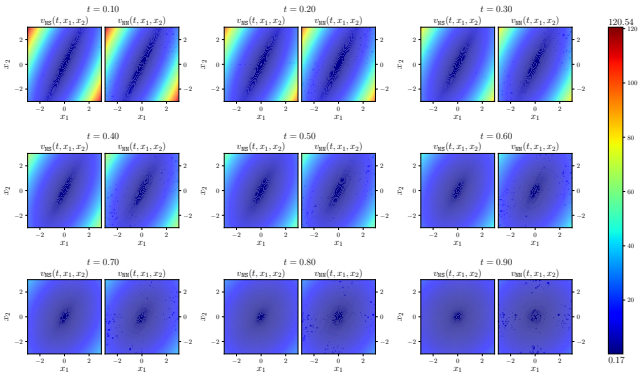
SCDAA Coursework Report



Yuebo Yang[*], Yu Wang[†], Zheyan Lu[‡]

April 11, 2024

The project source code can be found on GitHub at:`https://github.com/Warren12138/SCDAA`

[*]s2601126
[†]s1973765
[‡]s2559490

1 Linear quadratic regulator

Exercise 1.1 Solving LQR using from Ricatti ODE.

According to the requirement, we created a class named `LQR_solver` for solving Linear quadratic regulator, which mainly contains 3 methods listed as follows:

1. `solve_riccati_ode`: a numerical solver for Ricatti ODE. We offered two different numerical algorithm for solving the ODE for user to choose, which are 1) Euler method and 2) order 4 Runge-Kutta method. The corresponding code is presented below.

```python
def solve_riccati_ode(self, time_grids):
    ...
    time_grids_in = torch.flip(time_grids, dims=[1])
    S = self.R.clone()
    repl = torch.ones(time_grids_in.shape)
    rep_exd = repl.unsqueeze(-1).unsqueeze(-1)
    S_exd = S.unsqueeze(0).unsqueeze(0)
    S_repl = rep_exd*S_exd
    dt = time_grids_in[:,1:]-time_grids_in[:,:-1]

    for i in range(dt.shape[1]):

        if self.method == "euler":
            S_repl[:,i+1] = self.euler_step(S_repl[:,i], dt[:,i])
        elif self.method == "rk4":
            S_repl[:,i+1] = self.rk4_step(S_repl[:,i], dt[:,i])
        else:
            raise ValueError("Unsupported method")

    return torch.flip(S_repl, dims=[1])

def euler_step(self, S_in, dt_in):
    dS_in = -2 * self.H.T @ S_in + S_in @ self.M @ torch.inverse(self.D) @ self.M.T @
        S_in - self.C
    dt_resized = dt_in[:, None, None]

    return S_in + dS_in * dt_resized

def rk4_step(self, S_in, dt_in):
    def riccati_derivative(S):
        return -2 * self.H.T @ S_in + S_in @ self.M @ torch.inverse(self.D) @ self.M.T
            @ S_in - self.C
    dt_resized = dt_in[:, None, None]
    k1 = riccati_derivative(S_in)
    k2 = riccati_derivative(S_in + 0.5 * k1 * dt_resized)
    k3 = riccati_derivative(S_in + 0.5 * k2 * dt_resized)
    k4 = riccati_derivative(S_in + k3 * dt_resized)

    return S_in + (k1 + 2*k2 + 2*k3 + k4)*dt_resized/6
```

2. `value_function`: computation of the value function $v(t, x)$, for which we developed two strategies to calculate $S(t)$, an essential component in determining $v(t, x)$. Our

preferred method involves computing a uniformly discrete $S(t)$ across the entire time horizon $[0, T]$. Subsequently, we employed cubic spline interpolation of $S(t)$ to efficiently compute $v(t, x)$ for each distinct (t, x) within a single batch of inputs. This approach guarantees a consistent level of numerical error across computations. The corresponding code is presented below.

```
def value_function(self, t_batch, x_batch, sol_method = 'interpolation'):

   if sol_method == 'interpolation':

      N_step = 2*self.N_step

      if not (t_batch.dim() == 1 and torch.all((t_batch >= 0) & (t_batch <= 1))):
         raise TypeError("t_batch should be a 1D tensor in which every entry is in
             [0,1].")
      else:
         if not (x_batch.dim() == 3 and x_batch.size()[0] == len(t_batch) and
              x_batch.size()[1] == 1 and x_batch.size()[2] == self.H.size()[0]):
            raise TypeError("x_batch should have shape (%d, 1,
                %d)."%(len(t_batch),self.H.size(2)))

      time_grid = torch.stack([torch.linspace(0, self.T, N_step, dtype =
          torch.double) for i in [0]])

      S_tensor_tensor = self.solve_riccati_ode(time_grid)

      index_s_1 = torch.searchsorted(time_grid[0,:], torch.min(t_batch), right=True)
          - 1
      time_grid_for_intpl = time_grid[0,index_s_1:]
      S_tensor_tensor_for_intpl = S_tensor_tensor[0,index_s_1:]

      time_grid_for_intpl_np = time_grid_for_intpl.numpy()
      S_tensor_tensor_for_intpl_np = S_tensor_tensor_for_intpl.numpy()
      t_batch_np = t_batch.numpy()

      S_c_spl = CubicSpline(time_grid_for_intpl_np, S_tensor_tensor_for_intpl_np)

      traces = torch.einsum('bii->b', self.sigma @ self.sigma.transpose(1,2) @
          S_tensor_tensor_for_intpl).unsqueeze(1).unsqueeze(2)

      trace_cubic = CubicSpline(time_grid_for_intpl_np, traces.numpy())

      def S_intpl(t_batch_np_in):
         return torch.from_numpy(S_c_spl(t_batch_np_in))

      S_t_s = S_intpl(t_batch_np)
      x_batch_T = x_batch.transpose(1, 2)

      xTSx = x_batch @ S_t_s @ x_batch_T

      for i in range(len(t_batch_np)):

         t1_index = (time_grid_for_intpl >
             t_batch[i]).nonzero(as_tuple=True)[0].min().item() if
             (time_grid_for_intpl > t_batch[i]).any() else None
```

```
            time_grid_after_t1 = time_grid_for_intpl[t1_index:]
            dt_t1 = time_grid_after_t1[0] - t_batch[i]
            dt_after_t1 = time_grid_after_t1[1:]-time_grid_after_t1[:-1]
            traces_after_t1 = traces[t1_index:]
            traces_after_t1_for_int = torch.tensor(0.5,dtype = torch.double) *
                (traces_after_t1[1:] + traces_after_t1[:-1])
            int_t1 = torch.tensor(0.5,dtype = torch.double) *
                (torch.from_numpy(trace_cubic(t_batch_np[i])).to(dtype = torch.double)
                + traces_after_t1[0])
            intgl = (dt_t1*int_t1).squeeze() + (dt_after_t1.view(1,
                -1)@traces_after_t1_for_int.squeeze(1))

            xTSx[i,0]+= intgl.squeeze()

        v_tx = xTSx.squeeze()

    ...

    return v_tx
```

3. `markov_control`: a computation of Markov control function $a(t, x)$, which shares the same strategies and structure of coding as `value_function`. The corresponding code is presented below.

```
def markov_control(self, t_batch, x_batch, sol_method = 'interpolation'):

    if sol_method == 'interpolation':

        N_step = 2*self.N_step

        if not (t_batch.dim() == 1 and torch.all((t_batch >= 0) & (t_batch <= 1))):
            raise TypeError("t_batch should be a 1D tensor in which every entry is in
                [0,1].")
        else:
            if not (x_batch.dim() == 3 and x_batch.size()[0] == len(t_batch) and
                x_batch.size()[1] == 1 and x_batch.size()[2] == self.H.size()[0]):
                raise TypeError("x_batch should have shape (%d, 1,
                    %d)."%(len(t_batch),self.H.size(2)))

        time_grid = torch.stack([torch.linspace(0, self.T, N_step, dtype =
            torch.double) for i in [0]])

        S_tensor_tensor = self.solve_riccati_ode(time_grid)

        index_s_1 = torch.searchsorted(time_grid[0,:], torch.min(t_batch), right=True)
            - 1
        time_grid_for_intpl = time_grid[0,index_s_1:]
        S_tensor_tensor_for_intpl = S_tensor_tensor[0,index_s_1:]

        time_grid_for_intpl_np = time_grid_for_intpl.numpy()
        S_tensor_tensor_for_intpl_np = S_tensor_tensor_for_intpl.numpy()
        t_batch_np = t_batch.numpy()

```

```
        S_c_spl = CubicSpline(time_grid_for_intpl_np, S_tensor_tensor_for_intpl_np)

        def S_intpl(t_batch_np_in):
            return torch.from_numpy(S_c_spl(t_batch_np_in))

        S_t_s = S_intpl(t_batch_np)
        x_batch_T = x_batch.transpose(1, 2)

        MT = self.M.T
        D_inv = torch.inverse(self.D)
        x = torch.transpose(x_batch,dim0 = 2,dim1 = 1)
        a_tx = - D_inv @ MT @ S_t_s @ x_batch_T
        a_tx = torch.transpose(a_tx,dim0 = 1,dim1 = 2).squeeze()

    ...

    return a_tx.unsqueeze(1)
```

Exercise 1.2 LQR MC check.

Utilizing the LQR solver from **Exercise 1.1**, we employed Monte Carlo simulation to compare the outcomes with those obtained in **Exercise 1.1**. Specifically, we varied the number of Monte Carlo samples and adjusted the number of time steps to explore how these changes affect the simulation results.

Initialization

We set the required matrices to values as follows:

$$H = \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, \quad M = \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, \quad C = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix},$$
$$D = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, \quad R = \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma = \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}.$$

This part in our Python code is presented as follows, where the shapes of the vectors are adjusted due to the convenience of matrix multiplication.

```
H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = torch.double)
M = torch.tensor([[0.5, 0.7], [0.3, 1.0]], dtype = torch.double)
sigma = torch.tensor([[[0.08], [0.11]]], dtype = torch.double)
C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = torch.double)
D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = torch.double)
R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = torch.double)
T = torch.tensor(1.0, dtype = torch.double)
```

Then, we first planned to do this comparison on a wider range of the variable space, which is

$$
\begin{aligned}
t &\in \{0.1, 0.2, \ldots, 0.9\}, \\
x_1 &\in \{0.1, 0.2, \ldots, 0.9\}, \\
x_2 &\in \{0.1, 0.2, \ldots, 0.9\},
\end{aligned}
$$

i.e.,

$$
\begin{aligned}
\{(t, x)^i, i = 1, \ldots, N\} &= \{0.1, 0.2, \ldots, 0.9\} \times \{0.1, 0.2, \ldots, 0.9\}^2, \\
N &= 9 \times 9 \times 9.
\end{aligned}
$$

Finally, due to the limitation of our computational resources, we chose a much smaller one, which is

$$
\begin{aligned}
t &\in \{0.1\}, \\
x_1 &\in \{0.5\}, \\
x_2 &\in \{0.5\},
\end{aligned}
$$

i.e.,

$$
\begin{aligned}
\{(t, x)^i, i = 1\} &= \{0.1\} \times \{0.5\}^2, \\
N &= 1.
\end{aligned}
$$

Setting for Monte Carlo Simulation

In order to analyze the two parameters' impact on the accuracy of Monte Carlo simulation, we divided our calculation tasks into two parts:

1. Fix sampling size (`FSS = int(1e5)`), and variate time step number (`FSS_VTSN = [int(x) for x in [1e0,1e1,5e1,1e2,5e2,1e3,5e3]]`)
2. Fix time step number (`FTSN = int(5e3)`), and variate sampling size (`FTSN_VSS = [int(x) for x in[1e1,5e1,1e2,5e2,1e3,5e3,1e4,5e4,1e5]]`)

For all computation tasks, we crafted a shell script named `run_MCs.sh`, which is executed in the terminal. This script invokes `Exercise1_2_parallel_MC.py`, our custom Python script designed for conducting the Monte Carlo simulation's main process. It performs the simulation through parallel computation and ultimately generates the corresponding data files storing the $J^\alpha(t, x)$. These files are then organized into subfolders, each named according to different parameter settings.

One of our group members developed both the explicit scheme, utilizing a `for` loop approach, and the implicit scheme, through direct computation of a massive sparse matrix's inverse. For the implicit scheme, we further optimized the process by decomposing the large sparse matrix into relatively smaller blocks to facilitate the computation of its inverse.

The corresponding codes of the two schemes is presented below.

1. Explicit scheme:

```
def MonteCarloSampler(iteration, params):
    #Explicit scheme
    C = params['C']
    D = params['D']
    N = params['N']
    R = params['R']
    S = params['S']
    X0 = params['X0']
    H = params['H']
    dt = params['dt']
    multX = params['multX']
    multa = params['multa']
    sig = params['sig']

    X_0_N = X0

    for i in range(N):

        X_next = ((torch.eye(2) + dt[i] * (H + multX@S[i])) @ X_0_N[i:].transpose(1,2)
            + sig * torch.sqrt(dt[i]) * torch.randn(1)).transpose(1,2)
        X_0_N = torch.cat((X_0_N, X_next), dim=0)

    alp = multa @ X_0_N.transpose(1,2)

    int_ = X_0_N @ C @ X_0_N.transpose(1,2) + alp.transpose(1,2) @ D @ alp
    J = X_0_N[-1] @ R @ X_0_N[-1].T + torch.tensor(0.5)*dt @
        ((int_.squeeze(1)[1:]+int_.squeeze(1)[:-1]))

    return J
```

2. Implicit scheme:

```
def implicit(it,params):

    #Implicit code by Yuebo Yang Mar.25.2024

    device = params['device']
    AA = params['AA'].to(device)

    C = params['C'].to(device)
    D = params['D'].to(device)
    N = params['N']
    R = params['R'].to(device)

    Step_limit = params['Step_limit']
    X0 = params['X0'].to(device)
    dt = params['dt'].to(device)
    multa = params['multa'].to(device)
    sig = params['sig'].to(device)

    b = torch.cat((X0.squeeze(),sig.squeeze().repeat(N-1) *
        torch.sqrt(dt).repeat_interleave(len(X0.squeeze())) *
```

```
        torch.randn(len(sig.squeeze().repeat(N-1)), dtype=torch.float32, device =
        device)))

    if N//Step_limit >= 1:

        X_0_N = torch.clone(X0.squeeze().repeat(N))

        for i in range(N//Step_limit):

            b_in_batch = torch.clone(b[i*Step_limit*2:(i+1)*Step_limit*2])

            if i == 0:
                _X_0 = torch.zeros_like(b_in_batch[:2],dtype = torch.float32, device =
                    device)
            else:
                _X_0 = torch.clone(X_0_N[i*(Step_limit)*2-2:i*(Step_limit)*2])

            b_in_batch[:2] = b_in_batch[:2]+_X_0

            AA_in_batch = torch.clone(AA[i * Step_limit * 2 : (i+1) * Step_limit * 2,
                i * Step_limit * 2 : (i+1) * Step_limit * 2])

            X_0_N[i * Step_limit * 2 : (i+1) * Step_limit * 2] =
                (torch.inverse(AA_in_batch) @ b_in_batch)

        if (N%Step_limit != 0):
            _X_0 = torch.clone(X_0_N[-(N % Step_limit + 1) * 2 : - (N % Step_limit) *
                2])

            #final_section

            b_fin = torch.clone(b[-(N % Step_limit) * 2 : ])
            b_fin[:2] = torch.clone(b_fin[:2]+_X_0)
            AA_fin = AA[-(N%Step_limit)*2:,-(N%Step_limit)*2:]

            X_0_N[-(N%Step_limit)*2:] = (torch.inverse(AA_fin)@b_fin)

        X_0_N = X_0_N.reshape(N,1,2)

    else:

        X_0_N = (torch.inverse(AA)@b).reshape(N,1,2)

    alpha = multa@X_0_N.transpose(1,2)

    int_ = X_0_N@C@X_0_N.transpose(1,2) + alpha.transpose(1,2)@D@alpha

    J = X_0_N[-1]@R@X_0_N[-1].T + torch.tensor(0.5, dtype=torch.float32, device =
        device)*dt@((int_.squeeze(1)[1:]+int_.squeeze(1)[:-1]))

    return J.to('cpu')
```

Analysis and Conclusion

Upon completing all simulations, we compiled the data across various settings and graphed the squares of the differences between them and the numerical solution.

The evident convergence behavior observed in both fig. 1.1 and fig. 1.2 underscores the accuracy of our Monte Carlo simulation code.

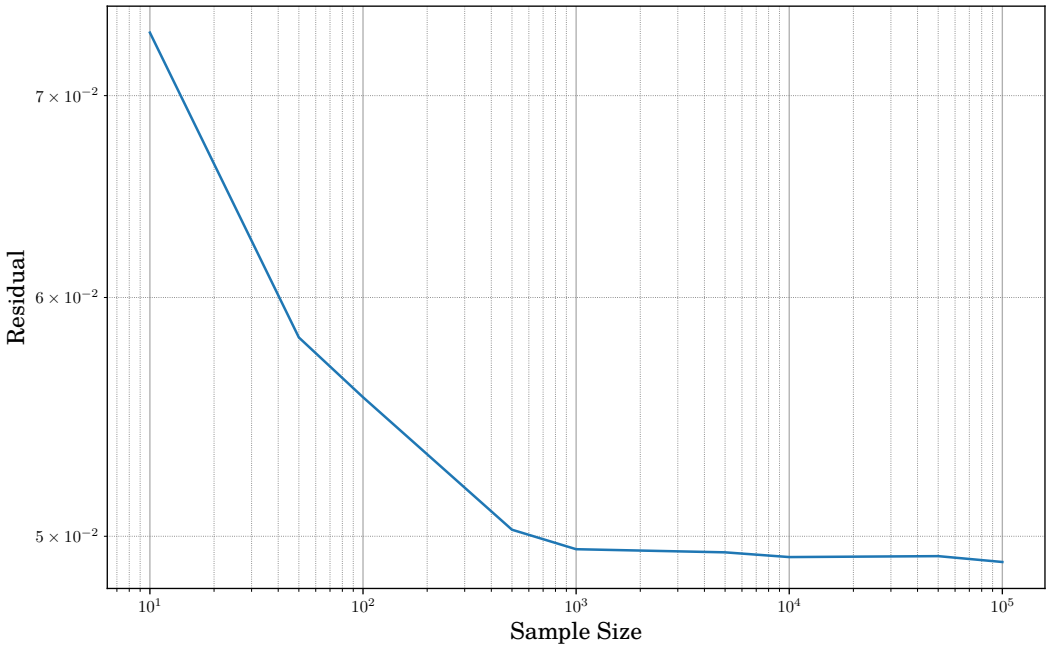


Figure 1.1: Residual convergence over increasing sample size

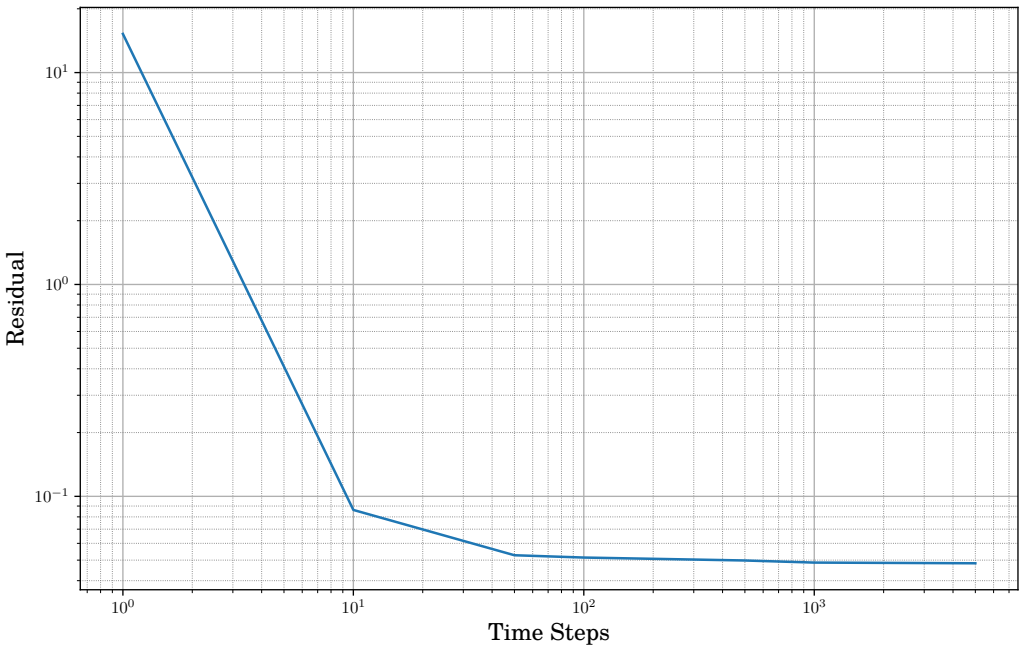


Figure 1.2: Residual convergence over increasing time steps

2 Supervised learning, checking the NNs are good enough

Exercise 2.1 Supervised learning of value function v.

Exercise 2.2 Supervised learning of Markov control a.

Initialization

To approximate the value function, which has been explicitly calculated in **Exercise 1.1**, we employed a neural network model as an estimation tool. This necessitates an initial configuration for the implementation.

$$H = \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, \qquad M = \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, \qquad C = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix},$$
$$D = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, \qquad R = \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \sigma = \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}.$$

```
H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = torch.double)
M = torch.tensor([[0.5,0.7], [0.3,1.0]], dtype = torch.double)
sigma = torch.tensor([[[0.8],[1.1]]], dtype = torch.double)
C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = torch.double)
D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = torch.double)
R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = torch.double)
T = torch.tensor(1.0, dtype = torch.double)
```

Framework

We trained a neural network model for **Exercise 2.1** in conjunction with the optimizer `torch.optim.Adam` to identify network parameters that yield an accurate approximation of the value functions.

```
class ValueFunctionNN(nn.Module):
    def __init__(self):
        # Inherit properties from the parent class nn.Module
        super(ValueFunctionNN, self).__init__()
        # First layer takes 3 inputs (time t and spatial variables x1 and x2) and
            outputs 100 features
        self.layer1 = nn.Linear(3, 100)
        # Second and third layers are fully connected layers with 100 neurons each
        self.layer2 = nn.Linear(100, 100)
        self.layer3 = nn.Linear(100, 100)
        # ReLU (Rectified Linear Unit) activation function for introducing
            non-linearity
        self.relu = nn.ReLU()
        # Output layer: reduces the dimension from 100 to 1, representing the value
            function
```

```
        self.output = nn.Linear(100, 1)

    def forward(self, x):
        # Defines the forward pass of the neural network
        # Applies ReLU activation function after each layer except for the output layer
        x = self.relu(self.layer1(x)) # Input passes through the first layer and ReLU
        x = self.relu(self.layer2(x)) # Then, it passes through the second layer and
            ReLU
        x = self.relu(self.layer3(x)) # Finally, it passes through the third layer and
            ReLU
        return self.output(x) # The output layer produces the final value function
            approximation
```

Upon addressing the computational challenges, we executed the model training employing the following strategic approach:

i. Set `iteration` $= 5$. In each `iteration` we set `epochs` $= 1000$.

ii. In each `epoch`, we choose a `sample_size` $= N$ to sample the required space $(t, x) \in [0, T) \times [-3, 3]^2$. After we get one batch of this sample points $\{(t, x)^i,\ i = 1, \ldots, N\}$, we can calculate the corresponding $\{(u(t, x))^i,\ i = 1, \ldots, N\}$, and their gradients as well as Hessian matrices.

iii. Following the expression of $R(\theta)$, we can construct the `loss` function for the training.

iv. During the training process, we employ `torch.optim.Adam` as our optimizer with an initial `learning_rate` $= 0.001$.

Subsequently, we proceeded to employ the previously configured neural network model to undertake the data training process.

```
# Initializes the model for approximating the value function
model_value = ValueFunctionNN().double()

# Sets up the optimizer with a learning rate of 0.001
# Adam optimizer is used for its adaptive learning rate properties
optimizer_value = torch.optim.Adam(model_value.parameters(), lr=0.001)

# Defines the loss function as Mean Squared Error (MSE) to measure
# the difference between the predicted and true values
criterion_value = nn.MSELoss()

# Initializes a list to store the loss for each epoch, useful for visualizing the
    training process
epoch_losses = []

# Specifies the batch size and number of epochs for training
batch_size = 5
epochs = 1000

# Training loop over the specified number of batches
for batch in range(batch_size):
    print(f'Batch {batch+1}/{batch_size}'+'\n')

```

```
    # Generates new data for training
    t_data, x_data, v_data = new_data(10000, 'value')

    # Prepares the input by concatenating time and spatial data
    inputs = torch.cat((t_data.unsqueeze(-1), x_data.squeeze(1)), dim=1)

    # Creates a DataLoader for batch processing
    dataset = TensorDataset(inputs, v_data)
    dataloader = DataLoader(dataset, batch_size=512, shuffle=True)

    # Iterates over each epoch for training
    for epoch in range(epochs):
        model_value.train() # Sets the model to training mode
        total_loss = 0

        # Iterates over each mini-batch
        for batch_idx, (data, target) in enumerate(dataloader):
            optimizer_value.zero_grad() # Clears the gradients of all optimized tensors
            output = model_value(data) # Forward pass: computes the model's output
            loss = criterion_value(output, target.unsqueeze(1)) # Computes the loss
            loss.backward() # Backward pass: computes the gradient of the loss w.r.t.
                the parameters
            optimizer_value.step() # Updates the parameters
            total_loss += loss.item() # Aggregates the loss

        # Stores the average loss for this epoch
        epoch_losses.append(total_loss / len(dataloader))

        # Prints loss information at specific epochs
        if epoch == 0 or (epoch + 1) % 100 == 0:
            print(f'Epoch {epoch+1}/{epochs} \t Loss: {total_loss / len(dataloader)}')
    print('\n')
```

In this section, we have only demonstrated the framework and training process code for the value function aspect. The setup and training for the Markov control are analogous.

Analysis and Conclusion

Figure 2.1 illustrates a discernible decline in the training loss for the value function approximation as the training progresses, indicating continuous convergence across successive

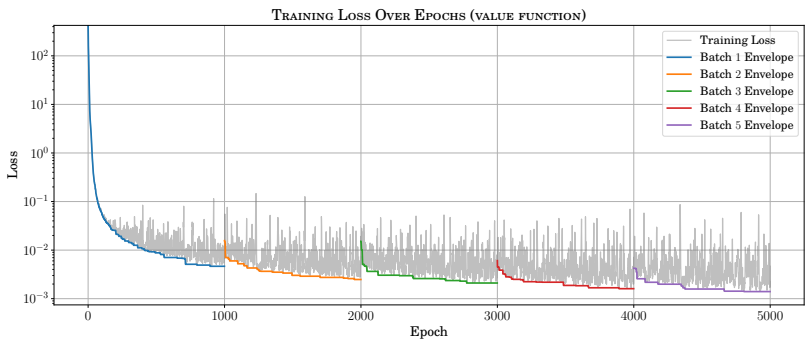


Figure 2.1: Training loss over epochs (value function)

training batches.

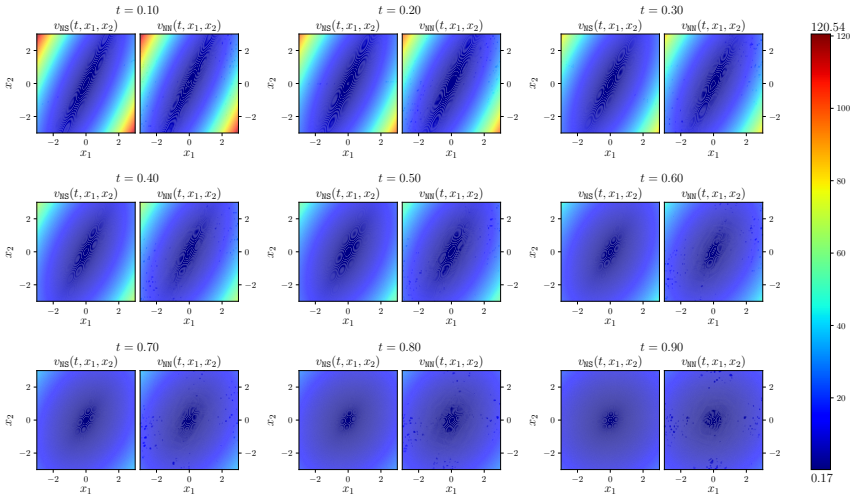


Figure 2.2: Visual comparison between v_{NS} and v_{NN}, $t \in [0.1, ..., 0.9]$

We created a comparative visualization of the value function in fig. 2.2, illustrating the neural network approximation against numerical methods across time which is $t \in \{0.1, \ldots, 0.9\}$ and a two-dimensional spatial scale. From the graph, it is evident that the two approaches yield closely aligned results.

Similarly to the previous model for the value function, after employing the same training setup, we obtain the training loss plot for the Markov control model:

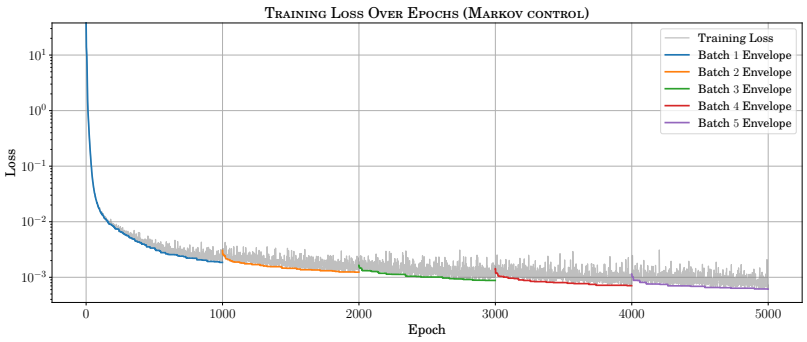


Figure 2.3: Training Loss over epochs (Markov control)

Observing fig. 2.3 in comparison to fig. 2.1, it is evident that the training loss for the Markov control model converges swiftly to a significantly lower value, resulting in a tidier curve that continues to decrease across subsequent batches. This suggests that, in theory, enhancing the neural network architecture with additional layers and extending the training duration could potentially lead the training loss to approach zero. Nonetheless, constrained by available computational resources such as CPUs or GPUs, this represents the extent of optimization achievable under current conditions.

3 Deep Galerkin approximation for a linear PDE

Exercise 3.1 Deep galerkin, Linear PDE.

Initialization

To solve the linear PDE

$$\partial_t u + \tfrac{1}{2}\mathrm{tr}(\sigma\sigma^\top \partial_{xx} u) + (\partial_x u)^\top Hx + (\partial_x u)^\top M\alpha + x^\top Cx + \alpha^\top D\alpha = 0 \quad \text{on } [0,T)\times\mathbb{R}^2,$$
$$u(T,x) = x^\top Rx \quad \text{on } \mathbb{R}^2,$$

where $\alpha = (1,1)^\top$, we set the matrices to values as follows:

$$H = \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, \qquad M = \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, \qquad C = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix},$$
$$D = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, \qquad R = \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \sigma = \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}.$$

Together with α and time horizon $T = 1$, this section in our Python code is presented as follows, where the shapes of the vectors are adjusted due to the convenience of matrix multiplication.

```
Proj_dtype = torch.double
Proj_device = 'cpu'

H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = Proj_dtype, device = Proj_device)
M = torch.tensor([[0.5, 0.7], [0.3, 1.0]], dtype = Proj_dtype, device = Proj_device)
C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = Proj_dtype, device = Proj_device)
D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = Proj_dtype, device = Proj_device)
R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = Proj_dtype, device = Proj_device)
T = torch.tensor(1.0, dtype = Proj_dtype, device = Proj_device)
sigma = torch.tensor([[[0.08],[0.11]]], dtype = Proj_dtype, device = Proj_device)
alpha = torch.tensor([[[1.0],[1.0]]], dtype = Proj_dtype, device = Proj_device)
```

Priliminary Work

We primarily opted for the simplest architecture, a Fully Connected Neural Network (FCNN), as the foundation for our model, the corresponding code is in the Jupyter Notebook file `Exercise3_1_primary.ipynb`.

Here is a code presentation of our primary NN structure.

```
class DGMNN(nn.Module):

    def __init__(self):
        super(DGMNN, self).__init__()
        # Define three linear layers with 100 neurons each
```

```
        self.layer1 = nn.Linear(3, 100) # Input layer (takes 3-dimensional input)
        self.layer2 = nn.Linear(100, 100) # Hidden layer
        self.layer3 = nn.Linear(100, 100) # Hidden layer
        # Define activation functions
        self.tanh = nn.Tanh() # Tangent hyperbolic activation function
        self.relu = nn.ReLU() # Rectified Linear Unit activation function
        # Define the output layer
        self.output = nn.Linear(100, 1) # Output layer (produces 1-dimensional output)

    def forward(self, x):
        # Forward pass through the network
        x = self.relu(self.layer1(x)) # Activation after first layer
        x = self.tanh(self.layer2(x)) # Activation after second layer
        x = self.relu(self.layer3(x)) # Activation after third layer

        return self.output(x) # Return the output of the network
```

In this exercise, the aspect we found more challenging than the others was computing the Hessian matrix within the framework of PyTorch. This difficulty arises because directly applying the `torch.autograd.grad` method to $\partial_x u$ yields only the diagonal entries of the Hessian matrices. Such an outcome does not meet our requirements, as we need the full Hessian matrix for its multiplication. Therefore, we had to resort to calculating it through a loop-based approach. Here is the code for this crucial step.

```
def get_hessian(grad, x):

    Hessian = torch.tensor([], device=Proj_device) # Initialize empty tensor for
        storing Hessian matrices

    # Iterate over each element in x to compute second derivatives
    for i in range(len(x)):
        hessian = torch.tensor([], device=Proj_device) # Temporary tensor for storing
            Hessian of the current element
        # Compute second derivative for each component of the gradient
        for j in range(len(grad[i])):
            # Compute the second derivative using autograd.grad
            u_xxi = torch.autograd.grad(
                                        grad[i][j],
                                        x,
                                        grad_outputs=torch.ones_like(grad[i][j]),
                                        retain_graph=True,
                                        create_graph=True,
                                        allow_unused=True
                                        )[0]
            # Concatenate the second derivative to the temporary Hessian tensor
            hessian = torch.cat((hessian, u_xxi[i].unsqueeze(0)))
        # Concatenate the computed Hessian for the current element to the final
            Hessian tensor
        Hessian = torch.cat((Hessian, hessian.unsqueeze(0)), dim=0)

    return Hessian
```

After solving all the computation problems, we performed the model training with our

training strategy as follows:

i. Set `iteration` $= 5$. In each `iteration` we set `epochs` $= 40$.

ii. In each `epoch`, we choose a `sample_size` $= N$ to sample the required space $(t, x) \in [0, T) \times [-3, 3]^2$. After we get one batch of this sample points $\{(t, x)^i,\ i = 1, \ldots, N\}$, we can calculate the corresponding $\{(u(t, x))^i,\ i = 1, \ldots, N\}$, and their gradients as well as Hessian matrices.

iii. Following the expression of $R(\theta)$, we can construct the `loss` function for the training.

iv. During the training process, we employ `torch.optim.Adam` as our optimizer with an initial `learning_rate` $= 0.001$, and we enhance it with an `ExponentialLR` learning rate scheduler, which is configured with a decay factor of $\gamma = 0.9$.

Priliminary Analysis and Conclusion

We studied the convergence of the model with different N. For $N \in \{100, 1000, 5000\}$, the training records are presented below. In each graph, we marked an approximate region (green) to which they ultimately converge.

From fig. 3.1 we can observe that with the utilization of this type of random sampling set for training, the model's performance does not achieve steady convergence when the sample size is relatively small, as exemplified by a size of 100. It becomes essential to increase the sample size to a sufficiently large scale to ensure stable convergence.

Nevertheless, the ultimate capacity of these models to align with the results obtained from the Monte Carlo method remains significantly inadequate. This observation is supported by the subsequent graph, where we compare the values generated by each method.

The comparison study is structured as follows:

i. After the model has been trained, we choose a set of input $\{(t, x)^i, i = 1, \ldots, N\}$ for both the trained model and the Monte Carlo simulation. In detail,

$$\begin{aligned} t &\in \{0.1, 0.2, \ldots, 0.9\}, \\ x_1 &\in \{-3, 0, 3\}, \\ x_2 &\in \{-3, 0, 3\}, \end{aligned}$$

i.e.,

$$\begin{aligned} \{(t, x)^i, i = 1, \ldots, N\} &= \{0.1, 0.2, \ldots, 0.9\} \times \{-3, 0, 3\}^2, \\ N &= 9 \times 3 \times 3. \end{aligned}$$

ii. For the trained neural network model, we simply turn it to the evaluation mode and then use this set as input to generate the corresponding $\{(u(t, x))^i, i = 1, \ldots, N\}_{\mathtt{NN}}$.

iii. For the Monte Carlo simulation, we use the program build in **Exercise 1.2**, then set `time_step_number` $= 5000$, `sample_size` $= 10000$, and instead of using the S from the theoretical numerical solution to compute control vector, we use $\alpha = (1, 1)^\top$ this time. Denote the solution as $\{(u(t, x))^i, i = 1, \ldots, N\}_{\mathtt{MC}}$.

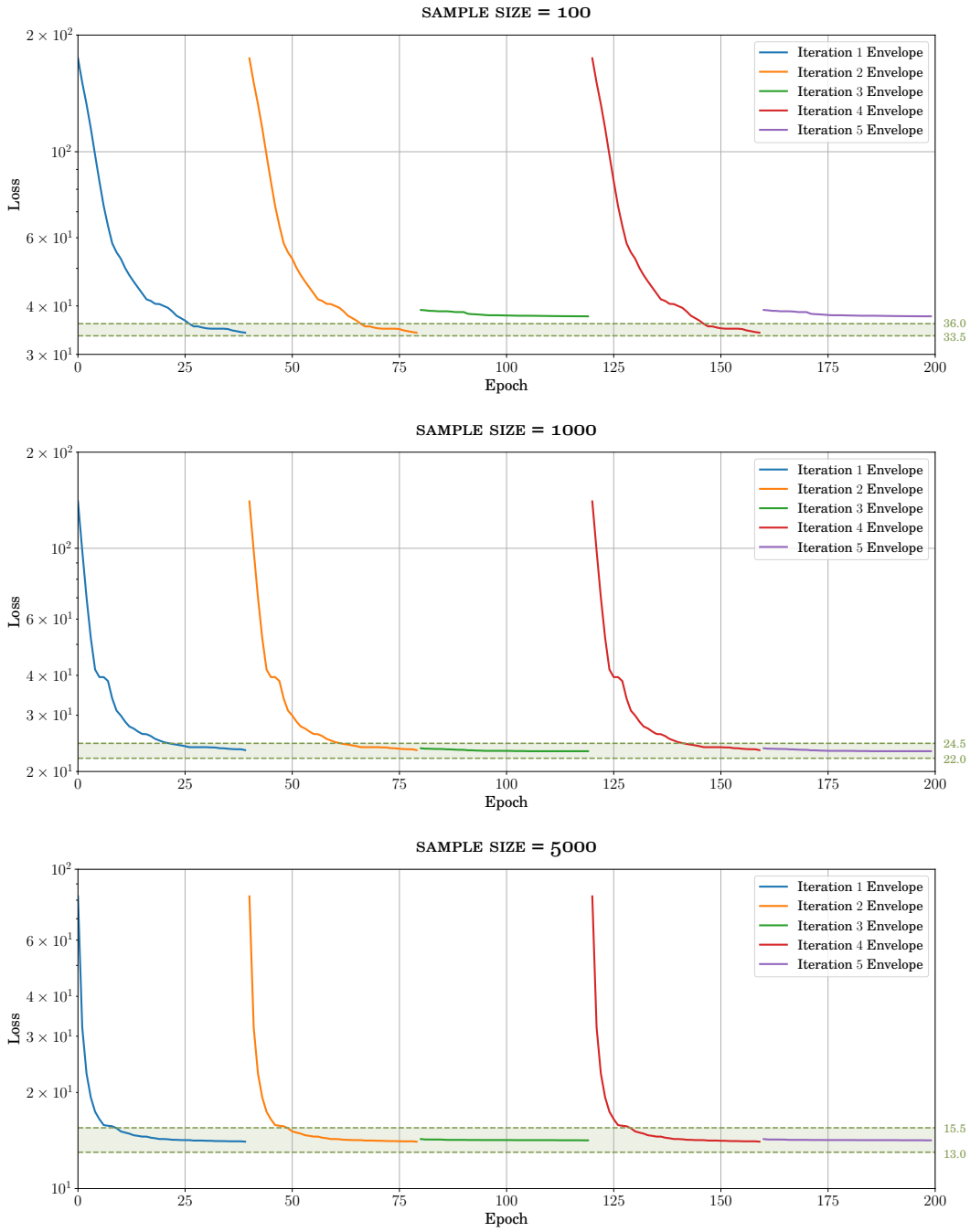


Figure 3.1: Training loss over epochs

iv. After obtaining both sets of results, we plotted them for comparison. Due to the excessively large error, we opted not to plot the error directly. Instead, we represented both $\{(u(t,x))^i, i = 1, \ldots, N\}_{\texttt{NN}}$ and $\{(u(t,x))^i, i = 1, \ldots, N\}_{\texttt{MC}}$ as 1-Dimensional functions of the index i. This approach guarantees a one-to-one correspondence between the two sets of results, facilitating a clearer comparison.

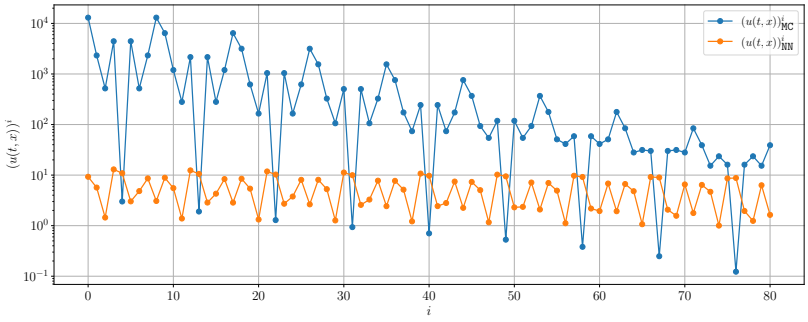


Figure 3.2: Comparison between $(u(t,x))_{\texttt{NN}}$ and $(u(t,x))_{\texttt{MC}}$

Figure 3.2 is the plot of the comparison between the two results, from which we can conclude that

1. The plots revealed a clear periodic pattern, attributed to the indices being organized sequentially according to time and spatial positioning.

2. From a numerical perspective, the fitting accuracy of the Neural Network (NN) model is profoundly unsatisfactory.

Further Research

To enhance the performance of the model employed in the Deep Galerkin Method (DGM), we delved into the relevant literature and identified a particularly pertinent study by Justin Sirignano and Konstantinos Spiliopoulos [1]. In their paper, the authors propose that the model's structure plays a pivotal role in accurately capturing the "sharp turns" exhibited by the function at boundaries (terminal conditions). This aspect is vital not only for fidelity in representation but also for training efficiency.

Consequently, we adopted their research approach by implementing the network structure they proposed in [1, Equation 4.2]. The relevant code implementation is detailed below.

```
class DGMhiddenlayerYYBver(nn.Module):

    # From the original paper of Justin's, presented by Yuebo Yang Apr.9th 2024

    def __init__(self, input_f, output_f, activation = 'tanh'):

        super(DGMhiddenlayerYYBver, self).__init__()

        self.input_f = input_f
        self.output_f = output_f
```

```
11
12      # Params
13
14      # Zl's
15
16      self.Uzl = nn.Parameter(torch.Tensor(output_f, input_f))
17      self.Wzl = nn.Parameter(torch.Tensor(output_f, output_f))
18      self.bzl = nn.Parameter(torch.Tensor(output_f))
19
20      # Gl's
21
22      self.Ugl = nn.Parameter(torch.Tensor(output_f, input_f))
23      self.Wgl = nn.Parameter(torch.Tensor(output_f, output_f))
24      self.bgl = nn.Parameter(torch.Tensor(output_f))
25
26      # Rl's
27
28      self.Url = nn.Parameter(torch.Tensor(output_f, input_f))
29      self.Wrl = nn.Parameter(torch.Tensor(output_f, output_f))
30      self.brl = nn.Parameter(torch.Tensor(output_f))
31
32      # Hl's
33
34      self.Uhl = nn.Parameter(torch.Tensor(output_f, input_f))
35      self.Whl = nn.Parameter(torch.Tensor(output_f, output_f))
36      self.bhl = nn.Parameter(torch.Tensor(output_f))
37
38
39      if activation == 'tanh':
40         self.activation = torch.tanh
41      else:
42         self.activation = None
43
44      self.init_method = 'normal' # or 'uniform'
45
46      self._initialize_params()
47
48   def _initialize_params(self):
49
50      if self.init_method == 'uniform':
51         for param in self.parameters():
52            if param.dim() > 1:
53               init.xavier_uniform_(param)
54            else:
55               init.constant_(param, 0)
56
57      if self.init_method == 'normal':
58         for param in self.parameters():
59            if param.dim() > 1:
60               init.xavier_normal_(param)
61            else:
62               init.constant_(param, 0)
63
64   def forward(self, x, S1, Sl):
65
66      Zl = self.activation(torch.mm(x, self.Uzl.t())+ torch.mm(Sl, self.Wzl.t()) +
          self.bzl)
```

```
        Gl = self.activation(torch.mm(x, self.Ugl.t())+ torch.mm(S1, self.Wgl.t()) +
             self.bgl)

        Rl = self.activation(torch.mm(x, self.Url.t())+ torch.mm(Sl, self.Wrl.t()) +
             self.brl)

        Hl = self.activation(torch.mm(x, self.Uhl.t())+ torch.mm(torch.mul(Sl,Rl),
             self.Whl.t()) + self.bhl)

        Sl_1 = torch.mul((1-Gl),Hl) + torch.mul(Zl,Sl)

        return Sl_1


class DGMNN_YYBver(nn.Module):

    # From the original paper of Justin's, presented by Yuebo Yang Apr.9th 2024

    def __init__(self, init_method = 'uniform'):
        super(DGMNN_YYBver, self).__init__()

        self.nodenum = 50

        self.layer1 = DGMhiddenlayerYYBver(3, self.nodenum)
        self.layer2 = DGMhiddenlayerYYBver(3, self.nodenum)
        self.layer3 = DGMhiddenlayerYYBver(3, self.nodenum)

        self.tanh = nn.Tanh()

        # Params

        # S1's

        self.W1 = nn.Parameter(torch.Tensor(self.nodenum, 3))
        self.b1 = nn.Parameter(torch.Tensor(self.nodenum))

        # Output's

        self.W = nn.Parameter(torch.Tensor(1, self.nodenum))
        self.b = nn.Parameter(torch.Tensor(1))

        self.activation = torch.tanh

        self.init_method = 'normal' # or 'uniform'

        self._initialize_params()

    def _initialize_params(self):

        if self.init_method == 'uniform':
            for param in self.parameters():
                if param.dim() > 1:
                    init.xavier_uniform_(param)
                else:
                    init.constant_(param, 0)

```

```
        if self.init_method == 'normal':
            for param in self.parameters():
                if param.dim() > 1:
                    init.xavier_normal_(param)
                else:
                    init.constant_(param, 0)


    def forward(self, x):

        S_1 = self.activation(torch.mm(x, self.W1.t()) + self.b1)
        # l=1
        S_2 = self.layer1(x,S_1,S_1)
        # l=2
        S_3 = self.layer2(x,S_1,S_2)
        # l=3
        S_4 = self.layer3(x,S_1,S_3)

        output = torch.mm(S_4, self.W.t()) + self.b

        return output
```

However, the authors outlined their training strategy towards the latter part of [1, Section 4.2], revealing a computational demand of 100,000 iterations with 5,000 points per iteration, where each iteration processes batches of 1,000 across GPU nodes. This level of computational investment significantly surpasses what we can feasibly achieve in this coursework. Despite conducting several training runs using this architecture, the improvements were minimal, limited by the computational resources at our disposal.

4 Policy iteration with DGM

Exercise 4.1 PIA with DGM, Linear PDE.

Initialization

To use the Deep Galerkin Method and to solve the same linear PDE listed in **Exercise 3.1**, the class `DGMNN_YYBver` in **Further Research** is introduced.

$$\partial_t u + \tfrac{1}{2}\mathrm{tr}(\sigma\sigma^\top \partial_{xx} u) + (\partial_x u)^\top Hx + (\partial_x u)^\top M\alpha + x^\top Cx + \alpha^\top D\alpha = 0 \quad \text{on } [0,T)\times\mathbb{R}^2,$$
$$u(T,x) = x^\top Rx \quad \text{on } \mathbb{R}^2,$$

where the initial Markov control is $\alpha = (1,1)^\top$.
Also, the H, M, C, D, R, T are initialized as in the previous sections.

Preparation

We use class `value_diff` to compute the later difference.

```
class value_diff:
    def __init__(self, t_b, x_b):
        self.items = []
        self.t_b = t_b
        self.x_b = x_b

    def add_item(self, item):
        self.items.insert(0, item)
        if len(self.items) > 2:
            self.items.pop()

    def calculate_difference(self):
        if len(self.items) == 2:
            v_n = self.items[0](torch.cat((self.t_b.unsqueeze(1),
                self.x_b.squeeze(1)),dim=1)).squeeze()
            v_n1 = self.items[1](torch.cat((self.t_b.unsqueeze(1),
                self.x_b.squeeze(1)),dim=1)).squeeze()
            return (v_n1 - v_n).pow(2).mean()
        else:
            inf1 = float('inf')
            return inf1
```

We include the method to train the model at first part, defining two main function we will use in our iteration called `value_update` and `control_update`. The following is the excerpt.
The total residual is defined above as the sum of equation residual and boundary residual.

```
def value_update(H, M, sigma, alpha_i, C, D, R, T, iter_p):

```

```
    model_DGM = DGMNN_YYBver(1).double()
    optimizer_DGM = torch.optim.Adam(model_DGM.parameters(), lr=0.0001)
    scheduler_DGM = lr_scheduler.ExponentialLR(optimizer_DGM, gamma=0.9)

    ...

              for batch_idx, (_t_data,_x_data) in enumerate(dataloader):
                 optimizer_DGM.zero_grad()
                 t_data = _t_data.clone().requires_grad_(True)
                 x_data = _x_data.clone().requires_grad_(True)
                 if iter_p == 0:
                    alpha = alpha_i
                 else:
                    alpha = alpha_i(torch.cat((t_data.unsqueeze(1),
                        x_data),dim=1)).squeeze()
                 loss = total_residual(model_DGM, t_data, x_data, H, M, sigma, alpha,
                     C, D, R, T)
                 loss.backward()
                 optimizer_DGM.step()
                 total_loss += loss.item()

              avg_loss = total_loss / len(dataloader)
              epoch_losses.append(avg_loss)

    ...

    return model_DGM
```

We will return an updated model of value function called model_DGM.
The Hamiltonian is defined in the following box.

```
def Hamiltonian(model_control, model_value, t, x, H, M, C, D):

    input = torch.cat((t.unsqueeze(1), x),dim=1)

    u = model_value(input)
    a = model_control(input)

    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),
        create_graph=True, retain_graph=True)[0]

    H_i = u_x.unsqueeze(1) @ H @ x.unsqueeze(1).transpose(1,2) + u_x.unsqueeze(1) @ M
        @ a.unsqueeze(1).transpose(1,2) + x.unsqueeze(1) @ C @
        x.unsqueeze(1).transpose(1,2) + a.unsqueeze(1) @ D @
        a.unsqueeze(1).transpose(1,2)

    # The true hamiltonian
    H = H_i.mean()
    return H
```

```
def control_update(H, M, C, D, T, model_value, iter_p):

    control_DGM = DGMNN_YYBver(2).double()
    optimizer_control = torch.optim.Adam(control_DGM.parameters(), lr=0.0001)
    scheduler_control = lr_scheduler.ExponentialLR(optimizer_control, gamma=0.9)

...

          for epoch in range(epochs):

              control_DGM.train()
              total_loss = 0

              for batch_idx, (_t_data,_x_data) in enumerate(dataloader):
                  optimizer_control.zero_grad()
                  t_data = _t_data.clone().requires_grad_(True)
                  x_data = _x_data.clone().requires_grad_(True)
                  loss = Hamiltonian(control_DGM, model_value, t_data, x_data, H, M,
                      C, D)
                  loss.backward()
                  optimizer_control.step()
                  total_loss += loss.item()

              avg_loss = total_loss / len(dataloader)
              epoch_losses.append(avg_loss)

...
    return control_DGM
```

It will return an updated neural network model of Markov control.

Policy Iteration

In the main function, we run policy iteration by calling the functions defined above to approximate and update the value function and Markov controls.
After the initialization section,i.e., making a guess of the control $a_0 = \alpha = (1, 1)^\top$, the `while` loop continues until the difference of v_{n+1} and v_n is large and we ran the iteration no more than 5 times. In this setting, we set the limit to 0.01.

```
    iter_p = 0
    iter_max = 5
    alpha = torch.tensor([[1., 1.]], dtype = Proj_dtype, device = Proj_device)

    t_b, x_b = new_data(T, 10000)

    v_diff = float('inf')
    v_stack = value_diff(t_b, x_b)

    while (v_diff > 0.01) and (iter_p < iter_max) :
        print(f'Current policy iteration is {iter_p}/{iter_max}, and current value
            difference is {v_diff}.')
```

```
        model_value = value_update(H, M, sigma, alpha, C, D, R, T, iter_p)
        alpha = control_update(H, M, C, D, T, model_value, iter_p)
        v_stack.add_item(model_value)
        v_diff = v_stack.calculate_difference()
        iter_p += 1

    print(f'Policy iteration finished. Current iteration is {iter_p}/{iter_max}, and
        current value difference is {v_diff}.')
```

Analysis and Conclusion

The ultimate capacity of these models to align with the results obtained from the Deep Galerkin method remains significantly inadequate. This observation is supported by the subsequent graph, where we compared the values of numerical solution in **Exercise 1.1** and values generated by models.

i. After the model has been trained, we chose the same set of input $\{(t,x)^i, i = 1,\dots,N\}$ for both the iteration and the numerical solution as stated in **Exercise 3.1**.

ii. For the trained *neural network model*, we used them to generate the corresponding $\{(v(t,x))^i, i = 1,\dots,N\}_{\texttt{NN}}$ and $\{(a(t,x))^i, i = 1,\dots,N\}_{\texttt{NN}}$.

iii. For the *numerical* value and control, we used the solver built in **Exercise 1.1**, then set `time_step_number` $= 10000$, `sample_size` $= 10000$, and denote the solution as $\{(v(t,x))^i, i = 1,\dots,N\}_{\texttt{NS}}$ and $\{(a(t,x))^i, i = 1,\dots,N\}_{\texttt{NS}}$.

iv. After obtaining both sets of results, we plotted value and print controls for comparison. Due to the excessively large error, we opted not to plot the error directly. Instead, we represented both $\{(v(t,x))^i, i = 1,\dots,N\}_{\texttt{NN}}$ and $\{(v(t,x))^i, i = 1,\dots,N\}_{\texttt{NS}}$ as 1-Dimensional functions of the index i. This approach guarantees a one-to-one correspondence between the two sets of results, facilitating a clearer comparison.

The comparison study is structured as follows:

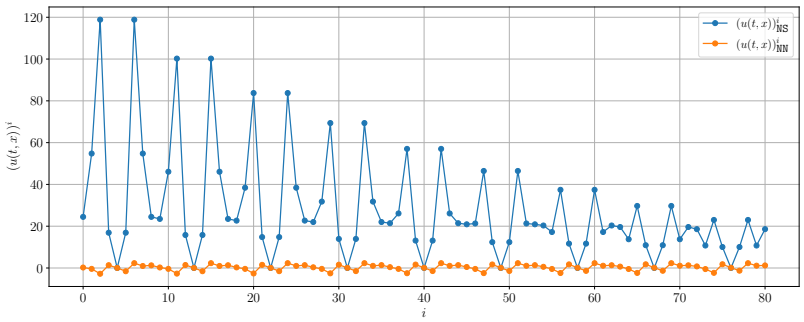


Figure 4.1: Comparison between $(v(t,x))_{\texttt{NN}}$ and $(v(t,x))_{\texttt{NS}}$

Figure 4.1 is the plot of the comparison between the two results, from which we can conclude that

Table 4.1: Comparison between $(a(t,x))_{\mathsf{NS}}$ and $(a(t,x))_{\mathsf{NN}}$

i	$(a(t,x))^i_{\mathsf{NS}}$	$(a(t,x))^i_{\mathsf{NN}}$
1	[6.4607, 9.9608]	[-5.5239e-01, 7.4345e-01]
2	[17.4128, 16.2514]	[1.2431e-01, 1.3142e-01]
3	[28.3648, 22.5420]	[-1.6899e+00, 2.4003e+00]
⋮	⋮	⋮
81	[-4.8322, -7.6592]	[1.7105e+00, 1.0442e+00]

1. From a numerical perspective, the convergence of ultimate value of the policy iteration with Deep Galerkin Method is profoundly unsatisfactory. So does the convergence of control.

2. This is mainly because of our inadequate training data point. This result could be improved by higher computing performance.

References

[1] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.