Stevens Institute Of Technology

FINANCIAL ENGINEERING

SABR volatility surface fitting (model calibration) using Artificial Neural Network

---

Author:

Fang Yih Chan

Supervisor:

Prof. Dan Pirjol

December 16, 2021

Abstract

Recent progress in the field of artificial intelligence (AI) or Machine Learning (ML) or Deep Learning (DL) has drawn attention to apply these techniques to solve sophisticated mathematical finance problem. This work proposes the use of an Artificial Neural Network (ANN) to implement the calibration of the stochastic volatility model: SABR model to Swaption volatility surfaces or market quotes. SABR has become benchmark in fixed-income or interest-rate derivative market due to its readily closed-form Hagan's approximation formula for implied volatility. An efficient and fast reliable calibration is central consideration for the usability of any pricing model. ANN enabled this rapid performance by becoming the direct online pricer or Universal Function Approximator replacing the convention slow pricer (thus remove bottleneck) by learning from input parameters-asset price data pairs that were computed offline separately. This stage of training the ANN-solver and adjusting weights is called the forward-pass. This work proposes generating this synthetic data of asset price (specifically implied volatilities) using different model parameters via Finite Difference Method (FDM) and Dekker's root-finding method. The calibration process is complete when it is followed by an optimization process to match the price to market data to determine the optimal model parameters nown as the backward-pass. Instead of global optimization method that many other authors have used, this work applied the optimization-free Inverse Map approach which utilize an inverse ANN and numerical calculation involving gradients of forward-pass ANN with respect to inputs. The neural network has been implemented using the open source library Tensorflow Keras in Python. After testing this approach by calibrating to a few market volatility surfaces, the results confirm that this machine learning technique can be employed to calibrate parameters of high-dimensional stochastic volatility models like SABR efficiently and accurately.

1. Introduction

Pricing or valuation of financial derivative contracts and assets is a common daily task in financial institution for hedging or risk management purposes. Features and nonlinear behaviour observed in financial market are captured via mathematical models. The first arbitrage free model introduced by Black and Scholes (1973) and Merton (1973) enabled the pricing of European call/put options with a closed-form solution. This simplified model comes with a few assumptions such as no dividend payout or transaction cost, constant risk free rate, underlying returns of log-normal distribution and constant volatility. However, more advanced models are often multi-dimensional and do not have closed-form solutions. Examples of such complex models include Local stochastic volatility (Dupire) or stochastic volatility model (Heston or SABR - Stochastic Alpha Beta Rho). Thus the pricing of such derivative requires

solving partial differential equation (PDE) via numerical methods and computation-intensive effort like finite difference, Fourier methods and Monte-Carlo simulation. The computation time of these approaches are long, unattractive and often do not scale linearly with accuracy.

Option market prices are often quoted conveniently by traders in terms of implied volatility as it provides intuition regarding market dynamics. The Implied volatility (IV) surface generated when plotted against moneyness and time maturity (or volatility cube if include tenor) generally do not exhibit a flat surface but with skew or smile/smirk. One such example could be found in the work of Dimitroff and Kock (2011) on volatility cube and ATM (At-The-Money) surface. In financial derivative pricing, the inverse problem in which model parameters inferred and fitted to market data is called calibration. This is essential and needs to be done in an efficient way as calibrated parameters are used in a model for actual pricing. In the presence of closed form expression, the calibration problem could be solved through various root-finding methods. Otherwise, it has to be solved iteratively. A slow forward pricing process like Monte Carlo would create extra bottleneck and thus make efficient calibration prohibitively expensive.

The recent progress in Artificial intelligence (AI) among academic or practitioners as well as the advancement in hardware computing (CPU/GPU) and software (Tensorflow/Keras in Python or R packages), has drawn popularity to the of use machine learning (ML) technique in solving derivative pricing and calibration problems which is purpose of this work. Its success attributed to the capability of multi-layered Artificial Neural Networks (ANN) or Deep Learning (DL) to closely micmic a function by just learning through input-output pairs. The calibration problem is usually solved in two steps in which the slow model pricer is replaced with a trained ANN acting as a direct price approximator and followed by a standard optimization process.

2. Literature review & motivation

There is a substantial body of literature treating pricing or calibration models parameters of financial mathematical models exists, with many implemented with the aid of neural networks. A comprehensive consolidation of related papers dated as early as two decades ago with cross-sectional information like model involved, neural network input/output features, data methods, performance measurements can be found in Ruf and Wang (2021).

Liu et al. (2019a) successfully computed the implied volatilities based on Black-Scholes and Heston model (option pricing via COS method) in a fast and efficient way using ANN with training done off-line and root-finding via Brent's method. In separate occasion, Liu et al.

(2019b) extended the work to calibrate model parameters of Heston & Bates model using market implied volatilities. Their CaNN (Calibration Neural Network) merges separate ANNs into a single forward-pass ANN and consist of training, prediction & calibration. Stochastic Gradient Descent (SGD) is used for training in forward-pass and Differential Evolution (DE) - a derivative-free global optimizer is used for calibration in backward-pass to address problem of multiple local minima.

Hernandez (2016) demonstrated fast calibration of Hull-White model to 156 swaptions using ANN by offloading the training process offline separately. He generated random normal distributed vectors based on covariance of rescaled log-parameters, PCA-processed yield curve and used them to apply inverse transform to obtain the parameters. Dimitroff et al. (2018) showcased the calibration of Heston model parameters onto equity derivatives prices using convolution neural network (CNN). The implied volatilities were stacked together with moneyness, forward, strike, and maturity to form 3-dimensional input tensor and fed into neural network with specific kernels. The CNN find relevant features of volatility surface and utilized Scaled Exponential Linear Units (SELUs) and sigmoid activation function.

McGhee (2018) utilized relative simple single-layer ANN with sigmoid activation function and Adam (Adaptive Moment Estimation) optimizer to predict SABR stochastic volatility model under β=1. The computation was fast and maintained high degree of accuracy. The ANN trained using 250,000 input-output pairs derived from various combinations of pre-defined volatility of volatility factor, tenor and maturity, strikes on 3 separate methods: Hagan's SABR approximation method, SABR integration and Finite Difference Scheme. The closed-form SABR approximation of implied volatility using forward price was developed by Hagaan et al. (2002).

Horvath et. al (2019) presented a fast neural network based calibration of rough volatility models: Bergomi models for the implied volatility surface using two steps (price-parameter mapping/training) and optimization. The optimization algorithms used are Levenberg-Marquardt (LM), Broyden-Fletcher-Goldfarb-Shanno (BFGS), Limited-memory BFGS (L-BFGS-B) and Sequential-Least Squares Programming (SLSQP). Meanwhile, Bayer and Stemper (2018) illustrated 4-layers ReLU Fully Connected Neural Network (FCNN) to calibrate Heston and Bergomi rough stochastic volatility model to SPX option implied volatilities with high accuracy and speed. The calibration was based on LM optimization which involves finding Jacobian of residuals.

Despite the vast body of literature on pricing and calibration using ANN, most are related to Black-Scholes, Heston/Rough volatility or others like GARCH based model. Since not many similar works were found under SABR model, this work serves as an addition to the literature of SABR model calibration or volatility surface fitting.

3. Theoretical Background

3.1 Swaption & Black's equation

This section briefly introduce the mathematical background of Futures Option and Swaption pricing and relation to Black's equation. The mathematical framework by Harrison and Pliska (1981) of a unique no-arbitrage pricing associated with any attainable contingent claim V at any time t with an equivalent martingale measure Q (with existence of Radon-Nikodym derivative) is given by:

$$V(t) = E^Q\left[B(t)\frac{V(T)}{B(T)}|\mathcal{F}_t\right] = E^Q[D(t,T)V(T)|\mathcal{F}_t] \quad (3.1a)$$

where it involves expectation (E) under Q numeraire (bank-account numeraire), money-market account process B ($B(t) = B(0)\exp(\int_0^t r(s)\,ds)$), filtration process $\mathcal{F}_t$ and continuous compounded discount factor: $D(t,T) = \frac{B(t)}{B(T)} = \exp(-\int_t^T r(s)\,ds)$. The use of other possible numeraires under their relative measure is also possible as described below (Brigo, D. and Mercuio, 2006).

Using zero-coupon bond with maturity T as numeraire, i.e. $P(t,T) = \exp(-\int_t^T r(s)\,ds)$ denoting present value of a unit notional paid at T and the property $P(T,T) = 1$, the valuation under T-forward measure is given by:

$$V(t) = E^Q\left[P(t,T)\frac{V(T)}{P(T,T)}|\mathcal{F}_t\right] = P(t,T).E^Q[V(T)|\mathcal{F}_t] \quad (3.1b)$$

Consider for example a European Futures option on asset A with exercise date t_{ex} and settlement date t_{set} , holder pays K and receives asset A when exercise the option. With forward price of asset $F(t)$ and discount factor $D(t)$, value of call option is

$$V(0) = D(t_{set})E^Q[(F(t_{ex}) - K)^+|\mathcal{F}_0] = D(t_{set})(F_0N(d_+) - KN(d_-));\ d_\pm = \frac{\log(\frac{F}{K}) \pm \frac{1}{2}{\sigma_B}^2 t_{ex}}{\sigma_B\sqrt{t_{ex}}} \quad (3.1c)$$

with $F_0 = F(0)$ and forward price $F(t)$ is martingale under forward measure.

Meanwhile, using a portfolio of zero coupon bonds with maturity $T_1, \ldots, T_n$ as numeraire, *i.e.* $C_{T_0,T_n}(t) = \sum_{i=1}^n \tau_i P(t,T_i)$ where $\tau_i = T_i - T_{i-1}$ is year fraction of interval, the valuation under forward-swap measure is given by:

$$V(t) = C_{T_0,T_n}(t)E^Q\left[\frac{V(T_0)}{C_{T_0,T_n}(T_0)}|\mathcal{F}_t\right] \quad (3.1d)$$

Consider a European swaption with exercise date t_{ex} and settlement date t_{set},, holder pay fix rate K and receives forward swap rate $F_s(t)$ at t. With annuity $A_o = \sum_{i=1}^n \tau_i P(0,T_i)$, value of payer swaption is

$$V(0) = A_0 E^Q[(F_s(t_{ex}) - K)^+ | \mathcal{F}_0] = A_0(F_0 N(d_+) - KN(d_-)) \; ; d_\pm = \frac{log(\frac{F}{K}) \pm \frac{1}{2}\sigma_B{}^2 t_{ex}}{\sigma_B \sqrt{t_{ex}}} \quad (3.1e)$$

with $F_0 = F(0)$ and forward swap rate $F(t)$ is martingale under forward-swap measure.

The value of call option and swaption above are exactly in accordance to Black's model. Black (1976) postulated a martingale representation where volatility σ_B is constant and forward price $F(t)$ has a geometric Brownian Motion, I.e. $dF = \sigma_B F(t) dW$ with the European call option price given by:

$$V_{call} = D(t_{set})(f_0 N(d_+) - KN(d_-)) \quad (3.1f)$$

3.2 SABR Model

The shortcoming of Black Schole (or Black) model is the assumption of constant volatility across strikes, maturity and do not capture the stylized facts of smile/slew present in empirical IV surfaces. To address this issue, Dupire's (1994) local volatility and stochastic volatility such as SABR (Hagan et al. 2002) or Heston (1993) have been developed. However, the dynamic behaviour of Dupire's local volatility model is inconsistent with observation in interest rate (swaption) market, specifically the smile shifts to higher prices when the underlying forward rate decreases or vice versa. The Stochastic Alpha Beta Rho (SABR) model introduced by Hagan et al.(2002) overcome this problem and enjoys a high popularity as interpolator of implied volatilities.

SABR has become benchmark in fixed-income market due to the traceable closed-form asymptotic formula for the implied volatility (Oosterlee and Grzelak, 2020). For the interest rate derivatives market, implied volatility is readily available from ATM (at-the-money) swaption volatility and Black volatilities of caplets/floorlets. SABR model is a CEV (Constant Elasticity of Variance) asset process (Cox, 1975) where the volatility is specified as a Geometric Brownian motion (GBM) that has some correlation with the underlying forward rate (F) as shown below:

$$dF = \nu F^\beta dW_1 \qquad (0 \leq \beta \leq 1) \quad (3.2a)$$

$$d\nu = \xi \nu dW_2 \qquad (\nu \geq 0) \quad (3.2b)$$

$$\rho dt = < dW_1, dW_2 > \qquad (-1 \leq \rho \leq 1) \quad (3.2c)$$

It has parameters ν(volatility or α originally), ξ (volatility of volatility) and β (power factor) which controls the dynamic of the CEV function and is usually fixed and determined from historical series analysis. SABR model does not incorporate mean-reversion in volatility

and the volatility of volatility parameter, ξ often decreases with expiry of the underlying option. Under $\beta=0$, the forward process, conditional on the volatility path, is normal in accordance to Bachelier's (1900) model which allows negative rate (not exact normal distribution due to correlation). Under $\beta=1$, the forward process, conditional on the volatility path, is lognormal in accordance to Black's (1976) model which allow only positive rate. Under $0.5<\beta<1.0$, it is a CEV asset process. While increasing initial volatility ν_0 increases the level of smile, increasing ρ and ξ increases steepness of curve and curvature of smile respectively. Most fixed-income traders (except Japanese traders) prefer quote prices in terms of Black (lognormal) model. This work will be focusing on the case of $\beta=1$.

As mentioned, Hagan et al. (2002) developed a closed-form approximation for SABR implied volatility. For the case of $\beta=1$, the implied Black volatility (σ_B) with different forward rate (F), strike (K) and option maturity or exercise time (T) is given as below:

$$\sigma_B(K,F) = \frac{\nu_0}{q(K,F)}\frac{z}{X(z)}\left\{1+\left[\frac{(1-\beta)^2}{24(FK)^{1-\beta}}+\frac{\rho\xi\beta\nu_0}{4(FK)^{(1-\beta)/2}}+\frac{2-3\rho^2}{24}\xi^2\right]T+\cdots\right\} \tag{3.2d}$$

$$q(K,F) = (FK)^{(1-\beta)/2}\left(1+\frac{(1-\beta)^2}{24}log^2\left(\frac{F}{K}\right)+\frac{(1-\beta)^4}{1920}+log^4\left(\frac{F}{K}\right)+\cdots\right) \tag{3.2e}$$

$$z = \frac{\xi}{\nu_0}(FK)^{(1-\beta)/2}\,log\left(\frac{F}{K}\right) \tag{3.2f}$$

$$X(z) = log\left(\frac{\sqrt{1-2\rho z+z^2}-\rho+z}{1-\rho}\right) \tag{3.2g}$$

This asymptotic solution is developed under the assumption that $\nu^2 T \ll 1$ and F not too far off from K.

3.3 Artificial Neural Network

Artificial neural network (ANN) have become popular supervised machine learning techniques to extract features and detect patterns from a large data set, ranging from multi-layer neural network to convolutional neural network for image recognition and recurrent neural networks for time series analysis. It thrives due to Universal Approximation Theorem by Hornik et al. (1989) which states that a neural network with at least one hidden layer, sufficient number of neurons and a non-constant continuous activation function can approximate any continuous function well without assuming some mathematical form. Recent advances in data science have shown that even highly non-linear multi-dimensional functions such as solutions to PDEs can be accurately represented through this deep learning technique.

A neural network is biologically-inspired construction consist of collections of interconnected independent nodes, a basic processing unit that performs a weighted sum of its inputs plus a bias and then feed that linearity to a non-linear function known as activation

function. Formally, this is represented by output $y_j = \varphi(\sum_i^n w_i x_i + b_j)$ with x_i as inputs, w_i as weights, b as bias term and φ as activation function (Bishop, 2006). A number of non-overlapping neurons make up one layer and the activation function will determine whether a particular neuron is active. Information flows from the input layer through the inner (hidden) layers and subsequently to the output layer, in the form of a multi-layer perceptron (MLP) represented as composite function $F(x) = f^{(L)}(\dots f^{(2)}\left(f^{(1)}\left(x, \theta^{(1)}\right), \theta^{(2)}\right) \dots, \theta^{(L)})$ where $\theta^{(j)} = (w_j, b_j)$ containing weight matrix and bias vector. For activation function, ReLU (Rectified Linear Unit) given by $\varphi(x) = max(0, x)$ is widely used in recent years in comparison to functions like sigmoid, $\varphi(x) = 1/(1 + e^{-x})$ or tanh, $\varphi(x) = (e^x - e^{-x})/(e^x + e^{-x})$. Sigmoid has desired properties such as smoothness and differentiability but the saturating output may give rise to null (vanishing) gradient problem which impede learning. ReLU also have similar vanishing gradient problem when $x < 0$. There are alternatives that tackle this problem like Leaky ReLU, $\varphi(x) = max(x, ax)$ $; 0 < a < 1$ and eLU (Exponential Linear Unit) $\varphi(x) = x$ when $x > 0$ or $\alpha(e^x - 1)$ when $x < 0$.

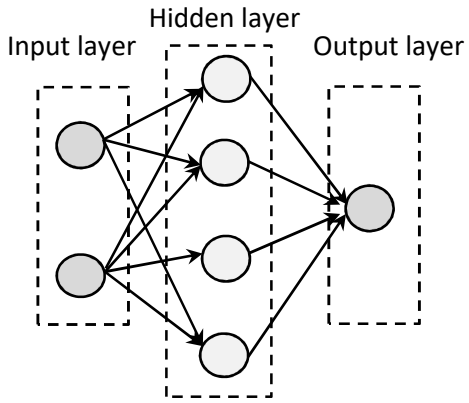


Figure 3.3a – Neural Network

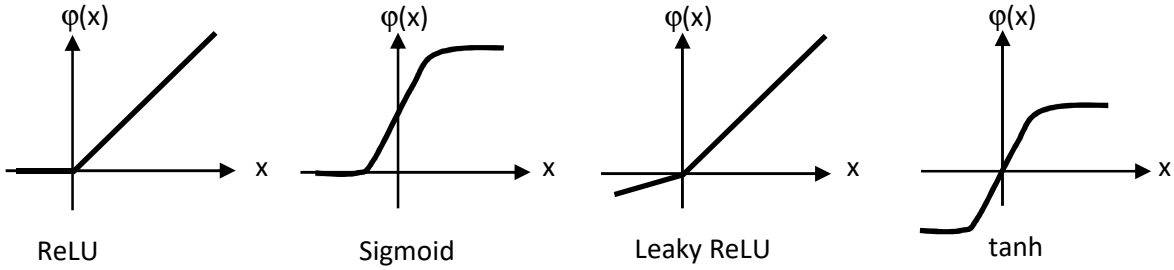


Figure 3.3b – Few types of activation functions

The aim of neural network is to minimize the loss or error function (difference) between neural network composite function $F(x)$ and the target function, $L(\theta) = \|F(x) - f(x)\|$ by adjusting and seeking the optimal weights and biases. This learning process known as 'training' and the whole process in which input translated to loss function through hidden layers is called forward-pass (or propagation). The above optimization problem has been successful solved using few gradient descent methods. Back-propagation methods is an automatic differentiation (AD) algorithm which uses series of differentiation chain rules to find

the gradients or partial derivatives of the loss function with respect to weights and biases of the nodes (eg. $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}.\frac{\partial y}{\partial w}$). This differentiation not only enables training of ANN by founding out how much each node contribute to the error and adjust the weight accordingly, but also allow output sensitivity analysis. Example optimization methods include Stochastic Gradient Descent (SGD) (Robbins and Monro, 1951) and its variants Adam and RMSprop. Stochastic Gradient Descent performs one update on one training dataset at a time and is faster than Batch gradient descent which uses entire training dataset (Faul, 2020). Mini-batch gradient descent takes the middle path approach by using subset of training data during each iteration to calculate the gradient (eg. Adam).

In general, the gradient descent algorithms start with initial values and move in the direction in which the loss function decreases (steepest descent). The expression for updating parameter $\theta = (w, b)$, weights or bias iteratively along negative gradient is given below:

$$\theta \leftarrow \theta - \eta_i \frac{\partial L}{\partial \theta} \qquad i = 0,1,2,\dots,N_T \tag{3.3a}$$

where N_T is the number of trainings, η is the learning rate. Choosing the correct learning rate is important as large value causes oscillation and small values results in slow convergence or trapping in local optima. An adaptive learning rate is often preferred and was implemented in algorithm such as AdaGrad which scale down the steps as moving towards optimum point. RMSProp is a Leaky version of AdaGrad that fixes the AdaGrad problem of slowing down too fast but never converge. Adam incorporates momentum to dampen the oscillation of Gradient Descent and adaptively modifies step size to accelerate the optimization process (Geron, 2019).

Determining network hyperparameters like the number of hidden layers (depth) and the number of neurons (width) requires a separate treatment. Shallower network would require larger number of neurons to reach the same accuracy. Generally, the greater the number of neurons, or a deeper structure, the better the approximations but at a cost of heavier computation or reduction in parallel efficiency. Other considerations include proper weight initialization (usually randomized) for training. Batch normalization to scale output can often speed up the training process. A droupout rate p randomly deactivate portions of neurons in a layer during training to prevent over-fitting and force network to learn more robust feature. Since each activation in the entire network is reduced by a factor p in the testing phase as well, it could slowdown prediction (approximation). Higher mini-batch size yield more accurate or true gradient. High enough epoch (training cycle of processed whole training dataset) ensures loss function has reached minimum but not causing overfitting. The optimum of these hyperparameters could be found in a systematic way (grid search) or random search manner. The later could be more efficient.

4. Methodology

4.1 Finite Difference Method

The first step to obtain the implied volatility (IV) surface or data is to compute the asset or option prices. Black Schole partial differential equation (PDE) that was derived using Feynman-Kac or Ito's Lemma leads to European option price with underlying GBM stock in a closed form solution. SABR model showed the dynamic of the forward rate (F) in the form of Wiener process with GBM volatility and a European option with this underlying can be valued similarly and analytically via PDE. The discussion in section 3.1 shown that the price of Swaption (or Futures option) as function of forward swap rate (or forward price) has annuity factor (or discount factor) that can be decoupled (factored out) under each forward martingales. The PDE for SABR model is similar to Black Scholes PDE in many ways but is two dimensional with zero discount factor (zero drift) and non-constant stochastic volatility:

$$\frac{1}{2}\nu^2 F^{2\beta}\frac{\partial^2 U}{\partial F^2} + \rho\xi F^{\beta}\nu^2\frac{\partial^2 U}{\partial F \partial \alpha} + \frac{1}{2}\xi^2\nu^2\frac{\partial^2 U}{\partial \nu^2} + \frac{\partial U}{\partial \tau} = 0 \tag{4.1a}$$

Defining the PDE in terms of forward rate instead of spot (discounted forward) simplify things and is consistent with Black's equation (equation (3.1f)) when calculating implied volatility (decouple discount factor). However, the SABR PDE does not give rise to closed form option value and thus have to be solved using numerical method such as Finite Difference Method (FDM) or integration methods. FDM would be used as it is a proven numerical procedure to obtain accurate solution approximations to relevant PDE. The problem of using the original SABR PDE above is that coefficients are relative large especially containing F power terms. This would affect the linear approximation and has larger truncation error especially for price of deep ITM options. Hence, a recommended approach is to perform logarithm transformation using $X = ln\, F;\ Y = ln\, \nu$ which yields the PDE with smaller coefficients:

$$\frac{1}{2}\nu^2 F^{2(\beta-1)}\frac{\partial^2 U}{\partial X^2} - \frac{1}{2}\nu^2 F^{2(\beta-1)}\frac{\partial U}{\partial X} + \rho\xi F^{\beta-1}\nu\frac{\partial^2 U}{\partial X \partial Y} + \frac{1}{2}\xi^2\frac{\partial^2 U}{\partial Y^2} - \frac{1}{2}\xi^2\frac{\partial U}{\partial Y} + \frac{\partial U}{\partial \tau} = 0 \tag{4.1b}$$

Using uniform grid approach in discretization for simplicity (i.e. uniform ΔX, ΔY and Δt) and thus setting $F = e^{(Xo+i\Delta X)}\ \ (i = 0, \dots, M)$, $\nu = e^{(Yo+j\Delta Y)}\ \ (j = 0, \dots, N)$ and $t_n = n\Delta t\ \ (n = 0, \dots, P)$ and using central difference scheme approximation (Mariani, and Florescu, 2020):

$$\frac{\partial U}{\partial X} = \frac{(U^n{}_{i+1,j} - U^n{}_{i-1,j})}{X_{i+1} - X_{i-1}} = \frac{(U^n{}_{i+1,j} - U^n{}_{i-1,j})}{2\Delta X} \tag{4.1c}$$

$$\frac{\partial U}{\partial Y} = \frac{(U^n{}_{i,j+1} - U^n{}_{i,j-1})}{Y_{j+1} - Y_{j-1}} = \frac{(U^n{}_{i,j+1} - U^n{}_{i,j-1})}{2\Delta Y} \tag{4.1d}$$

$$\frac{\partial^2 U}{\partial X^2} = \frac{\frac{U^n{}_{i+1,j} - U^n{}_{i,j}}{X_{i+1} - X_i} - \frac{U^n{}_{i,j} - U^n{}_{i-1,j}}{X_i - X_{i-1}}}{X_{i+1} - X_i} = \frac{(U^n{}_{i+1,j} - U^n{}_{i,j}) - (U^n{}_{i,j} - U^n{}_{i-1,j})}{\Delta X^2} \tag{4.1e}$$

$$\frac{\partial^2 U}{\partial Y^2} = \frac{\frac{U^n{}_{i,j+1} - U^n{}_{i,j}}{Y_{j+1} - Y_j} - \frac{U^n{}_{i,j} - U^n{}_{i,j-1}}{Y_j - Y_{j-1}}}{Y_{j+1} - Y_j} = \frac{(U^n{}_{i,j+1} - U^n{}_{i,j}) - (U^n{}_{i,j} - U^n{}_{i,j-1})}{\Delta Y^2} \quad (4.1f)$$

$$\frac{\partial^2 U}{\partial X \partial Y} = \frac{U^n{}_{i+1,j+1} - U^n{}_{i-1,j+1} - U^n{}_{i+1,j-1} + U^n{}_{i-1,j-1}}{(X_{i+1} - X_{i-1})(Y_{j+1} - Y_{j-1})} = \frac{U^n{}_{i+1,j+1} - U^n{}_{i-1,j+1} - U^n{}_{i+1,j-1} + U^n{}_{i-1,j-1}}{4(\Delta X)(\Delta Y)} \quad (4.1g)$$

$$\frac{\partial U}{\partial t} = \frac{U^{n+1}{}_{i,j} - U^n{}_{i,j}}{t_{n+1} - t_n} = \frac{U^{n+1}{}_{i,j} - U^n{}_{i,j}}{\Delta t} \quad (4.1h)$$

where $U^n{}_{i,j}$represent the option price at specific grid point and $t = T - \tau$ reflects the time to maturity ($\frac{\partial U}{\partial t} = -\frac{\partial U}{\partial \tau}$), will lead to expression (explicit method):

$$U^{n+1}{}_{i,j} = \boldsymbol{a_u} U^n{}_{i+1,j+1} + \boldsymbol{a_m} U^n{}_{i,j+1} + \boldsymbol{a_d} U^n{}_{i-1,j+1} + \boldsymbol{b_u} U^n{}_{i+1,j} + \boldsymbol{b_m} U^n{}_{i,j} + \boldsymbol{b_d} U^n{}_{i-1,j} + \boldsymbol{c_u} U^n{}_{i+1,j-1} + \boldsymbol{c_m} U^n{}_{i,j-1} + \boldsymbol{c_d} U^n{}_{i-1,j-1} \quad (4.1i)$$

where

$$\boldsymbol{a_u} = -\boldsymbol{a_d} = -\boldsymbol{c_u} = \boldsymbol{c_d} = \rho \xi e^{(Y_o + j\Delta Y)} e^{(\beta-1)(X_o + i\Delta X)} \frac{\Delta t}{4 \Delta X \Delta Y} \quad (4.1j)$$

$$\boldsymbol{a_m} = (\frac{1}{2}\xi^2 \frac{1}{\Delta Y^2} - \frac{1}{2}\xi^2 \frac{1}{2\Delta Y})\Delta t \quad (4.1k)$$

$$\boldsymbol{c_m} = (\frac{1}{2}\xi^2 \frac{1}{\Delta Y^2} + \frac{1}{2}\xi^2 \frac{1}{2\Delta Y})\Delta t \quad (4.1l)$$

$$\boldsymbol{b_u} = (\frac{1}{2} e^{2(Y_o + j\Delta Y)} e^{2(\beta-1)(X_o + i\Delta X)} \frac{1}{\Delta X^2} - \frac{1}{2} e^{2(Y_o + j\Delta Y)} e^{2(\beta-1)(X_o + i\Delta X)} \frac{1}{2\Delta X})\Delta t \quad (4.1m)$$

$$\boldsymbol{b_m} = (-\frac{1}{2}\xi^2 \frac{1}{\Delta Y^2} - e^{2(Y_o + j\Delta Y)} e^{2(\beta-1)(X_o + i\Delta X)} \frac{1}{\Delta X^2})\Delta t + 1 \quad (4.1n)$$

$$\boldsymbol{b_d} = (\frac{1}{2} e^{2(Y_o + j\Delta Y)} e^{2(\beta-1)(X_o + i\Delta X)} \frac{1}{\Delta X^2} + \frac{1}{2} e^{2(Y_o + j\Delta Y)} e^{2(\beta-1)(X_o + i\Delta X)} \frac{1}{2\Delta X})\Delta t \quad (4.1o)$$

The logarithm terms ensures that forward rate and volatility cannot be negative. FDM has finite number of grids and requires artificial boundary conditions. Two types of boundary conditions are considered: Dirichlet and Neumann-boundary conditions, where the former assumes value is known or the later assumes derivative of function is known at the boundary. In the case of a European call option $U(F, \nu, t)$, the boundary conditions include $U(0, \nu, t) = 0$ (call option worthless at $F_{min}=0$) , $\frac{\partial U}{\partial F}(F_{max}, \nu, t) = 1$ (delta of option becomes one at maximum F), $\frac{\partial U}{\partial F}(F, \nu_{max}, t) = 1$ (option is essentially F-K at maximum volatility). Interpreting them in the logarithm domain would be $U = 0$ at lower boundary $X_{min}=Xo$ and $\frac{\partial U}{\partial F} = 1$ translated into $\frac{\partial U}{\partial X} = e^X$ (since chain rule: $\frac{\partial U}{\partial F} = \frac{\partial U}{\partial X} \cdot \frac{\partial X}{\partial F} = \frac{\partial U}{\partial X} \frac{1}{e^X}$), i.e. $\frac{U^n{}_{i,j} - U^n{}_{i-1,j}}{\Delta X} = e^{(Xo + i\Delta X)}$. Under minimum volatility $\nu = 0$, the PDE (equation (4.1b)) becomes $\frac{1}{2}\xi^2 \frac{\partial^2 U}{\partial Y^2} - \frac{1}{2}\xi^2 \frac{\partial U}{\partial Y} + \frac{\partial U}{\partial \tau} = 0$ which leads to $U^{n+1}{}_{i,j} = \boldsymbol{r_u} U^n{}_{i,j+1} + \boldsymbol{r_m} U^n{}_{i,j} + \boldsymbol{r_d} U^n{}_{i,j-1}$ where $\boldsymbol{r_u} = (\frac{1}{2}\xi^2 \frac{1}{\Delta Y^2} - \frac{1}{2}\xi^2 \frac{1}{2\Delta Y})\Delta t$, $\boldsymbol{r_m} = 1 - \Delta t \xi^2 \frac{1}{\Delta Y^2}$ and $\boldsymbol{r_d} = (\frac{1}{2}\xi^2 \frac{1}{\Delta Y^2} + \frac{1}{2}\xi^2 \frac{1}{2\Delta Y})\Delta t$. As the explicit equation is using time to maturity convention and starts at $t=0$ (maturity), the FDM becomes an initial value problem with

option value equal to payoff $U(F,\nu,0) = (F-K)^+ = (e^{(Xo+i\Delta X)} - K)^+$. Representing the explicit difference equation (equation (4.1i)) and boundary conditions in a matrix, we have finite difference computation simplified to matrix multiplication problem (A. B =C) and the option price is then $U^P{}_{I,J}$ where $F_o = e^{(Xo+I\Delta X)}$, $\nu_o = e^{(Yo+J\Delta X)}$ and $T = P\Delta t$. For example, for the case of N=5 and M=5 shown below:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ : & : & : & : & : & : & : & : & : & : & : & : & : & : & : \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_u & a_m & a_d & 0 & 0 & 0 & b_u & b_m & b_d & .. & c_u & c_m & c_d & .. & 0 \\ 0 & a_u & a_m & a_d & 0 & 0 & 0 & b_u & b_m & b_d & 0 & 0 & c_u & .. & 0 \\ 0 & 0 & a_u & a_m & a_d & 0 & 0 & 0 & b_u & b_m & b_d & 0 & 0 & .. & 0 \\ 0 & 0 & 0 & a_u & a_m & a_d & 0 & 0 & 0 & b_u & b_m & b_d & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & .. & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_u & a_m & a_d & 0 & 0 & 0 & b_u.. & .. & 0 \\ : & : & : & : & : & : & : & : & : & : & : & : & ...r_u & r_m & r_d \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & ... & 0 & 1 \end{pmatrix} \begin{pmatrix} U^n_{5,5} \\ U^n_{4,5} \\ U^n_{3,5} \\ : \\ U^n_{0,5} \\ U^n_{5,4} \\ U^n_{4,4} \\ U^n_{3,4} \\ U^n_{2,4} \\ U^n_{1,4} \\ U^n_{0,4} \\ U^n_{5,3} \\ U^n_{4,3} \\ : \\ U^n_{0,0} \end{pmatrix} = \begin{pmatrix} \Delta X.e^{(Xo+5\Delta x)} \\ \Delta X.e^{(Xo+4\Delta x)} \\ \Delta X.e^{(Xo+3\Delta x)} \\ : \\ 0 \\ \Delta X.e^{(Xo+5\Delta x)} \\ U^{n+1}_{4,4} \\ U^{n+1}_{3,4} \\ U^{n+1}_{2,4} \\ U^{n+1}_{1,4} \\ 0 \\ \Delta X.e^{(Xo+5\Delta x)} \\ U^{n+1}_{4,3} \\ : \\ 0 \end{pmatrix} \quad (4.1p)$$

The upper and lower part of matrix A impose the boundary conditions and middle part ($a_w, a_m, \dots, c_d$ terms) is the actual computation where multiplication happens and propagates the option values across time (to maturity). Notice that matrix A is not the typical tridiagonal matrix. Unlike implicit method which is unconditionally stable, explicit method has a stability requirement for convergence to avoid matrix multiplication exploding into extremely large values. A thorough stability analysis would involve complex spectral radius analysis of the matrix A or Fourier analysis. A more direct stability method check is to use matrix norms (Conlisk, 1973). Under equation of the form $y_t = Ay_{t-1} + b$, a sufficient stability condition is $\mu(A) \leq 1$ where $\mu(\cdot)$ is spectral radius (largest absolute value of eigenvalues). Using the matrix norm property of $\mu(A) \leq f(A)$, if matrix norm $f(A) \leq 1$ then stability is guaranteed. The downside of this method is that sufficient condition $f(A) \leq 1$ may be far stronger than necessary. Using the infinity norm $f(A) = \|A\|_\infty \leq 1$ which is the maximum of row sum of absolute matrix elements $max_{1\leq i\leq n}\left(\sum_{j=1}^{n} |a_{ij}|\right)$ (Goddard consulting, 2021) and focusing on the middle part of matrix A, we have $|\mathbf{a_u}| + |\mathbf{a_m}| + |\mathbf{a_d}| + |\mathbf{b_u}| + \dots + |\mathbf{c_d}| \leq 1$ and taking $|\mathbf{a_u}|, |\mathbf{a_d}|, |\mathbf{c_u}|$ and $|\mathbf{c_d}|$ term to be much smaller in comparison (negligible) thus lead to:

$$\Delta t \leq \frac{1}{\left(\frac{\xi_{max}{}^2}{\Delta Y^2} + \frac{e^{(Yo+N\Delta Y)}.e^{2(\beta-1)(Xo+M\Delta X)}}{\Delta X^2}\right)} \quad (4.1q)$$

The above constraint for time steps will be the stability criteria. The same results could be obtained if the maximum element norm $f(A) = n\, max_{i,j}(|a_{ij}|) \leq 1$ was used, where $\boldsymbol{b_m}$ is the maximum element and using approximation $1/n \approx 0$ (n = dimension of matrix).

4.2 Data

In order to define how different parameters are chosen to compute SABR option price data sets for ANN training and calibration, actual market data is reviewed. For example, the swaption data (implied volatilities) typically has different maturities, tenors and strikes. The maturity and tenor typically range from few months (1M, 3M, 6M) to tenths of years (up to 10,20 or 30 years). The volatility of volatility and correlation are typically higher for short maturity (or tenor) as slope and curvature increase (McGhee, 2018). Thus, implied volatilities or option price will be in smaller range for longer maturity (or tenor) and neural network prediction or calibration is expected to be better. Fixed levels of parameter values are used for simplicity and input parameters range for FDM computation has been chosen as below:

$$\nu_0 = \{0.02, 0.10, 0.30, 0.50\}$$

$$\xi = \{0.005, 0.05, 0.10, 0.30, 0.50\}$$

$$\rho = \{-0.70, -0.40, -0.10, 0.10, 0.40, 0.70\}$$

Ten strike values (K) with uniform spacing and a two specific forward rates $\mathrm{F_o}$ were chosen below since they yield a complete sets of relative strikes with respect to ATM rate $(K - F_o)$, i.e. {-200bps, -150bps, - 100bps, -50 bps, 0, +50bps, +100bps, +150 bps, +200 bps}:

$$K = \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5\}$$

$$F_0 = \{3.0, 3.5\}$$

The maturities selected span from reasonably short term to long term that allow fitting of most available market data while maintaining reasonable amount of computation time for data collection:

$$T = \{0.5Y,\ 1Y,\ 2Y,\ 5Y,\ 10Y, 20Y\}$$

With the range of parameters chosen above, it would generate all combinations of 6[T]x10[K]x6[ρ]x5[ξ] x2[F_0]x4[ν_0] = 14,400 data values which is sufficiently large. As nine implied volatilities appear at the ANN output concurrently (section 4.4), there is 1,600 data sets for neural actual network training and calibration.

The option prices that were computed from FDM will be converted into implied volatilities using root-finding method on Black's equation (equation 3.1f). Specifically, Dekker's method was employed to avoid non-convergence or stalling when Secant or Bisection method alone was applied. It is a hybrid method that retains the root bracketing interval of Bisection

and superlinear convergence of Secant method. When the function values are very close to each other, Secant method will experience difficulty (or slow convergence) and Dekker circumvent this by switching to Bisection approach. The algorithm of Dekker's can be briefly summarized as the following: The root of function $f(x)$ is b_k (has smaller function value) and the contra point is a_k such that $f(a_k)$ and $f(b_k)$ have opposite signs and the root lies between $[a_k, b_k]$. Previous b_{k-1} was initially set as a_0 and the provisional value of next iterate is either from s (Secant method) or m (Bisection Method) in which the former will be chosen if s lies between b_k and m or otherwise

$$s = b_k - \frac{b_k - b_{k-1}}{f(b_k) - f(b_{k-1})} f(b_k) \tag{4.2a}$$

$$m = \frac{1}{2}(a_k + b_k) \tag{4.2b}$$

The next iterate of contra value a_{k+1} would remain the same as a_k if $f(a_k)$ and $f(b_{k+1})$ have opposite signs or otherwise be assigned to b_k. The value of a_k and b_k needs to be swapped whenever $|f(a_k)| < |f(b_k)|$ to ensure b_k has smaller function value (the better approximate). The iteration stops when difference between consecutive b_k is very small ($b_{k+1} - b_k <$ tolerance).

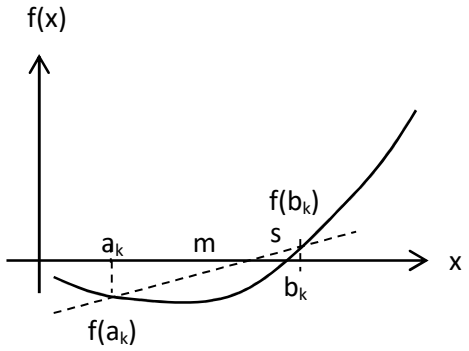


Figure 4.2 – Dekker's method for root-finding

4.3 Calibration

The purpose of using artificial neural network (ANN) in calibration to market data is to offload the computationally intensive forward pricing of asset to offline processing (remove bottleneck) and allow the network to price (approximate or predict) online quickly after learning (or training) using data that defines the input-output relationship between parameters and asset price. The performance of trained ANN is usually evaluated by inputting out-of-sample testing data. In this work, the pricing is in the form of implied volatilities derived from option prices. The direct pricing (forward-pass) then proceeds to next step (backward-pass) of finding parameters that closes the gap with market data. This second step usually involves classical explicit method of minimization or optimization technique like LM or BFGS

algorithm (gradient based) or DE technique (gradient free approach). Alternatively, it is also feasible to utilize the gradient information from the trained ANN to compute the search direction and solution such as used in the inverse map approach.

In the inverse map approach (backward-pass), an inverse ANN is established such that input layer now becomes learnable layer and outputs becomes inputs taking in market volatilities data to solve inverse problem. Mathematically, what the inverse ANN does is represented as: $\mathcal{F}^{-1}\{\{\sigma_{mkt}\}_{i,j},\{F\}_j,\{K\}_i,\{T\}_j\} \rightarrow p$. This is best described in the figure 4.3 below. There are two sets of parameters, the parameters to be calibrated denoted by $p = \{\rho, \xi, v_o\}$ and the observable parameters $\theta_{i,j} = \{F_o, K, T\}$. Given a set of market implied volatilities σ_{mkt} for a set of maturities T_j, forward rate F_j, and strikes K_i, i.e. $\theta_{i,j} = \{F_o, K, T\}$, the inverse ANN returns a unique value of $p_{ANN,b}$.

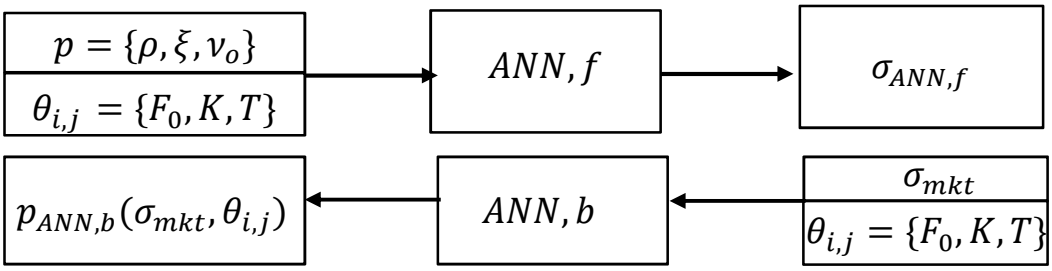


Figure 4.3 Two stages of Inverse Map approach for calibration

The output objective function that solves the calibration or optimization problem is given by $arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{i,j} \left\|\sigma_{mkt}(\theta_{i,j}) - \sigma(\theta_{i,j}, p)\right\|$ which can be reformulated as input-side objective function $arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m} \bar{\omega}_{i,j} \left\|p_{ANN,b}(\sigma_{mkt}, \theta_{i,j}) - p\right\|$

$= arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{N} \bar{\omega}_{i,j,k} \left\|p_{ANN,b}(\sigma_{mkt}, \theta_{i,j}) - p_k\right\|$ where the calibrated parameters are p_k (k=1,…,N) (Itkin, 2019). The reformulation is correct as both the objective functions solve the same problem as proven below:

$$= arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{i,j} \left\|\sigma_{mkt}(\theta_{i,j}) - \sigma(\theta_{i,j}, p)\right\|$$

$$= arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{i,j} \left\|\sigma_{ANN,f}(\theta_{i,j}, p_{ANN,b}(\sigma_{mkt}, \theta_{i,j})) - \sigma_{ANN,f}(\theta_{i,j}, p)\right\|$$

$$\leq arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{i,j} \sum_{k=1}^{N} \left|\frac{\partial \sigma_{ANN,f}(\theta_{i,j}, p_{k,ANN,b})}{\partial p_{k,ANN,b}}\right| \left|p_{k,ANN,b} - p_k\right|$$

$$\leq arg\ \min_p \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{N} \omega_{i,j,k} \left|\frac{\partial \sigma_{ANN,f}(\theta_{i,j}, p_{k,ANN,b})}{\partial p_{k,ANN,b}}\right| \left|p_{k,ANN,b} - p_k\right| \quad (4.3a)$$

using Taylor series expansion on all parameters p:

$$\sigma_{ANN,f}\left(\theta_{i,j}, p_{ANN,b}\left(\sigma_{mkt}, \theta_{i,j}\right)\right) - \sigma_{ANN,f}\left(\theta_{i,j}, p\right) = \sum_{k=1}^{N} \frac{\partial \sigma_{ANN,f}\left(\theta_{i,j}, p_{k,ANN,b}\right)}{\partial p_{k,ANN,b}} \left(p_{k,ANN,b} - p_k\right) + \cdots \quad (4.3b)$$

Assigning $\bar{\omega}_{i,j,k} = \omega_{i,j,k} \left|\frac{\partial \sigma_{ANN,f}\left(\theta_{i,j}, p_{k,ANN,b}\right)}{\partial p_{k,ANN,b}}\right|$ completes the proof. The gradient of forward ANN output with respect to each input parameter can be found as it is computed by ANN via automatic differentiation (AAD) in backpropagation during training (eg. GradientTape in Tensorflow)

The obvious solution to the objective function (thus optimum parameters) is given by

$$p_k = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} \bar{\omega}_{i,j,k} \; p_{k,ANN,b}}{\sum_{i=1}^{n} \sum_{j=1}^{m} \bar{\omega}_{i,j,k}} \quad (4.3c)$$

4.4 ANN Construct

The artificial neural network (forward or inverse/backward ANN) implemented is constructed from two hidden layers with same number of nodes, specifically 160. This relative large number of nodes (width) maintains reasonable level of accuracy comparable to those using three hidden layers but lesser nodes. The network hyperparameters were determined via random search. The optimizer used for training is Adam optimizer with loss function of mean squared error (MSE) as metric. The original default learning rate of 0.001 (maximum adaptive rate) was used as it produce the best results. Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. The activation function selected is ReLU as it yields the best results compared to others such as eLU or tanh. Alternatively, LeakyReLU function would have comparable good results if used. A dropout rate of 1/5 was applied to the first layer to reduce the possible effects of overfitting. The training was also done using 250 epochs with default batch size of 32 and all these settings yield good results.

The forward ANN takes five inputs with three being the SABR model parameters to be calibrated: ξ, v_o and ρ and the other two being the observable variable maturity, T and forward swap rate, F_o. There will be 9 outputs of implied volatility values corresponding to 9 fixed relative strike values with respect to forward (ATM) rate, i.e. K- F_o. Since the (relative) strike values are always fixed, they are not part of the input of neural network. Thus, each data is formed from 9 input-output pair data that is conglomerated together, leading to actual total unique data sets of 14,400/9=1,600. With this, data sets are split into two portions randomly such that 80% become training data and 20% ended up as testing data. Large training data is utilized as prediction accuracy increases with the training data set size.

The backward or inverse ANN flips the output volatilities as inputs (number of inverse inputs depend on the availability market volatility data corresponding to relative position to ATM) and still uses the two observable variable maturity, T and forward swap rate, F_o as inputs. The original 3 input parameters (ξ, v_o and ρ) are converted to inverse ANN outputs. The training data to test data ratio of 80:20 is maintained. Both the ANN constructs are shown in figure 4.4 below:

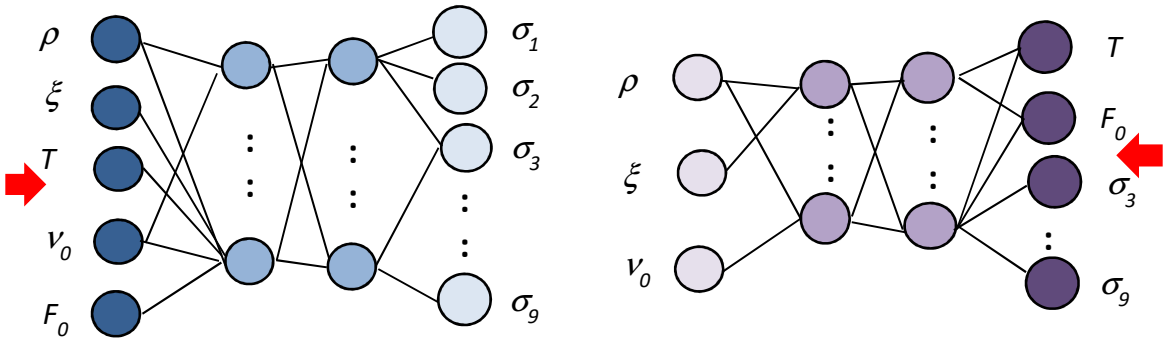


Figure 4.4 Architecture of forward ANN (left) and inverse ANN (right)

5. Results & Discussion

As the quality of data has an impact on the performance of the resulting ANN, the implied volatilities derived from option prices that are computed using FDM are checked against Hagan's approximation values. Table 5.1 below shows the RMSE (Root Mean Square Error) of volatility values (i.e. $RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left(\sigma_{FDM,k} - \sigma_{Hagan,k}\right)^2}$) for the case of small parameters (small volatility) and large parameters (large volatility) for both short maturity and long maturity. Larger parameters have larger RMSE as expected due to higher non-linearity introduced in finite difference (see PDE equation (4.1b)). Smaller parameters curve are not smooth (zig-zag) due to difficulty of root-finding as Black's equation curve flatten at small volatilities as shown in figure 5.1. Longer maturity has smaller RMSE in comparison to shorter maturity. Overall, the implied volatilities data from FDM has reasonably well values and thus used for ANN training. The 8 sub-plots (figure 5.2) illustrates the FDM implied volatilities and Hagan's approximated volatilities plot against different strikes.

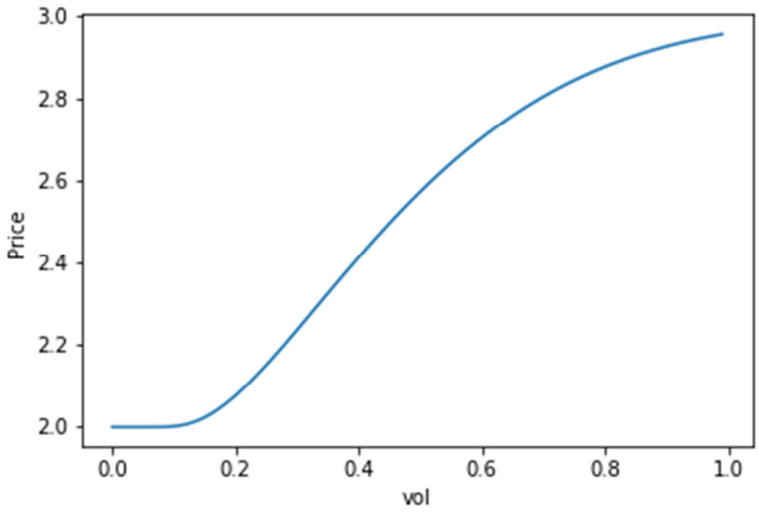


Figure 5.1 – Black's equation price vs. volatility for case of F_0=3.0, K=1.0 flattens at small volatilities

ρ	ξ	v_o	T	F_o	RMSE
-0.7	0.05	0.02	1	3.0	0.023471
-0.7	0.05	0.02	1	3.5	0.013046
0.7	0.5	0.3	1	3.0	0.13209
0.7	0.5	0.3	1	3.5	0.09551
-0.7	0.05	0.02	10	3.0	0.008311
-0.7	0.05	0.02	10	3.5	0.008379
0.7	0.5	0.3	10	3.0	0.077828
0.7	0.5	0.3	10	3.5	0.077190

Table 5.1 – RMSE between FD's implied volatilities and Hagan's approximation for different parameters and maturities

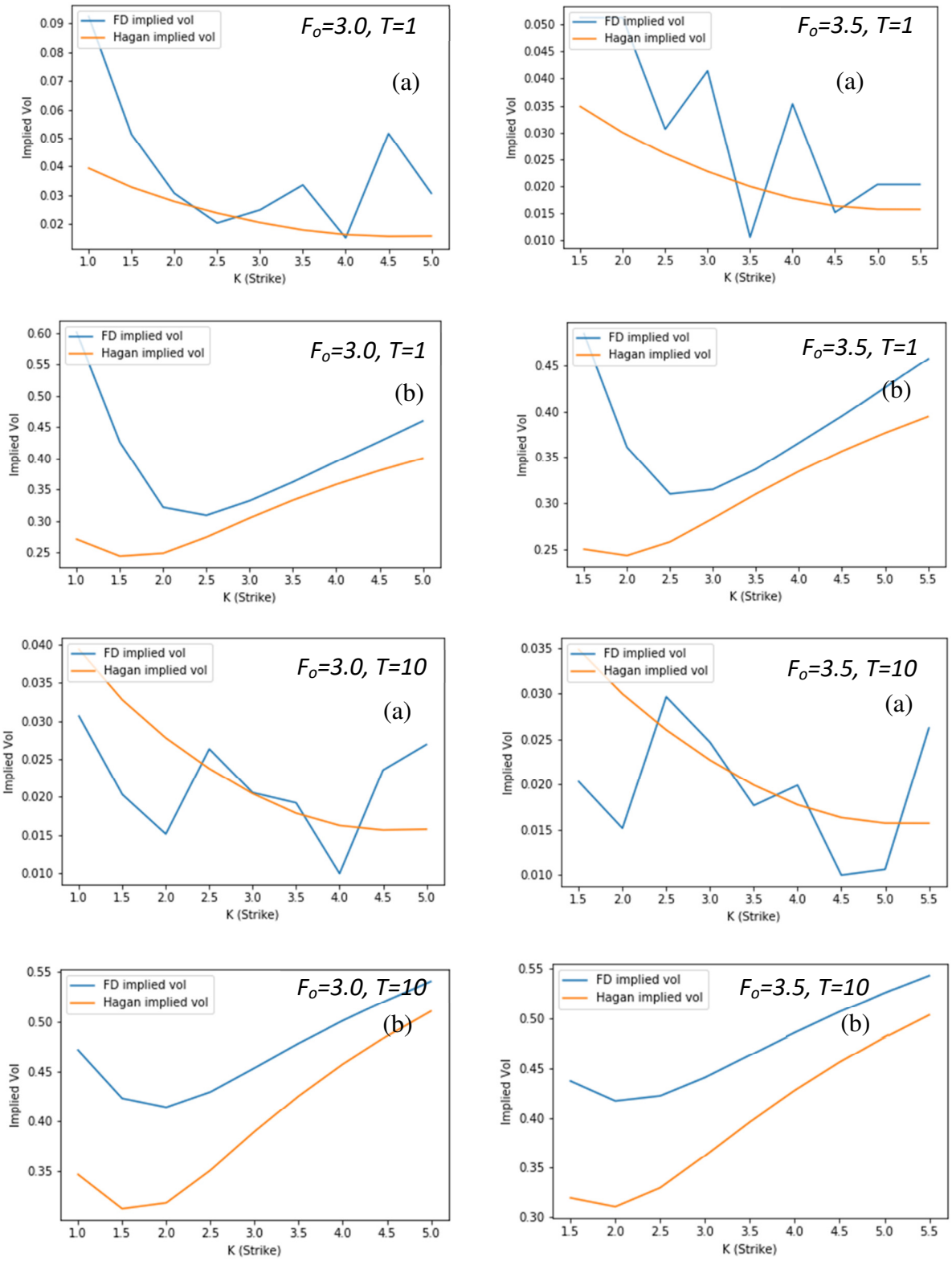


Figure 5.2 – FD implied volatilities and Hagan's approximated implied volatilities plots for (a) small parameters $\rho = -0.7, \xi = 0.05, v_o = 0.02$ and (b) large parameters $\rho = 0.7, \xi = 0.5, v_o = 0.3$

The performance of the trained ANN (forward- or backward-pass) was evaluated based on Mean Squared Error (MSE) loss (function) which is the average of the squared difference between predicted and actual value:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.1}$$

where y_i is the actual value and $\hat{y}_i$ is the ANN predicted value. The 250 epochs allows the MSE to be reduced to small values sufficiently during training for both forward-pass and backward-pass ANN as shown in figure 5.3 and 5.4. The network hyperparameters were chosen and verified via K-fold cross validation (K=5) such that it produces minimum MSE. Cross-validation is a resampling method that estimates a model's predictive performance by using different portions of the data for testing and training on different iterations to avoid the problem of overfitting or selection bias.

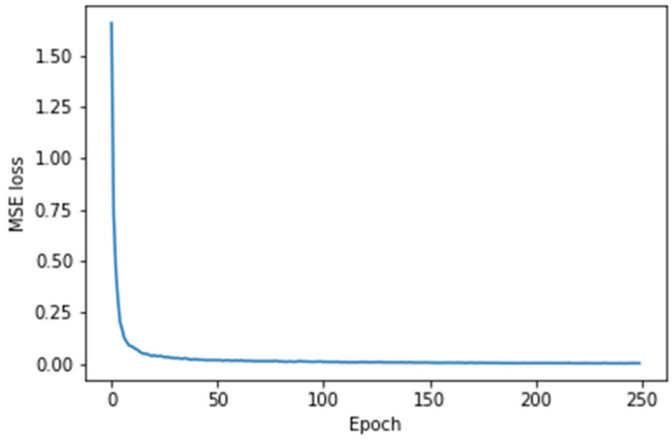


Figure 5.3 – Total MSE Loss of the forward ANN during training

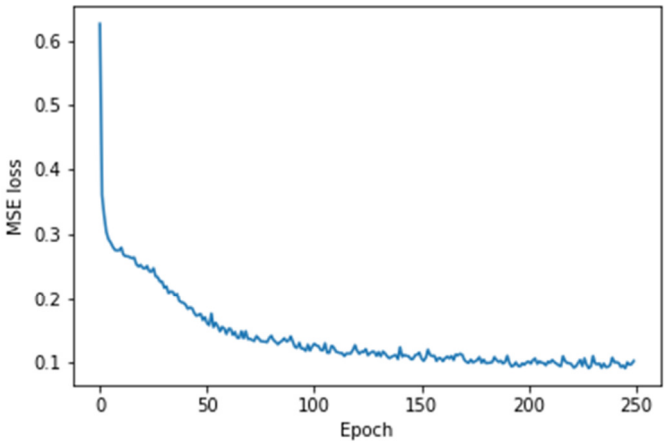


Figure 5.4 – Total MSE Loss of the inverse (backward) ANN during training

Iteration 1	loss: 1.16%
Iteration 2	loss: 1.38%
Iteration 3	loss: 1.65%
Iteration 4	loss: 1.02%
Iteration 5	loss: 1.28%
Average	1.30% (+/- 0.21%)

Iteration 1	loss: 9.20%
Iteration 2	loss: 8.24%
Iteration 3	loss: 8.19%
Iteration 4	loss: 11.14%
Iteration 5	loss: 9.71%
Average	9.30% (+/- 1.09%)

Table 5.2 – MSE loss over K-fold cross validation for forward ANN (left) and backward ANN (right)

Out-of-sample performance of trained ANN was evaluated using explained variance, another accuracy metric which measure the discrepancy between predicted value and actual test data (higher score is desired):

$$Explain\ var = 1 - \frac{Var(y-\hat{y})}{Var(y)} \tag{5.2}$$

The forward ANN has good explained variance (99%) as shown in figures 5.5a and 5.5b with two out of the nine implied volatility outputs shown. On the other hand, the inverse ANN has reasonable and acceptable explained variance for both outputs ρ and ξ (66% and 77% respectively) and good explained variance for ν_0 (99%) (see figure 5.6a-c). It has less than ideal explained variance due to possibility of presence of multiple-to-one mapping between original input-output pair in which inversion ANN would produce average value of original ANN inputs instead. This depends greatly on what are the combination of inputs and combination of grouped output values and is the limitation of inverse ANN solver approach.

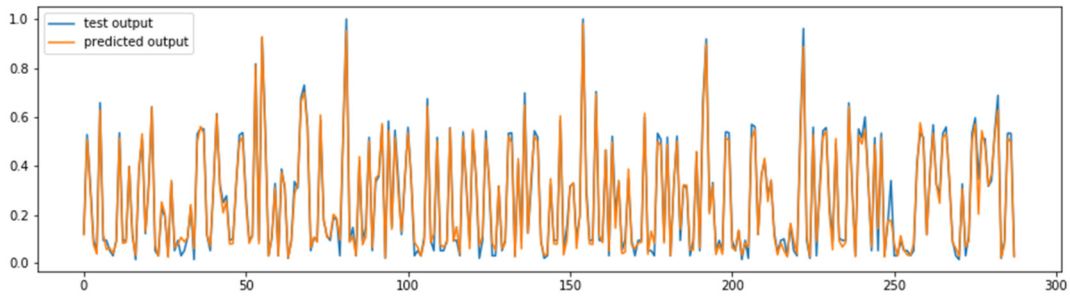


Figure 5.5a – Forward ANN: Test data vs. predicted output for first ouput (σ_1) with explained variance of 98.5%

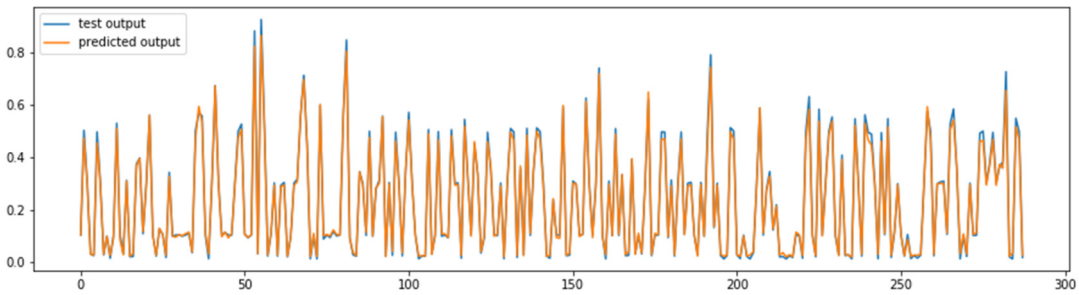


Figure 5.5b – Forward ANN: Test data vs. predicted output for fifth output (σ_5) with explained variance of 99.5%

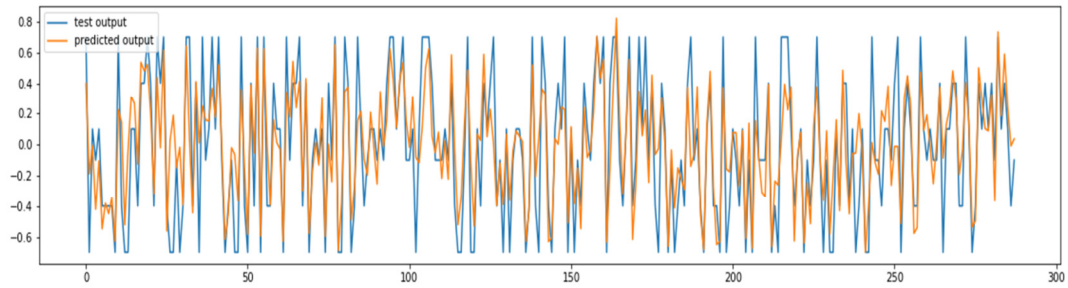


Figure 5.6a – Inverse ANN: Test data vs. predicted output for output (ρ) with explained variance of 65.7%

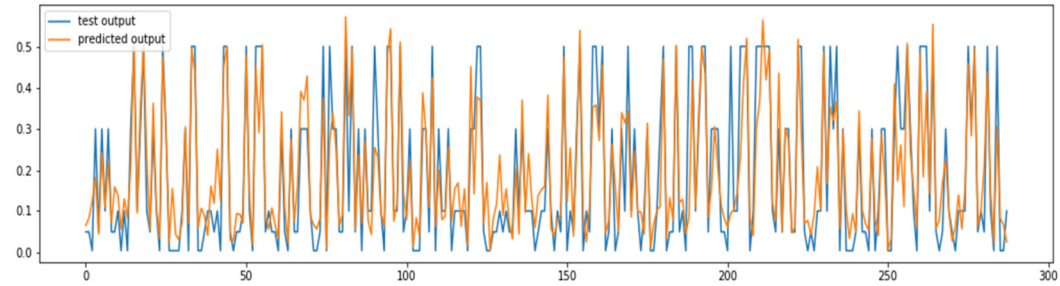


Figure 5.6b– Inverse ANN: Test data vs. predicted output for output (ξ) with explained variance of 77.3%

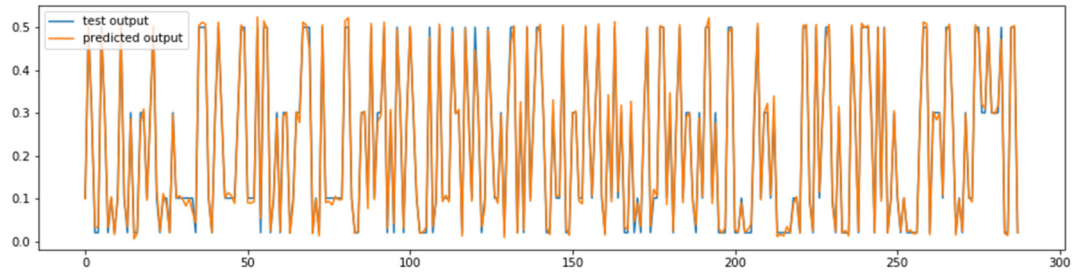
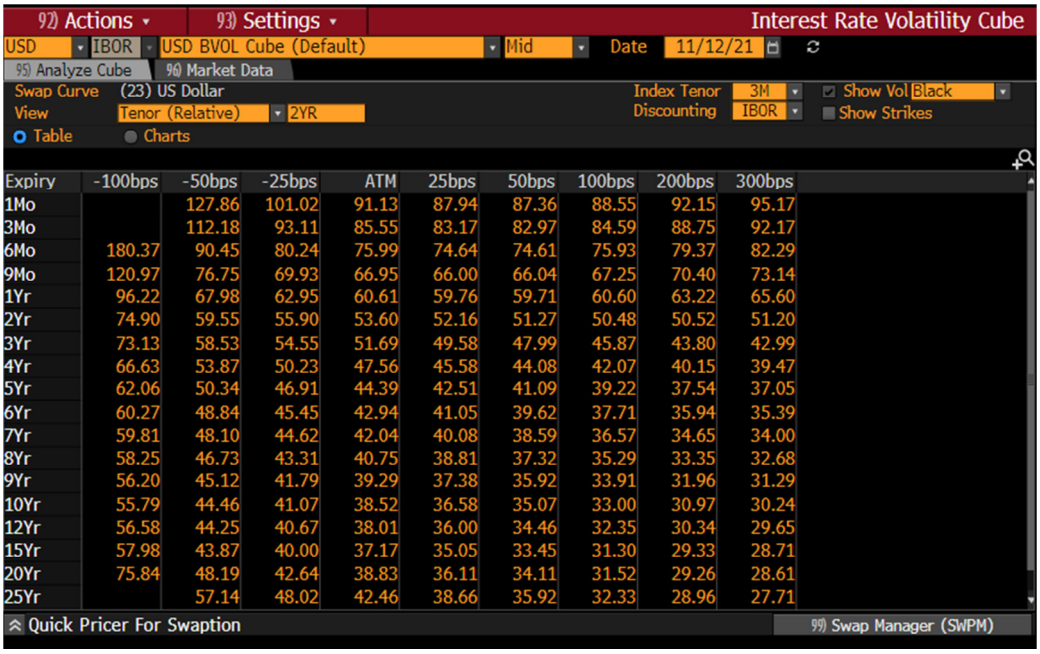


Figure 5.6c– Inverse ANN: Test data vs. predicted output for output (ν_0) with explained variance of 99.5%

Calibration of SABR model parameters was done using daily swaption market data (implied volatilities) extracted from Bloomberg VCUB and one example (USD currency for 2Y tenor) is shown in figure below. The data with maturity T={0.5, 1, 2, 5, 10, 20} and relative strike values of -100bps, -50 bps, 0 bps (ATM), 50bps, 100bps and 200 bps would be used which correspond to our σ_3, σ_4, σ_5, σ_6, σ_7 and σ_9.



Expiry	-100bps	-50bps	-25bps	ATM	25bps	50bps	100bps	200bps	300bps
1Mo		127.86	101.02	91.13	87.94	87.36	88.55	92.15	95.17
3Mo		112.18	93.11	85.55	83.17	82.97	84.59	88.75	92.17
6Mo	180.37	90.45	80.24	75.99	74.64	74.61	75.93	79.37	82.29
9Mo	120.97	76.75	69.93	66.95	66.00	66.04	67.25	70.40	73.14
1Yr	96.22	67.98	62.95	60.61	59.76	59.71	60.60	63.22	65.60
2Yr	74.90	59.55	55.90	53.60	52.16	51.27	50.48	50.52	51.20
3Yr	73.13	58.53	54.55	51.69	49.58	47.99	45.87	43.80	42.99
4Yr	66.63	53.87	50.23	47.56	45.58	44.08	42.07	40.15	39.47
5Yr	62.06	50.34	46.91	44.39	42.51	41.09	39.22	37.54	37.05
6Yr	60.27	48.84	45.45	42.94	41.05	39.62	37.71	35.94	35.39
7Yr	59.81	48.10	44.62	42.04	40.08	38.59	36.57	34.65	34.00
8Yr	58.25	46.73	43.31	40.75	38.81	37.32	35.29	33.35	32.68
9Yr	56.20	45.12	41.79	39.29	37.38	35.92	33.91	31.96	31.29
10Yr	55.79	44.46	41.07	38.52	36.58	35.07	33.00	30.97	30.24
12Yr	56.58	44.25	40.67	38.01	36.00	34.46	32.35	30.34	29.65
15Yr	57.98	43.87	40.00	37.17	35.05	33.45	31.30	29.33	28.71
20Yr	75.84	48.19	42.64	38.83	36.11	34.11	31.52	29.26	28.61
25Yr		57.14	48.02	42.46	38.66	35.92	32.33	28.96	27.71

Figure 5.7 – USD Swaption Market Implied Volatilities on 11/12/2021 from Bloomberg (VCUB)

The optimum calibrated parameters were found to be $\{\rho = -0.433, \xi = 0.677, v_o = 0.537\}$ using equation (4.3c) restated as below:

$$p_k = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} \bar{\omega}_{i,j,k}\ p_{k,ANN,b}}{\sum_{i=1}^{n}\sum_{j=1}^{m} \bar{\omega}_{i,j,k}} \tag{5.3}$$

The calibrated parameters were fed into the forward ANN to generate predicted implied volatilities to compare with the market quotes and the performance of calibration was evaluated using RMSE below:

$$RMSE_{ANN} = \sqrt{\frac{1}{N}\sum(\sigma_{MKT} - \sigma_{ANN}(p_k))^2} \tag{5.4}$$

where $\sigma_{ANN}(p_k)$ is the ANN predicted volatilities using calibrated parameters and σ_{MKT} are the N market quotes. The $RMSE_{ANN}$ were found to be 0.217 (21.7%). The ANN predicted volatilities and market quotes are juxtaposed in table 5.3. If the calibrated parameters were used in Hagan's approximation and the implied volatility values are compared with market quotes, the RMSE (equation below) was found to be 0.187 (18.7%) and quite close to $RMSE_{ANN.}$

$$RMSE_{Hagan} = \sqrt{\frac{1}{N}\sum(\sigma_{MKT} - \sigma_{Hagan}(p_k))^2} \quad (5.5)$$

T	*Fo*	*K*	*-100 bps*	*-50 bps*	*ATM*	*50 bps*	*100 bps*	*200 bps*
0.5	1.13	Market	180.37	90.45	75.99	74.61	75.93	79.37
		ANN	72.73	66.73	60.03	59.10	60.18	62.32
1	1.43	Market	96.22	67.98	60.61	59.71	60.6	63.22
		ANN	73.73	68.50	62.27	60.87	62.27	62.96
2	1.67	Market	74.9	59.55	53.6	51.27	50.48	50.52
		ANN	72.68	67.83	61.79	60.69	61.56	62.26
5	1.91	Market	62.06	50.34	44.39	41.09	39.22	37.54
		ANN	62.52	58.96	52.71	52.56	51.45	52.88
10	1.99	Market	55.79	44.46	38.52	35.07	33.00	30.97
		ANN	45.83	43.36	39.08	38.52	37.89	38.61
20	1.77	Market	75.84	48.19	38.83	34.11	31.52	29.26
		ANN	35.38	33.77	33.68	32.09	31.05	30.20

Table 5.3 – Comparison between ANN predicted implied volatility values using calibrated parameters and market quotes for 2Y tenor (USD)

Based solely on Hagan's approximation, the optimized parameters that fit the market quote was found (via Excel Solver) to be $\{\rho = -0.487, \xi = 1.00, v_o = 0.628\}$ and the corresponding $RMSE_{Hagan}$ and $RMSE_{ANN}$ would be 0.140 (14.0%) and 0.212 (21.2%) respectively. They are not too far off from our calibrated parameters using FDM data and ANN.

The calibration of SABR model parameters were further extended to market data of other tenors (4Y, 6Y, 8Y and 10Y) and the results were summarized in table 5.4 below. The results showed that the majority of calibrations are good quality with RMSE around 10-15%. Figure 5.8 and 5.9 respectively show the two dimensional and three dimensional (surface) plot of ANN predicted implied volatilities (calibrated) fitting with market quotes based on 2Y tenor data.

Tenor	ρ	ξ	v_o	$RMSE_{ANN}$	$RMSE_{Hagan}$
2Y	-0.433	0.677	0.537	0.217	0.187
4Y	-0.479	0.616	0.523	0.120	0.100
6Y	-0.531	0.628	0.524	0.120	0.0929
8Y	-0.587	0.634	0.526	0.125	0.118
10Y	-0.629	0.679	0.529	0.129	0.152

Table 5.4 – Calibrated parameters using ANN for different tenors and their RMSE values

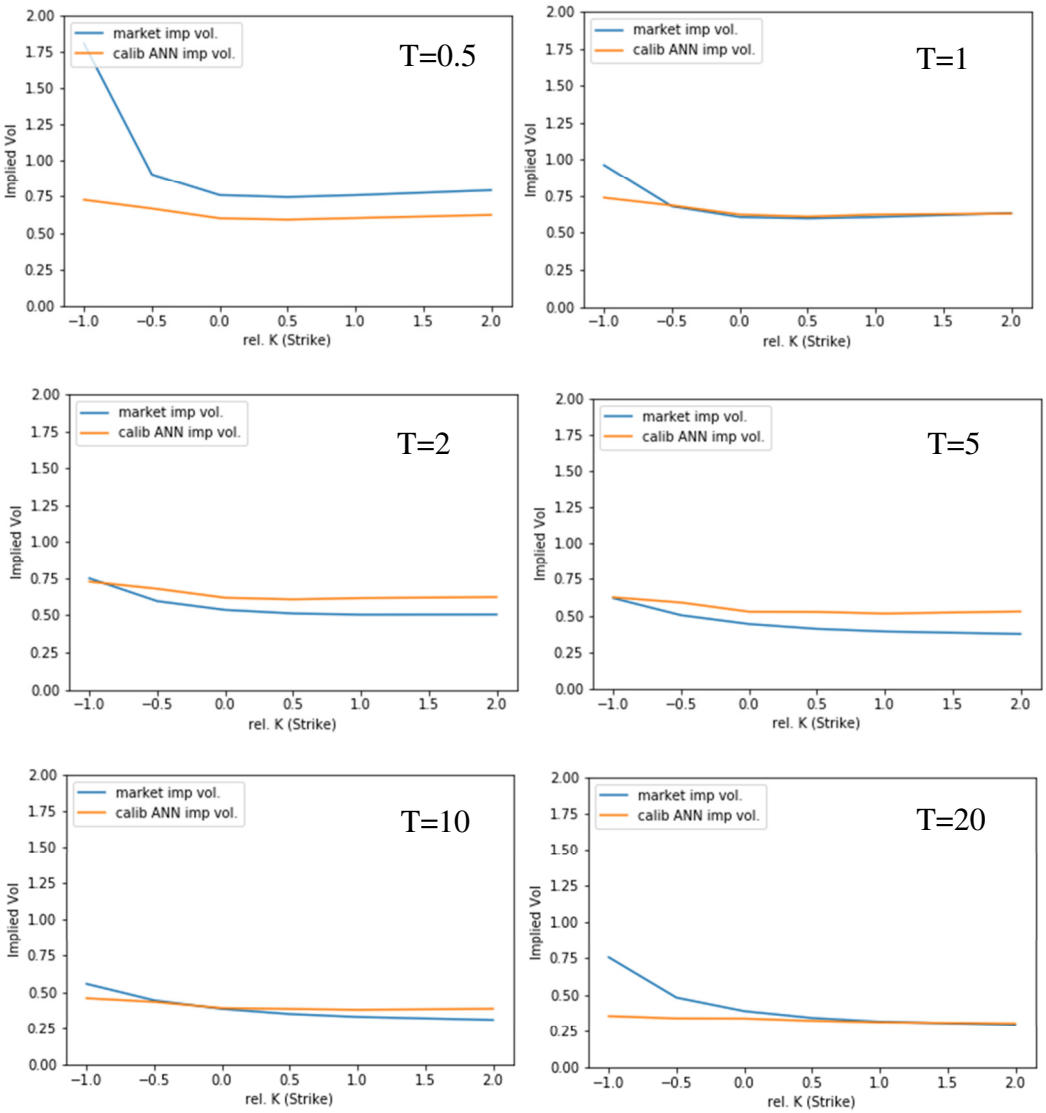


Figure 5.8 – ANN predicted implied volatilities (calibrated) with market quotes of different maturities for 2Y tenor (USD)

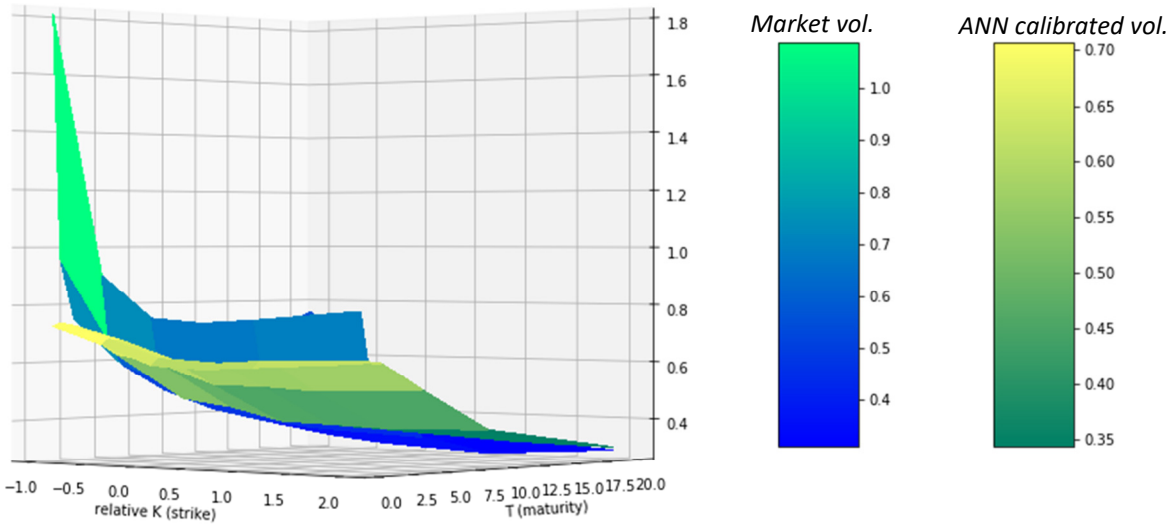


Figure 5.9 - ANN predicted implied volatilities surface fitting with market volatilities for 2Y tenor (USD)

6. Conclusion

This work proposes Artificial Neural Network (ANN) approach for calibration of SABR stochastic volatility model to market volatility surface which offer speed over traditional approach. The ANN is efficient in pricing or computing the implied volatilities by offloading the computational intensive part of generating price information offline and then trained with these data (forward-pass stage). Finite difference method (FDM) was applied to PDE to generate asset prices and then converted to implied volatilities using Dekker's root-finding method. The calibration process (backward-pass stage) utilized an inverse ANN with readily accessible gradient information of the original ANN to determine the optimal model parameters (Inverse Map approach). The neural network consist of two hidden layers with 160 nodes with ReLU (Rectified Linear Unit) activation function and k-fold cross validation was used to fully evaluate the predictive performance of the trained ANN. The inverse ANN predictive performance may be less than ideal depending greatly on presence of multi-to-one mapping in the original data and the output combinations which were later inverted to become inputs. The results of FDM implied volatilities are consistent with Hagan's approximation and the calibration results to Swaption market implied volatilities or surface are promising with RMSE of 10-20% attained.

To conclude, ANN can indeed be used to perform accurate price predictions and calibration well. Finally, this work opened up interesting avenues for future work such as applying calibration to Futures Options market data or other stochastic volatility Models.

Different synthetic data generation method other than uniform grid DFM such as double Integral method for more accurate option pricing could be explored too.

References:

Bachelier, L. (1900). Theorie de la speculation. *Annales scientifiques de l'Ecole normale superieure*, Vol. 17, 21-86.

Bayer, C. and Stemper, B., Horvath, B. Tomas, M. (2019). Deep calibration of rough stochastic volatility models. arVix: 1901.09647

Bishop, C. M. (2006). Pattern Recognition and Machine Learning, 1st Ed. US: Springer Publishing, Chapter 5

Black, F., Scholes, M. (1973). The Pricing of Options and Corporate Liabilities. *Journal of Political Economy.* Vol. 81 No. 3, 637–654

Black, F. (1976). The pricing of commodity contracts. *Journal of financial economics*, vol. 3 No. 1-2, 167-179

Brigo, D. and Mercurio, F. (2006). Interest Rate Models – Theory and Practice, 2nd ed. DE: Springer Publishing, Chapter 4.

Conlisk, J. (1973). Quick Stability Checks and Matrix Norms. *Economica New Series*, Vol. 40, No. 160., 402-409

Cox J. (1975). Notes on option pricing I: Constant elasticity of diffusions. *Unpublished Draft (Stanford University)*

Dekker, T. J. (1969). Finding a zero by means of successive linear interpolation in Constructive Aspects of the Fundamental Theorem of Algebra. In B. Dejon and P. Henrici, Constructive aspects of the fundamental theorem of algebra, US: Wiley-Interscience

Dimitroff, G., Kock (2011). Calibrating and completing the volatility cube in the SABR Model. *Fraunhofer-Institut fur Techno- und Wirtschaftsmathematik ITWM publication.*

Dimitroff, G., Roder D., Fries, C. P. (2018). Volatility Model Calibration With Convolutional Neural Networks. SSRN: 3252432

Dupire, B. (1994). Pricing with a smile. *Risk*, 7, 18-20

Faul. A. C. (2020). A Concise Introduction to Machine Learning, US: CRC Press, Chapter 9

Geron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Ed. CA: O'Reilly Mediam, Chapter 11

Goddard Consulting (2021). Option Pricing Using the Explicit Finite Difference Method. https://www.goddardconsulting.ca/option-pricing-finite-diff-explicit.html#ExplicitMatrixFormulation

Hagan, P.S. , Kumar, D. , Lesniewski, A. S. and Woodward, D. E. (2002). Managing Smile Risk, *Wilmott magazine*.

Harrison, J. M. and Pliska, S. R. (1981). Martingales and stochastic integrals in the theory of continuous trading. *Stochastic Processes and Their Applications,* Vol. 11 No. 3, 215-260.

Hernandez, A. (2016). Model calibration with Neural Networks . SSRN:2812140

Heston,S. L. (1993). A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *The Review of Financial Studies,* Vol. 6 No. 2, 327-343

Hornik, K. , Stinchcombe M., White H. (1989). Multilayer feedfoward networks are universal approximators. *Neural Networks*, Vol. 2 No. 5, 350-366.

Horvath, B., Muguruza, A. , Tomas M. (2019). Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance*, Vol. 21 No. 1, 11-27

Itkin, A. (2019). Deep learning calibration of option pricing models: some pitfalls and solutions. arXiv:1906.03507v1

Liu S. , Borovykh A. , Grzelak L. A., Oosterlee C. W. (2019b). A neural network-based framework for financial. *Journal of Mathematics in Industry*. 9:9

Liu S. , Oosterlee C. W., Bohte S. M. (2019a). Pricing options and computing implied volatilities using neural networks. *Risk*, Vol. 7 (1), 16

Mariani, M. C and Florescu I. (2020). Quantitative Finance, 1st Ed. US: John Wiley & Sons, Chapter 7

McGhee, W. A. (2018). An artificial neural network representation of the SABR stochastic volatility model. Preprint, SSRN:3288882

Merton, R (1973). Theory of Rational Option Pricing. *Bell Journal of Economics and Management Science*. Vol 4 No. 1, 141–18

Oosterlee, C. W, Grzelak, L. A. (2020). Mathematical Modeling and Computational in Finance, 1st ed. UK: World Scientific Publishing, Chapter 4.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, Vol. 22 No. 3, 400-407.

Ruf, J. Wang W. (2019). Neural Networks for Option Pricing and Hedging: A Literature Review. *Journal of Computational Finance,* Vol. 24 No. 1

A. Python Code Listing

```
# Import libraries

import numpy as np
import pandas as pd
import csv
import datetime
from scipy.stats import norm
from numpy import dot
import matplotlib.pyplot as plt
import random
import tensorflow as tf
from tensorflow.keras.layers import Dense
from tensorflow.keras import Input,Model
from keras.layers import Dropout
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error,
explained_variance_score
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Parameters definition

M=160         # Discretization max. count
N=100
dt=0.0040     # time step
Xo=-8         # lower bound
Yo=-10
dX=(2-Xo)/M      # Discretization steps
dY=(0-Yo)/N
beta=1.0
xi=0.1
rho=0.5
r=0.05
T=0.5   # Maturity
P=T/dt
K=1     # Strike
Klist=[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5]  #Strikes
xilist=[0.005,0.05,0.1,0.3,0.5]     #xi
rholist=[-0.7,-0.4,-0.1,0.1,0.4,0.7]  #rho


# Define matrix A elements functions for Explicit equation

def au_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.25*dt*rho*xi*np.exp(Yp)*np.exp((beta-1)*(Xp))/(dX*dY)
```

```
def am_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.5*dt*(xi**2)*(1/(dY**2)-1/(2*dY))

def ad_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return -0.25*dt*rho*xi*np.exp(Yp)*np.exp((beta-1)*(Xp))/(dX*dY)

def bu_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.5*dt*np.exp(2*Yp)*np.exp(2*(beta-1)*(Xp))*(1/(dX**2)-1/(2*dX))

def bm_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 1+dt*(-np.exp(2*Yp)*np.exp(2*(beta-1)*(Xp))/(dX**2)-(xi**2)/(dY**2))

def bd_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.5*dt*np.exp(2*Yp)*np.exp(2*(beta-1)*(Xp))*(1/(dX**2)+1/(2*dX))

def cu_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return -0.25*dt*rho*xi*np.exp(Yp)*np.exp((beta-1)*(Xp))/(dX*dY)

def cm_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.5*dt*(xi**2)*(1/(dY**2)+1/(2*dY))

def cd_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r):
    Xp=Xo+i*dX
    Yp=Yo+j*dY
    return 0.25*dt*rho*xi*np.exp(Yp)*np.exp((beta-1)*(Xp))/(dX*dY)

def ru_func(i,dt,xi,dY,r):
    return dt*0.5*(xi**2)*(1/(dY**2)-1/(2*dY))

def rm_func(i,dt,xi,dY,r):
    #return 1 #(1-r*dt)
    return 1-dt*(xi**2)/(dY**2)

def rd_func(i,dt,xi,dY,r):
     return dt*0.5*(xi**2)*(1/(dY**2)+1/(2*dY))

#=== Computation - Finite Difference Method (FDM) for particular T ===

for oo in range(len(rholist)):   # Loop rho
    rho=rholist[oo]
    for ooo in range(len(xilist):     # Loop xi
        xi=xilist[ooo]
        n=0
        A=np.zeros(((M+1)*(N+1),(M+1)*(N+1)))
        # Setup matrix A
        for j in range(N+1):
            for i in range(M+1):
                row=(M+1)*(N+1)-j*(M+1)-i-1

                if(j==0):
                    if (i==0):
                        A[row,(M+1)*(N+1)-1]=1
                    elif (i>0) & (i<M):
                        A[row,(M+1)*(N+1)-1-i]=rm_func(i,dt,xi,dY,r)
```

```
                        A[row,(M+1)*(N+1)-1-(i+1)]=ru_func(i,dt,xi,dY,r)
                        A[row,(M+1)*(N+1)-1-(i-1)]=rd_func(i,dt,xi,dY,r)
                    elif (i==M):
                        A[row,(M+1)*(N+1)-1-i]=1
                        A[row,(M+1)*(N+1)-1-i+1]=-1
                elif(j==N) & (i>0):
                    A[row,(M+1)*(N+1)-j*(M+1)-1-i]=1
                    A[row,(M+1)*(N+1)-j*(M+1)-1-i+1]=-1
                elif(i==0):
                    A[row,(M+1)*(N+1)-j*(M+1)-1]=1
                elif(i==M):
                    A[row,(M+1)*(N+1)-(j+1)*(M+1)+1-1]=1
                    A[row,(M+1)*(N+1)-(j+1)*(M+1)+2-1]=-1
                elif(i>0) & (i<M):
                    A[row,(M+1)*(N+1)-j*(M+1)-1-
i]=bm_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-j*(M+1)-1-
(i+1)]=bu_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-j*(M+1)-1-(i-
1)]=bd_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)

                    A[row,(M+1)*(N+1)-(j-1)*(M+1)-1-
i]=cm_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-(j-1)*(M+1)-1-
(i+1)]=cu_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-(j-1)*(M+1)-1-(i-
1)]=cd_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)

                    A[row,(M+1)*(N+1)-(j+1)*(M+1)-1-
i]=am_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-(j+1)*(M+1)-1-
(i+1)]=au_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)
                    A[row,(M+1)*(N+1)-(j+1)*(M+1)-1-(i-
1)]=ad_func(i,j,n,dt,beta,rho,xi,dY,dX,Yo,Xo,r)

        for o in range(len(Klist)):            # Loop K
            K=Klist[o]
            B=np.zeros(((M+1)*(N+1),1))
            n=0
             # Initial values t=0
            for j in range(N+1):
                for i in range(M+1):
                    row=(M+1)*(N+1)-j*(M+1)-i-1

                    B[row,0]=max(np.exp(Xo+i*dX)-K,0)
                    if(j==N) & (i>0):
                        B[row,0]=B[row+1,0]+dX*np.exp(Xo+i*dX)
                    if(i==M):
                        B[row,0]=B[row+1,0]+dX*np.exp(Xo+i*dX)

            while (n<int(P)):                  # Loop time
                # Matrix Multiplication
                C = dot(A,B)

                # Reinforce boundary conditions
                for j in range(N+1):
                    row1=(M+1)*(N+1)-j*(M+1)-1      # when i=0
                    row2=(M+1)*(N+1)-j*(M+1)-M-1    # when i=M
                    C[row1,0]=0
                    C[row2,0]=C[row2+1,0]+dX*np.exp(Xo+M*dX)
                for i in range(1,M+1):
                    row3=(M+1)*(N+1)-N*(M+1)-i-1    # when j=N
                    C[row3,0]=C[row3+1,0]+dX*np.exp(Xo+i*dX)

                B=C
                n+=1

             # Append/Write to CSV file - price data
```

```
            paraml=[rho,xi,T,K,beta,dY,dX,dt,str(datetime.datetime.now())]
            wdata=B.T.tolist()[0]
            wdata.extend(paraml)  # Combine B matrix and parameters info
            with open(r'C:\Users\Documents\Python files\SABRdata1.csv', 'a',
newline='') as file:
                writer = csv.writer(file)
                writer.writerow(wdata)


#=== Read price data, store in 2D-matrix, convert into implied vol. ===

# Read from CSV file - price data
ix=0;
dtr=[['']*((M+1)*(N+1)+9) for s in range(300)]
with open(r'C:\Users\Documents\Python files\SABRdata1.csv', 'r',) as file:
    reader = csv.reader(file)
    for rowr in reader:
        datar=rowr
        dtr[ix]=np.array(datar)
        ix+=1

# Store price data in 2-D matrix
fix_list=[0.5,1.0,1.5,2.0,2.5,3.0,3.5]    # Fo values
vix_list=[0.02,0.10,0.30,0.50]          # v0 values
N_comb=len(xilist)*len(rholist)         # number of combinations
datamat=np.zeros((N_comb,len(Klist),len(fix_list)*len(vix_list))) # data matrix


# Function return B matrix element position for a particular F0 & V0
def Findx(M,N,F0,dX,V0,dY):
    lnV0=np.log(V0)
    lnF0=np.log(F0)
    posV=np.round((lnV0-Yo)/dY)
    posF=np.round((lnF0-Xo)/dX)
    return (M+1)*(N+1)-1-int(posV)*(M+1)-int(posF)


cur_T=0.5
for ix in range(0,300):
    cur_rho=dtr[ix][(M+1)*(N+1)-1+1]
    cur_xi=dtr[ix][(M+1)*(N+1)-1+2]
    cur_K=dtr[ix][(M+1)*(N+1)-1+4]
    rho_idx = rholist.index(float(cur_rho))  # rho index position
    xi_idx= xilist.index(float(cur_xi)) # xi index position
    comb_idx=rho_idx*(len(xilist))+xi_idx  # combination index
    K_idx=Klist.index(float(cur_K))            # K index position

    fi=0
    for fix in fix_list:   # Loop Fo
        vi=0
        for vix in vix_list:            # Loop v0
            pos=Findx(M,N,fix,dX,vix,dY)  # element position of B matrix
            datamat[comb_idx][K_idx][fi*len(vix_list)+vi]=dtr[ix][pos]
            vi+=1
        fi+=1


# Function return call option price with vol. input - Black's equation
def f(sigma):
    d1=(np.log(F0/(K+1e-99))+0.5*(sigma**2)*T)/(sigma*np.sqrt(T))
    d2=(np.log(F0/(K+1e-99))-0.5*(sigma**2)*T)/(sigma*np.sqrt(T))
    return F0*norm.cdf(d1)-(K+1e-99)*norm.cdf(d2)


# Dekker's root-finding function
def Dekker(target,a,b,tol,f):
    fa=f(a)
    fb=f(b)
```

```
    if (abs(fa-target)<abs(fb-target)):  # Swap if function of b is bigger
        c=a
        d=b
        c0=d
    else:
        c=b
        d=a
        c0=d

    while (abs(c-d)>tol):  # loop while bigger than tolerance

        s= c-(f(c)-target)*(c-c0)/(f(c)-f(c0))  # Secant value
        m= 0.5*(c+d)                                  # Bisection value

        if (s>m) & (s<c):
            cn=s
        else:
            cn=m

        c0=c
        dn=d
        if ((f(d)-target)*(f(cn)-target)>0):
            dn=c

        if (abs(f(dn)-target)<abs(f(cn)-target)):
            xtemp=dn
            dn=cn
            cn=xtemp

        d=dn
        c=cn

    return c

# Convert price data in 2-D matrix to volatilities. Pick only Fo=3.0 & 3.5
T_=0.5
for rho_idx in range(len(rholist)):     #Loop rho
    rho_=rholist[rho_idx]
    for xi_idx in range(len(xilist)):    #Loop xi
        xi_=xilist[xi_idx]
        for K_idx in range(len(Klist)):        #Loop K
            K_=Klist[K_idx]
            for vi in range(len(vix_list)):
                v_=vix_list[vi]
                fi1=fix_list.index(3.0)  # Fo=3.0 index
                fi2=fix_list.index(3.5)  # Fo=3.5 index
                comb_idx=rho_idx*(len(xilist))+xi_idx  #comb. index
                price1=datamat[comb_idx][K_idx][fi1*len(vix_list)+vi]
                price2=datamat[comb_idx][K_idx][fi2*len(vix_list)+vi]
                T=T_
                K=K_
                F01=np.exp(Xo+dX*np.round((np.log(3.0)-Xo)/dX))
                F0=F01
                vol1=Dekker(price1,0.01,1,1e-6,f)     # compute implied vol
                F02=np.exp(Xo+dX*np.round((np.log(3.5)-Xo)/dX))
                F0=F02
                vol2=Dekker(price2,0.01,1,1e-6,f)     # compute implied vol

                 # Append/Write imp volatility to CSV file
                param2=[rho_,xi_,T_,K_,v_,beta]
                x=[price1,price2,vol1,vol2,F01,F02]
                x.extend(param2)

                with open(r'C:\Users\Documents\Python files\SABRdata_pro1.csv', 'a',
newline='') as file2:
                    writer = csv.writer(file2)
                    writer.writerow(x)
```

```
#=== Read All Implied volatility data & prepare data as Data Frame ===

# Read imp volatility from CSV files (containing T=0.5,1,2,5,10,20)
idx=0;
inpdat=[]  # Hold Implied Vol. for Fo=3.0 & 3.5
with open(r'C:\Users\Documents\Python files\SABRdata_pro1.csv', 'r',) as filedat:#
T=0.5 imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))
with open(r'C:\Users\Documents\Python files\SABRdata_pro2.csv', 'r',) as filedat:#
T=1 imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))
with open(r'C:\Users\Documents\Python files\SABRdata_pro3.csv', 'r',) as filedat:#
T=2 imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))
with open(r'C:\Users\Documents\Python files\SABRdata_pro4.csv', 'r',) as filedat:#
T=5    imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))
with open(r'C:\Users\Documents\Python files\SABRdata_pro5.csv', 'r',) as filedat:#
T=10 imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))
with open(r'C:\Users\Documents\Python files\SABRdata_pro6.csv', 'r',) as filedat:#
T=20 imp. vol. data
    reader = csv.reader(filedat)
    for rowd in reader:
        datar=rowd
        inpdat.append(np.array(datar))


df=pd.DataFrame(columns=['rho','xi','T','v0','F0','vol1','vol2','vol3','vol4','vol5
','vol6','vol7','vol8','vol9'])  # setup Data Frame

fix_list=[0.5,1.0,1.5,2.0,2.5,3.0,3.5]  # Fo values
vix_list=[0.02,0.10,0.30,0.50]          # v0 values
Tlist=[0.5,1.0,2.0,5.0,10.0,20.0]       # T values

for Tidx in range(len(Tlist)):  # Loop T
    off_T=Tidx*len(rholist)*len(xilist)*len(vix_list)*len(Klist) #offset

    for rhoidx in range(len(rholist)): # Loop rho
        off_rho=rhoidx*len(xilist)*len(vix_list)*len(Klist)  #offset

        for xiidx in range(len(xilist)): # Loop xi
            off_xi=xiidx*len(vix_list)*len(Klist)     #offset

            for v0idx in range(len(vix_list)):
                off_base=off_T+off_rho+off_xi+v0idx
                rho_1=float(inpdat[off_base][6])  # extract the parameters
                xi_1=float(inpdat[off_base][7])
                T_1=float(inpdat[off_base][8])
                v0_1=float(inpdat[off_base][10])

                F01_v=float(inpdat[off_base][4])
```

```
                F02_v=float(inpdat[off_base][5])
                vol_1=np.zeros(len(Klist)-1)
                vol_2=np.zeros(len(Klist)-1)
                indx=0
                for Kidx in range(1,len(Klist)): # conglomerate 9 imp vols
                    nix1=off_base+(Kidx-1)*len(vix_list)  #index for Fo=3.0
                    nix2=off_base+Kidx*len(vix_list)    #index for Fo=3.5
                    vol_1[indx]=float(inpdat[nix1][2]) # Imp vol for Fo=3.0
                    vol_2[indx]=float(inpdat[nix2][3]) # Imp vol for Fo=3.5
                    indx+=1

                 # Append to Data Frame
                 df=df.append({'rho':rho_1,'xi':xi_1,'T':T_1,'v0':v0_1,'F0':F01_v,'
                 vol1':vol_1[0],'vol2':vol_1[1],'vol3':vol_1[2],'vol4':vol_1[3],'vo
                 l5':vol_1[4],'vol6':vol_1[5],'vol7':vol_1[6],'vol8':vol_1[7],'vol9
                 ':vol_1[8]},ignore_index=True)

                 df=df.append({'rho':rho_1,'xi':xi_1,'T':T_1,'v0':v0_1,'F0':F02_v,'
                 vol1':vol_2[0],'vol2':vol_2[1],'vol3':vol_2[2],'vol4':vol_2[3],'vo
                 l5':vol_2[4],'vol6':vol_2[5],'vol7':vol_2[6],'vol8':vol_2[7],'vol9
                 ':vol_2[8]},ignore_index=True)


#=== Plot DFM implied vol. with Hagan's implied vol ===

# Function: Hagan's approximation for implied volatilities
def calc_impvol(rho_,xi_,V0_,T_,F0_,K_,beta_):
    q=(F0_*K_)**((1-beta_)/2)*(1+((1-beta_)**2)*(np.log(F0_/K_)**2)/24 + (((1-
beta_)**4)/1920)*(np.log(F0_/K_)**4))
    z=(xi_/V0_)*((F0_*K_)**((1-beta_)/2)*np.log(F0_/K_))
    X_z=np.log((np.sqrt(1-2*rho_*z+z**2)-rho_+z)/(1-rho_))
    vol_calc= (V0_/q)*((z+1e-99)/(X_z+1e-99))*( 1+( ((1-
beta_)**2)/(24*((F0_*K_)**(1-beta_))) + (rho_*xi_*beta_*V0_)/(4*(F0_*K_)**((1-
beta_)/2)) + ((2-3*rho_**2)*xi_**2)/24 )*T_)
    return vol_calc


i=476 # a particular row
rho_p=float(df[['rho']].loc[i]) # Get parameter values
xi_p=float(df[['xi']].loc[i])
T_p=float(df[['T']].loc[i])
v0_p=float(df[['v0']].loc[i])
F0_p=float(df[['F0']].loc[i])
beta_p=1
yp=[df[['vol1']].loc[i],df[['vol2']].loc[i],df[['vol3']].loc[i],df[['vol4']].loc[i]
,df[['vol5']].loc[i],df[['vol6']].loc[i],df[['vol7']].loc[i],df[['vol8']].loc[i],df
[['vol9']].loc[i]]  # DFM plot values
yhag=[]   # Hold Hagan's plot values
for n in range(len(Klist)-1):  # Choose the correct Strikes for Fo=3.0 or Fo=3.5
    if (i%2)==0:  # Fo=3.0
        K_p=Klist[n]
    else:
        K_p=Klist[n+1]
    xpp=calc_impvol(rho_p,xi_p,v0_p,T_p,F0_p,K_p+1e-199,beta_p) # Hagan's imp vol.
    yhag.append(xpp)

if (i%2)==0:  # Fo=3.0
      plt.plot(Klist[0:9],yp[0:9], label='FD implied vol')
      plt.plot(Klist[0:9],yhag[0:9], label='Hagan implied vol')
else:
      plt.plot(Klist[1:10],yp[0:9], label='FD implied vol')
      plt.plot(Klist[1:10],yhag[0:9], label='Hagan implied vol')

plt.legend(loc="upper left")
plt.xlabel("K (Strike)")
plt.ylabel("Implied Vol")
```

```
#=== Forward Artificial Neural Network (ANN) training ===

X1=df[['rho','xi','T','v0','F0']].values  #inputs & outputs
y=df[['vol1','vol2','vol3','vol4','vol5','vol6','vol7','vol8','vol9']].values

# Split data sets to 80% training data 20% testing data
X_train,X_test,y_train,y_test =
train_test_split(X1,y,test_size=0.2,random_state=101)

Y1_train=y_train[:,[0]]    #individual outputs of training data
Y2_train=y_train[:,[1]]
Y3_train=y_train[:,[2]]
Y4_train=y_train[:,[3]]
Y5_train=y_train[:,[4]]
Y6_train=y_train[:,[5]]
Y7_train=y_train[:,[6]]
Y8_train=y_train[:,[7]]
Y9_train=y_train[:,[8]]

Y1_test=y_test[:,[0]]      #individual outputs of testing data
Y2_test=y_test[:,[1]]
Y3_test=y_test[:,[2]]
Y4_test=y_test[:,[3]]
Y5_test=y_test[:,[4]]
Y6_test=y_test[:,[5]]
Y7_test=y_test[:,[6]]
Y8_test=y_test[:,[7]]
Y9_test=y_test[:,[8]]

# Configure TensorFlow Keras
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
)

# Setup the Forward ANN
inputs=Input(shape=(5,),name='input')   # 5 inputs
x=Dense(160,activation='relu',name='160L1')(inputs)
x=Dropout(0.2)(x)                       # with dropout rate
x=Dense(160,activation='relu',name='160L2')(x)
output1=Dense(1,name='vol1_out')(x)     # 9 outputs
output2=Dense(1,name='vol2_out')(x)
output3=Dense(1,name='vol3_out')(x)
output4=Dense(1,name='vol4_out')(x)
output5=Dense(1,name='vol5_out')(x)
output6=Dense(1,name='vol6_out')(x)
output7=Dense(1,name='vol7_out')(x)
output8=Dense(1,name='vol8_out')(x)
output9=Dense(1,name='vol9_out')(x)
model = Model(inputs=inputs,outputs=[output1,output2,output3,output4,output5,
output6,output7,output8,output9])
model.compile(loss={'vol1_out':'mean_squared_error',
                    'vol2_out': 'mean_squared_error',
                    'vol3_out': 'mean_squared_error',
                    'vol4_out': 'mean_squared_error',
                    'vol5_out': 'mean_squared_error',
                    'vol6_out': 'mean_squared_error',
                    'vol7_out': 'mean_squared_error',
                    'vol8_out': 'mean_squared_error',
                    'vol9_out': 'mean_squared_error'
                    }, optimizer='adam')       # MSE Lose function & Adam optimizer
# Training
```

```
history=model.fit(X_train,{'vol1_out':Y1_train,'vol2_out':Y2_train,'vol3_out':Y3_tr
ain,'vol4_out':Y4_train,'vol5_out':Y5_train,'vol6_out':Y6_train,'vol7_out':Y7_train
,'vol8_out':Y8_train,'vol9_out':Y9_train},batch_size=32,epochs=250)

loss_df = pd.DataFrame(history.history)  # Training history Loss
loss_df['vol1_out_loss'].plot() # Plot individual output loss
loss_df['vol2_out_loss'].plot()
loss_df['vol3_out_loss'].plot()
loss_df['vol4_out_loss'].plot()
loss_df['vol5_out_loss'].plot()
loss_df['vol6_out_loss'].plot()
loss_df['vol7_out_loss'].plot()
loss_df['vol8_out_loss'].plot()
loss_df['vol9_out_loss'].plot()
loss_df['loss'].plot()  # Plot total output loss
plt.xlabel("Epoch")
plt.ylabel("MSE loss")

test_predictions=model.predict(X_test) # Predict testing data

explained_variance_score(Y1_test,test_predictions[0]) # variance regression score
explained_variance_score(Y2_test,test_predictions[1])
explained_variance_score(Y3_test,test_predictions[2])
explained_variance_score(Y4_test,test_predictions[3])
explained_variance_score(Y5_test,test_predictions[4])
explained_variance_score(Y6_test,test_predictions[5])
explained_variance_score(Y7_test,test_predictions[6])
explained_variance_score(Y8_test,test_predictions[7])
explained_variance_score(Y9_test,test_predictions[8])

%matplotlib qt   # Configure separate plot window
plt.plot(Y1_test,label='test output')  # Plot each predicted output with test data
plt.plot(test_predictions[0],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y2_test,label='test output')
plt.plot(test_predictions[1],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y3_test,label='test output')
plt.plot(test_predictions[2],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y4_test,label='test output')
plt.plot(test_predictions[3],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y5_test,label='test output')
plt.plot(test_predictions[4],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y6_test,label='test output')
plt.plot(test_predictions[5],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y7_test,label='test output')
plt.plot(test_predictions[6],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y8_test,label='test output')
plt.plot(test_predictions[7],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Y9_test,label='test output')
plt.plot(test_predictions[8],label='predicted output')
plt.legend(loc="upper left")


#=== K-fold cross validation for forward ANN ===

allL=[i for i in range(len(X1))] # Hold indices
allL2=allL
ncv=5
partL=[[]*1 for i in range(ncv)] # To store part element indices
for j in range((ncv-1)):         # Generate 5 parts through random sampling
    partL[j]=random.sample(allL2,int(len(X1)/5))
```

```
    allL2=[i for i in allL2 if i not in partL[j]]
partL[ncv-1]=allL2  # last part
random.shuffle(partL[ncv-1])

cvscores=[]  # Store cross validation Scores
for i in range(ncv):
    test=partL[i]          # 1 of the 5 parts become testing data
    train=[id for id in allL if id not in test]  # remaining becomes training data
    random.shuffle(train)
    inputs=Input(shape=(5,),name='input')
    x=Dense(160,activation='relu',name='160L1')(inputs)
    x=Dropout(0.2)(x)
    x=Dense(160,activation='relu',name='160L2')(x)
    output1=Dense(1,name='vol1_out')(x)
    output2=Dense(1,name='vol2_out')(x)
    output3=Dense(1,name='vol3_out')(x)
    output4=Dense(1,name='vol4_out')(x)
    output5=Dense(1,name='vol5_out')(x)
    output6=Dense(1,name='vol6_out')(x)
    output7=Dense(1,name='vol7_out')(x)
    output8=Dense(1,name='vol8_out')(x)
    output9=Dense(1,name='vol9_out')(x)
    model2 = Model(inputs=inputs,outputs=[output1,output2,output3,output4,output5,
    output6,output7,output8,output9])
    model2.compile(loss={'vol1_out':'mean_squared_error',
                         'vol2_out': 'mean_squared_error',
                         'vol3_out': 'mean_squared_error',
                         'vol4_out': 'mean_squared_error',
                         'vol5_out': 'mean_squared_error',
                         'vol6_out': 'mean_squared_error',
                         'vol7_out': 'mean_squared_error',
                         'vol8_out': 'mean_squared_error',
                         'vol9_out': 'mean_squared_error'
                         }, optimizer='adam')

    history2=model2.fit(X1[train],{'vol1_out':y[train],'vol2_out':y[train],'vol3_ou
    t':y[train],'vol4_out':y[train],'vol5_out':y[train],'vol6_out':y[train],'vol7_o
    ut':y[train],'vol8_out':y[train],'vol9_out':y[train]},batch_size=32,epochs=250,
    verbose=0)

    scores=model2.evaluate(X1[test],{'vol1_out':y[test],'vol2_out':y[test],'vol3_ou
    t':y[test],'vol4_out':y[test],'vol5_out':y[test],'vol6_out':y[test],'vol7_out':
    y[test],'vol8_out':y[test],'vol9_out':y[test]},verbose=0)
    print("%s: %.2f%%" % (model2.metrics_names[0],scores[0]*100))

    cvscores.append(scores[0]*100)  # Append scores

print("%.2f%% (+/- %.2f%%)"% (np.mean(cvscores),np.std(cvscores))) # print scores

loss_df2 = pd.DataFrame(history2.history)  # Training history Loss
loss_df2['loss'].plot()                 # Plot total output loss

test_predictions_2=model2.predict(X1[test]) # Predict testing data
explained_variance_score(y[test,[1]],test_predictions_2[1]) # var regression score

%matplotlib qt   # separate plot window
plt.plot(y[test,[1]],label='test output')  # Plot predicted output with test data
plt.plot(test_predictions_2[1],label='predicted output')
plt.legend(loc="upper left")


#=== Inverse Artificial Neural Network (ANN) training ===

Xinv1=df[['vol3','vol4','vol5','vol6','vol7', 'vol9','T','F0']].values #inputs
yinv=df[['rho','xi','v0']].values  #outputs

# Split data sets to 80% training data 20% testing data
```

```
Xinv_train,Xinv_test,yinv_train,yinv_test =
train_test_split(Xinv1,yinv,test_size=0.2,random_state=101)

Yinv1_train=yinv_train[:,[0]]   #individual outputs of training data
Yinv2_train=yinv_train[:,[1]]
Yinv3_train=yinv_train[:,[2]]

Yinv1_test=yinv_test[:,[0]]     #individual outputs of testing data
Yinv2_test=yinv_test[:,[1]]
Yinv3_test=yinv_test[:,[2]]

# Setup the Inverse ANN
inputsinv=Input(shape=(8,),name='inputinv')    # 8 inputs
xinv=Dense(160,activation='relu',name='160L1')(inputsinv)
xinv = Dropout(0.2)(xinv)                       # with dropout rate
xinv=Dense(160,activation='relu',name='160L2')(xinv)
outputinv1=Dense(1,name='rho_out')(xinv)       # 3 outputs
outputinv2=Dense(1,name='xi_out')(xinv)
outputinv3=Dense(1,name='v0_out')(xinv)
modelinv = Model(inputs=inputsinv,outputs=[outputinv1,outputinv2,outputinv3])
modelinv.compile(loss={'rho_out':'mean_squared_error',
                       'xi_out': 'mean_squared_error',
                       'v0_out':'mean_squared_error'},
                   optimizer='adam')  # MSE Lose function & Adam optimizer
# Training
historyinv=modelinv.fit(Xinv_train,{'rho_out':Yinv1_train,'xi_out':Yinv2_train,'v0_
out':Yinv3_train},batch_size=32, epochs=250)


lossinv_df = pd.DataFrame(historyinv.history)  # Training history Loss
lossinv_df['rho_out_loss'].plot()  # Plot individual output loss
lossinv_df['xi_out_loss'].plot()
lossinv_df['v0_out_loss'].plot()
lossinv_df['loss'].plot()    # Plot total output loss
plt.xlabel("Epoch")
plt.ylabel("MSE loss")

test_predictions_inv=modelinv.predict(Xinv_test)  # Predict testing data

%matplotlib qt   # Separate Plot Window
plt.plot(Yinv1_test,label='test output')   # Plot predicted output with test data
plt.plot(test_predictions_inv[0],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Yinv2_test,label='test output')
plt.plot(test_predictions_inv[1],label='predicted output')
plt.legend(loc="upper left")
plt.plot(Yinv3_test,label='test output')
plt.plot(test_predictions_inv[2],label='predicted output')
plt.legend(loc="upper left")

explained_variance_score(Yinv1_test,test_predictions_inv[0]) # Variance regression
score
explained_variance_score(Yinv2_test,test_predictions_inv[1])
explained_variance_score(Yinv3_test,test_predictions_inv[2])


#=== K-fold cross validation for inverse ANN ===

allL=[i for i in range(len(Xinv1))]     # Hold indices
allL2=allL
ncv=5
partL=[[]*1 for i in range(ncv)]        # To store part element indices
for j in range((ncv-1)):                # Generate 5 parts through random sampling
    partL[j]=random.sample(allL2,int(len(Xinv1)/5))
    allL2=[i for i in allL2 if i not in partL[j]]
partL[ncv-1]=allL2  # last part
random.shuffle(partL[ncv-1])
```

```
cvscores=[]  # Store cross validation Scores
for i in range(ncv):
    test=partL[i]          # 1 of the 5 parts become testing data
    train=[id for id in allL if id not in test]  # remaining becomes training data
    random.shuffle(train)
    inputsinv2=Input(shape=(8,),name='inputinv')
    xinv2=Dense(160,activation='relu',name='160L1')(inputsinv2)
    xinv2 = Dropout(0.2)(xinv2)
    xinv2=Dense(160,activation='relu',name='160L2')(xinv2)
    outputinv1b=Dense(1,name='rho_out')(xinv2)
    outputinv2b=Dense(1,name='xi_out')(xinv2)
    outputinv3b=Dense(1,name='v0_out')(xinv2)
    modelinv2 = Model(inputs=inputsinv2,outputs=[outputinv1b,outputinv2b,
outputinv3b])
    modelinv2.compile(loss={'rho_out':'mean_squared_error',
                            'xi_out': 'mean_squared_error',
                            'v0_out':'mean_squared_error'},
                      optimizer='adam')

    historyinv2=modelinv2.fit(Xinv1[train],{'rho_out':yinv[train,[0]],'xi_out':yinv
[train,[1]],'v0_out':yinv[train,[2]]},verbose=0,epochs=250)

    scores=modelinv2.evaluate(Xinv1[test],{'rho_out':yinv[test,[0]],'xi_out':yinv[t
    est,[1]],'v0_out':yinv[test,[2]]},verbose=0)
    print("%s: %.2f%%" % (modelinv2.metrics_names[0],scores[0]*100))

    cvscores.append(scores[0]*100)  # Append Scores

print("%.2f%% (+/- %.2f%%)"% (np.mean(cvscores),np.std(cvscores))) # print scores

lossinv_df2 = pd.DataFrame(historyinv2.history)  # Training history Loss
lossinv_df2['loss'].plot()                    # Plot total output loss

test_predictions_inv2=modelinv2.predict(Xinv1[test])  # Predict testing data
explained_variance_score(yinv[test,[1]],test_predictions_inv2[1]) #var regression
score

%matplotlib qt   # Separate Plot Window
plt.plot(yinv[test,[1]],label='test output') # Plot predicted output with test data
plt.plot(test_predictions_inv2[1],label='predicted output')
plt.legend(loc="upper left")


#=== Input Market Data ===

# Read market imp volatility from CSV file (of different tenors 2Y,4Y,...)
mktdatraw=[]  # Hold raw market data
with open(r'C:\Users\Documents\Python files\swap_USD_tenor2yr.csv', 'r',) as
fileinp:
    reader = csv.reader(fileinp)
    for rowm in reader:
        datam=rowm
        mktdatraw.append(np.array(datam))

mktdat=[['']*1 for s in range(len(mktdatraw)-1)] # Hold market data without header
for n in range(1,len(mktdatraw)):
    mktdat[n-1]=mktdatraw[n]
    for m in range(len(mktdatraw[0])-2):
        mktdat[n-1][m]=float(mktdat[n-1][m])/100     # convert % to decimal

mktinp=[[]*1 for s in range(len(mktdat))]      # Hold array of list of market data
for n in range(len(mktdat)):            # Loop each T
    mktinp[n]=mktdat[n].tolist()               # Convert into list
    for m in range(len(mktinp[0])):
        mktinp[n][m]=float(mktinp[n][m])

pk1=[]   # store inverse ANN inferred rho
pk2=[]   # store inverse ANN inferred xi
```

```
pk3=[]   # store inverse ANN inferred v0
F0n=[]
for n in range(len(mktinp)):            # Loop each T
    mktpred=modelinv.predict([mktinp[n]])  # Inverse ANN predictions
    pk1.append(float(mktpred[0][0]))    # Append rho
    pk2.append(float(mktpred[1][0]))    # Append xi
    pk3.append(float(mktpred[2][0]))    # Append v0
    F0n.append(mktinp[n][len(mktinp[n])-1])    # Append F0


#=== Calibration (Find Optimized Parameters) ===

# Prepare list of inverse ANN parameters for different T
input_data=pd.DataFrame([[pk1[0],pk2[0],0.5,pk3[0],F0n[0]],[pk1[1],pk2[1],1,pk3[1],
F0n[1]],[pk1[2],pk2[2],2,pk3[2],F0n[2]],[pk1[3],pk2[3],5,pk3[3],F0n[3]],[pk1[4],pk2
[4],10,pk3[4],F0n[4]],[pk1[5],pk2[5],20,pk3[5],F0n[5]]])

x_tensor = tf.convert_to_tensor(input_data, dtype=tf.float32) #conv. to tensor obj.

# Configure TensorFlow Automatic Differentiation for gradient computation
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x_tensor)
    output = model(x_tensor)

grad=[]  # Hold gradients
grad.append(tape.gradient(output[2], x_tensor))  # gradient of f-ANN output 2
grad.append(tape.gradient(output[3], x_tensor))  # gradient of f-ANN output 3
grad.append(tape.gradient(output[4], x_tensor))  # gradient of f-ANN output 4
grad.append(tape.gradient(output[5], x_tensor))  # gradient of f-ANN output 5
grad.append(tape.gradient(output[6], x_tensor))  # gradient of f-ANN output 6
grad.append(tape.gradient(output[8], x_tensor))  # gradient of f-ANN output 8

# rho
prod_sum1=0
weight_sum1=0
weight1=0
for i in range(len(mktinp)):  # no. of T
    for j in range(6):   # no. of impvol of different K
        weight1=(1/(6*len(mktinp)))*abs(grad[j][i].numpy()[0])
        prod_sum1+=weight1*pk1[i]
        weight_sum1+=weight1
rho_opt=prod_sum1/weight_sum1     # Calibrated/Optimum Rho

# xi
prod_sum2=0
weight_sum2=0
weight2=0
for i in range(len(mktinp)):  # no. of T
    for j in range(6):   # no. of impvol of different K
        weight2=(1/(6* len(mktinp)))*abs(grad[j][i].numpy()[1])
        prod_sum2+=weight2*pk2[i]
        weight_sum2+=weight2
xi_opt=prod_sum2/weight_sum2      # Calibrated/Optimum Xi

# V0
prod_sum3=0
weight_sum3=0
weight3=0
for i in range(len(mktinp)):  # no. of T
    for j in range(6):   # no. of impvol of different K
        weight3=(1/(6*len(mktinp)))*abs(grad[j][i].numpy()[3])
        prod_sum3+=weight3*pk3[i]
        weight_sum3+=weight3
v0_opt=prod_sum3/weight_sum3      # Calibrated/Optimum V0


#=== Calibration (Check Performance) ===
```

```
# Calculate RMSE with market data based on Hagan's approximation
rho_=rho_opt
xi_=xi_opt
v0_=v0_opt
#rho_=-0.4874  # Excel Solver's Hagan's approx. optimized parameters
#xi_=1
#v0_=0.6279
pdif_sum=0
voldat=[[]*1 for s in range(len(mktinp))]
perdif=[[]*1 for s in range(len(mktinp))]
results=[[]*1 for s in range(len(mktinp))]  # Holds Hagan's implied vol.
off_list=mktdatraw[0][0:-2]
for n in range(len(mktinp)):      # Loop each T
    Tn_=mktinp[n][len(mktinp[n])-2]
    F0n_=mktinp[n][len(mktinp[n])-1]
    for m in range(len(off_list)):
        Km_=F0n_+0.01*float(off_list[m])  # Relative Strikes
        impvol=calc_impvol(rho_,xi_,v0_,Tn_,F0n_,Km_,beta)  # Hagan's imp. vol.
        results[n].append(impvol)
        print("%.4f vs %.4f"%(impvol,mktinp[n][m]))
        pdif=(impvol-mktinp[n][m])**2   # Square Error
        pdif_sum+=pdif                  # Sum
        voldat[n].append(impvol)
        perdif[n].append(pdif)
    print('\n')
print(np.sqrt(pdif_sum/(6*len(mktinp))))  # Print RMSE

# Eg. mktinp[0] = [1.8037, 0.9045, 0.7598, 0.7461, 0.7593, 0.7937, 0.5, 1.13]

# Calculate RMSE with market data based on ANN prediction
rho_=rho_opt
xi_=xi_opt
v0_=v0_opt
#rho_=-0.4874 # Excel Solver's Hagan's approx. optimized parameters
#xi_=1
#v0_=0.6279
pdif_sum2=0
perdif2=[[]*1 for s in range(len(mktinp))]
results2=[[]*1 for s in range(len(mktinp))]  # Holds forward ANN implied vol.
for n in range(len(mktinp)):                   # Loop each T
    Tn_=mktinp[n][len(mktinp[n])-2]
    F0n_=mktinp[n][len(mktinp[n])-1]
    opt_inp=pd.DataFrame([[rho_,xi_,Tn_,v0_,F0n_]])  # Optimized param inputs
    vol_pred=model.predict(opt_inp)     # Forward ANN prediction based on param
    results2[n].append(float(vol_pred[2]))
    print("%.4f vs %.4f"%(float(vol_pred[2]),mktinp[n][0]))
    pdif2=(float(vol_pred[2])-mktinp[n][0])**2       # Square Error
    perdif2[n].append(pdif2)
    pdif_sum2+=pdif2                                 # sum
    results2[n].append(float(vol_pred[3]))
    print("%.4f vs %.4f"%(float(vol_pred[3]),mktinp[n][1]))
    pdif2=(float(vol_pred[3])-mktinp[n][1])**2       # Square Error
    perdif2[n].append(pdif2)
    pdif_sum2+=pdif2                                 # sum
    results2[n].append(float(vol_pred[4]))
    print("%.4f vs %.4f"%(float(vol_pred[4]),mktinp[n][2]))
    pdif2=(float(vol_pred[4])-mktinp[n][2])**2       # Square Error
    perdif2[n].append(pdif2)
    pdif_sum2+=pdif2                                 # sum
    results2[n].append(float(vol_pred[5]))
    print("%.4f vs %.4f"%(float(vol_pred[5]),mktinp[n][3]))
    pdif2=(float(vol_pred[5])-mktinp[n][3])**2       # Square Error
    perdif2[n].append(pdif2)
    pdif_sum2+=pdif2                                 # sum
    results2[n].append(float(vol_pred[6]))
    print("%.4f vs %.4f"%(float(vol_pred[6]),mktinp[n][4]))
    pdif2=(float(vol_pred[6])-mktinp[n][4])**2       # Square Error
    perdif2[n].append(pdif2)
```

```
    pdif_sum2+=pdif2                                    # sum
    results2[n].append(float(vol_pred[8]))
    print("%.4f vs %.4f"%(float(vol_pred[8]),mktinp[n][5]))
    pdif2=(float(vol_pred[8])-mktinp[n][5])**2        # Square Error
    perdif2[n].append(pdif2)
    pdif_sum2+=pdif2                                    # sum
    print('\n')
print(np.sqrt(pdif_sum2/(6*len(mktinp))))      # Print RMSE

#=== Plot 3D surface ===

off_list2=[]  # Holds Relative strikes w.r.t ATM
for m in range(len(mktdatraw[0])-2):
        off_list2.append(float(mktdatraw[0][m])/100)

mktq=[[]*1 for s in range(len(mktinp))] # Holds Market quotes (exclude T, Fo)
for n in range(len(mktinp)):
    mktq[n]=mktinp[n][0:-2]

X=off_list2
Y=Tlist
X,Y = np.meshgrid(X,Y)
Z1=np.array(mktq)
Z2=np.array(results2)
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})  # Separate 3D plot
ax.view_init(0, -50)   # Adjust 3D view angle
surf1=ax.plot_surface(X,Y,Z1,cmap=cm.winter,linewidth=0,antialiased=False,label='ma
rket imp vol')
surf2=ax.plot_surface(X,Y,Z2,cmap=cm.summer,linewidth=0,antialiased=False,label='AN
N calib imp vol')
fig.colorbar(surf2,shrink=0.5,aspect=5) # plot surface
fig.colorbar(surf1,shrink=0.5,aspect=5) # plot surface
ax.set_xlabel('relative K (strike)')
ax.set_ylabel('T (maturity)')
ax.legend()
```