Question 4. (40 points): Binary Addition

In this question, you will be implementing a recurrent neural network which operates binary addition. The inputs are two arbitrary binary sequences, starting with the least significant binary digit. Two sequences are padded by zeros (at least one zero) at the end, two have the same length. At each time step, the recurrent neural network takes in two binary digits from both inputs, and outputs a result digit. Example:

- Equation: 10110100 + 10111 = 11001011
- Input 1: 0,0,1,0,1,1,0,1,0
- Input 2: 1,1,1,0,1,0,0,0,0
- Output: 1,1,0,1,0,0,1,1,0 Loss function: as the output at each time step is binary (0 or 1), you can apply Binary Cross Entropy to train the model.

What to report:

- A plot of training loss values after every epoch for each part.
- Show 5 examples of deployment procedure. Every example includes two input binary sequences and an output sequence.

(20 points) Part 1:

You are expected to write a complete system that trains the RNN model on this task. The system expects you to include the following modules:

• A pytorch Dataset, that has a getitem () function, which, for each call, samples two binary sequences of arbitrary length and at most 16 digits, and the exact solution when we add these two binary numbers. The binary sequences and the solution are pre-processed so that they follow least-to-most-significance order, and is padded with at least one zero so that two input sequences have the same length (as shown in the above example).

• A pytorch Dataloader to process Dataset, and creates mini-batches of samples, every sample as a pair of binary sequences, to train the model. Batch size is set to 128 samples, and an epoch is set to 100 iterations on Dataloader.

• A pytorch RNN model that has two input units, one output units, a hidden layer of 10 perceptrons, and use ReLU activation function.

• A training procedure that trains the model through 100 epochs. During training process, save the best performed model. The performance of the model is represented as the average loss of the model through an epoch the model through an epoch.

• A deployment procedure. In this procedure, you will load the best performed model from training procedure. Then, the system takes as input two arbitrary integers, and pre-processes them into binary sequences with the order of least-to-most-significant-digits, and feeds them to the model to obtain the result. Also, you are required to program a checking function to check how many percent of the predicted digits matched with the exact solution.

```
%%capture
%%bash
pip install torchmetrics
```

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset
import torchvision.transforms as transforms
import torch.autograd as autograd
from torch.utils.data import DataLoader
from torchmetrics.classification import BinaryAccuracy
import copy
```

```
print('Pytorch version: ', torch.__version__)
print('GPU availability: ', torch.cuda.is_available())
```

```
Pytorch version:  2.1.0+cu118
GPU availability:  True
```

```
# For visualization:
class BinaryDataset(Dataset):
    def __init__(self, size=100, num_bits=16, zfills=17,verbose=False,base_seed=1871):
        '''
        Inputs:
            `size`: The size of dataset.
            `num_bits`: Integer numbers are sampled so that they are represented by at r
            `zfills`: Fill the sampled numbers to have a fixed length of `zfills`.
        '''
        self.num_bits = num_bits
        self.size = size
        self.verbose=verbose
        self.base_seed=base_seed

```

```
        assert zfills > num_bits
        self.zfills = zfills

    def __len__(self):
        return self.size

    def _get_bin_sequence(self,index):
      np.random.seed(self.base_seed+index)
      a = np.random.randint(0, 2**self.num_bits)
      b = np.random.randint(0, 2**self.num_bits)
      c = a + b

      # Convert to binary string, then invert it
      a_bin = format(a, 'b').zfill(self.zfills)
      b_bin = format(b, 'b').zfill(self.zfills)
      c_bin = format(c, 'b').zfill(self.zfills)

      a_rev=a_bin[::-1]
      b_rev=b_bin[::-1]
      c_rev=c_bin[::-1]
      return [a, b, c], [a_bin, b_bin, c_bin], [[int(i) for i in a_rev],[int(i) for i in


    def __getitem__(self, index):
      s, s_bin, s_rev = self._get_bin_sequence(i)
      x=torch.from_numpy(np.array([s_rev[0],s_rev[1]])).T
      y=torch.from_numpy(np.array(s_rev[2])).unsqueeze(0).T
      if self.verbose:
        print(s)
        print(s_bin)
        print(s_rev)

      return x.type(torch.FloatTensor), y.type(torch.FloatTensor)
```

```
d = BinaryDataset(num_bits=16, zfills=17)
for i in range(5):
    x,y = d[i]
    print(x)
    print(x.shape)
    print(y)
    print(y.shape)
```

```
tensor([[0., 0.],
        [1., 1.],
        [0., 1.],
        [0., 0.],
        [0., 1.],
        [1., 1.],
        [0., 0.],
        [0., 1.],
        [0., 0.],
```

```
        [1., 0.],
        [0., 0.],
        [1., 0.],
        [1., 1.],
        [1., 1.],
        [1., 0.],
        [1., 1.],
        [0., 0.]])
torch.Size([17, 2])
tensor([[0.],
        [0.],
        [0.],
        [1.],
        [1.],
        [0.],
        [1.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.]])
torch.Size([17, 1])
tensor([[1., 1.],
        [0., 1.],
        [0., 0.],
        [0., 1.],
        [1., 1.],
        [0., 1.],
        [0., 0.],
        [0., 1.],
        [1., 0.],
        [0., 0.],
        [1., 0.],
        [0., 0.],
        [0., 0.],
        [1., 0.],
        [0., 1.],
        [1., 1.],
        [0., 0.]])
torch.Size([17, 2])
tensor([[0.],
        [0.],
        [1.],
        [1.],
```

```
train_dataloader = DataLoader(BinaryDataset(num_bits=16, zfills=17,size=128*100,base_see
test_dataloader = DataLoader(BinaryDataset(num_bits=16, zfills=17,size=128*100,base_seed
```

```
class Binary_Adder1(nn.Module):
```

```
class Binary_Adder1(nn.Module):
  def __init__(self, input_size, hidden_size, sequence_length):
    super(Binary_Adder1, self).__init__()
    self.input_size=input_size
    self.hidden_size=hidden_size
    self.output_size=sequence_length
    self.lstm=nn.LSTM(input_size, hidden_size,batch_first=True)
    self.activation=nn.ReLU()
    self.out_layer=nn.Linear(sequence_length*hidden_size, sequence_length)
    self.out_activation=nn.Sigmoid()

  def forward(self, x):
    #x: [BxLx2]
    #y: [BxLx1]
    lstm_out,_ =self.lstm(x)
    #out: [BxLxh]
    B,L,H  = lstm_out.size(0),lstm_out.size(1) , lstm_out.size(2)
    lstm_out = lstm_out.contiguous()
    lstm_out = lstm_out.view(B,L*H)
    x=self.activation(lstm_out)
    x=self.out_layer(x)
    out=self.out_activation(x)
    return out.unsqueeze(-1)
    # return x
  def predict(self, x):
    y_hat=self.forward(x)
    return y_hat.type(torch.IntTensor)
```

```
class Binary_Trainer():
  def __init__(self,
               model,
               train_loader,
               test_loader,
               lr=0.001):
    self.model=model
    self.train_loader=train_loader
    self.test_loader=test_loader
    self.optimizer=optim.Adam(self.model.parameters(),lr=lr)
    self.loss_fn=nn.BCELoss()
    self.accuracy=BinaryAccuracy()

  def _train_step(self,x,y):
    # self.model.zero_grad()
    y_hat = self.model(x)
    loss=self.loss_fn(y_hat,y)
    loss.backward()
    loss_batch=loss.item()
    # self.optimizer.zero_grad()
    self.optimizer.step()
    return loss_batch
```

```
  def _train_epoch(self):
    total_loss=0
    for batch in self.train_loader:
      x,y=batch
      loss_batch=self._train_step(x,y)
      total_loss+=loss_batch
    avg_loss=total_loss/len(self.train_loader.dataset)
    return avg_loss

  def _test_step(self,x,y):
    with torch.no_grad():
      y_hat = self.model(x)
      loss=self.loss_fn(y_hat,y)
      return loss.item()

  def _test_epoch(self):
    total_loss=0
    for batch in self.test_loader:
      x,y=batch
      loss_batch=self._test_step(x,y)
      total_loss+=loss_batch
    avg_loss=total_loss/len(self.test_loader.dataset)
    return avg_loss

  def _epoch(self):
    avg_train_loss=self._train_epoch()
    self.train_loss+=[avg_train_loss]

    avg_test_loss=self._test_epoch()
    self.test_loss+=[avg_test_loss]
    if avg_train_loss<=self.best_loss:
      self.best_test_loss=avg_test_loss
      self.best_model=copy.deepcopy(self.model)


  def train(self,epochs=100):
    self.train_loss=[]
    self.test_loss=[]
    self.best_model=None
    self.best_loss=np.Inf
    for e in range(epochs):
      self._epoch()
      torch.save(self.model.state_dict(), "model_weight_"+str(e)+"_.pt")

  def plot_loss(self):
    fig, ax = plt.subplots()
    ax.plot(self.train_loss,color="blue")
    ax.plot(self.test_loss,color="orange")

  def _best_accuracy_eval(self):
```

```
    with torch.no_grad():
      self.accuracy.reset()
      for batch in self.train_loader:
        x,y=batch
        y_hat=self.best_model.predict(x)
        self.accuracy.update(y_hat,y)
      acc_train=self.accuracy.compute()
      self.accuracy.reset()
      for batch in self.test_loader:
        x,y=batch
        y_hat=self.best_model.predict(x)
        self.accuracy.update(y_hat,y)
      acc_test=self.accuracy.compute()
    return acc_train.item(),acc_test.item()
```

```
model=Binary_Adder1(2,10,17)
trainer=Binary_Trainer(model=model,train_loader=train_dataloader,test_loader=test_datal
```

```
%%time
trainer.train(epochs=100)
```

```
CPU times: user 11min 41s, sys: 685 ms, total: 11min 42s
Wall time: 2min 55s
```

```
trainer.plot_loss()
```

```
%%time
trainer._best_accuracy_eval()
```

```
CPU times: user 6.29 s, sys: 6.99 ms, total: 6.3 s
Wall time: 1.58 s
(1.0, 0.47058823704719543)
```

```
model=Binary_Adder1(2,10,17)
model.load_state_dict(torch.load("model_weight_98_.pt"))
input1 = torch.randint(2,(1,17,2))

print(input)
output1 = model(input1.float())
print("king",output1)
input2 = torch.randint(2,(1,17,2))

print(input)
output2 = model(input2.float())

input3 = torch.randint(2,(1,17,2))

print(input)
output3 = model(input3.float())

input4 = torch.randint(2,(1,17,2))

print(input)
output4 = model(input4.float())

input5 = torch.randint(2,(1,17,2))

print(input)
output5 = model(input5.float())

```

```
tensor([[[0, 1],
         [1, 1],
         [1, 1],
         [0, 1],
         [1, 1],
         [1, 0],
         [0, 0],
         [0, 0],
         [0, 0],
         [0, 1],
         [1, 1],
         [1, 1],
         [1, 1],
         [0, 0],
         [1, 1],
         [1, 1],
```

```
         [0, 1]]])
king tensor([[[1.],
         [1.],
         [0.],
         [1.],
         [1.],
         [0.],
         [1.],
         [0.],
         [1.],
         [1.],
         [0.],
         [1.],
         [0.],
         [0.],
         [0.],
         [1.],
         [1.]]], grad_fn=<UnsqueezeBackward0>)
tensor([[[0, 1],
         [1, 1],
         [1, 1],
         [0, 1],
         [1, 1],
         [1, 0],
         [0, 0],
         [0, 0],
         [0, 0],
         [0, 1],
         [1, 1],
         [1, 1],
         [1, 1],
         [0, 0],
         [1, 1],
         [1, 1],
         [0, 1]]])
tensor([[[0, 1],
         [1, 1],
         [1, 1],
         [0, 1],
         [1, 1],
         [1, 0],
         [0, 0],
```

(20 points) Part 2:

In the above part, we just tried the fourth scheme of RNN as shown in Fig. 1, which is a straightforward way to solve the problem of binary addition, just like how we learned to perform such computation. However, RNNs also have the ability of memorizing inputs. In this part, we will do an investigation on such ability of RNN, using the third RNN scheme in Fig. 1. To do that, you are required to:

• Modify the Dataset class to return all the input binary sequences and the result binary sequence in the typical most to least significant digit order sequence in the typical most-to-least-significant-digit order.

• The new RNN model will be fed by every pair of digits from the input sequences without producing any output. After processing through all input digits, it starts predicting the one-by-one digit of the output.

• As the RNN model now will need to memorize much more information, it will need a bigger hidden layer, try increasing hidden layer to 100 perceptrons.

• Keeping the training and deployment procedure as in Part 1.

```
class Binary_Adder2(nn.Module):
  def __init__(self, input_size, hidden_size, sequence_length):
    super(Binary_Adder2, self).__init__()
    self.input_size=input_size
    self.hidden_size=hidden_size
    self.output_size=sequence_length
    self.lstm=nn.LSTM(input_size, hidden_size,batch_first=True)
    self.activation=nn.ReLU()
    self.out_layer=nn.Linear(hidden_size, sequence_length)
    self.out_activation=nn.Sigmoid()

  def forward(self, x):
    #x: [BxLx2]
    #y: [BxLx1]
    lstm_out, (hidden,cell) =self.lstm(x)
    #out: [BxLxh]
    #hidden: [1xBxH]
    B,L,H  = hidden.size(0), hidden.size(1) , hidden.size(2)
    hidden = hidden.contiguous()
    hidden = hidden.squeeze(0)
    x=self.activation(hidden)
    x=self.out_layer(x)
    out=self.out_activation(x)
    return out.unsqueeze(-1)
  def predict(self, x):
    y_hat=self.forward(x)
    return y_hat.type(torch.IntTensor)
```

```
model2=Binary_Adder2(input_size=2,hidden_size=100,sequence_length=17)
trainer2=Binary_Trainer(model=model2,train_loader=train_dataloader,test_loader=test_data
```

```
%%time
trainer2.train(epochs=100)
```

```
CPU times: user 17min 6s, sys: 1.17 s, total: 17min 7s
Wall time: 4min 16s
```

```
trainer2.plot_loss()
```

```
%%time
trainer2._best_accuracy_eval()
```

```
CPU times: user 8.01 s, sys: 15 ms, total: 8.02 s
Wall time: 2.01 s
(1.0, 0.470588237704719543)
```

```
model=Binary_Adder2(2,100,17)
model.load_state_dict(torch.load("model_weight_99_.pt"))
input1 = torch.randint(2,(1,17,2))

print(input)
output1 = model(input1.float())
print("king",output1)
input2 = torch.randint(2,(1,17,2))

print(input)
output2 = model(input2.float())

input3 = torch.randint(2,(1,17,2))

print(input)
output3 = model(input3.float())

input4 = torch.randint(2,(1,17,2))
```

```python
input4 = torch.randint(2,(1,17,2))

print(input)
output4 = model(input4.float())

input5 = torch.randint(2,(1,17,2))

print(input)
output5 = model(input5.float())

```

```
tensor([[[0, 1],
         [1, 1],
         [1, 1],
         [0, 1],
         [1, 1],
         [1, 0],
         [0, 0],
         [0, 0],
         [0, 0],
         [0, 1],
         [1, 1],
         [1, 1],
         [1, 1],
         [0, 0],
         [1, 1],
         [1, 1],
         [0, 1]]])
king tensor([[[1.],
         [1.],
         [0.],
         [1.],
         [1.],
         [0.],
         [1.],
         [0.],
         [1.],
         [1.],
         [0.],
         [1.],
         [0.],
         [0.],
         [0.],
         [1.],
         [1.]]], grad_fn=<UnsqueezeBackward0>)
tensor([[[0, 1],
         [1, 1],
         [1, 1],
         [0, 1],
         [1, 1],
         [1, 0],
         [0, 0],
         [0, 0],
         [0, 0],
         [0, 1],
```

```
                  [1, 1],
                  [1, 1],
                  [1, 1],
                  [0, 0],
                  [1, 1],
                  [1, 1],
                  [0, 1]]])
         tensor([[[0, 1],
                  [1, 1],
                  [1, 1],
                  [0, 1],
                  [1, 1],
                  [1, 0],
                  [0, 0],
```