PART B. GEAR SESSIONS
SESSION 2: MACHINE LEARNING FOR BIG DATA

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

• CS535 Online
• Please read announcement on Canvas
  • If you have any questions, please post on Piazza
Topics of Today's Class

- Distributed PyTorch
  - Some common advanced optimizations
  - You will use it for your term project
- Automatic Differentiation with Backpropagation
- Computation Graph
- Distributed PyTorch Application

GEAR Session 2. Machine Learning for Big Data
Lecture 4. Distributed Neural Networks-PyTorch

PyTorch: Introduction
This material is built based on


Observations

- Array-based programming
  - Multidimensional arrays (A.K.A. tensors) became critical mathematical data type
- Automatic differentiation enabled fully automated computing of derivatives
- Open-source Python ecosystem for numerical analysis
  - NumPy, SciPy and Pandas
- Availability and commoditization of general-purpose massively parallel hardware
  - GPUs
  - Specialized libraries, cuDNN
- Caffe, Torch7, TensorFlow take advantage of these hardware accelerators
Programming Environment

- Coping with increased computational complexity
- Easy implementation of new neural network architectures
- Layers
  - Expressed as Python classes
- Models
  - Classes that compose layers

```python
class LinearLayer(nn.Module):
    def __init__(self, in_sz, out_sz):
        super().__init__()
        t1 = torch.randn(in_sz, out_sz)
        self.w = nn.Parameter(t1)
        t2 = torch.randn(out_sz)
        self.b = nn.Parameter(t2)
    def forward(self, activations):
        t = torch.mm(activations, self.w)
        return t + self.b

class FullBasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 128, 3)
        self.fc = LinearLayer(128, 10)
    def forward(self, x):
        t1 = self.conv(x)
        t2 = nn.functional.relu(t1)
        t3 = self.fc(t1)
        return nn.functional.softmax(t3)
```

Building Generative Adversarial Networks

```
def step(real_sample):
    # (1) Update Discriminator
    errD_real = loss(discriminator(real_sample), real_label)
    errD_real.backward()
    fake = generator(get_noise())
    errD_fake = loss(discriminator(fake.detach()), fake_label)
    errD_fake.backward()
    optimD.step()
    # (2) Update Generator
    errG = loss(discriminator(fake), real_label)
    errG.backward()
    optimG.step()
```
Training Networks

- Gradient based optimization is critical to deep learning
- Automatically compute gradients of models specified by users

Challenge
- Python is a dynamic programming language that allows changing most behaviors at runtime

PyTorch uses the operator overloading approach
- builds up a representation of the computed function every time it is executed
What is an Automatic Differentiation (AD)?

- A set of techniques to numerically evaluate the derivative of a function specified by a computer program
  - *Automatic Differentiation lets you compute exact derivatives in constant time*

Is AD same as Symbolic Differentiation?

- No
  - *Symbolic differentiation* breaks apart a complex expression into a bunch of simpler expressions by using various rules
  - **Examples**
    - *Sum rule*: \( \frac{d}{dx} (f(x) + g(x)) = \frac{d}{dx} f(x) + \frac{d}{dx} g(x) \)
    - *Constant rule*: \( \frac{d}{dx} C = 0 \)
    - *Derivatives of powers rule*: \( \frac{d}{dx} x^n = n x^{n-1} \)
  - **Disadvantages**
    - For complicated functions, the result expression can be extremely large
    - Wasteful to keep around intermediate symbolic expressions if we only need a numeric value of the gradient in the end
    - Prone to error
Is AD same as Numeric Differentiation?

- No

**Numeric Differentiation** is an algorithm for estimating the derivative of a mathematical function or function subroutine.

- Example: Simple approximation of the first derivative
  - Two-point estimation
  - Slope of a nearby secant line through the points \((x, f(x))\) and \((x+h, f(x+h))\) for small number \(h\)
  - 
  \[
  f'(x) \approx \frac{f(x+h) - f(x)}{h}
  \]
  - Where we assume that \(h > 0\)

---

Numeric Differentiation

- **Pros**
  - A powerful tool to check the correctness of implementation, usually use \(h = 1e^{-6}\).

- **Cons**
  - Rounding error and slow to compute
AD with a Simple Example

• Dual numbers
• Numbers of the form $a + b\epsilon$, where $\epsilon^2 = 0$
• Suppose that there are two dual numbers, $a + b\epsilon$ and $c + d\epsilon$
  - $(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$
  - $(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + bde^2 = ac + (ad + bc)\epsilon$

Taylor’s series with a dual number

• Plain Taylor’s series
  - $f(a) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x - a)^n = f(a) + \frac{f'(a)}{1!} (x - a) + \frac{f''(a)}{2!} (x - a)^2 + \cdots$
  - Approximate f about a real number $a + \epsilon$
    - $f(a + \epsilon) = f(a) + \frac{f'(a)}{1!} \epsilon + \frac{f''(a)}{2!} \epsilon^2 + \cdots$
    - $f(a + \epsilon) = f(a) + \epsilon f'(a)$
• Example
  - $f(x) = x^2 + 1$
  - $f(x + \epsilon) = (x + \epsilon)^2 + 1 = x^2 + \epsilon^2 + 2x\epsilon + 1$
  - $f(x + \epsilon) = (x^2 + 1) + 2x\epsilon$
  - Therefore $2x$ is the derivative of $x^2 + 1$
GEAR Session 2. Machine Learning for Big Data
Lecture 4. Distributed Neural Networks-PyTorch

PyTorch: Automatic Differentiation

Backpropagation

Training Neural Networks

- A **forward pass** to compute the **value of the loss function**
- A **backward pass** to compute the **gradients of the learnable parameters**
Backpropagation

\[ z = f(x, y) \]

Computing gradient becomes local computation

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial x} \]

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial y} \]
Simple Backpropagation Example

\[ f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \]
Simple Backpropagation Example

\[ f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \]

\[
\begin{align*}
\text{f(x)} &= \frac{1}{x} \\
\frac{df}{dx} &= -\frac{1}{x^2} \\
\frac{d^2f}{dx^2} &= -\frac{2}{x^3}
\end{align*}
\]
Simple Backpropagation Example

\[ f(x) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \]

\[ f(x, w) = xw \rightarrow \frac{\partial f}{\partial x} = w, \frac{\partial f}{\partial w} = x \]
GEAR Session 2. Machine Learning for Big Data
Lecture 4. Distributed Neural Networks-PyTorch
PyTorch: Automatic Differentiation

Computation Graph

AD with Computation Graph

• Similar to the graph that we have used
• However, the nodes in a computation graph are “operators”
  • Mathematical operators or user defined variable (for specific cases)
  • Leaf nodes for the leaf variables
Automatic Differentiation with Computation Graph

- Create computation graph for gradient computation

\[
f = \frac{1}{1 + e^{-(w_0 + w_1x_1 + w_2x_2)}}
\]

\[
f(x) = \frac{1}{x} \Rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}
\]
Automatic Differentiation with Computation Graph

• Create computation graph for gradient computation

\[ f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \]

\[ f(x) = x + 1 \Rightarrow \frac{df}{dx} = 1 \]
Automatic Differentiation with Computation Graph

- Create computation graph for gradient computation

\[ f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} \]

\[ f(x, w) = xw - \frac{\partial f}{\partial w} = x \]

\[ \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2} \]
Example

• What is \( \frac{\partial f}{\partial w_2} \)?

• Step 1: Trace down all possible paths from the first node to \( w_2 \)
  • There is only one such path

• Step 2: Multiply all the edges along this path (in this case)

\[
\begin{align*}
\frac{\partial f}{\partial x_1} &= a \cdot b \\
\frac{\partial f}{\partial x_2} &= c \cdot d \\
\frac{\partial f}{\partial w_0} &= e \\
\frac{\partial f}{\partial w_1} &= f \\
\frac{\partial f}{\partial w_2} &= g
\end{align*}
\]

GEAR Session 2. Machine Learning for Big Data
Lecture 4. Distributed Neural Networks-PyTorch
PyTorch: Automatic Differentiation
PyTorch AutoGrad
PyTorch AutoGrad [1/2]

- Implementation of computational graph in PyTorch

- Tensor
  - Data structure similar to numpy arrays (ndarray)
  - Supporting parallelism with GPU

```python
In [1]: import torch
In [2]: tsr = torch.Tensor(3,5)
In [3]: tsr
```

```
tensor([[ 0.0000e+00, 0.0000e+00, 8.4452e-29, -1.0842e-19, 1.2413e-35],
        [ 1.4013e-45, 1.2416e-35, 1.4013e-45, 2.3331e-35, 1.4013e-45],
        [ 1.0108e-36, 1.4013e-45, 8.3641e-37, 1.4013e-45, 1.0040e-36]])
```

PyTorch AutoGrad [2/2]

- `requires_grad` attribute of the Tensor should be set at True

```python
>> t1 = torch.randn((3,3), requires_grad = True)
>> t2 = torch.FloatTensor(3,3) # No way to specify requires_grad while initiating
>> t2.requires_grad = True
```

- `requires_grad` is propagated
  - If any of tensors that the current tensor is operating on has the `requires_grad` attribute as true, the current tensor will be also set as true
Simple example with PyTorch

- Consider a very simple network with 5 neurons
  - \( b = w_1 \times a \)
  - \( c = w_2 \times a \)
  - \( d = w_3 \times b + w_4 \times c \)
  - \( L = 10 - d \)

\[
\begin{aligned}
  \frac{\partial L}{\partial w_4} &= \frac{\partial L}{\partial d} \times \frac{\partial d}{\partial w_4} \\
  \frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial d} \times \frac{\partial d}{\partial w_3} \\
  \frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial d} \times \frac{\partial d}{\partial c} \times \frac{\partial c}{\partial w_2} \\
  \frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial d} \times \frac{\partial d}{\partial b} \times \frac{\partial b}{\partial w_1}
\end{aligned}
\]

- All these gradients have been computed by applying the chain rule
Implementing with PyTorch

- **grad_fn** attribute
  - Mathematical operator that creates the variable
  - If `requires_grad` is false, `grad_fn` would be `None`

```python
import torch
a = torch.randn((3,3), requires_grad = True)
w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)
b = w1*a
c = w2*a
d = w3*b + w4*c
L = 10 - d
print("The grad fn for a is", a.grad_fn) print("The grad fn for d is", d.grad_fn)
```

Implementing with PyTorch: Results

```python
import torch
a = torch.randn((3,3), requires_grad = True)
w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)
b = w1*a
c = w2*a
d = w3*b + w4*c
L = 10 - d
print("The grad fn for a is", a.grad_fn) print("The grad fn for d is", d.grad_fn)
```

```
The grad fn for a is None
The grad fn for d is <AddBackward0 object at 0x1033afe48>
```
Implementing with PyTorch: Functions  

- All mathematical operations in PyTorch are implemented by the `torch.nn.Autograd.Function` class
  
- **forward function**
  - Computes the output using its inputs
  
- **backward function**
  - Takes the incoming gradient coming from the part of the network in front of it
  - Generate the gradient to be backpropagated from a function $f$
    - (Gradient that is backpropagated to $f$ from the previous layers) $\times$ (Local gradient of the output of $f$ with respect to its inputs)

Tensor here is $d$

$d$’s `grad_fn` is `<ThAddBackward>` (an addition operation)

- **forward function** of $d$’s `grad_fn`
  - Inputs: $w_3b$ and $w_4c$
  - Operation: addition
  - This value is stored in the $d$

- **backward function** of $d$’s `grad_fn`
  - Inputs: incoming gradient from the previous layers ($L$)
  - Operation: Compute the local gradients and send them to inputs by invoking the backward method of the `grad_fn` of the inputs

\[
\begin{align*}
  b &= w_3 \times a \\
  c &= w_4 \times a \\
  d &= w_3 \times b + w_4 \times c \\
  L &= 10 - d
\end{align*}
\]
```python
def backward(incoming_gradients):
    self.Tensor.grad = incoming_gradients
    for inp in self.inputs:
        if inp.grad_fn is not None:
            new_incoming_gradients = incoming_gradient * local_grad(self.Tensor, inp)
            inp.grad_fn.backward(new_incoming_gradients)
        else:
            pass
```

- `backward` function is called **recursively**
  - Leaf node (grad_fn is None)

- `backward` can be called only on a **scalar valued Tensor**
  - Not on the vector-valued Tensor
PyTorch's graphs vs. TensorFlow graphs

• **PyTorch generates a Dynamic Computation Graph**
  • Graph is generated on the fly
  • Until the `forward` function of a Variable is called, no node for the Tensor in the graph

```python
a = torch.randn((3,3), requires_grad = True) #No graph yet, as a is a leaf
w1 = torch.randn((3,3), requires_grad = True) #Same logic as above
b = w1*a #Graph with node `mulBackward` is created.
```

• In PyTorch
  • When forward function is invoked
  • Buffers for the non-leaf nodes are allocated for the graph
  • When backward function is called
  • Above buffers (for non-leaf variables) are freed
  • Once the gradient is computed, the graph is destroyed
  • Next time, for `forward` on the same set of tensors
  • **The leaf node buffers from the previous run will be shared**
  • **The non-leaf nodes buffers will be created again**
PyTorch’s graphs vs. TensorFlow graphs

- **TensorFlow uses static computation graph**
  - The graph is declared *before* running the program
  - Then the graph is "run" by feeding inputs

- PyTorch’s dynamic graph allows changing the network architecture *during runtime*
  - A graph may be redefined during the lifetime for a program
  - Easy to debug
    - Easy to locate the source of error

---

**GEAR Session 2. Machine Learning for Big Data**

**Lecture 4. Distributed Neural Networks-PyTorch**

**PyTorch: Building a Distributed Application**
GEAR Session 2. Machine Learning for Big Data  
Lecture 4. Distributed Neural Networks-PyTorch

PyTorch: Building a Distributed Application

1. Message Passing Semantics
2. Communication Backends

Distributed PyTorch Application

- `torch.distributed`

- Parallelize computations across processes and clusters of machines
  - Message passing semantics
```python
# run.py
import os
import torch
import torch.distributed as dist
from torch.multiprocessing import Process

def run(rank, size): 
    
    def init_process(rank, size, fn, backend='gloo'):
        """ Initialize the distributed environment. """
        os.environ['MASTER_ADDR'] = '127.0.0.1'
        os.environ['MASTER_PORT'] = '29500'
        dist.init_process_group(backend, rank=rank, world_size=size)
        fn(rank, size)

    if __name__ == "__main__":
        size = 2
        processes = []
        for rank in range(size):
            p = Process(target=init_process, args=(rank, size, run))
            p.start()
            processes.append(p)
        for p in processes:
            p.join()
```

### Point-to-Point Communication

- A transfer of data from one process to another
- `send` and `recv` functions
  - Or their immediate counterparts, `isend` and `irecv`

- `send/recv` are **blocking**
  - Both processes stop until the communication is completed

- `isend/irecv` are **non-blocking**
  - Script continues its execution and the methods return a Work object upon which we can choose to `wait()`
  - Do not modify the sent tensor nor access the received tensor before `req.wait()` has completed
Collective Communication

- Communication patterns across all processes in a **group**
  - A group is a subset of all current processes
  - Creating a new group
    - Pass a list of ranks to dist.new_group(group)
    - By default, collectives are executed on all the processes (A.K.A. "world")

- Example
  - Calculate the sum of all tensors
  - ```python
  dist.all_reduce(tensor, op, group) collective
  ```

Collective Communication: Scatter

```python
dist.scatter(tensor, src, scatter_list, group)
```
-- Copies the \(i\)th tensor \(scatter\_list[i]\) to the \(i\)th process
Collective Communication: Gather

\[ \text{dist.gather(tensor, dst, gather_list, group)} \]
-- Copies tensor from all processes in dst

Collective Communication: Reduce

\[ \text{dist.reduce(tensor, dst, op, group)} \]
-- Applies op to all tensor and stores the result in dst
Collective Communication: All-Reduce

dist.all_reduce(tensor, op, group)
--Same as reduce, but the result is stored in all processes

Collective Communication: Broadcast

dist.broadcast(tensor, src, group)
--Copies tensor from src to all other processes
Collective Communication: All-Gather

\[
\text{dist.all_gather(tensor_list, tensor, group)}
\]

--Copies tensor from all processes to tensor_list, on all processes

GEAR Session 2. Machine Learning for Big Data
Lecture 4. Distributed Neural Networks-PyTorch
PyTorch: Building a Distributed Application
1. Message Passing Semantics
2. Communication Backend
Gloo Backend

- A collective communications library
  - Supports all point-to-point and collective operations on CPU, and all collective operations on GPU
  - Supports both Linux (since 0.2) and macOS (since 1.3)
  - Included in the pre-compiled PyTorch binaries

- The implementation of the collective operations for CUDA tensors is not as optimized as the ones provided by the NCCL backend

NCCL Backend

- Stand-alone library of standard collective communication routines for GPUs
  - Implementing all-reduce, all-gather, reduce, broadcast, and reduce-scatter
  - Optimized to achieve high bandwidth on platforms using PCIe, NVLink, Nvswitch
  - InfiniBand Verbs or TCP/IP sockets
  - Supports an arbitrary number of GPUs installed in a single node or across multiple nodes
  - Supports single- or multi-process application (e.g. MPI application)
MPI Backend

- The Message Passing Interface (MPI)
- Highly available and optimized for large clusters
- Leverages CUDA IPC and GPU Direct technologies
  - To avoid memory copies through the CPU
- PyTorch’s binaries can not include an MPI implementation
  - You should recompile it by hand

Questions?