UNIVERSITÄT
BERN

Deep Feedforward Networks

Paolo Favaro

Contents

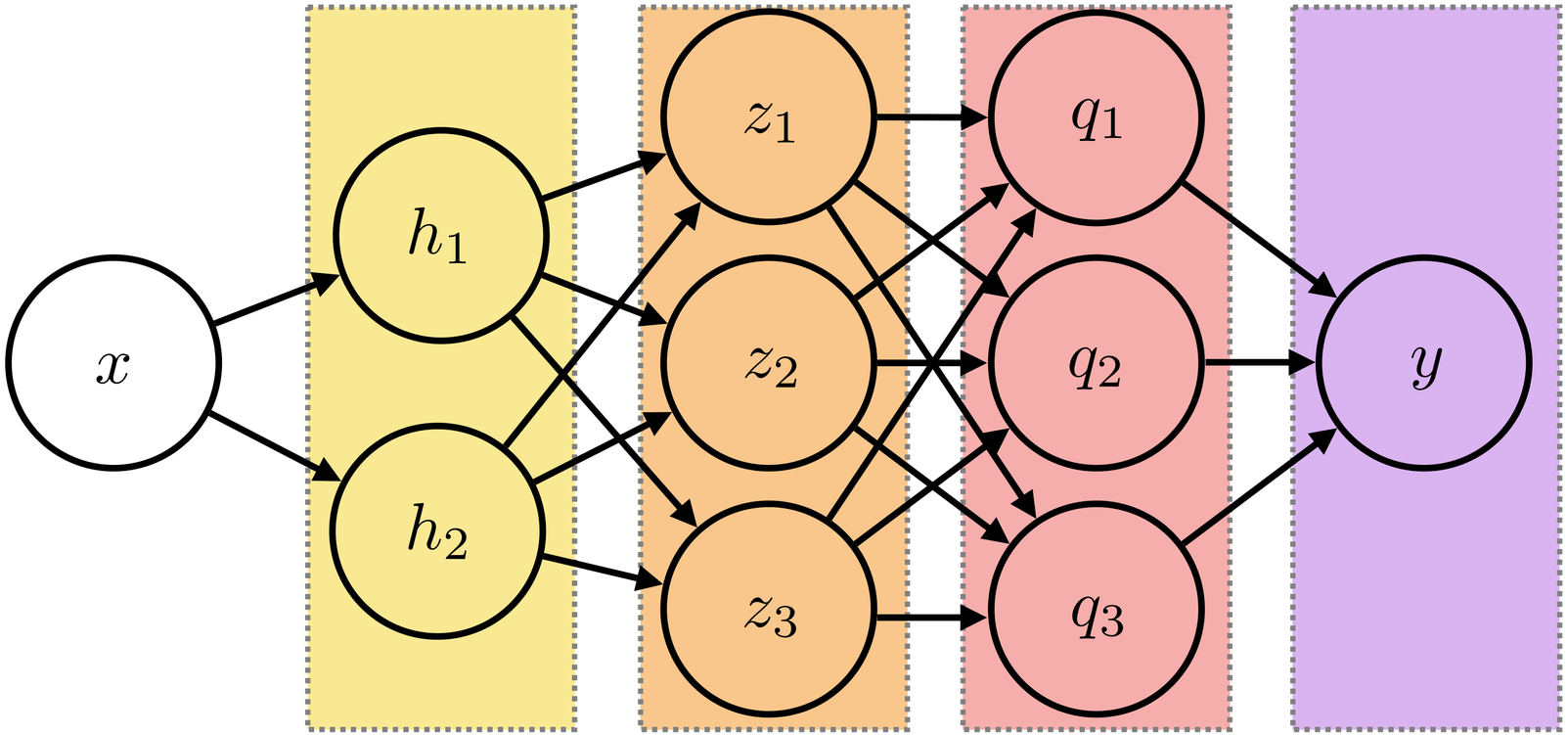
- Introduction to Feedforward Neural Networks: definition, design, training
- Based on **Chapter 6** (and 4) of Deep Learning by Goodfellow, Bengio, Courville
- References to Machine Learning and Pattern Recognition by Bishop

Resources

- Books and online material for further studies
  - CS231 @ Stanford (Fei-Fei Li)
  - **Pattern Recognition and Machine Learning**
    by Christopher M. Bishop
  - **Machine Learning: a Probabilistic Perspective** by Kevin P. Murphy

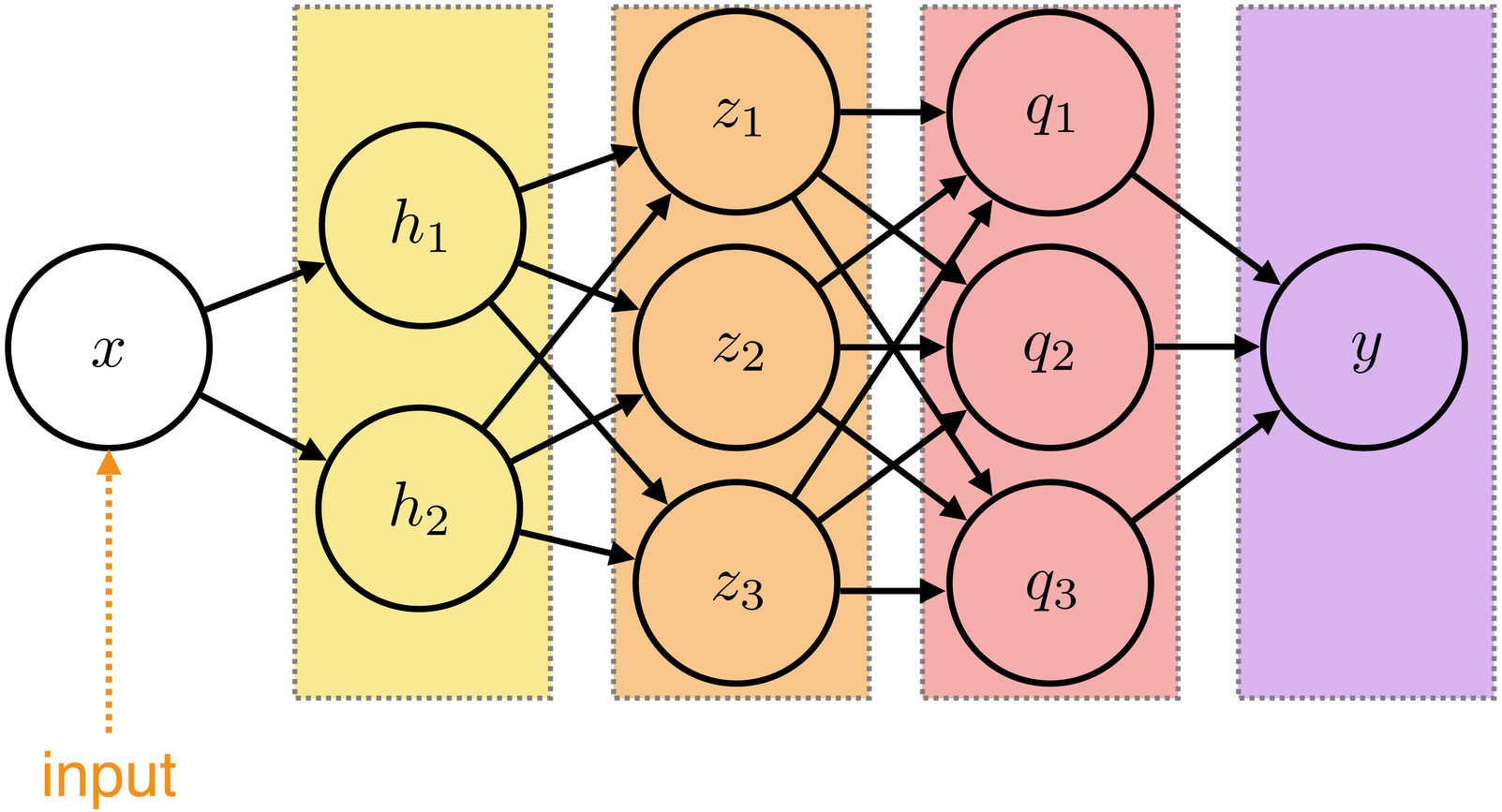
Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)



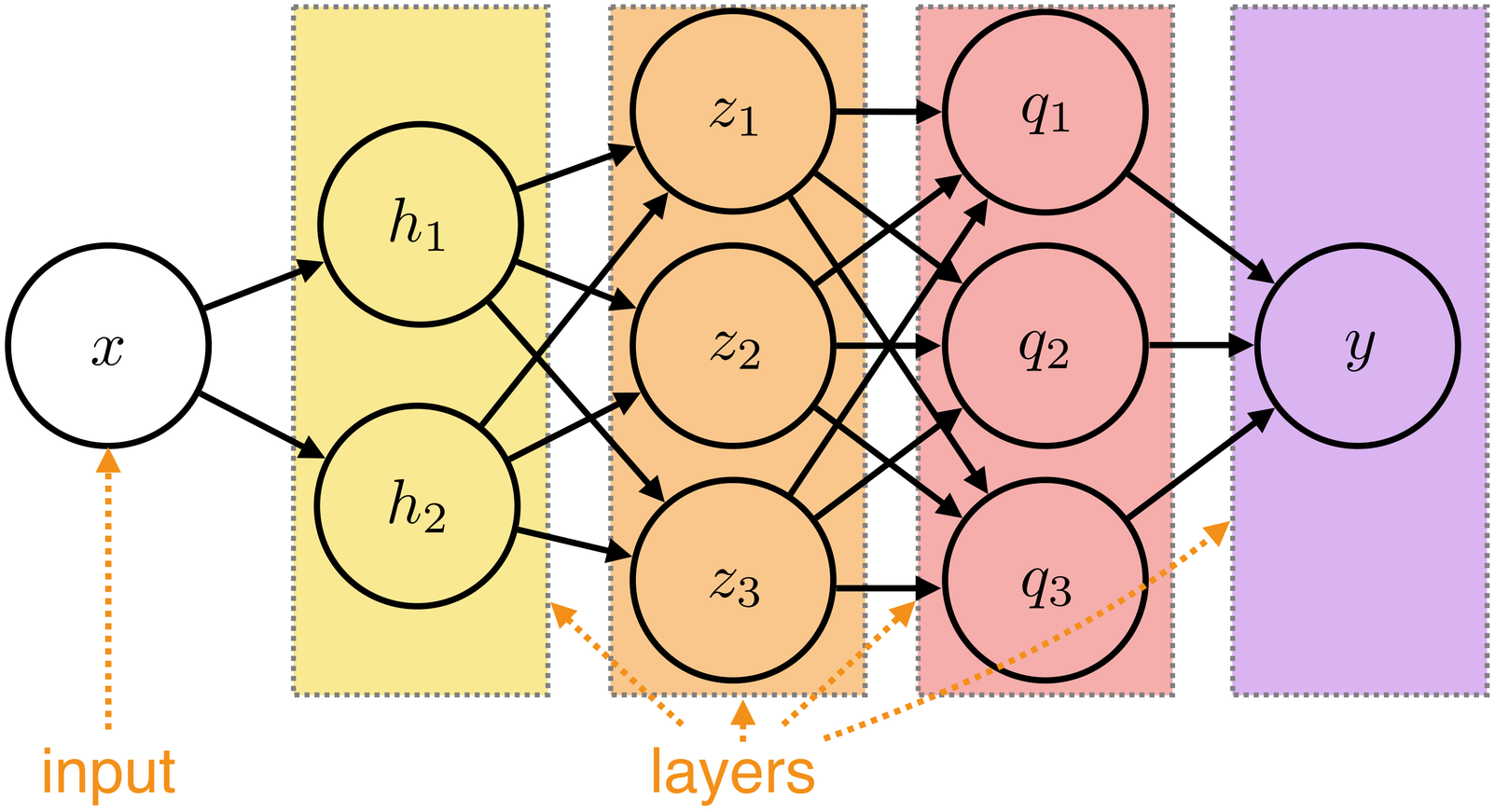
Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)



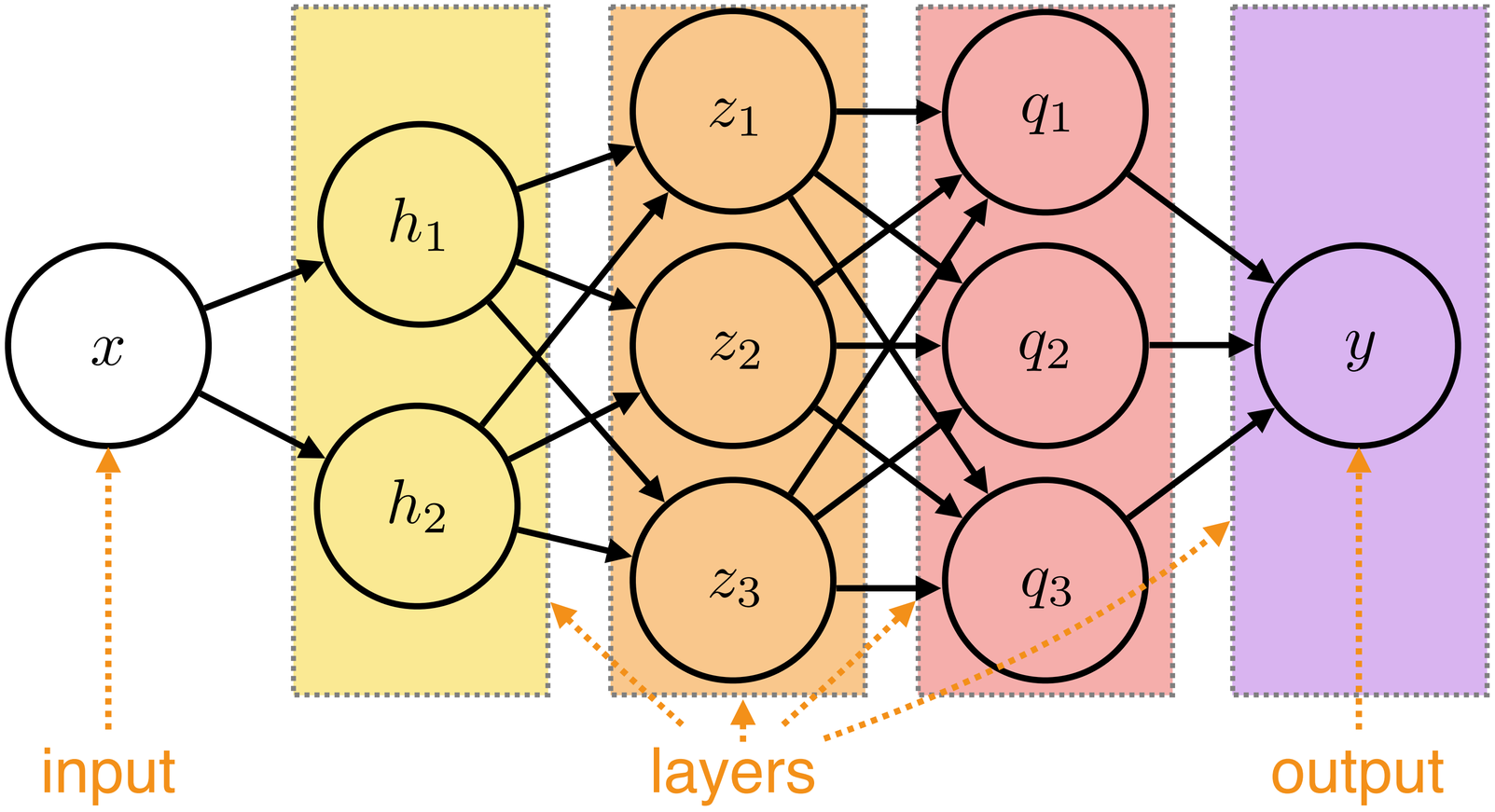
Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)

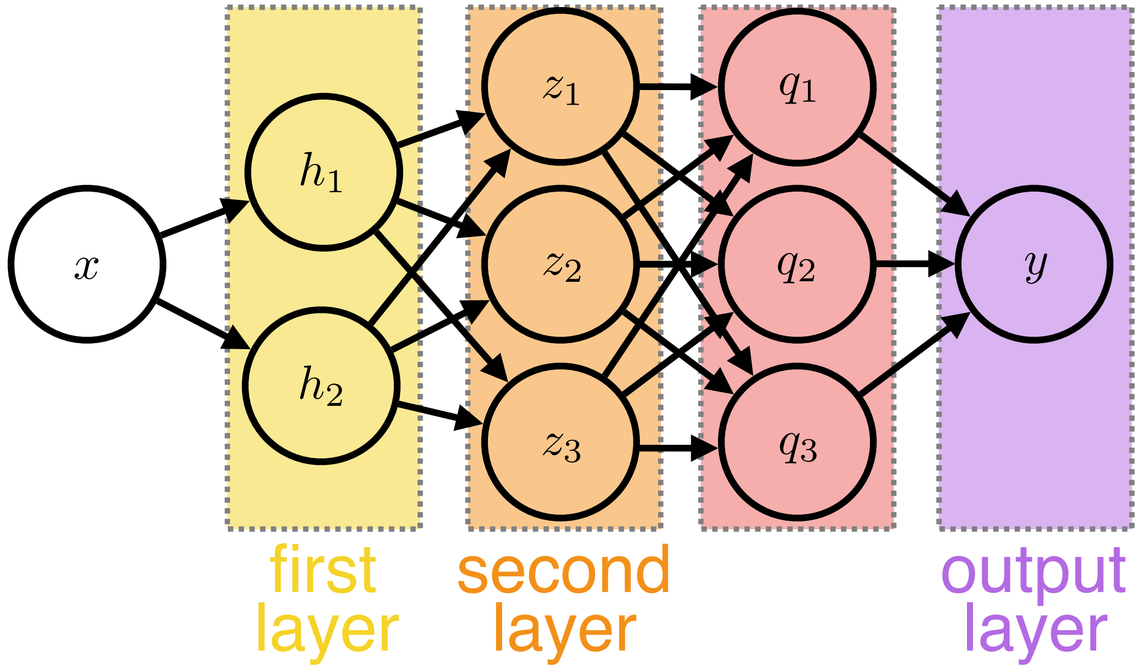


Feedforward Neural Networks

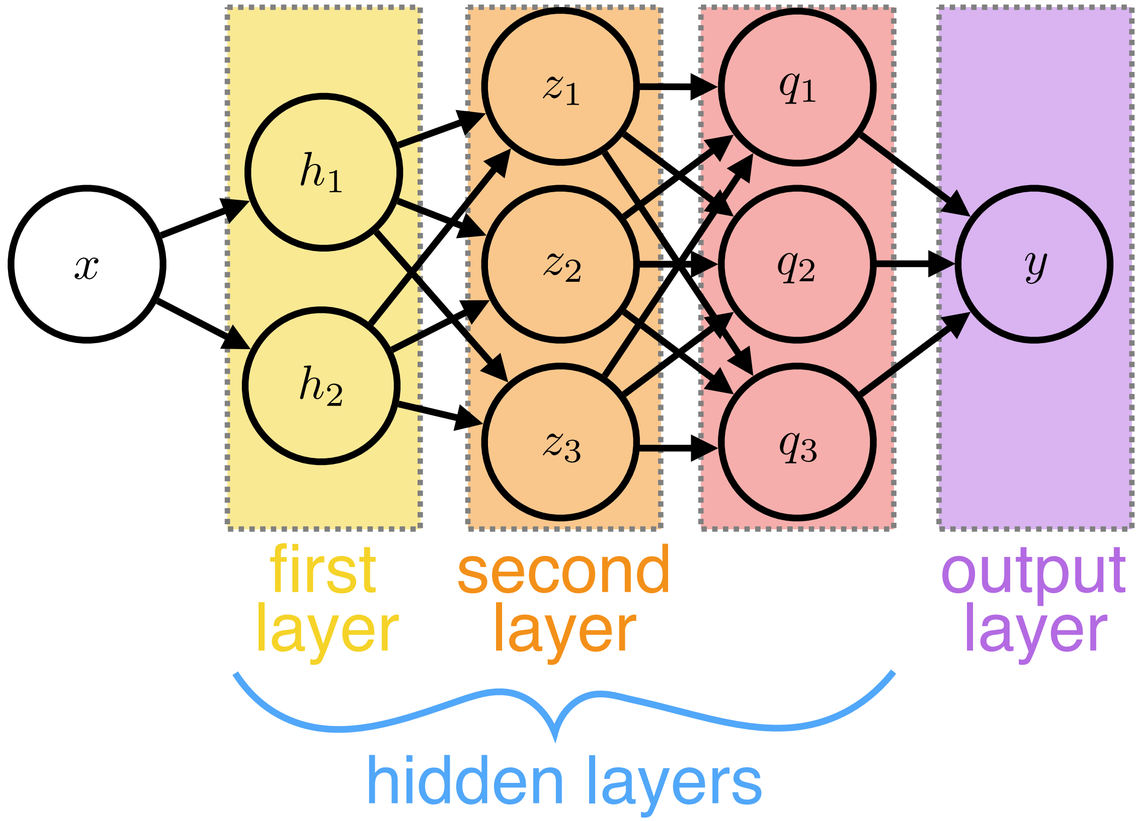
- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)



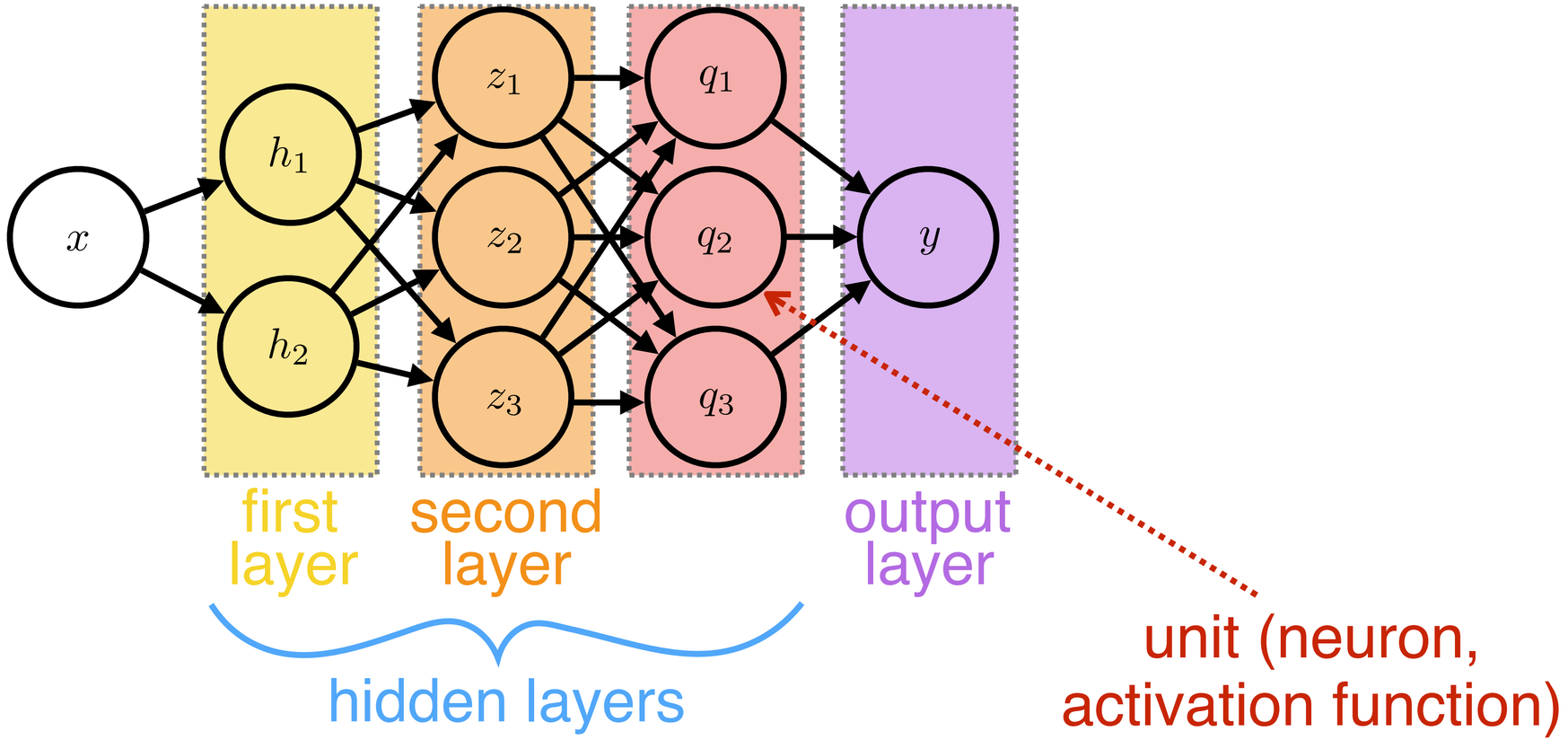
Feedforward Neural Networks



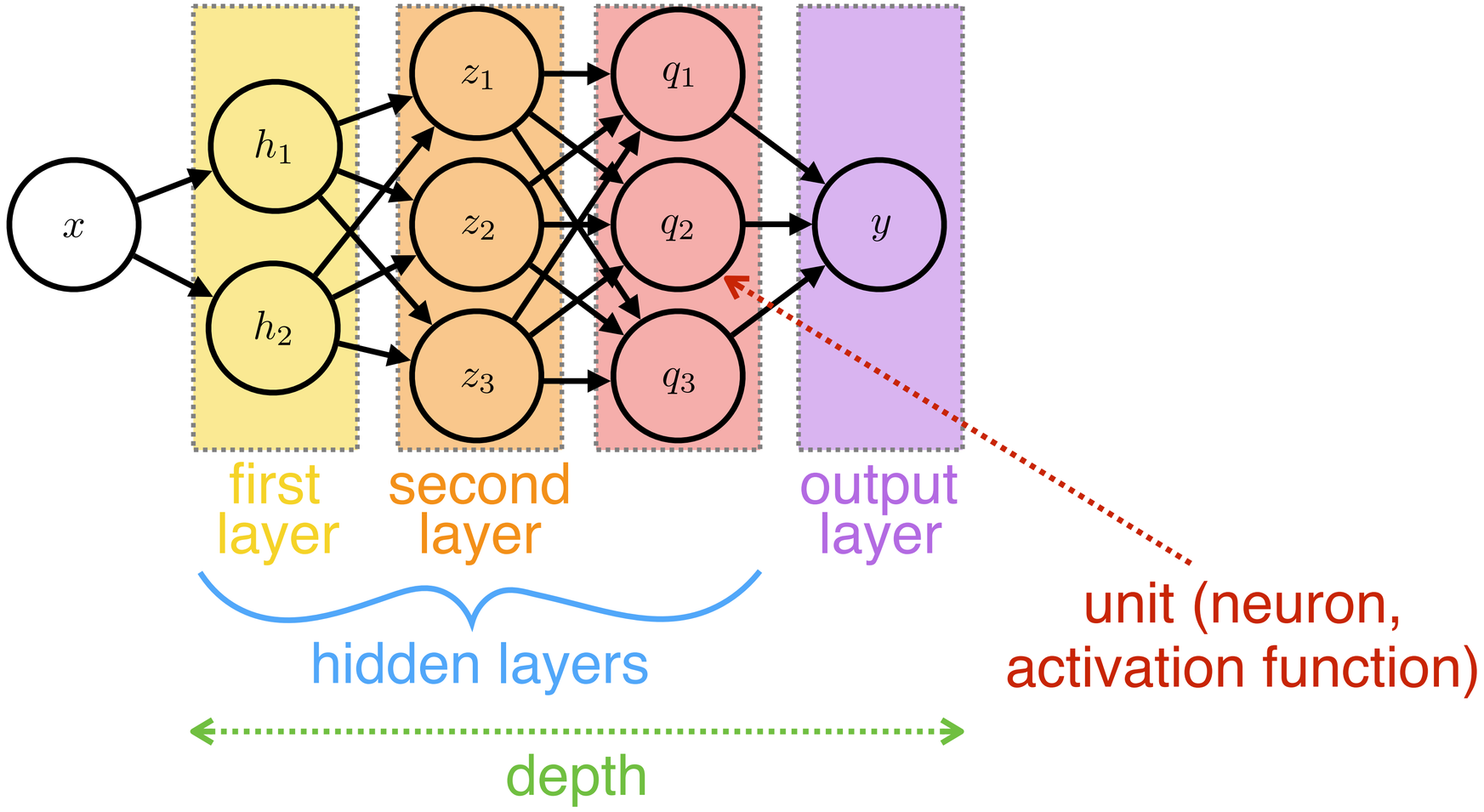
Feedforward Neural Networks



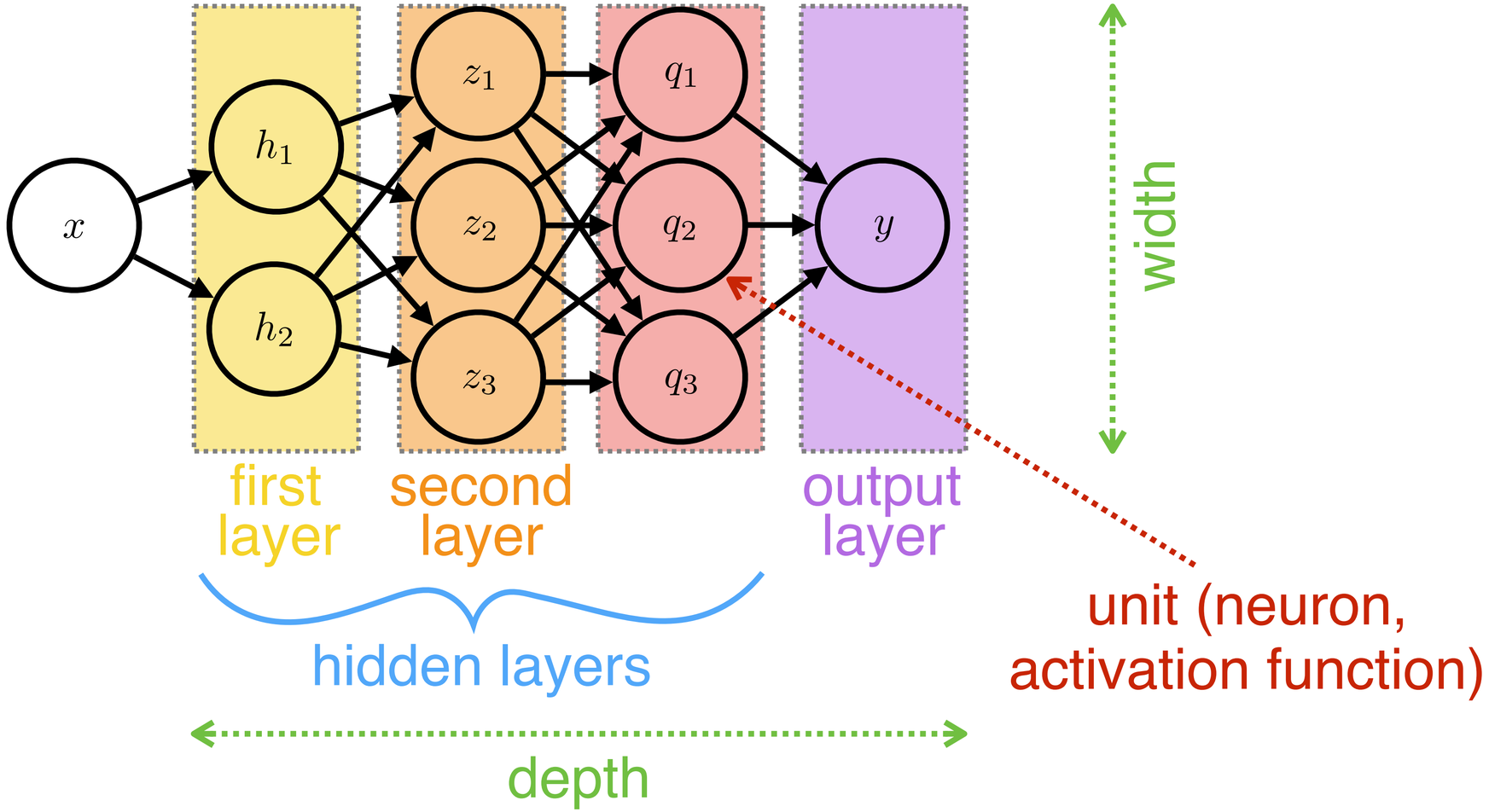
Feedforward Neural Networks



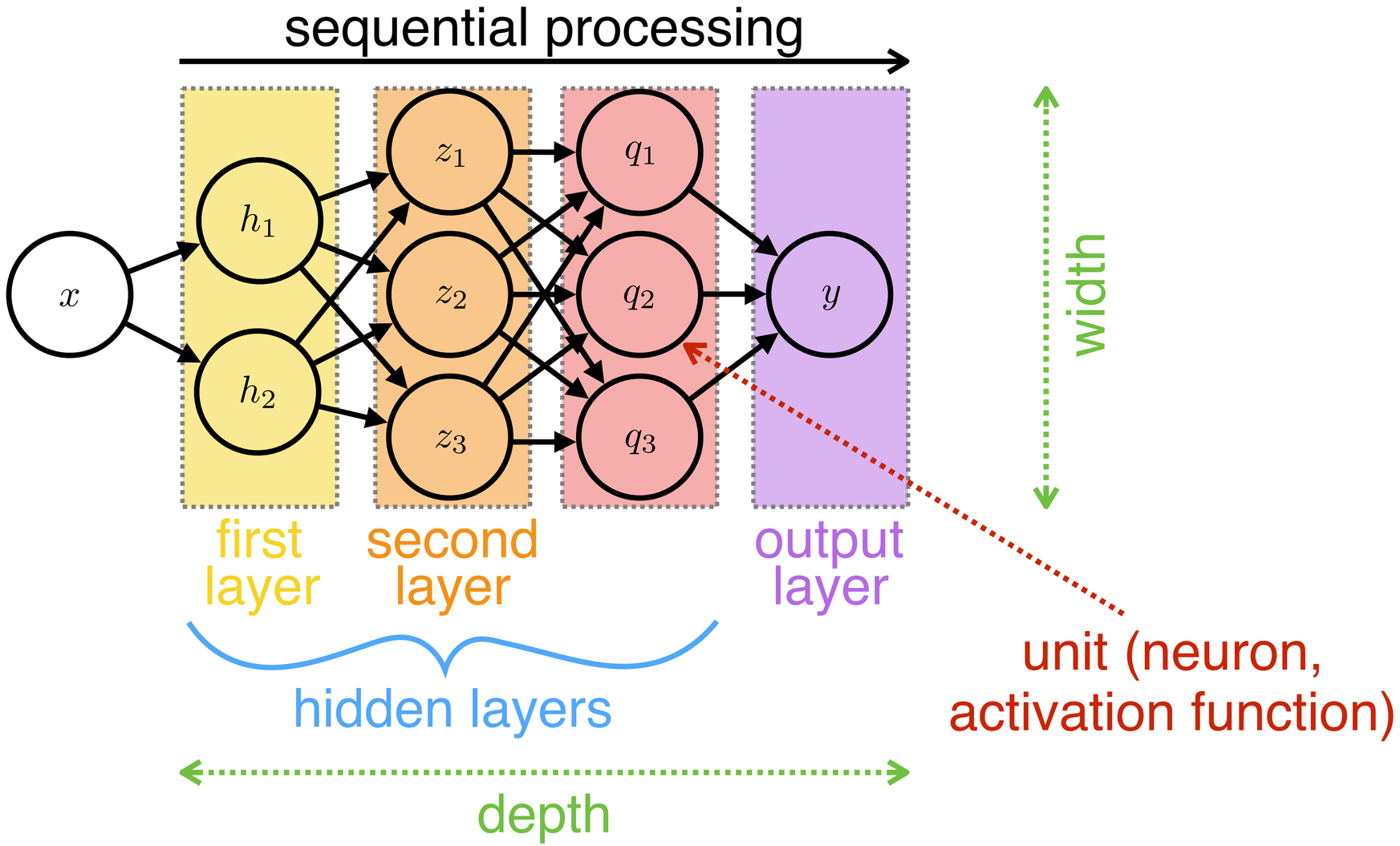
Feedforward Neural Networks



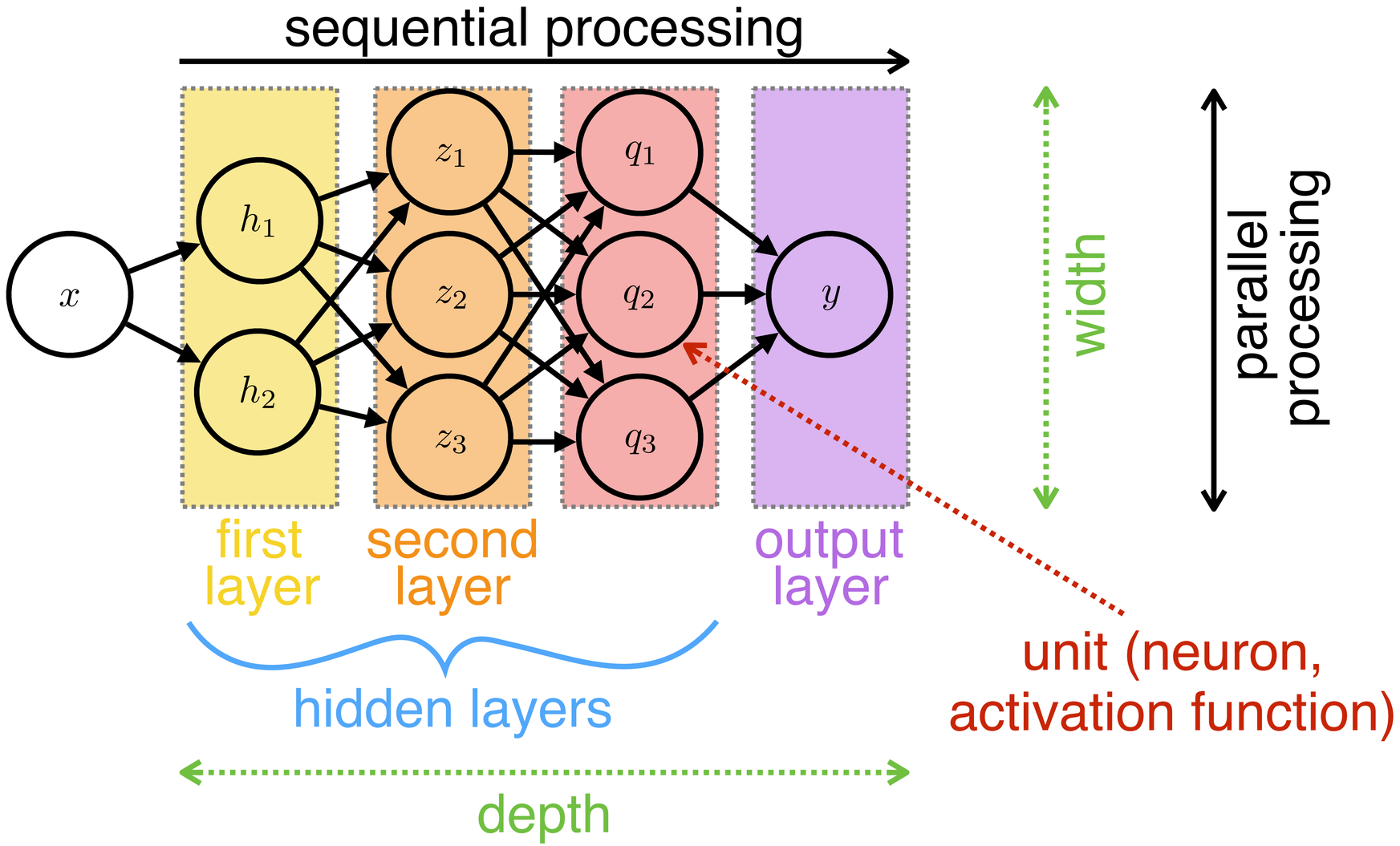
Feedforward Neural Networks



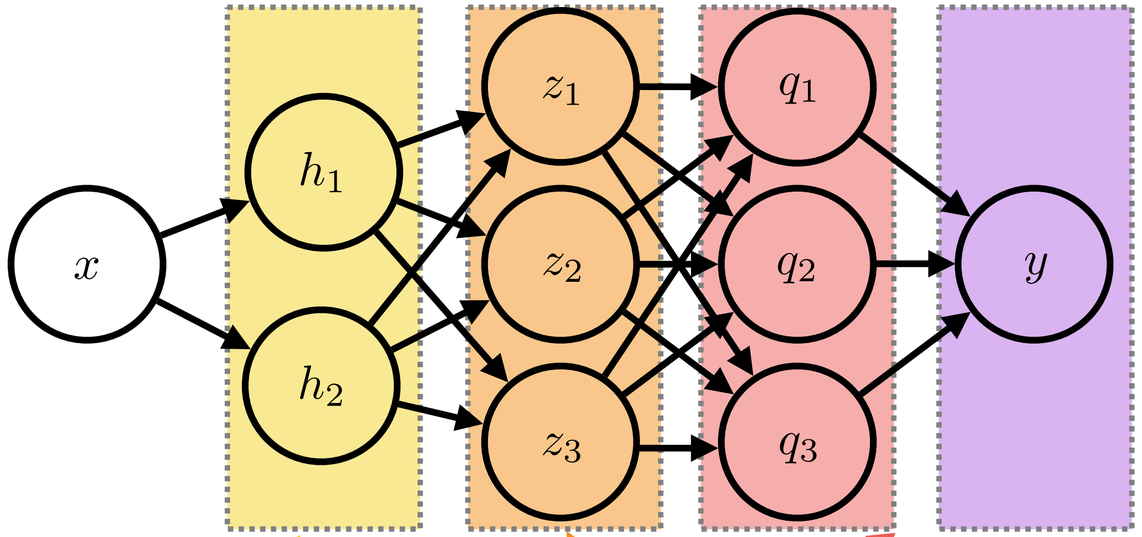
Feedforward Neural Networks



Feedforward Neural Networks



Feedforward Neural Networks



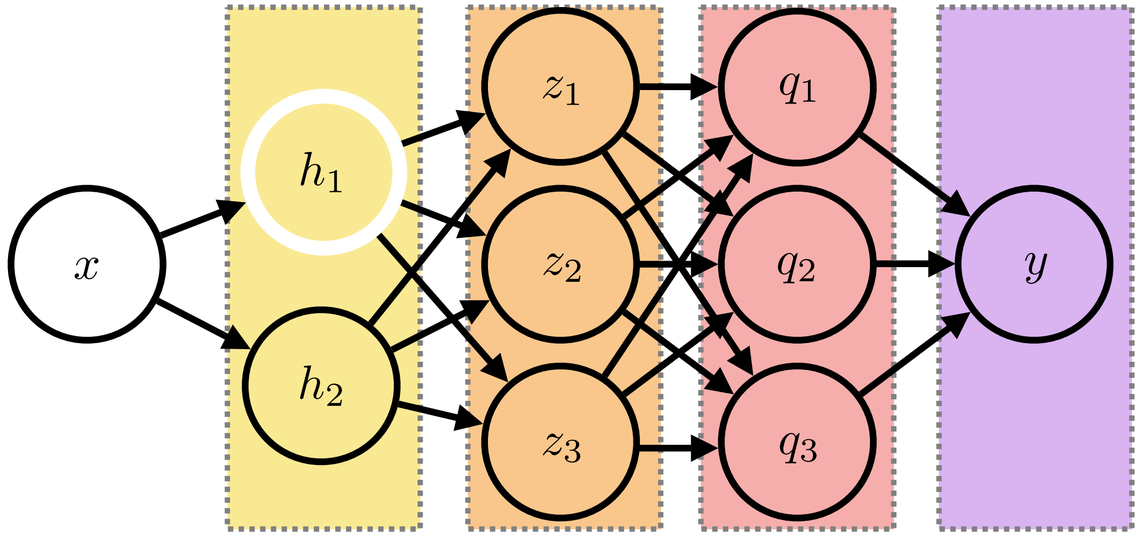
$$y = f_4(q_1, q_2, q_3)$$

$$h_1 = f_{1,1}(x)$$
$$h_2 = f_{1,2}(x)$$

$$z_1 = f_{2,1}(h_1, h_2)$$
$$z_2 = f_{2,2}(h_1, h_2)$$
$$z_3 = f_{2,3}(h_1, h_2)$$

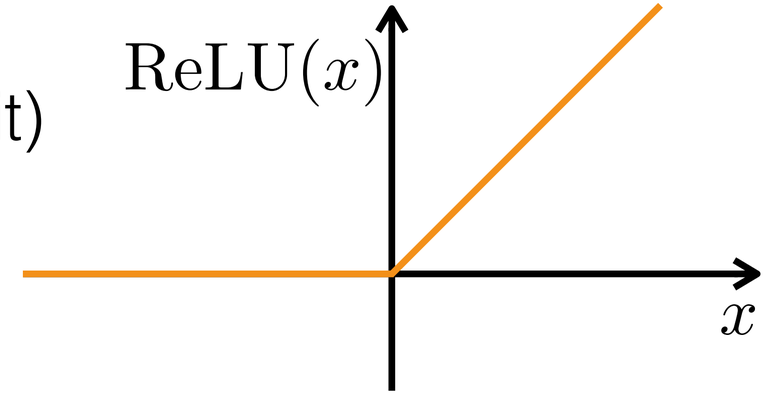
$$q_1 = f_{3,1}(z_1, z_2, z_3)$$
$$q_2 = f_{3,2}(z_1, z_2, z_3)$$
$$q_3 = f_{3,3}(z_1, z_2, z_3)$$

Feedforward Neural Networks

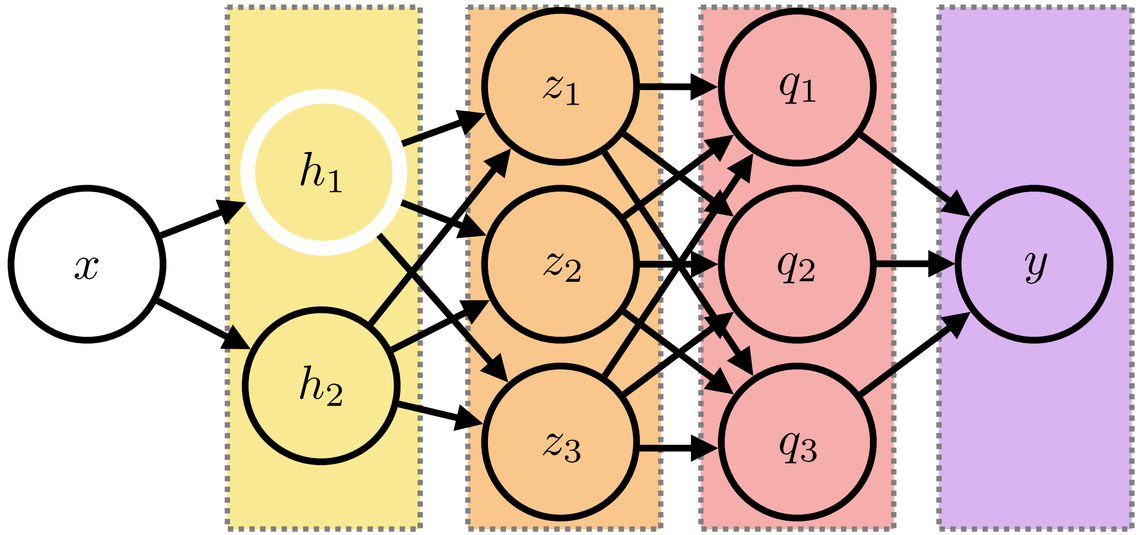


Example (rectified linear unit)

$f_{1,1}(x) = \mathrm{ReLU}(x)$

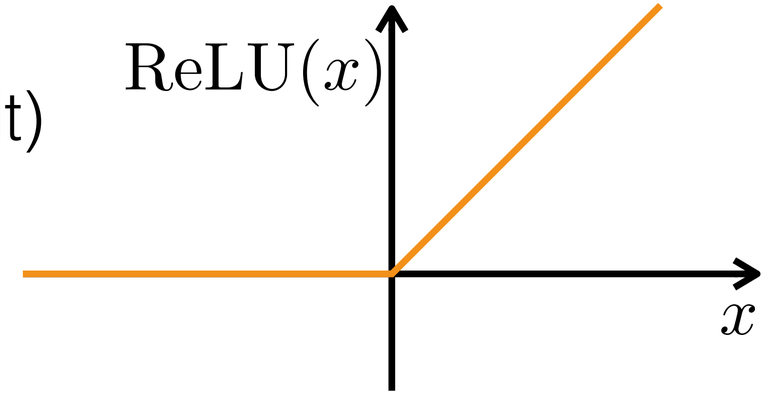


Feedforward Neural Networks

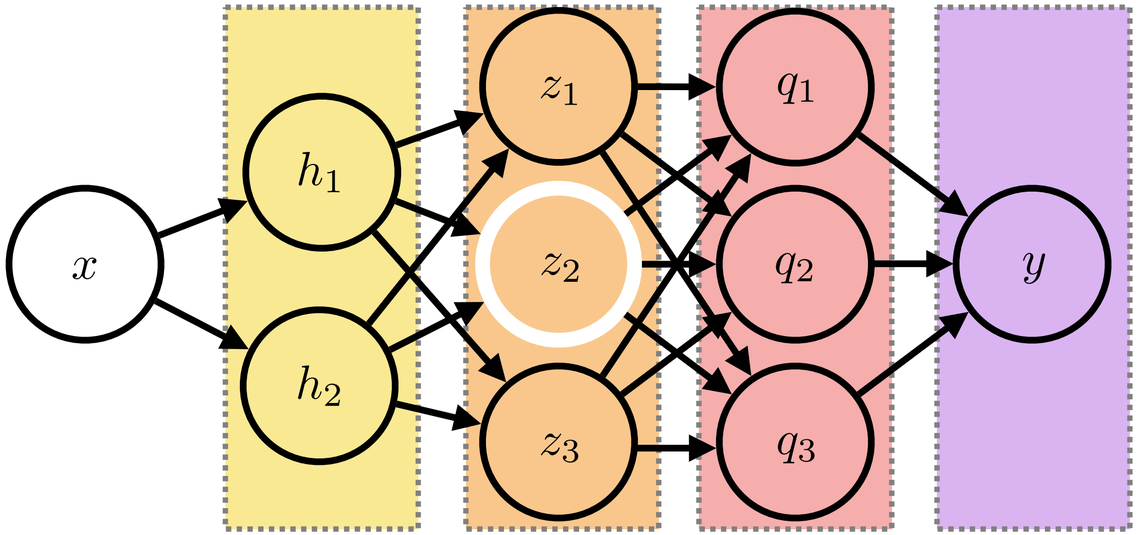


Example (rectified linear unit)

$f_{1,1}(x) = \mathrm{ReLU}(x)$



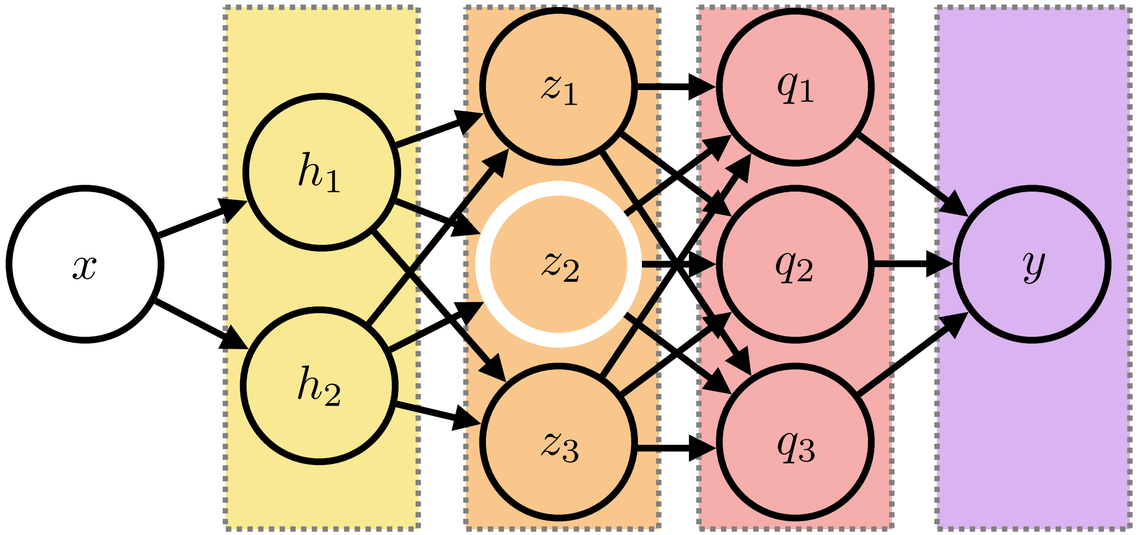
Feedforward Neural Networks



Example (fully connected unit)

$f_{2,2}(h_1, h_2) = w_1 h_1 + w_2 h_2$

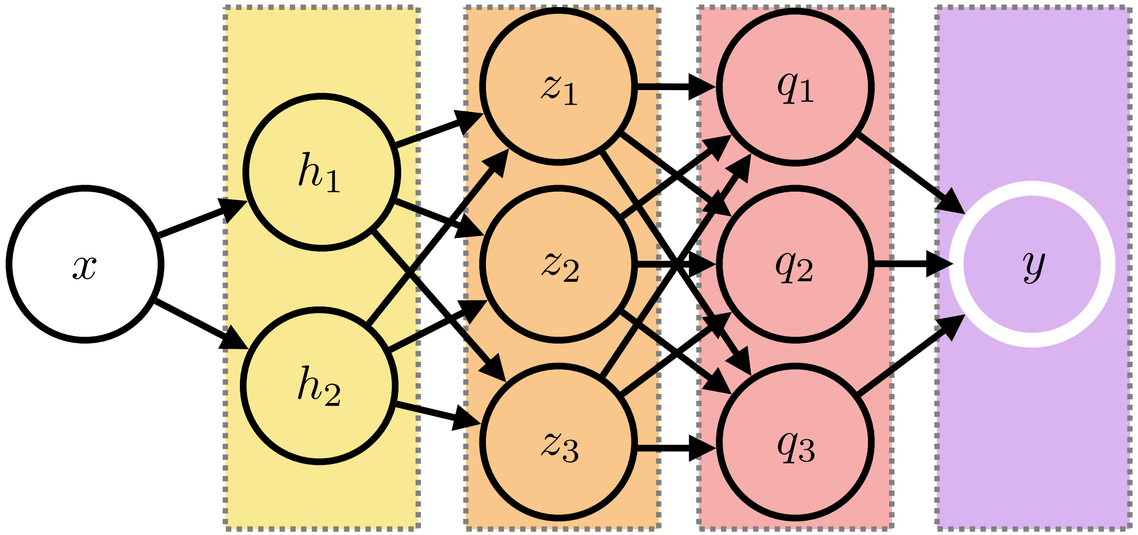
Feedforward Neural Networks



Example (fully connected unit)

$f_{2,2}(h_1, h_2) = w_1 h_1 + w_2 h_2$

Feedforward Neural Networks

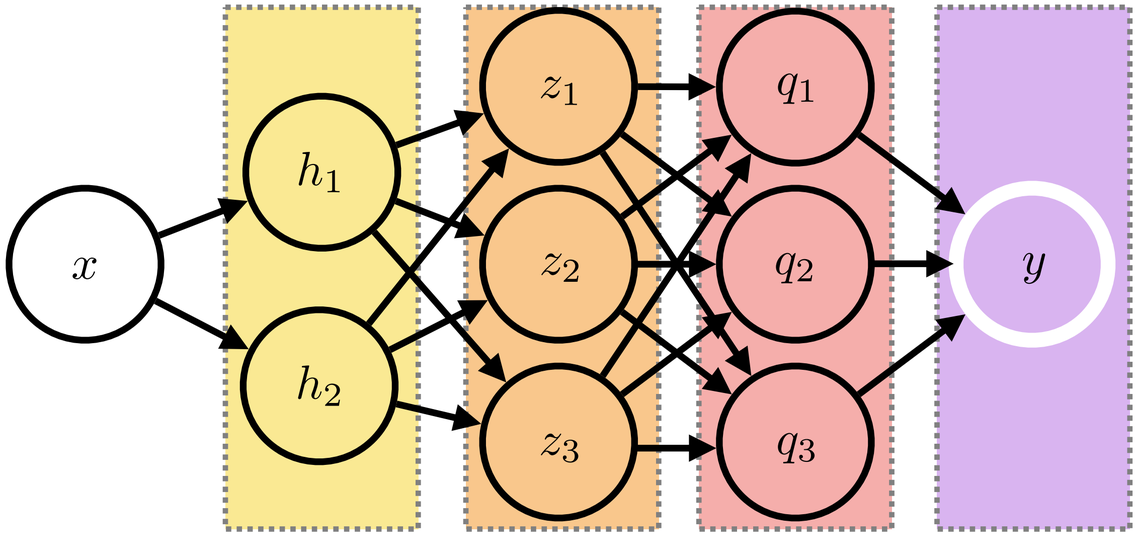


$$y = f_4(q_1, q_2, q_3)$$

Hierarchical composition of functions

$$y = f_4(f_{3,1}(f_{2,1}(f_{1,1}(x), f_{1,2}(x)), \dots), \dots)$$

Feedforward Neural Networks



$$y = f_4(q_1, q_2, q_3)$$

Hierarchical composition of functions

$$y = f_4(f_{3,1}(f_{2,1}(f_{1,1}(x), f_{1,2}(x)), \dots), \dots)$$

Feedforward Neural Networks

- Feedforward neural networks define a family of functions $f(x;\theta)$
- The goal is to find parameters θ that define the best mapping

$$y = f(x;\theta)$$

  between input x and output y

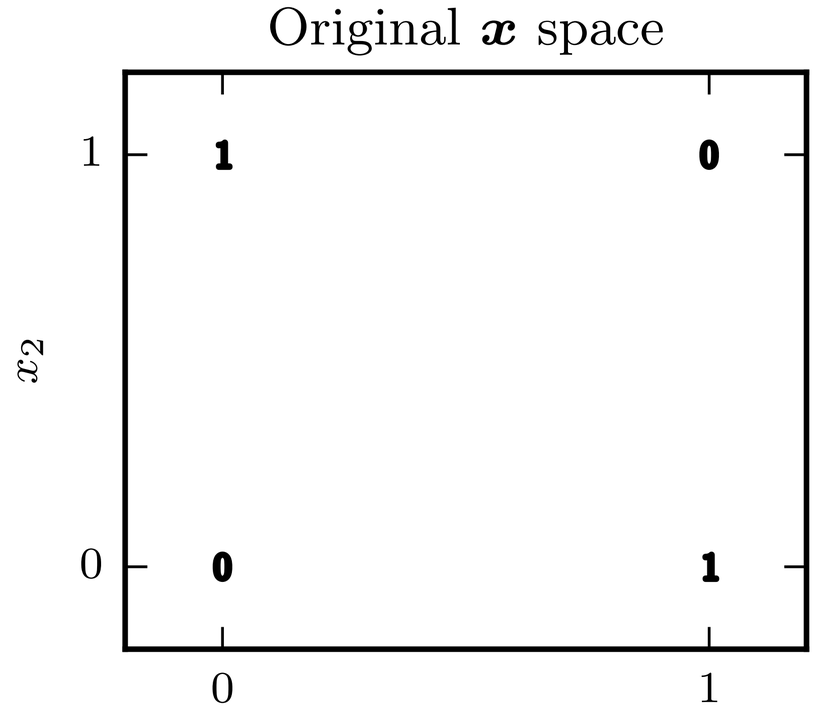
- The key constraints are the I/O dependencies

Deploying a Neural Network

- Given a **task** (in terms of I/O mappings)
- We need
  - **Cost function**
  - **Neural network model** (e.g., choice of units, their number, their connectivity)
  - **Optimization method** (back-propagation)

Example: Learning XOR

- Objective function is the XOR operation between two binary inputs x_1 and x_2



- Training set (x,y) pairs is

$$\left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 0 \right) \right\}$$

Cost Function

- Let us use the Mean Squared Error (MSE) as a first attempt

$$J(\theta) = \frac{1}{4}\sum_{i=1}^{4}\left(y^i - f(x^i;\theta)\right)^2$$

Linear Model

- Let us try a linear model of the form

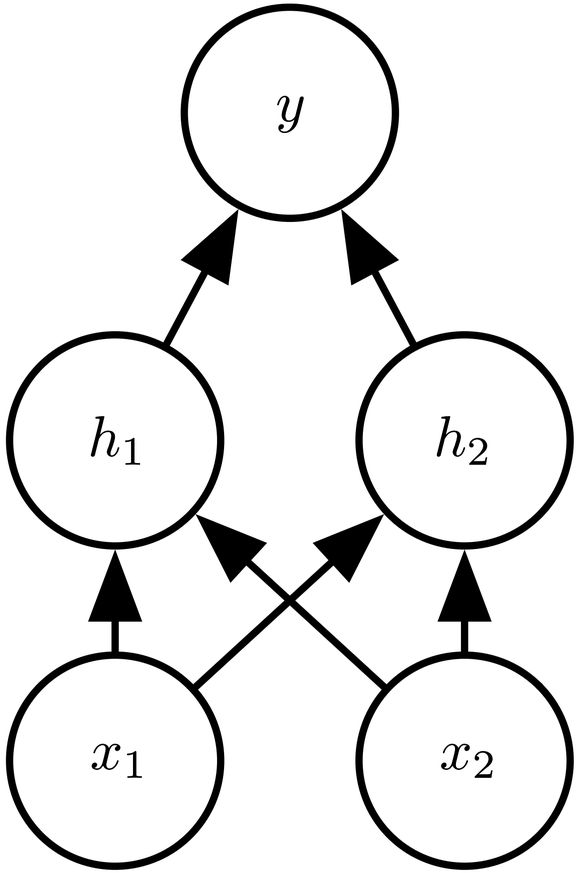
$$f(x; w, b) = w^\top x + b$$

- This choice leads to the normal equations (see slides on Machine Learning Review) and the following values for the parameters

$$\omega = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad b = \frac{1}{2}$$

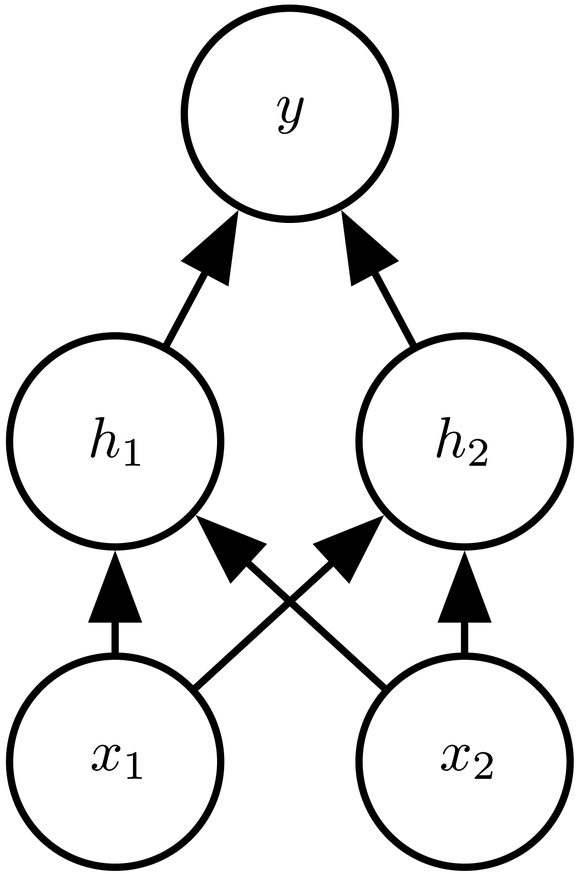
Nonlinear Model

- Let us try a simple feedforward network with one hidden layer and two hidden units



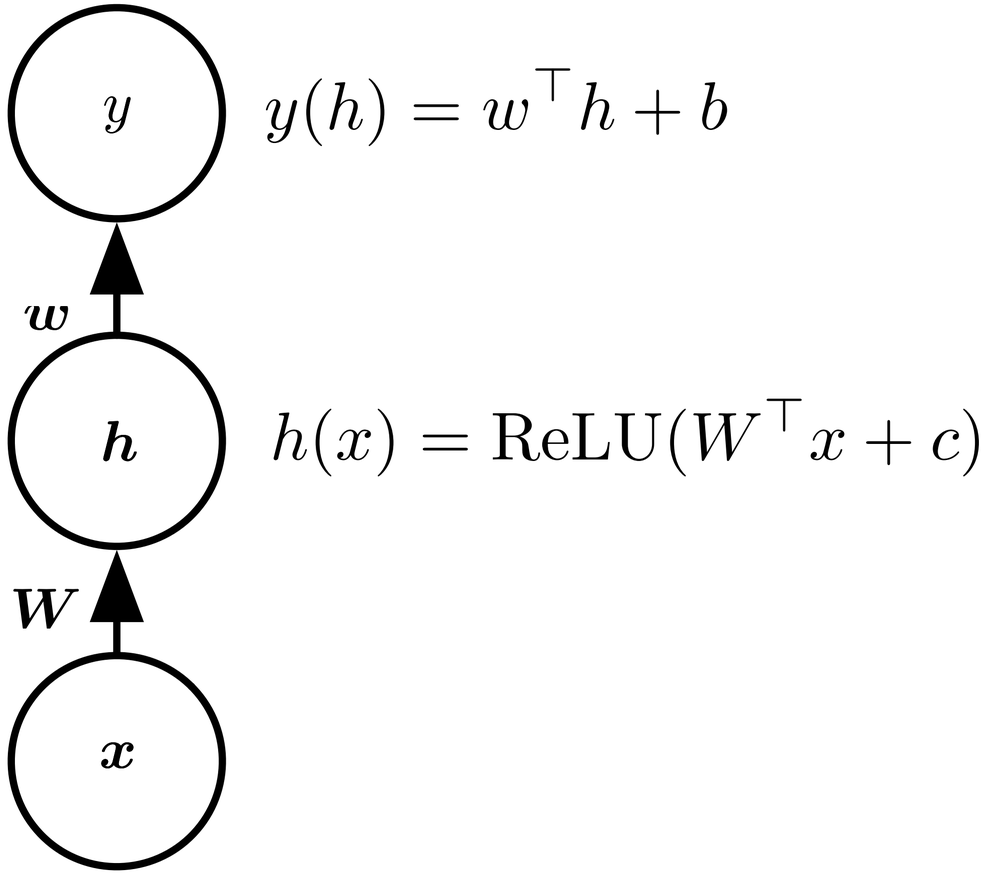
Nonlinear Model

- If each activation function is linear then the composite function would also be linear
- We would have the same poor result as before
- We must consider nonlinear activation functions



Nonlinear Model

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



Optimization

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

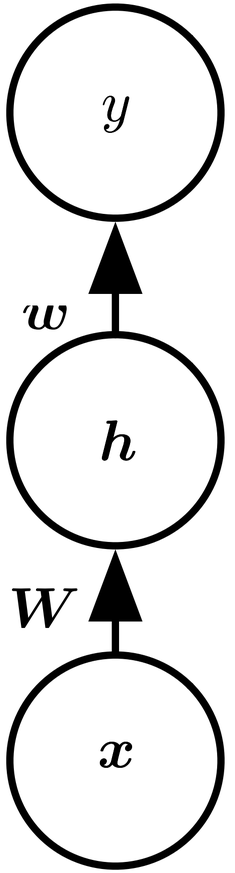
At this stage we would use optimization to fit f to the y in the training set. In this example, we skip this step and assume that some oracle gives us the parameters

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad b = 0$$

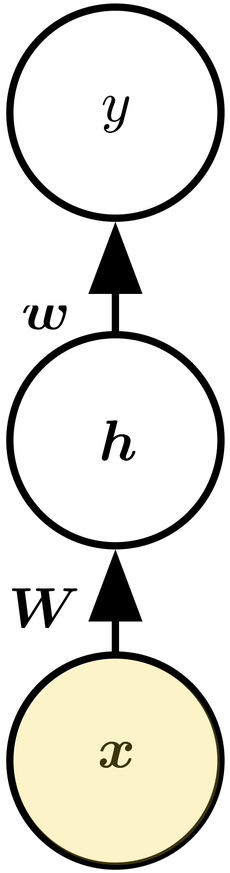
Simulation

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



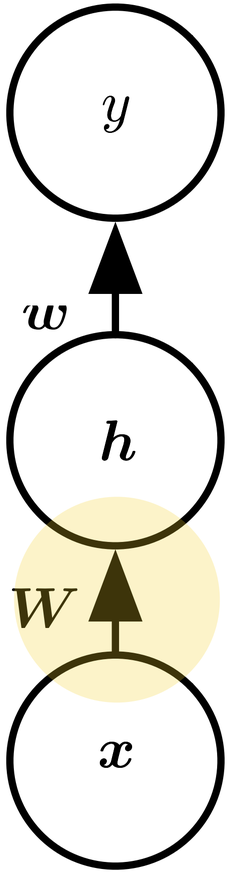
Simulation

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



Simulation

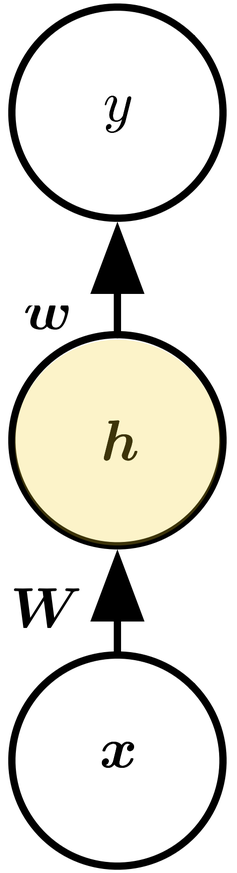
$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\Rightarrow \max\{0, XW + \mathbf{1}c\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

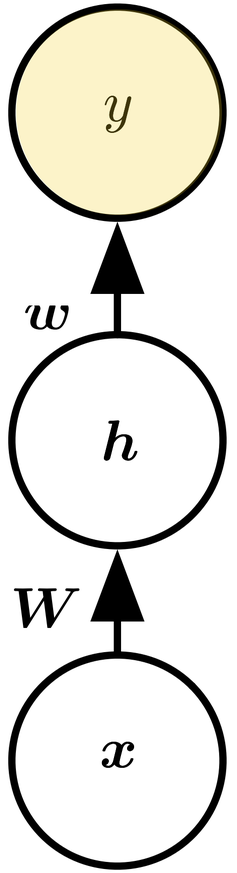


Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

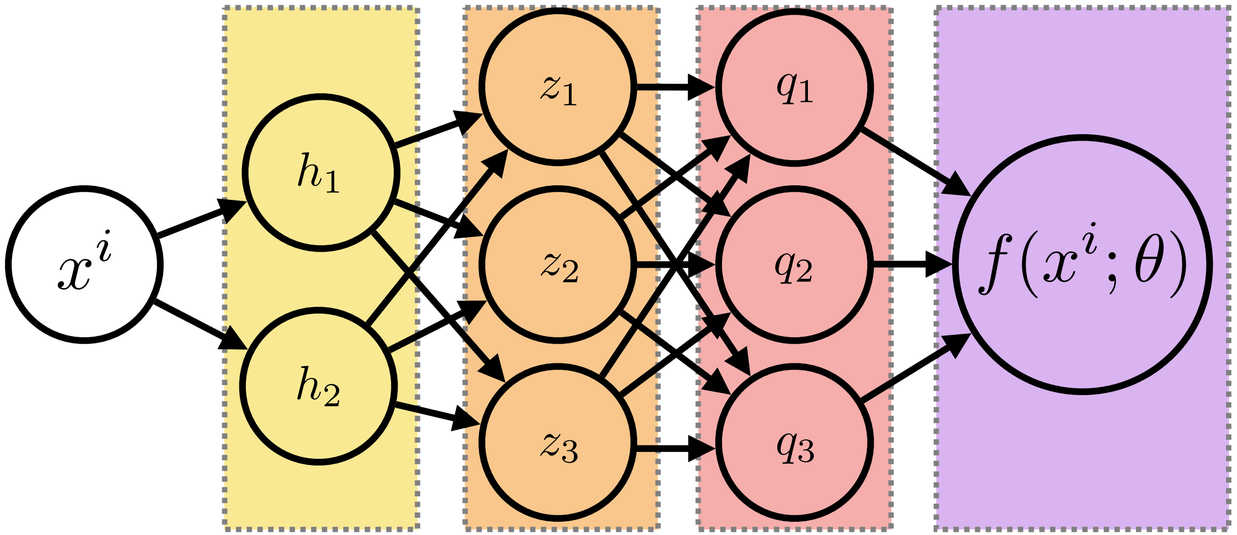
$$\rightarrow \max\{0, XW + \mathbf{1}c\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\rightarrow \max\{0, XW + \mathbf{1}c\}w + \mathbf{1}b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



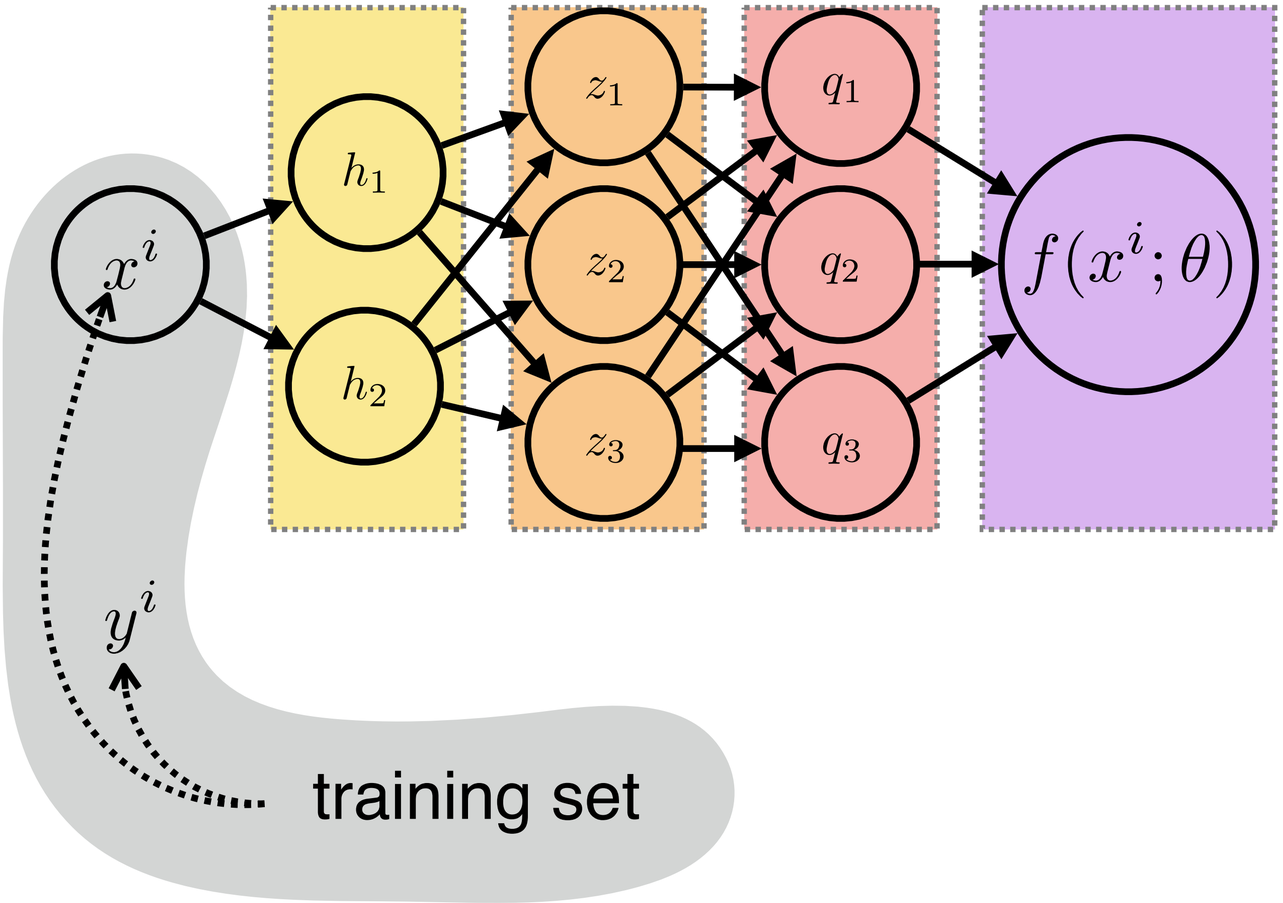
the XOR function
(matches Y)

Step-by-Step Analysis

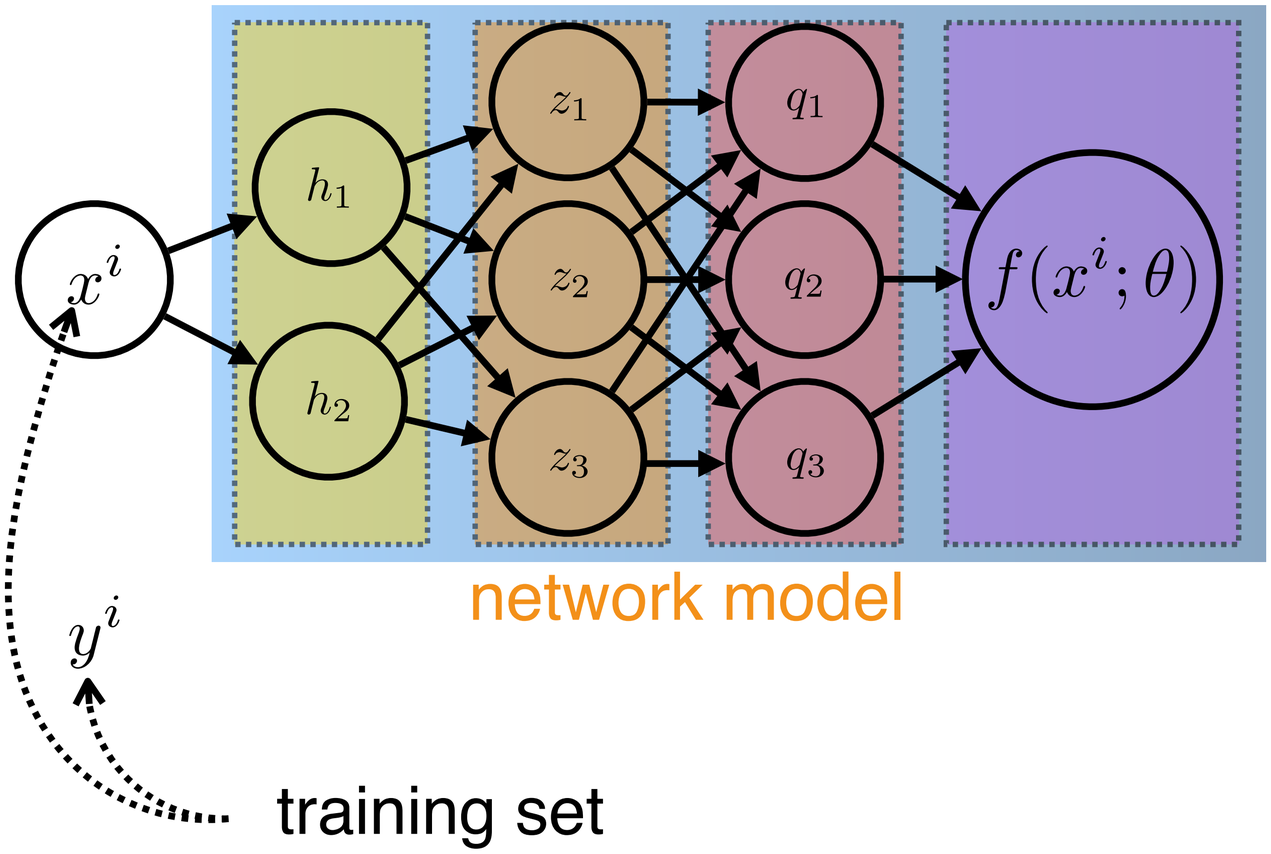


y^i

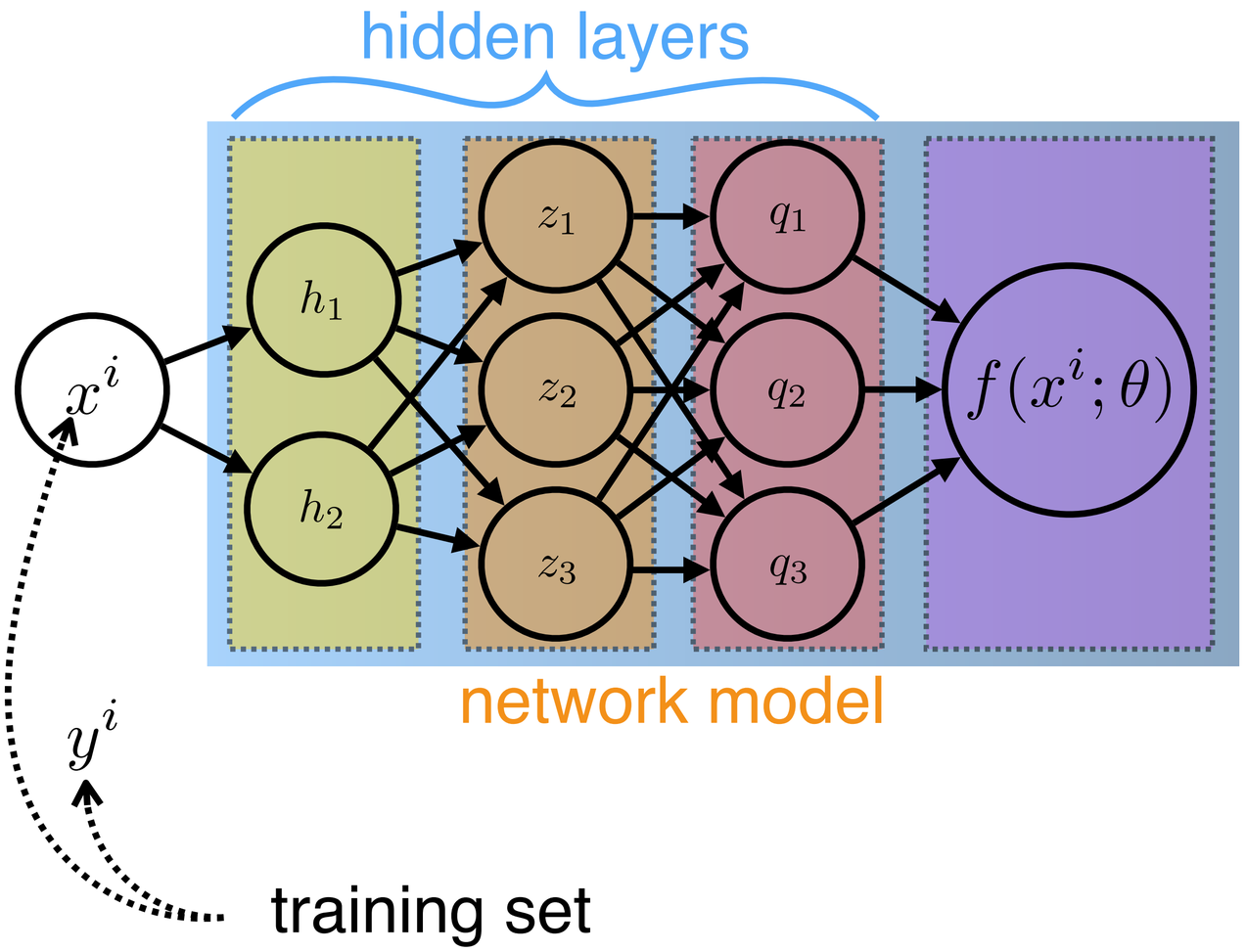
Step-by-Step Analysis



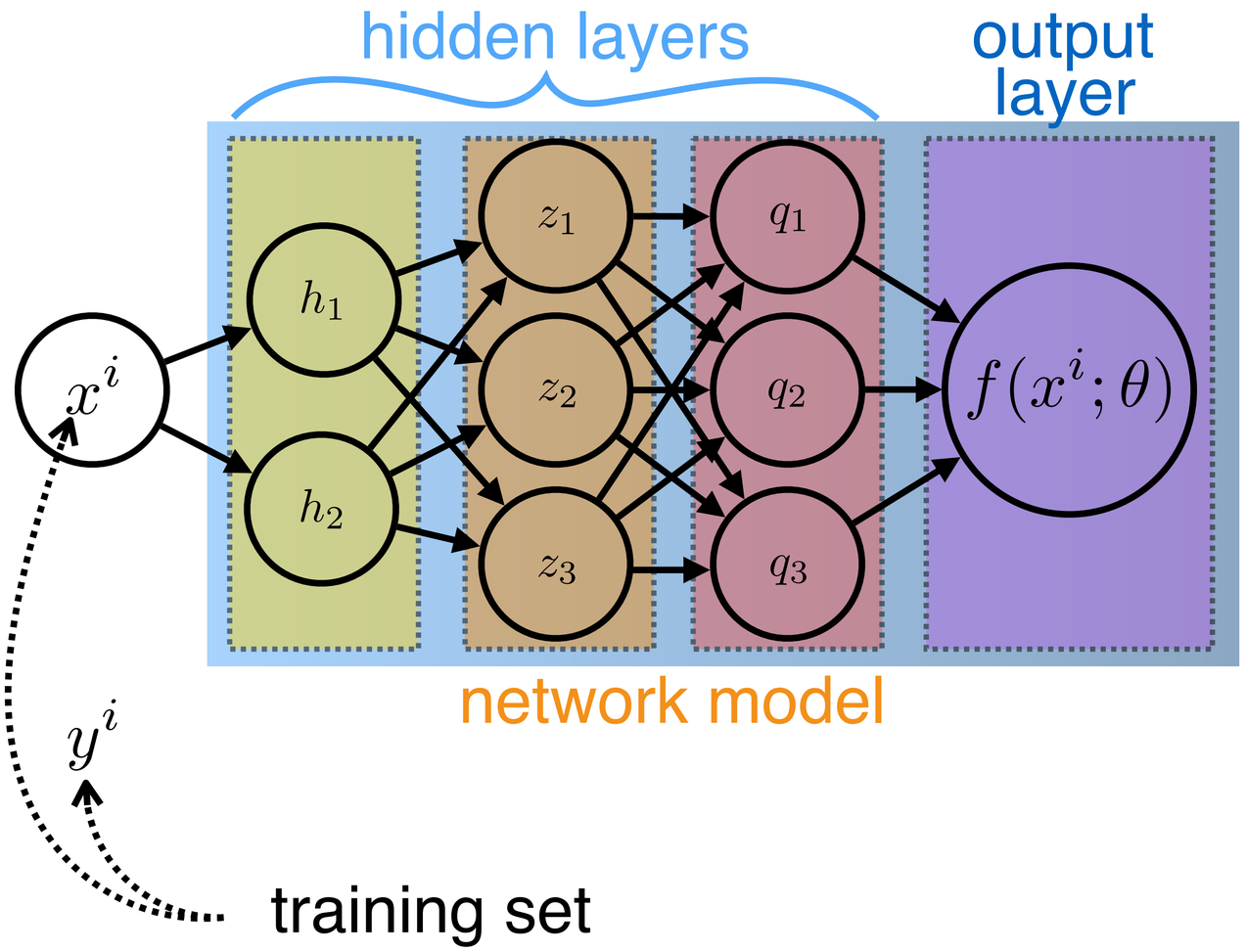
Step-by-Step Analysis



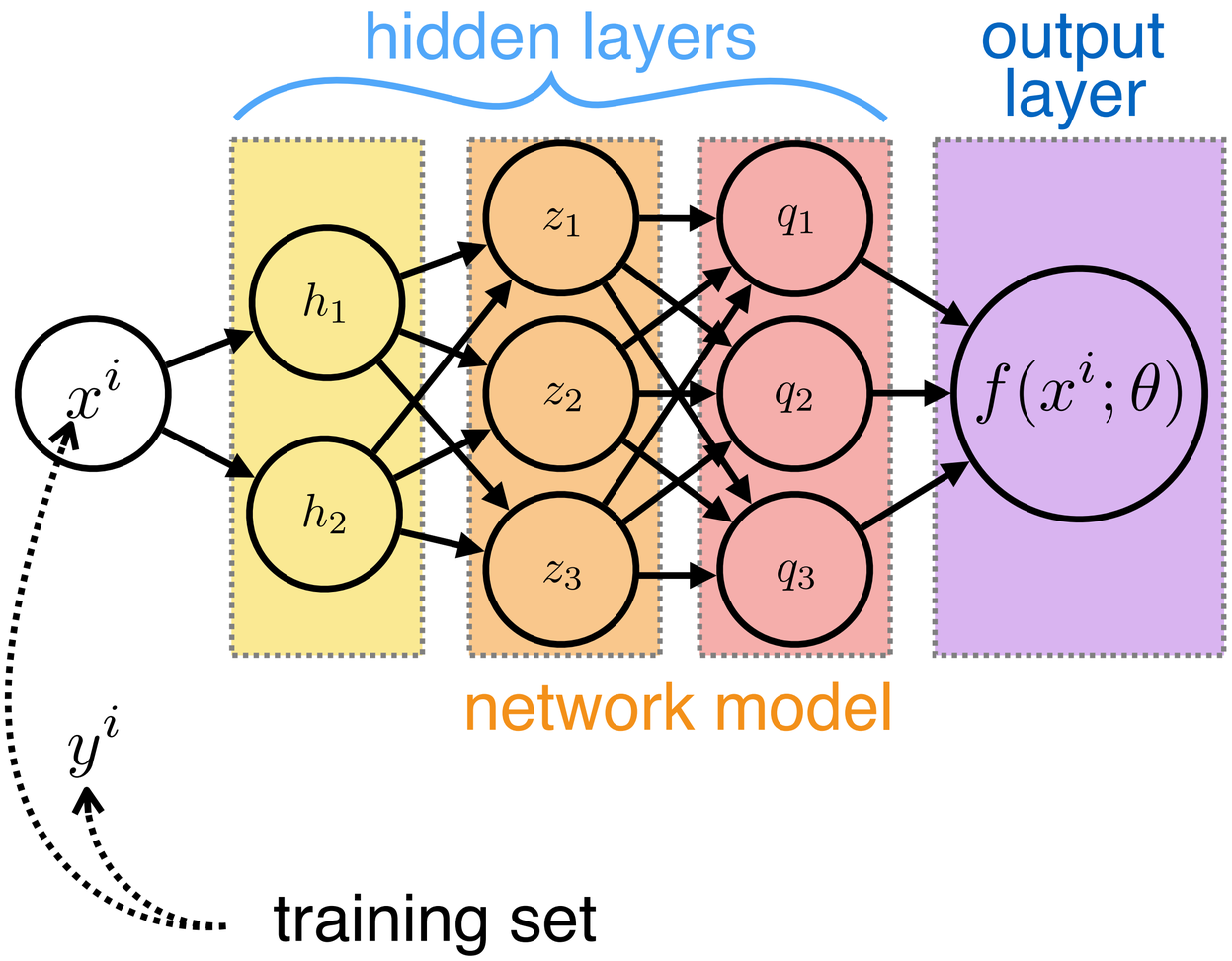
Step-by-Step Analysis



Step-by-Step Analysis



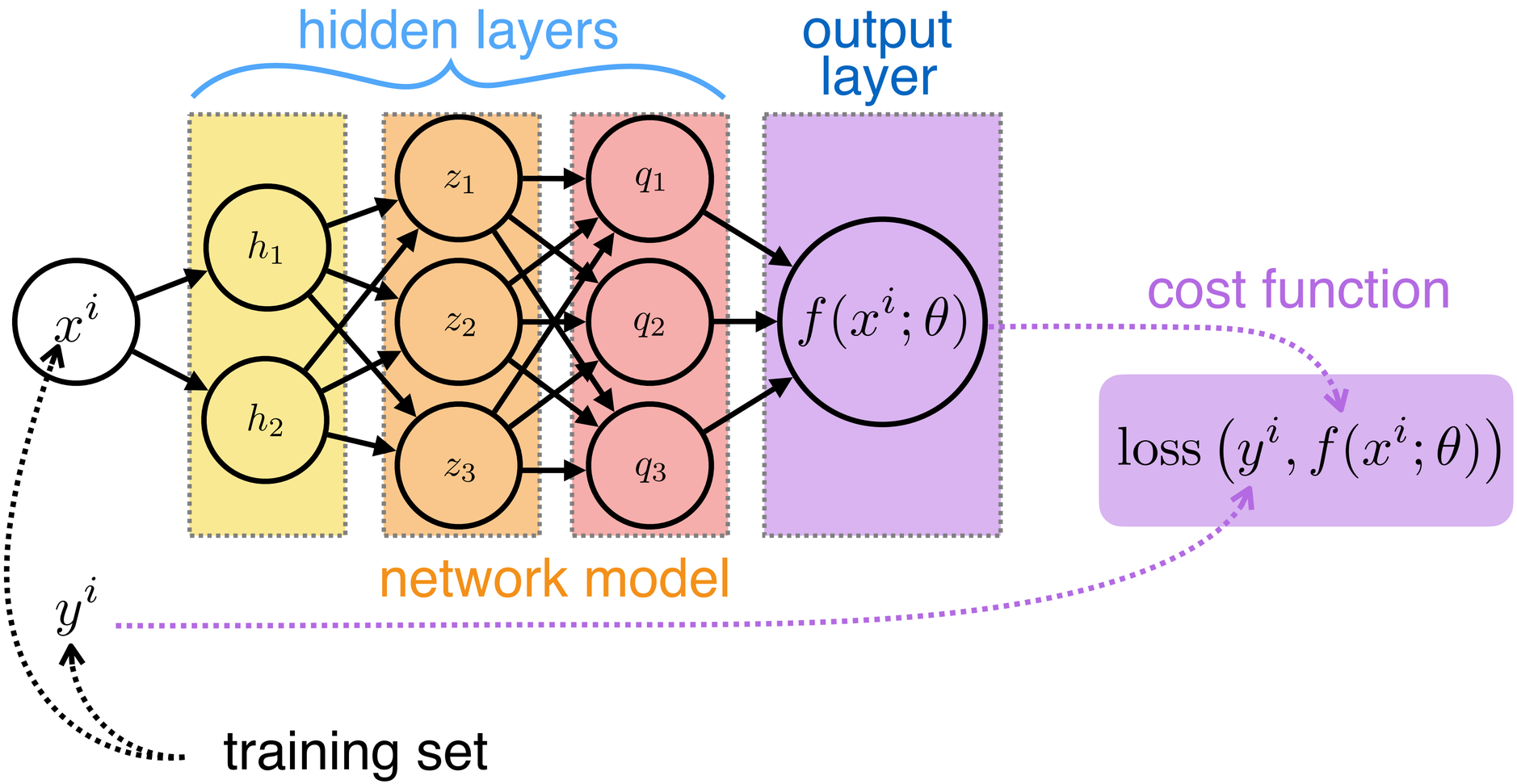
Step-by-Step Analysis



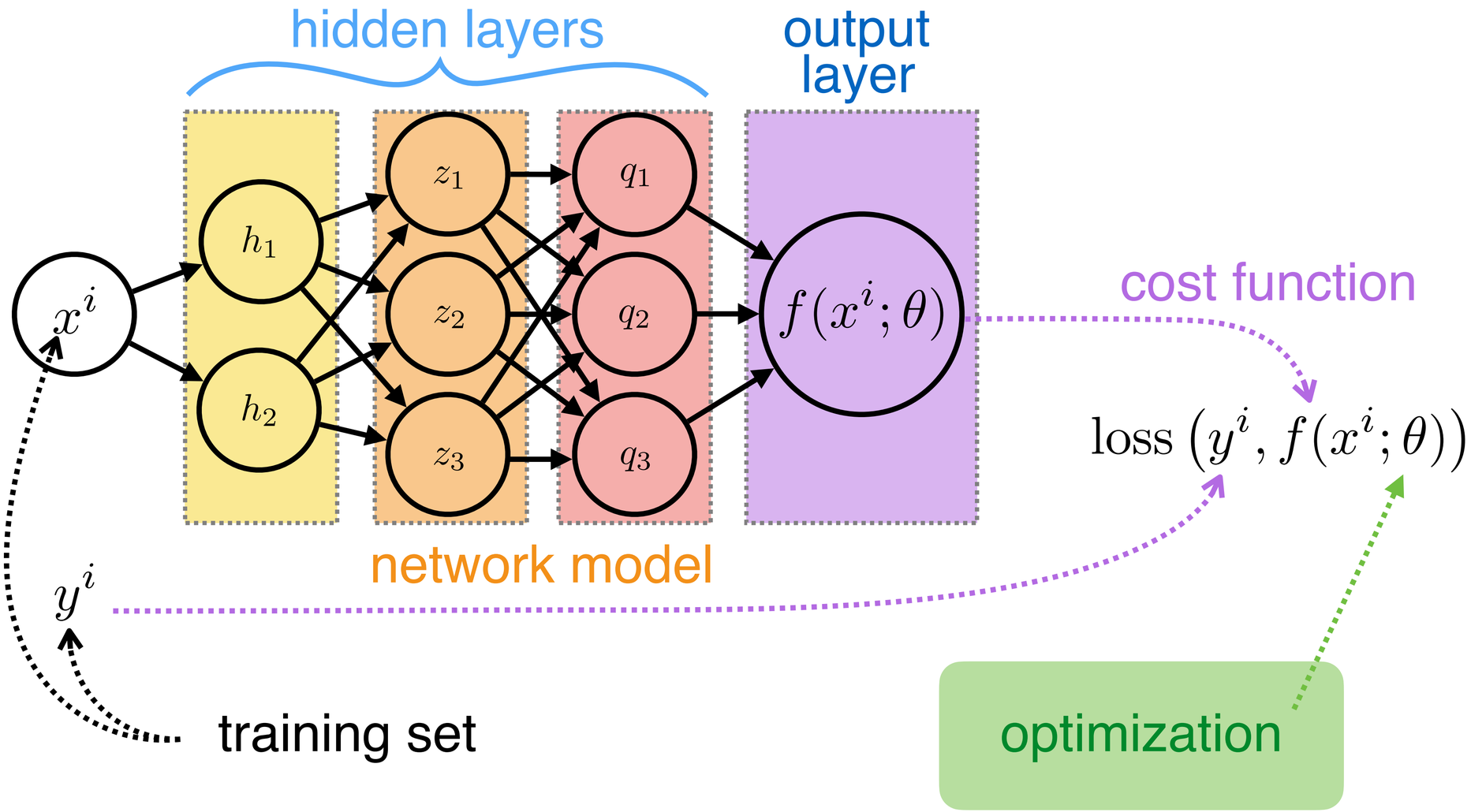
cost function

$$\text{loss}\left(y^i, f(x^i;\theta)\right)$$

Step-by-Step Analysis



Step-by-Step Analysis



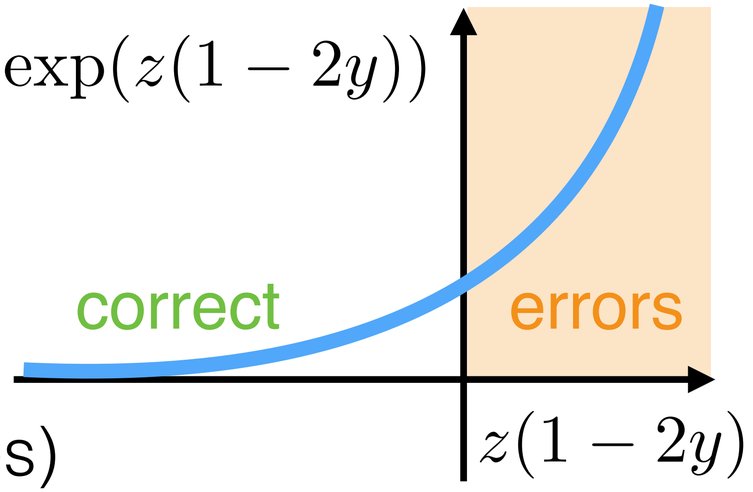
Cost Function

- Based on the **conditional distribution** $p_{\text{model}}(y|x;\theta)$
  - Maximum Likelihood (i.e., **cross-entropy** between model pdf and data pdf)

$$\min_{\theta} -E_{x,y\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(y|x;\theta)]$$

Saturation

- Functions that saturate (have flat regions) have a very small gradient and slow down gradient descent
- We choose loss functions that have a non flat region when the answer is incorrect (it might be flat otherwise)
- E.g., exponential functions saturate in the negative domain; with a binary variable $y \in \{0, 1\}$ map errors to the nonflat region and then minimize
- The logarithm also helps with saturation (see next slides)



Cost Function

- Based on **conditional statistics** $f(x;\theta)$ of $y|x$
- For example

$$f^* = \arg\min_f E_{x,y\sim\hat{p}_{\text{data}}} |y - f(x;\theta)|^2$$

gives the conditional mean

$$f^* = \arg\min_f E_{y\sim\hat{p}_{\text{data}}(y|x)}[y]$$

Output Units

- The choice of the output representation (e.g., a probability vector or the mean estimate) determines the cost function

- Let us denote with

$$h = f(x; \theta)$$

the output of the layer before the output unit

Linear Units

- With a little abuse of terminology, linear units include **affine transformations**

$$\hat{y} = W^\top h + b$$

  can be seen as the mean of the conditional Gaussian distribution (in the Maximum Likelihood loss)

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- The Maximum Likelihood loss becomes

$$-\log p(y|\hat{y}) = |y - \hat{y}|^2 + \text{const}$$

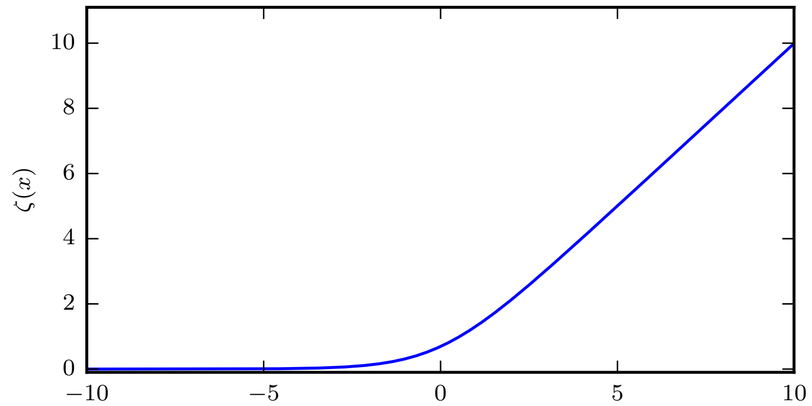
Softplus

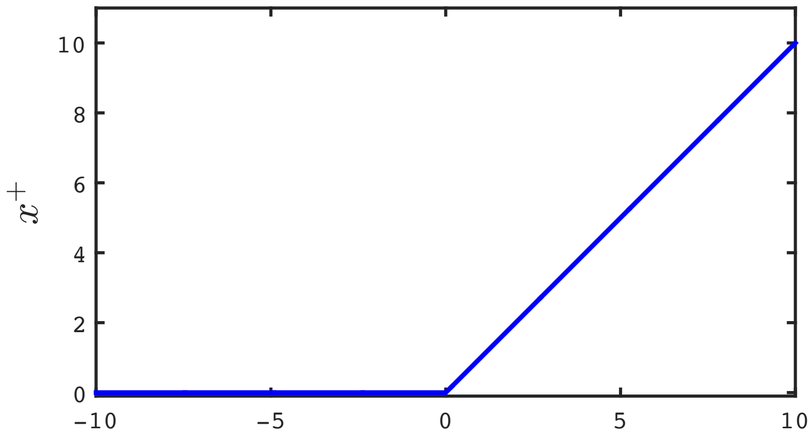
- The **softplus** function is defined as

$$\zeta(x) = \log(1 + \exp(x))$$

and it is a smooth approximation of the Rectified Linear Unit (ReLU)

$$x^+ = \max(0, x)$$





Sigmoid Units

- Use to predict binary variables or to predict the probability of binary variables

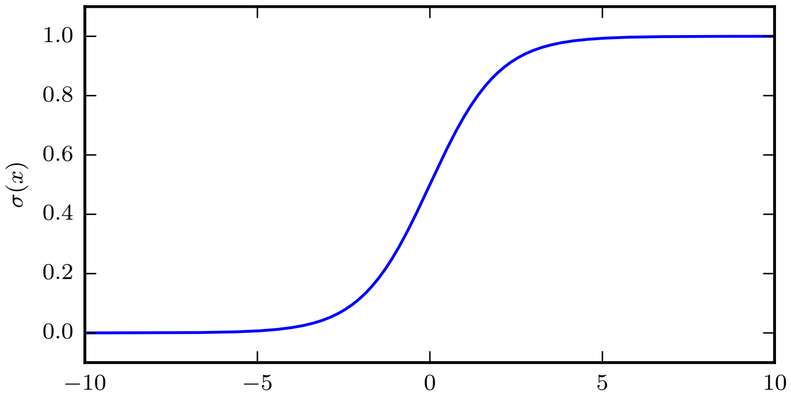
$$p(y = 0|x) \in [0, 1]$$

- The sigmoid unit defines a suitable mapping and has no flat regions (useful in gradient descent)

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

where we have used the **logistic sigmoid** function

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



Bernoulli Parametrization

- Let $z = w^\top h + b$. Then, we can define the Bernoulli distribution

$$p(y|z) = \sigma((2y-1)z)$$

- The loss function with Maximum Likelihood is then

$$-\log p(y|z) = \zeta((1-2y)z) \simeq \max(0, (1-2y)z)$$

and saturation occurs only when the output is correct (y=0 and z<0 or y=1 and z>0)

Smoothed Max

- An extension to the softplus function is the smoothed max

$$\log \sum_j \exp(z_j)$$

  which gives a smooth approximation to $\max_j z_j$

- If we rewrite the softplus function as

$$\log(1 + \exp(z)) = \log(\exp(0) + \exp(z))$$

  we can see that it is the case with $z_1 = 0, z_2 = z$

Softmax Units

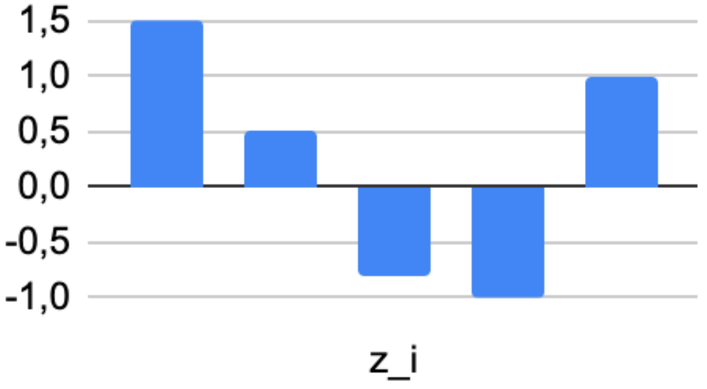
- An extension of the logistic sigmoid to multiple variables
- Used as the output of a multi-class classifier
- The **Softmax** function is defined as

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

- Shift-invariance: $\mathrm{softmax}(z + \mathbf{1}c) = \mathrm{softmax}(z)$

  allows the numerically stable implementation $\mathrm{softmax}(z - \max_j z_j) = \mathrm{softmax}(z)$

Softmax Units



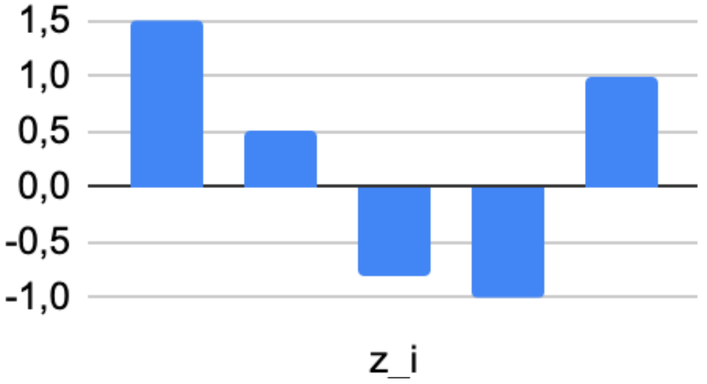
- An extension of the logistic sigmoid to multiple variables
- Used as the output of a multi-class classifier
- The **Softmax** function is defined as

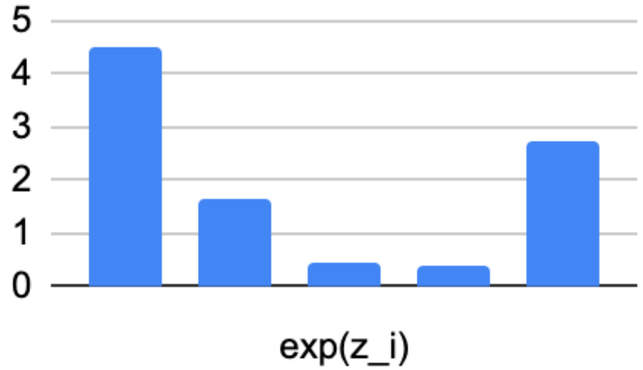
$$\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Shift-invariance: $\mathrm{softmax}(z + \mathbf{1}c) = \mathrm{softmax}(z)$

  allows the numerically stable implementation $\mathrm{softmax}(z - \max_j z_j) = \mathrm{softmax}(z)$

Softmax Units





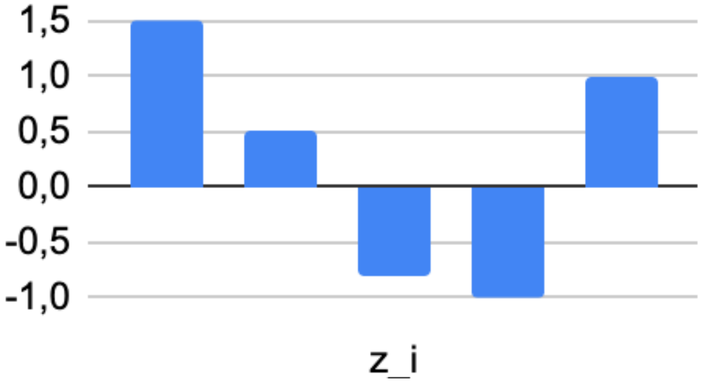
- An extension of the logistic sigmoid to multiple variables
- Used as the output of a multi-class classifier
- The **Softmax** function is defined as

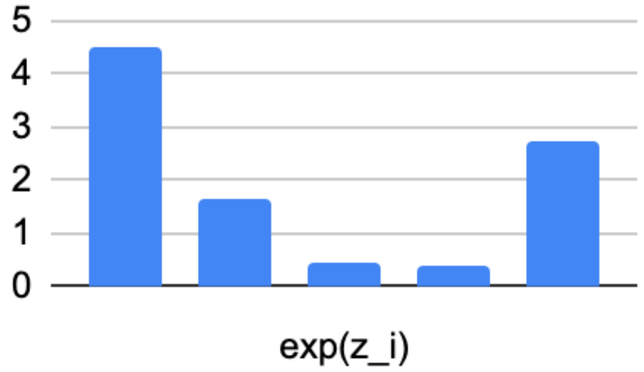
$$\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

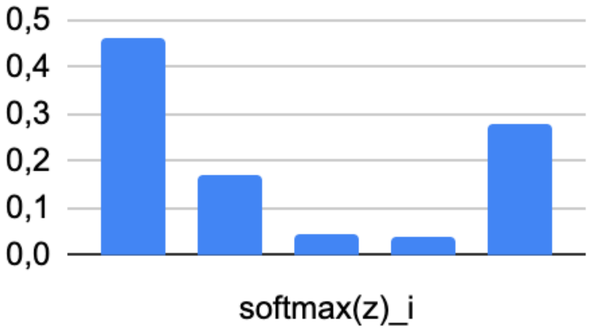
- Shift-invariance: $\mathrm{softmax}(z + \mathbf{1}c) = \mathrm{softmax}(z)$

  allows the numerically stable implementation $\mathrm{softmax}(z - \max_j z_j) = \mathrm{softmax}(z)$

Softmax Units







- An extension of the logistic sigmoid to multiple variables
- Used as the output of a multi-class classifier
- The **Softmax** function is defined as

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

- Shift-invariance: $\mathrm{softmax}(z + \mathbf{1}c) = \mathrm{softmax}(z)$

  allows the numerically stable implementation $\mathrm{softmax}(z - \max_j z_j) = \mathrm{softmax}(z)$

Softmax Units

- In Maximum Likelihood we have

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

- Recall the smoothed max, then we can write

$$\log \operatorname{softmax}(z)_i \simeq z_i - \max_j z_j$$

- Maximization, with $i = \arg\max_j z_j$, yields

$$\operatorname{softmax}(z)_i = 1 \qquad \text{and} \qquad \operatorname{softmax}(z)_{j \neq i} = 0$$

Softmax Units

- Softmax is an extension to the logistic sigmoid where we have 2 variables and $z_1 = 0, z_2 = z$

$$p(y = 1|x) = \text{softmax}(z)_1 = \sigma(z_2)$$

- Softmax is a winner-take-all formulation

- Softmax is more related to the arg max function than the max function

General Output Units

- A neural network can be written as a function $f(x;\theta)$
- This function could output the value of y or parameters ω of the pdf of y such that $f(x;\theta) = \omega$
- In this case the loss function (with ML) is

$$-\log p(y;\omega(x))$$

- For example, the parameters could represent the mean and precision of the Gaussian distribution of y

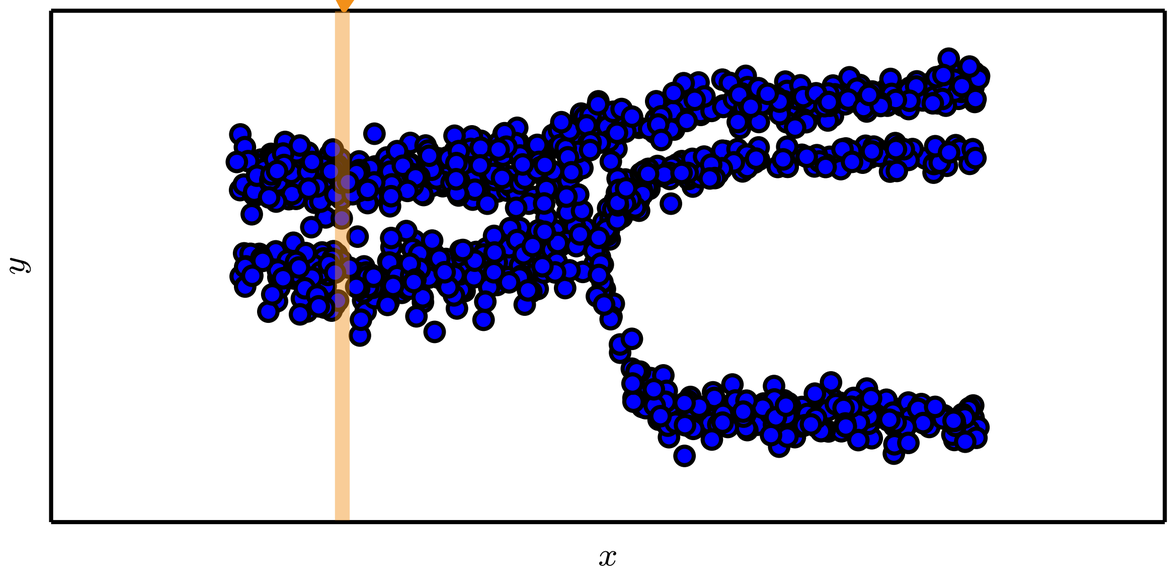
General Output Units

- Mixture density models are used for multimodal probability densities (i.e., multi-peaked outputs)

$$p(y|x) = \sum_i p(c = i|x)\mathcal{N}(y; \mu^i(x), \Sigma^i(x))$$

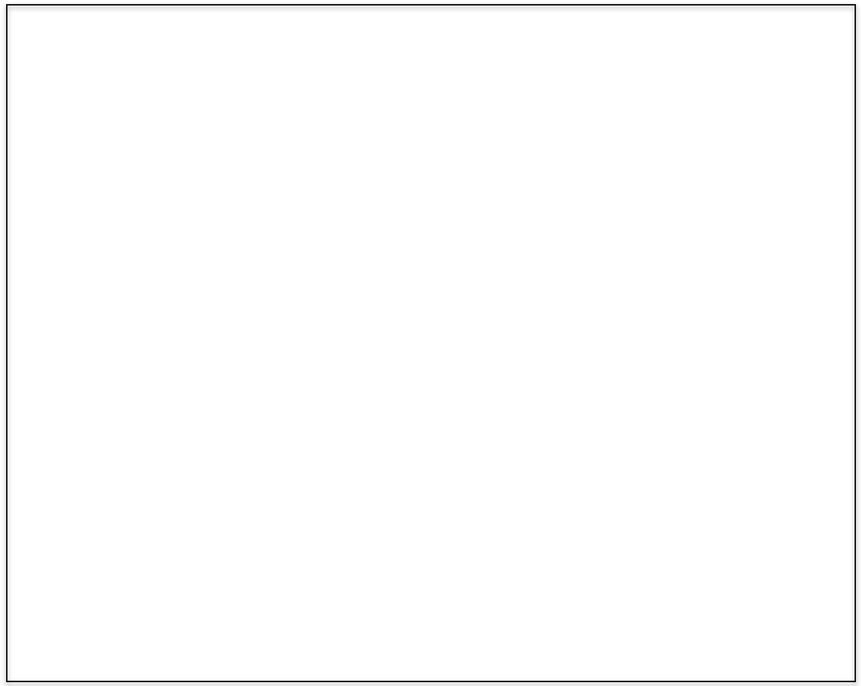
- The parameters include

$$p(c = i|x)$$
$$\mu^i(x)$$
$$\Sigma^i(x)$$



Hidden Units

- The design of a neural network is so far still an art
- The basic principle is the **trial and error** process:
  1. Start from a known model
  2. Modify
  3. Implement and test (go back to 2. if needed)
- A good default choice is ReLUs
- In general the hidden unit picks a g for $h(x) = g(W^\top x + b)$

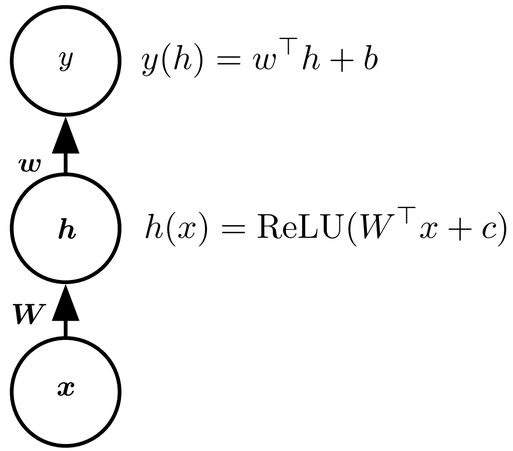


Hidden Units

- The design of a neural network is so far still an art
- The basic principle is the **trial and error** process:
  1. Start from a known model
  2. Modify
  3. Implement and test (go back to 2. if needed)
- A good default choice is ReLUs
- In general the hidden unit picks a g for $h(x) = g(W^\top x + b)$

Nonlinear Model

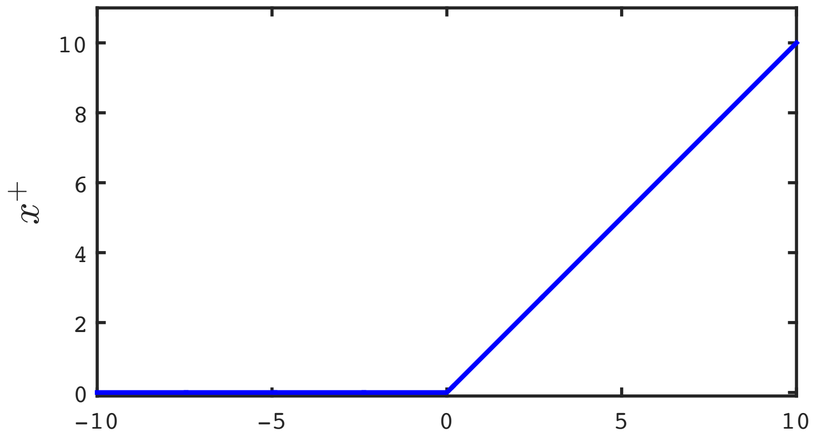
$$f(x; W, c, w, b) = w^\top \max\{0, W^\top x + c\} + b$$



Rectified Linear Units

- ReLUs typically use also an affine transformation

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



- Good initialization is b = 0.1 (initially, a linear layer)
- Negative axis cannot learn due to null gradient
- Generalizations help avoid the null gradient

Leaky ReLUs and More

- A generalization of ReLU is

$$g(z, \alpha) = \max\{0, z\} + \alpha \min\{0, z\}$$

- To avoid a null gradient the following are in use

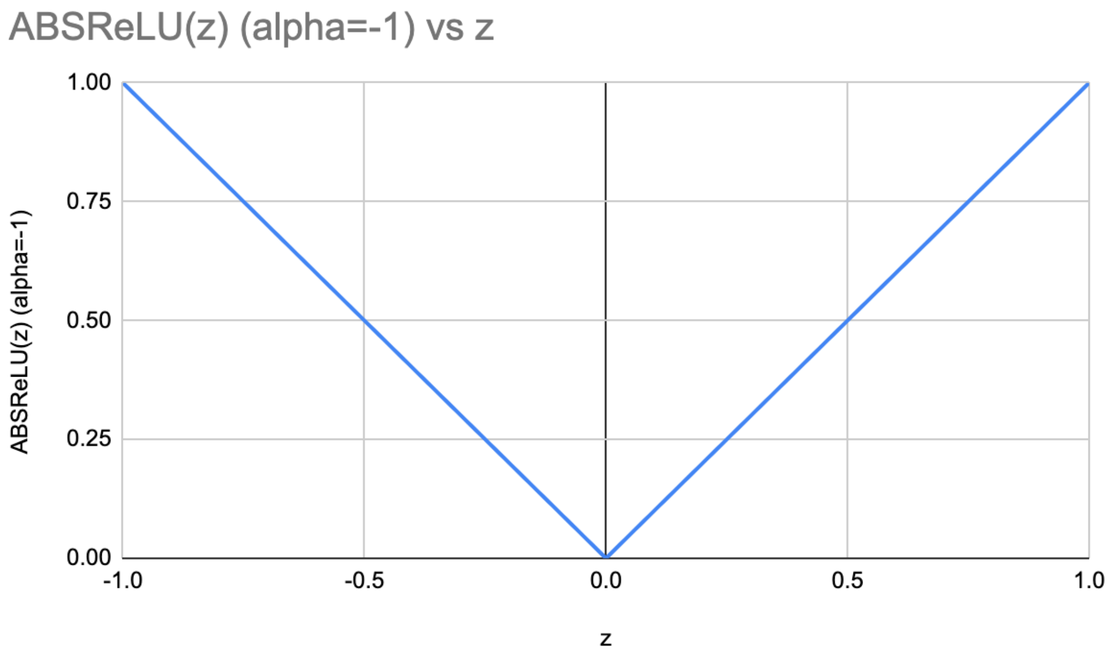
1. Absolute value rectification $\alpha = -1$
2. Leaky ReLU $\alpha = 0.01$
3. Parametric ReLU α learnable
4. Maxout Units $g(z)_i = \max_{j \in S_i} z_j$

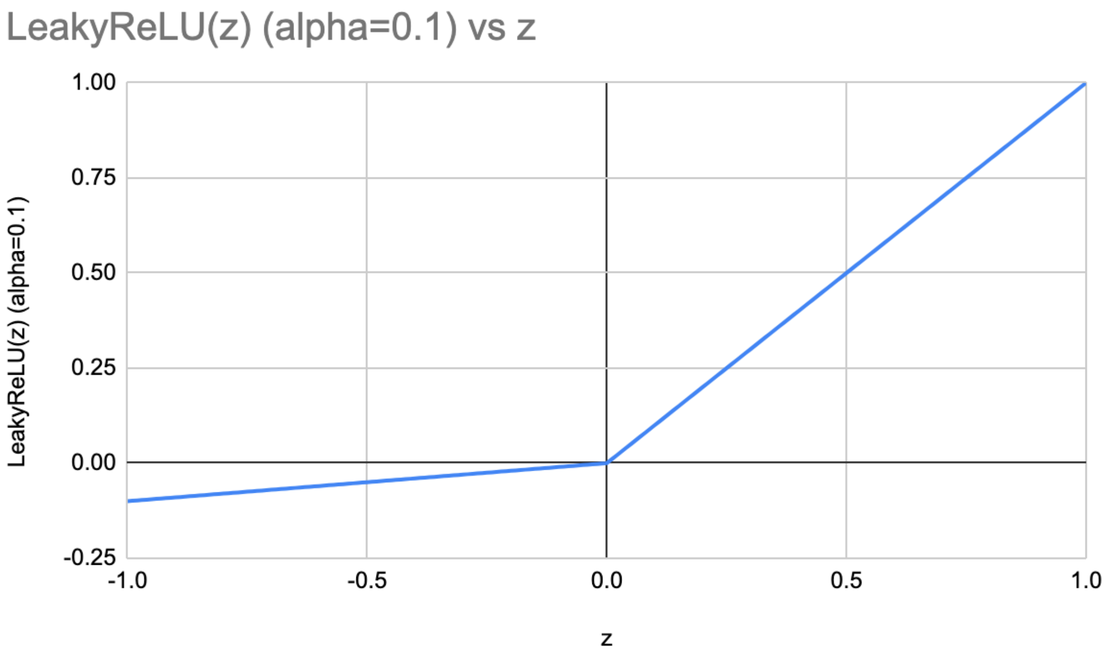
$\cup_i S_i = [1, \ldots, m]$

$S_i \cap S_j = \varnothing \quad i \neq j$

Leaky ReLUs and More

- A generalization of ReLU is $g(z, \alpha) = \max\{0, z\} + \alpha \min\{0, z\}$

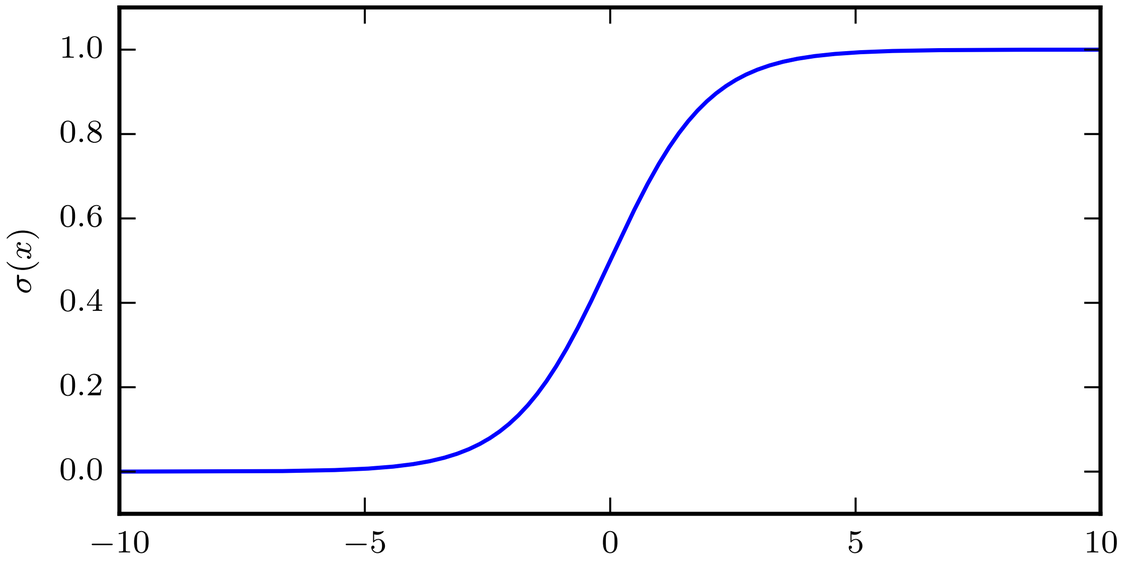




Sigmoid

- The sigmoid is defined as

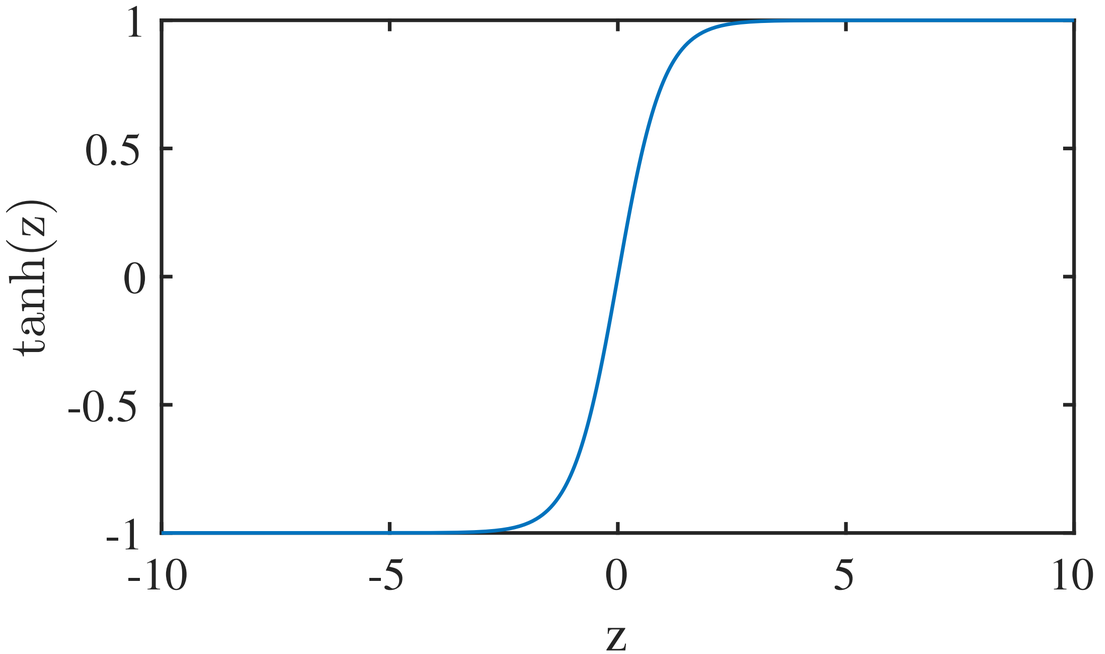
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Hyperbolic Tangent

- The hyperbolic tangent is defined as

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



and it is related to the logistic sigmoid via $\tanh(z) = 2\sigma(2z) - 1$

Similar to identity near zero - preferred over sigmoid

Other Units

- Linear projection
- Radial Basis Functions
- Softplus
- Hard tanh

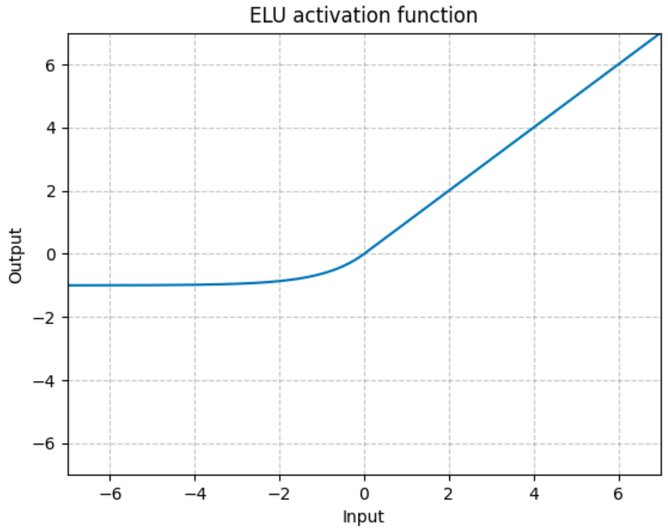
$$W = VU$$

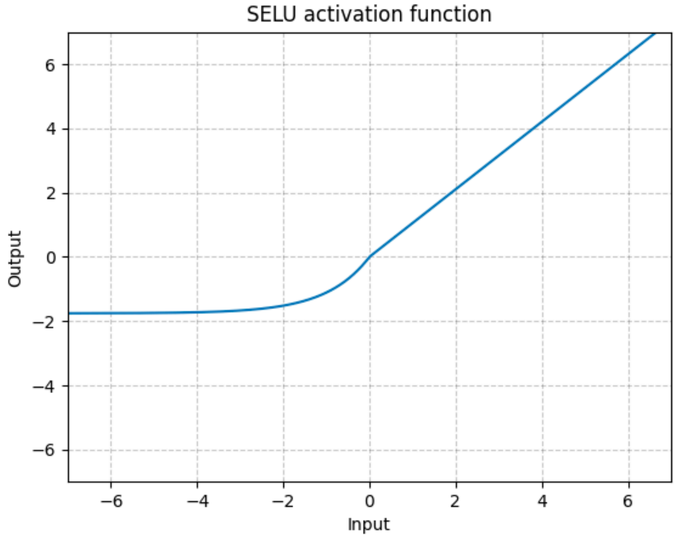
$$h_i(x) = \exp\left(-\frac{1}{\sigma_i^2}\left|x - W_i\right|^2\right)$$

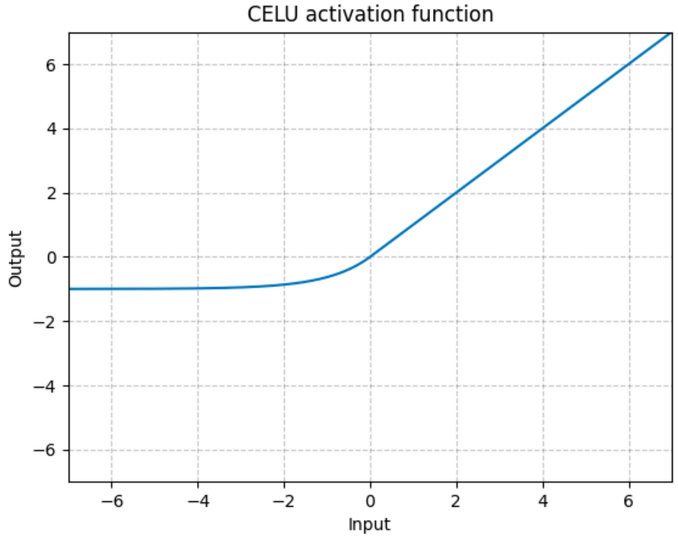
$$g(z) = \zeta(z) = \log(1 + \exp(z))$$

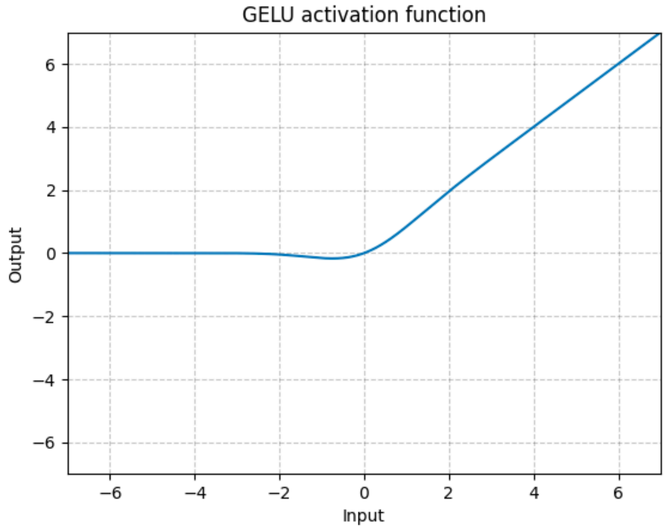
$$g(z) = \max\{-1, \min\{+1, z\}\}$$

…other units









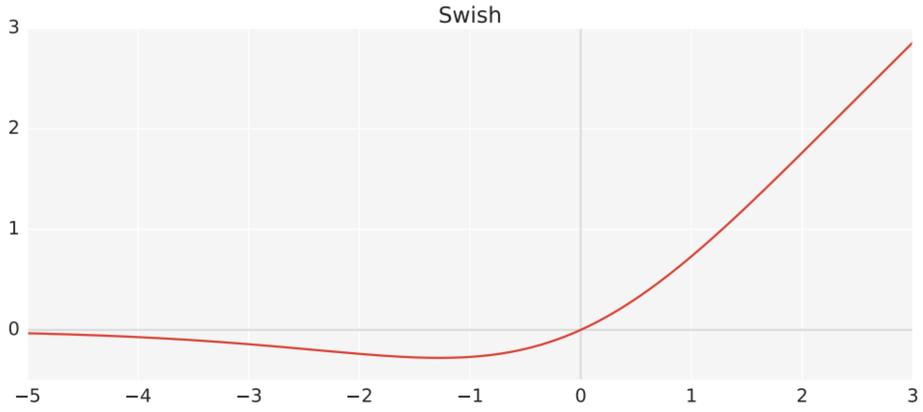


Figure 1: The Swish activation function.

Network Design

- The **network architecture** is the overall structure of the network: number of units and their connectivity

- Today, the design for a task must be found experimentally via a careful analysis of the training and validation error

Universal Approximation

- **Theorem**
  A feedforward network with a linear output layer and enough (but at least one) hidden nonlinear layers (e.g., the logistic sigmoid unit) can approximate up to any desired precision any (Borel measurable) function between two finite-dimensional spaces

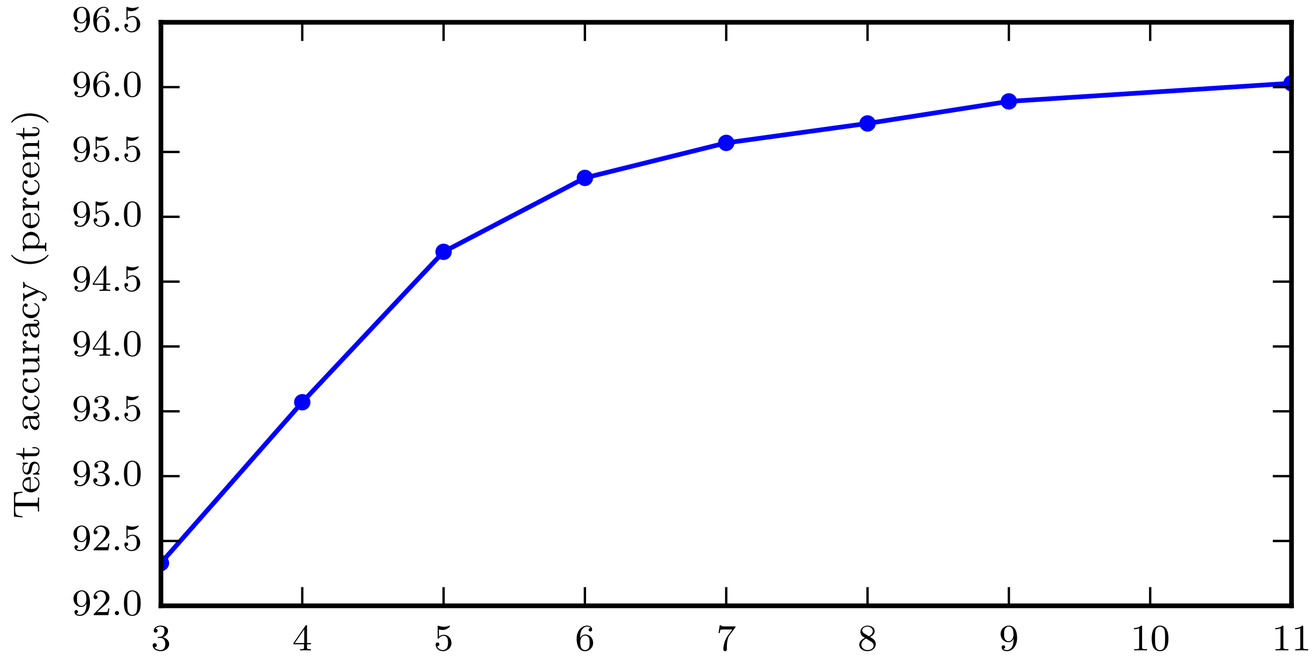
- This means that neural networks provide a **universal representation**/ approximation

Universal Approximation

- However, we are not guaranteed that the learning algorithm will be able to build that representation
- Learning might fail to find some good parameters
- Learning might fail due to overfitting (see “No Free Lunch” theorem)

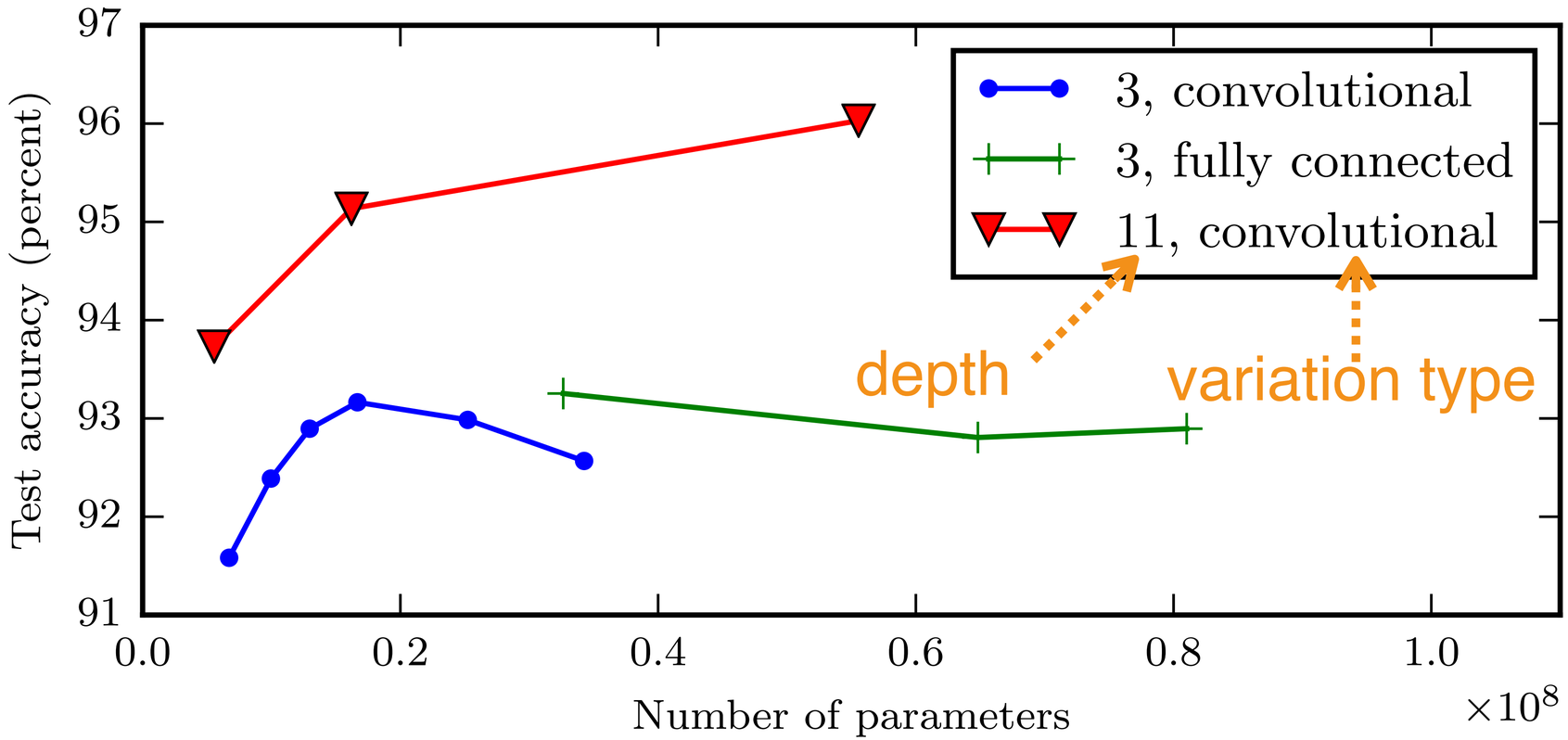
Depth

- A general rule is that depth helps generalization
- It is better to have many simple layers than few highly complex ones



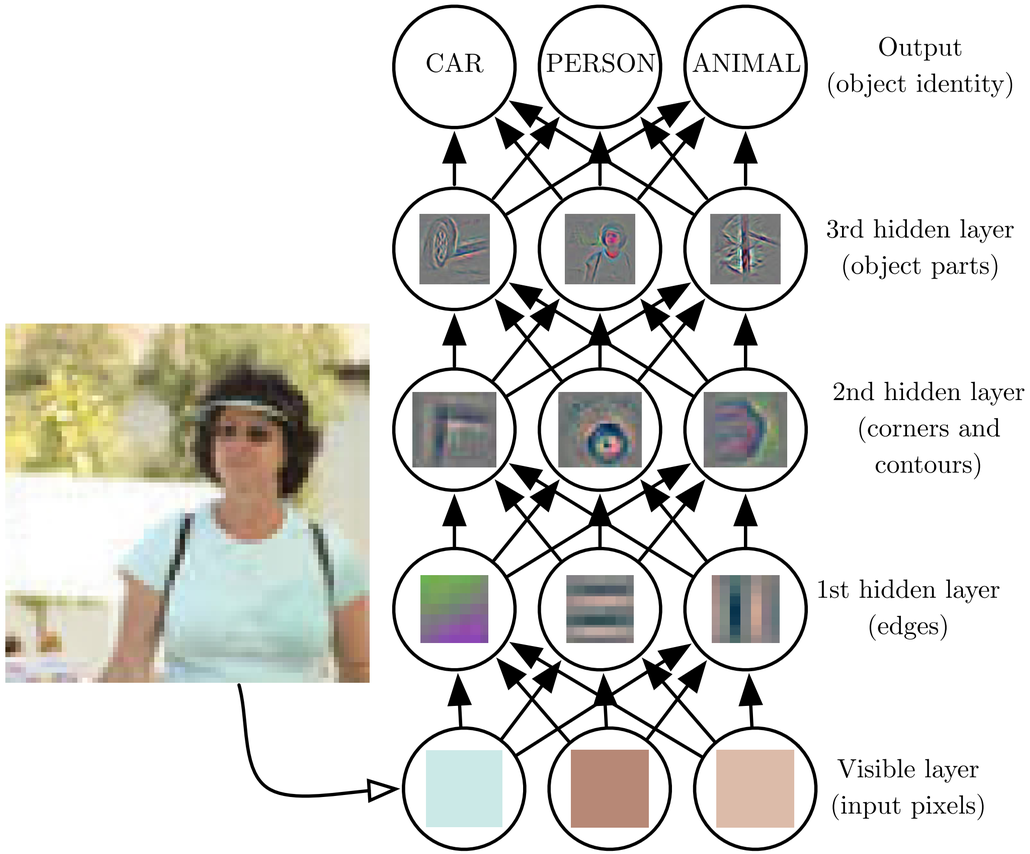
Depth

- Other network modifications do not have the same effect



Depth

- Another interpretation is that depth allows a more gradual abstraction

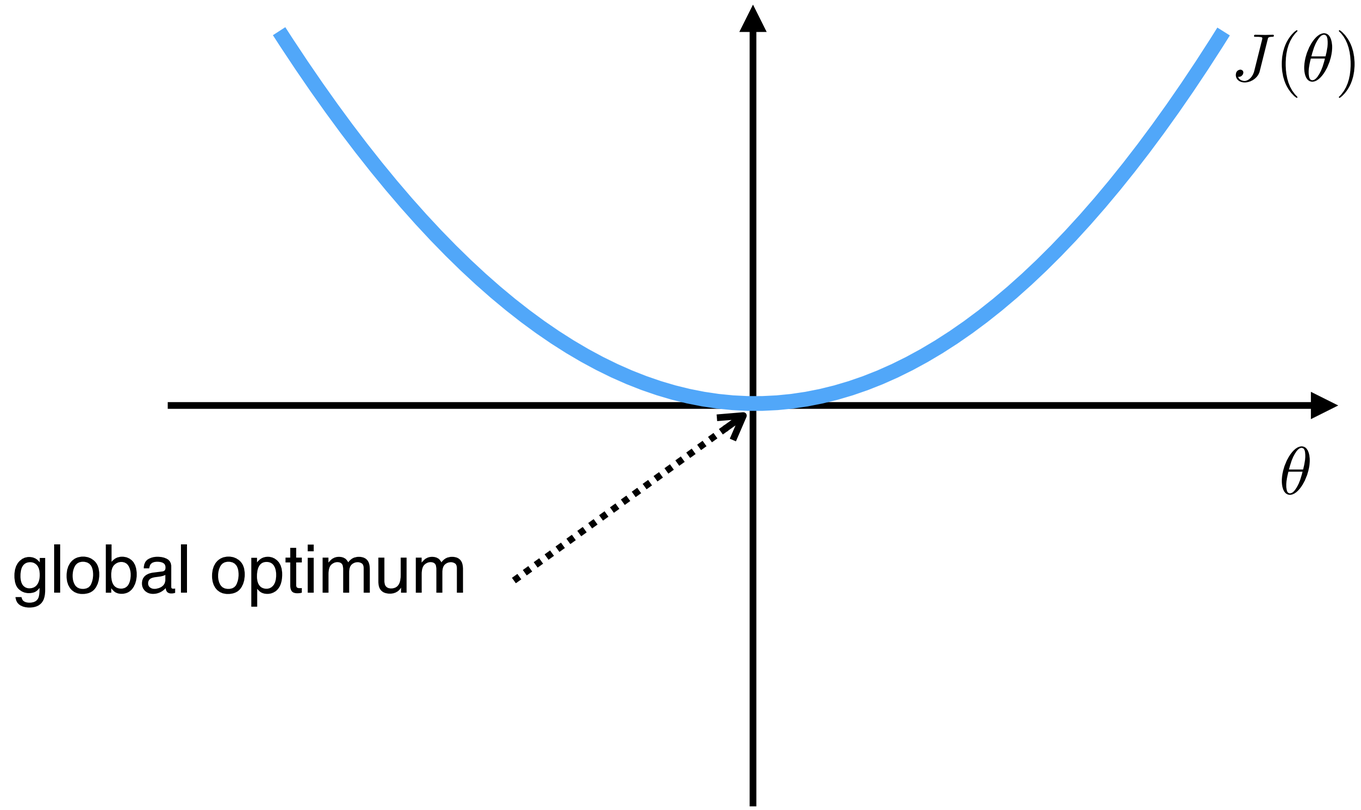


Optimization

- Given a task we define
  - The training data $\{x^i, y^i\}_{i=1,\ldots,m}$
  - A network design $f(x;\theta)$
  - The loss function $J(\theta) = \sum_{i=1}^{m} \text{loss}\left(y^i, f(x^i;\theta)\right)$
- Next, we **optimize** the network parameters θ
- This operation is called **training**

Optimization

- The MSE cost function $J(\theta)$ is **convex** with a linear model



Optimization

- However, since the cost function $J(\theta)$ is typically **non convex** in the parameters, we use an iterative solution

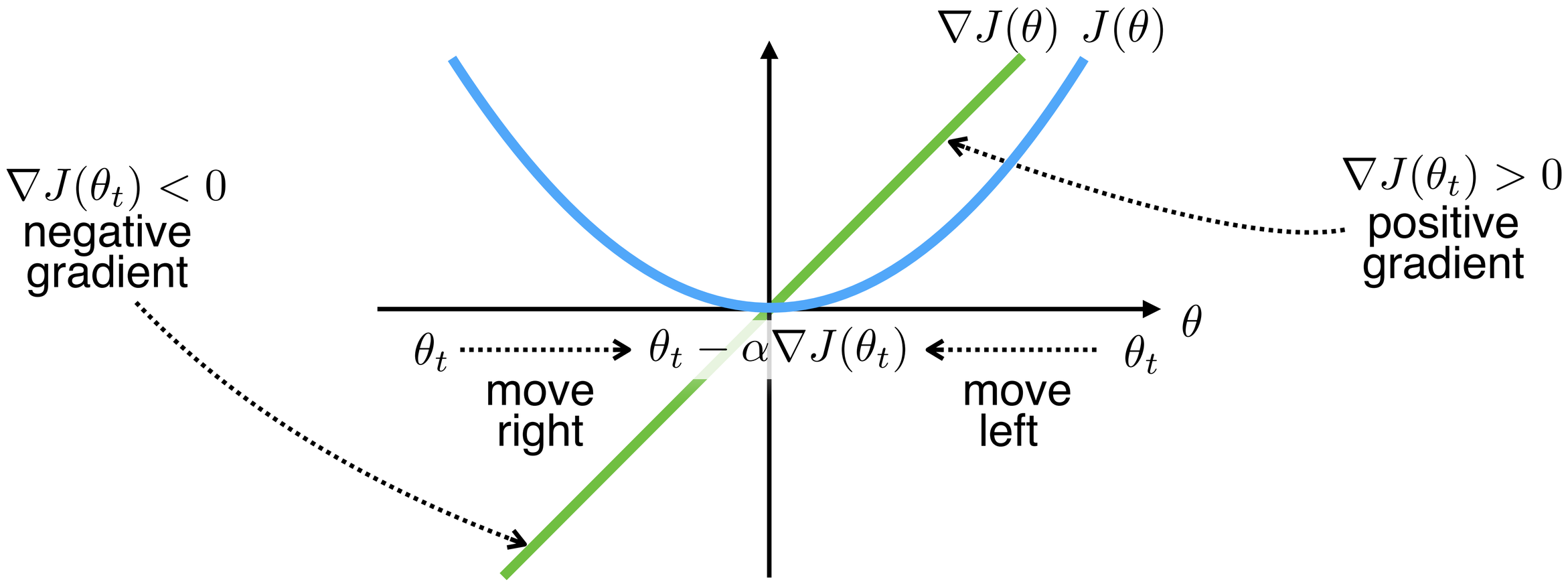
- We consider the **gradient descent** method

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

where $\alpha > 0$ is the learning rate

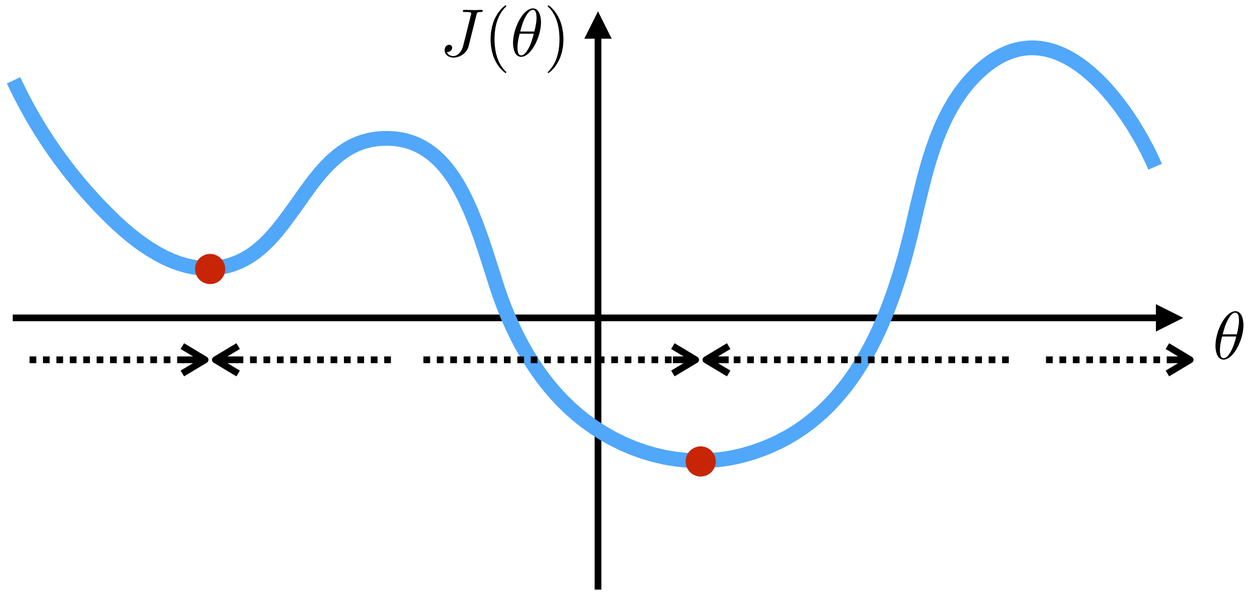
Optimization

gradient descent $\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$



Local Minima

- Does gradient descent reach a (local) minimum even with a non convex function?

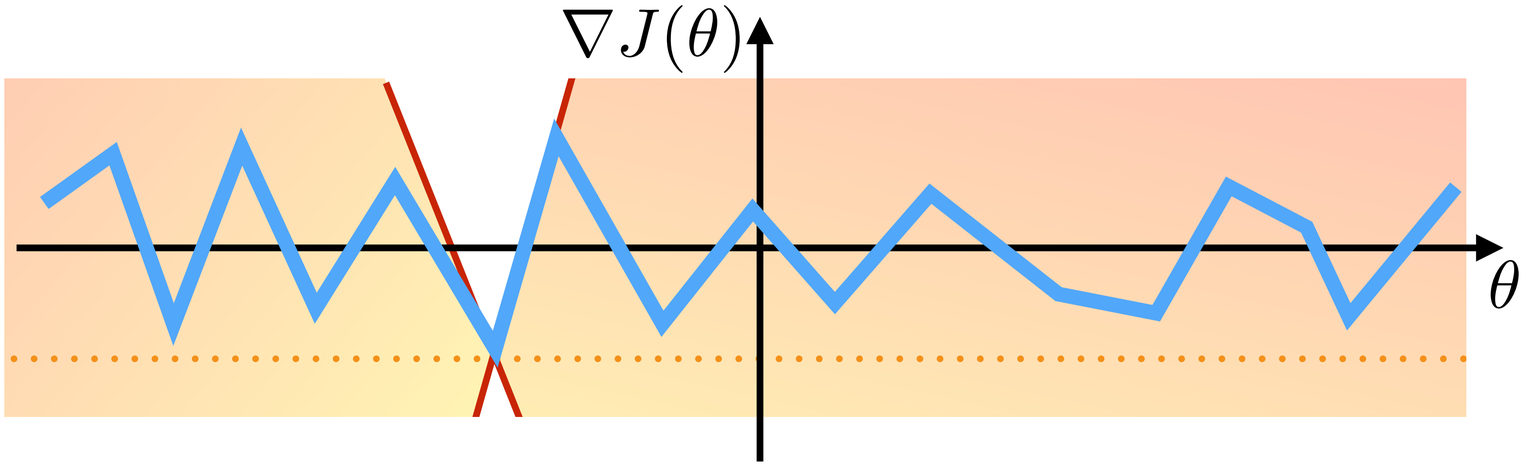


- How do we show that?

Assumptions

- Gradient descent will reach a local minimum under some **constraints** on both the cost function and the learning rate

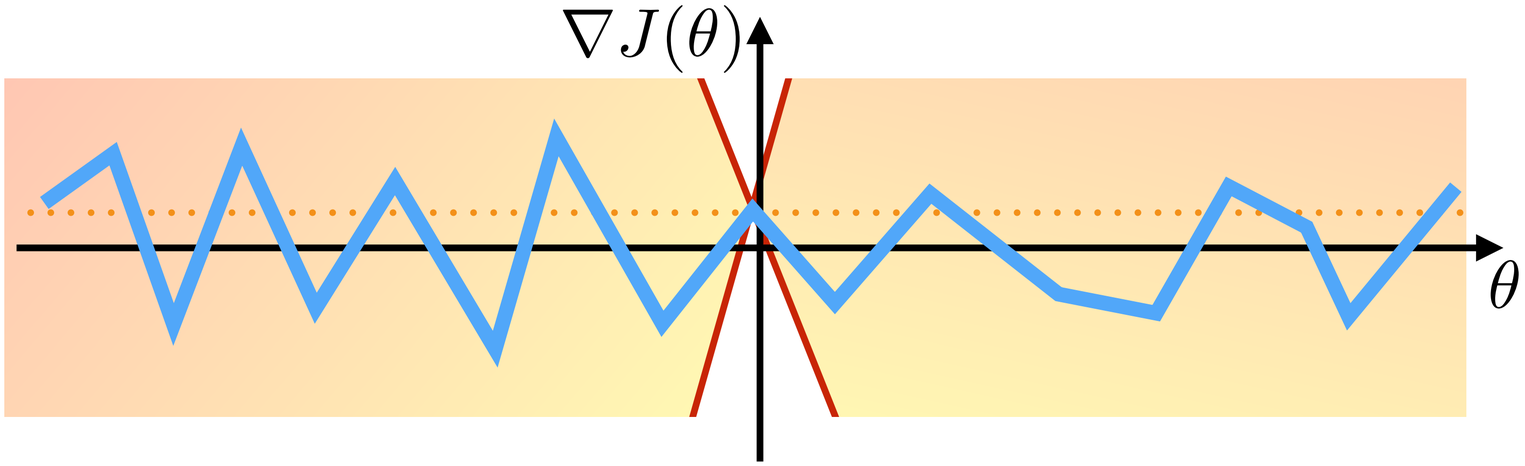
- We show that by using a smoothness assumption called **Lipschitz continuity** on $\nabla J(\theta)$



Assumptions

- Gradient descent will reach a local minimum under some **constraints** on both the cost function and the learning rate

- We show that by using a smoothness assumption called **Lipschitz continuity** on $\nabla J(\theta)$



Convergence

If we assume (Lipschitz continuity)

$$\exists L \geq 0: \quad \left|\nabla J(\theta) - \nabla J(\bar{\theta})\right| \leq L|\theta - \bar{\theta}|, \qquad \forall \theta, \bar{\theta}$$

then for a **small enough learning rate** α the gradient descent iteration will generate a sequence $\theta_1, \ldots, \theta_T$ such that*

$$J(\theta_{t+1}) < J(\theta_t)$$

decreasing

if $\nabla J(\theta_t) \neq 0$; i.e., it will converge to a local minimum.

***See Tutorial 2 of the Machine Learning Course**

Diagnosing GD

- In practice, what do we do when gradient descent does not work?
- Since it must work when the assumptions are true, it must be that the assumptions are violated
- The next step is to determine which assumptions are violated
- There are two: Lipschitzianity and small learning rate

Diagnosis 1/2

- **Case 1: Lipschitzianity**

Diagnosis 1/2

- **Case 1: Lipschitzianity**

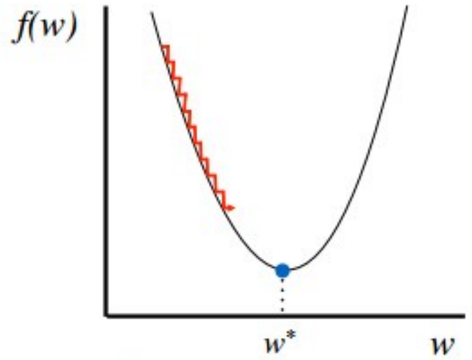
- The cost function does not satisfy the Lipschitz condition for any L

Diagnosis 1/2

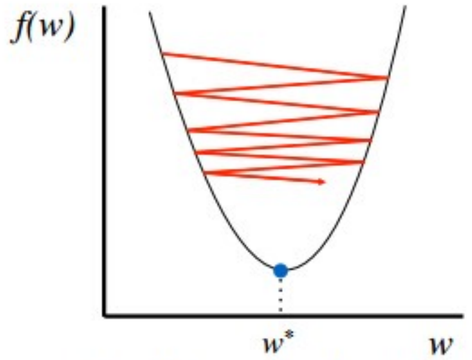
- **Case 1: Lipschitzianity**
- The cost function does not satisfy the Lipschitz condition for any L
- **Solution**: Smooth the cost function until an L exists

Diagnosis 2/2

- **Case 2: Learning rate**



Too small: converge very slowly

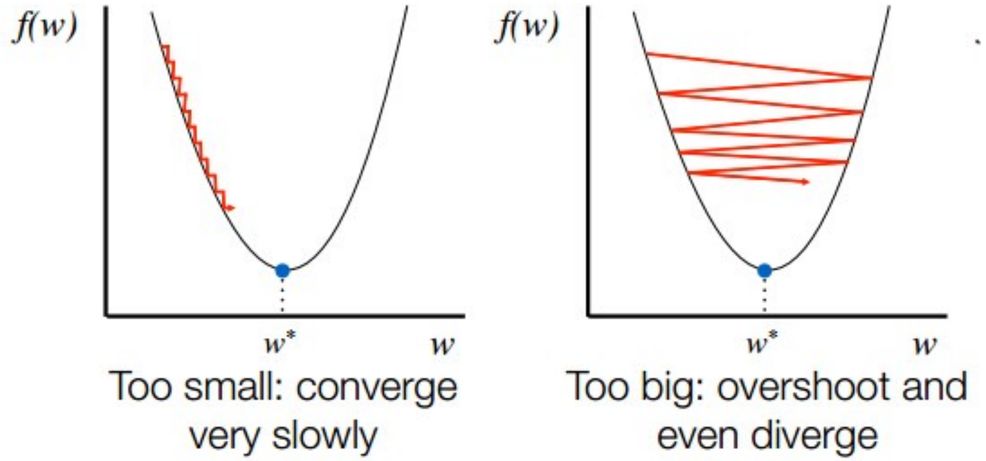


Too big: overshoot and even diverge

Diagnosis 2/2

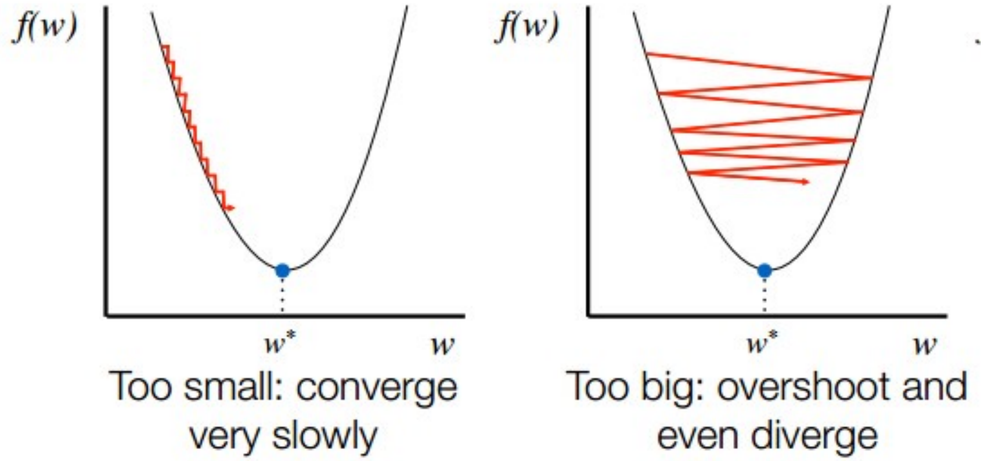
- **Case 2: Learning rate**

- The learning rate is not small enough



Diagnosis 2/2

- **Case 2: Learning rate**
- The learning rate is not small enough
- **Solution**: Make it smaller until gradient descent starts to work



Optimization

- For more efficiency, we use the **stochastic gradient descent** method
- The gradient of the loss function is computed on a small set of samples (a mini-batch) from the training set

$$\tilde{J}(\theta) = \sum_{i \in B \subset [1,\ldots,m]} \text{loss}\left(y^i, f(x^i;\theta)\right)$$

and the iteration is as before

$$\theta_{t+1} = \theta_t - \alpha \nabla \tilde{J}(\theta_t)$$

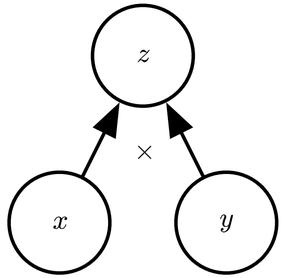
Back-Propagation

- (Stochastic) gradient descent boils down to the calculation of the loss gradient with respect to the parameters
- Due to the compositional structure of the network, the gradient can be computed in many ways
- **Back-propagation** (or simply backprop) refers to a particularly computationally efficient procedure for computing the gradient

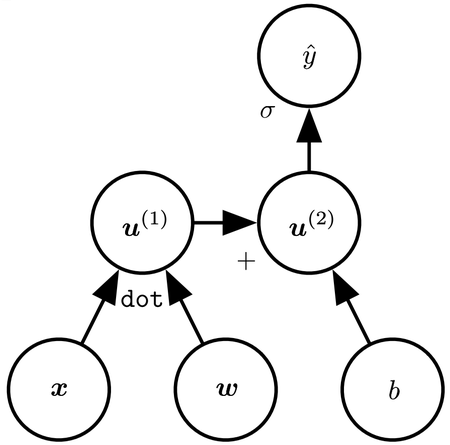
Computational Graphs

- Because neural networks can quickly become very complex, a clear representation is needed
- We use a **computational graph** to formalize the computations
- Each **node** is assigned to a variable (e.g., a scalar, a vector, a matrix, a tensor)
- An **operation** transforms one or more variables into another variable

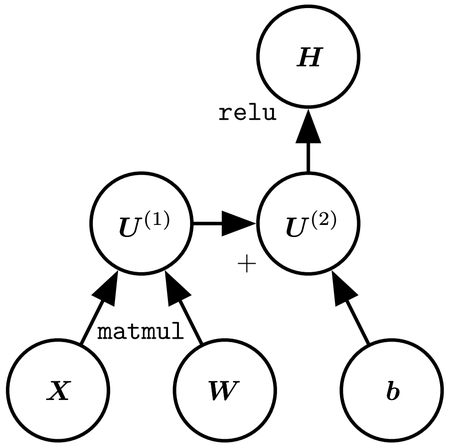
Examples



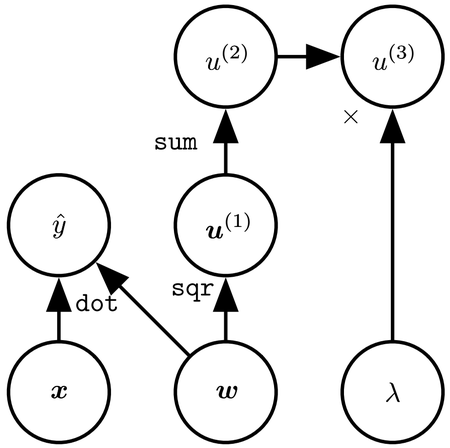
(a)



(b)

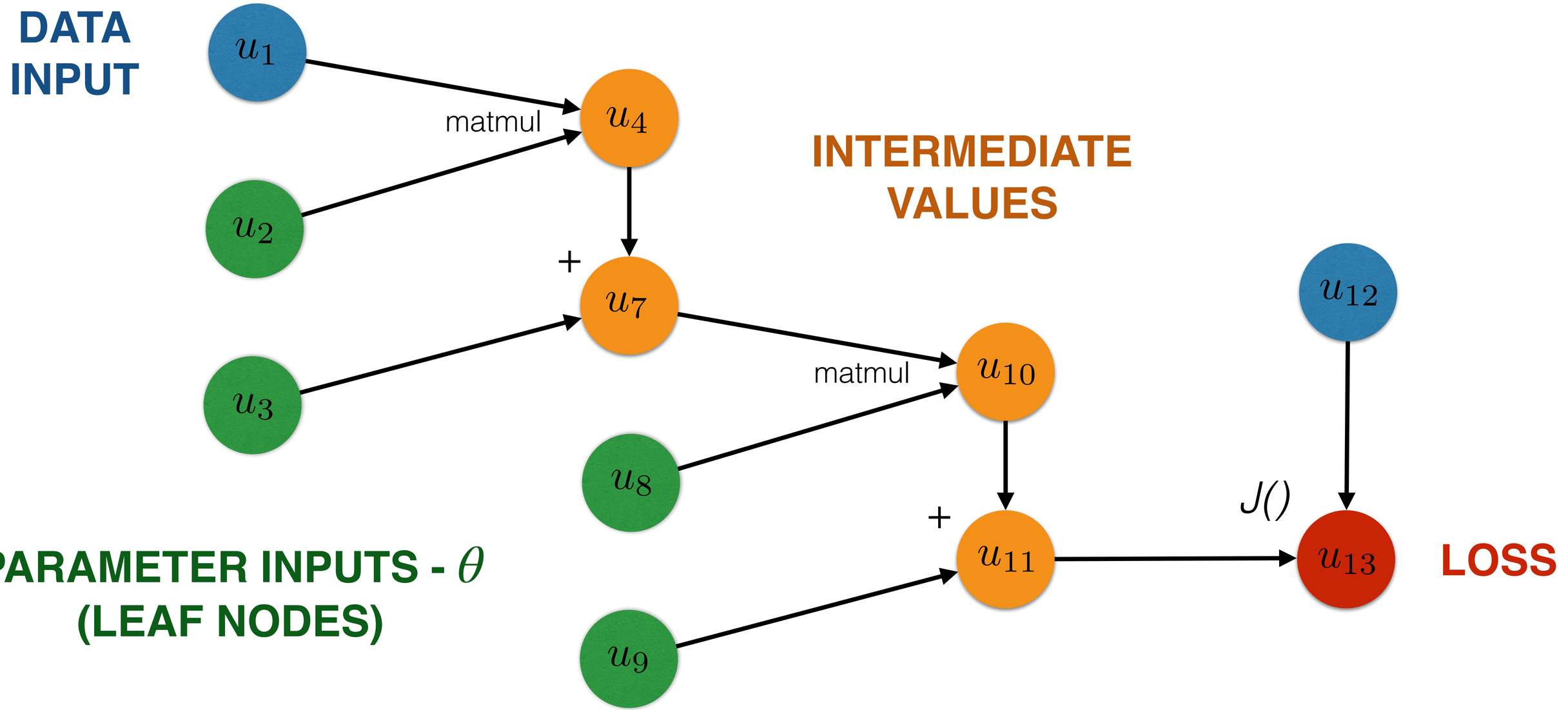


(c)



(d)

The Computational Graph



Computing Gradients

- The main objective is to compute the derivatives of the loss node with respect to the leaf nodes

$$\frac{\partial J}{\partial \theta} = \left\{ \frac{\partial u_{13}}{\partial u_i} \right\} \qquad i = 2,3,8,9$$

- The loss depends on the input nodes through functional composition

$$u_{13} = J(u_{12}, u_{11}) = J(u_{12}, u_{10} + u_9) = J(u_{12}, u_8 \cdot u_7 + u_9) = \dots$$

Chain Rule

- The derivatives of a function composition can be computed via the chain rule
- Consider $y = g(x)$ and $z = f(g(x)) = f(y)$

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

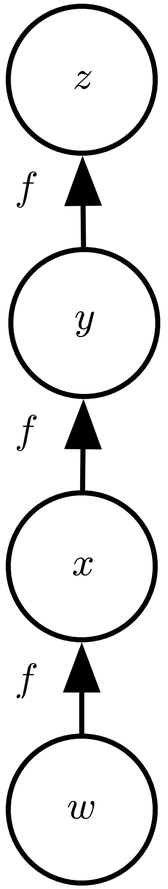
- In the multivariate case we have $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

  or, more compactly $\nabla_x z = (\nabla_x y)^\top \nabla_y z$

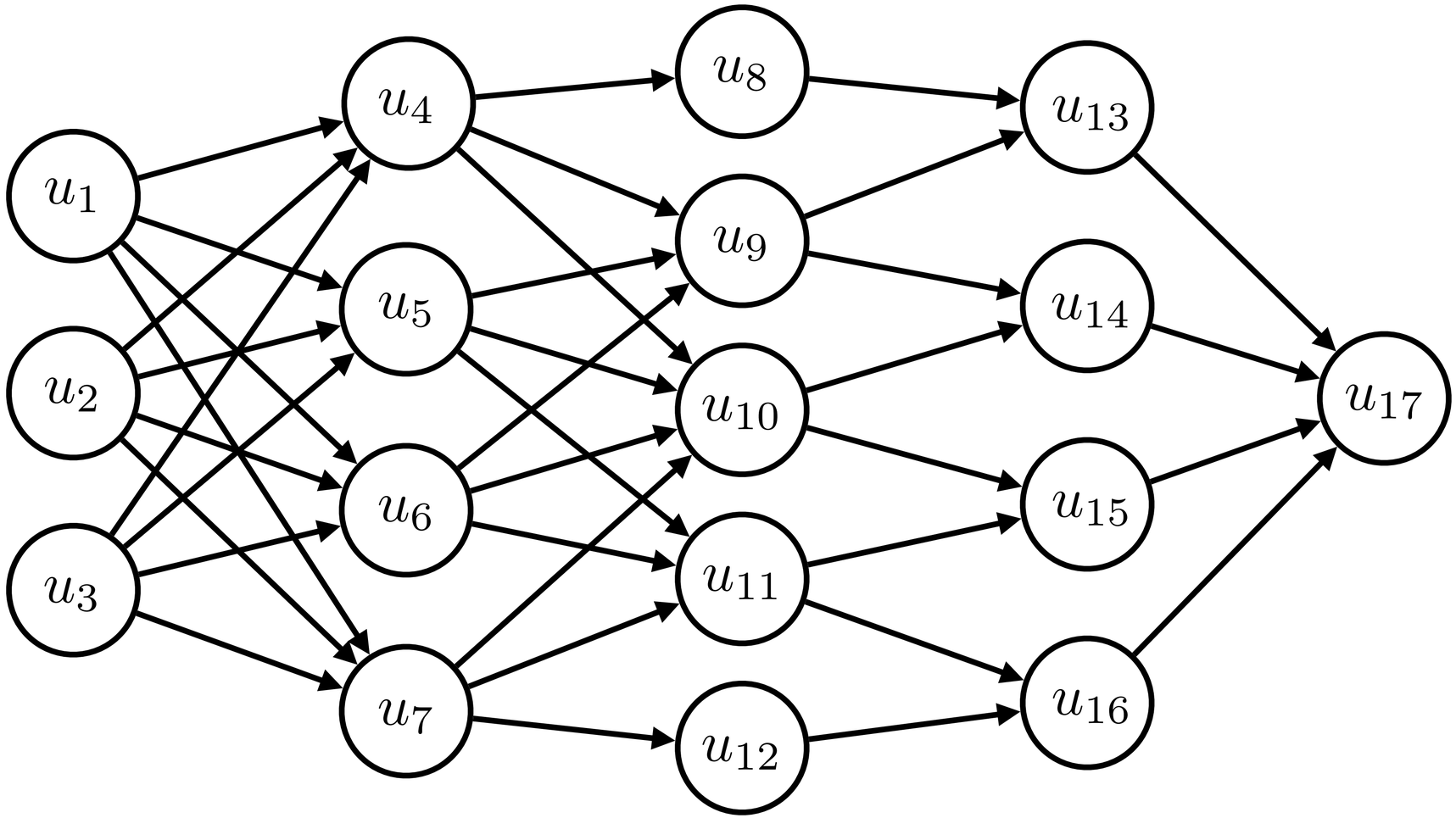
Computational Challenges

- In the chain rule, subexpressions may repeat
- Example

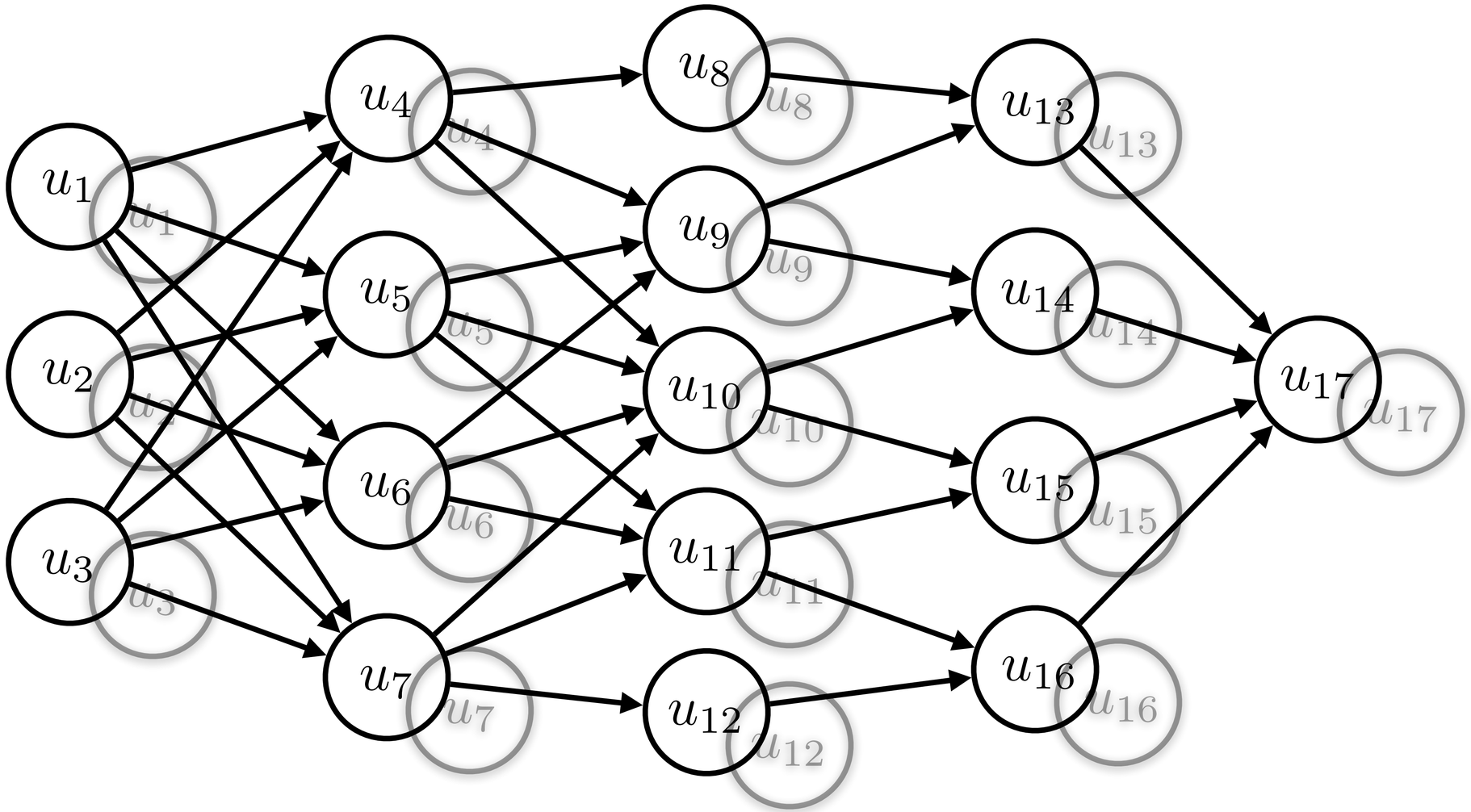
$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$



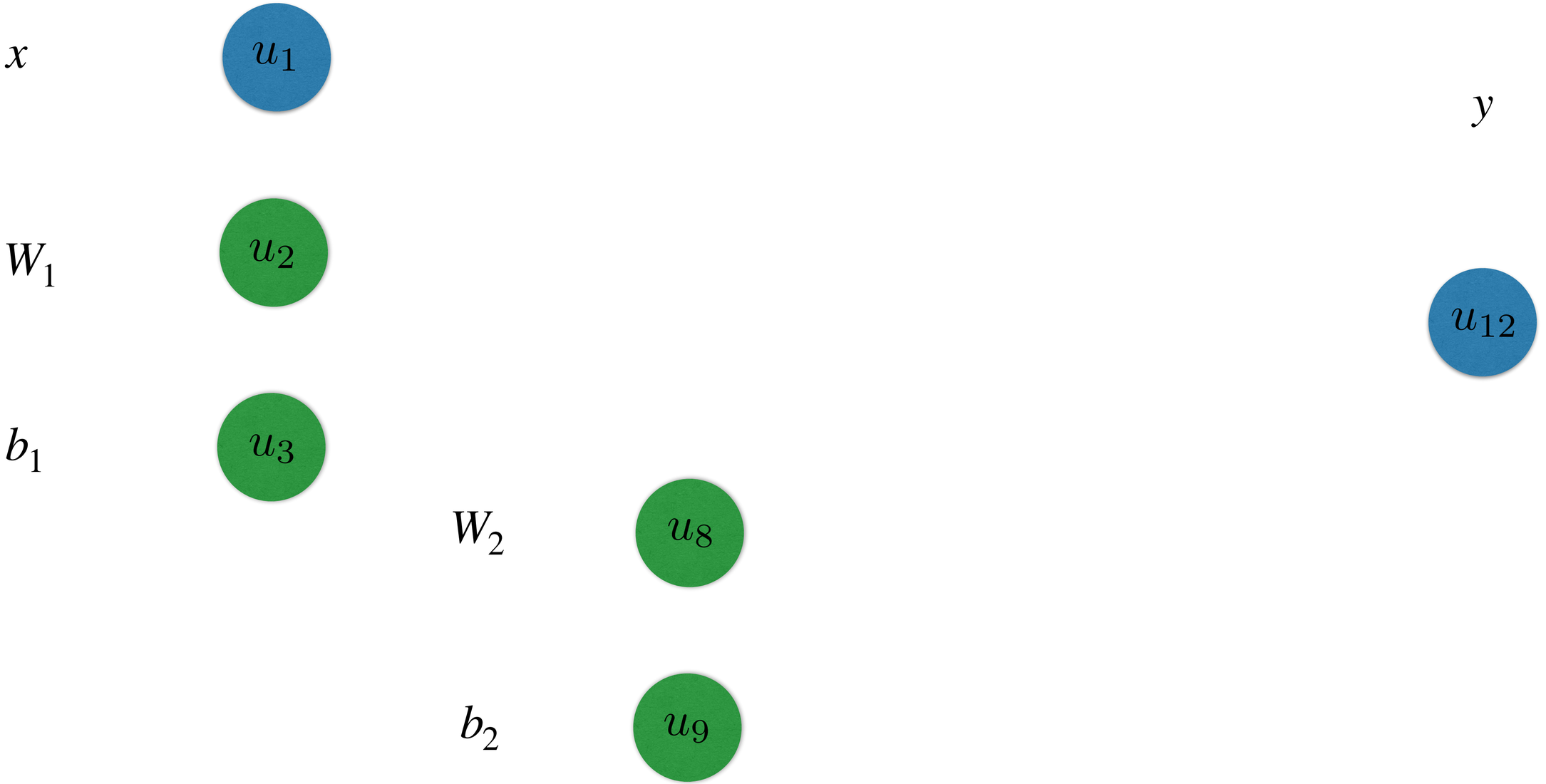
The Computational Graph



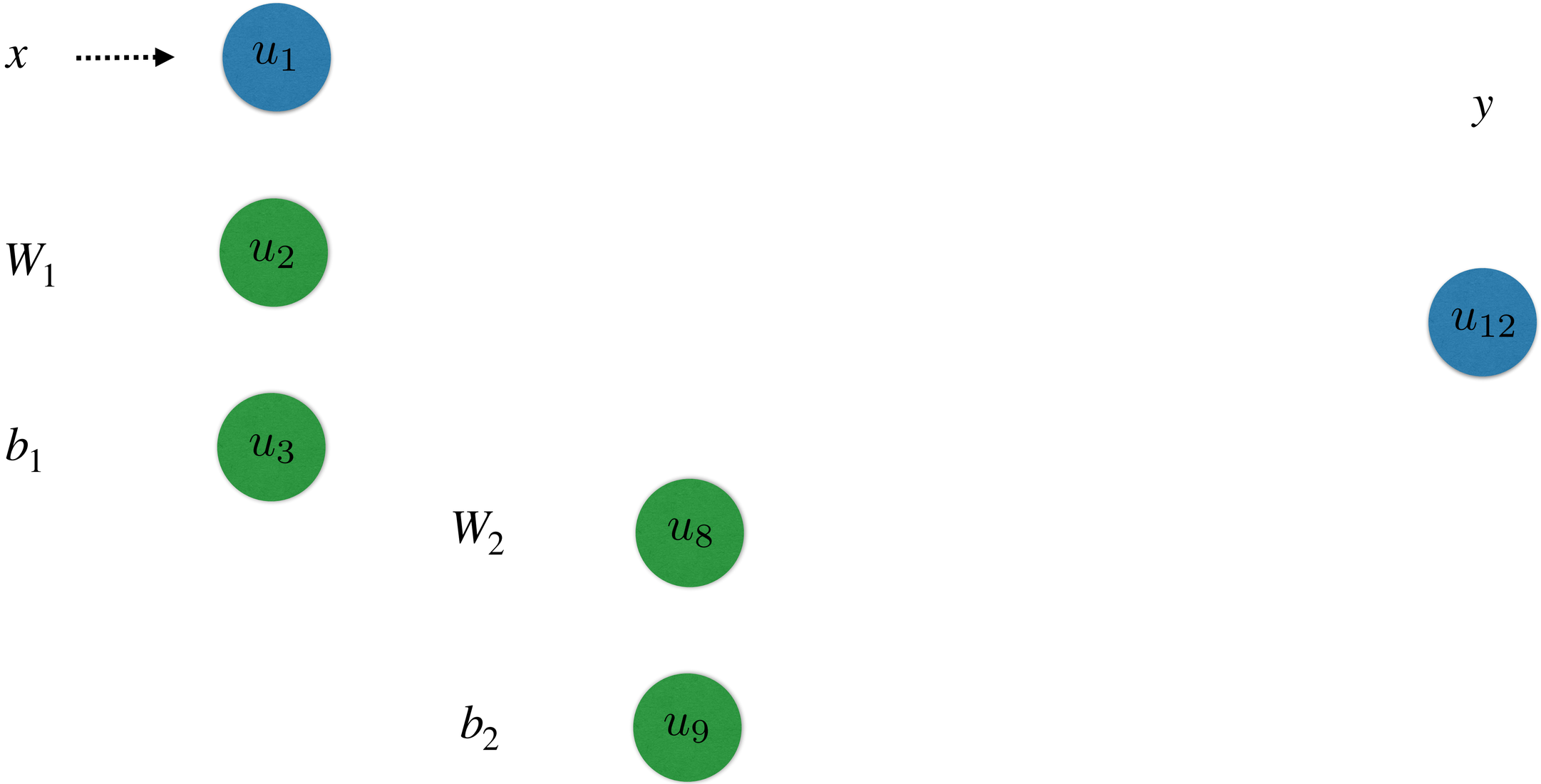
The Computational Graph



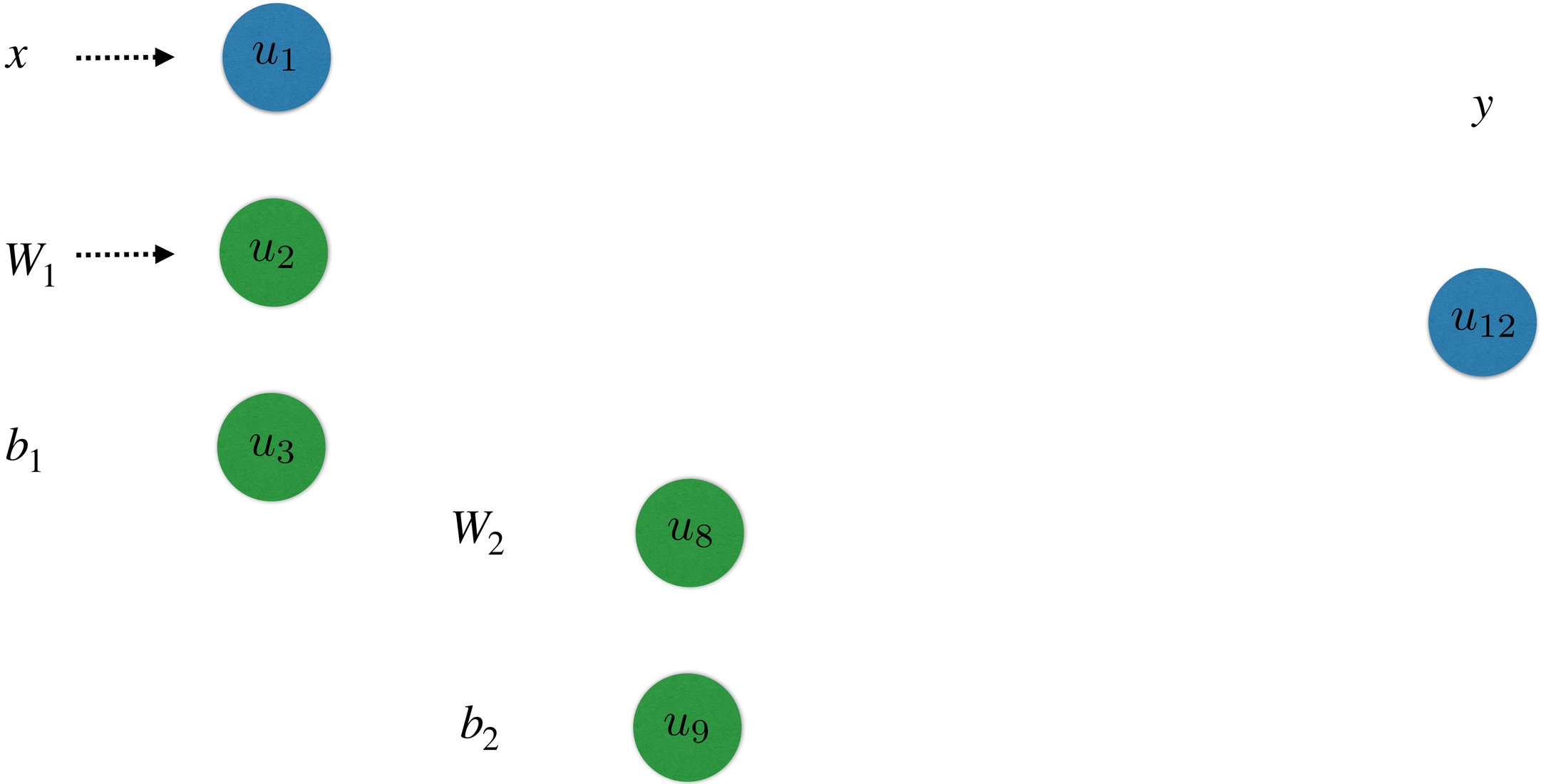
Forward Propagation



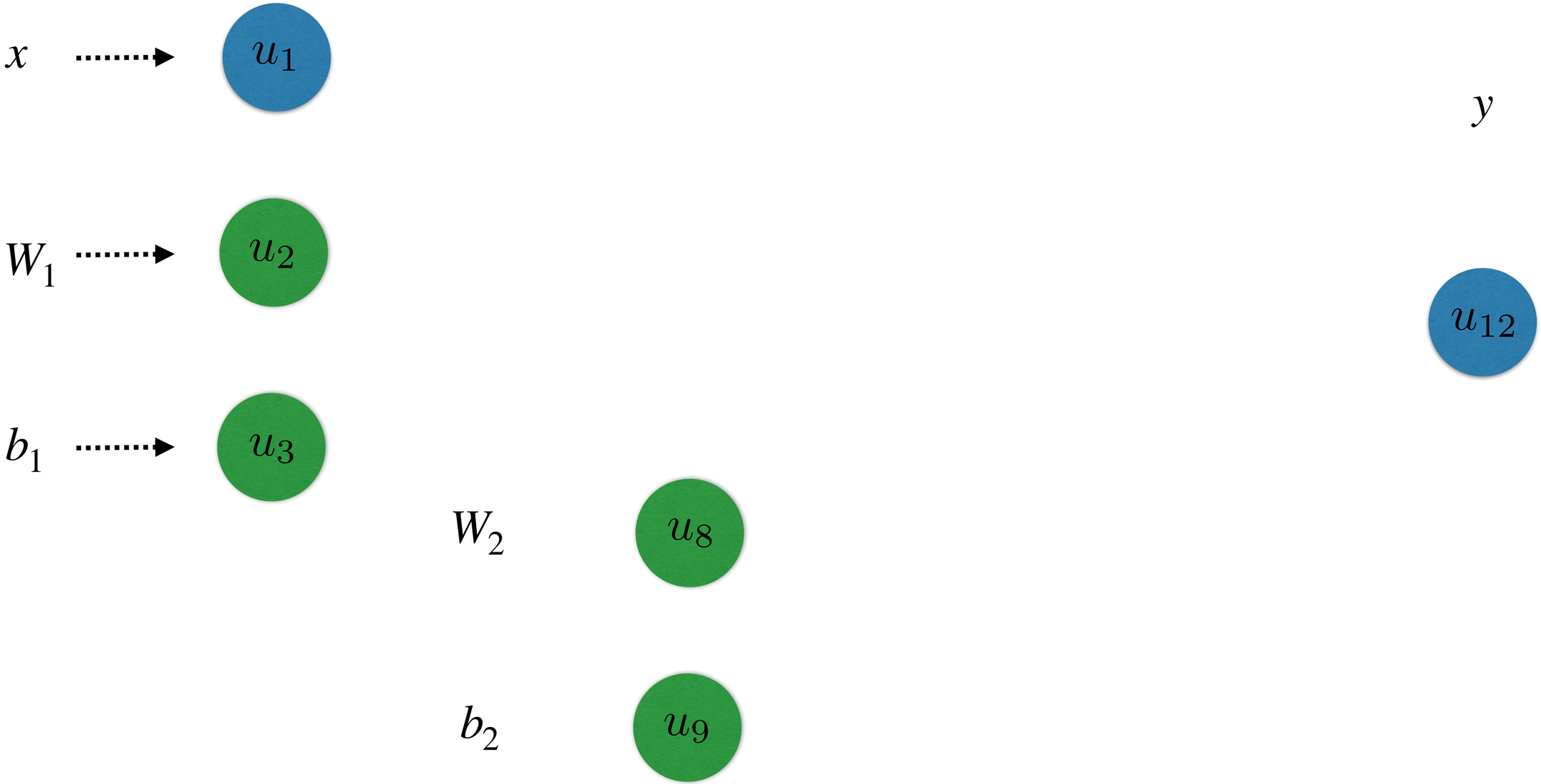
Forward Propagation



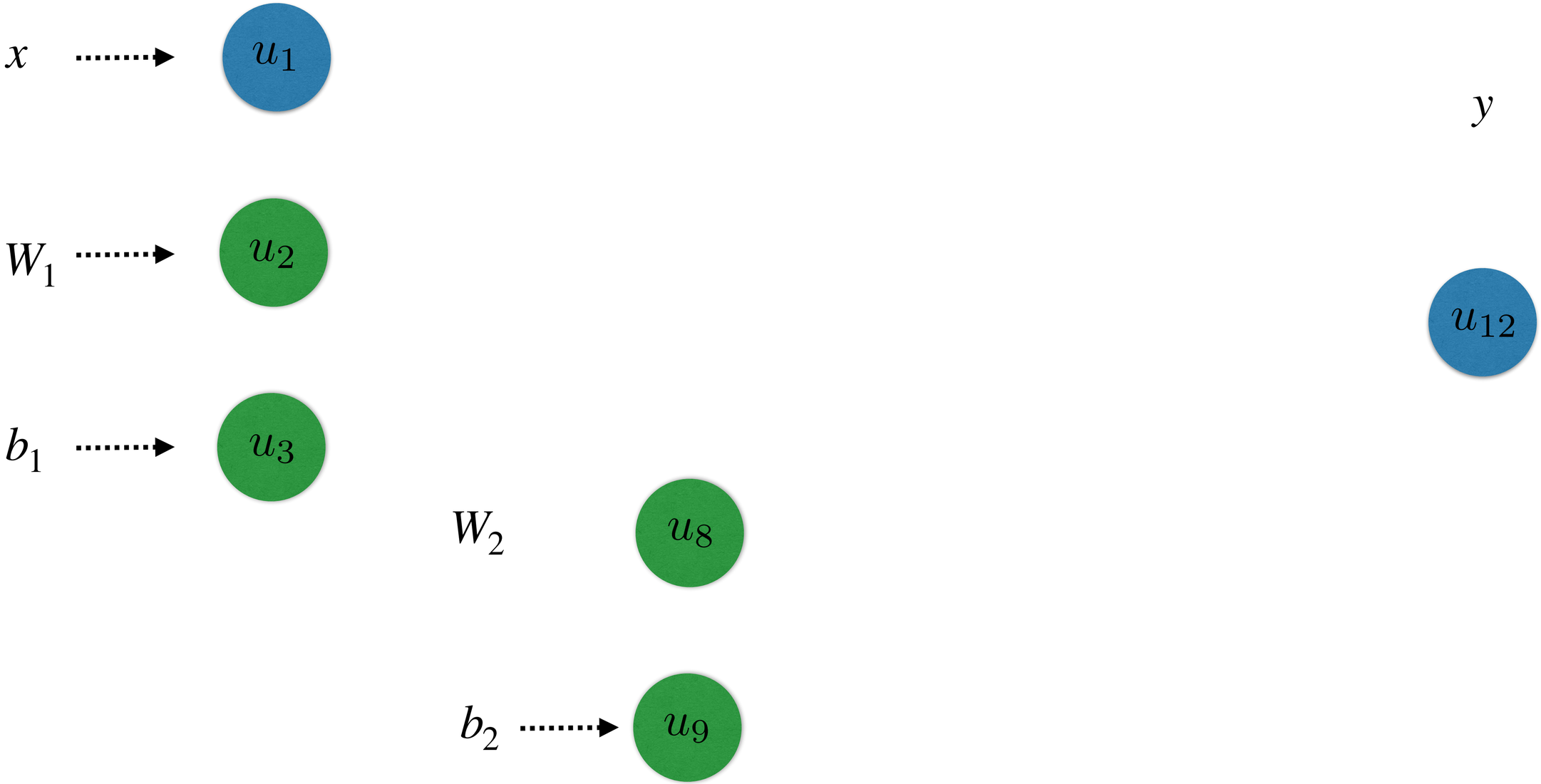
Forward Propagation



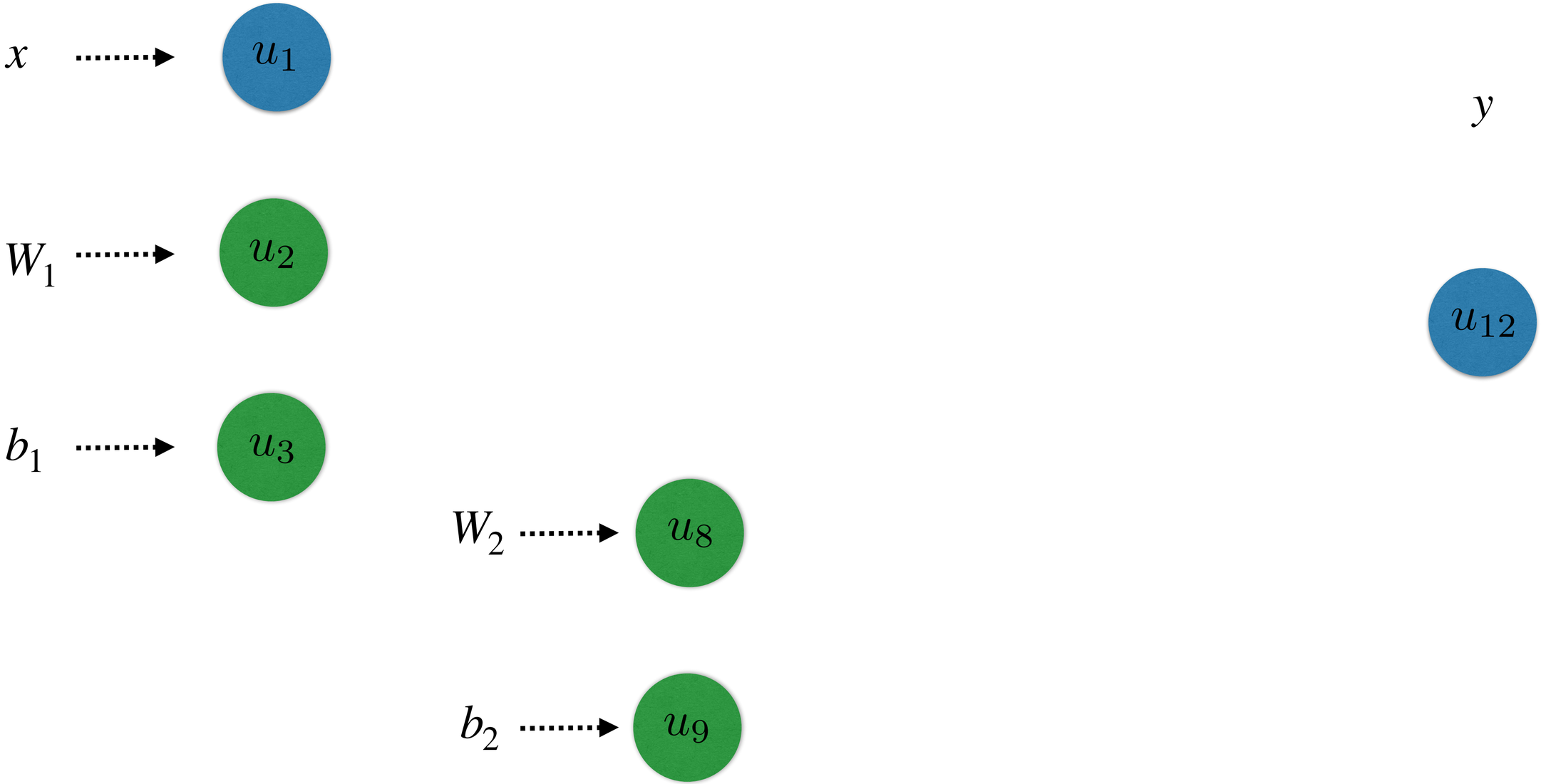
Forward Propagation



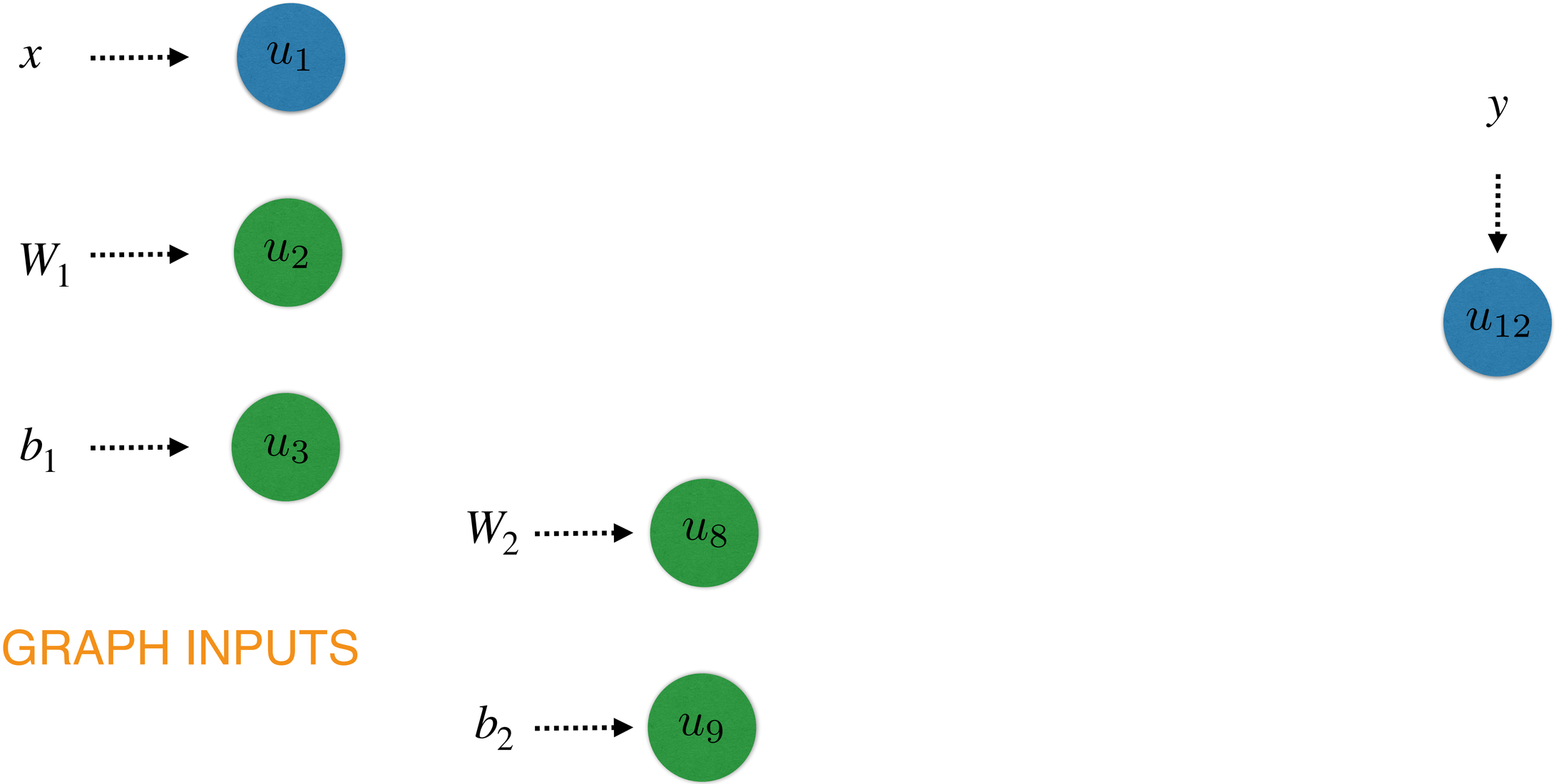
Forward Propagation



Forward Propagation

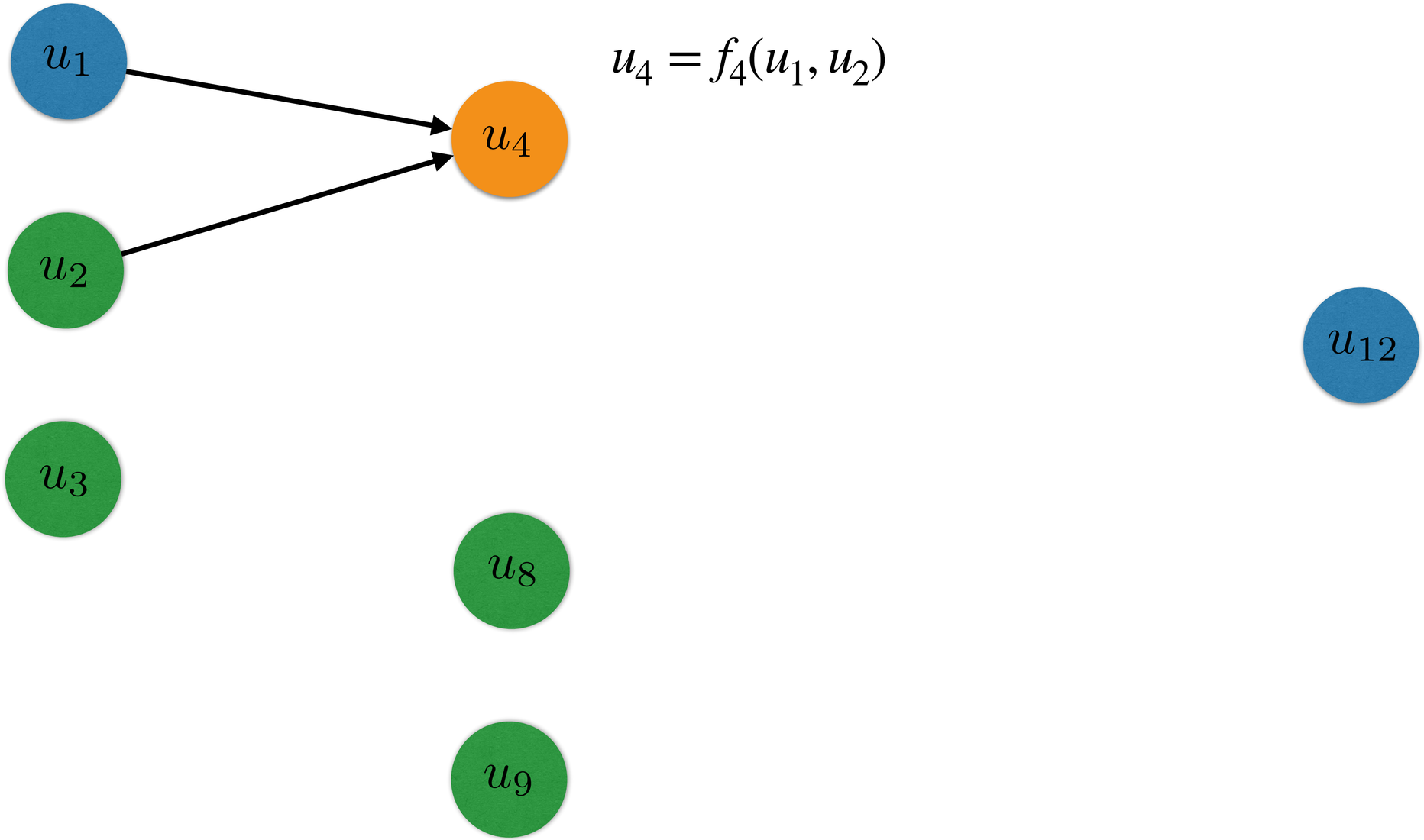


Forward Propagation



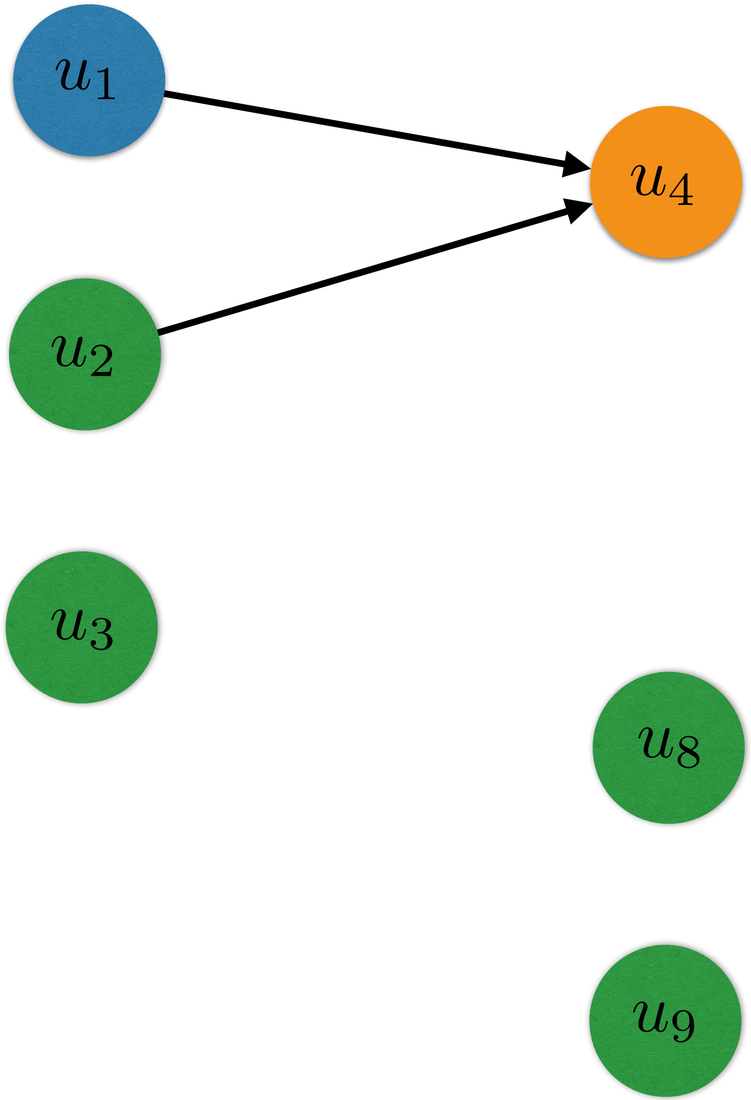
GRAPH INPUTS

Forward Propagation



$$u_4 = f_4(u_1, u_2)$$

Forward Propagation



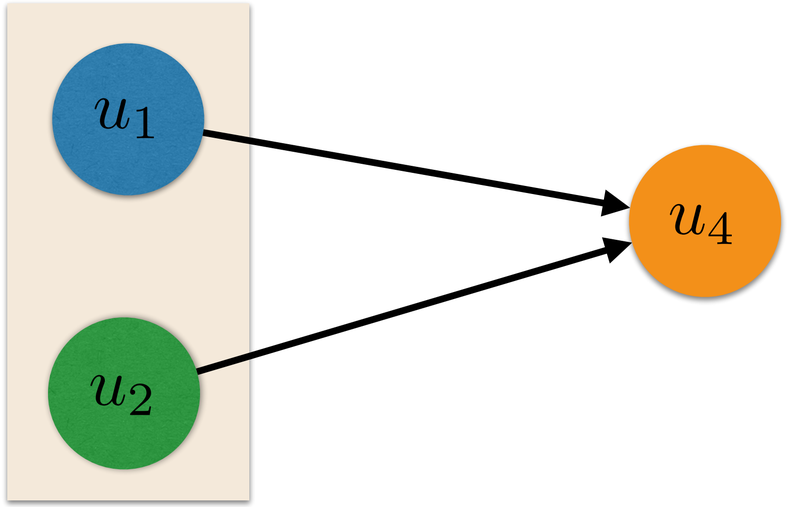
$$u_4 = f_4(u_1, u_2)$$

$$u_i = f_i\left(\mathbb{A}^i\right)$$

$$\mathbb{A}^i = \{u_j | j \in \text{Parents}(u_i)\}$$

parents of u_4

Forward Propagation



$$u_4 = f_4(u_1, u_2)$$

u_{12}

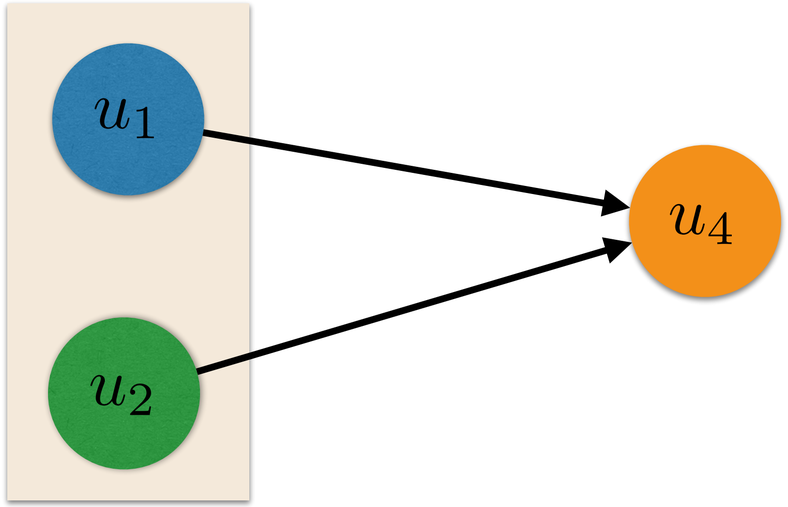
u_3

u_8

u_9

parents of u_4

Forward Propagation



$$u_4 = f_4(u_1, u_2)$$

$$u_i = f_i\left(\mathbb{A}^i\right)$$

$$\mathbb{A}^i = \{u_j | j \in \text{Parents}(u_i)\}$$

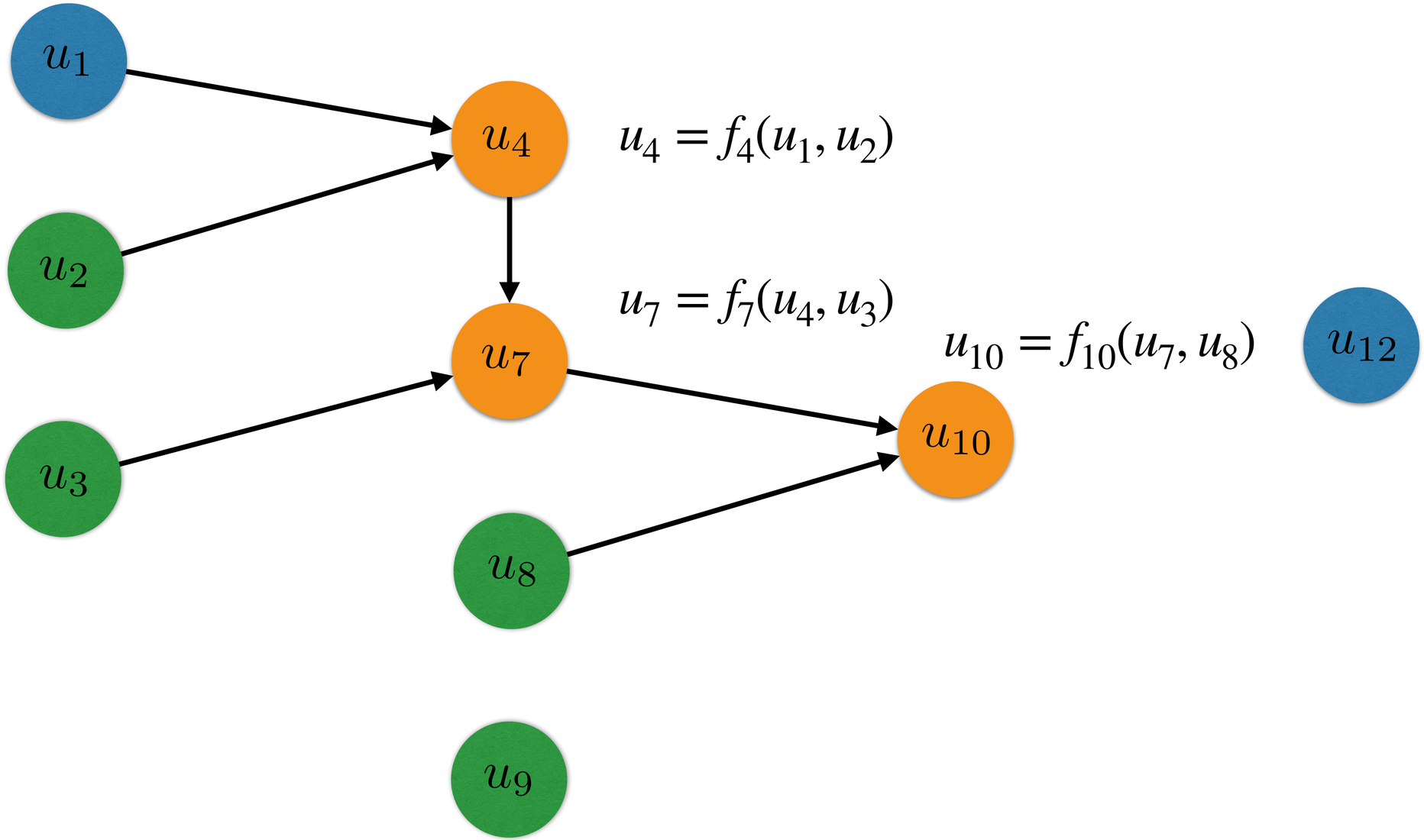
u_{12}

u_3

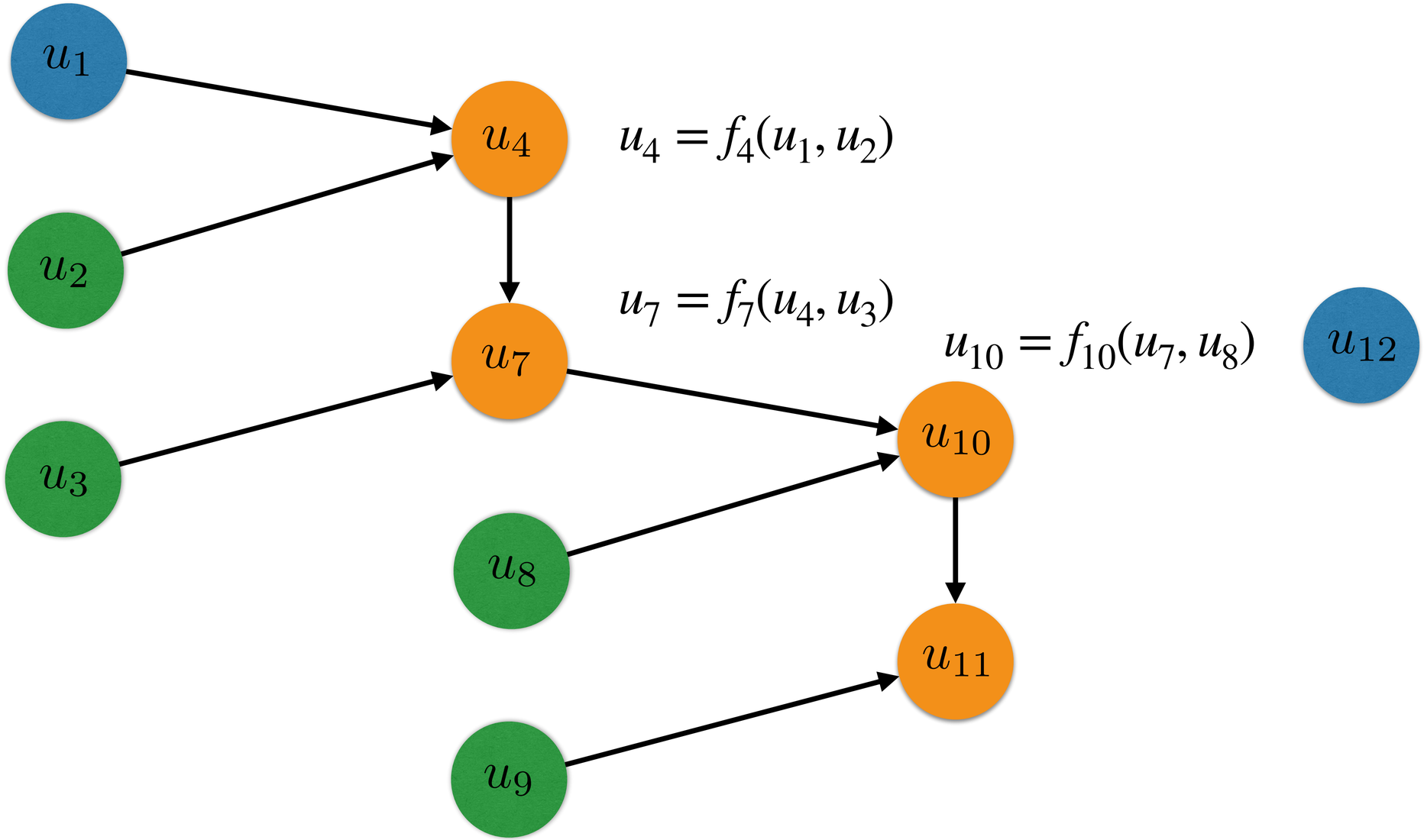
u_8

u_9

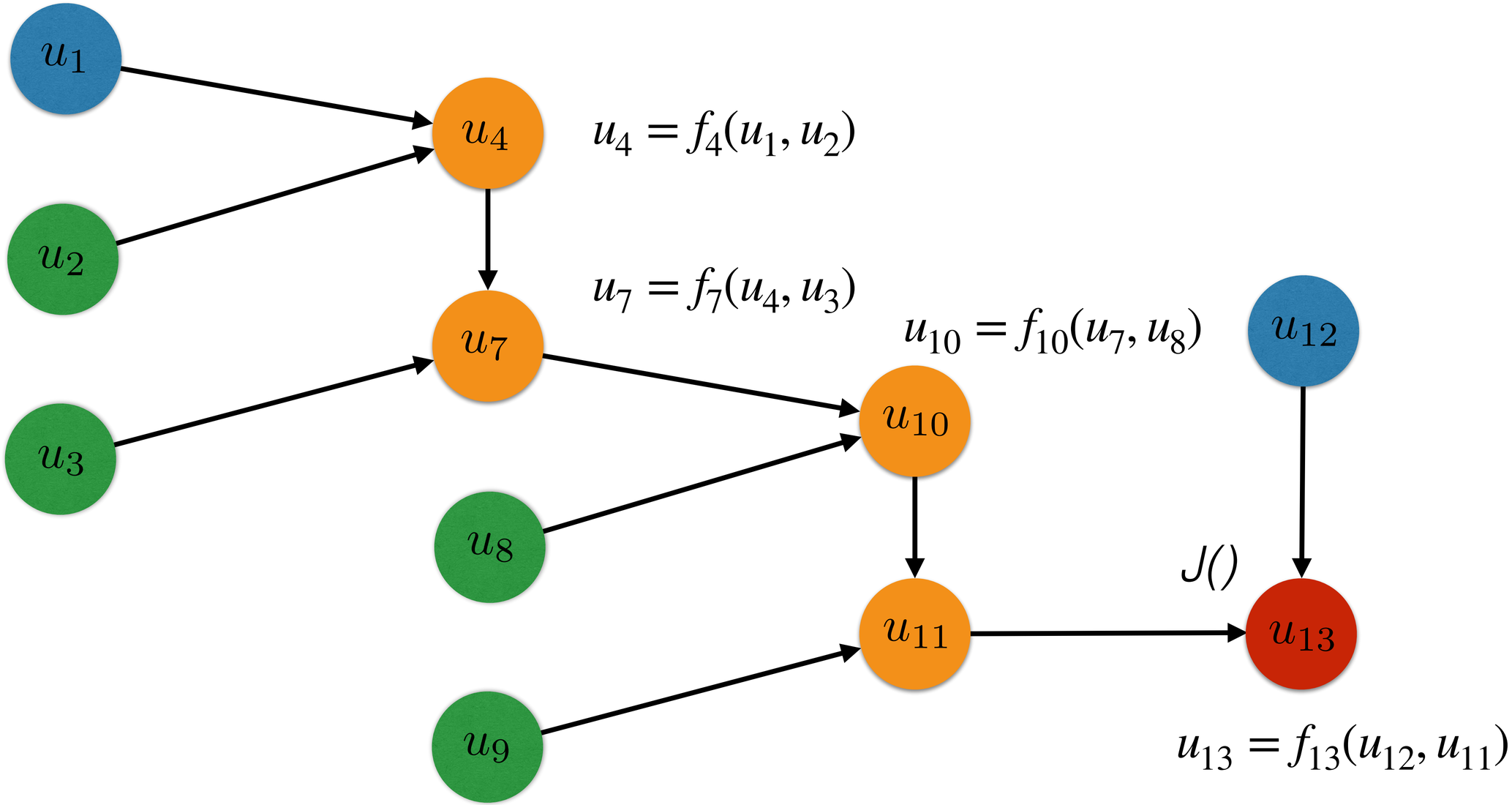
Forward Propagation



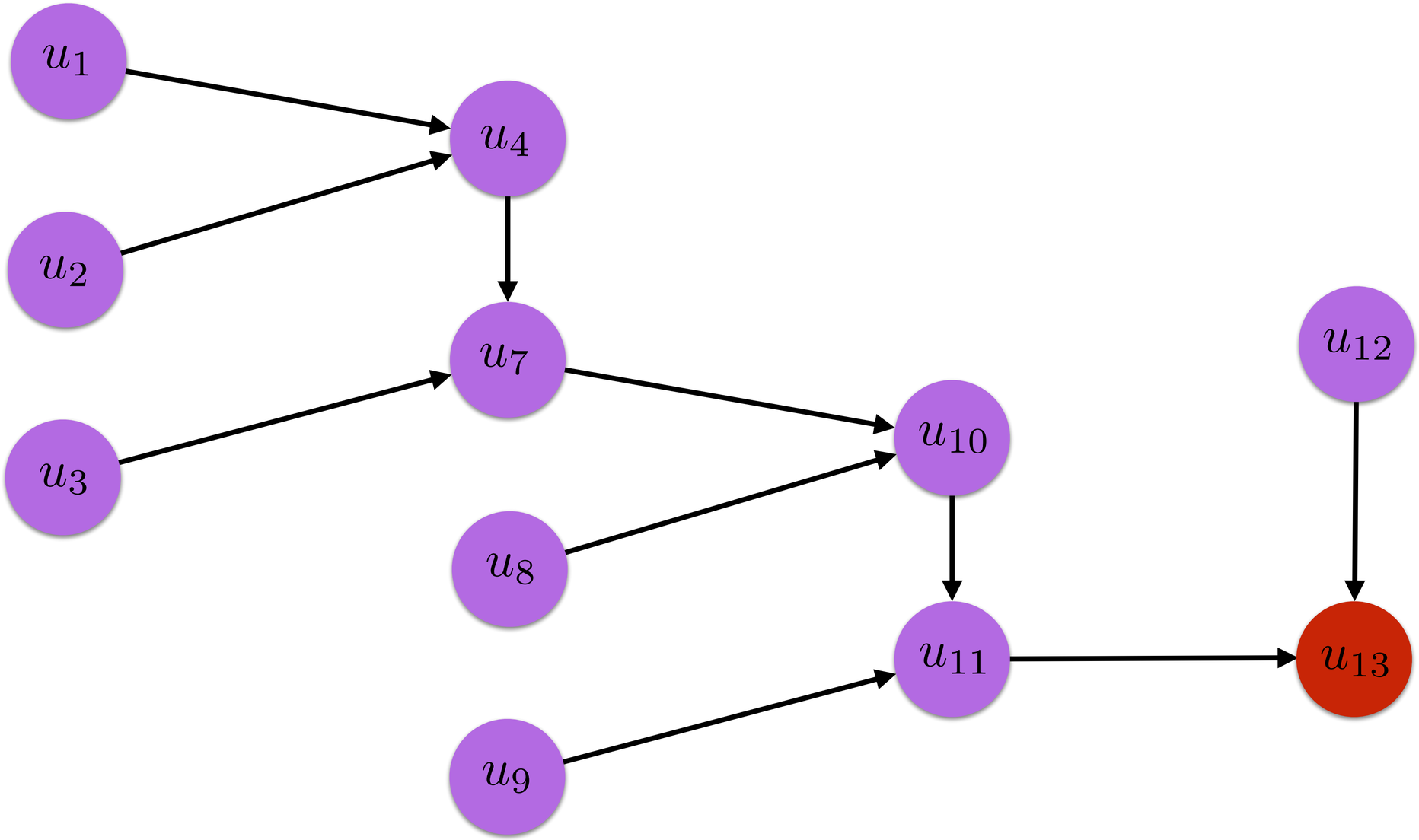
Forward Propagation



Forward Propagation



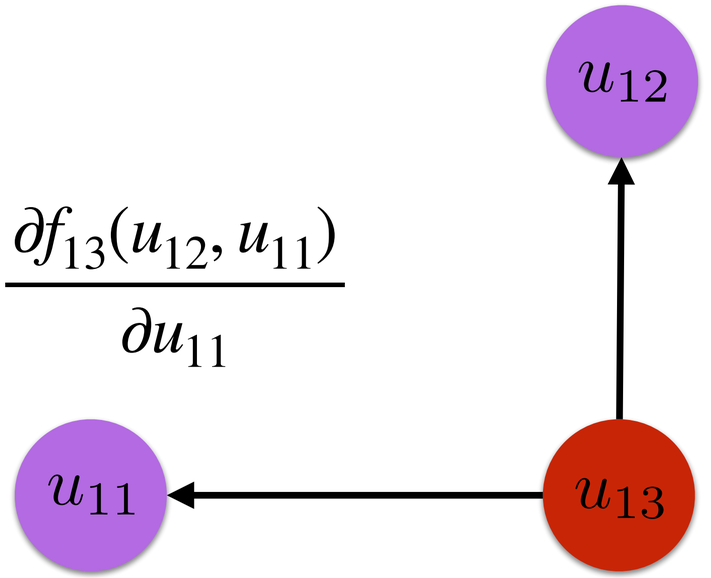
Backward Propagation



Backward Propagation

$$\frac{\partial u_{13}}{\partial u_{12}} = \frac{\partial f_{13}(u_{12}, u_{11})}{\partial u_{12}}$$

$$\frac{\partial u_{13}}{\partial u_{11}} = \frac{\partial f_{13}(u_{12}, u_{11})}{\partial u_{11}}$$

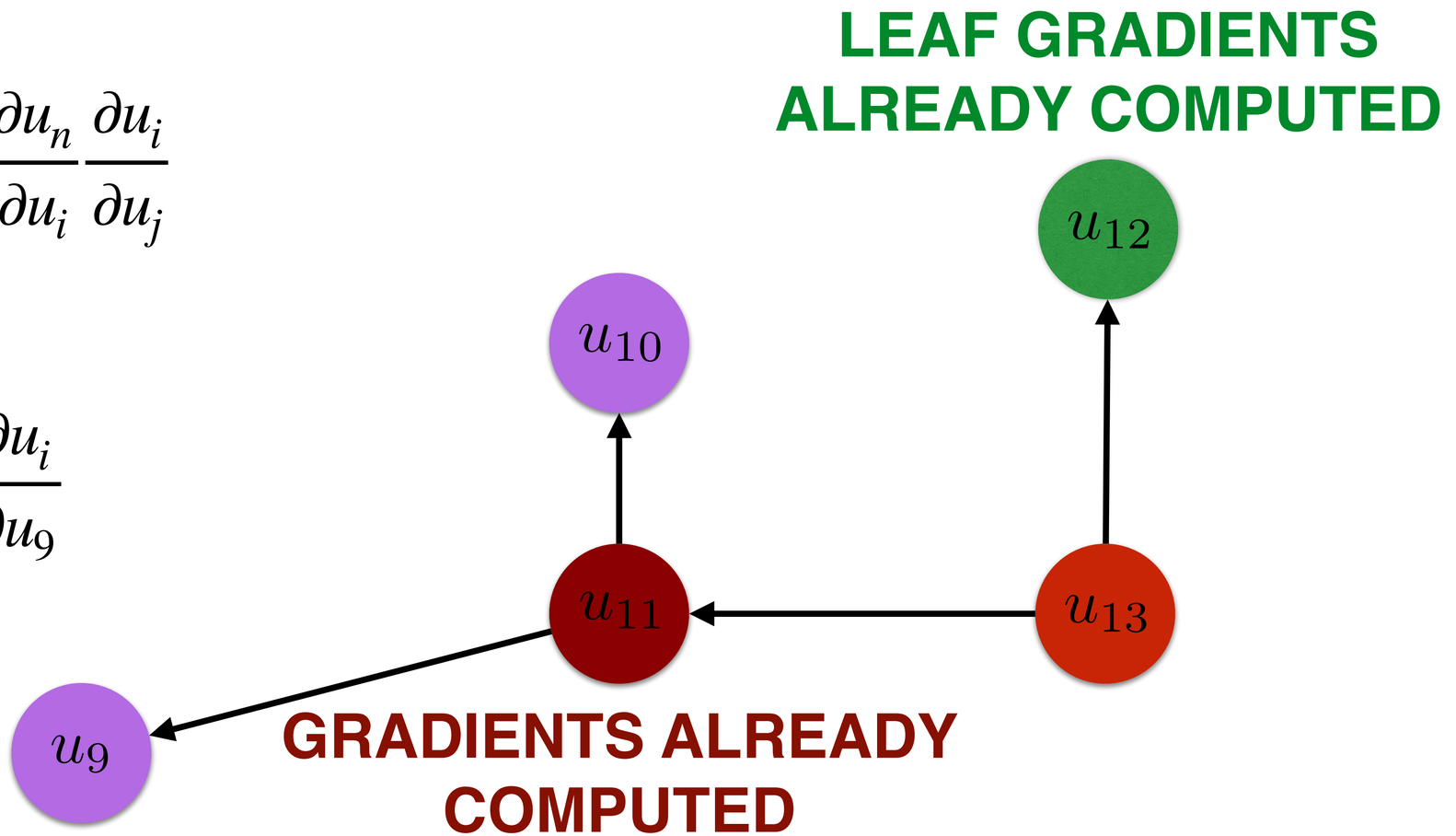


Backward Propagation

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathbf{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

$$\frac{\partial u_{13}}{\partial u_9} = \sum_{i:9 \in \mathbf{Pa}(u_i)} \frac{\partial u_{13}}{\partial u_i} \frac{\partial u_i}{\partial u_9}$$

$$= \frac{\partial u_{13}}{\partial u_{11}} \frac{\partial u_{11}}{\partial u_9}$$



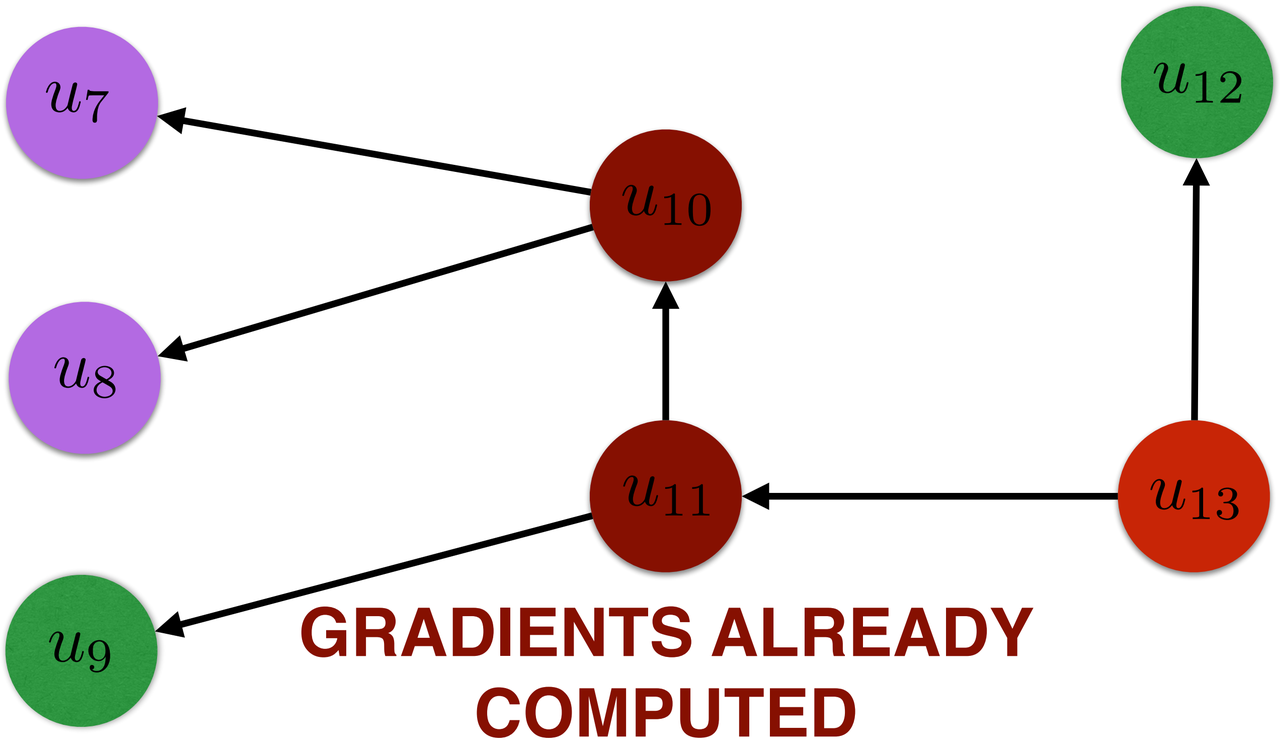
Backward Propagation

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathbf{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

$$\frac{\partial u_{13}}{\partial u_7} = \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial u_{10}}{\partial u_9}$$

$$\frac{\partial u_{13}}{\partial u_8} = \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial u_{10}}{\partial u_8}$$

LEAF GRADIENTS ALREADY COMPUTED



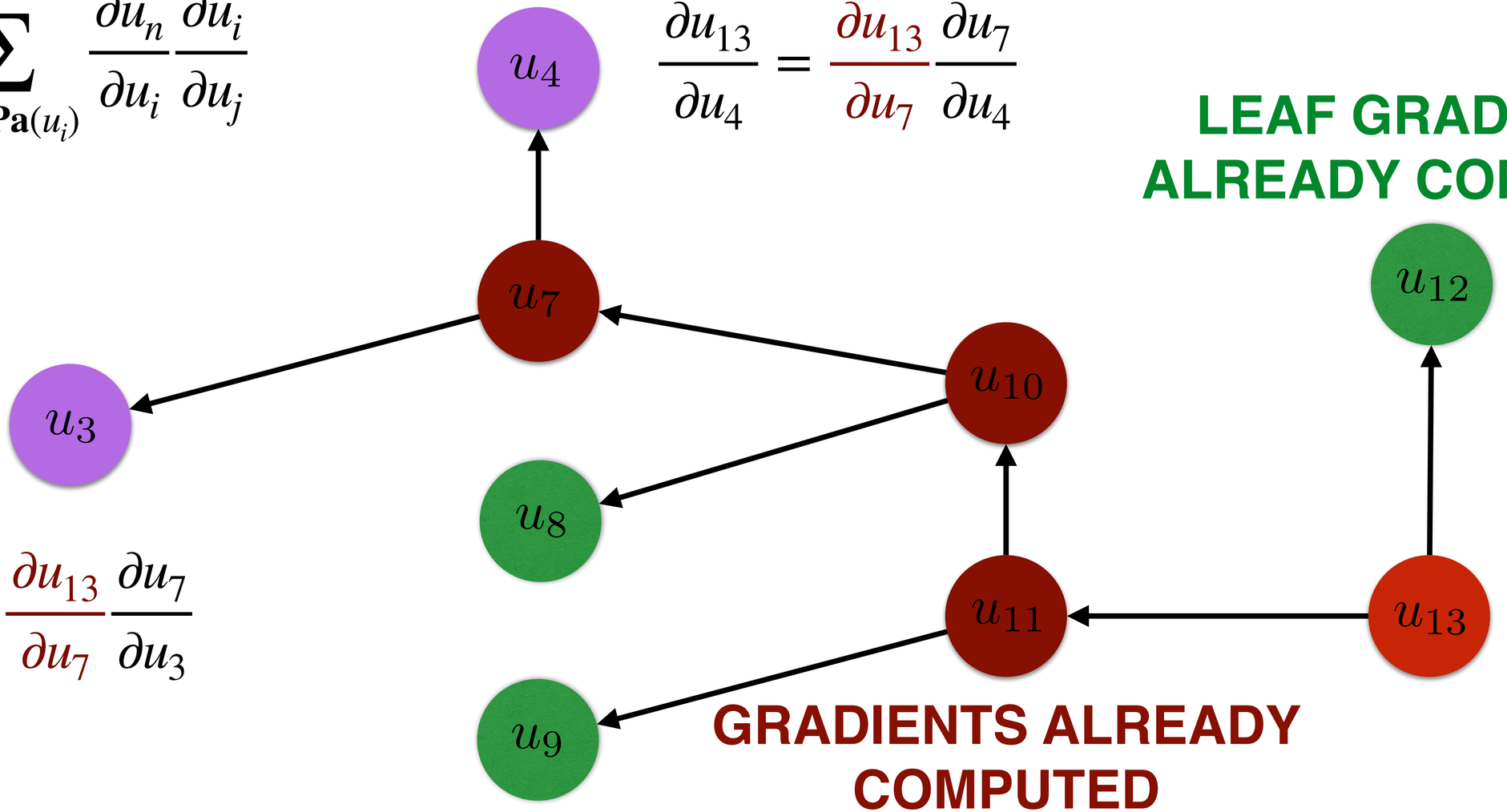
GRADIENTS ALREADY COMPUTED

Backward Propagation

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathbf{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

$$\frac{\partial u_{13}}{\partial u_4} = \frac{\partial u_{13}}{\partial u_7} \frac{\partial u_7}{\partial u_4}$$

$$\frac{\partial u_{13}}{\partial u_3} = \frac{\partial u_{13}}{\partial u_7} \frac{\partial u_7}{\partial u_3}$$



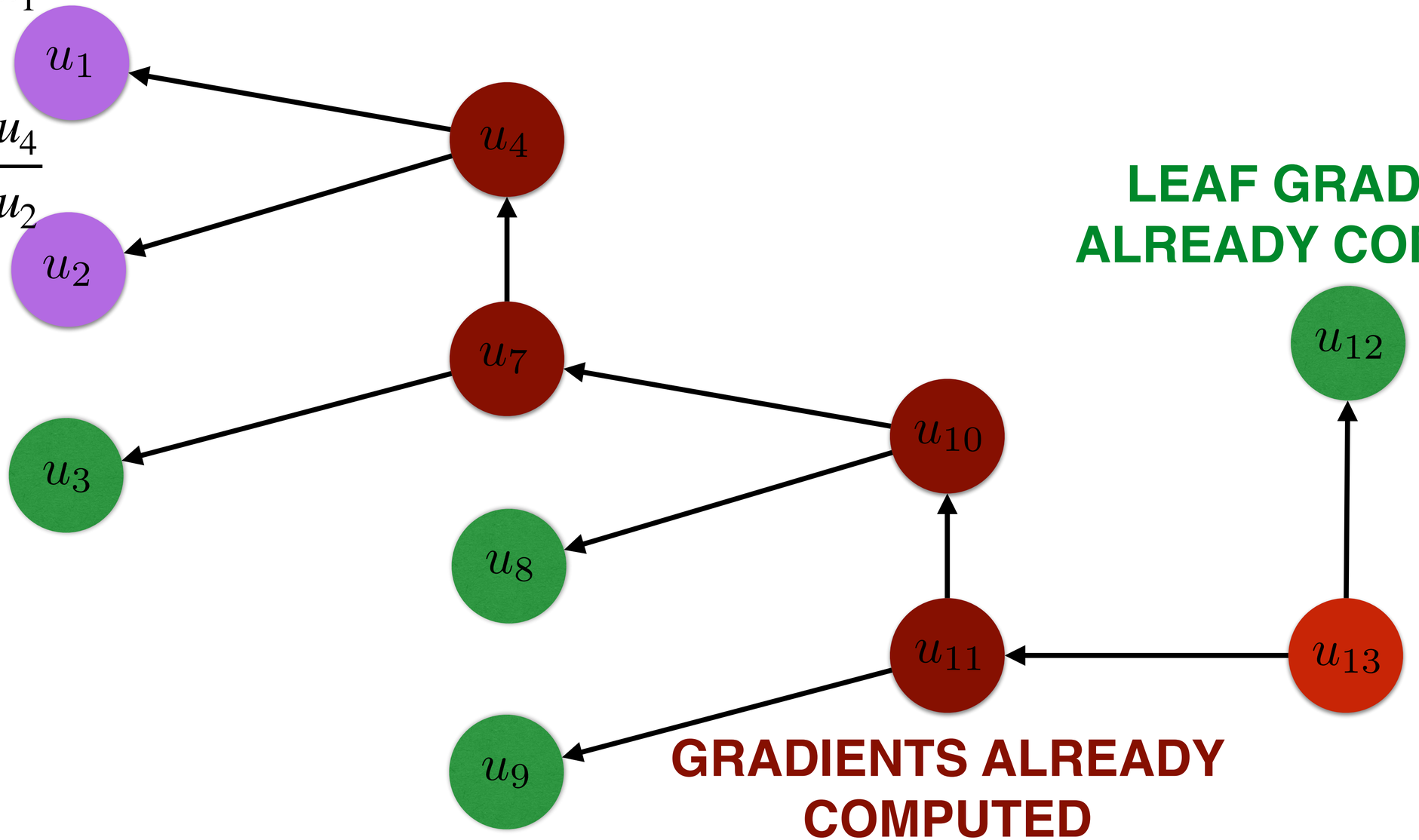
LEAF GRADIENTS ALREADY COMPUTED

GRADIENTS ALREADY COMPUTED

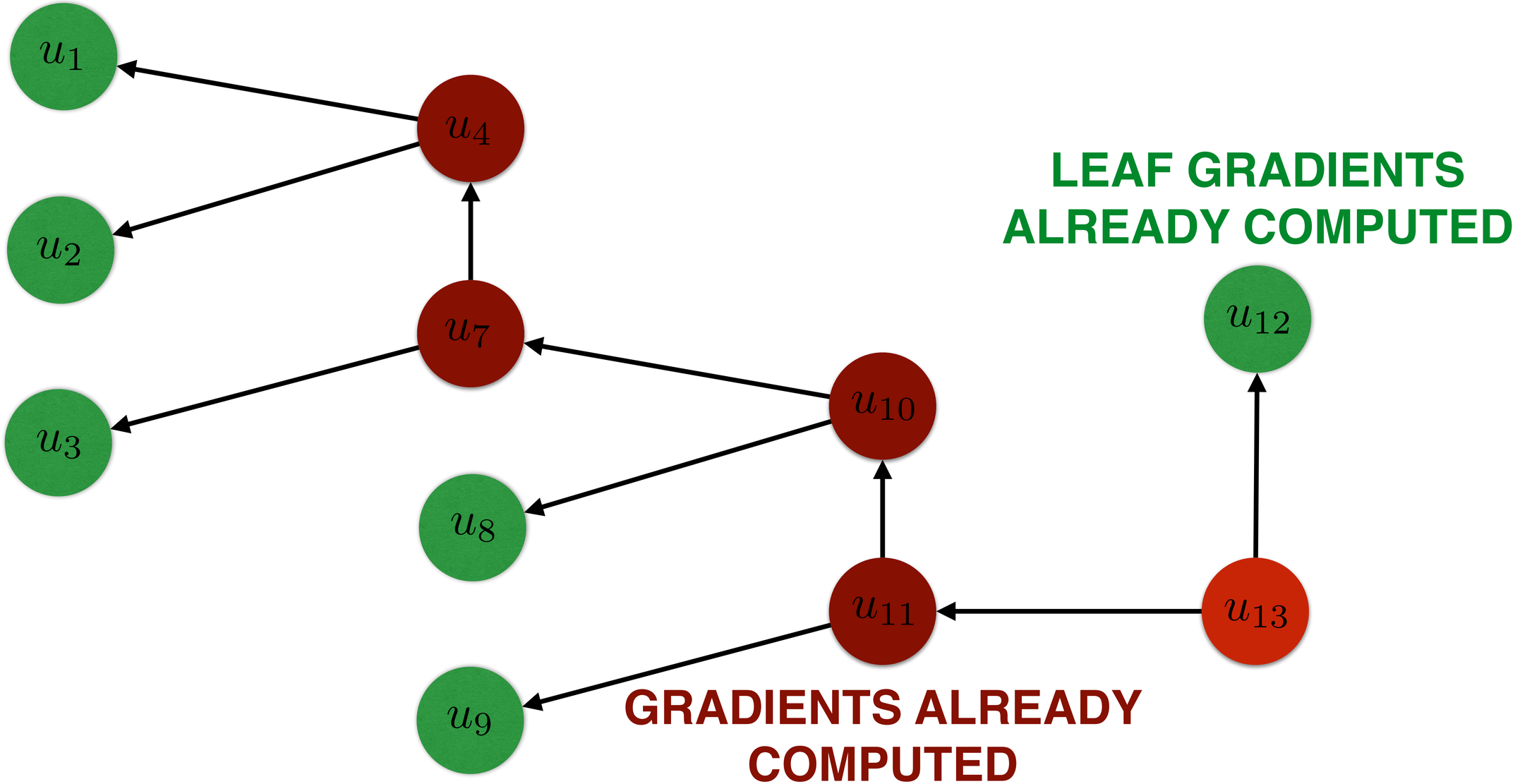
Backward Propagation

$$\frac{\partial u_{13}}{\partial u_1} = \frac{\partial u_{13}}{\partial u_4}\frac{\partial u_4}{\partial u_1}$$

$$\frac{\partial u_{13}}{\partial u_2} = \frac{\partial u_{13}}{\partial u_4}\frac{\partial u_4}{\partial u_2}$$



Backward Propagation



Back-Propagation

- The approach seen so far is called **symbol-to-value** differentiation
- This is used by libraries such as Caffe, PyTorch, TensorFlow (eager execution)

Back-Propagation

- The approach seen so far is called **symbol-to-value** differentiation
- This is used by libraries such as Caffe, PyTorch, TensorFlow (eager execution)

Back-Propagation

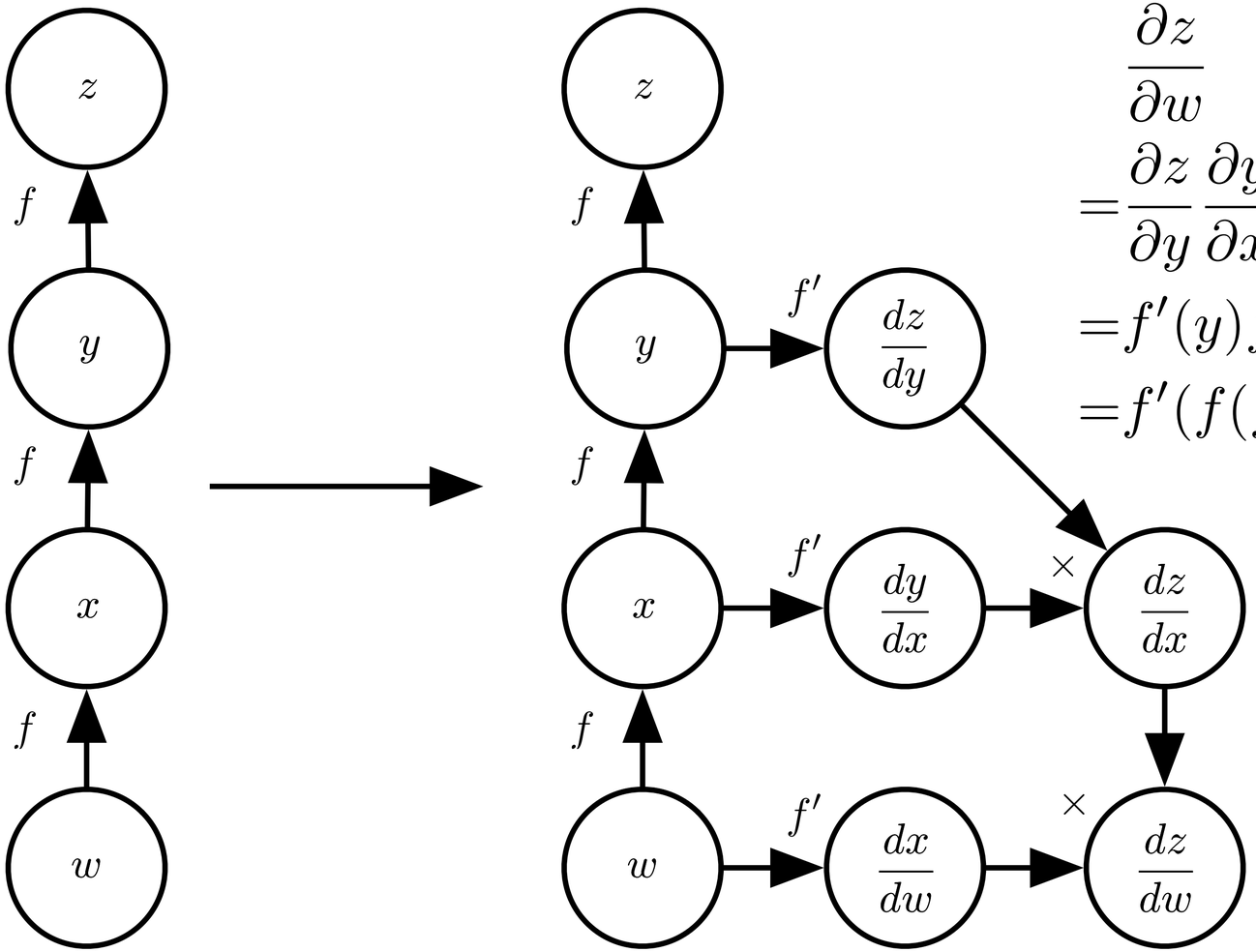
PyTorch implementation example

```
class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

Back-Propagation

- An alternative approach is the **symbol-to-symbol** differentiation
- This is used by libraries such as ~~Theano~~ and TensorFlow

Symbol-to-Symbol



$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

Symbol-to-Symbol

- Advantages
  - Derivatives are computed as a forward propagation in another graph
  - Higher order derivatives can be easily computed

Back-Propagation Forms

- The back-propagation algorithm exploits a special case of the chain rule, written in recursive form

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

- In alternative one could use the sequential form

$$\frac{\partial u_n}{\partial u_j} = \sum_{\substack{\mathrm{path}(u_{\pi_1}, \ldots, u_{\pi_t}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u_{\pi_k}}{\partial u_{\pi_{k-1}}}$$

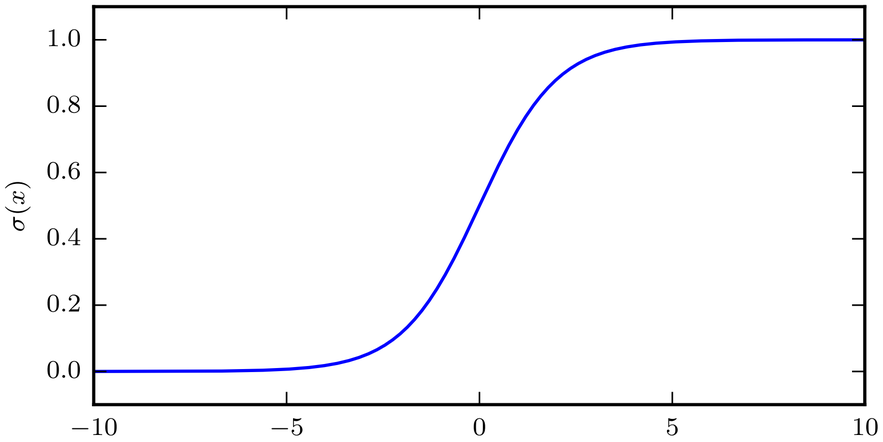
Further Issues

- Returning more that one output (1 tensor) might be more efficient
- Memory consumption
- Data types
- Undefined gradients (e.g. L1 norm)

Vanishing gradient

- Because of chain rule, we multiply a lot gradients $\frac{\partial u_{100}}{\partial u_1} = \frac{\partial u_{100}}{\partial u_{99}} \frac{\partial u_{99}}{\partial u_{98}} \dots$
- If their values are small $abs() << 1$, gradient for early layers approaches 0 - no feedback
- Problem with sigmoid $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$; $\sigma'(0) = \frac{1}{4}$
- Solutions: better activation (ReLU family), batch normalization, skip connections

Vanishing gra



- Because of chain rule, we multiply a lot gradients $\frac{\partial u_{100}}{\partial u_1} = \frac{\partial u_{100}}{\partial u_{99}} \frac{\partial u_{99}}{\partial u_{98}} \dots$
- If their values are small $abs() << 1$, gradient for early layers approaches 0 - no feedback
- Problem with sigmoid $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$; $\sigma'(0) = \frac{1}{4}$
- Solutions: better activation (ReLU family), batch normalization, skip connections

Extensions

- Automatic simplification of the derivatives (or the computational graph) — Theano, TensorFlow
- Reverse mode accumulation (what we have seen so far; efficient with a single output)
- Forward mode accumulation (efficient when outputs are more than the inputs)

Forward vs Reverse Mode

- The computation of the gradients involves the products (and sums) of Jacobians
- The order of these products determines the forward (left to right) or reverse (right to left) mode
- Suppose that there is one output $D \in \mathbb{R}^{m \times 1}$

$$ABCD \longleftarrow$$

reverse more efficient

Forward vs Reverse Mode

- The computation of the gradients involves the products (and sums) of Jacobians
- The order of these products determines the forward (left to right) or reverse (right to left) mode
- With multiple outputs $D \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{p \times q}$ and $p < n$

$$ABCD \longrightarrow$$

forward more efficient