- LSTM 和 Transformer 進行的資料預處理:

製作 index 和 char 之間 token 的對照表，並且 train 中 input 和 target label 的資料都變成對應 token 並在最前端和最後端加上 SOS、EOS，鑒於輸入長度一樣所以把空白處用 PAD 填滿，最後因為 input 可能有多於一個錯誤單字的可能所以將其拆解成一一對應。

```
c2i = {letter: i + 3 for i, letter in enumerate(string.ascii_lowercase)}
c2i.update({'SOS':0})
c2i.update({'EOS':1})
c2i.update({'PAD':2})

i2c = {i + 3 : letter  for i, letter in enumerate(string.ascii_lowercase)}
i2c.update({0:'SOS'})
i2c.update({1:'EOS'})
i2c.update({2:'PAD'})

def c2i_word(input, c2i):
    return [c2i[char] for char in input]
def i2c_word(input, i2c):
    return [i2c[char] for char in input]
```

```
for item in dataset:
    for i in range(len(item['input'])):
        input_data = item['input'][i]
        target_data = item['target']
        train_dataset.append((input_data, target_data))

        input = c2i_word(input_data, c2i)
        input = [SOS] + input
        input = input + [EOS]
        input = input + [PAD] * (word_len - len(input))

        target = c2i_word(target_data, c2i)
        target = [SOS] + target
        target = target + [EOS]
        target = target + [PAD] * (word_len - len(target))

        train_dataset_token.append((input,target))
train_dataset_token = torch.tensor(train_dataset_token)
```

```
train_input = []
train_target = []
for item in train_dataset_token:
    input = item[0]
    target = item[1]
    train_input.append(input)
    train_target.append(target)
```

- LSTM

  Encoder:

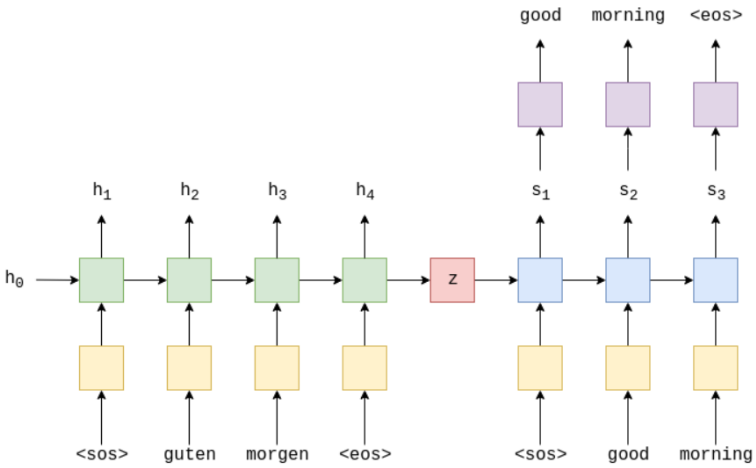
  在 Embedding 把 src 中的每個 token 映射到向量之中，然後透過 LSTM 獲得所需的 hidden 和 cell 來用在 Decoder 上。

  Decoder:

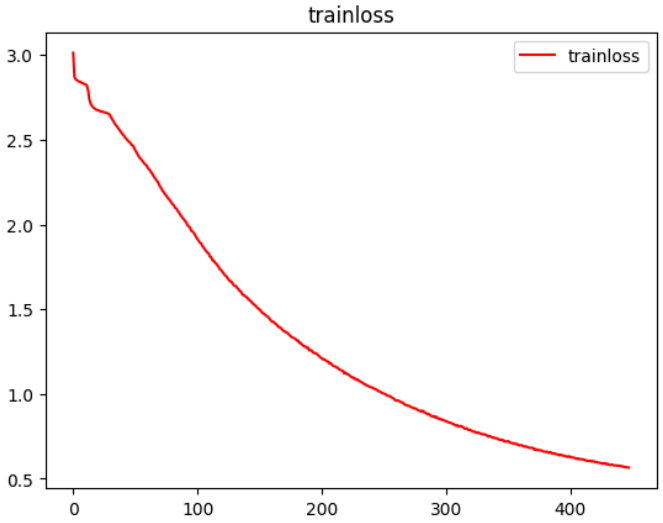
  從 trg 的 SOS 開始進行 embedding，透過 Decoder LSTM 得到 output、hidden 和 cell，然後透過攤平和全連階層得到預測。

  Teacher forcing:

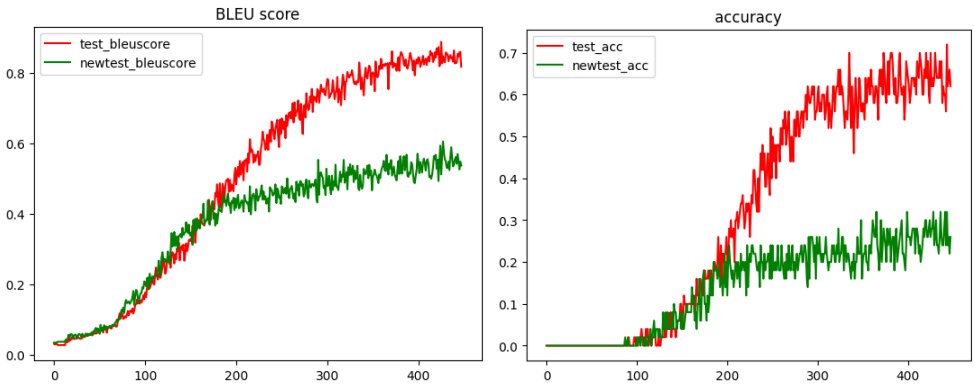
  LSTM 用下圖 SOS 輸出的 good 再次輸入來訓練模型，而使用 teacher forcing 的狀況則是在訓練時若 SOS 輸出非預期的下一個字時，把這個字替換為正確的字來進行訓練(SOS 輸出 good 之外的輸出，換成 good 繼續訓練)



Learning rate = 0.005



Accuracy 落在 72



Learning rate 設定較高時會出現 accuracy 高頻率大幅度變動的狀況，而且就算到了較後面的 epoch 仍不能改善，調整成較小 learning rate 就改善了這個問題。

- Transformer

  Transformer 因為 attention 機制所以信息不再受限於位置順序，所以 Positional encoding 能讓模型感知不同 sequence 的相對位置來幫助訓練。另外還使用了 mask 來防止模型使用未來才能用到的部分，還有填補空缺處，不過 PAD 設為-inf 來避免注意力計算。

  Encoder 用 padding mask，若是 padding 則為 TRUE，model 不考慮此訊息

```
def forward(self, src, src_pad_mask):
    src = self.tok_embedding(src)
    src = self.pos_embedding(src)
    output = self.encoder(src, src_key_padding_mask=src_pad_mask)
    return output
```

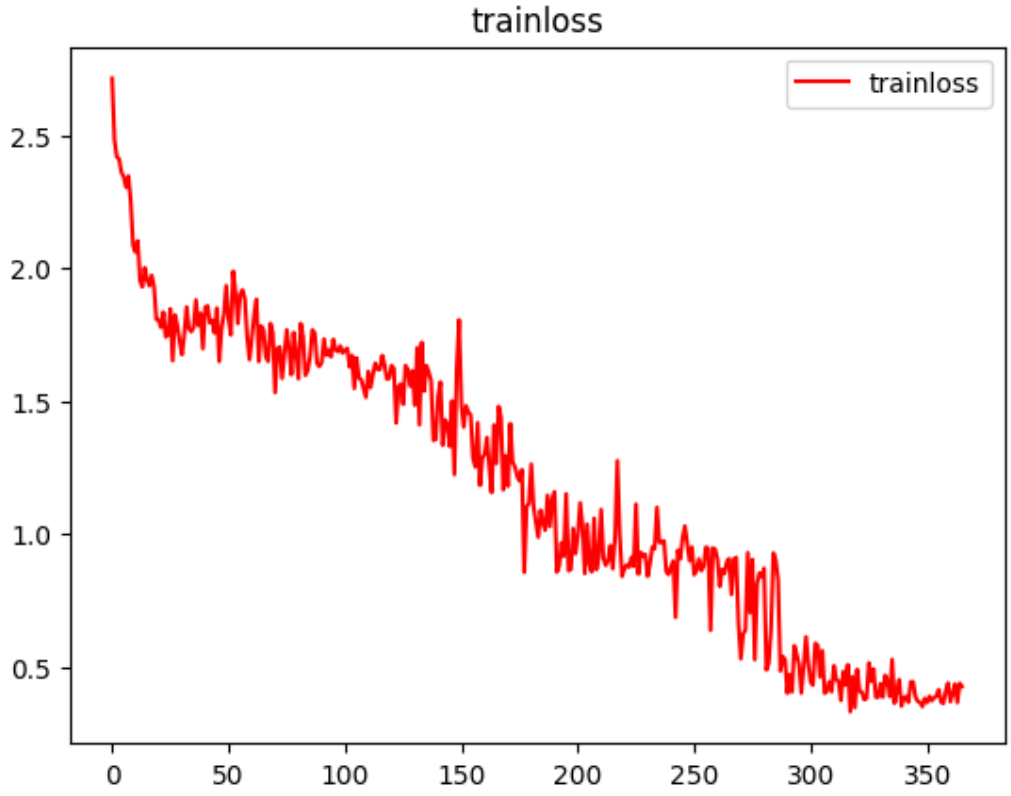
  Decoder 加上上三角矩陣設為-inf，避免 attention mechanism 使用未來資訊

```
def forward(self, tgt,memory, tgt_mask, tgt_pad_mask,src_pad_mask):
    tgt = self.tok_embedding(tgt)
    tgt = self.pos_embedding(tgt)
    output = self.decoder(tgt, memory,tgt_mask=tgt_mask, tgt_key_padding_mask=tgt_pad_mask,memory_key_padding_mask = src_pad_mask)
    output = self.fc(output)
    return output
```

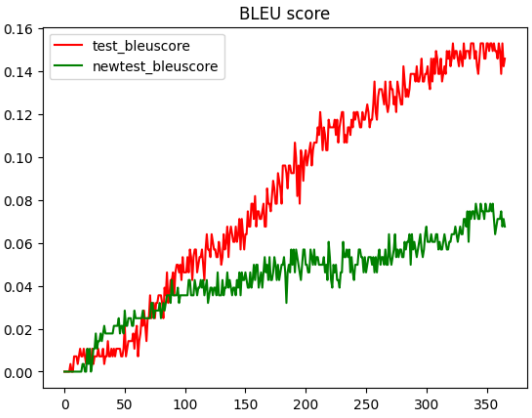
  下圖是 mask code

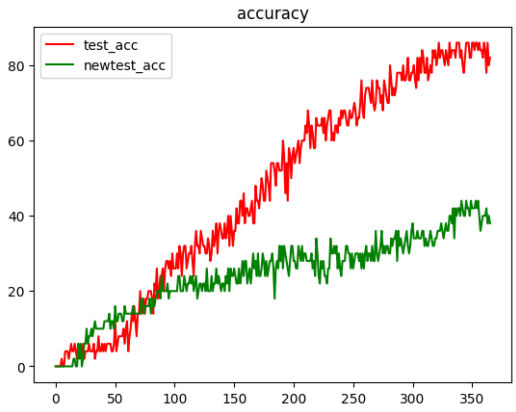
```
def gen_padding_mask(src, pad_idx):
    result = [[element == pad_idx for element in row] for row in src]
    return torch.tensor(result)

def gen_mask(seq):
    x = seq.shape[1]
    return torch.triu(torch.full((x,x), float('-inf')),diagonal=1)

def get_index(pred, dim=2):
    return pred.clone().argmax(dim=dim)
```

Learning rate = 0.03



Accuracy 落在 82





Mask 原本寫法如下導致訓練過程 iter 過低，訓練時間過高，改為第二章圖

```
def gen_padding_mask(src, pad_idx):
    result = [[element == pad_idx for element in row] for row in src]
    return torch.tensor(result)
```

```
def gen_padding_mask(src, pad_idx):
    return src.eq(pad_idx)
```

就結果而言在本此實驗中 Transformer 在準確率的表現略優於 LSTM

差異簡評

LSTM	Transformer
RNN	Attention mechanism
逐步處理	並行處理
不用 positional encoding	要用 positional encoding
參數相對較少	參數相對較多
不用 mask	需要 mask

Transformer 的 code 的結果沒存到然後我又再跑一次所以繳交的 code 的圖和報告不一樣。