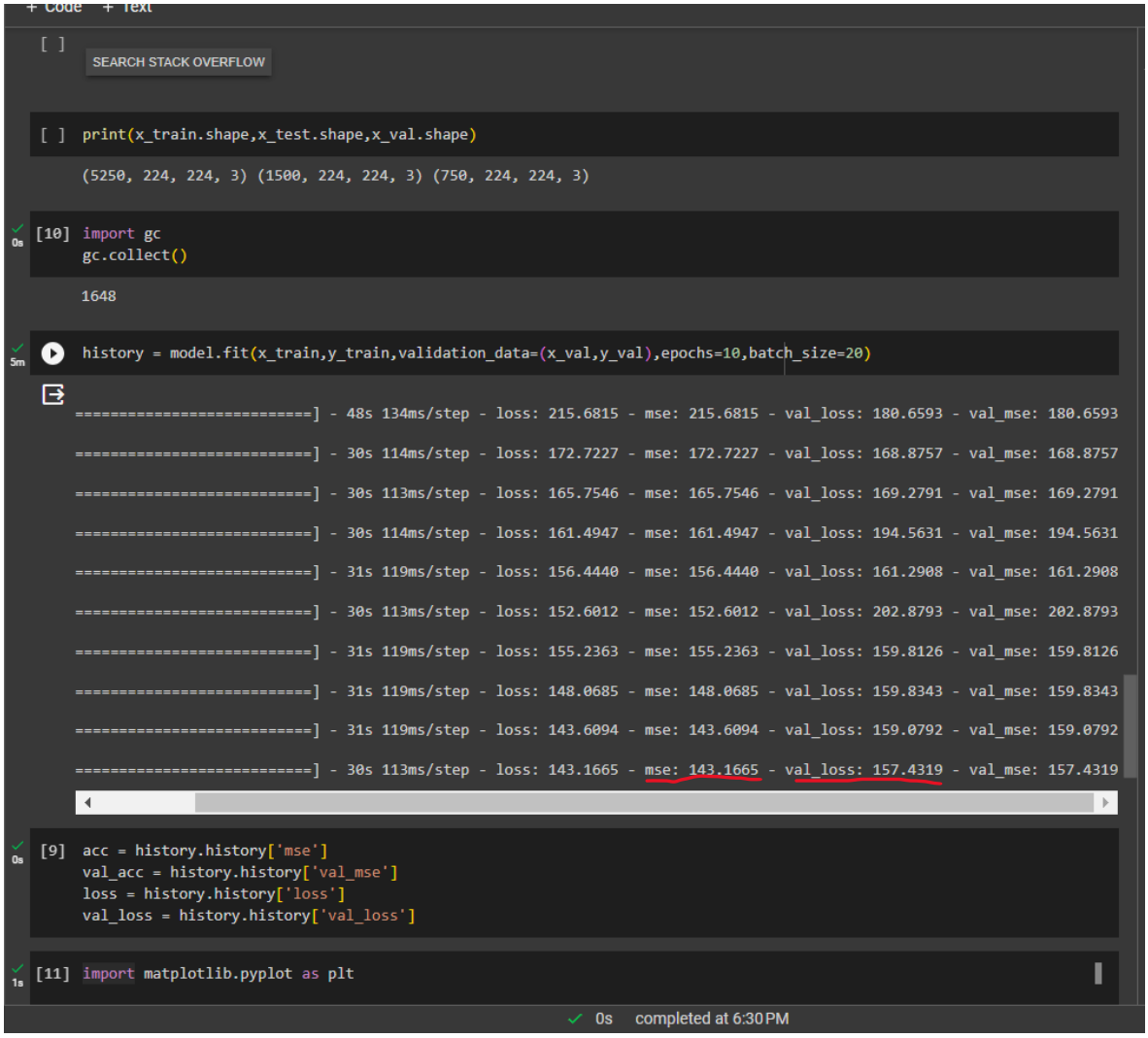
Homework – 5

Students in the Group:

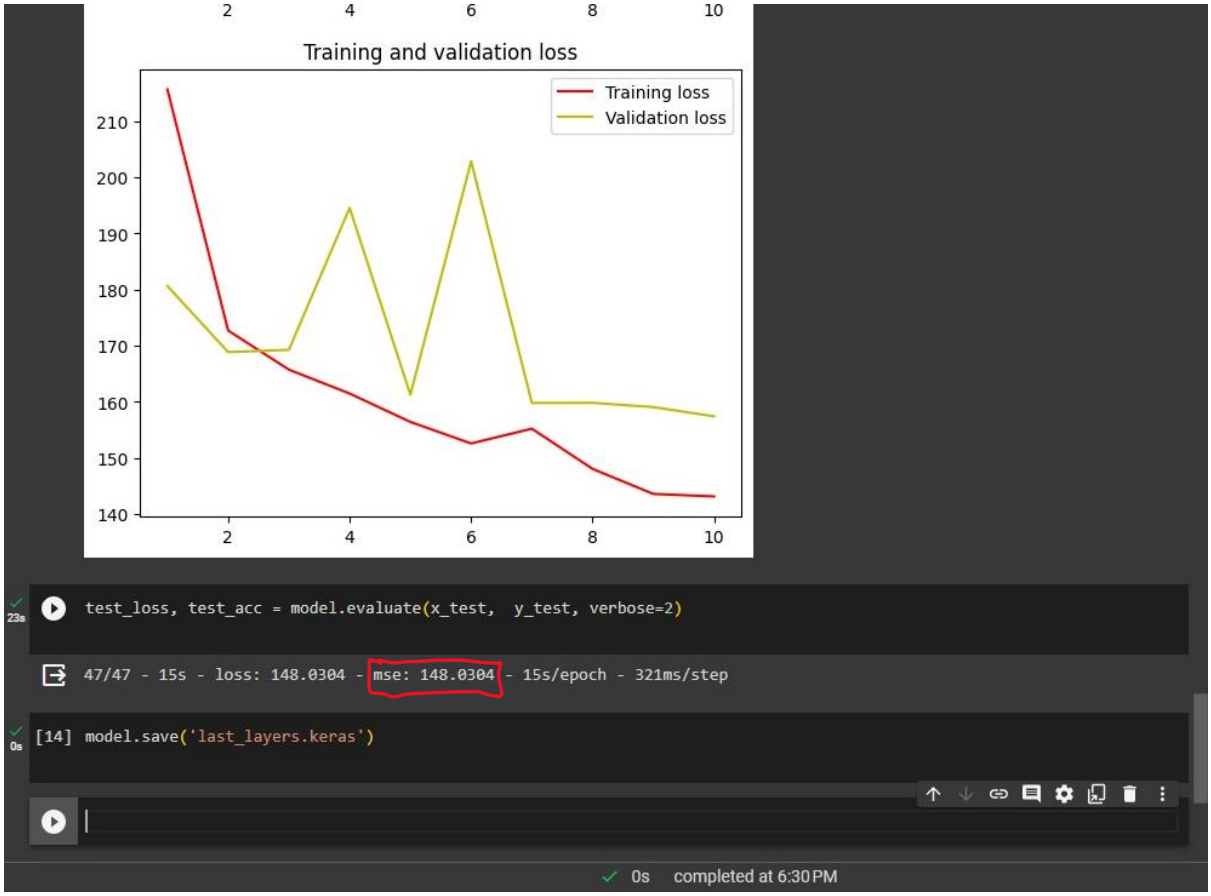
Smit Mitesh Shah (901021266)
Krunal Bhatt (901018587)
Dhiraj Kumar Rouniyar(901011701)

Question 1:

Training the new top layers:



Training and Validation loss graph
The MSE on Test Data is 148.030 (RMSE = 12.166)
MSE on Training Data is 143.166 (RMSE = 11.965)
MSE on Validation Data is 157.431 (RMSE = 12.547)

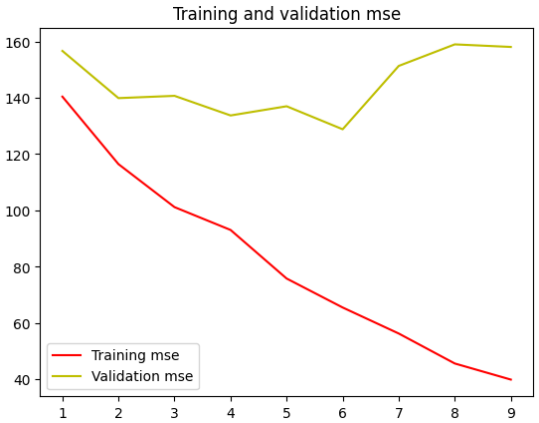


Fine Tuning and Hyper-parameter tuning:

1) Batch_size = 32
   Learning Rate = 1e-5
   Epochs = 10

   MSE on Test Data is 116.110 (RMSE = 10.775)
   MSE on Training Data is 65.540 (RMSE = 8.095)
   MSE on Validation Data is 128.828 (RMSE = 11.350)

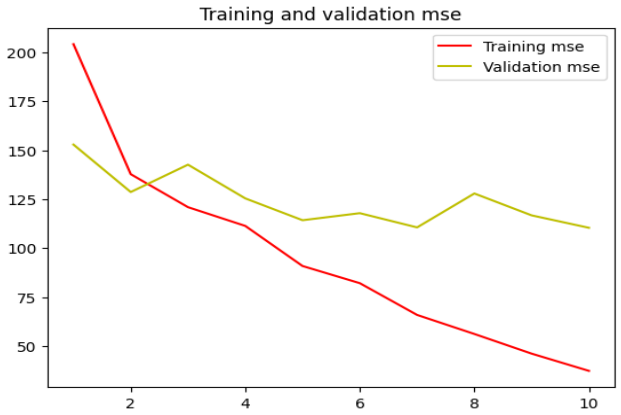
```
+ Code  + Text

[8] history2 = model.fit(x_train,y_train,validation_data=(x_val,y_val),epochs=10,callbacks=[early_stopping])

Epoch 1/10
165/165 [==============================] - 107s 490ms/step - loss: 140.4118 - mse: 140.4118 - val_loss: 156.6449 - val_mse: 156.6449
Epoch 2/10
165/165 [==============================] - 70s 424ms/step - loss: 116.4555 - mse: 116.4555 - val_loss: 139.8870 - val_mse: 139.8870
Epoch 3/10
165/165 [==============================] - 72s 434ms/step - loss: 101.1836 - mse: 101.1836 - val_loss: 140.6972 - val_mse: 140.6972
Epoch 4/10
165/165 [==============================] - 69s 421ms/step - loss: 93.0768 - mse: 93.0768 - val_loss: 133.7318 - val_mse: 133.7318
Epoch 5/10
165/165 [==============================] - 72s 435ms/step - loss: 75.8308 - mse: 75.8308 - val_loss: 136.9973 - val_mse: 136.9973
Epoch 6/10
165/165 [==============================] - 72s 435ms/step - loss: 65.5403 - mse: 65.5403 - val_loss: 128.8287 - val_mse: 128.8287
Epoch 7/10
165/165 [==============================] - 69s 420ms/step - loss: 56.3035 - mse: 56.3035 - val_loss: 151.3186 - val_mse: 151.3186
Epoch 8/10
165/165 [==============================] - 69s 420ms/step - loss: 45.6623 - mse: 45.6623 - val_loss: 158.9703 - val_mse: 158.9703
Epoch 9/10
165/165 [==============================] - 72s 435ms/step - loss: 39.9464 - mse: 39.9464 - val_loss: 158.0556 - val_mse: 158.0556

tf.keras.models.save_model(model,'fine_tuned.h5')

<ipython-input-10-da13306ad555>:1: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is co
  tf.keras.models.save_model(model,'fine_tuned.h5')

[12] test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)

47/47 - 10s - loss: 116.1108 - mse: 116.1108 - 10s/epoch - 220ms/step

[13] acc = history2.history['mse']
     val_acc = history2.history['val_mse']
     loss = history2.history['loss']
     val_loss = history2.history['val_loss']

[14] import matplotlib.pyplot as plt

     epochs = range(1, len(acc) + 1)

     plt.plot(epochs, acc, 'r', label='Training mse')
     plt.plot(epochs, val_acc, 'y', label='Validation mse')
     plt.title('Training and validation mse')
     plt.legend()

0s  completed at 7:54 PM
```



2) Batch_size = 20
   Learning Rate = 1e-4
   Epochs = 10

   MSE on Test Data is 107.742 (RMSE = 10.379)
   MSE on Training Data is 37.328 (RMSE = 6.109)
   MSE on Validation Data is 110.426 (RMSE = 10.508)

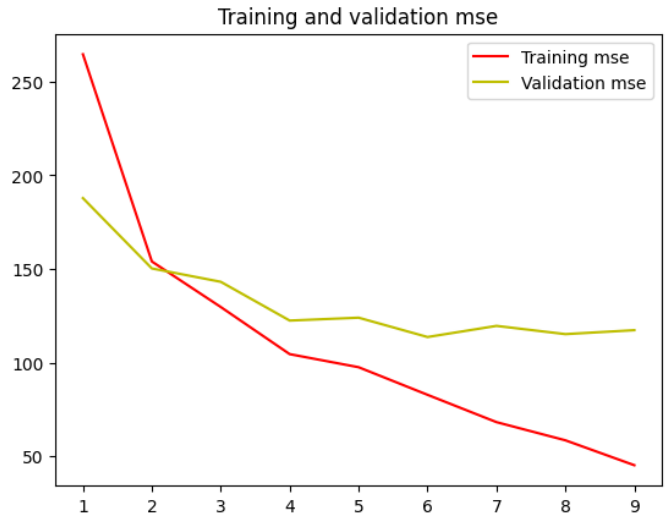
```
[10] model.compile(optimizer=tf.keras.optimizers.Adam(1e-4),
                   loss = tf.keras.losses.MeanSquaredError(),
                   metrics=['mse'])

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
history2 = model.fit(x_train,y_train,validation_data=(x_val,y_val),epochs=10,callbacks=[early_stopping],batch_size=20)

Epoch 1/10
263/263 [==============================] - 112s 345ms/step - loss: 204.1857 - mse: 204.1857 - val_loss: 152.9165 - val_mse: 152.9165
Epoch 2/10
263/263 [==============================] - 83s 315ms/step - loss: 137.8004 - mse: 137.8004 - val_loss: 128.6820 - val_mse: 128.6820
Epoch 3/10
263/263 [==============================] - 84s 319ms/step - loss: 120.9536 - mse: 120.9536 - val_loss: 142.6567 - val_mse: 142.6567
Epoch 4/10
263/263 [==============================] - 84s 321ms/step - loss: 111.3876 - mse: 111.3876 - val_loss: 125.4384 - val_mse: 125.4384
Epoch 5/10
263/263 [==============================] - 84s 321ms/step - loss: 90.9149 - mse: 90.9149 - val_loss: 114.2884 - val_mse: 114.2884
Epoch 6/10
263/263 [==============================] - 84s 319ms/step - loss: 82.1534 - mse: 82.1534 - val_loss: 117.8893 - val_mse: 117.8893
Epoch 7/10
263/263 [==============================] - 84s 319ms/step - loss: 65.9413 - mse: 65.9413 - val_loss: 110.6204 - val_mse: 110.6204
Epoch 8/10
263/263 [==============================] - 84s 319ms/step - loss: 56.2193 - mse: 56.2193 - val_loss: 127.9421 - val_mse: 127.9421
Epoch 9/10
263/263 [==============================] - 84s 318ms/step - loss: 46.1819 - mse: 46.1819 - val_loss: 116.7697 - val_mse: 116.7697
Epoch 10/10
263/263 [==============================] - 83s 314ms/step - loss: 37.3820 - mse: 37.3820 - val_loss: 110.4264 - val_mse: 110.4264

[12] tf.keras.models.save_model(model,'fine_tuned2.h5')

<ipython-input-12-e2c972c85c0d>:1: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is consid
  tf.keras.models.save_model(model,'fine_tuned2.h5')

[15] test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)

47/47 - 15s - loss: 107.7426 - mse: 107.7426 - 15s/epoch - 311ms/step

acc = history2.history['mse']
val_acc = history2.history['val_mse']
loss = history2.history['loss']
val_loss = history2.history['val_loss']

22s completed at 8:38 PM
```



3) Batch_size = 20
   Learning Rate = 1e-4
   Epochs = 20

   MSE on Test Data is 103.709 (RMSE = 10.183)
   MSE on Training Data is 82.808 (RMSE = 9.099)
   MSE on Validation Data is 113.649 (RMSE = 10.660)

```
[6] model.compile(optimizer=tf.keras.optimizers.Adam(1e-4),
                  loss = tf.keras.losses.MeanSquaredError(),
                  metrics=['mse'])

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
history2 = model.fit(x_train,y_train,validation_data=(x_val,y_val),epochs=20,callbacks=[early_stopping],batch_size=20)

Epoch 1/20
263/263 [==============================] - 113s 345ms/step - loss: 264.5435 - mse: 264.5435 - val_loss: 187.7857 - val_mse: 187.7857
Epoch 2/20
263/263 [==============================] - 85s 323ms/step - loss: 153.9708 - mse: 153.9708 - val_loss: 150.2539 - val_mse: 150.2539
Epoch 3/20
263/263 [==============================] - 85s 324ms/step - loss: 129.6646 - mse: 129.6646 - val_loss: 143.1142 - val_mse: 143.1142
Epoch 4/20
263/263 [==============================] - 84s 318ms/step - loss: 104.5193 - mse: 104.5193 - val_loss: 122.4331 - val_mse: 122.4331
Epoch 5/20
263/263 [==============================] - 84s 318ms/step - loss: 97.5338 - mse: 97.5338 - val_loss: 123.9738 - val_mse: 123.9738
Epoch 6/20
263/263 [==============================] - 84s 318ms/step - loss: 82.8085 - mse: 82.8085 - val_loss: 113.6495 - val_mse: 113.6495
Epoch 7/20
263/263 [==============================] - 84s 318ms/step - loss: 68.2309 - mse: 68.2309 - val_loss: 119.6000 - val_mse: 119.6000
Epoch 8/20
263/263 [==============================] - 84s 318ms/step - loss: 58.5283 - mse: 58.5283 - val_loss: 115.1949 - val_mse: 115.1949
Epoch 9/20
263/263 [==============================] - 85s 323ms/step - loss: 45.2397 - mse: 45.2397 - val_loss: 117.3037 - val_mse: 117.3037

[8] tf.keras.models.save_model(model,'fine_tuned3.h5')

<ipython-input-8-18fea50cb010>:1: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is considered
  tf.keras.models.save_model(model,'fine_tuned3.h5')

[11] test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)

47/47 - 15s - loss: 103.7097 - mse: 103.7097 - 15s/epoch - 316ms/step

[12] acc = history2.history['mse']
     val_acc = history2.history['val_mse']
     loss = history2.history['loss']
     val_loss = history2.history['val_loss']

0s  completed at 9:02 PM
```



Hyper-parameters Finalized are:

Batch_size = 20
Learning Rate = 1e-4
Epochs = 10

Homework 5

Team members: Dhiraj Kumar Rouniyar, Smit Shah, Krunal Bhatt

Question 2,

Sub question 1,2,3,4 and 5 (some part) – They have already been included in the python code and for the same .py and .ipynb file with output was generated and is attached as pdf along with this file.

Further,

Sub question - 5

Solution:

(a) -> The model is trained with the test dataset and is mentioned in the code.

(b) ->

i). **MSE -** The performance of the all three models in terms of MSE was calculated through the program and the respective output is generated.

➔ `MSE for Vanilla RNN -`
`Training MSE: 0.0011888565495610237`
`Test MSE: 0.002450475934892893`

➔ `MSE for NN similar to Vanilla RNN, truncated without shared weights –`
`Training MSE: 0.009202002547681332`
`Test MSE: 0.009862154722213745`

➔ `MSE for NN similar to Vanilla RNN, padding without shared weights –`
`Training MSE: 0.00991019792854786`
`Test MSE: 0.011621176265180111`

ii). **Overfitting tendencies -** To evaluate overfitting, we should assess the model's performance on both the training and test datasets. After training, the MSE is calculated on both datasets to compare their performance and generally if the training loss is significantly lower than the test loss, it may indicate overfitting.

Hence, Overfitting tendency:

Vanilla RNN > truncated RNN > Padded RNN

iii). **Convergence Speed -** Convergence speed can be monitored by observing how the training loss changes over epochs. A faster convergence implies that the training loss decreases quickly within a few epochs, while slower convergence may require more training epochs to achieve a low loss.

Looking to the output of the program we can see,

Vanilla RNN > truncated RNN > Padded RNN

(c) ->

1. **Vanilla RNN with Shared Weights:**

Advantages:
Parameter Sharing: The shared weights in a Vanilla RNN allow the model to capture dependencies across different time steps in a sequence effectively.
Less Computational Overhead: Since the weights are shared, the model has fewer parameters, making it computationally efficient and suitable for simple sequential patterns.

Disadvantages:
Limited Modeling of Varying Lengths: Vanilla RNNs are less effective when sequences have varying lengths because they require fixed-length input sequences. Truncation or padding is often necessary to make all sequences the same length.
Difficulty in Capturing Long-term Dependencies: Vanilla RNNs can struggle to capture long-range dependencies in sequences due to the vanishing gradient problem, limiting their ability to model complex patterns.

2. **NN Similar to Vanilla RNN without Shared Weights and Truncated Sequences:**

Advantages:
No Parameter Sharing: Without shared weights, this architecture allows for more flexibility and independence between time steps in the sequence, which can be advantageous for modeling certain types of data.
No Fixed Sequence Length: Sequences can have varying lengths, as they are truncated to the same length before training.

Disadvantages:
Difficulty in Modeling Long Sequences: Like Vanilla RNNs, models without shared weights and truncated sequences may still struggle to model long sequences effectively. Truncation can lead to the loss of information in lengthy sequences, impacting their performance.
Possibility of Information Loss: Truncating sequences can lead to information loss, as important patterns at the end of sequences may be discarded.

3. **NN Similar to Vanilla RNN without Shared Weights and Padded Sequences:**

Advantages:
No Parameter Sharing: Similar to the second approach, there is no parameter sharing, allowing for flexibility.
Preservation of Sequence Length: Padding sequences to a common length preserves the full sequence information, which can be essential for certain tasks.

Disadvantages:
Increased Computational Overhead: Padding sequences to the same length can result in a significant increase in computational overhead, as you are processing unnecessary padding values.
Possibility of Noisy Data: Padded values can introduce noise into the data, which may affect model performance, particularly if the padding values are not well-chosen.

Question 2 - sub question 1, 4 and 5

In [8]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Load training and test data
def loadData():
    X_train = np.load('X_train.npy',allow_pickle=True)
    y_train = np.load('y_train.npy',allow_pickle=True)
    X_test = np.load('X_test.npy',allow_pickle=True)
    y_test = np.load('y_test.npy',allow_pickle=True)

    X_train = [torch.Tensor(x) for x in X_train]  # List of Tensors (SEQ_LEN[i],INPUT_DIM) i=0..NUM_SAMPLES-
    X_test = [torch.Tensor(x) for x in X_test]  # List of Tensors (SEQ_LEN[i],INPUT_DIM)
    y_train = torch.Tensor(y_train) # (NUM_SAMPLES,1)
    y_test = torch.Tensor(y_test) # (NUM_SAMPLES,1)

    return X_train, X_test, y_train, y_test


# Define a Vanilla RNN Layer by hand
class RNNLayer(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNNLayer, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)
        self.activation = nn.Tanh()

    def forward(self, x, hidden):
        combined = self.W_xh(x) + self.W_hh(hidden)
        hidden = self.activation(combined)
        return hidden

# Define a sequence prediction model using the Vanilla RNN
class SequenceModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SequenceModel, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = RNNLayer(input_size, hidden_size)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, seq_lengths):
        batch_size = len(input_seq)
        last_hidden = torch.zeros(batch_size, self.hidden_size).to(device)

        for b in range(batch_size):
            hidden = torch.zeros(1, self.hidden_size).to(device)

            seq_length = seq_lengths[b]
            for t in range(seq_length):
                hidden = self.rnn(input_seq[b][t], hidden)

            # Store the last hidden state in the output tensor
            last_hidden[b] = hidden

        output = self.linear(last_hidden)
        return output

# Define hyperparameters and other settings
input_size = 10
hidden_size = 64
output_size = 1
num_epochs = 10
learning_rate = 0.001
batch_size = 32

# Load data
X_train, X_test, y_train, y_test = loadData()
device = y_train.device

# Create the model using min length input
seq_lengths = [seq.shape[0] for seq in X_train]
```

```
# Create the model
model = SequenceModel(input_size, hidden_size, output_size).to(device)

# Training Loop
def train(model, num_epochs, lr, batch_size, X_train, y_train, seq_lengths):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    for epoch in range(num_epochs):
        for i in range(0, len(X_train), batch_size):
            inputs = X_train[i:i+batch_size]
            targets = y_train[i:i+batch_size]
            lengths = seq_lengths[i:i+batch_size]

            optimizer.zero_grad()
            outputs = model(inputs, lengths)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}')

    return model

# initialize and train Vanilla RNN
trained_model = train(model, num_epochs, learning_rate, batch_size, X_train, y_train, seq_lengths)

# Evaluate the trained model
def evaluate(model, X, y):
    model.eval()
    with torch.no_grad():
        predictions = model(X, [seq.shape[0] for seq in X])
    mse = nn.MSELoss()(predictions, y)
    return mse.item()

train_mse = evaluate(trained_model, X_train, y_train)
test_mse = evaluate(trained_model, X_test, y_test)

print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

```
Epoch 1/10, Loss: 0.02664298191666603
Epoch 2/10, Loss: 0.004362978041172028
Epoch 3/10, Loss: 0.00336488988250494
Epoch 4/10, Loss: 0.002841379027813673
Epoch 5/10, Loss: 0.0024992539547383785
Epoch 6/10, Loss: 0.002202842151746154
Epoch 7/10, Loss: 0.0019441695185378194
Epoch 8/10, Loss: 0.0017336936434730887
Epoch 9/10, Loss: 0.0015591580886393785
Epoch 10/10, Loss: 0.001412449637427926
Training MSE: 0.0011888565495610237
Test MSE: 0.002450475934892893
```

Question 2 - sub question 2, 4 and 5

In [5]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Load training and test data
def loadData():
    X_train = np.load('X_train.npy', allow_pickle=True)
    y_train = np.load('y_train.npy', allow_pickle=True)
    X_test = np.load('X_test.npy', allow_pickle=True)
    y_test = np.load('y_test.npy', allow_pickle=True)

    X_train = [torch.Tensor(x) for x in X_train]
    X_test = [torch.Tensor(x) for x in X_test]
    y_train = torch.Tensor(y_train)
    y_test = torch.Tensor(y_test)

    return X_train, X_test, y_train, y_test

# Defining a simple RNN-based sequence prediction model with fixed time steps
class SequenceModelFixedTimeSteps(nn.Module):
```

```
    def __init__(self, input_dim, hidden_dim, output_dim, seq_length):
        super(SequenceModelFixedTimeSteps, self).__init__()
        self.hidden_dim = hidden_dim
        self.seq_length = seq_length
        self.rnn = nn.RNN(input_size=input_dim, hidden_size=hidden_dim, num_layers=1, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(1, x.size(0), self.hidden_dim).to(x.device)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Use the output from the last time step
        return out

# Define hyperparameters and other settings
input_dim = 10
hidden_dim = 64
output_dim = 1
num_epochs = 10
learning_rate = 0.001
batch_size = 32

# Load data
X_train, X_test, y_train, y_test = loadData()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Determine the minimum sequence length
min_seq_length = min(len(seq) for seq in X_train)

# Trim or pad sequences to the minimum length
X_train = [seq[:min_seq_length] for seq in X_train]
X_test = [seq[:min_seq_length] for seq in X_test]

# Convert the data to tensors and create dataloaders
X_train = torch.stack(X_train)
X_test = torch.stack(X_test)
y_train = y_train.view(-1, 1)  # Ensure the correct shape for the labels

# Initialize the model
model = SequenceModelFixedTimeSteps(input_dim, hidden_dim, output_dim, min_seq_length).to(device)

# Training loop
def train(model, num_epochs, lr, batch_size, X_train, y_train):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        for i in range(0, len(X_train), batch_size):
            inputs = X_train[i:i+batch_size]
            targets = y_train[i:i+batch_size]


            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item()}')
    return model

# initialize and train Sequential NN fixing #timesteps to the minimum sequence length
trained_model = train(model, num_epochs, learning_rate, batch_size, X_train, y_train)

# -----  Question 2_Sub question 5 -----#

# Evaluate the trained model
def evaluate(model, X, y):
    model.eval()
    with torch.no_grad():
        predictions = model(X)
    mse = nn.MSELoss()(predictions, y)
    return mse.item()

train_mse = evaluate(trained_model, X_train, y_train)
test_mse = evaluate(trained_model, X_test, y_test)

print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

```
Epoch [1/10], Loss: 0.009028473868966103
Epoch [2/10], Loss: 0.009373912587761879
Epoch [3/10], Loss: 0.008757087402045727
Epoch [4/10], Loss: 0.008849642239511013
Epoch [5/10], Loss: 0.008843314833939075
Epoch [6/10], Loss: 0.008863409049808979
Epoch [7/10], Loss: 0.00888094399124384
Epoch [8/10], Loss: 0.008892207406461239
Epoch [9/10], Loss: 0.008898316882550716
Epoch [10/10], Loss: 0.008899688720703125
Training MSE: 0.009202002547681332
Test MSE: 0.009862154722213745
```

Question 2 - sub question 3, 4 and 5

In [4]:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Load training and test data
def loadData():
    X_train = np.load('X_train.npy',allow_pickle=True)
    y_train = np.load('y_train.npy',allow_pickle=True)
    X_test = np.load('X_test.npy',allow_pickle=True)
    y_test = np.load('y_test.npy',allow_pickle=True)

    X_train = [torch.Tensor(x) for x in X_train]  # List of Tensors (SEQ_LEN[i],INPUT_DIM) i=0..NUM_SAMPLES-
    X_test = [torch.Tensor(x) for x in X_test]  # List of Tensors (SEQ_LEN[i],INPUT_DIM)
    y_train = torch.Tensor(y_train) # (NUM_SAMPLES,1)
    y_test = torch.Tensor(y_test) # (NUM_SAMPLES,1)

    return X_train, X_test, y_train, y_test


# Define a Vanilla RNN Layer by hand
class RNNLayer(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNNLayer, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)
        self.activation = nn.Tanh()

    def forward(self, x, hidden):
        hidden = self.activation(self.W_xh(x) + self.W_hh(hidden))
        return hidden

# Define a sequence prediction model for variable length sequences, NO SHARED WEIGHTS
class SequenceModelVarLen(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SequenceModelVarLen, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layer = RNNLayer(input_size, hidden_size)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, seq_lengths):
        batch_size = input_seq.size(0)
        max_seq_length = input_seq.size(1)

        # Initialize hidden state for each sequence in the batch
        hidden = torch.zeros(batch_size, self.hidden_size).to(input_seq.device)

        for t in range(max_seq_length):
            # Apply RNN layer at each time step
            hidden = self.rnn_layer(input_seq[:, t, :], hidden)
        output = self.linear(hidden)
        return output

# Define hyperparameters and other settings
input_size = 10
hidden_size = 64
output_size = 1
num_epochs = 10
learning_rate = 0.001
batch_size = 32
```

```python
# Load data
X_train, X_test, y_train, y_test = loadData()
device = X_train[0].device

# Create the model
model = SequenceModelVarLen(input_size, hidden_size, output_size).to(device)

# Training Loop
def train(model, num_epochs, lr, batch_size, X_train, y_train):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    for epoch in range(num_epochs):
        for i in range(0, len(X_train), batch_size):
            inputs = X_train[i:i+batch_size]
            targets = y_train[i:i+batch_size]

            # Pad sequences to the same length (maximum sequence length in the batch)
            max_seq_length = max([seq.size(0) for seq in inputs])
            padded_inputs = torch.zeros(batch_size, max_seq_length, input_size).to(device)
            for j, seq in enumerate(inputs):
                seq_len = seq.size(0)
                padded_inputs[j, :seq_len, :] = seq

            optimizer.zero_grad()
            outputs = model(padded_inputs, None)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item()}')
    return model

# initialize and train Sequential NN fixing #timesteps to the maximum sequence length
trained_model = train(model, num_epochs, learning_rate, batch_size, X_train, y_train)

# -----  Question 2_Sub question 5 -----#

# Evaluate the trained model
def evaluate(model, X, y):
    model.eval()
    with torch.no_grad():
        mse_sum = 0.0
        for i in range(len(X)):
            max_seq_length = X[i].size(0)
            outputs = model(X[i].unsqueeze(0), [max_seq_length])
            mse_sum += nn.MSELoss()(outputs, y[i:i+1])  # Calculate MSE for each sequence individually
        mse = mse_sum / len(X)
    return mse.item()

train_mse = evaluate(trained_model, X_train, y_train)
test_mse = evaluate(trained_model, X_test, y_test)

print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

```
Epoch [1/10], Loss: 0.014521529898047447
Epoch [2/10], Loss: 0.007491786032915115
Epoch [3/10], Loss: 0.006498521659523249
Epoch [4/10], Loss: 0.006131777539849281
Epoch [5/10], Loss: 0.005994045175611973
Epoch [6/10], Loss: 0.006073987111449242
Epoch [7/10], Loss: 0.0072938185185194016
Epoch [8/10], Loss: 0.007871345616877079
Epoch [9/10], Loss: 0.007722165901213884
Epoch [10/10], Loss: 0.007495713885873556
Training MSE: 0.00991019792854786
Test MSE: 0.011621176265180111
```

In [ ]: