作業二 報告

資訊 114　H44091196 洪茂菘

1. Data:

- ✓ Load the data and split them into train and validation (10 pts). You can use pandas, Dataset and Dataloader.

```
dataset = Dataset(df['src_char_id_list'])
train_data,val_data=torch.utils.data.random_split(dataset,[int(len(dataset) *0.9),int(len(dataset)*0.1)])
```

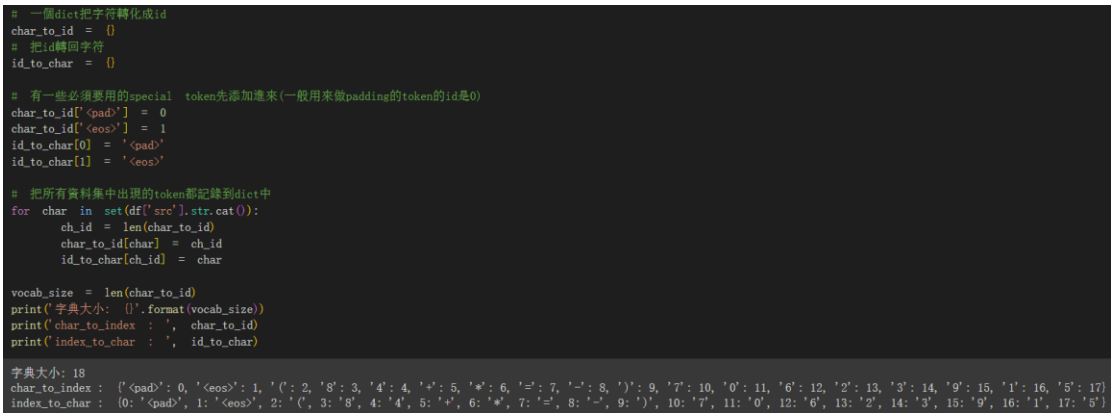
如上圖所示，我使用 Dataset 將資料做讀入，並且將其依 9:1 切分為 train 和 validation。

- ✓ Generate your own version of data (10 pts), At least three two-digit addition and subtraction problems, and if you'd like, you can also try multiplication and division

```
#隨機產生資料(最少三個數運算)
def generate_data():
        first_num = np.random.randint(low=0,high=20)
        second_num = np.random.randint(low=0,high=20)
        third_num = np.random.randint(low=0,high=20)
        add = np.random.randint(low=0, high=4) #第一次運算
        add2 = np.random.randint(low=0, high=4) #第二次運算

        match add:
            case 0:
                expression = str(first_num) + '+' + str(second_num)
                result = first_num+second_num
            case 1:
                expression = str(first_num) + '-' + str(second_num)
                result = first_num-second_num
            case 2:
                expression = "("+str(first_num) + '+' + str(second_num)+")"
                result = first_num+second_num
            case 3:
                expression = "("+str(first_num) + '-' + str(second_num)+")"
                result = first_num-second_num
            case 4:
                expression = str(first_num) + '*' + str(second_num)
                result = first_num*second_num


        if add2 == 0: #加法
                expression = expression + '+' + str(third_num)+ "="
                result = result + third_num
        elif add2==1: #減法
                expression = expression + '-' + str(third_num)+ "="
                result = result - third_num
        else : #乘法
                expression = expression+ '*' + str(third_num) + "="
                match add:
                    case 0:
                        result = first_num+(second_num*third_num)
                    case 1:
                        result = first_num-(second_num*third_num)
                    case 2:
                        result = (first_num+second_num)*third_num
                    case 3:
                        result = (first_num-second_num)*third_num
                    case 4:
                        result = result * third_num
        return expression, result
```

我設計了一個隨機產生資料的 function，其產生的結果分成 expression 和 result，中間用逗號隔開，例如: ('(17+14)-12=', 19)。

- ✓ Tokenize the text (10 pts). You can design your own tokenizer or use any API (if available).



```
# 一個dict把字符轉化成id
char_to_id = {}
# 把id轉回字符
id_to_char = {}

# 有一些必須要用的special token先添加進來(一般用來做padding的token的id是0)
char_to_id['<pad>'] = 0
char_to_id['<eos>'] = 1
id_to_char[0] = '<pad>'
id_to_char[1] = '<eos>'

# 把所有資料集中出現的token都記錄到dict中
for char in set(df['src'].str.cat()):
    ch_id = len(char_to_id)
    char_to_id[char] = ch_id
    id_to_char[ch_id] = char

vocab_size = len(char_to_id)
print('字典大小: {}'.format(vocab_size))
print('char_to_index : ', char_to_id)
print('index_to_char : ', id_to_char)
```

```
字典大小: 18
char_to_index :  {'<pad>': 0, '<eos>': 1, '(': 2, '8': 3, '4': 4, '+': 5, '*': 6, '=': 7, '-': 8, ')': 9, '7': 10, '0': 11, '6': 12, '2': 13, '3': 14, '9': 15, '1': 16, '5': 17}
index_to_char :  {0: '<pad>', 1: '<eos>', 2: '(', 3: '8', 4: '4', 5: '+', 6: '*', 7: '=', 8: '-', 9: ')', 10: '7', 11: '0', 12: '6', 13: '2', 14: '3', 15: '9', 16: '1', 17: '5'}
```

```
# 把資料集的所有資料都變成id
df['src_char_id_list'] = df['src'].apply(lambda text: [char_to_id[char] for char in list(text)] ) + df['tgt'].apply(lambda text: [char_to_id[char] for char in list(text)])

df[['src','tgt','src_char_id_list']].head()
```

	src	tgt	src_char_id_list
0	19+2*15=	49	[16, 15, 5, 13, 6, 16, 17, 7, 4, 15, 1]
1	(3-1)-17=	-15	[2, 14, 8, 16, 9, 8, 16, 10, 7, 8, 16, 17, 1]
2	17-16-10=	-9	[16, 10, 8, 16, 12, 8, 16, 11, 7, 8, 15, 1]
3	(11-18)+5=	-2	[2, 16, 16, 8, 16, 3, 9, 5, 17, 7, 8, 13, 1]
4	2+15*0=	2	[13, 5, 16, 17, 6, 11, 7, 13, 1]

以上第一張圖是切分 token 的部分，我是依照助教提供的範例程式碼下去做修改的，可以看到有成功將資料中的算式切成一個個的 token；第二張圖則是將資料轉為 id 的結果。

2. Generation Models

- ✓ Model design (10 pts). You can use RNN, GRU or LSTM... whatever. (at least one sequential model)

LSTM:

```python
class CharRNN(torch.nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super(CharRNN, self).__init__()

        # Embedding層
        self.embedding = torch.nn.Embedding(num_embeddings=vocab_size,
                                            embedding_dim=embed_dim,
                                            padding_idx=char_to_id['<pad>'])

        # LSTM層
        self.rnn_layer1 = torch.nn.LSTM(input_size=embed_dim,
                                        hidden_size=hidden_dim,
                                        batch_first=True)

        # output層
        self.linear = torch.nn.Sequential(torch.nn.Linear(in_features=hidden_dim,
                                                          out_features=hidden_dim),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(in_features=hidden_dim,
                                                          out_features=vocab_size))
```

RNN(bouns 使用):

```python
class CharRNN(torch.nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super(CharRNN, self).__init__()

        # Embedding層
        self.embedding = torch.nn.Embedding(num_embeddings=vocab_size,
                                            embedding_dim=embed_dim,
                                            padding_idx=char_to_id['<pad>'])

        # RNN層
        self.rnn_layer1 = torch.nn.RNN(input_size=embed_dim,
                                       hidden_size=hidden_dim,
                                       batch_first=True)

        # output層
        self.linear = torch.nn.Sequential(torch.nn.Linear(in_features=hidden_dim,
                                                          out_features=hidden_dim),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(in_features=hidden_dim,
                                                          out_features=vocab_size))
```

✓ Train(finetune) the model (10 pts).

訓練:

```python
from tqdm import tqdm    # 顯示訓練進度條
model = model.to(device)    # 將模型移動到指定的計算設備（GPU 或 CPU）
model.train()    # 設置模型為訓練模式
test_acc = []    # 用於儲存測試準確率的列表
val_losses = []    # 用於儲存驗證集損失的列表
train_losses = []    # 用於儲存訓練集損失的列表
i = 0    # 用於計算訓練過程中的迭代次數
j = 0    # 用於計算驗證過程中的迭代次數
losstemp = 0    # 用於暫存損失值的變數
for epoch in range(1, epochs+1):    # 迭代每個訓練 epoch
    process_bar = tqdm(train_data_loader, desc=f"Training epoch {epoch}")    # 用 tqdm 顯示訓練進度條
    for batch_x, batch_y, batch_x_lens, batch_y_lens, eqidex in process_bar:    # 迭代每個訓練批次
        optimizer.zero_grad()    # 清空梯度
        batch_pred_y = model(batch_x.to(device), batch_x_lens)    # 使用模型進行預測
        batch_pred_y = batch_pred_y[:, :, :]    # 調整預測結果的形狀
        batch_pred_y = batch_pred_y.reshape(batch_size*batch_pred_y.shape[1], vocab_size)    # 調整預測結果的形狀
        batch_y = batch_y.reshape(-1).to(device)    # 調整目標值的形狀
        loss = criterion(batch_pred_y, batch_y)    # 計算損失
        loss.backward()    # 反向傳播
        torch.nn.utils.clip_grad_value_(model.parameters(), grad_clip)    # 梯度裁剪
        optimizer.step()    # 更新模型參數
        i += 1    # 更新迭代次數
        losstemp += loss.item()    # 更新損失暫存值
        if i % 20 == 0:    # 每隔20個批次更新一次進度條顯示
            process_bar.set_postfix(loss=loss.item())    # 更新進度條顯示損失值
    train_losses.append(losstemp)    # 將訓練損失值加入列表
    losstemp = 0    # 重置損失暫存值
    for batch_x, batch_y, batch_x_lens, batch_y_lens, eqidex in val_data_loader:    # 迭代每個驗證批次
        optimizer.zero_grad()    # 清空梯度
        batch_pred_y = model(batch_x.to(device), batch_x_lens)    # 使用模型進行預測
        batch_pred_y = batch_pred_y[:, :, :]    # 調整預測結果的形狀
        batch_pred_y = batch_pred_y.reshape(batch_size*batch_pred_y.shape[1], vocab_size)    # 調整預測結果的形狀
        batch_y = batch_y.reshape(-1).to(device)    # 調整目標值的形狀
        j += 1    # 更新迭代次數
        losstemp += loss.item()    # 更新損失暫存值
        loss = criterion(batch_pred_y, batch_y)    # 計算損失
    val_losses.append(losstemp*4)    # 將驗證損失值加入列表
    validation_process_bar = tqdm(range(50))    # 創建用於測試的進度條
    correctCount = 0    # 用於計算正確的次數
    for i in validation_process_bar:    # 迭代每個測試樣本
        expression, result = generate_data()    # 生成測試數據
        pred = model.generator(expression)    # 使用生成器進行預測
        genStr = ""    # 用於儲存預測的結果字符串
        for x in pred:    # 將預測的結果拼接為字符串
            genStr += x
        if genStr == expression + str(result):    # 判斷預測結果是否正確
            correctCount += 1    # 正確次數加一
    test_acc.append(correctCount/50.0)    # 將測試準確率加入列表
    if epoch % 500 == 0:    # 每500個epoch保存一次模型
        torch.save(model, './models/model_epoch'+str(epoch)+'.pt')    # 保存模型至指定路徑
```

✓ Evaluate your model when you are training and test the model. (10 pts)

節錄部份訓練過程:

```
Training epoch 18: 100%|██████████| 45/45 [00:01<00:00, 28.14it/s, loss=0.248]
100%|██████████| 50/50 [00:00<00:00, 348.58it/s]
Training epoch 19: 100%|██████████| 45/45 [00:01<00:00, 27.86it/s, loss=0.226]
100%|██████████| 50/50 [00:00<00:00, 416.88it/s]
Training epoch 20: 100%|██████████| 45/45 [00:01<00:00, 28.34it/s, loss=0.28]
100%|██████████| 50/50 [00:00<00:00, 311.79it/s]
Training epoch 21: 100%|██████████| 45/45 [00:02<00:00, 20.95it/s, loss=0.254]
100%|██████████| 50/50 [00:00<00:00, 269.60it/s]
Training epoch 22: 100%|██████████| 45/45 [00:02<00:00, 21.53it/s, loss=0.225]
100%|██████████| 50/50 [00:00<00:00, 397.46it/s]
Training epoch 23: 100%|██████████| 45/45 [00:01<00:00, 28.90it/s, loss=0.212]
100%|██████████| 50/50 [00:00<00:00, 366.18it/s]
Training epoch 24: 100%|██████████| 45/45 [00:01<00:00, 29.12it/s, loss=0.195]
```

3. Analysis

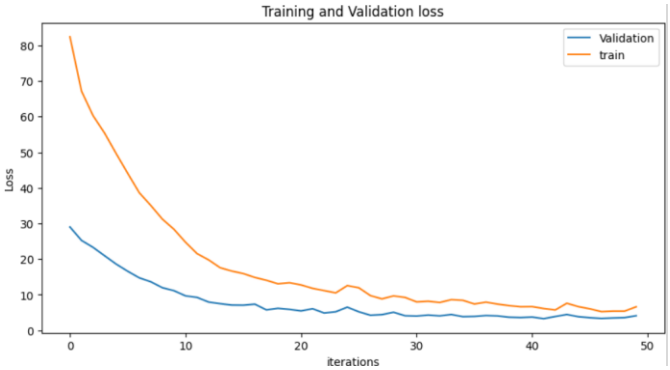
i. Model analysis

model design

我的模型設計包括以下幾個部分，各層的功能如下：

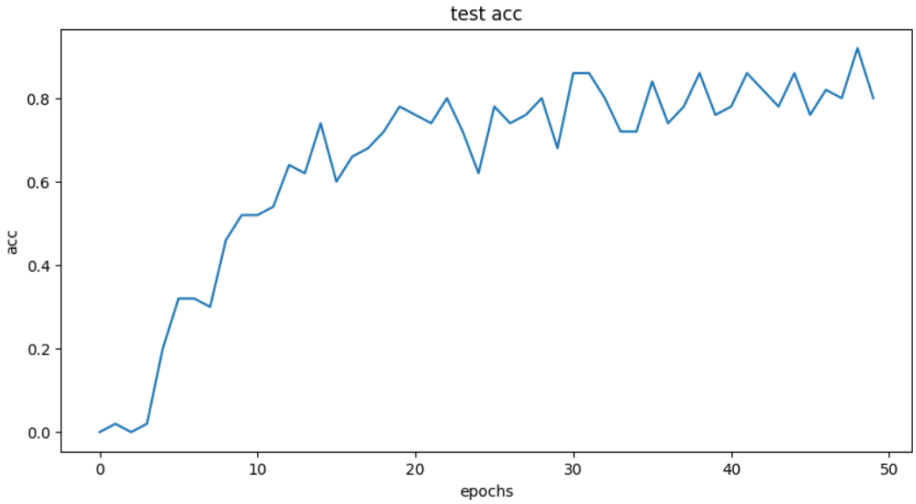
- Embedding layer：用於將輸入序列中的每個標記映射到指定維度的向量表示，意即將 src 進行編碼。
- RNN layer：用於處理嵌入後的序列數據，提取序列中的時間訊息。
- Output layer：用於將 RNN 輸出的隱藏狀態映射到輸出標記的 logits。

Loss Reduction:



隨著 iteration 的增加，可以看到 training 和 validation loss 兩者皆有顯著地下降，代表訓練正常，該網絡仍在學習，且沒有過擬合(overfitting)的情況發生。

results of validation and testing:



上圖是我訓練了 50 個 epoch 後所繪製的 test accuracy，記錄了每次訓練後的 acc 率，可以看到其逐漸上升，最終停在約 0.8。

```
acc = 0.75
```

```
生成正確 9-5+12=16 16
生成有誤差 17-9*19=-144 -154
生成正確 10+16-16=10 10
生成正確 (13+14)*14=378 378
生成正確 (9+4)-8=5 5
生成有誤差 1+10*11=113 111
生成正確 7+12-7=12 12
生成正確 (1+8)*7=63 63
生成正確 1+18*1=19 19
生成正確 (13-7)-16=-10 -10
生成有誤差 18-16*16=-254 -238
生成正確 (0+16)-12=4 4
生成有誤差 2-6*5=-32 -28
生成正確 4+18-18=4 4
生成正確 (5-8)-9=-12 -12
生成正確 3-13+5=-5 -5
生成有誤差 12-11*4=-34 -32
生成有誤差 9-17*1=-10 -8
生成正確 (9-8)*16=16 16
生成正確 7-15+4=-4 -4
生成正確 17+7-0=24 24
生成正確 (17+11)-16=12 12
```

關於驗證的部分，上面兩張圖是我做驗證後的結果，其 acc 率為 0.75。第二張圖是將使用模型生成的答案與正確答

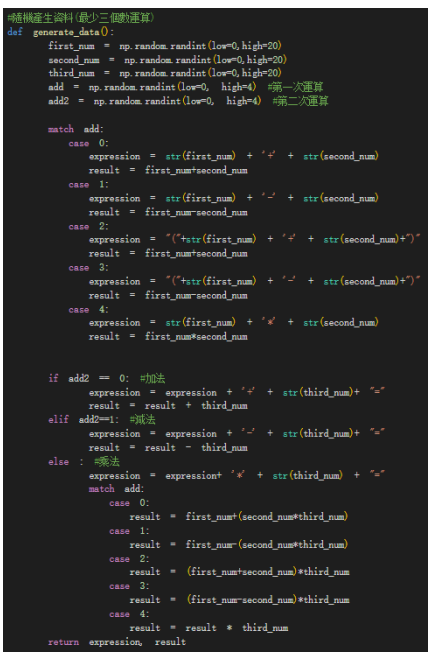
案做比較的結果，前面的算式是透過訓練產生的，空格後的數值則是正確答案，若前後結果相同我將其標為”正確”，反之則標記成”有誤差”，最後再透過計算出成功算對的次數有幾次來得到 validation accuracy，我認為以第一次做這類型的作業來看，結果算是相當不錯了!

ii. Dataset analysis

characteristics of the datasets and understanding of it:

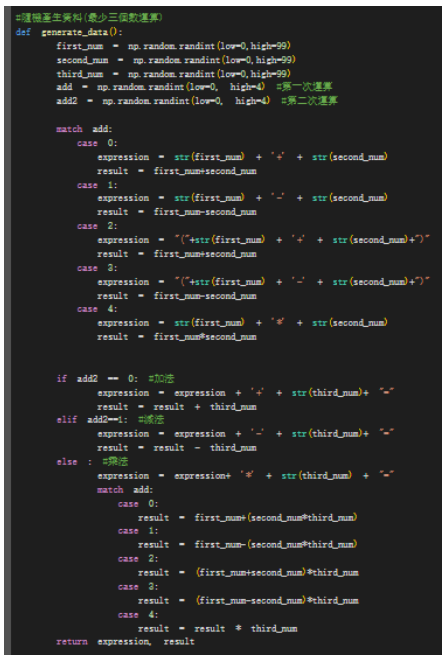
本次作業的資料包含數學算式的輸入和輸出序列，其中輸入序列是算式的表達式，輸出序列是算式的結果。測試資料中的算式類型多樣，包含了加法、減法、乘法等不同類型的算式，以下將會對我生成的**兩種資料**下去作分析與比較:

一、20000 筆範圍 0~20 的資料:



```
#隨機產生資料(最少三個數運算)
def generate_data():
    first_num = np.random.randint(low=0,high=20)
    second_num = np.random.randint(low=0,high=20)
    third_num = np.random.randint(low=0,high=20)
    add = np.random.randint(low=0,  high=4)  #第一次運算
    add2 = np.random.randint(low=0,  high=4)  #第二次運算

    match add:
        case 0:
            expression = str(first_num) + '+' + str(second_num)
            result = first_num+second_num
        case 1:
            expression = str(first_num) + '-' + str(second_num)
            result = first_num-second_num
        case 2:
            expression = "("+str(first_num) + '+' + str(second_num)+")"
            result = first_num+second_num
        case 3:
            expression = "("+str(first_num) + '-' + str(second_num)+")"
            result = first_num-second_num
        case 4:
            expression = str(first_num) + '*' + str(second_num)
            result = first_num*second_num


    if add2 == 0: #加法
            expression = expression + '+' + str(third_num)+ "="
            result = result + third_num
    elif add2==1: #減法
            expression = expression + '-' + str(third_num)+ "="
            result = result - third_num
    else : #乘法
            expression = expression+ '*' + str(third_num) + "="
            match add:
                case 0:
                    result = first_num+(second_num*third_num)
                case 1:
                    result = first_num-(second_num*third_num)
                case 2:
                    result = (first_num+second_num)*third_num
                case 3:
                    result = (first_num-second_num)*third_num
                case 4:
                    result = result * third_num
    return expression, result
```

二、20000 筆範圍 0~99 的資料:



```
#隨機產生資料(最少三個數運算)
def generate_data():
    first_num = np.random.randint(low=0,high=99)
    second_num = np.random.randint(low=0,high=99)
    third_num = np.random.randint(low=0,high=99)
    add = np.random.randint(low=0,  high=4)  #第一次運算
    add2 = np.random.randint(low=0,  high=4)  #第二次運算

    match add:
        case 0:
            expression = str(first_num) + '+' + str(second_num)
            result = first_num+second_num
        case 1:
            expression = str(first_num) + '-' + str(second_num)
            result = first_num-second_num
        case 2:
            expression = "("+str(first_num) + '+' + str(second_num)+")"
            result = first_num+second_num
        case 3:
            expression = "("+str(first_num) + '-' + str(second_num)+")"
            result = first_num-second_num
        case 4:
            expression = str(first_num) + '*' + str(second_num)
            result = first_num*second_num


    if add2 == 0: #加法
            expression = expression + '+' + str(third_num)+ "="
            result = result + third_num
    elif add2==1: #減法
            expression = expression + '-' + str(third_num)+ "="
            result = result - third_num
    else : #乘法
            expression = expression+ '*' + str(third_num) + "="
            match add:
                case 0:
                    result = first_num+(second_num*third_num)
                case 1:
                    result = first_num-(second_num*third_num)
                case 2:
                    result = (first_num+second_num)*third_num
                case 3:
                    result = (first_num-second_num)*third_num
                case 4:
                    result = result * third_num
    return expression, result
```

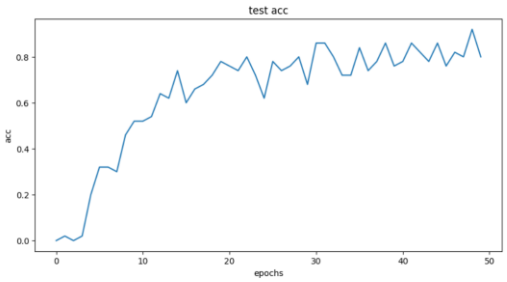
Dataset Adjustment and Impact

```python
class Dataset(torch.utils.data.Dataset):
    def __init__(self, sequences):
        self.sequences = sequences

    def __getitem__(self, index):
        # input:  1 + 2 + 3 = 6
        # output: / / / / / 6 <eos>
        x = self.sequences.iloc[index][:-1]
        y = self.sequences.iloc[index][1:]
        # 找到=的索引
        index = y.index(char_to_id['='])
        # 將=左邊的元素設置為0
        y[:index+1] = [0] * (index+1)

        return x, y

    def __len__(self):
        return len(self.sequences)
```

如同助教給予的提示所說，對於加減法的任務，我們須將算是拆解成 1+2+3=和 6 兩部分，其目的是要讓模型去學習，並且在看到等號之後能產生答案及結數字符<eos>；此外，在計算 loss 時，只需預測等號的後面，故只要計算這部分即可，前面斜線部分可不用採計。

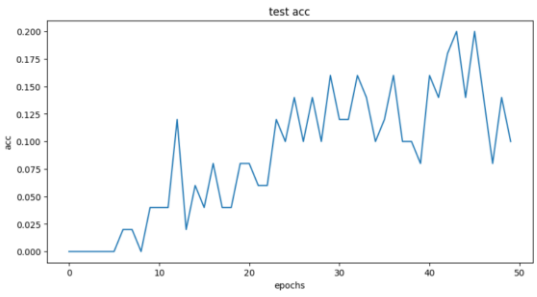
Results:

一、20000 筆範圍 0~20 的資料:



此時的 Loss 都正常下降且收斂，代表在訓練得不錯，也無出現過擬合的問題，acc 率也穩定上升。

二、20000 筆範圍 0~99 的資料:



可以看到 Loss 都有下降但無法像第一次完美收斂，且 acc 率的分數普遍都很低，我推測可能是由於資料數太少或學習率太高所導致。

iii. Discussion

Impact of Different Learning Rates

不同的學習率會對模型的訓練過程和結果產生影響。

根據我的實測，學習率較大時，模型可能會快速收斂，但容易導致損失函數波動較大或無法收斂；反之若學習率較小時，模型收斂速度較慢，但可以更穩定地學習和優化模型參數。

而**本次作業我認為學習率在 0.001 時表現最佳**。

Impact of Different Batch Sizes

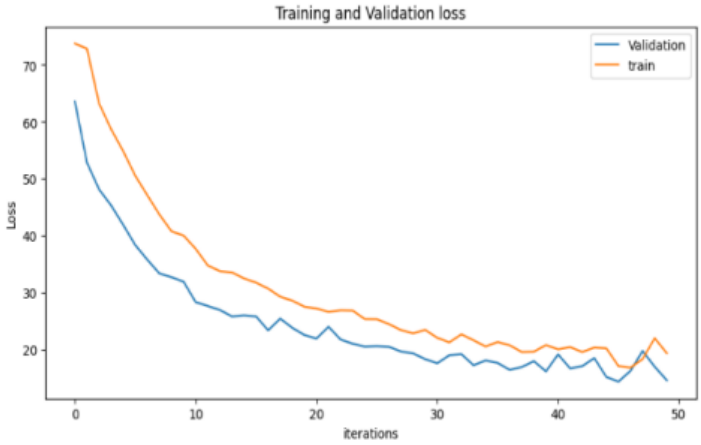
依據我多次改動 Batch size 的結果來看，不同的 Batch 大小會影響模型的訓練速度和泛化能力；另外較大的 Batch 大小可以加快訓練速度，但可能導致模型的泛化能力下降；而較小的 Batch 大小可以提高模型的泛化能力，但訓練速度較慢。本次作業我的 Batch size 最終採用 400。

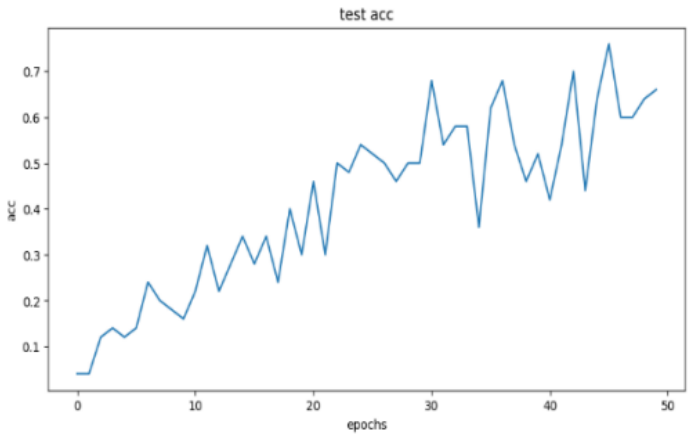
Model Characteristics

我使用的模型具有適用於處理序列數據的特點，包括嵌入層和循環神經網絡層。這種模型適用於處理具有時間序列結構的數據，如自然語言處理任務中的文本序列，以及這次作業中的數學算式序列。

4. Bonus (10 pts)

i. Compare the performance of multiple models and provide a brief analysis. (10 pts)





上圖結果是使用 RNN 作為模型，可以明顯地觀察到，這裡 accuracy rate 較低，根據網路上查詢的資料解釋，LSTM 訓練效果上更佳主要是因為它能有效解決梯度消失和梯度爆

炸的問題，因此，在處理需要記憶較長時間序列訊息的任務時， LSTM 通常能達到更好的訓練和預測效果。