Deep Neural Networks for Natural Language Processing (AI6127)

JUNG-JAE KIM

TUTORIAL 6: SEQUENCE-TO-SEQUENCE WITH ATTENTION

Question 1: Run machine translation (MT) with seq2seq and attention

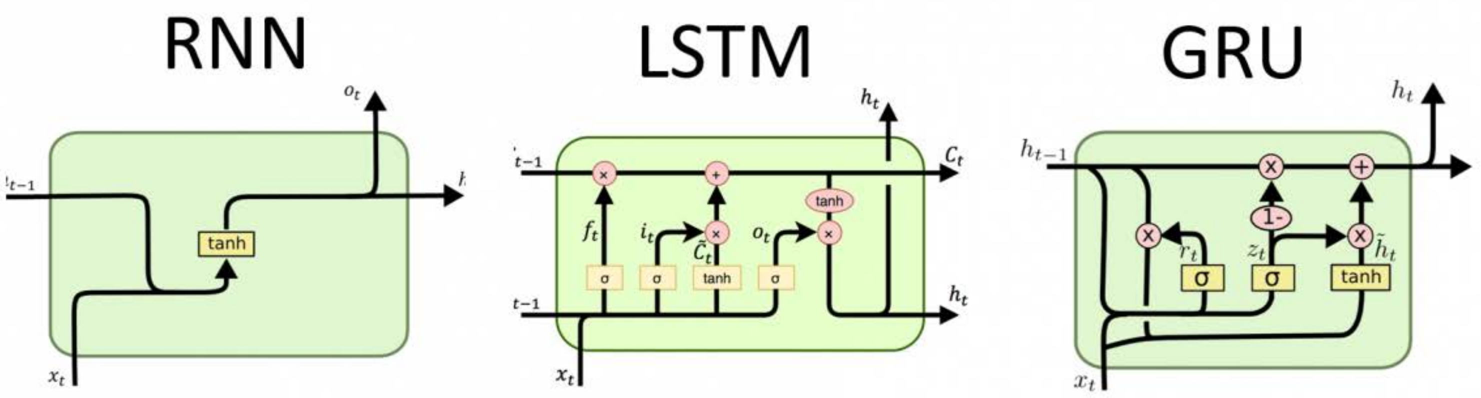
- https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
  - Download data file from https://download.pytorch.org/tutorial/data.zip
  - Revise the function filterPairs in the section “Loading data files”
    - filterPairs: filter sentence pairs whose length is less than MAX_LENGTH ~~and which start with eng_prefixes (e.g. “I am”, “you are”)~~

Question 2: Answer questions about the MT model of Question 1

- Which RNN model is used as encoder?
- Which RNN model is used as decoder?

review

RNN variants



Question 3: Measure with BLUE

- Split the sentence pairs as follows:
  - Randomly select 10% of the pairs for testing/evaluation
  - For each English sentence in the selected pairs, find all its French translations from the whole set of sentence pairs
  - Make test data as the list of (English sentence, list of its French translations)
  - Select all sentence pairs whose English sentences are not included in the test data, as training data
- Train the model of Question 1 with the training data

Question 3: Measure with BLUE

- Evaluate the trained model with the test data by using NLTK library
  - https://www.nltk.org/_modules/nltk/translate/bleu_score.html

review

How do we evaluate Machine Translation?

- BLEU (Bilingual Evaluation Understudy)
- BLEU compares the machine-written translation to one or several human-written translation(s), and computes a similarity score based on:
  - n-gram precision (usually for 1, 2, 3 and 4-grams)
  - Plus a penalty for too-short system translations
- BLEU is useful but imperfect
  - There are many valid ways to translate a sentence
  - So a good translation can get a poor BLEU score because it has low *n*-gram overlap with the human translation

Source: ” BLEU: a Method for Automatic Evaluation of Machine Translation", Papineni et al, 2002.

Question 4: Implement beam search

- The notebook currently implements the greedy decoding method. Change it to the beam search decoding method.
  - Use https://github.com/budzianowski/PyTorch-Beam-Search-Decoding
  - Beam size = 10
  - Skip returning decoder attention outputs

Question 4

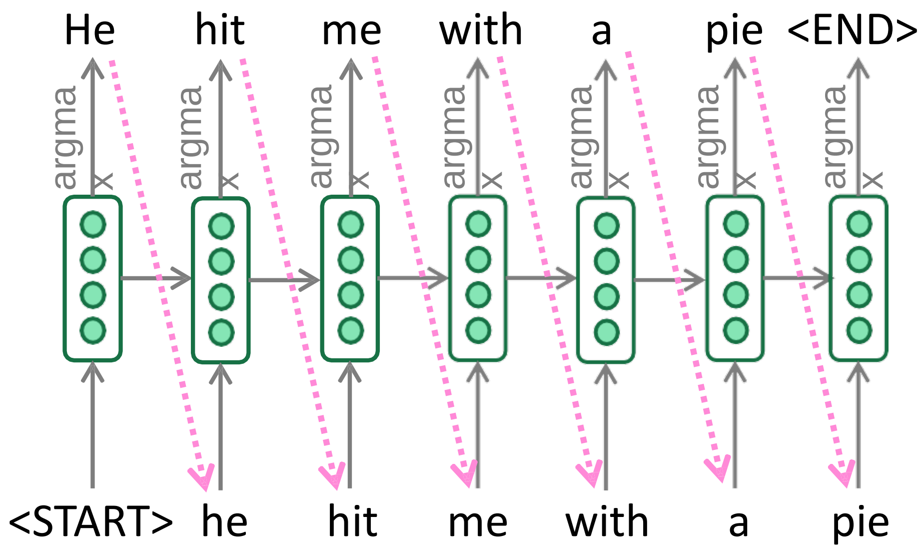
```
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    …
    for di in range(max_length):
        decoder_output, decoder_hidden, decoder_attention = decoder(
            decoder_input, decoder_hidden, encoder_outputs)
        decoder_attentions[di] = decoder_attention.data
        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()
```

review

Greedy decoding

- We saw how to generate (or “decode”) the target sentence by taking argmax on each step of the decoder



- This is greedy decoding (take most probable word on each step)
- Problems with this method?

Problems with greedy decoding

- Greedy decoding has no way to undo decisions!
  - Input: *il a m'entarté* *(he hit me with a pie)*
  - → *he* ______
  - → *he hit* _____
  - → *he hit a* _____ (whoops! no going back now…)
- How to fix this?

review

Beam search decoding

- Core idea: On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses)
  - k is the beam size (in practice around 5 to 10)
- A hypothesis $y_1, \ldots, y_t$ has a score which is its log probability:

$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

  - Scores are all negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step
- Beam search is not guaranteed to find optimal solution
- But much more efficient than exhaustive search!

Question 5

- Find the answer of the questions about parameters of beam search implemented in the https://github.com/budzianowski/PyTorch-Beam-Search-Decoding
  - What stopping criteria are used?
  - What normalization is used?

review

Beam search decoding

- Core idea: On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses)
  - k is the beam size (in practice around 5 to 10)
- A hypothesis $y_1, \ldots, y_t$ has a score which is its log probability:

$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t|x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1, \ldots, y_{i-1}, x)$$

  - Scores are all negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step
- Beam search is not guaranteed to find optimal solution
- But much more efficient than exhaustive search!

review

Beam search decoding: stopping criterion

- In greedy decoding, usually we decode until the model produces a <END> token
  - For example: *<START> he hit me with a pie <END>*
- In beam search decoding, different hypotheses may produce <END> tokens on different timesteps
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via beam search.
- Usually we continue beam search until:
  - We reach timestep T (where T is some pre-defined cutoff), or
  - We have at least n completed hypotheses (where n is pre-defined cutoff)

review

Beam search decoding: finishing up

- We have our list of completed hypotheses.
- How to select top one with highest score?
- Each hypothesis $y_1, \ldots, y_t$ on our list has a score

$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t|x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1, \ldots, y_{i-1}, x)$$

- Problem with this: longer hypotheses have lower scores
- Fix: Normalize by length. Use this to select top one instead:

$$\frac{1}{t} \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1, \ldots, y_{i-1}, x)$$

Question 6: Optimize beam search

- Alternate the above two parameters to achieve higher performance
- And, compare its performance with the greedy decoding method

Hint 6: Alternatives

- Stopping criteria
  - In greedy decoding, usually we decode until the model produces a <END> token
  - Decode until we reach timestep T
  - Decode until we have at least n completed hypotheses (where n is pre-defined cutoff)
- Normalization
  - With or without length normalization

Hands-on

Answer 1

```
def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH #and \
        #p[1].startswith(eng_prefixes)
```

Answer 2

- Encoder: GRU

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        self.gru = nn.GRU(hidden_size, hidden_size)
```

- Decoder: GRU

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        self.gru = nn.GRU(hidden_size, hidden_size)
```

Answer 3 - Split the sentence pairs

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2,
reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))

    // collect test pairs
    num_test = int(len(pairs)*0.1)
    random.shuffle(pairs)
    test_pairs = pairs[:num_test]
    set_test_eng = set([sent_eng for sent_eng, _ in test_pairs])

    test_pair_dict = {}
    for sent_eng, sent_fre in pairs:
        if sent_eng not in set_test_eng: continue
        elif sent_eng not in test_pair_dict:
            test_pair_dict[sent_eng] = set([sent_fre])
        else: test_pair_dict[sent_eng].add(sent_fre)
    test_pairs = [(sent_eng, list(test_pair_dict[sent_eng])) for
sent_eng in test_pair_dict]
    print("Number of test pairs (sent + list):", len(test_pairs))
```

Answer 3 - Split the sentence pairs

```
    # collect train pairs
    train_pairs = [(sent_eng, sent_fre) for sent_eng, sent_fre in pairs[num_test:] if sent_eng not in set_test_eng]
    print("Number of train pairs:", len(train_pairs))

    print("Counting words...")
    for pair in train_pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, train_pairs, test_pairs

input_lang, output_lang, train_pairs, test_pairs = prepareData('eng', 'fra', False)
print(random.choice(train_pairs))
```

Answer 3 – train with the training data

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):

    …

    training_pairs = [tensorsFromPair(random.choice(train_pairs))
                      for i in range(n_iters)]
```

Answer 3

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ') if word in lang.word2index]


from nltk.translate.bleu_score import corpus_bleu
def evaluateBleu(encoder, decoder):
    references, candidates = [], []
    for sent_eng, sents_fre in test_pairs:
        sents_fre = [sent_fre.split(' ') for sent_fre in sents_fre]
        output_words, _ = evaluate(encoder, decoder, sent_eng)
        references.append(sents_fre)
        candidates.append(output_words)
    score = corpus_bleu(references, candidates)
    return score


evaluateBleu(encoder1, attn_decoder1)


0.1638904170559944
```

- Notebook
- Beam search implementation
- Revisions

Answer 4

```
class BeamSearchNode(object):
    def __init__(self, hiddenstate, previousNode, wordId, logProb, length):
        self.h = hiddenstate
        self.prevNode = previousNode
        self.wordid = wordId
        self.logp = logProb
        self.leng = length

    def eval(self, alpha=1.0):
        reward = 0
        # Add here a function for shaping a reward

        return self.logp / float(self.leng - 1 + 1e-6) + alpha * reward

    def __lt__(self, other):
        return self.eval() < other.eval()
```

```
from queue import PriorityQueue
nodes = PriorityQueue()
node = BeamSearchNode(decoder_hidden, None, decoder_input, 0, 1)
nodes.put((-node.eval(), node))
```

- Notebook
- Beam search implementation
- Revisions

Answer 4

```
from queue import PriorityQueue

def evaluate_beam_search(encoder, decoder, sentence, max_length=MAX_LENGTH, beam_size=2):
    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)
        decoder_hidden = encoder_hidden
        # Number of sentence to generate
        endnodes, number_required = [], 1
        # starting node
        node = BeamSearchNode(decoder_hidden, None, decoder_input, 0, 1)
        nodes = PriorityQueue()
```

- Notebook
- Beam search implementation
- Revisions

Answer 4

```
# start the queue
nodes.put((-node.eval(), node))
qsize = 1
# start beam search
while True:
    # give up when decoding takes too long
    if qsize > 2000: break
    # fetch the best node
    score, n = nodes.get()
    decoder_input, decoder_hidden = n.wordid, n.h
    if n.wordid.item() == EOS_token and n.prevNode != None:
        endnodes.append((score, n))
        # if we reached maximum # of sentences required
        if len(endnodes) >= number_required: break
        else: continue
    # decode for one step using decoder
    decoder_output, decoder_hidden, _ = decoder(decoder_input, decoder_hidden, encoder_outputs)
    # PUT HERE REAL BEAM SEARCH OF TOP
    log_prob, indexes = torch.topk(decoder_output, beam_size)
```

- Notebook
- Beam search implementation
- Revisions

Answer 4

```
for new_k in range(beam_size):
    decoded_t = indexes[0][new_k].view(1, -1)
    log_p = log_prob[0][new_k].item()
    node = BeamSearchNode(decoder_hidden, n,
decoded_t, n.logp + log_p, n.leng + 1)
    score = -node.eval()
    nodes.put((score, node))
    qsize += 1 # increase qsize
# choose nbest paths, back trace them
if len(endnodes) == 0:
    endnodes = [nodes.get() for _ in
range(number_required)]
    _, n = endnodes[0]
    utterance = [output_lang.index2word[n.wordid.item()]]
    # back trace
    while n.prevNode != None:
        n = n.prevNode
        utterance.append(
output_lang.index2word[n.wordid.item()])
    utterance = utterance[::-1]
  return utterance, None
```

Answer 4

```
from nltk.translate.bleu_score import corpus_bleu
def evaluateBleu_beam_search(encoder, decoder, beam_size):
    references, candidates = [], []
    for sent_eng, sents_fre in test_pairs:
        sents_fre = [sent_fre.split(' ') for sent_fre in sents_fre]
        output_words, _ = evaluate_beam_search(encoder, decoder, sent_eng, beam_size=beam_size)
        references.append(sents_fre)
        candidates.append(output_words)
    score = corpus_bleu(references, candidates)
    return score
evaluateBleu_beam_search(encoder1, attn_decoder1, 10)
0.1458494830044386
```

Answer 5

- Stopping criteria

```
def evaluate_beam_search(…): …
    if qsize > 2000: break
    if n.wordid.item() == EOS_token and n.prevNode != None:
            endnodes.append((score, n))
            # if we reached maximum # of sentences required
            if len(endnodes) >= number_required: break
            else: continue
```

at least *n* completed hypotheses

decode until the model produces a <END> token

- Normalization

```
def eval(…): …
    return self.logp / float(self.leng - 1 + 1e-6) + alpha * reward
```

Normalize by length

Answer 6

- with length normalization and w/o max_length (original implementation)
  - 0.1458494830044386
- with length normalization and with max_length
  - 0.14496111292459146

```
if n.wordid.item() == EOS_token and n.prevNode != None:
    endnodes.append((score, n))
    # if we reached maximum # of sentences required
    if len(endnodes) >= number_required: break
    else: continue
elif n.leng > max_length: continue
```

- w/o length normalization and w/o max_length
  - 0.165968280155587