Attention Mechanisms with Tensorflow

Keon Kim
DeepCoding
2016. 08. 26

Today, We Will Study...

1. **Attention Mechanism and Its Implementation**
   - Attention Mechanism (Short) Review
   - Attention Mechanism Code Review

Today, We Will Study...

1. **Attention Mechanism and Its Implementation**
   - Attention Mechanism (Short) Review
   - Attention Mechanism Code Review

2. **Attention Mechanism Variant (Pointer Networks) and Its Implementation**
   - Pointer Networks (Short) Review
   - Pointer Networks Code Review

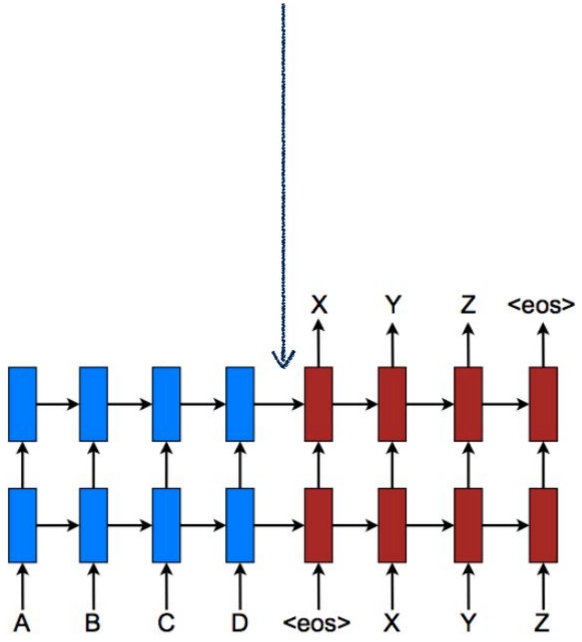
Attention Mechanism Review

Global Attention

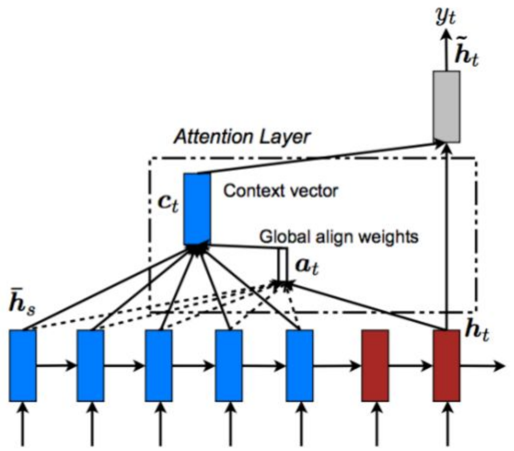
Attention Mechanism Review

Figures from [Luong+2015] for comparison

Encoder compresses input series into one vector
Decoder uses this vector to generate output



simple enc-dec

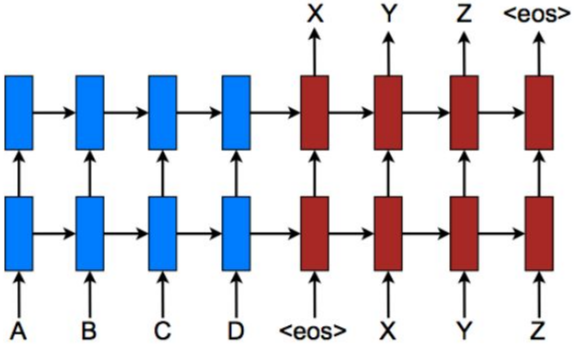


enc-dec + attention

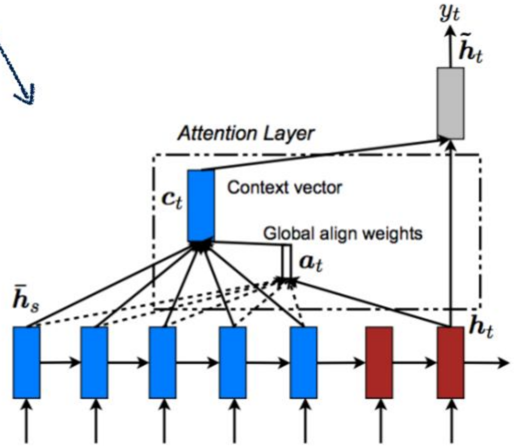
Figures from [Luong+2015] for comparison

Encoder compresses input series into one vector
Decoder uses this vector to generate output

Attention Mechanism predicts the output y_t with a weighted average context vector c_t, not just the last state.



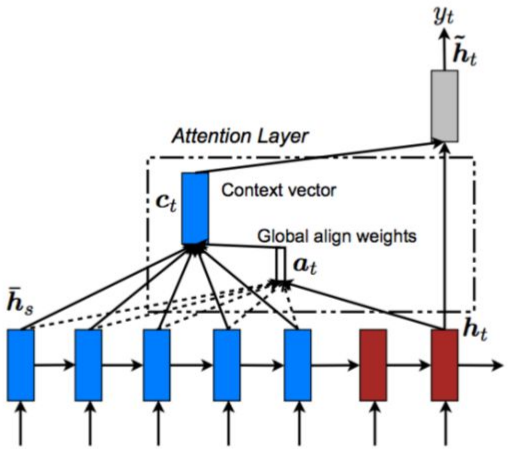
enc-dec



attention

Figures from [Luong+2015] for comparison

Attention Mechanism predicts the output y_t with a weighted average context vector c_t, not just the last state.



$$\boldsymbol{a}_t(s) = \frac{\exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

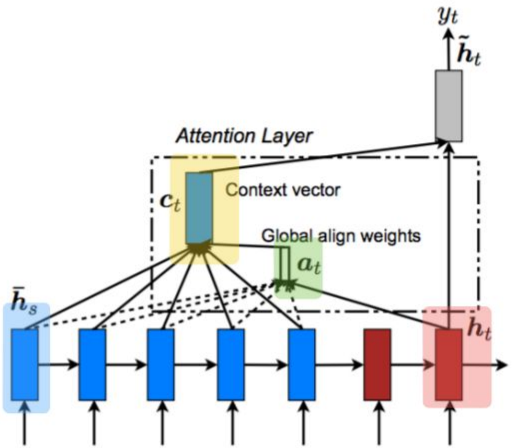
$$c_t = \sum_s a_t(s)\bar{h}_s$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W_c}[\boldsymbol{c}_t; \boldsymbol{h}_t])$$

$$p(y_t | y_{<t}, x) = \mathrm{softmax}(\boldsymbol{W_s}\tilde{\boldsymbol{h}}_t)$$

Figures from [Luong+2015] for comparison

Attention Mechanism predicts the output y_t with a weighted average context vector c_t, not just the last state.



$$\boldsymbol{a}_t(s) = \frac{\exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

$$c_t = \sum_s a_t(s) \bar{h}_s$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{c}_t; \boldsymbol{h}_t])$$

$$p(y_t|y_{<t}, x) = \mathrm{softmax}(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t)$$

Attention Mechanism

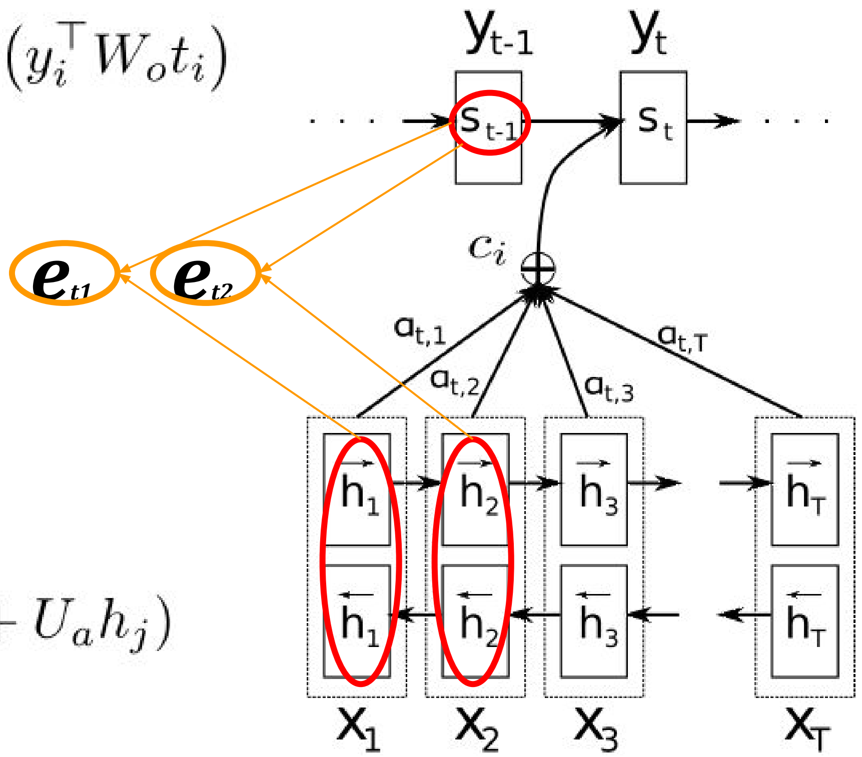
Softmax $$p(y_i|s_i, y_{i-1}, c_i) \propto \exp\left(y_i^\top W_o t_i\right)$$

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

Weight of h

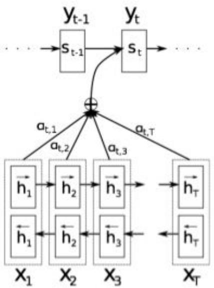
$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

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

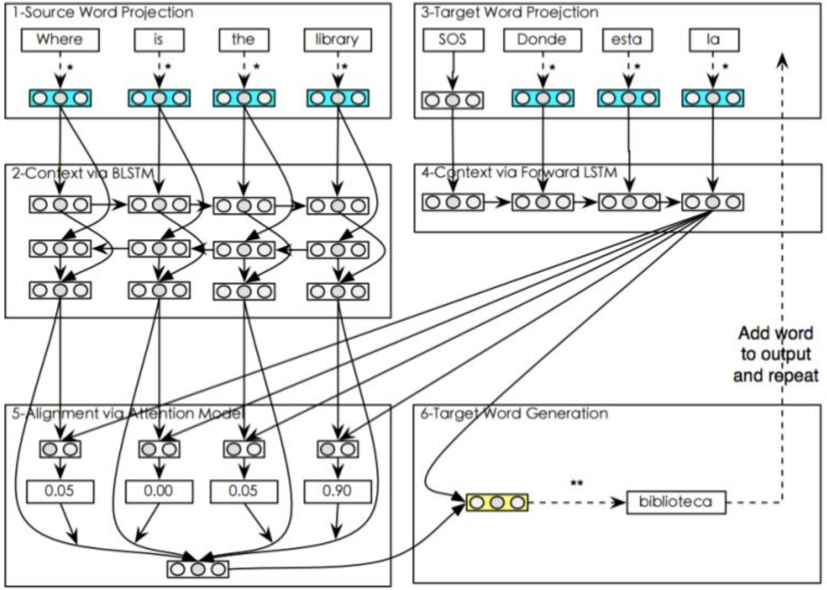


http://arxiv.org/pdf/1511.04586v1.pdf

Character-based Neural Machine Translation [Ling+2015]



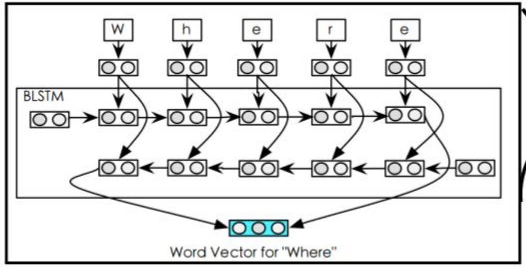
[Bahdanau+2014]



attention

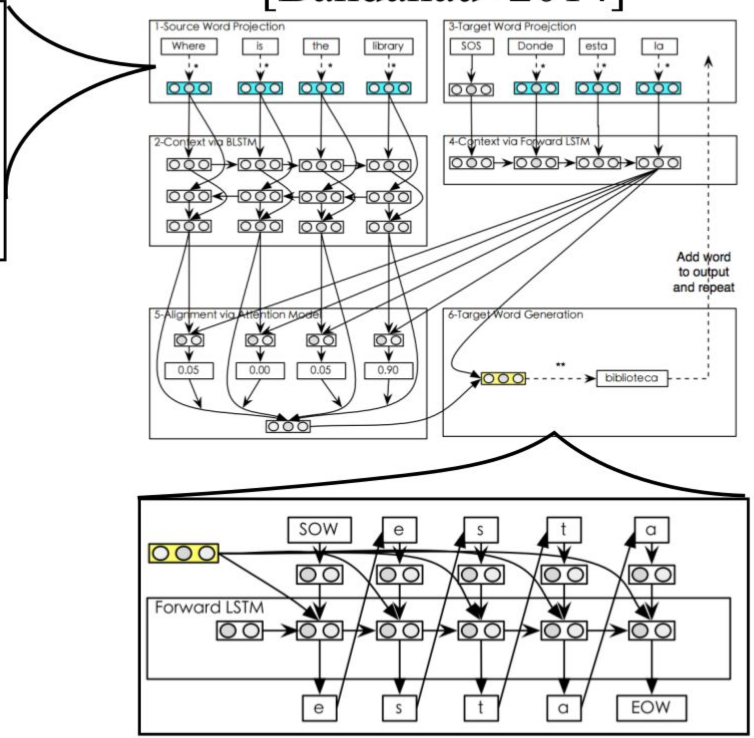
http://arxiv.org/pdf/1511.04586v1.pdf

Character-based Neural Machine Translation [Ling+2015]



String to Vector

[Bahdanau+2014]



String Generation

Attention Mechanism Code Review

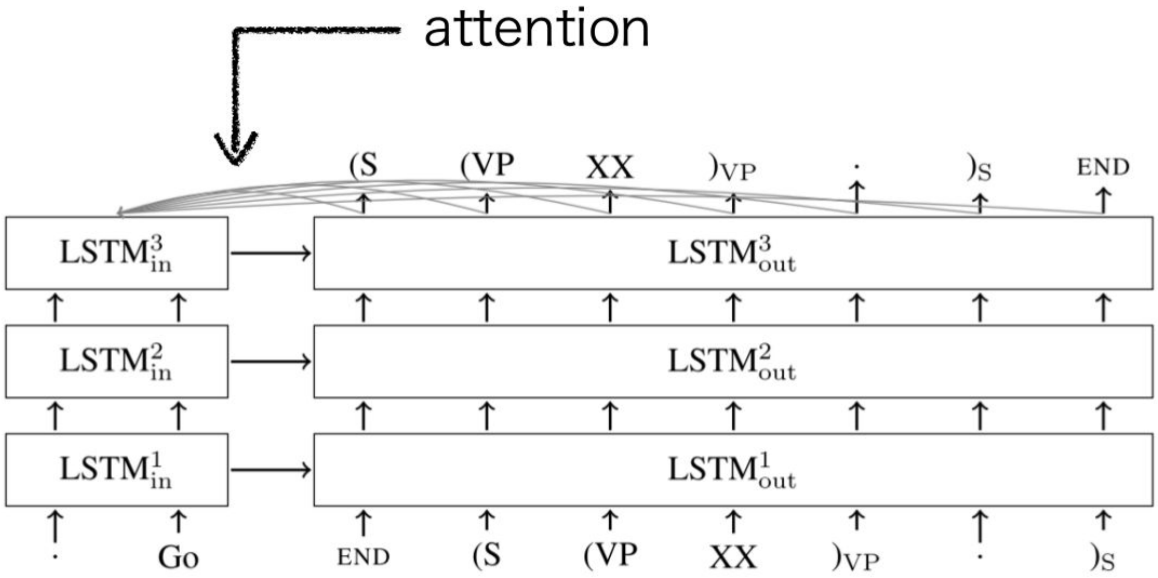
Attention Mechanism

Code Review

translate.py
example in tensorflow
https://github.com/tensorflow/tensorflow/tree/master/tensorflow/models/rnn/translate

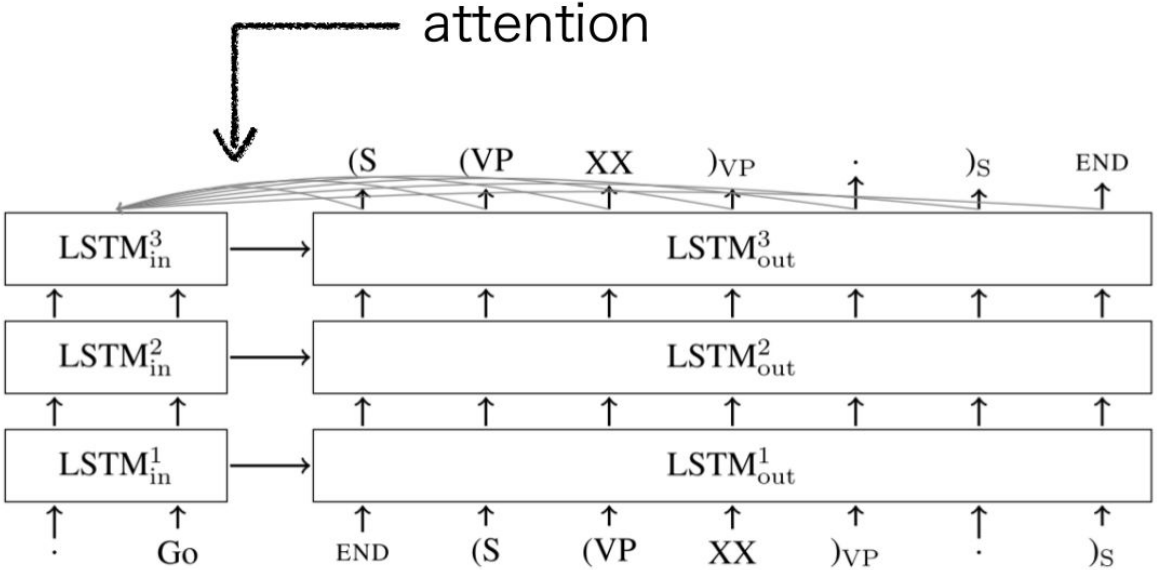
Implementation of Grammar as a Foreign Language

Given a input string, outputs a syntax tree



Implementation of Grammar as a Foreign Language

Given a input string, outputs a syntax tree



translate.py uses the same model, but for en-fr translation task

Basic Flow of the Implementation

Preprocess

Create Model

Train

Test

Use of Bucket

Preprocess

Create Model

Train

Test

Bucket is a method to efficiently handle sentences of different lengths

English sentence with length L1, French sentence with length L2 + 1 (prefixed with GO symbol)

English sentence -> encoder_inputs

French sentence -> decoder_inputs

we should in principle create a seq2seq model for every pair (L1, L2+1) of lengths of an English and French sentence. This would result in an enormous graph consisting of many very similar subgraphs.

On the other hand, we could just pad every sentence with a special PAD symbol. Then we'd need only one seq2seq model, for the padded lengths. But on shorter sentence our model would be inefficient, encoding and decoding many PAD symbols that are useless.

Use of Bucket

Preprocess

Create Model

Train

Test

As a compromise between constructing a graph for every pair of lengths and padding to a single length, we use a number of buckets and pad each sentence to the length of the bucket above it. In translate.py we use the following default buckets.

buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]

This means that if the input is an English sentence with 3 tokens, and the corresponding output is a French sentence with 6 tokens, then they will be put in the first bucket and padded to length 5 for encoder inputs, and length 10 for decoder inputs.

Encoder and Decoder Inputs in Bucket (5, 10)

Preprocess

Create Model

Train

Test

Original English	I go.
Original French	Je vais.

Encoder and Decoder Inputs in Bucket (5, 10)

Preprocess

Create Model

Train

Test

Original English	I go.
Tokenization	["I", "go", "."]
Original French	Je vais.
Tokenization	["Je", "vais", "."]

Encoder and Decoder Inputs in Bucket (5, 10)

Preprocess

Create Model

Train

Test

Original English	I go.
Tokenization	["I", "go", "."]
Encoder Input	["PAD""PAD"".", "go", "I"]

Original French	Je vais.
Tokenization	["Je", "vais", "."]
Decoder Input	["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"]

Creating Seq2Seq Attention Model

Preprocess

Create Model

Train

Test

Encoder Inputs [["PAD""PAD"".", "go", "I"], ...]

Decoder Inputs [["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"], ...]

model embedding_rnn_seq2seq(encoder_inputs, decoder_inputs, ..., feed_prev=False)

Creating Seq2Seq Attention Model

Preprocess

Create Model

Train

Test

Encoder Inputs [["PAD""PAD"".", "go", "I"], …]

Decoder Inputs [["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"], …]

model embedding_rnn_seq2seq(encoder_inputs, decoder_inputs, …, feed_prev=False)

embedding_rnn_seq2seq() is made of encoder + embedding_attention_decoder

Creating Seq2Seq Attention Model

Preprocess

Create Model

Train

Test

Encoder Inputs [["PAD""PAD"".", "go", "I"], …]

Decoder Inputs [["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"], …]

model embedding_rnn_seq2seq(encoder_inputs, decoder_inputs, …, feed_prev=False)

embedding_rnn_seq2seq() is made of encoder + embedding_attention_decoder()
embedding_attention_decoder() is made of embedding + attention_decoder()

Creating Seq2Seq Attention Model

Preprocess

Create Model

Train

Test

Encoder Inputs [["PAD""PAD"".", "go", "I"], ...]

Decoder Inputs [["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"], ...]

model embedding_rnn_seq2seq(encoder_inputs, decoder_inputs, ..., feed_prev=False)

"feed_prev = False" means that the decoder will use decoder_inputs tensors as provided

Creating Seq2Seq Attention Model

Preprocess

Create Model

Train

Test

Encoder Inputs [["PAD""PAD"".", "go", "I"], ...]

Decoder Inputs [["GO", "Je", "vais", ".","EOS","PAD","PAD","PAD","PAD","PAD"], ...]

model embedding_rnn_seq2seq(encoder_inputs, decoder_inputs, ..., feed_prev=False)

outputs, states = model_with_buckets(encoder_inputs, decoder_inputs, model, ...)

(Just a wrapper function that helps with using buckets)

Training

Preprocess

Create Model

Train

Test

session.run()

with:

- GradientDescentOptimizer
- Gradient Clipping by Global Norm

Training

Preprocess

Create Model

Train

Test

session.run()

with:

- GradientDescentOptimizer
- Gradient Clipping by Global Norm

```
global step 200 learning rate 0.5000 step-time 1.39 perplexity 1720.62
  eval: bucket 0 perplexity 184.97
  eval: bucket 1 perplexity 248.81
  eval: bucket 2 perplexity 341.64
  eval: bucket 3 perplexity 469.04
global step 400 learning rate 0.5000 step-time 1.38 perplexity 379.89
  eval: bucket 0 perplexity 151.32
  eval: bucket 1 perplexity 190.36
  eval: bucket 2 perplexity 227.46
  eval: bucket 3 perplexity 238.66
```

Testing

Preprocess

Create Model

Train

Test

decode()

with:

- feed_prev = True

"feed_prev = True" means that the decoder would only use the first element of decoder_inputs and previous output of the encoder from the second run.

Result

```
python translate.py --decode
  --data_dir [your_data_directory] --train_dir [checkpoints_directory]

Reading model parameters from /tmp/translate.ckpt-340000
>  Who is the president of the United States?
 Qui est le président des États-Unis ?
```

Let’s go to TensorFlow

```
import tensorflow as tf
```

Attention Mechanism and Its Variants

- Global attention

- Local attention

- Pointer networks

- Attention for image (image caption generation)

…

Attention Mechanism and Its Variants

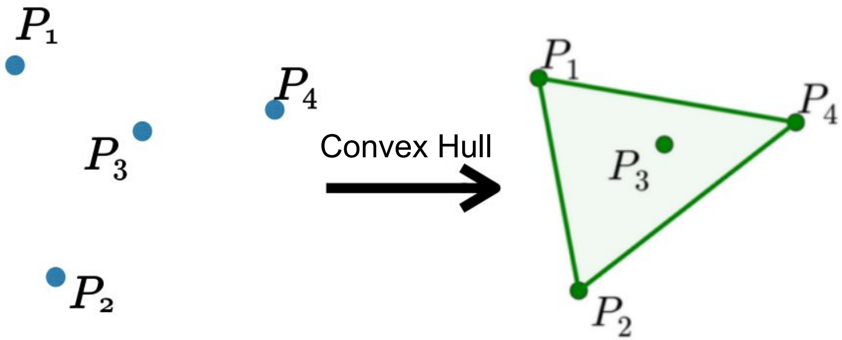
- Global attention

- Local attention

- Pointer networks ← this one for today

- Attention for image (image caption generation)

…

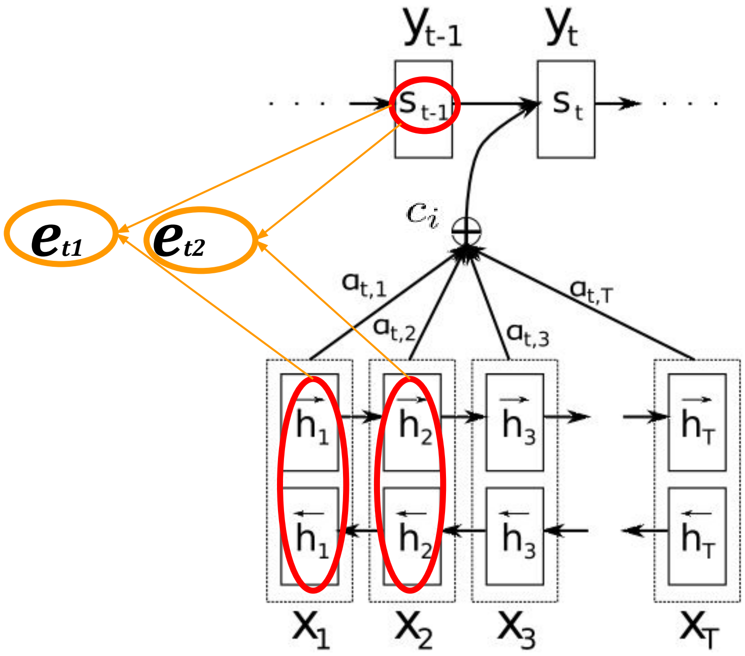


Pointer Networks Review

Pointer Networks 'Point' Input Elements!

In Ptr-Net, we do not blend the encoder state to propagate extra information to the decoder like standard attention mechanism.

But instead ...



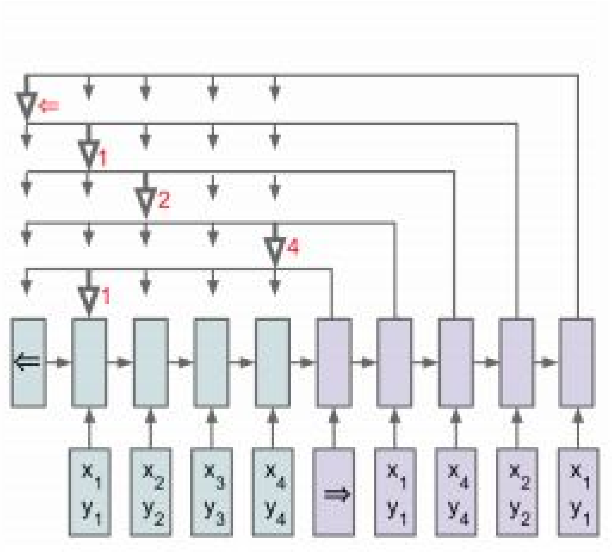
Standard Attention mechanism

Pointer Networks ‘Point’ Input Elements!

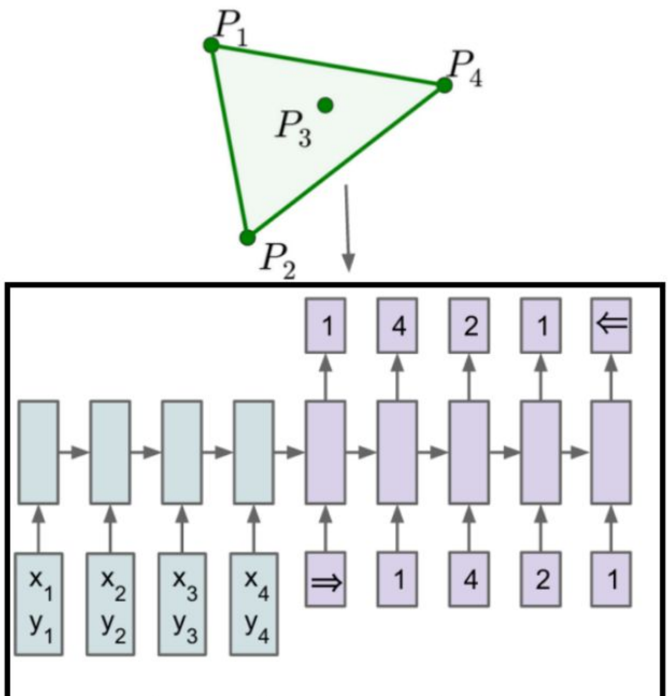
We use $e_{ij} = v_a^\top \tanh\left(W_a s_{i-1} + U_a h_j\right)$

as pointers to the input elements

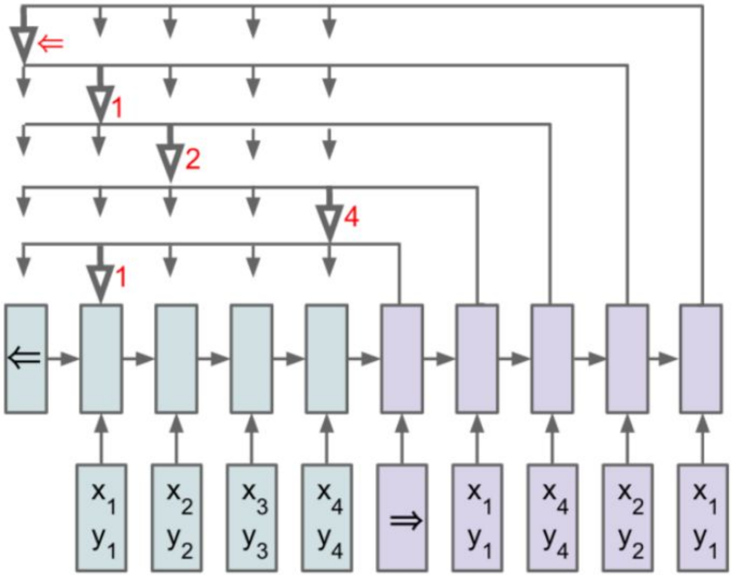
Ptr-Net approach specifically targets problems whose outputs are discrete and correspond to positions in the input

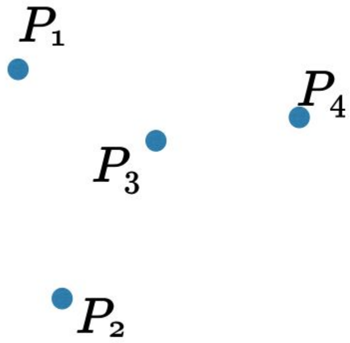


Pointer Network

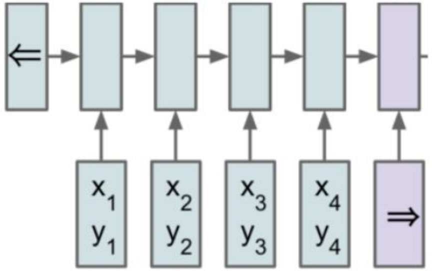


Another model for each number of points?

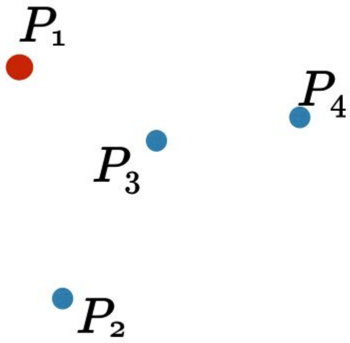




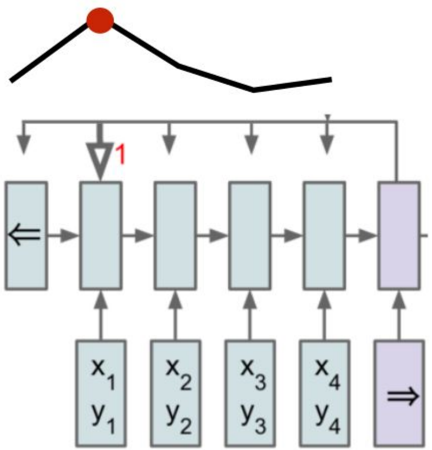
distribution of u



$$u_j^i = v^T \tanh(W_1 e_j + W_2 d_i)$$



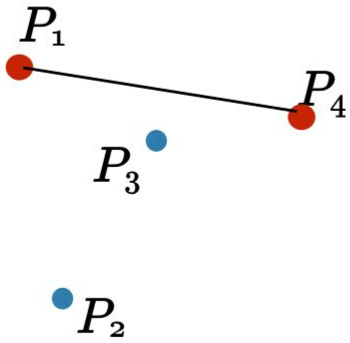
distribution of u

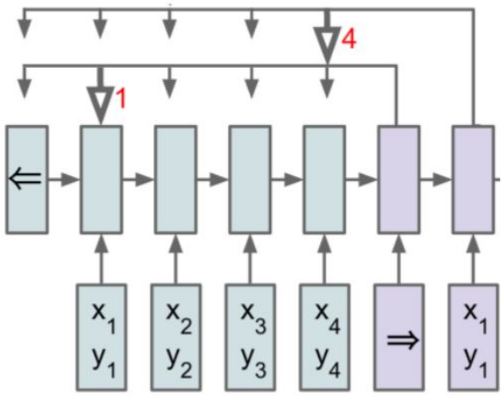


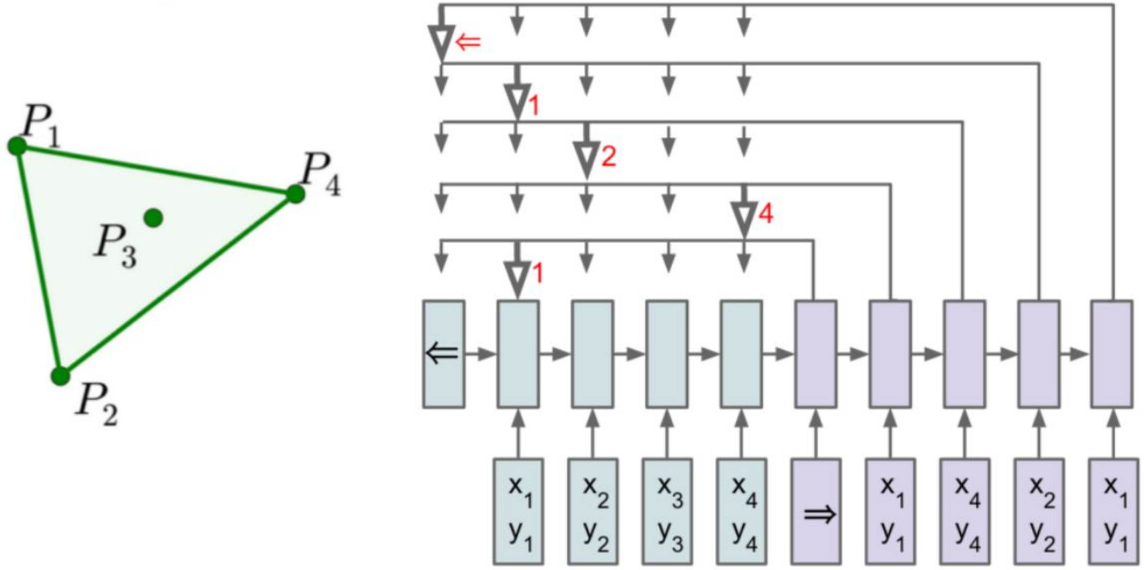
$$u_j^i = v^T \tanh(W_1 e_j + W_2 d_i)$$

$$p(C_i | C_1, \ldots, C_{i-1}, \mathcal{P}) = \text{softmax}(u^i)$$

Distribution of the Attention is the Answer!


P_1
P_4
P_3
P_2


4
1
⇐
x_1 y_1
x_2 y_2
x_3 y_3
x_4 y_4
⇒
x_1 y_1



P_1
P_4
P_3
P_2
⇐
1
2
4
1
⇐
x_1 y_1
x_2 y_2
x_3 y_3
x_4 y_4
⇒
x_1 y_1
x_4 y_4
x_2 y_2
x_1 y_1

Attention Mechanism vs Pointer Networks

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

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

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

Attention mechanism

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

$$p(C_i | C_1, \ldots, C_{i-1}, \mathcal{P}) = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

Ptr-Net

Softmax normalizes the vector e_{ij} to be an output distribution over the dictionary of inputs

Pointer Networks
Code Review

Pointer Networks
Code Review

Sorting Implementation

https://github.com/ikostrikov/TensorFlow-Pointer-Networks

Characteristics of the Implementation

This Implementation:

- The model code is a slightly modified version of attention_decoder from seq2seq tensorflow model
- Simple implementation but with poor comments

Characteristics of the Implementation

This Implementation:

- The model code is a slightly modified version of attention_decoder from seq2seq tensorflow model
- Simple implementation but with poor comments

Focus on:

- The general structure of the code (so you can refer while creating your own)
- How original Implementation of attention_decoder is modified

Task: "Order Matters" 4.4

- Learn “How to sort N ordered numbers between 0 and 1”
- ex) 0.2 0.5 0.6 0.23 0.1 0.9 => 0.1 0.2 0.23 0.5 0.6 0.9

Published as a conference paper at ICLR 2016

ORDER MATTERS: SEQUENCE TO SEQUENCE FOR SETS

Oriol Vinyals, Samy Bengio, Manjunath Kudlur
Google Brain
{vinyals, bengio, keveman}@google.com

ABSTRACT

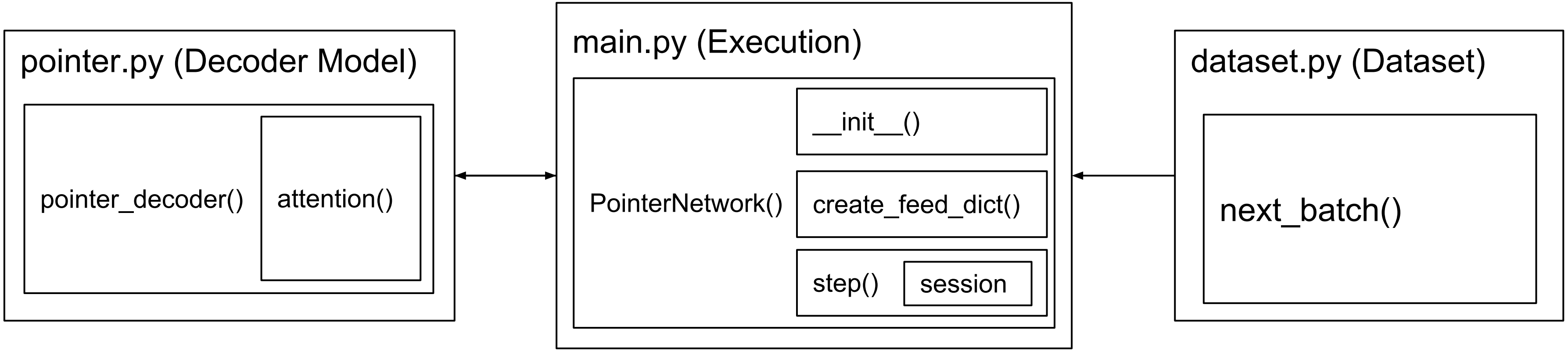
Sequences have become first class citizens in supervised learning thanks to the resurgence of recurrent neural networks. Many complex tasks that require mapping from or to a sequence of observations can now be formulated with the

4.4 SORTING EXPERIMENT

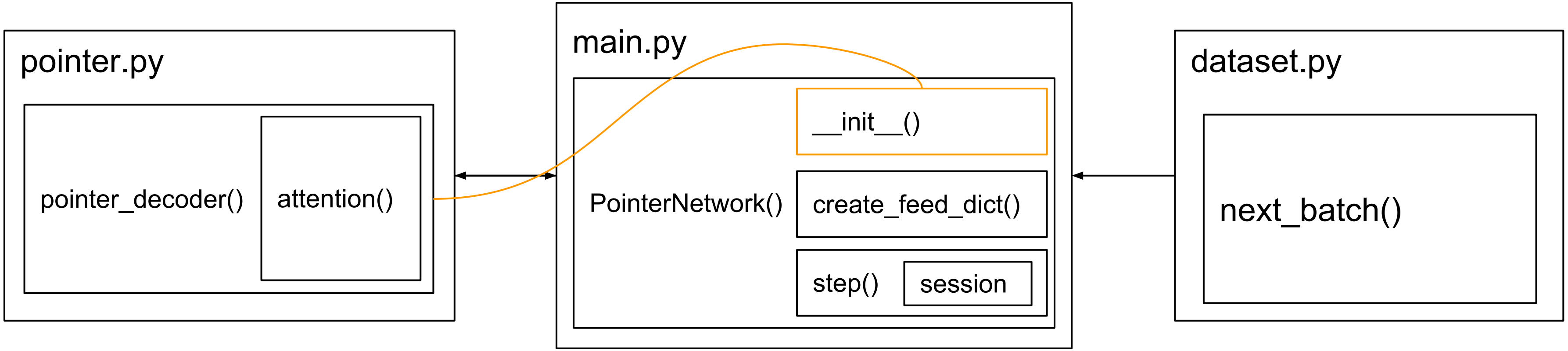
In order to verify if our model handles sets more efficiently than the vanilla seq2seq approach, we ran the following experiment on artificial data for the task of sorting numbers: given N unordered random floating point numbers between 0 and 1, we return them in a sorted order. Note that this

Structure of the Implementation

How the Implementation is Structured

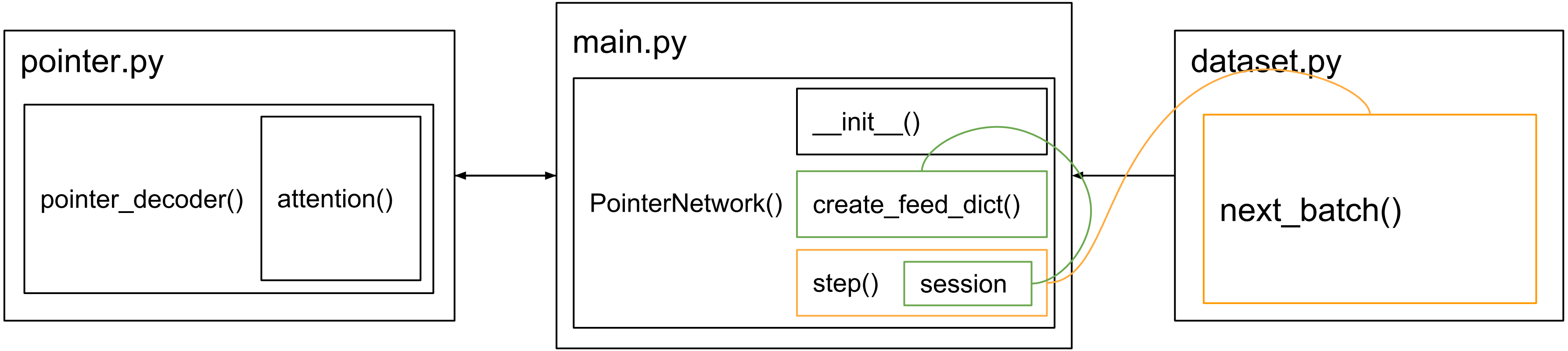


How the Implementation is Structured



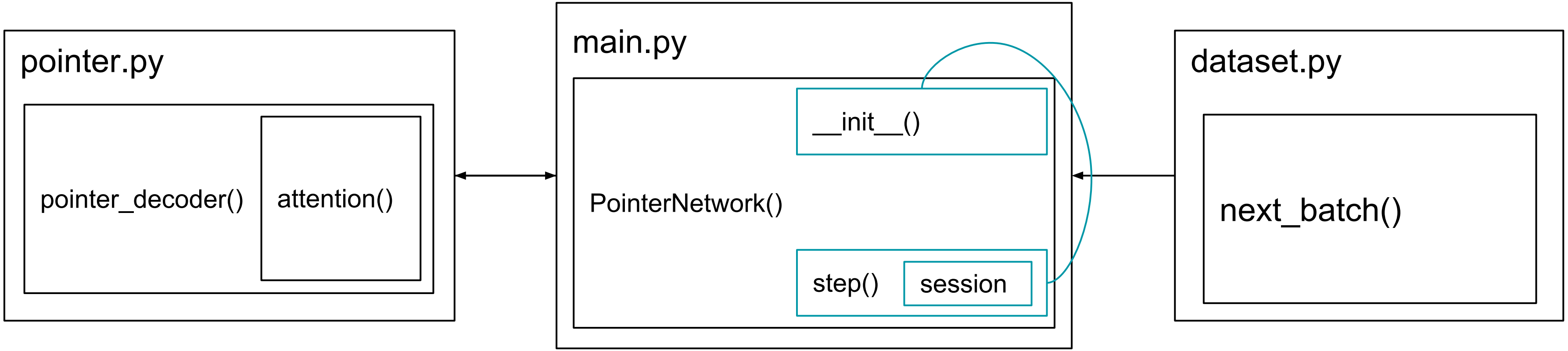
Encoder and decoder are instantiated in __init__()

How the Implementation is Structured



1. Dataset is instantiated and called in every step() (batch)
2. Create_feed_dict() is then used to feed the dataset into session.

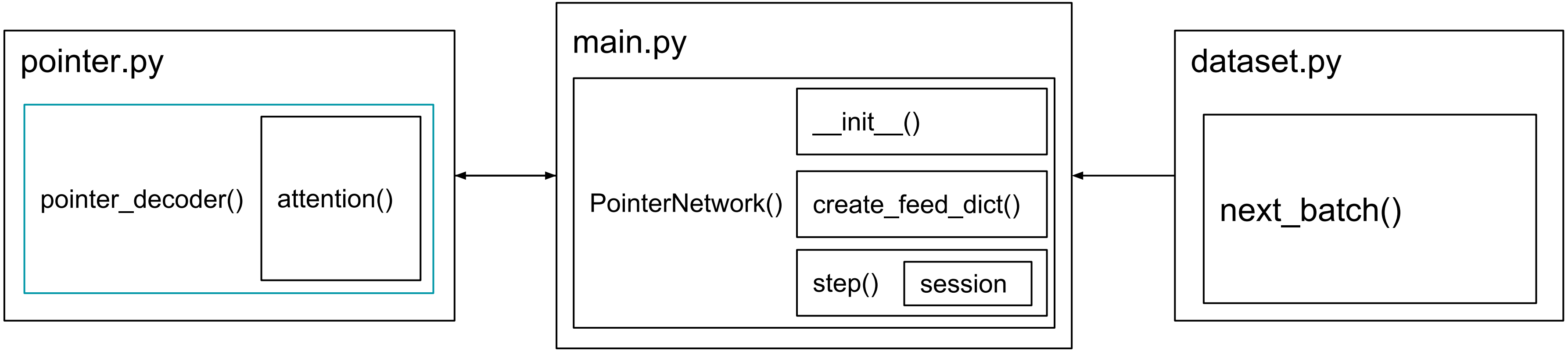
How the Implementation is Structured



Finally, step() uses the model created in __init__() to run the session

Brief Explanations of The Model Part

pointer_decoder()



pointer_decoder()

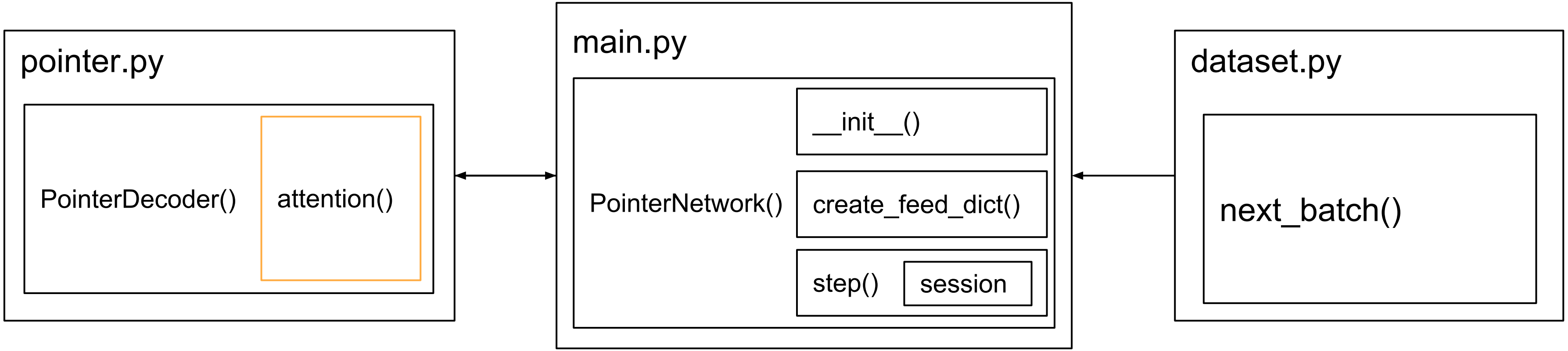
A Simple Modification to the attention_decoder tensorflow model

```
def pointer_decoder(decoder_inputs, initial_state, attention_states, cell,
                    feed_prev=True, dtype=dtypes.float32, scope=None):
```

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

$$p(C_i | C_1, \ldots, C_{i-1}, \mathcal{P}) = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

pointer_decoder(): attention()



pointer_decoder(): attention()

query == states

```
def attention(query):
    """Point on hidden using hidden_features and query."""
    with vs.variable_scope("Attention"):
        y = rnn_cell._linear(query, attention_vec_size, True)
        y = array_ops.reshape(y, [-1, 1, 1, attention_vec_size])
        # Attention mask is a softmax of v^T * tanh(...).
        s = math_ops.reduce_sum(
            v * math_ops.tanh(hidden_features + y), [2, 3])
        return s
```

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

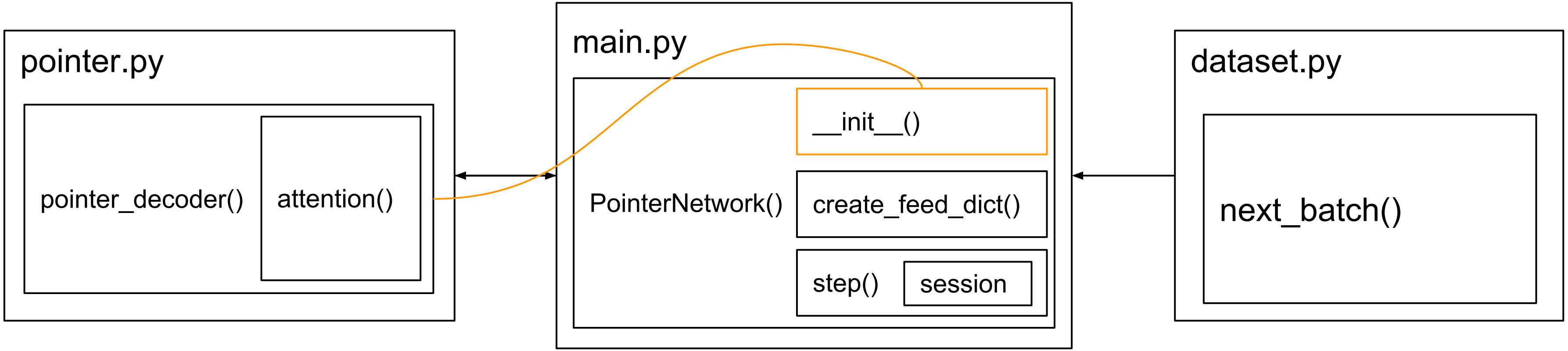
Standard Attention vs Pointer Attention in Code

```
def attention(query):
  """Put attention masks on hidden using hidden_features and query."""
  ds = []  # Results of attention reads will be stored here.
  if nest.is_sequence(query):  # If the query is a tuple, flatten it.
    query_list = nest.flatten(query)
    for q in query_list:  # Check that ndims == 2 if specified.
      ndims = q.get_shape().ndims
      if ndims:
        assert ndims == 2
    query = array_ops.concat(1, query_list)
  for a in xrange(num_heads):
    with variable_scope.variable_scope("Attention_%d" % a):
      y = linear(query, attention_vec_size, True)
      y = array_ops.reshape(y, [-1, 1, 1, attention_vec_size])
      # Attention mask is a softmax of v^T * tanh(...).
      s = math_ops.reduce_sum(
          v[a] * math_ops.tanh(hidden_features[a] + y), [2, 3])
      a = nn_ops.softmax(s)
      # Now calculate the attention-weighted vector d.
      d = math_ops.reduce_sum(
          array_ops.reshape(a, [-1, attn_length, 1, 1]) * hidden,
          [1, 2])
      ds.append(array_ops.reshape(d, [-1, attn_size]))
  return ds
```

attention fucntion from attention_decoder()

```
def attention(query):
    """Point on hidden using hidden_features and query."""
    with vs.variable_scope("Attention"):
        y = rnn_cell._linear(query, attention_vec_size, True)
        y = array_ops.reshape(y, [-1, 1, 1, attention_vec_size])
        # Attention mask is a softmax of v^T * tanh(...).
        s = math_ops.reduce_sum(
            v * math_ops.tanh(hidden_features + y), [2, 3])
        return s
```

attention fucntion from pointer_decoder()



Encoder and decoder are instantiated in __init__()

__init__()

Whole Encoder and Decoder Model is Made in Here

```
cell = tf.nn.rnn_cell.GRUCell(size)

# Need for attention
encoder_outputs, final_state = tf.nn.rnn(cell, self.encoder_inputs, dtype = tf.float32)

# Need a dummy output to point on it. End of decoding.
encoder_outputs = [tf.zeros([FLAGS.batch_size, FLAGS.rnn_size])] + encoder_outputs

# First calculate a concatenation of encoder outputs to put attention on.
top_states = [tf.reshape(e, [-1, 1, cell.output_size])
              for e in encoder_outputs]
attention_states = tf.concat(1, top_states)
```

```
with tf.variable_scope("decoder"):
    outputs, states, _ = pointer_decoder(
        self.decoder_inputs, final_state, attention_states, cell)

with tf.variable_scope("decoder", reuse=True):
    predictions, _, inps = pointer_decoder(
        self.decoder_inputs, final_state, attention_states, cell, feed_prev=True)

self.predictions = predictions

self.outputs = outputs
self.inps = inps
```

Let’s go to TensorFlow

```
import tensorflow as tf
```

Result

Step: 0
Train: 1.07584562302
Test: 1.07516384125
Correct order / All order: 0.000000
Step: 100
Train: 8.91889034099
Test: 8.91508702453
Correct order / All order: 0.000000
….
Step: 9800
Train: 0.437000320964
Test: 0.459392405155
Correct order / All order: 0.841875
Step: 9900
Train: 0.424404183739
Test: 0.636979421763
Correct order / All order: 0.825000

82%

Original Theano Implementation (Part)

```
def _ptr_probs(xm_, x_, h_, c_, _, hprevs, hprevs_m):
    xemb = p[x_, tensor.arange(n_samples), :]  # n_samples * dim_proj
    h, c = _lstm(xm_, xemb, h_, c_, 'lstm_de')
    u = tensor.dot(hprevs, tparams['ptr_W1']) + tensor.dot(h, tparams['ptr_W2'])  # n_steps * n_samples * dim
    u = tensor.tanh(u)  # n_sizes * n_samples * dim_proj
    u = tensor.dot(u, tparams['ptr_v'])  # n_sizes * n_samples
    # prob = tensor.nnet.softmax(u.T).T  # n_sizes * n_samples
    prob = softmax(hprevs_m, u)
    return h, c, prob
```

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

$$p(C_i|C_1,\ldots,C_{i-1},\mathcal{P}) = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x}\exp\left(e_{ik}\right)}$$

Thank you :D

References

- Many Slides from: http://www.slideshare.net/yutakikuchi927/deep-learning-nlp-attention
- Character Based Neural Machine Translation: http://arxiv.org/abs/1511.04586
- Grammar as a Foreign Language: https://arxiv.org/abs/1412.7449
- Tensorflow Official Tutorial on Seq2Seq Models: https://www.tensorflow.org/versions/r0.10/tutorials/seq2seq
- Pointer Networks: https://arxiv.org/abs/1506.03134
- Order Matters: http://arxiv.org/abs/1511.06391
- Pointer Netoworks Implementation: https://github.com/ikostrikov/TensorFlow-Pointer-Networks