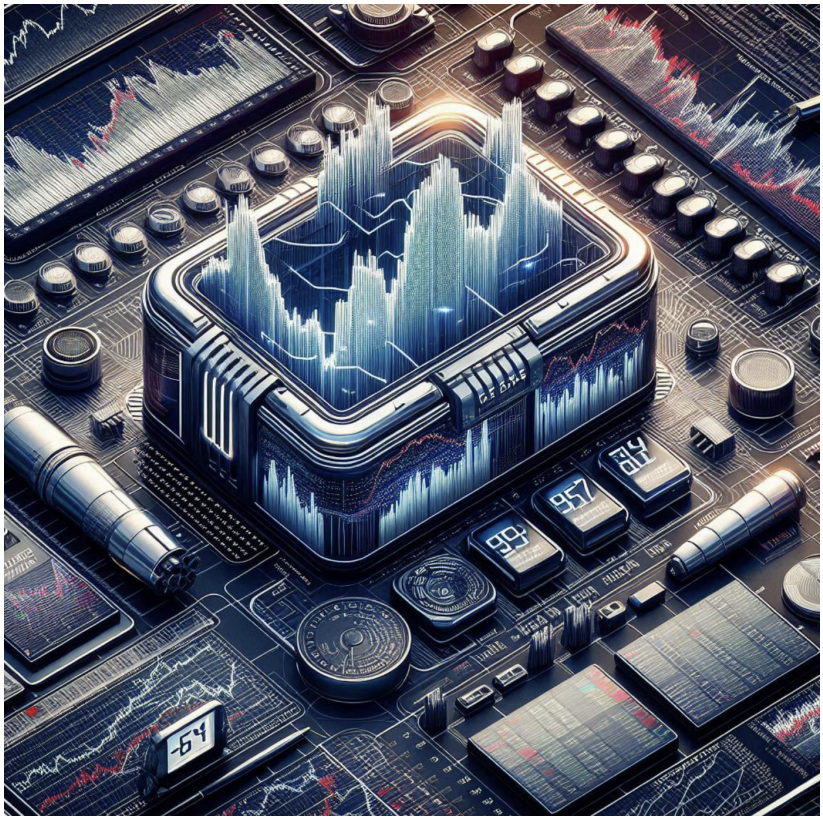
Project Report

Solving SOC Problems via SMP with Deep Learning Methods

Zakaria Magdoul

2024/2025



Contents

1 Introduction

The problem at hand, is a classic stochastic control one:
Problem I

$$\inf_{u(\cdot)\in\mathcal{U}_{ad}[0,T]} \mathbb{E}\left\{\int_0^T f(t,x_t,u_t)dt + h(x_T)\right\}, \tag{1}$$

$$\text{s.t. } x_t = x_0 + \int_0^t b(t,x_s,u_s)ds + \int_0^t \sigma(t,x_s,u_s)dW_s. \tag{2}$$

The control set is **convex** in what follows ($U \subset \mathbb{R}^k$ is convex).

1.1 Notation

The authors failed to be clear on some of the notation used throughout the paper – we present them clearly.

1.2 Setting the Dimensions

- d: dimension of Brownian motion $W : [0,T] \times \Omega \to \mathbb{R}^d$.
- n: dimension of X.
- k: dimension of control u.

Thus, naturally:

$$b : [0,T] \times \mathbb{R}^n \times U \to \mathbb{R}^n,$$
$$\sigma : [0,T] \times \mathbb{R}^n \times U \to \mathbb{R}^{n\times d}.$$

1.3 Setting the Sets

- $U \subset \mathbb{R}^k, \quad u : [0,T] \times \Omega \to U$.
- U_{ad}: set of admissible controls.
- $p \in L_F^2$ and q is a vector function valued in $\mathbb{R}^n$, also in L_F^2.
- $H : [0,T] \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{n\times d} \to \mathbb{R}$.

2 Problem Formulation

2.1 Preliminaries

Let $T > 0$ and $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ be a filtered probability space, where $W : [0,T] \times \Omega \to \mathbb{R}^d$ is a $d-$dimensional standard $\mathbb{F}$-Brownian motion on $(\Omega, \mathcal{F}, \mathbb{P}), \mathbb{F} = \{\mathcal{F}_t\}_{0\le t\le T}$ is the natural filtration generated by the Brownian motion W. Suppose that $(\Omega, \mathcal{F}, \mathbb{P})$ is complete, $\mathcal{F}_0$ contains all the $\mathbb{P}-$null sets in $\mathcal{F}$ and $\mathbb{F}$ is right continuous. Considering the following controlled stochastic differential equation:

$$\begin{cases} \mathrm{d}x_t = b\left(t,x_t,u_t\right)\mathrm{d}t + \sigma\left(t,x_t,u_t\right)\mathrm{d}W_t \\ x_0 = x_0 \in \mathbb{R}^n \end{cases}$$

where $(u_t)_{t\in[0,T]}$, is an admissible control process, i.e. a $\mathbb{F}$-adapted square-integrable process valued in a given subset U of $\mathbb{R}^k$. We define the distance $\|\cdot\|$ in an Euclidean space. b and σ are the drift coefficient and diffusion coefficient of last equation, respectively. They are deterministic functions

$$b : [0,T] \times \mathbb{R}^n \times U \to \mathbb{R}^n,$$
$$\sigma : [0,T] \times \mathbb{R}^n \times U \to \mathbb{R}^{n\times d}.$$

The cost functional is

$$J(u(\cdot)) = \mathbb{E}\left\{\int_0^T f\left(t, x_t, u_t\right) \mathrm{d}t + h\left(x_T\right)\right\}.$$

The set of all admissible controls is denoted by $\mathcal{U}_{ad}[0,T]$

$$\mathcal{U}_{ad}[0,T] \triangleq \left\{u : [0,T] \times \Omega \to U \mid u \in L^2_{\mathcal{F}}\left(0,T;\mathbb{R}^k\right)\right\}$$

where

$$L^2_{\mathcal{F}}\left(0,T;\mathbb{R}^k\right) \triangleq \left\{x : [0,T] \times \Omega \to \mathbb{R}^k \mid x \text{ is } \mathbb{F}\text{-adapted and } \mathbb{E}\left[\int_0^T |x_t|^2 \ \mathrm{d}t\right] < \infty\right\}$$

Our stochastic optimal control problem can be stated as minimizing last equation over $\mathcal{U}_{ad}[0,T]$. The goal is to find $u^*(\cdot) \in \mathcal{U}_{ad}[0,T]$ (if it exists) such that

$$J\left(u^*(\cdot)\right) = \inf_{u(\cdot)\in\mathcal{U}_{ad}[0,T]} \mathbb{E}\left\{\int_0^T f\left(t, x_t, u_t\right) \mathrm{d}t + h\left(x_T\right)\right\}.$$

Any $u^*(\cdot) \in \mathcal{U}_{ad}[0,T]$ satisfying the last equation is called an optimal control. The corresponding state process $x^*(\cdot)$ and the state-control pair $(x^*(\cdot), u^*(\cdot))$ are called an optimal state process and an optimal pair respectively.

2.1 Stochastic Maximum Principle

Based on the works of Hu [1] the Stochastic Maximum Principle states:

Let us define the following set of hypothesis (Assumptions):

- The functions b, σ, f and h are measurable functions.
- $x \to b(x,u)$, $x \to \sigma(x,u)$ and $x \to h(x)$ are $\mathcal{C}^2$.
- There exists a constant $K > 0$ such that:

$$\begin{cases} |\phi_x| + |\phi_x x| + |h_x| + |h_{xx}| \le K, \\ |\phi| + |h| \le K(1 + |x| + |u|). \end{cases}$$

Adjoint Equation
We introduce the following backward stochastic differential equations (BSDEs).

$$\begin{cases} \mathrm{d}p^*_t = -\left\{b_x\left(t, x^*_t, u^*_t\right)^{\mathrm{T}} p^*_t + \sum_{j=1}^d \sigma^j_x\left(t, x^*_t, u^*_t\right)^{\mathrm{T}} q^*_{jt} - f_x\left(t, x^*_t, u^*_t\right)\right\} \mathrm{d}t + q^*_t \ \mathrm{d}W_t, \\ p^*_T = -h_x\left(x^*_T\right), \quad t \in [0,T]. \end{cases}$$

We now define the Hamiltonian operator :

$$\begin{aligned} H(t,x,u,p,q) =& \langle p, b(t,x,u)\rangle + \mathrm{tr}\left[q^{\mathrm{T}}\sigma(t,x,u)\right] - f(t,x,u), \\ & (t,x,u,p,q) \in [0,T] \times \mathbb{R}^n \times U \times \mathbb{R}^n \times \mathbb{R}^{n\times d}, \end{aligned}$$

Fundamental Theorem of Stochastic Maximum Principle
Let Assumptions hold. Let $(x^*(\cdot), u^*(\cdot), p^*(\cdot), q^*(\cdot))$ be an admissible 4-tuple. Suppose further that h is convex and H concave and

$$H\left(t, x^*_t, u^*_t, p^*_t, q^*_t\right) = \max_{u\in U} H\left(t, x^*_t, u, p^*_t, q^*_t\right), \quad \text{a.e. } t \in [0,T], \quad \mathbb{P}\text{-a.s.}$$

hen $(x^*(\cdot), u^*(\cdot))$ is an optimal pair.

Formulation 1 of the problem: by combining the sufficient conditions, we find that a quadruple (x, u, p, q) is optimal iff:

$$\begin{cases} \mathrm{d}x_t^* = b\left(t, x_t^*, u_t^*\right)\mathrm{d}t + \sigma\left(t, x_t^*, u_t^*\right)\mathrm{d}W_t, \\ \mathrm{d}p_t^* = -H_x\left(t, x_t^*, u_t^*, p_t^*, q_t^*\right)\mathrm{d}t + q_t^*\ \mathrm{d}W_t, \quad t \in [0,T] \\ x_0^* = x_0, \quad p_T^* = -h_x\left(x_T^*\right) \\ H\left(t, x_t^*, u_t^*, p_t^*, q_t^*\right) = \max_{u\in U} H\left(t, x_t^*, u, p_t^*, q_t^*\right), \end{cases}$$

A key observation is that the max is taken over a scalar set U whereas in the initial miniziation problem, it was taken over the set of admissible controls. For future formulation, let's define a function $\bar{H}(t,x,p,q) = \max_{u\in U} H(t,x,u,p,q)$.

- The equation is reffered to as a FBSDE, coupled with a maximum principle. Indeed, x and u each follow a forward SDE and p and q follow a BSDE.
- Remark: the next conclusions, and in particular, the use of deep learning to solve this formulation can be generalized to the case where U is not convex, but this requires to add a second order adjoint equation in the previous application of the SMP.

Formulation 2 of the problem : An equivalent Forward form of Formulation 1 is

$$\begin{cases} \mathrm{d}\tilde{x}_t = b\left(t, \tilde{x}_t, \tilde{u}_t\right)\mathrm{d}t + \sigma\left(t, \tilde{x}_t, \tilde{u}_t\right)\mathrm{d}W_t, \\ \mathrm{d}\tilde{p}_t = -H_x\left(t, \tilde{x}_t, \tilde{u}_t, \tilde{p}_t, \tilde{q}_t\right)\mathrm{d}t + \tilde{q}_t\ \mathrm{d}W_t \\ \tilde{x}_0 = x_0, \quad \tilde{p}_0 = \tilde{p}_0 \\ H\left(t, \tilde{x}_t, \tilde{u}_t, \tilde{p}_t, \tilde{q}_t\right) = \max_{u\in U} H\left(t, \tilde{x}_t, u, \tilde{p}_t, \tilde{p}_t\right) \end{cases}$$

Here, both conditions on x and p are initial conditions.

3 Turning the Problem Into a Learning Problem

The equations from Formulation 2, are turned into a new variational problem **Problem II**. The new problem has a quadratic cost and is more structured than the initial variational problem.

$$\begin{aligned} &\inf_{\tilde{p}_0, \{\tilde{q}_t\}_{0\le t\le T}} \mathbb{E}\left[\left|-h_x\left(\tilde{x}_T\right) - \tilde{p}_T\right|^2\right] \\ \text{s.t. } \tilde{x}_t =& x_0 + \int_0^t b\left(t, \tilde{x}_s, \tilde{u}_s\right)\mathrm{d}s + \int_0^t \sigma\left(t, \tilde{x}_s, \tilde{u}_s\right)\mathrm{d}W_s, \\ \tilde{p}_t =& \tilde{p}_0 - \int_0^t H_x\left(s, \tilde{x}_s, \tilde{u}_s, \tilde{p}_s, \tilde{q}_s\right)\mathrm{d}s + \int_0^t \tilde{q}_s\ \mathrm{d}W_s, \\ \tilde{u}_t =& \arg\max_{u\in U} H\left(t, \tilde{x}_t, u, \tilde{p}_t, \tilde{q}_t\right). \end{aligned}$$

Fundamental Result:
Suppose the assumptions in Theorems 1 and 2 hold, if there exists a solution $(\tilde{x}_t, \tilde{u}_t, \tilde{p}_t, \tilde{q}_t)_{0\le t\le T}$ of Formulation 1 satisfying

$$\mathbb{E}\left[\left|-h_x\left(\tilde{x}_T\right) - \tilde{p}_T\right|^2\right] = 0$$

then $(\tilde{x}_t, \tilde{u}_t, \tilde{p}_t, \tilde{q}_t)_{0\le t\le T}$ is a solution of formulation 1, and the cost functional can be obtained by

$$J\left(u^*(\cdot)\right) = J(\tilde{u}(\cdot)) = \mathbb{E}\left\{\int_0^T f\left(t, \tilde{x}_t, \tilde{u}_t\right)\mathrm{d}t + h\left(\tilde{x}_T\right)\right\}$$

4 Introducing the three Algorithms

4.1 Algorithm 1: 1-NNet (Feed-Forward Neural Network)

4.1.1 Principle and Pseudo-Code

The first algorithm trains a neural network ϕ to calculate q, and does a gradient descent to find u which is a solution of an maximizing problem at every iteration.

Pseudo-code (the original paper omits an important outer for loop)

Algorithm 1 Numerical algorithm with 1-NNet

Require: Brownian motion ΔW_i, initial parameters (θ^0, x^0, p_0^0), learning rate η
Ensure: The 4-tuple processes $(x_i^t, u_i^t, p_i^t, q_i^t)$

for $t = 0$ to $maxstep$ **do**
  for $k = 0$ to M **do**
    $x_0^t = x_0$, $p_0^t = p_0^t$
    for $i = 0$ to $N-1$ **do**
      $q_i^t = \phi(t_i, x_i^t, p_i^t; \theta^t)$
      $u_i^t = \arg\max_{u \in U} H(t_i, x_i^t, p_i^t, q_i^t, u)$
      $x_{i+1}^t = x_i^t + b(t_i, x_i^t, u_i^t)\Delta t + \sigma(t_i, x_i^t, u_i^t)\Delta W_i$
      $p_{i+1}^t = p_i^t + f_x(t_i, x_i^t, u_i^t)\Delta t + q_i^t \Delta W_i$
    end for
  end for
  $J(\tilde{u}(\cdot)) = \frac{1}{M}\sum_{j=1}^{M}\left[\frac{T}{N}\sum_{i=0}^{N-1} f(t_i, x_i^t, u_i^t) + h(x_T^t)\right]$
  $\text{loss} = \frac{1}{M}\sum_{j=1}^{M}\left[h_x(x_T^t) - p_T^t\right]^2$
  $(\theta^{t+1}, p_0^{t+1}) = (\theta^t, p_0^t) - \eta\nabla\text{loss}$
end for

- Euler scheme for forward SDE.
- Neural network estimates q_t^t.
- Compute u_i^t using gradient descent (omitted from the algorithm).
- Optimize over trajectories using back-propagation.

4.1.2 Pros and Cons of Algorithm 1

Pros:

One neural network to train, very intuitive approach, the loss function is directly related to that of **Problem II**, doesn't require knowing any closed formulas for u.

Cons:

Even before running this procedure, we can intuit that the algorithm will be very slow, because of the gradient descent which is needed at every time step of every training step. Higher dimensions would obviously make matters worse, or even impossible, because this maximizing becomes harder as the dimension increases. Additionally, the algorithms numerically approximates two variables, u and q, means there are two independent error sources, we therefore also expect a high error.

4.2 Algorithm 2: 2-NNet (Feed-Forward Neural Network)

4.2.1 Principle and Pseudocode

The motivation here is to solve the control problem without having a closed formula for u. It turns out that under regularity hypotheses, the optimality condition for the quadruplet can be reduced to

$$\begin{cases} \mathrm{d}x_t^* = b\left(t, x_t^*, u_t^*\right)\mathrm{d}t + \sigma\left(t, x_t^*, u_t^*\right)\mathrm{d}W_t, \\ \mathrm{d}p_t^* = -H_x\left(t, x_t^*, u_t^*, p_t^*, q_t^*\right)\mathrm{d}t + q_t^*\ \mathrm{d}W_t, \quad t \in [0, T], \\ x_0^* = x_0, \quad p_T^* = -h_x\left(x_T^*\right), \\ H_u\left(t, x_t^*, u_t^*, p_t^*, q_t^*\right) = 0, \quad \forall u \in U \end{cases} \tag{3.6}$$

Once again, we need turn this into a learnable problem. One way is to balance the need to cancel $h_x(x_T) + p_T$, and cancel $H_u(t, x_t^*, u_t^*, p_t^*, q_t^*)$, this balance will be quantified by a new hyperparameter

λ, we get to solve **Problem III**.

$$\begin{aligned}
\text{loss} =& \frac{1}{M}\sum_{j=1}^{M}\left[\left|-h_x\left(\tilde{x}_T^{\pi}\right)-\tilde{p}_T^{\pi}\right|^2+\lambda\sum_{i=0}^{N-1}H_u\left(t,\tilde{x}_{t_i}^{\pi},\tilde{u}_{t_i}^{\pi},\tilde{x}_{p_i}^{\pi},\tilde{q}_{t_i}^{\pi}\right)^2\right],\\
\text{s.t. } \tilde{x}_t =& x_0+\int_0^t b\left(t,\tilde{x}_s,\tilde{u}_s\right)\mathrm{d}s+\int_0^t \sigma\left(t,\tilde{x}_s,\tilde{u}_s\right)\mathrm{d}W_s,\\
\tilde{p}_t =& \tilde{p}_0-\int_0^t H_x\left(s,\tilde{x}_s,\tilde{u}_s,\tilde{p}_s,\tilde{q}_s\right)\mathrm{d}s+\int_0^t \tilde{q}_s\ \mathrm{d}W_s,\\
\tilde{u}_t =& \arg\max_{u\in U} H\left(t,\tilde{x}_t,u,\tilde{p}_t,\tilde{q}_t\right).
\end{aligned}$$

Where we will learn q and u, each with a neural network (with weights $\theta^{q,0}$ and $\theta^{u,0}$), and these networks are independent of p.

Pseudo-code (the original paper omits an important outer for loop)

Algorithm 2 Numerical algorithm with 2-NNet

Require: Brownian motion ΔW_i, initial parameters $(\theta^{q,0}, \theta^{u,0}, x^0, p_0^0)$, learning rate η, and hyperparameter λ

Ensure: The 4-tuple processes $(x_i^t, u_i^t, p_i^t, q_i^t)$

for $t = 0$ to $maxstep$ **do**
  for $k = 0$ to M **do**
    $x_0^t = x_0$, $p_0^t = p_0^t$, and $H = 0$
    for $i = 0$ to $N-1$ **do**
      $\tilde{q}_i^t = \phi^1(t_i, \tilde{x}_i^t, ; \theta^{q,t})$
      $\tilde{u}_i^t = \phi^2(t_i, \tilde{x}_i^t; \theta^{u,t})$
      $\tilde{x}_{i+1}^t = \tilde{x}_i^t + b(t_i, \tilde{x}_i^t, \tilde{u}_i^t)\Delta t + \sigma(t_i, \tilde{x}_i^t, \tilde{u}_i^t)\Delta W_i$
      $p_{i+1}^t = p_i^t - H_x\left(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{u}_{t_i}^{l,\pi}, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi}\right)\Delta t_i + \tilde{q}_{t_i}^{l,\pi}\Delta W_i$
      $H = H + H_u\left(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{u}_{t_i}^{l,\pi}, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi}\right)^2$
    end for
  end for
  $J(\tilde{u}_{t_i}^{l,\pi}(\cdot)) = \frac{1}{M}\sum_{j=1}^{M}\left[\frac{T}{N}\sum_{i=0}^{N-1} f(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{u}_{t_i}^{l,\pi}) + h(\tilde{x}_T^{l,\pi})\right]$
  $\text{loss} = \frac{1}{M}\sum_{j=1}^{M}\left[\left[-h_x(\tilde{x}_T^{l,\pi}) - p_T^t\right]^2 + \lambda H\right]$
  $(\theta^{q,t+1}, \theta^{u,t+1}, p_0^{t+1}) = (\theta^{q,t}, \theta^{u,t}, p_0^t) - \eta\nabla\text{loss}$
end for

4.2.2 Pros and Cons

Pros:

Since u is the result of the prediction by a neural network instead of through gradient descent, Algorithm 2 can be scaled, and used for a wide range of high dimensional problems, without requiring optimal u to be solved. This is useful when optimal u cannot be found explicitly.

Cons:

The method introduces a new hyperparameter λ, needing fine-tuning. Additionally, the problem is ill-defined, since the gradient descent is done using a loss that does not involve the actual optimal u – we don't calculate it here – instead, the procedure uses a 'proxy loss'. Furthermore, as the new neural network introduces a new set of weights and a new source of variance, it can be expected to be sub-perfect.

4.3 Algorithm 3: Numerical Algorithm with explicit expression of $\bar{H}$

4.3.1 Principle and Pseudo-code

The key idea here is that in previous 2 algorithms, we only estimate u at every update of the other elements x, p, q because they are dependent on it. However, this dependence is through the partial

derivatives of the Hamiltonian, so if we knew the closed formula of $\bar{H}$, we could bypass the calculations of u at every update, and do it after optimal x, p and q are found. Given this observation, we return to the FBSDE formulation with constraint (aka **Formulation 1**, where we replace b and σ by the partial derivatives of the Hamiltonian, but instead of putting the Hamiltonian evaluated in u^*. We get :

$$\begin{cases} \mathrm{d}x_t^* = \bar{H}_p\left(t, x_t^*, p_t^*, q_t^*\right)\mathrm{d}t + \bar{H}_q\left(t, x_t^*, p_t^*, q_t^*\right)\mathrm{d}W_t, \\ \mathrm{d}p_t^* = -\bar{H}_x\left(t, x_t^*, p_t^*, q_t^*\right)\mathrm{d}t + q_t^*\ \mathrm{d}W_t, \quad t \in [0,T] \\ x_0^* = x_0, \quad p_T^* = -h_x\left(x_T^*\right) \end{cases}$$

So we only need a neural network for q, and u will not be involved in solving the problem.

Algorithm 3 Numerical algorithm with explicit expression of $\bar{H}$

Require: Brownian motion ΔW_i, initial parameters $(\theta^{q,0}, x^0, p_0^0)$, learning rate η
Ensure: The Triple processes (x_i^t, p_i^t, q_i^t)

for $t = 0$ to $maxstep$ **do**
  for $k = 0$ to M **do**
    $x_0^t = x_0$, $p_0^t = p_0^t$
    for $i = 0$ to $N-1$ **do**
      $\tilde{q}_i^t = \phi^1(t_i, \tilde{x}_i^t, \tilde{p}_{t_i}^{l,\pi}; \theta^t)$
      $\tilde{x}_{i+1}^t = \tilde{x}_i^t + \bar{H}_p\left(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi}\right)\Delta t_i + \bar{H}_q\left(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi}\right)\Delta W_i$
      $p_{i+1}^t = p_i^t - \bar{H}_x\left(t_i, \tilde{x}_{t_i}^{l,\pi}, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi}\right)\Delta t_i + \tilde{q}_{t_i}^{l,\pi}\Delta W_i$
    end for
  end for
  loss $= \frac{1}{M}\sum_{j=1}^{M}\left[[-h_x(x_T^t) - p_T^t]^2\right]$
  $(\theta^{t+1}, p_0^{t+1}) = (\theta^t, p_0^t) - \eta\nabla\text{loss}$
end for
$\tilde{u}_{t_i}^{l,\pi} = \arg\max_{u\in U} H(t_i, \tilde{x}_{t_i}^{l,\pi}, u, \tilde{p}_{t_i}^{l,\pi}, \tilde{q}_{t_i}^{l,\pi})$
$J(\tilde{u}_{t_i}^{l,\pi,t}(\cdot)) = \frac{1}{M}\sum_{j=1}^{M}\left[\frac{T}{N}\sum_{i=0}^{N-1} f(t_i, x_i^t, u_i^t) + h(x_T^t)\right]$

4.3.2 Pros and Cons

Pros

Avoid the need of controlling u at every iteration, only solving the maximizing equation for u at the end, which means it runs very quickly.

Cons

Requires knowing the analytical formula of $\bar{H}$ and its partial derivatives.

5 Results

We run the three algorithms on three examples.

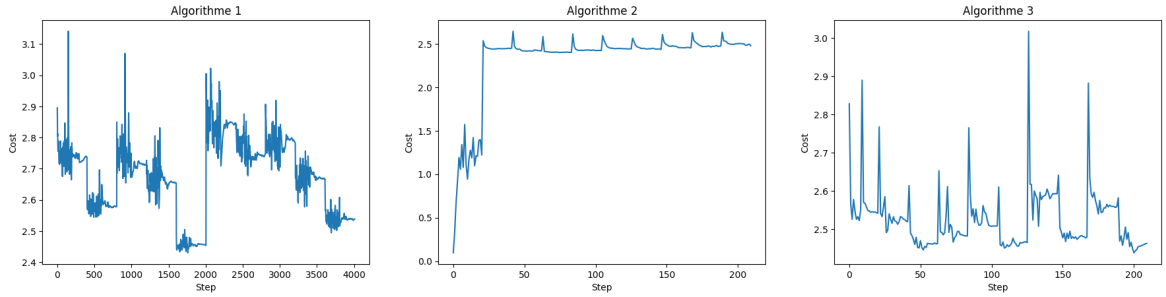
5.1 Example One : Low-Dimension LQ Problem

$$\begin{cases} \mathrm{d}x_t = \left(-\frac{1}{4}x_t + u_t\right)\mathrm{d}t + \left(\frac{1}{5}x_t + u_t\right)\mathrm{d}W_t \\ x(0) = x_0 \end{cases}$$

and the cost functional is defined as

$$J\left(0, x_0; u(\cdot)\right) = \mathbb{E}\left\{\frac{1}{2}\int_0^T \left[\left\langle\frac{1}{2}x_t, x_t\right\rangle + \langle 2u_t, u_t\rangle\right]\mathrm{d}t + \frac{1}{2}\langle Qx_T, x_T\rangle\right\}$$

We are more interested in the performance of the algorithms rather than the solution itself. We therefore plot the evolution of the cost function across 10 subsequent training runs, (Implementation is in the Annex). **Remarks:** The computations for this example were conducted on an Apple MacBook powered by an Apple M1 (ARM-based) processor with 8 GB of unified memory.
The relative errors are always computed with respect to Matlab ODE45 module results.



(a) Algorithm 1 (b) Algorithm 2 (c) Algorithm 3

Figure 1: Plots of the evolution of the cost function for the three algorithm on Example One

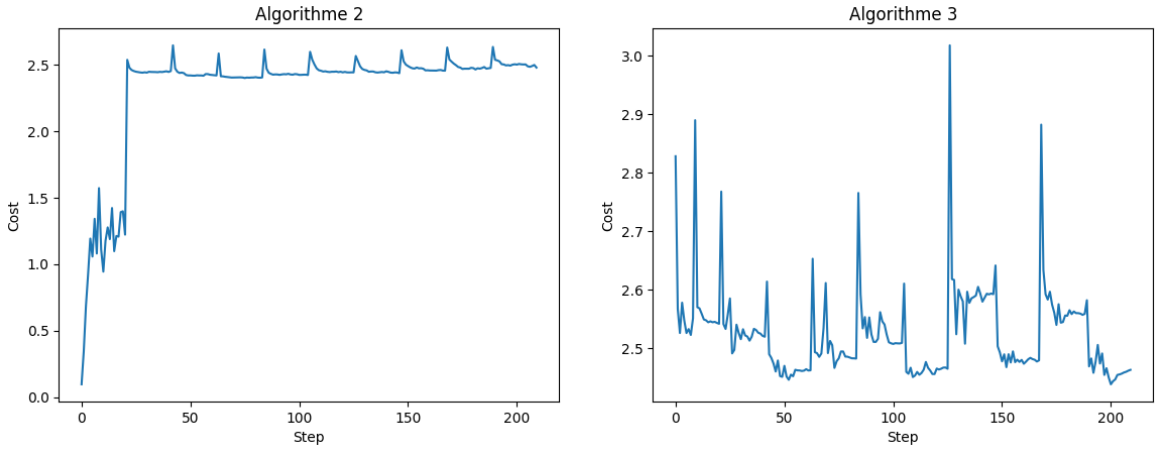
Results :

Method	p_0	Cost	Time(s)	Iteration step	Relative error
Alg 1	-0.95734	2.4008	41,794.2	2000	0.13%
Alg 2	-0.95775	2.3900	340.0	2000	0.09%
Alg 3	-0.95863	2.3880	180.5	2000	0.003%

Table 1: Comparison of our different algorithms for $n = 5$

5.2 Example Two : High-Dimensional LQ Problem

We take the same problem, only now $n = 100$ *Achtung!* Algorithm 1 is completely unuseable in this case as it would take infinite time to run.



(a) Algorithm 2 (b) Algorithm 3

Figure 2: Plots of the evolution of the cost function for the three algorithm on Example One

Running our tests for n going from 1 to 20, we get the following table.

		$n = 1$	$n = 2$	$n = 5$	$n = 10$	$n = 20$
Solution with ODE45		-0.9586	-1.8275	-4.3638	-8.5306	-16.821
Solution with Neural Network	Alg 2	-0.9519	-1.8239	-4.3535	-8.4974	-16.730
	Alg 3	-0.9585	-1.8276	-4.3571	-8.4782	-16.663
Relative error	Alg 2	0.705%	0.197%	0.234%	0.390%	0.541%
	Alg 3	0.007%	0.003%	0.153%	0.615%	0.939%
Cost functional	Alg 2	0.4812	1.8375	10.863	42.850	169.80
	Alg 3	0.4892	1.8262	10.916	43.284	167.64

Table 2: Comparison between ODE45 and our algorithms for different dimensions

6 Conclusion

To summearize the conclusions of this paper, it offers 3 algorithms to solve the SOC problem: If a closed-form solution for u^* exists, use Algorithm 3. For high dimensions, use Algorithm 2. Algorithm 1 is generally suboptimal. In this paper, we have solved the stochastic optimal control problem by reformulating it, in view of the stochastic maximum principle. This leads to three different algorithms calculating solutions by leveraging deep learning. We have compared the proposed algorithms through numerical results and pointed out their applicative benefits. The numerical results for different examples demonstrate the effectiveness of our proposed algorithms, depending on the setting.

7 Appendix

Listing 1: Algorithm 1 Implementation

```
import time
import numpy as np
import tensorflow as tf
import math
from scipy.stats import multivariate_normal as normal
import matplotlib.pyplot as plt
from scipy.optimize import minimize

tf.keras.backend.set_floatx('float64')

class Solver(object):
    def __init__(self,):
        self.valid_size = 512
        self.batch_size = 64
        self.num_iterations = 2000
        self.logging_frequency = 5
        self.lr_values = [5e-2, 5e-3, 1e-3]

        self.lr_boundaries = [1000, 1500]
        self.config = Config()

        self.model = WholeNet()
        self.y_init = self.model.y_init
        lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(self.lr_bou
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule, epsilon=1e

    def train(self):
        """Training the model"""
        start_time = time.time()
        training_history = []
        dW = self.config.sample(self.valid_size)
        valid_data = dW
```

```
        for step in range(self.num_iterations+1):
            if step % self.logging_frequency == 0:
                loss, cost = self.model(valid_data, training=True)
                y_init = self.y_init.numpy()[0][0]
                elapsed_time = time.time() - start_time
                training_history.append([step, cost, y_init, loss])
                print("step: %5u, loss: %.4e, Y0: %.4e, cost: %.4e, elapsed time: %3u"
            self.train_step(self.config.sample(self.batch_size))
#           print('Y0_true: %.4e' % y_init)
        self.training_history = training_history

    def train_step(self, train_data):
        """Updating the gradients"""
        with tf.GradientTape(persistent=True) as tape:
            loss, cost = self.model(train_data, training = True)
        grad = tape.gradient(loss, self.model.trainable_variables)
        del tape
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_variables))

class WholeNet(tf.keras.Model):
    """Building the neural network architecture"""
    def __init__(self):
        super(WholeNet, self).__init__()
        self.config = Config()
        self.y_init = tf.Variable(tf.random.normal([1, self.config.dim_y], 0.0, 1.0, d
        self.z_net = FNNet()

    def call(self, dw, training):
        x_init = tf.ones([1, self.config.dim_x], dtype=tf.dtypes.float64) * 1.0
        time_stamp = np.arange(0, self.config.num_time_interval) * self.config.delta_t
        all_one_vec = tf.ones([tf.shape(dw)[0], 1], dtype=tf.dtypes.float64)
        x = tf.matmul(all_one_vec, x_init)
        y = tf.matmul(all_one_vec, self.y_init)
        l = 0.0 # The cost functional
        for t in range(0, self.config.num_time_interval):
            data = time_stamp[t], x, y
            z = self.z_net(data, training=True)
            u = self.config.u_fn_np(time_stamp[t], x.numpy(), y.numpy(), z.numpy())

            l = l + self.config.f_fn(time_stamp[t], x, u) * self.config.delta_t

            b_ = self.config.b_fn(time_stamp[t], x, u)
            sigma_ = self.config.sigma_fn(time_stamp[t], x, u)
            f_ = self.config.Hx_fn(time_stamp[t], x, u, y, z)

            x = x + b_ * self.config.delta_t + sigma_ * dw[:, :, t]
            y = y - f_ * self.config.delta_t + z * dw[:, :, t]

        delta = y + self.config.hx_tf(self.config.total_T, x)
        loss = tf.reduce_mean(tf.reduce_sum(delta**2, 1, keepdims=True))

        l = l + self.config.h_fn(self.config.total_T, x)
        cost = tf.reduce_mean(l)

        return loss, cost

class FNNet(tf.keras.Model):
    """ Define the feedforward neural network """
```

```
    def __init__(self):
        super(FNNet, self).__init__()
        self.config = Config()
        num_hiddens = [self.config.dim_x*2+10, self.config.dim_x*2+10, self.config.dim
        self.bn_layers = [
            tf.keras.layers.BatchNormalization(
                momentum=0.99,
                epsilon=1e-6,
                beta_initializer=tf.random_normal_initializer(0.0, stddev=0.1),
                gamma_initializer=tf.random_uniform_initializer(0.1, 0.5)
            )
            for _ in range(len(num_hiddens) + 2)]
        self.dense_layers = [tf.keras.layers.Dense(num_hiddens[i],
                                                   use_bias=False,
                                                   activation=None)
                             for i in range(len(num_hiddens))]
        # final output should be gradient of size dim_z
        self.dense_layers.append(tf.keras.layers.Dense(self.config.dim_z, activation=N

    def call(self, inputs, training):
        """structure: bn -> (dense -> bn -> relu) * len(num_hiddens) -> dense -> bn"""
        t, x, y = inputs
        ts = tf.ones([tf.shape(x)[0], 1], dtype=tf.dtypes.float64) * t
        x = tf.concat([ts, x, y], axis=1)
        x = self.bn_layers[0](x, training=True)
        for i in range(len(self.dense_layers) - 1):
            x = self.dense_layers[i](x)
            x = self.bn_layers[i+1](x, training=True)
            x = tf.nn.relu(x)
        x = self.dense_layers[-1](x)
        x = self.bn_layers[-1](x, training=True)
        return x

class Config(object):
    """Define the configs in the systems"""
    def __init__(self):
        super(Config, self).__init__()
        self.dim_x = 5
        self.dim_y = 5
        self.dim_z = 5
        self.dim_u = 5
        self.num_time_interval = 20
        self.total_T = 0.1
        self.delta_t = (self.total_T + 0.0) / self.num_time_interval
        self.sqrth = np.sqrt(self.delta_t)
        self.t_stamp = np.arange(0, self.num_time_interval) * self.delta_t

    def sample(self, num_sample):
        dw_sample = normal.rvs(size=[num_sample, self.num_time_interval]) * self.sqrth
        return dw_sample[:, np.newaxis, :]

    # define the f function: f_fn
    def f_fn(self, t, x, u):
        #define f

    def h_fn(self, t, x):
        #here we define function h
```

```
    def b_fn(self, t, x, u):
        #here we define function b

    def sigma_fn(self, t, x, u):
        #here we define function sigma

    def Hx_fn(self, t, x, u, y, z):
        #here we define function Hx

    def hx_tf(self, t, x):
        #here we define function hx

    # define the Hamiltonian - numpy form
    def H_fn(self, t, x, u, y, z):
        #here we define function H_fn

    # define the numpy function
    def u_fn_np(self, t, x, y, z):


def main():
    print('Training time 1:')
    solver = Solver()
    solver.train()
    k = 10
    data = np.array(solver.training_history)
    output = np.zeros((len(data[:, 0]), 3 + k))
    output[:, 0] = data[:, 0] # step
    output[:, 1] = data[:, 2] # loss
    output[:, 2] = data[:, 3] # y_init
    output[:, 3] = data[:, 1] # cost

    a = ['%d', '%.5e', '%.5e']
    for i in range(k):
        a.append('%.5e')

    np.savetxt('./LQ_data_Scipy_d5.csv', output, fmt=a, delimiter=',')

    for i in range(k - 1):
        print('Training time %3u:' % (i + 2))
        solver = Solver()
        solver.train()
        data = np.array(solver.training_history)
        output[:, 4 + i] = data[:, 1]
        np.savetxt('./LQ_data_Scipy_d5.csv', output, fmt=a, delimiter=',')

    print('Solving is done!')
```

Listing 2: Algorithm 2 Implementation

```
import time
import numpy as np
import tensorflow as tf
import math
from scipy.stats import multivariate_normal as normal
import matplotlib.pyplot as plt
from scipy.optimize import minimize

tf.keras.backend.set_floatx('float64')

class Solver(object):
    def __init__(self,):
        self.valid_size = 512
        self.batch_size = 64
        self.num_iterations = 2000
        self.logging_frequency = 5
        self.lr_values = [5e-2, 5e-3, 1e-3]

        self.lr_boundaries = [1000, 1500]
        self.config = Config()

        self.model = WholeNet()
        self.y_init = self.model.y_init
        lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(self.lr_bou
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule, epsilon=1e

    def train(self):
        """Training the model"""
        start_time = time.time()
        training_history = []
        dW = self.config.sample(self.valid_size)
        valid_data = dW
        for step in range(self.num_iterations+1):
            if step % self.logging_frequency == 0:
                loss, cost = self.model(valid_data, training=True)
                y_init = self.y_init.numpy()[0][0]
                elapsed_time = time.time() - start_time
                training_history.append([step, cost, y_init, loss])
                print("step: %5u, loss: %.4e, Y0: %.4e, cost: %.4e, elapsed time: %3u"
            self.train_step(self.config.sample(self.batch_size))
#           print('Y0_true: %.4e' % y_init)
        self.training_history = training_history

    def train_step(self, train_data):
        """Updating the gradients"""
        with tf.GradientTape(persistent=True) as tape:
            loss, cost = self.model(train_data, training = True)
        grad = tape.gradient(loss, self.model.trainable_variables)
        del tape
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_variables))

class WholeNet(tf.keras.Model):
    """Building the neural network architecture"""
    def __init__(self):
        super(WholeNet, self).__init__()
        self.config = Config()
        self.y_init = tf.Variable(tf.random.normal([1, self.config.dim_y], 0.0, 1.0, d
```

```
        self.z_net = FNNet()

    def call(self, dw, training):
        x_init = tf.ones([1, self.config.dim_x], dtype=tf.dtypes.float64) * 1.0
        time_stamp = np.arange(0, self.config.num_time_interval) * self.config.delta_t
        all_one_vec = tf.ones([tf.shape(dw)[0], 1], dtype=tf.dtypes.float64)
        x = tf.matmul(all_one_vec, x_init)
        y = tf.matmul(all_one_vec, self.y_init)
        l = 0.0 # The cost functional
        for t in range(0, self.config.num_time_interval):
            data = time_stamp[t], x, y
            z = self.z_net(data, training=True)
            u = self.config.u_fn_np(time_stamp[t], x.numpy(), y.numpy(), z.numpy())

            l = l + self.config.f_fn(time_stamp[t], x, u) * self.config.delta_t

            b_ = self.config.b_fn(time_stamp[t], x, u)
            sigma_ = self.config.sigma_fn(time_stamp[t], x, u)
            f_ = self.config.Hx_fn(time_stamp[t], x, u, y, z)

            x = x + b_ * self.config.delta_t + sigma_ * dw[:, :, t]
            y = y - f_ * self.config.delta_t + z * dw[:, :, t]

        delta = y + self.config.hx_tf(self.config.total_T, x)
        loss = tf.reduce_mean(tf.reduce_sum(delta**2, 1, keepdims=True))

        l = l + self.config.h_fn(self.config.total_T, x)
        cost = tf.reduce_mean(l)

        return loss, cost

class FNNet(tf.keras.Model):
    """ Define the feedforward neural network """
    def __init__(self):
        super(FNNet, self).__init__()
        self.config = Config()
        num_hiddens = [self.config.dim_x*2+10, self.config.dim_x*2+10, self.config.dim
        self.bn_layers = [
            tf.keras.layers.BatchNormalization(
                momentum=0.99,
                epsilon=1e-6,
                beta_initializer=tf.random_normal_initializer(0.0, stddev=0.1),
                gamma_initializer=tf.random_uniform_initializer(0.1, 0.5)
            )
            for _ in range(len(num_hiddens) + 2)]
        self.dense_layers = [tf.keras.layers.Dense(num_hiddens[i],
                                                   use_bias=False,
                                                   activation=None)
                             for i in range(len(num_hiddens))]
        # final output should be gradient of size dim_z
        self.dense_layers.append(tf.keras.layers.Dense(self.config.dim_z, activation=N

    def call(self, inputs, training):
        """structure: bn -> (dense -> bn -> relu) * len(num_hiddens) -> dense -> bn"""
        t, x, y = inputs
        ts = tf.ones([tf.shape(x)[0], 1], dtype=tf.dtypes.float64) * t
        x = tf.concat([ts, x, y], axis=1)
        x = self.bn_layers[0](x, training=True)
```

```
        for i in range(len(self.dense_layers) - 1):
            x = self.dense_layers[i](x)
            x = self.bn_layers[i+1](x, training=True)
            x = tf.nn.relu(x)
        x = self.dense_layers[-1](x)
        x = self.bn_layers[-1](x, training=True)
        return x

class Config(object):
    """Define the configs in the systems"""
    def __init__(self):
        super(Config, self).__init__()
        self.dim_x = 5
        self.dim_y = 5
        self.dim_z = 5
        self.dim_u = 5
        self.num_time_interval = 20
        self.total_T = 0.1
        self.delta_t = (self.total_T + 0.0) / self.num_time_interval
        self.sqrth = np.sqrt(self.delta_t)
        self.t_stamp = np.arange(0, self.num_time_interval) * self.delta_t

    def sample(self, num_sample):
        dw_sample = normal.rvs(size=[num_sample, self.num_time_interval]) * self.sqrth
        return dw_sample[:, np.newaxis, :]

    # define the f function: f_fn
    def f_fn(self, t, x, u):
        return 0.25 * tf.reduce_sum(x ** 2, 1, keepdims=True) + tf.reduce_sum(u ** 2,

    def h_fn(self, t, x):
        return 0.5 * tf.reduce_sum(x ** 2, 1, keepdims=True)

    def b_fn(self, t, x, u):
        return -0.25 * x + u

    def sigma_fn(self, t, x, u):
        return 0.2 * x + u

    def Hx_fn(self, t, x, u, y, z):
        return -0.5 * x - 0.25 * y + 0.2 * z

    def hx_tf(self, t, x):
        return x

    # define the Hamiltonian - numpy form
    def H_fn(self, t, x, u, y, z):
        return np.sum((-0.25 * x + u) * y + (0.2 * x + u) * z - 0.25 * x ** 2 - u ** 2

    # define the numpy function
    def u_fn_np(self, t, x, y, z):
        U = np.random.randn(x.shape[0], self.dim_u)
        def H_fun(u):
            u = u.reshape(x.shape[0], self.dim_u)
            h = np.mean((-0.25 * x + u) * y + (0.2 * x + u) * z - 0.25 * x ** 2 - u **
            return -h
        x_opt = minimize(H_fun, U.reshape(-1), method='L-BFGS-B').x
        return tf.constant(x_opt.reshape(x.shape[0], self.dim_u))
```

```
def main():
    print('Training time 1:')
    solver = Solver()
    solver.train()
    k = 10
    data = np.array(solver.training_history)
    output = np.zeros((len(data[:, 0]), 3 + k))
    output[:, 0] = data[:, 0] # step
    output[:, 1] = data[:, 2] # loss
    output[:, 2] = data[:, 3] # y_init
    output[:, 3] = data[:, 1] # cost

    a = ['%d', '%.5e', '%.5e']
    for i in range(k):
        a.append('%.5e')

    np.savetxt('./LQ_data_Scipy_d5.csv', output, fmt=a, delimiter=',')

    for i in range(k - 1):
        print('Training time %3u:' % (i + 2))
        solver = Solver()
        solver.train()
        data = np.array(solver.training_history)
        output[:, 4 + i] = data[:, 1]
        np.savetxt('./LQ_data_Scipy_d5.csv', output, fmt=a, delimiter=',')

    print('Solving is done!')
```

Listing 3: Algorithm 1 Implementation

```
\begin{thebibliography}{9}
    \bibitem{hu2014stochastic} Hu, Y. (2014). Stochastic Maximum Principle. \textit{En
    \bibitem{Paper} Shaolin Ji, Shige Peng, Ying Peng, Xichuan Zhang (2020) Solving sto
import time
import numpy as np
import tensorflow as tf
import math
from scipy.stats import multivariate_normal as normal
import matplotlib.pyplot as plt

tf.keras.backend.set_floatx('float64')

class Solver(object):
    def __init__(self,):
        self.valid_size = 512
        self.batch_size = 64
        self.num_iterations = 2000
        self.logging_frequency = 100
        self.lr_values = [5e-2, 5e-3, 1e-3]

        self.lr_boundaries = [1000, 1500]
        self.config = Config()

        self.model = WholeNet()
        self.y_init = self.model.y_init

        lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(self.lr_boun
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule, epsilon=1e

    def train(self):
        """Training the model"""
        start_time = time.time()
        training_history = []
        dW = self.config.sample(self.valid_size)
        valid_data = dW
        for step in range(self.num_iterations+1):
            if step % self.logging_frequency == 0:
                loss, cost = self.model(valid_data, training=True)
                y_init = self.y_init.numpy()[0][0]
                elapsed_time = time.time() - start_time
                training_history.append([step, cost, y_init, loss])
                print("step: %5u, loss: %.4e, Y0: %.4e, cost: %.4e, elapsed time: %3u"
            self.train_step(self.config.sample(self.batch_size))
        print('Y0_true: %.4e' % y_init)
        self.training_history = training_history

    @tf.function
    def train_step(self, train_data):
        """Updating the gradients"""
        with tf.GradientTape(persistent=True) as tape:
            loss, cost = self.model(train_data, training = True)
        grad = tape.gradient(loss, self.model.trainable_variables)
        del tape
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_variables))


class WholeNet(tf.keras.Model):
```

```
    """Building the neural network architecture"""
    def __init__(self):
        super(WholeNet, self).__init__()
        self.config = Config()
        self.y_init = tf.Variable(tf.random.normal([1, self.config.dim_y], mean=0, stdd
        self.z_net = FNNet()

    def call(self, dw, training):
        x_init = tf.ones([1, self.config.dim_x], dtype=tf.dtypes.float64) * 1.0
        time_stamp = np.arange(0, self.config.num_time_interval) * self.config.delta_t
        all_one_vec = tf.ones([tf.shape(dw)[0], 1], dtype=tf.dtypes.float64)
        x = tf.matmul(all_one_vec, x_init)
        y = tf.matmul(all_one_vec, self.y_init)
        l = 0.0 # The cost functional
        for t in range(0, self.config.num_time_interval):
            data = time_stamp[t], x, y
            z = self.z_net(data, training= True)
            u = self.config.u_fn(time_stamp[t], x, y, z)

            l = l + self.config.f_fn(time_stamp[t], x, u) * self.config.delta_t

            b_ = self.config.b_fn(time_stamp[t], x, y, z)
            sigma_ = self.config.sigma_fn(time_stamp[t], x, y, z)
            f_ = self.config.Hx_fn(time_stamp[t], x, y, z)

            x = x + b_ * self.config.delta_t + sigma_ * dw[:, :, t]
            y = y - f_ * self.config.delta_t + z * dw[:, :, t]

        delta = y + self.config.hx_tf(self.config.total_T, x)
        loss = tf.reduce_mean(tf.reduce_sum(delta**2, 1, keepdims=True))

        l = l + self.config.h_fn(self.config.total_T, x)
        cost = tf.reduce_mean(l)

        return loss, cost

class FNNet(tf.keras.Model):
    """ Define the feedforward neural network """
    def __init__(self):
        super(FNNet, self).__init__()
        self.config = Config()
        num_hiddens = [self.config.dim_x+10, self.config.dim_x+10, self.config.dim_x+1
        self.bn_layers = [
            tf.keras.layers.BatchNormalization(
                momentum=0.99,
                epsilon=1e-6,
                beta_initializer=tf.random_normal_initializer(0.0, stddev=0.1),
                gamma_initializer=tf.random_uniform_initializer(0.1, 0.5)
            )
            for _ in range(len(num_hiddens) + 2)]

        self.dense_layers = [tf.keras.layers.Dense(num_hiddens[i],
                                                   use_bias=False,
                                                   activation=None)
                             for i in range(len(num_hiddens))]

        self.dense_layers.append(tf.keras.layers.Dense(self.config.dim_z, activation=N
```

```
    def call(self, inputs, training):
        """structure: bn -> (dense -> bn -> relu) * len(num_hiddens) -> dense -> bn"""
        t, x, y = inputs
        ts = tf.ones([tf.shape(x)[0], 1], dtype=tf.dtypes.float64) * t
        x = tf.concat([ts, x, y], axis=1)
        x = self.bn_layers[0](x, training= True)
        for i in range(len(self.dense_layers) - 1):
            x = self.dense_layers[i](x)
            x = self.bn_layers[i+1](x, training= True)
            x = tf.nn.relu(x)
        x = self.dense_layers[-1](x)
        x = self.bn_layers[-1](x, training= True)
        return x

class Config(object):
    """Define the configs in the systems"""
    def __init__(self):
        super(Config, self).__init__()
        self.dim_x = 5
        self.dim_y = 5
        self.dim_z = 5
        self.num_time_interval = 25
        self.total_T = 0.1
        self.delta_t = (self.total_T + 0.0) / self.num_time_interval
        self.sqrth = np.sqrt(self.delta_t)
        self.t_stamp = np.arange(0, self.num_time_interval) * self.delta_t

    def sample(self, num_sample):
        dw_sample = normal.rvs(size=[num_sample, self.num_time_interval]) * self.sqrth
        return dw_sample[:, np.newaxis, :]

    def f_fn(self, t, x, u):
        return 0.25 * tf.reduce_sum(x ** 2, 1, keepdims=True) + tf.reduce_sum(u ** 2,

    def h_fn(self, t, x):
        return 0.5 * tf.reduce_sum(x ** 2, 1, keepdims=True)

    def b_fn(self, t, x, y, z):
        return -0.25 * x + 0.5*y + 0.5*z

    def sigma_fn(self, t, x, y, z):
        return 0.2 * x + 0.5*y + 0.5*z

    def Hx_fn(self, t, x, y, z):
        return -0.5 * x - 0.25 * y + 0.2 * z

    def hx_tf(self, t, x):
        return x

    def u_fn(self, t, x, y, z):
        return 0.5 * (y+z)

def main():
    print('Training time 1:')
    solver = Solver()
    solver.train()
    k = 10
    data = np.array(solver.training_history)
```

```
output = np.zeros((len(data[:, 0]), 3 + k))
output[:, 0] = data[:, 0] # step
output[:, 1] = data[:, 2] # y_init
output[:, 2] = data[:, 3] # loss
output[:, 3] = data[:, 1] # cost

for i in range(k - 1):
    print('Training time %3u:' % (i + 2))
    solver = Solver()
    solver.train()
    data = np.array(solver.training_history)
    output[:, 4 + i] = data[:, 1]

a = ['%d', '%.5e', '%.5e']
for i in range(k):
    a.append('%.5e')
np.savetxt('./LQ_data_d100.csv', output, fmt=a, delimiter=',')

print('Solving is done!')
```

References

[1] Hu, Y. (2014). Stochastic Maximum Principle. *Encyclopedia of Systems and Control*. Springer.

[2] Shaolin Ji, Shige Peng, Ying Peng, Xichuan Zhang (2020) Solving stochastic optimal control problem via stochastic maximum principle with deep learning method