MF810 Project

Forward-Backward Stochastic Neural Networks with Converting Code

Team Members

Congshan Leng
Diego Vivero
Hanrui Deng
Jinjing Ge
Jung Cheng Chang
Shiyue Xia

Introduction

In the report, we will review the work of Maziar Raissi on Forward-Backward Stochastic Neural Networks. There are two main components in this paper - the building and training of neural networks, and the application of the neural network on solving backward-forward stochastic differential equations. Our project extends the existing framework by converting tensorflow 1.0 to tensorflow 2.0. To test the effectiveness of the neural network, we computed the result of 100-dimensions Black-Scholes-Barenblatt equation, and compared it to the explicit solution of the PDE.

Mathematical Background and Neural Network

The curse of dimensionality is a key limit in solving partial differential equations with classical numerical methods. Raissi devised an algorithm that is scalable to high-dimensions by using deep neural networks.

- The general form of coupled Backward-Forward Partial Differential Equation:

$$dX_t = \sigma diag(X_t) dW_t, \quad t \in [0,T], X_0 = \xi \qquad (1)$$

$$dY_t = r(Y_t - Z_t' X_t)dt + \sigma Z_t' diag(X_t) dW_t, \quad t \in [0,T), Y_T = g(X_T) \qquad (2)$$

- It is well-known that coupled forward-backward stochastic differential equations are related to quasi-linear partial differential equations of form.

$$u_t = f(t, x, u, Du, D^2 u) \qquad (3)$$

The terminal condition is $u(T,x) = g(x)$, where $u(t,x)$ is the unknown solution and

$$f(t,x,y,z,\gamma) = \varphi(t,x,y,z) - \mu(t,x,y,z)' z - \frac{1}{2} Tr[\sigma(t,x,y)\sigma(t,x,y)'\gamma] \qquad (4)$$

- We approximate the unknown solution $u(t,x)$ by using a 5-layers deep neural network with 256 neurons per hidden layer. Use the Adam optimizer with mini batches of size 100 (i.e., 100 realizations of the underlying Brownian motion).

We apply chain rule to obtain the required gradient vector $Du(t, X_t)$. And The loss function for the neural network is:

$$\sum_{m=1}^{M} \sum_{n=0}^{N-1} \left| Y_m^{n+1} - Y_m^n - \phi_m^n \Delta t^n - (Z_m^n)' \sum_m^n \Delta W_m^n \right|^2 + \sum_{m=1}^{M} \left| Y_m^N - g(X_m^N) \right|^2 \qquad (5)$$

Conversion of tensorflow 1.0 to tensorflow 2.0

Tensorflow 2.0 is a new version of the tensorflow library released in 2019. It introduced many positive changes, but some of the tensorflow 1.0 functions are not compatible with tensorflow 2.0. Therefore, it is essential for us to adjust the neural network framework to make it work.

Tensorflow provides a command that allows you to swiftly translate a TF 1.0 code into TF 2.0 that replaces the most straightforward functions and reorganizes function arguments. A report is given upon the completion of automatic conversion to show which functions have been updated. By using this functionality we were able to save some time and avoid unnecessary errors. Still, most of the conversion requires to completely change the structure of the code as the philosophy of the language changed significantly. The following are the notable differences between the two versions:

- Neural network building

  To initialize a neural network in tensorflow 1.0, it requires the user to initialize weights and biases as a matrix and assign them randomly to the layers. Furthermore, the activation function has to be manually code. On the other hand, tensorflow 2.0's tf.Sequential( ) allows easy addition of layers with activation functions encoded.

- Placeholder

  Tensorflow 1.0 relies on the variable tf.placeholder, which allows to assign to a variable the value of a function without having defined the input to the function. This makes the structure of the coding opposite to the normal coding in Python. In Tensorflow 1.0 all functions and variables are defined beforehand and then the program is more limited to calling these functions through the tf.session.run() command.

- Behavioral difference

  Although there are existing functions in tensorflow 1.0 being kept in tensorflow 2.0, they do not have identical functions. A notable example in our program is tf.gradient. It gives a result different from what we expected. Therefore, it is replaced by tf.GradientTape, which is more efficient and more flexible with variables.

Applications and Testing

The updated neural network with tensorflow 2.0 is tested with 100 dimensions of Black-Scholes-Barenblatt Equation. In figure 1, 5 representative examples of the 100 realizations were plotted.

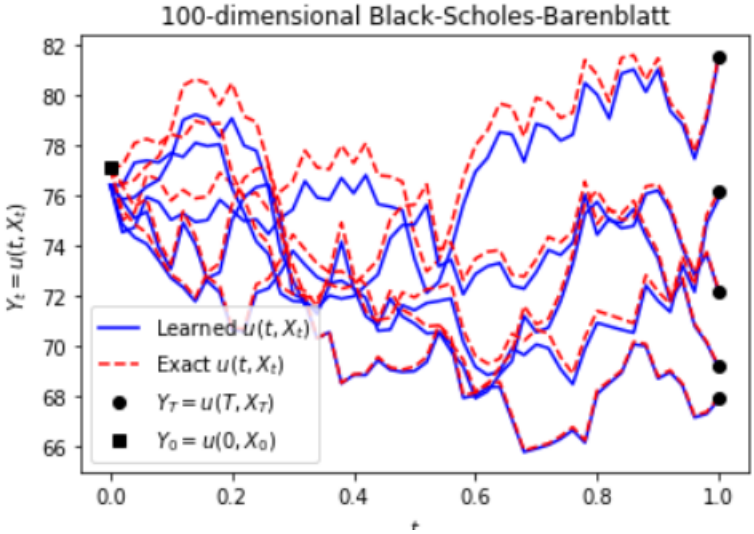

Figure 1: Black-Scholes-Barenblatt Equation in 100D: Evaluations of the learned solution Yt = u(t, Xt) at representative realizations of the underlying high-dimensional process Xt.

To further investigate the performance of our algorithm, in figure 2 we report the mean and mean plus two standard deviations of the relative errors between model predictions and the exact solution computed based on 100 independent realizations of the underlying Brownian motion.

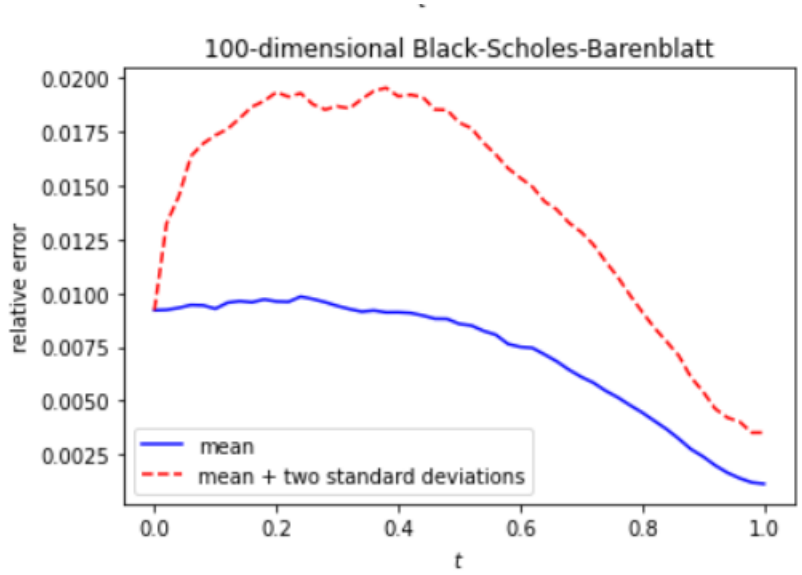

Figure 2: Black-Scholes-Barenblatt Equation in 100D: Mean and mean plus two standard deviations of the relative errors between model predictions and the exact solution computed based on 100 realizations of the underlying Brownian motion.

The results are obtained after 2×10^3, 3×10^3, 3×10^3, and 2×10^3 consecutive iterations of the Adam optimizer with learning rates of 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}, respectively. The total number of iterations is therefore given by 10^4. Every 10 iterations of the optimizer take about 6.70 seconds on our own computer. In each iteration of the Adam optimizer we are using 100 different realizations of the underlying Brownian motion. The original code does 100,000 iterations, here for time's sake we showed the result of 10,000 iterations, but if we also finish 100,000 iterations, the total number of Brownian motion trajectories observed by the algorithm is given by 10^7.

It is also worth highlighting that the algorithm converges to the exact value $Y_0 = u(0, X_0)$ in the first few hundred iterations of the Adam optimizer. For instance after only a few hundred steps of training, the algorithms achieve an accuracy level of 10^{-3} in terms of relative error. This is comparable to the results reported in some other research papers, both in terms of accuracy and the speed of the algorithm. However, to obtain more accurate estimates for $Y_t = u(t, X_t)$ at later times t > 0 we need to train the algorithm using more iterations of the Adam optimizer.

Reference

Raissi, Maziar. *Forward-Backward Stochastic Neural Networks.* (2018)
https://maziarraissi.github.io/FBSNNs/

Tensorflow API documents for Python
https://www.tensorflow.org/api_docs/python/tf/