CS6423: Scalable Computing

Gregory Provan

Spring 2020

Lecture 8: TensorFlow and Computation Graphs

Based on notes from Hung-Yi Lee,
Andrej Karpathy, Fei-Fei Li, Justin Johnson

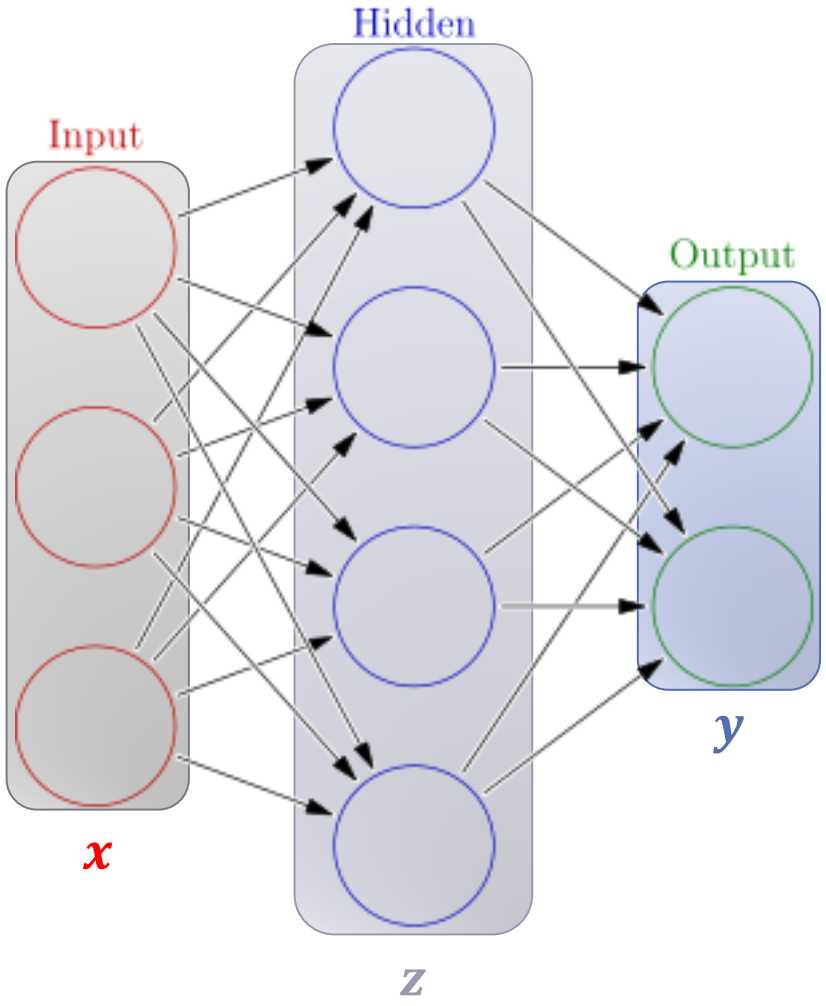
Motivation

- 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

$$z = f(\mathbf{W}_1 x + b_1)$$

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

Activation functions

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

---

26 learnable **parameters**

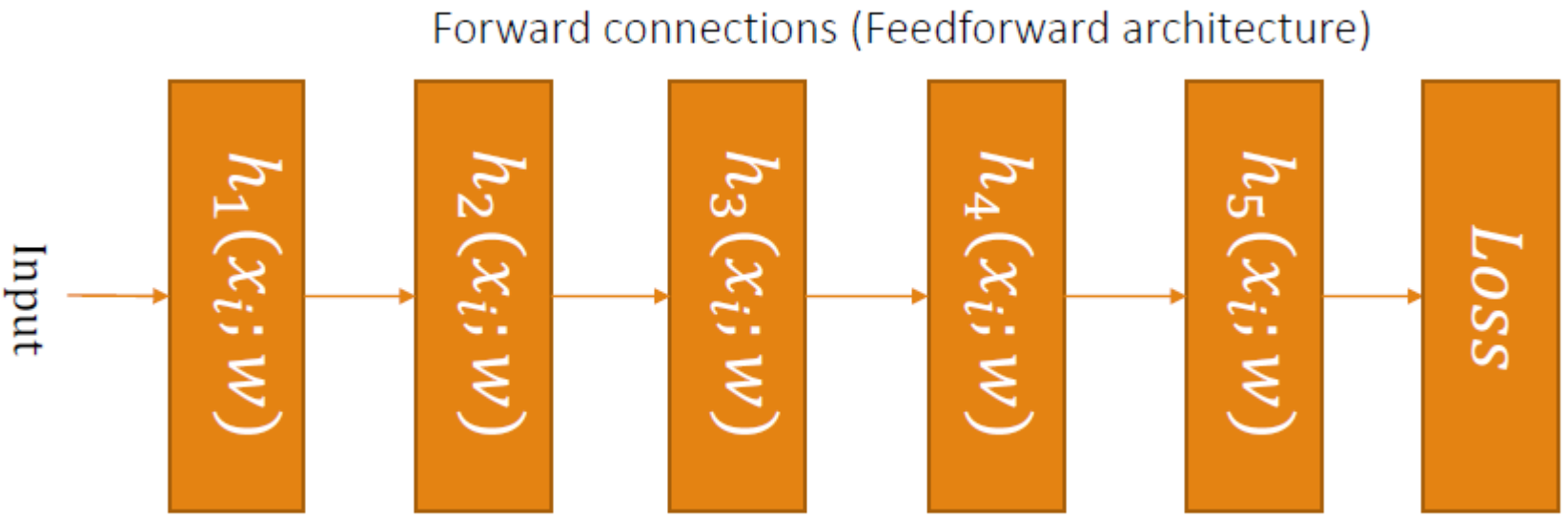
Demo

Neural Network: Definition

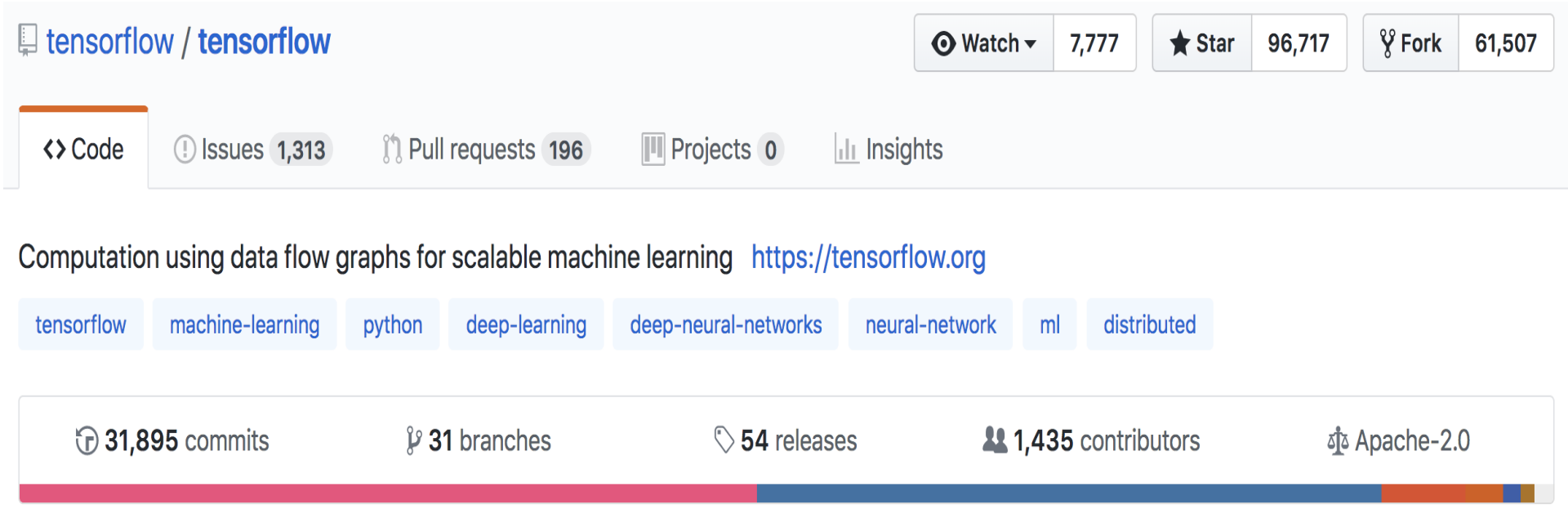
- A family of parametric, non-linear and hierarchical representation learning functions
  - massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a^L(x;w^1,\ldots,w^L)=h^L(h^{L\text{-}1}\ldots h^1(x,w^1),w^{L\text{-}1}),w^L)$
  - x:input,
  - 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



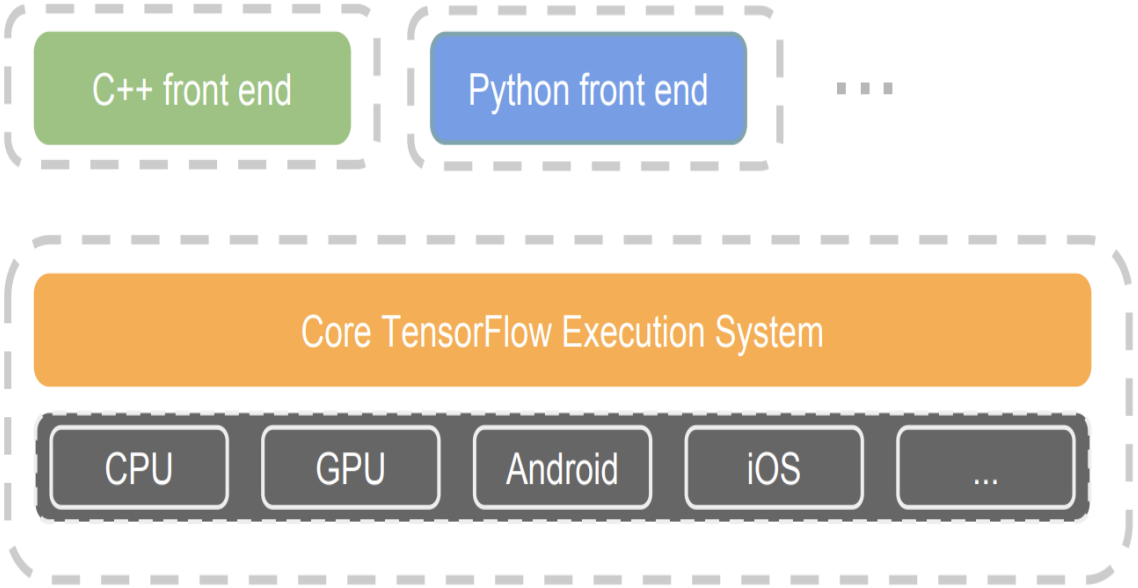
What is TensorFlow?



- 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

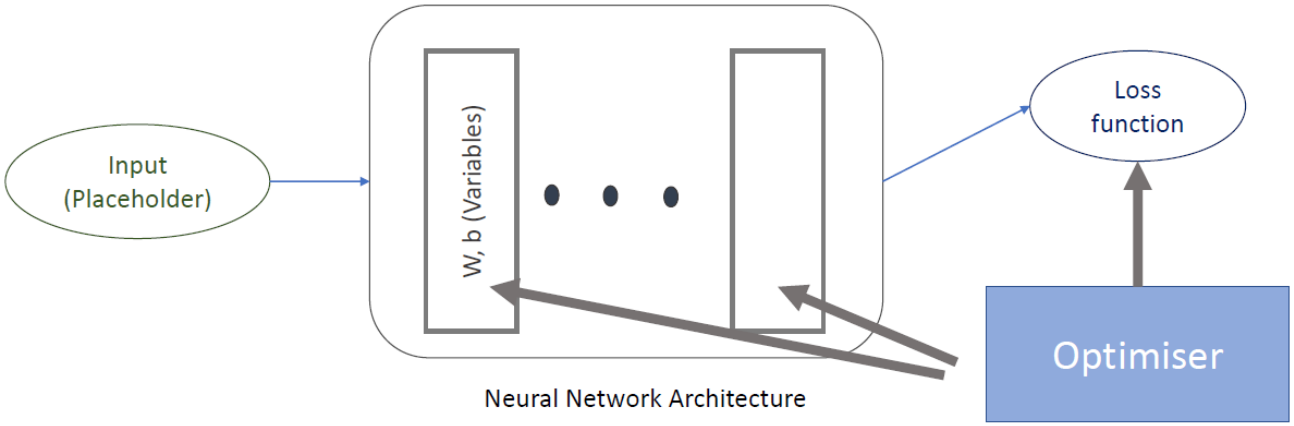
TF Learn

- 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)
```

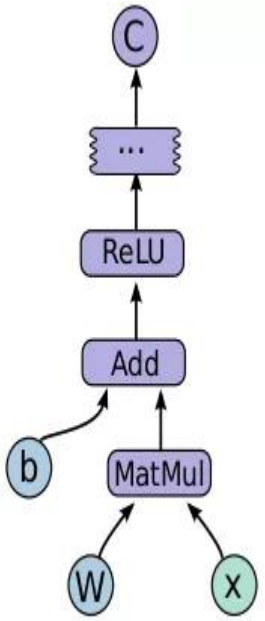
How Does TensorFlow Work?

- 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



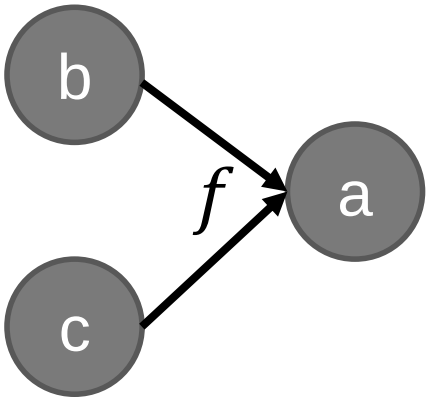

```
import tensorflow
import numpy

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

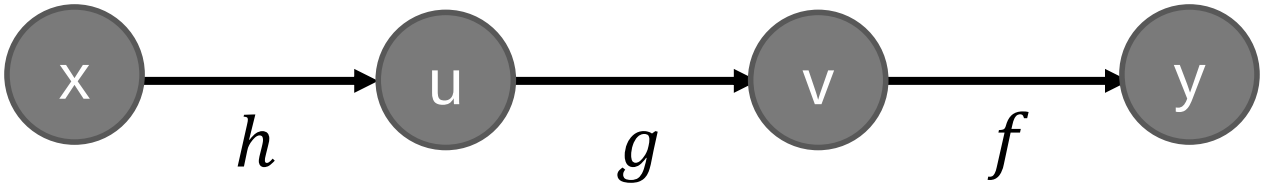
- A “language” describing a function
  - ***Node***: variable (scalar, vector, tensor ……)
  - ***Edge***: operation (simple function)



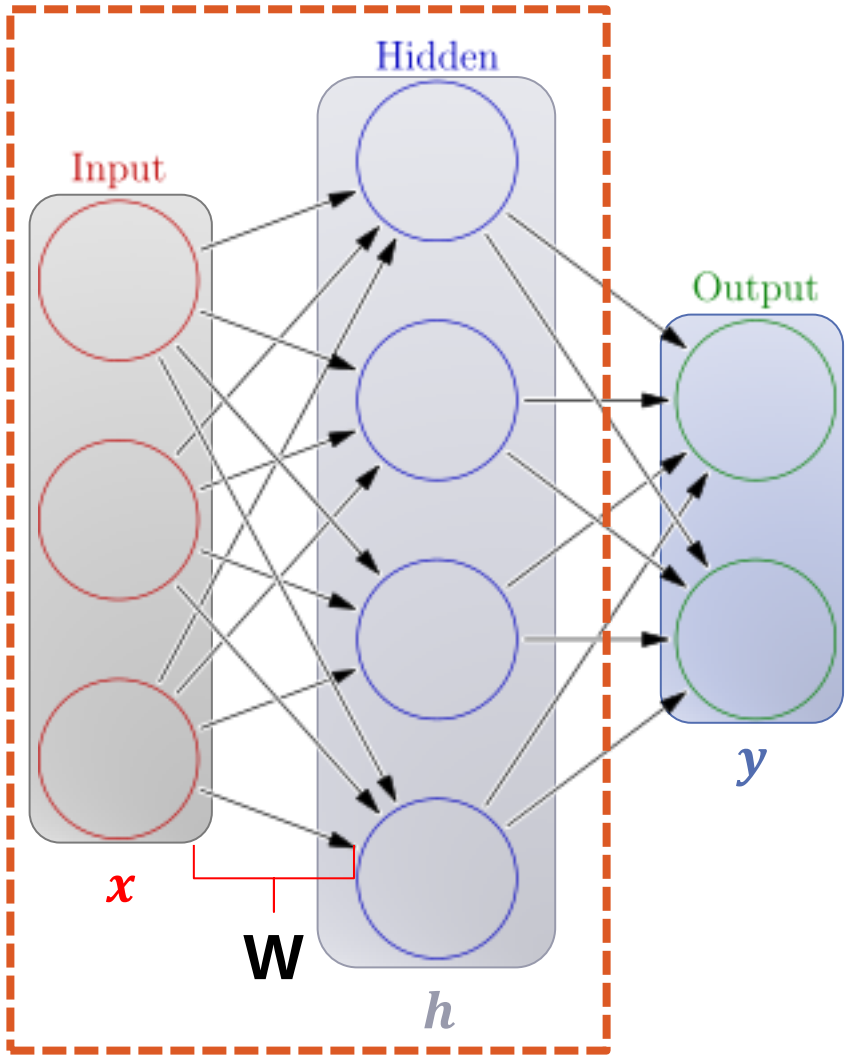
$a = f(b, c)$

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

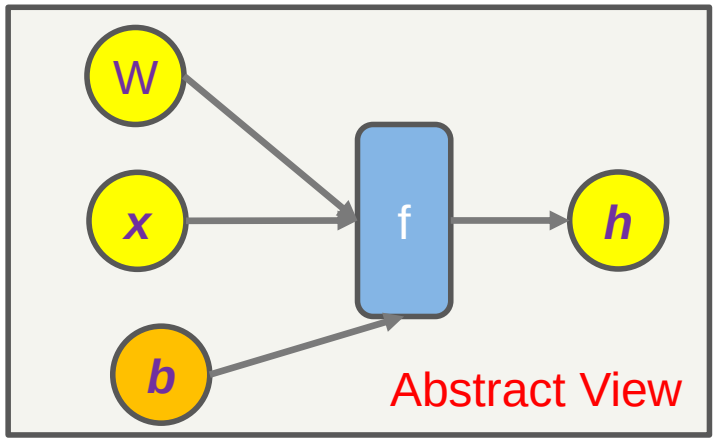
$u = h(x) \quad v = g(u) \quad y = f(v)$

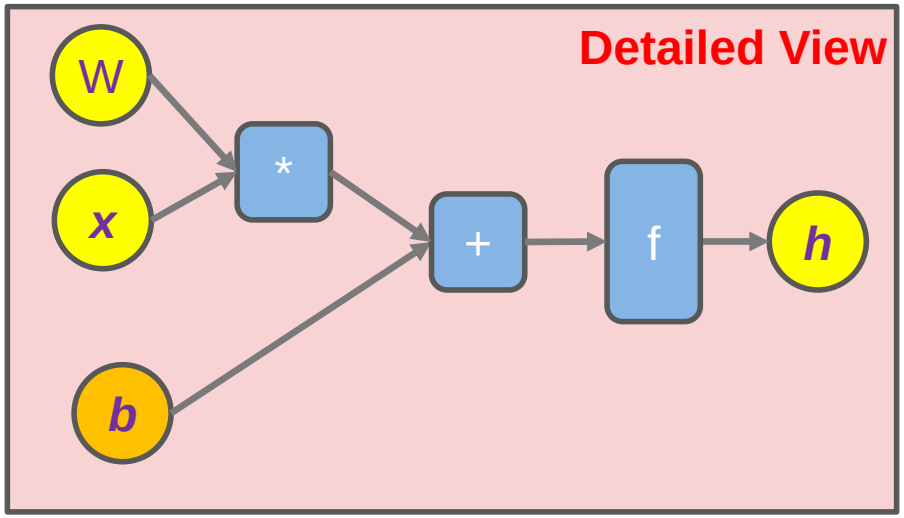


Modular Operations: Computation Graph Definition



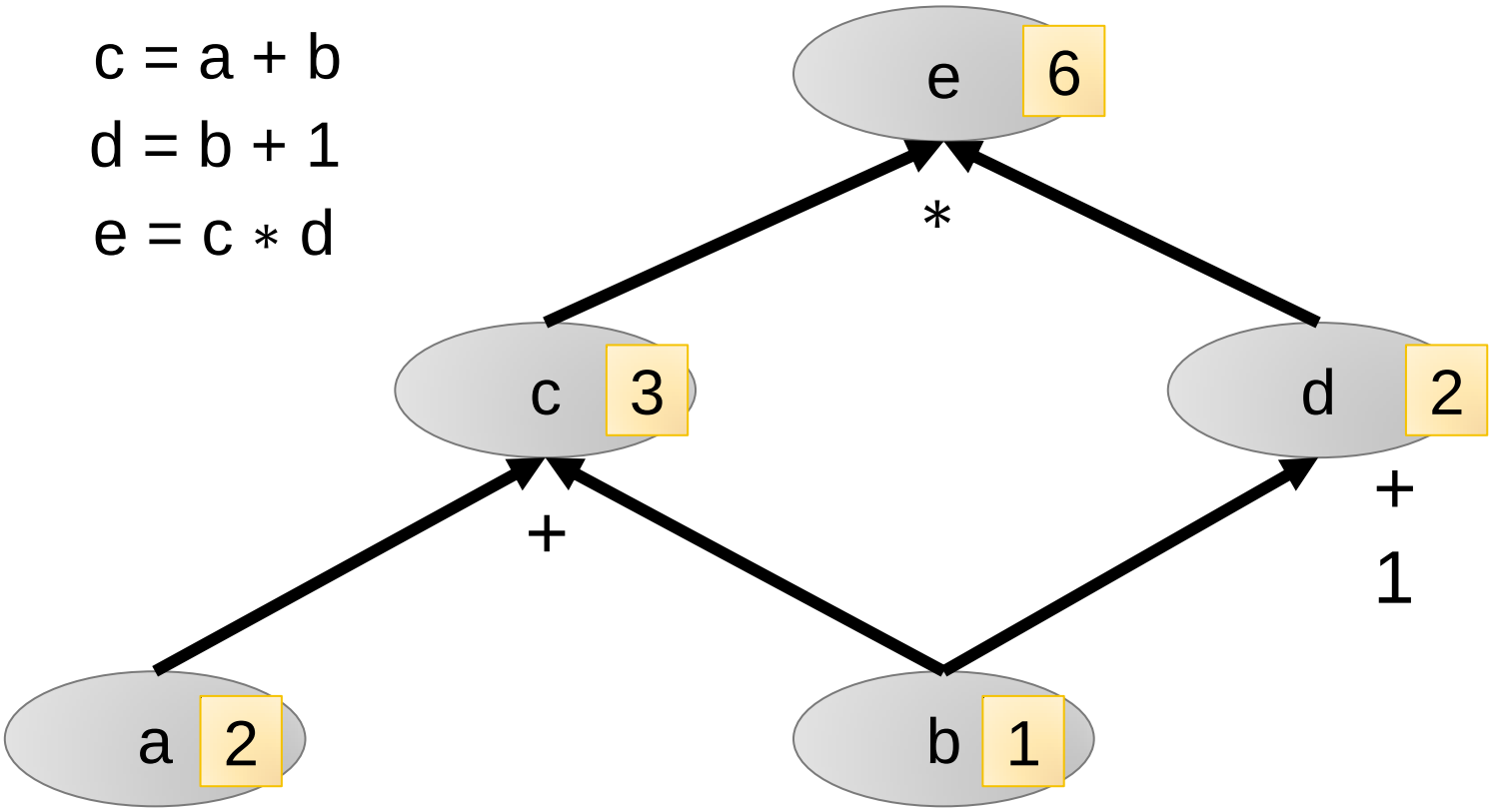
$$h = f(Wx + b)$$





Computational Graph

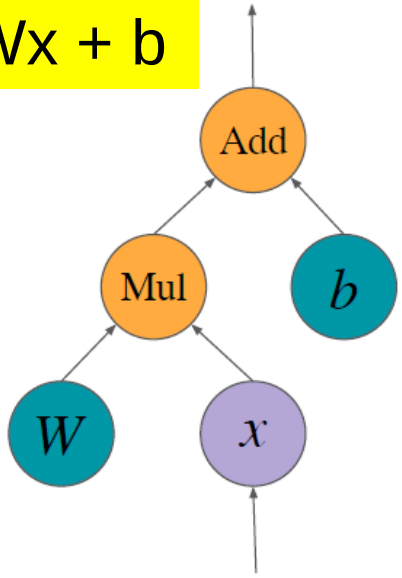
- Example: e = (a+b) ∗ (b+1)



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.

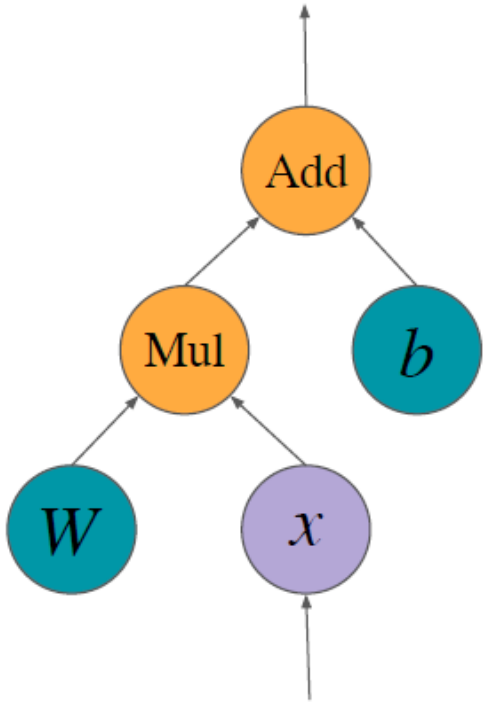
y = Wx + b



- TensorFlow = Tensor + Flow = Data + Flow

Computation Graph

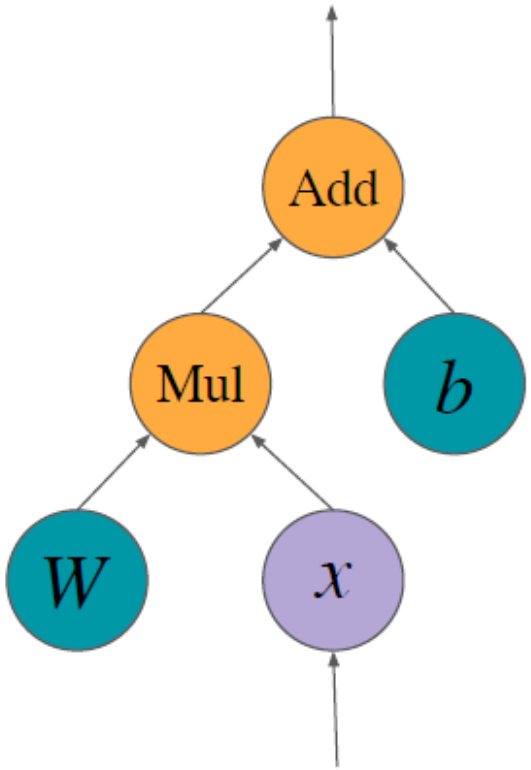
- **Constants**
  - fixed value tensors - not trainable
- **Variables**
  - tensors initialized in a session - trainable
- **Placeholders**
  - tensors of values that are unknown during the graph construction
  - passed as input during a session
- **Ops**
  - functions on tensors



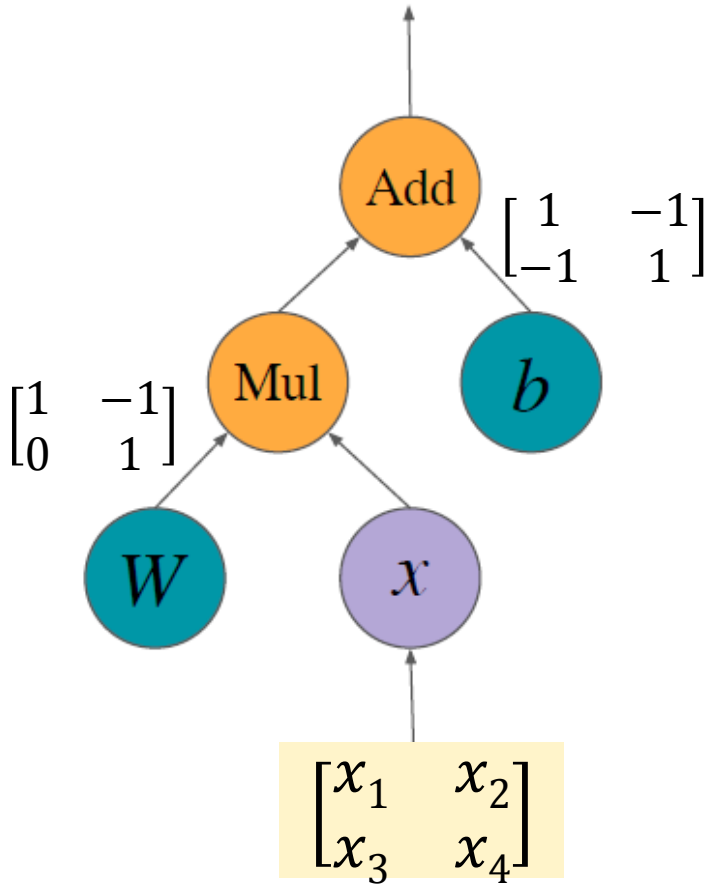
Computation Graph Structure

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

variable ops placeholder variable



Example

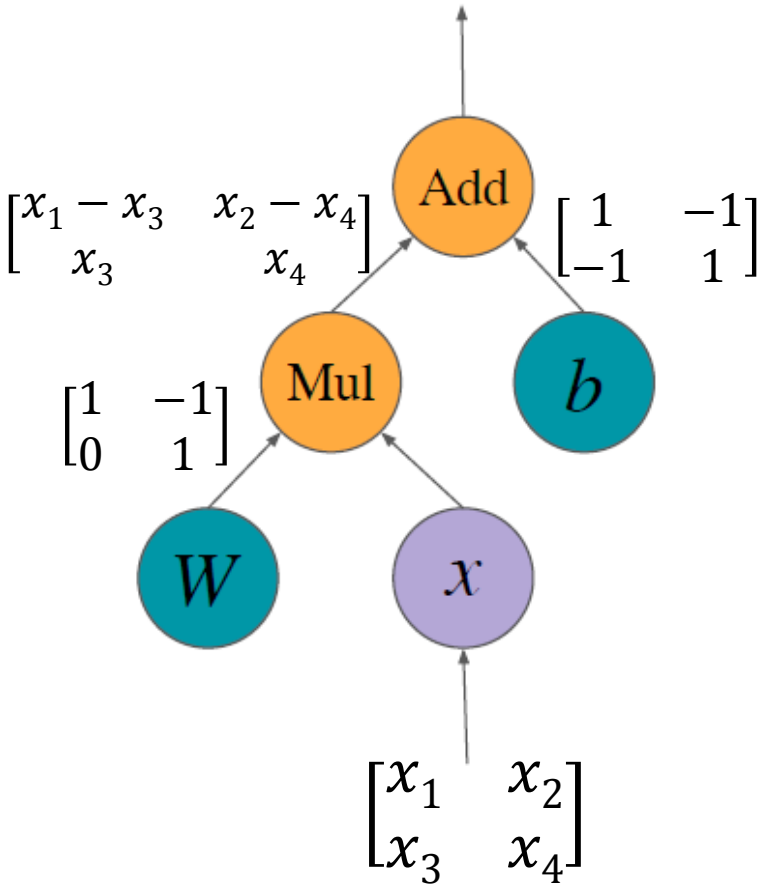


y = Wx + b

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

W x b

Evaluating Mult



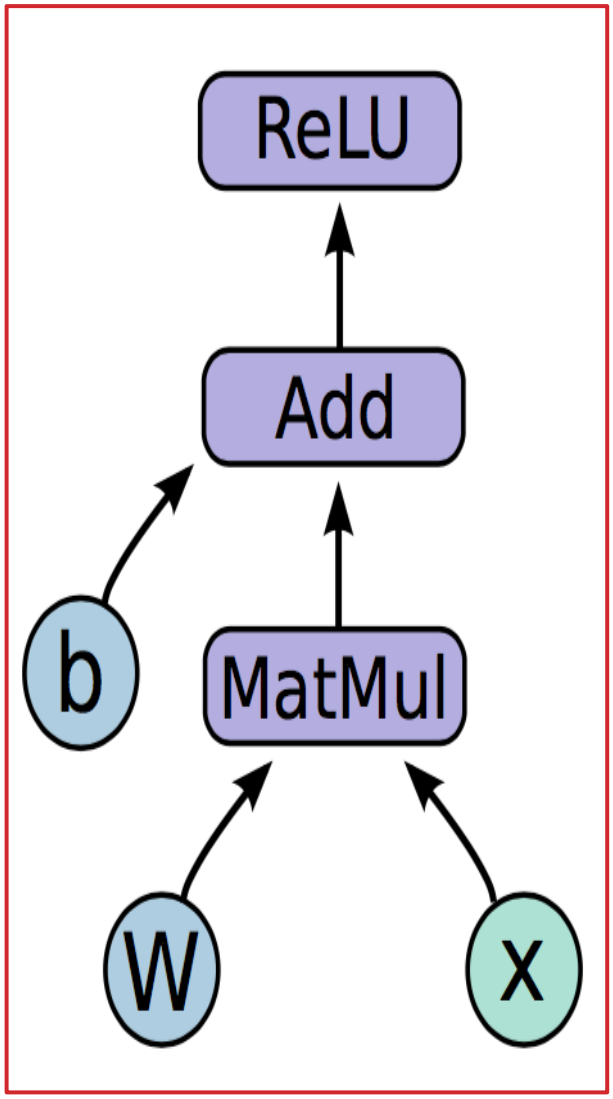
y = Wx + b

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

Wx b

Programming Model: Computation Graph

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



Programming Model : Steps for DL Implementation

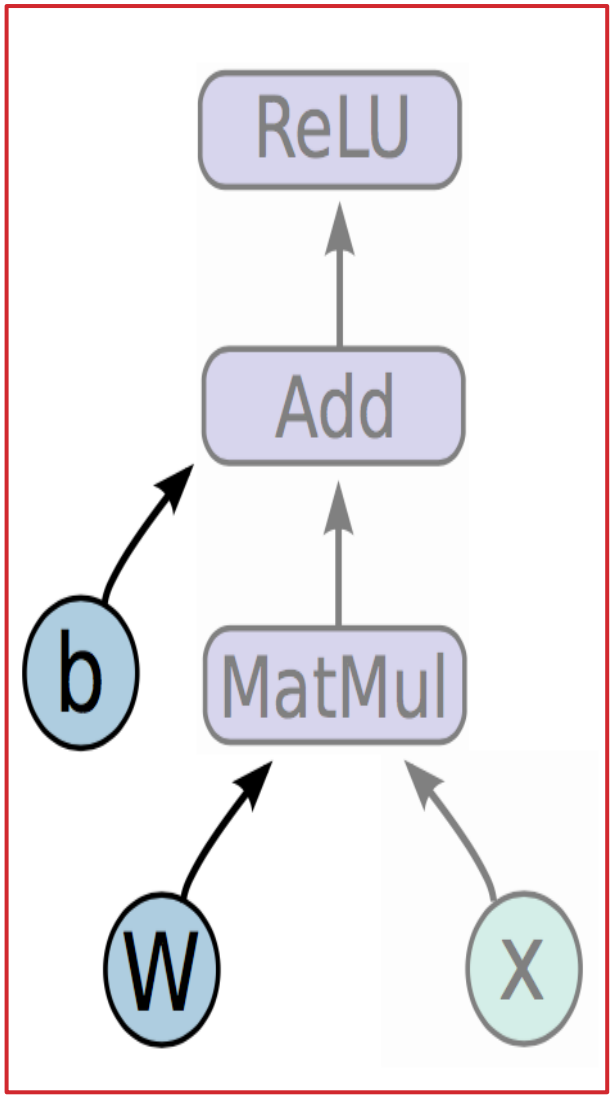
1. Define the Computation Graph
2. Initialize your data
3. Assign to hardware (CPU, GPU, etc.)

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)

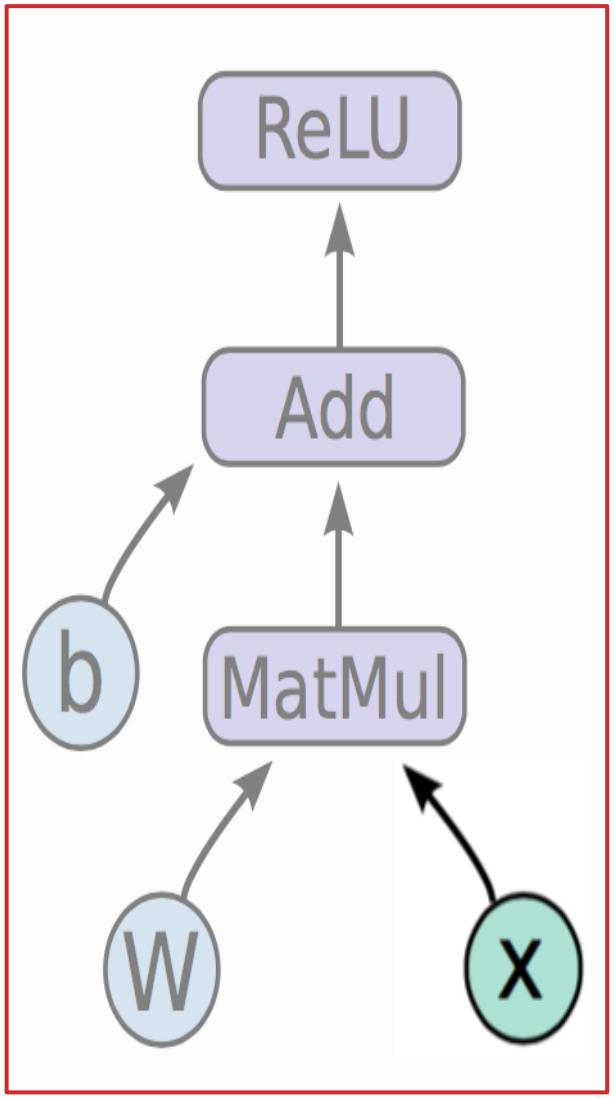


Step 1: Computation Graph

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

Placeholders are nodes whose value is fed in at execution time

(inputs, labels, …)



Step 1: Computation Graph

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

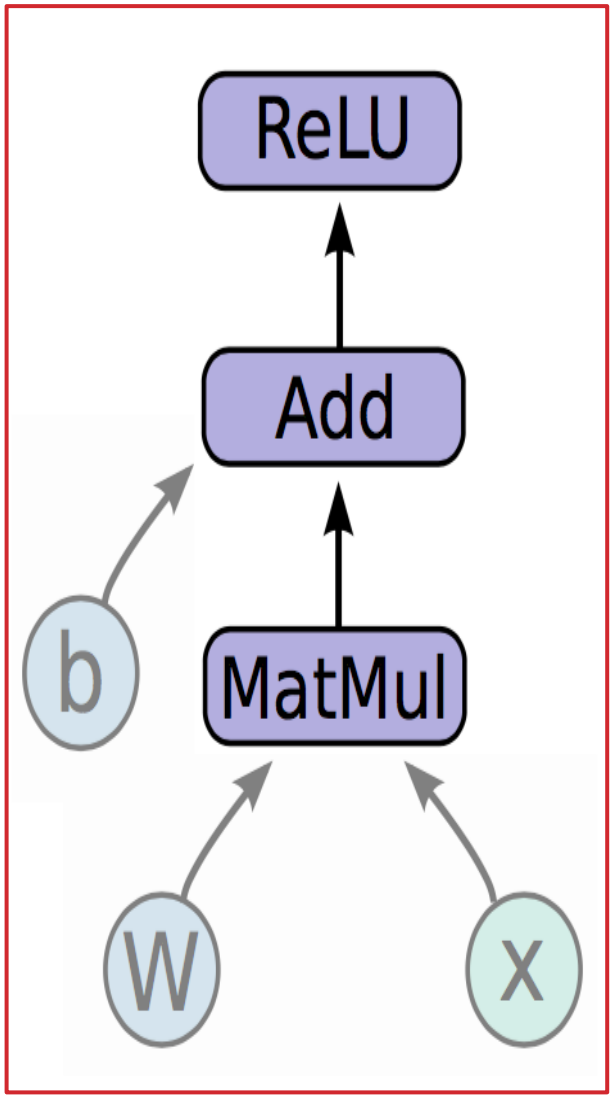
Mathematical operations:

MatMul: Multiply two matrices

Add: Add elementwise

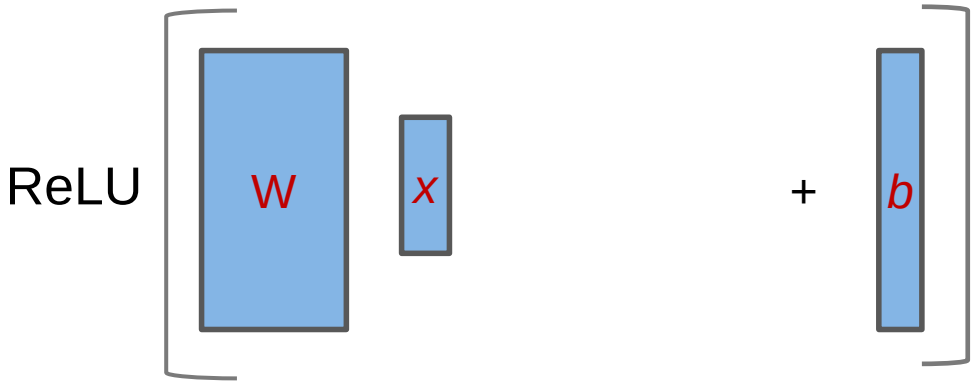
ReLU: Activate with elementwise rectified linear function

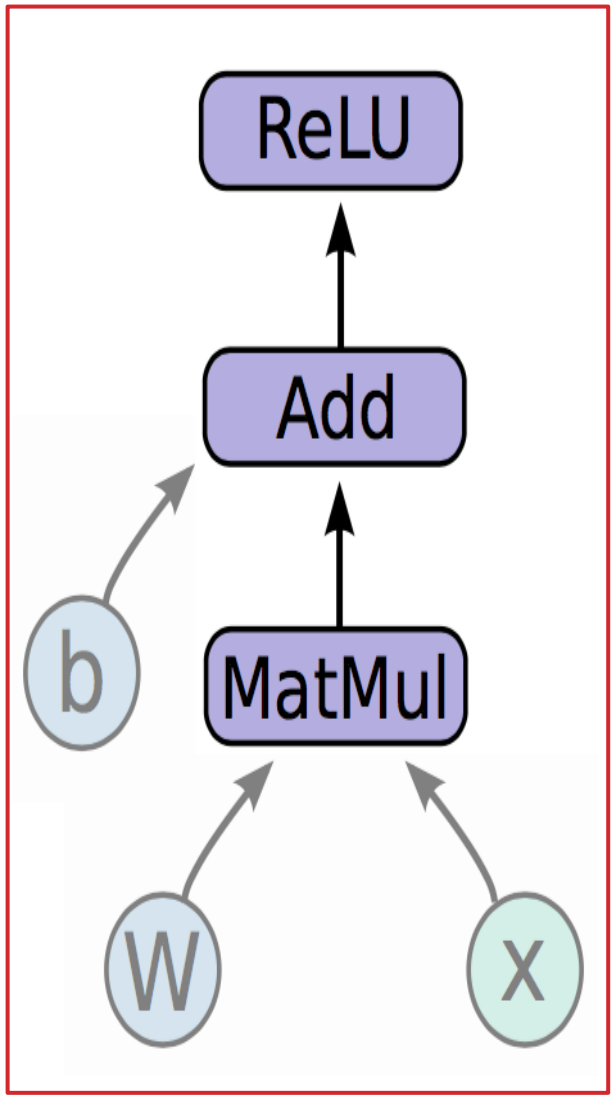
$$ReLu(x) = \begin{cases} 0, x \leq 0 \\ x, x > 0 \end{cases}$$



Step 2: Code and Variable Assignment

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



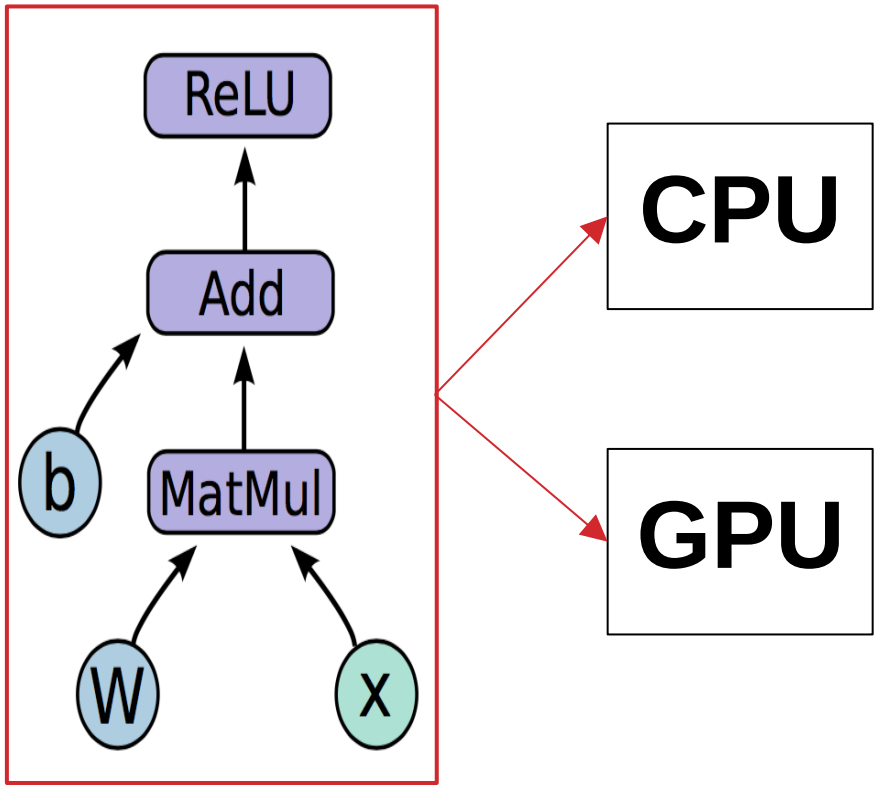



```
import tensorflow as tf

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**

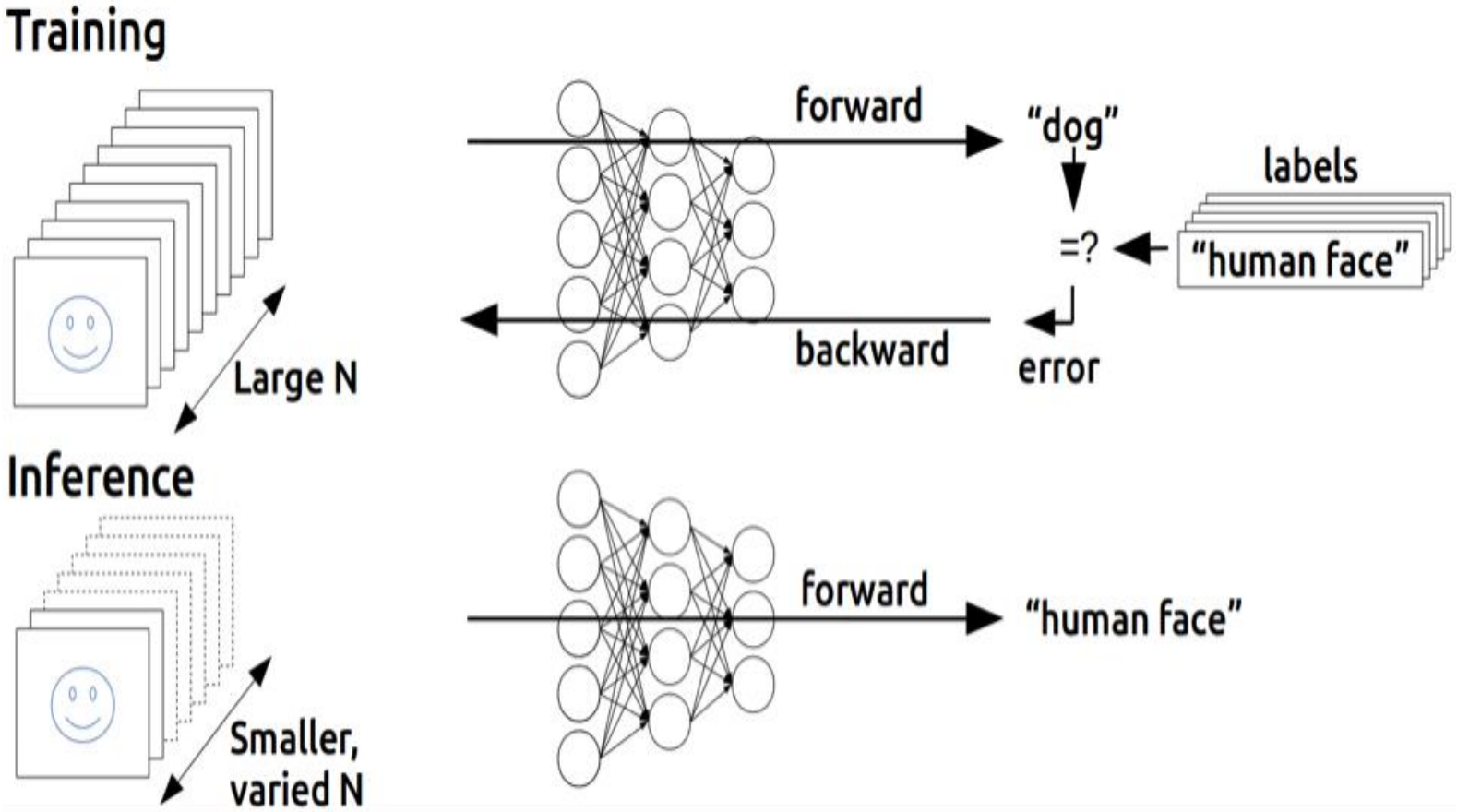
- Next: train model
  - Define loss function
  - Compute gradients

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

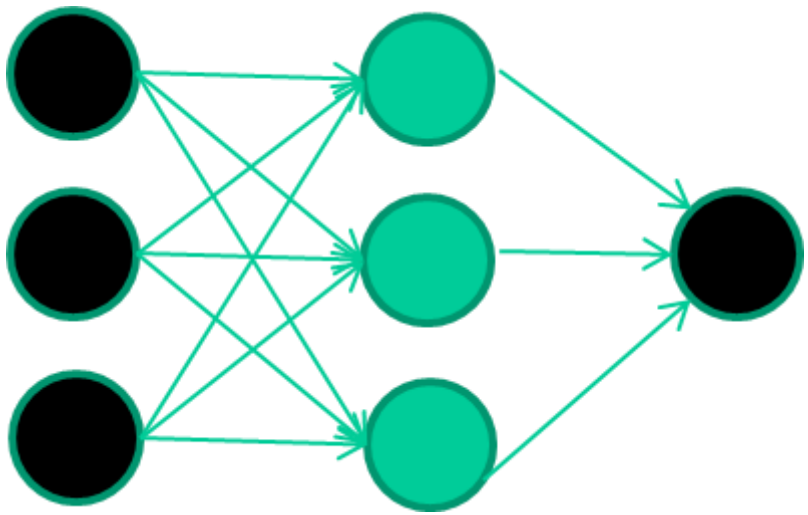


From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/

Example: Backpropagation

A dataset

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

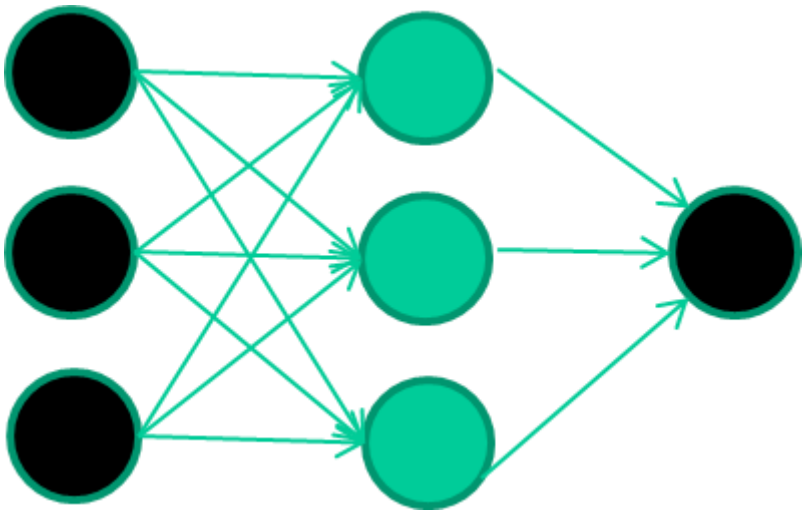


Example: Backpropagation

Training the neural network

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …



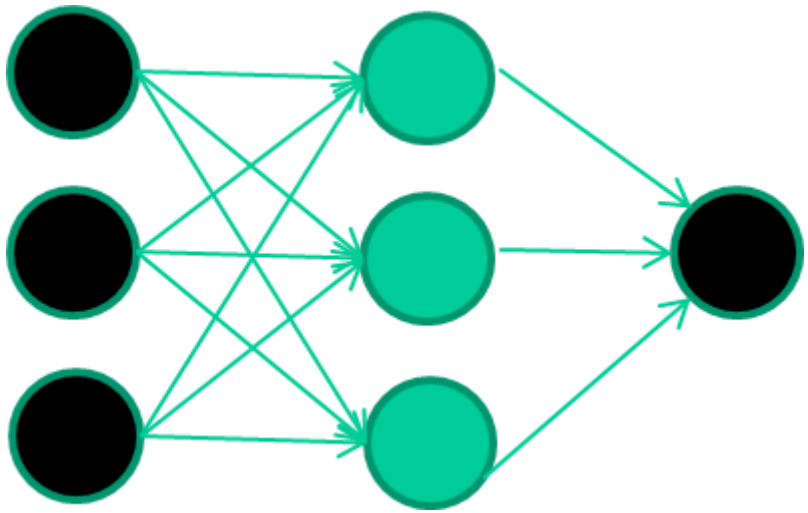
Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Initialise with random weights

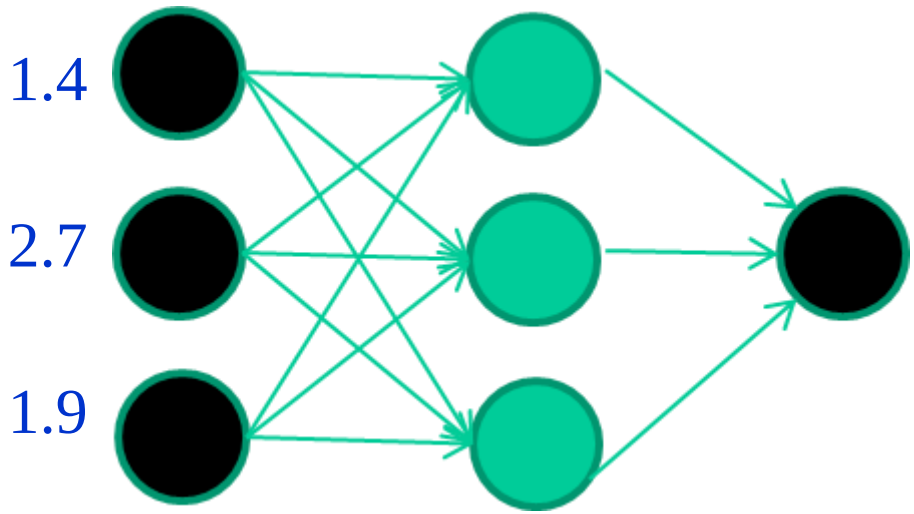


Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Present a training pattern

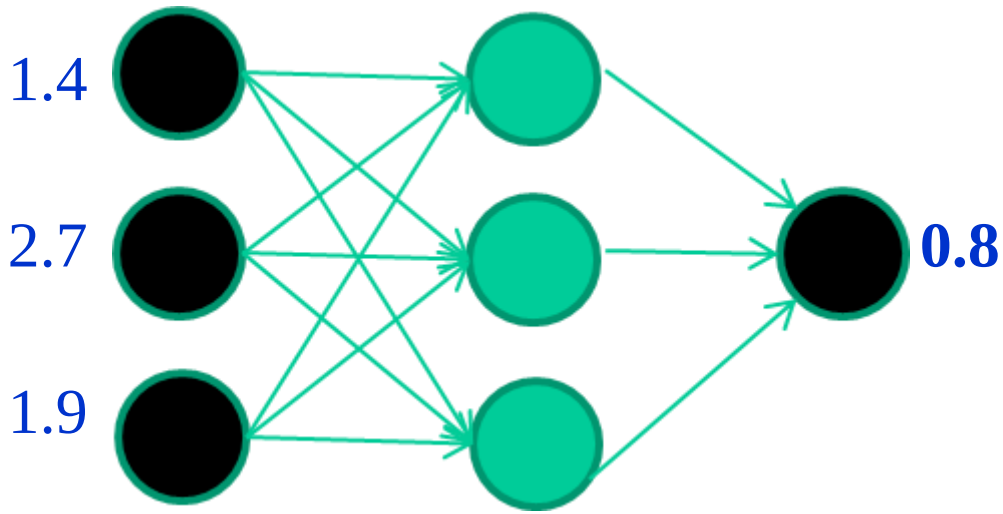


Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Feed it through to get output

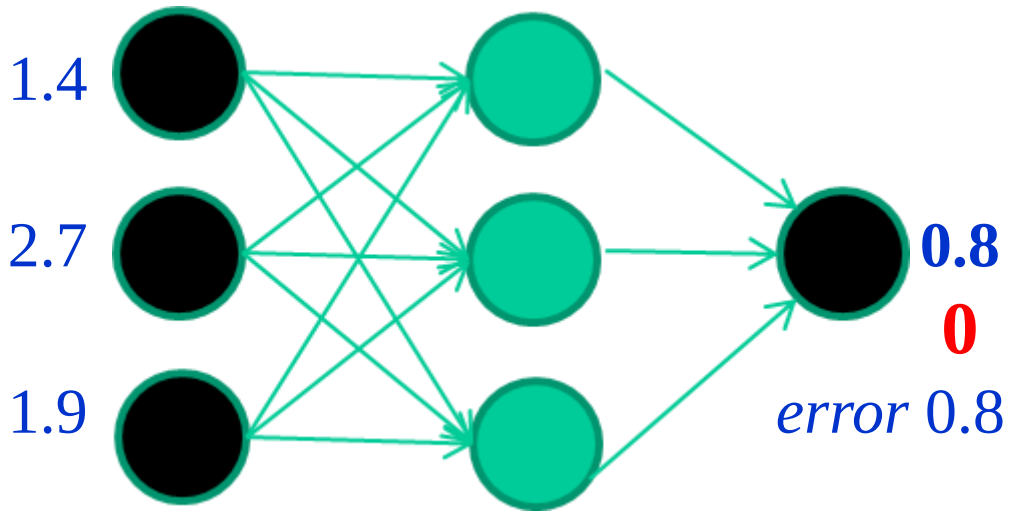


Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Compare with target output

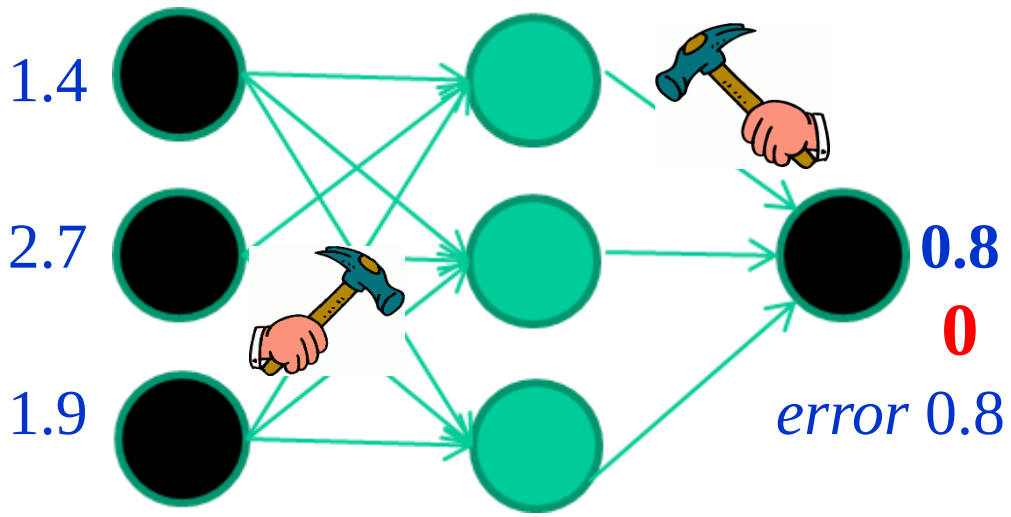


Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Adjust weights based on error



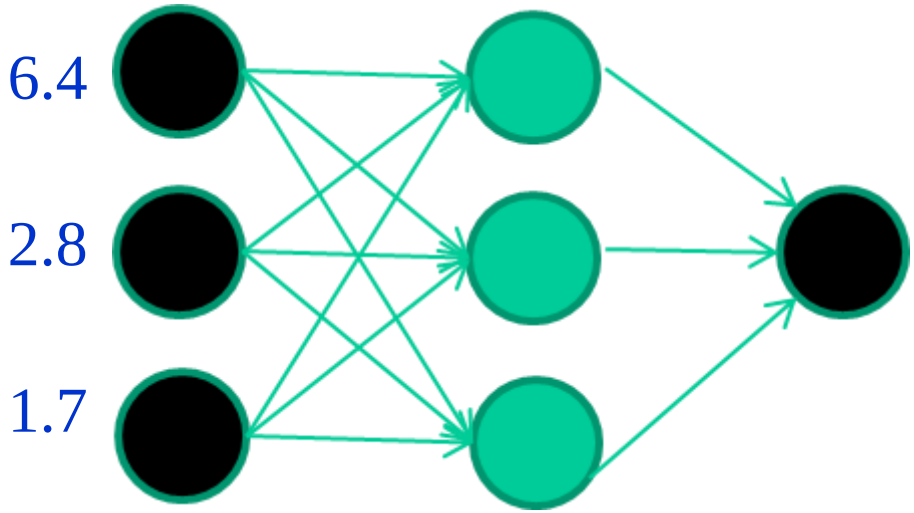
Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Present a training pattern



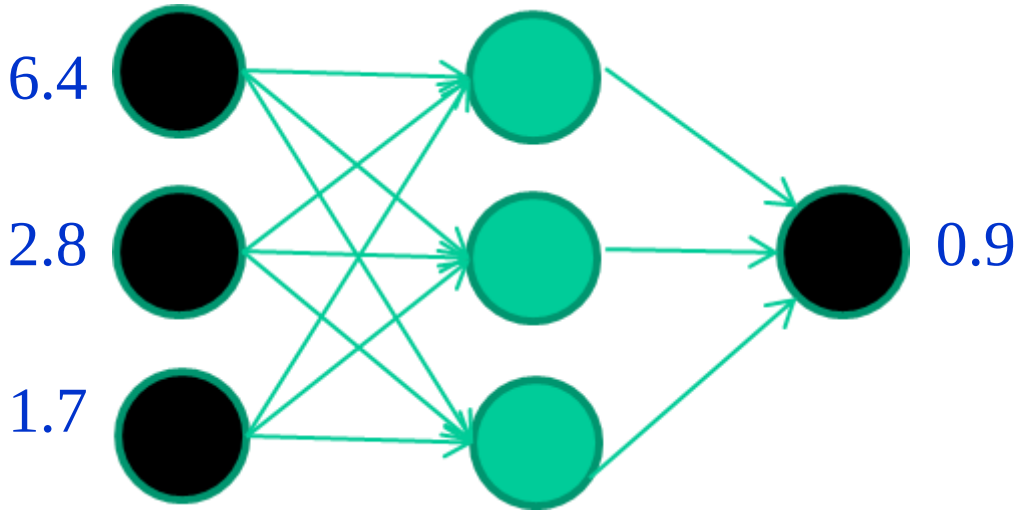
Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Feed it through to get output



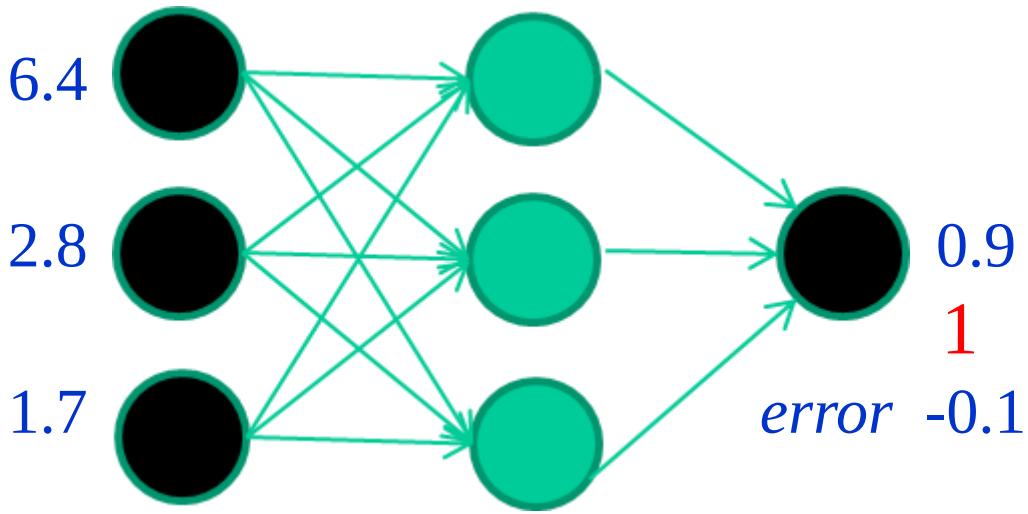
Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Compare with target output



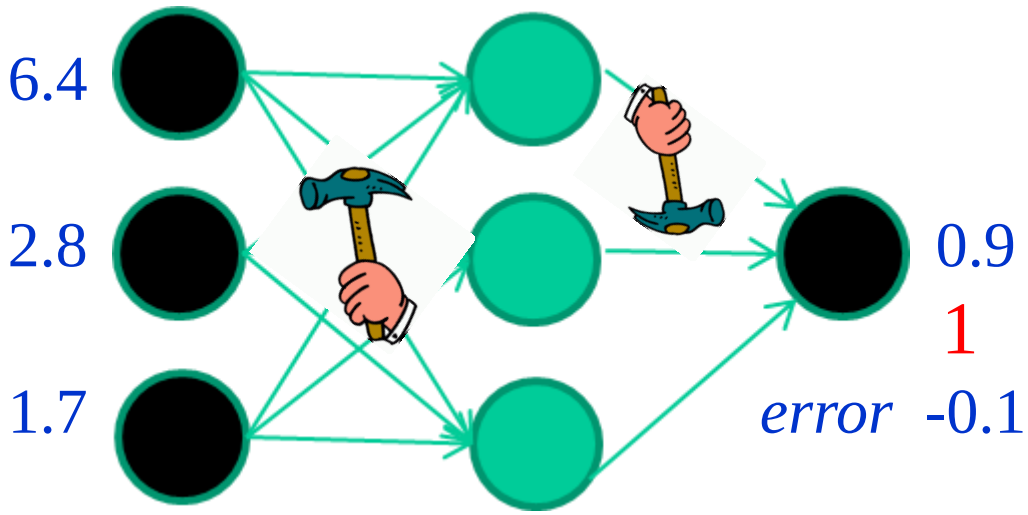
Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Adjust weights based on error

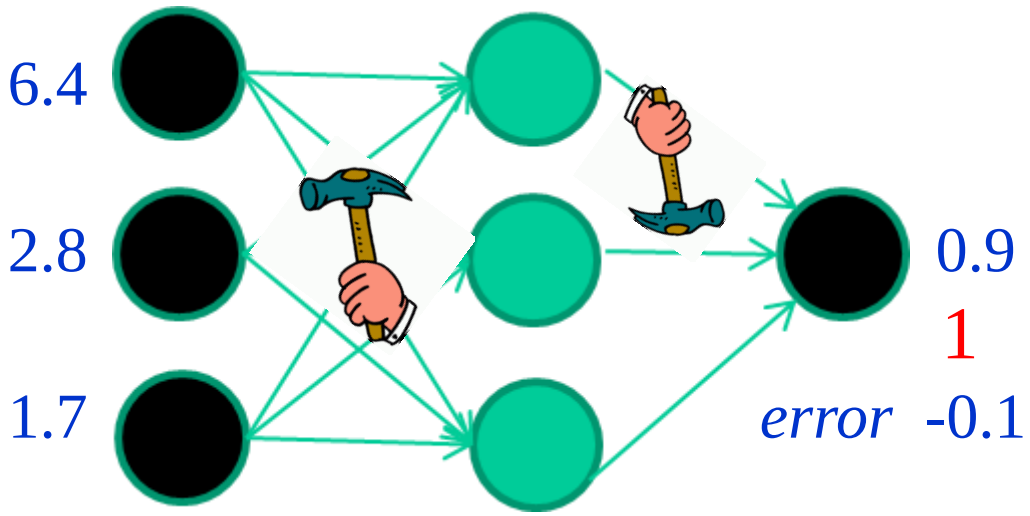


Example: Backpropagation

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

And so on ….



Repeat this thousands, maybe millions of times – each time taking a random training instance, and making slight weight adjustments

Algorithms for weight adjustment are designed to make changes that will reduce the error

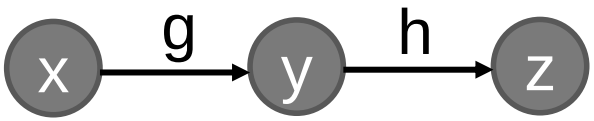
Backpropagation and Gradients

- 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

Case 1

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

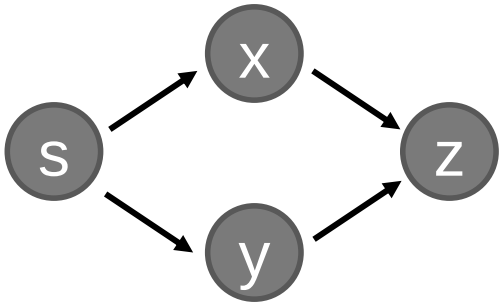


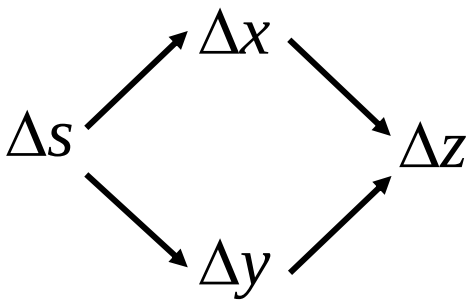
$$\Delta x \to \Delta y \to \Delta z$$

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

Case 2

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





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

Computational Graph

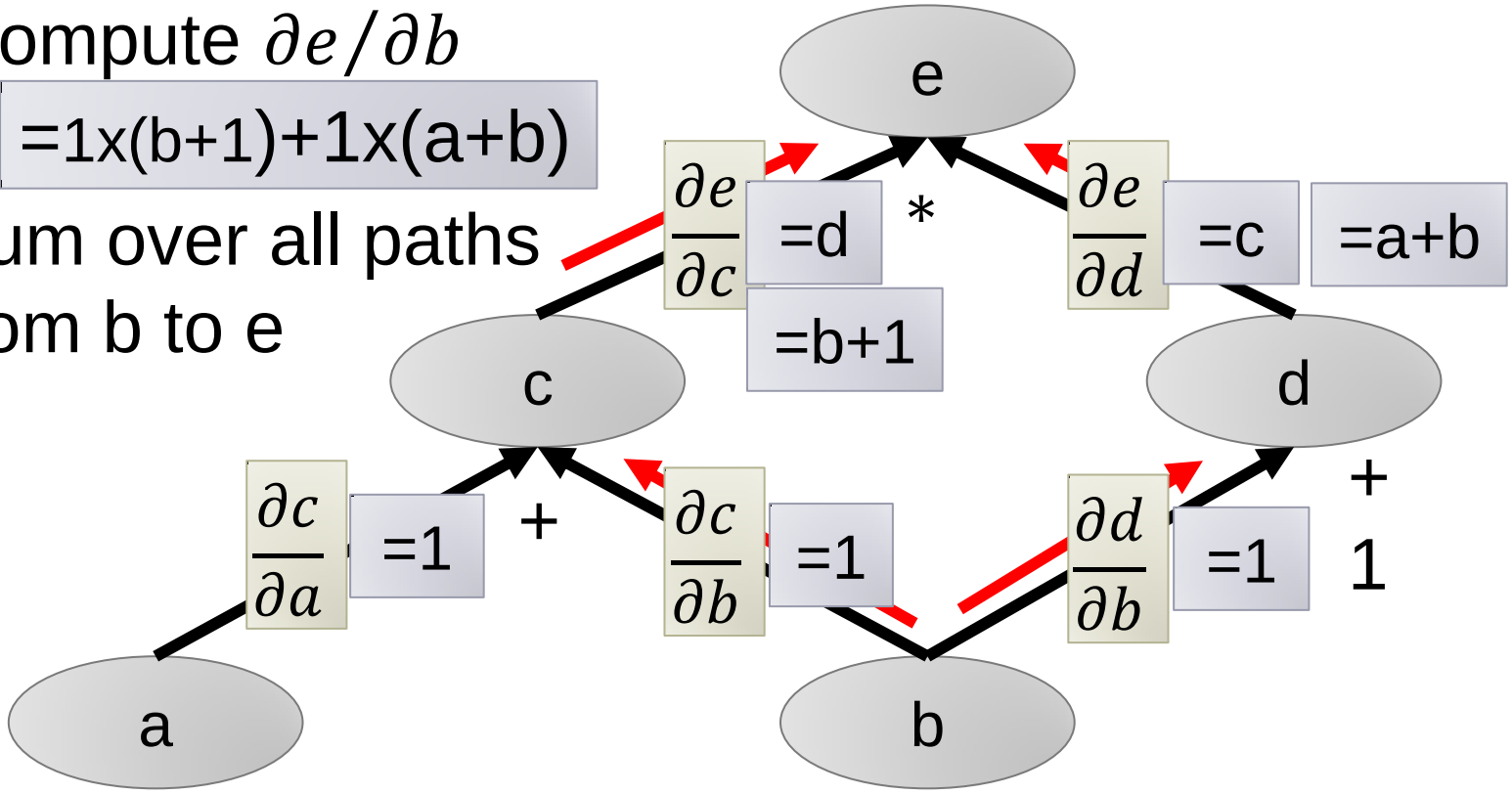
- Example: e = (a+b) ∗ (b+1)

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

Compute $\partial e/\partial b$

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

Sum over all paths from b to e



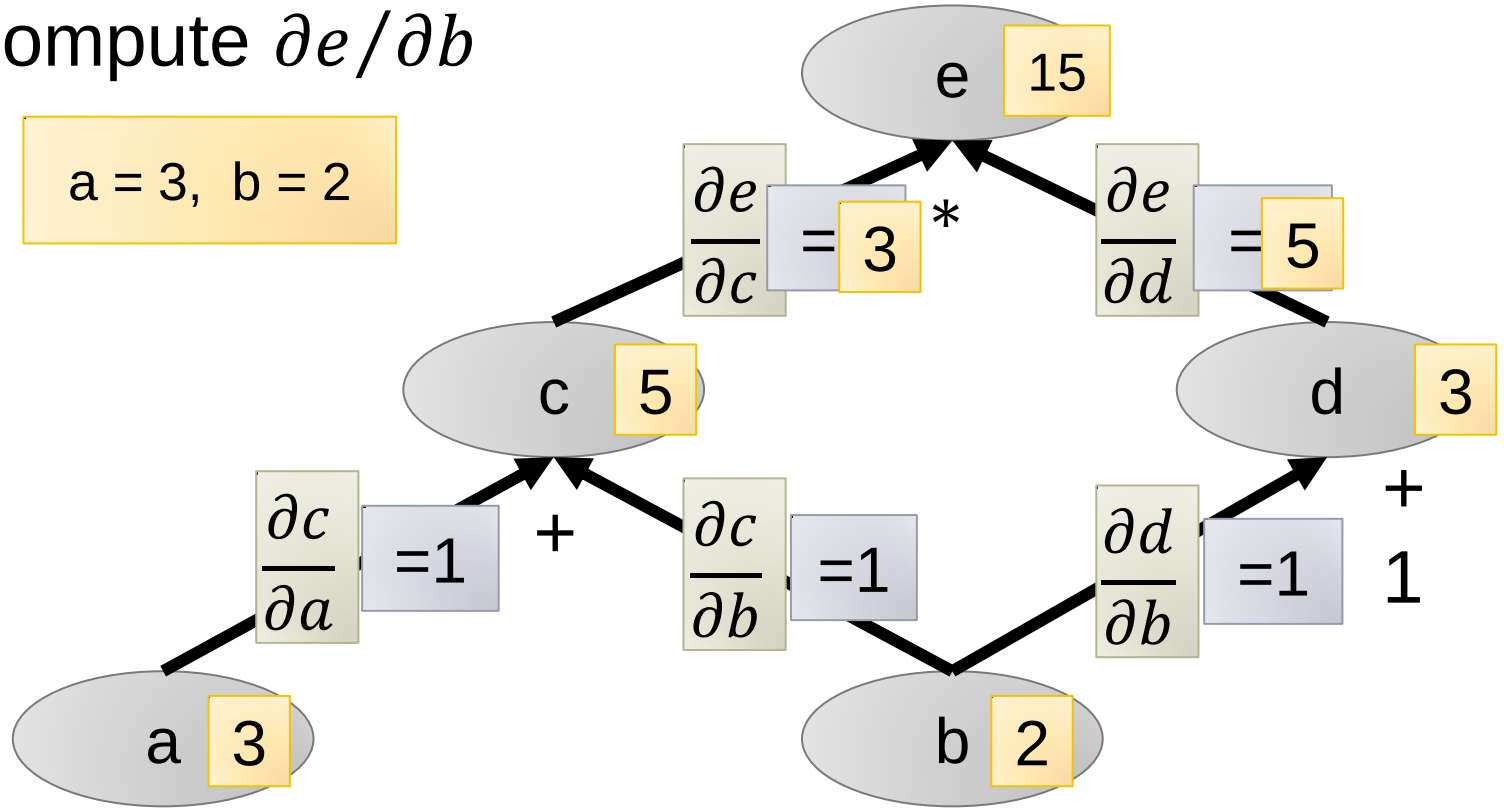
Computational Graph

- Example: e = (a+b) ∗ (b+1)

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

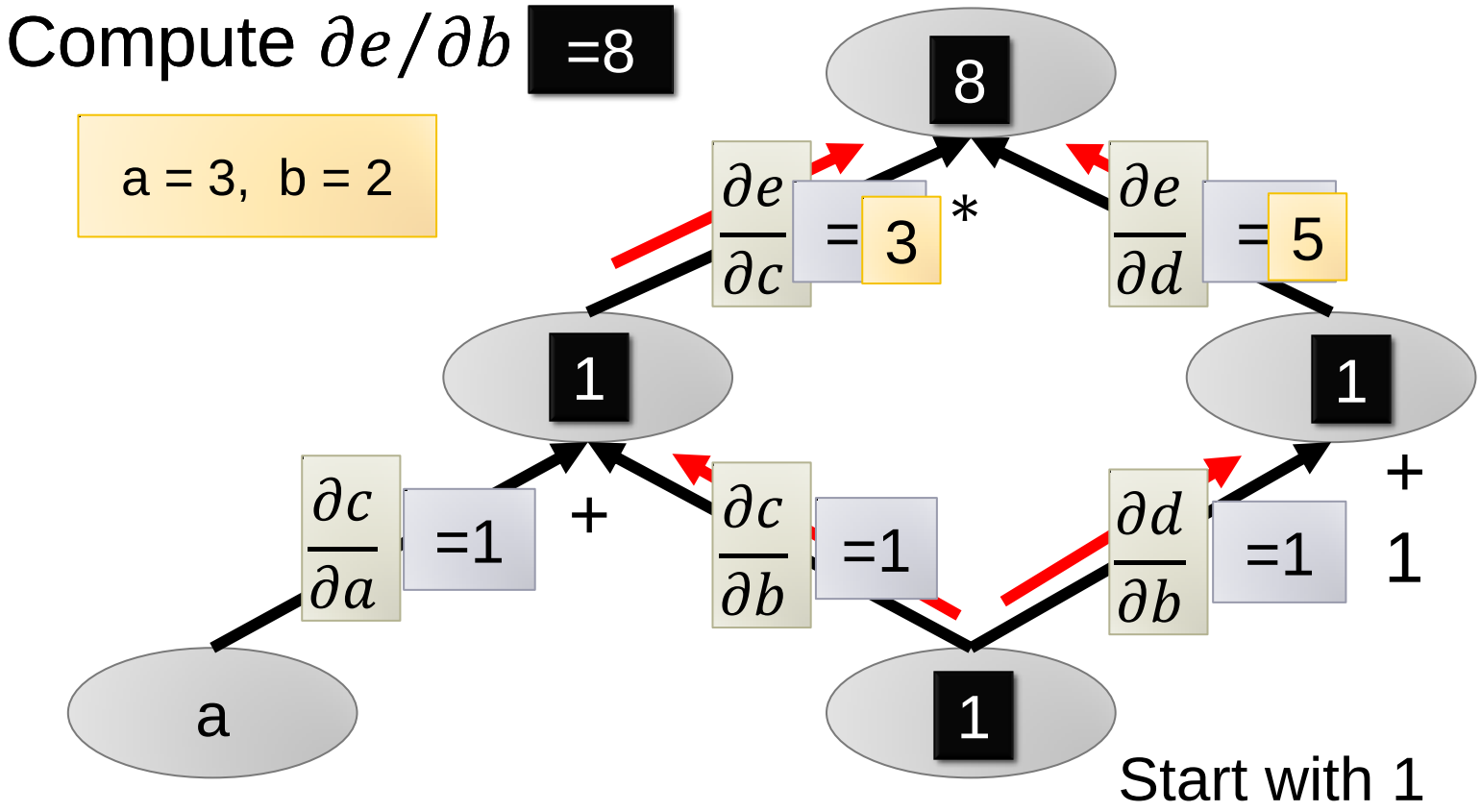
Compute $\partial e / \partial b$

a = 3, b = 2



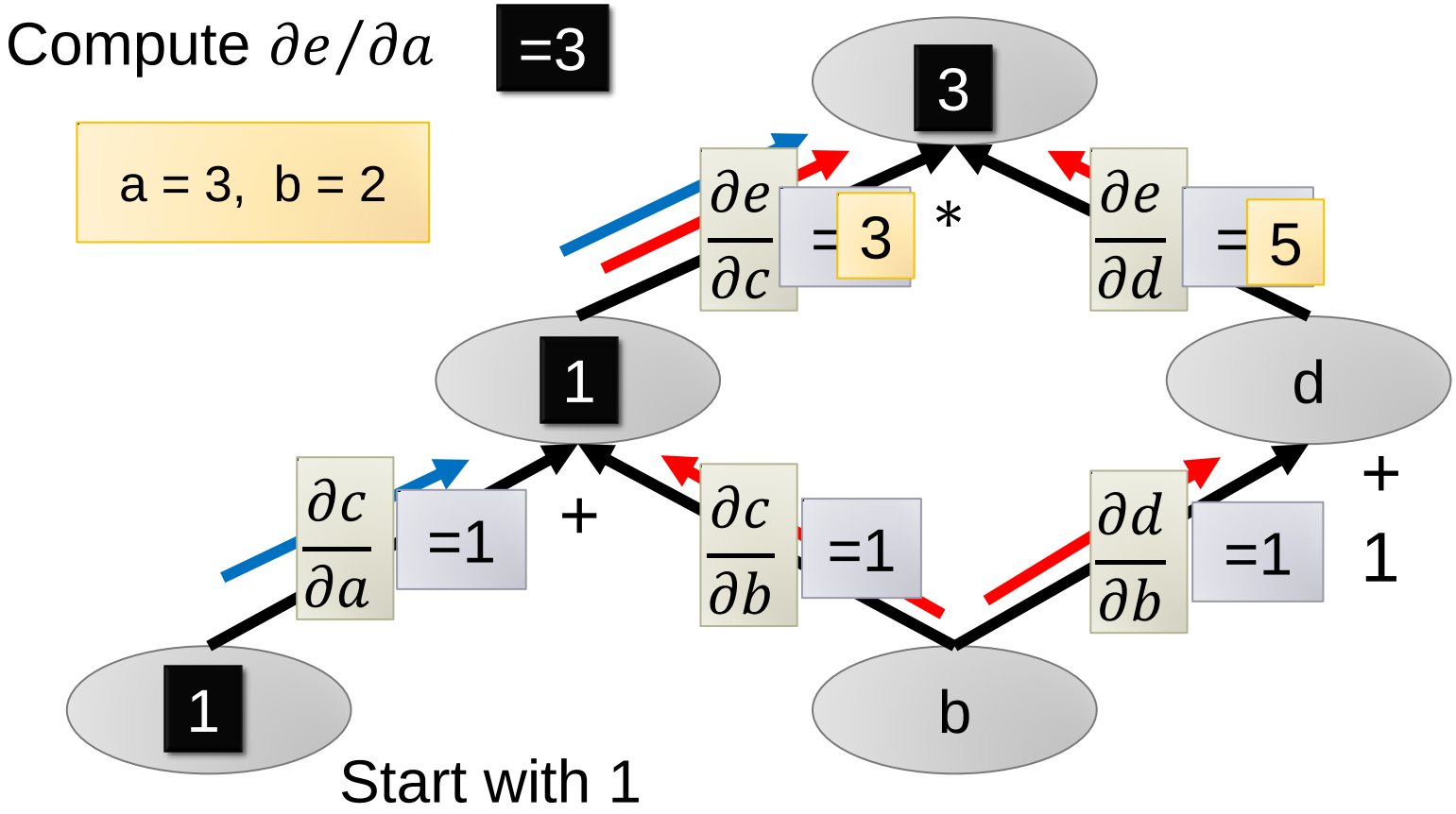
Computational Graph

- Example: e = (a+b) ∗ (b+1)



Computational Graph

- Example: e = (a+b) ∗ (b+1)

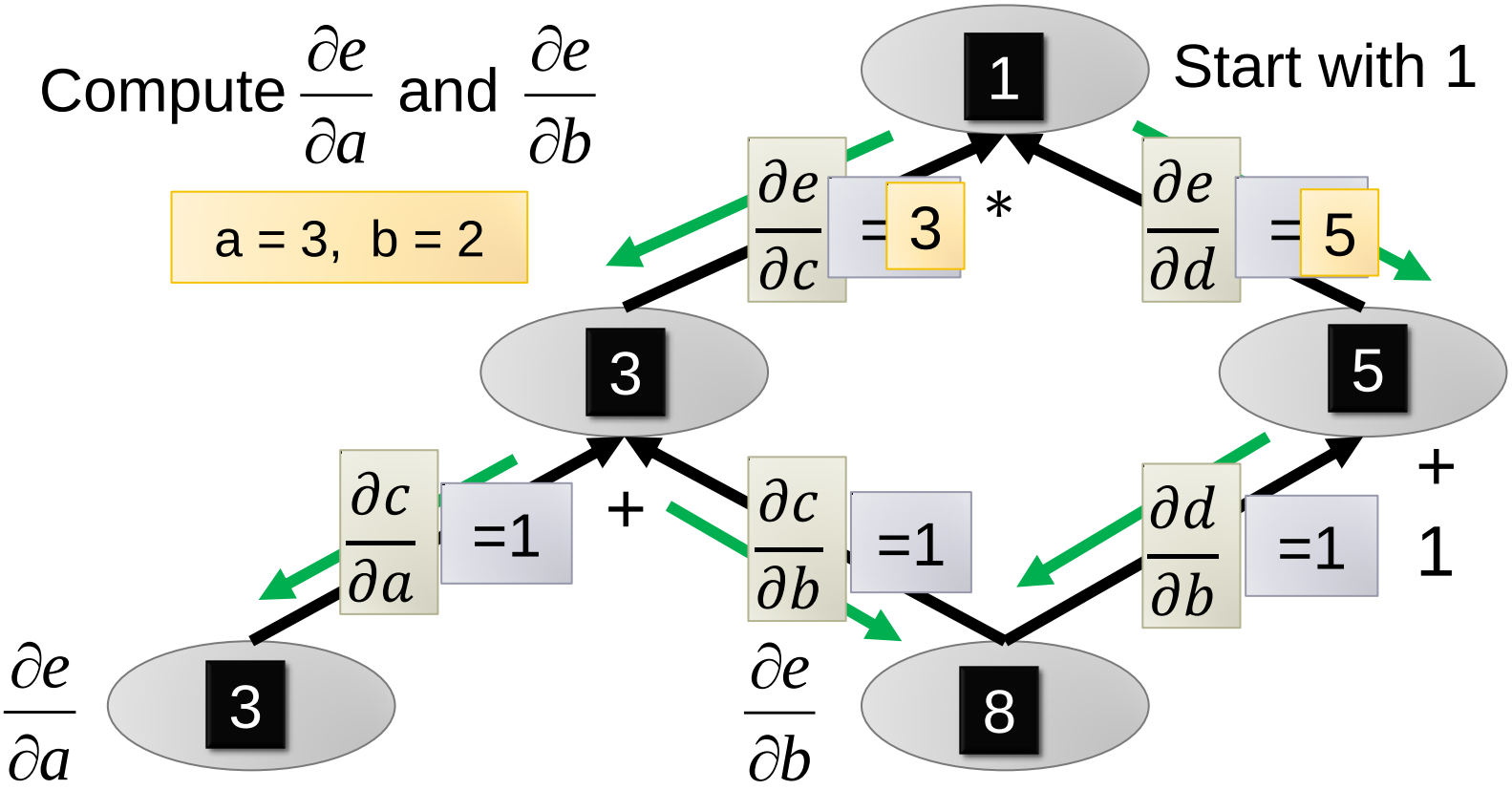


Computational Graph

- Example: e = (a+b) ∗ (b+1)

Reverse mode

What is the benefit?



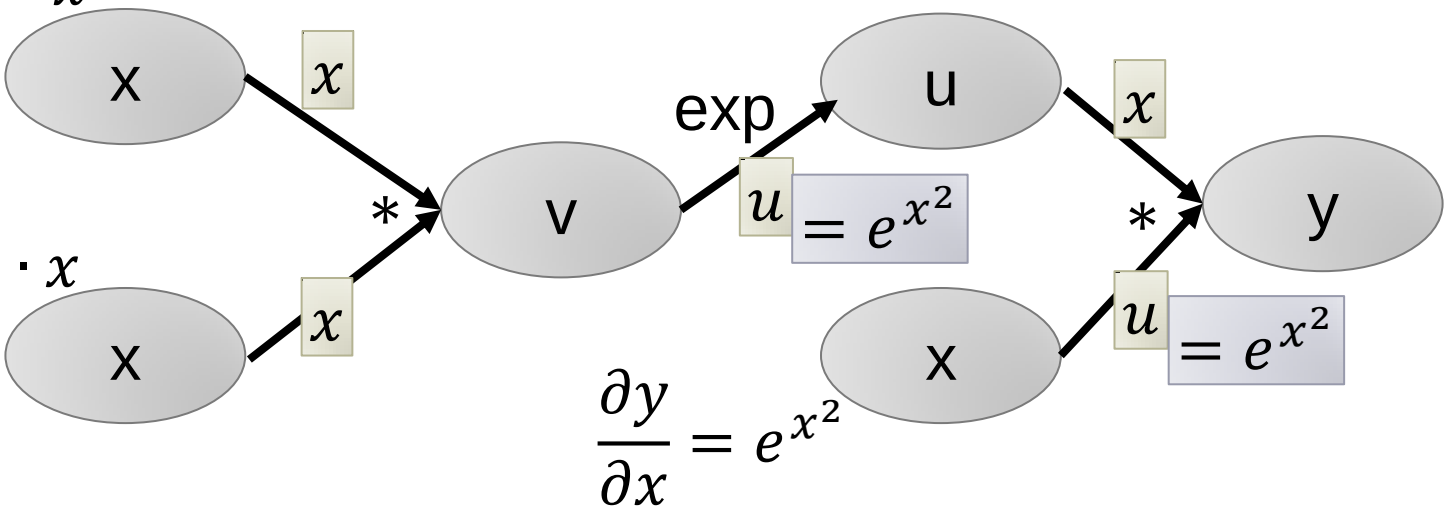
Computational Graph

- ***Parameter sharing***: the same parameters appearing in different nodes

$$y = xe^{x^2} \qquad \frac{\partial y}{\partial x} = ? \quad e^{x^2} + x \cdot e^{x^2} \cdot 2x$$

$$\frac{\partial y}{\partial x} = x \cdot e^{x^2} \cdot x$$

$$\frac{\partial y}{\partial x} = x \cdot e^{x^2} \cdot x$$



$$\frac{\partial y}{\partial x} = e^{x^2}$$