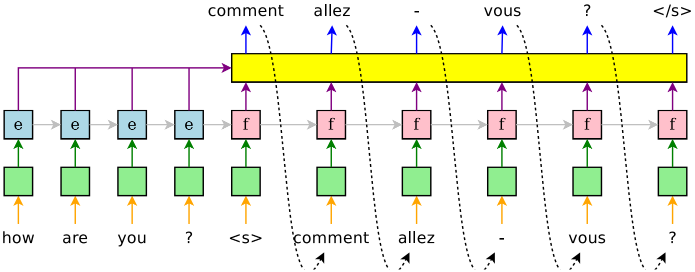
11-695: Competitive Engineering
Recurrent Neural Networks

Spring 2018

Outline

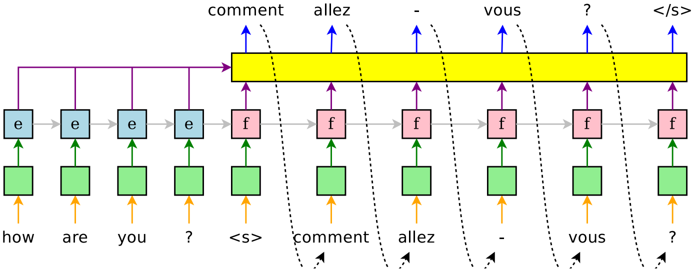
1. Regularization in Recurrent Neural Networks

2. Coding an RNN with TF Dynamic Graph



- Each colored arrowed can be dropped using *the same mask*.
  - Word embeddings dropout mean to remove *the whole word*

Other Strategies: ℓ_p



- ℓ_2 norm of all or some parameters
- ℓ_2 norm of all or some hidden states: $\sum_i \|\mathbf{e}_i\|^2$, $\sum_j \|\mathbf{f}_j\|^2$
- ℓ_2 difference of all or some hidden states: $\sum_i \|\mathbf{e}_i - \mathbf{e}_{i-1}\|^2$, $\sum_j \|\mathbf{f}_j - \mathbf{f}_{j-1}\|^2$

Outline

1. Regularization in Recurrent Neural Networks

2. Coding an RNN with TF Dynamic Graph

Hello, Old Friend: `tf.while_loop`

tf_while_loop.py

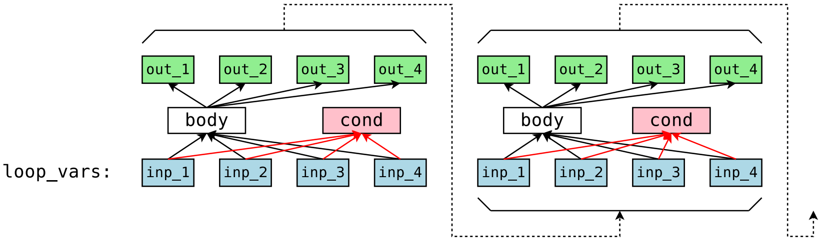
```python
def build_tf_graph:
  def condition(i, *args): return tf.less(i, 10)

  def body(i, a, b): return i+1, b, a+b

  loop_vars = [tf.constant(0, dtype=tf.int32), tf.constant(1, dtype=tf.int32),
               tf.constant(1, dtype=tf.int32)]

  loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?
  - Computes the Fibonacci numbers.

Hello, Old Friend: tf.while_loop

tf_while_loop.py

```
def build_tf_graph:
  def condition(i, *args): return tf.less(i, 10)

  def body(i, a, b): return i+1, b, a+b

  loop_vars = [tf.constant(0, dtype=tf.int32), tf.constant(1, dtype=tf.int32),
               tf.constant(1, dtype=tf.int32)]

  loop_outputs = tf.while_loop(condition, body, loop_vars)
```



What Exactly Is `loop_outputs`?

`tf_while_loop_outputs.py`

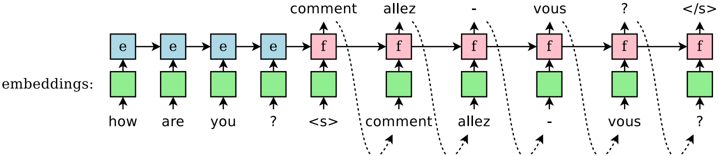
```python
def build_tf_graph:
  def condition(i, *args): return tf.less(i, 10)
  def body(i, a, b): return i+1, b, a+b
  loop_vars = [tf.constant(0, dtype=tf.int32), tf.constant(1, dtype=tf.int32),
               tf.constant(1, dtype=tf.int32)]

  loop_outputs = tf.while_loop(condition, body, loop_vars)
  print(type(loop_outputs))  # <type 'list'>
  for loop_output in loop_outputs:
    print(type(loop_output))  # TF Tensor
```

- `loop_outputs` is a *nested structure*
  - the same structure with `loop_vars`
  - Each *inner-most* element is the TF ops that triggers the loop and returns the corresponding output.

Using tf.while_loop to Build Machine Translation

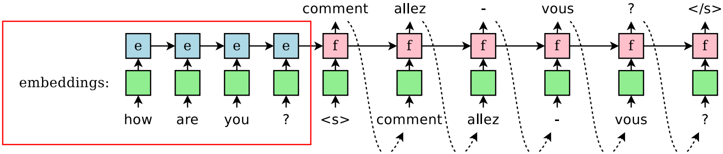
tf_rnns.py

```python
def build_tf_graph:
  # en_sent, fr_sent: Tensors with unknown shapes [1, E], [1, F].
  en_sent, fr_sent = read_data()

  # encoder_states: [1, E, hidden_size]
  encoder_states = encoder(en_sent)

  # decoder_states: [1, F, hidden_size]
  decoder_states = decoder(encoder_states, fr_sent)
```



- We will use *2 while loops*
  - one for encoder; one for decoder

Coding the Encoder with `tf.while_loop`



tf_seq2seq_encoder.py

```python
def encoder(sent, vocab_size=10000, hidden_size=128):
  # sent: Tensors with unknown shape [1, E]
  with tf.variable_scope("encoder"):
    w_emb = tf.get_variable("w_emb", [vocab_size, hidden_size])  # "encoder/w_emb"

  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])

  def body(i, sent, *args): # later
```

- Use the *unknown* length of `sent` to stop the loop
- Problem: we do not know how much memory to allocate

Coding the Encoder with `tf.while_loop`

tf_seq2seq_encoder.py

```python
def encoder(sent, vocab_size=10000, hidden_size=128):
  # sent: Tensors with unknown shape [1, E]
  with tf.variable_scope("encoder"):
    w_emb = tf.get_variable("w_emb", [vocab_size, hidden_size])  # "encoder/w_emb"

  hidden_states = tf.TensorArray(tf.float32, size=tf.shape(E)[-1],
                                clear_after_read=False)
  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])

  def body(i, sent, hidden_states):
    # do the RNN computations; write the new information to hidden_states
    return i+1, sent, hidden_states

  loop_vars = [tf.constant(0, dtype=tf.int32), sent, hidden_states]
  loop_outputs = tf.while_loop(condition, body, loop_vars)
  hidden_states = loop_outputs[-1].stack()  # [|E|, 1, hidden_states]
  return tf.transpose(hidden_states, [1, 0, 2])
```

- Problem: we do not know how much memory to allocate
  - Solution: `tf.TensorArray` supports *dynamic* memory allocation.

Writing to tf.TensorArray

CarnegieMellon

tf_seq2seq_encoder.py

```
def encoder(sent, vocab_size=10000, hidden_size=128):
  # create variables: w_emb, ...
  hidden_states = tf.TensorArray(tf.float32, size=tf.shape(E)[-1],
                                 clear_after_read=False)
  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])

  def body(i, sent, prev_state, hidden_states):
    # sent[:, i]: [1, 1] --> emb: [1, 1, hidden_size]
    emb = tf.nn.embedding_lookup(w_emb, sent[:, i])

    # do the RNN computations; write the new information to hidden_states
    next_state = prev_state + emb
    hidden_states = hidden_states.write(i, next_state)
    return i+1, sent, next_state, hidden_states

  loop_vars = [tf.constant(0, dtype=tf.int32), sent,
               tf.zeros([1, 1, hidden_size], dtype=tf.float32), hidden_states]
  loop_outputs = tf.while_loop(condition, body, loop_vars)
  hidden_states = loop_outputs[-1].stack()  # [|E|, 1, hidden_states]
  return tf.transpose(hidden_states, [1, 0, 2])
```

- A dummy encoder network

Vanilla RNN

Carnegie Mellon

tf_seq2seq_encoder.py

```
def encoder(sent, vocab_size=10000, hidden_size=128):
  with tf.variable_scope("encoder"):
    w_emb = tf.get_variable("w_emb", [vocab_size, hidden_size])
    w_rnn = tf.get_variable("w_rnn", [hidden_size, hidden_size])
    w_inp = tf.get_variable("w_inp", [hidden_size, hidden_size])

  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])
  def body(i, sent, prev_state, hidden_states):
    # sent[:, i]: [1, 1] --> emb: [1, 1, hidden_size]
    emb = tf.nn.embedding_lookup(w_emb, sent[:, i])

    # do the RNN computations; write the new information to hidden_states
    next_state = tf.tanh(tf.matmul(prev_state, w_rnn) + tf.matmul(emb, w_inp))
    hidden_states = hidden_states.write(i, next_state)
    return i+1, sent, next_state, hidden_states

  # create loop_vars, calling tf.while\_loop, stack, transpose, return
```

- A less dummy encoder network:

$$f(\mathbf{x}_t, \mathbf{h}_{t-1}) = \tanh\left(\mathbf{h}_{t-1} \cdot \mathbf{W}_h + \mathbf{x}_t \cdot \mathbf{W}_x\right) \tag{1}$$

LSTM? No Problem!

Carnegie Mellon

- Want Long Short-Term Memory (LSTM)? No problem!

$$\begin{aligned}
\mathbf{i}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xi} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hi}) \\
\mathbf{f}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xf} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hf}) \\
\mathbf{o}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xo} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{ho}) \\
\mathbf{g}_t &= \tanh(\mathbf{x}_t \cdot \mathbf{W}_{xg} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hg}) \\
\mathbf{c}_t &= \mathbf{f}_t \otimes \mathbf{c}_{t-1} + \mathbf{i}_t \otimes \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \otimes \tanh(\mathbf{c}_t)
\end{aligned} \tag{2}$$

LSTM? No Problem!

Carnegie Mellon

- Want Long Short-Term Memory (LSTM)? No problem!

$$
\begin{aligned}
\mathbf{i}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xi} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hi}) \\
\mathbf{f}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xf} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hf}) \\
\mathbf{o}_t &= \text{Sigmoid}(\mathbf{x}_t \cdot \mathbf{W}_{xo} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{ho}) \\
\mathbf{g}_t &= \tanh(\mathbf{x}_t \cdot \mathbf{W}_{xg} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hg}) \\
\mathbf{c}_t &= \mathbf{f}_t \otimes \mathbf{c}_{t-1} + \mathbf{i}_t \otimes \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \otimes \tanh(\mathbf{c}_t)
\end{aligned} \tag{2}
$$

tf_lstm.py

```python
def lstm(x, prev_c, prev_h, w_lstm):
  ifog = tf.matmul(tf.concat([x, prev_h], axis=2))
  i, f, o, g = tf.split(ifog, 4, axis=1)
  i, f, o, g = tf.sigmoid(i), tf.sigmoid(f), tf.sigmoid(o), tf.tanh(g)
  next_c = f * prev_c + i * g
  next_h = o * tf.tanh(next_c)
  return next_c, next_h
```

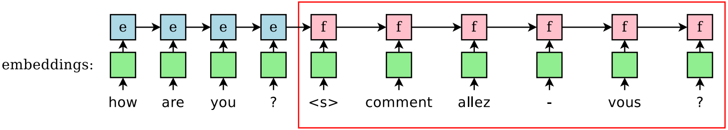
Changing The "Footprint" of `body`

Carnegie Mellon

tf_seq2seq_encoder.py

```python
def lstm(x, prev_c, prev_h, w_lstm): # previous slide
def encoder(sent, vocab_size=10000, hidden_size=128):
  with tf.variable_scope("encoder"):
    w_emb = tf.get_variable("w_emb", [vocab_size, hidden_size])
    w_lstm = tf.get_variable("w_lstm", [2 * hidden_size, 4 * hidden_size])

  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])
  def body(i, sent, prev_c, prev_h, hidden_states):
    emb = tf.nn.embedding_lookup(w_emb, sent[:, i])
    next_c, next_h = lstm(emb, prev_c, prev_h, w_lstm)
    hidden_states = hidden_states.write(i, next_h)
    return i+1, sent, next_c, next_h, hidden_states

  # create loop_vars, calling tf.while\_loop, stack, transpose, return
```

Coding the Decoder with tf.while_loop

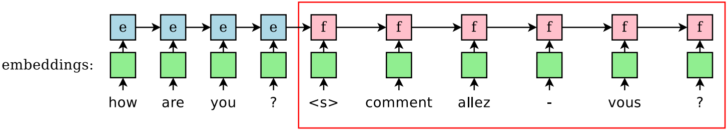


tf_seq2seq_decoder.py

```
def build_tf_graph:
  # en_sent, fr_sent: Tensors with unknown shapes [1, E], [1, F].
  en_sent, fr_sent = read_data()

  # encoder_states: [1, E, hidden_size]
  encoder_states = encoder(en_sent)

  # decoder_states: [1, F, hidden_size]
  decoder_states = decoder(encoder_states, fr_sent)
```

- Idea: almost the same with Encoder...

Handle encoder_states from encoder



tf_seq2seq_decoder.py

```python
def decoder(sent, encoder_states, vocab_size=10000, hidden_size=128):
  # sent: Tensor of size [1, F]
  with tf.variable_scope("decoder"):
    w_emb = tf.get_variable("w_emb", [vocab_size, hidden_size])
    w_rnn = tf.get_variable("w_rnn", [hidden_size, hidden_size])
    w_inp = tf.get_variable("w_inp", [hidden_size, hidden_size])

  def condition(i, sent, *args): return tf.less(i, tf.shape(E)[-1])
  def body(i, sent, prev_state, hidden_states):
    emb = tf.nn.embedding_lookup(w_emb, sent[:, i])
    next_state = tf.tanh(tf.matmul(prev_state, w_rnn) + tf.matmul(emb, w_inp))
    hidden_states = hidden_states.write(i, next_state)
    return i+1, sent, next_state, hidden_states

  # In encoder: tf.zeros([1, hidden_size]). Now: prev_state
  loop_vars = [tf.constant(0, dtype=tf.int32), sent, encoder_state, hidden_states]
```

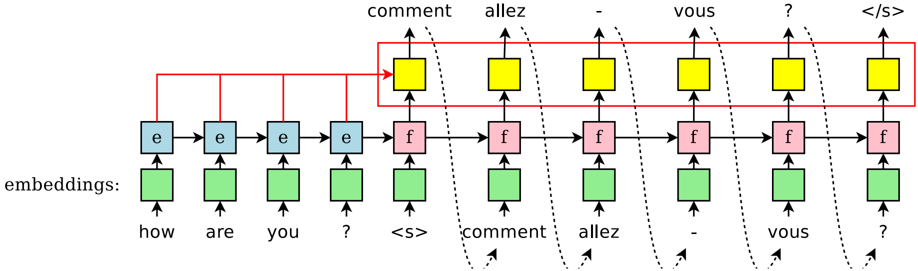
Dealing with hidden_states

Carnegie Mellon

tf_seq2seq_decoder.py

```
def decoder(sent, encoder_states, vocab_size=10000, hidden_size=128):
  # sent: [1, F]. define other variables, define while_loop, etc.
  loop_outputs = tf.while_loop(condition, body, loop_vars)

  hidden_states = loop_outputs[-1].stack()                # [F, 1, hidden_size]
  hidden_states = tf.transpose(hidden_states, [1, 0, 2])  # [1, F, hidden_size]

  return hidden_states

def build_tf_graph():
  # calling read_data, calling encoder, etc.
  decoder_states = decoder(encoder_states, fr_sent[1, :-1])

  # matmul and softmax
  with tf.variable_scope("softmax"):
    w_soft = tf.get_variable("w_soft", [hidden_size, vocab_size])
  decoder_states = tf.reshape(decoder_states, [-1, hidden_size])

  logits = tf.matmul(decoder_states, w_soft)  # [1 * F, vocab_size]
  logits = tf.reshape(logits, [1, tf.shape(sent)[1], vocab_size])
  labels = tf.reshape(fr_sent[:, 1:], [-1])

  loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
```

Attention?



- **Attention:** how is $\mathbf{a}(\mathbf{f}, \mathbf{e}_{1 \ldots |\mathbf{x}|})$ computed?

$$\alpha_i = g(\mathbf{f}, \mathbf{e}_i); \; a_i = \mathrm{Softmax}(\alpha_{1 \ldots |\mathbf{x}|}); \; \mathbf{a}(\mathbf{f}, \mathbf{e}_{1 \ldots |\mathbf{x}|}) = \sum_{i=1}^{|\mathbf{x}|} a_i \mathbf{e}_i \qquad (3)$$

  - Luong attention: $g(\mathbf{f}, \mathbf{e}_i) = \mathbf{f} \cdot \mathbf{e}_i^{\top}$. Quite easy to implement.

Attention?

tf_seq2seq_decoder.py

```python
def build_tf_graph():
  encoder_states = encoder(en_sent)
  decoder_states = decoder(encoder_states, fr_sent[1, :-1])
  attn_states = attention(encoder_states, decoder_states)
```

- **Attention:** how is $\mathbf{a}(\mathbf{f}, \mathbf{e}_{1 \dots |\mathbf{x}|})$ computed?

$$\alpha_i = g(\mathbf{f}, \mathbf{e}_i);\ a_i = \text{Softmax}(\alpha_{1 \dots |\mathbf{x}|});\ \mathbf{a}(\mathbf{f}, \mathbf{e}_{1 \dots |\mathbf{x}|}) = \sum_{i=1}^{|\mathbf{x}|} a_i \mathbf{e}_i \quad (4)$$

  - Luong attention: $g(\mathbf{f}, \mathbf{e}_i) = \mathbf{f} \cdot \mathbf{e}_i^\top$. Quite easy to implement.
  - Bahdanau attention: $g(\mathbf{f}, \mathbf{e}_i) = \tanh\left(\mathbf{f} \cdot \mathbf{w}_f + \mathbf{e}_i \cdot \mathbf{w}_e\right) \cdot \mathbf{v}$, where $\mathbf{w}_f, \mathbf{w}_e \in \mathbf{R}^{H \times H}$ and $\mathbf{v} \in \mathbb{R}^{H \times 1}$ are trainable parameters

Coding Attention

Carnegie Mellon

tf_seq2seq_decoder.py

```python
def attention(enc_states, dec_states):
  # enc_states, dec_states: [N, E, hidden_size], [N, F, hidden_size]
  enc_len = tf.shape(enc_states)[1]
  dec_len = tf.shape(dec_states)[1]
  hidden_size = tf.shape(enc_states)[-1]

  # auto batch matmul. attn_logits: [N, F, E]
  attn_logits = tf.matmul(dec_states, enc_states, transpose_b=True)
  attn_logits = tf.reshape(attn_logits, [-1, E])
  attn_weights = tf.nn.softmax(attn_logits)

  # weighted sum
  attn_weights = tf.reshape(attn_weights, [-1, dec_len, enc_len])  # [N, F, E]
  attn_outputs = tf.matmul(attn_weights, enc_states)
  return attn_outputs

def build_tf_graph():
  encoder_states = encoder(en_sent)
  decoder_states = decoder(encoder_states, fr_sent[1, :-1])
  attn_states = attention(encoder_states, decoder_states)
```