Lec12 Deep Learning
12-2 Multilayer Perceptron

Yang Shu

School of Data Science and Engineering

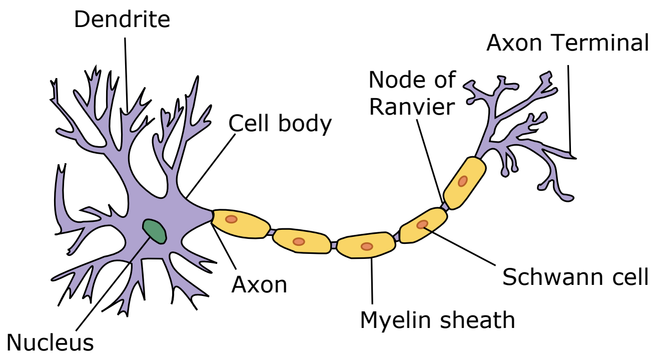
yshu@dase.ecnu.edu.cn

[Acknowledgement: Slides are adapted from Deep Learning Course, Mingsheng Long, THU]

Outline

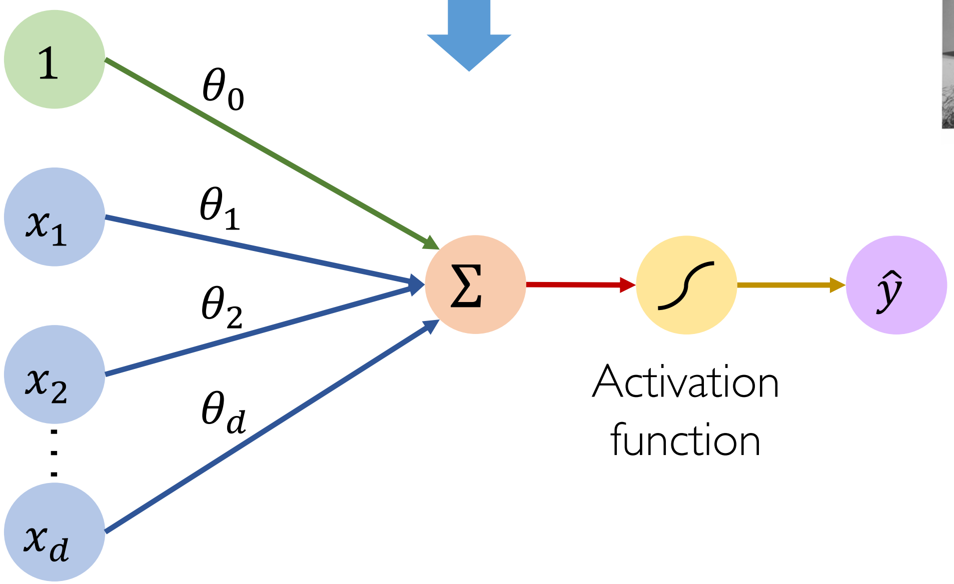
- **Feedforward Network**
  - **Perceptron**
  - Multilayer Perceptron
- Backpropagation
- Practical Training Strategies

Neuron and Perceptron



Rosenblatt 1958
An psychologist

Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms



Inputs Weights Sum Non-Linearity Output

Linear combination of inputs

Output

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{d} x_i\theta_i\right)$$

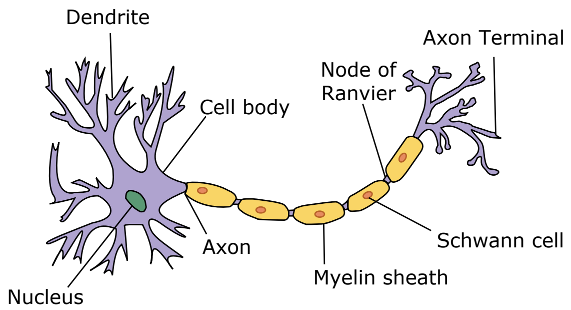
Non-Linearity
Activation function

Bias

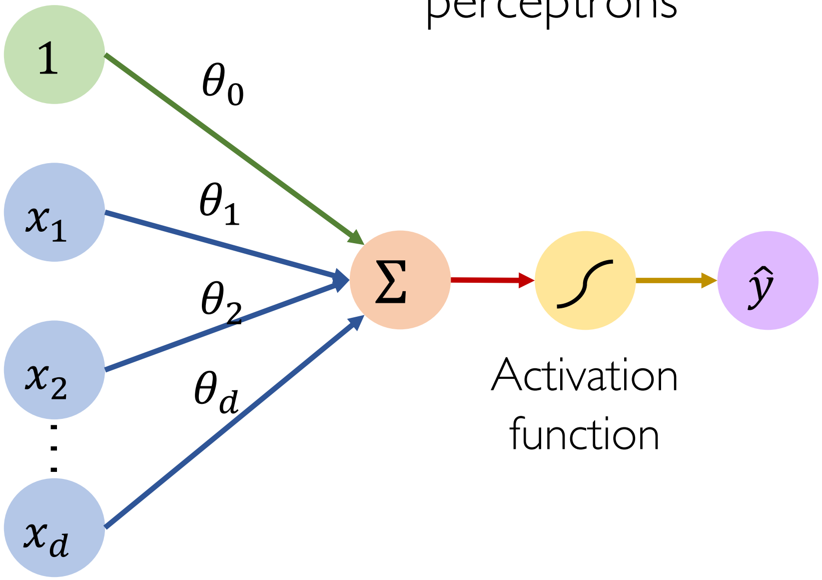
Outline

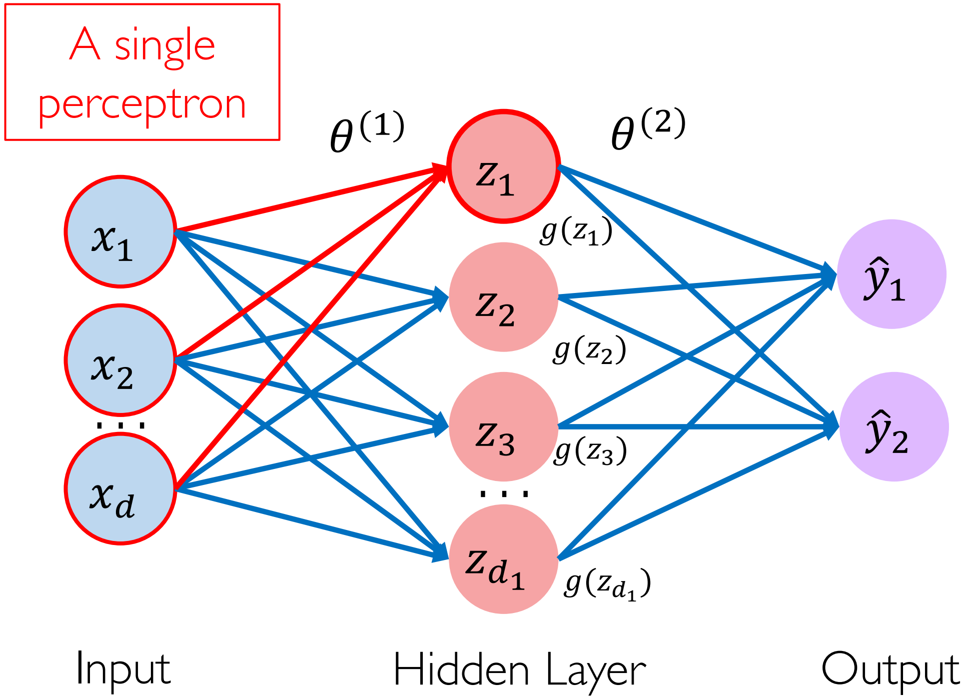
- **Feedforward Network**
  - Perceptron
  - **Multilayer Perceptron**
- Backpropagation
- Practical Training Strategies

Multilayer Perceptron (MLP)



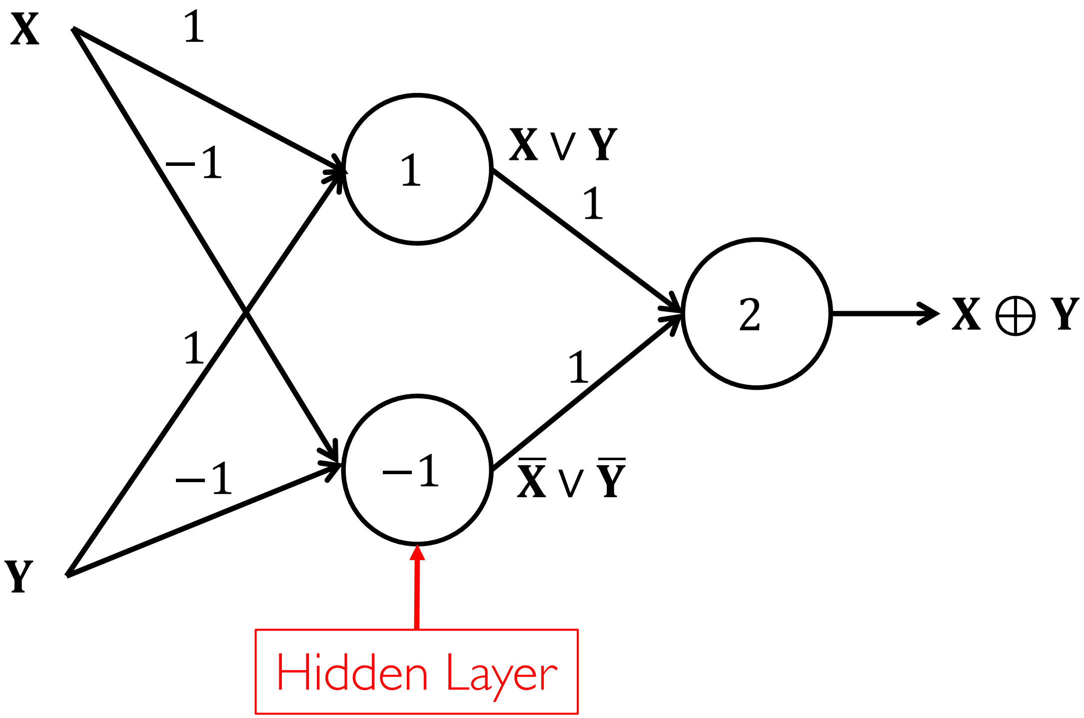
MLP is layering of perceptrons

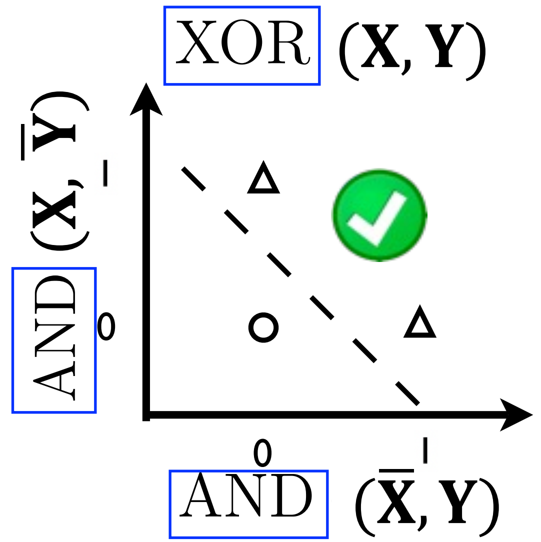




MLP for XOR

$$X \oplus Y = (X \vee Y)\&(\bar{X} \vee \bar{Y})$$

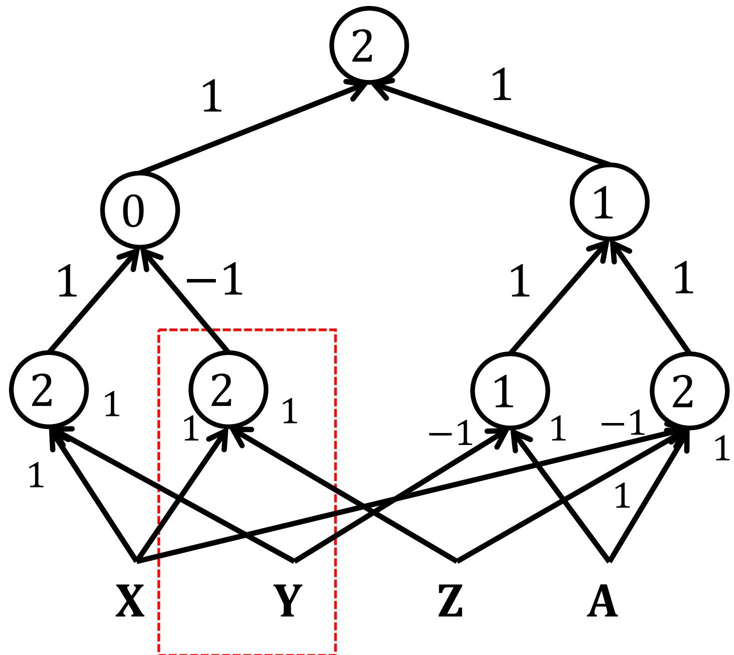




Marvin Minsky and Seymour Papert. Perceptrons. An Introduction to Computational Geometry. 1969.

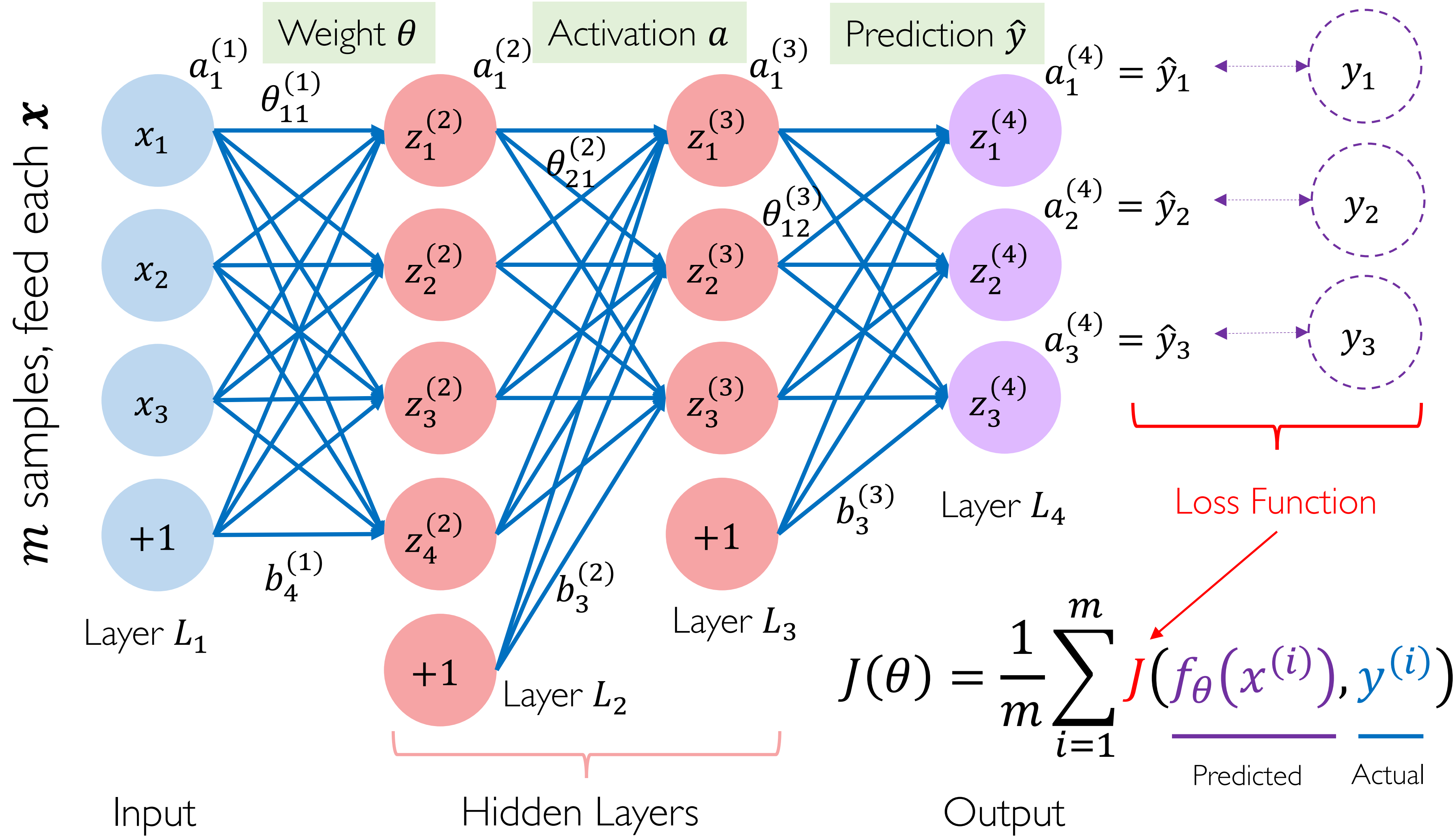
MLP: Boolean Functions

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&\left((X\&Y)|\overline{(X\&Z)}\right)$$



- MLP is universal to represent arbitrarily complex Boolean functions
- The connections in MLP can be sparse - a phenomenon in the **Brain**

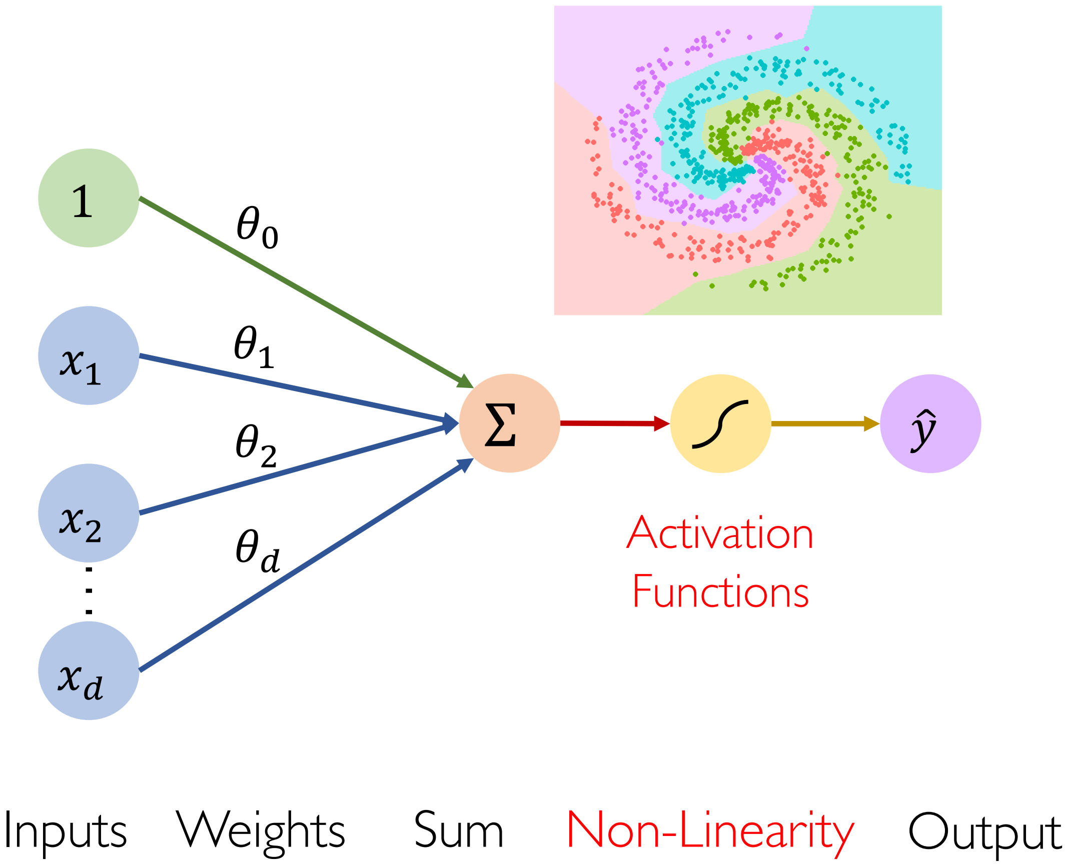
Multilayer Perceptron (MLP)



$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} J\big(\underbrace{f_\theta(x^{(i)})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\big)$$

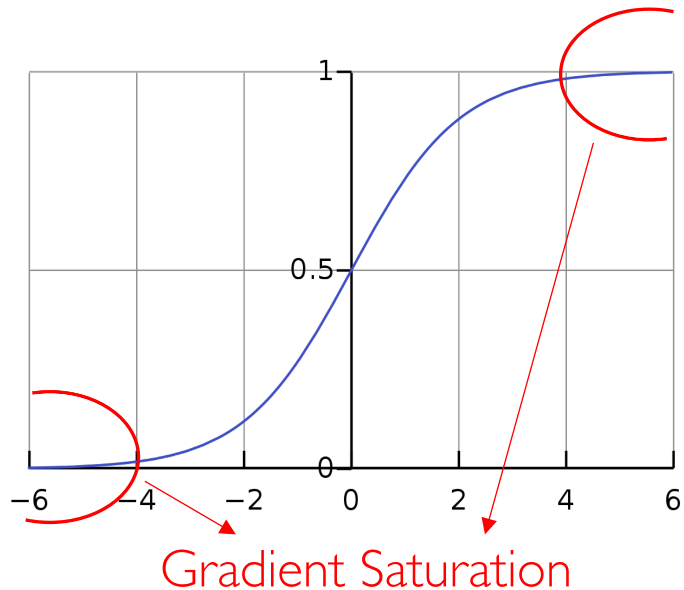
Activation Functions

Non-linearity through activation functions $\hat{y} = g\left(\theta_0 + \boldsymbol{X}^{\mathrm{T}}\boldsymbol{\theta}\right)$



sigmoid function

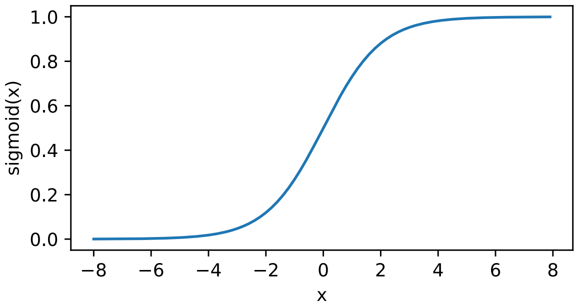
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



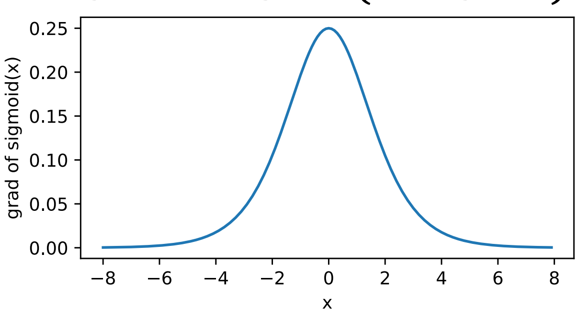
Activation Functions

Sigmoid

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



$$g'(z) = g(z)\big(1-g(z)\big)$$

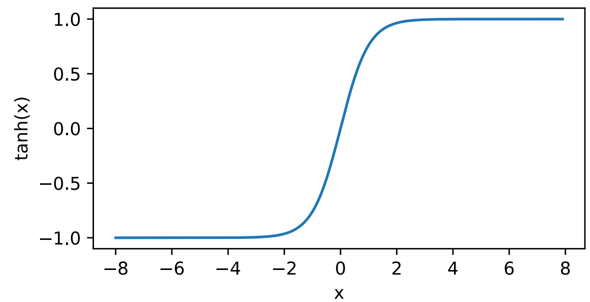


Range in (0, 1), probability

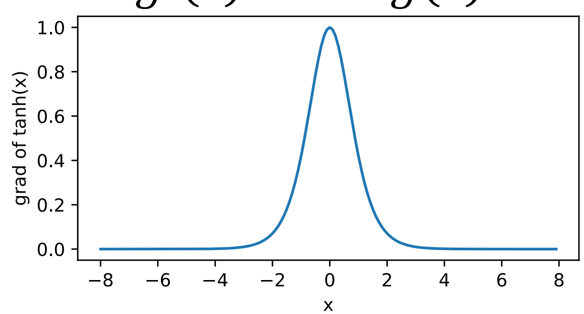
Relative smaller gradient

Hyperbolic Tangent (tanh)

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



$$g'(z) = 1 - g(z)^2$$

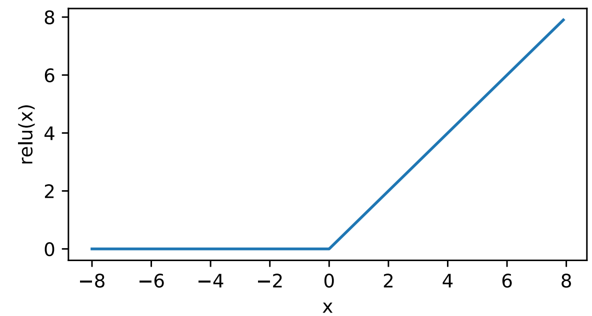


Zero-centered

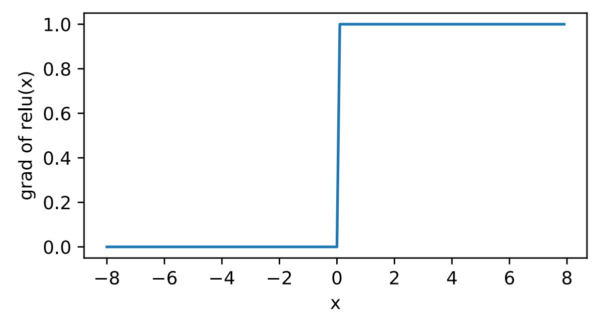
Relatively larger saturation region

Rectified Linear Unit (**ReLU**)

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



$$g'(z) = \mathrm{sgn}(z)$$



Time-efficient and faster convergence, but neurons are prone to death.

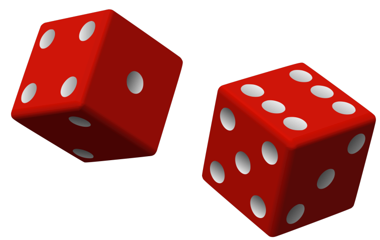
Activation Functions

- Softmax function

$$g(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

  - Winner takes all
  - The biggest one dominates the outputs

Categorical distribution



rooster

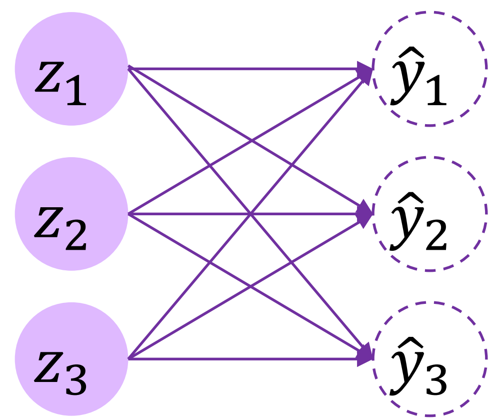
cat

dog

donkey

- Exponent arithmetic
  - Cause numerical overflow
  - For numerical stability, implementation is

$$g(\mathbf{z})_i = \frac{e^{z_i - z_u}}{\sum_{j=1}^{k} e^{z_j - z_u}}, u = \operatorname{argmax}(z_j)$$



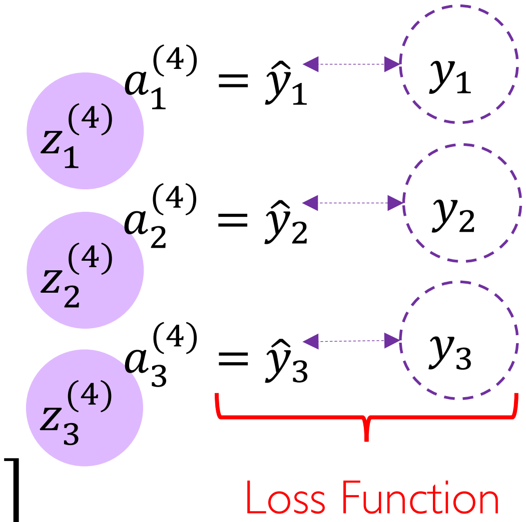
$z_j \in (-\infty, +\infty)$ $\quad \hat{y}_j \in [0,1], \sum_{j=1}^{k} \hat{y}_j = 1$

Cost Function

- Softmax function in the output layer

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{(n_l)} = f_\theta\left(\boldsymbol{x}^{(i)}\right) = \begin{bmatrix} p\left(y^{(i)} = 1 \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \\ p\left(y^{(i)} = 2 \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \\ \vdots \\ p\left(y^{(i)} = k \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} \exp(z_j^{(n_l)})} \begin{bmatrix} \exp(z_1^{(n_l)}) \\ \exp(z_2^{(n_l)}) \\ \vdots \\ \exp(z_k^{(n_l)}) \end{bmatrix}$$

- Cross-entropy $J(q,p) = -\sum_{j=1}^{k} q_j \log p_j$
  - Loss function: $J(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{j=1}^{k} y_j \log \hat{y}_j$
  - $\boldsymbol{y}$ follows one-hot coding
    - » $y_j = 1$ if $y^{(i)} = j$ and $y_j = 0$ otherwise



- Cost function

$$\min_\theta J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[\sum_{j=1}^{k} \mathbf{1}\{y^{(i)} = j\} \log \frac{\exp\left(z_j^{(n_l)}\right)}{\sum_{j'=1}^{k} \exp\left(z_{j'}^{(n_l)}\right)} \right]$$

Outline

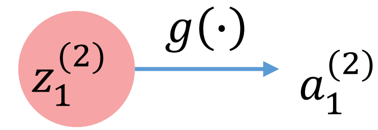
- Feedforward Network
    - Perceptron
    - Multilayer Perceptron
- **Backpropagation**
- Practical Training Strategies

Multilayer Perceptron: Notations

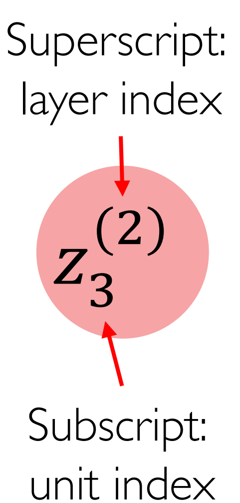
- Recursive transformations in propagation
  - $a_i^{(1)} = x_i$
  - $z_i^{(l)} = \sum_{j=1}^{s_{l-1}} \theta_{ij}^{(l-1)} a_j^{(l-1)} + b_i^{(l-1)}$
  - $a_i^{(l)} = g\left(z_i^{(l)}\right)$
  - $\hat{y}_i = a_i^{(n_l)}$

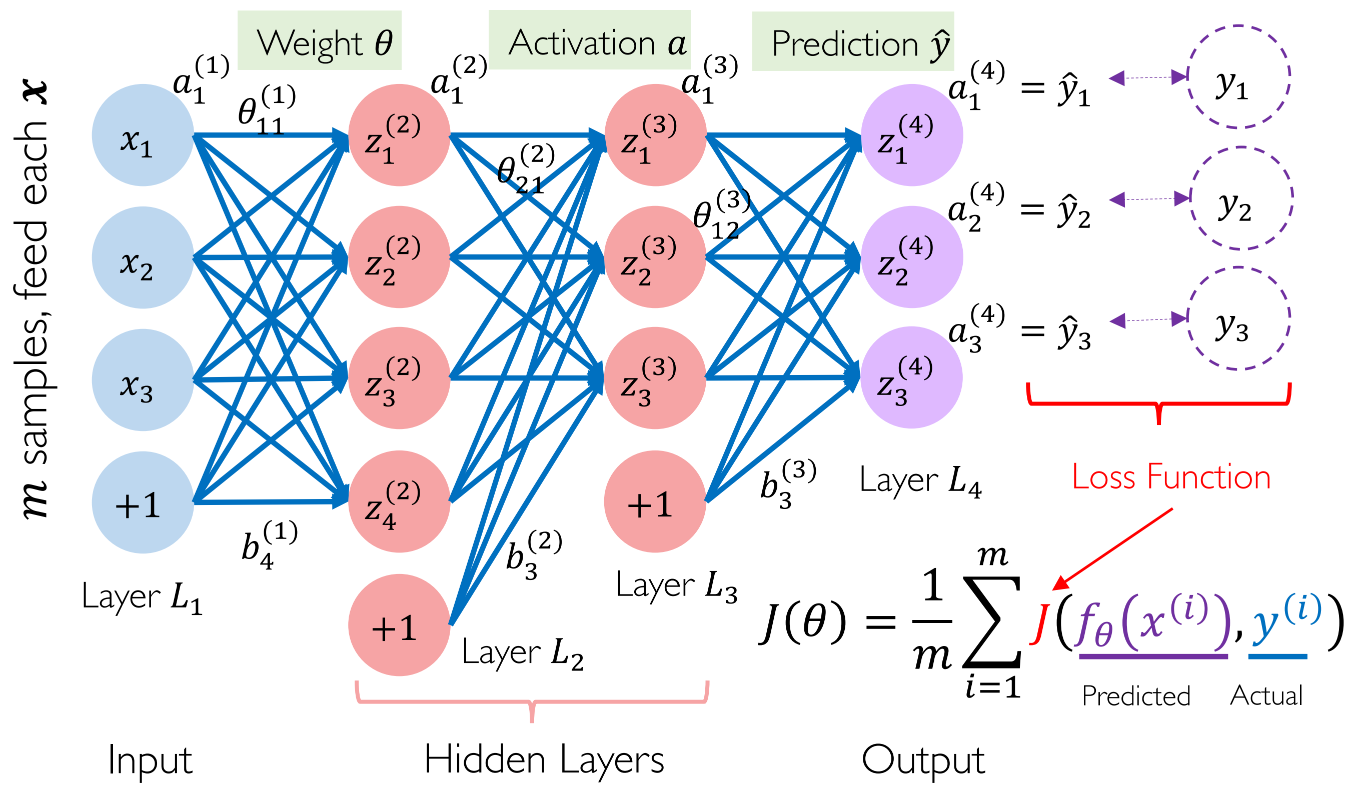
#units

#layers



Activation function (element-wise)





$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} J\Big(\underbrace{f_\theta\big(x^{(i)}\big)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

Gradient-Based Training

$$\arg\min_{\theta} O(\mathcal{D};\theta) = \sum_{i=1}^{m} L\left(y_i, f\left(x_i\right);\theta\right) + \Omega(\theta)$$

Risk Minimization

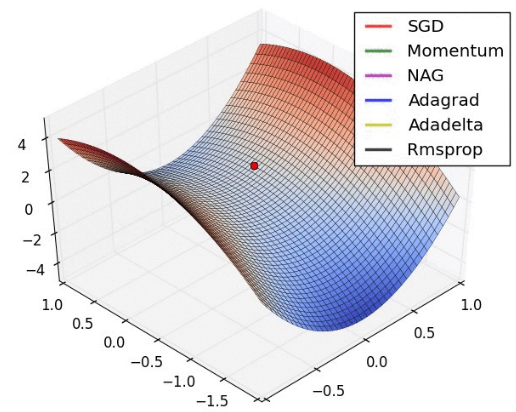
Data Hypothesis Parameters

- Iterative Algorithm (Convergence Guarantee)

```
for (t = 1 to T) {
    1. ForwardPropagation()
    2. BackwardPropagation()
    3. θ^(t+1) ← θ^(t) − η∇_θ O(D;θ^(t))
}
```

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla_{\theta} O\left(\mathcal{D};\theta^{(t)}\right)$$

Parameter Updates



Gradient Descent (GD)

$$J(\theta, b) = \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\theta, b; x^{(i)}, y^{(i)}\right)\right]$$

- Gradient descent
  - $\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \eta \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b)$
  - $b_i^{(l)} = b_i^{(l)} - \eta \frac{\partial}{\partial b_i^{(l)}} J(\theta, b)$
  - $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{ij}^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)$
  - $\frac{\partial}{\partial b_i^{(l)}} J(\theta, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)$

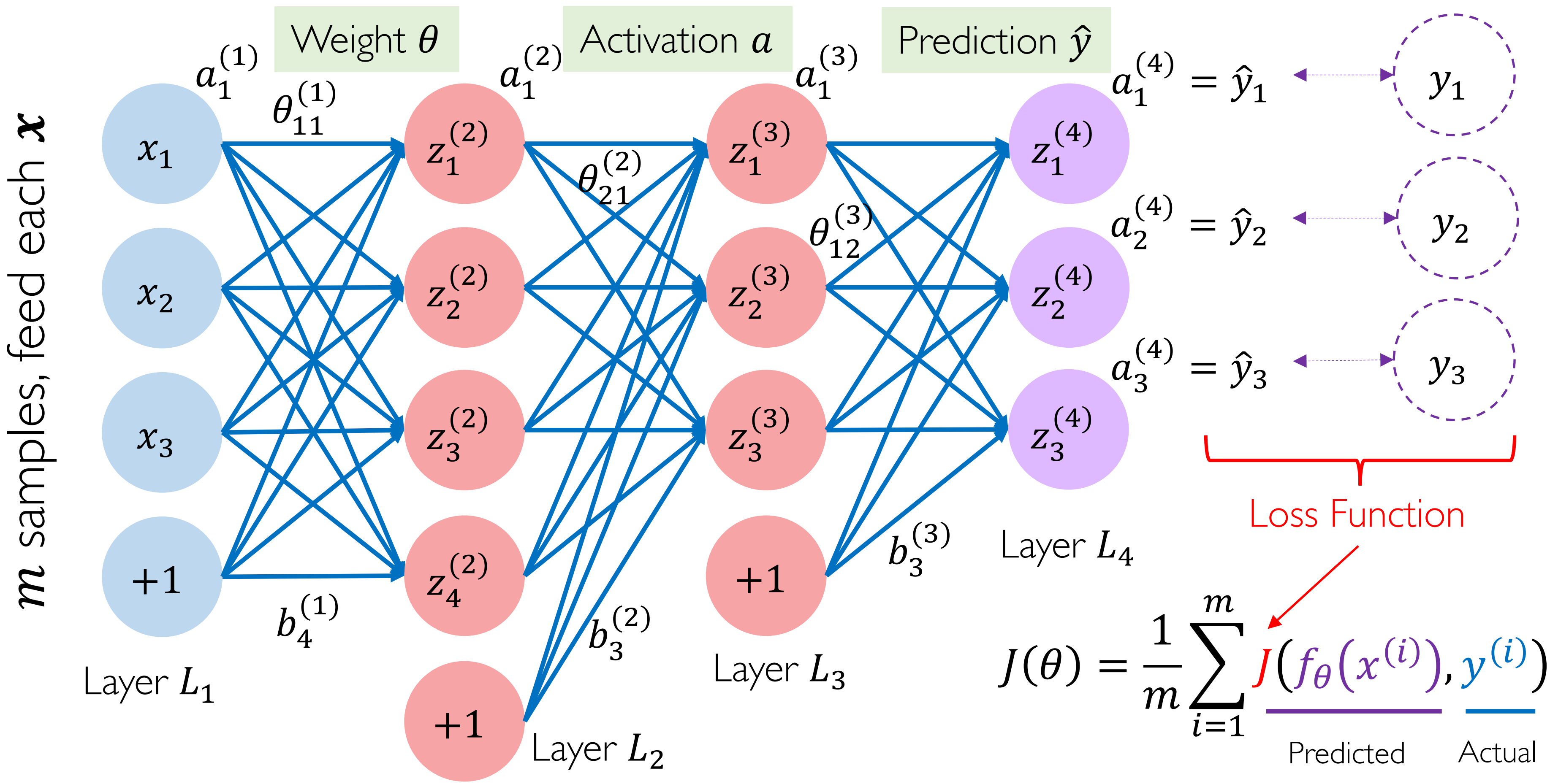
Derivatives of last layer

$$J\left(\theta, b; x^{(i)}, y^{(i)}\right) = -\sum_{j=1}^{k} \mathbf{1}\{y^{(i)} = j\} \log \frac{\exp(z_j^{(n_l)})}{\sum_{j'=1}^{k} \exp(z_{j'}^{(n_l)})}$$

$$\frac{\partial J(\theta, b)}{\partial z_j^{(n_l)}} = -\left(\mathbf{1}\{y^{(i)} = j\} - p\left(y^{(i)} = j | x^{(i)}; \theta\right)\right)$$

How to compute the derivatives for parameters in hidden layers?

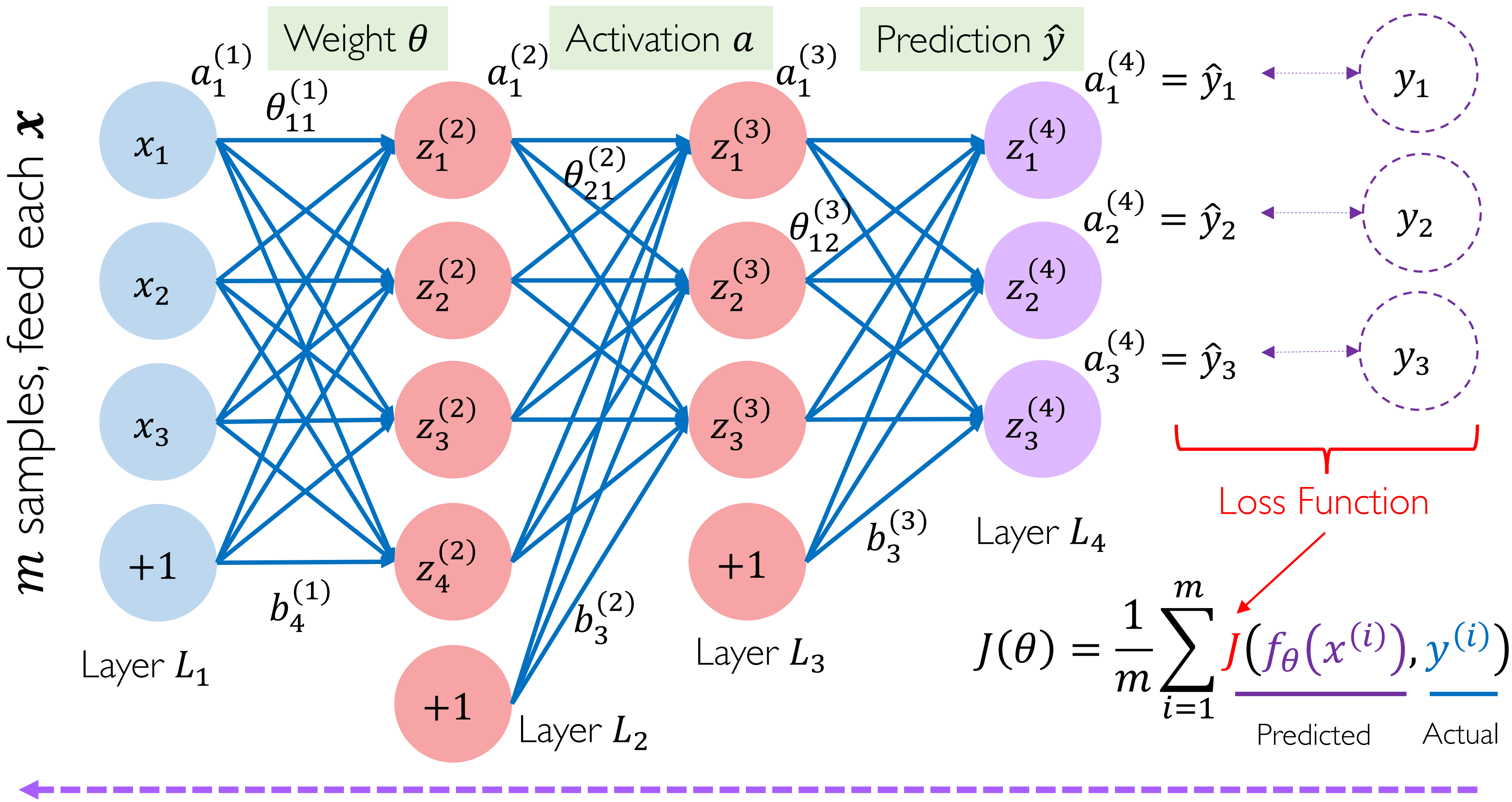
Step 1: Forward Propagation



$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} J\left(\underbrace{f_\theta\left(x^{(i)}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

- Forward propagate all $(x, y)'s$ to compute activations and objective $J(\theta, b)$

Step 2: Backward Propagation



- Backward propagate the gradients $\nabla J(\theta, b)$ to update parameters in all layers

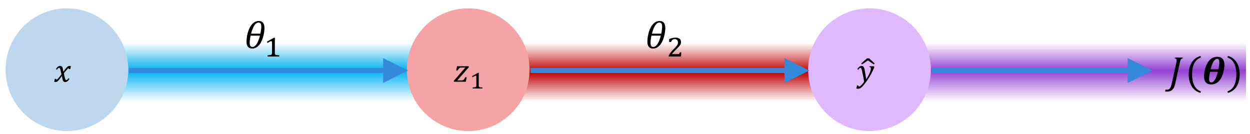
Backpropagation (BP) Algorithm

- The backpropagation algorithm applies the <u>chain rule</u> of calculus to the parameters $\boldsymbol{\theta}$ of each layer

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

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

Jacobian matrix



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

- Backpropagation is an <u>efficient</u> implementation of the chain rule
  - Uses **dynamic programming** (table filling)
  - Avoids re-computing repeated subexpressions (in dependency)
  - Speed vs memory tradeoff (sensitive to #samples $\boldsymbol{m}$)

Computing the Residual

- For each (node i in layer l), compute the residual $\delta_i^{(l)}$

$$\delta_i^{(l)} \triangleq \frac{\partial}{\partial z_i^{(l)}} J(\theta, b; x, y)$$

  - $\delta_i^{(l)}$ measures how much each (node i in layer l) is responsible for any errors $J(\theta, b)$ of the network output $z_i^{(n_l)}$
  - Among all the nodes, output nodes are directly related to the loss, and we firstly compute $\delta_i^{(n_l)}$ for each output node i in layer n_l

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} J(\theta, b; x, y) = \frac{\partial}{\partial \hat{y}_i} J(\theta, b; x, y) g'(z_i^{(n_l)})$$

  - How about hidden nodes (not directly related to the loss)?

Computing the Residual

- For each (node i in layer l), compute the <u>residual</u> $\delta_i^{(l)}$

$$\delta_i^{(l)} \triangleq \frac{\partial}{\partial z_i^{(l)}} J(\theta, b; x, y)$$

- Note that $z_j^{(l+1)} = \sum_{i=1}^{s_l} \theta_{ji}^{(l)} a_i^{(l)} + b_j^{(l)}$
- For each (hidden node i in layer l), compute $\delta_i^{(l)}$ by a weighted average of the residuals of the nodes taking $a_i^{(l)}$ as input

$$\frac{\partial J(\theta,b;x,y)}{\partial z_i^{(l)}} = \boxed{\sum_{j=1}^{s_{l+1}}} \frac{\partial J(\theta,b;x,y)}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}$$

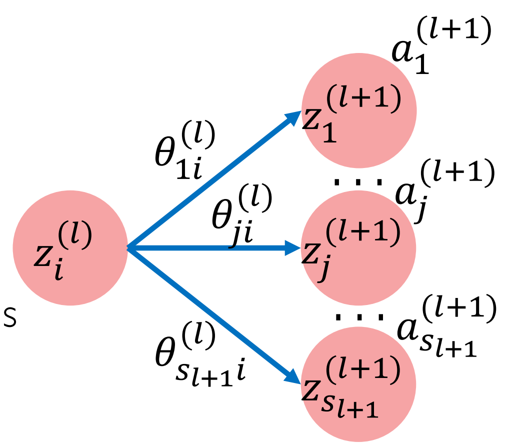
$$\delta_i^{(l)} = \sum_{j=1}^{s_{l+1}} \delta_j^{(l+1)} \theta_{ji}^{(l)} g'(z_i^{(l)})$$

layer index

$$\theta_{ji}^{(l)}$$

next layer node

previous layer node



Computing the Gradients

- For (each node i in layer l), we have computed
  - Forward propagation: $z_j^{(l+1)} = \sum_{i=1}^{s_l} \theta_{ji}^{(l)} a_i^{(l)} + b_j^{(l)}$
    - » Non-linearity $a_j^{(l)} = g\left(z_j^{(l)}\right)$
  - Backward propagation: $\delta_i^{(l)} \triangleq \frac{\partial}{\partial z_i^{(l)}} J(\theta, b; x, y)$
- Applying the chain rule, the gradients of parameters $\theta_{ij}^{(l)}$ and $b_i^{(l)}$:

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b; x, y) = \frac{\partial}{\partial z_i^{(l+1)}} J(\theta, b; x, y) \frac{\partial z_i^{(l+1)}}{\partial \theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(\theta, b; x, y) = \frac{\partial}{\partial z_i^{(l+1)}} J(\theta, b; x, y) \frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

Step 3: Parameter Updates

$$J(\theta, b) = \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\theta, b; x^{(i)}, y^{(i)}\right)\right]$$

- Gradient descent
  - $\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \eta \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b)$
  - $b_i^{(l)} = b_i^{(l)} - \eta \frac{\partial}{\partial b_i^{(l)}} J(\theta, b)$
  - $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial \theta_{ij}^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)$
  - $\frac{\partial}{\partial b_i^{(l)}} J(\theta, b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)$

Derivatives of last layer

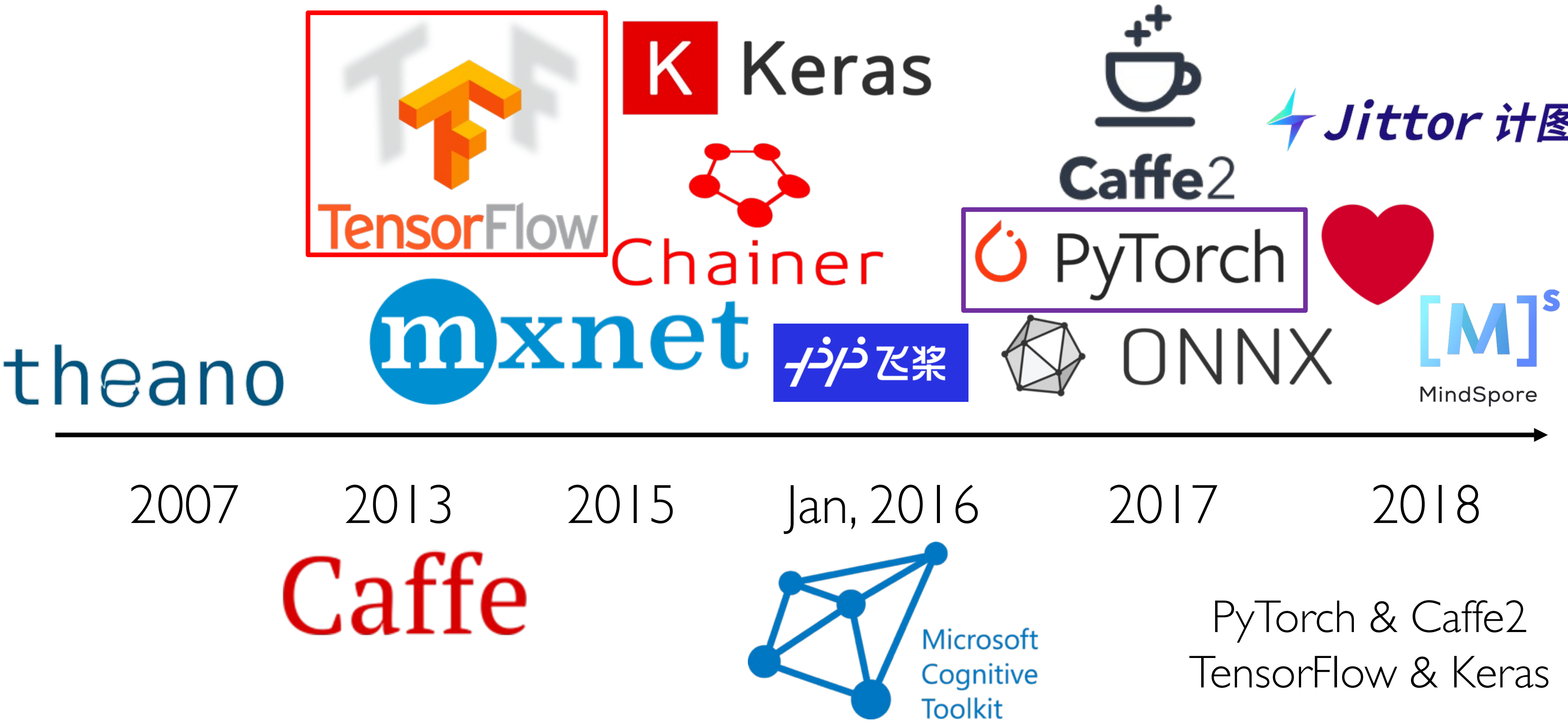
$$J\left(\theta, b; x^{(i)}, y^{(i)}\right) = -\sum_{j=1}^{k} \mathbf{1}\{y^{(i)} = j\} \log \frac{\exp(z_j^{(n_l)})}{\sum_{j'=1}^{k} \exp(z_{j'}^{(n_l)})}$$

$$\frac{\partial J(\theta, b)}{\partial z_j^{(n_l)}} = -\left(\mathbf{1}\{y^{(i)} = j\} - p\left(y^{(i)} = j | x^{(i)}; \theta\right)\right)$$

Computing all these by hand is painful

Deep Learning Systems

- Symbolic vs. Imperative
- Computational Graph and Automatic Differentiation



PyTorch & Caffe2
TensorFlow & Keras

Outline

- Feedforward Network
  - Perceptron
  - Multilayer Perceptron
- Backpropagation
- **Practical Training Strategies**

Stochastic Gradient Descent (SGD)

$$J(\theta, b) = \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\theta, b; x^{(i)}, y^{(i)}\right)\right]$$

- Stochastic Gradient Descent

  - $\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \eta \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b)$

  Stochasticity helps escaping from saddle points.

  - $b_i^{(l)} = b_i^{(l)} - \eta \frac{\partial}{\partial b_i^{(l)}} J(\theta, b)$

  Too expensive when $\boldsymbol{m}$ is very large!

  - $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b) = \left[\frac{1}{m} \cancel{\sum_{i=1}^{m}} \frac{\partial}{\partial \theta_{ij}^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)\right]$

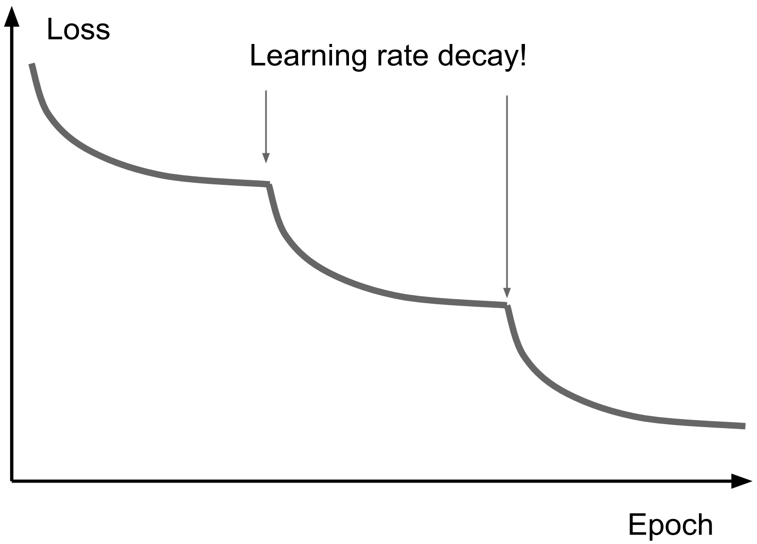
  For each iteration, compute gradients for a mini-batch (小批量) of data points

  - $\frac{\partial}{\partial b_i^{(l)}} J(\theta, b) = \frac{1}{m} \cancel{\sum_{i=1}^{m}} \frac{\partial}{\partial b_i^{(l)}} J\left(\theta, b; x^{(i)}, y^{(i)}\right)$

  For each epoch, Shuffle (洗牌) the dataset

Learning Rate Decay

- All SGD algorithms take learning rate η as a hyperparameter
- Though some algorithms can adjust learning rate adaptively, a good choice of learning rate η could result in better performance
- To make network converge **stably and quickly**, we could set learning rate that **decays over time**



- Exponential decay strategy:

$$\eta = \eta_0 e^{-kt}$$

- 1/t decay strategy:

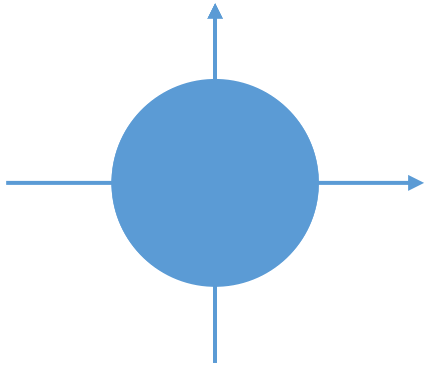
$$\eta = \eta_0/(1 + kt)$$

- Step strategy: decay very T iterations (widely used in practice!)

Weight Decay

- *L2* regularization:

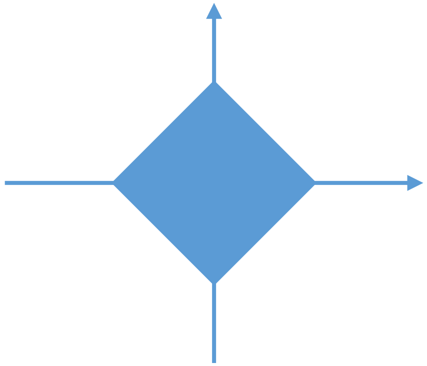
$$\Omega(\theta) = \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$



$$\frac{\partial}{\partial\theta^{(l)}}\Omega(\theta)= \lambda\theta^{(l)}$$

- *L1* regularization:

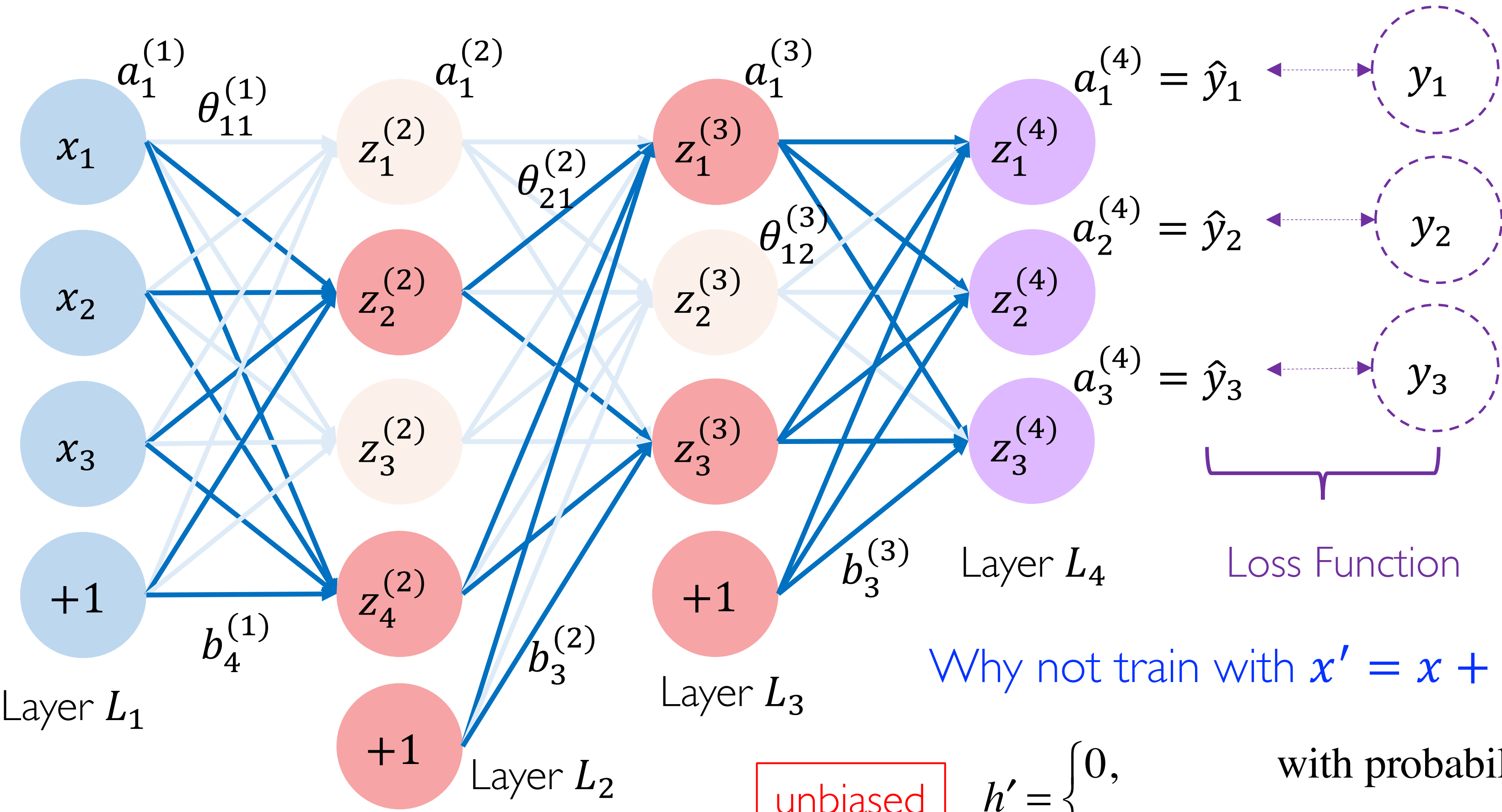
$$\Omega(\theta) = \lambda\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}|\theta_{ji}^{(l)}|$$



$$\frac{\partial}{\partial\theta^{(l)}}\Omega(\theta)_{ji}= \lambda(1_{\theta_{ji}^{(l)}>0} - 1_{\theta_{ji}^{(l)}<0})$$

Dropout

- During training, randomly drop 50% of activations in each layer to $\mathbf{0}$



Why not train with $x' = x + \epsilon$?

unbiased

$$h' = \begin{cases} 0, & \text{with probability } p \\ h/(1-p), & \text{otherwise} \end{cases}$$