- Introduction (5%)

In Lab4, we implemented English spelling corrector by sequence-to-sequence recurrent network. For correcting English spelling, the input must be a word liked ‘recetion’, and output is the corresponded correct word got from dataset, i.e., recession. Therefore, the length of input may be different from the length of output. Obviously, it doesn’t work in traditional RNN architecture. Sequence-to-sequence architecture, or called Encoder-Decoder Framework, was proposed to solve this problem. Fig. 1 shows the Seq2Seq model in Lab4. Furthermore, we can unfold the Encoder and Decoder to graph shown in Fig. 2.

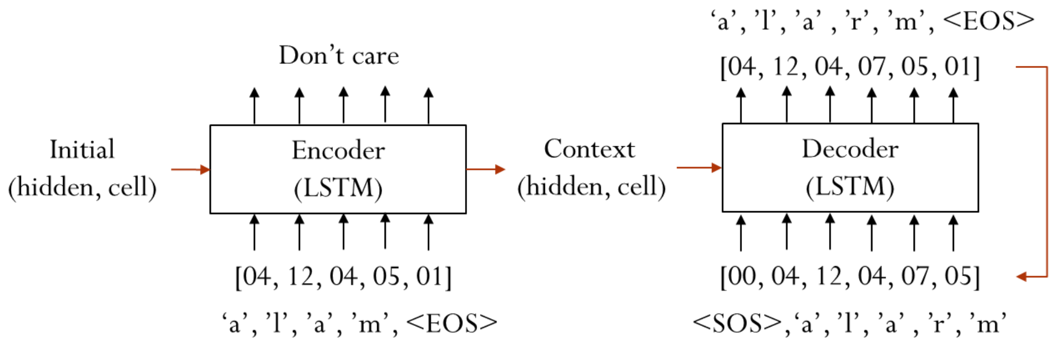


Fig. 1 the Seq2Seq model in Lab4

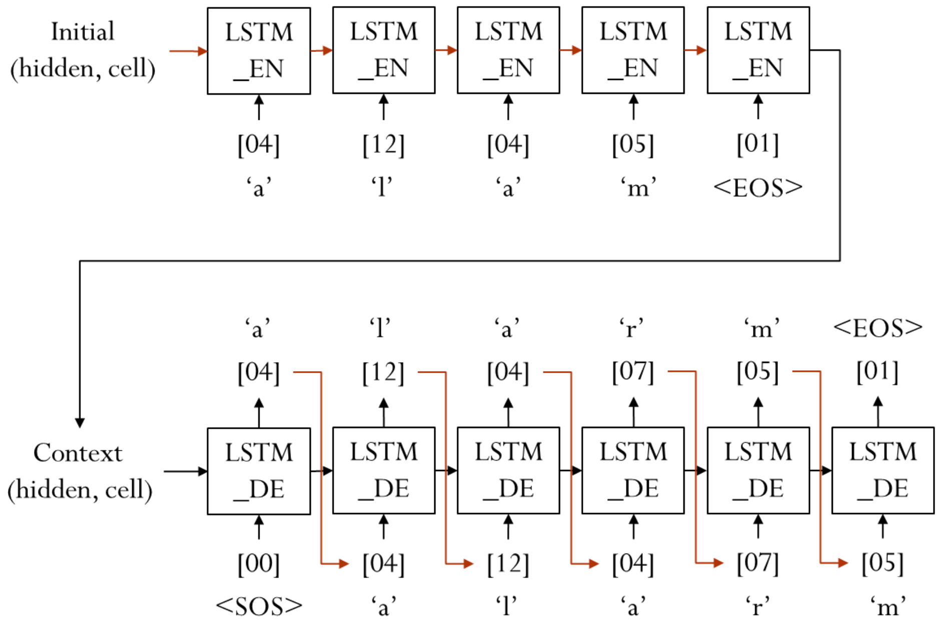
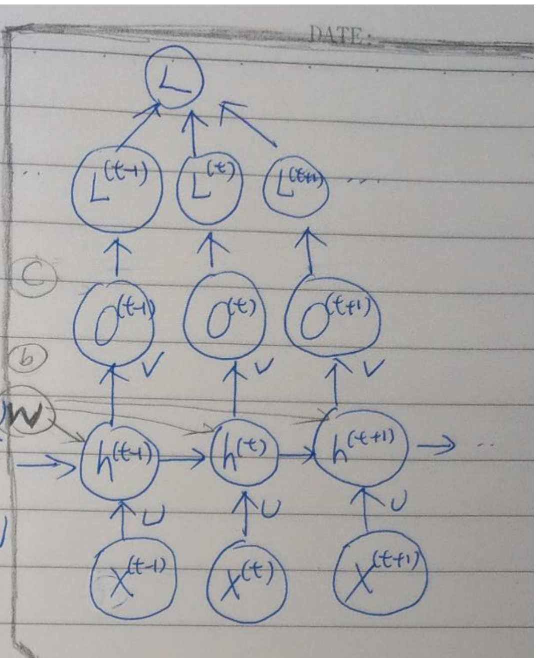


Fig. 2 unfolded Encoder and Decoder in Fig. 1.

- Derivation of BPTT (5%)

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \cdot \nabla_W h_i^{(t)}$$

$$= \sum_t \boxed{\frac{\partial L}{\partial h^{(t)}}} \cdot \boxed{\nabla_W h^{(t)}}$$



根据右边的图和公式,可以得到

$$\frac{\partial L}{\partial h^{(t)}} = \frac{\partial L}{\partial O^{(t)}} \cdot \frac{\partial O^{(t)}}{\partial h^{(t)}} + \frac{\partial L}{\partial h^{(t+1)}} \cdot \frac{\partial h^{(t+1)}}{\partial h^{(t)}}$$

$$= \frac{\partial L}{\partial O^{(t)}} \cdot V + \frac{\partial L}{\partial h^{(t+1)}} \cdot \frac{\partial \tanh(b+Wh^{(t)}+Ux^{(t+1)})}{\partial h^{(t)}}$$

$$= \frac{\partial L}{\partial O^{(t)}} \cdot V + \frac{\partial L}{\partial h^{(t+1)}} \left[1-\tanh^2(b+Wh^{(t)}+Ux^{(t+1)})\right] \cdot W$$

$$= V^T \frac{\partial L}{\partial O^{(t)}} + W^T H^{(t+1)} \cdot \nabla_{h^{(t+1)}} L$$

, where

$$H^{(t+1)} = \begin{bmatrix} 1-(h_1^{(t+1)})^2 & 0 & \cdots & 0 \\ 0 & 1-(h_2^{(t+1)})^2 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & 1-(h_n^{(t+1)})^2 \end{bmatrix}$$

and $\frac{\partial L}{\partial O^{(t)}} = (\hat{y}^{(t)} - y^{(t)})$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = \tanh(a^{(t)})$$

$$O^{(t)} = c + Vh^{(t)}$$

Then, we consider the second term in $\nabla_W L$

$$\boxed{\nabla_W h^{(t)}} = \frac{\partial h^{(t)}}{\partial W} = (1-\tanh^2(a^{(t)})) \cdot h^{(t-1)}$$

$$= H^{(t)} \cdot h^{(t-1)}$$

Thus, $\nabla_W L = \sum_t \nabla_{h^{(t)}} L \cdot H^{(t)} \cdot h^{(t-1)}$

$$= \sum_t H^{(t)} \cdot \nabla_{h^{(t)}} L \cdot h^{(t-1)^T}$$

- Implementation details. (30%)
  - A. Describe how you implement your model. (encoder, decoder, dataloader, etc.).

    To implement seq2seq recurrent network, we need to build the following parts. Then we used these parts to train the dataset.

    - Dataloader

      Because we need a unique index per letter(字母) to use as the inputs and targets of the networks later, we built *Vocabulary* class to collect letters, transform a word to sequence of letters with unique index, and reverse transformation.

```python
class Vocabulary(object):
    def __init__(self, name):
        self.name = name
        self.char2index = {'SOS': 0, 'EOS': 1, 'PAD': 2, 'UNK': 3}
        self.char2count = {}
        self.index2char = {0: 'SOS', 1: 'EOS', 2: 'PAD', 3: 'UNK'}
        self.n_chars = 4  # Count SOS and EOS

    def addWord(self, word):...

    def addChar(self, char):...

    def split_sequence(self, word):...

    def sequence_to_indices(self, sequence, add_eos=False, add_sos=False):...

    def indices_to_sequence(self, indices):...
```

Then, we defined how to read data from json file to pair list.

```python
def readWords(data_path):
    print("Reading data...")

    max_len = 0
    pairs = []
    with open(data_path, 'r') as json_file:
        data = json.load(json_file)
        for p in data:
            words = p['input']
            target = p['target']
            w_num = len(words)
            for i in range(w_num):
                pairs.append([words[i], target])
                if max_len < len(words[i]):
                    max_len = len(words[i])
                if max_len < len(target):
                    max_len = len(target)

    data_vocab = Vocabulary('input_target')

    return data_vocab, pairs, max_len
```

‘prepareData’ is used to fill vocabulary instance.

```python
def prepareData(path):

    data_vocab, pairs, max_len = readWords(path)
    print("Read %s word pairs" % len(pairs))
    print("Counting chars...")
    for pair in pairs:
        data_vocab.addWord(pair[0])
        data_vocab.addWord(pair[1])
    print("Counted chars:")
    print(data_vocab.name, data_vocab.n_chars, data_vocab.char2index)

    return data_vocab, pairs, max_len
```

Now, we can use 'prepareData' to load training dataset and testing dataset.

```python
data_vocab, pairs, MAX_LENGTH = prepareData('train.json')
MAX_LENGTH = MAX_LENGTH + 1
print(random.choice(pairs))
_, test_pairs, _ = prepareData('test.json')
```

- Encoder class

  Encoder includes embedding and LSTM components. Embedding component embedded input to a fixed length vector. Then the fixed length vector and initial (hidden, cell) would be feed into LSTM. The output of LSTM includes predicted output and (hidden, cell), in which the former one is dropped and others would be as hidden input of decoder.

```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, embedding_size, hidden_size, num_layers, dropout):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(input_size, embedding_size)
        self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers, dropout=dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden):
        # input = [input len, batch size]

        embedded = self.dropout(self.embedding(input)).view(1, 1, -1)
        # embedded = [input len, batch size, emb dim]

        # hidden = (hidden, cell)
        output, hidden = self.lstm(embedded, hidden)

        return output, hidden
```

```
def initHidden(self):
    h0 = torch.zeros(1, 1, self.hidden_size)
    c0 = torch.zeros(1, 1, self.hidden_size)
    nn.init.xavier_normal(h0)
    nn.init.xavier_normal(c0)

    hidden = (Variable(nn.Parameter(h0, requires_grad=True)).to(device),
              Variable(nn.Parameter(c0, requires_grad=True)).to(device))

    return hidden
```

- Decoder class

Decoder includes embedding, LSTM, Linear components. Embedding component embedded input to a fixed length vector. Then the fixed length vector and encoder (hidden, cell) would be feed into LSTM. The logsoftmax following by Linear would be apply to output of LSTM.

```
class DecoderRNN(nn.Module):
    def __init__(self, output_size, embedding_size, hidden_size, num_layers, dropout):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.embedding_size = embedding_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(output_size, embedding_size)
        self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers, dropout=dropout)
        self.out = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):

        # input = [1, batch size]
        output = self.dropout(self.embedding(input)).view(1, 1, -1)
        # embedded = [1, batch size, emb dim]

        output = F.relu(output)
        output, hidden = self.lstm(output, hidden)
        output = self.softmax(self.out(output[0]))

        return output, hidden
```

```
def initHidden(self):
    h0 = torch.zeros(1, 1, self.hidden_size)
    c0 = torch.zeros(1, 1, self.hidden_size)
    nn.init.xavier_normal(h0)
    nn.init.xavier_normal(c0)

    hidden = (Variable(nn.Parameter(h0, requires_grad=True)).to(device),
              Variable(nn.Parameter(c0, requires_grad=True)).to(device))

    return hidden
```

- Train function (seq2seq part)

In Train function, we implemented seq2seq model. Firstly, we initial encoder parameters and then feed each letter of input to encoder. The final output of Encoder includes predicted output and (hidden, cell) which would be as hidden input of decoder.

```
encoder_hidden = encoder.initHidden()

encoder_optimizer.zero_grad()
decoder_optimizer.zero_grad()

input_length = input_tensor.size(0)
target_length = target_tensor.size(0)

encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

loss = 0

for ei in range(input_length):
    encoder_output, encoder_hidden = encoder(
        input_tensor[ei], encoder_hidden)
    encoder_outputs[ei] = encoder_output[0, 0]
```

Secondly, we considered the decoder. The first input of decoder is the letter of <SOS>. If use_teacher_forcing is true, we use the target letter as next input. Otherwise, the predicted output would be used as next input of decoder.

```
decoder_input = torch.tensor([[SOS_token]], device=device)

decoder_hidden = encoder_hidden

use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
    # Teacher forcing: Feed the target as the next input
    for di in range(target_length):
        decoder_output, decoder_hidden = decoder(
            decoder_input, decoder_hidden)

        loss += criterion(decoder_output, target_tensor[di])
        decoder_input = target_tensor[di]  # Teacher forcing
```

```
else:
    # Without teacher forcing: use its own predictions as the next input
    for di in range(target_length):
        decoder_output, decoder_hidden = decoder(
            decoder_input, decoder_hidden)
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # detach from history as input
        loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break

loss.backward()

encoder_optimizer.step()
decoder_optimizer.step()

return loss.item() / target_length
```

- Train Iterations function

Here, we defined optimizer, loss function, and how to use data pair to train.

```
encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

pairs_random = pairs
random.shuffle(pairs_random)
training_pairs = []
for k in range(n_iters):
    training_pairs.append(tensorsFromPair(pairs_random[k % len(pairs_random)]))

# ignore_index = PAD_token
criterion = nn.NLLLoss()
# criterion = nn.CrossEntropyLoss(ignore_index=ignore_index)

for iter in range(1, n_iters + 1):
    training_pair = training_pairs[iter - 1]
    input_tensor = training_pair[0]
    target_tensor = training_pair[1]

    loss = train(input_tensor, target_tensor, encoder,
                 decoder, encoder_optimizer, decoder_optimizer, criterion)
    print_loss_total += loss
    plot_loss_total += loss
```

- evaluation function

  BLEU-4 score would be used to evaluate the model results. BLEU (bilingual evaluation understudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. The range of BLEU-4 score is (0, 1), while 1 means the best result.

```
# compute BLEU-4 score
def compute_bleu(output, reference):
    cc = SmoothingFunction()
    if len(reference) == 3:
        weights = (0.33,0.33,0.33)
    else:
        weights = (0.25,0.25,0.25,0.25)
    return sentence_bleu([reference], output, weights=weights, smoothing_function=cc.method1)
```

- Steps of how to train the dataset:
  1. Create data pair (pair) by some functions of dataloader.
  2. Create encoder instance (encoder1) by encoder class.
  3. Create decoder instance (decoder1) by decoder class.
  4. Call training iterations to train data by encoder1, decoder1, and data pair.

```
encoder1 = EncoderRNN(data_vocab.n_chars, embedding_size, hidden_size, num_layers=1, dropout=0.5).to(device)
decoder1 = DecoderRNN(data_vocab.n_chars, embedding_size, hidden_size, num_layers=1, dropout=0.5).to(device)
trainIters(encoder1, decoder1, 904705, print_every=5000, learning_rate=LR)
```

B. You must screen shot the code of evaluation part to prove that you do not use ground truth while testing, otherwise you will get no point at this part.

```python
def evaluate(encoder, decoder, word, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(data_vocab, word)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden

        decoded_words = []
```

```python
        for di in range(max_length):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden)
            # decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(data_vocab.index2char[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words
```

```python
def evaluate_all(encoder, decoder, _pairs):
    bleu_score = 0.0
    for i in range(len(_pairs)):
        pair = _pairs[i]
        # print('input: ', pair[0])
        # print('target:', pair[1])
        output_words = evaluate(encoder, decoder, pair[0])
        output_word = ''
        for k in range(len(output_words) - 1):
            output_word += str(output_words[k])
        # print('pred:  ', output_word)
        # print('')

        bleu_score += compute_bleu(output_word, pair[1])

    bleu_score /= len(_pairs)
    return bleu_score
```

```python
def load_modle_and_evaluate():
    encoder = EncoderRNN(data_vocab.n_chars, hidden_size, num_layers=1).to(device)
    decoder = DecoderRNN(hidden_size, data_vocab.n_chars, num_layers=1).to(device)

    encoder.load_state_dict(torch.load('encoder.dict'))
    decoder.load_state_dict(torch.load('decoder.dict'))

    evaluate_all(encoder, decoder, pairs)
```

- Results and discussion (20%)

A. Show your results of spelling correction and plot the training loss curve and BLUE-4 score testing curve during training. (10%)

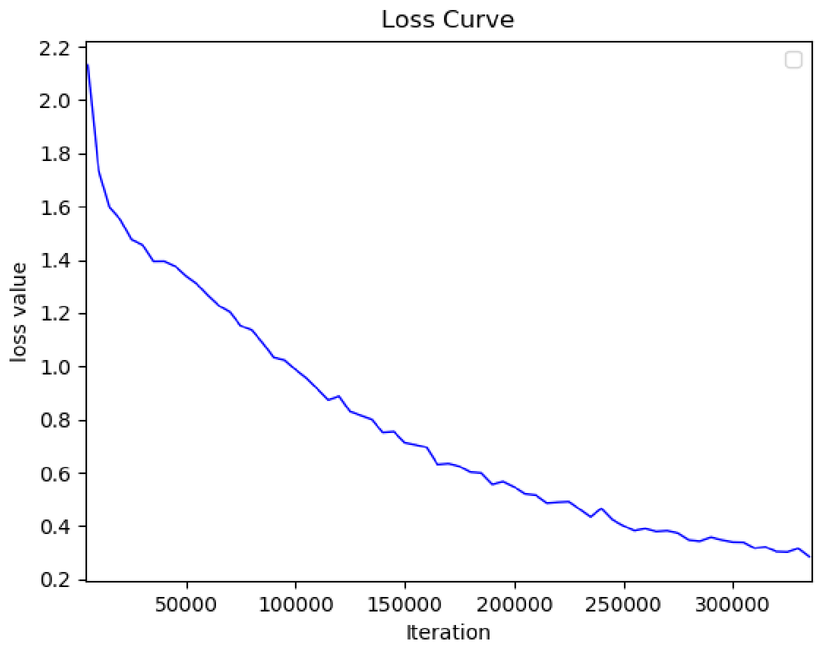
teacher_forcing_ratio = 0.8

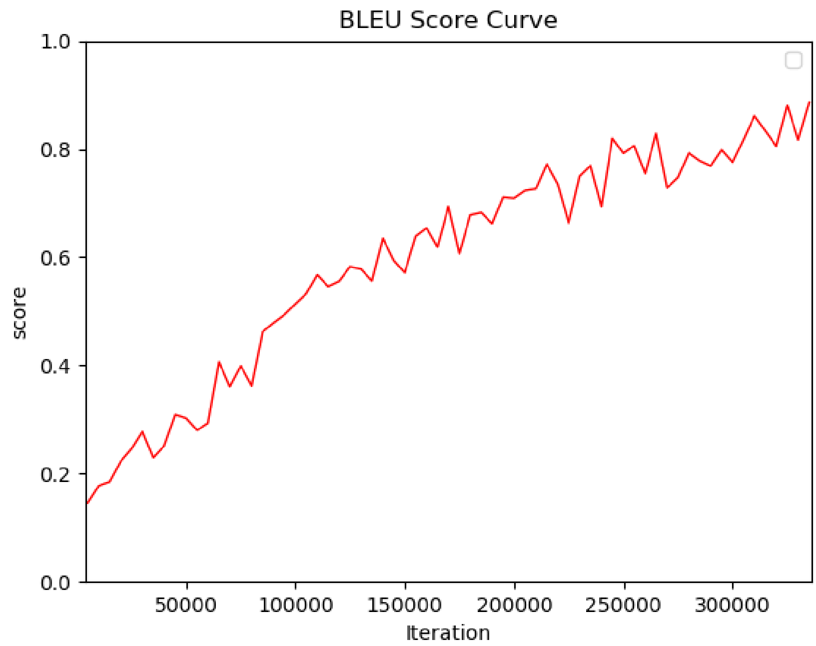
embedding_size = 512

hidden_size = 512

LR=0.05

Iteration = 335000 (it means running almost 26 times of training data)





All outputs of test dataset are list below.

```
input:  contenpted
target: contented
pred:   contenpted

input:  begining
target: beginning
pred:   beginning

input:  problam
target: problem
pred:   problem

input:  dirven
target: driven
pred:   driven

input:  ecstacy
target: ecstasy
pred:   ecstasy
```

```
input:  juce
target: juice
pred:   juice

input:  localy
target: locally
pred:   locally

input:  compair
target: compare
pred:   compare

input:  pronounciation
target: pronunciation
pred:   pronunciation

input:  transportibility
target: transportability
pred:   transportability

input:  miniscule
target: minuscule
pred:   minuscule
```

```
input:  independant
target: independent
pred:   independent

input:  aranged
target: arranged
pred:   arranded

input:  poartry
target: poetry
pred:   pourter

input:  leval
target: level
pred:   level

input:  basicaly
target: basically
pred:   basically

input:  triangulaur
target: triangular
pred:   triangular
```

```
input:  unexpcted
target: unexpected
pred:   unexpected

input:  stanerdizing
target: standardizing
pred:   standardizing

input:  varable
target: variable
pred:   variable

input:  neigbours
target: neighbours
pred:   neighbors

input:  enxt
target: next
pred:   next

input:  powerfull
target: powerful
pred:   powerful
```

```
input:  practial
target: practical
pred:   practical

input:  repatition
target: repartition
pred:   repetition

input:  repentence
target: repentance
pred:   repentance

input:  substracts
target: subtracts
pred:   subtracts

input:  beed
target: bead
pred:   bead

input:  beame
target: beam
pred:   beem
```

```
input:  decieve
target: deceive
pred:   deceive

input:  decant
target: decent
pred:   decent

input:  dag
target: dog
pred:   dog

input:  daing
target: doing
pred:   doing

input:  expence
target: expense
pred:   expense

input:  feirce
target: fierce
pred:   fierce
```

```
input:  fought
target: fort
pred:   fought

input:  fourth
target: forth
pred:   forrt

input:  ham
target: harm
pred:   hame

input:  havest
target: harvest
pred:   harvest

input:  immdiately
target: immediately
pred:   immediately

input:  inehaustible
target: inexhaustible
pred:   inexhauitable
```

```
input:  journel
target: journal
pred:   journal

input:  leason
target: lesson
pred:   lesson

input:  mantain
target: maintain
pred:   manpation

input:  miricle
target: miracle
pred:   miracle

input:  oportunity
target: opportunity
pred:   opportunity

input:  parenthasis
target: parenthesis
pred:   parenthesis
```

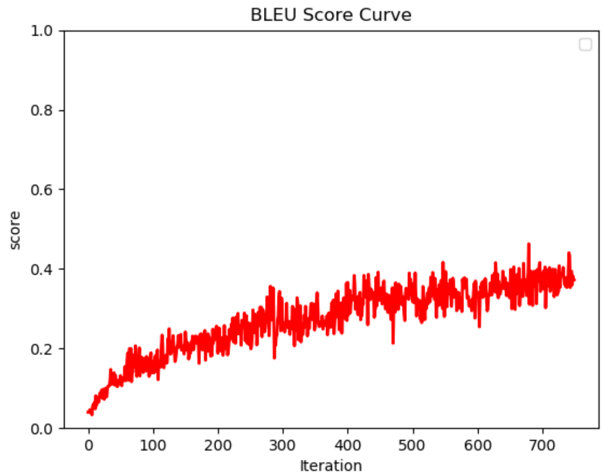
Here is the final word in test dataset and the final test BLEU-4 score 0.90 .

```
input:  scadual
target: schedule
pred:   schedule

0.8998360303705081
```

B. Discussion of the results. (10%)

1. To get higher bleu score, it should learn at least 26 times dataset (12925). It's very time consuming while we train one pair per iteration.
2. It's difficult to get higher bleu score. Usually we get bad results in other parameter settings.



3. *teacher_forcing_ratio* acts as training wheels for the decoder, aiding in

more efficient training. However, teacher forcing can lead to model instability during inference, as the decoder may not have a sufficient chance to truly craft its own output sequences during training. Thus, we must be mindful of how we are setting the *teacher_forcing_ratio*, and not be fooled by fast convergence.

4. LSTM is easy to overfitting, therefore dropout may be needed.