机器翻译

本教程源代码目录在book/machine_translation,初次使用请您参考Book文档使用说明。

说明

1. 硬件要求 本文可支持在CPU、GPU下运行
2. 对docker file cuda/cudnn的支持 如果您使用了本文配套的docker镜像，请注意：该镜像对GPU的支持仅限于CUDA 8，cuDNN 5
3. 文档中代码和train.py不一致的问题 请注意：为使本文更加易读易用，我们拆分、调整了train.py的代码并放入本文。本文中代码与train.py的运行结果一致，如希望直接看到训练脚本输出效果，可运行train.py。

背景介绍

机器翻译（machine translation, MT）是用计算机来实现不同语言之间翻译的技术。被翻译的语言通常称为源语言（source language），翻译成的结果语言称为目标语言（target language）。机器翻译即实现从源语言到目标语言转换的过程，是自然语言处理的重要研究领域之一。

早期机器翻译系统多为基于规则的翻译系统，需要由语言学家编写两种语言之间的转换规则，再将这些规则录入计算机。该方法对语言学家的要求非常高，而且我们几乎无法总结一门语言会用到的所有规则，更何况两种甚至更多的语言。因此，传统机器翻译方法面临的主要挑战是无法得到一个完备的规则集合[1]。

为解决以上问题，统计机器翻译（Statistical Machine Translation, SMT）技术应运而生。在统计机器翻译技术中，转化规则是由机器自动从大规模的语料中学习得到的，而非我们人主动提供规则。因此，它克服了基于规则的翻译系统所面临的知识获取瓶颈的问题，但仍然存在许多挑战：1）人为设计许多特征（feature），但永远无法覆盖所有的语言现象；2）难以利用全局的特征；3）依赖于许多预处理环节，如词语对齐、分词或符号化（tokenization）、规则抽取、句法分析等，而每个环节的错误会逐步累积，对翻译的影响也越来越大。

近年来，深度学习技术的发展为解决上述挑战提供了新的思路。将深度学习应用于机器翻译任务的方法大致分为两类：1）仍以统计机器翻译系统为框架，只是利用神经网络来改进其中的关键模块，如语言模型、调序模型等（见图1的左半部分）；2）不再以统计机器翻译系统为框架，而是直接用神经网络将源语言映射到目标语言，即端到端的神经网络机器翻译（End-to-End Neural Machine Translation, End-to-End NMT）（见图1的右半部分），简称为NMT模型。

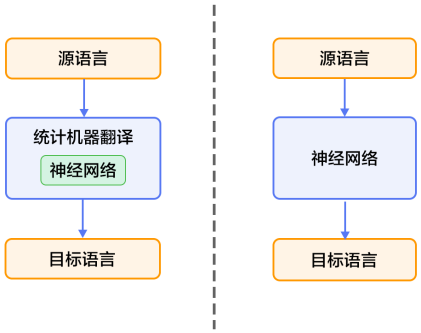

图1. 基于神经网络的机器翻译系统

本教程主要介绍NMT模型，以及如何用PaddlePaddle来训练一个NMT模型。

效果展示

以中英翻译（中文翻译到英文）的模型为例，当模型训练完毕时，如果输入如下已分词的中文句子：

```
这些 是 希望 的 曙光 和 解脱 的 迹象 .
```

```
0 -5.36816   These are signs of hope and relief . <e>
1 -6.23177   These are the light of hope and relief . <e>
2 -7.7914  These are the light of hope and the relief of hope . <e>
```

- 左起第一列是生成句子的序号；左起第二列是该条句子的得分（从大到小），分值越高越好；左起第三列是生成的英语句子。
- 另外有两个特殊标志：`<e>` 表示句子的结尾，`<unk>` 表示未登录词（unknown word），即未在训练字典中出现的词。

模型概览

本节依次介绍GRU（Gated Recurrent Unit，门控循环单元），双向循环神经网络（Bi-directional Recurrent Neural Network），NMT模型中典型的编码器-解码器（Encoder-Decoder）框架和注意力（Attention）机制，以及柱搜索（beam search）算法。

GRU

我们已经在情感分析一章中介绍了循环神经网络（RNN）及长短时间记忆网络（LSTM）。相比于简单的RNN，LSTM增加了记忆单元（memory cell）、输入门（input gate）、遗忘门（forget gate）及输出门（output gate），这些门及记忆单元组合起来大大提升了RNN处理远距离依赖问题的能力。

GRU[2]是Cho等人在LSTM上提出的简化版本，也是RNN的一种扩展，如下图所示。GRU单元只有两个门：

- 重置门（reset gate）：如果重置门关闭，会忽略掉历史信息，即历史不相干的信息不会影响未来的输出。
- 更新门（update gate）：将LSTM的输入门和遗忘门合并，用于控制历史信息对当前时刻隐层输出的影响。如果更新门接近1，会把历史信息传递下去。

图2. GRU（门控循环单元）

一般来说，具有短距离依赖属性的序列，其重置门比较活跃；相反，具有长距离依赖属性的序列，其更新门比较活跃。另外，Chung等人[3]通过多组实验表明，GRU虽然参数更少，但是在多个任务上都和LSTM有相近的表现。

双向循环神经网络

我们已经在语义角色标注一章中介绍了一种双向循环神经网络，这里介绍Bengio团队在论文[2,4]中提出的另一种结构。该结构的目的是输入一个序列，得到其在每个时刻的特征表示，即输出的每个时刻都用定长向量表示到该时刻的上下文语义信息。

具体来说，该双向循环神经网络分别在时间维以顺序和逆序——即前向（forward）和后向（backward）——依次处理输入序列，并将每个时间步RNN的输出拼接成为最终的输出层。这样每个时间步的输出节点，都包含了输入序列中当前时刻完整的过去和未来的上下文信息。下图展示的是一个按时间步展开的双向循环神经网络。该网络包含一个前向和一个后向RNN，其中有六个权重矩阵：输入到前向隐层和后向隐层的权重矩阵（W_1, W_3），隐层到隐层自己的权重矩阵（W_2, W_5），前向隐层和后向隐层到输出层的权重矩阵（W_4, W_6）。注意，该网络的前向隐层和后向隐层之间没有连接。

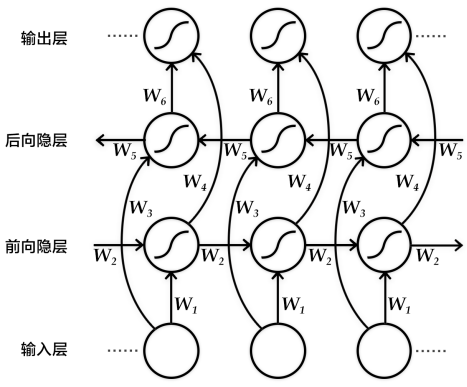

图2. 按时间步展开的双向循环神经网络

编码器-解码器框架

编码器-解码器（Encoder-Decoder）[2]框架用于解决由一个任意长度的源序列到另一个任意长度的目标序列的变换问题。即编码阶段将整个源序列编码成一个向量，解码阶段通过最大化预测序列概率，从中解码出整

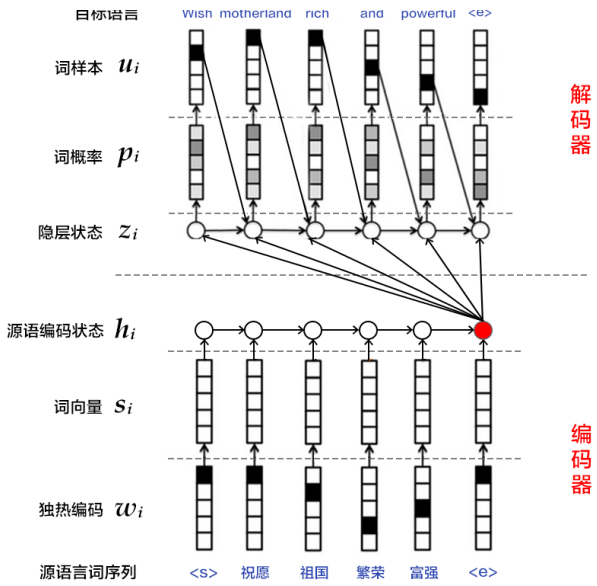

图3. 编码器-解码器框架

编码器

编码阶段分为三步：

1. one-hot vector表示：将源语言句子x=\left { x_1,x_2,...,x_T \right }的每个词x_i表示成一个列向量w_i\epsilon \left { 0,1 \right }^{\left | V \right |},i=1,2,...,T。这个向量w_i的维度与词汇表大小$|V|$相同，并且只有一个维度上有值1（该位置对应该词在词汇表中的位置），其余全是0。
2. 映射到低维语义空间的词向量：one-hot vector表示存在两个问题，1）生成的向量维度往往很大，容易造成维数灾难；2）难以刻画词与词之间的关系（如语义相似性，也就是无法很好地表达语义）。因此，需再one-hot vector映射到低维的语义空间，由一个固定维度的稠密向量（称为词向量）表示。记映射矩阵为$C\epsilon R^{K\times|V|}$，用$s_i = Cw_i$表示第i个词的词向量，K为向量维度。
3. 用RNN编码源语言词序列：这一过程的计算公式为$h_i=\varnothing \theta \left (h_{i-1}, s_i \right)$，其中$h_0$是一个全零的向量，$\varnothing$是一个非线性激活函数，最后得到的$\mathbf{h}=\left { h_1,..., h_T \right }$就是$RNN$依次读入源语言$T$个词的状态编码序列。整句话的向量表示可以采用$\mathbf{h}$在最后一个时间步$T$的状态编码，或使用时间维上的池化（pooling）结果。

第3步也可以使用双向循环神经网络实现更复杂的句编码表示，具体可以用双向GRU实现。前向GRU按照词序列$(x_1, x_2, \ldots, x_T)$的顺序依次编码源语言端词，并得到一系列隐层状态$(\overrightarrow{h_1}, \overrightarrow{h_2}, \ldots, \overrightarrow{h_T})$。类似的，后向GRU按照$(x_T, x_{T-1}, \ldots, x_1)$的顺序依次编码源语言端词，得到$(\overleftarrow{h_1}, \overleftarrow{h_2}, \ldots, \overleftarrow{h_T})$。最后对于词$x_i$，通过拼接两个GRU的结果得到它的隐层状态，即$h_i = \left[\overrightarrow{h_i^T}, \overleftarrow{h_i^T}\right]^T$。

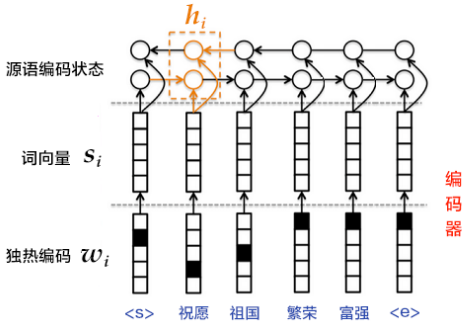

图4. 使用双向GRU的编码器

解码器

机器翻译任务的训练过程中，解码阶段的目标是最大化下一个正确的目标语言词的概率。思路是：

1. 每一个时刻，根据源语言句子的编码信息（又叫上下文向量，context vector）c、真实目标语言序列的第i个词u_i和i时刻RNN的隐层状态z_i，计算出下一个隐层状态z_{i+1}。计算公式如下：

$$z_{i+1} = \phi_{\theta'}(c, u_i, z_i)$$

其中ϕ_θ是一个非线性激活函数；c是源语言句子的上下文向量，在不使用注意力机制时，如果编码器的输出是源语言句子编码后的最后一个元素，则可以定义$c = h_T$；u_i是目标语言序列的第i个单词，u_0是目标语言序列的开始标记 <s> ，表示解码开始；z_i是i时刻解码RNN的隐层状态，z_0是一个全零的向量。

其中$W_s z_{i+1} + b_z$是对每个可能的输出单词进行打分，再用softmax归一化就可以得到第$i+1$个词的概率p_{i+1}。

1. 根据p_{i+1}和u_{i+1}计算代价。
2. 重复步骤1~3，直到目标语言序列中的所有词处理完毕。

机器翻译任务的生成过程，通俗来讲就是根据预先训练的模型来翻译源语言句子。生成过程中的解码阶段和上述训练过程的有所差异，具体介绍请见柱搜索算法。

注意力机制

如果编码阶段的输出是一个固定维度的向量，会带来以下两个问题：1）不论源语言序列的长度是5个词还是50个词，如果都用固定维度的向量去编码其中的语义和句法结构信息，对模型来说是一个非常高的要求，特别是对长句子序列而言；2）直觉上，当人类翻译一句话时，会对与当前译文更相关的源语言片段上给予更多关注，且关注点会随着翻译的进行而改变。而固定维度的向量则相当于，任何时刻都对源语言所有信息给予了同等程度的关注，这是不合理的。因此，Bahdanau等人[4]引入注意力（attention）机制，可以对编码后的上下文片段进行解码，以此来解决长句子的特征学习问题。下面介绍在注意力机制下的解码器结构。

与简单的解码器不同，这里z_i的计算公式为：

$$z_{i+1} = \phi_{\theta'}(c_i, u_i, z_i)$$

可见，源语言句子的编码向量表示为第i个词的上下文片段c_i，即针对每一个目标语言中的词u_i，都有一个特定的c_i与之对应。c_i的计算公式如下：

$$c_i = \sum_{j=1}^{T} a_{ij} h_j, a_i = [a_{i1}, a_{i2}, \ldots, a_{iT}]$$

从公式中可以看出，注意力机制是通过对编码器中各时刻的RNN状态h_j进行加权平均实现的。权重a_{ij}表示目标语言中第i个词对源语言中第j个词的注意力大小，a_{ij}的计算公式如下：

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T} exp(e_{ik})}$$

$$e_{ij} = align(z_i, h_j)$$

其中，$align$可以看作是一个对齐模型，用来衡量目标语言中第i个词和源语言中第j个词的匹配程度。具体而言，这个程度是通过解码RNN的第i个隐层状态z_i和源语言句子的第j个上下文片段h_j计算得到的。传统的对齐模型中，目标语言的每个词明确对应源语言的一个或多个词（hard alignment）；而在注意力模型中采用的是soft alignment，即任何两个目标语言和源语言词间均存在一定的关联，且这个关联强度是由模型计算得到的实数，因此可以融入整个NMT框架，并通过反向传播算法进行训练。

图6. 基于注意力机制的解码器

柱搜索算法

柱搜索（beam search）是一种启发式图搜索算法，用于在图或树中搜索有限集合中的最优扩展节点，通常用在解空间非常大的系统（如机器翻译、语音识别）中，原因是内存无法装下图或树中所有展开的解。如在机器翻译任务中希望翻译"<s>你好<e>"，就算目标语言字典中只有3个词（<s>，<e>，hello），也可能生成无限句话（hello循环出现的次数不定），为了找到其中较好的翻译结果，我们可采用柱搜索算法。

柱搜索算法使用广度优先策略建立搜索树，在树的每一层，按照启发代价（heuristic cost）（本教程中，为生成词的log概率之和）对节点进行排序，然后仅留下预先确定的个数（文献中通常称为beam width、beam size、柱宽度等）的节点。只有这些节点会在下一层继续扩展，其他节点就被剪掉了，也就是说保留了质量较高的节点，剪枝了质量较差的节点。因此，搜索所占用的空间和时间大幅减少，但缺点是无法保证一定获得最优解。

使用柱搜索算法的解码阶段，目标是最大化生成序列的概率。思路是：

1. 每一个时刻，根据源语言句子的编码信息c、生成的第i个目标语言序列单词u_i和i时刻RNN的隐层状态z_i，计算出下一个隐层状态z_{i+1}。
2. 将z_{i+1}通过softmax归一化，得到目标语言序列的第$i+1$个单词的概率分布p_{i+1}。
3. 根据p_{i+1}采样出单词u_{i+1}。
4. 重复步骤1~3，直到获得句子结束标记<e>或超过句子的最大生成长度为止。

注意：z_{i+1}和p_{i+1}的计算公式同解码器中的一样。且由于生成时的每一步都是通过贪心法实现的，因此并不能保证得到全局最优解。

本教程使用WMT-16新增的multimodal task中的translation task的数据集。该数据集为英德翻译数据，包含29001条训练数据，1000条测试数据。

数据预处理

我们的预处理流程包括两步：

- *将每个源语言到目标语言的平行语料库文件合并为一个文件：*
- *合并每个 `XXX.src` 和 `XXX.trg` 文件为 `XXX`。*
- *`XXX` 中的第i行内容为 `XXX.src` 中的第i行和 `XXX.trg` 中的第i行连接，用'\t'分隔。*
- *创建训练数据的"源字典"和"目标字典"。每个字典都有**DICTSIZE**个单词，包括：语料中词频最高的（DICTSIZE - 3）个单词，和3个特殊符号 `<s>`（序列的开始）、`<e>`（序列的结束）和 `<unk>`（未登录词）。*

示例数据

为了验证训练流程，PaddlePaddle接口 `paddle.dataset.wmt16` 中提供了对该数据集预处理后的版本，调用该接口即可直接使用，因为数据规模限制，这里只作为示例使用，在相应的测试集上具有一定效果但在更多测试数据上的效果无法保证。

模型配置说明

下面我们开始根据输入数据的形式配置模型。首先引入所需的库函数以及定义全局变量：

```
from __future__ import print_function
import os
import six

import paddle
import paddle.fluid as fluid

dict_size = 30000  # 词典大小
source_dict_size = target_dict_size = dict_size  # 源/目标语言字典大小
word_dim = 512  # 词向量维度
hidden_dim = 512  # 编码器中的隐层大小
decoder_size = hidden_dim  # 解码器中的隐层大小
max_length = 256  # 解码生成句子的最大长度
beam_size = 4  # beam search的柱宽度
batch_size = 64  # batch 中的样本数

is_sparse = True
model_save_dir = "machine_translation.inference.model"
```

然后如下实现编码器框架：

```
def encoder():
    # 定义源语言id序列的输入数据
    src_word_id = fluid.layers.data(
        name="src_word_id", shape=[1], dtype='int64', lod_level=1)
    # 将上述编码映射到低维语言空间的词向量
    src_embedding = fluid.layers.embedding(
        input=src_word_id,
        size=[source_dict_size, word_dim],
        dtype='float32',
        is_sparse=is_sparse)
    # 用双向GRU编码源语言序列，拼接两个GRU的编码结果得到h
    fc_forward = fluid.layers.fc(
        input=src_embedding, size=hidden_dim * 3, bias_attr=False)
    src_forward = fluid.layers.dynamic_gru(input=fc_forward, size=hidden_dim)
    fc_backward = fluid.layers.fc(
        input=src_embedding, size=hidden_dim * 3, bias_attr=False)
    src_backward = fluid.layers.dynamic_gru(
        input=fc_backward, size=hidden_dim, is_reverse=True)
    encoded_vector = fluid.layers.concat(
        input=[src_forward, src_backward], axis=1)
    return encoded_vector
```

再实现基于注意力机制的解码器：

- *首先定义解码器中单步的计算，即$z_{i+1} = \phi_\theta\left(c_i, u_i, z_i\right)$，如下：*

```
    # 定义RNN中的单步计算
    def cell(x, hidden, encoder_out, encoder_out_proj):
```

```
            input=decoder_state, size=decoder_size, bias_attr=False)
        # sequence_expand将单步内容扩展为与encoder输出相同的序列
        decoder_state_expand = fluid.layers.sequence_expand(
            x=decoder_state_proj, y=encoder_proj)
        mixed_state = fluid.layers.elementwise_add(encoder_proj,
                                                   decoder_state_expand)
        attention_weights = fluid.layers.fc(
            input=mixed_state, size=1, bias_attr=False)
        attention_weights = fluid.layers.sequence_softmax(
            input=attention_weights)
        weigths_reshape = fluid.layers.reshape(x=attention_weights, shape=[-1])
        scaled = fluid.layers.elementwise_mul(
            x=encoder_vec, y=weigths_reshape, axis=0)
        context = fluid.layers.sequence_pool(input=scaled, pool_type='sum')
        return context

    context = simple_attention(encoder_out, encoder_out_proj, hidden)
    out = fluid.layers.fc(
        input=[x, context], size=decoder_size * 3, bias_attr=False)
    out = fluid.layers.gru_unit(
        input=out, hidden=hidden, size=decoder_size * 3)[0]
    return out, out
```

- *基于定义的单步计算，使用 DynamicRNN 实现多步循环的训练模式下解码器，如下：*

```
def train_decoder(encoder_out):
    # 获取编码器输出的最后一步并进行非线性映射以构造解码器RNN的初始状态
    encoder_last = fluid.layers.sequence_last_step(input=encoder_out)
    encoder_last_proj = fluid.layers.fc(
        input=encoder_last, size=decoder_size, act='tanh')
    # 编码器输出在attention中计算结果的cache
    encoder_out_proj = fluid.layers.fc(
        input=encoder_out, size=decoder_size, bias_attr=False)

    # 定义目标语言id序列的输入数据，并映射到低维语言空间的词向量
    trg_language_word = fluid.layers.data(
        name="target_language_word", shape=[1], dtype='int64', lod_level=1)
    trg_embedding = fluid.layers.embedding(
        input=trg_language_word,
        size=[target_dict_size, word_dim],
        dtype='float32',
        is_sparse=is_sparse)

    rnn = fluid.layers.DynamicRNN()
    with rnn.block():
        # 获取当前步目标语言输入的词向量
        x = rnn.step_input(trg_embedding)
        # 获取隐层状态
        pre_state = rnn.memory(init=encoder_last_proj, need_reorder=True)
        # 在DynamicRNN中需使用static_input获取encoder相关的内容
        # 对decoder来说这些内容在每个时间步都是固定的
        encoder_out = rnn.static_input(encoder_out)
        encoder_out_proj = rnn.static_input(encoder_out_proj)
        # 执行单步的计算单元
        out, current_state = cell(x, pre_state, encoder_out, encoder_out_proj)
        # 计算归一化的单词预测概率
        prob = fluid.layers.fc(input=out, size=target_dict_size, act='softmax')
        # 更新隐层状态
        rnn.update_memory(pre_state, current_state)
        # 输出预测概率
        rnn.output(prob)

    return rnn()
```

接着就可以使用编码器和解码器定义整个训练网络；为了进行训练还需要定义优化器，如下：

```
def train_model():
    encoder_out = encoder()
    rnn_out = train_decoder(encoder_out)
    label = fluid.layers.data(
        name="target_language_next_word", shape=[1], dtype='int64', lod_level=1)
    # 定义损失函数
    cost = fluid.layers.cross_entropy(input=rnn_out, label=label)
    avg_cost = fluid.layers.mean(cost)
    return avg_cost

def optimizer_func():
    # 设置梯度裁剪
    fluid.clip.set_gradient_clip(
        clip=fluid.clip.GradientClipByGlobalNorm(clip_norm=5.0))
    # 定义先增后降的学习率策略
    lr_decay = fluid.layers.learning_rate_scheduler.noam_decay(hidden_dim, 1000)
    return fluid.optimizer.Adam(
```

以上是训练所需的模型构件，预测（生成）模式下基于beam search的解码器需要借助 `while_op` 实现，如下：

```
def infer_decoder(encoder_out):
    # 获取编码器输出的最后一步并进行非线性映射以构造解码器RNN的初始状态
    encoder_last = fluid.layers.sequence_last_step(input=encoder_out)
    encoder_last_proj = fluid.layers.fc(
        input=encoder_last, size=decoder_size, act='tanh')
    # 编码器输出在attention中计算结果的cache
    encoder_out_proj = fluid.layers.fc(
        input=encoder_out, size=decoder_size, bias_attr=False)

    # 最大解码步数
    max_len = fluid.layers.fill_constant(
        shape=[1], dtype='int64', value=max_length)
    # 解码步数计数变量
    counter = fluid.layers.zeros(shape=[1], dtype='int64', force_cpu=True)

    # 定义 tensor array 用以保存各个时间步的内容，并写入初始id，score和state
    init_ids = fluid.layers.data(
        name="init_ids", shape=[1], dtype="int64", lod_level=2)
    init_scores = fluid.layers.data(
        name="init_scores", shape=[1], dtype="float32", lod_level=2)
    ids_array = fluid.layers.array_write(init_ids, i=counter)
    scores_array = fluid.layers.array_write(init_scores, i=counter)
    state_array = fluid.layers.array_write(encoder_last_proj, i=counter)

    # 定义循环终止条件变量
    cond = fluid.layers.less_than(x=counter, y=max_len)
    while_op = fluid.layers.While(cond=cond)
    with while_op.block():
        # 获取解码器在当前步的输入，包括上一步选择的id，对应的score和上一步的state
        pre_ids = fluid.layers.array_read(array=ids_array, i=counter)
        pre_score = fluid.layers.array_read(array=scores_array, i=counter)
        pre_state = fluid.layers.array_read(array=state_array, i=counter)

        # 同train_decoder中的内容，进行RNN的单步计算
        pre_ids_emb = fluid.layers.embedding(
            input=pre_ids,
            size=[target_dict_size, word_dim],
            dtype='float32',
            is_sparse=is_sparse)
        out, current_state = cell(pre_ids_emb, pre_state, encoder_out,
                          encoder_out_proj)
        prob = fluid.layers.fc(
            input=current_state, size=target_dict_size, act='softmax')

        # 计算累计得分，进行beam search
        topk_scores, topk_indices = fluid.layers.topk(prob, k=beam_size)
        accu_scores = fluid.layers.elementwise_add(
            x=fluid.layers.log(topk_scores),
            y=fluid.layers.reshape(pre_score, shape=[-1]),
            axis=0)
        accu_scores = fluid.layers.lod_reset(x=accu_scores, y=pre_ids)
        selected_ids, selected_scores = fluid.layers.beam_search(
            pre_ids, pre_score, topk_indices, accu_scores, beam_size, end_id=1)

        fluid.layers.increment(x=counter, value=1, in_place=True)
        # 将 search 结果写入 tensor array 中
        fluid.layers.array_write(selected_ids, array=ids_array, i=counter)
        fluid.layers.array_write(selected_scores, array=scores_array, i=counter)
        # sequence_expand 作为 gather 使用以获取search结果对应的状态，并更新
        current_state = fluid.layers.sequence_expand(current_state,
                                                     selected_ids)
        fluid.layers.array_write(current_state, array=state_array, i=counter)
        current_enc_out = fluid.layers.sequence_expand(encoder_out,
                                                       selected_ids)
        fluid.layers.assign(current_enc_out, encoder_out)
        current_enc_out_proj = fluid.layers.sequence_expand(
            encoder_out_proj, selected_ids)
        fluid.layers.assign(current_enc_out_proj, encoder_out_proj)

        # 更新循环终止条件
        length_cond = fluid.layers.less_than(x=counter, y=max_len)
        finish_cond = fluid.layers.logical_not(
            fluid.layers.is_empty(x=selected_ids))
        fluid.layers.logical_and(x=length_cond, y=finish_cond, out=cond)

    # 根据保存的每一步的结果，回溯生成最终解码结果
    translation_ids, translation_scores = fluid.layers.beam_search_decode(
        ids=ids_array, scores=scores_array, beam_size=beam_size, end_id=1)

    return translation_ids, translation_scores
```

```
def infer_model():
    encoder_out = encoder()
    translation_ids, translation_scores = infer_decoder(encoder_out)
    return translation_ids, translation_scores
```

训练模型

构建训练程序

定义用于训练的 Program ，在其中创建训练的网络结构并添加优化器。同时还要定义用于初始化的 Program ，在创建训练网络的同时隐式的加入参数初始化的操作。

```
train_prog = fluid.Program()
startup_prog = fluid.Program()
with fluid.program_guard(train_prog, startup_prog):
    with fluid.unique_name.guard():
        avg_cost = train_model()
        optimizer = optimizer_func()
        optimizer.minimize(avg_cost)
```

定义训练环境与执行器

定义您的训练环境，可以指定训练是发生在CPU还是GPU上；并基于这个训练环境定义执行器。

```
use_cuda = False
# 定义使用设备和执行器
place = fluid.CUDAPlace(0) if use_cuda else fluid.CPUPlace()
exe = fluid.Executor(place)
```

构建数据提供器

使用封装的 paddle.dataset.wmt16.train 接口定义数据生成器，其每次产生一条样本，shuffle和组完batch后作为训练的输入；另外还需要指明输入数据中各字段和 data_layer 定义的各输入的对应关系，这可以通过 DataFeeder 完成。下面的feeder将产生数据的第一列映射到 src_word_id 这个输入。

```
# 定义训练数据生成器
train_data = paddle.batch(
    paddle.reader.shuffle(
        paddle.dataset.wmt16.train(source_dict_size, target_dict_size),
        buf_size=10000),
    batch_size=batch_size)
# DataFeeder完成
feeder = fluid.DataFeeder(
    feed_list=[
        'src_word_id', 'target_language_word', 'target_language_next_word'
    ],
    place=place,
    program=train_prog)
```

训练主循环

通过训练循环数（EPOCH_NUM）来进行训练循环，并且每次循环都保存训练好的参数。注意，循环训练前要首先执行初始化的 Program 来初始化参数。另外作为示例这里EPOCH_NUM设置较小，该数据集上实际大概需要20个epoch左右收敛。

```
# 执行初始化 Program，进行参数初始化
exe.run(startup_prog)
# 循环迭代执行训练
EPOCH_NUM = 2
for pass_id in six.moves.xrange(EPOCH_NUM):
    batch_id = 0
    for data in train_data():
        cost = exe.run(
            train_prog, feed=feeder.feed(data), fetch_list=[avg_cost])[0]
        print('pass_id: %d, batch_id: %d, loss: %f' % (pass_id, batch_id,
                                                       cost))
        batch_id += 1
```

应用模型

构建预测程序

定义用于预测的 Program ，在其中创建预测的网络结构。

```
infer_prog = fluid.Program()
startup_prog = fluid.Program()
with fluid.program_guard(infer_prog, startup_prog):
    with fluid.unique_name.guard():
        translation_ids, translation_scores = infer_model()
```

构建数据提供器

和训练类似，这里使用封装的 paddle.dataset.wmt16.test 接口定义测试数据生成器，测试数据共1000条，组完batch后作为预测的输入；另外我们获取源语言和目标语言id到word的词典，以将id序列转换为明文序列打印输出。

```
test_data = paddle.batch(
    paddle.dataset.wmt16.test(source_dict_size, target_dict_size),
    batch_size=batch_size)
src_idx2word = paddle.dataset.wmt16.get_dict(
    "en", source_dict_size, reverse=True)
trg_idx2word = paddle.dataset.wmt16.get_dict(
    "de", target_dict_size, reverse=True)
```

测试

首先要加载训练过程保存下来的模型，然后就可以循环测试数据进行预测了。这里每次运行我们都会创建 data_layer 对应输入数据的 dict 传入，这个和 DataFeeder 相同的效果。生成过程对于每个测试数据都会将源语言句子和 beam_size 个生成句子打印输出。

```
    fluid.io.load_params(exe, model_save_dir, main_program=infer_prog)

    for data in test_data():
        src_word_id = fluid.create_lod_tensor(
            data=[x[0] for x in data],
            recursive_seq_lens=[[len(x[0]) for x in data]],
            place=place)
        # init_ids内容为start token
        init_ids = fluid.create_lod_tensor(
            data=np.array([[0]] * len(data), dtype='int64'),
            recursive_seq_lens=[[1] * len(data)] * 2,
            place=place)
        # init_scores为beam search过程累积得分的初值
        init_scores = fluid.create_lod_tensor(
            data=np.array([[0.]] * len(data), dtype='float32'),
            recursive_seq_lens=[[1] * len(data)] * 2,
            place=place)
        seq_ids, seq_scores = exe.run(
            infer_prog,
            feed={
                'src_word_id': src_word_id,
                'init_ids': init_ids,
                'init_scores': init_scores
            },
            fetch_list=[translation_ids, translation_scores],
            return_numpy=False)
        # 如何解析翻译结果详见 train.py 中对应代码的注释说明
        hyps = [[] for i in range(len(seq_ids.lod()[0]) - 1)]
        scores = [[] for i in range(len(seq_scores.lod()[0]) - 1)]
        for i in range(len(seq_ids.lod()[0]) - 1):
            start = seq_ids.lod()[0][i]
            end = seq_ids.lod()[0][i + 1]
            print("Original sentence:")
            print(" ".join([src_idx2word[idx] for idx in data[i][0][1:-1]]))
            print("Translated score and sentence:")
            for j in range(end - start):
                sub_start = seq_ids.lod()[1][start + j]
                sub_end = seq_ids.lod()[1][start + j + 1]
                hyps[i].append(" ".join([
                    trg_idx2word[idx]
```

可以观察到如下的预测结果输出：

```
Original sentence:
Two adults and two children sit on a park bench .
Translated score and sentence:
-2.5993705 Zwei Erwachsene und zwei Kinder sitzen auf einer Parkbank .
-2.6617606 Zwei Erwachsene und zwei Kinder spielen auf einer Parkbank .
-3.186554 Zwei Erwachsene und zwei Kinder sitzen auf einer Bank .
-3.4353821 Zwei Erwachsene und zwei Kinder spielen auf einer Bank .
```

总结

端到端的神经网络机器翻译是近几年兴起的一种全新的机器翻译方法。本章中，我们介绍了NMT中典型的"编码器-解码器"框架。由于NMT是一个典型的Seq2Seq（Sequence to Sequence，序列到序列）学习问题，因此，Seq2Seq中的query改写（query rewriting）、摘要、单轮对话等问题都可以用本教程的模型来解决。

参考文献

1. Koehn P. Statistical machine translation[M]. Cambridge University Press, 2009.
2. Cho K, Van Merriënboer B, Gulcehre C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation[C]//Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014: 1724-1734.
3. Chung J, Gulcehre C, Cho K H, et al. Empirical evaluation of gated recurrent neural networks on sequence modeling[J]. arXiv preprint arXiv:1412.3555, 2014.
4. Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate[C]//Proceedings of ICLR 2015, 2015.
5. Papineni K, Roukos S, Ward T, et al. BLEU: a method for automatic evaluation of machine translation[C]//Proceedings of the 40th annual meeting on association for computational linguistics. Association for Computational Linguistics, 2002: 311-318.