Introduction to PyTorch

COS484 2023 Spring

Howard Chen

Princeton NLP
Agenda

1. Introduction
2. Tensors
3. Autograd
4. Loss
Why use deep learning libraries?

- Quickly implement and test new ideas
- No need to implement your own neural networks, just use their (likely more efficient) implementation
- Automatically compute gradients (!!)
- Efficiently run on GPUs to speed up computations with few changes
Deep Learning Frameworks

Caffe
(UC Berkeley)

Torch
(NYU / Facebook)

Theano
(U Montre‌al)

2007-2015
Deep Learning Frameworks

Caffe (UC Berkeley) → Caffe2 (Facebook)

Torch (NYU / Facebook) → PyTorch (Facebook)

Theano (U Montreal) → TensorFlow (Google)

2016-2020
Deep Learning Frameworks

- Caffe (UC Berkeley)
- Torch (NYU / Facebook)
- Theano (U Montreal)
- Caffe2 (Facebook)
- PyTorch (Linux Foundation)
- TensorFlow (Google)
- JAX (Google)

2020-present
Deep Learning Frameworks

- **Caffe** (UC Berkeley)
- **Torch** (NYU / Facebook)
- **Theano** (U Montreal)
- **Caffe2** (Facebook)
- **PyTorch** (Linux Foundation)
- **TensorFlow** (Google)
- **JAX** (Google)
Deep Learning Frameworks

DyNet
(AMU)

CNTK
(Microsoft)

Chainer
(Preferred Networks)

MXNet
(Amazon)

Other frameworks that tried to make a dent
Installing PyTorch: Google Colab

· We assume you’ll be using Google Colab for your projects
· torch (PyTorch) should come pre-installed in the colab environment
  · If it isn’t, you can always install outside packages using:
    !pip install torch
PyTorch Preview

- API is very clean and code is very readable
- No need to compute gradients, just use `.backward()`!

```
import numpy as np
np.random.seed(0)

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
C = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

```
import torch
torch.manual_seed(0)

N, D = 3, 4
x = torch.randn((N, D), requires_grad=True)
y = torch.randn((N, D), requires_grad=True)
z = torch.randn((N, D), requires_grad=True)

a = x * y
b = a + z
C = torch.sum(b)

c.backward()
```
torch.Tensor: Introduction

- Exactly like NumPy arrays, but can be run efficiently on GPUs
- Supports the same operations like indexing, slicing, reshaping, transpose, cross product, matrix product, element-wise multiplication, …
torch.Tensor: Gradients

requires_grad - Makes this a trainable parameter

- False by default
- Turn on:
  
```
t.requires_grad_()
t = torch.randn(1, requires_grad=True)
```
- Tensor value = t.data
- Gradient value = t.grad
- History of autograd operations = t.grad_fn

Note: Functions that end with an underscore _ modify the tensor in-place!
Tensors: Devices

Check if a GPU is available:
```
torch.cuda.is_available()
```

Convert a `numpy.array` to `torch.Tensor`:
```
torch.from_numpy(x_train)
```
# this returns a cpu tensor

Convert back to `numpy`:
```
t.numpy()
```

Move tensor to a device:
```
t.to('cuda') or t.to('cpu')
```

Check tensor or array type:
```
type(t) or t.type()
```
Autograd: Introduction

- Automatic differentiation package
- `t.backward()` calculates gradients along the computational graph
- Gradients are accumulated by default
  - Need to zero them out after each update
    - `t.grad.zero_()`
    - (Typically done with Optimizer!)
Autograd: Introduction

Automatic differentiation package

We can easily compute gradients with:

```python
t.backward()  # computes along the computation graph
```

Gradients are accumulated by default

We can update the parameter weight using the gradient as:

```python
w -= lr * w.grad
```

After updating, you need to reset the gradients by clearing their values:

```python
w.grad_.zero()```
**Autograd: Manual Update Example**

- Definition
- Forward pass
- Backward pass
- Weight update
- Reset grads

```python
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    with torch.no_grad():
        a -= lr * a.grad
        b -= lr * b.grad

    a.grad.zero_()
    b.grad.zero_()

print(a, b)
```
Optimizer

The `torch.optim` library makes updating the parameters easier:

We can use different optimization algorithms, like Adam, SGD, RMSprop.

We compute the gradients like before: `t.backward()`

But now we use the optimizer to apply param updates: `optimizer.step()`

Zero gradients with: `optimizer.zero_grad()`

```python
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
Loss Functions

- Use pre-implemented loss functions too!
- L1, MSE, Cross-Entropy, …

```python
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
Precept on Friday

- Model
- Dataset
- Evaluation