chapter-2-1

April 22, 2022

Run in Google Colab

1 Deep Learning for Option Pricing

[154]:
```
import sys
import math
import time
import numpy             as np
import pandas            as pd
import matplotlib.pyplot as plt

from scipy.stats             import norm
from sklearn.model_selection import train_test_split
```

1.1 Introduction

The problem of this first part of our lessons can be stated simply as follow. Let's say, from a very general point of view, that we have a contingent claim $\mathcal{V}$ that depends on D parameters:

$$\mathcal{V}(\mathbf{x}),\ \mathbf{x} = \{§_\infty, \ldots, §_\mathcal{D}\} \in \mathbb{R}^\mathcal{D}$$

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.

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.

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. Plain vanilla options are therefore not priced using a model. They are simply priced to be consistent with the market.

The volatility surface derived from the Black–Scholes–Merton model is a convenient interpolation tool for doing this.

Exotic options are generally not as actively traded as plain vanilla options and, as a result, a model is required for pricing. A variety of different models are used in practice. 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.

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.

Many different models in the first category involving stochastic volatility and jumps have been proposed. Examples of stochastic volatility models are Hull and White (1987) and Heston (1993). Merton (1976) proposed a model that overlays Black-Scholes-Merton model with jumps. Bates (1996) adds jumps to Heston (1993). Madan et al (1998) propose a variance–gamma model where there are only jumps. More recently, rough volatility models where the process for volatility is non-Markov have been suggested by authors such as Gatheral et al (2018).

The second category of models are referred to as local volatility models. The original onefactor local volatility model was suggested by Dupire (1994) and Derman and Kani (1994). It has been extended by authors such as Ren et al (2007) and Saporito et al (2019). Local volatility models by design satisfy condition B.

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.

Neural networks can be used in conjunction with MCA to provide fast pricing once model parameters have been determined. Consider an exotic option that is valued using Monte Carlo simulation. As a first step, it is necessary to devote computational resources to creating a data set relating model parameters and exotic option parameters to the price. The pricing model can then be replicated with a neural network. Once this has been done, valuation is several orders of magnitude faster than Monte Carlo simulation because it involves nothing more than a forward pass through the neural network.

In this first notebook we will look at a simple example of this approach. For the sake of simplicity we will consider an example based on the Black and Scholes model for the pricing of a simple European option.

In the second part of these lessons we'll see how to train a neural network on market data (volatility surface) using the heston model as the underlying model.

1.2 The Black and Scholes Model

We consider an underlying process $S(t)$ described by the sde

$$dS(t) = a(S,t)dt + b(S,t)dW \tag{1}$$

A scenario is a set of values $\hat{S}^j(t_i)$, $i = 1,\ldots,I$ that are an approximation to the $j-th$ realization,$S^j(t_i)$, of the solution of the sde evaluated at times $0 \leq t_i \leq T$, $i = 1,\ldots,I$. A scenario is also called a trajectory. A trajectory can be visualized as a line in the state-vs-time plane describing the path followed by a realization of the stochastic process (actually by an approzimation to the stochastic process).

The Black and Scholes model assumes a market in which the tradable assets are:

1. A risky asset, whose evolution is driven by a geometric brownian motion

$$dS = \mu S dt + \sigma S dw \Rightarrow S(T) = S(t_0)e^{(\mu - \frac{1}{2}\sigma^2)(T-t_0)+\sigma[w(T)-w(t_0)]} \tag{2}$$

2. The money market account, whose evolution is deterministic

$$dB = Brdt \Rightarrow B(T) = B(t_0)e^{r(T-t_0)} \tag{3}$$

The value for a non-dividend-paying underlying stock in terms of the Black–Scholes parameters is:

$$C(S_t,t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)} \tag{4}$$

$$P(S_t,t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t \tag{5}$$

where

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \tag{6}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{7}$$

For both:

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution
- $T-t$ is the time to maturity (expressed in years)

- S_t is the spot price of the underlying asset
- K is the strike price
- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset

1.2.1 Simpler formulation for vanilla options

Up until now, we described the pricing of instruments with general payoffs and in particular of vanilla options. In the latter case, starting from the expression of the vanilla put option price

$$P_t = \mathbb{E}\left[e^{-r(T-t)}(K - S_T)^+\right]$$

if we extract from the expected value S_t we get

$$P_t = S_t \mathbb{E}\left[\left(e^{-r(T-t)}\frac{K}{S_T} - M_T\right)^+\right]$$

where $M_T = e^{-r(T-t)}S_T/S_t$ is the stochastic process of a martingale with the same volatility of S_t and $M_t = 1$. By comparing this expression with the one from the Black Scholes formula, we notice that we can in general rewrite the price of a vanilla put option as

$$P_(S_t, K, T, \sigma, r) = S_t P(1, e^{-r(T-t)}K/S_T, T, \sigma, 0)$$

From this we can see that we can calculate the price of any vanilla put option by calculating the value of a simpler put option with $r = 0$, $S_t = 1$ and $e^{-r(T-t)}K/S_t$ in place of the strike price. The exact same argument can be made for a vanilla call option. For simplicity, in the rest of this work we will always consider $r = 0$, since we don't lose any generality when the risk-free rate is independent of the asset price.

[155]:
```
#
# NumPy enabled asset or nothing
#
def __npan__ ( Fw, T, sgma, k):
    s           = sgma*np.sqrt(T)
    mask        = np.where( s <  1.e-8, 1, 0)
    MASK        = np.where( s >= 1.e-8, 1, 0)
    s           = np.where(mask, 1., s)
    dm          = ( np.log(k/Fw) -.5*s*s)/s
    an          = norm.cdf(dm)

    m1          = np.logical_and(mask, Fw <= k)
    res         = np.where(m1, 1., 0.)
    res        += np.where(MASK,an,0.0);

    return res
#
```

```
# NumPy enabled cash or nothing
#
def __npcn__ ( Fw, T, sgma, k):
    s           = sgma*np.sqrt(T)
    #print("npcn: s = %f" %s)
    mask        = np.where( s <  1.e-8, 1, 0)
    MASK        = np.where( s >= 1.e-8, 1, 0)
    s           = np.where(mask, 1., s)
    dp          = ( np.log(k/Fw) + .5*s*s)/s
    cn          = norm.cdf(dp)

    m1          = np.logical_and(mask, Fw <= k)
    res         = np.where(m1, 1., 0.)
    res        += np.where( MASK,cn,0.0);

    return res

def np_fw_euro_put(F, T, sgma, k):
    return k*__npcn__( F, T, sgma, k)  - F*__npan__( F, T, sgma, k)

def np_fw_euro_call(F, T, sgma, k):
    return np_fw_euro_put(F, T, sgma, k) + F - k

def np_euro_put(So, r, T, sigma, k):
    Fw = np.exp(r*T)*So
    return np.exp(-r*T)*np_fw_euro_put(Fw, T, sigma, k)

def np_euro_call(So, r, T, sigma, k):
    Fw = np.exp(r*T)*So
    return np.exp(-r*T)*np_fw_euro_call(Fw, T, sigma, k)
```

1.3 Training Data Generation

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 the Latin Hypercube Sampling (LHS) (see below) within the ranges usually found in real market.

1.3.1 Sampling Utilities

Latin Hypercube Sampling (from Wikipedia)

Latin hypercube sampling (LHS) is a statistical method for generating a near-random sample of parameter values from a multidimensional distribution. The sampling method is often used to construct computer experiments or for Monte Carlo integration. In the context of statistical sampling, a square grid containing sample positions is a Latin square if (and only if) there is only one sample in each row and each column. A Latin hypercube is the generalisation of this concept to an arbitrary number of dimensions, whereby each sample is the only one in each axis-aligned

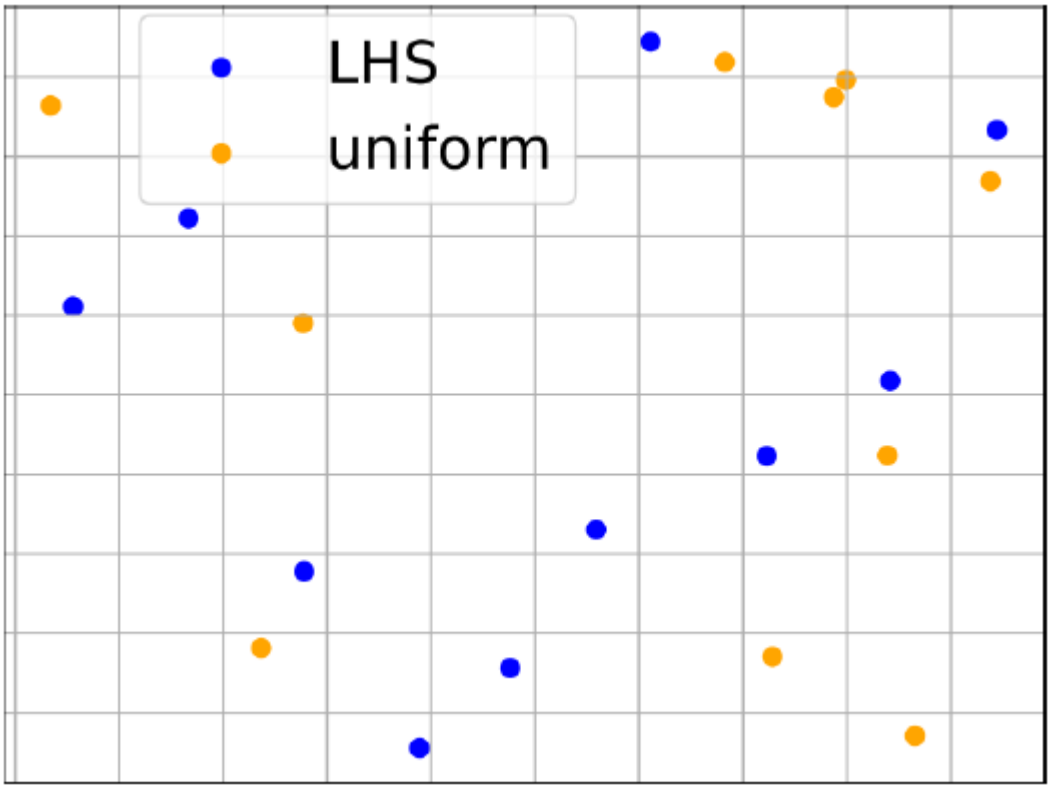
hyperplane containing it.

When sampling a function of N variables, the range of each variable is divided into M equally probable intervals. M sample points are then placed to satisfy the Latin hypercube requirements; this forces the number of divisions, M, to be equal for each variable. This sampling scheme does not require more samples for more dimensions (variables); this independence is one of the main advantages of this sampling scheme. Another advantage is that random samples can be taken one at a time, remembering which samples were taken so far.

In two dimensions the difference between random sampling, Latin hypercube sampling, and orthogonal sampling can be explained as follows:

- In random sampling new sample points are generated without taking into account the previously generated sample points. One does not necessarily need to know beforehand how many sample points are needed.
- In Latin hypercube sampling one must first decide how many sample points to use and for each sample point remember in which row and column the sample point was taken. Such configuration is similar to having N rooks on a chess board without threatening each other.
- In orthogonal sampling, the sample space is divided into equally probable subspaces. All sample points are then chosen simultaneously making sure that the total set of sample points is a Latin hypercube sample and that each subspace is sampled with the same density.

Thus, orthogonal sampling ensures that the set of random numbers is a very good representative of the real variability, LHS ensures that the set of random numbers is representative of the real variability whereas traditional random sampling (sometimes called brute force) is just a set of random numbers without any guarantees.



[156]:
```
from smt.sampling_methods import LHS
```

[157]:
```
def lhs_sampling(rand, NUM, bounds=None):

    mInt = (1 << 15)
    MInt = (1 << 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
```

1.3.2 Sample Generator

We assume $r = q = 0, S_0 = 1$ and create a data set of 1000000 observations by randomly sampling from uniform distributions for the other five inputs to the Black-Scholes-Merton formula. The lower and upper bounds of the uniform distributions are as indicated in the `bounds` dictionary. For each set of parameters sampled, we calculate the Black-Scholes-Merton price using equations (4) and (5).

[201]:
```
# 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 (simply defined as␣
 ↪'Strike')
# and the asset volatility sigma.
#
# 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
#
```

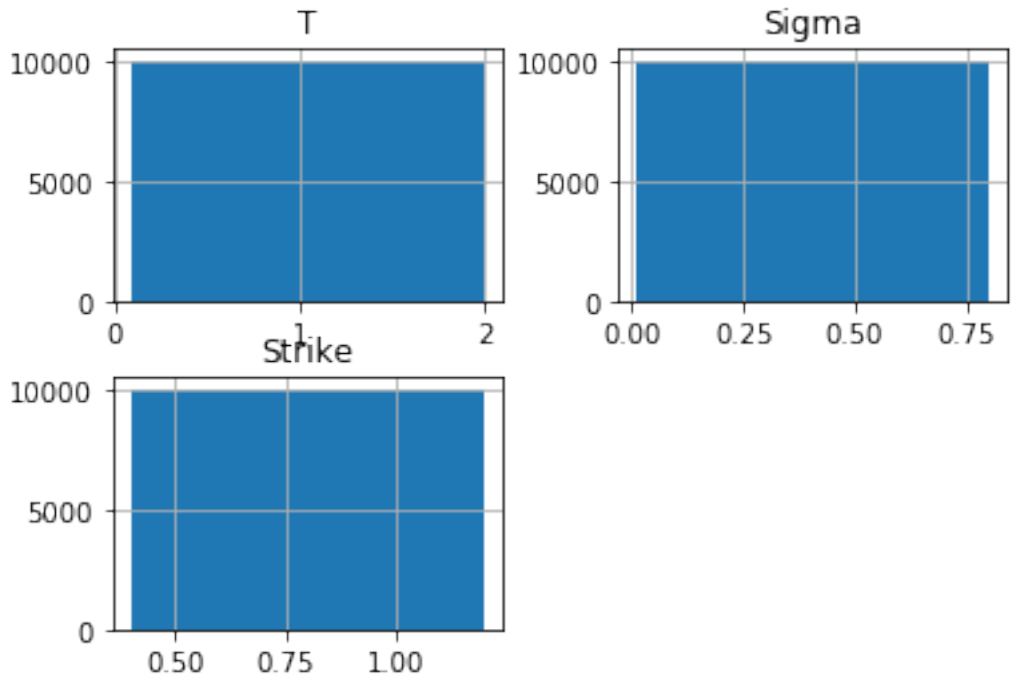
```
# Random number generator
#
seed = 42
rand = np.random.RandomState(seed)
#
# Latin Hypercube Sampling
#
xDF = lhs_sampling(rand, NUM, bounds=bounds)
xDF.head()
```

[201]:

	T	Sigma	Strike
0	1.994528	0.030157	1.006996
1	1.413241	0.423980	1.055836
2	0.891927	0.673083	0.677284
3	0.903523	0.308853	0.623468
4	1.298069	0.134919	0.743884

[202]:
```
xDF.hist()
```

[202]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000001DD418D0BC8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001DD41869888>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001DD397B0408>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001DD397CA8C8>]],
      dtype=object)
```



[203]:
```
def gen(NUM, lhs):

    '''
    A NumPy optimized data generator
    '''
    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
```

[204]:
```
__tStart = time.perf_counter()
df =  gen(NUM, xDF)
__tEnd = time.perf_counter()
print("@ %-34s: elapsed %.4f sec" %("GEN", __tEnd - __tStart) )

df.head()
```

```
@ NP pricing                         : elapsed 0.0484 sec
@ GEN                                : elapsed 0.0501 sec
```

[204]:

	T	Sigma	Strike	Price
0	1.994528	0.030157	1.006996	0.013779
1	1.413241	0.423980	1.055836	0.177768
2	0.891927	0.673083	0.677284	0.406779
3	0.903523	0.308853	0.623468	0.381761
4	1.298069	0.134919	0.743884	0.257487

[205]:
```
X_train, X_test = train_test_split(df, test_size=0.33, random_state=42)
print(len(X_train), len(X_test))
X_train.head()
```

```
67000 33000
```

[205]:

	T	Sigma	Strike	Price
59428	1.196045	0.739988	0.925964	0.340934
34957	1.703252	0.036864	1.071172	0.001712
4264	1.762209	0.204376	0.707052	0.303768
53791	1.717263	0.545387	0.556180	0.503275
82114	0.548441	0.620800	1.118916	0.138778

[206]:
```
X_test.head()
```

[206]:

	T	Sigma	Strike	Price
75721	0.117191	0.771398	1.086372	0.071709
80184	1.026228	0.200512	0.801388	0.211295
19864	0.342879	0.041699	0.713412	0.286588
76699	0.226135	0.336637	1.169668	0.014981
92991	0.936758	0.730681	1.048660	0.259367

To make the illustration more interesting, we then add a random error to each calculated price. The random error is normally distributed with a mean of zero and a standard deviation of $\epsilon \cdot (P_{max} - P_{min})$ where ϵ is a scale parameter defined by the user.

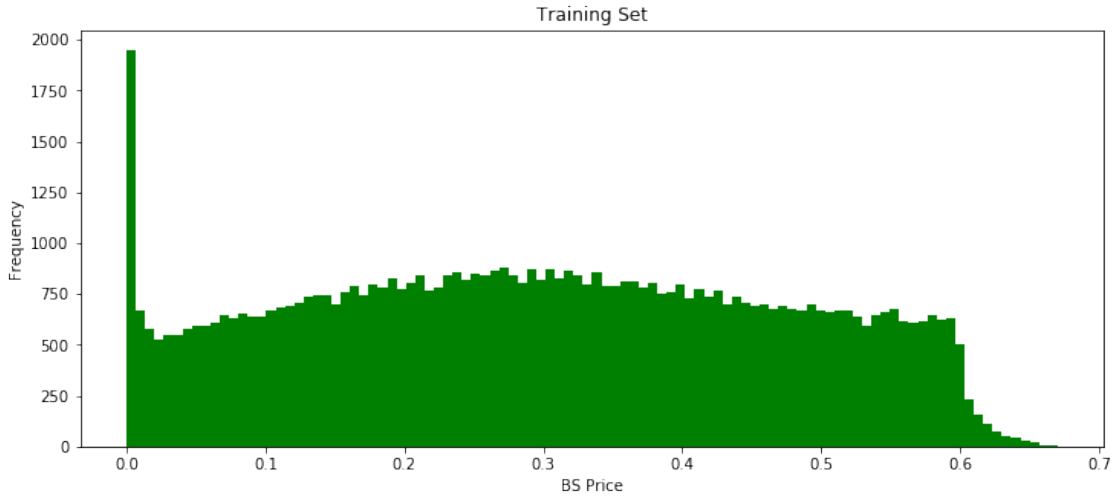
[207]:
```
def add_noise(rand, Xv, eps):
    X  = Xv.copy()
    xl = np.min(X["Price"])
    xh = np.max(X["Price"])

    xi = rand.normal( loc = 0.0, scale = eps*(xh-xl), size=X.shape[0])
    X["Price"] += xi
    return X
```

[208]:
```
EPS = 0.0
if EPS > 0.0:
    X_train = add_noise(rand, X_train, EPS)
```

[209]:
```
y_train = X_train['Price']
y_test  = X_test['Price']
```

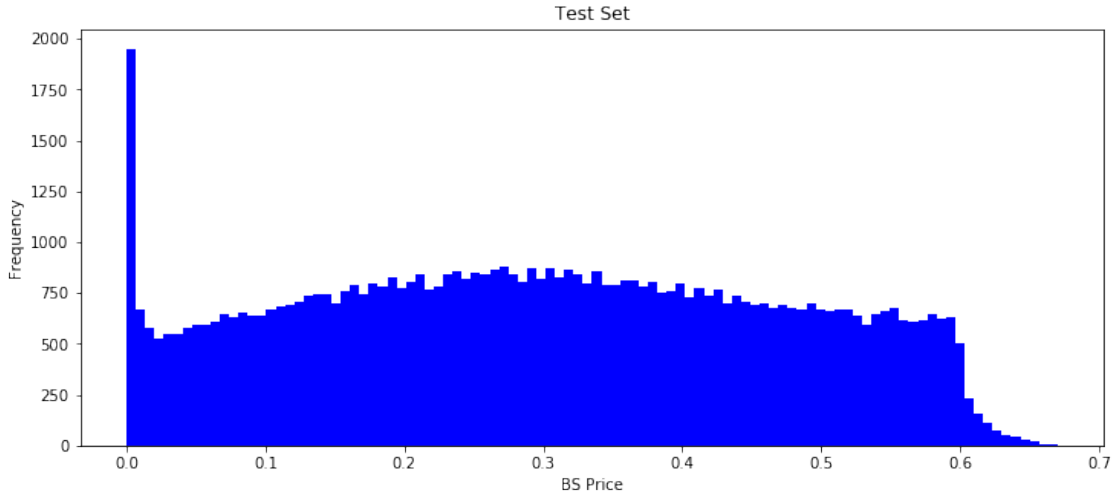
[210]:
```
ax=y_train.plot(kind='hist', bins=100, grid=False, figsize=(12,5),␣
 ↪color='green', title = 'Training Set')
ax.set_xlabel('BS Price')
```

[210]:
```
Text(0.5, 0, 'BS Price')
```



[211]:
```
ax=y_train.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='blue',␣
↪title = 'Test Set')
ax.set_xlabel('BS Price')
```

[211]: Text(0.5, 0, 'BS Price')



[212]:
```
X_train = X_train.drop(['Price'], axis=1)
X_test  = X_test.drop(['Price'], axis=1)
X_train.head()
```

[212]:

	T	Sigma	Strike
59428	1.196045	0.739988	0.925964

```
34957   1.703252   0.036864   1.071172
4264    1.762209   0.204376   0.707052
53791   1.717263   0.545387   0.556180
82114   0.548441   0.620800   1.118916
```

1.3.3 Scaling Data

[213]:
```
from sklearn.preprocessing import StandardScaler

scaling = True

if scaling:
    # Scale the train set
    scaler = StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)

    # Scale the test set
    scaler = StandardScaler().fit(X_test)
    X_test = scaler.transform(X_test)

    # Transform into pd DataFrame
    X_train = pd.DataFrame(X_train)
    X_test = pd.DataFrame(X_test)
```

1.4 Create the Model

1.4.1 Neural Network Architecture

We use mean absolute error (mae) as the cost function. The neural network has two hidden layers and a decreasing number of neurons per layer. The relu activation function is used.

[214]:
```
from keras.models import Sequential
from keras.layers import Dense
```

The input shape

What flows between layers are tensors. Tensors can be seen as matrices, with shapes. Shapes are consequences of the model's configuration. Shapes are tuples representing how many elements an array or tensor has in each dimension.

> Ex: a shape (30,4,10) means an array or tensor with 3 dimensions, containing 30 elements in the first dimension, 4 in the second and 10 in the third, totaling 30410 = 1200 elements or numbers.

In Keras, the input layer itself is not a layer, but a tensor. It's the starting tensor you send to the first hidden layer. This tensor must have the same shape as your training data.

> Ex: if you have 30 images of 50x50 pixels in RGB (3 channels), the shape of your input data is (30,50,50,3). Then your input layer tensor, must have this shape.

Since the input shape is the only one you need to define, Keras will demand it in the first layer.

[215]:
```
def model_builder( inputShape = (1,)):

    # Initialize the constructor
    model = Sequential()

    # Start from the first hidden layer, since the input is not actually a layer␣
 ↪
    # But inform the shape of the input, with inputShape elements.
    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))
    # End model construction

    # 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
```

[216]:
```
model = model_builder( inputShape = (X_train.shape[1],))
```

```
model.output_shape: (None, 1)
Model.summary
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_28 (Dense)             (None, 200)               800
_________________________________________________________________
dense_29 (Dense)             (None, 200)               40200
_________________________________________________________________
dense_30 (Dense)             (None, 200)               40200
_________________________________________________________________
dense_31 (Dense)             (None, 1)                 201
=================================================================
```

```
Total params: 81,401
Trainable params: 81,401
Non-trainable params: 0
_________________________________________________________________
Model.config
```

1.5 Train the Model

[217]:
```
def show_scattered( y, t, TAG, ax = None):
    #x      = model.predict(X)
    #y      = np.ravel(x)
    xMin = min(t)
    xMax = max(t)
    v    = np.arange(xMin, xMax, (xMax-xMin)/100.)

    diff   = np.fabs(y - t)
    print("@ %-24s: E[y-t]: %.6f Std(y-t): %.6f" %( TAG, np.mean(diff), np.
 ↪std(diff)))
    if ax == None: return

    ax.plot( y, t, ".")
    ax.plot( v, v, color="red")
    ax.set_title("%s mae=%8.4f, std=%8.4f" %(TAG, np.mean(diff), np.std(diff)))
    ax.set_xlabel("predicted")
    ax.set_ylabel("target")
```

[218]:
```
def display_nn_results( model, X_train, X_test, t_train, t_test, resFile=None):

    fig, ax = plt.subplots(1,2, figsize=(12,6))
    fig.suptitle("Scattered plots")
    #
    # The numpy module ravel of NumPy provides a function, called numpy.ravel,␣
 ↪which is used to change
    # a 2-dimensional array or a multi-dimensional array into a contiguous␣
 ↪flattened array. The returned
    # array has the same data type as the source array or input array.
    #
    y_train  = np.ravel(model.predict(X_train))
    show_scattered( y_train, t_train, "InSample", ax = ax[0])
    diff   = np.fabs(y_train - t_train)
    delta  = y_train - t_train
    RES    = pd.DataFrame({"predicted": y_train, "target": t_train, "err": diff,␣
 ↪"delta": delta})
    h1     = RES['delta']
    RES.to_csv("res_in_sample.csv", sep=',', float_format="%.6f", index=True)
    print("@")
```

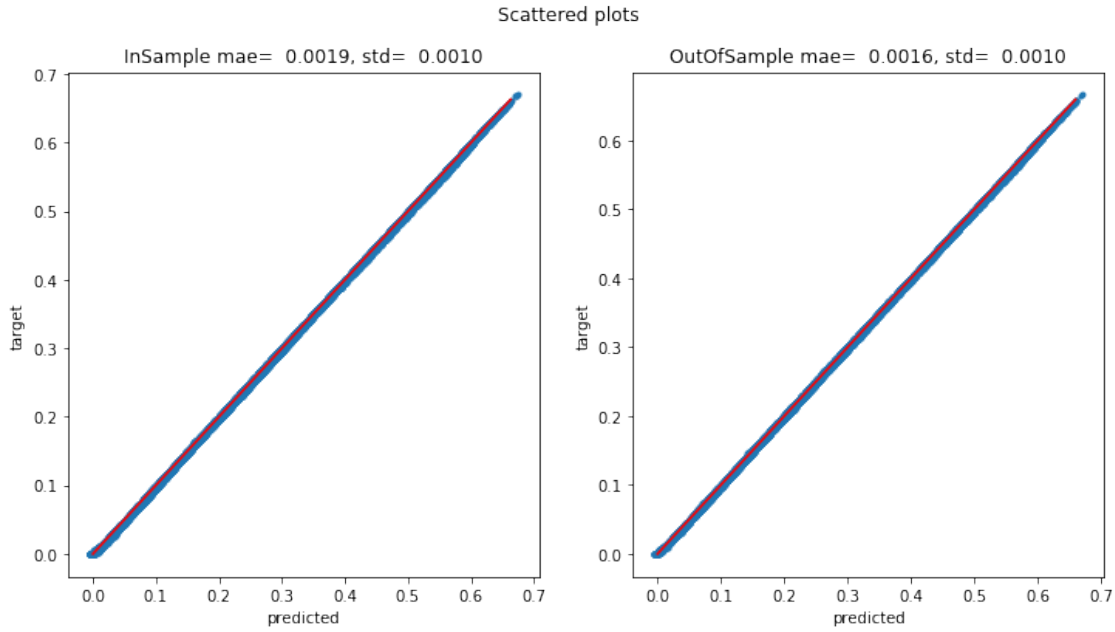
```
    y_test  = np.ravel(model.predict(X_test))
    show_scattered( y_test , t_test, "OutOfSample", ax= ax[1])
    diff    = np.fabs(y_test-t_test)
    delta   = y_test - t_test
    RES     = pd.DataFrame({"predicted": y_test, "target": t_test, "err": diff,␣
 ↪"delta": delta})
    h2      = RES['delta']
    RES.to_csv("res_ou_sample.csv", sep=',', float_format="%.6f", index=True)
    print("@")

    if resFile != None:
        plt.savefig(resFile, format="png")
        print("@ %-12s: results saved to '%s' "%("Info", resFile))
    plt.show()


    score = model.evaluate(X_test, t_test, verbose=1)
    print('Score:'); print(score)

    h1.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='blue',␣
 ↪title = 'err distribution')
    h2.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='red')
```

[219]:
```
frames = [X_train, X_test]
X = pd.concat(frames)

frames = [y_train, y_test]
Y = pd.concat(frames)
```

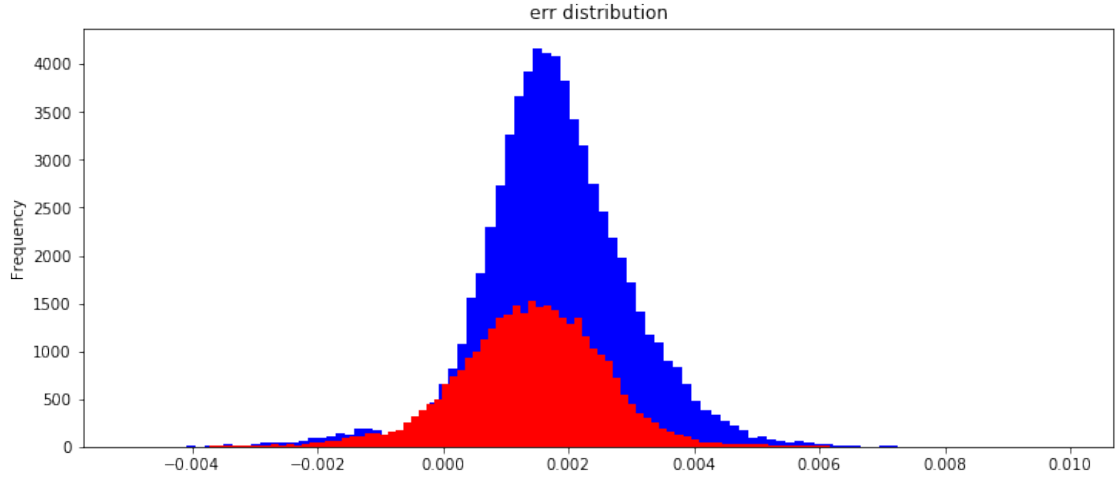
[1]:
```
# Fit the model
history = model.fit(X, Y, validation_split=0.33, epochs=50, verbose=1)
```

[221]:
```
import warnings
warnings.simplefilter('ignore')

display_nn_results(model, X_train, X_test, y_train, y_test, resFile='training.
 ↪png')
```

```
@ InSample               : E[y-t]: 0.001893 Std(y-t): 0.001032
@
@ OutOfSample            : E[y-t]: 0.001559 Std(y-t): 0.000966
@
@ Info        : results saved to 'training.png'
```




```
1032/1032 [==============================] - 1s 1ms/step - loss: 3.3641e-06 -
mae: 0.0016
Score:
[3.364141548445332e-06, 0.0015591253759339452]
```



1.5.1 Access Model Training History in Keras

Keras provides the capability to register callbacks when training a deep learning model. One of the default callbacks that is registered when training all deep learning models is the **History callback**. It records training metrics for each epoch. This includes the loss and the accuracy (for

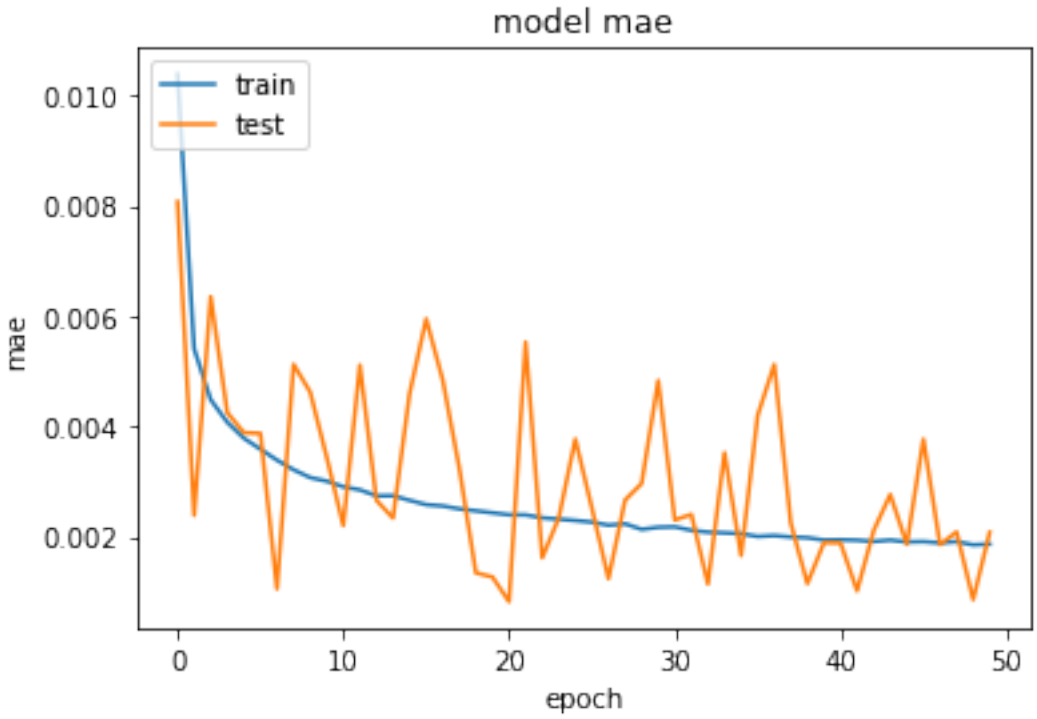
classification problems) as well as the loss and accuracy for the validation dataset, if one is set.

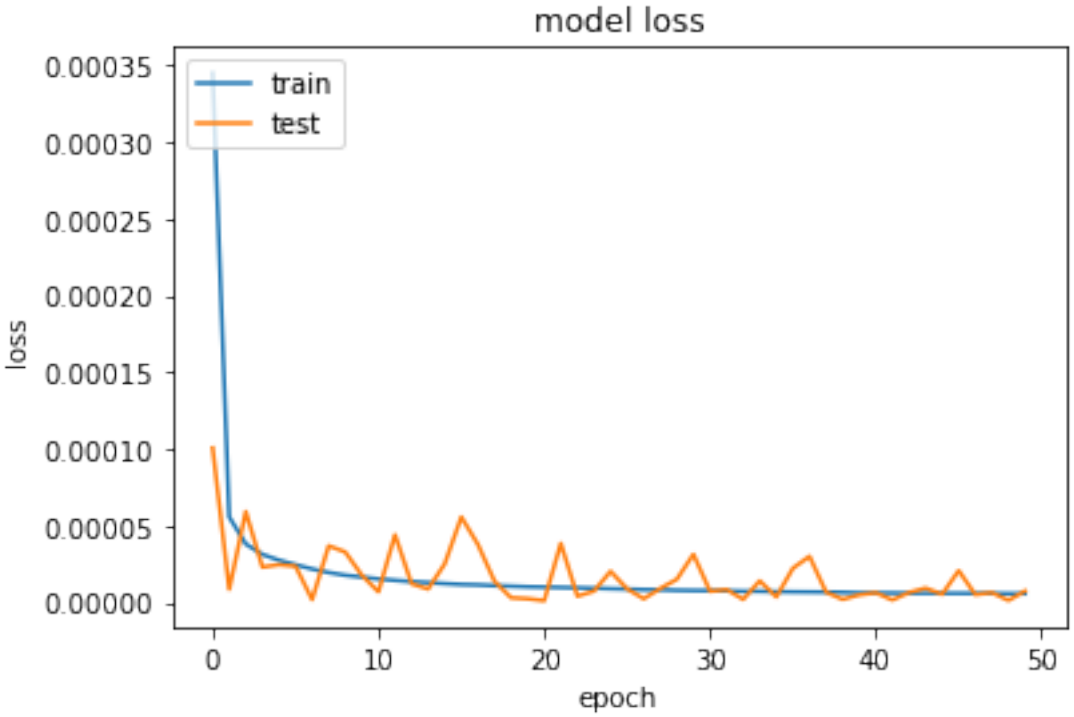
The history object is returned from calls to the fit() function used to train the model. Metrics are stored in a dictionary in the history member of the object returned.

For example, you can list the metrics collected in a history object using the following snippet of code after a model is trained:

[153]:
```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'mae', 'val_loss', 'val_mae'])
```





1.6 Saving the Model

[ ]:
```
TAG         = 'bs-0100000-050-T'
mdlDir      = "model_%s.krs" %(TAG)


model.save(mdlDir)
print("@ %-24s: model saved to '%s'" %("Info", mdlDir))
```

[ ]:
```

```