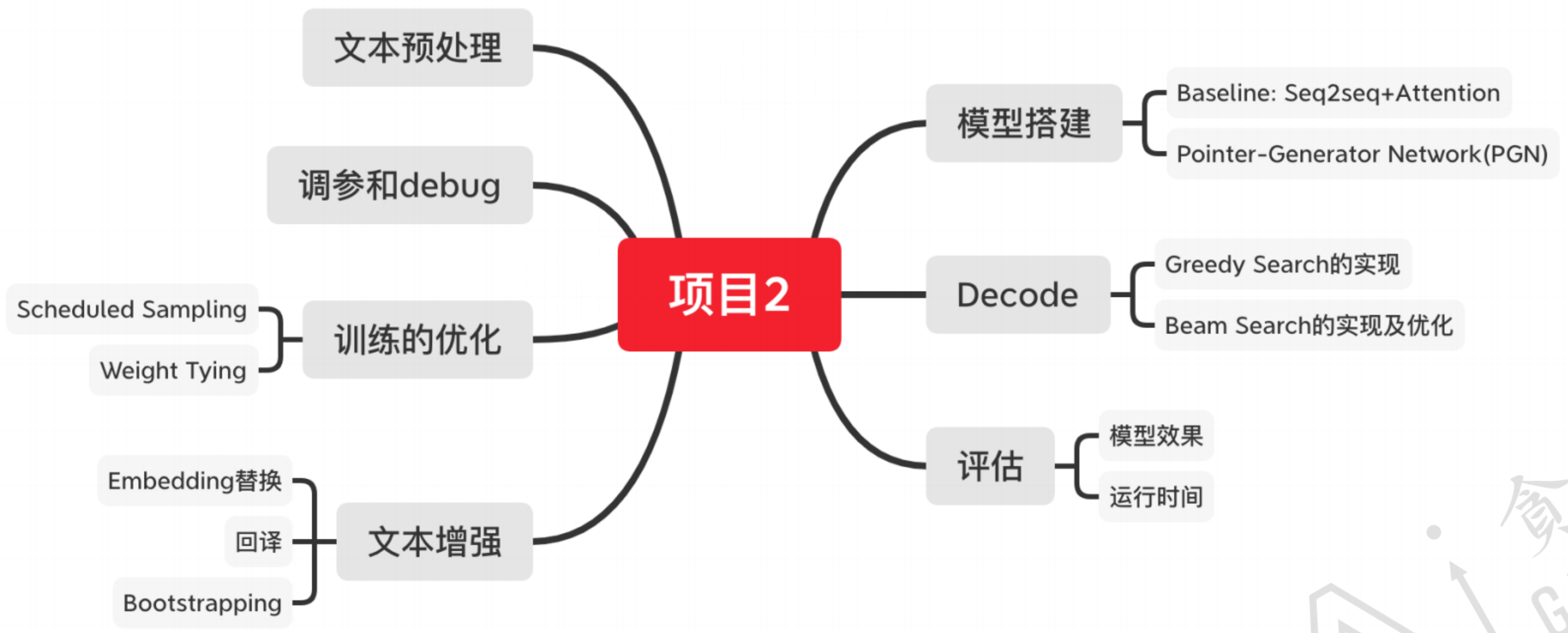
项目2：基于京东电商的营销文本生成



项目2：基于京东电商的营销文本生成

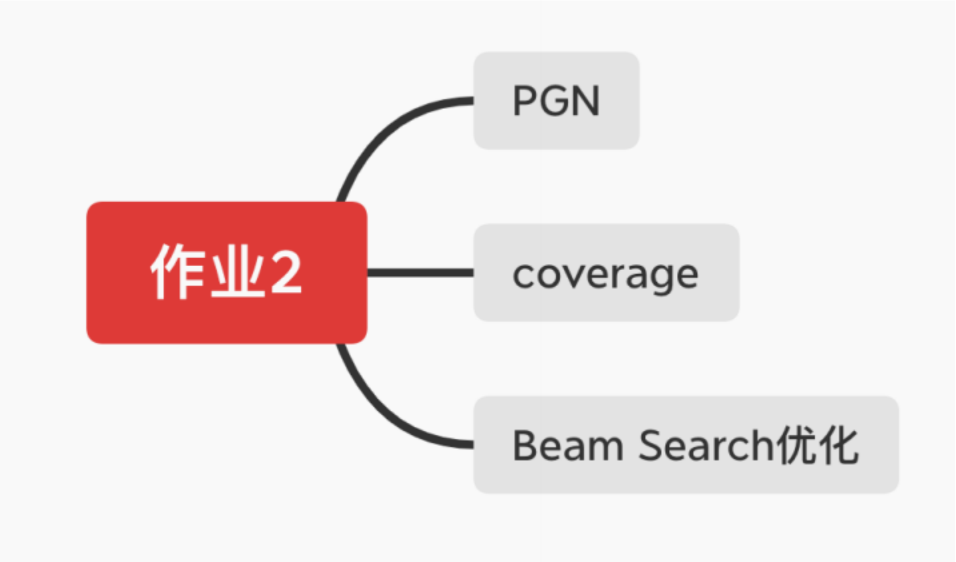
本项目我们分为三个Assignments:

- 生成式摘要的方法构建一个 Seq2seq+Attention 的模型作为 baseline。
- 构建一个结合了生成式和抽取式两种方法的 Pointer-Generator Network 模型
- 加入优化技巧优化结果，并引入数据增强来提高效果。

Contents

Assignment 2:
对于本次任务，需要完成如下的部分：

- 第一：将我们的 baseline 模型改造成一个 Pointer-Generator Network(PGN)。
- 第二：加上 coverage 机制。
- 第三：上次未讲的Beam Search 实现
- 第四：实现对 Beam Search 的优化以达到对输出更好的控制。



模块1: OOV tokens处理

model/utils.py:

任务1: 完成abstract2ids函数。

由于PGN可以 成在source 出现过的OOV tokens，所以这次我们对reference的token ids需要换 种映射 式，即将在source出现过的OOV tokens也记录下来并给 个临时的id， 不是直接替换为"<UNK>" token，以便在训练阶段准确的计算损失。

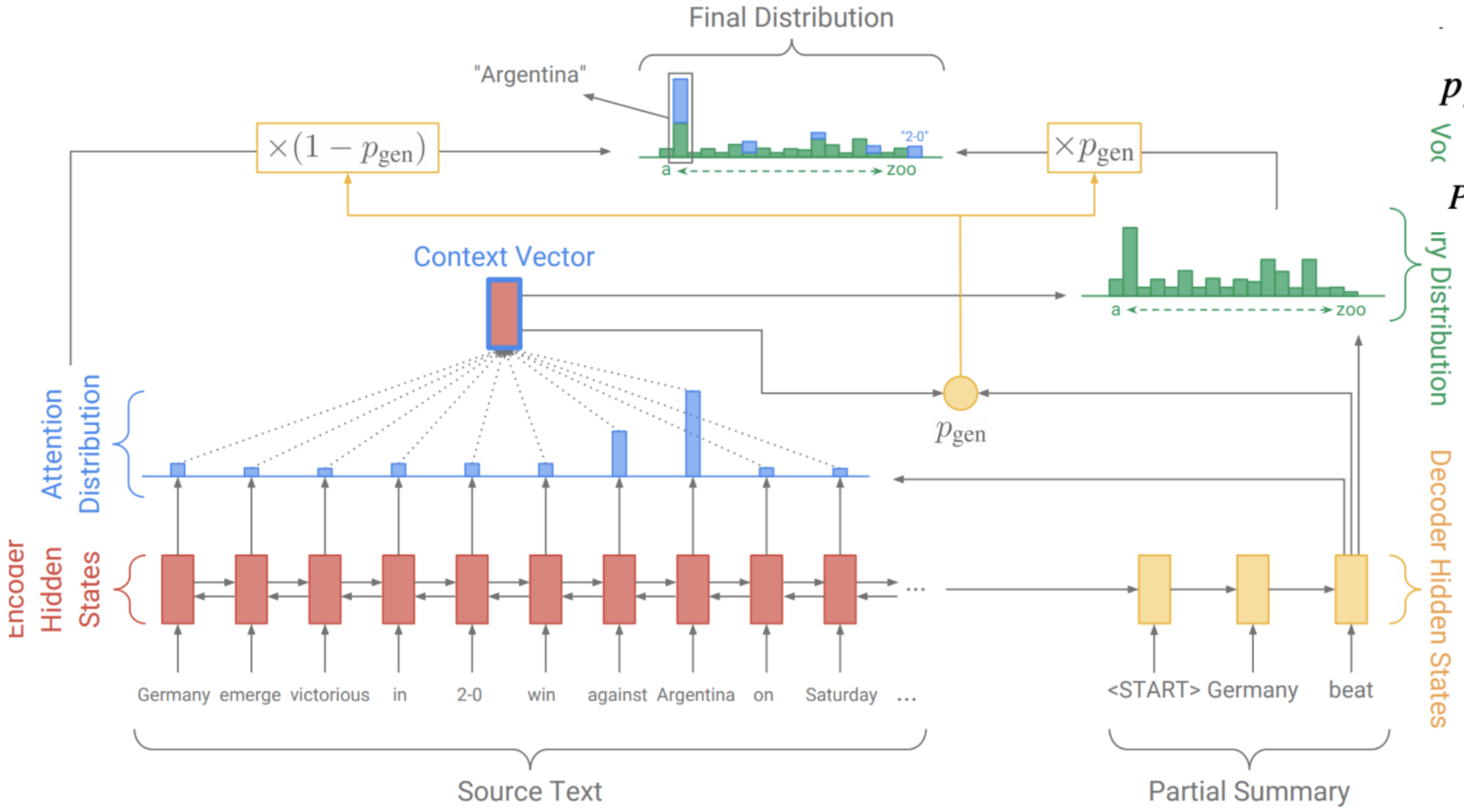
```python
def abstract2ids(abstract_words, vocab, source_oovs):
    """Map tokens in the abstract (reference) to ids.
       OOV tokens in the source will be remained.

    Args:
        abstract_words (list): Tokens in the reference.
        vocab (vocab.Vocab): The vocabulary.
        source_oovs (list): OOV tokens in the source.

    Returns:
        list: The reference with tokens mapped into ids.
    """
    ###########################################
    #           TODO: module 1 task 1         #
    ###########################################
    ids = []
    unk_id = vocab.UNK
    for w in abstract_words:
        i = vocab[w]
        if i == unk_id:  # If w is an OOV word
            if w in source_oovs:  # If w is an in-source OOV
                # Map to its temporary source OOV number
                vocab_idx = vocab.size() + source_oovs.index(w)
                ids.append(vocab_idx)
            else:  # If w is an out-of-source OOV
                ids.append(unk_id)  # Map to the UNK token id
        else:
            ids.append(i)
    return ids
```

模块1: OOV tokens处理

model/dataset.py:
任务2: 完成SampleDataset类中的
__getitem__函数。
任务1完成的abstract2ids函数来对Y
进 处理。

```python
def __getitem__(self, index):
    ##########################################
    #           TODO: module 1 task 2          #
    ##########################################
    x, oov = source2ids(self.src_sents[index], self.vocab)
    return {
        'x': [self.vocab.SOS] + x + [self.vocab.EOS],
        'OOV': oov,
        'len_OOV': len(oov),
        'y': [self.vocab.SOS] + abstract2ids(self.trg_sents[index], self.vocab, oov) + [self.vocab.EOS],
        'x_len': len(self.src_sents[index]),
        'y_len': len(self.trg_sents[index])
    }
```

模块2：实现PGN和coverage



$$p_{gen} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{ptr}) \quad (8)$$

$$P(w) = p_{gen} P_{vocab}(w) + (1 - p_{gen}) \sum_{i:w_i=w} a_i^t \quad (9)$$

What if OOV ?

Considering a word w may appear multiple times in the input sequence S, we define the *output distribution* of word w by summing probability mass from all corresponding parts of the attention distribution, as in [38]:

$$p(y_t = w|S, y_{<t}) = \sum_{i:w_i=w} a_{ti}$$

模块2：实现PGN和coverage

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}) \quad (8)$$

通过一个 sigmoid 函数的来计算一个阈值以决定给 pointer 和 generator 的输出各自分配多大的权重，有点类似于 LSTM 或者 GRU 中的门机制。
这个门的输入是context vector 、Decoder 当前 time step 的隐状态和输入

模块2：实现PGN和coverage

model/model.py:
任务1: 完成Decoder。
1. 定义 个线性层w_gen。
2. 实现p_gen的计算，详 公式(8)。

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}) \qquad (8)$$

```python
class Decoder(nn.Module):
    def __init__(self,
                 vocab_size,
                 embed_size,
                 hidden_size,
                 enc_hidden_size=None,
                 is_cuda=True):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.DEVICE = torch.device('cuda') if is_cuda else torch.device('cpu')
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size

        self.lstm = nn.LSTM(embed_size, hidden_size, batch_first=True)

        self.W1 = nn.Linear(self.hidden_size * 3, self.hidden_size)
        self.W2 = nn.Linear(self.hidden_size, vocab_size)

        ###########################################
        #          TODO: module 2 task 1.1        #
        ###########################################
        if config.pointer:
            self.w_gen = nn.Linear(self.hidden_size * 4 + embed_size, 1)
```

模块2：实现PGN和coverage

model/model.py:

任务1: 完成Decoder。

1. 定义 个线性层w_gen。
2. 实现p_gen的计算，详 公式(8)。

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}) \quad (8)$$

```python
        ###########################################
        #          TODO: module 2 task 1.2        #
        ###########################################
        p_gen = None
        if config.pointer:
            # Calculate p_gen.
            # Refer to equation (8).
            x_gen = torch.cat([
                context_vector,
                s_t.squeeze(0),
                decoder_emb.squeeze(1)
            ],
                dim=-1)
            p_gen = torch.sigmoid(self.w_gen(x_gen))

        return p_vocab, decoder_states, p_gen
```

模块2：实现PGN和coverage

任务2: 完成Attention。

1. 定义 个线性层w_c。
2. 定义前向传导。

a. 计算attention weights时加
   coverage vector，参考公式(11)。

b. 对coverage vector进 更新，参考公式(10)

```
class Attention(nn.Module):
    def __init__(self, hidden_units):
        super(Attention, self).__init__()
        # Define feed-forward layers.
        self.Wh = nn.Linear(2*hidden_units, 2*hidden_units, bias=False)
        self.Ws = nn.Linear(2*hidden_units, 2*hidden_units)

        ###########################################
        #          TODO: module 2 task 2.1        #
        ###########################################
        # wc for coverage feature
        self.wc = nn.Linear(1, 2*hidden_units, bias=False)
        self.v = nn.Linear(2*hidden_units, 1, bias=False)
```

模块2：实现PGN和coverage

任务2: 完成Attention。
1. 定义 个线性层w_c。
2. 定义前向传导。
a. 计算attention weights时加
 coverage vector，参考公式(11)。
b. 对coverage vector进 更新，参考公式(10)

problem. In our coverage model, we maintain a *coverage vector* c^t, which is the sum of attention distributions over all previous decoder timesteps:

$$c^t = \sum_{t'=0}^{t-1} a^{t'} \tag{10}$$

The coverage vector is used as extra input to the attention mechanism, changing equation (1) to:

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + w_c c_i^t + b_{\text{attn}}) \tag{11}$$

模块2：实现PGN和coverage

任务2: 完成Attention。

1. 定义 个线性层w_c。
2. 定义前向传导。

a. 计算attention weights时加 coverage vector，参考公式(11)。

b. 对coverage vector进 更新，参考公式(10)

```
##########################################
#          TODO: module 2 task 2.2       #
##########################################
# Add coverage feature.
if config.coverage:
    coverage_features = self.wc(coverage_vector.unsqueeze(2))  # wc c
    att_inputs = att_inputs + coverage_features
```

problem. In our coverage model, we maintain a *coverage vector* c^t, which is the sum of attention distributions over all previous decoder timesteps:

$$c^t = \sum_{t'=0}^{t-1} a^{t'} \qquad (10)$$

The coverage vector is used as extra input to the attention mechanism, changing equation (1) to:

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + w_c c_i^t + b_{\text{attn}}) \qquad (11)$$

```
##########################################
#          TODO: module 2 task 2.3       #
##########################################
# Update coverage vector.
if config.coverage:
    coverage_vector = coverage_vector + attention_weights

return context_vector, attention_weights, coverage_vector
```

模块2：实现PGN和coverage

任务3：实现 个get_final_distribution函数。

先对 P_vocab 进 扩展，将 source 中的 oov 添加到 P_vocab 的尾部，得到 P_vocab_extend

这样 attention weights 中的每 个 token 都能在 P_vocab_extend 中找到对应的位置，然后 将对应的 attention weights 叠加到扩展后的 P_vocab_extend 中的对 应位置，得到 final distribution。

$$P(w) = p_{\text{gen}} P_{\text{vocab}}(w) + (1 - p_{\text{gen}}) \sum_{i:w_i=w} a_i^t \quad (9)$$

```python
###########################################
#          TODO: module 2 task 3          #
###########################################

if not config.pointer:
    return p_vocab

batch_size = x.size()[0]
# Clip the probabilities.
p_gen = torch.clamp(p_gen, 0.001, 0.999)
# Get the weighted probabilities.
# Refer to equation (9).
p_vocab_weighted = p_gen * p_vocab
# (batch_size, seq_len)
attention_weighted = (1 - p_gen) * attention_weights

# Get the extended-vocab probability distribution
# extended_size = len(self.v) + max_oovs
extension = torch.zeros((batch_size, max_oov)).float().to(self.DEVICE)
# (batch_size, extended_vocab_size)
p_vocab_extended = torch.cat([p_vocab_weighted, extension], dim=1)

# Add the attention weights to the corresponding vocab positions.
# Refer to equation (9).
final_distribution = \
    p_vocab_extended.scatter_add_(dim=1,
                                  index=x,
                                  src=attention_weighted)

return final_distribution
```

模块2：实现PGN和coverage

任务4: 完成整个model的前向传导。'

这 的关键是要实现coverage loss，详 公式(12)(13)。

We find it necessary (see section 5) to additionally define a *coverage loss* to penalize repeatedly attending to the same locations:

$$\text{covloss}_t = \sum_i \min(a_i^t, c_i^t) \qquad (12)$$

ing repeated attention. Finally, the coverage loss, reweighted by some hyperparameter λ, is added to the primary loss function to yield a new composite loss function:

$$\text{loss}_t = -\log P(w_t^*) + \lambda \sum_i \min(a_i^t, c_i^t) \qquad (13)$$

模块2：实现PGN和coverage

```python
def forward(self, x, x_len, y, len_oovs, batch, num_batches):
    """Define the forward propagation for the model.

    Args:
        x (Tensor):
            Input sequences as source with shape (batch_size, seq_len)
        x_len ([int): Sequence length of the current batch.
        y (Tensor):
            Input sequences as reference with shape (bacth_size, y_len)
        len_oovs (Tensor):
            The numbers of out-of-vocabulary words for samples in this batch.
        batch (int): The number of the current batch.
        num_batches(int): Number of batches in the epoch.

    Returns:
        batch_loss (Tensor): The average loss of the current batch.
    """

    x_copy = replace_oovs(x, self.v)
    x_padding_masks = torch.ne(x, 0).byte().float()
    encoder_output, encoder_states = self.encoder(x_copy)
    # Reduce encoder hidden states.
    decoder_states = self.reduce_state(encoder_states)
    # Initialize coverage vector.
    coverage_vector = torch.zeros(x.size()).to(self.DEVICE)
```

模块2：实现PGN和coverage

```
# Calculate loss for every step.
step_losses = []
for t in range(y.shape[1]-1):

    # Do teacher forcing.
    x_t = y[:, t]
    x_t = replace_oovs(x_t, self.v)

    y_t = y[:, t+1]
    # Get context vector from the attention network.
    context_vector, attention_weights, coverage_vector = \
        self.attention(decoder_states,
                       encoder_output,
                       x_padding_masks,
                       coverage_vector)
    # Get vocab distribution and hidden states from the decoder.
    p_vocab, decoder_states, p_gen = self.decoder(x_t.unsqueeze(1),
                                                  decoder_states,
                                                  context_vector)

    final_dist = self.get_final_distribution(x,
                                             p_gen,
                                             p_vocab,
                                             attention_weights,
                                             torch.max(len_oovs))

    # Get the probabilities predict by the model for target tokens.
    if not config.pointer:
        y_t = replace_oovs(y_t, self.v)
    target_probs = torch.gather(final_dist, 1, y_t.unsqueeze(1))
    target_probs = target_probs.squeeze(1)
```

模块2：实现PGN和coverage

```
# Apply a mask such that pad zeros do not affect the loss
mask = torch.ne(y_t, 0).byte()
# Do smoothing to prevent getting NaN loss because of log(0).
loss = -torch.log(target_probs + config.eps)

if config.coverage:
    # Add coverage loss.
    ct_min = torch.min(attention_weights, coverage_vector)
    cov_loss = torch.sum(ct_min, dim=1)
    loss = loss + config.LAMBDA * cov_loss

mask = mask.float()
loss = loss * mask

step_losses.append(loss)
```

模块2：实现PGN和coverage

```python
sample_losses = torch.sum(torch.stack(step_losses, 1), 1)
# get the non-padded length of each sequence in the batch
seq_len_mask = torch.ne(y, 0).byte().float()
batch_seq_len = torch.sum(seq_len_mask, dim=1)

# get batch loss by dividing the loss of each batch
# by the target sequence length and mean
batch_loss = torch.mean(sample_losses / batch_seq_len)
return batch_loss
```

Beam Search 实现

完成best_k函数。这个做的事情与greedy search很接近，不过要选出最好的k个token，然后扩展出k个新的beam容器。

```
# use decoder to generate vocab distribution for the next token
decoder_input_t = torch.tensor(beam.tokens[-1]).reshape(1, 1)
decoder_input_t = decoder_input_t.to(self.DEVICE)

# Get context vector from attention network.
context_vector, attention_weights = \
    self.model.attention(beam.decoder_states,
                         encoder_output,
                         x_padding_masks)

# Replace the indexes of OOV words with the index of OOV token
# to prevent index-out-of-bound error in the decoder.
decoder_input_t = self.replace_oov(decoder_input_t)
p_vocab, decoder_states = self.model.decoder(decoder_input_t,
                                              beam.decoder_states,
                                              encoder_output,
                                              context_vector)

# Calculate log probabilities.
log_probs = torch.log(p_vocab.squeeze())
# Filter forbidden tokens.
if len(beam.tokens) == 1:
    forbidden_ids = [
        self.vocab[u"这"],
        self.vocab[u"此"],
        self.vocab[u"采用"],
        self.vocab[u"，"],
        self.vocab[u"。"],
        self.vocab.UNK
    ]
    log_probs[forbidden_ids] = -float('inf')
```

Beam Search 实现

完成best_k函数。这个做的事情与greedy search很接近，不过要选出最好的k个token，然后扩展出k个新的beam容器。

```
# Get top k tokens and the corresponding logprob.
topk_probs, topk_idx = torch.topk(log_probs, k)

# Extend the current hypo with top k tokens, resulting k new hypos.
best_k = [beam.extend(x,
          log_probs[x],
          decoder_states,
          attention_weights,
          beam.max_oovs,
          beam.encoder_input) for x in topk_idx.tolist()]

return best_k
```

Beam Search 实现

完成best_k函数。这个做的事情与greedy search很接近，不过要选出最好的k个token，然后扩展出k个新的beam容器。

```
# Get top k tokens and the corresponding logprob.
topk_probs, topk_idx = torch.topk(log_probs, k)

# Extend the current hypo with top k tokens, resulting k new hypos.
best_k = [beam.extend(x,
          log_probs[x],
          decoder_states,
          attention_weights,
          beam.max_oovs,
          beam.encoder_input) for x in topk_idx.tolist()]

return best_k
```

Beam Search 实现

完成beam search函数。初始化encoder、attention和decoder的输□，然后对于每□个decode step，对于现有的k个beam，我们分别利□best_k函数来得到各□最佳的k个extended beam，也就是每个decode step我们会得到k*k个新的beam，然后只保留分数最□的k个，作为下□轮需要扩展的k个beam。为了只保留分数最□的k个beam，我们可以□□个堆(heap)来实现，堆的中只
保存k个节点，根结点保存分数最低的beam

```
# run body_sequence input through encoder
encoder_output, encoder_states = self.model.encoder(encoder_input)

# initialize decoder states with encoder forward states
decoder_states = self.model.reduce_state(encoder_states)

# initialize the hypothesis with a class Beam instance.
attention_weights = torch.zeros(
    (1, encoder_input.shape[1])).to(self.DEVICE)

init_beam = Beam([self.vocab.SOS],
                 [0],
                 decoder_states,
                 attention_weights,
                 max_oovs,
                 encoder_input)

# get the beam size and create a list for stroing current candidates
# and a list for completed hypothesis
k = beam_width
curr, completed = [init_beam], []
```

Beam Search 实现

完成beam search函数。初始化encoder、attention和decoder的输□，然后对于每□个decode step，对于现有的k个beam，我们分别利□best_k函数来得到各□最佳的k个extended beam，也就是每个decode step我们会得到k*k个新的beam，然后只保留分数最□的k个，作为下□轮需要扩展的k个beam。为了只保留分数最□的k个beam，我们可以□□个堆(heap)来实现，堆的中只保存k个节点，根结点保存分数最低的beam

```python
# use beam search for max_sum_len (maximum length) steps
for _ in range(max_sum_len):
    # get k best hypothesis when adding a new token

    topk = []
    for beam in curr:
        # When an EOS token is generated, add the hypo to the completed
        # list and decrease beam size.
        if beam.tokens[-1] == self.vocab.EOS:
            completed.append(beam)
            k -= 1
            continue
        for can in self.best_k(beam,
                               k,
                               encoder_output,
                               x_padding_masks):
            # Using topk as a heap to keep track of top k candidates.
            # Using the sequence scores of the hypos to campare
            # and object ids to break ties.
            add2heap(topk, (can.seq_score(), id(can), can), k)

    curr = [items[2] for items in topk]
    # stop when there are enough completed hypothesis
    if len(completed) == k:
        break
# When there are not engouh completed hypotheses,
# take whatever when have in current best k as the final candidates.
completed += curr
# sort the hypothesis by normalized probability and choose the best one
result = sorted(completed,
                key=lambda x: x.seq_score(),
                reverse=True)[0].tokens
return result
```

模块3: Beam Search优化

- model/utils.py

任务1: 实现length normalization和coverage normalization。

这部分请在Beam类下的seq_score函数中实现。

Length normalization

Scores are normalized by the following formula as defined in Wu et al. (2016):

$$lp(Y) = \frac{(5 + |Y|)^\alpha}{(5 + 1)^\alpha}$$

where $|Y|$ is the current target length and α is the length normalization coefficient `-length_norm`.

Coverage normalization

Scores are penalized by the following formula as defined in Wu et al. (2016):

$$cp(X, Y) = \beta \sum_{i=1}^{|X|} \log(\min(\sum_{j=1}^{|Y|} p_{i,j}, 1.0))$$

where $p_{i,j}$ is the attention probability of the j-th target word y_j on the i-th source word x_i, $|X|$ is the source length, $|Y|$ is the current target length and β is the coverage normalization coefficient `-coverage_norm`.

```python
###########################################
#          TODO: module 3 task 1          #
###########################################
len_Y = len(self.tokens)

# Lenth normalization
ln = (5 + len_Y) ** config.alpha / (5 + 1) ** config.alpha
cn = config.beta * torch.sum(  # Coverage normalization
    torch.log(
        config.eps +
        torch.where(
            self.coverage_vector < 1.0,
            self.coverage_vector,
            torch.ones((1, self.coverage_vector.shape[1])).to(torch.device(config.DEVICE))
        )
    )
)

score = sum(self.log_probs) / ln + cn
return score
```

模块3: Beam Search优化

model/predict.py
任务2: 实现EOS token normalization， 并选择□些禁□词汇。
这□部分请在best_k函数中实现。

End of sentence normalization

The score of the end of sentence token is penalized by the following formula:

$$ep(X, Y) = \gamma \frac{|X|}{|Y|}$$

where $|X|$ is the source length, $|Y|$ is the current target length and γ is the end of sentence normalization coefficient `-eos_norm`.

```python
###########################################
#           TODO: module 3 task 2         #
###########################################
# Filter forbidden tokens.
if len(beam.tokens) == 1:
    forbidden_ids = [
        self.vocab[u"这"],
        self.vocab[u"此"],
        self.vocab[u"采用"],
        self.vocab[u"，"],
        self.vocab[u"。"],
    ]
    log_probs[forbidden_ids] = -float('inf')
# EOS token penalty. Follow the definition in
# https://opennmt.net/OpenNMT/translation/beam_search/.
log_probs[self.vocab.EOS] *= config.gamma * x.size()[1] / len(beam.tokens)

log_probs[self.vocab.UNK] = -float('inf')
# Get top k tokens and the corresponding logprob.
topk_probs, topk_idx = torch.topk(log_probs, k)
```