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

```
lambda $0 e
  ( exists $1 ( and ( from $1 ci0 )
                    ( to $1 ci1 )
                    ( = ( fare $1 ) $0 ) ) )
```

Code Generation


```
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']
```

```
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()))
    ])
```

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 → 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:

$$q_t^{\text{raw}} = \mathbf{e}_t^\top \mathbf{W}\mathbf{x}$$

  - Compute a separate attention score for each input component:

$$q_c^{\text{comp}} = \mathbf{w}_c^\top \mathbf{x}$$

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: $\mathbf{a} = \text{softmax}\left(\mathbf{q}\right)$
- Multiply each token's encoding by its attention vector and sum the results to get an attention-based context vector: $\mathbf{c} = \sum_t a_t \mathbf{e}_t$
- 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:

$$p\left(\mathrm{y} \mid \mathtt{T}, \mathbf{v}\right) = \left[\mathrm{softmax}\left(f_{\mathtt{T}}\left(\mathbf{v}\right)\right)\right]_{\mathrm{y}}$$

- 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:

$$p\left(\mathtt{C} \mid \mathtt{T}, \mathbf{v}\right) = \left[\mathrm{softmax}\left(f_{\mathtt{T}}\left(\mathbf{v}\right)\right)\right]_{\mathtt{C}}$$

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:

$$\tilde{\mathbf{e}}_{\mathtt{F}} = f_{\mathtt{C}}\left(\mathbf{e}_{\mathtt{F}},\ \mathbf{c}\right)$$

- Compute an intermediate state in the vertical LSTM for the current field:

$$\mathbf{v}_{u,\mathtt{F}} = \mathrm{LSTM}^{v}\left(\mathbf{v}_{u},\ \tilde{\mathbf{e}}_{\mathtt{F}}\right)$$

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:

$$\mathbf{v}_w = \mathbf{v}_{u,\mathtt{F}}$$

- *Optional*: use a feedforward network on the vertical LSTM state → apply a sigmoid function to determine the probability of generating a child:

$$p(z_{\mathtt{F}} = 1 \mid \mathbf{v}_{u,\mathtt{F}}) = \mathrm{sigmoid}\left(f_{\mathtt{F}}^{\mathrm{gen}}(\mathbf{v}_{u,\mathtt{F}})\right)$$

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

Decision

$$p\left(z_{\mathtt{F},i} = 1 \mid \mathbf{s}_{\mathtt{F},i-1},\ \mathbf{v}_{u,\mathtt{F}}\right) = \mathrm{sigmoid}\left(f_{\mathtt{F}}^{\mathrm{gen}}\left(\mathbf{s}_{\mathtt{F},i-1},\ \mathbf{v}_{u,\mathtt{F}}\right)\right)$$

Vertical Update

$$\mathbf{v}_{w_i} = \mathrm{LSTM}^{v}(\mathbf{v}_{u,\mathtt{F}},\ \tilde{\mathbf{e}}_{\mathtt{F},i})$$

Context Update

$$\tilde{\mathbf{e}}_{\mathtt{F},i} = f_{\mathtt{F}}^{\mathrm{update}}(\mathbf{v}_{u,\mathtt{F}},\ \mathbf{s}_{u,i-1},\ \mathbf{c}_{\mathtt{F},i})$$

Horizontal Update

$$\mathbf{s}_{u,i} = \mathrm{LSTM}^{h}(\mathbf{s}_{u,i-1},\ \tilde{\mathbf{e}}_{\mathtt{F},i})$$

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

ATIS		GEO		JOBS	
System	Accuracy	System	Accuracy	System	Accuracy
ZH15	84.2	ZH15	88.9	ZH15	85.0
ZC07	84.6	KCAZ13	89.0	PEK03	88.0
WKZ14	**91.3**	WKZ14	**90.4**	LJK13	90.7
DL16	84.6	DL16	87.1	DL16	90.0
ASN	85.3	ASN	85.7	ASN	**91.4**
+ SUPATT	85.9	+ SUPATT	87.1	+ SUPATT	**92.9**

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

Results – Code Generation

System	Accuracy	BLEU	F1
NEAREST	3.0	65.0	65.7
LPN	6.1	67.1	–
ASN	**18.2**	**77.6**	**72.4**
+ SUPATT	**22.7**	**79.2**	**75.6**

- 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

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