



基于规则的机器翻译



Rule-based machine translation

基于规则的机器翻译，是最古老也是见效最快的一种翻译方式。

大致流程：



根据翻译的方式可以分为：

- 直接基于词的翻译
- 结构转换的翻译
- 中间语的翻译

举个例子：

we do chicken right → we do chiken right → 我们 做 鸡 右 → 我们 做 鸡 右 → 我们做鸡右

基于统计的机器翻译

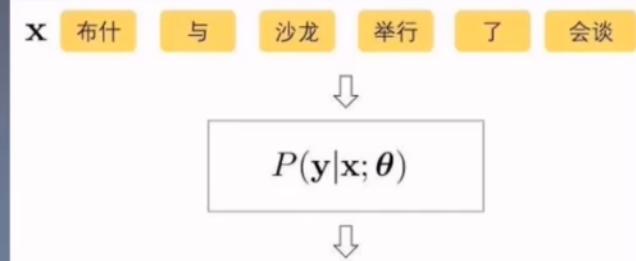


Statistical machine translation

基本思想：通过对大量的平行语料进行统计分析，构建统计翻译模型，进而使用此模型进行翻译。

核心问题：为翻译过程建立**概率模型**

大致流程：



隐变量：生产过程中不可观测的随机变量

隐变量对数线性模型：在隐式语言结构上设计特征

关键问题：如何设置**特征函数**

$$P(y|x; \theta) = \sum_z \frac{\exp(\theta \cdot \phi(x, y, z))}{\sum_{y'} \sum_{z'} \exp(\theta \cdot \phi(x, y', z'))}$$

(Och and Ney, 2002)

y Bush held a talk with Sharon

基于神经网络的机器翻译

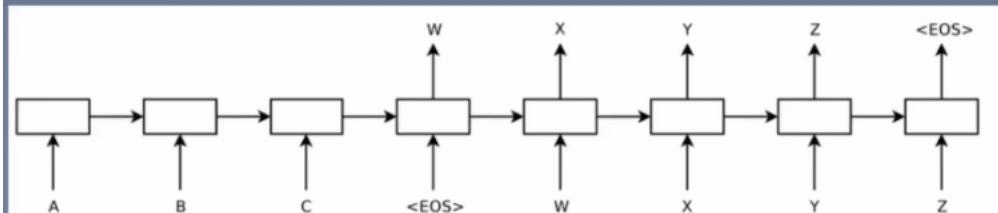
Neural Machine Translation

基于神经网络的机器翻译：通过学习大量成对的语料让神经网络自己学习语言的特征，找到输入和输出之间的关系。



2014 年时，Kyunghyun Cho 和 Sutskever 先后提出了一种 End-to-End 即所谓「端到端」的模型，直接对输入输出建立联系。前者将其模型命名为 Encoder-Decoder 模型，后者则将其命名为 Sequence-to-Sequence 模型。

核心思想：端到端 (End-to-End)



基本思想：利用神经网络实现自然语言的映射

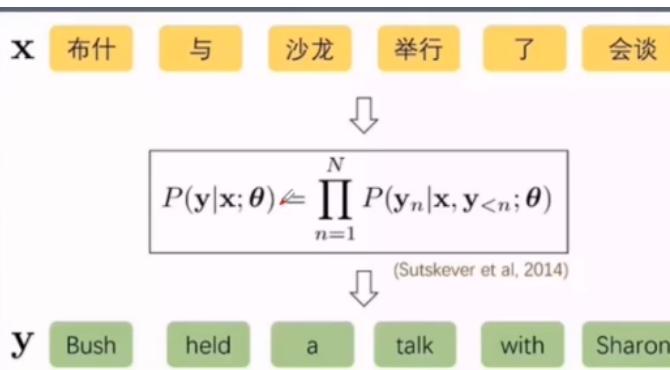
核心问题：条件概率建模

$$P(y_n | x, y_{<n}; \theta)$$

y_n 当前目标语言词

x 源语言句子

$y_{<n}$ 已经生成的目标语言句子



如何对条件概率进行建模

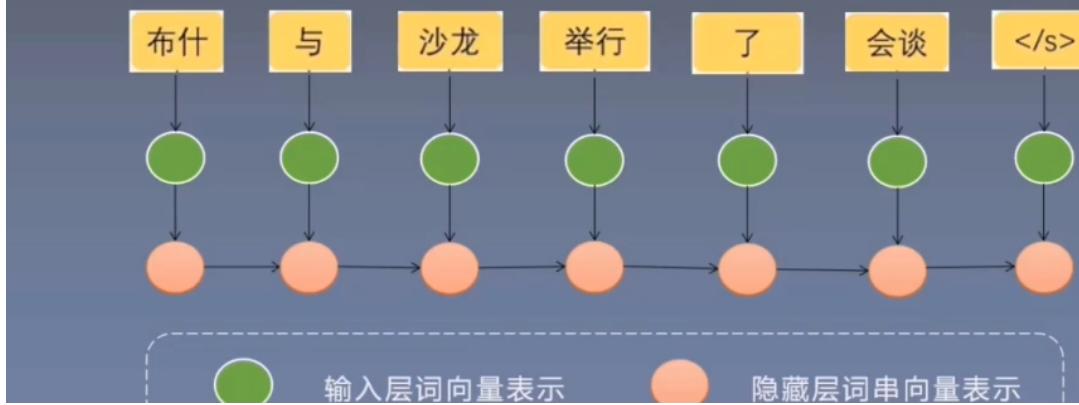
$$\begin{aligned} & P(\mathbf{y}_n | \mathbf{x}, \mathbf{y}_{<n}; \boldsymbol{\theta}) \\ &= \frac{\exp(\varphi(\mathbf{y}_n, \mathbf{x}, \mathbf{y}_{<n}, \boldsymbol{\theta}))}{\sum_{y \in \mathcal{Y}} \exp(\varphi(y, \mathbf{x}, \mathbf{y}_{<n}, \boldsymbol{\theta}))} \\ &= \frac{\exp(\varphi(\mathbf{v}_{\mathbf{y}_n}, \mathbf{c}_s, \mathbf{c}_t, \boldsymbol{\theta}))}{\sum_{y \in \mathcal{Y}} \exp(\varphi(\mathbf{v}_y, \mathbf{c}_s, \mathbf{c}_t, \boldsymbol{\theta}))} \end{aligned}$$

\mathbf{v}_y 目标语言词向量 \mathcal{Y} 目标语言词汇

\mathbf{c}_s 源语言上下文向量 \mathbf{c}_t 目标语言上下文向量

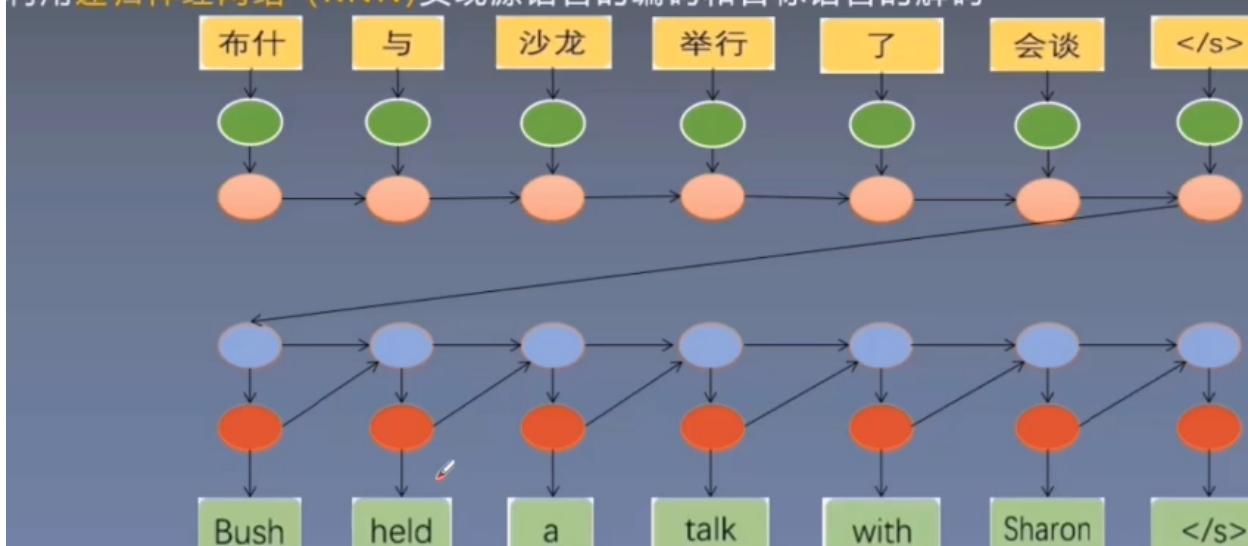
句子向量表示

利用递归神经网络 (RNN) 计算句子的向量表示



Encoder-Decoder

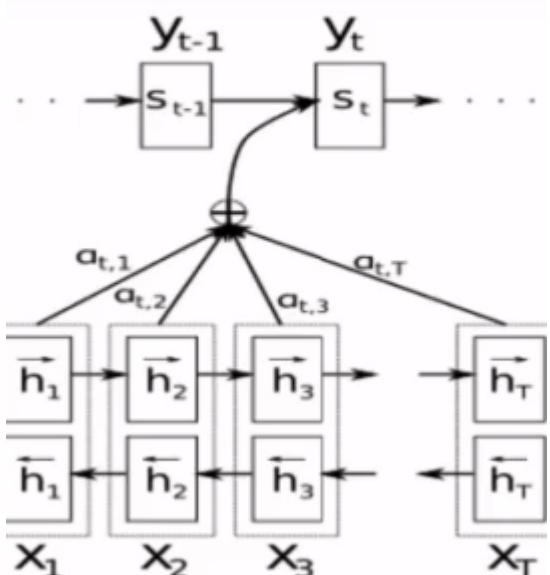
利用递归神经网络 (RNN) 实现源语言的编码和目标语言的解码



优点：利用长短记忆时记忆处理长距离依赖

缺点：任意长度的句子都编码为固定维度的向量

引入注意力机制





ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder-decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder-decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

神经机器翻译的任务定义

传统神经机器翻译所用的编码器-解码器模型的缺陷

本文提出一种能够自动搜索原句中与预测目标词相关的神经机器翻译模型

所提出模型的效果

传统模型详解 改进前的RNNEc模型

任务定义

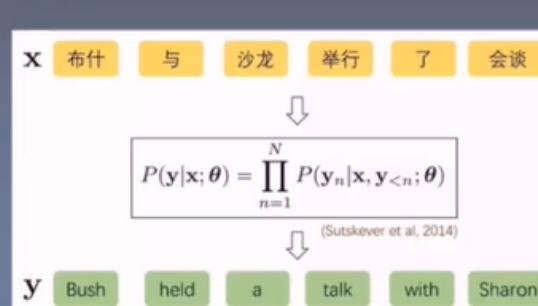
该模型采用1到K编码的字向量的源语言句子作为输入：

$$\mathbf{x} = (x_1, \dots, x_{T_x}), x_i \in \mathbb{R}^{K_x}$$

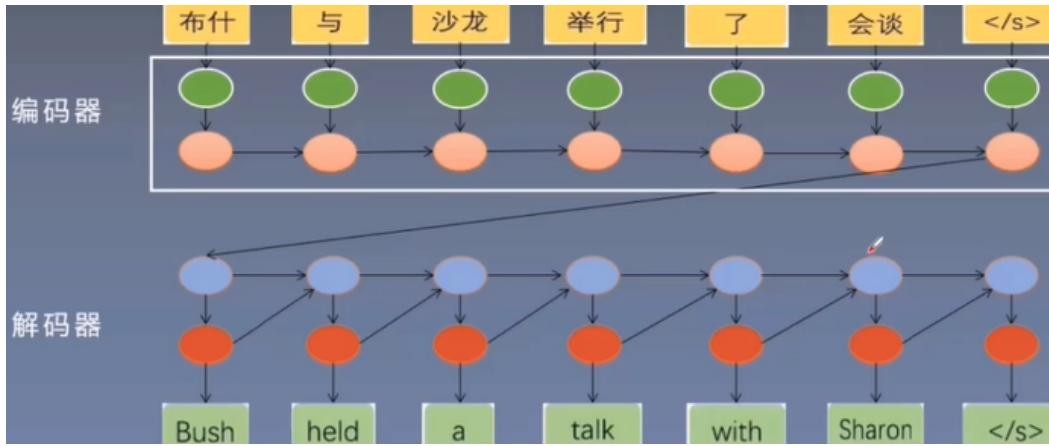
并输出由1到K编码的字向量的目标语言句子：

$$\mathbf{y} = (y_1, \dots, y_{T_y}), y_i \in \mathbb{R}^{K_y},$$

任务目标：评估函数 $\arg \max_y p(y|x)$



Encoder-Decoder baseline



Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho

Bart van Merriënboer Caglar Gulcehre

Université de Montréal

firstname.lastname@umontreal.ca

Dzmitry Bahdanau

Jacobs University, Germany

d.bahdanau@jacobs-university.de

Fethi Bougares Holger Schwenk

Université du Maine, France

Université de Montréal, CIFAR Senior Fellow

firstname.lastname@lum.univ-lemans.fr

Yoshua Bengio

find.me@on.the.web

论文来源：《Learning phrase representations using RNN encoder-decoder for statistical machine translation》

学习使用RNN编码器 - 解码器进行统计机器翻译的短语表示

模型名称：RNNenc

Encoder (baseline)

$$\mathbf{x} = (x_1, \dots, x_{T_x})$$

$$h_t = f(x_t, h_{t-1})$$

x 表示一个输入句子的序列

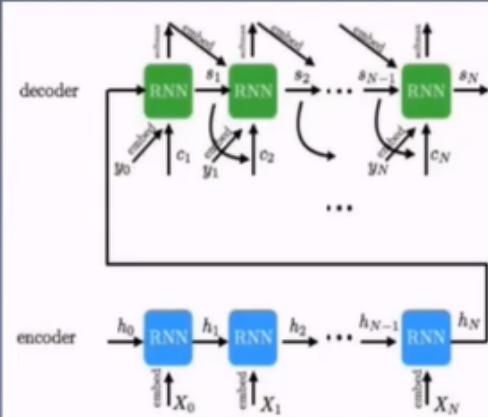
h_t 表示 t 时间生成句子的隐藏状态

f 表示非线性函数

$$c \not\equiv q(h_1, \dots, h_{T_x})$$

c 表示从句子序列中生成的上下文向量

q 表示非线性函数



$$\mathbf{y} = (y_1, \dots, y_{T_y})$$

\mathbf{h} 表示编码器的隐层状态

\mathbf{y} 表示生成一个句子的序列

\mathbf{s} 表示解码器的隐层状态

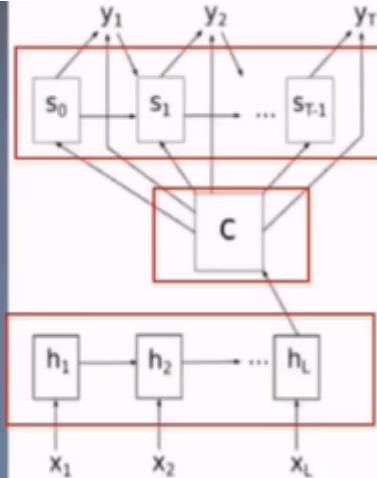
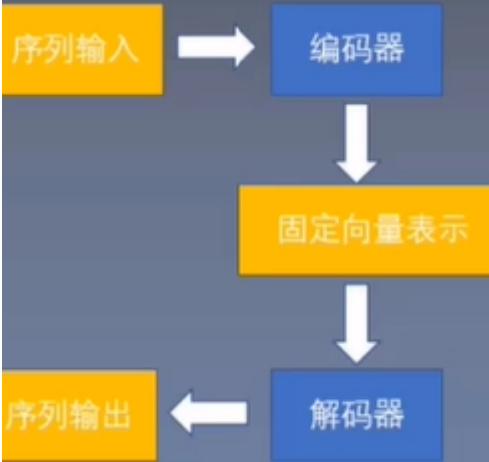
$$s_t = f(c, y_{t-1}, s_{t-1})$$

\mathbf{j} 表示编码器的输入

$$p(y_t | \{y_1, \dots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$$

\mathbf{i} 表示解码器的输入

s_t 表示循环神经网络的隐层状态



RNNenc模型效果

The history of machine translation

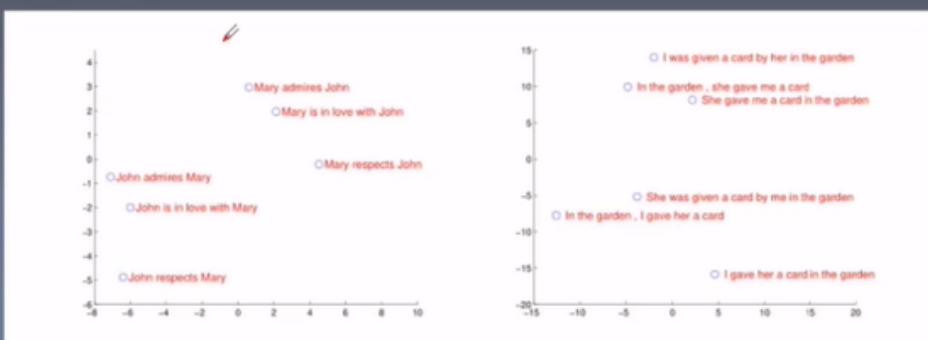
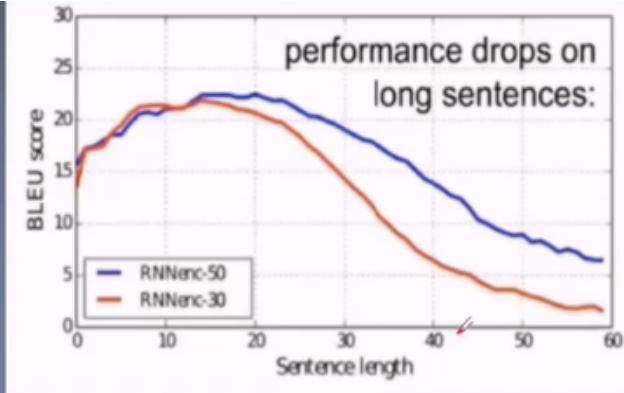


Figure 2: The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the figures. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

在机器翻译领域，使用Seq2Seq模型在英法翻译任务中表现接近技术最先进水平比传统的词袋模型效果好。

RNNec模型存在问题

必须记住整个句子序列的语义信息
把无论长度多长的句子都编码成固定向量，这样限制了翻译过程中长句子的表示
与人类翻译时的习惯不同，人们不会在生成目标语言翻译时关注源语言句子的每一个单词



论文贡献

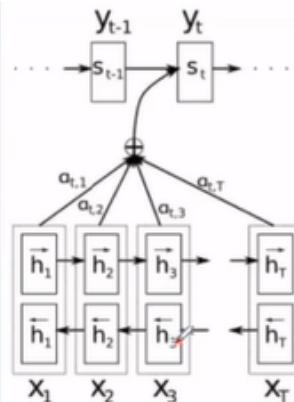
提出一种新的神经机器翻译模型 **RNNsearch** 模型

编码器：采用 **双向循环神经网络**

- 隐藏状态同时对当前单词前面和后面的信息编码

解码器：提出一种扩展（**注意力**）模型

- 注意力机制：对输入的隐藏状态求权重



RNNenc Vs RNNsearch

RNNenc

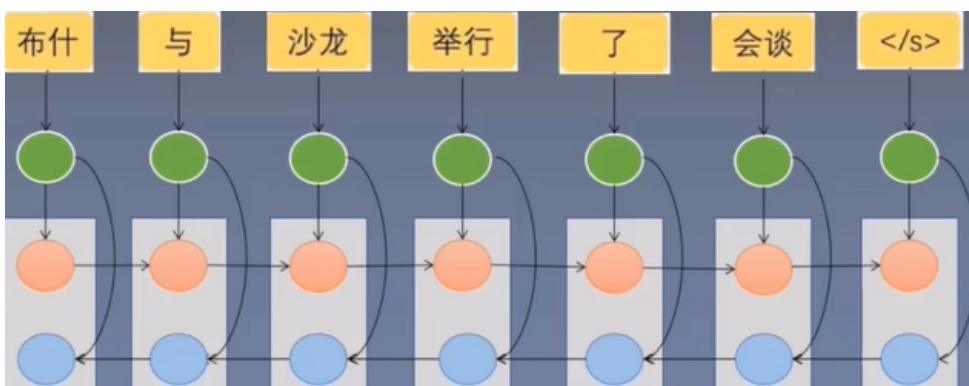
将整个输入语句编码成一个**固定长度**的向量

使用**单向循环神经网络** (RNN)

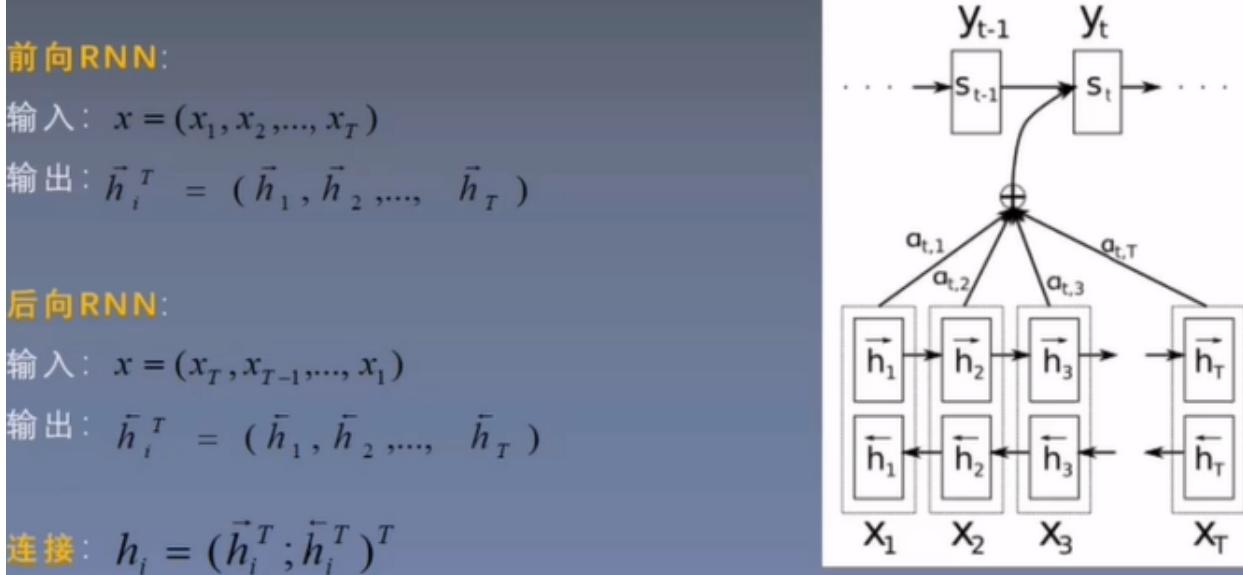
RNNsearch

- 将输入的句子编码为**变长向量序列**
- 在解码翻译时，**自适应的选择**这些向量的子集
- 使用**双向循环神经网络** (Bi-RNN)

双向循环神经网络



The encoder of RNNsearch



The encoder of RNNsearch

目标端词 y_i 的条件概率 :

$$p(y_i | y_1, \dots, y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i)$$

s_i 表示 i 时间的隐层状态 :

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

与 RNNenc 模型不同点 :

C Ci

The thought of attention 思想 : 集中关注的上下文

The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .
 The dog is chasing a cat on the ground .

(Cheng et al., 2016a)

计算上下文向量 c_i :

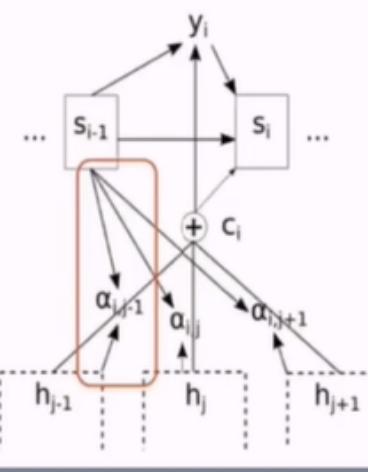
$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

权重 (注意力分数) α_{ij} :

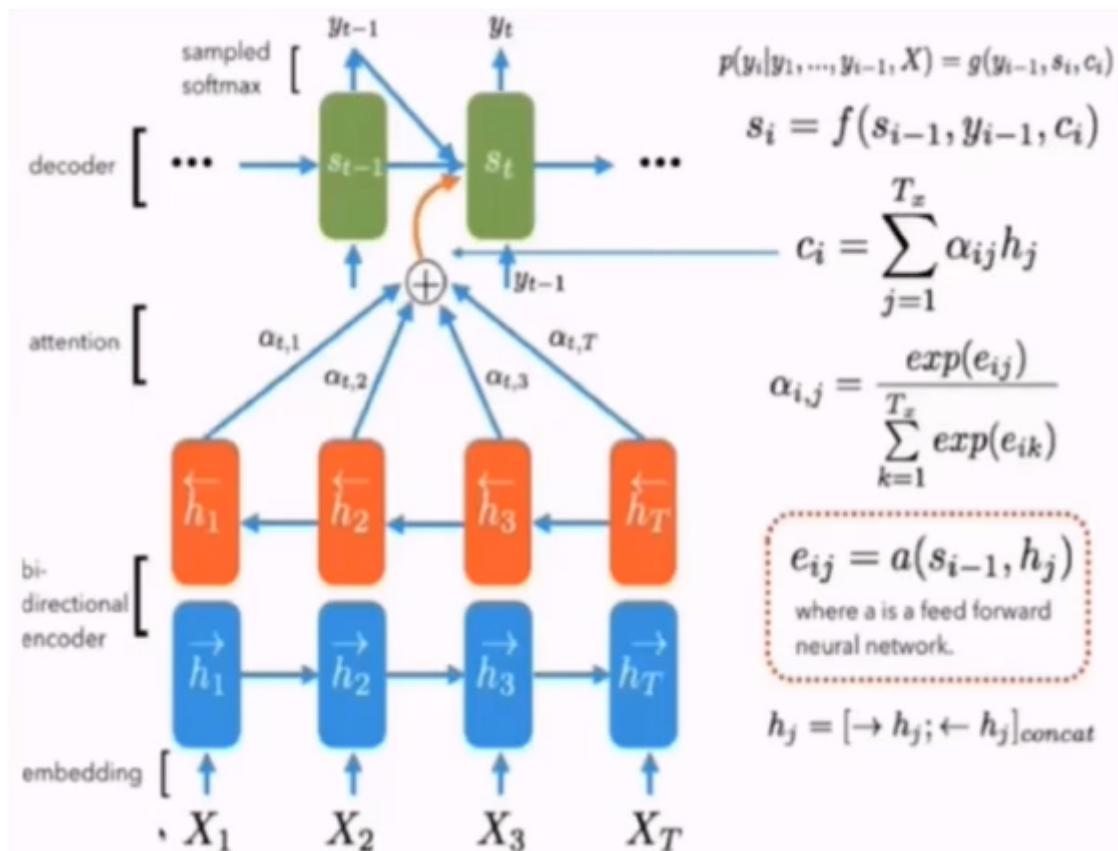
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

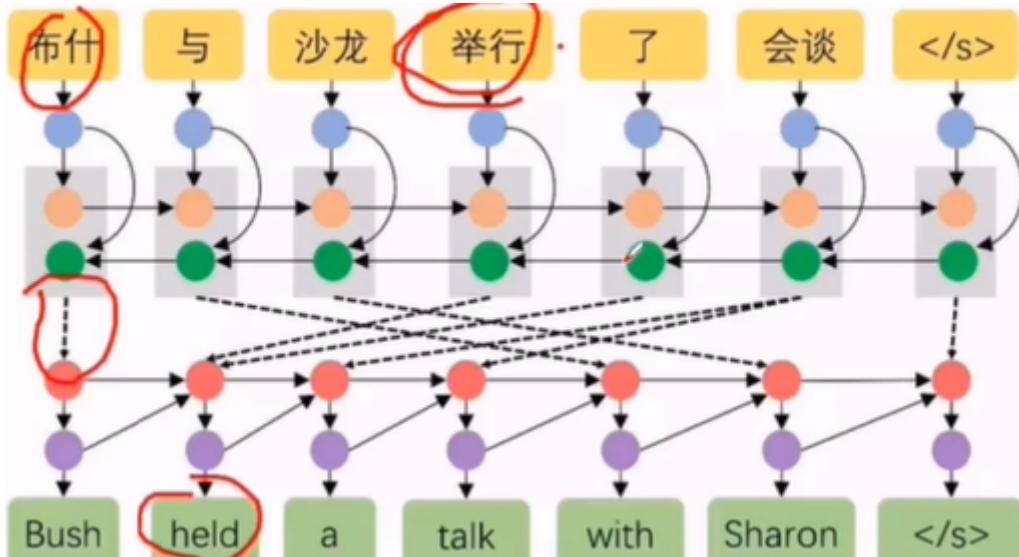
对齐模型:

$$e_{ij} = a(s_{i-1}, h_j)$$

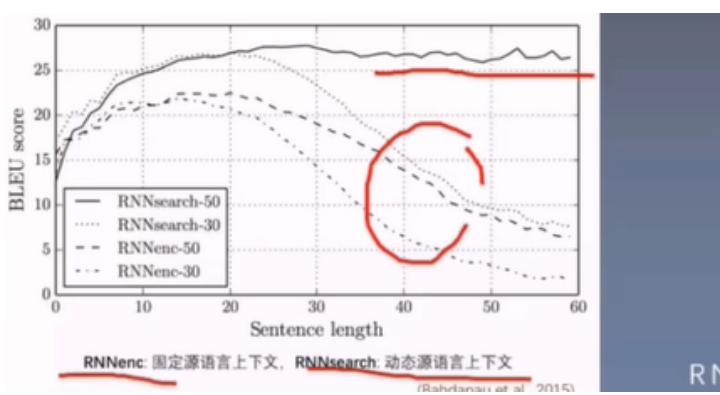
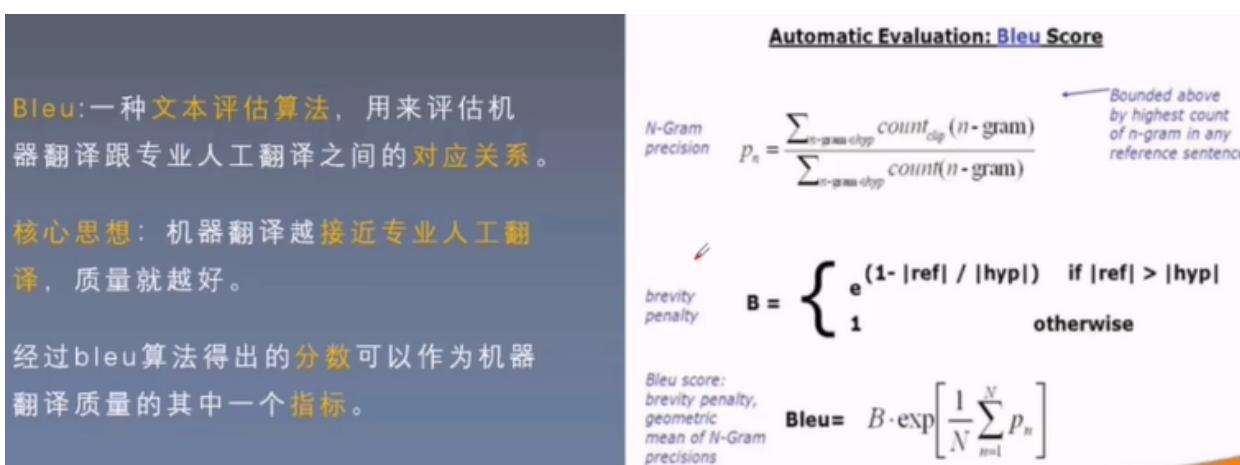


RNNsearch模型





实验模型	RNN search 和 RNNenc
实验任务	从英语（源语言）到法语（目标语言）的翻译
数据集	WMT'14数据集
对比试验	分别取最大长度为30和最大长度为50的句子长度进行实验



Model	All	No UNK°
RNNencdec-30	13.93	24.19
RNNsearch-30	21.50	31.44
RNNencdec-50	17.82	26.71
RNNsearch-50	26.75	34.16
RNNsearch-50*	28.45	36.15
Moses	33.30	35.63

RNNsearch模型在长句子上表现优异

问题	解决
注意力机制能够提升多少性能？	Luong等人证明使用不同注意力机制计算会导致不同的结果
双向循环神经网络能够提升多少性能？	Luong等人证明使用单向lstm和使用计算注意力分数具有同样的效果
有其他注意力分数计算方法吗？	Luong等人证明提出其他的注意力分数计算方法

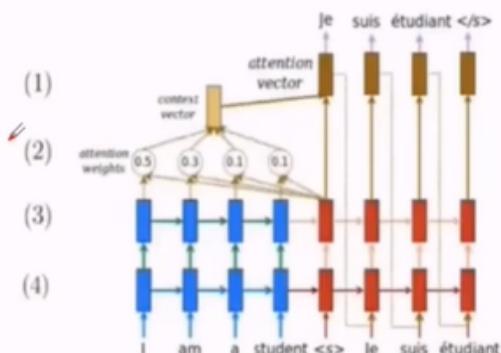
Effective Approaches to Attention-based Neural Machine Translation (Luong, 2015)

$$\alpha_{ts} = \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'=1}^S \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))} \quad [\text{Attention weights}]$$

$$\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s \quad [\text{Context vector}]$$

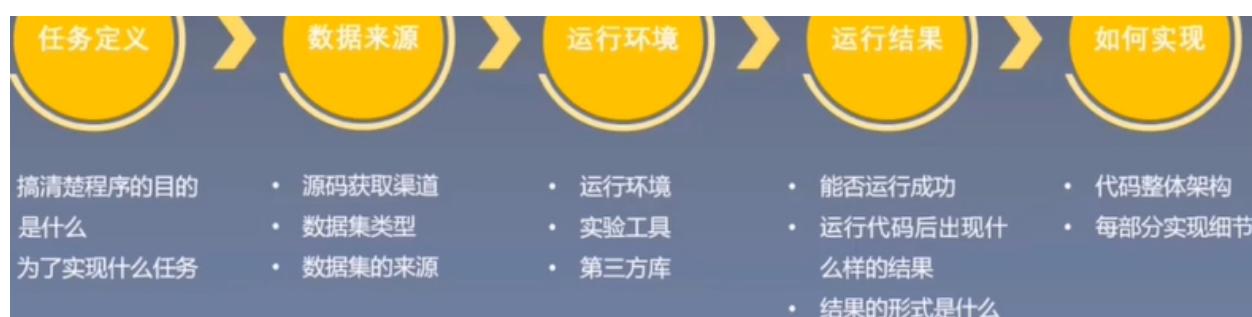
$$\mathbf{a}_t = f(\mathbf{c}_t, \mathbf{h}_t) = \tanh(W_c[\mathbf{c}_t; \mathbf{h}_t]) \quad [\text{Attention vector}]$$

$$\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^\top W \bar{\mathbf{h}}_s & \text{[Luong's multiplicative style]} \\ \mathbf{v}_a^\top \tanh(W_1 \mathbf{h}_t + W_2 \bar{\mathbf{h}}_s) & \text{[Bahdanau's additive style]} \end{cases}$$



论文重新点

- A 提出一种新的神经机器翻译方法
 - 1. 没有将输入编码到固定维度向量
 - 2. 采用注意力思想
- B 适用于其他结构化的输入输出问题
- C 一些设计的选择出于实际的考虑
 - 后续工作做了很多权衡分析



从上到下

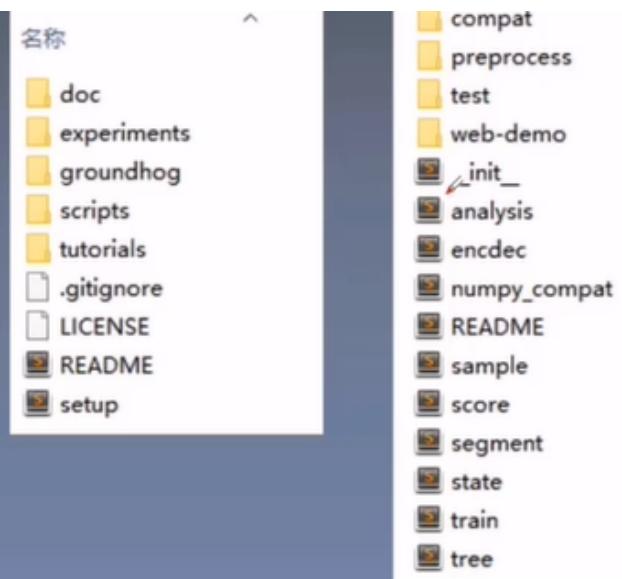
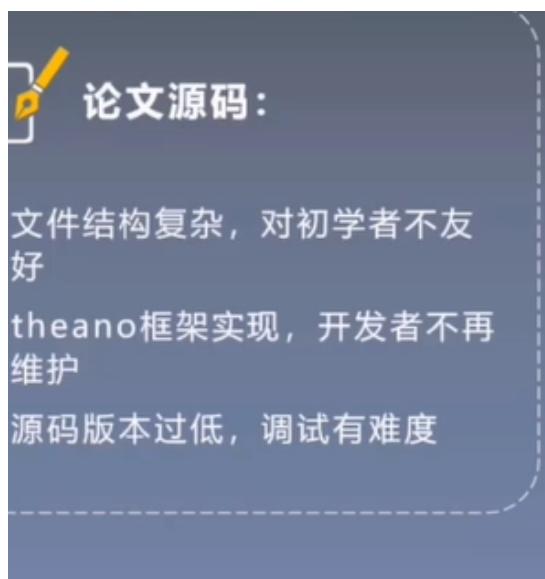
由点及面，先阅读各函数部分，弄懂各模块完成功能再阅读主程序部分

适用**结构简单的程序**

从下到上

先阅读主程序内容，再根据主程序中的调用函数阅读被调用部分，直到主程序执行完毕

适用**结构复杂的程序**



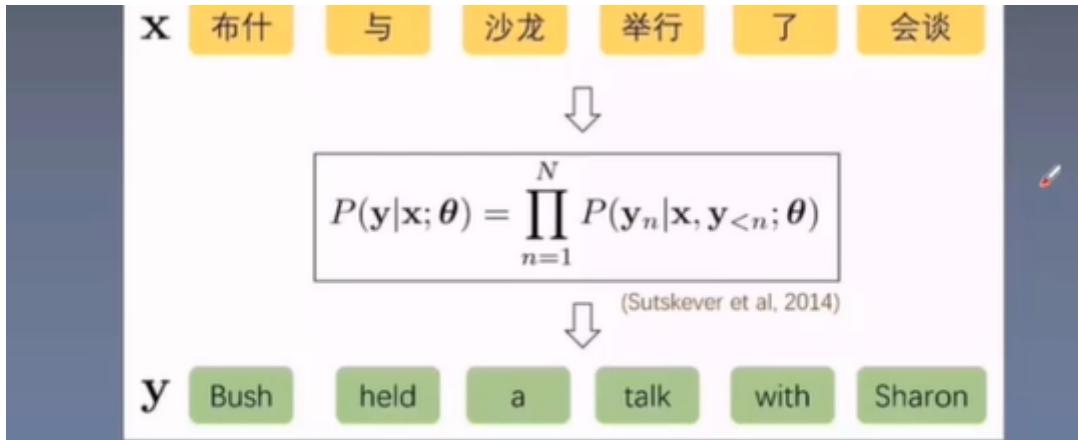
代码**结构简单**，只含有一个训练文件train.py和一个测试文件test.py

便于初学者理解代码**实现思想**

采用**TensorFlow框架**，使用人数多
受众广，便于调试



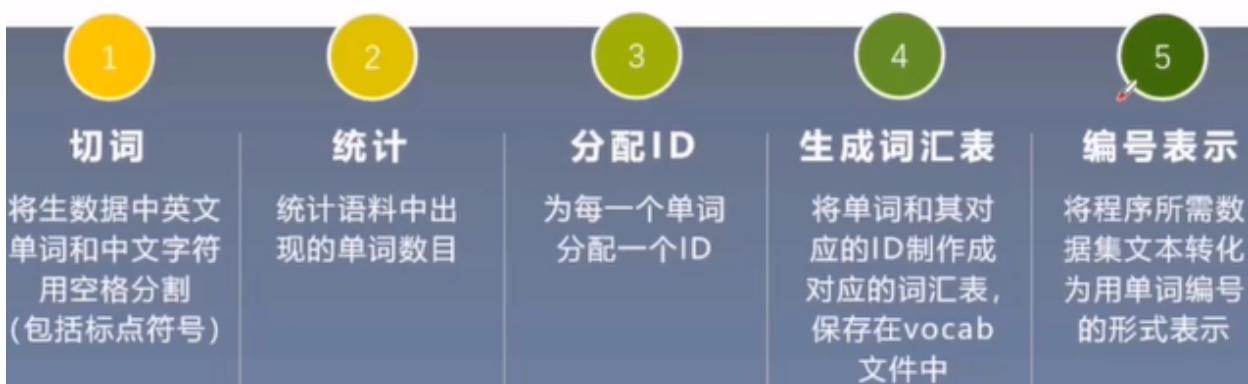
任务定义



训练一个可以由英文翻译成中文的神经网络翻译模型



语料的预处理过程



填充：用于填充长度而填充的位置

A ₁	A ₂	A ₃	A ₄
B ₁	B ₂	0	0

C ₁	C ₂	C ₃	C ₄	C ₅	C ₆
D ₁	0	0	0	0	0

```
tf.data.Dataset.padded_batch(  
    batch_size,  
    padded_shapes)  
  
tf.nn.dynamic_rnn(  
    cell, inputs,  
    sequence_length=None,  
    initial_state=None,  
    dtype=None,  
    parallel_iterations=None,  
    swap_memory=False,  
    time_major=False,  
    scope=None)
```

使用**Dataset**从一个文件中读取一个语言的数据。

数据的格式为每行一句话，单词已经转化为单词编号。

```
# 使用Dataset从一个文件中读取一个语言的数据。
def MakeDataset(file_path):
    dataset = tf.data.TextLineDataset(file_path)
    # 根据空格将单词编号切分开并放入一个一维向量。
    dataset = dataset.map(lambda string: tf.string_split([string]).values)
    # 将字符串形式的单词编号转化为整数。
    dataset = dataset.map(
        lambda string: tf.string_to_number(string, tf.int32))
    # 统计每个句子的单词数量，并与句子内容一起放入Dataset中。
    dataset = dataset.map(lambda x: (x, tf.size(x)))
return dataset
```

```
def MakeSrcTrgDataset(src_path, trg_path, batch_size):
    # 首先分别读取源语言数据和目标语言数据。
    src_data = MakeDataset(src_path)
    trg_data = MakeDataset(trg_path)
    # 通过zip操作将两个Dataset合并为一个Dataset。
    dataset = tf.data.Dataset.zip((src_data, trg_data))
    # 删除内容为空（只包含<EOS>）的句子和长度过长的句子。
    def FilterLength(src_tuple, trg_tuple):
        .....
        dataset = dataset.filter(FilterLength)

    def MakeTrgInput(src_tuple, trg_tuple):
        .....
        dataset = dataset.map(MakeTrgInput)
        # 随机打乱训练数据。
        dataset = dataset.shuffle(10000)
```

```
def MakeSrcTrgDataset(src_path, trg_path, batch_size):
    .....
    # 随机打乱训练数据。
    dataset = dataset.shuffle(10000)

    # 规定填充后输出的数据维度。
    padded_shapes = (
        (tf.TensorShape([None]), # 源句子是长度未知的向量
         tf.TensorShape([])), # 源句子长度是单个数字
        (tf.TensorShape([None]), # 目标句子（解码器输入）长度未知
         tf.TensorShape([None]), # 目标句子（解码器输出）长度未知
         tf.TensorShape([]))) # 目标句子长度是单个数字

    # 调用padded_batch方法进行batching操作。
    batched_dataset = dataset.padded_batch(batch_size,
                                            padded_shapes)

    return batched_dataset
```

```

SRC_TRAIN_DATA = "./train.en"          # 源语言输入文件。
TRG_TRAIN_DATA = "./train.zh"          # 目标语言输入文件。
CHECKPOINT_PATH = "./attention_ckpt"   # checkpoint保存路径。

HIDDEN_SIZE = 1024                    # LSTM的隐藏层规模。
DECODER_LAYERS = 2                   # 解码器中LSTM结构的层数。
SRC_VOCAB_SIZE = 10000                # 源语言词汇表大小。
TRG_VOCAB_SIZE = 4000                # 目标语言词汇表大小。
BATCH_SIZE = 100                     # 训练数据batch的大小。
NUM_EPOCH = 5                        # 使用训练数据的轮数。
KEEP_PROB = 0.8                      # 节点不被dropout的概率。
MAX_GRAD_NORM = 5                   # 用于控制梯度膨胀的梯度大小上限。
SHARE_EMB_AND_SOFTMAX = True        # 在Softmax层和词向量层之间共享参数。

MAX_LEN = 50    # 限定句子的最大单词数量。
SOS_ID = 1     # 目标语言词汇表中<sos>的ID。

```

训练步骤：

1. 初始化
2. 定义模型
3. 定义输入数据
4. 定义前向图
5. 训练模型

```

def main():
    # 1. 定义初始化函数。
    initializer = tf.random_uniform_initializer(-0.05, 0.05)
    # 2. 训练用的循环神经网络模型。
    with tf.variable_scope("nmt_model", reuse=None,
                           initializer=initializer):
        train_model = NMTModel()
    # 3. 定义输入数据。
    data = MakeSrcTrgDataset(SRC_TRAIN_DATA, TRG_TRAIN_DATA, BATCH_SIZE)
    iterator = data.make_initializable_iterator()
    (src, src_size), (trg_input, trg_label, trg_size) = iterator.get_next()

```

```

# 4. 定义前向计算图。输入数据以张量形式提供给forward函数。
cost_op, train_op = train_model.forward(src, src_size, trg_input,
                                         trg_label, trg_size)

# 5. 训练模型。
saver = tf.train.Saver()
step = 0
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(NUM_EPOCH):
        print("In iteration: %d" % (i + 1))
        sess.run(iterator.initializer)
        step = run_epoch(sess, cost_op, train_op, saver, step)

```

在模型的初始化函数中定义模型要用到的变量

```
class NMTModel(object):
    def __init__(self):
        # 定义编码器和解码器所使用的LSTM结构。
        self.enc_cell_fw = tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE) # 前向LSTM
        self.enc_cell_bw = tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE) # 后向LSTM
        self.dec_cell = tf.nn.rnn_cell.MultiRNNCell([
            tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE)
            for _ in range(DECODER_LAYERS)])

```

编码器定义：Bi-LSTM

解码器定义

接下来我们调用
tf.get_variable
来定义目标语言
和源语言词向量

```
class NMTModel(object):
    def __init__(self):
        .....
        # 为源语言和目标语言分别定义词向量
        self.src_embedding = tf.get_variable(
            "src_emb", [SRC_VOCAB_SIZE, HIDDEN_SIZE])
        self.trg_embedding = tf.get_variable(
            "trg_emb", [TRG_VOCAB_SIZE, HIDDEN_SIZE])

```

源语言

目标语言

定义**softmax**
层的变量

```
class NMTModel(object):
    def __init__(self):
        .....
        # 定义softmax层的变量
        if SHARE_EMB_AND_SOFTMAX:
            self.softmax_weight = tf.transpose(self.trg_embedding)
        else:
            self.softmax_weight = tf.get_variable(
                "weight", [HIDDEN_SIZE, TRG_VOCAB_SIZE])
        self.softmax_bias = tf.get_variable("softmax_bias",
                                           [TRG_VOCAB_SIZE])

```

定义偏置

定义权重

接下来我们调用
tf.get_variable
来定义目标语言
和源语言词向量

```
class NMTModel(object):
    .....
    # 在forward函数中定义模型的前向计算图。
    def forward(self, src_input, src_size, trg_input, trg_label, trg_size):
        batch_size = tf.shape(src_input)[0]

        # 将输入和输出单词编号转为词向量。
        src_emb = tf.nn.embedding_lookup(self.src_embedding, src_input)
        trg_emb = tf.nn.embedding_lookup(self.trg_embedding, trg_input)

        # 在词向量上进行dropout。
        src_emb = tf.nn.dropout(src_emb, KEEP_PROB)
        trg_emb = tf.nn.dropout(trg_emb, KEEP_PROB)

```

构造编码器,使用双

向循环神经网络

前向传播 构造编码器 Forward propagation

```
def forward(self, src_input, src_size, trg_input, trg_label, trg_size):  
    ...  
    with tf.variable_scope("encoder"):  
        # 构造编码器时, 使用bidirectional_dynamic_rnn构造双向循环网络。  
        enc_outputs, enc_state = tf.nn.bidirectional_dynamic_rnn(  
            self.enc_cell_fw,  
            self.enc_cell_bw,  
            src_emb, src_size,  
            dtype=tf.float32)  
        # 将两个LSTM的输出拼接为一个张量。  
        enc_outputs = tf.concat([enc_outputs[0], enc_outputs[1]], -1)
```

前向传播构造 解码器

选择注意力权重的计算模型

BahdanauAttention是使用一个隐藏层的前馈神经网络

- memory_sequence_length是一个维度为[batch_size]的张量, 代表batch中每个句子的长度, Attention需要根据这个信息把填充位置的注意力权重设置为0

```
tf.contrib.seq2seq.BahdanauAttention(  
    HIDDEN_SIZE,  
    enc_outputs,  
    memory_sequence_length  
    =src_size)
```

前向传播构造 解码器

tf.contrib.seq2seq.AttentionWrapper

将解码器的循环神经网络层和注意力层结合, 成为一个更高层的循环神经网络

```
tf.contrib.seq2seq.AttentionWrapper(  
    self.dec_cell,  
    attention_mechanism,  
    attention_layer_size  
    =HIDDEN_SIZE)
```

```
with tf.variable_scope("decoder"):  
    # 选择注意力权重的计算模型。  
    attention_mechanism = tf.contrib.seq2seq.BahdanauAttention(  
        HIDDEN_SIZE, enc_outputs,  
        memory_sequence_length=src_size)  
    # 将解码器的循环神经网络self.dec_cell和注意力一起封装成更高层的循环  
    # 神经网络。  
    attention_cell = tf.contrib.seq2seq.AttentionWrapper(  
        self.dec_cell,  
        attention_mechanism,  
        attention_layer_size=HIDDEN_SIZE)  
    # 使用attention_cell和dynamic_rnn构造编码器。  
    dec_outputs, _ = tf.nn.dynamic_rnn(  
        attention_cell,  
        trg_emb, trg_size, dtype=tf.float32)
```

计算损失函数

然后我们就可以像普通神经网络的一个隐藏层一样，添加偏置项（就是神经网络中的‘ b ’）和卷积输出相加并在最后添加一个激活层。

```
class NMTModel(object):
    ...
    def forward(self, src_input, src_size, trg_input, trg_label, trg_size):
        ...
        # 计算解码器每一步的log perplexity。
        output = tf.reshape(dec_outputs, [-1, HIDDEN_SIZE])
        logits = tf.matmul(output, self.softmax_weight) + self.softmax_bias
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
            labels=tf.reshape(trg_label, [-1]), logits=logits)
        ...
        ↴
```

计算平均损失函数，最后输出
cost_per_token

```
class NMTModel(object):
    ...
    def forward(self, src_input, src_size, trg_input, trg_label, trg_size):
        ...
        # 在计算平均损失时，需要将填充位置的权重设置为0，以避免无效位置的预测干扰模型的训练。
        label_weights = tf.sequence_mask(
            trg_size, maxlen=tf.shape(trg_label)[1], dtype=tf.float32)
        label_weights = tf.reshape(label_weights, [-1])
        cost = tf.reduce_sum(loss * label_weights)
        cost_per_token = cost / tf.reduce_sum(label_weights)
```

只在训练时定义
反向传播操作

```
class NMTModel(object):
    ...
    def forward(self, src_input, src_size, trg_input, trg_label, trg_size):
        ...
        # 定义反向传播操作。
        trainable_variables = tf.trainable_variables()
        # 控制梯度大小，定义优化方法和训练步骤。
        grads = tf.gradients(cost / tf.to_float(batch_size),
                             trainable_variables)
        grads, _ = tf.clip_by_global_norm(grads, MAX_GRAD_NORM)
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
        train_op = optimizer.apply_gradients(zip(grads, trainable_variables))
        return cost_per_token, train_op
```

```

def run_epoch(session, cost_op, train_op, saver, step):
    # 训练一个epoch。
    # 重复训练步骤直至遍历完Dataset中所有数据。
    while True:
        try:
            # 运行train_op并计算损失值。训练数据在main()函数中以Dataset方式提供
            cost, _ = session.run([cost_op, train_op])
            if step % 10 == 0:
                print("After %d steps,
                      per token cost is %.3f" % (step, cost))
            # 每200步保存一个checkpoint。
            if step % 200 == 0:
                saver.save(session, CHECKPOINT_PATH, global_step=step)
            step += 1
        except tf.errors.OutOfRangeError:
            break
    return step

```

虽然输入只有一个句子，但因为 **dynamic_rnn** 要求输入是batch的形式，因此这里将输入句子整理为大小为1的batch

```

class NMTModel(object):
    def __init__(self):
        .....
    def inference(self, src_input):
        src_size = tf.convert_to_tensor([len(src_input)], dtype=tf.int32)
        src_input = tf.convert_to_tensor([src_input], dtype=tf.int32)
        src_emb = tf.nn.embedding_lookup(self.src_embedding, src_input)
        with tf.variable_scope("encoder"):
            .....
        with tf.variable_scope("decoder"):

```

```

def inference(self, src_input):
    .....
    # 设置解码的最大步数。这是为了避免在极端情况出现无限循环的问题。
    MAX_DEC_LEN=100

    with tf.variable_scope("decoder/rnn/attention_wrapper"):
        # 使用一个变长的TensorArray来存储生成的句子。
        init_array = tf.TensorArray(dtype=tf.int32, size=0,
                                    dynamic_size=True, clear_after_read=False)
        # 填入第一个单词<sos>作为解码器的输入。
        init_array = init_array.write(0, SOS_ID)
        # 调用attention_cell.zero_state构建初始的循环状态。
        init_loop_var = (
            attention_cell.zero_state(batch_size=1, dtype=tf.float32),
            init_array, 0)
    # tf.while_loop的循环条件：循环直到解码器输出<eos>，或者达到最大步数为止
    def continue_loop_condition(state, trg_ids, step):
        return tf.reduce_all(tf.logical_and(
            tf.not_equal(trg_ids.read(step), EOS_ID),
            tf.less(step, MAX_DEC_LEN-1)))

```

```
def inference(self, src_input):
    ...
    def loop_body(state, trg_ids, step):
        # 读取最后一步输出的单词，并读取其词向量。
        trg_input = [trg_ids.read(step)]
        trg_emb = tf.nn.embedding_lookup(self.trg_embedding, trg_input)
        # 调用attention_cell向前计算一步。
        dec_outputs, next_state = attention_cell.call(state=state, inputs=trg_emb)
        # 计算每个可能的输出单词对应的logit，并选取logit值最大的单词作为这一步的而输出。
        output = tf.reshape(dec_outputs, [-1, HIDDEN_SIZE])
        logits = (tf.matmul(output, self.softmax_weight) + self.softmax_bias)
        next_id = tf.argmax(logits, axis=1, output_type=tf.int32)
        # 将这一步输出的单词写入循环状态的trg_ids中。
        trg_ids = trg_ids.write(step+1, next_id[0])
        return next_state, trg_ids, step+1
    # 执行tf.whileLoop，返回最终状态。
    state, trg_ids, step = tf.while_loop(continue_loop_condition,
                                         loop_body, init_loop_var)
    return trg_ids.stack()
```

测试主函数

The main of train

1. 定义模型
2. 定义测试句子
3. 转化源语言单词
id
4. 建立计算图
5. 转化目标语言单
词id
6. 转换文字
7. 输出

```
def main():
    # 定义训练用的循环神经网络模型。
    with tf.variable_scope("nmt_model", reuse=None):
        model = NMTModel()
    # 定义个测试句子。
    test_en_text = "This is a test . <eos>"
    # 根据英文词汇表，将测试句子转为单词ID。
    with codecs.open(SRC_VOCAB, "r", "utf-8") as f_vocab:
        src_vocab = [w.strip() for w in f_vocab.readlines()]
        src_id_dict = dict((src_vocab[x], x) for x in range(len(src_vocab)))
    test_en_ids = [(src_id_dict[token]
                    if token in src_id_dict else src_id_dict['<unk>'])
                    for token in test_en_text.split()]
```

```
def main():
    .....
    # 建立解码所需的计算图。
    output_op = model.inference(test_en_ids)
    sess = tf.Session()
    saver = tf.train.Saver()
    saver.restore(sess, CHECKPOINT_PATH)
    # 读取翻译结果。
    output_ids = sess.run(output_op)
    # 根据中文词汇表，将翻译结果转换为中文文字。
    with codecs.open(TRG_VOCAB, "r", "utf-8") as f_vocab:
        trg_vocab = [w.strip() for w in f_vocab.readlines()]
    output_text = ''.join([trg_vocab[x] for x in output_ids])
    # 输出翻译结果。
    print(output_text.encode('utf8').decode(sys.stdout.encoding))
    sess.close()
```

1、RNNsearch的源码如何才可以自己写出来，思路应该怎么整理？

01

数据预处理

02

模块函数

03

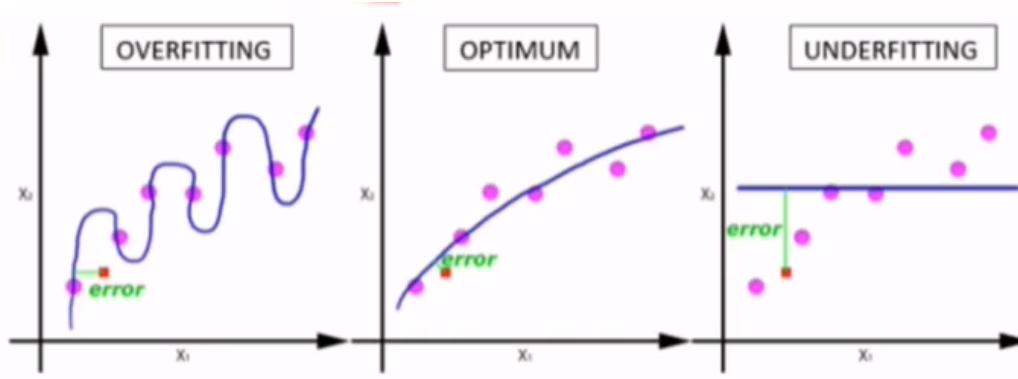
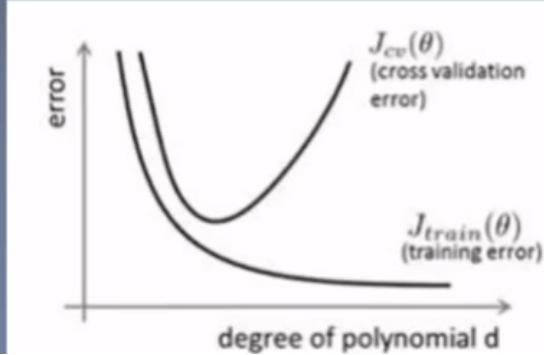
主函数

2、像这种任务的话，迭代多少代是根据什么来确定的？

epoch: 一个完整的数据集通过了神经网络一次并且返回了一次

batch size: 批大小

iteration: 使用batchsize个样本训练一次



过拟合：模型拟合能力过强

欠拟合：模型拟合能力过弱

3、我发现这次的实现代码基本是用tensorflow写完的，即使很多用python和numpy可以轻松做的事情，也用tensorflow来处理。个人不太明白这样做的意义，比如一个while循环，虽然我也看得懂tensorflow写的版本，但是是否直接写普通的while会更方便呢？工作中，是不是什么功能都要用tensorflow来实现？

使用深度学习框架目的：

一方面，它的迭代速度是比较高效的；

另一方面，一个通用的框架也比较方便进行合作和拓展，方便程序员们进行交流。

4、最新模型state of art啥样 效果如何
不同风格咋办

Understanding Back-Translation at Scale

Sergey Edunov[△] Myle Ott[△] Michael Auli[△] David Grangier

[△]Facebook AI Research, Menlo Park, CA & New York, NY.

[▽]Google Brain, Mountain View, CA.



bleu分数：35