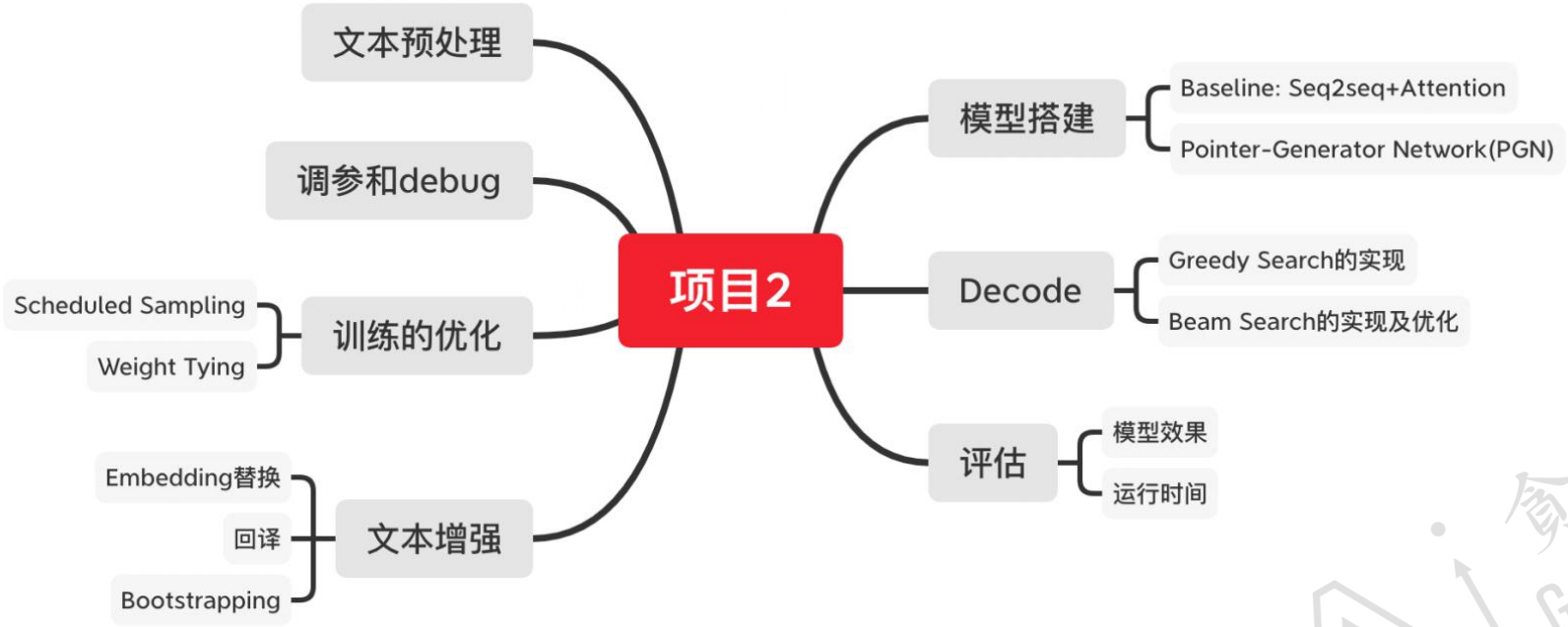
项目 2：基于京东电商的营销文本生成



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[3] 的模型作为 baseline。
- 然后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 1:
对于本次任务，需要完成如下的部分：

- 第一：构建 Seq2seq+Attention 的模型
- 第二：实现训练模块。
- 第三：实现预测模块，包括 Greedy Search 和 Beam Search。
- 第四：实现评估模块。

模块1: 词典处理

model/vocab.py:

任务1: 完成add_words函数。

- 向词典里加入一个新词，需要完成对word2index、index2word和word2count三个变量的更新。

```
def add_words(self, words):
    """Add a new token to the vocab and do mapping between word and index.

    Args:
        words (list): The list of tokens to be added.
    """
    ##########################################
    #          TODO: module 1 task 1         #
    ##########################################
    for word in words:
        if word not in self.word2index:
            self.word2index[word] =len(self.index2word)
            self.index2word.append(word)

    self.word2count.update(words)
```

模块1: 词典处理

任务2: 完成PairDataset类中的build_vocab函数。
需要实现控制数据集词典的大小（从config.max_vocab_size）读取这一参数。建议使用python的collection模块中的Counter来做，这个数据类型跟dict很像，但有两个好处：
1. 加入新的key时不需要判断是否存在，会自动将其对应的值初始化为0。
2. 可以通过most_common函数来获取数量最多的k个key。

```python
def build_vocab(self, embed_file: str = None) -> Vocab:
    """Build the vocabulary for the data set.

    Args:
        embed_file (str, optional):
        The file path of the pre-trained embedding word vector.
        Defaults to None.

    Returns:
        vocab.Vocab: The vocab object.
    """
    # word frequency
    word_counts = Counter()
    count_words(word_counts,
                [src + tgr for src, tgr in self.pairs])
    vocab = Vocab()
    ##########################################
    #          TODO: module 1 task 2         #
    ##########################################

    # Filter the vocabulary by keeping only the top k tokens in terms of
    # word freguncy in the data set, where k is the maximum vocab size set
    # in "config.py".
    for word, count in word_counts.most_common(config.max_vocab_size):
        vocab.add_words([word])

    if embed_file is not None:
        count = vocab.load_embeddings(embed_file)
        print("%d pre-trained embeddings loaded." % count)

    return vocab
```

模块1: 词典处理

model/utils.py:

任务3: 完成source2ids函数。

当我们训练好模型要对测试集进行测试时，测试集中的样本往往会包含OOV tokens。这个函数需要你将在词典中的token映射到相应的index，对于oov tokens则需要记录下来并返回。

```python
def source2ids(source_words, vocab):
    """Map the source words to their ids and return a list of OOVs in the source.
    Args:
        source_words: list of words (strings)
        vocab: Vocabulary object
    Returns:
        ids:
        A list of word ids (integers); OOVs are represented by their temporary
        source OOV number. If the vocabulary size is 50k and the source has 3
        OOVs tokens, then these temporary OOV numbers will be 50000, 50001,
        50002.
    oovs:
        A list of the OOV words in the source (strings), in the order
        corresponding to their temporary source OOV numbers.
    """

    ##########################################
    #          TODO: module 1 task 3         #
    ##########################################

    ids = []
    oovs = []
    unk_id = vocab["<UNK>"]

    for word in source_words:
        i = vocab[word]
        if i == unk_id:  # If w is OOV
            if word not in oovs:   # Add to list of OOVs
                oovs.append(word)
            # This is 0 for the first source OOV, 1 for the second source OOV
            oov_num = oovs.index(word)
            # This is e.g. 20000 for the first source OOV, 50001 for the second
            ids.append(vocab.size()+oov_num)
        else:
            ids.append(i)
    return ids, oovs
```

模块1: 词典处理

任务4: 完成outputids2words函数。与任务3相反的过程，将token id映射到对应的词，并输出。

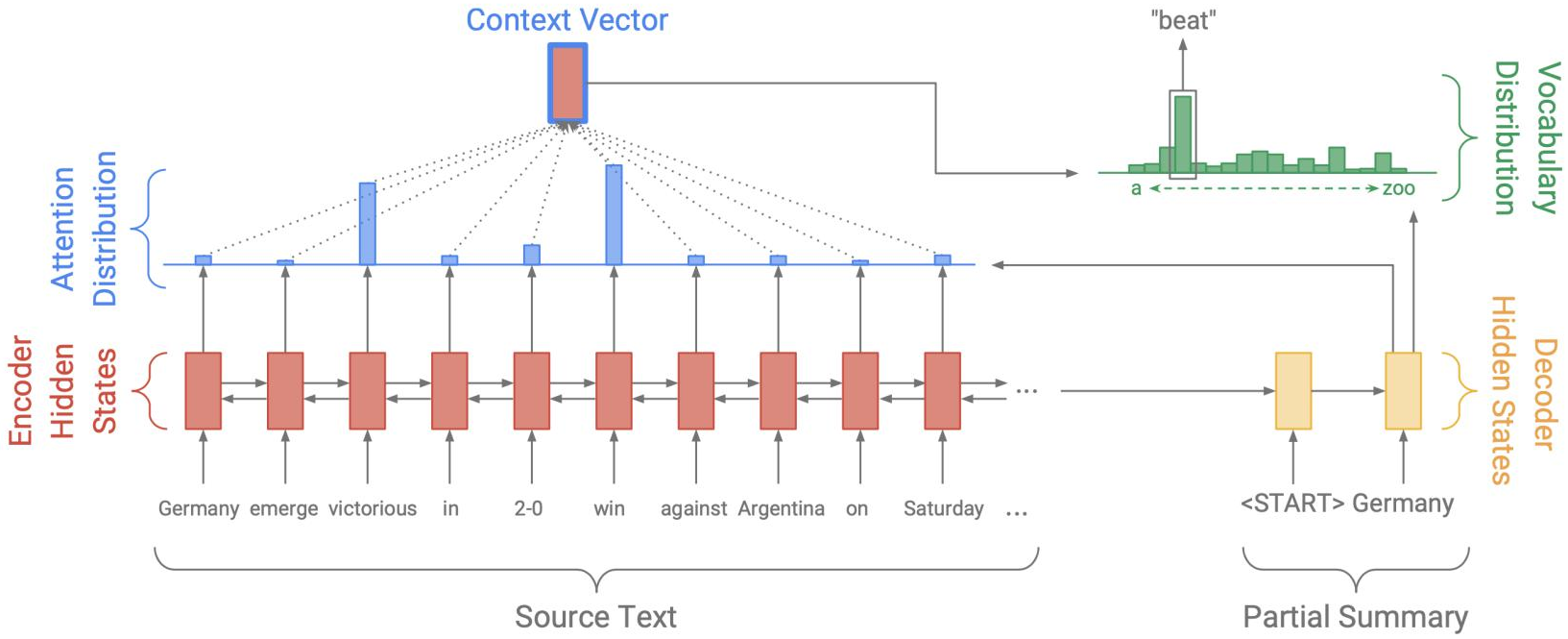
```python
def outputids2words(id_list, source_oovs, vocab):
    """
        Maps output ids to words, including mapping in-source OOVs from
        their temporary ids to the original OOV string (applicable in
        pointer-generator mode).
        Args:
            id_list: list of ids (integers)
            vocab: Vocabulary object
            source_oovs:
                list of OOV words (strings) in the order corresponding to
                their temporary source OOV ids (that have been assigned in
                pointer-generator mode), or None (in baseline mode)
        Returns:
            words: list of words (strings)
    """

    ##########################################
    #          TODO: module 1 task 4         #
    ##########################################
    words = []
    for i in id_list:
        try:
            w = vocab.index2word[i]  # might be [UNK]
        except IndexError: #w is OOV
            assert_msg = "ERROR ID can't find"
            assert source_oovs is not None, assert_msg
            source_oov_idx = i - vocab.size()
            try:
                w = source_oovs[source_oov_idx]
            except ValueError: # i doesn't correspond to an source oov
                raise ValueError("ERROR ID can't find OOV")
        words.append(w)
    return ' '.join(words)
```

模块2: 模型搭建

model/model.py:

任务1: 完成Encoder。

1. 定义embedding层和BiLSTM层。
2. 实现前向传导（输入输出详见代码）。



模块2: 模型搭建

model/model.py:

任务1: 完成Encoder。

1. 定义embedding层和BiLSTM层。

```
class Encoder(nn.Module):
    def __init__(self,
                 vocab_size,
                 embed_size,
                 hidden_size,
                 rnn_drop: float = 0):
        ########################################
        #          TODO: module 2 task 1.1       #
        ########################################
        super(Encoder, self).__init__()

        self.hidden_size = hidden_size
        self.embedding =nn.Embedding(vocab_size,embed_size)
        self.lstm =nn.LSTM(embed_size,hidden_size,bidirectional=True,dropout=rnn_drop,batch_first=True)
```

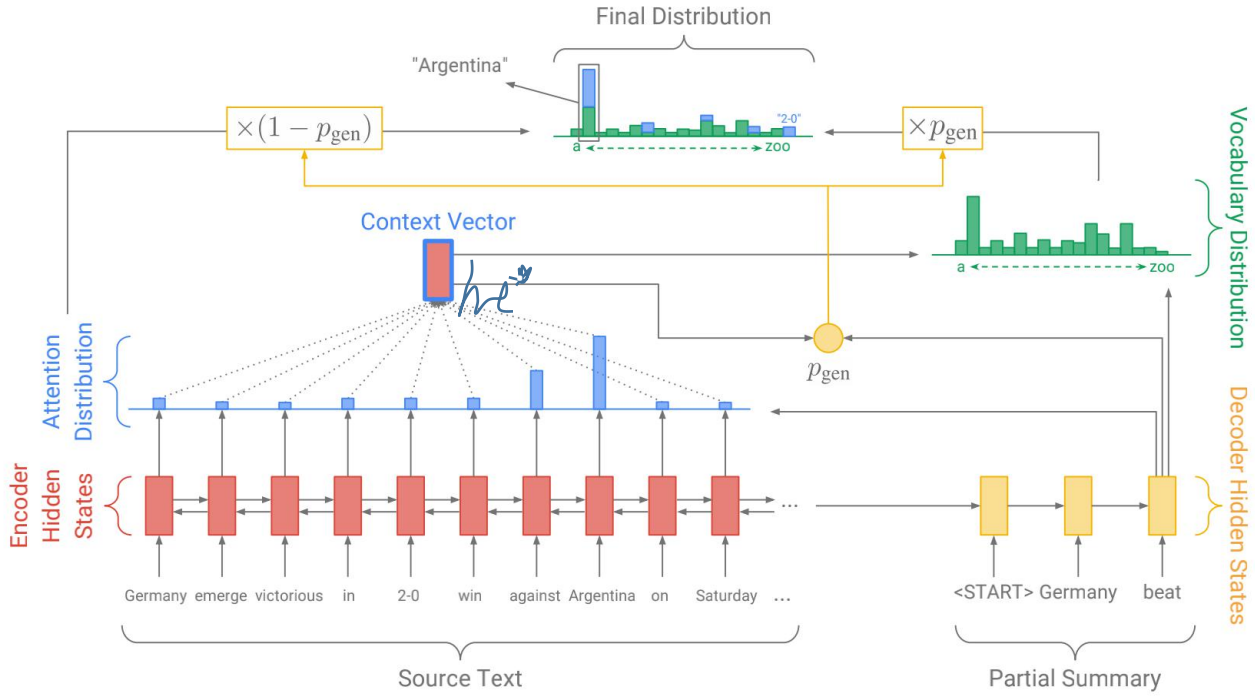
模块2: 模型搭建

model/model.py:
任务1: 完成Encoder。
2. 实现前向传导（输入输出详见代码）。

```python
def forward(self, x):
    """Define forward propagation for the endoer.

    Args:
        x (Tensor): The input samples as shape (batch_size, seq_len).

    Returns:
        output (Tensor):
            The output of lstm with shape
            (batch_size, seq_len, 2 * hidden_units).
        hidden (tuple):
            The hidden states of lstm (h_n, c_n).
            Each with shape (2, batch_size, hidden_units)
    """
    ############################################
    #          TODO: module 2 task 1.2         #
    ############################################
    embedded = self.embedding(x)
    output, hidden = self.lstm(embedded)

    return output, hidden
```

模块2: 模型搭建

任务2: 完成Decoder。
1. 定义embedding层和LSTM层（单向）； 定义两个线性层（前馈层）W1和W2。
2. 实现前向传导。具体实现参见论文中的公式(4)和(5)。代码中会给出每一个步骤的提示。



$$h_t^* = \sum_i a_i^t h_i \qquad (3)$$

$$P_{\text{vocab}} = \text{softmax}(V'(V[s_t, h_t^*] + b) + b') \qquad (4)$$

$$P(w) = P_{\text{vocab}}(w) \qquad (5)$$

模块2: 模型搭建

任务2: 完成Decoder。
1. 定义embedding层和LSTM层（单向）； 定义两个线性层（前馈层）W1和W2。

```
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
        ########################################
        #          TODO: module 2 task 2.1       #
        ########################################

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

        #最终Decoder部分结合Encoder的输入状态，Context向量，以及Decoder的历史输入
        self.W1 = nn.Linear(3 * self.hidden_size, self.hidden_size)
        self.W2 = nn.Linear(self.hidden_size, vocab_size)
```

$$h_t^* = \sum_i a_i^t h_i \qquad (3)$$

$$P_{\text{vocab}} = \text{softmax}(V'(V[s_t, h_t^*] + b) + b') \qquad (4)$$

$$P(w) = P_{\text{vocab}}(w) \qquad (5)$$

模块2: 模型搭建

任务2: 完成Decoder。
2. 实现前向传导。具体实现参见论文中的公式(4)和(5)。代码中会给出每一个步骤的提示。

```
    ##########################################
    #          TODO: module 2 task 2.2       #
    ##########################################

    decoder_emb = self.embedding(decoder_input)

    decoder_output, decoder_states = self.lstm(decoder_emb,decoder_states)

    # concatenate context vector and decoder state
    # (batch_size, 3*hidden_units)
    decoder_output =  decoder_output.view(-1,config.hidden_size)# Reshape.
    concat_vector =torch.cat([decoder_output,context_vector],dim=-1)

    # calculate vocabulary distribution
    # (batch_size, hidden_units)
    FF1_out = self.W1(concat_vector)
    # (batch_size, vocab_size)
    FF2_out = self.W2(FF1_out)
    # (batch_size, vocab_size)
    p_vocab = F.softmax(FF2_out,dim=1)

    return p_vocab, decoder_states
```

$$h_t^* = \sum_i a_i^t h_i \qquad (3)$$

$$P_{\text{vocab}} = \text{softmax}(V'(V[s_t, h_t^*] + b) + b') \qquad (4)$$

$$P(w) = P_{\text{vocab}}(w) \qquad (5)$$

模块2: 模型搭建

任务3: 完成Attention。
1. 定义三个线性层Wh、Ws和v。维度详见论文中的公式(1)和(2)。
2. 定义前向传导。
    a. 处理decoder的隐状态h和c，将二者拼接得到s_t，并处理成合理的shape。
    b. 参考论文中的公式(1)和(2)，实现attention weights的计算。
    c. 由于训练过程中会对batch中的样本进行padding，对于进行了padding的输入我们需要把填充的位置的attention weights给过滤掉（padding mask），然后对剩下位置的attention weights进行归一化。
    d. 根据论文中的公式(3)计算context vector (hint: 可以使用torch.bmm)。

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}}) \qquad (1)$$

$$a^t = \text{softmax}(e^t) \qquad (2)$$

$$h_t^* = \sum_i a_i^t h_i \qquad (3)$$

模块2: 模型搭建

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}}) \quad (1)$$

$$a^t = \text{softmax}(e^t) \quad (2)$$

任务3: 完成Attention。

1. 定义三个线性层Wh、Ws和v。维度详见论文中的公式(1)和(2)。

```
class Attention(nn.Module):
    def __init__(self, hidden_units):
        super(Attention, self).__init__()
        ##########################################
        #           TODO: module 2 task 3.1      #
        ##########################################
        self.Wh = nn.Linear(2 * hidden_units, 2 * hidden_units, bias=False)  #
        self.Ws = nn.Linear(2 * hidden_units, 2 * hidden_units,bias=False)  #
        self.v = nn.Linear(2 * hidden_units, 1, bias=False)
```

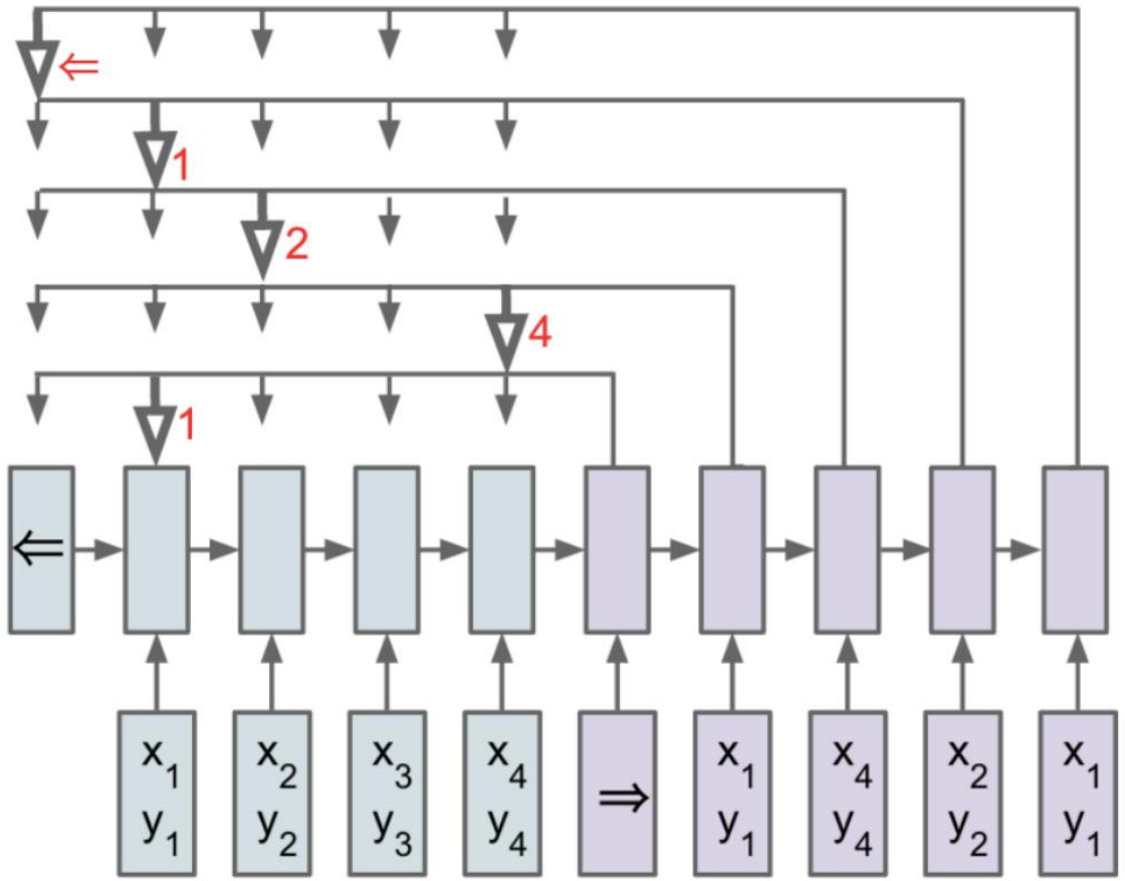
模块2: 模型搭建

任务3: 完成Attention。
2. 定义前向传导。
a. 处理decoder的隐状态h和c，将二者拼接得到s_t，并处理成合理的shape。
b. 参考论文中的公式(1)和(2)，实现attention weights的计算。
c. 由于训练过程中会对batch中的样本进行padding，对于进行了padding的输入我们需要把填充的位置的attention weights给过滤掉（padding mask），然后对剩下位置的attention weights进行归一化。
d. 根据论文中的公式(3)计算context vector (hint: 可以使用torch.bmm)。

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}}) \quad (1)$$

$$a^t = \text{softmax}(e^t) \quad (2)$$

$$h_t^* = \sum_i a_i^t h_i \quad (3)$$



(b) Ptr-Net

模块2: 模型搭建

任务3: 完成Attention。

2. 定义前向传导。

a. 处理decoder的隐状态h和c，将二者拼接得到s_t，并处理成合理的shape。

b. 参考论文中的公式(1)和(2)，实现attention weights的计算。

c. 由于训练过程中会对batch中的样本进行padding，对于进行了padding的输入我们需要把填充的位置的attention weights给过滤掉（padding mask），然后对剩下位置的attention weights进行归一化。

d. 根据论文中的公式(3)计算context vector (hint: 可以使用torch.bmm)。

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}}) \quad (1)$$

$$a^t = \text{softmax}(e^t) \quad (2)$$

$$h_t^* = \sum_i a_i^t h_i \quad (3)$$

```
##########################################
#          TODO: module 2 task 3.2       #
##########################################

# Concatenate h and c to get s_t and expand the dim of s_t.
h_dec, c_dec = decoder_states
# (1, batch_size, 2*hidden_units)
s_t = torch.cat([h_dec,c_dec],dim = 2)
# (batch_size, 1, 2*hidden_units)
s_t = s_t.transpose(0,1)
# (batch_size, seq_length, 2*hidden_units)
s_t = s_t.expand_as(encoder_output).contiguous()

# calculate attention scores
# Equation(11).
# Wh h_* (batch_size, seq_length, 2*hidden_units)
encoder_features = self.Wh(encoder_output.contiguous())
# Ws s_t (batch_size, seq_length, 2*hidden_units)
decoder_features = self.Ws(s_t)
# (batch_size, seq_length, 2*hidden_units)
att_inputs = encoder_features+decoder_features
# (batch_size, seq_length, 1)
score = self.v(torch.tanh(att_inputs))
# (batch_size, seq_length)
attention_weights = F.softmax(score, dim=1).squeeze(2)
attention_weights = attention_weights * x_padding_masks
# Normalize attention weights after excluding padded positions.
normalization_factor = attention_weights.sum(1, keepdim=True)
attention_weights = attention_weights / normalization_factor
# (batch_size, 1, 2*hidden_units)
context_vector = torch.bmm(attention_weights.unsqueeze(1),
                           encoder_output)
# (batch_size, 2*hidden_units)
context_vector = context_vector.squeeze(1)

return context_vector, attention_weights
```

模块2: 模型搭建

任务4: 完成整个model的前向传导。
1. 对输入序列x进行处理，对于oov的token，需要将他们的index转换成<UNK> token (hint: 可以使用torch.where)。
2. 生成输入序列x的padding mask (hint: 可以使用torch.ne)。
在实践中，为了 batch 训练，一般会把不定长的序列 padding 到相同长度，再用 mask 去区分非 padding 部分和 padding 部分。
3. 得到encoder的输出和隐状态，并对隐状态进行降维后作为decoder的初始隐状态。
4. 对于每一个time step，以输入序列y的y_t作为输入，y_t+1作为target，计算attention，然后用decoder得到p_vocab，找到target对应的词在p_vocab中对应的概率target_probs (hint: 可以使用torch.gather)，然后计算time step t的损失（NLL loss，详见论文公式(6)）。然后加上padding mask。
5. 计算整个序列的平均loss，详见论文公式(7)。
6. 计算整个batch的平均loss并返回。

$$\text{loss}_t = -\log P(w_t^*) \qquad (6)$$

$$\text{loss} = \frac{1}{T}\sum_{t=0}^{T} \text{loss}_t \qquad (7)$$

模块2: 模型搭建

任务4: 完成整个model的前向传导。
1. 对输入序列x进行处理，对于oov的token，需要将他们的index转换成<UNK> token (hint: 可以使用torch.where)。

```
oov_token = torch.full(x.shape, self.v.UNK).long().to(self.DEVICE)
x_copy = torch.where(x > len(self.v) - 1, oov_token, x)
```

模块2: 模型搭建

任务4: 完成整个model的前向传导。
2. 生成输入序列x的padding mask (hint: 可以使用torch.ne)。
在实践中，为了 batch 训练，一般会把不定长的序列 padding 到相同长度，再用 mask 去区分非 padding 部分和 padding 部分。

```
x_padding_masks = torch.ne(x_copy, 0).byte().float()
```

模块2: 模型搭建

任务4: 完成整个model的前向传导。

3. 得到encoder的输出和隐状态，并对隐状态进行降维后作为decoder的初始隐状态。

```
encoder_output, encoder_states = self.encoder(x_copy)
# Reduce encoder hidden states.
decoder_states = self.reduce_state(encoder_states)
```

模块2: 模型搭建

任务4: 完成整个model的前向传导。

4. 对于每一个time step,

- 以输入序列y的y_t作为输入，y_t+1作为target，计算attention，
- 然后用decoder得到p_vocab，找到target对应的词在p_vocab中对应的概率target_probs (hint: 可以使
- 用torch.gather)，
- 然后计算time step t的损失（NLL loss，详见论文公式(6)）。
- 然后加上padding mask。

$$\text{loss}_t = -\log P(w_t^*) \qquad (6)$$

```
# Calculate loss for every step.
step_losses = []
for t in range(y.shape[1]-1):
    decoder_input_t = y[:, t]  # x_t
    decoder_target_t = y[:, t+1]  # y_t
    # Get context vector from the attention network.
    context_vector, attention_weights = self.attention(
        decoder_states, encoder_output, x_padding_masks)
    # Get vocab distribution and hidden states from the decoder.
    p_vocab, decoder_states = self.decoder(
        decoder_input_t.unsqueeze(1), decoder_states, encoder_output,
        context_vector)

    # Get the probabilities predict by the model for target tokens.
    target_probs = torch.gather(p_vocab,
                                1,
                                decoder_target_t.unsqueeze(1))
    target_probs = target_probs.squeeze(1)
    # Apply a mask such that pad zeros do not affect the loss
    mask = torch.ne(decoder_target_t, 0).byte()
    # Do smoothing to prevent getting NaN loss because of log(0).
    loss = -torch.log(target_probs+config.eps)
    mask = mask.float()
    loss = loss * mask
    step_losses.append(loss)
```

模块2: 模型搭建

任务4: 完成整个model的前向传导。
5. 计算整个序列的平均loss，详见论文公式(7)。
6. 计算整个batch的平均loss并返回。

$$\text{loss} = \frac{1}{T}\sum_{t=0}^{T}\text{loss}_t \qquad (7)$$

```
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
```

模块3: 训练

model/train.py
任务1: 实现训练过程。
任务2: 在适当的位置实现梯度剪裁。
Hint: 参见torch.nn.utils模块下的clip_grad_norm_函数。
任务3: 用TensorboardX记录训练过程的损失并实现可视化。

模块3: 训练

model/train.py
任务1: 实现训练过程。

```
##########################################
#          TODO: module 1 task 2         #
##########################################

# Define the optimizer.
optimizer = optim.Adam(model.parameters(),
                       lr=config.learning_rate)
train_dataloader = DataLoader(dataset=train_data,
                              batch_size=config.batch_size,
                              shuffle=True,
                              collate_fn=collate_fn)
```

```
##########################################
#          TODO: module 3 task 1         #
##########################################
model.train()  # Sets the module in training mode.
optimizer.zero_grad()  # Clear gradients.
# Calculate loss.
loss = model(x, x_len, y, len_oovs, batch=batch)
batch_losses.append(loss.item())
loss.backward()  # Backpropagation.
```

模块3: 训练

model/train.py
任务2: 在适当的位置实现梯度剪裁。
Hint: 参见torch.nn.utils模块下的clip_grad_norm_函数。

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

# Do gradient clipping to prevent gradient explosion.
clip_grad_norm_(model.encoder.parameters(),
                config.max_grad_norm)
clip_grad_norm_(model.decoder.parameters(),
                config.max_grad_norm)
clip_grad_norm_(model.attention.parameters(),
                config.max_grad_norm)
optimizer.step()

# Update weights.
```

模块3: 训练

model/train.py

任务3: 用TensorboardX记录训练过程的损失并实现可视化。

```
writer = SummaryWriter(config.log_path)
```

模块4: 解码

model/predict.py:
任务1: 实现Greedy search。
这一块比较简单，跟着代码的提示，用encoder编码输入，传递每一个time step的信息给decoder，计算attention，得到decoder的p_vocab，根据p_vocab选出概率最大的词作为下一个token。

```
    ########################################
    #          TODO: module 4 task 1       #
    ########################################

    # Get encoder output and states.
    encoder_output, encoder_states = self.model.encoder(encoder_input)

    # Initialize decoder's hidden states with encoder's hidden states.
    decoder_states = self.model.reduce_state(encoder_states)

    # Initialize decoder's input at time step 0 with the SOS token.
    decoder_input_t = torch.ones(1) * self.vocab.SOS
    decoder_input_t = decoder_input_t.to(self.DEVICE, dtype=torch.int64)
    summary = [self.vocab.SOS]

    # Generate hypothesis with maximum decode step.
    while int(decoder_input_t.item()) != (self.vocab.EOS) \
            and len(summary) < max_sum_len:
        context_vector, attention_weights = \
            self.model.attention(decoder_states,
                                 encoder_output,
                                 x_padding_masks)
        p_vocab, decoder_states = \
            self.model.decoder(decoder_input_t.unsqueeze(1),
                               decoder_states,
                               encoder_output,
                               context_vector)
        # Get next token with maximum probability.
        decoder_input_t = torch.argmax(p_vocab, dim=1).to(self.DEVICE)
        decoder_word_idx = decoder_input_t.item()
        summary.append(decoder_word_idx)
        decoder_input_t = self.replace_oov(decoder_input_t)

    return summary
```

模块4: 解码

任务2: 实现Beam search。
1. 看懂Beam这一个类需要传递的变量（实现在model/utils.py中）。
2. 完成best_k函数。这里做的事情与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。
3. 完成beam search函数。初始化encoder、attention和decoder的输入，然后对于每一个decode step，对于现有的k个beam，我们分别利用best_k函数来得到各自最佳的k个extended beam，也就是每个decode step我们会得到k*k个新的beam，然后只保留分数最高的k个，作为下一轮需要扩展的k个beam。为了只保留分数最高的k个beam，我们可以用一个堆(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实现堆的方法详见https://docs.python.org/2/library/heapq.html。
4. Hint: 用heapq模块时，各种操作（push, pop）需要比较堆中元素的大小，如果以tuple的形式来存储，会从tuple的第一个位置开始比较，如果第一个位置的值相同，会继续比较后面的位置的值。所以建议以（分数，对象id，对象）三元组的形式来存储，其中对象id的作用是快速break ties。

模块4: 解码

任务2: 实现Beam search。

1. 看懂Beam这一个类需要传递的变量（实现在model/utils.py中）。

```
class Beam(object):
    """The contianer for a temperay sequence used in beam search.
    """
    def __init__(self,
                 tokens,
                 log_probs,
                 decoder_states,
                 attention_weights,
                 max_oovs,
                 encoder_input):
        self.tokens = tokens
        self.log_probs = log_probs
        self.decoder_states = decoder_states
        self.attention_weights = attention_weights
        self.max_oovs = max_oovs
        self.encoder_input = encoder_input
```

模块4: 解码

任务2: 实现Beam search。

2. 完成best_k函数。这里做的事情与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。

```
    ########################################
    #          TODO: module 4 task 2.2       #
    ########################################

    # use decoder to generate vocab distribution for the next token
    decoder_input_t =
    decoder_input_t = decoder_input_t.to(self.DEVICE)

    # Get context vector from attention network.
    context_vector, attention_weights =

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

    # Calculate log probabilities.
    log_probs =

    # Get top k tokens and the corresponding logprob.
    topk_probs, topk_idx =

    # Extend the current hypo with top k tokens, resulting k new hypos.
    best_k =

    return best_k
```

模块4: 解码

任务2: 实现Beam search。

3. 完成beam search函数。初始化encoder、attention和decoder的输入，然后对于每一个decode step，对于现有的k个beam，我们分别利用best_k函数来得到各自最佳的k个extended beam，也就是每个decode step我们会得到k*k个新的beam，然后只保留分数最高的k个，作为下一轮需要扩展的k个beam。为了只保留分数最高的k个beam，我们可以用一个堆(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