Introduction

■ Why we need deep learning frameworks
  (1) Easily build **big** computational graphs
  (2) Easily compute **gradients** in computational graphs
  (3) Run it all efficiently on **GPU**

■ Why we need Pytorch?
  (1) Easy to implement, code, and debug
  (2) More flexible due to its **dynamic** computational graph.
  (3) High execution efficiency, since it developed from C.
PyTorch: Three Levels of Abstraction

**Tensor**: Like array in Numpy, but runs on GPU

**Variable**: Node in a computational graph; stores data and gradient

**Module**: A neural network layer; may store state or learnable weights
PyTorch: Tensors

- PyTorch Tensors are just like numpy arrays, but they can run on GPU.
- No built-in notion of computational graph, or gradients, or deep learning.
- Here we fit a two-layer net using PyTorch Tensors

```python
import torch
dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
h_relu = h.clamp(min=0)
y_pred = h_relu.mm(w2)
loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```
PyTorch: Tensors

To run on GPU, just cast tensors to a cuda data type!
*(E.g. torch.cuda.FloatTensor)*

Create random tensors for data and weights.

Forward pass:
compute predictions and loss

Backward pass:
manually compute gradients

Gradient descent step on weights

```python
import torch
dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h.relu().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```
Tensor: Operations

Create a tensor:

An empty tensor can be constructed by specifying its size:

```python
>>> torch.IntTensor(2, 4).zero_()
 0 0 0 0
 0 0 0 0
[Torch.IntTensor of size 2x4]
```

Math Operation:

```python
torch.dot(tensor1, tensor2) → float
```

Computes the dot product (inner product) of two tensors.

Other operations:

```python
torch.squeeze(input, dim=None, out=None)
```

Returns a tensor with all the dimensions of `input` of size 1 removed.

For example, if `input` is of shape: 

\[(A \times 1 \times B \times C \times 1 \times D)\]

then the `out` tensor will be of shape:

\[(A \times B \times C \times D)\].

```python
torch.unsqueeze(input, dim, out=None)
```

Returns a new tensor with a dimension of size one inserted at the specified position.

```python
torch.cat(seq, dim=0, out=None) → Tensor
```

Concatenates the given sequence of `seq` Tensors in the given dimension.
A PyTorch Variable is a node in a computational graph
- x.data is a Tensor
- x.grad is a Variable of gradients (same shape as x.data)
- x.grad.data is a Tensor of gradients
- PyTorch Tensors and Variables have the same API!
- Variables remember how they were created (for backprop)

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad:
        w1.grad.data.zero_()
    if w2.grad:
        w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```
PyTorch: Autograd

We will not want gradients (of loss) with respect to data.

Do want gradients with respect to weights.

Forward pass looks exactly the same as the Tensor version, but everything is a variable now.

Compute gradient of loss with respect to $w_1$ and $w_2$ (zero out grads first).

Make gradient step on weights.

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

w1.data -= learning_rate * w1.grad.data
w2.data -= learning_rate * w2.grad.data
```
PyTorch: New Autograd Functions

```python
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)
    def backward(self, self.saved_tensors, grad_y):
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    relu = ReLU()
    y_pred = relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Can use our new autograd function in the forward pass.
PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward for Tensors.

```python
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```
PyTorch: nn

Higher-level wrapper for working with neural nets

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```
PyTorch: nn

Define our model as a sequence of layers

nn also defines loss functions

Forward pass: feed data to model, and prediction to loss function

Backward pass: compute all gradients

Make gradient step on each model parameter

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```
Example: Define your own Module

```python
import torch.nn as nn

class discriminator(nn.Module):
    def __init__(self, d=128):
        super(discriminator, self).__init__()
        self.conv1 = nn.Conv2d(1, d, 4, 2, 1)
        self.conv2 = nn.Conv2d(d, d**2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(d**2)
        self.conv3 = nn.Conv2d(d**2, d**4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(d**4)
        self.conv4 = nn.Conv2d(d**4, d**8, 4, 2, 1)
        self.conv4_bn = nn.BatchNorm2d(d**8)
        self.conv5 = nn.Conv2d(d**8, 1, 4, 1, 0)

    def forward(self, input):
        x = F.leaky_relu(self.conv1(input), 0.2)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = F.leaky_relu(self.conv4_bn(self.conv4(x)), 0.2)
        x = F.sigmoid(self.conv5(x))
        return x
```

You can find more details about API in the following link:

http://pytorch.org/docs/master/index.html
PyTorch: optim

Use an optimizer for different update rules

Update all parameters after computing gradients

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```
PyTorch: nn Define new Modules

Define our whole model as a single Module

Initializer sets up two children (Modules can contain modules)
*Note: No need to define backward - autograd will handle it*

Define forward pass using child modules and autograd ops on Variables

Construct and train an instance of our model

```python
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```
MNIST Example

https://github.com/pytorch/examples/blob/master/mnist/main.py
Extending PyTorch

- `torch.autograd`
- customize `forward()` and `backward()`
- check gradient
- from `torch.autograd` import `gradcheck`
Extending PyTorch

- torch.autograd
- Adding function to a Module

```python
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        super(Linear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Variable, that will get # automatically registered as Module's parameter once it's assigned # as an attribute. Parameters and buffers need to be registered, or # they won't appear in .parameters() (doesn't apply to buffers), and # won't be converted when e.g. .cuda() is called. You can use # .register_buffer() to register buffers. # nn.Parameters can never be volatile and, different than Variables, # they require gradients by default.
        self.weight = nn.Parameter(torch.Tensor(output_features, input_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(output_features))
        else:
            # You should always register all possible parameters, but the # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        self.weight.data.uniform_(-0.1, 0.1)
        if bias is not None:
            self.bias.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return LinearFunction.apply(input, self.weight, self.bias)
```
Extending PyTorch

- **C-extension**
  - a. prepare your C code

```c
/* src/my_lib.c */
#include <TH/TH.h>

int my_lib_add_forward(THFloatTensor *input1, THFloatTensor *input2, THFloatTensor *output) {
    if (THFloatTensor_isSameSizeAs(input1, input2))
        return 0;
    THFloatTensor_resizeAs(output, input1);
    THFloatTensor_cadd(output, input1, 1.0, input2);
    return 1;
}

int my_lib_add_backward(THFloatTensor *grad_output, THFloatTensor *grad_input) {
    THFloatTensor_resizeAs(grad_input, grad_output);
    THFloatTensor_fill(grad_input, 1);
    return 1;
}
```

header with all functions

c file
Extending PyTorch

- C-extension
  - b. build custom extension
    
    ```python
    # build.py
    from torch.utils.ffi import create_extension
    ffi = create_extension(
        name='_ext.my_lib',
        headers='src/my_lib.h',
        sources=['src/my_lib.c'],
        with_cuda=False
    )
    ffi.build()
    ```

    run it and get a new folder with a .so file

  - c. include it in your Python code
    
    ```python
    # functions/add.py
    import torch
    from torch.autograd import Function
    from _ext import my_lib

    class MyAddFunction(Function):
        def forward(self, input1, input2):
            output = torch.FloatTensor()
            my_lib.my_lib_add_forward(input1, input2, output)
            return output

        def backward(self, grad_output):
            grad_input = torch.FloatTensor()
            my_lib.my_lib_add_backward(grad_output, grad_input)
            return grad_input
    ```
Torchvision

- popular datasets
  - cifar10, coco, lsun, mnist, ...

- popular model architectures (pretrained)
  - alexnet, densenet, inception, resnet, squeezenet, vgg

- common image transformations
  - Normalize, Scale, CenterCrop, Pad, RandomCrop, RandomFlip, ...
  - Compose
Visualization

- **TensorboardX**
  - pip install tensorboardX
  - tensorboard for pytorch (and chainer, mxnet, numpy, ...)
  - Support scalar, image, histogram, audio, text, graph, onnx_graph, embedding and pr_curve

```python
import torch
import torchvision.transforms as transforms
import numpy as np
import torchvision.models as models
from torchvision import datasets
from tensorboardX import SummaryWriter

resnet18 = models.resnet18(False)
writer = SummaryWriter()
sample_rate = 44100
freq = [262, 294, 330, 349, 392, 440, 440, 440, 440, 440]

for n_iter in range(100):
    dummy_s1 = torch.randn(1)
    dummy_s2 = torch.randn(1)
    writer.add_scalar('data/scalar1', dummy_s1[0], n_iter)
    writer.add_scalar('data/scalar2', dummy_s2[0], n_iter)
    writer.add_scalars('data/scalar_group', {'sin': n_iter * np.sin(n_iter),
                                            'cos': n_iter * np.cos(n_iter),
                                            'sine': np.sin(n_iter) * n_iter})

dummy_img = torch.randn(32, 3, 64, 64) # output from network
if n_iter % 10 == 0:
    writer.add_image('image', dummy_img[0], n_iter)

dummy_audio = torch.zeros(sample_rate * 2)
for i in range(10):
    dummy_audio[i] = np.cos(freq[i] / 10) * np.exp(i / (10 * sample_rate))
writer.add_audio('myAudio', dummy_audio, n_iter, sample_rate=sample_rate)

for name, param in resnet18.named_parameters():
    writer.add_histogram(name, param.clone().cpu().data.numpy(), n_iter)

writer.add_pr_curve('x2o', np.random.randint(2, size=100), np.random.randint(100, n_iter))

dataset = datasets.MNIST('MNIST', train=False, download=True)
images = dataset.test_data[100].float()
labels = dataset.test_labels[100]

writer.add_embedding(features, metadata=label, label_img=images.unsqueeze(1))

# export scalar data to JSON for external processing
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```
Visualization

- TensorboardX
  a. Run the demo script: python demo.py
  b. Use TensorBoard with tensorboard --logdir runs (needs to install TensorFlow)
Visualization

- Other choices

- Visdom

- Seaborn
Onnx

- Open Neural Network Exchange (ONNX)
- an open source format to move models between tools
- defines an extensible computation graph model, built-in operators and standard data types
- Hardware Optimizations
- Supported Tools

Frameworks

Converters

Runtimes
References

1. http://pytorch.org/
Thank You!