Abstract Syntax Networks for Code Generation and Semantic Parsing
Maxim Rabinovich, Mitchell Stern, Dan Klein

Presented by Patrick Crain
Background

• The Problem
  – Semantic parsing is structured, but asynchronous
  – Output *must* be well-formed → diverges from input

• Prior Solutions
  – S2S models [Dong & Lapata, 2016; Ling et al., 2016]
  – Encoder-decoder framework
  – Models don't consider output structure constraints
    • e.g., well-formedness, well-typedness, executability
Semantic Parsing

Show me the fare from ci0 to ci1

\[
\text{lambda } \ 0 \ e \\
( \exists \ 1 \ ( \text{and} \ ( \text{from} \ 1 \ ci0 ) \\
\ ( \text{to} \ 1 \ ci1 ) \\
\ ( \ = \ ( \text{fare} \ 1 ) \ 0 ) ) )
\]
class DireWolfAlpha(MinionCard):
    def __init__(self):
        super().__init__(
            "Dire Wolf Alpha", 2, CHARACTER_CLASS.ALL,
            CARD_RARITY.COMMON, minion_type=MINION_TYPE.BEAST)
    def create_minion(self, player):
        return Minion(2, 2, auras=[
            Aura(ChangeAttack(1), MinionSelector(Adjacent()))
        ])

name: ['D', 'i', 'r', 'e', '', 'W', 'o', 'l', 'f', 'A', 'l', 'p', 'h', 'a']
cost: ['2']
type: ['Minion']
rarity: ['Common']
race: ['Beast']
class: ['Neutral']
description: ['Adjacent', 'minions', 'have', '+', '1', 'Attack', '.']
health: ['2']
attack: ['2']
durability: ['1']
Abstract Syntax Networks

- Extends encoder-decoder framework
- Use ASTs to enforce output well-formedness
- Decoder is modular; submodels natively generate ASTs in a top-down manner
- Structure of ASTs mirrors input’s call graph
- Decoder input has both a fixed encoding and an attention-based representation
Related Work

• Encoder-decoder architectures
  – Machine translation (sequence prediction)
  – Constituency parsing (tree prediction)
    • Flattened output tree [Vinyals et al., 2015]
    • Construction decisions [Cross & Huang, 2016; Dyer et al., 2016]
  – ASNs created with recursive top-down generation → keeps tree structure of output
Related Work (cont.)

- **Neural modeling of code** [Allamanis et al., 2015; Maddison & Tarlow, 2014]
  - Neural language model + CSTs
  - Used for snippet retrieval
- **Grammar-based variational autoencoder for top-down generation** [Shin et. al., 2017]
- **Program induction from IO pairs** [Balog et al., 2016; Liang et al., 2010; Menon et al., 2013]
Structure of ASTs

• Code fragments $\rightarrow$ trees with typed nodes
• Primitive types (integers, identifiers)
  – Typed nodes with a value of that type (atomic)
• Composite types (expressions, statements)
  – Typed nodes with one of the type's constructors
  – Constructors specify the language constructs nodes represent, including children and their cardinalities

• ASTs can represent semantic parsing grammars
Input Representation

- Collections of named components, each consisting of a sequence of tokens
- Semantic parsing: single component containing the query sentence
- HEARTHSTONE: name and description are sequences of characters & tokens; attributes are single-token sequences
Model Details

• Decoder: collection of mutually recursive modules
  – Structure of modules mirrors AST being generated
  – Vertical LSTM stores info throughout decoding process
  – More on modules shortly

• Encoder: bi-LSTMs for embedding components
  – Final forward / backward encodings are concatenated
  – Linear projection is applied to encode entire input for decoder initialization
Attention

• Attention solves the need to encode arbitrary-length data with fixed-length vectors
• Idea: keep the encoder's intermediate outputs so we can relate input items to output items
  – Compute each input token’s raw attention score using its encoding & the decoder's current state:
    \[ q_t^{\text{raw}} = e_t^\top W x \]
  – Compute a separate attention score for each input component:
    \[ q_c^{\text{comp}} = w_c^\top x \]
Attention (cont.)

• Sum raw token- and component-level scores to get final token-level scores: \( q_t = q_t^{\text{raw}} + q_{c(t)}^{\text{comp}} \)

• Obtain attention vector using a softmax over the token-level attention scores: \( a = \text{softmax} \,(q) \)

• Multiply each token's encoding by its attention vector and sum the results to get an attention-based context vector: \( c = \sum_t a_t e_t \)

• Supervised attention: concentrate attention on a subset of tokens for each node
Primitive Type Module

- Each type has a module for selecting an appropriate value from the type's domain.
- Values generated from a closed list by applying softmax to vertical LSTM's state:
  \[
  \rho(y \mid T, v) = \left[\text{softmax} \left( f_T(v) \right) \right]_y
  \]
- String types may be generated using either a closed list or a char-level LSTM.
Composite Type Module

• Each composite type has a module for selecting among its constructors

• Constructors are selected using the vertical LSTM’s state as input & applying a softmax to a feedforward net’s output:

\[ p(C | T, v) = \left[ \text{softmax} \left( f_T(v) \right) \right]_c \]
Constructor Module

- Each constructor has a module for computing an intermediate LSTM state for each of its fields.
- Concatenate an embedding of each field with an attention vector and use a feedforward net to obtain a context-dependent field embedding:
  \[
  \tilde{e}_F = f_c(e_F, c)
  \]
- Compute an intermediate state in the vertical LSTM for the current field:
  \[
  v_{u,F} = \text{LSTM}^v(v_u, \tilde{e}_F)
  \]
Constructor Field Module

• Each constructor field has a module to determine the number of children associated with it, and to propagate the state of the vertical LSTM to them

• *Singular*: forward LSTM state unchanged:

\[ \mathbf{v}_w = \mathbf{v}_{u,F} \]

• *Optional*: use a feedforward network on the vertical LSTM state \( \rightarrow \) apply a sigmoid function to determine the probability of generating a child:

\[ p(z_F = 1 \mid \mathbf{v}_{u,F}) = \text{sigmoid} \left( f_{F}^{\text{gen}} (\mathbf{v}_{u,F}) \right) \]
Constructor Field Module (cont.)

- **Sequential**: use a decision LSTM to iteratively decide whether to generate a new child; after a "yes", update a state LSTM with the new context-dependent embedding

\[
p(z_{F,i} = 1 \mid s_{F,i−1}, v_{u,F}) = \text{sigmoid}(f_{F}^{\text{gen}}(s_{F,i−1}, v_{u,F}))
\]

**Context Update**
\[
\tilde{e}_{F,i} = f_{F}^{\text{update}}(v_{u,F}, s_{u,i−1}, c_{F,i})
\]

**Vertical Update**
\[
v_{w_i} = \text{LSTM}^v(v_{u,F}, \tilde{e}_{F,i})
\]

**Horizontal Update**
\[
s_{u,i} = \text{LSTM}^h(s_{u,i−1}, \tilde{e}_{F,i})
\]
Evaluation

• Semantic Parsing:
  - Uses query → logical representation pairs
  - Lowercase, stemmed, abstract entity identifiers
  - Accuracies computed with tree exact match

• Code Generation (HEARTHSTONE):
  - Uses card text → code implementation pairs
  - Accuracies computed with exact match & BLEU
Results – Semantic Parsing

<table>
<thead>
<tr>
<th></th>
<th>ATIS</th>
<th></th>
<th>GEO</th>
<th></th>
<th>JOBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td>System</td>
<td></td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>ZH15</td>
<td>84.2</td>
<td>ZH15</td>
<td>88.9</td>
<td>ZH15</td>
<td>85.0</td>
</tr>
<tr>
<td>ZC07</td>
<td>84.6</td>
<td>KCAZ13</td>
<td>89.0</td>
<td>PEK03</td>
<td>88.0</td>
</tr>
<tr>
<td>WKZ14</td>
<td>91.3</td>
<td>WKZ14</td>
<td>90.4</td>
<td>LJK13</td>
<td>90.7</td>
</tr>
<tr>
<td>DL16</td>
<td>84.6</td>
<td>DL16</td>
<td>87.1</td>
<td>DL16</td>
<td>90.0</td>
</tr>
<tr>
<td>ASN</td>
<td>85.3</td>
<td>ASN</td>
<td>85.7</td>
<td>ASN</td>
<td>91.4</td>
</tr>
<tr>
<td>+ SUPATT</td>
<td>85.9</td>
<td>+ SUPATT</td>
<td>87.1</td>
<td>+ SUPATT</td>
<td>92.9</td>
</tr>
</tbody>
</table>

- JOBS: SotA accuracy, even without supervision
- ATIS and GEO: Falls short of SotA, but exceeds / matches [Dong & Lapata, 2016]
  - ASN don’t use typing information or rich lexicons
Results – Code Generation

- **HEARTHSTONE**: Significant improvement over initial results
  - Near perfect on simple cards; idiosyncratic errors on nested calls
  - Variable naming / control flow prediction are more challenging
- **Current metrics approximate functional equivalence**
  - Future metrics that canonicalize the code may be more effective
- **Enforcement of semantic coherence is an open challenge**

<table>
<thead>
<tr>
<th>System</th>
<th>Accuracy</th>
<th>BLEU</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest</td>
<td>3.0</td>
<td>65.0</td>
<td>65.7</td>
</tr>
<tr>
<td>LPN</td>
<td>6.1</td>
<td>67.1</td>
<td>–</td>
</tr>
<tr>
<td>ASN</td>
<td>18.2</td>
<td>77.6</td>
<td>72.4</td>
</tr>
<tr>
<td>+ SupAtt</td>
<td>22.7</td>
<td>79.2</td>
<td>75.6</td>
</tr>
</tbody>
</table>
Conclusion

• ASNs are very effective for ML tasks that transform partially unstructured input into well-structured output
• Recursive decomposition in particular helps by ensuring the decoding process mirrors the structure of the output
• ASNs attained SotA accuracies on JOBS / HEARTHSONE; supervised attention vectors led to further improvements
• ASNs could not match SotA accuracies on ATIS or GEO due to lack of sufficient typing information or lexicons
• Overcoming more challenging tasks, evaluation issues, and modeling issues remain open problems