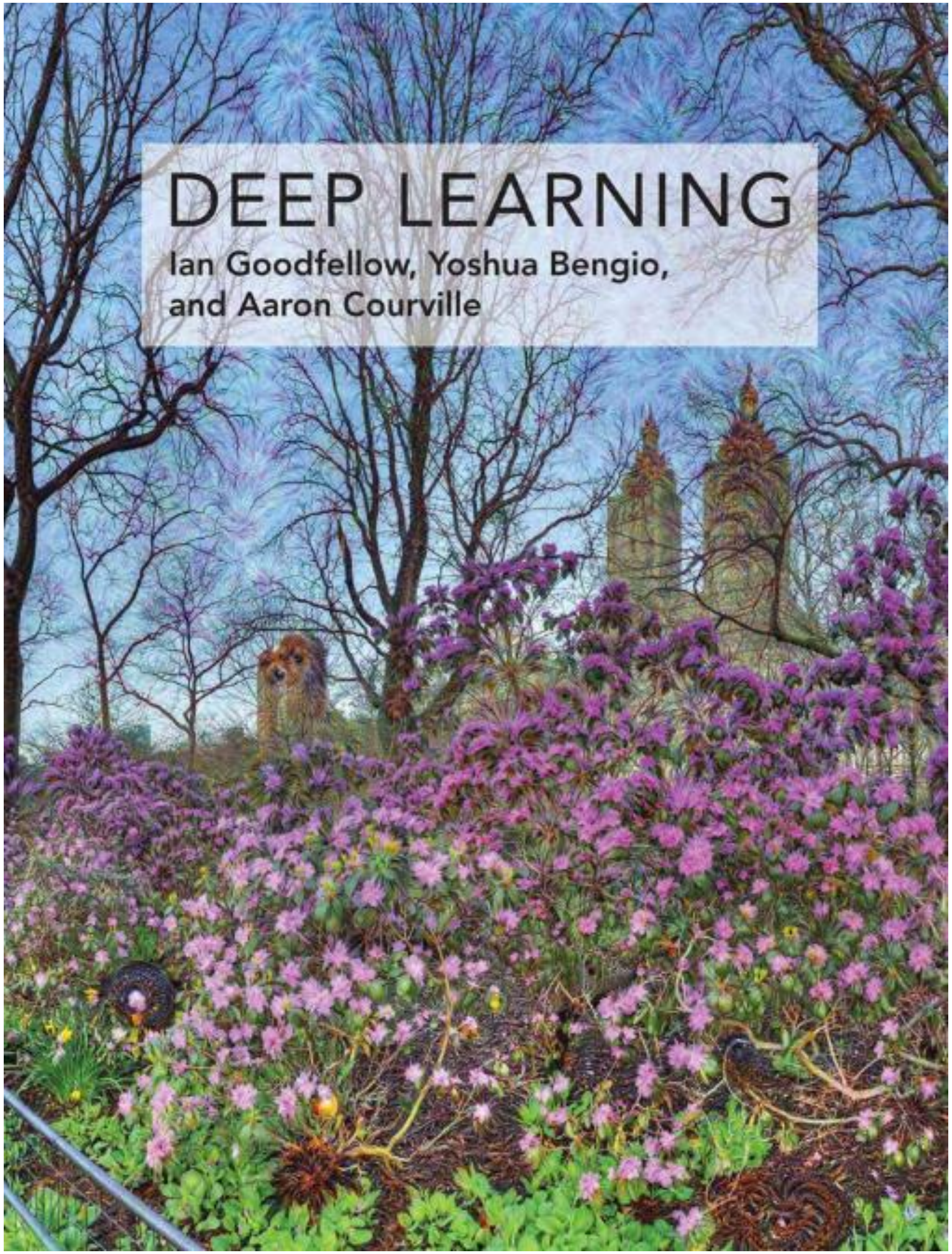
Fundamentals of Media Processing (Machine Learning Part)

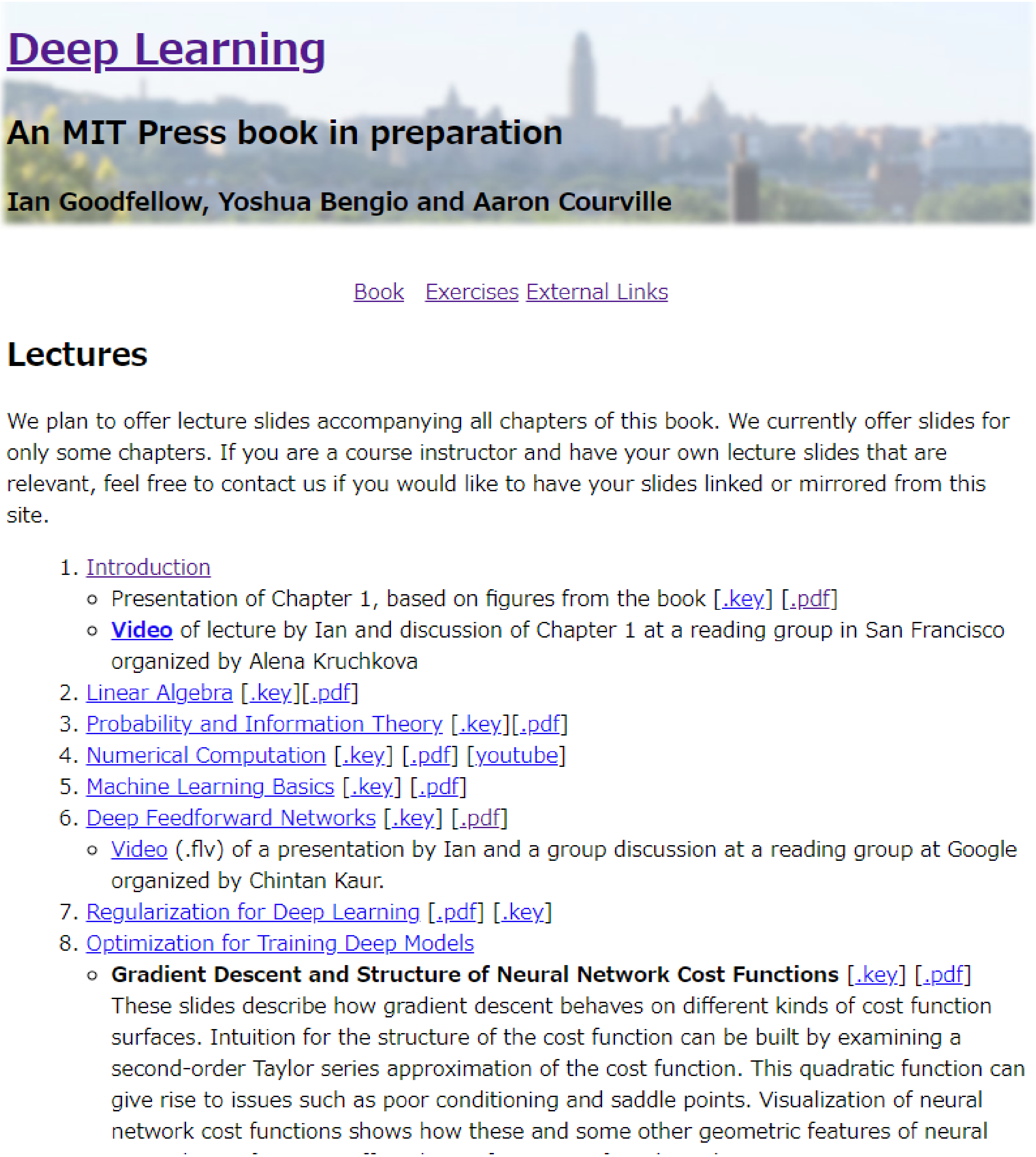
Lecturer:
佐藤　真一（Prof. SATO　Shinichi）
池畑　諭（Prof. IKEHATA　Satoshi）10/27, 11/10, 11/17, 11/21, 12/1, 12/8
山岸　順一（Prof. Junichi Yamagishi）
児玉　和也（Prof. KODAMA　Kazuya）
孟　洋（Prof. MO　Hiroshi）



Chapter 1-9 (out of 20)

An introduction to a broad range of topics in deep learning, covering mathematical and conceptual background, deep learning techniques used in industry, and research perspectives.

- Due to my background, I will mainly talk about “image”
- I will introduce some applications beyond this book



Deep Learning

An MIT Press book in preparation

Ian Goodfellow, Yoshua Bengio and Aaron Courville

Book Exercises External Links

Lectures

We plan to offer lecture slides accompanying all chapters of this book. We currently offer slides for only some chapters. If you are a course instructor and have your own lecture slides that are relevant, feel free to contact us if you would like to have your slides linked or mirrored from this site.

1. Introduction
   - Presentation of Chapter 1, based on figures from the book [.key] [.pdf]
   - **Video** of lecture by Ian and discussion of Chapter 1 at a reading group in San Francisco organized by Alena Kruchkova
2. Linear Algebra [.key][.pdf]
3. Probability and Information Theory [.key][.pdf]
4. Numerical Computation [.key] [.pdf] [youtube]
5. Machine Learning Basics [.key] [.pdf]
6. Deep Feedforward Networks [.key] [.pdf]
   - Video (.flv) of a presentation by Ian and a group discussion at a reading group at Google organized by Chintan Kaur.
7. Regularization for Deep Learning [.pdf] [.key]
8. Optimization for Training Deep Models
   - **Gradient Descent and Structure of Neural Network Cost Functions** [.key] [.pdf] These slides describe how gradient descent behaves on different kinds of cost function surfaces. Intuition for the structure of the cost function can be built by examining a second-order Taylor series approximation of the cost function. This quadratic function can give rise to issues such as poor conditioning and saddle points. Visualization of neural network cost functions shows how these and some other geometric features of neural

Free copy of the book and useful materials are available at

https://www.deeplearningbook.org/lecture_slides.html

Schedule

10/27 (Today)	Introduction	Chap. 1
	probability, information theory, numerical computation	Chap. 2,3,4
11/10	Machine Learning Basics	Chap. 5
11/17, 11/27, 12/1	Deep Feedforward Networks	Chap. 6
	Regularization and Deep Learning	Chap. 7
	Optimization for Training Deep Models	Chap. 8
12/8	Convolutional Neural Networks	Chap. 9 and more

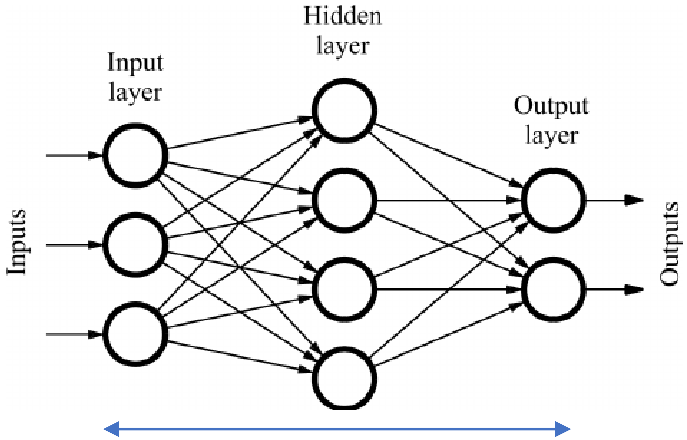
Deep Feedforward Networks

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 $y = f(x;\theta,\boldsymbol{\omega}) = \phi(x;\theta)^{T}\boldsymbol{\omega}$**:** The deep neural networks try to learn ϕ

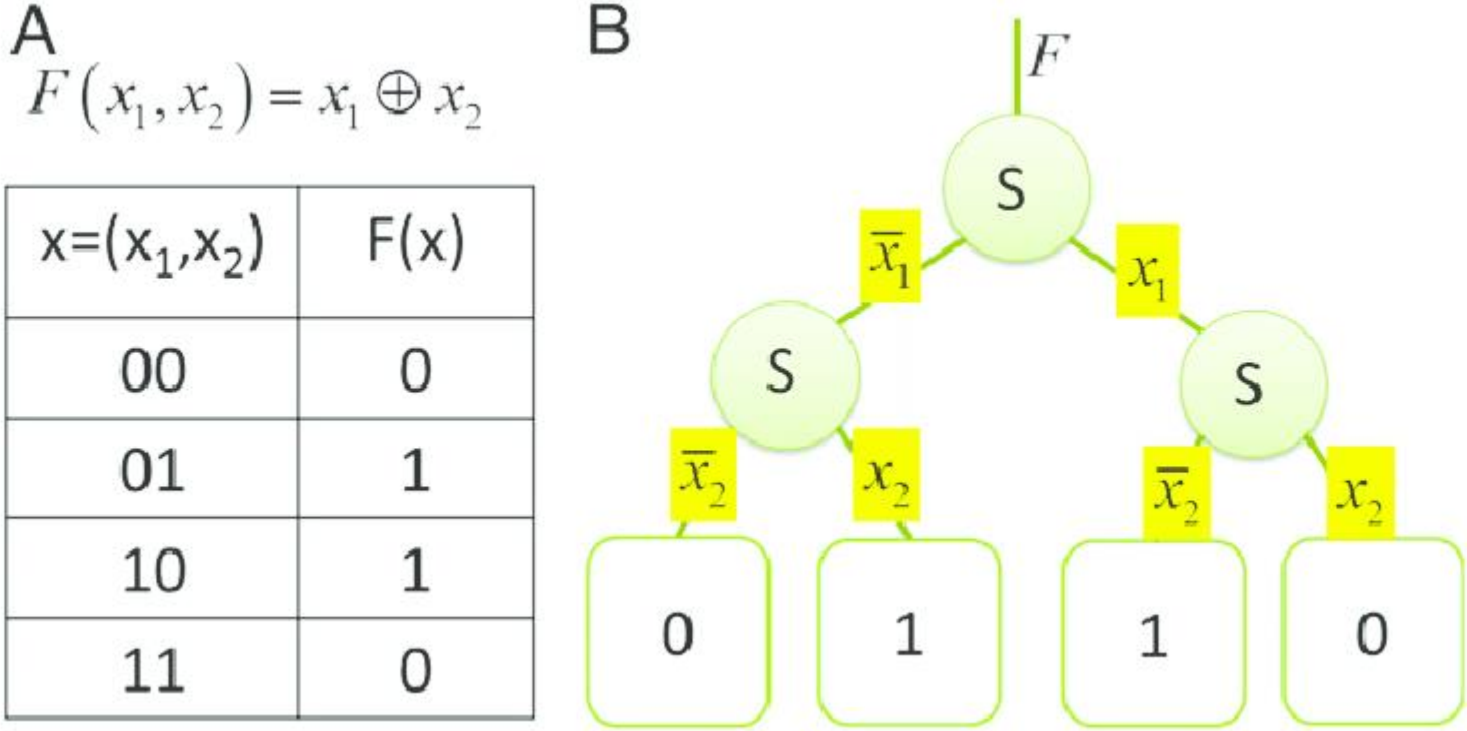


Depth of the network

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

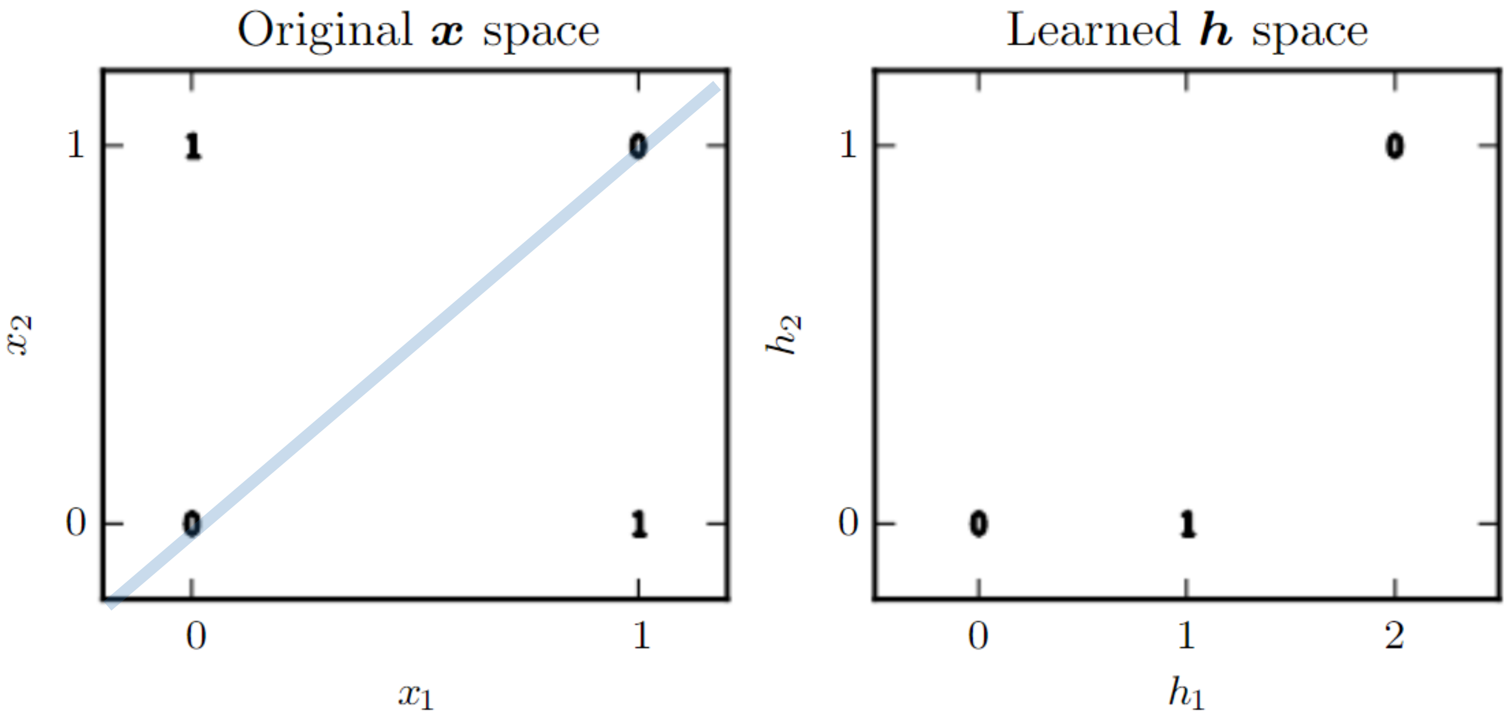
$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta})\right)^2$$



x=(x$_1$,x$_2$)	F(x)
00	0
01	1
10	1
11	0

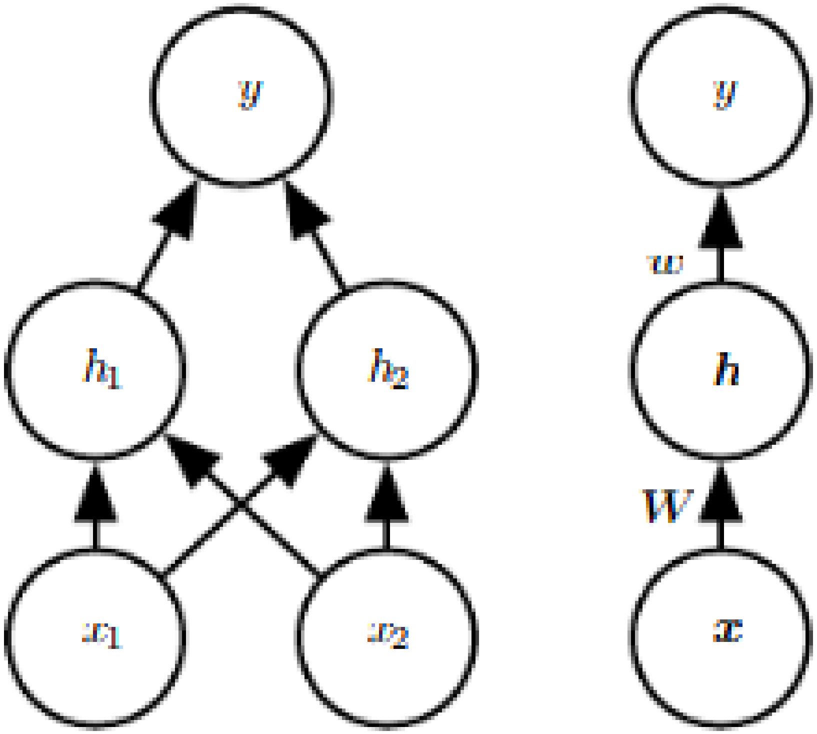
Learning XOR Network (2)

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



Learning XOR Network (3)

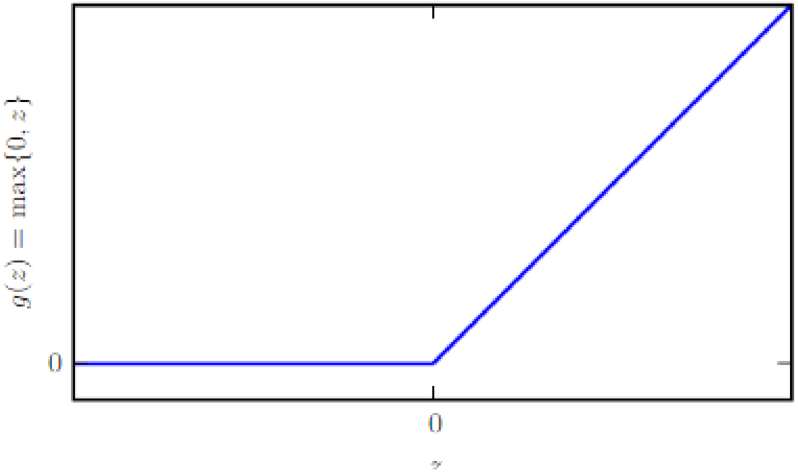
- Consider a simple feedforward network:
  - $\boldsymbol{h} = f^1(\boldsymbol{x}; W, \boldsymbol{c}); W^T\boldsymbol{x} + \boldsymbol{c}$
  - $y = f^2(h; \boldsymbol{\omega}, b); \boldsymbol{\omega}^T\boldsymbol{h} + b$
  - $y = f^2\left(f^1(x)\right) = f(x; W, \boldsymbol{\omega}, b, \boldsymbol{c}) = \boldsymbol{x}^T W\boldsymbol{\omega} = \boldsymbol{x}^T\boldsymbol{\omega}'$



Learning XOR Network (4)

- Since the linear function cannot solve the problem, we need some "nonlinearity" in the function $y = f^2\left(f^1(x)\right)$
  - $\boldsymbol{h} = W^T\boldsymbol{x} + \boldsymbol{c} \rightarrow \boldsymbol{h} = g(W^T\boldsymbol{x} + \boldsymbol{c})$: g is a nonlinear function

- In modern neural networks, the default recommendation of g is to use the ***rectified linear unit*** (ReLU; Jarrettet 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
  - $f(x; W, \boldsymbol{\omega}, b, \boldsymbol{c}) = \boldsymbol{\omega}^T \max(0, W^T \boldsymbol{x} + \boldsymbol{c}) + b$

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

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

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

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

Training result

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

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., $\frac{\mathrm{d}e^{\|y-y\|^2}}{dx} \sim e^x, \frac{\mathrm{d}\log(\mathrm{e}^x)}{dx} \sim x$)

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

Cross entropy loss

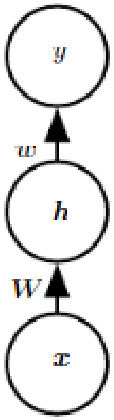
$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}$$

When $p_{\text{model}} = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x}; \boldsymbol{\theta}); I)$

- 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 $\boldsymbol{h}$, it outputs as
  - $\hat{y} = W^T\boldsymbol{h} + b$
  - Often used to produce a mean of a conditional Gaussian distribution



- ***Sigmoid unit***: Given features $\boldsymbol{h}$, it outputs as
  - $\hat{y} = \sigma(W^T\boldsymbol{h} + b)$: 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

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}$$

  - 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 $\widehat{y_i} = P(y = i|\boldsymbol{x})$ within the range of [0,1] and summation of $\widehat{y_i}$ 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

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \qquad \boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b},$$

i = 1,…,# of class

Hidden Units (1)

- ***Rectified linear units*** $g(z) = \max\{0, z\}$ are a default choice of hidden unit but one drawbacks 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)

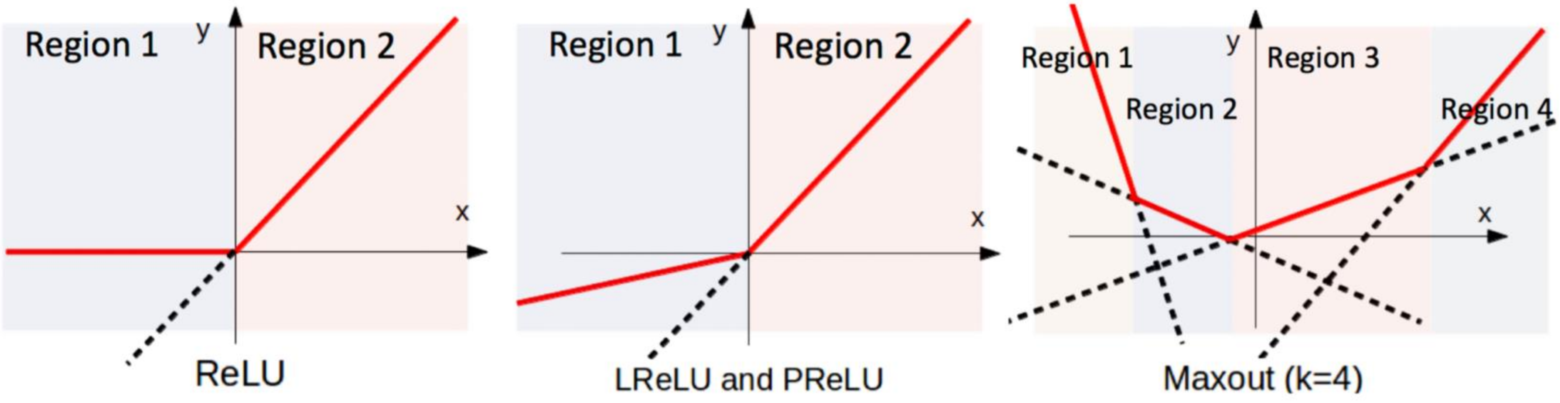
  - ***A parametric ReLU (PReLU)leaky ReLU*** : α_i is a learnable parameter (He2015)

Hidden Units (2)

- ***Maxout units:***
  - Instead of applying an element-wise function, maxout units divide ***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 ***x*** space

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

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



ReLU

LReLU and PReLU

Maxout (k=4)

https://cs.adelaide.edu.au/~carneiro/publications/226.pdf

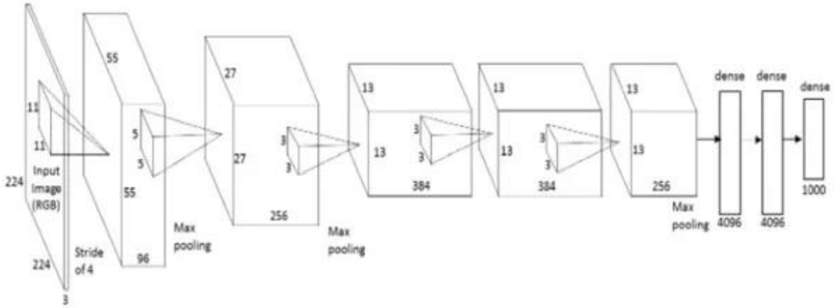
Hidden Units (3)

- ***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

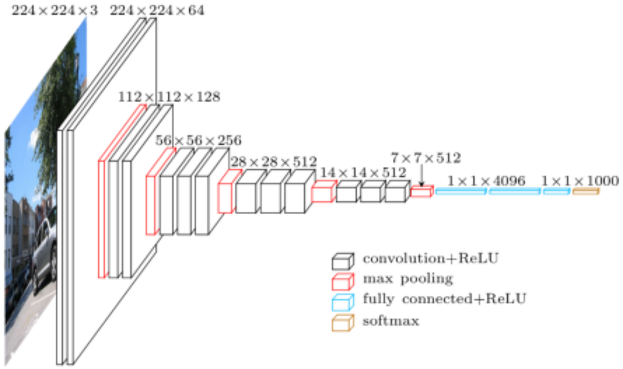
- Other hidden units (not commonly used)
  - Radial basis function unit
  - Softplus ($g(a) = \log(1 + e^{a})$)
  - Hard tanh ($g(a) = \max(-1, \min(1, a))$)
  - Etc…

Architecture Design (1)

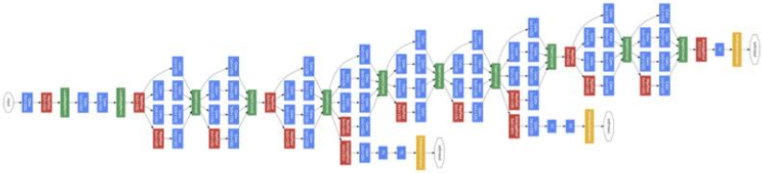
- ***Architecture*** refers to the overall structure of the network
  - Most neural networks are organized into groups of units called ***layers*** (i.e., unit ≠ layer)



Alex-Net (Krizhevsky2012)
8 layers

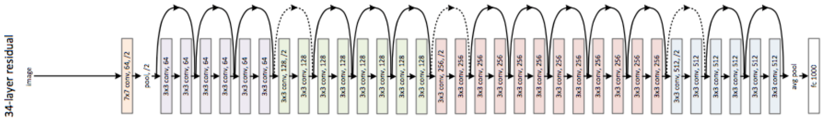


VGG-Net (Simonyan2013)
16 layers



Google-Net (Segedy2015)
22 layers

Convolution
Pooling
Softmax
Other



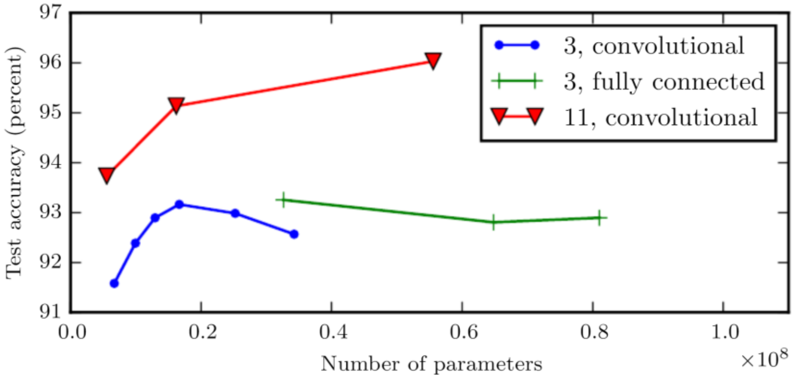
Res-Net (He2015)
125 layers

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)

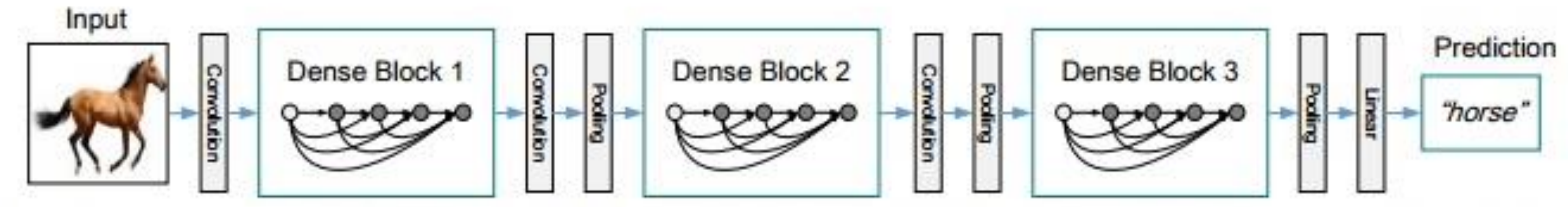
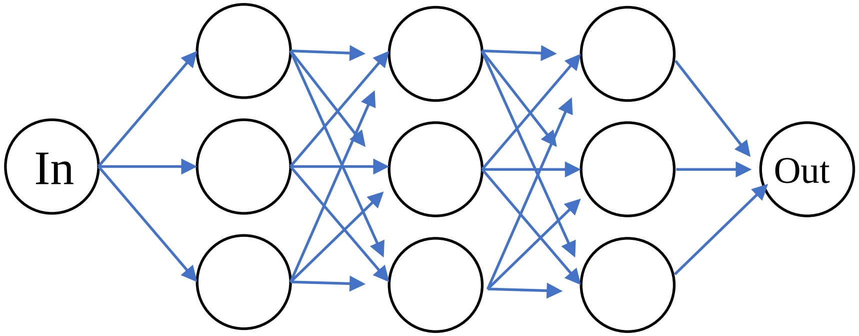


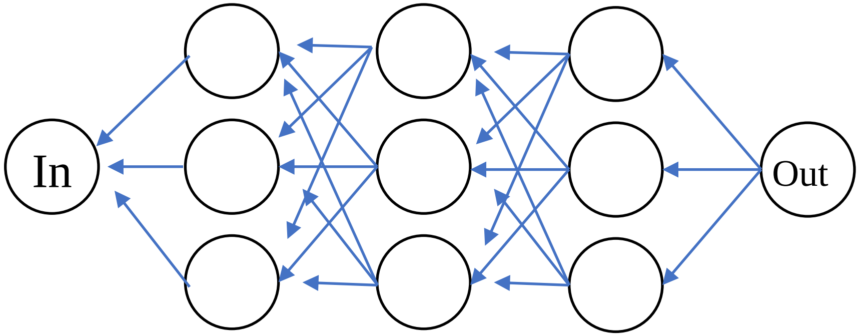
Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

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*)





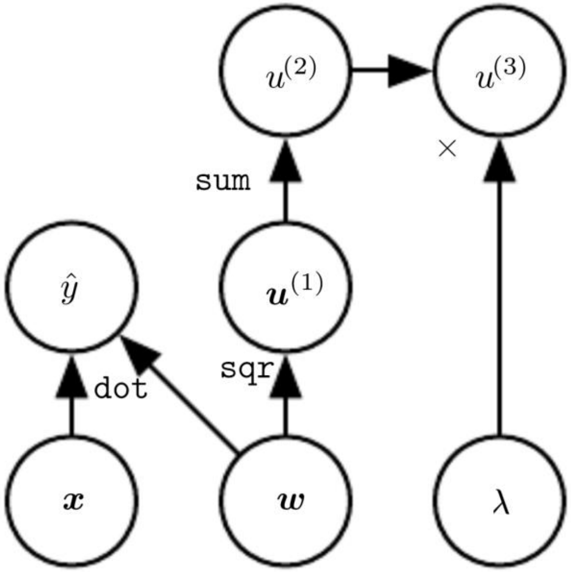
Back Propagation (2)

- ***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

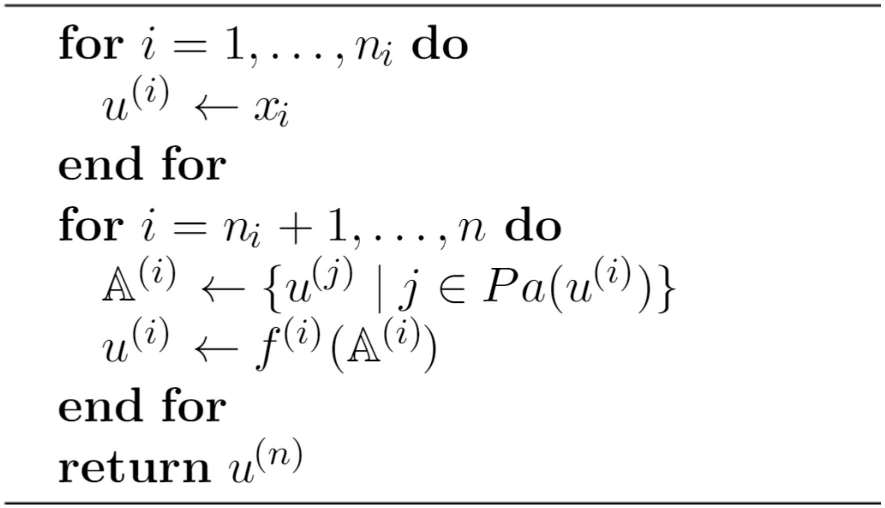
$$\nabla_{\mathbf{X}} z = \sum_j \left(\nabla_{\mathbf{X}} Y_j\right) \frac{\partial z}{\partial Y_i}$$

(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 $u^{(i)} = f\left(\mathbb{A}^{(i)}\right)$, where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$



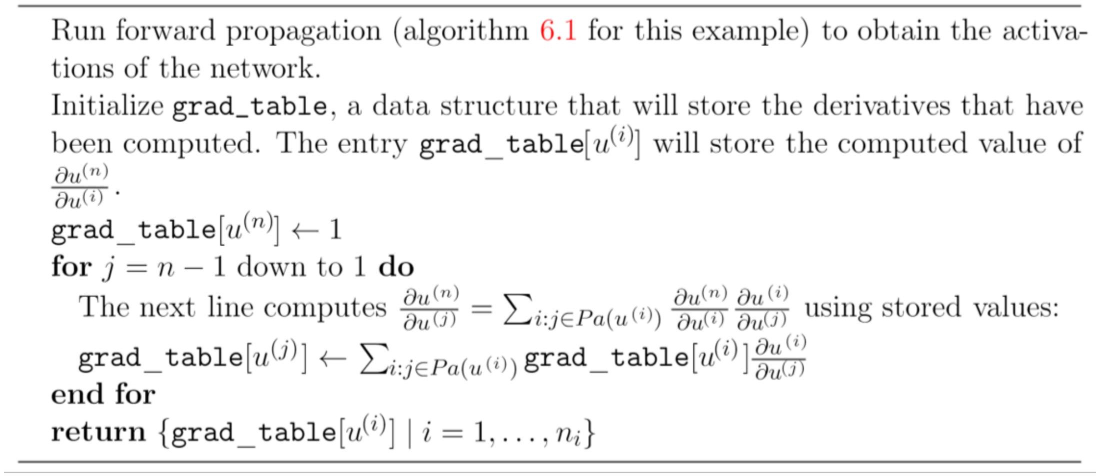
```
for $i = 1, \dots, n_i$ do
    $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \dots, n$ do
    $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
    $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$
```

Algorithm of forward propagation

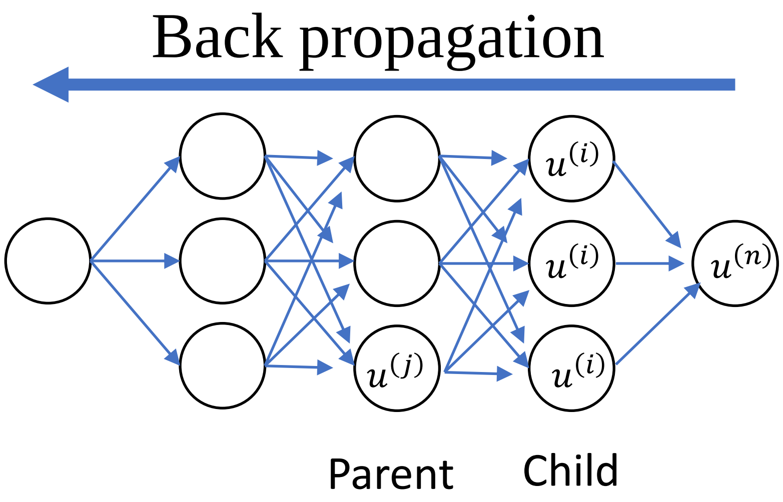
Back Propagation (4)

- 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:

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



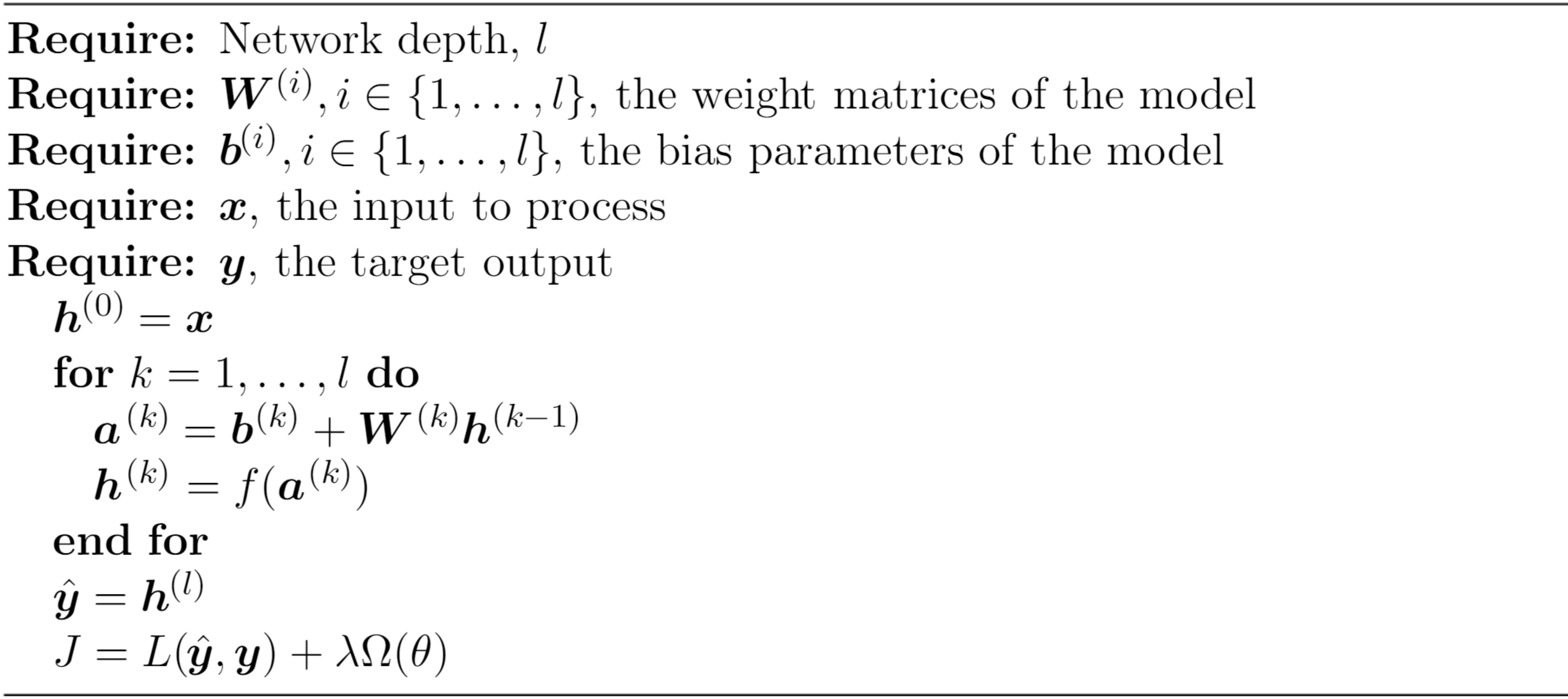
```
Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.
Initialize grad_table, a data structure that will store the derivatives that have been computed. The entry grad_table[$u^{(i)}$] will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
grad_table[$u^{(n)}$] $\leftarrow 1$
for $j = n - 1$ down to 1 do
    The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
    grad_table[$u^{(j)}$] $\leftarrow \sum_{i:j\in Pa(u^{(i)})}$ grad_table[$u^{(i)}$]$\frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {grad_table[$u^{(i)}$] | $i = 1, \ldots, n_i$}
```



Back Propagation (5)

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

Algorithm: Forward Propagation



```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$
```

Back Propagation (6)

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

Algorithm: Back Propagation

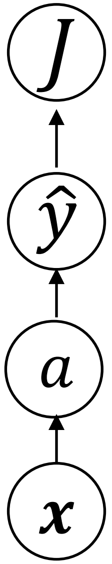
```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
  Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for
```

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

e.g., $\frac{\partial J}{\partial \boldsymbol{b}^k} = \frac{\partial J}{\partial \boldsymbol{a}^k} \frac{\partial \boldsymbol{a}^k}{\partial \boldsymbol{b}^k} = \frac{\partial J}{\partial \boldsymbol{a}^k} + (\boldsymbol{reg.})$

e.g., $\frac{\partial J}{\partial W^k} = \frac{\partial J}{\partial \boldsymbol{a}^k} \frac{\partial \boldsymbol{a}^k}{\partial W} = \frac{\partial J}{\partial \boldsymbol{a}^k} \boldsymbol{h}^{k^T} + (\boldsymbol{reg.})$

e.g., $\frac{\partial J}{\partial \boldsymbol{h}^{k-1}} = \frac{\partial J}{\partial \boldsymbol{a}^k} \frac{\partial \boldsymbol{a}^k}{\partial \boldsymbol{h}^{k-1}} = W^T \frac{\partial J}{\partial \boldsymbol{a}^{k-1}}$



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

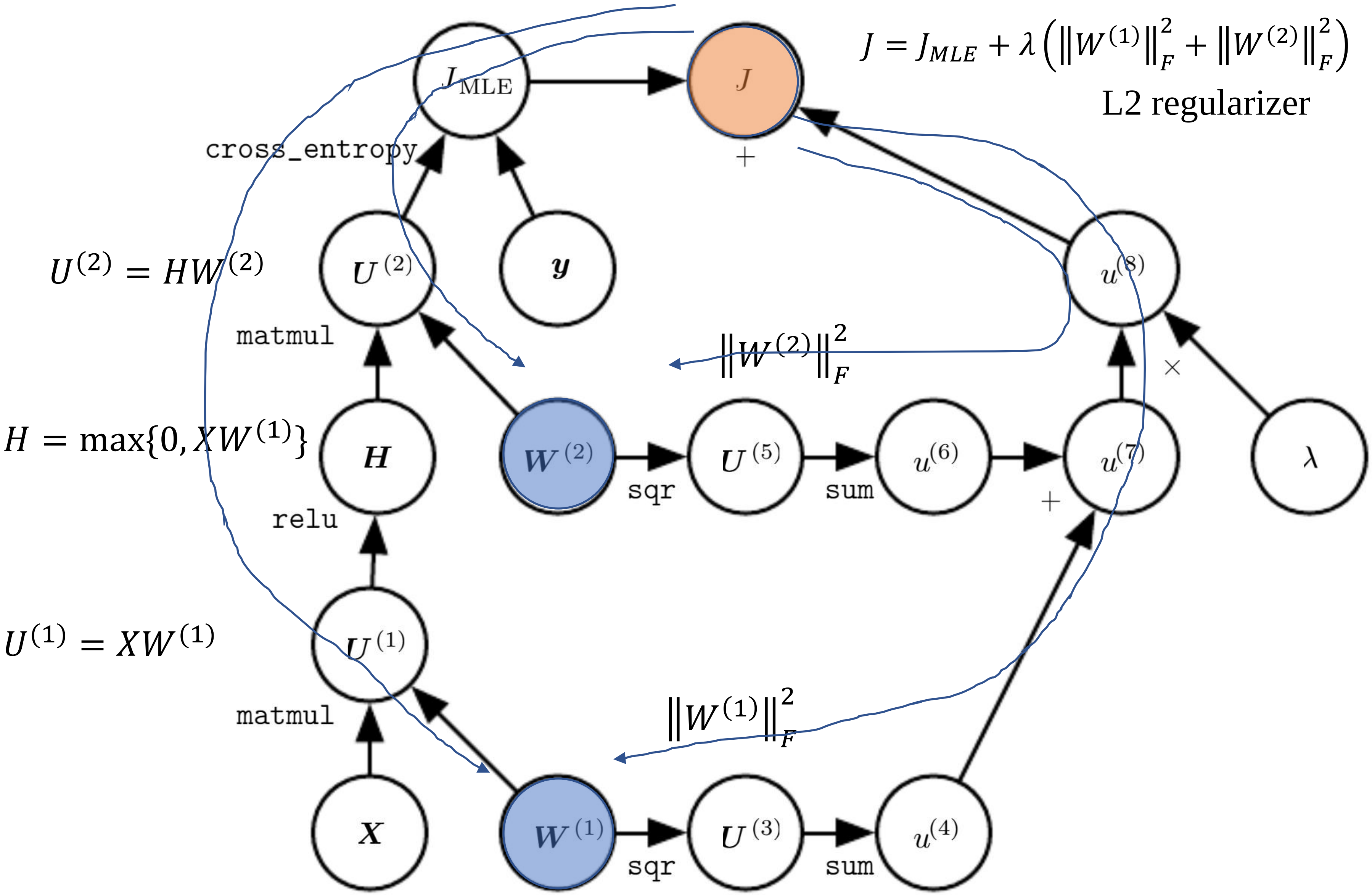
$\hat{y} = f(\boldsymbol{a})$

$\boldsymbol{a} = \boldsymbol{b} + W^T \boldsymbol{x}$

Back Propagation (7)

$$J = J_{MLE} + \lambda\left(\left\|W^{(1)}\right\|_F^2 + \left\|W^{(2)}\right\|_F^2\right)$$

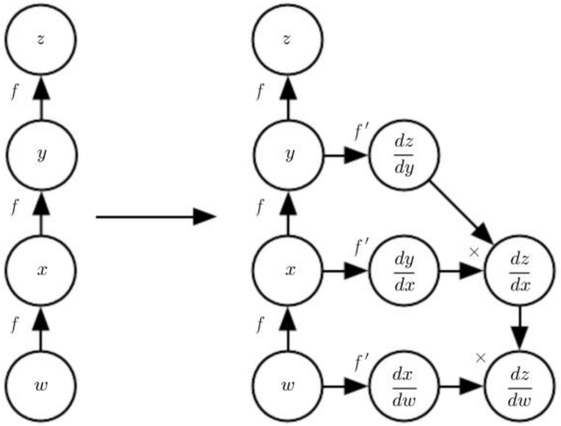
L2 regularizer



Implementation of General Back-Propagation (1)

- ***Symbolic-to-number differentiation*** (Used in Torch and Caffe)
  - Take a computational graph and as 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 provides a symbolic description of the desired derivatives.
  - It is possible to run backpropagation again, differentiating the derivatives to obtain higher derivatives (e.g., for computing Hessian)



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:
  - get_operation($\mathbf{V}$): Returns the operation that computes $\mathbf{V}$, represented by the edges coming into $\mathbf{V}$ in the graph
  - get_consumers($\mathbf{V}, \mathcal{G}$): Returns the list of variables that are children of $\mathbf{V}$ in the computational graph $\mathcal{G}$
  - 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

$$\nabla_{\mathbf{X}} z = \sum_j \left(\nabla_{\mathbf{X}} Y_j\right) \frac{\partial z}{\partial Y_i}$$

  - 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:

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

  - 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 mush 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 $f\colon \mathbb{R}^n \to \mathbb{R}$, 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:

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

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