Recall: Dependencies

- Dependency syntax: syntactic structure is defined by dependencies
- Head (parent, governor) connected to dependent (child, modifier)
- Each word has exactly one parent except for the ROOT symbol
- Dependencies must form a directed acyclic graph

Recall: Shift-Reduce Parsing

- State: Stack: [ROOT I ate] Buffer: [some spaghetti bolognese]
- Left-arc (reduce operation): Let $\sigma$ denote the stack
- “Pop two elements, add an arc, put them back on the stack”
  $\sigma | w_{-2}, w_{-1} \rightarrow \sigma | w_{-1}$, $w_{-2}$ is now a child of $w_{-1}$
- Train a classifier to make (shift, left-arc, right-arc) decisions sequentially — that classifier can parse sentences for you
### Where are we now?
- Classification, then sequences, then trees
- Now we can understand sentences in terms of tree structures as well
- Why is this useful? What does this allow us to do?
- We’re going to see how parsing can be a stepping stone towards more formal representations of language meaning. We’ll contrast with these approaches when we revisit the same problems later with neural nets.

### Today
- Montague semantics:
  - Model theoretic semantics
  - Compositional semantics with first-order logic
- CCG parsing for database queries
- Seq2seq semantic parsing

---

### Model Theoretic Semantics
- Key idea: can ground out natural language expressions in set-theoretic expressions called *models* of those sentences
- Natural language statement $S \Rightarrow$ interpretation of $S$ that models it
  - *She likes going to that restaurant*
- Interpretation: defines who *she* and *that restaurant* are, make it able to be concretely evaluated with respect to a *world*
- Entailment (statement $A$ implies statement $B$) reduces to: in all worlds where $A$ is true, $B$ is true
- Our modeling language is *first-order logic*
First-order Logic

- Powerful logic formalism including things like entities, relations, and quantifications
  
  Lady Gaga sings

- `sings` is a *predicate* (with one argument), function `f: entity \rightarrow true/false`

- `sings(Lady Gaga) = true or false`, have to execute this against some database *(world)*

- Quantification: “for all” operator, “there exists” operator
  
  `\forall x \text{sings}(x) \lor \text{dances}(x) \rightarrow \text{performs}(x)`

  “Everyone who sings or dances performs”

Montague Semantics

- Sentence expresses something about the world which is either true or false

- Denotation: evaluation of some expression against this database

  `[[ Lady Gaga ]] = e470  [[ \text{sings}(e470) ]] = True`

  denotation of this string is an entity

  denotation of this expression is T/F

Montague Semantics

- Function application: apply this to `e470`

  `\lambda y. \text{sings}(y)`

- We can use the syntactic parse as a bridge to the lambda-calculus representation, build up a logical form (our model) *compositionally*

Parses to Logical Forms

- General rules:
  
  `VP: \lambda y. a(y) \land b(y) \rightarrow VP: \lambda y. a(y) CC VP: \lambda y. b(y)`

  `S: f(x) \rightarrow NP: x VP: f`
Parses to Logical Forms

- Function takes two arguments: first x (date), then y (entity)
- How to handle tense: should we indicate that this happened in the past?

Tricky things

- Adverbs/temporality: *Lady Gaga sang well yesterday*
  \[\text{sings(Lady Gaga, time=yesterday, manner=well)}\]
- "Neo-Davidsonian" view of events: things with many properties:
  \[\exists e. \text{type}(e,\text{sing}) \land \text{agent}(e,e470) \land \text{manner}(e,\text{well}) \land \text{time}(e,...)\]
- Quantification: *Everyone is friends with someone*
  \[\exists y \forall x \text{friend}(x,y) \quad \forall x \exists y \text{friend}(x,y)\]
  - (one friend)
  - (different friends)
- Same syntactic parse for both! So syntax doesn’t resolve all ambiguities
- Indefinite: *Amy ate a waffle*
  \[\exists w. \text{waffle}(w) \land \text{ate}(Amy,w)\]
- Generic: *Cats eat mice* (all cats eat mice? most cats? some cats?)

Semantic Parsing

- For question answering, syntactic parsing doesn’t tell you everything you want to know, but indicates the right structure
- Solution: *semantic parsing*: many forms of this task depending on semantic formalisms
- CCG parsers can produce these kinds of expressions, which can be used for database querying/question answering

CCG Parsing
Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Parallel derivations of syntactic parse and lambda calculus expression
- Syntactic categories (for this lecture): S, NP, “slash” categories
- S\NP: “if I combine with an NP on my left side, I form a sentence” — verb
- When you apply this, there has to be a parallel instance of function application on the semantics side

\[ S \text{sings}(e728) \]
\[ \lambda y. \text{sings}(y) \]
\[ \lambda x. \text{borders}(y, x) \]

Combinatory Categorial Grammar

- Steedman+Szabolcsi 1980s: formalism bridging syntax and semantics
- Syntactic categories (for this lecture): S, NP, “slash” categories
- S\NP: “if I combine with an NP on my left side, I form a sentence” — verb
- (S\NP)\NP: “I need an NP on my right and then on my left” — verb with a direct object

\[ S \text{sings}(e728) \]
\[ \lambda x. \text{borders}(y,e89) \]

CCG Parsing

- What is a very complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (border Texas)

\[ \frac{(S/(S\NP))/N}{\text{states}} \]
\[ \frac{\lambda f. \lambda g. \lambda x. f(x) \land g(x)}{\lambda x. \text{state}(x)} \]

\[ \lambda x. \text{borders}(y,x) \]

“What” is a very complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (border Texas)

Lexicon is highly ambiguous — all the challenge of CCG parsing is in picking the right lexicon entries

Zettlemoyer and Collins (2005)
CCG Parsing

Show me flights to Prague

<table>
<thead>
<tr>
<th>S/N</th>
<th>λf.f</th>
<th>N</th>
<th>λx. flight(x)</th>
<th>λy. λf. λx.(f(x)) ∧ to(x,y)</th>
<th>NP</th>
<th>PRG</th>
</tr>
</thead>
</table>

What states border Texas

λx. state(x) ∧ borders(x, e89)

Problem: we don’t know the derivation

Texas corresponds to NP | e89 in the logical form (easy to figure out)

What corresponds to (S/(S\NP))/N | λf. λg. λx. f(x) ∧ g(x)

How do we infer that without being told it?

Building these parsers is very hard...let’s try a different approach!

Slide credit: Dan Klein

Building CCG Parsers

Training data looks like pairs of sentences and logical forms

What states border Texas

λx. state(x) ∧ borders(x, e89)

Problem: we don’t know the derivation

Texas corresponds to NP | e89 in the logical form (easy to figure out)

What corresponds to (S/(S\NP))/N | λf. λg. λx. f(x) ∧ g(x)

How do we infer that without being told it?

Building these parsers is very hard...let’s try a different approach!

Zettlemoyer and Collins (2005)

Encoder-Decoder

Can view many tasks as mapping from an input sequence of tokens to an output sequence of tokens

Semantic parsing:

What states border Texas → λx. state(x) ∧ borders(x, e89)

Syntactic parsing

The dog ran → (S (NP (DT the) (NN dog)) (VP (VBD ran)))

(but what if we produce an invalid tree or one with different words?) 😐

Machine translation, summarization, dialogue can all be viewed in this framework as well — our examples will be MT for now.
Encoder-Decoder

- Encode a sequence into a fixed-sized vector
  
  
  \[ \text{le film était bon [STOP]} \]

- Now use that vector to produce a series of tokens as output from a separate LSTM decoder

Sutskever et al. (2014)

Model

- Generate next word conditioned on previous word as well as hidden state

  \[ P(y_1|x, y_1, \ldots, y_{i-1}) = \text{softmax}(W h) \]

  \[ P(y|x) = \prod_{i=1}^{n} P(y_i|x, y_1, \ldots, y_{i-1}) \]

  Decoder has separate parameters from encoder, so this can learn to be a language model (produce a plausible next word given current one)

Inference

- Generate next word conditioned on previous word as well as hidden state

  \[ \text{le film était bon [STOP]} \]

- During inference: need to compute the argmax over the word predictions and then feed that to the next RNN state

- Need to actually evaluate computation graph up to this point to form input for the next state

- Decoder is advanced one state at a time until [STOP] is reached
Implementing Encoder-Decoder Models

- Encoder: consumes sequence of tokens, produces a vector. Analogous to encoders for classification/tagging tasks.
- Decoder: separate module, single cell. Takes two inputs: hidden state (vector $h$ or tuple $(h, c)$) and previous token. Outputs token + new state.

Training

- Objective: maximize $\sum_{(x,y)} \sum_{i=1}^n \log P(y_i^* | x, y_1^*, \ldots, y_{i-1}^*)$
- One loss term for each target-sentence word, feed the correct word regardless of model’s prediction (called “teacher forcing”).

Training: Scheduled Sampling

- Model needs to do the right thing even with its own predictions.
- Scheduled sampling: with probability $p$, take the gold as input, else take the model’s prediction.
- Starting with $p = 1$ (teacher forcing) and decaying it works best.
- “Right” thing: train with reinforcement learning.
  Bengio et al. (2015)
Implementations Details

- Sentence lengths vary for both encoder and decoder:
  - Typically pad everything to the right length and use a mask or indexing to access a subset of terms
  - Encoder: looks like what you did in Mini 2
  - Decoder: execute one step of computation at a time, so computation graph is formulated as taking one input + hidden state
- Test time: do this until you generate the stop token
- Training: do this until you reach the gold stopping point

Implementations Details (cont’d)

- Batching is pretty tricky: decoder is across time steps, so you probably want your label vectors to look like [num timesteps x batch size x num labels], iterate upwards by time steps
- Beam search: can help with lookahead. Finds the (approximate) highest scoring sequence:
  \[ \arg\max_y \prod_{i=1}^{n} P(y_i|x, y_1, \ldots, y_{i-1}) \]

Beam Search

- Maintain decoder state, token history in beam
- Keep both film states! Hidden state vectors are different

Other Architectures

- What’s the basic abstraction here?
  - Encoder: sentence -> vector
  - Decoder: hidden state, output prefix -> new hidden state, new output
    - OR: sentence, output prefix -> new output (more general)
- Wide variety of models can apply here: CNN encoders, decoders can be any autoregressive model including certain types of CNNs
- Transformer: another model discussed next lecture
Takeaways

- Can represent meaning with first order logic and lambda calculus
- Can bridge syntax and semantics and create semantic parsers that can interpret language into lambda-calculus expressions
- seq2seq models provide an easier way to do this
- Next time: continue seq2seq semantic parsing, discuss attention