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

Antoine Savine

http://antoinesavine.com

http://github.com/asavine/CompFinance/Intro2AADinMachineLearningAndFinance.pdf

http://tinyurl.com/savineAadTalk

follow asavine on GitHub and watch the 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

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

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

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

- 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

Application: model fitting

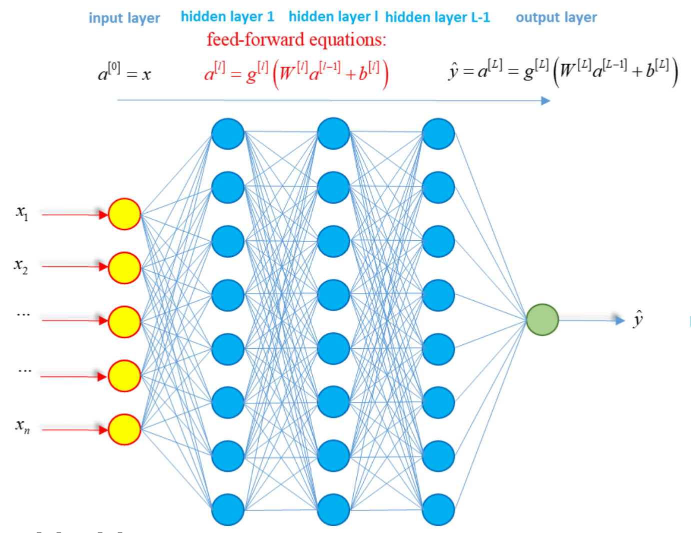
- Model: $\hat{y} = f\left(x;\vartheta\right)$
  - Makes a prediction $\hat{y}$ from a set of inputs $x = \left(x_1,\ldots,x_n\right)^T$
  - Given parameters $\vartheta = \left(\vartheta_1,\ldots,\vartheta_D\right)^T$

- Learns its parameters by minimizing a *cost* function
  - E.g. sum of squared errors: $C\left(\vartheta\right) = \sum_{i=1}^{m}\left[f\left(x^{(i)};\vartheta\right) - y^{(i)}\right]^2 = \sum_{i=1}^{m} c_i\left(\vartheta\right)$ (c_i : error or *loss* on training example *i*)
  - Over a training set of *m* labelled examples: $\left(x^{(i)},y^{(i)}\right)_{1\le i\le m}$ ($x^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R}$)

- Must compute gradient of the cost function for optimization algorithm: $\frac{\partial C}{\partial \vartheta} = \left(\frac{\partial C}{\partial \vartheta_1},\ldots,\frac{\partial C}{\partial \vartheta_D}\right)$
  - Standard case (sum of squared errors): $\frac{\partial C}{\partial \vartheta} = \sum_{i=1}^{m}\frac{\partial c_i}{\partial \vartheta}$, $\frac{\partial c_i}{\partial \vartheta} = 2c_i\frac{\partial f\left(x_i;\vartheta\right)}{\partial \vartheta}$

Training a deep learning model

- Multi-Layer Perceptron (MLP)
  simplest and most common deep learning model:
  - Prediction: feed-forward equations $a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$
  - 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 $W^{[1]}, b^{[1]}, \ldots, W^{[l]}, b^{[l]}, \ldots, W^{[L]}, b^{[L]}$
    to minimise cost = sum of errors (losses) on the training set $c^{(i)} = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$
  - Then apply trained model to make predictions on new examples with unknown label



$W^{[1]}, b^{[1]}, \ldots, W^{[l]}, b^{[l]}, \ldots, W^{[L]}, b^{[L]}$

parameters: $W^{[l]} : matrix\left(n_l, n_{l-1}\right)$ $\quad$ $b^{[l]} : vector\left(n_l\right)$

$n_0 = n, n_L = 1$

- Requires an iterative minimization algorithm
  - On each iteration, we must provide *C*: *m* predictions through a deep network
  - And all their derivatives to parameters $\dfrac{\partial c^{(i)}}{\partial W_{j,k}^{[l]}}, \dfrac{\partial c^{(i)}}{\partial b_j^{[l]}}$

Calibrating a financial pricing model

- Calibration of a pricing model to market data:
    - Prediction: valuation of a financial product $\hat{y} = \text{product value} = f\left(\text{product parameters } x; \text{model parameters } \vartheta\right)$
    - Training set: *m* products, each with *n* known features $x^{(i)}$ and market price $y^{(i)}$
    - Calibration: fit parameters $\vartheta = \left(\vartheta_1, \ldots, \vartheta_D\right)^T$ (e.g. volatilities) to minimise cost = sum of errors (losses) on the training set
    - Then apply calibrated model to price other products which prices are unknown
- Minimization algorithms (gradient descent, conjugate gradients, pseudo-Newton, Levenberg-Marquardt…)
    - Requires derivatives $\frac{\partial C}{\partial \vartheta} = \left(\frac{\partial C}{\partial \vartheta_1}, \ldots, \frac{\partial C}{\partial \vartheta_D}\right)$ on each iteration, we have as many differentials as we have parameters
    - in addition *C* may be expensive to evaluate, must value all *n* products in the calibration set

Notes on calibration

- 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

- 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 (e.g. correlation) typically estimated
  - Often calibrated on European options to value more complicated *exotic* options
  - “Extrapolation” problem

Application: model risk

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

- Model parameters:
  - Todays underlying asset prices
  - Volatilities, correlations
  - Mean-reversions
  - Etc.

- Model risks: $\frac{\partial f}{\partial \vartheta}$ with potentially massive number of parameters
  - Black-Scholes: spot, volatility, rate
  - 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 (+ mean-reversion…)
  - Multi-currency models (Baskets, CVA…): all of the above for all currencies + forex rates and their volatilities + giant correlation matrix
  - Typically in the thousands or tens of thousands
  - Valuation generally expensive: Monte-Carlo simulations

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 $V = g(b) = g\left[c(a)\right] = f(a), f : \mathbb{R}^k \to \mathbb{R}$

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

- 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 $\frac{\partial f}{\partial a} = \frac{\partial g}{\partial b}\frac{\partial c}{\partial a}$

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

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

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 financial valuation
  - We must compute its sensitivities to all its inputs $\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_D} \right)$

- 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 $\frac{\partial f}{\partial x_i} \approx \frac{f(x + \varepsilon e_i) - f(x)}{\varepsilon}, e_i = (\delta_{ij})_{1 \le j \le n}$
  - 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*: $\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_D} \right)$
  - 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 on the 30 greatest numerical algorithms of the 20th century (Trefethen, 2015)

- Did not take firm hold in the computer science community until 1980
- 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 framework in machine learning
  - Written in C++/CUDA (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: challenges

- However it is challenging to develop or even use AAD frameworks efficiently:
  - AD is somewhat “mind twisting” and generally poorly understood
  - Memory management: AAD consumes vast amounts of RAM
  - RAM, cache and parallel efficiency are hard to achieve
  - Aspects of practical implementation are topics of advanced research

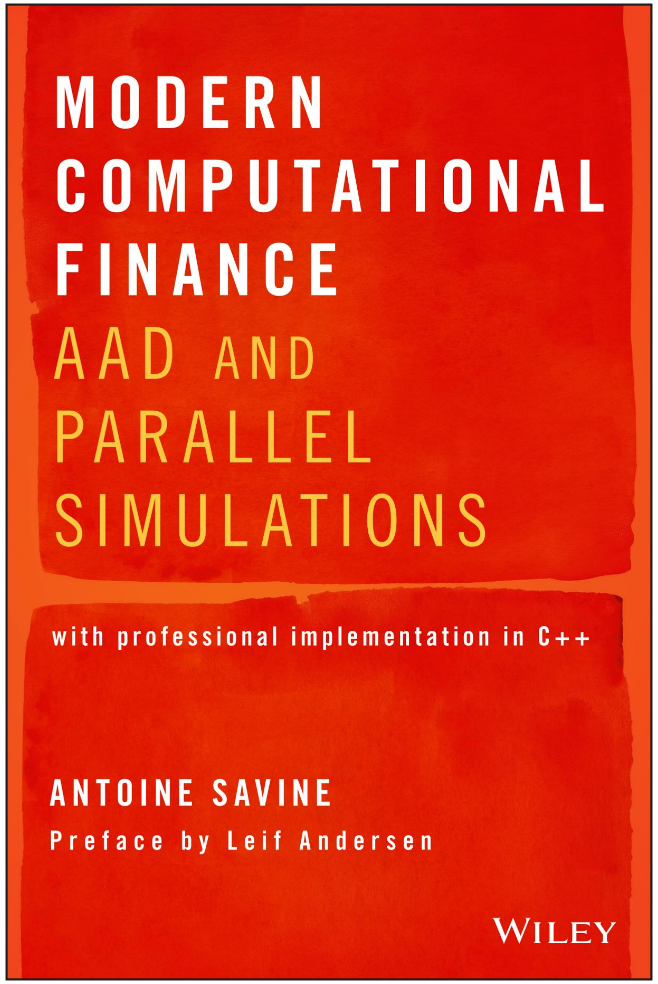
- Blindly applying a framework almost never works
  - Not quite as easy as coding an evaluation and sending it to the framework for differentiation
  - Requires a deep understanding of the theory and implementation
  - Otherwise often results in slow differentiation, incorrect results, crashes, etc.

- This presentation covers the bases of AD and AAD and introduces the book:

Modern Computational Finance: AAD and Parallel Simulations

which explains in deep detail all aspects of AAD and its practical implementation

AAD book



MODERN COMPUTATIONAL FINANCE

AAD AND PARALLEL SIMULATIONS

with professional implementation in C++

ANTOINE SAVINE

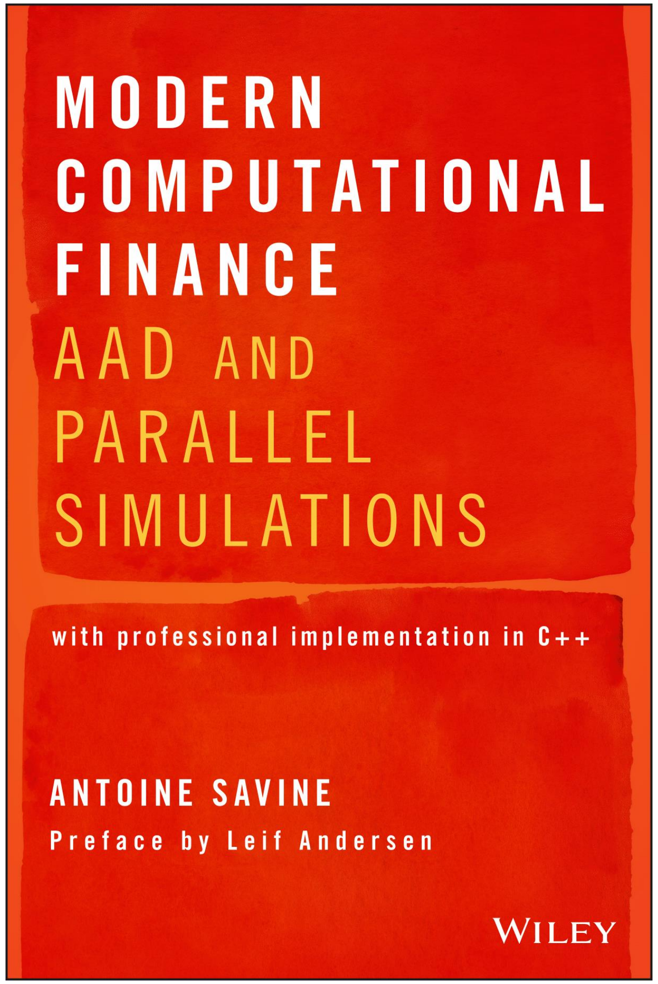
Preface by Leif Andersen

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


MODERN COMPUTATIONAL FINANCE
AAD AND PARALLEL SIMULATIONS
with professional implementation in C++
ANTOINE SAVINE
Preface by Leif Andersen
WILEY

From Amazon: https://www.amazon.com/gp/product/1119539455

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

Demonstration: Dupire's model (1992)

- Extended Black & Scholes dynamics (in the absence of rates, dividends etc.): $\frac{dS}{S} = \sigma(S,t)\,dW$
- Calibrated with Dupire's celebrated formula: $\sigma(K,T) = \frac{2\frac{\partial C}{\partial T}}{\frac{\partial^2 C}{\partial K^2}}$ 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: 30 spots (every 5 points 50 to 150) and 36 times (every month from 0 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*: $v(S_t,t) = E\left[(S_T - K)^+ 1_{\{\max(S_{T_1},\dots,S_{T_K})<B\}} \middle| S_t\right]$
- 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 on http://www.github.com/asavine/CompFinance/wiki)
  - Generic library design with efficient implementation (Chapter 6)
  - Parallel implementation (Chapters 3 and 7)
  - Sobol quasi-random numbers (Chapters 5 and 6)
  - Advanced AAD with expression templates (Chapter 15)

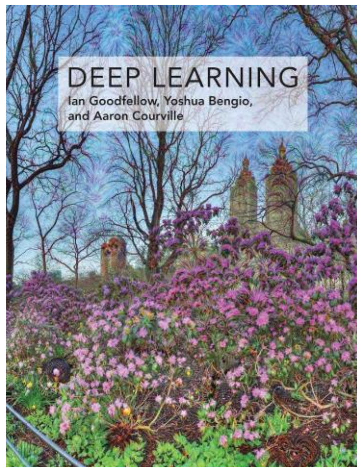
- Hardware: quad-code laptop (surface book 2, 2017)

- 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

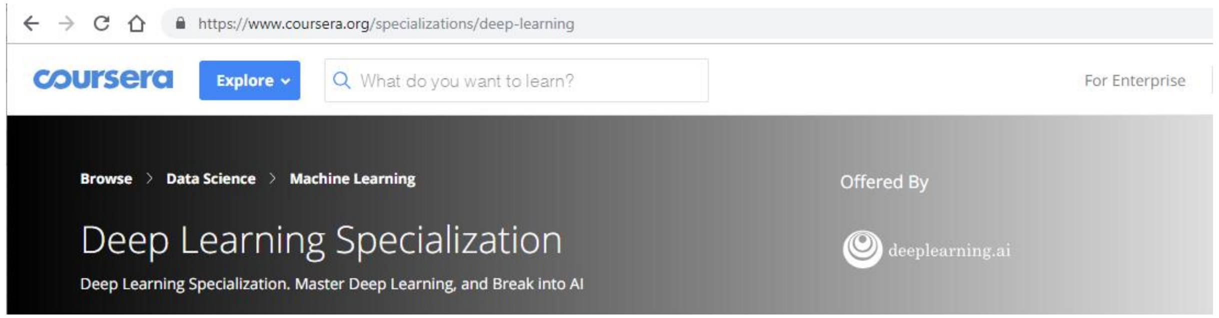
Deep Learning

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 presentations are found in:



Goodfellow's book



Deep Learning Specialization

Deep Learning Specialization. Master Deep Learning, and Break into AI

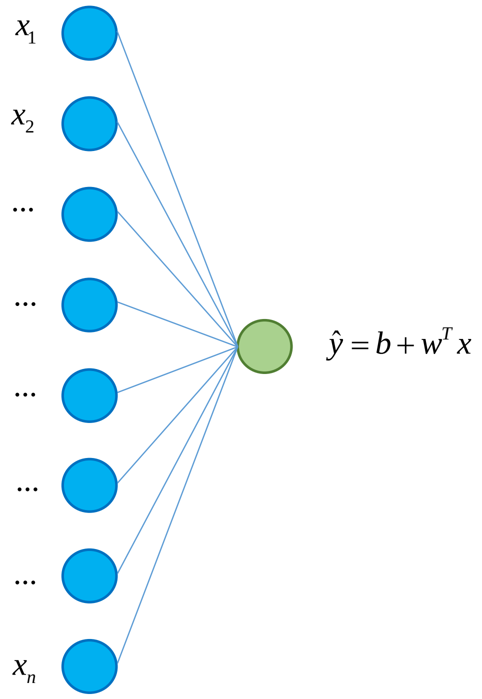
Andrew Ng's videos on Coursera

…and many other books and resources

Linear regression: prediction

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

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

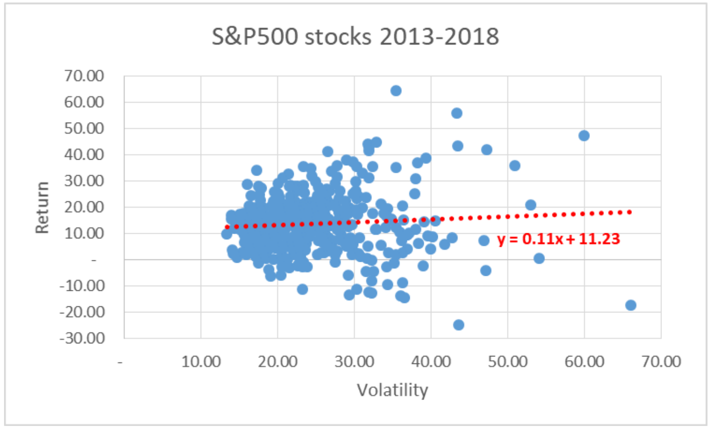


Linear regression and classification (1)

- 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

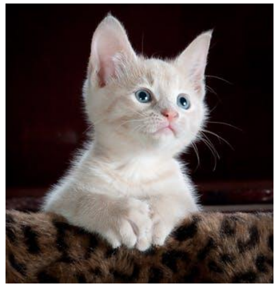


Linear regression and classification (2)

- 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

$$\in \left[0, 2^{24} - 1\right]$$

$$x = \begin{pmatrix} x_1 \\ \dots \\ x_{2,073,600} \end{pmatrix} \xrightarrow[\text{softmax regression}]{} \hat{y} = s\left(b + wx\right) = \begin{bmatrix} \Pr\left(\text{no animal}\right) \\ \Pr\left(\text{cat}\right) \\ \Pr\left(\text{dog}\right) \\ \Pr\left(\text{bird}\right) \\ \Pr\left(\text{other animal}\right) \end{bmatrix}$$

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

s: softmax function

$s : \mathbb{R}^5 \rightarrow \left(0,1\right)^5$ summing to 1

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

- In this presentation, we stick with regression
- Everything that follows generalises easily to classification and the equations remain essentially identical
- See literature for details, for example Stanford's CS229 on http://cs229.stanford.edu/syllabus.html

Linear regression: training

- 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

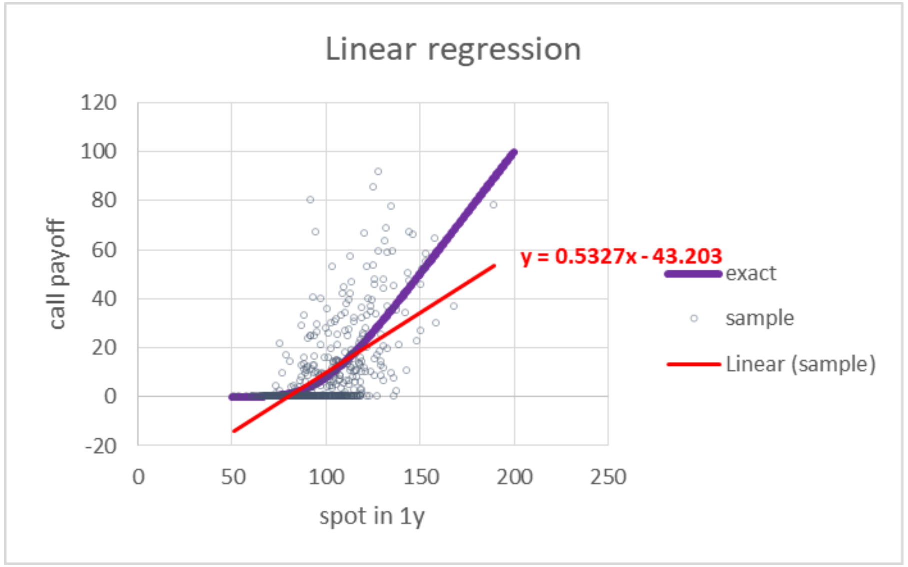
- *b** and *w** are found analytically, solving for $\frac{\partial C}{\partial b} = 0$ and $\frac{\partial C}{\partial w} = 0$

- 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 $\mathrm{Y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(m)} \end{pmatrix}$

Lin reg only captures linear functions

- Example:
  - Predict the future price in 1y of a European call strike 100, maturity 2y
  - By regression of the payoff in 2y: $y^{(i)} = \left(S_{2y}{}^{(i)} - K \right)^{+}$
  - Over the underlying asset price in 1y: $x^{(i)} = S_{1y}{}^{(i)}$
  - With a training set of *m* paths $\left(S_{1y}{}^{(i)}, S_{2y}{}^{(i)} \right)_{1 \le i \le m}$

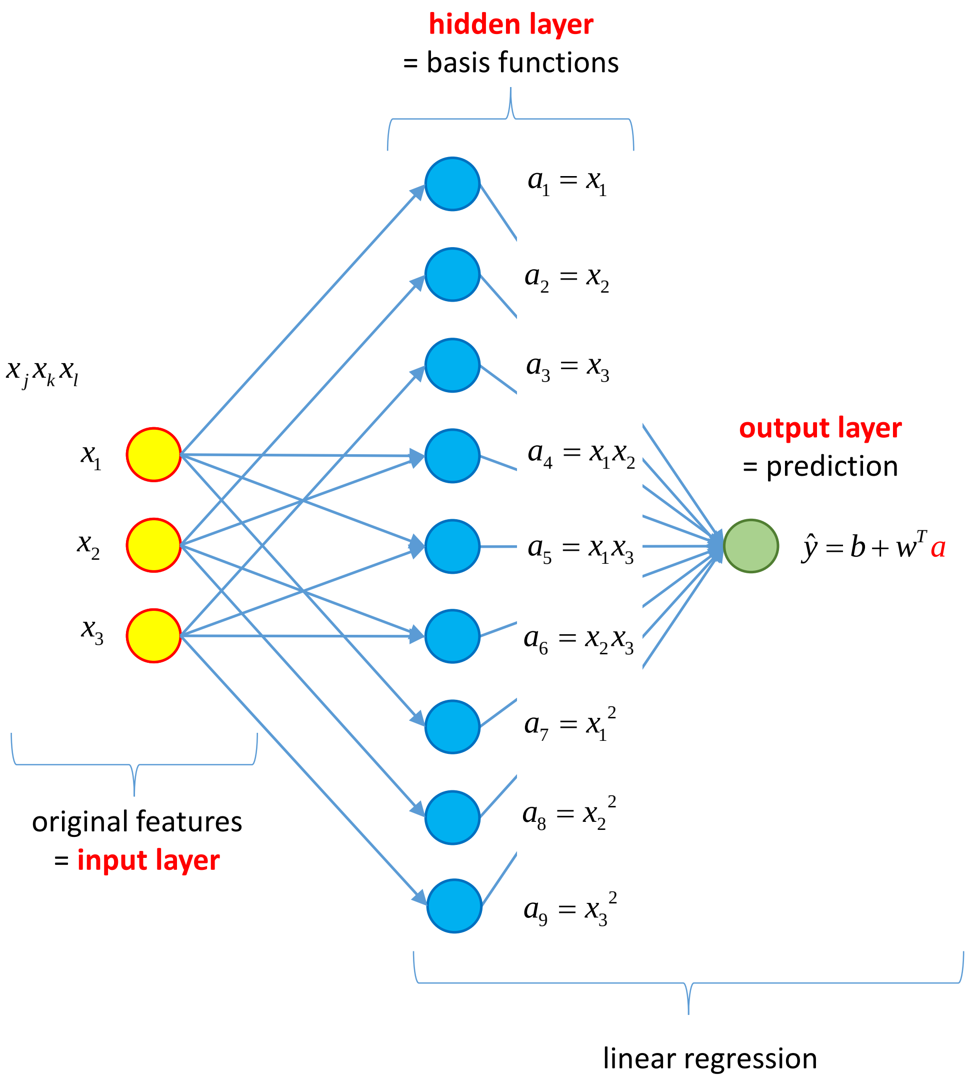
    generated under Black & Scholes' model (spot = 100, volatility of 20%)
  - We know the exact solution is given by Black and Scholes' formula so we can assess the quality of the regression
  - Linear regression obviously fails to approximate the correct function because it cannot capture non-linearities



Basis function regression

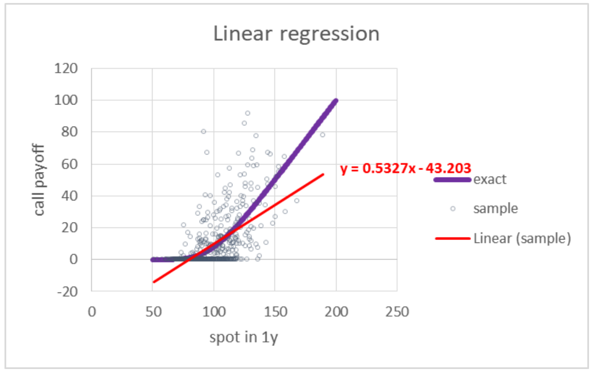
- Solution: regress not on *x* but on basis functions *a* of *x*
- Example: polynomial regression
  - Basis functions = monomials (*x* is in dimension n_0):
    - 1st degree: all the xs $a_i = x_i$
    - 2nd degree: all the squares $a_i = x_j^2$ and pair-wise products $a_i = x_i x_j$
    - 3rd degree: 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*

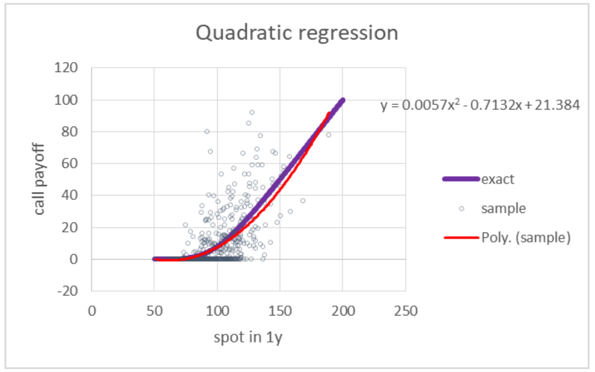
$$\begin{pmatrix} b \\ w_1 \\ w_2 \\ \dots \\ w_{n_1} \end{pmatrix} = \left(A^T A\right)^{-1} A^T Y \quad \text{where} \quad A = \begin{pmatrix} 1 & a_1^{(1)} & \dots & a_{n_1}^{(1)} \\ 1 & a_1^{(2)} & \dots & a_{n_1}^{(2)} \\ \dots & a_1^{(i)} & a_j^{(i)} & a_{n_1}^{(i)} \\ 1 & a_1^{(m)} & \dots & a_{n_1}^{(m)}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



Basis function regression: performance

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

- Quadratic regression: $n_1 = \frac{(n_0+2)(n_0+1)}{2} - 1$ dimension grows e.g. from 10 to 65, from 100 to 5,151, from 1,000 to 501,500
- Cubic regression: $n_1 = \frac{(n_0+3)(n_0+2)(n_0+1)}{6} - 1$ 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)

Overfitting and the rule of ten

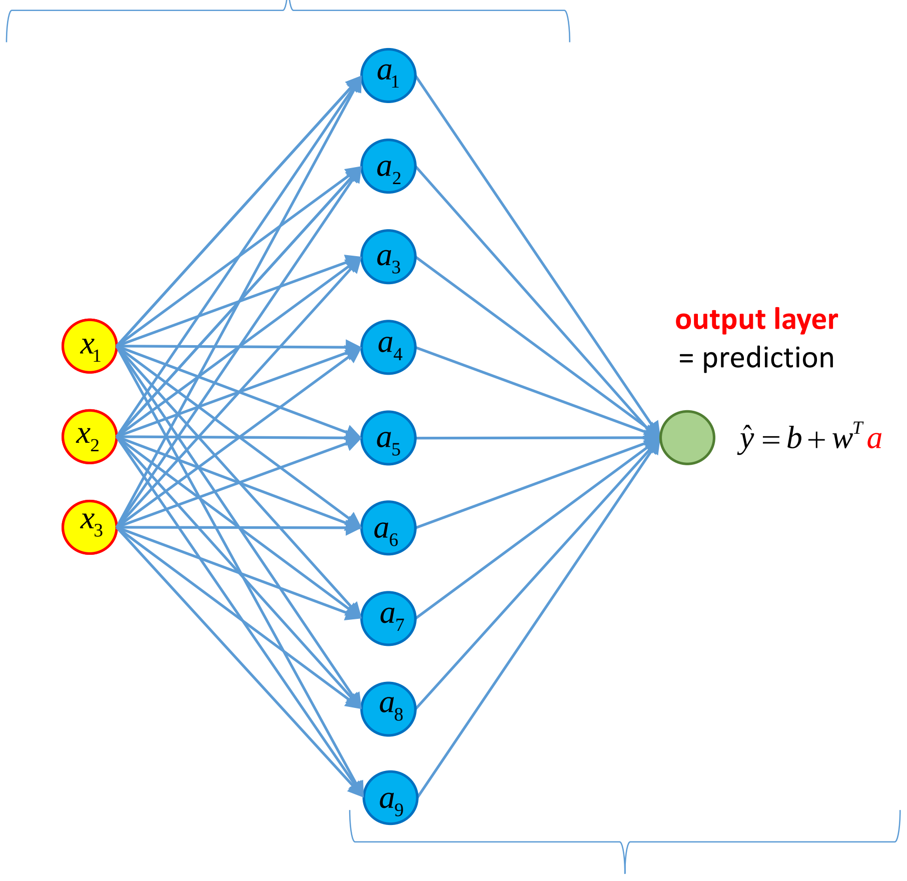
- If you fit a linear model with *n* free parameters to *n* data points
  - You will find a perfect fit
  - But the model may not generalize well to new examples
  - Because it captured the noise of the particular training set, “overfitting”
- For error analysis (“variance-bias trade-off”), see:

  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

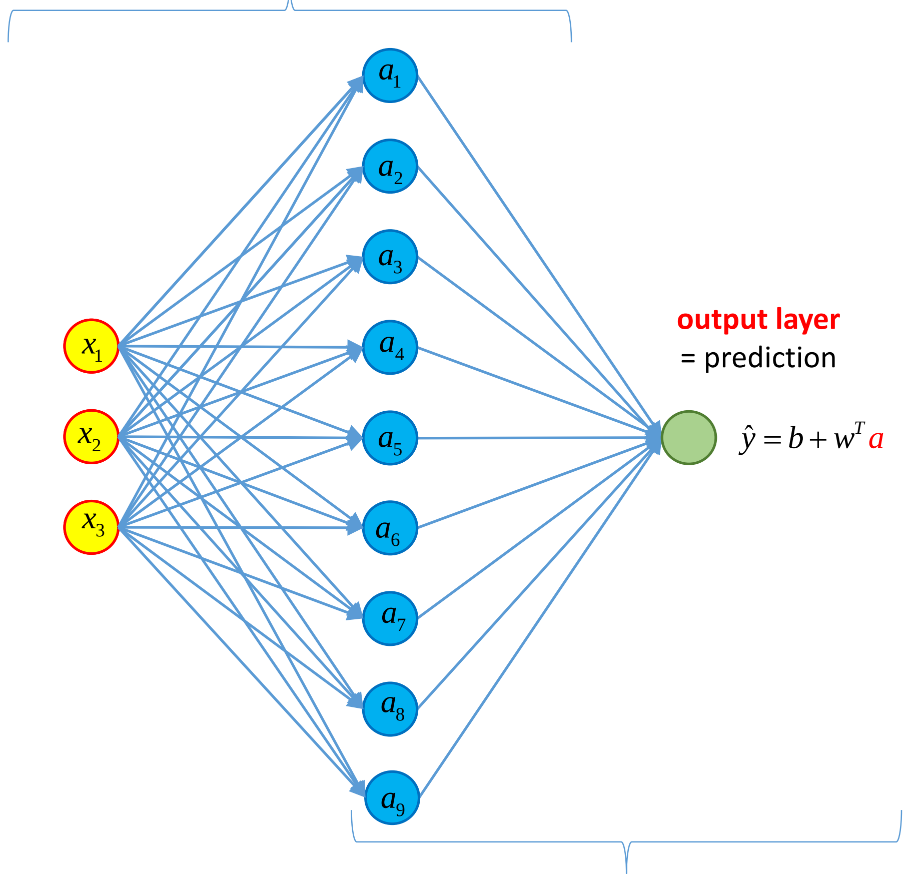
Idea (ANN): learn basis functions from data

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



linear regression over basis functions

ANN
hidden layer = **learned** set of basis functions
part of the model, subject to trainable parameters



linear regression over basis functions
identical

Shallow ANN: parametric basis functions

- 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 the *same function of x*, with *different parameters theta*
  - 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 xs (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, $W = \begin{pmatrix} w_1^T \\ \ldots \\ w_{n_1}^T \end{pmatrix}$ is a $n_1 \times n_0$ matrix and $b = \begin{pmatrix} b_1 \\ \ldots \\ b_{n_1} \end{pmatrix}$ is a n_1 dimensional vector
  - Then the vector of basis functions is $a = g(Wx + b)$ where *g* is a scalar function applied element-wise to the n_1 vector $z = Wx + b$

ANN: prediction

- Prediction in two steps

  - Step 1: hidden layer (basis functions) **inputs *x* → basis functions *a*** $a = g\left(W^{[1]}x + b^{[1]}\right)$
    - 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 $z^{[1]} = W^{[1]}x + b^{[1]}$
    - *g* must be non-linear or we are back to linear regression
      (linear regression on linear functions is identical to linear regression on inputs)

  - Step 2: regression **basis functions *a* → prediction** $\hat{y} = w^{[2]}a + b^{[2]}$
    - Parameters: $w^{[2]}$ column vector in dimension n_1 and $b^{[2]}$ real number
    - $\hat{y}$ is a real number

- Unified notation for both steps = feed-forward equation $a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$

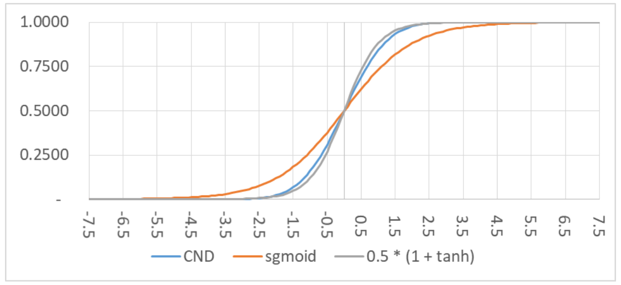
  - With the notations

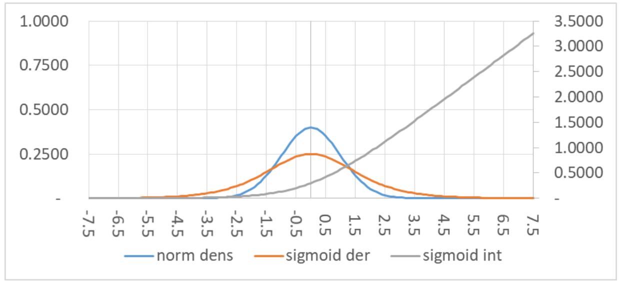
$a^{[0]} = x$	input layer	dim n_0
$a^{[1]} = a$	hidden layer	dim n_1
$\hat{y} = a^{[2]}$	output layer	dim $n_2 = 1$
$g^{[1]} = g, g^{[2]} = id$	hidden and output activations	

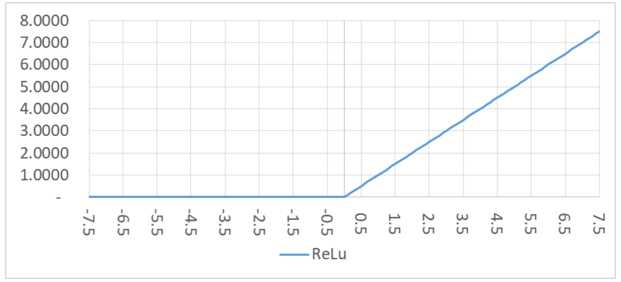
    and parameters $W^{[l]}$ of dim $n_l \times n_{l-1}$ and $\mathrm{b}^{[l]}$ of dim n_l for $l = 1, 2$

Choice of activation function

- The hidden activation function $g^{[1]}$ must be non-linear – the most common choices are:
  - "Cheap Gaussians"
    $$\text{sigmoid} \quad \sigma(x) = \frac{1}{1+\exp(-x)}$$
    $$\text{tanh} \quad t(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$
  - Derivatives and integrals of sigmoid
    $$\sigma'(x) = \sigma(x)\left(1-\sigma(x)\right)$$
    $$\int_{-\infty}^{x} \sigma(x) = \log\left(1+\exp(x)\right) \ (\text{"softPlus"})$$
  - **Re**ctified **L**inear **U**nits (ReLu) $r(x) = (x)^{+}$
- The output activation function $g^{[2]}$ is:
  - Identity $g^{[2]}(x) = x$ for regression
  - SoftMax for classification





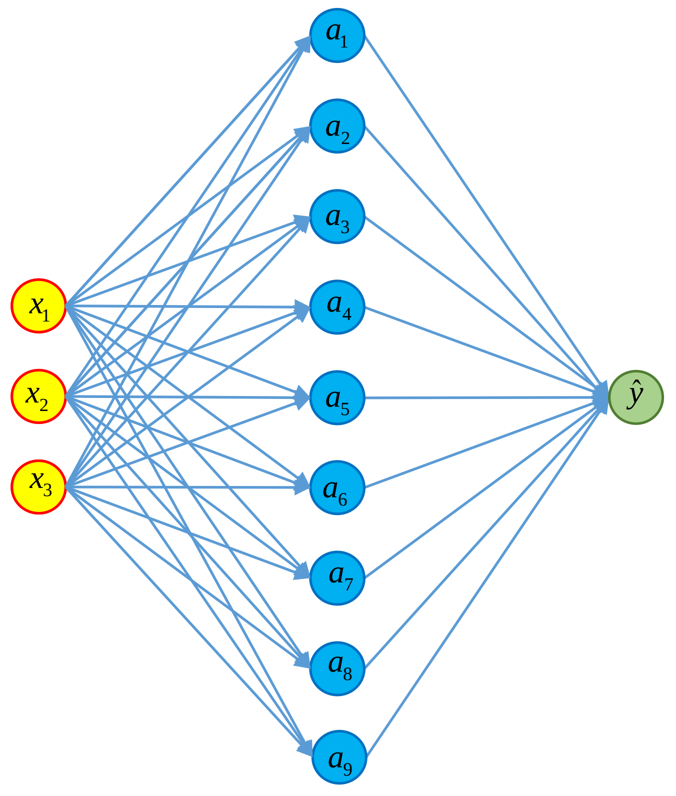


ANN: computation graph

Input layer l=0 | Hidden layer l=1 | Output layer l=2

$$a^{[0]} = x \longrightarrow a^{[1]} = g^{[1]}\left(W^{[1]}a^{[0]} + b^{[1]}\right) \longrightarrow a^{[2]} = g^{[2]}\left(W^{[2]}a^{[1]} + b^{[2]}\right) = \hat{y}$$

n_0 units | n_1 units | 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 $\sim Ln^2$

Examples:

- With 1,000 features/basis functions and 2 layers complexity ~ 2M, fraction of a second on modern CPU
- With 1M features/basis functions and 100 layers (not unrealistic in computer vision) complexity = 200 trillion, long time even on best GPU

ANN: training

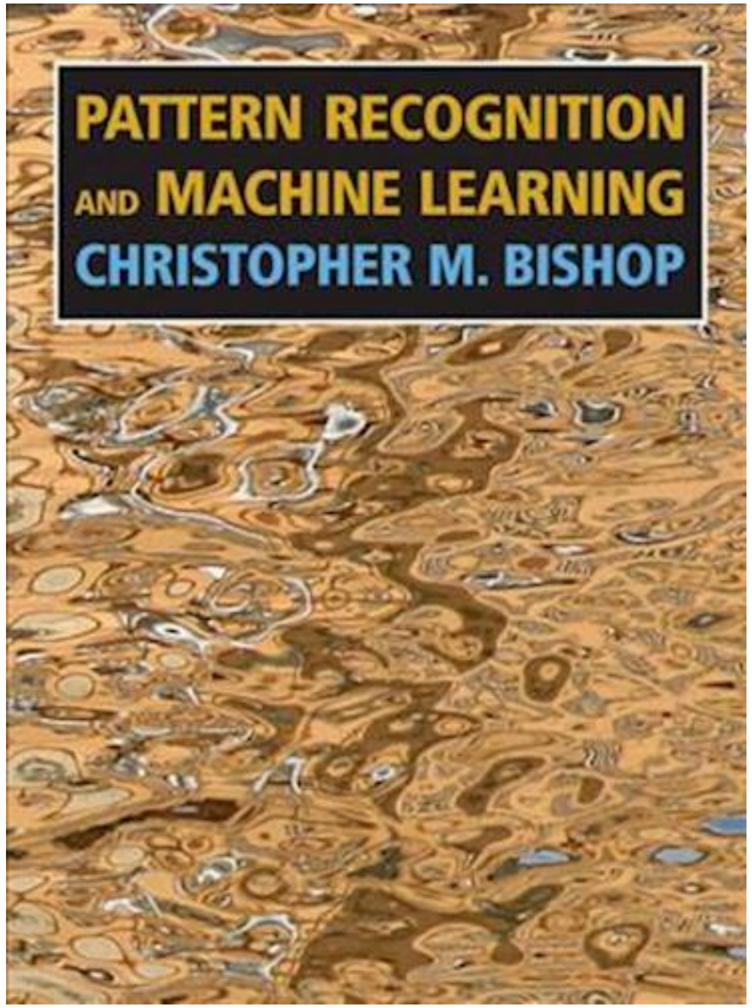
- Training set of *m* examples $x^{(1)},\dots,x^{(m)}$ (each a vector in dimension n_0) with corresponding labels $y^{(1)},\dots,y^{(m)} \in \mathbb{R}$
- Learn parameters: $b^{[1]} \in \mathbb{R}^{n_1}, b^{[2]} \in \mathbb{R}^{n_2=1}$ and $W^{[1]} \in \mathbb{R}^{n_1 \times n_0}, W^{[2]} \in \mathbb{R}^{(n_2=1)\times n_1}$ 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 $c_i\left(b^{[1]},b^{[2]},W^{[1]},W^{[2]}\right) = \left(\underbrace{a^{[2](i)}}_{=\hat{y}^{(i)}} - y^{(i)}\right)^2$
- 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
  - Gradient descent and variants
  - Pseudo-Newton methods based on an estimation of the hessian (second order derivatives) matrix
  - Levenberg and Marquard's algorithm
  - And many others, see **Numerical Recipes**
- Most algorithms (and all efficient ones) 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 $D \sim O\left(n_1 n_0\right)$

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 can capture (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 ten times that much data)
- Similarly, feed-forward ANNs with "perceptron" basis functions offer mathematical guarantees
  - Known as 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, which sigmoidal and variants all pass)
  - See sketch of proof (with ReLu activation in dimension 1):

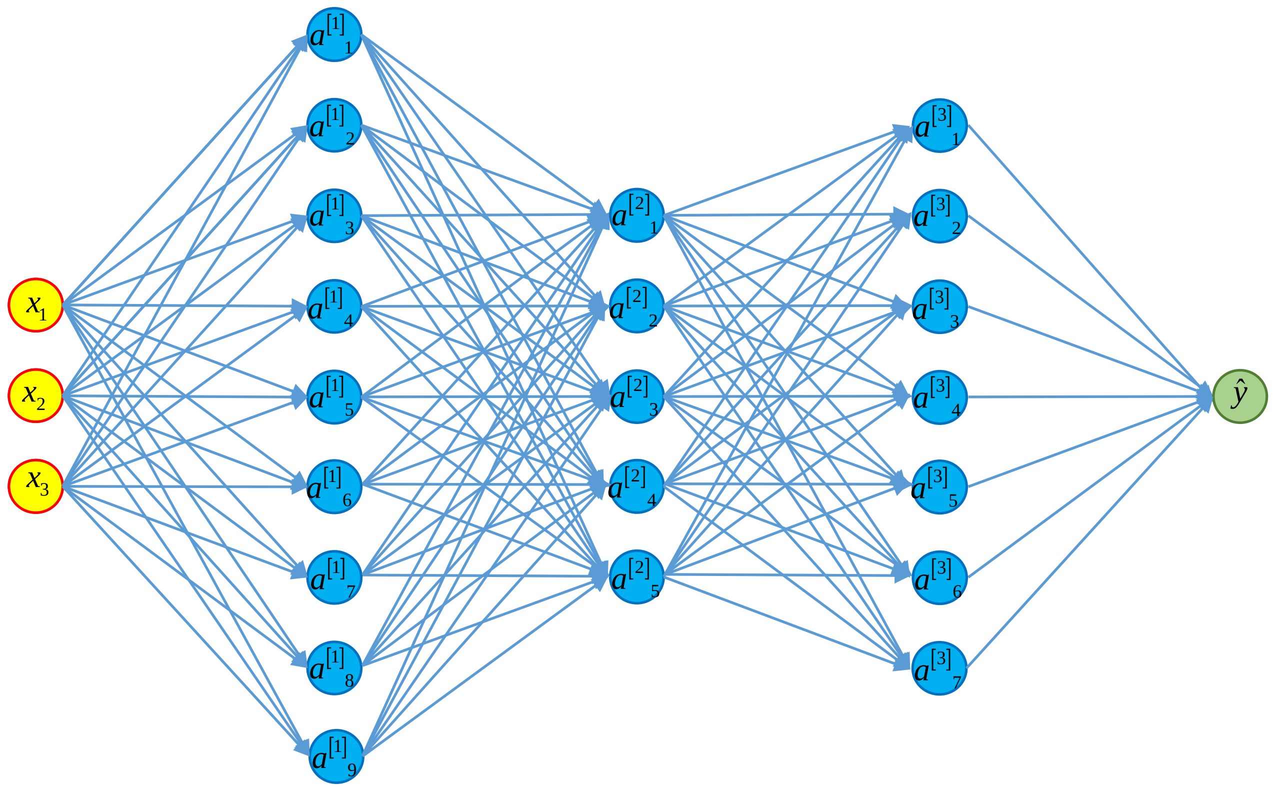
    https://www.quora.com/In-Machine-Learning-ReLU-activation-function-is-said-to-be-a-universal-function-approximation-can-you-show-just-how-it-is-possible-to-approximate-any-continuous-function-by-purely-combining-many-instances-of-ReLU/answer/Antoine-Savine-1
- In conclusion
  - ANNs are superior to basis function regression in what they learn basis functions from the data
  - 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



PATTERN RECOGNITION AND MACHINE LEARNING
CHRISTOPHER M. BISHOP

Deep learning: multiple hidden layers



feed-forward equation

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

Deep learning: composing basis functions

- 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 **basis functions of basis functions**

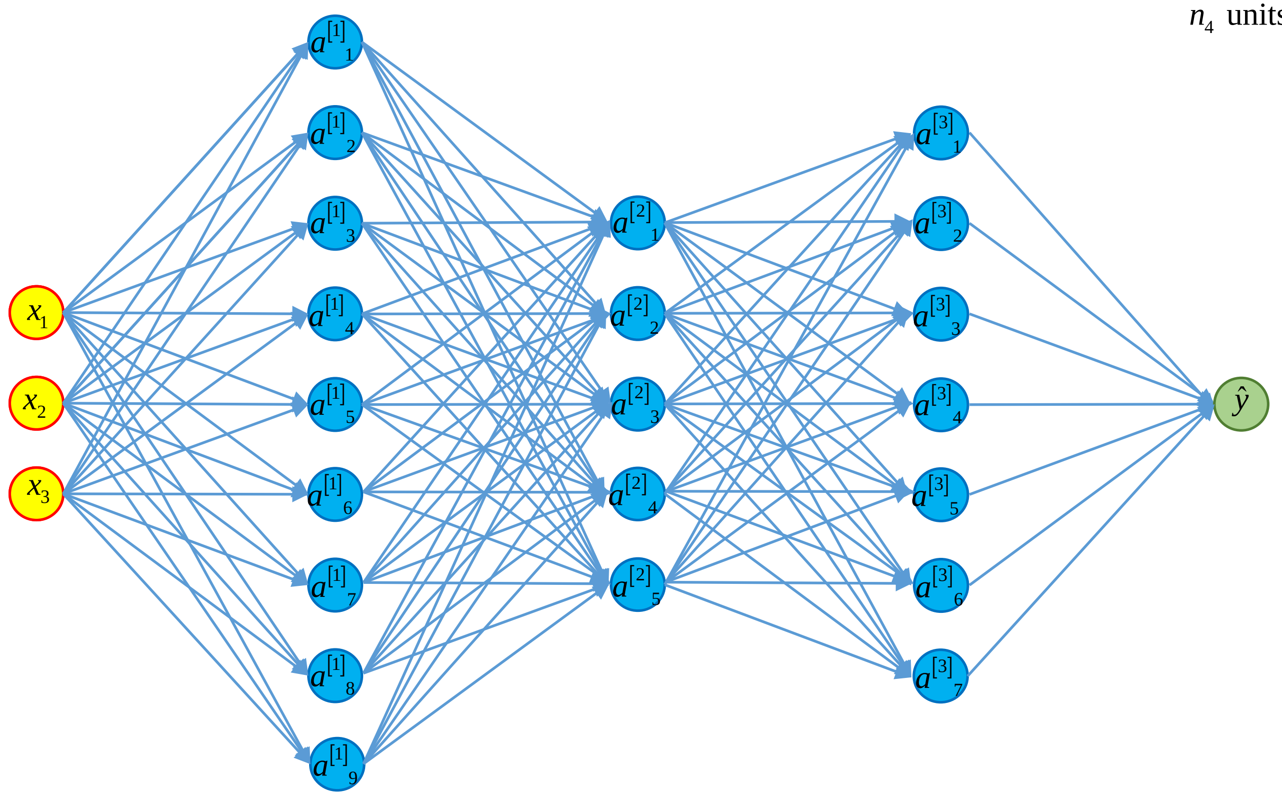
- Deep ANN with L layers (L-1 hidden layers)
  - Prediction is a linear combination of the **last** hidden layer: $\hat{y} = W^{[L]} a^{[L-1]} + b^{[L]}$
  - Last hidden layer contains basis functions of basis functions of basis functions … of basis functions of the inputs from the feed-forward equation: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

Why deep learning?

- 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?
- Composition of basis function offers (exponentially) better function representation abilities
    - With a limited number of units, a single layer ANN only correctly approximates a limited sub-space of functions
    - But if we allow basis functions to compose (and learn composition from data)
      we can represent an exponentially wider variety of functions
    - This result should be intuitive, it was demonstrated in particular cases
      and widely illustrated and empirically validated in a vast number of contexts
    - In short, 10 layers of 10 units each approximate a (much) wider sup-space of functions than 1 layer with 100 units
- This is the basis for deep learning
- Deep ANNs are otherwise a trivial generalization of shallow ANNs

Deep feed-forward networks

Input layer l=0 $a^{[0]} = x$ n_0 units $\rightarrow$ Hidden layer l=1 $a^{[1]} = g^{[1]}\left(W^{[1]}a^{[0]} + b^{[1]}\right)$ n_1 units $\rightarrow$ Hidden layer l=2 $a^{[2]} = g^{[2]}\left(W^{[2]}a^{[1]} + b^{[2]}\right)$ n_2 units $\rightarrow$ Hidden layer l=3 $a^{[3]} = g^{[3]}\left(W^{[3]}a^{[2]} + b^{[3]}\right)$ n_3 units $\rightarrow$ Output layer l=4 $a^{[4]} = g^{[4]}\left(W^{[4]}a^{[3]} + b^{[4]}\right) = \hat{y}$ n_4 units



x_1
x_2
x_3
$a^{[1]}_1$ … $a^{[1]}_9$
$a^{[2]}_1$ … $a^{[2]}_5$
$a^{[3]}_1$ … $a^{[3]}_7$
$\hat{y}$

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

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

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

- Training: find all $\mathrm{W}^{[l]}$ 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

Summary and notations: deep ANN

- Prediction
  - 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 ➔ $\begin{bmatrix} a^{[l]}{}_1 \\ \dots \\ a^{[l]}{}_{n_l} \end{bmatrix} = \varphi^{[l]}\left(\begin{bmatrix} a^{[l-1]}{}_1 \\ \dots \\ a^{[l-1]}{}_{n_{l-1}} \end{bmatrix} ; \begin{bmatrix} \vartheta^{[l]}{}_1 \\ \dots \\ \vartheta^{[l]}{}_{D_l} \end{bmatrix} \right)$
  - Prediction = layer L $\hat{y} = a^{[L]}{}_1$
- Parameters: $\begin{bmatrix} \vartheta^{[1]}{}_1 \\ \dots \\ \vartheta^{[1]}{}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}{}_1 \\ \dots \\ \vartheta^{[2]}{}_{D_2} \end{bmatrix}, \dots, \begin{bmatrix} \vartheta^{[L-1]}{}_1 \\ \dots \\ \vartheta^{[L-1]}{}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}{}_1 \\ \dots \\ \vartheta^{[L]}{}_{D_L} \end{bmatrix}$ so, ultimately: $\hat{y} = f\left(\begin{bmatrix} x_1 \\ \dots \\ x_{n_0} \end{bmatrix} ; \begin{bmatrix} \vartheta^{[1]}{}_1 \\ \dots \\ \vartheta^{[1]}{}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}{}_1 \\ \dots \\ \vartheta^{[2]}{}_{D_2} \end{bmatrix}, \dots, \begin{bmatrix} \vartheta^{[L-1]}{}_1 \\ \dots \\ \vartheta^{[L-1]}{}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}{}_1 \\ \dots \\ \vartheta^{[L]}{}_{D_L} \end{bmatrix} \right)$
- Training: find all the parameters theta
  - To minimize sum of errors on training set with known labels $\min_{\begin{bmatrix} \vartheta^{[1]}{}_1 \\ \dots \\ \vartheta^{[1]}{}_{D_1} \end{bmatrix}, \begin{bmatrix} \vartheta^{[2]}{}_1 \\ \dots \\ \vartheta^{[2]}{}_{D_2} \end{bmatrix}, \dots, \begin{bmatrix} \vartheta^{[L-1]}{}_1 \\ \dots \\ \vartheta^{[L-1]}{}_{D_{L-1}} \end{bmatrix}, \begin{bmatrix} \vartheta^{[L]}{}_1 \\ \dots \\ \vartheta^{[L]}{}_{D_L} \end{bmatrix}} C = \sum_{i=1}^{m} c_i = \left(\hat{y}_i - y_i\right)^2$
  - Takes iterative algorithm where all the differentials must be computed on every iteration: $\frac{\partial c_i}{\partial \vartheta^{[l]}{}_j}$ for all i, l and j

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 $\min_{\begin{bmatrix} W^{[l]}{}_{1,1} & \dots & W^{[l]}{}_{1,n_{l-1}} \\ \dots & \dots & \dots \\ W^{[l]}{}_{n_l,1} & \dots & W^{[l]}{}_{n_l,n_{l-1}} \end{bmatrix}, \begin{bmatrix} b^{[l]}{}_1 \\ \dots \\ b^{[l]}{}_{n_l} \end{bmatrix}, 1 \le l \le L}$ $C = \sum_{i=1}^{m} c_i = \left(\hat{y}_i - y_i\right)^2$
  - Takes iterative algorithm where all the differentials must be computed on every iteration: $\dfrac{\partial c_i}{\partial W^{[l]}{}_{j,k}}, \dfrac{\partial c_i}{\partial b^{[l]}{}_j}$ for all i, l, j and k

Back-Propagation in deep neural nets

Differentials of the cost function

- Mutli-layer perceptrons (MLPs) predict with the feed-forward equations:
  - Recall: $a_0 = x \ , \ a^{[l]} = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right) \ , \ \hat{y} = a^{[L]}$
  - Parameters: $\text{for } 1 \leq l \leq L \ : \ \mathrm{W}^{[l]}\left(n_l \times n_{l-1}\right), b^{[l]}\left(n_l\right) \ , \ D = \sum_{l=1}^{L} n_l\left(1 + n_{l-1}\right) \sim Ln^2$ parameters
  - Cost: $C = \sum_{i=1}^{m} c_i \ , \ c_i = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$
  - Must compute m*D* differentials $\frac{\partial c_i}{\partial W^{[l]}{}_{j,k}}, \frac{\partial c_i}{\partial b^{[l]}{}_j}$ on each iteration
- Feed-forward complexity:
  - For a "reasonably large" network with 10 layers of 100 units, D=100,000
  - To make one prediction: complexity ~ D ~ 100,000
  - We need at least m=10D training examples to avoid overfitting, so to compute one cost, complexity $> 10D^2 \sim 10^{11}$
  - With linear differentials algorithms, we compute the cost D times to calculate D differentials, complexity $> 10D^3 \sim 10^{16}$
  - Not viable, even in parallel on most powerful GPUs

Differentials by finite differences

- Finite difference (FD) algorithm:
  - First compute the cost C_0
  - To compute the sensitivity of the cost to each (scalar) parameter $\vartheta_d, 1 \leq d \leq D$:
    - Bump the parameter by a small amount ε
    - Re-compute the cost C_1 , repeating feed-forward equations for all training examples
    - $\partial C / \partial \vartheta_i \approx \left(C_1 - C_0 \right) / \varepsilon$
- FD is (obviously) linear in *D*, therefore not viable
- But it has very desirable properties

FD automation

- 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 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

Computing cost differentials: general ANN

- First notice: $C = \sum_{i=1}^{m} c_i$ so $\frac{\partial C}{\partial \vartheta^{[l]}} = \sum_{i=1}^{m} \frac{\partial c_i}{\partial \vartheta^{[l]}}$
- Then: $c = (\hat{y} - y)^2 = \left(a^{[L]} - y\right)^2$ (dropping the example index *i* to simplify notations) so $\frac{\partial c}{\partial \hat{y}} = \frac{\partial c}{\partial a^{[L]}} = 2\left(a^{[L]} - y\right) = \text{"2 prediction errors"}$ (**eq1**)
- From the feed-forward equation: $\begin{bmatrix} a^{[l]}{}_1 \\ \ldots \\ a^{[l]}{}_{n_l} \end{bmatrix} = \varphi^{[l]}\left(\begin{bmatrix} a^{[l-1]}{}_1 \\ \ldots \\ a^{[l-1]}{}_{n_{l-1}} \end{bmatrix}; \begin{bmatrix} \vartheta^{[l]}{}_1 \\ \ldots \\ \vartheta^{[l]}{}_{D_l} \end{bmatrix}\right)$ where the functions phi and their differentials are known
  - We get the Jacobian matrix of $a^{[l]}$ to $\text{a}^{[l-1]}$ (**eq2**) ➔ $\frac{\partial a^{[l]}}{\partial a^{[l-1]}} = \frac{\partial \varphi^{[l]}}{\partial a^{[l-1]}} = \begin{bmatrix} \frac{\partial a^{[l]}{}_1}{\partial a^{[l-1]}{}_1} & \ldots & \frac{\partial a^{[l]}{}_1}{\partial a^{[l-1]}{}_{n_{l-1}}} \\ \ldots & \ldots & \ldots \\ \frac{\partial a^{[l]}{}_{n_l}}{\partial a^{[l-1]}{}_1} & \ldots & \frac{\partial a^{[l]}{}_{n_l}}{\partial a^{[l-1]}{}_{n_{l-1}}} \end{bmatrix}$
  - And from the chain rule we have **eq3**: $\frac{\partial c}{\partial \vartheta^{[l]}} = \frac{\partial c}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial \vartheta^{[l]}} = \frac{\partial c}{\partial a^{[l]}} \frac{\partial \varphi^{[l]}}{\partial \vartheta^{[l]}}$

Computing cost differentials: strategy

- Eq1 gives $\frac{\partial c}{\partial a^{[L]}}$ and Eq2 gives $\frac{\partial a^{[l]}}{\partial a^{[l-1]}}$
- So we can compute for every layer l: $\frac{\partial c}{\partial a^{[l]}} = \frac{\partial c}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \cdots \frac{\partial a^{[l+1]}}{\partial a^{[l]}}$ (**back-prop equation**)
- Then, we use Eq3 to get $\frac{\partial c}{\partial \vartheta^{[l]}} = \frac{\partial c}{\partial a^{[l]}} \frac{\partial \varphi^{[l]}}{\partial \vartheta^{[l]}}$
- Back-prop equation for layer 0 (recall the output layer is a real number):

Forward computation, in the order of the layers: 0 to L

$$\longleftarrow$$

$$\frac{\partial c}{\partial a^{[0]}} = \frac{\partial c}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \cdots \frac{\partial a^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial a^{[0]}} = \underbrace{2\left(a^{[L]} - y\right)}_{\frac{\partial c}{\partial a^{[L]}}\text{: real number}} \underbrace{\begin{bmatrix} \frac{\partial a^{[L]}}{\partial {a^{[L-1]}}_{1}} & \cdots & \frac{\partial a^{[L]}}{\partial {a^{[L-1]}}_{n_{L-1}}} \end{bmatrix}}_{\frac{\partial a^{[L]}}{\partial a^{[L-1]}}\text{: vector}} \underbrace{\begin{bmatrix} \frac{\partial {a^{[L-1]}}_{1}}{\partial {a^{[L-2]}}_{1}} & \cdots & \frac{\partial {a^{[L-1]}}_{1}}{\partial {a^{[L-2]}}_{n_{L-2}}} \\ \cdots & \cdots & \cdots \\ \frac{\partial {a^{[L-1]}}_{n_{L-1}}}{\partial {a^{[L-2]}}_{1}} & \cdots & \frac{\partial {a^{[L-1]}}_{n_{L-1}}}{\partial {a^{[L-2]}}_{n_{L-2}}} \end{bmatrix} \cdots \begin{bmatrix} \frac{\partial {a^{[2]}}_{1}}{\partial {a^{[1]}}_{1}} & \cdots & \frac{\partial {a^{[2]}}_{1}}{\partial {a^{[1]}}_{n_1}} \\ \cdots & \cdots & \cdots \\ \frac{\partial {a^{[2]}}_{n_2}}{\partial {a^{[1]}}_{1}} & \cdots & \frac{\partial {a^{[2]}}_{n_2}}{\partial {a^{[1]}}_{n_1}} \end{bmatrix} \begin{bmatrix} \frac{\partial {a^{[1]}}_{1}}{\partial {a^{[0]}}_{1}} & \cdots & \frac{\partial {a^{[1]}}_{1}}{\partial {a^{[0]}}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial {a^{[1]}}_{n_1}}{\partial {a^{[0]}}_{1}} & \cdots & \frac{\partial {a^{[1]}}_{n_1}}{\partial {a^{[0]}}_{n_0}} \end{bmatrix}}_{\frac{\partial a^{[l]}}{\partial a^{[l-1]}}\text{: matrices}}$$

$$\longrightarrow$$

Backward computation, in the reverse order: L to 0

Backprop equation: forward evaluation

- Start with the Jacobian matrix of first hidden to input layer: $\frac{\partial a^{[1]}}{\partial a^{[0]}} = \begin{bmatrix} \frac{\partial a^{[1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[1]}{}_1}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$

- Compute the Jacobian of second hidden to input layer ➔

$$\frac{\partial a^{[2]}}{\partial a^{[0]}} = \frac{\partial a^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial a^{[0]}} = \begin{bmatrix} \frac{\partial a^{[2]}{}_1}{\partial a^{[1]}{}_1} & \cdots & \frac{\partial a^{[2]}{}_1}{\partial a^{[1]}{}_{n_1}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[2]}{}_{n_2}}{\partial a^{[1]}{}_1} & \cdots & \frac{\partial a^{[2]}{}_{n_2}}{\partial a^{[1]}{}_{n_1}} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[1]}{}_1}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[2]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[2]}{}_1}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[2]}{}_{n_2}}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[2]}{}_{n_2}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$$

- Carry on multiplying matrices up to layer L-1
  to sequentially obtain the Jacobians matrices of hidden layer 3, 4, …, L-1 to input layer:

$$\frac{\partial a^{[L-1]}}{\partial a^{[0]}} = \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \frac{\partial a^{[L-2]}}{\partial a^{[0]}} = \begin{bmatrix} \frac{\partial a^{[L-1]}{}_1}{\partial a^{[L-2]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_1}{\partial a^{[L-2]}{}_{n_{L-2}}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[L-2]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[L-2]}{}_{n_{L-2}}} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[L-2]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-2]}{}_{n_{L-2}}}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[L-2]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-2]}{}_{n_{L-2}}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[L-1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[L-1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$$

- Finally, compute the gradient of output to input layer: $\frac{\partial a^{[L]}}{\partial a^{[0]}} = \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[0]}} = \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[L-1]}{}_1} & \cdots & \frac{\partial a^{[L]}}{\partial a^{[L-1]}{}_{n_{L-1}}} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[L-1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[L-1]}{}_1}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L]}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$

- And the gradient of the loss: $\frac{\partial c}{\partial a^{[0]}} = \frac{\partial c}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial a^{[0]}} = 2\left(a^{[L]} - y\right) \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[0]}{}_1} & \cdots & \frac{\partial a^{[L]}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$

Forward evaluation: comments

- We recursively propagate **Jacobian matrices** to inputs $\frac{\partial a^{[l]}}{\partial a^{[0]}}$ from $\frac{\partial a^{[1]}}{\partial a^{[0]}}$ to $\frac{\partial a^{[L]}}{\partial a^{[0]}}$

- For each layer, we multiply matrices to obtain matrices, which is a **cubic** operation

- After we have $\frac{\partial a^{[L]}}{\partial a^{[0]}}$ and $\frac{\partial c}{\partial a^{[0]}}$, we must repeat the process to find $\frac{\partial c}{\partial a^{[1]}}, \frac{\partial c}{\partial a^{[2]}}, \ldots, \frac{\partial c}{\partial a^{[L-1]}}$

- The last comment is particular to ANNs (where we need gradients of loss to all layers)
  But the other comments are general (they apply whenever we need gradients of a final result to inputs)

Backprop equation: backward evaluation

- Start with the gradient vector of loss (real number) to last hidden layer L-1 (vector): $\frac{\partial c}{\partial a^{[L-1]}} = \underbrace{2\left(a^{[L]}-y\right)}_{\frac{\partial c}{\partial a^{[L]}}\text{: real number}} \underbrace{\begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[L-1]}{}_{1}} & \cdots & \frac{\partial a^{[L]}}{\partial a^{[L-1]}{}_{n_{L-1}}} \end{bmatrix}}_{\frac{\partial a^{[L]}}{\partial a^{[L-1]}}\text{: vector}}$

- Compute the gradient vector of loss to hidden layer L-2: $\frac{\partial c}{\partial a^{[L-2]}} = \frac{\partial c}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} = \begin{bmatrix} \frac{\partial c}{\partial a^{[L-1]}{}_{1}} & \cdots & \frac{\partial c}{\partial a^{[L-1]}{}_{n_{L-1}}} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[L-1]}{}_{1}}{\partial a^{[L-2]}{}_{1}} & \cdots & \frac{\partial a^{[L-1]}{}_{1}}{\partial a^{[L-2]}{}_{n_{L-2}}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[L-2]}{}_{1}} & \cdots & \frac{\partial a^{[L-1]}{}_{n_{L-1}}}{\partial a^{[L-2]}{}_{n_{L-2}}} \end{bmatrix} = \begin{bmatrix} \frac{\partial c}{\partial a^{[L-2]}{}_{1}} & \cdots & \frac{\partial c}{\partial a^{[L-2]}{}_{n_{L-2}}} \end{bmatrix}$

- Carry on multiplying vectors by matrices down to layer 0
  to recursively obtain the gradient vectors of loss to layer L-3, L-4, …, 1, 0:

$$\frac{\partial c}{\partial a^{[0]}} = \frac{\partial c}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial a^{[0]}} = \begin{bmatrix} \frac{\partial c}{\partial a^{[1]}{}_{1}} & \cdots & \frac{\partial c}{\partial a^{[1]}{}_{n_1}} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[1]}{}_{1}}{\partial a^{[0]}{}_{1}} & \cdots & \frac{\partial a^{[1]}{}_{1}}{\partial a^{[0]}{}_{n_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_{1}} & \cdots & \frac{\partial a^{[1]}{}_{n_1}}{\partial a^{[0]}{}_{n_0}} \end{bmatrix} = \begin{bmatrix} \frac{\partial c}{\partial a^{[0]}{}_{1}} & \cdots & \frac{\partial c}{\partial a^{[0]}{}_{n_0}} \end{bmatrix}$$

Backward evaluation: comments

- We recursively propagate **adjoint vectors** $\frac{\partial c}{\partial a^{[l]}}$ from $\frac{\partial c}{\partial a^{[L-1]}}$ to $\frac{\partial c}{\partial a^{[0]}}$

- Definition: given a result *c*, the **adjoint** of some number/vector/matrix *x* is $\frac{\partial c}{\partial x}$
- Note: the adjoint of a scalar result *c* is of the dimension of *x*, hence $\frac{\partial c}{\partial a^{[l]}}$ is a vector

- For each layer, we multiply vectors by matrices to obtain vectors, which is a **quadratic** operation

- For ANNs, we repeat the process only once to find all the $\frac{\partial c}{\partial a^{[L-1]}},\ldots,\frac{\partial c}{\partial a^{[2]}},\frac{\partial c}{\partial a^{[1]}},\frac{\partial c}{\partial a^{[0]}}$

- It follows that backward evaluation is one order of magnitude faster than forward evaluation
  - Because the output layer is scalar, its adjoints are vectors
  - Whereas the input layer is a vector, hence, its Jacobians are matrices
  - To propagate adjoints is (one order of magnitude) more efficient!

Computing cost differentials: MLPs

- Differentiation equations
  - Eq1 is unchanged: $c=(\hat{y}-y)^2=\left(a^{[L]}-y\right)^2$ so $\frac{\partial c}{\partial \hat{y}}=\frac{\partial c}{\partial a^{[L]}}=2\left(a^{[L]}-y\right)=\text{"2 prediction errors"}$
  - Recall MLPs are defined by the feed-forward equation: $a^{[l]}=g^{[l]}\left(\underbrace{W^{[l]}a^{[l-1]}+b^{[l]}}_{z^{[l]}}\right)$ where *g* is applied element-wise to the vector *z*
    - By differentiation, we have the Jacobian matrix of $a^{[l]}$ to a$^{[l-1]}$ (eq2): $\underbrace{\frac{\partial a^{[l]}}{\partial a^{[l-1]}}}_{n_l\times n_{l-1}}=\frac{\partial a^{[l]}}{\partial z^{[l]}}\frac{\partial z^{[l]}}{\partial a^{[l-1]}}=\underbrace{diag\left[\underbrace{g^{[l]}{}'\left(z^{[l]}\right)}_{n_l\times 1}\right]}_{n_l\times n_l}\underbrace{W^{[l]}}_{n_l\times n_{l-1}}$
    - And derivatives to parameters (eq3): $\frac{\partial c}{\partial W^{[l]}{}_{ij}}=\frac{\partial c}{\partial a^{[l]}{}_i}\frac{\partial a^{[l]}{}_i}{\partial W^{[l]}{}_{ij}}=\frac{\partial c}{\partial a^{[l]}{}_i}g^{[l]}{}'\left(z^{[l]}{}_i\right)a^{[l-1]}{}_j$ and $\frac{\partial c}{\partial b^{[l]}{}_i}=\frac{\partial c}{\partial a^{[l]}{}_i}\frac{\partial a^{[l]}{}_i}{\partial b^{[l]}{}_i}=\frac{\partial c}{\partial a^{[l]}{}_i}g^{[l]}{}'\left(z^{[l]}{}_i\right)$
- Differentiation strategy
  - Apply eq1 and eq2 to compute the adjoints $\frac{\partial c}{\partial a^{[l]}}$ for all layers l
  - Then, apply eq3 to compute the desired sensitivities to parameters $\frac{\partial c}{\partial W^{[l]}{}_{ij}}$ and $\frac{\partial c}{\partial b^{[l]}{}_i}$

Jacobian propagation

- Computation of $\frac{\partial c}{\partial a^{[l]}}$ by Jacobian propagation in increasing order of layers:
  - Start with the Jacobian matrix $\frac{\partial a^{[l+1]}}{\partial a^{[l]}}$, known from eq2
  - Multiply by the Jacobian matrix $\frac{\partial a^{[l+2]}}{\partial a^{[l+1]}}$ to produce the Jacobian matrix $\frac{\partial a^{[l+2]}}{\partial a^{[l]}} = \frac{\partial a^{[l+2]}}{\partial a^{[l+1]}} \frac{\partial a^{[l+1]}}{\partial a^{[l]}}$
  - Perform *L-l* matrix by matrix products (cubic complexity) to obtain Jacobian matrices $\frac{\partial a^{[L]}}{\partial a^{[l]}} = \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \cdots \frac{\partial a^{[l+1]}}{\partial a^{[l]}}$
  - Finally, $\frac{\partial c}{\partial a^{[l]}} = \frac{\partial c}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial a^{[l]}}$ where $\frac{\partial c}{\partial a^{[L]}}$ is known from eq1
  - Repeat for every layer l $\quad \frac{\partial a^{[l+1]}}{\partial a^{[l]}} \longrightarrow \frac{\partial a^{[l+2]}}{\partial a^{[l]}} \longrightarrow \frac{\partial a^{[l+3]}}{\partial a^{[l]}} \longrightarrow \frac{\partial a^{[l+k]}}{\partial a^{[l]}} \longrightarrow \frac{\partial a^{[L]}}{\partial a^{[l]}} \longrightarrow \frac{\partial c}{\partial a^{[l]}}$

- Obviously inefficient:
  - Must repeat for every level l
  - Sequence of expensive (cubic) matrix by matrix products

Adjoint propagation

- Computation of all the $\frac{\partial c}{\partial a^{[l]}}$ by adjoint propagation in the reverse order of layers:
  - Start with the **real number** $\frac{\partial c}{\partial a^{[L]}}$ known from eq1
  - Multiply by the **gradient vector** $\frac{\partial a^{[L]}}{\partial a^{[L-1]}}$ known from eq2 to produce the **adjoint vector** $\frac{\partial c}{\partial a^{[L-1]}} = \frac{\partial c}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial a^{[L-1]}}$
  - Perform *L* **matrix by vector** products $\frac{\partial c}{\partial a^{[l-1]}} = \frac{\partial c}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial a^{[l-1]}}$ (quadratic complexity) to obtain all the adjoint vectors $\frac{\partial c}{\partial a^{[l]}}$

$$\begin{tikzcd}
\frac{\partial c}{\partial a^{[1]}} & \frac{\partial c}{\partial a^{[2]}} \arrow[l] & \frac{\partial c}{\partial a^{[3]}} \arrow[l] & n_l & \frac{\partial c}{\partial a^{[l]}} \arrow[l] & \frac{\partial c}{\partial a^{[L-1]}} \arrow[l] & \frac{\partial c}{\partial a^{[L]}} \arrow[l]
\end{tikzcd}$$

- Complexity: all $\frac{\partial c}{\partial a^{[l]}}$ and hence all $\frac{\partial c}{\partial W^{[l]}_{ij}}$ and $\frac{\partial c}{\partial b^{[l]}_{i}}$ are computed with *L* matrix by vector products
  - Same complexity as for making a prediction / computing a loss
  - **Constant time differentiation**: all differentials of *c* are produced with same (leading) complexity as one evaluation of *c*

Back-Propagation

- 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
  - Relies on the cost being scalar, hence its adjoints are vectors
  - Is the only viable means to train deep nets in reasonable time
  - Is the reason why deep learning achieved such spectacular success and why your phone quickly learns to recognise you
- 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 propagate adjoints in the reverse order through evaluation graphs
  - We will learn to automatically generate graphs
  - Not only for deep nets, but for any calculation
  - Including a large Monte-Carlo simulation

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” computations involved in a calculation
  - Any calculation defines an evaluation graph
  - To some extent, every calculation defines a sequence of feed-forward operations, similarly to a neural net
  - Chapter 9 of the AAD book teaches to build evaluation graphs in memory with operator overloading in C++

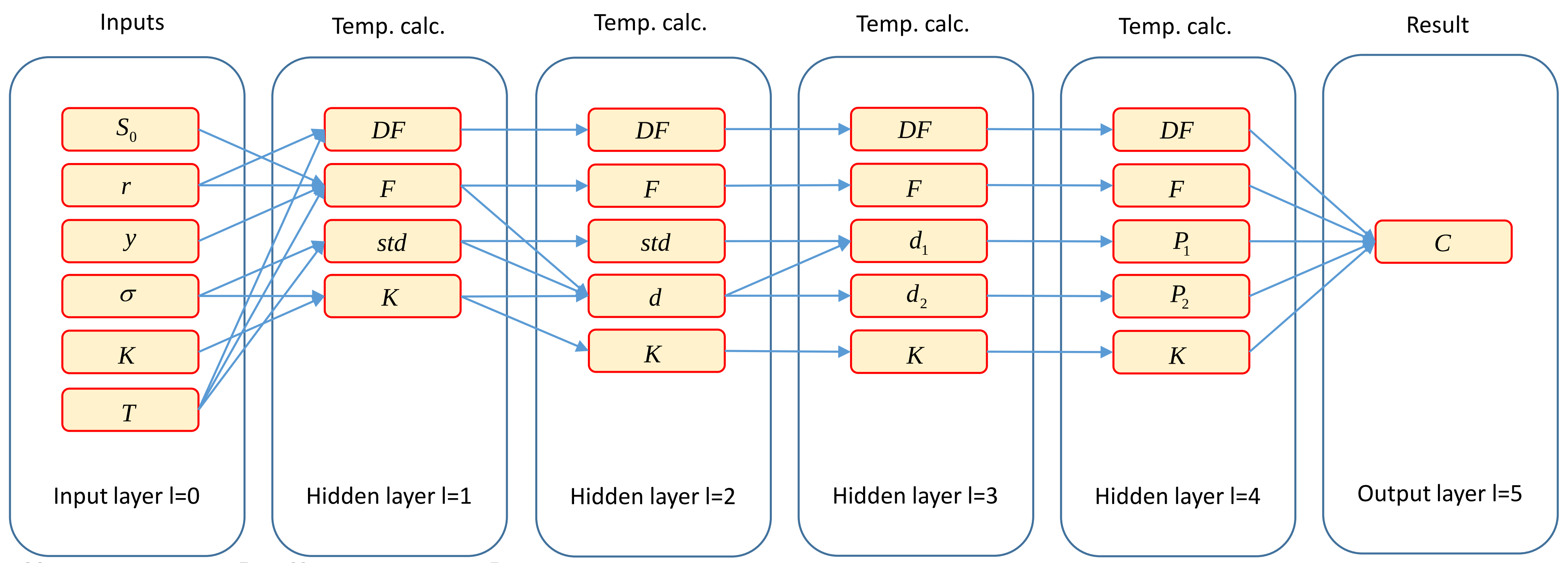
- Adjoint Differentiation
  - Is not limited to neural nets
  - Is applicable to any calculation
  - Propagates adjoints through the evaluation graph
  - Offers constant time efficiency (all differentials for an expense similar to one evaluation) for scalar calculations
  - (A scalar calculation is one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

- Once we have an evaluation graph, AD may be automated

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



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

$n_0 = 6$

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

$n_1 = 4$

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

$n_2 = 5$

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

$n_3 = 5$

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

$n_4 = 5$

$a^{[5]} = C = g^{[5]}\left(a^{[4]}\right)$

$n_5 = 1$

Feed-forward equations

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

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

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

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

- In Black & Scholes, we have

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

Differentiation equations

- For **any** calculation, where we compute the differentials of the result to the inputs: $\frac{\partial y}{\partial x}=\frac{\partial a^{[L]}}{\partial a^{[0]}}$
- If we split the calculation into small enough pieces, we can easily compute the Jacobians $g^{[l]}{}'=\frac{\partial a^{[l]}}{\partial a^{[l-1]}}$
- In Black & Scholes

$$g^{[1]}{}'=\begin{pmatrix} 0 & -x_6\exp(-x_2x_6) & 0 & 0 & 0 & -x_2\exp(-x_2x_6) \\ \exp[(x_2-x_3)x_6] & x_1x_6\exp[(x_2-x_3)x_6] & -x_1x_6\exp[(x_2-x_3)x_6] & 0 & 0 & x_1(x_2-x_3)\exp[(x_2-x_3)x_6] \\ 0 & 0 & 0 & \sqrt{x_6} & 0 & \frac{x_4}{2\sqrt{x_6}} \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$g^{[2]}{}'=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{x_2x_3} & -\frac{\log(x_2/x_4)}{x_3{}^2} & -\frac{1}{x_3x_4} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$g^{[3]}{}'=\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 & 0 \\ 0 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$g^{[4]}{}'=\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & n(x_3) & 0 & 0 \\ 0 & 0 & 0 & n(x_4) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$g^{[5]}{}'=\left((x_2x_3-x_5x_4),x_1x_3,x_1x_2,-x_1x_5,-x_1x_1x_4\right)$$

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

Forward and backward differentiation

- For **any** calculation, we have: $\frac{\partial y}{\partial x}=\frac{\partial a^{[L]}}{\partial a^{[0]}}=\frac{\partial a^{[L]}}{\partial a^{[L-1]}}\frac{\partial a^{[L-1]}}{\partial a^{[L-2]}}\cdots\frac{\partial a^{[l]}}{\partial a^{[l-1]}}\cdots\frac{\partial a^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial a^{[0]}}$

- If we compute this product right to left:
  - We follow the order of the calculation
  - We propagate Jacobians to the inputs up from layer 1 to the output layer
  - We repeatedly multiply matrices with matrices, with cubic complexity

- If we compute this product left to right:
  - We follow the reverse order of the calculation
  - We propagate adjoints of the result down from layer *L-1* to the input layer
  - Provided the final result is scalar, we repeatedly multiply matrices by vectors, with quadratic complexity

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

Evaluation graphs: conclusion

- All calculations define a graph
- Like deep nets, evaluation graphs may be organized in successive layers function of one another: $a^{[l]} = g^{[l]}\left(a^{[l-1]}\right)$
- Once the evaluation graph is known down to elementary operations:
  - The Jacobians $g^{[l]\,\prime} = \frac{\partial a^{[l]}}{\partial a^{[l-1]}}$ are found trivially
  - Adjoint differentiation may be conducted an order of magnitude faster than Jacobian propagation (provided the calculation is scalar):

$\longrightarrow$ adjoint propagation, quadratic

$$\frac{\partial y}{\partial x} = \frac{\partial a^{[L]}}{\partial a^{[0]}} = \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial a^{[L-2]}} \cdots \frac{\partial a^{[l]}}{\partial a^{[l-1]}} \cdots \frac{\partial a^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial a^{[0]}}$$

$\longleftarrow$ Jacobian propagation, cubic

- Next, we show how evaluation graphs may be **automatically** generated

Recording calculations on tape

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 AAD book for professional, generic, parallel, efficient implementation
- **All the code is freely available on GitHub, in the file toyCode.h**

http://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 call TensorFlow API to evaluate or differentiate the graphs efficiently: AD, parallel CPU, 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 similar to coding calculations)
  - What we want here is take some calculation code and automatically extract its graph
- Source transformation
  - Code that reads and understands evaluation code, extracts graph 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) developers decide what exactly is executed:

```
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 operator+
myNumberType t = log(x);    //  executes code in log
```

Recording operations

- We apply operator overloading to **record** all operations on tape:

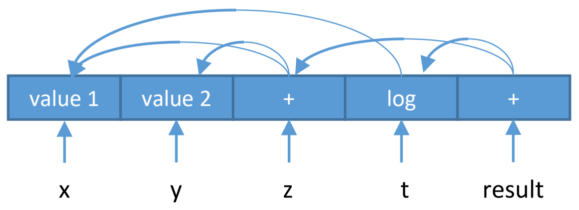
```cpp
class myNumberType
{
    myNumberType(const double x)
    {
        //  constructor initializes value to x and record node
    }
};

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
myNumberType t = log(x);        //  records log
myNumberType result = t + z;    //  records addition
```

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 sequence 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 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 $f = +, *, -, /, \log, \sqrt{\ }, \ldots$
  - 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

```
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

```
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

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

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

 //  ...

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

 //  ...
}
```

```
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;

 //  ...
}
```

```
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:

```
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

```
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;
}
```

```
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;
}
```

```
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):

```
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;
}
```

```
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:

```
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

```
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

```
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

```
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

```
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
```

AAD

State of the tape after recording

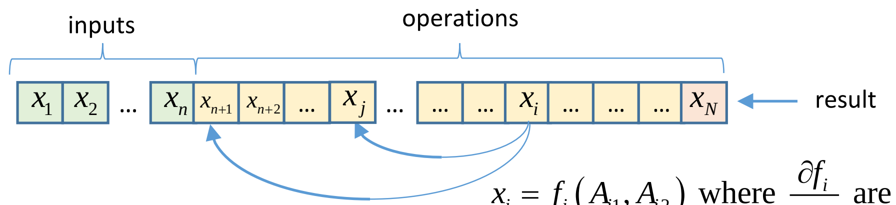
- We call the instrumented instance of our code:

```
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 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 $\frac{\partial f_i}{\partial x_j}$ for all $j \in A_i$ have been computed and recorded on tape during evaluation

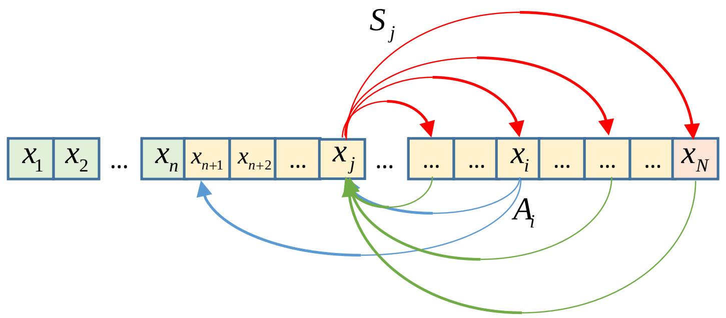
- Our tape therefore looks like:



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

Successors and adjoints

- Successors
  - 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_i may contain many indices
  - Whereas an empty S_i reveals an unused input or intermediate result



- Adjoint equation
  - Denote $\bar{x}_i \equiv \frac{\partial y}{\partial x_i} = \frac{\partial x_N}{\partial x_i}$ the adjoint of x_i
  - Then (evidently): $\bar{x}_N = 1$
  - And in a direct application of the chain rule: $\bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$ because $\frac{\partial y}{\partial x_j} = \sum_{i \in S_j} \frac{\partial y}{\partial x_i} \frac{\partial x_i}{\partial x_j}$ and we recall that all the $\frac{\partial x_i}{\partial x_j}$ are on tape

Adjoint propagation

- Adjoints therefore satisfy a backward recursion
  - The last adjoint $\bar{x}_N = 1$ is given
  - All other adjoints are a function of future adjoints: for all j $\bar{x}_j = f\left(\bar{x}_i, i \in S_j, i > j\right)$
- Therefore the following algorithm is guaranteed to correctly accumulate all adjoints
  - Initialize all adjoints to 0 and the last adjoint to 1 (this is called *seeding* the tape): $\bar{x}_j = \delta_{N-j}$
  - Repeat for *i* iterating backwards from *N* to *0:* $\text{for all } j \in A_i : \bar{x}_j \leftarrow \bar{x}_j + \frac{\partial f_i}{\partial x_j} \bar{x}_i$
  - 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 adjoint propagation

Adjoint propagation code

```
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;
}
```

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

Adjoint propagation code

- After we record a tape by calling 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
```

- We proceed with adjoint propagation:

```
//  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

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

- Next, we apply AAD to a barrier option in a Monte-Carlo simulation of Dupire's model

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

Simulation code, version 1

```
//  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)
```

```
//  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 scenarios
  - 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 phones are multi-core today, professional code is always 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 functions
    - 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:

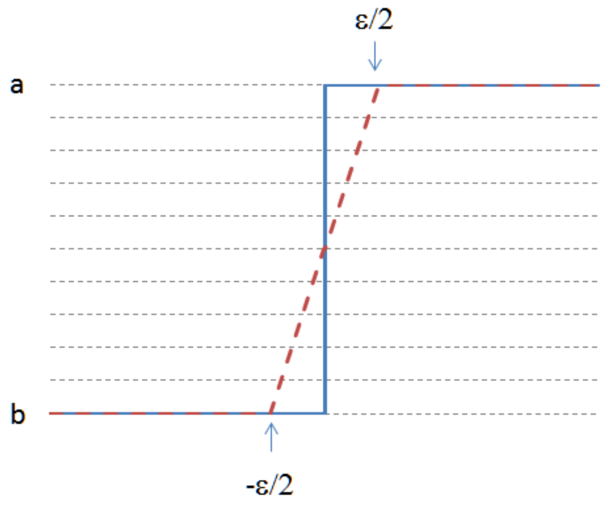
http://www.slideshare.net\AntoineSavine\stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic

freely available on slideShare

- Here, we briefly explain the “smooth barrier” algorithm, universally 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



Simulation code with smooth barrier

```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

Differentiation code

```
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 DupureRisksMiniBatch(), call it sequentially from a wrapper function

```
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 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

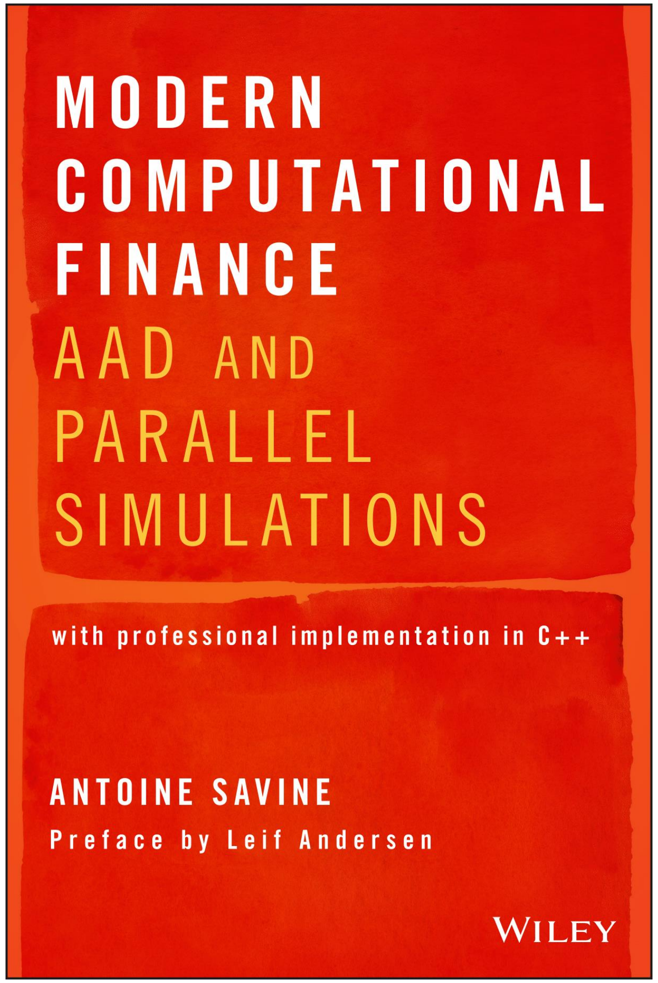
Conclusion

- 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:

http://github.com/asavine/CompFinance/Intro2AADinMachineLearningAndFinance.pdf

http://tinyurl.com/savineAadTalk

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