Deep Learning with PyTorch 1.8.1

Presenter: Hao-Ting Li (李皓庭)

Date: 2021/04/07 (Wed.)

https://hackmd.io/@aquastripe/deep-learning-with-pytorch-1-8-1

Outline

- Introduction
- Installation
- Basic Data Type and Operation
- Data Processing
- Building Neural Networks
- Training, Validating and Testing

What is PyTorch

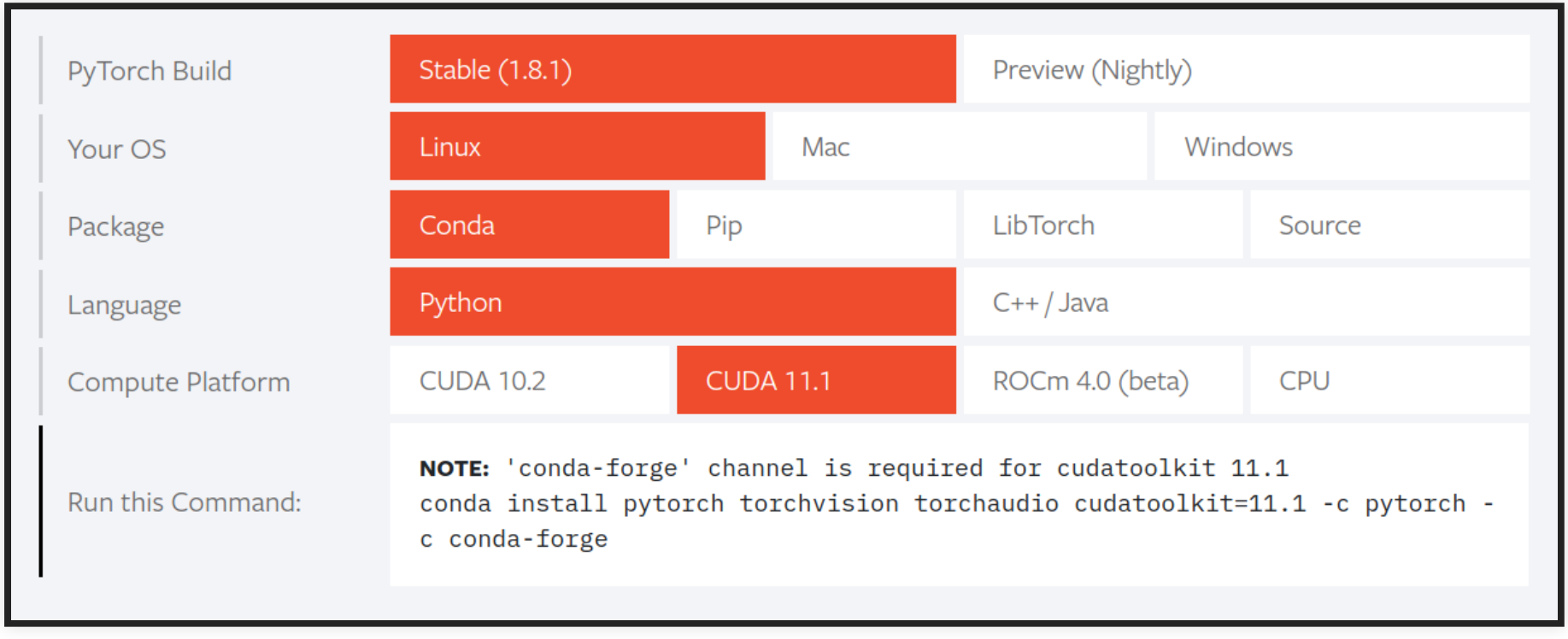
PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). [wiki]

Features:

- GPU computing
    - CUDA for NVIDIA graphic card (recommended)
    - ROCm for AMD graphic card
- Automatic differentiation
- Dynamic Computational Graphs

Installation

https://pytorch.org/get-started/locally/



Checking GPU Support

```
import torch
torch.cuda.is_available()
```

Output:

```
True
```

Basic Data Types and Math Operations

- Terms
  - tensor: a multi-dimensional array
  - shape: the length (number of elements) of each of the axes of a tensor
    - e.g., $(10, 3, 256, 256)$

In PyTorch, images are represented as `[batch, channels, height, width]` in $[0, 1]$, where the channel's order is **RGB**.

Initializing a Tensor

From a Python built-in `List`:

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data, dtype=torch.float32)
```

From a NumPy array:

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

Initializing a Tensor

From another tensor:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Output:

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.3894, 0.5955],
        [0.8305, 0.0720]])
```

Attributes of a Tensor

https://pytorch.org/docs/stable/tensor_attributes.html

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Output:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operations on Tensors

Standard numpy-like indexing and slicing:

```
tensor = torch.ones(4, 4)
print('First row: ',tensor[0])
print('First column: ', tensor[:, 0])
print('Last column:', tensor[..., -1])
tensor[:,1] = 0
print(tensor)
```

Output:

```
First row:  tensor([1., 1., 1., 1.])
First column:  tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Ref: https://numpy.org/doc/stable/reference/arrays.indexing.html

Operations on Tensors

Joining (concatenating) tensors:

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

Output:

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

Operations on Tensors

Arithmetic operations:

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)
y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)

# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)
z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

Operations on Tensors

Single-element tensors: you can convert it to a Python primitive numerical value using `item()`.

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

Output:

```
12.0 <class 'float'>
```

Operations on Tensors

In-place operations: store the result into the operand.

```
print(tensor, "\n")
tensor.add_(5)
print(tensor)
```

Output:

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

Tensor to NumPy Array

call `numpy()`

```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

Output:

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

Move Tensors to the GPU/CPU

1. `to()`
2. `cuda()` or `cpu()`

```
# 1.1. input a string
tensor = tensor.to('cuda')
tensor = tensor.to('cuda:0')    # specify the GPU ID

# 1.2. input a device
device = torch.device('cuda')
tensor = tensor.to(device)

# 2.
tensor = tensor.cuda()
tensor = tensor.cuda(1)         # specify the GPU ID
tensor = tensor.cpu()
```

Automatic Differentiation

```
>>> x = torch.randn(5, 5)  # requires_grad=False by default
>>> y = torch.randn(5, 5)  # requires_grad=False by default
>>> z = torch.randn((5, 5), requires_grad=True)
>>> a = x + y
>>> a.requires_grad
False
>>> b = a + z
>>> b.requires_grad
True
```

Context Managers for `requires_grad` Settings

1. Set `requires_grad` to `True`:

```
with torch.set_grad_enabled(True):
    ...
```

or

```
with torch.enable_grad():
    ...
```

2. Set `requires_grad` to `False`:

```
with torch.set_grad_enabled(False):
    ...
```

or

```
with torch.no_grad():
    ...
```

Decorators for `requires_grad` Settings

```
@torch.enable_grad()
def doubler(x):
    return x * 2
```

Freeze the Model's Parameters

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

Ref: https://pytorch.org/docs/stable/notes/autograd.html

Data Processing

- Raw data loading & data augmentation
  - `torchvision.transforms.*`, e.g.:
    - `torchvision.transforms.Compose(transforms)`
    - `torchvision.transforms.Resize(size, interpolation=<InterpolationMode.BILINEAR: 'bilinear'>)`
    - `torchvision.transforms.Normalize(mean, std, inplace=False)`
    - Ref: https://pytorch.org/vision/stable/transforms.html
- `Dataset` & `DataLoader`
  - `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`
  - `torch.utils.data.DataLoader`
  - Ref: https://pytorch.org/docs/stable/data.html

Image Loading Example

```
from PIL import Image
import torchvision.transforms.transforms as T

transforms = T.Compose([
    # ...
    # data augmentation operations
    T.Resize(256),
    T.ToTensor(),
])

with Image.open(filename).convert('RGB') as image:  # open image as `PIL.Image` object
    if transforms:
        image = transforms(image)
        # here, the image are transformed to `torch.Tensor`
```

Dataset

- An interface for `DataLoader`
- **Map-style** and **iterable-style**
    - Map-style: supports random access.

```
idx = 123
dataset[idx]
```

    - Iterable-style: is efficient for large scale datasets.

```
next(dataset)
```

Ref: https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

Map-style Dataset Usage

inherit the `Dataset` and implement:

- `def __getitem__(self, index)`
  - return a batch data, return type: `torch.Tensor` or built-in collections (e.g., `dict`, `list`, ...) of `torch.Tensor`
- `def __len__(self)`
  - return the size of the dataset, return type: `int`

Iterable-style Dataset Usage

inherit the `IterableDataset` and implement:

- `def __iter__(self):`
    - return an **iterator** of a batch data.

DataLoader

```
dataloader = DataLoader(
        dataset,
        batch_size=32,
        shuffle=True,    # set to True in training stage and False in others
        sampler=None,
        num_workers=8,   # for multi-processing
        ...
)

for i, batch in enumerate(dataloader):
    ...
```

Building Neural Networks

- `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

```
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))

model = Model()
output = model(batch_data)  # forward
```

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.l1_loss`
    - `torch.nn.functional.mse_loss`
    - Ref: https://pytorch.org/docs/stable/nn.functional.html#loss-functions
- define own loss functions
    - need to check if the loss function is differentiable

Optimizer

- Optimizers are optimization algorithms for learning.
- `torch.optim.*`

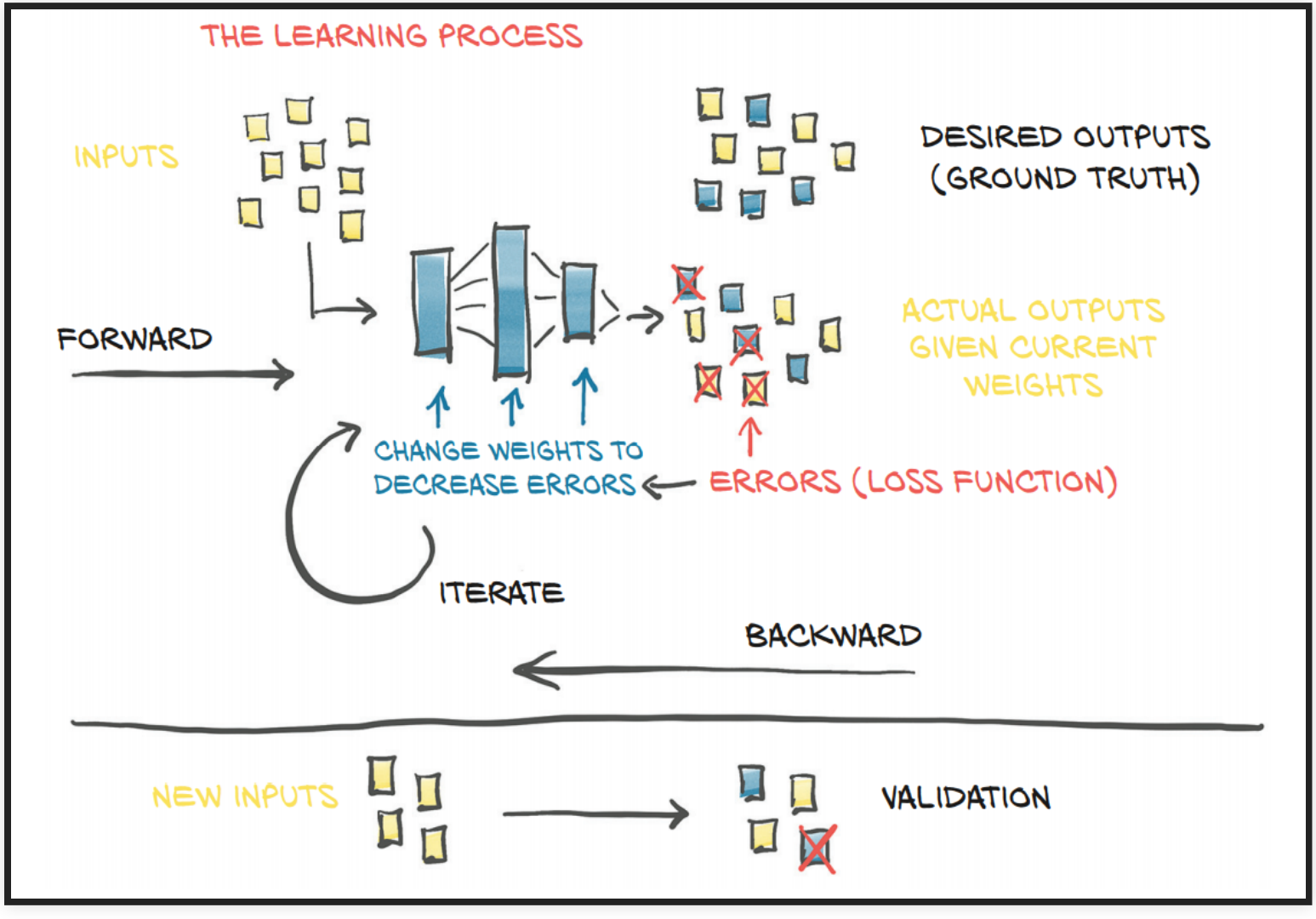
```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

- Update the parameters
    1. compute gradients
    2. call `optimizer.step()`

Per-parameter Options

```
optim.SGD([
    {'params': model.base.parameters()},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
], lr=1e-2, momentum=0.9)
```

Training, Validating and Testing



Training

1. Loop: 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 the loops until convergence

Example

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()   # for Dropout, Batch Normalization, etc.

    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

```
def test(model, device, test_loader):
    model.eval()    # equals to model.train(False)
    correct = 0
    for inputs, target in test_loader:
        inputs, target = inputs.to(device), target.to(device)

        with torch.no_grad():
            # forward
            outputs = model(inputs)

            # evaluation, e.g.
            pred = outputs.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
```

Q&A