Nested Monte Carlo for asian options and training

Saad Souilmi - Germain Vivier–Ardisson

March 2024

Summary

Problem statement

$$dS_t = rS_t\,dt + \sigma S_t\,dW_t$$

The Euler-Maruyama scheme on a uniform grid $t_k = k\delta$, $k \in \{0, ..., n\}$:

$$\forall k \in \{0, ..., n-1\}\ \ S_{t_{k+1}} = S_{t_k}\left(1 + r\delta + \sigma\sqrt{\delta}Z_{k+1}\right)$$

Where Z are i.i.d standard normal variables. Finally we compute $F(t, S_t, I_t) = e^{-r(T-t)}\mathbb{E}\left[(S_T - I_T)^+ \middle| S_t, I_t\right]$ via the proxy :

$$F(t, S_t, I_t) \approx \frac{e^{-r(T-t)}}{n_{\text{paths}}} \sum_{i=1}^{n_{\text{paths}}} \left(S_T^i - \frac{t}{T} I_t - \frac{\delta}{T}\sum_{k=1}^{n} S_{t+t_k}^i\right)^+$$

Parallelizing over trajectories

- We fix initial conditions (t, S_t, I_t)
- Every thread generates one sample path for following the diffusion
- To aggregate the results of all the threads we can either implement the array reduction inside the kernel or use the numba.cuda.reduce api.

Parallelizing over initial conditions

- The problem with the last approach is that we need to loop in cpu in order to generate multiple prices for different initial conditions.
- We can either assign a triplet (t, S_t, I_t) to each thread that will sample n_{paths}.
- Or we can assign a triplet (t, S_t, I_t) to each block and each thread will sample $\frac{n_{\text{paths}}}{n_{\text{threads per block}}}$.

Learning problem

We aim to learn the map :

$$F(t, S_t, I_t, T, r, \sigma) = e^{-r(T-t)}\mathbb{E}\left[(S_T - I_T)^+ \middle| S_t, I_t\right]$$

For convenience, we will write :

$$x = (t, S_t, I_t, T, r, \sigma) \text{ and } X = e^{-r(T-t)}(S_T - I_T)^+$$

We can thus write F as $F(x) = \mathbb{E}[X|x]$.

Learning problem

We want to find :

$$\theta^* \in \text{argmin}_{\theta \in \Theta} L(\theta) = \text{argmin}_{\theta \in \Theta} \mathbb{E}_{x \sim \mathcal{D}} \left[(F(x) - T_\theta(x))^2 \right]$$

Where $\mathcal{D}$ is some prior distribution over the parameter space, and T_θ is a neural network.

$$L(\theta) = \underbrace{E_{x \sim \mathcal{D}} \left[\mathbb{E} \left[(X - T_\theta(x))^2 \, | x \right] \right]}_{= \tilde{L}(\theta)} - \mathbb{E}_{x \sim \mathcal{D}} \left[\mathbb{V}(X | x) \right]$$

$$\text{Thus} \qquad \text{argmin}_{\theta \in \Theta} L(\theta) = \text{argmin}_{\theta \in \Theta} \tilde{L}(\theta)$$

We can thus train our network directly on the sampled payoffs.

Data generation

Parameter	Interval
S_t	$[30, 70]$
$T - t$	$[0.2, 1]$
t	$[0, 0.8]$
σ	$[0.1, 0.5]$
r	$[0, 0.1]$

Table – Grid of parameter intervals.

- When drawing a set of parameters we draw the five parameters below witht he corresponding interval, and if t is small enough, we set I_t to be equal to S_t otherwise, we sample I_t in $[0.5S_t, 2S_t]$.

Data generation

- For training data, we drew 10^6 sample parameters from the grid defined before using the scrambeled Halton sequence. For each sample parameter we generated 10^3 paths.
- For validation data, we drew 10^4 sample parameters form the grid defined before using the scrambeled Halton sequence. For each sample parameter we generated 10^6 paths and computed the monte carlo estimation of F.

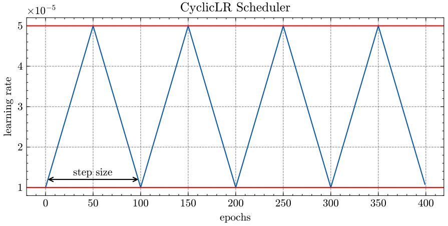
Architecture

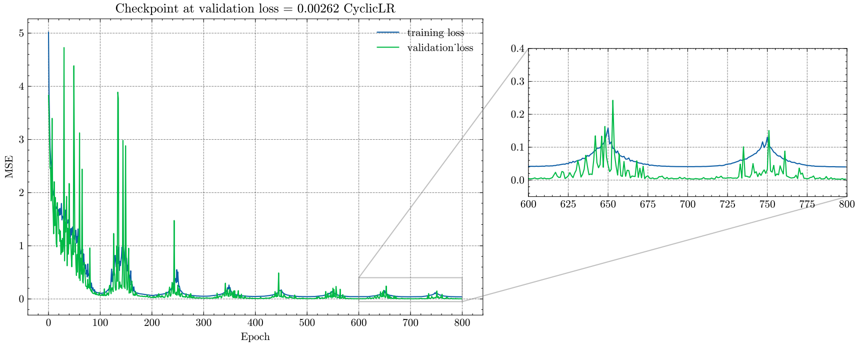
- The neural network we trained is a fully connected MLP, with 4 hidden layers, each containing 400 neurons.
- We chose for our activation the SiLU function.
- We introduced LayerNorm between the hidden layers.
- Since we predict a positive outcome (price), we run our final output through a ReLU.

Training

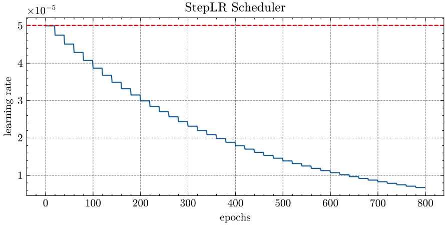
- For training we first attempted to train our network directly over payoffs.
- Training was very slow, even with big batch sizes ($\approx 3 * 10^4$), as in this case our training dataset contains 10^9 samples. And also very unstable.
- We opted to train our network on MC estimations of the price. Even though the estimations are noisy since they only utilize $1k$ paths, our network manages to converge.

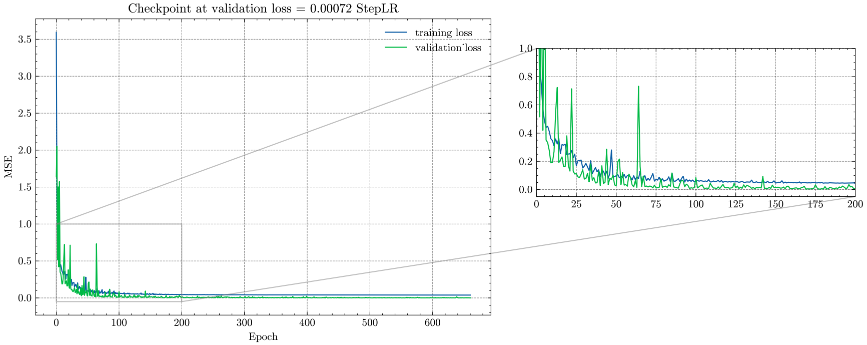
CyclicLR


CyclicLR Scheduler
×10^-5
learning rate
step size
epochs


Checkpoint at validation loss = 0.00262 CyclicLR
training loss
validation loss
MSE
Epoch

StepLR


StepLR Scheduler
$\times 10^{-5}$
learning rate
epochs


Checkpoint at validation loss = 0.00072 StepLR
training loss
validation loss
MSE
Epoch

LDLt decomposition

- In the LDLt implementation each block works on a portion of the matrix a.
- Each block is divided into groups of n threads, and each group works on a column of the portion of the matrix associated to the block.
- int tidx = threadIdx.x%n determines the row index the thread is working on.
- int Qt = (threadIdx.x-tidx)/n determines the index of the column that the thread is working on, within the portion of a associated with the block.
- int gb_index_x = Qt + blockIdx.x*(blockDim.x/n) is the global memory index of the column the thread is working on this time in the matrix a.