CS6421: Deep Neural Networks

Gregory Provan

Spring 2020

Lecture 5: TensorFlow and Computation Graphs

Based on notes from Hung-Yi Lee,

Andrej Karpathy, Fei-Fei Li, Justin Johnson

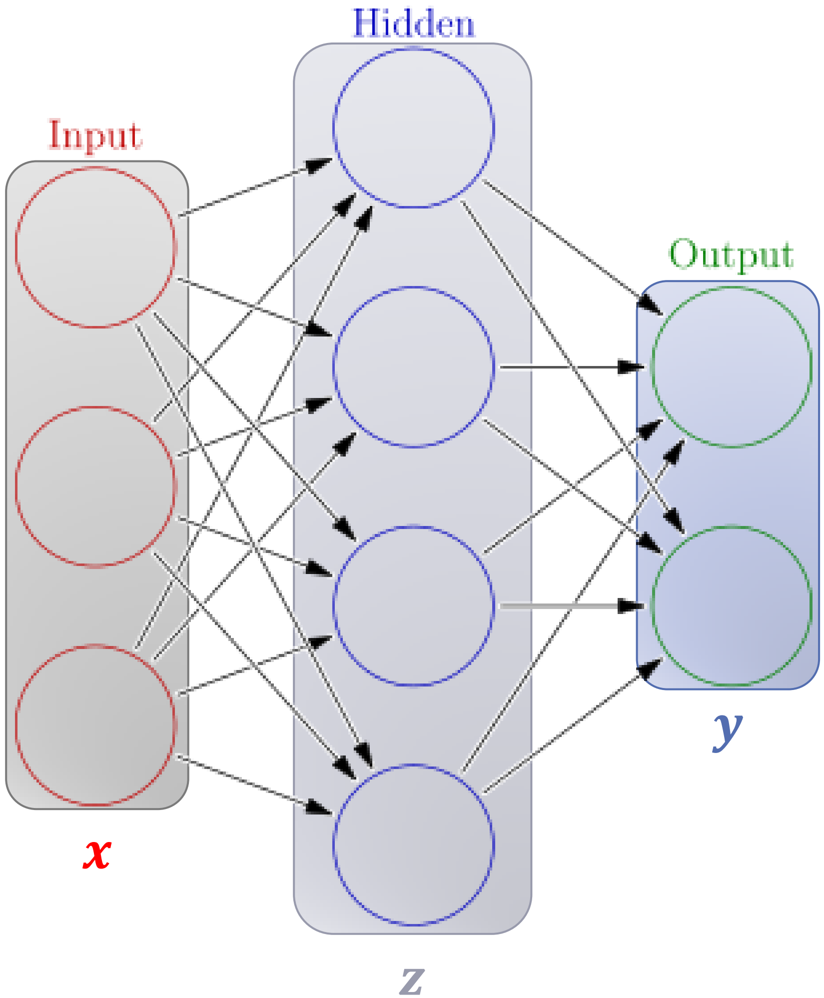
Motivation

- Deep network
  - an inter-connected set of modules
- Network inference uses this modular decomposition
  - All DL platforms: TensorFlow, Theano, etc.
- Computation graph
  - Mathematical foundations for inference

Overview

- Modularity and Computation Graphs
- Inference in Computation Graphs
- Backpropagation
  - Computation Graphs viewpoint
- DL Platforms
  - TensorFlow, Theano, etc.

Neural Network: Definition



Weights

$$z = f(W_1 x + b_1)$$

$$y = g(W_2 z + b_2)$$

Activation functions

4 + 2 = 6 neurons (not counting inputs)
[3 x 4] + [4 x 2] = 20 weights
4 + 2 = 6 biases

26 learnable **parameters**

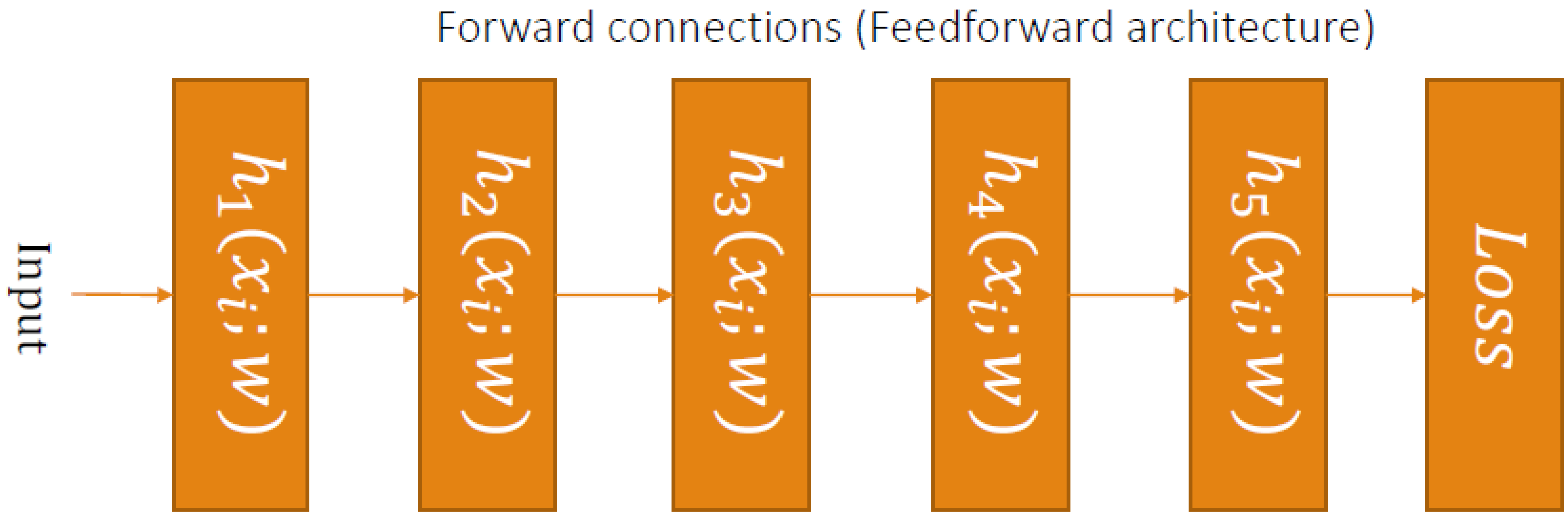
Demo

Neural Network: Definition

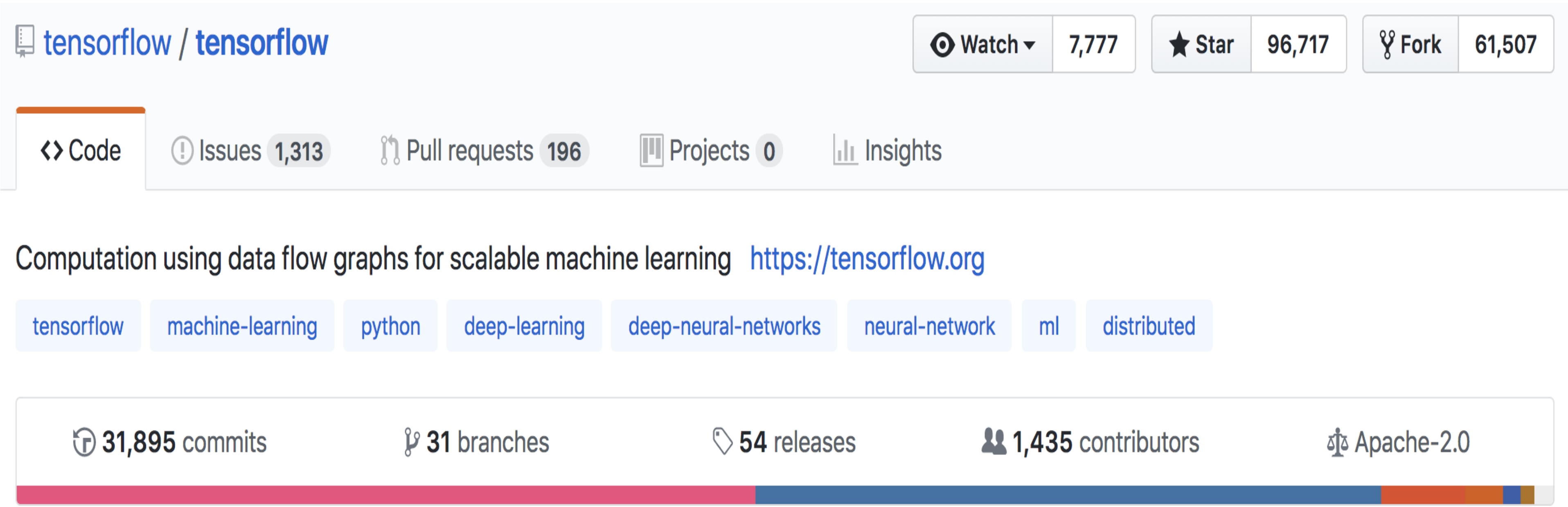
- A family of parametric, non-linear and hierarchical representation learning functions
  - massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a^L(x; w^1, \ldots, w^L) = h^L(h^{L-1} \ldots h^1(x, w^1), w^{L-1}), w^L)$
  - x:input,
  - w^l: parameters for layer l,
  - $a^l = h^l(x, w^l)$: (non-) linear function
- Given training corpus $\{X, Y\}$ find optimal parameters
  - $w^* \leftarrow \text{argmin}_w \sum_{(x,y) \subseteq (X,Y)} L(y, aL(x))$

Architectural View of Deep Networks

- A neural network model is a series of hierarchically connected functions
- These hierarchies can be very complex



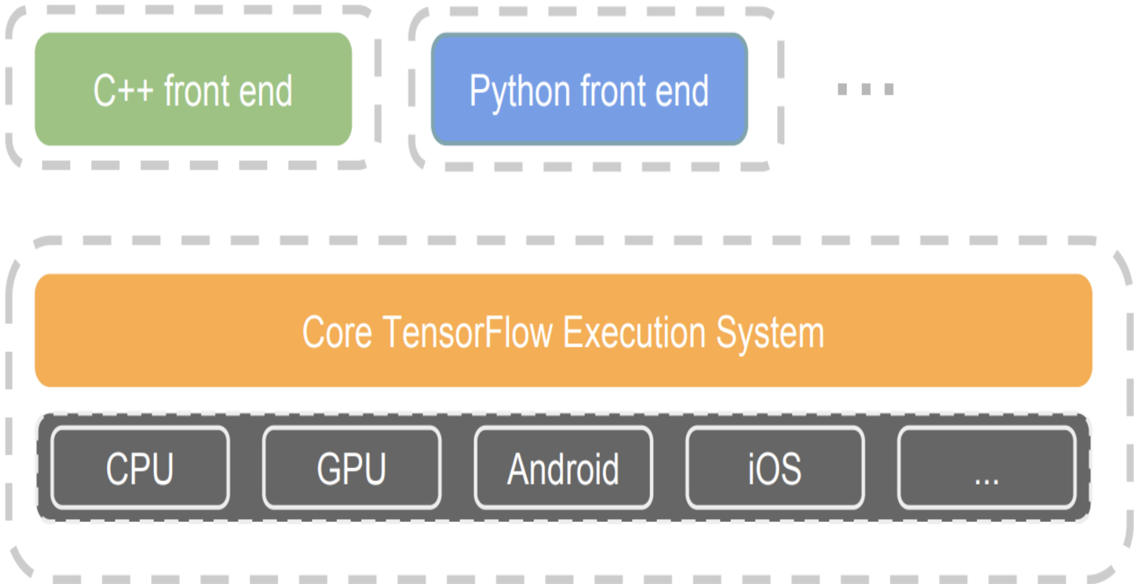
What is TensorFlow?



- Open source library for numerical computation using **computation graphs**
- Developed by Google Brain Team to conduct machine learning research
  - Based on DisBelief used internally at Google since 2011
- “TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms”

TensorFlow Architecture

- Core in C++
  - Very low overhead
- Different front ends for specifying/driving the computation
  - Python and C++ today, easy to add more



From: http://www.wsdm-conference.org/2016/slides/WSDM2016-Jeff-Dean.pdf

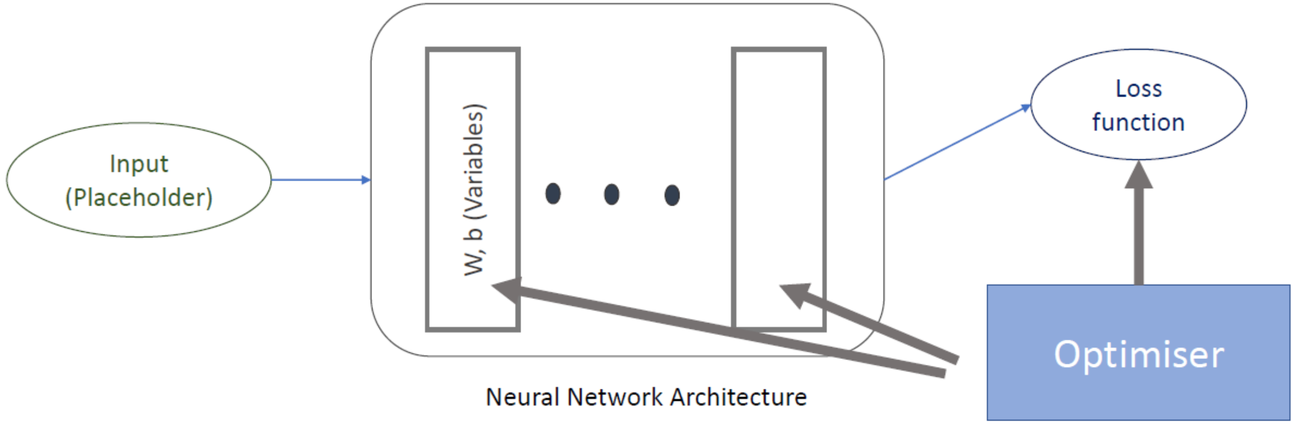
TF Learn

- TensorFlow with an SkLearn API
  - model.fit()
  - model.predict()
  - model.evaluate()
  - model.save() and model.load()
- GPU accelerated deep learning in 8 lines of Python

```
import tflearn

tflearn.init_graph()

net = tflearn.input_data(shape=[None, 3])
net = tflearn.fully_connected(net, 4,
                              activation='relu')
net = tflearn.fully_connected(net, 2,
                              activation='relu')
net = tflearn.regression(net, optimizer='adam')

model = tflearn.DNN(net)
model.fit(X, Y)
```

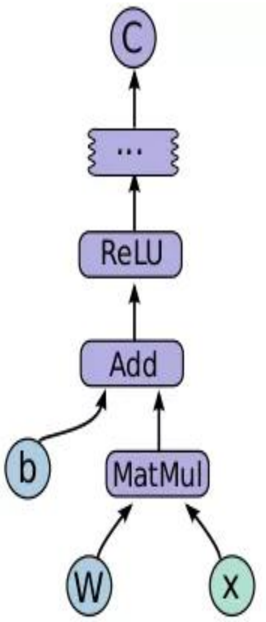
How Does TensorFlow Work?

- TensorFlow input elements
  - Input data (placeholder)
  - Deep-network architecture
  - Loss function
- Applies optimising compiler
  - Uses structure of network
  - Distributes inference to available processors (CPU, GPU, etc.)
  - Generates *computation graph* for optimising inference



TensorFlow

- Large-Scale Machine Learning Across Heterogeneous Distributed Systems
- Normally, we run programs (largely) sequentially
- TensorFlow:
  - builds a *computation graph*
  - assigns operations to optimal hardware
  - **then** runs the computations



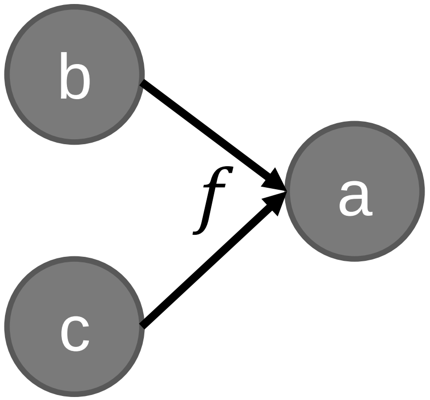

```
import tensorflow
import numpy

C = tf.relu(tf.add(tf.matmul(W, X), b))
```

What is TensorFlow

- **Key idea**: express a numeric computation as a **graph**
- Graph nodes are **operations** with any number of inputs and outputs
- Graph edges are **tensors** which flow between nodes

Computation Graph

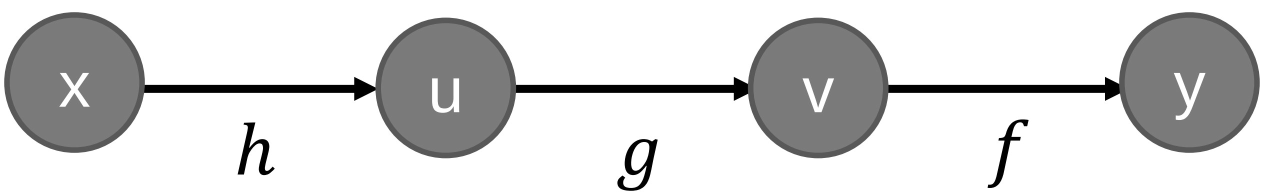
- A "language" describing a function
  - ***Node***: variable (scalar, vector, tensor ……)
  - ***Edge***: operation (simple function)



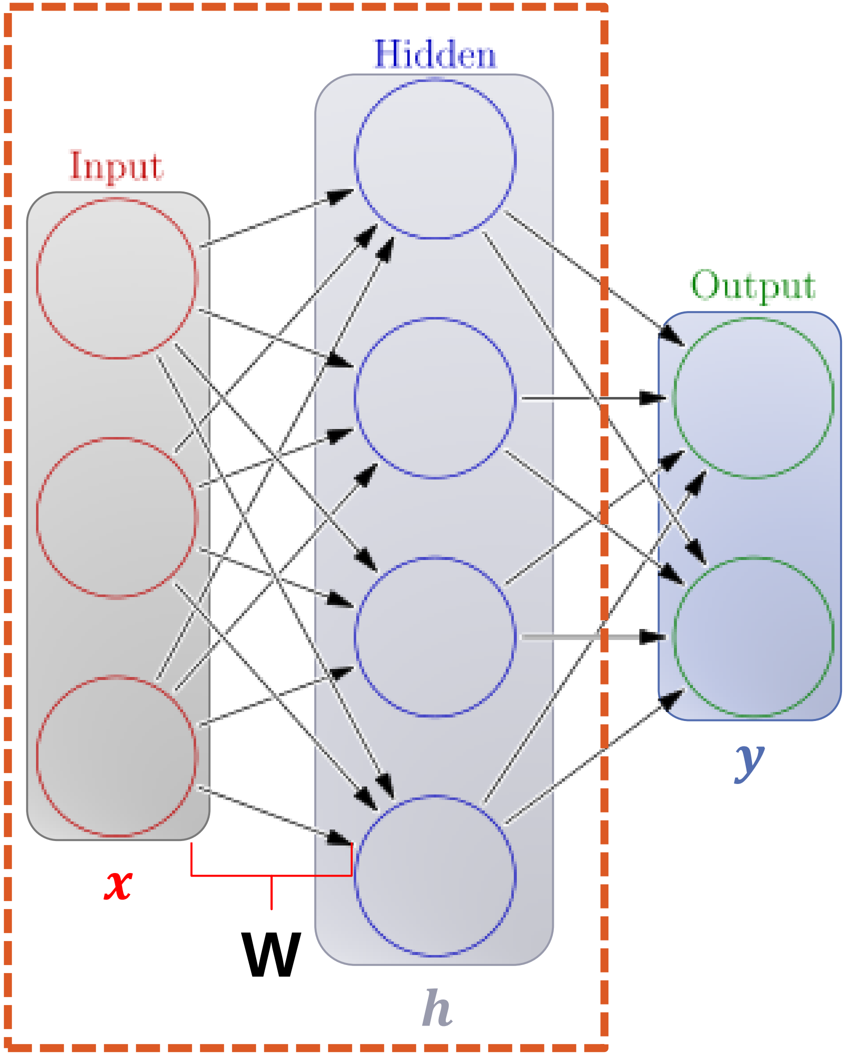
$a = f(b,c)$

Example $$y = f\left(g\left(h(x)\right)\right)$$

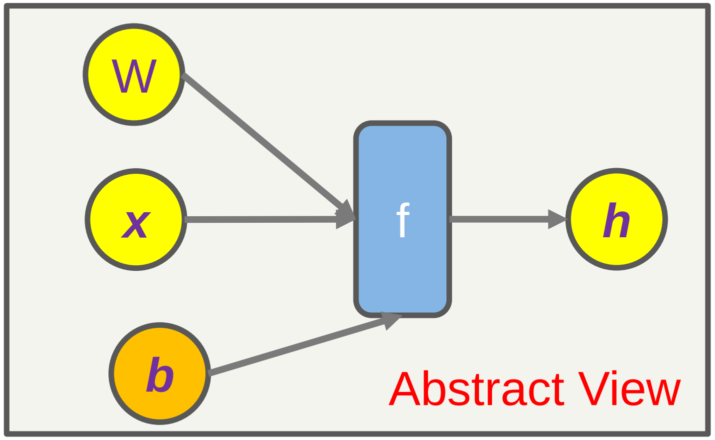
$$u = h(x) \qquad v = g(u) \qquad y = f(v)$$

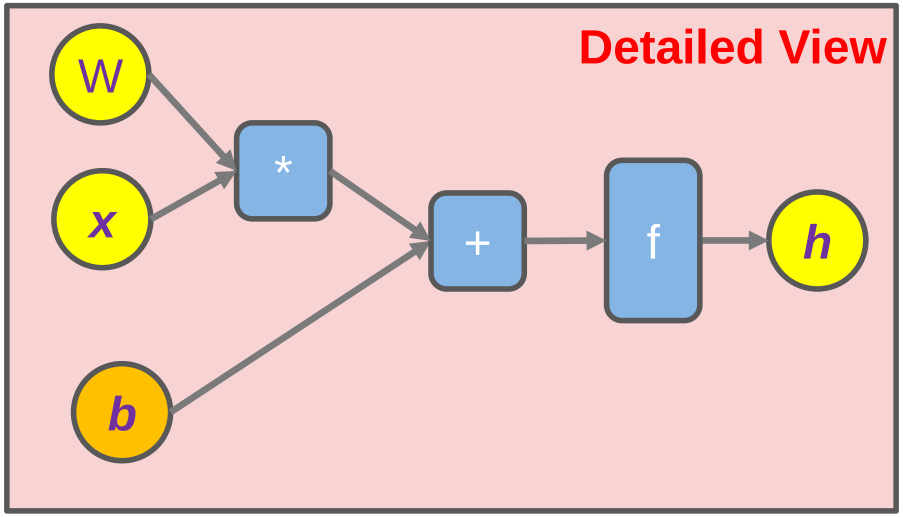


Modular Operations: Computation Graph Definition



$$h = f(Wx + b)$$





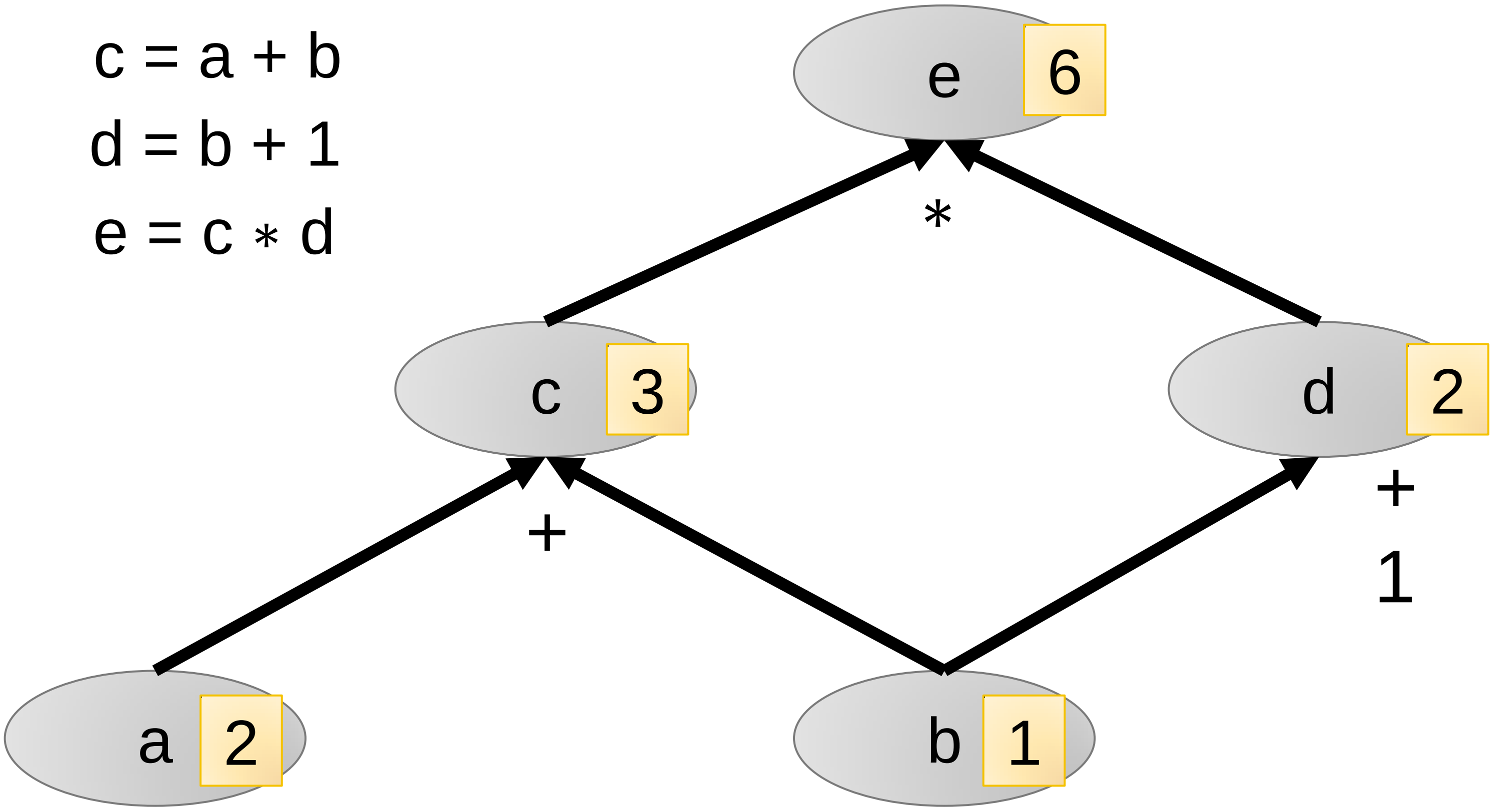
Computational Graph

- Example: $e = (a+b) * (b+1)$

$c = a + b$

$d = b + 1$

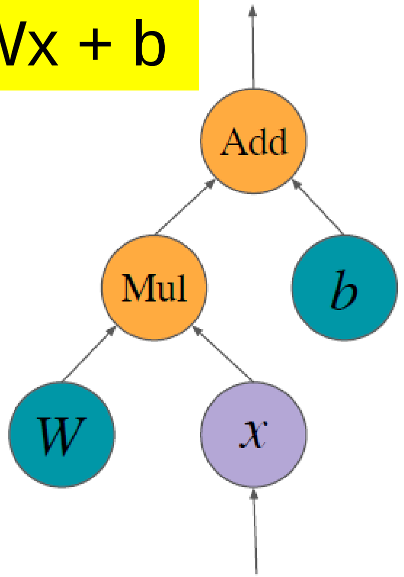
$e = c * d$



Basic Code Structure - Graphs

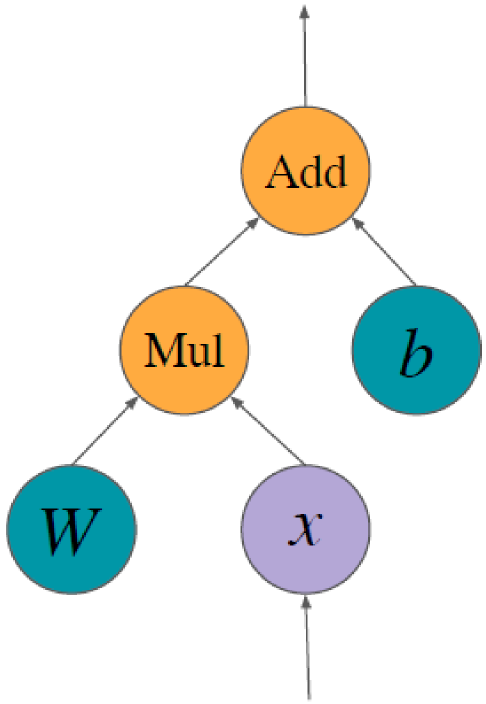
- Nodes are operators (ops), variables, and constants
- Edges are tensors
  - 0-d is a scalar
  - 1-d is a vector
  - 2-d is a matrix
  - Etc.
- TensorFlow = Tensor + Flow = Data + Flow

y = Wx + b



Computation Graph

- **Constants**
  - fixed value tensors - not trainable
- **Variables**
  - tensors initialized in a session - trainable
- **Placeholders**
  - tensors of values that are unknown during the graph construction
  - passed as input during a session
- **Ops**
  - functions on tensors



Computation Graph Structure

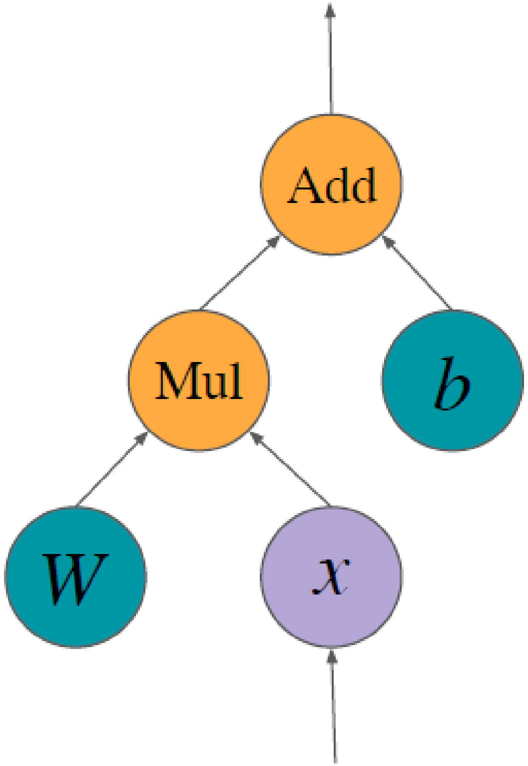
$$\hat{y} = Wx + b$$

variable

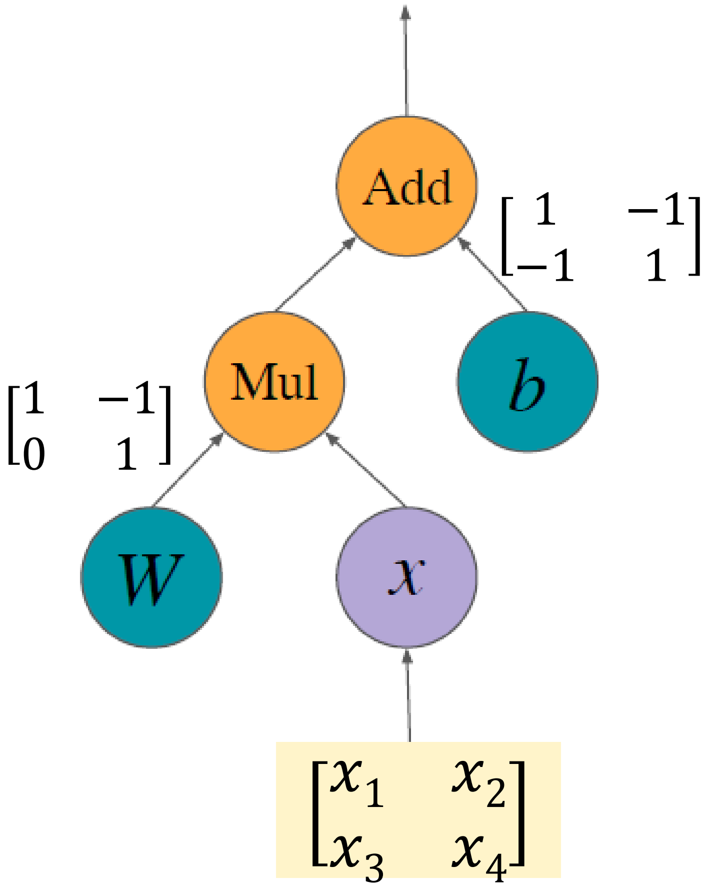
ops

placeholder

variable



Example

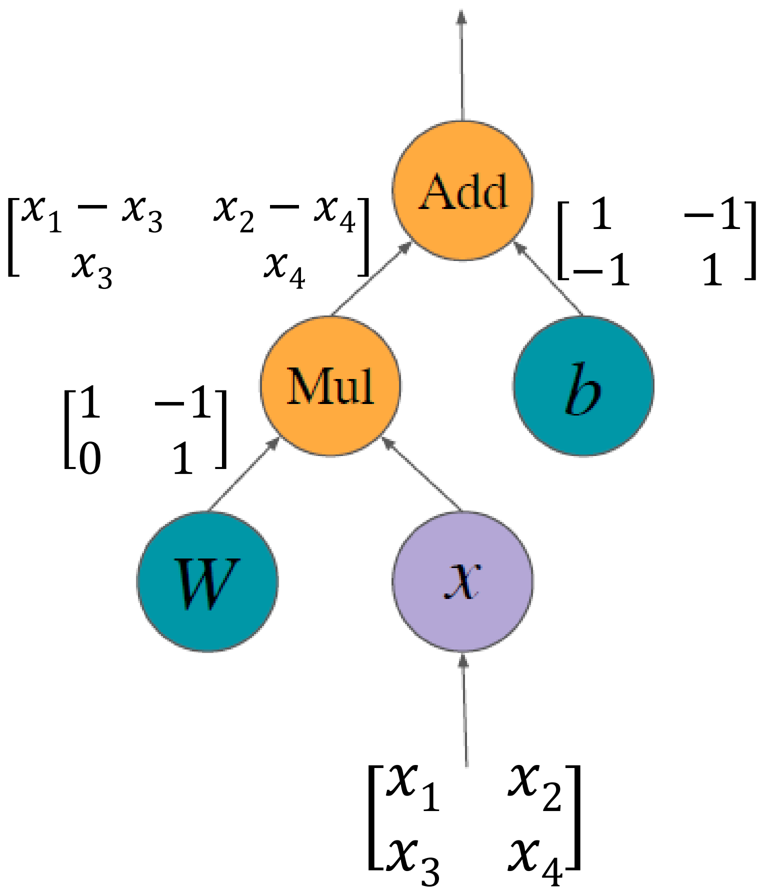


$$y = \quad Wx + b$$

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

W x b

Evaluating Mult

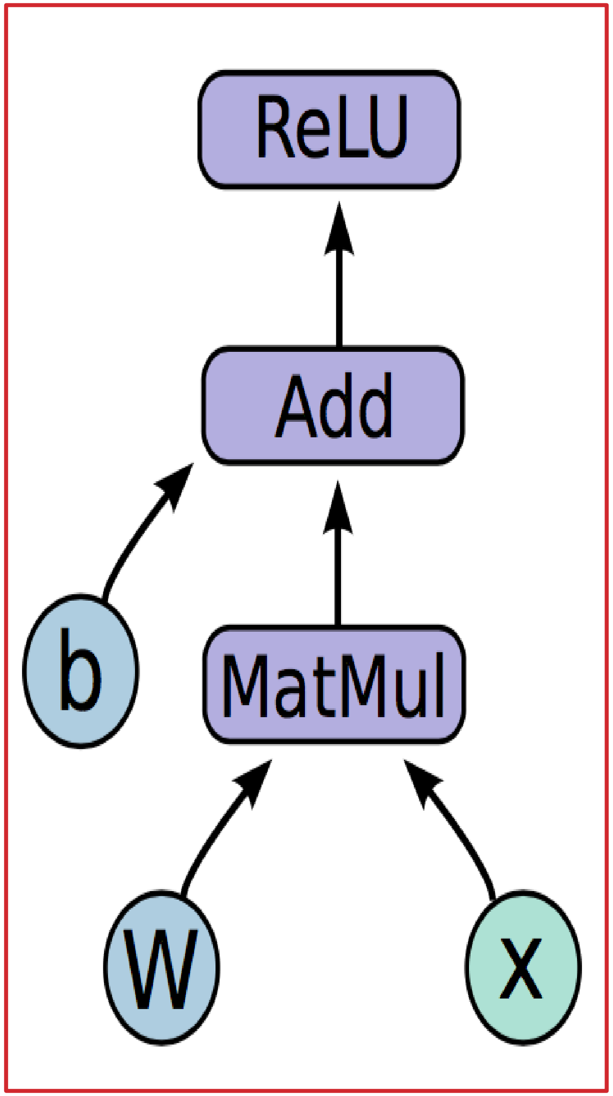


y = Wx + b

$$\underbrace{\begin{bmatrix} x_1 - x_3 & x_2 - x_4 \\ x_3 & x_4 \end{bmatrix}}_{\text{Wx}} + \underbrace{\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}}_{\text{b}}$$

Programming Model: Computation Graph

$$h = ReLU(Wx + b)$$



Programming Model : Steps for DL Implementation

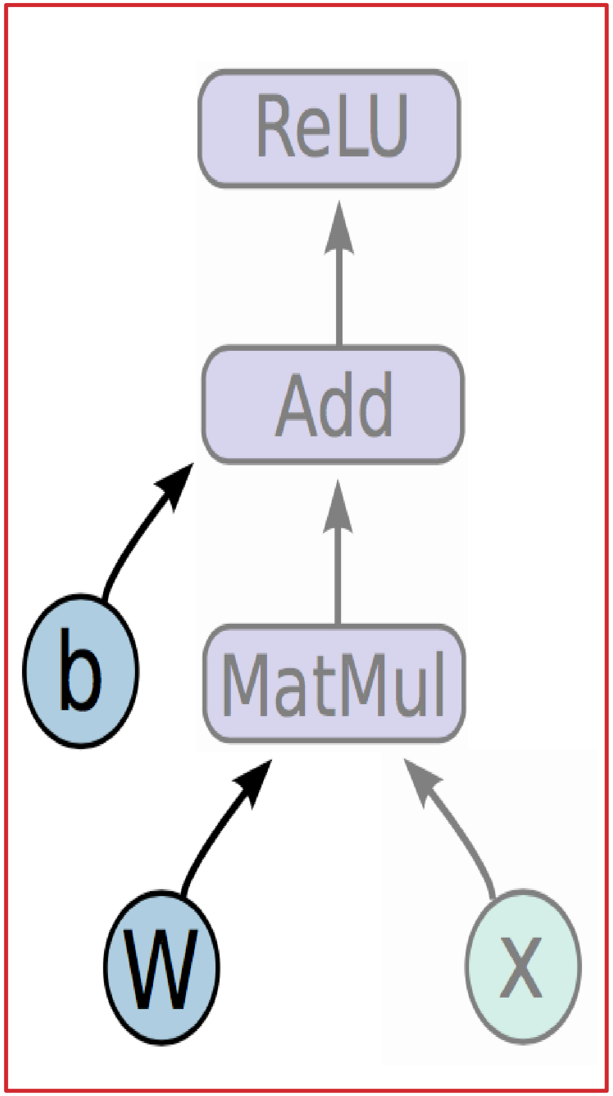
1. Define the Computation Graph
2. Initialize your data
3. Assign to hardware (CPU, GPU, etc.)

Step 1: Computation Graph

$$h = ReLU(Wx + b)$$

Variables are stateful nodes which output their current value. State is retained across multiple executions of a graph

(mostly parameters)

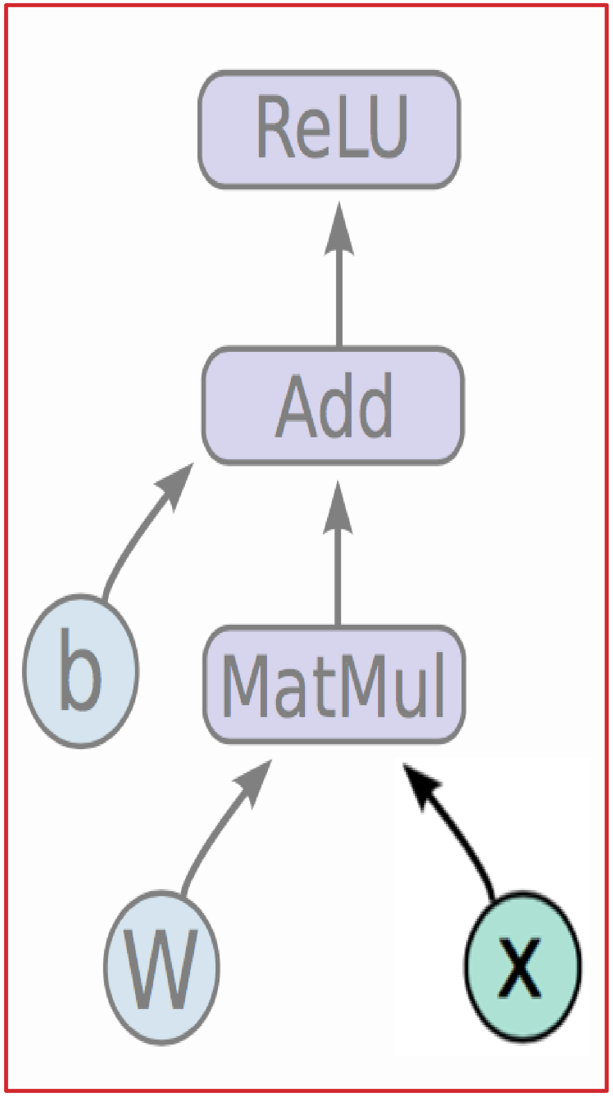


Step 1: Computation Graph

$$h = ReLU(Wx + b)$$

Placeholders are nodes whose value is fed in at execution time

(inputs, labels, …)



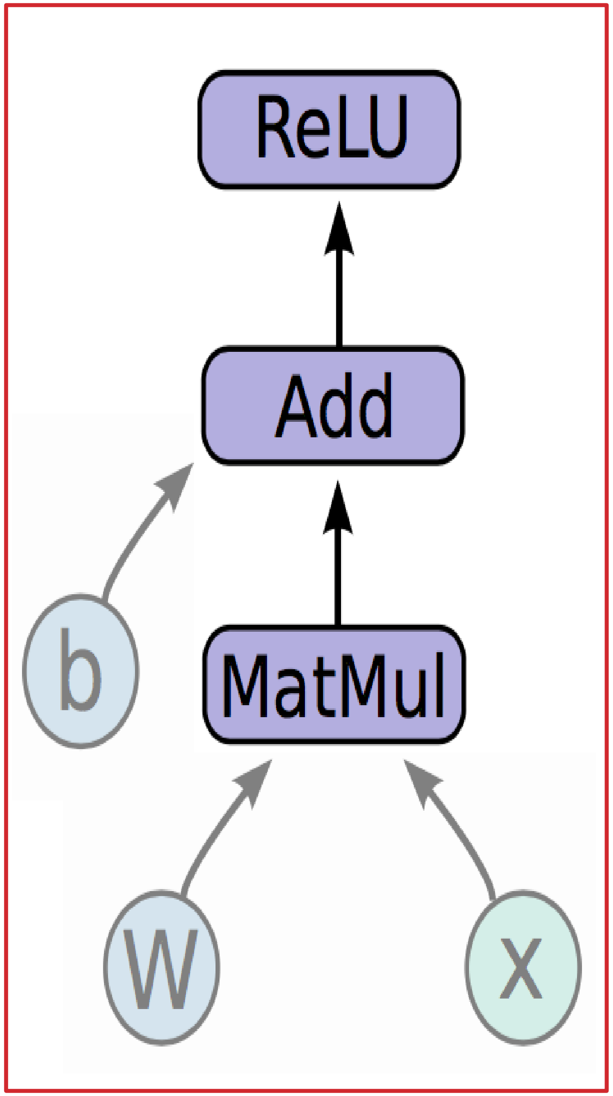
Step 1: Computation Graph

$$h = ReLU(Wx + b)$$

Mathematical operations:

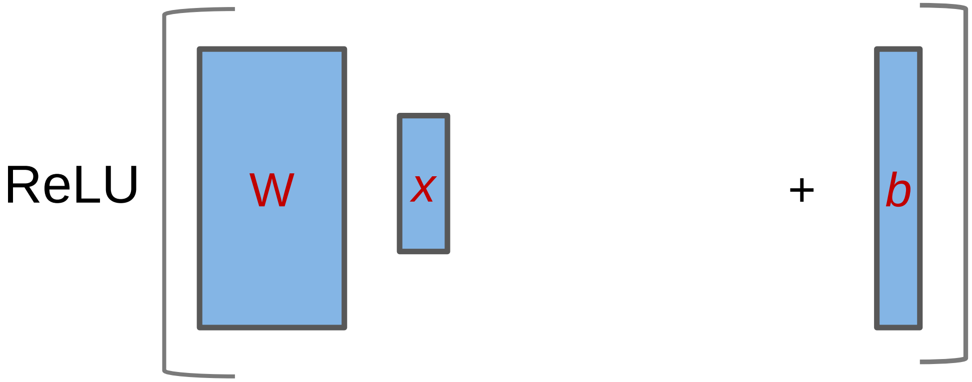
MatMul: Multiply two matrices
Add: Add elementwise
ReLU: Activate with elementwise rectified linear function

$$ReLu(x) = \begin{cases} 0, x \leq 0 \\ x, x > 0 \end{cases}$$

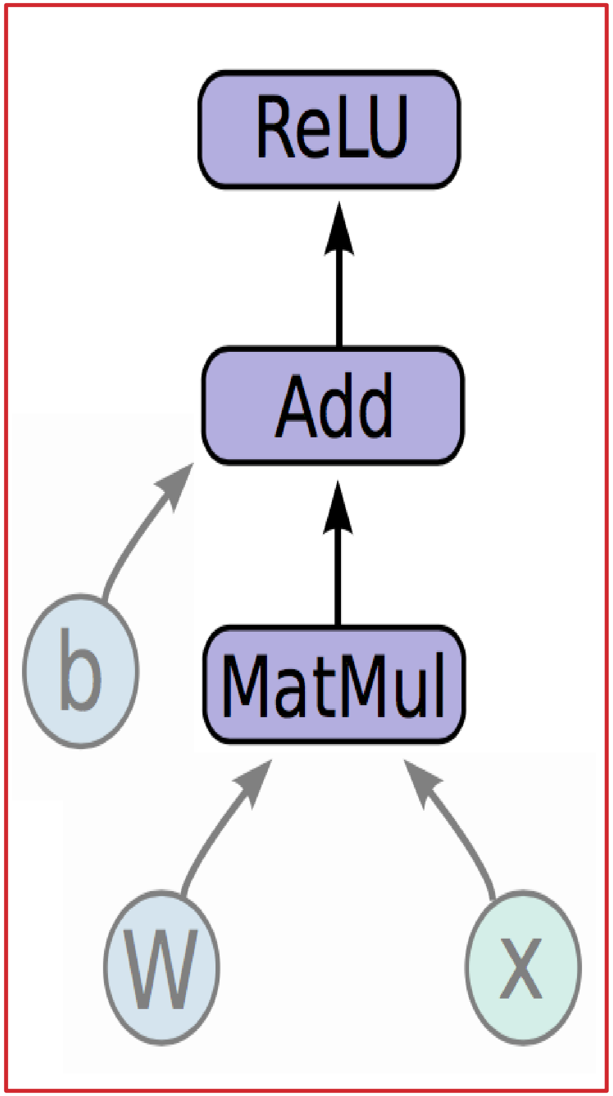


Step 2: Code and Variable Assignment

$$h = ReLU(Wx + b)$$



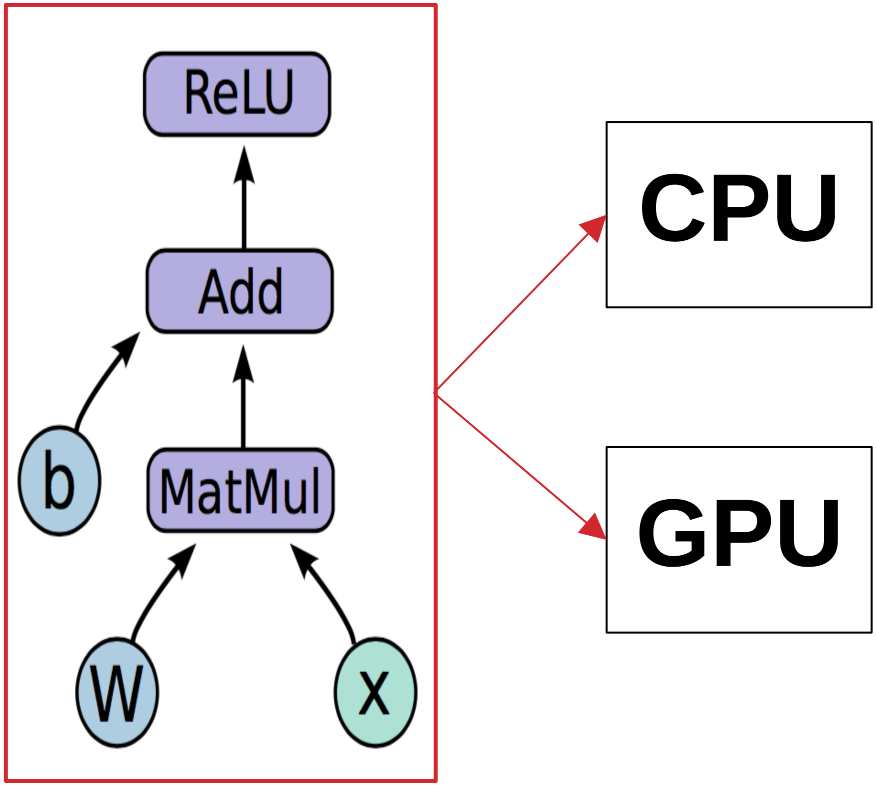

```python
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (1, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```



Step 3: Running the graph

Deploy graph with a **session**: a binding to a particular execution context

- e.g. CPU, GPU



End-to-end

- So far:
  1. Built a **graph** using **variables** and **placeholders**
  2. **Assign variable bindings**
  3. Deploy the graph onto a **session**, i.e., **execution environment**

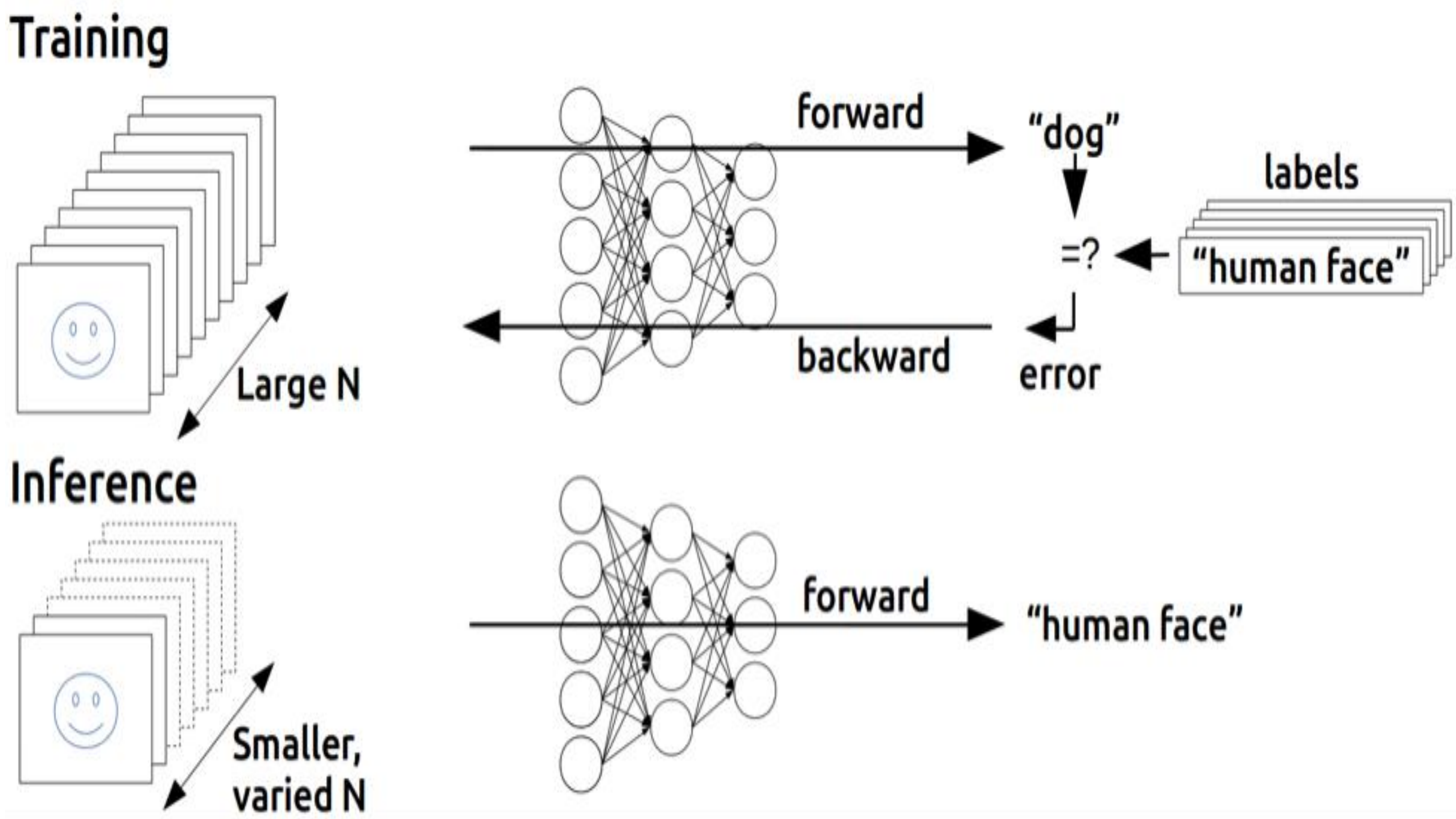
- Next: train model
  - Define loss function
  - Compute gradients

Defining loss

- Use **placeholder** for **labels**
- Build loss node using labels and **prediction**

```
prediction = tf.nn.softmax(...)  #Output of neural network
label = tf.placeholder(tf.float32, [100, 10])

cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)
```

Learning: Backpropagation



From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/

Overview

- Modularity and Computation Graphs
- Inference in Computation Graphs
- Backpropagation
  - Computation Graphs viewpoint
- DL Platforms
  - TensorFlow, Theano, etc.

Backpropagation and Gradients

- Backpropagation: an efficient way to compute the gradient
- Prerequisite
  - Backpropagation for feedforward net:
    - http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20backprop.ecm.mp4/index.html
    - Simple version: https://www.youtube.com/watch?v=ibJpTrp5mcE
  - Backpropagation through time for RNN: http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/RNN%20training%20(v6).ecm.mp4/index.html
- Understanding backpropagation by computational graph
  - Tensorflow, Theano, CNTK, etc.

Review: Chain Rule

<u>Case 1</u>

$$z = f(x) \Rightarrow y = g(x) \quad z = h(y)$$

$$x \xrightarrow{g} y \xrightarrow{h} z$$

$$\Delta x \rightarrow \Delta y \rightarrow \Delta z$$

$$\frac{dz}{dx}$$

<u>Case 2</u>

$$z = f(s) \Rightarrow x = g(s) \quad y = h(s) \quad z = k(x, y)$$

$$\begin{tikzcd} & x \arrow[dr] & \\ s \arrow[ur] \arrow[dr] & & z \\ & y \arrow[ur] & \end{tikzcd}$$

$$\begin{tikzcd} & \Delta x \arrow[dr] & \\ \Delta s \arrow[ur] \arrow[dr] & & \Delta z \\ & \Delta y \arrow[ur] & \end{tikzcd}$$

$$\frac{dz}{ds}$$

Computational Graph

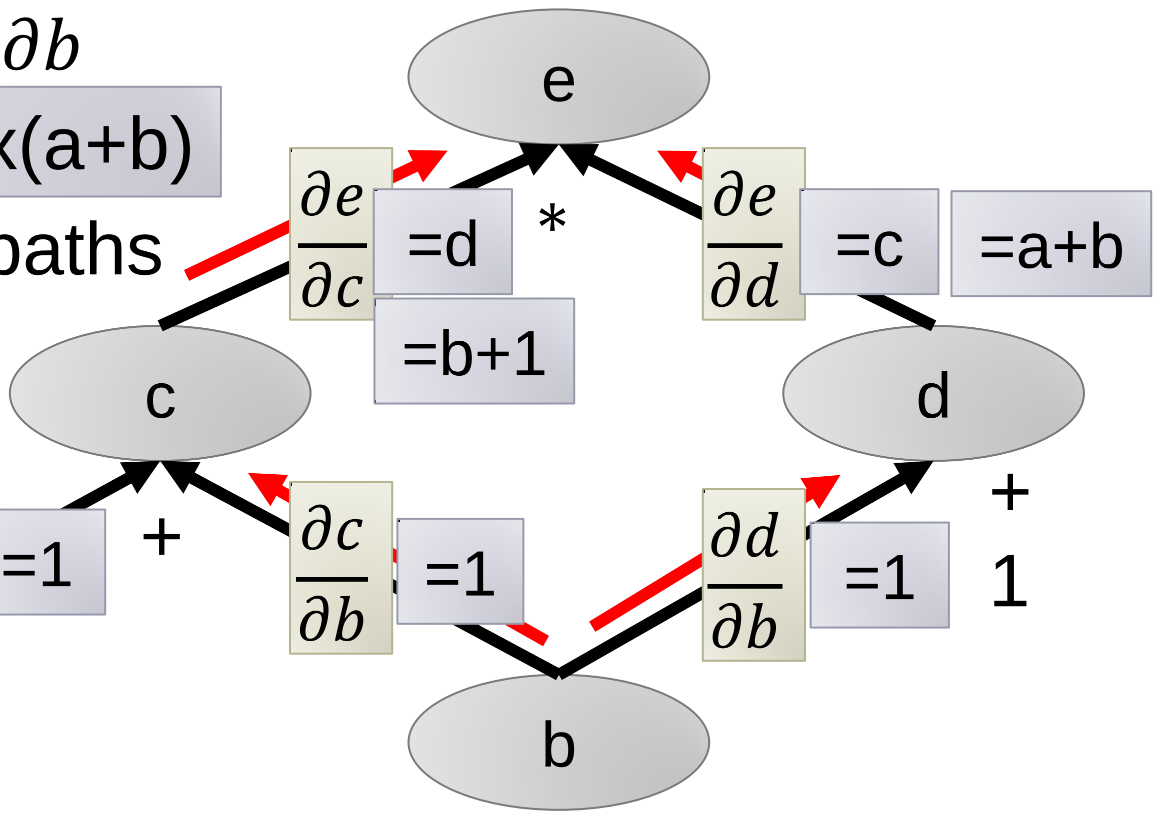
- Example: e = (a+b) $*$ (b+1)

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

Compute $\partial e/\partial b$

=1x(b+1)+1x(a+b)

Sum over all paths from b to e



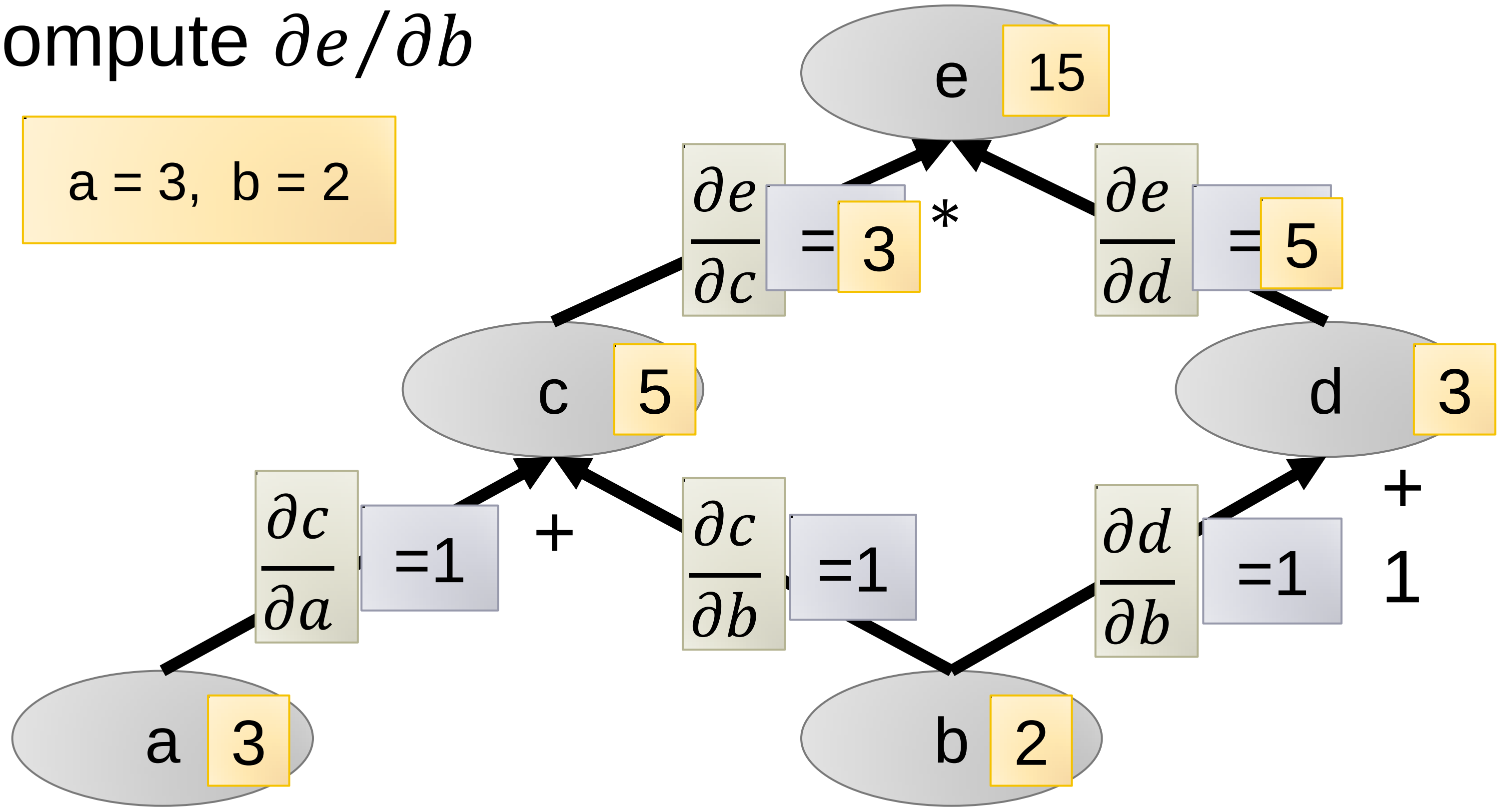
Computational Graph

- Example: e = (a+b) $*$ (b+1)

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

Compute $\partial e/\partial b$

a = 3, b = 2

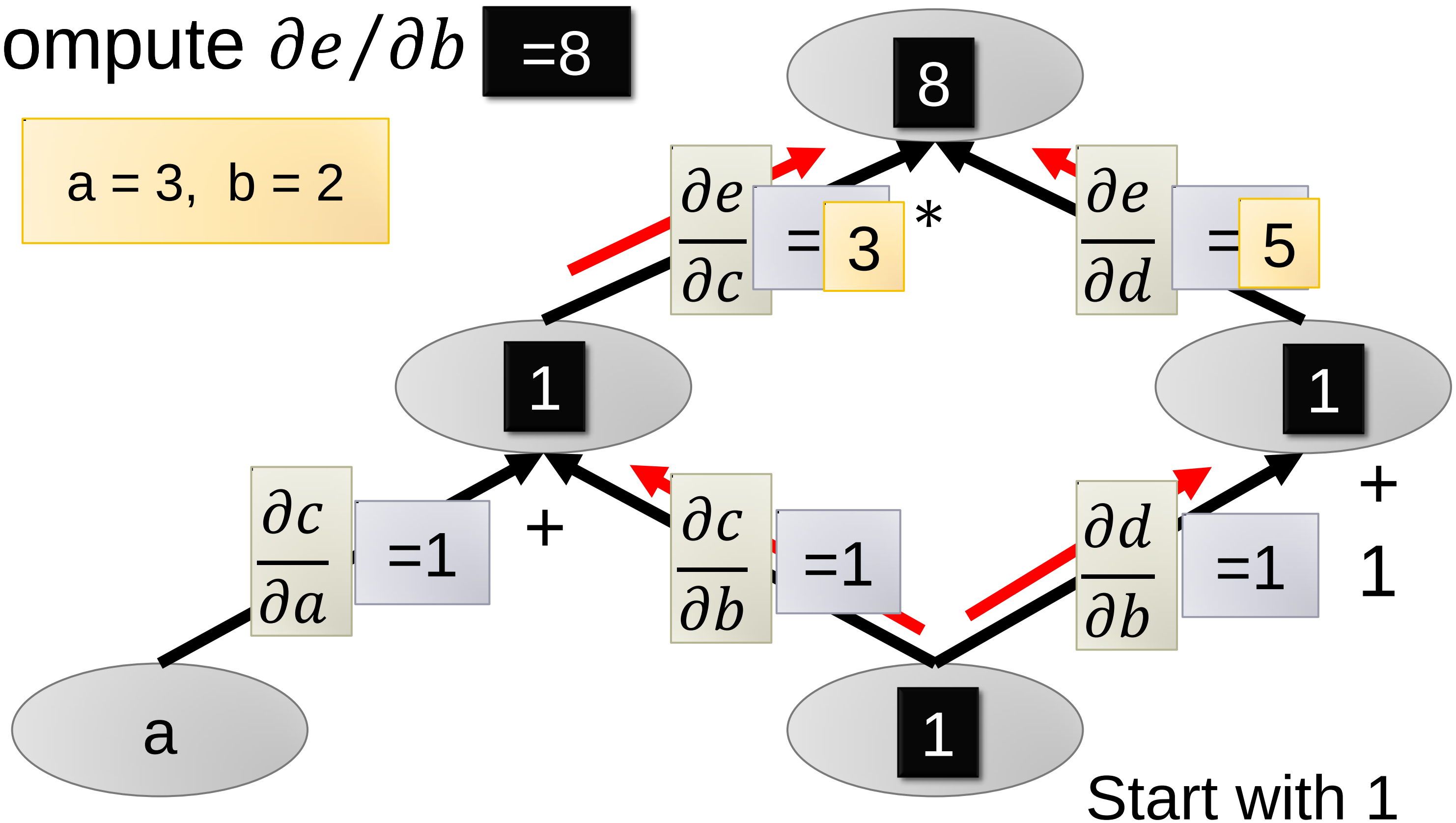


Computational Graph

- Example: e = (a+b) $*$ (b+1)

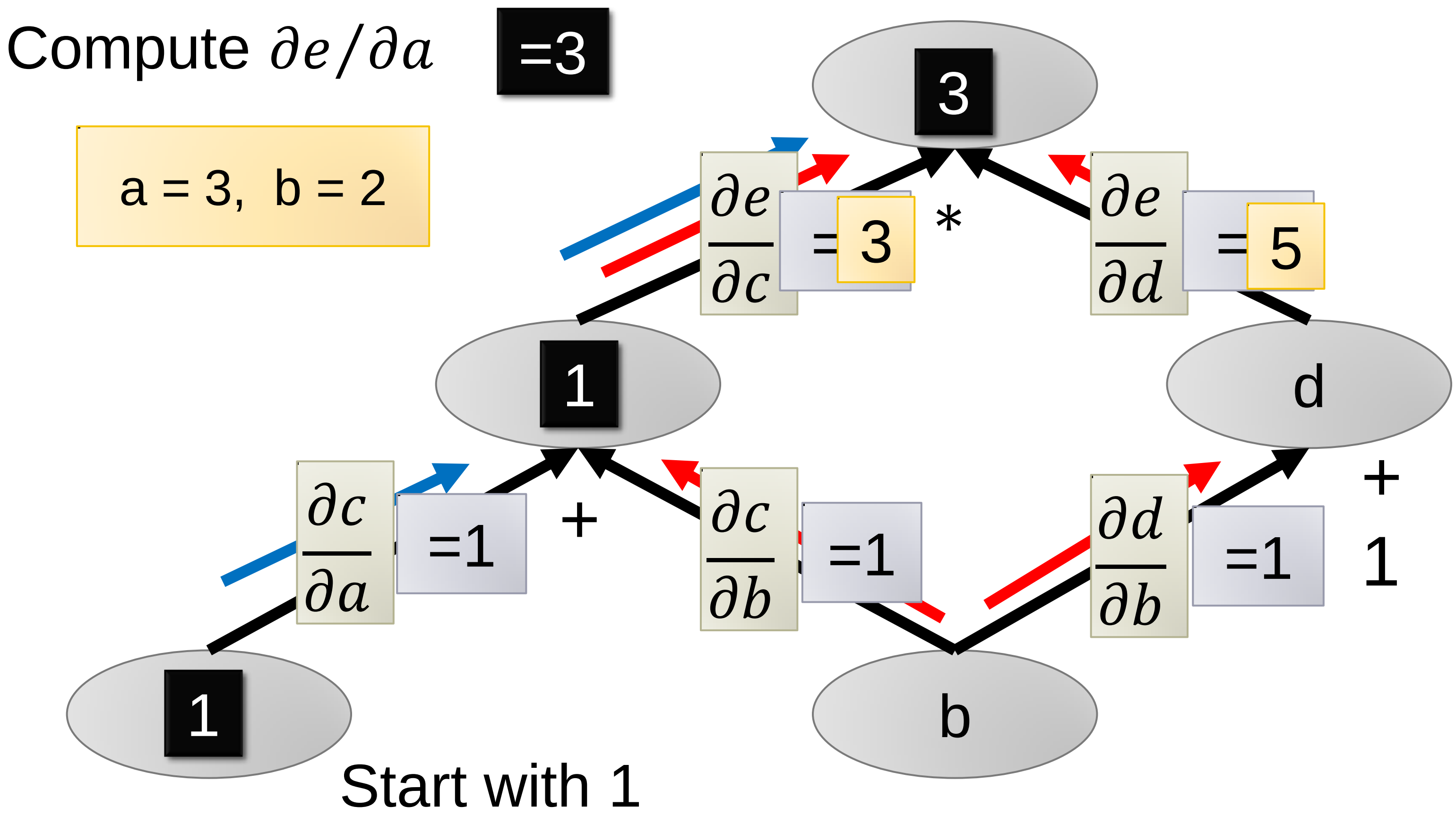
Compute $\partial e/\partial b$ =8

a = 3, b = 2



Computational Graph

- Example: e = (a+b) ∗ (b+1)



Computational Graph

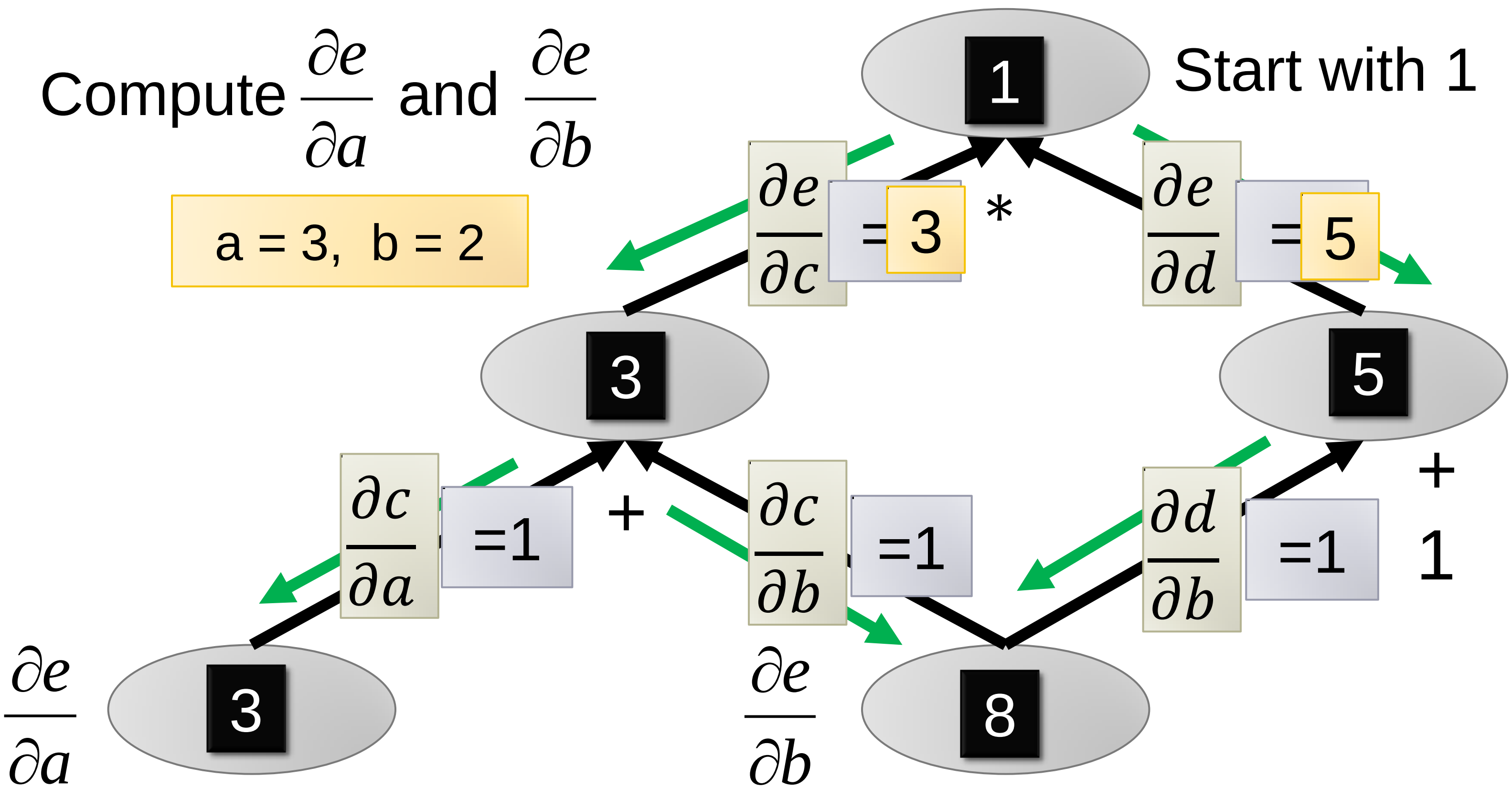
- Example: e = (a+b) $*$ (b+1)

Reverse mode

What is the benefit?

Compute $\frac{\partial e}{\partial a}$ and $\frac{\partial e}{\partial b}$

a = 3, b = 2

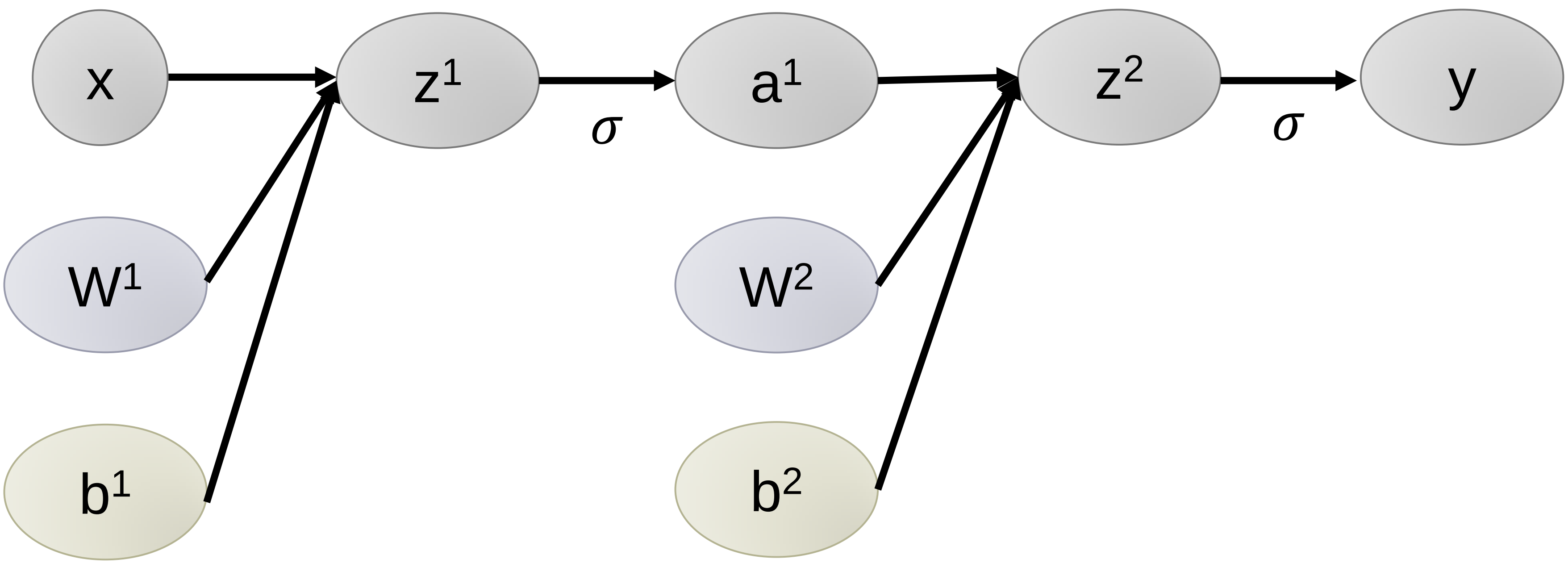


Computation Graph for FeedForward Network

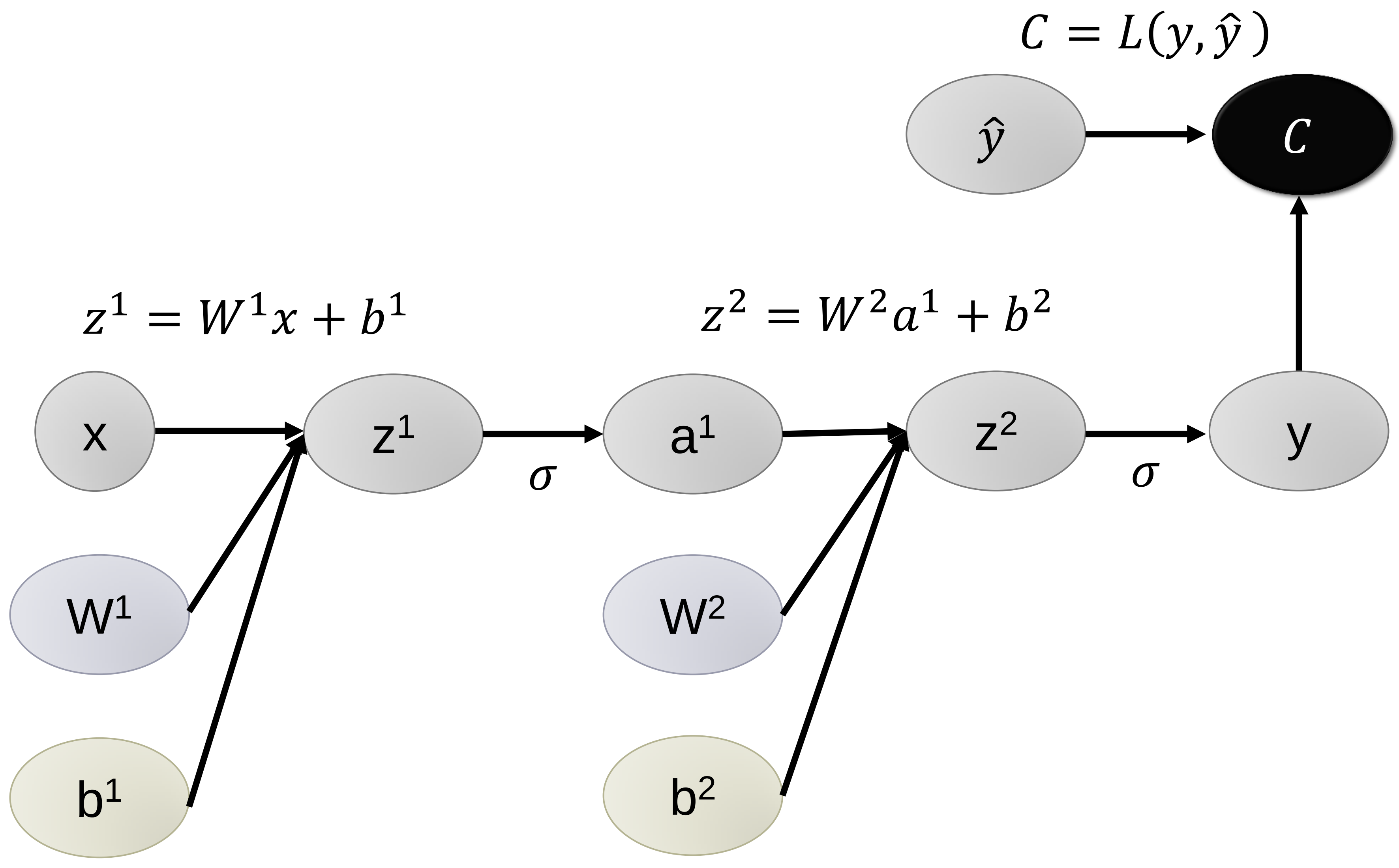
Feedforward Network

$$y = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

$$z^1 = W^1 x + b^1 \qquad z^2 = W^2 a^1 + b^2$$



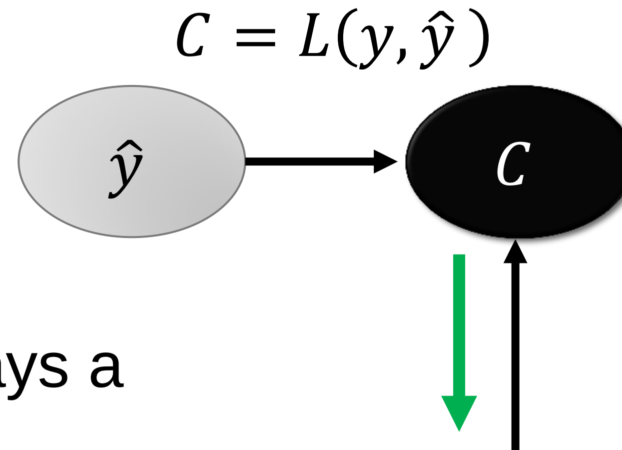
Loss Function of Feedforward Network

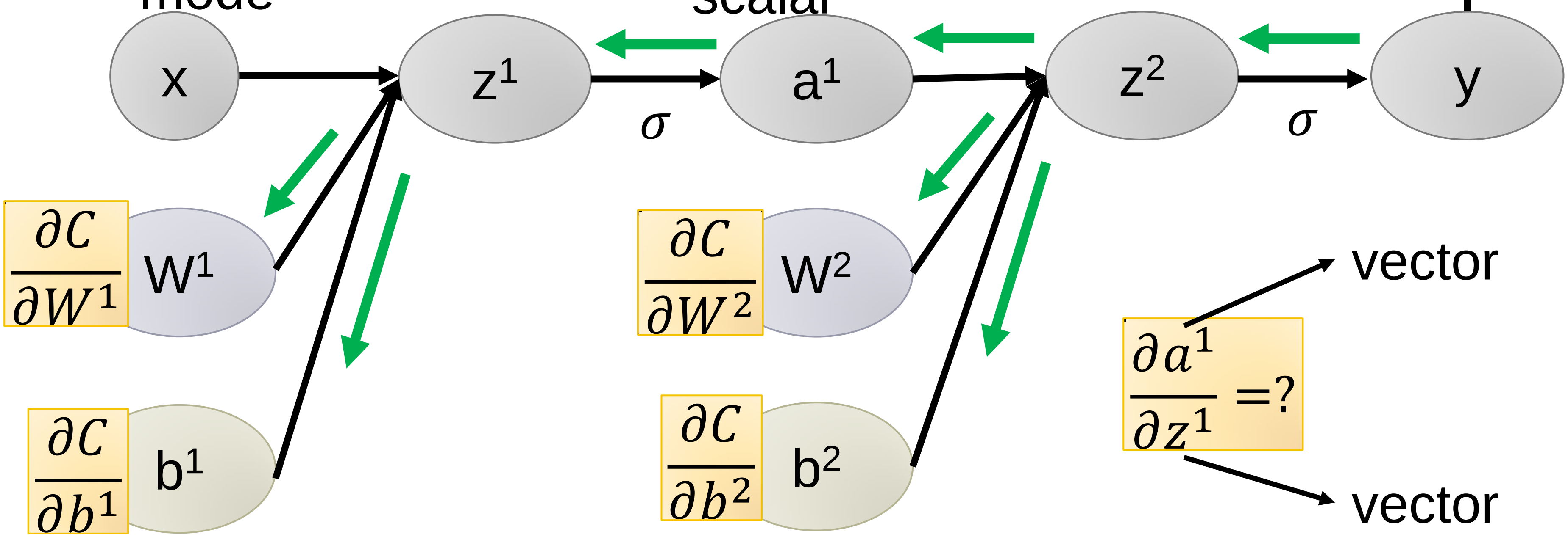


Gradient of Cost Function

To compute the gradient …

- Computing the partial derivative on the edge
- Using reverse mode ➡ Output is always a scalar





Jacobian Matrix

$$y = f(x) \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

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

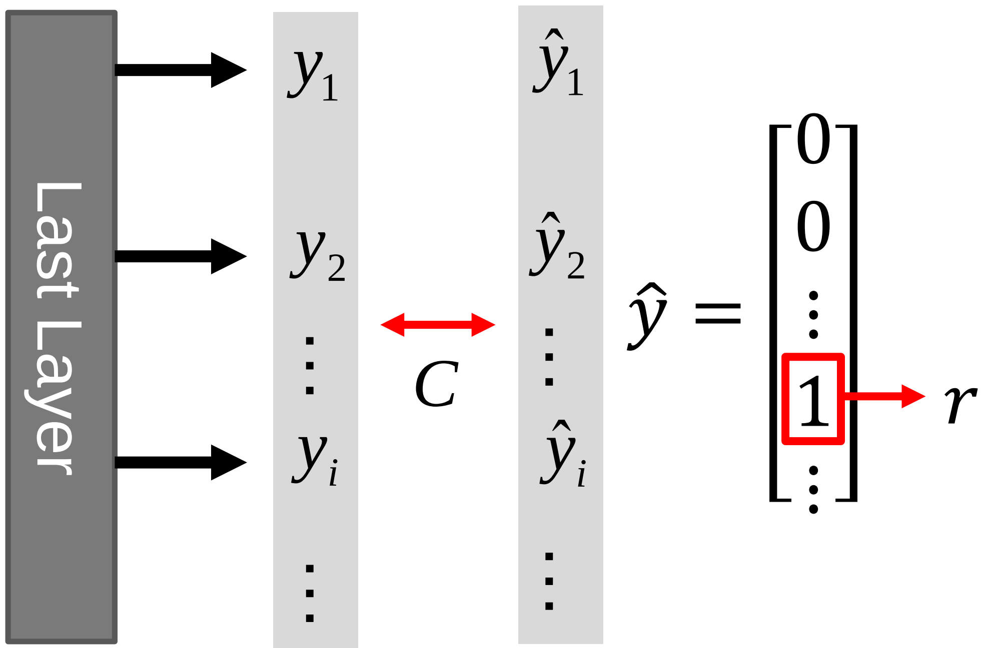
size of y

size of x

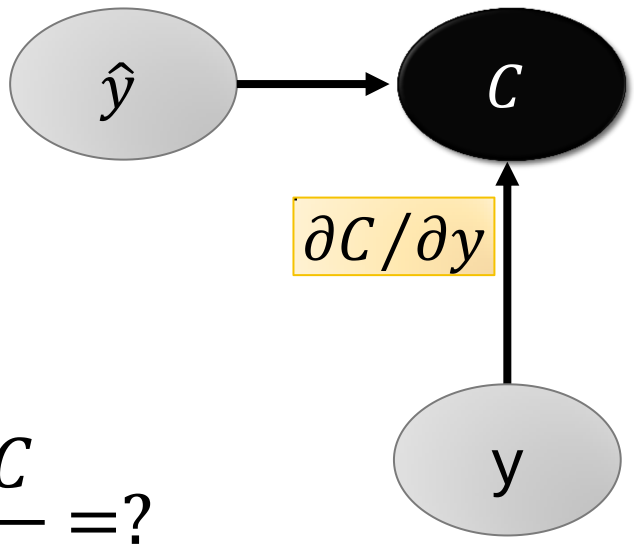
<u>Example</u>

$$\begin{bmatrix} x_1 + x_2 x_3 \\ 2x_3 \end{bmatrix} = f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) \qquad \frac{\partial y}{\partial x} = \begin{bmatrix} \qquad\qquad\qquad \end{bmatrix}$$

Gradient of Cost Function



$C = L(y, \hat{y})$



$$\frac{\partial C}{\partial y_i} = ?$$

Cross Entropy: $C = -log y_r$

$$\frac{\partial C}{\partial y} = [\qquad]$$

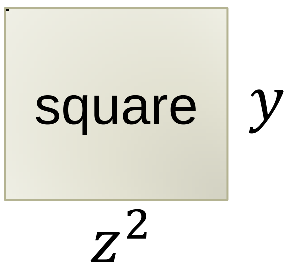
$i = r$:

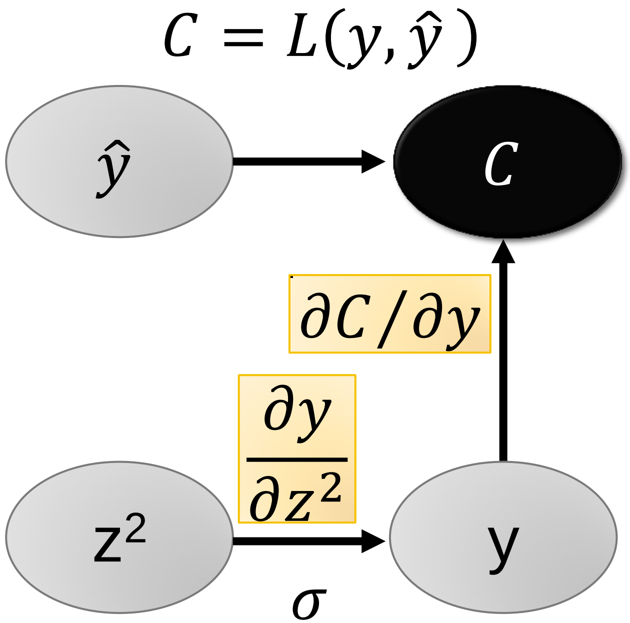
$$\partial C / \partial y_r = -1/y_r$$

$i \neq r$: $\partial C / \partial y_i = 0$

Gradient of Cost Function

$\frac{\partial y}{\partial z^2}$ is a Jacobian matrix



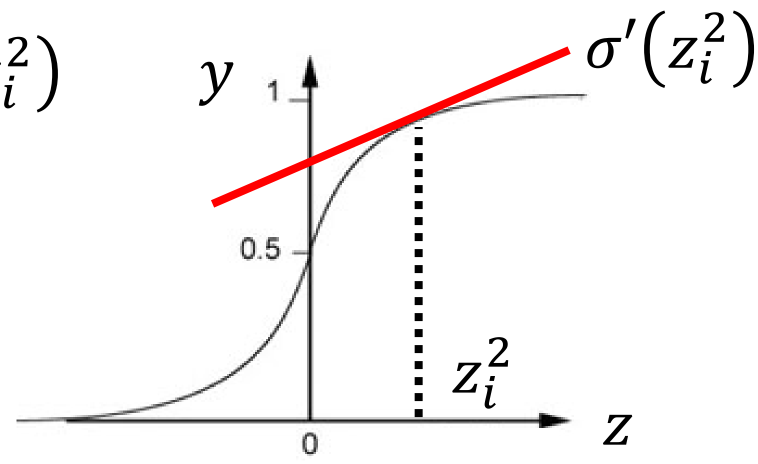


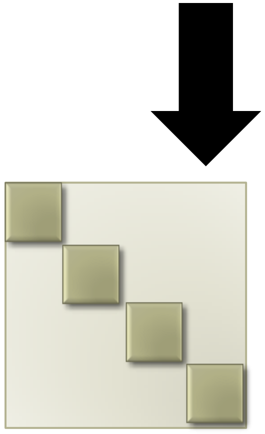
i-th row, j-th column: $\partial y_i / \partial z_j^2$

$i \neq j$: $\partial y_i / \partial z_j^2 = 0$

$i = j$: $\partial y_i / \partial z_i^2 = \sigma'(z_i^2)$

$y_i = \sigma(z_i^2)$

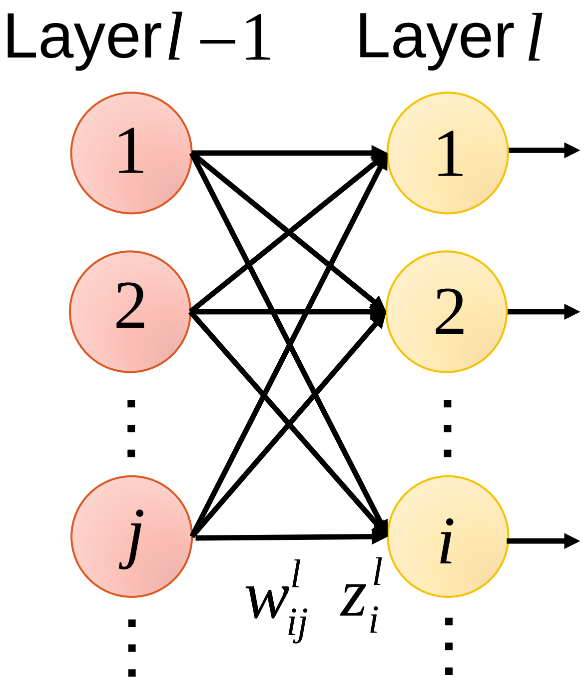




Diagonal Matrix

Review: Backpropagation

$$\frac{\partial C}{\partial w_{ij}^{l}} = \boxed{\frac{\partial z_i^l}{\partial w_{ij}^l}} \boxed{\frac{\partial C}{\partial z_i^l}}$$



$$\frac{\partial z_i^l}{\partial w_{ij}^l} = \begin{cases} a_j^{l-1} & l > 1 \\ x_j & l = 1 \end{cases}$$

$$\frac{\partial C}{\partial z_i^l} = \delta_i^l$$

Error signal

Forward Pass

$$z^1 = W^1 x + b^1$$

$$a^1 = \sigma\left(z^1\right)$$

$$\cdots\cdots$$

$$z^{l-1} = W^{l-1} a^{l-2} + b^{l-1}$$

$$a^{l-1} = \sigma\left(z^{l-1}\right)$$

Backward Pass

$$\delta^L = \sigma'\left(z^L\right) \bullet \nabla_y C$$

$$\delta^{L-1} = \sigma'\left(z^{L-1}\right) \bullet \left(W^L\right)^T \delta^L$$

$$\cdots\cdots$$

$$\delta^l = \sigma'\left(z^l\right) \bullet \left(W^{l+1}\right)^T \delta^{l+1}$$

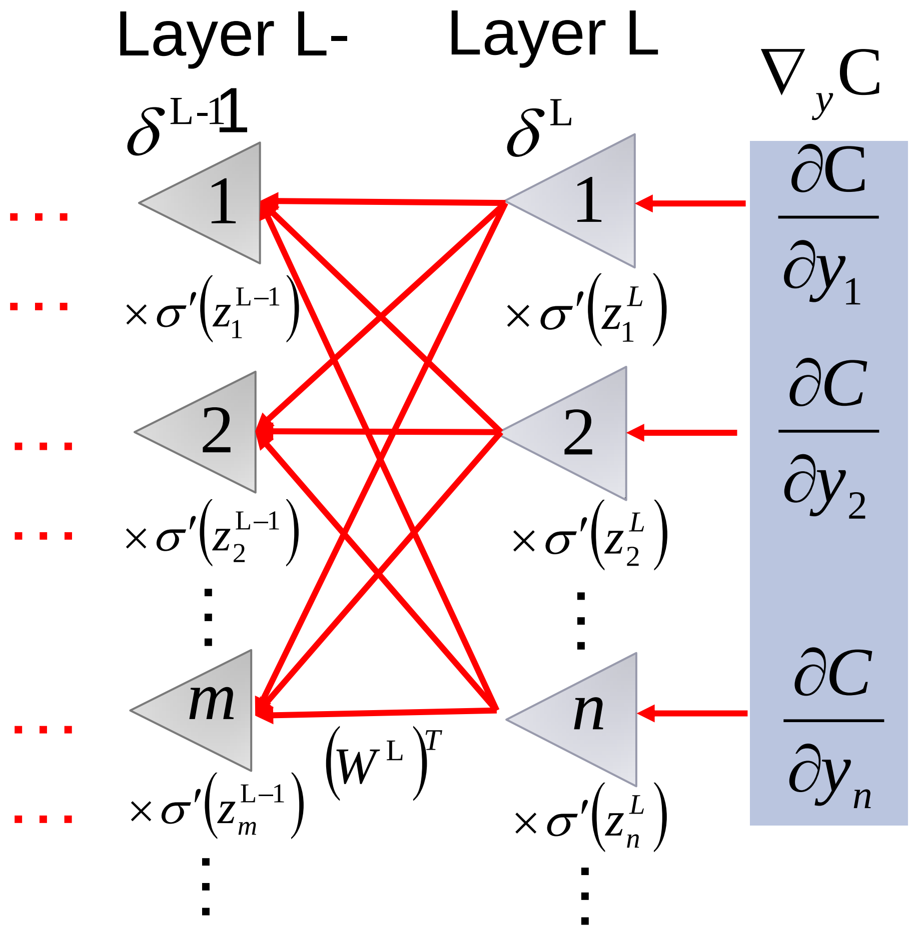
$$\cdots\cdots$$

Review: Backpropagation

$$\frac{\partial C}{\partial w_{ij}^l} = \boxed{\frac{\partial z_i^l}{\partial w_{ij}^l}} \boxed{\frac{\partial C}{\partial z_i^l}}$$

Error signal

$$\boxed{\delta_i^l}$$



(we do not use softmax here)

Backward Pass

$$\delta^L = \sigma'(z^L) \bullet \nabla_y C$$

$$\delta^{L-1} = \sigma'(z^{L-1}) \bullet (W^L)^T \delta^L$$

$$\cdots\cdots$$

$$\delta^l = \sigma'(z^l) \bullet (W^{l+1})^T \delta^{l+1}$$

$$\cdots\cdots$$

Review

- Modularity and Computation Graphs
- Inference in Computation Graphs
- Backpropagation
  - Computation Graph viewpoint
- DL Platforms
  - TensorFlow, Theano, etc.

SoftMax

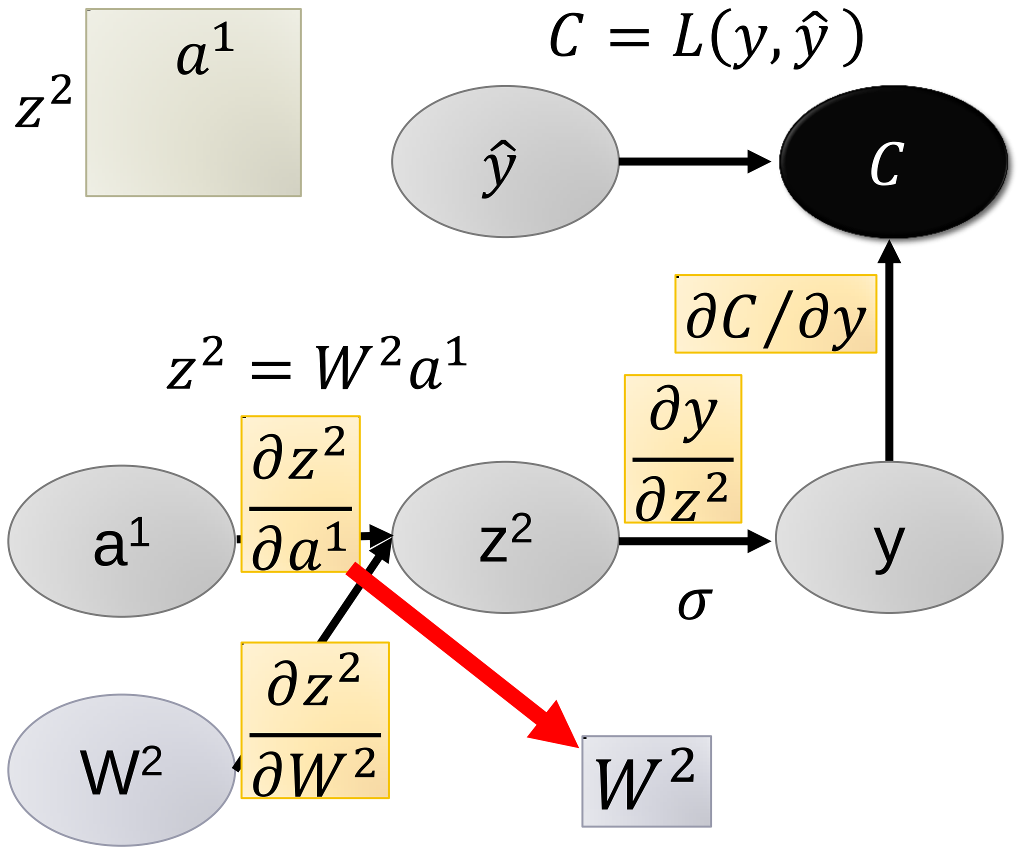
$\frac{\partial z^2}{\partial a^1}$ is a Jacobian matrix

i-th row, j-th column:

$$\frac{\partial z_i^2}{\partial a_j^1} =$$

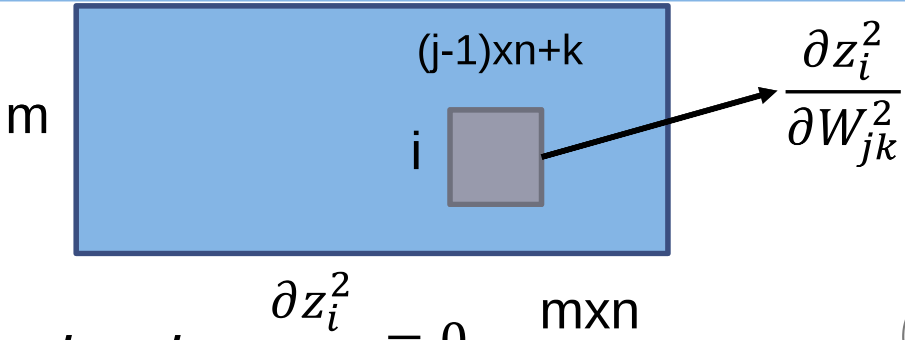
$$z_i^2 = w_{i1}^2 a_1^1 + w_{i2}^2 a_2^1 + \cdots + w_{in}^2 a_n^1$$

$$\begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_i^l \\ \vdots \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ \vdots \\ a_i^{l-1} \\ \vdots \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$



How about softmax? ☺

$$\frac{\partial z^2}{\partial W^2} =$$



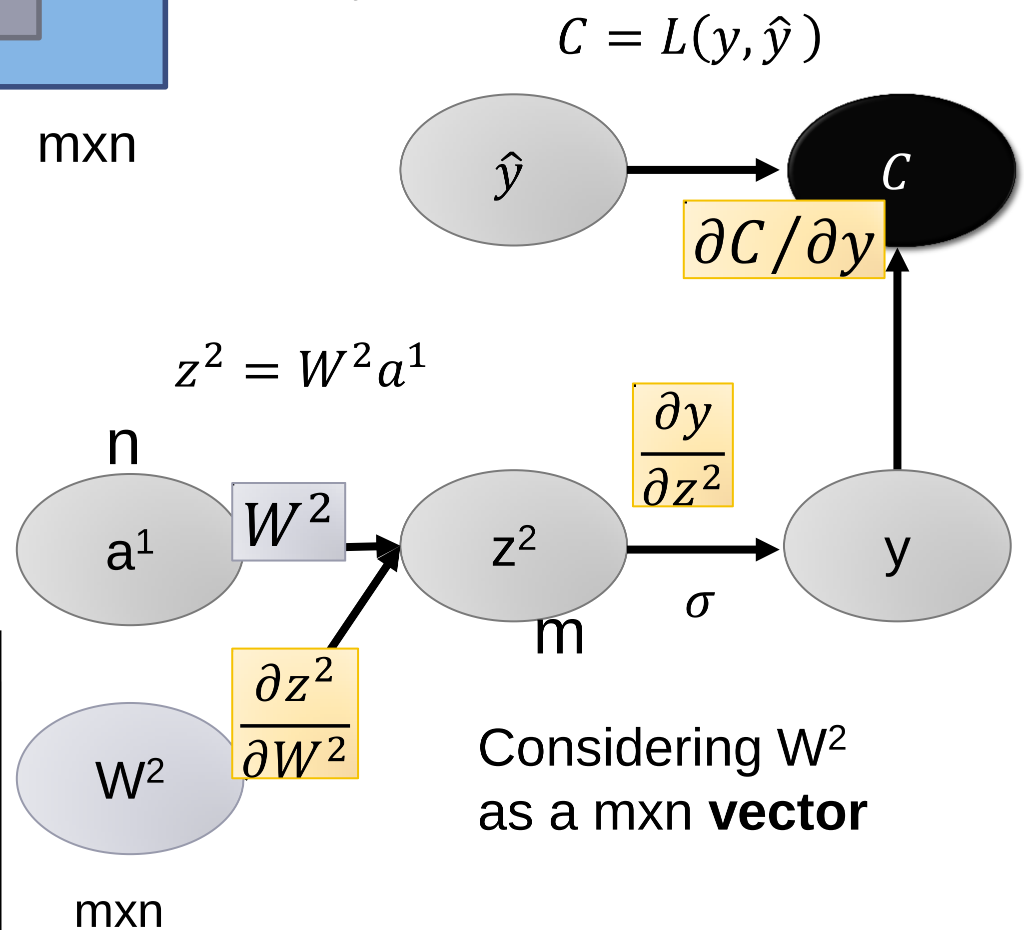
$$\frac{\partial z_i^2}{\partial W_{jk}^2} = ?$$

$$i \neq j: \quad \frac{\partial z_i^2}{\partial W_{jk}^2} = 0$$

$$i = j: \quad \frac{\partial z_i^2}{\partial W_{ik}^2} = a_k^1$$

$$z_i^2 = w_{i1}^2 a_1^1 + w_{i2}^2 a_2^1 + \cdots + w_{in}^2 a_n^1$$

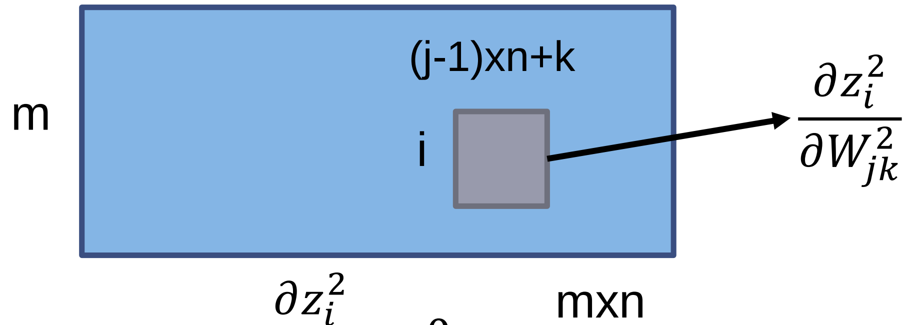
$$\begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_i^l \\ \vdots \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ w_{21}^l & w_{22}^l & \\ \vdots & & \ddots \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ \vdots \\ a_i^{l-1} \\ \vdots \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_i^l \\ \vdots \end{bmatrix}$$



Considering W^2 as a mxn **vector**

(considering $\partial z^2 / \partial W^2$ as a tensor makes thing easier)

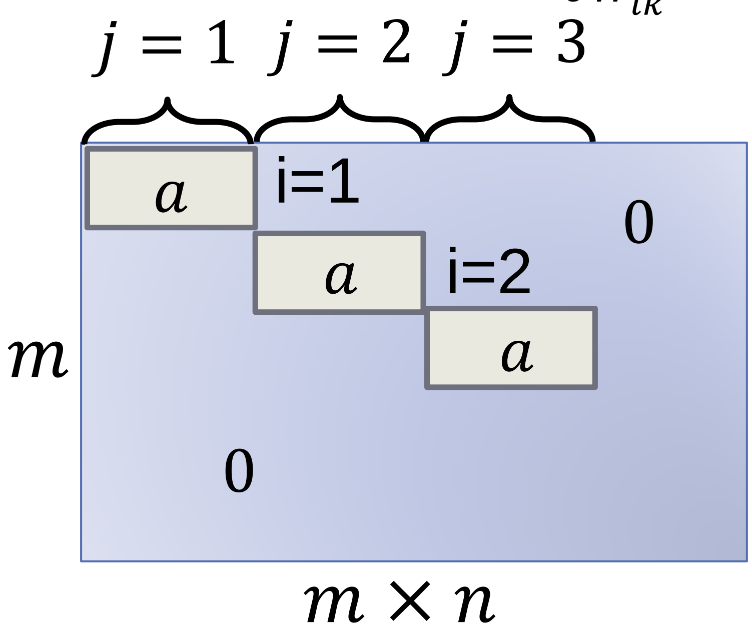
$$\frac{\partial z^2}{\partial W^2} =$$

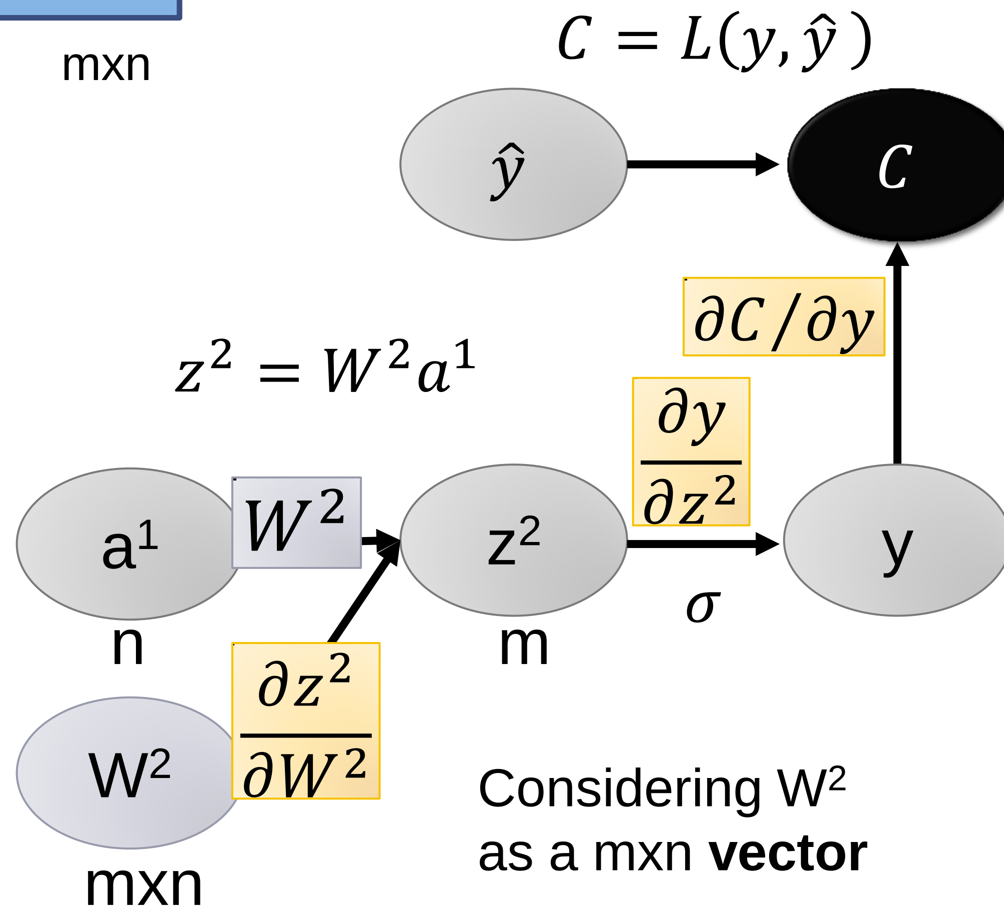


$$\frac{\partial z_i^2}{\partial W_{jk}^2} = ?$$

$$i \neq j: \quad \frac{\partial z_i^2}{\partial W_{jk}^2} = 0$$

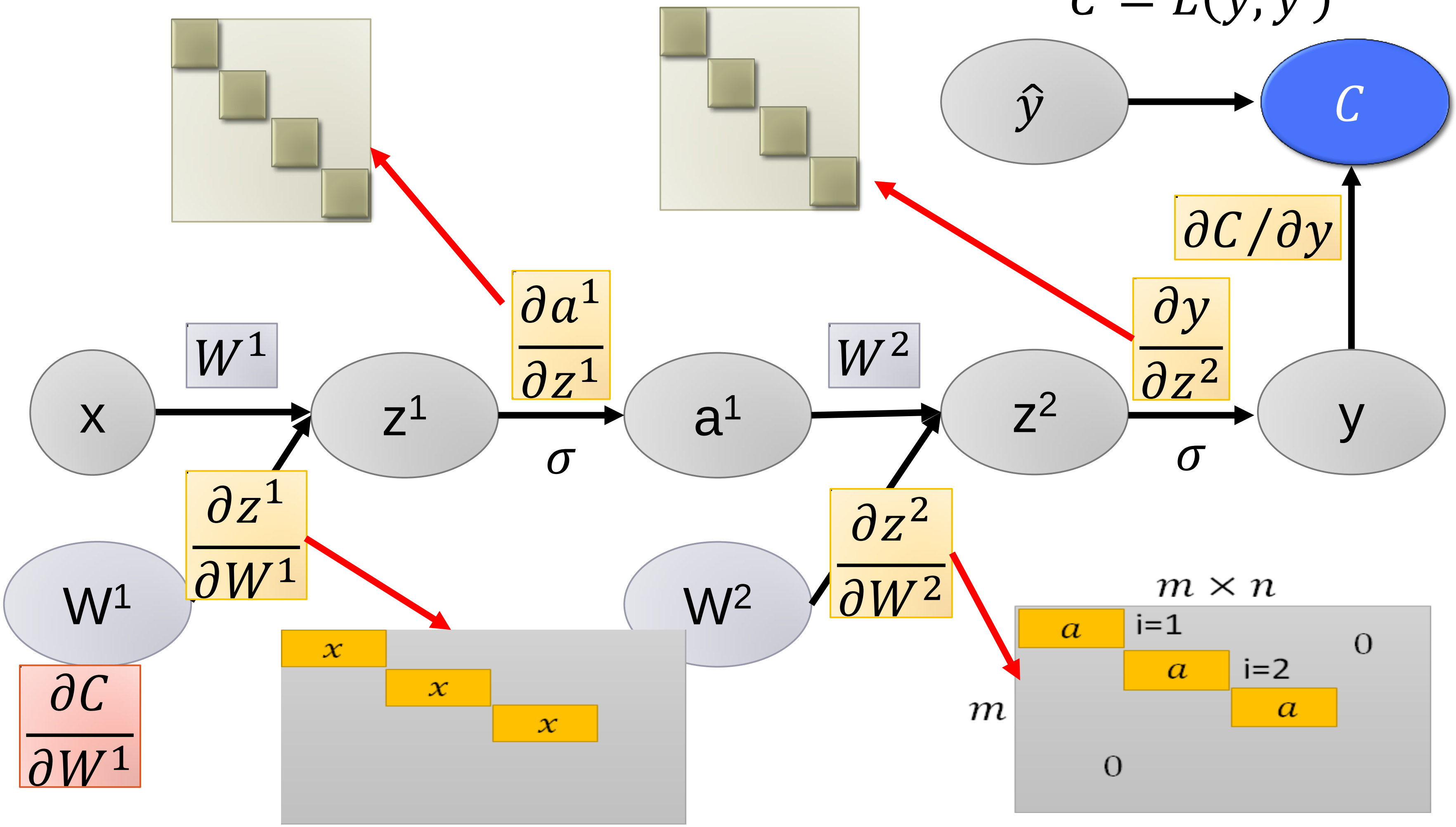
$$i = j: \quad \frac{\partial z_i^2}{\partial W_{ik}^2} = a_k^1$$





$$\frac{\partial C}{\partial W^1} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial z^2} W^2 \frac{\partial a^1}{\partial z^1} \frac{\partial z^1}{\partial W^1} = [\cdots \quad \frac{\partial C}{\partial W^1_{ij}} \quad \cdots]$$

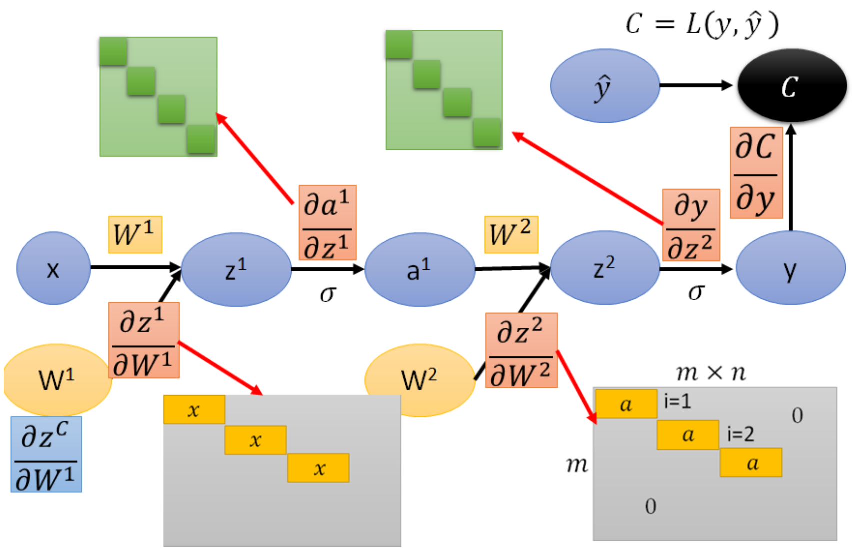
$$C = L(y, \hat{y})$$



Question

$$\frac{\partial C}{\partial w_{ij}^{l}} = \boxed{\frac{\partial z_i^l}{\partial w_{ij}^l}} \boxed{\frac{\partial C}{\partial z_i^l}}$$

Error signal



Forward Pass

$$z^1 = W^1 x + b^1$$

$$a^1 = \sigma(z^1)$$

$$\cdots\cdots$$

$$z^{l-1} = W^{l-1} a^{l-2} + b^{l-1}$$

$$a^{l-1} = \sigma(z^{l-1})$$

Backward Pass

$$\delta^L = \sigma'(z^L) \bullet \nabla_y C$$

$$\delta^{L-1} = \sigma'(z^{L-1}) \bullet (W^L)^T \delta^L$$

$$\cdots\cdots$$

$$\delta^l = \sigma'(z^l) \bullet (W^{l+1})^T \delta^{l+1}$$

$$\cdots\cdots$$

Q: Only backward pass for computational graph?

Q: Do we get the same results from the two different approaches?