

AI504: Programming for Artificial Intelligence

Week 3: Neural Nets & Backpropagation

Edward Choi

Grad School of AI

edwardchoi@kaist.ac.kr

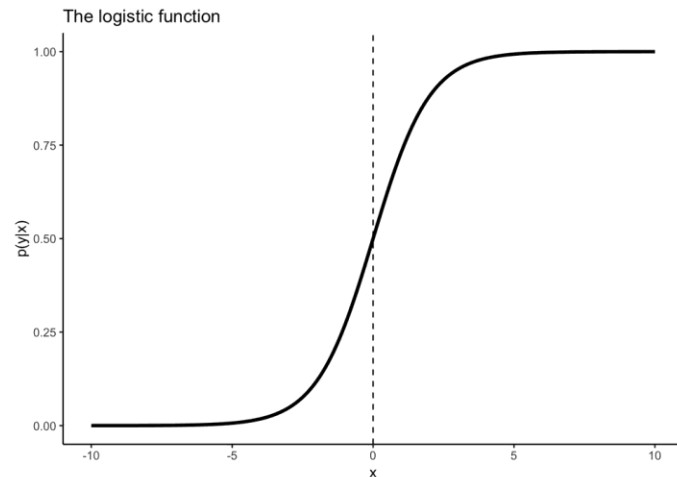
Today's Topic

- Logistic Regression
- Neural Networks
- Backpropagation
- Autograd (in PyTorch)

Logistic Regression

Logistic Regression

- Maybe the most popular model in statistical studies
 - A well-studied model. (Used since 19th century)
 - Analysis of coefficients of predictor variables (i.e. explanatory variable, independent variable, feature).



“Logistic Function”

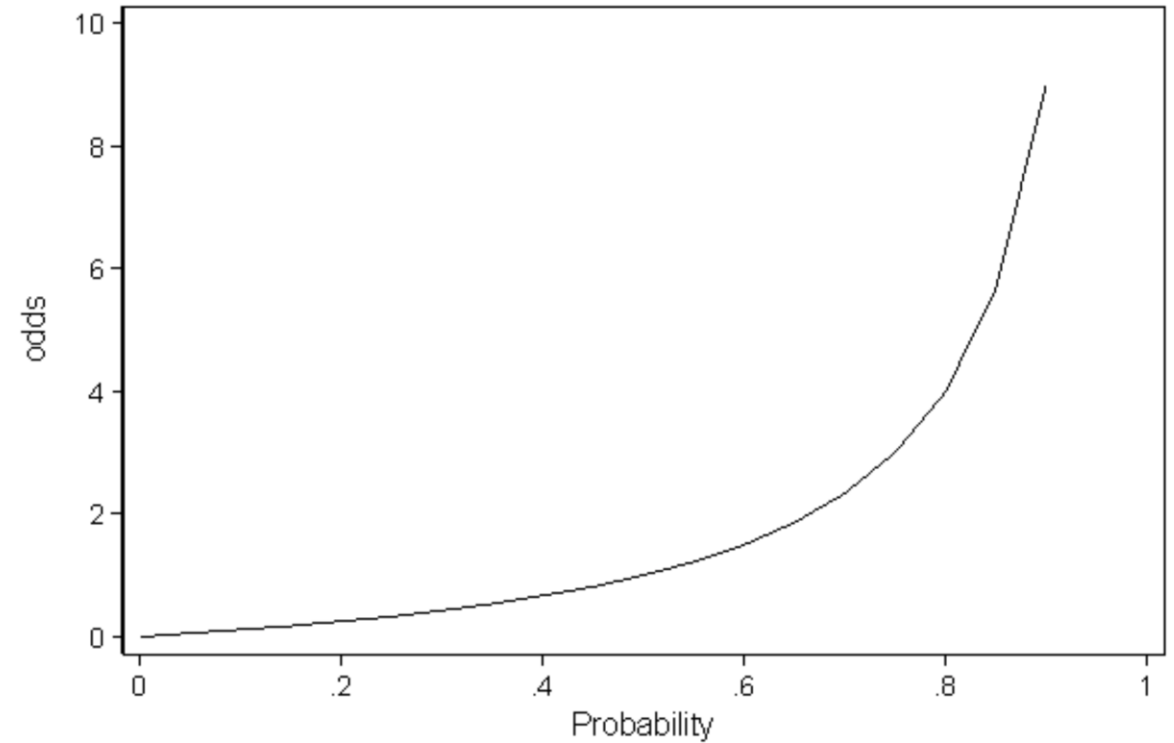
Multivariate Logistic Regression

$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_m x_m)}}$$

where usually $b = e$.

Odds

- Win Probability p
 - 0.2, 0.5, 0.8
 - $[0, 1]$
- Odds
 - $odds(p) = \frac{p}{1-p}$
 - $p = 0.8 \rightarrow Odds = 4$
 - $p = 0.5 \rightarrow Odds = 1$
 - $p = 0.1 \rightarrow Odds = 0.1111\dots$
 - $[0, \infty)$



Log-Odds (Logit)

- Logit

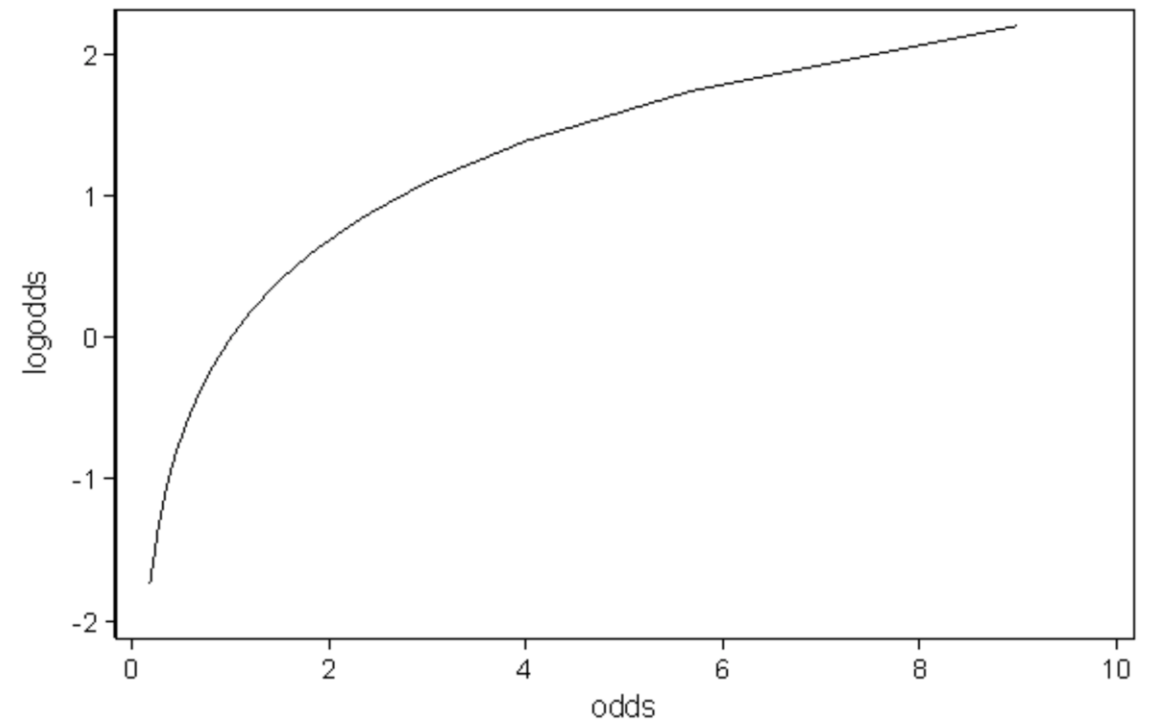
- $\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$

- $p = 0.8 \rightarrow \text{logit} = 1.3863$

- $p = 0.5 \rightarrow \text{logit} = 0$

- $p = 0.1 \rightarrow \text{logit} = -0.9542$

- $(-\infty, \infty)$



Logistic Regression

- Linear Modeling of Logit

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$



$$\frac{1-p}{p} = \frac{1}{\exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k)}$$



$$p = \frac{\exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k)}{1 + \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k)}$$



$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m)}}$$

where usually $b = e$.

Logistic Regression

- Vector form
 - Inner product

$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_m x_m)}} \quad \Rightarrow \quad h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$

Logistic Regression

- Vector form
 - Inner product

$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_m x_m)}} \quad \Rightarrow \quad h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$



Logistic Regression

- Vector form
 - Inner product

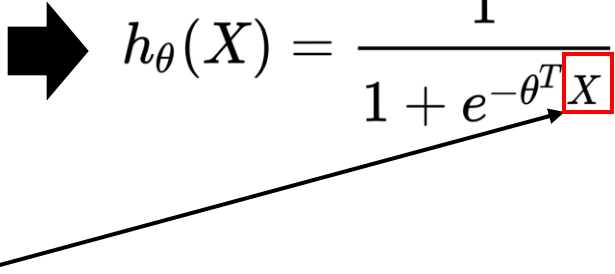
$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_m x_m)}} \quad \Rightarrow \quad h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$


Image Representation Vector

0	1	0	0	0	1	0	0	0	0	1	1		...		...		...	
---	---	---	---	---	---	---	---	---	---	---	---	--	-----	--	-----	--	-----	--

Logistic Regression

- Vector form
 - Inner product

$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_m x_m)}} \quad \Rightarrow \quad h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$


Need to learn this to correctly predict for $\mathbf{X}$ (i.e. image representation vector)

Maximum Likelihood Estimate

- Need to estimate θ .
 - Learn from data → Training pairs $(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N)$
- Log Likelihood Function

Probability Function

$$h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$

↓

Likelihood Function

$$\begin{aligned} L(\theta \mid x) &= \Pr(Y \mid X; \theta) \\ &= \prod_i \Pr(y_i \mid x_i; \theta) \\ &= \prod_i h_{\theta}(x_i)^{y_i} (1 - h_{\theta}(x_i))^{(1-y_i)} \end{aligned}$$

Y is a binary variable following the Bernoulli Distribution

Log Likelihood Function

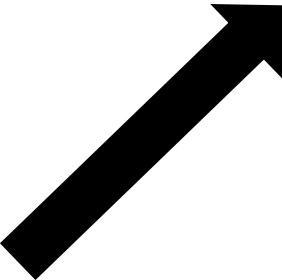
$$N^{-1} \log L(\theta \mid x) = N^{-1} \sum_{i=1}^N \log \Pr(y_i \mid x_i; \theta)$$

↓

Negative Log Likelihood Function

$$- \frac{1}{N} \sum_{n=1}^N \left[y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n) \right]$$

Negative Log Likelihood (NLL) is the same as the Cross Entropy loss



Maximum Likelihood Estimate

- Need to estimate θ .
 - Learn from data → Training pairs $(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N)$
- Log Likelihood Function

Probability Function

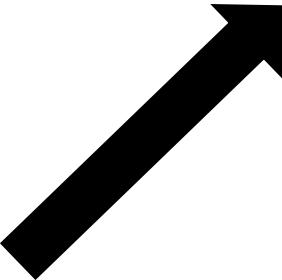
$$h_{\theta}(X) = \frac{1}{1 + e^{-\theta^T X}} = \Pr(Y = 1 \mid X; \theta)$$

↓

Likelihood Function

$$\begin{aligned} L(\theta \mid x) &= \Pr(Y \mid X; \theta) \\ &= \prod_i \Pr(y_i \mid x_i; \theta) \\ &= \prod_i h_{\theta}(x_i)^{y_i} (1 - h_{\theta}(x_i))^{(1-y_i)} \end{aligned}$$

Y is a binary variable following the Bernoulli Distribution



Log Likelihood Function

$$N^{-1} \log L(\theta \mid x) = N^{-1} \sum_{i=1}^N \log \Pr(y_i \mid x_i; \theta)$$

↓

Negative Log Likelihood Function

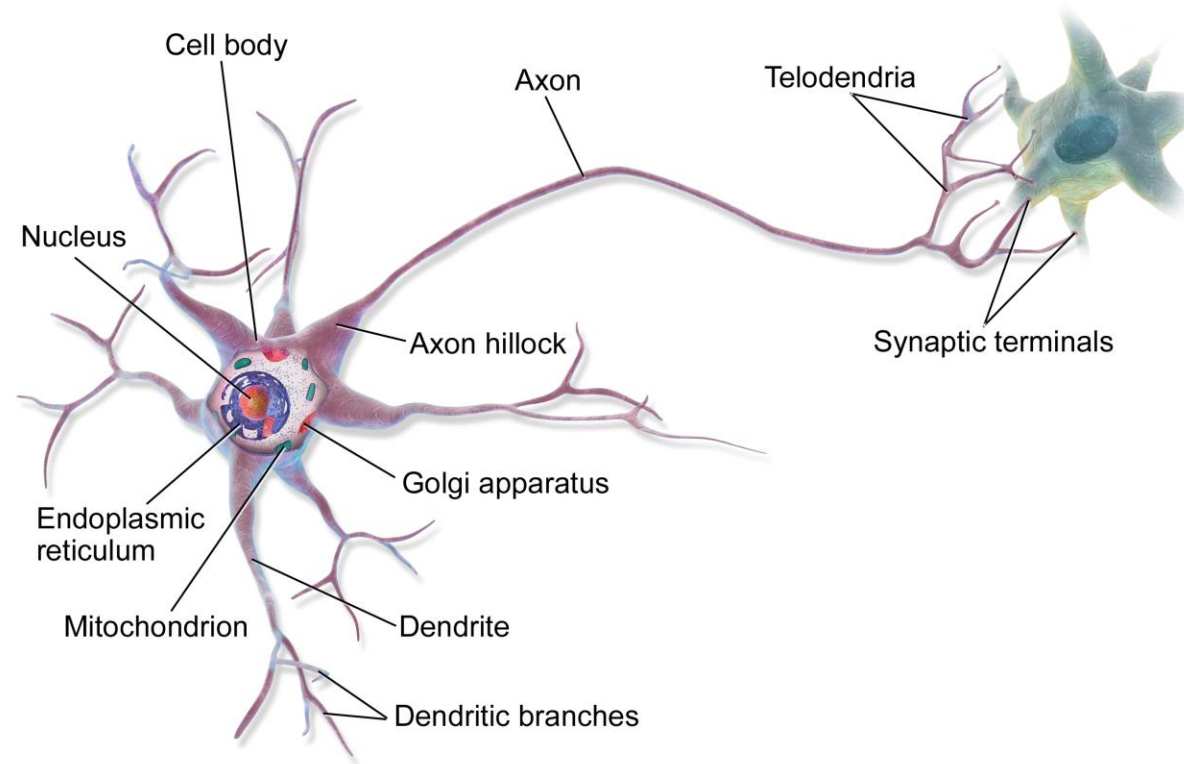
$$- \frac{1}{N} \sum_{n=1}^N \left[y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n) \right]$$

Minimize the negative log likelihood (NLL)
by Gradient Descent

Neural Networks

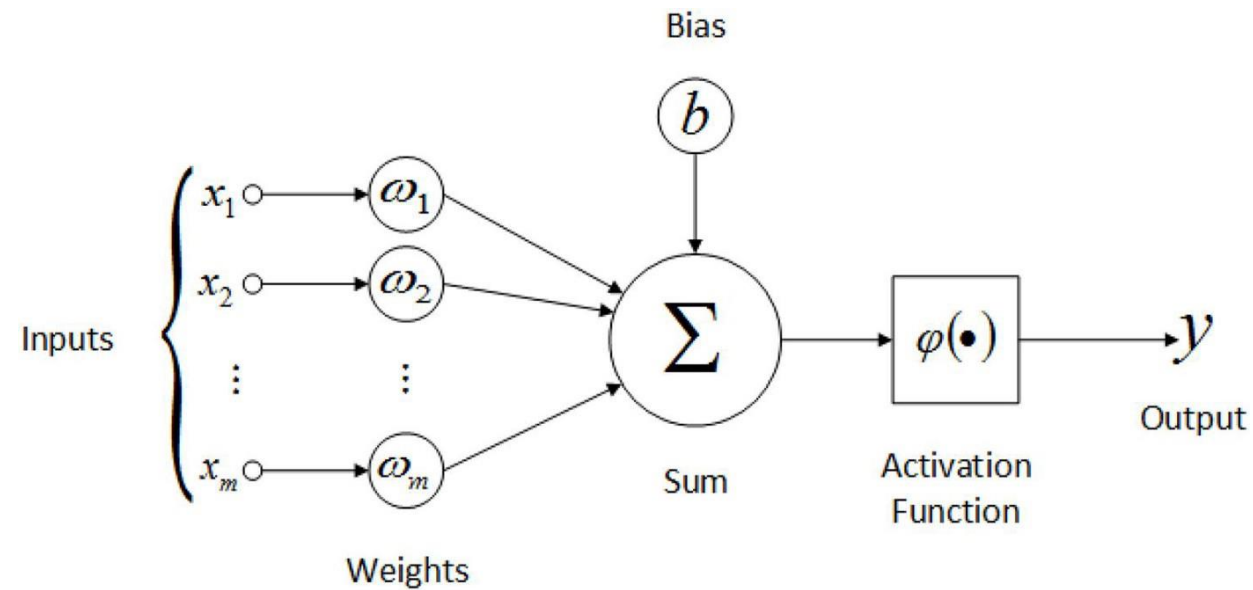
Neural Networks

- Neuron



Neural Networks

- Artificial neuron

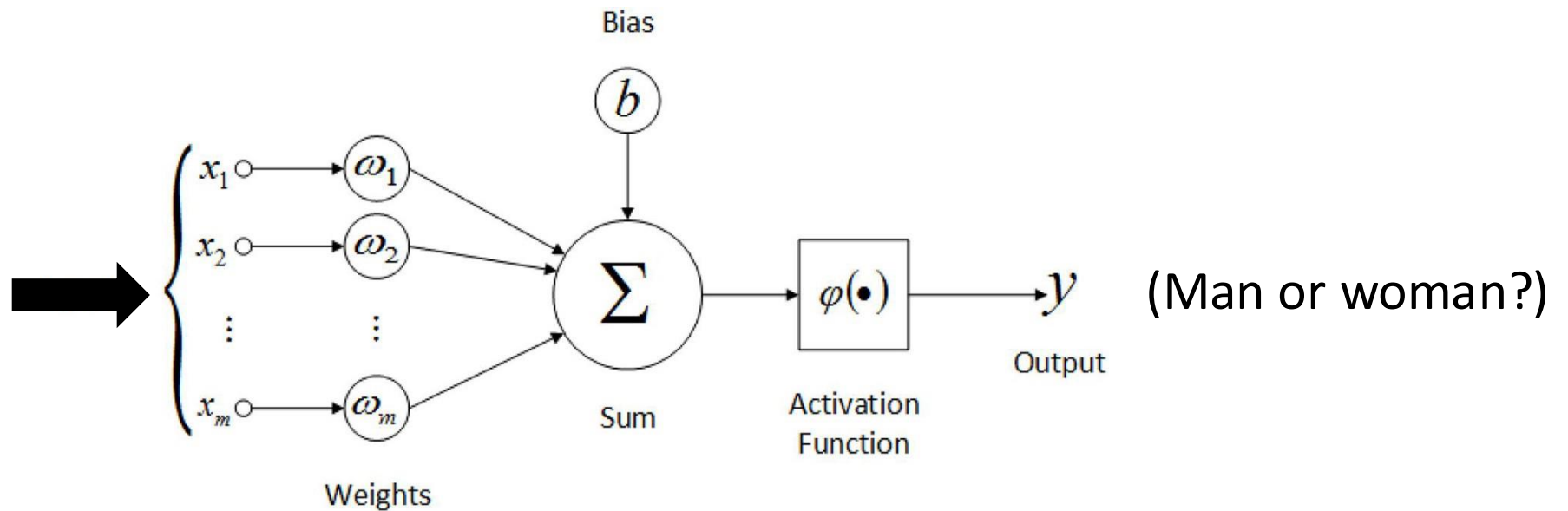


Neural Networks

- Artificial neuron

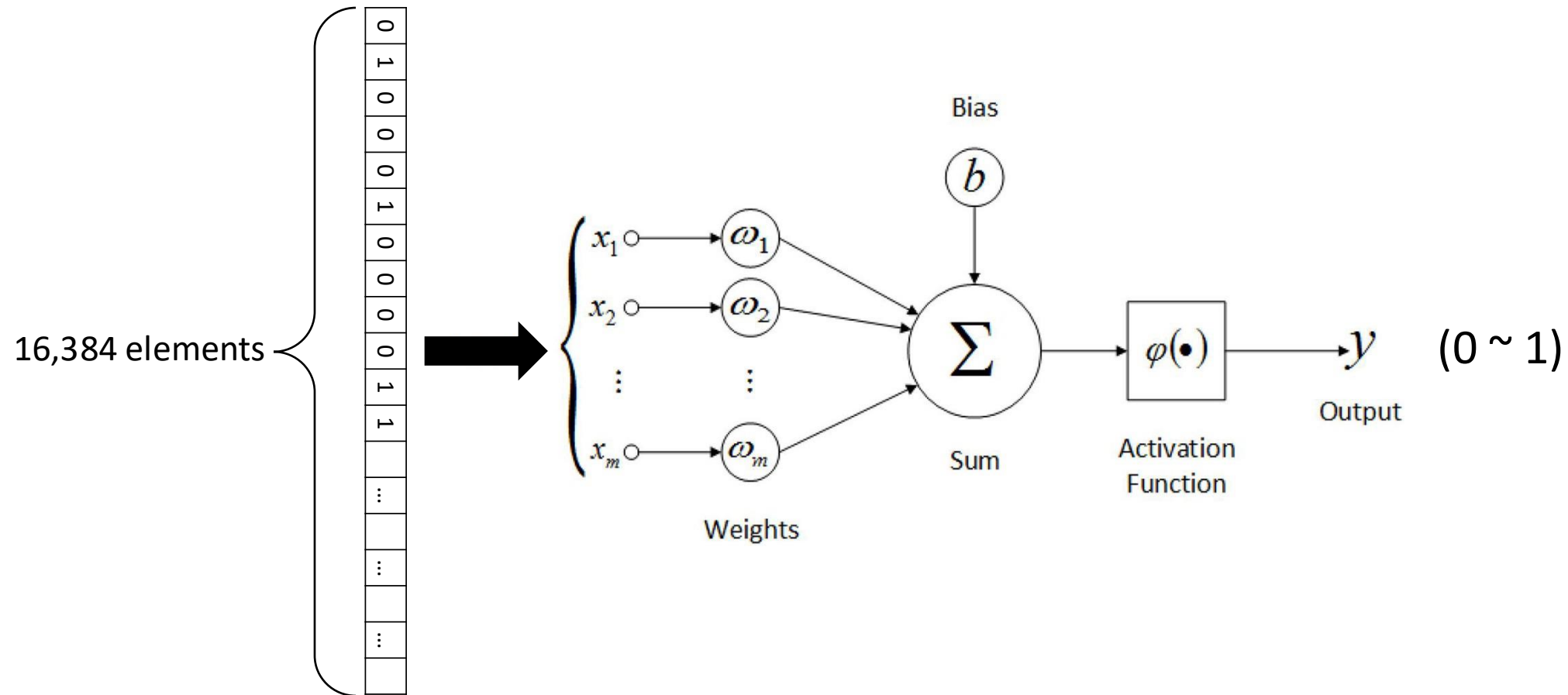


128 X 128 Image



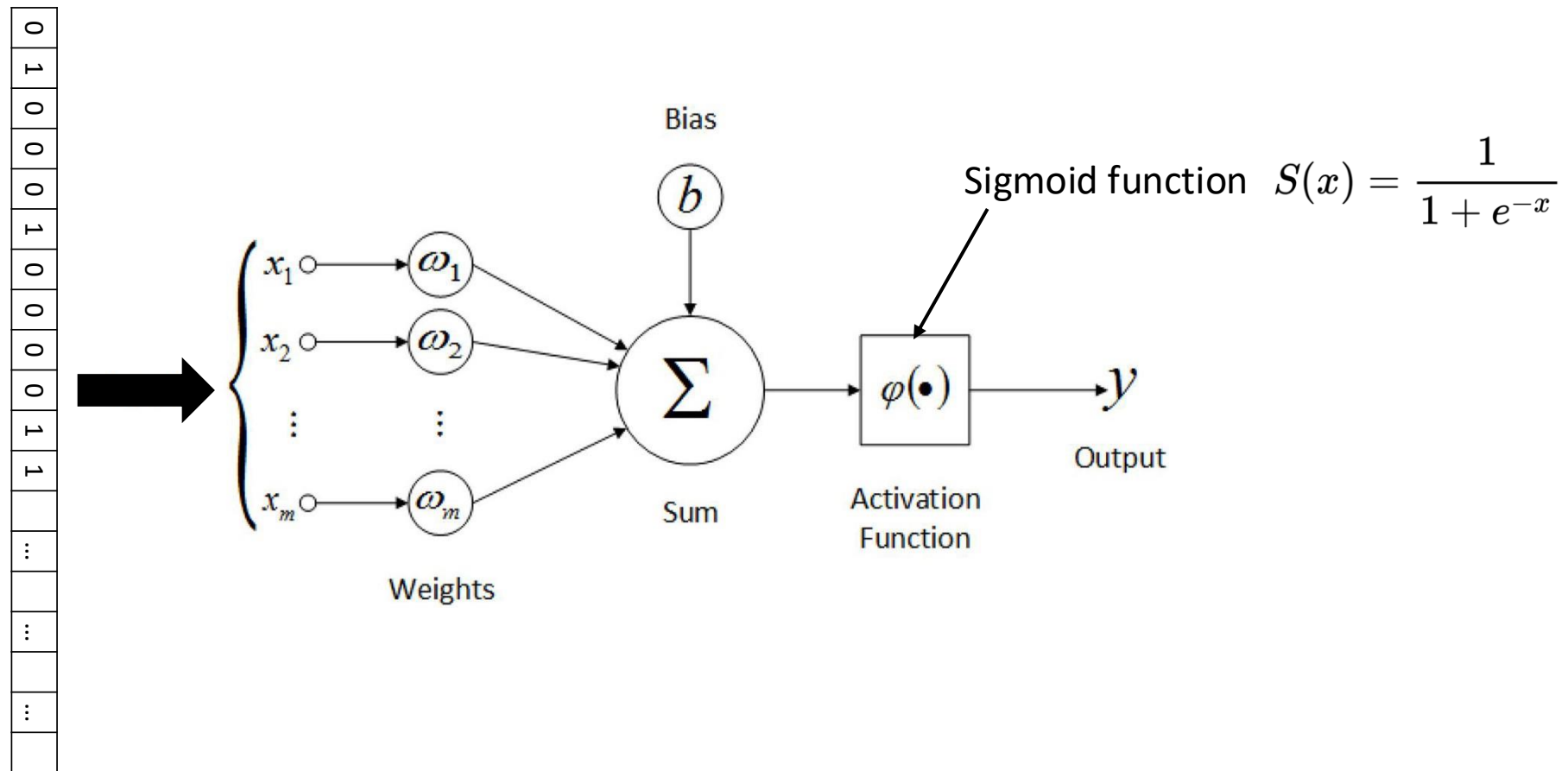
Neural Networks

- Artificial neuron



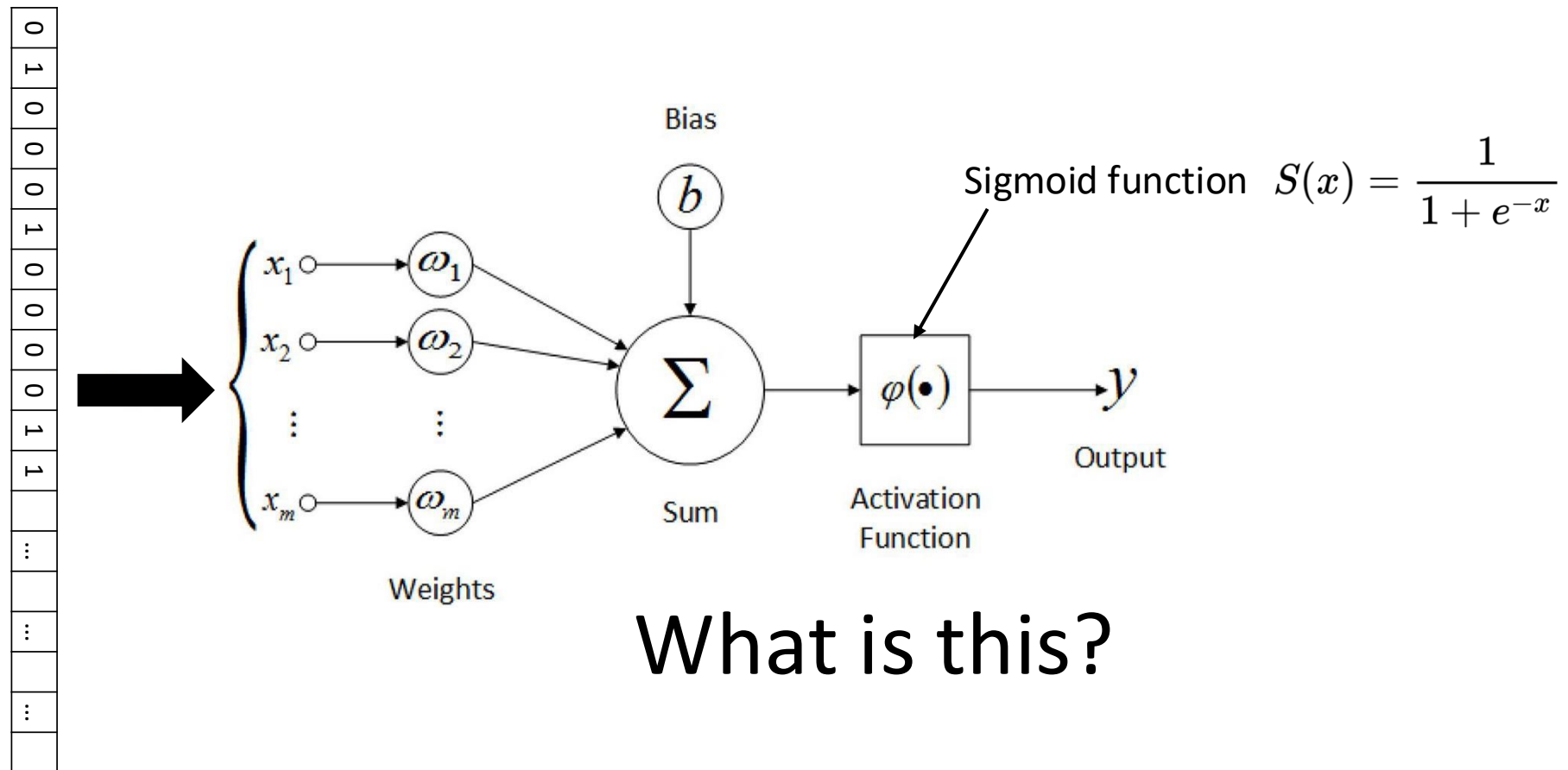
Neural Networks

- Artificial neuron



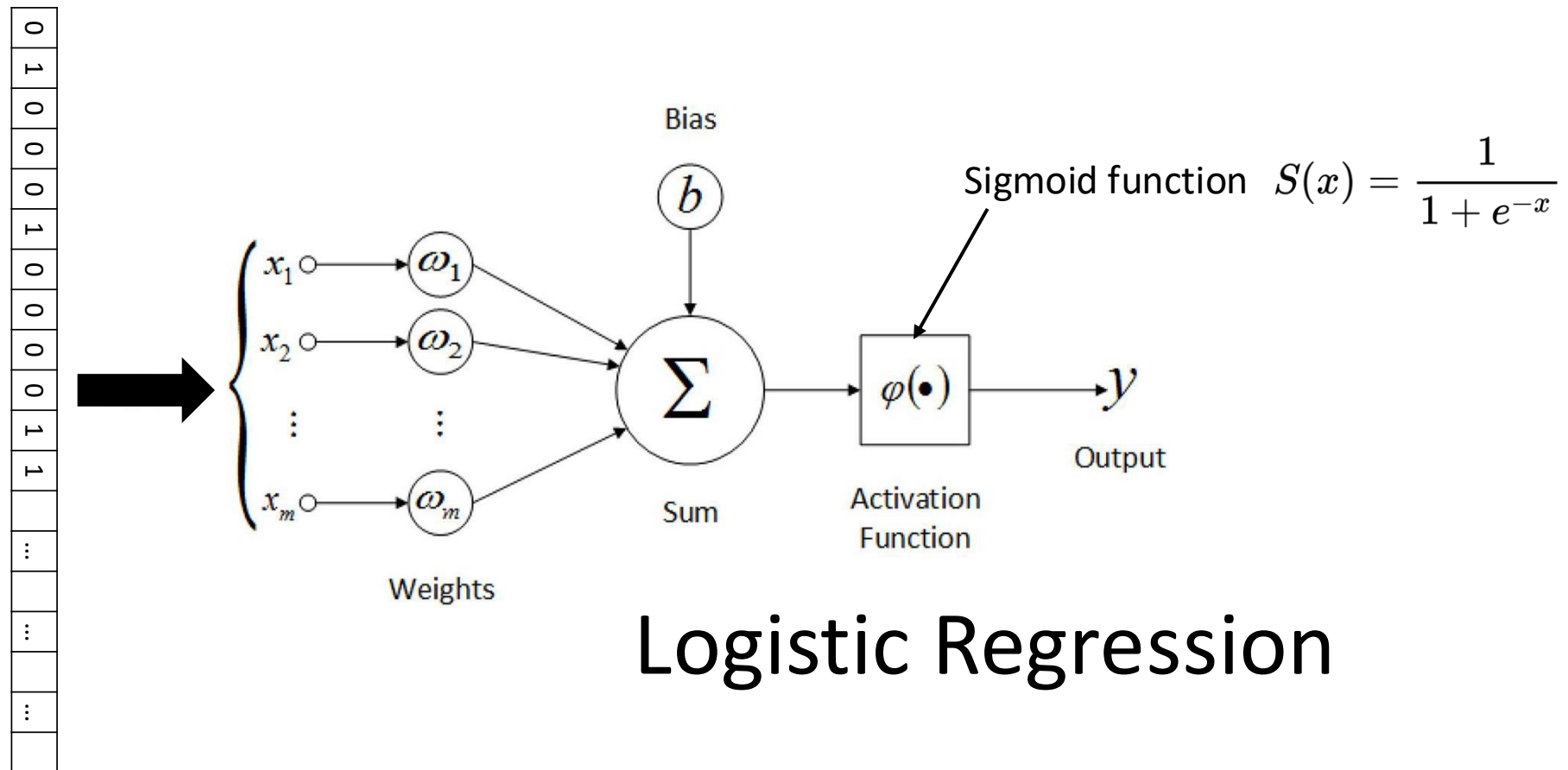
Neural Networks

- Artificial neuron



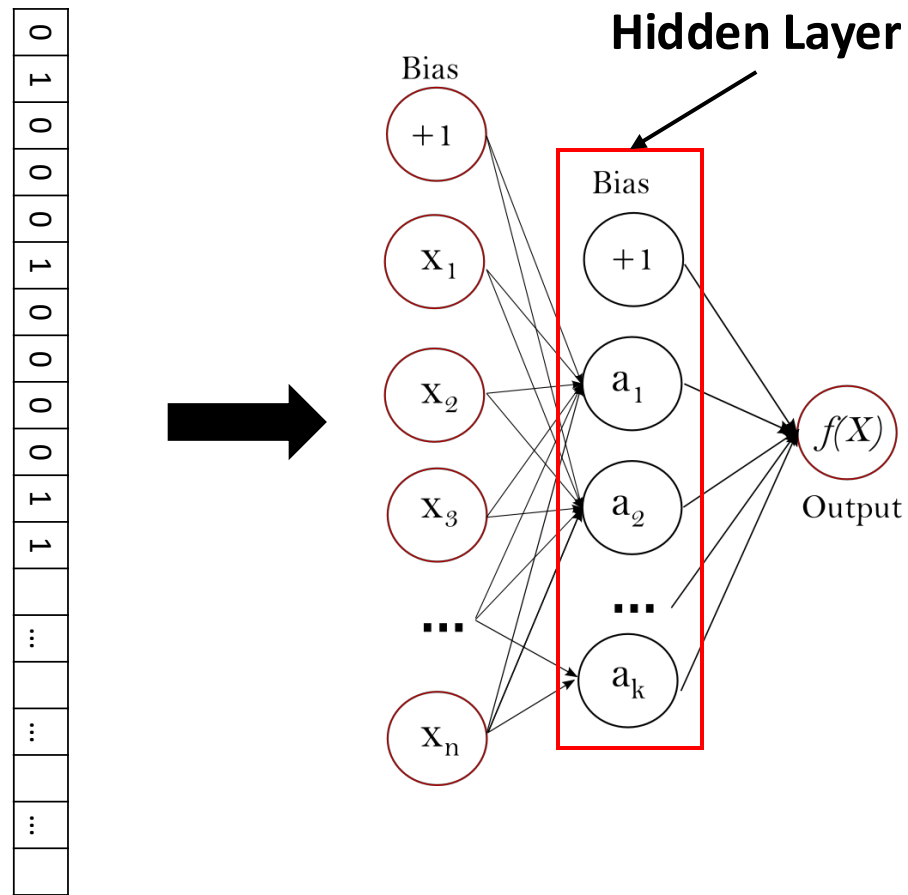
Neural Networks

- Logistic Regression



Neural Networks

- Another layer of neurons.



Neural Networks

- Another layer of neurons.

0
1
0
0
0
1
0
0
0
0
1
1
...
...
...

h_1

h_2

h_3

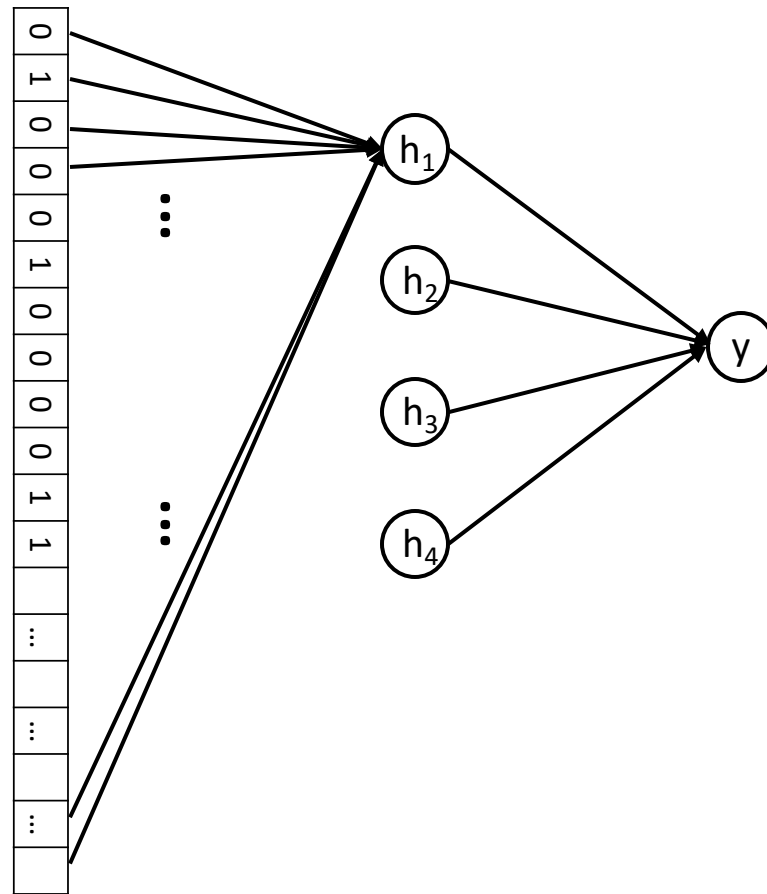
h_4

y

Example: Feedforward neural network with one hidden layer of 4 neurons.
Assume hidden layer activation function → [sigmoid function](#)

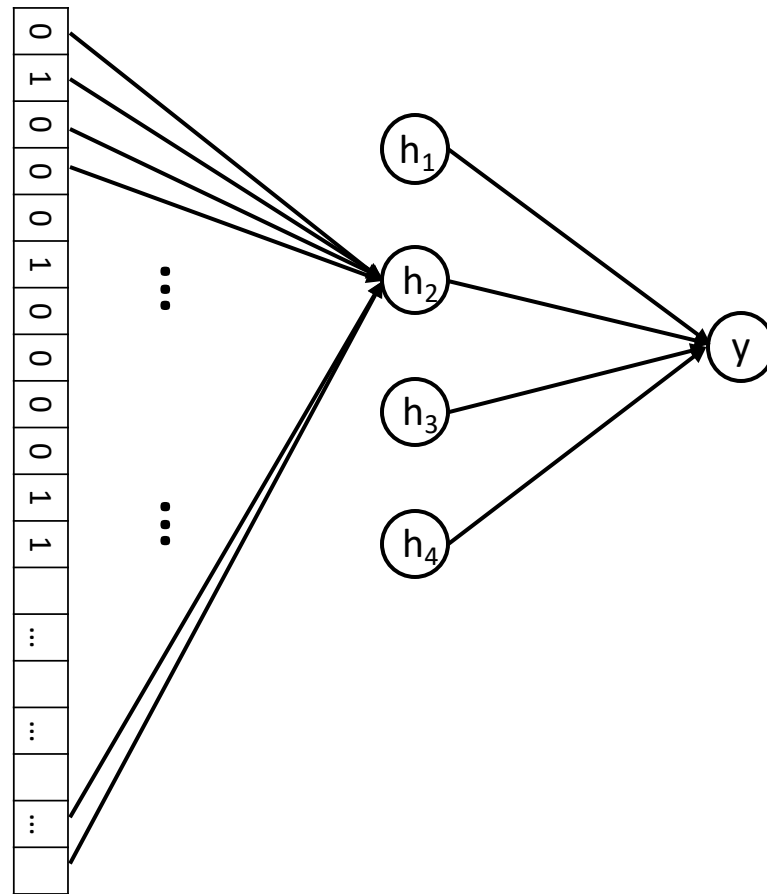
Neural Networks

- Another layer of neurons.



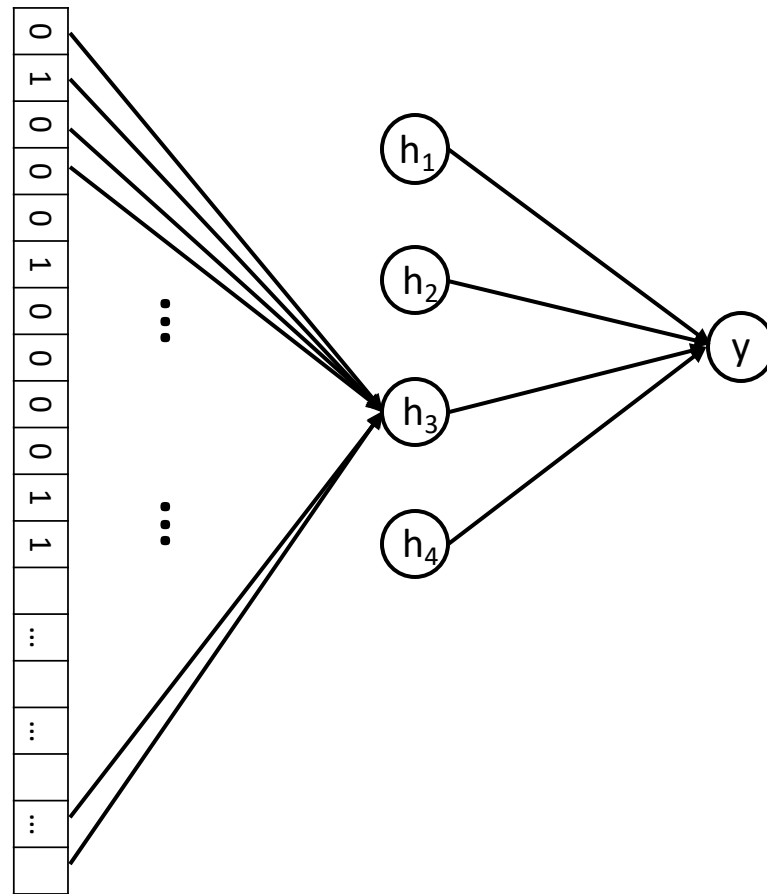
Neural Networks

- Another layer of neurons.



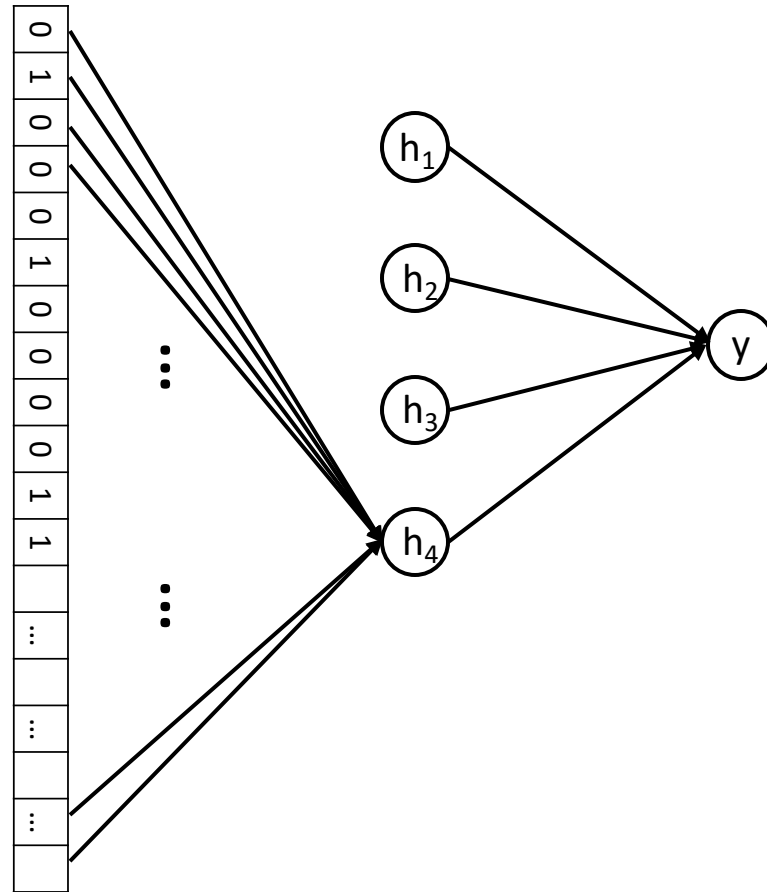
Neural Networks

- Another layer of neurons.



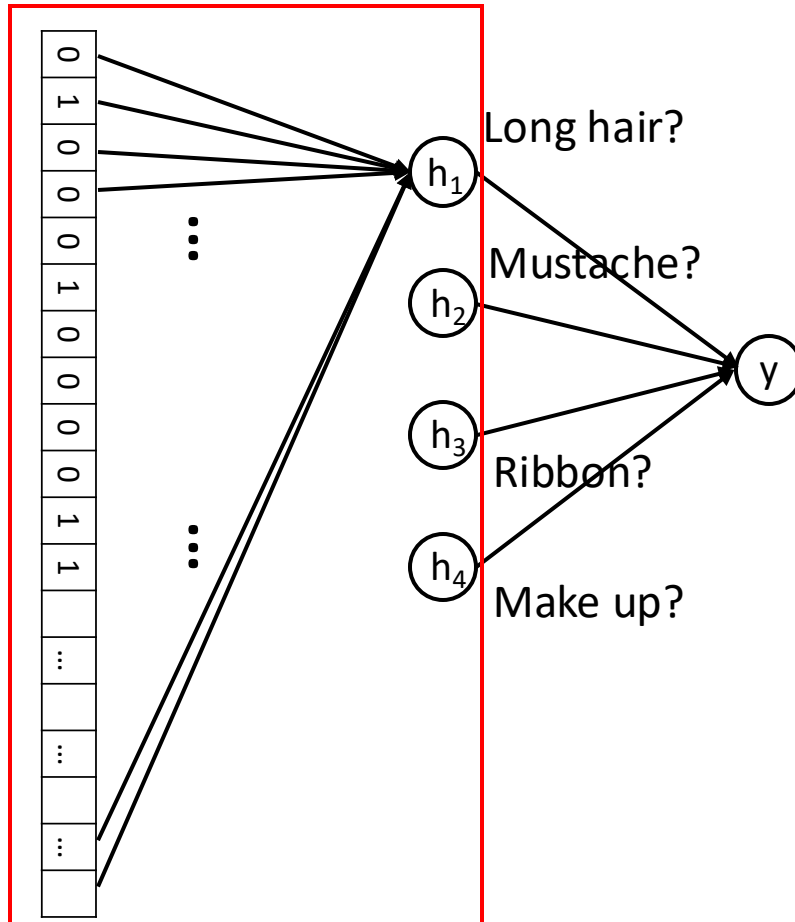
Neural Networks

- Another layer of neurons.



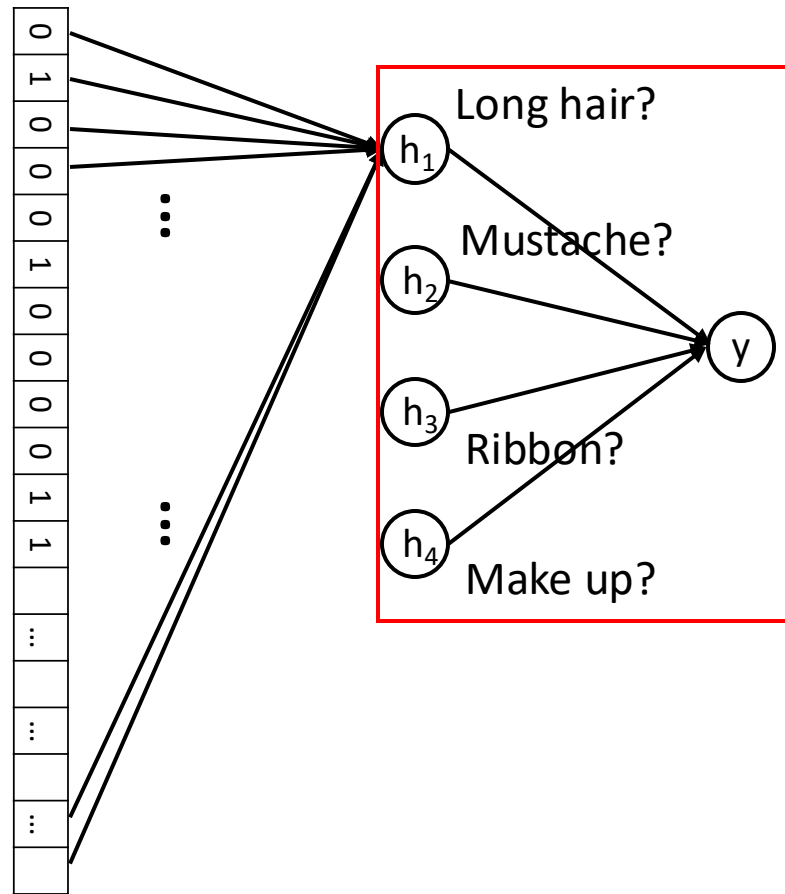
Neural Networks

- 4 logistic regression classifiers (when using sigmoid activation)



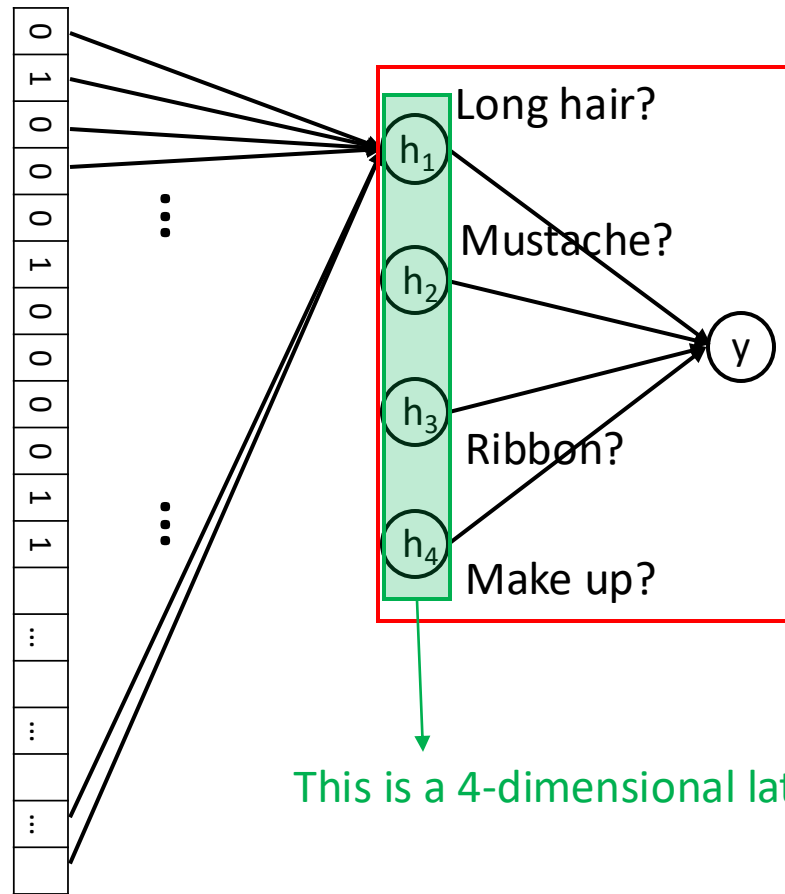
Neural Networks

- Higher-level logistic regression classifiers



Neural Networks

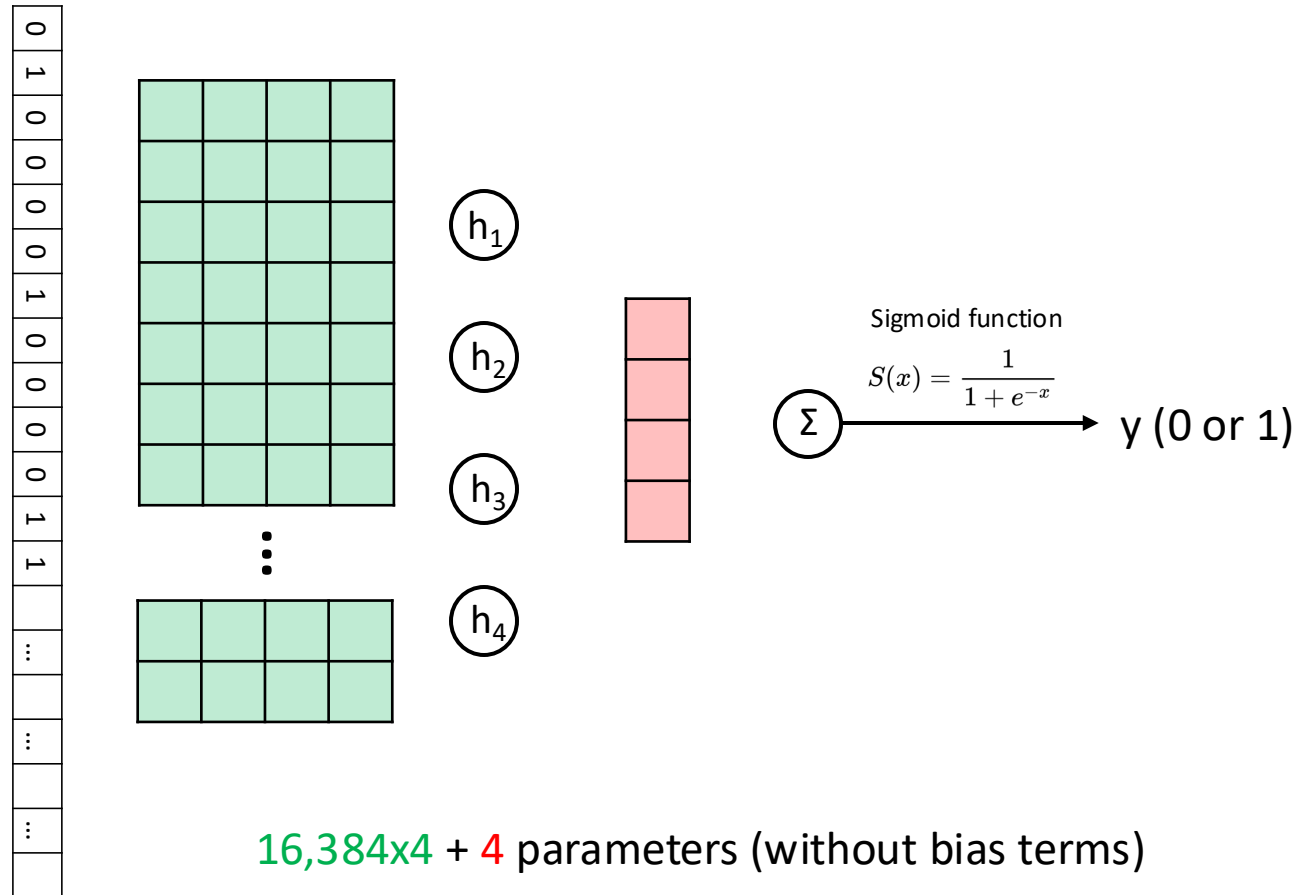
- Higher-level logistic regression classifiers



This is a 4-dimensional latent representation vector for the given image.

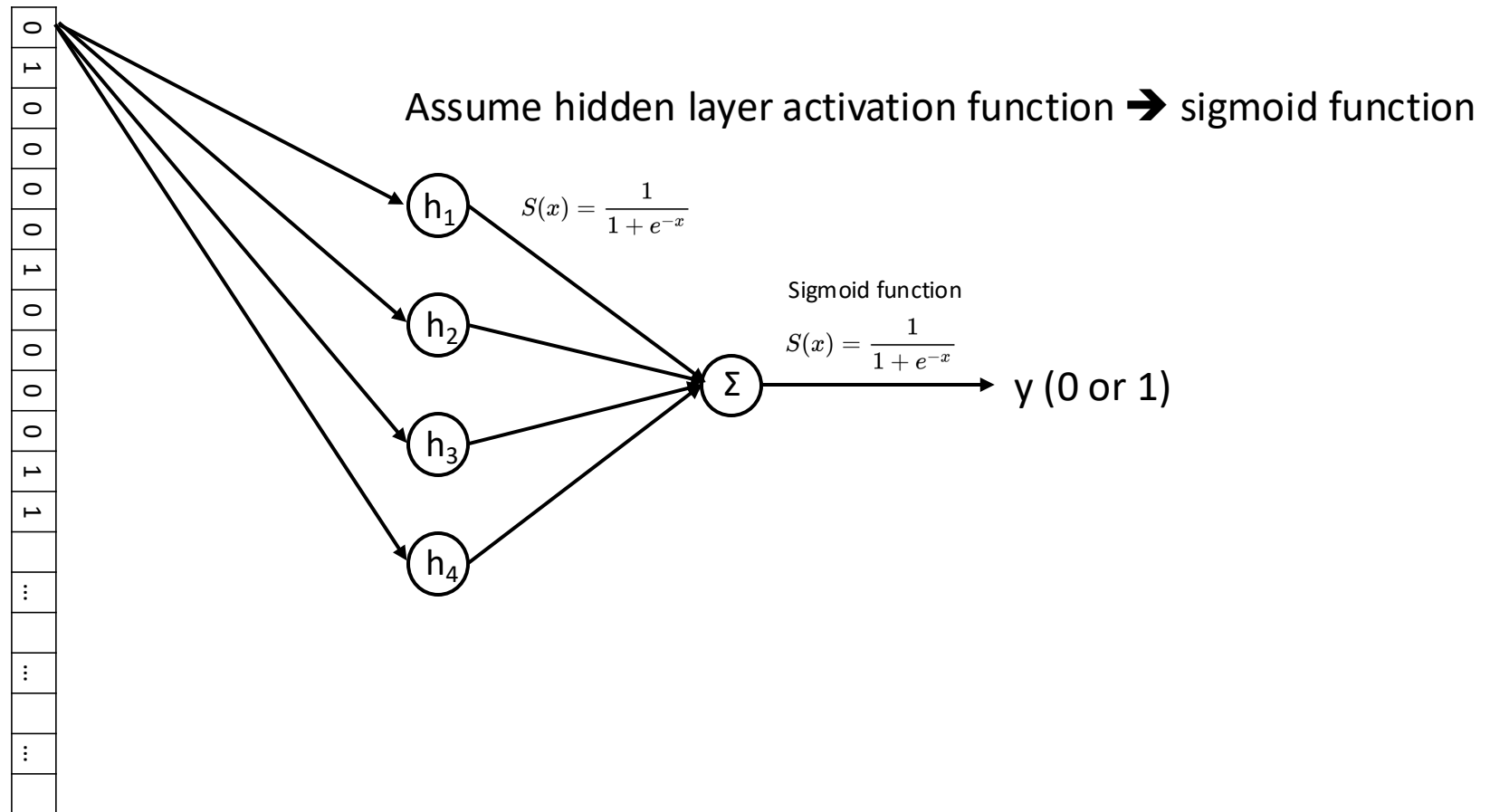
Training

- Need to estimate all parameter values



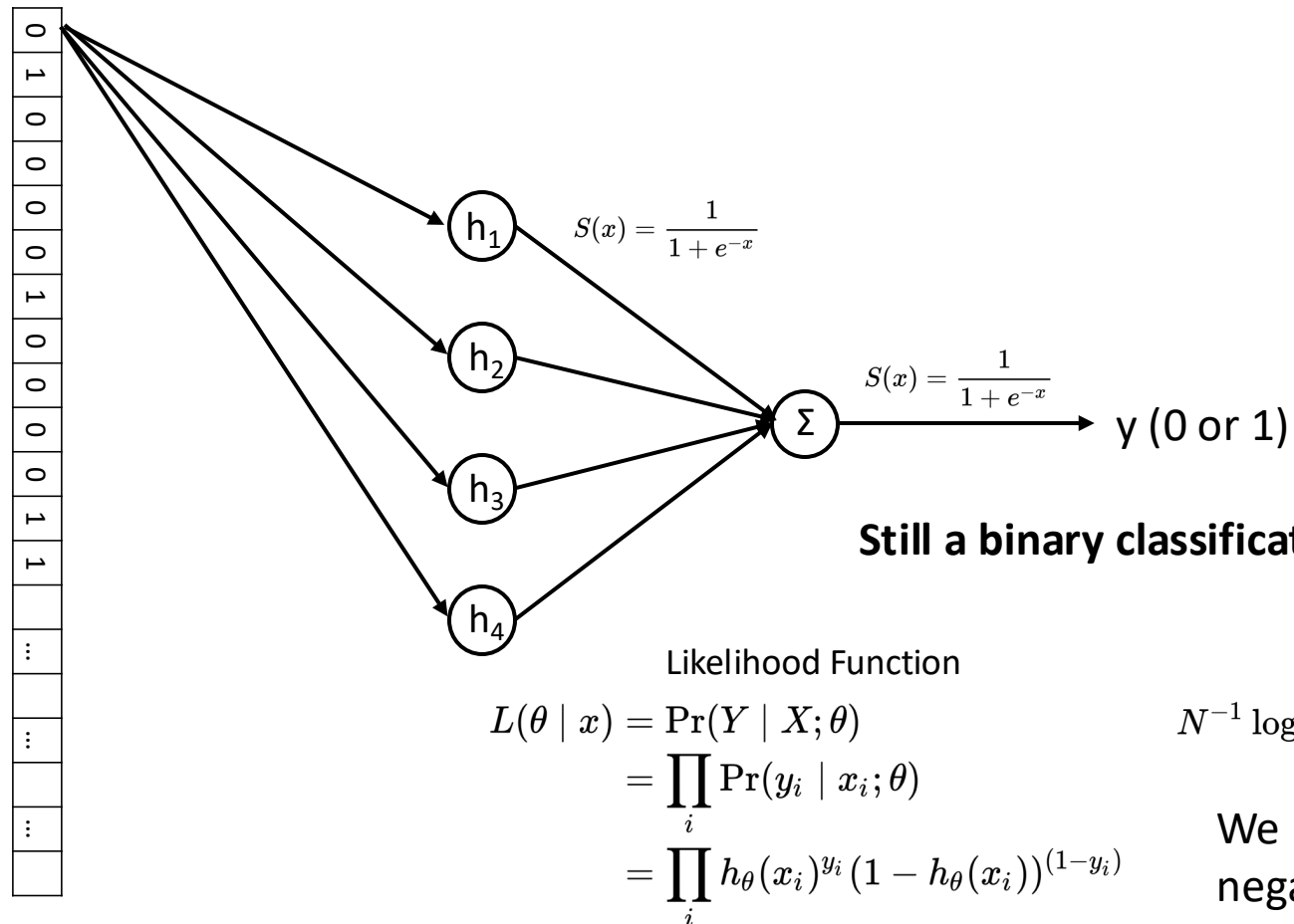
Training

- Need to estimate all parameter values



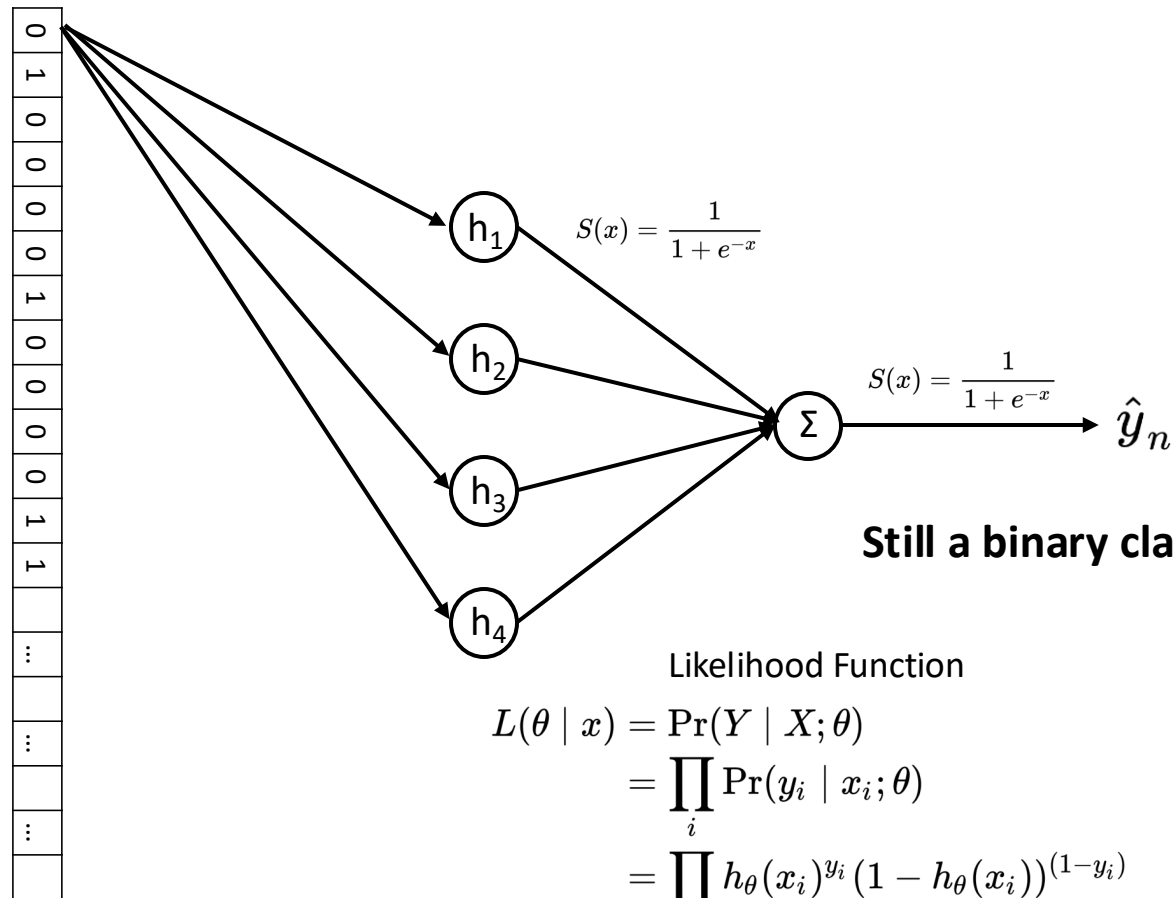
Training

- Maximum likelihood estimation



Training

- Minimize NLL with gradient descent.



Likelihood Function

$$\begin{aligned}
 L(\theta | x) &= \Pr(Y | X; \theta) \\
 &= \prod_i \Pr(y_i | x_i; \theta) \\
 &= \prod_i h_\theta(x_i)^{y_i} (1 - h_\theta(x_i))^{(1-y_i)}
 \end{aligned}$$

Log Likelihood Function

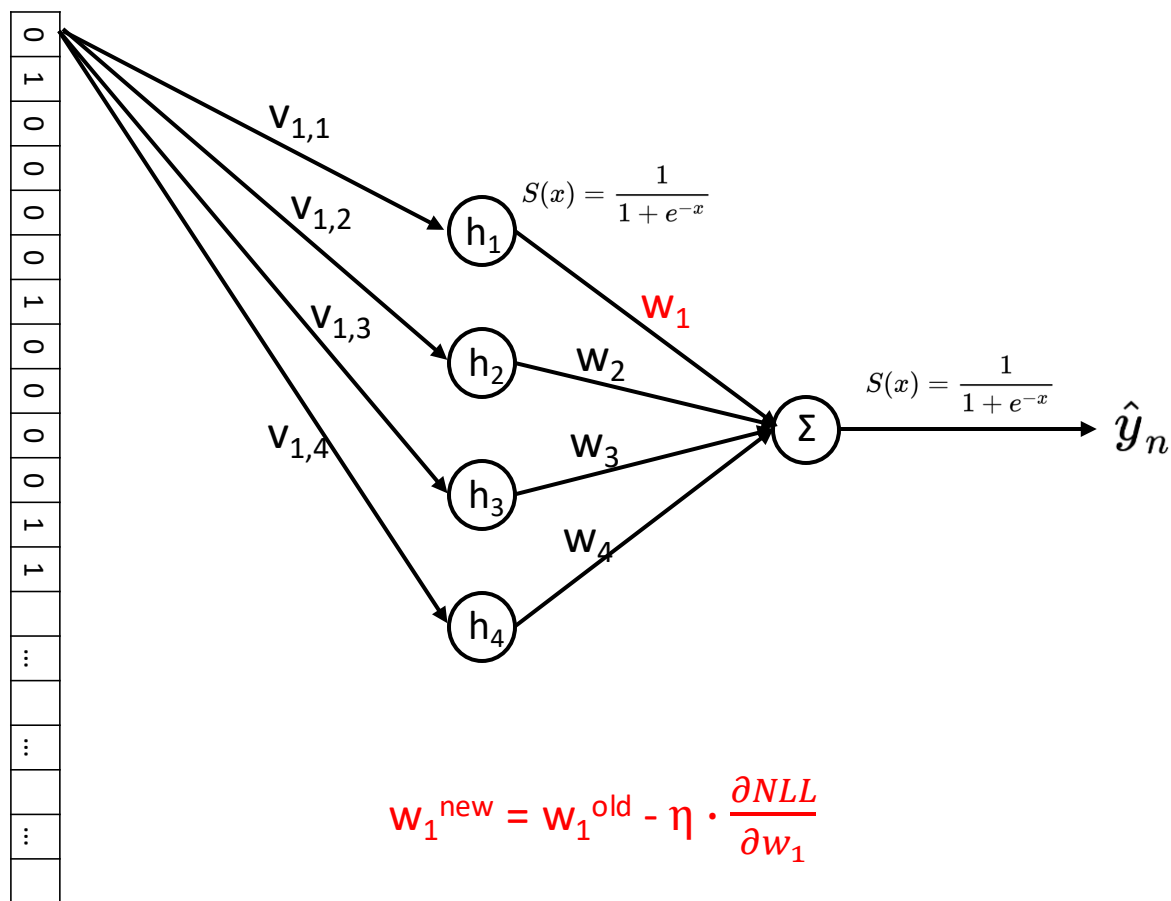
$$N^{-1} \log L(\theta | x) = N^{-1} \sum_{i=1}^N \log \Pr(y_i | x_i; \theta)$$

We need to minimize

$$-\frac{1}{N} \sum_{n=1}^N \left[y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n) \right]$$

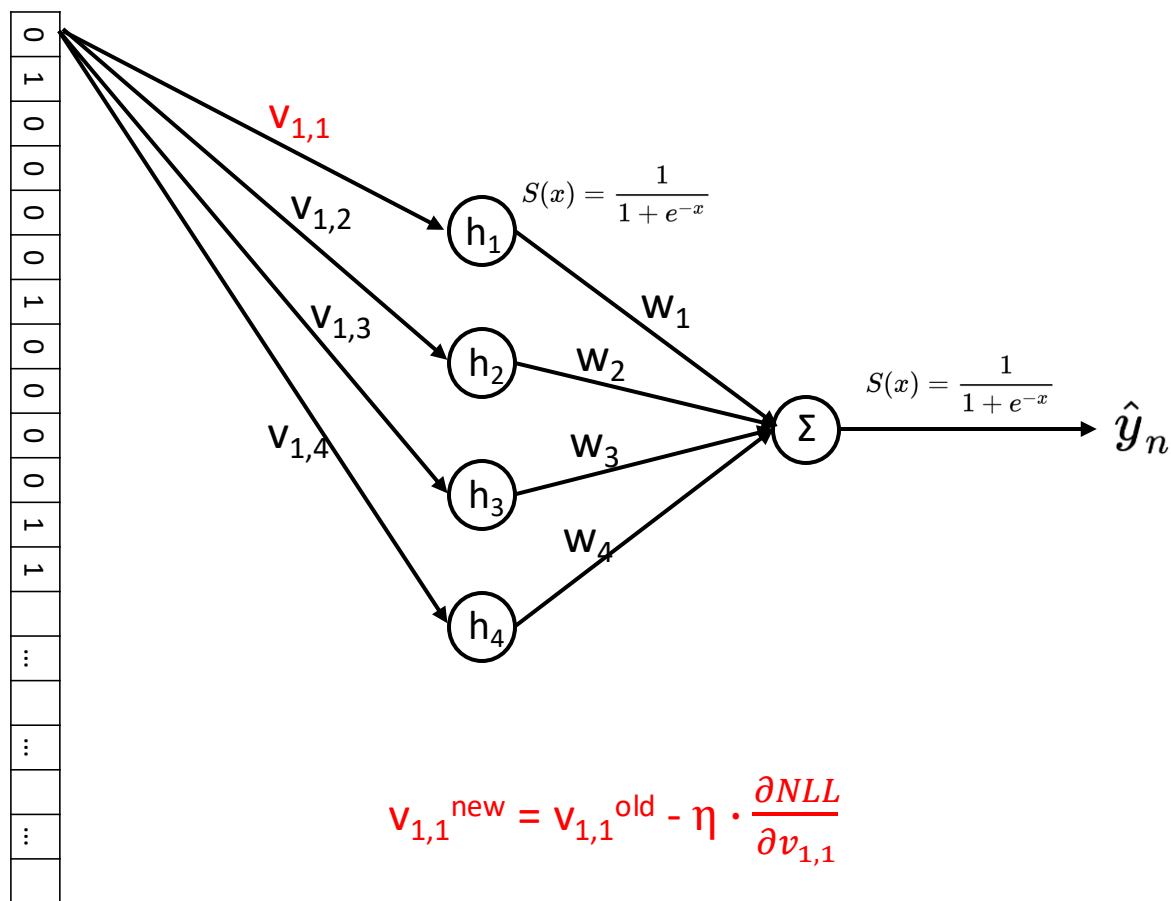
Training

- Updating w_1



Training

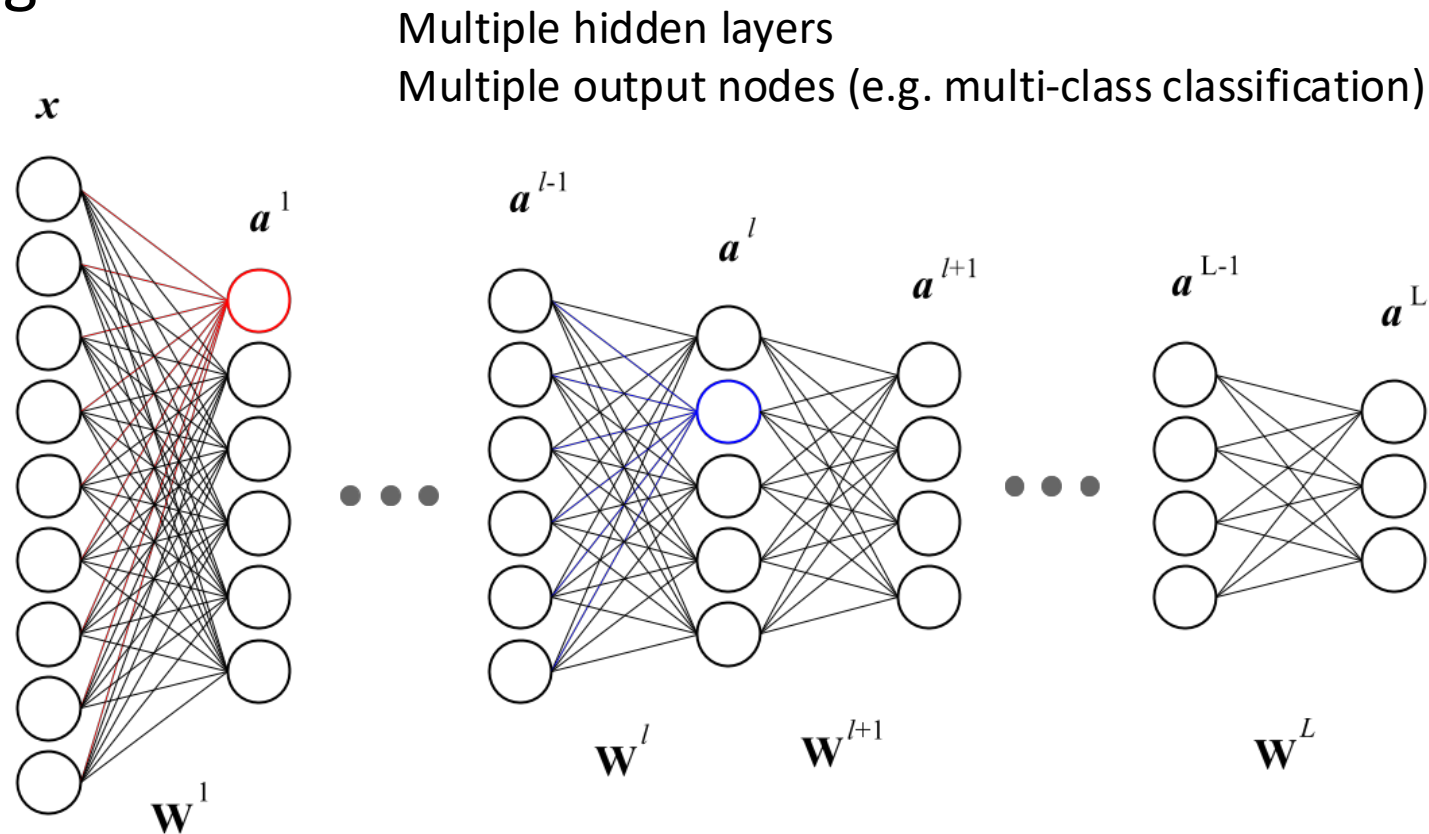
- Updating $v_{1,1}$



Backpropagation

Training

- Backpropagation

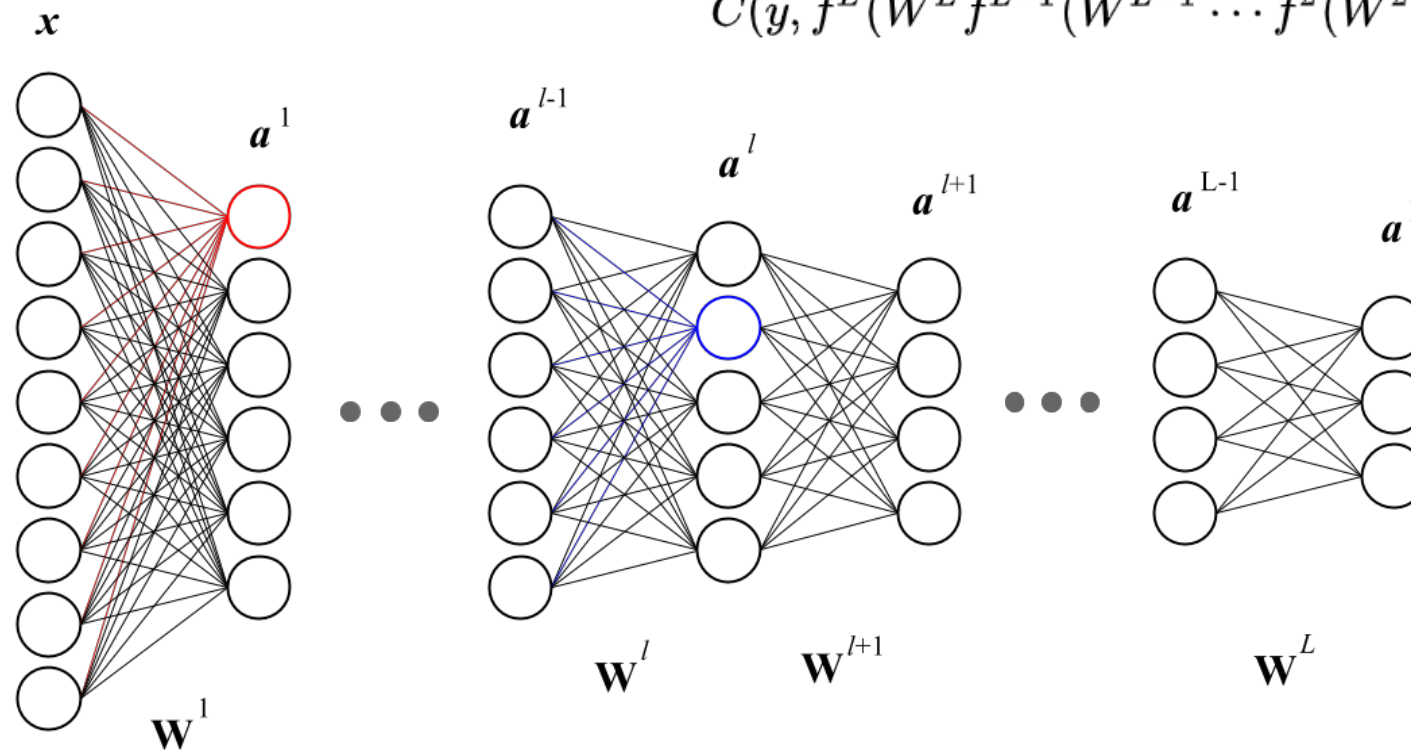


Training

- Backpropagation

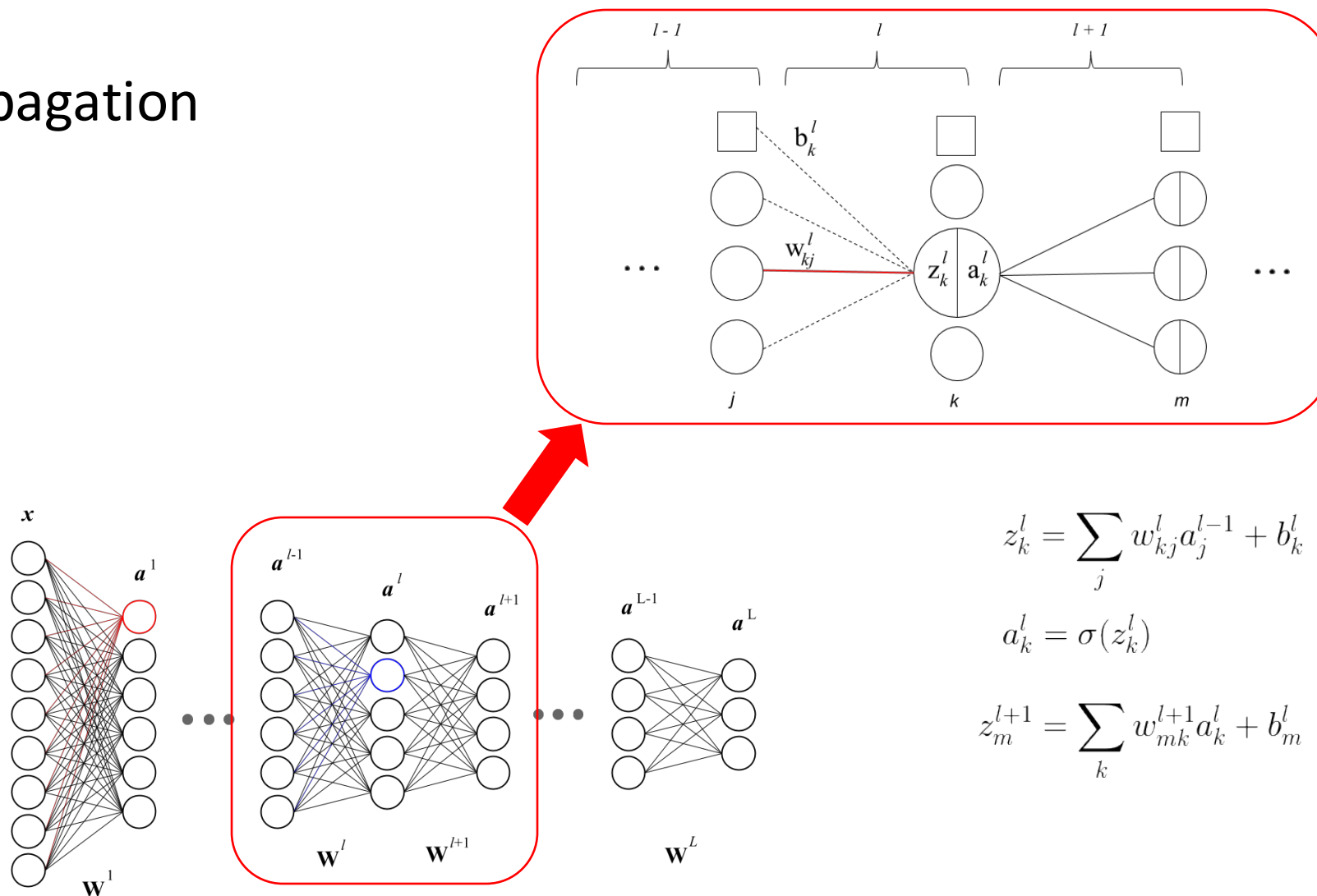
Given an input-output pair (x, y) , the loss is:

$$C(y, f^L(W^L f^{L-1}(W^{L-1} \dots f^2(W^2 f^1(W^1 x)) \dots)))$$



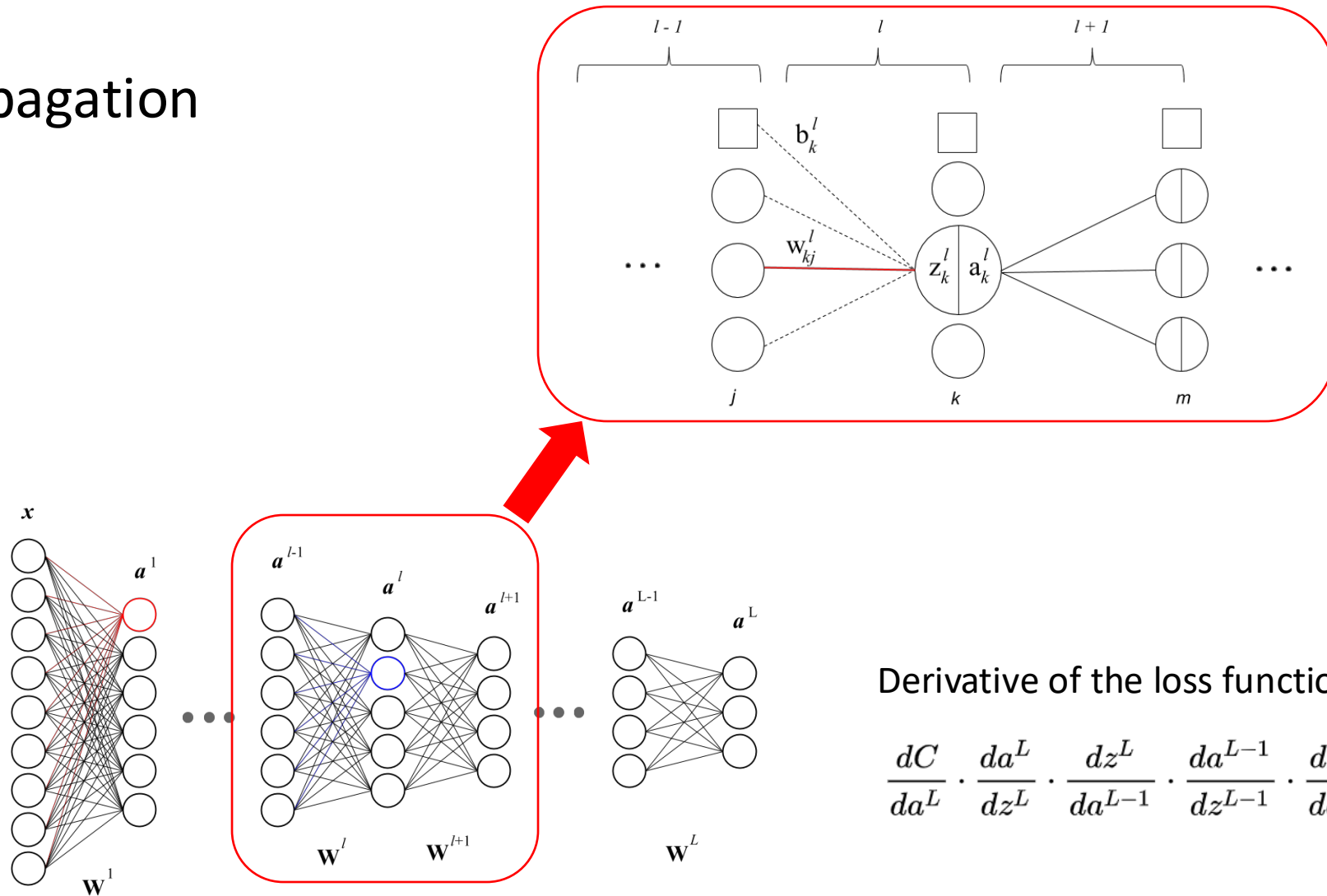
Training

- Backpropagation



Training

- Backpropagation

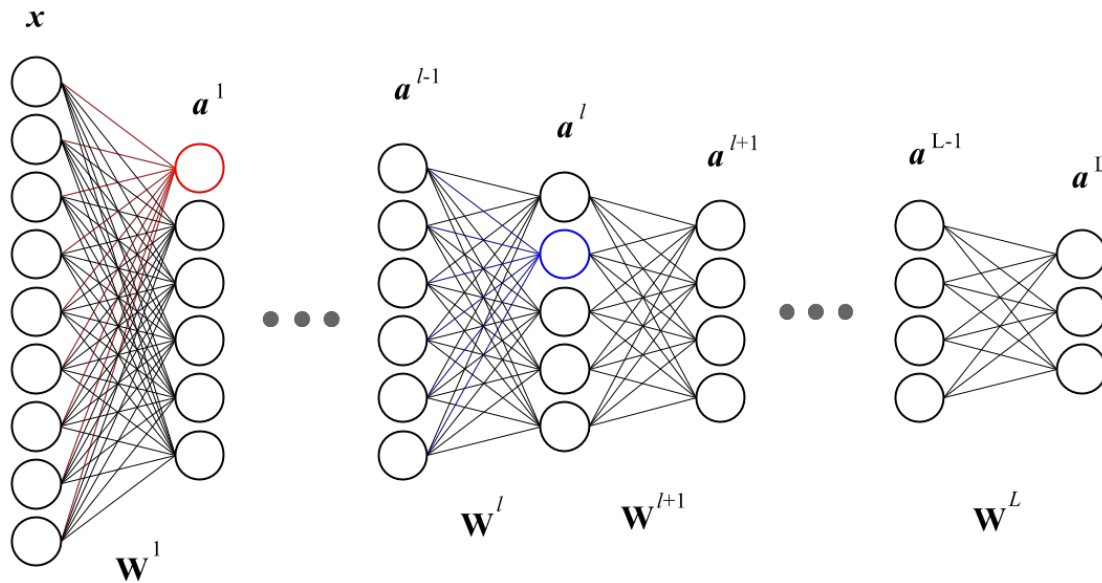


Derivative of the loss function C w.r.t. input x

$$\frac{dC}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x}$$

Training

- Backpropagation



Derivative of the loss function C w.r.t. input x

$$\frac{dC}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x}$$



$$\frac{dC}{da^L} \cdot (f^L)' \cdot W^L \cdot (f^{L-1})' \cdot W^{L-1} \cdots (f^1)' \cdot W^1$$



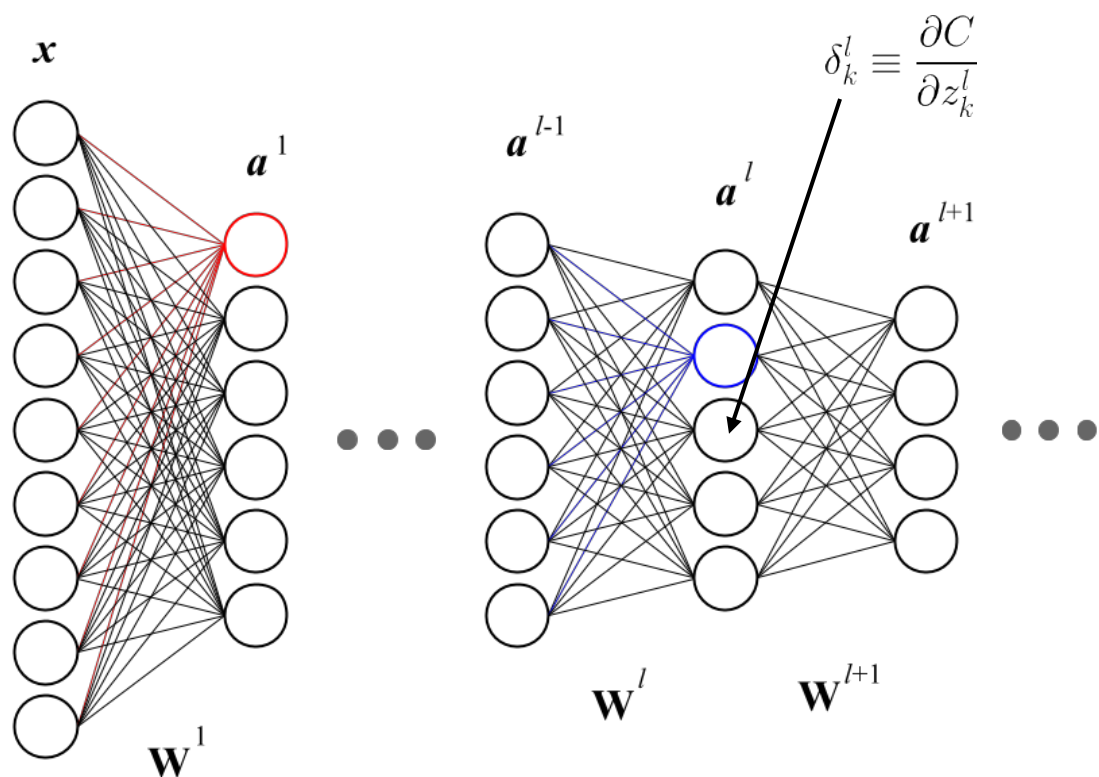
$$\nabla_x C = (W^1)^T \cdot (f^1)' \cdots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Define **delta** (error at layer l)

$$\delta^l := (f^l)' \cdot (W^{l+1})^T \cdots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Training

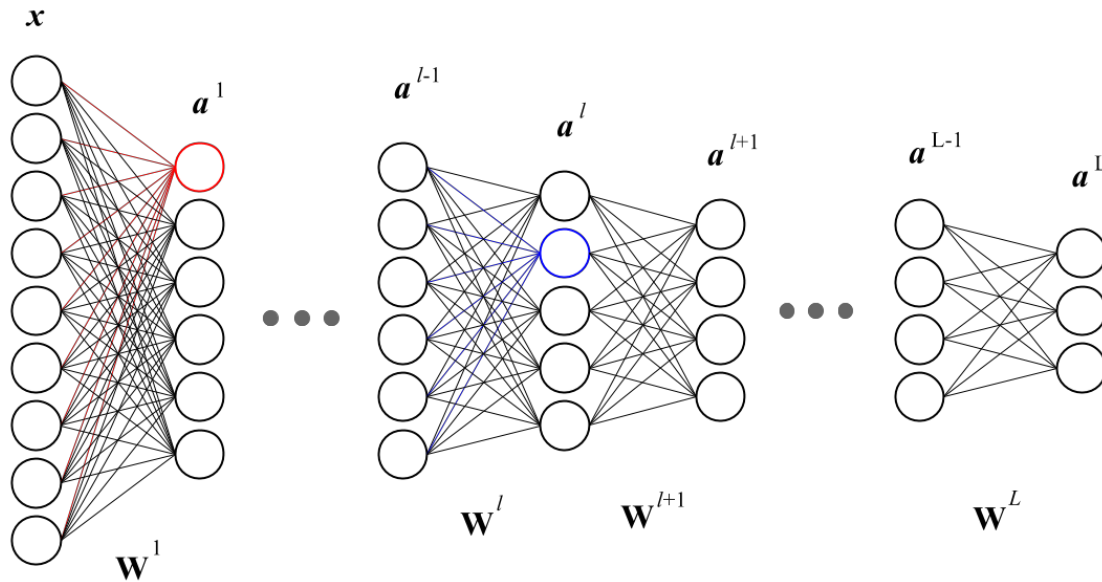
- Backpropagation



Define *delta* at each neuron
 δ_k^l : How much the cost (loss) function changes
when the input to the k -th neuron at layer l changes.

Training

- Backpropagation



Derivative of the loss function C w.r.t. input x

$$\nabla_x C = (W^1)^T \cdot (f^1)' \dots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Define delta (error at layer l)

$$\delta^l := (f^l)' \cdot (W^{l+1})^T \dots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Deltas at each layer

$$\delta^1 = (f^1)' \cdot (W^2)^T \cdot (f^2)' \dots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

$$\delta^2 = (f^2)' \dots (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

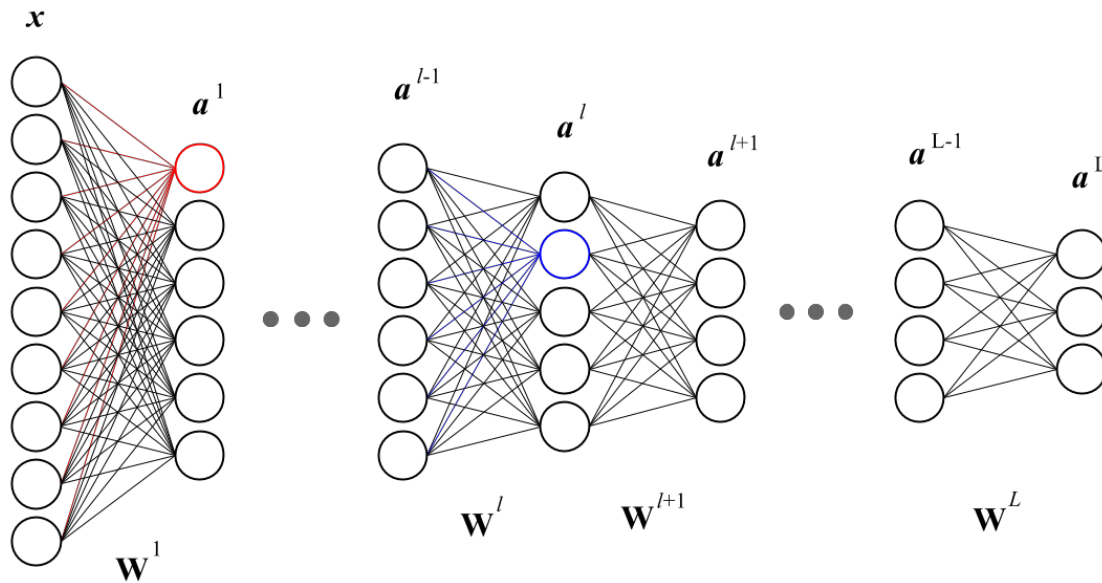
$\vdots$

$$\delta^{L-1} = (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

$$\delta^L = (f^L)' \cdot \nabla_{a^L} C,$$

Training

- Backpropagation



Derivative of the loss function C w.r.t. input x

$$\nabla_x C = (W^1)^T \cdot (f^1)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Define delta (error at layer l)

$$\delta^l := (f^l)' \cdot (W^{l+1})^T \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Deltas at each layer

$$\delta^1 = (f^1)' \cdot (W^2)^T \cdot (f^2)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

$$\delta^2 = (f^2)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

$\vdots$

$$\delta^{L-1} = (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

$$\delta^L = (f^L)' \cdot \nabla_{a^L} C,$$

Delta can be **recursively** defined!

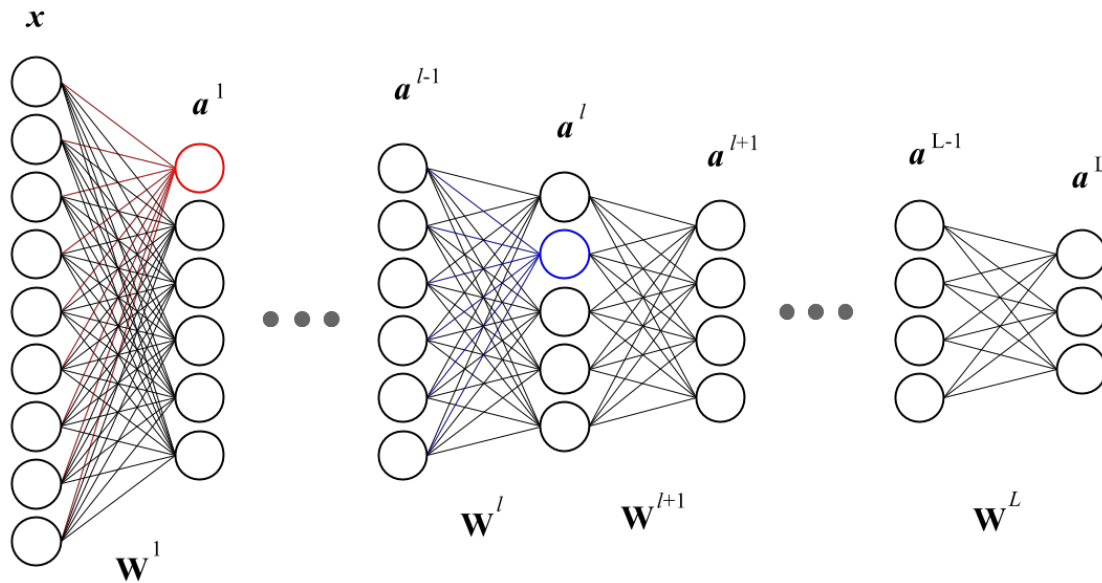
$$\delta^{l-1} := (f^{l-1})' \cdot (W^l)^T \cdot \delta^l$$

Derivative of C w.r.t weights can be defined via delta

$$\nabla_{W^l} C = \delta^l (a^{l-1})^T \quad \text{Needed for } \rightarrow w_1^{\text{new}} = w_1^{\text{old}} - \eta \cdot \frac{\partial NLL}{\partial w_1}$$

Training

- Backpropagation



Backprop is faster than forward-prop!
 But need to store $(f^l)'$, a_k^l for **every node** in **every layer**.

Derivative of the loss function C w.r.t. input x

$$\nabla_x C = (W^1)^T \cdot (f^1)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Define delta (error at layer l)

$$\delta^l := (f^l)' \cdot (W^{l+1})^T \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C$$

Deltas at each layer

$$\begin{aligned} \delta^1 &= (f^1)' \cdot (W^2)^T \cdot (f^2)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C \\ \delta^2 &= (f^2)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C \\ &\vdots \\ \delta^{L-1} &= (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} C \\ \delta^L &= (f^L)' \cdot \nabla_{a^L} C, \end{aligned}$$

Delta can be **recursively** defined!

$$\delta^{l-1} := (f^{l-1})' \cdot (W^l)^T \cdot \delta^l$$

Derivative of C w.r.t weights can be defined via delta

$$\nabla_{W^l} C = \delta^l (a^{l-1})^T \quad \text{Needed for } \rightarrow w_1^{\text{new}} = w_1^{\text{old}} - \eta \cdot \frac{\partial NLL}{\partial w_1}$$

Autograd (in PyTorch)

In Practice

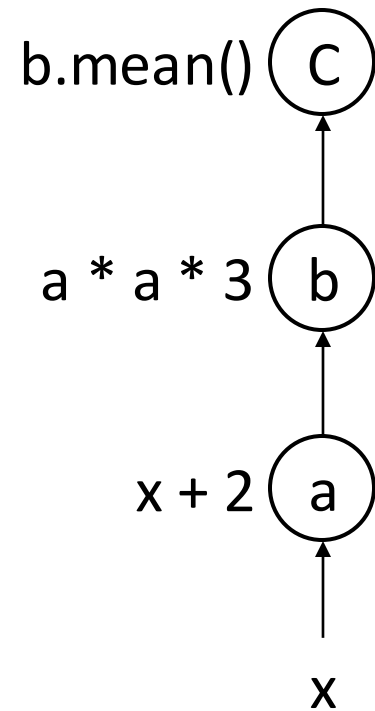
- Computer does backpropagation for you.
 - Autograd
 - Theano, TensorFlow, PyTorch
- A neural network model is represented as a Directed Acyclic Graph
 - Each node contains a mathematical operation.
 - Each node contains the derivative of the math operation.
 - The error signal is propagated from the output nodes to the input nodes.

Math Program as DAG

- $x: [[1, 1], [1, 1]]$
- $a = x + 2$
- $b = a * a * 3$
- $c = b.mean()$

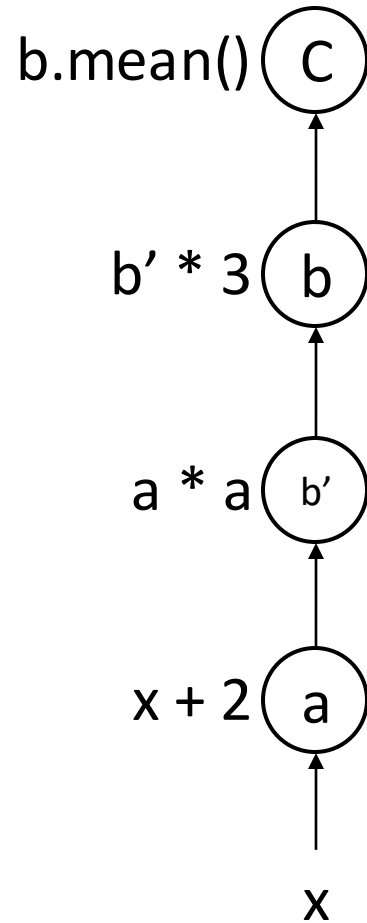
Math Program as DAG

- $x: [[1, 1], [1, 1]]$
- $a = x + 2$
- $b = a * a * 3$
- $c = b.\text{mean}()$



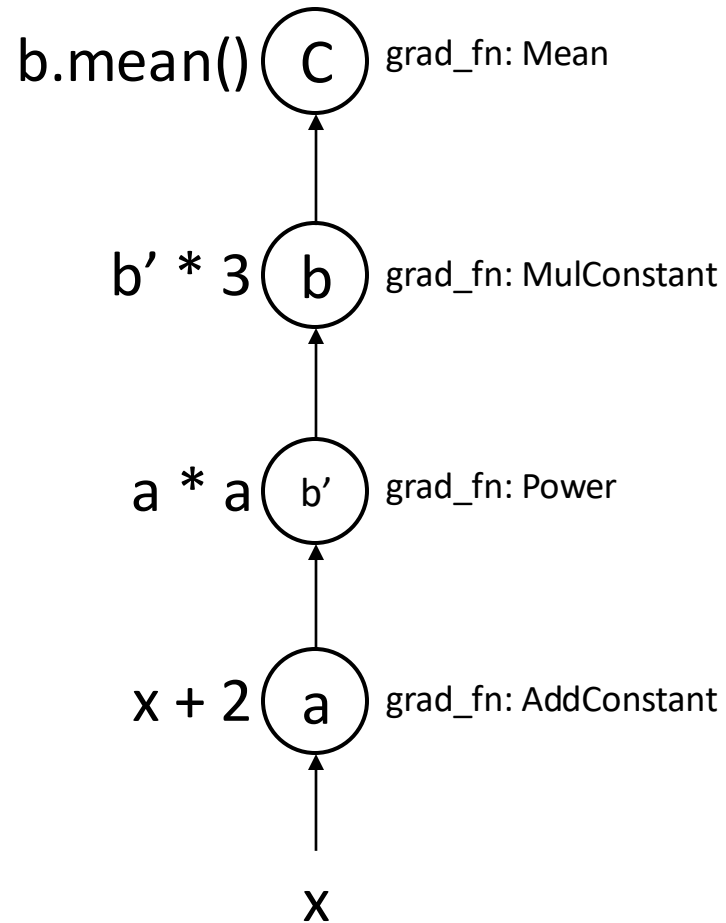
Math Program as DAG

- $x: [[1, 1], [1, 1]]$
- $a = x + 2$
- $b' = a * a$
- $b = b' * 3$
- $c = b.\text{mean}()$



Math Program as DAG

- $x: [[1, 1], [1, 1]]$
- $a = x + 2$
- $b' = a * a$
- $b = b' * 3$
- $c = b.\text{mean}()$

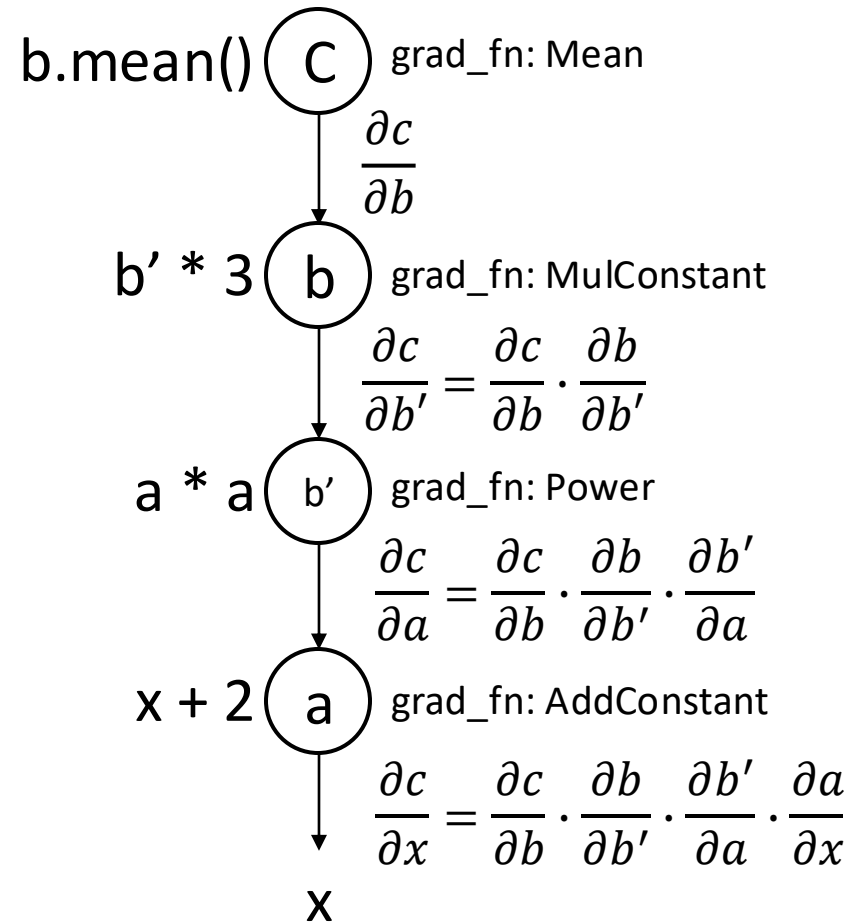


Each `grad_fn` has

- forward
 - Compute forward propagation
 - Save input & misc. for backward
- backward
 - Given δ^l , calculate δ^{l-1}
 - Calculate $\nabla_{W^l} C = \delta^l (a^{l-1})^T$, if necessary

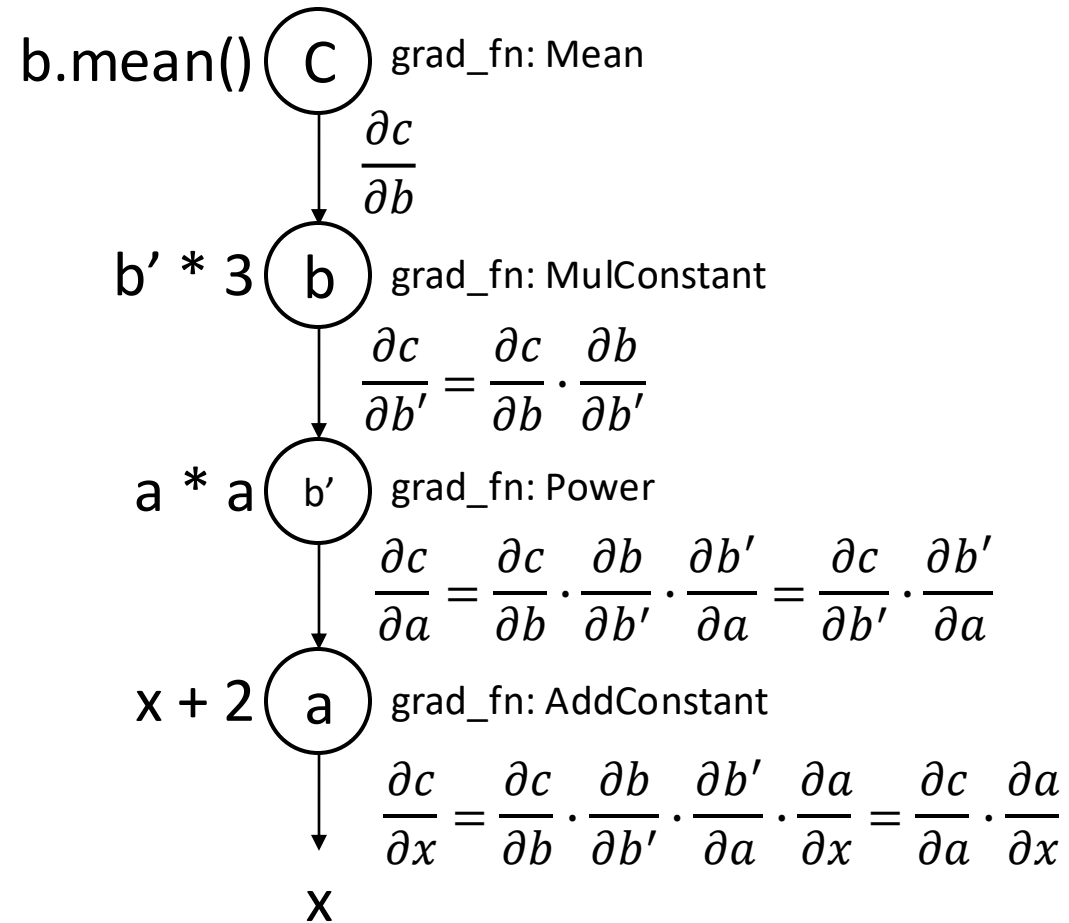
Math Program as DAG

- `x: [[1, 1], [1, 1]]`
- `a = x + 2`
- `b' = a * a`
- `b = b' * 3`
- `c = b.mean()`



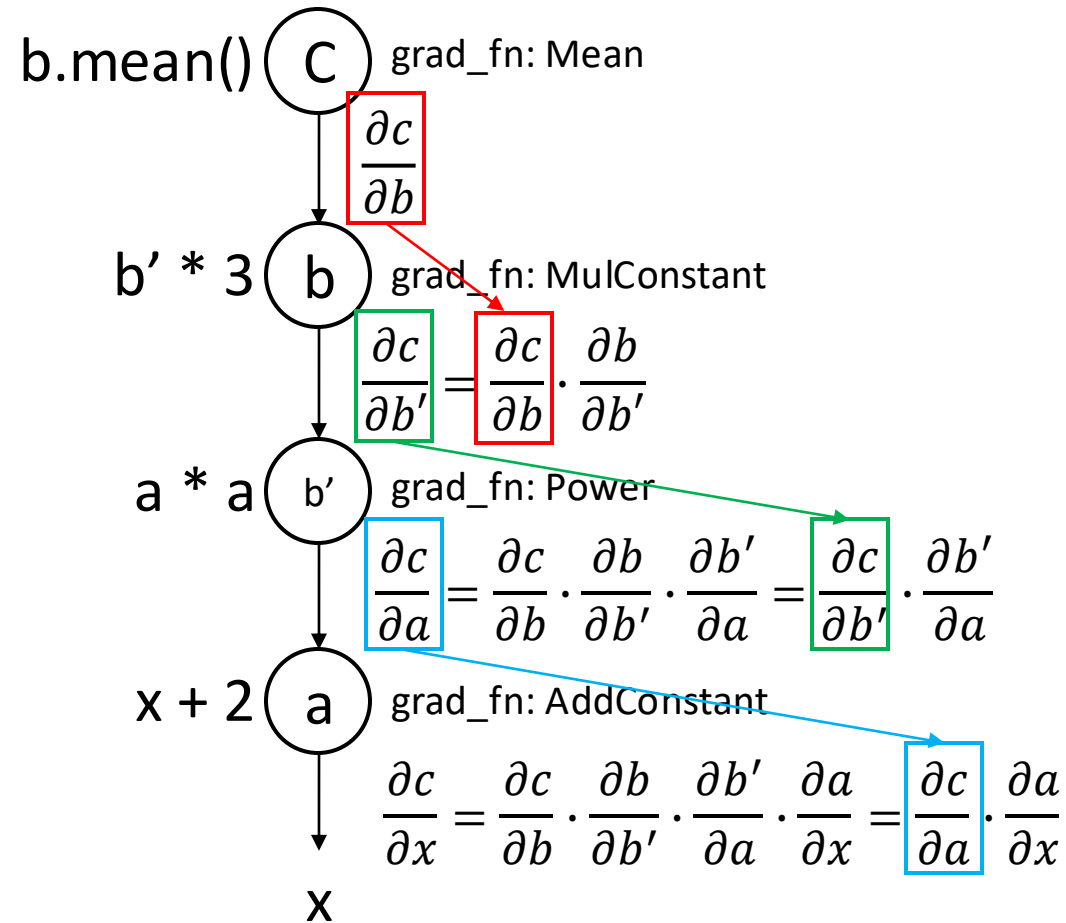
Math Program as DAG

- `x: [[1, 1], [1, 1]]`
- `a = x + 2`
- `b' = a * a`
- `b = b' * 3`
- `c = b.mean()`



Math Program as DAG

- $x: [[1, 1], [1, 1]]$
- $a = x + 2$
- $b' = a * a$
- $b = b' * 3$
- $c = b.\text{mean}()$



AI504: Programming for Artificial Intelligence

Week 3: Neural Nets & Backpropagation

Edward Choi

Grad School of AI

edwardchoi@kaist.ac.kr