Introduction to
Automatic Adjoint Differentiation
- AAD -
in Machine Learning
and Finance

Antoine Savine

Modern Computational Finance

http://antoinesavine.com

Find the slides and examples here:

go to http://github.com/asavine

click on the repo 'CompFinance'

download slides 'Intro2AADinMachineLearningAndFinance.pdf'
download examples in the 'Workshop' folder
see tutorial in the 'xlCpp' folder for exporting C++ code to Excel
see basic AAD code in C++ in the file 'toyCode.h'
the rest of the repo is companion code for the book Modern Computational Finance

follow asavine on GitHub and watch the CompFinance repo to be notified of updates

Introduction

Adjoint Differentiation - AD -

- Algorithm to compute all differentials
  of a scalar function quickly and accurately

  - Quickly (*constant time*):
    Compute all differentials in a time
    similar to one function evaluation

  - Accurately:
    Analytic differentiation,
    accurate to machine precision

$$f: \quad \mathbb{R}^D \quad \rightarrow \quad \mathbb{R}$$

$$x = \begin{pmatrix} x_1 \\ ... \\ x_D \end{pmatrix} \quad \rightarrow \quad y = f(x)$$

$$AD : compute : \frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_D} \right) \text{ in } O(1)$$

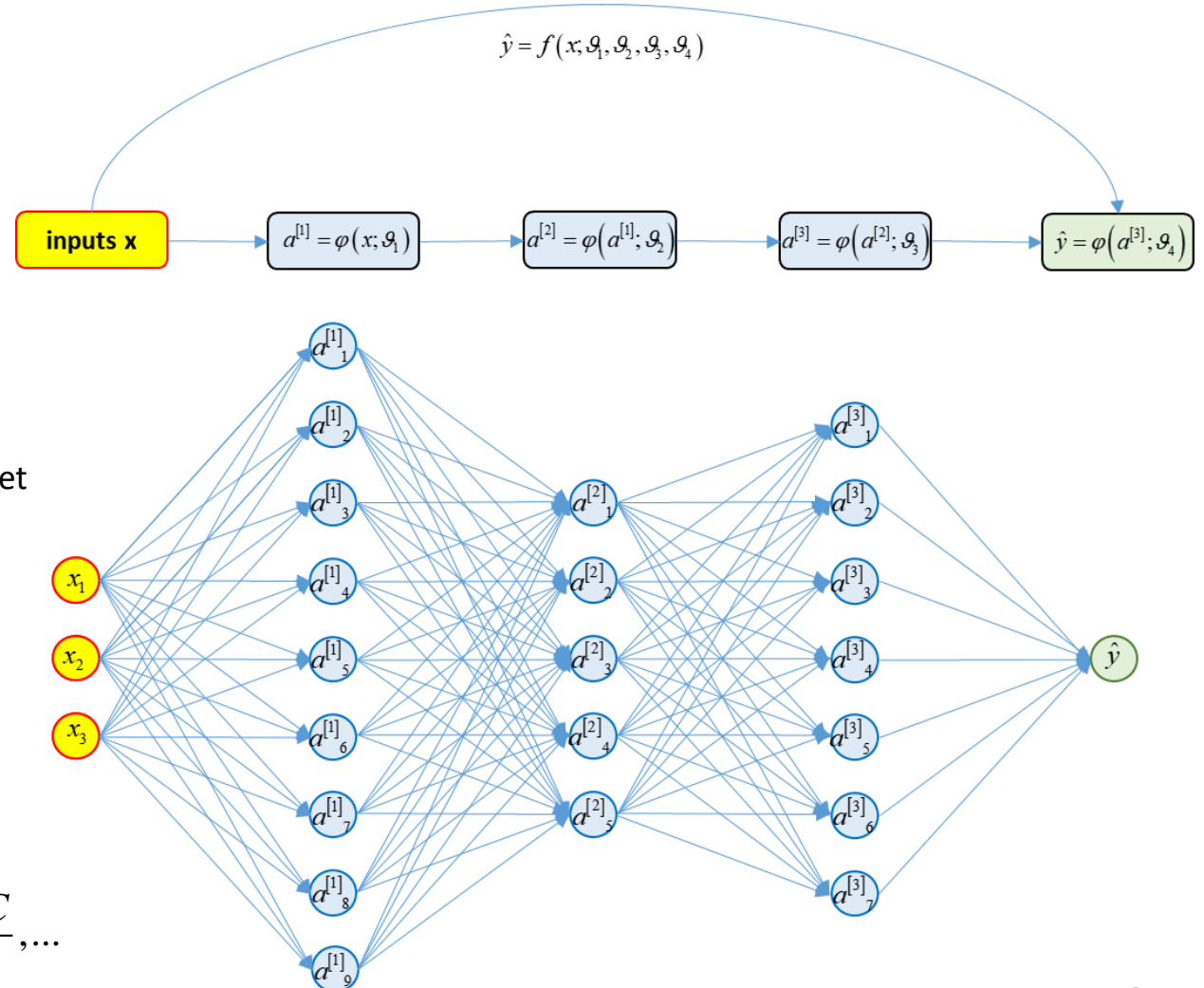- Application:
  whenever we need to compute many derivatives of one result

- AD is *only* about speed, yet a massive difference in Machine Learning and Finance – without AD:

  - Deep ANNs could not learn their parameters in reasonable time

  - Financial risks could not be computed in real time

- See article on QuantMinds:
  https://knect365.com/quantminds/article/2d9e3329-cae1-4320-86a2-a4e2303b1bfe/a-brief-introduction-to-automatic-adjoint-differentiation-aad

Application 1: model fitting

- Model: $\hat{y} = f(x; \vartheta)$

  - Makes a prediction $\hat{y}$ from a set of inputs $x = (x_1, ..., x_n)^T$

  - Given parameters $\vartheta = (\vartheta_1, ..., \vartheta_D)^T$

- Learns its parameters by (iteratively) minimizing a *cost* function

  - E.g. sum of squared errors: $C(\vartheta) = \sum_{i=1}^{m} \left[f(x^{(i)}; \vartheta) - y^{(i)} \right]^2 = \sum_{i=1}^{m} c_i(\vartheta)$ (c_i : error or *loss* on training example i)

  - Over a training set of *m* labelled examples: $\left(x^{(i)}, y^{(i)} \right)_{1 \leq i \leq m}$ ($x^{(i)} \in \mathbb{R}^n$, $y^{(i)} \in \mathbb{R}$)

- On each iteration, we must compute the gradient of cost to parameters: $\dfrac{\partial C}{\partial \vartheta} = \left(\dfrac{\partial C}{\partial \vartheta_1}, ..., \dfrac{\partial C}{\partial \vartheta_D} \right)$

  - Standard case C = sum of squared errors: $\dfrac{\partial C}{\partial \vartheta} = \sum_{i=1}^{m} \dfrac{\partial c_i}{\partial \vartheta}$, $\dfrac{\partial c_i}{\partial \vartheta} = 2 c_i \dfrac{\partial f(x_i; \vartheta)}{\partial \vartheta}$

- Predict target *y* from inputs *x*:
  Sequence of parametric functions
  of the same shape, with different parameters

- Training set:
  m inputs $x^{(i)}$, each a vector in dimension n_0
  with corresponding (scalar) labels $y^{(i)}$

- Train the network: learn (fit) parameters $\vartheta_1, \vartheta_2, \vartheta_3, \ldots$
  to minimise a *cost* = prediction errors on the training set

- Then apply trained model
  to make predictions on new examples

- Training requires an iterative optimization algorithm

- On each iteration, we must compute the cost *C*

- And its derivatives *to all the parameters* $\dfrac{\partial C}{\vartheta_1}, \dfrac{\partial C}{\vartheta_2}, \dfrac{\partial C}{\vartheta_3}, \ldots$

$$\hat{y} = f(x; \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$$

inputs x $\rightarrow$ $a^{[1]} = \varphi(x; \vartheta_1)$ $\rightarrow$ $a^{[2]} = \varphi(a^{[1]}; \vartheta_2)$ $\rightarrow$ $a^{[3]} = \varphi(a^{[2]}; \vartheta_3)$ $\rightarrow$ $\hat{y} = \varphi(a^{[3]}; \vartheta_4)$

Finance example: calibration of pricing models

- Pricing model: $\hat{y} = \text{product value} = f\left(\text{product parameters } x; \text{model parameters } \vartheta\right)$

- For example: *x = (strike, expiry, …)* and θ *= (volatilities, …)*

- Calibration set: *m* products, each with known features $x^{(i)}$ and observed market price $y^{(i)}$

- Calibration: fit parameters $\vartheta = \left(\vartheta_1, ..., \vartheta_D\right)^T$ to minimise cost *C* = (sum of squared) pricing errors

- Then, apply calibrated model to price other products, which values are unknown

- Iterative optimization (eg. Levenberg-Marquardt: see free ebook Introduction to Applied Linear Algebra by Boyd & Vandenberghe)

- On each iteration, we must compute cost *C*, pricing all calibration instruments

- And its derivatives $\dfrac{\partial C}{\partial \vartheta} = \left(\dfrac{\partial C}{\partial \vartheta_1}, ..., \dfrac{\partial C}{\partial \vartheta_D}\right)$ to all the parameters

Two comments on calibration

1. Calibration is considered best practice in financial derivatives (as opposed to estimation of parameters)

To understand exactly why, see

https://www.slideshare.net/AntoineSavine/60-years-birthday-30-years-of-ground-breaking-innovation-a-tribute-to-bruno-dupire-by-antoine-savine

2. Calibration is only similar to training ML models in appearance

- ML models are *entirely* trained on data and make predictions on new examples *drawn from the same distribution*
- "Interpolation" problem

- Financial models have a strong structure: arbitrage-free, realistic dynamics, some parameters (like correlation) typically estimated
- Often calibrated on European options to value more complicated *exotic* options
- "Extrapolation" problem

Application 2: model risk

- Calibrated Pricing model: $value = f\left(\text{product features } x; \text{model parameters } \vartheta\right)$

- Model parameters:
  - Todays underlying asset prices
  - Volatilities
  - Correlations, Mean-reversions, Volatility of volatility
  - Etc.

- Model risks: $\dfrac{\partial f}{\partial \vartheta}$ with potentially massive number of parameters

  - Black-Scholes: spot, volatility, rate, dividend yield
  - Dupire: two-dimensional local volatility surface, + with stochastic volatility: vol of vol, correlation spot/vol
  - Interest Rate Models: initial yield curve + spread curve(s) + volatility curve or surface (+ correl, mean-reversion…)
  - Multi-currency models (Baskets, CVA…): all of the above for all currencies + forex rates and their volatilities + giant correlation matrix
  - Routinely in the thousands or tens of thousands
  - Valuation generally expensive: typically, Monte-Carlo simulations

Model and market risk

- Usually, we are not interested in risks to model parameters but risks to market variables
E.g. Dupire: not local volatilities but implied volatilities
Model parameters are generally calibrated to market variables

  a: vector of k market variables (underlying asset prices, rates, spreads, implied volatilities…)
  b: vector of n model parameters (calibrated to market variables so $b = c(a), c : \mathbb{R}^k \to \mathbb{R}^n$)
  g: valuation function of model (generally numerical such as Monte-Carlo simulation) $V = g(b), g : \mathbb{R}^n \to \mathbb{R}$
  f: valuation function of market $V = g(b) = g[c(a)] = f(a), f : \mathbb{R}^k \to \mathbb{R}$

- Risk report computes model risks $\dfrac{\partial g}{\partial b}$ and/or market risks $\dfrac{\partial f}{\partial a}$

- Market risks computed from model risks: $a \in \mathbb{R}^k \xrightarrow{c} b \in \mathbb{R}^n \xrightarrow{g} v \in \mathbb{R}$ so by the chain rule $\dfrac{\partial f}{\partial a} = \dfrac{\partial g}{\partial b}\dfrac{\partial c}{\partial a}$

  - To compute $\dfrac{\partial c}{\partial a}$ and market risks $\dfrac{\partial f}{\partial a} = \dfrac{\partial g}{\partial b}\dfrac{\partial c}{\partial a}$ knowing model risks $\dfrac{\partial g}{\partial b}$, see for instance:
    http://papers.ssrn.com/sol3/papers.cfm?abstract_id=3262571

  - In this presentation, we focus on model risks $\dfrac{\partial g}{\partial b}$

Differentiation

- In all these applications (and many others)
  - We have a scalar function *f* of many inputs $f : \mathbb{R}^D \to \mathbb{R}$
  - This function is typically expensive to evaluate
    Losses of deep neural nets on large training set or a Monte-Carlo valuation may take several seconds, even on parallel hardware
  - It typically takes a large number *D* of inputs
    Thousands of weights for deep neural nets, or thousands of market/model variables for pricing
  - We must compute its sensitivities to all its inputs $\dfrac{\partial f}{\partial x} = \left(\dfrac{\partial f}{\partial x_1}, \dfrac{\partial f}{\partial x_2}, ..., \dfrac{\partial f}{\partial x_D} \right)$

- Conventional differentiation is of linear complexity in the number of inputs/differentials
  - To compute **one** differential takes time
    similar to **one** evaluation of the function
  - For example, with (one-sided) finite differences $\dfrac{\partial f}{\partial x_i} \approx \dfrac{f(x + \varepsilon e_i) - f(x)}{\varepsilon}, e_i = \left(\delta_{ij} \right)_{1 \le j \le n}$
  - We have *(D+1)* function evaluations
    to compute *D* differentials
  - Illustration: if one function evaluation takes 1sec and we have 1,000 inputs
    It takes 1,001sec (15+ minutes) to evaluate its gradient once
  - This is not viable, a different technology is necessary

Adjoint Differentiation - AD -

- Algorithm to compute analytically all the differentials of f: $\dfrac{\partial f}{\partial x} = \left(\dfrac{\partial f}{\partial x_1}, \dfrac{\partial f}{\partial x_2}, ..., \dfrac{\partial f}{\partial x_D} \right)$
  - In a time independent of D
  - With an efficient implementation, it takes 4 to 10 times one evaluation of f to compute all its differentials
  - So AD can compute the 1,000 differentials of a function that takes 1sec in ~4 to 10sec as opposed to 15min+
  - Without loss of accuracy (if anything, AD differentials are *more* accurate)

- Hence the importance of this technology today
  - In finance, AD computes thousands of complex risks accurately and very quickly
  - In machine learning, AD computes the gradient of cost functions to many parameters in constant time
  - The same technology powers banks to estimate financial risks and deep nets to learn their weights
  - AD
    - Allows Banks to compute massive amount of regulatory risks in real time
    - Powers your telephones to learn to recognize you and your friends in reasonable time
    - Is largely credited for the recent and spectacular successes in the field of deep learning
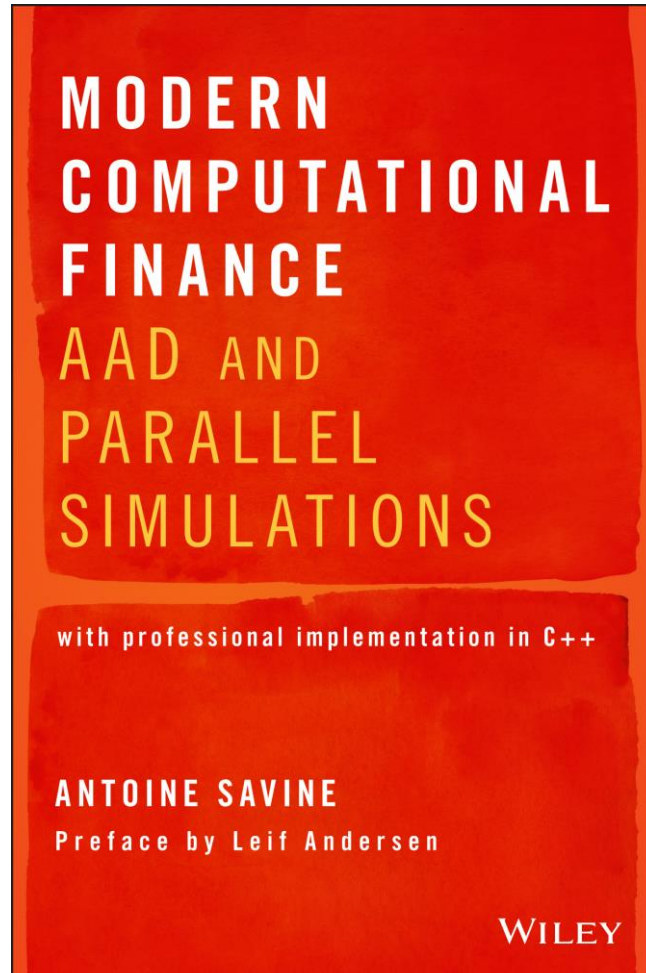
A brief history of AD

- Invented in the 1960s (Wengert, 1964)

- Called "holy grail" of sensitivity computation (Griewank, 2012)

- Classified as one of the 30 greatest numerical algorithms of the 20th century (Trefethen, 2015)

- Did not take firm hold in the computer science community until the 1980s

- Applied in deep learning to efficiently compute gradients of cost functions in learning algorithms

- Known under the names
  Back-Propagation (or simply back-prop, in deep learning),
  reverse differentiation, backward differentiation, adjoint accumulation, algorithmic differentiation, etc.

- Not adopted in Finance until 2006 (Giles and Glasserman's "Smoking Adjoints")

- Large scale implementation in Danske Bank with parallel Monte-Carlo simulations won In-House System of the Year 2015 Risk Award

- Widely adopted today for risk and calibration

- Delays in adoption mainly due to complexity, lack of teaching material and challenges in practical implementation (Andersen, 2018)

# Automatic Adjoint Differentiation - AAD -

- Automatic implementation of AD
  - Developers only produce the evaluation code for the function f
  - The framework automatically produces (constant time) differentiation "code" for $\partial f / \partial x$
  - So developers don't need to write (complicated and error prone) AD code
  - And differentiation code is automatically updated when evaluation code changes

- Many commercial and open source AAD frameworks exist
  - Some are generic, some specialize in finance or machine learning
  - Some are free, some as extremely expensive
  - Work in a variety of programming languages: Python, C++, …
  - Work either with source transformation, operator overloading or by letting clients build evaluation graphs

- Example: TensorFlow, one of most popular frameworks in machine learning
  - Written in C++/CUDA (multicore CPU/GPU)
  - With APIs in Python, JavaScript, Java, Go, Swift…
  - Allow clients to create evaluation graphs
  - TensorFlow evaluates the graphs and automatically applies AD to compute derivatives through the graphs

AAD book

MODERN COMPUTATIONAL FINANCE

AAD AND PARALLEL SIMULATIONS

with professional implementation in C++

ANTOINE SAVINE
Preface by Leif Andersen

WILEY

- Written by (some of) the people who wrote Danske Bank's award winning systems
- Prefaced by Leif Andersen, read preface on

  researchgate.net/publication/328042479_Modern_Computational_Finance_AAD_and_Parallel_Simulations

- Teaches:
  - The modern design and efficient implementation of financial simulation libraries
  - The implementation of *parallel* simulation libraries
  - And of course AAD

- AAD is covered in deep detail:
  - Conceptual and mathematical foundations
  - Practical implementation
  - Memory management and check-pointing
  - Application in the context of large Monte-Carlo simulations, including in parallel
  - Application to model and market risks
  - Cutting-edge implementation with meta-programming and expression templates
  - And much more

- Ships with complete, professional code in C++
  - Explained in deep detail in the book
  - Freely available on the GitHub repo http://www.github.com/asavine/CompFinance/wiki

AAD book reviews
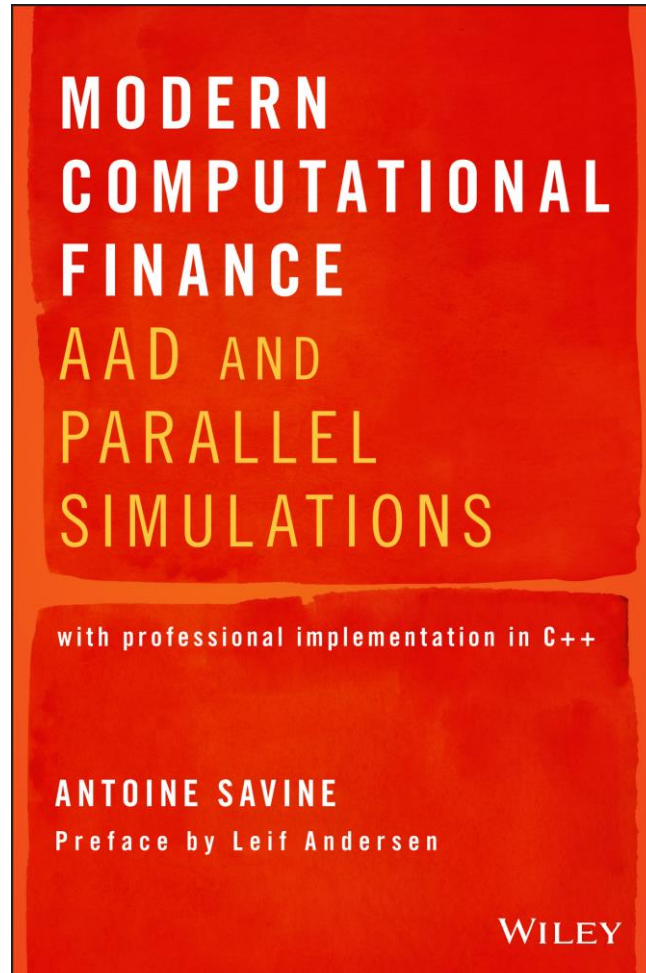
It would not be much of an exaggeration to say that Antoine Savine's book ranks as the 21st century peer to Merton's 'Continuous-Time Finance'.

Vladimir Piterbarg

This book [...] addresses the challenges of AAD head on. [...] The exposition is [...] ideal for a Finance audience. The conceptual, mathematical, and computational ideas behind AAD are patiently developed in a step-by-step manner, where the many brain-twisting aspects of AAD are de-mystified. For real-life application projects, the book is loaded with modern C++ code and battle-tested advice on how to get AAD to run for real. [...] Start reading!

Leif Andersen

An indispensable resource for any quant. Written by experts in the field and filled with practical examples and industry insights that are hard to find elsewhere, the book sets a new standard for computational finance.

Paul Glasserman

Overview

AAD: Demonstration

1. Adjoint Differentiation for Deep Learning
    1. A brief introduction to Artificial Neural Networks (ANN)
    2. Back-Propagation through ANNs

2. Adjoint Differentiation for arbitrary calculations
    1. Evaluation graphs
    2. Back-Propagation through evaluation graphs

3. Automatic Adjoint Differentiation for financial derivatives in (simple) C++
    1. AAD with operator overloading
    2. AAD over Monte-Carlo simulations

Demonstration: Dupire's model (1992)

- Extended Black & Scholes dynamics (in the absence of rates, dividends etc.): $\dfrac{dS}{S} = \sigma(S,t)\,dW$

- Calibrated with Dupire's celebrated formula: $\sigma(K,T) = \dfrac{2\dfrac{\partial C}{\partial T}}{\dfrac{\partial^2 C}{\partial K^2}}$ with *C(K,T)* = call prices of strike *K*, maturity *T*

- Implemented with a (bi-linearly interpolated) local volatility matrix: $\sigma_{ij} = \sigma(S_i, T_j)$

- Volatility matrix: 21 spots (every 5 points 50 to 200) and 36 times (every month from now to 3y)
  we have 1,080 volatilities + 1 initial spot (=100) = 1,081 model parameters

- Valuation of a 3y (weekly monitored) barrier option strike *K=120*, barrier *B=150*: $v(S_t,t) = E\left[(S_T - K)^+ 1_{\{\max(S_{T_1},\ldots,S_{T_K})<B\}} \middle| S_t \right]$

- Solved with Monte-Carlo or FDM over the equivalent PDE (from Feynman-Kac's theorem)

- We focus on Monte-Carlo simulations here: 500,000 paths, 156 (weekly) time steps

Demonstration: Results

- Implementation
  - C++ code exported to Excel (see **tutorial ExportingCpp2xl.pdf** on the GitHub repo, folder **xlCpp**)
  - Generic library design with efficient implementation (Chapter 6)
  - Parallel implementation (Chapters 3 and 7)
  - Sobol quasi-random numbers (Chapters 5 and 6), excerpt here: https://medium.com/@antoine_savine/sobol-sequence-explained-188f422b246b
  - Advanced AAD with expression templates (Chapter 15)

- Hardware: quad-core laptop (surface book 2, 2017)

- Performance:
  - One evaluation with 500,000 paths over 156 time steps take ~0.8sec
  - We have 1,081 risk sensitivities, take about 15 minutes to produces model risk report with linear differentiation
  - With AAD the 1,081 differentials are produced in ~1.5sec

Demonstration material

- GitHub repo (http://github.com/asavine/CompFinance)

- Prebuilt Xl addin **xlComp.xll** (may need to install redistributables VC_redist.x86.exe and VC_redist.x64.exe)

- Demonstration spreadsheet **xlTest.xlxs**

Deep Learning

Neural networks and deep learning

- Adjoint differentiation is best known and explained in the context of deep learning

- We therefore *briefly* introduce deep learning

- (Much) deeper learning material is found, for example, in:
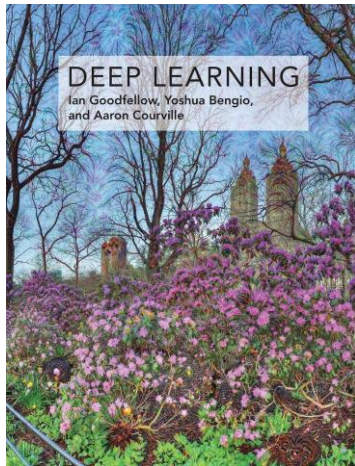
Goodfellow's reference book
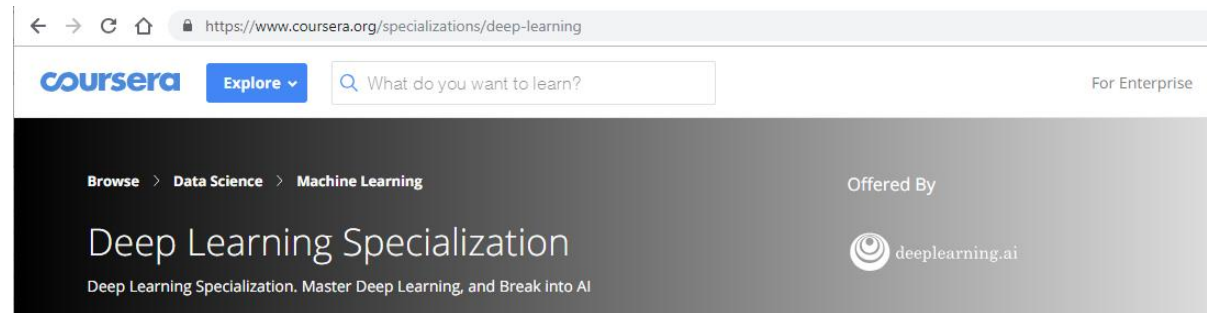
Andrew Ng's videos on Coursera
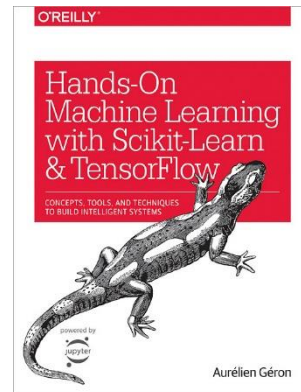
Geron's practical introduction
with TensorFlow

...and many other books and resources

Linear regression

- Linear model (joint Gaussian assumptions): $\hat{y} = E[y|x] = b + \sum_{i=1}^{n} w_i x_i = b + w^T x$

$$\hat{y} = b + w^T x$$

- Parameters: $b \in \mathbb{R}$ and $w \in \mathbb{R}^n$
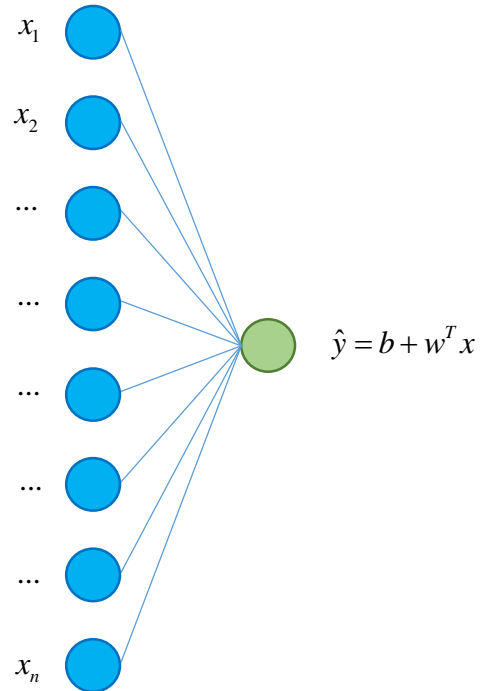
Linear regression: example

- In *regression* problems we predict real numbers

Example: predict return from volatility

- Each point is a stock in the S&P500
- Horizontal axis is annual volatility
- Vertical axis is annual return
- Estimated with daily data, 2013-2018
- Evidence of (small) risk premium
- Poor regression quality

S&P500 stocks 2013-2018

$y = 0.11x + 11.23$
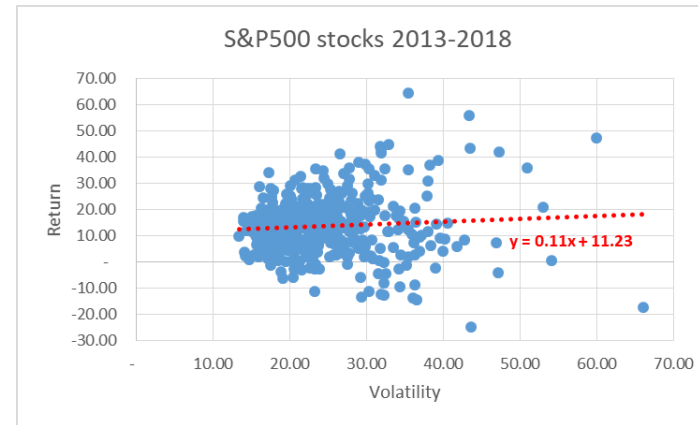
Linear classification

- Alternatively, *classification* problems predict discrete categories

Example: identify animals in pictures: 0: not an animal, 1: cat, 2: dog, 3: bird, 4: other animal

picture 1920x1080x24bit colour

vector of 1920x1080 pixels
$\in \left[0, 2^{24}-1\right]$

$$x = \begin{pmatrix} x_1 \\ ... \\ x_{2,073,600} \end{pmatrix}$$

softmax regression

$$\hat{y} = s(b + wx) = \begin{bmatrix} \Pr(\text{no animal}) \\ \Pr(\text{cat}) \\ \Pr(\text{dog}) \\ \Pr(\text{bird}) \\ \Pr(\text{other animal}) \end{bmatrix}$$

Parameters:
b: vector of dimension 5
w: matrix 5 x 2,073,600

s: softmax function
$s : \mathbb{R}^5 \to (0,1)^5$ summing to 1

$$s(z) = \left[\frac{e^{z_i}}{\sum e^{z_j}} \right]$$

- Regression more relevant than classification in quantitative finance
- Deep learning community generally focuses more on classification
- See literature for details, for example Stanford's CS229 on http://cs229.stanford.edu/syllabus.html
- This presentation sticks with regression
- Everything that follows generalises easily to classification and the maths remain essentially identical

Linear fit

- Training set of m examples $x^{(1)}, \dots, x^{(m)}$ (each a vector in dimension n) with corresponding labels $y^{(1)}, \dots, y^{(m)} \in \mathbb{R}$

- Learn parameters: $b \in \mathbb{R}$ and $w \in \mathbb{R}^n$ by minimizing prediction errors (cost function) $C(b,w) = \sum_{i=1}^{m} \left(\underbrace{b + w^T x^{(i)}}_{= \hat{y}^{(i)}} - y^{(i)} \right)^2$

- Note that this is the same as maximizing (log) likelihood under Gaussian assumptions, hence:

$$b^*, w^* = \arg\min C(b,w)$$ are the maximum likelihood estimators (MLE) of the parameters

- b^* and w^* are found analytically, solving for $\dfrac{\partial C}{\partial b} = 0$ and $\dfrac{\partial C}{\partial w} = 0$

- Result ("normal equation"): $\begin{pmatrix} b \\ w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} = \left(X^T X \right)^{-1} X^T Y$ where $X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \dots & x_n^{(2)} \\ \dots & x_1^{(i)} & x_j^{(i)} & x_n^{(i)} \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}$ and $Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(m)} \end{pmatrix}$

Linear limitations

- Important revaluation example:

  - Predict the future price in 1y of a European call strike 100, maturity 2y

  - By regression of the payoff in 2y: $\quad y^{(i)} = \left(S_{2y}^{(i)} - K \right)^{+}$

  - Over the underlying asset price in 1y: $\quad x^{(i)} = S_{1y}^{(i)}$

  - With a training set of m paths $\quad \left(S_{1y}^{(i)}, S_{2y}^{(i)} \right)_{1 \le i \le m}$

    generated under Black & Scholes' model (spot = 100, volatility of 20%)

  - We know the exact solution is given by Black and Scholes' formula so we can assess the quality of the regression
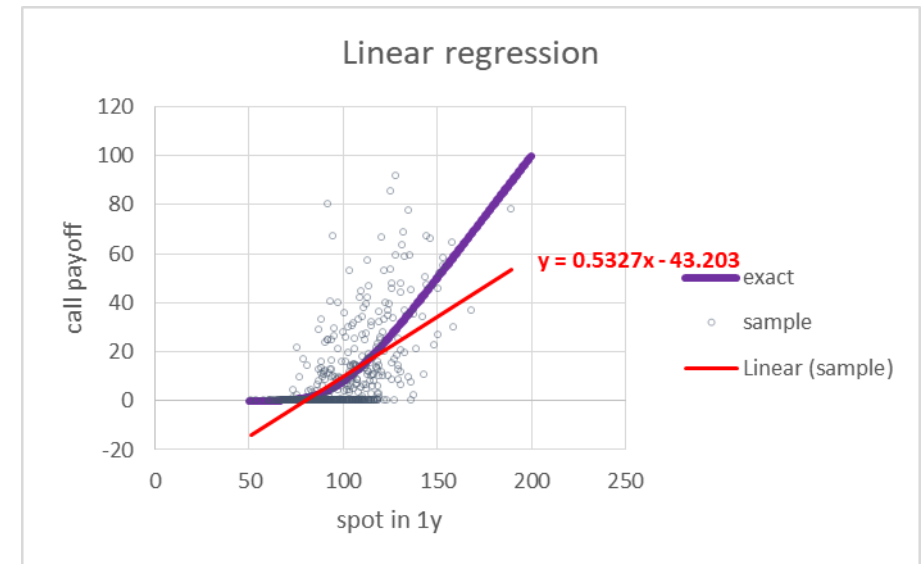
  - See spreadsheet reval.xlsx on the repo http://github.com/asavine/CompFinance/Workshop/

  - Linear regression obviously fails to approximate the correct function because it cannot capture non-linearities

Linear regression

$y = 0.5327x - 43.203$

Basis function regression

- Solution: regress not on *x* but on basis functions *a* of *x*

- Example: polynomial regression
  - Basis functions = monomials (*x* is in dimension n_0):
    - 1st degree: all the xs $a_i = x_i$
    - 2nd degree: all the squares $a_i = x_j^2$ and pair-wise products $a_i = x_i x_j$
    - 3rd degree: all the cubes $a_i = x_j^3$ and pair-wise $a_i = x_j x_k^2$ and triplet-wise $a_i = x_j x_k x_l$
    - Etc.

- Prediction in two steps:
  - Start with the vector *x* of n_0 features
  - Compute vector of n_1 basis functions: $a = \varphi(x)$
  - Predict linearly in the basis functions: $\hat{y} = b + w^T a$

- Training: identical to linear regression on *a* in place of *x*

$$\begin{pmatrix} b \\ w_1 \\ w_2 \\ ... \\ w_{n_1} \end{pmatrix} = \left(A^T A \right)^{-1} A^T Y \quad \text{where} \quad A = \begin{pmatrix} 1 & a_1^{(1)} & ... & a_{n_1}^{(1)} \\ 1 & a_1^{(2)} & ... & a_{n_1}^{(2)} \\ ... & a_1^{(i)} & a_j^{(i)} & a_{n_1}^{(i)} \\ 1 & a_1^{(m)} & ... & a_{n_1}^{(m)} \end{pmatrix} \quad \text{and} \quad a^{(i)} = \varphi\left(x^{(i)} \right)$$

hidden layer
= basis functions

$a_1 = x_1$
$a_2 = x_2$
$a_3 = x_3$
$a_4 = x_1 x_2$
$a_5 = x_1 x_3$
$a_6 = x_2 x_3$
$a_7 = x_1^2$
$a_8 = x_2^2$
$a_9 = x_3^2$

x_1
x_2
x_3

output layer
= prediction

$\hat{y} = b + w^T a$

original features
= **input layer**
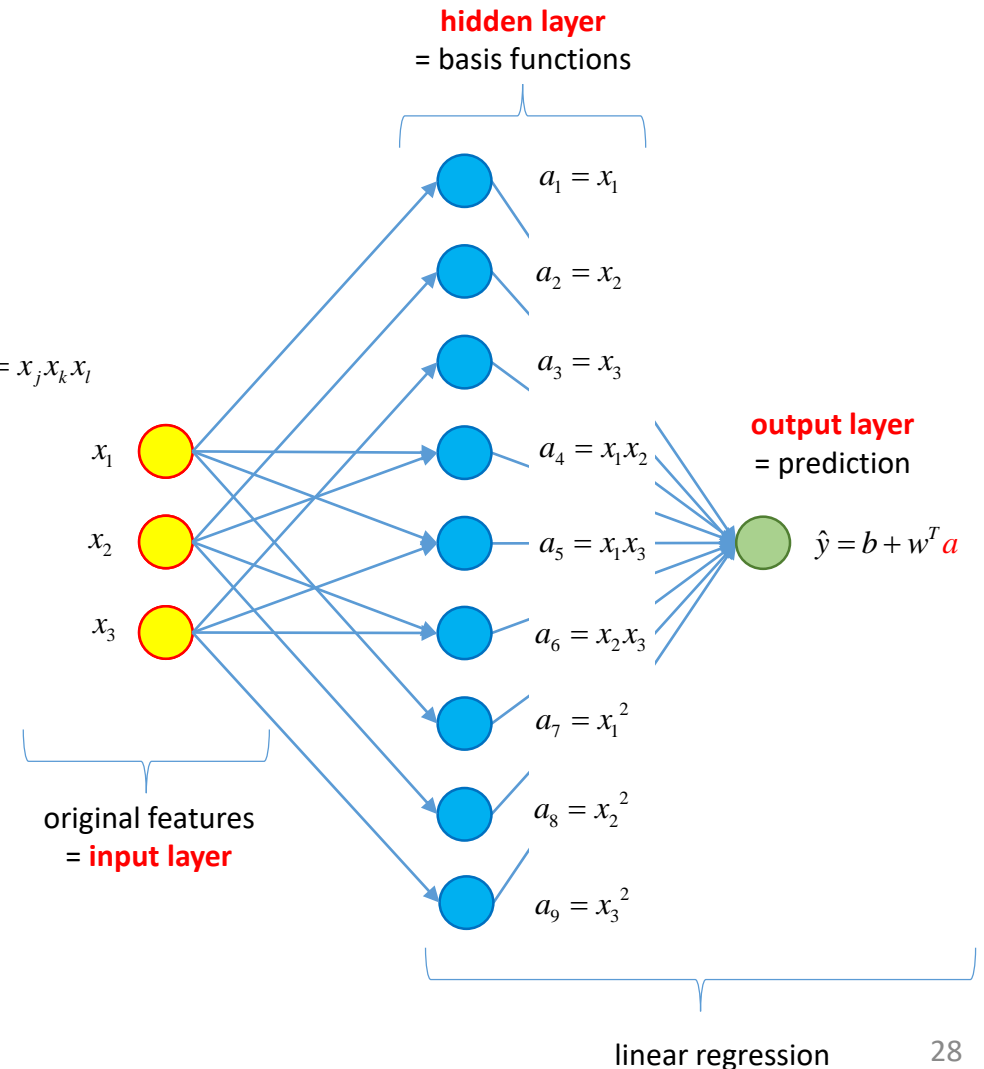
linear regression

Still linear, no more limited

- Works nicely: quadratic regression of $\left(S_{T_2} - K\right)^+$ over S_{T_1} and S_{T_2}
  approximates Black & Scholes' formula well in simulated example
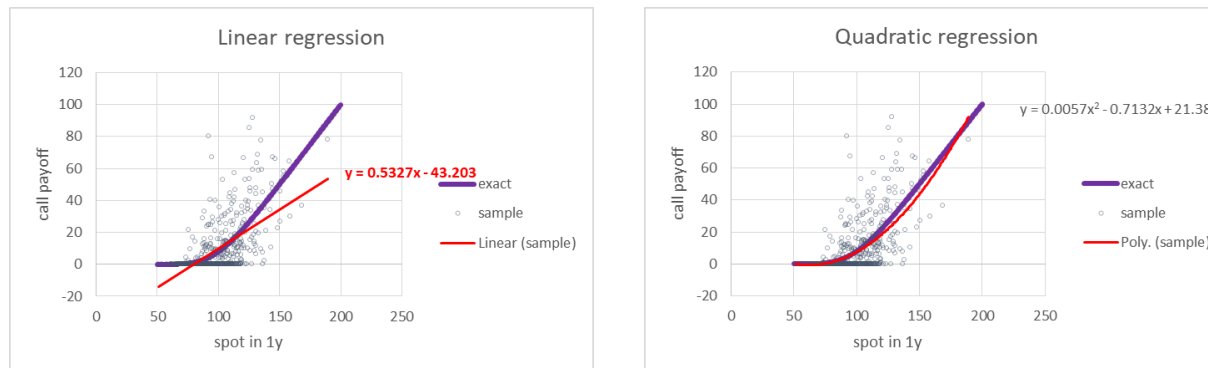
- Remarkable how the algorithm manages to detect Black & Scholes' pattern in noisy data

- Mathematically:
  - Combinations of polynomials can approximate any smooth function to arbitrary precision
  - Hence, detect any (smooth) non-linear pattern in data
  - *But only with a large number of basis functions / high polynomial degree*

Curse of dimensionality

- How many monomials in a p-degree polynomial regression?
  Number n_1 of basis functions (dimension of a) grows exponentially in number n_0 of features (dimension of x)

  Precisely: $n_1 = \dfrac{(n_0 + p)!}{n_0!\, p!} - 1$

- Quadratic regression: $n_1 = \dfrac{(n_0 + 2)(n_0 + 1)}{2} - 1$ dimension grows e.g. from 10 to 65, from 100 to 5,151, from 1,000 to 501,500

- Cubic regression: $n_1 = \dfrac{(n_0 + 3)(n_0 + 2)(n_0 + 1)}{6} - 1$ dim grows 10 to 286, 100 to 176,851, 1,000 to 167,668,501

- In general, number of basis functions increases exponentially in dimension

- "Rule of ten":
  to avoid overfitting small data set with many parameters, we need an amount of data (training examples) of approximately ***m > 10n***

- Hence, basis function regression requires amount of data exponential in (original) dimension n_0

- Basis function regression works nicely in low dimension but doesn't scale (think of image processing where $n0$ = 1920 x 1080 > 2M)

Introduction to overfitting

- If you fit a linear model with n free parameters to n data points
  - You will find a perfect fit
  - But the model may not generalize well to new examples
  - Because it captured the noise of the particular training set, "overfitting"

- For error analysis ("variance-bias trade-off"), see:

  https://www.quora.com/Does-neural-network-generalize-better-if-it-is-trained-on-a-larger-training-data-set-Are-there-any-scientific-papers-showing-it/answer/Antoine-Savine-1

- Solutions exist such as regularization
  See **Stanford's Machine Learning class on Coursera** or **Bishop's Pattern Recognition and Machine Learning**

- But the most effective means to avoid overfitting is increase the size of the learning set

- An empirical rule of thumb called "rule of ten"
    recommends a data set of at least 10x the number of fitted parameters

ANN: learn basis functions from data

BASIS FUNCTION REGRESSION
hidden layer = **arbitrary** set of basis functions
pre-processing = not part of the model, no parameters
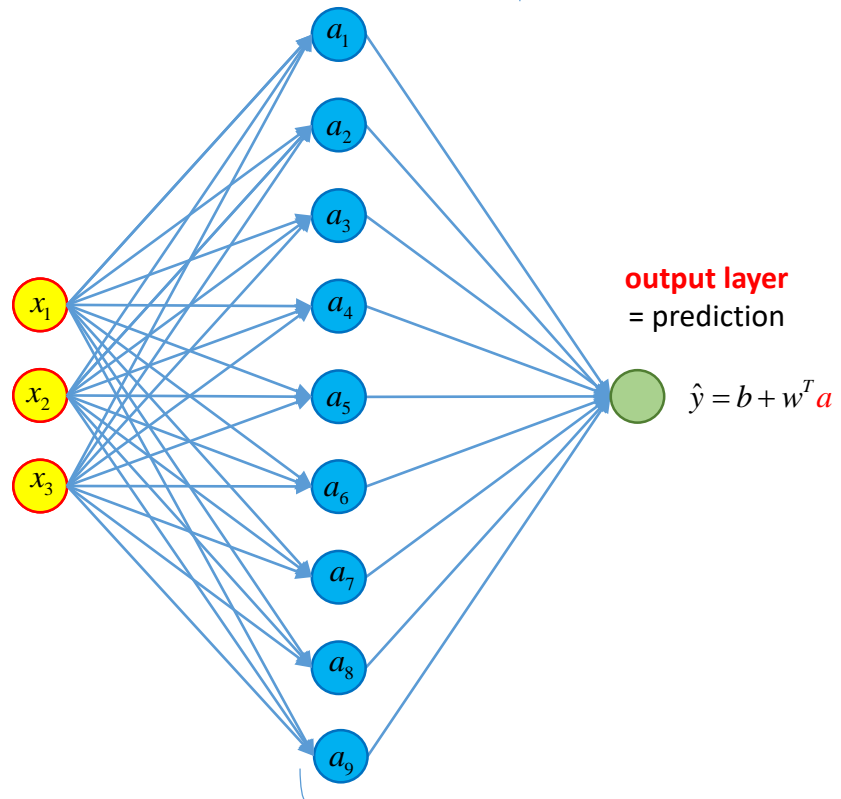
ANN
hidden layer = **learned** set of basis functions
part of the model, subject to trainable parameters

x_1 x_2 x_3 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9

output layer
= prediction

$$\hat{y} = b + w^T a$$

linear regression over basis functions
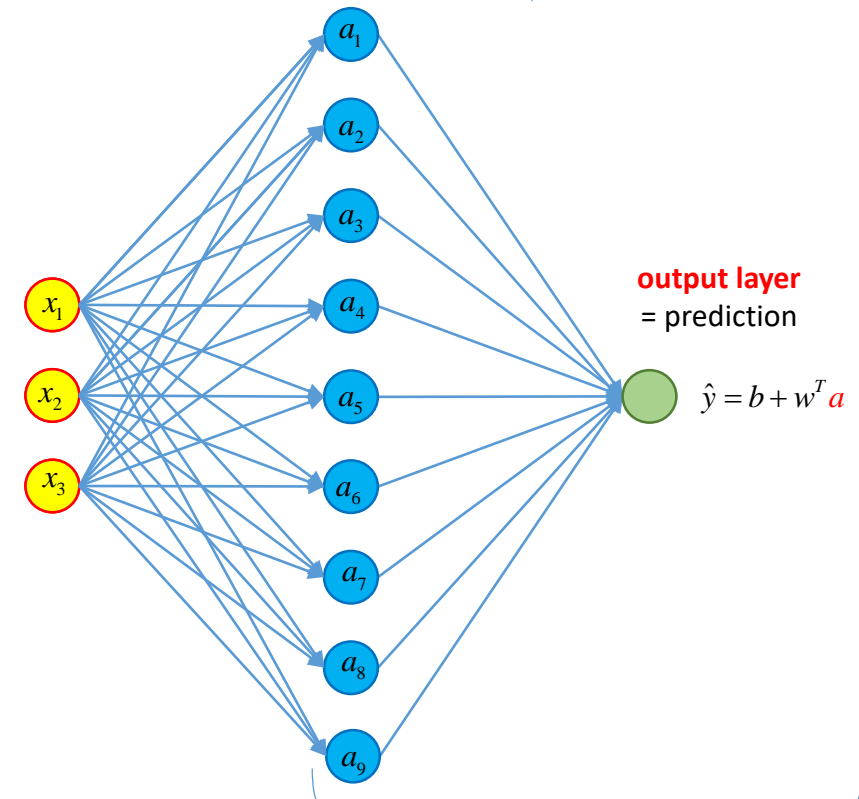
linear regression over basis functions
identical

Automatic feature extraction

- How do we learn basis functions *a* from the data?

  - Parametric family of basis functions: $a_i = \varphi(x; \vartheta_i)$ ➜ each basis function is a parametric function of *the same vector of inputs x* *each with different parameters θi*

  - Learn parameters ϑ (along with the weights *w* and bias *b* of regression) by minimization of the cost (e.g. sum of squared errors)

- Most common choice of parametric basis functions: "perceptron"

  - Each basis function is a non-linear scalar function *g* of a linear combination of the *x*s (different for each basis function): $a_i = g\left(w_i^T x + b_i\right)$

  - Where each $w_i, 1 \le i \le n_1$ is a vector in dimension n_0 and each $b_i, 1 \le i \le n_1$ is a real number

  - Hence, $W = \begin{pmatrix} w_1^T \\ \dots \\ w_{n_1}^T \end{pmatrix}$ is a $n_1 \times n_0$ matrix and $b = \begin{pmatrix} b_1 \\ \dots \\ b_{n_1} \end{pmatrix}$ is a n_1 dimensional vector

  - Then the vector of basis functions is $a = g(Wx + b)$ where *g* is a <u>scalar</u> function applied element-wise to the n_1 vector $z = Wx + b$

Perceptron

$$z_1 = \sum_{i=1}^{n_0} w_{1i}x_i + b_1 = w_1^T x + b_1 \qquad a_1 = g(z_1)$$

$$z_2 = \sum_{i=1}^{n_0} w_{2i}x_i + b_2 = w_2^T x + b_2 \qquad a_2 = g(z_2)$$

$$z_3 = \sum_{i=1}^{n_0} w_{3i}x_i + b_3 = w_3^T x + b_3 \qquad a_3 = g(z_3)$$

...

...

$$\hat{y} = \sum_{i=1}^{n_1} w^{[2]}{}_i a_i + b^{[2]} = w^{[2]T} a + b^{[2]}$$

output layer: combination of basis functions

$$z_9 = \sum_{i=1}^{n_0} w_{9i}x_i + b_9 = w_9^T x + b_9 \qquad a_9 = g(z_9)$$

hidden layer: compute basis functions
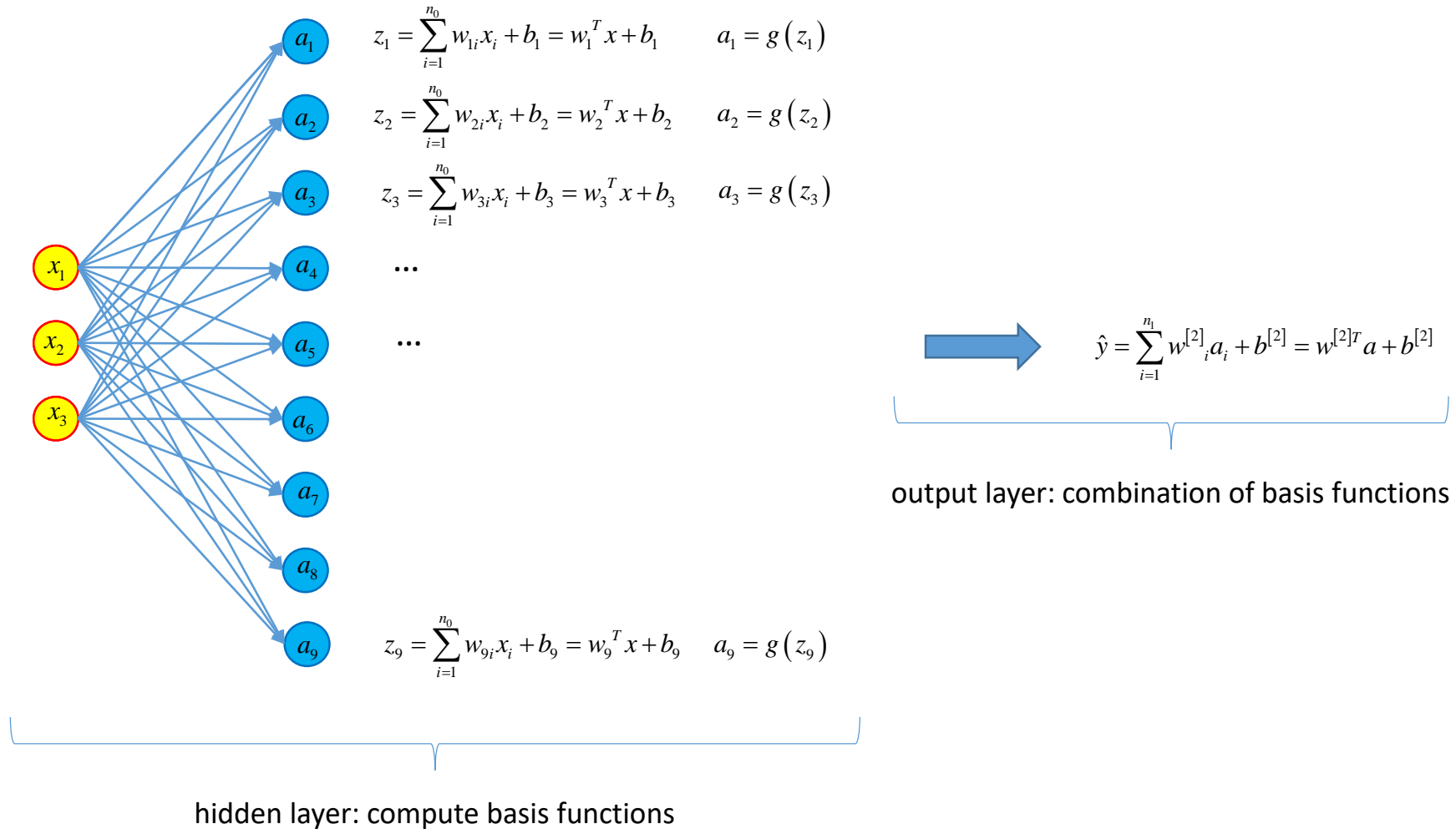
Prediction with perceptrons

- Prediction in two steps

  - Step 1 (hidden layer): feature (=basis function) extraction **inputs x (dim n0) $\rightarrow$ basis functions a (dim n1)** $\quad a = g\left(W^{[1]}x + b^{[1]}\right)$
    - Parameters: $W^{[1]}$ matrix in dimension $n_1 \times n_0$ and $b^{[1]}$ vector in dimension n_1
    - The *activation* function g is scalar and applied element-wise on the n_1 vector $\quad z^{[1]} = W^{[1]}x + b^{[1]}$
    - g must be non-linear or we are back to linear regression
      (a linear combination of linear functions is a linear function of the inputs)

  - Step 2: (output layer): regression **basis functions a (dim n1) $\rightarrow$ prediction (real number)** $\quad \hat{y} = w^{[2]}a + b^{[2]}$
    - Parameters: $w^{[2]}$ column vector in dimension n_1 and $b^{[2]}$ real number

- Unified notation for both steps = **feed-forward equation** $\quad a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$ **or** $\quad z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \rightarrow a^{[l]} = g^{[l]}\left(z^{[l]}\right)$
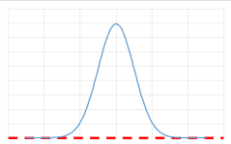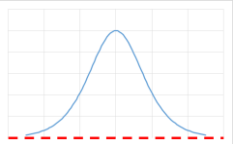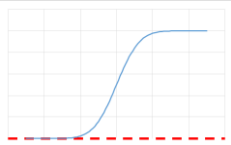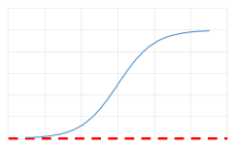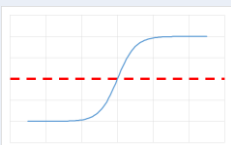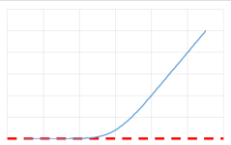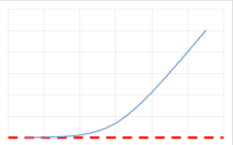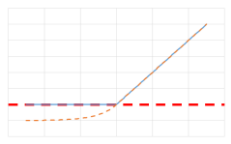
  - With the notations

$a^{[0]} = x$	input layer	dim n_0
$a^{[1]} = a$	hidden layer	dim n_1
$\hat{y} = a^{[2]}$	output layer	dim $n_2 = 1$
$g^{[1]} = g, g^{[2]} = id$	hidden and output activations	

    and parameters $\quad W^{[l]}$ of dim $n_l \times n_{l-1}$ and b$^{[l]}$ of dim n_l for $l = 1, 2$

Activations

The hidden activation function g must be non-linear – here are the most common choices :

Context	Intuitive solution	Cheaper computation	Improves training*
Activate localized spheres Application: interpolation/spline	Gaussian density $g(z) = n(z) = \dfrac{e^{-\frac{z^2}{2}}}{\sqrt{2\pi}}$	Derivative of sigmoid $g(z) = \sigma'(z)$ $= \sigma(z)[1 - \sigma(z)]$	
Activate above threshold Application: probability/classification/ flat extrapolation	Gaussian distribution $g(z) = N(z) = \int^z n(t)\,dt$	Sigmoid $g(z) = \sigma(z) = \dfrac{1}{1 + e^{-z}}$	Hyperbolic tangent $g(z) = \tanh(z)$ $= \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$
Activate proportionally to distance to threshold Application: **regression with linear extrapolation**	Integral of Guassian (Bachelier) $g(z) = \int^z N(t)\,dt$	SoftPlus = Integral of sigmoid $g(z) = \int^z \sigma(t)\,dt$ $= \log(1 + e^x)$	RELU / ELU $RELU : g(z) = \max(0, z)$ $ELU : g(z) = z$ if $z > 0$ else $e^z - 1$

derivative ↕ integral

*antisymmetric around zero, saturates slowly on the sides   **default choice is ELU**

Computation graph

Input layer l=0

$a^{[0]} = x$

n_0 units

Hidden layer l=1

$a^{[1]} = g^{[1]}\left(W^{[1]}a^{[0]} + b^{[1]}\right)$

n_1 units

Output layer l=2

$a^{[2]} = g^{[2]}\left(W^{[2]}a^{[1]} + b^{[2]}\right) = \hat{y}$

$n_2 = 1$ units
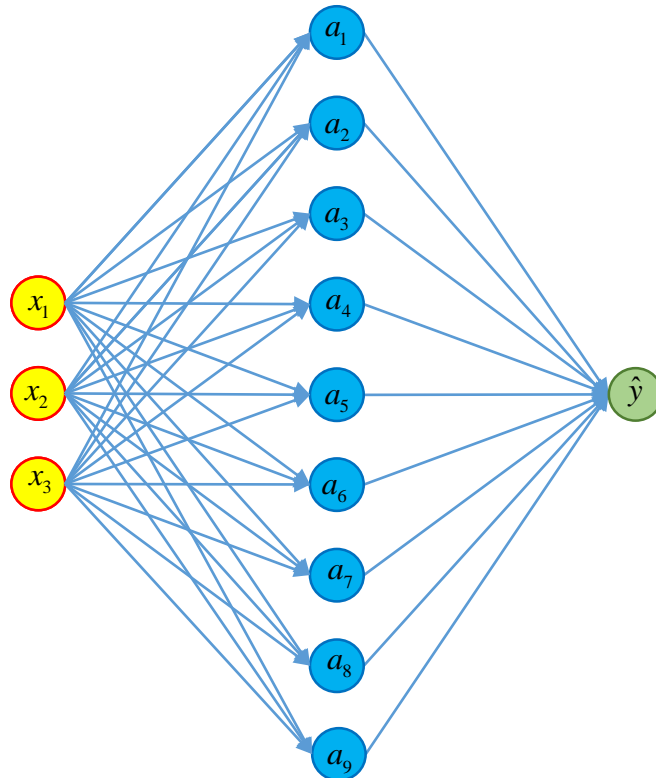
feed-forward equation

$a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$

Computation complexity (time)

To the leading order (ignoring activations and additions)

We have 2 matrix by vector products $\underset{n_1 \times n_0}{W^{[1]}} \underset{n_0}{a_0}$ and $\underset{n_2 \times n_1}{W^{[2]}} \underset{n_1}{a_1}$

Hence quadratic complexity $n_1 n_0 + n_2 n_1$

To simplify when all layers have *n* units, complexity $\sim 2n^2$

More generally, with *L* hidden layers, complexity $\sim Ln^2$

Complexity is linear in layers, quadratic in units

How to train your perceptron

- Training set of m examples $x^{(1)},...,x^{(m)}$ (each a vector in dimension n_0) with corresponding labels $y^{(1)},...,y^{(m)} \in \mathbb{R}$

- Learn parameters: $b^{[1]} \in \mathbb{R}^{n_1}, b^{[2]} \in \mathbb{R}^{n_2=1}$ and $W^{[1]} \in \mathbb{R}^{n_1 \times n_0}, W^{[2]} \in \mathbb{R}^{(n_2=1) \times n_1}$ that is $D = n_1 + n_2 + n_1 n_0 + n_2 n_1$ parameters

- By minimizing the cost function $C\left(b^{[1]}, b^{[2]}, W^{[1]}, W^{[2]}\right) = \sum_{i=1}^{m} c_i\left(b^{[1]}, b^{[2]}, W^{[1]}, W^{[2]}\right)$ with loss $c_i\left(b^{[1]}, b^{[2]}, W^{[1]}, W^{[2]}\right) = \left(\underbrace{a^{[2](i)}}_{=\hat{y}^{(i)}} - y^{(i)}\right)^2$

- Note we learn the basis functions and the regression weights at the same time

- No more analytic solution (C is not even convex in the D parameter space...)

- We must apply a numerical minimization algorithm

- Many iterative algorithms exist in literature and standard libraries

- The simplest one is gradient descent:
  repeatedly move parameters one step down the steepest slope of the cost function (gradient of cost to parameters)

- Gradient descent and other algorithms require differentials of cost function to all the parameters

- On each iteration, we must compute not only the cost function but also its D derivatives where we recall that $D \sim O(n_1 n_0)$

Universal Representation Theorem

- Polynomial regression offers mathematical guarantees
  - Combinations of monomials can approximate all smooth functions to arbitrary accuracy (with max degree p high enough)
  - Hence, polynomial regression (with enough monomials) can capture any (smooth) relation in the data
  - (At the cost of a high number of basis functions and the requirement of at least ten times that much data)

- Similarly, feed-forward ANNs with "perceptron" basis functions also offer mathematical guarantees
  - Known as the Universal Representation Theorem
  - States that a feed-forward perceptron may approximate any smooth function to arbitrary accuracy (with enough hidden units)
  - (Provided technical assumptions on activation functions, satisfied by all variants we presented)
  - See sketch of proof (with ReLu activation in dimension 1):
    - https://www.quora.com/In-Machine-Learning-ReLU-activation-function-is-said-to-be-a-universal-function-approximation-can-you-show-just-how-it-is-possible-to-approximate-any-continuous-function-by-purely-combining-many-instances-of-ReLU/answer/Antoine-Savine-1

- In conclusion
  - ANNs are superior to basis function regression in what they learn basis functions from the data
  - They scale to high dimension by regressing on a limited number of "most relevant" basis functions, found automatically
  - MLPs offer the same guarantees as basis function regression, also computationally friendly (matrix operations fit for parallel proc)
  - But they lose analytic solution for training and require iterative cost minimization
  - Iterative minimization procedures require gradients (all differentials) of the cost function on every iteration
  - Therefore a quick computation of derivatives is key to training in reasonable time

More on regression, basis functions and ANNs
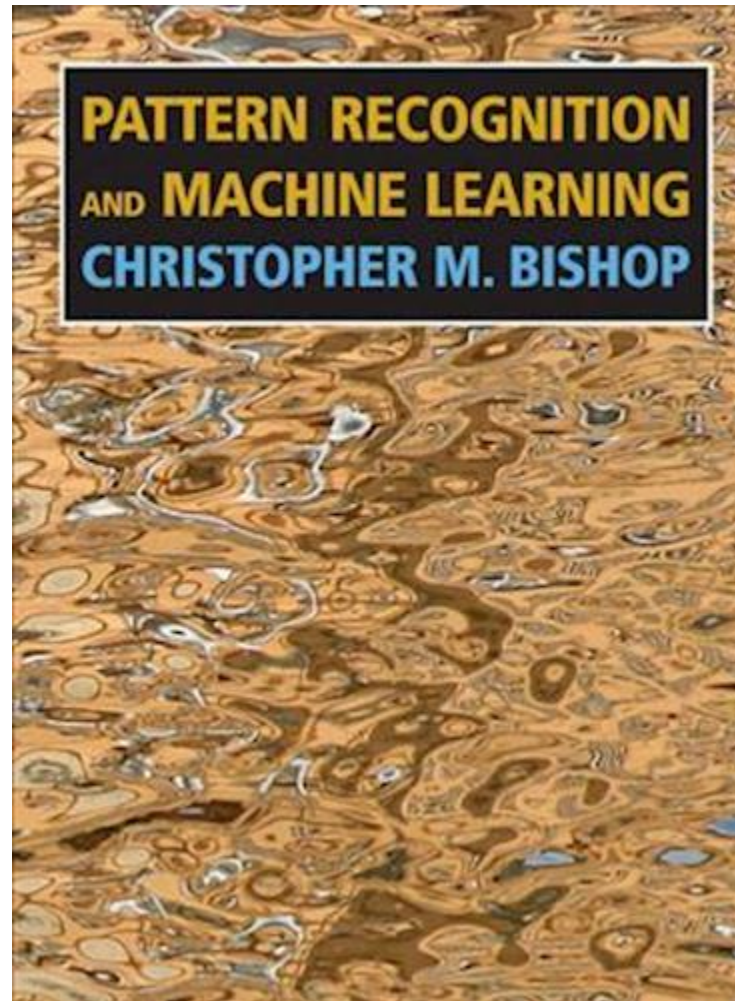
Deep learning: multiple hidden layers

feed-forward equation, same as before, now through multiple layers

$$a^{[l]} = g^{[l]} \left(W^{[l]} a^{[l-1]} + b^{[l]} \right)$$

Find basis functions by **composition**
new

linear regression over basis functions
identical
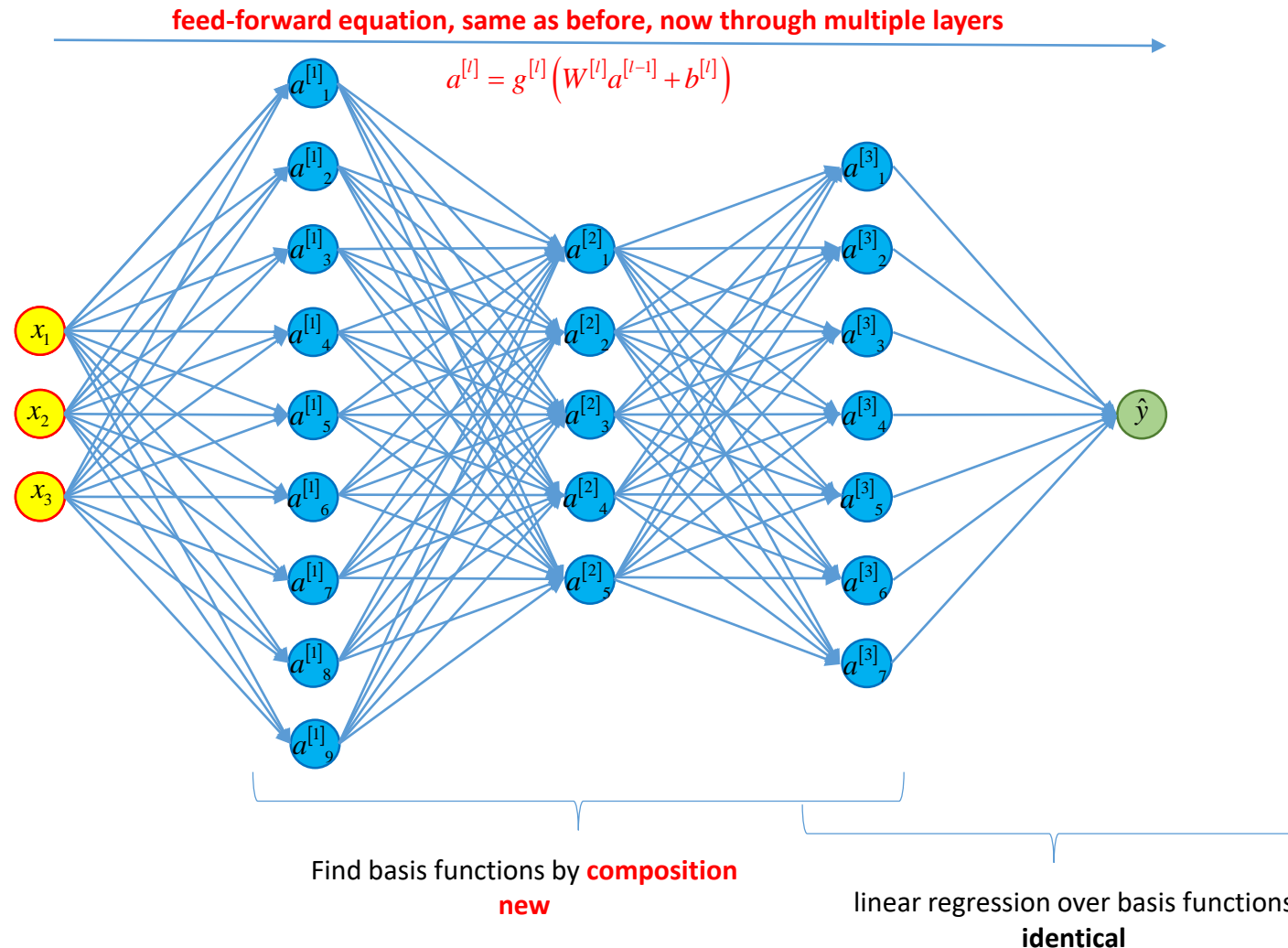
Basis function composition

- Shallow (one hidden layer) ANN:
  - Prediction is a linear combination of hidden layer activations: $\hat{y} = W^{[2]}a^{[1]} + b^{[2]}$
  - Hidden layer activations are basis functions of the input layer: $a^{[1]} = g^{[1]}\left(W^{[1]}x + b^{[1]}\right)$
  - So we linearly regress on (learnt) basis functions of inputs

- ANN with two hidden layers:
  - Prediction is a linear combination of the **second** hidden layer: $\hat{y} = W^{[3]}a^{[2]} + b^{[3]}$
  - Second layer activations are basis functions of the first hidden layer: $a^{[2]} = g^{[2]}\left(W^{[2]}a^{[1]} + b^{[2]}\right)$
  - First hidden layer activations are basis functions of the input layer: $a^{[1]} = g^{[1]}\left(W^{[1]}x + b^{[1]}\right)$
  - So we regress on **functions of basis functions**

- Deep ANN with L layers (L-1 hidden layers)
  - Prediction is a linear combination of the **last** hidden layer: $\hat{y} = W^{[L]}a^{[L-1]} + b^{[L]}$
  - Last hidden layer contains basis functions of basis functions
    of basis functions … of basis functions of the inputs
    from the feed-forward equation: $a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$
  - So we regress on many **layers of composition** of basis functions $a^{[0]} = x$

Power of depth

- We know from the Universal Representation Theorem that:
  - An ANN with a single hidden layer can approximate any (smooth) function given enough units
  - So what is the point of multiple layers?

- In practice:
  - We have a finite -and limited- number of units
  - So we can only attain a limited sub-space of functions

- With composition, we attain a wider, richer space of functions
  - This intuition was demonstrated in limited cases and widely validated empirically
  - A convincing, general mathematical demonstration is (yet) unavailable
  - But a vast amount of empirical results confirm that deep ANNs much better represent target functions
  - In short, 5 layers of 10 units each attain a wider, richer sup-space of functions than 1 layer with 50 units

- This is the basis for deep learning

- The mechanics of deep ANNs is otherwise identical to shallow ANNs

Multi-layer perceptron (MLP)

Input layer l=0
$a^{[0]} = x$
n_0 units

Hidden layer l=1
$a^{[1]} = g^{[1]}\left(W^{[1]}a^{[0]} + b^{[1]}\right)$
n_1 units

Hidden layer l=2
$a^{[2]} = g^{[2]}\left(W^{[2]}a^{[1]} + b^{[2]}\right)$
n_2 units

Hidden layer l=3
$a^{[3]} = g^{[3]}\left(W^{[3]}a^{[2]} + b^{[3]}\right)$
n_3 units

Output layer l=4
$a^{[4]} = g^{[4]}\left(W^{[4]}a^{[3]} + b^{[4]}\right)$
$= \hat{y}$
n_4 units
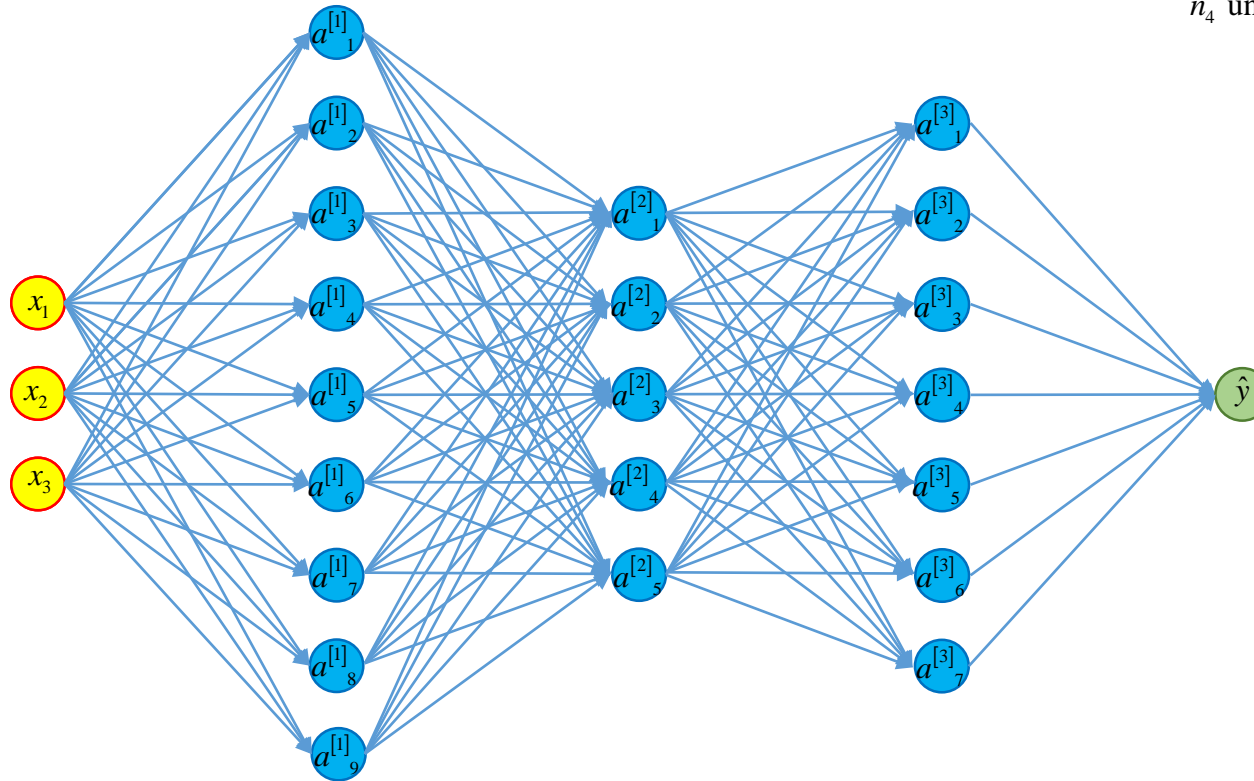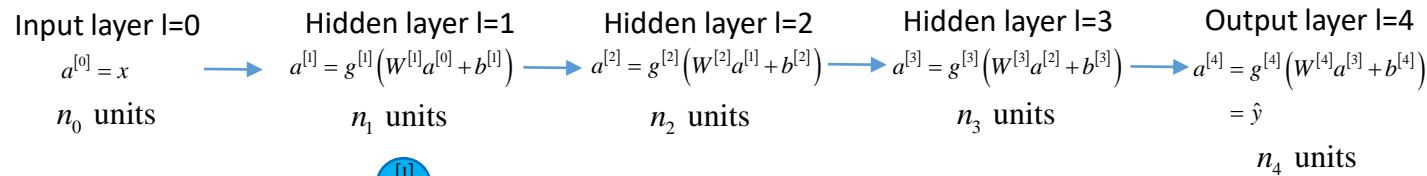
feed-forward equation
$a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$

- Prediction: feed-forward equations
$$a^{[0]} = x \ , \ a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right) \ , \ y = a^{[L]}$$

- Parameters: for $1 \leq l \leq L \ : \ W^{[l]}\left(n_l \times n_{l-1}\right), b^{[l]}\left(n_l\right)$

- Number of parameters: $D = \sum_{l=1}^{L} n_l \left(1 + n_{l-1}\right)$

- Complexity $\ \sim D = \sum_{l=1}^{L} n_l n_{l-1} \ " = Ln^2 \ "$

- Training: find all $W^{[l]}$ and $b^{[l]}$ to minimise cost
$$C = \sum_{i=1}^{m} c_i \ , \ c_i = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

- Use iterative algorithms
  compute all D differentials on each iteration

- Key: quickly compute
  large number D of differentials

44

Summary and notations: general deep ANN

- Prediction

  - Start with layer 0 = inputs $\begin{bmatrix} a^{[0]}_1 \\ \ldots \\ a^{[0]}_{n_0} \end{bmatrix} = \begin{bmatrix} x_1 \\ \\ x_{n_0} \end{bmatrix}$

  - Feed-forward from layer l-1 to layer l ➔ $\begin{bmatrix} a^{[l]}_1 \\ \ldots \\ a^{[l]}_{n_l} \end{bmatrix} = \varphi^{[l]}\left(\begin{bmatrix} a^{[l-1]}_1 \\ \ldots \\ a^{[l-1]}_{n_{l-1}} \end{bmatrix}; \begin{bmatrix} \vartheta^{[l]}_1 \\ \ldots \\ \vartheta^{[l]}_{D_l} \end{bmatrix} \right)$

  - Prediction = layer L $\hat{y} = a^{[L]}_1$

- Parameters: $\begin{bmatrix} \vartheta^{[1]}_1 \\ \ldots \\ \vartheta^{[1]}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}_1 \\ \ldots \\ \vartheta^{[2]}_{D_2} \end{bmatrix}, \ldots, \begin{bmatrix} \vartheta^{[L-1]}_1 \\ \ldots \\ \vartheta^{[L-1]}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}_1 \\ \ldots \\ \vartheta^{[L]}_{D_L} \end{bmatrix}$ so, ultimately: $\hat{y} = f\left(\begin{bmatrix} x_1 \\ \ldots \\ x_{n_0} \end{bmatrix}; \begin{bmatrix} \vartheta^{[1]}_1 \\ \ldots \\ \vartheta^{[1]}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}_1 \\ \ldots \\ \vartheta^{[2]}_{D_2} \end{bmatrix}, \ldots, \begin{bmatrix} \vartheta^{[L-1]}_1 \\ \ldots \\ \vartheta^{[L-1]}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}_1 \\ \ldots \\ \vartheta^{[L]}_{D_L} \end{bmatrix} \right)$

- Training: find all the parameters theta

  - To minimize sum of errors on training set with known labels $\min_{\begin{bmatrix} \vartheta^{[1]}_1 \\ \ldots \\ \vartheta^{[1]}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}_1 \\ \ldots \\ \vartheta^{[2]}_{D_2} \end{bmatrix}, \ldots, \begin{bmatrix} \vartheta^{[L-1]}_1 \\ \ldots \\ \vartheta^{[L-1]}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}_1 \\ \ldots \\ \vartheta^{[L]}_{D_L} \end{bmatrix}} C = \sum_{i=1}^{m} c_i = \left(\hat{y}_i - y_i \right)^2$

  - Takes iterative algorithm where all the differentials must be computed on every iteration: $\dfrac{\partial c_i}{\partial \vartheta^{[l]}_j}$ for all i, j and l

Summary and notations: MLP

- Special case of ANN with mathematical and computational benefits – by far the most common form of ANN

  - Start with layer 0 = inputs
  $$\begin{bmatrix} a^{[0]}_1 \\ \dots \\ a^{[0]}_{n_0} \end{bmatrix} = \begin{bmatrix} x_1 \\ \dots \\ x_{n_0} \end{bmatrix}$$

  - **Feed-forward from layer l-1 to layer l** ➡
  $$\begin{bmatrix} a^{[l]}_1 \\ \dots \\ a^{[l]}_{n_l} \end{bmatrix} = g^{[l]}\left(\begin{bmatrix} W^{[l]}_{1,1} & \dots & W^{[l]}_{1,n_{l-1}} \\ \dots & \dots & \dots \\ W^{[l]}_{n_l,1} & \dots & W^{[l]}_{n_l,n_{l-1}} \end{bmatrix} \begin{bmatrix} a^{[l-1]}_1 \\ \dots \\ a^{[l-1]}_{n_{l-1}} \end{bmatrix} + \begin{bmatrix} b^{[l]}_1 \\ \dots \\ b^{[l]}_{n_l} \end{bmatrix} \right)$$
  where g is scalar, applied element-wise

  - Prediction = layer L $\hat{y} = a^{[L]}_1$

- Parameters: all the $\begin{bmatrix} W^{[l]}_{1,1} & \dots & W^{[l]}_{1,n_{l-1}} \\ \dots & \dots & \dots \\ W^{[l]}_{n_l,1} & \dots & W^{[l]}_{n_l,n_{l-1}} \end{bmatrix}$ and all the $\begin{bmatrix} b^{[l]}_1 \\ \dots \\ b^{[l]}_{n_l} \end{bmatrix}$ for all layers l

- Training: find all the parameters W and b

  - Minimize sum of errors on training set with known labels $\min_{\begin{bmatrix} w^{[l]}_{1,1} & \dots & w^{[l]}_{1,n_{l-1}} \\ \dots & \dots & \dots \\ w^{[l]}_{n_l,1} & \dots & w^{[l]}_{n_l,n_{l-1}} \end{bmatrix}, \begin{bmatrix} b^{[l]}_1 \\ \dots \\ b^{[l]}_{n_l} \end{bmatrix}, 1 \le l \le L} \left[C = \sum_{i=1}^{m} \left\{ c_i = (\hat{y}_i - y_i)^2 \right\} \right]$

  - Takes iterative algorithm where all the differentials must be computed on every iteration: $\dfrac{\partial c_i}{\partial W^{[l]}_{j,k}}, \dfrac{\partial c_i}{\partial b^{[l]}_j}$ for all i, j, k and l

Back-Propagation

Elements of multivariate calculus

- We will derive the differentials of cost functions in general ANNs and MLPs

- Along with an algorithm to compute them extremely fast

- Since ANNs and MLPs are sequences of matrix operations: addition, product, element-wise activation, etc.

- We apply matrix differential calculus

- We briefly introduce a few elementary results, for a complete reference, see:
  - Mathematics for machine learning by Imperial College on Coursera
  - M. Giles, Extended collection of matrix derivative results (2008)
  - R. J. Barnes, Matrix Differentiation
  - B. Huge , AD of matrix calculations (public on ResearchGate)

Gradients

- Consider a scalar function of a vector: $f : \mathbb{R}^n \to \mathbb{R}$

- We call *gradient of f on x* and denote $\nabla f(x)$ the <u>column</u> vector of its derivatives:

$$\nabla f(x) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}(x) \\[2mm] \dfrac{\partial f}{\partial x_2}(x) \\[1mm] \ldots \\[1mm] \dfrac{\partial f}{\partial x_n}(x) \end{bmatrix}$$

derivatives in rows

function in column

- Key property of gradient:
  the gradient vector gives the direction where the slope of f is steepest

- Demonstration: to the 1st order

$$f(x + \Delta x) = f(x) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \Delta x_i = \nabla f(x)^T \Delta x = \langle \nabla f(x), \Delta x \rangle = \| \nabla f(x) \| \| \Delta x \| \cos(\nabla f(x), \Delta x)$$

Hence, for a fixed step size $\| \Delta x \|$
- steepest ascent is in the direction of $\nabla f(x)$ *cos=1*
- steepest descent is in the direction of $-\nabla f(x)$ *cos=-1*
- *f* is unchanged moving orthogonally to $\nabla f(x)$ *cos=0*

- This key property is the basis of optimization algorithms and the process of training in machine learning

steepest descent

f(x)

f unchanged

x

$\nabla f(x)$

steepest ascent
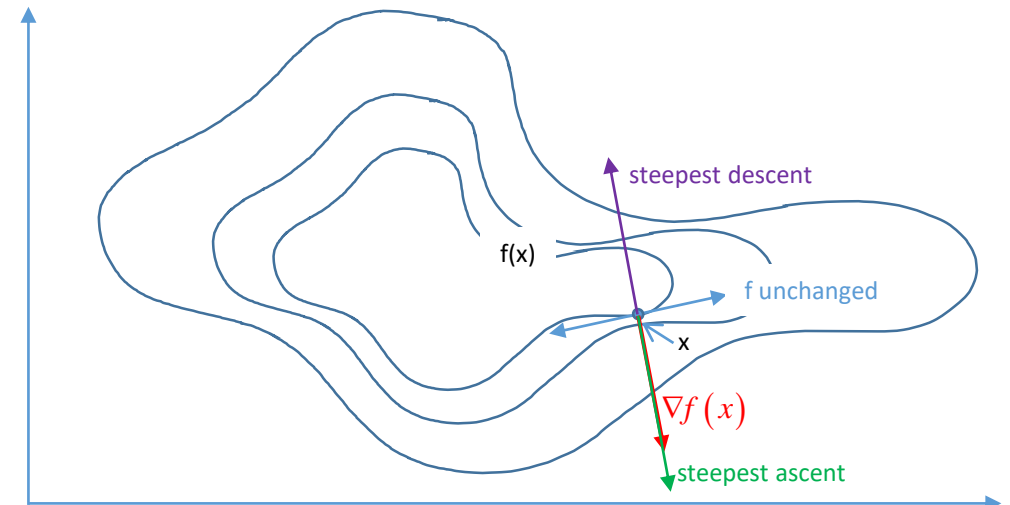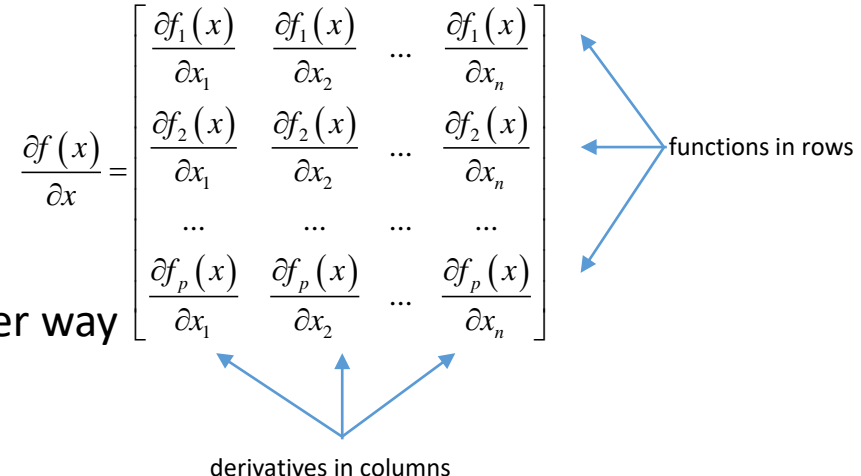
Matrix gradients

- The gradient of *f(x)* has the same shape *(nx1)* as *x*

- We want the same property scalar functions of a matrices: $f : \mathbb{R}^{n \times p} \to \mathbb{R}$

- We denote $\dfrac{\partial f(M)}{\partial M}$ the matrix of its derivatives, of the same shape *(nxp)* as *M*

$$\frac{\partial f(M)}{\partial M} = \begin{bmatrix} \dfrac{\partial f(M)}{\partial M_{11}} & \dfrac{\partial f(M)}{\partial M_{12}} & \cdots & \dfrac{\partial f(M)}{\partial M_{1p}} \\ \dfrac{\partial f(M)}{\partial M_{21}} & \dfrac{\partial f(M)}{\partial M_{22}} & \cdots & \dfrac{\partial f(M)}{\partial M_{2p}} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial f(M)}{\partial M_{n1}} & \dfrac{\partial f(M)}{\partial M_{n2}} & \cdots & \dfrac{\partial f(M)}{\partial M_{np}} \end{bmatrix}.$$

Jacobians

- Consider a vector function of a vector (*p* functions of *x, x* being of dim *n*) $f : \mathbb{R}^n \to \mathbb{R}^p, f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \dots \\ f_p(x) \end{bmatrix}$

- The matrix $\dfrac{\partial f(x)}{\partial x}$ of the derivatives of the *p* functions to the *n* inputs is called Jacobian matrix

- Conventionally, Jacobians are often written with functions in rows and derivatives in columns: $\dfrac{\partial f(x)}{\partial x} = \begin{bmatrix} \dfrac{\partial f_1(x)}{\partial x_1} & \dfrac{\partial f_1(x)}{\partial x_2} & \dots & \dfrac{\partial f_1(x)}{\partial x_n} \\ \dfrac{\partial f_2(x)}{\partial x_1} & \dfrac{\partial f_2(x)}{\partial x_2} & \dots & \dfrac{\partial f_2(x)}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \dfrac{\partial f_p(x)}{\partial x_1} & \dfrac{\partial f_p(x)}{\partial x_2} & \dots & \dfrac{\partial f_p(x)}{\partial x_n} \end{bmatrix}$ functions in rows

- This is inconsistent with gradients, written the other way Hence, confusing

derivatives in columns

- In this case, $\dfrac{\partial f(x)}{\partial x}$ is of shape *(pxn)*

$g : \mathbb{R}^n \to \mathbb{R}^p$

$f : \mathbb{R}^p \to \mathbb{R}^q$

$h(x) = f[g(x)] : \mathbb{R}^n \to \mathbb{R}^q$

- And you can check that the chain rule is written: $\Rightarrow \underbrace{\dfrac{\partial h(x)}{\partial x}}_{q \times n} = \underbrace{\dfrac{\partial f(x)}{\partial g(x)}}_{q \times p} \underbrace{\dfrac{\partial g(x)}{\partial x}}_{p \times n}$ (outer function first)

Jacobians the other way

- We will write Jacobians the other way, consistently with gradients:

- This is consistent with gradients
  - Gradient = Jacobian of one-dimensional functions
  - Jacobian = horizontal concatenation of gradients

- In this case, $\frac{\partial f(x)}{\partial x}$ is of shape *(nxp)*

- And you can check that the chain rule is written:

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \dfrac{\partial f_1(x)}{\partial x_1} & \dfrac{\partial f_2(x)}{\partial x_1} & \cdots & \dfrac{\partial f_p(x)}{\partial x_1} \\[2ex] \dfrac{\partial f_1(x)}{\partial x_2} & \dfrac{\partial f_2(x)}{\partial x_2} & \cdots & \dfrac{\partial f_p(x)}{\partial x_2} \\[2ex] \cdots & \cdots & \cdots & \cdots \\[2ex] \dfrac{\partial f_1(x)}{\partial x_n} & \dfrac{\partial f_2(x)}{\partial x_n} & \cdots & \dfrac{\partial f_p(x)}{\partial x_n} \end{bmatrix}$$

derivatives in rows

functions in columns

$$g : \mathbb{R}^n \to \mathbb{R}^p$$
$$f : \mathbb{R}^p \to \mathbb{R}^q$$
$$h(x) = f\big[g(x)\big] : \mathbb{R}^n \to \mathbb{R}^q \quad \text{(inner function first)}$$
$$\Rightarrow \underbrace{\frac{\partial h(x)}{\partial x}}_{n \times q} = \underbrace{\frac{\partial g(x)}{\partial x}}_{n \times p} \underbrace{\frac{\partial f(x)}{\partial g(x)}}_{p \times q}$$

Linear Jacobians

- If $f : \mathbb{R}^n \to \mathbb{R}^p$ is linear, i.e. $f(x) = Ax$ where *A* is a matrix of shape (*pxn*)

- Then, you can convince yourself that its Jacobian matrix (with derivatives in rows) is the constant:

$$\frac{\partial f(x)}{\partial x} = A^T$$

and not *A* which is maybe why people like Jacobians the other way around...

$$f_i = A_i * x$$

Element-wise Jacobians

- Consider a scalar function: $f : \mathbb{R} \to \mathbb{R}$
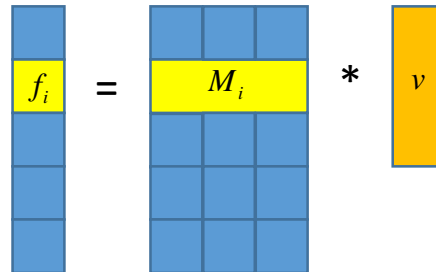
- We apply f element-wise to a vector x of dim n to obtain another vector of dim n: $f(x) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \dots \\ f(x_n) \end{bmatrix}$
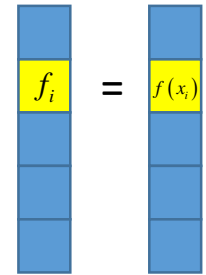
- Then, the Jacobian of f is the nxn matrix: $\dfrac{\partial f}{\partial x} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} = f'(x_1) & \dfrac{\partial f_2}{\partial x_1} = 0 & \dots 0 \dots & 0 \\ \dfrac{\partial f_1}{\partial x_2} = 0 & \dfrac{\partial f_2}{\partial x_2} = f'(x_2) & \dots 0 \dots & 0 \\ 0 & 0 & \dots \dfrac{\partial f_i}{\partial x_i} = f'(x_i) \dots & \\ 0 & 0 & \dots 0 \dots & \dfrac{\partial f_n}{\partial x_n} = f'(x_n) \end{bmatrix} = diag\left[f'(x) \right]$

Vector functions of matrices

- Consider now a vector function of a matrix: $f : \mathbb{R}^{n \times p} \rightarrow \mathbb{R}^q$

- In principle, its "derivative" is a 3D tensor of shape *(nxpxq)*: $\dfrac{\partial f}{\partial M} = \left(\dfrac{\partial f_k}{\partial M_{ij}} \right)$

- In the linear case where $f(M) = Mv$, *v* is a constant vector of size *p=q*

- We have $\dfrac{\partial f_i}{\partial M_i} = v^T$ and $\dfrac{\partial f_{j \neq i}}{\partial M_i} = 0$

$f_i \quad = \quad M_i \quad * \quad v$

- Consider the series of calculations:
  - Initial input: x_0 vector of dim *n0*
  - Apply $x_1 = f_1(x_0)$ with $f_1 : \mathbb{R}^{n_0} \to \mathbb{R}^{n_1}$ (for avoidance of doubt, not necessarily element-wise) to compute a vector of dim *n1*
  - Apply $x_2 = f_2(x_1), x_3 = f_3(x_2),..., x_{l+1} = f_{l+1}(x_l),..., x_L = f_L(x_{L-1})$ to compute a sequence of vectors of dims *n2, n3, etc.*
  - Finally, apply a final <u>scalar</u> function to the final result of the chain: $c = h(x_L)$ with $h : \mathbb{R}^{n_L} \to \mathbb{R}$

$$x_0 \in \mathbb{R}^{n_0} \xrightarrow{f_1:\mathbb{R}^{n_0}\to\mathbb{R}^{n_1}} x_1 \in \mathbb{R}^{n_1} \xrightarrow{f_2:\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}} x_2 \in \mathbb{R}^{n_2} \xrightarrow{f_3:\mathbb{R}^{n_2}\to\mathbb{R}^{n_3}} ... \xrightarrow{f_l:\mathbb{R}^{n_{l-1}}\to\mathbb{R}^{n_l}} x_l \in \mathbb{R}^{n_l} \xrightarrow{f_{l+1}:\mathbb{R}^{n_{l+1}}\to\mathbb{R}^{n_l}} ... \xrightarrow{f_L:\mathbb{R}^{n_{L-1}}\to\mathbb{R}^{n_L}} x_L \in \mathbb{R}^{n_L} \xrightarrow{h:\mathbb{R}^{n_L}\to\mathbb{R}} c \in \mathbb{R}$$

$\partial c / \partial x_l$

- Consider the vector $\partial c / \partial x_l$ of dim *nl*
  - Means sensitivities of the final result *c* to small bumps in the components of *xl*
    - All computations *before* stage l are fixed
    - All computations *after* stage l are impact by the bump to *xl*
  - Is called the <u>adjoint</u> of *xl* and denoted $\overline{x_l}$
  - Evidently, $\boxed{\overline{x_L} = \nabla h(x_L)}$ and by a straight application of the chain rule, we get the <u>adjoint equation</u>:
  - Note that we have a formula for the last adjoint $\overline{x_L}$ and for all adjoints $\overline{x_l}$ function of the <u>following</u> adjoint $\overline{x_{l+1}}$
  - Adjoints are therefore computed *in the reverse order*

$$\underbrace{\overline{x_l}}_{\substack{\text{adjoint of } x_l \\ n_l \times 1}} = \frac{\partial c}{\partial x_l} = \frac{\partial x_{l+1}}{\partial x_l} \frac{\partial c}{\partial x_{l+1}} = \underbrace{\frac{\partial f_{l+1}(x_l)}{\partial x_l}}_{\substack{\text{Jacobian of } f_{l+1} \\ n_l \times n_{l+1}}} \underbrace{\overline{x_{l+1}}}_{\substack{\text{adjoint of } x_{l+1} \\ n_{l+1} \times 1}}$$

How to train your network, more formally

- **Recap**: we have an ANN of the general form implementing the following sequence of computations:
  - Starting with inputs $a^{[0]} = x \in \mathbb{R}^{n_0}$
  - Sequentially compute $a^{[l]} = \varphi^{[l]}\left(a^{[l-1]}; \vartheta^{[l]}\right) \in \mathbb{R}^{n_l}$
  - To make predictions $\hat{y} = a^{[L]} \in \mathbb{R}$

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{\varphi^{[1]}\left(\cdot; \vartheta^{[1]}\right): \mathbb{R}^{n_0} \to \mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{\varphi^{[2]}\left(\cdot; \vartheta^{[2]}\right): \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{\varphi^{[3]}\left(\cdot; \vartheta^{[3]}\right): \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}} \ldots \xrightarrow{\varphi^{[l]}\left(\cdot; \vartheta^{[l]}\right): \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} \ldots \xrightarrow{\varphi^{[L]}\left(\cdot; \vartheta^{[L]}\right): \mathbb{R}^{n_{L-1}} \to \mathbb{R}^{n_L}} \hat{y} = a^{[L]} \in \mathbb{R}$$

- We also have a training set of m independent samples $x^{(m)}$ for the inputs x (each in dim $n0$), along with corresponding labels $y^{(m)}$

- We want to train the ANN so its predictions correspond to the "best" predictions for y knowing x, that is, produce predictions $\hat{y} = E\left[y|x\right]$

- By definition, a conditional expectation is the closest function in the sense of least squares: $E\left[y|x\right] = \arg\min_f E\left[\left(y - f(x)\right)^2\right]$

- And we know/have seen that:
  - (given enough height and depth) the ANN can represent any function f to arbitrary accuracy by adjusting its weights θ

  - (given enough training examples) $E\left[\left(y - f(x)\right)^2\right] \approx \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} - f\left(x^{(i)}\right)\right]^2 = \frac{1}{m}\sum_{i=1}^{m} c_i$

- Therefore, "all we need to do" is find the weights $\theta*$ that minimize the mean square prediction error on the training set: $\vartheta^* = \arg\min_\vartheta C(\vartheta) = \frac{1}{m}\sum_{i=1}^{m} c_i(\vartheta)$

57

Bad news good news

- There is no analytical solution for the optimal weights (unlike with linear regression)

- The cost $C(\vartheta)$ is a **non-convex** function of the parameters ϑ

- An algorithm guaranteed to find the minimum of a non-convex function **does not exist**, and probably never will

- So, the problem of training ANNs cannot, in theory, be solved

- This realizations caused mathematicians to step away from deep learning in the 1990s

- Deep learning was revived recently by engineers along the lines of
  "this is not supposed to work, but let's try anyway and see how we can take it"

- (And did it work!)

- The starting point is gradient descent:
    - Choose initial weights randomly
    - Take repeated steps
        - by a small amount called learning rate
        - along the direction of steepest descent, which we know is the direction opposite to the gradient

- Many <u>heuristic</u> improvements were made to improve and accelerate learning, but none is guaranteed to converge

- However, we understand the intuitions and take comfort in that they work remarkably well in a multitude of contexts

Differentials by finite differences

- Optimization algorithms
  - All learning algorithms start with a random guess and make steps according to the current gradient
  - We must compute and provide the gradient $\partial C/\partial \vartheta$ of the cost C to weights θ, repeatedly
  - Our goal is to compute it quickly

- Gradients by finite differences (FD):
  - First compute the cost C_0 , making predictions for all training examples
  - To compute the sensitivity of the cost to each (scalar) parameter ϑ_i:
    - Bump the parameter by a small amount ε
    - Re-compute the cost C_1 , repeating calculations for all training examples
    - $\partial C/\partial \vartheta_i \approx (C_1 - C_0)/\varepsilon$

- FD is (obviously) linear in the (potentially immense) number of weights, therefore not viable
- But it has very desirable properties

Automatic derivatives with FD

- Implementation is straightforward with little scope for error

- Importantly, FD differentiation is **automatic**
  - Developers only write prediction (feed-forward) code
  - FD computes differentials automatically by calling the feed-forward code repeatedly
  - Developers don't need to write any differentiation code

- More importantly, FD automatically **synchronises** with modifications to feed-forward code
  - Tricks of the trade (see e.g. Coursera's Deep Learning Spec) to train deep nets faster and better:
    - Regularization: add a regularization term to cost function
    - Dropout: randomly drop connections between units
    - Batch Norm: normalize the mean and variance on all layers
    - And many more
  - All these modify feed-forward equations and code, and affect differentials
  - With manual differentiation, developers must consistently ensure consistency of evaluation and differentiation code
  - This is painful and prone to error
  - With automatic differentiation, differentiation always remains consistent with evaluation

- This being said, FD and other linear algorithms are not viable for training deep nets

- Recall the sequence of calculations in the network:

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{\varphi^{[1]}\left(\cdot\,;\vartheta^{[1]}\right):\mathbb{R}^{n_0}\to\mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{\varphi^{[2]}\left(\cdot\,;\vartheta^{[2]}\right):\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{\varphi^{[3]}\left(\cdot\,;\vartheta^{[3]}\right):\mathbb{R}^{n_2}\to\mathbb{R}^{n_3}} \ldots \xrightarrow{\varphi^{[l]}\left(\cdot\,;\vartheta^{[l]}\right):\mathbb{R}^{n_{l-1}}\to\mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} \ldots \xrightarrow{\varphi^{[L]}\left(\cdot\,;\vartheta^{[L]}\right):\mathbb{R}^{n_{L-1}}\to\mathbb{R}^{n_L}} \hat{y} = a^{[L]} \in \mathbb{R} \to c = \left(\hat{y} - y\right)^2 \in \mathbb{R}$$

- We want to (quickly) compute all the $\dfrac{\partial C}{\partial \vartheta^{[l]}}$

- Where the cost is the mean of the losses in the training set: $C = \dfrac{1}{m}\sum_{i=1}^{m} c_i$ so $\dfrac{\partial C}{\partial \vartheta^{[l]}} = \dfrac{1}{m}\sum_{i=1}^{m} \dfrac{\partial c_i}{\partial \vartheta^{[l]}}$

- We derive the derivatives of one loss c (dropping its index i in the training set) and sum them up in the end

- By the chain rule: $\boxed{\dfrac{\partial c}{\partial \vartheta^{[l]}} = \dfrac{\partial a^{[l]}}{\partial \vartheta^{[l]}}\dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial \varphi^{[l]}\left(a^{[l-1]};\vartheta^{[l]}\right)}{\partial \vartheta^{[l]}}\dfrac{\partial c}{\partial a^{[l]}}}$ and $\dfrac{\partial \varphi^{[l]}\left(a^{[l-1]};\vartheta^{[l]}\right)}{\partial \vartheta^{[l]}}$ is **known**:

  the shape of φ is a hyperparameter that defines the architecture of the network

  the weights ϑ represent the current known state of the network

  the activations a must be known, so a prior feed-forward evaluation is necessary before differentiation

- So we can first compute all the $\dfrac{\partial c}{\partial a^{[l]}}$, and then, apply the equation above to find all the $\dfrac{\partial c}{\partial \vartheta^{[l]}}$

- Still from the chain rule, we see that: $\dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial a^{[l+1]}}{\partial a^{[l]}} \dfrac{\partial c}{\partial a^{[l+1]}} = \dfrac{\partial \varphi^{[l+1]}\left(a^{[l]}; \vartheta^{[l+1]}\right)}{\partial a^{[l]}} \dfrac{\partial c}{\partial a^{[l+1]}}$

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{\varphi^{[1]}\left(\cdot\,;\vartheta^{[1]}\right):\mathbb{R}^{n_0}\to\mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{\varphi^{[2]}\left(\cdot\,;\vartheta^{[2]}\right):\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{\varphi^{[3]}\left(\cdot\,;\vartheta^{[3]}\right):\mathbb{R}^{n_2}\to\mathbb{R}^{n_3}} \cdots \xrightarrow{\varphi^{[l]}\left(\cdot\,;\vartheta^{[l]}\right):\mathbb{R}^{n_{l-1}}\to\mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} \cdots \xrightarrow{\varphi^{[L]}\left(\cdot\,;\vartheta^{[L]}\right):\mathbb{R}^{n_{L-1}}\to\mathbb{R}^{n_L}} \hat{y} = a^{[L]} \in \mathbb{R} \to c = \left(\hat{y}-y\right)^2 \in \mathbb{R}$$

Note that this is the adjoint equation for the adjoints $\overline{a^{[l]}} = \dfrac{\partial c}{\partial a^{[l]}}$

- Therefore, we can calculate all the (adjoints) $\dfrac{\partial c}{\partial a^{[l]}}$:

$$\frac{\partial c}{\partial a^{[1]}} = \frac{\partial a^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[4]}}{\partial a^{[3]}} \cdots \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}} = \frac{\partial \varphi^{[2]}\left(a^{[1]};\vartheta^{[2]}\right)}{\partial a^{[1]}} \frac{\partial \varphi^{[3]}\left(a^{[2]};\vartheta^{[3]}\right)}{\partial a^{[2]}} \frac{\partial \varphi^{[4]}\left(a^{[3]};\vartheta^{[4]}\right)}{\partial a^{[3]}} \cdots \frac{\partial \varphi^{[L]}\left(a^{[L-1]};\vartheta^{[L]}\right)}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}}$$

$$\frac{\partial c}{\partial a^{[2]}} = \frac{\partial a^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[4]}}{\partial a^{[3]}} \cdots \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}} = \frac{\partial \varphi^{[3]}\left(a^{[2]};\vartheta^{[3]}\right)}{\partial a^{[2]}} \frac{\partial \varphi^{[4]}\left(a^{[3]};\vartheta^{[4]}\right)}{\partial a^{[3]}} \cdots \frac{\partial \varphi^{[L]}\left(a^{[L-1]};\vartheta^{[L]}\right)}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}}$$

$$\frac{\partial c}{\partial a^{[3]}} = \frac{\partial a^{[4]}}{\partial a^{[3]}} \cdots \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}} = \frac{\partial \varphi^{[4]}\left(a^{[3]};\vartheta^{[4]}\right)}{\partial a^{[3]}} \cdots \frac{\partial \varphi^{[L]}\left(a^{[L-1]};\vartheta^{[L]}\right)}{\partial a^{[L-1]}} \frac{\partial c}{\partial a^{[L]}}$$

$$\cdots$$

$$\frac{\partial c}{\partial a^{[l]}} = \left[\prod_{i=l}^{L-1} \frac{\partial a^{[i+1]}}{\partial a^{[i]}}\right] \frac{\partial c}{\partial a^{[L]}} = \left[\prod_{i=l}^{L-1} \frac{\partial \varphi^{[i+1]}\left(a^{[i]};\vartheta^{[i+1]}\right)}{\partial a^{[i]}}\right] \frac{\partial c}{\partial a^{[L]}}$$

where we recall that all the $\dfrac{\partial \varphi^{[i+1]}\left(a^{[i]};\vartheta^{[i+1]}\right)}{\partial a^{[i]}}$ are known, we "just" need to compute their matrix products

- We have our formula: $\dfrac{\partial c}{\partial a^{[l]}} = \left[\displaystyle\prod_{i=l}^{L-1} \dfrac{\partial \varphi^{[i+1]}\left(a^{[i]}; \vartheta^{[i+1]}\right)}{\partial a^{[i]}} \right] \dfrac{\partial c}{\partial a^{[L]}} = \left[\displaystyle\prod_{i=l}^{L-1} \dfrac{\partial \varphi^{[i+1]}\left(a^{[i]}; \vartheta^{[i+1]}\right)}{\partial a^{[i]}} \right] \dfrac{\partial c}{\partial \hat{y}}$

- And we know that $c = (\hat{y} - y)^2$ so $\dfrac{\partial c}{\partial a^{[L]}} = \dfrac{\partial c}{\partial \hat{y}} = 2(\hat{y} - y)$ "two prediction errors", which we recall is a real number

- We must compute $\displaystyle\prod_{i=l}^{L-1} \dfrac{\partial \varphi^{[i+1]}\left(a^{[i]}; \vartheta^{[i+1]}\right)}{\partial a^{[i]}}$ for every layer l, where we recall that all the $\dfrac{\partial \varphi^{[i+1]}\left(a^{[i]}; \vartheta^{[i+1]}\right)}{\partial a^{[i]}}$ are known from:

  - The architecture of the network, which gives the form all the vector functions φ, hence, also their Jacobians
  - The parameters θ on the current iteration
  - The activations a computed while making predictions
    (prior to computing derivatives, we must compute the predictions and store all the activations on all the layers)

- What is the most efficient way to compute this matrix product $\displaystyle\prod_{i=l}^{L-1} \dfrac{\partial \varphi^{[i+1]}\left(a^{[i]}; \vartheta^{[i+1]}\right)}{\partial a^{[i]}}$ knowing all the matrices in the product?

- For example, let us compute the derivatives to the 1st layer: $\dfrac{\partial c}{\partial a^{[1]}} = \dfrac{\partial a^{[2]}}{\partial a^{[1]}} \dfrac{\partial a^{[3]}}{\partial a^{[2]}} \dfrac{\partial a^{[4]}}{\partial a^{[3]}} \cdots \dfrac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \dfrac{\partial a^{[L]}}{\partial a^{[L-1]}} \dfrac{\partial c}{\partial a^{[L]}}$

- We could compute it left to right, layer *1* to layer *L*, in the order of the feed-forward equations

- Or, we could compute it right to left, layer *L* to layer *1*, in the reverse order

- Note that $a^{[L]} = \hat{y}$ and $c = (\hat{y} - y)^2$ are real numbers, so the rightmost term $\dfrac{\partial c}{\partial a^{[L]}} = \dfrac{\partial c}{\partial \hat{y}} = 2(\hat{y} - y)$ is also a real number

and the second rightmost term $\dfrac{\partial a^{[L]}}{\partial a^{[L-1]}} = \dfrac{\partial \hat{y}}{\partial a^{[L-1]}}$ is a vector of dim n_{L-1}, the gradient of the prediction to the basis functions in the final hidden layer

- All the other terms $\dfrac{\partial a^{[l]}}{\partial a^{[l-1]}}, 2 \leq l < L$ are matrices of shape $n_{l-1} \times n_l$

- Crucially, the result $\dfrac{\partial c}{\partial a^{[1]}}$ is a vector (of dim *n1*) because the loss *c* is scalar

- So:

$$\underbrace{\left[\frac{\partial c}{\partial a^{[1]}}\right]}_{vector} = \underbrace{\left[\frac{\partial a^{[2]}}{\partial a^{[1]}}\right]}_{matrix}\underbrace{\left[\frac{\partial a^{[3]}}{\partial a^{[2]}}\right]}_{matrix}\underbrace{\left[\frac{\partial a^{[4]}}{\partial a^{[3]}}\right]}_{matrix}\cdots\underbrace{\left[\frac{\partial a^{[L-1]}}{\partial a^{[L-2]}}\right]}_{matrix}\underbrace{\left[\frac{\partial a^{[L]}}{\partial a^{[L-1]}}\right]}_{vector}\underbrace{\frac{\partial c}{\partial a^{[L]}}}_{scalar}$$

forward calculation →

← backward calculation

- Forward calculation:
  - Start with a matrix
  - Repeatedly multiply matrices by matrices to obtain matrices, which is of <u>cubic</u> complexity
  - Collapse to a vector by right multiplication with a vector on the final hidden layer

- Backward calculation:
  - Start with a vector
  - Repeatedly multiply matrices by vectors to obtain vectors, which is of <u>quadratic</u> complexity

- Backward calculation is one order of magnitude faster!

Back-propagation algorithm

- Back-prop: algorithm to compute derivatives in the reverse order, one order of magnitude faster

- Algorithm:

1. Compute prediction by feed-forward through the network, storing all hidden activations *al*

2. Compute the adjoints $\overline{a^{[l]}} = \partial c / \partial a^{[l]}$ of all the layer activations in the reverse order

   a. Start with $\overline{a^{[L]}} = \dfrac{\partial c}{\partial a^{[L]}} = \dfrac{\partial c}{\partial \hat{y}} = 2(\hat{y} - y)$

   b. Recursively compute $\overline{a^{[l]}} = \dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial a^{[l+1]}}{\partial a^{[l]}} \dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial \varphi^{[l+1]}\left(a^{[l]}; \vartheta^{[l+1]}\right)}{\partial a^{[l]}} \dfrac{\partial c}{\partial a^{[l+1]}} = \underbrace{\dfrac{\partial \varphi^{[l+1]}\left(a^{[l]}; \vartheta^{[l+1]}\right)}{\partial a^{[l]}}}_{\text{known}} \underbrace{\overline{a^{[l+1]}}}_{\text{known by backward recursion}}$ from layer *L-1* to layer *1*

   Note we get all the $\overline{a^{[l]}} = \partial c / \partial a^{[l]}$ in a single backward sweep, computing them all from one another

   Also recall the Jacobian matrices $\dfrac{\partial \varphi^{[l+1]}\left(a^{[l]}; \vartheta^{[l+1]}\right)}{\partial a^{[l]}}$ are known from the architecture, current weights and pre-calculated activations

3. Compute the desired derivatives $\dfrac{\partial c}{\partial \vartheta^{[l]}} = \dfrac{\partial a^{[l]}}{\partial \vartheta^{[l]}} \dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial \varphi^{[l]}\left(a^{[l-1]}; \vartheta^{[l]}\right)}{\partial \vartheta^{[l]}} \dfrac{\partial c}{\partial a^{[l]}} = \dfrac{\partial \varphi^{[l]}\left(a^{[l-1]}; \vartheta^{[l]}\right)}{\partial \vartheta^{[l]}} \overline{a^{[l]}}$ where, again, $\dfrac{\partial \varphi^{[l]}\left(a^{[l-1]}; \vartheta^{[l]}\right)}{\partial \vartheta^{[l]}}$ is known

Back-propagation = AD

- The core of back-prop is the recursive reverse equation: $\overline{a^{[l]}} = \frac{\partial a^{[l+1]}}{\partial a^{[l]}} \overline{a^{[l+1]}}$

- We have seen this equation before and called it <u>adjoint equation</u>

- Back-prop is the application of the adjoint equation to neural nets

- **Back-prop (also called AD) is not limited to neural nets**

- **It is applicable to <u>any calculation</u> as long as:**
  - The final result is <u>scalar</u>, so adjoints are <u>vectors</u> and not matrices
  - The computation is organized in <u>layers</u>, where every layer is a known function of the previous layer only
  - We will see that any computation can be organized in layers, this is called the <u>computation graph</u>

- So back-prop (AD) is applicable to any computation, **as long as the final result is scalar**

- And it always computes all adjoints (derivatives) in a single (backward) sweep through the calculation graph

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{\varphi^{[1]}\left(\cdot;\vartheta^{[1]}\right):\mathbb{R}^{n_0} \to \mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{\varphi^{[2]}\left(\cdot;\vartheta^{[2]}\right):\mathbb{R}^{n_1} \to \mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{\varphi^{[3]}\left(\cdot;\vartheta^{[3]}\right):\mathbb{R}^{n_2} \to \mathbb{R}^{n_3}} \ldots \xrightarrow{\varphi^{[l]}\left(\cdot;\vartheta^{[l]}\right):\mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} \ldots \xrightarrow{\varphi^{[L]}\left(\cdot;\vartheta^{[L]}\right):\mathbb{R}^{n_{L-1}} \to \mathbb{R}^{n_L}} \hat{y} = a^{[L]} \in \mathbb{R} \to c = \left(\hat{y} - y\right)^2 \in \mathbb{R}$$

- MLP is a particular case where:

$$\varphi^{[l]}\left(a^{[l-1]};W^{[l]},b^{[l]}\right) = g^{[l]}\left(W^{[l]}a^{[l-1]}+b^{[l]}\right)$$

scalar activation function
for layer l
applied element-wise

vector of biases
for layer l

matrix of weights
for layer l

- It is convenient to split each layer in two: a linear combination followed by an activation:

$$(1) \quad z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \quad \longrightarrow \quad (2) \quad a^{[l]} = g^{[l]}\left(z^{[l]}\right)$$

- And we have the adjoint equations in the reverse order:

$$(1) \quad \overline{a^{[l-1]}} \equiv \frac{\partial c}{\partial a^{[l-1]}} = \frac{\partial z^{[l]}}{\partial a^{[l-1]}}\frac{\partial c}{\partial z^{[l]}} \equiv \frac{\partial z^{[l]}}{\partial a^{[l-1]}}\overline{z^{[l]}} \quad \longleftarrow \quad (2) \quad \overline{z^{[l]}} \equiv \frac{\partial c}{\partial z^{[l]}} = \frac{\partial a^{[l]}}{\partial z^{[l]}}\frac{\partial c}{\partial a^{[l]}} \equiv \frac{\partial a^{[l]}}{\partial z^{[l]}}\overline{a^{[l]}}$$

- Where $\dfrac{\partial z^{[l]}}{\partial a^{[l-1]}}$ and $\dfrac{\partial a^{[l]}}{\partial z^{[l]}}$ are given by a direct application of matrix calculus (see next slide)

feed-forward

$$a^{[0]} = x \quad \rightarrow \quad \cdots \quad \rightarrow \quad a^{[l-1]} \quad \rightarrow \quad z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \quad \rightarrow \quad a^{[l]} = g^{[l]}\left(z^{[l]}\right) \quad \rightarrow \quad \cdots \quad \rightarrow \quad \hat{y} = a^{[L]} \quad \rightarrow \quad c = \left(a^{[L]} - y\right)^2$$

$$\overline{a^{[0]}} \quad \leftarrow \quad \cdots \quad \leftarrow \quad \overline{a^{[l-1]}} = \frac{\partial z^{[l]}}{\partial a^{[l-1]}} \overline{z^{[l]}} \quad \leftarrow \quad \overline{z^{[l]}} = \frac{\partial a^{[l]}}{\partial z^{[l]}} \overline{a^{[l]}} \quad \leftarrow \quad \overline{a^{[l]}} \quad \leftarrow \quad \overline{a^{[L]}} = 2\left(\hat{y} - y\right)$$

back-prop

from matrix calculus

$$\frac{\partial z^{[l]}}{\partial a^{[l-1]}} = W^{[l]\,\mathrm{T}} \qquad\qquad \frac{\partial a^{[l]}}{\partial z^{[l]}} = diag\left[g^{[l]\,\prime}\left(z^{[l]}\right) \right] \quad \text{so} \quad \overline{z^{[l]}} = g^{[l]\,\prime}\left(z^{[l]}\right) \otimes \overline{a^{[l]}} \quad \text{where } \otimes \text{ denotes element-wise product}$$
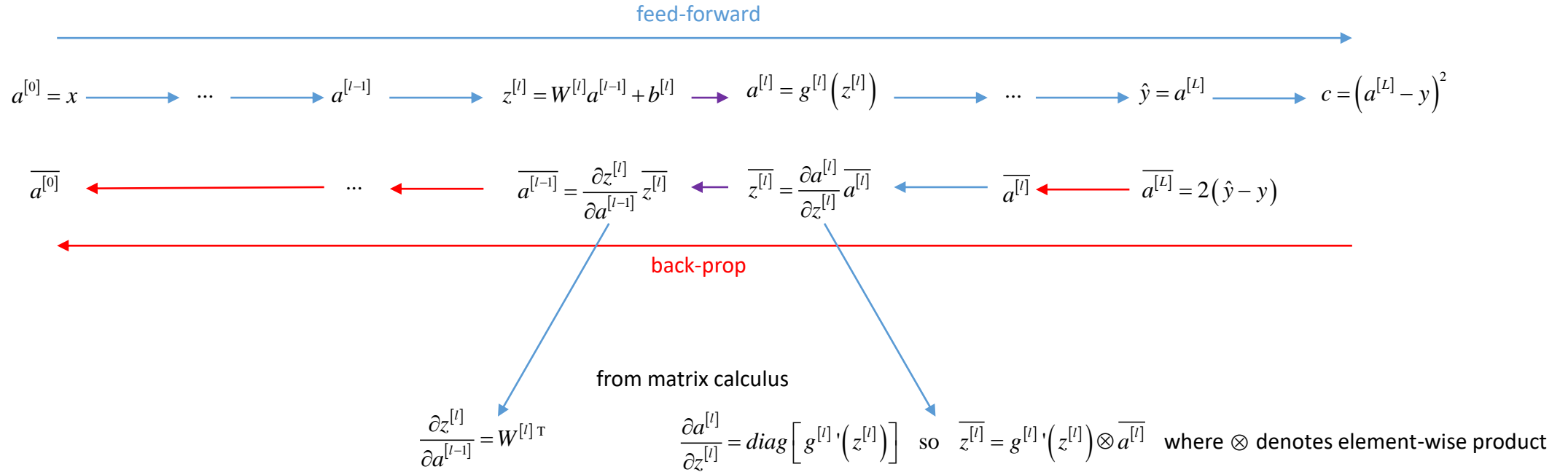
and finally (also from matrix calculus) we have the sensitivities to weights from the sensitivities to activation:

$$\frac{\partial c}{\partial W^{[l]}} = \overline{z^{[l]}} a^{[l-1]\,\mathrm{T}} \qquad\qquad\qquad \frac{\partial c}{\partial b^{[l]}} = \frac{\partial z^{[l]}}{\partial b^{[l]}} \frac{\partial c}{\partial z^{[l]}} = \frac{\partial c}{\partial z^{[l]}} = \overline{z^{[l]}}$$

because: $\quad \dfrac{\partial c}{\partial W_i^{[l]}} = \dfrac{\partial z_i^{[l]}}{\partial W_i^{[l]}} \dfrac{\partial c}{\partial z_i^{[l]}} = \dfrac{\partial z_i^{[l]}}{\partial W_i^{[l]}} \overline{z_i^{[l]}} = \overline{z_i^{[l]}} a^{[l-1]\,\mathrm{T}}$

MLP back-propagation algorithm

- Which gives us the MLP back-prop, generally described in literature and implemented in frameworks

- Algorithm:

1. Compute prediction by feed-forward through the network, storing <u>all hidden combinations *zl* and activations *al*</u>

2. Compute the adjoints $\overline{a^{[l]}} = \partial c/\partial a^{[l]}$ and $\overline{z^{[l]}} = \partial c/\partial z^{[l]}$ of all the layer combinations and activations in the reverse order

   a. Start with $\overline{a^{[L]}} = 2(\hat{y} - y)$

   b. Recursively compute $\overline{z^{[l]}} = g^{[l]}{}'\left(z^{[l]}\right) \otimes \overline{a^{[l]}}$ <u>and then</u> $\overline{a^{[l-1]}} = W^{[l]\,\mathrm{T}} \overline{z^{[l]}}$ from layer *L-1* to layer *1*

3. Compute the desired derivatives $\dfrac{\partial c}{\partial W^{[l]}} = \overline{z^{[l]}} a^{[l-1]\,\mathrm{T}}$ and $\dfrac{\partial c}{\partial b^{[l]}} = \overline{z^{[l]}}$ of loss to weights and biases

Back-propagation: complexity

- Recall the leading complexity of feed-forward is a matrix by vector product by layer of quadratic complexity in the number of units

- We need a prior feed-forward before back-prop to compute and store all the *zl* and *al*

- The back-prop equations are executed once on every layer (in reverse order) to compute derivatives to activations and weights:

  - $\overline{z^{[l]}} = g^{[l]}{}'\left(z^{[l]}\right) \otimes \overline{a^{[l]}}$ is of linear complexity (element-wise multiplication of two vectors)

  - $\overline{a^{[l-1]}} = W^{[l]\,\mathrm{T}} \overline{z^{[l]}}$ is of quadratic complexity (matrix by vector product)

  - $\dfrac{\partial c}{\partial W^{[l]}} = \overline{z^{[l]}} a^{[l-1]\,\mathrm{T}}$ is of quadratic complexity

  - $\dfrac{\partial c}{\partial b^{[l]}} = \overline{z^{[l]}}$ is immediate

Complexity and memory considerations

- So the complexity of a complete back-prop to compute all derivatives is 3 times the complexity of a feed-forward prediction:
  - One necessary feed-forward prediction to compute and store all the *zl* and *al*
  - The back-propagation itself takes one matrix by vector product, quadratic in units, like feed-forward (but in reverse order)
  - Finally, the computation of derivatives to weights adds another matrix by vector product on each layer
- As promised, we compute a large number of derivatives in constant time
  - The actual back-prop is as fast as <u>one</u> prediction (the other 2 are for the prior prediction and derivatives to weights)
  - Independently on the number of derivatives
- It works because:
  - The end result is <u>scalar</u>, so its derivatives to vectors are <u>vectors</u>
  - Reverse adjoint propagation collapses matrix by vector products into vectors, so we have quadratic complexity, like feed-forward
  - Derivatives are computed analytically in one sweep, there is no need to repeatedly bump many parameters one by one
- It consumes significant memory
  - All combinations and activations must be stored for all layers
  - In addition to all the weights and biases

Automatic backprop?

- Reverse adjoint propagation
  - Also called back-propagation or back-prop in ML lingo, or **adjoint differentiation** (AD) in general
  - Computes all differentials of the cost to the many parameters in a deep net in a time similar to one evaluation of the cost
  - Achieves this remarkable result by reversing the order of the calculations in the differentiation
  - Is the only viable means to train deep nets in reasonable time
  - Relies on the cost being scalar, hence its adjoints are vectors
  - **Applies to any scalar calculation, not just neural nets**

- However manual AD code is
  - Complicated and prone to error
  - Painful to maintain and synchronize as we modify feed-forward code

- AAD: **automatic** adjoint differentiation
  - Next, we learn to **automate** AD
  - Since AD propagates adjoints in the reverse order through evaluation graphs
  - We will learn to automatically generate graphs so we can back-prop derivatives through them
  - Not only for deep nets, but for any calculation
  - Including large Monte-Carlo simulations

- First, we illustrate the application of deep learning to financial derivatives

Deep learning in finance

- We reuse the same example
  as for linear and quadratic regression

- Simulate data
  for the spot in 1y
  and a call payoff in 2y

- Train a neural net to find
  the future price of the call in 1y
  as a function of the spot prevailing in 1y

- See if it can learn Black & Scholes
  from data alone

max(S2-K,0)

S1

Epoch 50
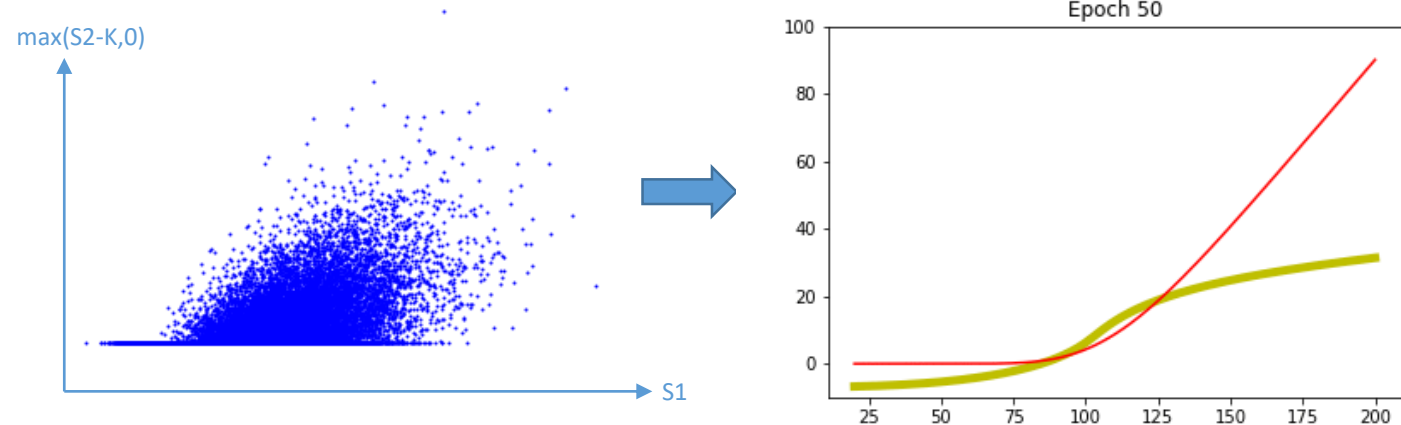
- The net learns Black & Scholes from the data cloud alone
- The red line (Black & Scholes's formula) is only for the viewer's reference
- The net doesn't know anything about Black & Scholes at any stage

Simple ANNs with TensorFlow

- TensorFlow
  - Machine learning framework open sourced by Google
  - The most popular framework for machine learning and deep learning

- What is TensorFlow exactly?
  - Another math/numerical/matrix library with API in Python
  - With 3 perks (in increasing order of perkness)
    1. High level functions to work with ANNs easily and conveniently
    2. Matrix calculus executed on multi-core CPU and GPU
    3. Built-in back-prop so TF computes derivatives of your code, in constant time, behind the scenes (a form of AAD)

- Working with TensorFlow
  - To do all this, TF needs to know computation graphs: all computations with order and dependency, <u>before</u> they are executed
  - Hence, working with TF is surprising at first, you must:
    - First "register" the calculations with TF, that is, build the graph in TF's memory space
    - Then, ask TF to run the graph, and back-prop through the graph when derivatives are needed

Simple example walkthrough

- Head to http://GitHub/asavine/CompFinance/Workshop/

- Run the notebook
  dlBlackScholes.ipynb
  (TensorFlow must be installed)
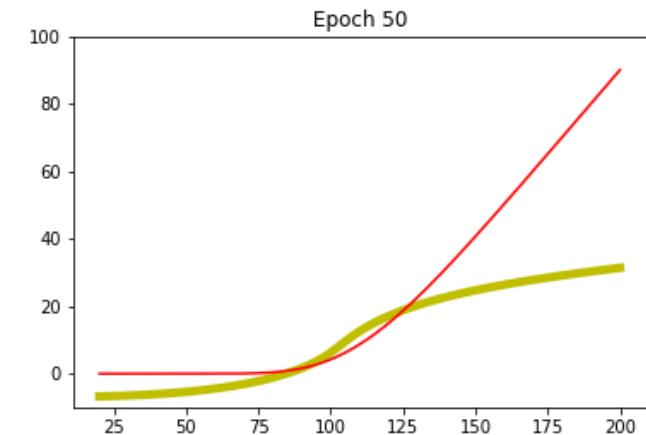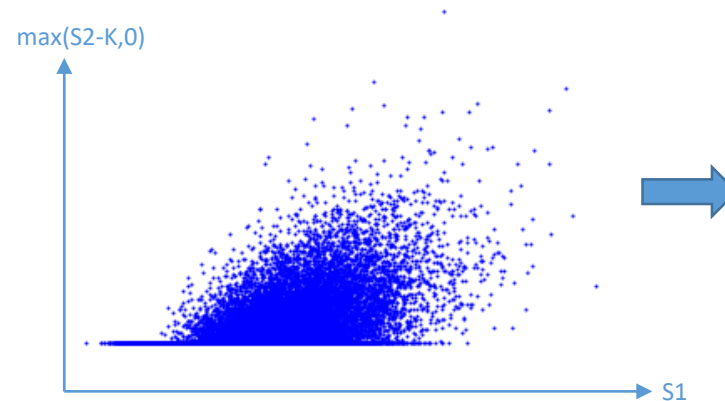
- This notebook demonstrates
  - The (basic) operation of TensorFlow
  - How to build and train (simple) neural nets in practice
  - Application to financial derivatives (revaluation)
  - How to implement a simple example in Black & Scholes

Learning TensorFlow
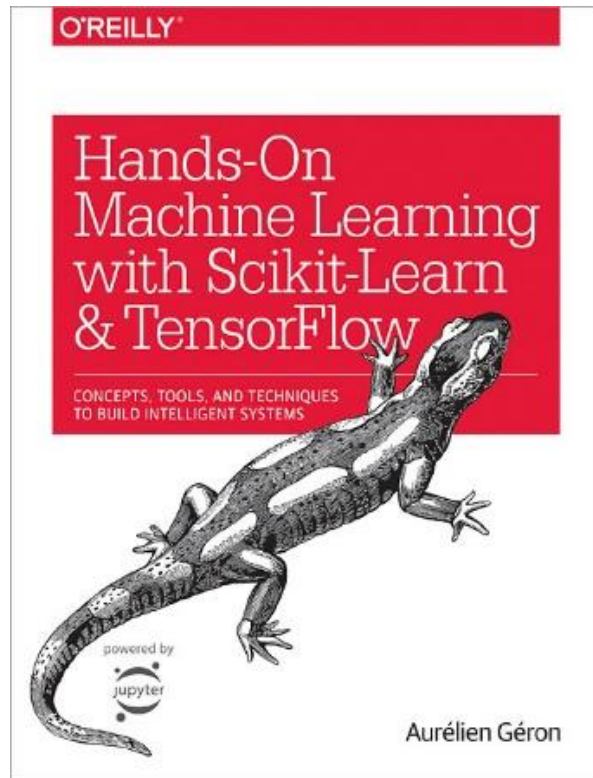
Summer 2019
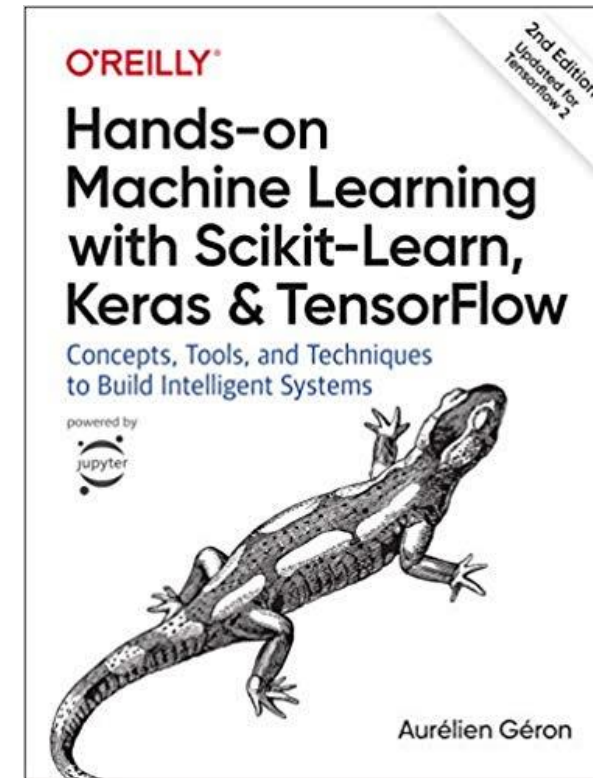
Revaluation nets

- What we just saw is a (very simple) example of a <u>revaluation</u> problem:
  What is the <u>future</u> value in <u>simulated</u> scenarios of a derivatives transaction or book of transactions?

- Vast domain of application:
  - Pricing callable options and exotics with Monte-Carlo
    - Simulate state variables up to the exercise date
    - Apply the net to find the <u>continuation</u> value on the exercise date
    - Determine if the call is exercised or not in this scenario
    - Extension of the well-known "Least Square Method" of Longstaff-Schwartz and Carriere (late 1990s) to modern deep learning
  - More importantly, regulatory simulations like XVA, MVA, CCR or FRTB
    - Simulate the future value of derivatives books in a large number of scenarios to compute exposures
    - Normally require nested simulations (MC to estimate book value inside MC to simulate market to exposure date!)
    - Revaluation nets offer alternative orders of magnitude faster and more accurate
  - Read the intro of the notebook in http://GitHub/asavine/CompFinance/Workshop/dlBlackScholes.ipynb

Revaluation nets (2)

- Revaluation nets are disposable
    - Trained for a given set of transactions in a given model, calibrated to current market data
    - Market, model and transaction parameters get encoded in the weights of the network
    - Only input is simulated state vector
    - The input is only valid with the market, model and transaction it was trained with
    - Revaluation nets are trained once, used once
    - Hence, training must be very fast (seconds)

- Revaluation nets are very general, work with models and transactions of any complexity

- Revaluation nets resolve key problems with regulatory simulations, in particular XVA, FRTB, CCR and MVA

- Upcoming article "Deep Analytics" by Brian Huge and Antoine Savine

Pricing nets

- An <u>apparently</u> similar problem is <u>pricing</u>

- Context:
  - Given a dynamic model with a number of parameters
    - Current market: asset prices, interest rates, etc.
    - Dynamic parameters: volatilities, volatilities of volatilities, correlations, etc.
    - Not an exotic model a la Dupire with vast (term structures) of parameters
    - But a European model a la SABR/Heston/RoughBergomi with a limited number of parameters
  - And some transaction, also with a number of parameters
    - Generally, European option or first order exotic (barrier) with few parameters like expiry, strike, barrier

- Pricing problem:
  - Only simplest models like Black & Scholes have explicit closed form solutions
  - More realistic models require
    - Expansions/approximations (SABR/ZABR)
    - Numerical integration (Heston)
    - Multi-Factor FDM
    - Slow Monte-Carlo simulations (RoughVol)

Pricing nets (2)

- Can we do better with a neural net?
  - Generate a vast training set, sampling the space of all market, model and transaction parameters
  - For each sampled parameter set, produce a price, e.g. with Monte-Carlo simulations
  - Note the training set alone may take hours or days
  - Train a neural net to generalize pricing from the training set
  - Apply the trained net as a very fast, very accurate ~almost closed form~ pricer

- Train once, use forever

- Very different from reval nets
  - Train on <u>prices</u>, not <u>realizations</u>: training set is sampled, not simulated (sampling correctly is perhaps the major challenge)
  - May train for hours or days, forever reusable
  - Weights only encode pricing mechanics. Market, dynamic and transaction parameters are inputs
  - One net per model (e.g. McGhee's SABRNet for SABR or Horvath and al's RoughNet for RoughBergomi)
  - Limited to European models and transactions, not likely to extend to exotics

Pricing nets (3)

- Recent results

  - McGhee (2018) built a SABR net, more accurate than Hagan's approximation and orders of magnitude faster than 2D FDM

  - Ferguson and Greene (also 2018) built a net for pricing basket options with parameters (inputs) including correlation

  - Horvath and al. (2019) built a net for pricing with <u>Rough Volatility</u>
    - This is particularly significant
    - Because there exist no alternatives for pricing with rough vol other than slow Monte-Carlo
    - RoughNet therefore opens new applications for calibration, pricing and risk management with rough vol
    - So far, rough vol was restricted to academic circles due to absence of fast pricing, it may now enter production

Deeper learning for financial derivatives

	Reval nets	Pricing nets
Output	Future price depending on scenario	Current price
Inputs	Simulated state variables	Market, model and transaction parameters
Training set	State variables to **payoffs** = realizations per scenario	Model parameters to **prices** = expectations across scenarios, per parameter set
Weight encoding	Valuation mechanics + market, model and transaction parameters	Valuation mechanics only
Applications	Callable exotics with MC XVA, MVA, FRTB, CCR, etc.	Europeans and first order exotics Stochastic volatility/rough vol/jump/etc. Calibration/pricing/risk management
Reuse	Disposable	Reusable
Model and transaction scope	Any complexity	Restricted to Europeans and simple exotics
Accuracy requirement	Medium	High
Training time	Seconds	Hours
Reference nets	DeepAnalytics (Huge & Savine, coming 2019)	McGhee's SABRNet (2018) Ferguson & Greene's BasketNet (2018) Horvath and al's RoughNet (2019)

Deeper learning

- We studied in detail the fundamental machinery of feed-forward and back-propagation

- There is a lot more to deep learning:
  - Speed and accuracy improvements: regularization, dropout, batch-norm, stochastic gradient descent, etc.
  - Convolutional nets with many impressive applications in computer vision
  - Recurrent nets with many impressive applications in natural language processing (NLP)
    And also time series forecasting, proprietary trading strategies, etc.
  - Reinforcement learning, leveraging DL to learn strategies for playing games, driving cars or trading currencies

- Deep learning is here to stay
  - Overhyped due to recent successes in computer vision, NLP and chess/go --- hype will fade
  - This being said, DL genuinely offers efficient, elegant solutions to otherwise intractable problems in finance and elsewhere
  - DL will not replace financial models but it will be part of the financial quant toolbox
  - Financial quants and derivatives professionals cannot ignore it

AD through evaluation graphs

Evaluation graphs and adjoint propagation

- Evaluation graph
  - We have seen evaluation graphs for neural nets
  - Evaluation graphs express the order and dependencies of "atomic" calculations involved in a larger evaluation
  - Any evaluation defines a graph
  - Hence, every evaluation defines a sequence of feed-forward operations, similarly to a neural net
  - Chapter 9 of the Modern Computational Finance book builds evaluation graphs in memory with operator overloading in C++

- Adjoint Differentiation is back-propagation over graphs
  - Not limited to neural nets, applicable to any evaluation
  - Propagates adjoints backwards through the evaluation graph
  - Offers constant time efficiency (all differentials for an expense similar to one evaluation) for scalar calculations
    (A scalar calculation is one where the final result –or output layer, is scalar)

Feed-forward to compute results: $a^{[l+1]} = g^{[l]}\left(a^{[l]}\right)$ $\quad\quad a^{[0]}$: known inputs

(compact form) evaluation graph $\quad\quad a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{g^{[1]}:\mathbb{R}^{n_0}\to\mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{g^{[2]}:\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{g^{[3]}:\mathbb{R}^{n_2}\to\mathbb{R}^{n_3}} ... \xrightarrow{g^{[l]}:\mathbb{R}^{n_{l-1}}\to\mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} ... \xrightarrow{g^{[L]}:\mathbb{R}^{n_{L-1}}\to\mathbb{R}^{n_L}} a^{[L]} \in \mathbb{R}$

Back-propagate to compute derivatives: $\overline{a^{[l]}} = g'\left(a^{[l]}\right)\overline{a^{[l+1]}}$ $\quad\quad \overline{a^{[L]}} = 1$
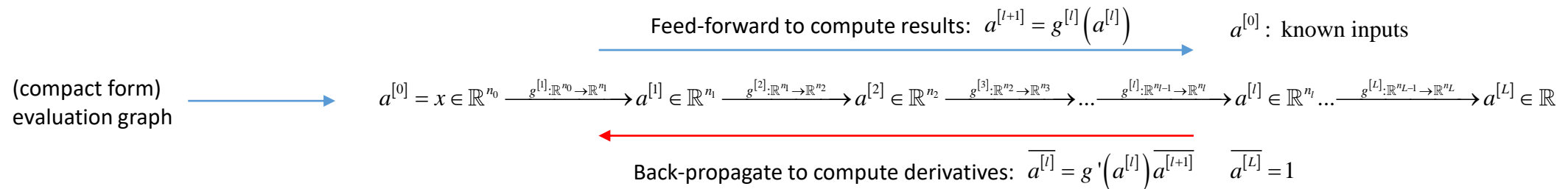
the step functions g are "building blocks" of calculation, which Jacobians are known --- __not__ activations (we are no longer in DL)

Example: Black & Scholes

- Black & Scholes' formula: $C(S_0, r, y, \sigma, K, T) = DF\left[FN(d_1) - KN(d_2)\right]$ with

  - Discount factor to maturity: $DF = \exp(-rT)$

  - Forward: $F = S_0 \exp\left[(r-y)T\right]$

  - Standard deviation: $std = \sigma\sqrt{T}$

  - Log-moneyness: $d = \dfrac{\log\left(\dfrac{F}{K}\right)}{std}$

  - D's: $d_1 = d + \dfrac{std}{2}$, $d_2 = d - \dfrac{std}{2}$

  - Probabilities to end in the money, resp. under spot and risk-neutral measures: $P_1 = N(d_1)$, $P_2 = N(d_2)$

  - Call price: $C = DF\left[FP_1 - KP_2\right]$

Black & Scholes: evaluation graph

Inputs	Temp. calc.	Temp. calc.	Temp. calc.	Temp. calc.	Result
S_0	DF	DF	DF	DF	C
r	F	F	F	F	
y	std	std	d_1	P_1	
σ	K	d	d_2	P_2	
K		K	K	K	
T					

Input layer l=0	Hidden layer l=1	Hidden layer l=2	Hidden layer l=3	Hidden layer l=4	Output layer l=5

$$a^{[0]} = \left(S_0, r, y, \sigma, K, T \right)^T$$

$$n_0 = 6$$

$$a^{[1]} = \left(DF, F, std, K \right)^T$$
$$= g^{[1]}\left(a^{[0]} \right)$$
$$n_1 = 4$$

$$a^{[2]} = \left(DF, F, std, d, K \right)^T$$
$$= g^{[2]}\left(a^{[1]} \right)$$
$$n_2 = 5$$

$$a^{[3]} = \left(DF, F, d_1, d_2, K \right)^T$$
$$= g^{[3]}\left(a^{[2]} \right)$$
$$n_3 = 5$$

$$a^{[4]} = \left(DF, F, P_1, P_2, K \right)^T$$
$$= g^{[4]}\left(a^{[3]} \right)$$
$$n_4 = 5$$

$$a^{[5]} = C$$
$$= g^{[5]}\left(a^{[4]} \right)$$
$$n_5 = 1$$
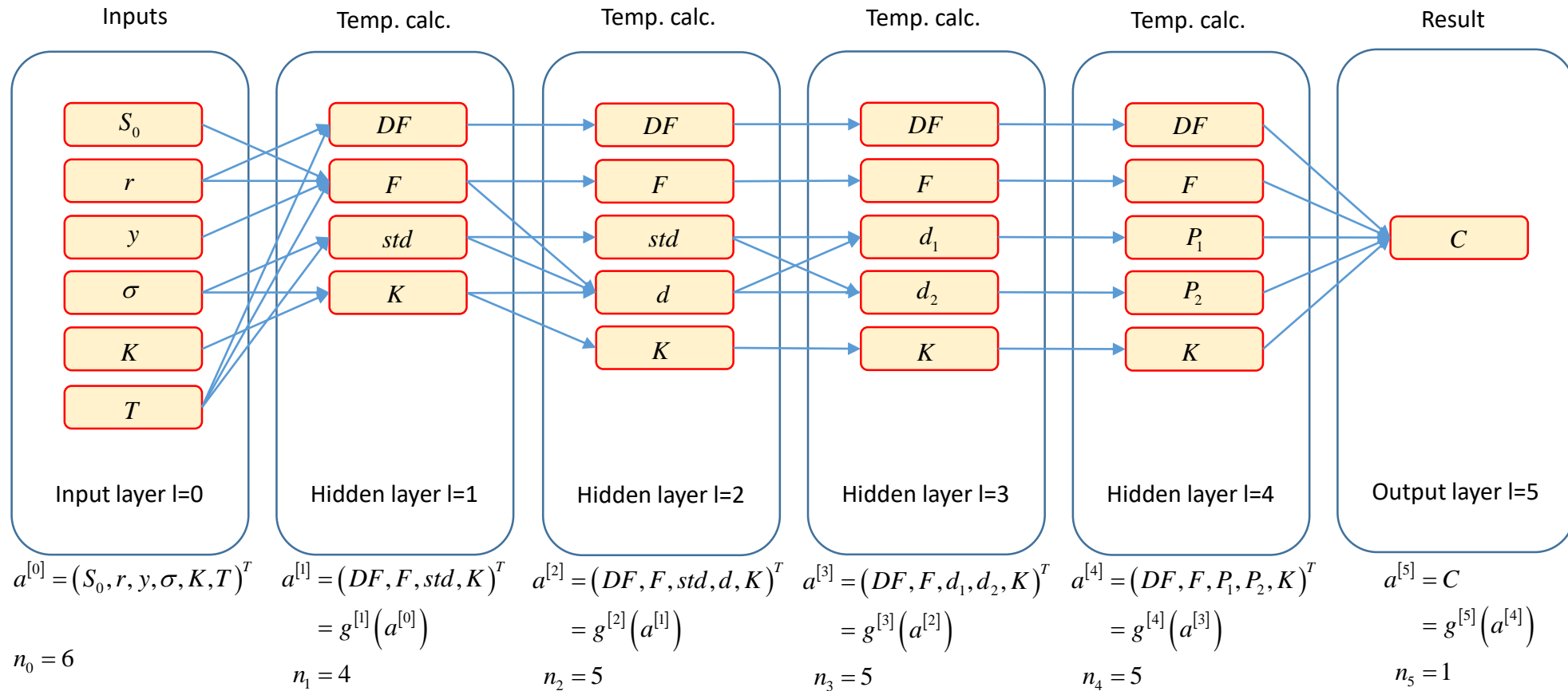
Feed-forward equations

- As in deep nets, for **any** calculation, we have the feed-forward equation defining the evaluation graph:

inputs: $a^{[0]} = x$

feed-forward: $a^{[l]} = g^{[l]}\left(a^{[l-1]}\right)$

result: $y = a^{[L]}$

- In Black & Scholes, we have the steps:

$$g^{[1]}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} \exp\left(-x_2 x_6\right) \\ x_1 \exp\left[\left(x_2 - x_3\right) x_6\right] \\ x_4 \sqrt{x_6} \\ x_5 \end{pmatrix} \quad g^{[2]}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \dfrac{\log\left(x_2/x_4\right)}{x_3} \\ x_4 \end{pmatrix} \quad g^{[3]}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_4 + \dfrac{x_3}{2} \\ x_4 - \dfrac{x_3}{2} \\ x_5 \end{pmatrix} \quad g^{[4]}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ N\left(x_3\right) \\ N\left(x_4\right) \\ x_5 \end{pmatrix} \quad g^{[5]}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = x_1\left(x_2 x_3 - x_5 x_4\right)$$

Back-prop equations

- For **any** calculation, where we compute the differentials of the result to the inputs: $\dfrac{\partial y}{\partial x} = \dfrac{\partial a^{[L]}}{\partial a^{[0]}}$

- If we split the calculation into elementary pieces g, we know the Jacobians $g^{[l]'} = \dfrac{\partial a^{[l]}}{\partial a^{[l-1]}}$

- In Black & Scholes

$$g^{[1]'}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 0 & -x_6\exp(-x_2 x_6) & 0 & 0 & 0 & -x_2\exp(-x_2 x_6) \\ \exp[(x_2-x_3)x_6] & x_1 x_6\exp[(x_2-x_3)x_6] & -x_1 x_6\exp[(x_2-x_3)x_6] & 0 & 0 & x_1(x_2-x_3)\exp[(x_2-x_3)x_6] \\ 0 & 0 & 0 & \sqrt{x_6} & 0 & \dfrac{x_4}{2\sqrt{x_6}} \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}^T$$

$$g^{[2]'}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \dfrac{1}{x_2 x_3} & -\dfrac{\log(x_2/x_4)}{x_3^{\,2}} & -\dfrac{1}{x_3 x_4} \\ 0 & 0 & 0 & 1 \end{pmatrix}^T$$

$$g^{[3]'}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \dfrac{1}{2} & 1 & 0 \\ 0 & 0 & -\dfrac{1}{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T$$

$$g^{[4]'}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & n(x_3) & 0 & 0 \\ 0 & 0 & 0 & n(x_4) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T$$

$$g^{[5]'}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \big((x_2 x_3 - x_5 x_4),\, x_1 x_3,\, x_1 x_2,\, -x_1 x_5,\, -x_1 x_1 x_4\big)^T$$

- Note that the Jacobian of the output layer is a vector, as with any scalar calculation

Back-propagation algorithm

- We have identified the calculation graph with all its steps *g* and their Jacobians *g'*

- We have run it once and stored all the intermediate results *a*

- We can compute all the derivatives of in constant time by back-prop:

$$\overline{a^{[L]}} = 1 \qquad \overline{a^{[l]}} = g'\left(a^{[l]}\right)\overline{a^{[l+1]}} \qquad \frac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$$

$$a^{[L]} = \frac{\partial a^{[L]}}{\partial a^{[L]}} = 1 \rightarrow seeding$$

$$picking \ results$$

$$\overline{a^{[l]}} = \frac{\partial a^{[L]}}{\partial a^{[l]}} = \underbrace{\frac{\partial a^{[l+1]}}{\partial a^{[l]}} \frac{\partial a^{[L]}}{\partial a^{[l+1]}}}_{chain \ rule} = g'\left(a^{[l]}\right)\overline{a^{[l+1]}} \rightarrow backprop$$

- Not that interesting with Black & Scholes: only 6 inputs/derivatives: delta, vega, rho, …

- The exact same equations apply with large evaluations taking 6,000 inputs --- very interesting then

- Left as an exercise:
  Show that Black & Scholes' Jacobians of the previous slide lead to the well known Black-Scholes "Greeks"

"Atomic" building blocks

- What we just did is:
  - Split a calculation

$$x = \left(S_0, r, y, \sigma, K, T\right) \longrightarrow c = g(x) = BS\left(S_0, r, y, \sigma, K, T\right)$$

  - Into a sequence of simpler calculations

$$x \longrightarrow a^{[1]} = g^{[1]}(x) \longrightarrow a^{[2]} = g^{[2]}\left(a^{[1]}\right) \longrightarrow a^{[3]} = g^{[3]}\left(a^{[2]}\right) \longrightarrow a^{[4]} = g^{[4]}\left(a^{[3]}\right) \longrightarrow c = g^{[5]}\left(a^{[4]}\right)$$

  - Then, we applied back-propagation to compute derivatives

$$\overline{x} = g^{[1]}{}'(x)\overline{a^{[1]}} \longleftarrow \overline{a^{[1]}} = g^{[2]}{}'\left(a^{[1]}\right)\overline{a^{[2]}} \longleftarrow \overline{a^{[2]}} = g^{[3]}{}'\left(a^{[2]}\right)\overline{a^{[3]}} \longleftarrow \overline{a^{[3]}} = g^{[4]}{}'\left(a^{[3]}\right)\overline{a^{[4]}} \longleftarrow \overline{a^{[4]}} = g^{[5]}{}'\left(a^{[4]}\right)\overline{c} \longleftarrow \overline{c} = 1$$

- Pushing this logic to limit, we can:
  - Split <u>any</u> calculation into a sequence of "atomic" building blocks: +, -, *, /, pow(), exp(), log(), sqrt(), sin(), cos(), …
  - Note that we have a very limited number of building blocks (about 20 in total)
  - **And, yes, any <u>numerical</u> calculation is, always, a (long) sequence of these 20-something building blocks!**
  - This is another mind-twisting fact, see Rolf Poulsen's Wilmott article: antoinesavine.files.wordpress.com/2018/12/poulsen-2018-wilmott.pdf
  - For these 20-something building blocks, we know the derivatives g' : exp' = exp, log' = 1/x, sqrt' = 1 / 2sqrt, etc.
  - Therefore, we can code these 20-something derivatives once and for all, and automate back-prop ➔ this gives us AAD
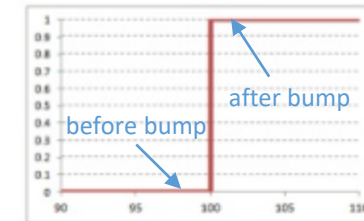
Control flow

It is a somewhat vertiginous realization that any calculation, like the Monte-Carlo estimate of the counterparty credit value adjustment on a netting set of thousands of exotic transactions, is still a sequence of elementary operations. A very, very long sequence, but a sequence all the same.

Rolf Poulsen

f(x) = 1 if x > 100, else 0

- Mathematical equations are more than sequences of building blocks
  We also have limits, differentials, integrals, etc.

- But their numerical implementation is always nothing more, nothing less than a sequence of building block operations…

$$FD: f'(x) \approx \frac{f(x+h) - f(x)}{h} = unstable$$

- … and <u>control flow</u>, also called *branching*: "if this then that" statements

- Control flow makes functions discontinuous, hence, not differentiable

$$AD: f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} = 0$$

- Attempts to differentiate control flow with bumping results into instabilities

- AD/AAD computes exact differentials – same as <u>infinitesimal</u> bumps

- So derivatives are always taken on the same side of the control flow, and not through control flow

  Smoothing
  - Approx *f* by close continuous func
  - Instabilities/incorrectness vanish

- This is not always the desired result, so something to take into account

- The solution is <u>smoothing</u>, see
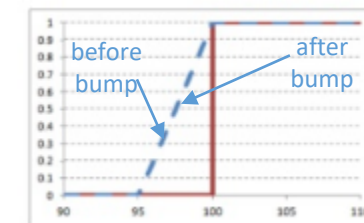 https://www.slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic

Conclusion

- All calculations define a graph, down to 20-something "atomic" building blocks gs

- Like deep nets, all evaluation graphs may be arranged in successive layers, each an elementary function g of the previous one (which Jacobians are known):

  inputs: $a^{[0]} = x$

  feed-forward: $a^{[l]} = g^{[l]}\left(a^{[l-1]}\right)$

  result: $y = a^{[L]}$

- Once the evaluation graph and its steps g are known, along with their Jacobians g' (and the graph was run once so intermediate results a were stored)

- All derivatives $\dfrac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$ can be computed in constant time by back-prop $\overline{a^{[L]}} = 1$, $\overline{a^{[l]}} = g'\left(a^{[l]}\right)\overline{a^{[l+1]}}$

- If we can automatically produce a graph from a code, then we can automate the whole process

AAD

$$\overline{a^{[L]}} = 1, \overline{a^{[l]}} = g'\left(a^{[l]}\right)\overline{a^{[l+1]}}$$

$$\frac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$$

we don't know <u>yet</u> how to do that

back-prop (AD) we know this already

all derivatives, analytic and in constant time

code

graph
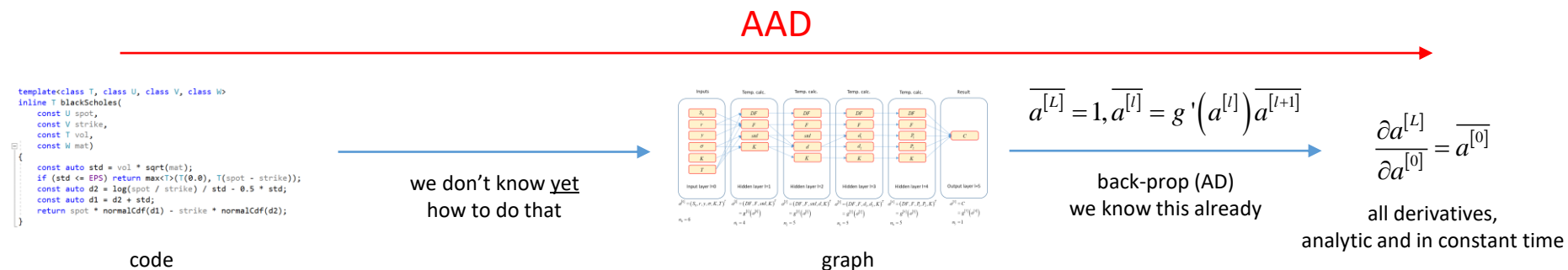
Recording calculations on tape

C++ code

- From there on, we work with C++ code

- We only work with simplistic, beginner level C++

- We show that we can still implement AAD with rather spectacular results

- We refer to the Modern Computational Finance book
  for a professional, generic, parallel, efficient implementation

- **All the code is freely available on GitHub, in the file toyCode.h**

https://github.com/asavine

Automatic differentiation

- Differentiation by finite differences

  - Computes derivatives by running evaluation repeatedly

  - Only requires executable evaluation code

  - Linear complexity in the number of differentials

- AAD

  - Computes derivatives by running AD in reverse order over an evaluation graph

  - Requires a graph to traverse – executable code is not enough

  - Constant complexity in the number of differentials : fast differentiation of scalar code

  - To implement AAD we must extract an evaluation graph: sequence of calculations (nodes) and their dependencies (edges)

Building evaluation graphs

- Explicit evaluation graphs

    - Solution implemented in TensorFlow
    - Lets users explicitly build graphs by calling graph creation functions exported to many languages
    - Then ask a TensorFlow <u>session</u> to evaluate or differentiate the graphs efficiently: evaluation and AD, on multicore CPU or GPU

    - Smart and efficient
    - But forces developers to explicitly build graphs in place of calculation code
    - (TensorFlow has a nice API that makes building graphs somewhat similar to coding calculations)
    - But what we want here is take some calculation code and automatically extract its graph

- Source transformation

    - Code that reads and understands evaluation code, then builds graphs and writes backward differentiation code automatically
    - Complex, specialized work similar to writing compilers
    - Variant: template meta-programming and introspection (introduced in chapter 15)

Operator overloading

- Alternative: operator overloading (in languages that support it like C++)

- When code applies operators (like + or *) or math functions (like log or sqrt) to real numbers (*double* type) the corresponding operations are evaluated immediately (or *eagerly*):

```
double x = 1, y = 2;        //  x and y are doubles
double z = x + y;           //  evaluates x + y and stores result in z
double t = log(x);          //  evaluates log(x) and stores result in t
```

- When the same operators are applied to **custom types** (our own type to store real numbers) <u>we</u> decide exactly what happens:

```
class myNumberType
{
    //  class definition here
};

myNumberType operator+(const myNumberType& lhs, const myNumberType& rhs)
{
    //  this code is executed anytime two numbers of type myNumberType are added
}

myNumberType log(const myNumberType& arg)
{
    //  this code is executed anytime log is called on a number of type myNumberType
}

myNumberType x, y;           //  x and y are of type myNumberType
myNumberType z = x + y;      //  executes code in the overloaded operator+()
myNumberType t = log(x);     //  executes code in the overloaded log()
```

Recording operations

- We apply operator overloading to **record** all operations along with their dependencies:

```cpp
class myNumberType
{
    myNumberType(const double x)
    {
        //  constructor initializes value to x and records node
    }
};

myNumberType operator+(const myNumberType& lhs, const myNumberType& rhs)
{
    recordAddition(lhs, rhs);    //  records addition with dependency on lhs and rhs
}

myNumberType log(const myNumberType& arg)
{
    recordLog(arg);              //  records log with dependency on arg
}

myNumberType x = 1, y = 2;       //  initializes x to 1 and y to 2 and records them
myNumberType z = x + y;          //  records addition with dependencies on x and y
myNumberType t = log(x);         //  records log with dependency on x
myNumberType result = t + z;     //  records addition with dependency on t and z
```

Lazy evaluation

- When this code is executed:

```
myNumberType x = 1, y = 2;      //  initializes x to 1 and y to 2 and records them
myNumberType z = x + y;         //  records addition
myNumberType t = log(x);        //  records log
myNumberType result = t + z;    //  records addition
```

- Nothing is calculated, instead the following graph is recorded in memory:
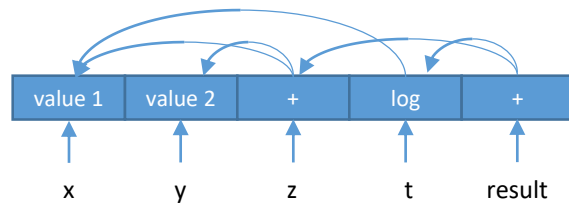
- This sequence can be evaluated later (*lazy evaluation*) or differentiated (applying AD from back to front)

- This is how we build the evaluation graph at run time, by executing calculation code with a custom type for which operators are overloaded to perform recording

Conventional implementation

- We introduce the conventional implementation of AAD
  - Where every math operation: +, -, *, /, pow, log, exp, sqrt, … is recorded on a data structure called **tape**
  - (It is really an evaluation graph but we call it "tape" in AAD lingo)

- Cutting-edge implementation records whole expressions
  - With template meta-programming and expression templates
  - Resulting in 2x to 5x faster code
  - Sometimes called "tape compression"
  - Extreme instance: "tapeless AD" where we don't record anything!
  - All implemented in professional code on GitHub and explained in detail in chapter 15
  - In this presentation, we stick with simpler, conventional implementation

- We record every mathematical operation involved in a calculation
  - Every calculation, up to complex Monte-Carlo simulations, is a sequence of +, -, *, /, pow, log, exp, sqrt, … !
  - We record every single one
  - **Note that all operations have either 0, 1 or 2 arguments**

Simplistic implementation

- In this presentation, we focus on the simplicity of the code
  - We use basic C++ code and disregard efficiency, scalability and best practice
  - Our aim is to explain the key ideas with simplistic code
  - This code works, just not efficiently

- In the book, on the contrary, we build professional, scalable, efficient code in modern C++

- This is particularly important for AAD because of:
  - Recording overhead
    - Every addition, multiplication, etc. produces a record
    - Recording necessarily involves an overhead
    - An efficient implementation must minimize overhead and make recording as efficient as possible
  - Vast memory consumption
    - We store in memory all the operations involved in a large calculation, this is a very large number of records
    - Estimated RAM consumption around 5GB per second
    - Efficient memory and cache management are key to an efficient implementation

- An efficient implementation of conventional AAD is given and explained in chapter 10
  - Effective, custom memory management
  - Recording with minimum overhead, cache efficiency and so on

Record and tape data structures

- A record
  - Stores one operation $y = f(x)$ where $f = +, *, -, /, \log, \sqrt{}, \dots$
  - Knows the number and location of the 0, 1 or 2 arguments x_i
  - Stores the 0, 1 or 2 partial derivatives to its arguments $\partial f / \partial x_i$ so we can apply AD

- The tape stores the sequence of records

```
struct Record
{
    int     numArg;        //  number of arguments: 0, 1 or 2
    int     idx1;          //  index of first argument on tape
    int     idx2;          //  index of second argument on tape
    double  der1;          //  partial derivative to first argument
    double  der2;          //  partial derivative to second argument
};

//  The tape, declared as a global variable
vector<Record> tape;
```

Custom real number

- Our custom number
  - Holds its value and
  - Knows the index of the corresponding operation on tape
  - May be initialized with a value
    to create a record (without arguments) on tape

```cpp
struct Number
{
    double  value;
    int     idx;

    //  default constructor does nothing
    Number() {}

    //  constructs with a value and record
    Number(const double& x) : value(x)
    {
        //  create a new record on tape
        tape.push_back(Record());
        Record& rec = tape.back();

        //  reference record on tape
        idx = tape.size() - 1;

        //  populate record on tape
        rec.numArg = 0;
    }
};
```

- We overload all mathematical operators and functions to:
  - Evaluate and store result as usual
  - Additionally, record the operation and its derivatives on tape
  - So code is evaluated and recorded at the same time

```cpp
Number operator+(const Number& lhs, const Number&rhs)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = lhs.value + rhs.value; //  calling double overload

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 2;
    rec.idx1 = lhs.idx;
    rec.idx2 = rhs.idx;

    //  compute derivatives, both derivatives of addition are 1
    rec.der1 = 1;
    rec.der2 = 1;

    return result;
}
```

Operator overloading

- Similarly, we overload -, *, and /
- Same code exactly, only values and derivatives change

```cpp
Number operator-(const Number& lhs, const Number&rhs)
{
 //  ...

 //  compute value
    result.value = lhs.value - rhs.value;

 //  ...

 //  compute derivatives
    rec.der1 = 1;
    rec.der2 = -1;

 //  ...
}
```

```cpp
Number operator*(const Number& lhs, const Number&rhs)
{
 //  ...

 //  compute value
    result.value = lhs.value * rhs.value;

 //  ...

 //  compute derivatives
    rec.der1 = rhs.value;
    rec.der2 = lhs.value;

 //  ...
}
```

```cpp
Number operator/(const Number& lhs, const Number&rhs)
{
 //  ...

 //  compute value
    result.value = lhs.value / rhs.value;

 //  ...

 //  compute derivatives
    rec.der1 = 1.0 / rhs.value;
    rec.der2 = - lhs.value / (rhs.value * rhs.value);
 //  ...
}
```

On-class operator overloading

- We must also overload +=, -=, *- and /=, as well as unary + and -, on class

The final Number class is therefore:

```cpp
struct Number
{
    double  value;
    int     idx;

    //  default constructor does nothing
    Number() {}

    //  constructs with a value and record
    Number(const double& x) : value(x)
    {
        //  create a new record on tape
        tape.push_back(Record());
        Record& rec = tape.back();

        //  reference record on tape
        idx = tape.size() - 1;

        //  populate record on tape
        rec.numArg = 0;
    }

    Number operator +() const { return *this; }
    Number operator -() const { return Number(0.0) - *this; }

    Number& operator +=(const Number& rhs) { *this = *this + rhs; return *this; }
    Number& operator -=(const Number& rhs) { *this = *this - rhs; return *this; }
    Number& operator *=(const Number& rhs) { *this = *this * rhs; return *this; }
    Number& operator /=(const Number& rhs) { *this = *this / rhs; return *this; }
};
```

Function overloading

- Similarly, we overload log, exp, sqrt

- We should really overload all standard math functions

- Code is identical for all functions, only value and derivatives change

```cpp
Number log(const Number& arg)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = log(arg.value);

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 1;
    rec.idx1 = arg.idx;

    //  compute derivative
    rec.der1 = 1.0 / arg.value;

    return result;
}
```

```cpp
Number exp(const Number& arg)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = exp(arg.value);

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 1;
    rec.idx1 = arg.idx;

    //  compute derivative
    rec.der1 = result.value;

    return result;
}
```

```cpp
Number sqrt(const Number& arg)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = sqrt(arg.value); //  calling double overload

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 1;
    rec.idx1 = arg.idx;

    //  compute derivative
    rec.der1 = 0.5 / result.value;

    return result;
}
```

Custom function overloading

- Also overload building blocks that are not standard to C++ but frequently applied in applications

- In financial application, we use cumulative normal distributions and normal densities all the time

- The functions are defined in the file gaussians.h in the repo

- We overload here (once again, only change is in value and derivatives):

```cpp
Number normalDens(const Number& arg)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = normalDens(arg.value);

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 1;
    rec.idx1 = arg.idx;

    //  compute derivative
    rec.der1 = - result.value * arg.value;

    return result;
}
```

```cpp
Number normalCdf(const Number& arg)
{
    //  create a new record on tape
    tape.push_back(Record());
    Record& rec = tape.back();

    //  compute result
    Number result;
    result.value = normalCdf(arg.value); //  calling double overload in gaussians.h

    //  reference record on tape
    result.idx = tape.size() - 1;

    //  populate record on tape
    rec.numArg = 1;
    rec.idx1 = arg.idx;

    //  compute derivative
    rec.der1 = normalDens(arg.value);

    return result;
}
```

Avoiding code duplication

- The code for all binary operators and for all unary functions is identical

- Only values and derivatives are different

- This is poor design:
  If we change something in the logic, we must consistently modify many different functions

- In the professional code of chapters 10 and 15
  we structure code and apply "policy design" (Alexandresku, 2001) to avoid duplication without overhead

- Here, we stick with duplicated code

Comparison operator overloading

- Our custom number must do everything a double does

- In particular, we must be able to compare two Numbers

- Hence, to complete our simple framework, we must also overload comparison operators:

```cpp
bool operator==(const Number& lhs, const Number& rhs) { return lhs.value == rhs.value; }
bool operator!=(const Number& lhs, const Number& rhs) { return lhs.value != rhs.value; }
bool operator>(const Number& lhs, const Number& rhs) { return lhs.value > rhs.value; }
bool operator>=(const Number& lhs, const Number& rhs) { return lhs.value >= rhs.value; }
bool operator<(const Number& lhs, const Number& rhs) { return lhs.value < rhs.value; }
bool operator<=(const Number& lhs, const Number& rhs) { return lhs.value <= rhs.value; }
```

Applying the recording framework

- Our simple recording framework is complete, see complete code in the GitHub repo, file toyCode.h

- We may use it to record calculations

- Example: Black and Scholes

```cpp
inline double blackScholes(
    //  input layer 0
    const double spot, const double rate, const double yield, const double vol, const double strike, const double mat)
{
/* layer 1 */        double df = exp(-rate * mat), fwd = spot * exp((rate - yield) * mat), std = vol * sqrt(mat);
/* layer 2 */        double d = log(fwd / strike) / std;
/* layer 3 */        double d1 = d + 0.5 * std, d2 = d - 0.5 * std;
/* layer 4 */        double p1 = normalCdf(d1), p2 = normalCdf(d2);
/* output layer 5 */ return df * (fwd * p1 - strike * p2);
}
```

Instrumenting computation code

- To record a Black & Scholes calculation, we must call it with our number type

- This is called *instrumentation*

- We can replace all doubles by Numbers:

```
inline Number blackScholes(
    //  input layer 0
    const Number spot, const Number rate, const Number yield, const Number vol, const Number strike, const Number mat)
{
/* layer 1 */         Number df = exp(-rate * mat), fwd = spot * exp((rate - yield) * mat), std = vol * sqrt(mat);
/* layer 2 */         Number d = log(fwd / strike) / std;
/* layer 3 */         Number d1 = d + 0.5 * std, d2 = d - 0.5 * std;
/* layer 4 */         Number p1 = normalCdf(d1), p2 = normalCdf(d2);
/* output layer 5 */ return df * (fwd * p1 - strike * p2);
}
```

Instrumenting computation code

- Better solution: template code on number representation type

```cpp
template <class T> inline T blackScholes(
    //  input layer 0
    const T spot, const T rate, const T yield, const T vol, const T strike, const T mat)
{
/* layer 1 */        T df = exp(-rate * mat), fwd = spot * exp((rate - yield) * mat), std = vol * sqrt(mat);
/* layer 2 */        T d = log(fwd / strike) / std;
/* layer 3 */        T d1 = d + 0.5 * std, d2 = d - 0.5 * std;
/* layer 4 */        T p1 = normalCdf(d1), p2 = normalCdf(d2);
/* output layer 5 */ return df * (fwd * p1 - strike * p2);
}
```

- Best practice: produce templated code in the first place

- To evaluate only, call with doubles as arguments

```cpp
double spot = 100, rate = 0.02, yield = 0.05, vol = 0.2, strike = 110, mat = 2; // initializes inputs
auto result = blackScholes(spot, rate, yield, vol, strike, mat);                 // evaluates operations
```

- To evaluate and record code, call with Numbers as arguments

```cpp
Number spot = 100, rate = 0.02, yield = 0.05, vol = 0.2, strike = 110, mat = 2; // initializes and records inputs
auto result = blackScholes(spot, rate, yield, vol, strike, mat);                 // evaluates and records operations
```

Tape vs graph

- Operator overloading gives us a tape of the mathematical operations involved in a calculation

- After execution of the templated code for Black & Scholes, we get the following tape:
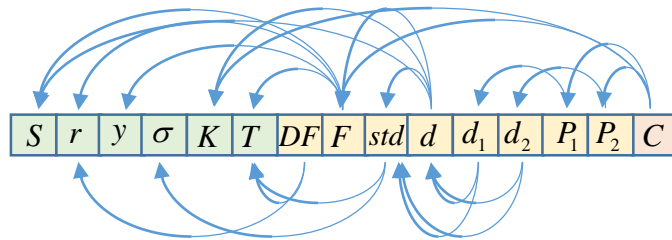
Where we recall that operator overloading also gave us:

- The operations from ancestors A (arguments) to successors S (results)

- And their derivatives $\dfrac{\partial S}{\partial A}$

- A tape is not exactly the same as an evaluation graph, where operations are neatly arranged by layer:
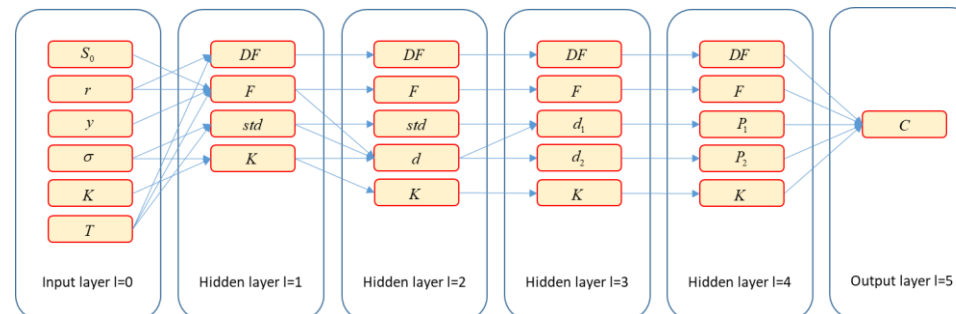
Tape vs graph (2)

- The tape and the graph however express the exact same information, differently

- We can always turn a tape into a graph or vice-versa

- If we had a graph, we could run back-prop to compute all derivatives in constant time

Feed-forward to compute results: $a^{[l+1]} = g^{[l]}\left(a^{[l]}\right)$

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{g^{[1]}:\mathbb{R}^{n_0}\to\mathbb{R}^{n_1}} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{g^{[2]}:\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{g^{[3]}:\mathbb{R}^{n_2}\to\mathbb{R}^{n_3}} \ldots \xrightarrow{g^{[l]}:\mathbb{R}^{n_{l-1}}\to\mathbb{R}^{n_l}} a^{[l]} \in \mathbb{R}^{n_l} \ldots \xrightarrow{g^{[L]}:\mathbb{R}^{n_{L-1}}\to\mathbb{R}^{n_L}} a^{[L]} \in \mathbb{R}$$

Back-propagate to compute derivatives: $\overline{a^{[l]}} = g\,'\left(a^{[l]}\right)\overline{a^{[l+1]}}$

- We could turn our tape into a graph, but this would be a waste of precious computational resources

- Instead, we work with a slightly different version of the back-prop/AD algorithm

- So we can back-propagate adjoints directly through the tape (although it is <u>not</u> arranged in layers)

- Using the notions of *ancestor* (argument to an operation) and *successor* (result of an operation)

- AAD is the sum of a *taping framework* (e.g. operator overloading) and back-propagation through the tape

AAD

Ancestors

- We call the instrumented instance of our code:

```
Number spot = 100, rate = 0.02, yield = 0.05, vol = 0.2, strike = 110, mat = 2; // initializes and records inputs
auto result = blackScholes(spot, rate, yield, vol, strike, mat);                // evaluates and records operations
cout << result.value;   //  5.03705
```
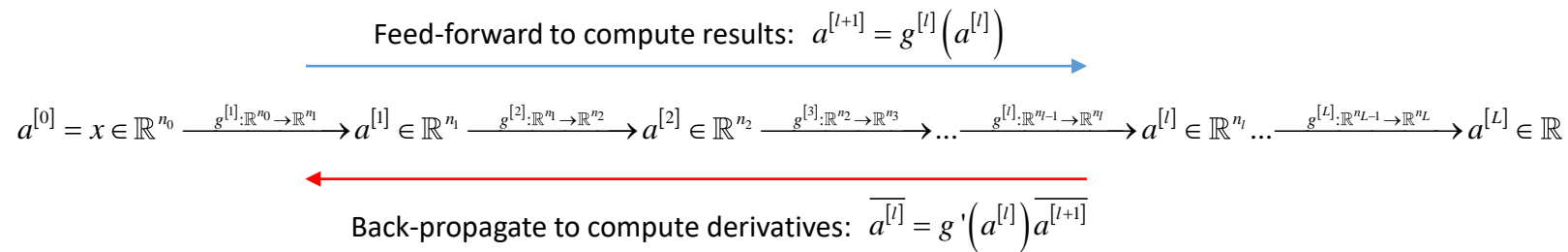
- Which evaluates the calculation and records all operations on tape
    - Denote f_i the operation number *i*, this is a function of 0, 1 or 2 arguments
    - The arguments must also be on tape, with indices < *i*
    - Denote A_i the set of indices of the ancestors (arguments) to f_i - this is a set of 0, 1 or 2 indices, all < *i*
    - Denote *n* the number of inputs and *N* the number of operations on tape, including inputs
    - Denote x_i the result of operation *i* and $y = x_N$ the final result

    - Note that all the local derivatives $\dfrac{\partial f_i}{\partial x_j}$ for all $j \in A_i$ have been computed and recorded on tape, on evaluation, by our operators

- Our tape therefore looks like:

$$x_i = f_i\left(x_{A_{i1}}, x_{A_{i2}}\right) \text{ where } \frac{\partial f_i}{\partial x_{A_{ij}}} \text{ are known and recorded}$$

119
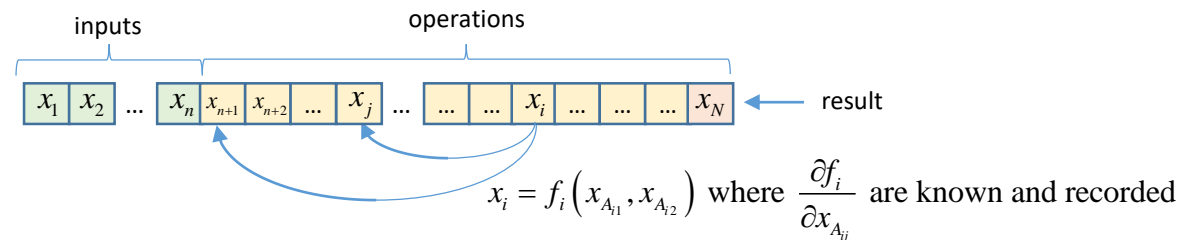
Successors

- Successors: *j* is a successor to *i* if *i* is an ancestor to *j*
  - Denote S_j the set of indices of the *successors* of x_j on tape
  - These are the indices *i* of all functions f_i , calculated **after** x_j that use x_j as an argument
  - Formally: $S_j = \{i > j, j \in A_i\}$
  - Note that S_j may contain many indices, although the size of A_i is 2 or less
  - Whereas an empty S_j reveals an unused input or intermediate result
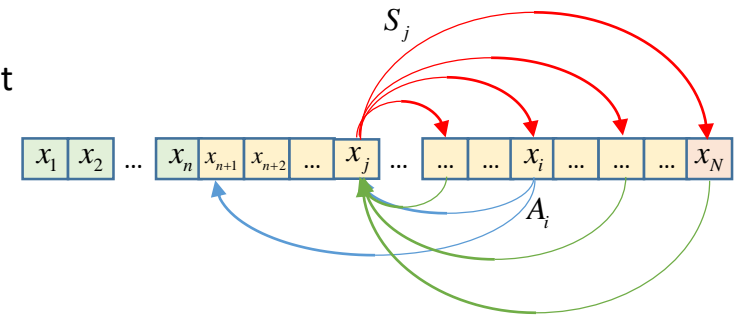
- Adjoint equation

  - Denote $\bar{x}_i \equiv \dfrac{\partial y}{\partial x_i} = \dfrac{\partial x_N}{\partial x_i}$ the adjoint of x_i

  - Then (evidently): $\bar{x}_N = 1$

  - And in a direct application of the chain rule: $\bar{x}_j = \sum_{i \in S_j} \dfrac{\partial f_i}{\partial x_j} \bar{x}_i$ because $\dfrac{\partial y}{\partial x_j} = \sum_{i \in S_j} \dfrac{\partial y}{\partial x_i} \dfrac{\partial x_i}{\partial x_j}$ and we recall that all the $\dfrac{\partial f_i}{\partial x_j}$ are all on tape

120

Adjoint accumulation V1

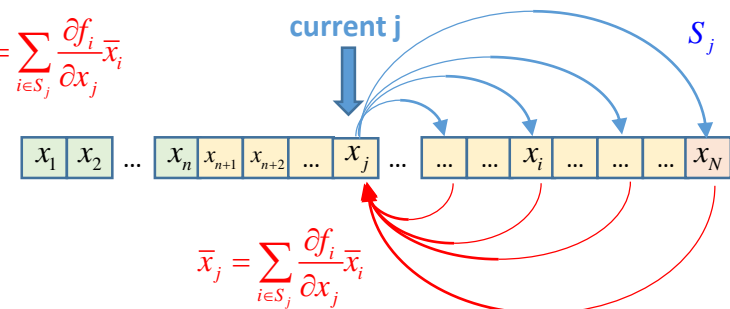- Adjoints therefore satisfy a backward recursion

  - The last adjoint $\bar{x}_N = 1$ is given

  - All other adjoints are a weighted sum of future adjoints: for all j $\quad \bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$

- Hence, the following (flawed) algorithm:

  Starting with $\bar{x}_N = 1$, compute for every j, in the reverse order, from N-1 to 1: $\quad \bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$

  Summing over the successors of xj:

  $$\bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$$
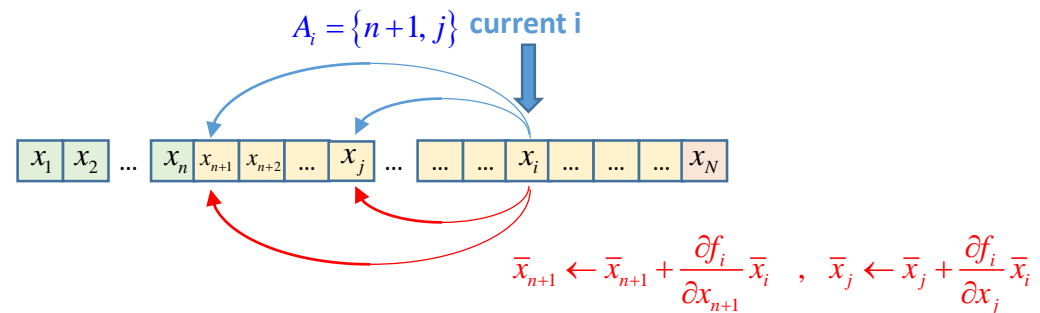
- Problems:
  - Complexity: a node may have many successors to sum over
  - Impracticality: ancestors don't store their successors, it is the successors who store their (up to 2) ancestors and partial derivatives

Adjoint accumulation V2

- A more efficient/practical means of computing the same numbers:

  - The last adjoint $\bar{x}_N = 1$ is still given

  - Initialize all other adjoints to 0

  - Traverse tape in the reverse order, as previously, accumulating the sum $\bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$ **from successors**

$A_i = \{n+1, j\}$ **current i**

$x_1 \quad x_2 \quad \dots \quad x_n \quad x_{n+1} \quad x_{n+2} \quad \dots \quad x_j \quad \dots \quad \dots \quad \dots \quad x_i \quad \dots \quad \dots \quad \dots \quad x_N$

$$\bar{x}_{n+1} \leftarrow \bar{x}_{n+1} + \frac{\partial f_i}{\partial x_{n+1}} \bar{x}_i \quad , \quad \bar{x}_j \leftarrow \bar{x}_j + \frac{\partial f_i}{\partial x_j} \bar{x}_i$$

- Problems resolved:
  - Max complexity 2: a node may have up to 2 ancestors
  - Successors who store their ancestors and partial derivatives

Adjoint accumulation algorithm

1. Initialize all adjoints to 0 and the last adjoint to 1 (this is called *seeding* the tape): $\bar{x}_j = \delta_{N-j}$

2. Repeat for *i* iterating backwards from *N* to *0:* $\text{for all } j \in A_i : \bar{x}_j \leftarrow \bar{x}_j + \dfrac{\partial f_i}{\partial x_j} \bar{x}_i$

3. The differentials of the calculation to its inputs $x_1, x_2, \ldots, x_n$ are, by definition, $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$

- This algorithm is called (reverse) adjoint propagation

- AAD is the sum of a recording framework and an implementation of adjoint propagation

Adjoint accumulation code

```cpp
vector<double> calculateAdjoints(Number& result)
{
    //  initialization
    vector<double> adjoints(tape.size(), 0.0);  //  initialize all to 0
    int N = result.idx;                          //  find N
    adjoints[N] = 1.0;                           //  seed aN = 1

    //  backward propagation
    for(int j=N; j>0; --j)  //  iterate backwards over tape
    {
        if (tape[j].numArg > 0)
        {
            adjoints[tape[j].idx1] += adjoints[j] * tape[j].der1;        //  propagate first argument

            if (tape[j].numArg > 1)
            {
                adjoints[tape[j].idx2] += adjoints[j] * tape[j].der2;    //  propagate second argument
            }
        }
    }

    return adjoints;
}
```

Application to Black & Scholes

- After we record a tape by calling the instrumented instance of our code:

```cpp
Number spot = 100, rate = 0.02, yield = 0.05, vol = 0.2, strike = 110, mat = 2; // initializes and records inputs
auto result = blackScholes(spot, rate, yield, vol, strike, mat);               // evaluates and records operations
cout << result.value;   //  5.03705
```

- We proceed with adjoint propagation:

```cpp
//  propagate adjoints
vector<double> adjoints = calculateAdjoints(result);

//  show derivatives
cout << "Derivative to spot (delta) = " << adjoints[spot.idx] << endl;        //  0.309
cout << "Derivative to rate (rho) = " << adjoints[rate.idx] << endl;         //  51.772
cout << "Derivative to dividend yield = " << adjoints[yield.idx] << endl;    //  -61.846
cout << "Derivative to volatility (vega) = " << adjoints[vol.idx] << endl;   //  46.980
cout << "Derivative to strike (-digital) = " << adjoints[strike.idx] << endl;   //  -0.235
cout << "Derivative to maturity (-theta) = " << adjoints[mat.idx] << endl;        //  1.321
```

- The code works nicely
  and produces correct values

Complexity

- One *evaluation* of the calculation
  - Sweeps forward through the sequence of its operations
  - Executes every operation exactly once
  - Therefore its complexity is N, the number of records that end up on tape

- Adjoint propagation
  - Also sweeps through the sequence of operations, backward
  - Executes 0, 1 or 2 operations on every record, depending on the number of arguments
  - Therefore, as advertised, AAD computes **all n** differentials in **constant time**
  - In theory, all differentials are propagated in less than 2x one evaluation
  - In addition, the calculation must evaluated (and recorded) first so the theoretical upper bound is 3x one evaluation
  - Due to recording and tape traversal overhead, a good implementation generally produces many differentials in 4x to 10x

- Our professional code from chapters 10, 12 and 15 beats the theoretical bound!
  - Recall from the demonstration, one evaluation = 0.8sec, 1,081 differentials = 1.5sec, less than 2x one evaluation
  - Due to "selective instrumentation", a strong optimization, explained, along many others, in chapter 12

Conclusion

- We implemented AAD in the simplest possible manner, scratching the surface of possibilities

- Part III (Chapters 8 to 15) gives the details of a complete, professional, efficient implementation
  - How to minimize recording overhead
  - Efficient memory management constructs
  - Apply check-pointed AAD to differentiate a calculation piece by piece to mitigate RAM footprint and cache inefficiency
  - Efficiently differentiate non-scalar calculations that return multiple results
  - Cutting-edge implementation with template meta-programming and expression templates, faster by 2x to 5x
  - Parallel implementation
  - Advise for debugging and optimization
  - And much more

- Still the simplistic code works and produces the correct values

- Not very interesting for Black & Scholes
  - Fast, analytic evaluation
  - Only 6 differentials to compute

- Next, we apply the framework to a barrier option in Dupire Monte-Carlo
  - Long, complex evaluation
  - 1,081 differentials to compute

AAD for financial simulations

Simple simulation code

- We implement a simplistic simulation code for a barrier option in Dupire's model

    - Recall local volatility is given in a matrix and bi-linearly interpolated in spot and time

- We need the following pieces, which we assume are given here

    - A matrix class to hold local volatilities (matrix.h in the repo, chapters 1 and 2 in the book)

    - A bi-linear interpolation function (interp.h in the repo, chapter 6, section 6.4 in the book)

    - Random number generators to produce independent Gaussian increments (chapters 5 and 6)

- The code is templated on the real number representation type

```cpp
//  Signature

template <class T>
inline T toyDupireBarrierMc(
    //  Spot
    const T              S0,
    //  Local volatility
    const vector<T>      spots,
    const vector<T>      times,
    const matrix<T>      vols,
    //  Product parameters
    const T              maturity,
    const T              strike,
    const T              barrier,
    //  Number of paths and time steps
    const int            Np,
    const int            Nt,
    //  Initialized random number generator
    RNG&                 random)
```

```cpp
//  Implementation

//  Initialize
T result = 0;
// double because the RNG is not templated (and doesn't need to be, see chapter 12)
vector<double> gaussianIncrements(Nt);
const T dt = maturity / Nt, sdt = sqrt(dt);

//  Loop over paths
for (int i = 0; i < Np; ++i)
{
    //  Generate Nt Gaussian Numbers
    random.nextG(gaussianIncrements);
    //  Euler's scheme, step by step
    T spot = S0, time = 0;
    bool alive = true;
    for (size_t j = 0; j < Nt; ++j)
    {
        //  Interpolate volatility
        const T vol = interp2D(spots, times, vols, spot, time);
        time += dt;
        //  Simulate return
        spot *= exp(-0.5 * vol * vol * dt + vol * sdt * gaussianIncrements[j]);
        //  Monitor barrier
        if (spot > barrier)
        {
            alive = false;
            break;
        }
    }
    //  Payoff
    if (alive && spot > strike) result += spot - strike;
}   //  paths

return result / Np;
```

A simplistic code

- This code "does the job" but is not acceptable by professional standards

- The code is specific to Dupire's model and an up & out call, therefore not scalable
  - To price another product (Asian option, Ratchet option, …) copy the code and change the lines that evaluate payoffs
  - To price in another model (Heston, SLV, …) copy the code and change the lines that generate the path
  - End up with many different functions implementing the same simulation logic for different couples of models and products
  - To modify the simulation logic, consistently change all the functions! This is obviously not viable

- Chapter 6 teaches a professional architecture for generic simulation libraries
  - Encapsulate scenario generation in Model objects
  - Encapsulate payoff evaluation in Product objects
  - Encapsulate simulation logic in a generic Monte-Carlo engine
  - Code every model and every product exactly once, mix and match at run time

- We stick with the simplistic code for demonstration purposes

An inefficient code

- The code does too much work repeatedly during simulations
  - Key to efficient Monte-Carlo code:
    Do as much work as possible once, on initialization, and as little as possible repeatedly, during simulations
  - Example: we perform an expensive bi-linear interpolation in the innermost loop, for every path, on every time step
  - We interpolate in spot and time, spot is stochastic (scenario dependent), time is not
  - Therefore we can (and should) pre-interpolate in time on initialization
    And perform only 1D interpolations in the innermost loop

- Chapter 6 teaches and builds fully optimized code

- The code is serial, it executes sequentially on one core
  - Since even our phones are multi-core today, professional code must be parallel
  - With Monte-Carlo simulations, it is relatively easy to obtain a speed-up by the number of physical cores

- Chapter 3 teaches modern parallel C++, chapter 7 builds a professional *parallel* simulation library
  And section 12.5 instruments it with AAD in parallel

- For the purpose of demonstration, we stick with the not so efficient, serial version

Smoothing barrier options

- Discretely monitored barrier options are discontinuous, their value jumps to 0 at the barrier
    - Therefore, our code is not differentiable
    - AAD does not help: it cannot perform the impossible task of differentiating a discontinuous function
    - With finite differences, barrier risks are unstable, with AAD they are all zero
    - Find the reason why as an exercise!

- Therefore traders always smooth discontinuous transactions (barriers, digitals etc.)

- Smoothing = applying a close, continuous approximation in place of the discontinuous function

- Smoothing in finance, and its connection to fuzzy logic, are explained in the presentation:

    http://www.slideshare.net\AntoineSavine\stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic

freely available on slideShare

- Here, we briefly explain the "smooth barrier" algorithm, usually applied on derivatives desks

Smooth barrier

- Hard barrier
  - 100% dead above the barrier, 100% alive below the barrier
  - Hence, discontinuous

- Soft barrier
  - 100% dead above the barrier **plus epsilon**, 100% alive below the barrier **minus epsilon**
  - In between, lose a fraction of notional
    interpolated between (barrier-epsilon,0) and (barrier+epsilon,1)
  - And continue with the remaining notional
  - Hence, continuous

- Smoothing and fuzzy logic
  - Like Schrodinger's cat, the transaction is in a superposition of dead and alive states
  - Smoothing is achieved by replacing sharp logic (dead or alive?) by fuzzy logic (how much alive?)
  - More in the presentation
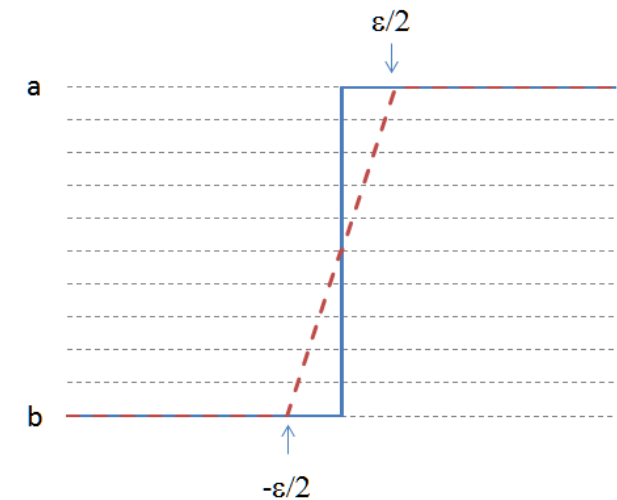
```cpp
template <class T>
inline T toyDupireBarrierMc(
    //  Spot
    const T             S0,
    //  Local volatility
    const vector<T>     spots,
    const vector<T>     times,
    const matrix<T>     vols,
    //  Product parameters
    const T             maturity,
    const T             strike,
    const T             barrier,
    //  Number of paths and time steps
    const int           Np,
    const int           Nt,
    //  Smoothing
    const T             epsilon,
    //  initialized random number generator
    RNG&                random)
```

```cpp
//  Initialize
T result = 0;
// double because the RNG is not templated (and doesn't need to be, see chapter 12)
vector<double> gaussianIncrements(Nt);
const T dt = maturity / Nt, sdt = sqrt(dt);

//  Loop over paths
for (int i = 0; i < Np; ++i)
{
    //  Generate Nt Gaussian Numbers
    random.nextG(gaussianIncrements);
    //  Step by step
    T spot = S0, time = 0;
    /*  bool alive = true; */ T alive = 1.0; // alive is a real number in (0,1)
    for (size_t j = 0; j < Nt; ++j)
    {
        //  Interpolate volatility
        const T vol = interp2D(spots, times, vols, spot, time);
        time += dt;
        //  Simulate return
        spot *= exp(-0.5 * vol * vol * dt + vol * sdt * gaussianIncrements[j]);
        //  Monitor barrier
        /* if (spot > barrier) { alive = false; break; } */
        if (spot > barrier + epsilon) { alive = 0.0; break; }          //   definitely dead
        else if (spot < barrier - epsilon) { /* do nothing */ };       //   definitely alive
        else /* in between, interpolate */ alive *= 1.0 - (spot - barrier + epsilon) / (2 * epsilon);

    }
    //  Payoff paid on surviving notional
    /* if (alive && spot > strike) result += spot - strike; */ if (spot > strike) result += alive * (spot - strike);
}   //  paths
return result / Np;
```

Simulation code

- The toy simulation code is found on the file ToyCode.h in the repo

- To be compared with professional code in the files with names prefixed by "mc"

- Our simple code returns the same result as the professional code

- It is twice slower than the serial version of the professional code

- On a quad-code computer, it is 8x slower than the parallel version

- Next, we differentiate it with our simple AAD framework

Differentiation

Just like we did for Black & Scholes, to compute differentials, we:

1. Initialize the inputs as Numbers
   Which also records them on tape

2. Call our templated evaluation code, instantiated with the Number type
   Which performs the evaluation *and* records operations on tape

3. Propagate adjoints backwards through the tape

4. Pick differentials as the adjoints of the parameters

```cpp
void toyDupireBarrierMcRisks(
const double S0, const vector<double> spots, const vector<double> times, const matrix<double> vols,
const double maturity, const double strike, const double barrier,
const int Np, const int Nt, const double epsilon, RNG& random,
/* results: value and dV/dS, dV/d(local vols) */ double& price, double& delta, matrix<double>& vegas)
{

//  1. Initialize inputs

Number nS0(S0), nMaturity(maturity), nStrike(strike), nBarrier(barrier), nEpsilon(epsilon);
vector<Number> nSpots(spots.size()), nTimes(times.size());
matrix<Number> nVols(vols.rows(), vols.cols());

for (int i = 0; i < spots.size(); ++i) nSpots[i] = Number(spots[i]);
for (int i = 0; i < times.size(); ++i) nTimes[i] = Number(times[i]);
for (int i = 0; i < vols.rows(); ++i) for (int j = 0; j < vols.cols(); ++j) nVols[i][j] = Number(vols[i][j]);

//  2. Call instrumented evaluation code, which evaluates the barrier option price and records all operations
Number nPrice = toyDupireBarrierMc(nS0, nSpots, nTimes, nVols, nMaturity, nStrike, nBarrier, Np, Nt, nEpsilon, random);

//  3. Adjoint propagation, the exact same code as before, should be encapsulated in a dedicated function
vector<double> adjoints = calculateAdjoints(nPrice);

//  4. Pick results
price = nPrice.value;
delta = adjoints[nS0.idx];
for (int i = 0; i < vols.rows(); ++i) for (int j = 0; j < vols.cols(); ++j) vegas[i][j] = adjoints[nVols[i][j].idx];
}
```
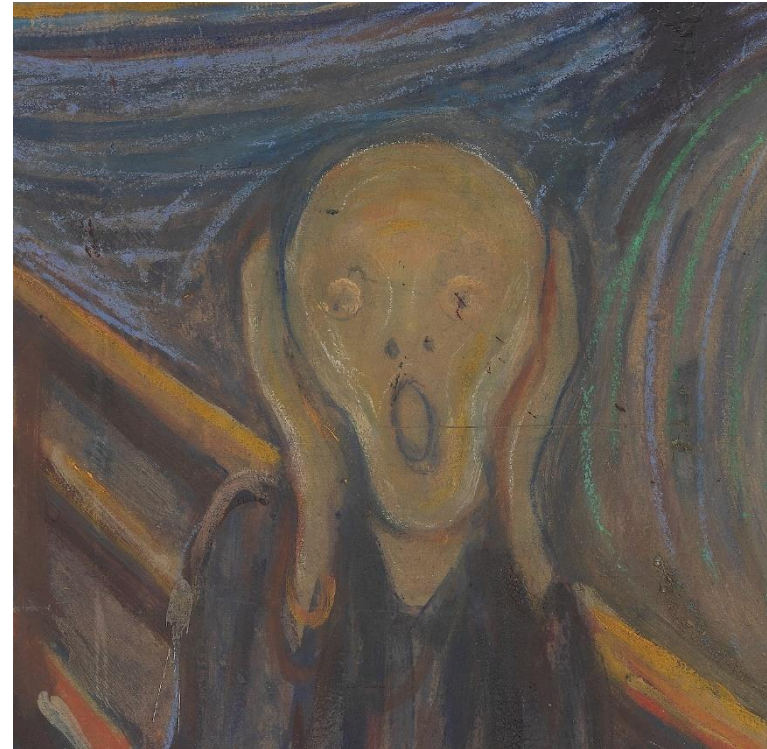
Testing the code

- We run the code in the same context as the initial demonstration but with 100,000 paths instead of 500,000

- The computer runs out of memory and crashes!

- Running AAD on a simulation with 100,000 paths consumes an insane amount of RAM

- Even on a computer with enough memory, such large tape is cache inefficient

Solution in principle

- Run a series of risks on mini-batches of say, 1024 paths and average in the end

- Wipe the tape in between mini-batches

- The average of differentials is the differential of the average

- So we get the same results while reducing memory footprint to operations recorded over 1,024 paths

- Note with mini-batches of size 1, this is known as "path-wise differentiation"

- This is also a particular, and simple case of the general check-pointing algorithm explained in chapter 13

Solution in code

- Rename our function DupireRisksMiniBatch(), call it sequentially from a wrapper function

```cpp
void toyDupireBarrierMcRisks(
    const double S0, const vector<double> spots, const vector<double> times, const matrix<double> vols,
    const double maturity, const double strike, const double barrier,
    const int Np, const int Nt, const double epsilon, RNG& random,
    /* results: value and dV/dS, dV/d(local vols) */ double& price, double& delta, matrix<double>& vegas)
{
    price = delta = 0;
    for (int i = 0; i < vegas.rows(); ++i) for (int j = 0; j < vegas.cols(); ++j) vegas[i][j] = 0;
    double batchPrice, batchDelta; matrix<double> batchVegas(vegas.rows(), vegas.cols());
    int pathsToGo = Np, pathsPerBatch = 1024;
    //  calculate btach sensitivities sequentially
    while (pathsToGo > 0)
    {
        //  wipe tape
        tape.clear();

        //  do mini batch
        int paths = min(pathsToGo, pathsPerBatch);
        dupireRisksMiniBatch(S0, spots, times, vols, maturity, strike, barrier, paths, Nt, epsilon, random, batchPrice, batchDelta, batchVegas);

        //  update results
        price += batchPrice * paths / Np;
        delta += batchDelta * paths / Np;
        for (int i = 0; i < vegas.rows(); ++i) for (int j = 0; j < vegas.cols(); ++j) vegas[i][j] += batchVegas[i][j] * paths / Np;

        pathsToGo -= paths;
    }
}
```
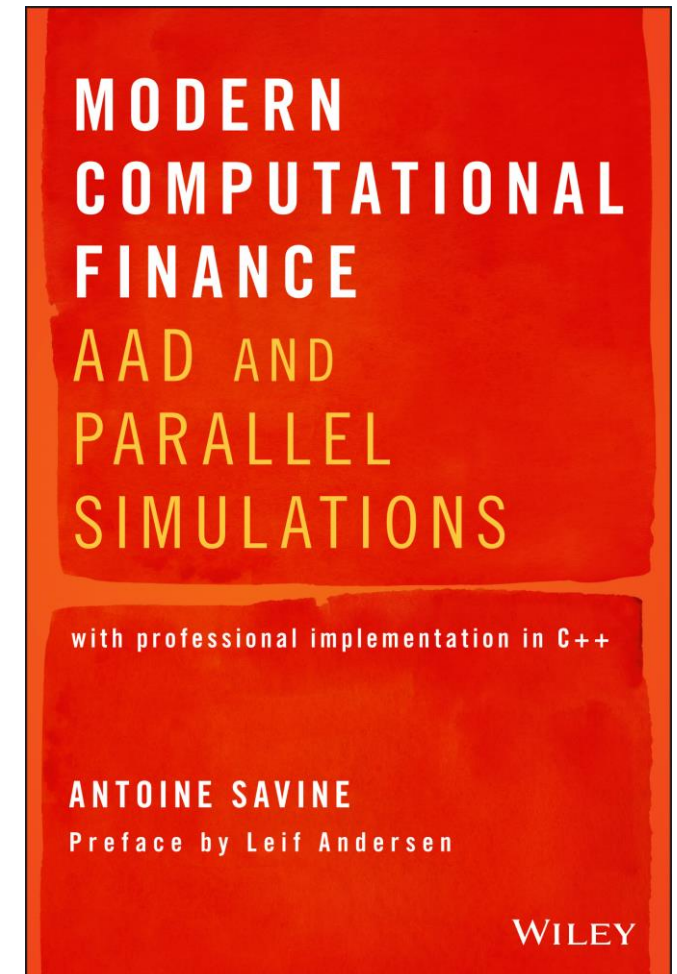
Performance

- With 100,000 paths, 156 steps, we compute the 1,081 differentials in around 7 seconds

- This is 1,081 differentials in the time of around 6 evaluations

- This is a very remarkable result, especially with such simplistic code

- Try it yourself with the toy code in the repo!

- This being said, the professional code is around 8 times faster in serial mode,
  32 times faster in parallel mode on a quad-core laptop

Conclusion

- We learned AAD in principle and in code
  And applied it to machine learning and finance

- But we really just scratched the surface

- For example:
  - How to efficiently differentiate the multiple results of non-scalar functions?
  - How to compute risks not on model parameters like local volatilities
    But on tradable market variables like implied volatilities?
  - How to implement AAD in modern C++ and manage memory efficiently?
  - How to implement AAD over parallel simulations and run it at least 32x faster?

- The answers, and much more, are in the book

MODERN
COMPUTATIONAL
FINANCE
AAD AND
PARALLEL
SIMULATIONS

with professional implementation in C++

ANTOINE SAVINE
Preface by Leif Andersen

WILEY

Thank you for your attention. Find the slides here:

go to http://github.com/asavine

click on the repo 'CompFinance'

download slides 'Intro2AADinMachineLearningAndFinance.pdf'
download examples in the 'Workshop' folder
see tutorial in the 'xlCpp' folder for exporting C++ code to Excel
see basic AAD code in C++ in the file 'toyCode.h'
the rest of the repo is companion code for the book Modern Computational Finance

follow asavine on GitHub and watch the CompFinance repo to be notified of updates