Differentiable Learning of Logical Rules for Knowledge Base Reasoning

Fan Yang
Carnegie Mellon University

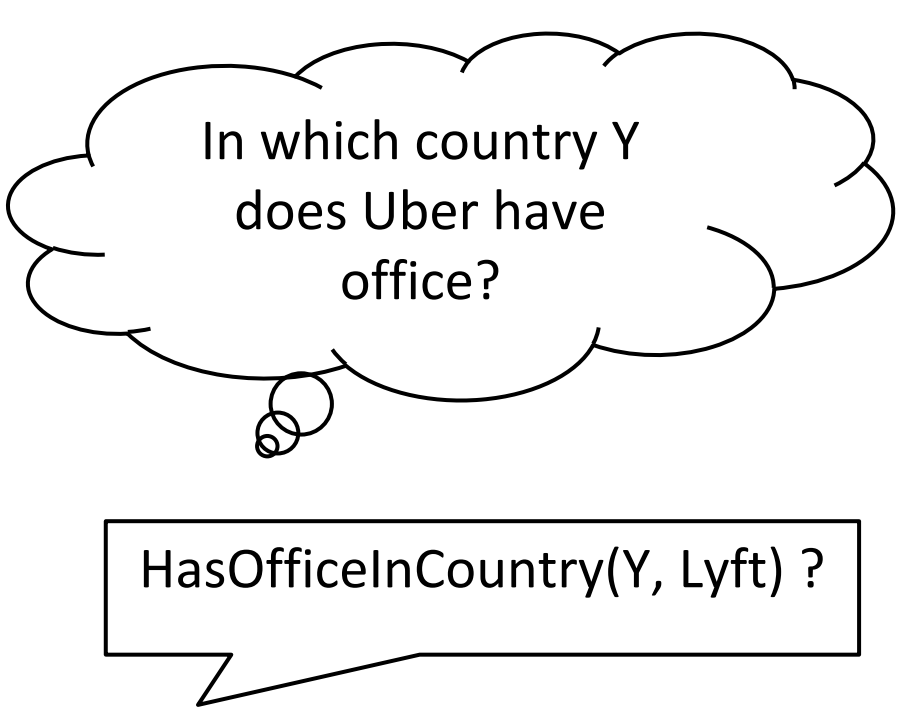
Joint work with Zhilin Yang and William W. Cohen

NIPS 2017 Workshop
Dec 9th, 2017

Carnegie Mellon University

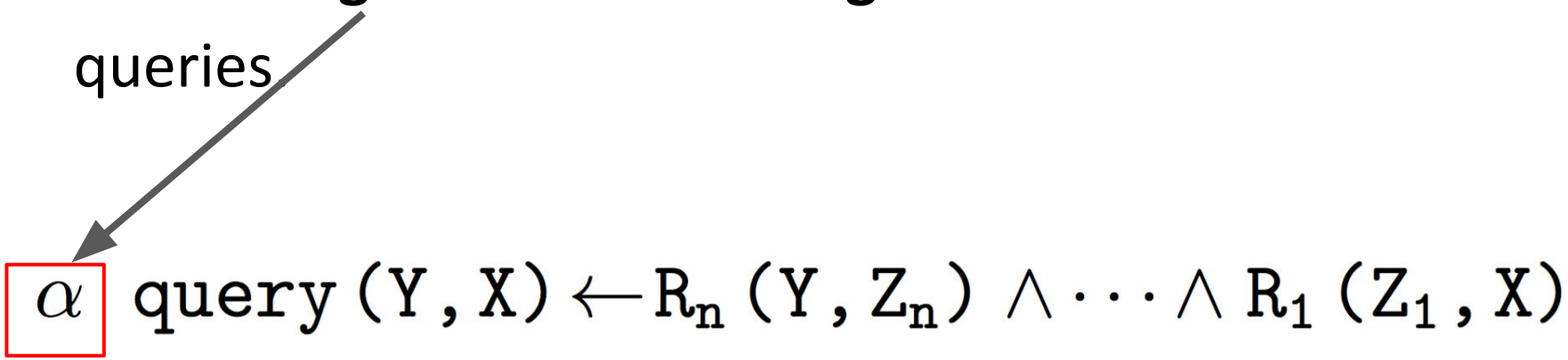
Problem definition

- Knowledge base (KB) reasoning
  - KB: a set of binary relations between entities.
  - Reasoning: retrieve entities from KB based on queries.



Our approach

- Learn **weighted chain-like logical rules** to answer queries



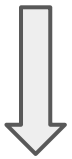
0.7 HasOfficeInCountry (Y, X) ← CityInCountry(Y, W) ⋀ HasOfficeInCity(W, X)

ungrounded variables

TensorLog Operators

- Represent **logical rule inference** using TensorLog operators (i.e. sparse matrices).
  - E = set of entities. R = set of binary relations.
  - For each entity i, define one-hot vector v_i.
  - For each relation R, define adjacency matrix M_R.

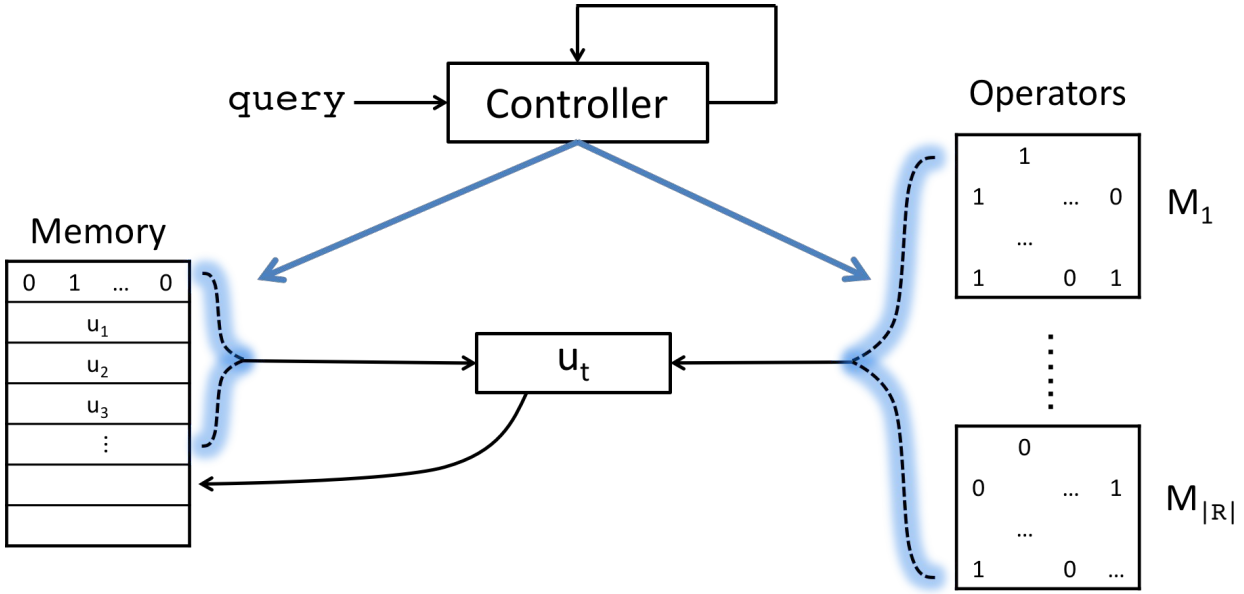
0.7 HasOfficeInCountry (Y, x) ← CityInCountry(Y, W) ⋀ HasOfficeInCity(W, x)



$$v_y = 0.7\ M_{\text{CityInCountry}}\ M_{\text{HasOfficeInCity}}\ v_x$$

Learn the logical rules

- Equivalent to **learning to compose** the operators.



$$\mathbf{u_0} = \mathbf{v_x}$$

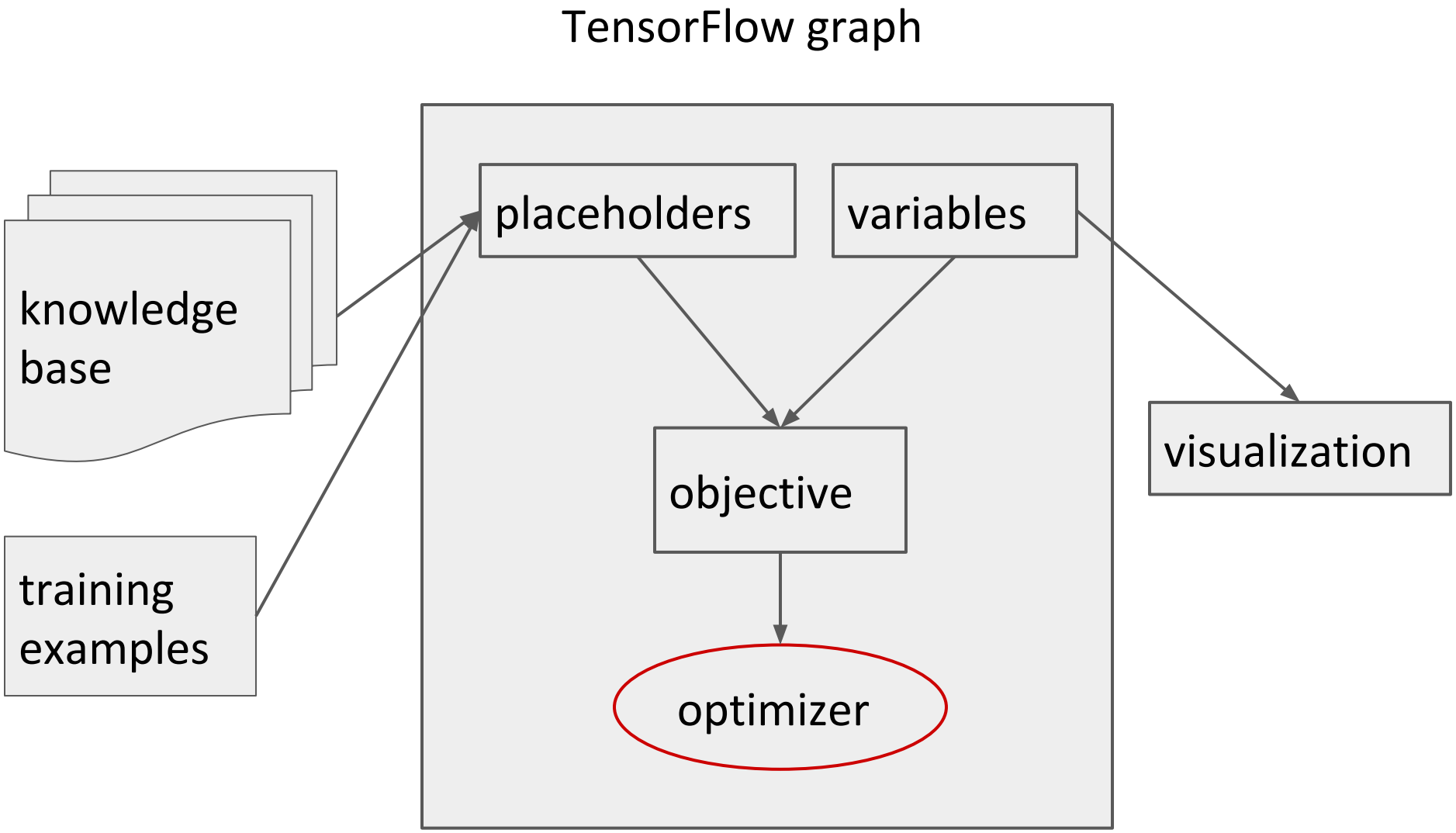
$$\mathbf{u_t} = \sum_{\mathbf{k}}^{|\mathbf{R}|} a_t^{\mathbf{k}} \mathbf{M}_{\mathbf{R_k}} \left(\sum_{\tau=0}^{t-1} b_t^{\tau} \mathbf{u}_{\tau} \right) \quad \text{for } 1 \leq t \leq T$$

$$\mathbf{u_{T+1}} = \sum_{\tau=0}^{T} b_{T+1}^{\tau} \mathbf{u}_{\tau}$$

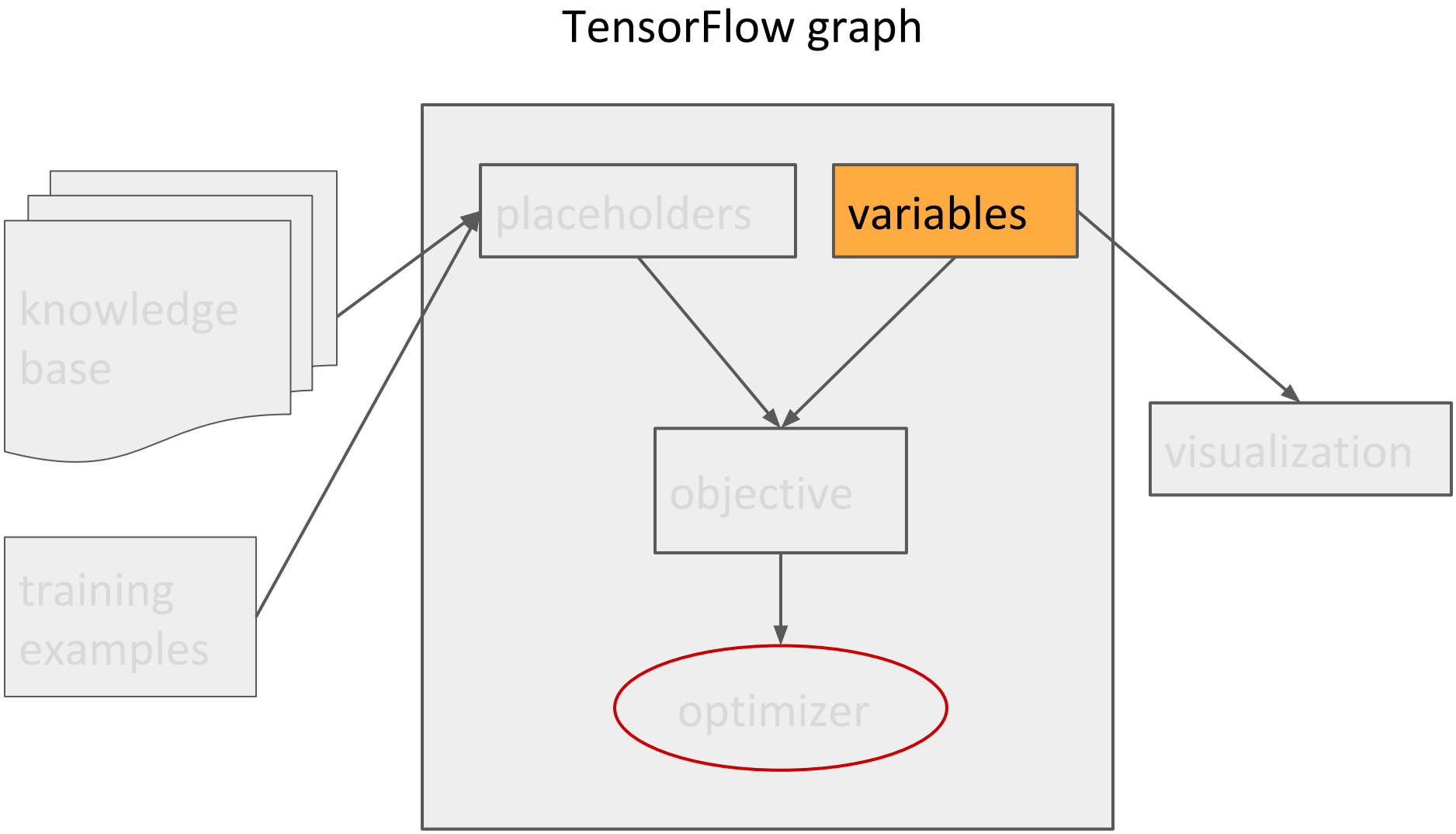
Implementation Outline

- Overview
- Learnable parameters
- Data flow and representation
- Training
- Sidelines for visualization

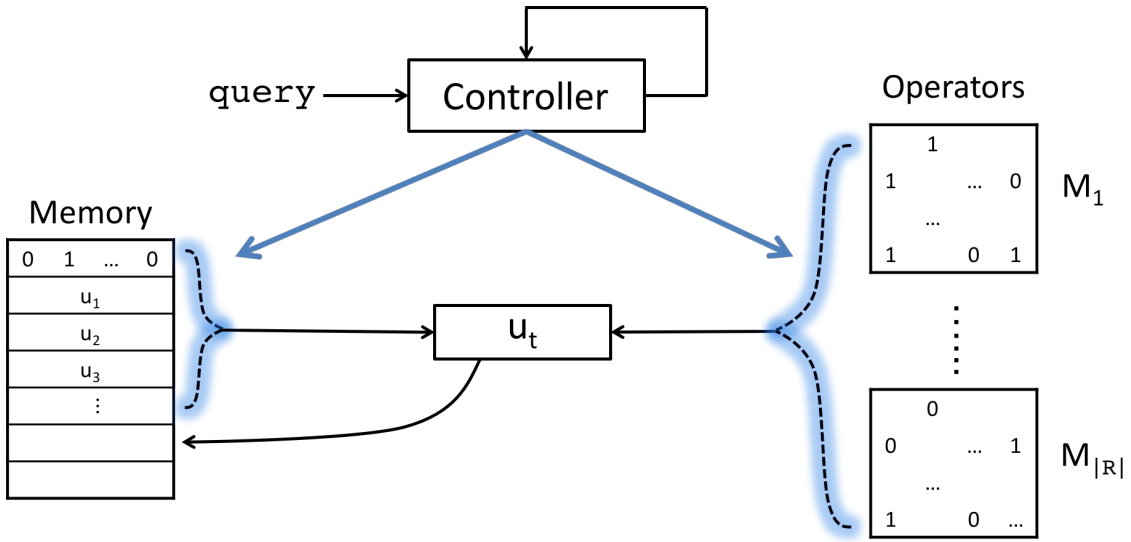
Implementation overview



Learnable parameters



Learnable parameters



Embeddings

RNN cell

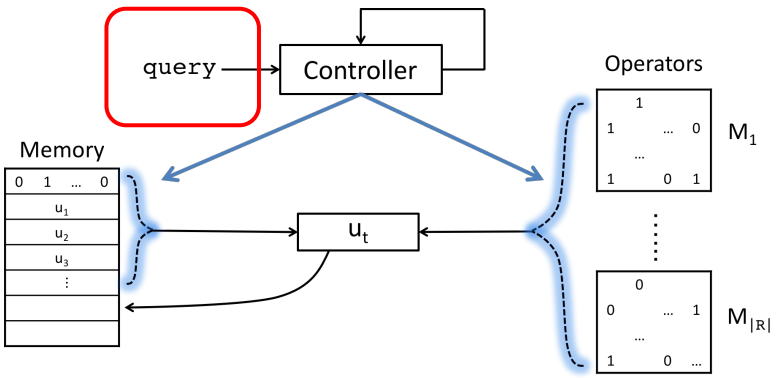
A linear layer

$$\mathbf{h_t} = \text{update}\left(\mathbf{h_{t-1}}, \text{input}\right)$$

$$\mathbf{a_t} = \text{softmax}\left(W\mathbf{h_t} + b\right)$$

$$\mathbf{b_t} = \text{softmax}\left(\left[\mathbf{h_0}, \ldots, \mathbf{h_{t-1}}\right]^T \mathbf{h_t}\right)$$

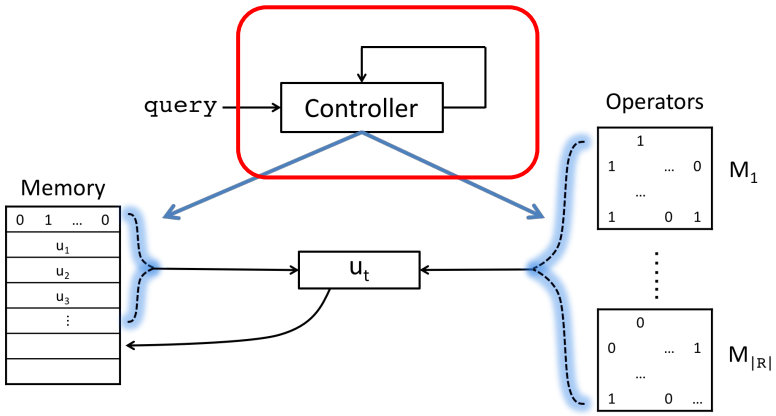
Implementing the architecture 1/5




```
if not self.query_is_language:
    self.queries = tf.placeholder(tf.int32, [None, self.num_step])
    self.query_embedding_params = tf.Variable(self._random_uniform_unit(
                                                self.num_query + 1, # <END> token
                                                self.query_embed_size),
                                              dtype=tf.float32)

    rnn_inputs = tf.nn.embedding_lookup(self.query_embedding_params,
                                        self.queries)
else:
    self.queries = tf.placeholder(tf.int32, [None, self.num_step, self.num_word])
    self.vocab_embedding_params = tf.Variable(self._random_uniform_unit(
                                                self.num_vocab + 1, # <END> token
                                                self.vocab_embed_size),
                                              dtype=tf.float32)
    embedded_query = tf.nn.embedding_lookup(self.vocab_embedding_params,
                                            self.queries)
    rnn_inputs = tf.reduce_mean(embedded_query, axis=2)

return rnn_inputs
```

- Query format
  - Natural language
  - Structured

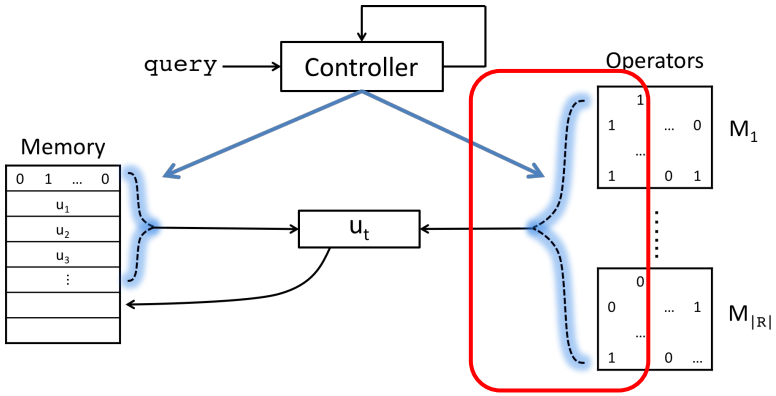
Implementing the architecture 2/5



$$\mathbf{h_t} = \text{update}\left(\mathbf{h_{t-1}}, \text{input}\right)$$
$$\mathbf{a_t} = \text{softmax}\left(W\mathbf{h_t} + b\right)$$
$$\mathbf{b_t} = \text{softmax}\left(\left[\mathbf{h_0}, \ldots, \mathbf{h_{t-1}}\right]^T \mathbf{h_t}\right)$$

```
""" Build a computation graph that represents the model """
rnn_inputs = self._build_input()
# rnn_inputs: a list of num_step tensors,
# each tensor of size (batch_size, query_embed_size).
self.rnn_inputs = [tf.reshape(q, [-1, self.query_embed_size])
                   for q in tf.split(rnn_inputs,
                                     self.num_step,
                                     axis=1)]

cell = tf.contrib.rnn.core_rnn_cell.LSTMCell(self.rnn_state_size,
                                             state_is_tuple=True)
self.cell = tf.contrib.rnn.core_rnn_cell.MultiRNNCell(
                                             [cell] * self.num_layer,
                                             state_is_tuple=True)
init_state = self.cell.zero_state(tf.shape(self.tails)[0], tf.float32)

# rnn_outputs: a list of num_step tensors,
# each tensor of size (batch_size, rnn_state_size).
self.rnn_outputs, self.final_state = tf.contrib.rnn.static_rnn(
                                        self.cell,
                                        self.rnn_inputs,
                                        initial_state=init_state)
```

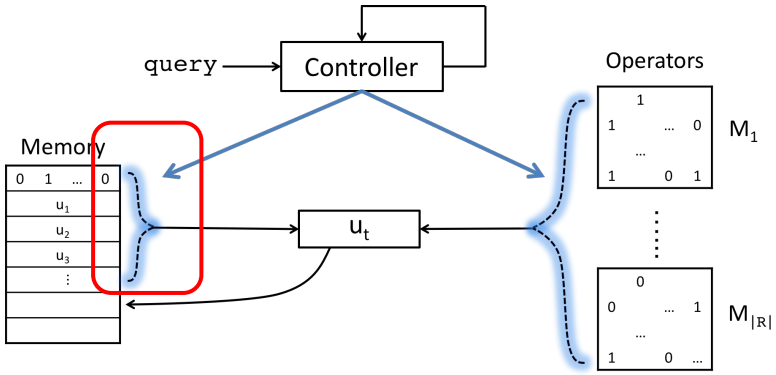
Implementing the architecture 3/5



$$\mathbf{h_t} = \text{update}\left(\mathbf{h_{t-1}}, \text{input}\right)$$
$$\mathbf{a_t} = \text{softmax}\left(W\mathbf{h_t} + b\right)$$
$$\mathbf{b_t} = \text{softmax}\left(\left[\mathbf{h_0}, \ldots, \mathbf{h_{t-1}}\right]^T \mathbf{h_t}\right)$$

```
self.W = tf.Variable(np.random.randn(
                         self.rnn_state_size,
                         self.num_operator),
                     dtype=tf.float32)
self.b = tf.Variable(np.zeros(
                         (1, self.num_operator)),
                     dtype=tf.float32)

# attention_operators: a list of num_step lists,
# each inner list has num_operator tensors,
# each tensor of size (batch_size, 1).
# Each tensor represents the attention over an operator.
self.attention_operators = [tf.split(
                             tf.nn.softmax(
                               tf.matmul(rnn_output, self.W) + self.b),
                             self.num_operator,
                             axis=1)
                             for rnn_output in self.rnn_outputs]
```

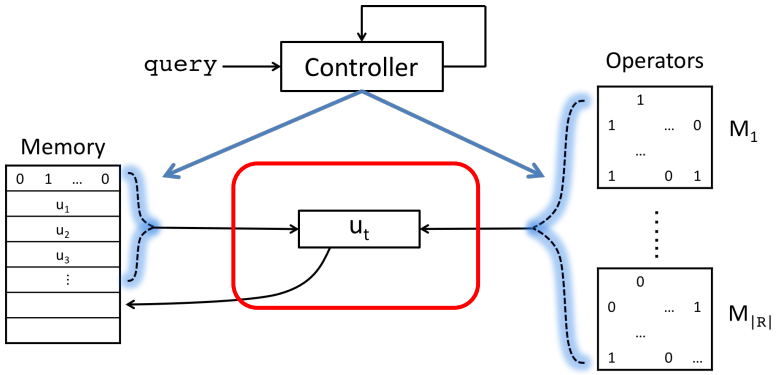
Implementing the architecture 4/5



$$\mathbf{h_t} = \text{update}\left(\mathbf{h_{t-1}}, \text{input}\right)$$
$$\mathbf{a_t} = \text{softmax}\left(W\mathbf{h_t} + b\right)$$
$$\mathbf{b_t} = \text{softmax}\left(\left[\mathbf{h_0}, \ldots, \mathbf{h_{t-1}}\right]^T \mathbf{h_t}\right)$$

```
for t in xrange(self.num_step):
    self.attention_memories.append(
                    tf.nn.softmax(
                    tf.squeeze(
                        tf.matmul(
                            tf.expand_dims(self.rnn_outputs[t], 1),
                            tf.stack(self.rnn_outputs[0:t+1], axis=2)),
                    squeeze_dims=[1])))

    # memory_read: tensor of size (batch_size, num_entity)
    memory_read = tf.squeeze(
                    tf.matmul(
                        tf.expand_dims(self.attention_memories[t], 1),
                        self.memories),
                    squeeze_dims=[1])
```

Implementing the architecture 5/5



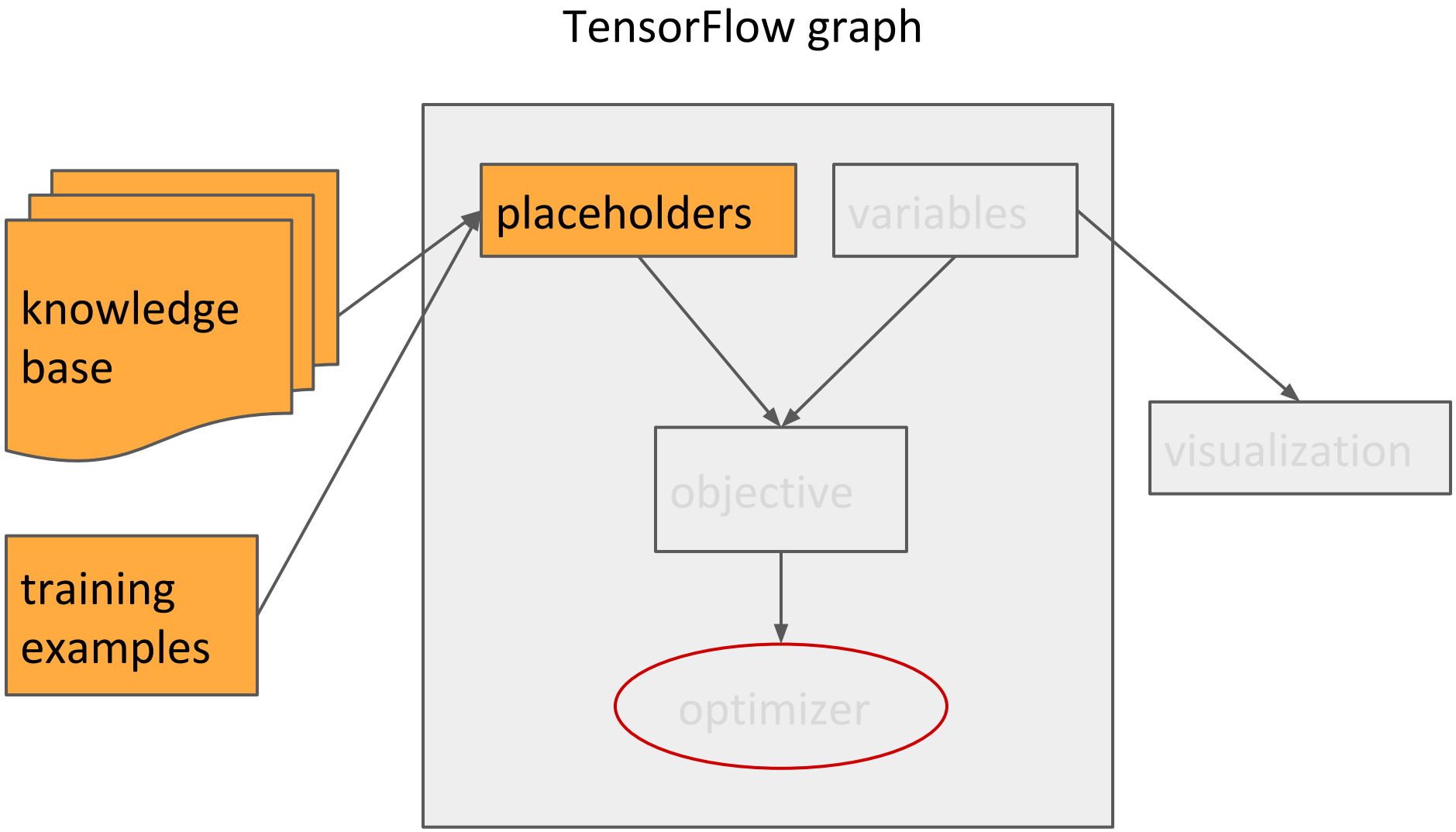
$$\mathbf{u_0} = \mathbf{v_x}$$

$$\mathbf{u_t} = \sum_{\mathbf{k}}^{|\mathbf{R}|} a_t^{\mathbf{k}} \mathbf{M}_{\mathbf{R_k}} \left(\sum_{\tau=0}^{t-1} b_t^{\tau} \mathbf{u}_{\tau} \right)$$

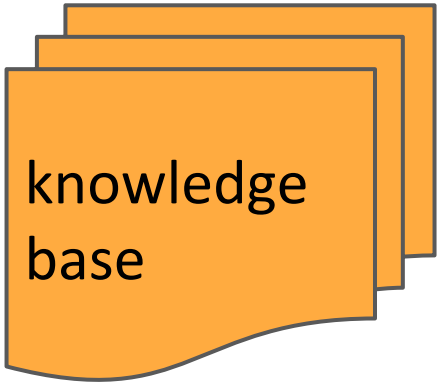
$$\mathbf{u_{T+1}} = \sum_{\tau=0}^{T} b_{T+1}^{\tau} \mathbf{u}_{\tau}$$

```
for r in xrange(self.num_operator/2):
    for op_matrix, op_attn in zip(
                    [self.database[r],
                     tf.sparse_transpose(self.database[r])],
                    [self.attention_operators[t][r],
                     self.attention_operators[t][r+self.num_operator/2]]):
        product = tf.sparse_tensor_dense_matmul(op_matrix, memory_read)
        database_results.append(tf.transpose(product) * op_attn)

added_database_results = tf.add_n(database_results)

# Populate a new cell in memory by concatenating.
self.memories = tf.concat(
    [self.memories,
    tf.expand_dims(added_database_results, 1)],
    axis=1)
```

Data flow


TensorFlow graph
knowledge base
training examples
placeholders
variables
objective
optimizer
visualization

Data representation



Format:

- e_A Relation e_B

num_operator

num_entity

training
examples

Format:

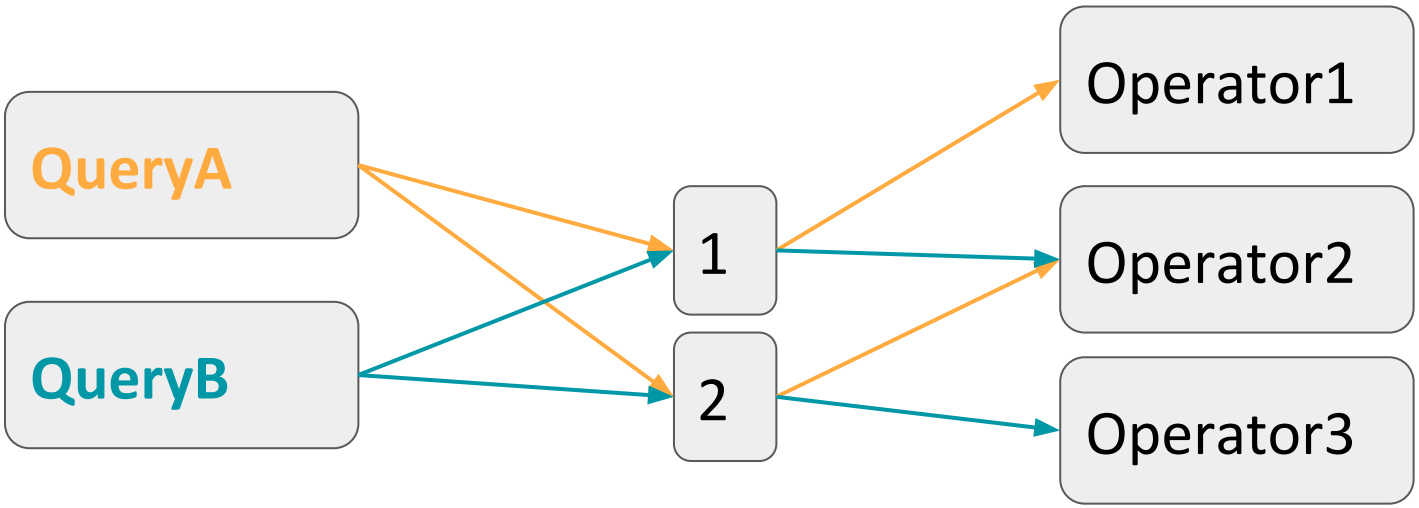
- natural language
  - e_{head}, w_1, …, w_n, e_{tail}
- structured
  - e_{head}, query, e_{tail}

num_vocab

num_query

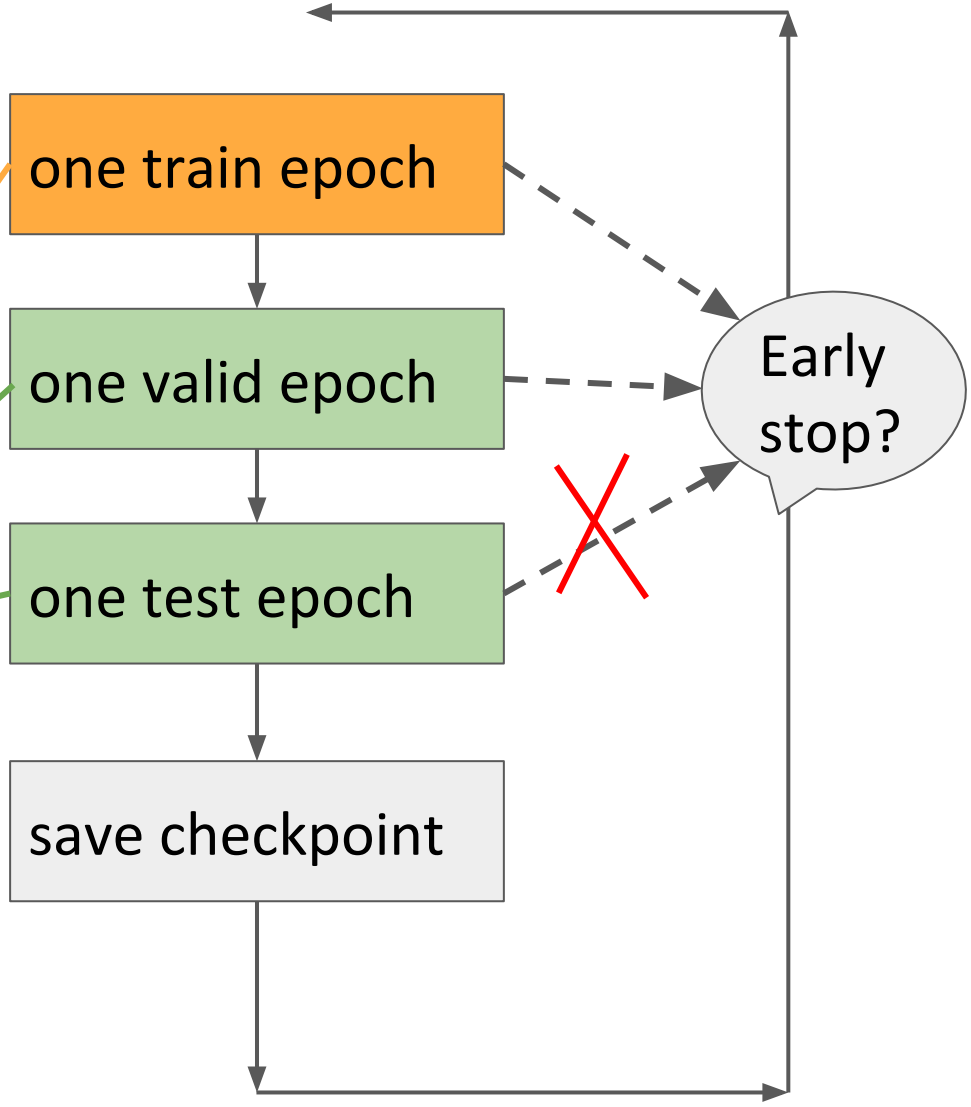
Data representation

- Mapping
  - query_to_id
    - Used to look up embeddings.
  - id_to_operator
    - Can vary across queries.



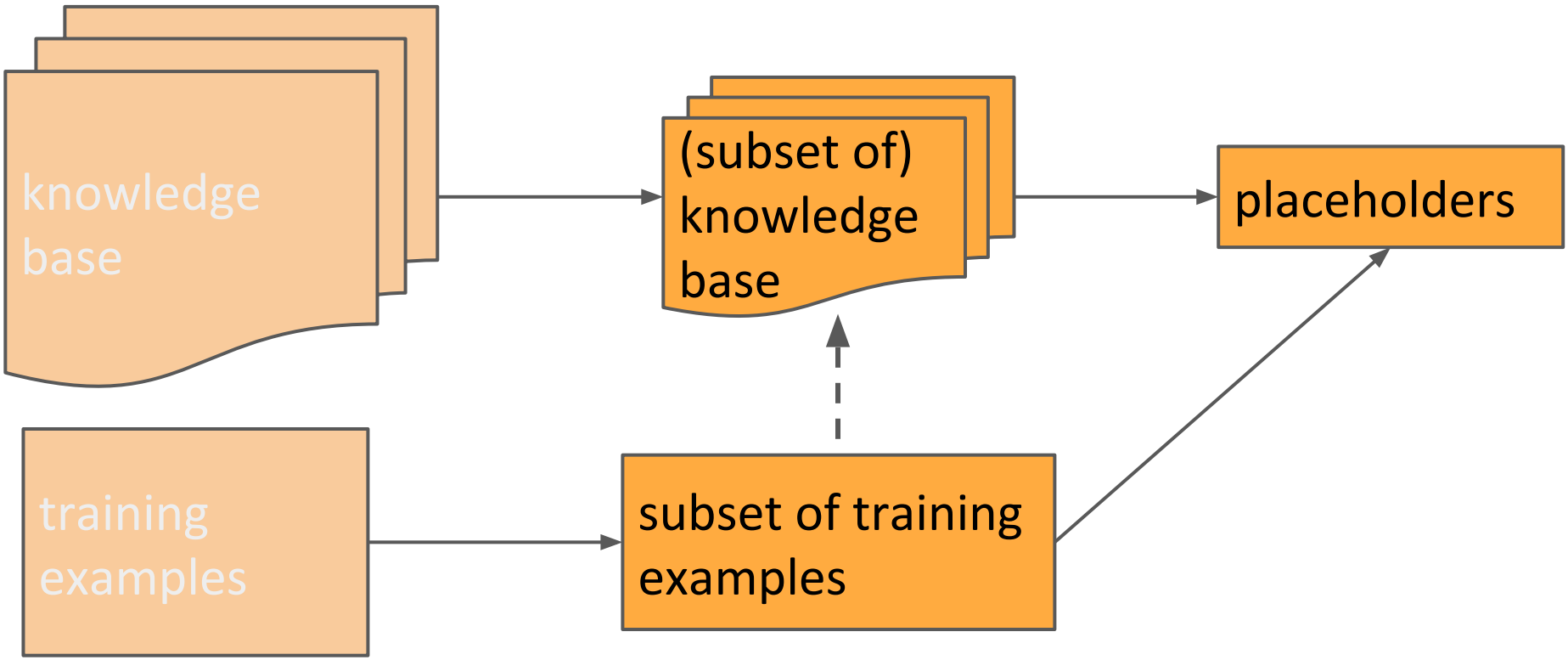
Training

```
def one_epoch(self, mode, num_batch, next_fn):
    epoch_loss = []
    epoch_in_top = []
    for batch in xrange(num_batch):
        if (batch+1) % max(1, (num_batch / self.option.pr
            sys.stdout.write("%d/%d\t" % (batch+1, num_ba
            sys.stdout.flush()

        (qq, hh, tt), mdb = next_fn()
        if mode == "train":
            run_fn = self.learner.update
        else:
            run_fn = self.learner.predict
        loss, in_top = run_fn(self.sess,
                              qq,
                              hh,
                              tt,
                              mdb)
        epoch_loss += list(loss)
        epoch_in_top += list(in_top)
```

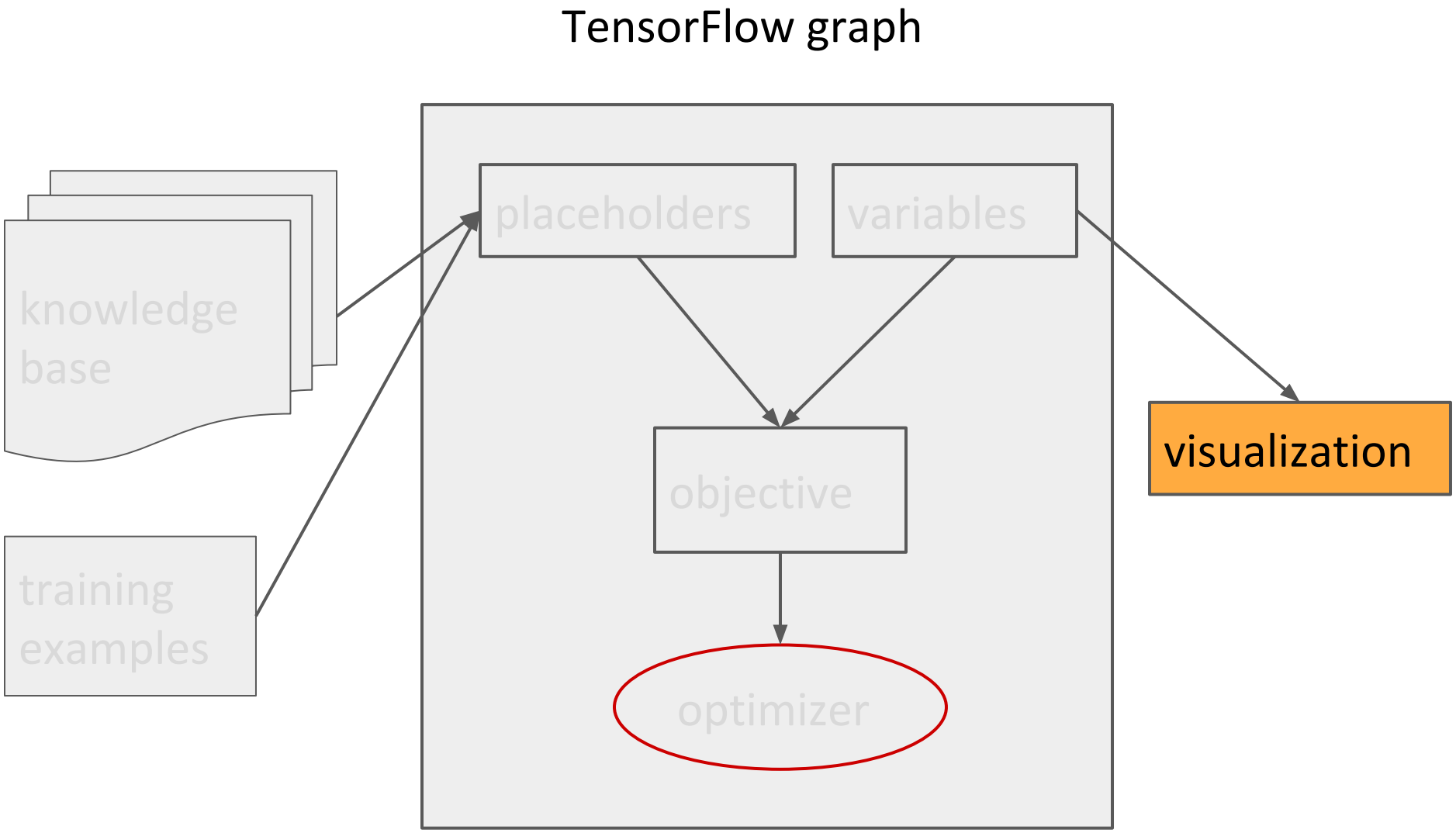


Training

- Batching
  - Batch size depends on how much CPU/GPU memory is available.
  - Examples in the batch use the same knowledge base.



Sidelines


TensorFlow graph
placeholders
variables
knowledge base
objective
visualization
training examples
optimizer

Future work

- Scalability
  - Use graph sampling to reduce num_entity.
- Expressiveness
  - Extend chain-like rules to poly-tree.
- Application
  - Queries in the format of images.

Thanks!

Checkout our implementation at:

https://github.com/fanyangxyz/Neural-LP