Introduction of Convolution Neural Network

Yan Li

CCL@ILC

Yan3.li@intel.com

2015/7/20

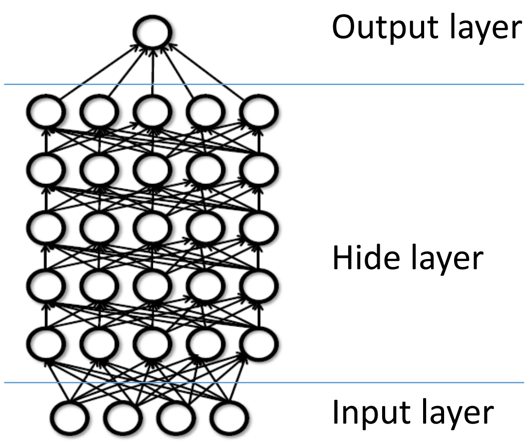
Deep Learning： Why so hot?

- Precondition： Big Data + HPC

  "Deep architectures are compositions of many layers of adaptive non-linear components, in other words, they are cascades of parameterized non-linear modules that contain trainable parameters at all levels". ---- Bengio & Yann LeCun

- Deep networks are powerful than shallow ones
  - A highly flexible way to specify prior knowledge
  - More efficiently when the number of training examples becomes larger
  - Handle large families of functions, parameterized with millions of individual parameters
- Different deep architectures
  - Convolutional Neural Networks(CNNs)
  - Deep Belief Networks(DBNs)
- Open source

软件	核心开发语言	其它支持语言	CPU 支持	GPU 支持
Kaldi	C++/CUDA		√	√
Cuda-convnet	C++/CUDA	Python		√
Caffe	C++/CUDA	Python, Matlab	√	√
Theano	Python		√	√
OverFeat	C++	Lua,Python	√	
Torch7	Lua		√	√

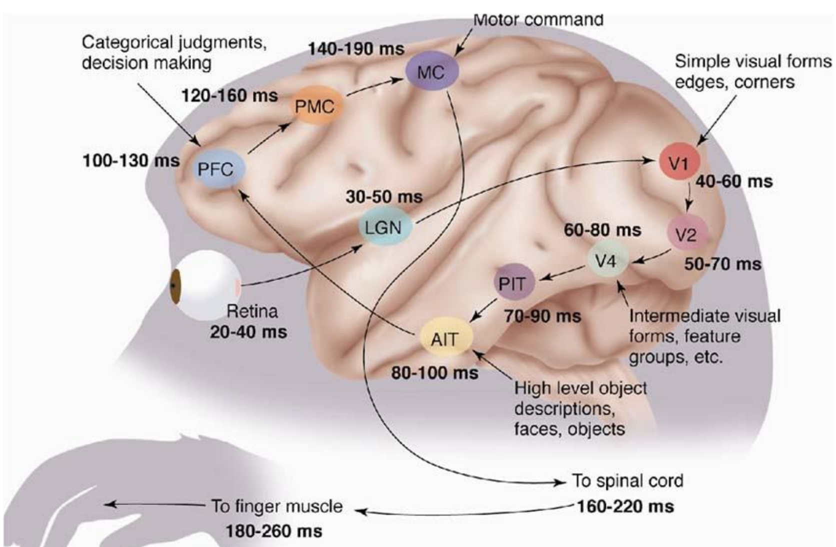


Deep Learning： Why so hot?

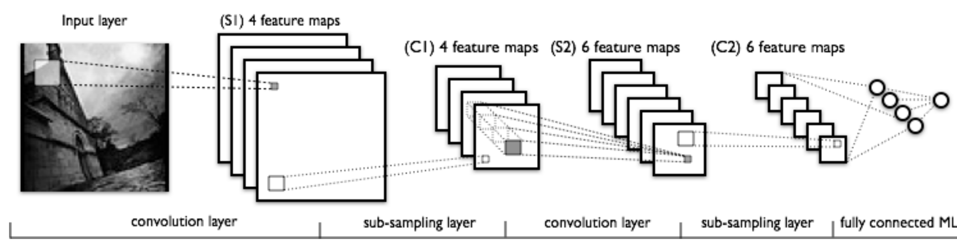
- Industry
  - Google: DistBelief
    - 1000 CPU nodes, each node has 16 cores
    - speech recognition & image recognition
  - Microsoft: Adam
    - 120 nodes, each node : dual Intel Xeon E5-2450L (16 cores)
  - Baidu: Minwa & Paddle
    - Speed & image recognition
    - Deep Image
  - Tencent: Mariana
    - Advertising in QQ and QQ Zone
    - Speed & image recognition in WeChat

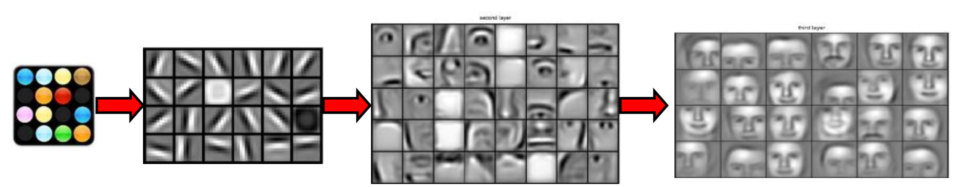
Deep Learning： Why so hot?

- Multiple layers work to build an improved feature space
- Highly varying functions can be efficiently represented with deep architectures
- Learning Representations / Features
    - Image recognition
        - Image -> low-level feature -> mid-level feature -> High-level feature -> Trainable Classifier
        - Pixel -> Edge -> texton -> motif -> part ->object
    - Text
        - Character->word->word group->clause->sentence->story
    - Speed
        - Sample->spectral band -> sound ->..->phone->phoneme->word



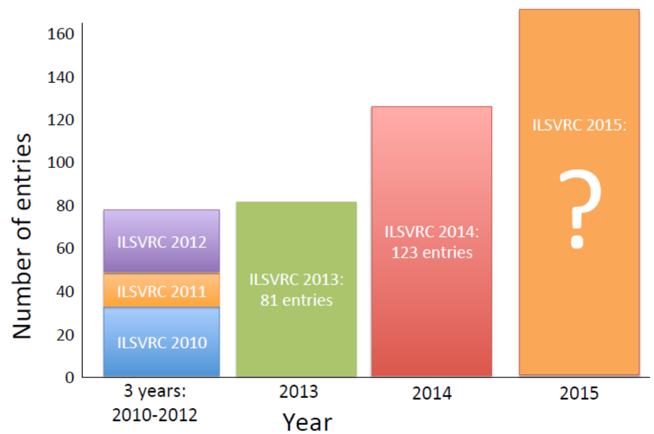
picture from Simon Thorpe





ImageNet Large Scale Visual Recognition Challenge

- A benchmark in object category classification and detection
  - http://image-net.org/challenges/LSVRC/2015/
- Participation in ILSVRC over years



ILSVRC overview: past, present, and future (http://image-net.org/tutorials/cvpr2015/)

Image classification annotations (1000 object classes)

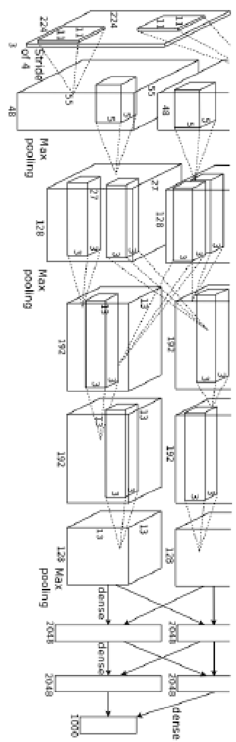
Year	Train images (per class)	Val images (per class)	Test images (per class)
ILSVRC2010	1,261,406 (668-3047)	50,000 (50)	150,000 (150)
ILSVRC2011	1,229,413 (384-1300)	50,000 (50)	100,000 (100)
ILSVRC2012-14	1,281,167 (732-1300)	50,000 (50)	100,000 (100)

- Image classification
  - Down to < 0.05 error (top-5 error) since ILSVRC2014
  - 2011: 0.26 -> 2012: 0.16
    - Deep learning method

ImageNet Large Scale Visual Recognition Challenge

Year 2012

SuperVision



8-layer

[Krizhevsky NIPS 2012]

Year 2014

GoogLeNet

22-layer

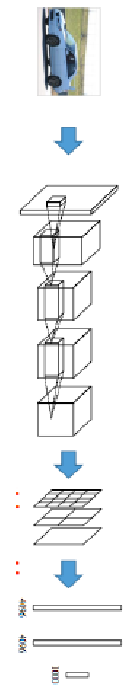
[Szegedy arxiv 2014]

VGG

16-layer

[Simonyan arxiv 2014]

MSRA



22-layer

[He arxiv 2014]

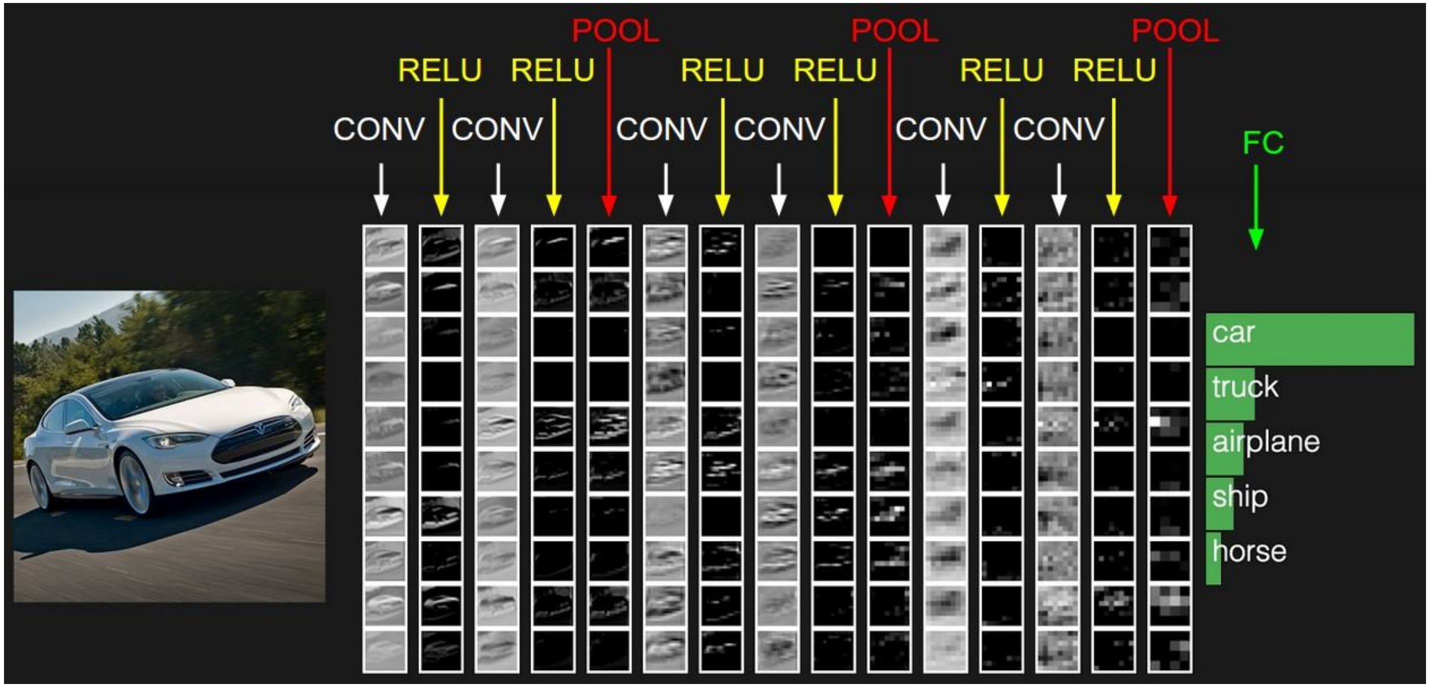
35/36 teams used deep learning

20/36 teams used open-source Caffe implementation

ILSVRC overview: past, present, and future (http://image-net.org/tutorials/cvpr2015/)

The CNN Architecture

- The activations of an example ConvNet architecture



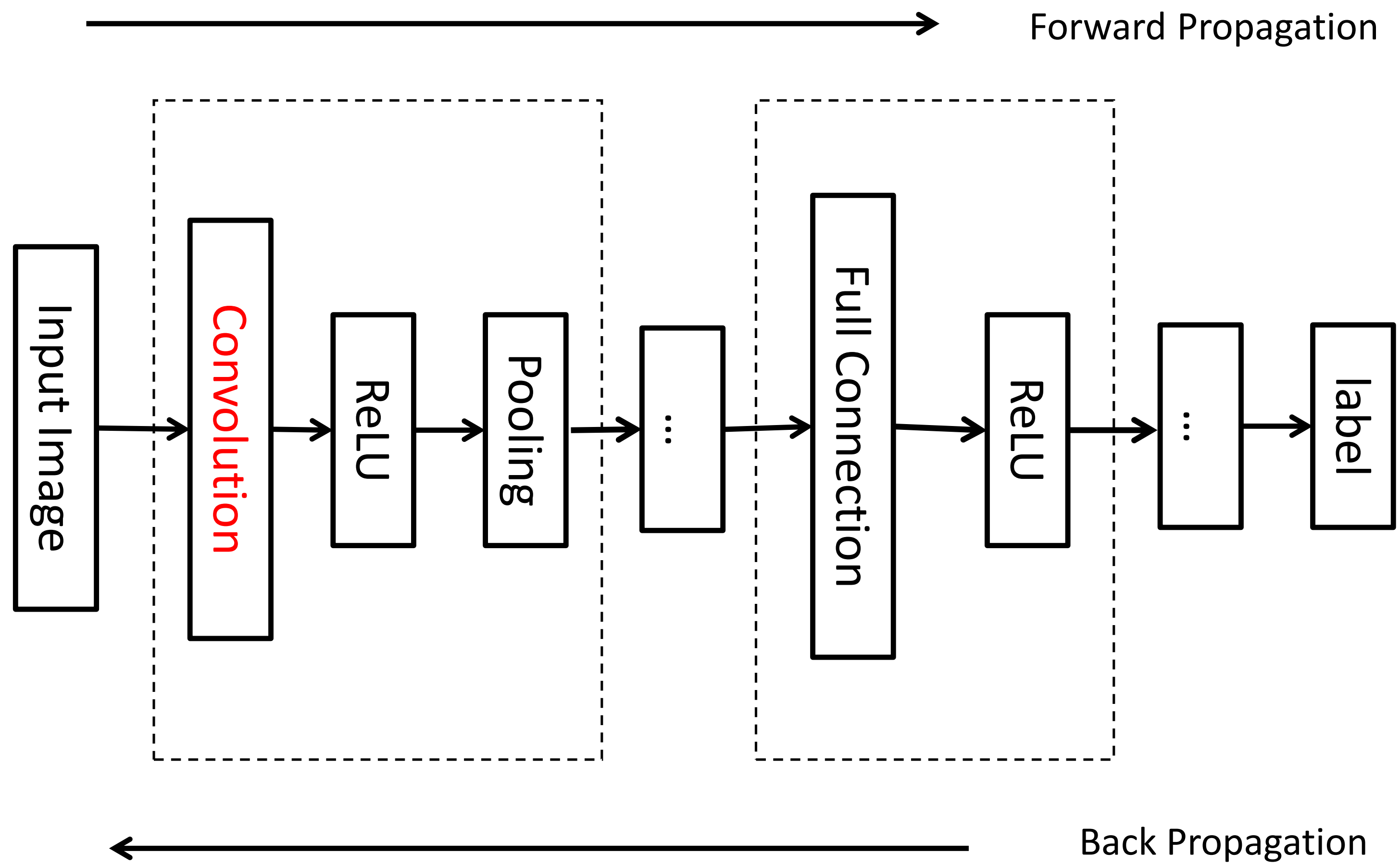
- Typical ConvNets
  - [CONV-RELU-POOL] x N, [FC-RELU] x M, Softmax or
  - [CONV-RELU-CONV-RELU-POOL] x N, [FC-RELU] x M, FC, SOFTMAX

  N >= 0, M >=0

Fei-Fei Li & Andrej Karpthy. Visualizing and Understanding Convolution Neural Networks.

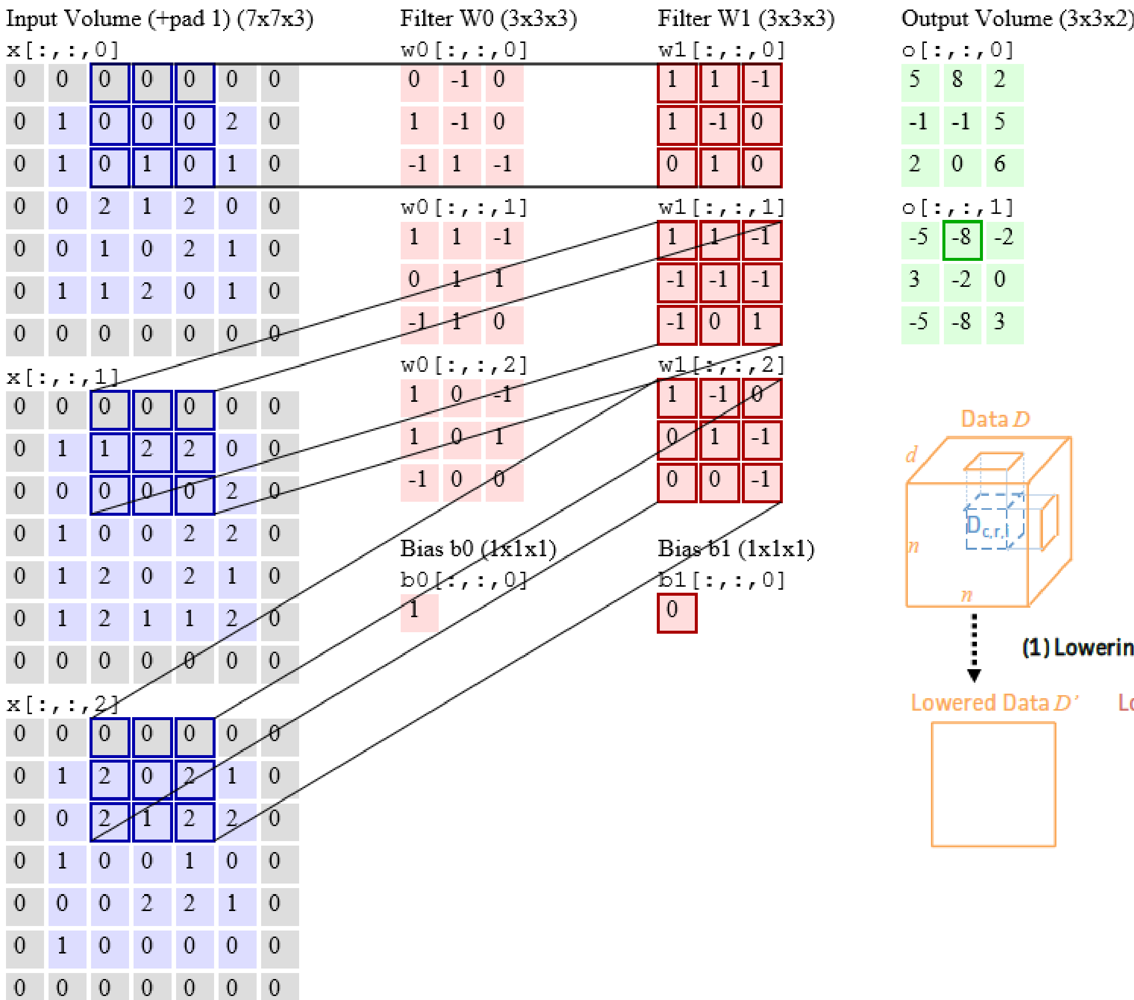
The CNN Architecture

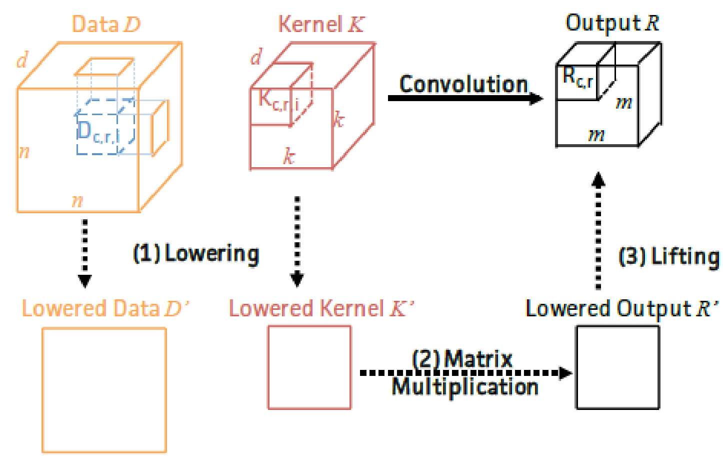
- The classical computation flow for ConvNet



The CNN Architecture

- The convolution layer

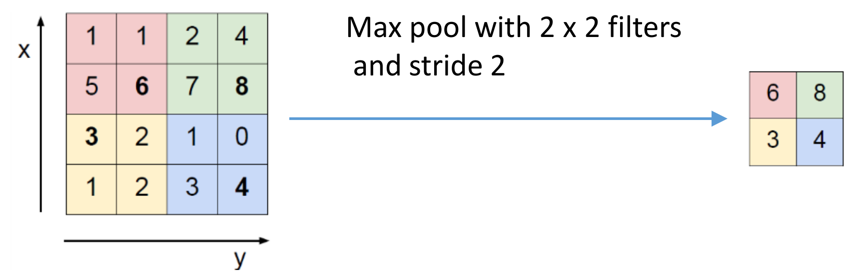




Convert to GEMM

The CNN Architecture

- CONV:
  - The Convolution layer
- RELU:
  - Non-linearity function
  - Rectifier function, such as $f(x) = \mathrm{Max}(0, x)$
- POOL
  - downsamples



The CNN Architecture

- An example ConvNet architecture (VGG: 16 weight layers)

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

Note:

Most memory is in early CONV

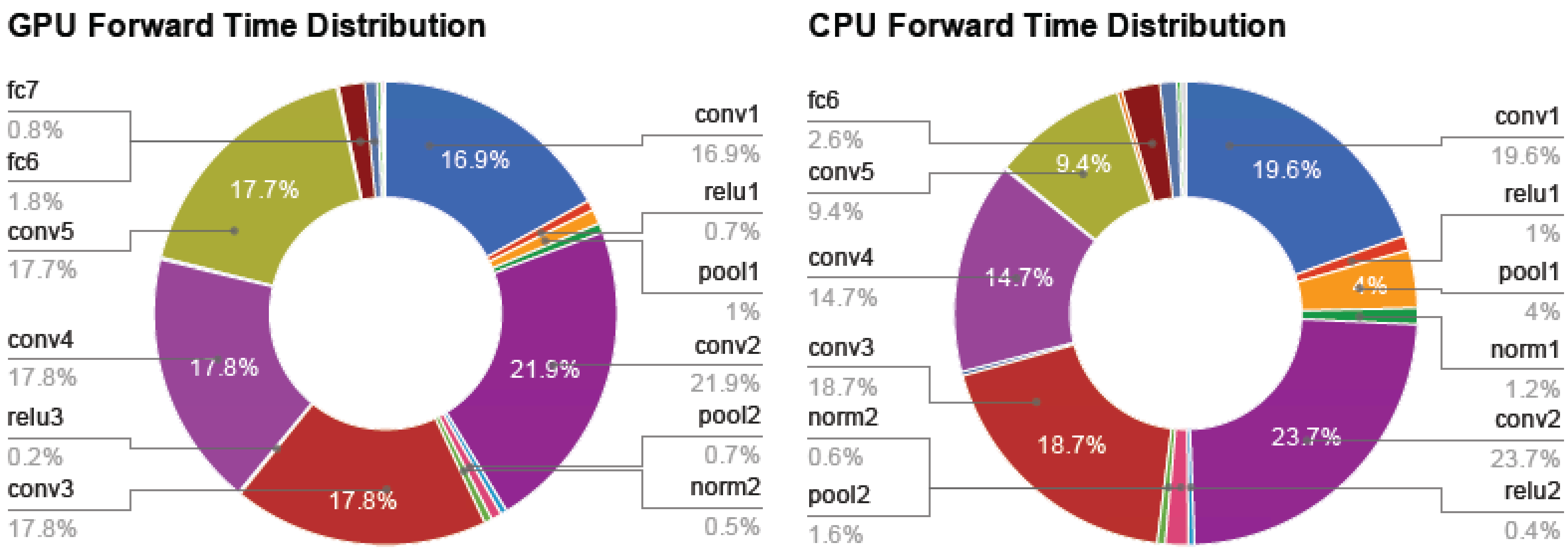
Most params are in late FC

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Fei-Fei Li & Andrej Karpathy. Visualizing and Understanding Convolution Neural Networks.

The CNN Architecture

- Computation time distribution of individual layers (Alex ConvNet)



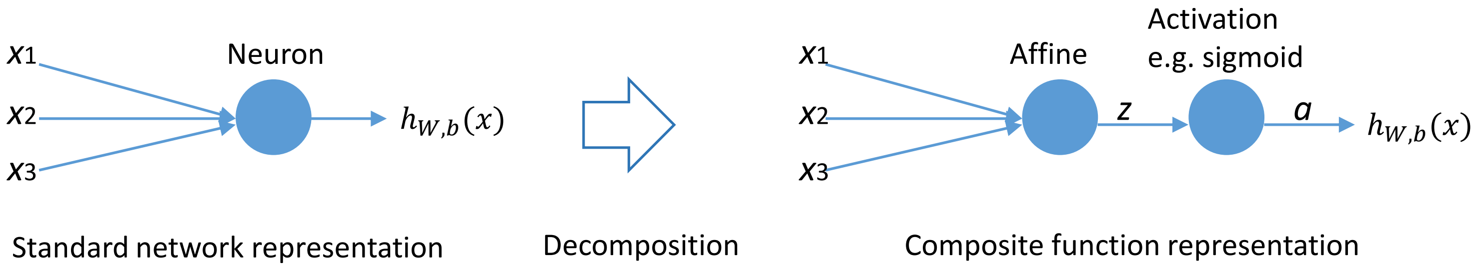
Yangqing Jia. "Learning Semantic Image Representations at a Large Scale"

Computation : $\text{Rate}_{\text{Conv}} > 80\%$

Communication : $\text{Rate}_{\text{FC}} > 80\%$

Backpropagation in CNN

- A neural network is decomposed into a composite function where each function element corresponds to a differentiable operation.
- Single neuron (the simplest neural network) example
  - A single neuron is decomposed into a composite function of an affine function element parameterized by ***W*** and ***b*** and an activation function element ***f*** which we choose to be the sigmoid function.
  - Derivative of both affine and sigmoid function elements w.r.t. both inputs and parameters are known. Note that sigmoid function doesn't have neither parameters nor derivatives parameters.
    - Sigmoid function is applied element-wise. "•" denotes Hadam and product, or element-wise product.



$$h_{W,b}(x) = f(W^T x + b) = sigmoid(affine_{W,b}(x)) = (\text{sigmoid} \circ \text{affine}_{W,b})(x)$$

$$\frac{\partial a}{\partial z} = a \bullet (1 - a) \text{ where } a = h_{W,b}(x) = \text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$$

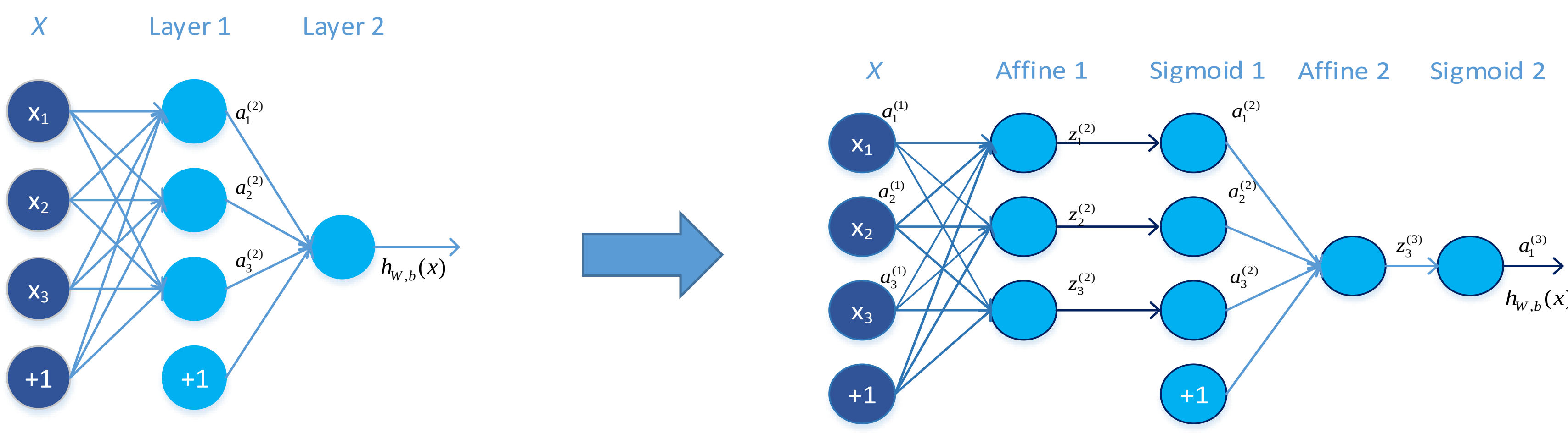
$$\frac{\partial z}{\partial x} = W, \ \frac{\partial z}{\partial W} = x, \ \frac{\partial z}{\partial b} = I \text{ where } z = \text{affine}_{W,b}(x) = W^T x + b$$

Backpropagation in CNN

- Error signals are defined as the derivative of any cost function *J* which we choose to be the square error. Error signals are computed (propagated backward) by the chain rule of derivative and useful for computing the gradient of the cost function.
- Single neuron example
  - Suppose we have m labeled training examples $\{(x^{(1)}, y^{(1)}),\ldots,(x^{(m)}, y^{(m)})\}$. Square error cost function for each example if as follows. Overall cost function is the summation of cost functions over all examples.
  $$J(W, b; x, y) = \frac{1}{2}||y - h_{W,b}(x)||^2$$
  - Error signals of the square error cost function for each example are propagated using derivatives of function elements w.r.t. input.
  $$\delta^{(a)} = \frac{\partial}{\partial a} J(W, b; x, y) = -(y - a)$$
  $$\delta^{(z)} = \frac{\partial}{\partial z} J(W, b; x, y) = \frac{\partial J}{\partial a}\frac{\partial a}{\partial z} = \delta^{(a)} \bullet a \bullet (1 - a)$$
  - Gradient of the cost function w.r.t parameters for each example is computed using error signals and derivatives of function elements w.r.t parameters. Summing gradients for all examples gets overall gradient.
  $$\nabla_W J(W, b; x, y) = \frac{\partial}{\partial w} J(W, b; x, y) = \frac{\partial J}{\partial z}\frac{\partial z}{\partial W} = \delta^{(z)} x^T$$
  $$\nabla_b J(W, b; x, y) = \frac{\partial}{\partial b} J(W, b; x, y) = \frac{\partial J}{\partial z}\frac{\partial z}{\partial b} = \delta^{(z)}$$

Backpropagation in CNN

- Composite function representation of a multi-layer neural network



Standard network representation   Decomposition   Composite function representation

$$h_{W,b}(x) = \left(\text{sigmoid} \circ \text{affine}_{W^{(2)},b^{(2)}} \circ \text{sigmoid} \circ \text{affine}_{W^{(1)},b^{(1)}}\right)(x)$$

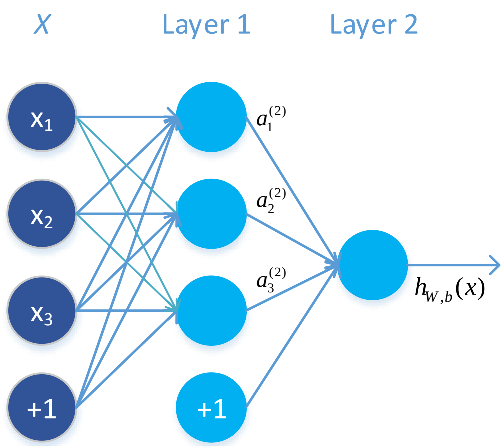
- Derivative of function elements w.r.t. inputs and parameters

$$a^{(1)} = x,\ a^{(l_{\max})} = h_{W,b}(x)$$

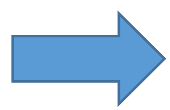
$$\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} = a^{(l+1)} \bullet \left(1 - a^{(l+1)}\right) \text{ where } a^{(l+1)} = \text{sigmoid}(z^{(l+1)}) = \frac{1}{1 + \exp(-z^{(l+1)})}$$

Backpropagation in CNN

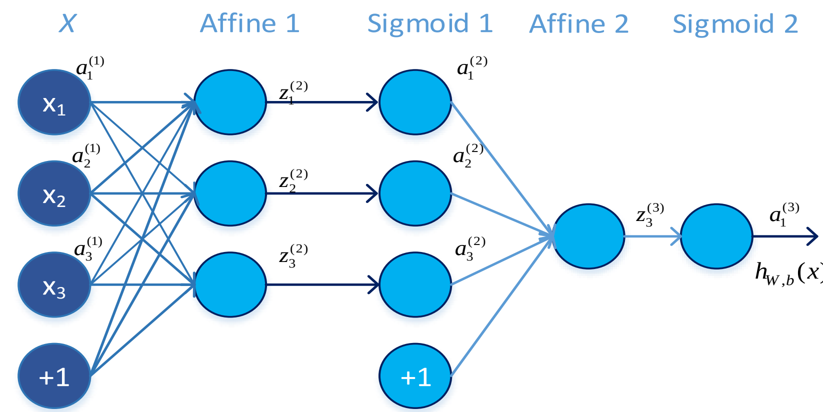
- Error signals of the square error cost function for each example



Standard network representation



Decomposition



Composite function representation

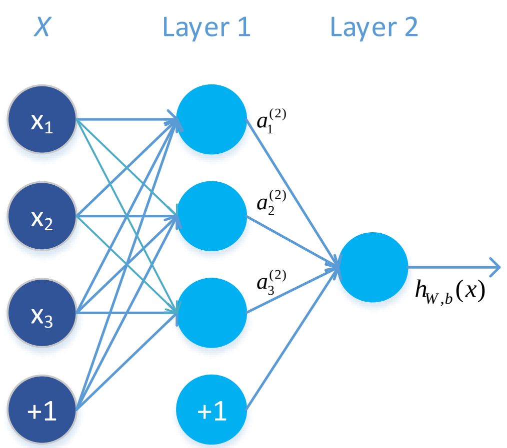
$$h_{W,b}(x) = \left(\text{sigmoid} \circ \text{affine}_{W^{(2)},b^{(2)}} \circ \text{sigmoid} \circ \text{affine}_{W^{(1)},b^{(1)}}\right)(x)$$

$$\delta^{(a^{(l)})} = \frac{\partial}{\partial a^{(l)}} J(W,b;x,y) = \begin{cases} -\left(y - a^{(l)}\right) & \text{for } \; l = l_{\max} \\ \dfrac{\partial J}{\partial z^{(l+1)}} \dfrac{\partial z^{(l+1)}}{\partial a^{(l)}} = \left(W^{(l)}\right)^T \delta^{(z^{(l+1)})} & \text{otherwise} \end{cases}$$

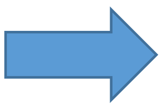
$$\delta^{(z^{(l)})} = \frac{\partial}{\partial z^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} = \delta^{(a^{(l)})} \bullet a^{(l)} \bullet (1 - a^{(l)})$$

Backpropagation in CNN

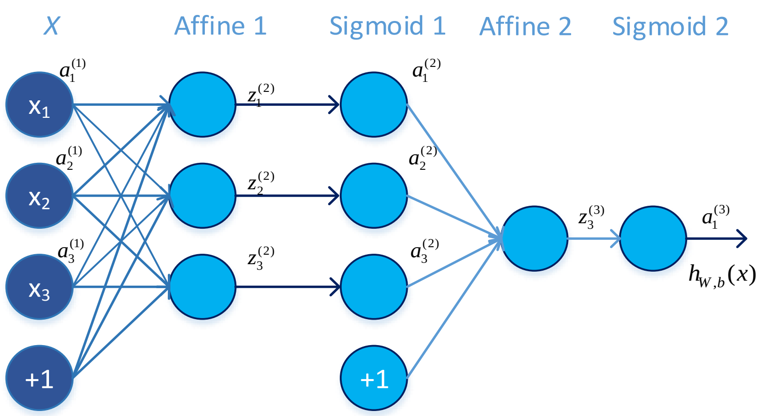
- Gradient of the cost function w.r.t. parameters for each example



Standard network representation



Decomposition



Composite function representation

$$h_{W,b}(x) = \left(\text{sigmoid} \circ \text{affine}_{W^{(2)},b^{(2)}} \circ \text{sigmoid} \circ \text{affine}_{W^{(1)},b^{(1)}}\right)(x)$$

$$\nabla_{W^{(l)}} J(W,b;x,y) = \frac{\partial}{\partial W^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial W^{(l)}} = \delta^{(z^{(l+1)})} \left(a^{(l)}\right)^T$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \frac{\partial}{\partial b^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial b^{(l)}} = \delta^{(z^{(l+1)})}$$

Derivative in Convolutional Layer

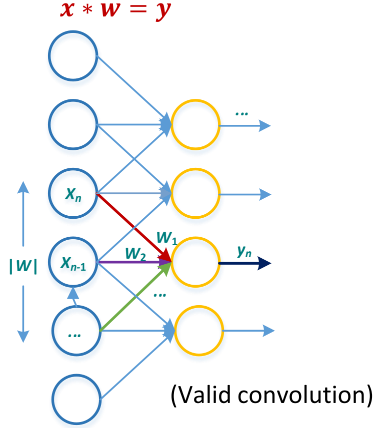
- Error signals and gradient for each example are computed by convolution using the commutativity property of convolution and the multivariable chain rule of derivative.
  - Let's focus on single elements of error signals and a gradient w.r.t. *w*.

$$\delta_n^{(x)} = \frac{\partial J}{\partial x_n} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_n} = \sum_{i=1}^{|w|} \frac{\partial J}{\partial y_{n-i+1}}\frac{\partial y_{n-i+1}}{\partial x_n} = \sum_{i=1}^{|w|} \delta_{n-i+1}^{(y)} w_i = \left(\delta^{(y)} * \mathrm{flip}(w)\right)[n], \delta^{(x)} = [\delta_n^{(x)}] = \delta^{(y)} * \mathrm{flip}(w)$$

Reverse order linear combination

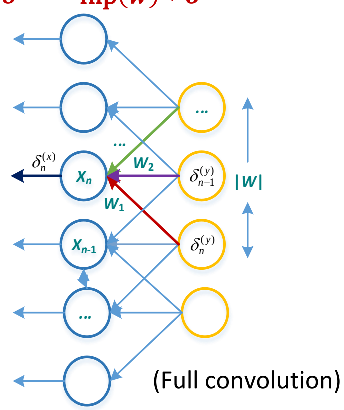
$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w_i} = \sum_{n=1}^{|x|-|w|+1} \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_i} = \sum_{n=1}^{|x|-|w|+1} \delta_n^{(y)} x_{n+i-1} = \left(\delta^{(y)} * x\right)[i], \frac{\partial J}{\partial w} = \left[\frac{\partial y}{\partial w_i}\right] = \delta^{(y)} * x = x * \delta^{(y)}$$

$\boldsymbol{x * w = y}$



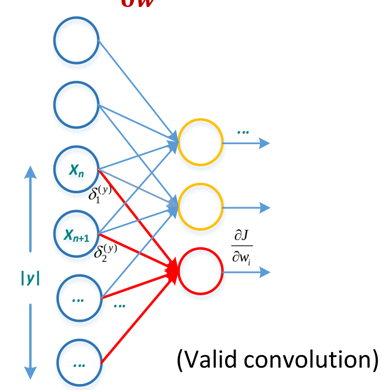
Forward propagation (convolution)

$\boldsymbol{\delta^{(x)} = \mathrm{flip}(w) * \delta^{(y)}}$



backward propagation

$\boldsymbol{x * \delta^{(y)} = \frac{\partial J}{\partial w}}$



Gradient computation

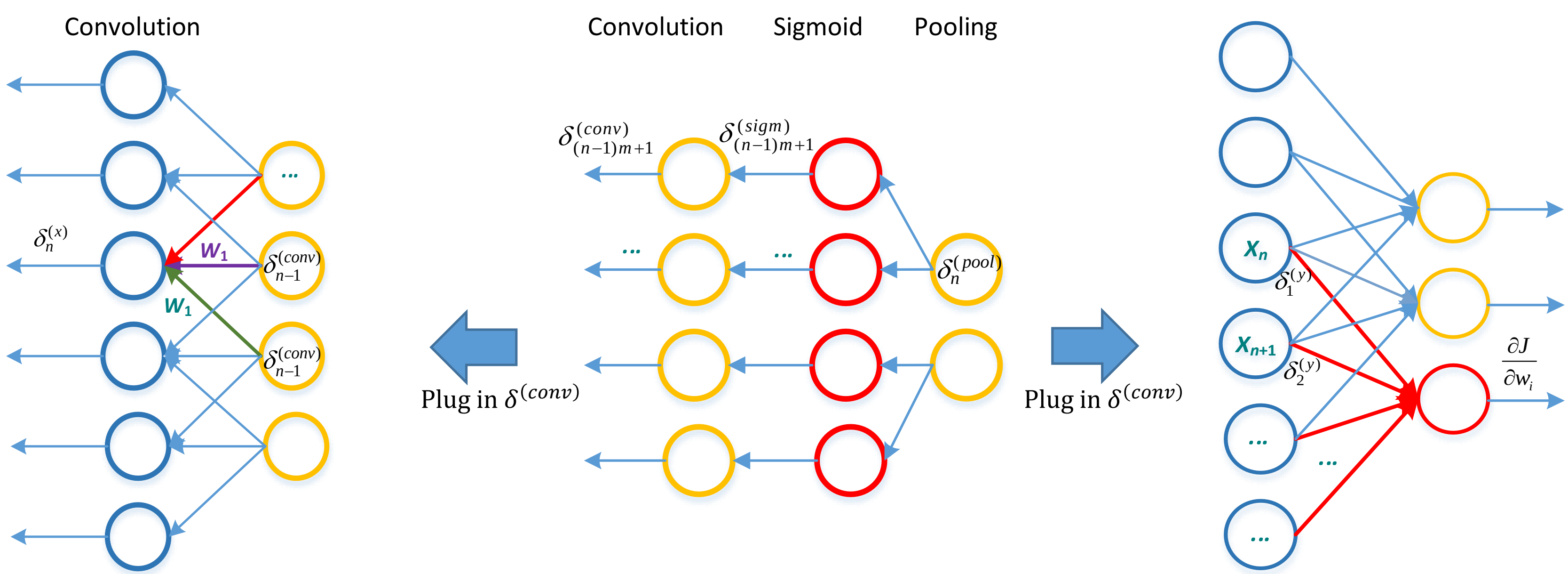
Convolutional Neural Network

- Summary of Backpropagation

2. Propagate error signals $\delta^{(conv)}$

1. Propagate error signals $\delta^{(\text{pool})}$

3. Compute gradient $\nabla_w J$



$\delta^{(x)} = \delta^{(conv)} * \text{flip}(w)$

$\delta^{(conv)} = upsample\left(\delta^{(conv)}, g'\right) \bullet \boxed{f^{(sigm)} \bullet \left(1 - f^{(sigm)}\right)}$

Derivative of sigmoid

$x * \delta^{(conv)} = \nabla_w J$

Code Review: Full-connected Layer

- LayerSetUp()
  - Parameter initialization
  - For one-time initialization: reading parameters, fixed-size allocations, etc.
- Reshape()
  - For computing the sizes of top blobs, allocating buffers, and any other work that depends on the shapes of bottom blobs
- Differentiate
  - Reshape is called before every forward pass;
  - LayerSetUp is only called once at initialization. This allows networks to change their blob shapes while running.
  - https://github.com/BVLC/caffe/issues/1385

256

Code Review: Full-connected Layer

- GEMM: $C = \alpha A \times B + \beta C$
  - A: *M* x *K*
  - B: *K* x *N*
  - C: *M* x *N*

```
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
    const int M, const int N, const int K, const float alpha,
    const float* A, const float* B, const float beta, float* C)
```

- GEMV: $Y = \alpha AX + \beta Y$
  - A: *M* x *N*
  - X: *N* x 1
  - Y: *M* x 1

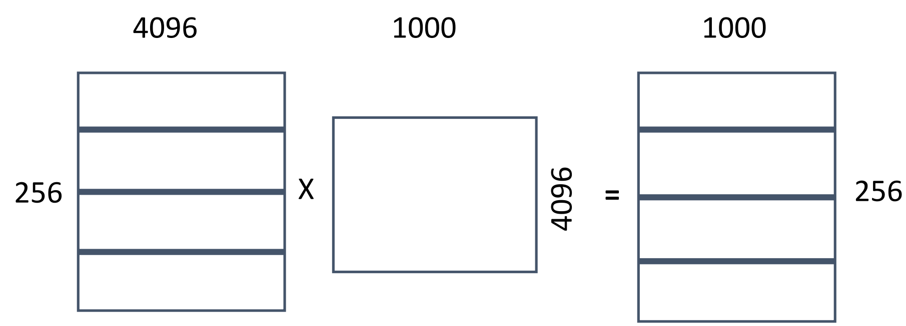
```
void caffe_cpu_gemv<float>(const CBLAS_TRANSPOSE TransA, const int M,
    const int N, const float alpha, const float* A, const float* x,
    const float beta, float* y)
```

Code Review: Full-connected Layer

- Forward Computation

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const Dtype* weight = this->blobs_[0]->cpu_data();
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasTrans, M_, N_, K_, (Dtype)1.,
      bottom_data, weight, (Dtype)0., top_data);          y=wx or y=xw'
  if (bias_term_) {
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, N_, 1, (Dtype)1.,
        bias_multiplier_.cpu_data(),     y=y+b
        this->blobs_[1]->cpu_data(), (Dtype)1., top_data);
  }
}
```

- Function: y = wx + b
  - M_: the number of images
  - K_: the number of features per image
  - N_: the number of output neurons
    - X: M x K   Y: N x 1
    - W: N x K   b: N x 1

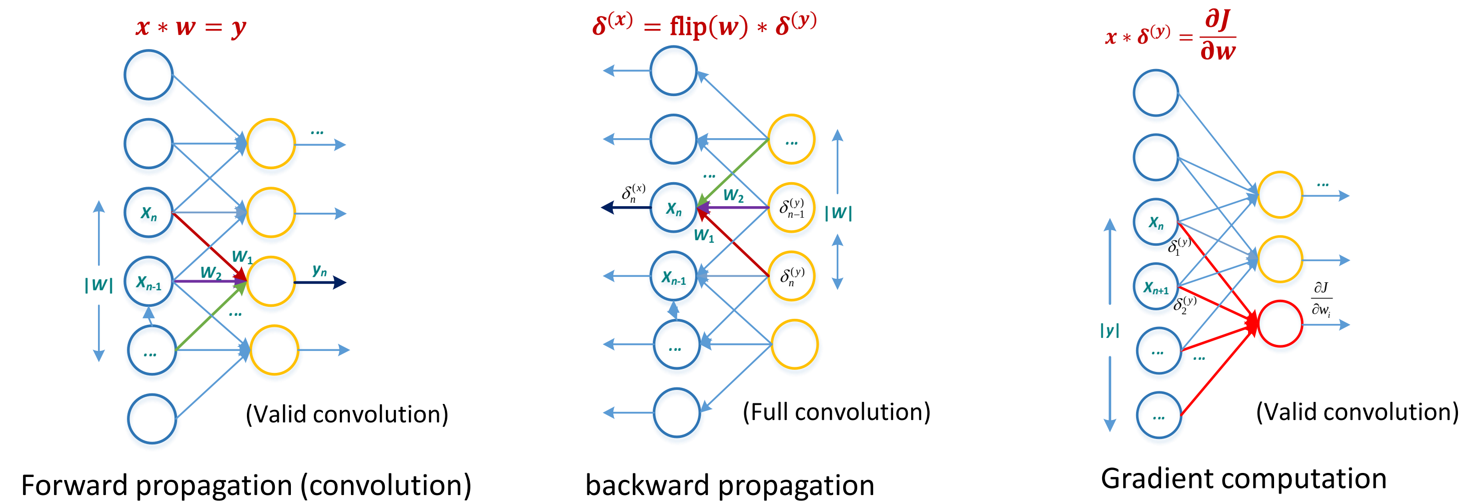


Code Review: Full-connected Layer

- Backward Computation

$$\nabla_W J(W,b;x,y) = \frac{\partial}{\partial w} J(W,b;x,y) = \frac{\partial J}{\partial z}\frac{\partial z}{\partial W} = \delta^{(z)} x^T$$

$$\nabla_b J(W,b;x,y) = \frac{\partial}{\partial b} J(W,b;x,y) = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta^{(y)}$$

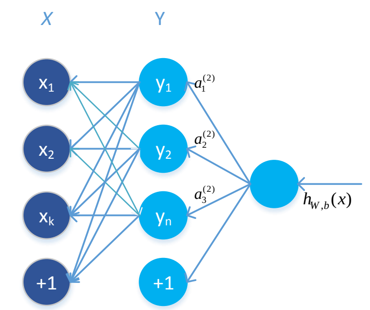


$$\delta_i^{(x)} = \frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_i} = \sum_{j=1}^{|w|} \frac{\partial J}{\partial y_{i-j+1}} \frac{\partial y_{i-j+1}}{\partial x_n} = \sum_{j=1}^{|w|} \delta_{i-j+1}^{(y)} w_i = \left(\delta^{(y)} * \text{flip}(w)\right)[n], \delta^{(x)} = [\delta_n^{(x)}] = \delta^{(y)} * \text{flip}(w)$$

Reverse order linear combination

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w_{ij}} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w_{ij}} = \delta_j^{(y)} * x_i \qquad \frac{\partial J}{\partial b_i} = \delta_j^{(y)}$$

Code Review: Full-connected Layer



- Backward Computation

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  if (this->param_propagate_down_[0]) {
    const Dtype* top_diff = top[0]->cpu_diff();
    const Dtype* bottom_data = bottom[0]->cpu_data();
    // Gradient with respect to weight
    caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans, N_, K_, M_, (Dtype)1.,
        top_diff, bottom_data, (Dtype)0., this->blobs_[0]->mutable_cpu_diff());
  }
  if (bias_term_ && this->param_propagate_down_[1]) {
    const Dtype* top_diff = top[0]->cpu_diff();
    // Gradient with respect to bias
    caffe_cpu_gemv<Dtype>(CblasTrans, M_, N_, (Dtype)1., top_diff,
        bias_multiplier_.cpu_data(), (Dtype)0.,
        this->blobs_[1]->mutable_cpu_diff());
  }
  if (propagate_down[0]) {
    const Dtype* top_diff = top[0]->cpu_diff();
    // Gradient with respect to bottom data
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, K_, N_, (Dtype)1.,
        top_diff, this->blobs_[0]->cpu_data(), (Dtype)0.,
        bottom[0]->mutable_cpu_diff());
  }
}
```

$$\nabla_W J(W,b;x,y) = \frac{\partial}{\partial w} J(W,b;x,y) = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \delta^{(y)} x^T$$

$$\nabla_b J(W,b;x,y) = \frac{\partial}{\partial b} J(W,b;x,y) = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta^{(y)}$$

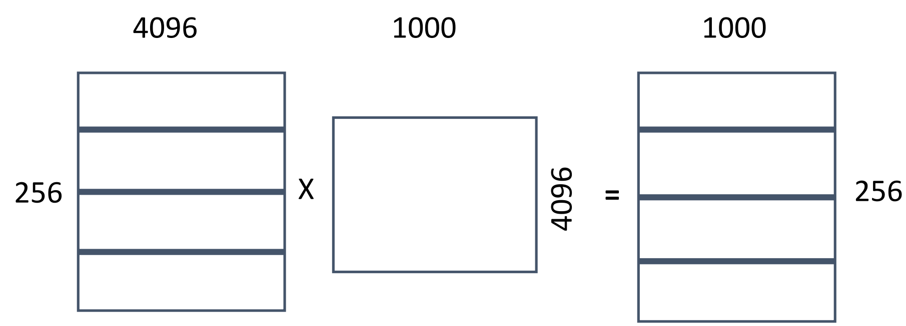
$$\delta_i^{(x)} = \left(\sum_{j=1}^{|y|} \delta_{i-j+1}^{(y)} w_i\right)\cancel{f'(z_i^{(x)})}$$

Code Review: Convolution Layer

- Forward Computation

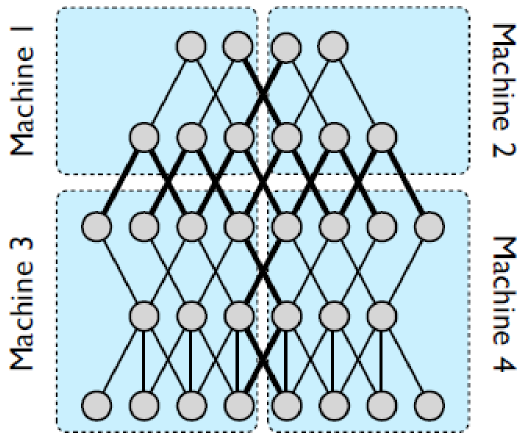
```
template <typename Dtype>
void InnerProductLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const Dtype* weight = this->blobs_[0]->cpu_data();
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasTrans, M_, N_, K_, (Dtype)1.,
      bottom_data, weight, (Dtype)0., top_data);          y=wx or y=xw'
  if (bias_term_) {
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, N_, 1, (Dtype)1.,
        bias_multiplier_.cpu_data(),     y=y+b
        this->blobs_[1]->cpu_data(), (Dtype)1., top_data);
  }
}
```

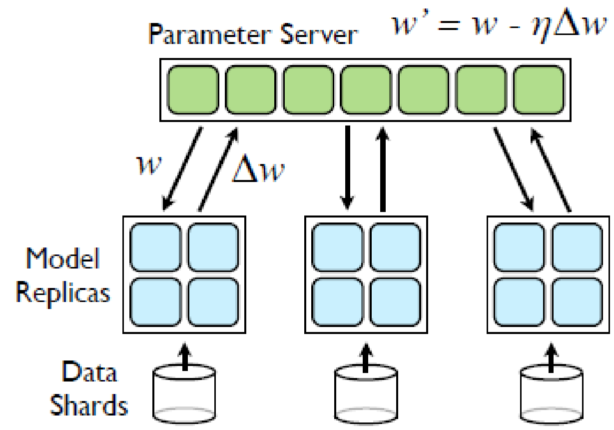
- Function: $y = wx + b$
  - M_: the number of images
  - K_: the number of features per image
  - N_: the number of output neurons
    - X: M x K   Y: N x 1
    - W: N x K   b: N x 1



Distributed Deep Learning Networks

- Model Parallelism

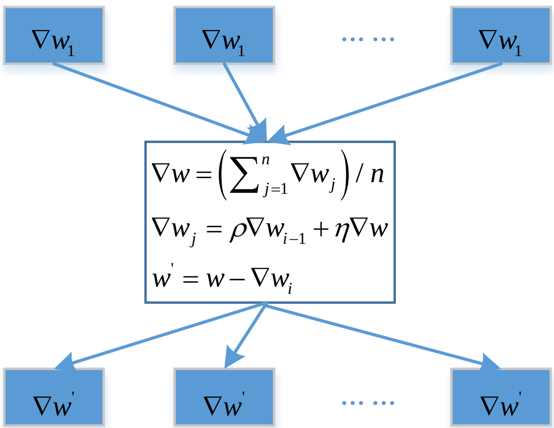


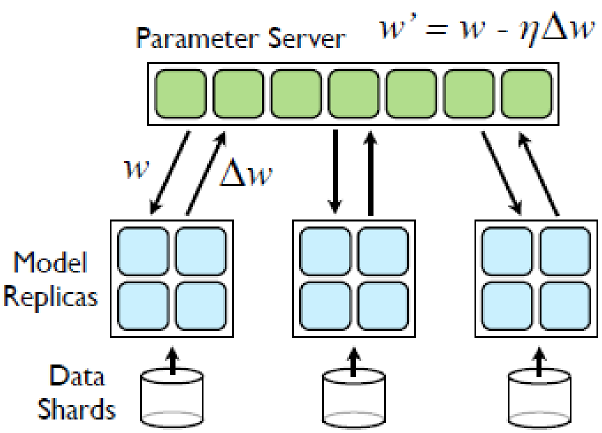


- Data Parallelism
  - Based on mini-batch
  - A single parameter server

Our Previous Work

- Data Parallelism





- Data Parallelism
  - Based on mini-batch
  - A single parameter server
- MPI + OpenMP
  - MKL BlAS (GEMM & GEMV)

Distributed implementation

```
for i = 0; i < out_iter; i++ do
    loss=net→ForwardBackward();          // 前向反向，获得 ∇w 和 loss
    ComputeUpdateValue();                 // 计算增量，Δw_i = ρΔw_{i−1} + η∇w
    net→Update();                         // 更新参数，Δw′ = w − Δw_i
```

算法 1.1: Caffe 的串行训练过程伪代码

```
for i = 0; i < out_iter; i++ do
    for j = 1 to n do in parallel
        loss_j=net→ForwardBackward();     // 前向反向，获得 ∇w_j 和 loss_j
    GlobalSyncDiff();                     // 全局归约 ∇w 并求平均
    GlobalSyncLoss();                     // 全局归约 loss 并求平均
    if j == root then
        ComputeUpdateValue();             // 计算增量，Δw_i = ρΔw_{i−1} + η∇w
        net→Update();                     // 更新参数，Δw′ = w − Δw_i
    GlobalSyncData();                     // 广播新模型的参数 w′
────────────────────────────────────────
void GlobalSyncDiff()                     // length 是 w 的维度
    MPI_Reduce(∇w_j, ∇w, length, MPI_FLOAT, MPI_SUM, root, COMM);
    if j == root then
        cblas_sscal(1/n, ∇w);             // 对 ∇w 求平均
void GlobalSyncLoss()
    MPI_Reduce(loss_j, loss, 1, MPI_FLOAT, MPI_SUM, root, COMM);
    if j == root then
        loss = loss/n;
void GlobalSyncData()
    MPI_Bcast(w′, length, MPI_FLOAT, root, COMM);
```

算法 1.2: Caffe 的分布式训练过程伪代码

Partial Synchronization

- 采用部分同步的非阻塞方式实现数据
  - 迭代内：保持不变
  - 迭代间：主节点只接收前s(s < n)个返回的梯度，更新为最新的模型 w'和b'后(更新公式不变)，也只广播给这s个节点
  - 缺点：如果存在慢节点，它每次都不能在前 s(s < n) 个返回梯度，那么这个节点将始终无法获得最新的模型参数

- 采用有界延迟策略保证训练过程的一致性
  - 每 t 次迭代就强制所有的节点同步更新参数

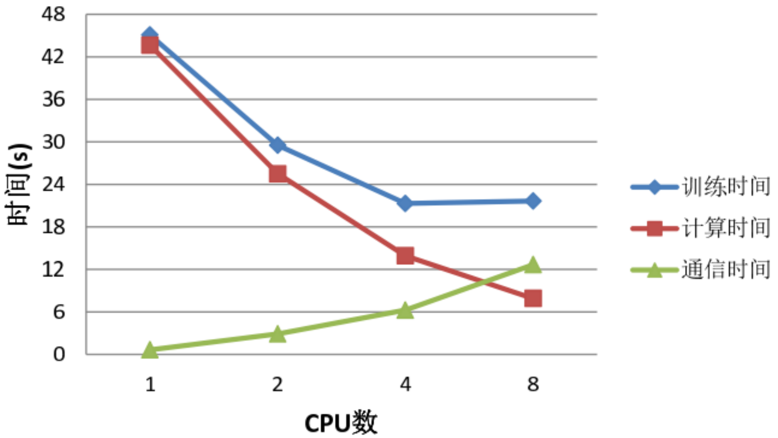
```
for i = 0; i < out_iter; i + + do
    for j = 1 to n do in parallel
        loss_j=net→ForwardBackward();                // 同算法1.2的第 3 行
    GlobalAsyncDiff();     // 对前 s 个返回梯度的节点，进行部分全局归约
    GlobalSyncLoss();                     // 计算 loss 值，同算法1.2的第 5 行
    if j == root then
        ComputeUpdateValue();             // 计算增量，同算法1.2的第 7 行
        net→Update();                     // 更新参数，同算法1.2 的第 8 行
    GlobalAsyncData();              // 将新模型的参数 w' 广播给这 s 个节点
```

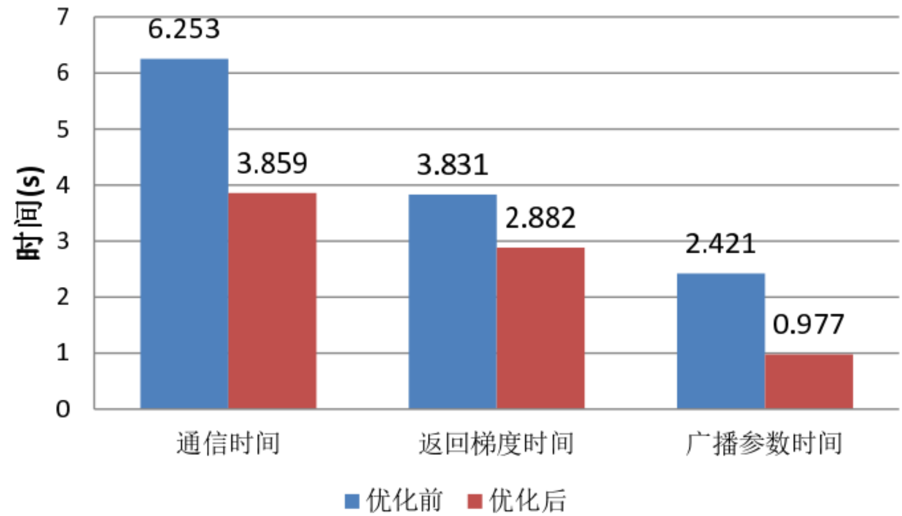
Evaluation

- 实验环境
  - 4 nodes, each node has dual Intel Xeon E5-2697 v3 (14 x 2 cores)
    - Infiniband QDR， Intel MKL 11.0.5.192 & MPI 4.1.1.036
  - Nvidia GeForce GTX Titan
    - 2688 CUDA cores， 6GB global memory, cuBLAS
- 测试案例
  - Caffe example
    - 模型：包含6 千万个参数的卷积神经网络经典模型AlexNet
    - 数据集：ImageNet数据库中，127 万张训练图片和 5 万张验证图片
    - Batch-size=256 (训练过程中每次迭代处理的图片数)
      Mini-Batch=256/n (每个处理器完成的图片数，n是处理器数)

Evaluation

- 5120 images





n=4 s=3 **38.3%**

- communication size: 233MB/node
  - Model size: 233MB

硬件	运算能力 (Tflops)	数学库	训练时间 (s)
2×E5-2697 v3	2×1.164≈2.32	MKL	29.5
4×E5-2697 v3	4×1.164≈4.65	MKL	21.3/18.9*
1×GTX770	3.2	cuBLAS cuDNN	33 24.3
1×K20	3.52	cuBLAS	36
1×K40	4.29	cuBLAS cuDNN	26.5 19.2
1×Titan	4.5	cuBLAS cuDNN	26.26 20.25

Summary

- CNN architecture
- Our previous work

Next Plan

- Many-core Parallelism within a single node
  - Hybrid data parallelism and model parallelism
- Heterogeneous computing
  - Intel CPU + Integrated GPU / FPGA
- How to overlap communication with computation in the distributed context
- Model resize
  - matrix decomposition or approximate matrix