Machine Learning for Finance

UCEMA 2020

Lecture 3

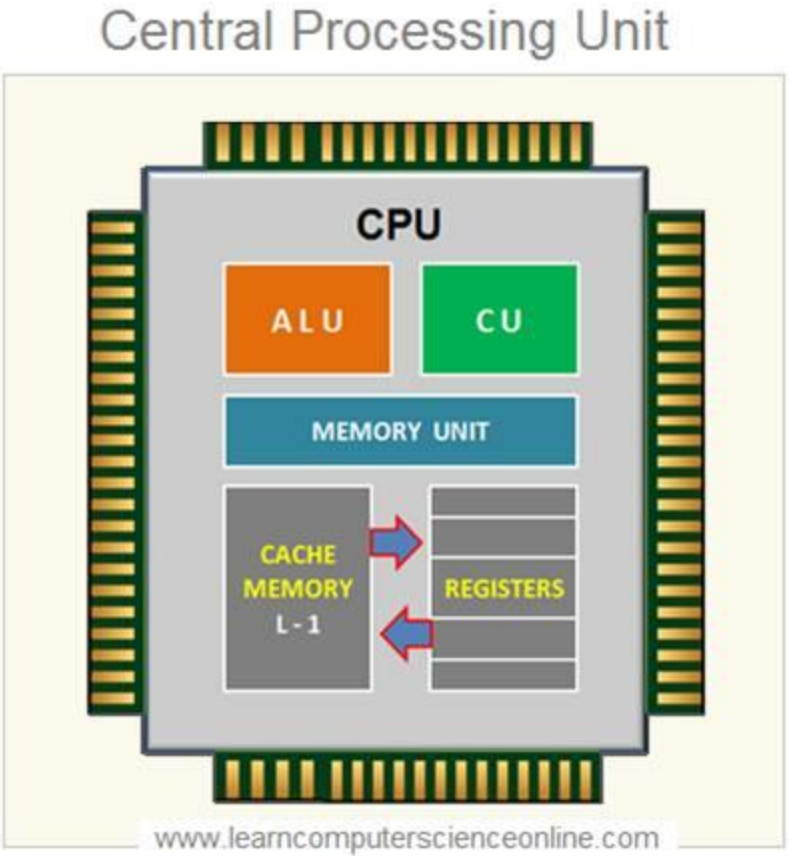
- TensorFlow(1)
- Data Flow Graph and Lazy execution
- Automatic Differentiation
- Automatic Differentiation in Quant Finance
- First TensorFlow Demo
- Quant Finance Demo in TensorFlow ()
- artificial neural networks
- Multi Layer Perceptron and Deep Learning
- Neural Networks TensorFlow Demo

Lectures Structure

- 1st block
  - ~1h: Presentation of subjects + code explanation.
  - ~15': Discussion + Q&A.
- 2nd block
  - ~1h: Presentation of subjects + code explanation.
  - ~15': Discussion + Q&A.

TensorFlow

TensorFlow

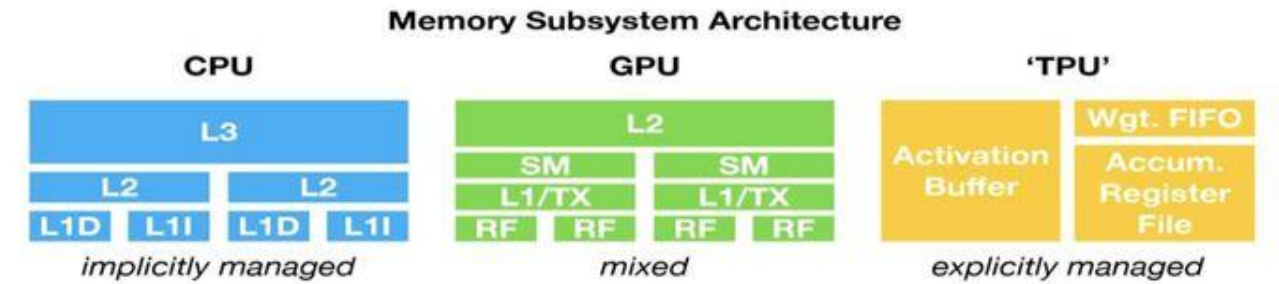

Central Processing Unit
CPU
ALU
CU
MEMORY UNIT
CACHE MEMORY L-1
REGISTERS
www.learncomputerscienceonline.com

TensorFlow

CPU vs GPU vs TPU

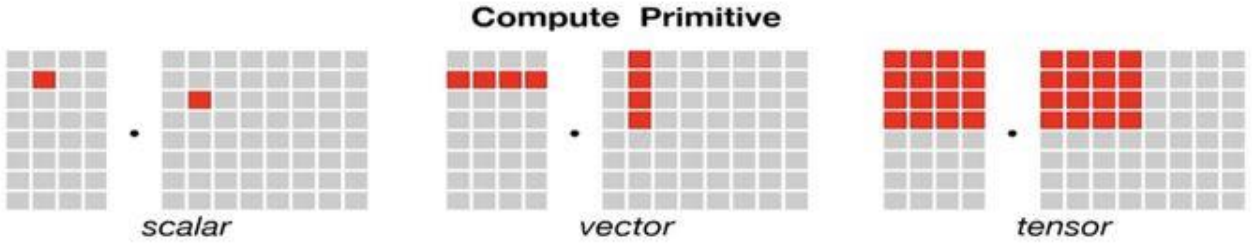
Memory Subsystem Architecture

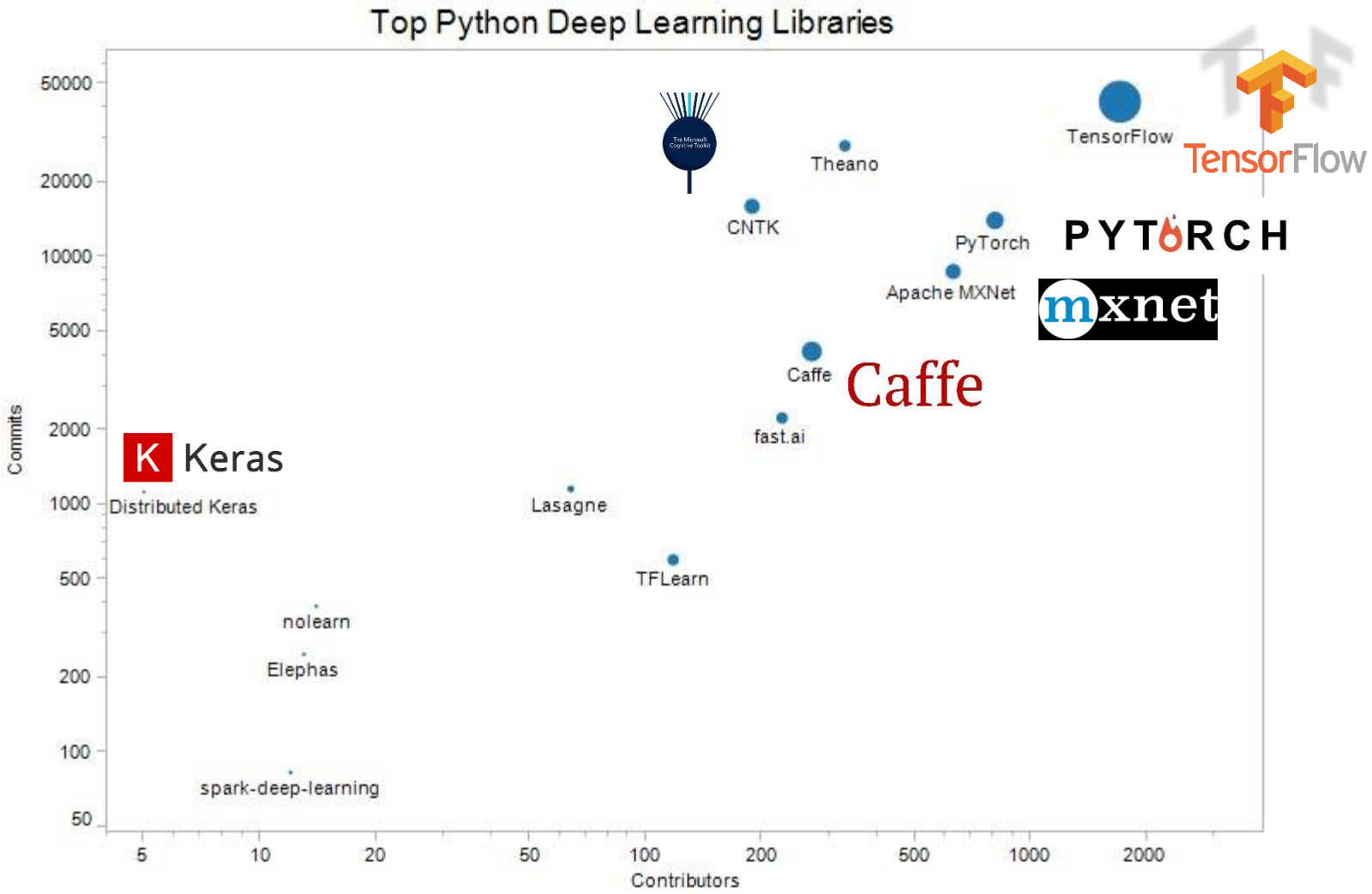
This image captures the memory architecture of CPU, GPU and TPU:



Compute Primitive

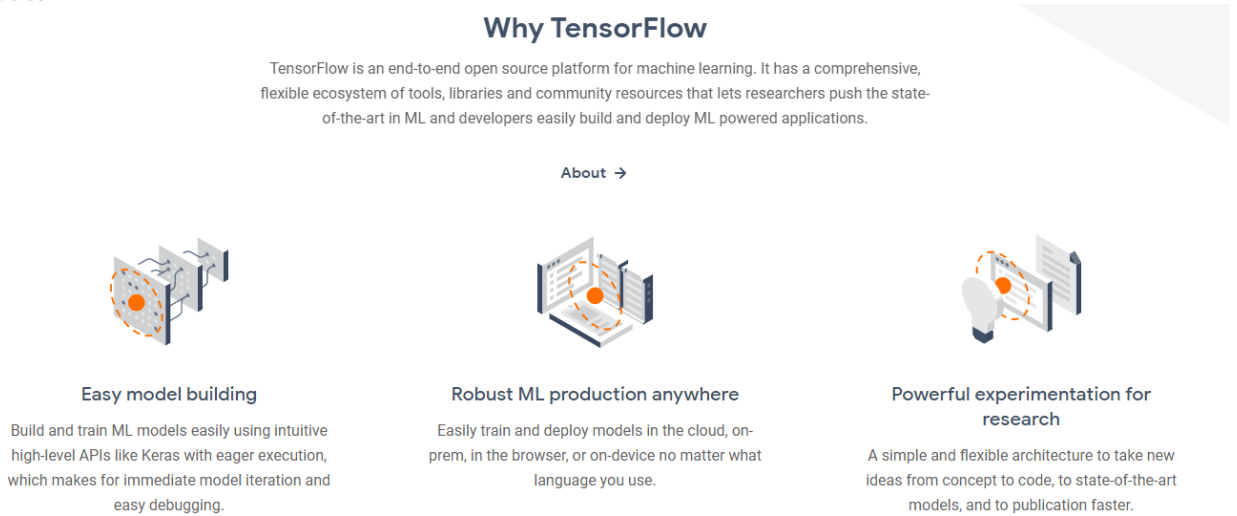
This image summarizes the compute primitive (smallest unit) in CPU, GPU and TPU:




Top Python Deep Learning Libraries
Commits
Contributors
50000
20000
10000
5000
2000
1000
500
200
100
50
5
10
20
50
100
200
500
1000
2000
TensorFlow
Theano
CNTK
PyTorch
Apache MXNet
Caffe
fast.ai
Keras
Distributed Keras
Lasagne
TFLearn
nolearn
Elephas
spark-deep-learning

Why TensorFlow

- Flexibility and clean design
- Automatic differentiation
- Great visualization via TensorBoard
- Good documentation
- Supported by Google and a wide community of developers
- Scalable and production-ready
- Portable to other operating systems and devices (e.g. on mobile devices)
- Many high-level APIs (TF.Learn, Keras, etc.)
- We will be using TF primarily for neural network models, but will start low, with simple demos and implementation of simple algorithms such as linear regression.

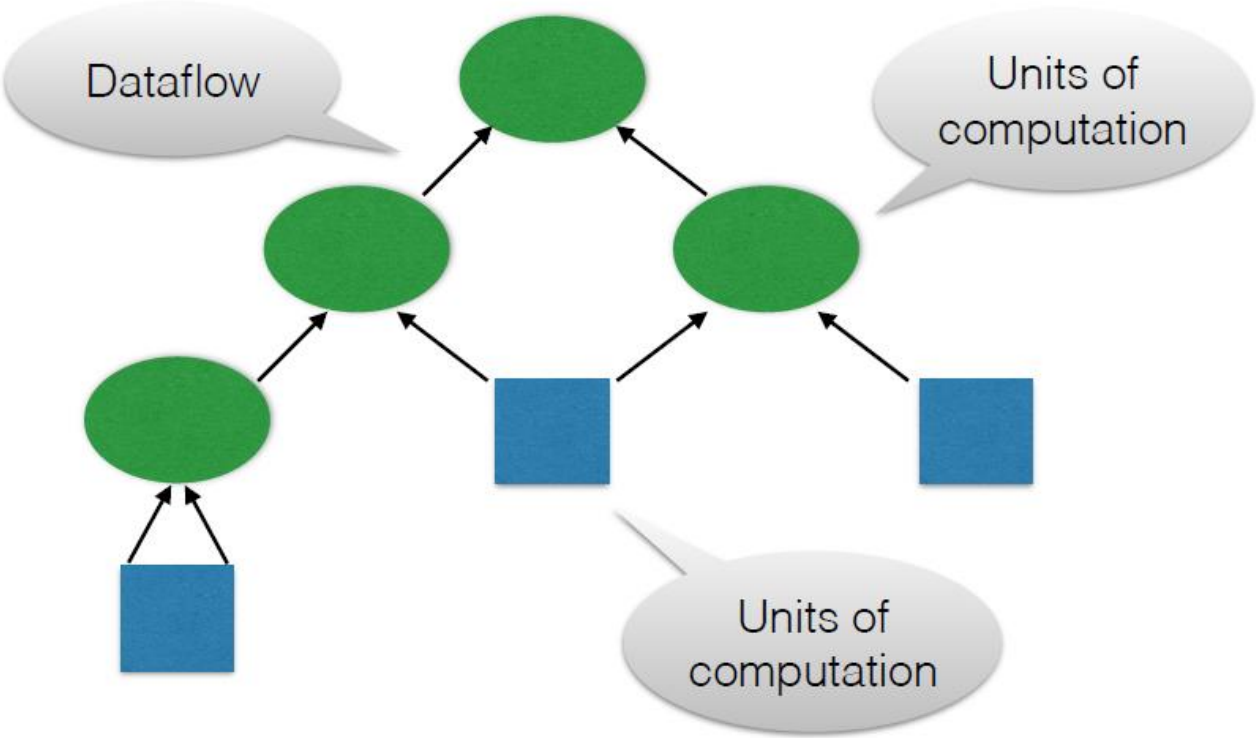


Dataflow Graph

Dataflow Graph

Dataflow programming model for parallel computing:

- Nodes represent units of computation
- Edges represent data consumed or produced by a node

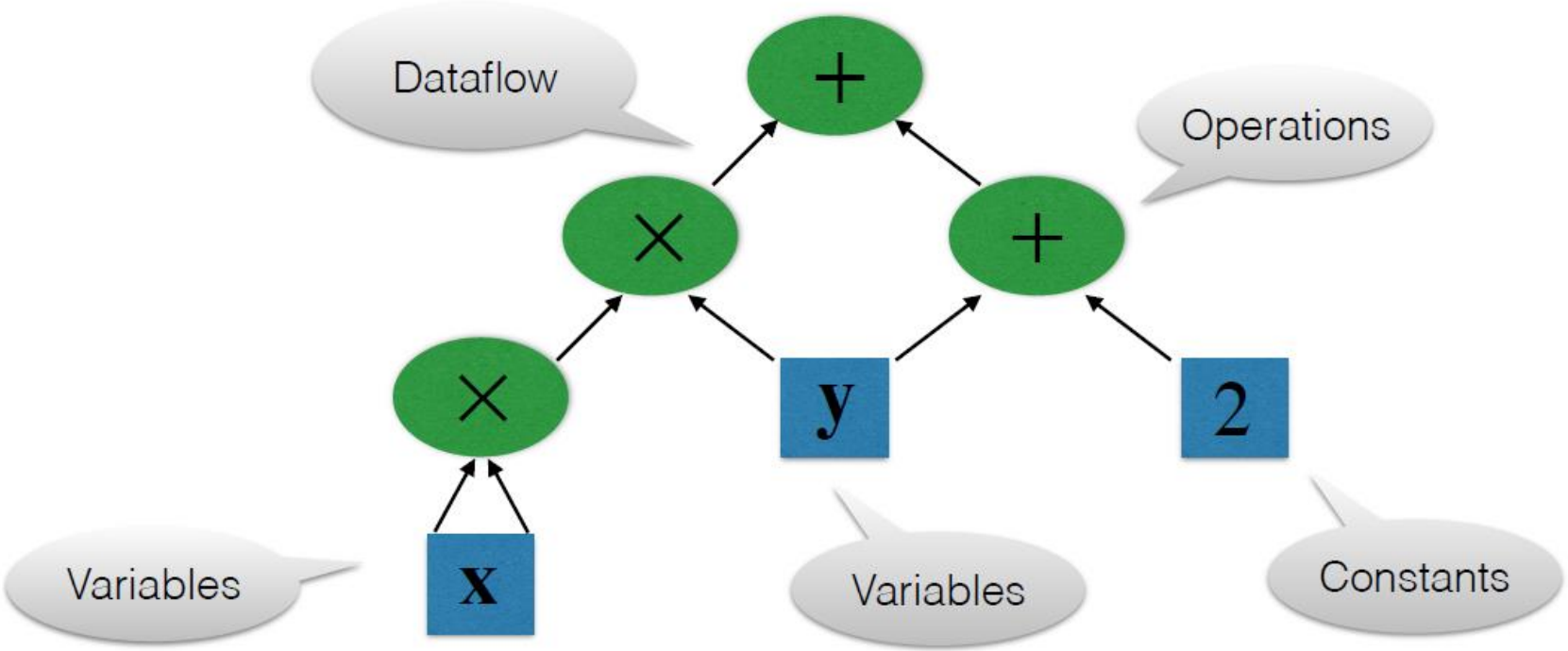


Dataflow Graph

Dataflow graph nodes:

- Variables
- Operations
- Constants

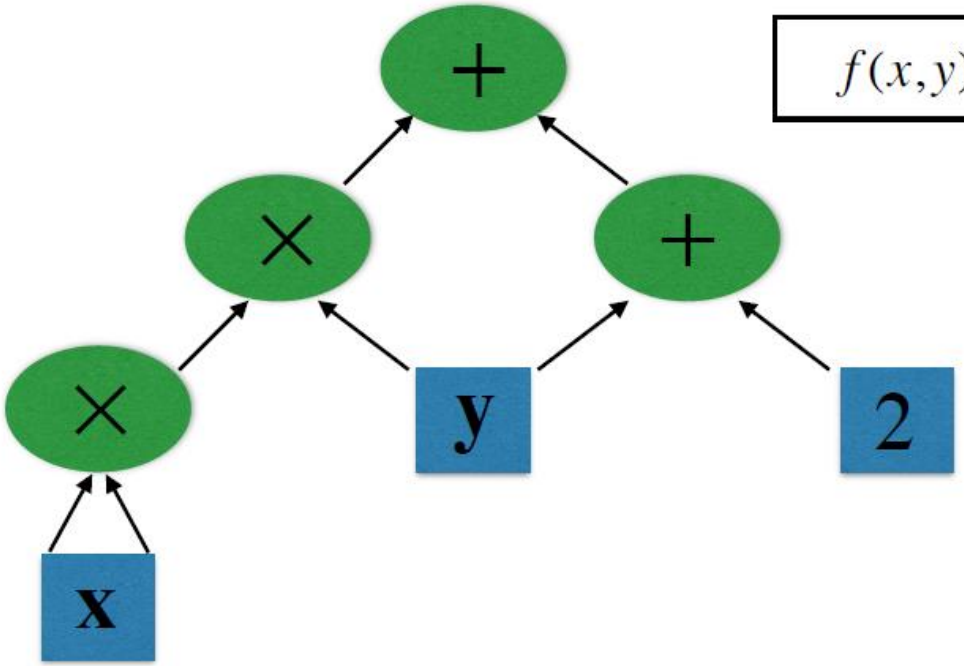
$$f(x,y) = x^2 y + y + 2$$



Dataflow Graph

Tensors:
Rank n= 0: a number
Rank n=1: a list of numbers
Rank n=2: a list of lists
Rank n=3: a list of lists of lists…

Implemented as a Nd-array in numpy



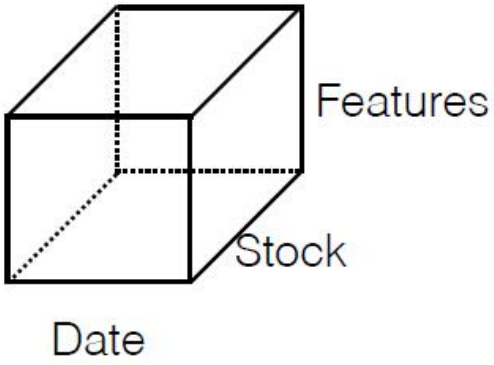
$$f(x,y) = x^2 y + y + 2$$

Dataflow Graph

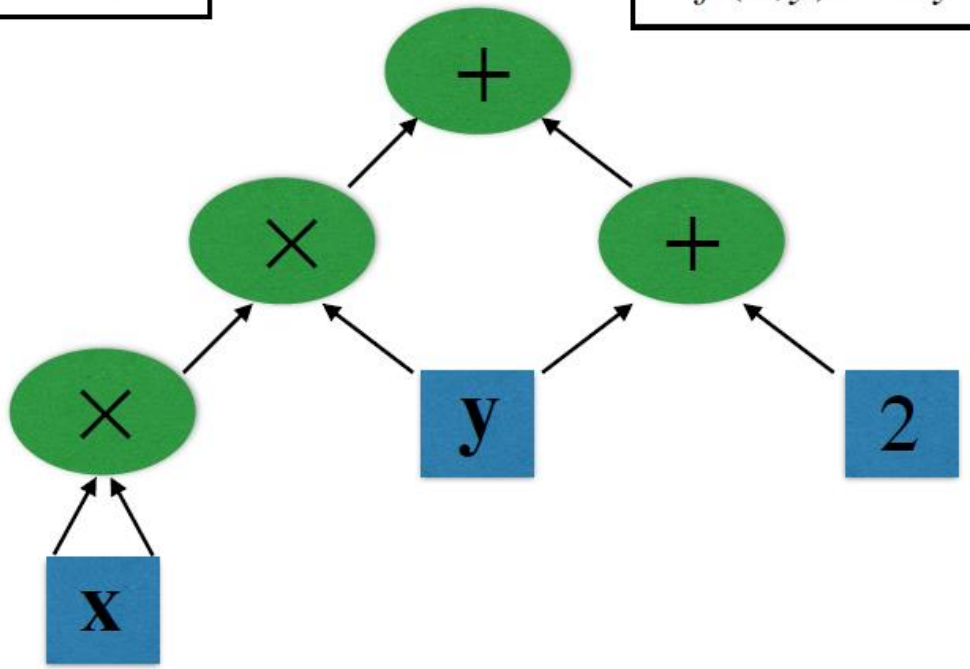
Tensors:
Rank n= 0: a number
Rank n=1: a list of numbers
Rank n=2: a list of lists
Rank n=3: a list of lists of lists…

Implemented as a Nd-array in numpy

Stock data as a rank-3 tensor:



$$f(x,y) = x^2y + y + 2$$

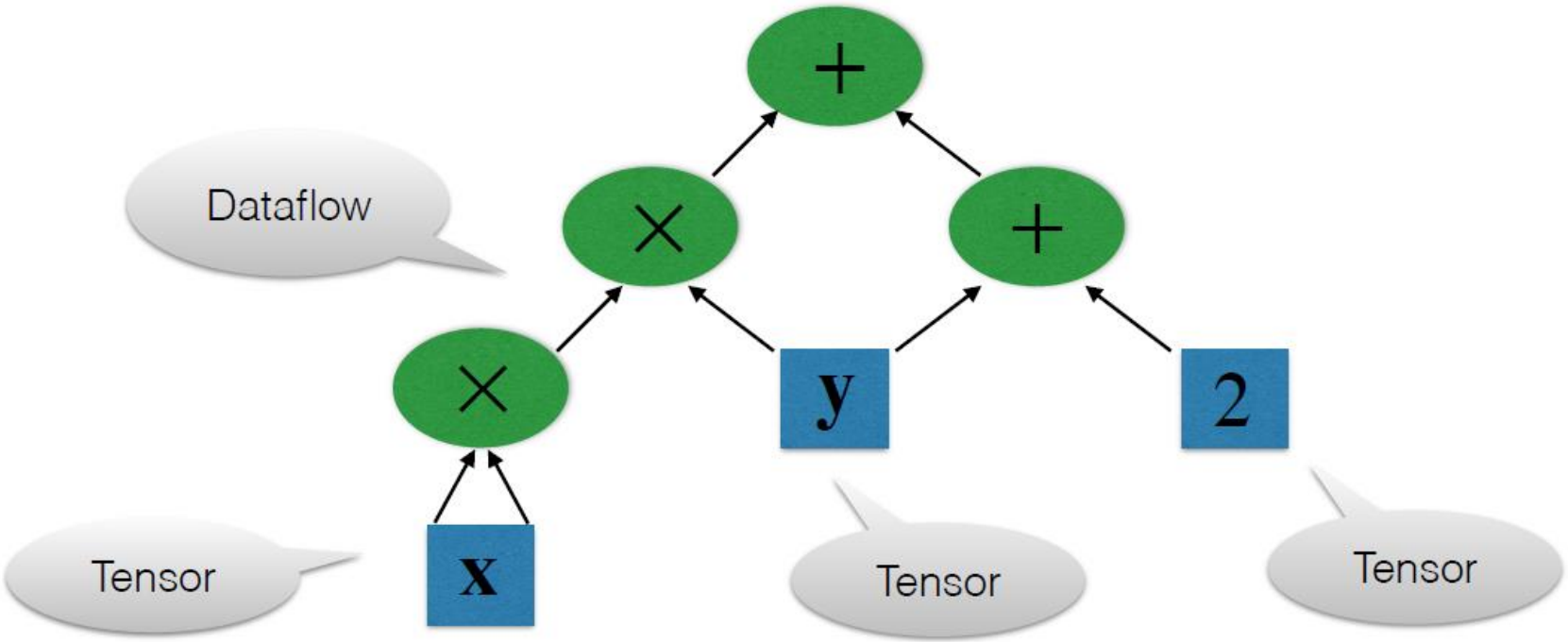


TensorFlow

TensorFlow:

- Construct a a dataflow graph (in Python)
- Run the graph using optimized C++ code
- Open-sourced by Google in 2015

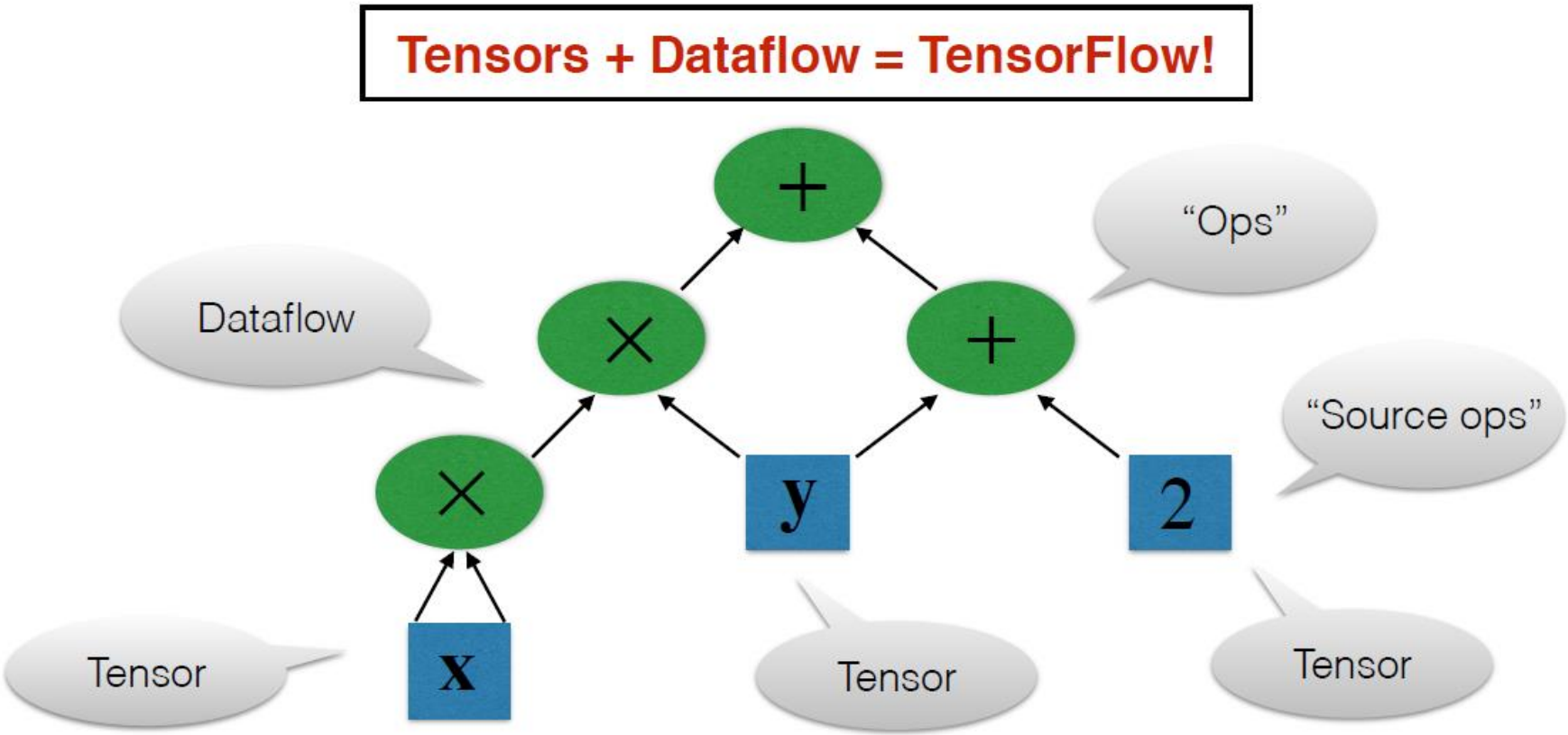
Tensors + Dataflow = TensorFlow!



TensorFlow: Lazy Evaluation

TensorFlow:

- Construct a a dataflow graph (in Python) (**no evaluation yet**!)
- Run the graph using optimized C++ code (**execute to evaluate** the graph)



TensorFlow: Reverse-Mode Autodiff

TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)

TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f\left(n_i(x)\right)$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

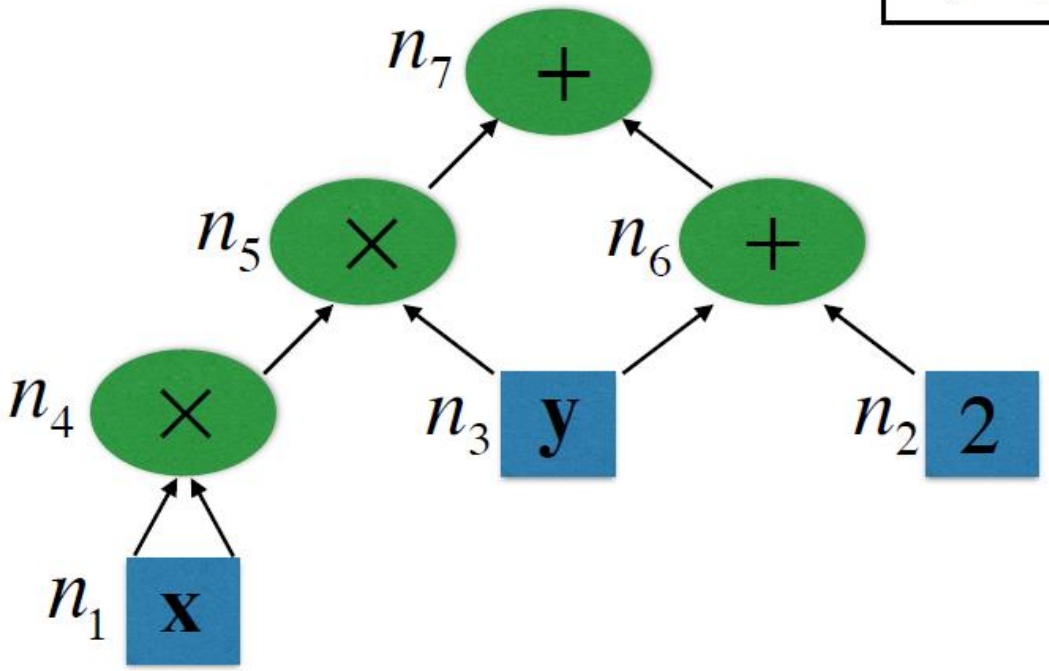
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



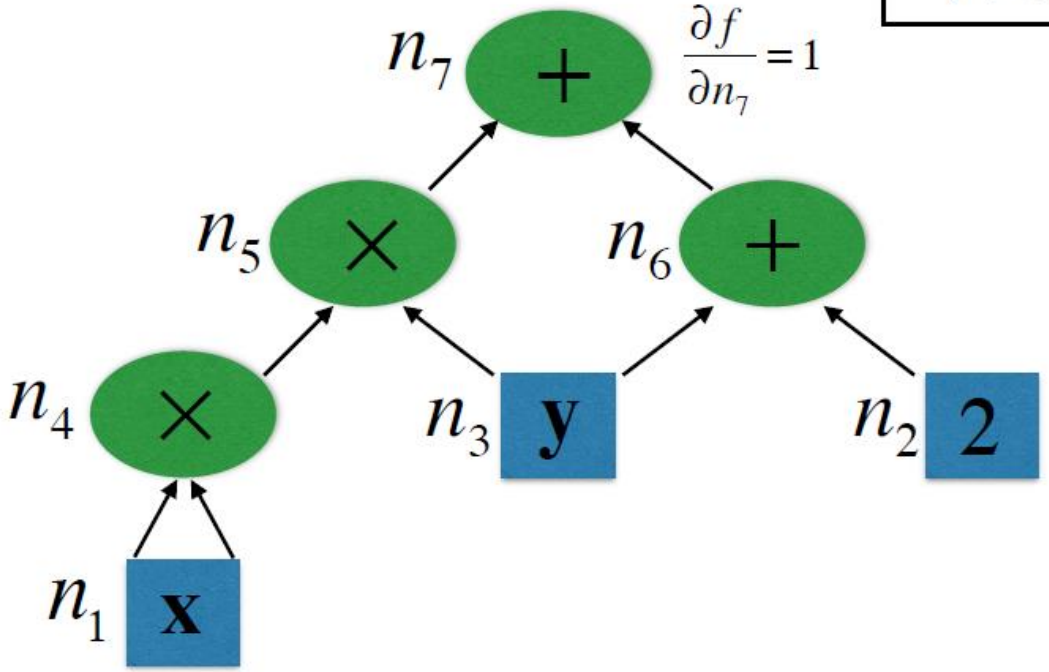
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



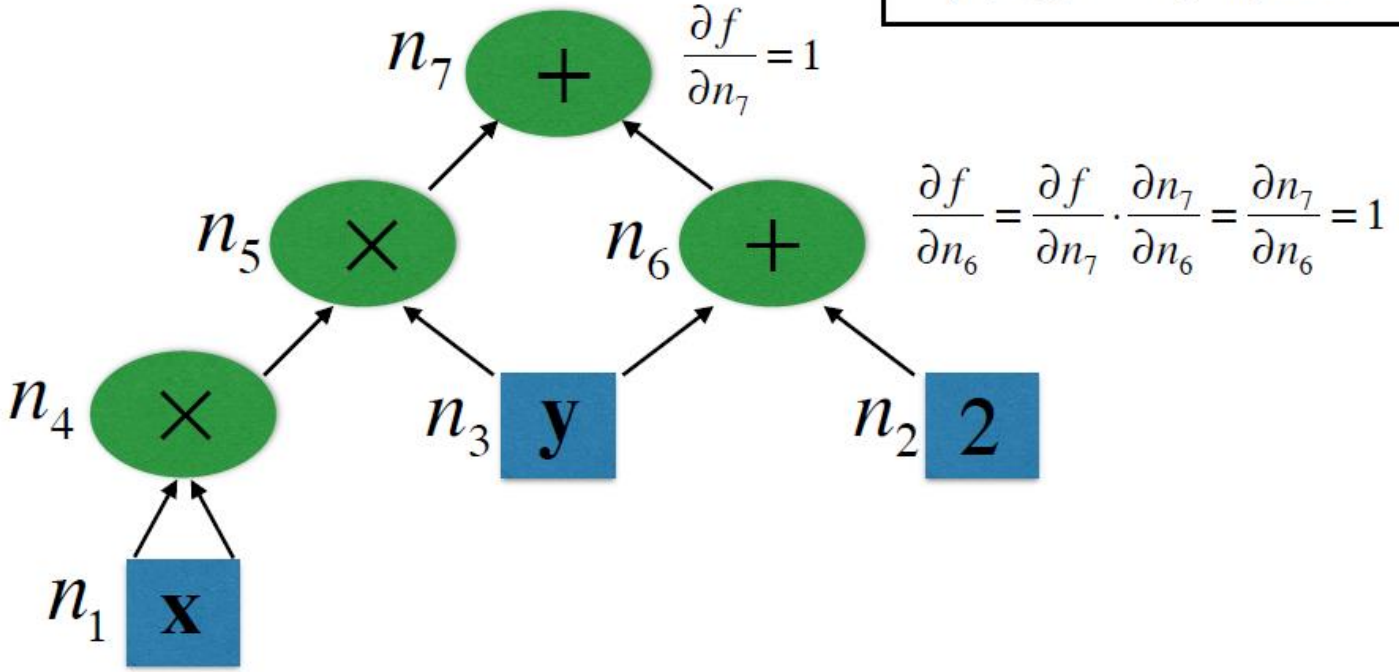
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



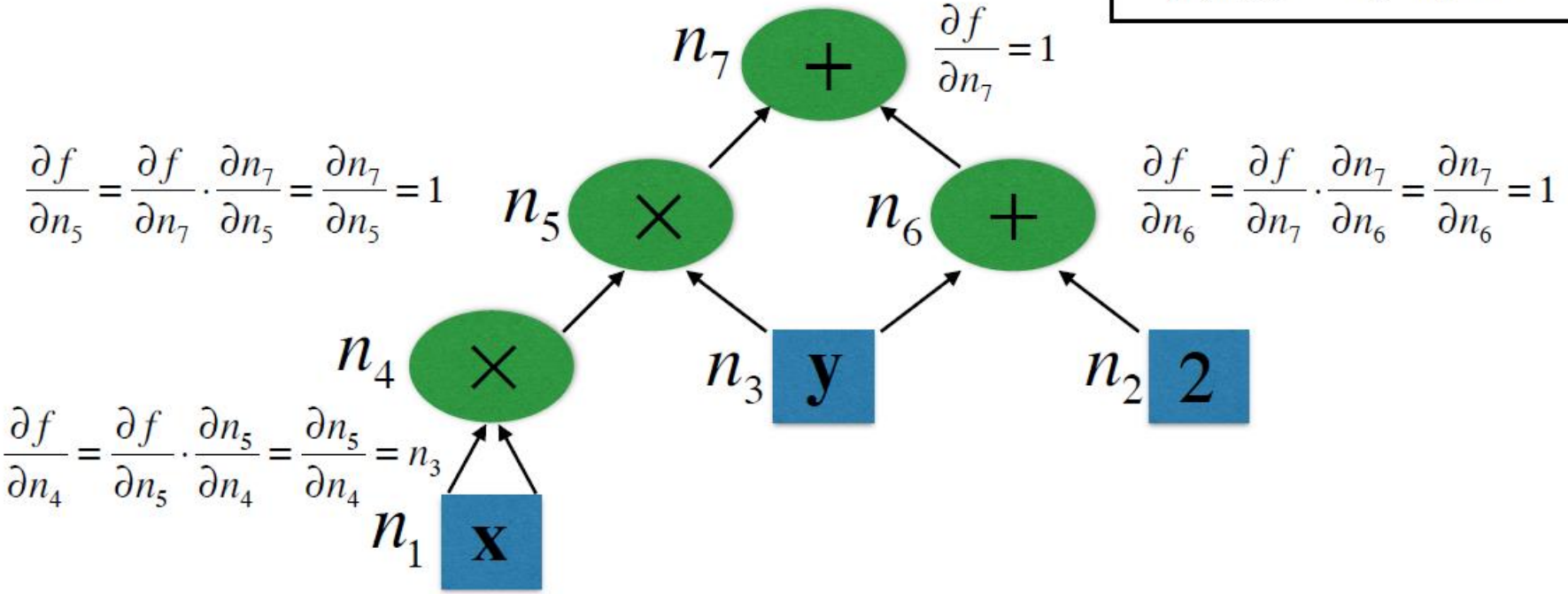
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



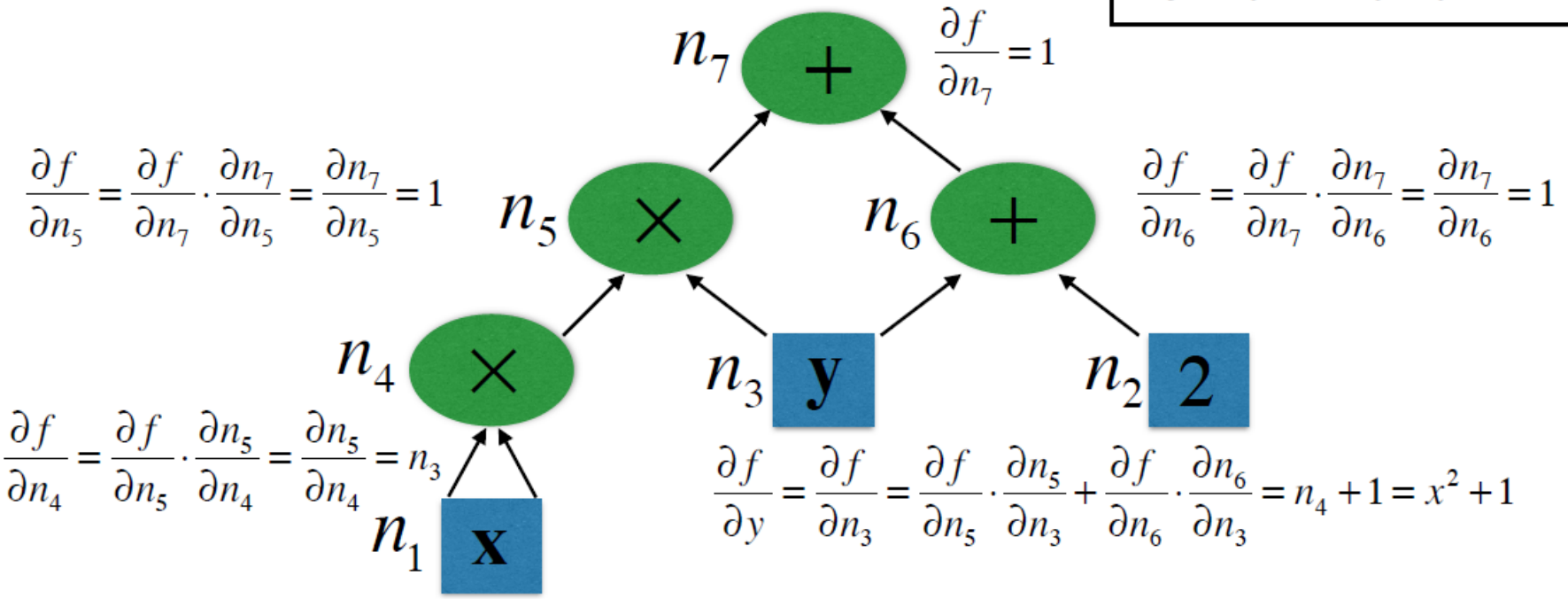
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



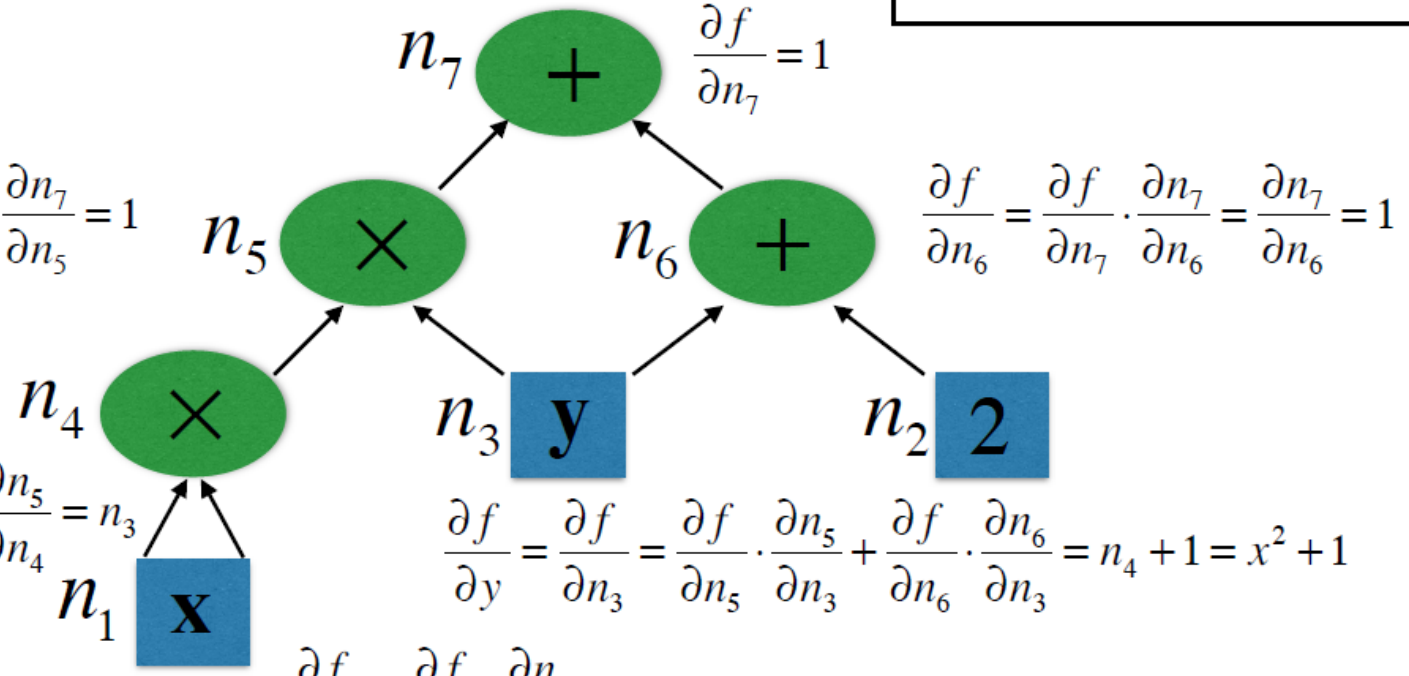
TensorFlow: Reverse-Mode Autodiff

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



AD in Quant Finance

AD in Quant Finance

Motivation

- The technique was only recently introduced to finance via the award-winning work in Glasserman & Giles (2006).
- Even after that it remained largely confidential until the regulations of 2008-2011 forced banks to calculate frequent and massively multi-dimensional bank-wide risks.
- Imagine you want to value a barrier option in Dupire's local volatility model by simulating 1,000,000 paths each with 252 steps. With Monte-Carlo simulation, that would take about one second to run on a standard laptop.
- But now suppose that you want to do a vol risk report, say calculate vegas wrt. 30 $\times$ 60 points in the local volatility matrix.
  - Standard one-sided bump-and-recalculate would take about half an hour.
  - With automatic adjoint differentiation, you get the same numbers in a few seconds!!!

AD in Quant Finance

Idea Sketch

- If we compute the product $M_1 M_2 v$, where the M's are $n \times n$ matrices and v is a vector in $\mathbb{R}^n$, $M_1 M_2$ is of $O\left(n^3\right)$ complexity and results in a square matrix. The complexity of its product with v is of $O\left(n^2\right)$. Thus the whole computation is $O\left(n^3 + n^2\right)$.
- But suppose we go from right to left: Each calculation is "vector times matrix", so complexity is now only $\mathrm{O}\left(2n^2\right)$, **an order of magnitude lower.** **(in powers of *n*)**
- The same applies to derivatives. Consider the scalar function $F : \mathbb{R}^n \mapsto \mathbb{R}$ such that

$$F(x) = H\{K[G(x)]\}$$

for smooth functions $G, K : \mathbb{R}^n \mapsto \mathbb{R}^n$ and $H : \mathbb{R}^n \mapsto \mathrm{R}$. The multi-variate chain rule then tells us that

$$\underbrace{\frac{\partial F}{\partial x}}_{n\times 1} = \underbrace{\frac{\partial G}{\partial x}}_{n\times n}\underbrace{\frac{\partial K}{\partial G}}_{n\times n}\underbrace{\frac{\partial H}{\partial K}}_{n\times 1}$$

and we are back to the matrix by matrix by vector product, which we just saw computes faster in reverse order. Because F is a scalar function, the rightmost Jacobian is a vector and when we multiply in reverse order, every operation collapses into a vector and the resulting complexity **drops by an order.**

AD in Quant Finance

- The point now is: how can we compute Greeks for pricers with no analytical formula?
- How can we use these ideas for model calibration?
- I will sketch an answer for the first question when we price using the Monte Carlo method. Other questions or methods can be coped in similar way

AD in Quant Finance

Example in Monte Carlo framework

We consider the SDE for a N-dimensional X vector

$$dX = a(X,t)dt + \sigma(X,t)dW$$

The initial value for X, at time $\mathrm{t} = 0$, is denoted by $X^0 = \left(X_1\left(T^0\right), \ldots, X_N\left(T^0\right)\right)$ For simplicity of exposition, we consider that we discretize it using Euler-Maruyama scheme, with time points $T^k, k = 1 \ldots M$ and $T^0 = 0$

$$X\left(T^{k+1}\right) - X\left(T^k\right) = a\left(X\left(T^k\right), T^k\right)\Delta t + \sigma\left(X\left(T^k\right), T^k\right)\Delta W^k$$

$$\Delta T^k = T^{k+1} - T^k$$

$$\Delta W^k = \varepsilon \cdot \sqrt{T^{k+1} - T^k}$$

where the random number is chosen from $\mathcal{N}(0,1)$ We can recast this discretization scheme into the following expression

$$X\left(T^{k+1}\right) = \Phi\left(X\left(T^k\right), \Theta\right)$$

where Φ may also depend on a set of model parameters $\Theta = \{\theta_1, \ldots, \theta_P\}$. We also consider that we have to price an option with payoff $G(X(T))$ and want to compute price sensitivities ("Greeks") with respect to X^0 and, respectively, to the model parameters Θ

Example in MC: Forward Tangent Linear Mode

We consider first the sensitivities with respect to initial conditions

$$\frac{\partial G(X(T))}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \frac{\partial X_j(T)}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \Delta_{ij}\left(T^0, T\right)$$

where we have used notation $\Delta_{ij}\left(T^0, T^k\right) \triangleq \frac{\partial X_j\left(T^k\right)}{\partial X_i\left(T^0\right)}$ We rewrite (5.2) in vector matrix notation

$$\left[\frac{\partial G(X(T))}{\partial X(T^0)}\right]^{\mathrm{T}} = \left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot \Delta\left(T^0, T\right)$$

For simplicity we write the previous relationship in the following format, using the fact that $T^M = T$

$$\left[\frac{\partial G}{\partial X}[0]\right]^{T} = \left[\frac{\partial G}{\partial X}[M]\right]^{T} \cdot \Delta[M]$$

Differentiating the SDE yields

$$\frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$$

$$\Rightarrow \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^0\right)} = \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = D[k] \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)}$$

where $D[k] \triangleq \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$. We now can rewrite the above equation as

$$\Delta[k+1] = D[k] \cdot \Delta[k]$$

Example in MC: Forward Tangent Linear Mode

We consider first the sensitivities with respect to initial conditions

$$\frac{\partial G(X(T))}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \frac{\partial X_j(T)}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \Delta_{ij}\left(T^0, T\right)$$

where we have used notation $\Delta_{ij}\left(T^0, T^k\right) \triangleq \frac{\partial X_j\left(T^k\right)}{\partial X_i\left(T^0\right)}$ We rewrite (5.2) in vector matrix notation

$$\left[\frac{\partial G(X(T))}{\partial X(T^0)}\right]^{\mathrm{T}} = \left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot \Delta\left(T^0, T\right)$$

For simplicity we write the previous relationship in the following format, using the fact that $T^M = T$

$$\boxed{\left[\frac{\partial G}{\partial X}[0]\right]^{T} = \left[\frac{\partial G}{\partial X}[M]\right]^{T} \cdot \Delta[M]}$$

Differentiating the SDE yields

$$\frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$$

$$\Rightarrow \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^0\right)} = \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = D[k] \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)}$$

where $D[k] \triangleq \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$. We now can rewrite the above equation as

$$\Delta[k+1] = D[k] \cdot \Delta[k]$$

Example in MC: Forward Tangent Linear Mode

We consider first the sensitivities with respect to initial conditions

$$\frac{\partial G(X(T))}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \frac{\partial X_j(T)}{\partial X_i(T^0)} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \Delta_{ij}\left(T^0, T\right)$$

where we have used notation $\Delta_{ij}\left(T^0, T^k\right) \triangleq \frac{\partial X_j\left(T^k\right)}{\partial X_i\left(T^0\right)}$ We rewrite (5.2) in vector matrix notation

$$\left[\frac{\partial G(X(T))}{\partial X(T^0)}\right]^{\mathrm{T}} = \left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot \Delta\left(T^0, T\right)$$

For simplicity we write the previous relationship in the following format, using the fact that $T^M = T$

$$\boxed{\left[\frac{\partial G}{\partial X}[0]\right]^{T} = \left[\frac{\partial G}{\partial X}[M]\right]^{T} \cdot \Delta[M]}$$

Differentiating the SDE yields

$$\frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$$

$$\Rightarrow \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^0\right)} = \frac{\partial X\left(T^{k+1}\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)} \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)} = D[k] \frac{\partial X\left(T^k\right)}{\partial X\left(T^0\right)}$$

where $D[k] \triangleq \frac{\partial \Phi\left(X\left(T^k\right), \Theta\right)}{\partial X\left(T^k\right)}$. We now can rewrite the above equation as

$$\boxed{\Delta[k+1] = D[k] \cdot \Delta[k]}$$

Example in MC: Forward Tangent Linear Mode

Then we have an iterative process

$$\left[\frac{\partial G}{\partial X}[0]\right]^{\mathrm{T}} = \left[\frac{\partial G}{\partial X}[M]\right]^{\mathrm{T}} \cdot D[M-1]\cdots D[0]\cdot \Delta[0],$$

where $\Delta[0]$ is the identity matrix, since

$$\Delta[0] = \left(\frac{\partial X_j\left(T^0\right)}{\partial X_k\left(T^0\right)}\right)_{jk} = \begin{pmatrix} 1 & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & 1 & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

The iterations here employ matrix-matrix product. We will see later that the adjoint mode will involve matrix-vector product instead, which will offer important computational savings.

Now we move to the sensitivities with respect to model parameters

$$\frac{\partial G(X(T))}{\partial \theta_p} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \frac{\partial X_j(T)}{\partial \theta_p} = \sum_{j=1}^{N} \frac{\partial G(X(T))}{\partial X_j(T)} \Psi_{jp}(T)$$

where we have used notation $\Psi_{jp}\left(T^k\right) \triangleq \frac{\partial X_j\left(T^k\right)}{\partial \theta_p}$

Example in MC: Forward Tangent Linear Mode

Differentiating the payoff with respect to parameters yields

$$\frac{\partial X\left(T^{k+1}\right)}{\partial \Theta}=\frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial X\left(T^{k}\right)} \frac{\partial X\left(T^{k}\right)}{\partial \Theta}+\frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial \Theta}$$

Making the notations $D[k] \triangleq \frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial X\left(T^{k}\right)}$ and $B[k] \triangleq \frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial \Theta}$ for the corresponding matrices, we rewrite the above equation as

$$\Psi[k+1]=D[k] \cdot \Psi[k]+B[k]$$

Then we have an iterative process

$$\left[\frac{\partial G(X(T))}{\partial \Theta}\right]^{\mathrm{T}}=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot \Psi[M]$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(D[M-1] \cdot \Psi[M-1]+B[M-1])$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(D[M-1] \cdot(D[M-1] \cdot \Psi[M-1]+B[M-1])+B[M-1])$$

$$=\cdots$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(B[M-1]+D[M-1] \cdot B[M-2]+\cdots+D[M-1] \cdots \quad \cdot D[1] \cdot B[0])$$

Example in MC: Forward Tangent Linear Mode

Differentiating the payoff with respect to parameters yields

$$\frac{\partial X\left(T^{k+1}\right)}{\partial \Theta}=\frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial X\left(T^{k}\right)} \frac{\partial X\left(T^{k}\right)}{\partial \Theta}+\frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial \Theta}$$

Making the notations $D[k] \triangleq \frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial X\left(T^{k}\right)}$ and $B[k] \triangleq \frac{\partial \Phi\left(X\left(T^{k}\right), \Theta\right)}{\partial \Theta}$ for the corresponding matrices, we rewrite the above equation as

$$\Psi[k+1]=D[k] \cdot \Psi[k]+B[k]$$

Then we have an iterative process

$$\left[\frac{\partial G(X(T))}{\partial \Theta}\right]^{\mathrm{T}}=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot \Psi[M]$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(D[M-1] \cdot \Psi[M-1]+B[M-1])$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(D[M-1] \cdot(D[M-1] \cdot \Psi[M-1]+B[M-1])+B[M-1])$$

$$=\cdots$$

$$=\left[\frac{\partial G(X(T))}{\partial X(T)}\right]^{\mathrm{T}} \cdot(B[M-1]+D[M-1] \cdot B[M-2]+\cdots+D[M-1] \cdots \quad \cdot D[1] \cdot B[0])$$

Example in MC: Reverse Adjoint Mode

For computing the sensitivities with respect to initial conditio, we just adjoint the previous recursive equation

$$\frac{\partial G}{\partial X}[0] = D^T[0] \cdot D^T[1] \cdots D^T[M-1] \cdot \frac{\partial G}{\partial X}[T]$$

where we have already set $\Delta[0]$ equal to the identity matrix. We note that the product in this iterative process is of the matrix-vector type, not matrix-matrix as it was for tangent linear mode.
Now we move to sensitivities with respect to model parameters. We use again the adjoint version of previous equations

$$\frac{\partial G(X)}{\partial \Theta}[M] = \Big(B^{\mathrm{T}}[M-1] + B^{\mathrm{T}}[M-2] \cdot D^{\mathrm{T}}[M-2] + \cdots + B^{\mathrm{T}}[0] \cdot D^{\mathrm{T}}[1] \cdots D^{\mathrm{T}}[M-1] \Big) \frac{\partial G(X)}{\partial X}[M]$$

The values of adjoint vectors $V[k]$ were computed as part of the adjoint approach for sensitivities with respect to initial conditions.
The values of $B[k]$ can be precomputed. We may be able to compute them analytically (e.g., if the parameters are volatilities and vol surface is assumed to have a certain parametric representation, such as cubic spline), otherwise we can employ the adjoint

TensorFlow Demo

Introduction to TensorFlow

TensorFlow makes it easy for beginners and experts to create machine learning models for desktop, mobile, web, and cloud. See the sections below to get started.

TensorFlow

Learn the foundation of TensorFlow with tutorials for beginners and experts to help you create your next machine learning project.

Learn more

For JavaScript

Use TensorFlow.js to create new machine learning models and deploy existing models with JavaScript.

Learn more

For Mobile & IoT

Run inference with TensorFlow Lite on mobile and embedded devices like Android, iOS, Edge TPU, and Raspberry Pi.

Learn more

For Production

Deploy a production-ready ML pipeline for training and inference using TensorFlow Extended (TFX).

Learn more

Swift for TensorFlow

Integrate directly with Swift for TensorFlow, the next generation platform for deep learning and differentiable programming.

Learn more

AD in Quant Finance

- Exercises:
  - compute Delta, Theta, Rho, Vega for the Black-Scholes formula in TensorFlow
  - Simulate GBM in MC using TensorFlow. Now, recompute the Greeks

Structure of your TensorFlow model

- Phase 1: assemble your graph
  1. Define placeholders for input and output
  2. Define the weights
  3. Define the inference model
  4. Define loss function
  5. Define optimizer
- Phase 2: execute the computation (for ML, training your model):
  1. Initialize all model variables for the first time.
  2. Feed in the training data. Might involve randomizing the order of data samples.
  3. Execute the inference model on the training data, so it calculates for each training input example the output with the current model parameters.
  4. Compute the cost
  5. Adjust the model parameters to minimize/maximize the cost depending on the model.

Artificial Neural Networks

Artificial Neural Networks

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's Law, but also thanks to the gaming industry, which has produced powerful GPU cards.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.

Artificial Neural Networks

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's Law, but also thanks to the gaming industry, which has produced powerful GPU cards.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.

Artificial Neural Networks

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's Law, but also thanks to the gaming industry, which has produced powerful GPU cards.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.

Artificial Neural Networks

Artificial Neural Networks

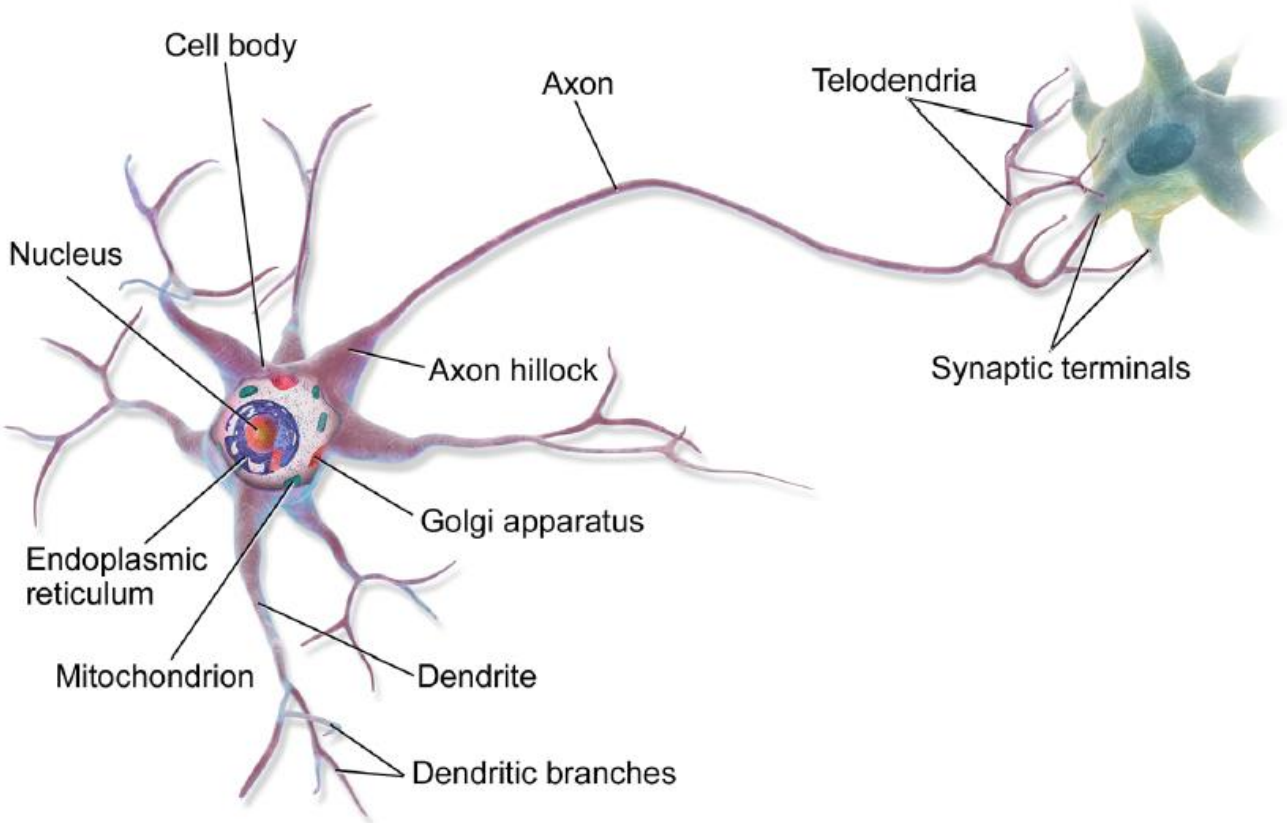
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (or when it is the case, they are usually fairly close to the global optimum).
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress, and even more amazing products.

Artificial Neural Networks

- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (or when it is the case, they are usually fairly close to the global optimum).
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress, and even more amazing products.

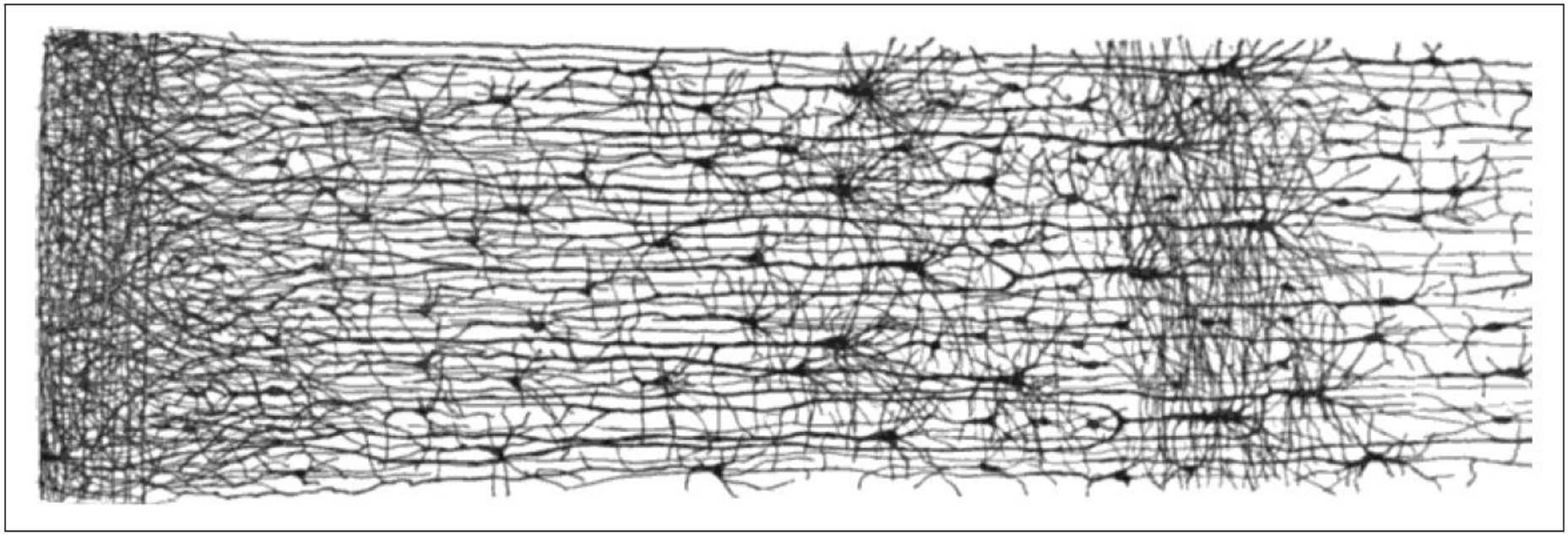
Biological Networks

Biological Networks



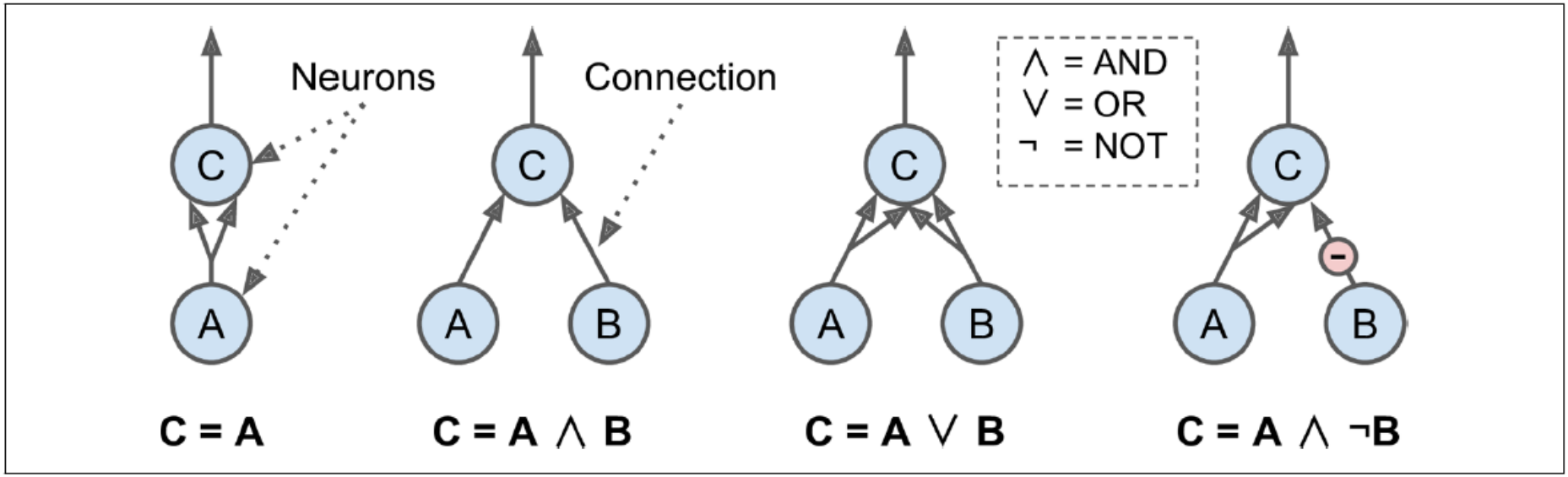
Biological neuron

Biological Networks



Multiple layers in a biological neural network (human cortex)

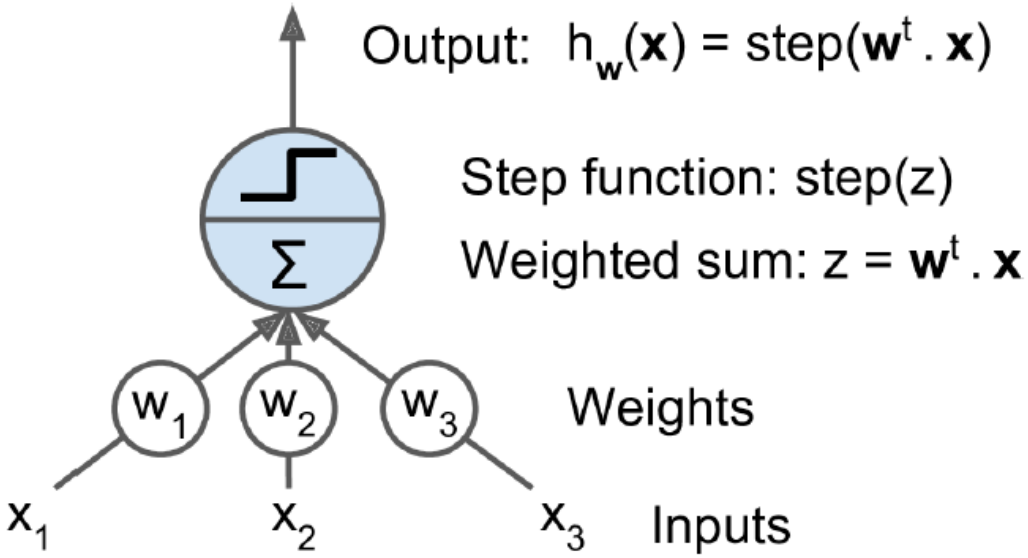
Logical Computations with Neurons



ANNs performing simple logical computations

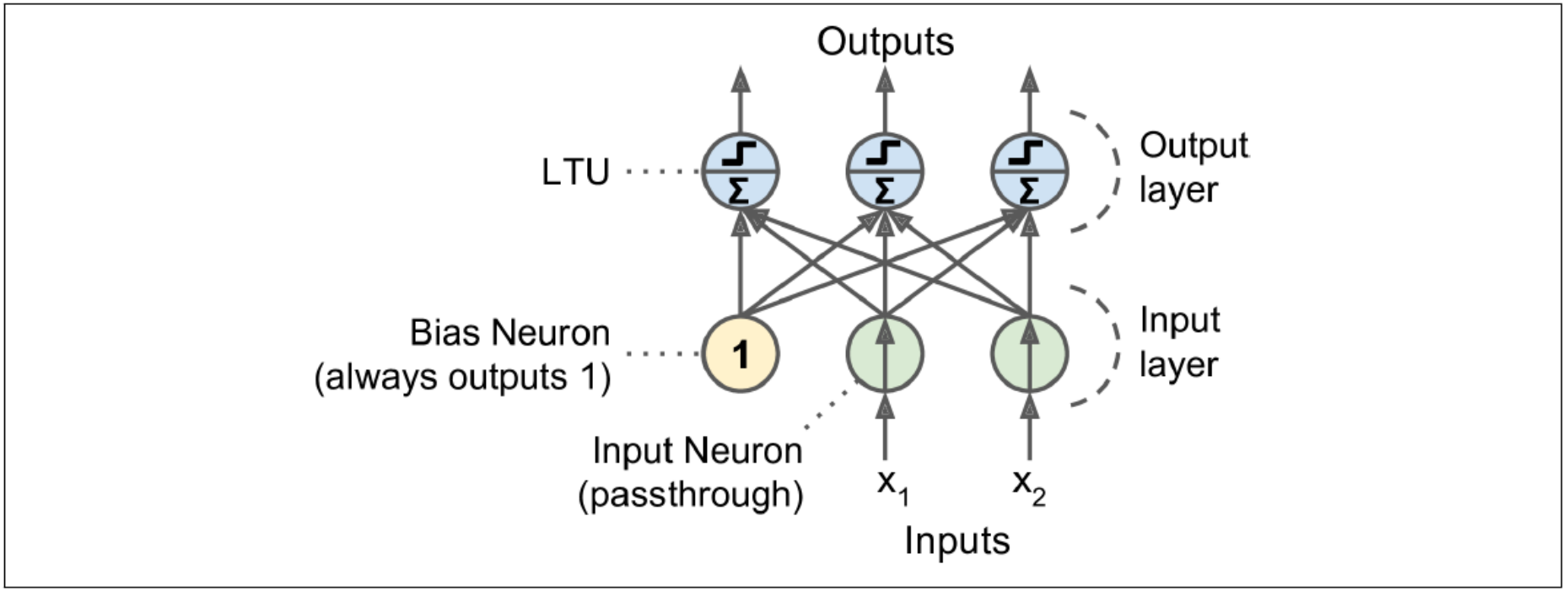
The Perceptron

The Perceptron



Linear threshold unit

The Perceptron

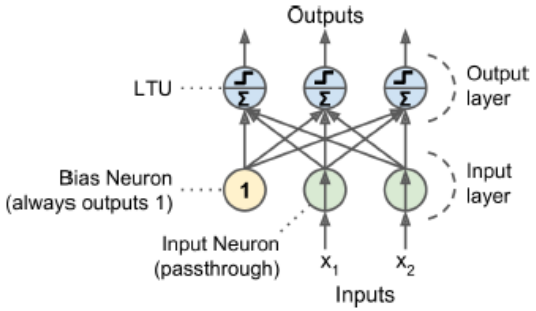


Perceptron diagram

The Perceptron

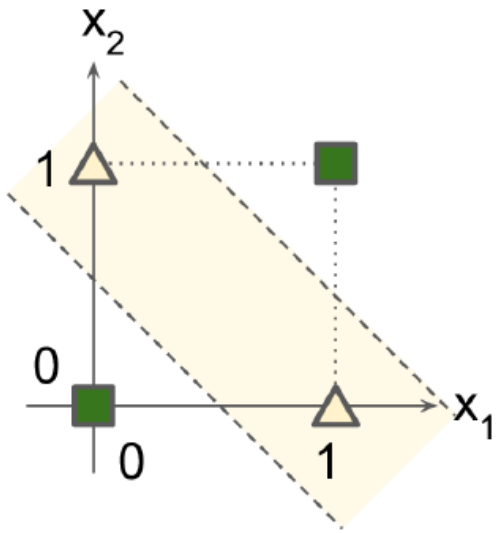
Perceptron learning rule (weight update)

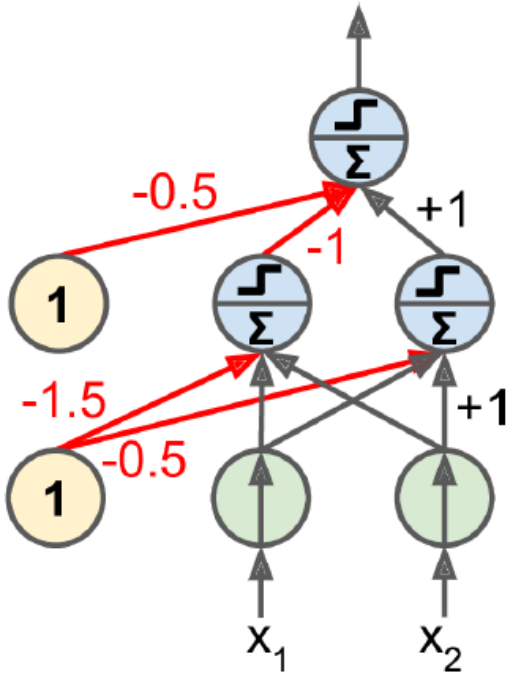
$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\hat{y}_j - y_j\right)x_i$$



- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

Multilayer Perceptron





XOR classification problem and an MLP that solves it

Linear Reg as a Functional Transform

Linear Reg as a Functional Transform

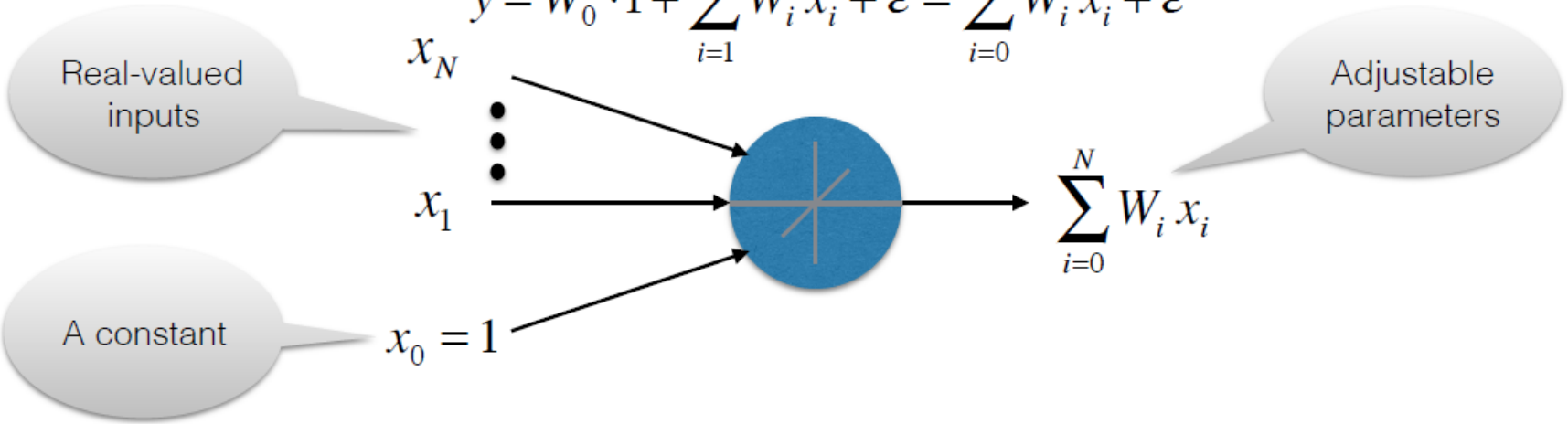
- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **linear function** of inputs
- Weights can be tuned to fit the data

$$y = W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i + \varepsilon = \sum_{i=0}^{N} W_i x_i + \varepsilon$$

Linear Reg as a Functional Transform

- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **linear function** of inputs
- Weights can be tuned to fit the data

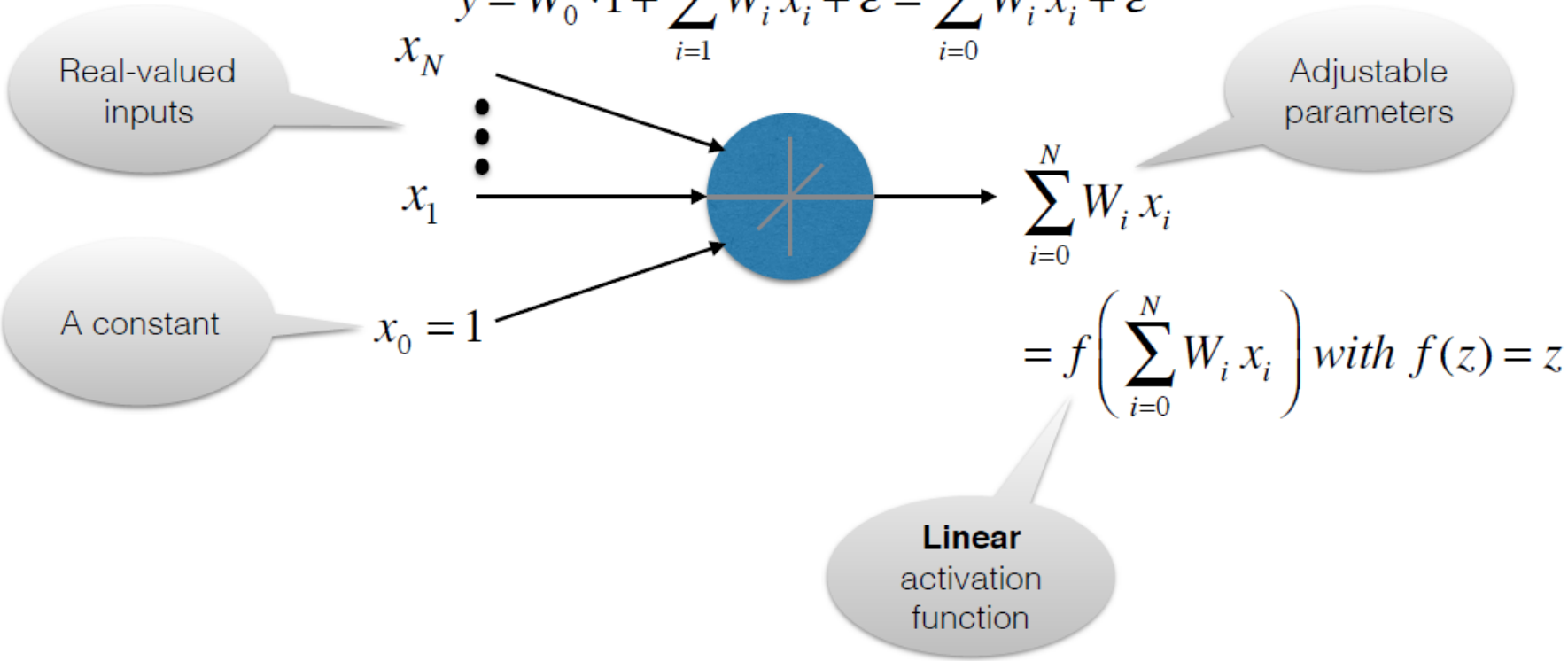
$$y = W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i + \varepsilon = \sum_{i=0}^{N} W_i x_i + \varepsilon$$



Linear Reg as a Functional Transform

- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **linear function** of inputs
- Weights can be tuned to fit the data

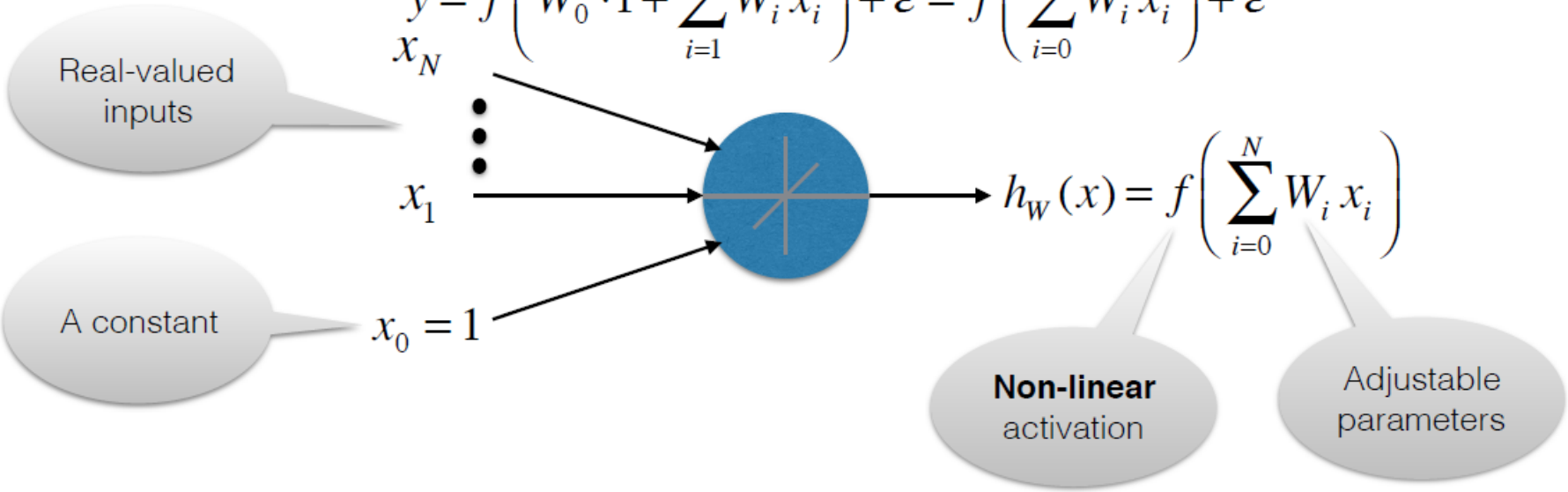
$$y = W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i + \varepsilon = \sum_{i=0}^{N} W_i x_i + \varepsilon$$



Non-Linear Reg as Functional Transf

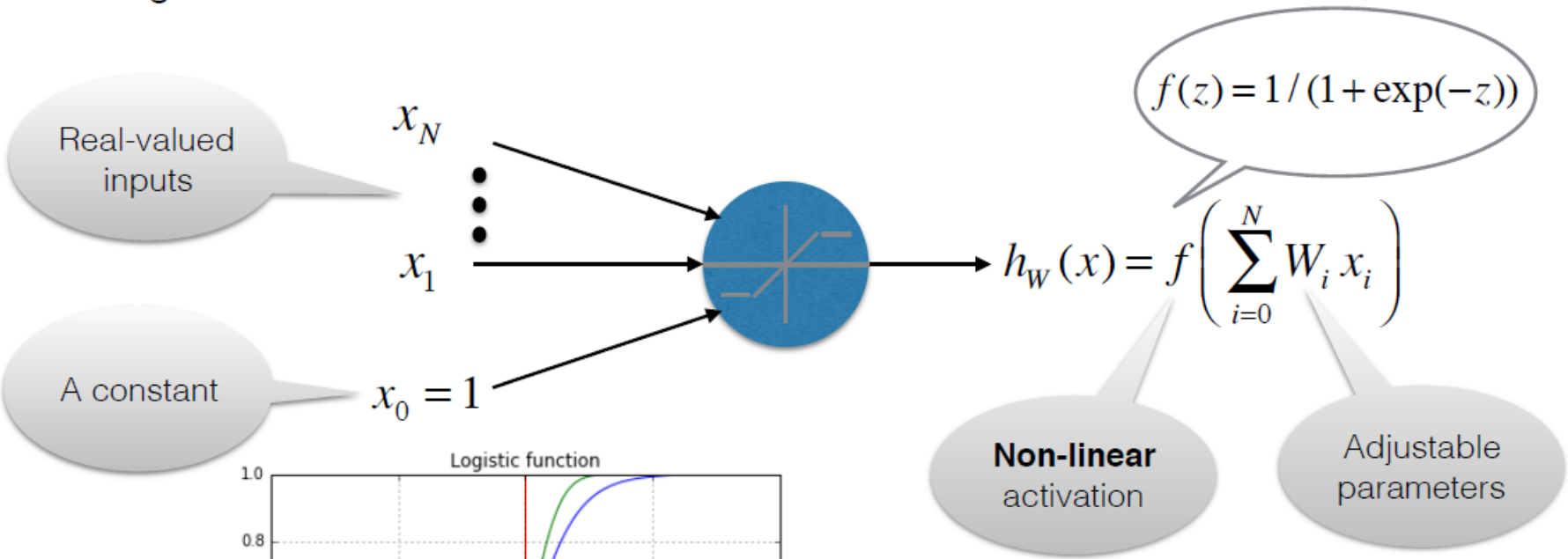
- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **non-linear function** of inputs
- Weights can be tuned to fit the data

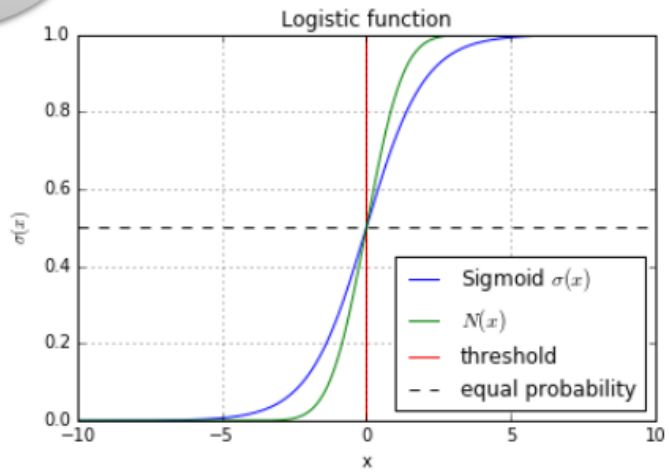
$$y = f\left(W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i\right) + \varepsilon = f\left(\sum_{i=0}^{N} W_i x_i\right) + \varepsilon$$



Logistic Reg as Functional Transform

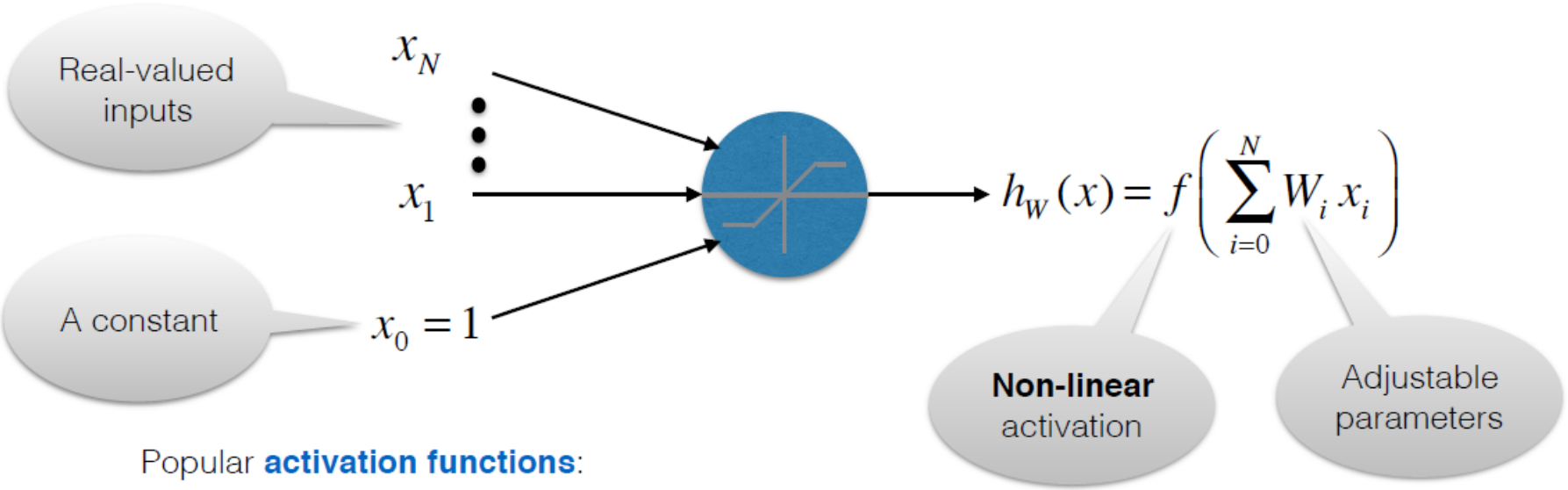
- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **non-linear function** of inputs
- Weights can be tuned to fit the data





Artificial Neuron

- **Artificial Neuron:** (perceptron): a function implementing a non-linear transformation of its input data (Rosenblatt, 1957)
- Can be viewed as a caricature of a physical neuron



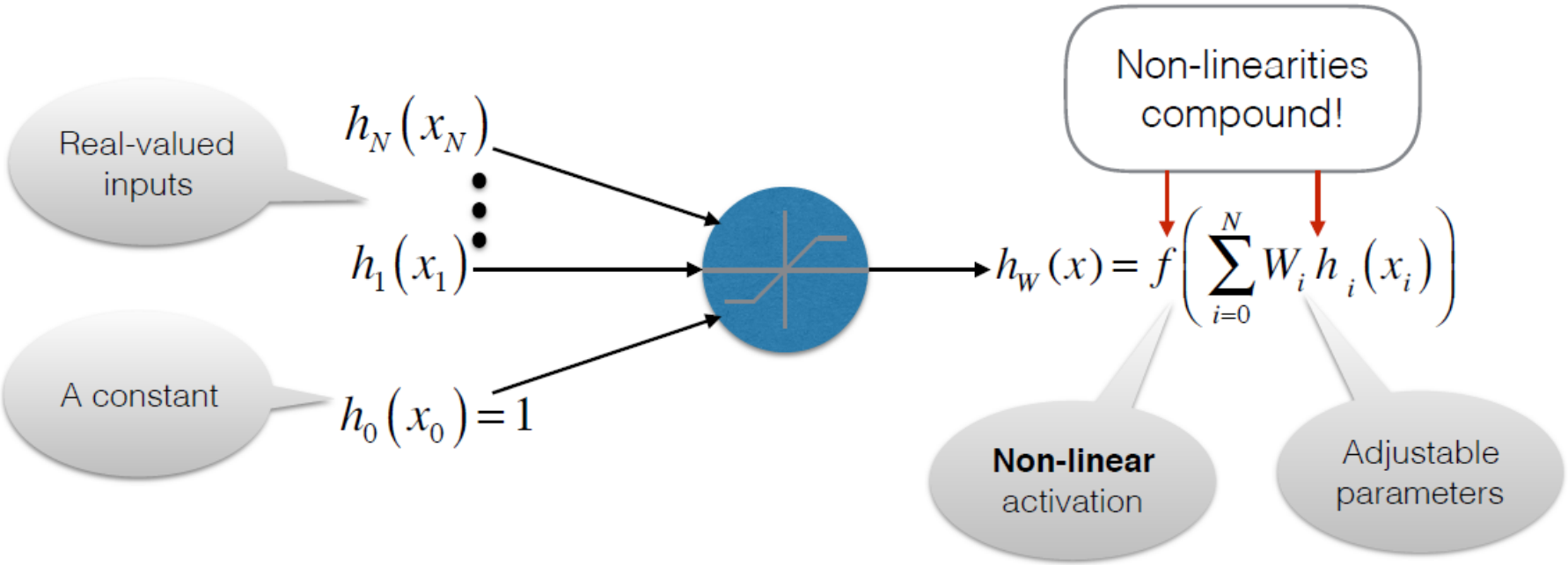
Popular **activation functions**:

- Sigmoid (logistic) function $f(z) = 1/(1+\exp(-z))$
- Tanh function $f(z) = (e^z - e^{-z})/(e^z + e^{-z})$
- Rectified linear neuron $f(z) = \max(z,0) \equiv z_+$

Logistic regression

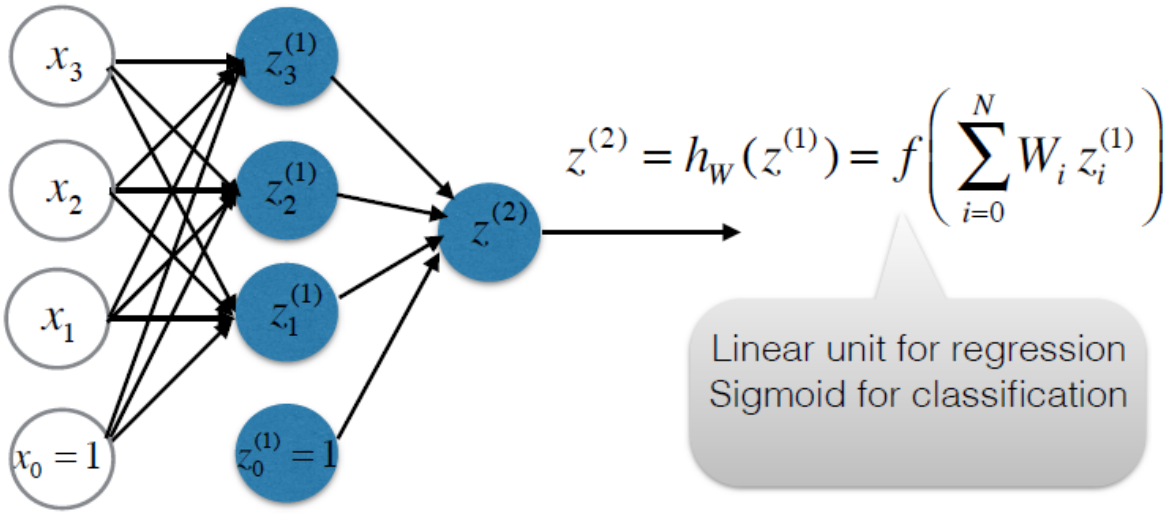
Composite Function Transform 1N

- Inputs to a neuron can themselves be non-linear transforms of raw inputs
- The output will have a composite non-linearity
- Can fit very complex functions



Artificial Neural Network

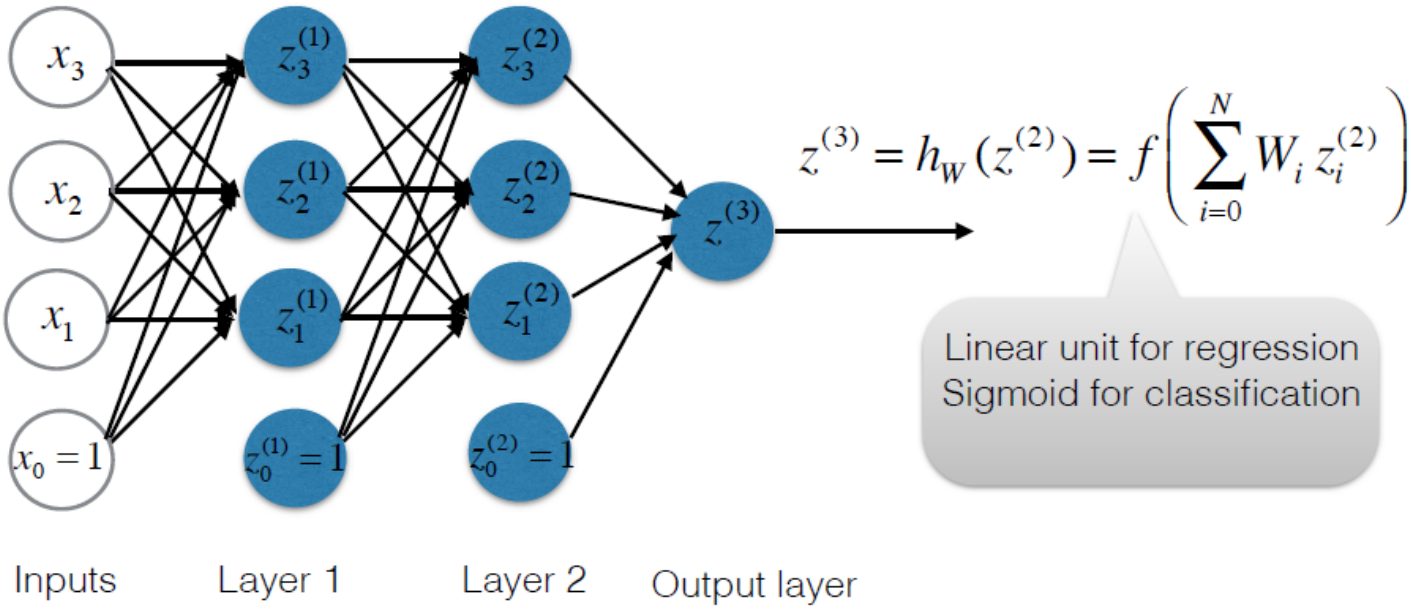
- **Feedforward Neural Network** is composed of layers of artificial neurons (perceptrons)



- This is a single (hidden) layer feedforward network
- Artificial Neural Network (ANN) is a highly stylized model of how the neocortex in mammals' brain processes visual and audio signals
- **Logistic regression = No hidden layer**

Neural Networks with more Layers

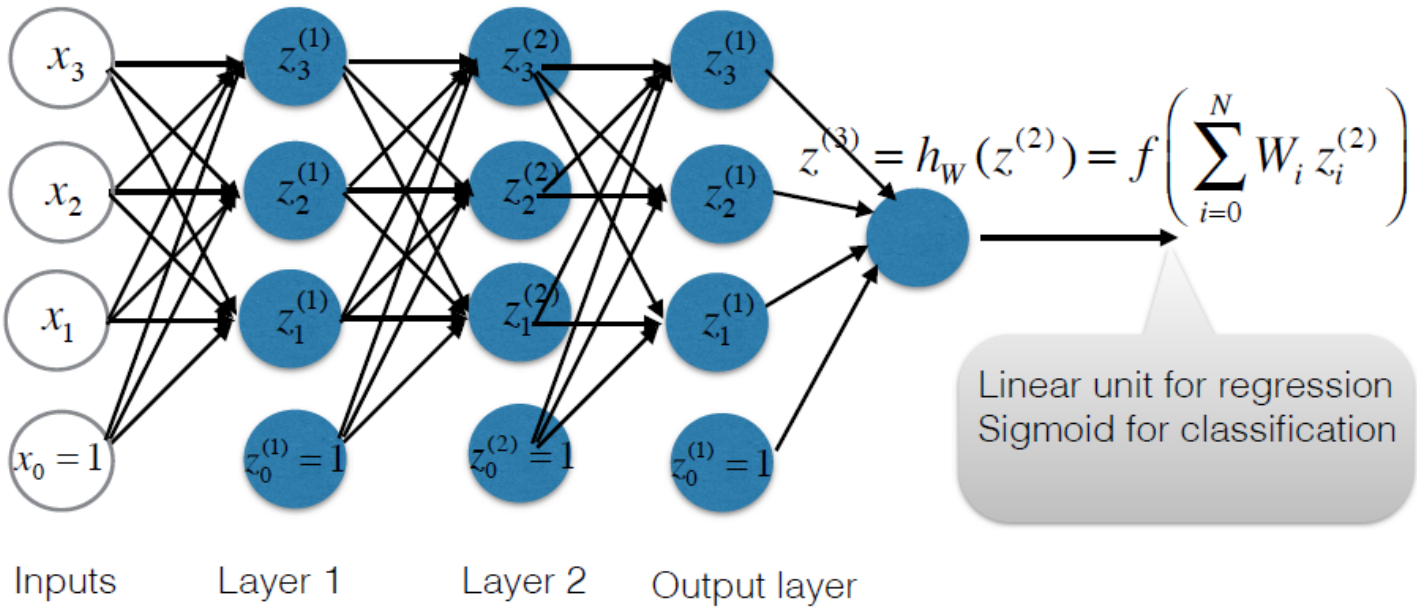
- **Feedforward Neural Network** is composed of layers of artificial neurons (perceptrons)



- This is a two-layer feedforward network

Deep Neural Networks

- **Feedforward Neural Network** is composed of layers of artificial neurons (perceptrons)



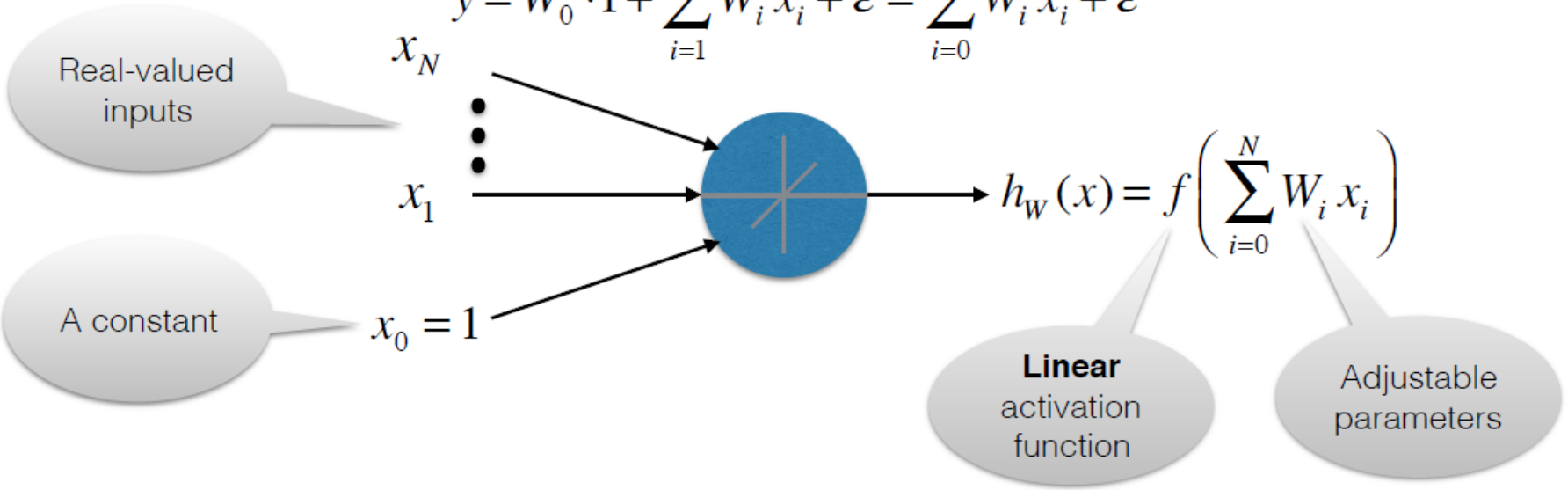
- More than 2 hidden layers = **Deep Neural Network (Deep Learning)**
- Trained by **Gradient Descent**

Linear Regressions as 1N Caclulation

Linear Regressions as 1N Caclulation

- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **linear function** of inputs
- Weights can be tuned to fit the data

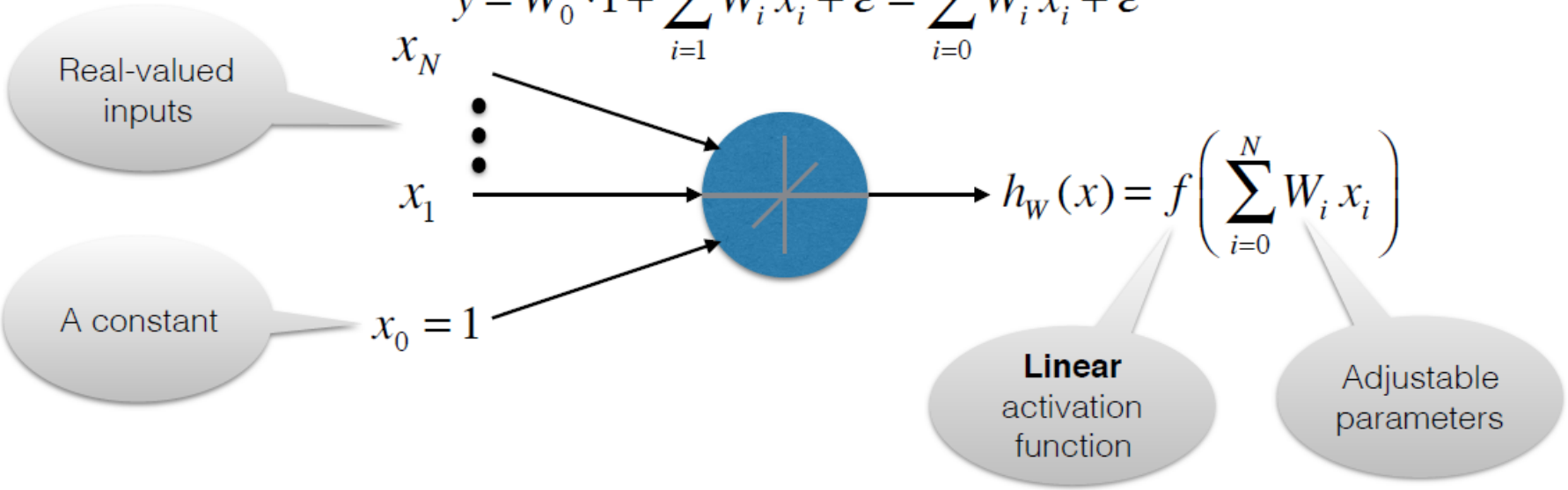
$$y = W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i + \varepsilon = \sum_{i=0}^{N} W_i x_i + \varepsilon$$



Linear Regressions as 1N Caclulation

- **Inputs**: real-valued numbers
- **Output:** a real-valued number
- Can be thought of as a node computing a **linear function** of inputs
- Weights can be tuned to fit the data

$$y = W_0 \cdot 1 + \sum_{i=1}^{N} W_i x_i + \varepsilon = \sum_{i=0}^{N} W_i x_i + \varepsilon$$

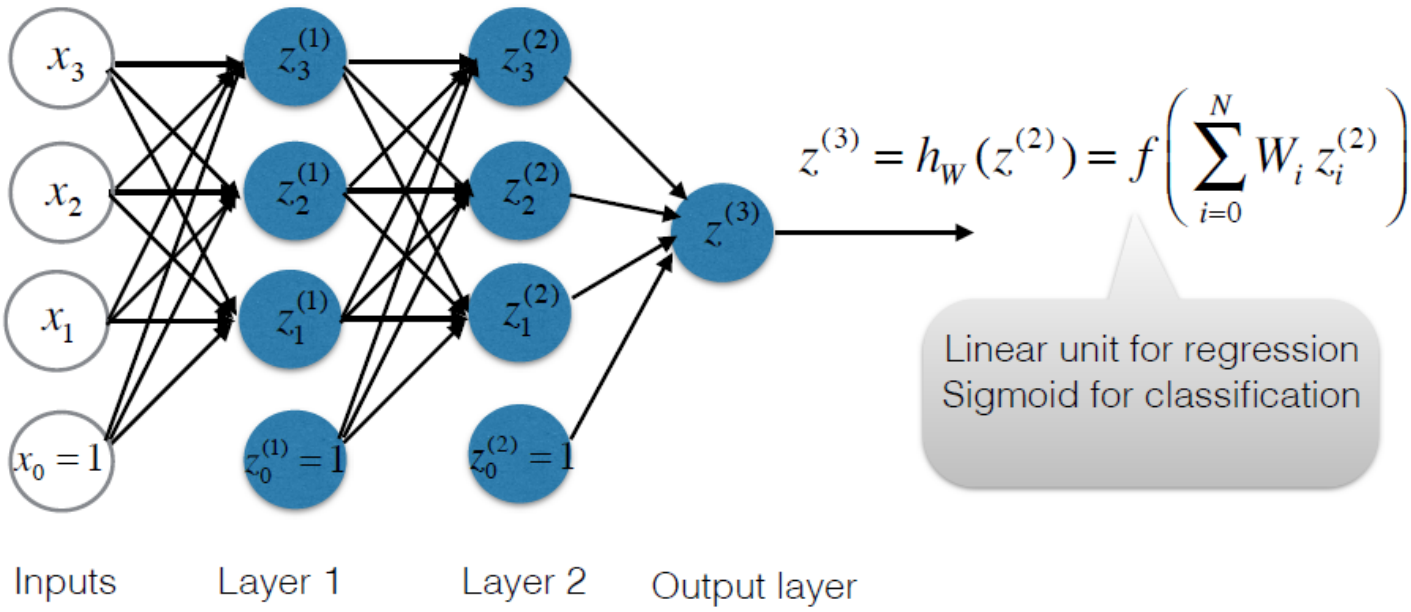


- For Linear Regression, there is an **analytical (one-step) solution** via the normal equation

Gradient Descent for NN

Gradient Descent for NN

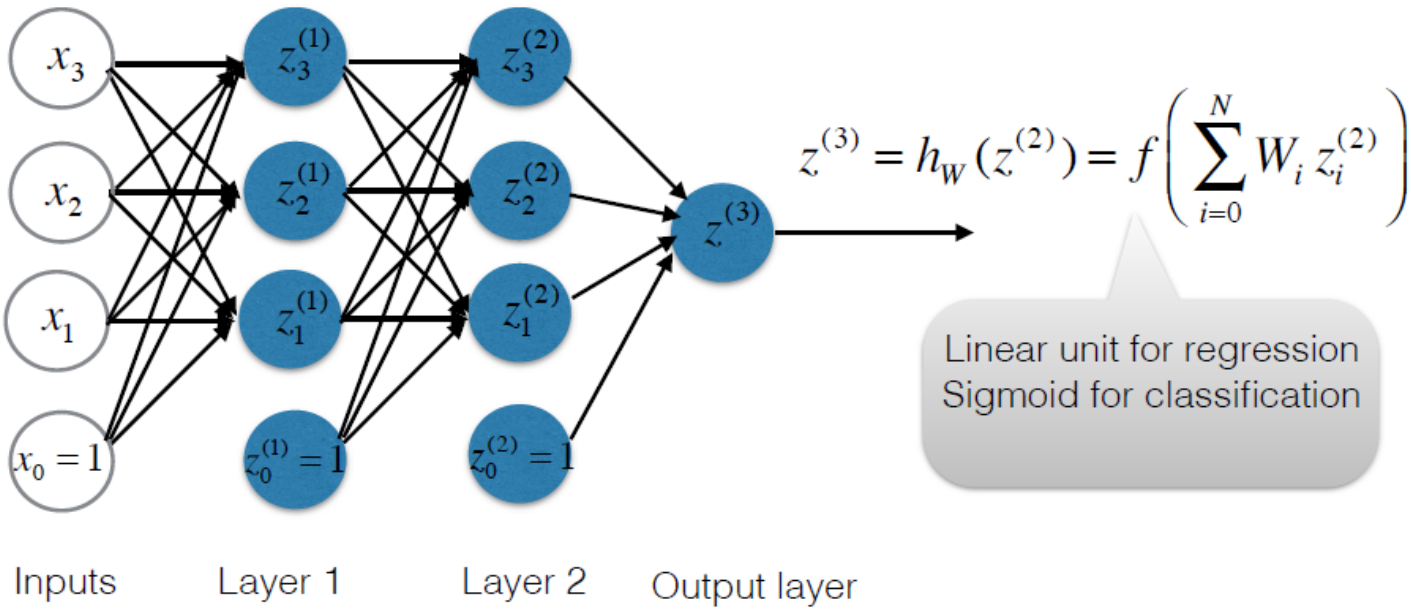
- Gradient descent can be apply not only to one neuron, but to a neural network too:



- Training by backpropagation = reverse-mode autodiff

Gradient Descent for NN

- Gradient descent can be applied not only to one neuron, but to a neural network too:



- Trained by **backpropagation** - a groundbreaking algorithm for neural networks proposed by Rumelhart, Hinton and Williams in 1986
- Backpropagation = Gradient Descent with a reverse-mode autodiff

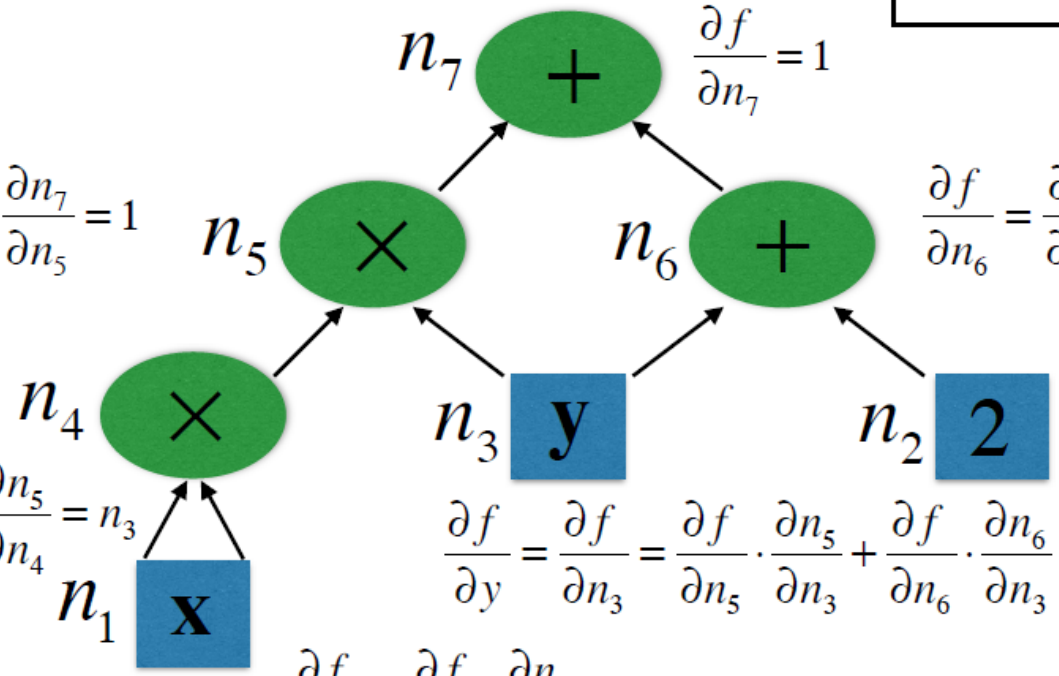
Reverse-Mode Autodiff in TF

Reverse-mode autodiff implements automatic derivatives of any functions:

- **Forward pass** (from inputs to outputs)
- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = f(n_i(x))$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \cdot \frac{\partial n_i}{\partial x}$$

$$f(x,y) = x^2 y + y + 2$$



$$\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \cdot \frac{\partial n_7}{\partial n_5} = \frac{\partial n_7}{\partial n_5} = 1$$

$$\frac{\partial f}{\partial n_6} = \frac{\partial f}{\partial n_7} \cdot \frac{\partial n_7}{\partial n_6} = \frac{\partial n_7}{\partial n_6} = 1$$

$$\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \cdot \frac{\partial n_5}{\partial n_4} = \frac{\partial n_5}{\partial n_4} = n_3$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial n_3} = \frac{\partial f}{\partial n_5} \cdot \frac{\partial n_5}{\partial n_3} + \frac{\partial f}{\partial n_6} \cdot \frac{\partial n_6}{\partial n_3} = n_4 + 1 = x^2 + 1$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_4} \cdot \frac{\partial n_4}{\partial x} = n_3 \cdot 2x = 2xy$$

Gradient Descent w/ Backpropagation

Gradient Descent w/ Backpropagation

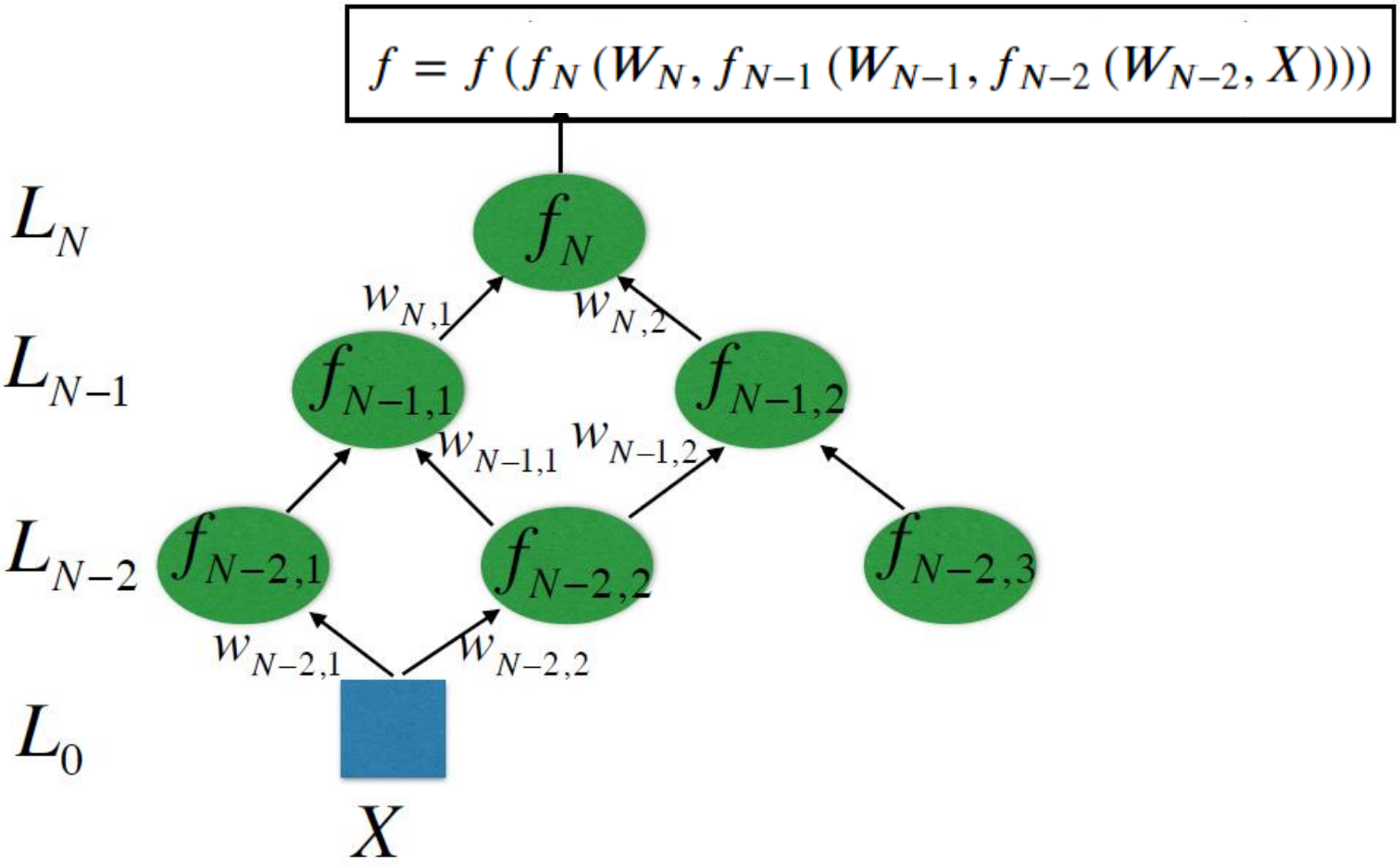
Gradient Descent with Backpropagation is similar to the reverse-mode autodiff:

- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = \text{MSE}_{train} = \frac{1}{N_{train}} \left\| \hat{\mathbf{Y}}^{train}(w) - \mathbf{Y}^{train} \right\|_2^2$

Gradient Descent w/ Backpropagation

Gradient Descent with Backpropagation is similar to the reverse-mode autodiff:

- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = \text{MSE}_{train} = \frac{1}{N_{train}} \left\| \hat{\mathbf{Y}}^{train}(w) - \mathbf{Y}^{train} \right\|_2^2$

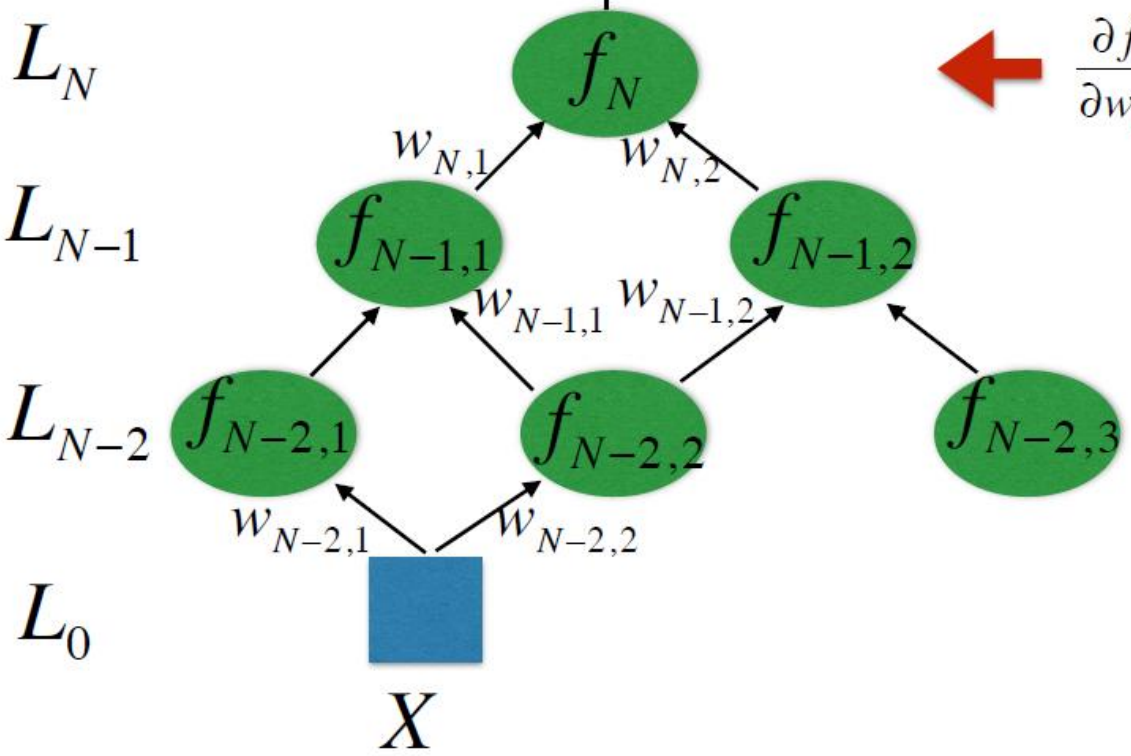


Gradient Descent w/ Backpropagation

Gradient Descent with Backpropagation is similar to the reverse-mode autodiff:

- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = \text{MSE}_{train} = \frac{1}{N_{train}} \left\| \hat{\mathbf{Y}}^{train}(w) - \mathbf{Y}^{train} \right\|_2^2$

$$f = f\left(f_N\left(W_N, f_{N-1}\left(W_{N-1}, f_{N-2}\left(W_{N-2}, X\right)\right)\right)\right)$$



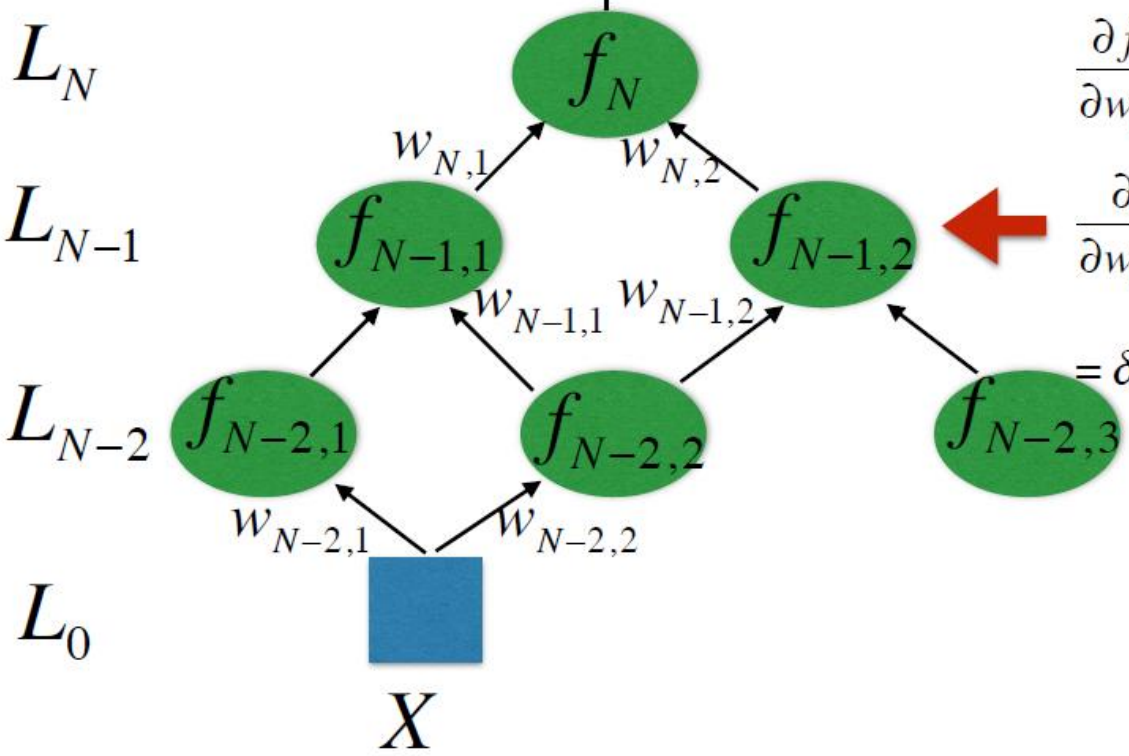
$$\frac{\partial f}{\partial w_{Ni}} = \frac{\partial f}{\partial f_N} \cdot \frac{\partial f_N}{\partial w_{Ni}} \equiv \delta_N \frac{\partial f_N}{\partial w_{Ni}}$$

Gradient Descent w/ Backpropagation

Gradient Descent with Backpropagation is similar to the reverse-mode autodiff:

- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = \text{MSE}_{train} = \frac{1}{N_{train}} \left\| \hat{\mathbf{Y}}^{train}(w) - \mathbf{Y}^{train} \right\|_2^2$

$$f = f\left(f_N\left(W_N, f_{N-1}\left(W_{N-1}, f_{N-2}\left(W_{N-2}, X\right)\right)\right)\right)$$



$$\frac{\partial f}{\partial w_{Ni}} = \frac{\partial f}{\partial f_N} \cdot \frac{\partial f_N}{\partial w_{Ni}} \equiv \delta_N \frac{\partial f_N}{\partial w_{Ni}}$$

$$\frac{\partial f}{\partial w_{N-1,i}} = \frac{\partial f}{\partial f_N} \frac{\partial f_N}{\partial f_{N-1,i}} \cdot \frac{\partial f_{N-1,i}}{\partial w_{N-1,i}}$$

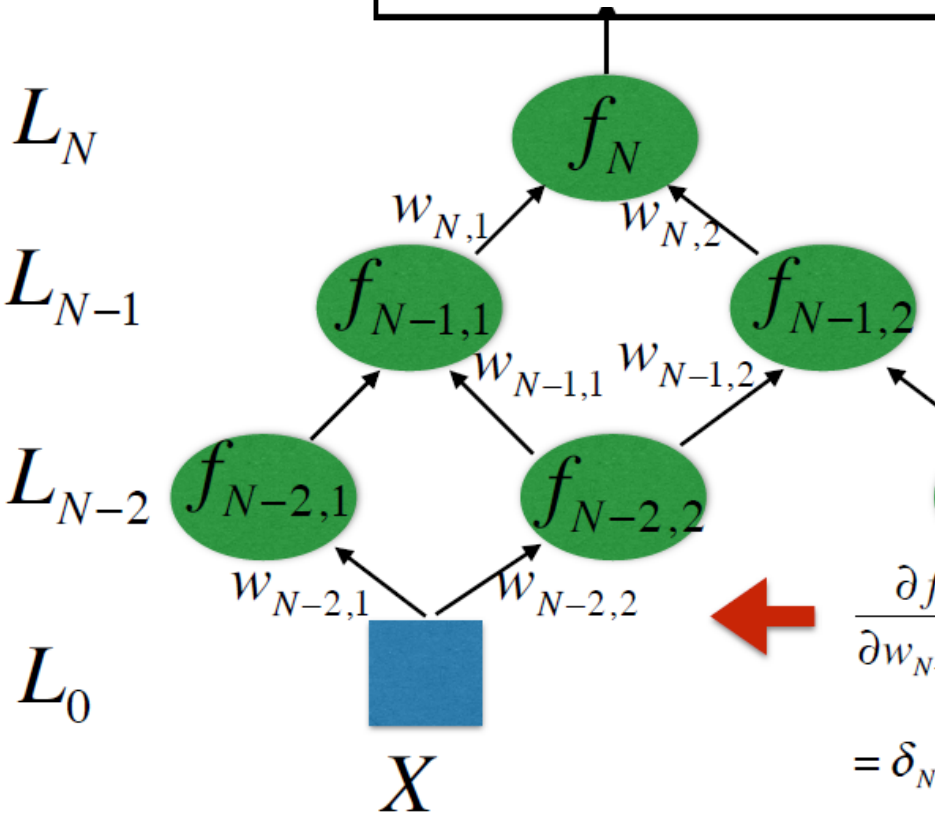
$$= \delta_{N-1,i} \frac{\partial f_{N-1,i}}{\partial w_{N-1,i}}, \quad \delta_{N-1,i} = \delta_N \frac{\partial f_N}{\partial f_{N-1,i}}$$

Gradient Descent w/ Backpropagation

Gradient Descent with Backpropagation is similar to the reverse-mode autodiff:

- **Backward pass** (from outputs to inputs)
- Use the chain rule for a composite function $f = \text{MSE}_{train} = \frac{1}{N_{train}} \left\| \hat{\mathbf{Y}}^{train}(w) - \mathbf{Y}^{train} \right\|_2^2$

$$f = f\left(f_N\left(W_N, f_{N-1}\left(W_{N-1}, f_{N-2}\left(W_{N-2}, X\right)\right)\right)\right)$$



$$\frac{\partial f}{\partial w_{Ni}} = \frac{\partial f}{\partial f_N} \cdot \frac{\partial f_N}{\partial w_{Ni}} \equiv \delta_N \frac{\partial f_N}{\partial w_{Ni}}$$

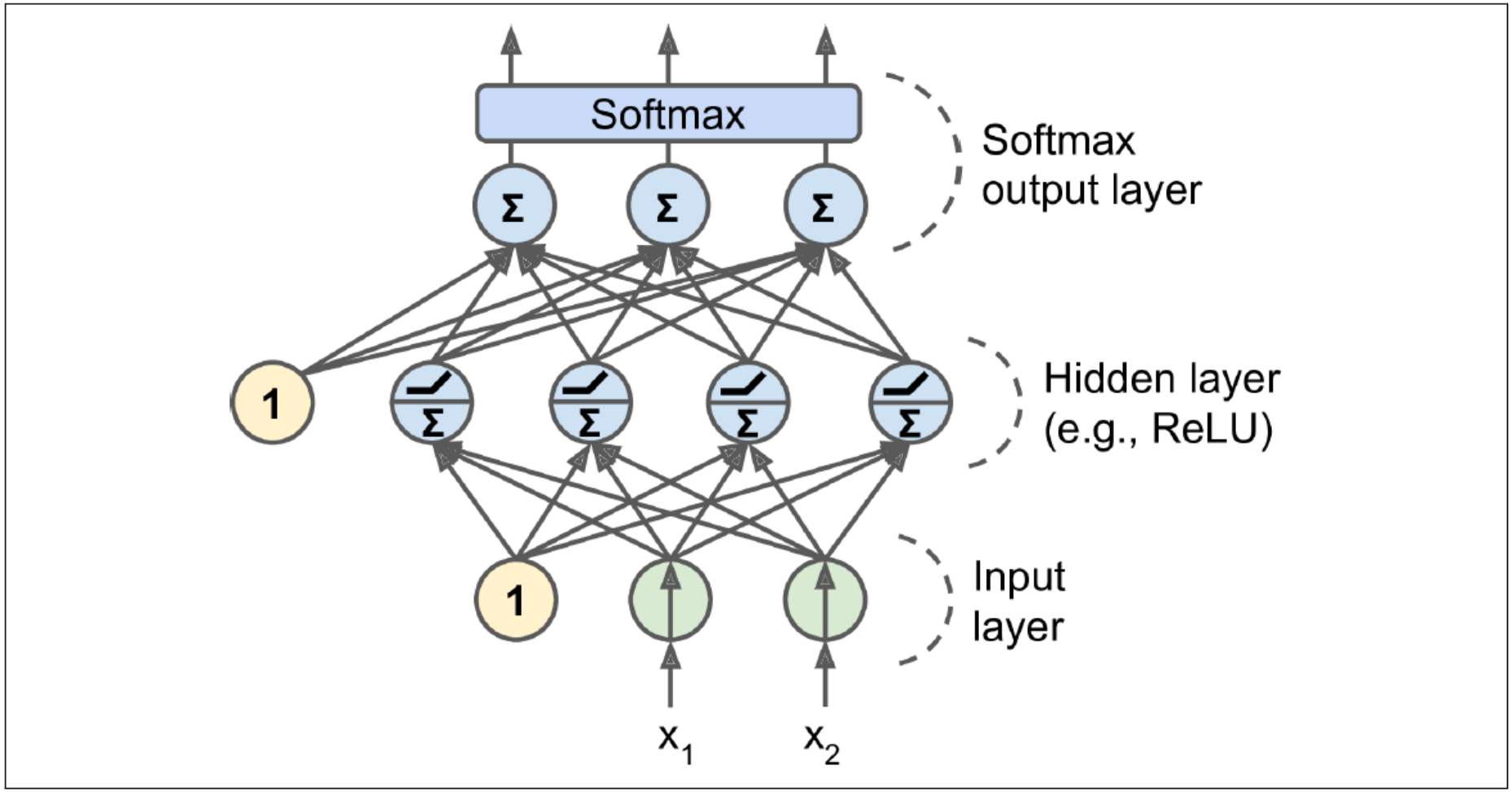
$$\frac{\partial f}{\partial w_{N-1,i}} = \frac{\partial f}{\partial f_N} \frac{\partial f_N}{\partial f_{N-1,i}} \cdot \frac{\partial f_{N-1,i}}{\partial w_{N-1,i}}$$

$$= \delta_{N-1,i} \frac{\partial f_{N-1,i}}{\partial w_{N-1,i}}, \quad \delta_{N-1,i} = \delta_N \frac{\partial f_N}{\partial f_{N-1,i}}$$

$$\frac{\partial f}{\partial w_{N-2,2}} = \sum_{i=1,2} \frac{\partial f}{\partial f_N} \frac{\partial f_N}{\partial f_{N-1,i}} \frac{\partial f_{N-1,i}}{\partial f_{N-2,2}} \cdot \frac{\partial f_{N-2,2}}{\partial w_{N-2,2}}$$

$$= \delta_{N-2,2} \frac{\partial f_{N-2,2}}{\partial w_{N-2,2}}, \quad \delta_{N-2,2} = \sum_{i=1,2} \delta_{N-1,i} \frac{\partial f_{N-1,i}}{\partial f_{N-2,i}}$$

MLP and Backpropagation



A modern MLP (including ReLU and softmax) for classification

ANN in TensorFlow Demo

ANN in TensorFlow Demo

A few digits from the MNIST dataset