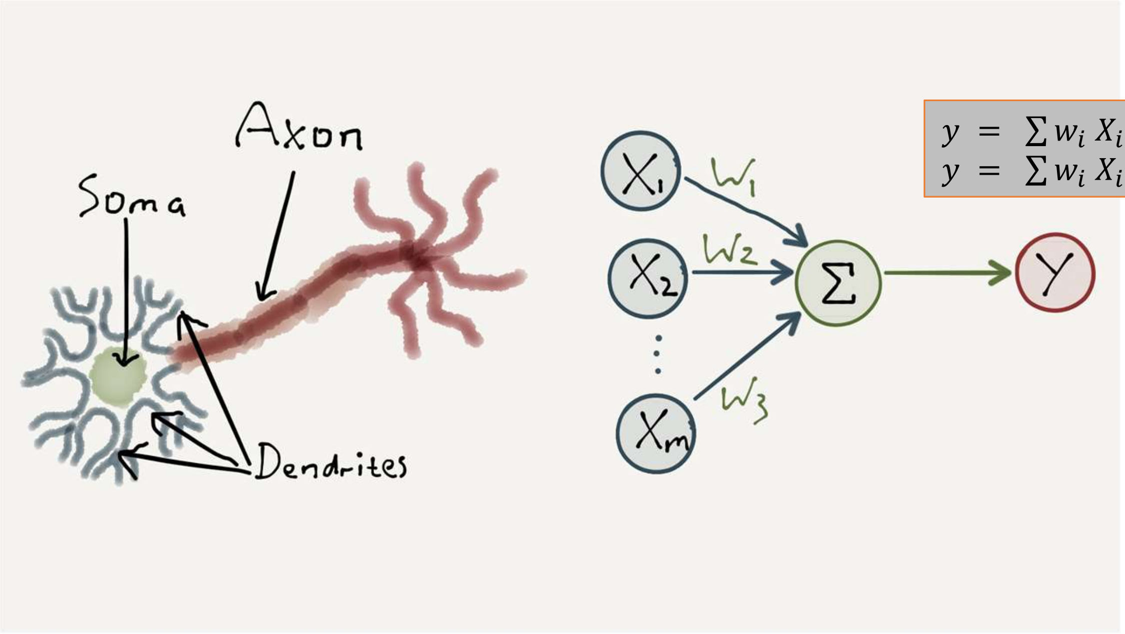
Introduction to AI for postgraduate students

Lecture Note 6
Deep Forward Networks

Perceptron

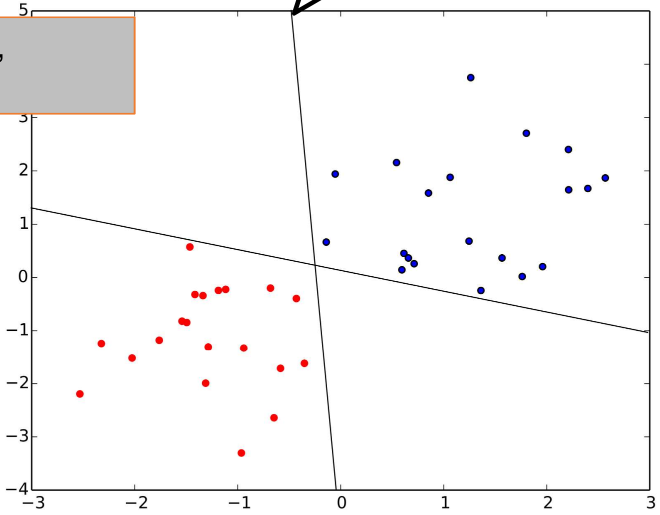
Perceptron

- Artificial neuron
- algorithm for supervised learning of binary linear classifiers



$y = \sum w_i X_i \geq 0$, then class 0,
$y = \sum w_i X_i < 0$, then class 1

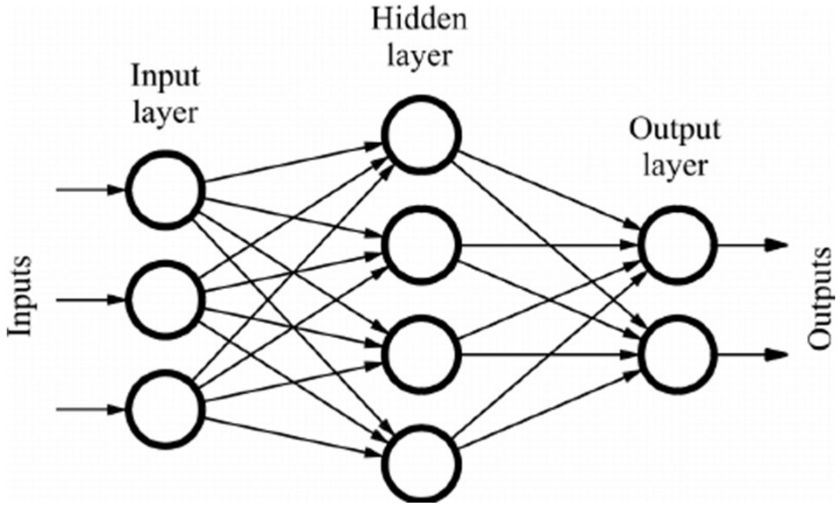
$\sum w_i X_i = 0$



Deep Forward Networks

DFN = feedforward neural network = multilayer perceptrons (MLPs)

- Goal: approximate some function f^*
  - e.g.) DFN defines a mapping $\mathbf{y}=f(\mathbf{x};\boldsymbol{\theta})$ and learns the parameters $\boldsymbol{\theta}$
- loosely inspired by neuroscience
- the goal is not to perfectly model the brain, but to implement function approximation machines



There is no ***feedback*** connection.

Example: **Learning XOR**

- Input domain: $\mathbb{X} = -\{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$
- Goal: find the parameters $\boldsymbol{\theta}$ such that our model $y = f(\boldsymbol{x};\Theta)$ becomes as similar as possible to $y = f^*(\boldsymbol{x})$
- We treat this problem as a regression problem and use a MSE loss function

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

Limitation in Linear Model

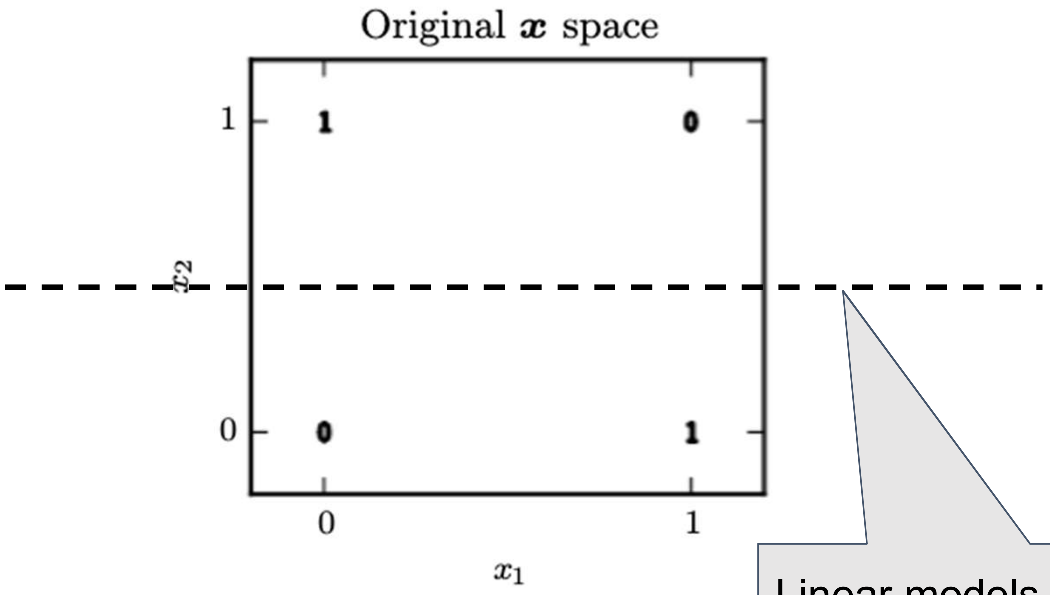
We choose a linear model $f(\boldsymbol{x};\boldsymbol{w},b)=\boldsymbol{x}^{\top}\boldsymbol{w}+b$

- Solving the normal equations ⇒ $\boldsymbol{w}=\boldsymbol{0}$ and $b=\frac{1}{2}$

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

Insert four $\boldsymbol{x}$ values and corresponding $f^{*}(\boldsymbol{x})$ values to the cost function $J(\theta)$

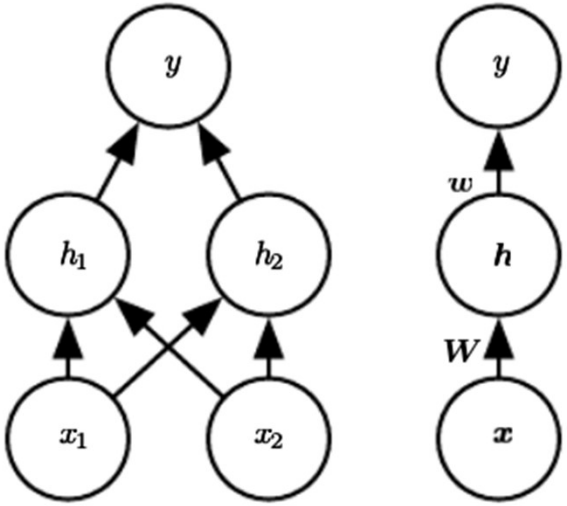
Find $\boldsymbol{w}$ and b from $\frac{\nabla J}{\nabla \boldsymbol{w}}=0$ and $\frac{\nabla J}{\nabla b}=0.$



Linear models cannot find a proper function

Limitation in Linear Model

We choose to use a DFN



$$y = f^{(2)}(\boldsymbol{h};\boldsymbol{w},b) = f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$$

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$$

$$\boldsymbol{h} = g(\boldsymbol{W}^{\top}\boldsymbol{x} + \boldsymbol{c})$$

activation function

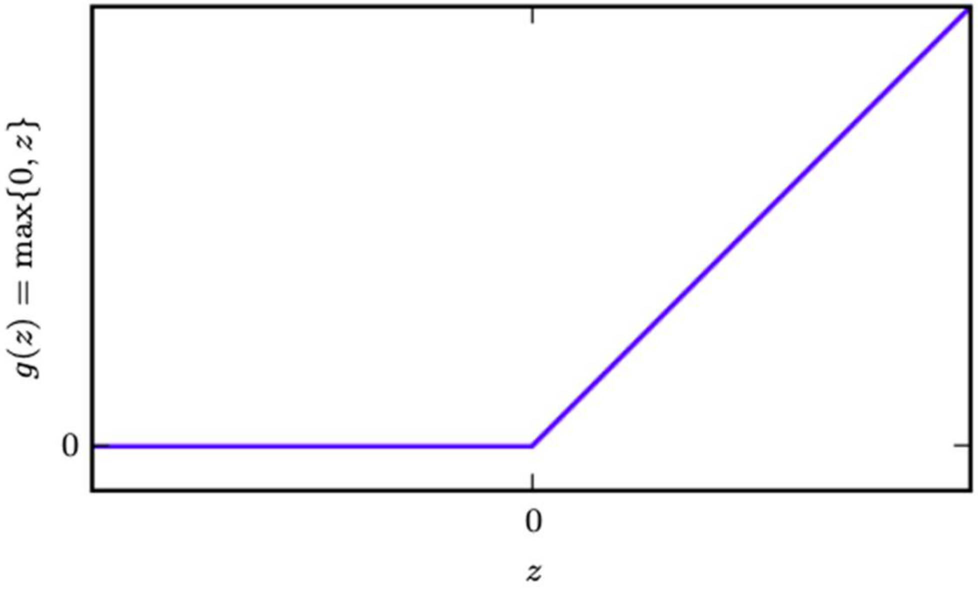
- **Non-linearity** seasoned

- With rectified linear unit (ReLU) activation function: $g(z) = \max\{0, z\}$

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = \boldsymbol{w}^{\top}\max\{0, \boldsymbol{W}^{\top}\boldsymbol{x} + \boldsymbol{c}\} + b.$$

ReLU

- nonlinear transformation, but piecewise linear
- nearly linear ⇒ preserve many properties of linear models ⇒ easy to optimize with gradient-based methods



XOR with Forward Network

- Let the parameters be

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad b = 0$$

- Since

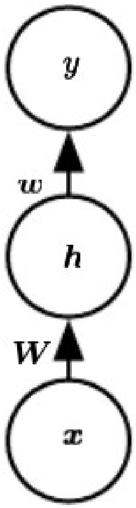
$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

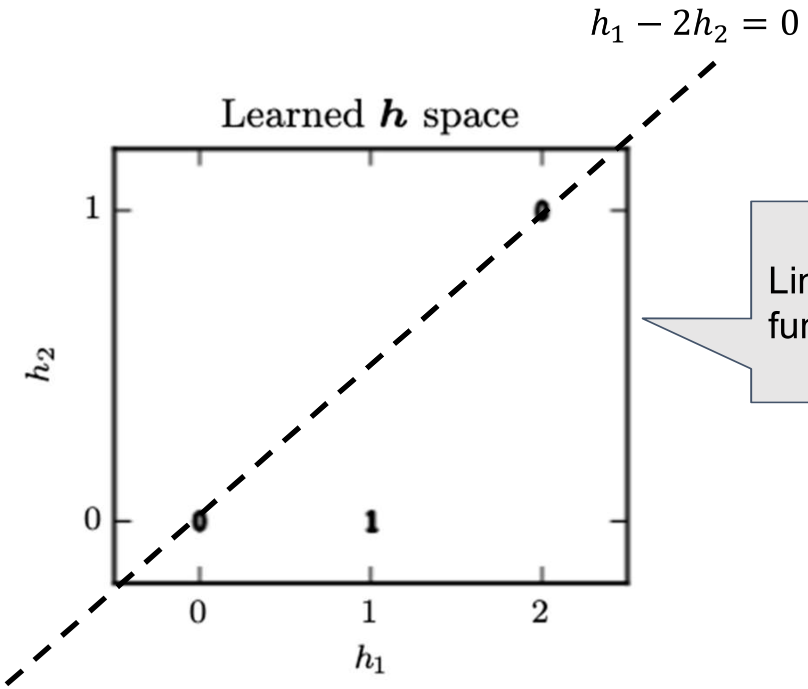
- We get

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \Rightarrow \mathbf{XW}+\mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow g(\mathbf{XW}+\mathbf{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow \mathbf{w}^{\mathsf{T}} g(\mathbf{XW}+\mathbf{c}) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

XOR with Forward Network

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow \boldsymbol{h} = g(\boldsymbol{W}^{\top}\boldsymbol{x} + \boldsymbol{c}) = g(\mathbf{XW}+\mathbf{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$





Linear models now can find a proper function because of the transformation

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

Gradient-Based Learning

- Generally, the gradient-based learning cannot guarantee global optimum
- Step size and batch size should be carefully chosen for convergence
- **Cost functions**
  - ML (=minimizing cross entropy)

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

  - If $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$

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

Mean Square Error

- Consideration factors
  - Gradient of the cost function must be large and predictable enough to serve as a good guidance in learning
  - Cost functions saturate if the activation functions saturate
  - Several output units involve an exp function that can saturate when its argument is very negative
    - ✓ Negative log-likelihood helps to avoid this problem

Gradient-Based Learning

Calculus of variations

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \quad\Longrightarrow\quad f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}]$$

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1 \quad\Longrightarrow$$

a function that predicts the median value of **y** for each **x**

Gradient-Based Learning: Output Unit

Output units

- Linear units $\boxed{\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}}$
  - ➢ mainly used to predict the mean of a conditional Gaussian model
- Signomid unit for Bernoulli output distributions
  - ➢ We want the neural net to predict $\boxed{P(y=1 \mid \boldsymbol{x})}$
    - ✓ Linear model:

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

      - any time that $\mathbf{w}^\top\mathbf{h}+b$ strayed outside the unit interval, the gradient would be **0**.
    - ✓ Note that we have binary outputs: y=0, y=1, and let z (called **logit**) denote $z=\mathbf{w}^\top\mathbf{h}+b$
    - ✓ We want the unnormalized log probability is proportional to yz as

$$\log \tilde{P}(y) = yz, \quad \Rightarrow \quad \tilde{P}(y) = \exp(yz),$$

    - ✓ Then, we have normalized probability

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)}, \quad \Rightarrow \quad \boxed{P(y=1) = \frac{\exp(z)}{1+\exp(z)} = \sigma(z)} \quad \Rightarrow \quad P(y) = \sigma\left((2y-1)z\right)$$

$$P(y=0) = \frac{1}{1+\exp(z)}$$

Gradient-Based Learning: Output Unit

- Output units
  - Signomid unit for Bernoulli output distributions
    - ✓ Thus, we choose the output of the neural net as

$$\boxed{\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right)}$$

  - Design tip
    - ✓ It is better to choose the maximum likelihood as a cost function, since the log in the cost undoes the exp of the sigmoid
      - Otherwise (e.g., with MSE cost function), the saturation of the sigmoid could prevent gradient-based learning
    - ✓ Example
      - Preliminary: Softplus function

$$\boxed{\zeta(x) = \log\left(1 + \exp(x)\right)}$$

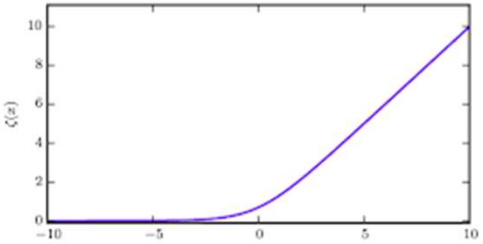


Figure 3.4: The softplus function.

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$1 - \sigma(x) = \sigma(-x)$$

$$\log \sigma(x) = -\zeta(-x)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x)$$

$$\forall x \in (0,1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right)$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log\left(\exp(x) - 1\right)$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy$$

$$\zeta(x) - \zeta(-x) = x$$

Gradient-Based Learning: Output Unit

- Output units
  - Design tip (cont'd)
    - ✓ Loss function for ML learning

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log P(y \mid \boldsymbol{x}) \\ &= -\log \sigma\left((2y-1)z\right) \\ &= \zeta\left((1-2y)z\right). \end{aligned}$$

    - ✓ It saturates only when (1-2*y*)*z* is very negative
    - ✓ It saturates when the model already has the right answer: (y=1 and z is very positive) or (y=0 and z is very negative)

Gradient-Based Learning: Output Unit

- Softmax Units for Multinoulli Output Distributions
  - Generalization from the binary case with the sigmoid unit to the case with *n* variables
  - We define the neural net output for the i-th variable

$$\hat{y}_i = P(y = i \mid \boldsymbol{x})$$

  - Unnormalized log probabilities: $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$ $\boxed{\text{where } z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})}$
  - Softmax function is defined by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

  - For ML maximization, we use the fact that

$$\log P(\mathrm{y} = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i = z_i - \boxed{\log \sum_j \exp(z_j)}$$

z_i has a directi contribution to the cost function, which cannot saturate

Approximated by max z_j (since exp(z_k) is insignificant for any z_k that is noticeably less than max z_j.

This term $\approx \log \max\left(\exp(z_j)\right)$ $= \max(z_j)$

Cost function penalizes the most active incorrect prediction

If the correct answer already has the largest input, then the cost function becomes zero

Hidden Units

Some Observations

- Design of hidden units: still an active area of research
- Predicting in advance which hidden units will work best is usually impossible
- Many hidden units are not actually differentiable at all input points
  - e.g., max{0,z}
  - However, it is very unlikely that the value during the training is truly 0
  - It is safe to disregard the non-differentiability of the hidden units

ReLU: $g(z)$ = max{0,z}

- almost **linear**
- derivatives are 1 everywhere that the unit is active ⇒ useful for learning with gradient-descent algorithms
- ReLUs are typically used on top of an affine transformation: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + b)$
- typically, b is chosen as small values such as 0.1
- drawback: cannot learn via gradient-based methods if their activation is zero

Generalized ReLU:

- **absolute value rectification**: α_i=-1 to obtain g(z)=|z|

$$\boxed{h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)}$$

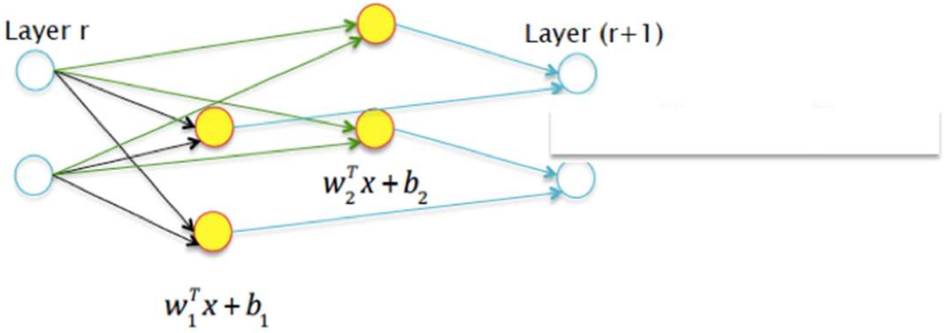
Hidden Units

Generalized ReLU

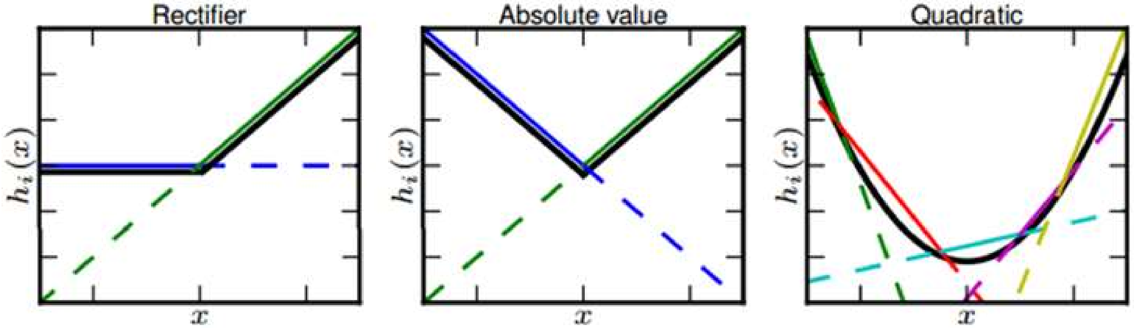
- **Maxout**

$$\boxed{g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j}$$

where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i



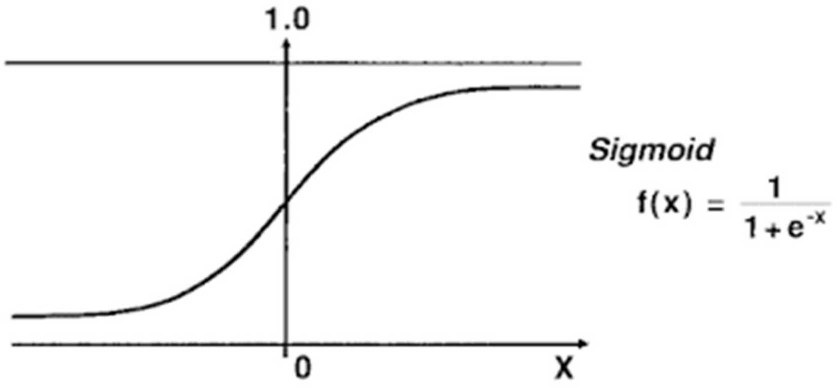
$$z'_i = \max(a_i(1), a_i(2)) = \max\left(\sum_j w_{ij}(1) z_j + b_i(1), \sum_j w_{ij}(2) z_j + b_i(2)\right)$$



Hidden Units

Logistic Sigmoid $g(z) = \sigma(z)$

- saturates across most of their domain
- appropriate cost function should undo the saturation

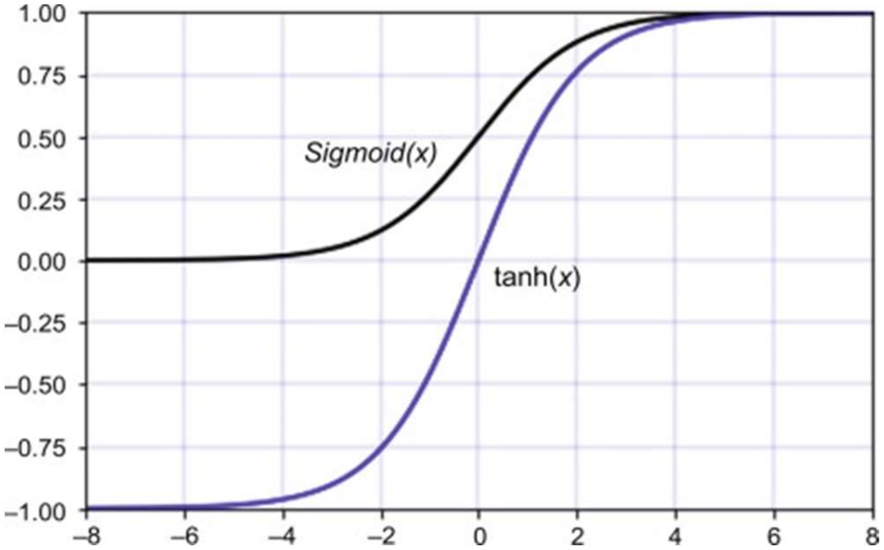


Hidden Units

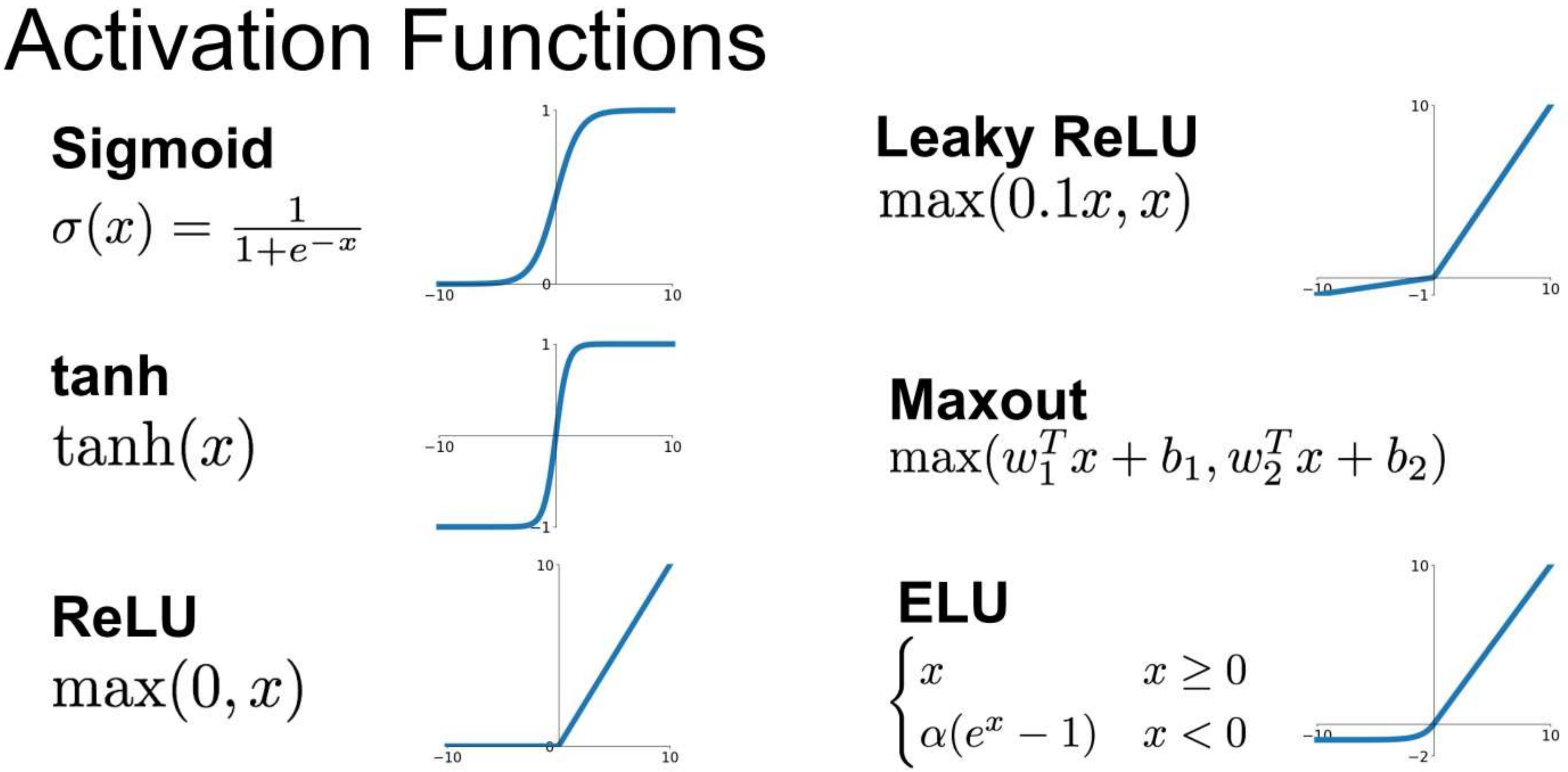
Hyperbolic Tangent $g(z) = \tanh(z)$

- generally performs better than the sigmoid
- It is nearly linear when its argument is small
  - ➢ $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$ becomes close to $\hat{y} = \boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$
  - ➢ becomes easy to train

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{\frac{e^x - e^{-x}}{2}}{\frac{e^x + e^{-x}}{2}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



Comparison of Activation Functions



Activation Functions

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$

Leaky ReLU

$\max(0.1x, x)$

Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Architecture Design

Architecture: overall structure of the network

- **how many** units? how these units should be **connected** to each other?

In **chain-based** architectures

- Main consideration is choosing the **depth** of the network and the **width** of each layer
- architecture design requires **trial and error**

Universal approximation Theorem

- Feedforward network with a **linear output** unit and at least **one hidden layer** with any squishing activation function (such as the logistic sigmoid activation function) can approximate **any** Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given **enough hidden units**
  - *any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable

One step further

- the theorem has been proved also for a wider class of activation functions such as **rectified linear** unit

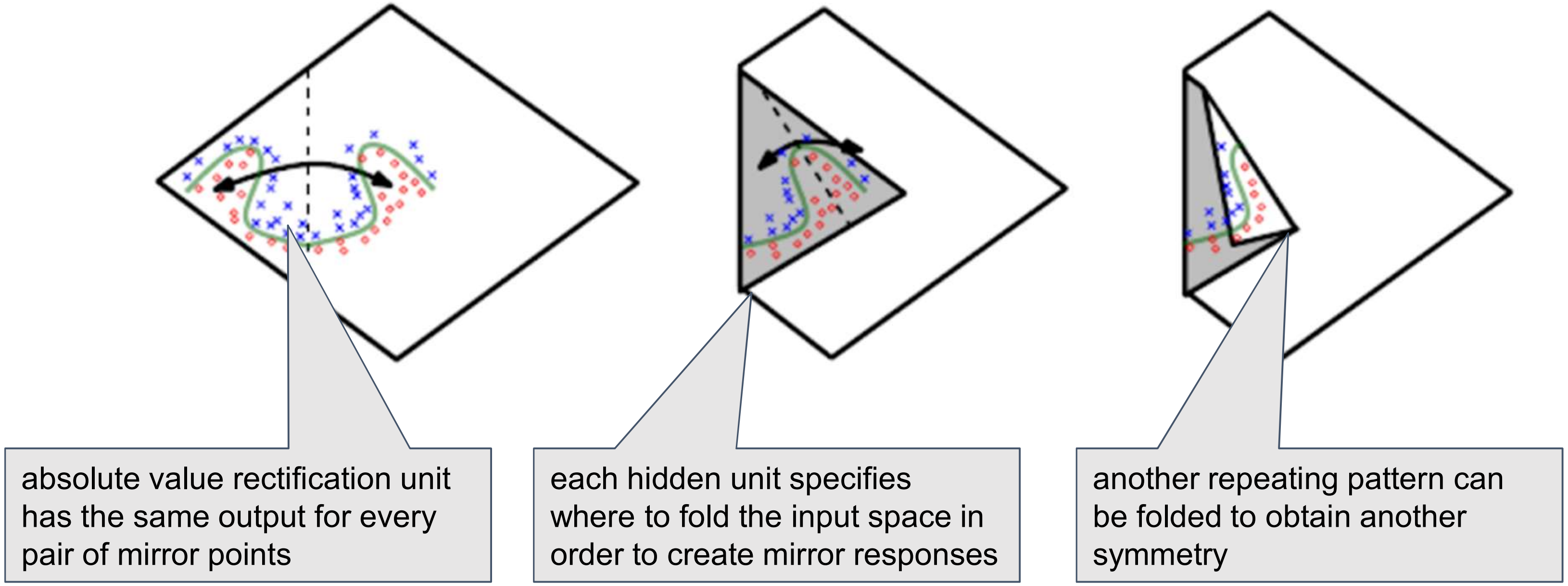
Architecture Design

The theorem does not guarantee the training algorithm can actually learn the functions

- optimization algorithm used for training may not be able to find the proper values
- training algorithm might choose wrong function as a result of overfitting

Then, how large the network should be?

- in the worst case, an exponential (corresponding to each input size) number of hidden units is needed



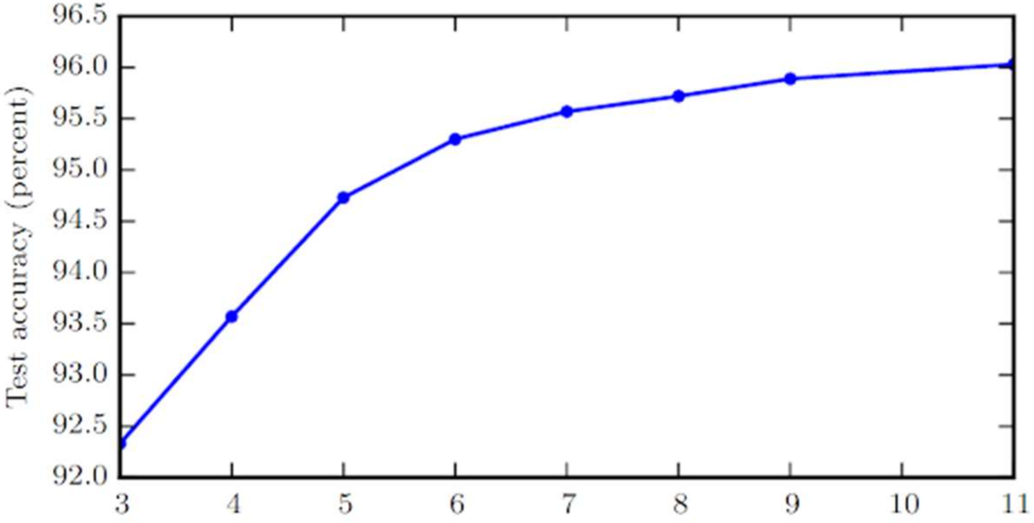
absolute value rectification unit has the same output for every pair of mirror points

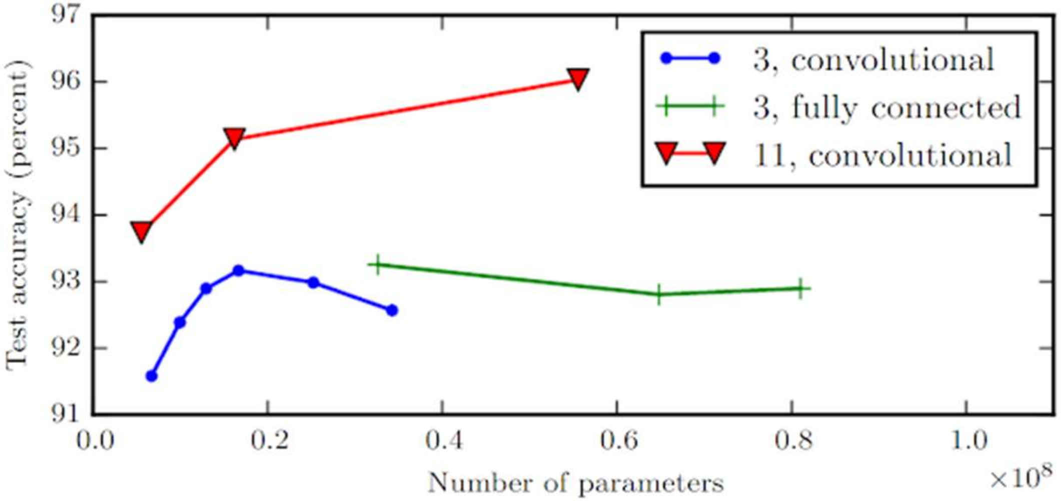
each hidden unit specifies where to fold the input space in order to create mirror responses

another repeating pattern can be folded to obtain another symmetry

Architecture Design

In general, deeper models are better

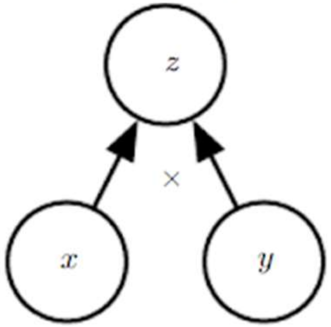




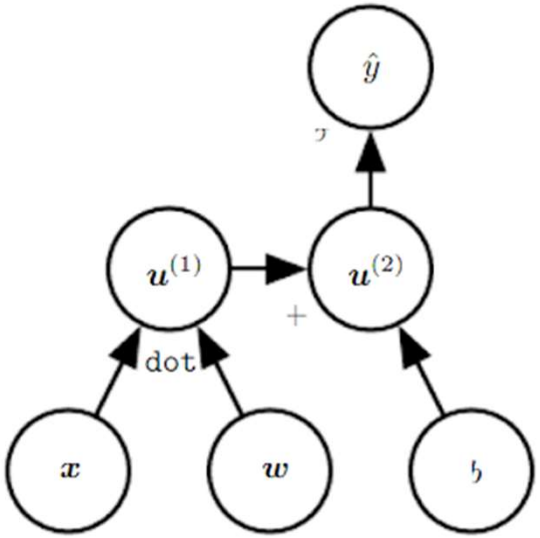
Back-Propagation

Computational graph

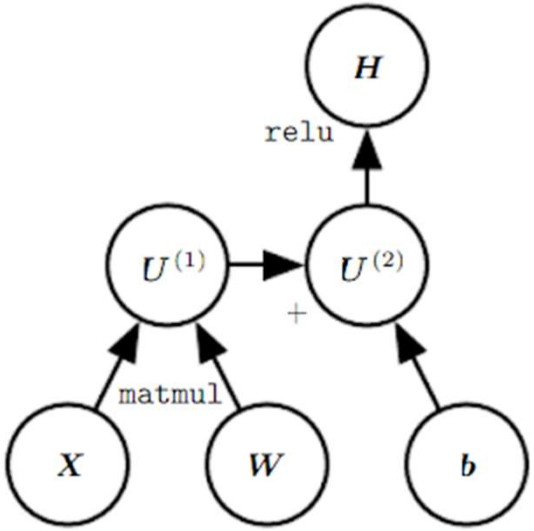
$$z = xy$$



$$\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$$



$$\boldsymbol{H} = \max\{0, \boldsymbol{XW} + \boldsymbol{b}\}$$



Back-Propagation

Chain rule

- For $y = g(x)$ and $z = f(g(x)) = f(y)$
- we have
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- For $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ where $\boldsymbol{x} \in \mathbb{R}^m,\ \boldsymbol{y} \in \mathbb{R}^n$
  g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$

- we have
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

$$\Downarrow$$

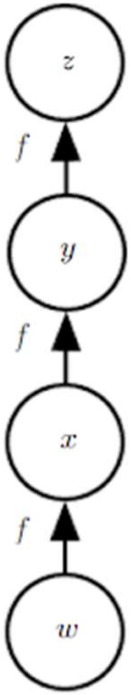
$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z$$

$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ $\boxed{\text{Jacobian matrix}}$ of g

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Back-Propagation

Two different implementations



computational graph for

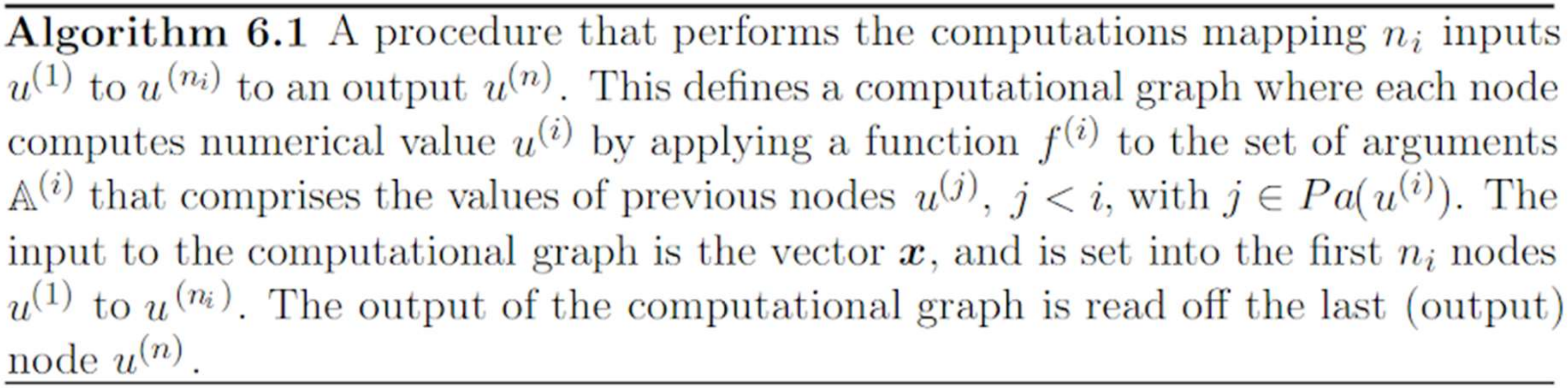
$x = f(w),\ y = f(x),\ z = f(y)$

$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
&= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
&= f'(y)f'(x)f'(w) \\
&= f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

Fast but requires more memory

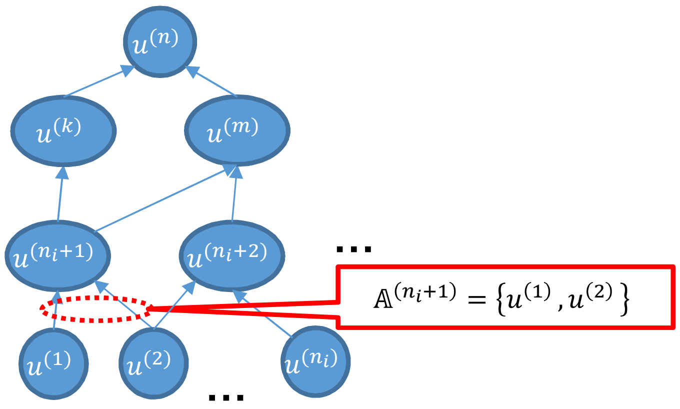
Slow but requires less memory

Back-Propagation: Scalar Node Example – Forward Computation



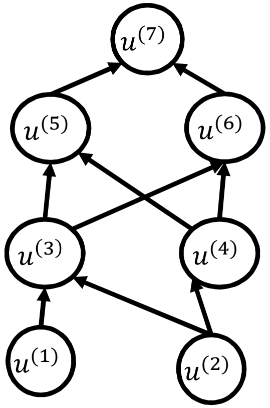
Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

```
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 $u^{(n)}$
```



$\mathbb{A}^{(n_i+1)} = \{u^{(1)}, u^{(2)}\}$

Back-Propagation: Scalar Node Example – Backward Computation



$$\frac{\partial u^{(7)}}{\partial u^{(6)}} = \sum_{i:6\in Pa\left(u^{(i)}\right)} \frac{\partial u^{(7)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(6)}} = \frac{\partial u^{(7)}}{\partial u^{(6)}}$$

The derivative of this direct-input-output function is not that difficult to compute. For instance, for

$$u^{(7)} = \sigma\left(w_1 u^{(5)} + w_2 u^{(6)} + b\right)$$

We get

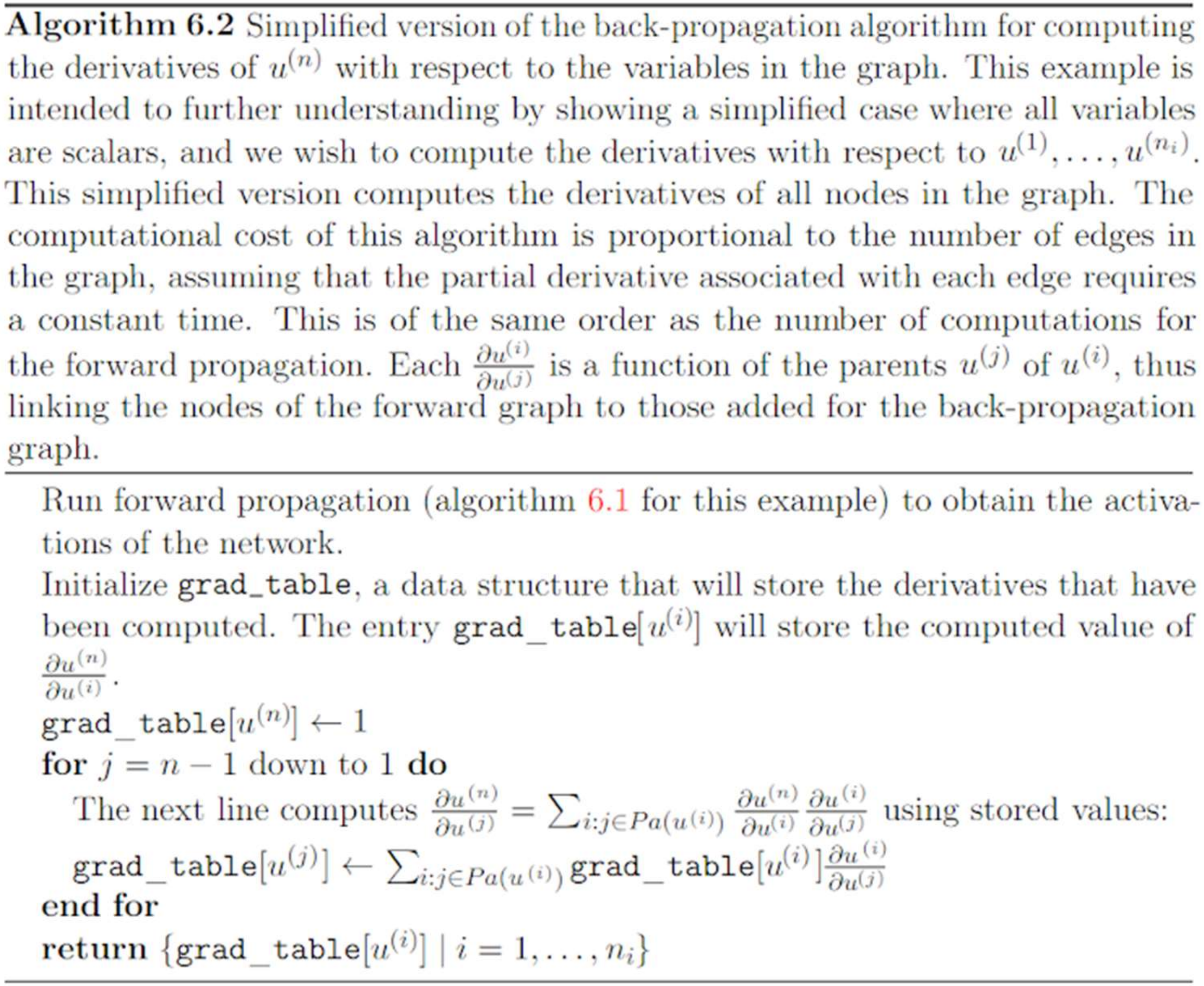
$$\begin{aligned}\frac{\partial u^{(7)}}{\partial u^{(5)}} &= \sigma'\left(w_1 u^{(5)} + w_2 u^{(6)} + b\right) \frac{\partial\left(w_1 u^{(5)} + w_2 u^{(6)} + b\right)}{\partial u^{(5)}} \\ &= \sigma\left(w_1 u^{(5)} + w_2 u^{(6)} + b\right)\sigma\left(1 - w_1 u^{(5)} - w_2 u^{(6)} - b\right) w_1\end{aligned}$$

$$\frac{\partial u^{(7)}}{\partial u^{(5)}} = \sum_{i:5\in Pa\left(u^{(i)}\right)} \frac{\partial u^{(7)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(6)}} = \frac{\partial u^{(7)}}{\partial u^{(5)}}$$

$$\frac{\partial u^{(7)}}{\partial u^{(4)}} = \sum_{i:4\in Pa\left(u^{(i)}\right)} \frac{\partial u^{(7)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(4)}} = \frac{\partial u^{(7)}}{\partial u^{(5)}} \cdot \frac{\partial u^{(5)}}{\partial u^{(4)}} + \frac{\partial u^{(7)}}{\partial u^{(6)}} \cdot \frac{\partial u^{(6)}}{\partial u^{(4)}}$$

Back-Propagation: Scalar Node Example – Backward Computation

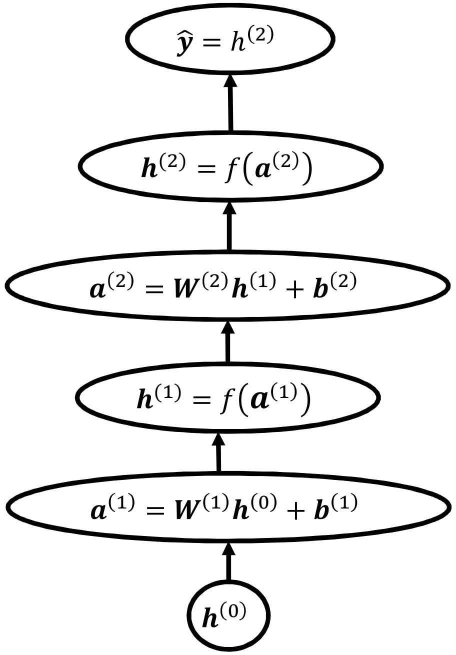
General algorithm



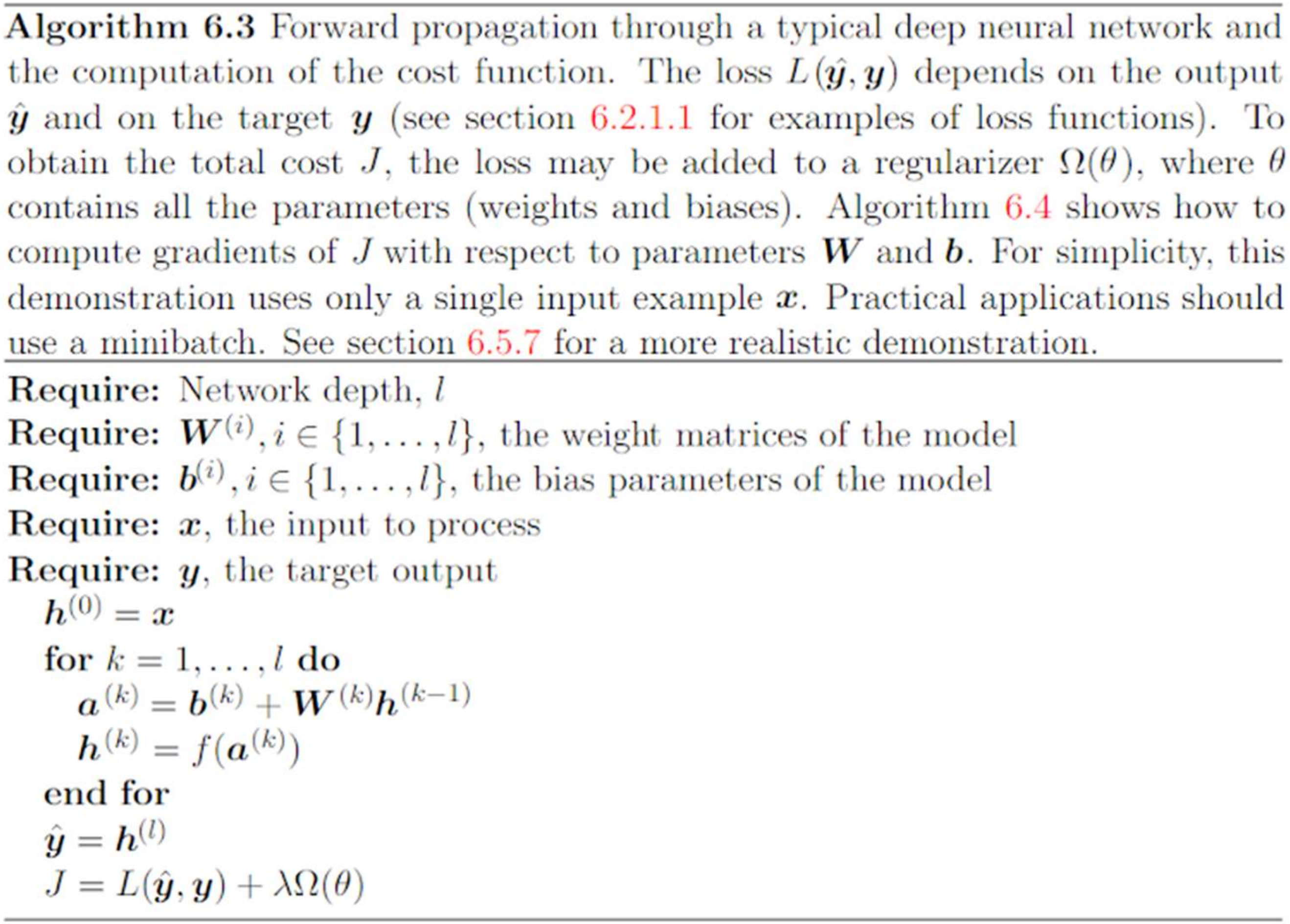
Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

```
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[$u^{(i)}$] $\mid i = 1, \ldots, n_i$}
```

Back-Propagation: Fully Connected MLP



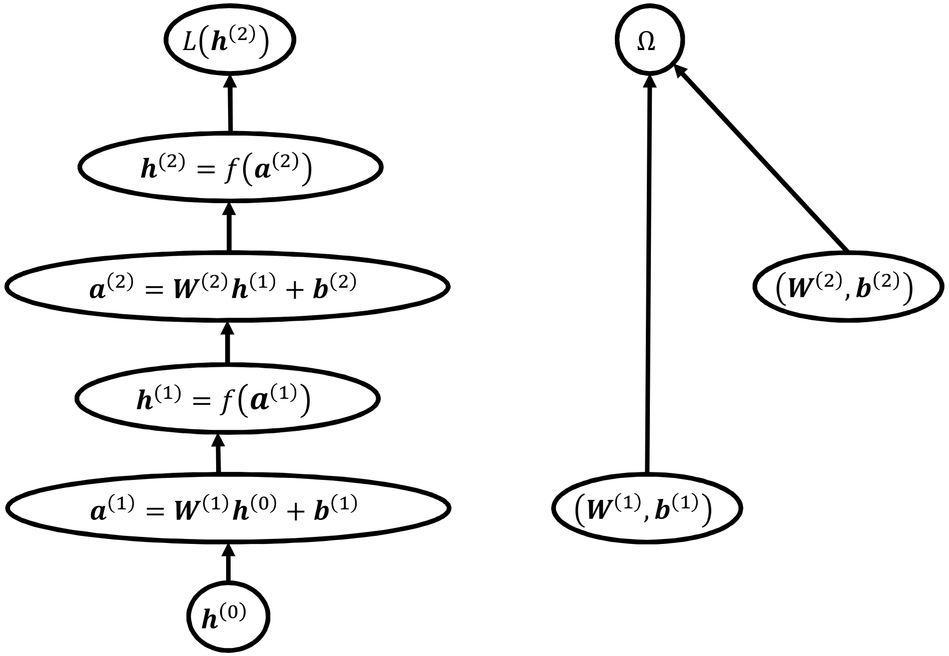
$\hat{y} = h^{(2)}$
$\boldsymbol{h}^{(2)} = f(\boldsymbol{a}^{(2)})$
$\boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}$
$\boldsymbol{h}^{(1)} = f(\boldsymbol{a}^{(1)})$
$\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{h}^{(0)} + \boldsymbol{b}^{(1)}$
$\boldsymbol{h}^{(0)}$



Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

```
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: Fully Connected MLP



$L(\boldsymbol{h}^{(2)})$
$\boldsymbol{h}^{(2)} = f(\boldsymbol{a}^{(2)})$
$\boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}$
$\boldsymbol{h}^{(1)} = f(\boldsymbol{a}^{(1)})$
$\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{h}^{(0)} + \boldsymbol{b}^{(1)}$
$\boldsymbol{h}^{(0)}$
Ω
$(\boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)})$
$(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)})$

We start with computing

$$\nabla_{\boldsymbol{h}^{(2)}} J = \nabla_{\boldsymbol{h}^{(2)}} \left(L(\boldsymbol{h}^{(2)}) + \lambda\Omega(\theta) \right) = \nabla_{\boldsymbol{h}^{(2)}} L(\boldsymbol{h}^{(2)})$$

For instance, if $L = \left\| \boldsymbol{h}^{(2)} - \boldsymbol{y} \right\|^2$, we get

$$\nabla_{\boldsymbol{h}^{(2)}} L(\boldsymbol{h}^{(2)}) = 2\boldsymbol{h}^{(2)} - 2\boldsymbol{y}$$

We compute then $\nabla_{\boldsymbol{a}^{(2)}} J = \nabla_{\boldsymbol{a}^{(2)}} L$

Denoting the i-th element of $\boldsymbol{a}^{(2)}$ and $\boldsymbol{h}^{(2)}$ by $a_{2,i}$ and $h_{2,i}$, we get

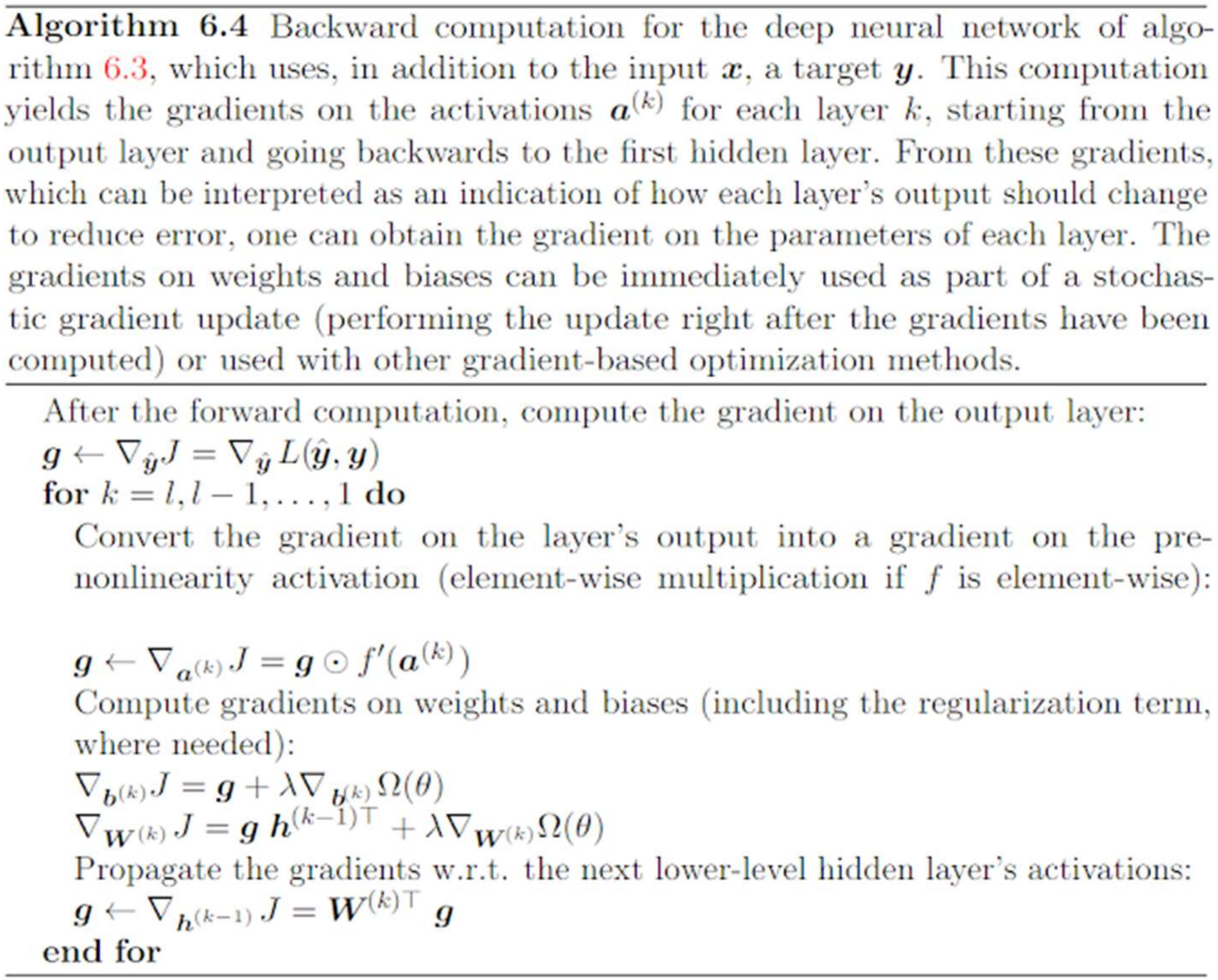
$$\frac{\partial L}{\partial a_{2,i}} = \frac{\partial L}{\partial h_{2,i}} \cdot \frac{\partial h_{2,i}}{\partial a_{2,i}} = \frac{\partial L}{\partial h_{2,i}} \cdot f'(a_{2,i})$$

Thus, augmenting all the elements, we get

$$\nabla_{\boldsymbol{a}^{(2)}} L = \nabla_{\boldsymbol{h}^{(2)}} L \odot f'(\boldsymbol{a}^{(2)})$$

$$\begin{aligned} \nabla_{\boldsymbol{b}^{(2)}} J &= \nabla_{\boldsymbol{b}^{(2)}} (L + \lambda\Omega(\theta)) \\ &= \nabla_{\boldsymbol{h}^{(2)}} L \odot f'(\boldsymbol{a}^{(2)}) \cdot \frac{\partial(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)})}{\partial \boldsymbol{b}^{(2)}} + \lambda\nabla_{\boldsymbol{b}^{(2)}} \Omega(\theta) \\ &= \nabla_{\boldsymbol{h}^{(2)}} L \odot f'(\boldsymbol{a}^{(2)}) + \lambda\nabla_{\boldsymbol{b}^{(2)}} \Omega(\theta) \end{aligned}$$

Back-Propagation: Fully Connected MLP



Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

```
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
```