Deep Learning Derivatives Pricing

Giovanni Della Lunga
giovanni.dellalunga@unibo.it

Advanced Machine Learning for Finance

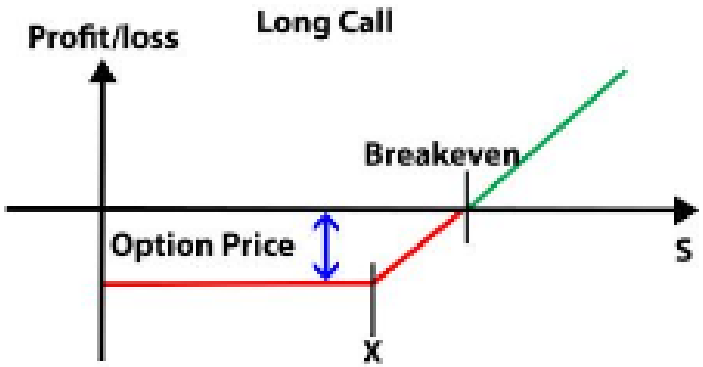
Bologna - April-May, 2022

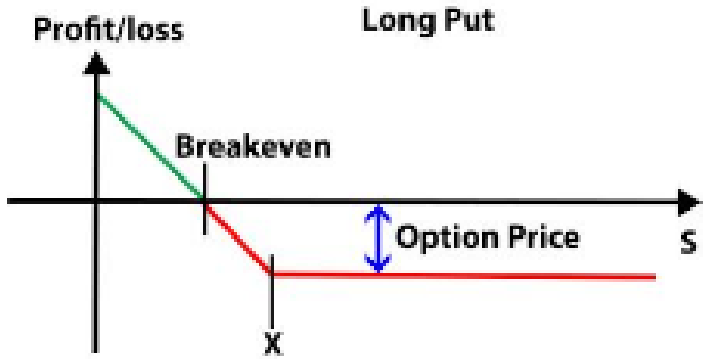
The Option Pricing Problem

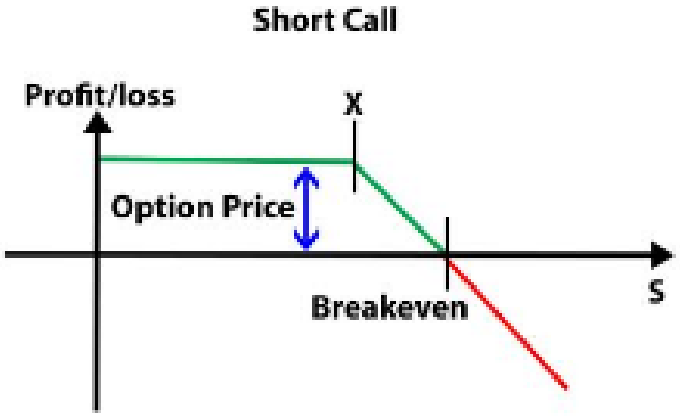
Call and Put Options

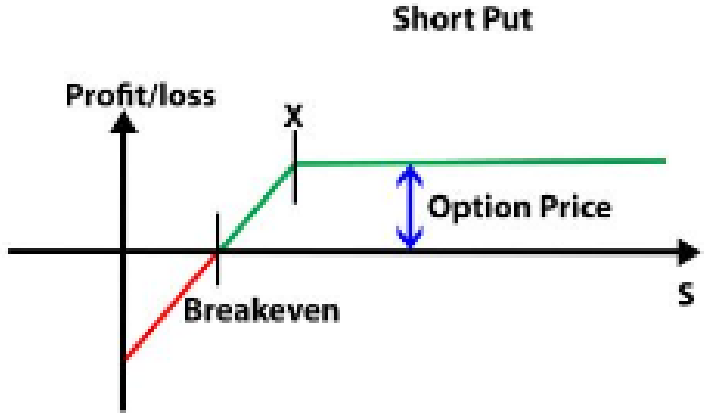
- Call option is an option to buy an asset on a certain future date (the maturity) for a certain price (the strike price)
- Put option is an option to sell an asset on a certain future date (the maturity) for a certain price (the strike price)

The Payoffs


Profit/loss
Long Call
Breakeven
Option Price
S
X

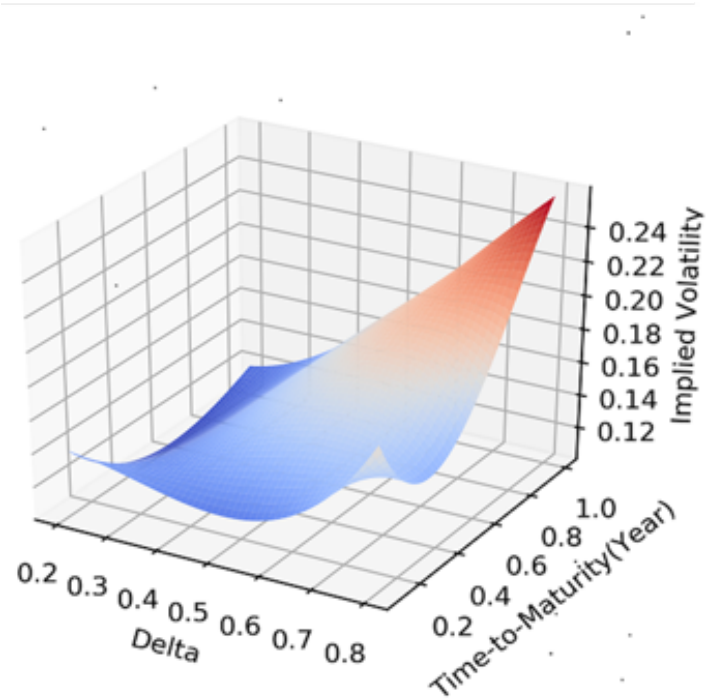

Profit/loss
Long Put
Breakeven
Option Price
S
X


Short Call
Profit/loss
X
Option Price
S
Breakeven

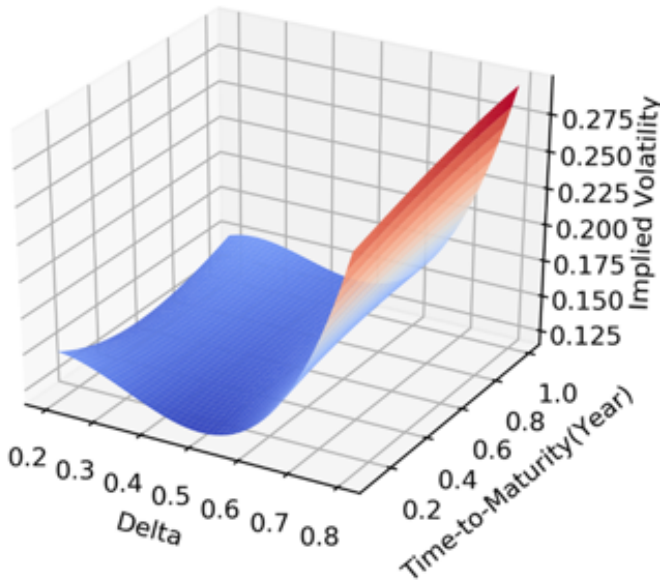

Short Put
Profit/loss
X
Option Price
S
Breakeven

Importance of volatility

The asymmetry in the payoff means that volatility, σ, is important in determining option prices. As volatility increases, the price of a call or put option increases.



Jan 31, 2019



June 25, 2019

Moneyness

- Moneyness is a measure of the extent to which an option is likely to be exercised
- Popular definitions (S = asset price, K = strike price)
- At-the money: $S = K$
- In-the money: $S > K$ for call options and $S < K$ for put options
- Out-of-the-money: $S < K$ for call options and $S > K$ for put options

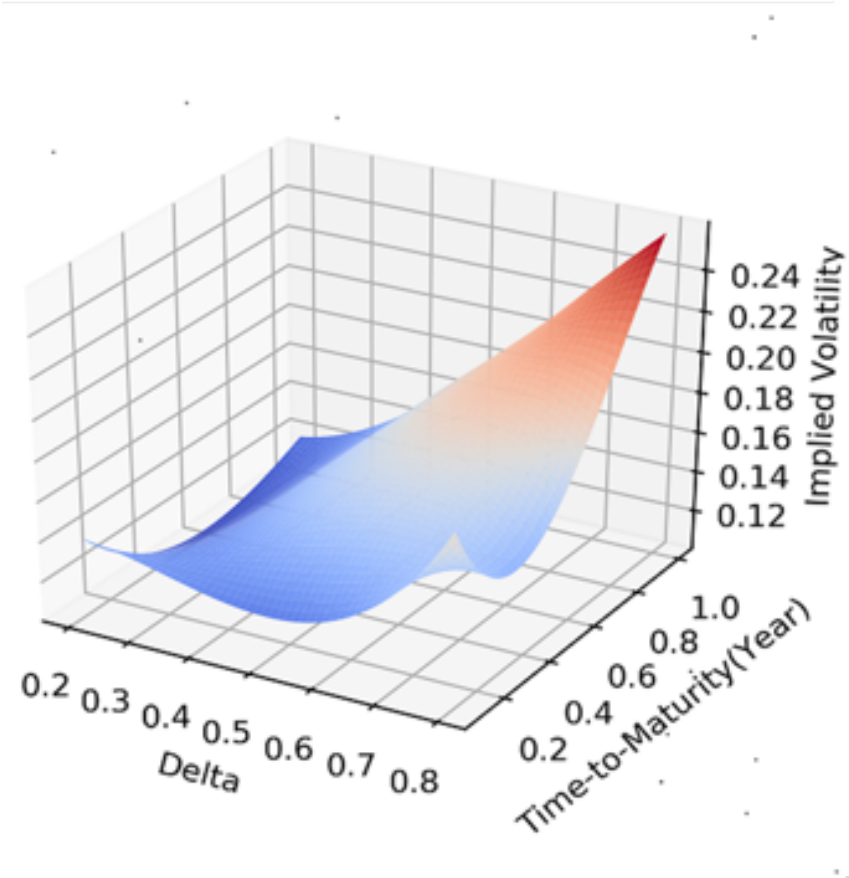
Implied volatilities

- The Black-Scholes-Merton price of an option depends on S, K, r, q, σ, and T.
- S, K, and T are known for an option r and q can be estimated from futures or forward contracts
- This means that σ is the only unknown.
- There is a one-to-one correspondence between option price and σ
- The implied volatility of an option is the volatility that when substituted into Black-Scholes-Merton formula gives the price of the option in the market

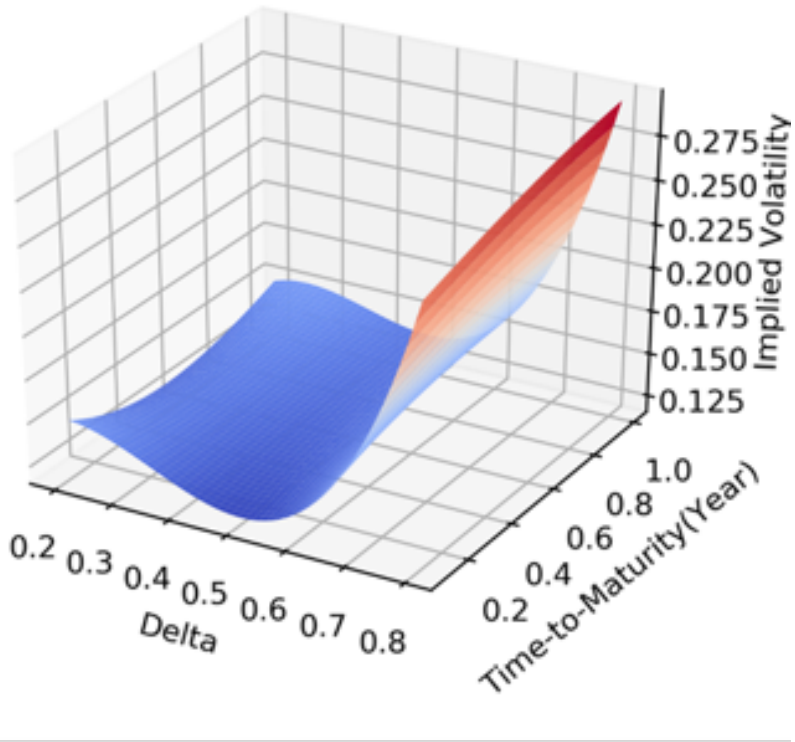
Implied volatilities (continued)

- If Black-Scholes-Merton was used for pricing, all options on an asset would have the same implied volatility
- In fact, there is quite a variation in implied volatilities
- Nevertheless implied volatilities are used to communicate prices and it is therefore important for traders to monitor implied volatilities
- The volatility surface shows implied volatilities as a function of Moneyness (measured by delta) and Time to maturity, T

Volatility Surfaces for S&P 500



Jan 31, 2019



June 25, 2019

Option Pricing Models

Why we need option pricing models?

- Not all plain vanilla options listed on the market are sufficient for customer needs;
- **exotic options** are generally not as actively traded as plain vanilla options
- As a result, a model is required for pricing;
- A variety of different models are used in practice.

Option Pricing Models

Two conditions that traders would like the model to satisfy are:

- (A) The stochastic behavior assumed for the underlying asset price should correspond reasonably well to its observed behavior, and
- (B) The volatility surface derived from the model should be reasonably consistent with the volatility surface used to price plain vanilla options.

Option Pricing Models

- Two categories of models that are used in practice can be distinguished.
- The models in the first category focus on condition (A) by assuming a process for the asset price that is roughly consistent with its observed behavior.
- The models have parameters that can be chosen to provide an approximate fit to the current volatility surface.
- Models in the second category focus on condition (B) and are designed to be exactly consistent with the current volatility surface.

Option Pricing Models

- The usual approach to implementing models in the first category is to choose model parameters to fit the volatility surface as closely as possible.
- This approach, which we refer to as the **model calibration approach** or **MCA**.
- A drawback of the approach is that some of the points on the volatility surface are likely to be more important than others for any particular exotic option that is considered.
- It is of course possible to vary the weights assigned to different points on the volatility surface according to the instrument being valued.
- However, it is difficult to determine in advance what these weights should be, as a result, the points are usually given equal weight when model parameters are determined.

Deep Learning Option Pricing

Deep Learning Option Pricing

- A model is essentially a sort of complex mapping from a set of parameters to the market price;
- Let's say, from a very general point of view, that we have a contingent claim $\mathcal{V}$ that depends on D parameters:

$$\mathcal{V}(\mathbf{x}),\ \mathbf{x} = \{x_1, \ldots, x_D\} \in \mathbb{R}^D$$

- We want to **approximate the pricing function with a neural network**.
- Remember that we can always consider the NN as a sort of mapping function

$$\Phi : \mathbb{R}^D \to \mathbb{R}$$

trained to compute prices given a point in $\mathbb{R}^D$ representing a particular set of parameters.

Deep Learning Option Pricing

- The use of a NN as a pricing functions has a number of advantages.
- The first is that in this way we are able to compute efficiently thousands of prices in a small amount of time, even when the derivative contract has complicated conditions and when the model is complex.
- This comes with the downside that the neural network **may introduce systematic errors** that could affect our estimation of the sensitivities in a number of ways.

Deep Learning Option Pricing

- The second advantage is that instead of training the network on the model parameters, which in general are not observable, we could train the network **using data that is directly observable in the market**.
- For example quotes and trades by market participants provide points on the volatility surface.
- Interpolating between these points as necessary, a trader can derive a reasonable estimate of the implied volatility appropriate for any new plain vanilla European or American option that is of interest.
- The volatility surface derived from the Black–Scholes–Merton model is a convenient interpolation tool for doing this.

Deep Learning Option Pricing

This allows us to define a new type of pricing approach. We refer to this approach as the **volatility feature approach** or **VFA**.

- We create a neural network where the inputs are **the volatility surface points** and **the exotic option parameters** and **the target is the price**.
- We randomly generate many sets of parameters for (a) the model under consideration and (b) the exotic option under consideration.
- For each set of parameters, a volatility surface and a price for the exotic option are calculated.
- The neural network is then constructed.

The model parameters are used to create the training set, but **they are not inputs to the neural network**.

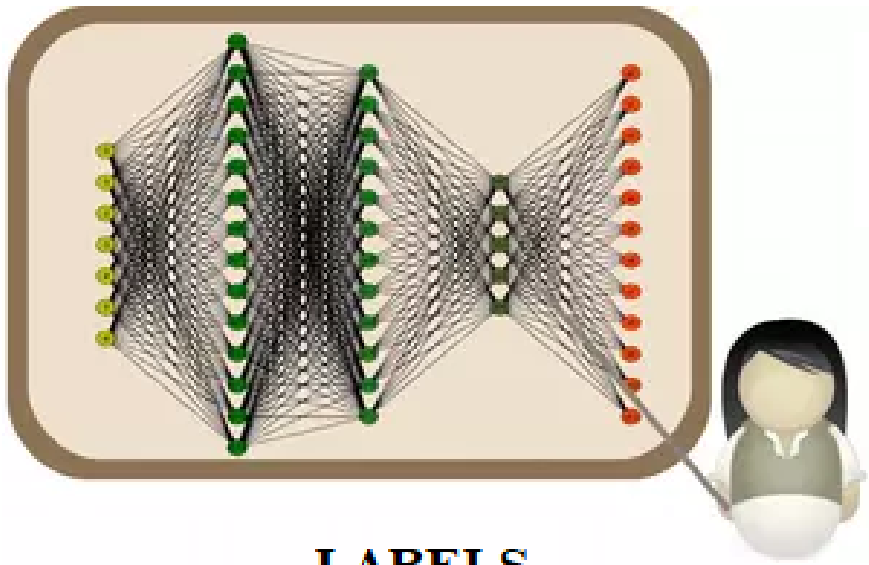
Deep Learning Option Pricing

TRAINING THE NETWORK - MCA APPROACH

FEATURES

OPTION PARAMETERS
- Strike
- Maturity

MODEL PARAMETERS
- Sigma



LABELS

Theoretical Option Prices

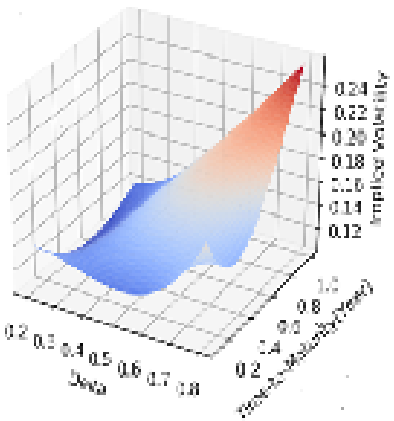
Deep Learning Option Pricing

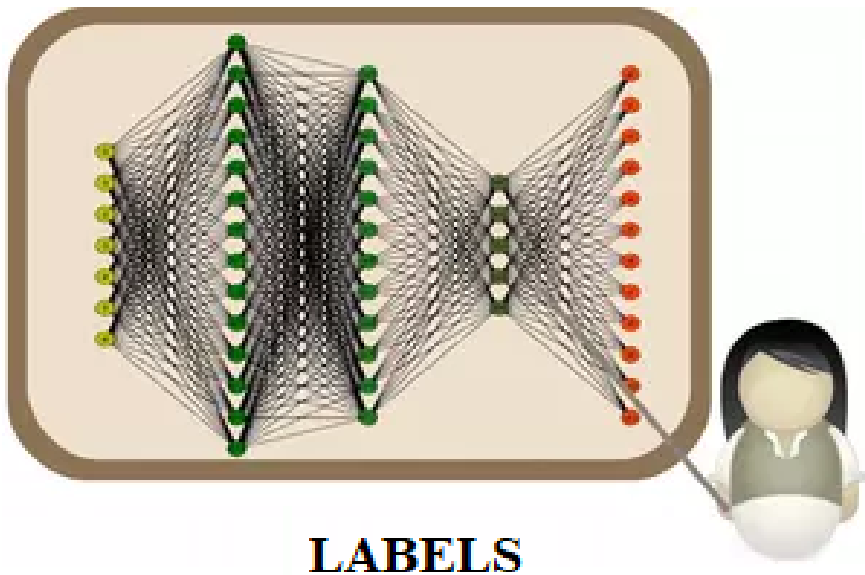
TRAINING THE NETWORK - VFA APPROACH

FEATURES

OPTION PARAMETERS
- Strike
- Maturity

MARKET PARAMETERS
- Volatility SUrface





LABELS

Theoretical Option Prices

Deep Learning Option Pricing

- The VFA approach is a compromise between models that satisfy condition (A) and those that satisfy condition (B).
- It retains some of the structure of an analyst's preferred model while learning which points on the volatility surface are most important for the exotic option being valued.
- The idea behind this method is that by sampling the volatility smile with enough points one should be able to obtain all the informations that would be contained in the model parameters, which is an assumption already implicitly made by market practitioners when they calibrate their model parameters on the volatility smile.
- For this reason, it produces prices for exotic options that are more consistent with the prices of vanilla options than MCA.

Deep Learning Option Pricing

Let's call $\Pi(\vec{\alpha}, G(\vec{g}))$ the function pricing a position G with a model described collectively by the parameters $\vec{\alpha}$, and the position described by the parameters $\vec{g}$. To be clear, in case of an option $\vec{g}$ could be the the pair maturity and strike.
A standard ML exercise would call for

- generating, according to some random rule, a set of parameters $\vec{\alpha}_n,\ \vec{g}_n\ 1 \leq n \leq N$
- for each $\vec{\alpha}_n\, \vec{g}_n$ compute the function pricing a derivative G with a model described by the parameter set $\vec{\alpha}_n$. We define the total parameter set as $\Pi_n := \Pi(\vec{\alpha}_n, G(\vec{g}_n))$

Deep Learning Option Pricing

Iterating the procedure described above, we can build a large matrix

Features (Regressors)		Label (Target)
α_1	g_1	Π_1
α_2	g_2	Π_2
	...	
α_N	g_N	Π_N

and use it to train a neural network that, if all goes well, will learn the map

$$\phi_{NN} : \vec{\alpha}, \vec{g} \rightarrow \Pi(\vec{\alpha}, G). \tag{1}$$

A Worked Example

Working Path

During these lessons we will develop the approach described through 3 examples:

- Description of the training process of a network using the Black and Scholes model;
- Training of a stochastic volatility model (Heston) starting from the data of the model itself (MCA) and ...
- Training of a model starting from market data (volatility surface) following the VFA approach;

The network will be trained on a large synthetic dataset generated by drawing each parameter from an appropriate distribution and by calculating the corresponding exact vanilla option price (and the implied volatility surface for the last example). The parameter's sampling will be done using the **Latin Hypercube Sampling (LHS)** approach.

Creation of the synthetic sets

What is Latin Hypercube Sampling?

- Latin hypercube sampling is a method that can be used to sample random numbers in which samples are distributed evenly over a sample space.
- The idea behind one-dimensional latin hypercube sampling is simple: Divide a given CDF into n different regions and randomly choose one value from each region to obtain a sample of size n.
- It is a sort of **stratified sampling**
- The benefit of this approach is that **it ensures that at least one value from each region is included in the sample**.

Creation of the synthetic sets

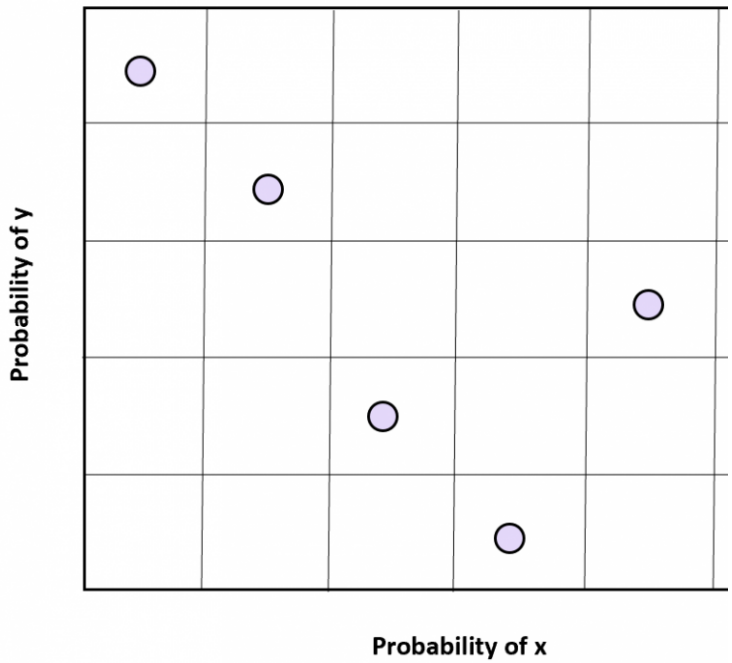
What is Latin Hypercube Sampling?

- Suppose we'd like to obtain a sample of 2 values from a dataset that is normally distributed with a mean of 0 and a standard deviation of 1.
- If we used a simple random number generator to obtain this sample, it's possible that both values could be greater than 0 or that both values could be less than 0.
- However, if we used latin hypercube sampling to obtain this sample then it would be guaranteed that one value would be above 0 and one would be below 0 because we could specifically partition the sample space into one region with values above 0 and one region with values below 0, then select a random sample from each region.

Creation of the synthetic sets

What is Latin Hypercube Sampling?

- We can easily extend the idea of one-dimensional latin hypercube sampling into two dimensions as well.
- For two variables, x and y, we can divide the sample space of each variable into n evenly spaced regions and pick a random sample from each sample space to obtain random values across two dimensions.



Creation of the synthetic sets

What is Latin Hypercube Sampling?

- To perform latin hypercube sampling in greater dimensions, we can simply extend the idea of two-dimensional latin hypercube sampling into even more dimensions.
- Each variable is simply split into evenly spaced regions and random samples are then chosen from each region to obtain a controlled random sample.
- The main advantage of latin hypercube sampling is that it produces samples that reflect the true underlying distribution and it tends to require much smaller sample sizes than simple random sampling.
- This method of sampling can be particularly advantageous if you're working with data that has a high number of dimensions and you need to obtain random samples that are sure to reflect the true underlying distribution of the data.

Creation of the synthetic sets

- To train the network to approximate the pricing function, we need a training dataset.
- For the network approximating the Black Scholes pricing function, we created one by drawing T, K/S_0 and σ using LHS (Latin Hypercube Sampling) within the ranges usually found in real market.

```
# Lower and upper boundaries for each parameter
bounds = {  "T"     : [1./12., 2.00]
          , "Sigma" : [ .01   ,  .80]
          , "Strike": [ .4    , 1.20]
         }
# Number of Observations
NUM = 100000
# Random number generator
rand = np.random.RandomState(42)
# Latin Hypercube Sampling
xDF = lhs_sampling(rand, NUM, bounds=bounds)
xDF.head()
```

Creation of the synthetic sets

LHS Function Definition

```
from smt.sampling_methods import LHS

def lhs_sampling(rand, NUM, bounds=None):
    mInt = 2**15
    MInt = 2**16
    kw   = list(bounds)
    # builds the array of bounds
    limits = np.empty(shape=(0,2))
    for k in kw:
        limits = np.concatenate((limits, [bounds[k]]), axis=0)

    sampling = LHS(xlimits=limits, random_state=rand.randint(mInt,MInt))
    x        = sampling(NUM)
    X = pd.DataFrame()
    for n in range(len(kw)):
        tag    = kw[n]
        X[tag] = x[:,n]

    return X
```

Creation of the synthetic sets

Black and Scholes Price Generator

```
def gen(NUM, lhs):
    x    = lhs
    __tStart = time.perf_counter()
    S0       = np.full(NUM, 1.0, dtype = np.double)
    r        = np.full(NUM, 0.0, dtype = np.double)
    price    = np_euro_call(S0, r, x["T"], x["Sigma"], x["Strike"])
    __tEnd = time.perf_counter()
    print("@ %-34s: elapsed %.4f sec" %("NP pricing", __tEnd-__tStart) )

    df = pd.DataFrame(x)
    df["Price"] = price

    return df
```

Creation of the synthetic sets

- As for the network approximating the pricing function in the Heston model with the MCA approach, a synthetic set has been created by drawing Heston Model parameters with LHS without accounting for the Feller condition.
- We also trained a network to calculate the price of a vanilla put option in the Heston model using the VFA approach with the contract parameters T and K/S_0 and the implied volatility smile sampled on the 9 moneyness values
  $K/S_0 \in \{0.4; 0.55; 0.7; 0.85; 1.0; 1.15; 1.3; 1.45; 1.6\}$.
- The implied volatility smiles are generated using the same parameters as those generated in the dataset for the MCA.

Creation of the synthetic sets

Heston Price Generator (MCA Approach)

```
def parms_gen( lhs = None, Xc=10, strikes=None):
    x = lhs
    NUM = len(x["T"])
    for m in range(NUM):
        K      = x["Strike"][m]
        fwPut = HestonPut( St      = 1.0, Strike = K, T = x["T"][m]
                         , kappa   = x["k"][m], theta  = x["theta"][m]
                         , sigma   = x["sigma"][m], v0  = x["v0"][m]
                         , r       = 0, rho = x["rho"][m], Xc = Xc)
        for tag in list(x):
            X[tag][n] = x[tag][m]
        X["Price"][n] = fwPut
        n += 1
    df = pd.DataFrame()
    for s in X.keys(): df[s] = np.copy(X[s][0:nSamples])
    return df
```

Creation of the synthetic sets

Heston Price Generator (VFA Approach)

```
def mkt_gen( lhs = None, kw = None, Xc=10, strikes=None):
        ......................

    for m in range(NUM):

            ..........

        fwPut = HestonPut(Fw, K, T, Kappa, Theta, sgma, Ro, Rho, 0.0, Xc)

        vol = build_smile(strikes, Fw, T, Kappa, Theta, sgma, Ro, Rho, Xc)

        for k in strikes:
            tag = "k=%5.3f" %k
            X[tag][n] = vol[tag]
        X["Price"][n]  = fwPut
        X["Strike"][n] = K
        X["T"][n]      = T

            ..........
```

Creation of the synthetic sets

Heston Price Generator (VFA Approach)

```
from Lib.euro_opt import impVolFromFwPut

def build_smile(strikes=None, Fw=1.0, T= 1.0, Kappa=1.,
                Theta=1., sgma=1.0, Ro=0.0, Rho=0.0, Xc=10):
    vol = {}
    for k in strikes:
        tag = "k=%5.3f" %k
        fwPut = HestonPut(Fw, k, T, Kappa, Theta, sgma, Ro, Rho, 0.0, Xc)
        if fwPut < max( k-Fw, 0.0): return None
        vol[tag] = impVolFromFwPut(price = fwPut, T = T, kT = k)

    return vol
```

Network Architecture

- All the neural networks used are composed of three dense layers of 200 neurons with 'relu' activations.
- For the neural network calculating the vanilla option prices in the Black Scholes model, the features used are the time to maturity T, the moneyness K/S_0 and the asset volatility σ.
- For the network using the MCA approach to the Heston model the features are the time to maturity T, the moneyness K/S_0 and the five model parameters $\{\kappa; \theta; \nu_0; \xi; \rho\}$.
- For the network calculating the prices in the Heston model using the VFA the features used are again the time to maturity T, the moneyness K/S_0 and 9 samples of the implied volatility smile calculated in $K/S_0 \in \{0.4; 0.55; 0.7; 0.85; 1.0; 1.15; 1.3; 1.45; 1.6\}$.

Network Architecture

Black and Scholes Pricing Network

```
def model_builder( inputShape = (1,)):
    model = Sequential()
    model.add(Dense(200, activation='relu', input_shape=inputShape))
    # Add one more hidden layer
    model.add(Dense(200, activation='relu'))
    # Add one more hidden layer
    model.add(Dense(200, activation='relu'))
    # Add an output layer
    model.add(Dense(1))
    # Model output shape
    print("model.output_shape: %s" %(str(model.output_shape)))
    # Model summary
    print("Model.summary"); model.summary()
    # Model config
    print("Model.config"); model.get_config()
    model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
    return model
```

Let's code ...

Understanding Volatility Surface Movements

Volatility Surface Movements

- When the price of the underlying asset increases (decreases), implied volatilities tend to decrease (increase)
- However not all implied volatilities change by the same amount
- This explains the variation in volatility surfaces

Volatility Surface Movements

Understanding how the volatility surface moves is important for a number of reasons:

- It can help a trader hedge her/his exposure;
- It can help a quant determine a stochastic volatility model reflecting how options are priced in the market;
- It can help a trader adjust implied volatilities in a market where asset prices are changing fast.

Understanding Volatility Surface Movements

To understand volatility surface movements we used data on S&P 500 call options to construct a neural network

- **Input layer**:
  - Daily asset price return
  - Moneyness (measured by delta)
  - Time to maturity
- **Output layer**:
  - Change in implied volatility

The objective is to minimize the mean squared error between the predicted change in the implied volatility and the actual change.

Neural Network Architecture

- 3 hidden layers
- 20 neurons per layer
- Observations from 2014-2019
- Randomly sampled 100 options per day
- 125700 options in total
- 60% for training set
- 20% for validation set
- 20% for test set
- Z-score scaling

Neural Network Architecture

```
# Create ML Model
#
# Sequential function allows you to define your Neural Network in sequential
# order. Within Sequential, use Dense function to define number of nodes,
# activation function and other related parameters. For more information
# regrading to activation function, please refer to
#
# https://keras.io/activations/
#
model = keras.models.Sequential([Dense(20,activation = "sigmoid",
                                       input_shape = (3,))
                                 ,Dense(20,activation = "sigmoid")
                                 ,Dense(20,activation = "sigmoid")
                                 ,Dense(1)])
# Model summary function shows what you created in the model
model.summary()
# Complie function allows you to choose your measure of loss and optimzer
# For other optimizer, please refer to https://keras.io/optimizers/
model.compile(loss = "mse",optimizer = "Adam")
```

Benchmark Model

- In constructing a machine learning model, it is always useful to have a simpler model as a benchmark. In this case, we use the following model

$$\text{Expected Change in Implied Volatility} = R\frac{a + b\delta + c\delta^2}{\sqrt{T}} \tag{2}$$

where R is the return on the asset (= change in price divided by initial price), T is the option's time to maturity, δ is the option's moneyness (measured as delta) and a, b, and c are constants.

- This model was suggested by Hull and White (2017) and is quite popular with practitioners.
- The a, b, and c can be estimated by regressing implied volatility changes against $R/\sqrt{T}$, $R\delta/\sqrt{T}$, and $R\delta^2/\sqrt{T}$.

Benchmark Model

Building variables for linear regression

```
R     = raw['SPX Return']
RAD_T = np.sqrt(raw['Time to Maturity in Year'])
DELTA = raw['Delta']
# construct the 3 variables for regression
raw['x1'] = R / RAD_T
raw['x2'] = (R / RAD_T) * DELTA
raw['x3'] = (R / RAD_T) * DELTA * DELTA

# Put the X and Y variable in data frame for regression
y = raw['Implied Volatility Change']
X = raw[['x1', 'x2', 'x3','SPX Return','Time to Maturity in Year','Delta']]
```

Benchmark Model

Create training, test and validation set

```
# Divide data into training set and test set(note that random seed is set)
X_train,X_test,y_train,y_test=train_test_split(  X
                                               , y
                                               , test_size    = 0.2
                                               , random_state = 100)

# Divide training set into training and validation set
X_train,X_val,y_train,y_val=train_test_split(  X_train
                                             , y_train
                                             , test_size    = 0.25
                                             , random_state = 100)
```

Benchmark Model

Run Regression

```
# Run the regression on the training data
lr = LinearRegression(fit_intercept=False)
lr.fit(X_scaled_train[:,:3], y_train)

# Get the prediction
y_pred = lr.predict(X_scaled_test[:,:3])

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

print('Test loss (MSE):', mse)
```

Check-Point Deep Learning Models

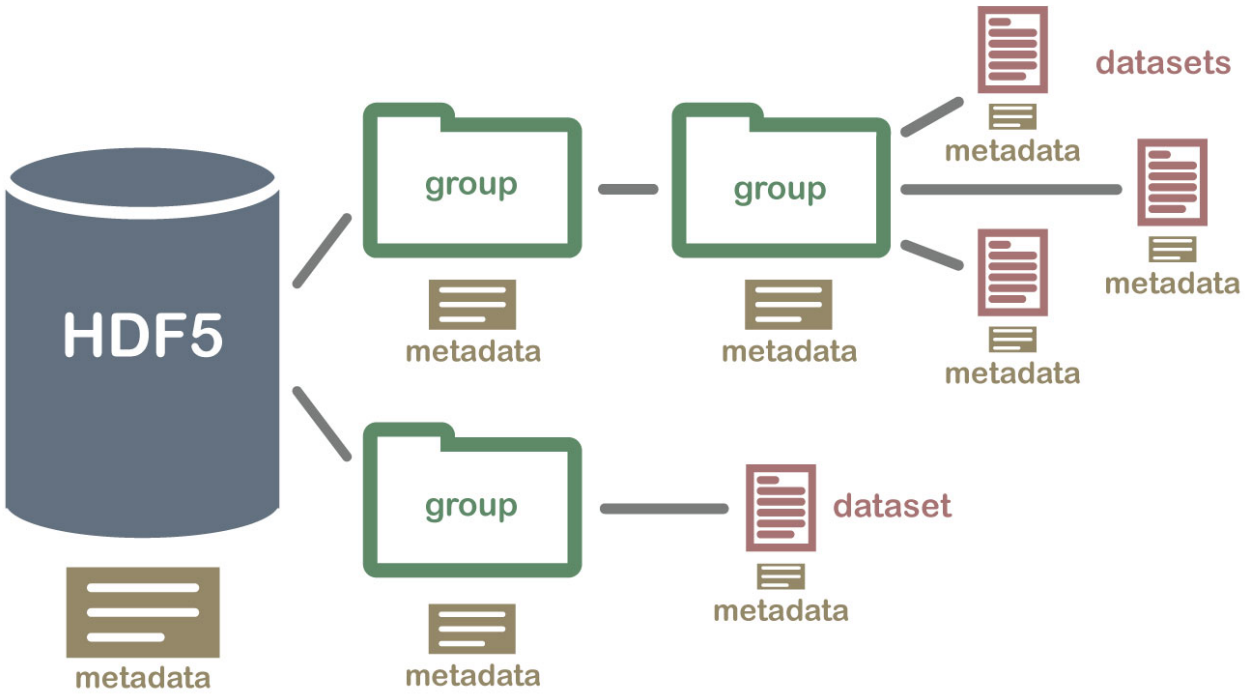
- Deep learning models can take hours, days or even weeks to train.
- If the run is stopped unexpectedly, you can lose a lot of work.
- You can check-point your deep learning models during training in Python using the Keras library.
- A snapshot of the state of the system is taken in case of system failure. If there is a problem, not all is lost.
- The checkpoint may be used directly, or used as the starting point for a new run, picking up where it left off.

Check-Point Deep Learning Models

- The ModelCheckpoint callback class allows you to define where to checkpoint the model weights, how the file should named and under what circumstances to make a checkpoint of the model.
- The API allows you to specify which metric to monitor, such as loss or accuracy on the training or validation dataset. You can specify whether to look for an improvement in maximizing or minimizing the score. Finally, the filename that you use to store the weights can include variables like the epoch number or metric.
- The ModelCheckpoint can then be passed to the training process when calling the fit() function on the model.
- Note, you may need to install the h5py library to output network weights in HDF5 format.

Check-Point Deep Learning Models

- The Hierarchical Data Format version 5 (HDF5), is an open source file format that supports large, complex, heterogeneous data.
- HDF5 uses a "file directory" like structure that allows you to organize data within the file in many different structured ways, as you might do with files on your computer.



Check-Point Deep Learning Models

- Checkpointing is setup to save the network weights only when there is an improvement in classification accuracy on the validation dataset (monitor='val_accuracy' and mode='max').
- The weights are stored in a file 'implied_vol_model_vFinal.h5.h5'

```
# Checkpoint function is used here to periodically save a copy of the model.
# Currently it is set to save the best performing model
checkpoint_cb = keras.callbacks.ModelCheckpoint("implied_vol_model_vFinal.h5"
                                                , monitor='val_accuracy'
                                                , verbose=1
                                                , save_best_only=True
                                                , mode='max')

# Early stopping allows you to stop your training early if no improvement is
# shown after certain period. Currently it is set at if no improvement
# occured in 1000 epochs, at the stop the model will also revert back to
# the best weight.
early_stopping_cb = keras.callbacks.EarlyStopping(patience = 1000
```

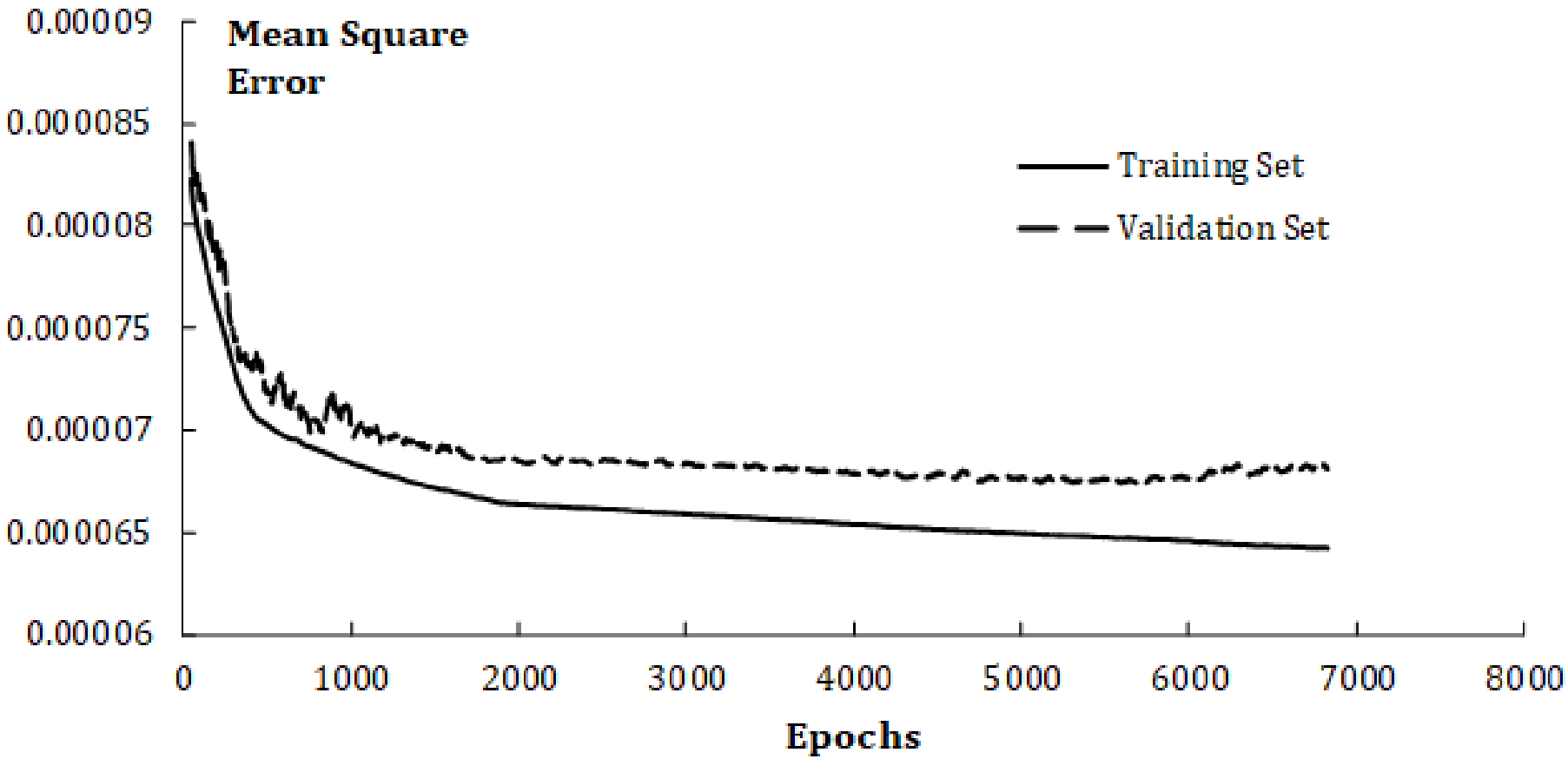
Train the Model

```
# The fit function allows you to train a NN model. Here we have training data,
# number of epochs,batch size, validation data, and callbacks as input
# Callback is an optional parameters that allow you to enable tricks for
# training such as early stopping and checkpoint

# Remarks: Altough we put 50000 epochs here, the model will stop its training
# once our early stopping criterion is triggered

history=model.fit(  X_scaled_train[:,3:6]
                  , y_train
                  , epochs=10
                  , batch_size = 128
                  , verbose = 1
                  , validation_data=(X_scaled_vals[:,3:6],y_val)
                  , callbacks=[checkpoint_cb, early_stopping_cb])
```

Train the Model

```
# Load the best model you saved and calcuate MSE for testing set

model = keras.models.load_model("implied_vol_model_vFinal.h5")
mse_test = model.evaluate(X_scaled_test[:,3:6],y_test,verbose=0)

print('Test Loss(MSE):', mse_test)

# Calculate Gain Ratio

gain = 1 - mse_test/mse

print('Gain Ratio:', gain)
```

Mean Squared Error (Training stopped after 5,826 epochs)



Sensitivities

Delta

- The delta of a portfolio is the sensitivity of the portfolio to the underlying asset price
- A delta-neutral portfolio is not sensitive to small changes in the underlying asset price
- If

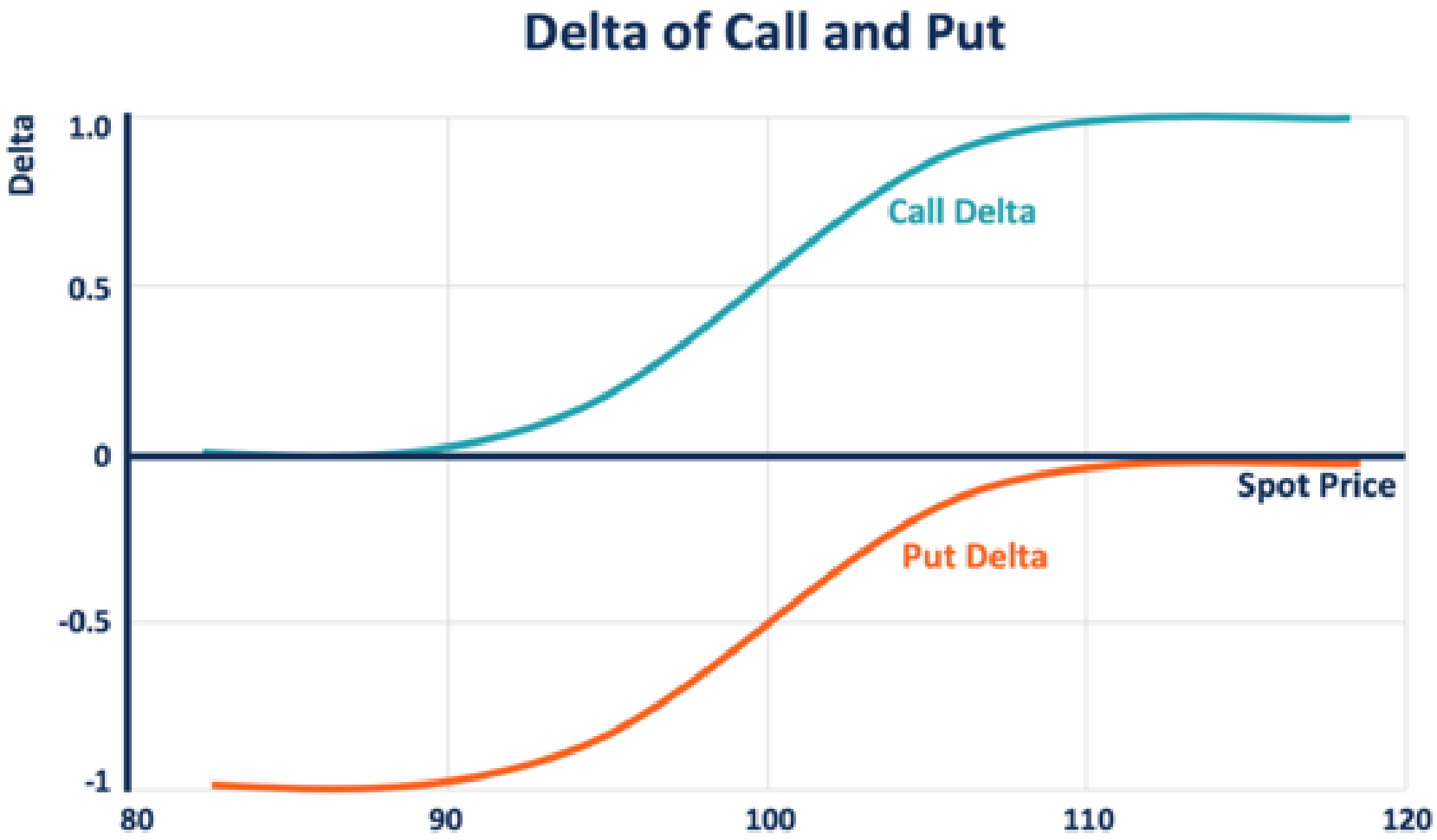
$$d_1 = \frac{\ln(S/K) + (r - q + \sigma^2/2)T}{\sigma\sqrt{T}}$$

then

$$\Delta_C = e^{-qT} N(d_1)$$
$$\Delta_P = e^{-qT} (N(d_1) - 1)$$

where T is time to maturity, r is risk-free rate, and q is dividend yield

Delta (continued)



Delta continued

- Traders can make themselves delta-neutral by trading the underlying asset
- They also use delta as a measure of moneyness for call and put options
- Call is at-the-money when delta =0.5
- Call is in-the-money when delta ¿ 0.5
- Call is out-of-the-money when delta ¡ 0.5
- Put is at-the-money when delta = -0.5
- Put is in-the-money when delta ¡ -0.5
- Put is out-of-the-money when delta ¿ -0.5