SABR volatility surfaces with TensorTrains

Industry Project Report
submitted to

And

MFE293, Haas School of Business

Advisor:
Peter Decrem, Citi
Jacob Gallice, MFE

Students:
Ayush Agarwal
Bhuvnesh Khandelwal

Contents

1 Introduction **2**
1.1 Abstract 2
1.2 Methodology 2
1.3 SABR Volatility Model 3
1.4 Hagan Approximation 3

2 Data Generation **5**
2.1 Finite-Difference Method 5
2.2 Symmetry Transformation 6
2.3 Range Transformation 6
2.3.1 Strike Range Transformation 7
2.3.2 Volvol Range Transformation 7
2.4 Target Function 8
2.5 Parameter Ranges 9
2.5.1 Final Parameter Set 9
2.6 Dataset Statistics 10

3 Tensor Neural Networks **11**
3.1 Tensor Contraction 11
3.2 Tensorized fully connected layer 12
3.2.1 Block Term Tucker (BlockTT) 12
3.2.2 Tensor Train (TT) 12

4 Modeling **14**
4.1 Network 14
4.2 Parameters 14
4.3 Schedulers 15
4.3.1 Learning Rate Schedulers 15
4.3.2 Batch Scheduling 15
4.4 Error Stratification 16
4.5 Splines/Polynomial Fitting 17

5 Results and Plots **17**

1 Introduction

1.1 Abstract

One of the fundamental aspects of risk modeling involves pricing options and measuring their implied volatility in real time. Success in this area provides a better real time understanding of the financial markets, assists with risk management practices, and can even be used to generate trade ideas. From an implementation point of view, valuation of option prices and volatility revolves around solving high dimensional problems quickly, efficiently, and within a desired range of accuracy. Deep learning is often seen as one of the ways of succeeding in this endeavour. Below is a discussion on the development, implementation, and results of some deep learning applications: a Tensor Neural Network approach as well as a discussion on potential applications of novel techniques such as Diffusion models. The frameworks developed seek to achieve a robust calibration of the SABR volatility model and its underlying parameters with special adherence towards accuracy and speed. We also explore some standardized techniques for Network Optimisation in Deep Learning models with an emphasis on better estimation.

Source Code : Datalore Link

These sections are borrowed from Nathan & Anirudh's project as it builds upon their data generation. The excerpt from their report is as below:

1.2 Methodology

Prior to constructing our deep learning models, it was important to generate the data used to train, test, and refine them. To that end, we sought to gain a thorough understanding of the SABR volatility model with special attention to understanding its parameters. Once we obtained ranges for our multidimensional parameter space, we utilized QuantLib's FDM solver to obtain our ground truth values. This was a rather significant undertaking and our data generation process is discussed in great detail in Section 3. QuantLib offers us an accurate numerical approximation to the solution to the SABR PDE for a specified input. However, the issue that we, and other practitioners face with this approach (in spite of its accuracy/reliability) is that it takes about a second to compute the desired result. Our central objective was to train a deep learning model to attain the accuracy of QuantLib's FDM solver within the order of a couple microseconds, and in particular succeed over the regions of the parameter space that may result from calibrations to market data. Something that was of particular interest to us was the Hagan approximation. With this approximation we can obtain estimates in under a microsecond.

The innovation in our idea of using Tensor Neural Networks is focused on speed and accuracy. In a large and wide-range parameter setting, neural networks are often painfully slow and computationally heavy. The idea was to accelerate our sampling using a much lighter Tensorized neural network. A detailed discussion on it can be found in section 4.

1.3 SABR Volatility Model

The SABR volatility model is a model describing the movement of a forward rate introduced by Hagan et al. [**?**]. The dynamics of the forward rate and its instantaneous volatility are given by

$$\begin{aligned} dF_t &= \sigma_t F_t^{\beta}\, dW_t \\ d\sigma_t &= \nu \sigma_t\, dZ_t \\ dW_t dZ_t &= \rho\, dt \end{aligned}$$

with initial conditions $F_0 = F$ and $\sigma_0 = \alpha$. Then, any contingent claim on the future value of a forward rate following the SABR model satisfies the arbitrage-free PDE

$$\frac{\partial u}{\partial t} - \frac{\nu^2}{2}\frac{\partial u}{\partial \sigma} + \frac{1}{2}e^{2\sigma_t}F_t^{2\beta}\frac{\partial^2 u}{\partial F^2} + \frac{\nu^2}{2}\frac{\partial^2 u}{\partial \sigma^2} + \rho\nu e^{\sigma_t}F_t^{\beta}\frac{\partial^2 u}{\partial F \partial \sigma} - ru = 0$$

With this PDE, we can approximate a call option's value using either the finite-difference method or with Monte Carlo simulation. Subsequently, we can approximate the implied volatility as well using Black's formula.

1.4 Hagan Approximation

The same Hagan et al. paper provides an approximation (known as the Hagan approximation) to the implied volatility of the call option. While Hagan is supposed to be a good first order approximation, we quickly figured out that it can be way off in some regions as we will present in the results. Hence, this is only used as an additional feature without much gain expectations.

The Hagan approximation and its corresponding asymptotics can be summarized as follows:

$$\begin{aligned}\sigma_B(K,f) \quad &\approx \quad \frac{\alpha}{(fK)^{(1-\beta)/2}\left[1+\frac{(1-\beta)^2}{24}\log^2(f/K)+\frac{(1-\beta)^4}{1920}\log^4(f/K)+\cdots\right]}\cdot\frac{z}{x(z)}\\ &\cdot\left\{1+\left[\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(fK)^{1-\beta}}+\frac{1}{4}\frac{\rho\beta\nu\alpha}{(fK)^{(1-\beta)/2}}+\frac{2-3\rho^2}{24}\nu^2\right]t_{ex}+\cdots\right\}\end{aligned}$$

where

$$z=\frac{\nu}{\alpha}(fK)^{(1-\beta)/2}\log(f/K),\ x(z)=\log\left\{\frac{\sqrt{1-2\rho z+z^2}+z-\rho}{1-\rho}\right\}$$

For the special case of at-the-money options, options struck at $K=f$, this formula reduces to

$$\sigma_{ATM}=\sigma_B(f,f)\approx\frac{\alpha}{f^{(1-\beta)}}\left\{1+\left[\frac{(1-\beta)^2}{24}\frac{\alpha^2}{f^{2-2\beta}}+\frac{1}{4}\frac{\rho\beta\alpha\nu}{f^{(1-\beta)}}+\frac{2-3\rho^2}{24}\nu^2\right]t_{ex}+\cdots\right\}$$

An alternative representation is through normal volatility:

$$\begin{aligned}\sigma_N(K,f) \quad &\approx \quad \frac{1}{\alpha}\cdot\frac{f^{1-\beta}-K^{1-\beta}}{(1-\beta)(f-K)}\cdot\frac{x(z)}{z}\\ &\cdot\left\{\frac{1+\frac{\beta(2-\beta)}{24}\frac{1-\frac{2-2\beta+\beta^2}{120}\log^2(f/K)}{1+\frac{(1-\beta)^2}{12}\log^2(f/K)}\frac{\alpha^2t_{ex}}{(fK)^{(1-\beta)}}+\frac{\beta(2-\beta)}{80}\left[(1-\beta)^2+\frac{1}{72}\beta(2-\beta)\right]\frac{\alpha^4t_{ex}^2}{(fK)^{2-2\beta}}}{1+\frac{\beta\rho}{4}\frac{\alpha\nu t_{ex}}{(fK)^{(1-\beta)/2}}+\frac{2-3\rho^2}{24}\nu^2t_{ex}}\right\}\end{aligned}$$

For the special case of at-the-money options, this formula reduces to

$$\sigma_N(f,f)\approx\frac{1}{\alpha f^\beta}\left\{\frac{1+\frac{\beta(2-\beta)}{24}\frac{\alpha^2t_{ex}}{f^{2-2\beta}}+\frac{\beta(2-\beta)}{80}\left[(1-\beta)^2+\frac{1}{72}\beta(2-\beta)\right]\frac{\alpha^4t_{ex}^2}{f^{4-4\beta}}}{1+\frac{\beta\rho}{4}\cdot\frac{\alpha\nu t_{ex}}{f^{1-\beta}}+\frac{2-3\rho^2}{24}\nu^2t_{ex}}\right\}.$$

Assuming $\beta<1$, then $\lim_{K\to\infty}z=-\infty$ and $x(z)\sim-\log(-z)$. So as $K\to\infty$,

$$\sigma_B(K,f)\sim\frac{\nu\log(f/K)}{1+\frac{(1-\beta)^2}{24}\log^2(f/K)+\frac{(1-\beta)^4}{1920}\log^4(f/K)}\cdot\frac{1}{-\log(-z)}\cdot\left(1+\frac{2-3\rho^2}{24}\nu^2t_{ex}\right).$$

Since $\log(-z)\sim\frac{1-\beta}{2}\log K$,

$$\lim_{K\to\infty}\frac{\nu\log(f/K)}{-\log(-z)}=\frac{2\nu}{1-\beta},$$

and we conclude

$$\lim_{K\to\infty}\sigma(K,f)=0.$$

Assuming $\beta = 1$, Hagan's formula becomes

$$\sigma_B(K,f) \approx \frac{\nu \log(f/K)}{x(z)} \left[1 + \left(\frac{\rho\nu\alpha}{4} + \frac{2-3\rho^2}{24}\nu^2\right) t_{ex}\right],$$

where

$$z = \frac{\nu}{\alpha}\log(f/K),\ x(z) = \log\left\{\frac{\sqrt{1-2\rho z + z^2} + z - \rho}{1-\rho}\right\}.$$

We still have $\lim_{K\to\infty} z = -\infty$ and $x(z) \sim -\log(-z)$. Hence $\lim_{K\to\infty} \sigma_B(K,f) = \infty$.

In summary,

$$\lim_{K\to\infty} \sigma_B(K,f) = \begin{cases} 0 & \beta < 1 \\ \infty & \beta = 1 \end{cases}$$

We further note when $\beta = 1$, Hagan's formula satisfies

$$\sigma_B(K,f) \sim \frac{\nu \log K}{\log(\log K)} \left[1 + \left(\frac{\rho\nu\alpha}{4} + \frac{2-3\rho^2}{24}\nu^2\right) t_{ex}\right], \text{ as } K \to \infty,$$

2 Data Generation

In order to train our models, we needed to generate samples whose independent variables were calibration parameters for the SABR model as well as a call option strike price and tenor, and whose dependent variables were the Black implied volatility, price, and price sensitivities of that call option under the specified SABR model. The SABR model has no arbitrage-free closed-form solution, so we had to approximate the true solution. All of our data was generated using QuantLib, an open-source library for quantitative financial modeling.

2.1 Finite-Difference Method

QuantLib offers two solutions for the above task: a finite-difference method (FDM) solver and a Monte Carlo (MC) engine. We decided to choose the FDM solver for the following reasons:

- Our diffusion process is only two-dimensional, so our efficiency is not dramatically affected by the curse of dimensionality. The curse of dimensionality is a common reason to choose MC for certain use cases.
- Furthermore, precision is not of utmost importance to us, since our goal is to fit any arbitrary set of data. Rather, speed is more important, since we have to generate a significant quantity of data, and faster data generation allows us to iterate quicker. If we had chosen MC, using too few paths could produce highly volatile results, which

may be difficult to model. On the other hand, using too few grid points in FDM usually still produces fairly smooth results, even if they differ significantly from the true values.

- FDM offers a cheap solution for generating price sensitivities — taking finite differences of the FDM inputs. Since FDM is fairly smooth at any scale, we can get relatively accurate sensitivities by taking very small finite differences. On the other hand, at small scales, MC becomes less smooth due to the randomness dominating. This limitation could in theory be overcome by performing the finite differencing within the MC simulation rather than without, but this behavior is not supported out-of-the-box in QuantLib, and it was not within the scope of this project to implement this.

2.2 Symmetry Transformation

In order to reduce the dimensionality of our modeling, we utilized the following scaling symmetry in the SABR model:

$$\begin{aligned} F &\to \lambda F \\ K &\to \lambda K \\ \alpha &\to \lambda^{1-\beta}\alpha \end{aligned}$$

Without loss of generality, we can always choose $\lambda = \frac{1}{F}$, so that the transformed F parameter becomes invariant. This allows us to consider a reduced modeling problem with one fewer variable parameter:

$$\begin{aligned} \sigma_{imp}(K, T, F, \alpha, \beta, \nu, \rho) &= \sigma_{imp}\left(\frac{K}{F}, 1, T, \frac{\alpha}{F^{1-\beta}}, \beta, \nu, \rho\right) \\ &= \sigma_{imp}(K', 1, T, \alpha', \beta, \nu, \rho) \end{aligned}$$

where we define

$$\begin{aligned} K' &= \frac{K}{F} \\ \alpha' &= \frac{\alpha}{F^{1-\beta}} \end{aligned}$$

Reducing the input space helps control overfitting by reducing the number of degrees of freedom without reducing the amount of information contained in the inputs.

2.3 Range Transformation

The ranges of K' values and ν values that we are interested in scale with the maturity of the call option. A hyperrectangular grid over the original parameters disallows us from considering a different range of one parameter depending on a second parameter. We overcame this restriction by constructing transformed parameters that entangle the originally dependent parameters such that the ranges of the original parameters vary as we desire but the ranges of the transformed parameters remain fixed.

2.3.1 Strike Range Transformation

For short maturities and for small initial volatilities, values of K' that are too large or too small can cause instability in the implied volatility calculation. Far OTM calls are worth nearly zero and contribute large negative values to d_1 and d_2 in the Black formula. Far ITM calls are worth nearly the value of the underlying and contribute large positive values to d_1 and d_2. Either way, these situations cause the implied volatility to be very sensitive to small fluctuations in the FDM estimated prices due to approximation error, which makes the implied volatility difficult to model for these parameters.

Furthermore, we are not interested in modeling far OTM or ITM call options anyway, since they are not generally traded with high liquidity in the market. However, a strike that might be considered far OTM or ITM at a short maturity and small initial volatility may not be that far OTM or ITM at a long maturity or with a large initial volatility, since the forward rate has more time to evolve or can move more drastically. Therefore, we construct a transformation that scales the strike price with both T and α.

$$K' \to \frac{\ln(K')}{\alpha'\sqrt{T}}$$

Loosely speaking, this transformed parameter can be interpreted as the number of standard deviation up- or down-moves at the initial volatility required for the forward rate to reach the given strike. Then, for any fixed range of this transformed parameter, the associated range of strikes grows larger as α and T increase, as desired.

2.3.2 Volvol Range Transformation

For long maturities, values of ν that are too large can cause instability in the FDM solver. Over a sufficiently long amount of time, the distribution of volatilities will become very wide, and in particular, there will be a significant probability of volatility increasing by several orders of magnitude. As a concrete example, consider the SABR parameters $\{F = 100, \alpha = 1, \beta = 0, \nu = 4, \rho = 0\}$. In words, the forward rate begins at 100 bp and evolves Normally with initial Normal vol 1 bp. Over the course of 1 year, a one-standard-deviation up-move in the Normal vol would be

$$\begin{aligned}\sigma_1 &= \alpha e^{\nu\sqrt{1}} \\ &= 55\,\text{bp}\end{aligned}$$

which is still at least a believable state of the world. However, over the course of 25 years, a one-standard-deviation up-move in the Normal vol would be

$$\begin{aligned}\sigma_{25} &= \alpha e^{\nu\sqrt{25}} \\ &= 485165195\,\text{bp}\end{aligned}$$

which is obviously not a reasonable state of the world. Within the FDM solver, such large variation in the volatility can result in outsized contributions from boundary condition approximations and increased numerical instability.

Furthermore, we are not interested in SABR parameters that could not result from a reasonable calibration of market data anyway. However, a volvol that would be unreasonably large over long horizons may still be reasonable over short horizons. Therefore, we construct a transformation that scales the volvol inversely with T.

$$\nu \to \nu\sqrt{T}$$

Loosely speaking, this transformed parameter can be interpreted as the total volvol over the specified horizon. Then, for any fixed range of this transformed parameter, the associated range of volvols shrinks as T increases, as desired.

2.4 Target Function

With these three transformations, we arrive at our final target function for our modeling problem:

$$\begin{aligned}\sigma_{imp}(K,T,F,\alpha,\beta,\nu,\rho) &= f\left(\frac{\ln(K')}{\alpha'\sqrt{T}}, T, \alpha', \beta, \nu\sqrt{T}, \rho\right)\\ &= f(\hat{K},T,\hat{\alpha},\beta,\hat{\nu},\rho)\end{aligned}$$

where we define

$$\begin{aligned}\hat{K} &= \frac{\ln(K')}{\alpha'\sqrt{T}}\\ &= \frac{F^{1-\beta}}{\alpha\sqrt{T}}\ln\left(\frac{K}{F}\right)\\ \hat{\alpha} &= \alpha'\\ &= \frac{\alpha}{F^{1-\beta}}\\ \hat{\nu} &= \nu\sqrt{T}\end{aligned}$$

Note that, for the purpose of data generation, there is a fundamental difference between the symmetry transformation and the range transformations. After the symmetry transformation, we can directly run the FDM solver on the transformed values K' and α', and we do not need to generate any samples with $F \neq 1$. This is because, by symmetry, we can reduce any arbitrary problem to one in which $F = 1$. However, after the range transformation, we cannot directly run the FDM solver on the transformed values $\hat{K}$ and $\hat{\nu}$, because these are no longer valid arguments to the SABR model at all. Therefore, once we select desired ranges for $\hat{K}, T, \hat{\alpha}, \beta, \hat{\nu}, \rho$, we must invert the range transformations (but not the symmetry transformations) before simulating them using FDM.

2.5 Parameter Ranges

The challenge in selecting ranges for each (transformed) parameter is in finding the correct balance between ease of modeling and usefulness in a variety of market conditions. If we choose too large a range, our target function becomes prone to numerical instability, as described in Section 2.3. This results in an ill-behaved target function, which reduces the accuracy of any model we might want to train against it. On the other hand, if we choose too small a range, our trained model may not perform well on market parameters that are too far out from the ranges on which the model was trained.

2.5.1 Final Parameter Set

For our final set of parameter ranges, we took advantage of the range transformations, we correctly accounted for the symmetry transformation's effect on the range of α, and we increased the accuracy of the FDM solver by using more grid points. The final ranges were

$$\begin{aligned} \hat{K} &\in [-2.0, 2.0] \\ T &\in [0.5, 30.0] \\ \hat{\alpha} &\in [0.001, 0.09] \\ \beta &\in [0.001, 0.5] \\ \hat{\nu} &\in [0.1, 5.0] \\ \rho &\in [-0.4, 0.4] \end{aligned}$$

For initial values of F around 100 bp, the corresponding α range is approximately

$$\alpha \in \left[\hat{\alpha}_{min} F^{1-\beta_{max}}, \hat{\alpha}_{max} F^{1-\beta_{min}}\right] = [0.01, 8.96]$$

As for the strike and volvol, the ranges depend on the maturity of the call option and the initial volatility. For initial values of F around 100 bp, maturity $T = 0.5$, and transformed initial volatility $\hat{\alpha} = 0.001$, the corresponding K and ν ranges are approximately

$$\begin{aligned} K &\in \left[F e^{\hat{K}_{min}\hat{\alpha}\sqrt{T}}, F e^{\hat{K}_{max}\hat{\alpha}\sqrt{T}}\right] \approx [99.9, 100.1] \\ \nu &\in \left[\frac{\hat{\nu}_{min}}{\sqrt{T}}, \frac{\hat{\nu}_{max}}{\sqrt{T}}\right] \approx [0.1, 7.1] \end{aligned}$$

Meanwhile, for initial values of F around 100 bp, maturity $T = 30.0$, and transformed initial volatility $\hat{\alpha} = 0.09$, the corresponding K and ν ranges are approximately

$$\begin{aligned} K &\in \left[F e^{\hat{K}_{min}\hat{\alpha}\sqrt{T}}, F e^{\hat{K}_{max}\hat{\alpha}\sqrt{T}}\right] \approx [37, 268] \\ \nu &\in \left[\frac{\hat{\nu}_{min}}{\sqrt{T}}, \frac{\hat{\nu}_{max}}{\sqrt{T}}\right] \approx [0.02, 0.91] \end{aligned}$$

2.6 Dataset Statistics

We generated training and testing datasets according to the transformed parameter set detailed in Section 2.5.1. For the training dataset, we generated samples randomly on a 6-dimensional hyperrectangular grid, with approximately equally-spaced grid points in each dimension. Specifically, we chose grid points

$$\begin{aligned}
\hat{K} &\in [-2, -1.875, -1.75, -1.625, -1.5, -1.25, -1.125, -1, -.875, -.75, -.5, -.25, 0, .25, .5,\\
&\quad .75, 0.875, 1, 1.125, 1.25, 1.5, 1.625, 1.75, 1.875, 2]\\
T &\in [0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 16.0, 18.0, 21.0, 24.0, 27.0, 30.0]\\
\hat{\alpha} &\in [0.001, 0.005, .01, .02, 0.025, .03, .04, 0.045, .05, .06, 0.065, .07, .08, 0.085, .09]\\
\beta &\in [0.001, 0.050, 0.100, 0.150, 0.200, 0.250, 0.300, 0.350, 0.400, 0.450, 0.500]\\
\hat{\nu} &\in [0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]\\
\rho &\in [-0.40, -0.35, -0.30, -0.25, -0.20, -0.15, -0.10, -0.05, 0.00, 0.05, 0.10, 0.15,\\
&\quad 0.20, 0.25, 0.30, 0.35, 0.40]
\end{aligned}$$

The probabilities of sampling K and T are also different to account for the fact that the edges of ranges are exactly the points where model generally has the largest errors and thus, a simple strategy is to interpolate and sample more points in those specific ranges.

$$\begin{aligned}
P(\hat{K}) &\in [0.05, 0.05, 0.05, 0.05, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.03, 0.03, 0.03, 0.03,\\
&\quad 0.03, 0.03, 0.03, 0.03, 0.04, 0.04, 0.04, 0.05, 0.05, 0.05, 0.05]\\
P(T) &\in [0.07, 0.07, 0.06, 0.05, 0.04, 0.04, 0.04, 0.04, 0.04, 0.05, 0.05, 0.04, 0.04,\\
&\quad 0.04, 0.04, 0.04, 0.05, 0.06, 0.07, 0.07]
\end{aligned}$$

And for the testing dataset, we generated samples uniformly on the continuous 6-dimensional hyperrectangle. Shown below are the statistics for the two datasets. Note that the first six parameters are the symmetry- and range-transformed parameters, while K' and ν are the actual strike and volvol values on which the FDM solver was executed.

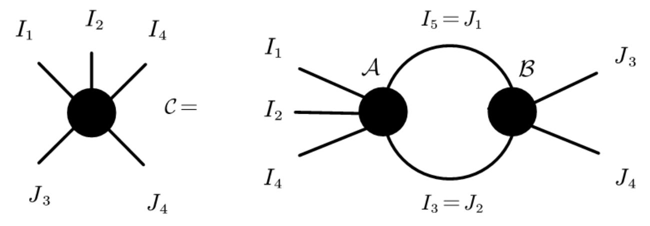
3 Tensor Neural Networks

Deep neural networks currently demonstrate state-of-the-art performance in several domains. At the same time, models of this class are very demanding in terms of computational resources. In particular, a large amount of memory is required by commonly used fully-connected layers, making it hard to use the models on low-end devices and stopping the further increase of the model size.

Consider the most frequently used layer of the neural networks: the fully connected layer. This layer consists in a linear transformation of a high-dimensional input signal to a high-dimensional output signal with a large dense matrix defining the transformation. For example, in modern CNNs the dimensions of the input and output signals of the fully-connected layers are of the order of thousands, bringing the number of parameters of the fully-connected layers up to millions. We use a compact multiliniear format – Tensor-Train (TT-format) and Block-Term Tucker Layers (BlockTT) – to represent the dense weight matrix of the fully-connected layers using few parameters while keeping enough flexibility to perform signal transformations. The resulting layer is compatible with the existing training algorithms for neural networks because all the derivatives required by the back-propagation algorithm can be computed using the properties of the this format. We call the resulting layer a Tensor layer and refer to a network with one or more Tensor layers as Tensor Network.

3.1 Tensor Contraction

Tensor contraction is the most typical operation for tensors, contracting two tensors into one tensor along the associated pairs of indices. As a result, the corresponding connected edges disappear while the dangling edges persist. The Tensor Network representation of such operation can be illustrated as:



As show in above figure, contraction between a 5th-order tensor $\mathcal{A}$ and a 4th-order tensor $\mathcal{B}$ along the index pairs (i_5, j_1) and (i_3, j_2) yields a 5th-order tensor $\mathcal{C}$, with entries

$$\mathcal{C}_{i_1 i_2 i_4 j_3 j_4} = \sum_{i_3, i_5} \mathcal{A}_{i_1 i_2 i_3 i_4 i_5} \mathcal{B}_{i_5, i_3, j_3, j_4}$$

Tensor contractions among multiple tensors can be computed by performing tensor contraction between two tensors many times. Hence, the order (or number of modes) of an entire Tensor Network is given by the number of dangling edges which are not contracted.

Tensor decomposition is a common technique for compressing Neural Networks, by decomposing a higher-order tensor into several lower-order tensors (usually matrices or 3rd-order tensors) that are sparsely interconnected through the tensor contraction operator. The basic tensor decomposition include CANDECOMP/PARAFAC (CP), Tucker, Block Term (BT), Tensor Train (TT) and so on. And such decomposition formats can be illustrated as the corresponding Tensor Network diagrams.

3.2 Tensorized fully connected layer

By replacing Fully-Connected (FC) layers or Convolution Layers with tensorized layers, large amount of parameters can be reduced. For example, a FC layer is formulated as $y = Wx$, By a simple reshaping method, we can reformulate the FC layer as:

$$\mathcal{Y}_{j_1,\ldots,j_M} = \sum_{i_1,\ldots,i_N=1}^{I_1,\ldots,I_N} \mathcal{W}_{i_1,\ldots,i_N,j_1,\ldots,j_M} x_{i_1,\ldots,i_N}$$

3.2.1 Block Term Tucker (BlockTT)

We factorize a higher-order tensor into a sum of several rank-1 tensor components. Recently, a more generalized decomposition method called Block Term (BT) decomposition, which generalizes CP and Tucker via imposing a block diagonal constraint on the core tensor, has been proposed to make a trade-off between them. The BT decomposition aims to decompose a tensor into a sum of several Tucker decompositions with low Tucker-ranks. The mathematical neural network layer format is given below. R_T denotes the Tucker-rank (which means the Tucker-rank equals $\{R_1, ..., R_N\}$) and C represents the CP-rank. They are together called BT-ranks.

3.2.2 Tensor Train (TT)

Matrix Tensor train (mTT) decomposition(sometimes also called Tensor Train), also called Matrix Product Operator(MPO) in quantum physics, factorizes a higher-order tensor into a linear multiplication of a series of 4th-order core tensors. The mathematical neural network layer format is as in figure. $\{R_1, ...R_{N-1}\}$ denote the TT-ranks.

$$\boldsymbol{\mathcal{Y}}_{j_1,\ldots,j_M} = \sum_{i_1,\ldots,i_N=1}^{I_1,\ldots,I_N} \sum_c^C \sum_{r_1,\ldots,r_N=1}^{R_1,\ldots,R_N} \boldsymbol{g}_{r_1,\ldots,r_N} \boldsymbol{a}^{(1)}_{i_1,c,r_1} \ldots \boldsymbol{a}^{(N)}_{i_N,c,r_N} \boldsymbol{a}^{(N+1)}_{j_1,c,r_{N+1}} \ldots \boldsymbol{a}^{(N+M)}_{j_M,c,r_{N+M}} \boldsymbol{x}_{i_1,i_2,\ldots,i_N}.$$

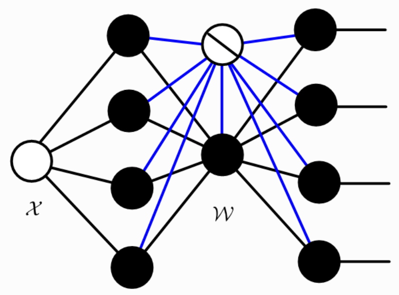


Figure 1: Block Term Tucker

$$\boldsymbol{\mathcal{Y}}_{j_1,\ldots,j_N} = \sum_{i_1,\ldots,i_N=1}^{I_1,\ldots,I_N} \sum_{r_1,\ldots,r_N=1}^{R_1,\ldots,R_N} \boldsymbol{g}^{(1)}_{i_1,j_1,r_1} \boldsymbol{g}^{(2)}_{r_1,i_2,j_2,r_2} \cdots \boldsymbol{g}^{(N)}_{r_N,i_N,j_N} \boldsymbol{x}_{i_1,i_2,\ldots,i_N}.$$

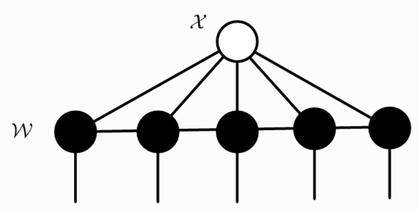
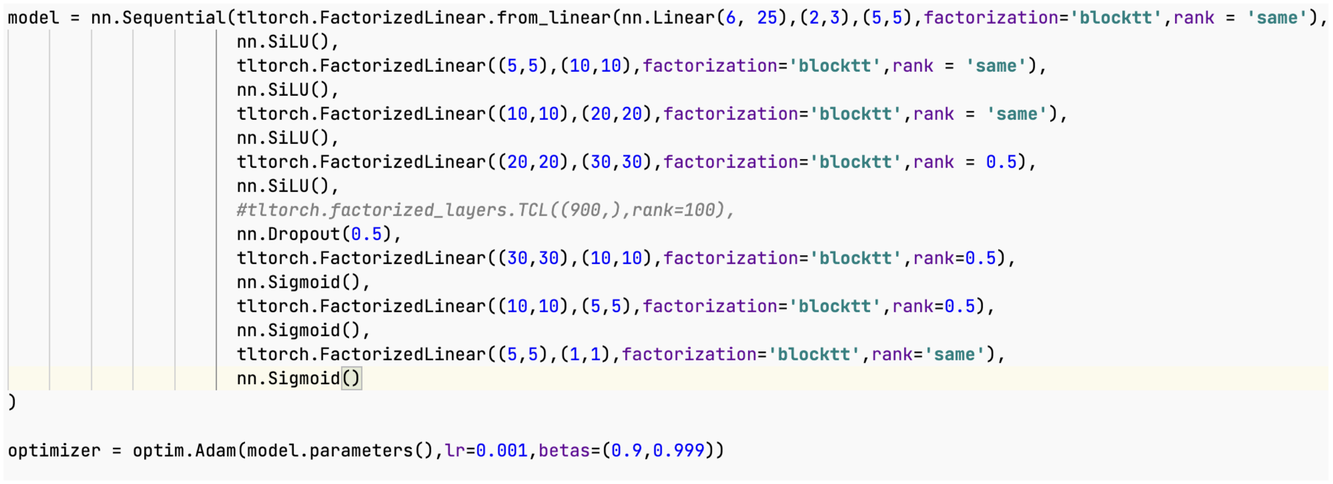


Figure 2: Tensor Train

4 Modeling

4.1 Network

The following is the the structure we used for our network:



```
model = nn.Sequential(tltorch.FactorizedLinear.from_linear(nn.Linear(6, 25),(2,3),(5,5),factorization='blocktt',rank = 'same'),
                      nn.SiLU(),
                      tltorch.FactorizedLinear((5,5),(10,10),factorization='blocktt',rank = 'same'),
                      nn.SiLU(),
                      tltorch.FactorizedLinear((10,10),(20,20),factorization='blocktt',rank = 'same'),
                      nn.SiLU(),
                      tltorch.FactorizedLinear((20,20),(30,30),factorization='blocktt',rank = 0.5),
                      nn.SiLU(),
                      #tltorch.factorized_layers.TCL((900,),rank=100),
                      nn.Dropout(0.5),
                      tltorch.FactorizedLinear((30,30),(10,10),factorization='blocktt',rank=0.5),
                      nn.Sigmoid(),
                      tltorch.FactorizedLinear((10,10),(5,5),factorization='blocktt',rank=0.5),
                      nn.Sigmoid(),
                      tltorch.FactorizedLinear((5,5),(1,1),factorization='blocktt',rank='same'),
                      nn.Sigmoid()
)

optimizer = optim.Adam(model.parameters(),lr=0.001,betas=(0.9,0.999))
```

4.2 Parameters

- The structure that we aim to utilise is to use BlockTT decomposition for representation as explained in the earlier section.
- The network is set up in a way that we expand our feature space by going gradually from a 6-dimensional space to 900 dimensional one. The idea is to keep the same ranks for smaller transformation so as to not lose accuracy, whereas we quickly use rank reduction to 0.5 to ease computation and perform compression.
- The activation function used is SiLu as in the Swish function, defined as:

$$f(x) = x \times \text{sigmoid}(x)$$

  The motivation is that swish is considered better for gains than ReLu and induces smoothness by eliminating the kink in ReLu.
- Factorized Linear Layers: These are tensorized Fully-Connected Layers. The weight matrix is tensorized to a tensor of size (d_1, d_2). That tensor is expressed as a low-rank tensor. During inference, the full tensor is reconstructed, and unfolded back into a matrix, used for the forward pass in a regular linear layer.
- Dropout layer is introduced for further randomized reduction in parameter space and avoid overfitting.

- We used the smoothL1Loss function, defined as:
$$l_n = \begin{cases} 0.5(x_n - y_n)^2/beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$
- We use Adam optimizer with learning rate = 0.001

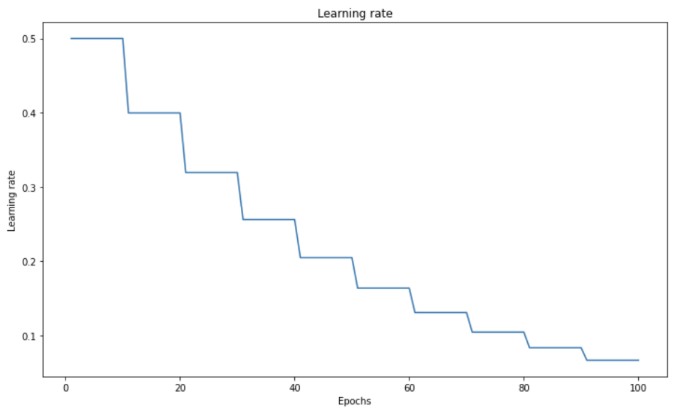
4.3 Schedulers

4.3.1 Learning Rate Schedulers

A Learning rate schedule is a predefined framework that adjusts the learning rate between epochs or iterations as the training progresses. For the training process, this is good. Early in the training, the learning rate is set to be large in order to reach a set of weights that are good enough. Over time, these weights are fine-tuned to reach higher accuracy by leveraging a small learning rate. We tried two scheduling techniques, namely, stepLR and model performance benchmarking.

- stepLR: Under this policy, our learning rate is scheduled to reduce a certain amount every N epochs, where the ‘drop-rate‘ specifies the amount that learning rate is modified, and the ‘epochs-drop‘ specifies how frequent the modification is.

$$rate = (rate_{initial}) * (droprate)^{\lfloor epoch/epochsdrop \rfloor}$$



- Benchmarking based: Under this policy, there is a user-defined metric to monitor such as validation loss, which enables learning rate information to be adjusted based on movements in the metric.

4.3.2 Batch Scheduling

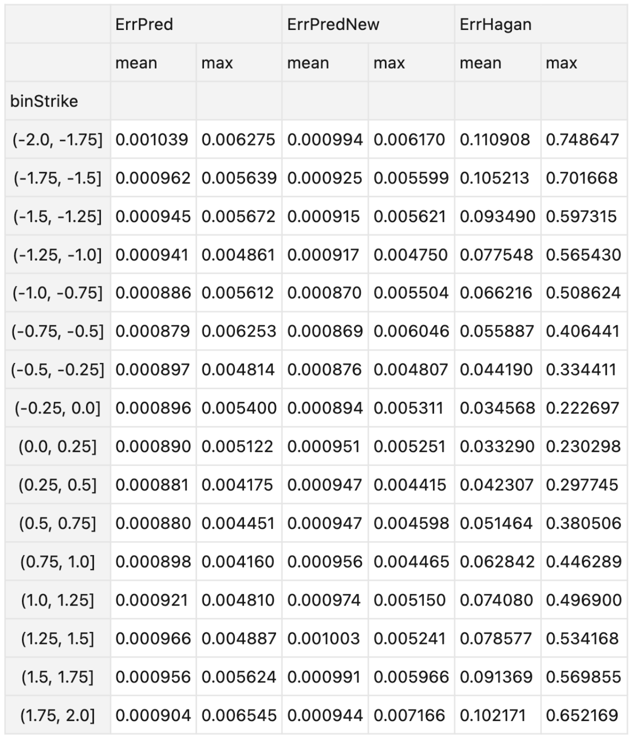
Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le showed that learning rate and batch size are inversely proportional in their study “Do not Decay the Learning

Rate, Increase the Batch Size." The authors proposed that instead of using learning rate schedulers, the batch size should be increased during training which can empirically achieve the same results as decaying learning rates. We tried to implement a hybrid version where batch size is kept constant until few epochs and then reverting rate to initial rate, batch sizes are increased. However, later we removed them from the code as gains were not significant albeit at a loss of training time.

4.4 Error Stratification

A significant discovery in our research was that there were specific buckets of parameter ranges that tended to have higher prediction errors in our model. We implemented 2 ways of dealing with this issue:-

- OverSampling: The idea was to bucket the data for each parameter and observe which ranges are particularly problematic. The observations are as below. Note that mostly the maximum errors are achieved near the edges of our ranges for strike and tenor. The idea is now to change the probability of sampling in the data generation process to oversample from these regions. For example, wrt strikes:



	ErrPred		ErrPredNew		ErrHagan	
	mean	max	mean	max	mean	max
binStrike						
(-2.0, -1.75]	0.001039	0.006275	0.000994	0.006170	0.110908	0.748647
(-1.75, -1.5]	0.000962	0.005639	0.000925	0.005599	0.105213	0.701668
(-1.5, -1.25]	0.000945	0.005672	0.000915	0.005621	0.093490	0.597315
(-1.25, -1.0]	0.000941	0.004861	0.000917	0.004750	0.077548	0.565430
(-1.0, -0.75]	0.000886	0.005612	0.000870	0.005504	0.066216	0.508624
(-0.75, -0.5]	0.000879	0.006253	0.000869	0.006046	0.055887	0.406441
(-0.5, -0.25]	0.000897	0.004814	0.000876	0.004807	0.044190	0.334411
(-0.25, 0.0]	0.000896	0.005400	0.000894	0.005311	0.034568	0.222697
(0.0, 0.25]	0.000890	0.005122	0.000951	0.005251	0.033290	0.230298
(0.25, 0.5]	0.000881	0.004175	0.000947	0.004415	0.042307	0.297745
(0.5, 0.75]	0.000880	0.004451	0.000947	0.004598	0.051464	0.380506
(0.75, 1.0]	0.000898	0.004160	0.000956	0.004465	0.062842	0.446289
(1.0, 1.25]	0.000921	0.004810	0.000974	0.005150	0.074080	0.496900
(1.25, 1.5]	0.000966	0.004887	0.001003	0.005241	0.078577	0.534168
(1.5, 1.75]	0.000956	0.005624	0.000991	0.005966	0.091369	0.569855
(1.75, 2.0]	0.000904	0.006545	0.000944	0.007166	0.102171	0.652169

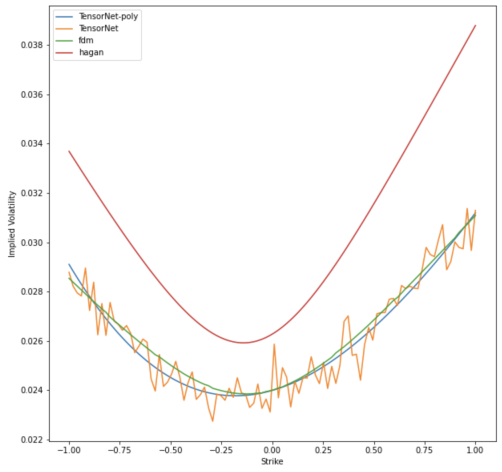
- Boosting: Essentially the idea is to use boosting techniques such as XGBoostRegressor to put the emphasis on points causing errors and try to eliminate using a non-linear model. We try to model the residuals of our prediction with the same features as above. However, we observe that the scale of the errors have now reached the stage where they are less than our general tolerance levels. Therefore become random and hence this method does not produce results as expected.

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, enable_categorical=False,
             eta=0.01, eval_metric=['rmse', 'mae'], gamma=0, gpu_id=-1,
             importance_type=None, interaction_constraints='',
             learning_rate=0.00999999978, max_delta_step=0, max_depth=5,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=5000, n_jobs=2, num_parallel_tree=1, predictor='auto',
             random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
             subsample=0.5, tree_method='exact', validate_parameters=1,
             verbose=2, ...)
```

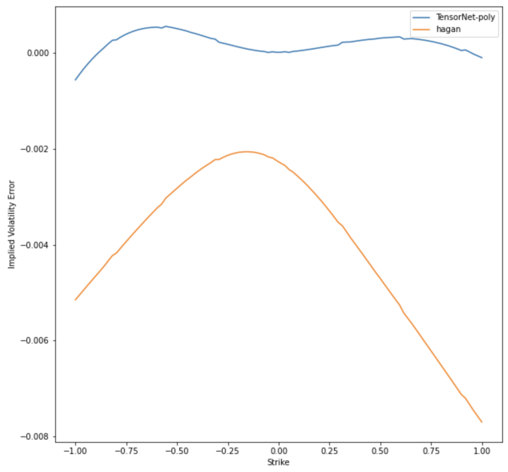
4.5 Splines/Polynomial Fitting

The output from tensor network is not as smooth due to limited training computation power. The idea is then to use the model estimates for market estimated parameters and fit a spline/polynomial fitting in strike and tenor to achieve required smoothness for stable greeks.

5 Results and Plots

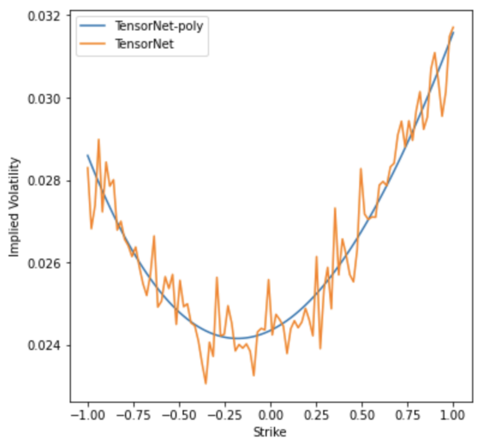


(a) Implied vol

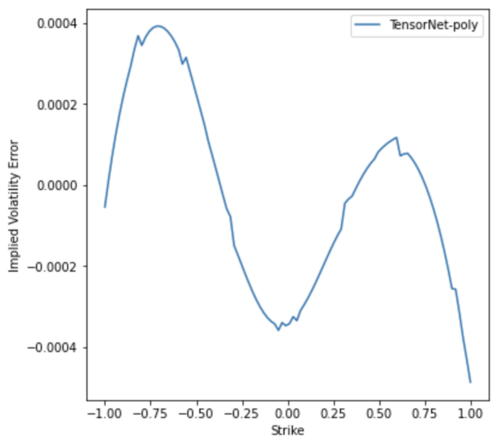


(b) Implied vol Error

Figure 3: f = 100,T = 2,$\alpha = 0.02$,$\beta = 0.2$,$\nu = 2$,$\rho = 0.2$

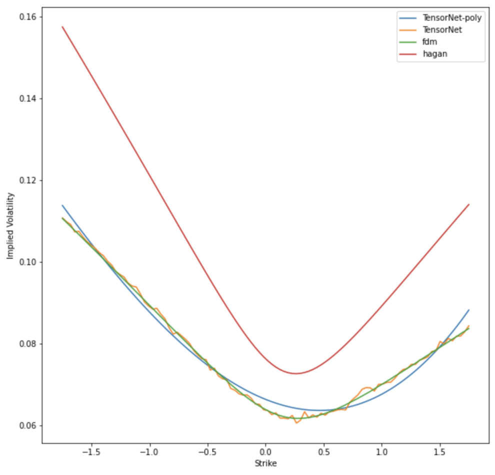


(a) Implied vol

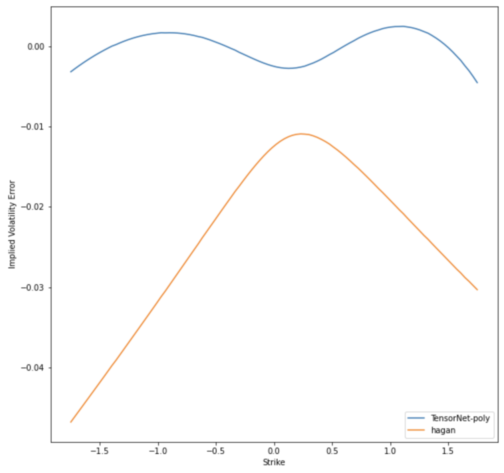


(b) Implied vol Error

Figure 4: f = 100,T = 2,$\alpha = 0.02$,$\beta = 0.2$,$\nu = 2$,$\rho = 0.2$

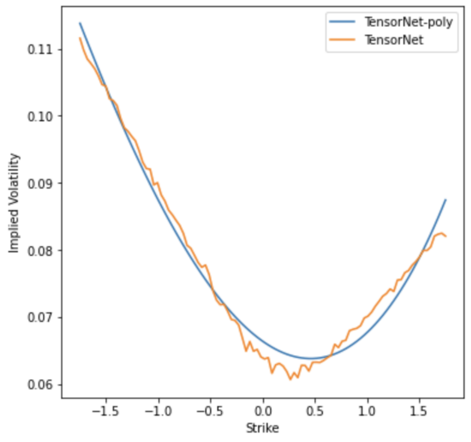


(a) Implied vol

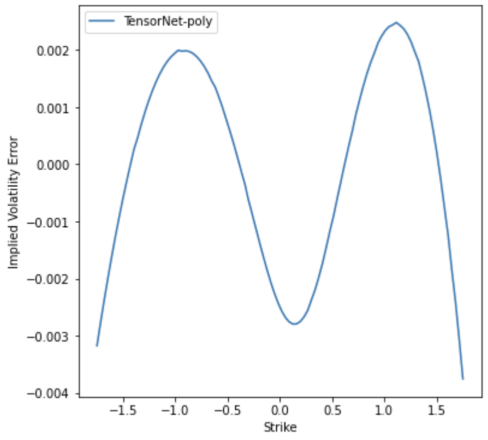


(b) Implied vol Error

Figure 5: f = 200,T = 15,$\alpha = 0.06$,$\beta = 0.5$,$\nu = 2$,$\rho = -0.3$

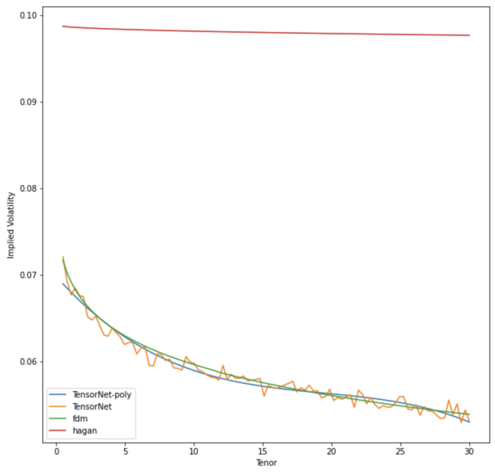


(a) Implied vol

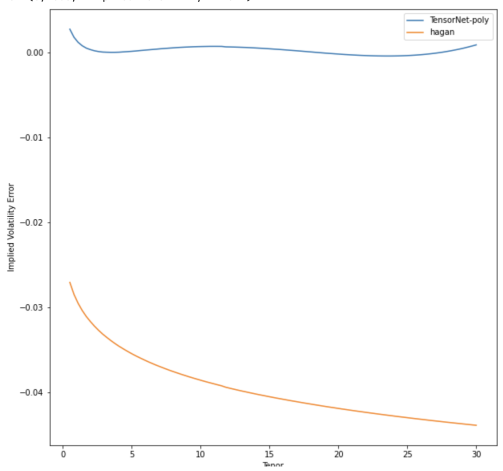


(b) Implied vol Error

Figure 6: f = 200,T = 15,$\alpha = 0.06$,$\beta = 0.5$,$\nu = 2$,$\rho = -0.3$

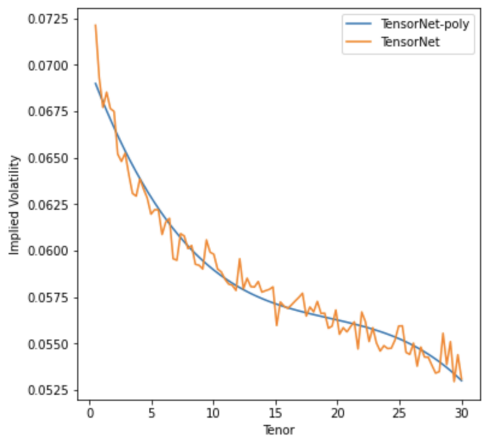


(a) Implied vol

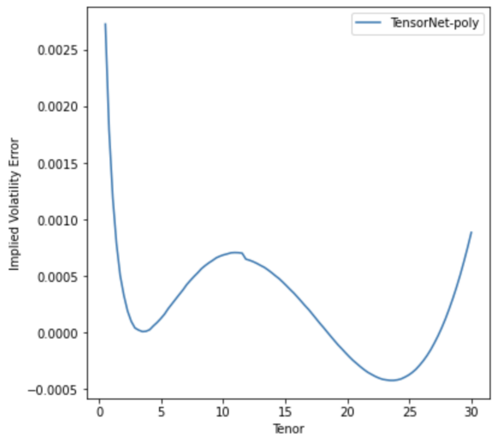


(b) Implied vol Error

Figure 7: f = 200,K' = 0,$\alpha = 0.06$,$\beta = 0.3$,$\nu = 3$,$\rho = -0.3$



(a) Implied vol



(b) Implied vol Error

Figure 8: f = 200,K' = 0,$\alpha = 0.06$,$\beta = 0.3$,$\nu = 3$,$\rho = -0.3$

Bibliography

- Hagan, Patrick S.; Kumar, Deep; Kesniewski, Andrew S.; Woodward, Diana E. (January 2002). "Managing Smile Risk". Wilmott. Vol. 1. pp. 84–108.
- Almost exact SABR Interpolation using Neural Networks and Gradient Boosted Trees (10- 12-2019). https://hpcquantlib.wordpress.com/2019/10/12/almost-exact-sabr-interpolation-using-neural- networks-and-gradient-boosted-trees/.
- Zhou Tingting, Yang Luhua. DNN in pricing derivatives with stochastic volatility.
- Jean Kossaifi, Yannis Panagakis, Anima Anandkumar and Maja Pantic, TensorLy: Tensor Learning in Python, Journal of Machine Learning Research, Year: 2019, Volume: 20, Issue: 26, Pages: 1-6. http://jmlr.org/papers/v20/18-277.html.
- Ruiz, Ignacio, et al. Machine Learning for Risk Calculations: A Practitioner's View. Wiley, 2022.
- Andy Wang, Why Using Learning Rate Schedulers in NNs May Be a Waste of Time,
- Katherine (Yi) Li, How to Choose a Learning Rate Scheduler for Neural Networks
- Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, Dmitry Vetrov, Tensorizing Neural Networks,
- TedNet: A Pytorch Toolkit for Tensor Decomposition Networks,
- Tensorly-Torch, http://tensorly.org/torch/dev/install.html

Applications of diffusion processes in Finance

Diffusion for generative learning (1/5)

- A successful generative learning model will satisfy the following -
  - High-quality sampling
  - Capture sample diversity
  - Fast generation of samples
- Current methods focus mainly on the high-quality sampling
- Capturing data diversity to avoid biases in learned models is highly desirable
- Eg: Cases where long-tailed distribution are important, we don’t want to undersample in tails
- A need for faster sampling is required in applications for real-time usage

Diffusion for generative learning (2/5)

- Diffusion processes can be used as a substitute for GANs
- Models called Denoising diffusion models or score-based generative models give high quality samples, outperforming GANs and model data distribution in tails very well
- Denoising diffusion models consists of -
    - Forward diffusion - Maps data to noise(Gaussian) by sampling Gaussian noise and incrementally add it to the data
    - Reverse process - Undoes forward diffusion and iteratively denoises random noise into realistic data [Requires training of a NN here]

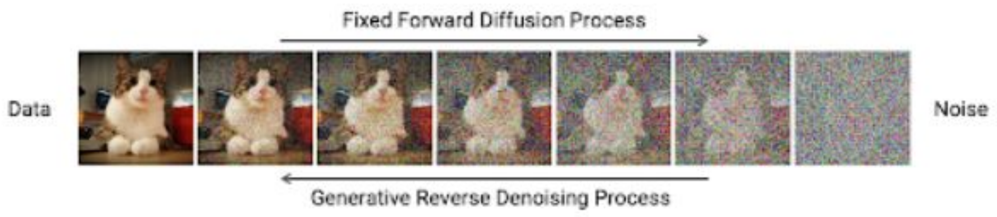
Diffusion for Generative learning (3/5)

- Example -

Denoting a data point by x_0, and it's diffused version at timestep t by x_t, the forward process is defined as - $q(x_{1:T}|x_0) = \prod_{t\geq 1} q(x_t|x_{t-1}), \quad q(x_t|x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$

The reverse generative process is similarly defined in reverse order by -

$$p_\theta(x_{0:T}) = p(x_T)\prod_{t\geq 1} p_\theta(x_{t-1} \mid x_t), \quad p_\theta(x_{t-1} \mid x_t) = N(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I)$$


Fixed Forward Diffusion Process
Data
Noise
Generative Reverse Denoising Process

Diffusion for generative learning (4/5)

- The discrete processes described on the slide before can be approximated by SDEs leading to continuous time diffusion models
- Working with SDEs is easier since-
    - More generic, can be converted to discrete time counterpart whenever needed
    - Can be solved using many available SDE solvers
    - Can be converted to ODEs (Dynkin's formula) which are easy to work with
- However, solving the generative SDE is often complex and requires lot of computation for NN to generate a sample, hence GANs beat diffusion model in terms of speed

Diffusion for Generative learning (5/5)

- NVIDIA research team proposed 3 solutions for speeding up generative process - Denoising diffusion GAN leads to massive speed up
- Denoising Diffusion GAN Idea -
  - Reduce the number of steps required for denoising in the reverse process
  - Observation - Learned denoising distribution in reverse synthesis process can be approximated by Gaussian distribution, but only for small denoising steps, leading to slow reversal
  - When large steps are used, a non-Gaussian multimodal distribution is needed
- Denoising Diffusion GANs represent denoising model using a multimodal conditional GAN, enabling to efficiently generate data in as few as 2 steps

Application in finance - sampling data

- Currently, we are using Quantlib’s FDM for generating ground truth for training our model which takes a lot of time
- We can possibly, train a denoising diffusion GAN using few ground truth values and continuously sample from trained diffusion process to quickly sample more data points

Policy regularization using diffusion models (1/2)

- Offline RL aims to learn policy on a static dataset and hence tends to perform poor on out of sample dataset
- There exist regularization methods for Offline RL algorithms but they tend to restrict policy space a lot often and hence lead to sub-optimal solutions
- Zhendong, Jonathan, Mingyuan explored policy regularization using diffusion processes
- They introduced Diffusion-QL, a new offline RL algorithm that leverages diffusion models to do precise policy regularization

Policy regularization using diffusion models (2/2)

- Condition on state, an action pair is generated using conditional diffusion process
- A separate reward model is learned to predict the cumulative reward for each trajectory generated. This is then injected into reverse sampling stage to learn a trajectory given cumulative reward
- Benefits - g distribution matching technique and hence it could be seen as a powerful sample-based policy regularization method without the need for extra behavior cloning
- Applications - Portfolio construction and management, trading bots

Detecting change in drift of Brownian Motion (1/1)

- Shiryayev and Roberts proposed a diffusion process model (Shiryayvev-Roberts process) for detecting change in drift of Brownian motion
- Given a Brownian motion process W(t) which has some drift μ_1 in time interval $[0,t_0]$ and drift μ_2 for time $t > t_0$, the model proposes a stopping rule T which detects the change point t_0 as soon as possible
- Application - A shock in market may change drift of Brownian motion for an asset and might be needed to be incorporated into pricing models or trading strategies to quickly adjust to the new market conditions