Modern Numerical Methods

An interactive presentation using Python

manos.venardos@gmail.com

Agenda

- Introduction
- Payoff & Density Differentiation
- Dependency Graphs
- Automatic Differentiation
- Machine Learning

Using many standard 3rd party libraries, and purpose-built analytics.

In [1]:
```
#Numerics & Analytics
import datetime
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from scipy.stats import norm
import pydot

#Graphs & widgets
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from IPython.display import Image, display, clear_output, HTML

#Option Pricing
import black_scholes as bs_model
import implied_vol as iv_model
```

Introduction

Risk management of derivatives businesses requires significant computational resources and advanced numerical methods in order to compute the plethora of risk and PnL measures required.

Consider a path-dependent option on a basket of $\#N_U$ underlyings, requiring $\#N_T$ points the path. Such a product is valued with Monte Carlo methods simulating $\#N_P$ independent paths

$$\hat{V} = \frac{1}{N_P} \sum_{p=1}^{N_P} \Pi(S(p))$$

where Π is the payoff at maturity and $S(p)$ the p-th simluated collection of asset paths. Typically 10s or 100s of thousands of path are simulated for this to reliably converge.

The cost of 1 valuation scales linearly with N_P and is approximately given by

$$c_V = c_f + (N_U \times N_T \times c_\omega + c_\Pi) \times N_P$$

where

- c_f is the fixed cost of setting up the simulation, due to e.g. a calibration
- c_ω is the cost of generating 1 random normal variate
- c_Π is cost of evaluating the payoff at expiry, given the simulated asset paths

We employ the multi-asset Black scholes model and assume interest rates and interim payments are zero. Each asset has a volatility Σ_u and the correlation matrix is ρ. The dynamics are in vector form given by

$$dS = CSdW$$

where $CC' = diag(\Sigma) \times \rho \times diag(\Sigma)$. Of interest are the parameters $S_0 \in \Re^{N_U}$, $\Sigma \in \Re_+^{N_U}$ and $\rho \in \Re^{N_U \times N_U}$.

Risk Management involves

- The calculation of numerous ***risk sensitivities***
- Trade re-valuation under different ***scenarios***

Of interest are $\frac{\partial V}{\partial S_0} \in \Re^{N_U}$, $\frac{\partial V}{\partial \Sigma} \in \Re^{N_U}$, $\frac{\partial V}{\partial \rho} \in \Re^{N_U \times N_U}$ of which only $\frac{1}{2} N_U(N_U - 1)$ are needed, and $\frac{\partial^2 V}{\partial S_0^2} \in \Re^{N_U \times N_U}$ of which only $\frac{1}{2} N_U(N_U + 1)$ are needed. When using finite differences

$$\frac{\partial V}{\partial \theta} \sim \frac{V(\theta+\delta) - V(\theta-\delta)}{2\delta}$$
$$\frac{\partial^2 V}{\partial \theta^2} \sim \frac{V(\theta+\delta) - 2V(\theta) + V(\theta-\delta)}{\delta^2}$$

the total computational cost amounts to $2N_U(2 + 2N_U) \times c_V$.

We are also interested in valuing the trade under $\#N_S$ different scenario shocks $(\delta_n^{S_0}, \delta_n^{\Sigma}, \delta_n^{\rho})_{n \le N_S}$.

The total computational cost of valuation, risk measures and scenarios is
$$C = (1 + 2N_U(2 + N_U) + N_S) \times c_V$$

And if we are interested in

- Equidistand spot price scenarios in the hypercube, in which case $\ln N_S \sim o(N_U)$
- Calculating risk measuers in such a hypercube, in which each scenario requires $o(N_U)$ or $o(N_U^2)$ valuations

It gets very costly, very soon.

Need efficient techniques for risk measures and scenarios.

We simulate correlated paths across 5 assets

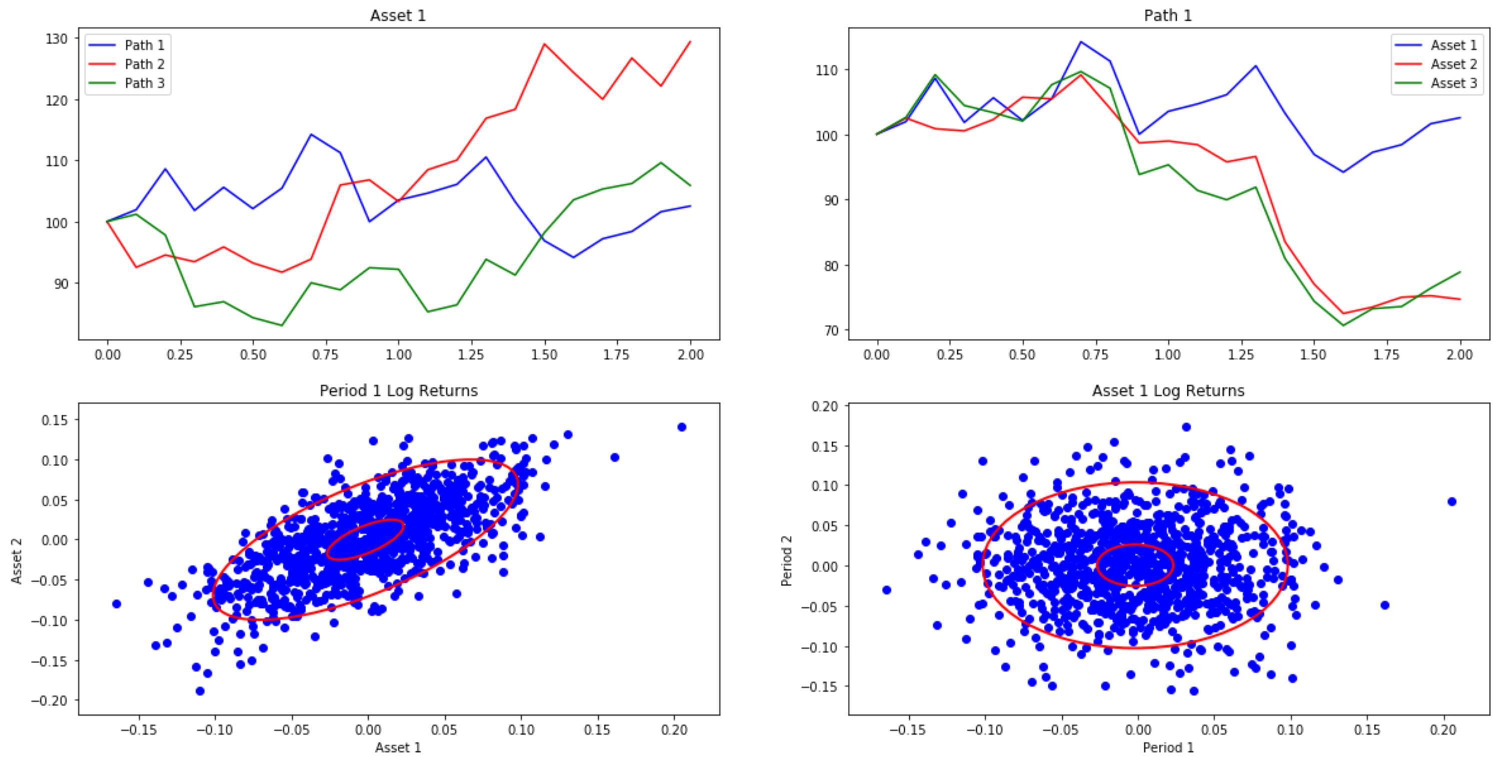
```
In [4]:
%%time
rate = 0.02
spots = [100] * 5
vols = [0.16] * 5
corrs = np.full((5,5), 0.7)
np.fill_diagonal(corrs, 1.0)
timepoints = np.linspace(0.1, 2.0, 20)
N = 1000

p = bs_diffusion(rate, spots, vols, corrs, timepoints, N)
```

```
CPU times: user 52.4 ms, sys: 44 ms, total: 96.3 ms
Wall time: 103 ms
```

And look at their statistical properties.

```
In [6]:
plot_bs_diffusion(time, p)
```



Risk Sensitivities

We focus on computing the quantity

$$\frac{\partial V}{\partial \theta} = \frac{\partial}{\partial \theta} E_0^Q \left[PV_0(T)\Pi(S) \right]$$

where S is in vector form the set of spot prices across assets and timepoints that the payoff depends on, and θ is a model related quantity.

We discuss 2 methods as an alternative to Finite Difference approximation

- Payoff differentiation
- Density differentiation

Payoff Differentiation

We interpret the simulated asset paths as a function of model quantities θ, so that

$$\frac{\partial}{\partial\theta} PV_0(T) E_0^Q \left[\Pi(S(\theta))\right] = PV_0(T) E_0^Q \left[\frac{\partial\Pi(S)}{\partial S}\frac{\partial S}{\partial\theta}\right]$$

assuming Π is well-behaved.

Here, $\frac{\partial\Pi(S)}{\partial S} \in \mathfrak{R}^{N_U \cdot N_T}$ and $\frac{\partial S}{\partial\theta}$ is a path derivative.

In the single asset BS case, $\ln S_t = \ln S_0 + \left(r - \frac{1}{2}\Sigma^2\right) t + \Sigma W_t$ and it is straight-forward to deduce

$$\frac{\partial S_t}{\partial S_0} = \frac{S_t}{S_0}$$
$$\frac{\partial S_t}{\partial \Sigma} = S_t \times (-\Sigma t + W_t)$$

Generalising to more general diffusions is possible but requires significantly more complicated frameworks e.g. see Malliavin calculus.

For a single asset Asian option $\Pi(S) = \max(A_T - K, 0)$ where $A_T = \frac{1}{N_T} \sum_{t=1}^{N_T} S_t$

$$\frac{\partial \Pi(S)}{\partial S} = \frac{1_{A_T > K}}{N_T} \times 1 \in \Re^{N_T}$$

Therefore

$$\frac{\partial V}{\partial S_0} = PV_0(T)E_0^Q\left[\frac{1_{A_T>K}}{S_0}A_T\right]$$

$$\frac{\partial V}{\partial \Sigma} = PV_0(T)E_0^Q\left[\frac{1_{A_T>K}}{N_T}\sum_{t=1}^{N_T} S_{t_i} \times (-\Sigma(t_i - t_{i-1}) + (W_t - W_{t-1}))\right]$$

where $\frac{\partial A}{\partial S_t}$ is the sensitivity to a single summand S_t, with all other summands $S_{t^*}, t^* \neq t$ fixed.

In summary,

- Each risk measure is an expected value, and the computational cost is half what FD requires
- It works for simple continuous payoffs for which we can calculate the vector derivative $\frac{\partial \Pi(S)}{\partial S}$ e.g. call, put, Asian. But will not work for discontinuous payoffs e.g. digital option

- It works for simple models for which we can compute the vector path derivative $\frac{\partial S}{\partial \theta}$. This generalises to more complex models, but needs significantly more advanced techniques

Note that the expectation $E_0^Q[]$ is taken w.r.t. the underlying stochastic normal process $\{W\}_{0 \le t \le T}$ whose probability distribution is independent of θ.

Density Differentiation

An alternative angle is to express the valuation as an integration over the risk neutral probability density q of the spot process $\{S\}_{0 \le t \le T}$, which is now itself a function of θ

$$\frac{\partial}{\partial \theta} PV_0(T) \int \Pi(S) q(S|\theta) dS = PV_0(T) \int \Pi(S) \frac{\partial q(S|\theta)}{\partial \theta} dS$$

Since $\frac{dy}{dx} = \frac{d \ln y}{dx} y$, we express this as

$$PV_0(T)\int \Pi(S)\frac{\partial q(S|\theta)}{\partial\theta}dS = PV_0(T)$$

$$\int \Pi(S)\frac{\partial \ln q(S|\theta)}{\partial\theta}q(S|\theta)dS$$

For the single asset BS case, S_T is distributed as

$$q(S|\theta) = \frac{1}{S\Sigma\sqrt{T}}\phi\left(\frac{\ln\frac{S}{K} - \left(r - \frac{\Sigma^2}{2}\right)T}{\Sigma\sqrt{T}}\right)$$

and from which we can calculate

$$\frac{\partial \ln q(S|S_0)}{\partial S_0} = \frac{\ln\frac{S}{S_0} - \left(r - \frac{\Sigma^2}{2}\right)T}{S_0\Sigma^2 T}$$

In summary,

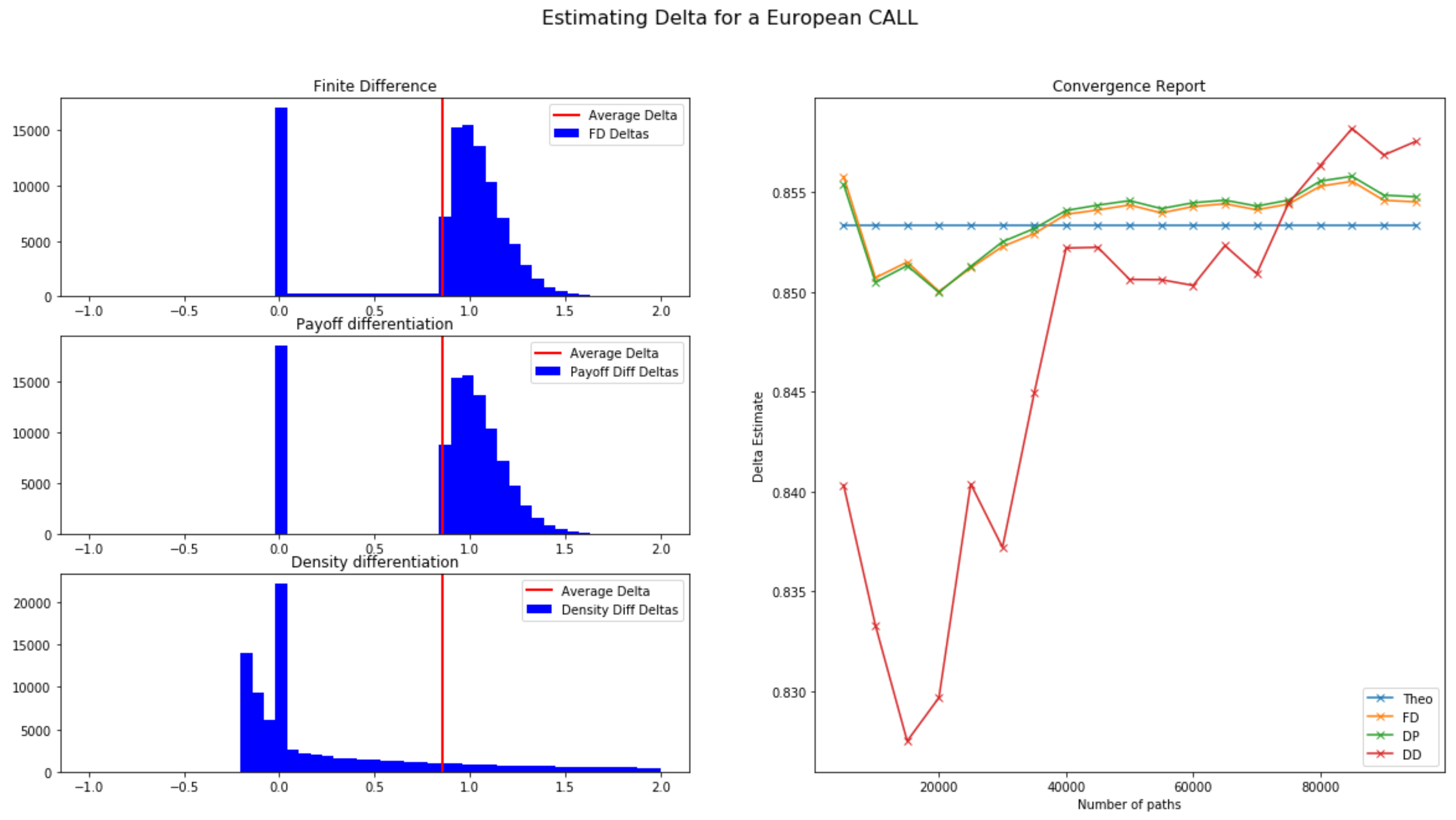
- Each risk measure is an expected value, and the computational cost is half what FD requires
- It works for all payoffs
- It works for simple models for which we can compute the desnity derivative $\frac{\partial \log q}{\partial \theta}$

Quiz 1 - Payoff Vs Density Differentiation

Which of the below statements is incorrect for a call option

1. When using path differentiation, the Delta estimate converges to a positive value as N_P increases
2. When using path differentiation, the Delta is positive in each path p
3. When using density differentiation, the Delta estimate converges to a positive value as N_P increases
4. When using desnity differentiation, the Delta is positive in each path p
5. With either technique, OTM paths contribute 0 to Delta

```
In [9]: df = plot_option_deltas(100.0, 0.16, 0.05, 90.0, 1.0, 'CALL', 100000, 97)
```



Dependency Graph

Sequence of relationships between intermediate variables and previous ones they depend on.

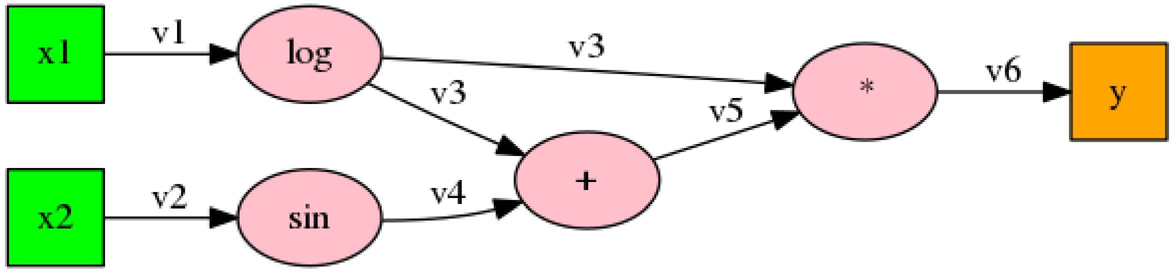
$$y = f(x_1, x_2) = \log(x_1)(\log(x_1) + \sin(x_2))$$

A likely implementation is

In [10]:
```
def f(x1, x2):
    v3 = np.log(x1)
    v4 = np.sin(x2)
    v5 = v3 + v4
    y = v3 * v5
    return y
```

This is represented as a ***Directed Acyclic Graph*** (DAG) where the ***nodes*** represent the operations and the ***edges*** the data flow.

In [13]:
```
view_pydot(g)
```



The Excel calculation engine is based on such a concept

- It knows each cell's ancestors
- When `x2` changes, the nodes `sin`, `+` and `*` need to be recalculated
- But `log` is independent, so the current value of `v3` will be re-used for free

Automatic Differentiation (AD)

AD is the programmatic logic that calculates the value of the derivative of a function (as opposed to the value of the function). AD isn't Symbolic Differentiation nor Numerical Differentiation.

Consider the DAG for `f` which consumes $x \in \Re^N$ and, via a sequence of intermediate operations, produces $y \in \Re^M$. AD is the code associated with computing

$$\frac{dy}{dx} \in \Re^{N \times M}$$

It should be possible because `f` ultimately uses elementary operations to transform vector variables to others i.e. $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{K-1} \rightarrow u_K$.

Quiz 2

In order to calculate the matrix product

$$\underbrace{A}_{1\times L} \times \underbrace{B}_{L\times M} \times \underbrace{C}_{M\times N}$$

1. It is better to calculate $A \times (B \times C)$
2. It is better to calculate $(A \times B) \times C$
3. It makes no difference

Quiz 3

In order to calculate the matrix product

$$\underbrace{A}_{L\times M} \times \underbrace{B}_{M\times N} \times \underbrace{C}_{N\times 1}$$

1. It is better to calculate $A \times (B \times C)$
2. It is better to calculate $(A \times B) \times C$
3. It makes no difference

Forward Propagation

Denote by $\dot{u_n}$ the derivative of intermediate variable u_n w.r.t. the input values u_0. Then

$$\dot{u_{n+1}} = D_n \dot{u_n}$$
$$D_n = \frac{\partial u_{n+1}}{\partial u_n}$$

Therefore,

$$\dot{u_K} = D_{K-1} D_{K-2} \ldots D_1 D_0 \dot{u_0}$$

Multiply iteratively from right to left. Nice and intuitive.

Note that

- Forward AD augments the `f` code with operations to calculate D_i and $\dot{u}_i$, next to calculating u_i
- The augmented code is swept forward to simultaneously evaluate f and f'
- To evaluate $\dot{u}_K \in \Re^{N \times M}$, N re-runs of the augmented code are needed
- Setting $\dot{u}_0 = (0, \ldots, 0, \underbrace{1}_{i\text{-th}}, 0, \ldots, 0) \in \Re^N$ will evaluate $\frac{\partial u_K}{\partial u_{0i}} \in \Re^M$
- Forward AD is efficient for $N << M$

A likely implementation `f` is

```
In [14]:
def f(x1, x2):
    v3 = np.log(x1)
    v4 = np.sin(x2)
    v5 = v3 + v4
    y = v3 * v5
    return y
```

```python
def f_dot_symbolic(x1, x2):
    t = np.log(x1)
    deriv = (2.0 * t/x1 + np.sin(x2)/x1, t * np.cos(x2))
    return deriv
```

Augment the code with variables to calculate the forward accumulation in a single forward sweep.

In [15]:

```python
def f_dot_ad_fwd(x1, x2, x1_dot, x2_dot):
    #Forward sweep to calculate f, D and dot
    v1, v2 = x1, x2;                                          v1_dot, v2_dot = 
x1_dot, x2_dot
    v3 = np.log(v1);  dv3_dv1 = 1.0 / v1;                     v3_dot = dv3_dv1
* v1_dot
    v4 = np.sin(v2);  dv4_dv2 = np.cos(v2);                   v4_dot = dv4_dv2
* v2_dot
    v5 = v3 + v4;     dv5_dv3 = 1.0;         dv5_dv4 = 1.0;   v5_dot = dv5_dv3
* v3_dot + dv5_dv4 * v4_dot
    v6 = v3 * v5;     dv6_dv3 = v5;          dv6_dv5 = v3;    v6_dot = dv6_dv3
* v3_dot + dv6_dv5 * v5_dot

    return (v6, v6_dot)
```

Simple illustration

In [16]:
```
a = 2.19
b = 1.65

print("f: " + str(f(a, b)))
print("f' Symbolic: " + str(f_dot_symbolic(a, b)))
```

```
f: 1.3959456652101419
f' Symbolic: (1.1710813315574145, -0.06202298688467458 6)
```

In [17]:
```
print("f' AD_Fwd arg1: " + str(f_dot_ad_fwd(a, b, 1.0, 0.0)[1]))    #deriv=(1,
0) for f' w.r.t arg1
print("f' AD Fwd arg2: " + str(f_dot_ad_fwd(a, b, 0.0, 1.0)[1]))    #deriv=(0,
1) for f' w.r.t arg2
```

```
f' AD_Fwd arg1: 1.1710813315574145
f' AD Fwd arg2: -0.062022986884674586
```

Forward AD needs 2 sweeps to evaluate f', one per input variable and using an appropriate $\dot{u}_0$ seed.

But no extra sweeps would be needed if `f` returned multiple outputs.

Backward Propagation

Denote by $\bar{u_n}$ the derivative of the output variable u_K w.r.t. the intermediate variable u_n. Then

$$\bar{u_n} = D_n^T \bar{u_{n+1}}$$

Therefore,

$$\bar{u_0} = D_0^T D_1^T \ldots D_{K-2}^T D_{K-1}^T \bar{u_K}$$

And multiply iteratively from right to left.

Note that

- Backward AD also augments the `f` code with operations to calculate D_i
- The augmented code is swept forward to evaluate f and D, and then backwards for f'

- To evaluate $\bar{u_0} \in \Re^{N \times M}$, M re-runs of the augmented code are needed
- Setting $\bar{u_K} = (0, \ldots, 0, \underbrace{1}_{i\text{-th}}, 0, \ldots, 0) \in \Re^M$ will evaluate $\frac{\partial u_{Ki}}{\partial u_0} \in \Re^N$
- Backward AD is efficient for $N >> M$

Augment the code with variables to calculate the backwards accumulation

In [18]:
```
def f_dot_ad_bwd(x1, x2, y_dot):
    #Forward sweep to calculate f and D, similar to Fwd AD (but no need to cac
lulate the dot)
    v1, v2 = x1, x2
    v3 = np.log(v1);   dv3_dv1 = 1.0 / v1
    v4 =  np.sin(v2);  dv4_dv2 = np.cos(v2)
    v5 = v3 + v4;      dv5_dv3 = 1.0;         dv5_dv4 = 1.0
    v6 = v3 * v5;      dv6_dv3 = v5;          dv6_dv5 = v3

    #Backward sweep to calculate bar
    v6_bar = y_dot
    v5_bar = dv6_dv5 * v6_bar
    v3_bar = dv6_dv3 * v6_bar + dv5_dv3 * v5_bar
    v4_bar = dv5_dv4 * v5_bar
    v2_bar = dv4_dv2 * v4_bar
    v1_bar = dv3_dv1 * v3_bar

    return (v6, (v1_bar, v2_bar))
```

Simple illustration

In [19]:

```
print("f: " + str(f(a, b)))
print("f' Symbolic: " + str(f_dot_symbolic(a, b)))
```

```
f: 1.3959456652101419
f' Symbolic: (1.1710813315574145, -0.062022986884674586)
```

In [20]:

```
print("f' AD_Bwd 1: " + str(f_dot_ad_bwd(a, b, 1.0)[1]))
```

```
f' AD_Bwd 1: (1.1710813315574145, -0.062022986884674586)
```

Backward AD needs 1 sweep to evaluate f' for both inputs, and a single value of $\bar{u}_0$ seed.

But extra sweeps would be needed if `f` returned multiple outputs.

AD Implementations

Manually interleaving code for $\dot{u}_i$, $\bar{u}_i$ and D_i inside the code for f is error prone and complex.

Software techniques have emerged to automate this process

- Source code transformation consumes the code for f and produces the code for f'
- Graph libraries allow for calculations to be represented as a DAG, and offer significant toolkit to process these

The Black Scholes closed form looks like this in TensorFlow

In [21]:
```
def bs_call_option_price_cf_graph(graph):
    with graph.as_default():
        #Declare placeholders for the graph inputs
        S=tf.placeholder(tf.float32,name='S'); V=tf.placeholder(tf.float32,nam
e='V'); K=tf.placeholder(tf.float32,name='K'); T=tf.placeholder(tf.float32,nam
e='T')
```

```
        #The usual BS formula, but using tf. notation
        Phi = tf.distributions.Normal(0.0, 1.0).cdf
        var = V**2 * T;  sqrtvar = tf.sqrt(var)
        d1 = (tf.log(S/K) + var / 2.0) / sqrtvar;  d2 = d1 - sqrtvar
        price =  S * Phi(d1) - K * Phi(d2)

        #AD to the rescue
        m_risk_1 = tf.gradients(price, [S,V]);  p_risk = tf.gradients(price, [
K,T]);  m_risk_2 = tf.gradients(m_risk_1[0], S)
        results = {'Price': price, 'Delta': m_risk_1[0], 'Gamma': m_risk_2[0],
'Vega': m_risk_1[1], 'dPrice_dK': p_risk[0], 'dPrice_dT': p_risk[1]}

    def calc(s, v, k, t):
        with graph.as_default(), tf.Session() as sess:
            return sess.run(results, {S: s, V: v, K: k, T: t})

    return calc
```

First, build the graph-based closed form pricer, once.

In [22]:
```
#Build the graph
bs_call_cf_graph = tf.Graph()
cf_pricer = bs_call_option_price_cf_graph(bs_call_cf_graph)
```

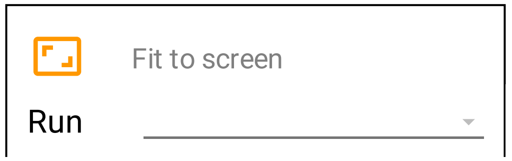
Then, invoke calculations with different arguments every time.

In [23]:
```
S = 100.0;   V = 0.16
K = 100.0;   T = 1.0;  PT = bs_model.CALL
```

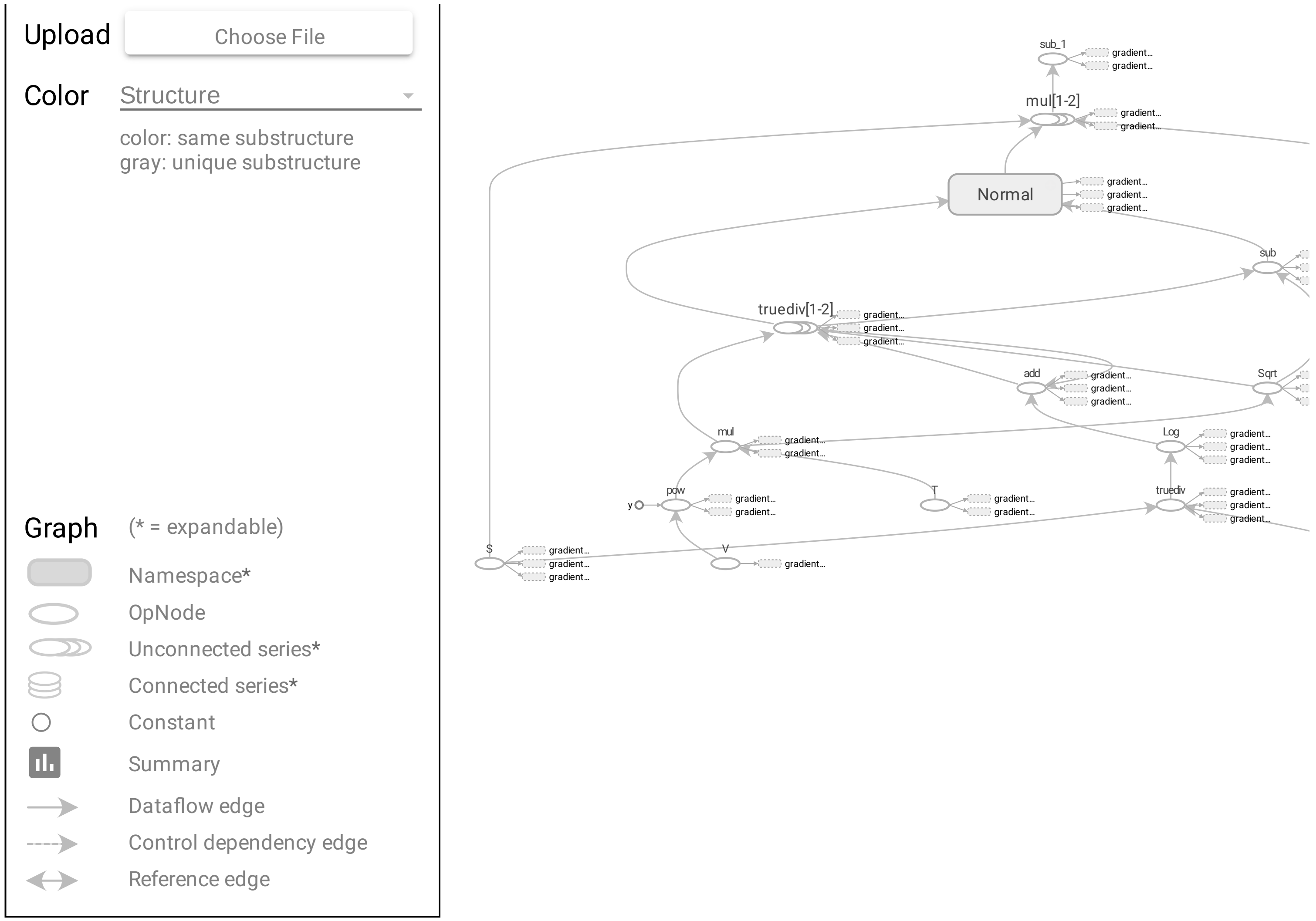
In [24]:
```
%%time
#Run the graph
print('AD CF Risk: ' + str(cf_pricer(S, V, K, T)))
print('CF Risk: ' + str(bs_model.option_risk(S, V, 0.0, K, T, PT)))
```

```
AD CF Risk: {'Price': 6.376278, 'Delta': 0.5318814, 'Gamma': 0.024854232,
'Vega': 39.766773, 'dPrice_dK': -0.46811864, 'dPrice_dT': 3.181342}
CF Risk: {'Price': 6.376274402797485, 'Delta': 0.5318813720139874, 'Gamm
a': 0.024854231594475557, 'Vega': 39.76677055116089}
CPU times: user 696 ms, sys: 7.82 ms, total: 704 ms
Wall time: 700 ms
```

In [26]:
```
view_tf(bs_call_cf_graph)
```



Main Graph


Upload
Choose File
Color
Structure
color: same substructure
gray: unique substructure
Graph
(* = expandable)
Namespace*
OpNode
Unconnected series*
Connected series*
Constant
Summary
Dataflow edge
Control dependency edge
Reference edge
sub_1
mul[1-2]
Normal
sub
truediv[1-2]
add
Sqrt
mul
Log
pow
y
T
truediv
S
V
gradient...

The Black Scholes monte carlo simulation looks like this in TensorFlow

In [27]:
```
def bs_call_option_price_mc_graph(graph):
    with graph.as_default():
        #Declare placeholders for the graph inputs
        S=tf.placeholder(tf.float32,name='S'); V=tf.placeholder(tf.float32,nam
e='V'); K=tf.placeholder(tf.float32,name='K'); T=tf.placeholder(tf.float32,nam
e='T')
        N=tf.placeholder(tf.int32,name='NbSims')

        #MC Simulation
        e = tf.random.normal((N, 1))
        S_T = S * tf.exp( (-V**2 / 2.0) * T + V * tf.sqrt(T) * e)
        C_T = tf.maximum(S_T[:,-1] - K, 0)
        price = tf.reduce_mean(C_T)

        #AD to the rescue - probably looks like payoff differentiation
        m_risk_1 = tf.gradients(price, [S,V]);  p_risk = tf.gradients(price, [
K,T]);
        results = {'Price': price, 'Delta': m_risk_1[0], 'Gamma': tf.constant(
'AD CANNOT COPE'), 'Vega': m_risk_1[1], 'dPrice_dK': p_risk[0], 'dPrice_dT': p
_risk[1]}
```

```
    def calc(s, v, k, t, n):
        with graph.as_default(), tf.Session() as sess:
            return sess.run(results, {S: s, V: v, K: k, T: t, N: n})

    return calc
```

First, build the graph-based Monte Carlo pricer, once.

In [28]:
```
#Build the graph
bs_call_mc_graph = tf.Graph()
mc_pricer = bs_call_option_price_mc_graph(bs_call_mc_graph)
```

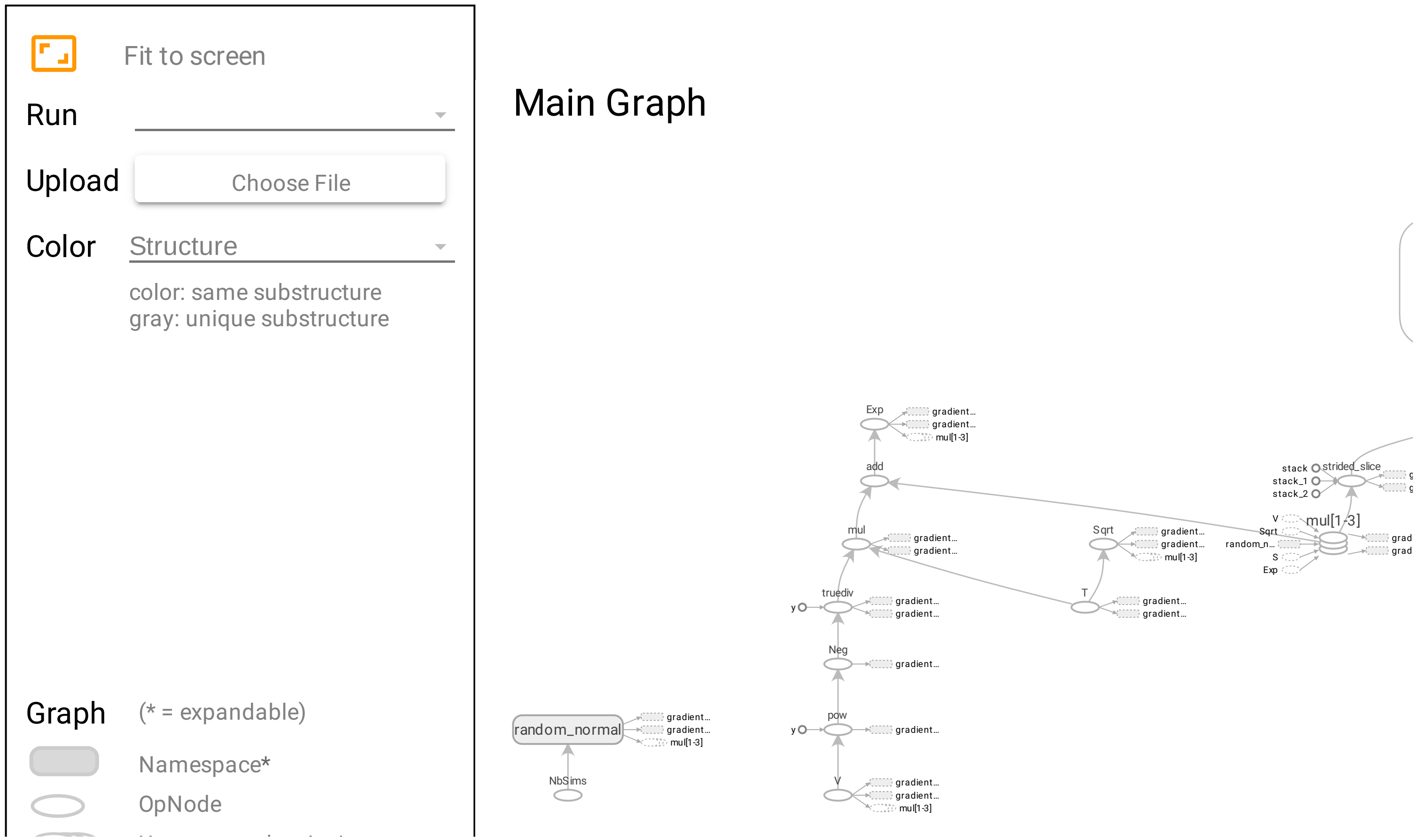
Then, invoke calculations with different arguments every time.

In [29]:
```
%%time
#Run the graph
print('AD MC Risk: ' + str(mc_pricer(S, V, K, T, 10000)))
print('CF Risk: ' + str(bs_model.option_risk(S, V, 0.0, K, T, PT)))
```

```
AD MC Risk: {'Price': 6.367797, 'Delta': 0.5241779, 'Gamma': b'AD CANNOT C

OPE', 'Vega': 39.769043, 'dPrice_dK': -0.46050322, 'dPrice_dT': 3.1815233}
CF Risk: {'Price': 6.376274402797485, 'Delta': 0.5318813720139874, 'Gamm
a': 0.024854231594475557, 'Vega': 39.76677055116089}
CPU times: user 115 ms, sys: 13 ms, total: 128 ms
Wall time: 124 ms
```

In [30]:
```
view_tf(bs_call_mc_graph)
```



Fit to screen

Run

Upload Choose File

Color Structure

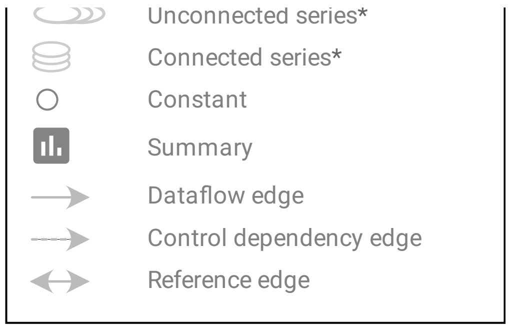
color: same substructure
gray: unique substructure

Graph (* = expandable)

Namespace*

OpNode

Main Graph



AD Performance

AD has gained significant popularity as a tool for calculating risk sensitivities

- Valuation models are ultimately a DAG with 100s of inputs (spot prices, volatilities, term structures etc.) and 1 output i.e. the price. AD backward propagation is very promising
- The computational cost tails off as the number of risk measures increases, in contrast to finite differences which increases linearly
- It requires a particular coding style and structure. Dedicated libraries have emerged to facilitate this, and some support hardware abstraction e.g. CPU vs GPU
- Discontinuities are a blocker, smoothing is a potential solution

Machine Learning

The set of techniques a computer system is using in order to perform a task, without resorting to explicit instructions, but instead relying on patterns detected during training.

Involves fitting a model in-sample, and then using it to make predictions out-of-sample.

The theory and algorithms behind ML are known for decades, but there is significant recent momentum due to

- Abundant and publicly available digital datasets to train the models e.g. the internet
- Abundant and affordable computational resources e.g. the cloud

Consider the familiar linear model

$$y = X'\beta + \epsilon$$

- Learning process: given a training dataset (X_T, y_T), the fitted model parameters are $\hat{\beta} = (X_T' X_T)^{-1} X_T' y_T$
- Prediction process: given an observed value x_p and the fitted model $\hat{\beta}$, the prediction is $y_p = x_p' \hat{\beta}$
- Performing a task without explicit instructions, based on learnt patterns
- Used extensively across disciplines and contexts

Depending on the nature of the dependent variable y

- ***Regression*** models a continuous variable e.g. $y =$ price
- ***Classification*** models a discrete-valued variable e.g. $y =$ {cat, dog, neither}

Depending on the availability and use of training data

- ***Supervised*** learning works with a complete dataset (X_T, y_T)
- ***Semi-supervised*** learning allows for some training results y_T to be missing
- ***Unsupervised*** learning only has access to X_T, and therefore focuses on data clustering and groupings

- ***Reinforcement*** learning receives a feedback rule from each prediction

A few ML models include

- Linear Models
- Artificial Neural Networks (ANN)
- Support Vector Machines (SVM)

The learning / fitting process employs a vast range of numerical algorithms, including steepest descent, least squares, genetic algorithms etc.

ANN Models

An interconnected set of neurons $n_{i,j}$ organised in layers $j \in \{1, 2, .., J\}$ with $I(j)$ neurons per layer. Each $n_{i,j}$ accepts as inputs the outputs from ancestors $n_{k,j-1}$, and produces as output

$$o_{i,j} = A\left(\sum_{k=1}^{I(j-1)} \left(w_{i,j,k} \times o_{k,j-1} + w_{i,j,0}\right)\right)$$

where A is an activation function e.g. the logistic function $\frac{1}{1+e^{-x}}$, the hyperbolic tangent $\tanh(x)$ and the Rectified Linear Unit (ReLU) $\max(x, 0)$. In this setup,

- $w_{i,j,k}$ are model parameters to be fitted as part of the learning process
- J, $I(j)$ and A are the geometry of the neural network, typically fixed during the learning process

The Universal Approxmation Theorem for ANNs states that a continuous $f : [0, 1]^N \to [0, 1]^M$ can be arbitrarily well approximated by a single hidden layer feed-forward ANN, given a reasonable activation function. For example, the single layer ANN is

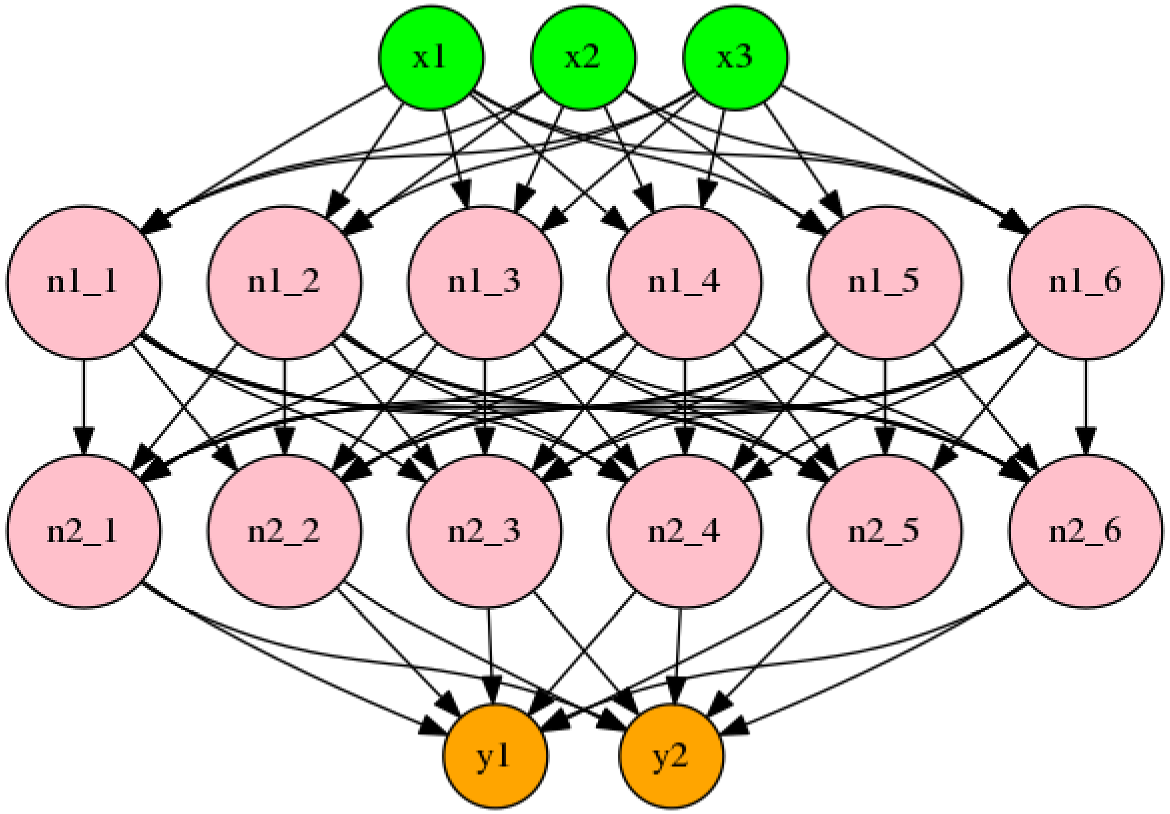
$$f_{ANN}(x) = \sum_{k=1}^{I} v_k A(w_{k,1}x + w_{k,0})$$

For a given $\epsilon > 0$, there exist I, A, w and v such that $|f(x) - f_{ANN}(X)| < \epsilon$.

So f_{ANN} is dense in the space of continuous functions.

Illustration of an ANN with 2 hidden layers

In [32]:
```
view_pydot(ann)
```



We illustrate the use of an ANN to learn the Black Scholes option pricing formula. First, create a training dataset (θ_T, c_T) where $\theta = [S, \Sigma, r]$ and c is the analytic BS price. We sample θ as multi-variate normal shocks around their base values.

In [34]:
```
means = np.asarray([0.0, 0.0, 0.0])
stdevs = np.asarray([V, 0.05, 0.02])
correls = np.asarray([[1.0, -0.7, 0.2], [-0.7, 1.0, 0.3], [0.2, 0.3, 1.0]])
R = 0.05

market_training = random_market([S, V, R], means, stdevs, correls, 1000, 97)
price_training = value_option(market_training, K, T, PT)
```

We then create the topology of an ANN, and fit it to the training dataset.

In [35]:
```
scaler = StandardScaler()
scaler.fit(market_training)
X_training = scaler.transform(market_training)

mlp = MLPRegressor(activation='tanh', solver='lbfgs')
mlp.fit(X_training, price_training)
```

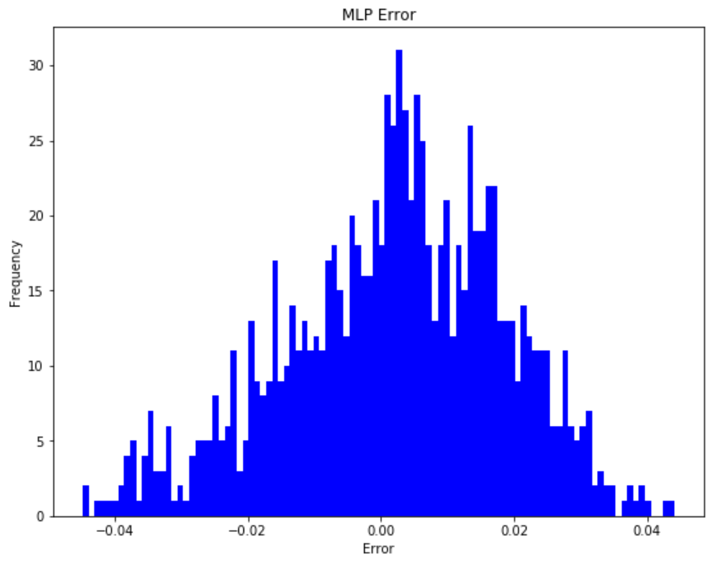
Out[35]:
```
MLPRegressor(activation='tanh', alpha=0.0001, batch_size='auto', beta_1=0.
9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=None, shuffle=True, solver='lbfgs', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
```

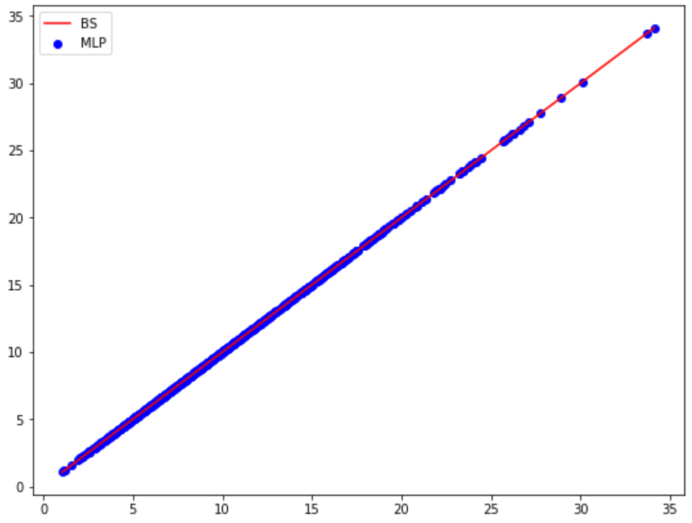
We then create an out-of-sample set θ_P and analyse the prediction error.

In [37]:
```
market_test = random_market([S, V, R], means, 0.5 * stdevs, correls, 1000, 98)
plot_ml_performance(market_test, K, T, PT, mlp, scaler)
```

```
R^2: 0.9999894887730484
```

ML Performance for European CALL

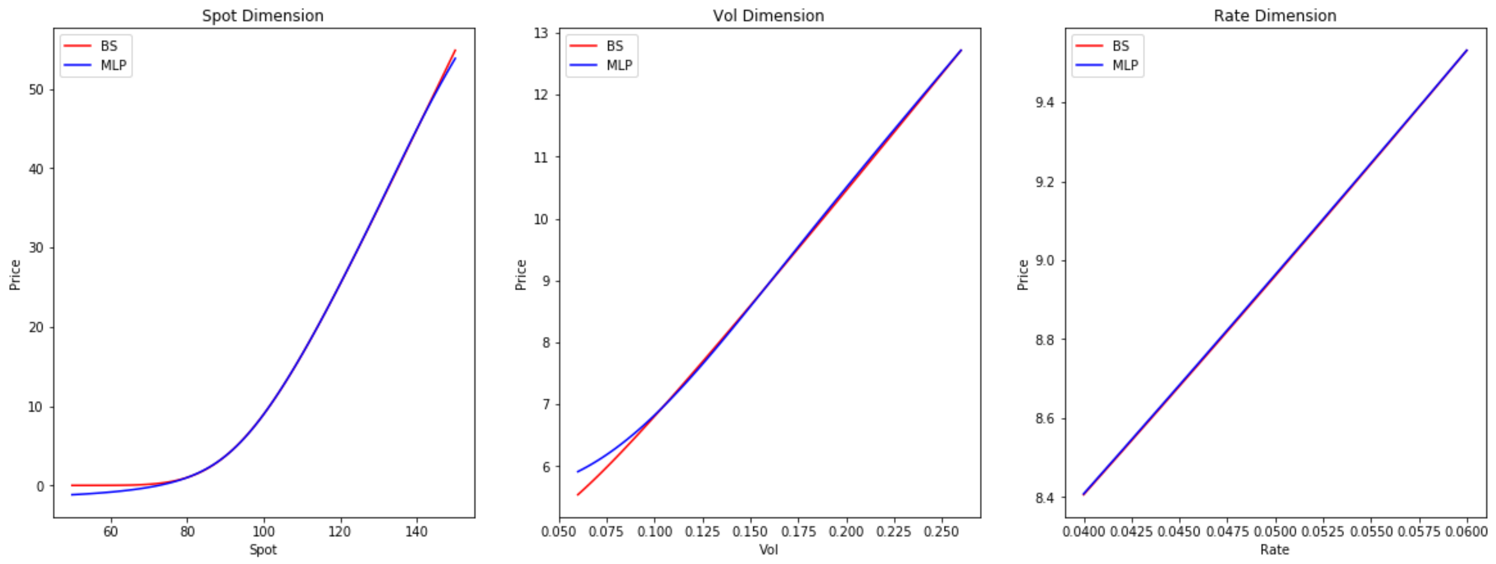




Finally, we create dedicated out-of-sample datasets spanning each model parameter independently, and benchmark the ANN vs the analytic formula.

In [39]:
```
plot_ml_marginal_performance([S, V, R], K, T, PT, mlp, scaler)
```

ML Performance for European CALL



ML models are highly commoditised nowadays

- They are available in many open-source libraries
- They can be used as services on the cloud, at low cost
- Their use appear straight-forward e.g. `model.fit(x_T, y_T); model.predict(x_P); model.score(x_P, y_P)`

ML models are essentialy DAGs. During the learning process, the optimizer needs to compute the derivative of an objective function w.r.t. the model parameters, so AD techniques are heavily employed.

Applications of ML in mathematical finance include

- As function approximations to complex valuation models
- Implied volatility parameterisation
- American Monte Carlo and early exercise decisions
- As approximators to PDE solutions
- Non-parametric hedging

With plenty more applications in financial services covering market making, statistical arbitrage, recommendation engines, middle and back office automation, chatbots etc.

With caveats

- The choice of the model matters i.e. the regressor or classifier type
- While a given regressor looks simple e.g. `MLPRegressor()`, it actually comes with a very long list of optional tuning parameters - and they matter
- Learning can require very large datasets
- Extrapolation can be problematic (interpolation too)

Thank you for your attention!