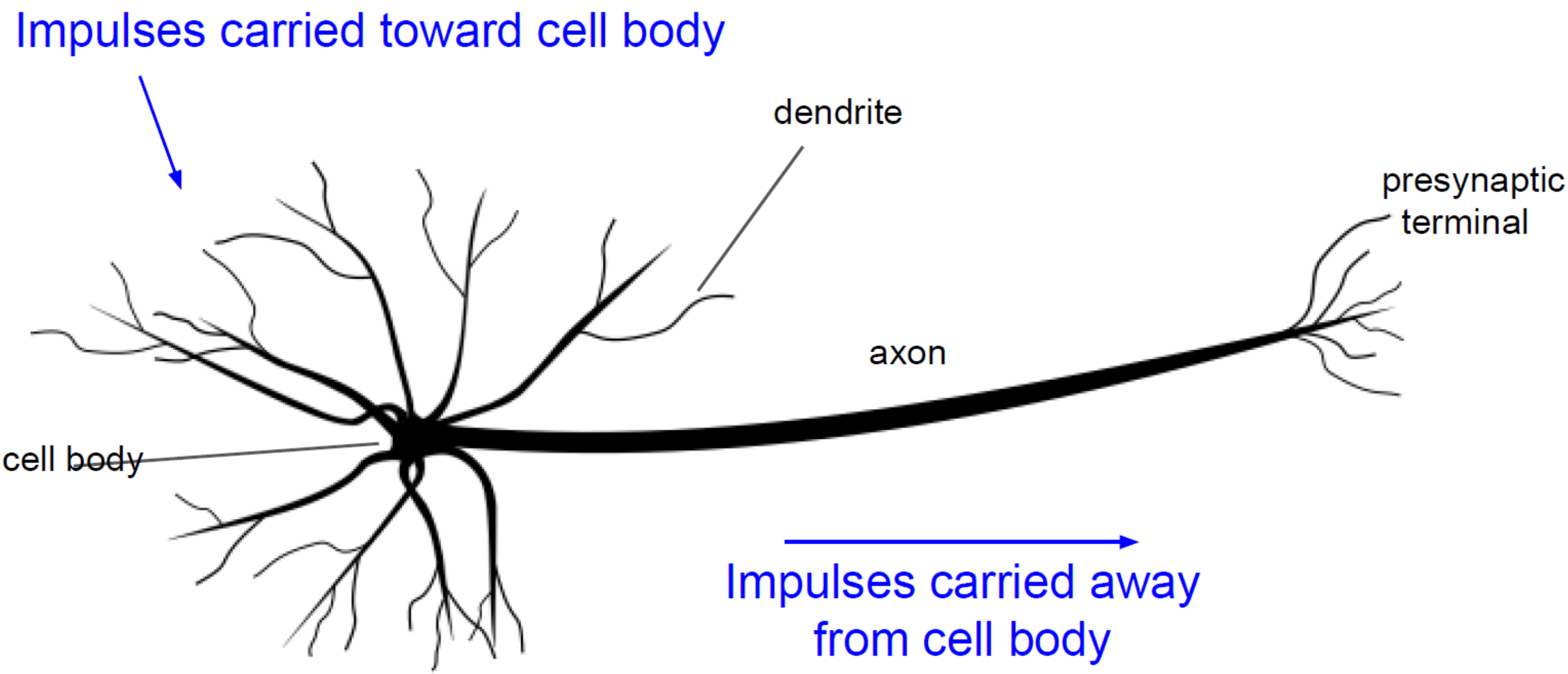
Neural networks

CE-477: Machine Learning - CS-828: Theory of Machine Learning
Sharif University of Technology
Fall 2024

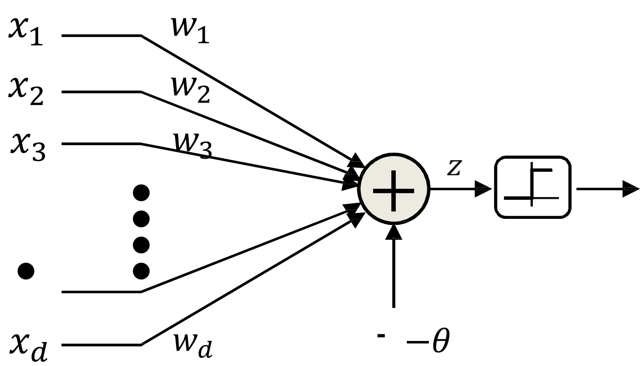
Fatemeh Seyyedsalehi

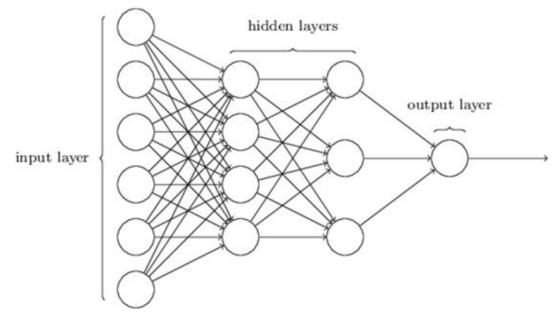
Neural cell

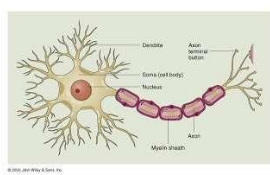


Impulses carried toward cell body
dendrite
presynaptic terminal
axon
cell body
Impulses carried away from cell body

Neural nets and the brain



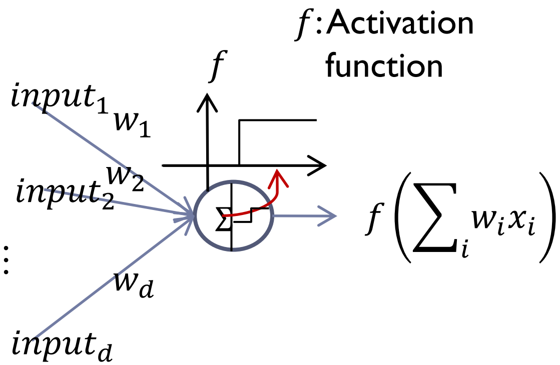


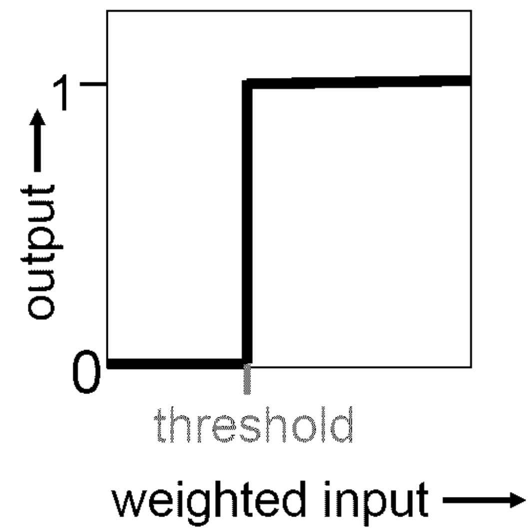


- Neural nets are composed of networks of computational models of neurons called perceptrons

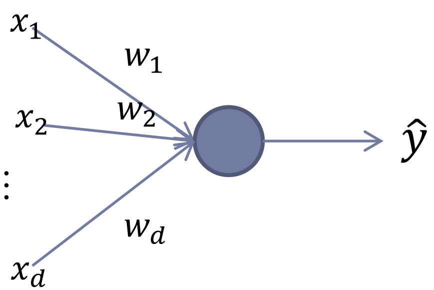
Binary threshold neurons

- McCulloch-Pitts (1943): influenced Von Neumann.
  - First compute a weighted sum of the inputs.
  - Send out a spike of activity if the weighted sum exceeds a threshold.

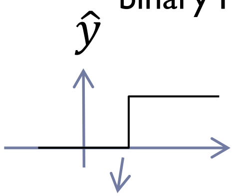




McCulloch-Pitts neuron: binary threshold



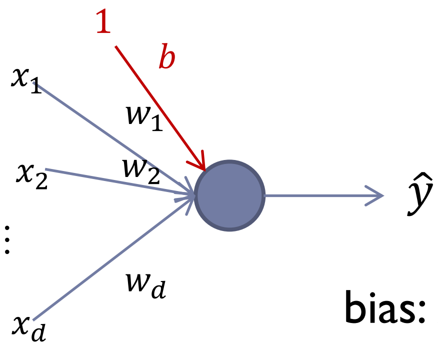
binary McCulloch-Pitts neuron

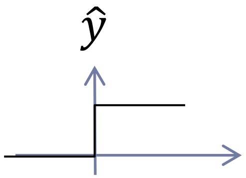


θ: activation threshold

$$\hat{y} = \begin{cases} 1, & z \geq \theta \\ 0, & z < \theta \end{cases}$$

Equivalent to



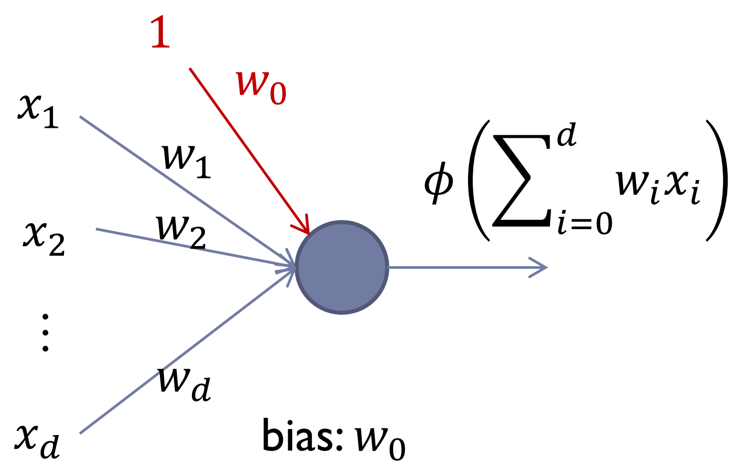


bias: $b = -\theta$

Single neuron

- Single neuron can be used as a linear decision boundary:
  - $\phi(\boldsymbol{w}^T\boldsymbol{x})$ shows the class of $\boldsymbol{x}$

- Training methods of a single neuron:
  - Perceptron (Rosenblatt, 1962)
    - We have seen it before!
  - ADALINE (Widrow and Hoff, 1960)
    - We have seen it before too !!!
  - …



Reminder: Perceptron Learning Algorithm

- Given N training instances $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(N)}, y^{(N)})$
  - $y^{(n)} = +1$ or -1

- Initialize $\boldsymbol{w}$
- Cycle through the training instance
- While more classification errors
  - For $i = 1 \ldots N_{train}$

    $$\hat{y}^{(i)} = sign(\boldsymbol{w}^T \boldsymbol{x}^{(i)})$$

    - If $\hat{y}^{(i)} \neq y^{(i)}$

      $$\boldsymbol{w} = \boldsymbol{w} + y^{(i)} \boldsymbol{x}^{(i)}$$

- If instance misclassified:
  - If instance is positive class

    $$\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{x}$$

  - If instance is negative class

    $$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}$$

Training of a single neuron

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \eta \nabla J_n(\boldsymbol{w}^{t})$$

- Weight update for a training pair $(\boldsymbol{x}^{(n)}, y^{(n)})$:
  - Perceptron: If $\text{sign}(\boldsymbol{w}^T \boldsymbol{x}^{(n)}) \neq y^{(n)}$ then

$$\nabla J_n(\boldsymbol{w}^{t}) = -\eta \boldsymbol{x}^{(n)} y^{(n)}$$

$J_n(\boldsymbol{w}) = -\boldsymbol{w}^T \boldsymbol{x}^{(n)} y^{(n)}$
if misclassified

  - ADALINE: $\nabla J_n(\boldsymbol{w}^{t}) = -\eta (y^{(n)} - \boldsymbol{w}^T \boldsymbol{x}^{(n)}) \boldsymbol{x}^{(n)}$
    - Widrow-Hoff, LMS, or delta rule

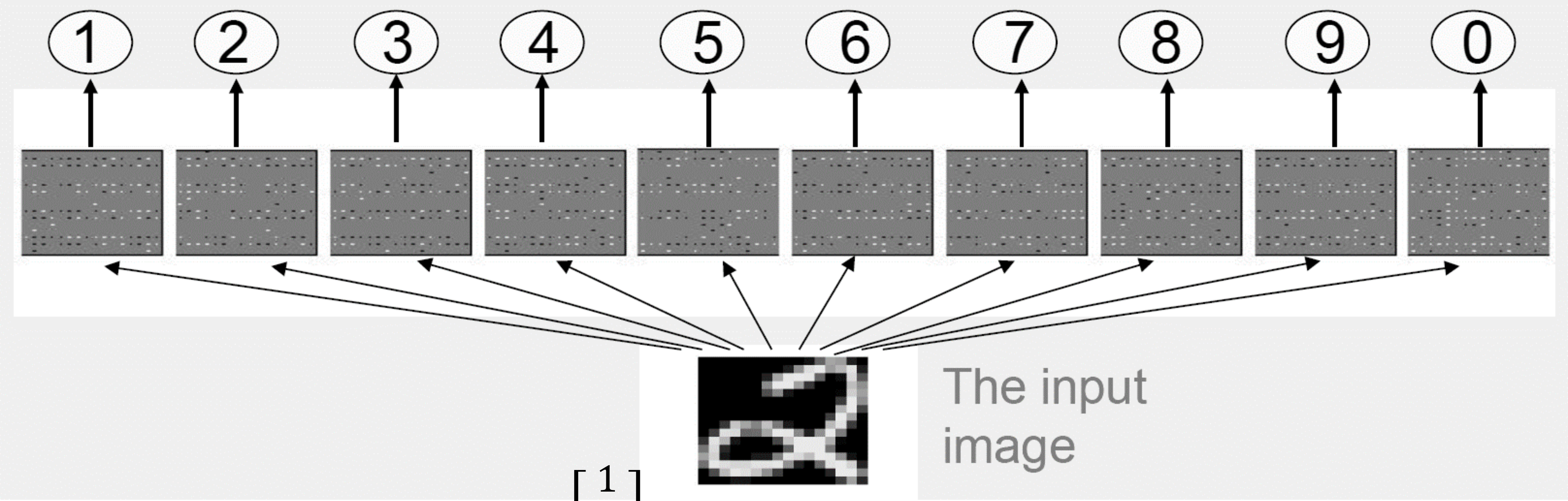
$J_n(\boldsymbol{w}) = \left(y^{(n)} - \boldsymbol{w}^T \boldsymbol{x}^{(n)}\right)^2$

Perceptron vs. Delta Rule

- Perceptron learning rule:
  - Guaranteed to succeed if training examples are linearly separable
- Delta rule:
  - Guaranteed to converge to the hypothesis with the minimum squared error
  - Succeed if sufficiently small learning rate
    - Even when training data contain noise or are not separable by a hyperplane
  - Can also be used for regression problems

How to learn the weights: multi class example

- 10 neuron next to each other to compose a single layer neural network

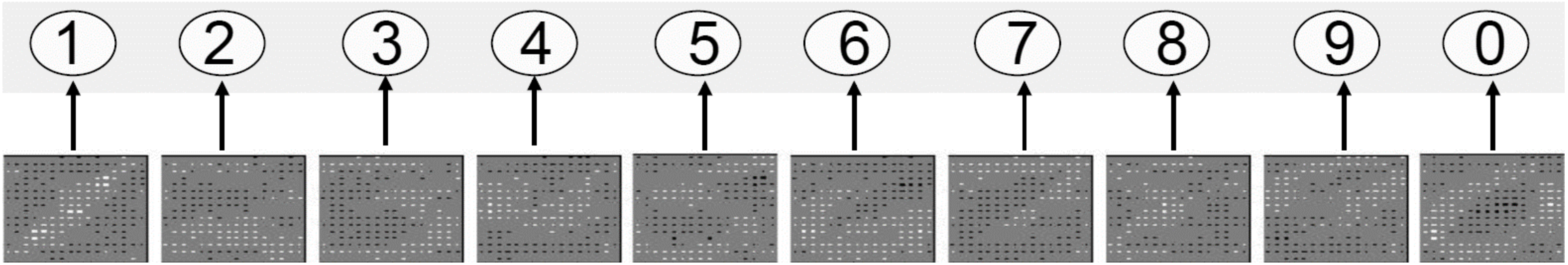


$$W = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1d} \\ \vdots & \ddots & \cdots & \vdots \\ b_k & w_{k1} & \cdots & w_{kd} \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

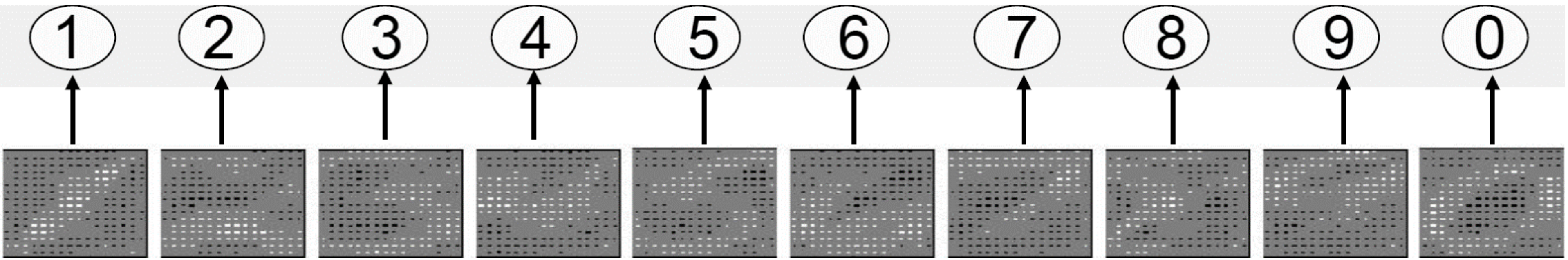
- Change of weights during training



This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

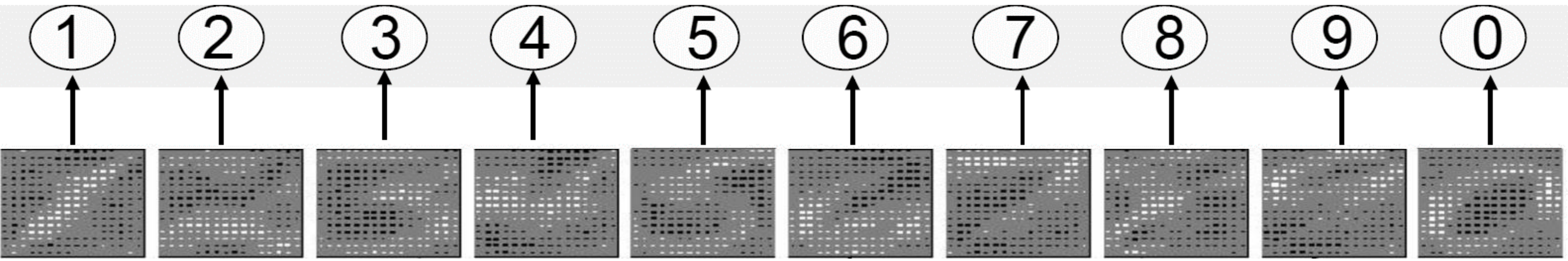
- Change of weights during training



 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

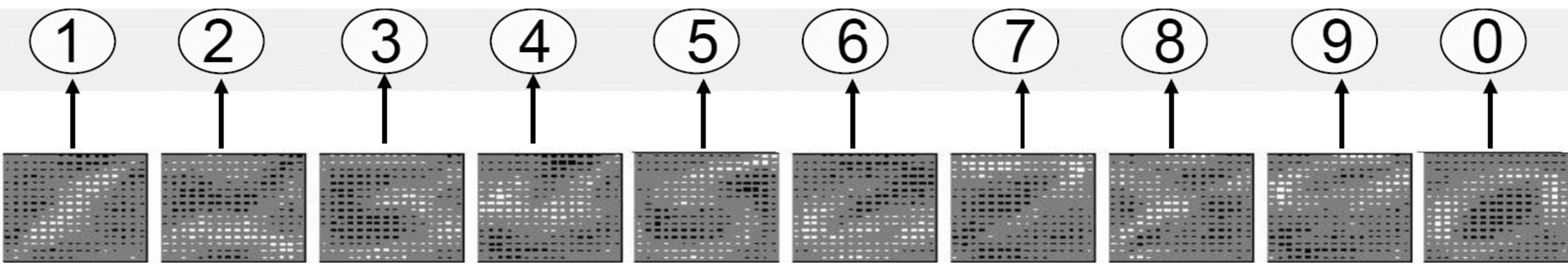
- Change of weights during training



 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

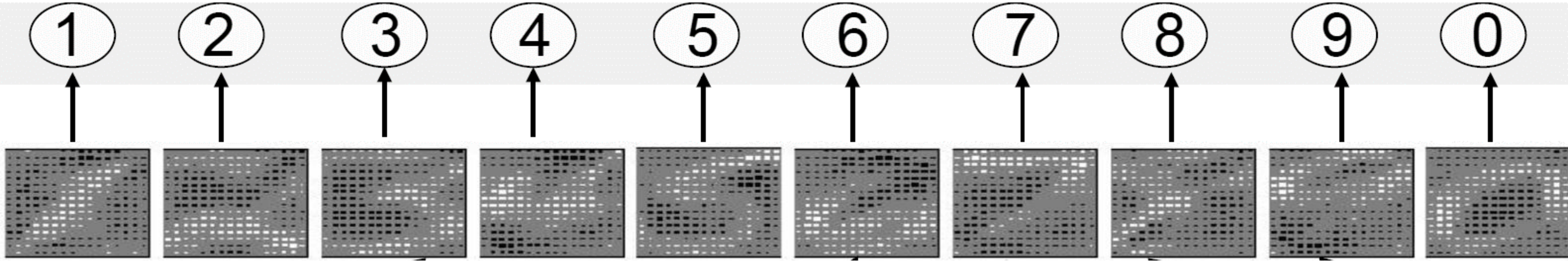
- Change of weights during training



 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

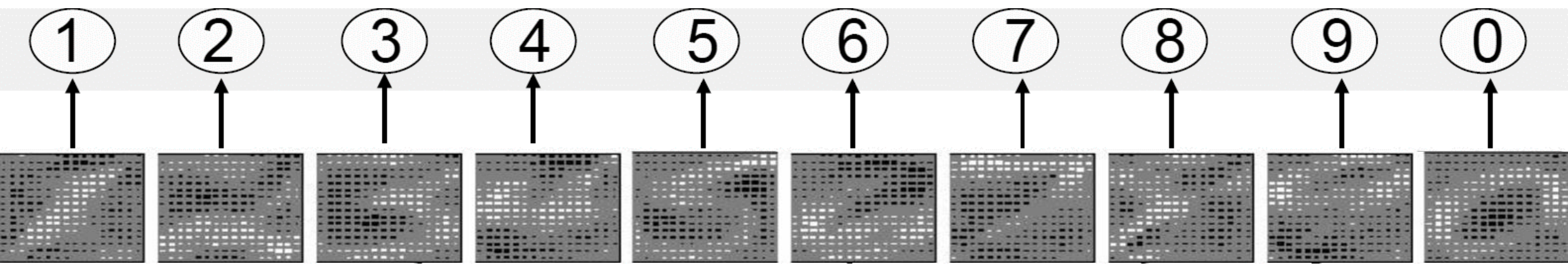
- Change of weights during training



 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

How to learn the weights: multi class example

- Change of weights during training



 This slide has been adopted from Hinton's lectures, "NN for Machine Learning" course, 2015.

Limitation of single layer network

- Single layer networks is equivalent to template matching
  - Weights for each class as a template for that class.

- The ways in which a digit can be written are much too complicated to be captured by simple template

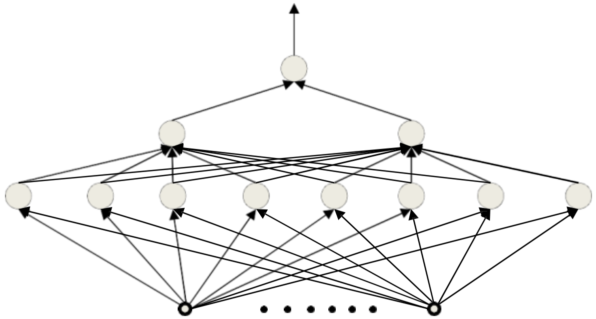
- Thus, networks without hidden units are very limited in the mappings that they can learn

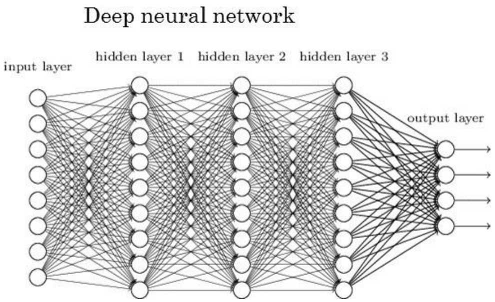
Networks with hidden units

- Networks without hidden units are very limited in the input-output mappings they can learn to model.
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough.

- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?

The multi-layer neural network

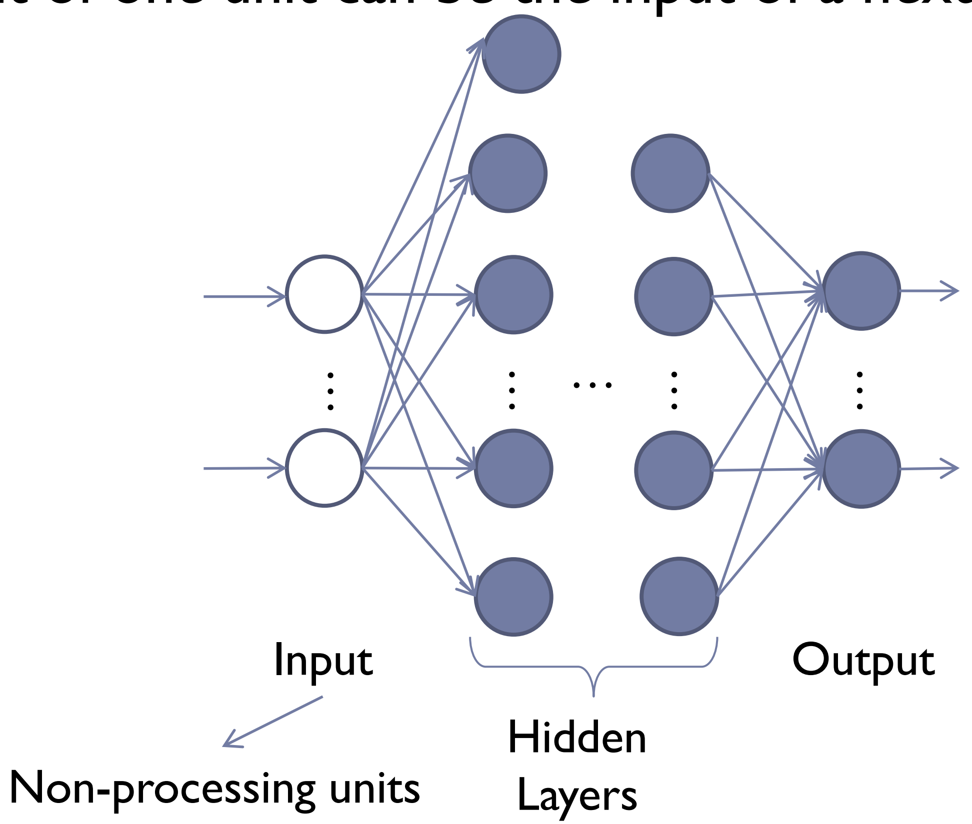




- A network of perceptrons
  - Generally “layered”
  - Also called multi layer perceptron

Feed-forward neural networks

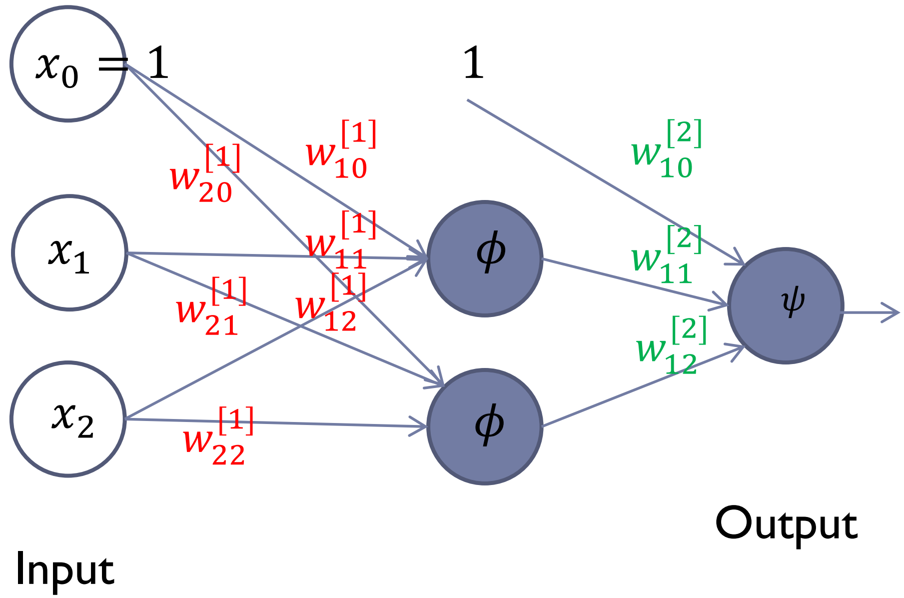
- We need multiple layers of adaptive, non-linear hidden units.
  - Also called **Multi-Layer Perceptron (MLP)**
- Each *unit* takes some inputs and produces one output.
  - Output of one unit can be the input of a next layer(s) unit.



Weights on links can be adapted using training data and a learning algorithm

Multi-layer Neural Network

- Φ: activation function
  - A nonlinear function



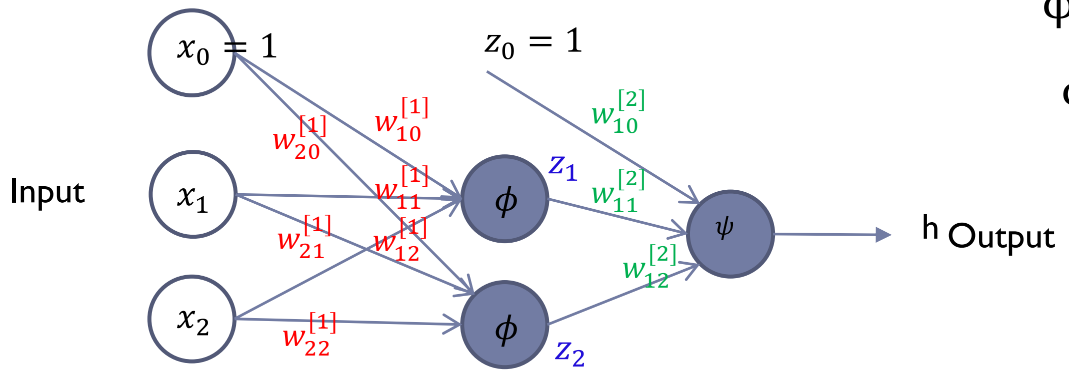
Multi-layer Neural Network

ϕ: fixed activation function

Examples:

$$\phi(z) = \max(0, z)$$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

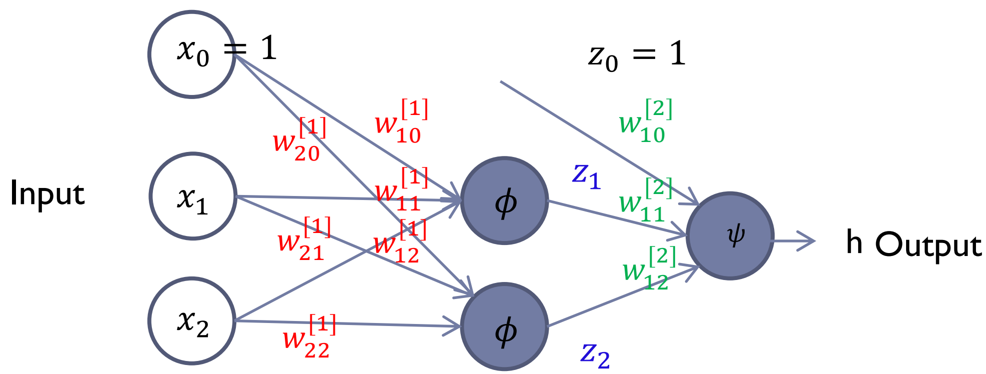


$$z_j = \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)$$

$$h = \psi\left(\sum_{j=0}^{2} w_{kj}^{[2]} z_j\right) = \left(\sum_{j=0}^{2} w_{kj}^{[2]} \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)\right)$$

Multi-layer Neural Network

$$W^{[1]} = \begin{bmatrix} w_{10}^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} \\ w_{20}^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix}$$



ϕ: fixed activation function

Examples:

$$\phi(z) = \max(0, z)$$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

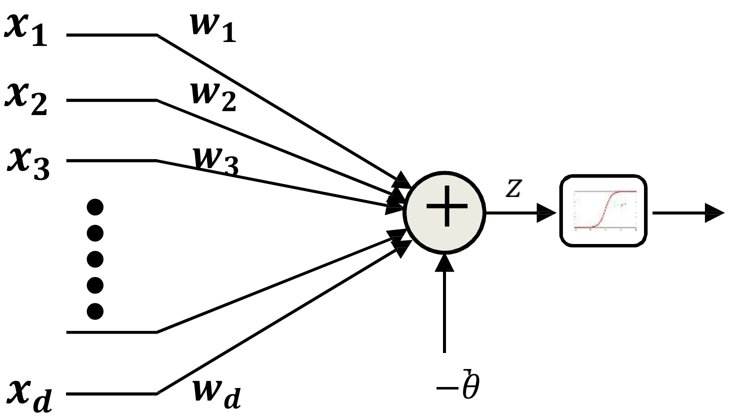
$$z_j = \phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)$$

$$\text{h} = \psi\left(\Sigma_{j=0}^{2} w_{kj}^{[2]} z_j\right) = \left(\Sigma_{j=0}^{2} w_{kj}^{[2]} \phi\left(\Sigma_{i=0}^{d} w_{ji}^{[1]} x_i\right)\right)$$

Matrix form: $\boldsymbol{z} = \phi\left(W^{[1]}\boldsymbol{x}\right)$

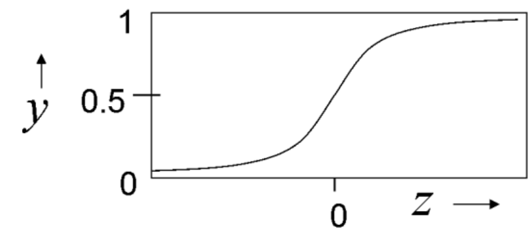
$$\text{h}(\boldsymbol{x}) = \psi(\boldsymbol{W}^{[2]}\boldsymbol{z}) = \psi\left(\boldsymbol{W}^{[2]}\phi(\boldsymbol{W}^{[1]}\boldsymbol{x})\right)$$

Sigmoid



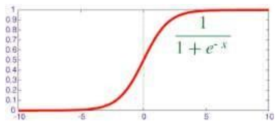
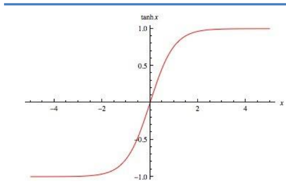
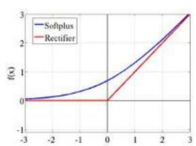
$$z = \sum_i w_i x_i - \theta$$

$$\hat{y} = \frac{1}{1 + \exp(-z)}$$



- A “squashing” function instead of a threshold
  - The sigmoid “activation” replaces the threshold
    - These give a real-valued output that is a smooth and bounded function of their input.
    - They have nice derivatives.

Activations and their derivatives

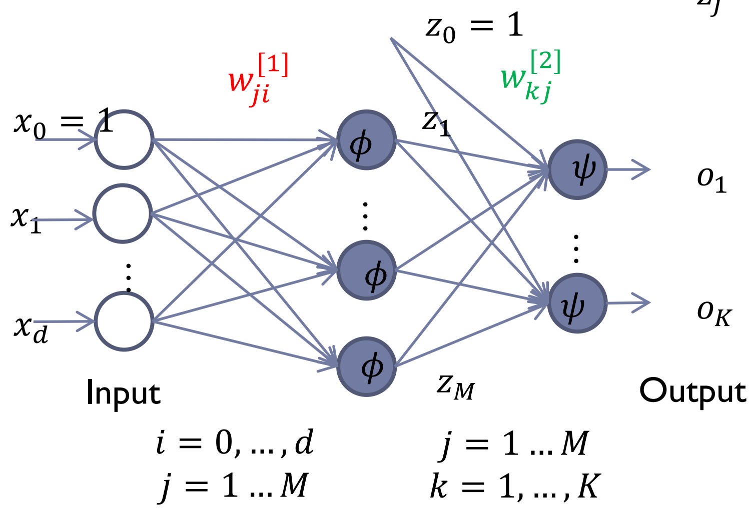
 $f(z) = \dfrac{1}{1+\exp(-z)}$	$f'(z) = f(z)(1 - f(z))$
 $f(z) = \tanh(z)$	f $f'(z) = 1 - f^2(z)$
 $f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$	$f'(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$
$f(z) = \log(1 + \exp(z))$	$f'(z) = \dfrac{1}{1+\exp(-z)}$

- Some popular activation functions and their derivatives

MLP with single hidden layer

- Two-layer MLP (Number of layers of adaptive weights is counted)

$$o_k(\boldsymbol{x}) = \psi\left(\sum_{j=0}^{M} w_{kj}^{[2]} z_j\right) \Rightarrow o_k(\boldsymbol{x}) = \psi\left(\sum_{j=0}^{M} w_{kj}^{[2]} \underbrace{\phi\left(\sum_{i=0}^{d} w_{ji}^{[1]} x_i\right)}_{z_j}\right)$$



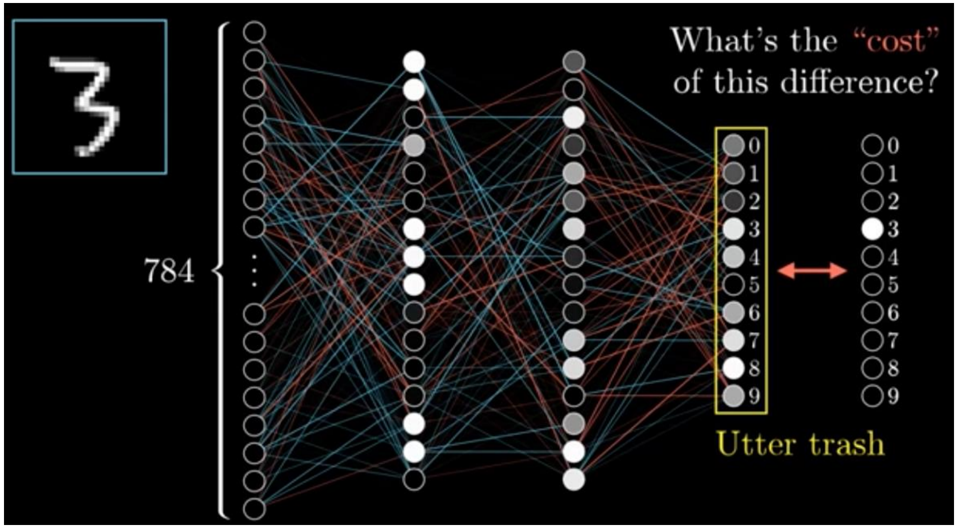
- Thus, we don't need expert knowledge or time consuming tuning of hand-crafted features
  - Compared to kernel method in which we need to perform a feature engineering phase

Training multi-layer networks

- Back-propagation
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
  - Use chain rule to efficiently compute gradients

Find the weights by optimizing the cost

- Start from random weights and then adjust them iteratively to get lower cost.
- Update the weights according to the gradient of the cost function



Source: http://3b1b.co

Training Neural Nets through Gradient Descent

- The cost function to train the neural network,
  - $\boldsymbol{h}_{\boldsymbol{W}}^{(n)}$ as the output of a network with $\boldsymbol{n}$ layers

$$\text{J} = \sum_{n=1}^{N} loss\left(\boldsymbol{h}_{\boldsymbol{W}}^{(n)}, \boldsymbol{y}^{(n)}\right)$$

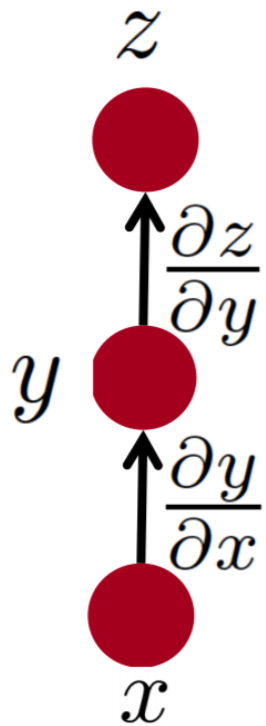
- Gradient descent algorithm
- Initialize all weights and biases $\left\{w_{ji}^{[k]}\right\}$
  - Using the extended notation : the bias is also weight
- Do :
  - For every layer k for all i, j update:
    - $w_{ji}^{[k]} = w_{ji}^{[k]} - \eta \frac{dJ}{dw_{ji}^{[k]}}$
- Until J has converged

Training multi-layer networks

- Back-propagation
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
  - Use chain rule to efficiently compute gradients

Simple chain rule

- $z = f\big(g(x)\big)$
- $y = g(x)$



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Calculus Refresher: Basic rules of calculus

$$y = f(x)$$

with derivative

$$\frac{dy}{dx}$$

the following must hold for sufficiently small Δx $\Rightarrow$ $\Delta y \approx \frac{dy}{dx}\Delta x$

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}$$

For any differentiable function

$$y = f(x_1, x_2, \ldots, x_M)$$

with partial derivatives

$$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_M}$$

the following must hold for sufficiently small $\Delta x_1, \Delta x_2, \ldots, \Delta x_M$

$$\Delta y \approx \frac{\partial y}{\partial x_1}\Delta x_1 + \frac{\partial y}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial y}{\partial x_M}\Delta x_M$$

Calculus Refresher: Chain rule

For any nested function $y = f(g(x))$

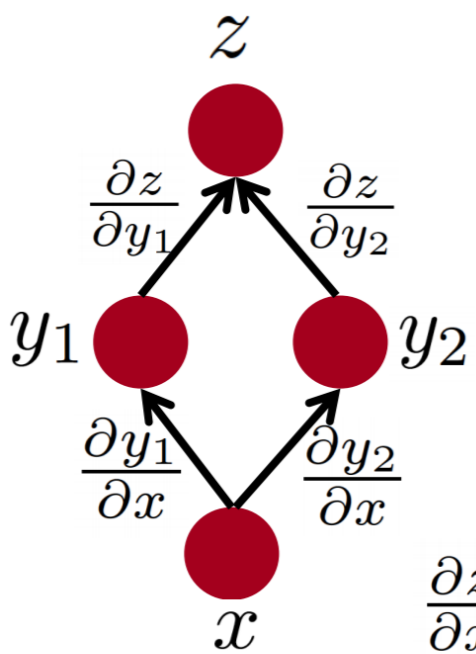
$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g(x)} \frac{dg(x)}{dx}}$$

Check – we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$

$y = f(z) \Rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dz} \frac{dg(x)}{dx} \Delta x$ ✓

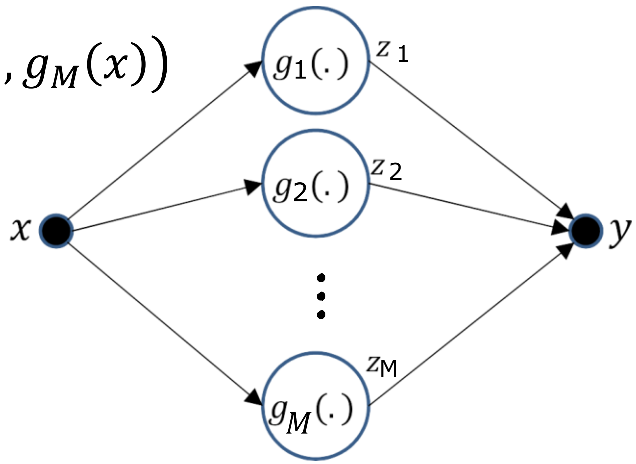
Multiple paths chain rule



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Distributed Chain Rule: Influence Diagram

$$y = f\big(g_1(x), g_2(x), \dots, g_M(x)\big)$$



- x affects y through each $g_1, \dots, g_M$

Calculus Refresher: Distributed Chain rule

$$y = f\left(g_1(x), g_1(x), \dots, g_M(x)\right)$$

$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}}$$

Check:

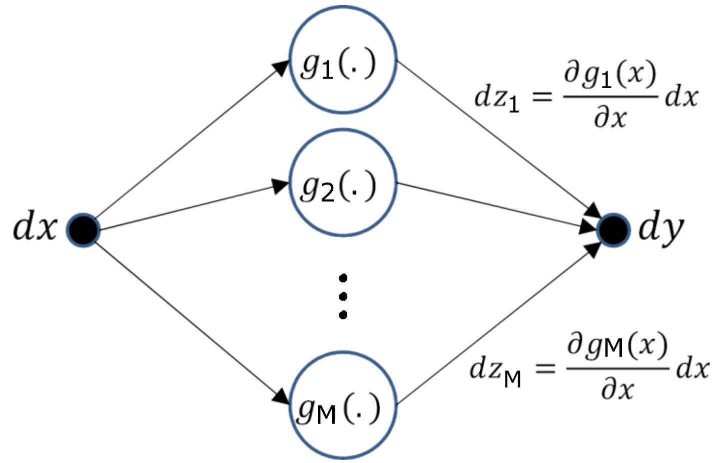
$$\Delta y = \frac{dy}{dx}\Delta x$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial f}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)}\Delta g_M(x)$$

$$\boxed{\Delta y = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx}\Delta x + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx}\Delta x + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\Delta x}$$

$$\boxed{\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x} \checkmark$$

Distributed Chain Rule: Influence Diagram



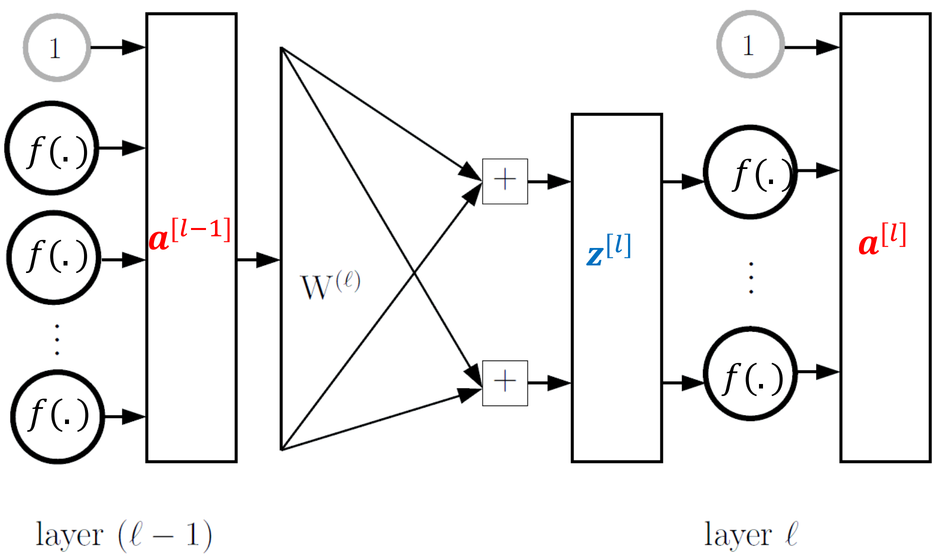
- Small perturbations in x cause small perturbations in each of $g_1 \cdots g_M$, which individually additively perturbs y

Returning to our problem

- How to compute $$\frac{d\,\boldsymbol{J}}{dw_{ji}^{[k]}}$$

Backpropagation: Notation

- $\boldsymbol{a}^{[0]} \leftarrow Input$
- $output \leftarrow \boldsymbol{a}^{[L]}$
- f as the activation function



Backpropagation: Last layer gradient

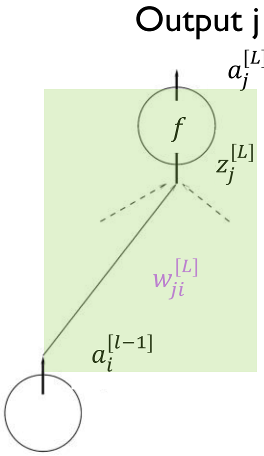
$$a_j^{[L]} = f\left(z_j^{[L]}\right)$$

$$z_j^{[L]} = \sum_{i=0}^{M} w_{ji}^{[L]} a_i^{[L-1]}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = \frac{\partial loss}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial w_{ji}^{[L]}}$$

$$\frac{\partial a^{[L]}}{\partial w_{ji}^{[L]}} = f'\left(z_j^{[L]}\right) \frac{\partial z_j^{[L]}}{\partial w_{ji}^{[L]}} = f'\left(z_j^{[L]}\right) a_i^{[L-1]}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = \frac{\partial loss}{\partial a_j^{[L]}} f'\left(z_j^{[L]}\right) a_i^{[L-1]}$$



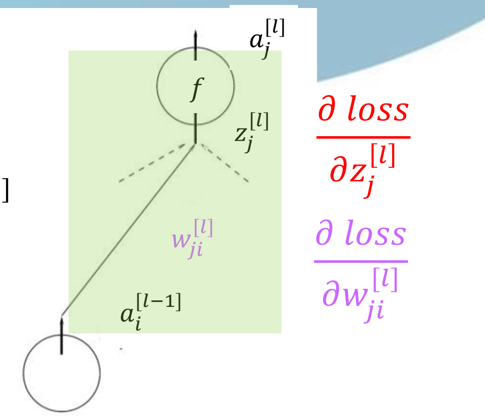
$$\frac{\partial loss}{\partial a_j^{[L]}}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}}$$

Previous layers gradients

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

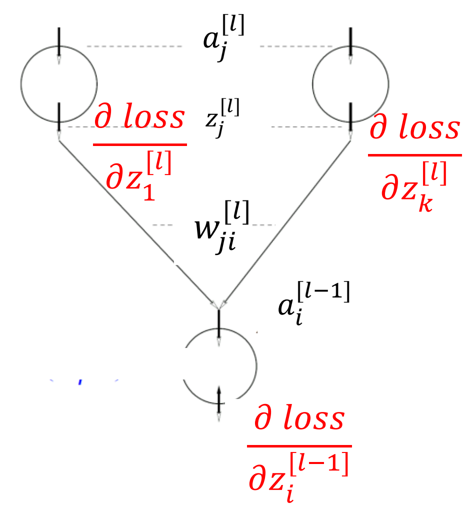
$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} a_i^{[l-1]}$$

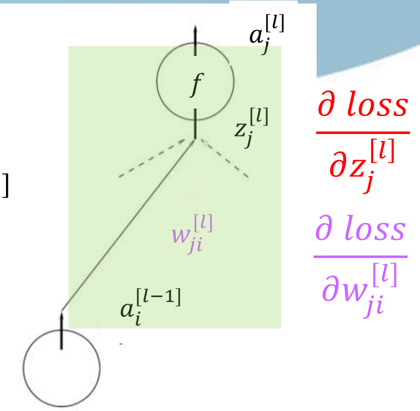
$$\frac{\partial\, loss}{\partial z_j^{[l]}} =?$$



Previous layers gradients

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

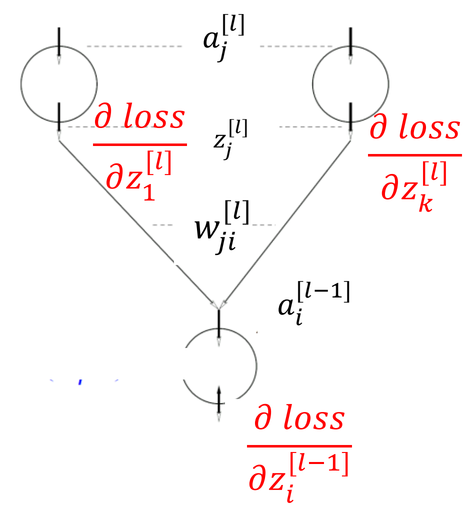
$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial\, loss}{\partial z_j^{[l]}} a_i^{[l-1]}$$

$$\frac{\partial\, loss}{\partial z_i^{[l-1]}} = \frac{\partial a_i^{[l-1]}}{\partial z_i^{[l-1]}} \sum_{j=1}^{d^{[l]}} \frac{\partial\, loss}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}}$$

$$= f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \frac{\partial\, loss}{\partial z_j^{[l]}} \times w_{ji}^{[l]}$$



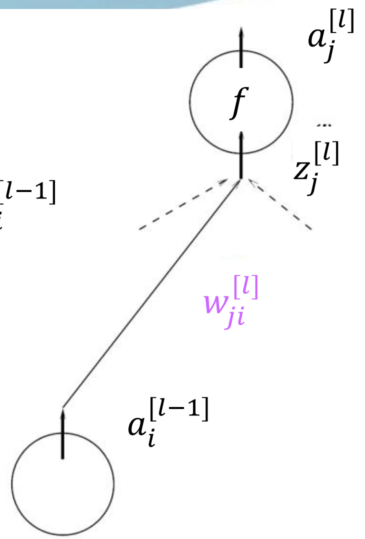
Backpropagation:

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \boxed{\frac{\partial\, loss}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}$$

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



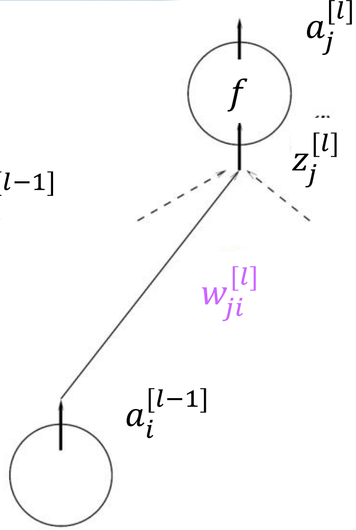
Backpropagation:

$$\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \boxed{\frac{\partial\, loss}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}$$

$$= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}$$

$$a_j^{[l]} = f\left(z_j^{[l]}\right)$$

$$z_j^{[l]} = \sum_{i=0}^{M} w_{ji}^{[l]} a_i^{[l-1]}$$



- $\delta_j^{[l]} = \frac{\partial\, loss}{\partial z_j^{[l]}}$ is the sensitivity of the loss to $z_j^{[l]}$
- Sensitivity vectors can be obtained by running a backward process in the network architecture (hence the name backpropagation.)

We will compute $\boldsymbol{\delta}^{[l-1]}$ from $\boldsymbol{\delta}^{[l]}$:

$$\delta_i^{[l-1]} = f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backward process on sensitivity vectors

- For the final layer $l = L$:

$$\delta_j^{[L]} = \frac{\partial\, loss}{\partial z_j^{[L]}}$$

- Compute $\boldsymbol{\delta}^{[l-1]}$ from $\boldsymbol{\delta}^{[l]}$: by running a backward process in the network architecture:

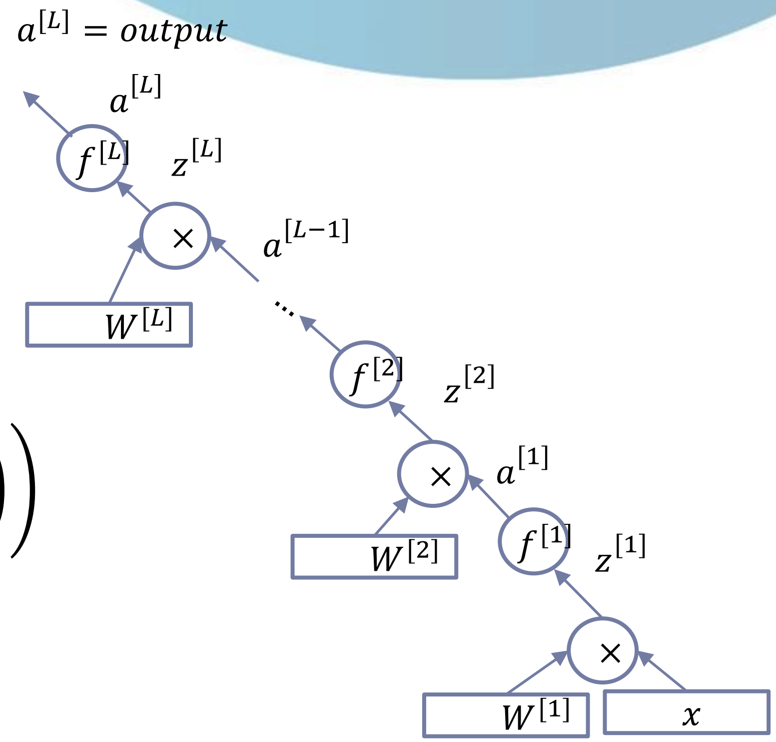
$$\delta_i^{[l-1]} = f'\left(z_i^{[l-1]}\right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backpropagation Algorithm

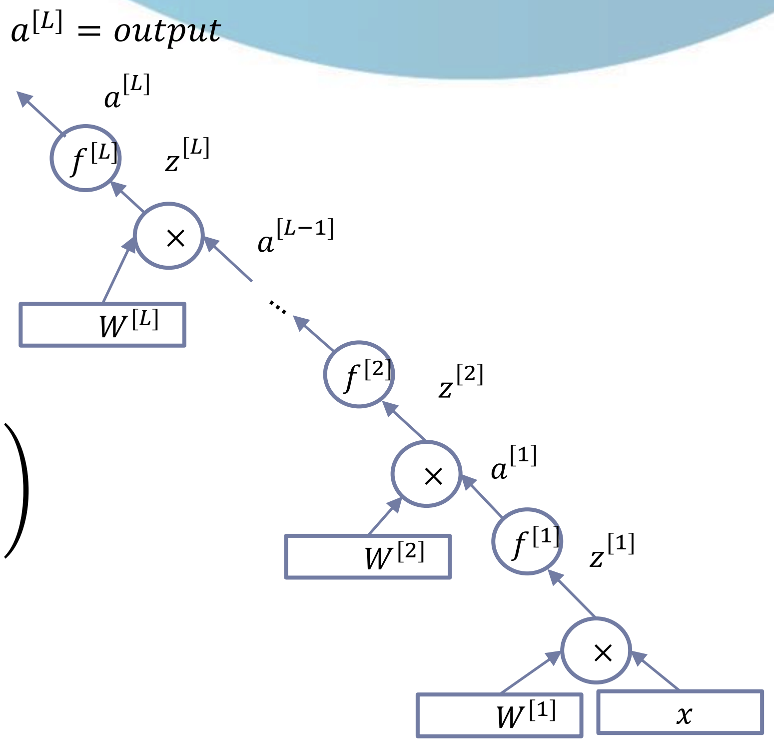
- Initialize all weights to small random numbers.
- **While not satisfied**
- **For** each training example **do**:
  1. Feed forward the training example to the network and compute the outputs of all units in forward step (z and a) and the loss
  2. For each unit find its δ in the backward step
  3. Update each network weight $w_{ji}^{[l]}$ as $w_{ji}^{[l]} \leftarrow w_{ji}^{[l]} - \eta \frac{\partial\, loss}{\partial w_{ji}^{[l]}}$ where $\frac{\partial\, loss}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} \times a_i^{[l-1]}$

Multi-layer network: Matrix notation

$$Output = a^{[L]}$$
$$= f\left(z^{[L]}\right)$$
$$= f\left(W^{[L]}a^{[L-1]}\right)$$
$$= f\left(W^{[L]}f(W^{[L-1]}a^{[L-2]}\right)$$
$$= f\left(W^{[L]}f\left(W^{[L-1]} \dots f\left(W^{[2]}f(W^{[1]}x)\right)\right)\right)$$



Multi-layer network: Matrix notation



$$Output = a^{[L]}$$
$$= f\left(z^{[L]}\right)$$
$$= f\left(W^{[L]}a^{[L-1]}\right)$$
$$= f\left(W^{[L]}f(W^{[L-1]}a^{[L-2]}\right)$$
$$= f\left(W^{[L]}f\left(W^{[L-1]}\dots f\left(W^{[2]}f(W^{[1]}x)\right)\right)\right)$$

$$\frac{\partial\, loss}{\partial W^{[l]}} = \frac{\partial\, loss}{\partial z^{[l]}} a^{[l-1]^T}$$

$$\frac{\partial\, loss}{\partial z^{[l]}} = f'(z^{[l]}) W^{[l+1]^T} \frac{\partial\, loss}{\partial z^{[l+1]}}$$

References

- Mahdieh Soleymani, Machine learning, Sharif university