ChatGPT

# Deep Volatility Arbitrageur: Neural SDEs for Arbitrage-Free Option Pricing

## 1. Theoretical Framework & Financial Logic

**Risk-Neutral Generative Networks (RNGN) – Stochastic Log-Return Curves:** A core idea of the Deep Volatility Arbitrageur prototype is to model the entire **risk-neutral distribution** of an asset's returns across maturities using a Neural SDE-based generative model. In practice, **Risk-Neutral Generative Networks (RNGNs)** treat the **log-returns** as a *stochastic function of time-to-maturity*. Formally, one assumes that the log-return $X(T)$ (for maturity $T$) is given by a deterministic function of two inputs: a standard normal random variable $Z$ (the source of randomness) and the time-to-maturity $T$ [1] [2]. In other words, the entire *term-structure of log-returns* is represented as a **stochastic curve** $X(T) = f(Z,\;T)$ where $f$ is parameterized by neural networks. This construction cleanly **separates the randomness** (captured by $Z$) from the **term-structure shape** (captured by neural network functions of $T$) [3]. By doing so, the model imposes an explicit functional form under the risk-neutral measure while remaining highly flexible to fit data. Crucially, the network outputs (e.g. location, scale, skewness term-structures of $X(T)$) are trained under **no-arbitrage constraints** so that the generated option prices are free of static arbitrage [3]. In particular, the RNGN approach **imposes six standard static arbitrage constraints** during training, ensuring that the learned price surface obeys all the model-independent no-arbitrage conditions. These include: (1) **monotonic decreasing call price in strike** (no call spread arbitrage), (2) **convexity in strike** (no butterfly arbitrage), (3) proper zero-value limits as strike → 0 or ∞, (4) degenerate distribution as $T \to 0$ (option price equals intrinsic value at expiry), (5) **monotonic increasing in time-to-maturity** (no calendar arbitrage), and (6) correct pricing bounds consistent with forward pricing (put-call parity and upper/lower bounds on option prices) [4] [5]. By construction, RNGNs *satisfy conditions 1–4 exactly* (e.g. using a non-negative density representation and enforcing a Dirac delta in the $T\to0$ limit), and enforce condition 5 via a monotonicity regularization term during training [4]. Condition 6 is ensured by anchoring the model to the risk-neutral martingale constraint (the expected future underlying price equals the forward price) [6] [7]. **In summary, the RNGN provides a generative, arbitrage-free model of the entire implied volatility or price surface**: given a sample of $Z$, one can generate a full curve $X(T)$ and thus compute option prices across strikes and maturities. This method is *risk-neutral by design* and **significantly outperforms traditional parametric models** in flexibility and accuracy [8]. Experiments have shown that a well-trained RNGN can fit market option prices with far lower error than classical models (e.g. outclassing several parametric and stochastic process benchmarks) [8], thanks to its ability to capture complex term-structure effects (skew, kurtosis term structures, multiple modes, etc.) while rigorously avoiding static arbitrage.

**Dual-Network Architecture – PAN and CCN:** Another key component of the prototype is a *two-stage neural network architecture* designed to enhance pricing accuracy and calibration robustness. This consists of: (a) a **Price Approximator Network (PAN)** and (b) a **Calibration Correction Network (CCN)** [9]. The **PAN** is a supervised feed-forward network trained to **learn the option price surface** (or implied volatility surface) *directly from market data*. It takes contract features (e.g. moneyness/strike and time-to-maturity) as inputs and outputs the option's model price, essentially acting as a high-dimensional interpolator or "surface

generator." By training on observed prices, the PAN produces a smooth, arbitrage-free approximation of the implied volatility or price surface [10]. In doing so it captures the broad shape of the market surface (volatility smiles, skews, term structure) without being tied to a particular parametric model. Next, the **CCN** is introduced to refine and correct any residual errors when using a simpler base model like Heston. Specifically, the **Calibration Correction Network** takes as input the initial option prices predicted by a calibrated Heston model (or another parametric model) *together with* the outputs of the PAN, and it learns to **adjust the Heston prices towards the true market prices** [9] [10]. In essence, CCN focuses on the *systematic pricing errors* of the parametric model (for example, Heston may misprice deep out-of-the-money options or short maturities), and it outputs a correction term to apply to those prices. By composing these two networks – first approximating the general surface, then correcting model-specific biases – the framework marries the **interpretability of Heston** with the **flexibility of deep learning** [11]. The **dual-network approach eliminates "static arbitrage" issues** as the PAN can be trained under no-arbitrage regularizers, and the CCN's corrections are learned in a way that preserves arbitrage conditions (e.g. by only making small smooth adjustments). The result is a calibrated model that fits market prices extremely well while still maintaining a theoretical backbone (the Heston model) for dynamic consistency. This approach directly addresses the difficulties of traditional calibration: instead of optimizing Heston parameters via iterative methods like Levenberg–Marquardt (which can be slow and get stuck in local minima), the networks learn the mapping from market data to prices in one training phase [12] [13]. **Empirically, this two-stage Deep Learning calibrator has been shown to outperform classical calibration across all error metrics** [14]. For example, on S&P 500 index options, the PAN+CCN framework achieved an out-of-sample pricing RMSE an order of magnitude smaller than that of a standard Heston calibration via Levenberg–Marquardt [15]. In one study, the traditional calibration had ~$10$ units of RMSE error on test data, whereas the deep-learning-enhanced approach brought that down below $1$ [15]. This was accompanied by dramatically lower mean absolute errors and percentage errors, indicating a much closer alignment with market prices [14]. The **faster convergence and better generalization** of the PAN+CCN approach means it can calibrate in real-time scenarios and avoid mispricings that would otherwise present arbitrage opportunities. By **learning the volatility surface and model corrections directly from data**, the Deep Volatility Arbitrageur can **eliminate static arbitrage** (by construction) and capture complex market patterns, giving it a clear edge over traditional single-model calibration approaches [10] [16].

**Eliminating Static Arbitrage & Outperforming Traditional Calibration:** The combination of RNGN-style stochastic curve modeling and the PAN/CCN networks yields a framework that inherently respects all static arbitrage constraints while providing superior calibration accuracy. The **"static arbitrage" problem** refers to violations of surface consistency that allow immediate risk-free profit (e.g. a butterfly spread yielding guaranteed gain if the implied vol surface isn't convex). By enforcing Constraints 1–6 (outlined above) either *explicitly* or via penalty terms, the prototype ensures the learned pricing function is arbitrage-free *by design*. For instance, because the log-return curve $X(T)$ is represented via an explicit density-driven model in RNGN, **call-spread and butterfly arbitrages are automatically precluded** (the call price will be a non-increasing and convex function of strike since it's computed as an integral over a valid density) [17] [18]. Calendar spread arbitrage is avoided by construction or penalized heavily if any violation occurs (prices at longer maturities are never lower than at shorter ones) [4]. These built-in guarantees mean the **network never produces a nonsensical implied vol surface** (a common headache when calibrating parametric models). In contrast, a raw Heston calibration via numerical optimization might find a parameter set that fits most of the market well but introduces a slight arbitrage (e.g. a hump in the volatility smile violating convexity), requiring additional adjustments or scrapping the result. The Deep Volatility Arbitrageur sidesteps that by **baking no-arbitrage into the neural architecture and loss function**, so that any candidate solution that violates arbitrage is immediately disfavored during training. Furthermore, by

leveraging deep networks' ability to fit complex, non-linear mappings, the prototype **captures features that a five-parameter Heston model cannot** (such as varying skewness/kurtosis term-structure or multiple volatility regimes). This leads to **markedly better calibration performance**: as noted above, error metrics like RMSE, MAE on both training and testing data improve by an order of magnitude relative to traditional calibration [14]. Studies have documented not only tighter in-sample fits but also better **out-of-sample generalization**, meaning the networks can predict option prices at strikes/maturities not seen in training more accurately than a calibrated Heston can [19] [20]. In practice, this equates to more reliable pricing of exotic or illiquid options and more robust risk management. The **Levenberg–Marquardt (L-M) algorithm** and similar local optimizers, while commonly used for model calibration, often struggle with the Heston model's rough error surface and can be very slow (running many iterations of a PDE or characteristic function solver). The neural approach **reduces calibration to a single forward pass (or a few passes)** of a trained network, yielding near-instantaneous pricing once the model is trained. In summary, the theoretical advances of the Deep Volatility Arbitrageur lie in combining risk-neutral theoretical rigor (no-arbitrage SDE modeling) with modern deep learning, delivering a model that **respects financial logic (martingale pricing, arbitrage constraints)** and achieves superior accuracy and speed compared to legacy calibration methods [13] [16].

## 2. Developer Implementation Guide

**Framework and Model Implementation (PyTorch):** From a developer's perspective, implementing the Deep Volatility Arbitrageur prototype can be done efficiently using modern deep learning libraries – **PyTorch** is recommended for its flexibility and intuitive style [21]. PyTorch's dynamic computation graph and auto-differentiation make it straightforward to define complex custom architectures (for example, multi-input networks for PAN or SDEs with neural drift/diffusion) and to compute gradients for optimization or Greeks. The prototype can be broken into modules – e.g., one module for the **Neural SDE generator** (which produces log-return samples or density parameters as a function of $T$), another for the **Price Approximator Network** (a feed-forward net mapping $(T, K)$ or $(\text{moneyness}, T)$ to an option price or implied vol), and a third for the **Calibration Correction Network** (which could take Heston parameters or preliminary prices as input and output adjustments). In PyTorch, one would define these as subclasses of `nn.Module`, compose them as needed (e.g., CCN takes both PAN output and other features), and use the library's automatic differentiation to compute loss gradients. PyTorch's strength lies in enabling dynamic, imperative model definitions – for example, you could have the Neural SDE simulate paths on the fly inside the forward pass, or incorporate constraint-enforcing layers that modify network outputs before pricing, all while still being able to backpropagate through the whole system [22] [23]. This flexibility is crucial for applying **financial constraints**: we can write custom forward passes that ensure, say, a positive volatility or enforce monotonicity by construction. The PyTorch ecosystem also provides tools like `torch.autograd` for Jacobian/Hessian computation (useful for computing hedge ratios or calibration sensitivities), and seamless GPU acceleration for handling large simulated datasets. For example, training might involve generating millions of synthetic option prices (as described below) – using PyTorch with a GPU allows this heavy number-crunching to be done in parallel and efficiently. In summary, PyTorch offers **an ideal environment for this project**, as it did in related research: it's widely adopted in both research and industry, and its *imperative, Pythonic interface* makes rapid prototyping and iterative model improvements convenient [21]. Developers can leverage PyTorch's rich library of neural network components (activations like Softplus for positive outputs, etc.) and optimizers (Adam, LBFGS, etc.) to experiment with architectures and training regimes for the Arbitrageur model.

**Loss Functions and No-Arbitrage Constraints:** Implementing the Deep Volatility Arbitrageur requires carefully designed **loss functions** that incorporate both data fit and no-arbitrage conditions. The overall training objective can be thought of as: **Loss = Pricing Error + Penalty for Arbitrage Violations**. The *pricing error* term is straightforward – e.g. mean squared error (MSE) between the network's predicted option prices and the observed market prices (or synthetic "ground truth" prices in pre-training) [24] [25]. The arbitrage penalties enforce the constraints that guarantee an arbitrage-free surface:

- **Monotonicity in Strike:** For any fixed maturity $T$, call option prices should be non-increasing as strike $K$ increases. To enforce this, one can add a penalty term for any violation of $C(T,K_1) \ge C(T,K_2)$ when $K_1 < K_2$. A simple implementation is to sample a pair $(K_1, K_2)$ in each mini-batch (or enforce across a grid) and compute `penalty += ReLU( C_pred(T,K2) - C_pred(T,K1) )` – this term will be zero when the condition holds and positive if a violation occurs. Summing such terms over many random strike pairs encourages the network to learn a non-increasing mapping $K \mapsto C$ [26]. This guarantees **no call-spread arbitrage** (Constraint 1), since it prevents creating dominant spreads.

- **Convexity in Strike:** Call prices should be convex functions of strike (ensuring no butterfly arbitrage). We can enforce convexity by penalizing any negative second difference. For example, given three strikes $K_1 < K_2 < K_3$, one can check the inequality $C(K_2) \le \frac{K_3 - K_2}{K_3 - K_1} C(K_1) + \frac{K_2 - K_1}{K_3 - K_1} C(K_3)$ (which is equivalent to convexity). In practice, one can take finite differences: $C(K_2)$ should lie below or on the line between $C(K_1)$ and $C(K_3)$. A penalty can be `ReLU( C_pred(K_mid) - (w * C_pred(K_low) + (1-w) * C_pred(K_high)) )` for appropriate weights $w$. If the network outputs are twice differentiable, an alternative is to include a term like $\lambda \max(0,\; -\frac{\partial^2 C}{\partial K^2})$ averaged over random points (penalize any negative second derivative). In the RNGN approach, this convexity was automatically satisfied by modeling a valid density (implying $\partial^2 C/\partial K^2 = e^{-rT} P(X_T=K) \ge 0$) [27]. For a more general network, we enforce it via such loss terms or by constructing the network output to be convex (e.g. using an **Input Convex Neural Network** architecture for the strike input [28]).

- **Monotonicity in Time-to-Maturity:** Option prices should **increase with maturity** $T$ (at least weakly, for non-dividend underlying), since having optionality for longer can't make the option less valuable (Constraint 5). We enforce no calendar spread arbitrage by penalizing any instance where $C(T_2,K) < C(T_1,K)$ for $T_2 > T_1$. Similar to strikes, one can sample pairs of maturities or enforce along the term structure: `penalty += ReLU( C_pred(T1) - C_pred(T2) )` if $T_2 > T_1$. The network will learn to produce surfaces that are **monotonic along the maturity axis**. In practice, this condition may sometimes be mildly violated in real data (due to noise), so one often imposes it as a **soft constraint** (regularization term) with a smaller weight, just enough to guide the network [4]. This ensures **no calendar arbitrage**.

- **Boundary Conditions (Intrinsic Bounds):** We incorporate known boundary limits as part of the loss to anchor the network's behavior in extreme regions. For very low strikes, a call's price approaches $(S_0\,e^{-qT} - K e^{-rT})^+$ (for underlying spot $S_0$ and dividend yield $q$) and for very high strikes it approaches $0$. We can generate some extreme strike samples (or include the analytic limits) and penalize the squared error between the network's output and these boundary values. For example, enforce $C(T, K_\text{max}) \approx 0$ and $C(T, K_\text{min}) \approx S_0 e^{-qT} - K_\text{min} e^{-rT}$ (if positive, else 0). Likewise for put options, ensure $P(T,K_\text{min}) \approx

0$ and as $K \to \infty$, $P$ approaches $K e^{-rT} - S_0 e^{-qT}$ (or 0 for finite large K given payoff caps). These ensure **Constraint 3** (options worthless at extreme strikes) [29] and are easy to build into the training (they can even be hard-coded as output layers for the extremes).

- **Martingale (Forward Price) Condition:** Although the network primarily prices options, we should ensure consistency with the underlying asset's forward price. In risk-neutral pricing, the **forward price** $F(0,T) = S_0 e^{(r-q)T}$ should equal the expectation of the future price under the risk-neutral measure. A practical check is enforcing **put-call parity** in the network's outputs: $C(T,K) - P(T,K) = S_0 e^{-qT} - K e^{-rT}$. If the network is only pricing calls (or only vols), put-call parity can be used as a post-process, but if it prices both, one can add a penalty for deviation from parity. Additionally, one can penalize any deviation from the equality $C(T,0) = S_0 e^{-qT}$ and $C(T,\infty)=0$, which along with convexity gives the upper bound $C(T,K) \le S_0 e^{-qT}$ and lower bound $C(T,K)\ge \max(0, S_0 e^{-qT}-K e^{-rT})$ [30] [7]. In practice, many of these inequalities are automatically satisfied if the above conditions hold, but including them in the loss (with small weight) can help stability.

All these constraints can be incorporated into the **loss function** as additional terms with tunable weights (hyperparameters). During training, the model then tries not only to fit the data but also to keep these penalties near zero. The result is a learned surface that is arbitrage-free. We note that one can also enforce many of these constraints in the **architecture itself** ("hard constraints"), which can be even more effective. For example, one approach is to have the network output intermediate unconstrained values (like raw drift or volatility factors) and then pass them through constraint-enforcing transformations. An illustration of this technique is shown below, where custom layers $\mathcal{G}*mu$ and $\mathcal{G}*\sigma$ transform the raw neural network outputs into arbitrage-free factor dynamics [31]:

*Embedding no-arbitrage in network architecture:* An example of a constrained neural network design, where the raw outputs (e.g. drift $\hat{\mu}$ and volatility $\hat{\sigma}$ from the NN) are passed through deterministic functions $\mathcal{G}*mu,\;\mathcal{G}*\sigma$ that ensure the resulting process stays within the **statically arbitrage-free** region (polytope). This guarantees the trained Neural SDE respects no-arbitrage constraints by construction [31].

In code, implementing such hard constraints could involve using **positive activation functions** (e.g. Softplus or ReLU to ensure densities/variances $\ge0$), **monotonic network architectures** (like monotonic splines or isotonic layers for enforcing ordering in strikes/maturities), or layering neural outputs with analytical formulas (like using an output as a parameter in an option pricing formula known to be arbitrage-free). For instance, one could output implied density parameters and integrate them to get prices, rather than output prices directly – this way arbitrage constraints 1 and 2 are inherently satisfied. In summary, the developer should combine *data-driven loss terms* and *architectural tricks* to ensure the model learns an arbitrage-free surface. The loss function might look like:

$$L_{\text{total}} = \text{MSE}(\hat{C}, C^{\text{market}}) + \lambda_{\text{mon-}K} L_{\text{mon-}K} + \lambda_{\text{convex}} L_{\text{convex}} + \lambda_{\text{mon-}T} L_{\text{mon-}T} + \lambda_{\text{bndry}} L_{\text{bndry}},$$

with appropriately chosen weights $\lambda$. Tuning these weights is important – too high, and the model might sacrifice fit excessively to satisfy constraints; too low, and it may learn spurious arbitrage (which you'd detect and correct). A practical tip is to gradually increase the penalty weights during training ("educating" the network to respect finance rules after it captures the general shape).

**Synthetic Data Generation for Pre-Training:** A significant challenge in training such deep models is obtaining enough data covering the full variety of market conditions. Here, a *model-driven data generation* approach is very useful: we **simulate synthetic option prices from known stochastic models** (like Heston or Rough Bergomi) to create a rich training dataset. The idea is to leverage these models to generate "realistic" arbitrage-free option surfaces under diverse parameter settings, and use those as training examples so that the neural networks can learn the mapping from model parameters (or market states) to prices. For example, consider the **Heston model** (a 5-parameter stochastic volatility model). One can sample thousands of random parameter sets $(v_0, \theta, \kappa, \xi, \rho)$ from reasonable distributions (ensuring they produce plausible vol smiles), and for each parameter set, compute the **European call prices** $C_{\text{Heston}}(T,K)$ across a grid of strikes and maturities. Heston has a semi-analytical pricing formula (via Fourier transform), so this can be done relatively efficiently. Similarly, for the **Rough Bergomi (rBergomi) model**, which lacks closed-form solutions, one can perform Monte Carlo simulations for a given parameter set to generate option prices (though slow, this is offline data generation). Each such simulation yields a labeled dataset: input (model parameters, $K$, $T$) and output (option price or implied vol). By doing this for tens of thousands of parameter sets, we build a comprehensive synthetic dataset of option surfaces. This **offline generation** step has been employed in research to great effect [32]. For instance, one study generated a million rBergomi samples and trained a neural net to serve as a **surrogate pricer** for rBergomi options [33] [32]. In our case, we can generate data from *both* a fast model like Heston and a more complex model like Rough Bergomi, to expose the network to a wide range of smile dynamics (Heston covers basic skew but Rough Bergomi introduces "rough" volatility and steep short-term skew). We might also add samples from Black-Scholes (as a sanity check limit case) or other models (jump diffusion) to further enrich the training distribution.

The **purpose of pre-training on synthetic data** is to give the neural networks a "head start." They learn the fundamental relationships between inputs and option prices: how changing volatility of volatility affects the smile, how correlation ($\rho$) generates skew, how time-to-maturity changes affect variance, etc. The networks also implicitly learn arbitrage boundaries from these model-generated surfaces, since all model prices are arbitrage-free. This pre-training can be done with a simple objective (e.g. MSE between network output and model price) since the synthetic data is noise-free. Once the networks (PAN and possibly parts of the RNGN) are trained on extensive synthetic data, we can **fine-tune** them on actual market data. Fine-tuning will adjust the networks to capture real-market peculiarities (which may not be perfectly captured by Heston or rBergomi), while starting from a sensible parameter region. This two-phase approach (offline simulation + online calibration) is a *proven strategy in deep calibration research* [32] [34]. It dramatically speeds up the calibration: instead of running Monte Carlo inside an optimizer, you do it once offline, train the net, and then **calibration becomes a fast inference task**. One can even use the trained surrogate in an optimizer to find best-fit parameters if needed (as done with neural network-assisted Levenberg–Marquardt in rough vol calibration [35]). In our context, because the goal is to directly price options and not necessarily recover the exact model parameters, we might not need an explicit optimizer at all – the trained network **directly outputs prices consistent with the market**. To generate synthetic data for training, here are concrete steps a developer can follow:

1. **Choose parameter distributions:** Decide ranges (or distributions) for Heston parameters (ensure Feller condition for realism, etc.) and for Rough Bergomi parameters (e.g. $H$ in $(0,0.5)$, $\eta$ vol-of-vol, etc.). Sample $N$ sets of parameters from these ranges (where $N$ could be on the order of thousands or more, subject to computing resources).

2. **Generate prices per parameter set:** For each sampled parameter set, generate option prices for a grid of maturities (e.g. 1M, 3M, 6M, 1Y, 2Y) and strikes (e.g. moneyness from 0.5 to 1.5 or whatever range covers OTM puts/calls). For Heston, use its closed-form characteristic function pricing formula to compute prices or implied vols. For Rough Bergomi, run a Monte Carlo simulation (or use an existing library) to get prices for the given strikes/maturities – this is the slowest part, but you can parallelize simulations or use fewer parameter samples here due to cost.

3. **Build training set:** Organize the data in a suitable format. If training the **Price Approximator Network**, inputs might be $(T, K, \text{other conditions})$ and output the price. If training a network to *emulate the model* (like mapping model parameters + $(K,T)$ to price), include the parameters in the input vector as well. In the Arbitrageur prototype, since the network ultimately should price given market state (which is the underlying and perhaps an implied vol level), we might not include model parameters explicitly at inference – the network's weights will have absorbed those effects. So a practical approach is to **train the PAN simply on $(K,T)$ to price mapping for many different surfaces**, thereby learning a *universal pricer* that can handle varied shapes. One can mix all the synthetic surfaces together during training, or even better, use each surface as a separate "batch" and train the network to minimize error across all of them (effectively learning to interpolate/extrapolate among different volatility shapes).

4. **Validation:** Set aside some generated surfaces as a validation set to ensure the network isn't just memorizing. You'd want to see it generalize to new parameter sets.

By the end of this process, you'll have a neural network that can produce option prices for strikes and maturities in **microseconds**, whereas doing so via Monte Carlo (especially for the rough model) would take *seconds per evaluation*. Literature reports show that after such training, a full implied volatility surface calibration that normally would take minutes via simulation can be done in on the order of **tens of milliseconds** using the neural network surrogate [36] [37]. Indeed, one study noted that a rough volatility model calibration which required numerous MC simulations could be accelerated to ~36ms total using a trained neural network in place of the simulator [37]. This highlights the benefit of the data-generation approach: a heavy upfront cost to simulate/train, but then lightning-fast inference. In implementing this, a developer should also be mindful of **data normalization** (scale inputs like strike and maturity to reasonable ranges, perhaps input moneyness $\log(K/S_0)$ instead of raw $K$, etc.) and include **random noise or perturbations** in training data if needed to improve generalization (since real market data is noisy). Using PyTorch, one can easily create a `Dataset` class for the synthetic data and leverage `DataLoader` for batching. Training is then a matter of running standard `.backward()` and optimizer steps. With a well-designed network and a sufficiently rich synthetic dataset, the model will be well-prepared to fit real market surfaces with minimal further adjustment.

## 3. Validation and Use-Cases

**Inference Speed: Neural SDE vs. Monte Carlo:** A major advantage of the Deep Volatility Arbitrageur approach is the **orders-of-magnitude speed-up** in pricing and calibration once the neural networks are trained. Traditional Monte Carlo or PDE methods for option pricing are computationally intensive – e.g., pricing a single exotic option via Monte Carlo might require $10^5$–$10^6$ random path simulations, and calibrating a model means repeating such simulations for many iterations of an optimizer. In contrast, a trained neural network (whether it's the RNGN model or the PAN surrogate) produces option prices via a simple forward pass, which is essentially a series of matrix multiplications and non-linear function

evaluations. This is typically *milliseconds or faster*. To concretely compare: imagine we want to compute an entire implied volatility surface (multiple strikes and maturities). Using Monte Carlo, we would have to simulate each maturity's payoff scenarios – often a very time-consuming process for low error. Using the neural approach, we input the strikes and maturities to the network and get *all prices at once* (since modern frameworks can do batch predictions). **Studies have shown neural network pricers can evaluate a full surface in a few milliseconds** [38], whereas a Monte Carlo might take minutes to achieve comparable accuracy. For example, Buehler et al. (2019) reported that their deep neural network approach could calibrate to an entire implied vol surface in ~5ms, versus the much longer times for traditional methods [38]. Our own rough-vol surrogate example above cited ~36ms for a full calibration of rBergomi via network, compared to potentially hours if one did a brute-force search with MC [37].

To validate the inference speed empirically, a developer can **benchmark** the following: Take a set of options (say 100 strikes x 10 maturities = 1000 options). Time how long it takes to price all 1000 with a traditional Monte Carlo pricer for a complex model (this could easily be tens of seconds, depending on simulation count). Then time the neural network pricing the same 1000 (likely under 0.01 seconds). The ratio of these times is the speedup (often in the hundreds or thousands). Another test is calibration: if calibrating Heston via L-M takes, say, 50 iterations and each iteration uses a numerical integrator for 1000 prices (taking 0.1s each), that's 5 seconds total. The neural calibrator would effectively do it in one shot or a few shots, like 0.005s, achieving a **~1000x speed-up**. This is not just academic; it enables new practical capabilities. For instance, desks can recalibrate models continuously in near real-time as market data comes in, or run thousands of scenarios of calibrations (for uncertainty quantification) which would be infeasible with classical methods. The trade-off is the **one-time training cost** – but training can be done offline (e.g., overnight or on GPUs in the cloud) and need not be repeated too often (one can update the network gradually with new data).

Another aspect of validation is to compare the **pricing accuracy** of the Neural SDE vs Monte Carlo. After training, one should test the neural model on a set of benchmarks: e.g., price a variety of options with the neural model and with a very high-precision Monte Carlo or analytical method, and ensure the errors are within acceptable tolerance (often, networks can achieve errors on the order of 0.1% of option prices or better [14]). If any outliers exist (perhaps in extreme tail scenarios), those can be addressed by incorporating more such cases in training or by enforcing stricter constraints there. The bottom line: **the neural approach drastically reduces inference time** while maintaining high accuracy, which is a critical validation point. This speed advantage grows when considering high-dimensional problems (like multiple underlying assets or American options, where simulation is particularly slow). In our Arbitrageur's context of **Neural SDE**, once the drift and diffusion networks are calibrated, simulating *one path* of the SDE is extremely fast (just applying SDE integrator with cheap neural function evaluations per step) and can be done in parallel for many paths on GPU. We can thus generate scenario distributions for risk management very quickly as well, something Monte Carlo would struggle with if it required a full recalibration each time. Indeed, deep learning methods have been highlighted for **significantly reducing computational burdens of Monte Carlo** in option pricing [39].

**"Deep Hedging" – Optimal Delta and Vega Hedging via the Trained Network:** Beyond pricing, a trained Deep Volatility Arbitrageur can be leveraged for **hedging strategy optimization**, often referred to as **Deep Hedging** in recent literature [40]. The term "deep hedging" (coined by Buehler et al., 2019) means using machine learning (often reinforcement learning or deep neural nets) to find optimal hedging strategies in incomplete markets, rather than relying on traditional Greeks-based hedging alone. In our context, we can

use the trained neural SDE model as a *market simulator* and compute hedging policies (particularly Delta and Vega hedges) that minimize risk. Here's how the Arbitrageur aids hedging:

- **Greeks via Autodiff:** First, the network provides easy access to **sensitivities** of option prices. Since our pricing function is differentiable, we can compute **Delta** ($\partial$ option/$\partial$ S) by simply taking the derivative of the network output w.rt the underlying price input. If the network takes moneyness as input, Delta can be obtained by differentiating w.rt that or by automatic differentiation through the SDE (which might involve differentiating through $\mathcal{G}*mu,*mathcal{G}*\sigma$ layers or similar). Similarly, **Vega** (sensitivity to volatility) can be derived. If the model has an explicit volatility state (like instantaneous variance $v$ in Heston or volatility-of-volatility $\eta$ etc.), differentiating the price w.rt those model inputs gives a Vega-like measure. Even if not, one can approximate Vega by bumping the implied vol (or equivalently, the network's output as if underlying's vol was slightly higher) and seeing the price change. The key is that because the network learned the pricing function, these greeks will typically be *arbitrage-consistent* and capture the true market sensitivities better than, say, a Black–Scholes delta on an implied vol input. A trader can thus **extract Delta from the network** and use it to delta-hedge the option in real time: i.e. hedge ratio = network's Delta. Since the network's pricing is more accurate, its Delta should, in principle, be more **optimal than the Black–Scholes Delta** that many practitioners use by plugging implied vol into the BS formula. (In static replication terms, the network's Delta is ensuring minimal local pricing error if the network is correctly calibrated).

- **Optimal Static Hedging (Vega hedging):** Beyond Delta (which hedges first-order underlying price risk), many traders hedge **Vega risk** by taking positions in other options. The Deep Volatility Arbitrageur can help identify the best such hedges. Because it provides a quick way to calculate the prices of many options under various scenarios, one can use it to run an optimization: for example, suppose you have a portfolio that is long an option (hence with some Delta and Vega exposure). You could consider adding a combination of other options (e.g. one shorter maturity and one longer maturity option) to hedge both Delta and Vega. The network can rapidly compute the **portfolio P&L under different market moves** (scenarios of underlying moves and volatility moves). Using this as a sandbox, you can solve for the weights of hedging instruments that minimize portfolio variance or another risk metric. In fact, the Medium article on VolGAN demonstrates an approach where they used a **LASSO regression** to determine optimal hedging weights for delta-vega neutrality [41]. In that case, they generated scenarios of underlying and IV surface moves (using a generative model akin to ours) and then picked hedging positions (in various strikes) such that the **portfolio's Delta and Vega exposures are neutralized** in a least-squares sense [42] [43]. We could replicate a similar approach: use the Neural SDE to simulate joint paths of the underlying and volatility (since it's a generative model for both), then at a given time step, set up equations for Delta and Vega exposure. Delta neutrality implies $\Delta_{\text{port}} + \sum_i \Delta_{h,i} \cdot \phi_i = 0$ (where $\phi_i$ are units of hedging instrument $i$), and Vega neutrality implies similarly the total vega is zero. Solving these gives hedge ratios. The network can supply $\Delta_{h,i}$ and $\text{Vega}_{h,i}$ for each candidate hedge instrument $i$ instantaneously, making this optimization fast. One might pick the simplest hedging instruments: the underlying itself for Delta, and one or two liquid options for Vega (e.g. an ATM option to hedge overall volatility level and maybe a risk-reversal or straddle to hedge skew curvature). By solving for $\phi_i$, we get the **optimal static hedge** that cancels out first-order exposures.

- **Dynamic Hedging via Reinforcement Learning:** The ultimate "deep hedging" view would use the network to simulate many future paths and then train another neural network (or use dynamic programming) to decide how to rebalance the hedge over time. Essentially, you treat the Neural SDE as the true market: it can simulate, say, daily moves in the underlying and volatility surface. Then you can employ an RL agent that at each time step chooses how much of underlying or other options to hold to minimize some risk measure (like variance of final P&L or CVaR). Buehler et al. did this with an LSTM network that observed past info and hedged accordingly, outperforming naive delta-hedging in their experiments. With our Arbitrageur, we have the crucial piece: a realistic, arbitrage-free market generator for options and underlying. We could thus implement an episodic training where each episode is a simulated price path (with jumps, volatility swings, etc. as captured by the Neural SDE). The agent (another neural net) learns to take actions (adjust hedge) to, say, **minimize the 95% VaR of the hedged portfolio**. Over many simulations, it will converge to a hedging strategy that might, for example, dynamically adjust delta based on volatility regime (something a Black–Scholes delta hedge wouldn't do, as it ignores volatility changes). This is advanced and beyond a quick implementation, but it leverages the **Neural SDE's ability to generate realistic joint dynamics** of the underlying and option prices. The result of such deep hedging could be an **optimal combination of Delta and Vega hedges over time** that outperforms traditional strategies under heavy-tailed or regime-changing conditions [43] [44].

In more immediate terms, a trained network allows one to compute **Greeks on the fly** and perform hedging *analytically*. For example, since the CCN corrects Heston's errors, one could use the *Heston model plus CCN* to derive hedge ratios. Heston already gives analytical formulas for Delta and Vega (or can be differentiated), and CCN's correction can be differentiated too. Thus the total Delta = Heston Delta + $\partial$(CCN correction)/$\partial S$. This merged Delta will hedge the true market much better. If CCN is small, it means Heston's delta was almost sufficient; if CCN is significant, it will adjust the delta accordingly.

As a validation, one can test the hedging performance of the network by **simulated hedging experiments**. Take historical data or simulated paths, and compare P&L of hedging the option using: (a) Black–Scholes delta, (b) Heston delta, (c) network delta, (d) network delta + vega hedging with another option. One would expect (c) or (d) to perform best (lowest variance of P&L, smallest tail losses) especially in scenarios where volatility moves are significant. Indeed, case studies have shown that machine learning-based hedging can **reduce hedging error** and better capture the joint dynamics of underlying and implied vol [45] [46]. For instance, the VolGAN model's hedging experiment found that a strategy using a learned volatility surface model achieved very precise replication of a straddle's payoff, with lower tracking error than traditional delta-gamma hedging [47] [45]. This underscores that by **incorporating a data-driven understanding of volatility**, the Deep Volatility Arbitrageur can suggest hedge ratios that *account for future volatility changes*, not just immediate delta risk.

In conclusion, the Deep Volatility Arbitrageur prototype provides a comprehensive toolkit: a fast and arbitrage-free pricing engine (via Neural SDE and RNGN concepts) and a means to extract hedging strategies (via differentiable networks and scenario simulation). It blends financial theory (risk-neutral pricing, no-arbitrage, stochastic volatility models) with cutting-edge deep learning, yielding a system that **prices options accurately and rapidly**, and **enables sophisticated hedging (Delta-Vega optimization)** that can adapt to complex market dynamics. This makes it highly appealing for real-world deployment in trading floors where speed and correctness are paramount. The approach exemplifies how deep learning can augment derivative modeling – achieving the holy grail of **fast, arbitrage-free, and robust** pricing and risk management [39] [20]. The developer-focused insights above should help implement the system in

PyTorch and integrate it into pricing libraries or trading systems, while the theoretical understanding ensures the model's outputs remain grounded in sound financial logic.

**Sources:** The concepts and results discussed are drawn from recent research in quantitative finance and machine learning, including risk-neutral generative modeling [2] [48], dual-network Heston calibration frameworks [9] [14], no-arbitrage machine learning methods [4] [5], and deep hedging strategies [41] [39], as cited throughout this report.

---


[1] [2] [3] [4] [5] [6] [7] [8] [17] [18] [27] [29] [30] [48] Risk-Neutral Generative Networks
https://arxiv.org/html/2405.17770v1

[9] [10] [11] [12] [13] [14] [15] [16] [19] [20] [21] [22] [23] [39] [40] [2510.24074] Deep Learning-Enhanced Calibration of the Heston Model: A Unified Framework
https://ar5iv.labs.arxiv.org/html/2510.24074

[24] [25] open.uct.ac.za
https://open.uct.ac.za/bitstreams/6dd8464f-d8d8-4dd9-b8cc-8295a39a83e8/download

[26] [31] Arbitrage-free neural-SDE market models | Mathematical Institute
https://www.maths.ox.ac.uk/node/39328

[28] [PDF] A new Input Convex Neural Network with application to options pricing
https://arxiv.org/pdf/2411.12854

[32] [33] [34] [35] [36] [37] Rough Bergomi: Fractional Volatility & Calibration
https://www.emergentmind.com/topics/rough-bergomi-model

[38] [1901.09647] Deep Learning Volatility - arXiv
https://arxiv.org/abs/1901.09647

[41] [42] [43] [45] [46] [47] Exploring VolGAN and its Application in Arbitrage-free Implied Volatility Surface Generation and Derivatives Hedging | by Petruskung | Dec, 2025 | Medium
https://medium.com/@petruskung/exploring-volgan-and-its-application-in-arbitrage-free-implied-volatility-surface-generation-and-c4e762dff700

[44] [PDF] Is the Difference between Deep Hedging and Delta Hedging ... - arXiv
https://arxiv.org/pdf/2407.14736