



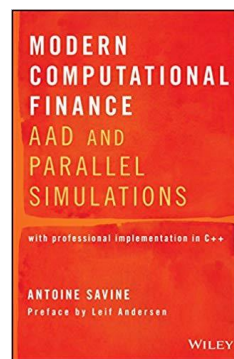
Computational Finance and Machine Learning in Finance

Antoine Savine

all the material: slides, code, notebooks, spreadsheets and more

[GitHub.com/aSavine/CompFinLecture/](https://github.com/aSavine/CompFinLecture/)

your curriculum:



Need for Speed

Need for Speed

- Recall from basic theory of derivatives:
- Institutions must compute the value of the many transactions in their derivatives books
- And their sensitivities to (many) market variables
remember sensitivities are hedge ratios
- Rapidly, so they can hedge risk
- *Before* market moves



Need for Speed: derivatives books

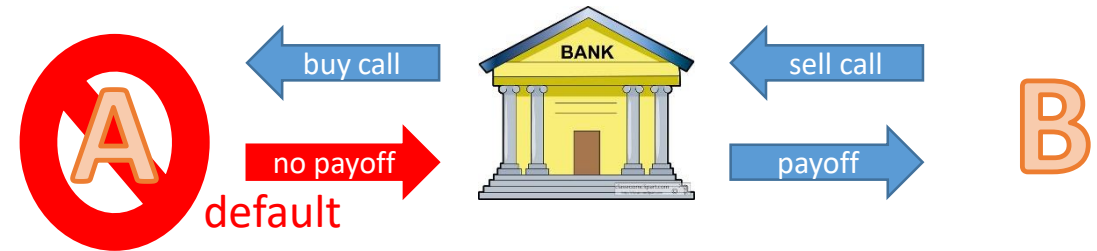
idealized world	real world
Transaction: European call option	Transactions: European calls and puts, basket/worst-of options, caps and swaptions, Bermudan (early callable) transactions, barrier and other path-dependent options, hybrids (dependent on many assets of different classes), volatility swaps, autocallables...
Model: Black & Scholes	Sophisticated models match complex risks <ul style="list-style-type: none"> - Local and stochastic volatility (Dupire) - Yield curve models (Heath-Jarrow-Morton) - Many more, all the way to Gatheral's recent "rough volatility" models
Parameters: Spot and Volatility	Many parameters reflect complex markets <ul style="list-style-type: none"> - Interest rate curves (by maturity) - Volatility surfaces (by expiry and strike) - Correlations, volatility of volatility and more
Pricing and differences all analytic $V_t = S_t N(d_1) - KN(d_2), \frac{\partial V_t}{\partial S_t} = N(d_1), \frac{\partial V_t}{\partial \sigma} = S_t n(d_1) \sqrt{T-t}, d_{1/2} = \frac{\log \frac{S_t}{K} \pm \frac{\sigma^2}{2}(T-t)}{\sigma \sqrt{T-t}}$	Due to complexity and dimensionality: <ul style="list-style-type: none"> - Analytics almost never feasible - FDM (finite difference methods) almost always impractical - Slower, less accurate Monte-Carlo the only practical solution
Pricing and risk takes a few calls to analytic functions	Pricing and risk takes thousands of differentials of prices of thousands of transactions, estimated with thousands of Monte-Carlo simulations

Need for Speed: regulations

- Modern models, algorithms and hardware can deal with derivatives books
- But regulators insist (for good reason) that we compute value and risk
- Not only today
- But also:
 - On a set of future dates: “exposure dates”
 - In multiple scenarios of market evolution
- So even basic transactions like swaps become highly exotic
- And computation requirements increase exponentially

Counterparty Value Adjustment (CVA)

- Prominent example of regulatory risk computation: Counterparty credit Risk Adjustment (CVA)
- Say I buy a 2Y ATM (spot=100) call from party A, sell it to party B
 - This is called “back to back trading”, in principle, risk-free
- One year from now, party A defaults
 - A will not deliver the payoff of the call one year later
 - My position against B is now unhedged
 - I can hedge it by buying another 1Y 100 call in the market
 - Or implement a replication strategy
 - Either way, I must pay the premium of the call, again
- Counterparty default may produce a loss (of a stochastic amount dependent on market on default date) but never a gain
- Every time I trade with a defaultable party, I am giving away *a real option*



CVA as an option

- At the time T when a counterparty defaults, I have a (negative) payoff of $\max(V_T, 0) = V_T^+$
- where V is the value, at T , of the sum of all transactions with the party (called “netting set”)
- including swaps, options, exotics, hybrids, etc.
- Note that V depends on the market value at T of all assets underlying to all transactions in netting set
- plus path-dependencies (were Bermudans early exercised before T ? were barriers hit before T ? Etc.)
- The payoff of the real option I am implicitly giving away when trading is therefore: $\sum_{i=1}^n 1_{\{T_{i-1} \leq \tau < T_i\}} V_{T_i}^+$
- where τ is the (stochastic) time of default
- this is also often written as a continuous integral
- This is a complicated, hybrid option that depends on credit and the joint dynamics of all underlyers in the netting set
- even when the transactions in the netting set are basic

CVA: pricing

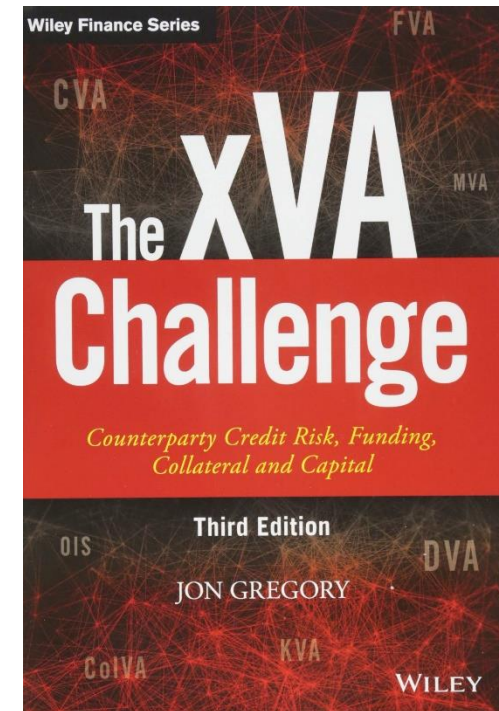
- CVA is priced like any other option (ignoring discounting for simplicity):
$$CVA = E^Q \left[\sum_{i=1}^n 1_{\{T_{i-1} \leq \tau < T_i\}} V_{T_i}^+ \right]$$
- Assuming credit is independent from market drivers of the netting set:
$$CVA = \sum_{i=1}^n Q(T_{i-1} \leq \tau < T_i) E^Q [V_{T_i}^+]$$
- the left term in the sum is the (risk neutral) probability of default, can be read in credit markets (Bonds, credit default swaps)
- the right term $E^Q [V_{T_i}^+]$ is the expected *loss given default* (LGD) , or “exposure”
- Exposures depend on the joint dynamics of all asset that affect the netting set --- so we need:
 - Complex hybrid models, in practice “stitching together” models of all underlying assets, with special care to remain consistent and arbitrage-free
 - Monte-Carlo simulations (given the high dimensionality)
 - Some way to compute future book values V from the simulated state of the market at T
 - for example with *nested simulations* (Monte-Carlo *within* Monte-Carlo)
 - can we do better? for example, “learn” future values from data, with machine learning?
 - All of this is a (very) active field of current research, see for example Huge & Savine’s [Deep Analytics](#)
- The computational requirement is massive: CVA is often computed in large data centres
- With better algorithms and platforms, we can compute it locally --- see Andreasen’s influential [Calculate CVA on your iPad Mini](#)
- But it may still take several minutes to compute the price alone

CVA: risk conundrum

- Regulators insist that institutions (frequently) compute CVA and incorporate it in accounts as a loss, hence “adjustment”
- From one day to next, market may move and modify CVA by hundreds of millions
- But CVA is part of the bank’s profit and loss account
- Banks cannot tolerate such swings
- Like any other option, CVA must be hedged
- Compute sensitivities of CVA to all market variables, and hedge them in the market, as usual
- Problem is: CVA typically depends on thousands of market variables
 - all the rate curves and volatility surfaces of all the currencies in the netting set
 - all the exchange rates and their volatility (surfaces)
 - plus of course all underlying assets and their volatility (surfaces)
- If it takes minutes to compute one CVA price, it may take *days* to compute risk of *one* netting set
- by traditional means, bump market variables one by one and recalculate, also called “finite differences”
- this is far too late: we must hedge before asset prices change in fast moving markets

Speed is a matter of life or death

- Speed (without loss of accuracy!) is a matter of survival
- In addition to CVA, banks must compute (and hedge!!) similar adjustment for funding (FVA), initial margin (MVA) and many more
- collectively called “XVA”, see Gregory’s classic XVA Challenge
- Banks who can’t compute and hedge risks on time are out of the business
- Principal focus of quantitative research and development groups is shifting towards fast models, algorithms and platforms



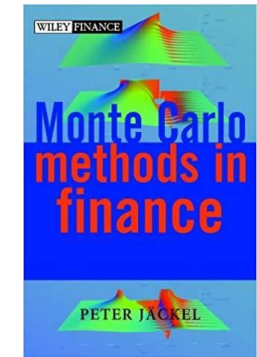
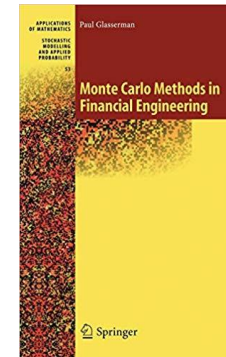
How to achieve speed: classic avenues



faster

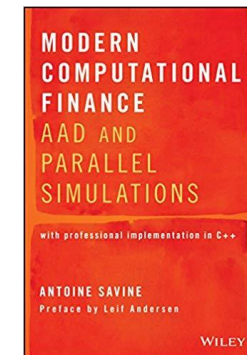
1. Improve Monte-Carlo convergence

- “Better random numbers”: Sobol’s sequence
- Control variates:
 - Antithetic: for every path, generate another with reflected Brownian
 - Model specific: adjust paths to hit discount factors, European options...
- Many flavours of variance reduction techniques (Generally problem specific)
- Well known and taught
all explained in Glasserman and Jaeckel’s classic books
- Improvement typically (sub-)linear



2. Leverage modern hardware

- Cache awareness
- Multi-threaded programming, SIMD
- GPU, TPU
- Less classic, seldom taught in finance
all explained in Savine’s Modern Computational Finance
- Yet major (linear) improvement on modern hardware



How to achieve speed: cutting edge



fastest

3. Adjoint Differentiation (AD)

- We compute differentials by bumping, repeatedly recalculating value
- What if we could compute value and all differentials, at once, in a time *independent of the number of differentials*?
- We would get full risk for a time similar to one computation of the value
- This is what AD and its automatic instance AAD offer:
an **order of magnitude** improvement in speed,
and the most prominent leap in computational finance since finite difference methods (FDM) in the 1990s
- Recent addition to finance
 - Smoking Adjoints paper by Giles & Glasserman, 2006
 - Award-winning Bank-wide implementation in Danske Bank, 2012-2015
- Not yet broadly understood or systematically taught in financial programs
- Similar tech also powers machine learning (where it is called “back-prop”)
- AD is the principal focus of this course
- and its curriculum Modern Computational Finance book

How to achieve speed: future tech



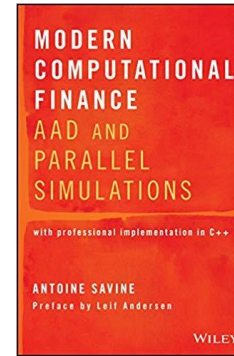
even
faster?

4. Rise of Machine Learning

- Values and risks of trading books are (very complicated) functions of the state (market + path-dependencies)
- In principle, this function can be taught to machines
- In a way similar to how your phone identifies people by “looking at” them: target (identity) is a (learned) function of state (pixels on camera)
- Many ideas from the Machine Learning (mainly Deep Learning) are being applied in finance as a very current, very active area of research & development
- May lead to another leap in computational speed
- See first public milestones (all from 2018-2019):
 - Horvath, [Deep Learning Volatility](#)
 - Mc Ghee, [An Artificial Neural Network Representation of the SABR Stochastic Volatility Model](#)
 - Huge & Savine, [Deep Analytics](#)

Summary of the course

1. Programming for speed: 1-2 hours
2. Deep Learning and back-prop: 4-5 hours
3. Adjoint Differentiation: 3-4 hours
4. Design Patterns for Simulation Systems: 1-2 hours
5. Parallel simulations in C++: 1-2 hours



your curriculum

[GitHub.com/aSavine/CompFinLecture/](https://github.com/aSavine/CompFinLecture/)

your companion repo

Important note:

- in these lectures, we implement basic, toy C++ code, optimized for *simplicity* and *pedagogy*
- professional code, optimized for *production* and *efficiency*, is available in the curriculum
- and its own GitHub repo [GitHub.com/aSavine/compFinance](https://github.com/aSavine/compFinance)

Programming for Speed

Demonstration 1: matrix product 30x faster

C++ code:

[GitHub.com/aSavine/CompFinLecture/MatrixProduct](https://github.com/aSavine/CompFinLecture/MatrixProduct)

Python notebook:

[GitHub.com/aSavine/CompFinLecture/MatrixProductPython](https://github.com/aSavine/CompFinLecture/MatrixProductPython)

Curriculum:

chapter 1

Demonstration 2: 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: $\sigma^2(K, T) = \frac{2 \frac{\partial C}{\partial T}}{\frac{\partial^2 C}{\partial K^2}}$ 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: 21 spots (every 5 points 50 to 200) and 36 times (every month from now to 3y)
we have 1,080 volatilities + 1 initial spot (=100) = 1,081 model parameters
- Valuation of a 3y (weekly monitored) barrier option strike $K=120$, barrier $B=150$: $v(S_t, t) = E \left[(S_T - K)^+ 1_{\{\max(S_{T_1}, \dots, S_{T_K}) < B\}} \middle| S_t \right]$
- 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 2: Results

- Implementation
 - C++ code exported to Excel (see [tutorial ExportingCpp2xl.pdf](#) on [GitHub.com/aSavine/compFinance/xlCpp/](https://github.com/aSavine/compFinance/xlCpp/))
 - Modern library design with efficient implementation (Chapter 6)
 - Parallel implementation (Chapters 3 and 7)
 - Sobol quasi-random numbers (Chapters 5 and 6), excerpt here: medium.com/@antoine_savine/sobol-sequence-explained-188f422b246b
 - Advanced AAD with expression templates (Chapter 15)
- Hardware: quad-core laptop (surface book 2, 2017)
- Performance:
 - One evaluation with 500,000 paths over 156 time steps take ~0.8sec
 - We have 1,081 risk sensitivities, take about 15 minutes to produces model risk report with linear differentiation
 - With AAD the 1,081 differentials are produced in ~1.5sec

Demonstration 2: material

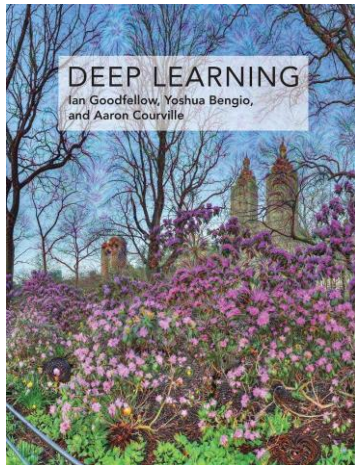
- GitHub repo of the curriculum [Github.com/aSavine/CompFinance](https://github.com/aSavine/CompFinance)
- Prebuilt Xl addin **xlComp.xll** (may need to install redistributables VC_redist.x86.exe and VC_redist.x64.exe, provided)
- Demonstration spreadsheet **xlTest.xlsx**

[illegible][illegible]

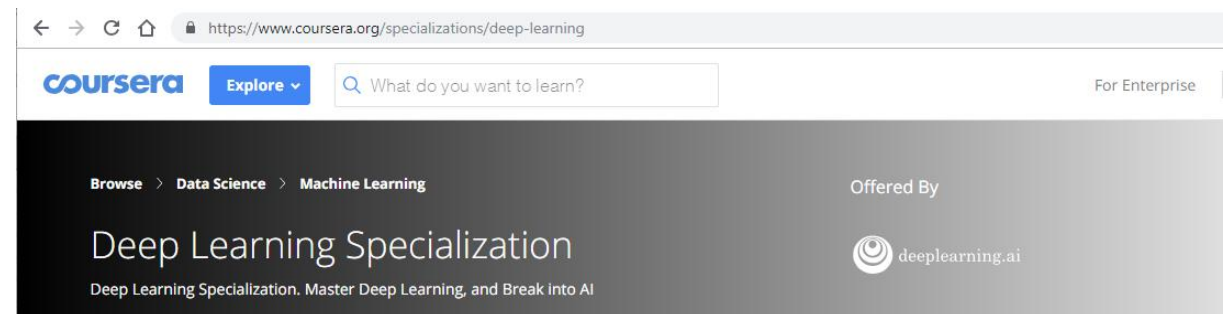
Deep Learning

Neural networks and deep learning

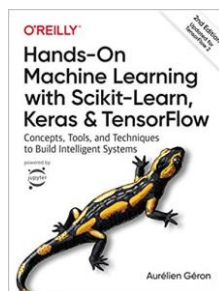
- We *briefly* introduce deep learning here
- A lot of learning material is found in books, MOOCs and blogs
- Quality is uneven, we recommend the following:



Goodfellow's reference book



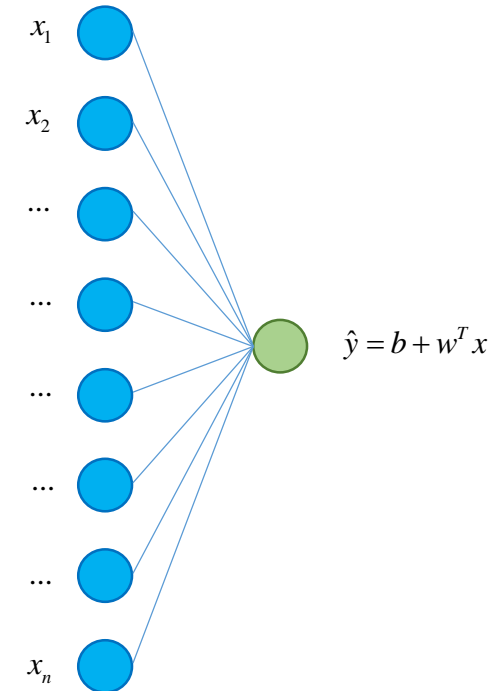
Andrew Ng's lectures on Coursera



Geron's practical introduction
with TensorFlow (2nd ed)

Linear regression

- Linear model (joint Gaussian assumptions): $\hat{y} = E[y|x] = b + \sum_{i=1}^n w_i x_i = b + w^T x$
- Parameters: $b \in \mathbb{R}$ and $w \in \mathbb{R}^n$
- Parameters are fitted to a *dataset* set of input vectors $x^{(1)}, \dots, x^{(m)}$ where targets $y^{(1)}, \dots, y^{(m)}$ are known
- *Fit* means find parameters that minimize prediction errors
- Then, the fitted model – with fitted parameters $b \in \mathbb{R}$ and $w \in \mathbb{R}^n$ is applied to new examples x to make predictions $\hat{y} = b + w^T x$ for unknown targets y



Linear classification

- Alternatively, *classification* problems predict discrete categories

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

picture 1920x1080x24bit colour



vector of 1920x1080 pixels

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

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

softmax regression

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

Parameters:

b: vector of dimension 5

w: matrix 5 x 2,073,600

s: softmax function

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

$$s(z) = \begin{bmatrix} \frac{e^{z_i}}{\sum e^{z_j}} \end{bmatrix}$$

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

Linear fit

- Training set of m examples $x^{(1)}, \dots, x^{(m)}$ (each a vector in dimension n) with corresponding labels $y^{(1)}, \dots, y^{(m)} \in \mathbb{R}$
- Learn parameters: $b \in \mathbb{R}$ and $w \in \mathbb{R}^n$ by minimizing prediction errors (cost function) $C(b, w) = \sum_{i=1}^m \left(\underbrace{b + w^T x^{(i)}}_{=\hat{y}^{(i)}} - y^{(i)} \right)^2$
- Note that this is the same as maximizing (log) likelihood under Gaussian assumptions, hence:

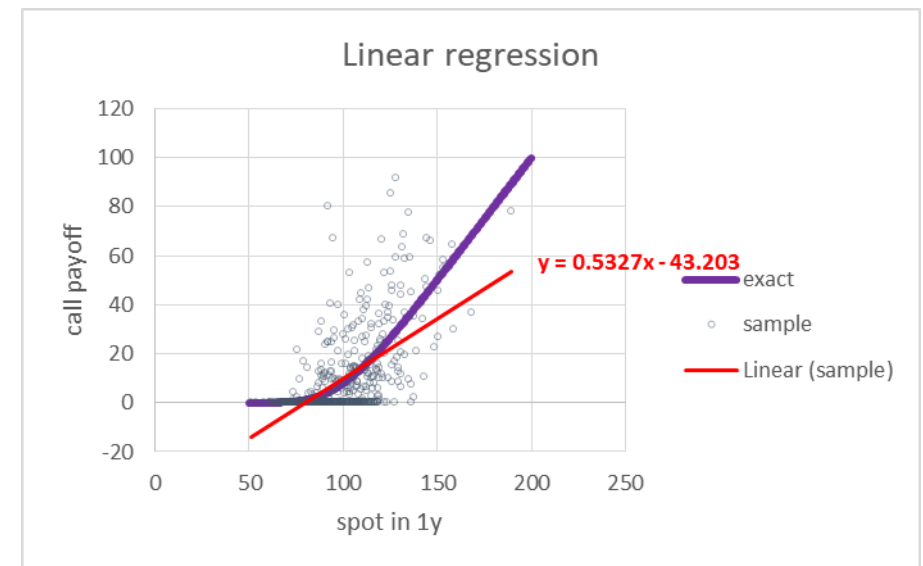
$b^*, w^* = \arg \min C(b, w)$ are the maximum likelihood estimators (MLE) for the Gaussian model

- b^* and w^* are found analytically, solving for $\frac{\partial C}{\partial b} = 0$ and $\frac{\partial C}{\partial w} = 0$
- Result (“normal equation”): $\begin{pmatrix} b \\ w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} = (X^T X)^{-1} X^T Y$ where $X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \dots & x_n^{(2)} \\ \dots & x_1^{(i)} & x_j^{(i)} & x_n^{(i)} \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}$ and $Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(m)} \end{pmatrix}$
- Solution is analytic and *unique*: no local minima since C is convex (quadratic) in b and w

Linear limitations

We introduce an important example: the *revaluation* of a European call

- Predict the future price in 1y of a European call strike 100, maturity 2y
- By regression of the payoff in 2y: $y^{(i)} = (S_{2y}^{(i)} - K)^+$
- Over the underlying asset price in 1y: $x^{(i)} = S_{1y}^{(i)}$
- Given a training set of m paths $(S_{1y}^{(i)}, S_{2y}^{(i)})_{1 \leq i \leq m}$
generated under Black & Scholes' model (spot = 100, volatility of 20%)
- We know the exact solution is given by Black and Scholes' formula so we can assess the quality of the regression

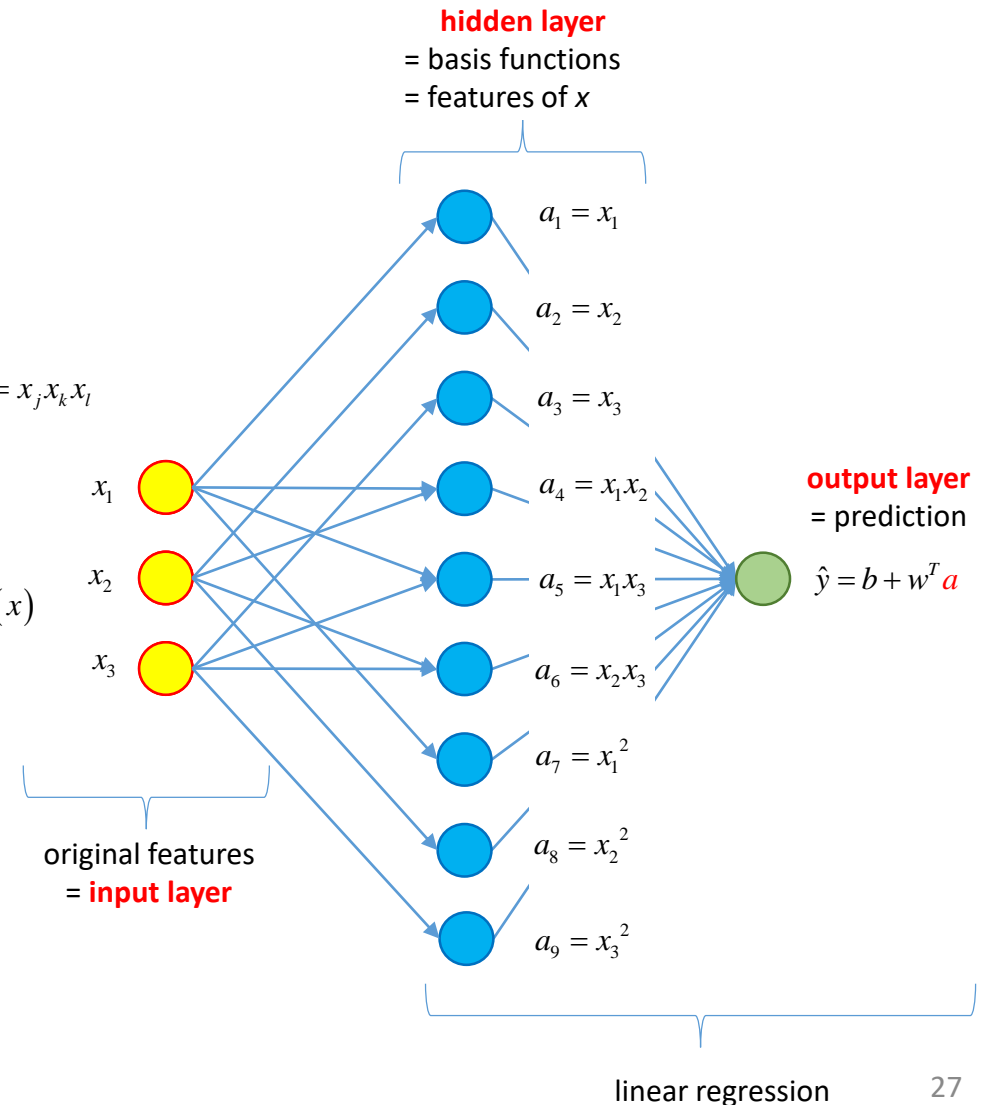


- See spreadsheet [reval.xlsx](#) on the repo [Github.com/aSavine/CompFinLecture/RegressionSheet/](https://github.com/aSavine/CompFinLecture/RegressionSheet/)
- Linear regression obviously fails to approximate the correct function because it cannot capture non-linearities

Basis function regression

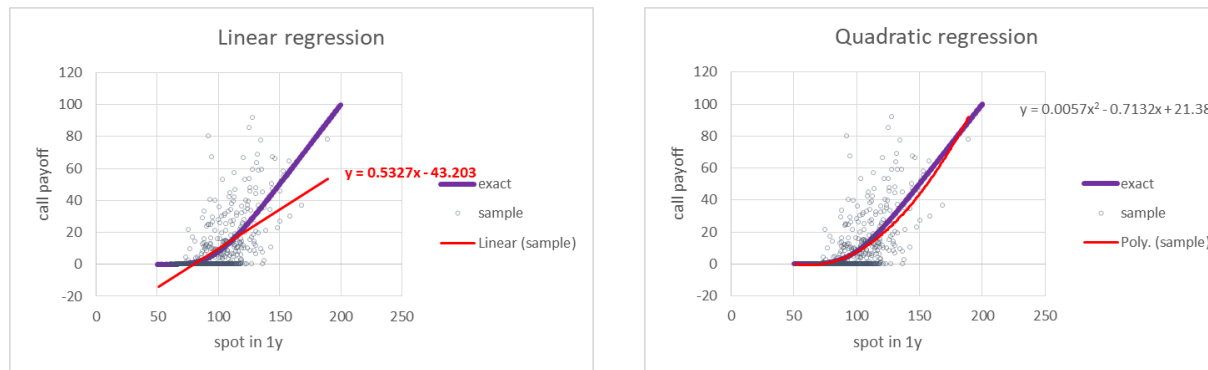
- Solution: regress not on x but on basis functions a of x
- Example: polynomial regression
 - Basis functions = monomials (x is in dimension n_0):
 - 1st degree: all the x s $a_i = x_i$
 - 2nd degree: all the squares $a_i = x_j^2$ and pair-wise products $a_i = x_i x_j$
 - 3rd degree: all the cubes $a_i = x_j^3$ 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 inputs
 - Compute vector of n_1 basis functions (or “features” in ML lingo): $a = \varphi(x)$
 - Predict linearly **in the basis functions**: $\hat{y} = b + w^T a$
- Fitting: identical to linear regression on a in place of x

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



basis function regression: performance

- Works nicely: quadratic regression of $(S_{T_2} - K)^+$ over S_{T_1} and $S_{T_1}^2$ approximates Black & Scholes' formula very decently in simulated example



- Remarkable how the algorithm manages to extract Black & Scholes' pattern from noisy data
- Run it yourself, [reval.xlsx](#) on the repo [GitHub.com/aSavine/CompFinLecture/RegressionSheet/](https://github.com/aSavine/CompFinLecture/RegressionSheet/)
- Solution remains unique and analytic
- Mathematical guarantee:
 - 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*
- Not limited to polynomials: same arguments apply to other bases like Fourier (harmonic) basis, spline basis, and many more

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: $n_1 = \frac{(n_0 + p)!}{n_0! p!} - 1$
- Examples:
 - 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: $n_1 = \frac{(n_0 + 3)(n_0 + 2)(n_0 + 1)}{6} - 1$ dimension grows 10 to 286, 100 → 176,851, 1,000 → 167,668,501

Explosion of basis functions

- In the general case, basis is not necessary polynomial
- Other possibilities include Fourier basis, spline basis and many more
- Number of basis functions always explodes when dimension grows

- Intuitively: we fix basis functions irrespective of data set
- We expect our set of fixed basis functions to perform well with *all* datasets
- So we need a large number basis functions

- Explosion of basis functions is a computational problem: prediction and fitting become prohibitively expensive
- Another problem with a large number of parameters is overfitting

Overfitting and the variance-bias trade-off

- 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
 - This is called “overfitting” and relates to “variance” because the estimated model strongly depends on the dataset
 - Variance vanishes when the size of the dataset grows
- Conversely, with too few parameters, you may fail to fit data
 - This is called “bias” because the estimated model fails to correctly represent data even asymptotically, when the dataset grows
 - Trying to fit a linear model to learn call prices (a few slides back) is one example of bias
- This is all properly formalized as the “variance-bias trade-off”
 - See for instance Bishop’s [Pattern Recognition and Machine Learning](#)
 - Or summary on Quora:
[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](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 to mitigate overfitting, called “regularization”
See **Stanford’s Machine Learning class on Coursera** or Bishop’s book
- The most obvious and effective regularization is to increase the size of the training set
 - Tule of thumb: at least 10 times as many training examples as the number of free parameters (“rule of 10”) – in practice we generally need much more
 - Point is size of training set must increase when parameter set increases, hence, number of training examples also explodes in high dimension

From Bishop (2006)
Pattern Recognition and Machine Learning

1.1. Example: Polynomial Curve Fitting

7

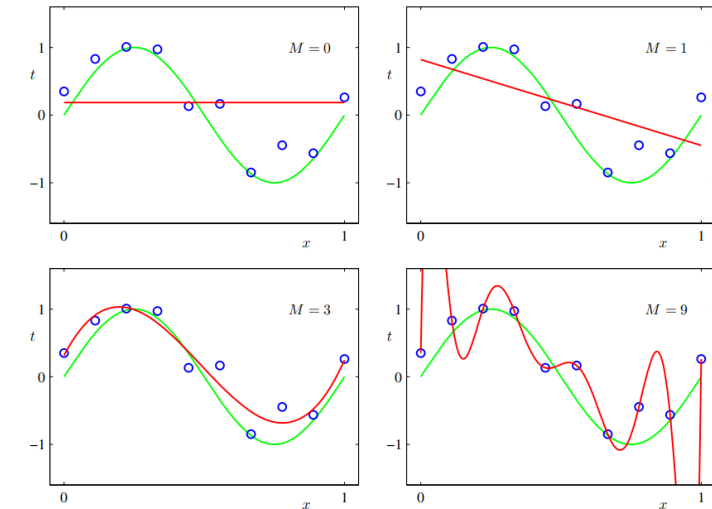
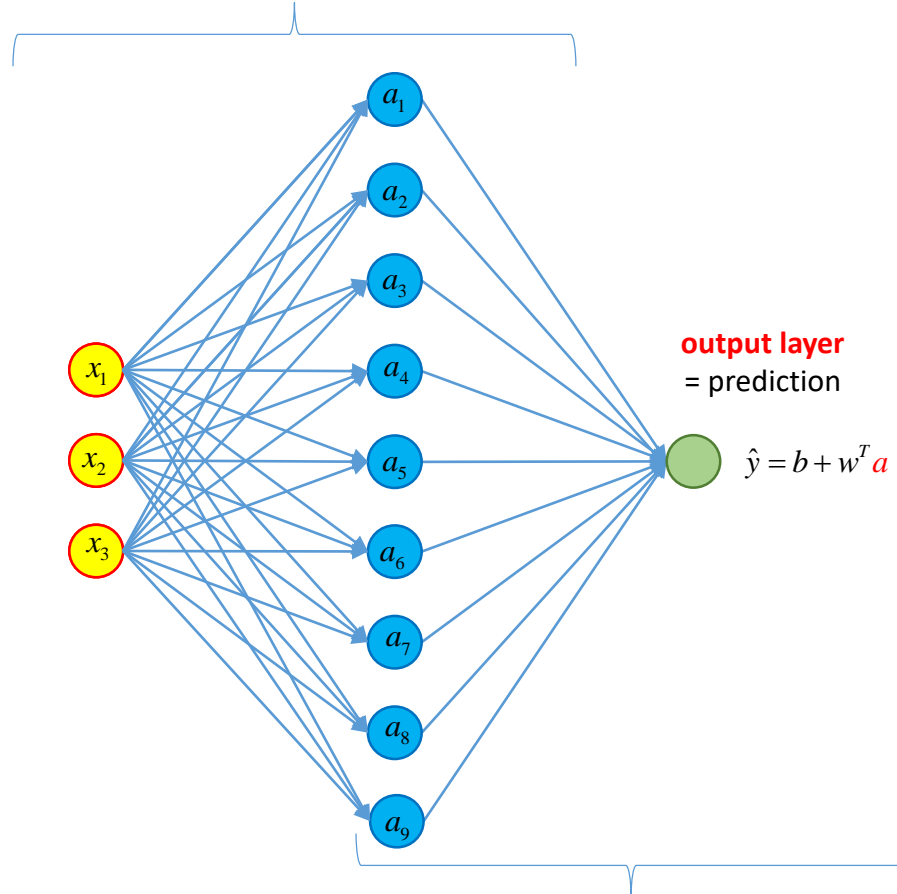


Figure 1.4 Plots of polynomials having various orders M , shown as red curves, fitted to the data set shown in Figure 1.2.

ANN: learn basis functions from data

BASIS FUNCTION REGRESSION

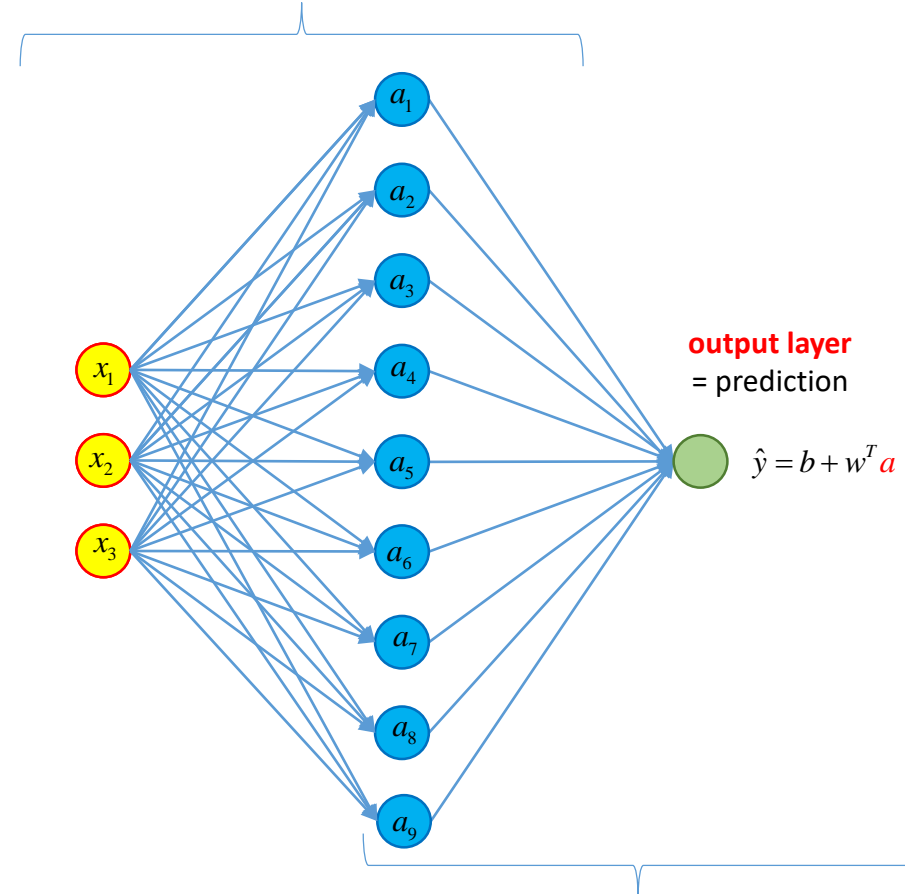
hidden layer = **arbitrary** set of basis functions
pre-processing = not part of the model, no learned parameters



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

Learn features from data

- Instead of a fixed set of basis functions, irrespective of data (hence we need many of those)
- We fix the *number* of basis functions (recall, also called “features”)
- And learn the “best” ones from data
- So we end up with features that adapt to data (hence we don’t need so many of those)
- How do we learn basis functions a from the data?
 - $a_i = \varphi(x; \theta_i)$
 - Parametric family of basis functions: \rightarrow each basis function a_i is a parametric function of *the same vector of inputs x each with different parameters θ_i*
 - \mathcal{G}
 - Learn parameters (along with the weights w and bias b of regression) by minimization of the cost (e.g. sum of squared errors)

Perceptron

- Perceptron = (by far) most common choice of parametric family of basis functions

- Each basis function a_i is some non-linear scalar function g of a linear combination of the x s
In other words, each feature is a different combination of the same x s, activated by a given scalar function g

$$a_i = g(w_i^T x + b_i) \quad g: \text{scalar function called "activation", applied element-wise to the vector} \quad z_i = w_i^T x + b_i$$

- Each $w_i, 1 \leq i \leq n_1$ is a vector of parameters of dimension n_0 , called "weights" and each $b_i, 1 \leq i \leq n_1$ is a real number parameter ("bias")

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

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

- The activation function g
 - Is, in principle, arbitrary
 - *Must be non-linear* or we are back to a linear regression (a linear combination of linear functions is a linear function of the inputs)
 - (Must satisfy some mild technical requirements for the perceptron to represent *any* function)

Perceptron (2)

- Perceptrons have two major benefits:

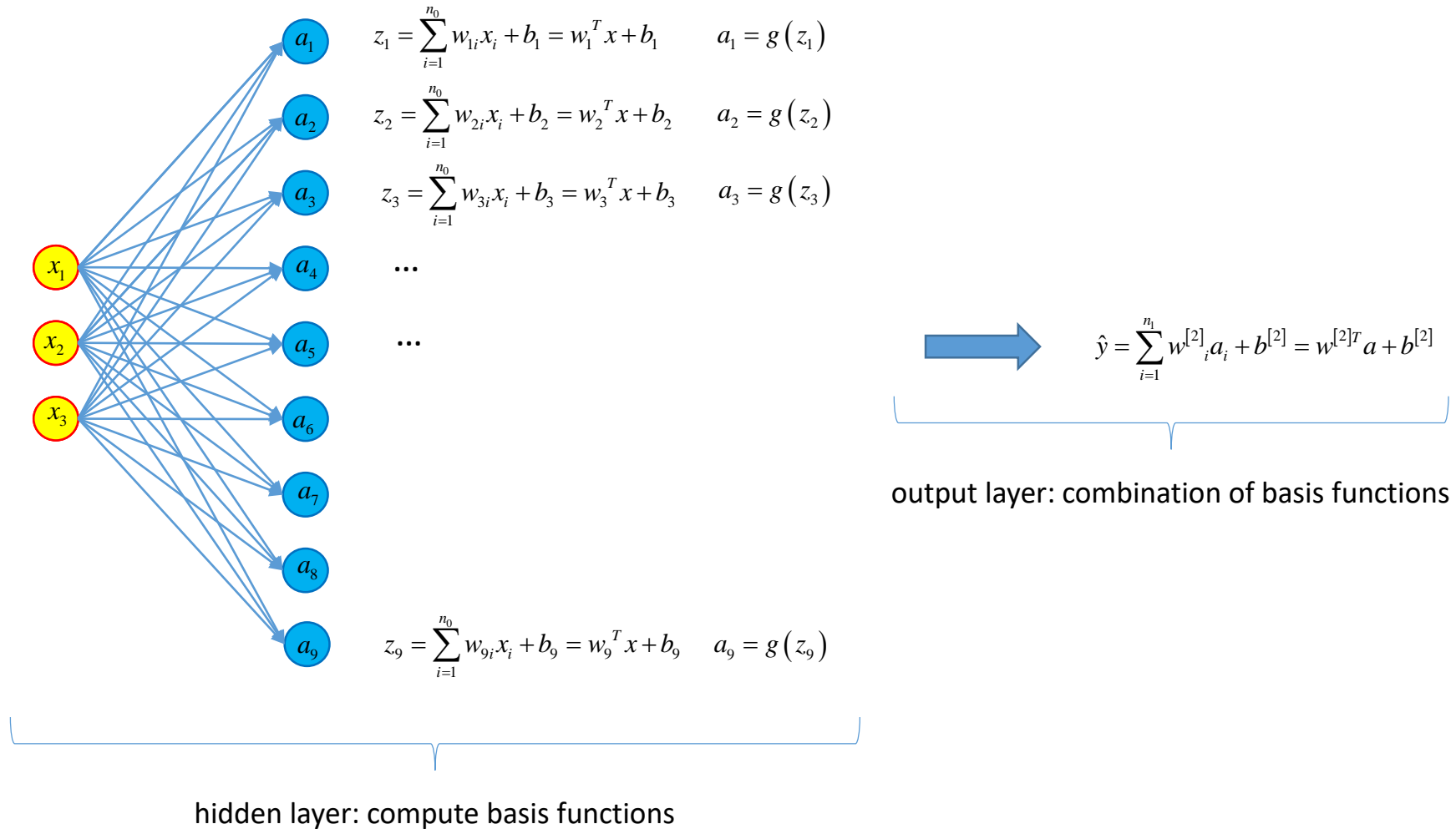
1. Computational

- Perceptron networks are sequences of matrix operations
- (Matrix products followed by element-wise activations)
- Executed very efficiently on modern parallel CPU, GPU and TPU

2. Representational

- Perceptrons can represent *any* function
- (in the asymptotic limit of an infinite number of basis functions, also called “units”)
- Not unlike polynomial or Fourier bases
- This is called the “Universal Representation Theorem”

Perceptron (3)



Prediction with perceptrons

- Prediction in two steps

- Step 1 (hidden layer): feature (=basis function) extraction **inputs x (dim n_0) \rightarrow basis functions a (dim n_1)** $a = g(W^{[1]}x + b^{[1]})$
 - 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 $z^{[1]} = W^{[1]}x + b^{[1]}$
- Step 2: (output layer): linear regression **basis functions a (dim n_1) \rightarrow prediction (real number)** $\hat{y} = w^{[2]}a + b^{[2]}$
 - Parameters: $w^{[2]}$ row vector in dimension n_1 and $b^{[2]}$ real number

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

- For layers $l = 0$ and 1

$$a^{[0]} = x \quad \text{input layer} \quad \text{dim } n_0$$

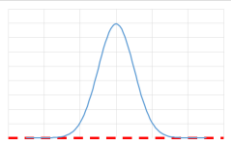
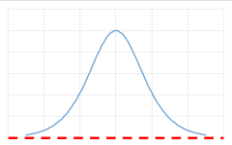
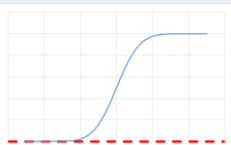
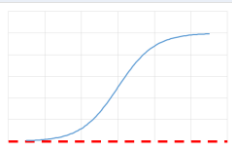
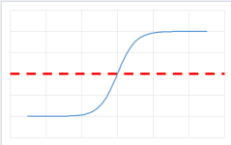
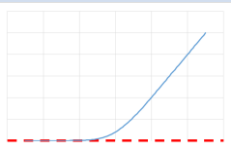
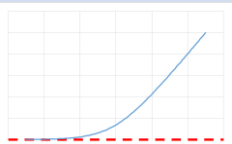
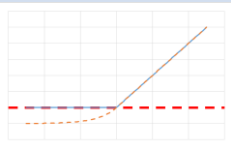
- With the notations $a^{[1]} = a$ hidden layer dim n_1 and parameters $W^{[l]}$ of dim $n_l \times n_{l-1}$ and $b^{[l]}$ of dim n_l for $l = 1, 2$

$$\hat{y} = a^{[2]} \quad \text{output layer} \quad \text{dim } n_2 = 1$$

$$g^{[1]} = g, g^{[2]} = id \quad \text{hidden and output activations}$$

Activations

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

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

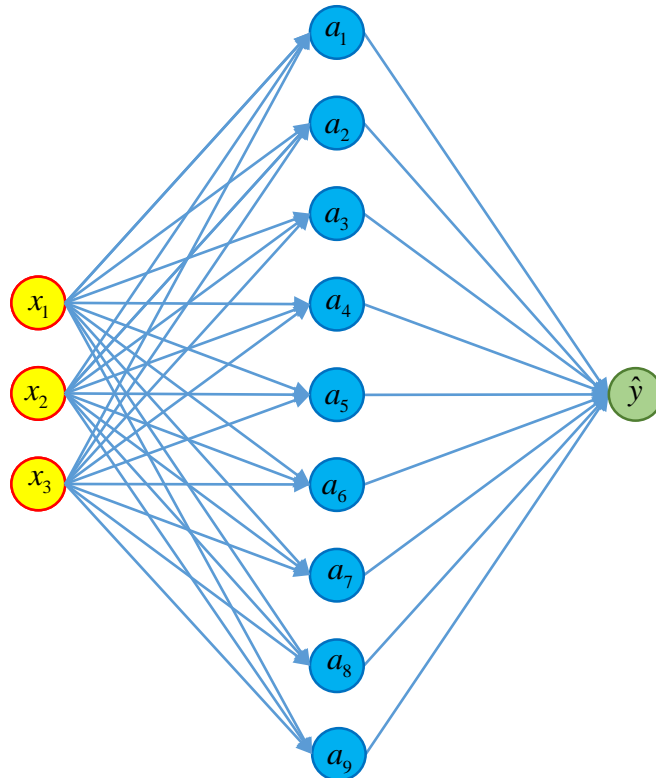
derivative
 integral

*antisymmetric around zero, saturates slowly on the sides

default choice is ELU

Computation graph and complexity

$$\begin{array}{ccc} \text{Input layer } l=0 & \text{Hidden layer } l=1 & \text{Output layer } l=2 \\ a^{[0]} = x & \longrightarrow a^{[1]} = g^{[1]}(W^{[1]}a^{[0]} + b^{[1]}) & \longrightarrow a^{[2]} = g^{[2]}(W^{[2]}a^{[1]} + b^{[2]}) = \hat{y} \\ n_0 \text{ units} & n_1 \text{ units} & n_2 = 1 \text{ units} \end{array}$$



feed-forward equation

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

Complexity

To the leading order (ignoring activations and additions)

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

Hence quadratic complexity $n_1n_0 + n_2n_1$

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

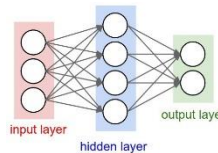
More generally, with L layers, complexity $\sim Ln^2$

Complexity is linear in layers, quadratic in units

Perceptron: summary

- Prediction

- Predict target y by linear regression on a set of n_1 features a of the input x (of dimension n_0): $\hat{y} = w^{[2]T}a + b^{[2]}$
- The features are not fixed, they are parametric functions of x , with learnable parameters: $a = \varphi(x; \mathcal{G})$
- The most common form is the Perceptron:
 - Features are combinations of inputs, activated element-wise: $a = g(W^{[1]}x + b^{[1]})$
 - Computational benefits: matrix operations faster than the sum of their operations on modern hardware
 - Universal Representation Theorem: can asymptotically represent any function
 - Historical legacy and some similarity with how human brain works
- Both feature extraction $x \rightarrow a$ and regression $a \rightarrow y$ are instances of the feed-forward equation: $a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$
- The learnable parameters of the network are all the *connection weights* (linear coefficients) in the extraction and regression steps:
 - Matrices of weights $W^{[l]}$ in dimension $n_l \times n_{l-1}$
 - Vectors of biases $b^{[l]}$ in dimension n_l
- The *computation graph* is of the familiar shape
- The complexity is
 - dominated by matrix by vector products
 - linear in layers, quadratic in units

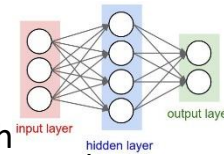


Perceptron: training

“Learn” or “train” = fit the parameters of the perceptron to data

- Given a “training set” of m training examples, each with an input vector x and a known target y (hence, “supervised learning”)
- Set all learnable parameters to minimize prediction errors
- Note that learnable parameters are no longer limited to regression coefficients, they also include feature extraction (hidden layer)
- Learnable parameters are all the connections between units of successive layers

- They are represented by the edges of the computation graph



- Formally, they are the union of matrices W (and biases b) in the feed-forward equations

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

- So we are looking for the solution of

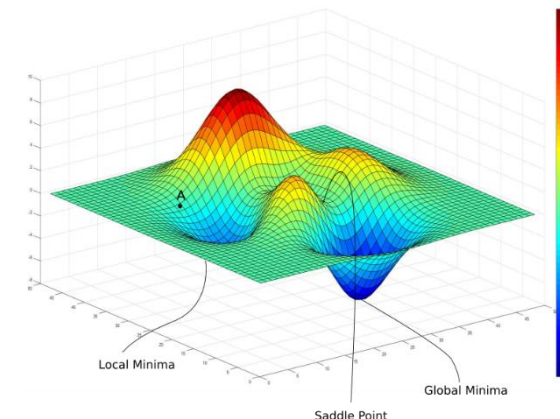
$$W^*, b^* = \arg \min \left\{ C(W, b) = \sum_{i=1}^m \left[\hat{y}(x^{(i)}; W, b) - y^{(i)} \right]^2 \right\}$$

--- unfortunately:

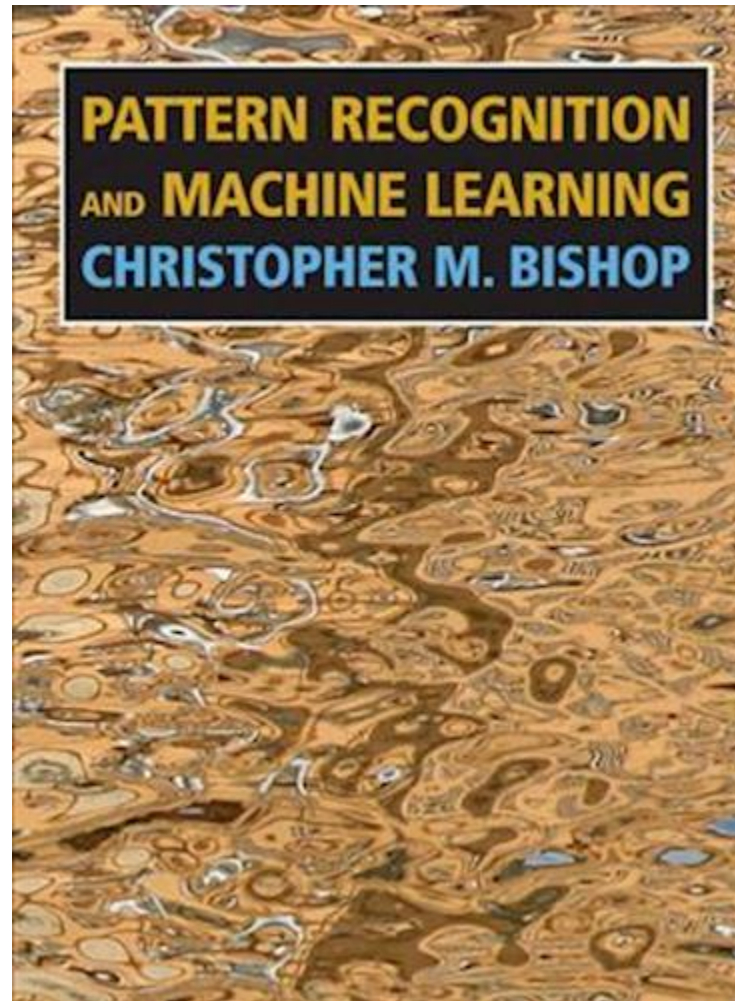
- Or, cancelling gradients:

$$\frac{\partial C}{\partial W} = 0, \frac{\partial C}{\partial b} = 0$$

1. The solution is no longer analytic, a numerical minimization procedure is necessary
2. C is not even convex in W and b , so the gradient equation has many solutions, corresponding to *saddle points* and *local minima*
3. There exist no algorithm guaranteed to converge to the global minimum
The best we have is guarantee of convergence towards a *local* minimum
Incredibly, this is good enough in practice in many cases of practical relevance



More on regression, basis functions and ANNs



Batch prediction

- Single example prediction sequentially applies the feed-forward equation $a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$

from vector of inputs $a^{[0]} = \begin{bmatrix} x \end{bmatrix}$ to output $a^{[L]} = [\hat{y}]$

- To predict *multiple* examples $1, \dots, m$ we could:

- Sequentially run feed-forward equations with $a^{[0](1)} = \begin{bmatrix} x^{(1)} \end{bmatrix}$ to find $a^{[L](1)} = [\hat{y}^{(1)}]$, then with $a^{[0](2)} = \begin{bmatrix} x^{(2)} \end{bmatrix}$ to find $a^{[L](2)} = [\hat{y}^{(2)}]$, etc.

- Or, we could run feed-forward equations once with $a^{[0]} = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(m)} \end{bmatrix}$ (stacking examples in columns)

to find $a^{[L]} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \hat{y}^{(3)}, \dots, \hat{y}^{(m)}]$ (with results also stacked in columns)

Batch prediction (2)

- Feed-forward equations remain absolutely *identical* $a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$
 - a s are now matrices, stacking batches of examples in column vectors
 - Adding a vector v to a matrix M means: add v to all columns of M
This is called “broadcasting” in programming languages
Implemented by default in Python’s numPy
But not in C++
- Why implement batch prediction?
 - Complexity is *identical*:
to process a matrix by matrix product at once or as a sequence of matrix by vector products, column by column, is identical
 - But modern hardware (CPU, GPU, TPU) is highly optimized for SIMD processing
 - SIMD: Single Instruction, Multiple Data: apply *the same* instructions to a lot of *different data* simultaneously
 - On GPU, (reasonably sized) batch prediction takes a time similar to one single prediction!
- In all that follows, for simplicity, we consider single example predictions where a s are vectors
- All equations apply identically with batch predictions with a s matrices with m (batch size) columns
- Practical implementations *always* predict in batches

Deep learning

- Deep learning = ANNs with *multiple* hidden layers --- exactly as before:

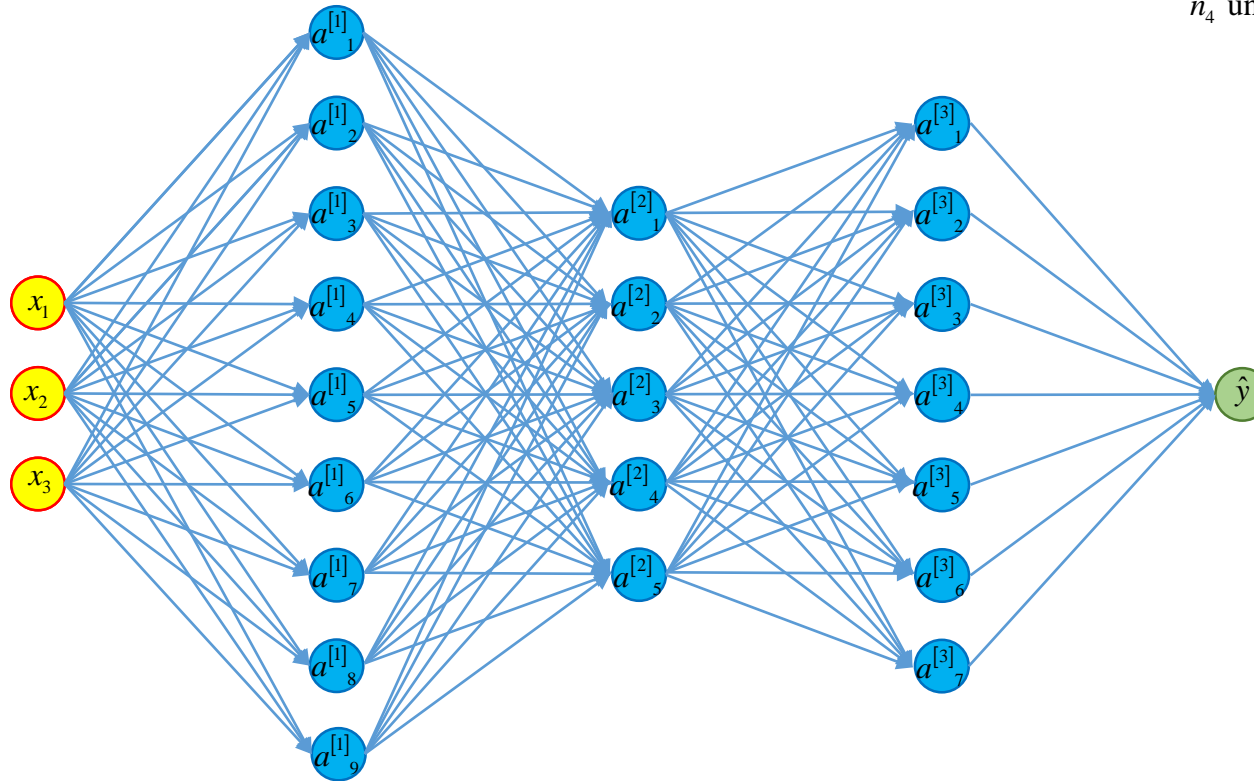
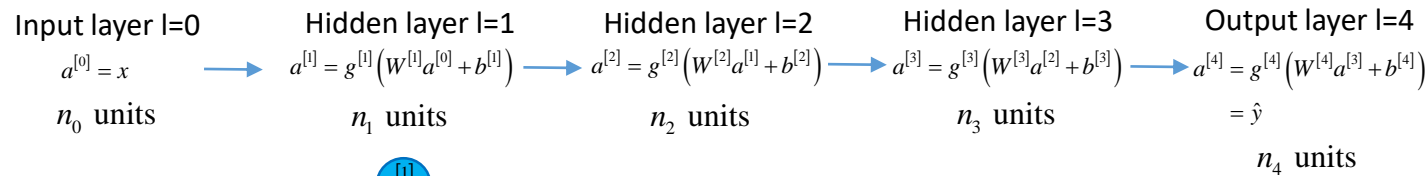
- Input layer is set to the input vector x : $a^{[0]} = \begin{bmatrix} x \end{bmatrix}$

- Successive layers are computed from the previous layer by application of the identical feed-forward equation: $a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$

- Prediction is the last (output) layer: $a^{[L]} = [\hat{y}]$

- The only difference is: we now have multiple hidden layers $1, 2, \dots, L-1$
- So the mechanics are identical to before
- The more interesting question is: why would we want more than one hidden layer?

Multi-layer perceptron (MLP)



feed-forward equation

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

- Prediction: feed-forward equations

$$a^{[0]} = x, a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]}), y = a^{[L]}$$

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

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

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

- Training: find all $W^{[l]}$ and $b^{[l]}$ to minimise cost

$$C = \sum_{i=1}^m c_i, c_i = (\hat{y}^{(i)} - y^{(i)})^2$$

- Use iterative algorithms
compute all D differentials on each iteration

- Key: quickly compute
large number D of differentials

Deep learning: rationale

- Universal Representation Theorem states that all functions are attainable with a single hidden layer
- This is an asymptotic result, as the number of units in the hidden layer increases to infinity
- In practice, we have a limited “computational budget” so we have only so many units
- It has been *empirically* observed, repeatedly, that:
 - Given a fixed computational budget
 - Networks with multiple layers (of fewer units)
 - Perform (exponentially) better than those with one layer (with more units)
 - This is the basis of modern Deep Learning
- This is not (yet?) formally demonstrated
- It is even unclear how to formalize what it is exactly we want to demonstrate?
- But deep learning was repeatedly validated with empirical success
- And we can try to explain why with heuristics and intuition if not formal maths

Learning by composition

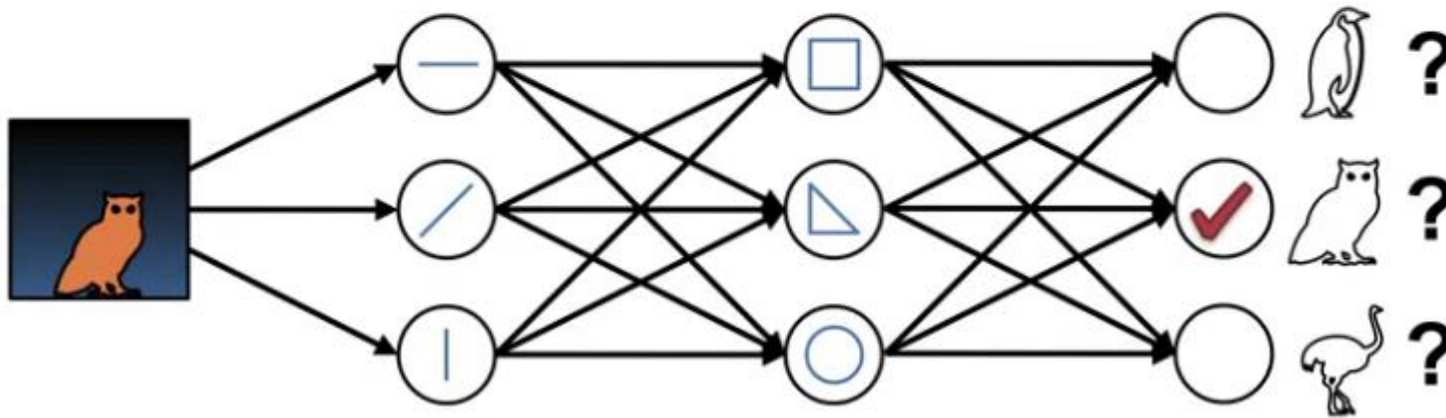
- The *heuristic* rationale is:

Deep Learning introduces function composition, allowing nets to represent a larger space of functions with a limited number of units

- With one hidden layer:
 - Each basis function is an activated combination of the input units
 - So additional hidden units allow the network to learn a larger number of basis functions of the same shape
- With multiple layers:
 - Each unit in a layer is a function of all the units in the previous layer
 - a_1 is a function of $a_0=x$, a_2 is a function of a_1 , etc.
 - In the end, we regress linearly on the penultimate layer a_{L-1} (“basis layer”)
 - Whose units are basis functions of basis functions of basis functions ... of the inputs
 - So, instead of simply increasing the number of possible combinations of the inputs
 - Deep nets learn to *compose* functions to find optimal basis functions for regression
 - Composition allows them to attain a larger, richer space of functions with a limited overall number of units
 - Hence, giving nets a higher representational power for a fixed computational budget

Example : image recognition by deep learning

Compositional features



- Image from Coursera's

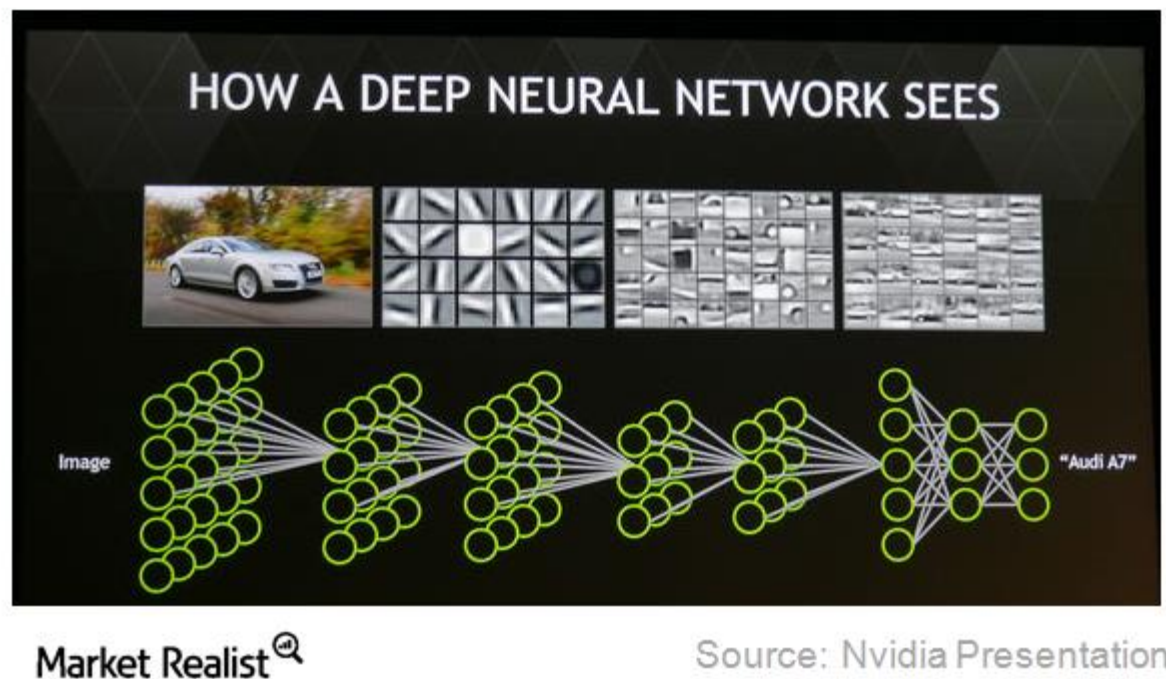
~stellar~

Reinforcement Learning Spec

by University of Alberta

- First layer(s) activate modular, reusable basic shapes: horizontal, vertical, diagonal edges
- Middle layers combine these features into composite shapes: here, squares, triangles, circles could be eyes, body shape, legs
- Output layer combine these features into objects to be identified

Another (more realistic) example



- Early layers detect low level features: mainly, edges, horizontal, vertical, diagonal...
- Subsequent layers combine them into more complex features: wheels, windshields, doors, windows, etc.
- Late layers combine these into cars
The degree of activation of different cars in the output layer gives the probability of identification

Deep learning heuristics

- Benefits of depth have been repeatedly illustrated by spectacular results, including:
 - Image recognition (with deep “convolutional” nets)
 - Time series prediction (with deep “recurrent” nets)
 - Natural Language Processing, Neural translation and speech recognition
 - And much more, out of scope in this lecture
- But we don’t have a formal demonstration (yet?)
- Such formalism would not only confirm empirical results
- But more importantly: guide neural net architecture and hyper-parameters
- At present, these are set heuristically, and/or from experience:
- For example, for regression problems (like we have in finance), performance sharply improves from 1 to 3-5 hidden layers, stabilizes thereafter
- On the contrary, deep convolutional nets in computer vision typically include hundreds of layers

More deep learning heuristics

- Similarly, training performance is not guaranteed
- Loss is non-convex in weights
- No algorithm exists to find global optimum of non-convex functions, and probably never will
- Empirically, it has been “shown” that local optima are “OK”
 - What does that mean exactly?
 - Deep learning is validated by (undeniable) results, not mathematics
 - Is it good enough? This is an open question
- Many heuristics were designed to better train networks, including:
 - Careful initialization of weights: the quality of a local optimum directly depends on starting point
 - Batch-normalization: normalize mean and standard dev between layers
 - Dropout normalization: randomly drop connections to reduce effective number of parameters
 - Inclusion of momentum and adaptive learning rate on different dimensions when applying gradients
 - Skip layers (more on that later)
 - and much much more

More deep learning heuristics (2)

- None of those multiple heuristics is guaranteed to help formally
 - But they tend to help immensely in practice
 - As for depth, we can't (yet?) formalize their contribution
 - But we can understand it intuitively
-
- This is out of scope in this lecture
 - We refer to Andrew Ng's highly recommended [Deep Learning Specialization](#) on Coursera

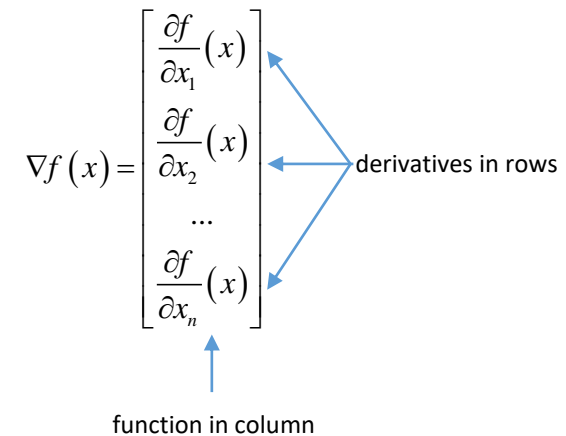
Back-Propagation

Elements of multivariate calculus

- We will derive the differentials of cost functions in neural nets, and learn an algorithm to compute them extremely quickly
- This algorithm, called “back-propagation”, or often “back-prop”, is what enables deep net training in reasonable time
- Deep learning would not be feasible without it
- Further, we will see that back-prop is only a particular case of a more general algorithm, applicable to *any* calculation that computes a scalar result from a (large) vector of inputs and/or parameters called “adjoint differentiation” (AD)
- Modern frameworks apply back-prop and AD automatically, behind the scenes
- In its most general form, this is called AAD: the first A stands for Automatic
- Since practical neural nets are sequences of matrix operations: matrix product and element-wise activation
- We have to apply matrix differential calculus
- Hence, we (briefly) introduce a few results we need to derive basic back-prop, for a more complete reference, see for instance:

Gradients

- Consider a scalar function of a vector: $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- We call *gradient of f on x* and denote $\nabla f(x)$ the *column* vector of its derivatives:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \frac{\partial f}{\partial x_2}(x) \\ \dots \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix}$$


derivatives in rows

function in column

Gradients: key properties

1. Gradient cancels on minima and maxima

- Reciprocal is *not* true: zero-gradient points may be global optima, local optima or saddle points
- Only for convex (concave) functions, zero-gradient points attain global minimum (maximum)

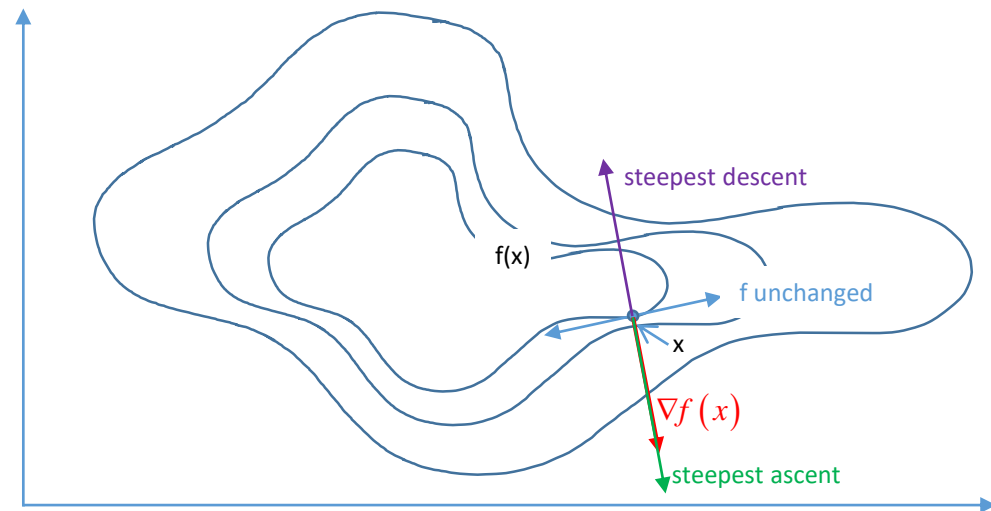
2. The gradient vector is the direction where the slope of f is steepest

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

Hence, for a fixed step size $\|\Delta x\|$:

- steepest ascent is in the direction of $\nabla f(x)$ $\cos=1$
- steepest descent is in the direction of $-\nabla f(x)$ $\cos=-1$
- f is unchanged moving orthogonally to $\nabla f(x)$ $\cos=0$

- This two properties form the basis of a fundamental optimization algorithm: gradient descent
 - Repeatedly move by small steps (called “learning rate”) in the direction of steepest ascent/descent given by gradient until gradient cancels
 - Guaranteed to converge to global optimum for convex functions local optimum or saddle point otherwise given technical conditions on progressively shrinking learning rate



Matrix gradients

- The gradient of $f(x)$ has the same shape $(n \times 1)$ as x
- We want the same property for scalar functions of matrices: $f: \mathbb{R}^{n \times p} \rightarrow \mathbb{R}$
- We denote $\frac{\partial f(M)}{\partial M}$ the matrix of its derivatives, of the same shape $(n \times p)$ as M

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

Jacobians

- Consider a vector function of a vector (or a set of p functions of x , x being of dim n) $f : \mathbb{R}^n \rightarrow \mathbb{R}^p, f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \dots \\ f_p(x) \end{bmatrix}$
- The matrix $\frac{\partial f(x)}{\partial x}$ of the derivatives of the p functions to the n inputs is called *Jacobian* matrix

- Conventionally, Jacobians are often written with functions in rows and derivatives in columns:

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \dots & \frac{\partial f_2(x)}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_p(x)}{\partial x_1} & \frac{\partial f_p(x)}{\partial x_2} & \dots & \frac{\partial f_p(x)}{\partial x_n} \end{bmatrix}$$

functions in rows

derivatives in columns

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

- In this case, $\frac{\partial f(x)}{\partial x}$ is of shape $(p \times n)$

- And you can check that the chain rule is written: $h(x) = f[g(x)] : \mathbb{R}^n \rightarrow \mathbb{R}^q$ (outer function first)

$$\Rightarrow \underbrace{\frac{\partial h(x)}{\partial x}}_{q \times n} = \underbrace{\frac{\partial f(x)}{\partial g(x)}}_{q \times p} \underbrace{\frac{\partial g(x)}{\partial x}}_{p \times n}$$

Jacobians the other way

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

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_1} & \dots & \frac{\partial f_p(x)}{\partial x_1} \\ \frac{\partial f_1(x)}{\partial x_2} & \frac{\partial f_2(x)}{\partial x_2} & \dots & \frac{\partial f_p(x)}{\partial x_2} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_1(x)}{\partial x_n} & \frac{\partial f_2(x)}{\partial x_n} & \dots & \frac{\partial f_p(x)}{\partial x_n} \end{bmatrix}$$

derivatives in rows

functions in columns

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

- In this case, $\frac{\partial f(x)}{\partial x}$ is of shape $(n \times p)$

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

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

- And you can check that the chain rule is written: $h(x) = f[g(x)]: \mathbb{R}^n \rightarrow \mathbb{R}^q$ (inner function first)

$$\Rightarrow \underbrace{\frac{\partial h(x)}{\partial x}}_{n \times q} = \underbrace{\frac{\partial g(x)}{\partial x}}_{n \times p} \underbrace{\frac{\partial f(x)}{\partial g(x)}}_{p \times q}$$

Jacobians of linear functions

- If $f: \mathbb{R}^n \rightarrow \mathbb{R}^p$ is linear, i.e. $f(x) = Ax$ where A is a matrix of shape $(p \times n)$
- Then, you can convince yourself that its Jacobian matrix (with derivatives in rows) is the constant:

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

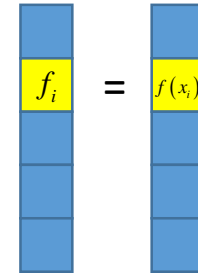
and not A (which is maybe why people like Jacobians the other way around...)

The diagram illustrates the linear function $f(x) = Ax$. It shows a vertical blue column vector with 5 cells, the second of which is yellow and labeled f_i . This is equal to a 5x3 blue grid with its second row yellow and labeled A_i , multiplied by a vertical orange column vector with 3 cells labeled x .

Jacobians of element-wise functions

- Consider a scalar function: $f: \mathbb{R} \rightarrow \mathbb{R}$

- We apply f element-wise to a vector x of dim n to obtain another vector of dim n : $f(x) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \dots \\ f(x_n) \end{bmatrix}$



- Then, the Jacobian of f is the $n \times n$ diagonal matrix of element-wise derivatives:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} = f'(x_1) & \frac{\partial f_2}{\partial x_1} = 0 & \dots 0 \dots & 0 \\ \frac{\partial f_1}{\partial x_2} = 0 & \frac{\partial f_2}{\partial x_2} = f'(x_2) & \dots 0 \dots & 0 \\ 0 & 0 & \dots \frac{\partial f_i}{\partial x_i} = f'(x_i) \dots & \\ 0 & 0 & \dots 0 \dots & \frac{\partial f_n}{\partial x_n} = f'(x_n) \end{bmatrix} = \text{diag}[f'(x)]$$

Vector functions of matrices

- Consider now a vector function of a matrix: $f : \mathbb{R}^{n \times p} \rightarrow \mathbb{R}^q$
- In principle, its “derivative” is a 3D tensor of shape $(n \times p \times q)$: $\frac{\partial f}{\partial M} = \left(\frac{\partial f_k}{\partial M_{ij}} \right)$
- In the linear case where $f(M) = Mv$, and v is a constant vector of size $p=q$

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

$$\begin{bmatrix} \vdots \\ f_i \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots & M_i & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} * \begin{bmatrix} \vdots \\ v \\ \vdots \end{bmatrix}$$

How to train your network

- **Recap:** we have an ANN of the MLP form implementing the following sequence of feed-forward computations:

- Starting with inputs $a^{[0]} = x \in \mathbb{R}^{n_0}$
- Sequentially compute $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$ then $a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l}$ (for $l = 1, 2, \dots, L$: feed-forward equations)
- To make predictions $\hat{y} = a^{[L]} \in \mathbb{R}$

$$a^{[0]} = x \in \mathbb{R}^{n_0} \xrightarrow{FF} a^{[1]} \in \mathbb{R}^{n_1} \xrightarrow{FF} a^{[2]} \in \mathbb{R}^{n_2} \xrightarrow{FF} \dots \xrightarrow{FF} a^{[L]} \in \mathbb{R}^{n_L} \dots \xrightarrow{FF} \hat{y} = a^{[L]} \in \mathbb{R}$$

$FF : \text{combination} + \text{activation}$

- We have a *training set* of m independent samples $x^{(m)}$ for the inputs x (each in dim n_0), along with corresponding *labels* $y^{(m)}$
- We want to train the ANN so it outputs the “best” predictions for y knowing x , that is $\hat{y} = E[y|x]$
- By definition, conditional expectation is the function closest in the sense of least squares: $E[y|x] = \arg \min_f E[(y - f(x))^2]$
- And we know/have seen that:
 - (given enough height and depth) the ANN can represent *any* function f to arbitrary accuracy by adjusting its parameters W and b
 - (given enough training examples) $E[(y - f(x))^2] \approx \frac{1}{m} \sum_{i=1}^m [y^{(i)} - f(x^{(i)})]^2 = \frac{1}{m} \sum_{i=1}^m c_i$ where $c_i = [y^{(i)} - f(x^{(i)})]^2$ is the loss of example i
- Therefore, we must find the parameters W^* and b^* minimizing mean square prediction error on the training set: $W^*, b^* = \arg \min C(W, b) = \frac{1}{m} \sum_{i=1}^m c(W, b)$

Bad news good news

- There is no analytical solution for the optimal weights (unlike with linear regression)
- The cost C is a non-convex function of the weights W biases b (in what follows we just call “weights” all the parameters)
- An algorithm guaranteed to find the minimum of a non-convex function does not exist, and probably never will
- So, the problem of training ANNs cannot, in theory, be resolved
- This realization caused mathematicians to step away from deep learning in the 1990s
- Deep learning was revived recently by engineers along the lines of “this is not supposed to work, but let’s try anyway and see how we can take it”
- And we all know it did work and produced spectacular results
- The starting point is gradient descent:
 - Choose initial weights randomly
 - Take repeated steps, changing weights
 - by a small amount called “learning rate”
 - along the direction of steepest descent, which we know is the direction opposite to the gradient
- Many heuristic improvements were made to improve and accelerate learning, but none is guaranteed to converge
- However, we understand the intuitions and take comfort in that they work remarkably well in a multitude of contexts

Differentials by finite differences

- Optimization algorithms
 - All learning algorithms start with a random guess and make steps opposite to the current gradient
 - Our job is compute the gradients $\partial C / \partial W$ and $\partial C / \partial b$ of the cost C , repeatedly
 - Our goal is to do it quickly
- Gradients by finite differences (FD):
 - First compute the cost C_0 with the current weights, making predictions for all training examples
 - To compute the sensitivity of the cost to each scalar weight $w_{ij}^{[l]}$ or bias $b_i^{[l]}$:
 - Bump the parameter by a small amount ε
 - Re-compute the cost C_1 , repeating feed-forward calculations for all training examples
 - The derivative to the bumped weight is approx. $(C_1 - C_0) / \varepsilon$
- Repeatedly re-compute predictions, bumping one weight at a time
- Linear complexity in the (potentially immense) number of weights, therefore not viable
- But some very desirable properties

Automatic derivatives with FD

- Implementation is straightforward with little scope for error
- Importantly, FD differentiation is **automatic**
 - Developers only write prediction (feed-forward) code
 - FD computes differentials automatically by calling the feed-forward code repeatedly
 - Developers don't need to write any differentiation code
- More importantly, FD automatically **synchronises** with modifications to feed-forward code
 - More complex layers: convolutional, recurrent, ...
 - Tricks of the trade to train deep nets faster and better:
 - Dropout: randomly drop connections between units
 - Batch Norm: normalize the mean and variance on all layers
 - And many more
 - See e.g. Coursera's [Deep Learning Spec](#)
 - All of these modify feed-forward equations and code
 - Manual differentiation code would have to be re-written and re-tested accordingly, every time
 - This is painful and prone to error
 - Automatic differentiation (obviously) guarantees effortless consistency

Analytic differentials in MLPs

- FD is a general, automatic differentiation algorithm but it is too slow for training ANNs
- Lets see if we can do better with analytic differentiation
- We will derive analytic differentials for linear regression first
- Then, generalize to shallow neural nets, with a single hidden layer
- Finally, generalize to vanilla deep nets
- Later, we will generalize to any kind of network and even any (scalar) computation
- And see how to compute those differentials automatically, behind the scenes, like with FD, but an order of magnitude faster

Analytic differentials in MLPs: Framing

- We know all the current parameters $w_{ij}^{[l]}$ and $b_i^{[l]}$ or in matrix form $W^{[l]}$ and $b^{[l]}$
- The activation function(s) $g^{[l]}$ are fixed
- We call them “hyper-parameters” or non-learnable parameters
- We also know their derivatives $g^{[l]'} \cdot$ -- recall these are scalar functions, applied element-wise
- Prediction $\hat{y}^{(i)} = a^{[L](i)}$ is obtained from example $x^{(i)} = a^{[0](i)}$
- By a sequence of feed-forward equations: combination + activation $z^{[l](i)} = W^{[l]}a^{[l-1](i)} + b^{[l]} \rightarrow a^{[l](i)} = g^{[l]}(z^{[l](i)})$
- The *loss* for example i is the (squared) prediction error $c_i = [\hat{y}^{(i)} - y^{(i)}]^2$
- The *cost* for the training set is $C = \frac{1}{m} \sum_{i=1}^m c_i$
- Our goal is to compute all the $\frac{\partial C}{\partial w_{ij}^{[l]}}$ and $\frac{\partial C}{\partial b_i^{[l]}}$ or in matrix form $\frac{\partial C}{\partial W^{[l]}}$ and $\frac{\partial C}{\partial b^{[l]}}$

Differentials for single example or batch

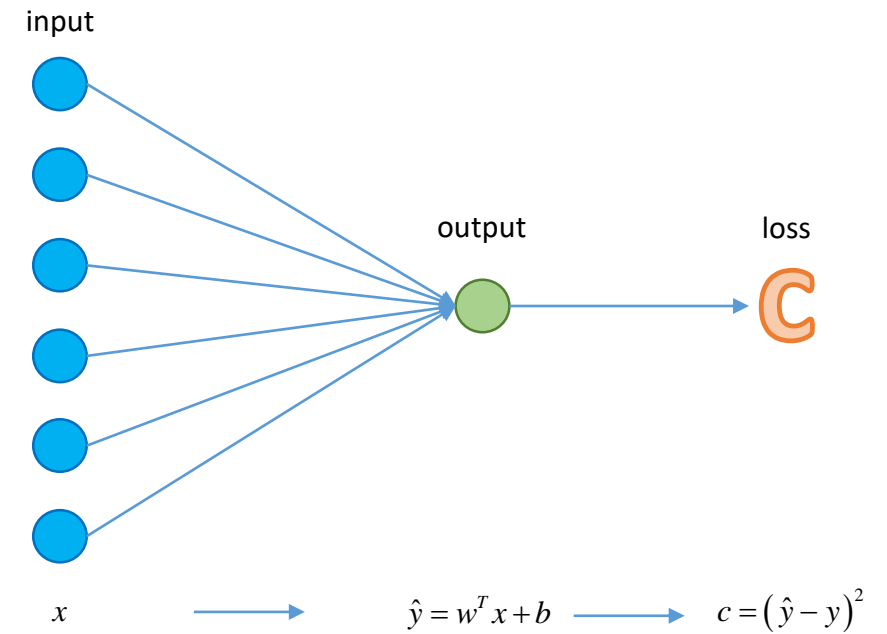
- We will compute $\frac{\partial c_i}{\partial W^{[l]}}$ and $\frac{\partial c_i}{\partial b^{[l]}}$
- Then $\frac{\partial C}{\partial W^{[l]}} = \frac{1}{m} \sum_{i=1}^m \frac{\partial c_i}{\partial W^{[l]}}$ and $\frac{\partial C}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^m \frac{\partial c_i}{\partial b^{[l]}}$
- The complexity is the same
 - To compute differentials of losses of single examples and sum them up
 - Or compute differentials of a batch of examples at once
- However, batch computation is more efficient on modern hardware
- Practical implementation are always batch implementations
- We start with a single example
- And drop the example index (i) to simplify notations

Differentials: linear regression

Comments

1. Derivatives are computed in the reverse order

- Parameter derivatives all depend on $\frac{\partial c}{\partial \hat{y}}$
- Hence, we first compute $\frac{\partial c}{\partial \hat{y}}$, then parameter derivatives
- To compute derivatives, we must first know
 - So we must make a prediction (“forward pass”)
 - *before* we compute derivatives (“reverse pass”)



chain rule

$$\begin{aligned}\frac{\partial c}{\partial w} &= \frac{\partial \hat{y}}{\partial w} \frac{\partial c}{\partial \hat{y}} = x \frac{\partial c}{\partial \hat{y}} \\ \frac{\partial c}{\partial b} &= \frac{\partial \hat{y}}{\partial b} \frac{\partial c}{\partial \hat{y}} = \frac{\partial c}{\partial \hat{y}} \\ \frac{\partial c}{\partial x} &= \frac{\partial \hat{y}}{\partial x} \frac{\partial c}{\partial \hat{y}} = w \frac{\partial c}{\partial \hat{y}}\end{aligned}$$
$$\frac{\partial c}{\partial \hat{y}} = 2(\hat{y} - y)$$

Differentials: shallow net

Comments

1. We compute derivatives in the reverse order, systematically reusing previous computations

2. We reuse $\frac{\partial c}{\partial a}$ known from difference of regression layer

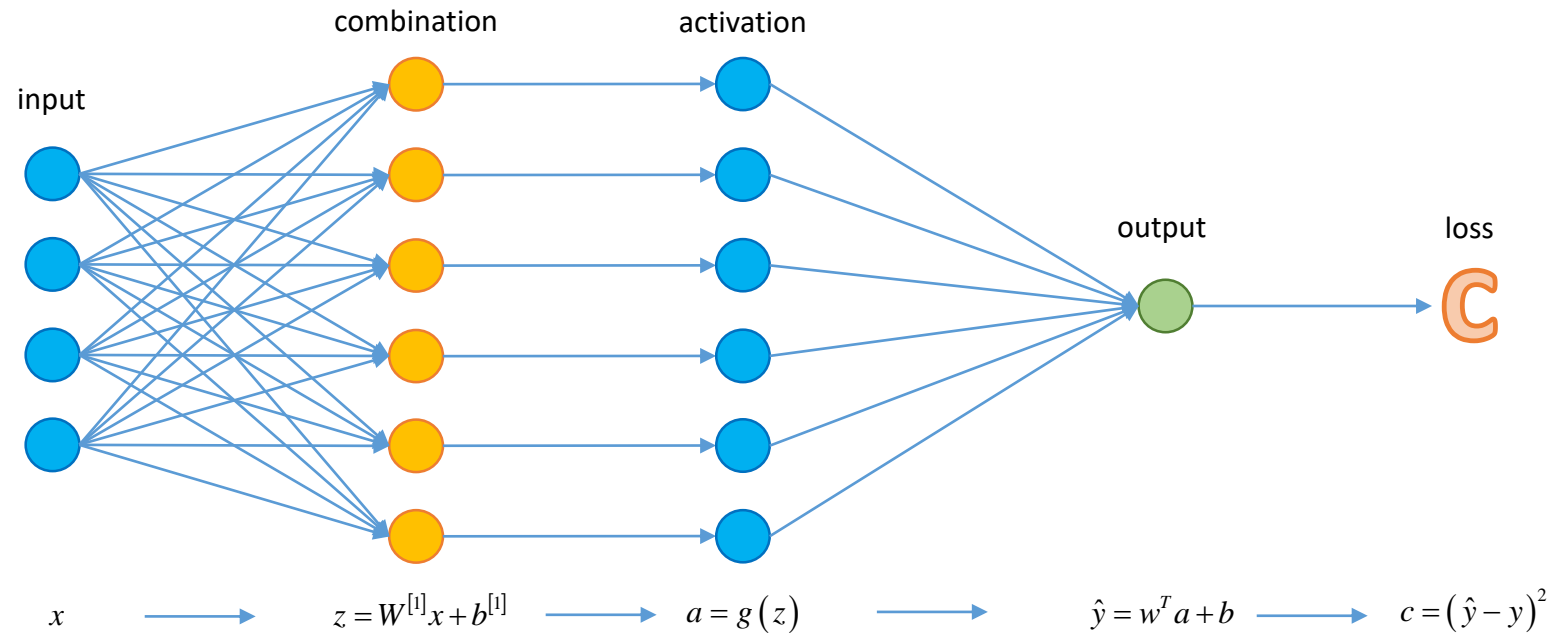
3. The chain rule gives us $\frac{\partial c}{\partial z} = \frac{\partial a}{\partial z} \frac{\partial c}{\partial a}$

4. And then:

- $\frac{\partial c}{\partial W^{[1]}} = \frac{\partial c}{\partial z} x^T$ (with a bit of effort)

- $\frac{\partial c}{\partial b^{[1]}} = \frac{\partial c}{\partial z}$ and $\frac{\partial c}{\partial x} = W^{[1]T} \frac{\partial c}{\partial z}$

5. We need values of z and a
So in the prior forward pass, we must remember all intermediate results



chain rule

$$\frac{\partial c}{\partial W_i^{[1]}} = \frac{\partial z_i}{\partial W_i^{[1]}} \frac{\partial c}{\partial z_i}$$

$$= x^T \frac{\partial c}{\partial z_i} = \frac{\partial c}{\partial z_i} x^T \Rightarrow \frac{\partial c}{\partial W^{[1]}} = \frac{\partial c}{\partial z} x^T$$

$$\frac{\partial c}{\partial b^{[1]}} = \frac{\partial z}{\partial b^{[1]}} \frac{\partial c}{\partial z} = \frac{\partial c}{\partial z}$$

$$\frac{\partial c}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial c}{\partial z} = W^{[1]T} \frac{\partial c}{\partial z}$$

$$\frac{\partial c}{\partial z} = \frac{\partial a}{\partial z} \frac{\partial c}{\partial a}$$

$$= \text{diag}[g'(z)] \frac{\partial c}{\partial a}$$

$$= g'(z) \otimes \frac{\partial c}{\partial a}$$

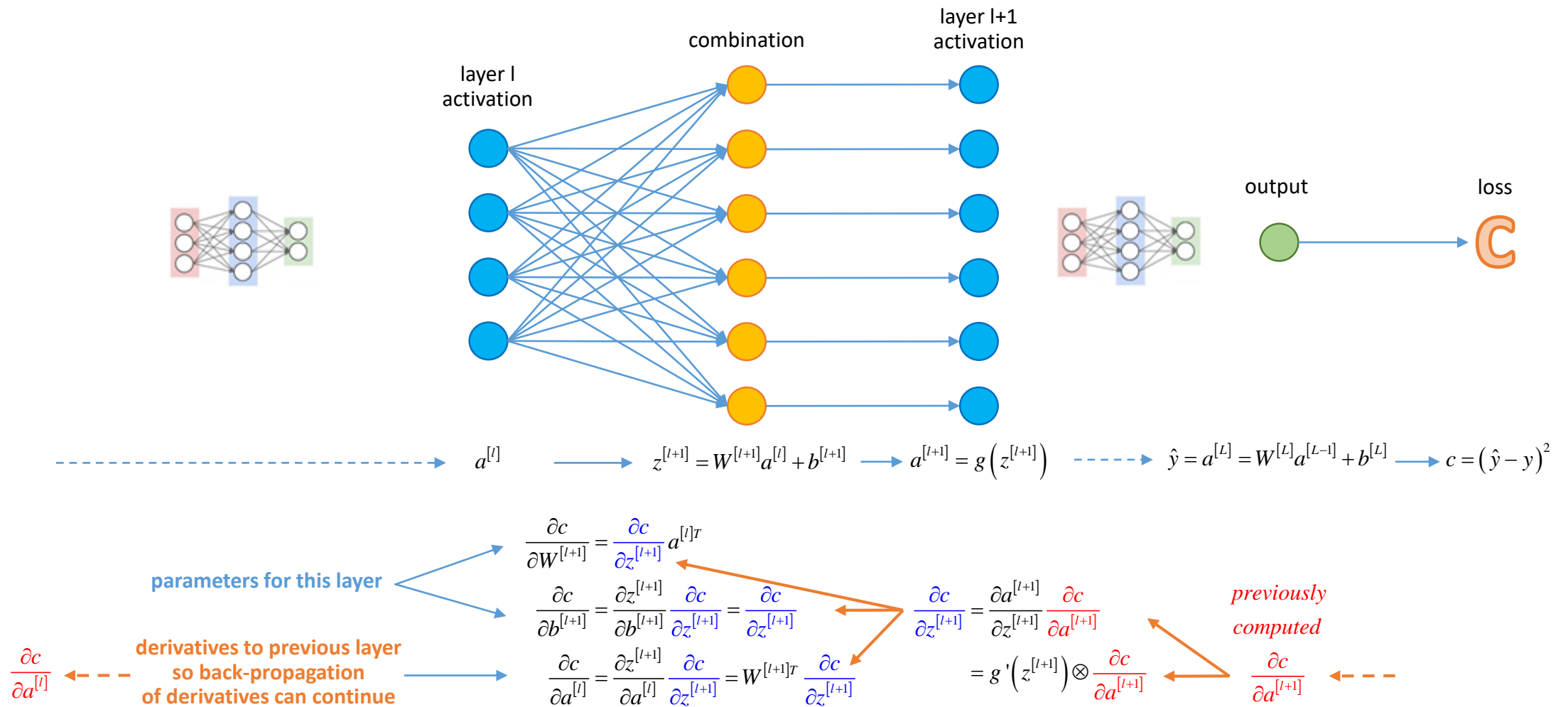
\otimes = element-wise \times

previously
computed

$$\frac{\partial c}{\partial w}, \frac{\partial c}{\partial b}$$

$$\frac{\partial c}{\partial a}$$

Differentials: deep net



Back-prop: single example, vanilla net

1. Forward pass: predict and store

a. Start with input layer $a^{[0]} = x$

b. Recursively compute (with feed-forward equations) and store combinations z and activations a of layer l , from $l=1$ to $l=L$

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \longrightarrow a^{[l]} = g^{[l]}(z^{[l]}) \quad \text{note activation } g \text{ may be different on every layer, for instance } g=\text{id} \text{ on the output layer}$$

c. Pick prediction $\hat{y} = a^{[L]}$ and loss $c = (\hat{y} - y)^2$

2. Backward pass: compute derivatives of loss c in the reverse order

a. Start with *output* layer $\frac{\partial c}{\partial \hat{y}} = \frac{\partial c}{\partial a^{[L]}} = 2(\hat{y} - y)$

b. Recursively compute for layer l , from $l=L$ to $l=1$, hence in the *reverse* order, knowing $\frac{\partial c}{\partial a^{[l]}}$

a. Derivatives of c to parameters of layer l : $\frac{\partial c}{\partial z^{[l]}} = \frac{\partial a^{[l]}}{\partial z^{[l]}} \frac{\partial c}{\partial a^{[l]}} = g^{[l]'}(z^{[l]}) \otimes \frac{\partial c}{\partial a^{[l]}} \quad \frac{\partial c}{\partial W^{[l]}} = \frac{\partial c}{\partial z^{[l]}} a^{[l-1]T} \quad \frac{\partial c}{\partial b^{[l]}} = \frac{\partial z^{[l]}}{\partial b^{[l]}} \frac{\partial c}{\partial z^{[l]}} = \frac{\partial c}{\partial z^{[l]}}$

b. Derivatives of c to layer $l-1$ so back-prop may proceed: $\frac{\partial c}{\partial a^{[l-1]}} = \frac{\partial z^{[l]}}{\partial a^{[l-1]}} \frac{\partial c}{\partial z^{[l]}} = W^{[l]T} \frac{\partial c}{\partial z^{[l]}}$

Back-prop: efficiency

- Reversing the order
we reuse derivatives to next layer inputs = current layer outputs, to compute derivatives to current layer parameters and inputs
- Therefore, we only differentiate *current layer* operations to produce derivatives
- Recall complexity of forward pass
 - To the leading term, one matrix by vector product (of complexity quadratic in units) per layer
 - Plus, negligible (linear) addition of bias and activation
- Complexity of backward pass:
 - One matrix by vector product (of complexity quadratic in units) for derivatives to previous layer
 - One outer product (also of complexity quadratic in units) for derivatives to weights
 - Plus, negligible (linear) element-wise multiplication and activation derivatives
- Hence, **back-prop computes all (maybe thousands of millions of) derivatives with complexity only 3 times a forward pass!**

Exercise: batch back-prop, vanilla net

- The only way to really get back-prop is re-derive it yourselves
- The exercise is re-derive back-prop in the vanilla net, but for a batch of m examples to leverage modern, parallel hardware:
- Inputs: the m examples stacked in the m columns of the input matrix $a^{[0]} = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(m)} \end{bmatrix}$
- Feed-forward equation: same as before $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \longrightarrow a^{[l]} = g^{[l]}(z^{[l]})$

except z and a are no longer vectors, but matrices with m columns

- Prediction: a row vector in dimension m $\hat{y} = a^{[L]} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \hat{y}^{(3)}, \dots, \hat{y}^{(m)}]$
- Cost: $C = \frac{1}{m}(\hat{y} - y)^T (\hat{y} - y)$

Solution

- Seed: $\frac{\partial C}{\partial \hat{y}} = \frac{\partial C}{\partial a^{[L]}} = \frac{2}{m}(\hat{y} - y)$
- Recursion for $l=L$ to 1:
 - Combination: $\frac{\partial c}{\partial z^{[l]}} = g^{[l]'}(z^{[l]}) \otimes \frac{\partial c}{\partial a^{[l]}}$
 - Weights: $\frac{\partial c}{\partial W^{[l]}} = \frac{\partial c}{\partial z^{[l]}} a^{[l-1]T}$
 - Biases: $\frac{\partial c}{\partial b^{[l]}} = \frac{\partial c}{\partial z^{[l]}} \mathbf{1}_m$
 - Recursion: $\frac{\partial c}{\partial a^{[l-1]}} = W^{[l]T} \frac{\partial c}{\partial z^{[l]}}$

Generalization

- We have so far studied “vanilla” neural nets
 - Compute current layer l from previous layer $l-1$ in two matrix operations (or *ops*, following TensorFlow nomenclature):
a combination $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$ followed by an activation $a^{[l]} = g^{[l]}(z^{[l]})$
 - Each layer admits two sets of parameters, or connection weights: the matrix $W^{[l]}$ and the vector $b^{[l]}$
 - The output layer $\hat{y} = a^{[L]}$ is a real number (the prediction)
 - The loss is the square prediction error: $c = (\hat{y} - y)^2 = (a^{[L]} - y)^2$
- Excluded components present in real-world ANNs like:
 - Improvements in training speed and stability like batch-norm or dropout → new ops, more than 2 per layer
 - Convolutional or recurrent nets heavily used in computer vision or natural language processing → yet different ops
 - Classification nets → $\hat{y} = a^{[L]}$ is a vector and $c = c(\hat{y}) = c(a^{[L]})$ is an arbitrary scalar function

General ANNs

We generalize the formalism of ANNs as follows:

- Starting with the vector $a^{[0]} = x$ in the input layer, as previously
- Apply a sequence of L vector-to-vector transformations (or ops) $\phi^{[l]}$, each parameterized by its vector of parameters $g^{[l]}$

$$a^{[l]} = \phi^{[l]}(a^{[l-1]}; g^{[l]})$$

- The ops $\phi^{[l]}$ in the sequence are arbitrary, but every transformation must be a function of the previous vector $a^{[l-1]}$ only (for now)
 - We call **ops** these vector-to-vector transformations $\phi^{[l]}$ --- the name comes from TensorFlow
 - Ops are algebraic building blocks: product by matrix, element-wise scalar function application, convolution, etc.
 - Jacobians $\frac{\partial \phi^{[l]}}{\partial a^{[l-1]}}$ and $\frac{\partial \phi^{[l]}}{\partial g^{[l]}}$ of all ops are known --- TensorFlow ops require code to produce Jacobians
 - l is no longer the index of the *layer*, but the index of the *op* $\phi^{[l]}$
For example, vanilla nets have 2 ops per layer: a combination, followed by an activation – general nets may have more
 - The vector $g^{[l]}$ is the collection of all scalar parameters to op $\phi^{[l]}$, including elements in matrices and vectors
For example, the matrix product step in vanilla nets is parameterized by a matrix, which we represent flattened in a vector
- Pick the prediction vector $a^{[L]}$ and arbitrary (but, importantly, *scalar*) loss $c = c(\hat{y}) = c(a^{[L]})$

General back-prop

Applying the same steps we used to derive back-prop for vanilla nets, we obtain a general back-prop algorithm:

1. Perform a forward pass, store all intermediate results $a^{[l-1]}$
2. Seed with the derivative to the output layer: $\frac{\partial c}{\partial a^{[L]}} = \frac{\partial c}{\partial \hat{y}} = \nabla c(\hat{y})$
3. Back-propagate derivatives with the chain rule, knowing $\frac{\partial c}{\partial a^{[l]}}$
 - a. Parameter derivatives: $\frac{\partial c}{\partial g^{[l]}} = \frac{\partial a^{[l]}}{\partial g^{[l]}} \frac{\partial c}{\partial a^{[l]}} = \frac{\partial \phi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial g^{[l]}} \frac{\partial c}{\partial a^{[l]}}$
 - b. Compute derivatives $\frac{\partial c}{\partial a^{[l-1]}}$ for recursion: $\frac{\partial c}{\partial a^{[l-1]}} = \frac{\partial a^{[l]}}{\partial a^{[l-1]}} \frac{\partial c}{\partial a^{[l]}} = \frac{\partial \phi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial a^{[l-1]}} \frac{\partial c}{\partial a^{[l]}}$

Adjoint

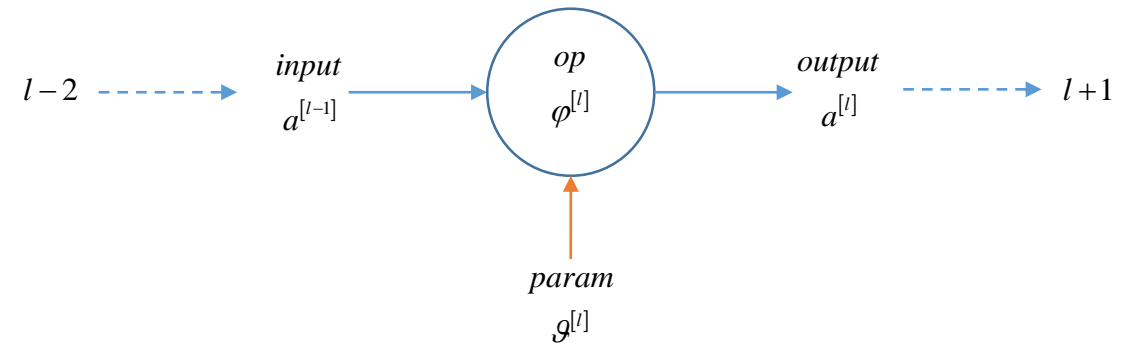
- For a given final *scalar* result c , we conveniently
 - define as the adjoint of some (real number, vector or matrix) x
 - and denote \bar{x}
 - the derivative of c to x : $\bar{x} = \frac{\partial c}{\partial x}$
- Note that the adjoint x is of the same shape of x :
 - If x is a real number, so is its adjoint
 - If x is a vector, its adjoint is a vector of the same size
 - If x is a matrix, its adjoint is a matrix of the same dimensions
- Then, the back-propagation equations may be written more compactly:
 - Parameter derivatives: $\bar{g}^{[l]} \leftarrow \bar{g}^{[l]} + \frac{\partial a^{[l]}}{\partial g^{[l]}} \bar{a}^{[l]}$
 - And recursion: $\bar{a}^{[l-1]} = \frac{\partial a^{[l]}}{\partial a^{[l-1]}} \bar{a}^{[l]}$

Adjoint differentiation

forward pass

- An op transforms input into output
a.k.a a *feed-forward* equation

$$a^{[l]} = \phi^{[l]}(a^{[l-1]}; g^{[l]})$$

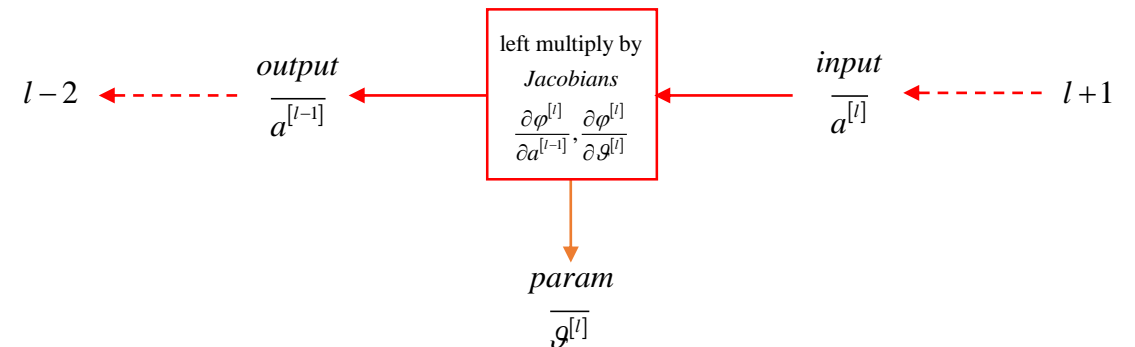


backward pass

- Corresponding *adjoint* equation computes derivatives of *input* from those of *output*
- And we accumulate parameter derivatives on the way

$$\overline{a^{[l-1]}} = \frac{\partial \phi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial a^{[l-1]}} \overline{a^{[l]}}$$

$$\overline{g^{[l]}} = \frac{\partial \phi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial g^{[l]}} \overline{a^{[l]}}$$



Adjoint differentiation (2)

- Performance:
 - For each application of an op in the forward pass
 - We have two matrix by vector multiplications in the backward pass, of quadratic complexity
 - Provided ops are of quadratic complexity in their inputs/outputs (like matrix by vector product)
 - Backward pass computes all derivatives in about 3 times a forward pass
- Manual implementation:
 - The general formulation of back-propagation
 - Gives a “recipe” for writing adjoint code by hand
 - Quite literally, an algorithm for writing code, hinting that the process may be automated
 - The general formulation of back-propagation
 - Seed the adjoint of final result $\overline{a^{[L]}} = \bar{y} = \nabla c(\hat{y})$
 - Going through ops in the sequence, in reverse order:
 - Compute adjoint of parameters $\overline{g^{[l]}} \leftarrow \overline{g^{[l]}} + \frac{\partial a^{[l]}}{\partial g^{[l]}} \overline{a^{[l]}}$
 - Recurse $\overline{a^{[l-1]}} = \frac{\partial a^{[l]}}{\partial a^{[l-1]}} \overline{a^{[l]}}$

Manual adjoint differentiation

Example:

prediction and derivation code in numPy for a vanilla net with 2 hidden layers (ReLU activation)

[GitHub.com/aSavine/CompFinLecture/Python/vanillaNetNP.py](https://github.com/aSavine/CompFinLecture/Python/vanillaNetNP.py)

- We applied to the letter the general back-prop algorithm
- Coding all adjoint equations in the reverse order
- Code is hard to write, read, tedious and prone to error
- Screams to be automated
- Since we followed a recipe to write this code
- It should be possible to write code that follows the recipe

```
import numpy as np

def g(x):
    return np.maximum(x, 0.0)

def dg(x):
    return np.where(x>0, 1.0, 0.0)

def forwardPass(x, W1, b1, W2, b2, W3, b3):
    z1 = W1.dot(x) + b1          # op 1
    a1 = g(z1)                   # op 2
    z2 = W2.dot(a1) + b2         # op 3
    a2 = g(z2)                   # op 4
    y = W3.dot(a2) + b3          # op 5
    return [x, z1, a1, z2, a2], y # we must remember everything

def backwardPass(y, dy, W1, b1, W2, b2, W3, b3, memory):
    x, z1, a1, z2, a2 = memory
    # op 5
    dW3 = dy.dot(a2.T) # param
    db3 = dy             # param
    da2 = W3.T.dot(dy)  # recurse
    # op 4
    dz2 = dg(z2) * da2  # recurse
    # op 3
    dW2 = dz2.dot(a1.T) # param
    db2 = dz2            # param
    da1 = W2.T.dot(dz2) # recurse
    # op 2
    dz1 = dg(z1) * da1  # recurse
    # op 1
    dW1 = dz1.dot(x.T)  # param
    db1 = dz1            # param
    dx = W1.T.dot(dz1)  # recurse
    return dx, dW1, db1, dW2, db2, dW3, db3
```

Automatic adjoint differentiation

Here is the same code in TensorFlow:

[GitHub.com/aSavine/CompFinLecture/Python/vanillaNetTF.py](https://github.com/aSavine/CompFinLecture/Python/vanillaNetTF.py)

- We only code the forward pass
- On execution of our code, TF does not *compute* anything
- All it does is build computation graph
= sequence and dependency of ops in the calculation
- Then, we ask TF to run the forward pass (on GPU!)
- And --with a call to *tf.gradients()* the adjoint pass (also on GPU)
- Since TF has the graph in memory, it knows the sequence of ops
- So TF can easily traverse it in reverse order and accumulate adjoints
- Recall, Jacobians of all ops are hard-coded
- TF does all this automatically, behind the scenes

```
import numpy as np
import tensorflow as tf

print(tf.__version__)
print(tf.test.is_gpu_available())

def forwardPassAuto(x, W1, b1, W2, b2, W3, b3):

    # use tf syntax, not np
    z1 = tf.matmul(W1, x) + b1
    a1 = tf.nn.relu(z1)
    z2 = tf.matmul(W2, a1) + b2
    a2 = tf.nn.relu(z2)
    y = tf.matmul(W3, a2) + b3

    return y # no need to explicitly remember anything

tf.reset_default_graph()

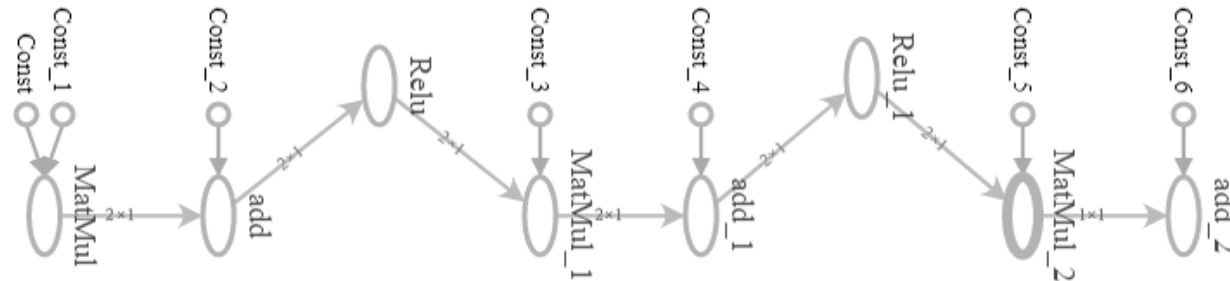
# these lines doesn't compute anything, just set graph in tf memory
y = forwardPassAuto(x, W1, b1, W2, b2, W3, b3)
loss = (y - target) ** 2
grads = tf.gradients(loss, [x, W1, b1, W2, b2, W3, b3])

# now we compute (on GPU!)
sess = tf.Session()
print(sess.run(y))
dx, dW1, db1, dW2, db2, dW3, db3 = sess.run(grads)
sess.close()
```

Computation graphs

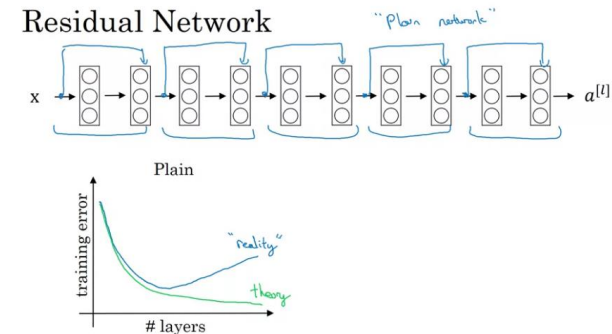
- AD is automated by building computation graphs in memory
- Calls to TF functions don't calculate anything, they build nodes in TF's computation graph
- You may visualize the graph with TensorBoard

```
# these lines doesn't compute anything, just set graph in tf memory
y = forwardPassAuto(x, W1, b1, W2, b2, W3, b3)
# write the graph
writer = tf.summary.FileWriter('c:/temp/graphs/', tf.get_default_graph())
writer.close()
```



Adjoint accumulation

- We need one more thing to make back-prop fully general
- Many networks implement parameter sharing, meaning that $g^{[l]}$ and $g^{[k]}$ may have common elements
- To compute derivatives to shared parameters, we must sum-up their contributions to different ops
(You may demonstrate this as an exercise, although we are going to the bottom of this later with AAD)
- Similarly, some nets implement skip layers
 - Means $a^{[l]}$ not only an input to $\phi^{[l+1]}$ but also some future $\phi^{[l+p]}$
 - For instance, He and al.'s "res nets"
Who won 1st place in ILSVRC 2015 and COCO 2015
 - (We also used this in financial nets)
- So back-prop must *accumulate* adjoints instead of writing them:
- And all adjoints must be initialized to zero before back-prop



[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

image from the
Deep Learning Specialization on Coursera

Final general back-prop algorithm

1. Perform a forward pass, store all results $a^{[l]}$ of all ops $\varphi^{[l]}$

2. Seed

a. Zero all adjoints $\overline{g^{[l]}}, \overline{a^{[l]}} \leftarrow 0$

b. Initialize the final result's adjoint $\overline{a^{[L]}} \leftarrow \bar{y} = \nabla c(\hat{y})$

3. Recursively (from the end) accumulate adjoints, knowing $\overline{a^{[l]}}$

a. Parameter derivatives: $\overline{g^{[l]}} \leftarrow \overline{g^{[l]}} + \frac{\partial \varphi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial g^{[l]}} \overline{a^{[l]}}$

b. Recursion: $\overline{a^{[l-1]}} \leftarrow \overline{a^{[l-1]}} + \frac{\partial \varphi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial a^{[l-1]}} \overline{a^{[l]}}$

Reverse order magic

- We have seen that back-propagation computes a multitude of derivatives with spectacular speed
- Around 3 times a forward pass
- By reversing order of calculation for the adjoint pass
 - When we compute $\overline{a^{[l-1]}}$ we already know $\overline{a^{[l]}}$
 - So we can compute $\overline{a^{[l-1]}}$ as a function of $\overline{a^{[l]}}$
 - And we only need differentiate one op: $\overline{a^{[l-1]}} \leftarrow \overline{a^{[l-1]}} + \frac{\partial \phi^{[l]}(a^{[l-1]}; g^{[l]})}{\partial a^{[l-1]}} \overline{a^{[l]}}$
 - We don't need to differentiate the whole network every time
- This may give the impression that reversing order is *only* about computing adjoints one from the other
- This impression, however, is wrong: there is an inherent efficiency to reversing order, at least for scalar computations (where the end result to be differentiated is a real number)

Reverse order magic (2)

- Forget neural nets, parameters, losses and consider the following calculation computing a scalar result y from a vector x , through successive transformations of x through a sequence of ops

$$a^{[0]} = x \xrightarrow{\varphi_1} a^{[1]} \xrightarrow{\varphi_2} a^{[2]} \xrightarrow{\varphi_3} a^{[3]} \dots \xrightarrow{\varphi_L} y = a^{[L]}$$

- Later, we will see that *any* calculation can be decomposed into a sequence of ops like this
- Recall, ops are vector to vector transformations:
 - Product by a matrix
 - Element-wise application of a scalar function
 - Convolution
 - ...
- Ops are limited in number, their Jacobians are known and they are building blocks to all calculations

Reverse order magic (3)

- We want to compute the gradient of y to x $a^{[0]} = x \xrightarrow{\varphi_1} a^{[1]} \xrightarrow{\varphi_2} a^{[2]} \xrightarrow{\varphi_3} a^{[3]} \dots \xrightarrow{\varphi_L} y = a^{[L]}$

- By the chain rule:
$$\begin{bmatrix} \frac{\partial y}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[0]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \varphi_1}{\partial a^{[0]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_2}{\partial a^{[1]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_3}{\partial a^{[2]}} \end{bmatrix} \dots \begin{bmatrix} \frac{\partial \varphi_{L-1}}{\partial a^{[L-2]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_L}{\partial a^{[L-1]}} \end{bmatrix}$$

- Where we highlighted what are matrices (Jacobians) and what are vectors (Gradient)
- The reason why the rightmost Jacobian is a gradient is that the calculation is *scalar*: $y = a^{[L]}$ is a real number, not a vector
- Which is the major assumption here: back-prop and AAD apply to scalar calculations:
the final result must be a real number
(actually, back-prop is more efficient than forward prop as long as $\dim(y) < \dim(x)$ as we will see)

Reverse order magic (4)

• This matrix product:

$$\begin{bmatrix} \frac{\partial y}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[0]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_1}{\partial a^{[0]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_2}{\partial a^{[1]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_3}{\partial a^{[2]}} \end{bmatrix} \dots \begin{bmatrix} \frac{\partial \phi_{L-1}}{\partial a^{[L-2]}} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_L}{\partial a^{[L-1]}} \end{bmatrix}$$

→ direct
← reverse

- Can be computed left to right (direct order) or right to left (reverse order)

direct order

- Compute $\begin{bmatrix} \frac{\partial a^{[2]}}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[1]}}{\partial x} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[2]}}{\partial a^{[1]}} \end{bmatrix}$
- Then $\begin{bmatrix} \frac{\partial a^{[2]}}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[1]}}{\partial x} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[2]}}{\partial a^{[1]}} \end{bmatrix}$ all the way to $\begin{bmatrix} \frac{\partial y}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial a^{[L-1]}}{\partial x} \end{bmatrix} \begin{bmatrix} \frac{\partial a^{[L]}}{\partial a^{[L-1]}} \end{bmatrix}$

- We forward-propagate **Jacobian matrices** to x from $a^{[1]}$ to $a^{[L]}$
- Every time, we multiply matrices to obtain matrices
- Complexity of matrix product is **cubic**

reverse order

- Compute $\begin{bmatrix} \overline{a^{[L-2]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_{L-1}}{\partial a^{[L-2]}} \end{bmatrix} \begin{bmatrix} \overline{a^{[L-1]}} \end{bmatrix}$
- Then $\begin{bmatrix} \overline{a^{[L-3]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_{L-2}}{\partial a^{[L-3]}} \end{bmatrix} \begin{bmatrix} \overline{a^{[L-2]}} \end{bmatrix}$ all the way to $\begin{bmatrix} \overline{x} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_1}{\partial a^{[0]}} \end{bmatrix} \begin{bmatrix} \overline{a^{[1]}} \end{bmatrix}$

- We back-propagate **adjoint vectors** from $\overline{a^{[L]}} = 1$ to $\overline{a^{[1]}} = \frac{\partial y}{\partial x}$
- Every time, we multiply matrices with vectors to obtain vectors
- Complexity of matrix by vector product is **quadratic**

Reverse order magic (5)

- Hence, reversing the order of calculation decreases complexity *by and order of magnitude*

- The cost is memory:

- Recall for instance the adjoint equations for recursion: $\overline{a^{[l-1]}} \leftarrow \overline{a^{[l-1]}} + \frac{\partial \phi^{[l]}(a^{[l-1]}; \mathcal{G}^{[l]})}{\partial a^{[l-1]}} \overline{a^{[l]}}$

- To execute them in reverse order,

- Either all Jacobians $\frac{\partial \phi^{[l]}(a^{[l-1]}; \mathcal{G}^{[l]})}{\partial a^{[l-1]}}$ must be computed during the forward pass and held in memory
- Or, all the results $a^{[l-1]}$ must be stored in memory
- We assumed the second option for simplicity so far, in reality, first option is often better

- Hence, back-prop consumes vast amounts of memory in return for spectacular speed

Conclusion

- Reversing order of calculation for differentiation makes it an order of magnitude faster, in return for memory consumption
- Computing differentials this way is the same as accumulating adjoints
- Back-prop is efficient and not only because of “telescopic” parameter derivatives
- The general back-prop algorithm is called Adjoint Differentiation or AD
- Its automated implementation is called Automatic AD or AAD
- We will now generalize it to any calculation, and code it in C++ ourselves
- First, let us demonstrate some important application of ANNs in derivatives finance

Deep learning in finance

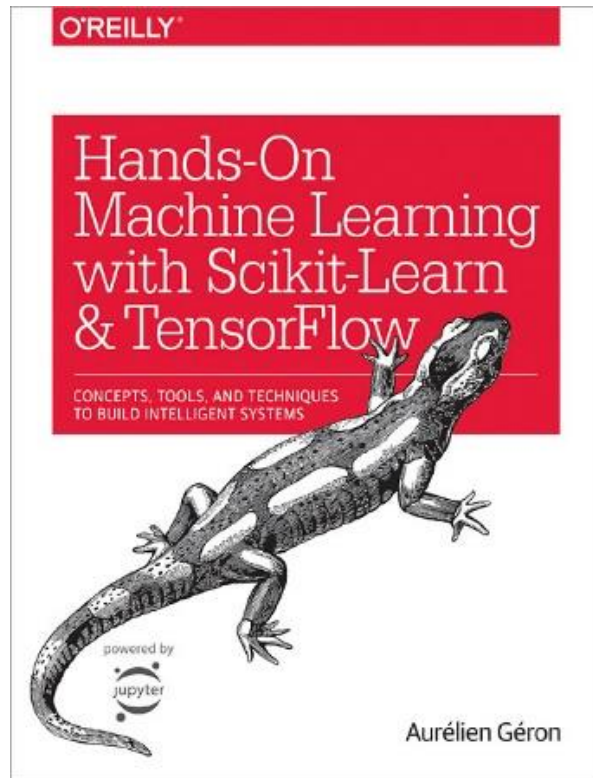
Milestones

- Deep Learning in derivatives finance is arguably the hottest topic of current research -- Milestones so far (all dated 2018-2019) include:
 - Learning derivatives *prices*
 - McGhee (2018) trained a net to compute SABR prices, ultra fast and more accurate than Hagan's original approximation
 - Ferguson and Greene (2018) trained a net to price basket options with inputs including correlation
 - Horvath and al. (Deeply Learning Volatility, 2019) built a net for pricing options with rough volatility -- This is particularly significant because:
 - There exist no alternatives for pricing with rough vol other than Monte-Carlo
 - Allows efficient, industrial calibration, pricing and risk management with rough vol
 - So far, rough vol was restricted to academic circles due to absence of efficient pricing
 - Learning *future prices and risks* of derivatives transactions, netting sets or trading books
 - Huge and Savine (Deep Analytics, 2019) established a DL-based methodology
 - for fast and accurate computation of *future* prices and risks
 - of arbitrary portfolios in arbitrary models
 - Also established *differential regularization* to significantly improve the performance of financial neural nets
 - Training derivatives hedging agents with Reinforcement Learning (RL) – Hans Buhler and al, Deep Hedging, 2018

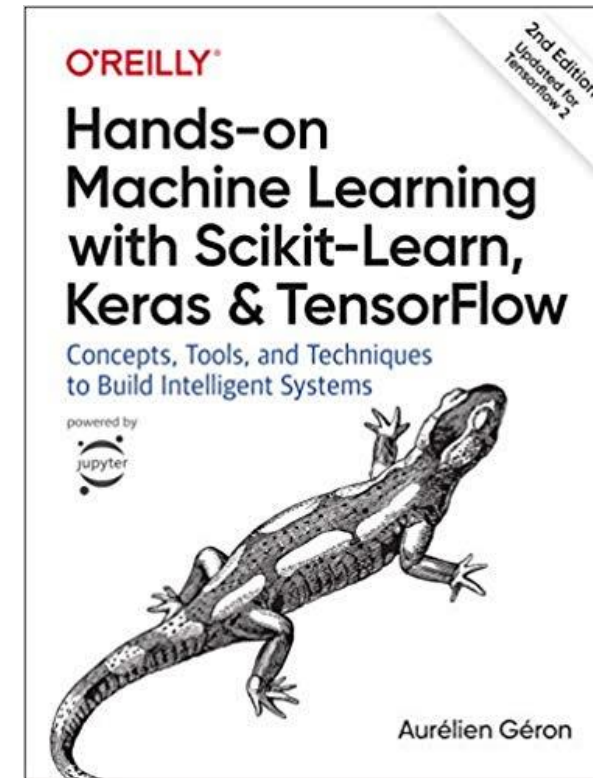
More TensorFlow

- We use TensorFlow to train pricing nets, we have already experimented with it -- TensorFlow is advertised as:
 - A Machine Learning framework open sourced by Google
 - The most popular framework for machine learning and deep learning
- TensorFlow is more than that:
 - A high performance math/numerical/matrix library with API in Python
 - With 3 perks (in increasing order of perkness)
 1. High level functions to work with ANNs easily and conveniently
 2. Calculations transparently executed on multi-core CPU or GPU
 3. **Built-in back-prop so derivatives of your code are computed in constant time, behind the scenes**
- Working with TensorFlow
 - To do all this, TF needs to know computation graphs: sequence and dependency of ops, **before** they are executed
 - Hence, working with TF is surprising at first, you must:
 - First *register* the calculations with TF, that is, build the graph in TF's memory space
 - Then, tell TF to run the graph, and back-prop through the graph when derivatives are needed

Learning TensorFlow



TensorFlow 1
which we use in these lectures



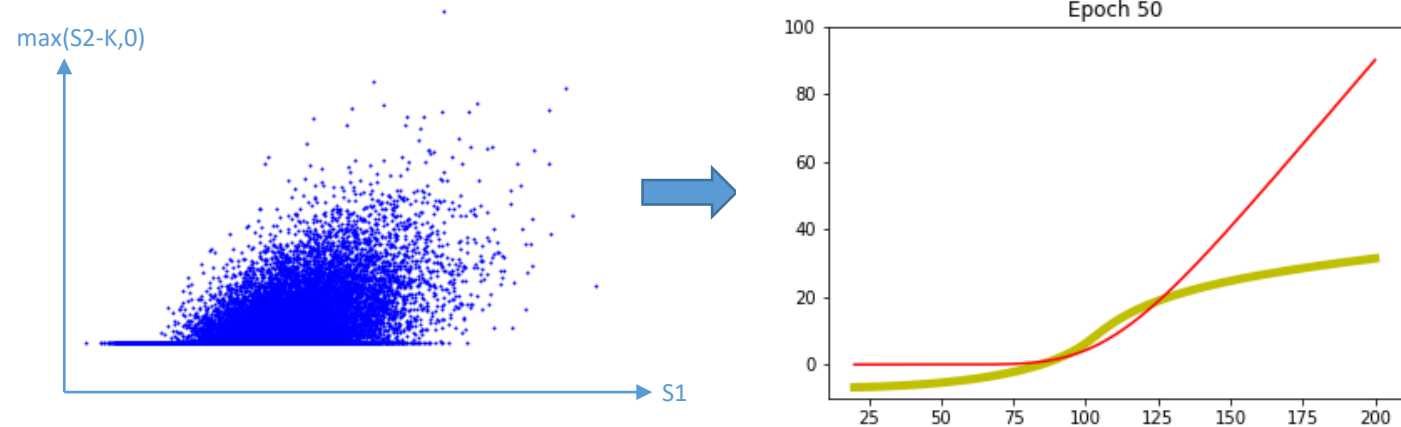
TensorFlow 2
*significantly different API
released October 2019*

Learn pricing from simulated data

- Simulation under risk-neutral probability Q in some arbitrary model
- x = state at time t_1 , including:
 - model state
example: spot
 - product state
example: is barrier hit?
- y = payoff at time $t_2 > t_1$
- For an ML model, correctly trained to minimize MSE on this data, we saw that:

$$\hat{y}(x) \rightarrow E[y|x] = E^Q[y_{t_2} | x_{t_1} = x]$$

- which is, indeed, the theoretical price at t_1
- Hinting that pricing can be learned from samples alone



Example: European call in Black & Scholes

- $\frac{dS}{S} = \sigma^2 dW^Q$ $x = S_{t_1}$ $y = (S_{t_2} - K)^+$

- True solution: Black & Scholes' formula

$$E^Q[y | S_{t_1} = x] = E^Q[(S_{t_2} - K)^+ | S_{t_1} = x] = BS_{K, t_2 - t_1, \sigma}(spot = x) = xN(d_1) - KN(d_2), d_{1/2} = \frac{\log\left(\frac{x}{K}\right) \pm \frac{\sigma^2}{2}(t_2 - t_1)}{\sigma\sqrt{t_2 - t_1}}$$

- For reference and assessment only
- The model only learns from the samples

Learn pricing from simulated data

2 Python notebooks, packaged as articles with code:

[GitHub.com/aSavine/CompFinLecture/MLFinance/](https://github.com/aSavine/CompFinLecture/MLFinance/)

[BlackScholesCall.ipynb](#) and [BachelierBasket.ipynb](#)

requires an installation of TensorFlow 1.x (**not** TF 2 – latest = 1.14 at the time of writing)

these notebooks are basically articles (with code) and may also be read (but not executed) without TensorFlow

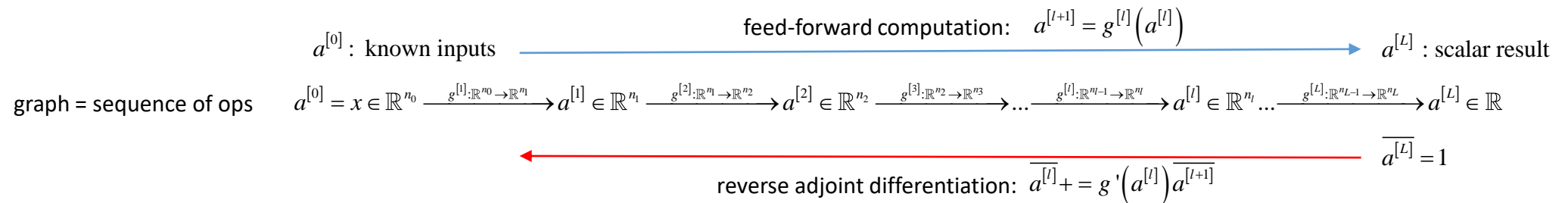
Conclusion

- We studied in detail the fundamental machinery of feed-forward and back-propagation
- There is a lot more to deep learning:
 - Speed and accuracy improvements: regularization, dropout, batch-norm, stochastic gradient descent, etc.
 - Convolutional nets with many impressive applications in computer vision
 - Recurrent nets with many impressive applications in natural language processing (NLP)
And also time series forecasting, proprietary trading strategies, etc.
 - Reinforcement learning, leveraging DL to learn strategies for playing games, driving cars or trading currencies
- Deep learning is here to stay
 - Overhyped due to recent successes in computer vision, NLP and chess/go --- hype will fade
 - This being said, DL genuinely offers efficient, elegant solutions to otherwise intractable problems in finance and elsewhere
 - DL will not replace financial models but it will be part of the financial quant toolbox
 - Financial quants and derivatives professionals cannot ignore it

AD through evaluation graphs

Computation graphs and adjoint propagation

- We have seen back-propagation through *computation graphs* of neural nets:



in this section, g are ops and g' are their (known) Jacobians --- not activations (we are no longer in Deep Learning)

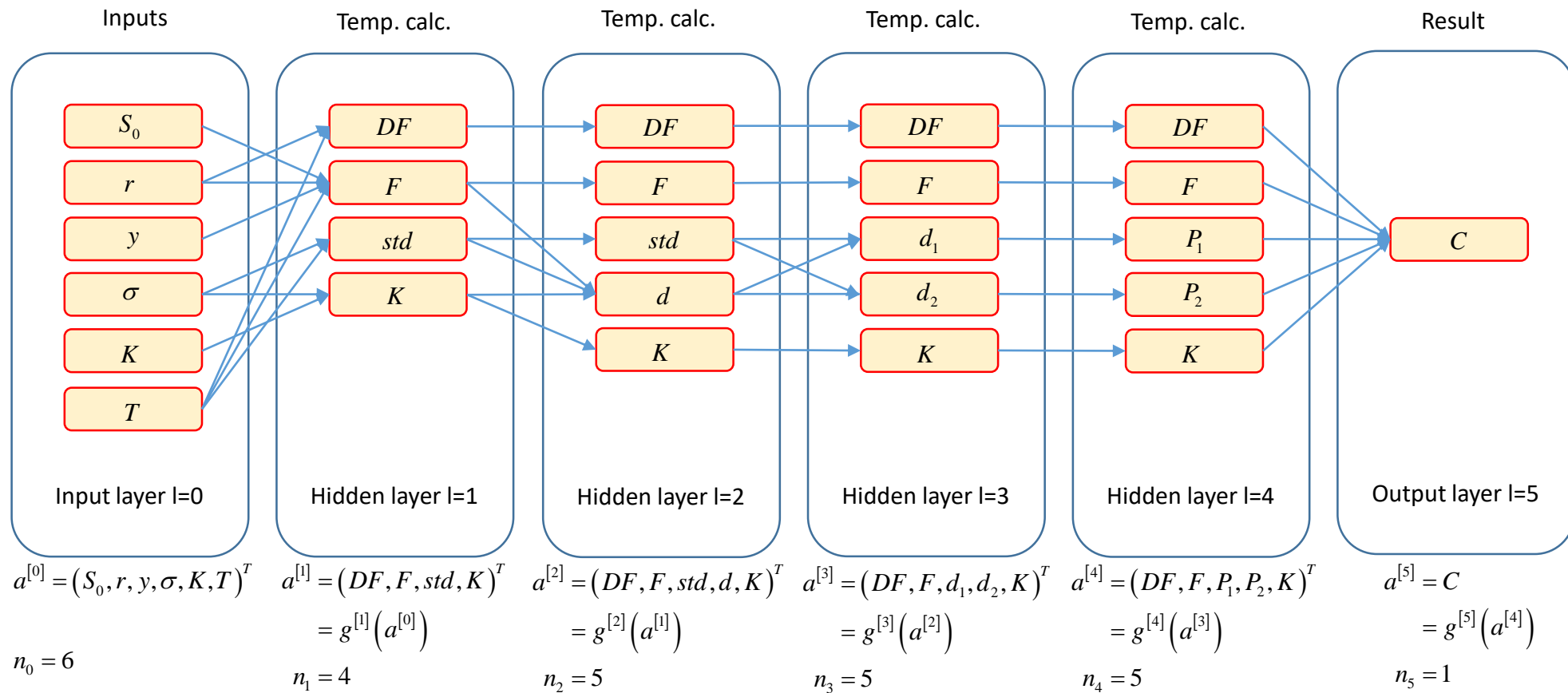
- In this section, we:
 - Illustrate that *any* scalar calculation defines a computation graph, same as neural nets
 - Therefore back-prop may compute efficiently the differentials of any scalar result
 - With the example of Black & Scholes' formula
 - And conclude with important remarks regarding op granularity and control flow

Example: Black & Scholes

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

- Discount factor to maturity: $DF = \exp(-rT)$
- Forward: $F = S_0 \exp[(r - y)T]$
- Standard deviation: $std = \sigma\sqrt{T}$
- Log-moneyness: $d = \frac{\log\left(\frac{F}{K}\right)}{std}$
- D's: $d_1 = d + \frac{std}{2}$, $d_2 = d - \frac{std}{2}$
- 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: $C = DF [FP_1 - KP_2]$

Black & Scholes: computation graph



Ops

- Like deep nets, **any** calculation is defined by its graph:

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

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

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

- In our Black & Scholes example, we defined the sequence of ops:

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

Jacobians

- For **any** calculation, we can compute the differentials of the result to the inputs: $\frac{\partial y}{\partial x} = \frac{\partial a^{[L]}}{\partial a^{[0]}}$
- By splitting the calculation into ops g , with known Jacobians $g^{[l]}, \frac{\partial a^{[l]}}{\partial a^{[l-1]}}$
- In our Black & Scholes example:

$$g^{[1]}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 0 & -x_6 \exp(-x_2 x_6) & 0 & 0 & 0 & -x_2 \exp(-x_2 x_6) \\ \exp[(x_2 - x_3) x_6] & x_1 x_6 \exp[(x_2 - x_3) x_6] & -x_1 x_6 \exp[(x_2 - x_3) x_6] & 0 & 0 & x_1 (x_2 - x_3) \exp[(x_2 - x_3) x_6] \\ 0 & 0 & 0 & \sqrt{x_6} & 0 & \frac{x_4}{2\sqrt{x_6}} \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}^T g^{[2]}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{x_2 x_3} & -\frac{\log(x_2/x_4)}{x_3^2} & -\frac{1}{x_3 x_4} \\ 0 & 0 & 0 & 1 \end{pmatrix}^T$$

$$g^{[3]}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 & 0 \\ 0 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T g^{[4]}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & n(x_3) & 0 & 0 \\ 0 & 0 & 0 & n(x_4) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T g^{[5]}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = ((x_2 x_3 - x_5 x_4), x_1 x_3, x_1 x_2, -x_1 x_5, -x_1 x_1 x_4)^T$$

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

Adjoint equations

- We have identified the calculation graph with its sequence of ops g and their Jacobians g'
- We have run it once and stored all the intermediate results a
- We may now compute all the derivatives of in constant time by back-prop:
 - Seed $\overline{a^{[l < L]}} = 0, \overline{a^{[L]}} = 1$
 - Back-prop $\overline{a^{[l]}}_+ = g'(a^{[l]}) \overline{a^{[l+1]}}$
 - Result $\frac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$
- Not that interesting with Black & Scholes: only 6 inputs/derivatives: delta, vega, rho, ...
- But the exact same algorithm applies with large evaluations taking 6,000 inputs --- very interesting then
- Exercise: unroll back-prop in the Black & Scholes' example and show that the results are the (well known) "Greeks"

“Atomic” ops



- What we just did is:

- Split a calculation

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

- Into a sequence of simpler calculations

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

- Then, we applied back-propagation to compute derivatives

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

- Pushing this logic to limit, we will:

- Split any calculation into a sequence of “atomic” ops: +, -, *, /, pow(), exp(), log(), sqrt(), sin(), cos(), ...
- Then, we have a very limited number of building block ops (about 20 in total, try to count them)
- **And, yet, any numerical calculation is, always, a (long) sequence of these 20-something building blocks!**
- This is another mind-twisting fact about AAD
see Rolf Poulsen’s Wilmott article: antoinesavine.files.wordpress.com/2018/12/poulsen-2018-wilmott.pdf
- For these 20-something building blocks, we know the derivatives g' : $\exp' = \exp$, $\log' = 1/x$, $\text{sqrt}' = 1 / 2\text{sqrt}$, etc.
- Therefore, we can hard-code these 20-something derivatives once and for all, and fully automate back-prop ➔ this is AAD

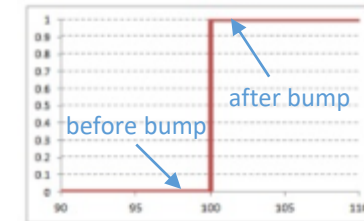
Control flow

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

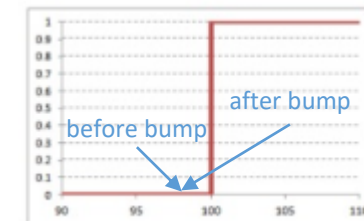
- Mathematical equations are more than sequences of building blocks
We also have limits, differentials, integrals, etc.
- But their numerical implementation is always nothing more, nothing less than a sequence of building block ops...
- ... and control flow, also called *branching*: “if this then that” statements
- Control flow makes functions discontinuous, hence, not differentiable
- Attempts to differentiate control flow with bumping results into instabilities
- AD/AAD computes exact differentials – same as infinitesimal bumps
- So derivatives are always taken on the same side of the control flow, and never through control flow
- This is not always the desired result, so something to take into account
- Best solution is smoothing, see slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic

Rolf Poulsen

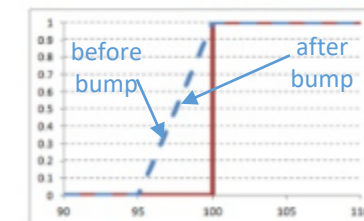
$f(x) = 1$ if $x > 100$, else 0



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



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



Smoothing

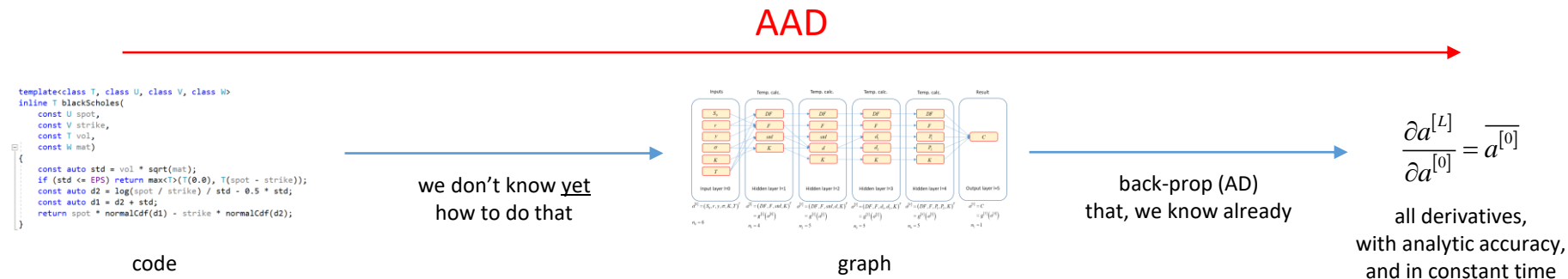
- Approx f by close continuous func
- Instabilities/incorrectness vanish

Conclusion

- All calculations define a graph, down to 20-something “atomic” ops **g**s
- Like deep nets, all those evaluation graphs may be arranged in successive layers, each an op **g** of the previous one (with known Jacobians **g'**):
- Once the evaluation graph and its ops **g** are known, along with their Jacobians **g'** (and the forward pass was executed once so all intermediate results **a** were stored)
- All derivatives $\frac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$ can be computed in constant time by back-prop
- All that remains is to figure the graph from some arbitrary calculation code

inputs: $a^{[0]} = x$
 feed-forward: $a^{[l]} = g^{[l]}(a^{[l-1]})$
 result: $y = a^{[L]}$

seed: $\overline{a^{[l < L]}} = 0, \overline{a^{[L]}} = 1$
 recurse: $\overline{a^{[l]}} = g'^{[l]}(a^{[l]}) \overline{a^{[l+1]}}$
 result: $\frac{\partial a^{[L]}}{\partial a^{[0]}} = \overline{a^{[0]}}$



Recording calculations on tape

C++ code

- From there on, we work with C++ code
- We write simplistic, beginner level C++
- We show AAD can still be implemented with rather spectacular results
- We refer to the curriculum for a professional, generic, parallel, efficient implementation
- For convenience, we export the C++ functions to Excel for test and visualization
- There is a tutorial for doing this: sites.google.com/view/antoinesavine-cpp2xl
- **All the code in these slides is freely available on GitHub** [GitHub.com/aSavine](https://github.com/aSavine)
 - professional code repo: [CompFinance](#)
 - C++ to Excel tutorial with all necessary files: [xlCppTutorial](#)
 - toy code repo: [CompFinLecture/ToyAAD](#)
 - main code for AAD, and instrumentation of Dupire's model: [toyCode.h](#)
 - utilities: [gaussians.h](#), [interp.h](#), [matrix.h](#)
 - random numbers: [random.h](#), [mrg32k3a.h](#), [sobol.h](#), [sobol.cpp](#)

Automatic differentiation

- Finite differences
 - Computes derivatives by running evaluation repeatedly
 - Only requires executable evaluation code
 - Linear complexity in the number of differentials, due to repetition of evaluation for every differential
- Adjoint Differentiation
 - Computes derivatives with back-prop in reverse order over an evaluation graph
 - Requires a graph to traverse -- executable code is *not* enough
 - Constant complexity in the number of differentials : efficient differentiation of scalar code
- With AD, we must extract an evaluation graph: sequence of ops (nodes) and their dependencies (edges)

Graph extraction

- Explicit graph construction
 - Solution implemented in TensorFlow
 - Lets clients explicitly build graphs by calling graph creation functions exported to many languages
 - Then use a TF *session* to evaluate or differentiate the graphs efficiently: run forward and backward passes, on multicore CPU or GPU
 - Smart and efficient
 - But forces developers to explicitly build graphs in place of calculation code
 - (TensorFlow has a nice API that makes building graphs somewhat similar to coding calculations)
 - But what we want here is take some arbitrary calculation code and automatically extract its graph
- Source transformation
 - Code that reads and understands evaluation code, then builds graphs and writes backward differentiation code automatically
 - Complex, specialized work similar to writing compilers
 - Variant: template meta-programming and introspection (chapter 15 of curriculum)

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) we decide exactly what happens:

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

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

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

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

Recording operations

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

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

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

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

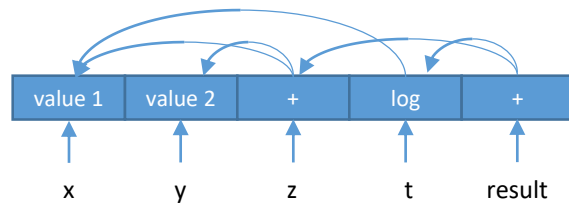
myNumberType x = 1, y = 2; // initializes x to 1 and y to 2 and records them
myNumberType z = x + y; // records addition with dependencies on x and y
myNumberType t = log(x); // records log with dependency on x
myNumberType result = t + z; // records addition with dependency on t and z
```

Lazy evaluation

- When this code is executed:

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

- Nothing is calculated, instead the following graph is recorded in memory:



- This sequence can be evaluated later (*lazy evaluation*) or differentiated (applying AD from back to front)
- This is how we build the evaluation graph at run time, by executing calculation code with a custom type for which operators are overloaded to perform recording

Conventional implementation

- We introduce the *conventional* implementation of AAD
 - Where every math operation: $+$, $-$, $*$, $/$, pow , log , exp , sqrt , ... is recorded on a data structure called **tape**
 - (It is really an evaluation graph but we call it “tape” in AAD lingo)
- Cutting-edge implementation records *whole expressions*
 - With template meta-programming and expression templates
 - Resulting in 2x to 5x faster code
 - Sometimes called “tape compression”
 - Extreme instance: “tapeless AD” where we don’t record anything!
 - All implemented in professional code on GitHub and explained in detail in chapter 15
 - In this presentation, we stick with simpler, conventional implementation
- We record every mathematical operation involved in a calculation
 - Every calculation, up to complex Monte-Carlo simulations, is a sequence of $+$, $-$, $*$, $/$, pow , log , exp , sqrt , ... !
 - We record every single one
 - **Note that all operations have either 0, 1 or 2 arguments**

Simplistic implementation

- In this presentation, we focus on the simplicity of the code
 - We use basic C++ code and disregard efficiency, scalability and best practice
 - Our aim is to explain the key ideas with simplistic code
 - This code works, just not efficiently, and it doesn't scale
- In the curriculum, on the contrary, we build professional, scalable, efficient code in modern C++
- This is particularly important for AAD because of:
 - Recording overhead
 - Every addition, multiplication, etc. produces a record
 - Recording necessarily involves an overhead
 - An efficient implementation must minimize overhead and make recording as efficient as possible
 - Vast memory consumption
 - We store in memory all the operations involved in a large calculation, this is a very large number of records
 - Estimated RAM consumption around 5GB per second
 - Efficient memory and cache management are key to an efficient implementation
- An efficient implementation of conventional AAD is given and explained in chapter 10
 - Effective, custom memory management
 - Recording with minimum overhead, cache efficiency and so on

Record and tape data structures

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

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

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

Custom real number

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

```
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

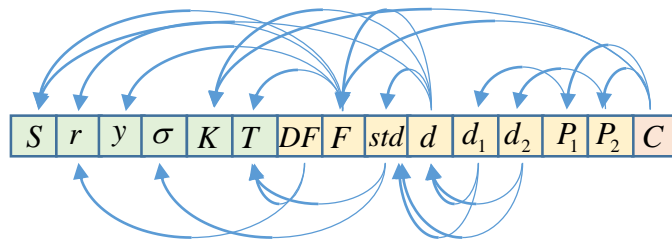
```
double spot = 100, rate = 0.02, yield = 0.05, vol = 0.2, strike = 110, mat = 2; // initializes inputs  
auto result = blackScholes(spot, rate, yield, vol, strike, mat);                // evaluates operations
```

- 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
```

Tape vs graph

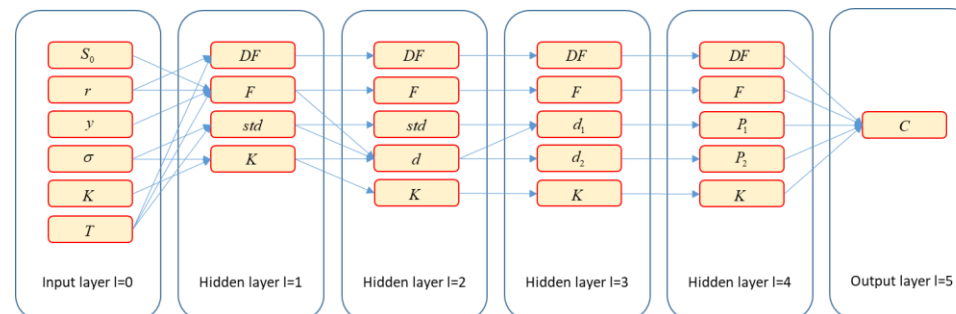
- Operator overloading gives us a tape of the mathematical operations involved in a calculation
- After execution of the templated code for Black & Scholes, we get the following tape:



Where we recall that operator overloading also gave us:

- The operations from ancestors A (arguments) to successors S (results)
- And their derivatives $\frac{\partial S}{\partial A}$

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



Tape vs graph (2)

- The tape and the graph express the exact same information, differently
- We can always turn a tape into a graph or vice-versa
- If we had a graph, we could run back-prop to compute all derivatives in constant time and be done
- We could turn our tape into a graph, but this would be a waste of precious computational resources
- Instead, we work with a slightly different version of the back-prop/AD algorithm
- So we can back-propagate adjoints directly through the tape (although it is not arranged in layers)
- Using the notions of *ancestor* (argument to an operation) and *successor* (result of an operation)
- AAD is the sum of a *recording framework* (e.g. operator overloading) and back-propagation through the tape

AAD

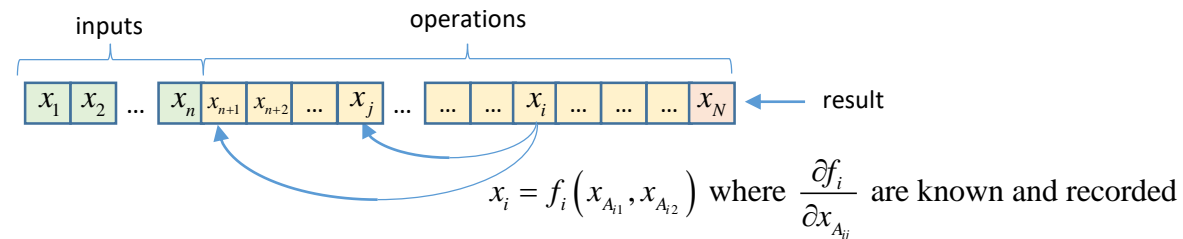
Ancestors

- We call the instrumented instance of our code:

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

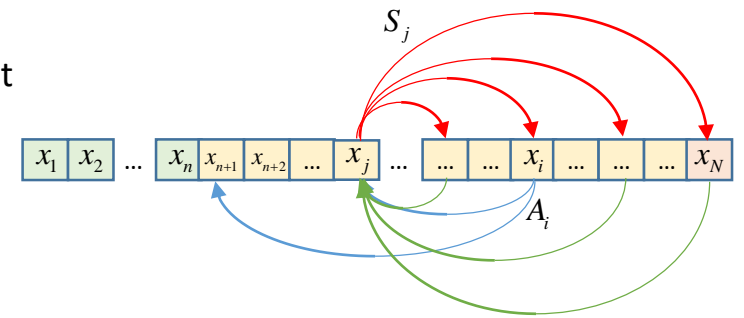
- Which evaluates the calculation and records all operations on tape
 - Denote f_i the operation number i , this is a function of 0, 1 or 2 arguments
 - The arguments must also be on tape, with indices $< i$
 - Denote A_i the set of indices of the ancestors (arguments) to f_i - this is a set of 0, 1 or 2 indices, all $< i$
 - Denote n the number of inputs and N the number of operations on tape, including inputs
 - Denote x_i the result of operation i and $y = x_N$ the final result
- Note that all the local derivatives $\frac{\partial f_i}{\partial x_j}$ for all $j \in A_i$ have been computed and recorded on tape, on evaluation, by our operators

- Our tape therefore looks like:



Successors

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



- Adjoint equation
 - Denote $\bar{x}_i \equiv \frac{\partial y}{\partial x_i} = \frac{\partial x_N}{\partial x_i}$ 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 $\frac{\partial f_i}{\partial x_j}$ are all on tape

Adjoint accumulation V1

- Adjoints therefore satisfy a backward recursion

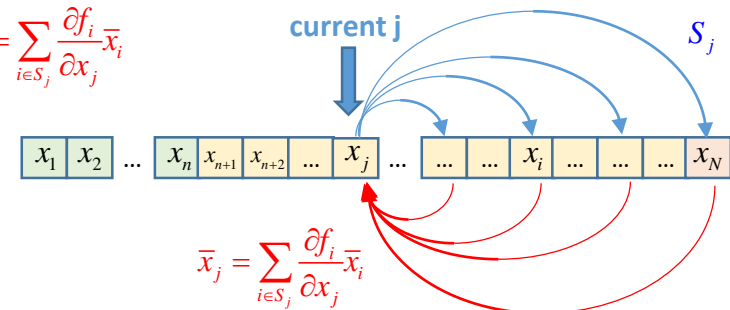
- The last adjoint $\bar{x}_N = 1$ is given

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

- Hence, the following (flawed) algorithm:

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

Summing over the successors of x_j :

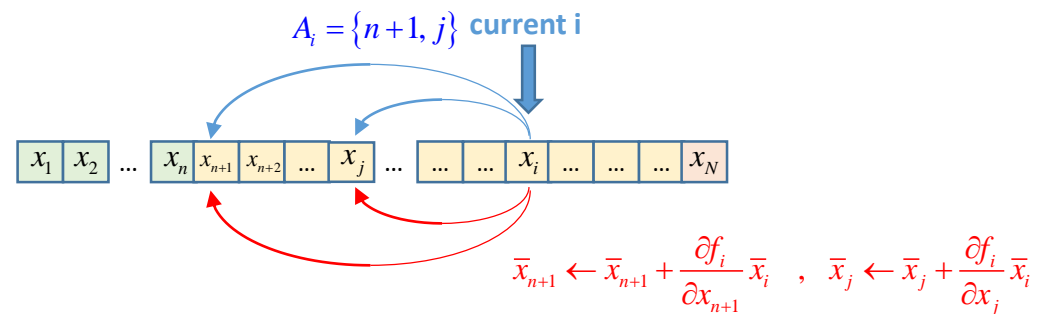


- Problems:

- Complexity: a node may have many successors to sum over
 - Impracticality: ancestors don't store their successors, it is the successors who store their (up to 2) ancestors and partial derivatives

Adjoint accumulation V2

- A more efficient/practical means of computing the same numbers:
 - The last adjoint $\bar{x}_N = 1$ is still given
 - Initialize all other adjoints to 0
 - Traverse tape in the reverse order, as previously, accumulating the sum $\bar{x}_j = \sum_{i \in S_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i$ **from successors**



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

Adjoint accumulation algorithm

1. Initialize all adjoints to 0 and the last adjoint to 1 (this is called *seeding* the tape): $\bar{x}_j = \delta_{N-j}$
 2. Repeat for i iterating backwards from N to 0 : for all $j \in A_i$: $\bar{x}_j \leftarrow \bar{x}_j + \frac{\partial f_i}{\partial x_j} \bar{x}_i$
 3. The differentials of the calculation to its inputs x_1, x_2, \dots, x_n are, by definition, $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$
- This algorithm is called (reverse) adjoint propagation
 - AAD is the sum of a recording framework and an implementation of adjoint propagation

Adjoint accumulation code

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

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

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

    return adjoints;
}
```

Application to Black & Scholes

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

```
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);
```

- The code works nicely
and produces correct values

```
// 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
```

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 be evaluated (and recorded) first so the theoretical upper bound is 3x one evaluation
 - Due to recording and tape traversal overhead, a good implementation generally produces many differentials in 4x to 10x
- Our professional code from chapters 10, 12 and 15 beats the theoretical bound!
 - Recall from the demonstration, one evaluation = 0.8sec, 1,081 differentials = 1.5sec, less than 2x one evaluation
 - Due to “selective instrumentation”, a strong optimization, explained, along many others, in chapter 12

Conclusion

- We implemented AAD in the simplest possible manner, scratching the surface of possibilities
- Part III (Chapters 8 to 15) gives the details of a complete, professional, efficient implementation
 - How to minimize recording overhead
 - Efficient memory management constructs
 - Apply check-pointed AAD to differentiate a calculation piece by piece to mitigate RAM footprint and cache inefficiency
 - Efficiently differentiate non-scalar calculations that return multiple results
 - Cutting-edge implementation with template meta-programming and expression templates, faster by 2x to 5x
 - Parallel implementation
 - Advise for debugging and optimization
 - And much more
- Still the simplistic code works and produces the correct values
- Not very interesting for Black & Scholes
 - Fast, analytic evaluation
 - Only 6 differentials to compute
- Next, we apply the framework to a barrier option in Dupire Monte-Carlo
 - Long, complex evaluation
 - 1,081 differentials to compute

AAD for financial simulations

Simple simulation code

- We implement a simplistic simulation code for a barrier option in Dupire's model
 - Recall local volatility is given in a matrix and bi-linearly interpolated in spot and time
- We need the following pieces, which we assume are given here
 - A matrix class to hold local volatilities (matrix.h in the repo, chapters 1 and 2 in the book)
 - A bi-linear interpolation function (interp.h in the repo, chapter 6, section 6.4 in the book)
 - Random number generators to produce independent Gaussian increments (chapters 5 and 6)
- The code is templated on the real number representation type

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 path
 - End up with many different functions implementing the same simulation logic for different couples of models and products
 - To modify the simulation logic, consistently change all the functions! This is obviously not viable
- Chapter 6 teaches a professional architecture for generic simulation libraries
 - Encapsulate scenario generation in Model objects
 - Encapsulate payoff evaluation in Product objects
 - Encapsulate simulation logic in a generic Monte-Carlo engine
 - Code every model and every product exactly once, mix and match at run time
- We stick with the simplistic code for demonstration purposes

An inefficient code

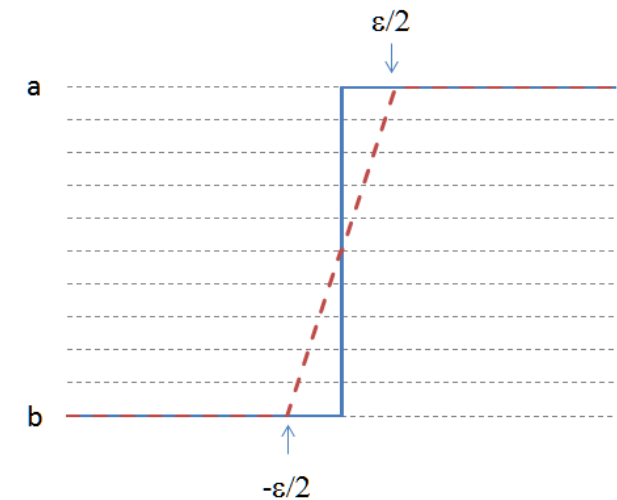
- The code does too much work repeatedly during simulations
 - Key to efficient Monte-Carlo code:
Do as much work as possible once, on initialization, and as little as possible repeatedly, during simulations
 - Example: we perform an expensive bi-linear interpolation in the innermost loop, for every path, on every time step
 - We interpolate in spot and time, spot is stochastic (scenario dependent), time is not
 - Therefore we can (and should) pre-interpolate in time on initialization
And perform only 1D interpolations in the innermost loop
- Chapter 6 teaches and builds fully optimized code
- The code is serial, it executes sequentially on one core
 - Since even our phones are multi-core today, professional code must be parallel
 - With Monte-Carlo simulations, it is relatively easy to obtain a speed-up by the number of physical cores
- Chapter 3 teaches modern parallel C++, chapter 7 builds a professional *parallel* simulation library
And section 12.5 instruments it with AAD in parallel
- For the purpose of demonstration, we stick with the not so efficient, serial version

Smoothing barrier options

- Discretely monitored barrier options are discontinuous, their value jumps to 0 at the barrier
 - Therefore, our code is not differentiable
 - AAD does not help: it cannot perform the impossible task of differentiating a discontinuous function
 - With finite differences, barrier risks are unstable, with AAD they are all zero
 - Find the reason why as an exercise!
- Therefore traders always smooth discontinuous transactions (barriers, digitals etc.)
- Smoothing = applying a close, continuous approximation in place of the discontinuous function
- Smoothing in finance, and its connection to fuzzy logic, are explained in the presentation:
[slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic](https://www.slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic)
- Here, we briefly explain the “smooth barrier” algorithm, usually applied on derivatives desks

Smooth barrier

- Hard barrier
 - 100% dead above the barrier, 100% alive below the barrier
 - Hence, discontinuous
- Soft barrier
 - 100% dead above the barrier **plus epsilon**, 100% alive below the barrier **minus epsilon**
 - In between, lose a fraction of notional interpolated between (barrier-epsilon,0) and (barrier+epsilon,1)
 - And continue with the remaining notional
 - Hence, continuous
- Smoothing and fuzzy logic
 - Like Schrodinger's cat, the transaction is in a superposition of dead and alive states
 - Smoothing is achieved by replacing sharp logic (dead or alive?) by fuzzy logic (how much alive?)
 - More in the presentation



Simulation code with smooth barrier

```
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)

// 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

- Our simple code returns the exact same result as the professional code
- It is twice slower than the serial version of the professional code
- On a octo-core computer, it is 16x 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]);

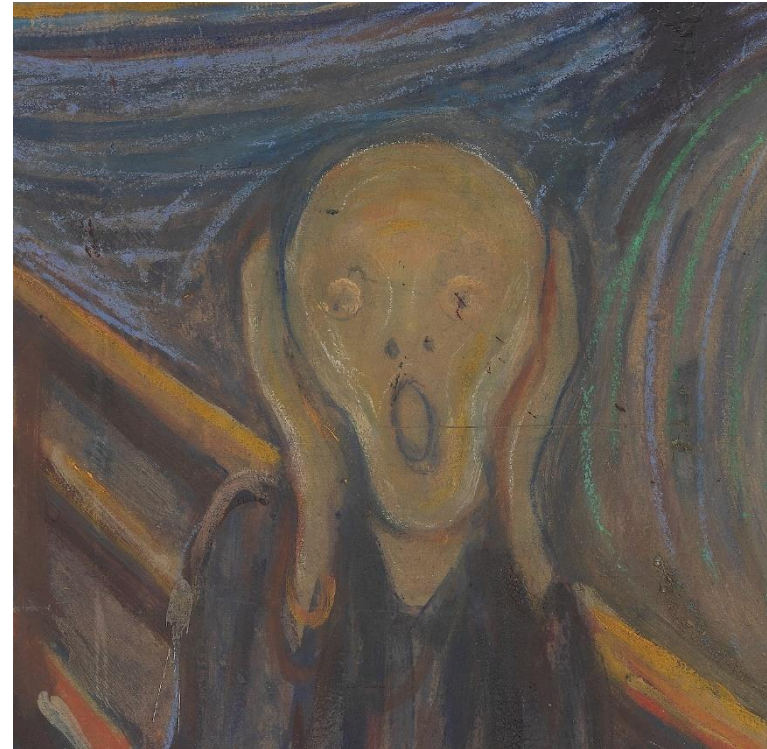
// 2. Call instrumented evaluation code, which evaluates the barrier option price and records all operations
Number nPrice = toyDupireBarrierMc(nS0, nSpots, nTimes, nVols, nMaturity, nStrike, nBarrier, Np, Nt, nEpsilon, random);

// 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 differentiation

- 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 batches of say, 1024 paths and average in the end
- Wipe the tape in between 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
- With mini-batches of size 1, this is known as “path-wise differentiation”
- This is also a particular, and simple case of the general check-pointing algorithm explained in chapter 13

Solution in code

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

```
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 = 512;
    // calculate batch sensitivities sequentially
    while (pathsToGo > 0)
    {
        // wipe tape
        tape.clear();

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

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

        pathsToGo -= paths;
    }
}
```

Performance

- With 100,000 paths, 156 steps, we compute the 1,081 differentials in around 7 seconds
- This is 1,081 differentials in the time of around 6 evaluations
- This is a very remarkable result, especially with such simplistic code
- Try it yourself with the toy code in the repo!
- This being said, the professional code is around 8 times faster in serial mode, 64 times faster in parallel mode on a octo-core CPU

Professional Financial Libraries

Interactive coding session

- We produced toy Monte-Carlo code that works
- And even produces many risks with spectacular accuracy and speed
- But this is not professional code
 - Tangled, not scalable to multiple models and products
 - Inefficient, speed may be easily and substantially improved
 - Serial
- Part II of the curriculum (chapters 4 through 7) teach the design and implementation of modern, professional, scalable, efficient, parallel financial libraries
- Professional code from the curriculum in: [GitHub.com/aSavine/CompFinance](https://github.com/aSavine/CompFinance)

Interactive coding session (2)

- Here, we review the main notions in a simplified context
- And refactor our Dupire/barrier code
- In an interactive coding session
- We work on the code in [GitHub.com/aSavine/CompFinLecture/interactiveToyCode](https://github.com/aSavine/CompFinLecture/interactiveToyCode) with entry point: main code: `toyCode.h`
- In particular, we review, interactively, directly in code, the following:
 1. Generic, scalable simulation code: separation, abstraction, encapsulation (chapters 4, 5, 6)
 2. Efficient Monte-Carlo and pre-processing (chapter 6)
 3. Parallel Monte-Carlo: const-correctness, mutation and race conditions, load balancing (chapters 3 and 7)
 4. Parallel AAD in a simplified architecture (chapter 12)

Thank you for your attention