

Fundamentals of Media Processing

Lecturer:

池畑 諭 (Prof. IKEHATA Satoshi)

児玉 和也 (Prof. KODAMA
Kazuya)

Support:

佐藤 真一 (Prof. SATO Shinichi)

孟 洋 (Prof. MO Hiroshi)

Course Overview (15 classes in total)

1-10 Machine Learning by Prof. Satoshi Ikehata

11-15 Signal Processing by Prof. Kazuya Kodama

Grading will be based on the final report.

10/16 (Today) Introduction Chap. 1

Basic of Machine Learning (Maybe for beginners)

10/23 Basic mathematics (1) (Linear algebra, probability, numerical computation) Chap. 2,3,4

10/30 Basic mathematics (2) (Linear algebra, probability, numerical computation) Chap. 2,3,4

11/6 Machine Learning Basics (1) Chap. 5

11/13 Machine Learning Basics (2) Chap. 5

Basic of Deep Learning

11/20 Deep Feedforward Networks Chap. 6

11/27 Regularization and Deep Learning Chap. 7

12/4 Optimization for Training Deep Models Chap. 8

CNN and its Application

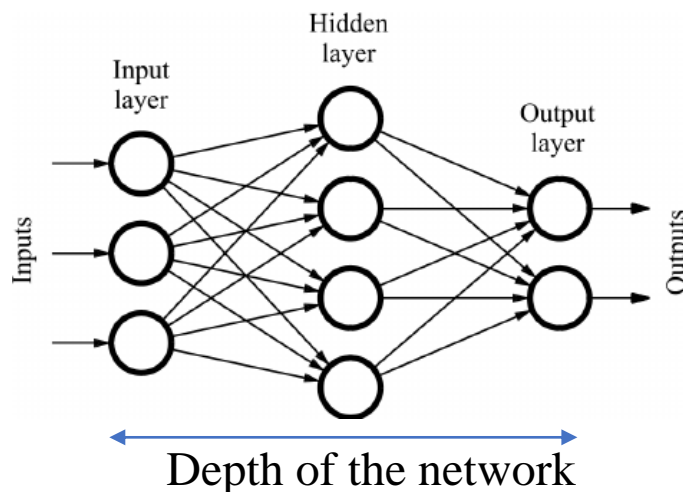
12/11 Convolutional Neural Networks and Its Application (1) Chap. 9 and more

12/18 Convolutional Neural Networks and Its Application (2) Chap. 9 and more

Deep Feedforward Networks

Deep Feedforward Networks

- Deep feedforward network (or multilayer perceptron; MLP) approximates a function $y = f^*(x)$ by $f(x; \theta)$ and learns θ
- Information flows through the function being evaluated from x , through the intermediate computation used to define f and finally to the output y . There are no feedbacks
- A simple linear model cannot learn. To get the nonlinearity of the function, we introduce the nonlinear transformation of x as $y = f(x; \theta, \omega) = \phi(x; \theta)^T \omega$: The deep neural networks try to learn ϕ



Learning XOR Network (1)

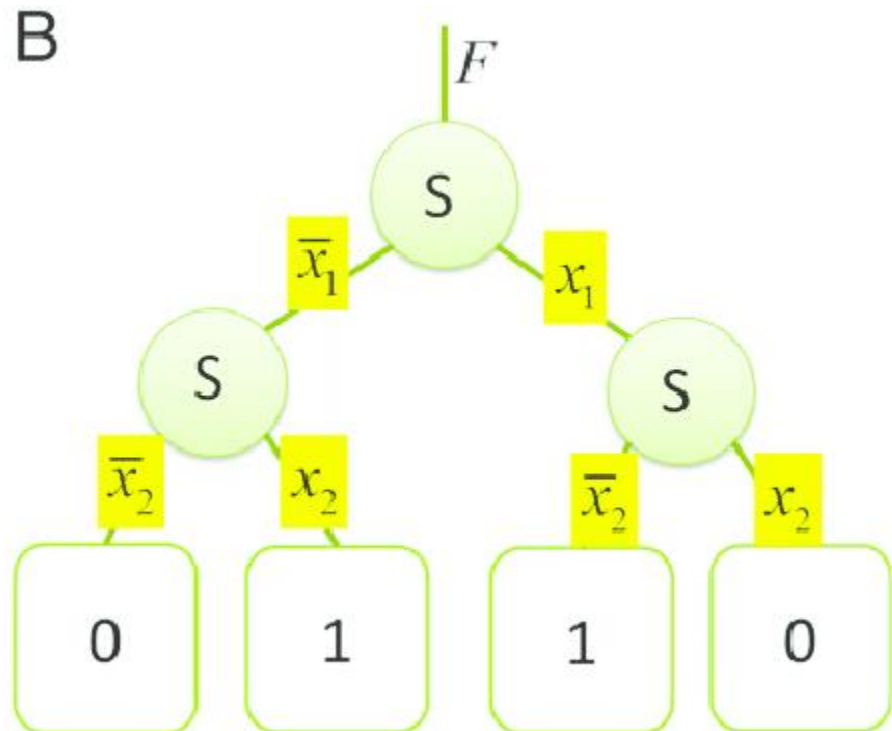
- Training from pairs of x_1, x_2 ($[0,0]$, $[0,1]$, $[1,0]$, $[1,1]$), learn the function that returns $x_1 \oplus x_2$ by minimizing

$$J(\theta) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(x) - f(x; \theta))^2$$

A

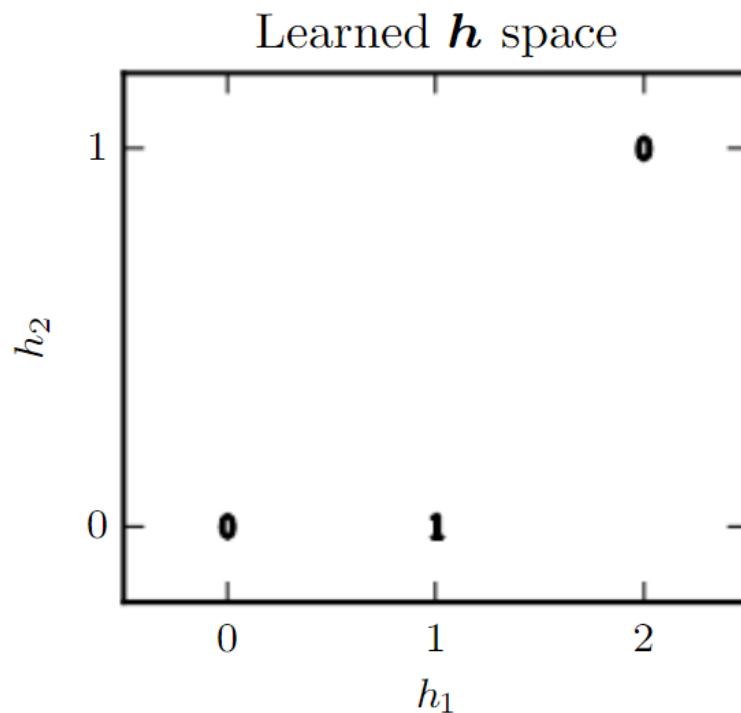
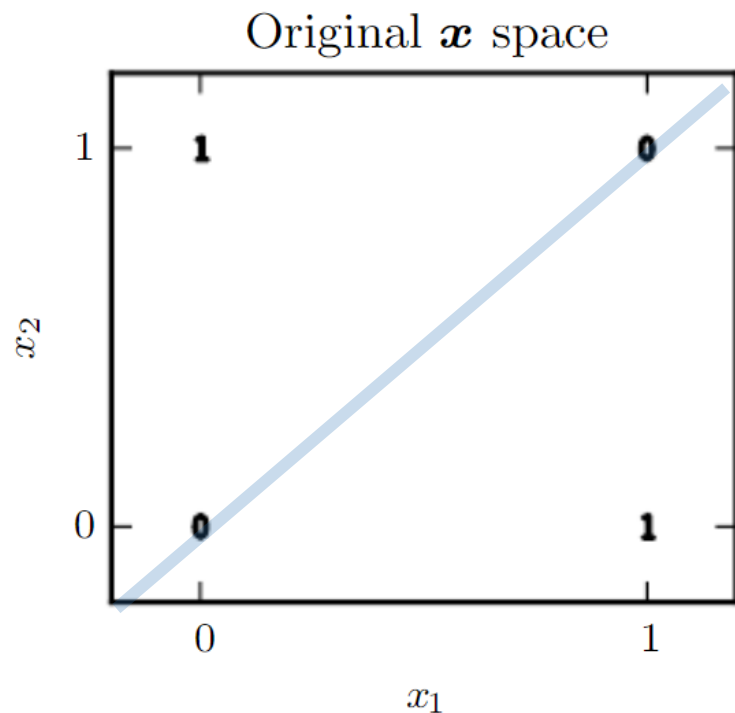
$$F(x_1, x_2) = x_1 \oplus x_2$$

$x=(x_1, x_2)$	$F(x)$
00	0
01	1
10	1
11	0



Learning XOR Network (2)

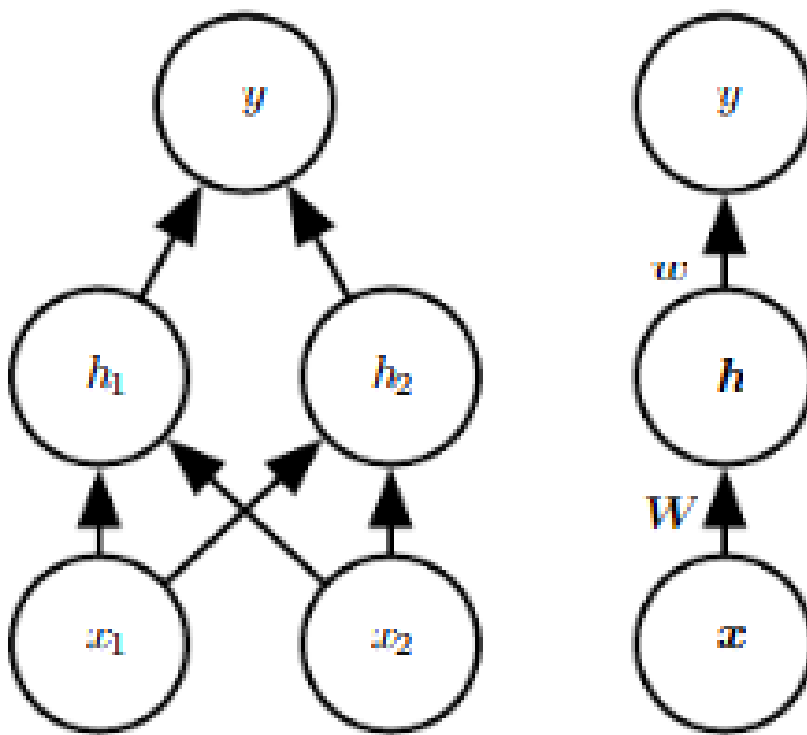
- Simple linear model ($y = \omega^T \mathbf{x} + b$) cannot solve this problem, since the problem is not linearly separable



Learning XOR Network (3)

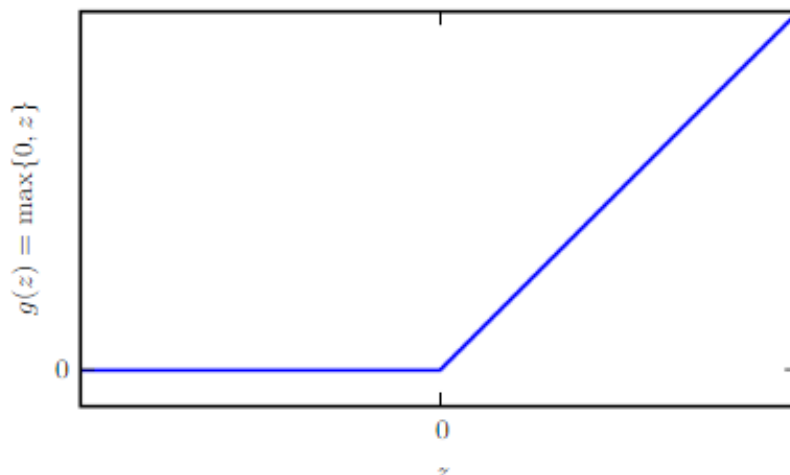
■ Consider a simple feedforward network:

- $\mathbf{h} = f^1(\mathbf{x}; \mathbf{W}, \mathbf{c}); \mathbf{W}^T \mathbf{x} + \mathbf{c}$
- $y = f^2(h; \boldsymbol{\omega}, b); \boldsymbol{\omega}^T \mathbf{h} + b$
- $y = f^2(f^1(x)) = f(x; \mathbf{W}, \boldsymbol{\omega}, b, \mathbf{c}) = \mathbf{x}^T \mathbf{W} \boldsymbol{\omega} = \mathbf{x}^T \boldsymbol{\omega}'$



Learning XOR Network (4)

- Since the linear function cannot solve the problem, we need some “nonlinearity” in the function $y = f^2(f^1(x))$
 - $\mathbf{h} = W^T \mathbf{x} + \mathbf{c} \rightarrow \mathbf{h} = g(W^T \mathbf{x} + \mathbf{c})$: g is a nonlinear function
- In modern neural networks, the default recommendation of g is to use the *rectified linear unit* (ReLU; Jarrett et al., 2009), defined by the activation function $g(z) = \max\{0, z\}$.



Learning XOR Network (5)

■ We can now specify our complete feedforward model as

- $f(x; W, \omega, b, c) = \omega^T \max(0, W^T x + c) + b$

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad b = 0$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

Training result

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \xrightarrow{XW + c} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{\omega, b} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Gradient-based Learning

- Cost function in neural network is generally non-convex, therefore iterative (stochastic, gradient-based) optimization is required
- All weights are initialized by small random values, biases are initialized by small positive values or zeros
- ***Back-propagation*** (BackProp) algorithm, which is common in deep neural networks, is used to compute the gradient

Cost Function

- Most modern deep neural networks are trained using the *maximum likelihood estimation*. Negative log-likelihood helps to avoid the gradient of the cost function being too small when its argument is too negative (i.e., $\frac{d e^{\|y-y\|^2}}{dx} \sim e^x, \frac{d \log(e^x)}{dx} \sim x$)

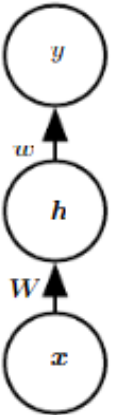
$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}). \quad \text{Cross entropy loss}$$

$$J(\theta) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const} \quad \text{When } p_{\text{model}} = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}); I)$$

- Cross-entropy cost function is more popular than mean-squared error or mean absolute error since some output units that saturate produce very small gradients when combined with these cost functions

Output Units (1)

- **Linear unit:** Given features \mathbf{h} , it outputs as
 - $\hat{y} = W^T \mathbf{h} + b$
 - Often used to produce a mean of a conditional Gaussian distribution
- **Sigmoid unit:** Given features \mathbf{h} , it outputs as
 - $\hat{y} = \sigma(W^T \mathbf{h} + b)$: σ is a logistic sigmoid function
 - Binary classification problem needs to predict the probability $P(y=1|x)$ within the range of $[0,1]$
 - We can not define gradient outside the intervals of $[0,1]$ when we simply use
$$P(y = 1 | \mathbf{x}) = \max \left\{ 0, \min \left\{ 1, \mathbf{w}^T \mathbf{h} + b \right\} \right\}$$
 - Derivative function of sigmoid is computed anywhere



Output Units (2)

- ***Softmax unit***: Output unit for multilabel classification problem
 - Multilabel classification problem needs to predict the probability $\hat{y}_i = P(y = i|\mathbf{x})$ within the range of $[0,1]$ and summation of \hat{y}_i must be one
 - Works well with the log-likelihood-based loss, but many other objective functions other than log-likelihood does not work because the gradient will vanish when the argument to exp becomes very negative
 - Named “softmax” because it is continuous and differentiable version of max

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}, \quad \mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b},$$

Hidden Units (1)

- **Rectified linear units** $g(z) = \max\{0, z\}$ are a default choice of hidden unit but one drawback of ReLU is they cannot learn via gradient-based methods on examples where their activation is zero
 - Nondifferentiable at $z=0$ isn't problematic in most cases
- Three variations are based on using a nonzero slope α_i when $z_i < 0$: $h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$
 - **Absolute value rectification** : $\alpha_i = -1$ ($g(z) = |z|$)
 - Was used for object recognition from images (Jarrett 2009)
 - **A leaky ReLU** : α_i is a fixed small value like 0.01 (Maas2013)
 - **A parametric ReLU (PReLU)leaky ReLU** : α_i is a learnable parameter (He2015)

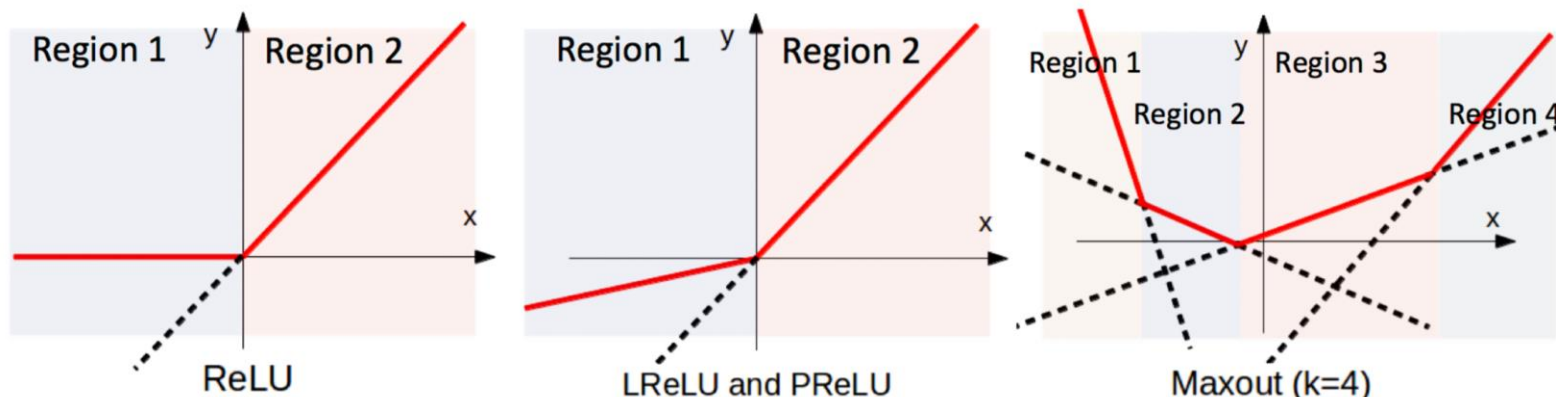
Hidden Units (2)

■ *Maxout units:*

- Instead of applying an element-wise function, maxout units divide \mathbf{z} into groups of k values. Each maxout unit then outputs the maximum element of one of these groups which provides a way of learning a piecewise linear function that responds to multiple directions in the input \mathbf{x} space

$$u_{ik}^{(l)} = \sum_{j=1}^m (w_{ijk}^{(l)} z_j^{(l-1)}) + b_{ik}^{(l)}$$

$$z_i^{(l)} = \max(u_{ik}^{(l)})$$



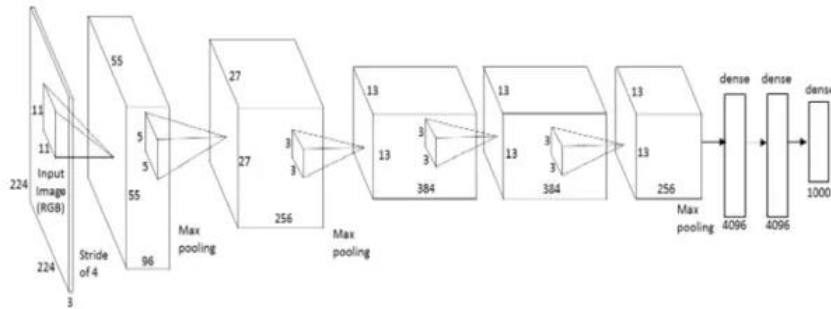
Hidden Units (3)

- ***Logistic sigmoid, hyperbolic tangent:*** $g(z) = \sigma(z), \tanh(z)$
 - Activation functions prior to the ReLU
 - They saturate to a high value when z is very positive, saturate to a low value when z is very negative, therefore a gradient-based algorithm was very difficult
 - Therefore, rather used as an output unit

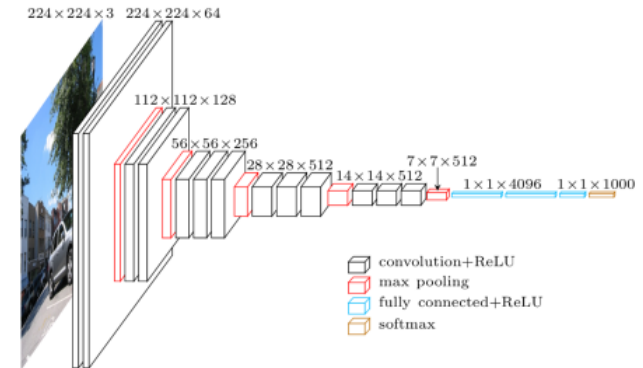
- Other hidden units (not commonly used)
 - Radial basis function unit
 - Softplus ($g(a) = \log(1 + e^a)$)
 - Hard tanh ($g(a) = \max(-1, \min(1, a))$)
 - Etc...

Architecture Design (1)

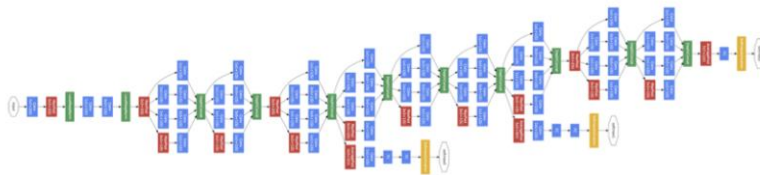
- **Architecture** refers to the overall structure of the network
 - Most neural networks are organized into groups of units called *layers* (i.e., unit \neq layer)



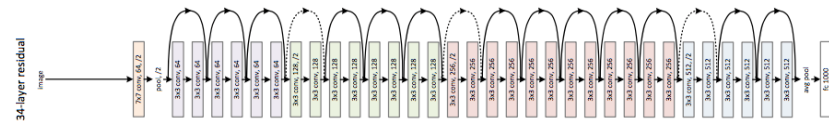
Alex-Net (Krizhevsky2012)
8 layers



VGG-Net (Simonyan2013)
16 layers



Google-Net (Segedy2015)
22 layers

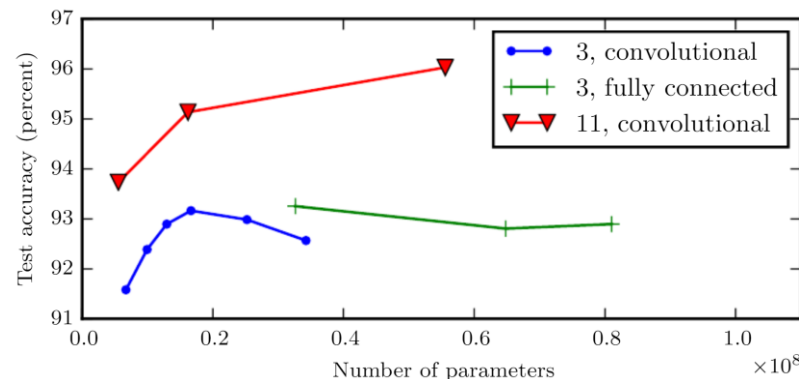


Res-Net (He2015)
125 layers

Architecture Design (2)

■ *Universal Approximation theorem* (Hornik1989)

- Regardless of what function we are trying to learn, a feedforward network with a single layer is sufficient to represent any function. However, the layer may be infeasibly large and may fail to learn and generalize correctly
- In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error



Architecture Design (3)

■ *Skip connections*

- Going from layer i to layer $i + 2$ or higher, which makes it easier for the gradient to flow from output layers to layers nearer the input (e.g., ResNet)

■ *Dense connections*

- Going from layer i , layer $i + 1$ to layer $i + 2$, which reduces the number of parameters to represent the function (e.g., DenseNet, Huan2017)

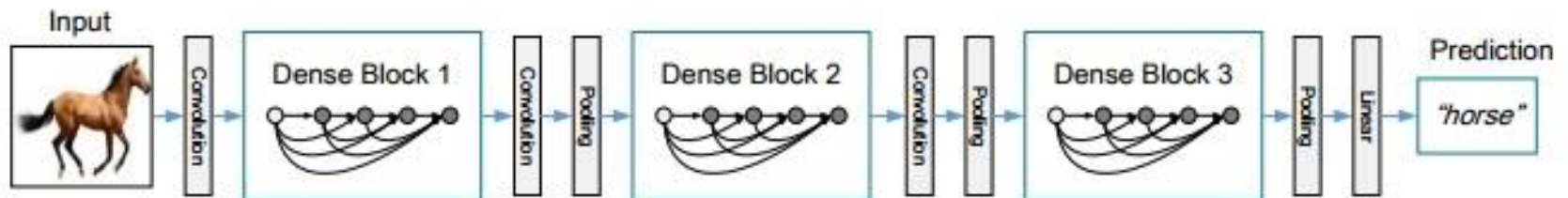


Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

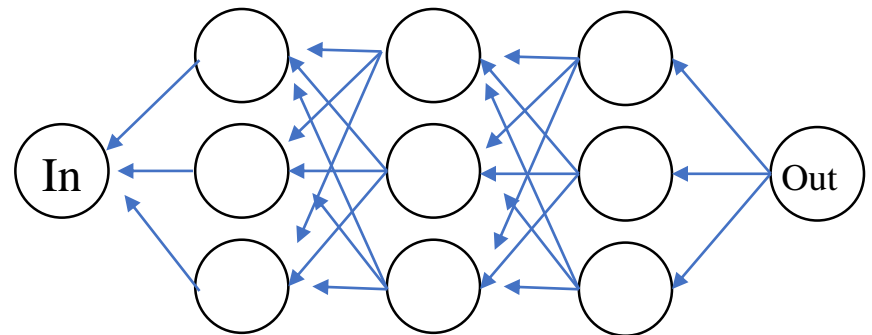
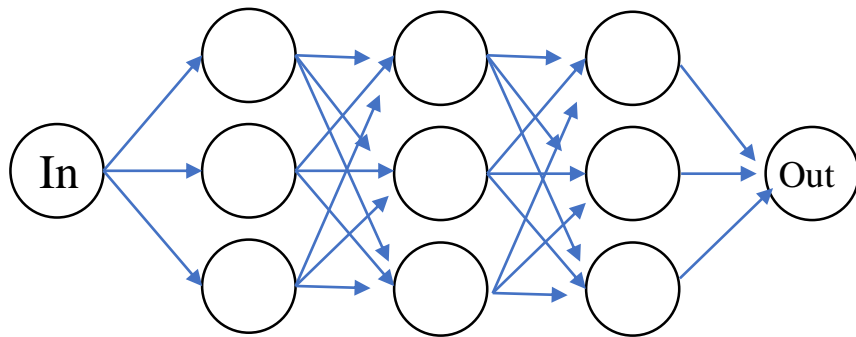
Back Propagation (1)

■ *Forward propagation*

- Given input x , the information is propagated up to the hidden units at each layer and finally an output \hat{y} is produced

■ *Back propagation (backprop; Rumelhart1986)*

- The back propagation algorithm allows information from the cost to then flow backward through the network (or graph) in order *to* compute the gradient of a function (*Not specific for the deep learning algorithm*)



Back Propagation (2)

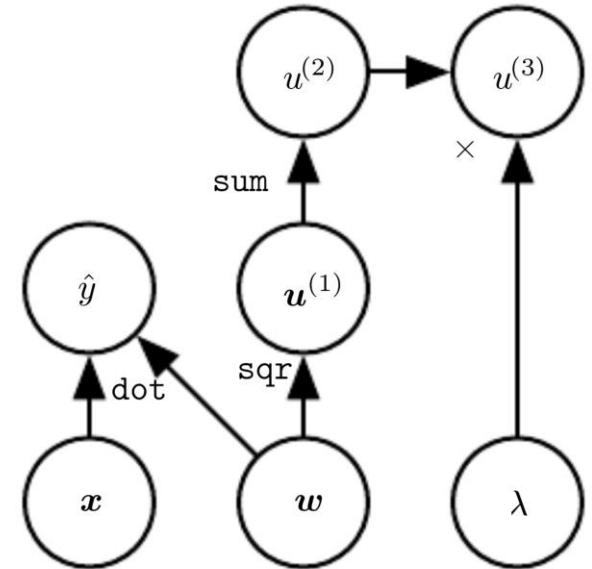
■ *Computational Graph*

- Each *node* in the graph indicates a variable
- An *operation* is a simple function of one or more variables
 - If a variable y is computed by applying an operation to a variable x , then we draw a directed edge from x to y

■ Notation about Gradient

- $\nabla_{\mathbf{X}} z$ denotes the gradient of value z with respect to a tensor \mathbf{X}
- $(\nabla_{\mathbf{X}} z)_i$ gives $\partial z / \partial x_i$
- If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j} \quad (\text{Chain rule})$$



Back Propagation (3)

- First, consider a computational graph describing how to compute a single scalar $u^{(n)}$ (i.e., the loss on a training example)
- We want to obtain gradient with respect to the n_i input nodes $(\frac{\partial u^{(n)}}{\partial u^{(i)}}, i \in \{1, 2, \dots, n_i\})$
- Each node w.r.t hidden units $i \in \{n_{i+1}, \dots, n_{n-1}\}$ is associated with an operator $f^{(i)}$ and is computed by evaluating the function $u^{(i)} = f(\mathbb{A}^{(i)})$, where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$

```
for  $i = 1, \dots, n_i$  do
   $u^{(i)} \leftarrow x_i$ 
end for
for  $i = n_i + 1, \dots, n$  do
   $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$ 
   $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$ 
end for
return  $u^{(n)}$ 
```

Back Propagation (4)

- We denote the computational subgraph \mathcal{B} for backprop with one node per node of graph \mathcal{G} for forward prop. Each node of \mathcal{B} computes the derivative $\partial u^{(n)} / \partial u^{(i)}$ via:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i: j \in \text{Parents}(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table[u(i)]` will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table[u(n)] ← 1`

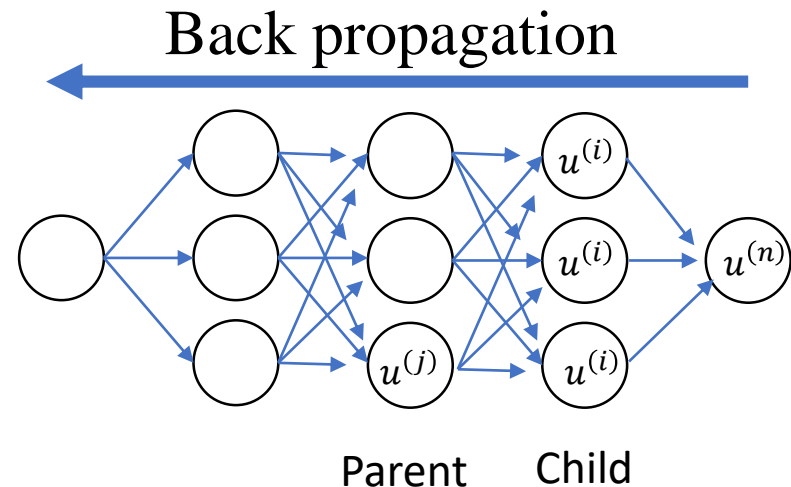
for $j = n - 1$ down to 1 **do**

 The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i: j \in \text{Pa}(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

`grad_table[u(j)] ← $\sum_{i: j \in \text{Pa}(u^{(i)})} \text{grad_table}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$`

end for

return `{grad_table[u(i)] | $i = 1, \dots, n_i$ }`



Back Propagation (5)

- Example: a fully connected multi-layer MLP (multi-layer perceptron)

Algorithm: Forward Propagation

Require: Network depth, l

Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model

Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model

Require: \mathbf{x} , the input to process

Require: \mathbf{y} , the target output

$$\mathbf{h}^{(0)} = \mathbf{x}$$

for $k = 1, \dots, l$ **do**

$$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$$

$$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$$

end for

$$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$$

$$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$$

Back Propagation (6)

$$\begin{aligned} \mathbf{a}^{(k)} &= \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)} \text{ opagation} \\ \mathbf{h}^{(k)} &= f(\mathbf{a}^{(k)}) \end{aligned}$$

After the forward computation, compute the gradient on the output layer:

$$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$$

for $k = l, l-1, \dots, 1$ **do**

Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

$$\text{e.g., } \frac{\partial J}{\partial \mathbf{a}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{a}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot f'(\mathbf{a})$$

Compute gradients on weights and biases (including the regularization term, where needed):

$$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$$

$$\text{e.g., } \frac{\partial J}{\partial \mathbf{b}} = \frac{\partial J}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{b}} = \frac{\partial J}{\partial \mathbf{a}} + (\text{reg.})$$

$$\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$$

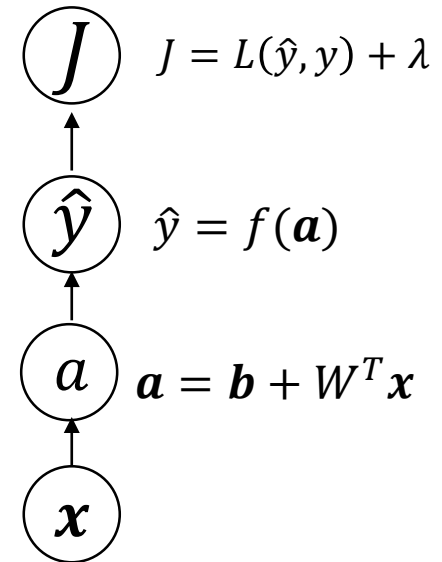
$$\text{e.g., } \frac{\partial J}{\partial \mathbf{W}} = \frac{\partial J}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{W}} = \frac{\partial J}{\partial \mathbf{a}} \mathbf{x}^T + (\text{reg.})$$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

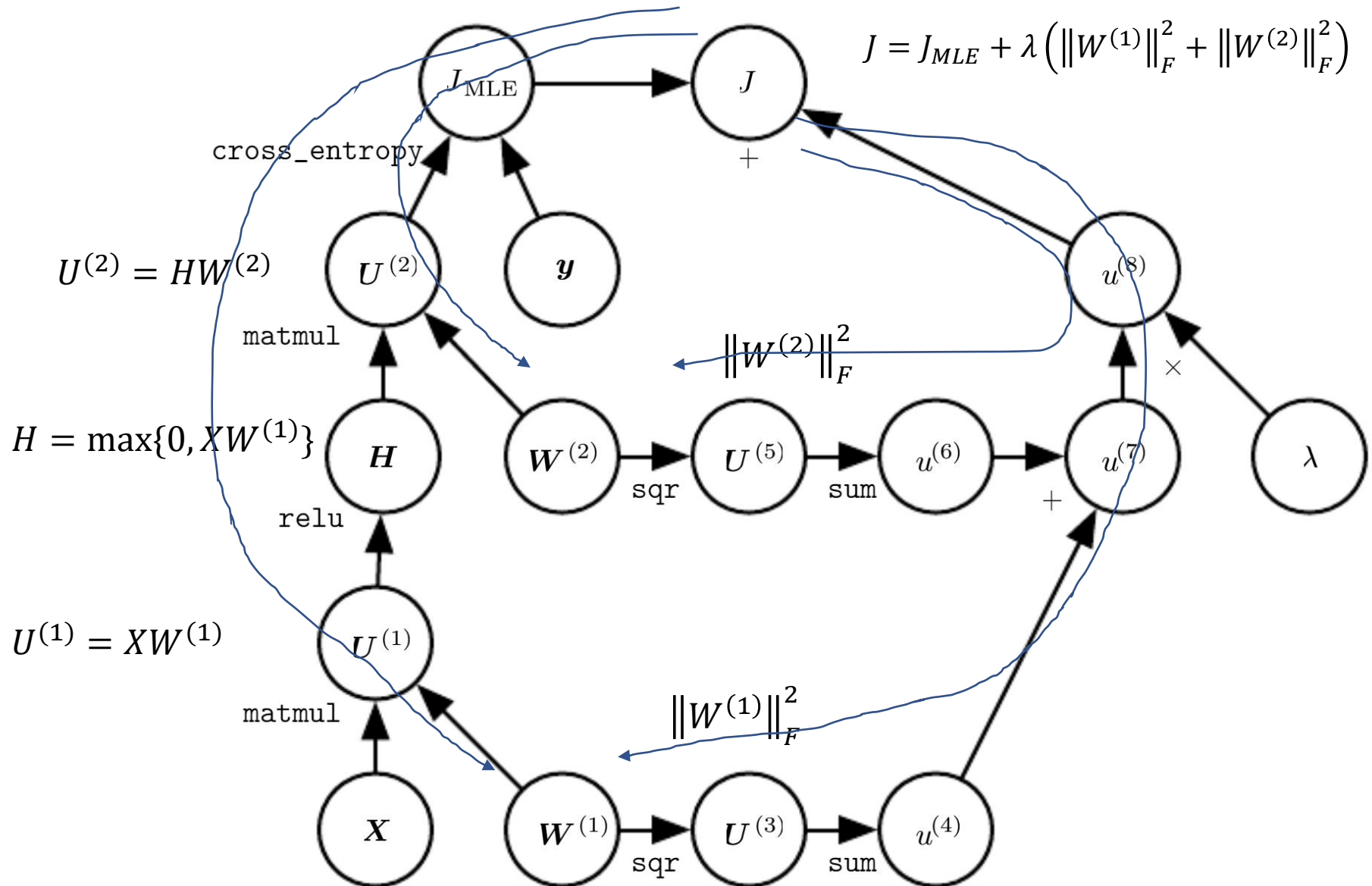
$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

$$\text{e.g., } \frac{\partial J}{\partial \mathbf{x}} = \frac{\partial J}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}} = \mathbf{W}^T \frac{\partial J}{\partial \mathbf{a}}$$

end for

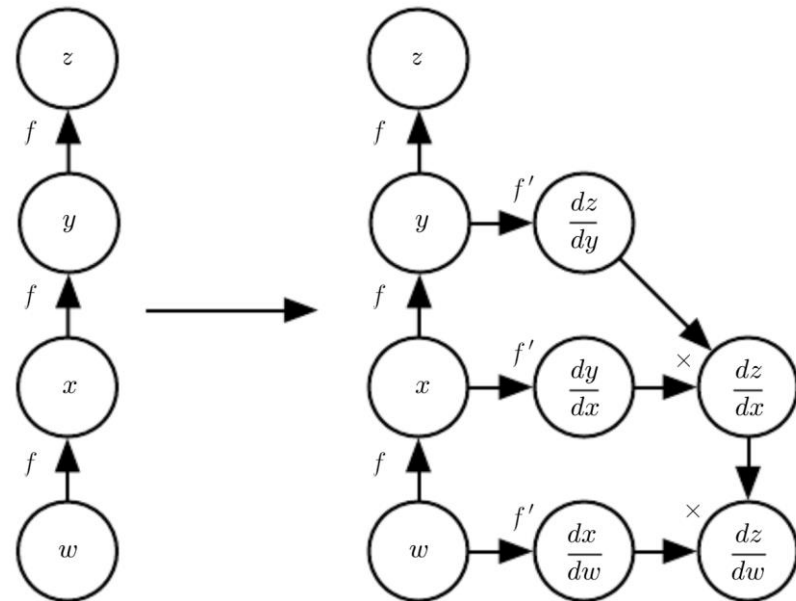


Back Propagation (7)



Implementation of General Back-Propagation (1)

- ***Symbolic-to-number differentiation*** (Used in Torch and Caffe)
 - Take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values
- ***Symbolic-to-symbol differentiation*** (Used in Theano and TensorFlow)
 - Take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives.
 - The backprop algorithm avoids access to any actual specific numeric values



Implementation of General Back-Propagation (2)

- Software implementations of backprop provide both the operations and their “bprop” method
- We assume that each variable \mathbf{V} associated with assumptions:
 - $\text{get_operation}(\mathbf{V})$: Returns the operation that computes \mathbf{V} , represented by the edges coming into \mathbf{V} in the graph
 - $\text{get_consumers}(\mathbf{V}, \mathcal{G})$: Returns the list of variables that are children of \mathbf{V} in the computational graph \mathcal{G}
 - $\text{get_inputs}(\mathbf{V}, \mathcal{G})$: Returns the list of variables that are parents of \mathbf{V} in the computational graph \mathcal{G}
 - Each operation is associated with a “bprop” operation, which computes a Jacobian-vector product as
$$\nabla_{\mathbf{x}} z = \sum_j (\nabla_{\mathbf{x}} Y_j) \frac{\partial z}{\partial Y_j}$$
 - For example, given a multiplication operation to create a variable $C = AB$, bprop requests the gradient w.r.t. A or B without knowing any differentiation rules.

Implementation of General Back-Propagation (3)

- When “bprop” is called, `op.bprop(inputs, X, G)` returns:

$$\sum_i (\nabla_X \text{op.f}(\text{inputs})_i) G_i$$

- Here, `inputs` is a list of inputs that are supplied to the operation
 - `op.f` is the mathematical function that the operation implements
 - `X` is the input whose gradient we wish to compute
 - `G` is the gradient on the output of the operation
- *Software engineers who build a new implementation of back-propagation or advanced users who need to add their own operation to an existing library must usually derive the `op.bprop` method for any new operations manually*

Implementation of General Back-Propagation (4)

- The deep learning community uses computational graphs that are usually represented by explicit data structures created by specialized libraries
- It requires the library developer to define the bprop methods for every operation and limiting the users of the library to only those operations that have been defined
- However it has benefit of allowing customized back-propagation rules to be developed for each operation, enabling the developer to improve speed or stability in nonobvious
- Back-propagation is not the only way of computing the gradient, but it is a practical method that continues to serve the deep learning community well

High-Order Derivatives

- We are often interested in computing the Hessian matrix.
 - If we have a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$, the Hessian matrix is of size $n \times n$
 - Since n will be the number of parameters, the entire Hessian matrix is infeasible to even present
- *Krylov method*
 - A set of iterative techniques for performing various operations, such as approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues without using any operation other than matrix-vector products. Using this, we can compute Hessian in the form of:

$$H\mathbf{v} = \nabla_x \left[\left(\nabla_x f(\mathbf{x}) \right)^T \mathbf{v} \right]$$

Conclusion with Historical Remarks

- The core ideas behind feedforward networks (BackProp, gradient descent) have not changed since the 1980s.
- Most of the improvement in neural network performance from 1986 to Now can be attributed two factors: larger datasets and larger networks
- One of the algorithmic changes was replacement of means squared error with the cross-entropy family of loss functions and the idea of maximum likelihood, which less suffers from saturation and slow learning than using the mean squared error loss
- Another algorithmic change was the replacement of hidden sigmoid unit with ReLu and its variants. Why ReLu is better than non-linear ones is still of interest