TensorFlow：可视化、代码调试及注意力机制

2018/03/07

阮 翀

ruanchong_ruby@163.com

*全部代码已更新至 https://github.com/soloice/tf-tutorial

提纲

- 官方教程
- Attention
  - seq2seq 回顾
  - AttentionWrapper
- 可视化
  - TensorBoard
- 代码调试
- 参数保存与恢复
- BLEU

官方教程

- 刚发现 TF 官网上的教程更新了
  - 结构重新组织了一下，清晰了很多
  - 讲解更详细了，示例代码也用上了较新的接口
- Programmer’s Guide
  - 介绍 TF 中引入的一些概念及工作原理
  - https://www.tensorflow.org/programmers_guide/
- Tutorials
  - 结合示例代码讲解机器学习模型
  - https://www.tensorflow.org/tutorials/

seq2seq

- 定义模型
  - 超参数和占位符

```
encoder_embedding_size, decoder_embedding_size = 30, 30
encoder_hidden_units, decoder_hidden_units = 50, 50
encoder_lstm_layers, decoder_lstm_layers = 2, 2

# [B, T]
encoder_inputs = tf.placeholder(shape=[None, None], dtype=tf.int32, name='encoder_inputs')
decoder_targets = tf.placeholder(shape=[None, None], dtype=tf.int32, name='decoder_targets')
decoder_inputs = tf.placeholder(shape=[None, None], dtype=tf.int32, name='decoder_inputs')
encoder_length = tf.placeholder(shape=[None], dtype=tf.int32, name='encoder_length')
decoder_length = tf.placeholder(shape=[None], dtype=tf.int32, name='decoder_length')
```

seq2seq

- 定义模型
  - 词向量

```
encoder_embedding_matrix = tf.Variable(tf.truncated_normal([vocab_size, encoder_embedding_size],
                                                           mean=0.0, stddev=0.1),
                                       dtype=tf.float32, name="encoder_embedding_matrix")

decoder_embedding_matrix = tf.Variable(tf.truncated_normal([vocab_size, decoder_embedding_size],
                                                           mean=0.0, stddev=0.1),
                                       dtype=tf.float32, name="decoder_embedding_matrix")

# [B, T, D]
encoder_inputs_embedded = tf.nn.embedding_lookup(encoder_embedding_matrix, encoder_inputs)
decoder_inputs_embedded = tf.nn.embedding_lookup(decoder_embedding_matrix, decoder_inputs)
```

seq2seq

- 定义模型
  - 编码器部分：忽略编码器的输出

```
with tf.variable_scope("encoder"):
    encoder_layers = [tf.contrib.rnn.BasicLSTMCell(encoder_hidden_units)
                      for _ in range(encoder_lstm_layers)]
    encoder = tf.contrib.rnn.MultiRNNCell(encoder_layers)

    _, encoder_final_state = tf.nn.dynamic_rnn(
        encoder, encoder_inputs_embedded,
        sequence_length=encoder_length,
        dtype=tf.float32, time_major=False, scope="seq2seq_encoder")
    print(encoder_final_state)
```

seq2seq

- 定义模型
  - 解码器部分：使用 TrainingHelper 和 BasicDecoder

```
with tf.variable_scope("decoder"):
    decoder_layers = [tf.contrib.rnn.BasicLSTMCell(encoder_hidden_units)
                      for _ in range(decoder_lstm_layers)]
    decoder = tf.contrib.rnn.MultiRNNCell(decoder_layers)
    fc_layer = tf.layers.Dense(vocab_size)

    training_helper = tf.contrib.seq2seq.TrainingHelper(decoder_inputs_embedded,
                                                        decoder_length)
    training_decoder = tf.contrib.seq2seq.BasicDecoder(decoder, training_helper,
                                                       encoder_final_state, fc_layer)
```

seq2seq

- 定义模型
  - 解码器：得到每一步的 logits

```
logits, final_state, final_sequence_lengths = \
    tf.contrib.seq2seq.dynamic_decode(training_decoder)

# decoder_logits: [B, T, V]
decoder_logits = logits.rnn_output
print("logits: ", decoder_logits)
```

seq2seq

- 定义模型
  - 解码器：loss 是平均交叉熵（用掩码盖住无效部位）

```
# [B, T]
stepwise_cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(
    labels=tf.one_hot(decoder_targets, depth=vocab_size, dtype=tf.float32),
    logits=decoder_logits)
print(stepwise_cross_entropy)

mask = tf.sequence_mask(decoder_length,
                        maxlen=tf.reduce_max(decoder_length),
                        dtype=tf.float32)

loss = tf.reduce_sum(stepwise_cross_entropy * mask) / tf.reduce_sum(mask)
train_op = tf.train.AdamOptimizer().minimize(loss)
```

seq2seq

- 定义模型
  - 解码器：推断算法用贪心解码

```
num_sequences_to_decode = tf.placeholder(shape=(), dtype=tf.int32, name="num_seq")
start_tokens = tf.tile([GO], [num_sequences_to_decode])
inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
    decoder_embedding_matrix, start_tokens, end_token=EOS)

greedy_decoder = tf.contrib.seq2seq.BasicDecoder(
    cell=decoder, helper=inference_helper,
    initial_state=encoder_final_state, output_layer=fc_layer)

greedy_decoding_result, _1, _2 = tf.contrib.seq2seq.dynamic_decode(
    decoder=greedy_decoder, output_time_major=False,
    impute_finished=True, maximum_iterations=20)
```

seq2seq

- 训练过程
  - 喂数据，反复执行 train_op

```
y_in, y_out = get_decoder_input_and_output(y)
# print(x, y, lx, ly, y_in, y_out)
feed = {encoder_inputs: x,
        decoder_inputs: y_in,
        decoder_targets: y_out,
        encoder_length: lx,
        decoder_length: ly}
_, loss_ = sess.run([train_op, loss], feed_dict=feed)
```

seq2seq

- 训练过程
  - 每 100 步让模型推断一下，看看学习进度
  - 可以同时对多个序列进行 seq2seq 解码，这里同时处理 3 条序列

```
number_samples_to_draw = 3
x, y, lx, ly = generate_data(num_samples=number_samples_to_draw)

print('batch {}'.format(batch_id))
print('  minibatch loss: {}'.format(loss_))
feed = {encoder_inputs: x,
        encoder_length: lx,
        num_sequences_to_decode: number_samples_to_draw}
greedy_prediction = sess.run(greedy_decoding_result,
                             feed_dict=feed)
```

Attention

- 文档
  - https://www.tensorflow.org/versions/master/api_guides/python/contrib.seq2seq#Attention
- 原理
  - 允许解码器在解码时再次回顾源序列
  - 进而利用和当前解码步骤更相关的信息

Attention

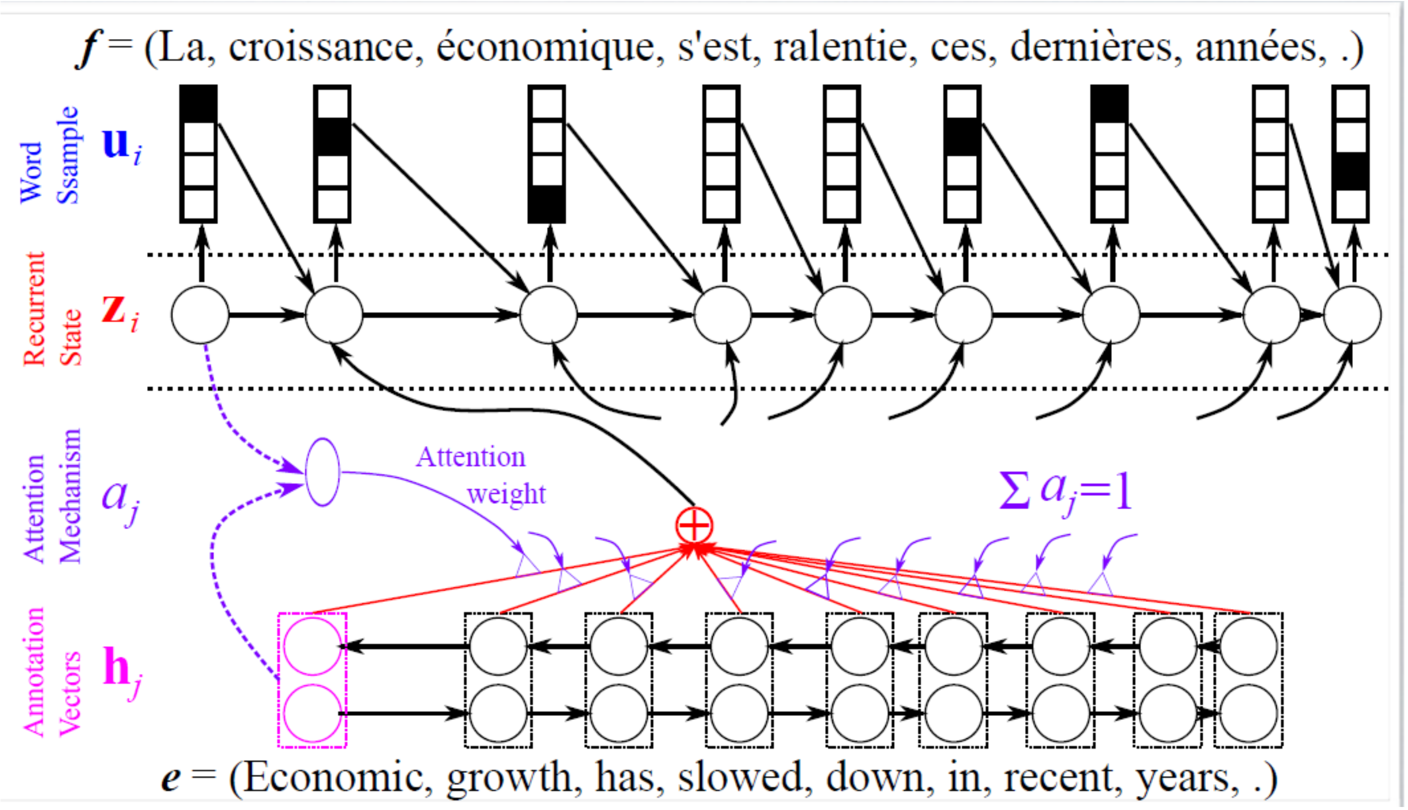
- 普通 seq2seq
  - $encoder([w_1, w_2, \ldots, w_n]) \to h_n$
    - 定长，维度为编码器隐层维度
  - $h_n \to s_0$
  - $decoder(s_{i-1}, w_i) \to s_i, y_i$

Attention

- 加入注意力机制后
  - $encoder([w_1, w_2, \dots, w_n]) \rightarrow [h_1, h_2, \dots, h_n] \triangleq C$
    - 变长，与源序列一样长
  - $C \rightarrow s_0$
    - 例如取平均，或者是取平均后再做一个线性变换
  - $decoder(s_{i-1}, w_i, C) \rightarrow s_i, y_i$
    - 不同模型/attention 算法的区别就在于如何利用 C

Attention

- 示意图



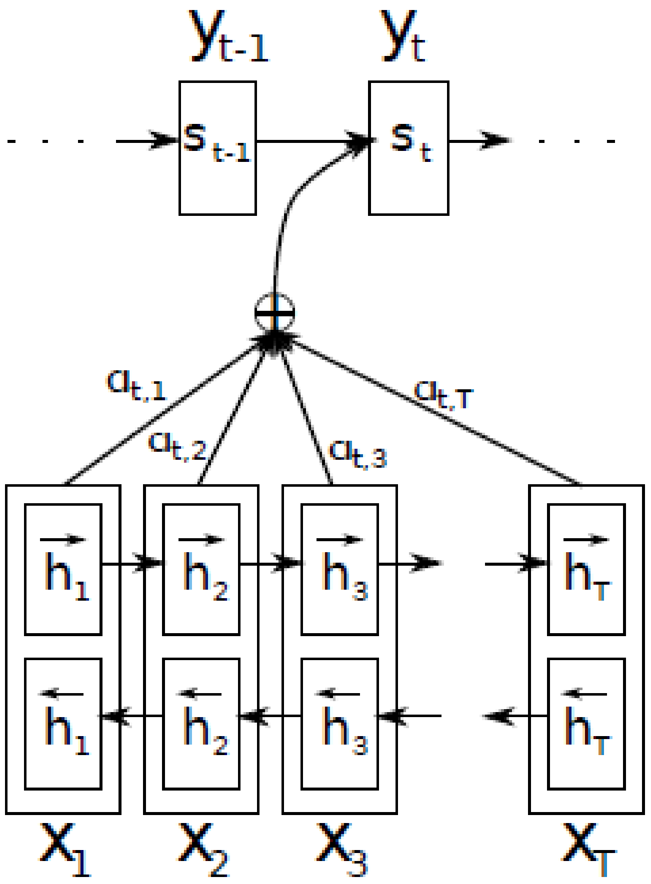
(Jörg Tiedemann, NMT-intro-2017-04-20)

Attention

- $decoder(s_{i-1}, w_i, C) \rightarrow s_i, y_i$
  - 首先生成一个查询 query
    - 想在 C 中查找什么样的信息
    - $s_{i-1}, w_i \rightarrow query$
  - 计算注意力权重
    - C 中的每一项和当前 query 有多相关
    - $query, C \rightarrow a_i$
  - 生成一个上下文向量（定长）
    - 从 C 中提取到的信息内容
    - $a_i, C \rightarrow c_i$
  - 预测当前输出
    - $c_i, w_i, s_{i-1}/s_i \rightarrow s_i, y_i$

Attention

- 两个实例
  - Bahdanau/Additive
    - 用解码器前一步的状态做查询
    - 上下文向量用在输入层
    - 打分函数为两层 MLP
  - $decoder(s_{i-1}, w_i, C) \rightarrow s_i, y_i$
    - $s_{i-1} \triangleq query$
    - $query, C \ \rightarrow a_i$
    - $a_i, C \ \rightarrow c_i$
    - $RNN([c_i, w_i], s_{i-1}) \ \rightarrow s_i, y_i$
  - *Dzmitry Bahdanau, KyungHuyn Cho, and Yoshua Bengio.* ***Neural Machine Translation by Jointly Learning to Translate and Align****. ICLR'15*



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

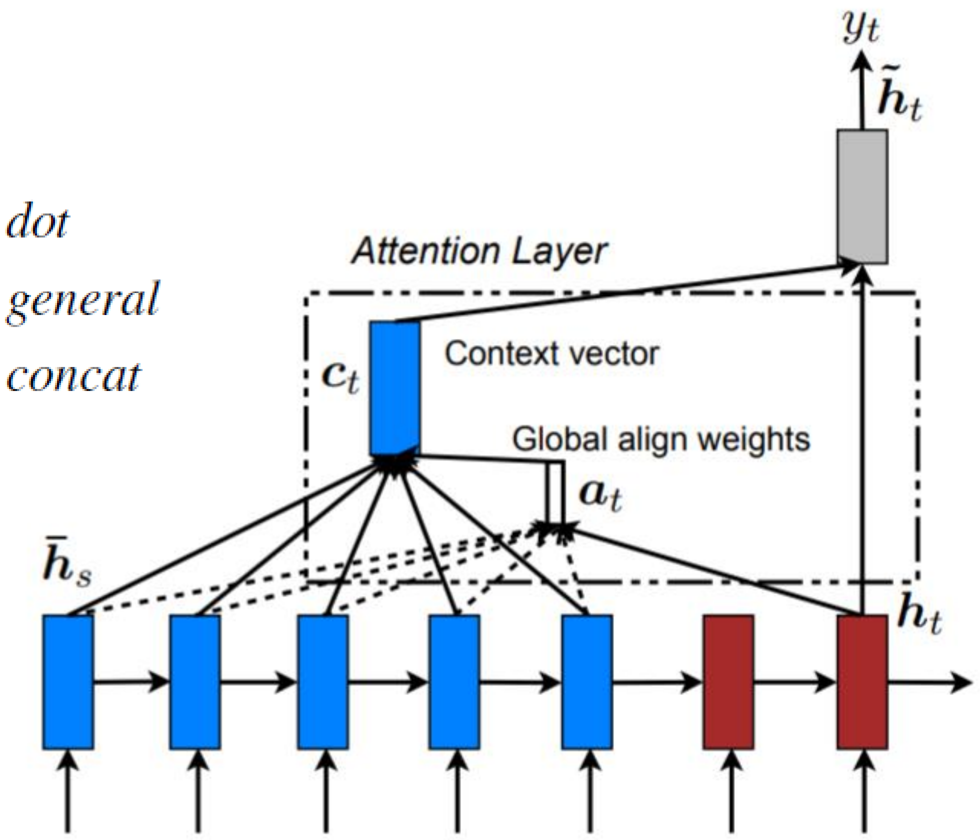
$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

$$\boxed{e_{ij} = v_a^{\top} \tanh(W_a s_{i-1} + U_a h_j)}$$

Attention

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boxed{\boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s} & \textit{general} \\ \boldsymbol{W}_a[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s] & \textit{concat} \end{cases}$$

- 两个实例
  - Luong/Multiplicative
    - 先更新解码器状态
    - 用解码器的新状态做查询
    - 打分函数为双线性
  - $decoder(s_{i-1}, w_i, C) \rightarrow s_i, y_i$
    - $RNN(s_{i-1}, w_i) \rightarrow s_i \triangleq query$
    - $query, C \rightarrow a_i$
    - $a_i, C \rightarrow c_i$
    - $2-layer-MLP(c_i, s_i) \rightarrow s_i, y_i$

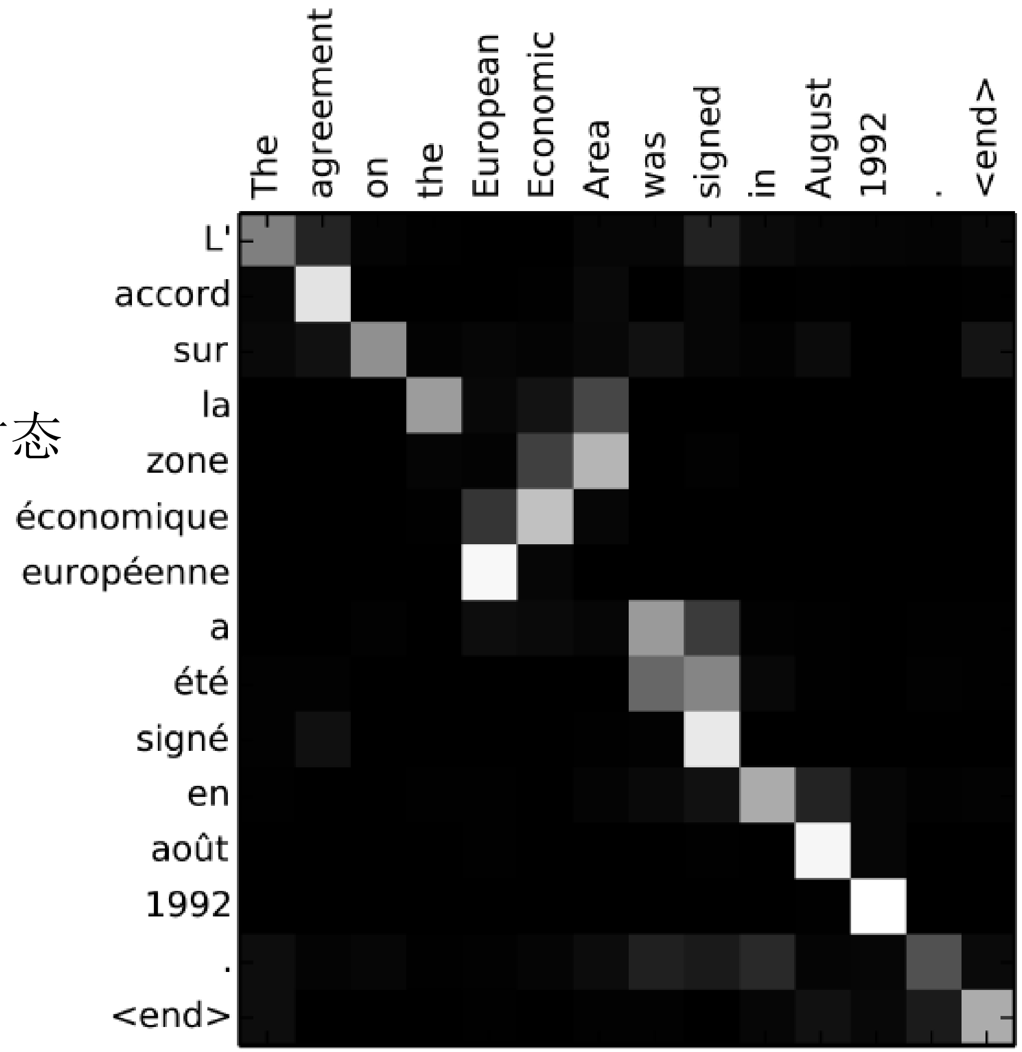


$$\begin{aligned} \boldsymbol{a}_t(s) &= \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \\ &= \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \end{aligned}$$

- *Thang Luong, Hieu Pham, and Chris Manning.* ***Effective Approaches to Attention-based Neural Machine Translation***. *EMNLP'15*

Attention

- 可视化注意力矩阵
  - 类似于软对齐，但是更一般
    - *Dzmitry Bahdanau, ICLR’15*
  - 模型可能会利用任何有帮助的信息
    - 例如在翻译动词时查看时间状语来确定时态



Attention Wrapper

- 源码
  - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py
- 文档
  - https://www.tensorflow.org/versions/master/api_docs/python/tf/contrib/seq2seq/AttentionWrapper
  - https://www.tensorflow.org/versions/master/api_docs/python/tf/contrib/seq2seq/AttentionWrapperState
  - 但还是不够详细，最好读源码

AttentionMechanism

- 继承关系
  - object
    - AttentionMechanism：仅仅实现了一些辅助函数，例如掩码的使用
      - _BaseAttentionMechanism：保存 encoder 部分的输出，提供各种对外接口
        - LuongAttention
        - BahdanauAttention
        - 这两个才是真正实际使用的类，而非抽象类

AttentionMechanism

- 基类 class _BaseAttentionMechanism
  - 构造函数需要的参数
    - num_units
      - 计算 attention 时中间隐层的维度
      - 以前面两种 attention 方法为例，即为红框部分向量的维度
    - memory
      - encoder 所有输出（或状态）
      - 形如 [B, T, D] 的张量
    - memory_sequence_length
      - 当前 batch 里每个样本的实际长度
      - 用于掩掉 memory 中的非法部分

$$e_{ij} = v_a^\top \tanh\left(\boxed{W_a s_{i-1}} + U_a h_j\right)$$

$$\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boxed{\boldsymbol{W}_a \bar{\boldsymbol{h}}_s} & \textit{general} \\ \boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s] & \textit{concat} \end{cases}$$

AttentionMechanism

- 其子类需要实现 __call__ 函数
  - 以 LuongAttention 为例
  - 参数
    - query
      - 形如 [batch_size, query_depth]
    - state
      - 前一时间步的注意力权重

```python
def __call__(self, query, state):
  """Score the query based on the keys and values.

  Args:
    query: Tensor of dtype matching `self.values` and shape
      `[batch_size, query_depth]`.
    state: Tensor of dtype matching `self.values` and shape
      `[batch_size, alignments_size]`
      (`alignments_size` is memory's `max_time`).

  Returns:
    alignments: Tensor of dtype matching `self.values` and shape
      `[batch_size, alignments_size]` (`alignments_size` is memory's
      `max_time`).
  """
  with variable_scope.variable_scope(None, "luong_attention", [query]):
    score = _luong_score(query, self._keys, self._scale)
  alignments = self._probability_fn(score, state)
  next_state = alignments
  return alignments, next_state
```

AttentionMechanism

```
Returns:
  alignments: Tensor of dtype matching `self.values` and shape
    `[batch_size, alignments_size]` (`alignments_size` is memory's
    `max_time`).
"""
with variable_scope.variable_scope(None, "luong_attention", [query]):
  score = _luong_score(query, self._keys, self._scale)
alignments = self._probability_fn(score, state)
next_state = alignments
return alignments, next_state
```

- 其子类需要实现 __call__ 函数
  - 返回值
    - alignments, next_state 值相等
    - 均形如 [batch_size, alignment_size]
      - alignment_size 就是 encoder 的时间步数
    - 表示该时间步计算出的注意力权重
      - 每一行是一个概率分布

  - 该函数的签名形式上类似于 RNNCell 的 __call__ 函数
    - 不过似乎是一个多余的设计
    - state 和 next_state 在代码别的地方都没用到

AttentionMechanism 与 AttentionWrapper

- TF 中注意力的实现需要两个类的配合
  - AttentionMechanism （及其子类）
    - 存储 encoder 输出
    - 给定 query 计算 attention 权重
    - 常使用其子类 LuongAttention 或 BahdanauAttention
  - AttentionWrapper
    - 对 RNNCell 的实例进行封装，封装后的类型还是 RNNCell
    - 从而可以和其他接口（例如 tf.nn.dynamic_rnn 等）结合使用
      - 就像使用普通的 RNNCell 一样
- 设计思想类似于 Helper 和 Decoder 的合作
  - Decoder 定义解码算法
  - Helper 负责给 Decoder 喂数据

AttentionWrapper

- RNNCell 的要点
  - __call__ 函数
    - $y_t, s_t = \_call\_(x_i, s_{i-1})$
    - 其中旧状态和新状态的类型都是该 RNNCell 的状态类型
      - 例如 LSTM 的状态类型就是 LSTMStateTuple

AttentionWrapper

- AttentionWrapper 的办法
  - 扩展状态的定义
    - 把原先的 RNN 状态变成 AttentionWrapperState 的一个域
  - AttentionWrapper 封装后的 RNNCell 依然有 __call__ 函数
    - 但是接受和生成的状态类型变了
      - 从被封装 RNNCell 的状态类型变成了 AttentionWrapperState 类型

AttentionWrapperState

- AttentionWrapperState 的各个域
  - cell_state： 被封装的 RNNCell 的状态
  - attention： 当前时间步的上下文向量（context vector）
  - time： 当前时间步数
  - alignments： 当前时间步的注意力权重（是一个概率分布）

```
class AttentionWrapperState(
    collections.namedtuple("AttentionWrapperState",
                           ("cell_state", "attention", "time", "alignments",
                            "alignment_history", "attention_state"))):
  """`namedtuple` storing the state of a `AttentionWrapper`.
```

AttentionWrapperState

- AttentionWrapperState 的各个域
  - attention_state
    - 目前的实现和 alignments 一样，都是当前时间步的注意力权重
    - 有可能是保留位，以后有别的用途
    - 或者是当时接口没设计清楚，代码重构以后忘了改
  - alignment_history
    - 从零到当前时间步的所有注意力权重
    - 显然是变长的序列（类型不是 list，而是 TensorArray）

```
class AttentionWrapperState(
    collections.namedtuple("AttentionWrapperState",
                           ("cell_state", "attention", "time", "alignments",
                            "alignment_history", "attention_state"))):
  """`namedtuple` storing the state of a `AttentionWrapper`.
```

AttentionWrapperState

- TensorArray
  - 特殊的数据类型
    - https://www.tensorflow.org/versions/master/api_docs/python/tf/TensorArray
  - 机器学习里通常不显式对变量赋值
    - 而是通过梯度下降等方式更新
  - 如何要显式操作变量？
    - tf.assign
    - 纯函数式的写法不是不可以，但是比较麻烦
  - 一种常见情景
    - 需要在变量上做循环
    - 例如把一个张量在 RNN 的各个时间步上传递
    - 每个时间步对这个张量做某种修改

AttentionWrapperState

- TensorArray 提供了一个比较优雅的解决方案
  - 可以把多个张量组织成 TensorArray（类似于张量的列表）
  - 可以读取任意下标处的元素
  - 每个位置可以写一次
  - 通常预先申明 TensorArray 的大小（能容纳的元素个数）
    - 也支持动态大小
  - 多于 tf.while_loop 结合使用
    - 沿时间轴做符号循环
    - 每个时间步做某种操作
  - 使用结束后，可以调用 stack() 方法将其转换成普通的 Tensor
    - 要求其中每个元素形状相同

AttentionWrapper

- 代码实现
  - att_seq2seq_delete_and_copy.py

```python
with tf.variable_scope("decoder"):
    decoder_layers = [tf.contrib.rnn.BasicLSTMCell(encoder_hidden_units)
                      for _ in range(decoder_lstm_layers)]
    decoder = tf.contrib.rnn.MultiRNNCell(decoder_layers)

    attention_mechanism = tf.contrib.seq2seq.LuongAttention(
        num_units=attention_depth,
        memory=encoder_all_outputs,
        memory_sequence_length=encoder_length)

    attn_decoder = tf.contrib.seq2seq.AttentionWrapper(
        decoder, attention_mechanism,
        # cell_input_fn=lambda inputs, attention: inputs,
        alignment_history=True, output_attention=True)
```

AttentionWrapper

- 代码实现
  - cell 是 RNNCell 的一个实例
    - 被封装的对象
  - attention_mechanism
    - AttentionMechanism 的一个实例
    - 其中已有 memory 张量
  - attention_layer_size
    - 得到 context vector 后是否需要再做一次变换
    - None 表示得到 context vector 就停止，否则再加一个全连接层

```
class AttentionWrapper(rnn_cell_impl.RNNCell):
  """Wraps another `RNNCell` with attention.
  """

  def __init__(self,
               cell,
               attention_mechanism,
               attention_layer_size=None,
               alignment_history=False,
               cell_input_fn=None,
               output_attention=True,
               initial_cell_state=None,
               name=None):
```

AttentionWrapper

```
class AttentionWrapper(rnn_cell_impl.RNNCell):
  """Wraps another `RNNCell` with attention.
  """

  def __init__(self,
               cell,
               attention_mechanism,
               attention_layer_size=None,
               alignment_history=False,
               cell_input_fn=None,
               output_attention=True,
               initial_cell_state=None,
               name=None):
```

- 代码实现
  - alignment_history
    - 是否保存每一步的注意力权重
    - 若是，保存在一个 TensorArray 中
  - output_attention
    - 控制每个时间步的输出选项
    - False：原 RNNCell 的输出
    - True：输出 attention 结果（context vector 或其做变换后的结果）
  - initial_cell_state
    - 打包后的 RNNCell 的初始状态
    - 默认不初始化，因为可以在别的地方初始化，如 tf.nn.dynamic_rnn

AttentionWrapper

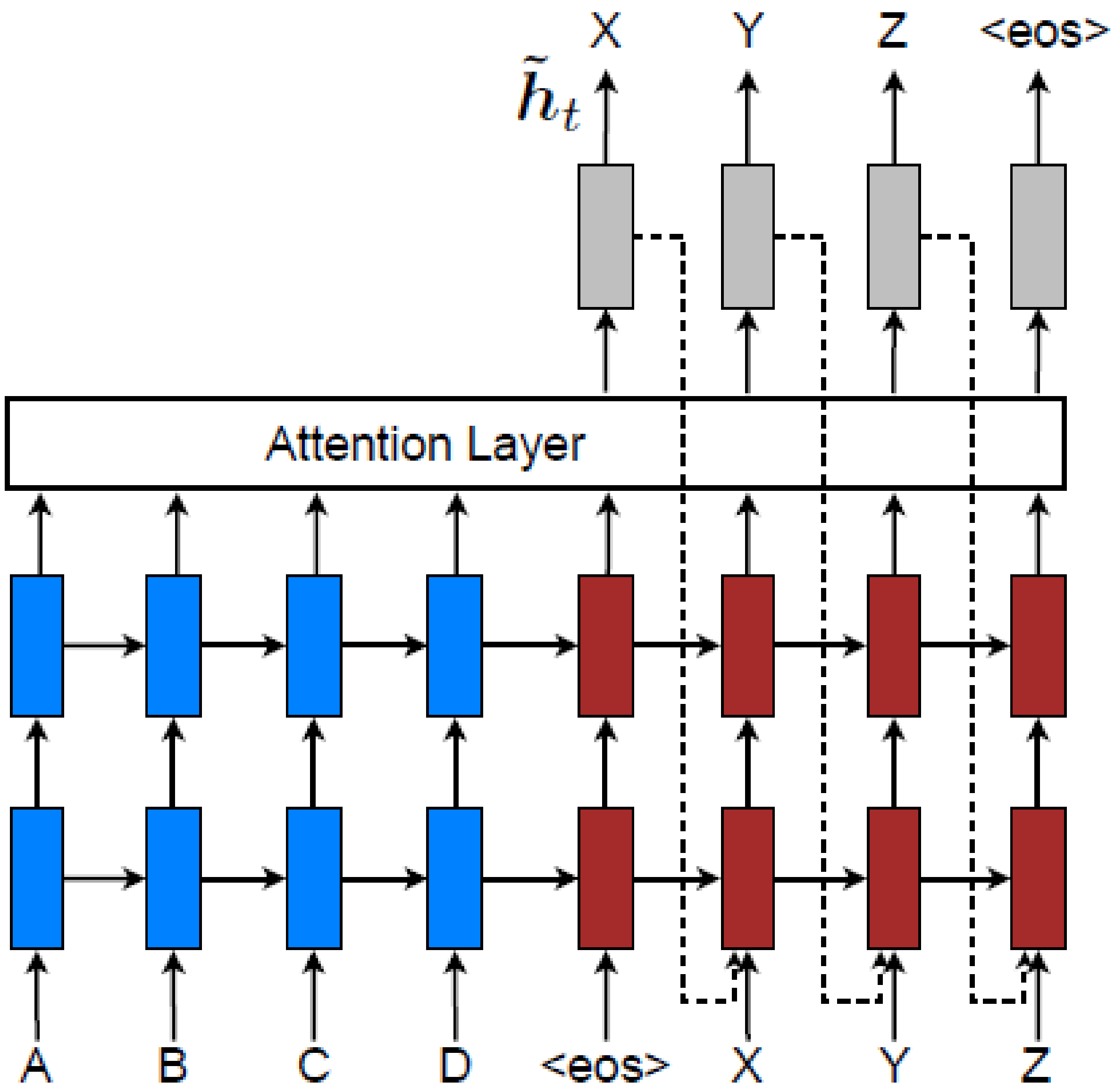
```python
class AttentionWrapper(rnn_cell_impl.RNNCell):
  """Wraps another `RNNCell` with attention.
  """

  def __init__(self,
               cell,
               attention_mechanism,
               attention_layer_size=None,
               alignment_history=False,
               cell_input_fn=None,
               output_attention=True,
               initial_cell_state=None,
               name=None):
```

- 代码实现
  - cell_input_fn
    - 函数，如何得到内层 RNNCell 的真正输入
    - 有两个参数
      - 当前输入
      - 前一步 attention 结果
    - 默认行为是把这两个值相拼接（input feeding）
    - 关闭 input feeding
      - cell_input_fn=lambda inputs, attention: inputs
      - 若 batch 大小固定，该写法可以正常运行
      - 否则 TF 的形状推断出错，疑为 TF bug
  - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py

AttentionWrapper

- Input feeding
  - *Thang Luong et al., EMNLP’15*
  - 前面的代码实现的就是这个模型



AttentionWrapper

- 整个 AttentionWrapper 计算流程的等价代码（单步）
  - https://www.tensorflow.org/versions/master/api_guides/python/contrib.seq2seq

```
cell_inputs = concat([inputs, prev_state.attention], -1)
cell_output, next_cell_state = cell(cell_inputs, prev_state.cell_state)
score = attention_mechanism(cell_output)
alignments = softmax(score)
context = matmul(alignments, attention_mechanism.values)
attention = tf.layers.Dense(attention_size)(concat([cell_output, context], 1))
next_state = AttentionWrapperState(
  cell_state=next_cell_state,
  attention=attention)
output = attention
return output, next_state
```

AttentionWrapper

- 其他源码阅读提示
  - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py
  - _compute_attention 的三个返回值
    - attention
      - context vector 或其经过再经过一个全连阶层变换的结果
    - alignments, next_attention_state
      - 都是当前步骤的注意力权重
      - 概率分布，长度为 encoder 的时间步数

AttentionWrapper

- 代码实现
  - 需要修改之前对 tf.contrib.seq2seq.BasicDecoder 的调用
    - 解码器部分 RNNCell 的状态类型变了
  - 用 AttentionWrapper 的 clone 方法重新封装编码器的最终状态

```
decoder_initial_state = attn_decoder.zero_state(batch_size, tf.float32).clone(
    cell_state=encoder_final_state)

training_decoder = tf.contrib.seq2seq.BasicDecoder(
    cell=attn_decoder, helper=training_helper,
    initial_state=decoder_initial_state, output_layer=fc_layer)
```

AttentionWrapper

- 代码实现
  - 提取 alignment_history，为后面可视化做准备

```
logits, final_state, final_sequence_lengths = \
    tf.contrib.seq2seq.dynamic_decode(training_decoder)

# decoder_logits: [B, T, V]
decoder_logits = logits.rnn_output
# [decoder_steps, batch_size, encoder_steps]
attention_matrices = final_state.alignment_history.stack(
    name="train_attention_matrix")
print("logits: ", decoder_logits)
```

AttentionWrapper

- 代码实现
  - 推断部分也做类似处理
    - 修改起始状态

```
inference_decoder_initial_state = attn_decoder.zero_state(
    num_sequences_to_decode, tf.float32).clone(
    cell_state=encoder_final_state)

greedy_decoder = tf.contrib.seq2seq.BasicDecoder(
    cell=attn_decoder, helper=inference_helper,
    initial_state=inference_decoder_initial_state, output_layer=fc_layer)
```

AttentionWrapper

- 代码实现
  - 推断部分也做类似处理
    - 获得推断时的注意力矩阵

```
greedy_decoding_result, greedy_final_state, _2 = tf.contrib.seq2seq.dynamic_decode(
    decoder=greedy_decoder, output_time_major=False,
    impute_finished=True, maximum_iterations=MAX_DECODE_STEP)

# [decoder_steps, batch_size, encoder_steps]
inference_attention_matrices = greedy_final_state.alignment_history.stack(
    name="inference_attention_matrix")
```

AttentionWrapper

- 任务描述
  - 原先的任务有点简单
    - 删掉奇数，留下偶数
  - 进阶版
    - 删掉奇数，留下偶数，再把偶数序列重复一遍
    - 要求模型回顾两遍原序列

AttentionWrapper

- 对生成数据的函数稍作修改即可
  - 加一个参数 copy_sequence

```
def generate_data(num_samples=batch_size, copy_sequence=True):
    num_odds = np.random.randint(low=1, high=max_len//2, size=num_samples)
    num_evens = np.random.randint(low=1, high=max_len//2, size=num_samples)
    batch_len_x = num_odds + num_evens
    if copy_sequence:
        batch_len_y = num_evens * 2 + 1  # append <EOS> (or prepend <GO>)
    else:
        batch_len_y = num_evens + 1  # append <EOS> (or prepend <GO>)
```

```
sample_y = list(filter(lambda x: x % 2 == 0, sample_x))
if copy_sequence:
    sample_y += sample_y
```

AttentionWrapper

- 普通 seq2seq 在新任务上的效果
  - seq2seq.py
  - 代码已更新，推到了 github 上

Attention Wrapper

- 可视化
  - 普通 seq2seq
  - step 1k

```
batch 1000
  minibatch loss: 0.8904578685760498
============================================================================
Sample x:
[[ 2  9  5  2 10  8  1  9 10  8 10  8  7  5  7  0  0  0]
 [ 4  6  6  4  8  5  1  5  1  0  0  0  0  0  0  0  0  0]
 [ 1  8  2  6  8  1  9  7 10  1 10  1  9  8  4  3  5  6]]
Expected y:
[[ 2  2 10  8 10  8 10  8  2  2 10  8 10  8 10  8  0  0  0]
 [ 4  6  6  4  8  4  6  6  4  8  0  0  0  0  0  0  0  0  0]
 [ 8  2  6  8 10 10  8  4  6  8  2  6  8 10 10  8  4  6  0]]
Greedy Decoding result:
[[ 2  8 10  8  8 10  8  8  8  8  6  8  8  6  8  8  0  0  0]
 [ 4  6  6  4  4  8  6  6  4  4  0  0  0  0  0  0  0  0  0]
 [ 8  8 10  8  4 10  8 10  8  8  6  8  8  6  6  8  8  4  0]]
```

Attention Wrapper

- 可视化
  - 普通 seq2seq
  - step 2k

```
batch 2000
  minibatch loss: 0.4207509756088257
==============================================================
Sample x:
[[ 4  8 10  8  5 10 10  2  8  2  0  0  0  0  0]
 [ 1 10  3  9  2  3  4  5  6  5  8  5  8  3  9]
 [ 2  5  1  5  7  5  8 10  1  7  5  0  0  0  0]]
Expected y:
[[ 4  8 10  8 10 10  2  8  2  4  8 10  8 10 10  2  8  2  0]
 [10  2  4  6  8  8 10  2  4  6  8  8  0  0  0  0  0  0  0]
 [ 2  8 10  2  8 10  0  0  0  0  0  0  0  0  0  0  0  0  0]]
Greedy Decoding result:
[[ 4  8 10 10  8  8  2  4 10  8  8 10  2 10  4  8 10  2  0]
 [10  4  8  2  6  8  6  8  2  4 10  8  0  0  0  0  0  0  0]
 [ 2  8 10  2  8 10  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

Attention Wrapper

- 可视化
  - 普通 seq2seq
  - step 3k

```
batch 3000
  minibatch loss: 0.28110671043395996
================================================================
Sample x:
[[ 9  5  7  4  9  2  5  8  6  8  9  7  2  8  8  3  1  4  2  7]
 [ 8  8  8 10  6  8  3  8  6  0  0  0  0  0  0  0  0  0  0  0]
 [ 8  9  3 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
Expected y:
[[ 4  2  8  6  8  2  8  8  4  2  4  2  8  6  8  2  8  8  4  2  0]
 [ 8  8  8 10  6  8  8  6  8  8  8 10  6  8  8  6  0  0  0  0  0]
 [ 8 10  8 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
Greedy Decoding result:
[[ 4  8  2  6  8  8  2  4  2  8  4  8  2  6  8  2  8  4  8  2]
 [ 8  8  8 10  6  8  8  6  8  8  8 10  8  6  8  6  0  0  0  0]
 [ 8 10  8 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

Attention Wrapper

- 可视化
  - 普通 seq2seq
  - step 4k
    - 全对
    - 但是只是运气好序列短

```
batch 4000
  minibatch loss: 0.2374555468559265
=====================================================
Sample x:
[[ 1  9  7  6  7  9  5  5  7  2  3  4  1  5 10]
 [ 7  9  3  6  9  7  1  3  6  9  5  9  0  0  0]
 [ 2  6  4  3  9  2  5  9  9  7  1  3  0  0  0]]
Expected y:
[[ 6  2  4 10  6  2  4 10  0]
 [ 6  6  6  6  0  0  0  0  0]
 [ 2  6  4  2  2  6  4  2  0]]
Greedy Decoding result:
[[ 6  2  4 10  6  2  4 10  0]
 [ 6  6  6  6  0  0  0  0  0]
 [ 2  6  4  2  2  6  4  2  0]]
```

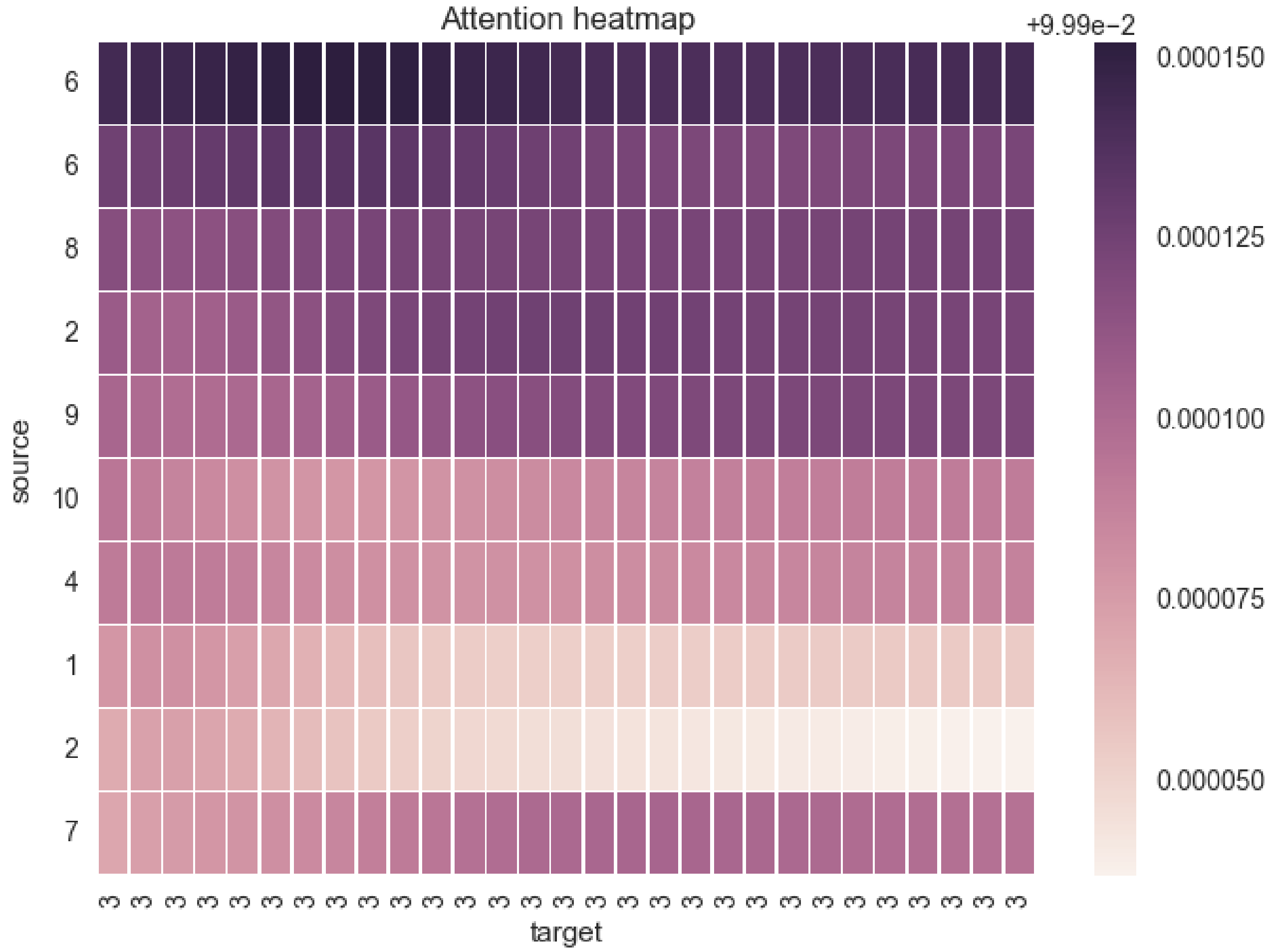
Attention Wrapper

- 可视化
  - 普通 seq2seq
  - step 5k
    - 长句还是有错
    - 这个任务确实难一些

```
batch 5000
  minibatch loss: 0.12330718338489532
=================================================================
Sample x:
[[ 9  9  8  1 10  4  0  0  0  0  0]
 [ 2  5  6  9 10  4  8  0  0  0  0]
 [ 2  7  4  2  8  6  2  8  4  9  8]]
Expected y:
[[ 8 10  4  8 10  4  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  6 10  4  8  2  6 10  4  8  0  0  0  0  0  0  0  0  0]
 [ 2  4  2  8  6  2  8  4  8  2  4  2  8  6  2  8  4  8  0]]
Greedy Decoding result:
[[ 8 10  4  8 10  4  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  6 10  4  8  2  6 10  4  8  0  0  0  0  0  0  0  0  0]
 [ 2  4  2  8  8  6  2  4  2  8  4  8  2  8  6  2  4  2  0]]
```

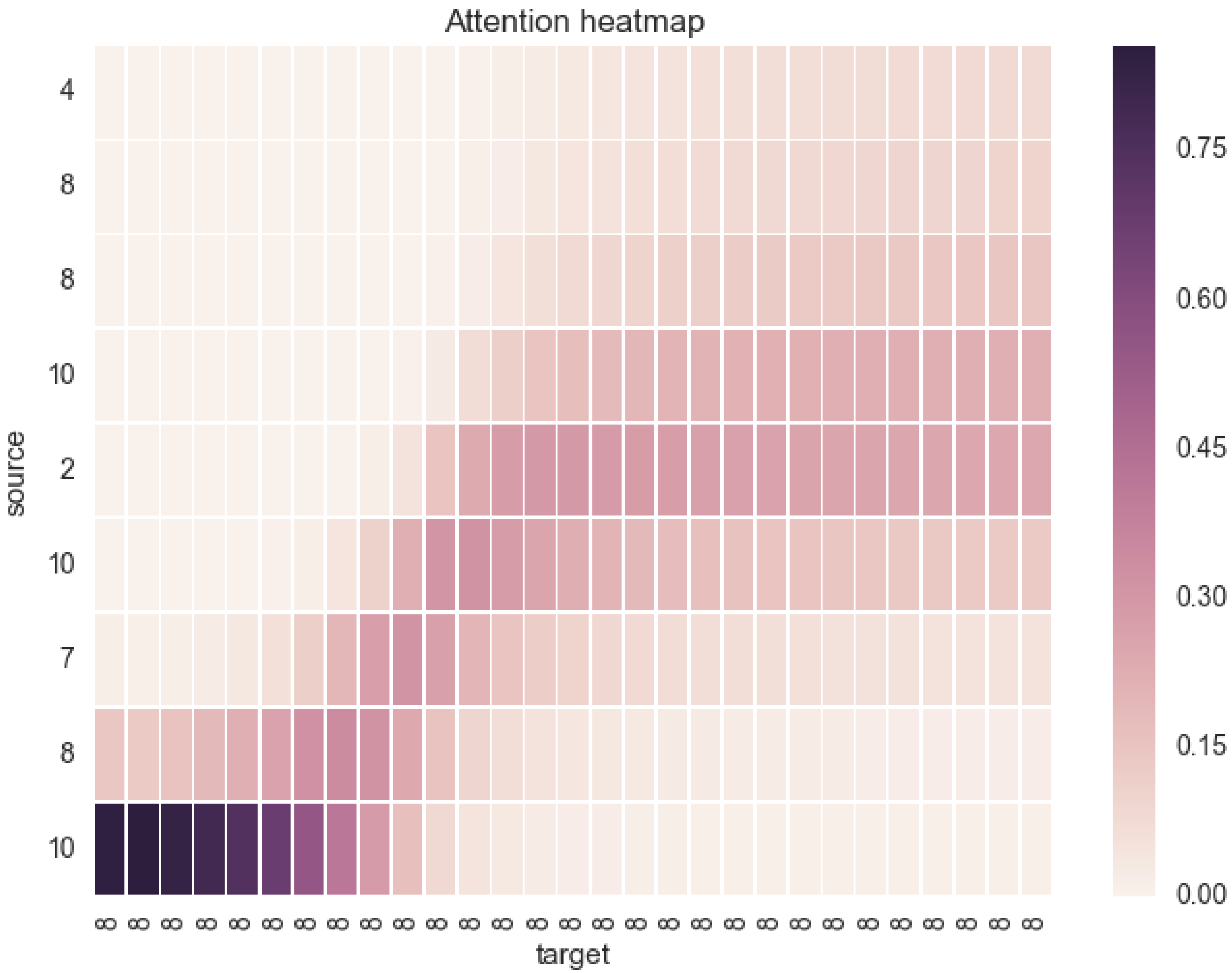
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 0
  - 随机初始化，全屏乱看



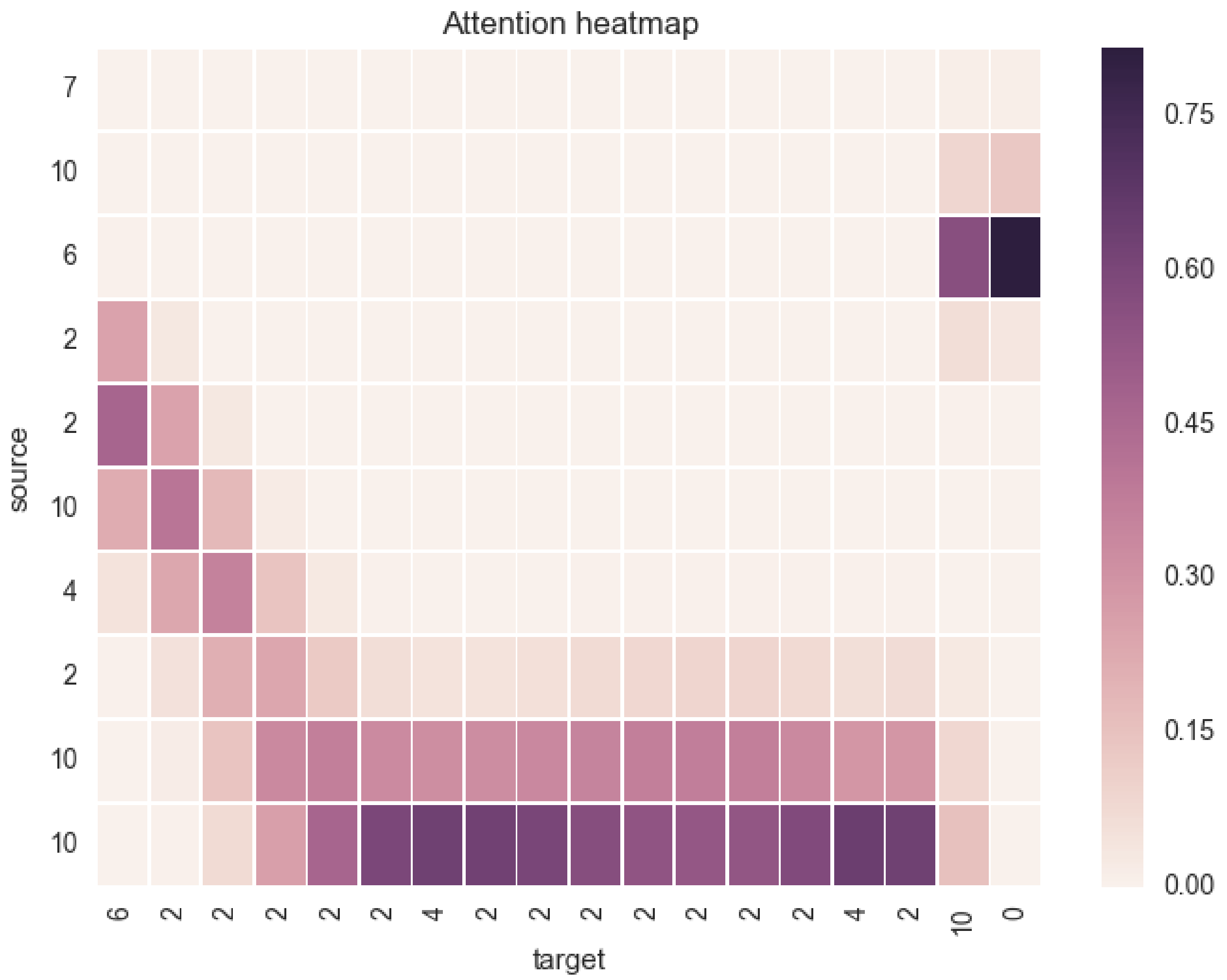
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 100
  - 开始关注个别偶数（10）



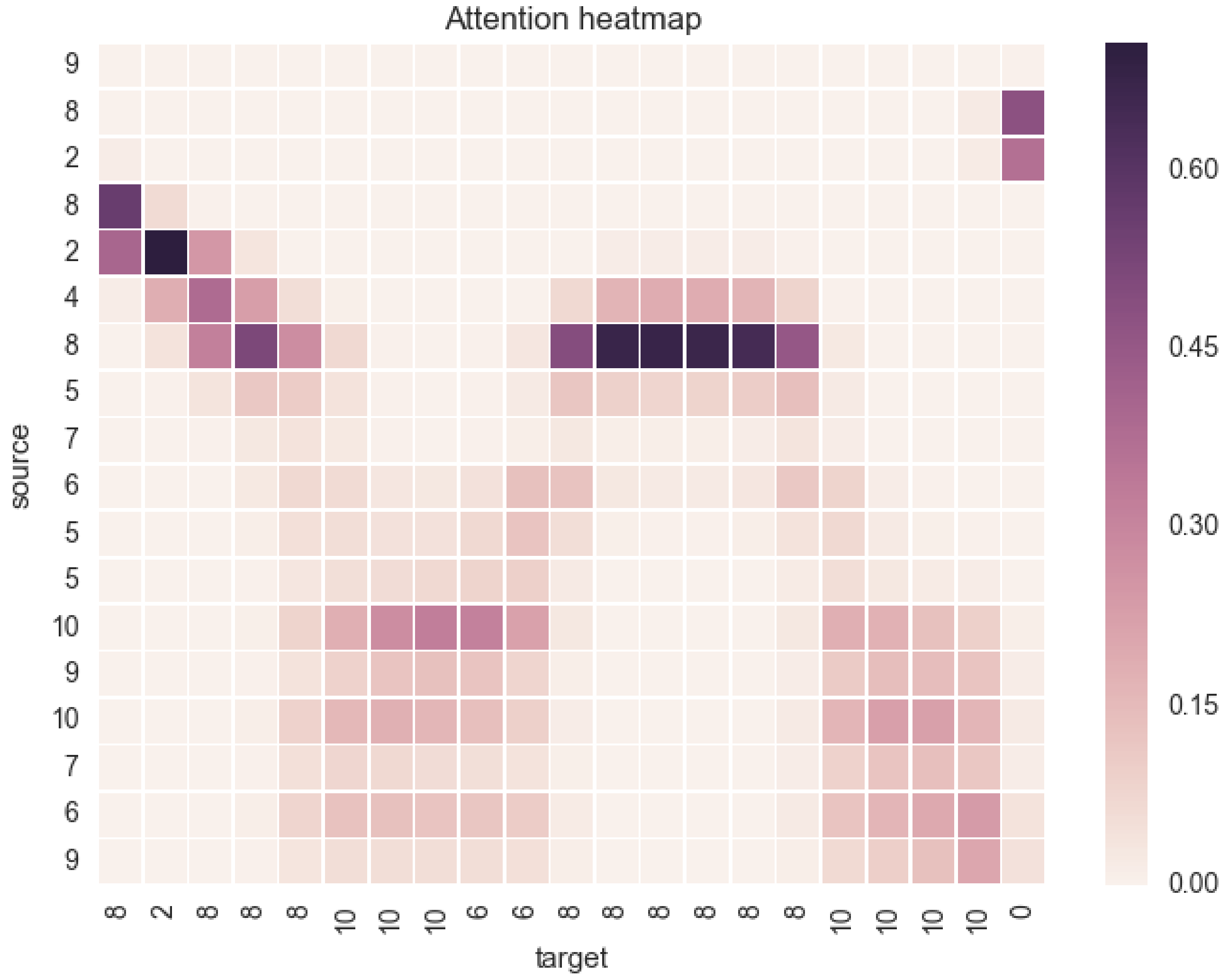
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 400
  - 认识了更多偶数



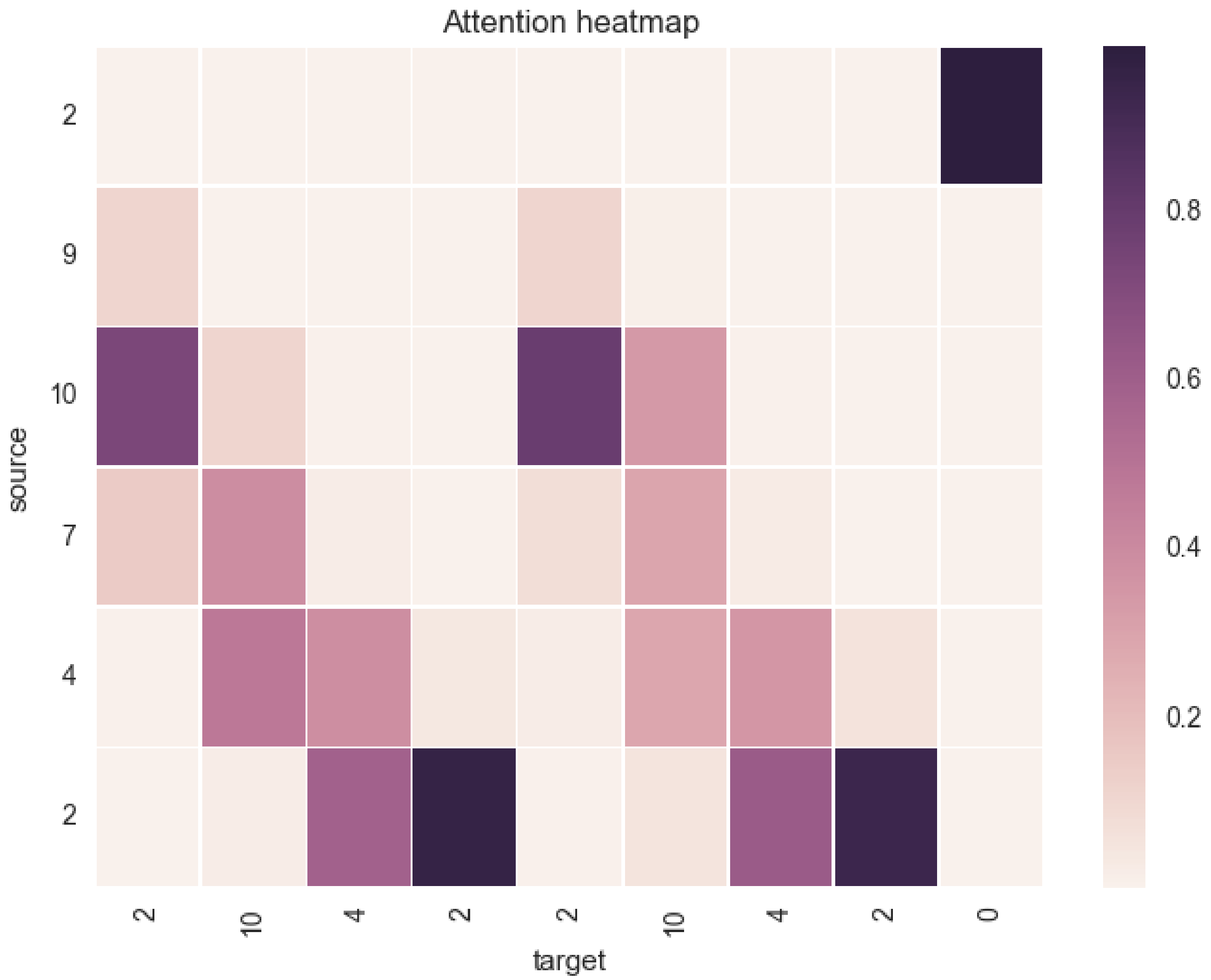
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 600
  - 开始意识到要走两遍了
  - 虽然输出还不太对



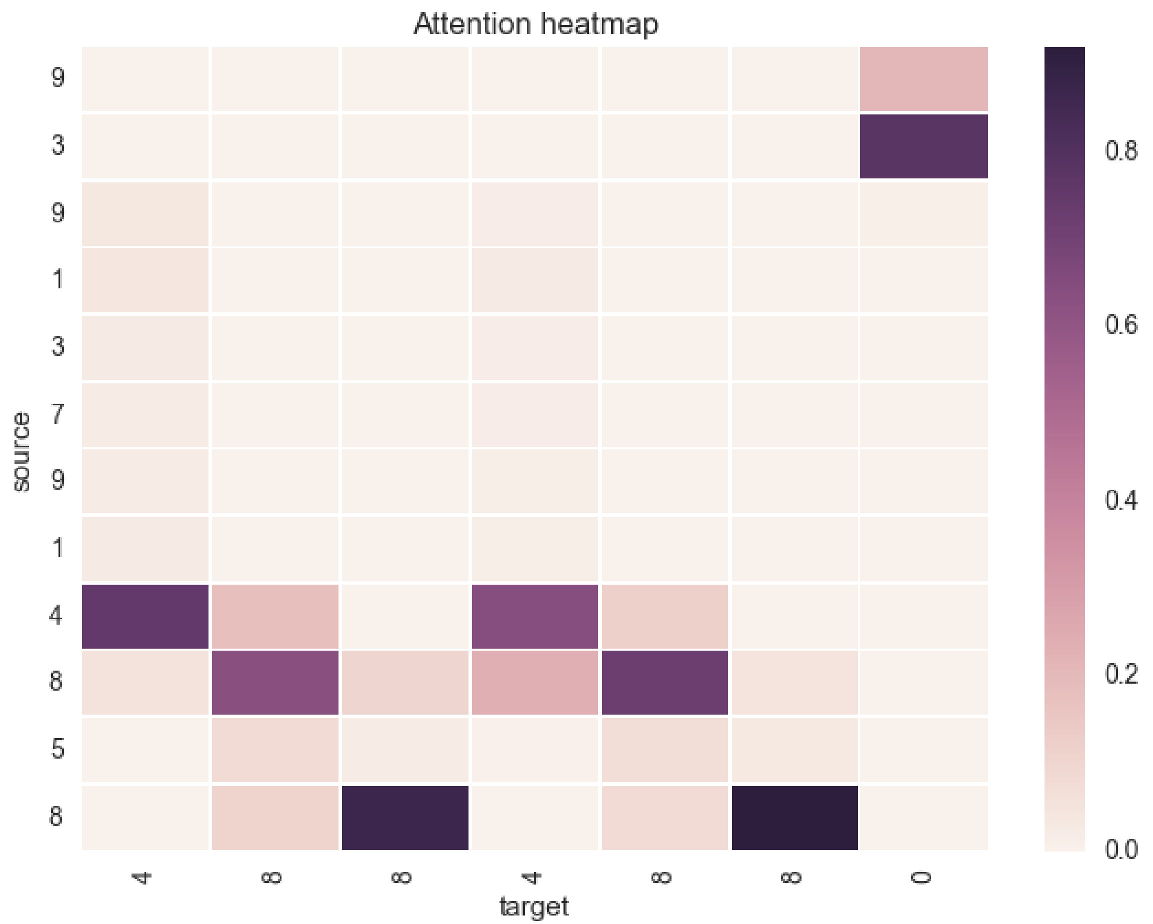
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 800
  - 短序列对了
    - 不过有点蒙的成分
    - 没有看最开头的 2



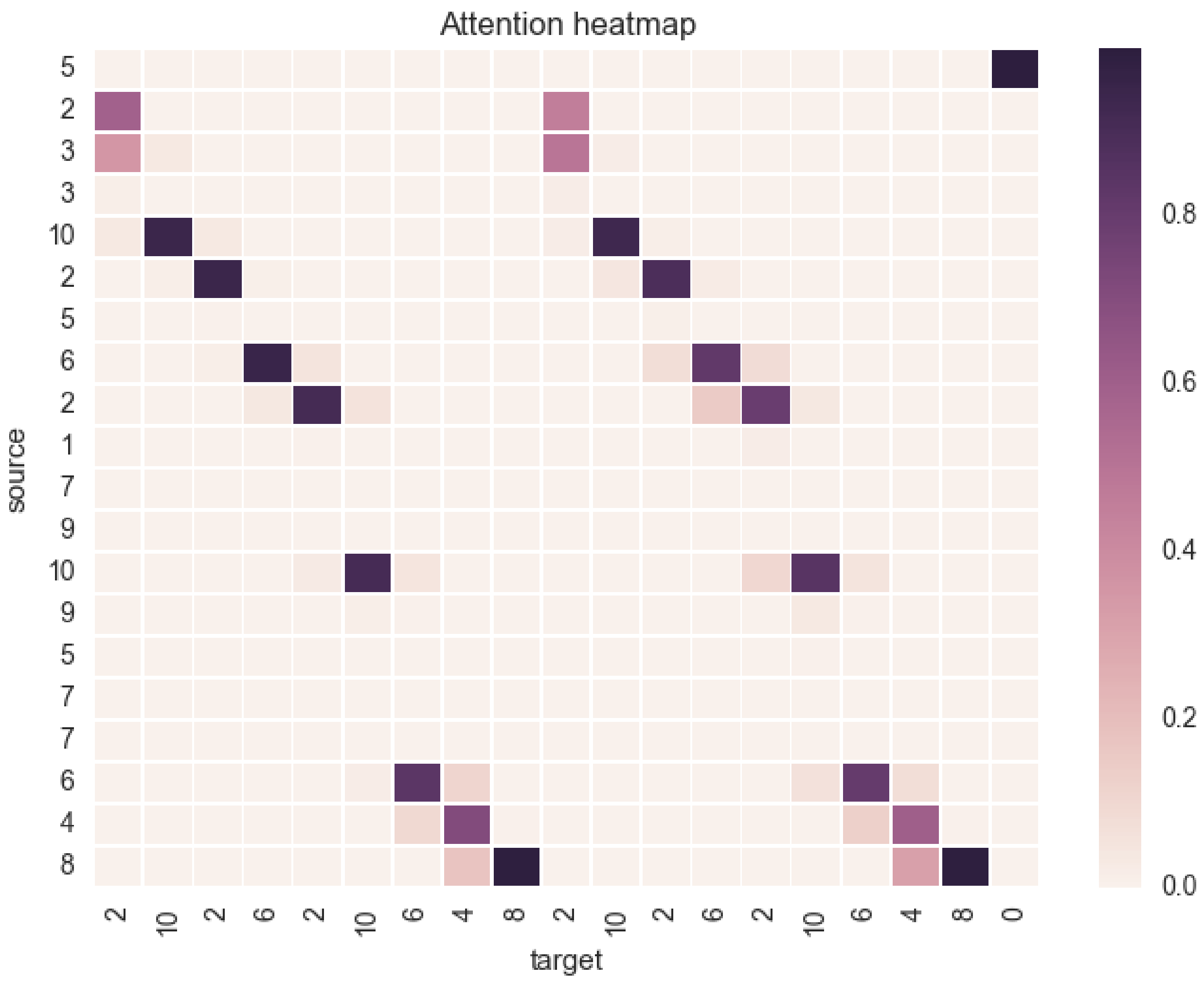
Attention Wrapper

- 可视化
  - 加入注意力后
  - step 1.1k
  - 这次看来是正经答对了



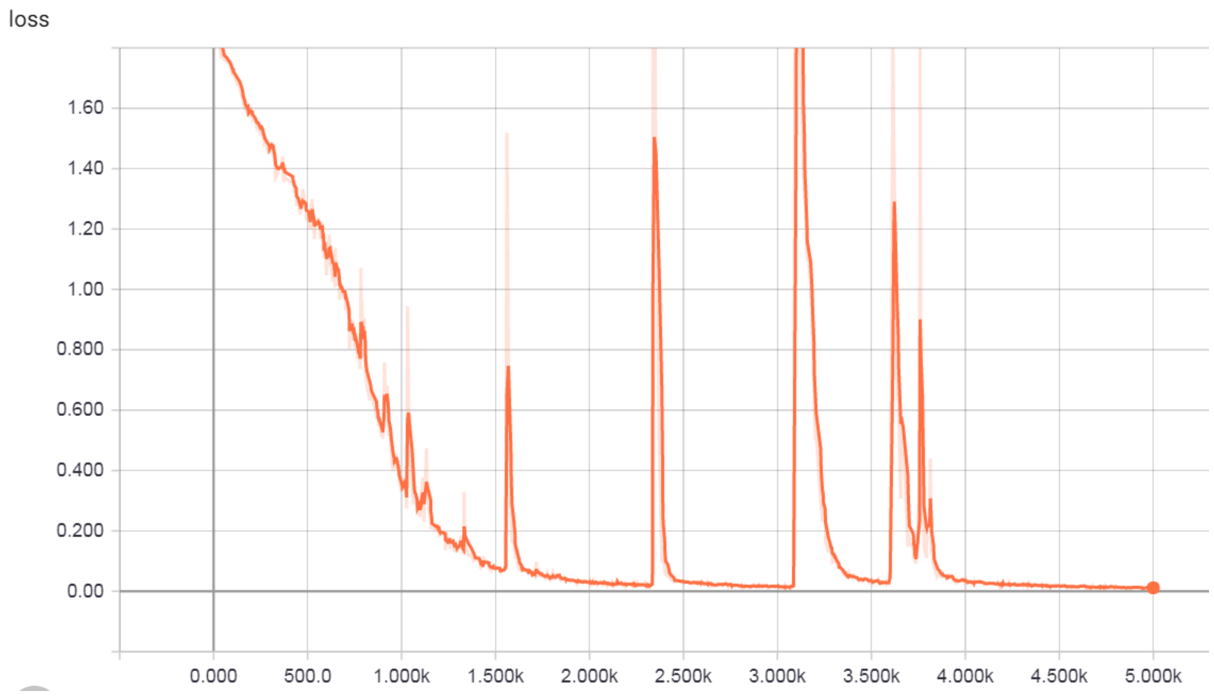
Attention Wrapper

- 可视化
    - 加入注意力后
    - step 2.3k
    - 注意力矩阵很漂亮



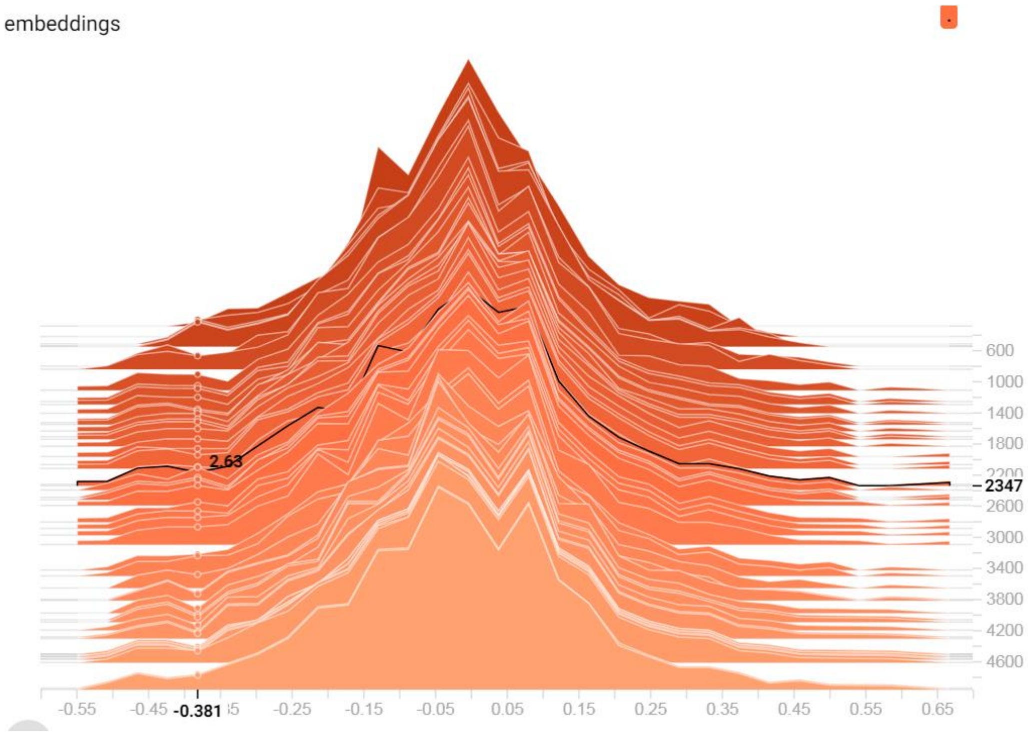
Attention Wrapper

- 可视化
  - 使用 Tensorboard，后面讲
  - loss 抖的原因可能是没做 gradient clipping



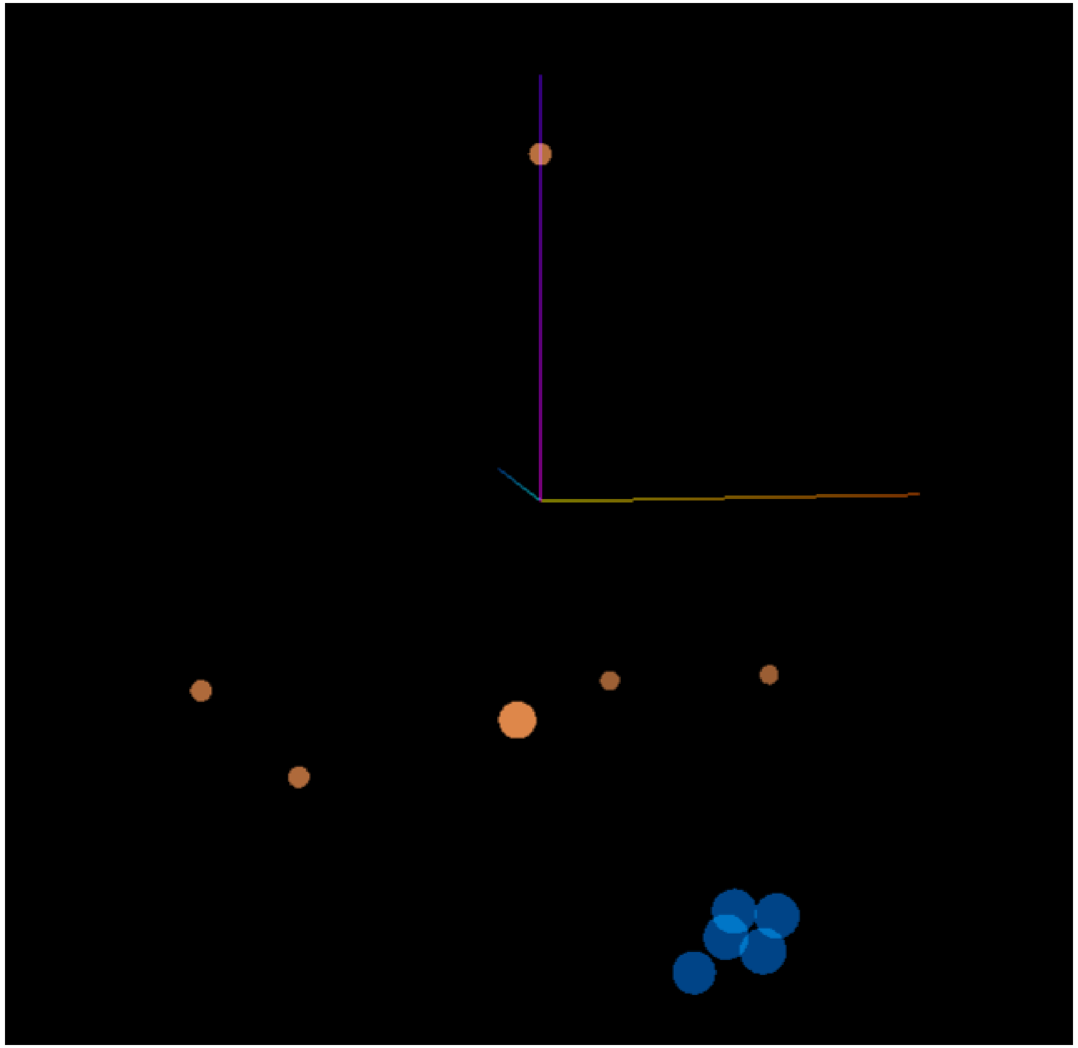
Attention Wrapper

- 可视化
  - 1k 步之后词向量就不怎么变了



Attention Wrapper

- 可视化
  - embedding 结果
  - 把 30 维的数字向量投影到 3 维
    - PCA 算法
  - 蓝色点是奇数，橘色是偶数
    - 奇数缩到了一起，因为没有信息量



Attention Wrapper

- 可视化 embedding 的代码
  - tsv: tab separated file
  - 第一行是元信息
  - 后面的行是每个样本的信息

```python
from tensorflow.contrib.tensorboard.plugins import projector
```

```python
label_file_name = "labels.tsv"
with open(os.path.join(model_path, label_file_name), "w") as f:
    f.write("Number\tIsOdd\n")
    for i in range(vocab_size):
        f.write(str(i) + "\t" + str(i%2) + "\n")
```

```
Number	IsOdd
0	0
1	1
2	0
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	0
```

Attention Wrapper

- 可视化 embedding 的代码
  - 标签文件要和模型检查点存到同一个目录下

```
train_writer = tf.summary.FileWriter(model_path, sess.graph)
config = projector.ProjectorConfig()
# config.model_checkpoint_path = model_name
embedding = config.embeddings.add()
embedding.tensor_name = encoder_embedding_matrix.name
embedding.metadata_path = label_file_name
projector.visualize_embeddings(train_writer, config)
```

Attention Wrapper

- 其他
  - 由于每次训练时会先检查是否有已保存的检查点
  - 所以如果程序意外中断（或是被 Ctrl+C 杀掉等）
  - 重启后可以接着原先的进度训练，而不是从头开始

可视化

- Tensorboard
  - 可以用于可视化损失函数/参数分布/模型结构等
  - 官方教程
    - https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard
    - 共三篇
  - 文档
    - https://www.tensorflow.org/api_guides/python/summary

可视化

- API 在 tf.summary 模块下
- 大致流程
  - 把想要记录的信息用 tf.summary 加到计算图中
  - 定义一个 FileWriter 类，打开文件
  - 在训练时不断向这个文件写入 summary

MNIST

- 三层 MLP 做数字分类 + summary
  - 改编自官方教程中的示例代码 https://github.com/tensorflow/tensorflow/blob/r1.6/tensorflow/examples/tutorials/mnist/mnist_with_summaries.py

- 准确率大约 96%~97%

MNIST

- 设置超参数
  - argparse 模块：解析命令行参数

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--fake_data', nargs='?', const=True, type=bool,
                        default=False,
                        help='If true, uses fake data for unit testing.')
    parser.add_argument('--max_steps', type=int, default=1000,
                        help='Number of steps to run trainer.')
    parser.add_argument('--learning_rate', type=float, default=0.001,
                        help='Initial learning rate')
    parser.add_argument('--dropout', type=float, default=0.9,
                        help='Keep probability for training dropout.')
```

MNIST

- 设置超参数
  - 默认的数据文件和日志文件的目录

```
parser.add_argument(
    '--data_dir',
    type=str,
    default=os.path.join(os.getenv('TEST_TMPDIR', '/tmp'),
                         'tensorflow/mnist/input_data'),
    help='Directory for storing input data')
parser.add_argument(
    '--log_dir',
    type=str,
    default=os.path.join(os.getenv('TEST_TMPDIR', '/tmp'),
                         'tensorflow/mnist/logs/mnist_with_summaries'),
    help='Summaries log directory')
FLAGS, unparsed = parser.parse_known_args()
tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

MNIST

- TF 官方示例代码喜欢把程序入口放到一个叫 main(_) 的函数里
- 然后用 tf.app.run(main=main, ...) 来调用
- 其实没什么用，绑架用户习惯而已

```python
def main(_):
  if tf.gfile.Exists(FLAGS.log_dir):
    tf.gfile.DeleteRecursively(FLAGS.log_dir)
  tf.gfile.MakeDirs(FLAGS.log_dir)
  train()
```

杂

- 更坑爹的是 tf.app.flags
- Python 已有解析命令行的工具
  - argparse 是事实上的标准，使用非常广泛
- TF 非要自己重新造一个解析命令行参数的模块，绑架用户
- 一些 Google 研究员的代码就用 argparse 不用 tf.app.flags
  - 这帮人就非常上道
- 技术是技术，商业是商业……

MNIST

- 把输入图像加入日志，最多存 10 张图
  - 注：目前 API 形如 tf.summary.<xxx>，在 TF 1.0 之前形如 tf.<xxx>_summary

```
# Import data
mnist = input_data.read_data_sets(FLAGS.data_dir,
                                  fake_data=FLAGS.fake_data)

sess = tf.InteractiveSession()
# Create a multilayer model.

# Input placeholders
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None, 784], name='x-input')
    y_ = tf.placeholder(tf.int64, [None], name='y-input')

with tf.name_scope('input_reshape'):
    image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])
    tf.summary.image('input', image_shaped_input, 10)
```

MNIST

- 定义变量的辅助函数
  - 类似前一讲

```python
# We can't initialize these variables to 0 - the network will get stuck.
def weight_variable(shape):
    """Create a weight variable with appropriate initialization."""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    """Create a bias variable with appropriate initialization."""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

MNIST

- 定义写日志的辅助函数
  - 把张量的一些统计量（均值、方差、最值）和直方图都写进文件

```python
def variable_summaries(var):
    """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar('stddev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

MNIST

- 定义一层神经网络的辅助函数
  - 把参数和激活值都加到日志里

```python
def nn_layer(input_tensor, input_dim, output_dim, layer_name, act=tf.nn.relu):
    # Adding a name scope ensures logical grouping of the layers in the graph.
    with tf.name_scope(layer_name):
        # This Variable will hold the state of the weights for the layer
        with tf.name_scope('weights'):
            weights = weight_variable([input_dim, output_dim])
            variable_summaries(weights)
        with tf.name_scope('biases'):
            biases = bias_variable([output_dim])
            variable_summaries(biases)
        with tf.name_scope('Wx_plus_b'):
            preactivate = tf.matmul(input_tensor, weights) + biases
            tf.summary.histogram('pre_activations', preactivate)
        activations = act(preactivate, name='activation')
        tf.summary.histogram('activations', activations)
        return activations
```

MNIST

- 定义 MLP
  - 784-500-dropout (0.9)-10

```
hidden1 = nn_layer(x, 784, 500, 'layer1')

with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
    tf.summary.scalar('dropout_keep_probability', keep_prob)
    dropped = tf.nn.dropout(hidden1, keep_prob)

# Do not apply softmax activation yet, see below.
y = nn_layer(dropped, 500, 10, 'layer2', act=tf.identity)
```

MNIST

- 定义交叉熵损失函数并记录到日志中
- 定义训练用的 Op

```
with tf.name_scope('cross_entropy'):
    # So here we use tf.losses.sparse_softmax_cross_entropy on the
    # raw logit outputs of the nn_layer above, and then average across
    # the batch.
    with tf.name_scope('total'):
        cross_entropy = tf.losses.sparse_softmax_cross_entropy(
            labels=y_, logits=y)
tf.summary.scalar('cross_entropy', cross_entropy)

with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer(FLAGS.learning_rate).minimize(
        cross_entropy)
```

MNIST

- 定义准确率并记录到日志中

```
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y, 1), y_)
    with tf.name_scope('accuracy'):
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', accuracy)
```

MNIST

- tf.summary.scalar 等函数是在向计算图中添加节点（Op）
- tf.summary.merge_all() 可以把所有 summary 相关的 Op 合成一个
- 定义 FileWriter 对象，指定向哪个文件夹里写日志
  - 再传一个 graph 参数可以把计算图也存下来
  - 可以定义任意多个 FileWriter，这里用了两个

```
# Merge all the summaries and write them out to
# /tmp/tensorflow/mnist/logs/mnist_with_summaries (by default)
merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter(FLAGS.log_dir + '/train', sess.graph)
test_writer = tf.summary.FileWriter(FLAGS.log_dir + '/test')
tf.global_variables_initializer().run()
```

MNIST

- 生成要喂给网络的字典

```python
def feed_dict(train):
    """Make a TensorFlow feed_dict: maps data onto Tensor placeholders."""
    if train or FLAGS.fake_data:
        xs, ys = mnist.train.next_batch(100, fake_data=FLAGS.fake_data)
        k = FLAGS.dropout
    else:
        xs, ys = mnist.test.images, mnist.test.labels
        k = 1.0
    return {x: xs, y_: ys, keep_prob: k}
```

MNIST

- 开始训练
  - 每 10 步从测试集取一组数据
  - 计算准确率，并执行 summary 操作写入日志
  - FileWriter().add_summary() 的第二个参数是 global step

```
for i in range(FLAGS.max_steps):
    if i % 10 == 0:  # Record summaries and test-set accuracy
        summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
        test_writer.add_summary(summary, i)
        print('Accuracy at step %s: %s' % (i, acc))
```

MNIST

- 在每 10 步剩下的 9 步里执行训练
  - 每 100 步加入元信息（内存占用、运行时间等）
  - 不用担心每一步都写日志会占满硬盘，TF 自己会对步数做蓄水池抽样

```python
else:  # Record train set summaries, and train
    if i % 100 == 99:  # Record execution stats
        run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
        run_metadata = tf.RunMetadata()
        summary, _ = sess.run([merged, train_step],
                              feed_dict=feed_dict(True),
                              options=run_options,
                              run_metadata=run_metadata)
        train_writer.add_run_metadata(run_metadata, 'step%03d' % i)
        train_writer.add_summary(summary, i)
        print('Adding run metadata for', i)
    else:  # Record a summary
        summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
        train_writer.add_summary(summary, i)
```

MNIST

- 最后要记得关闭文件

```
train_writer.close()
test_writer.close()
```

MNIST

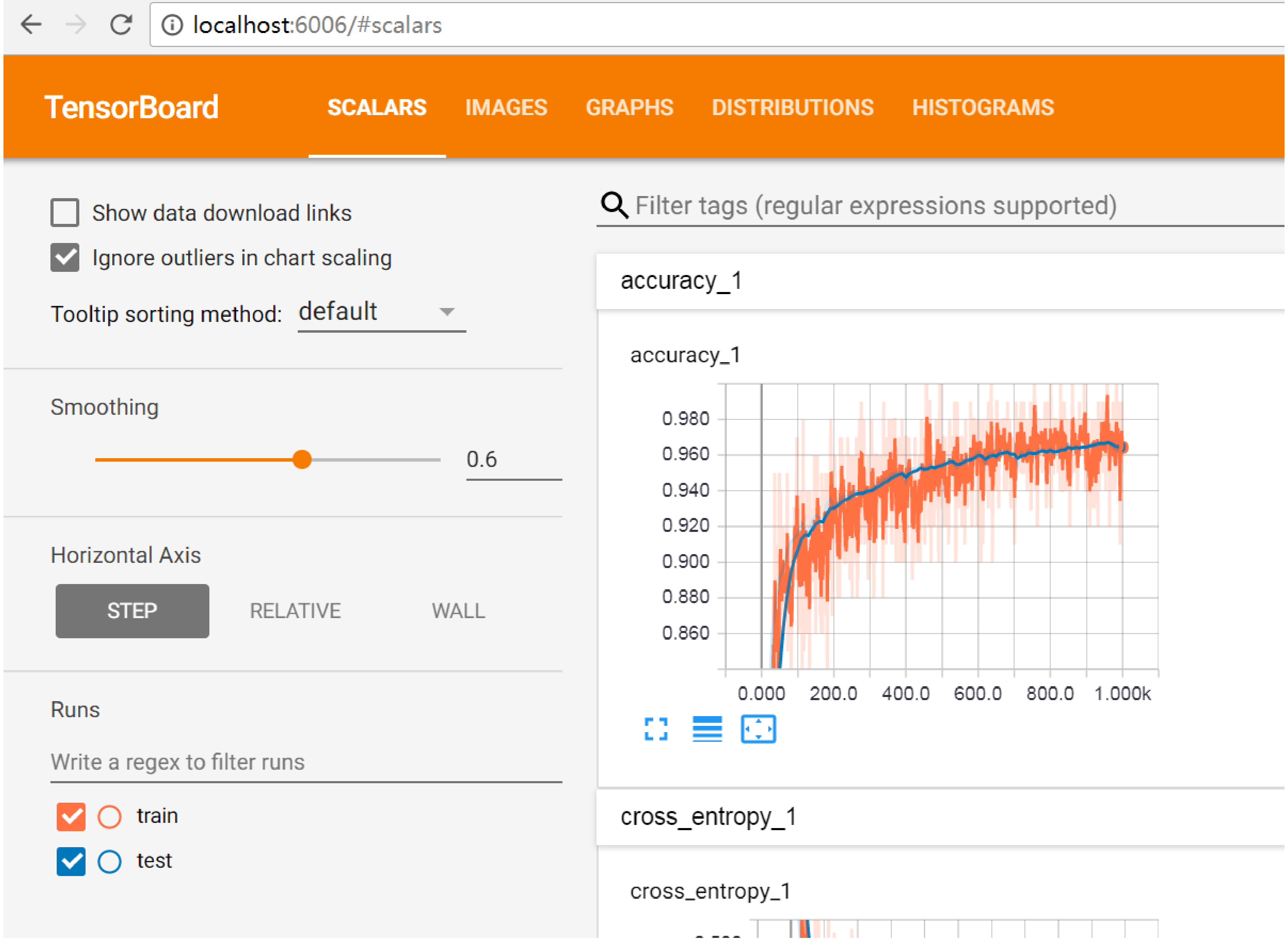
- 程序输出

```
Accuracy at step 0: 0.108
Accuracy at step 10: 0.6937
Accuracy at step 20: 0.8179
Accuracy at step 30: 0.8608
Accuracy at step 40: 0.8795
Accuracy at step 50: 0.8908
Accuracy at step 60: 0.8968
Accuracy at step 70: 0.9047
Accuracy at step 80: 0.9111
Accuracy at step 90: 0.9131
Adding run metadata for 99
Accuracy at step 100: 0.9158
```

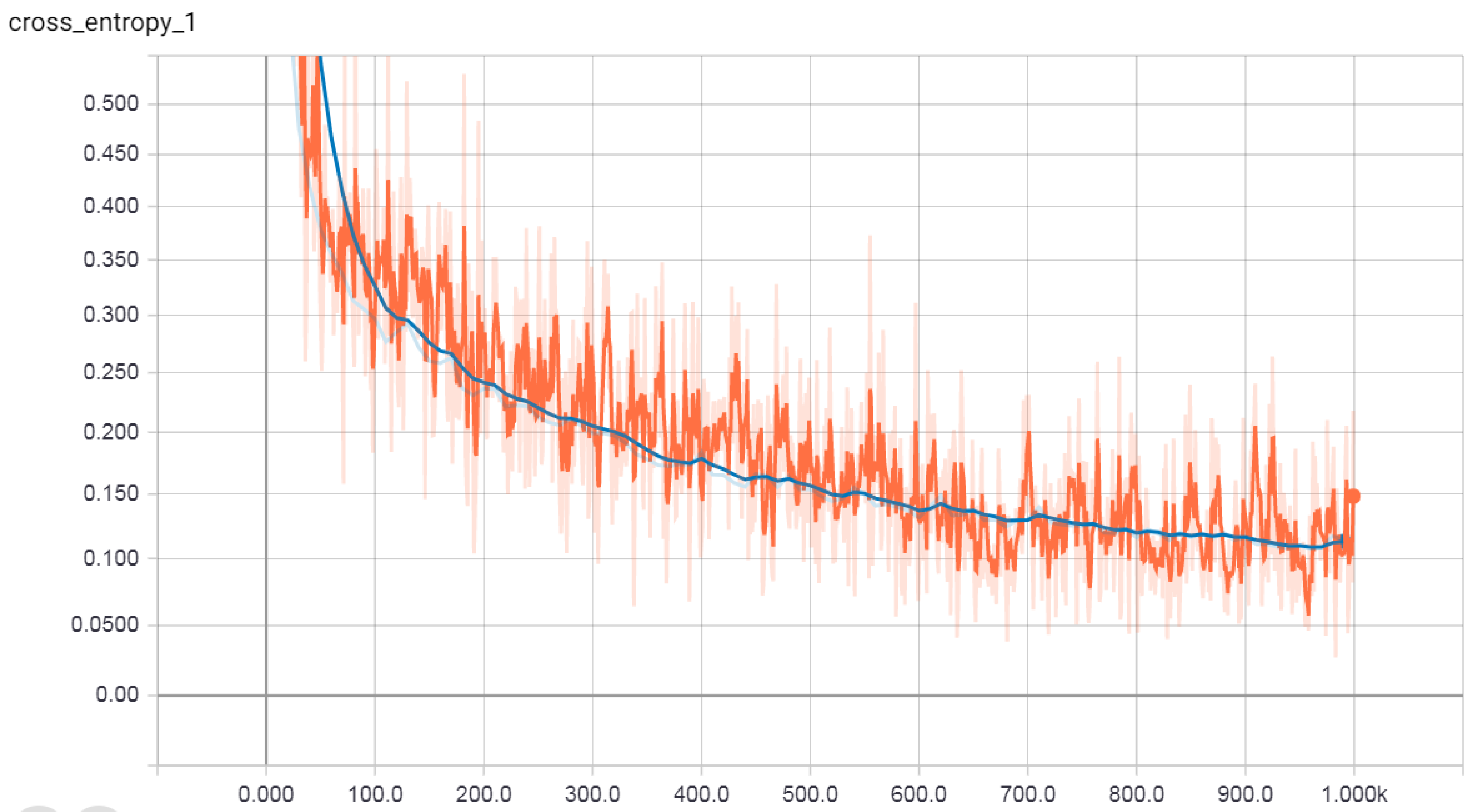
```
Accuracy at step 910: 0.9659
Accuracy at step 920: 0.966
Accuracy at step 930: 0.9671
Accuracy at step 940: 0.967
Accuracy at step 950: 0.9665
Accuracy at step 960: 0.9676
Accuracy at step 970: 0.9655
Accuracy at step 980: 0.9631
Accuracy at step 990: 0.9642
Adding run metadata for 999
```

TensorBoard 使用

- 在命令行中输入 tensorboard --logdir=<your-log-directory>
- 然后在浏览器中输入地址 http://localhost:6006/
- 不必等到程序运行结束再打开 tensorboard
  - 程序运行过程中也可以用
  - tensorboard 每隔几秒钟会刷新一次，读取最新的日志并显示

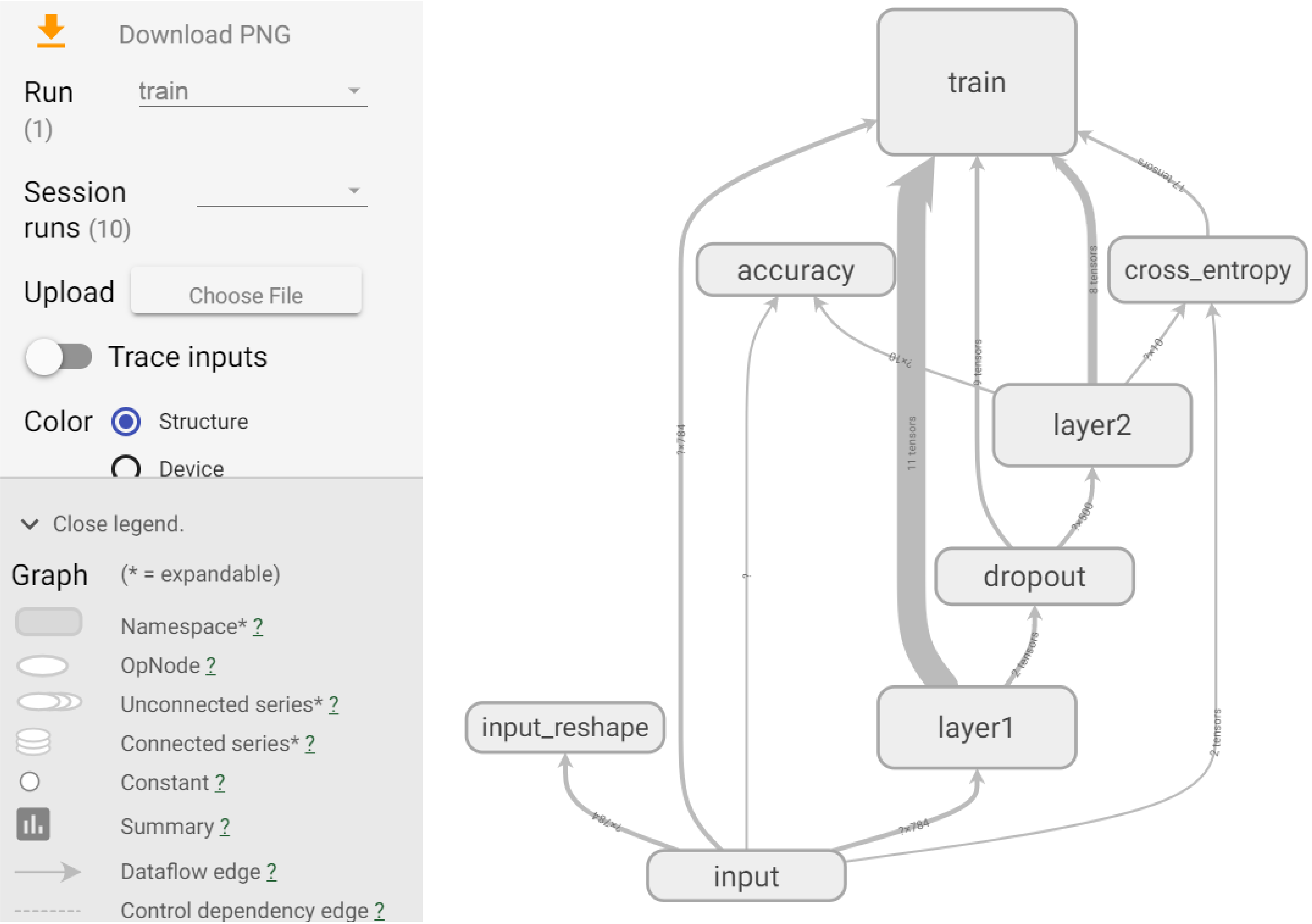


localhost:6006/#scalars
TensorBoard
SCALARS
IMAGES
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method:
default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
train
test
Filter tags (regular expressions supported)
accuracy_1
accuracy_1
0.980
0.960
0.940
0.920
0.900
0.880
0.860
0.000
200.0
400.0
600.0
800.0
1.000k
cross_entropy_1
cross_entropy_1



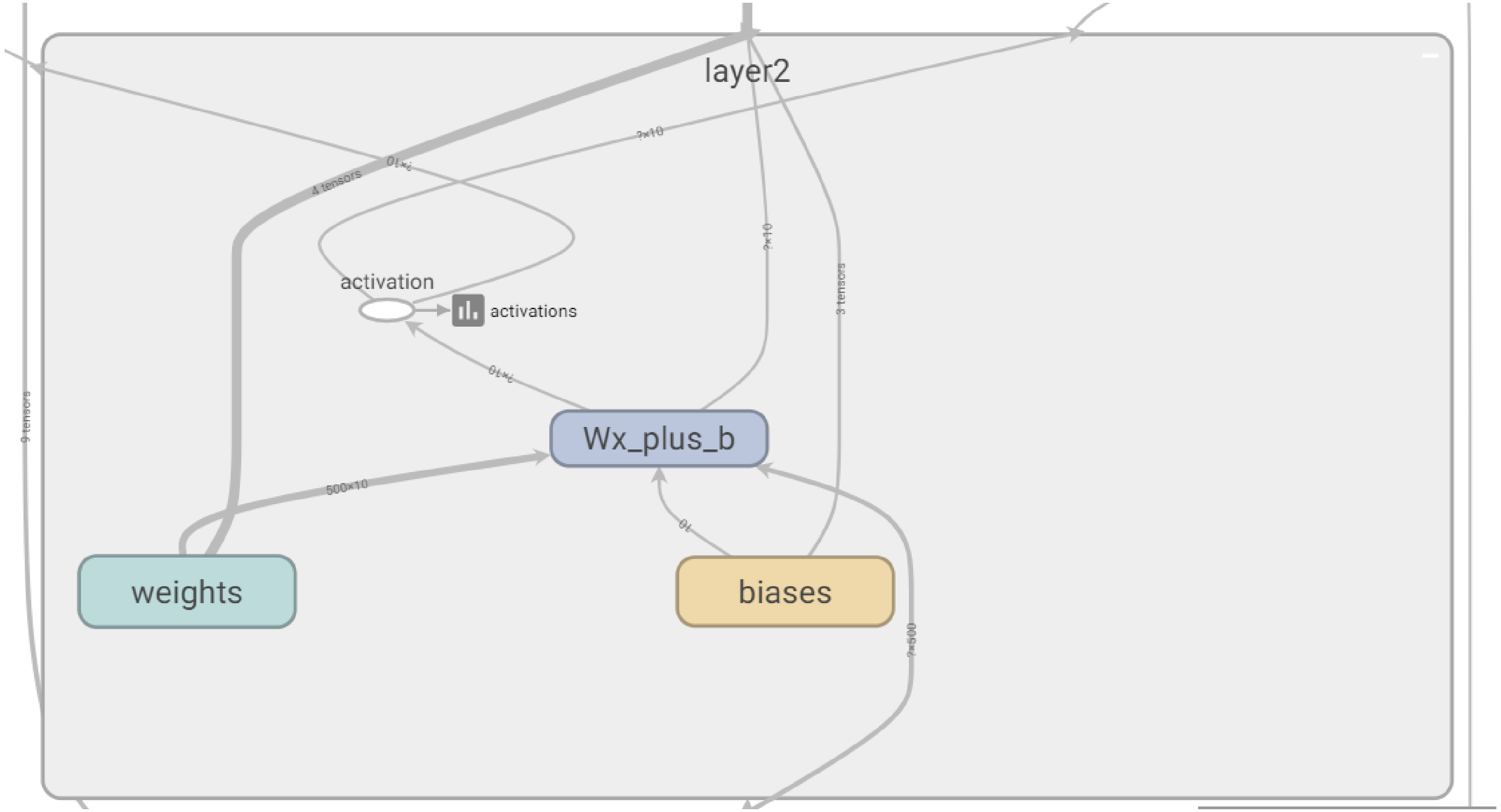
cross_entropy_1
0.500
0.450
0.400
0.350
0.300
0.250
0.200
0.150
0.100
0.0500
0.00
0.000
100.0
200.0
300.0
400.0
500.0
600.0
700.0
800.0
900.0
1.000k

- 计算图



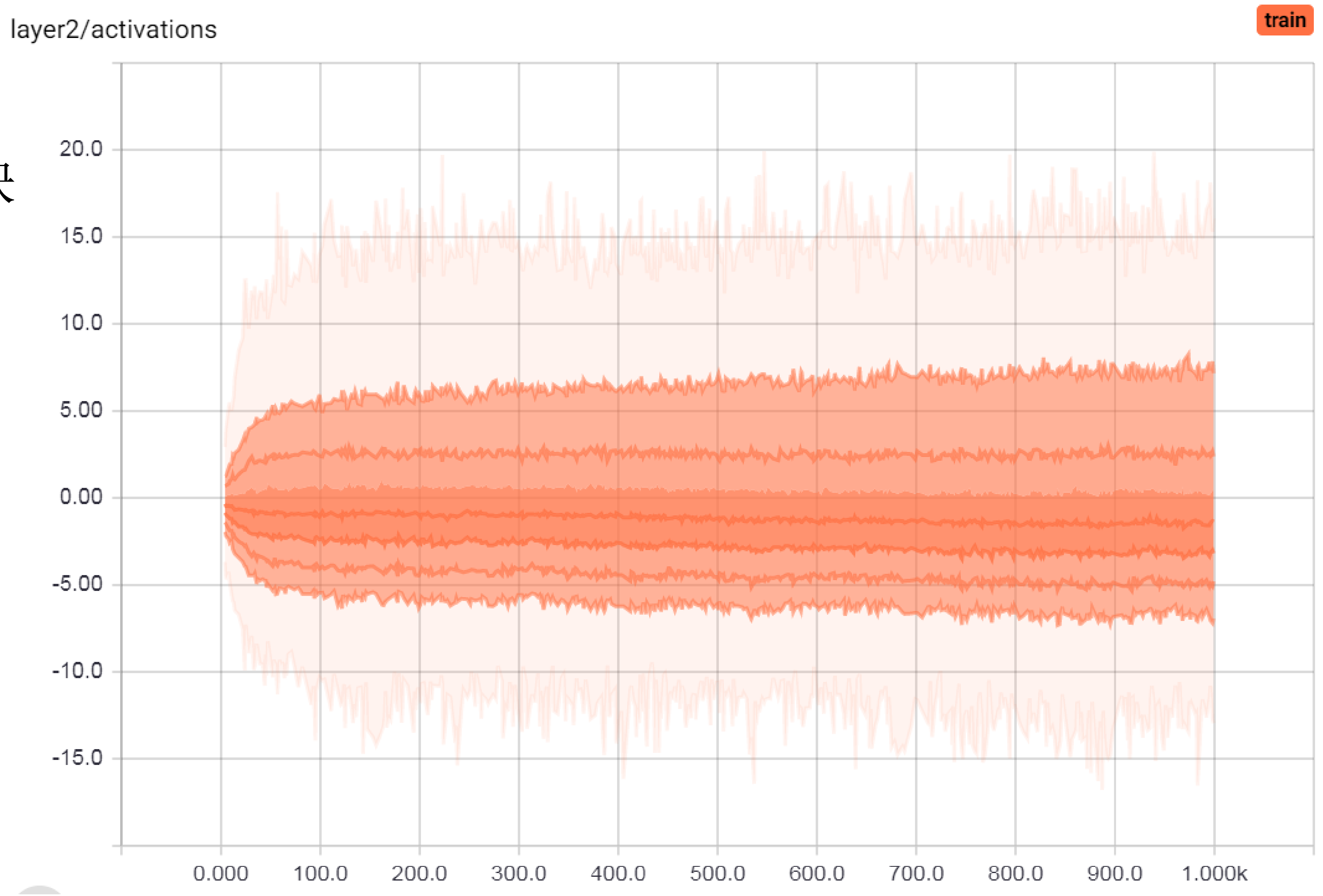
Download PNG
Run
train
(1)
Session runs (10)
Upload
Choose File
Trace inputs
Color
Structure
Device
Close legend.
Graph
(* = expandable)
Namespace* ?
OpNode ?
Unconnected series* ?
Connected series* ?
Constant ?
Summary ?
Dataflow edge ?
Control dependency edge ?
train
accuracy
cross_entropy
layer2
dropout
layer1
input_reshape
input

- 可以双击展开/收缩计算图中的节点
  - 连线越粗参数越多



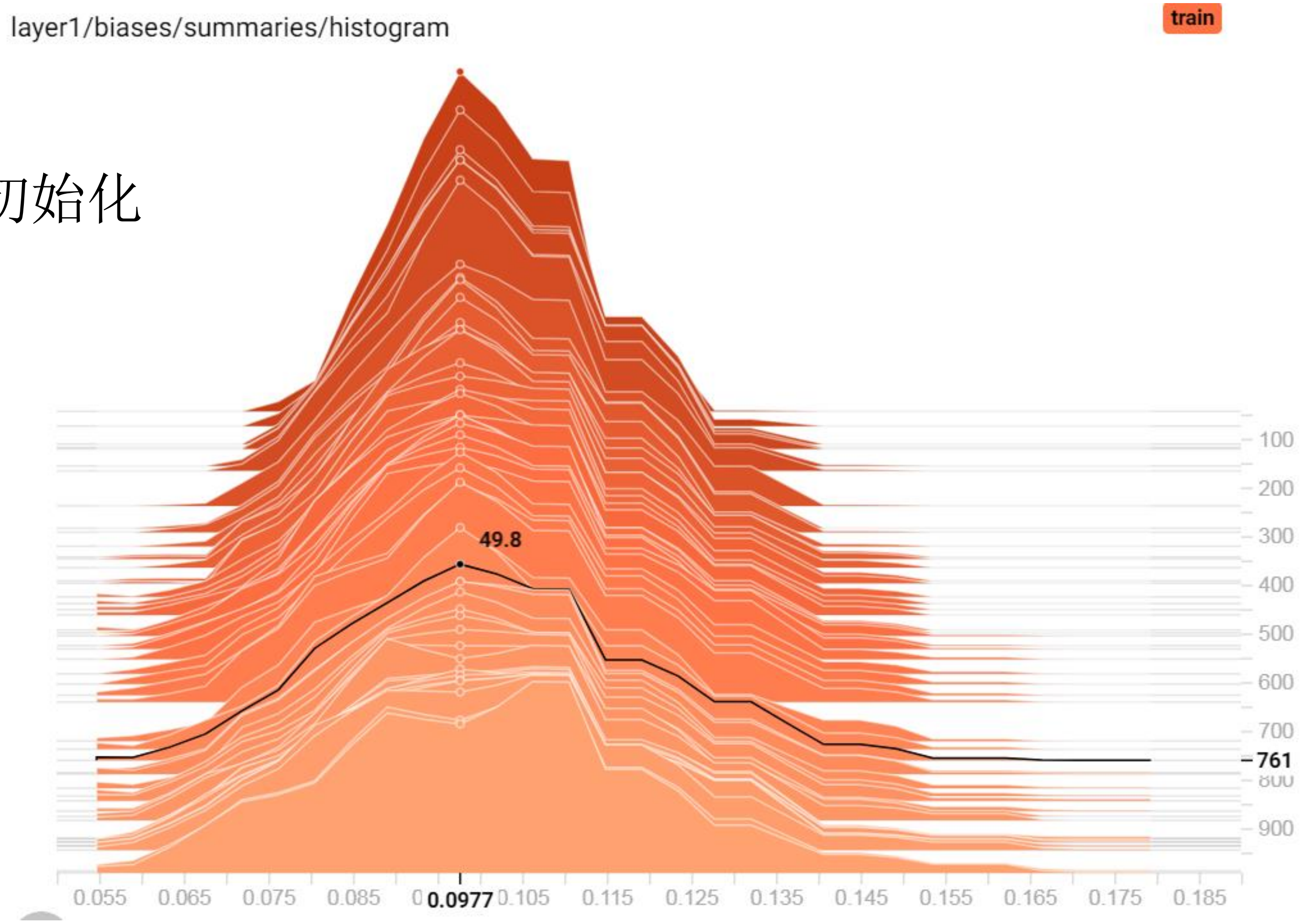
TensorBoard 使用

- 数据分布
  - 前 100 步变化较快
  - 之后趋于平稳



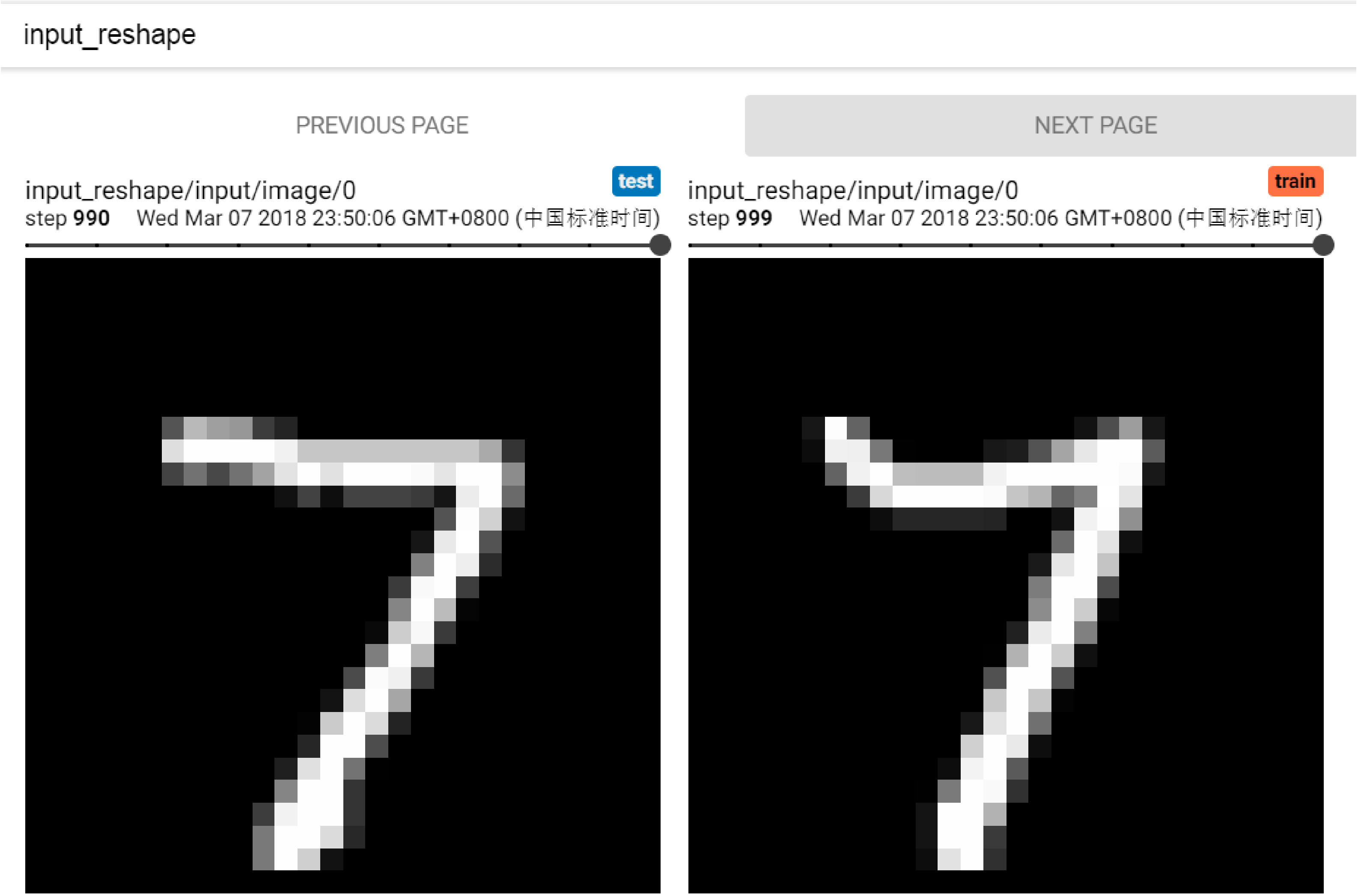
TensorBoard 使用

- 直方图
  - bias 用 0.1 初始化



TensorBoard 使用

- 图像



代码调试

- Jongwook Choi, “A Practical Guide for Debugging TensorFlow Codes”
  - https://github.com/wookayin/tensorflow-talk-debugging
- 基本方法
  - Session.run()
  - Tensorboard
  - tf.Print()
- 高阶方法
  - tfdbg

代码调试

- Session.run()
  - 最基本、最通用的方法
  - 代码写得模块化一点比较好修改
    - 例如把模型定义成一个类，或者相关 Op 都放到一个字典里

代码调试

- Tensorboard
  - 优点
    - 直观
  - 缺点
    - 如果要记录很大的张量的直方图会很慢，影响 GPU 利用率
  - 注意
    - 给变量起名字的时候用点儿心，可以根据一些正则表达式过滤/提取某些变量
    - 好好用 scope，计算图会比较好看

代码调试

tf.Print

- tf.Print()
  - 一个关于 input_ 的恒等 Op
  - 但是会输出 data 中的张量值

```
tf.Print(
    input_,
    data,
    message=None,
    first_n=None,
    summarize=None,
    name=None
)
```

代码调试

- tf.Print()

```
def multilayer_perceptron(x):
    fc1 = layers.fully_connected(x, 256, activation_fn=tf.nn.relu)
    fc2 = layers.fully_connected(fc1, 256, activation_fn=tf.nn.relu)
    out = layers.fully_connected(fc2, 10, activation_fn=None)
    out = tf.Print(out, [tf.argmax(out, 1)],
                   'argmax(out) = ', summarize=20, first_n=7)
    return out
```

For the first seven times (i.e. 7 feed-forwards or SGD steps), it will print the predicted labels for the 20 out of `batch_size` examples

```
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [6 6 6 4 4 6 4 4 6 6 4 0 6 4
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [6 6 0 0 3 6 4 3 6 6 3 4 4 4
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [3 4 0 6 6 6 0 7 3 0 6 7 3 6
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [6 1 0 0 0 3 3 7 0 8 1 2 0 9
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [6 0 0 9 0 4 9 9 0 8 2 7 3 9
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [6 0 1 1 9 0 8 3 0 9 9 0 2 6
I tensorflow/core/kernels/logging_ops.cc:79] argmax(out) = [3 6 9 8 3 9 1 0 1 1 9 3 2 3
[2016-06-03 00:11:08.661563] Epoch 00, Loss = 0.332199
```

代码调试

- tfdbg
  - 官方调试器
    - 类似 gdb 的命令行调试工具
  - 示例代码
    - https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/debug/examples
  - 文档
    - https://www.tensorflow.org/programmers_guide/debugger
  - 示例
    - https://developers.googleblog.com/2017/02/debug-tensorflow-models-with-tfdbg.html
      - 还有一小段视频
    - https://www.cnblogs.com/hellcat/articles/7812119.html
      - 很详细

代码调试

- 用法
  - 只需普通的代码上修改两行

```
from tensorflow.python import debug as tf_debug

sess = tf_debug.LocalCLIDebugWrapperSession(sess)
```

  - 支持两种 UI
    - ”curses”：鼠标点击
    - “readline”：读取命令行
  - Windows 下 “curses” 方式支持不好，建议用 “readline”
  - Mac/Linux 下 “curses” 更方便

代码调试

- 示例代码
  - mnist_debug.py

```
if FLAGS.debug and FLAGS.tensorboard_debug_address:
    raise ValueError(
        "The --debug and --tensorboard_debug_address flags are mutually "
        "exclusive.")
if FLAGS.debug:
    sess = tf_debug.LocalCLIDebugWrapperSession(sess, ui_type=FLAGS.ui_type)
elif FLAGS.tensorboard_debug_address:
    sess = tf_debug.TensorBoardDebugWrapperSession(
        sess, FLAGS.tensorboard_debug_address)
```

代码调试

- 运行 mnist_debug.py
  - 注意设置 --debug=True

```
run-start: run #1: 1 fetch (accuracy/accuracy/Mean:0); 2 feeds

TTTTTT FFFF DDD  BBBB   GGG
  TT   F    D  D B   B G
  TT   FFF  D  D BBBB  G  GG
  TT   F    D  D B   B G   G
  TT   F    DDD  BBBB   GGG

======================================
Session.run() call #1:

Fetch(es):
  accuracy/accuracy/Mean:0

Feed dict:
  input/x-input:0
  input/y-input:0
======================================
```

代码调试

- tfdbg 回显提示信息

```
Select one of the following commands to proceed ---->
  run:
    Execute the run() call with debug tensor-watching
  run -n:
    Execute the run() call without debug tensor-watching
  run -t <T>:
    Execute run() calls (T - 1) times without debugging, then ex
  run -f <filter_name>:
    Keep executing run() calls until a dumped tensor passes a gi
    Registered filter(s):
        * has_inf_or_nan
  invoke_stepper:
    Use the node-stepper interface, which allows you to interact

For more details, see help..
```

代码调试

- 输入 run 命令
- 执行一步 sess.run

```
tfdbg> run
run-end: run #1: 1 fetch (accuracy/accuracy/Mean:0); 2 feeds
22 dumped tensor(s):

t (ms)      Size (B) Op type    Tensor name
[0.000]     252      Const      accuracy/correct_prediction/ArgMax/dimension:0
[0.000]     2.16k    VariableV2 hidden/biases/Variable:0
[0.000]     1.50M    VariableV2 hidden/weights/Variable:0
[0.000]     248      VariableV2 output/biases/Variable:0
[0.000]     19.75k   VariableV2 output/weights/Variable:0
[62.400]    258      Identity   output/biases/Variable/read:0
[124.800]   19.76k   Identity   output/weights/Variable/read:0
[173.600]   214      Const      accuracy/accuracy/Const:0
[173.600]   78.36k   ArgMax     accuracy/correct_prediction/ArgMax_1:0
[204.800]   2.17k    Identity   hidden/biases/Variable/read:0
[267.200]   1.50M    Identity   hidden/weights/Variable/read:0
[519.801]   19.07M   MatMul     hidden/Wx_plus_b/MatMul:0
[776.402]   19.07M   Add        hidden/Wx_plus_b/add:0
[1108.002]  19.07M   Relu       hidden/Relu:0
```

代码调试

- 输入 pt <tensor_name>
  - pt = print tensor
- 打印出张量的值
- 若 ui_type="curses"
  - 可以直接点击张量名称

```
tfdbg> pt Softmax:0
Tensor "Softmax:0:DebugIdentity":
  dtype: float32
  shape: (10000, 10)

array([[ 0.08080047,  0.04707913,  0.14692004, ...,  0.33255816,
         0.0347424 ,  0.05852808],
       [ 0.01035291,  0.10903349,  0.03189213, ...,  0.44066665,
         0.09266753,  0.11560722],
       [ 0.02030662,  0.02630959,  0.02971563, ...,  0.38162729,
         0.14402775,  0.06281166],
       ...,
       [ 0.05561169,  0.01549218,  0.01102988, ...,  0.25585306,
         0.21737385,  0.02964481],
       [ 0.0826645 ,  0.03360071,  0.01598819, ...,  0.22908622,
         0.12518428,  0.1405647 ],
```

代码调试

- 常用命令

- `list_tensors` (`lt`) : Show the list of dumped tensor(s).
- `print_tensor` (`pt`) : Print the value of a dumped tensor.
- `node_info` (`ni`) : Show information about a node
  - `ni -t` : Shows the traceback of tensor creation
- `list_inputs` (`li`) : Show inputs to a node
- `list_outputs` (`lo`) : Show outputs to a node
- `run_info` (`ri`) : Show the information of current run (e.g. what to fetch, what feed_dict is)
- `invoke_stepper` (`s`) : Invoke the stepper!
- `run` (`r`) : Move to the next run

- https://www.tensorflow.org/programmers_guide/debugger#tfdbg_cli_frequently-used_commands

参数保存与恢复

- 保存
  - 定义 Saver 类
    - var_list：待保存变量组成的列表或字典
      - 默认保存所有变量
    - max_to_keep：保留的检查点的数目
      - 默认只保留最近 5 个检查点

```
__init__(
    var_list=None,
    reshape=False,
    sharded=False,
    max_to_keep=5,
    keep_checkpoint_every_n_hours=10000.0,
    name=None,
    restore_sequentially=False,
    saver_def=None,
    builder=None,
    defer_build=False,
    allow_empty=False,
    write_version=tf.train.SaverDef.V2,
    pad_step_number=False,
    save_relative_paths=False,
    filename=None
)
```

参数保存与恢复

- 保存

```
v1 = tf.Variable(..., name='v1')
v2 = tf.Variable(..., name='v2')

# Pass the variables as a dict:
saver = tf.train.Saver({'v1': v1, 'v2': v2})

# Or pass them as a list.
saver = tf.train.Saver([v1, v2])
# Passing a list is equivalent to passing a dict with the variable op names
# as keys:
saver = tf.train.Saver({v.op.name: v for v in [v1, v2]})
```

参数保存与恢复

- 保存
  - 调用 save() 方法保存模型到文件
    - save_path：保存路径（包括模型名称）
    - global_step：迭代步数
  - 返回值是新创建的检查点名称（字符串）

save

```
save(
    sess,
    save_path,
    global_step=None,
    latest_filename=None,
    meta_graph_suffix='meta',
    write_meta_graph=True,
    write_state=True,
    strip_default_attrs=False
)
```

参数保存与恢复

- 恢复
  - restore() 方法
  - 默认恢复最后一个检查点

```
restore(
    sess,
    save_path
)
```

参数保存与恢复

- 检查点命名
  - 文件名-步数

att-seq2seq-4300.data-00000-of-00001	2018/3/9 1:30
att-seq2seq-4300.index	2018/3/9 1:30
att-seq2seq-4300.meta	2018/3/9 1:30
att-seq2seq-4400.data-00000-of-00001	2018/3/9 1:31
att-seq2seq-4400.index	2018/3/9 1:31
att-seq2seq-4400.meta	2018/3/9 1:31
att-seq2seq-4500.data-00000-of-00001	2018/3/9 1:31
att-seq2seq-4500.index	2018/3/9 1:31
att-seq2seq-4500.meta	2018/3/9 1:31
att-seq2seq-4600.data-00000-of-00001	2018/3/9 1:31
att-seq2seq-4600.index	2018/3/9 1:31
att-seq2seq-4600.meta	2018/3/9 1:31
att-seq2seq-4700.data-00000-of-00001	2018/3/9 1:31
att-seq2seq-4700.index	2018/3/9 1:31
att-seq2seq-4700.meta	2018/3/9 1:31

参数保存与恢复

- 检查点格式
  - checkpoint 文件
    - 相当于整个目录的 readme.txt
    - 包含最近一个检查点的名称
    - 当前目录下所有的检查点

```
model_checkpoint_path: "att-seq2seq-5000"
all_model_checkpoint_paths: "att-seq2seq-3000"
all_model_checkpoint_paths: "att-seq2seq-3500"
all_model_checkpoint_paths: "att-seq2seq-4000"
all_model_checkpoint_paths: "att-seq2seq-4500"
all_model_checkpoint_paths: "att-seq2seq-5000"
```

  - .meta 文件保存了图结构
  - .index 文件保存了参数名
  - .data 文件保存了参数值

参数保存与恢复

- 示例
  - att_seq2seq_delete_and_copy.py
  - 创建目录

```python
save_path = "../attention-seq2seq/"
if not os.path.exists(save_path):
    os.mkdir(save_path)

picture_path = os.path.join(save_path, "pics")
if not os.path.exists(picture_path):
    os.mkdir(picture_path)

model_path = os.path.join(save_path, "model")
if not os.path.exists(model_path):
    os.mkdir(model_path)

label_file_name = "labels.tsv"
with open(os.path.join(model_path, label_file_name), "w") as f:
    f.write("Number\tIsOdd\n")
    for i in range(vocab_size):
        f.write(str(i) + "\t" + str(i%2) + "\n")
```

参数保存与恢复

- 示例
  - 定义 Saver，每 100 步保存一下所有变量

```
max_batches = 5001
save_period = 100

saver = tf.train.Saver()
model_name = os.path.join(model_path, "att-seq2seq")
```

```
if batch_id % save_period == 0:
    saver.save(sess, save_path=model_name, global_step=batch_id)
```

参数保存与恢复

- 示例
  - 打开 session 时先检查一下有没有保存过的检查点
    - 如果有，用最新的检查点的参数值来初始化模型，并重新计算 start_step

```
name = tf.train.latest_checkpoint(model_path)
start_step = 0
if name is not None:
    print("Restore from file " + name)
    saver.restore(sess, save_path=name)
    start_step = int(name.split("-")[-1]) + 1
else:
    print("No previous checkpoints!")

for batch_id in range(start_step, max_batches):
```

参数保存与恢复

- 以上方法只恢复变量值，不恢复计算图
  - 因为代码中已经构建了计算图
- 有时候没有构建计算图的代码
  - 例如发布模型给别人使用，但是没有开源代码
- 此时需要先调用 tf.train.import_meta_graph 加载图结构

参数保存与恢复

- 在单独的文件中加载模型（不构建计算图）
  - 参考 http://blog.csdn.net/qq_34197612/article/details/79249985

```python
import tensorflow as tf
import numpy as np
with tf.Session() as session:
    saver = tf.train.import_meta_graph('E:/data/model/model_test.meta')
    saver.restore(session, tf.train.latest_checkpoint('E:/data/model/'))
    #print(session.run("w:0"))
    graph = tf.get_default_graph()
    #w = graph.get_tensor_by_name('w:0')
    #根据名称加载placeholder
    x = graph.get_tensor_by_name('x:0')
    param = np.array([[1], [2], [3]])
    #根据名称获取保存的损失函数引用
    cost = graph.get_collection("cost")
    print(session.run(cost,feed_dict={x : param}))
```

BLEU

- 一种机器翻译的自动评价指标
  - 最早出现，使用最广泛
- *Kishore Papineni et al.* ***BLEU: a Method for Automatic Evaluation of Machine Translation****. ACL' 02*
- 大意
  - 计算候选翻译 (candidate) 里的一元词组到四元词组在参考翻译(references)中出现的频率
  - 几何平均值就代表了翻译质量

BLEU

- p_n: 修正的 n-gram 准确率

$$p_n = \frac{\sum_{\mathcal{C} \in \{Candidates\}} \sum_{n\text{-}gram \in \mathcal{C}} Count_{clip}(n\text{-}gram)}{\sum_{\mathcal{C}' \in \{Candidates\}} \sum_{n\text{-}gram' \in \mathcal{C}'} Count(n\text{-}gram')}$$

- Brevity Penalty (BP)
  - 否则模型将偏向于较短但是更正确的翻译
  - （其实加了 BP 还是偏向于短的）

$$\mathrm{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

- BLEU
  - $N = 4, w_n = 1/N$ → 几何平均
  - 理论上可以加权，实际上都取相同权重

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

BLEU

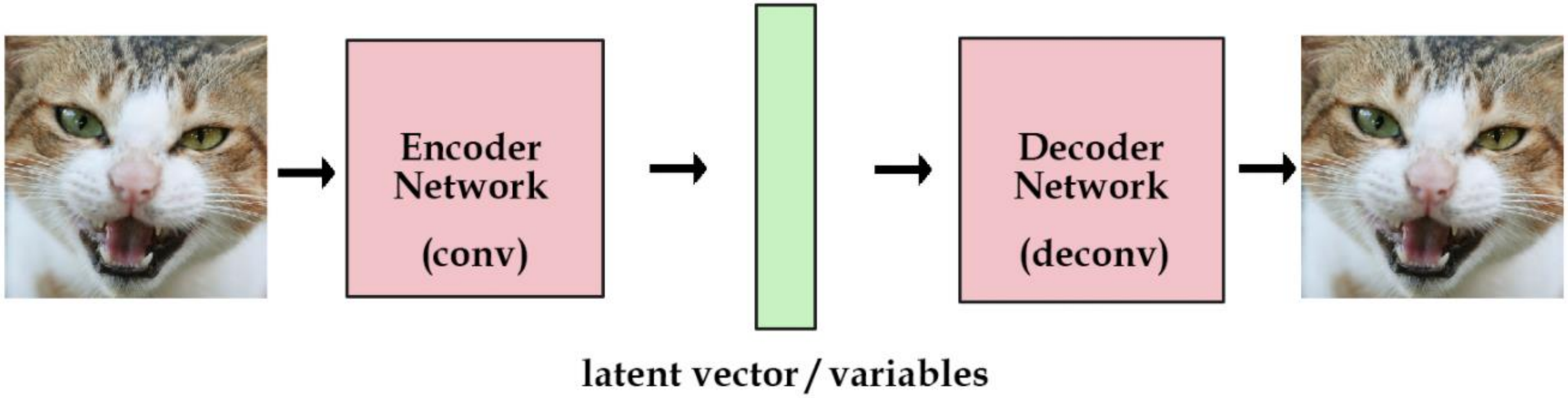
- p_n 的计算
  - Candidate
    - {“the the the cat”}
  - References
    - {“The cat is on the mat”, “There is a cat on the mat”}
- 一元到四元准确率:
  - 直接计算：4/4, 1/3, 0/2, and 0/1
  - 修正后：(min(3, max(1, 2)) + 1)/4 = 3/4, 1/3, 0/2, and 0/1

BLEU

- 范围
  - [0, 1]
  - 越大越好
  - 目前常用的机器翻译系统一般在 0.25~0.4 之间
  - 与数据集有关
  - 其他替代指标
    - Meteor, ROUGH, chrF3

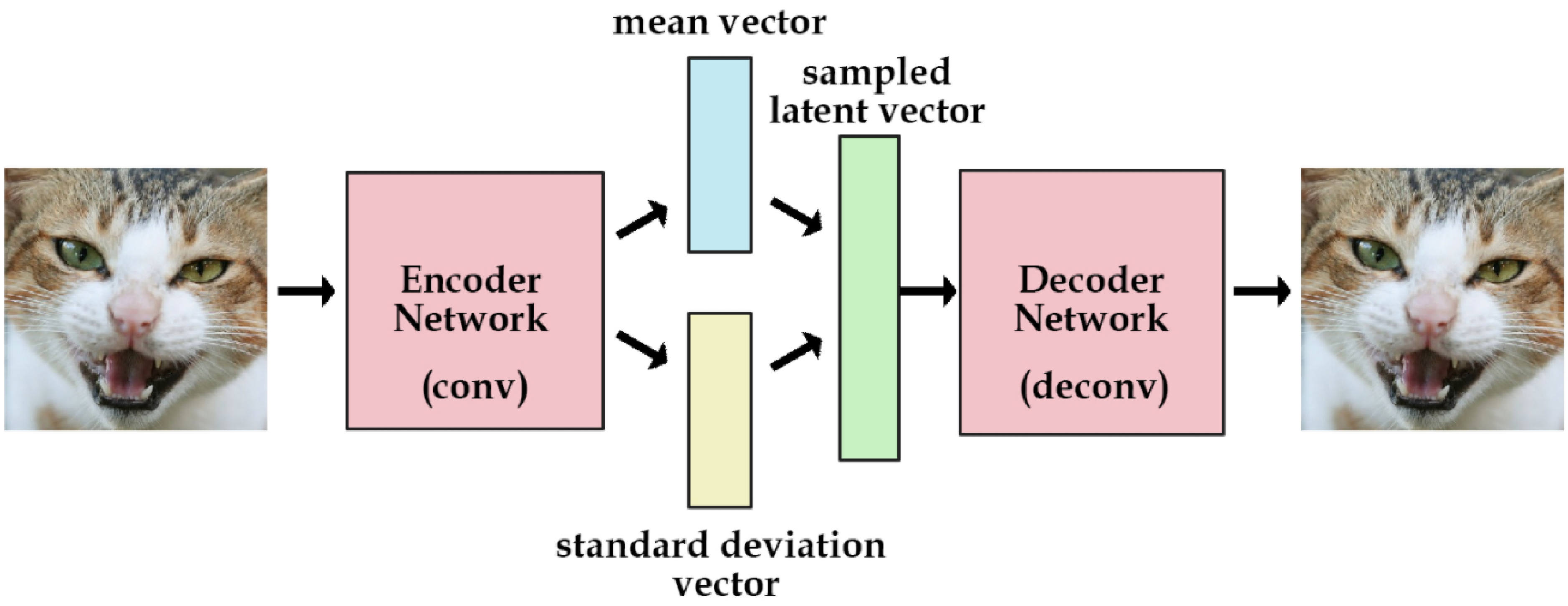
VAE (Variational Auto-Encoder)

- 一种生成模型
  - 在 auto-encoder 的基础上，把隐空间的点变成一个分布
- 简单解释
  - http://kvfrans.com/variational-autoencoders-explained/
- Auto-Encoder



VAE (Variational Auto-Encoder)

- VAE



VAE (Variational Auto-Encoder)

- 效果
  - AE 只是记住了训练样本，而 VAE 可以泛化，生成新样本
    - 其他理解
      - 参考 https://zhuanlan.zhihu.com/p/25939348 及 http://note.youdao.com/noteshare?id=e77bd545c249626e9d37cb935d967a87&sub=211034F5173C4362A8B25D922E211661

AE-random sample

AE-reconstruction

VAE

VAE (Variational Auto-Encoder)

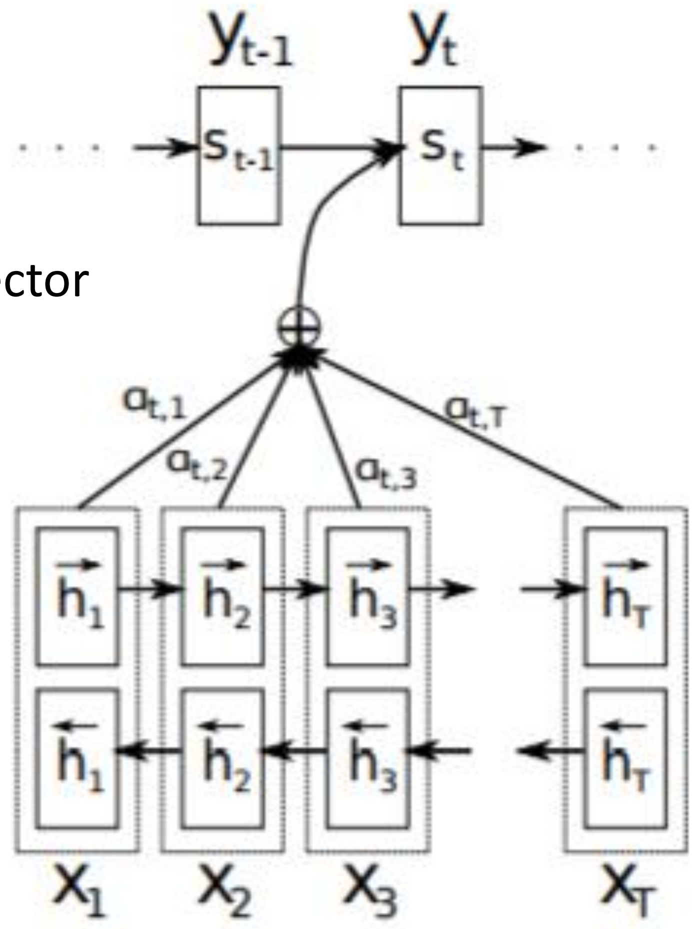
- 相关文献
  - 原论文（比较难读）
    - Diederik P Kingma & Max Welling, “Auto-Encoding Variational Bayes”
    - https://arxiv.org/abs/1312.6114
  - Carl Doersch, “Tutorial on Variational Autoencoders”
    - 无需变分贝叶斯基础，较为通俗易懂
    - 一开始是在 CMU/UCB 的讨论班上讲 VAE 的论文，后来整理成了这份讲义
    - https://arxiv.org/abs/1606.05908

NMT (Neural Machine Translation)

- RNNSearch
  - 第一篇提出给机器翻译加入 Attention 的模型
  - Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate”, ICLR 2015
  - https://arxiv.org/abs/1409.0473

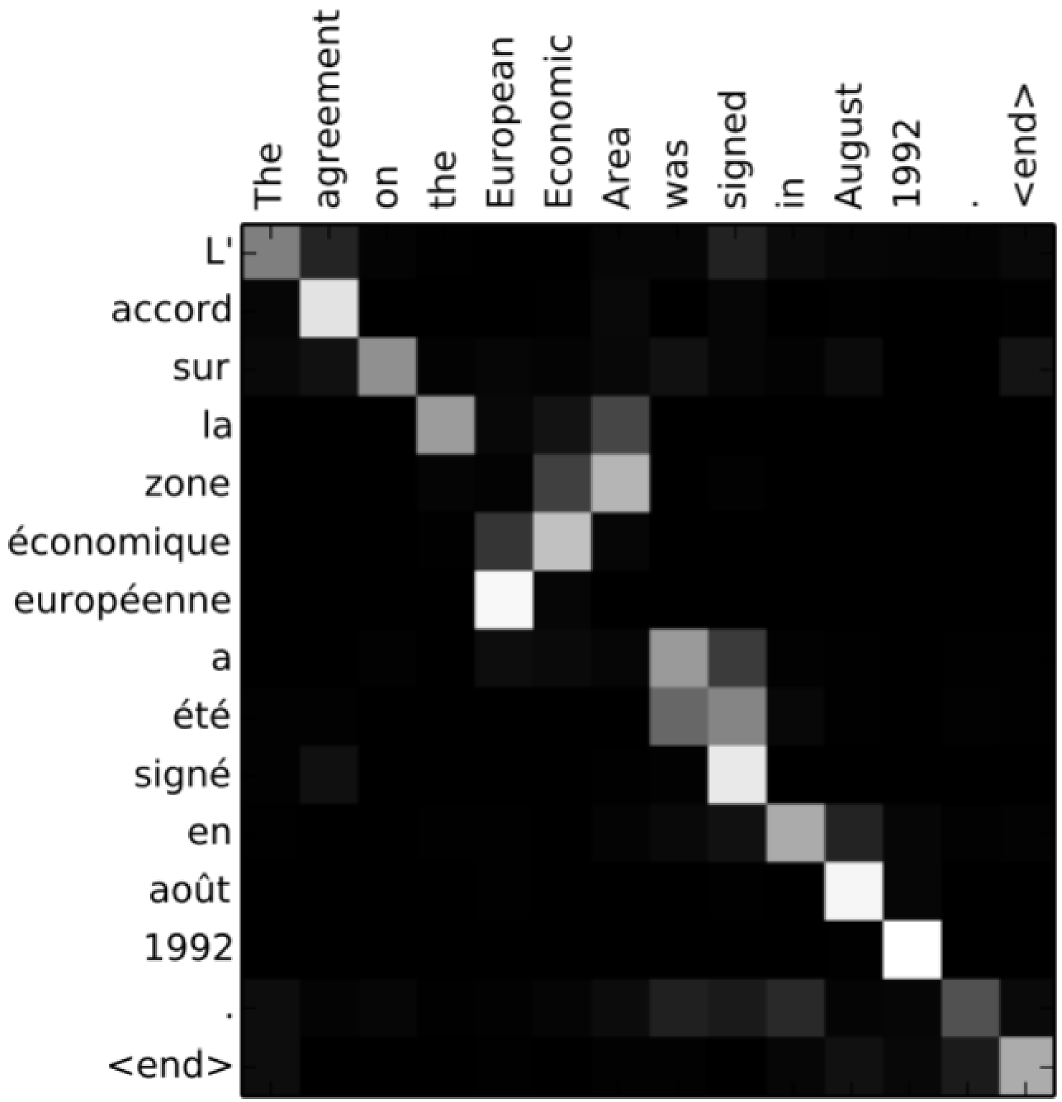
NMT (Neural Machine Translation)

- RNNSearch
  - 在解码的每一步，多加一个额外的输入
    - encoder 每一步的隐状态的加权和，称为 context vector



NMT (Neural Machine Translation)

- RNNSearch
  - 效果
    - 可被视为对齐
    - 可能包含比对其更多的信息
      - 如时间状语对动词过去式



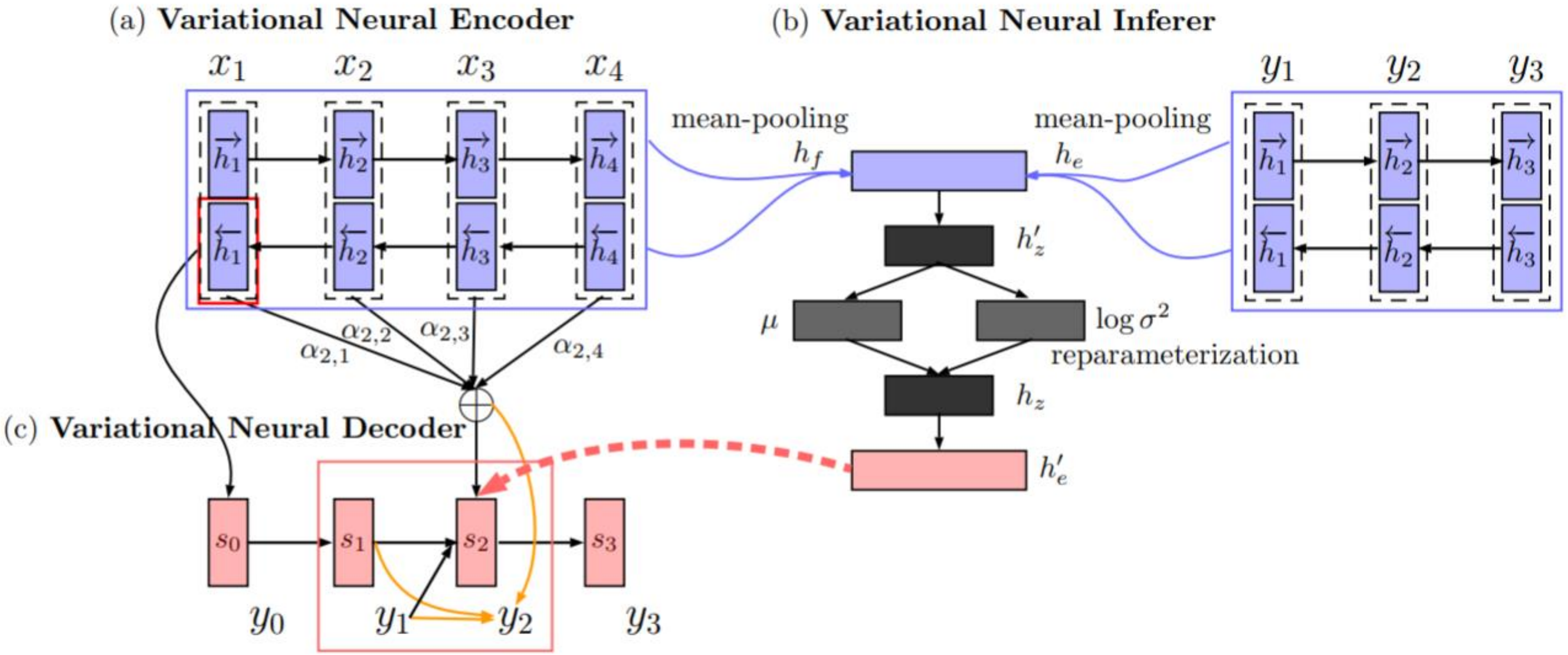
NMT (Neural Machine Translation)

- 代码开源，但是基于 Theano
  - Theano 已停止维护
- XMUNMT
  - 厦门大学用 TF 复现的 RNNSearch
  - https://github.com/XMUNLP/XMUNMT

VNMT (Variational Neural Machine Translation)

- 传统NMT 的问题
  - 流畅度较好，但有时不重视原文（如漏翻、改换人名等）
- Variational Neural Machine Translation
  - https://arxiv.org/abs/1605.07869
  - VNMT = VAE + NMT
  - 把扰动后的隐空间向量也作为 decoder 的输入，进行全局监督、防止漏翻

VNMT (Variational Neural Machine Translation)



VNMT (Variational Neural Machine Translation)

- 代码实现
  - https://github.com/DeepLearnXMU/VNMT
  - 基于 GroundHog（一个基于 Theano 的 RNN 库）
- 我的复现
  - 改编 XMUNMT
    - https://github.com/soloice/RNNsearch
  - 主要修改部分
    - https://github.com/soloice/RNNsearch/blob/master/xmunmt/models/vnmt.py
  - 源码阅读笔记 http://note.youdao.com/noteshare?id=1349a9635b81d841ba6e5f49f8fcdd87&sub=WEBbca6cbefced6458b5ccdff7d03c7c195

VNMT (Variational Neural Machine Translation)

- 复现效果
  - 从头训练不如朴素的 RNNSearch
  - 跟作者交流后
    - 作者表示看我的代码没发现问题
    - 可能的原因是从头训练确实比较难
    - 他们论文中的做法是先用 RNNSearch 预训练，然后用 VNMT 精调

VNMT (Variational Neural Machine Translation)

- 课后任务
  - 阅读 VNMT 源码，熟悉 VAE 和 RNNSearch 模型
    - 这份源码里对 tf.while_loop 和 TensorArray 的运用很纯熟，值得学习
  - 尝试从预训练的 RNNSearch 模型开始精调，观察效果
  - 尝试实现和训练其他模型
    - 例如这群作者又搞了一篇续作 Variational Recurrent Neural Machine Translation
      - VRNMT = VRNN (Variational RNN) + NMT
    - 在解码的每一步都引入一个噪声，而不是只在全局的隐空间引入一次噪声
    - https://arxiv.org/abs/1801.05119