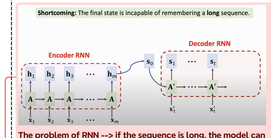
FIT5217 Tutorial 7 Seq2Seq model for Neural Machine Translation (2)

seq2seq with Attention

Problem of RNN

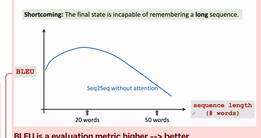
Shortcoming: The final state is incapable of remembering a **long** sequence.



The problem of RNN --> if the sequence is long, the model can forget the begining part of input

Seq2Seq Model

Shortcoming: The final state is incapable of remembering a **long** sequence.

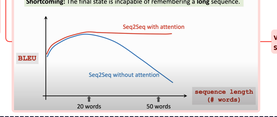


BLEU is a evaluation metric higher --> better

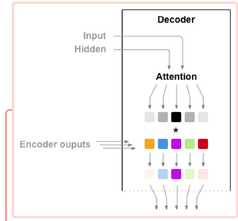
no attention when words increase, the performance drops

Seq2Seq Model

Shortcoming: The final state is incapable of remembering a **long** sequence.



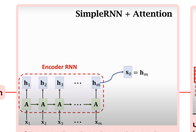
with attention seq2seq significantly improved



Seq2Seq + Attention

simpleRNN + attention

SimpleRNN + Attention



after encoder, decoder and attention starts to work

s_0 here is the encoder output/decoder input

SimpleRNN + Attention

Weight: $\alpha_i = \text{align}(h_i, s_0)$.

use align function to calculate weight for each input

have m α --> sum of them is 1

how to calculate α --> two options

Weight: $\alpha_i = \text{align}(h_i, s_0)$.

Option 1 (used in the original paper):

$\hat{\alpha}_i = v^T \cdot \tanh\left(W \cdot \begin{bmatrix} h_i \\ s_0 \end{bmatrix}\right)$

Then **normalize** $\hat{\alpha}_1, \cdots, \hat{\alpha}_m$ (so that they sum to 1):

$$[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\left([\hat{\alpha}_1, \cdots, \hat{\alpha}_m]\right).$$

option 1: concat h_i and s_0 multiply with w and linearly transformed to a scalar with v

Option 2 (more popular; the same to Transformer):

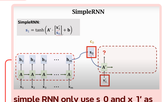
1. Linear maps:
   - $k_i = \mathbf{W}_k \cdot h_i$, for $i = 1$ to m.
   - $q_0 = \mathbf{W}_q \cdot s_0$.
2. Inner product:
   - $\hat{\alpha}_i = k_i^T q_0$, for $i = 1$ to m.
3. Normalization:
   - $[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\left([\hat{\alpha}_1, \cdots, \hat{\alpha}_m]\right)$.

option2: key/query

key vector: map h_i using w_k
query vector: map s_0 using w_q

α is the inner product of w_k and w_q

SimpleRNN



simple RNN only use s_0 and x_1' as input

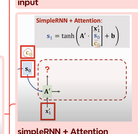
SimpleRNN + Attention

Context vector: $c_0 = \alpha_1 h_1 + \cdots + \alpha_m h_m$.

C_0 is the aggregated h

SimpleRNN + Attention:

$$s_1 = \tanh\left(A' \cdot \begin{bmatrix} x_1' \\ s_0 \\ c_0 \end{bmatrix} + \mathbf{b}\right)$$



simpleRNN + Attention

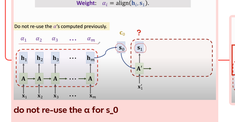
concat c_0, s_0, x_1' as input

the generated s_1 encode the information of h_1 to h_m as it includes c_0 as input

SimpleRNN + Attention

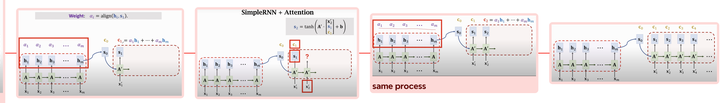
Weight: $\alpha_i = \text{align}(h_i, s_1)$.

Do not re-use the $\hat{\alpha}$'s computed previously.



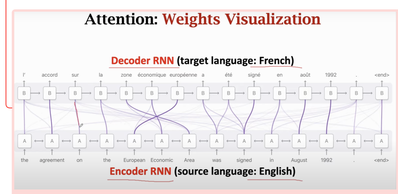
do not re-use the α for s_0

calculate s_1 with h_1 to _h_m to obtain new α



same process

Attention: Weights Visualization



see the relationship between one words and other words

four key components:
embedding layer
attention layer
attention combine layer
GRU
output layer

forward

1 concat embedding and hidden representation and put into the attention layer to get the attention weights

2 use attention weight to aggregate encoder outputs

3 use the aggregated representation concat with embedding and linearly transformer to output with attention_combine

Weight: $\alpha_i = \text{align}(h_i, s_0)$.

Option 1 (used in the original paper):

$\hat{\alpha}_i = v^T \cdot \tanh\left(W \cdot \begin{bmatrix} h_i \\ s_0 \end{bmatrix}\right)$

Then **normalize** $\hat{\alpha}_1, \cdots, \hat{\alpha}_m$ (so that they sum to 1):

$$[\alpha_1, \cdots, \alpha_m] = \text{Softmax}\left([\hat{\alpha}_1, \cdots, \hat{\alpha}_m]\right).$$

```
class AttnDecoderRNN(nn.Module):
    def __init__(self,hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(1),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

another version --> directly use dot product

```
teacher_forcing_ratio = 0.5

def train_attn(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    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]

    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_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            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_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            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
```

training process is similar to the previous version

```
def evaluate_attn(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        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 = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            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(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

similar evaluation process

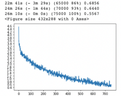
```
def evaluateRandomly_attn(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions= evaluate_attn(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

similar evaluation process

```
def trainIters_attn(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]

    criterion = nn.NLLLoss()

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

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)
```

similar evaluation process

```
hidden_size = 256
attn_encoder = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(device)

trainIters_attn(encoder, attn_decoder, 75000, print_every=5000)

1m 44s (- 25m 12s) (5000 6%) 2.8447
3m 35s (- 22m 53s) (10000 13%) 2.5280
5m 15s (- 20m 54s) (15000 20%) 1.9879
6m 57s (- 19m 7s) (20000 26%) 1.7520
8m 43s (- 17m 21s) (25000 33%) 1.5416
10m 29s (- 15m 44s) (30000 40%) 1.3966
12m 8s (- 13m 53s) (35000 46%) 1.2654
13m 54s (- 12m 9s) (40000 53%) 1.1203
15m 37s (- 10m 26s) (45000 60%) 0.9966
17m 23s (- 8m 42s) (50000 66%) 0.9114
19m 11s (- 6m 58s) (55000 73%) 0.8439
20m 54s (- 5m 14s) (60000 80%) 0.7825
22m 41s (- 3m 29s) (65000 86%) 0.6956
24m 26s (- 1m 45s) (70000 93%) 0.6577
26m 13s (- 0m 0s) (75000 100%) 0.5567
<Figure size 432x288 with 0 Axes>
```



Training attention-based RNN

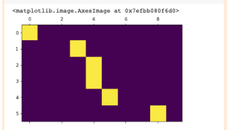
```
evaluateRandomly_attn(encoder, attn_decoder)
```

```
> nous sommes tres reconnaissants .
= we re very grateful .
< we re very grateful . <EOS>

> ils sont en colere apres toi .
= they re mad at you .
< they re mad at you . <EOS>

> je suis impatient de te voir .
= i m anxious to see you .
< i m looking forward to seeing you . <EOS>

> nous sommes en train de couler .
= we re sinking .
< we re meditating . <EOS>

> nous sommes toutes la .
= we re all here .
< we re all here . <EOS>

> elle est malade depuis mercredi dernier .
= she s been sick since last wednesday .
< she s been sick last with . <EOS>

> vous etes paresseux .
= you re lazy .
< you re lazy . <EOS>

> je suis a la maison .
= i m at home .
< i m at the <EOS>

> je suis conscient de la possibilite .
= i m aware of the possibility .
< i m aware of the possibility . <EOS>

> je ne suis pas embarrasse .
= i m not embarrassed .
< i m not embarrassed . <EOS>
```

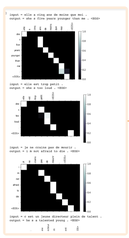
Visualizing Attention

```
%matplotlib inline
output_words, attentions = evaluate_attn(
    encoder, attn_decoder, "je suis trop froid .")
plt.matshow(attentions.numpy())
```

```
<matplotlib.image.AxesImage at 0x7efbb080f6d0>
```



get the output words and attention



```
def showAttention(input_sentence, output_words, attentions):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # Set up axes
    ax.set_xticklabels([''] + input_sentence.split(' ') +
                       [']<EOS>'], rotation=90)
    ax.set_yticklabels([''] + output_words)

    # Show label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()


def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate_attn(
        encoder, attn_decoder, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)


evaluateAndShowAttention("elle a cinq ans de moins que moi .")

evaluateAndShowAttention("elle est trop petit .")

evaluateAndShowAttention("je ne crains pas de mourir .")

evaluateAndShowAttention("c est un jeune directeur plein de talent .")
```

Presented with xmind