2 + 2y$
- $x = 8, y = 7$
- Set `requires_grad=True` to compute gradient
- 範例

```
>>> x = torch.tensor([8.], requires_grad=True)
>>> y = torch.tensor([7.])
>>> y.requires_grad_()
```

Autograd: Automatic Differentiation

- $f(x, y) = x^2 + 2y$
- $x = 8, y = 7$
- Computing $\nabla f(x, y) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) = (2x, 2)$
- e.g.

```
>>> f(x, y)
tensor([78.], grad_fn=<AddBackward0>)
>>> f(x, y).backward()                    # compute the gradient by chain rule
>>> x
tensor([8.], requires_grad=True)
>>> x.grad
tensor([16.])
>>> y.grad
tensor([2.])
```

In-place operation

- `.function_()`
- Compute the value and return the modified tensor
- e.g.

```
>>> x.add(y)
tensor([15.])
>>> x
tensor([8.])
>>> x.add_(y)
tensor([15.])
>>> x
tensor([15.])
```

CPU/GPU Computing

- CPU is default

```
# 1.
>>> x.cuda(1)
tensor([8.], device='cuda:1')

# 2.
>>> x.to('cuda:1')
tensor([8.], device='cuda:1')

# 3.
>>> device = torch.device('cuda:1')
>>> x.to(device)
tensor([8.], device='cuda:1')
```

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

Dataset & Dataloader

- `torch.utils.data.`**`Dataset`**
- `torch.utils.data.`**`DataLoader`**

`torch.utils.data.Dataset`

- An interface interact with `torch.utils.data.DataLoader`
- Usage:
  inherit `torch.utils.data.Dataset` and implement
  - `def __getitem__(self, index)`
    - return a batch data
    - return type: `torch.Tensor or built-in dict`
  - `def __len__(self)`
    - return the size of the dataset

torch.utils.data.**DataLoader**

- Usage:
  Pass the instance of the dataset class to the dataloader
  - ```
    torch.utils.data.DataLoader(
        dataset,
        batch_size=1,
        shuffle=False,
        sampler=None,
        batch_sampler=None,
        num_workers=0,
        ...)
    ```
    ```

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

`torch.nn.Module`

- `torch.nn.Module`
  - define the modules of the network
  - initialize the parameters
  - define the computing methods in the network
- Usage: inherit the `torch.nn.Module` and implement
  - `def forward(self, x)`
    - defines the computation performed at every call.

Example

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

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Loss Function

- Loss function
  - `torch.nn.*Loss`
    - `torch.nn.L1Loss`
    - `torch.nn.MSELoss`
    - … Ref: https://pytorch.org/docs/stable/nn.html#loss-functions
  - `torch.nn.functional.*`
    - `torch.nn.functional.binary_cross_entropy`
    - `torch.nn.functional.binary_cross_entropy_with_logits`
    - … Ref: https://pytorch.org/docs/stable/nn.html#id51
  - define your own loss
    - check if your loss function is differentiable

Optimizer: `torch.optim`

- Optimizer: optimization algorithm for loss function
- Example:

```
import torch.optim as optim
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Outline

- Installation
- Basic Data Type
- Data Processing
- Define a Model
- Train and test

Training

- Pipeline:
  1. Load data from the dataloader
  2. Forward: pass the data through the network and get the output
  3. Backward: run `loss.backward()` to compute the gradients for all the parameters
  4. Repeat until convergence

Example

```python
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        # forward
        outputs = model(inputs)
        loss = F.cross_entropy(outputs, targets)

        # backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Testing

- No need to use `.backward()`

```
def test(model, device, test_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for inputs, target in test_loader:
            inputs, target = inputs.to(device), target.to(device)
            outputs = model(inputs)
            pred = outputs.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
```

Reference

- https://pytorch.org/tutorials/
- https://pytorch.org/docs/stable/index.html

Q&A