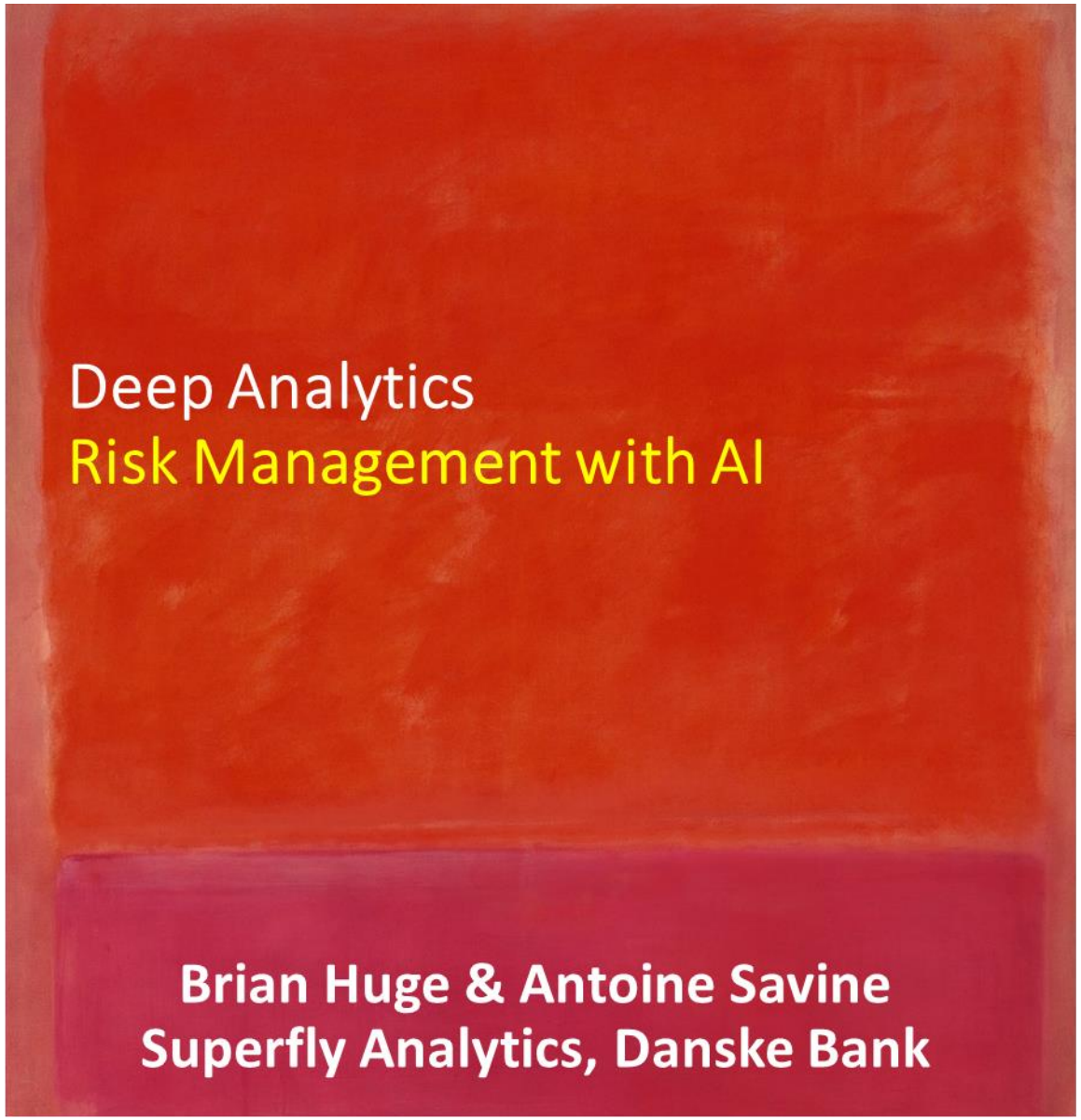



Superfly Analytics

# Contents


I. Review of (conventional) machine learning in the context of Derivatives pricing and risk
   1. Theoretical framework and notations for ML in the context of Derivatives pricing and risk
   2. Review of conventional ML: regression, PCA, neural networks

II. Differential machine learning
   3. Foundations
   4. Differential regression
   5. Differential PCA
   6. Differential ANN

# Part I
# Pricing with ML

How Machine Learning fits with Derivatives pricing

# Natural intelligence: SABR

**given**

**find**

**SABR:**
**efficient *approximation* of *v***

pricing **risk-neutral** model

$$\begin{cases} dF = \sigma F^{\beta} dW \\ \dfrac{d\sigma}{\sigma} = \alpha dZ \\ dW dZ = \rho dt \end{cases}$$

Derivatives instrument

$$y_T = (F_T - K)^+$$

**fast** pricing function

$$v(F_t, \sigma_t; t) = E\left[ y_T \,|F_t, \sigma_t \right]$$

$$\sigma_{\text{impl}} = \alpha \frac{\log(F_0/K)}{D(\zeta)} \left\{ 1 + \left[ \frac{2\gamma_2 - \gamma_1^2 + 1/F_{\text{mid}}^2}{24} \left( \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} \right)^2 + \frac{\rho\gamma_1}{4} \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} + \frac{2-3\rho^2}{24} \right] \varepsilon \right\},$$

$$F_{mid} = \sqrt{F_0 K} \qquad \zeta = \frac{\alpha}{\sigma_0} \int_K^{F_0} \frac{dx}{C(x)} = \frac{\alpha}{\sigma_0 (1-\beta)} \left( F_0^{1-\beta} - K^{1-\beta} \right), \quad \gamma_1 = \frac{C'(F_{\text{mid}})}{C(F_{\text{mid}})} = \frac{\beta}{F_{\text{mid}}},$$

$$C(F) = F^{\beta} \qquad \varepsilon = T\alpha^2 \qquad D(\zeta) = \log\left( \frac{\sqrt{1 - 2\rho\zeta + \zeta^2} + \zeta - \rho}{1 - \rho} \right), \quad \gamma_2 = \frac{C''(F_{\text{mid}})}{C(F_{\text{mid}})} = -\frac{\beta(1-\beta)}{F_{\text{mid}}^2}.$$

# Comments

| Natural intelligence: SABR | |
|---|---|
| - Here, the pricing model is **given**<br>- The goal is **not** to find a 'good' model<br>- But an **efficient computation** of the price<br><br>Hagan's formula brought stochastic volatility to trading desks and pioneered expansions in finance | |
| - Works with **one** model : SABR<br>and **one** instrument : European call<br>- Can we design **one** implementation<br>for **arbitrary** models and instruments? | |
| - Hagan cracked SABR formula with maths | |
| Derived **offline** and **reused** forever<br>- Took considerable skill and effort<br>- Subsequently reused trillions of times<br>on trading desks around the world | |

# Natural intelligence: SABR

**given**

**find**

**SABR:**
**efficient *approximation* of *v***

pricing **risk-neutral** model

$$\begin{cases} dF = \sigma F^{\beta} dW \\ \dfrac{d\sigma}{\sigma} = \alpha dZ \\ dW dZ = \rho dt \end{cases}$$

Derivatives instrument

$$y_T = (F_T - K)^+$$

**fast** pricing function

$$v(F_t, \sigma_t; t) = E\left[ y_T \,|F_t, \sigma_t \right]$$

$$\sigma_{\text{impl}} = \alpha \frac{\log(F_0/K)}{D(\zeta)} \left\{ 1 + \left[ \frac{2\gamma_2 - \gamma_1^2 + 1/F_{\text{mid}}^2}{24} \left( \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} \right)^2 + \frac{\rho\gamma_1}{4} \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} + \frac{2-3\rho^2}{24} \right] \varepsilon \right\},$$

$$F_{mid} = \sqrt{F_0 K} \qquad \zeta = \frac{\alpha}{\sigma_0} \int_K^{F_0} \frac{dx}{C(x)} = \frac{\alpha}{\sigma_0 (1-\beta)} \left( F_0^{1-\beta} - K^{1-\beta} \right), \quad \gamma_1 = \frac{C'(F_{\text{mid}})}{C(F_{\text{mid}})} = \frac{\beta}{F_{\text{mid}}},$$

$$C(F) = F^{\beta} \qquad \varepsilon = T\alpha^2 \qquad D(\zeta) = \log\left( \frac{\sqrt{1 - 2\rho\zeta + \zeta^2} + \zeta - \rho}{1 - \rho} \right), \quad \gamma_2 = \frac{C''(F_{\text{mid}})}{C(F_{\text{mid}})} = -\frac{\beta(1-\beta)}{F_{\text{mid}}^2}.$$

# Natural intelligence: SABR

**given**

**find**

**SABR:**
**efficient *approximation* of *v***

pricing **risk-neutral** model

$$\begin{cases} dF = \sigma F^{\beta} dW \\ \dfrac{d\sigma}{\sigma} = \alpha dZ \\ dW dZ = \rho dt \end{cases}$$

Derivatives instrument

$$y_T = (F_T - K)^+$$

**fast** pricing function

$$v(F_t, \sigma_t; t) = E\left[ y_T \,|F_t, \sigma_t \right]$$

$$\sigma_{\text{impl}} = \alpha \frac{\log(F_0/K)}{D(\zeta)} \left\{ 1 + \left[ \frac{2\gamma_2 - \gamma_1^2 + 1/F_{\text{mid}}^2}{24} \left( \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} \right)^2 + \frac{\rho\gamma_1}{4} \frac{\sigma_0 C(F_{\text{mid}})}{\alpha} + \frac{2-3\rho^2}{24} \right] \varepsilon \right\},$$

$$F_{mid} = \sqrt{F_0 K} \qquad \zeta = \frac{\alpha}{\sigma_0} \int_K^{F_0} \frac{dx}{C(x)} = \frac{\alpha}{\sigma_0 (1-\beta)} \left( F_0^{1-\beta} - K^{1-\beta} \right), \quad \gamma_1 = \frac{C'(F_{\text{mid}})}{C(F_{\text{mid}})} = \frac{\beta}{F_{\text{mid}}},$$

$$C(F) = F^{\beta} \qquad \varepsilon = T\alpha^2 \qquad D(\zeta) = \log\left( \frac{\sqrt{1 - 2\rho\zeta + \zeta^2} + \zeta - \rho}{1-\rho} \right), \quad \gamma_2 = \frac{C''(F_{\text{mid}})}{C(F_{\text{mid}})} = -\frac{\beta(1-\beta)}{F_{\text{mid}}^2}.$$

# Artificial intelligence: ML

**given**

**any** pricing **risk-neutral** model

$dx = \ldots$

**any** Derivatives instrument

$y_T = g(x_t)_{0 \le t \le T}$

**find**

**fast** pricing function

$v(x_t;t) = E[y_T | x_t]$

**efficient *approximation* of**

$v(x_t;t) = E[y_T | x_t]$

# Application

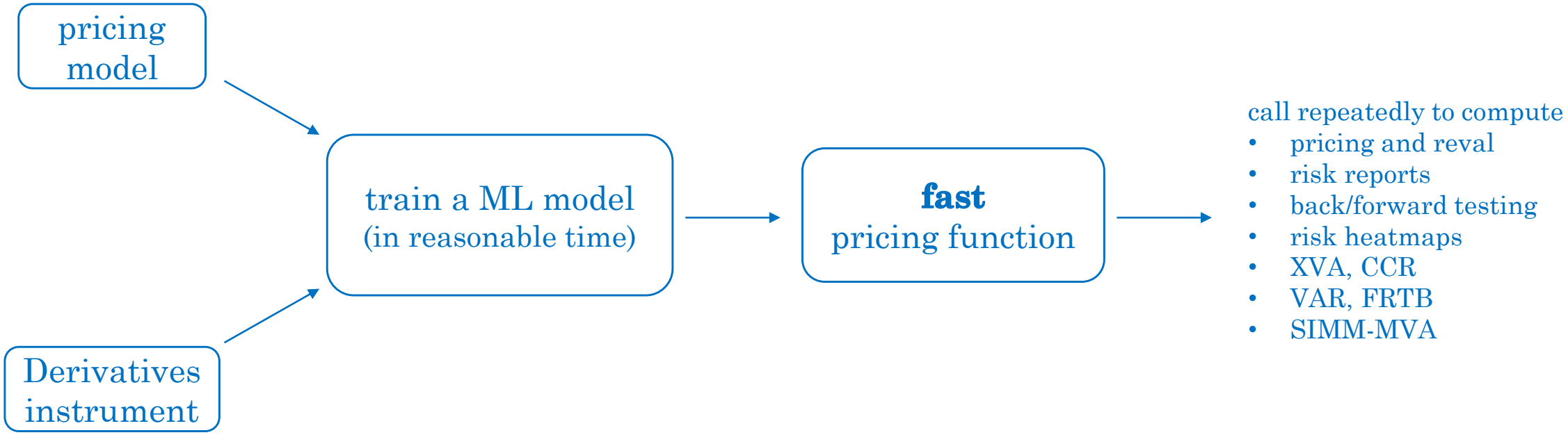
- Risk heatmaps, regulations like XVA, CCR, FRTB or SIMM-MVA
- Value the **same** Derivative transaction or trading book in the **same** pricing model
- In many **different** scenarios: often around 10,000
- Fast valuation by analytics, expansions, numerical integration or FDM in restricted cases
- In general, slow valuation with Monte-Carlo (MC)
- ➢ Massive computation cost of around 10,000 pricings by MC

- We propose to invest *some* time learning a fast computation and save the cost of 10,000 MCs
- The pricing function is learned and then used around 10,000 times
- Only makes sense if training cost is much less than 10,000 pricings by MC
- We can do it in around 5-10 pricings by MC, a speedup of x1000 to x2000

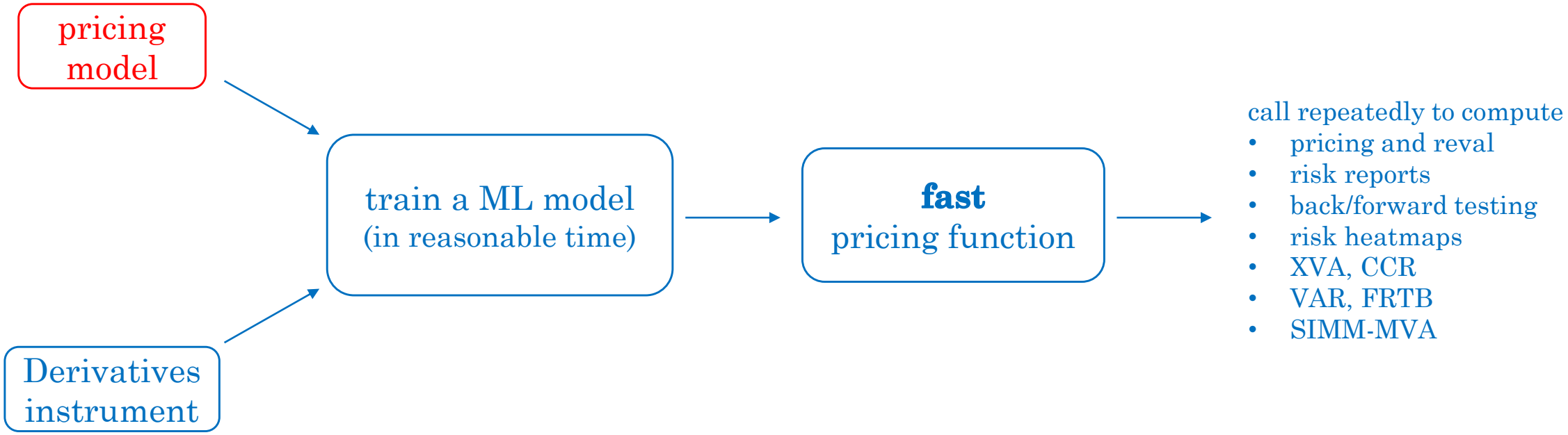
# Comments

| Natural intelligence: SABR | Artificial intelligence: ML |
|---|---|
| - Here, the pricing model is **given**<br>- The goal is **not** to find a 'good' model<br>- But an **efficient computation** of the price<br><br>Hagan's formula brought stochastic volatility to trading desks and pioneered expansions in finance | Same here:<br>We are not trying to learn a good (realistic or effective) model,<br>we want to learn an effective computation in a given model |
| - Works with **one** model : SABR<br>and **one** instrument : European call<br>- Can we design **one** implementation<br>for **arbitrary** models and instruments? | - Should work with any model<br>- And any Derivative instruments<br>- Including netting sets or trading books |
| - Hagan cracked SABR formula with **maths** | - ML learns from **examples** |
| Derived **offline** and **reused** forever<br>- Took considerable skill and effort<br>- Subsequently reused trillions of times<br>on trading desks around the world | Learned **online** and **disposable**<br>- Learned in the context of **one** risk calculation<br>- Used for this risk calculation only<br>- Learning must have limited computation cost |

# Big picture


pricing model
Derivatives instrument
train a ML model (in reasonable time)
fast pricing function
call repeatedly to compute
• pricing and reval
• risk reports
• back/forward testing
• risk heatmaps
• XVA, CCR
• VAR, FRTB
• SIMM-MVA

# Big picture


pricing model
Derivatives instrument
train a ML model (in reasonable time)
fast pricing function
call repeatedly to compute
• pricing and reval
• risk reports
• back/forward testing
• risk heatmaps
• XVA, CCR
• VAR, FRTB
• SIMM-MVA

# Pricing model

- Probabilistic description of the **risk-neutral** dynamics of the relevant market state variables
- Examples:

| | | |
|---|---|---|
| Black & Scholes/ Dupire | $dS_t = \mu(S_t,t)\,dt + \sigma(S_t,t)\,dW_t$ | state: underlying price $S_t$<br>dimension: 1 |
| Multi-underlying Bachelier/BS/Dupire | $d\vec{S_t} = \vec{\mu}(\vec{S_t},t)\,dt + \overline{\overline{\sigma(\vec{S_t},t)}}\,d\vec{W_t}$ | state: n underlying prices $\vec{S_t}$<br>dimension: n |
| Stochastic volatility ex: SABR | $dF_t = \sigma_t F_t^{\beta} dW_t \quad \frac{d\sigma_t}{\sigma_t} = \alpha dZ_t \quad dW_t dZ_t = \rho dt$ | state: forward and volatility<br>dimension: 2 |
| Yield curve ex: LMM | $dF(t,T_i) = \mu_i\left(t,\{F(t,T_j)\}\right)dt + \sigma_i\left(t,F(t,T_i)\right)d\vec{W_t}$ | state: all forward Libors<br>dimension: e.g. 120 for 30y and 3m Libors |
| Multi-asset Hybrid/Regulatory | consistent assembly of multiple models | state: concatenates underlying model states<br>dimension: sum of underlying dimensions |

# Pricing model

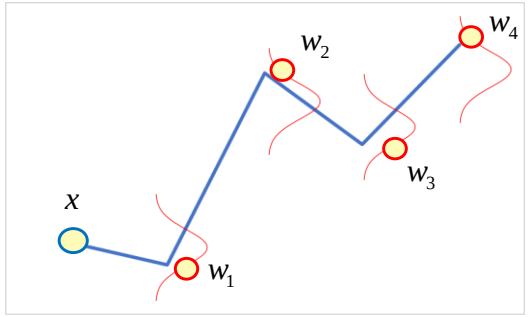
- General machinery: abstract definition of a pricing model
- A pricing model is a **function** $f : \mathbb{R}^n \times (0,1)^d \to$ set $\mathcal{P}$ of state vector paths
- That given inputs
  - $x \in \mathbb{R}^n$ initial state in dimension n
    (n: Markov dimension of the model)
  - $w \in (0,1)^d$ d uniform random numbers independently picked in (0,1)
    (d: how many random numbers are needed to build one path, i.e. MC dimension)
- Simulates a path for the state vector $(x_t) = f(x; w)$ **under the risk-neutral measure**

  with initial condition $x_0 = x$ and random numbers $w$

# Pricing model

- A pricing model is a **function** $f:\mathbb{R}^n\times(0,1)^d\to\mathcal{P}$
- In other terms, a model is a recipe for simulation of **risk-neutral** paths

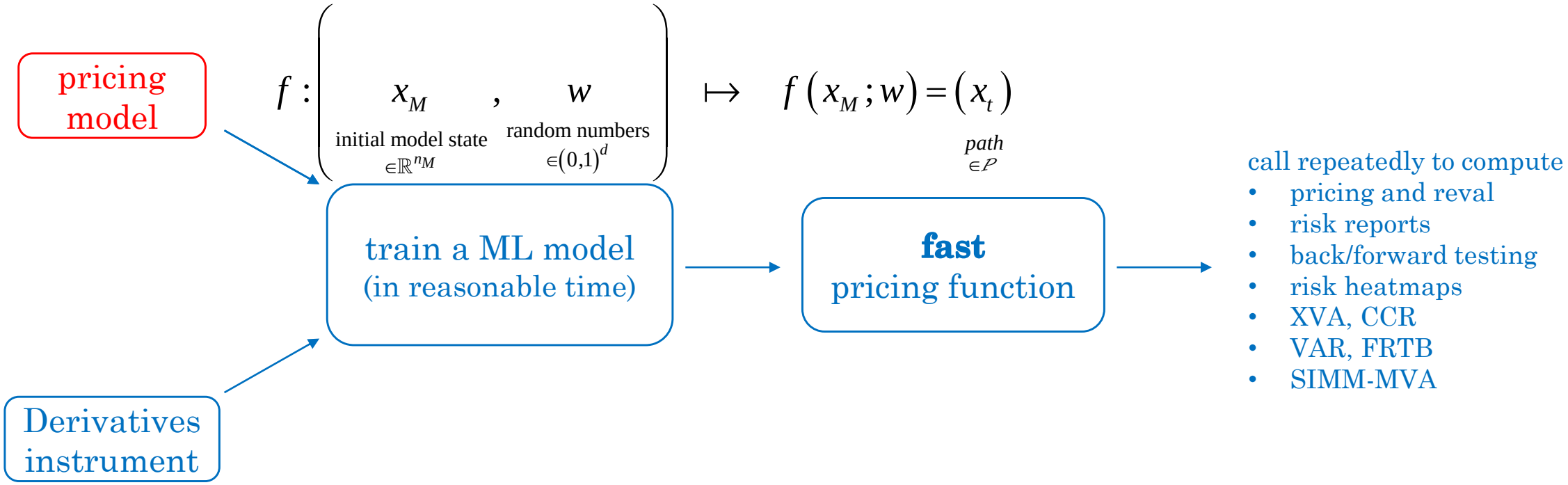
$$f(x;w)=$$



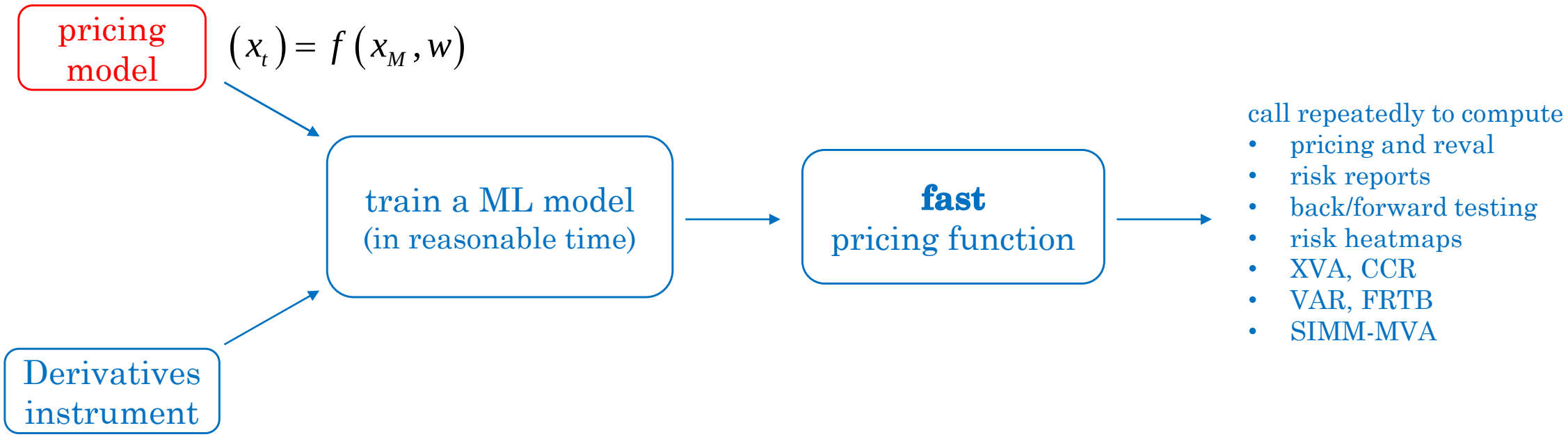

- Example: BS/Dupire under Euler's scheme $f(x;w)=\{S_{T_i}\}\quad S_0=x\quad S_{T_{i+1}}=\mu(S_{T_i},T_i)(T_{i+1}-T_i)+\sigma(S_{T_i},T_i)\sqrt{T_{i+1}-T_i}N^{-1}(w_i)$

  n=1, d: number of time steps

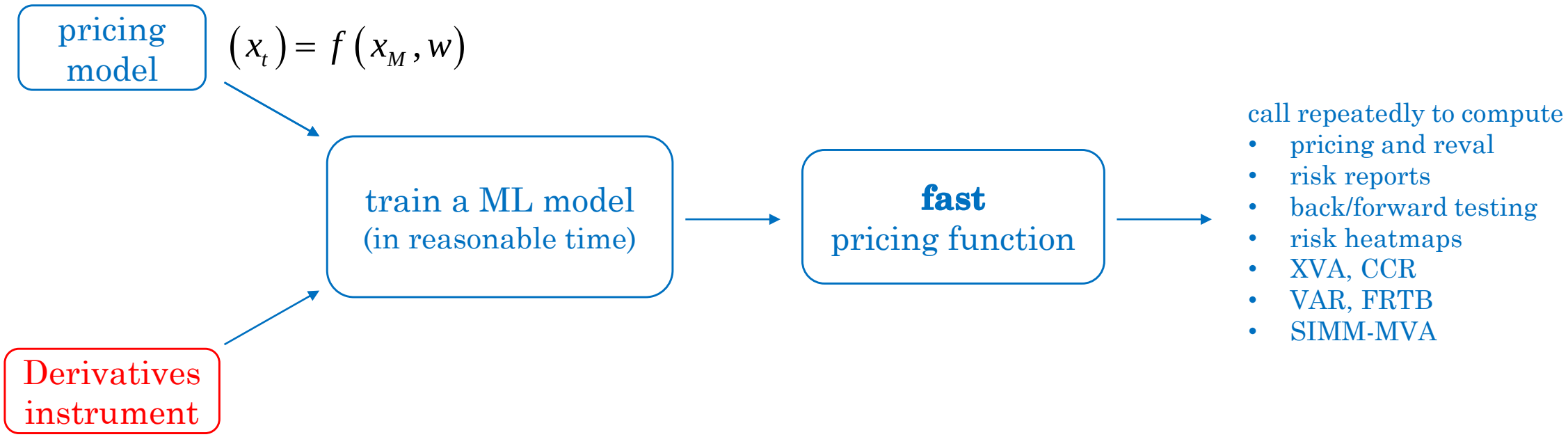
# Big picture


pricing model
$f : \left( x_M \in \mathbb{R}^{n_M} \text{ initial model state} \;,\; w \in (0,1)^d \text{ random numbers} \right) \mapsto f(x_M; w) = (x_t)$
path $\in \mathcal{P}$
Derivatives instrument
train a ML model (in reasonable time)
fast pricing function
call repeatedly to compute
• pricing and reval
• risk reports
• back/forward testing
• risk heatmaps
• XVA, CCR
• VAR, FRTB
• SIMM-MVA

# Big picture


pricing model
$(x_t) = f(x_M, w)$
train a ML model (in reasonable time)
Derivatives instrument
**fast** pricing function
call repeatedly to compute
• pricing and reval
• risk reports
• back/forward testing
• risk heatmaps
• XVA, CCR
• VAR, FRTB
• SIMM-MVA

# Big picture


pricing model
$(x_t) = f(x_M, w)$
train a ML model (in reasonable time)
Derivatives instrument
**fast** pricing function
call repeatedly to compute
• pricing and reval
• risk reports
• back/forward testing
• risk heatmaps
• XVA, CCR
• VAR, FRTB
• SIMM-MVA

# Derivatives instrument

- A Derivatives instrument is described by its term sheet
- Tells what cashflows are paid or received under what conditions
- Hence, a Derivative is a **function(al)** $g : \mathcal{P} \to \mathbb{R}$
- That given a path, returns **the sum of discounted cashflows** along that path
- Also called **payoff**
- Notes

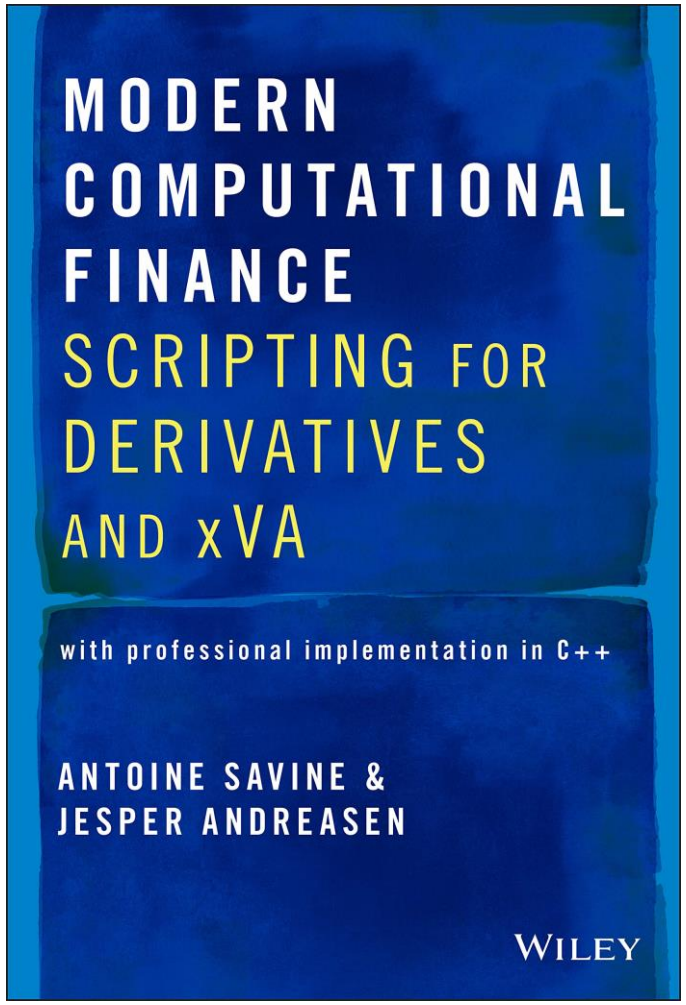
1. This definition also covers trading books by aggregation $g_{book} = \sum_{Derivative\ i} g_i$
2. - Things like early exercises are not directly covered by this definition
   - Classic LSM algorithm (Logstaff-Schwartz, 2001)
     transforms early exercises etc. into path-dependencies, which are covered

# Scripting

- A Derivative is a **function(al)** $g : \mathcal{P} \to \mathbb{R}$
- Best implemented with cashflow scripting
- Frames term sheet as a **smart contract**
- i.e. a computer program that executes the term sheet and computes the functional g given a path P

https://www.amazon.co.uk/Modern-Computational-Finance-Scripting-Derivatives/dp/111954078X

**November 2021**



# Derivative state

- A Derivative is a **function(al)** $g : \mathcal{P} \to \mathbb{R}$
- This is correct at launch, but we must handle Derivatives throughout lifetime
- For example
  - A barrier option or callable transaction may be alive or dead
  - A physical swaption may have been exercised into a swap
  - An Asian option (option on average) depends on the past accumulated average A $state = A \in \mathbb{R}$

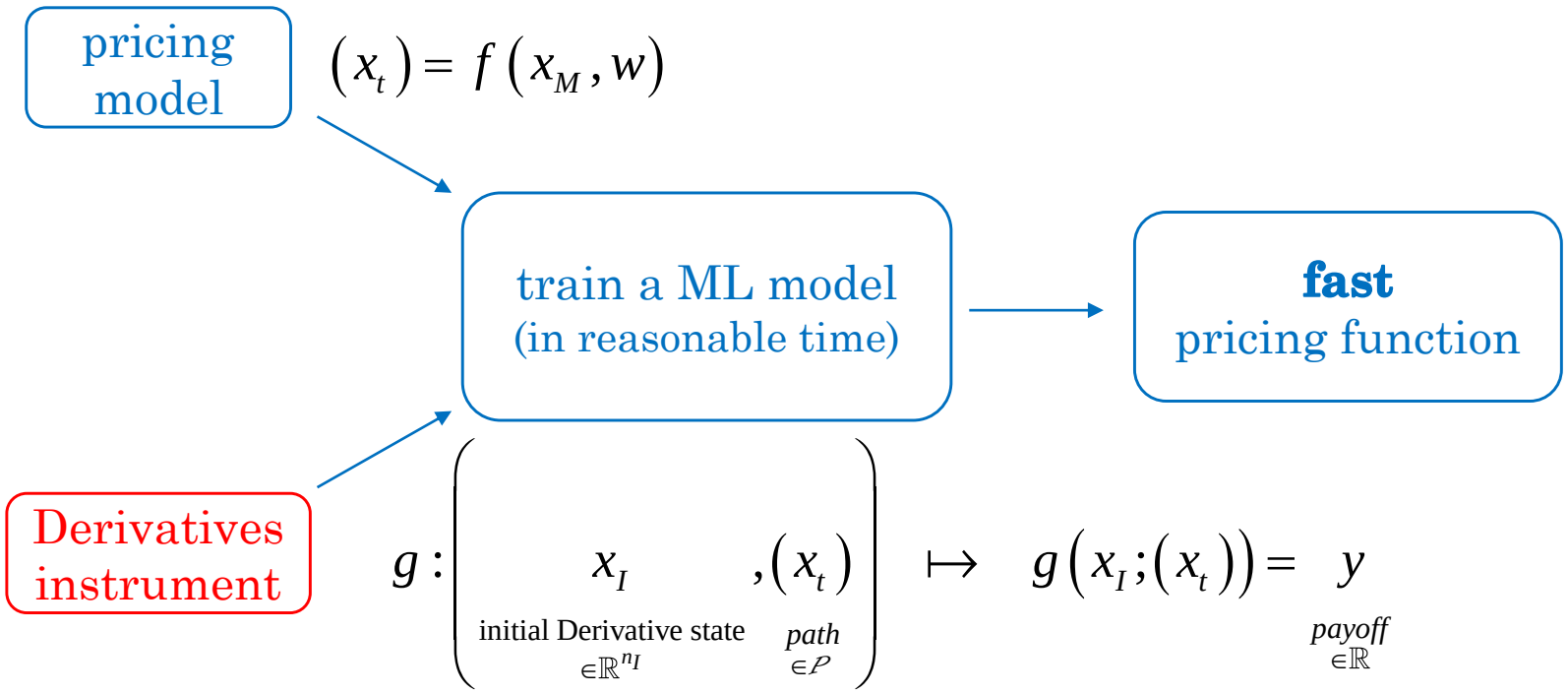
$state \in \{0,1\}$ or $[0,1]$ with smoothing

- In general, payoff depends on (future) path **and accumulated (past) dependencies**

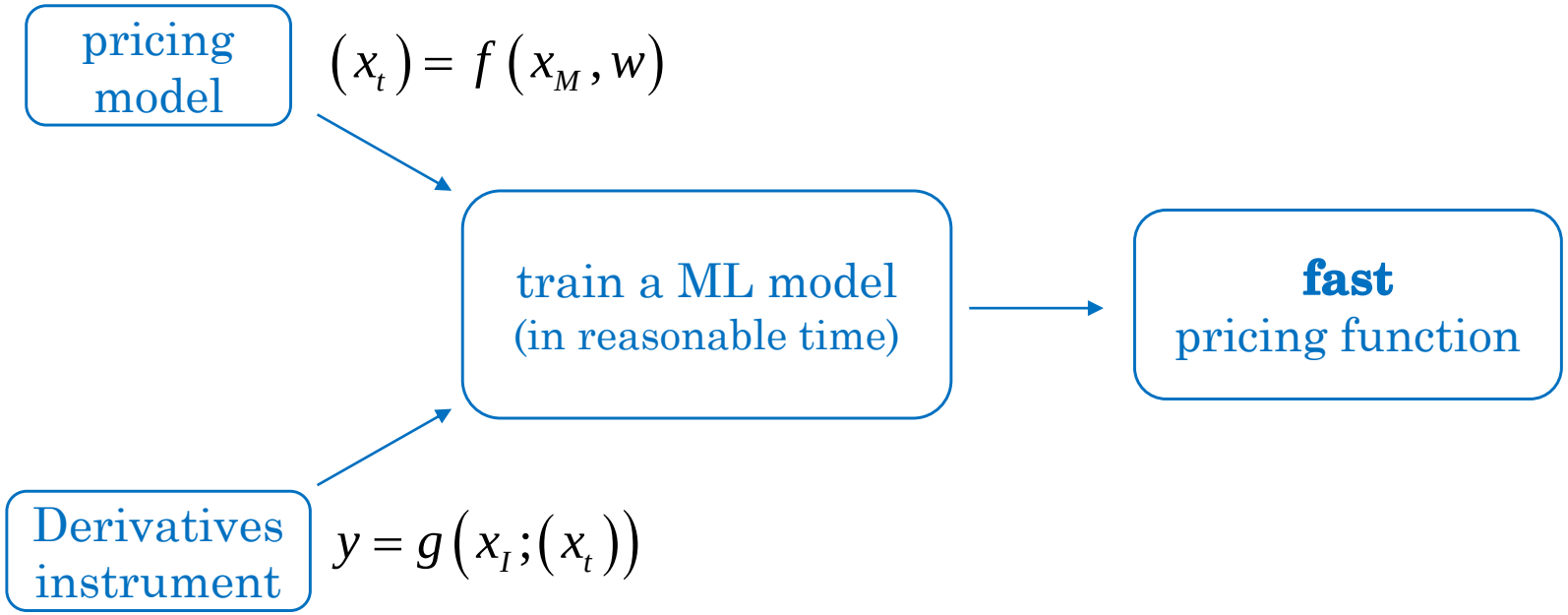
$$\begin{array}{rccc} & \mathbb{R}^{n_I} \times \mathcal{P} & \to & \mathbb{R} \\ g: & (x_I, (x_t)) & \mapsto & payoff \end{array}$$

- Where $x_I \in \mathbb{R}^{n_I}$ is the initial **transaction state,** i.e. all past dependencies that affect payoff
- And $n_I$ is its dimension, i.e. the number of necessary past dependencies

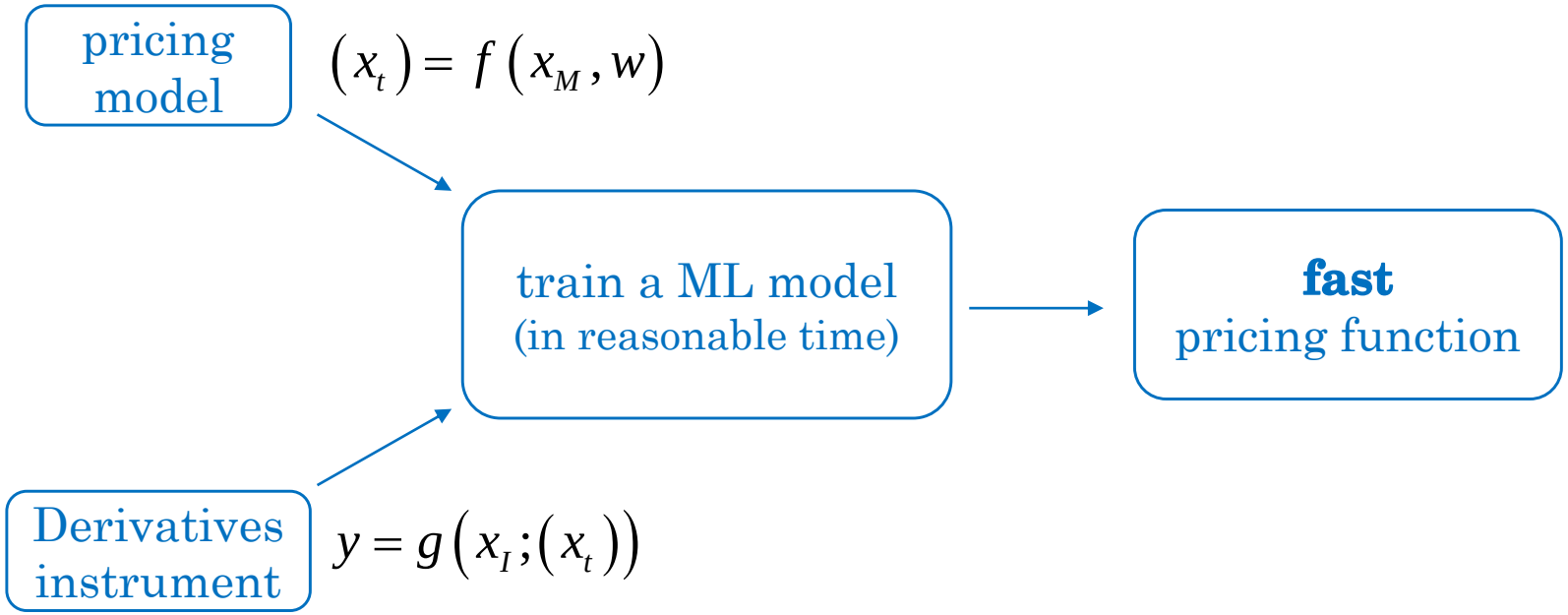
# Big picture


pricing model
$(x_t) = f(x_M, w)$
train a ML model (in reasonable time)
fast pricing function
Derivatives instrument
$g : \left( x_I \in \mathbb{R}^{n_I} \text{ initial Derivative state}, (x_t) \in \mathcal{P} \text{ path} \right) \mapsto g(x_I; (x_t)) = y \in \mathbb{R}$ payoff

# Big picture


pricing model
$(x_t) = f(x_M, w)$
train a ML model (in reasonable time)
fast pricing function
Derivatives instrument
$y = g(x_I; (x_t))$

# Big picture

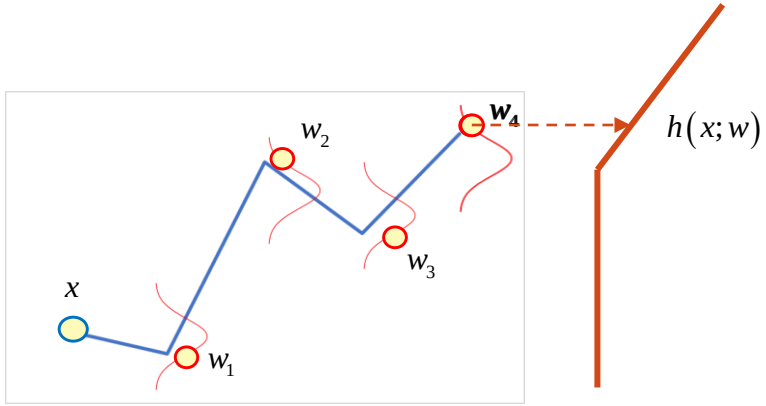



- The **composition** $y = h((x_M, x_I); w) = g(x_I; f(x_M; w))$ is called **pathwise payoff**
- $x = (x_M, x_I) \in \mathbb{R}^{n_M + n_I}$ is called **Markov state** and $n = n_M + n_I$ is the **Markov dimension**

# Pathwise payoff

- By definition: $y = h((x_M, x_I); w) = g(x_I; f(x_M; w))$
- In other terms, a model is a recipe for simulation of **payoffs:**

$$h(x;w) =$$



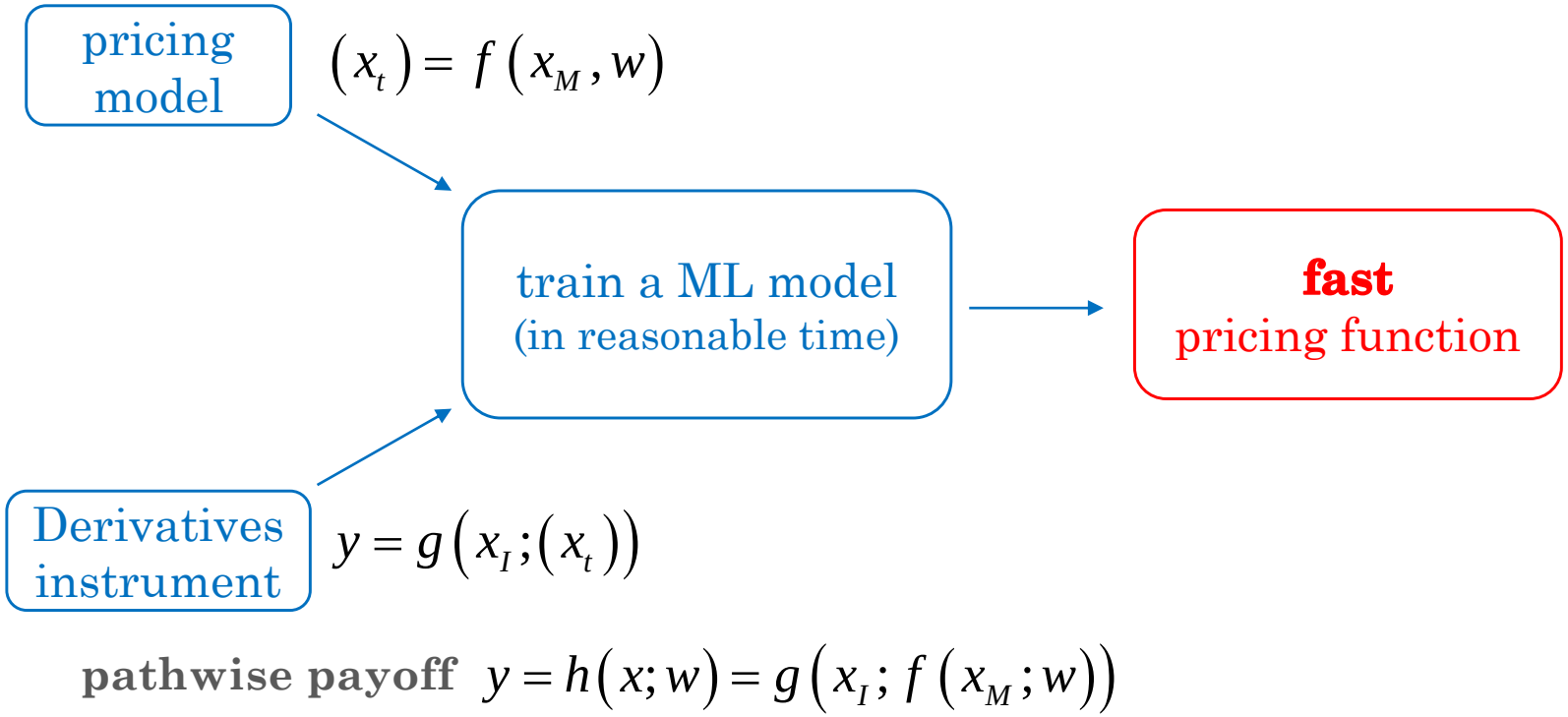

- Example: European call in Black & Scholes $h(x;w) = g(f(x;w)) = \left( x \exp\left[ -\frac{\sigma^2}{2} T + \sigma\sqrt{T} N^{-1}(w) \right] - K \right)^+$

# Example: Black and Scholes

- **European call**: $y = g((x_t)) = (x_T - K)^+$
- Black & Scholes model: $x_T = f(x;w) = x\exp\left[-\frac{\sigma^2}{2}T + \sigma\sqrt{T}N^{-1}(w)\right]$ N^(-1): inverse Gaussian CPD
- Pathwise payoff: $h(x;w) = g(f(x;w)) = \left(x\exp\left[-\frac{\sigma^2}{2}T + \sigma\sqrt{T}N^{-1}(w)\right] - K\right)^+$

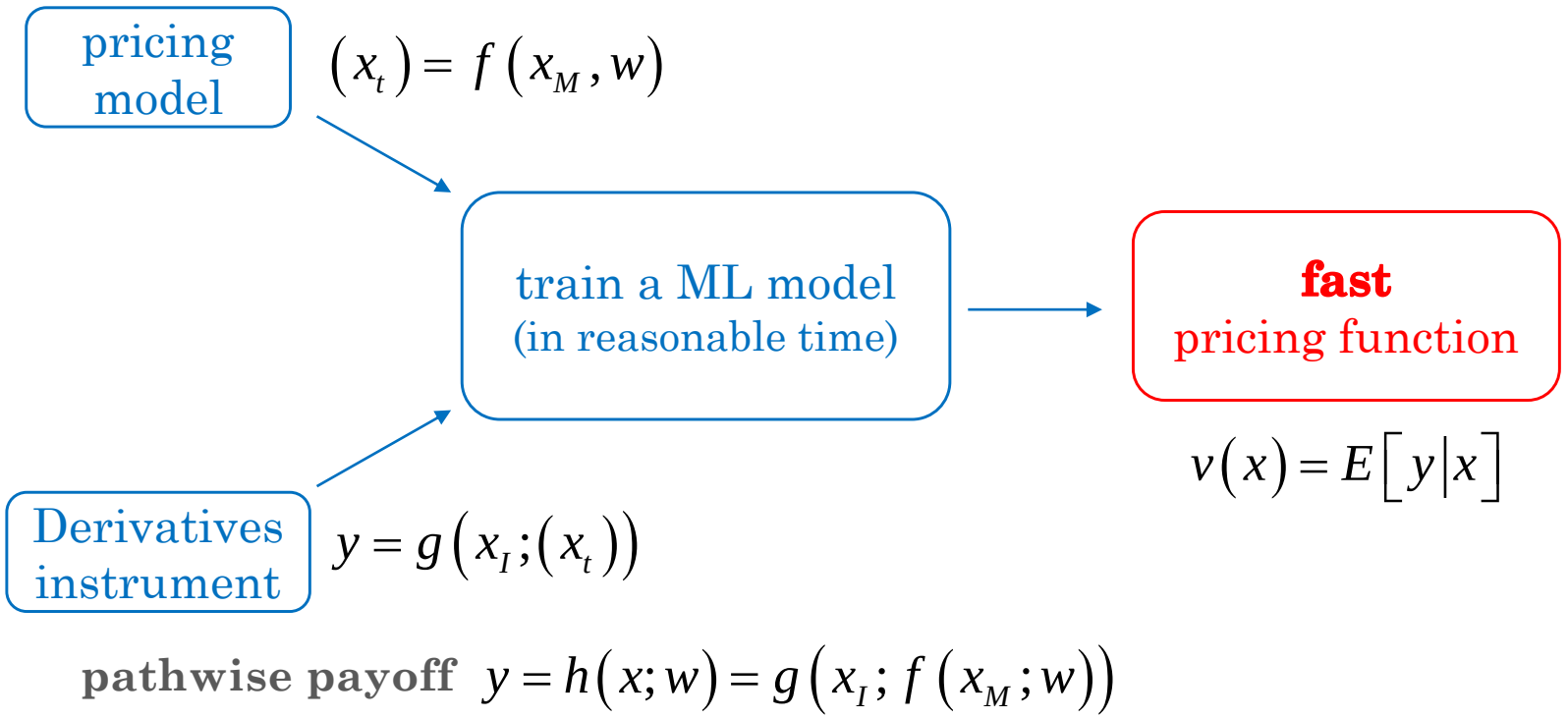
- **Barrier option**: discretely monitored barrier on dates $T_1, T_2, ..., T_p = T$
- Black and Scholes model: $f(x;w) = \left[x_0, x_{T_1}, x_{T_2}, ..., x_{T_p}\right]$ where $x_0 = x, x_{T_{i+1}} = x_{T_i}\exp\left[-\frac{\sigma^2}{2}(T_{i+1} - T_i) + \sigma\sqrt{T_{i+1} - T_i}N^{-1}(w_i)\right]$
- Barrier option: $g\left(alive, \left[x_0, x_{T_1}, x_{T_2}, ..., x_{T_p}\right]\right) = alive \cdot 1_{\left\{\max\left[x_{T_1}, x_{T_2}, ..., x_{T_p}\right] < Barrier\right\}} \cdot \left(x_{T_p} - K\right)^+$ alive: 1 if alive today, or 0
- Payoff: $h(alive, x; w) = g\left[alive, f(x;w)\right] = alive \cdot 1_{\left\{\max\left[x_{T_1}, x_{T_2}, ..., x_{T_p}\right] < Barrier\right\}} \cdot \left(x_{T_p} - K\right)^+$ where where $x_0 = x, x_{T_{i+1}} = x_{T_i}\exp\left[-\frac{\sigma^2}{2}(T_{i+1} - T_i) + \sigma\sqrt{T_{i+1} - T_i}N^{-1}(w_i)\right]$

# Big picture



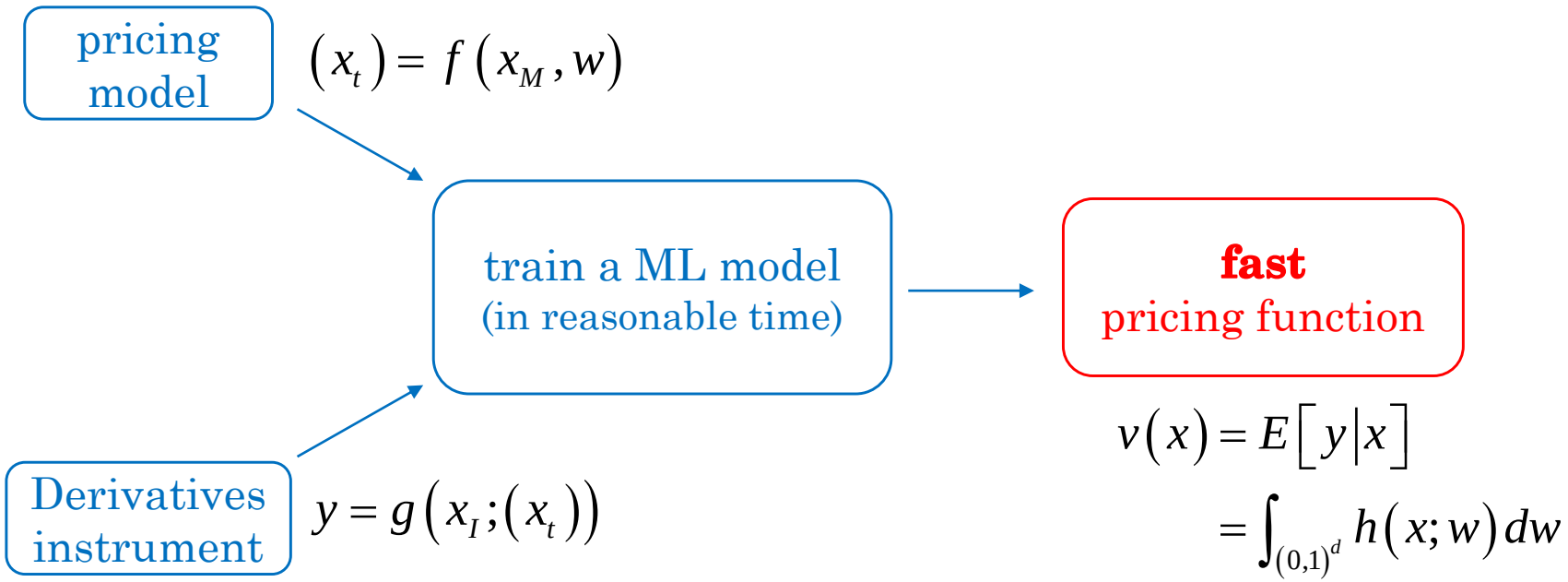

**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Big picture



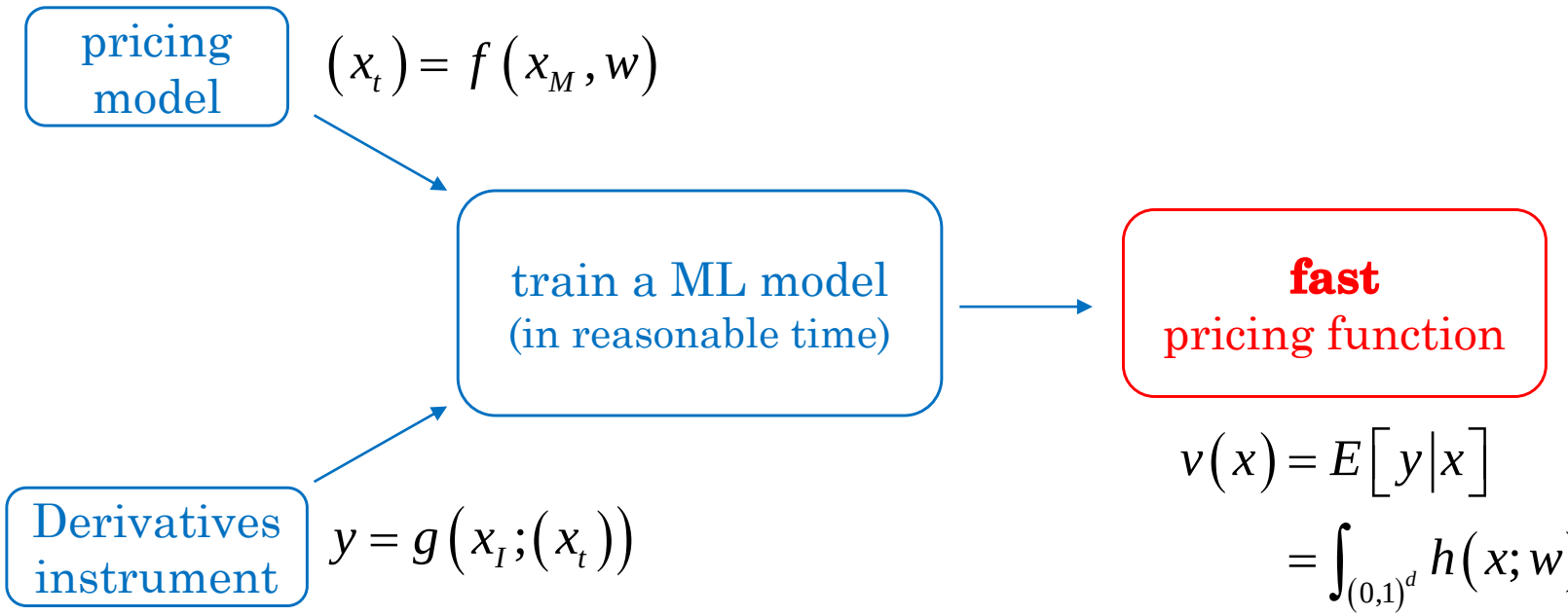

**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Big picture




**pathwise payoff** $y = h(x;w) = g(x_I; f(x_M;w))$

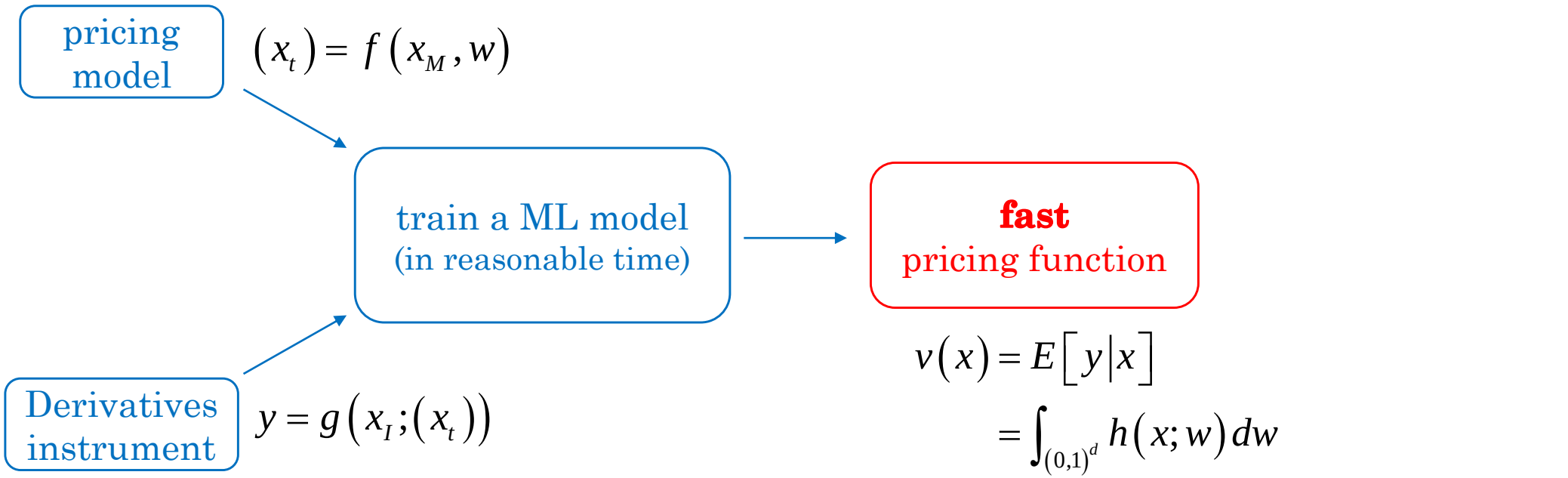
# Big picture



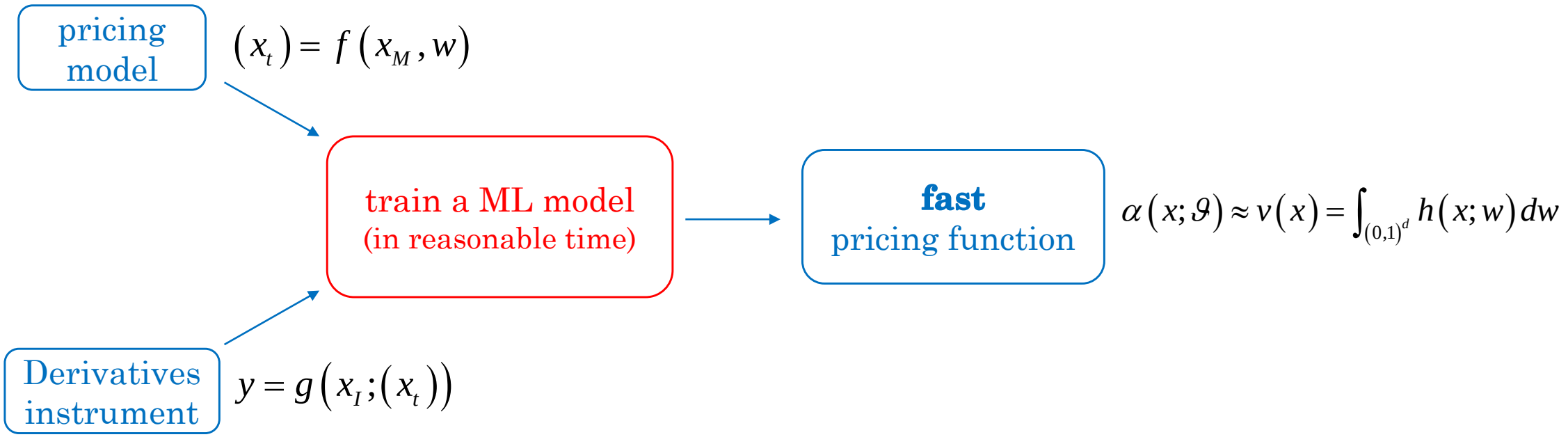

**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

$$v(x) = E[y|x]$$
$$= \int_{(0,1)^d} h(x; w)\, dw$$
$$\approx \frac{1}{N} \sum_{i=1}^{N} h(x; w_i)$$

# Big picture


pricing model
$(x_t) = f(x_M, w)$
train a ML model (in reasonable time)
fast pricing function
Derivatives instrument
$y = g(x_I; (x_t))$
pathwise payoff $y = h(x; w) = g(x_I; f(x_M; w))$
$v(x) = E[y|x]$
$= \int_{(0,1)^d} h(x; w)\, dw$
$\approx \frac{1}{N} \sum_{i=1}^{N} h(x; w_i)$
slow pricing by MC
large number of simulations
same initial state
random numbers sampled e.g. with Sobol

# Big picture




**pathwise payoff** $y = h(x;w) = g(x_I; f(x_M;w))$

# Big picture

pricing model

$(x_t) = f(x_M, w)$

train a ML model (in reasonable time)

**fast** pricing function

$\alpha(x;\vartheta) \approx v(x) = \int_{(0,1)^d} h(x;w)\,dw$

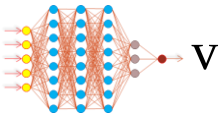
Derivatives instrument

$y = g(x_I;(x_t))$

**regression:** (Longstaff-Schwartz 2001)

$\alpha(x;\vartheta) = \sum_{k=1}^{K} \vartheta_k \varphi_k(x)$

**neural network:** (Lapeyre and al. 2020)

$\alpha(x;\vartheta) = x$  v

**pathwise payoff** $y = h(x;w) = g(x_I; f(x_M;w))$

# Big picture

pricing model

$(x_t) = f(x_M, w)$

train a ML model **in less than 10 MC**

**fast** pricing function

$\alpha(x;\vartheta) \approx v(x) = \int_{(0,1)^d} h(x;w)\,dw$

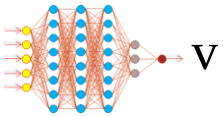
Derivatives instrument

$y = g(x_I;(x_t))$

**regression:** (Longstaff-Schwartz 2001)

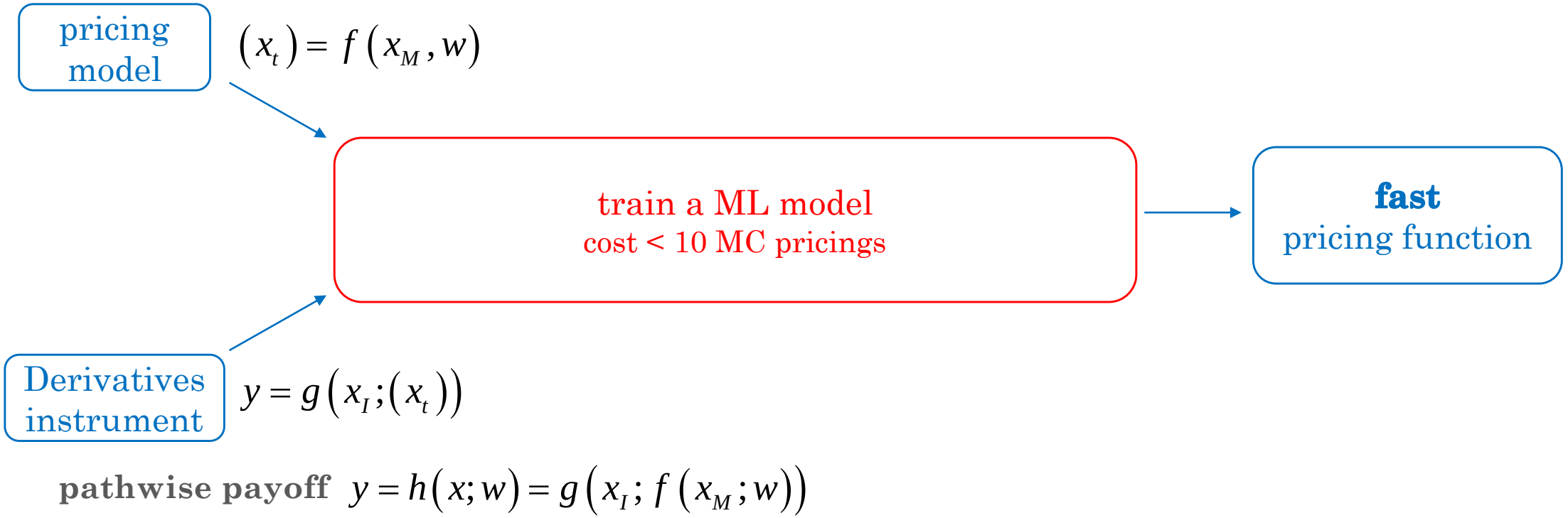
$\alpha(x;\vartheta) = \sum_{k=1}^{K} \vartheta_k \varphi_k(x)$

**neural network:** (Lapeyre and al. 2020)

$\alpha(x;\vartheta) = x$  v

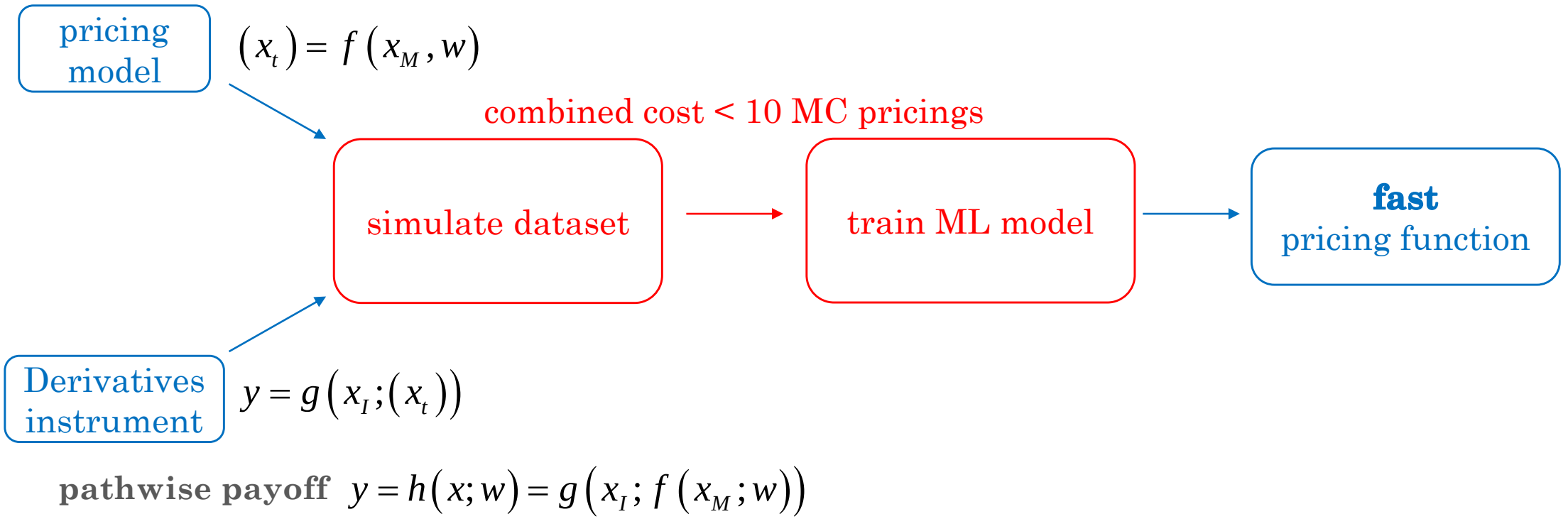
**pathwise payoff** $y = h(x;w) = g(x_I; f(x_M;w))$

# Zooming in



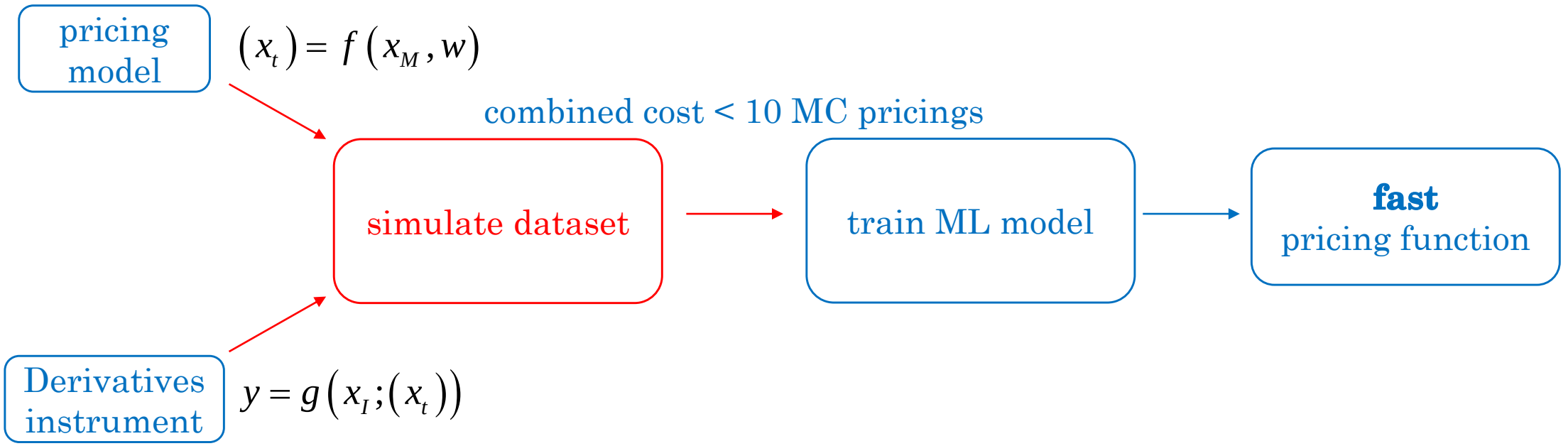

**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Zooming in




**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Zooming in

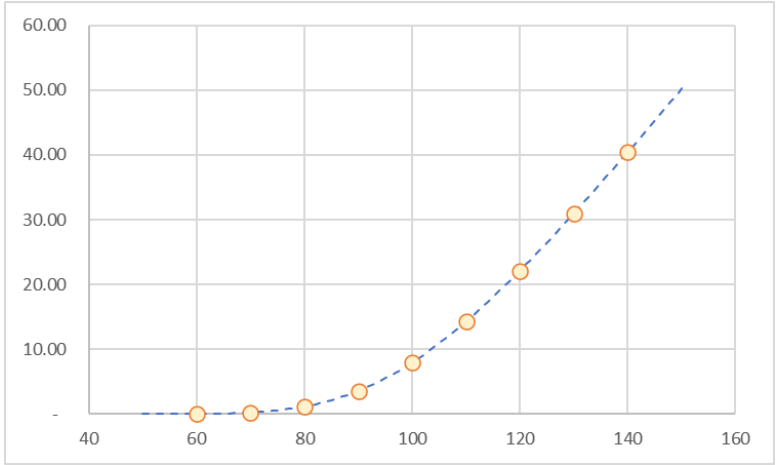



**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Classical supervised learning

- Learn function $v(x)$ x: initial state and v: price in that state
- From a training set of m examples $\left(x^{(i)}, y^{(i)} = v\left(x^{(i)}\right)\right)$

- This is an **interpolation** problem
- e.g. Horvath al al. (2019)
  in the context of rough volatility
- (Obviously) converges
  to the correct pricing function
- But requires many training examples
  (thousands, up to millions)
- Dataset alone takes computation cost of
  (at least) **thousands** of pricings by MC



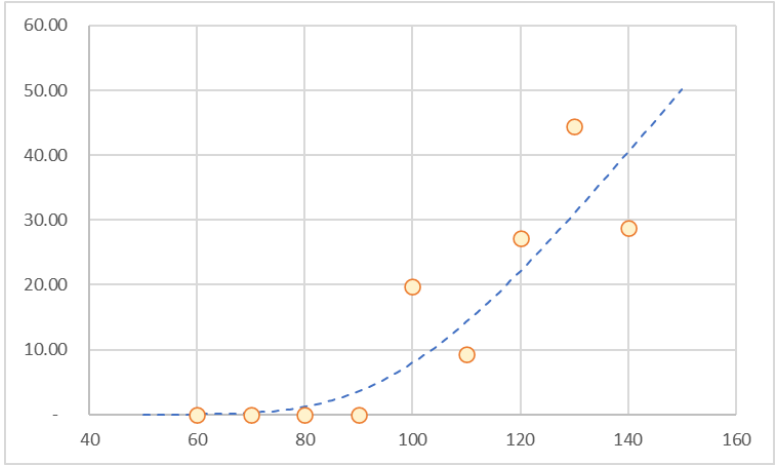

$$y^{(i)} = v\left(x^{(i)}\right) = \int_{(0,1)^d} h\left(x^{(i)}; w\right) dw \approx \frac{1}{N} \sum_{j=1}^{N} h\left(x^{(i)}; w_j\right)$$

- OK for **reusable** approximations, not **disposable** ones: cost of data is far too high

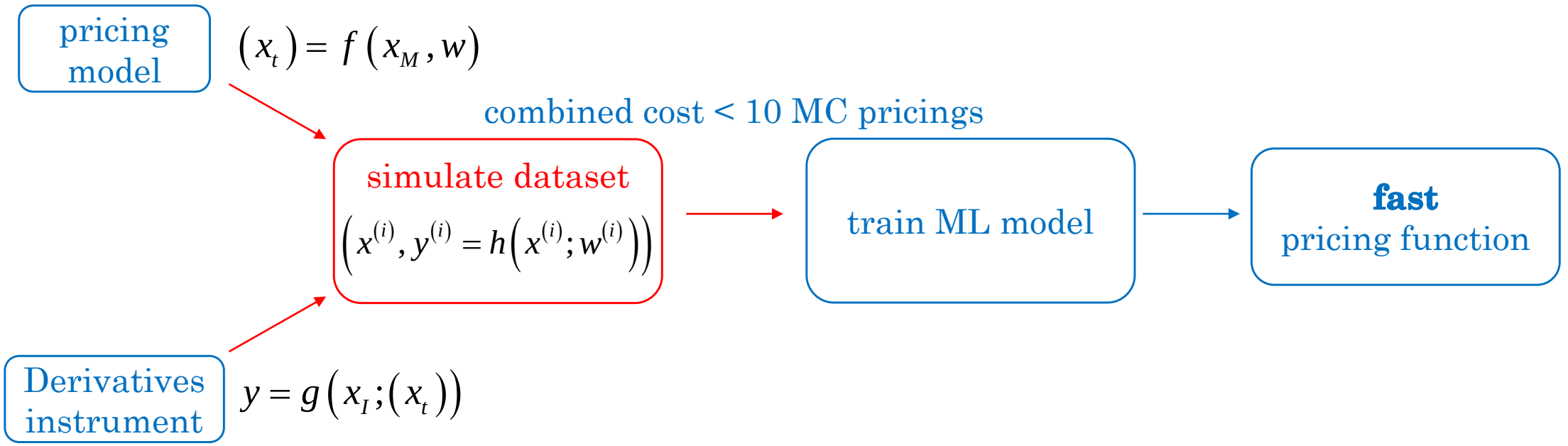
# LSM supervised learning

- Learn function $v(x)$ x: initial state and v: price in that state
- From a training set of m **payoffs** $\left(x^{(i)}, y^{(i)} = h\left(x^{(i)}; w^{(i)}\right)\right)$

- This is a **regression** problem
  in the spirit of the original LSM paper
- Where we learn **prices** from **payoffs**
- Still converges
  to the correct pricing function (proof next)
- Each training example computed for the
  cost of only **one** MC path
- The whole training set is computed for the
  cost of m paths, i.e. similar to **one** pricing by MC

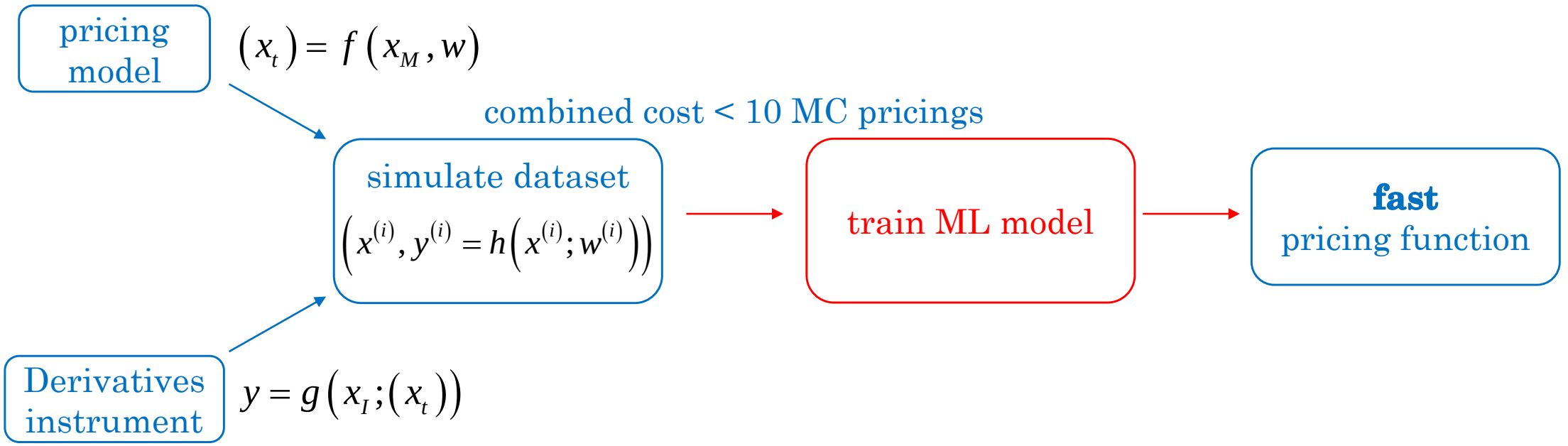


# Zooming in



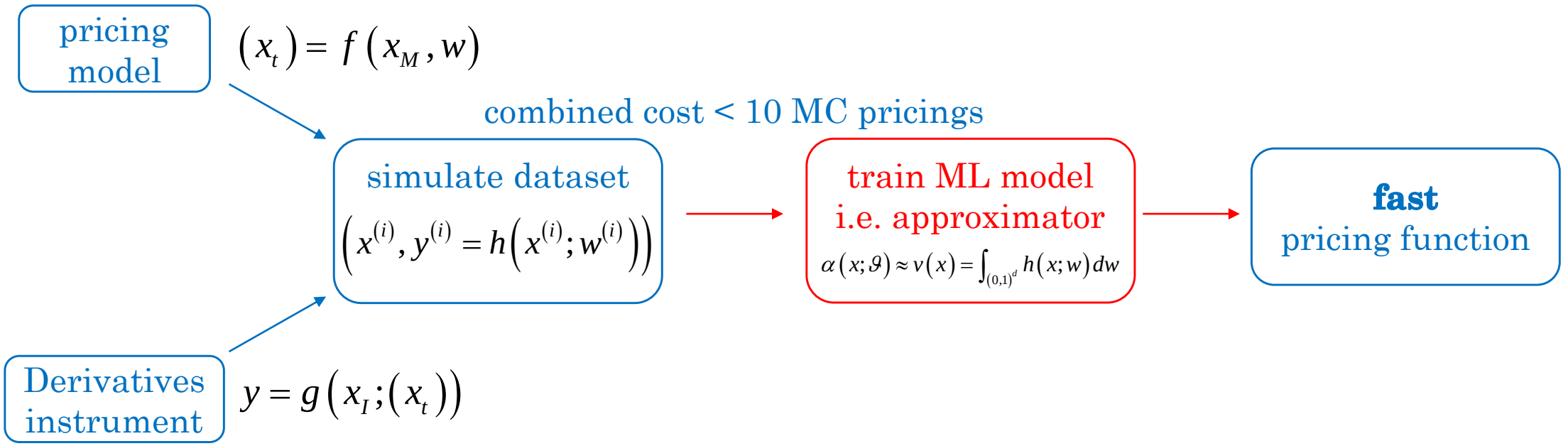

**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Zooming in




**pathwise payoff** $y = h(x; w) = g(x_I; f(x_M; w))$

# Zooming in




**pathwise payoff** $y = h(x;w) = g\left(x_I; f\left(x_M; w\right)\right)$

# Part II
# Conventional ML

Pricing with classic regression, PCA and neural networks

# Learning expectations

- Dataset: m independent realisations $\left(x^{(i)}, y^{(i)} = h\left(x^{(i)}; w^{(i)}\right)\right)$ of $\left(x, y = h\left(x; w\right)\right)$
- Goal: learn pricing function $\alpha\left(x; \vartheta\right) \approx v\left(x\right) = E\left[y|x\right]$
- By definition of the conditional expectation: $v\left(x\right) = E\left[y|x\right] = \underset{\text{all measurables functions } \alpha \text{ of x}}{\arg\min} \left\|y - \alpha\left(x\right)\right\|_2^2 = E\left[\left(y - \alpha\left(x\right)\right)^2\right]$

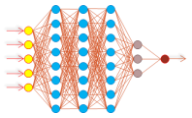
- Note that mean-squared error (MSE) is a proper estimator of L2 distance:

$$MSE\left(\alpha\right) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \alpha\left(x^{(i)}\right)\right)^2 \xrightarrow[m\to\infty]{} \left\|y - \alpha\left(x\right)\right\|_2^2$$

- Note that minimization criterion must be MSE and nothing else otherwise minimum does not converge to correct price, i.e. we get a **bias**

# Universal approximators

- Approximator $\alpha(x;\vartheta)$ is universal when it spans **all** functions of x
  when capacity (roughly, the dimension of learnable parameters theta) increases to infinity
- Polynomial regression is universal for smooth functions
  since all smooth functions are combinations of polynomials
- Other basis function regressions such as Fourier or spline basis are equally universal
- Universal Approximation Theorem (Cybenko, 1989):
  neural networks are also universal approximators



- UAT has been (over) hyped in the recent literature
- In the words of Leif Andersen: every approximator is (more or less) universal

# Side note: UAT and Carr-Madan

- As a side note, there is nothing mysterious about the UAT
- Consider a simple 1D ANN with one hidden ReLu layer, then: $\alpha(x;\vartheta) = \vartheta_0 + \sum_i \vartheta_i (x - K_i)^+$
- UAT tells that the ANN can represent any function
- i.e. all functions can be decomposed over the 'hockey stick' basis $\varphi_k(x) = (x-k)^+$
- We already knew that from Carr-Madan: all European profiles are combinations of calls
- Carr-Madan's formula even gives the weights theta_i explicitly
- This is the basis for e.g. the static replication of variance and CMS swaps

# Putting it all together

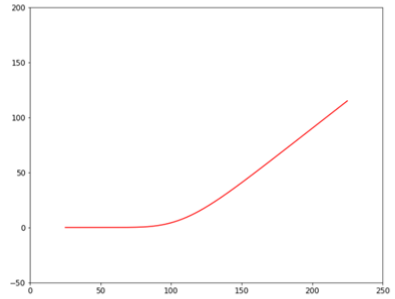
- Fundamental theorem of ML pricing

*A universal approximator trained by minimization of the MSE*
*over a training set of payoffs a la LSM converges to the correct pricing function*

$$\alpha\left(x;\arg\min MSE\left(\vartheta\right)\right)\xrightarrow[\substack{m\to\infty\\ \dim(\vartheta)\to\infty}]{}v\left(x\right)$$

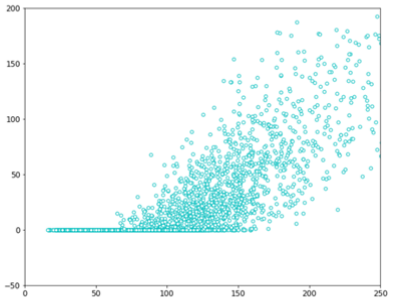
- Nontrivial and somewhat nonintuitive result that justifies pricing by ML:
*a ML model trained on payoff data converges to the correct pricing function*
*despite having never seen a price!*

learn this




$$E\left[\left(S_{T_2}-K\right)^+\middle|S_{T_1}\right]=S_{T_1}N\left(d_1\right)-KN\left(d_2\right)$$

from this

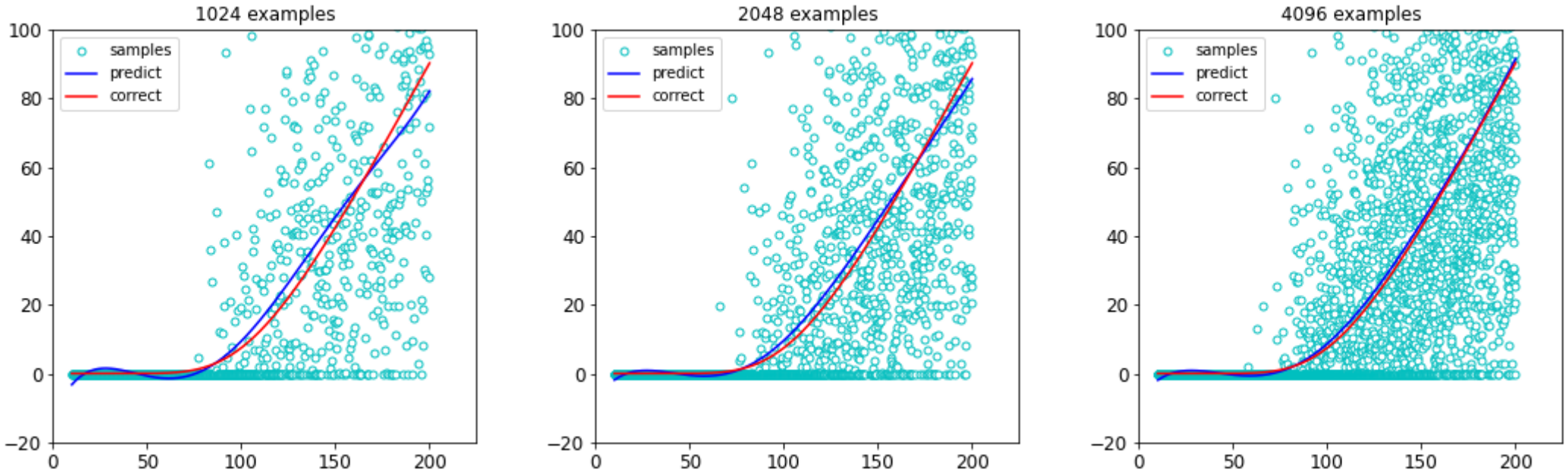



$$\left\{S_{T_1}^{(i)},\left(S_{T_2}^{(i)}-K\right)^+\right\}$$

- Note: you still need to find the (global) minimum of the MSE
- Easy for regression but NP-hard for neural networks

# Example: Black and Scholes

European call in Black & Scholes : degree 5 polynomial regression vs correct formula




source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

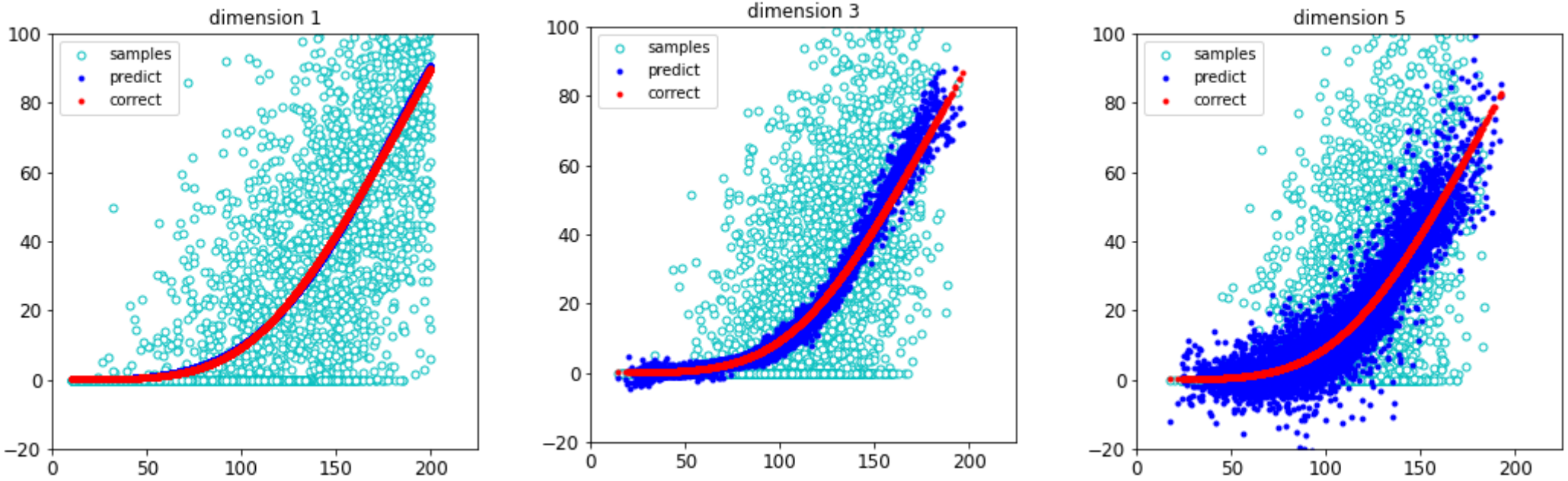
Open in Colab

# Example: Gaussian basket

- Black & Scholes is **not** a representative example
- Because its dimension is 1: x = {spot}
- Most situations in finance are high-dimensional
- e.g. LMM with 3m Libors over 30y: x = {all forward Libors}, dimension = 120
- With netting sets or trading books, dimension generally in the 100s or more
- A more representative example is the basket option in the correlated Bachelier model
  - Equally simple, with analytic solution
  - But, crucially, multidimensional x = {spot[1], spot[2], …, spot[n]}
  - Path generation: $x_T = f(x;w) = x + chol(\Sigma) N^{-1}(w)$
    chol: choleski decomp, sigma: covariance matrix and N^(-1) applied elementwise
  - Cashflow: $g(x_T) = \left(a^T x_T - K\right)^+$ a: (fixed) weights in the basket and K: strike
  - Payoff: $h(x;w) = g\left[f(x;w)\right] = \left(a^T\left[x + chol(\Sigma) N^{-1}(w)\right] - K\right)^+$
  - Analytic solution: $v(x) = \int_{(0,1)^n} h(x,w)\,dw = Bach\left(a^T x, \sqrt{a^T \Sigma a}; K, T\right)$ Bach: classic Bachelier formula

# Example: Gaussian basket

correlated Gaussian basket : polynomial deg 5 regression vs formula, 4096 training examples, independent test set




source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# Curse of dimensionality

- How many basis functions $\varphi_k(x)$ / free coefficients $\vartheta_k$ in the regression $\alpha(x;\vartheta)=\sum_{k=1}^{K}\vartheta_k\varphi_k(x)$ ?
- Consider a **cubic** regression
- In dimension 1: $v(x)\approx\vartheta_0+\vartheta_1 x+\vartheta_2 x^2+\vartheta_3 x^3$ 4 free coefficients
- In dimension 2: $v(x)\approx\vartheta_{00}+\vartheta_{01}x_2+\vartheta_{02}x_2{}^2+\vartheta_{03}x_2{}^3+\vartheta_{10}x_1+\vartheta_{11}x_1x_2+\vartheta_{12}x_1x_2{}^2+\vartheta_{20}x_1{}^2+\vartheta_{21}x_1{}^2x_2+\vartheta_{30}x_1{}^3$ 10 free coefficients
- In dimension n: (n+1)(n+2)(n+3)/6 free coefficients, grows in n^3
- Degree p: $\frac{(n+p)!}{n!\,p!}$ coefficients, grows in n^p
- With that many free coefficients, regression **overfits** the specific noise of the training set and ends up **interpolating** the noisy dataset
- Polynomial coefficient growth is not limited to polynomial regression
  it affects **all** flavours of basis function regression, such as Fourier or spline regression
- Curse of dimensionality makes (standard) regression
  practically inapplicable for real-world problems

# Mitigating the curse

- Classic means for overcoming the curse of dimensionality include:

1. Stopping the optimizer from interpolating the training set, i.e. **regularization**
2. Reducing dimension before regression, i.e. principal component analysis or **PCA**
3. Learning a limited number of "best" basis functions, i.e. artificial neural networks or **ANN**

- We (very) briefly review those methods (vastly covered in the ML literature) to see that:
  1. Regularization kind of works, at least in low dimension
  2. PCA does **not** work, it is unsafe and should never be used in this context
  3. ANN are effective for **offline** learning but not **online** learning as too many training examples are necessary

- And then, we will see how to effectively resolve all those problems

# Regularization

- Regression model: $\alpha(x;\vartheta)=\sum_{k=1}^{K}\vartheta_k\varphi_k(x)=\vartheta^T\varphi(x)$
- Mean-Squared error: $MSE(\vartheta)=\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}-\alpha\left(x^{(i)};\vartheta\right)\right)^2=\left\|y-\alpha(x;\vartheta)\right\|^2$
- Classic (unregularized) regression: $\vartheta^*=\arg\min MSE(\vartheta)=C_{\varphi\varphi}{}^{-1}C_{\varphi y}$
- Where covariances $C_{\varphi\varphi}=E\left[\varphi(x)\varphi(x)^T\right]\in\mathbb{R}^{K\times K}$ and $C_{\varphi y}=E\left[\varphi(x)y\right]\in\mathbb{R}^K$ are estimated over training data

- Tikhonov **regularized** (ridge) regression: penalize large weights $\vartheta^*=\arg\min\left[MSE(\vartheta)+\lambda\|\vartheta\|^2\right]$
- Analytic solution: $\vartheta^*=\left(C_{\varphi\varphi}+\lambda I_K\right)^{-1}C_{\varphi y}$
- Constrains the optimizer to prevent it from overfitting the minimum MSE on dataset

# Regularization: discussion

- Introduces **bias** $\vartheta^* = \arg\min\left[MSE(\vartheta) + \lambda\|\vartheta\|^2\right] \neq \arg\min MSE(\vartheta)$

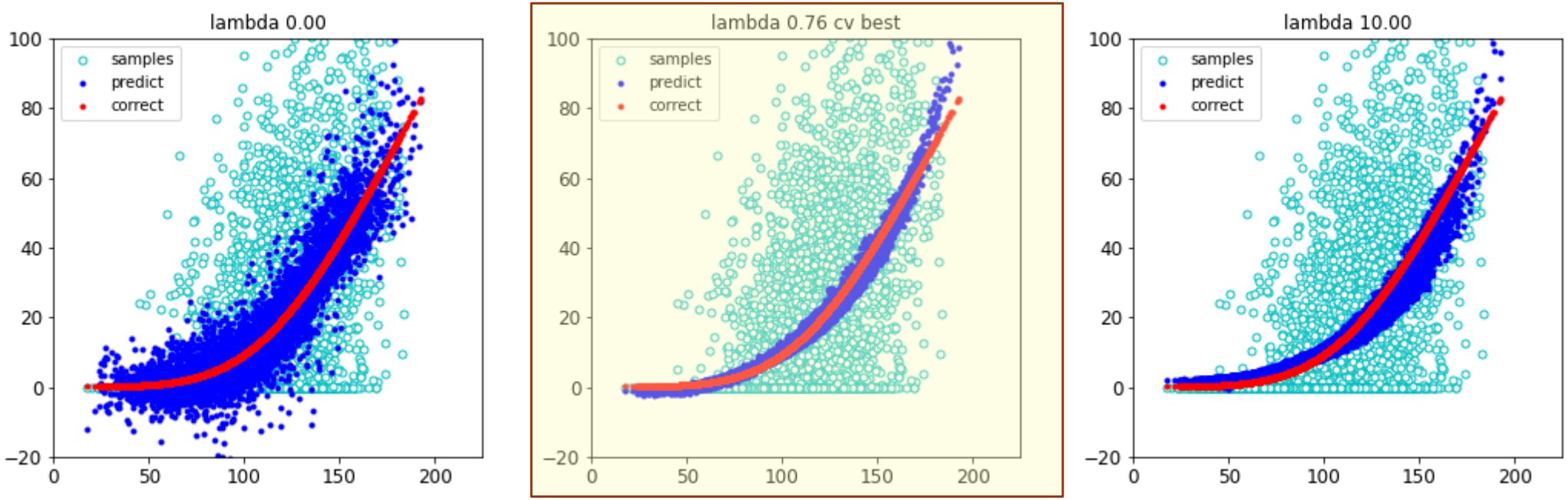
recall minimization of anything else than MSE does **not** converge to correct pricing

- Performance strongly depends on regularization strength lambda:
  - small regularization reverts to standard regression and overfitting
  - large lambda sets all coefficients to zero and always predicts average y
- called **bias/variance tradeoff**
- sweet spot must be found, generally by cross-validation (CV)

➢ expensive nested optimization and waste of data

# Regularization: performance

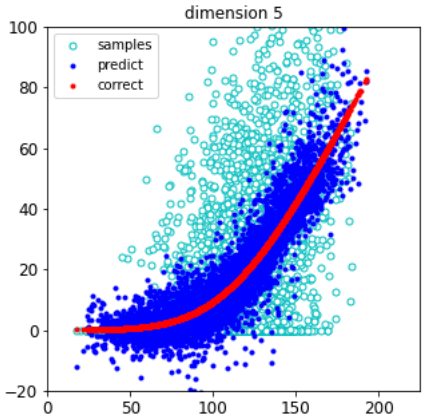
correlated Gaussian basket dim 5 : polynomial deg 5 regression vs formula, 4096 training examples, independent test set

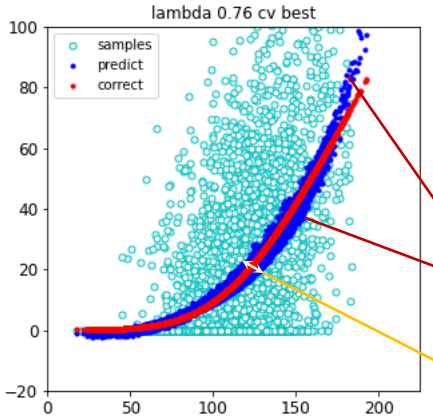



source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

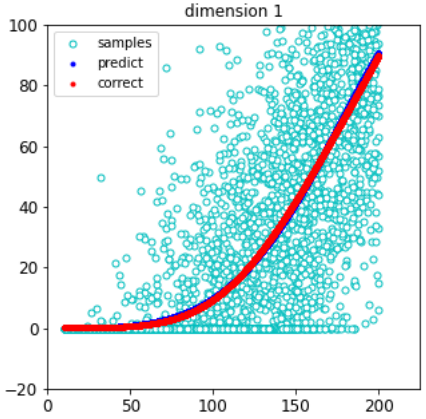
# Regularization: performance

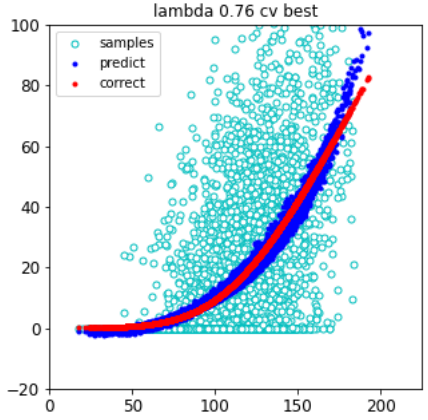



Compare with no regularization

- Significant improvement
  may be good enough depending on application
- Bias
- Variance is still significant




Compare with dimension 1

- Still significantly worse
- Frustrating since this is really a dim 1 problem:
  basket option only depends on initial basket,
  irrespective of the assets in the basket
  (at least in the Gaussian model here)

source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# Regularization in high dimension

- Recall normal equation: $\vartheta^* = \left(C_{\varphi\varphi} + \lambda I_K\right)^{-1} C_{\varphi y}$
- We must compute and invert a KxK matrix where $K = \frac{(n+p)!}{n!\,p!}$ p: polynomial degree
- For degree 5 K=6 in dim 1, 252 in dim 5, 3003 in dim 10 and 52,130 in dim 20
- Practically unfeasible in dimension above 10
- Use (variant of) gradient descent instead
  heavy and very expensive with cross-validation and nested optimization

# Regularization: LASSO

- A variant of ridge regression is LASSO, which penalises the L1 norm: $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \lambda \sum_{k=1}^{K} |\vartheta_k| \right]$
- (also biased and also optimized by iterative minimization nested in cross-validation)
- On the minimum, LASSO sets most weights theta_k to zero
- Effectively selecting a low number of basis function phi_k most relevant for a given problem
- We don't cover LASSO here because ANNs also perform feature selection, in a more effective, general and principled manner

# Derivatives regularization

- It is also common to penalize **large derivatives**: $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \sum_{p=1}^{P} \lambda_p \sum_{i=1}^{m} \left\| \frac{\partial^p \alpha\left(x^{(i)}; \vartheta\right)}{\partial x^{(i)p}} \right\|^{\gamma} \right]$

- e.g. penalize second order derivatives to model preference for linear functions

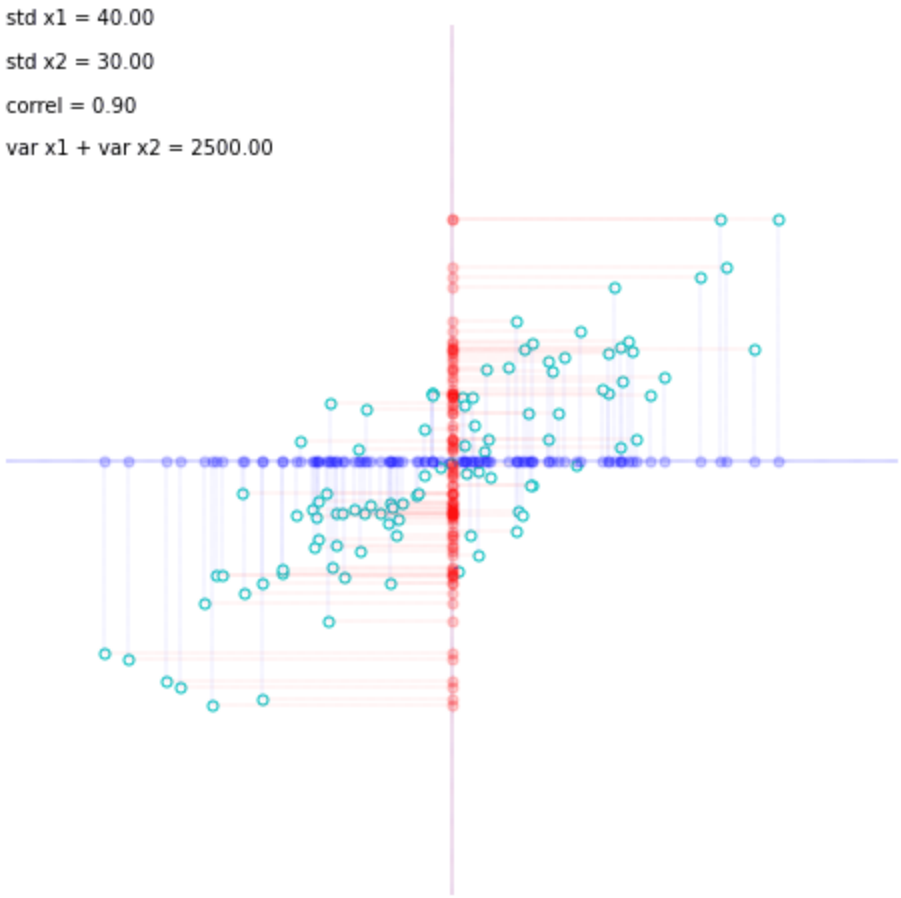
- **This is not at all the same thing as differential ML**
  - Differential ML penalizes **wrong** derivatives, not large derivatives (see next)
  - Derivatives regularization also introduces bias, differential ML does not

# Regularization: conclusion

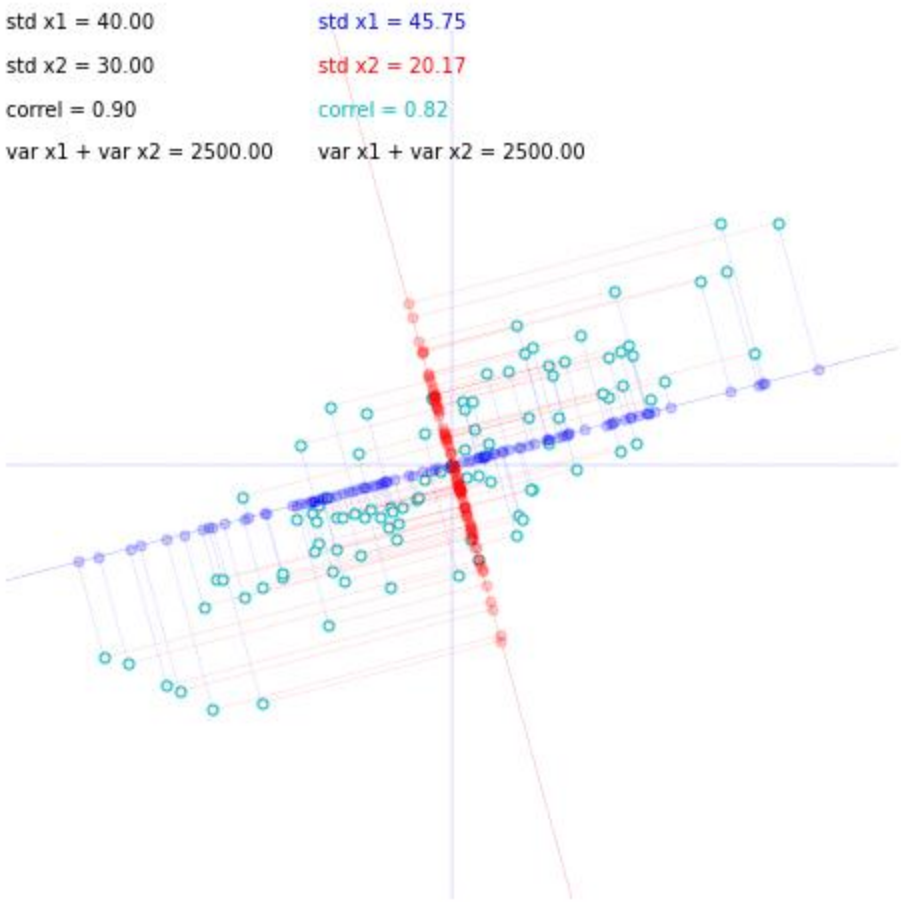
- Does a decent job in low dimension (but we can do a lot better)
- Introduces a bias by construction, forces cross-validation
- Impractical in high dimension

# Dimension reduction: PCA

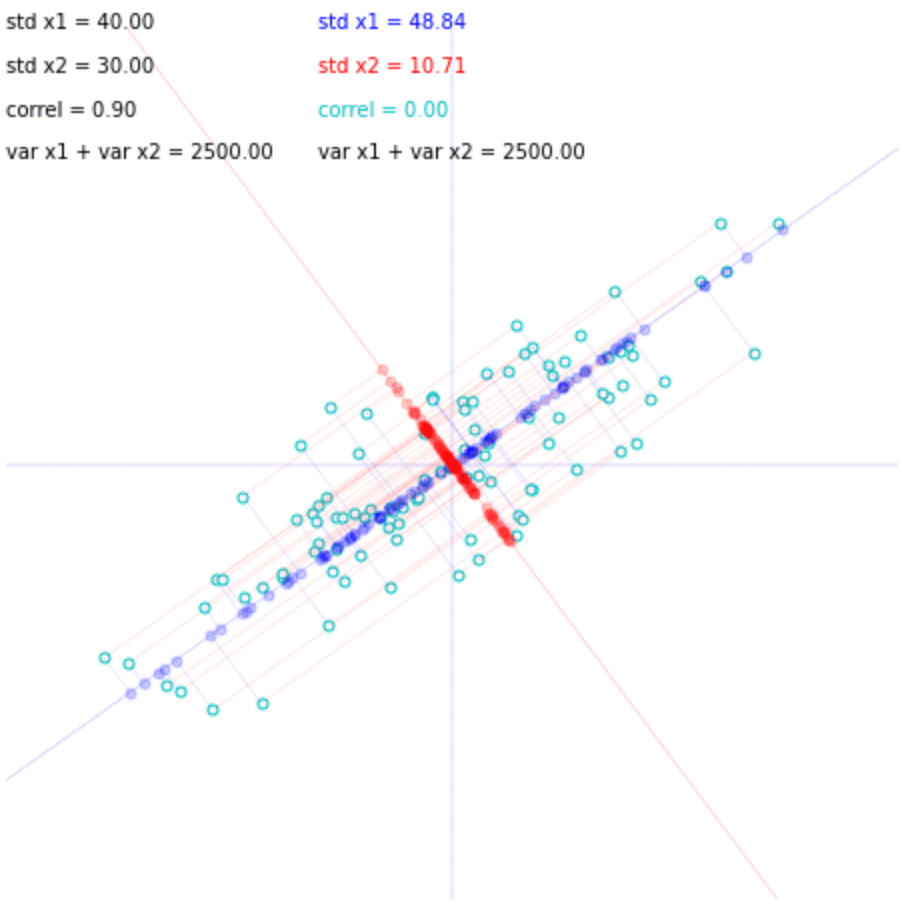
- Principle: reduce the dimension of x from n to p<n
- By 1) rewriting the n coordinates of x in a rotated basis where coordinates are uncorrelated
- i.e. eigenvector basis of the covariance matrix Cxx of x
- 2) Measuring the variance of the n coordinates of x in the rotated basis
- given by the eigenvalues
- And 3) dropping the coordinates with low variance
- Thereby locating x by p < n coordinates with minimal distance to original



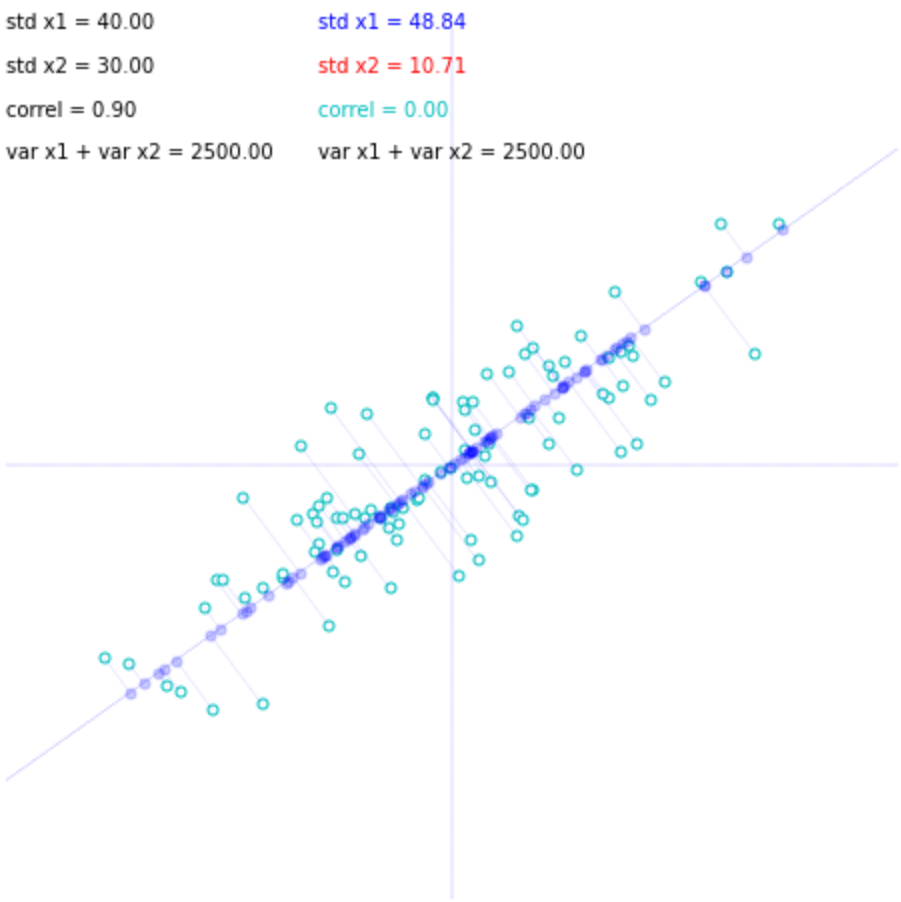

Two correlated assets x1 and x2 represented by points (x1, x2) in the standard basis, with a measure of covariances




Rotating the coordinate system changes x1 and x2 and modifies covariances but not total variance (var x1 + var x2)




In the eigenvector basis of Cxx, correlation is zero
and var x1 is maximized (biggest eigenvalue) while var x2 is minimized (lowest eigenvalue)

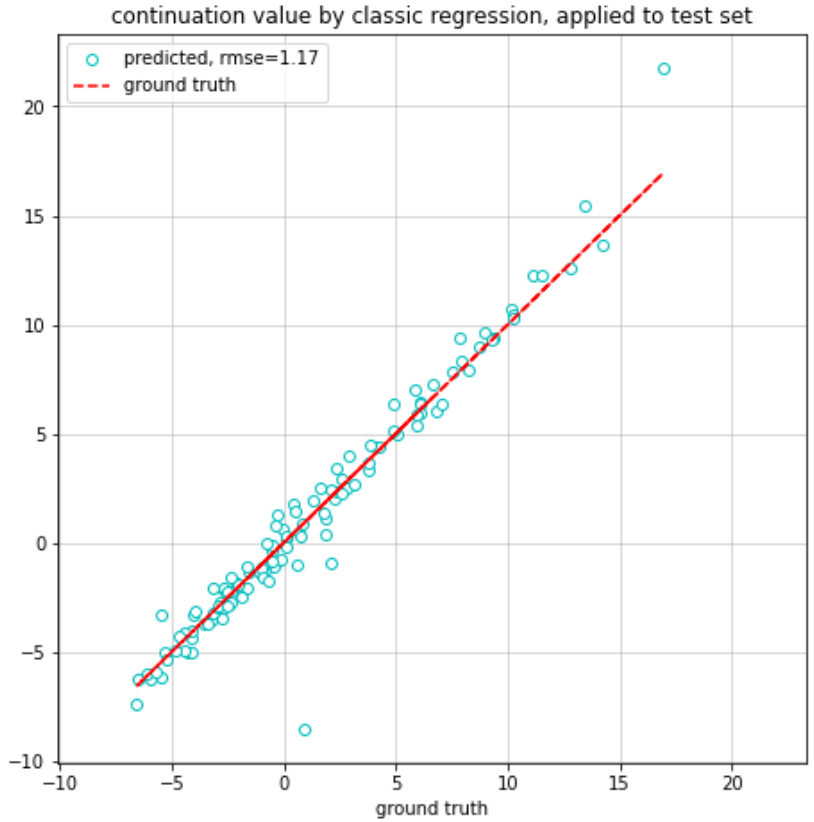



PCA reduces dimension by truncating red coordinates with low variation
i.e. projects data on axes of major variation = principal components (here the blue axis)
which also minimizes reconstruction error = distance between original and projected data

# PCA: performance

- Another example: continuation value of twice callable Bermudan option
  i.e. swap + swaption
- 5F Gaussian model (5F multi-factor Cheyette with normal volatility)
- Dataset simulated in Danske Bank's Superfly
  - x: initial state of the curve in dimension 5
  - y: sum of discounted cash-flows until maturity or early termination
- Source:
  https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb
- All details in the colab notebook – run it in your browser (or iPhone) 

# Bermudan option: 5F regression

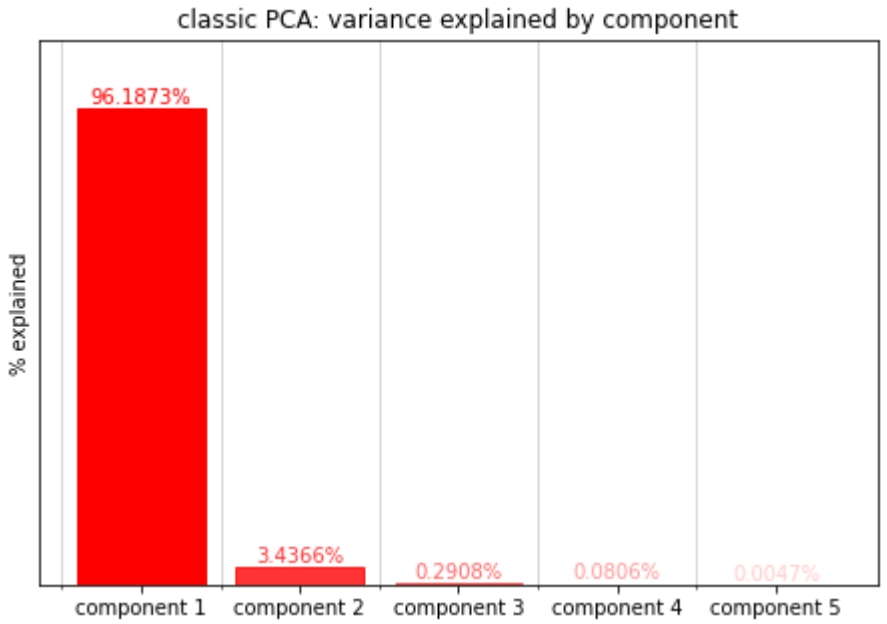



source:
https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb

Open in Colab

- Polynomial (degree 5) regression
- 8192 training examples, dimension 5
- Performance measured over 128 independent test examples
- Horizontal axis: ground truth price
- Vertical axis: prediction
- Error = distance to diagonal
- Sensible overall but large errors
- Could be marginally improved with regularization
- **Here: try to reduce dimension by PCA**

# Bermudan option: PCA

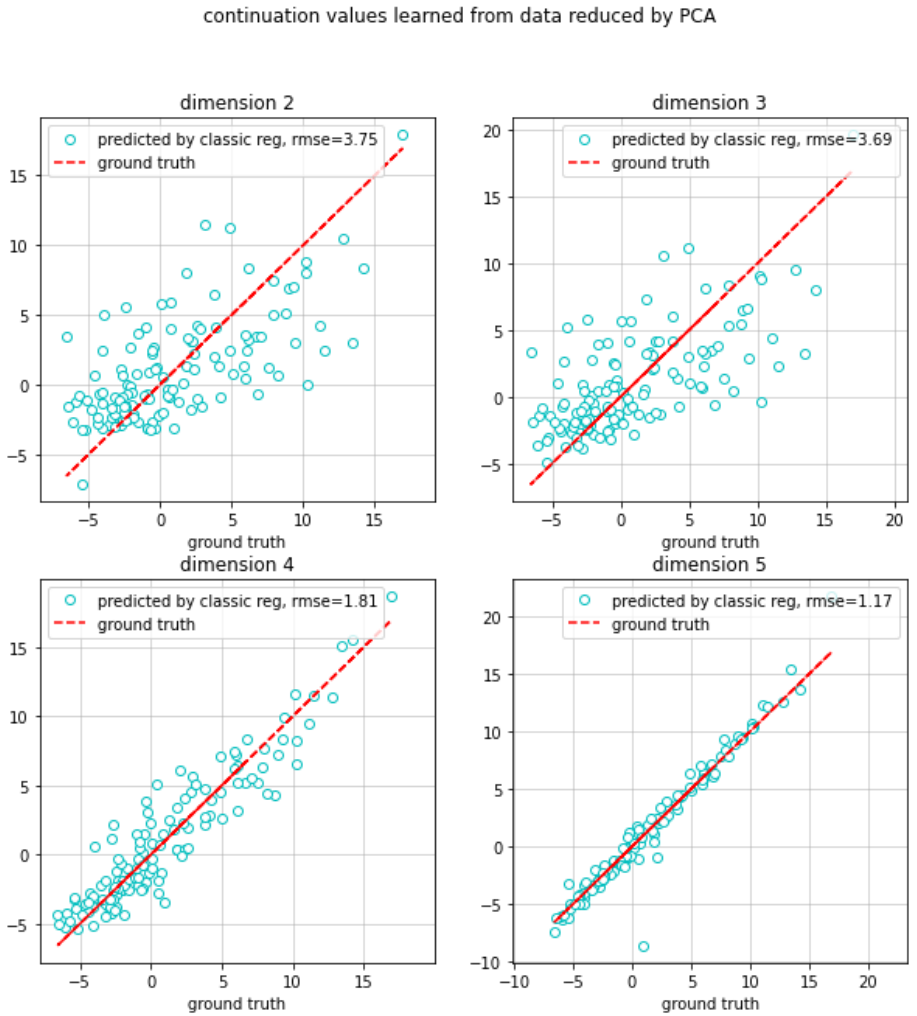



- Vast majority of variation over the two first factors
- Negligible variation over the two last factors
- Training in lower dimension:
  - Project training data on the first p factors
  - Regress over projected data
- Subsequent prediction:
  - Project test data on the p factors
  - Evaluate learned function of projected data

source:
https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb

Open in Colab

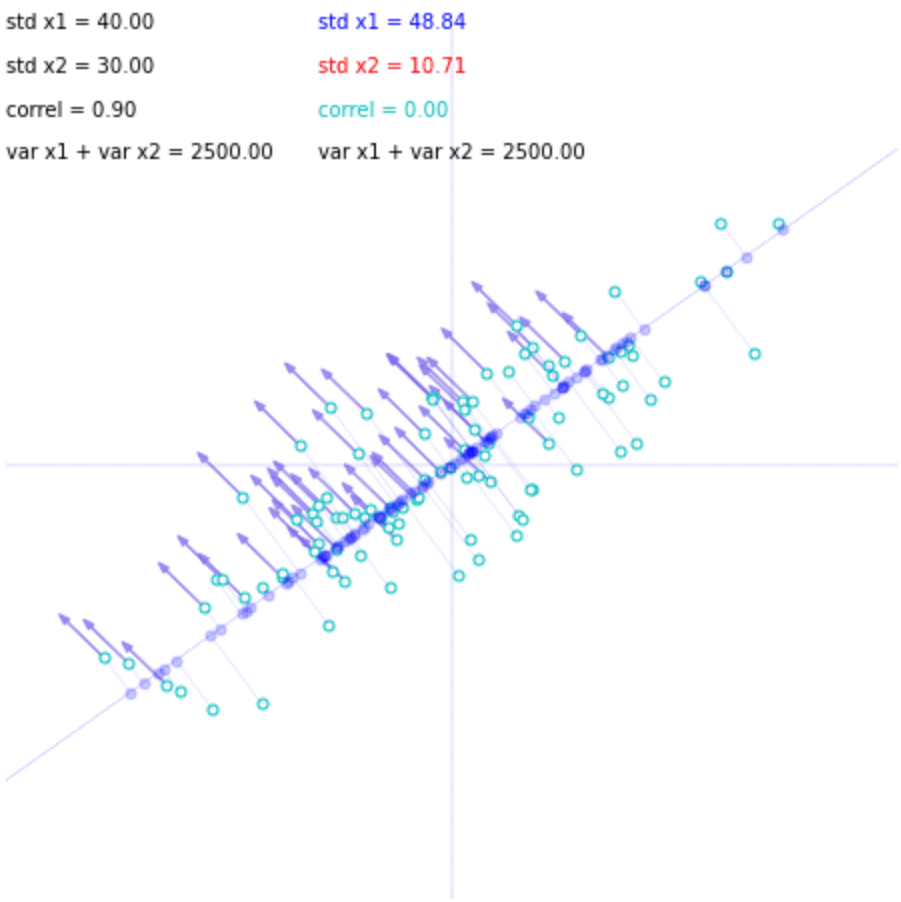
# Bermudan option: regression in reduced dimension




- **Terrible performance**
- No only in dimension 2, but also 3 and even 4!
- PCA truncated relevant information from data
- Despite negligible variation
- Accurate prediction no longer possible
  By regression or anything else
- See colab notebook for a detailed analysis of the Bermudan example
- Here, we explain inadequacy of PCA with a simpler illustration

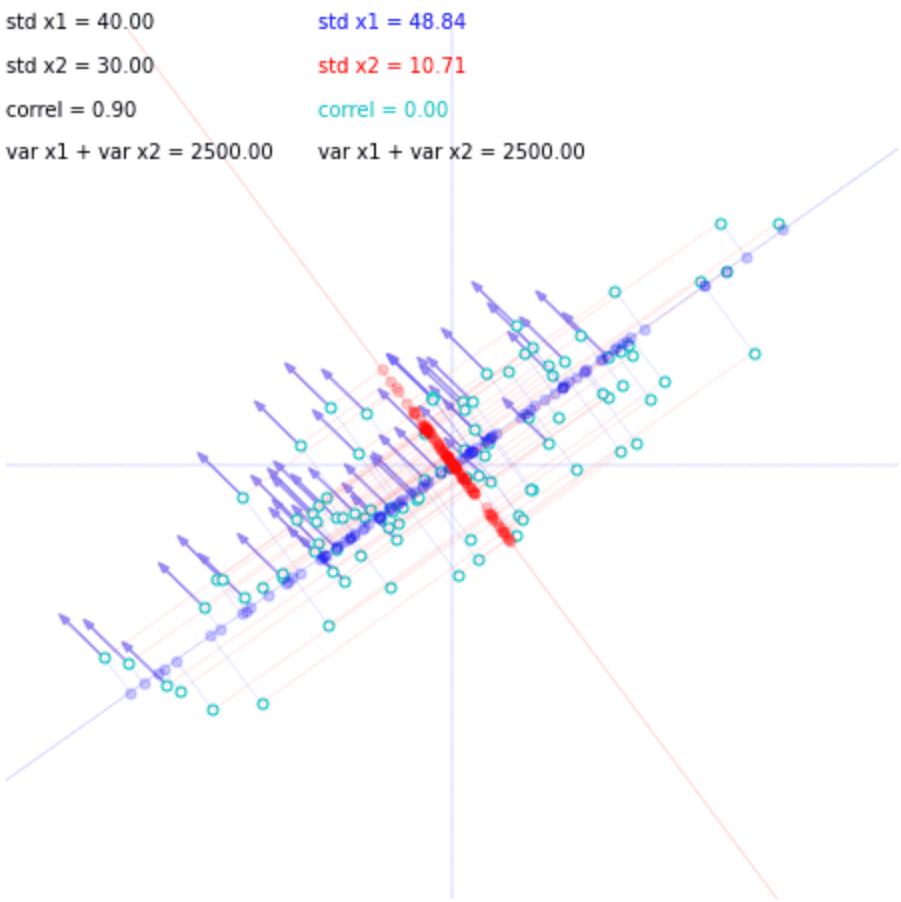
source:
https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb

Open in Colab



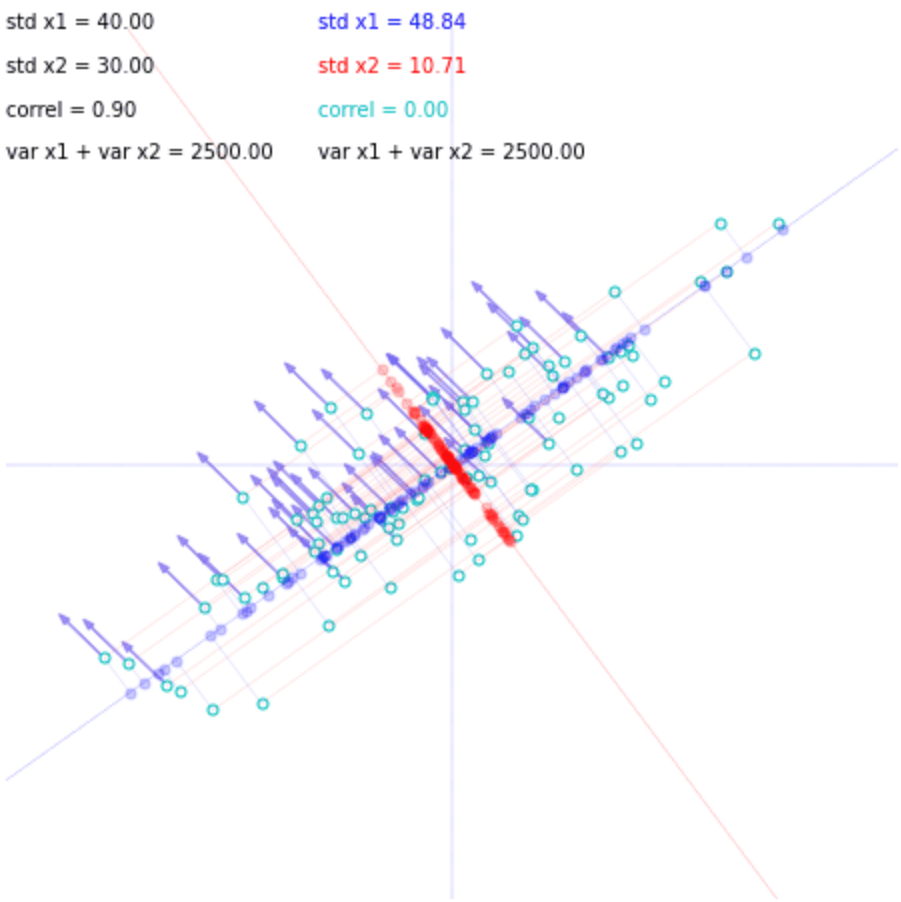

Consider a spread option on x2 - x1

Under Gaussian assumption, only depends on the underlying spread, so gradients are all aligned on the anti-diagonal




Gradients are aligned with the truncated axis

This axis may have low variation but for the spread option this is where all the action takes place




In fact, gradients are almost identically zero on the blue axis:
the spread option does not depend at all on the blue coordinate, only on the red one that PCA truncated!

➔ PCA truncation lost all hope to correctly evaluate price or risk

# PCA is one-size-fits-all

- PCA is a completely **unsupervised** algorithm
- PCA works with states x irrespective of labels y
- Hence, dimension reduction performed without regard any notion of cashflow, value or risk
- In particular, the basis is the same for all transactions or trading books

- For example, PCA does the same thing for
  - A basket option $\left(\sum_{i=1}^{n} a_i S_i - K\right)^+$ which is a known one-factor problem (at least under Gaussian assumption)
  - Or a basket of options $\sum_{i=1}^{n} a_i \left(S_i - K\right)^+$ where dimension in general cannot be reduced to one

# PCA is unsafe

- PCA may truncate **very relevant** coordinates when relevance doesn't align with variation

- Because PCA is one-size-fits-all
- For **any** one-size-fits all truncation
  There exists Derivatives which value and risk will be missed
- Consider a transaction with value > 0 in the truncated space, 0 otherwise

- Hence, a **safe** dimension reduction must be **supervised**
- PCA should never be applied for pricing or risk
- We propose a guaranteed safe, effective and efficient alternative called 'differential PCA'

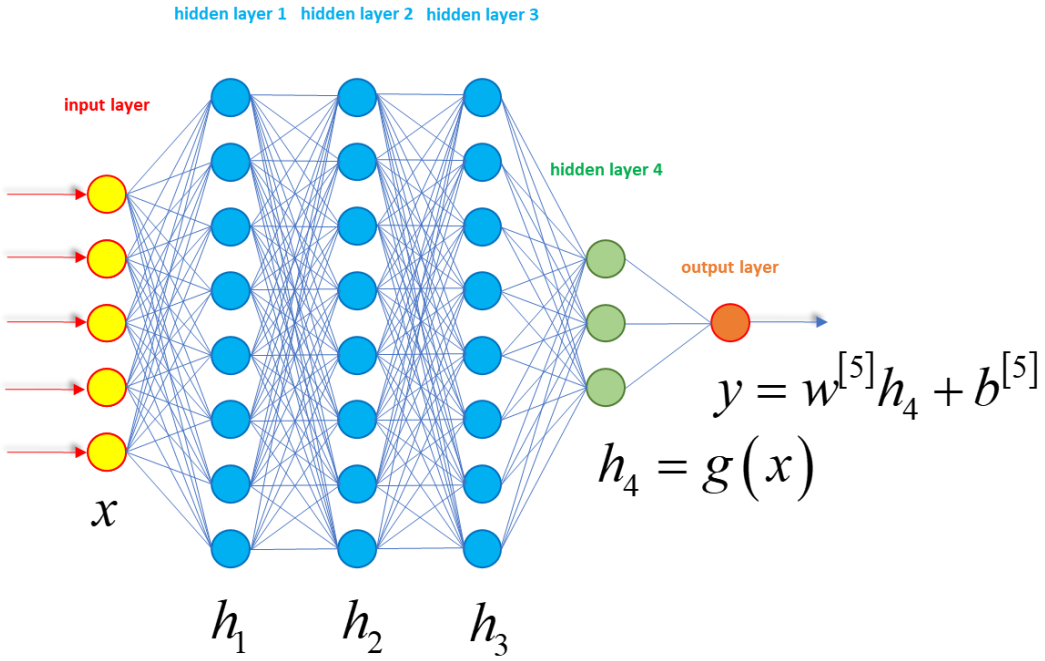
# Feedforward neural networks

- Artificial neural networks (ANN) generalize regression
- Unlike regression, ANN don't regress on a **fixed** set of basis functions
- Instead, they **learn** basis functions and encode them in the **hidden connection weights**
- **Hence, bypassing curse of dimensionality**
- Feedforward networks (FNN, also called multi-layer perceptrons or MLP)
  - (Universally) approximate functions y = v(x) with the sequence of **feedforward** computations:

$$\alpha(x;\vartheta) = \begin{cases} x^{[0]} = x \in \mathbb{R}^n & \text{input layer} \\ x^{[l]} = W^{[l]} g^{[l-1]}\left(x^{[l-1]}\right) + b^{[l]} \in \mathbb{R}^{n_l} & \text{hidden layers} \\ v = x^{[L]} \in \mathbb{R} & \text{output layer} \end{cases}$$

  - where $g^{[l]}$ is the nonlinear activation, scalar and applied elementwise e.g. $relu(x) = x^+$ or $softplus(x) = \log\left(1+e^x\right)$
  - and the learnable parameters are $\vartheta = \left\{\left(W^{[l]}, b^{[l]}\right), 1 \le l \le L\right\}$ where $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ are the connection weights and $b^{[l]} \in \mathbb{R}^{n_l}$ are the biases

# Feedforward neural networks




*g* encoded in the hidden connection weights

- capacity = dim(theta) = number of connection weights grows **linearly** in n = dim(x)
- ➢ escapes curse of dimensionality!
- The output layer is special: scalar and not activated
- ➢ it is a linear regression!
- where regression functions are the units of h4
- which are functions of x encoded in the hidden layers
- and (universally) represented by connection weights
- hence, learned from data
- ➢ You don't decide the basis functions a-priori
  - You decide how many of them you want
  - And the ANN learns the "best" ones to minimize MSE

# ANN: training

- Training = find the set of weights theta that minimizes MSE $\vartheta^* = \arg\min\left[MSE(\vartheta) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \alpha\left(x^{(i)};\vartheta\right)\right)^2\right]$
- ANN generally trained by **gradient descent**: $\vartheta_{i+1} = \vartheta_i - \lambda\frac{\partial MSE(\vartheta_i)}{\partial \vartheta_i}$
- (or a more effective variant like stochastic gradient descent (SGD), RMS-Prop or ADAM)
- With guarantee of finding a **local** minimum – finding the global min is NP-complete
- Where the gradients dMSE / dTheta are **efficiently** computed by **backpropagation**:

$$\bar{v} = \overline{x^{[L]}} = \frac{\partial MSE}{\partial v}$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

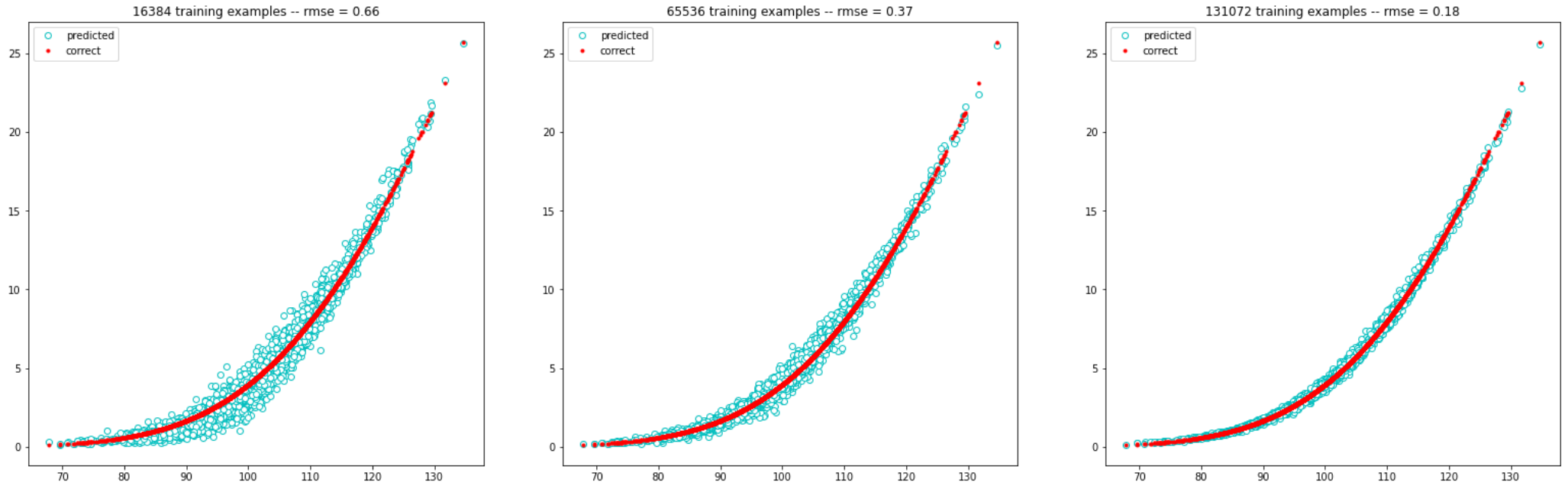
$$\frac{\partial MSE}{\partial W^{[l]}} = \overline{x^{[l]}} g^{[l-1]}\left(x^{[l-1]}\right)^{\mathrm{T}} \text{ and } \frac{\partial MSE}{\partial b^{[l]}} = \overline{x^{[l]}}$$

with the notation $\overline{thing} = \frac{\partial MSE}{\partial thing}$

- Backpropagation is the most critical technology in deep learning
- Explained in depth in a vast amount of books, articles, blogs, video tutorials and classes
- (Assumed known in this presentation)

# ANN: pricing performance

correlated Gaussian basket **dim 30** : ANN vs formula on independent test set




source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialMLTF2.ipynb

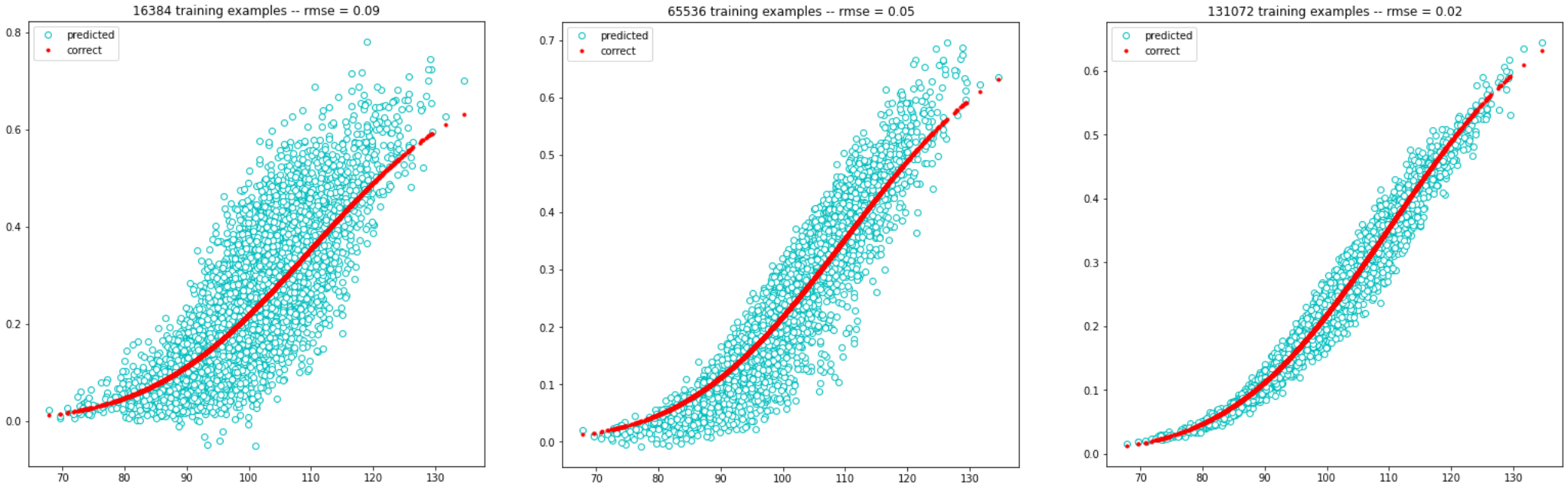
Open in Colab

# ANN: pricing performance

- Dimension 30 out of reach for regression
- ANN have no difficulty
- But they consume a vast number of training examples
- We need 131,072 examples for an accurate approximation
- This is confirmed in the literature, in finance and elsewhere
- See e.g. Horvath and al. (2019) or McGhee (2018) – training on millions of **price** examples
- This is perfectly fine for **offline learning** where resulting functions are **reusable**
- But not for **online learning** where resulting functions are **disposable**
- We can't afford to simulate 100K+ examples
- We need to get away with 1K to maximum 30K examples depending on required accuracy

# ANN: risk performance

delta wrt 1st asset : ANN vs formula on independent test set




source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialMLTF2.ipynb

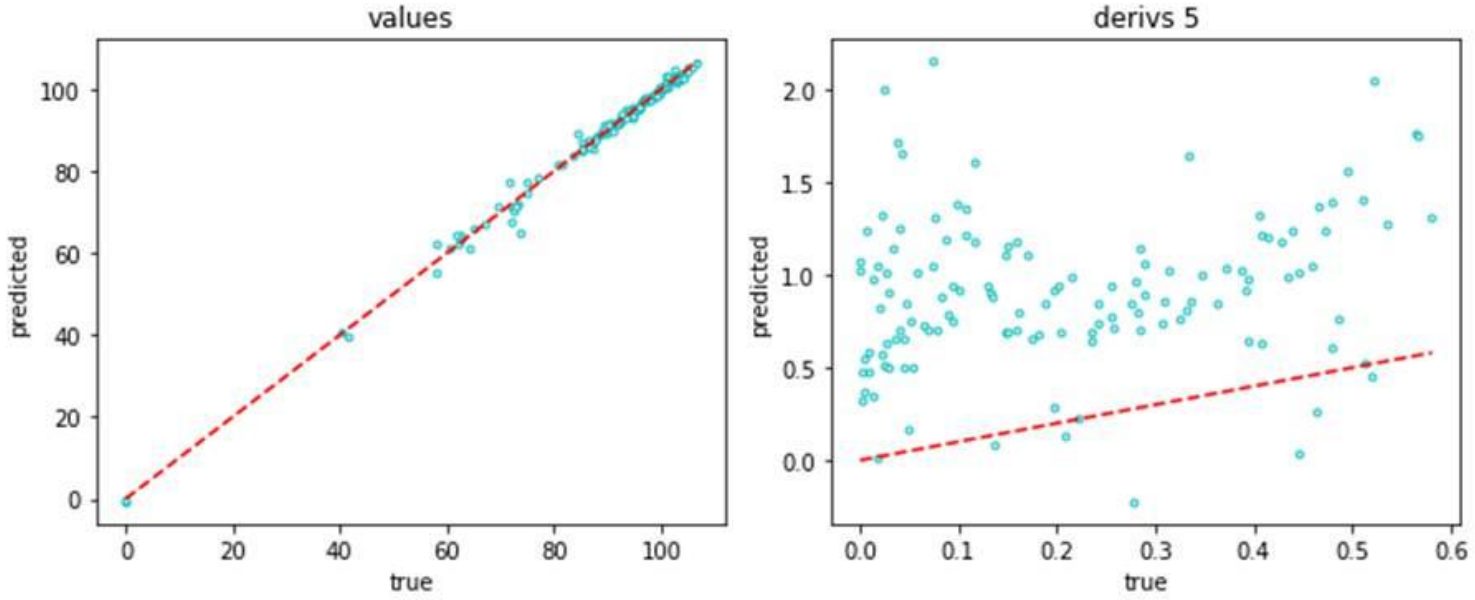
Open in Colab

# ANN: risk performance

- Risk sensitivities also converge
- Guaranteed by a variant of the universal approximation theorem (see Horvath and al. 2019)
- But they converge a lot slower than values and approximation is generally much worse
- Recall that risk sensitivities are at least as necessary as values
- (to some extent, valuation is a middle step towards differentiation:
  it is the risk sensitivities that give the all-important hedge ratios)
- Necessary for heatmaps
- Required in some regulations like SIMM-MVA

# Autocallable

- Another example: best-of-four autocallable in Danske Bank's Superfly
- Correlated local volatility models a la Dupire
- Low dimension 4 but complex model and instrument
- 32k training examples, decent approximation of values but deltas are completely off

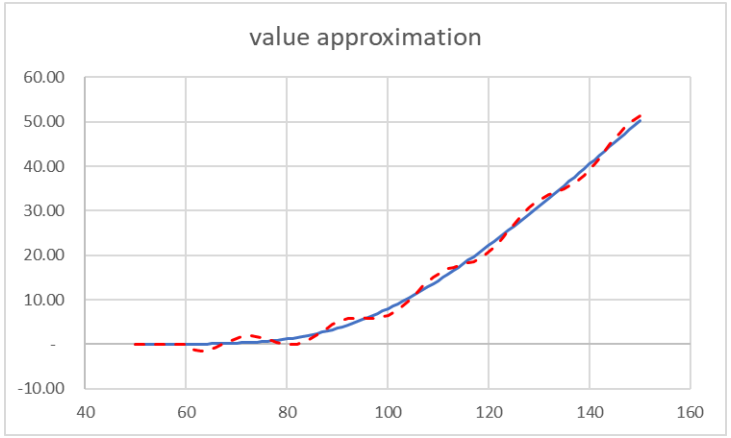


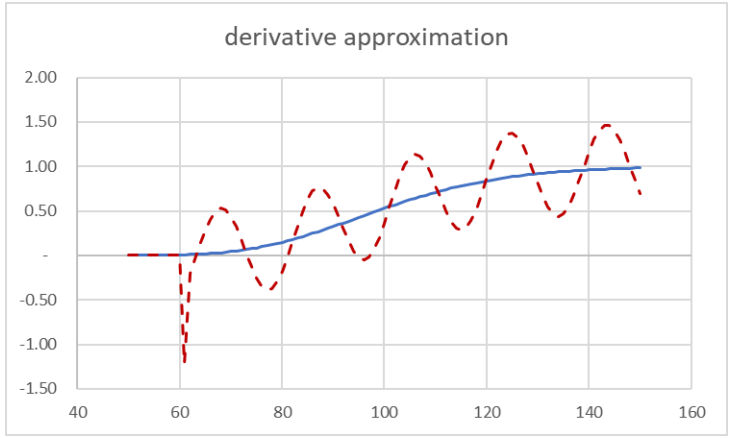

- Performance on independent test set
- Horizontal axis: correct values and deltas by nested MC
- Vertical axis: predictions of the trained ANN
- Error = distance to diagonal

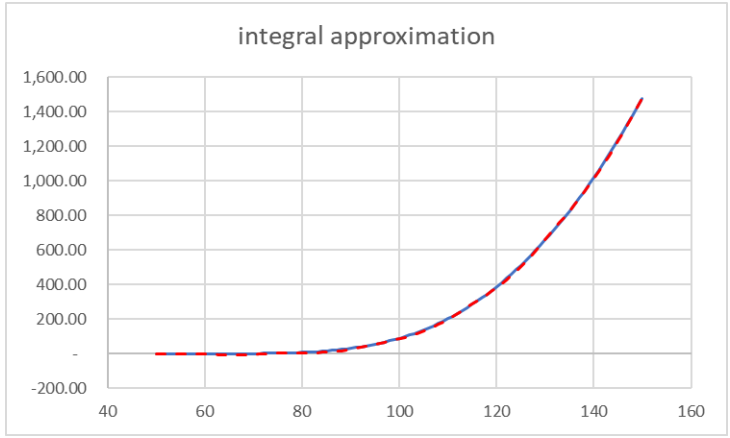
# Learning risk sensitivities

- Well-known in the field of function approximation:

  *the derivative of a good approximation is not a good approximation of the derivative*




- Conversely:
  the **integral** of a good (unbiased) approximation **is** a good approximation of the integral
- Should we not then learn from deltas and get good prices,
  rather than learning from prices and get poor deltas?
- Hold that though

# Conclusion

- We established the basis for pricing by ML and reviewed conventional ML algorithms
- Regression is only applicable in low dimension
- PCA is **never** applicable
- ANN perform very well on the pricing problem but consume vast amounts of examples
  - Well suited for offline learning
    where a pricing function is learned once and reused forever e.g. McGhee (2018), Horvath and al. (2019)
  - Not suitable for online learning
    where a pricing function is learned to be used only a few thousands of times
    then we must learn a lot faster
- A hint was given by the inspection of derivative sensitivities:
  we noticed that the derivatives of approximate functions are poor, whereas integrals are good
- Incentive to learn derivatives sensitivities, and obtain values by integration
  rather than learn values and obtain sensitivities by differentiation
  this is the cornerstone of **differential** machine learning

# Part III
# Differential Machine Learning

What is differential machine learning and how does it help?

# Differential Machine Learning

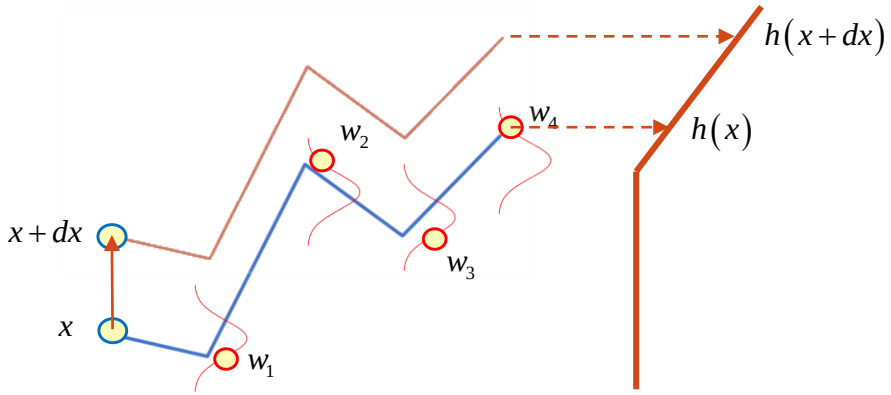
- Recall we are given a model $(x_t) = f(x_M, w)$ and a Derivative or book $y = g(x_I; (x_t))$
- Define the pathwise payoff: $y = h(x; w) = g(x_I; f(x_M; w))$
- Compute the dataset $\left(x^{(i)}, y^{(i)} = h\left(x^{(i)}; w^{(i)}\right)\right)$
  - Pick initial state $x^{(i)}$ and a random vector $w^{(i)}$
  - Compute payoff $y^{(i)} = h\left(x^{(i)}; w^{(i)}\right)$ and repeat m times
- Learn value function: $v(x) = E[y|x] \approx \alpha\left(x; \vartheta^*\right)$ where $\vartheta^* = \arg\min MSE(\vartheta) = \sum_{i=1}^{m}\left[y^{(i)} - \alpha\left(x^{(i)}; \vartheta\right)\right]^2$

- Differential dataset: **pathwise differentials** $z^{(i)} = \frac{\partial h\left(x^{(i)}; w^{(i)}\right)}{\partial x^{(i)}} \in \mathbb{R}^n$
- Learn **risk** function by min MSE over gradients: $\frac{\partial v(x)}{\partial x} \approx \frac{\partial \alpha\left(x; \vartheta^*\right)}{\partial x} \in \mathbb{R}^n$ where $\vartheta^* = \arg\min \sum_{i=1}^{m}\left\| z^{(i)} - \frac{\partial \alpha\left(x^{(i)}; \vartheta\right)}{\partial x^{(i)}}\right\|^2$

- And value function as a by-product $v(x) = v(x_0) + \int \frac{\partial v(x)}{\partial x} dx \approx v(x_0) + \int \frac{\partial \alpha\left(x; \vartheta^*\right)}{\partial x} dx$

# Pathwise differentials

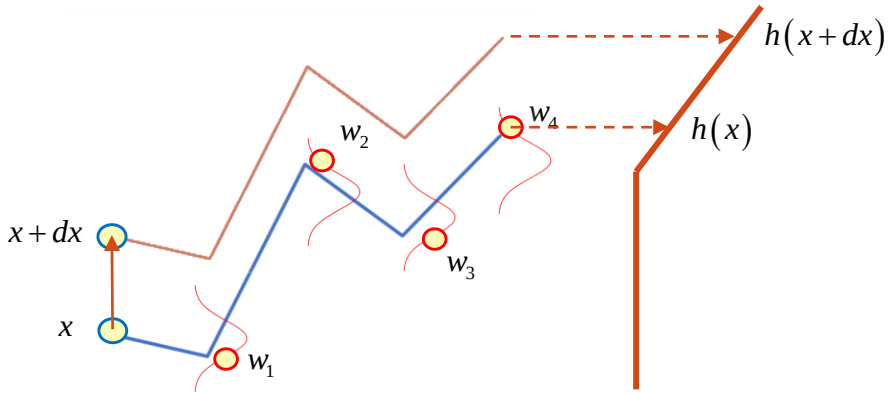
- By definition: $z^{(i)} = \dfrac{\partial h\left(x^{(i)}; w^{(i)}\right)}{\partial x^{(i)}} \in \mathbb{R}^n$
- Bump initial state, keep randomness unchanged




- Black & Scholes: $h(x;w) = (x_T - K)^+$ where $x_T = x\exp\left[-\dfrac{\sigma^2}{2}T + \sigma\sqrt{T}N^{-1}(w)\right]$ so $\dfrac{\partial h(x;w)}{\partial x} = 1_{\{x_T > K\}}\dfrac{dx_T}{dx} = 1_{\{x_T > K\}}\dfrac{x_T}{x}$
- Basket: $h(x;w) = (b_T - K)^+$ where $b_T = a^T\left[x + chol(\Sigma)N^{-1}(w)\right]$ so $\dfrac{\partial h(x;w)}{\partial x} = 1_{\{b_T > K\}}\dfrac{db_T}{dx} = 1_{\{b_T > K\}}a$

# Pathwise differentials

- By definition: $z^{(i)} = \dfrac{\partial h\left(x^{(i)}; w^{(i)}\right)}{\partial x^{(i)}} \in \mathbb{R}^n$
- Bump initial state, keep randomness unchanged




- Black & Scholes: $h(x;w) = (x_T - K)^+$ where $x_T = x \exp\left[-\dfrac{\sigma^2}{2}T + \sigma\sqrt{T}N^{-1}(w)\right]$ so $\dfrac{\partial h(x;w)}{\partial x} = 1_{\{x_T > K\}} \dfrac{dx_T}{dx} = 1_{\{x_T > K\}} \dfrac{x_T}{x}$
- Basket: $h(x;w) = (b_T - K)^+$ where $b_T = a^T\left[x + chol(\Sigma)N^{-1}(w)\right]$ so $\dfrac{\partial h(x;w)}{\partial x} = 1_{\{b_T > K\}} \dfrac{db_T}{dx} = 1_{\{b_T > K\}}\, a$
- **Note in both cases the payoff h and derivative dh are computed efficiently together as a lot of computations are common**

# How to compute pathwise derivatives?

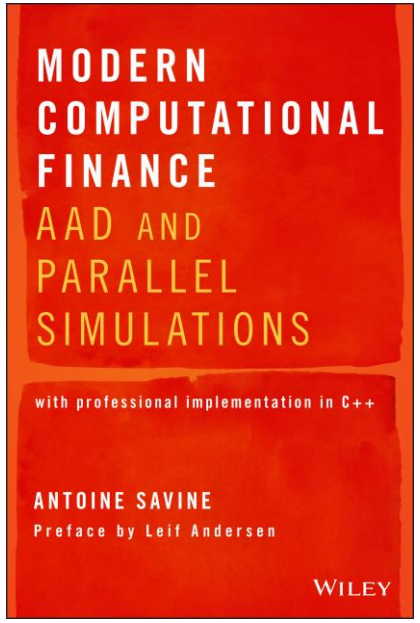
- In the general case, the derivatives of an arbitrary function $h:\mathbb{R}^n \to \mathbb{R}$ are computed with **AAD** (Automatic Adjoint Differentiation)
- AAD was called the "holy grail" of sensitivity computation (Griewank, 2012)
  - extremely efficient:
    computes differentials **in constant time** so 100s of sensitivities of some function h cost ~5 evaluations
  - very accurate, sensitivities are computed **analytically**
- In return, AAD has a steep learning curve, both in theory and in practice
  - relies on computation graphs
  - uses advanced programming techniques like operator overloading and expression templates
  - memory intensive so memory management is critical
- AAD was introduced to finance by Giles and Glasserman's Smoking Adjoints in 2006
- Because of its critical importance in finance (and ML: backprop is a form of AAD)
  - it is covered in a vast amount of literature: books, articles, tutorials, blogs…
  - and implemented in many libraries
    open-source (C++: ADEP, Python: TensorFlow, PyTorch), commercial (e.g. CompatiBL, NAG, Matlogica)

# AAD

- Critical prerequisite for differential ML
- **Not** covered in this presentation

- Our own AAD material:
  - Modern Computational Finance, volume 1
    All about AAD and application in Finance
    500+ pages with complete C++ code
  - A 15min introduction
    Recorded at Bloomberg in 2019

- Many other authors cover AAD too
- Not least the seminal paper by Giles & Glasserman




https://www.amazon.co.uk/Modern-Computational-Finance-Parallel-Simulations/dp/1119539455

https://www.youtube.com/watch?v=IcQkwgPwfm4

# Pathwise differentials: existence

- Recall: $h(x_M, x_I; w) = g(x_I; f(x_M; w))$

where f is the model : initial model state, random numbers → path

and g is the transaction : initial transaction state, path → payoff

- Hence: $\frac{\partial h(x_M, x_I; w)}{\partial x_M} = \frac{\partial f(x_M; w)}{\partial x_M} \frac{\partial g(x_I; f(x_M; w))}{\partial f(x_M; w)}$

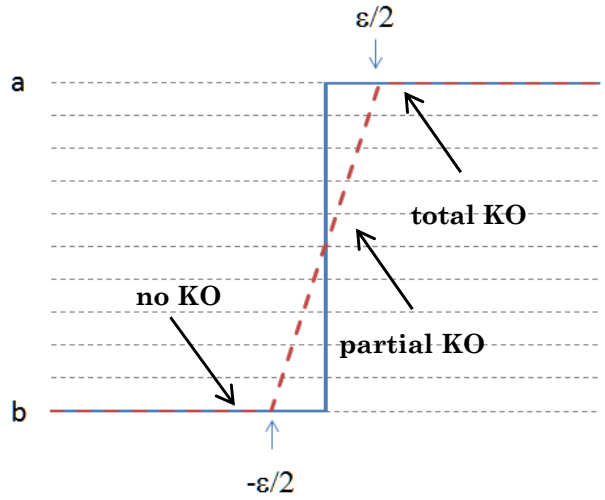
sensitivity of path wrt initial state
✓ (generally) well-defined

sensitivity of payoff wrt path
⊘ often discontinuous: e.g. digitals, barriers incl. autocallables, range floaters etc.

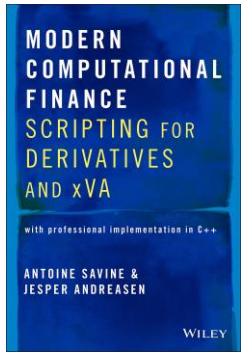
- So pathwise differentials **do not exist** for many families of discontinuous financial products
- **This is problem not only for Differential ML but also e.g. MC risks**
- So the problem is well-known, MC risks cannot be computed accurately
- The industry has long developed a standard solution: **smoothing**

# Smoothing

- Smoothing = approximate discontinuous product by a close one that is continuous
- e.g. digital → tight call spread, barrier → smooth (Parisian) barrier etc.
- More on smoothing (and fuzzy logic!):
  - QuantMinds 2016 presentation
    https://www.slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic


  - Modern Computational Finance, Volume 2, part IV



# Back to existence

- Here, we assume that all cashflows are smoothed
- It follows that pathwise derivatives $\frac{\partial h(x;w)}{\partial x}$ **always** exist
- Recall the pricing equation: $v(x) = \int_{(0,1)^d} h(x;w)\,dw$ price = integral of pathwise payoff dw
- Differentiating under the integral, we obtain the **risk equation**: $\frac{\partial v(x)}{\partial x} = \int_{(0,1)^d} \frac{\partial h(x;w)}{\partial x}\,dw$
  risks = integral of pathwise derivatives dw
- Recall prices are correctly learned by min MSE between price predictions alpha and payoffs h
- By **exactly the same argument** risks are correctly learned by min MSE between:
  - *d alpha / dx* the predicted risks
  - and *d h / dx* the pathwise derivatives

➢ (As long as smoothing is applied) differential ML converges to the correct risk sensitivities

# Combined differential learning

- Learn value function: $v(x) \approx \alpha\left(x;\vartheta^*\right)$ by minimization of MSE of:

  *conventional* or *differential*

- payoffs vs prices: $\vartheta^* = \arg\min \sum_{i=1}^{m}\left[y^{(i)} - \alpha\left(x^{(i)};\vartheta\right)\right]^2$ pathwise derivatives vs risks: $\vartheta^* = \arg\min \sum_{i=1}^{m}\left\| z^{(i)} - \frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}\right\|^2$

- In practice, best results are obtained by combining both:

$$\vartheta^* = \arg\min \left\{ \underbrace{\sum_{i=1}^{m}\left[y^{(i)} - \alpha\left(x^{(i)};\vartheta\right)\right]^2}_{\text{price/payoff MSE}} + \sum_{j=1}^{n}\lambda_j \underbrace{\sum_{i=1}^{m}\left( z^{(i)}[j] - \frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}[j]\right)^2}_{\text{risk/pathwise derivative MSE, j-th state variable}} \right\}$$

notation: $\overline{MSE}_j(\vartheta) = \sum_{i=1}^{m}\left( z^{(i)}[j] - \frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}[j]\right)^2$

- lambda_j = weight of j-th derivative in the combined cost function
- A good default: $\lambda_j = \frac{E\left[y^2\right]}{E\left[z_j{}^2\right]}$ (see discussion on GitHub differential-machine-learning)

# Benefits

1. Differential ML learns solid Greeks, by construction, and solid values, by integration
2. Increases dataset: for every x(i), we have 1 payoff y(i) **and n additional derivatives** z(i)
3. Bias-free regularization

| Standard regularizion | Differential ML |
|---|---|
| $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \lambda \|\vartheta\|^2 \right]$ | $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \sum_{j=1}^{n} \lambda_j \overline{MSE_j}(\vartheta) \right]$ |
| • Penalizes large weights<br>• Reduces offerfitting and variance<br>• Introduces bias<br>• And strong dependency on lambda, i.e. bias-variance tradeoff<br>• Hence, requires cross-validation to find sweet spot<br>• With generally disappointing performance | • Penalizes **wrong derivatives**<br>• Reduces offerfitting and variance<br>• Does **not** introduce any bias<br>*since correct derivatives give correct values by integration*<br>• No BV tradeoff, no need for CV, (very) small dependency on lambda<br>• Performance: see next, *and see for yourself on GitHub* |

4. We learn the **shape** of the function, not only the value on some points
   - Differential ML with m examples performs *better* than ML with (n+1).m examples → try it on GitHub!
   - Improves the speed and stability of iterative training algorithms

# Part IV
# Differential Regression

Our first differential ML algorithm

# Differential regression

| regression | objective | |
|---|---|---|
| standard | $\vartheta^* = \arg\min MSE(\vartheta)$ | |
| regularized | $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \lambda \|\vartheta\|^2 \right]$ | |
| differential | $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \sum_{j=1}^{n} \lambda_j \overline{MSE_j}(\vartheta) \right]$ | |

# Differential regression

| regression | objective | solution |
|---|---|---|
| standard | $\vartheta^* = \arg\min MSE(\vartheta)$ | $\vartheta^* = C_{\varphi\varphi}{}^{-1} C_{\varphi y}$ |
| regularized | $\vartheta^* = \arg\min \left[ MSE(\vartheta) + \lambda \|\vartheta\|^2 \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \lambda I_K \right)^{-1} C_{\varphi y}$ |
| differential | $\vartheta^* = \arg\min \left[ MSE(\vartheta) + \sum_{j=1}^{n} \lambda_j \overline{MSE_j}(\vartheta) \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \sum_{j=1}^{n} \lambda_j C_{jj}^{\varphi} \right)^{-1} \left( C_{\varphi y} + \sum_{j=1}^{n} \lambda_j C_j^{\varphi z} \right)$ |

# Differential regression

| regression | objective | solution |
|---|---|---|
| standard | $\vartheta^* = \arg\min MSE(\vartheta)$ | $\vartheta^* = C_{\varphi\varphi}{}^{-1} C_{\varphi y}$ |
| regularized | $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \lambda \lVert\vartheta\rVert^2 \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \lambda I_K \right)^{-1} C_{\varphi y}$ |
| differential | $\vartheta^* = \arg\min\left[ MSE(\vartheta) + \sum_{j=1}^{n} \lambda_j \overline{MSE_j}(\vartheta) \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \sum_{j=1}^{n} \lambda_j C_{jj}^{\varphi} \right)^{-1} \left( C_{\varphi y} + \sum_{j=1}^{n} \lambda_j C_j^{\varphi z} \right)$ |

- Where $C_{jj}^{\varphi} = E\left[ \varphi_j(x) \varphi_j(x)^T \right] \in \mathbb{R}^{K \times K}$ is the covariance matrix of the derivatives of basis functions wrt x_j
- $\varphi_j(x) = \left[ \frac{\partial \varphi_1(x)}{\partial x_j}, \ldots, \frac{\partial \varphi_K(x)}{\partial x_j} \right] \in \mathbb{R}^K$ example: cubic regr. dim 2

$$
\begin{array}{lccccccccccc}
\varphi(x_1,x_2) = & [ & 1 & x_1 & x_2 & x_1^2 & x_2^2 & x_1x_2 & x_1^2x_2 & x_1x_2^2 & x_1^3 & x_2^3 & ] \\
\varphi_1(x_1,x_2) = & [ & 0 & 1 & 0 & 2x_1 & 0 & x_2 & 2x_1x_2 & x_2^2 & 3x_1^2 & 0 & ] \\
\varphi_2(x_1,x_2) = & [ & 0 & 0 & 1 & 0 & 2x_2 & x_1 & x_1^2 & 2x_1x_2 & 0 & 3x^2 & ]
\end{array}
$$

- and $C_j^{\varphi z} = E\left[ \varphi_j(x) z[j] \right] \in \mathbb{R}^K$ is the vector of covariances of phi_j with z[j], the j-th pathwise derivative

# Differential regression

| regression | objective | solution |
|---|---|---|
| standard | $\vartheta^* = \arg\min MSE(\vartheta)$ | $\vartheta^* = C_{\varphi\varphi}{}^{-1} C_{\varphi y}$ |
| regularized | $\vartheta^* = \arg\min \left[ MSE(\vartheta) + \lambda \lVert\vartheta\rVert^2 \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \lambda I_K \right)^{-1} C_{\varphi y}$ |
| differential | $\vartheta^* = \arg\min \left[ MSE(\vartheta) + \sum_{j=1}^{n} \lambda_j \overline{MSE_j}(\vartheta) \right]$ | $\vartheta^* = \left( C_{\varphi\varphi} + \sum_{j=1}^{n} \lambda_j C_{jj}^{\varphi} \right)^{-1} \left( C_{\varphi y} + \sum_{j=1}^{n} \lambda_j C_j^{\varphi z} \right)$ |

- Proof: convex problem, set gradient of cost wrt theta to zero to find unique minimum

# Diff regression: implementation

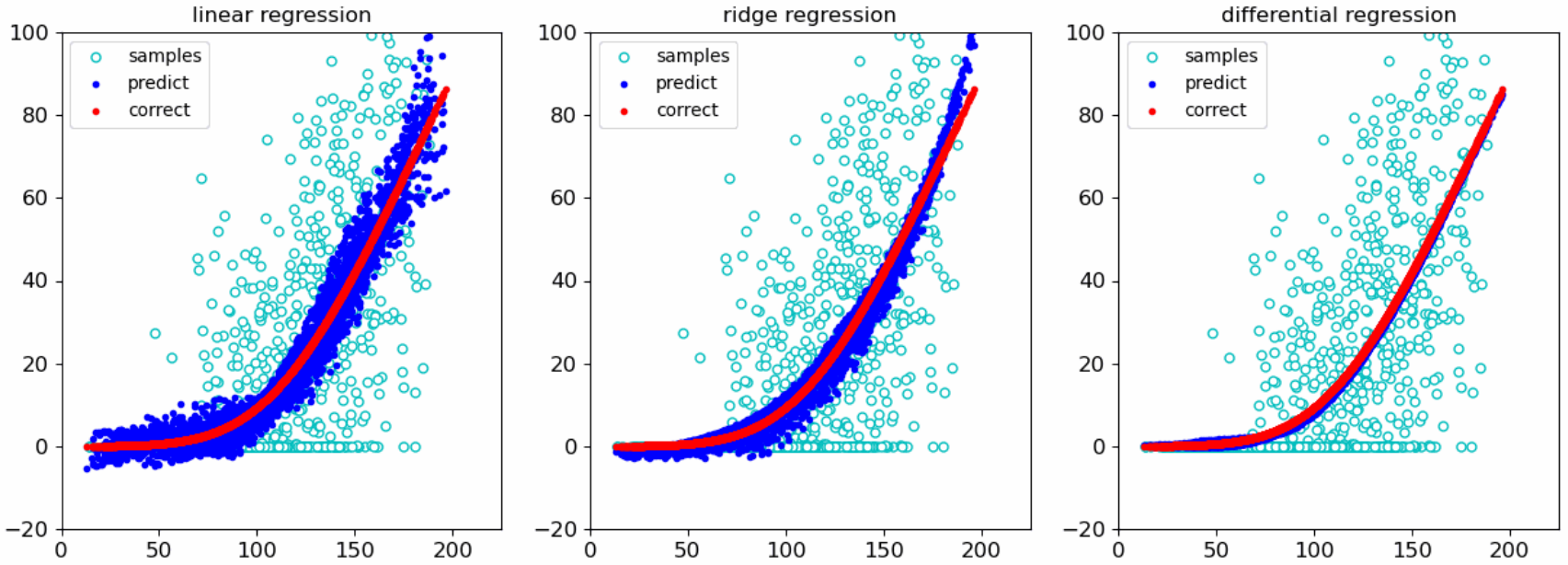
- Differential regression is not hard, but it is heavy in terms of notations
- Better understood with implementation code
- Available on GitHub (differential-machine-learning)

  https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

- With applications and a discussion of implementation details
- Run it on your browser (or iPhone): 

# Diff regression: performance

correlated Gaussian basket **dim 3** : deg 5 regression, only 1024 training examples, ridge optimized by CV

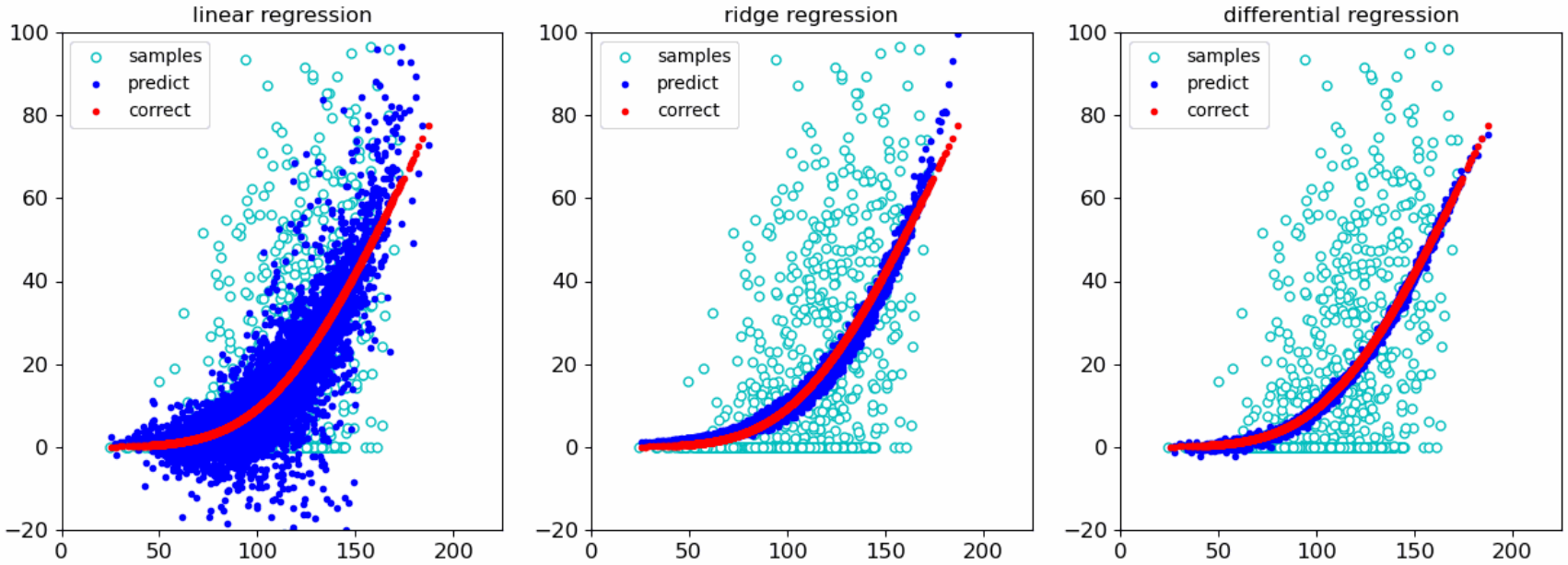



source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# Diff regression: performance

correlated Gaussian basket **dim 5** : deg 5 regression, only 1024 training examples, ridge optimized by CV

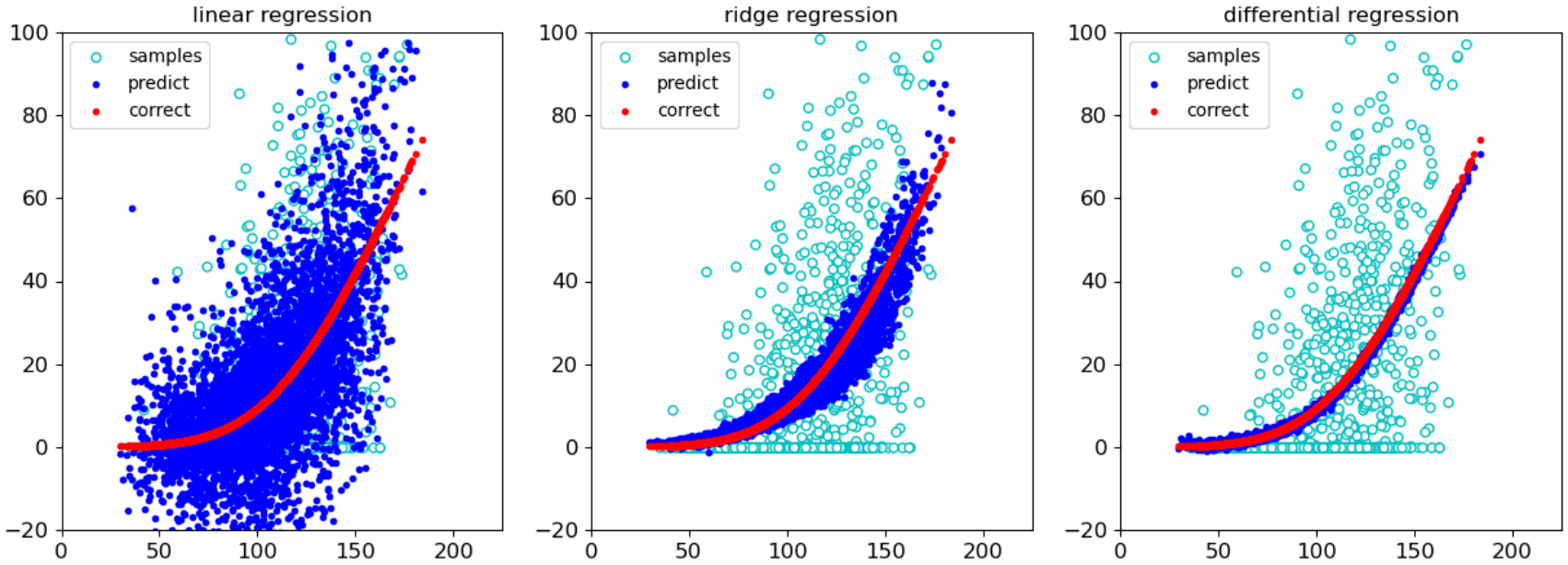



source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# Diff regression: performance

correlated Gaussian basket **dim 7** : deg 5 regression, only 1024 training examples, ridge optimized by CV




source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# Diff regression: performance

- Sharp improvement over ridge (optimized with CV)
- Without any CV!

- Still, deteriorates in higher dimension
- So differential regression in dim > 1 remains **worse than standard regression in dim 1**
- Recall the basket option is really a dim-1 problem
- Means we can do even better
- It should be possible to safely reduce dimension to 1, automatically, from data
- And perform effective regression in dimension 1,
  irrespective of original dimension, which may be, say, 500 or more

➢ Enter differential PCA

# Part V
# Differential PCA

A safe variant of PCA for an effective dimension reduction

# Data normalization

- As customary in ML, we don't really work with data x, y, z but with **normalized** data:

$$x = \frac{x - Ex}{\sigma_x}, y = \frac{y - Ey}{\sigma_y}, \hat{z} = \frac{\partial y}{\partial x} = \frac{\sigma_x}{\sigma_y} z$$

(where constant or redundant columns of x are filtered out)

- This is critical for working with risk sensitivities:

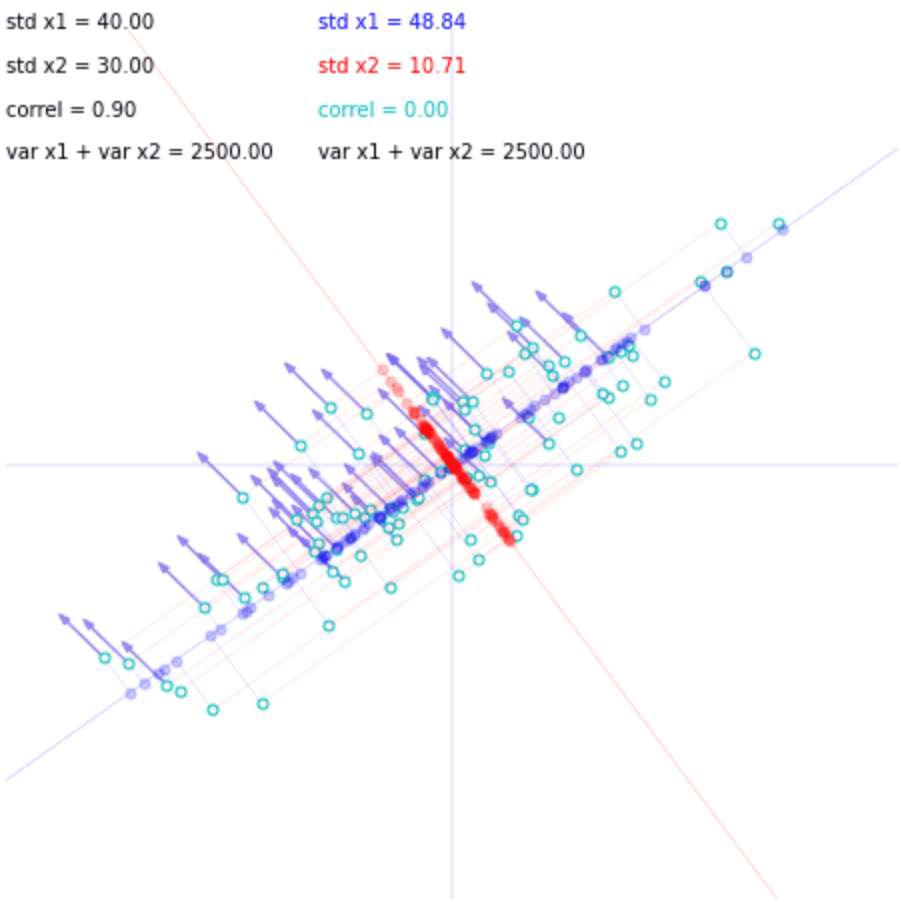
1. All risks are expressed in **consistent units** of std of y per std of x
2. Risks are not naked sensitivities but proportional to std_x times sensi

- So we can meaningfully talk about the **magnitude** of risks
  and consistently compare and aggregate different risks such as wrt rates or wrt equities

# Differential PCA

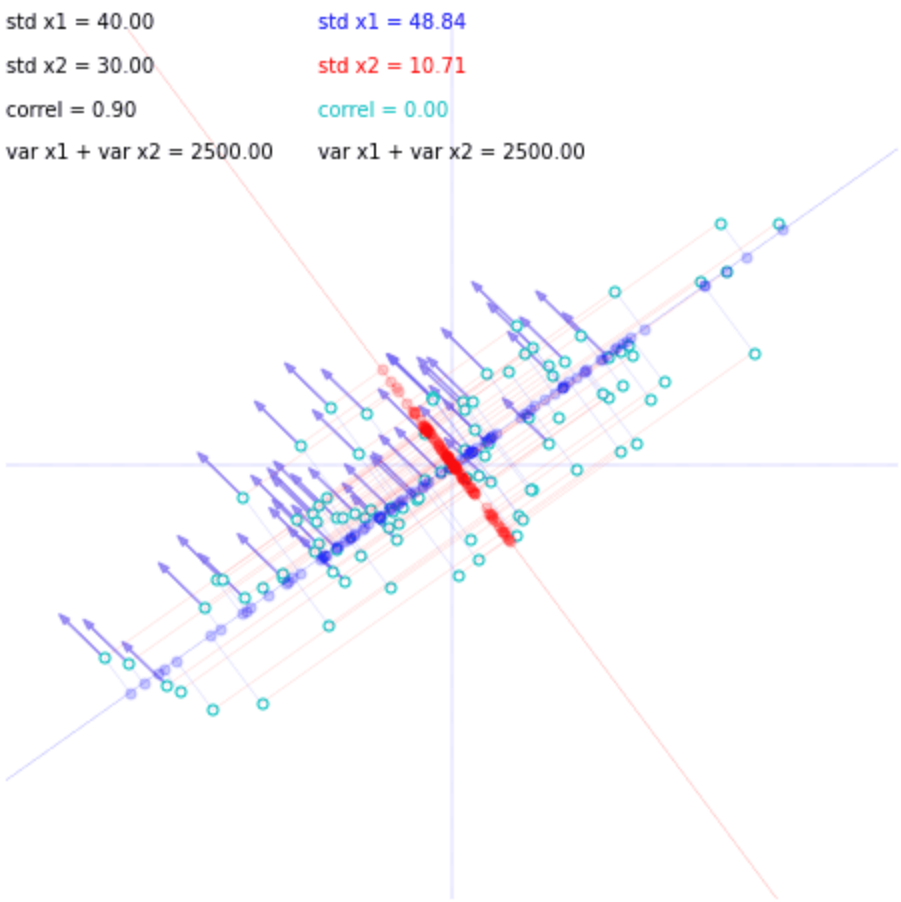
Introduced in the oct21 Risk paper

Axes that matter: PCA with a difference



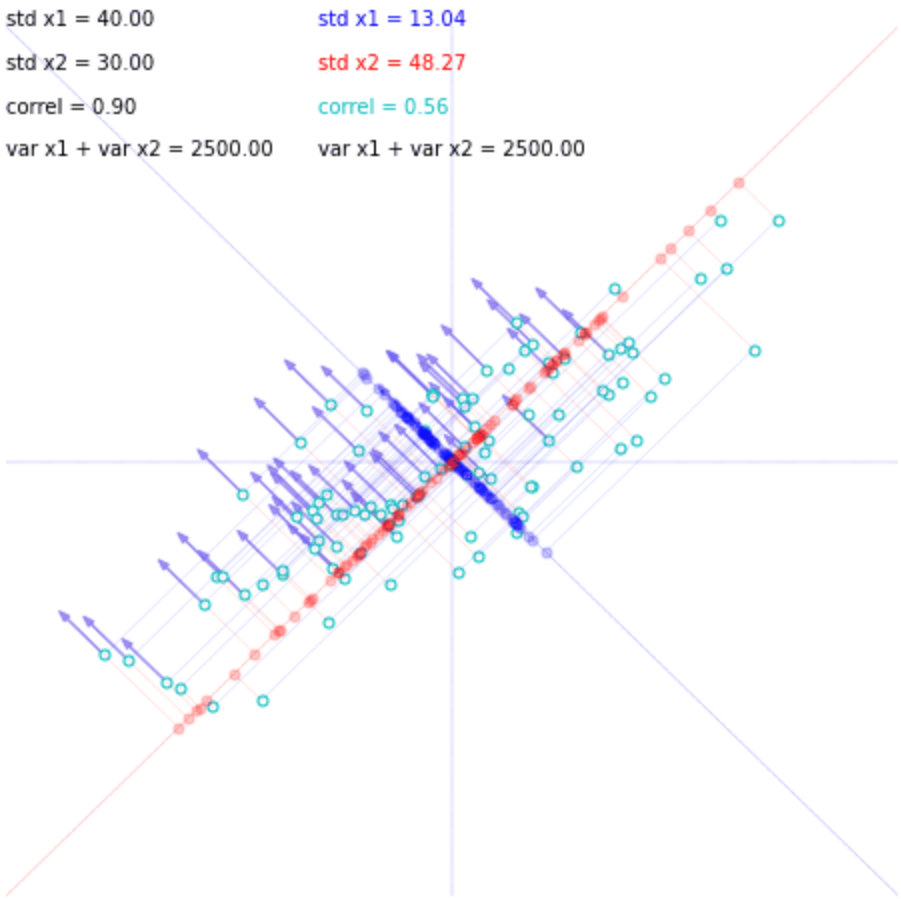

PCA projects the data on axes of maximum **variation** (in blue) and scraps coordinates of small variation (in red)

All the action (magnitude of **gradients**) occurs on the red axis, gradients are virtually 0 on the blue axis

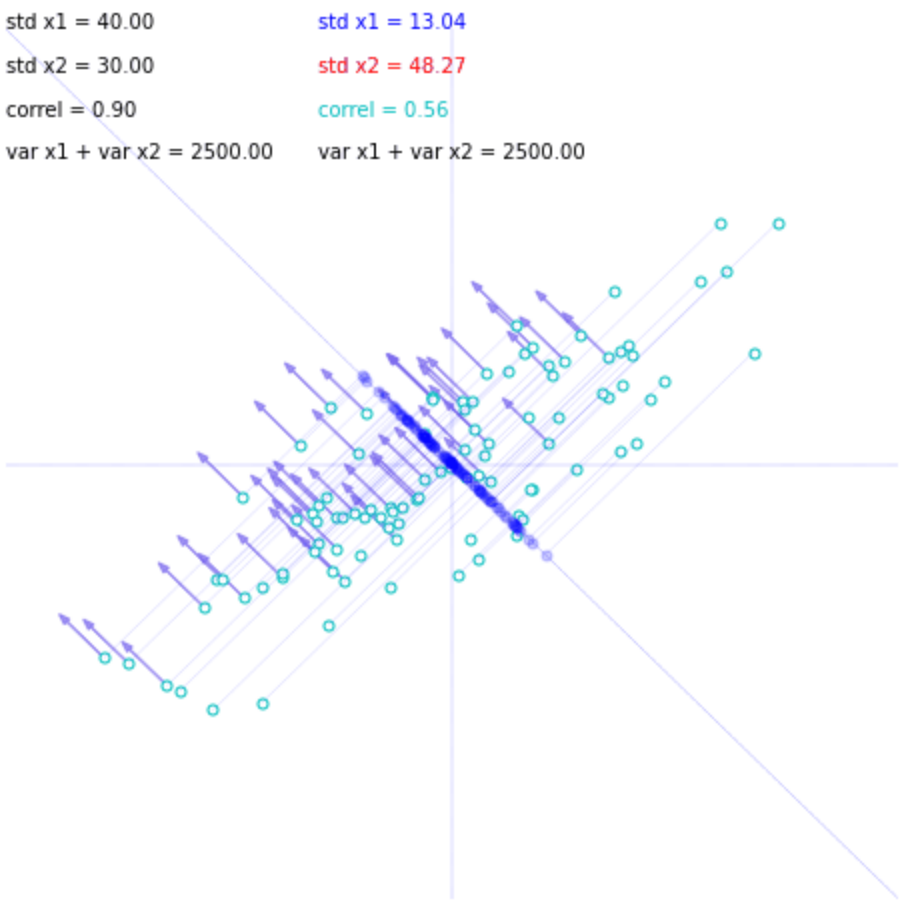



[Variation = magnitude of data coordinates] is the wrong metric

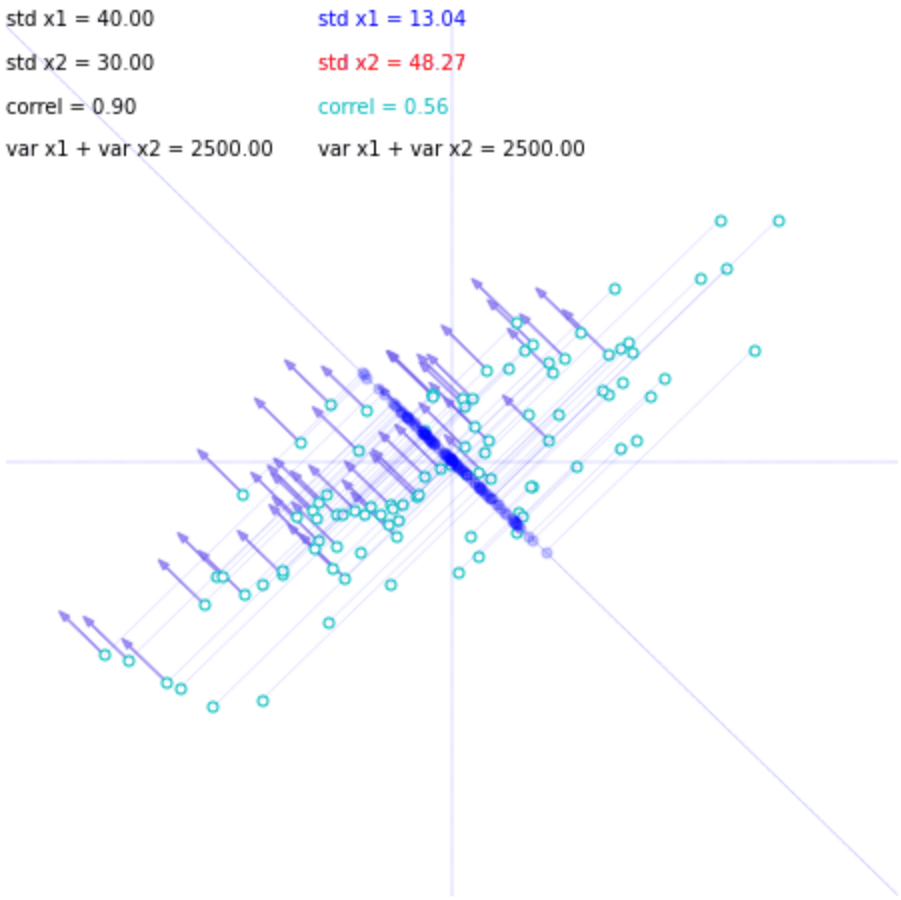
The correct metric is the **magnitude of directional derivatives**, called **relevance**




This picture suggests to align the coordinate system **with gradients**




And truncate the coordinates where the gradients magnitude is negligible




Note that this is very different from what PCA does
In this example, it is the opposite!

| | classic PCA | differential PCA |
|---|---|---|
| ranks axes by | variation: $\mathrm{var}(u) = E\left[(u \cdot x)^2\right]$ | relevance: $rel(u) = E\left[(u \cdot z)^2\right]$ |
| coordinate basis | eigenvectors of: $C_{xx} = E\left[xx^T\right] = PDP^T$<br>eigenvalues measure variation | eigenvectors of: $C_{zz} = E\left[zz^T\right] = PDP^T$<br>eigenvalues measure relevance |
| key property | coordinates of state $x$ are orthogonal | coordinates of gradient $z$ are orthogonal |
| truncation | eigenvalues sorted decreasing order, compute dimension reduction $p$ such that $\sum_{i=p+1}^{n} D_{ii} \le \varepsilon$<br>then, truncate $q=n-p$ rightmost columns of $P$ into $P \in \mathbb{R}^{n \times p}$ | |
| state encoding | encoding: $L = P^T z \in \mathbb{R}^p$ | reconstruction: $X = PL = PP^T x = \pi x \in \mathbb{R}^n$ error: $x - \tilde{x} = (I - \pi)x = \Sigma x$ |
| gradient encoding | encoding: $S = \frac{\partial y}{\partial L} = P^T z \in \mathbb{R}^p$ | reconstruction: $Z = PS = \pi z$ error: $z - \tilde{z} = \Sigma z$ |

# Differential PCA: safety guarantee

- Theorem, proved in the paper

*the magnitude of truncated risks is bounded by epsilon*

- Denote risk sensitivities $\Delta = \frac{\partial v(x)}{\partial x} = E\left[z|x\right] \in \mathbb{R}^n$
- Then: $E\left[\|\Sigma\Delta\|^2\right] \le E\left[\|\Sigma Z\|^2\right] \le \varepsilon$
- Recall that the magnitudes include sensitivity and volatility, by normalization
- Hence, differential PCA is **guaranteed** safe:
  it cannot truncate information relevant for the instrument in the model
  (subject to user specified threshold epsilon = 'acceptable' risk)

# Differential PCA: effectiveness

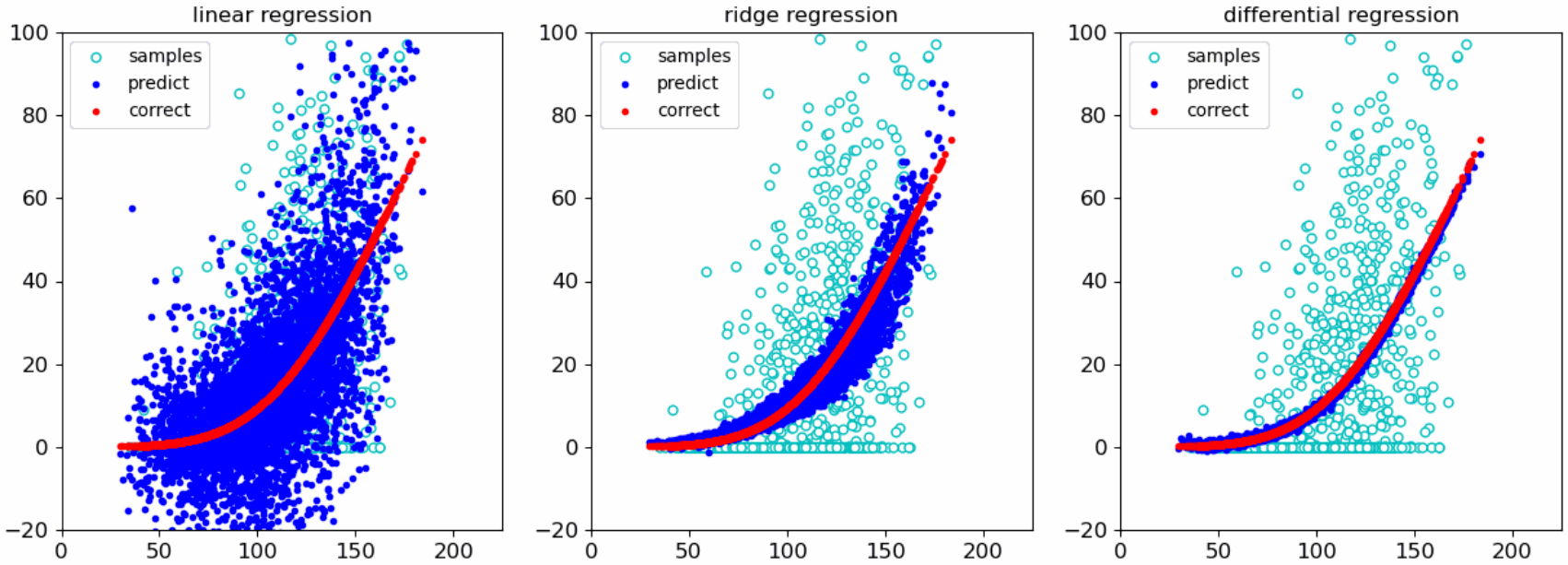
- Suppose that h is a nonlinear function of one linear feature of x: $h(x;w)=\beta\left(a^T x;w\right)$
- Like a basket option under the Gaussian assumption: $h(x;w)=\left(a^T x+a^T chol(\Sigma)N^{-1}(w)-K\right)^+$
- Then: $z=\frac{\partial h(x;w)}{\partial x}=\frac{\partial\beta\left(a^T x;w\right)}{\partial\left(a^T x\right)}a=\underbrace{\gamma(x;w)\|a\|}_{\text{random scalar}}\underbrace{\frac{a}{\|a\|}}_{\substack{\text{deterministic}\\ \text{unit vector}}}=\gamma(x;w)a$
- Hence: $C_{zz}=E\left[zz^T\right]=\|\gamma\|_2^2 aa^T$ only one nonzero eigenvalue $\|\gamma\|_2^2$ with eigenvector $a$

➢ Differential PCA reduces the dimension to 1 and finds the feature (e.g. basket) a from data!

➢ We finally resolved the basket problem in a fully satisfactory manner

# Recall

correlated Gaussian basket **dim 7** : deg 5 regression, only 1024 training examples, ridge optimized by CV

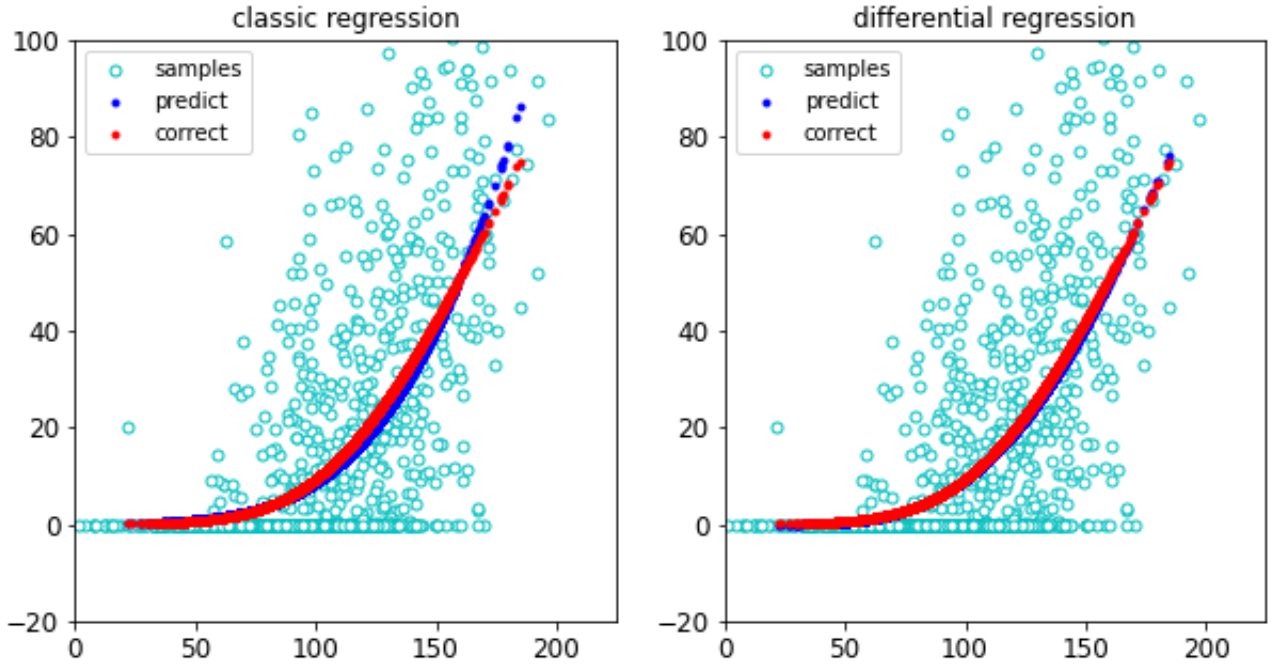



source: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialRegression.ipynb

Open in Colab

# With differential PCA

correlated Gaussian basket **dim 500** : deg 5 regression after differential PCA, **only 1024** training examples



# Differential PCA: effectiveness

- Every function h of x is also a function of n linearly independent features of x (change of basis) very often, h is (approximately) a (nonlinear) function of only p < n linear features of x:

$$h(x;w) = \beta\left(a_1{}^T x, a_2{}^T x, \ldots, a_p{}^T x, \ldots, a_n{}^T x; w\right) \approx \beta\left(a_1{}^T x, a_2{}^T x, \ldots, a_p{}^T x; w\right)$$

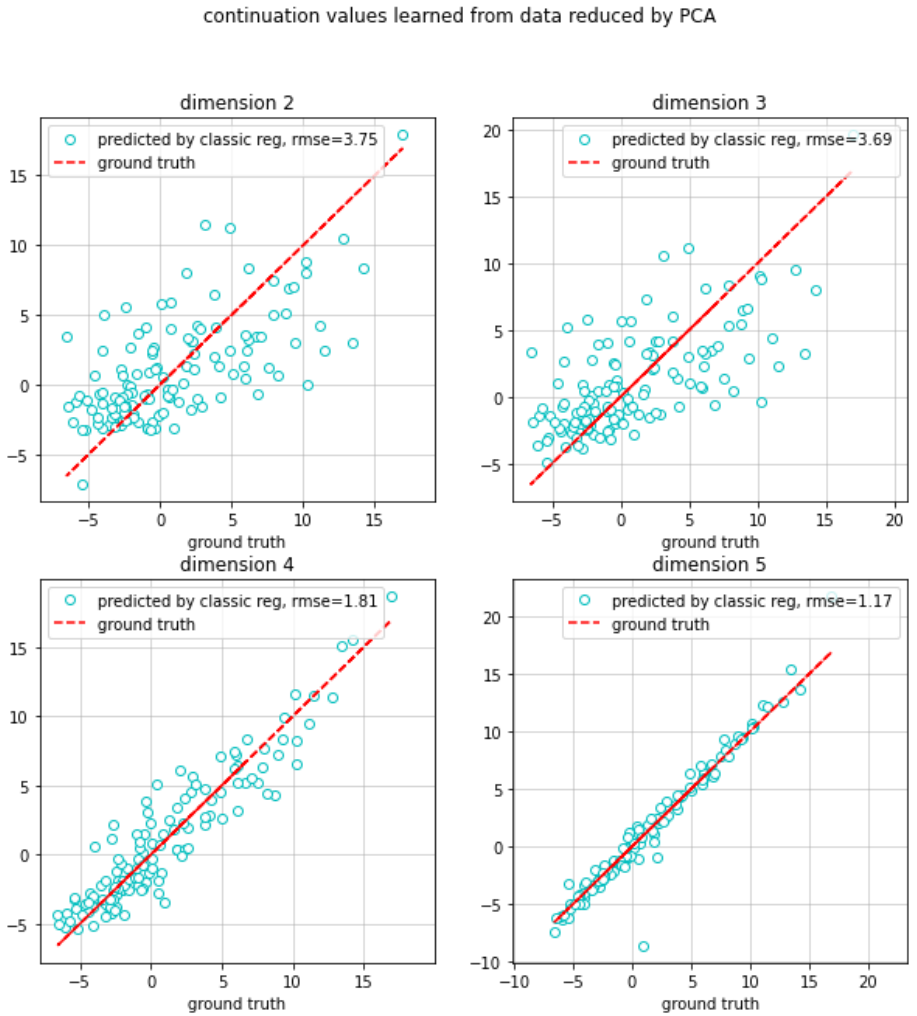
- Then differential PCA finds p material eigenvalues
- Which eigenvectors span the same space as a1,…,ap

➢ Hence, differential PCA always finds the axes that matter, i.e. relevant factors

- Recall PCA does the same thing for an option on a basket $\left(\sum_{i=1}^{n} a_i S_i - K\right)^+$ or a basket of options $\sum_{i=1}^{n} a_i (S_i - K)^+$
- Differential PCA always reduce dimension to 1 for an option on a basket
- And does not reduce dimension at all of a basket of options on independent underlying assets

➢ Differential PCA reduces dimension safely where possible, else keeps dimension unchanged

# Recall

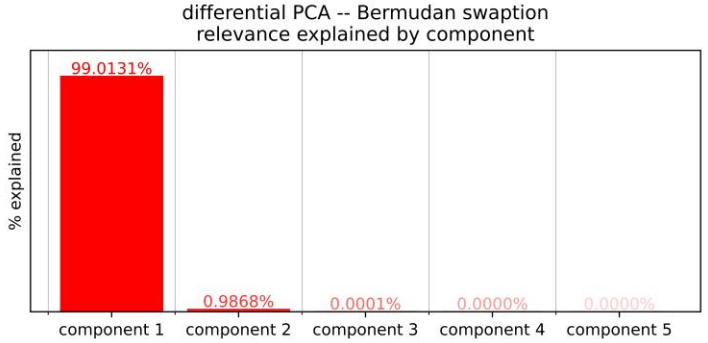


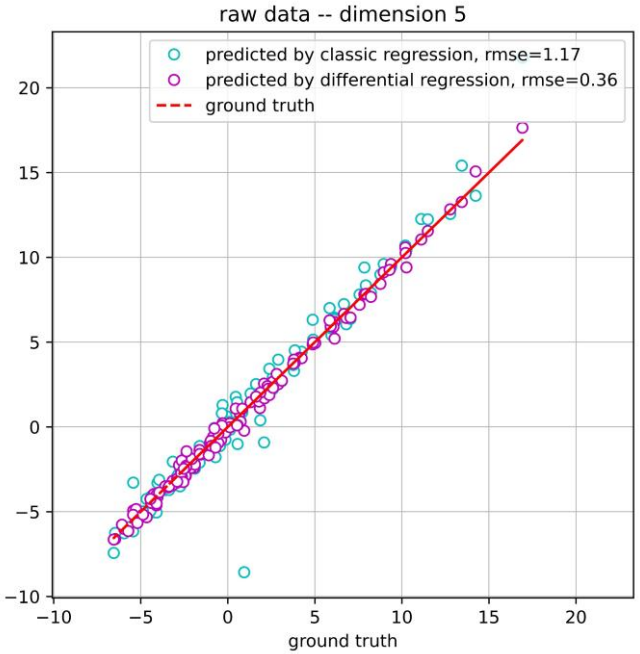

- **Continuation value of Bermudan option**
- 5F Gaussian model
- PCA could not reduce dimension
- Any attempt resulted in truncation of meaningful data
- So regression could no longer learn correct value

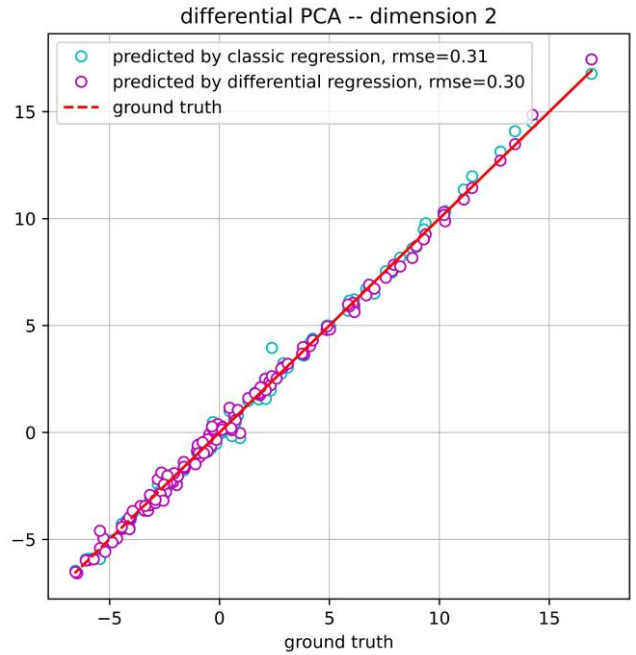
source:
https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb

Open in Colab

# With differential PCA




sources:

Risk.net: https://www.risk.net/cutting-edge/banking/7877231/axes-that-matter-pca-with-a-difference

GitHub: https://github.com/differential-machine-learning/notebooks/blob/master/Bermudan5F.ipynb

Open in Colab

# Differential PCA: efficiency

- Development cost: immediate drop-in replacement for PCA, see code on notebook

  https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialPCA.ipynb 
  

- Runtime cost:
  - Compute covariance matrix Czz: 0.25s on MKL for 32,768 examples in dimension 1,024 O(mn^2)
  - Eigenvalue decomposition of Czz = P D P^T: 0.05s on MKL for symmetric 1,024 matrix O(n^3)

- Both negligible by risk standards

# Differential PCA: conclusion

- PCA on covariance of gradients Czz
- Simple deployment with negligible runtime cost
- Guaranteed safety: never truncates meaningful information modulo user-specified acceptable risk threshold
- Designed to extract the linear features that matter
- Applications:
  - Safely reduce dimension for regression
  - Also useful for neural networks, training is faster and more stable in lower dimension
- Further applications:
  - Automatically identify regression variables in LSM (see Risk paper)
  - Reliably exhibit the risk factors of a given Derivative instrument or trading book
  - ➢ Help correctly specify and calibrate risk management models

# Part VI
# Differential Deep Learning

Effective differential machine learning with twin networks

# Differential Deep Learning

Introduced in the oct20 Risk paper

Differential Machine Learning:
the shape of things to come

Also on arXiv

# Differential ANN

- Recall the general differential ML training equation:

$$\vartheta^* = \arg\min \left\{ \underbrace{\sum_{i=1}^{m} \left[ y^{(i)} - \alpha\left(x^{(i)}; \vartheta\right) \right]^2}_{\text{price/payoff MSE}} + \sum_{j=1}^{n} \lambda_j \underbrace{\sum_{i=1}^{m} \left( z^{(i)}[j] - \frac{\partial \alpha\left(x^{(i)}; \vartheta\right)}{\partial x^{(i)}}[j] \right)^2}_{\text{risk/pathwise derivative MSE, j-th state variable}} \right\}$$

- Now, the (universal) approximator alpha(.;theta) is a neural network
- e.g. a feedforward network, defined by the equations:

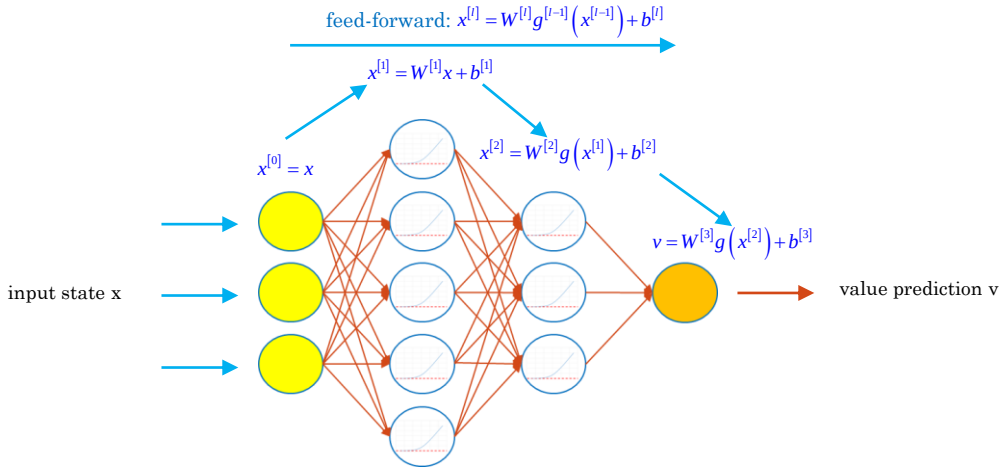
$$x^{[0]} = x \in \mathbb{R}^n \qquad \text{input layer}$$
$$x^{[l]} = W^{[l]} g^{[l-1]}\left(x^{[l-1]}\right) + b^{[l]} \in \mathbb{R}^{n_l} \qquad \text{hidden layers}$$
$$v = x^{[L]} \in \mathbb{R} \qquad \text{output layer}$$

- With the differential dataset (x,y,z) we have all the pieces we need to perform training

- **Except the risk predictions** $\frac{\partial \alpha\left(x^{(i)}; \vartheta\right)}{\partial x^{(i)}}$ (i.e. the derivatives of the value predictions wrt inputs)

# Differential ANN

- Recall the risk predictions $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ are the gradient of the output layer (v) wrt input layer (x)




- How can we compute this gradient efficiently?

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations

$$\bar{v} = \overline{x^{[L]}} = \frac{\partial MSE}{\partial v}$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

$$\frac{\partial MSE}{\partial W^{[l]}} = \overline{x^{[l]}} g^{[l-1]}\left(x^{[l-1]}\right)^{\mathrm{T}} \text{ and } \frac{\partial MSE}{\partial b^{[l]}} = \overline{x^{[l]}}$$

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations
- Here compute the derivatives of the value prediction v, not the MSE

$$\bar{v} = \overline{x^{[L]}} = \frac{\partial MSE}{\partial v}$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

$$\frac{\partial MSE}{\partial W^{[l]}} = \overline{x^{[l]}} g^{[l-1]}\left(x^{[l-1]}\right)^{\mathrm{T}} \text{ and } \frac{\partial MSE}{\partial b^{[l]}} = \overline{x^{[l]}}$$

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations
- Here compute the derivatives of the value prediction v, not the MSE

$$\bar{v} = \overline{x^{[L]}} = 1$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

$$\frac{\partial MSE}{\partial W^{[l]}} = \overline{x^{[l]}} g^{[l-1]}\left(x^{[l-1]}\right)^{\mathrm{T}} \text{ and } \frac{\partial MSE}{\partial b^{[l]}} = \overline{x^{[l]}}$$

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations
- Here compute the derivatives of the value prediction v, not the MSE
- wrt x not theta

$$\bar{v} = \overline{x^{[L]}} = 1$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

$$\frac{\partial MSE}{\partial W^{[l]}} = \overline{x^{[l]}} g^{[l-1]\prime}\left(x^{[l-1]}\right)^{T} \text{ and } \frac{\partial MSE}{\partial b^{[l]}} = \overline{x^{[l]}}$$

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations
- Here compute the derivatives of the value prediction v, not the MSE
- wrt x not theta

$$\bar{v} = \overline{x^{[L]}} = 1$$

$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

$$\bar{x} = \overline{x^{[0]}}$$

# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**

- Recall standard backprop equations
- Here compute the derivatives of the value prediction v, not the MSE
- wrt x not theta

$$\bar{v} = \overline{x^{[L]}} = 1$$

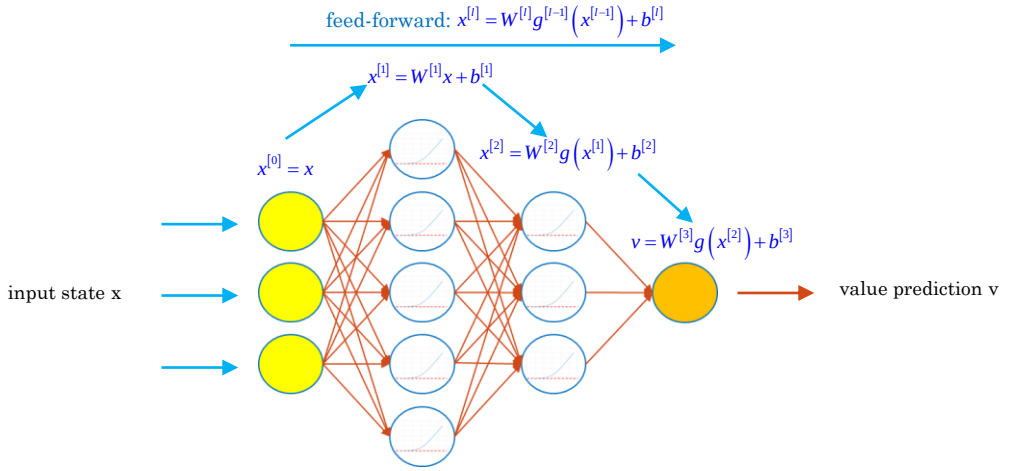
$$\overline{x^{[l-1]}} = g^{[l-1]\prime}\left(x^{[l-1]}\right) \otimes W^{[l]T}\,\overline{x^{[l]}}$$

with the notation $\overline{thing} = \frac{\partial v}{\partial thing}$

$$\bar{x} = \overline{x^{[0]}}$$

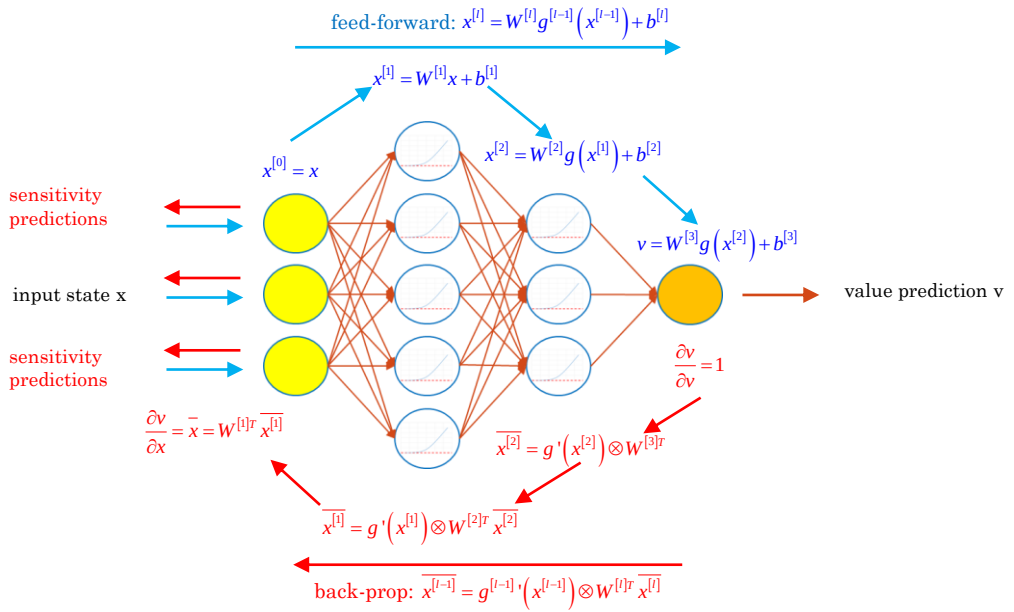
# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**



# Differential ANN

- How can we compute the gradient $\frac{\partial \alpha\left(x^{(i)};\vartheta\right)}{\partial x^{(i)}}$ efficiently? **With backpropagation equations!**



# Twin ANN

- In the 1990s, ML legend Yann Le Cun noticed that

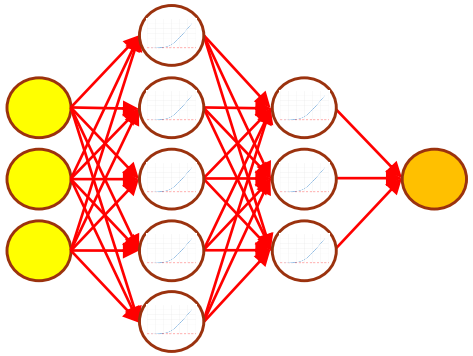
  *backpropagation equations define a dual network*

- Le Cun did not act on it
- But in finance, it makes a massive difference
- We can concatenate the original and dual networks
- Into a **twin network**
- Capable of predicting prices **and risks** for just twice the cost of price alone!
- So we can train the twin network efficiently $$\vartheta^* = \arg\min \left\{ \sum_{i=1}^{m} \left[ y^{(i)} - \alpha\left(x^{(i)}; \vartheta\right) \right]^2 + \sum_{j=1}^{n} \lambda_j \sum_{i=1}^{m} \left( z^{(i)}[j] - \frac{\partial \alpha\left(x^{(i)}; \vartheta\right)}{\partial x^{(i)}}[j] \right)^2 \right\}$$

- By (another layer of) (standard) backpropagation over the whole twin net
- And use the trained network to efficiently compute prices and risks

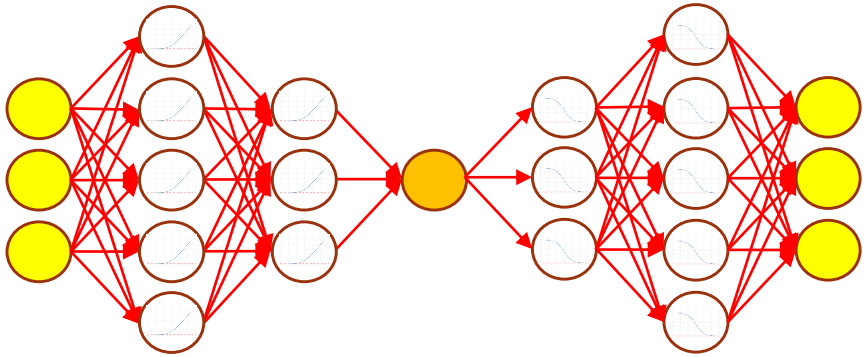
# Twin ANN

Combine value and risk prediction



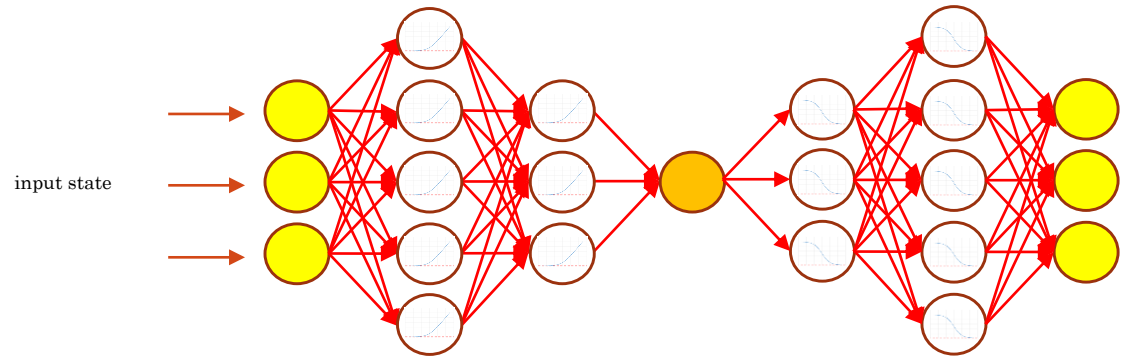
# Twin ANN

Combine value and risk prediction with a twin network



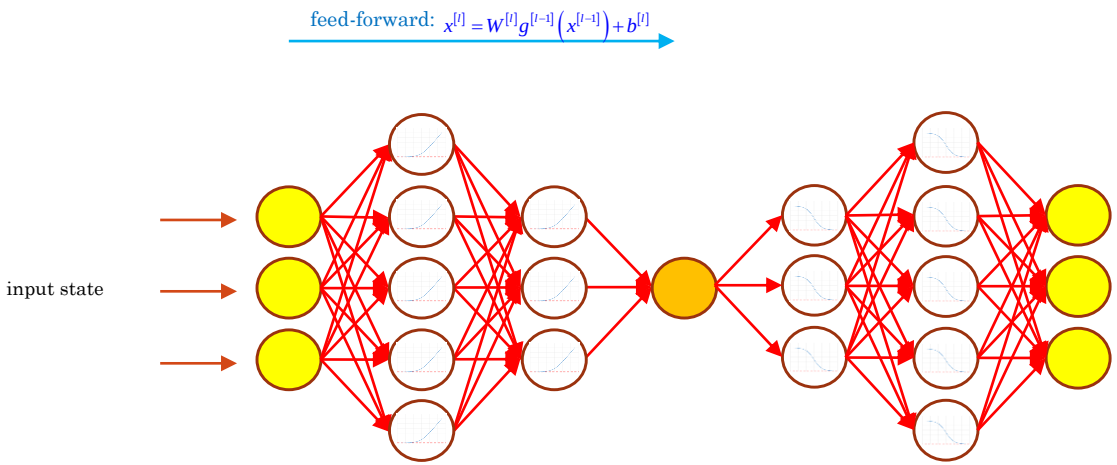
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction



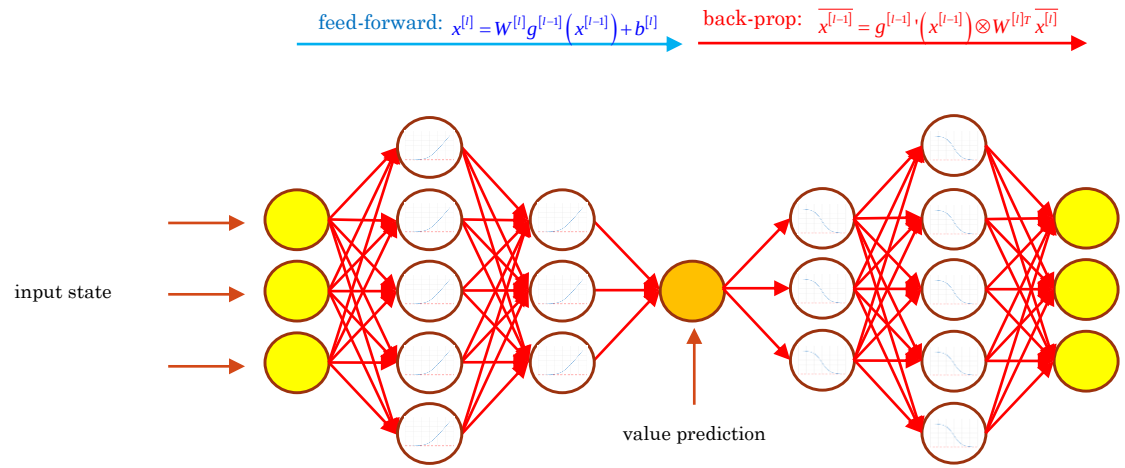
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction



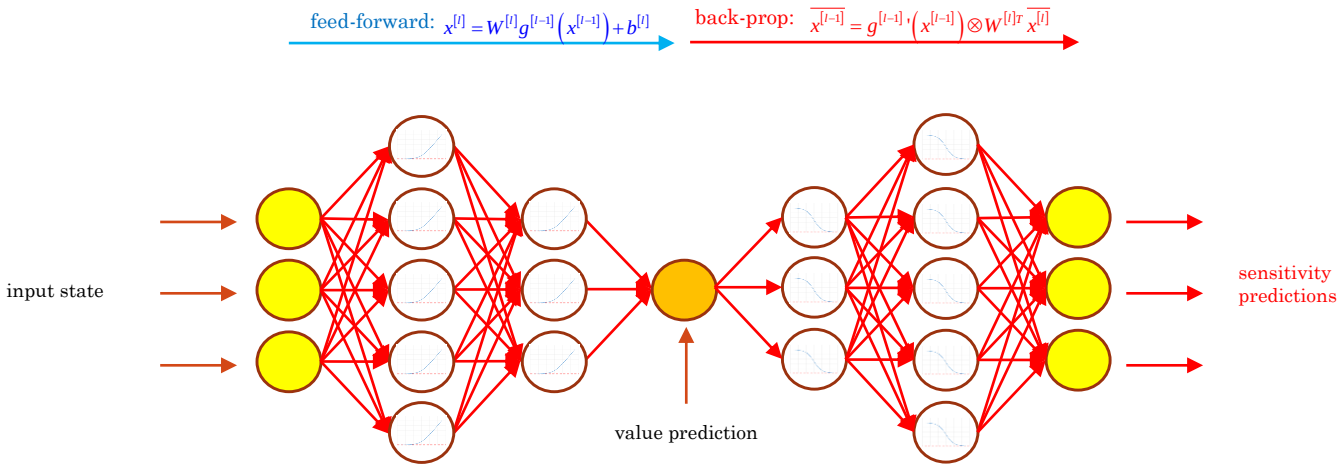
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction
And then risks with backprop equations through the dual network



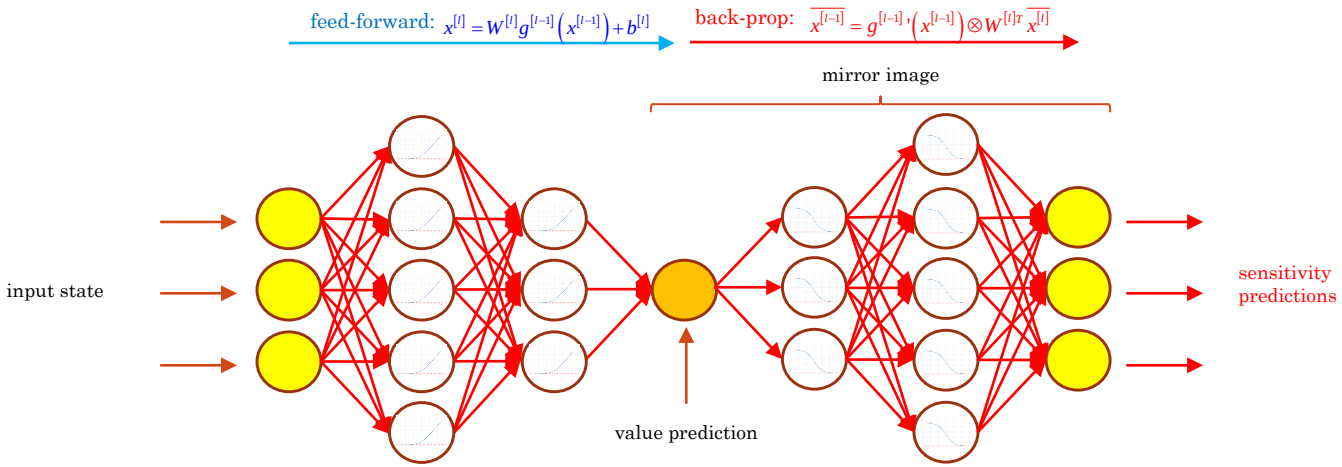
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction
And then risks with backprop equations through the dual network



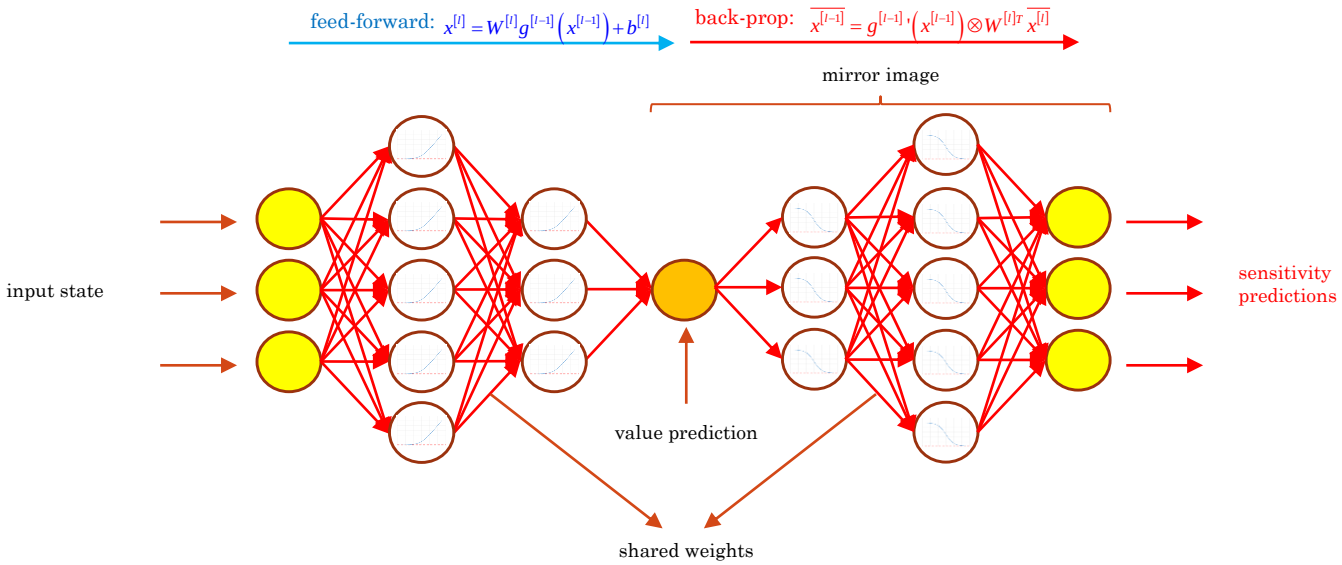
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction
And then risks with backprop equations through the dual network



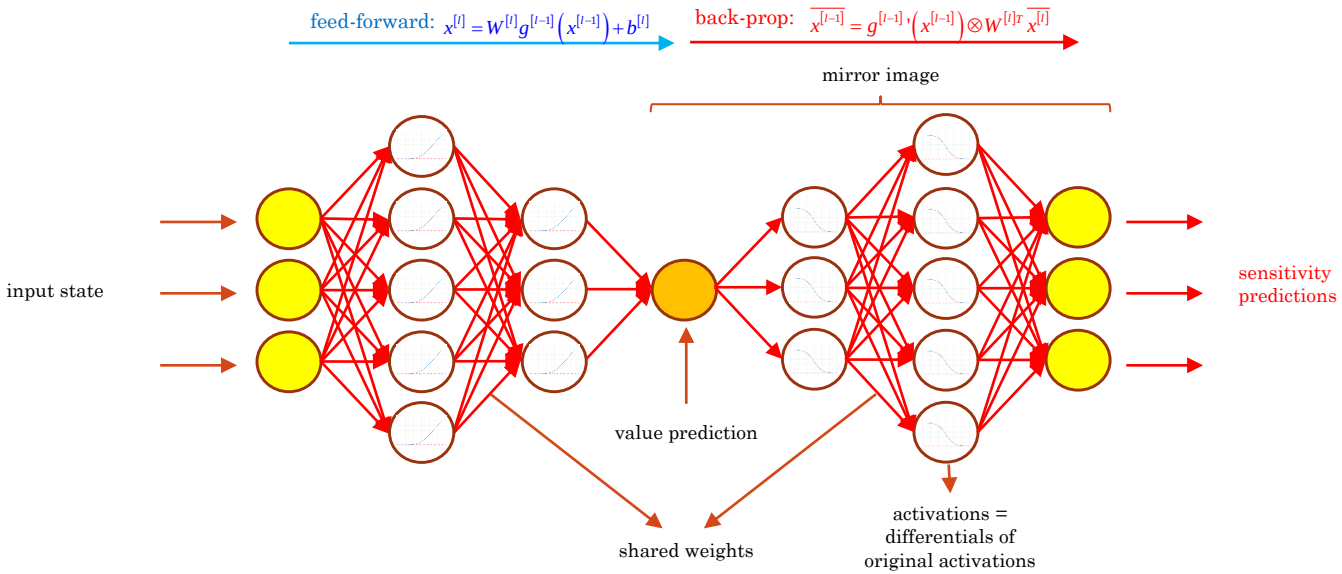
# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction
And then risks with backprop equations through the dual network



# Twin ANN

Combine value and risk prediction with a twin network
Predict values by feedforward induction
And then risks with backprop equations through the dual network



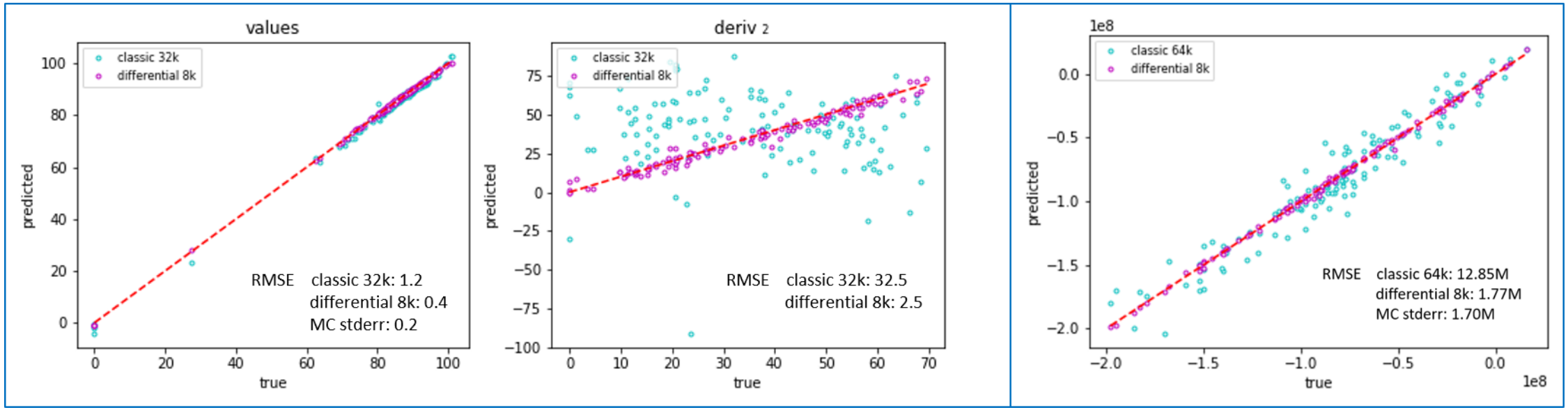
# Twin ANN: implementation

- Better understood with implementation code
- TensorFlow implementation available on GitHub (differential-machine-learning)

  https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialMLTF2.ipynb

- With applications and a discussion of implementation details
- Run it on your browser (or iPhone) **on Google's GPU servers**:

# Twin ANN: performance




**worst-of-four autocallable**

classical network predicts decent values with 32k examples but random deltas

twin network predicts almost perfect values with only 8k examples and solid deltas

**real-world netting set**

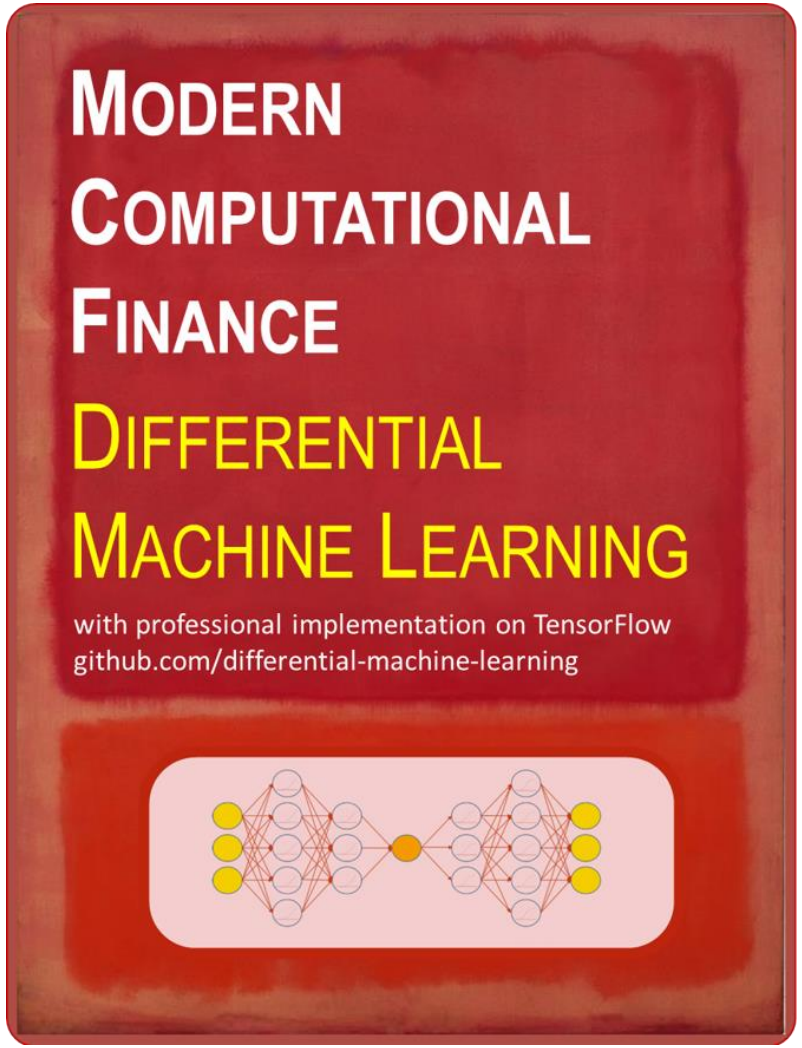
classical network overfits with 64k examples due to high dimensionality, it does converge but needs a much bigger dataset

twin network converges almost perfectly with only 8k examples

sources:
Risk.net: https://www.risk.net/cutting-edge/banking/7688441/differential-machine-learning-the-shape-of-things-to-come
GitHub: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialMLTF2.ipynb

Open in Colab

# Printing in 2022 with Chapman & Hall



# Thank you for your attention

Superfly Analytics

RiskMinds International

Excellence in Risk Management and Modelling, 2019
Winner: Superfly Analytics at Danske Bank

Risk Awards 2015

In-House system of the Year 2015
Winner: Superfly Analytics at Danske Bank