DEEP PRICING

FINAL PROJECT

- Group 1 -

Matthieu JULIEN

Léna TUVACHE

Daphnée CORREIA

Théo MIDAVAINE

Thomas JOUVE

ESILV A5 IF 2023-2024

Contents

1 Pricing and Hedging by Monte Carlo

1.1 Finite Differences Approach

1. Implement a standard Euler-Maruyama approximation to the Heston model. Define a function called `GeneratePathsHestonEuler()` whose arguments are: seed, model parameters, number of paths, number of steps.

```
def GeneratePathsHestonEuler(seed, avg_variance, k, gamma, corr_S_V, r, S0, V0, T,
 ↪  nb_paths, nb_steps):

  np.random.seed(seed)
  dt= T/nb_steps

  s_t = np.zeros((nb_steps + 1, nb_paths))
  v_t = np.zeros((nb_steps + 1, nb_paths))
  s_t[0, :] = S0
  v_t[0, :] = V0

  W_V = np.random.normal(size=(nb_steps, nb_paths))
  W_S = corr_S_V * W_V + np.sqrt(1 - corr_S_V ** 2) *
   ↪  np.random.normal(size=(nb_steps, nb_paths)) #Création d'un BM corrélé à W_V en
   ↪  utilisant la formule du TD

  for j in range(nb_paths):
    for i in range(1,nb_steps+1):

       ↪  s_t[i,j]=s_t[i-1,j]+r*s_t[i-1,j]*dt+mt.sqrt(v_t[i-1,j]*dt)*s_t[i-1,j]*W_S[i-1,j]
      v_t[i,j]=abs(v_t[i-1,j]+k* (avg_variance-v_t[i-1,j])*dt
      +gamma*mt.sqrt(v_t[i-1,j])*W_V[i-1,j])

  return s_t
```

The Heston model describes the dynamics of an asset price and its variance with two stochastic differential equations, while the Euler-Maruyama method is a numerical technique, used to approximate solutions to SDEs. In this function we use loops to iterate over each path and time step, updating the asset price and variance at each step. The function returns the matrix of simulated asset prices.

Below is a graph of the simulated prices using our function, using 100 steps and 10000 paths:

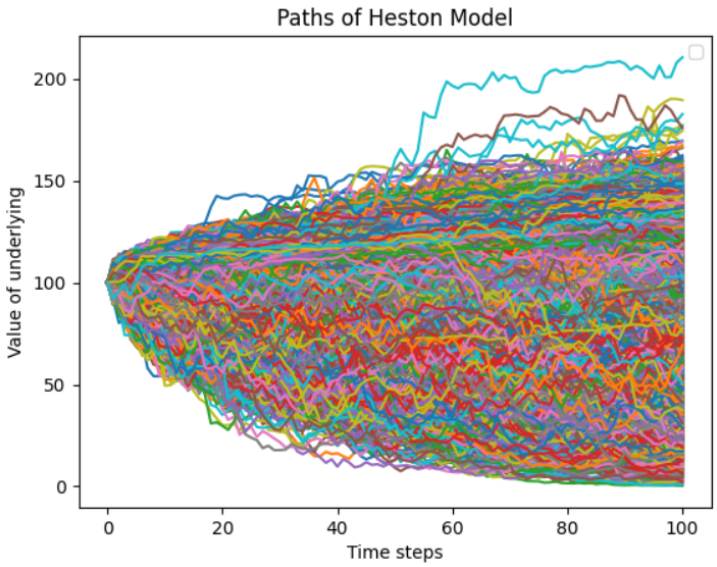

Figure 1: Paths of Heston Model.

2. Implement a function `Payoff` that returns the payoff of the down-and-out barrier call option defined above. This function takes the option parameters and the asset prices in argument and returns the payoff of the option.

```
def Payoff(strike, barrier, r, maturity, asset_prices):
  broken_barrier=False
  for price in asset_prices:
    if price<=barrier:
      broken_barrier=True
  if broken_barrier==False:
    payoff=mt.exp(-r*maturity)*max(0,asset_prices[len(asset_prices)-1]-strike)
  else:
    payoff=0

  return payoff
```

In this function, we check all the prices of the underlying taken in the lifetime of the option to ensure that it didn't break the barrier. If it does, the payoff is automatically set at 0. If not, the payoff is the maximum between 0 and ST (the price taken by the underlying at maturity) - K (the strike of the option) since it is a call option. This result is then discounted.

This can be summarized using the following equation (from the guidelines and Cox et al. [1], He et al. [2], Huge and Savine [3], and Longstaff and Schwartz [4]):

$$g = e^{-rT} \max(0, S_T - K) 1_{\min\{0 \le t \le T, S_t > B\}}$$

3. Implement a function `MC Pricing()` which takes the seed, the number of paths, the number of steps, the model parameters, and the option parameters in argument and returns the price of the option computed by Monte Carlo. This function must call `GeneratePathsHestonEuler()` and `Payoff()` to ensure code readability.

```
def MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0, V0,
 ↪  strike, barrier, maturity):

  matrix_Heston_path=GeneratePathsHestonEuler(seed, avg_variance, k, gamma, corr_S_V,
   ↪  r, S0, V0, maturity, nb_paths, nb_steps)
  matrix_Heston_path=matrix_Heston_path.T
  payoffs=[]
  option_price=[]
  for i in range(nb_paths):
    payoffs.append(Payoff(strike, barrier, r, maturity, matrix_Heston_path[i]))
  option_price= np.mean(payoffs)

  return option_price, payoffs
```

For the Monte Carlo Pricing, we generate a large number of path for the underlying price with `GeneratePathsHestonEuler()`. We then compute the payoff of each of these paths using our previous `Payoff()` function. The price of an option being the expectation of its payoff, we take the mean of all the payoffs computed to have the expectation, which is then the option price. The function also returns the payoffs vector to use it later on the code to compute the standard error.

$$E\left(e^{-rT} \max(0, S_T - K) 1_{\{\min\{0 \le t \le T, S_t > B\}\}}\right) = \text{option price} \tag{1}$$

4. Implement three functions `DeltaFD()`, `GammaFD()`, `RhoFD()` which estimate the delta, the gamma, and the rho of the option by using first-order finite differences (i.e., "bumping").

In this section, we want to compute three greeks of our barrier option, the δ, the γ and the ρ using first-order finite differences. Since δ is the first-order derivative of the call option price with respect to a change in the underlying price, we can compute its first-order finite difference as follows:

$$\delta \approx \frac{MC(S_0 + \text{bump}) - MC(S_0)}{\text{bump}} \tag{2}$$

Where $MC(S_0 + \text{bump})$ is the option price estimated using Monte Carlo simulation with the underlying asset price increased by a small amount, and $MC(S_0)$ is the original estimated option price.

γ is the second-order derivative of the call price with respect to the underlying price. Thus we must slightly modify the formula used to compute δ, even though we will still make the underlying vary. Then we will compute a central difference approximation :

$$\gamma \approx \frac{MC(S_0 + \text{bump}) - 2 \cdot MC(S_0) + MC(S_0 - \text{bump})}{\text{bump}^2} \tag{3}$$

ρ is the first-order derivative of the call price regarding a change in the risk-free rate. Therefore we can use a similar formula to the one we used to compute δ, but varying the risk-free rate this time.

$$\rho \approx \frac{MC(r + \text{bump}) - MC(r)}{\text{bump}} \tag{4}$$

The "bump" in each case represents a small change in the respective parameter (asset price for δ and γ, interest rate for ρ). In python, we get:

```
def DeltaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0, V0,
↪  strike, barrier, maturity, S_bump, mc_plus_bump, mc):
  delta=(mc_plus_bump -mc)/S_bump
  return delta
```

```
def GammaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0, V0,
↪  strike, barrier, maturity, S_bump, mc_plus_bump, mc, mc_moins_bump):
  gamma=(mc_plus_bump-2*mc+mc_moins_bump)/(S_bump**2)
  return gamma
```

```
def RhoFD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0, V0,
 ↪  strike, barrier, maturity, r_bump, mc_plus_bump_r, mc):
  rho=(mc_plus_bump_r-mc)/r_bump
  return rho
```

5. For the three sensitivities set the number of paths to 10000 and the number of steps to 100. Plot how the variance of the estimator changes with the bump size, and comment on the reasons for this.

In this part, we use the three previous functions to compute the greeks. We take various bumps, comprised between 1 to 10. For each bump value, we will compute the greeks multiple times to obtain an average of the greek computed. For example, for bump 1, we will generate 5 times the greeks (the number is arbitrary), and take the mean of the values as the final greek values for this bump.

```
delta_final=[]
gamma_final=[]
rho_final=[]
vega_final=[]

strike=100
barrier=90
seed=123
avg_variance=0.5
k=0.1
gamma=0.1
corr_S_V=-0.9
r=0.02
S0=100
V0=0.1
nb_paths=10000
nb_steps=100
T=1
bumps=[1,2,3,4,5,6,7,8,9,10]

for i in range(len(bumps)):
  S_bump=bumps[i]
  r_bump=bumps[i]
  v_bump=bumps[i]
  delta_list=[]
  gamma_list=[]
```

```
  rho_list=[]
  vega_list=[]

  for j in range(5):
    mc_plus_bump=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0 + S_bump, V0, strike, barrier, T)[0]
    mc=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0,
     ↪  V0, strike, barrier, T)[0]
    mc_moins_bump=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0 - S_bump, V0, strike, barrier, T)[0]
    mc_plus_bump_r=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r + r_bump, S0, V0, strike, barrier, T)[0]
    mc_plus_bump_v=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0+v_bump, strike, barrier, T)[0]

    delta_list.append(DeltaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0, strike, barrier, T, S_bump, mc_plus_bump,mc))
    gamma_list.append(GammaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0, strike, barrier, T, S_bump, mc_plus_bump, mc,
     ↪  mc_moins_bump))
    rho_list.append(RhoFD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V,
     ↪  r, S0, V0, strike, barrier, T, r_bump, mc_plus_bump_r, mc))
    vega_list.append(VegaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0, strike, barrier, T, v_bump, mc_plus_bump_v, mc))

  delta_final.append(delta_list)
  gamma_final.append(gamma_list)
  rho_final.append(rho_list)
  vega_final.append(vega_list)
```

We then compute the mean and the variance for each bump.

```
var_delta=[]
mean_delta=[]
var_gamma=[]
mean_gamma=[]
var_rho=[]
mean_rho=[]
var_vega=[]
mean_vega=[]

for i in range(len(delta_final)):
  var_delta.append(np.std(delta_final[i])**2)
```

```
mean_delta.append(np.mean(delta_final[i]))
var_gamma.append(np.std(gamma_final[i])**2)
mean_gamma.append(np.mean(gamma_final[i]))
var_rho.append(np.std(rho_final[i])**2)
mean_rho.append(np.mean(rho_final[i]))
var_vega.append(np.std(vega_final[i])**2)
mean_vega.append(np.mean(vega_final[i]))
```

Below are the plots of the standard deviations and means of δ, γ, ρ, ν against the bump sizes.

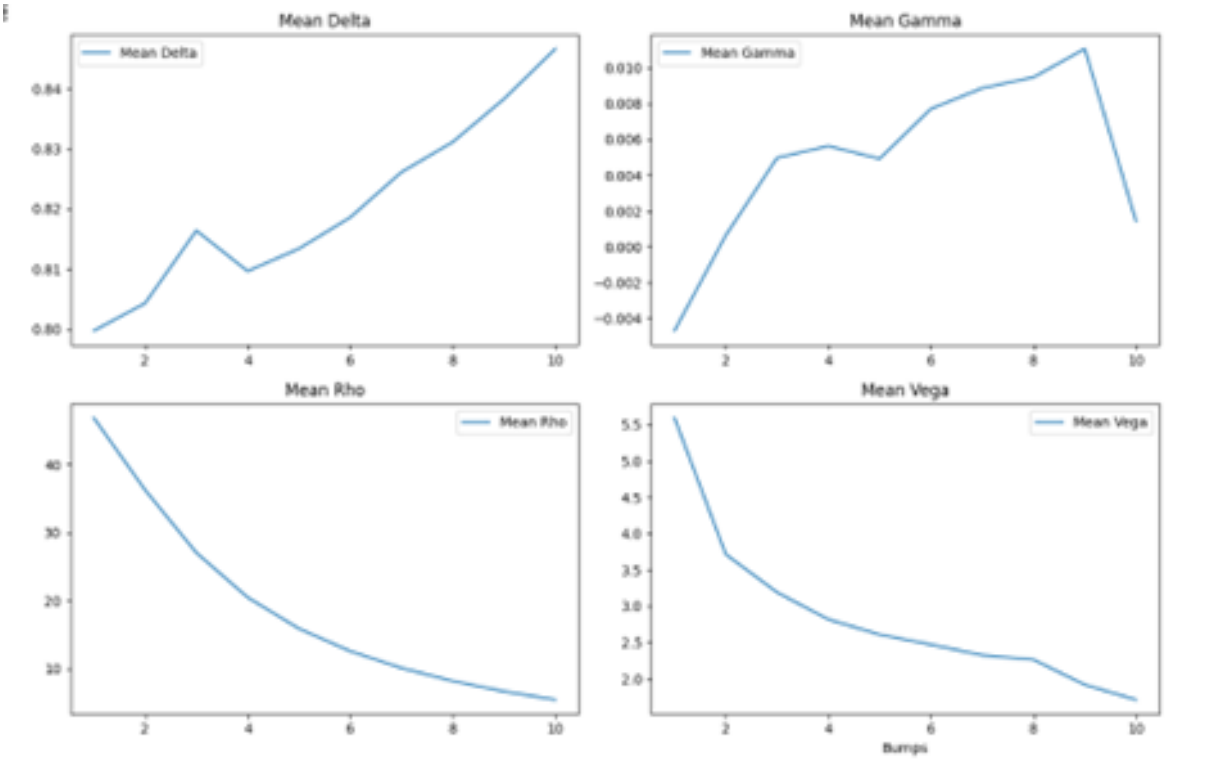

Figure 2: greeks mean values for each bump size

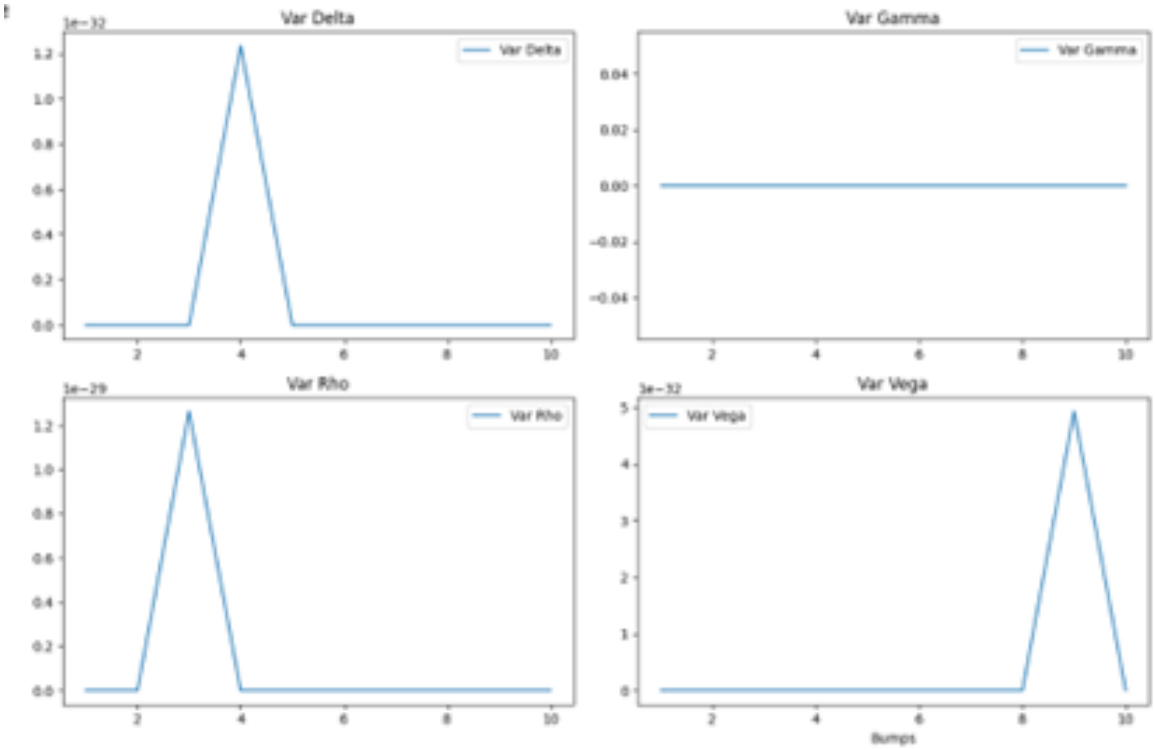

Figure 3: greeks variance for each bump size

The delta increases as the bump size increases, but it remains around 0.8. The gamma also increases until the bump size 9. Rho and vega are decreasing as the bump size increases. As we can observe from the results, we obtain very low variances, but the model is still not very precise. If we take the example of the gamma, as we are working on a call barrier, it must be between 0 and the infinite. For very little variations, the call price change can be very little, leading even sometimes to negative differences between the call price for S0+bump, the call price for S0-bump and 2 times the call price for S0, and thus to a negative gamma. Moreover, with little differences, the result is amplified. That's why we did not take values under 1: if we take a bump of 0.01, it is equal to 1/100, so dividing the call price differences by 0.01 is equal to multiplying it by 100. This a problem for multiple reasons. First, regarding the scale of vega and rho, it would have been better to choose bump sizes in the same range. Secondly, the closer to zero the bump is, the more precise the results obtained by first-order finite differences. Thus this method seems easier and faster to use than the following ones but is less precise.

The final greeks retained for the first-order finite differences are summarized in the table :

	Delta	Gamma	Rho	Vega
Value	0.820421	0.005	18.937415	2.859141

Figure 4: greeks final values for first-order finite differences method

6. Implement a function `StandardError()` which takes the number of paths and the payoff vector as arguments and returns the standard error of one pricing by Monte Carlo. Plot the 95 percent confidence interval for different values of the number of paths. Interpret the results.

In this part, we compute the Standard Error as the ratio between the standard deviation of the payoffs and the square root of the number of payoffs.

```
def StandardError(nb_paths, payoff):
  standard_error=np.std(payoff)/np.sqrt(nb_paths)
  return standard_error
```

We then make the number of paths of our Heston model vary from 10 to 10000 with a step of 100, and compute the standard error of the payoffs obtained to analyze how the standard error of one Monte Carlo pricing evolves with different values of number of paths.

```
for nb_paths in range(10,10000,100):
  payoff=MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r,
  ↪  S0, V0, strike, barrier, T)[1]
  standard_error=StandardError(nb_paths, payoff)
  standard_errors.append(standard_error)
  payoffs.append(payoff)
```

We compute the 95 percent confidence interval for the various number of paths. To do so, we compute the lower bound and the upper bound of the interval using the formula:

$$\text{IC} = [\text{mean} - z * \text{standard error};\ \text{mean} + z * \text{standard error}] \tag{5}$$

where z is the value from the standard normal distribution for the selected confidence level. Since z is a well-known result for the 95 percent confidence interval, we will directly use its value, which is 1.96.

```
nb_paths=[x for x in range(10,10000,100)]
lower_bounds=[]
upper_bounds=[]
z=1.96

for i in range(len(standard_errors)):
  lower_bounds.append(np.mean(payoffs[i])-z *standard_errors[i])
  upper_bounds.append(np.mean(payoffs[i]) +z *standard_errors[i])
```

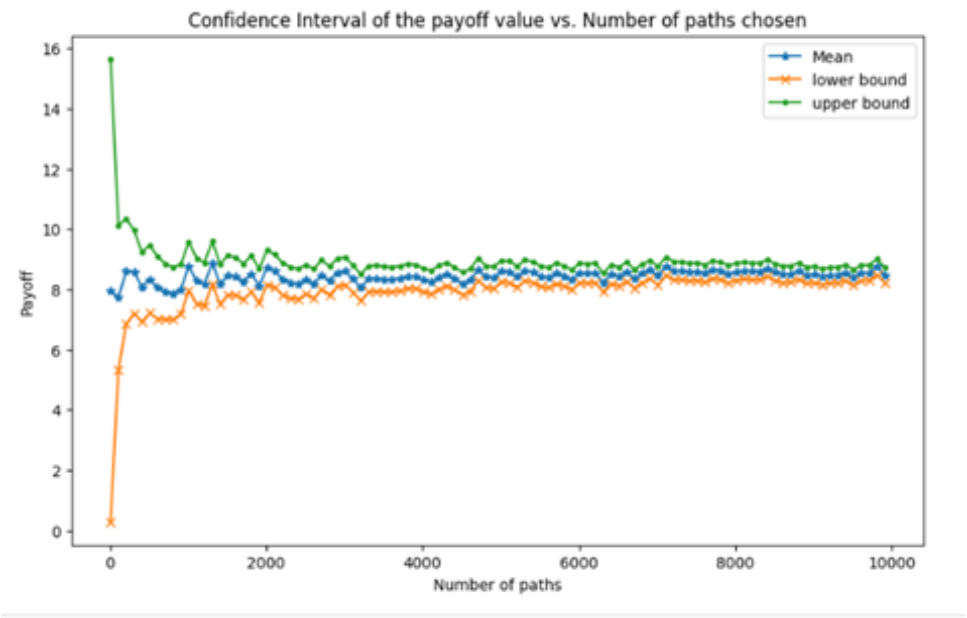

Figure 5: 95 % confidence interval for different values of the number of paths.

In 5, the blue curve represents the mean payoff for each number of paths used in the simulation. It's the average payout we would expect if the options were exercised. The other bounds are the range within which the true mean payoff is expected to lie with 95% certainty.

The width of the confidence interval is determined by the standard error. As we can see, the higher the number of paths, the tighter the confidence interval. This result is not surprising and is what was expected. Indeed, by raising the number of paths, our model takes more and more into account all the possible values that can be taken by the underlying, thus increasing the precision. Therefore to be as precise as possible, our model must generate a really high number of paths, but then we lose some speed of execution.

7. (Bonus) The ν is the sensitivity of the option with respect to the volatility. Since the

volatility is stochastic in the Heston model, propose a method for estimating the ν and justify the approach.

For this part, we constructed the `VegaFD()` function and then added it to the code in the part 5 to compute ν at the same time we were computing the other Greeks since the loop and the bumps were identical.

```
def VegaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0, V0,
 ↪  strike, barrier, maturity, v_bump, mc_plus_bump_v, mc):
  vega=(mc_plus_bump_v-mc)/v_bump
  return vega

for i in range(len(bumps)):
  ...
  v_bump= bumps[i]
  vega_list=[]

  for j in range(5):
    ...
    mc_plus_bump_v = MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0 + v_bump, strike,barrier, T)[0]
    mc = MC_Pricing(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r,
     ↪  S0, V0, strike,barrier, T)[0]


    ...
    vega_list.append(VegaFD(seed, nb_paths, nb_steps, avg_variance, k, gamma,
     ↪  corr_S_V, r, S0, V0, strike,barrier, T, v_bump, mc_plus_bump_v, mc))
  ...
  vega_final.append(vega_list)
```

We estimate the ν (Vega) using the `VegaFD()` function. When performing Monte Carlo simulations, two sets of option prices are computed: one with the original volatility parameter and another with a bumped volatility.

In the simulations, `mc_plus_bump_v` represents the option price calculated with the bumped volatility $V_0 + \text{v\_bump}$, and `mc` represents the option price with the original volatility V_0. The `VegaFD` function is then called with these two prices to estimate ν for each bump size.

The `VegaFD()` function calculates ν (Vega) by dividing the difference in these option prices by the magnitude of the volatility perturbation.

The reasoning here is that since the volatility is stochastic, the slight perturbation added to

V0 will be diffused to the value of the underlying for each time step in our Heston model since we use vt-1 to compute the underlying value at time t. This will then impact the payoff and thus the price of the option. That's why we can use first-order finite differences to compute ν.

1.2 Automatic Adjoint Differentiation Approach

In this section, you will benefit from the PyTorch implementation. Indeed, PyTorch gives you automatically the computation graph which makes AAD transparent for you. Therefore, you need to work with tensors instead of variables. Note that you may need to adapt your code of the Euler-Maruyama approximation of the Heston model to be able to back-propagate the derivatives into the computation graph. Hint: the `.clone()` method should be useful to prevent overwriting the tensors.

1. Apply the AAD approach to the computation of the Greeks and compare it to the results obtained by bumping. Note that to calculate the gamma, it requires remaking the computation graph for the delta.

In this part, we will use the functions `GeneratePathsHestonEuler()`, `Payoff()` and `MC Pricing()` constructed earlier, but we will slightly modify them to have tensors as initial parameters. This will give us the final price of the option as a tensor too, and then we will be able to obtain the derivatives for the parameters that we are interested in (S0 for δ and γ, r for ρ and V0 for ν) using a backward pass on the generated computation graph. We use a backward pass since we have many initial parameters and only one output which is our down and out call barrier option price.

```
def GeneratePathsHestonEulerAAD(seed, avg_variance,k,gamma,
↪  corr_S_V,r,S0,V0,T,nb_paths, nb_steps):
  torch.manual_seed(seed)
  dt=T/nb_steps

  s_t = torch.zeros((nb_steps + 1, nb_paths))
  v_t = torch.zeros((nb_steps + 1, nb_paths))
  s_t[0, :] = S0
  v_t[0, :] = V0

  W_V = torch.normal(0,1,size=(nb_steps, nb_paths))
  W_S = corr_S_V * W_V + torch.sqrt(1 - corr_S_V ** 2) *
  ↪  torch.normal(0,1,size=(nb_steps, nb_paths)) #Création d'un BM corrélé à W_V
```

```
  for j in range(nb_paths):
    for i in range(1,nb_steps+1):
      s_t_moins_1=s_t[i-1, j].clone()
      s_t[i, j] = (s_t_moins_1 + r * s_t_moins_1 * dt +
      torch.sqrt(v_t[i-1, j] * dt) * s_t_moins_1 * W_S[i-1, j])
      v_t[i,j]=torch.abs(v_t[i-1,j]+k*(avg_variance-v_t[i-1,j])*dt+gamma*
      torch.sqrt(v_t[i-1,j])*W_V[i-1,j])
  return s_t

def PayoffAAD(strike, barrier, r, maturity, asset_prices):
  broken_barrier=False
  for price in asset_prices:
    if price<=barrier:
      broken_barrier=True

  if broken_barrier==False:
    payoff=torch.exp(-r*maturity)*torch.max(torch.tensor(0),
    asset_prices[len(asset_prices)-1]-strike)
  else:
    payoff=torch.exp(-r*maturity)*torch.max(torch.tensor(0),
    asset_prices[len(asset_prices)-1]-strike)*torch.tensor(0)
  return payoff

def MC_PricingAAD(seed, nb_paths, nb_steps, avg_variance, k, gamma, corr_S_V, r, S0,
↪  V0, strike, barrier, maturity):
  matrix_Heston_path=GeneratePathsHestonEulerAAD(seed, avg_variance, k, gamma,
  ↪  corr_S_V, r, S0, V0, maturity, nb_paths, nb_steps)
  matrix_Heston_path=torch.t(matrix_Heston_path)

  payoffs=torch.zeros((nb_paths))
  for i in range(nb_paths):
    payoffs[i]=PayoffAAD(strike, barrier, r, maturity, matrix_Heston_path[i])
  option_price= torch.mean(payoffs)
  return option_price

seed_tensor=torch.tensor(123)
nb_paths_tensor=torch.tensor(1000)
nb_steps_tensor=torch.tensor(100)

avg_variance_tensor=torch.tensor(0.5)
k_tensor=torch.tensor(0.1)
gamma_tensor=torch.tensor(0.1)
```

```
corr_S_V_tensor=torch.tensor(-0.9)
r_tensor = torch.tensor(0.02, requires_grad=True)
S0_tensor=torch.tensor(100., requires_grad=True)
V0_tensor=torch.tensor(0.1, requires_grad=True)
K_tensor = torch.tensor(100.)
B_tensor = torch.tensor(90.)
T_tensor = torch.tensor(1)

price=MC_PricingAAD(seed_tensor, nb_paths_tensor, nb_steps_tensor,
 ↪  avg_variance_tensor, k_tensor, gamma_tensor, corr_S_V_tensor,
 ↪  r_tensor,S0_tensor,V0_tensor, K_tensor, B_tensor, T_tensor)

delta = torch.autograd.grad(price, S0_tensor, create_graph=True)
rho=torch.autograd.grad(price, r_tensor, create_graph=True)
vega=torch.autograd.grad(price, V0_tensor, create_graph=True)

gamma = torch.autograd.grad(delta, S0_tensor, allow_unused=True, create_graph=True)

greeks_AAD_df=pd.DataFrame({
    "Delta":delta[0].item(),
    "Gamma":gamma[0],
    "Rho":rho[0].item(),
    "Vega":vega[0].item()
}, index=["Value"])

greeks_AAD_df
```

	Delta	Gamma	Rho	Vega
Value	0.484916	None	38.814877	295.65741

Figure 6: Values of the greeks obtained by AAD

As can be seen in the results, we are unable to compute the gamma of the option using AAD. This is linked to the type of our option. Our payoffs are discontinuous since they are 0 if the underlying breaks down the barrier, thus it is not possible to compute the gamma using the grad function For the other Greeks, the δ and the ρ are in the same range that the values found using first-order finite differences, between 0 and 1 for the delta and between 10 and 50 for the rho.

2. (Bonus) Can you notice a strong difference for the ν computation? What do you conclude?

The ν is way higher than the value previously found. This may be because in our first-order finite differences, we took several small bump sizes, but not small enough to tend to zero (which is better in terms of precision but leads to an increase in truncate errors), whereas the AAD algorithm is more precise in using variations that tends to 0.

2 Pricing and Hedging by Differential Deep Learning

2.1 Dataset Generation

In this section, we are going to generate a dataset a la LSM [4] composed of the initial states, the payoffs, and the differentials of the initial states with respect to the payoffs. We denote X the training samples (i.e., the initial states), Y the labels (i.e., the payoffs), and $dYdX$ the pathwise differentials computed by AAD.

1. Implement a function `HestonLSM()` which takes the seed, the number of samples to generate, the number of paths, the number of steps, the model parameters, and the option parameters in arguments and returns the initial states, the payoffs, and the differentials of the payoff with respect to the inputs computed by AAD. The initial value of the asset denoted S_0 is taken into an equally spaced range from 10 to 200.

```
def HestonLSM(seed_tensor, avg_variance_tensor,k_tensor,gamma_tensor,
 ↪  corr_S_V_tensor,r_tensor,S0_tensor,V0_tensor,T_tensor,nb_paths_tensor,
 ↪  nb_steps_tensor, K_tensor, B_tensor):

  matrix_Heston_path=GeneratePathsHestonEulerAAD(seed_tensor,
   ↪  avg_variance_tensor,k_tensor,gamma_tensor,
   ↪  corr_S_V_tensor,r_tensor,S0_tensor,V0_tensor,T_tensor,nb_paths_tensor,
   ↪  nb_steps_tensor)
  matrix_Heston_path=torch.t(matrix_Heston_path)
  print(len(matrix_Heston_path))

  payoffs=torch.zeros((nb_paths_tensor))
  gradients = torch.zeros_like(payoffs)
  for i in range(nb_paths_tensor):
    payoffs[i]=PayoffAAD(K_tensor, B_tensor, r_tensor, T_tensor,
     ↪  matrix_Heston_path[i])

  X = matrix_Heston_path[1, :]
```

```
  return X.reshape([-1, 1]), payoffs, gradients

X,Y,dYdX=HestonLSM(seed_tensor, avg_variance_tensor,k_tensor,gamma_tensor,
 ↪  corr_S_V_tensor,r_tensor,S0_tensor,V0_tensor,T_tensor,nb_paths_tensor,
 ↪  nb_steps_tensor, K_tensor, B_tensor)
```

The `HestonLSM` function uses the `GeneratePathsHestonEulerAAD`, for simulating paths for underlying asset prices with considerations for stochastic volatility for the Heston model. The function then initializes tensors for storing payoffs and gradients of each path. It calculates the payoff for each path using the `PayoffAAD` function, which assesses the interaction of asset prices with the barrier. The function finally extracts and reshapes the state of the system (asset prices at a particular time step) and aligns them with the computed payoffs (we get tuples as output).

2. Implement a function `normalize data()` which takes the X, Y, and $dYdX$ as inputs and returns the normalized X, the mean of X, the std of X, the normalized Y, the mean of Y, the std of Y, the normalized $dYdX$, and the value λ_j computed as $\lambda_j = \sqrt{\frac{1}{N}\sum(dy/dx)^2}$ where dy/dx is the normalized version of $dYdX$. The value λ_j is called the differential weights of the cost function. Each differential with respect to input parameter j is also scaled by the average magnitude of the normalized differential with respect to j in the training set so as to let each differential have a similar magnitude in the loss function. Explain why it is important to work with normalized data in machine learning.

```
def normalize_data(X, Y, dYdX):
    X_mean, X_std = torch.mean(X), torch.std(X)
    Y_mean, Y_std = torch.mean(Y), torch.std(Y)
    dYdX_mean, dYdX_std = torch.mean(dYdX), torch.std(dYdX)
    X_normalized = (X-X_mean)/X_std
    Y_normalized = (Y-Y_mean)/Y_std
    dYdX_normalized = (dYdX-dYdX_mean)/dYdX_std
    n = len(X_normalized)
    lambda_j =1/torch.sqrt((1/n)*torch.sum(dYdX_normalized**2))

    return X_normalized,X_mean, X_std, Y_normalized, Y_mean, Y_std, dYdX_normalized,
     ↪  lambda_j
```

Normalisation is part of feature scaling, [5] where we preprocess the data. This process enhances data analysis and modeling accuracy by mitigating the influence of varying scales on the ML models. It is also known as the min max scaling, and the final value is in the range

0 to 1. Normalization transforms data to a common scale without distorting differences in the ranges of values. It makes it easier for the model to learn and converge.

2.2 Implementation of a Twin Network

In this section, you will implement a twin network [3] as defined in the last lecture. This particular neural network architecture allows learning accurate pricing functions as well as sensitivities. 1. Implement a class `Twin Network()` which inherits from the `torch.nn.Module` class. This way, the backward method will be inferred automatically. This deep neural network has a feedforward architecture with 4 hidden layers of 20 neurons each. These neurons are activated using the ReLu activation function. The output layer is made of one neuron which should not be activated (remember that we are in a regression problem). Define properly the `init` (number of inputs, number of hidden layers, number of neurons) and the `forward` methods to take into account this architecture. Use He initialization [2] (already implemented in PyTorch) to prevent the gradient vanishing problem.

Our code is inspired from [6].

```
class TwinNetwork(nn.Module):
    def __init__(self, input_size):
        super(TwinNetwork, self).__init__()

        #20 neurons for each hidden layer
        self.fc1 = nn.Linear(input_size, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 20)
        self.fc4 = nn.Linear(20, 20)
        self.out = nn.Linear(20, 1) #output layer with one neuron

        #weights
        nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc3.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc4.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.out.weight, nonlinearity='linear')
```

The `TwinNetwork` class, defines the deep learning model . This model features four hidden layers, each consisting of 20 neurons. The input size for the first layer is dynamically

set based on the `input size` parameter. Each of these layers is followed by a ReLU activation function, which introduces non linearity to the model. The output of the network is generated through a single neuron in the final layer, which does not use the activation function. The network's weights are initialized using He (Kaiming) initialisation which helps in maintaining a consistent scale of gradients during backpropagation, preventing the vanishing or exploding gradients problem [7].

```
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.out(x)
        return x
```

The forward method defines the data flow through the neural network, applying the ReLU (Rectified Linear Unit) activation after each hidden layer before passing the result through the output layer.ReLU is a piecewise linear function that replaces all negative input values with zero and leaves positive values unchanged. It is the default for most types of convutionnal neural networks [8].

2. Implement a method `predict price()` of the `Twin Network` class which takes as argument the input X to predict, the mean of X, the std of X, the mean of Y, and the std of Y. In this method, first, you normalize X, then you predict the values of Y given X, and finally, you unscale the predicted values Y.

```
    def predict_price(self, X, X_mean, X_std, Y_mean, Y_std):
        X_norm = (X - X_mean) / X_std
        Y_pred = self.forward(X_norm)
        Y_pred_unscaled = (Y_pred * Y_std) + Y_mean

        return Y_pred_unscaled
```

We normalise X here directly, since it is quicker than using the previsous function in terms of computation time.

3. Implement a method `predict price and diffs()` of the `Twin Network` class which takes as argument the input X to predict, the mean of X, the std of X, the mean of Y, the std of Y. In this method, first, you normalize X, second, you predict the values of Y given

X, third, you compute the gradient of the outputs with respect to the inputs (hint: you should use the `torch.autograd` method) and finally, you unscale the predicted values Y and $dYdX$.

```
    def predict_price_and_diffs(self, X, X_mean, X_std, Y_mean, Y_std):
        X_norm = (X - X_mean) / X_std
        X_norm.requires_grad_(True)
        Y_pred = self.forward(X_norm)
        self.zero_grad()
        Y_pred.backward(torch.ones_like(Y_pred))
        dYdX = X_norm.grad
        Y_pred_unscaled = (Y_pred.detach() * Y_std) + Y_mean
        dYdX_unscaled = dYdX * Y_std / X_std

        return Y_pred_unscaled, dYdX_unscaled
```

2.3 Neural Network Training

1. Implement a function `training()` which takes as argument the `Neural Network()` class instantiated, the normalized training samples X, the normalized training labels Y, the normalized differential training labels $dYdX$, λ_j, and the number of epochs necessary to train the model.

We want this function to be able to test two scenarios: (1) training the neural network only on the training samples (that is without the differential labels) and (2) training the neural network on the training samples and the differentials. Be careful, the loss must change according to the scenario we consider. The cost functions for both scenarios are given in the appendix B as a reminder.

The sequential organization of this function is as follows: 1. If we are in scenario (2) compute $\alpha = \frac{1}{1+N}$, where N is the number of inputs. 2. Define MSE as the cost function 3. Define the Adam optimizer with a fix learning rate of 0.1. Keep the other default parameters. 4. Enter the optimization loop: compute the predictions and the cost according to the scenario you are in, backpropagate the gradients to optimize the network weights and biases, and store the cost for each epoch.

```python
def training(network, X_train, Y_train, dYdX_train, lambda_j, epochs, scenario):
    if scenario not in [1, 2]:
        raise ValueError("Choose 1 for training without differentials or 2 with
        ↪  differentials.")

    if scenario == 2:
        X_train = X_train.clone().detach().requires_grad_(True)
        N = X_train.size(1)#nbr of inputs
        alpha = 1/(1 + N)
    else:
        alpha = None

    criterion = nn.MSELoss() #  MSE
    optimizer = optim.Adam(network.parameters(), lr=0.1) # ADAM OPTIMISER (learning
    ↪  rate =0.1)
    cost_per_epoch = []

    for epoch in range(epochs):
        optimizer.zero_grad()
        Y_pred = network(X_train)

        if scenario== 1:
            loss = criterion(Y_pred, Y_train)
        else:
            Y_pred.backward(torch.ones_like(Y_pred), retain_graph=True)
            dYdX_pred = X_train.grad
            loss_Y = criterion(Y_pred, Y_train)
            print('Y', loss_Y)
            loss_dYdX = criterion(dYdX_pred, dYdX_train)
            print('DYDX', loss_dYdX)
            loss = alpha*loss_Y+(1-alpha)*loss_dYdX
            X_train.grad.zero_()

        loss.backward() #backpropagate
        optimizer.step()
        cost_per_epoch.append(loss.item())

    return cost_per_epoch
```

2.4 Model Comparison

In this section, we want to compare the standard deep neural network and the deep differential neural network.

1. Create a training set consisting of 1000 samples using the `HestonLSM()` function.

```
seed_tensor=torch.tensor(123)
nb_paths_tensor=torch.tensor(1000)
nb_steps_tensor=torch.tensor(10)
avg_variance_tensor=torch.tensor(0.5)
k_tensor=torch.tensor(0.1)
gamma_tensor=torch.tensor(0.1)
corr_S_V_tensor=torch.tensor(-0.9)
r_tensor = torch.tensor(0.02, requires_grad=True)
S0_tensor=torch.tensor(100., requires_grad=True)
V0_tensor=torch.tensor(0.1)
K_tensor = torch.tensor(100.)
B_tensor = torch.tensor(90.)
T_tensor = torch.tensor(1)

X2,Y2,dYdX2 = HestonLSM2(seed_tensor, avg_variance_tensor,k_tensor,gamma_tensor,
↪  corr_S_V_tensor,r_tensor,S0_tensor,V0_tensor,T_tensor,nb_paths_tensor,
↪  nb_steps_tensor, K_tensor, B_tensor)
```

2. Normalize the data using the `normalizeData()` function on the generated dataset.

```
X2,Y2,dYdX2 = HestonLSM2(seed_tensor, avg_variance_tensor,k_tensor,gamma_tensor,
↪  corr_S_V_tensor,r_tensor,S0_tensor,V0_tensor,T_tensor,nb_paths_tensor,
↪  nb_steps_tensor, K_tensor, B_tensor)
```

3. Instantiate two networks with the `TwinNetwork()` class having the following architecture: 4 hidden layers consisting of 20 neurons each. Don't forget the seed.

```
twin_network = TwinNetwork(input_size=1)

X2_sample = torch.tensor(X2).float()
X2_mean, X2_std = torch.mean(X2_sample).float(), torch.std(X2_sample).float()
Y2 = torch.tensor(Y2)
Y2_mean, Y2_std = torch.mean(Y2), torch.std(Y2)
predicted_price2 = twin_network.predict_price(X2_sample, X2_mean, X2_std, Y2_mean,
↪  Y2_std)
```

```
predicted_price2, price_diffs2 = twin_network.predict_price_and_diffs(X2_sample,
↪  X2_mean, X2_std, Y2_mean, Y2_std)
```

4. Fix the number of epochs to 100. Run the classical neural network and the differential neural network. Plot in the same figure the cost of both neural networks for all epochs. Interpret this graph.

```
costs_classical_neural_network = training(twin_network,
↪  torch.tensor(X2_normalized).float(), torch.tensor(Y2_normalized).float(),
↪  torch.tensor(dYdX2_normalized).float(), lambda_j2, epochs=100, scenario=1)
costs_differential_neural_network = training(twin_network,
↪  torch.tensor(X2_normalized).float(), torch.tensor(Y2_normalized).float(),
↪  torch.tensor(dYdX2_normalized).float(), lambda_j2, epochs=100, scenario=2)
```

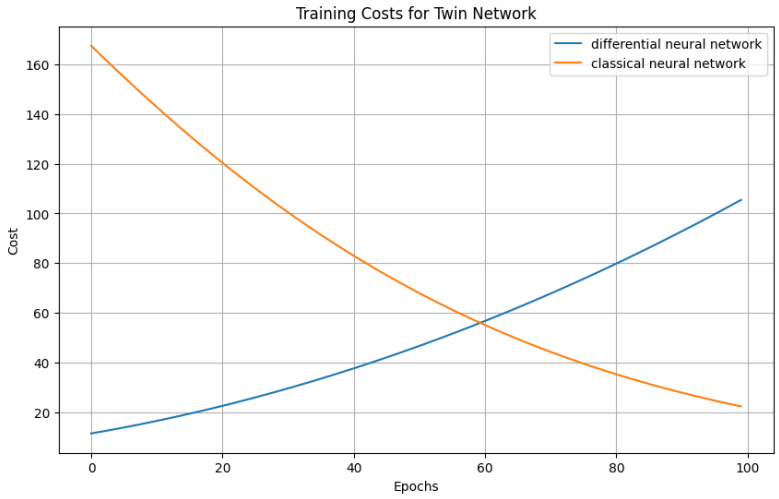

Figure 7: Training cost for Twin Network

As we can see on this graph, using differential neural network doesn't help reducing our cost if we use a lot of epochs. The classical neural network produces satisfying results.

5. Can you imagine a simple procedure to estimate the uncertainty of the results given by a deep neural network? Implement it and give the results for both neural networks (hint: you are allowed to change the seed to answer this question).

An effective approach to assess the uncertainty in the predictions of a deep neural network is the implementation of Monte Carlo Dropout. This technique diverges from the traditional use of dropout layers, which are typically employed during the training phase. In this method, dropout is also activated during the inference phase. This unique application enables the production of a varied set of predictions for the same input. Therefore, we can evaluate the network's prediction uncertainty. The initial step involves incorporating dropout layers into the neural network's architecture and training the network in the standard manner. Then, contrary to conventional practices where dropout is deactivated during inference, we maintain dropout activation during this phase as well. This is crucial for generating a spectrum of predictions. Following this, we run the network numerous times with the same input. In each iteration, due to the effect of dropout, a different set of neurons is omitted from the network. This results in a slight variation in the output each time. Finally, we analyze the variation in these outputs and measure the uncertainty using statistical metrics like the standard deviation of the series of predictions.

```
class TwinNetwork2(nn.Module):
    def __init__(self, input_size, dropout_rate=0.5):
        super(TwinNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 20)
        self.dropout1 = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(20, 20)
        self.dropout2 = nn.Dropout(dropout_rate)
        self.fc3 = nn.Linear(20, 20)
        self.dropout3 = nn.Dropout(dropout_rate)
        self.fc4 = nn.Linear(20, 20)
        self.dropout4 = nn.Dropout(dropout_rate)
        self.out = nn.Linear(20, 1)


    def forward(self, x, apply_dropout=False):
        x = F.relu(self.fc1(x))
        if apply_dropout: x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        if apply_dropout: x = self.dropout2(x)
        x = F.relu(self.fc3(x))
        if apply_dropout: x = self.dropout3(x)
        x = F.relu(self.fc4(x))
        if apply_dropout: x = self.dropout4(x)
        x = self.out(x)
        return x
```

```
    def predict_price(self, X, X_mean, X_std, Y_mean, Y_std, apply_dropout=False):
        X_norm = (X - X_mean) / X_std
        Y_pred = self.forward(X_norm, apply_dropout=apply_dropout)
        Y_pred_unscaled = (Y_pred * Y_std) + Y_mean
        return Y_pred_unscaled

    def predict_price_and_diffs(self, X, X_mean, X_std, Y_mean, Y_std):
        X_norm = (X - X_mean) / X_std
        X_norm.requires_grad_(True)
        Y_pred = self.forward(X_norm)

        self.zero_grad()
        Y_pred.backward(torch.ones_like(Y_pred))
        dYdX = X_norm.grad

        Y_pred_unscaled = (Y_pred.detach() * Y_std) + Y_mean
        dYdX_unscaled = dYdX * Y_std / X_std

        return Y_pred_unscaled, dYdX_unscaled

def uncertainty_calculation(model, X, X_mean, X_std, Y_mean, Y_std,
↪  n_iterations=100):
    predictions = []
    for i in range(n_iterations):
        Y_pred_unscaled = model.predict_price(X, X_mean, X_std, Y_mean, Y_std,
        ↪  apply_dropout=True)
        predictions.append(Y_pred_unscaled.detach().numpy())

    predictions = np.array(predictions)
    uncertainty = np.std(predictions, axis=0)

    return uncertainty

uncertainty = uncertainty_calculation(twin_network2, X, X_mean, X_std, Y_mean, Y_std)
```

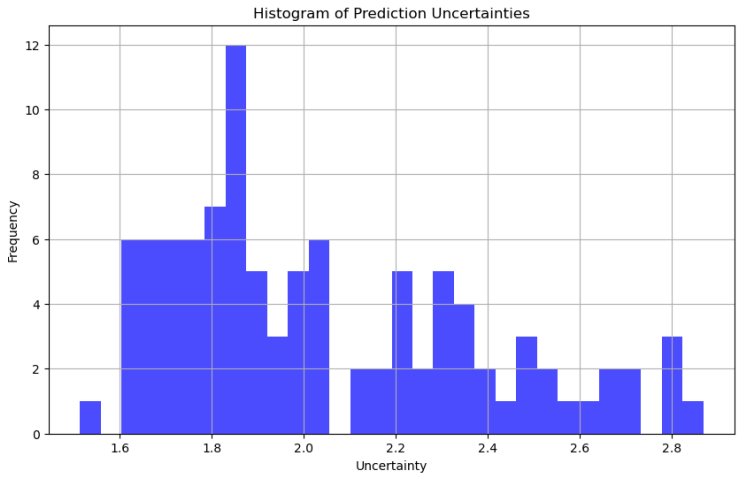

Figure 8: Prediction uncertainties for Twin Network

The histogram of the uncertainties of our predictions from our Twin Network reveals a distribution that is primarily concentrated towards the lower end of the uncertainty scale, with a majority of the predictions exhibiting relatively low uncertainty. This suggests that the model is quite confident in the predictions for most of the data points. However, the presence of bars further along the uncertainty axis indicates that there are instances where the model's predictions are less certain. The frequency of these higher uncertainty predictions is notably less, but they are not negligible, implying that for a subset of the data, the model's confidence significantly decreases. It is important to note that there are no extreme outliers, suggesting that the model does not frequently encounter instances where it is highly uncertain. The absence of a long tail on the higher end of the scale also indicates that the model does not often produce wildly varying predictions. Overall, the graph indicates a model that is generally reliable, with pockets of data that may require further investigation to understand why the model is less certain and whether these points share any common features or characteristics that could be addressed to improve the model's performance.

3 Conclusion

1. What is the interest of having "seeded" all the results?

The seed function implies that the randomly generated numbers will be fixed. It is used to save the state of a random function so that it can generate the same random numbers on multiple executions of the code on the same machine or on different machines (for a specific seed value). In our case, it allows us to generate the same paths for the underlying values

when we call our GeneratePathsHestonEuler functions. It ensures that the greeks will be computed on the same objects for all the methods, and so that the results are truly comparable. [9].

2. What is the advantage of using a neural network to do pricing compared to a Monte-Carlo pricer?

Using neural networks for financial pricing offers several benefits over the Monte Carlo method. Neural networks can provide faster and more efficient pricing, and accurately price a wide range of scenarios. They can recognize data patterns that might not be apparent in standard models. Neural networks also tend to have reduced error and variance compared to Monte Carlo simulations. However, we still have to deal with challenges like potential overfitting, the need for a lot of training data, and the lack of transparency (we don't necessarily know how they reach their output), so we may need to add explainable AI [10] like SHAP or PDP.

3. Create a table that summarizes the results of all approaches: for all models (FD, AAD, Neural network), report data generation time, pricing and Greeks computation time, convergence given the number of observations/convergence given the scheme (delta time), the uncertainty. What is the best pricer? Why?

To find the uncertainty in the same way for all methods, we iterate our cost functions 10 times in the NN model. We store the values in a list and keep the maximum observed standard deviation.

```
number_of_runs = 10

for seed in range(number_of_runs):
    seed_tensor = torch.tensor(seed)
    X2, Y2, dYdX2 = HestonLSM2(seed_tensor, avg_variance_tensor, k_tensor,
    ↪  gamma_tensor, corr_S_V_tensor, r_tensor, S0_tensor, V0_tensor, T_tensor,
    ↪  nb_paths_tensor, nb_steps_tensor, K_tensor, B_tensor)
    X2_normalized, X2_M, X2_S, Y2_normalized, Y2_M, Y2_S, dYdX2_normalized, lambda_j2
    ↪  = normalize_data(X2, Y2, dYdX2)
    twin_network = TwinNetwork(input_size=1)
    X2_sample = torch.tensor(X2).float()
    Y2 = torch.tensor(Y2)
    costs = training(twin_network, torch.tensor(X2_normalized).float(),
    ↪  torch.tensor(Y2_normalized).float(), torch.tensor(dYdX2_normalized).float(),
    ↪  lambda_j2, epochs=epoch_count, scenario=scenario)
    final_costs.append(costs[-1])
```

```
std_NN = np.max(final_costs)

results = {"FD": {}, "AAD": {}, "NN": {}}
results["FD"]["Computation Time"] = round(end_time_FD - start_time_FD, 3)
results["AAD"]["Computation Time"] = round(end_time_AAD - start_time_AAD, 3)
results["NN"]["Computation Time"] = round(end_time_NN - start_time_NN, 3)

results["FD"]["Greek Calculation Time"] = round(end_time_GFD - start_time_GFD, 3)
results["AAD"]["Greek Calculation Time"] = round(end_time_GAAD - start_time_GAAD, 3)
results["NN"]["Greek Calculation Time"] = round(end_time_GNN - start_time_GNN, 3)

results["FD"]["Uncertainty"] = std_FD
results["AAD"]["Uncertainty"] = std_AAD
results["NN"]["Uncertainty"] = std_NN

print(results)
```

In table 9 are the results of the 3 models.

Method	Computation Time	Greek Calculation Time	Uncertainty
FD	59.467	0.008	826.706
AAD	71.857	37.284	15.299
NN	68.511	0.025	0.909

Figure 9: Result Comparison of FD, AAD, and NN Methods

The AAD uses a lot of RAM and doesn't always give results when the number of simulated paths is too high (runs out of disk space before giving output). The AAD method, despite its precision, falls short for every category compared to the other models.

For the best pricer if the priority is speed, especially for Greeks calculation, FD stands out, for 10 simulation paths the speed is correct. However, the uncertainty (reflected in 9) is quite high, the average is low but we have outliers when iterating the simulation. If we take 10 000 paths for the simulation, the code takes over 20 min to finish running for this model .

If the focus is on precision and managing uncertainty, the NN method is far superior, offering high accuracy with relatively low uncertainty (and a relatively good compuation time as well).

The Neural Network approach appears to be the most balanced in terms of computation time and accuracy, especially considering its low uncertainty and fast Greek calculation time.

Below are the values we get for the Greeks using the different methods:

Greek	FD	AAD	NN
Delta	0.820421	0.484916	0.14030766
Gamma	0.005	None	None
Rho	18.937415	38.814877	8.33382
Vega	2.859141	295.65741	None

Table 1: Comparison of Greeks obtained by different methods

The FD method appears to provide a more balanced set of Greeks, AAD seems to offer higher sensitivity to interest rates and volatility, and NN appears to be the least sensitive, which might be either advantageous or detrimental depending on the market conditions.

4. Give the advantages and drawbacks of all the techniques presented above.

Neural Networks stand out for their flexibility and precision, capable of modeling complex financial structures through their capacity to approximate any continuous function. This adaptability comes at the cost of a considerable data requirement and potential overfitting, to which we must add the drawback of poor interpretability (important in model risk management). On the other hand, once an NN is adequately trained, it can provide rapid calculations of prices and Greeks.

Likewise, AAD has a good accuracy in derivative calculation and efficiency in computing multiple Greeks simultaneously, an advantage over the FD method. However, the complexity of its implementation and high resource consumption is problematic. FD methods offer a more intuitive and straightforward approach but can struggle with numerical errors and have stability issues.

References

[1] J. C. Cox, J. E. Ingersoll, and S. A. Ross, "A theory of the term structure of interest rates," *Econometrica* **53**, 385–407 (1985).

[2] K. He, X. Zhang, S. Ren, *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," (2015).

[3] B. Huge and A. Savine, *Differential Machine Learning* (2020).

[4] F. A. Longstaff and E. S. Schwartz, "Valuing american options by simulation: a simple least-squares approach," *The Review of Financial Studies* **14**, 113–147 (2001).

[5] A. Bhandari, "Feature scaling: Engineering, normalization, and standardization," (2024).

[6] A. Savine, "Differential deep learning, article: https://arxiv.org/abs/2005.02347," (2020).

[7] S. Goel, "Kaiming he initialization," (2019).

[8] J. Brownlee, "A gentle introduction to the rectified linear unit (relu)," (2020).

[9] D. Jain, "random.seed( ) in python," (2024).

[10] R. Hashimoto, "Application of machine learning to a credit rating classification model: Techniques for improving the explainability of machine learning," (2023).

Appendix A: Independent Brownian Motion

In this section, it is demanded to simulate two correlated Brownian motions. Here is a general example of how you can simulate it: Let Z_t and WX_t be two independent Brownian motions where $Z_t \sim \mathcal{N}(0, \sqrt{dt})$ and $WX_t \sim \mathcal{N}(0, \sqrt{dt})$. You can then create a Brownian motion WS_t of correlation parameter ρ with WX_t using this simple formula:

$$W_t^S = \sqrt{1-\rho^2} Z_t + \rho W_t^X.$$

You can convince yourself that the relation is true by computing $\text{Var}[W_t^X]$, $\text{Var}[W_t^S]$, and $\text{corr}(W_t^X, W_t^S)$.

Appendix B: Cost Functions

The cost function C of a standard feed-forward network used for regression is the Mean Squared Error (MSE). It is defined as:

$$C_{\text{standard}} = \frac{1}{M} \sum_{i=1}^{M} (y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)})^2,$$

where M is the total number of observations in the training set, $y_{\text{true}}^{(i)}$ is the true payoff value of the option, and $y_{\text{pred}}^{(i)}$ is its prediction by the model.

The cost function C of a differential feed-forward network used for regression is the MSE augmented with the MSE of the differentials scaled by the parameter λ_j. It is defined as:

$$C_{\text{differential}} = \alpha \frac{1}{M} \sum_{i=1}^{M} (y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)})^2 + (1-\alpha) \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{N} \lambda_j (z_{\text{true}}^{(i)}[j] - z_{\text{pred}}^{(i)}[j])^2,$$

where $\alpha = \frac{1}{1+N}$, $\lambda_j = \frac{1}{\| z_{\text{true}}^{(i)}[j] \|^2}$, and $z_{\text{true}}^{(i)}[j]$ is the true (resp. predicted) differential with respect to the input parameter j.