WHITE PAPER

# DEVELOPING A NEURAL EXOTIC PRICING LIBRARY WITH TENSORFLOW

GIUSEPPE BENEDETTI

Replacing slow exotics pricers with quick ones based on neural networks has been an active area of research and practical in-house implementations during the last few years. The promise of real-time pricing or blazing fast risk computations, even when dealing with very complex portfolios of exotics, is something that appeals to many institutions and vendors alike.

The high-level recipe to put that into practice is typically the following:

1. Generate a training set using traditional pricers, over a sufficiently large range of market scenarios.

2. Use the generated data to train a neural network.

3. Replace the traditional (slow) pricer with the trained network in the financial computation at hand (e.g. XVA, VaR, or real-time pricing).

While it may look relatively straightforward, there are quite a few different ways this can be achieved.

Much has already been written on the subject (see for example [1] and [2] and references therein). The objective of this article is to summarize some of the choices that need to be made and factors that need to be considered to produce satisfactory results, along with specific observations based on our experience in implementing our own neural pricing and risk system at FIS.

## Ground truth vs sample based

Each learning task of any kind starts with a training set, i.e. a set of examples that we want to learn from. The distinguishing factor in our specific context is that the training set is not exogenous but needs to be generated by the user, which is both a challenge and an opportunity.

When generating the training set, we first need to decide what to generate exactly. The first choice is to train on ground truth prices computed by an existing (slow) pricer on the different market scenarios. This is the most direct approach, and it works very well. However, it is likely to be very computationally demanding. If a pricer takes a couple of seconds to run, it could take several months to produce a few million examples (unless cloud computing is used).

A second approach is to generate Monte Carlo samples where each sample consists of a single Monte Carlo path simulated from each market scenario. One can also use a few Monte Carlo paths averaged out, to keep the training set a bit smaller and less noisy. This is based on an old idea that dates back to the classical American Monte Carlo (AMC) algorithm, that has been brought to new life in recent years and extended in various ways, particularly in the XVA context (see [1]). The advantage of this method is that generating samples is orders of magnitudes faster than generating prices, especially when they can be run in parallel. It's a more accurate method when run with a finite time budget.

Samples are therefore equivalent to very noisy Monte Carlo prices. One can switch from ground truth to sample based learning by reducing the number of paths required to compute a single valuation, assuming that the original pricing function is based on Monte Carlo simulation. That is usually very beneficial (as remarked very early on by [2]), although there is a strong limit in doing so when the single valuation includes fixed running costs that are independent from the number of paths (such as calibration and exposure estimations for callables). We will mention these aspects in the following sections.

## Input types

Before continuing, it is useful to settle some terminology on the different types of parameters (or variables) that can be used as inputs for neural pricers:

- Market parameters are raw data of traded instruments that are directly available from the market (like swap rates, implied volatility surfaces or CDS rates). Often one cannot directly plug these variables into a pricing model; a calibration step needs to be performed first.
- Model parameters are the results of the calibration step, that is values that can directly be used to generate simulation paths. Examples are zero rates, local volatilities, hazard rates and factor correlations.
- State variables are the quantities whose evolution is described by the pricing model. Examples are spot prices, short rates or stochastic volatilities. While model parameters stay constant over a path, state variables evolve based on the model parameters and a set of random draws.
- Payoff parameters are values that are specific to a certain deal type. Examples are strikes, knock-in barriers, coupon rates and trade maturity.

## Learning model calibration (or not?)

Ground truth learning generally also learns the model calibration along with the pricing, producing functions of market parameters. Alternatively, sample based learning (similarly to AMC) traditionally only takes state variables as inputs, which is enough in many risk applications like CVA. That doesn't need to be the case though: in addition to state variables, sample based learning can be extended to include model parameters and payoff parameters as inputs.

In the context of sample based learning, it is generally undesirable to use market parameters, which need to be processed through calibration before the actual path generation can start. This is because calibration is usually an optimization procedure that can add a significant overhead to the sample generation. This will likely prevent multiple samples from being run in parallel when starting from different market data, greatly reducing the benefits of working with samples of few simulation paths each. On the other hand, when using ground truth learning, a single valuation is already relatively slow. Including the calibration step makes more sense as it only adds a comparatively small overhead.

It is a matter of choice, which depends on the objectives and constraints of the final user. One needs to carefully evaluate the trade-off between training time and evaluation time of the trained model plus the calibration, based on the application at hand. We tend to prefer the use of sample based learning on state variables and model and payoff parameters when needed. This allows us to take full advantage of a very fast and parallel training set generation, and a quick training of any deal type almost on the fly.

The main limitation is that when a calibration is required, it will need to be performed externally from the trained pricer, potentially making it slower to use. However, optimizations are possible and relatively straightforward, as calibrations are generally reusable over many products in the same portfolio. This makes it redundant to inflate the training time to learn the same calibration for multiple deal types.

## Which (and how many) inputs?

We want to learn a pricing function, but a function of what? Sample based learning was born in the context of AMC. It traditionally learns a function of the model state variables, everything else being fixed such as the maturity/time horizon, the model and payoff parameters. This allows it to be executed relatively quickly as a preliminary step of a pricing task that involves some form of callability.

XVA regressors for future exposures can also be computed using the same logic as functions of state variables only. The reason is that XVA can generally be seen as the pricing of a complex option, with model and payoff parameters fixed at time zero.

On the other hand, the same logic of using only state variables as inputs of the learnt pricing functionals is no longer suitable for other types of risk computations. This includes Value-at-Risk (VaR), which requires multiple revaluations at the risk horizon using different (shocked) market/model parameters, or front office applications such as real-time pricing where many consecutive valuations with different market data (and potentially new payoffs) need to be performed in a short amount of time.

Therefore, it is sometimes necessary to train effective approximators that are functions not only of state variables, but also of model/market and payoff parameters. We will call these full pricers for simplicity. Given their additional complexity, these can no longer make use of polynomial regressions as for standard AMC. Instead, they need to use more sophisticated functionals such as neural networks, which can handle large input dimensions substantially better than traditional methods due to their approximation and generalization capabilities. Adding input dimensions will make the learning slower, although not necessarily by an excessive amount, thanks to Monte Carlo sampling.

It seems more common to train full pricers in a ground truth learning framework using a relatively high number of paths and including calibration. However, it appears to be less well known that these can also be learnt using sample based learning, with

substantial training performance improvements. The idea here is to simply generate each path, or minibatch of paths, starting from different model and/or payoff parameter values, by sampling them over some suitable domains. Explicit averages are only computed over the individual minibatches.

## Generating scenarios

We mentioned the need to generate, before each MC path evolution, scenarios for initial state variables and for market/model and payoff parameters, if training a full pricer. The first step is to choose some suitable ranges for these values. It's important to keep in mind that the trained pricer will only be valid when called with input values inside those ranges. Neural networks act very well as interpolators but can be unpredictable as extrapolators, although they're generally substantially more stable than polynomials. For example, one might decide to train a vanilla option pricer for strikes between 60% and 140%, maturities between 0 and 5 years, rates between -1% and 4%, and vols between 10% and 40%.

Once the domains are chosen, the next question is how to sample over those domains. A few options are available:

1. Sample all values uniformly over their domains, and independently from each other.

2. Sample all values uniformly (and independently) over some specific points of their domains. For example, Chebyshev points, that are known to have good interpolation properties.

3. For term structured or surface data, choose a common parametrization (like Nelson-Siegel or SSVI) and sample those shape parameters over some reasonable ranges. This will likely produce more realistic data than a naïve independent sampling of all tenors/nodes. Then it is up to the user if they want to use those shape parameters as network inputs, or only use them to generate data at fixed tenors/nodes, and then take that generated data as network inputs.

4. Use historical data (although there might not be enough data available) or more sophisticated generative market models calibrated on historical data.

For this particular application, the important thing to remember is that using "realistic" data samples does not have the same importance that it has in other contexts; for example, when running scenarios for risk calculations or performance projections. The main purpose here is to expose the training routine to a sufficiently rich range of scenarios. That way, when the pricer is used on real market data, we can be sure the network has already seen a similar data configuration, or can successfully interpolate between sufficiently close ones that it has already seen. A minimal amount of realism is desirable nonetheless, especially in term structure or surface data. For example, if we sample a 1Y rate at 4% on one given scenario, we will probably not want the 2Y rate to be at -2% on that same scenario. This would risk creating domain configurations where the objective pricing function is very steep in time and possibly harder to learn, with respect to a more realistic sampling where term structures behave more smoothly.

It's generally optimal to use the naïve approach 1. for scalar market or payoff data. On the other hand, 2. with Chebyshev points doesn't seem to add much value in our tests. For term structured and surface data, some form of 3. is always useful.

## Which network structure?

Traditional feed-forward neural networks are generally used as function approximators, as they are simple, versatile and well understood. Although, different architectures might also be explored, such as ResNets or even Transformers, which have proven incredibly successful in many different areas of AI in recent years. It has also been suggested in [3] to use special jump units as a means for replicating the jump behavior that is typically observed in certain regions of the pricing functions, for example close to maturity or coupon dates. It is even possible to radically depart from neural networks completely, as recently explored in [5]. If one wishes to use the differential training technique (described in the next section), all the internal activation functions need to be twice continuously differentiable.

The size and depth of the network will greatly depend on the complexity of the problem. When learning a simple Black-Scholes call option pricer as a function of a single spot input, a couple layers with a few dozen nodes might be enough for great accuracy. On the other hand, if we want to fit a complex callable PRD product with multiple coupons under a hybrid FX/IR model, as a function of, say, 100 input variables (knock in/out barriers, strike, maturity, rates, forwards, local volatilities, mean reversions), then we will need a substantially wider and deeper network, possibly with more than 10 layers and 100 nodes. Learning more complex functionals will also likely require to run more training epochs to achieve the desired accuracy.

The same standard mean-square-error (MSE) loss function can be used for both ground truth and sample based learning. So, a single learning framework can be built for both approaches, even though we are doing slightly different things in the two cases: a straightforward prediction of input labels in the former case, and an L2 projection (i.e. a conditional expectation) in the latter, theoretically leading to the same result. The extra advantage of sample based learning is that the network is never trying to exactly fit the input labels, which makes it empirically almost impossible to experience overfitting for normal configurations of sample sizes and parameter counts. That is a very valuable trait, considering that overfitting is one of the most recurring problems in deep learning.

It is generally good practice, at least when learning full pricers with many input variables, to also generate a testing set by running a full "classical" Monte Carlo pricing on an independent random set of input data, and use it to compare those "true" data points with the network-generated ones over each training epoch. This allows to track the network accuracy and potentially early stop the training once it achieves the desired precision.

## Differential machine learning

It has recently been proposed (by [1]) to perform the training using not only the standard cash-flow samples, but also including the corresponding pathwise differentials with respect to the various input variables into an additional term in the loss function. The idea is to boost the standard "level based" regressor by providing additional shape information and allowing it to converge faster and in a more stable way.

The downside of this approach is the additional time required to compute the pathwise differentials. Although with a good AAD system, that can be limited to a few times the base running time. Memory can also be a limiting factor, both at the training set generation phase (due to the need to keep a tape of all the operations on all the paths at the same time, when generated in parallel), and at the actual network training step (linked to the fact of having many more labels to store into GPU memory).

The effects of using this technique can be astonishing, with the potential of increasing the network accuracy by several orders of magnitude. We tend to observe a less pronounced performance boost when the number of training paths increases, the payoff gets more complex and as we start adding multiple input variables for learning a full pricer. In those cases, it is most probably optimal to avoid differentiating with respect to the full set of input variables, by only picking a smaller subset instead.

The application of differential learning should be evaluated and customized to each particular use case, in particular for pricing vs risk applications, and considering the time/memory budget for the current problem. Regardless, it remains a useful technique that should always be part of a neural pricing solution.

## Callable trades and neural AMC

An important limitation to fully exploiting the benefits of sample based learning is encountered when dealing with callable products. When pricing callables with Monte Carlo, one typically runs a set of American Monte Carlo (AMC) regressions before the main simulation, to estimate the optimal exercise region to be applied on every call date. In the context of learning, this would mean that every training sample that uses a different set of model and payoff parameters needs to run its own AMC regressions before it can be generated.

This approach has a huge impact on the performance of the learning set generation for two reasons: because the AMC step represents a very significant overhead (especially when performed only to allow the simulation of a handful of paths), and because such a routine will likely prevent different samples to be generated in parallel.

This limitation pushes towards using ground truth learning in the case of callables, using a large number of paths per sample to reduce the relative overhead of the repeated AMC steps on the overall calculation.

There is an alternative however, which is novel to the best of our knowledge. This consists of running only one single "big" preliminary AMC step, and then using the same trained AMC regressors in the generation of all training samples with all input parameters. To achieve that, one needs to:

- Run the forward AMC simulation with different sets of model/payoff parameters on each path, similarly to the main sample generation (also using the same parameter domains).
- When running the backward AMC pass, train AMC regressors on call dates in the form of neural networks. These can have same input dimension as the main simulation (minus the time dimension, as that is fixed for AMC exposures), although that is not strictly necessary.

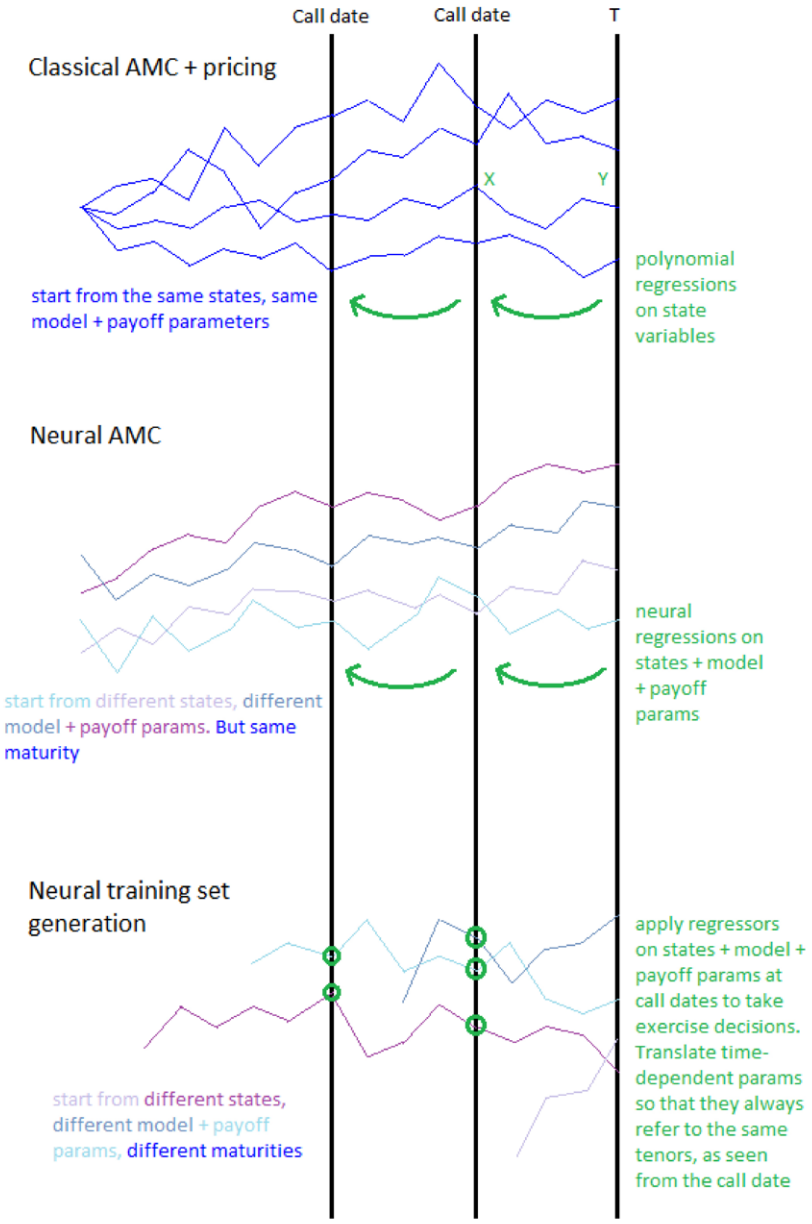
One apparent difficulty lies in that if we want to include the maturity as an input of the pricer, we need to train on samples with different maturities. However, by definition the AMC routine can only be run with a fixed maturity and exposure dates. The solution is to run the AMC on all paths with a single maturity, corresponding to the upper bound of the input maturity range. That will generate exercise regressors at fixed distances from maturity. These can later be used during sample generation to take exercise decisions at those times to maturity by any trade, independently of its original maturity.

The underlying hypothesis is that the exercise frequency doesn't change across trades in the training set. Otherwise, it would require running separate AMC routines for each call frequency. Another crucial implementation point is that time-dependent inputs such as zero rates or local vols need to be recentered to each specific call date before being fed to the exercise regressors. This ensures the input data for the regressor networks always keep the same meaning when used by samples with different initial maturities. See the figure below for a graphical visualization.

Since the accuracy of the AMC step only plays a second-order role in the accuracy of the overall pricing, it can be run using smaller network architectures than the one used for the main pricer, and with a limited number of paths. It seems reasonable to use whatever capacity is allowed by a single GPU batch.

Using neural AMC allows the generation of training samples for callable trades in a similar way as for non callables with all the associated performance benefits, by only adding a single extra global exercise estimation step. An additional benefit with respect to the traditional AMC is that here one does not need to manually specify a set of trade-specific regression variables, and that the exercise estimation with neural networks is substantially more accurate than with polynomials.

**Schematics of the different Monte Carlo modes.**



## Tensorflow implementation

Tensorflow is a well-known generic framework for parallel and distributed computing on CPU and GPU. It is specifically optimized for tensor and matrix operations, with native support for automatic differentiation. Those are the ideal characteristics for a deep learning library, which is the main reason why it has been built and it is still widely used.

What is slightly less known is that the same open-source technology is also suitable for a type of calculation that is ubiquitous in the financial industry, i.e. the Monte Carlo simulation of financial models whose dynamics can be expressed as matrix operations, and the associated computation of sensitivities of results with respect to the input variables. This implies that running highly efficient parallel simulations and vectorized computations on the latest CPUs and GPUs (or even TPUs) with AAD is today infinitely easier than it was years ago. It also no longer requires substantial investments and customized implementations for each hardware.

For these reasons, our neural pricing and risk solution uses Tensorflow for both training the network and generating the training data samples through Monte Carlo. We share below an extract of the Monte Carlo implementation which is at the heart of the system. This is to highlight that same small piece of code has the flexibility to handle simulation jobs in all of the following cases:

- With either the same or with different model and payoff parameters on each path (even different maturities), i.e. for "classical" Monte Carlo pricing or for generating training sets for sample based learning. In the former case, paths get averaged out to compute the final price, while in the latter they are returned as is to be fed to the network calibrator.
- With or without AAD. This is simply a matter of specifying, in the initialization methods, which input variables need to be tracked for differentiation. The same code produces either standard sensitivities in case of classical Monte Carlo pricing, or pathwise differentials in case of differential learning.
- With any model, whose parameters and state evolution need to be specified inside the generator object. New models (also written in Tensorflow) can be easily added to the library.
- With any payoff, whose evaluation rules and cash-flow dates are specified in dedicated classes. Callable payoff samples are also simulated by the same code, although they require a preliminary neural AMC step.
- With any hardware. Vectorized CPU or GPU run on the same code. Moreover, one can either run it in compiled or eager mode, based on whether it needs to be executed once or multiple times with higher efficiency.

### Snippet of the Tensorflow Monte Carlo code.

```
@tf.function
def montecarlo(systeminfo, mdgenerator, payoff, paths, mean = False):
    """Core Monte Carlo simulation function.

    This function performs a Monte Carlo simulation with AAD (when active)
    starting from a generator (determining the evolution of state variables) and a (set of) payoff description(s)
    The @tf.function decorator means the function will be compiled for greater efficiency. Comment out for clearer debugging."""

    with tf.GradientTape() as tape:
        ts = payoff.get_time_steps()
        market_factors, market_factors_values = mdgenerator.initialise_mc()
        payoff_sim_vars = payoff.initialise_simulation_variables(market_factors, market_factors_values, systeminfo, paths)
        for i in tf.range(ts.shape[1]-1):
            t = ts[:, i:i+1]
            dt = tf.reshape(ts[:, i+1:i+2] - t, [-1, 1])
            # Evolve the market data.
            market_factors_values = mdgenerator.evolve_states(market_factors_values, i, t, dt)
            # Evaluate payoff.
            payoff_sim_vars = payoff.evaluate_variables(systeminfo, market_factors, market_factors_values, payoff_sim_vars, i+1)
            # Evaluate the callability (without AAD, as regressors are only used as a proxy).
            with tape.stop_recording():
                payoff_sim_vars = apply_call_condition(systeminfo, payoff, market_factors, market_factors_values, payoff_sim_vars, i+1)

        # Compute average over small batch (and/or antithetic vars).
        Y = tf.reduce_mean(payoff_sim_vars[Payoff.PriceVarName], axis=1, keepdims=True)

        # When pricing in the classical way, we want to compute the average of all the MC paths.
        # When generating data for learning, we want to directly learn from the simulated payoffs.
        if mean:
            Y = tf.reduce_mean(Y, keepdims=True)

    # Compute differentials
    grad_vars = mdgenerator.get_training_variables()
    grad_vars.extend(payoff.get_training_variables())
    grad = tape.gradient(Y, grad_vars)

    return Y, grad
```

# CONCLUSIONS

We have outlined the main features and design choices of the neural exotic pricing library that we are currently developing at FIS. We see these functionalities as important building blocks for our future pricing and risk system. We hope that this white paper will help other industry participants by providing additional clarity and some new insights on such a rich subject.

**FOR FURTHER INFORMATION CONTACT US ON EMAIL GETINFO@FISGLOBAL.COM.**

**CONTACT US**

## Bibliography


1. Huge, B. and Savine, A. (2020): Differential Machine Learning. Arxiv

2. Ferguson, R. and Green A.D. (2018): Deeply Learning Derivatives. SSRN

3. Bergeron, M. and Sergienko, I. (2020): Deep Learning to Jump. Medium

4. Belletti, F. et al. (2020): Sensitivity Analysis in the Dupire Local Volatility Model with Tensorflow. Arxiv

5. Antonov, A. and Piterbarg, V. (2021): Alternatives to Deep Neural Networks in Finance. SSRN


## About FIS

FIS is a leading provider of technology solutions for financial institutions and businesses of all sizes and across any industry globally. We enable the movement of commerce by unlocking the financial technology that powers the world's economy. Our employees are dedicated to advancing the way the world pays, banks and invests through our trusted innovation, system performance and flexible architecture. We help our clients use technology in innovative ways to solve business-critical challenges and deliver superior experiences for their customers. Headquartered in Jacksonville, Florida, FIS is a member of the Fortune 500® and the Standard & Poor's 500® Index.


www.fisglobal.com

getinfo@fisglobal.com

twitter.com/fisglobal

linkedin.com/company/fis




ADVANCING THE WAY THE WORLD PAYS, BANKS AND INVESTS™