WI4224: Special Topics in Financial Engineering - Final Report

Hans de Munnik & Pim Keer

1st of July 2022

1 Introduction

In this report, we summarise our learning journey through the paper "A Generative Adversarial Network Approach to Calibration of Local Stochastic Volatility Models", by Cuchiero, Khosrawi and Teichmann [1]. In short, the paper proposes a new way to calibrate so-called local stochastic volatility models. These models describe the dynamics of a discounted asset price, and are characterised by a leverage function. In the approach by Cuchiero et al., this leverage function is parameterised by a family of feed-forward neural networks. The novelty here is that on one hand, a neural network specifies a leverage function, defining a certain volatility surface, while on the other hand, the quality of this surface can be determined by an adversarial neural network, which measures the implied distance to market option prices. This interaction between a generating neural network and a controlling neural network is well-known within the machine learning field, under the name of Generative Adversarial Networks (GANs for short).

We will discuss the necessary theory, starting from the structure of a local stochastic volatility model and describing how it can be parameterised with a feed-forward neural network. This will allow us to formulate the problem of calibration as a generative-adversarial optimisation problem. In a usual machine learning problem, this would be the point at which a typical optimisation method such as stochastic gradient descent would be used. However, due to the nonlinear nature of the problem, this will turn out to be impossible. As such, we will talk about how Cuchiero et al. manage to reduce the computational burden of the problem using variation-reduction methods in order to make standard gradient descent a feasible solution. Throughout this treatment, we will mention the difficulties we encountered. Finally, the key parts of the implementation of the ideas discussed will be expounded upon.

2 The Calibration Problem

Given a set of historical and current market data, we want to be able to find the optimal parameters of a certain model such that it best describes the aforementioned data. This is known as the calibration of the model. In the paper by Cuchiero et al. this calibration problem is regarded as the quest for the model (viz. the parameters of a neural network representing the model) which optimally generates the given market data. The way in which optimality is defined will be postponed to when we discuss the formulation of the optimisation problem. The model under consideration here is a combination of a stochastic volatility model and a local volatility model; the so-called local stochastic volatility (LSV) model, where the discounted asset price process $(S_t)_{t\geq 0}$ is given by

$$\mathrm{d}S_t = S_t L(t, S_t)\alpha_t \mathrm{d}W_t, \qquad S_0 > 0. \tag{1}$$

Here $(W_t)_{t\geq 0}$ is a one-dimensional Brownian motion, L is the leverage function (which represents the local part of the model; in the sense that it is only a function of time and the asset price, and not another source of randomness) and $(\alpha_t)_{t\geq 0}$ can be any stochastic volatility process. The latter marks stochastic part of the LSV model, as it depends on an "external" source of randomness. Cuchiero et al. chose for α to be described

by a geometric Brownian motion, i.e.

$$\mathrm{d}\alpha_t = \nu\alpha_t \mathrm{d}B_t, \qquad \mathrm{d}[W,B]_t = \rho \mathrm{d}t, \tag{2}$$

where $\nu \in \mathbb{R}$, $\alpha_0 > 0$, $-1 \leq \rho \leq 1$ such that $(B_t)_{t\geq 0}$ is a Brownian motion correlated to $(W_t)_{t\geq 0}$ with instantaneous correlation ρ. The model (1)-(2) is referred to as a SABR-LSV model. Such models yield a good fit to historical data and are even capable of perfectly calibrating to volatility smiles. The latter property is due to the local volatility part of the model. Traditionally, if we consider a local volatility model of the form $\mathrm{d}S_t = \sigma(t, S_t)\mathrm{d}W_t$ (this also extends to LSV models), Dupire shows that, given European call option market prices $C(T, K)$ for each maturity T and strike K, the local volatility model with local volatility function

$$\sigma_{\mathrm{Dup}}(T,K) = \sqrt{\frac{\frac{\partial C(T,K)}{\partial T}}{\frac{1}{2}K^2 \frac{\partial^2 C(T,K)}{\partial K^2}}} \tag{3}$$

will exactly reproduce the quoted market prices [2]. Hence, $\sigma_{\mathrm{Dup}}(t, S_t)$ is computed by setting $T = t$ and $K = S_t$ in equation (3) In practice however, we only know a discrete collection of market prices, say $\{C(T_i, K_j) : 1 \leq i \leq n, 1 \leq j \leq J_i\}$. The procedure to construct the Dupire local volatility function is then to first compute $\sigma_{\mathrm{Dup}}(T_i, K_j)$ for $1 \leq i \leq n, 1 \leq j \leq J_i$. This is done by using difference quotients to approximate the derivatives in (3). To define the local volatility function in between these points, we are bound to some interpolation scheme. A common approach is first to fit a certain interpolation method, like a cubic spline, through every strike, for every fixed maturity. Next, another interpolation method is used to connect the cubic splines over the different maturities in order to create a (smooth) volatility surface. Although this will yield a perfect fit for the quoted option prices, it is not immediately clear this will give good results for options outside of this small subset of strikes and maturities. Indeed, without further measures, this approach might not even be arbitrage-free. Apart from that, the choice of interpolation method is in a sense arbitrary; making this approach far from ideal.

Instead of the interpolation approach mentioned above, Cuchiero et al. represent the leverage function by a feed-forward neural network; which is trained in order to accurately generate the given market data. To this end, suppose the given market (European call) option prices have corresponding maturities $0 < T_1 < T_2 < \ldots < T_n = T$. Moreover, assume we have J_i options with maturity T_i, with respective strikes K_{ij} for $j \in \{1, \ldots, J_i\}$. If we let $T_0 = 0$, the leverage function can be written as

$$L(t, s; \theta_1, \ldots, \theta_n) = 1 + \sum_{i=1}^{n} F^i(s; \theta_i)\mathbb{1}_{[T_{i-1},T_i)}(t), \tag{4}$$

where $F^i : \mathbb{R} \to \mathbb{R}$ is a feed-forward neural network with parameters $\theta_i \in \Theta_i$. Note that we have a neural network for each maturity. The constant unit in front of the neural network is natural, as the leverage function is in fact just a multiplying factor in equation (1). The parameters θ_i will be chosen such that the distance between the market prices and the model prices are as small as possible. That is, for maturity T_i, we have the following optimisation problem:

$$\inf_{\theta_i} \sup_{\gamma_i} \sum_{j=1}^{J_i} w_{ij}^{\gamma_i} \ell^{\gamma_i}\big(\pi_{ij}^{\mathrm{mod}}(\theta_i) - \pi_{ij}^{\mathrm{mkt}}\big), \qquad i \in \{1, \ldots, n\}. \tag{5}$$

The infimum over all values of θ_i of a loss function ℓ between the market price of the option with strike K_{ij} and maturity T_i and the corresponding model price (which is of course a function of θ_i) should not come as a surprise. However, what is the reasoning introducing a supremum over some other parameter γ_i? This shows the adversarial nature of this neural network approach. The weights $w_{ij}^{\gamma_i}$ and the non-negative convex loss function ℓ^{γ_i} (with $\ell^{\gamma_i}(x) = 0$ only for $x = 0$) are parameterised by γ_i, which is chosen by a second neural network in such a fashion that the error made by the neural network producing the leverage function is as large as possible. For example, it can put more weight on model prices which deviate a lot from the actual market price. In other words, the optimisation problem could be interpreted as a zero-sum game where

one player, the neural network generating the leverage function (generator), follows a minimax decision rule when looking for the optimal parameters θ_i, minimising the maximal loss that occurs when playing against the other player, the neural network trying to spot the incorrect model prices (discriminator). This is the main idea behind generative adversarial networks (GANs). We will explore this concept in more detail, as well as its broad applicability outside of finance and a corresponding example, in Appendix A.

Let us now focus on the optimisation aspect of equation 5. In particular, we will look at a generic maturity $T > 0$, getting rid of the index i, and remove the dependence of w_{ij} and ℓ on the parameter γ. We implicitly assume that, at this point, we have found the optimal γ, such that we are only left with a minimisation problem over the parameter space Θ. This we can write as:

$$\inf_{\theta\in\Theta} \sum_{j=1}^{J} w_j \ell\big(\pi_j^{\text{mod}}(\theta) - \pi_j^{\text{mkt}}\big). \tag{6}$$

We can modify this expression a bit, by realising that the model prices $\pi_j^{\text{mod}}(\theta)$ are in fact found with the well-known payoff of a European call option, i.e. $\pi_j^{\text{mod}}(\theta) = \mathbb{E}\Big[(S_T(\theta) - K_j)^+\Big]$. Introducing the random variables $Q_j(\theta)(\omega) := (S_T(\theta)(\omega) - K_j)^+ - \pi^{\text{mkt}}$, the calibration problem becomes minimising the function $f(\theta)$ w.r.t. $\theta \in \Theta$, where $f(\theta) := \sum_{j=1}^{J} w_j \ell(\mathbb{E}[Q_j(\theta)])$. This quantity will be estimated via a Monte Carlo simulation of N runs, i.e. we are minimising

$$\inf_{\theta\in\Theta} f^{MC}(\theta) := \inf_{\theta\in\Theta} \sum_{j=1}^{J} w_j \ell\left(\frac{1}{N}\sum_{n=1}^{N} Q_j(\theta)(\omega_n)\right), \tag{7}$$

where the $\{\omega_n\}_{n=1,\dots,N} \subset \Omega$ are i.i.d. samples. At this point in time, one would employ a technique like stochastic gradient descent (SGD) in order to find the optimal θ. The shape of this problem however does not allow this. This was a key point in understanding the paper, and justified in fact the extra work of using control variates later on. The regular form of an objective function, as mentioned in the slides by Papapantoleon and Liu [3] is given by $g(\theta) = \frac{1}{N}\sum_{n=1}^{N} \ell(Q(\theta; x_n))$, where x_n is the given data. The general idea of SGD is, in each iteration k, to pick a random sample E_k^m of size m from $\{1, \dots, N\}$ and to update the neural network parameters by $\theta^{(k+1)} = \theta^{(k)} - \frac{\eta_k}{m}\sum_{n\in E_k^m} \nabla\ell\big(Q(\theta^{(k)}; x_n)\big)$. Here $(\eta_k)_{k\geq 1}$ is a non-increasing sequence of learning rates. Just as with ordinary gradient descent (OGD), which has as updating scheme $\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla g(\theta^{(k)})$, SGD will yield an optimal θ. The key reason for SGD working is that the gradient of the sum over the random sample $\frac{1}{m}\sum_{n\in E_k^m} \nabla\ell\big(Q(\theta^{(k)}; x_n)\big)$ is an unbiased estimator for the full gradient $\nabla g\big(\theta^{(k)}\big)$. Indeed, if we denote by $\mathcal{E}^m$ the set of all possible random samples of $\{1, \dots, N\}$ of size m,

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{m}\sum_{n\in E_k^m} \nabla\ell\Big(Q(\theta^{(k)}; x_n)\Big)\right] &= \frac{1}{m}\sum_{e^m\in\mathcal{E}^m}\left(\sum_{n\in e^m} \nabla\ell\Big(Q(\theta^{(k)}; x_n)\Big)\right) P(E_k^m = e^m) \\
&\overset{(*)}{=} \frac{1}{m}\sum_{e^m\in\mathcal{E}^m}\left(\sum_{n\in e^m} \nabla\ell\Big(Q(\theta^{(k)}; x_n)\Big)\right)\frac{1}{\binom{N}{m}} \\
&\overset{(**)}{=} \frac{\binom{N-1}{m-1}}{m\binom{N}{m}}\sum_{n=1}^{N} \nabla\ell\Big(Q(\theta^{(k)}; x_n)\Big) \\
&= \frac{1}{N}\sum_{n=1}^{N} \nabla\ell\Big(Q(\theta^{(k)}; x_n)\Big),
\end{aligned}$$

which is nothing else than $\nabla g(\theta^{(k)})$. At $(*)$ we used that there are $\binom{N}{m}$ possible samples of size m out of $\{1, \dots, N\}$. At $(**)$, we realised that the double sum is, by symmetry, in fact just a sum where each n occurs $\binom{N-1}{m-1}$ times, as there are exactly as many groups containing a given n. In the setting of equation (7)

however, if we just consider the $J = 1$ case, the objective function is of the form $g(\theta) = \ell\Big(\frac{1}{N}\sum_{n=1}^{N} Q(\theta; x_n)\Big)$. Naively performing the same steps as before – which feels a lot more awkward than in the previous case – we would in each iteration k pick a random sample E_k^m from $\{1, \dots, N\}$ and update the parameters by $\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla \ell\Big(\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\Big)$. Since ℓ is not necessarily (in our setting actually never) linear, $\nabla \ell\Big(\frac{1}{m}\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\Big)$ is a biased estimator for the full gradient $\nabla g(\theta^{(k)})$, meaning that SGD breaks down. Indeed,

$$\begin{aligned}
\mathbb{E}\left[\nabla \ell\left(\frac{1}{m}\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\right)\right] &= \sum_{e^m\in\mathcal{E}^m}\left(\nabla \ell\left(\frac{1}{m}\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\right)\right) P(E_k^m = e^m) \\
&= \nabla\left(\frac{1}{\binom{N}{m}}\sum_{e^m\in\mathcal{E}^m} \ell\left(\frac{1}{m}\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\right)\right) \\
&\neq \nabla \ell\left(\frac{1}{m\binom{N}{m}}\sum_{e^m\in\mathcal{E}^m}\sum_{n\in E_k^m} Q(\theta^{(k)}; x_n)\right) \\
&= \nabla \ell\left(\frac{1}{N}\sum_{n=1}^{N} Q(\theta^{(k)}; x_n)\right) \\
&= \nabla g(\theta^{(k)}),
\end{aligned}$$

where we followed the same reasoning as in the previous computation. Again, the inequality follows from the nonlinearity of ℓ.

Now that we have established that SGD cannot be applied here, we should look for other ways to solve the optimisation problem. One alternative would be to just use OGD, computing the full gradient $\nabla g(\theta^{(k)})$. However, note that this involves compute the gradient of the loss function of a sum consisting of N terms. In order to achieve a satisfactory level of precision, N should be chosen large enough; recall that the error of a Monte Carlo simulation decays as $\frac{1}{\sqrt{N}}$. This requirement on N renders ordinary gradient descent to be infeasible. Cuchiero et al. therefore propose to use a variation reduction technique, so-called control variates. This will allow to reach a level of precision with a smaller N, such that OGD is a realistic alternative to SGD.

In Appendix B, we briefly explain the concept of using hedging strategies as control variates, and will now talk about how to apply this concept to the current setting. We can introduce, just as Appendix B, for the j^{th} option (with $j \in \{1, \dots, J\}$), a hedging strategy $h_j : [0, T] \times \mathbb{R} \to \mathbb{R}$ with $h(\cdot, S) \in L^2(S)$ and a constant c. This hedging strategy can be found through delta-hedging, or via another neural network, as described in the Appendix. Now, we can define the random variables $X_j(\theta)(\omega) := Q_j(\theta)(\omega) - c(h_j(\cdot, S_\cdot(\theta)(\omega)) \bullet S_\cdot(\theta)(\omega))_T$. Replacing $Q_j(\theta)(\omega_n)$ by $X_j(\theta)(\omega_n)$ in (7), yields a optimisation problem which is feasible to solve with OGD, as a smaller N will yield a similar degree of precision as a larger N without this control variate modification.

3 Numerical Experiment

Now that we have described the method proposed by Cuchiero et al. in detail, let us demonstrate the method's capabilities. We will consider a certain "ground truth"; a simulated reality which will produce market option prices/volatility smiles by a parametric family. This will be our "given" market data (which is thus in fact simulated, and not actual market data), which we will calibrate using the aforementioned neural network approach, within some (acceptable) error margin. Let us briefly go through the setup of Cuchiero et al., and then mention where we deviated from this in our own experiment. Thereafter, we will present the results we obtained from this experiment.

3.1 Implementation

As we have seen earlier, Dupire's local volatility function makes it possible to exactly calibrate a local (stochastic) volatility model for a discrete collection of prices. Therefore, any price data which could be observed in the market, can be recreated by the dynamics

$$\mathrm{d}S_t = \sigma_{\mathrm{Dup}}(t, X_t) S_t \mathrm{d}W_t, \tag{8}$$

or equivalently by Itô's lemma,

$$\mathrm{d}X_t = -\frac{1}{2}\sigma^2_{\mathrm{Dup}}(t, X_t)\mathrm{d}t + \sigma_{\mathrm{Dup}}(t, X_t)\mathrm{d}W_t, \tag{9}$$

where $X_t := \log(S_t)$. As Cuchiero et al. call it, the ground truth assumption made here is that $\sigma_{\mathrm{Dup}}(t, x)$ can be replaced by a function $a_\xi(t, x)$ in (9), which is a member of a parametric family, indexed by the parameter vector ξ. For the details of this parametric family we refer to Cuchiero et al.[1]. Important for us is that this assumption entails that all observable sets of market prices can be reached solely by varying the parameters ξ.

In the numerical experiment performed by Cuchiero et al., 200 samples of ξ have been generated. For each ξ, option prices with a predetermined set of maturities and strikes were calculated. The maturities were $T_1 = 0.15$, $T_2 = 0.25$, $T_3 = 0.5$ and $T_4 = 1$, and the strikes were 20 equidistant points running from e^{-k_i} up to and including e^{k_i}, where i is the maturity index and $k_1 = 0.1$, $k_2 = 0.2$, $k_3 = 0.3$ and $k_4 = 0.5$. Throughout this experiment the present underlying price S_0 was set to 1. The option pricing was done using Monte Carlo simulation, where 10^7 underlying price paths were simulated according to the Euler discretised version of (9), with a time step of $\Delta t = 0.01$. A delta-hedging strategy, as described in appendix B, has been used as a control variate during the simulation.

Given this "market" data, Cuchiero et al. started calibrating the parameters in the model given by (1)-(2), i.e. retrieving values for ν, α_0, ρ and a set of in this case 4 neural networks (one per maturity) for the leverage function $L(t, S_t)$. Each of these neural networks had a feed-forward architecture, consisting of 4 hidden layer of dimension 64, with a leaky ReLU activation function ($\phi(x) = \alpha x \mathbb{1}_{\{x<0\}} + x\mathbb{1}_{\{x\geq 0\}}$; here

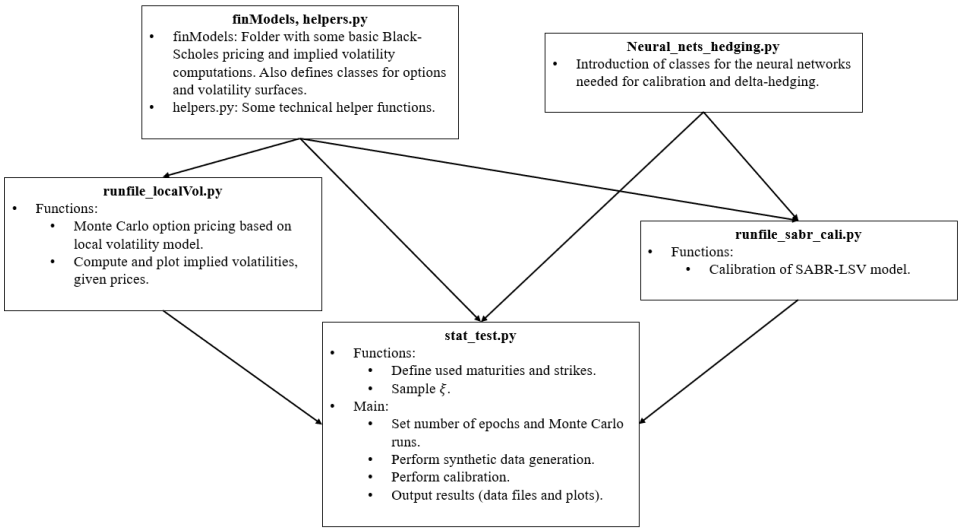

Figure 1: Main structure of the Github repository made available by Cuchiero et al. The boxes represent files in the repository with their respective purposes, and the arrows should be read as "is called in". When executed with the correct packages installed, `stat_test.py` returns all the necessary outputs.

with $\alpha = 0.2$) for the first 3 layers, and a hyperbolic tangent for the final layer. This choice of architecture and of activation functions was made by the authors as they found out after some trial and error that this yielded more smooth results, with a smaller calibration error and lower computation time.

The calibration process consists of two steps, per maturity. First, the leverage function is set to 1, and ν, α_0 and ρ are. Second, these three calibrated values are fixed and the leverage function is calibrated according to the variance-reduced version of (7). A delta-hedging strategy with volatility $L(t, S_t)\alpha_t$ has been used here, and the simulation of the underlying price paths is again done with $\Delta t = 0.01$. The amount N of simulations is dependent on the amount of epochs already performed. The precise scheme for this has been specified in Appendix D of Cuchiero et al.

As mentioned earlier, Cuchiero et al. generated 200 samples of ξ and performed the entire procedure described above for each of these samples. For each ξ, they reported an average computing time of around 27 minutes. They had however the possibility to perform computations on a high-end GPU, while we were limited to mid-tier CPUs. Consequently, performing the whole calibration process for one ξ took approximately a day to complete. Therefore, we decided to reduce, amongst others, the number of used Monte Carlo paths in the synthetic market data generation to 10^4, and the amount of maximum epochs. Moreover, we made some technical modifications in order for the code to be compatible with `Tensorflow v2`. Apart from that, the code we ran is identical to that of Cuchiero et al., which the authors made available by including a link to the GitHub repository in their paper. Although there is a large amount of code needed to implement the ideas mentioned in this report, we have tried to summarise the roles of the most important files in Figure 1.

3.2 Results

Below, we present a number of results from our experiment described in the previous part. To begin with, Figure 2 shows the results of calibrating the leverage function, using randomly generated synthetic market data.

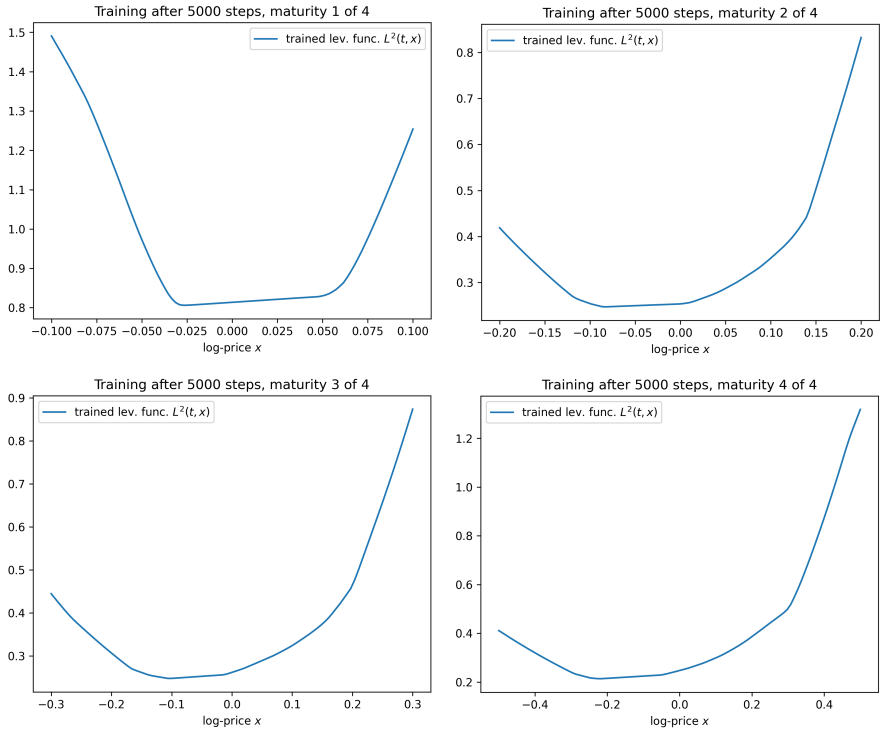

Figure 2: Plots of the calibrated leverage function at different maturities.

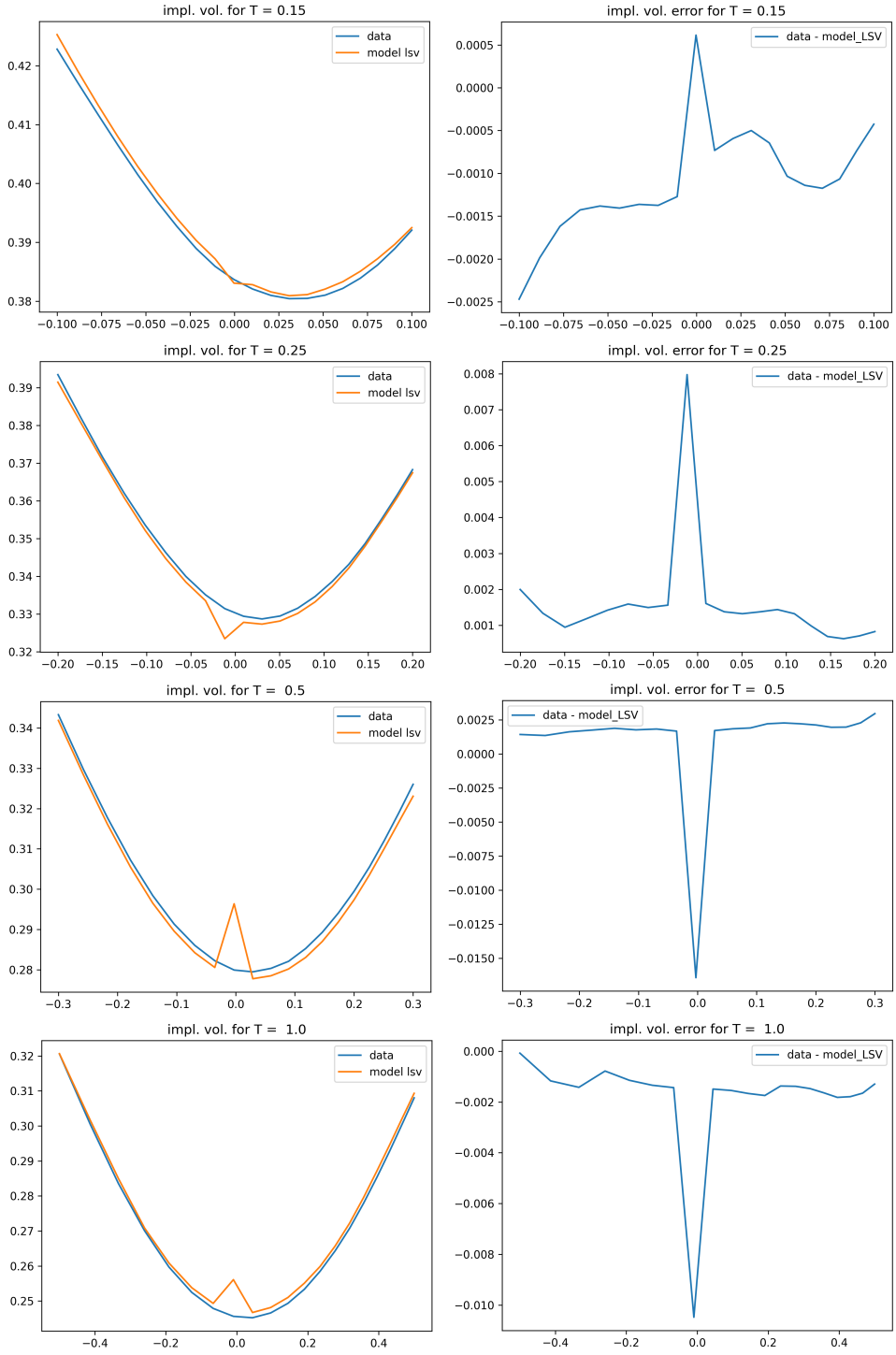

Figure 3: Left: plots of the implied volatilities for the calibrated model together with the synthetic market implied volatilities for different maturities. Right: calibration errors.

Figure 3 shows the final results of the calibration. In the left column we see the implied volatilities for the calibrated model together with the synthetic market implied volatilities for different maturities. In the right column we see the corresponding calibration errors. We see that the maximum absolute error observed in Figure 3 is about a factor 10 higher than in [1].

In comparison with the results in the article of Cuchiero et al., we observe that our calibration, using less trajectories and training steps, is already quite accurate, but performs relatively poor for at-the-money options. The fact that we are using less Monte Carlo runs should imply a decrease in accuracy over all strikes. We think this anomaly is due to the choice of activation functions. As indicated by Cuchiero et al., it seemed that playing around with which activation function to use where, would result in a varying amount of numerical instabilities. It could be that a employing a different activation function in the final hidden layer would give a more regularised result for our current setting.

4 Conclusion

During this course we learnt how machine learning can be used in the financial world. In particular, we studied its applicability in calibration of local stochastic volatility models by using feed-forward neural networks to represent the leverage function. The approach suggested by Cuchiero et al. allowed for an generative-adversarial interpretation. After taking a step back, in which we implemented a single-layer example of a generative adversarial network, we took a deep dive into the details of the machine learning approach to the calibration problem. We were able to reproduce the most important results from the paper, with a slightly lower accuracy due to our restricted computational time.

This all made us appreciate the power and versatility of machine learning in finance. Moreover, it contributed to our understanding of GANs, which still are a relatively recent development in the field of machine learning. Although the paper of Cuchiero et al. showed that GANs can be used to calibrate local stochastic volatility models, they do not fully make use of this generative-adversarial formulation until the very end of their paper. There, they mention one particular setting in which there is a bid-ask spread in the artificial market. This implies that the option price is not unique. One can simulate this generating a discrete number of perturbed parameter sets, leading to a number of different underlying dynamics (and hence different option prices). Here, the adversary would be able to pick the one option price that yields the largest error for the generated leverage function. This can even be generalised to a continuum of option prices, however this would be a recommendation for further research.

While extremely popular in the fields of image generation and enhancement, it remains an interesting question whether GANs really do have advantages for the financial industry yet. As mentioned by Cuchiero et al. themselves, their method has no intention of competing with existing methods. Although this method has a very broad applicability, this might often mean that another, more specialised approach is faster – and hence better suited – for the job.

References

[1] Christa Cuchiero, Wahid Khosrawi, and Josef Teichmann. "A Generative Adversarial Network Approach to Calibration of Local Stochastic Volatility Models". In: *Risks* 8.4 (Sept. 2020), p. 101. DOI: 10.3390/risks8040101.

[2] Bruno Dupire. "Pricing with a Smile". In: 1994.

[3] Antonis Papapantoleon and Chenguang Liu. "Stochastic Gradient Descent". Lecture, as part of the Special Topics in Financial Engineering course at TU Delft. Apr. 2022.

[4] Serrano Academy Luis Guiserrano. *A Friendly Introduction to Generative Adversarial Networks*. YouTube. 2020. URL: https://youtu.be/8L11aMN5KY8.

A Generative Adversarial Networks (GANs)

A generative adversarial network (GAN) is a recent revolution in the world of machine learning, dating from 2014, and is a form of unsupervised machine learning. In a generative adversarial network we make use of two neural networks, which are competing with each other in the form of a zero-sum game. In general, a GAN can be trained, using an existing data set, to generate a new, plausible data set with the same properties as the training set.

The two neural networks a GAN consists of are called the generator and the discriminator, each consisting of weight factors, biased parameters and an activation function. The generator generates new data, after which the discriminator makes a discriminating decision. Using error functions, both the generator and the discriminator are able to tell how good their neural networks perform, after which they both update their neural network.

Example
One of the most common applications of GANs is to generate realistic pictures of people who do not really exist, but look exactly like real persons. To achieve a better understanding of this topic, we will look into an example which trains a single-layer GAN to generate pictures with a certain pattern [4]. Let the following black-and-white pattern of 9x9 pixels be given, in which we can clearly see a diagonal pattern from the top left corner to the bottom right corner (4).

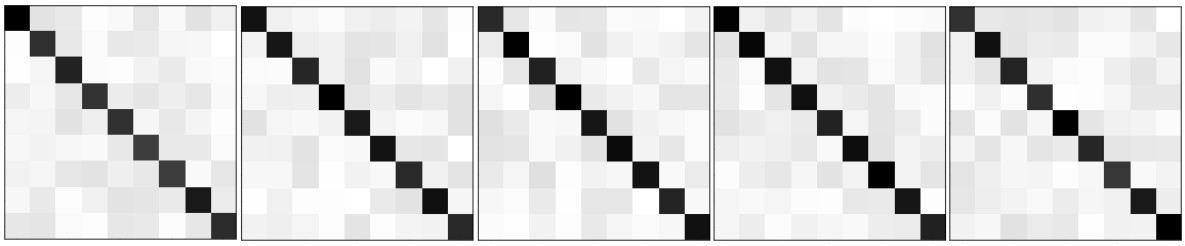

Figure 4: The GAN training set with diagonal patterns of 9x9 pixels.

In Figure 5 we can see how the generator (G) and the discriminator (D) work. To start with, we let $z\ in\mathbb{R}$ be random, after which every pixel of the pattern is determined via the weights, biased parameters and activation function of the generator (equation 10). We let the activation functions be given by the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, which is one of the most used activation functions. It takes values between 0 and 1, so we can define the pattern by setting $\sigma(x) = 1$ as a black pixel and $\sigma(x) = 0$ as a white pixel.

The discriminator uses again weights, a biased parameter and an activation function to output a number between 0 and 1 (equation 11). In the case of the discriminator, we can train the network to use its activation function as a probability density function: $\sigma(x) = 1$ implies we are looking at a real pattern, while $\sigma(x) = 0$ implies that the pattern is a fake one.

$$G(z) = (\sigma(v_1 z + c_1), ..., \sigma(v_{91} z + c_{91})), \tag{10}$$

$$D(x) = \sigma\left(\sum_{i=1}^{91} x_i w_i + b\right). \tag{11}$$

Next, we have to train the generative adversarial network. To be able to tell whether the previous prediction of the discriminator and the previous pattern of the generator were any good, we have to define error functions. When a pattern from the training set is inserted, the discriminator wants to output a 1, so we set the error function $e_{D,real} = 1 -$ prediction. When a generated pattern is inserted, the discriminator wants

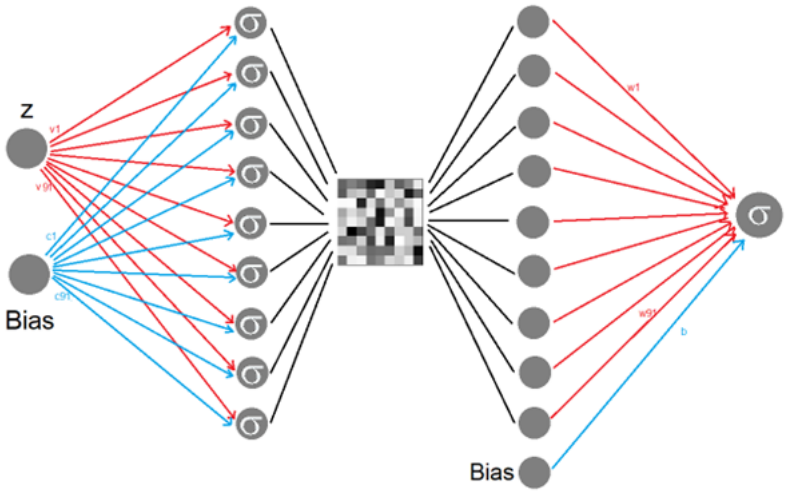

Figure 5: A schematic overview of the single-layer GAN structure, with on the left-hand side the generator and on the right-hand side the discriminator.

to output a 0, so we set this error function $e_{D,fake}$ = prediction. Finally, the generator always aims for a discriminator's prediction of 1, so $e_G = 1 -$ prediction.
In every iteration of the training cyclus, we train the GAN using a real image from the training set and an image generated by the generator. After picking the real image, we apply the discriminator and update its parameters using the gradient descent method. Then we let the generator generate a fake image, after which we apply the discriminator and now update the parameters of both the discriminator and the generator using the gradient descent method.

In this example, it takes about 5.000 training steps (each step consisting of) before the generator is trained good enough to create a pattern which looks like one of the training set. In Figure 6 the performance of the generator is shown after $n = 1.000, 2.000, 3.000, 4.000, 5.000$ training steps.

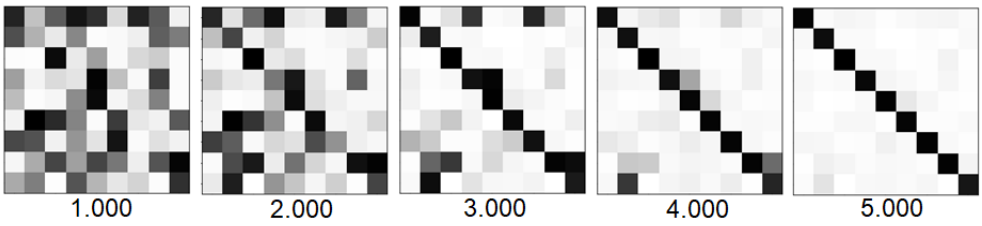

Figure 6: Randomly generated images by the generator, after $n = 1.000, 2.000, 3.000, 4.000, 5.000$ training steps.

Finally, we can evaluate the performances of both the generator and the discriminator by looking at the error plots in Figure 7. On the left, we can see that the error of the generator decreases because of the training of the neural network. In other words, it happens more often that a generated (fake) image is assessed as a real image by the discriminator. Since a generative adversarial network can be seen as a zero-sum game, as we mentioned in the beginning of this appendix, this explains immediately why the error function of the discriminator increases. When the generator performs better, the discriminator's performance becomes worse. However, also the discriminator's neural networks is being trained during the training cyclus, which explains the oscillations in the figures.

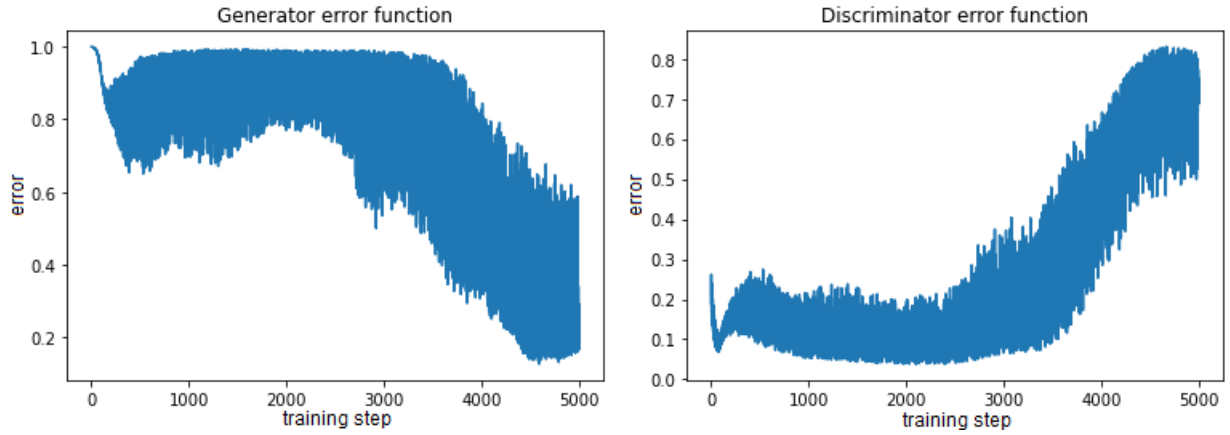

Figure 7: Error functions of the generator and the discriminator after n training steps, for $n = 1, ..., 5.000$.

The Python code used to execute this single-layer GAN-example is given below.

```
import numpy as np
from numpy import random
from matplotlib import pyplot as plt

size = 9

# Plots a square pattern from an array of numbers between 0-1
def view_sample(array):
  size = int(np.sqrt(len(array)))
  plt.imshow(1-array.reshape(size,size), cmap='Greys_r')
  plt.show()

# Creates a number of 'real' patterns (training set)
def correct_face_generator(size):
  mat = np.eye(size).flatten()
  for i in range(size**2):
    if mat[i] >= 0.5:
      mat[i] -= np.random.uniform(0,0.2)
    else:
      mat[i] += np.random.uniform(0,0.2)
  return mat

data_len = 5

faces = []

for _ in range(data_len):
  faces.append(correct_face_generator(size))

# Activation function
def sigmoid(x):
  return np.exp(x)/(1.0+np.exp(x))

class Discriminator():
  def __init__(self):
    self.weights = np.array([np.random.normal() for i in range(size**2)])
```

```
    self.bias = np.random.normal()

  # Apply the discriminator
  def forward(self, x):
    return sigmoid(np.dot(x, self.weights) + self.bias)

  # Error function of the discriminator in case of a real pattern
  def error_from_image(self, image):
    prediction = self.forward(image)
    return 1-prediction

  def derivatives_from_image(self, image):
    prediction = self.forward(image)
    derivatives_weights = -image * (1-prediction)
    derivative_bias = -(1-prediction)
    return derivatives_weights, derivative_bias

  # Backpropagation
  def update_from_image(self, x):
    ders = self.derivatives_from_image(x)
    self.weights -= learning_rate * ders[0]
    self.bias -= learning_rate * ders[1]

  # Error function of the discriminator in case of a fake pattern
  def error_from_noise(self, noise):
    prediction = self.forward(noise)
    return prediction

  def derivatives_from_noise(self, noise):
    prediction = self.forward(noise)
    derivatives_weights = noise * prediction
    derivative_bias = prediction
    return derivatives_weights, derivative_bias

  # Backpropagation
  def update_from_noise(self, noise):
    ders = self.derivatives_from_noise(noise)
    self.weights -= learning_rate * ders[0]
    self.bias -= learning_rate * ders[1]

class Generator():
  def __init__(self):
    self.weights = np.array([np.random.normal() for i in range(size**2)])
    self.biases = np.array([np.random.normal() for i in range(size**2)])

  # Apply the generator
  def forward(self, z):
    return sigmoid(z * self.weights + self.biases)

  # Error function of the generator
  def error(self, z, discriminator):
    x = self.forward(z)
    return 1-y

  def derivatives(self, z, discriminator):
    discriminator_weights = discriminator.weights
```

```
        discriminator_bias = discriminator.bias
        x = self.forward(z)
        y = discriminator.forward(x)
        factor = -(1-y) * discriminator_weights * x *(1-x)
        derivatives_weights = factor * z
        derivative_bias = factor
        return derivatives_weights, derivative_bias

    # Backpropagation
    def update(self, z, discriminator):
        error_before = self.error(z, discriminator)
        ders = self.derivatives(z, discriminator)
        self.weights -= learning_rate * ders[0]
        self.biases -= learning_rate * ders[1]
        error_after = self.error(z, discriminator)

# Set random seed
np.random.seed(42)

# Hyperparameters
learning_rate = 0.01
epochs = 1000

# The GAN
D = Discriminator()
G = Generator()

# Initialising for the error plots
errors_discriminator = []
errors_generator = []

fake_array = np.zeros((epochs, size**2))

for epoch in range(epochs):
    for face in faces:
        # Update the discriminator weights from the real image
        D.update_from_image(face)

        # Pick a random number to generate a fake image
        z = random.rand()

        # Build a fake image
        noise = G.forward(z)
        fake_array[epoch] = noise

        # Calculate the discriminator error
        errors_discriminator.append(0.5*D.error_from_noise(noise)
        +0.5*D.error_from_image(face))

        # Calculate the generator error
        errors_generator.append(G.error(z, D))

        # Update the discriminator weights from the fake image
        D.update_from_noise(noise)

        # Update the generator weights from the fake image
```

```
    G.update(z, D)

ilist = np.arange(1, 10+1)*epochs/10 - 1
for i in ilist:
  view_sample(fake_array[int(i)])

plt.plot(errors_generator)
plt.title("Generator error function")
plt.legend("gen")
plt.show()
plt.plot(errors_discriminator)
plt.legend('disc')
plt.title("Discriminator error function")
plt.show()
```

B Variance Reduction using Control Variates

Let us take a step back from the specific setting we have been working in so far in Section 2. Consider a finite time horizon $T > 0$, an underlying with discounted price process $(S_t)_{t\in[0,T]}$ on some filtered probability space. This process is assumed to be a square-integrable, càdlàg martingale. Furthermore, consider the random variable C representing a European option payoff $C := g(S_T)$ with underlying S and maturity T.

If we were to price this option, we could perform a Monte Carlo simulation, generating N realisations of the price process, and come up with the estimator $\pi = \sum_{n=1}^{N} C_n$ for the option price, where $C_i = g(S_T^{(i)})$ is the option price for the i-th simulated path of S, and has of course the same distribution as C, and is i.i.d. with the other C_j's ($j \neq i$). Let us now, in attempt to reduce the variance of this estimator, introduce a yet unspecified strategy $(h_t)_{t\in[0,T]} \in L^2(S)$ and a yet unspecified constant $c \in \mathbb{R}$. We denote by I the stochastic integral $I := (h \bullet S)_T = \int_0^T h_t \mathrm{d}S_t$, and correspondingly the i.i.d. random variables $I_1, \ldots, I_N$ by replacing S with $S^{(1)}, \ldots, S^{(N)}$ respectively. Now consider the estimator

$$\hat{\pi} = \frac{1}{N} \sum_{n=1}^{N} (C_n - cI_n), \tag{12}$$

and realise that $\mathbb{E}[\hat{\pi}] = \mathbb{E}[\pi] - \frac{c}{N} \sum_{n=1}^{N} \mathbb{E}[I_n] = \mathbb{E}[\pi]$, as the expectation of the stochastic integral is zero. Thus, $\hat{\pi}$ is also a good estimator for the option price. Moreover, if we define $H := \frac{1}{N} \sum_{n=1}^{N} I_n$, we see that the variance of $\hat{\pi}$ is given by:

$$\mathrm{Var}(\hat{\pi}) = \mathrm{Var}(\pi) - 2c\mathrm{Cov}(\pi, H) + c^2 \mathrm{Var}(H),$$

which becomes minimal and equal to $(1 - \rho_{\pi,H}^2)\mathrm{Var}(\pi)$, if we pick $c = \frac{\mathrm{Cov}(\pi,H)}{\mathrm{Var}(H)}$, where $\rho_{\pi,H} = \frac{\mathrm{Cov}(\pi,H)}{\sqrt{\mathrm{Var}(\pi)\mathrm{Var}(H)}}$ is the correlation between π and H. This implies that if we are able to find a strategy $(h_t)_{t\in[0,T]}$ such that π and H, or equivalently C and I, are strongly correlated, we can significantly reduce the variance of $\hat{\pi}$. Thus, we are looking for a suitable hedging strategy that can replicate the option payoff.

Two approaches for this are given in the paper. The first one is the delta-hedging strategy. We simplify the problem by looking at the claim $C = g(S_T)$ and considering instead its Black-Scholes price $\pi_{BS}^g(t, T, s, \sigma)$ at time t. If we now choose a hedging strategy with as only hedging instrument the price S and with $h_t = \partial_s \pi_{BS}^g(t, T, S_t, L(t, S_t)\alpha_t)$, quite significant variance reduction is reached already. Delta-hedging will not perfectly replicate the option of course, due to the different underlying dynamics, but it will certainly yield a strong correlation between C and I. Since h_t is $\mathcal{F}_t$-measurable, and relatively easy to compute, this is a very appealing approach already. One can even simplify computations even further by considering instead the strategy $h_t = \partial_s \pi_{BS}^g(t, T, S_t, \alpha_t)$. Notice the important trade-off between accuracy of the hedging

strategy, and computational cost. A less accurate strategy will be cheaper to compute, and might already yield a significant variance reduction such that more accurate strategies are often not needed.

The second approach uses neural networks in order to parameterise the hedging strategy. Assuming the hedging strategy is Markov, i.e. only depends on the current time and the current underlying price, we can write $h_t = h(t, s)$ as a neural network mapping (t, s) to $h(t, s; \delta)$, where $\delta \in \Delta$ denotes the parameters of the neural network. The machine learning approach to find the optimal hedging strategy for the option payoff C with a given market price π^{mkt} would be to solve the following minimisation problem

$$\inf_{\delta \in \Delta} \mathbb{E}\big[u\big(-C + \pi^{\text{mkt}} + (h(\cdot, S_\cdot; \delta) \bullet S_\cdot)_T\big)\big] \tag{13}$$

where $u : \mathbb{R} \to \mathbb{R}^+$ is some convex loss function. This expectation can again be estimated using a Monte Carlo simulation. Note that in contrast to the optimisation problem in (7), this one can be solved with SGD, as it is of the form $\inf_{\delta \in \Delta} \frac{1}{N} \sum_{n=1}^{N} u(Q(\delta; x_n))$. This will yield an optimal strategy $h(\cdot, \cdot; \delta^*)$. Note that in using SGD, some technicalities regarding smoothness and moving around gradients and stochastic integrals should be justified. For this we refer to the paper by Cuchiero et al [1].