ELEG5491:
Introduction to Deep Learning

Deep Learning in Action

Implementation, Techniques, and Frameworks

Yuanjun Xiong
PostDoc, MMLAB@CUHK

Outline

- Implementing Operations in Neural Networks
- Spinning Up Neural Networks
- Don’t Repeat Yourself: DL Frameworks

Implementing Operations

- Why is this not important?

```
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        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(input_features, output_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 Linear()(input, self.weight, self.bias)
```

Courtesy: PyTorch Documentation

Implementing Operations

- Then why waste my time?

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

# Inherit from Function
class Linear(Function):

    # bias is an optional argument
    def forward(self, input, weight, bias=None):
        self.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    def backward(self, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = self.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if self.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if self.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and self.needs_input_grad[2]:
            grad_bias = grad_output.sum(0).squeeze(0)

        return grad_input, grad_weight, grad_bias
```

What are these, seriously?

Implementing Operations

- Basic Concepts: GPU, Batching
- Implementing Basic Operations, in a Practical Manner
  - Linear Transformation
  - Convolution/Transposed Convolution
  - ReLu
  - SoftMax (with negative likelihood loss)
  - RNNs

Basic Concepts

- GPU: Parallelism Matters
  - A massively parallel computing device
  - Born to process graphics data
  - Emphasizes parallelism in programming practice
- Batching
  - Minibatch SGD
  - Process a number of samples/feature maps/input nodes in one shot
- Denotions:
  - L: loss function
  - $\delta = \frac{\partial L}{\partial f(\vec{x})}$: error signals

Decouple dependencies

Linear Transformation

- The fundamental operation
  - Inner Product, Fully-Connected, Matrix Multiplication, all you can name
  - Formulation

$$f_i(\vec{x}) = \sum w_{ij} x_j \qquad \frac{\partial L}{\partial x_j} = \sum \delta_i w_{ij} \qquad \frac{\partial L}{\partial w_{ij}} = \delta_i x_j$$

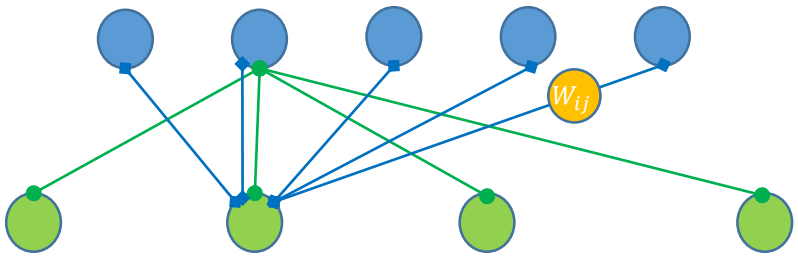
  - Batching n samples

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{pmatrix} \qquad W = \begin{pmatrix} w_{11} & \cdots & w_{k1} \\ \vdots & \ddots & \vdots \\ w_{1m} & \cdots & w_{km} \end{pmatrix}$$

$$f(X) = X \cdot W$$

$$\frac{\partial L}{\partial X} = \delta \cdot W^T$$

$$\frac{\partial L}{\partial W} = X^T \cdot \delta$$



Convolution

- What is convolution

$$[f * g](t) = \int_x f(x)g(t-x)\,dx$$

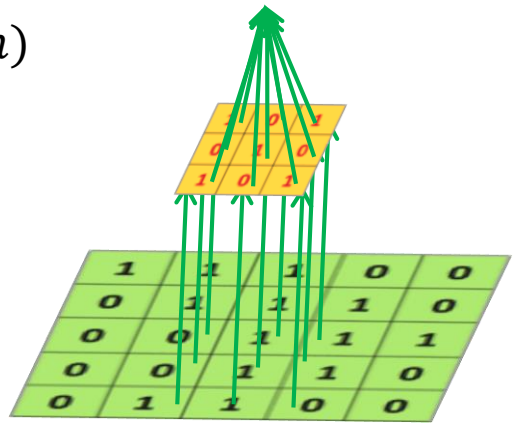
- Convolution in CNNs
  - One channel case: $X = x(i,j), W = w(i,j),$ k_w, k_h: $kernel\ sizes$

$$w(i,j) \neq 0 \Leftrightarrow |i| \leq k_h/2\ and\ |j| \leq k_w/2$$

$$f(l,m;X,W) = \sum_{i,j} x(i,j)w(i-l, j-m)$$

  - This is actually correlation
  - Multi-channel cases: inner products

$$f(l,m;X,W) = \sum_{i,j} < x(i,j), w(i-l, j-m) >$$



Convolution

- How about gradients?

$$\frac{\partial L}{\partial x_{ij}} = \sum_{l,m} \delta_{lm} \frac{\partial f_{lm}}{\partial x_{ij}} = \sum_{l,m} \delta(l,m)\, w(i-l, j-m)$$

Convolution, again!

$$\frac{\partial L}{\partial X} = \delta * W$$

$$\frac{\partial L}{\partial w_{ij}} = ?$$

$$f(l,m;X,W) = \sum_{i,j} x(i,j) w(i-l, j-m) = \sum_{i,j} x(i-l, j-m) w(i,j)$$

Convolution is commutative

$$\frac{\partial L}{\partial w_{ij}} = \sum_{l,m} \delta_{lm} \frac{\partial f_{lm}}{\delta w_{ij}} = \sum_{l,m} \delta(l,m) x(i-l, j-m)$$

$$\frac{\partial L}{\partial W} = \delta * X$$

Convolution

- Sounds good, how to implement?
  - 1. Matrix Multiplication

$$f(l,m;X,W)=\sum_{i,j}x(i-l,j-m)w(i,j)=<\vec{x}_{lm},\vec{w}>$$

    - Recall linear transform
  - 2. Direct Convolution/Correlation
    - $w(i,j)\neq 0 \Leftrightarrow |i|\leq k_h/2 \; and \; |j|\leq k_w/2$
    - Only compute the non-zero products and sum them up
- 3. Other considerations
  - Stride:
    - Compute $f(l,m;X,W)$ with stride of x,y
    - Gradients $\sum_{l,m}\rightarrow\sum_{l=\ ...,-x,0,x,...;m=\ ...,-y,0,y...}$, stride convolution
  - Padding

Deconvolution (Transposed Conv.)

- It's just fractional strided convolution (correlation)
  - Typically to up-sample input feature maps to larger sizes
  - Can learn up-sampling filters end-to-end

$$f_{deconv}(l, m; X, W) = \sum_{i,j} x\left(\lfloor \frac{i}{k} \rfloor, \lfloor \frac{j}{k} \rfloor\right) w(i - l, j - m), \qquad \text{k: stride}$$

Enlarge x

$$X' = x'(i, j) = x\left(\lfloor \frac{i}{k} \rfloor, \lfloor \frac{j}{k} \rfloor\right)$$

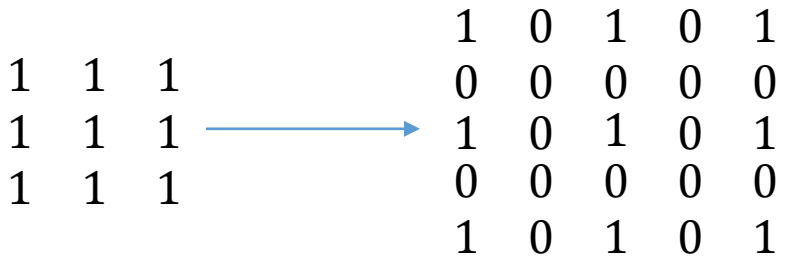
$$f_{deconv}(X) = X' * W$$

$$\frac{\partial L}{\partial X} = corr(\delta, W), stride\ k \qquad \frac{\partial L}{\partial W} = corr(\delta, X), stride\ k$$

  - Just reverse the forward and backward of normal convolution

Dilated Convolution

- Dilation: sparsifying kernels



$$f_{deconv}(l,m;X,W)=\sum_{i,j} w'\left(\frac{i}{2d},\frac{j}{2d}\right)x(i-l,j-m), \qquad \text{d: dilation}$$

$$w'(x,y)=\begin{cases} w(x,y), & if\ (x,y)\in Z^2 \\ 0, & else \end{cases}$$

- Same complexity, much larger kernel size
- Widely used in semantic segmentation

ReLU

- The dominant activation function
  - $f(x) = \max(0, x)$
  - Not everywhere differentiable: subgragient
  - Use $\delta f(x) = \begin{cases} 0, x \leq 0 \\ 1, x > 0 \end{cases}$
  - Computation:
    - $f(x) = \max(0, x)$
    - $\frac{\partial L}{\partial x} = \delta * 1[x > 0]$

SoftMax

- SoftMax: $R^n \rightarrow [0,1]^n$
- Normal formulation

$$SoftMax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- e^{x_i} can overflow! Try compute exp(100).

$$SoftMax(x_i) = \frac{e^{x_i - x_m}}{\sum_j e^{x_j - x_m}}, m = argmax_j\ (x_j)$$

- Use it with negative log likelihood loss $L = -\log(p_{i^*})$

$$\frac{\partial L}{\partial x_i} = 1[i = i^*] - SoftMax(x_i)$$

RNNs

- Recurrent, hmm…

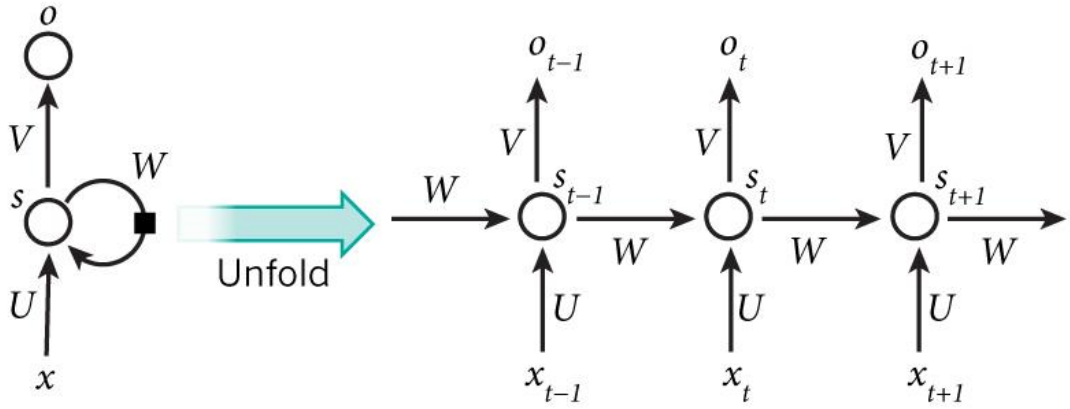


Photo from: Nature

- Just a bunch of linear transform and element wise add
- Try think of how to realize LSTM

Spinning Up Neural Networks

- With an arsenal of operations, how to start training your networks?
  - Which operation to start with?
  - Chain rules
  - Dynamic programming & topological sort
- Practical Issues
  - From multiple GPUs to multiple Nodes
  - Memory footprint
  - I/O: the (mostly) overlooked bottleneck

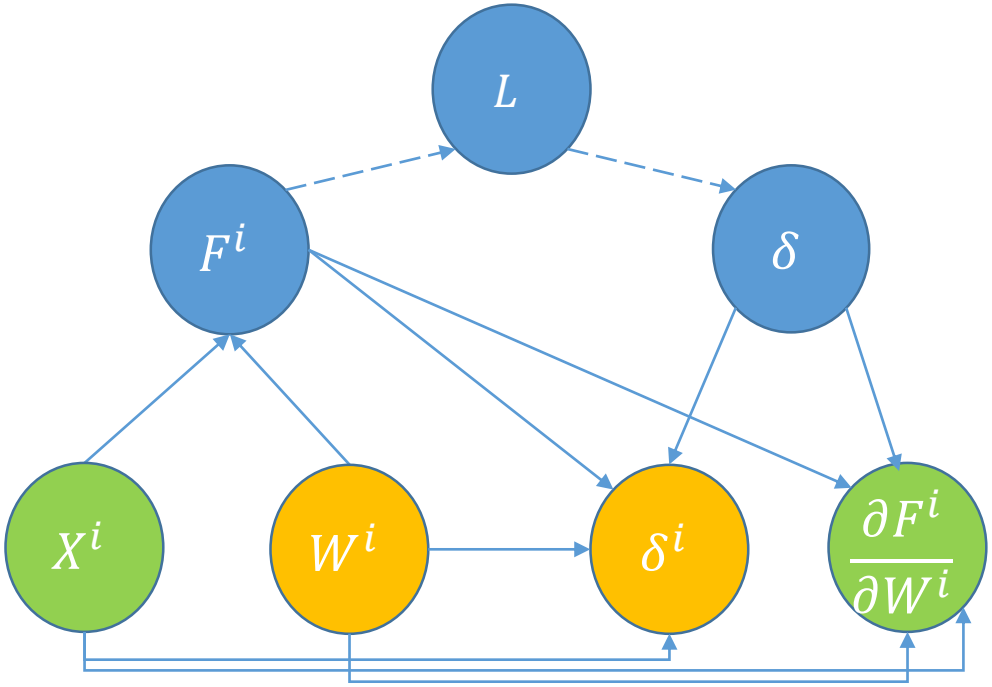
Chain Rule: Computing Gradients

- The plain old chain rule
  - Some operation in the networks

$$F^i = f^i(X^i; W^i)$$

$$\frac{\partial L}{\partial X^i} = \frac{\partial L}{\partial F^i}\frac{\partial F^i}{\partial X^i}, \qquad \frac{\partial L}{\partial W^i} = \frac{\partial L}{\partial F^i}\frac{\partial F^i}{\partial W^i}$$

- $\frac{\partial L}{\partial X^i} = \mathrm{g}(\delta, F^i, W^i, X^i)$
- $\frac{\partial F^i}{\partial W^i} = \mathrm{h}(\delta, F^i, W^i, X^i)$

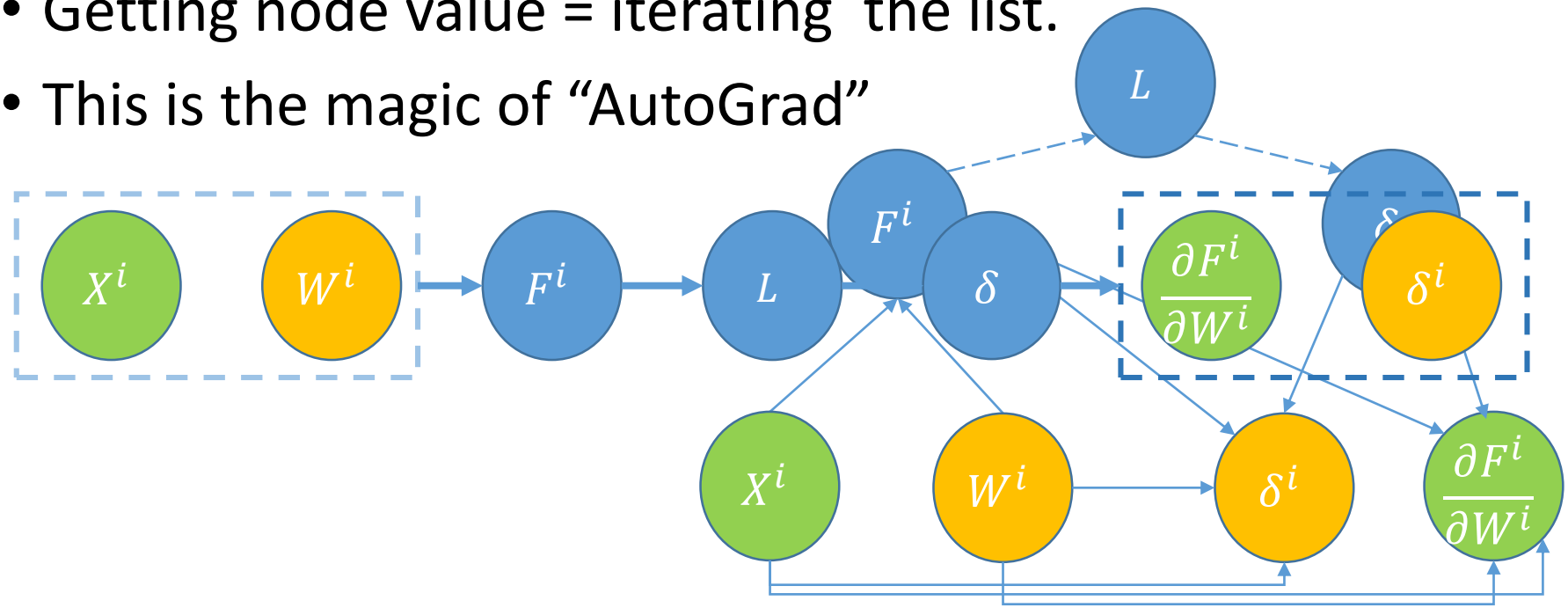


Rethink Chain Rule

- Dynamic Programming
  - $\frac{\partial L}{\partial X^i} = \mathrm{g}(\delta, F^i, W^i, X^i)$
  - Save all needed information to save computation
  - O(N) space (N – number of operations)
  - DP is topological sorting in the dependency graph.
- Use TS to run our networks

Topological Sort

- A nice video intro for it:
  - https://www.youtube.com/watch?v=ddTC4Zovtbc
- Translate a directional acyclic graph (DAG) with nodes representing the data to a linear list of nodes.
- Getting node value = iterating the list.
- This is the magic of “AutoGrad”



Application 1: Parallel Training

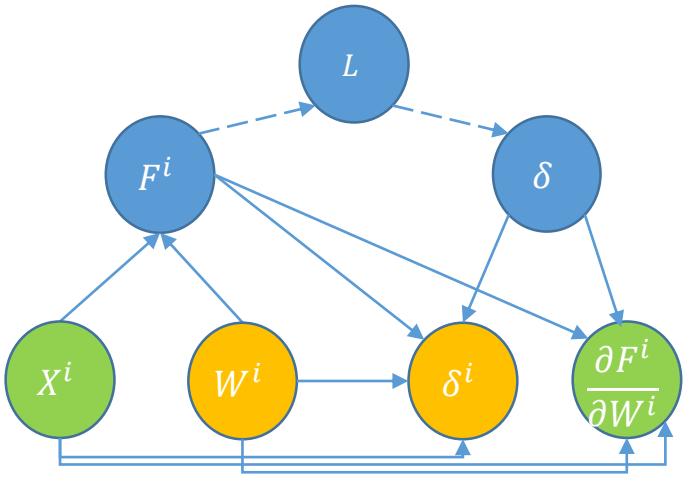
- We have multiple GPUs on a node, and multiple nodes.
- Recall SGD

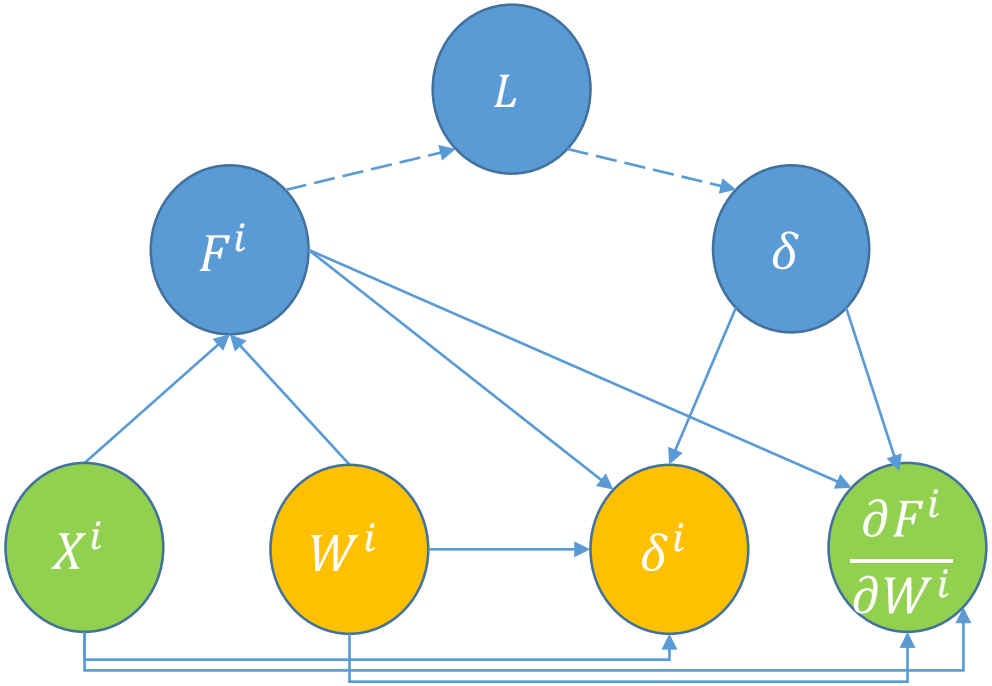
$$L(X, Y; W) = \frac{1}{m}\sum_i L(X_i, Y)\,, \qquad \frac{\partial L}{\partial W} = \frac{1}{m}\sum_i \frac{\partial L(X_i, Y)}{\partial W}$$

- No inter-sample dependency
  - Compute $\sum_i \frac{\partial L(X_i,Y)}{\partial W}$ in parallel using multiple devices
  - Data parallelism
  - Special Note: batch-normalization

Application 2: Memory Reduction

- Saving all δ, F^i, W^i, X^i for reducing computation is memory consuming
- No need to save them all the time
- Remove in-degrees after the node is computed
- Memory Management





I/O: Beyond Running Networks

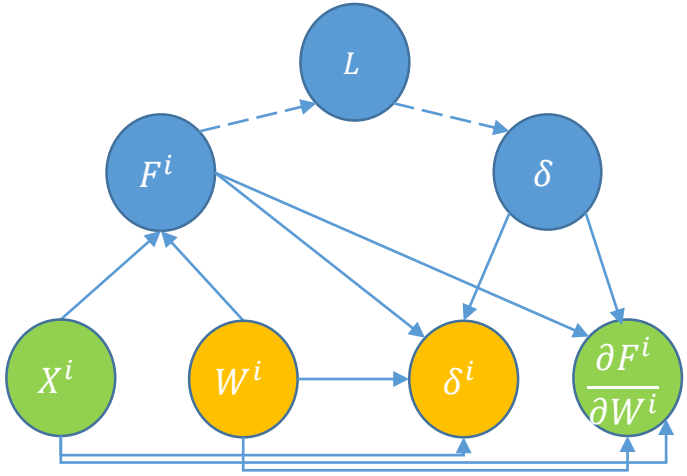
- Data, the food of deep learning
  - Read Data
  - Preprocessing
- A potential performance bottleneck
  - I/O contention (parallel training)
  - Preprocessing power
- Some basic principles
  - One concurrent reader is better than more (especially for HDD)
  - Use all our CPU powers for preprocessing – keep up with GPUs

DL Frameworks

- The battle of DL frameworks: why & what does it mean?
  - Frameworks is not only a tool
  - It affects how you deal with problems
  - Lock-in effect
- Categorize by goal (warning: personal opinion):
  - Research oriented: CUDA-ConvNet, Caffe, Torch, PyTorch, Chianer, Theano
  - Production oriented: TensorFlow, Caffe2, MXNet
- Categorize by graph solving manner
  - Static graph: CUDA-ConvNet, Caffe, TensorFlow*, Theano
  - Dynamic graph: Torch, PyTorch, TensorFlow,

Performance vs. Flexibility

- Static graph v.s. dynamic graph
  - Dynamic: solve TS every time
  - Static: solve TS before runtime
- Graph optimization
  - Dynamic: JIT, Hotspot
  - Static: compile time optimization
- Prebuilt-library v.s. ad-hoc ops
  - Prebuilt: cuDNN, Caffe layers
  - ad hoc modules
    - Configurable behavior at run time
    - Debug
    - Performance



Parallel Training

- Asynchronized SGD
  - DistBelief
  - Many new platforms support this
- Synchronized SGD
  - Most used
  - Communication patterns:
    - Average gradients, independent update
    - Reduce gradients, centralized updates
  - Implementation
    - MPI
    - NCCL: fast inter GPU communication
      - https://github.com/NVIDIA/nccl
    - Other message passaging interface
      - gRPC, ZeroMQ,…

Training vs. Inference

- Training is time consuming
- So could be inference: large testing data
- Multi-GPU/node testing
- Auxiliary libraries
  - Built on DL frameworks
  - Provide functionalities for some applications
    - RCNN/Fast RCNN/FasterRCNN – Object Detection
      - Google their names
    - DeepLab – Semantic Segmentation
      - https://bitbucket.org/deeplab/deeplab-public/
    - OpenID - Person ReID
      - https://github.com/Cysu/open-reid
    - TSN – Human Action Recognition
      - https://github.com/yjxiong/temporal-segment-networks

Parrots

- From MMLAB@CUHK
- Previous concerns taken care of
  - I/O: separated reading & processing
  - Runtime: supports dynamic graph & dynamic memory reduction
  - Primitives: carefully optimized kernels on all devices
  - Parallel: fast intra and inter node communication

Finally, which one to choose?

theano

dmlc
mxnet

Finally, which one to choose?

	Caffe	Theano	Torch	Chainer	TensorFlow	MxNet	Parrots
memory efficiency	1	2	1	1	2	2	3
single-GPU speed	1	1	3	2	2	3	3
multi-GPU speed	1	0	0	2	2	3	3
Weak scaling	0	0	0	0	0	2	3
Strong scaling	0	0	0	0	0	2	3

0	1	2	3
no out-of-box support	poor	average	top

Q&A