Brian Yu
brian@cs.harvard.edu

David J. Malan
malan@harvard.edu

Lecture 6

These notes reflect the new version of Lecture 6, released on 14 August 2023. If you watched the prior version of the lecture and wish to see its notes, **click here**.

Language

So far in the course, we needed to shape tasks and data such that an AI will be able to process them. Today, we will look at how an AI can be constructed to process human language.

Natural Language Processing spans all tasks where the AI gets human language as input. The following are a few examples of such tasks:

- automatic summarization, where the AI is given text as input and it produces a summary of the text as output.
- information extraction, where the AI is given a corpus of text and the AI extracts data as output.
- language identification, where the AI is given text and returns the language of the text as output.
- machine translation, where the AI is given a text in the origin language and it outputs the translation in the target language.
- named entity recognition, where the AI is given text and it extracts the names of the entities in the text (for example, names of companies).
- speech recognition, where the AI is given speech and it produces the same words in text.
- text classification, where the AI is given text and it needs to classify it as some type of text.
- word sense disambiguation, where the AI needs to choose the right meaning of a word that has multiple meanings (e.g. bank means both a financial institution and the ground on the sides of a river).

Syntax and Semantics

Syntax is sentence structure. As native speakers of some human language, we don't struggle with producing grammatical sentences and flagging non-grammatical sentences as wrong. For example, the sentence "Just before nine o'clock Sherlock Holmes stepped briskly into the room" is grammatical, whereas the sentence "Just before Sherlock Holmes nine o'clock stepped briskly the room" is non-grammatical. Syntax can be grammatical and ambiguous at the same time, as in "I saw the man with the telescope." Did I see (the man with the telescope) or did I see (the man), doing so by looking through the telescope? To be able to parse human speech and produce it, the AI needs to command syntax.

Semantics is the meaning of words or sentences. While the sentence "Just before nine o'clock Sherlock Holmes stepped briskly into the room" is syntactically different from "Sherlock Holmes stepped briskly into the room just before nine o'clock," their content is effectively identical. Similarly, although the sentence "A few minutes before nine, Sherlock Holmes walked quickly into the room" uses different words from the previous sentences, it still carries a very similar meaning. Moreover, a sentence can be perfectly grammatical while being completely nonsensical, as in Chomsky's example, "Colorless green ideas sleep furiously." To be able to parse human speech and produce it, the AI needs to command semantics.

Context-Free Grammar

Formal Grammar is a system of rules for generating sentences in a language. In **Context-Free Grammar**, the text is abstracted from its meaning to represent the structure of the sentence using formal grammar. Let's consider the following example sentence:

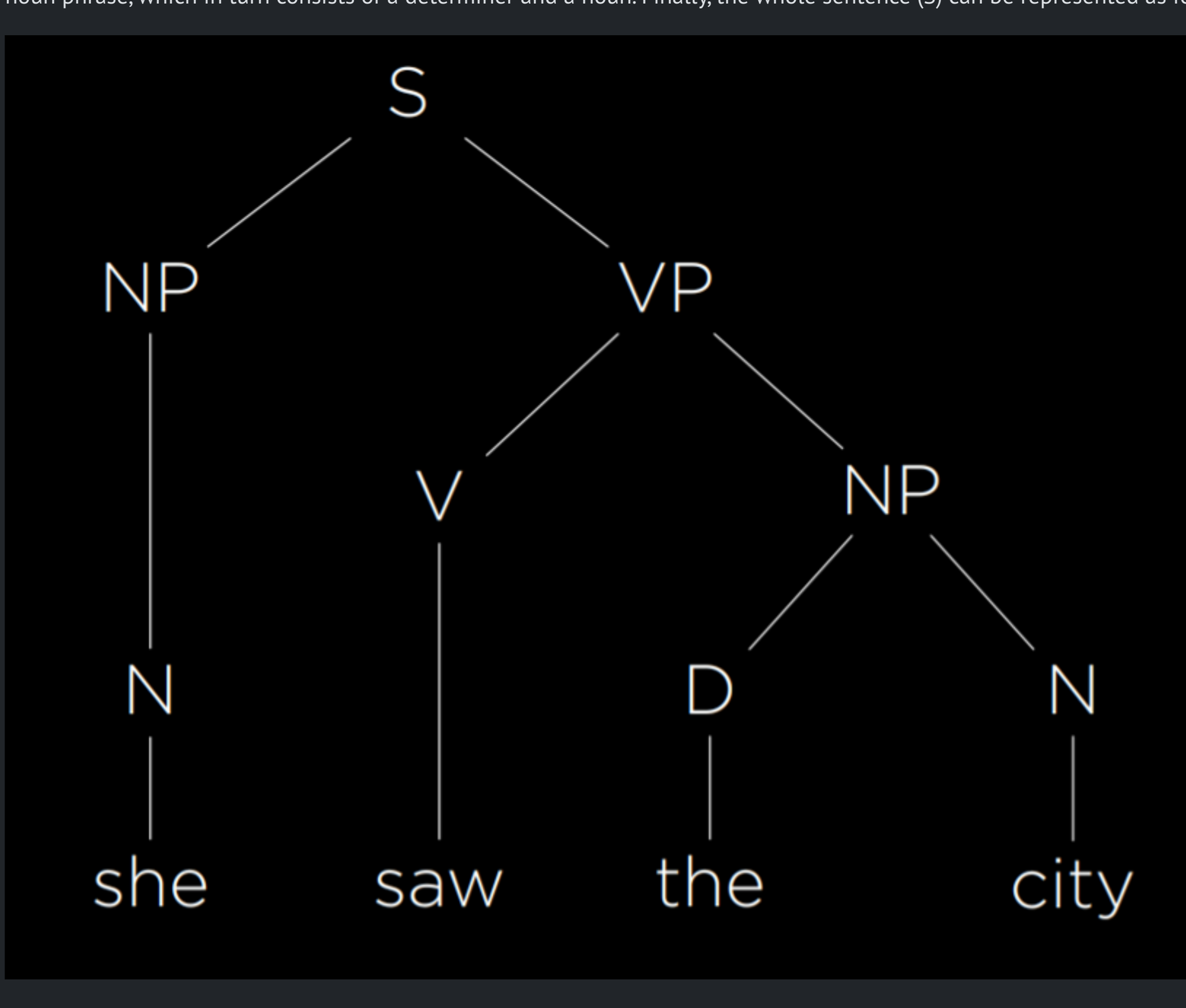
- She saw the city.

This is a simple grammatical sentence, and we would like to generate a syntax tree representing its structure.

We start by assigning each word its part of speech. *She* and *city* are nouns, which we will mark as N. *Saw* is a verb, which we will mark as V. *The* is a determiner, marking the following noun as definite or indefinite, and we will mark it as D. Now, the above sentence can be rewritten as

- N V D N

So far, we have abstracted each word from its semantic meaning to its part of speech. However, words in a sentence are connected to each other, and to understand the sentence we must understand how they connect. A noun phrase (NP) is a group of words that connect to a noun. For example, the word *she* is a noun phrase in this sentence. In addition, the words *the city* also form a noun phrase, consisting of a determiner and a noun. A verb phrase (VP) is a group of words that connect to a verb. The word *saw* is a verb phrase in itself. However, the words *saw the city* also make a verb phrase. In this case, it is a verb phrase consisting of a verb and a noun phrase, which in turn consists of a determiner and a noun. Finally, the whole sentence (S) can be represented as follows:



Using formal grammar, the AI is able to represent the structure of sentences. In the grammar we have described, there are enough rules to represent the simple sentence above. To represent more complex sentences, we will have to add more rules to our formal grammar.

nltk

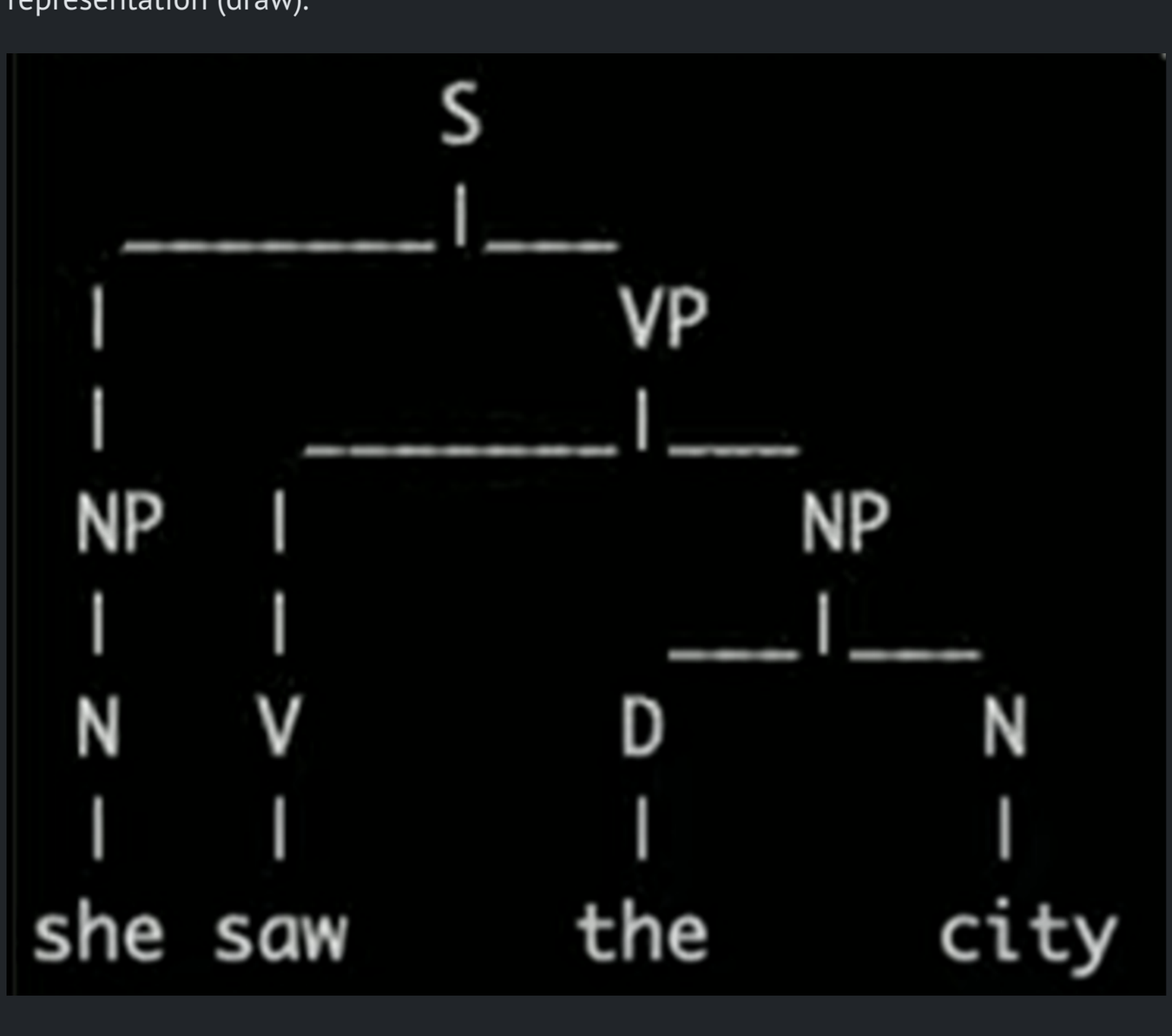
As is often the case in Python, multiple libraries have been written to implement the idea above. nltk (Natural Language Toolkit) is one such library. To analyze the sentence from above, we will provide the algorithm with rules for the grammar:

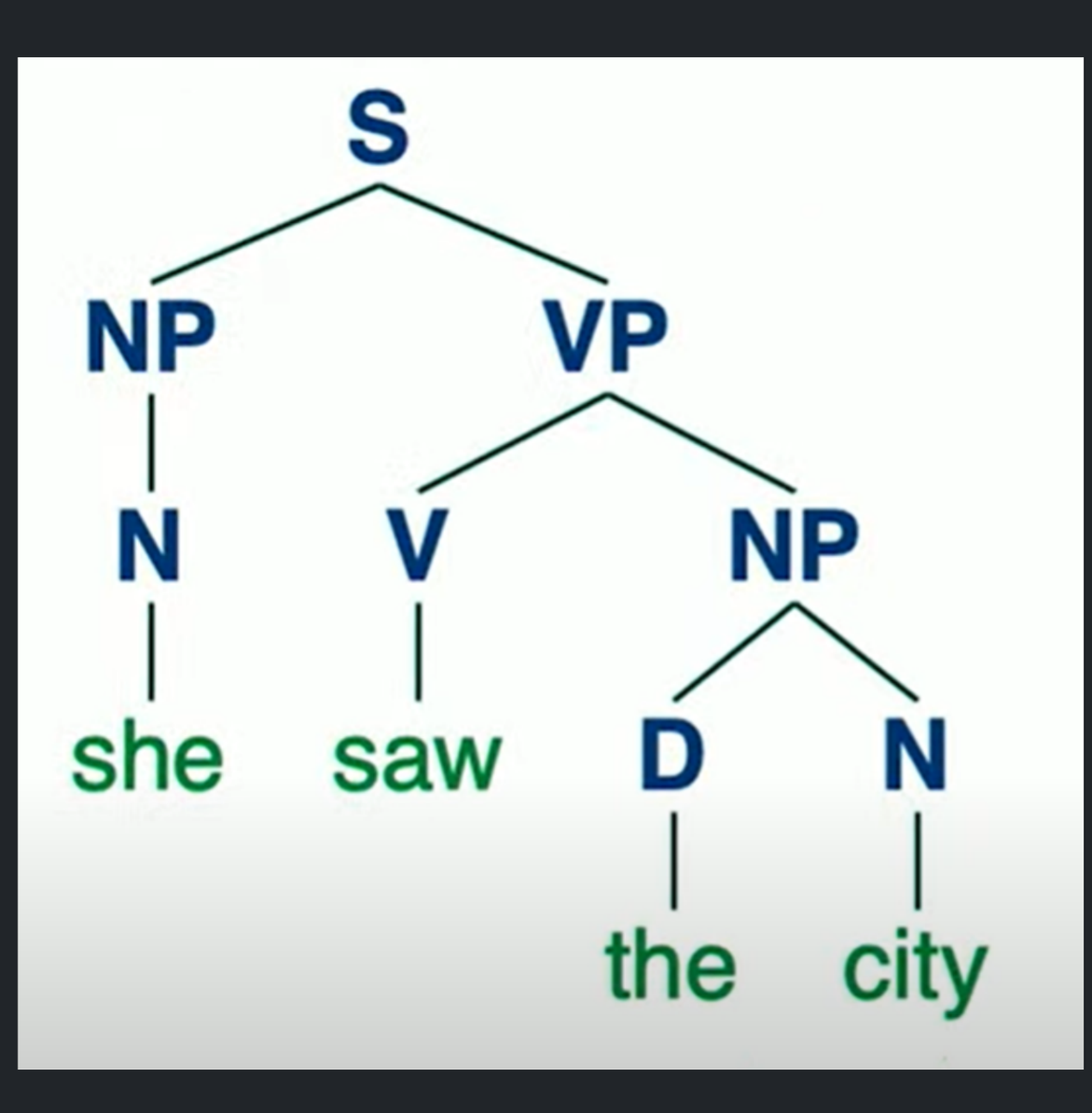
```
import nltk

grammar = nltk.CFG.fromstring("""
    S -> NP VP

    NP -> D N | N
    VP -> V | V NP

    D -> "the" | "a"
    N -> "she" | "city" | "car"
    V -> "saw" | "walked"
""")

parser = nltk.ChartParser(grammar)
```

Similar to what we did above, we define what possible components could be included in others. A sentence can include a noun phrase and a verb phrase, while the phrases themselves can consist of other phrases, nouns, verbs, etc., and, finally, each part of speech spans some words in the language.

```
sentence = input("Sentence: ").split()
try:
    for tree in parser.parse(sentence):
        tree.pretty_print()
        tree.draw()
except ValueError:
    print("No parse tree possible.")
```

After giving the algorithm an input sentence split into a list of words, the function prints the resulting syntactic tree (pretty_print) and also generates a graphic representation (draw).





n-grams

An *n*-gram is a sequence of *n* items from a sample of text. In a **character *n*-gram**, the items are characters, and in a **word *n*-gram** the items are words. A *unigram*, *bigram*, and *trigram* are sequences of one, two, and three items. In the following sentence, the first three *n*-grams are "how often have," "often have I," and "have I said."

"How often have I said to you that when you have eliminated the impossible whatever remains, however improbable, must be the truth?"

n-grams are useful for text processing. While the AI hasn't necessarily seen the whole sentence before, it sure has seen parts of it, like "have I said." Since some words occur together more often than others, it is possible to also predict the next word with some probability. For example, your smartphone suggests words to you based on a probability distribution derived from the last few words you typed. Thus, a helpful step in natural language processing is breaking the sentence into n-grams.

Tokenization

Tokenization is the task of splitting a sequence of characters into pieces (tokens). Tokens can be words as well as sentences, in which case the task is called **word tokenization** or **sentence tokenization**. We need tokenization to be able to look at *n*-grams, since those rely on sequences of tokens. We start by splitting the text into words based on the space character. While this is a good start, this method is imperfect because we end up with words with punctuation, such as "remains,". So, for example, we can remove punctuation. However, then we face additional challenges, such as words with apostrophes (e.g. "o'clock") and hyphens (e.g. "pearl-grey). Additionally, some punctuation is important for sentence structure, like periods. However, we need to be able to tell apart between a period at the end of the word "Mr." and a period in the end of the sentence. Dealing with these questions is the process of tokenization. In the end, once we have our tokens, we can start looking at *n*-grams.

Markov Models

As discussed in previous lectures, Markov models consist of nodes, the value of each of which has a probability distribution based on a finite number of previous nodes. Markov models can be used to generate text. To do so, we train the model on a text, and then establish probabilities for every *n*-th token in an *n*-gram based on the *n* words preceding it. For example, using trigrams, after the Markov model has two words, it can choose a third one from a probability distribution based on the first two. Then, it can choose a fourth word from a probability distribution based on the second and third words. To see an implementation of such a model using nltk, refer to generator.py in the source code, where our model learns to generate Shakespeare-sounding sentences. Eventually, using Markov models, we are able to generate text that is often grammatical and sounding superficially similar to human language output. However, these sentences lack actual meaning and purpose.

Bag-of-Words Model

Bag-of-words is a model that represents text as an unordered collection of words. This model ignores syntax and considers only the meanings of the words in the sentence. This approach is helpful in some classification tasks, such as sentiment analysis (another classification task would be distinguishing regular email from spam email). Sentiment analysis can be used, for instance, in product reviews, categorizing reviews as positive or negative. Consider the following sentences:

1. "My grandson loved it! So much fun!"
2. "Product broke after a few days."
3. "One of the best games I've played in a long time."
4. "Kind of cheap and flimsy, not worth it."

Based only on the words in each sentence and ignoring the grammar, we can see that sentences 1 and 3 are positive ("loved," "fun," "best") and sentences 2 and 4 are negative ("broke," "cheap," "flimsy").

Naive Bayes

Naive Bayes is a technique that can be used in sentiment analysis with the bag-of-words model. In sentiment analysis, we are asking "What is the probability that the sentence is positive/negative given the words in the sentence." Answering this question requires computing conditional probability, and it is helpful to recall Bayes' rule from lecture 2:

$$P(b \mid a) = \frac{P(b)\ P(a \mid b)}{P(a)}$$

Now, we would like to use this formula to find P(sentiment | text), or, for example, P(positive | "my grandson loved it"). We start by tokenizing the input, such that we end up with P(positive | "my", "grandson", "loved", "it"). Applying Bayes' ruled directly, we get the following expression: P("my", "grandson", "loved", "it" | positive)*P(positive)/P("my", "grandson", "loved", "it"). This complicated expression will give us the precise answer to P(positive | "my", "grandson", "loved", "it").

However, we can simplify the expression if we are willing to get an answer that's not equal, but proportional to P(positive | "my", "grandson", "loved", "it"). Later on, knowing that the probability distribution needs to sum up to 1, we can normalize the resulting value into an exact probability. This means that we can simplify the expression above to the numerator only: P("my", "grandson", "loved", "it" | positive)*P(positive). Again, we can simplify this expression based on the knowledge that a conditional probability of *a* given *b* is proportional to the joint probability of *a* and *b*. Thus, we get the following expression for our probability: P(positive, "my", "grandson", "loved", "it")*P(positive). Calculating this joint probability, however, is complicated, because the probability of each word is conditioned on the probabilities of the words preceding it. It requires us to compute P(positive)*P("my" | positive)*P("grandson" | positive, "my")*P(loved | positive, "my", "grandson")*P("it" | positive, "my", "grandson", "loved").

Here is where we use Bayes' rules naively: we assume that the probability of each word is independent from other words. This is not true, but despite this imprecision, Naive Bayes' produces a good sentiment estimate. Using this assumption, we end up with the following probability: P(positive)*P("my" | positive)*P("grandson" | positive)*P("loved" | positive)*P("it" | positive), which is not that difficult to calculate. P(positive) = the number of all positive samples divided by the number of total samples. P("loved" | positive) is equal to the number of positive samples with the word "loved" divided by the number of positive samples. Let's consider the example below, with smiling and frowning emojies substituting the words "positive" and "negative":

😀	🙁
0.49	0.51

😀 0.00014112
🙁 0.00006528

	😀	🙁
my	0.30	0.20
grandson	0.01	0.02
loved	0.32	0.08
it	0.30	0.40

On the right we are seeing a table with the conditional probabilities of each word on the left occurring in a sentence given that the sentence is positive or negative. In the small table on the left we are seeing the probability of a positive or a negative sentence. On the bottom left we are seeing the resulting probabilities following the computation. At this point, they are in proportion to each other, but they don't tell us much in terms of probabilities. To get the probabilities, we need to normalize the values, arriving at P(positive) = 0.6837 and P(negative) = 0.3163. The strength of naive Bayes is that it is sensitive to words that occur more often in one type of sentence than in the other. In our case, the word "loved" occurs much more often in positive sentences, which makes the whole sentence more likely to be positive than negative. To see an implementation of sentiment assessment using Naive Bayes with the nltk library, refer to sentiment.py.

One problem that we can run into is that some words may never appear in a certain type of sentence. Suppose none of the positive sentences in our sample had the word "grandson." Then, P("grandson" | positive) = 0, and when computing the probability of the sentence being positive we will get 0. However, this is not the case in reality (not all sentences mentioning grandsons are negative). One way to go about this problem is with **Additive Smoothing**, where we add a value α to each value in our distribution to smooth the data. This way, even if a certain value is 0, by adding α to it we won't be multiplying the whole probability for a positive or negative sentence by 0. A specific type of additive smoothing, **Laplace Smoothing** adds 1 to each value in our distribution, pretending that all values have been observed at least once.

Word Representation

We want to represent word meanings in our AI. As we've seen before, it is convenient to provide input to the AI in the form of numbers. One way to go about this is by using **One-Hot Representation**, where each word is represented with a vector that consists of as many values as we have words. Except for a single value in the vector that is equal to 1, all other values are equal to 0. How we can differentiate words is by which of the values is 1, ending up with a unique vector per word. For example, the sentence "He wrote a book" can be represented as four vectors:

- [1, 0, 0, 0] (he)
- [0, 1, 0, 0] (wrote)
- [0, 0, 1, 0] (a)
- [0, 0, 0, 1] (book)

However, while this representation works in a world with four words, if we want to represent words from a dictionary, when we can have 50,000 words, we will end up with 50,000 vectors of length 50,000. This is incredibly inefficient. Another problem in this kind of representation is that we are unable to represent similarity between words like "wrote" and "authored." Instead, we turn to the idea of **Distributed Representation**, where meaning is distributed across multiple values in a vector. With distributed representation, each vector has a limited number of values (much less than 50,000), taking the following form:

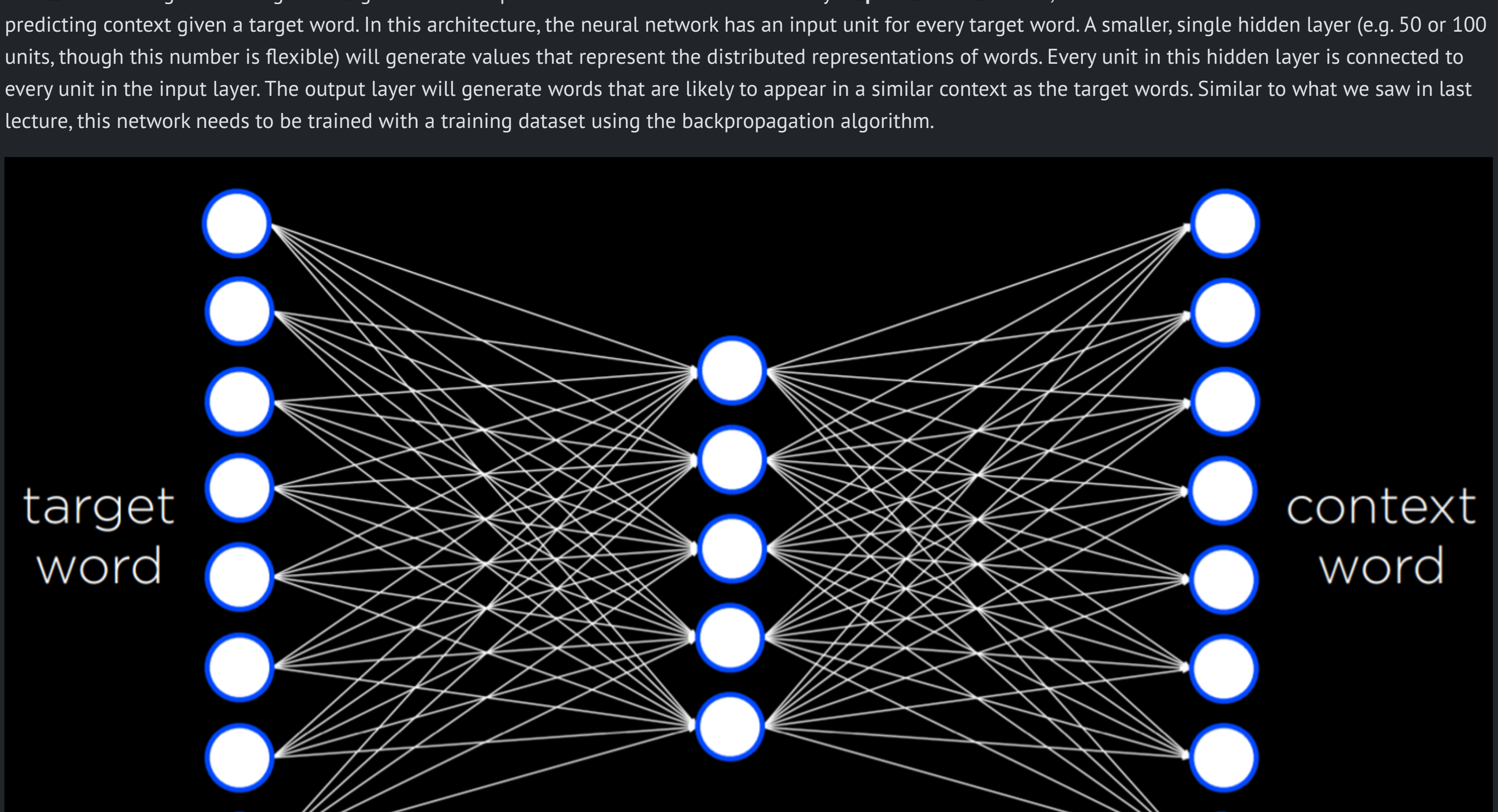
- [-0.34, -0.08, 0.02, -0.18, …] (he)
- [-0.27, 0.40, 0.00, -0.65, …] (wrote)
- [-0.12, -0.25, 0.29, -0.09, …] (a)
- [-0.23, -0.16, -0.05, -0.57, …] (book)

This allows us to generate unique values for each word while using smaller vectors. Additionally, now we are able to represent similarity between words by how different the values in their vectors are.

"You shall know a word by the company it keeps" is an idea by J. R. Firth, an English linguist. Following this idea, we can come to define words by their adjacent words. For example, there are limited words that we can use to complete the sentence "for ___ he ate." These words are probably words like "breakfast," "lunch," and "dinner." This brings us to the conclusion that by considering the environment in which a certain word tends to appear, we can infer the meaning of the word.

word2vec

word2vec is an algorithm for generating distributed representations of words. It does so by **Skip-Gram Architecture**, which is a neural network architecture for predicting context given a target word. In this architecture, the neural network has an input unit for every target word. A smaller, single hidden layer (e.g. 50 or 100 units, though this number is flexible) will generate values that represent the distributed representations of words. Every unit in this hidden layer is connected to every unit in the input layer. The output layer will generate words that are likely to appear in a similar context as the target words. Similar to what we saw in last lecture, this network needs to be trained with a training dataset using the backpropagation algorithm.



This neural network turns out to be quite powerful. At the end of the process, every word ends up being just a vector, or a sequence of numbers. For example,

book: [-0.226776 -0.155999 -0.048995 -0.569774 0.053220 0.124401 -0.091108 -0.606255 -0.114630 0.473384 0.061061 0.551323 -0.245151 -0.014248 -0.210003 0.316162 0.340426 0.232053 0.386477 -0.025104 -0.024492 0.342590 0.205586 -0.554390 -0.037832 -0.212766 -0.048781 -0.088652 0.042722 0.000270 0.356324 0.212374 -0.188433 0.196112 -0.223294 -0.014591 0.067874 -0.448922 -0.290960 -0.036474 -0.148416 0.448422 0.016454 0.071613 -0.078306 0.035400 0.330418 0.293890 0.202701 0.555509 0.447660 -0.361554 -0.266283 -0.134947 0.105315 0.131263 0.548085 -0.195238 0.062958 -0.011117 -0.226676 0.050336 -0.295650 -0.201271 0.014450 0.026845 0.403077 -0.221277 -0.236224 0.213415 -0.163396 -0.218948 -0.242459 -0.346984 0.282615 0.014165 -0.342011 0.370489 -0.372362 0.102479 0.547047 0.020831 -0.202521 -0.180814 0.035923 -0.296322 -0.062603 0.232734 0.191323 0.251916 0.150993 -0.024009 0.129037 -0.033097 0.029713 0.125488 -0.018356 -0.226277 0.437586 0.004913]

By themselves, these numbers don't mean much. But by finding which other words in the corpus have the most similar vectors, we can run a function that will generate the words that are the most similar to the word *book*. In the case of this network it will be: book, books, essay, memoir, essays, novella, anthology, blurb, autobiography, audiobook. This is not bad for a computer! Through a bunch of numbers that don't carry any specific meaning themselves, the AI is able to generate words that really are very similar to *book* not in letters or sounds, but in meaning! We can also compute the difference between words based on how different their vectors are. For example, the difference between *king* and *man* is similar to the difference between *queen* and *woman*. That is, if we add the difference between *king* and *man* to the vector for *woman*, the closest word to the resulting vector is *queen*! Similarly, if we add the difference between *ramen* and *japan* to *america*, we get *burritos*. By using neural networks and distributed representations for words, we get our AI to understand semantic similarities between words in the language, bringing us one step closer to AIs that can understand and produce human language.

Neural Networks

Recall that a **neural network** takes some input, passes it to the network, and creates some output. By providing the network with training data, it can do more and more of an accurate job of translating the input into an output. Commonly, machine translation uses neural networks. In practice, when we are translating words, we want to translate a sentence or paragraph. Since a sentence is a fixed size, we run into the problem of translating a sequence to another sequence where sizes are not fixed. If you have ever had a conversation with an AI chatbot, it needs to understand a sequence of words and generate an appropriate sequence as output.

Recurrent neural networks can re-run the neural network multiple times, keeping track of a state that holds all relevant information. Input is taken into the network, creating a hidden state. Passing a second input into the encoder, along with the first hidden state, produces a new hidden state. This process is repeated until an end token is passed. Then, a decoding state begins, creating hidden state after hidden state until we get the final word and another end token. Some problems, however, arise. One problem in the encoder stage where all the information from the input stage must be stored in one final state. For large sequences, it's very challenging to store all that information into a single state value. It would be useful to somehow combine all the hidden states. Another problem is that some of the hidden states in the input sequence are more important than others. Could there be some way to know what states (or words) are more important than others?

Attention

Attention refers to the neural network's ability to decide what values are more important than others. In the sentence "What is the capital of Massachusetts," attention allows the neural network to decide what values it will pay attention to at each stage of generating the output sentence. Running such a calculation, the neural network will show that when generating the final word of the answer, "capital" and "Massachusetts" are the most important to pay attention to. By taking the attention score, multiplying them by the hidden state values generated by the network, and adding them up, the neural network will create a final context vector that the decoder can use to calculate the final word. A challenge that arises in calculations such as these is that recurrent neural networks require sequential training of word after word. This takes a lot of time. With the growth of large language models, they take longer and longer to train. A desire for parallelism has steadily grown as larger and larger datasets need to be trained. Hence, a new architecture has been introduced.

Transformers

Transformers is a new type of training architecture whereby each input word is passed through a neural network simultaneously. An input word goes into the neural network and is captured as an encoded representation. Because all words are fed into the neural network at the same time, word order could easily be lost. Accordingly, **position encoding** is added to the inputs. The neural network, therefore, will use both the word and the position of the word in the encoded representation. Additionally, a **self-attention** step is added to help define the context of the word being inputted. In fact, neural networks will often use multiple self-attention steps such that they can further understand the context. This process is repeated multiple times for each of the words in the sequence. What results are encoded representations that will be useful when it's time to decode the information.

In the decoding step, the previous output word and its positional encoding are given to multiple self-attention steps and the neural network. Additionally, multiple attention steps are fed the encoded representation from the encoding process and provided to the neural network. Hence, words are able to pay attention to each other. Further, parallel processing is possible, and the calculations are fast and accurate.

Summing Up

We have looked at artificial intelligence in a wide array of contexts. We looked at search problems in how AI can look for solutions. We looked at how AI represents knowledge and create knowledge. We looked at uncertainty when it does not know things for sure. We looked at optimization, maximizing and minimizing function. We looked at machine learning, finding patterns by looking at training data. We learned about neural networks and how they use weights to go from input to output. Today, we looked at language itself and how we can get the computer to understand our language. We have just scratched the surface of this process. We truly hope you enjoyed this journey with us. This was Introduction to Artificial Intelligence with Python.