Chapter 8

Dialogue

The remainder of the course (we're trying something a little different this year) is about the problem of natural language *dialogue*. We'll start with a brief overview of the topic and then the various parts of a typical task-oriented dialogue[1] system will motivate the rest of the topics of the course.

Dialogue systems can be divided broadly into two different types: chatbots and task-oriented dialogue systems. Chatbots are supposed to be able to have a conversation about anything, whereas task-oriented dialogue systems are supposed to help users complete a specific task or tasks. Siri, Alexa, Cortana, and Google Assistant are task-oriented dialogue systems.

8.1 Chatbots

Turing's imitation game. The idea of a chatbot goes back (at least) to Turing's paper on artificial intelligence (Turing, 1950), in which he proposed to redefine whether a computer is intelligent by whether it can carry on a conversation well enough to be indistinguishable from a human. Turing predicted that in 50 years (that is, in 2000), computers would be able to store 10^9 bits(?) and pass a five-minute test with 70% success.

ELIZA. The oldest chatbot (that I'm aware of) was ELIZA (Weizenbaum, 1966). It used the pretense of being a Rogerian psychotherapist to turn the user's utterances back into questions, rather than trying to contribute new ideas to the conversation. The original source code was rediscovered only very recently,[2] but there are many approximate reimplementations; in any copy of Emacs, you can type M-x `doctor` (that is, Escape `x` `doctor` Return) to run one version.

Here's the conversation from the original paper (punctuation added):

P: Men are all alike.

E: In what way?

P: They're always bugging us about something or other.

[1] A *dialogue* is a conversation between two people, or a person and a machine; a *dialog* box is a window that pops up in a GUI that provides information and asks for a response.

[2] `https://elizagen.org`

E: Can you think of a specific example?

P: Well, my boyfriend made me come here.

E: Your boyfriend made you come here?

P: He says I'm depressed much of the time.

E: I am sorry to hear that you are depressed.

P: It's true. I am unhappy.

E: Do you think coming here will help you not to be unhappy?

P: I need some help, that much seems certain.

E: What would it mean to you if you got some help?

P: Perhaps I could learn to get along with my mother.

E: Tell me more about your family.

P: My mother takes care of me.

E: Who else in your family takes care of you?

P: My father.

E: Your father?

P: You are like my father in some ways.

E: What resemblance do you see?

P: You are not very aggressive but I think you don't want me to notice that.

E: What makes you think I am not very aggressive?

P: You don't argue with me.

E: Why do you think I don't argue with yoU?

P: You are afraid of me.

E: Does it please you to believe I am afaid of you?

P: My father is afraid of everybody.

E: What else comes to mind when you think of your father?

P: Bullies.

E: Does that have anything to do with the fact that your boyfriend made you come here?

ELIZA is extremely simple. Given a string w, for example, "You are very helpful," it performs the following steps:

1. Find the most important keyword, K, in the string. In the example, K is "you".
2. Find a template D for K that matches the string. In the example, D is "X you are Y," so $X = \epsilon$ and Y = very helpful.
3. Choose a response R for D. For example, one response for "X you are Y" is, "What makes you think I am Y?"

4. In X, Y, etc., change pronouns "me" to "you" and "you" to "me," etc., and substitute them into R. So the example R becomes"What makes you think I am very helpful?".

Additionally, there are certain keywords, like "my," that make ELIZA save a response for later. If it later has nothing to say, it will generate a saved response (like the last line of the above dialogue).

Essentially, what passes for syntax/semantics are templates like "X you are Y" and "What makes you think I am Y?". Although this is a far cry from the parse trees and logical forms we saw in earlier chapters, these templates get the job done, and we will see that they are still being used today.

A simple neural chatbot. A modern, but also simple, chatbot was based on an early neural machine translation system (Vinyals and Le, 2015). The model was just an encoder RNN whose final state was used to initialize the state of a decoder RNN (that is, there was no attention mechanism). It was trained to "translate" a line spoken by one movie character to a response from another movie character, or a line spoken by a customer to a response from a helpdesk. Given how simple the model was, some of the results were surprisingly good:

Human: what is the purpose of life ?

Machine: to serve the greater good .

Human: what is the purpose of living ?

Machine: to live forever .

Human: what is the purpose of existence ?

Machine: to find out what happens when we get to the planet earth .

Human: where are you now ?

Machine: i 'm in the middle of nowhere .

Human: what is the purpose of dying ?

Machine: to have a life .

Human: what is the purpose of being intelligent ?

Machine: to find out what it is .

Human: what is the purpose of emotions ?

Machine: i don 't know .

8.2 Task-Oriented Dialogue

In this section I'm relying heavily on Chapter 24 of *SLP* (Jurafsky and Martin, 2021).

The reason I've dwelt so much on ELIZA is that many of the elements of how ELIZA works are still part of the task-oriented dialogue systems that we use today. GUS (Bobrow et al., 1977) was another early example.

Dialogue act classification.. The first thing the computer does is to determine what kind of *dialogue act* the user input is. in the MultiWOZ data set (Budzianowski et al., 2018), for example, some possible dialogue acts are:

- `book_hotel`
- `book_restaurant`
- `book_train`
- `find_attraction`
- `find_hospital`
- `find_hotel`
- `find_restaurant`
- `find_taxi`
- `find_train`

If the user asks, "i need a place to dine in the center thats expensive," that's a `find_restaurant`.

Recall that in ELIZA, the first step was to identify the most important keyword; this corresponds (very) roughly to dialogue act classification.

Slot filling. In ELIZA, each keyword was associated with one or more patterns, and ELIZA would try to match the user input against each one. Similarly, in the task-oriented dialogue system, each dialogue act corresponds to one or more *frames* that contain *slots*. For example, a frame for `find_restaurant` might have slots for

- `restaurant-area`
- `restaurant-food`
- `restaurant-name`
- `restaurant-pricerange`

For the user input "i need a place to dine in the centre thats expensive," the slot fillers would be `restaurant-area` = centre, `restaurant-pricerange` = expensive.

In both cases, the frame (`find_restaurant`(`restaurant-area` = centre, `restaurant-pricerange` = expensive)) is what serves as both syntax and semantics. It's a *lot* simpler than we we studied in past weeks of the course, but it gets the job done!

Dialogue state tracking. The system maintains a *dialogue state* that persists from turn to turn. In ELIZA, the dialogue state took the form of a "memory" that stored X whenever the user says "my X." In a task-oriented dialogue system, the state might, for example, contain a partial frame. If the frame is incomplete, the computer will request more information from the user until the frame is complete.

Dialogue policy and content planning. The final step for ELIZA was to choose a response pattern (by cycling through a little of possible responses) and copying the slot fillers from the user input to the response slots (changing pronouns as needed).

A task-oriented dialogue does the same thing, using the current dialogue state to choose a dialogue act for the response and to fill the slots in the response. For example, the dialogue act might be `recommend_restaurant` and the filler might be `restaurant-name` = Bedouin. Of course, ELIZA's strategy of copying slot fillers from the user input to the response won't work very well, since the point of the system is to provide new information to the user. We're going to assume that this is taken care of by a database system and won't say any more about this step!

Sentence realization. Finally, the computer has to translate the response frame, e.g., `recommend_restaurant(restaurant-name` = Bedouin), into text, like "There is a place named Bedouin in the centre. How does that sound?"

(In ELIZA, this step was trivial because the frames are already literal text; or, you could consider the mapping of pronouns to be the sentence realization step.)

In the following sections, we will talk about dialogue act classification (and other classification tasks), slot filling (and other sequence labeling tasks), and finally generation.

8.3 Slot Filling (Sequence Labeling)

8.3.1 Problem

In slot-filling, the input is a sentence like

> Please find me a train from cambridge to stansted airport

and the output is

> Please find me a train from [cambridge]$_{\text{departure}}$ to [stansted airport]$_{\text{destination}}$

The most common way to formulate this kind of problem, where the computer has to identify a number of non-overlapping substrings of the input, is called *BIO tagging*.

O	O	O	O	O	O	B-departure	O	B-destination	I-destination
Please	find	me	a	train	from	cambridge	to	stansted	airport

B stands for "begin" and is used for the first word in each slot-filler; I stands for "inside" and is used for the second and subsequent words in each slot-filler. O stands for "outside" and is used for any word that does not belong to a slot-filler. Other schemes exist, like BILOU (L for the last word an entity, U 'unit' for the only word in an entity), but this is the simplest and most common.

Now we've reduced slot-filling to a *sequence labeling* task. Other examples of sequence labeling tasks are:

- Word segmentation: Given a representation of a sentence without any word boundaries, reconstruct the word boundaries. In some languages, like Chinese, words are written without any spaces in between them. (Indeed, it can be difficult to settle on the definition of a "word" in such languages.)
- Part of speech tagging: Given a sentence, label each word with its part of speech.
- Named entity detection/recognition: Given a sentence, identify all the proper names (Notre Dame, Apple, etc.) and classify them as persons, organizations, places, etc.

One of the hallmarks of sequence labeling problems is dependencies between the labels. For example, if we're doing named entity recognition, a model might learn that *Dame* has a high probability of being tagged I-org, as the last word of *Notre Dame* (and *University of Notre Dame*, *Cathedral of Notre Dame*, etc.). But in a sentence like

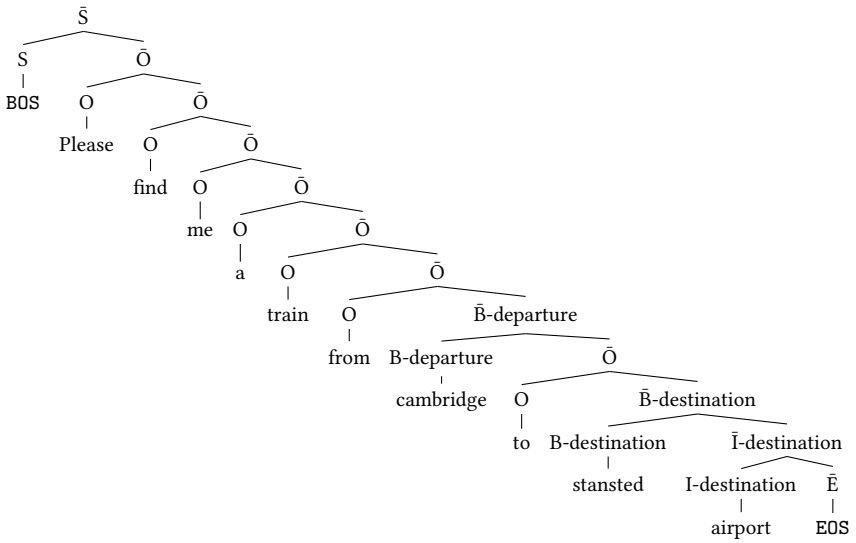
> In 2004, Dame Julie Andrews voiced Queen Lillian in *Shrek 2*,

Dame should be tagged B-person. Maybe the model can get enough clues from the surrounding words to tag it correctly, but the strongest clue should be that I-org absolutely cannot follow O.

8.3.2 Sequence labeling as parsing

The classic models used to solve sequence labeling problems are, in historical order, hidden Markov Models (HMMs), conditional random fields (CRFs), and biLSTM-CRFs. HMMs and CRFs are usually formulated as either finite automata or matrix operations. But since you have parsing with CFGs fresh in your mind, let's formulate them as CFGs. It may be overkill, but it's arguably the cleanest way to write them.

What would a parse tree look like for this task? The labels (B-*, I-*, O) would be like parts of speech, and since we don't have any other kind of structure, it only makes sense to use a purely right-branching or left-branching structure. Let's choose right-branching:



So the grammar has the following kinds of rules:

$\bar{X} \rightarrow X\ \bar{Y}$	X and Y are labels
$X \rightarrow a$	X is a label and a is a word (incl. `BOS`)
$\bar{\text{E}} \rightarrow \texttt{EOS}$	

The start symbol is $\bar{\text{S}}$ (not S). Let's call the first kind of rules *transition rules* and the last two kinds *emission rules*. (The reason we have to special-case the last rule is because it's the only emission rule whose left-hand side has a bar over it. Otherwise, it's not really that special.)

8.3.3 Hidden Markov models

If the grammar is a probabilistic CFG, then this is equivalent to a hidden Markov model. The probabilities of the transition rules $\bar{X} \rightarrow X\ \bar{Y}$ measure the probability of one label coming after another and are called *transition probabilities*. The emission rules, $X \rightarrow a$, measure the probability of generating a word given a label and are called *emission probabilities*.

8.3.4 Conditional random fields

If the grammar is a weighted (not necessarily probabilistic) CFG, then this is equivalent to a conditional random field. A rule can have any weight $p > 0$; we also call $\log p$ its *score*.

Note that in order for this CFG to be equivalent to a CRF, we have to include all rules of the form $X \rightarrow a$ and $\bar{X} \rightarrow X\ \bar{Y}$, even if they were not observed in the training data.

In the parsing chapter, we skipped straight from PCFGs to neural weighted CFGs, so we won't dwell on non-neural weighted CFGs any longer.

8.3.5 RNN+CRFs

In the parsing chapter and in HW3, we built a neural parser by using a neural network to compute the rule scores of a weighted CFG. We can do the exact same thing here, but with a slightly different neural network.

We start off, as usual, with a sequence encoder. Let $w = w_1 \cdots w_n$ be the input string, with $w_1 = \texttt{BOS}$ and $w_n = \texttt{EOS}$. Let Γ be the set of possible labels.

$$\begin{aligned} &\mathbf{V} \in \mathbb{R}^{n \times d} \\ &\mathbf{V}_i = \text{Embedding}^{\boxed{1}}(w_i) \qquad\qquad i = 1, \ldots, n \qquad (8.1) \\ &\mathbf{H} \in \mathbb{R}^{n \times d} \\ &\mathbf{H} = \text{RNN}^{\boxed{2}}(\mathbf{V}). \qquad (8.2) \end{aligned}$$

Usually the encoder is a fancier kind of RNN called a bidirectional LSTM, but we're sticking to a simple, left-to-right RNN here. Each $\mathbf{H}_i$ is the encoding of w_i. So far, this is the same as the neural parser from before.

Next, we need to define a function that assigns a score to every rule, possibly depending on its position in the string. We define this function for the three kinds of rules as follows:

$$s(\bar{X} \rightarrow {}_i\, X\ \bar{Y}\, {}_n) = \mathbf{T}_{X,Y} \qquad 0 \le i \le n-2 \qquad (8.3)$$

$$s(X \rightarrow {}_{i-1}\, a\, {}_i) = \mathbf{O}_{i,X} \qquad 1 \le i < n \qquad (8.4)$$

$$s(\bar{\text{E}} \rightarrow {}_{n-1}\, \texttt{EOS}\, {}_n) = \mathbf{O}_{i,\text{E}} \qquad (8.5)$$

where $\mathbf{T} \in \mathbb{R}^{|\Gamma| \times |\Gamma|}$ is a matrix of learnable parameters, so that every transition rule gets an independent score; and $\mathbf{O}$ is computed from the RNN encodings as

$$\begin{aligned} &\mathbf{O} \in \mathbb{R}^{n \times |\Gamma|} \\ &\mathbf{O}_i = \text{LinearLayer}^{\boxed{3}}(\mathbf{H}_i). \qquad (8.6) \end{aligned}$$

Now both training and labeling (= parsing) can be done exactly as before. But, this is an extremely inefficient way of implementing a RNN+CRF. Since the grammar includes all rules with the forms shown above, even if they were not observed in the training data, the grammar is quite large. In the next section, we'll see how to optimize this.

8.3.6 RNN+CRFs made more efficient

Recall that during training, we maximize

$$\begin{aligned} L &= \sum_{(w,\text{tree}) \in \text{data}} \log P(\text{tree} \mid w) \\ &= \sum_{(w,\text{tree}) \in \text{data}} \log \frac{\exp s(\text{tree})}{\sum_{\text{tree}'} \exp s(\text{tree}')} \\ &= \sum_{(w,\text{tree}) \in \text{data}} \left(s(\text{tree}) - \underbrace{\log \sum_{\text{tree}'} \exp s(\text{tree}')}_{\text{partition function}} \right) \end{aligned}$$

and the partition function is computed using a modified CKY algorithm. And during parsing, we use the CKY algorithm.

As a reminder, here's the algorithm, where we've plugged in the rule scores computed by the neural network. The symbol $\oplus$ is a generic operator that is max if we're looking for the best parse and logaddexp if we want the total score of all parses.

for all $0 \le i < j \le n$ **do**
  for all $X \in \Gamma$ **do**
    $chart[i, j][X] \leftarrow -\infty$
    $chart[i, j][\bar{X}] \leftarrow -\infty$
  end for
end for

▷ rules of the form $X \rightarrow w_i$
for all $i \leftarrow 1, \ldots, n-1$ **do**
  for all $X \in \Gamma$ **do**
    $chart[i-1, i][X] \leftarrow \mathbf{O}_{i,X}$
  end for
end for

▷ rule $\bar{\text{E}} \rightarrow \texttt{EOS}$
$chart[n-1, n][\bar{\text{E}}] \leftarrow \mathbf{O}_{n,\text{E}}$

▷ rules of the form $\bar{X} \rightarrow X\ \bar{Y}$
for $\ell \leftarrow 2, \ldots, n$ **do**
  for $i \leftarrow 0, \ldots, n-\ell$ **do**
    $j \leftarrow i + \ell$
    for $k \leftarrow i+1, \ldots, j-1$ **do**
      for all $X \in \Gamma$ **do**
        for all $Y \in \Gamma$ **do**
          $chart[i, j][\bar{X}] \leftarrow chart[i, j][\bar{X}] \oplus (\mathbf{T}_{X,Y} + chart[i, k][X] + chart[k, j][\bar{Y}])$
        end for
      end for
    end for
  end for
end for
return $chart[0, n][\bar{\text{S}}]$

Linear time. This is $O(n^3)$, but we would like to reduce this to $O(n)$. Remember that the cubic time complexity comes from the triple loop involving i, j, and k. But in the trees that our grammar generates, it's always the case that if $j - i > 1$, then $i \le n-2$, $k = i+1$, and $j = n$. We didn't even define the rule-scoring function for other values of i and j. So the above triple loop can be rewritten as a single loop:

for $i \leftarrow n-2, \ldots, 0$ **do**
  for all $X \in \Gamma$ **do**
    for all $Y \in \Gamma$ **do**

$chart[i,n][\bar{X}] \leftarrow chart[i,n][\bar{X}]$
$\quad\quad \oplus (\mathbf{T}_{X,Y} + chart[i,i+1][X] + chart[i+1,n][\bar{Y}])$
end for
end for
end for

which is $O(n)$ as desired.

Vectorization. The other thing that is special about our grammar is that it's very *dense*, in the sense that it has a rule $X \rightarrow a$ for every single X, and a rule $\bar{X} \rightarrow X\ \bar{Y}$ for every single X and Y. Instead of thinking of $chart[i,j]$ as a hash table from labels to numbers, we think of it as a vector of numbers. Instead of all those loops over labels, we can now use vector operations.

1: **for** $i \leftarrow 1, \ldots, n-1$ **do**
2: $\quad chart[i-1,i] \leftarrow \mathbf{O}_{i,*}$
3: **end for**
4: **for all** $X \neq \mathrm{E}$ **do**
5: $\quad chart[n-1,n]_X \leftarrow -\infty$
6: **end for**
7: $chart[n-1,n]_{\mathrm{E}} \leftarrow \mathbf{O}_{n,\mathrm{E}}$
8: **for** $i \leftarrow n-2, \ldots, 0$ **do**
9: $\quad chart[i,n]_X \leftarrow \bigoplus_Y \left(\mathbf{T}_{X,Y} + chart[i,i+1]_X + chart[i+1,n]_Y\right)$
10: **end for**
11: **return** $chart[0,n]_{\mathrm{S}}$

That's the whole algorithm! But line 9 requires some explanation.

- The meaning of the index X on the left-hand side means that this assignment should be performed for all values of X.
- The meaning of $\bigoplus_Y$ is to perform $\oplus$ over all values of Y. In other words: During training, $\oplus$ is logaddexp, so $\bigoplus$ is the log-sum-exp over all values of Y (PyTorch `logsumexp`). During labeling, $\oplus$ is max, so $\bigoplus$ takes the maximum over all values of Y.

This line can be coded in PyTorch without any loops using *broadcasting*.[3]

- $\mathbf{T}$ has rows indexed by X and columns indexed by Y.
- $chart[i,i+1]$ is indexed by X, so we can think of it as a column vector, or a matrix of size $|\Gamma| \times 1$. In PyTorch, there are many ways to reshape it to be $|\Gamma| \times 1$; I usually use `unsqueeze`.
- $chart[i+1,n]$ is indexed by Y, so we can think of it as a row vector, or a matrix of size $1 \times |\Gamma|$.

Unlike in math, in PyTorch you can add matrices of size $|\Gamma| \times |\Gamma|$, $|\Gamma| \times 1$, and $1 \times |\Gamma|$. The column vector and row vector are effectively replicated until they have size $|\Gamma| \times |\Gamma|$; then they are added to form a matrix of size $|\Gamma| \times |\Gamma|$.

[3] Please see `https://pytorch.org/docs/stable/notes/broadcasting.html` for more details.

The above algorithm with $\oplus = \max$ finds the score of the best-scoring labeling, but it doesn't actually find the labeling itself. To do that, we use back-pointers as in CKY:

for $i \leftarrow 1, \ldots, n-1$ **do**
    $chart[i-1, i] \leftarrow \mathbf{O}_{i,*}$
end for
for all $X \neq \mathrm{E}$ **do**
    $chart[n-1, n]_X \leftarrow -\infty$
end for
$chart[n-1, n]_{\mathrm{E}} \leftarrow \mathbf{O}_{n,\mathrm{E}}$
for $i \leftarrow n-2, \ldots, 0$ **do**
    $chart[i, n]_X \leftarrow \max_Y \left(\mathbf{T}_{X,Y} + chart[i, i+1]_X + chart[i+1, n]_Y\right)$
    $back[i, n]_X \leftarrow \arg\max_Y \left(\mathbf{T}_{X,Y} + chart[i, i+1]_X + chart[i+1, n]_Y\right)$
end for

Reconstructing the best label sequence is just like reconstructing the best parse tree in CKY. Namely, $back[i, n]_X = Y$ means that the best labeling of $w_{i+1} \cdots w_n$ that starts with X continues with Y. So we can start with $back[0, n]_{\mathrm{S}}$ and follow the pointers to reconstruct the whole label sequence.

One final note: It's a little weird that this algorithm runs right-to-left. If we had made our original tree left-branching instead of right-branching, the final algorithm would run left-to-right.

8.4 Dialogue Act Classification

Dialogue act classification is one of numerous text classification problems in NLP; another very well-known example is sentiment analysis, which is the task of classifying documents into positive or negative sentiment.

Assume that K is a set of possible classes (`book_hotel`, etc.) and let k range over possible classes. Our training data consists of N strings, each of which is labeled with a correct class. Broadly, we can divide classification models into generative and discriminative models. Since you've seen more sophisticated versions of all these models before, we can hopefully describe them briefly.

8.4.1 Generative classifiers

Generative models are defined by the (by now very familiar) equation

$$\arg\max_k P(k \mid w) = \arg\max_k P(k, w) \tag{8.7}$$

$$= \arg\max_k P(k)\, P(w \mid k) \tag{8.8}$$

where $P(k)$ is just a categorical distribution ($P(k) = c(k)/N$) and $P(w \mid k)$ is some sort of language model that depends on k.

If $P(w \mid k)$ is a unigram language model, this is called a *naïve Bayes* classifier. But it can be a fancier language model, like an n-gram model or an RNN.

8.4.2 Discriminative classifiers

Discriminative classifiers try to directly estimate $P(k \mid w)$. Here, we limit our attention to models of the form

$$\mathbf{h} \in \mathbb{R}^d$$
$$\mathbf{h} = \text{Encoder}(w) \tag{8.9}$$
$$\mathbf{y} = \text{SoftmaxLayer}^{\boxed{1}}(\mathbf{h}) \tag{8.10}$$
$$P(k \mid w) = \mathbf{y}_k \tag{8.11}$$

where $\text{Encoder}(\cdot)$ is some function that encodes strings as vectors.

The simplest choice of encoding would be a bag of words. That is, $d = |\Sigma|$, the words of Σ are numbered $1, \ldots, d$, and $\mathbf{h}_\sigma$ is the number of times that σ occurs in w. The parameters of the SoftmaxLayer would be weights $\mathbf{W}_{k,\sigma}$ that capture how much word σ is predictive of class k; for example, we would expect $\mathbf{W}_{\texttt{find\_hotel},\text{stay}}$ to be high, but $\mathbf{W}_{\texttt{find\_hotel},\text{eat}}$ to be low. This kind of classifier is usually known as *logistic regression*.

There are many imaginable neural networks that can be used as string encoders, but the one that is most often used now works as follows. Prepend a special token `CLS` to the string, so the string is now $\texttt{CLS}w_1 w_2 \cdots w_n$. Then apply a Transformer encoder, that is, look up word and position embeddings for each word, then apply a stack of alternating self-attention layers and position-wise feedforward neural networks. The result is a sequence of $(n + 1)$ vectors. Take the first of these (the one corresponding to `CLS`) to be $\mathbf{h}$.

8.5 Generation

In the "traditional" NLP pipeline, a dialogue system would have processed user input by parsing it (syntax), interpreting it (semantics), and doing something (dialogue policy) that yields semantics for a response (content planning). Then the job of content generation would be to convert that semantics to an output string. Just as semantics has always grappled with the problem of what the output semantic representations should look like, generation has always grappled with the problem of what the *input* semantic representations should look like.

Fortunately, in the task-oriented dialogue system we are sketching out, instead of full semantics we just have frames with slot fillers, and so generation (sentence realization) just involves mapping frames plus fillers into text. For example,

(8.12) `recommend_restaurant`(`name` = Fiddler's Hearth, `neighborhood` = downtown)

might map to

(8.13) I found a place in downtown called Fiddler's Hearth.

We can treat this as a translation problem, but are faced with the problem that the slot fillers are very diverse and the translation model will likely have trouble with slot fillers it's never seen before. For example, it's easy to imagine

that it would map the above frame to "I found a place in downtown called Cafe Navarre," substituting the name of one downtown restaurant for another. One solution would be a *copy mechanism* that we alluded to briefly when talking about semantic parsing.

The solution that Jurafsky and Martin (2021) present is called *delexicalization*. The idea is to train the system to translate from frames like

(8.14) `recommend_restaurant(name, neighborhood)`

to

(8.15) I found a place in `neighborhood` called `name`.

Since these delexicalized frames and sentences abstract away from the slot fillers, they are much less diverse and therefore much easier to learn.

So the training procedure goes like this:

1. For each training frame and sentence,
   (a) Remove the slot fillers from the frame.
   (b) Replace the slot fillers in the sentence with the slot names.
2. Train a translation model on the delexicalized frames and sentences.

And the generation procedure goes like this:

- For each frame and fillers that come from the dialogue policy,
  (a) Remove the slot fillers from the frame.
  (b) Translate the delexicalized frame into a delexicalized sentence.
  (c) Put the slot fillers back into the sentence.

References

Bobrow, D. G. et al. (1977). "GUS, A frame driven dialog system". In: *Artificial Intelligence* 8, pp. 155–173.

Budzianowski, Paweł et al. (Oct. 2018). "MultiWOZ - A Large-Scale Multi-Domain Wizard-of-Oz Dataset for Task-Oriented Dialogue Modelling". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, pp. 5016–5026. DOI: 10.18653/v1/D18-1547. URL: https://aclanthology.org/D18-1547.

Jurafsky, Daniel and James H. Martin (2021). *Speech and Language Processing*. 3rd. Draft of September 21, 2021.

Turing, Alan M. (1950). "Computing Machinery and Intelligence". In: *Mind* LIX.236, pp. 433–460. DOI: 10.1093/mind/LIX.236.433.

Vinyals, Oriol and Quoc Le (2015). "A Neural Conversational Model". In: *ICML Deep Learning Workshop*. URL: https://arxiv.org/abs/1506.05869.

Weizenbaum, Joseph (1966). "ELIZA—A Computer Program For the Study of Natural Language Communication Between Man and Machine". In: *Communications of the ACM* 9.1, pp. 36–45. DOI: 10.1145/365153.365168.