Forward-Backward Stochastic Neural Networks: Accompanying TensorFlow 2 Code Explanation

Abstract

This code package is derived from the author's open-source TensorFlow 1.X code (Raissi, 2018) (https://maziarraissi.github.io/FBSNNs/) and has been converted to TensorFlow 2.X. Compared to the original code, it achieves a level of simplicity and readability with an exceptional fidelity to the original. The code comprises four Python files: FBSNN.py, BlackScholesBarenblatt100D.py, HamiltonJacobiBellman100D.py, and AllenCahn20D.py. Here, FBSNN serves as the parent class for the other three files, which correspond to three different methods that can run independently and produce plots. The following sections will explain each in detail.

1 Introduction

This paper will provide a detailed comparative analysis of the paper "Forward-Backward Stochastic Neural Networks: Deep Learning of High-Dimensional Partial Differential Equations" and the accompanying code, focusing on how the code implements the methods proposed in the paper.

2 Main Content of the Paper

The paper aims to address the numerical solution of high-dimensional partial differential equations (PDEs). Traditional methods encounter the curse of dimensionality in high dimensions. The author leverages the relationship between forward-backward stochastic differential equations (FBSDEs) and PDEs to propose a deep learning-based solution method.

2.1 Forward-Backward Stochastic Differential Equations (FBSDE)

The general form of the coupled forward-backward stochastic differential equations considered in the paper is given by:

$$\begin{cases} dX_t = \mu(t, X_t, Y_t, Z_t)dt + \sigma(t, X_t, Y_t)dW_t, & t \in [0, T] \\ X_0 = \xi, & \\ dY_t = \varphi(t, X_t, Y_t, Z_t)dt + Z_t^{\top}\sigma(t, X_t, Y_t)dW_t, & t \in [0, T) \\ Y_T = g(X_T) & \end{cases} \tag{1}$$

where:

- X_t is the state process,
- Y_t is the backward process,
- Z_t is the adjoint variable,
- μ is the drift coefficient function,

- σ is the diffusion coefficient function,
- φ is the generator function,
- g is the terminal condition function,
- W_t is standard Brownian motion,
- X_0 is the initial condition.

2.2 Relationship between FBSDE and PDE

Through proofs in (Antonelli, 1993) and (Ma et al., 1994), the formula (1) is related to a quasi-linear partial differential equation of the form:

$$u_t = f(t, x, u, Du, D^2u), u(T, x) = g(x) \quad (2)$$

where $u(T, x)$ is the unknown solution, and

$$f(t, x, y, z, \gamma) = \phi(t, x, y, z) - \mu(t, x, y, z)^T z - \text{Tr}\left[\sigma(t, x, y)\sigma(t, x, y)^T \gamma\right] \quad (3)$$

Here, Du represents the gradient vector of u, and D^2u represents the Hessian matrix of u. By the Itô formula, it can be seen that the solutions to equations (1) and (2) have the following relationship:

$$Y_t = u(t, X_t), Z_t = Du(t, X_t) \quad (4)$$

3 Code Implementation

The code comprises four main components:

- **The base class FBSNN, which implements the general framework for solving FBSDEs.**
- **The subclass BlackScholesBarenblatt, designed for solving the Black-Scholes-Barenblatt equation.**
- **The subclass HamiltonJacobiBellman, intended for solving the Hamilton-Jacobi-Bellman equation.**
- **The subclass AllenCahn, which is used for solving the Allen-Cahn equation.**

The following sections will analyse each of these code components and explain their correspondence with the paper.

3.1 Base Class FBSNN

3.1.1 Initialisation Method __init__

```
class FBSNN(ABC):
def __init__(self, Xi, T, M, N, D, layers):
self.Xi = Xi.astype(np.float32)
self.T = T
self.M = M
self.N = N
```

```
self.D = D
self.layers = layers
self.weights = []
self.biases = []
num_layers = len(layers)
for l in range(num_layers - 1):
in_dim = layers[l]
out_dim = layers[l + 1]
std_dev = np.sqrt(2 / (in_dim + out_dim))
W = tf.Variable(tf.random.truncated_normal([in_dim, out_dim], stddev=std_dev),
dtype=tf.float32)
b = tf.Variable(tf.zeros([1, out_dim], dtype=tf.float32))
self.weights.append(W)
self.biases.append(b)

self.optimizer = tf.keras.optimizers.Adam()
```

This method initialises the neural network's parameters, including input dimensions, output dimensions, the number of layers, and the number of neurons in each layer. The Xavier initialisation method is used to set the weights.

3.1.2 Neural Network Structure

```
def neural_net(self, X):
    H = X
    for l in range(len(self.weights) - 1):
        W = self.weights[l]
        b = self.biases[l]
        H = tf.sin(tf.matmul(H, W) + b)
    W = self.weights[-1]
    b = self.biases[-1]
    Y = tf.matmul(H, W) + b
    return Y
```

This method defines the neural network, which approximates the unknown function $u(t, x)$. The hidden layers use the sine activation function, while the output layer is linear.

3.1.3 Compute Network Output and Gradient net_u

```
def net_u(self, t, X):
    with tf.GradientTape() as tape:
        tape.watch(X)
        u = self.neural_net(tf.concat([t, X], axis=1))
    Du = tape.gradient(u, X)
    return u, Du
```

This method computes the network output Y_t, approximating the solution of the partial differential equation $u(t, X_t)$. The gradient Z_t is calculated using TensorFlow's automatic

differentiation, representing the derivative of the neural network output $u(t, X_t)$ with respect to the state X_t, i.e., $Z_t = \nabla_x u(t, X_t)$.

3.1.4 Terminal Condition Gradient Dg_tf

```
def Dg_tf(self, X):
    with tf.GradientTape() as tape:
        tape.watch(X)
        g = self.g_tf(X)
    Dg = tape.gradient(g, X)
    return Dg
```

This method computes the gradient of the terminal condition $g(X_T)$ with respect to X_T, reinforcing the terminal condition $Z_T = Du(T, X_T)$.

3.1.5 Loss Function loss_function

In the paper, the loss function is defined based on the discretisation error of the forward-backward stochastic differential equations (FBSDEs). Using the Euler-Maruyama method to discretise the FBSDE, we obtain:

$$\begin{aligned} X_{n+1} &\approx X_n + \mu(t_n, X_n, Y_n, Z_n)\Delta t_n + \sigma(t_n, X_n, Y_n)\Delta W_n \\ Y_{n+1} &\approx Y_n + \phi(t_n, X_n, Y_n, Z_n)\Delta t_n + (Z_n)^T \sigma(t_n, X_n, Y_n)\Delta W_n \end{aligned} \quad (5)$$

Where X is the state variable, Y is the target variable, and Z is the gradient. The corresponding loss function is composed of two parts:

1. **Time step error**:

$$\sum_{m=1}^{M} \sum_{n=0}^{N-1} |Y_m^{n+1} - Y_m^n - \phi(t_n, X_m^n, Y_m^n, Z_m^n)\Delta t_n - (Z_m^n)^T \sigma(t_n, X_m^n, Y_m^n)\Delta W_m^n|^2$$

2. **Terminal condition error**:

$$\sum_{m=1}^{M} |Y_m^N - g(X_m^N)|^2$$

The final loss function is:

$$\sum_{m=1}^{M} \sum_{n=0}^{N-1} |Y_m^{n+1} - Y_m^n - \phi(t_n, X_m^n, Y_m^n, Z_m^n)\Delta t_n - (Z_m^n)^T \sigma(t_n, X_m^n, Y_m^n)\Delta W_m^n|^2 + \sum_{m=1}^{M} |Y_m^N - g(X_m^N)|^2$$

This loss corresponds to M different realisations of the underlying Brownian motion, where the subscript mmm refers to the mmm-th realisation of the underlying Brownian motion, while the superscript nnn corresponds to time tnt^ntn. From equation (4), we know that $Y_m^n = u(t^n, X_m^n)$ and $Z_m^n = Du(t^n, X_m^n)$, so

this loss is the loss function for the neural network parameters $u(t,x)$. Furthermore, from equation (5), we get:

$$X_m^{n+1} = X_m^n + \mu(t^n, X_m^n, Y_m^n, Z_m^n)\Delta t^n + \sigma(t^n, X_m^n, Y_m^n)\Delta W_m^n (6)$$

Where, for each m, $X_m^0 = \xi$.

```
def loss _function(self , t, W): loss = 0
X _list = [] Y _list = []
M = tf . shape(t)[0] N = self . N
D = self . D
t0 = t[:, 0, :]
X0 = tf . tile(self . Xi, [M, 1]) Y0, Z0 = self . net _u(t0, X0)
X _list . append(X0) Y _list . append(Y0)
for n in range (N):
t1 = t[:, n + 1, :] W1 = W[:, n + 1, :] dt = t1 - t0
dW = W1 - W0
mu = self . mu _tf(t0, X0, Y0, Z0)
sigma = self . sigma _tf(t0, X0, Y0)
sigma _dW = tf .matmul(sigma , tf . expand _dims(dW, -1)) sigma _dW = tf .
squeeze(sigma _dW , axis=-1)
X1 = X0 + mu * dt + sigma _dW
phi = self . phi _tf(t0, X0, Y0, Z0)
sigma _ Z _dW = tf . reduce _sum(Z0 * sigma _dW , axis=1,
keepdims=True)
Y1 _tilde = Y0 + phi * dt + sigma _ Z _dW Y1, Z1 = self . net _u(t1,
X1)
loss += tf . reduce _sum(tf . square(Y1 - Y1 _tilde))
t0 = t1 W0 = W1 X0 = X1 Y0 = Y1 Z0 = Z1
X _list . append(X0) Y _list . append(Y0)
loss += tf . reduce _sum(tf . square(Y1 - self .g _tf(X1))) loss += tf .
reduce _sum(tf . square(Z1 - self .Dg _tf(X1)))
X = tf . stack(X _list , axis=1) Y = tf . stack(Y _list , axis=1) Y0 _pred = Y
_list[0]
return loss , X, Y, Y0 _pred
```

Y_0 and Z_0 are computed through the neural network self.net_u, where $Y_0 = u(t_0, X_0)$ and $Z_0 = \nabla_x u(t_0, X_0)$. In the paper, the code corresponding to the update equation for the state variable X (equation (5)) is as follows:

```
X1 = X0 + mu * dt + sigma _dW
```

Similarly, the code for calculating the predicted value of Yn+1Y_{n+1}Yn+1 is:

```
Y1_tilde = Y0 + phi * dt + sigma_Z_dW
```

The following code implements the time step loss function:

```
loss += tf.reduce_sum(tf.square(Y1 - Y1_tilde))
```

The following code handles the terminal loss function:

```
loss += tf.reduce_sum(tf.square(Y1 - self.g_tf(X1)))
loss += tf.reduce_sum(tf.square(Z1 - self.Dg_tf(X1)))
```

The final results for X and Y at each time step are stacked using tf.stack:

```
X = tf.stack(X_list, axis=1)
Y = tf.stack(Y_list, axis=1)
```

Through the above analysis, it is clear that the loss function defined in the paper is rigorously implemented via the loss_function method. The neural network is used to approximate the

solution of the partial differential equation $Y_t = u(t, X_t)$ and $Z_t = \nabla_x u(t, X_t)$, and the network is trained based on the Euler-Maruyama discretisation scheme for FBSDEs. The loss function adjusts the network parameters through the time step error and terminal condition error, making Y_t and Z_t closer to the true solution of the differential equation.

3.1.6 Mini-Batch Data Generation fetch_minibatch

```
def fetch_minibatch(self):
    dt = self.T / self.N
    t = np.linspace(0, self.T, self.N + 1)
    t = np.tile(t.reshape(1, -1, 1), (self.M, 1, 1)).astype(np.float32)

    dW = np.sqrt(dt) * np.random.randn(self.M, self.N + 1,
self.D).astype(np.float32)
    dW[:, 0, :] = 0
    W = np.cumsum(dW, axis=1)
    return t, W
```

This method generates simulated time sequences and Brownian motion increments, which are used for training.

3.1.7 Training Method train

```
def train(self, N_Iter, learning_rate_schedule=None):
    for it in range(N_Iter):
        if learning_rate_schedule is not None:
            lr = learning_rate_schedule(it)
            self.optimizer.learning_rate.assign(lr)

        t_batch, W_batch = self.fetch_minibatch()

        with tf.GradientTape() as tape:
            loss_value, _, _, Y0_pred = self.loss_function(t_batch, W_batch)

        grads = tape.gradient(loss_value, self.weights + self.biases)
        self.optimizer.apply_gradients(zip(grads, self.weights + self.biases))
```

This method minimises the loss function using the Adam optimiser to train the neural network parameters.

3.1.8 Prediction Method predict

```
def predict(self, Xi_star, t_star, W_star):
    Xi_star = Xi_star.astype(np.float32)
    t_star = t_star.astype(np.float32)
    W_star = W_star.astype(np.float32)

    _, X_pred, Y_pred, _ = self.loss_function(t_star, W_star)
    return X_pred.numpy(), Y_pred.numpy()
```

The purpose of this method is to use the trained model to make predictions for new inputs (time and Brownian motion paths), obtaining the corresponding state variables X and solution Y. In this context, the ultimate goal of the model is to solve stochastic differential equations (SDEs) and partial differential equations (PDEs), particularly by predicting the future state X_t and solution Y_t. In this method, $Y_t = u(t, X_t)$, which is the solution of the PDE, and X_t is the state variable. The predict method provides

predictions for X_t and Y_t under the input conditions, with X_t and Y_t representing the state and the solution, respectively. The method iterates through the stochastic differential equation to obtain predictions over the entire time interval $[0, T]$. In the predict method's loss_function, it not only calculates the loss, but also predicts X_t and Y_t across the whole time series during forward propagation. In the paper, the predictions of X_t and Y_t are realised through the discretisation of the forward-backward stochastic differential equations (FBSDEs), and this step in the code is carried out using the Euler-Maruyama method.

3.1.9 Abstract Methods

```
@abstractmethod
def phi_tf(self, t, X, Y, Z):
    pass

@abstractmethod
def g_tf(self, X):
    pass

@abstractmethod
def mu_tf(self, t, X, Y, Z):
    pass

@abstractmethod
def sigma_tf(self, t, X, Y):
    pass
```

These methods are implemented in subclasses, defining the non-linear terms, terminal conditions, drift, and diffusion coefficients for specific PDEs, which will be implemented in the subsequent three classes.

3.2 Implementation of the Black-Scholes Barenblatt Equation

3.2.1 Equation Description

In the paper, the Black-Scholes Barenblatt equation is described as follows:

$$\begin{gathered} dX_t = \operatorname{diag}(X_t) dW_t, t \in [0, T] \\ X_0 = \xi \\ dY_t = r(Y_t - Z_t^T X_t) dt + \sigma Z_t^T \operatorname{diag}(X_t) dW_t, t \in [0, T] \\ Y_T = g(X_T) \end{gathered} \tag{7}$$

where $T = 1, \sigma = 0.4, r = 0.05$, and $\xi = (1, 0.5, 1, 0.5, \ldots, 1, 0.5) \in \mathbb{R}^{100}$, and $g(x) = \| x \|^2$.

The above equation (7) is related to the Black-Scholes Barenblatt equation:

$$u_t = -\frac{1}{2} \operatorname{Tr}\left[\sigma^2 \operatorname{diag}\left(X^2\right) D^2 u\right] + r(u - (Du)^T x) \quad (8)$$

3.2.2 Code Implementation

```
class BlackScholesBarenblatt(FBSNN):
    def __init__(self, Xi, T, M, N, D, layers):
        super().__init__(Xi, T, M, N, D, layers)

    def phi_tf(self, t, X, Y, Z):
        return 0.05 * (Y - tf.reduce_sum(X * Z, axis=1, keepdims=True))
```

```
    def g_tf(self, X):
        return tf.reduce_sum(X ** 2, axis=1, keepdims=True)

    def mu_tf(self, t, X, Y, Z):
        return tf.zeros_like(X)

    def sigma_tf(self, t, X, Y):
        def diag_fn(x):
            return tf.linalg.diag(0.4 * x)

        sigma = tf.map_fn(diag_fn, X)
        return sigma
```

phi_tf implements the non-linear term $\phi(t, X_t, Y_t, Z_t) = r(Y_t - Z_t X_t)$.

g_tf implements the terminal condition $g(X_T) = \|X_T\|^2$.

mu_tf sets the drift coefficient to zero.

sigma_tf implements the diffusion coefficient $\sigma(t, X_t, Y_t) = \sigma \cdot \text{diag}\,(X_t)$.

3.2.3 Training Process

```
if __name__ == "__main__":
    M = 100
    N = 50
    D = 100

    layers = [D + 1] + 4 * [256] + [1]
    Xi = np.array([1.0, 0.5] * int(D / 2))[None, :]
    T = 1.0

    model = BlackScholesBarenblatt(Xi, T, M, N, D, layers)
    model.train(N_Iter=20, learning_rate_schedule=learning_rate_schedule)

    t_test, W_test = model.fetch_minibatch()
X_pred, Y_pred = model.predict(Xi, t_test, W_test)
```

A neural network with 5 layers, each containing 256 neurons, is used. The number of time steps N=50, and the batch size M=100.

3.3 Implementation of the Hamilton-Jacobi-Bellman Equation

3.3.1 Equation Description

In the paper, the Hamilton-Jacobi-Bellman equation is described as follows:

$$\begin{cases} dX_t = \sigma dW_t, t \in [0, T] \\ X_0 = \xi \\ dY_t = \|Z_t\|^2 dt + \sigma Z_t^T dW_t, t \in [0, T] \\ Y_T = g(X_T) \end{cases} \tag{9}$$

where $T = 1, \sigma = \sqrt{2}, \xi = (0,0,\ldots,0) \in \mathbb{R}^{100}$, and $g(x) = \ln\left(0.5(1+\| x \|^2)\right)$.

The above equation (9) is related to the Hamilton-Jacobi-Bellman equation:

$$u_t = -\mathrm{Tr}\,[D^2 u] + \| Du \|^2 \quad (10)$$

3.3.2 Code Implementation

```
class HamiltonJacobiBellman(FBSNN):
    def __init__(self, Xi, T, M, N, D, layers):
        super().__init__(Xi, T, M, N, D, layers)

    def phi_tf(self, t, X, Y, Z):
        return tf.reduce_sum(Z ** 2, axis=1, keepdims=True)

    def g_tf(self, X):
        return tf.math.log(0.5 + 0.5 * tf.reduce_sum(X ** 2, axis=1, keepdims=True))

    def mu_tf(self, t, X, Y, Z):
        return tf.zeros_like(X)

    def sigma_tf(self, t, X, Y):
        sqrt_2 = tf.sqrt(2.0)
        batch_size = tf.shape(X)[0]
        D = tf.shape(X)[1]
        sigma = sqrt_2 * tf.eye(D, batch_shape=[batch_size])
        return sigma
```

phi_tf implements the non-linear term $\phi(t, X_t, Y_t, Z_t) = \|Z_t\|^2$.

g_tf implements the terminal condition $g(X_T) = \ln\left(0.5\left(1 + \|X_T\|^2\right)\right)$..

mu_tf sets the drift coefficient to zero.

sigma_tf implements the diffusion coefficient $\sigma(t, X_t, Y_t) = \sqrt{2} \cdot I$.

3.3.3 Training Process

```
if __name__ == "__main__":
    M = 100
    N = 50
    D = 100
    layers = [D + 1] + 4 * [256] + [1]
    Xi = np.zeros([1, D], dtype=np.float32)
    T = 1.0

    model = HamiltonJacobiBellman(Xi, T, M, N, D, layers)
    model.train(N_Iter=50, learning_rate_schedule=learning_rate_schedule)

    t_test, W_test = model.fetch_minibatch()
X_pred, Y_pred = model.predict(Xi, t_test, W_test)
```

Similar to the previous models, the same network structure and training parameters are used.

3.4 Implementation of the Allen-Cahn Equation

The Allen-Cahn equation is a non-linear reaction-diffusion equation describing a phase field model, and its form is:

$$\frac{\partial u}{\partial t} = \Delta u + u - u^3, u(T, x) = g(x) \quad (11)$$

where $u(T,x)$ is the function to be solved, Δu denotes the Laplace operator, and $g(x))$ is the terminal condition. The following sections will provide a detailed derivation.

Let X_t satisfy the following forward stochastic differential equation:

$$dX_t = \mu(t, X_t, Y_t, Z_t)dt + \sigma(t, X_t, Y_t)dW_t \text{ (12)}$$

where $\mu(t, X_t, Y_t, Z_t)$ is the drift coefficient, $\sigma(t, X_t, Y_t)$ is the diffusion coefficient, and W_t is standard Brownian motion. In the Allen-Cahn equation, we take:

$$\mu(t, X_t, Y_t, Z_t) = 0, \sigma(t, X_t, Y_t) = I \text{ (13)}$$

where I is the identity matrix. The backward stochastic differential equation is then:

$$dY_t = \phi(t, X_t, Y_t, Z_t)dt + Z_t^T \sigma(t, X_t, Y_t)dW_t \text{ (14)}$$

with the terminal condition $Y_T = g(X_T)$. In the Allen-Cahn equation, the non-linear term and terminal condition are given as:

$$\phi(t, X_t, Y_t, Z_t) = -Y_t + Y_t^3, g(X_T) = \frac{1}{2+0.4\|X_T\|^2}.(15)$$

3.4.1 Application of Itô's Formula

Assuming $Y_t = u(t, X_t)$ and $Z_t = D_x u(t, X_t)$, where $D_x u$ is the gradient of uuu with respect to x, by Itô's formula, we have:

$$du(t, X_t) = \left(\frac{\partial u}{\partial t} + \mu^T D_x u + \frac{1}{2}\text{Tr}\left(\sigma\sigma^T D_x^2 u\right)\right)dt + (\sigma^T D_x u)^T dW_t \text{ ,(16)}$$

where D_x^2 is the Hessian matrix of u with respect to x.

3.4.2 Comparing $du(t, X_t)$ and dY_t

Since $dY_t = du(t, X_t)$ and $\sigma = I, \mu = 0$, we can substitute and obtain:

$$du(t, X_t) = \left(\frac{\partial u}{\partial t} + \frac{1}{2}\Delta u\right)dt + Du^T dW_t.(17)$$

On the other hand, dY_t is:

$$dY_t = (-Y_t + Y_t^3)dt + Z_t^T dW_t.(18)$$

Because $Y_t = u(t, X_t)$ and $Z_t = Du(t, X_t)$, by comparing the dtdtdt and dWtdW_tdWt terms, we get:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\Delta u + Y_t - Y_t^3 = 0.(19)$$

Rearranging:

$$\frac{\partial u}{\partial t} = \frac{1}{2}\Delta u + (-u + u^3).(20)$$

Since $u^3 - u = u(u^2 - 1)$, we can see that this PDE is indeed in the form of the Allen-Cahn equation.

3.4.3 Equation Description

In the paper, the Allen-Cahn equation is:

$$\begin{cases} dX_t = dW_t, t \in [0, T] \\ X_0 = \xi \\ dY_t = (-Y_t + Y_t^3)dt + Z_t^T dW_t, t \in [0, T] \\ Y_T = g(X_T) \end{cases} \text{(21)}$$

where $T = 0.3, \xi = (0,0,\ldots,0) \in \mathbb{R}^{20}$, and $g(x) = (2 + 0.4 \parallel x \parallel^2)^{-1}$.

The above equation (21) is related to the Allen-Cahn equation:

$$u_t = -\frac{1}{2}\mathrm{Tr}\left[D^2 u\right] - u + u^3. (22)$$

3.4.4 Code Implementation

```
class AllenCahn(FBSNN):
    def __init__(self, Xi, T, M, N, D, layers):
        super().__init__(Xi, T, M, N, D, layers)

    def phi_tf(self, t, X, Y, Z):
        return -Y + Y ** 3

    def g_tf(self, X):
        return 1.0 / (2.0 + 0.4 * tf.reduce_sum(X ** 2, axis=1, keepdims=True))

    def mu_tf(self, t, X, Y, Z):
        return tf.zeros_like(X)

    def sigma_tf(self, t, X, Y):
        batch_size = tf.shape(X)[0]
        D = tf.shape(X)[1]
        sigma = tf.eye(D, batch_shape=[batch_size])
        return sigma
```

Drift coefficient μ\muμ: In the code, the mu_tf method returns a zero vector, corresponding to $\mu(t, X_t, Y_t, Z_t) = 0$.

Diffusion coefficient σ: The sigma_tf method returns the identity matrix, corresponding to $\sigma(t, X_t, Y_t) = I$.

Non-linear term ϕ: The phi_tf method returns $-Y_t + Y_t^3$, corresponding to $\phi(t, X_t, Y_t, Z_t) = -Y_t + Y_t^3$.

Terminal condition g: The g_tf method implements $g(X_T) = \left(2 + 0.4\|X_T\|^2\right)^{-1}$.

3.4.5 Training Process

```
if __name__ == "__main__":
    M = 100
    N = 15
    D = 20
    layers = [D + 1] + 4 * [256] + [1]
    T = 0.3
    Xi = np.zeros([1, D], dtype=np.float32)

    boundaries = [20000, 50000, 80000]
    values = [1e-3, 1e-4, 1e-5, 1e-6]
    learning_rate_fn =
tf.keras.optimizers.schedules.PiecewiseConstantDecay(boundaries, values)
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate_fn)

    model = AllenCahn(Xi, T, M, N, D, layers)
    model.optimizer = optimizer
    N_Iter = 50
    model.train(N_Iter)

    t_test, W_test = model.fetch_minibatch()
    X_pred, Y_pred = model.predict(Xi, t_test, W_test)
```

```
    samples = 5
    Y_test_terminal = 1.0 / (2.0 + 0.4 * np.sum(X_pred[:, -1, :] ** 2, axis=1,
keepdims=True))
```

Model parameters: Batch size M=100, time steps N=15, dimension D=20, terminal time T=0.3.

Neural network structure: The input layer has a dimension of D+1 (the concatenation of time t and state X), with 4 hidden layers, each with 256 neurons, and the sine activation function. The output layer has a dimension of 1.

Optimizer: The Adam optimizer is used, and the learning rate follows a piecewise constant decay schedule.

Initial conditions: $X_0 = \xi = 0$.

Training process: The training is performed by calling model.train(N_Iter).

Prediction process: The model.predict method is used to make predictions on the test data.

4 Code and Paper Correspondence

4.1 Neural Network Approximation of $u(t,x)$

In the paper, the authors propose using a deep neural network to approximate the unknown solution $u(t,x)$ and to compute the gradient $Du(t,x)$ using automatic differentiation. In the code, the neural_net method achieves this, with the input being the concatenation of time and space, and the output being the predicted value of $u(t,x)$.

4.2 Automatic Differentiation to Compute $Du(t,x)$

In the code, tf.GradientTape is used in the net_u method to compute the gradient of the network's output with respect to the input, i.e., $Z_t = Du(t, X_t)$.

4.3 Construction of the Loss Function

In the paper, the loss function is based on the Euler-Maruyama discretisation scheme, including the time step error and the deviation from the terminal condition. In the code, the loss_function method implements this loss function by accumulating the error at each time step, along with the deviations of Y_T and $g(X_T)$.

4.4 Training Process

The paper uses the Adam optimiser to minimise the loss function with a batch size of 100. In the code, the train method implements this process and provides the ability to schedule the learning rate.

4.5 Approximation of the Solution Function $u(t,x)$

The paper emphasises that the method can approximate the entire solution function, not just at the initial time and initial point. In the code, the predict method allows for predictions of $u(t,x)$ at any time and state, demonstrating this capability.