## Tasks

1. NLP preprocessing
2. N-gram Models
3. Word2Vec (Words Embeddings)
4. Sentimental Analysis using Sequence models

For tasks, please refer to the attached notebook.

# Task 1: NLP Preprocessing

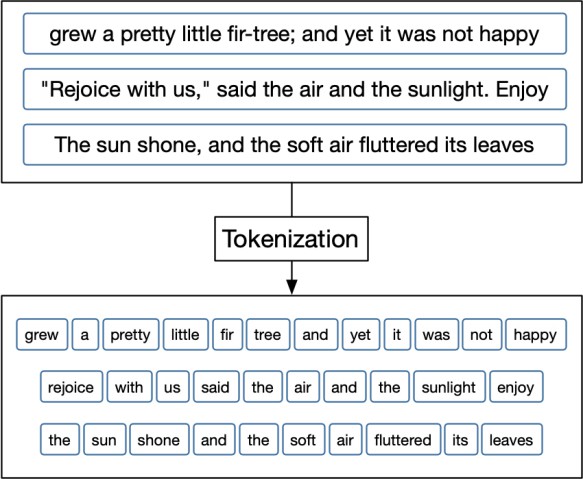
(Below is theoretical foundation, for implementation task have a look at associated ipynb file)

## Conceptual Overview of NLP Preprocessing Techniques

Understanding the foundational concepts of Natural Language Processing (NLP) is essential for grasping how Large Language Models (LLMs) function. Core techniques like tokenization, lemmatization, and part- of-speech tagging serve as the building blocks that enable LLMs to interpret, generate, and reason with human language. A solid grounding in these preprocessing steps will equip you with the conceptual tools needed to work effectively with advanced models like BERT and GPT.

## Tokenization

Tokenization is the task of splitting raw text into smaller units called tokens (e.g., words or punctuation symbols). Formally, “given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation”. These tokens serve as the basic linguistic units for further processing.



**Figure 1 Shows how text is split into word tokens for processing in NLP tasks.**

## Example

Let’s say we have a phrase,

*“Friends, Romans, Countrymen, lend me your ears;”*.

The input string is segmented into the following tokens:-

*“Friends”*, *“Romans”*, *“Countrymen”*, *“lend”*, *“me”*, *“your”*, *“ears”*

The punctuation is removed and it shows how tokenization breaks a sentence into words while discarding characters (like commas or semicolons) that are not needed for semantic processing.

## Stop Word Filtering

Stop words are extremely common words that carry very little meaningful information and are often removed from text during preprocessing. In information retrieval and NLP, some extremely common words which would appear to be of little value in helping select documents matching a user’s query are excluded from the vocabulary entirely. These words are called stop words. The rationale is that removing such high-frequency, low-content words can improve efficiency or effectiveness in tasks like search or text classification.



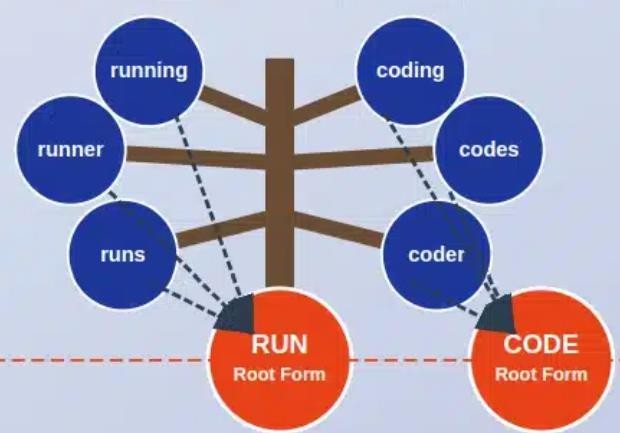
**Figure 2 Stop words filtering**

## Example

Typical stop words include articles and prepositions like *“the”*, *“an”*, *“and”*, *“to”*, *“of”*, *“be”*, etc., which may comprise a large portion of text but contribute minimally to its meaning. For instance, in an IR system, a stop list might filter out words such as *“the”* and *“to”* so that a query *“President of the United States”* is processed mainly for the content words *“President”* and *“United States.*

## Stemming

Stemming is a rule-based process for reducing words to their root or base form by stripping off derivational affixes. In other words, stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. The resulting *stem* is not necessarily a valid word; it is a generic base form which groups together words with similar meaning.



**Figure 3 Reducing words to their Roots**

## Example

A classic stemming algorithm is Porter’s stemmer, which would reduce *“connections”*, *“connected”*, *“connecting”* to the stem **connect**.

However, stems can be imprecise. For example, if confronted with the token *“saw”*, stemming might reduce it to *“s”* (a crude truncation). This happens because a stemmer blindly chops off suffixes (*“saw”*

*→ “s”*) without understanding context.

Despite this crudeness, stemming can be effective for tasks like search indexing where *connect*, *connected*, *connection* should be treated as the same concept.

## Lemmatization

Lemmatization is a more sophisticated normalization process that converts a word to its canonical *lemma* form (the form found in dictionaries), using vocabulary and morphological analysis. In contrast to stemming, lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. In essence, lemmatization considers a word’s part of speech and meaning to ensure that the true root word (lemma) is obtained.

## Examples

A lemmatizer will transform *“machines”* to *“machine”* (singular noun) and *“running”* to *“run”* (present participle to base verb). Unlike a stemmer, it can handle irregular forms: for instance, given the word *“saw”*, a lemmatizer would attempt to determine the context — if *“saw”* is used as a verb, the lemma is *“see”*; if used as a noun (the tool), the lemma remains *“saw”*.

Another example is the NLTK WordNet lemmatizer, which maps *“women”* to *“woman”* (correct plural- to-singular reduction) but leaves *“lying”* unchanged because *“lying”* as a verb form of *“lie”* requires

context that a simple lookup can’t resolve. Lemmatization thus produces valid dictionary forms, making it especially useful when we need actual words (e.g., for display or linguistic analysis) rather than abstract stems.

## Part -of-Speech (POS) Tagging

Part-of-speech tagging is the process of assigning each word in a text a tag that indicates its grammatical category (noun, verb, adjective, etc.) based on its role in the sentence. In formal terms, the process of classifying words into their parts-of-speech and labeling them accordingly is known as part-of-speech tagging, POS tagging, or simply tagging. The set of POS tags (tagset) is a predefined inventory of possible syntactic categories (also called word classes or lexical categories) for the language. POS tagging is a fundamental step in parsing and understanding sentence structure, often preceding more complex analyses.

## Examples

For instance, given the sentence *“And now for something completely different”*, a POS tagger will output a sequence of tagged tokens: -

### And/CC, now/RB, for/IN, something/NN, completely/RB, different/JJ

Here :-

* + - *“And”* is tagged *CC* (coordinating conjunction)
    - *“now”* is *RB* (adverb)
    - *“for”* is *IN* (preposition)
    - *“something”* is *NN* (noun)
    - *“completely”* is *RB* (adverb)
    - *“different”* is *JJ* (adjective)

This example illustrates how each word is disambiguated by context: e.g., “light” in “light rain” would be tagged as JJ (adjective) but in “light the candle” as VB (verb). Modern taggers use algorithms (like Hidden Markov Models or neural networks) trained on labeled data to achieve high accuracy in predicting the correct tag for each word.

|  |  |  |
| --- | --- | --- |
| Tag | Meaning | Example |
| CC | Coordinating conjunction | and, but, or |
| CD | Cardinal number | one, two, 3 |
| DT | Determiner | the, a, an |
| EX | Existential there | there (is) |
| FW | Foreign word | d'accord, faux pas |

|  |  |  |
| --- | --- | --- |
| IN | Preposition or subordinating conj. | in, of, like, because |
| JJ | Adjective | green, quick |
| JJR | Adjective, comparative | greener, faster |
| JJS | Adjective, superlative | greenest, fastest |
| LS | List item marker | 1), A) |
| MD | Modal | can, must, will |
| NN | Noun, singular or mass | book, dog |
| NNS | Noun, plural | books, dogs |
| NNP | Proper noun, singular | John, London |
| NNPS | Proper noun, plural | Americans, Beatles |
| PDT | Predeterminer | all, both, half |
| POS | Possessive ending | ’s, ’ |
| PRP | Personal pronoun | I, you, he, she |
| PRP$ | Possessive pronoun | my, your, his |
| RB | Adverb | quickly, never |
| RBR | Adverb, comparative | faster, better |
| RBS | Adverb, superlative | fastest, best |
| RP | Particle | up, off, out |
| SYM | Symbol | $, %, + |
| TO | to (as in to go) | to |
| UH | Interjection | oh, wow, hey |
| VB | Verb, base form | run, go, eat |
| VBD | Verb, past tense | ran, went, ate |
| VBG | Verb, gerund or present participle | running, eating |
| VBN | Verb, past participle | eaten, taken |
| VBP | Verb, non-3rd person singular present | run, eat (I/you/we) |
| VBZ | Verb, 3rd person singular  present | runs, eats (he/she) |
| WDT | Wh-determiner | which, that |
| WP | Wh-pronoun | who, what |
| WP$ | Possessive wh-pronoun | whose |
| WRB | Wh-adverb | where, when, why |

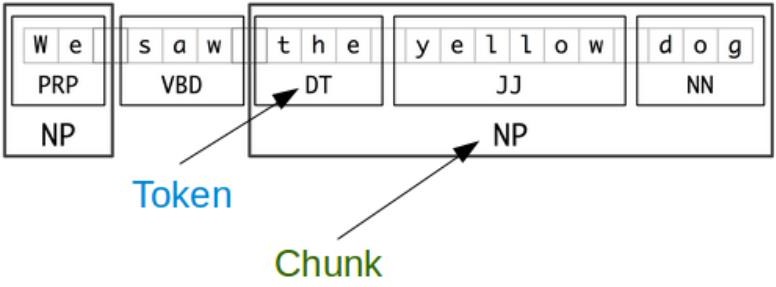
**Table 1 Common Part-of-Speech (POS) tags from the Penn Treebank tagset, along with their meanings and example.**

## Chunking

Chunking (also known as *shallow parsing*) is a technique for grouping adjacent tokens in a sentence into higher-level units or *chunks*, such as noun phrases or verb phrases. In practice, chunking *segments and labels multi-token sequences* as meaningful phrases. Each chunk is a contiguous substring of the sentence (typically a syntactic constituent) that does not overlap with other chunks. For example, a chunker might label “[John] [lives in New York]” where “[John]” is a noun phrase chunk and “[lives in

New York]” is a verb phrase or prepositional phrase chunk, simplifying the parse structure by focusing

only on shallow groupings.



**Figure 4 Tokens and Chunks**

## Examples

Let’s consider the Wall Street Journal sentence,

“The market for system-management software for Digital’s hardware is fragmented enough that a giant such as Computer Associates should do well there.”

An NP-chunker would identify noun phrase chunks as bracketed segments:

* [The/DT market/NN]
* for/IN
* [system-management/NN software/NN]
* for/IN
* [Digital/NNP 's/POS hardware/NN]
* is/VBZ
* fragmented/JJ
* enough/RB
* that/IN
* [a/DT giant/NN]
* such/JJ
* as/IN
* [Computer/NNP Associates/NNPS]

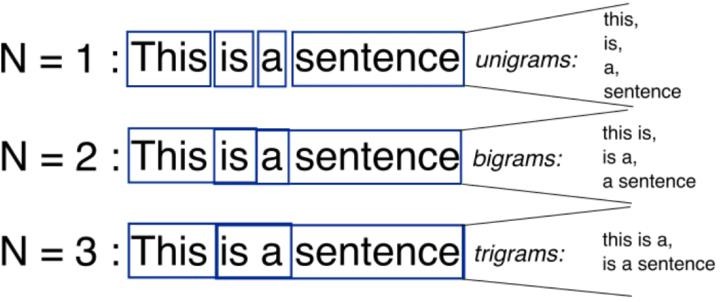
The chunking splits the sentence into single-word tokens and multi-word noun phrase chunks according to the context of the sentence.

# Task 2: N-gram Language Models

(Below is theoretical foundation, for implementation task have a look at associated ipynb file)

# N-gram Model Generation:

**N-gram generation** refers to generating text by *sampling* from an N-gram language model. Starting from a beginning-of-sentence token <s>, words are chosen **randomly according to the model’s learned probabilities** and appended one by one until an end-of-sentence token is produced. In practice, *we continue choosing random numbers and generating words until we generate the final token </s> of the sentence*. This method (first described by Shannon in 1951) treats the N-gram model as a probability distribution from which we can draw words in sequence.

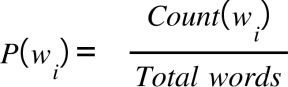


**Figure 5. Examples of unigrams, bigrams, and trigrams.**

## Examples

Generating Shakespeare using the N-gram approach.

# Unigrams (N=1)

Mathematically: -

**(1)**

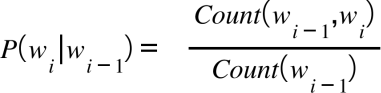
## Example

“To him swallowed confess hear both. Which. Of save on trial for are ay device and rote life have c.

Hill he late speaks; or! A more or legless first you enter.”

* Each word is selected independently, based only on its overall frequency in the corpus.
* Results in nonsensical text with no grammatical structure, just random frequent words.

# Bigrams (N=2)

Mathematically: -

**(2)**

## Example

“What means, sir. I confess she? Then all sorts, he is trim, captain. Why doest stand forth they canopy, forsooth he is this palpable hit the King Henry. Live king. Follow”

* Each word depends on the **previous word**.
* Result: Slightly more grammatical but still awkward.

# Next Word Prediction using N-gram Models

Text generation involves predicting the next word or character in a sequence of text based on the preceding context. The process involves the following steps:

## Tokenization

First, the text is split into individual words or tokens. In English, words are often separated by whitespace, though simple whitespace splitting is not always sufficient (e.g. “New York” contains a space but is a single logical unit. Tokenization ensures the input text is segmented into a sequence of discrete tokens (words) for further processing.

## n-Gram Generation

Next, we generate all *n*-grams from the tokenized text. For example, a 2-gram (bigram) might

be “The water” or “water of,” and a 3-gram (trigram) could be “The water of” or “water of

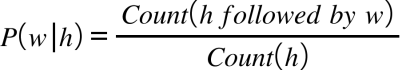
Walden”. Each n-gram includes a group of previous words (the context) and a current word (the target).

## Frequency Calculation

Using a large corpus of text, we count how often each n-gram occurs. For example, to estimate the likelihood of a particular next word *w* following a history *h*, one would *count the number of times we see* h *and count the number of times this is followed by* w”*.*This effectively answers the question: “Out of the times we saw history *h*, how many times was it followed by word *w*?. All n-gram counts (and especially counts of the shorter *(n−1)*-gram contexts) are compiled from the training corpus.

## Probability Computation:

The conditional probability of a word *w* given a preceding context *h* is estimated by the relative frequency of that n-gram. In other words, the maximum likelihood estimate for an n-gram model is:



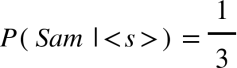
For example, suppose we have a tiny corpus of three sentences:-

* I am Sam
* Sam I am
* I do not like green eggs and ham The result of tokenization is:-
* <s> I am Sam </s>
* <s> Sam I am </s>
* <s> I do not like green eggs and ham </s>

From this processed corpus, we can extract bigrams and compute their frequencies. For example:

* The bigram <s> I occurs 2 times
* The bigram <s> Sam occurs 1 time

Therefore, the conditional probabilities for the first word after <s> are:

* P open parentheses I space vertical line less than s greater than close parentheses equals 2 over 3  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mi>P</mi><mfenced><mrow><mi>I</mi><mo>&#xA0;</mo><mo>|</mo><mo>&lt;</mo><mi>s</mi><mo>&gt;</mo></mrow></mfenced><mo>=</mo><mfrac><mn>2</mn><mn>3</mn></mfrac></math>","origin":"MathType Legacy","version":"v3.18.1"}
* 

## Next-Word Prediction:

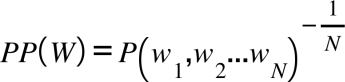
Finally, given a specific history (context), the model considers all possible next words and their computed probabilities, and predicts the word with the highest probability as the most likely next word. For instance in the previous case that we explained the model will predict the word “I” ad the next word after the start of the sentence. Essentially, the N-gram model provides a probability distribution over the next-word candidates, and the predicted word is the one that maximizes this conditional probability.

# Perplexity

*Is the standard evaluation metric for language models, indicating how well a model predicts a sample of text. Formally, perplexity is defined as the inverse probability of the test set, normalized by the number of words.*

Mathematically, given a test word sequence: -

W equals w subscript 1 comma w subscript 2 horizontal ellipsis w subscript n  {"mathml":"<math style=\"font-family:Times New Roman;font-size:12px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mi>W</mi><mo>=</mo><msub><mi>w</mi><mn>1</mn></msub><mo>,</mo><msub><mi>w</mi><mn>2</mn></msub><mo>&#x2026;</mo><msub><mi>w</mi><mi>n</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}(3)

The perplexity is: -

(4)

Intuitively, *the higher the model’s assigned probability to the test data, the lower the perplexity*. A better model makes the test data less surprising. T*he higher the conditional probability of the word sequence, the lower the perplexity. Thus minimizing perplexity is equivalent to maximizing the test set probability according to the model.*

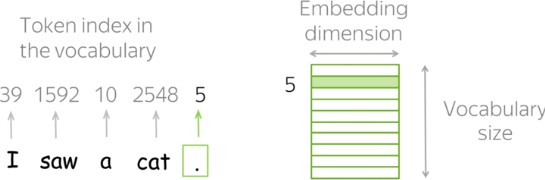
# Task 3: Word2Vec

(Below is theoretical foundation, for implementation task have a look at associated ipynb file)

# Word Embedding (word2vec)

The way machine learning models "see" data is different from how we (humans) do. For example, we can easily understand the text "I saw a cat", but our models can not - they need vectors of features. Such vectors, or word embeddings, are representations of words which can be fed into your model.

In practice, you have a vocabulary of allowed words, you choose this vocabulary in advance. For each vocabulary word, a look-up table contains its embedding. This embedding can be found using the word index in the vocabulary (i.e., you to look up the embedding in the table using word index).



To account for unknown words (the ones which are not in the vocabulary), usually a vocabulary contains a special token UNK. Alternatively, unknown tokens can be ignored or assigned a zero vector.

## Why Use Embeddings?

The core idea behind embeddings is to make it possible for computers to process and analyze data. In the case of text data, for example, words are essentially symbols that need to be translated into numbers. Embeddings enable machines to:

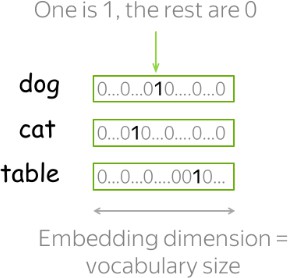
1. Understand relationships between different pieces of information.
2. Perform arithmetic operations on data, such as measuring similarities and clustering.
3. Use vector-based models for tasks such as classification, recommendation, and information retrieval.

## Methods to get vectors corresponding to words

There are several methods available in the literature to produce robust word embeddings that can be used in any of the downstream task.

## One-hot Vectors

* + - * The easiest you can do is to represent words as one-hot vectors: for the i-th word in the vocabulary, the vector has 1 on the i-th dimension and 0 on the rest. In Machine Learning, this is the most simple way to represent categorical features.
      * You probably can guess why one-hot vectors are not the best way to represent words. One of the problems is that for large vocabularies, these vectors will be very long: vector dimensionality is equal to the vocabulary size. This is undesirable in practice, but this problem is not the most crucial one.
      * What is really important, is that these vectors know nothing about the words they represent. For example, one-hot vectors "think" that cat is as close to dog as it is to table! We can say that one-hot vectors do not capture meaning.



## Count-Based Methods

Main idea is that we have to put information about contexts into word vectors. The general procedure to do so comprises the two steps:

1. construct a word-context matrix
2. reduce its dimensionality.

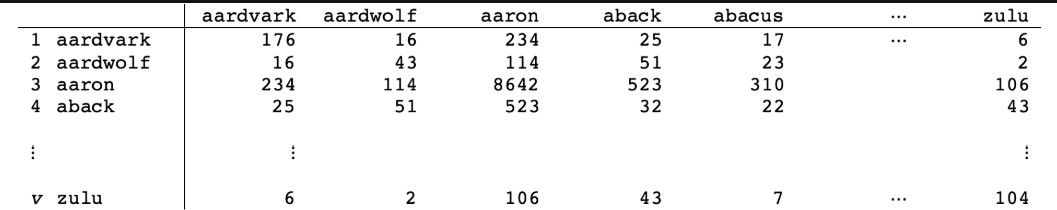
There are two reasons to reduce dimensionality. First, the raw matrix is very large. Second, since a lot of words appear in only a few possible contexts, this matrix potentially has a lot of uninformative elements (e.g., zeros). To estimate similarity between words/contexts, usually you need to evaluate the dot-product of normalized word/context vectors (i.e., cosine similarity). To define a count-based method, we need to define two things:

* + possible contexts (including what does it mean that a word appears in a context),
  + the notion of association, i.e., formulas for computing matrix elements.

Below we provide the details of a popular way of doing this i.e., the Co-Occurance counts.

**Co-occurrence**

Now, let’s look at an example of a co-occurrence matrix. The figure below is a table that shows how often two words appear together within each document in a corpus. This table can also be considered a matrix of size *v×v* that is independent of the number of documents. Because the matrix quantifies word proximity, the values point towards word meaning according to the distributional hypothesis in linguistics.



Each row of this matrix can be used as a word vector of length v, where v is the vocabulary. The matrix is a model of word proximity and is a denser matrix than the one hot representation based matrix shown previously. But a co-occurrence matrix need not be used solely for words per document. One could look instead at words paired in paragraphs, sentences, or within a window of a certain number of words. It turns out that setting a relevant window size is quite important. Window size defines the number of words to the left and right of a target word that are considered its context.

**Example**

For example, in the sentence:

"The quick brown fox jumps over the lazy dog"

If the window size is 2, the context for the word "fox" is:

* + Left: "quick", "brown"
  + Right: "jumps", "over"

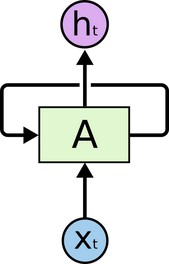
So the model will learn relationships between "fox" and those four words. In essence, co-occurrence- based methods provide a meaningful foundation for word embeddings by capturing how words are used together in context, laying the groundwork for models to understand and quantify semantic similarity.

# Task 4: Sentimental Analysis using LSTM

(Below is theoretical foundation, for implementation task have a look at associated ipynb file)

## Theoretical foundations of RNNs

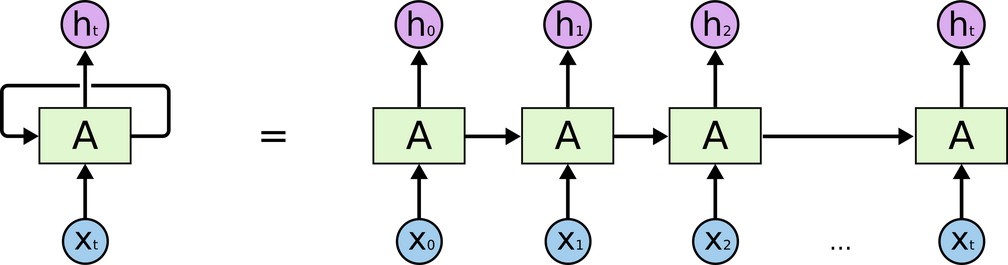
Simple multi-layered neural networks are classifiers which when given a certain input, tag the input as belonging to one of the many classes. They are trained using the existing backpropagation algorithms. These networks are great at what they do but they are not capable of handling inputs which come in a sequence. For example, for a neural net to identify the nouns in a sentence, having just the word as input is not helpful at all. A lot of information is present in the context of the word which can only be determined by looking at the words near the given word. The entire sequence is to be studied to determine the output. This is where Recurrent Neural Networks (RNNs) find their use. As the RNN traverses the input sequence, output for every input also becomes a part of the input for the next item of the sequence.



**Figure 6. Basic structure of a Recurrent Neural Network (RNN) cell**

In the above diagram, a chunk of neural network, *A*, looks at some input x subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>x</mi><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"} and outputs a value h subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>h</mi><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren’t all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



**Figure 7. An unrolled recurrent neural network.**

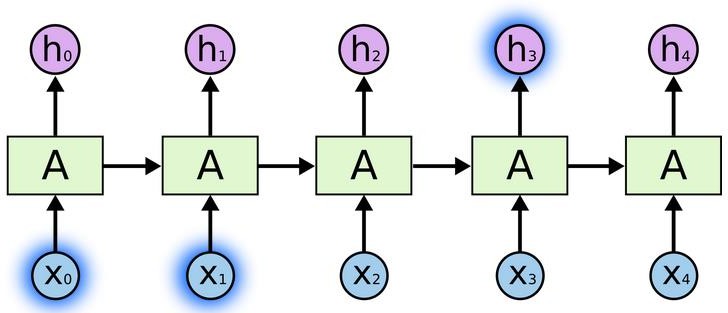
This chain-like nature reveals that recurrent neural networks are intimately related to sequences and

lists. They’re the natural architecture of neural network to use for such data.

“LSTMs,” a very special kind of recurrent neural network which works, for many tasks, much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It’s these LSTMs that will be used in this lab to perform the task of sentiment analysis.

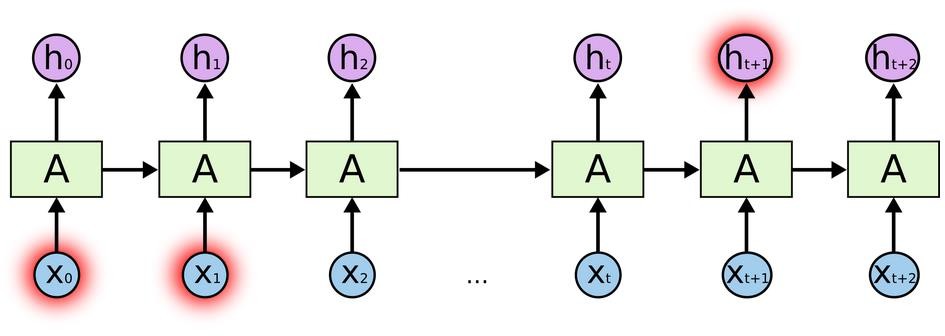
## The Problem of Long-Term Dependencies

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the *sky*,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.



**Figure 8. Recurrent Neural Network (RNN) across time steps, processing sequential inputs and generating hidden states.** But there are also cases where we need more context. Consider trying to predict the last word in the text: “I grew up in France… I speak fluent *French*.”

Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it needs to become very large. Unfortunately, as that gap grows, RNNs become unable to learn how to connect the information.



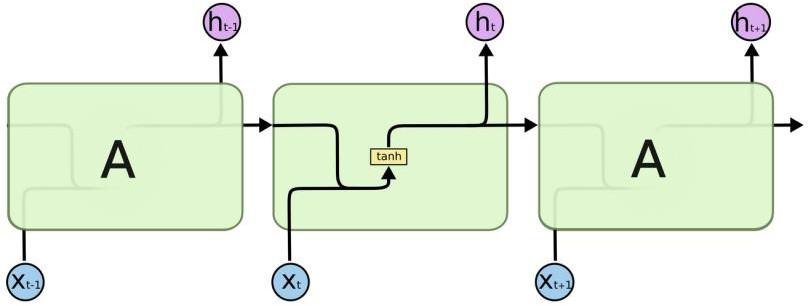
**Figure 9. Unrolled RNN architecture processing long sequence**

In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. The LSTMs were proposed to address this issue of long term dependencies.

## LSTM Networks

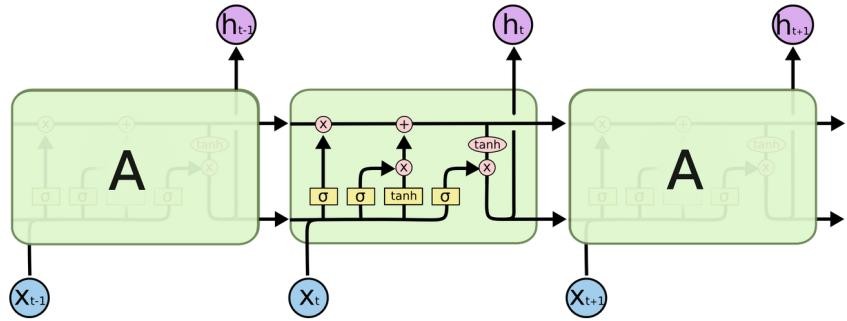
Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced in [1], and were refined and popularized by their usage in various applications afterwards. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



**Figure 10. The repeating module in a standard RNN**

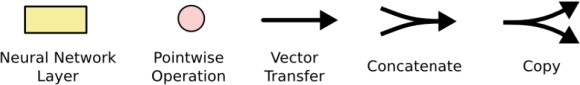
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



**Figure 11. The repeating module in an LSTM**

## Architectural Components

In the diagram given below, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.

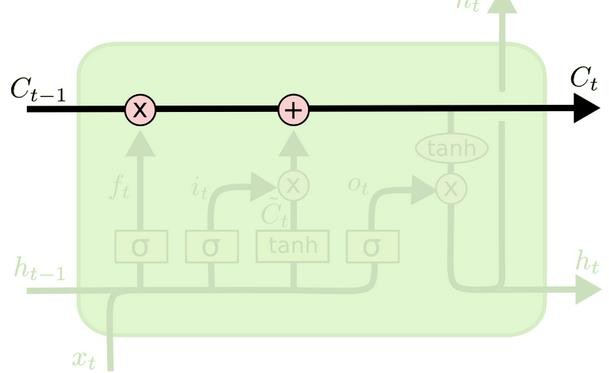


**Figure 12. Notations**

## The Core Idea Behind LSTMs

* + 1. **Cell State**

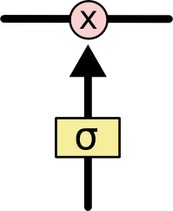
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along unchanged.



**Figure 13. Cell state in LSTMs**

## Gates

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. The gates are composed of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!” An LSTM has three of these gates, to protect and control the cell state.



**Figure 14. Gate in an LSTM controlling the information flow.**

## Step-by-Step LSTM Walk Through

* + - * h subscript t minus 1 end subscript  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>h</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at and

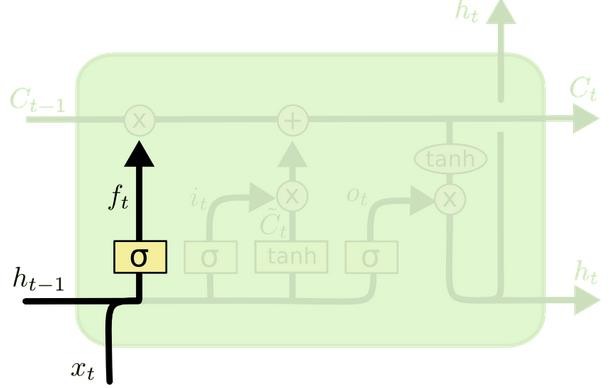
x subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>x</mi><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"} , and outputs a number between 0 and 1 for each number in the cell state C subscript t minus 1 end subscript  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>C</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub></math>","origin":"MathType Legacy","version":"v3.18.1"} . Where 1

represents “completely keep this” while a 0 represents “completely get rid of this.

## Example

Let’s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

**f subscript t equals sigma open parentheses W subscript f times left square bracket h subscript t minus 1 end subscript comma x subscript t right square bracket plus b subscript f close parentheses  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>f</mi><mi>t</mi></msub><mo>=</mo><mi>&#x3C3;</mi><mfenced><mrow><msub><mi>W</mi><mi>f</mi></msub><mo>&#xB7;</mo><mo>[</mo><msub><mi>h</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub><mo>,</mo><msub><mi>x</mi><mi>t</mi></msub><mo>]</mo><mo>+</mo><msub><mi>b</mi><mi>f</mi></msub></mrow></mfenced></math>","origin":"MathType Legacy","version":"v3.18.1"}(5)**

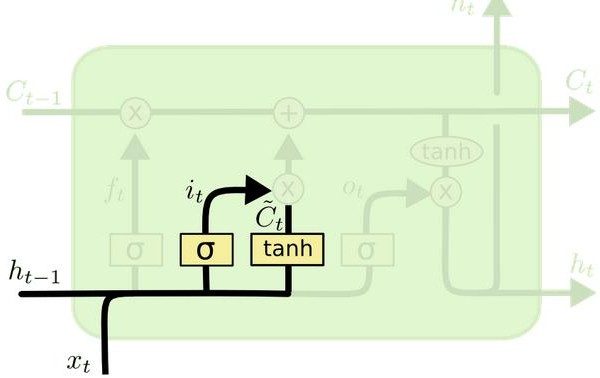
****

* + - * The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, stack C subscript t with tilde on top  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mover><msub><mi>C</mi><mi>t</mi></msub><mo>~</mo></mover></math>","origin":"MathType Legacy","version":"v3.18.1"} , that could be added to the state. In the next step, we’ll combine these two to create an update to the state.

In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.

i subscript t equals sigma open parentheses W subscript i times open square brackets h subscript t minus 1 end subscript comma x subscript t close square brackets plus b subscript i close parentheses  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>i</mi><mi>t</mi></msub><mo>=</mo><mi>&#x3C3;</mi><mfenced><mrow><msub><mi>W</mi><mi>i</mi></msub><mo>&#xB7;</mo><mfenced open=\"[\" close=\"]\"><mrow><msub><mi>h</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub><mo>,</mo><msub><mi>x</mi><mi>t</mi></msub></mrow></mfenced><mo>+</mo><msub><mi>b</mi><mi>i</mi></msub></mrow></mfenced></math>","origin":"MathType Legacy","version":"v3.18.1"} **(6)**

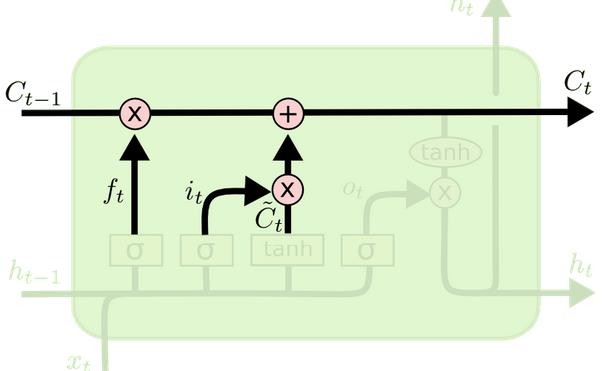
**C with tilde on top subscript t equals tanh open parentheses W subscript C times open square brackets h subscript t minus 1 end subscript comma x subscript t close square brackets plus b subscript C close parentheses  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mover><mi>C</mi><mo>~</mo></mover><mi>t</mi></msub><mo>=</mo><mi>tanh</mi><mfenced><mrow><msub><mi>W</mi><mi>C</mi></msub><mo>&#xB7;</mo><mfenced open=\"[\" close=\"]\"><mrow><msub><mi>h</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub><mo>,</mo><msub><mi>x</mi><mi>t</mi></msub></mrow></mfenced><mo>+</mo><msub><mi>b</mi><mi>C</mi></msub></mrow></mfenced></math>","origin":"MathType Legacy","version":"v3.18.1"}(7)**

****

* + - * It’s now time to update the old cell state, C subscript t minus 1 end subscript  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>C</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}, into the new cell state C subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>C</mi><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>f</mi><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"}, forgetting the things we decided to forget earlier. Then we add i subscript t asterisk times stack C subscript t with tilde on top  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>i</mi><mi>t</mi></msub><mo>*</mo><mover><msub><mi>C</mi><mi>t</mi></msub><mo>~</mo></mover></math>","origin":"MathType Legacy","version":"v3.18.1"}. This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.

C subscript t equals f subscript t asterisk times C subscript t minus 1 end subscript plus i subscript t asterisk times C with tilde on top subscript t  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>C</mi><mi>t</mi></msub><mo>=</mo><msub><mi>f</mi><mi>t</mi></msub><mo>*</mo><msub><mi>C</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub><mo>+</mo><msub><mi>i</mi><mi>t</mi></msub><mo>*</mo><msub><mover><mi>C</mi><mo>~</mo></mover><mi>t</mi></msub></math>","origin":"MathType Legacy","version":"v3.18.1"} **(8)**

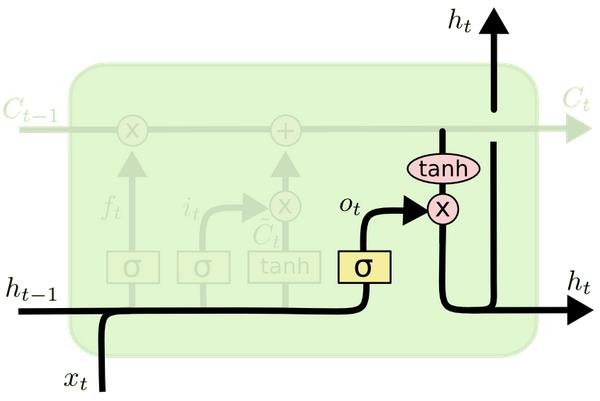
****

* + - * Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through *tanh* (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that’s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that’s what follows next.

o subscript t equals sigma open parentheses W subscript o open square brackets h subscript t minus 1 end subscript comma x subscript t close square brackets plus b subscript o close parentheses  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>o</mi><mi>t</mi></msub><mo>=</mo><mi>&#x3C3;</mi><mfenced><mrow><msub><mi>W</mi><mi>o</mi></msub><mfenced open=\"[\" close=\"]\"><mrow><msub><mi>h</mi><mrow><mi>t</mi><mo>-</mo><mn>1</mn></mrow></msub><mo>,</mo><msub><mi>x</mi><mi>t</mi></msub></mrow></mfenced><mo>+</mo><msub><mi>b</mi><mi>o</mi></msub></mrow></mfenced></math>","origin":"MathType Legacy","version":"v3.18.1"} **(9)**

**h subscript t equals o subscript t times tanh open parentheses C subscript t close parentheses  {"mathml":"<math style=\"font-family:Times New Roman;font-size:10px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><msub><mi>h</mi><mi>t</mi></msub><mo>=</mo><msub><mi>o</mi><mi>t</mi></msub><mo>&#xB7;</mo><mi>tanh</mi><mfenced><msub><mi>C</mi><mi>t</mi></msub></mfenced></math>","origin":"MathType Legacy","version":"v3.18.1"}(10)**

****

## Encoding schemes

Now you are familiar with the concepts of LSTMs to be used in the lab. An other concept to briefly know is the encoding schemes.

## Word Encoding (Integer Encoding)

It is a type of encoding that assigns numerical indices to words. Each unique word receives a unique integer. To standardize sequence length for batch processing in models, sequences are padded with zeros.

### Example

* + - * Sentence: "I love NLP"
      * Encoded: [1, 2, 3]
      * Padded (max length 5): [1, 2, 3, 0, 0]

## Label Encoding

converts categorical labels into numerical format. For binary sentiment classification:

### Example

* + - * "Positive" → 1, "Negative" → 0