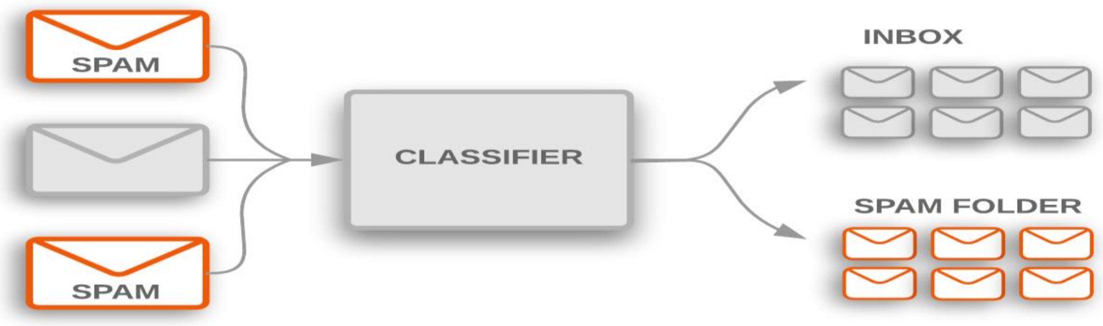
Sentence Classification

- It is the task where an NLP model receives a sentence and assigns some label to it.
- A spam filter is an application of sentence classification. It receives an email message and assigns whether or not it is spam.
- If you want to classify news articles into different topics (business, politics, sports, and so on), it's also a sentence-classification task.
- Sentence classification is one of the simplest NLP tasks that has a wide range of applications, including document classification, spam filtering, and sentiment analysis.
- With an emerging field of deep learning, performing complex operations has become faster and easier.
- The first step in sentence classification is to represent variable-length sentences using neural networks.
- Many modern NLP models use RNNs in some way.
- As studies NLP is a field of Artificial Intelligence in which we try to process human language as text or speech to make computers understand the language.
- To make a machine learn from the raw text we need to transform this data into a vector format which then can easily be processed by our computers. This transformation of raw text into a vector format is known as word representation.



- The first step in sentence classification is to represent variable-length sentences using neural networks (RNNs).

Recurrent neural networks (RNNs)

- Recurrent neural networks are one of the most important concepts in deep NLP.
- RNNs are a powerful and robust type because of their internal memory, RNNs can remember important things about the input they received, which allows them to be very precise in predicting what's coming next.

- This is why they’re the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms.
- Because of their internal memory, RNNs can remember important things about the input they received, which allows them to be very precise in predicting what’s coming next. This is why they’re the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms.

Why Recurrent Neural Networks?

RNN were created because there were a few issues in the feed-forward neural network:

- Cannot handle sequential data
- Considers only the current input
- Cannot memorize previous inputs

The solution to these issues is the RNN. An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory.

How Do Recurrent Neural Networks Work?

Sequential data is basically just ordered data in which related things follow each other.

The most popular type of sequential data is perhaps time series data, which is just a series of data points that are listed in time order.

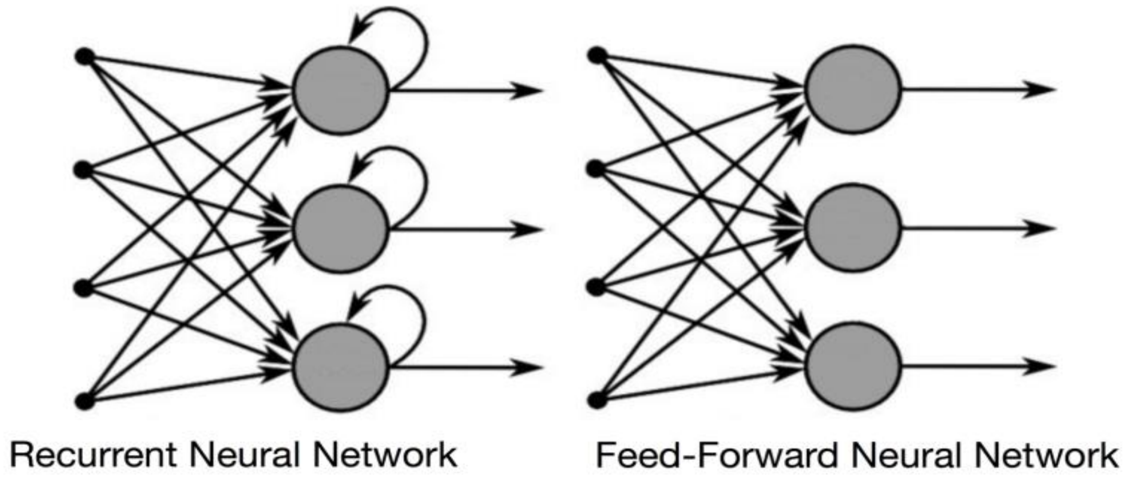
Recurrent vs. Feed-forward neural networks

RNNs and feed-forward neural networks get their names from the way they channel information.

In a feed-forward neural network, the information only moves in one direction — from the input layer, through the hidden layers, to the output layer. The information moves straight through the network.

Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, it has no notion of order in time. It simply can't remember anything about what happened in the past except its training.

In an RNN, the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously.



The two images below illustrate the difference in information flow between an RNN and a feed-forward neural network.

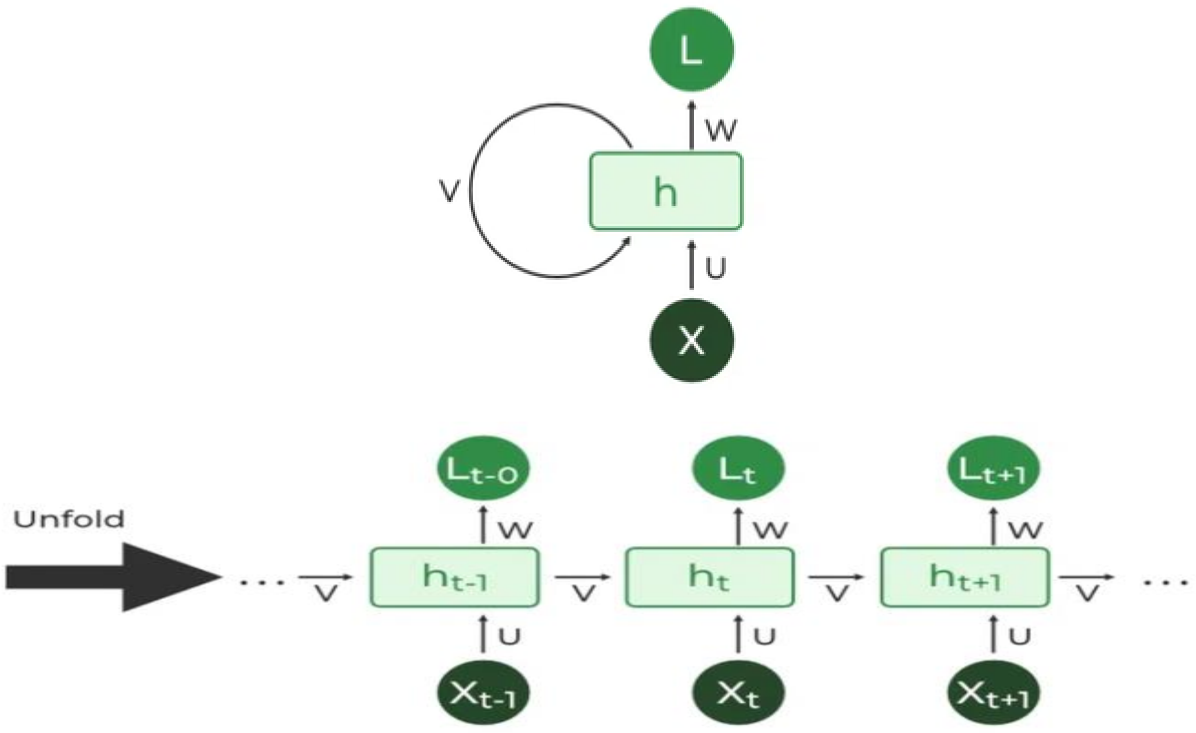
A recurrent neural network, however, is able to remember those characters because of its internal memory. It produces output, copies that output and loops it back into the network.

Simply put: Recurrent neural networks add the immediate past to the present.

Therefore, an RNN has two inputs: the present and the recent past. This is important because the sequence of data contains crucial information about what is coming next, which is why an RNN can do things other algorithms can't.

Recurrent Neuron and RNN Unfolding

The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a "Recurrent Neuron." This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing.



By unrolling we mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 3 words, the network would be unrolled into a 3-layer neural network, one layer for each word.

Recurrent Neural Networks and Backpropagation Through Time

To understand the concept of backpropagation through time (BPTT), you'll need to understand the concepts of forward and backpropagation first. We could spend an entire article discussing these concepts, so I will attempt to provide as simple a definition as possible.

WHAT IS BACKPROPAGATION?

Backpropagation (BP or backprop) is known as a workhorse algorithm in machine learning. Backpropagation is used for calculating the gradient of an error function with respect to a neural network's weights. The algorithm works its way backwards through the various layers of gradients to find the partial derivative of the errors with respect to the weights. Backprop then uses these weights to decrease error margins when training.

In neural networks, you basically do forward-propagation to get the output of your model and check if this output is correct or incorrect, to get the error. Backpropagation is nothing but going backwards through your neural network to find the partial derivatives of the error with respect to the weights, which enables you to subtract this value from the weights.

Those derivatives are then used by gradient descent, an algorithm that can iteratively minimize a given function. Then it adjusts the weights up or down, depending on which decreases the error. That is exactly how a neural network learns during the training process.

So, with backpropagation you basically try to tweak the weights of your model while training.

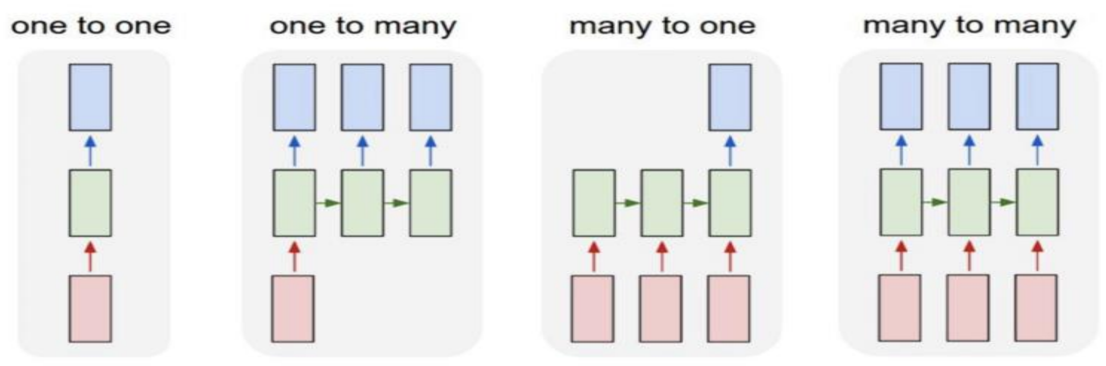
The image below illustrates the concept of forward propagation and backpropagation in a feed-forward neural network:

BPTT is basically just a fancy buzzword for doing backpropagation on an unrolled recurrent neural network. Unrolling is a visualization and conceptual tool, which helps you understand what’s going on within the network. Most of the time when implementing a recurrent neural network in the common programming frameworks, backpropagation is automatically taken care of, but you need to understand how it works to troubleshoot problems that may arise during the development process.

You can view an RNN as a sequence of neural networks that you train one after another with backpropagation.

TYPES OF RECURRENT NEURAL NETWORKS (RNNS)

- One to One
- One to Many
- Many to One
- Many to Many



One-to-one:

This is also called **Plain Neural networks**. It deals with a fixed size of the input to the fixed size of output, where they are independent of previous information/output.

Example: Image classification.

One-to-Many:

It deals with a fixed size of information as input that gives a sequence of data as output.

Example: Image Captioning takes the image as input and outputs a sentence of words.

Many-to-One:

It takes a sequence of information as input and outputs a fixed size of the output.

Example: sentiment analysis where any sentence is classified as expressing the positive or negative sentiment.

Many-to-Many:

It takes a Sequence of information as input and processes the recurrently outputs as a Sequence of data.

Example: Machine Translation, where the RNN reads any sentence in English and then outputs the sentence in French.

Advantages of Recurrent Neural Network

- RNN can model a sequence of data so that each sample can be assumed to be dependent on previous ones.

- A recurrent neural network is even used with convolutional layers to extend the active pixel neighborhood.

Disadvantages of Recurrent Neural Network

- Gradient vanishing and exploding problems.
- Training an RNN is a complicated task.
- It could not process very long sequences
- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) versions improve the RNN’s ability to handle long-term dependencies.

What are some variants of recurrent neural network architecture?

The RNN architecture laid the foundation for ML models to have language processing capabilities. Several variants have emerged that share its memory retention principle and improve on its original functionality.

- Bidirectional recurrent neural network (BRNN)
- Long short-term memory (LSTM)
- A gated recurrent unit (GRU)

Introduction

The first step in sentence classification is to represent variable-length sentences using neural networks (RNNs).

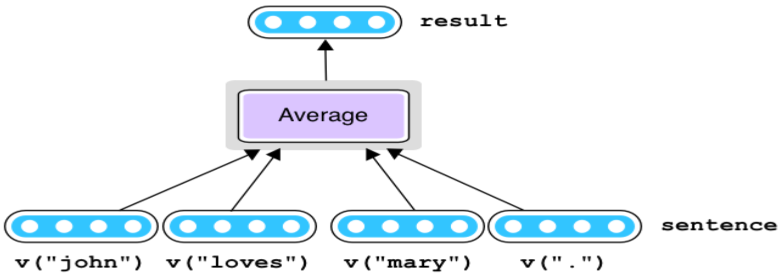
Handling variable-length input

- The Skip-gram network structure takes a word vector of a fixed size, runs it through a linear layer, and obtains a distribution of scores over all the context words. The structure and the size of the input, output, and the network are all fixed throughout the training.
- However, many, if not most, of what we deal with in NLP are sequences of variable lengths. For example, words, which are sequences of characters, can be short (“a,” “in”) or long (“internationalization”). Sentences (sequences of words) and documents (sequences of sentences) can be of any lengths.
- neural networks can handle only numbers and arithmetic operations. That was why we needed to convert words and documents to numbers through embeddings. We used linear layers to convert a fixed-length vector into another.
- But to do something similar with variable-length inputs, we need to figure out how to structure the neural networks so that they can handle them.

One idea is to first convert the input (e.g., a sequence of words) to embeddings, that is, a sequence of vectors of floating-point numbers, then average them.

Let’s assume the input sentence is sentence = ["john", "loves", "mary", "."]

result = (v("john") + v("loves") + v("mary") + v(".")) / 4



This method is quite simple and it’s used in many NLP applications, but it has one critical issue, which is that it can’t take word order into account. Because the order of input elements doesn’t affect the result of averaging, you’d get the same vector for both “Mary loves John” and “John loves Mary.”

Now, if we step back and reflect how we humans read language, this “averaging” is far from

the reality. We usually scan the sentence from the beginning, one word at a time, as we hold what the "partial" sentence means up until the part you're reading in our short-term memory. You maintain some sort of mental representation of the sentence as you read it.
Can we design a neural network structure that simulates this incremental reading of the input? The answer is a resounding yes. That structure is called Recurrent Neural Networks (RNNs), which I'll explain in detail below.

RNN abstraction

If you break down the reading process mentioned above, its core is the repetition of the following series of operations:

- Read a word
- Based on what has been read this far (your "mental state"), figure out what the word means
- Update the mental state
- Move on to the next word

Let's see how this works using a concrete example. If the input sentence is sentence = ["john", "loves", "mary", "."] and each word is already represented as a word embedding vector.

Also, let's denote your "mental state" as state, which is initialized by init_state(). Then, the reading process is represented by the following incremental operations:

```
state = init_state()
state = update(state, v("john"))
state = update(state, v("loves"))
state = update(state, v("mary"))
state = update(state, v("."))
```

The final value of state becomes the representation of the entire sentence from this process. Notice that if you change the order in which these words are processed (for example, by flipping "John" and "Mary"), the final value of state also changes, meaning that the state also

encodes some information about the word order.
pseudo-Python, it'd be like:

```
def rnn(words):
    state = init_state()
    for word in words:
        state = update(state, word)
    return state
```

Notice that there's state that gets initialized first and passed around during the iteration. For every input word, state is updated based on the previous state and the input using the function update. The network substructure corresponding to this step (the code block inside the loop) is called a cell. This stops when the input is exhausted, and the final value of state becomes the result of this RNN. See figure 2 for the illustration.

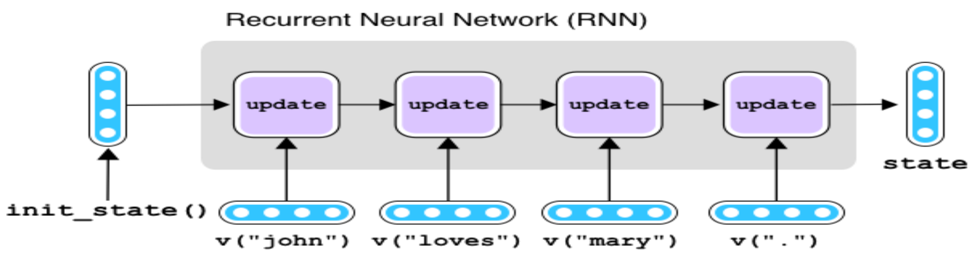


Figure 2: RNN abstraction

Now you see the parallelism here. When you're reading a sentence (sequence of words), your internal mental representation of the sentence, state, gets updated after reading each word. You can assume that the final state encodes the representation of the entire sentence.

The only remaining work is to design two functions — init_state() and update(). The state is usually initialized with zero (a vector filled with zeros), and you usually don't have to worry about how to go about defining the former. The more important issue is how you design update(), which determines the characteristics of the RNN.

Simple RNN and Nonlinearity

Here, we're going to implement update(), which is a function that takes two input variables and produces one output variable? After all, a cell is a neural network with its own input and output, right? The answer is yes, and it'd look like this:

```
def update_simple(state, word):
    return f(w1 * state + w2 * word + b)
```

An RNN defined by this type of the update function is called a **simple RNN.**

The function, called activation function or nonlinearity, takes a single input (or a vector) and transforms it (or every element of a vector) in a non-linear fashion.

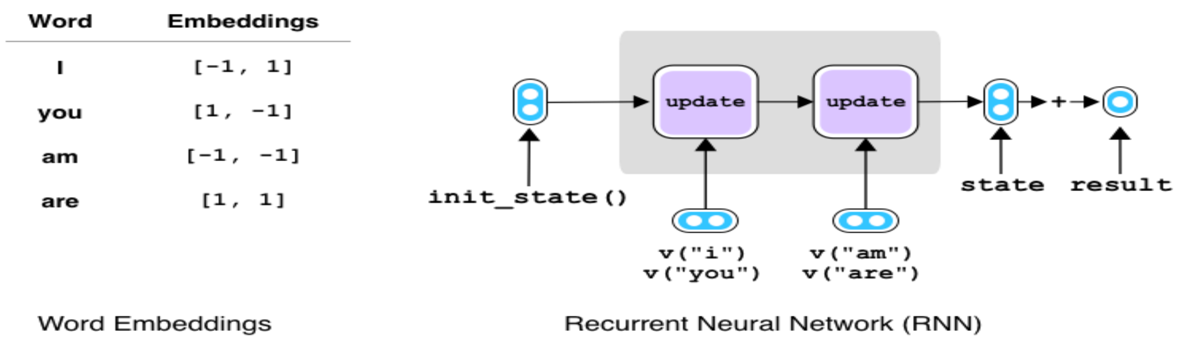
Lets imagine we are building an RNN that recognizes "grammatical" English sentences. Recognizing grammatical sentences from ungrammatical ones is a difficult NLP problem, which is a well-established research, but let's simplify it and only consider agreement between the subject and the verb.

Let's further simplify it and assume that there are only four words in this "language" — "I", "you", "am", and "are."

- If the sentence is either "I am" or "you are," it's grammatical.
- Other two combinations, "I are" and "you am," are incorrect.

What you want to build is an RNN that outputs 1 for these correct sentences as it produces 0 for these incorrect ones.

The first step in almost every modern NLP model is to represent words with embeddings. Embeddings are usually learned from a large dataset of natural language text, but we're going to give them some pre-defined values, as shown in figure 3.



Word	Embeddings
I	[-1, 1]
you	[1, -1]
am	[-1, -1]
are	[1, 1]

Figure 3: Recognizing grammatical English sentences using an RNN

<u>Without activation function:</u>

The update_simple() function above simplifies to:

def update_simple_linear(state, word):

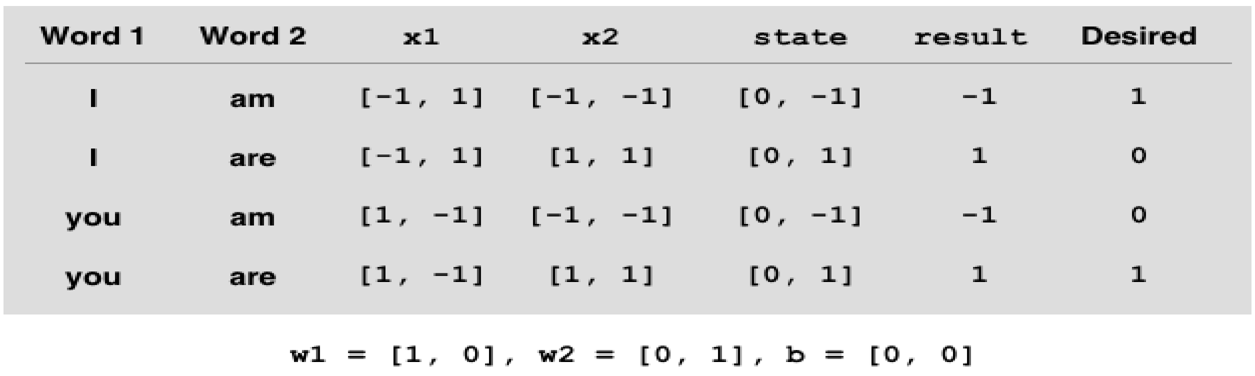
    return w1 * state + w2 * word + b

We assume the initial value of state are [0, 0], because the specific initial values aren't relevant to the discussion here.

- The RNN takes the first word embedding, x1, updates state, takes the second word embedding, x2, then produces the final state, which is a two-dimensional vector.
- Finally, the two elements in this vector are summed up and converted to result.
- If result is close to 1, the sentence is grammatical. Otherwise, it's not.

w1 * w2 * x1 + w2 * x2 + w1 * b + b

Remember, w1, w2, and b are parameters of the model (aka "magic constants") that need to be trained (adjusted). Here, instead of adjusting these parameters using a training dataset, let's assign some arbitrary values and see what happens. For example, when w1 = [1, 0], w2 = [0, 1], and b = [0, 0], the input and the output of this RNN is shown in figure 4.

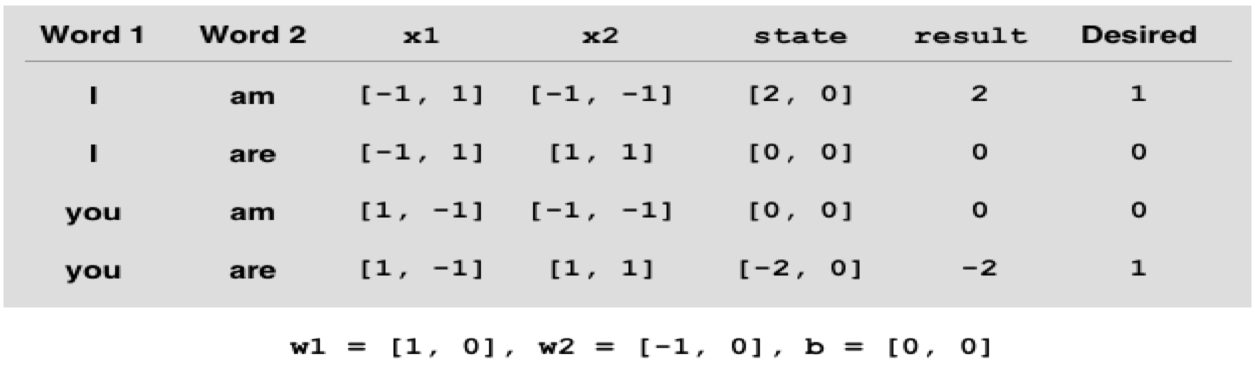


Word 1	Word 2	x1	x2	state	result	Desired
I	am	[-1, 1]	[-1, -1]	[0, -1]	-1	1
I	are	[-1, 1]	[1, 1]	[0, 1]	1	0
you	am	[1, -1]	[-1, -1]	[0, -1]	-1	0
you	are	[1, -1]	[1, 1]	[0, 1]	1	1

w1 = [1, 0], w2 = [0, 1], b = [0, 0]

Figure 4: Input and output when w1 = [1, 0], w2 = [0, 1], and b = [0, 0] without an activation function

If you look at the values of result, this RNN groups ungrammatical sentences (for example, "I are") with grammatical ones (for example, "you are"), which isn't the desired behavior. How about we try another set of values for the parameters? Let's use w1 = [1, 0], w2 = [-1, 0], and b = [0, 0] and see what happens (figure 5).



Word 1	Word 2	x1	x2	state	result	Desired
I	am	[-1, 1]	[-1, -1]	[2, 0]	2	1
I	are	[-1, 1]	[1, 1]	[0, 0]	0	0
you	am	[1, -1]	[-1, -1]	[0, 0]	0	0
you	are	[1, -1]	[1, 1]	[-2, 0]	-2	1

w1 = [1, 0], w2 = [-1, 0], b = [0, 0]

Figure 5: Input and output when w1 = [1, 0], w2 = [-1, 0], and b = [0, 0] without an activation function

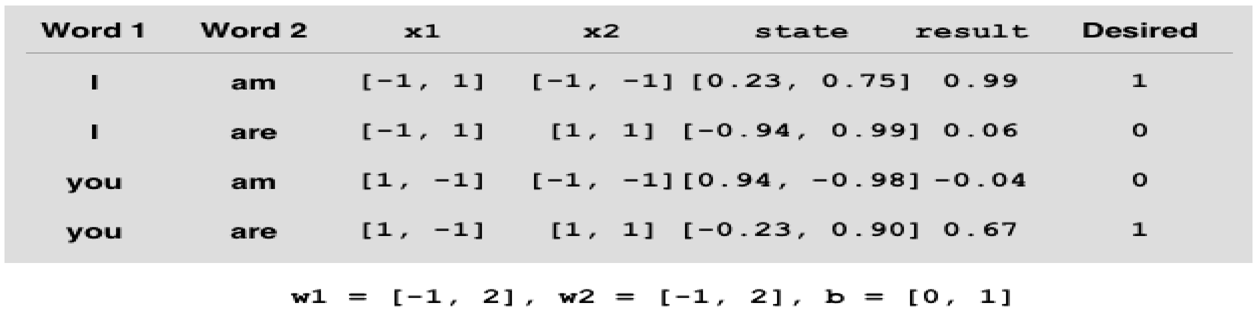
This is much better, because the RNN is successful in grouping ungrammatical sentences by assigning 0 to both "I are" and "you am." It also assigns completely opposite values (2 and -2) to grammatical sentences ("I am" and "you are").

we can't use this neural network to classify grammatical sentences from ungrammatical ones no matter how hard you try. No matter what values you assign to the parameters, this RNN can't produce results that are close enough to the desired values and are able to group sentences by their grammaticality.

With activation function:

Now, let’s put the activation function f() back and see what happens. The specific activation function we’ll use is called the hyperbolic tangent function, or more commonly, tanh, which is one of the most commonly used activation functions in neural networks. tanh doesn’t do much to the input when it’s close to zero, for example, 0.3 or -0.2. The input passes through the function almost unchanged. When the input is far from zero, tanh tries to squeeze it between -1 and 1.

When w1 = [-1, 2], w2 = [-1, 2], b = [0, 1], and the tanh activation function is used, the result of the RNN becomes a lot closer to what we desire (see figure 6). If you round them to the closest integers, the RNN successfully groups sentence by their grammaticality.



Word 1	Word 2	x1	x2	state	result	Desired
I	am	[-1, 1]	[-1, -1]	[0.23, 0.75]	0.99	1
I	are	[-1, 1]	[1, 1]	[-0.94, 0.99]	0.06	0
you	am	[1, -1]	[-1, -1]	[0.94, -0.98]	-0.04	0
you	are	[1, -1]	[1, 1]	[-0.23, 0.90]	0.67	1

w1 = [-1, 2], w2 = [-1, 2], b = [0, 1]

Figure 6: Input and output when w1 = [-1, 2], w2 = [-1, 2], and b = [0, 1] with an activation function

RNNs they’re trained like any other neural networks. The final outcome is compared with the desired outcome using the loss function, then the difference between the two, the loss, is used for updating the “magic constants.” The magic constants are, in this case, w1, w2, and b in the update_simple() function. Note that the update function and its magic constants are identical across all the timesteps in the loop. This means that what RNNs are learning is a general form of updates that can be applied to any situation.

The simple RNNs are rarely used in real-world NLP applications due to one problem called the vanishing gradients problem.

2.2.1 Vanishing gradients problem

Just like any programming language, if you know the length of the input, you can rewrite a loop without using one. An RNN can also be rewritten without using a loop, which makes it look just like a regular neural network with many layers. For example, if you know that there are only six words in the input, the rnn() from earlier can be rewritten as follows:

```
def rnn(sentence):
    word1, word2, word3, word4, word5, word6 = sentence
    state = init_state()
        state = update(state, word1)
    state = update(state, word2)
    state = update(state, word3)
    state = update(state, word4)
    state = update(state, word5)
    state = update(state, word6)
     return state
```

Representing RNNs without loops is called **unrolling.**

Now we know what update() looks like for a simple RNN (update_simple), so we can replace the function calls with their bodies, as shown here:

```
def rnn_simple(sentence):
    word1, word2, word3, word4, word5, word6 = sentence
    state = init_state()
    state = f(w1 * f(w1 * f(w1 * f(w1 * f(w1 * f(w1 * state + w2 * word1 + b) + w2 * word2 +
b) + w2 * word3 + b) + w2 * word4 + b) + w2 * word5 + b) + w2 * word6 + b)
    return state
```

Let’s say the input is sentence = ["The", "books", "I", "read", "yes- terday", "were"].

In this case, the innermost function call processes the first word “The,” the next one processes the second word “books,” and so on, all the way to the outermost function call, which processes “were.”

we rewrite the previous pseudocode slightly, as shown

```
def is_grammatical(sentence):
    word1, word2, word3, word4, word5, word6 = sentence
    state = init_state()
     state = process_main_verb(w1 *
        process_adverb(w1 *
            process_relative_clause_verb(w1 *
                process_relative_clause_subject(w1 *
                    process_main_subject(w1 *
                        process_article(w1 * state + w2 * word1 + b) +
                    w2 * word2 + b) +
                w2 * word3 + b) +
            w2 * word4 + b) +
        w2 * word5 + b) +
    w2 * word6 + b)
    return state
```

The deep learning literature calls this the vanishing gradients problem. A gradient is a mathematical term that corresponds to the message signal that each function receives from the next one that states how exactly they should improve their process (how to change their magic constants

Because of the vanishing gradients problem, simple RNNs are difficult to train and rarely used in practice nowadays.

Long short-term memory (LSTM)

The nested functions process information about grammar seems too inefficient. The outermost function cannot tell which function was responsible for which part of the message from only the final output.

so instead of passing the information through an activation function every time and changing its shape completely, how about adding and subtracting information relevant to the part of the sentence being processed at each step.

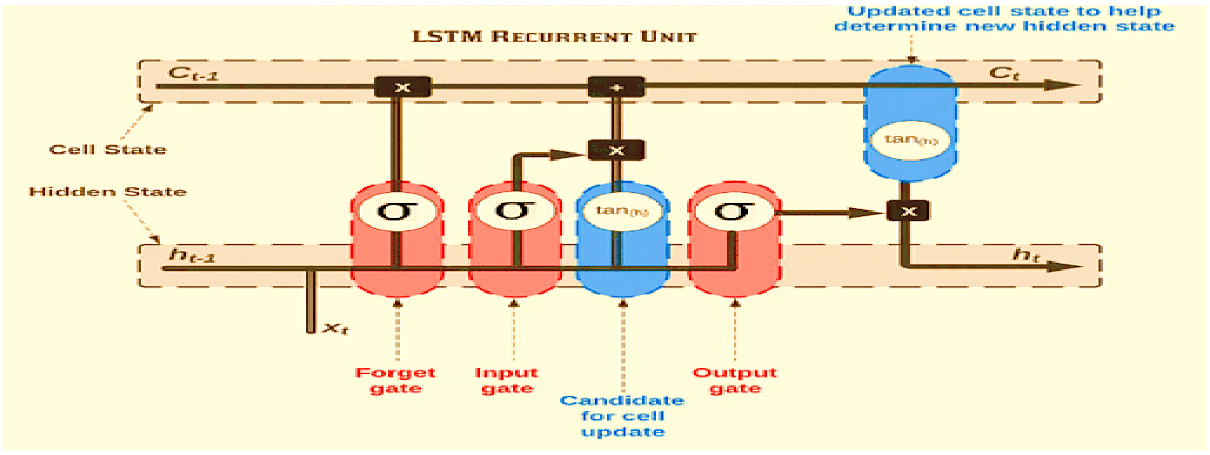
Long short-term memory units (LSTMs) are a type of RNN cell that is proposed based on this insight. Instead of passing around states, LSTM cells share a “memory” that each cell can remove old information from and/or add new information to.

Specifically, LSTM RNNs use the following function for updating states:

```
def update_lstm(state, word):
 cell_state, hidden_state = state
cell_state *= forget(hidden_state, word)
cell_state += add(hidden_state, word)
hidden_state = update_hidden(hidden_state, cell_state, word)
 return (cell_state, hidden_state)
```

Structure of LSTM

An LSTM (Long Short-Term Memory) network is a type of RNN recurrent neural network that is capable of handling and processing sequential data. The structure of an LSTM network consists of a series of LSTM cells, each of which has a set of gates (input, output, and forget gates) that control the flow of information into and out of the cell. The gates are used to selectively forget or retain information from *the previous time steps, allowing the LSTM to maintain long-term dependencies in the input data.*



A common LSTM unit is composed of a **cell**, an **input gate**, an **output gate** and a **forget gate**.

The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

- Forget gate decide what information to discard from a previous state by assigning a previous state, compared to a current input, a value between 0 and 1. A (rounded) value of 1 means to keep the information, and a value of 0 means to discard it.
- Input gates decide which pieces of new information to store in the current state, using the same system as forget gates.
- Output gates control which pieces of information in the current state to output by assigning a value from 0 to 1 to the information, considering the previous and current states. Selectively outputting relevant information from the current state allows the LSTM network to maintain useful, long-term dependencies to make predictions, both in current and future time-steps.

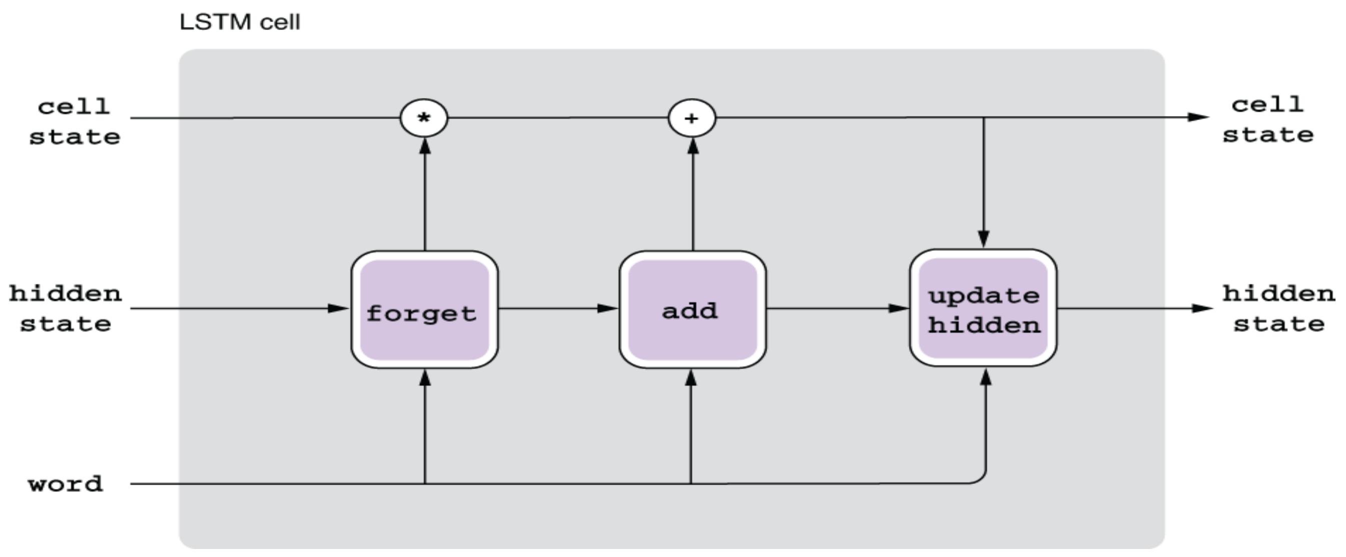
How do LSTMs Work?

The LSTM architecture is similar to RNN, but instead of the feedback loop it has an LSTM cell. The sequence of LSTM cells in each layer is fed with the **output of the last cell**. This enables the cell to get the previous inputs and sequence information. A cyclic set of steps happens in each LSTM cell

- The Forget gate is computed.
- The Input gate value is computed.
- The Cell state is updated using the above two outputs.
- The output(hidden state) is computed using the output gate.
- These series of steps occur in every LSTM cell. The intuition behind LSTM is that the Cell and Hidden states carry the previous information and pass it on to future time steps.
- The Cell state is **aggregated** with all the past data information and is the **long-term** information retainer. The Hidden state carries the output of the last cell, i.e. **short-term memory**. This combination of Long term and short-term memory techniques enables LSTM's to perform well In time series and sequence data.

The LSTM states comprise two halves—the cell state (the "memory" part) and the hidden state (the "mental representation" part).

- The function forget() returns a value between 0 and 1, so multiplying by this number means erasing old memory from cell_state. How much to erase is determined from hidden_state and word (input). Controlling the flow of information by multiplying by a value between 0 and 1 is called gating. LSTMs are the first RNN architecture that uses this gating mechanism.
- The function add()returns a new value added to the memory. The value again is determined from hidden_state and word.
- Finally, hidden_state is updated using a function, whose value is computed from the previous hidden state, the updated memory, and the input word



Because LSTMs have this cell state that stays constant across different timesteps unless explicitly modified, they are easier to train and relatively well behaved. Because you have a shared "memory" and functions are adding and removing information related to different parts of the input sentence, it is easier to pinpoint which function did what and what went wrong. The error signal from the outermost function can reach responsible functions more directly.

Advantages:

1. **Long-term dependency handling**: LSTMs are capable of learning long-term dependencies in sequential data, making them suitable for tasks where context over a long sequence is important.
2. **Gradient flow**: LSTMs use a gating mechanism to control the flow of information, allowing gradients to flow more easily during training and mitigating the vanishing gradient problem.

Disadvantages:

1. **Complexity**: LSTMs are more complex than traditional RNNs, which can make them harder to train and tune for optimal performance.
2. **Computational cost**: LSTMs can be more computationally expensive than simpler models, especially for large datasets or complex architectures.

What is Gated Recurrent Unit(GRU) ?

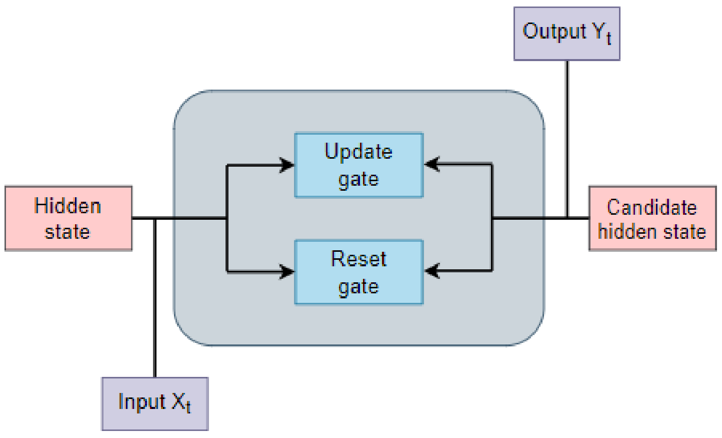
- GRU stands for Gated Recurrent Unit, which is a type of recurrent neural network (RNN) architecture that is similar to LSTM (Long Short-Term Memory).
- Like LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time. However, GRU has a simpler architecture than LSTM, with fewer parameters, which can make it easier to train and more computationally efficient.

The main difference between GRU and LSTM is the way they handle the memory cell state.

- In LSTM, the memory cell state is maintained separately from the hidden state and is updated using three gates: the input gate, output gate, and forget gate.
- In GRU, the memory cell state is replaced with a "candidate activation vector," which is updated using two gates: the reset gate and update gate.
- The reset gate determines how much of the previous hidden state to forget, while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state.
- Overall, GRU is a popular alternative to LSTM for modeling sequential data, especially in cases where computational resources are limited or where a simpler architecture is desired.

GRU architecture

- The architecture of the gated recurrent unit (GRU) is designed with two specific gates - the update gate and the reset gate.
- Each gate serves a unique purpose, significantly contributing to the GRU's high efficiency. The reset gate identifies short-term relationships, while the update gate recognizes long-term connections.



GRU architecture

The various components of the architecture are:

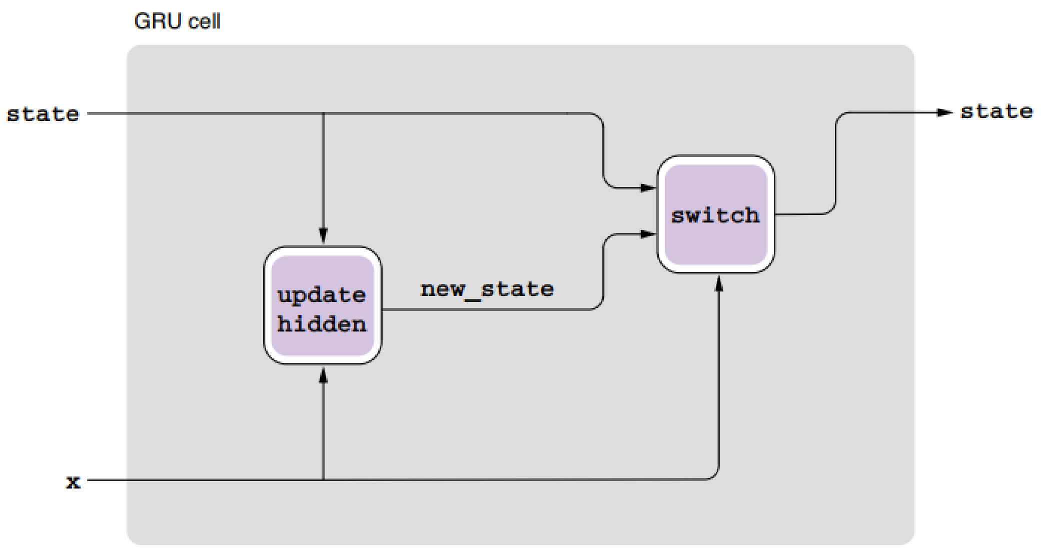
- **Update gate (Z):** Determines the degree of past information forwarded to the future.
- **Reset gate (R):** Decides the amount of past information to discard.
- **Candidate hidden state (H'):** Creates new representations, considering both the input and the past hidden state.
- **Final hidden state (H):** A blend of the new and old memories governed by the update gate.

The philosophy behind GRUs is similar to that of LSTMs, but GRUs use only one set of states instead of two halves. The update function for GRUs is shown next:

```
def update_gru(state, word):
    new_state = update_hidden(state, word)
    switch = get_switch(state, word)
    state = swtich * new_state + (1 – switch) * state
    return state
```

- Instead of erasing or updating the memory, GRUs use a switching mechanism. The cell first computes the new state from the old state and the input.
- It then computes switch, a value between 0 and 1.
- The state is chosen between the new state and the old one based on the value of switch. If it's 0, the old state passes through intact. If it's 1, it's overwritten by the new state.

- If it’s somewhere in between, the state will be a mix of two. See figure below for an illustration of the GRU update function.

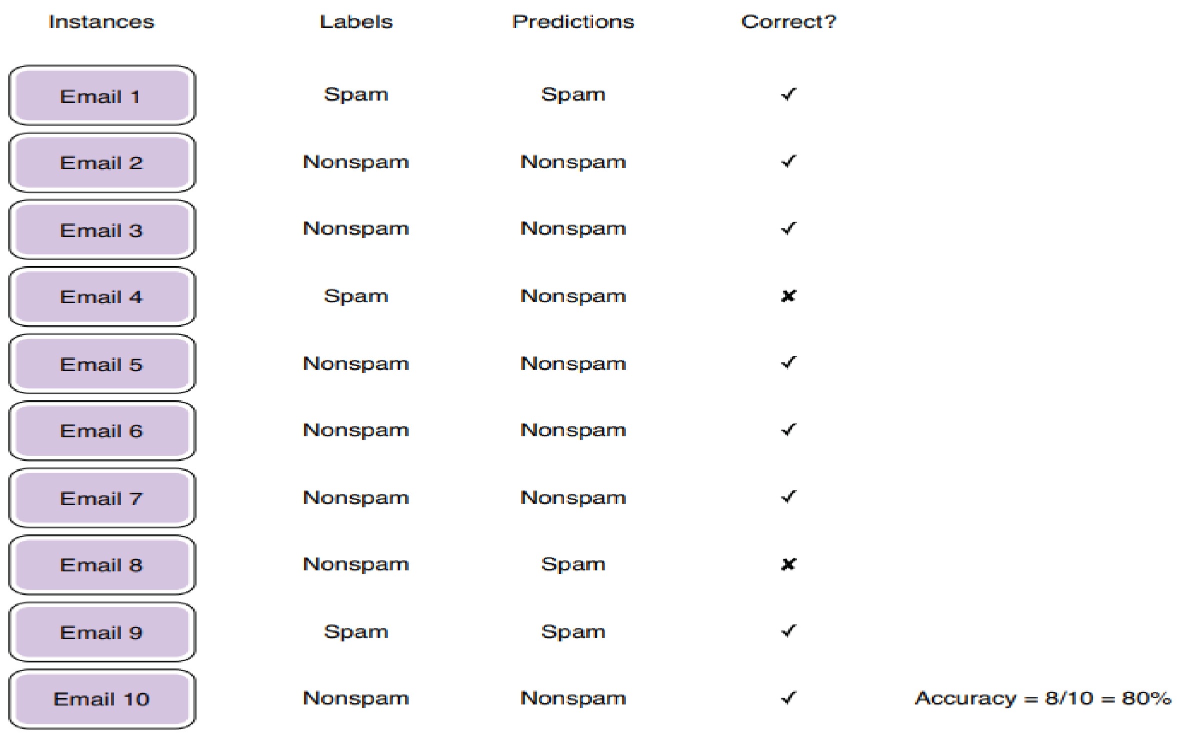


- Notice that the update function for GRUs is much simpler than that for the LSTMs. Indeed, it has fewer parameters (magic constants) that need to be trained compared to LSTMs. Because of this, GRUs are faster to train than LSTMs

Accuracy, Precision, Recall and F-measure

Accuracy

- Accuracy is probably the simplest of all the evaluation metrics.
- In a classification setting, accuracy is the fraction of instances that your model got right.
- For example, if there are 10 emails and your spam-filtering model got 8 of them correct, the accuracy of your prediction is 0.8, or 80%.



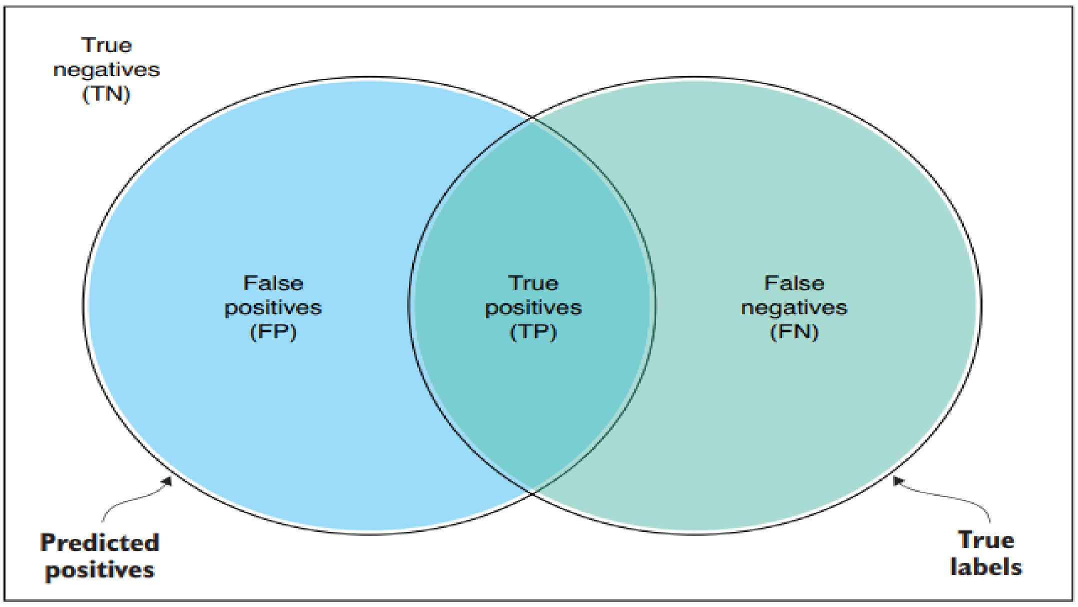
Instances	Labels	Predictions	Correct?	
Email 1	Spam	Spam	✓	
Email 2	Nonspam	Nonspam	✓	
Email 3	Nonspam	Nonspam	✓	
Email 4	Spam	Nonspam	✗	
Email 5	Nonspam	Nonspam	✓	
Email 6	Nonspam	Nonspam	✓	
Email 7	Nonspam	Nonspam	✓	
Email 8	Nonspam	Spam	✗	
Email 9	Spam	Spam	✓	
Email 10	Nonspam	Nonspam	✓	Accuracy = 8/10 = 80%

- Though simple, accuracy is not without its limitations.
- Specifically, accuracy can be misleading when the test set is imbalanced.
- An imbalanced dataset contains multiple class labels that greatly differ in their numbers.
- For example, if a spam-filtering dataset is imbalanced, it may contain 90% nonspam emails and 10% spams.

- In such case, even a stupid classifier that labels everything as nonspam would be able to achieve an accuracy of 90%.

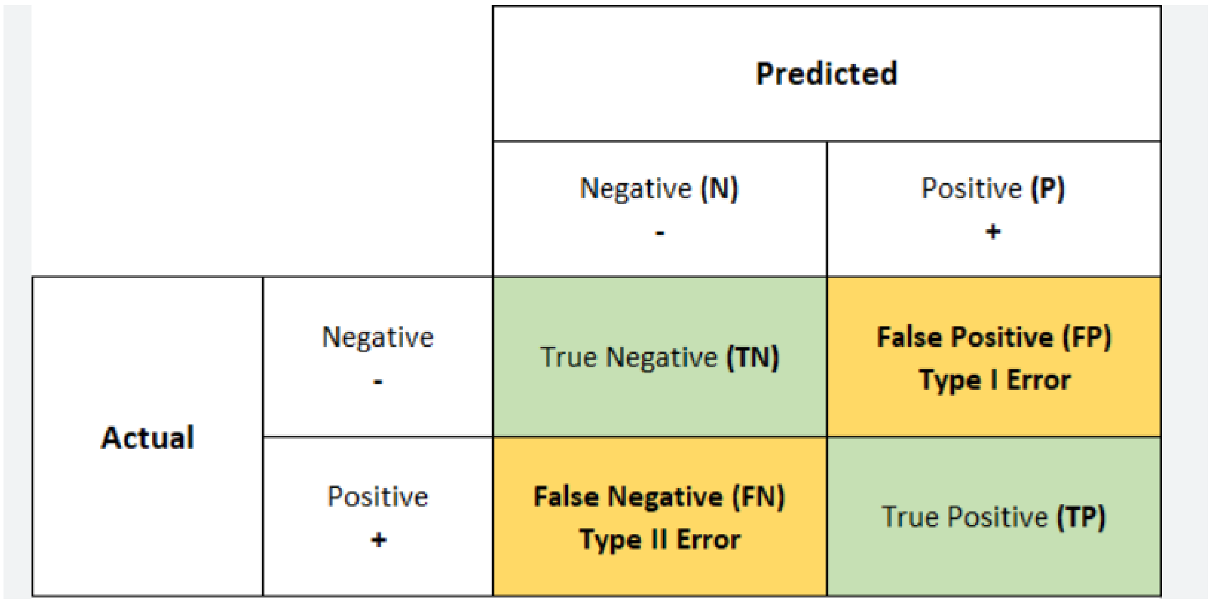
Precision and recall

- The rest of the metrics—precision, recall, and F-measure—are used in a binary classification setting.
- The goal of a binary classification task is to identify one class (called a positive class) from the other (called a negative class).
- In the spam-filtering setting, the positive class is spam, whereas the negative class is nonspam



$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN} \qquad \text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}$$

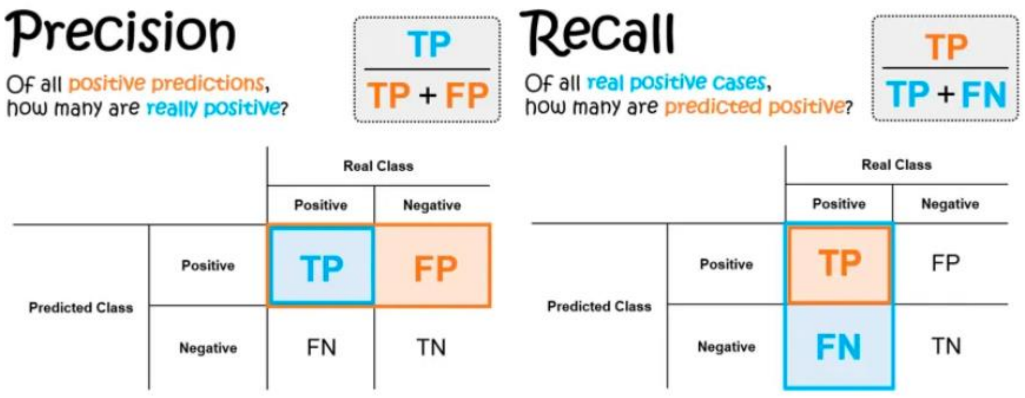
The Venn diagram in figure above contains four subregions: true positives, false positives, false negatives, and true negatives.



		Predicted	
		Negative (N) -	Positive (P) +
Actual	Negative -	True Negative (TN)	**False Positive (FP) Type I Error**
	Positive +	**False Negative (FN) Type II Error**	True Positive (TP)

- True positives (TP) are instances that are predicted as positive (= spam) and are indeed in the positive class.
- False positives (FP) are instances that are predicted as positive (= spam) but are actually not in the positive class. These are noises in the prediction, that is, innocent nonspam emails that are mistakenly caught by the spam filter and end up in the spam folder of your email client.
- On the other hand, false negatives (FN) are instances that are predicted as negative but are actually in the positive class. These are spam emails that slip through the spam filter and end up in your inbox. Finally, true negatives (TN) are instances that are predicted as negative and are indeed in the negative class (nonspam emails in your inbox).
- Precision is the fraction of instances that the model classifies as positive that are indeed correct. For example, if your spam filter identifies three emails as spam, and two of them are indeed spam, the precision will be 2/3, or about 66%.

- Recall is somewhat opposite of precision. It is the fraction of positive instances in your dataset that are identified as positive by your model. Again, using spam filtering as an example, if your dataset contains three spam emails and your model identifies two of them as spam successfully, the recall will be 2/3, or about 66%.



Precision
Of all positive predictions, how many are really positive?
TP / TP + FP
Recall
Of all real positive cases, how many are predicted positive?
TP / TP + FN
Real Class
Positive
Negative
Predicted Class
Positive
Negative
TP
FP
FN
TN

F-measure

- notice a tradeoff between precision and recall.
- Imagine there’s a spam filter that is very, very careful in classifying emails. It outputs only one out of several thousand emails as spam, but when it does, it is always correct. This is not a difficult task, because some spam emails are pretty obvious.
- Improving precision or recall alone while ignoring the other is not a good practice, because of the tradeoff between them.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Introducing sequential labeling

In sentence classification the task is to assign some label to a given sentence. Spam filtering, sentiment analysis, and language detection are some concrete examples of sentence classification.

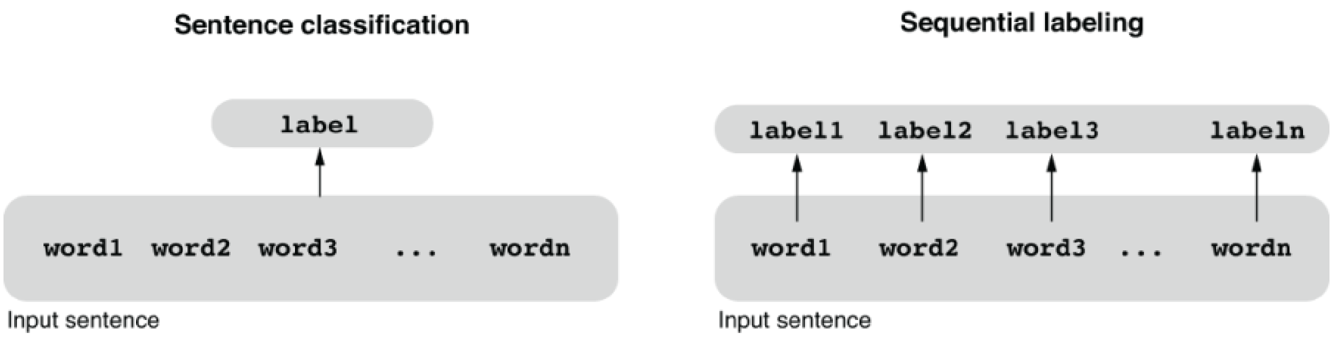
Although many real-world NLP problems can be formulated as a sentence-classification task, this method can also be quite limited, because the model, by definition, allows us to assign only a single label to the whole sentence.

But what if you wanted something more granular? For example, what if you wanted to do something with individual words, not just with the sentence? The most typical scenario you encounter is when you want to extract something from the sentence, which cannot be easily solved by sentence classification. This is where sequential labeling comes into play.

2.4.1 What is sequential labeling?

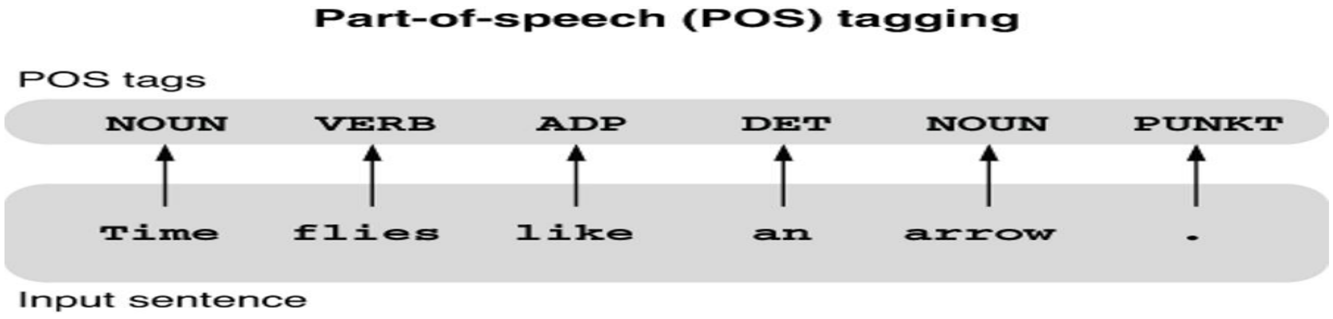
Sequential labeling is an NLP task where, given a sequence such as a sentence, the NLP system assigns a label to each element (e.g., word) of the input sequence. This con- trasts with sentence classification, where a label is assigned just to the input sentence.

Figure below illustrates this


Sentence classification
label
word1 word2 word3 ... wordn
Input sentence
Sequential labeling
label1 label2 label3 labeln
word1 word2 word3 ... wordn
Input sentence

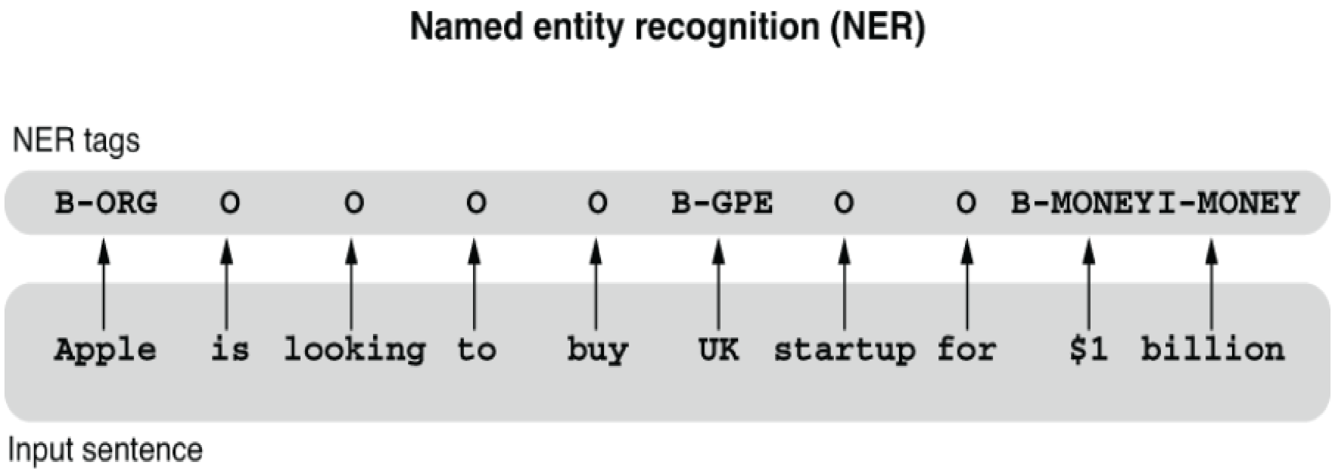
When do we need a label per word?

<u>Case 1:</u> A typical scenario where sequential labeling comes in handy is when you want to analyze a sentence and produce some linguistic information per word. For example, part-of-speech (POS) tagging which produces a POS tag such as nouns, verbs, and prepositions for each word in the input sentence and is a perfect match for sequential labeling. See figure below for an illustration.



POS tagging is one of the most fundamental, important NLP tasks. Many English words (and words in many other languages as well) are ambiguous, meaning that they have multiple possible interpretations. For example, the word "book" can be used to describe a physical or electronic object consisting of pages ("I read a book") or an action for reserving something ("I need to book a flight").
POS tagging is an important first step toward solving this ambiguity.

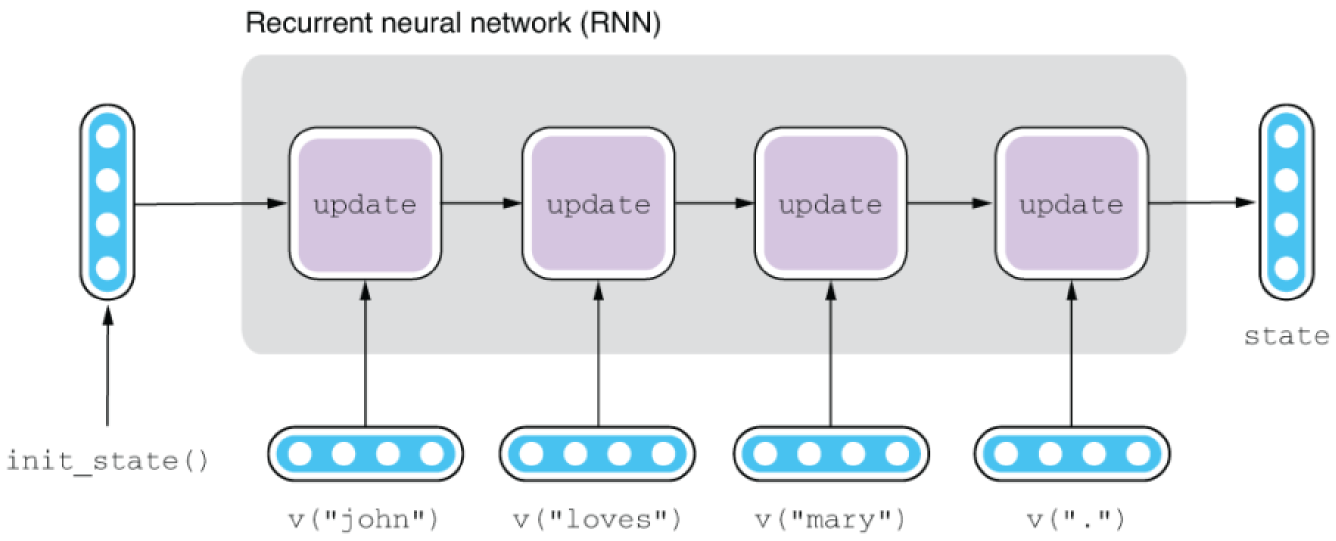
Case 2: Another scenario is when you want to extract some pieces of information from a sentence.
An example of this is **named entity recognition (NER),** which is a task to identify mentions to real-world entities, such as proper nouns and numerical expressions, from a sentence illustrated in figure below.



2.4.2 Using RNNs to encode sequences

In sentence classification, we used recurrent neural networks (RNNs) to convert an input of variable length into a fixed-length vector. The fixed-length vector, which is converted to a set of "scores" by a linear layer, captures the information about the input sentence that is necessary for deriving the sentence label. what RNN does can be represented by the following pseudocode and the diagram shown in figure below:

```
def rnn_vec(words):
state = init_state()
            for word in words:
state = update(state, word)
return state
```



What kind of neural network could be used for sequential tagging? We seem to need some information for every input word in the sentence, not just at the end. If you look at the pseudocode for rnn_vec() carefully, you can notice that we already have information for every word in the input, which is captured by state. The function just happens to return only the final value of state, but there is no reason we can't store intermediate values of state and return them as a list instead, as in the following function:

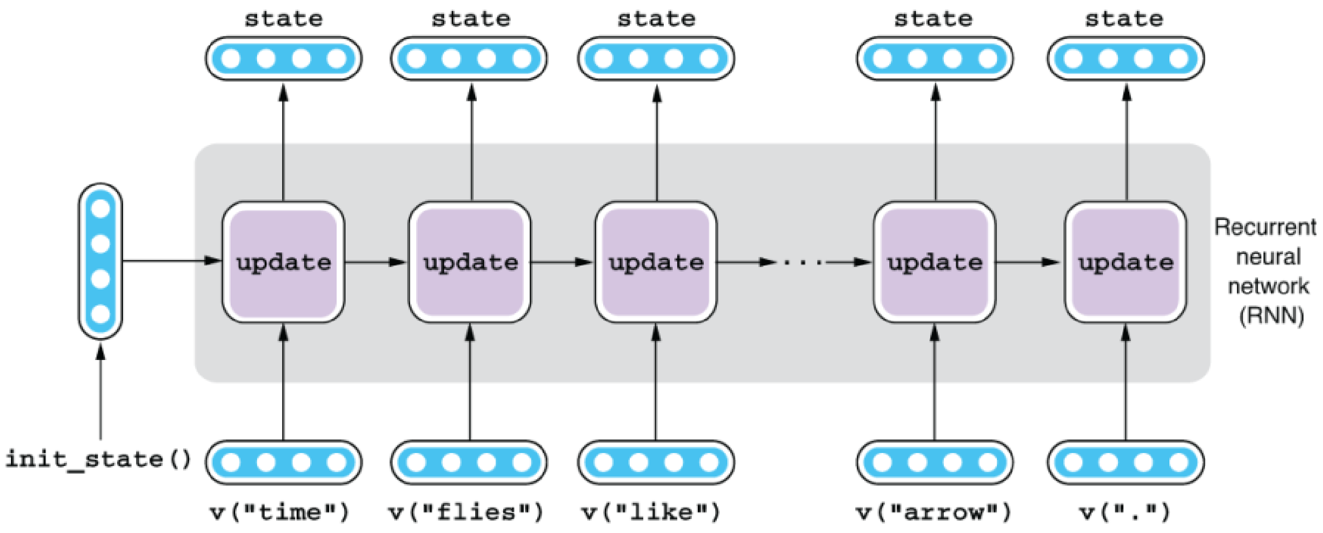
```
def rnn_seq(words):
 state = init_state()
 states = []
for word in words:
state = update(state, word)
states.append(state)
return states
```

If we apply this function to the "time flies" example that is, write it without using a loop—it will look like the following:

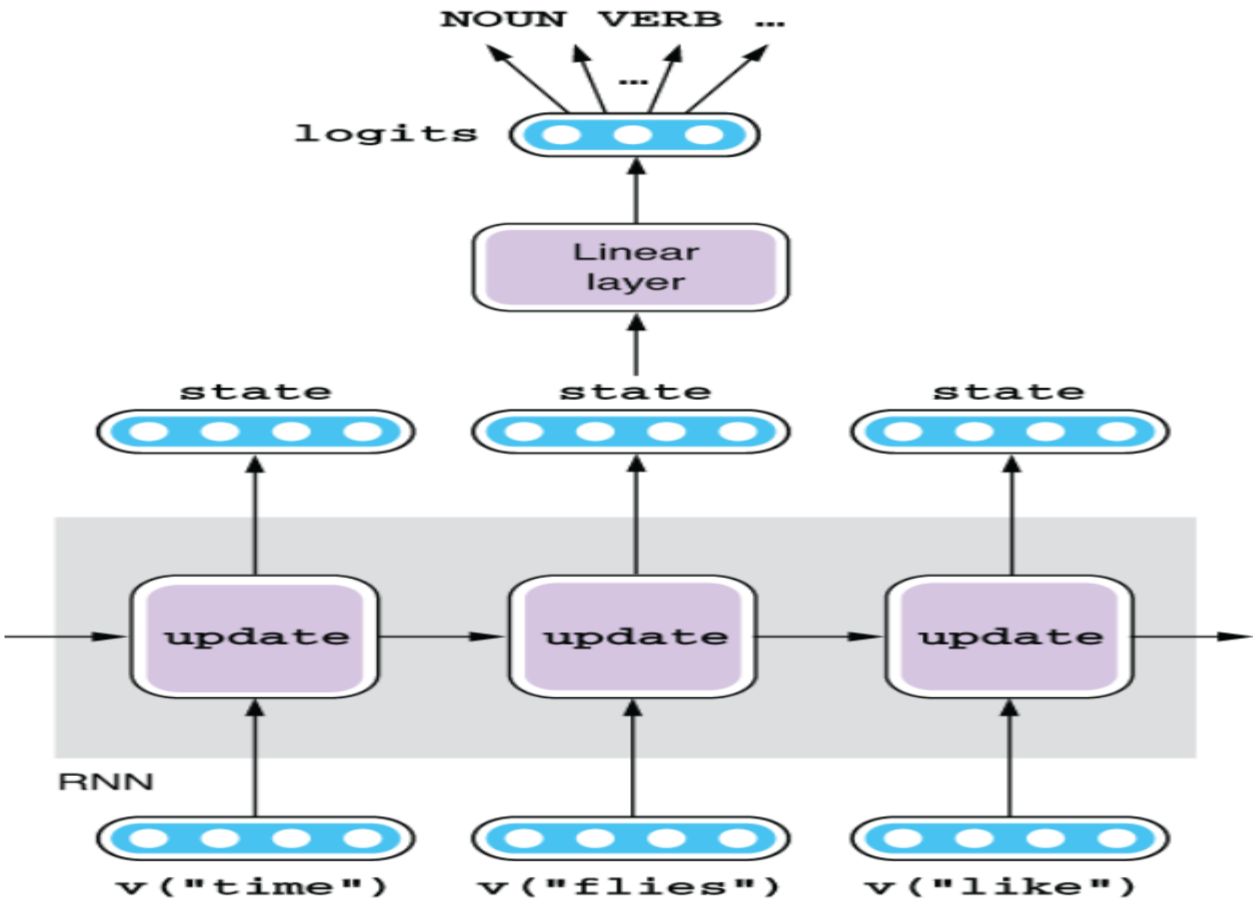
```
state = init_state()
```

```
states = []
state = update(state, v("time")) states.append(state)
state = update(state, v("flies")) states.append(state)
state = update(state, v("like")) states.append(state)
state = update(state, v("an")) states.append(state)
state = update(state, v("arrow")) states.append(state)
state = update(state, v(".")) states.append(state)
```

Note that v() here is a function that returns the embedding for the given word. This can be visualized as shown in figure below.



Notice that for each input word, the network produces the corresponding state that captures some information about word. The length of the list states is the same as that of words.
If we think of this RNN as a black box, it takes a sequence of something (e.g., word embeddings) and converts it to a sequence of vectors that encode some information about individual words in the input, so this architecture is called a Seq2Seq (for “sequence-to-sequence”) encoder.


NOUN VERB ...
...
logits
Linear layer
state
state
state
update
update
update
RNN
v("time")
v("flies")
v("like")

BUILDING POS TAGGER

What is POS Tagging?

- Part-of-speech (POS) tagging is the process of assigning a part-of-speech tag to each word in a given text. These tags represent the syntactic category of a word such as noun, verb, adjective, adverb, etc.
- POS tagging is a crucial step in many NLP tasks because it helps in understanding the grammatical structure of sentences, which is essential for accurate text analysis and understanding.

Why is POS Tagging Important?

POS tagging serves as the foundation for many NLP tasks including:

- Syntactic parsing: POS tags help in parsing sentences into their grammatical components such as subject, verb, object, etc.
- Named entity recognition (NER): POS tags provide information about the word's role in a sentence, aiding in the identification of named entities like people, organizations, locations, etc.
- Sentiment analysis: POS tags can provide insights into the sentiment expressed by different parts of a sentence. For example, adjectives and adverbs may indicate the sentiment of the text.

POS Tagging Process:

The process of POS tagging typically involves the following steps:

- **Tokenization**: The input text is split into individual words or tokens.
- **Word Normalization**: Words are normalized to a standard form (e.g., converting all words to lowercase).
- **Tagging**: Each word is assigned a POS tag based on its context and grammatical role in the sentence.

POS Tagging Libraries:

- There are several libraries and tools available for POS tagging in various programming languages. In Python, the Natural Language Toolkit (NLTK) and spaCy are popular libraries for NLP tasks including POS tagging.

POS TAGGING USING SPACY

```
import spacy
# Load the English language model
nlp = spacy.load("en_core_web_sm")
# Sample sentence
sentence = "POS tagging is essential for natural language processing."
# Process the sentence using spaCy
doc = nlp(sentence)
# Extract POS tags
pos_tags = [(token.text, token.pos_) for token in doc]
# Print the POS tags
print(pos_tags)
```

Explanation of the Code:

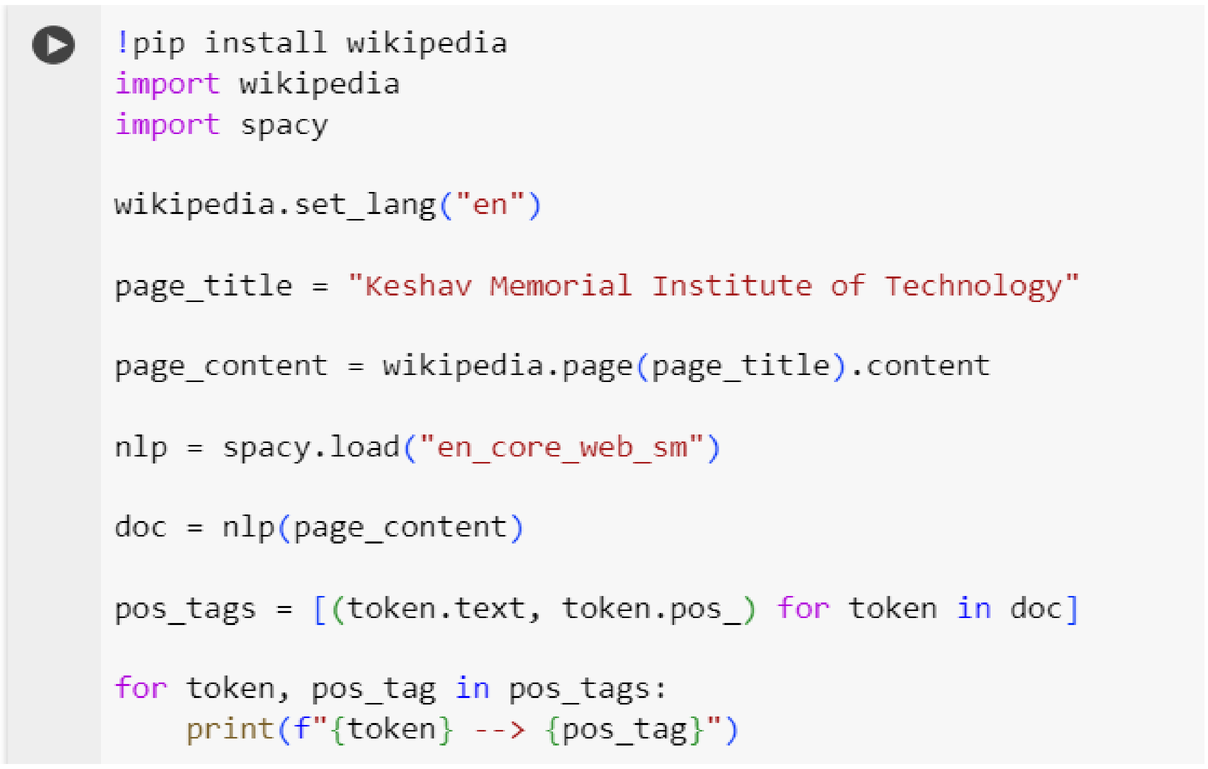
- We start by importing the spaCy library.
- We load the English language model (**en_core_web_sm**) provided by spaCy. This model contains pre-trained word vectors and POS tagger.
- We define a sample sentence to be tagged.
- The sentence is processed using spaCy's **nlp()** function, which tokenizes the sentence and performs various linguistic annotations including POS tagging.
- For each token in the processed document (**doc**), we extract its text and POS tag using the **text** and **pos_** attributes respectively.
- We store the word and its corresponding POS tag in a list of tuples (**pos_tags**).

- Finally, we print the list of tuples containing the words and their corresponding POS tags.

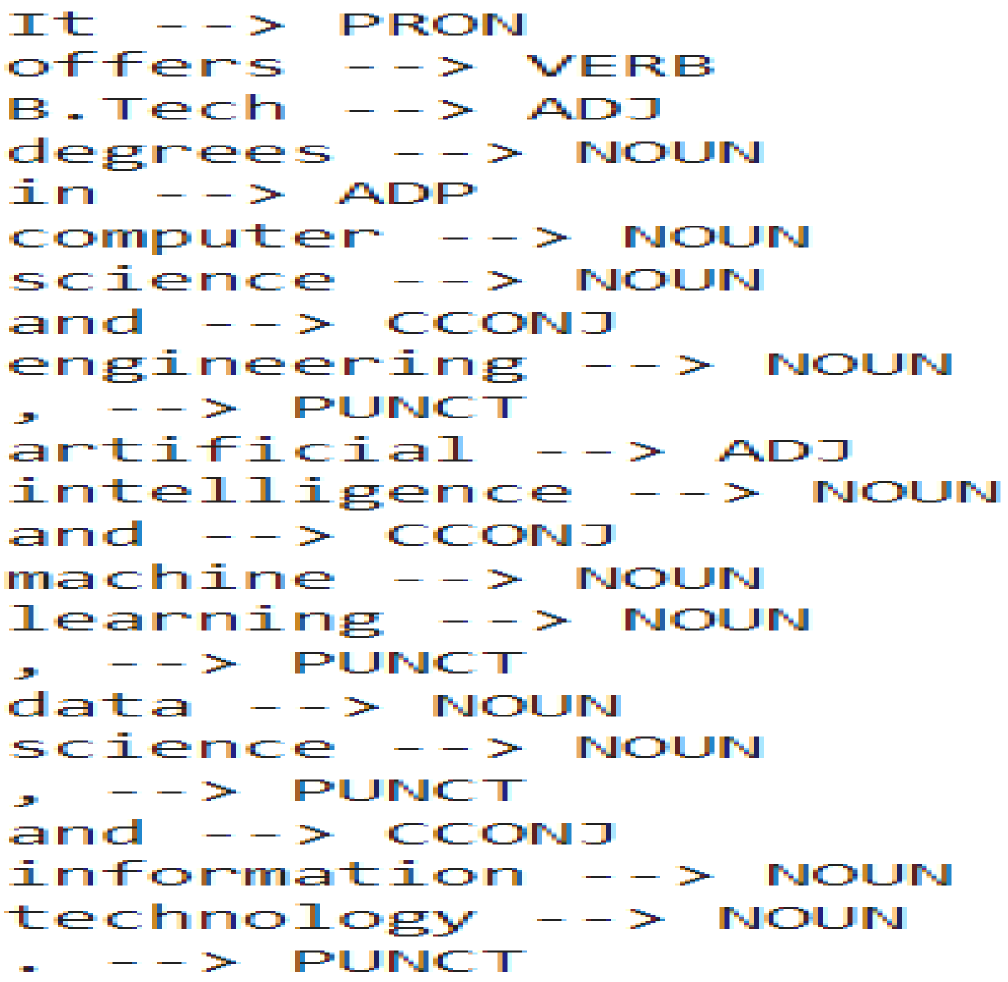
Output:

The output will be a list of tuples, where each tuple contains a word and its corresponding POS tag, similar to the NLTK example.

Using spaCy for POS tagging provides robust linguistic annotations and supports a wide range of languages and models.



```
!pip install wikipedia
import wikipedia
import spacy

wikipedia.set_lang("en")

page_title = "Keshav Memorial Institute of Technology"

page_content = wikipedia.page(page_title).content

nlp = spacy.load("en_core_web_sm")

doc = nlp(page_content)

pos_tags = [(token.text, token.pos_) for token in doc]

for token, pos_tag in pos_tags:
    print(f"{token} --> {pos_tag}")
```



```
It --> PRON
offers --> VERB
B.Tech --> ADJ
degrees --> NOUN
in --> ADP
computer --> NOUN
science --> NOUN
and --> CCONJ
engineering --> NOUN
, --> PUNCT
artificial --> ADJ
intelligence --> NOUN
and --> CCONJ
machine --> NOUN
learning --> NOUN
, --> PUNCT
data --> NOUN
science --> NOUN
, --> PUNCT
and --> CCONJ
information --> NOUN
technology --> NOUN
. --> PUNCT
```

POS TAGGING USING NLTK

```python
import nltk

nltk.download('punkt')
nltk.download()
from nltk.tokenize import word_tokenize

# Sample sentence
sentence = "POS tagging is essential for natural language processing."

# Tokenize the sentence into words
words = word_tokenize(sentence)

# Perform POS tagging
pos_tags = nltk.pos_tag(words)

# Print the POS tags
print(pos_tags)
```

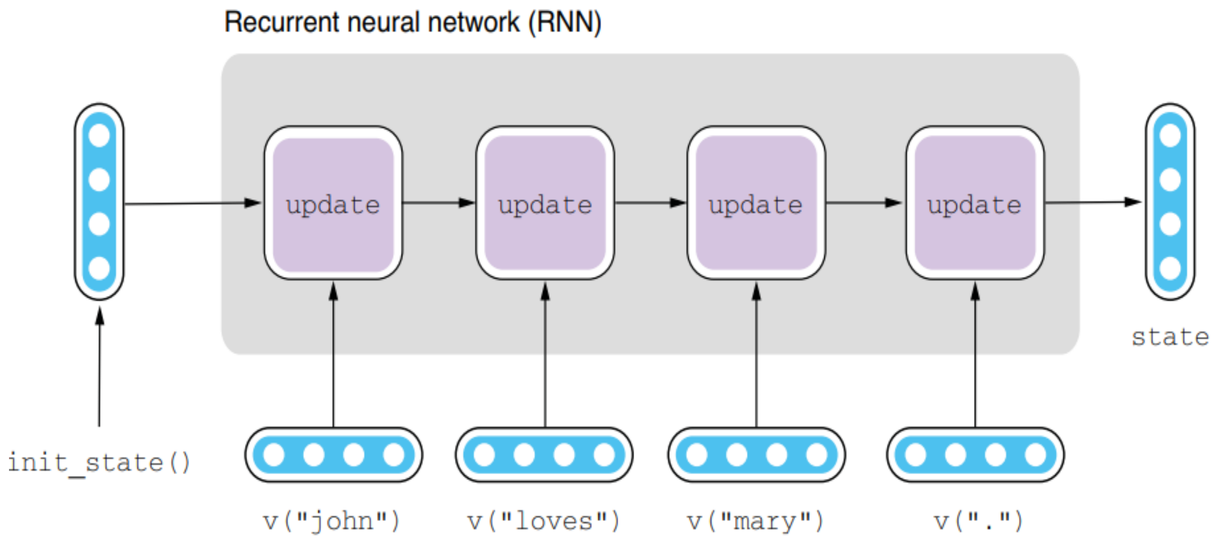
- We start by importing necessary modules from NLTK.
- We define a sample sentence to be tagged.
- The **word_tokenize()** function is used to tokenize the sentence into individual words.
- **pos_tag()** function is applied to the tokenized words, which assigns a POS tag to each word based on its context.
- Finally, we print the list of tuples containing the words and their corresponding POS tags.

```
output:

[('POS', 'NNP'), ('tagging', 'NN'), ('is', 'VBZ'), ('essential', 'JJ'), ('for', 'IN'), ('natural', 'JJ'), ('language', 'NN'), ('processing', 'NN'), ('.', '.')]
```

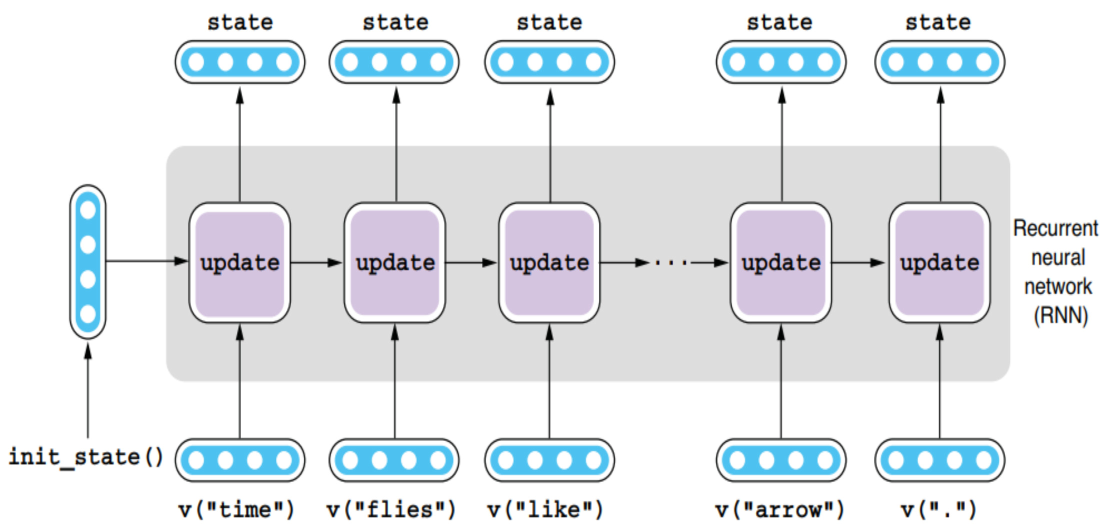
MULTILAYER AND BIDIRECTIONAL RNNs

Multilayer and bidirectional RNNs

RNNs are a powerful tool for building NLP applications but multilayer and bidirectional RNNs are even more powerful components for building highly accurate NLP applications.



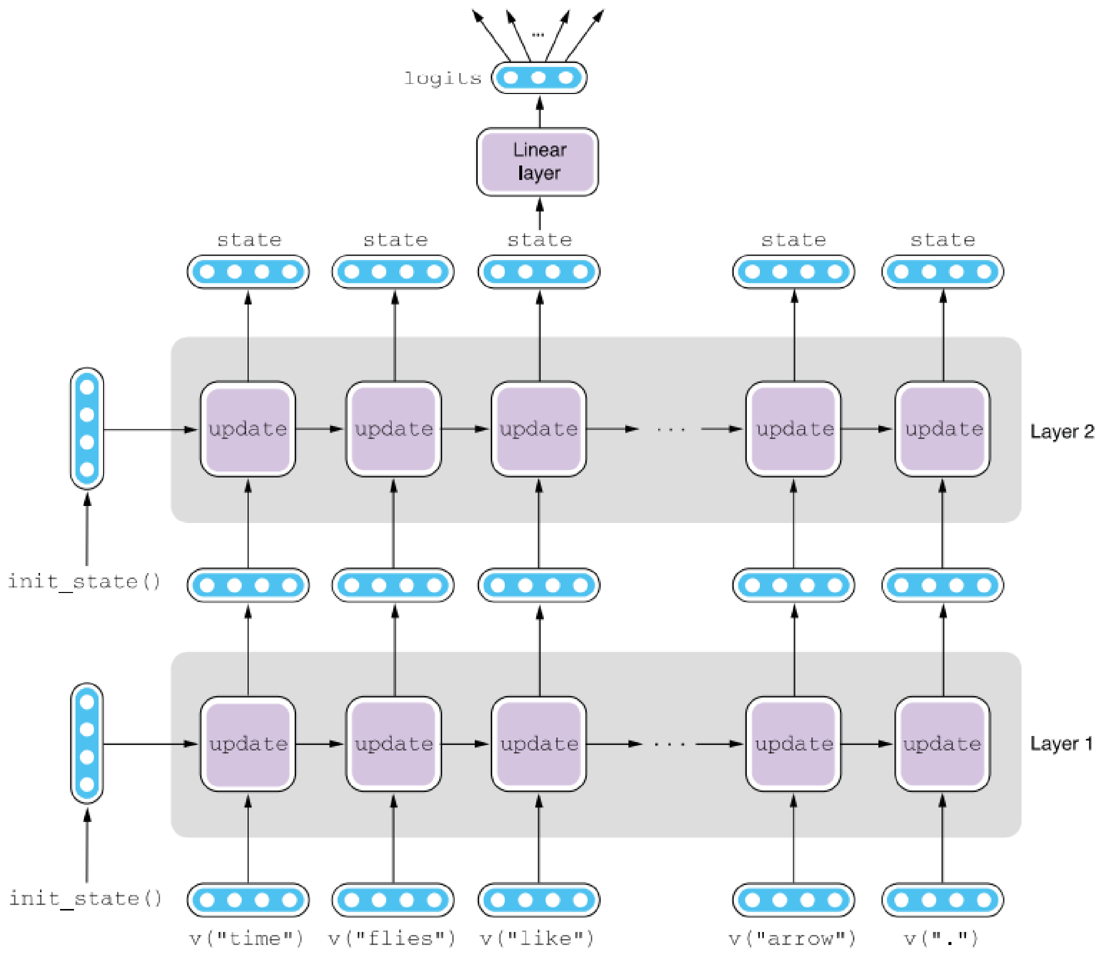
Recurrent neural network (RNN) for sentence classification



Recurrent neural network (RNN) for sequential labeling

Multilayer RNNs

If you look at an RNN as a black box, it is a neural network structure that converts a sequence of vectors (word embeddings) into another sequence of vectors (hidden states). The input and output sequences are of the same length, usually the number of input tokens. This means that you can repeat this "encoding" process multiple times by stacking RNNs on top of each other. The output (hidden states) of one RNN becomes the input of another RNN that is just above the previous one. A substructure (such as a single RNN) of a bigger neural network is called a layer, because you can stack them together like layers. The structure of a two-layered RNN is shown in figure below



If we think of a layer of RNN as a machine that takes in something concrete (e.g., word embeddings) and extracts some abstract concepts (e.g., scores for POS tags), you can expect that, by repeating this process, RNNs are able to extract increasingly more abstract concepts as the number of layers increases. Although not fully theoretically proven, many real-world NLP applications use multilayer RNNs.

Bidirectional RNNs

We have been feeding words to RNNs as they come in—from the beginning of the sentence to the end. This means that when an RNN is processing a word, it can leverage only the information it has encountered so far, which is the word's left context.

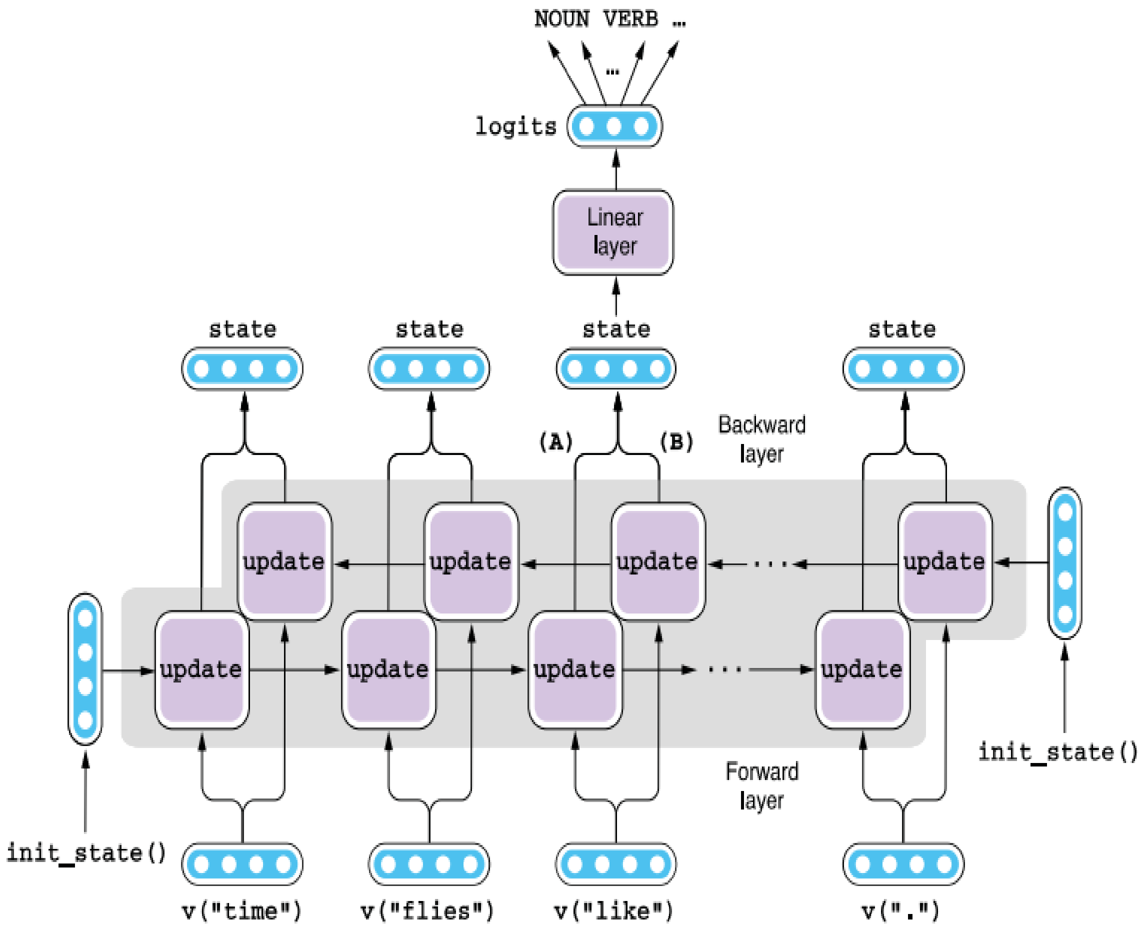
True, you can get a lot of information from a word's left context. For example, if a word is preceded by a modal verb (e.g., "can"), it is a strong signal that the next word is a verb.

However, the right context holds a lot of information as well. For example, if you know that the next word is a determiner (e.g., "a"), it is a strong signal that "book" on its left is a verb, not a noun.

Bidirectional RNNs (or simply biRNNs) solve this problem by combining two RNNs with opposite directions.

- A forward RNN is a forward-facing RNN that we've been using so far in this book—it scans the input sentence left to right and uses the input word and all the information on its left to update the state.
- A backward RNN, on the other hand, scans the input sentence right to left. It uses the input word and all the information on its right to update the state.

This is equivalent to flipping the order of the input sentence and feeding it to a forward RNN. The final hidden states produced by biRNNs are concatenations of hidden states from the forward and backward RNNs. See figure below for an illustration.



Assume the input sentence is **"time flies like an arrow"** and you'd like to know the POS tag for the word "like" in the middle of this sentence.

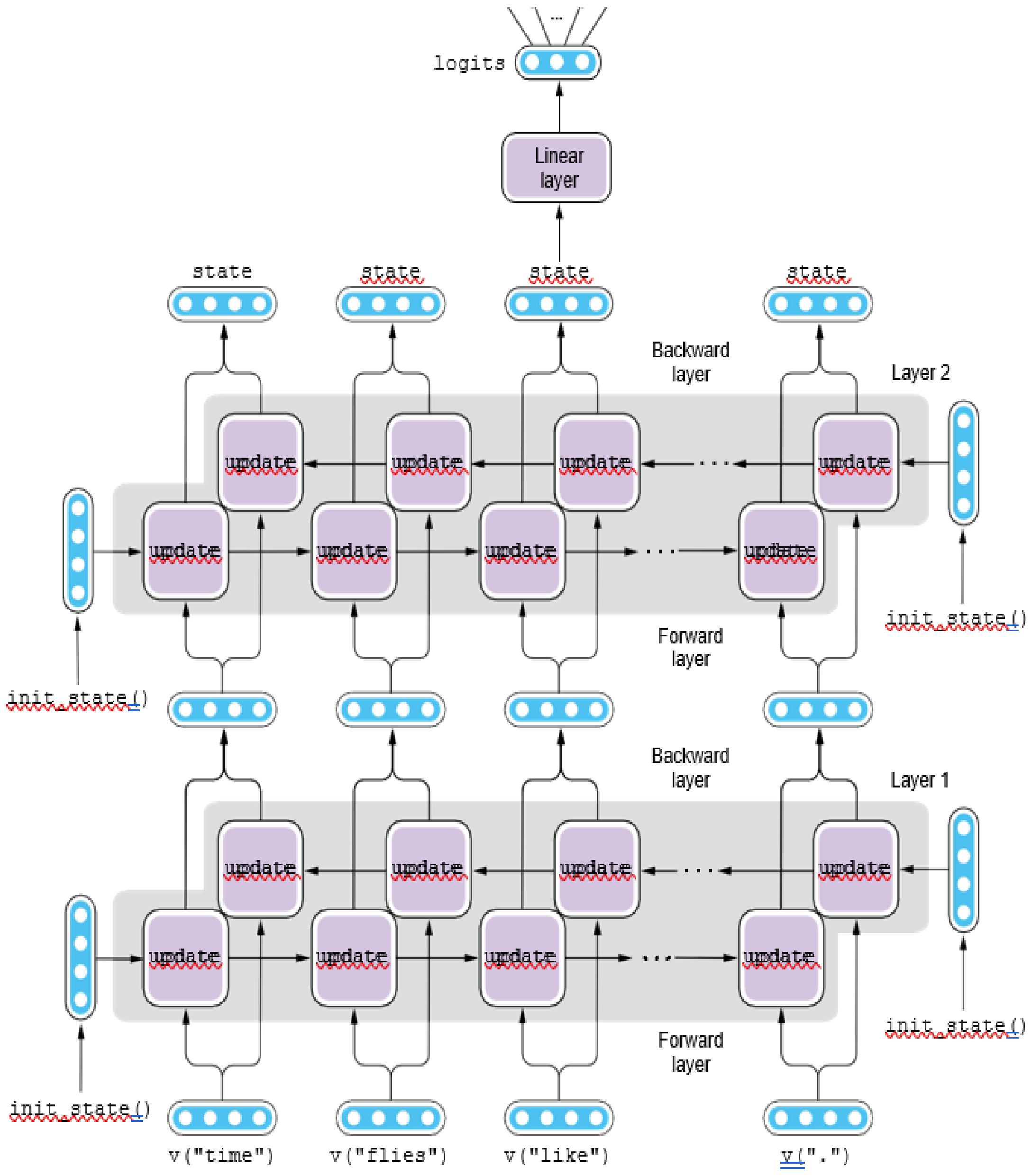
- The forward RNN processes "time" and "flies," and by the time it reaches "like," its internal state encodes all the information about "time flies like."
- Similarly, the backward RNN processes "arrow" and "an," and by the time it reaches "like," the internal state has encoded all the information about "like an arrow."

The internal state from the biRNN for "like" is the concatenation of these two states (A + B).

we literally stitch together two vectors—no mathematical operations involved. As a result, the internal state for “like” encodes all the information from the entire sentence. This is a great improvement over just knowing half the sentence!

we can combine the two techniques introduced in this section by stacking bidirectional RNNs.

The output from one layer of biRNN (concatenation of a forward and a backward layer) becomes the input to another layer of biRNN as shown in figure below.



Named entity recognition

Named Entity Recognition (NER) is a task in natural language processing (NLP) that involves identifying and classifying named entities in text into predefined categories such as person names, organization names, locations, dates, and more. NER is a crucial step in many NLP applications, including information extraction, question answering, and text summarization.

Given a piece of text, the goal of NER is to locate and classify named entities mentioned in the text into predefined categories such as:

- Person: Individual people's names
- Organization: Names of companies, institutions, etc.
- Location: Geographical locations such as cities, countries, etc.
- Date: Specific points in time mentioned in the text
- Time: Time expressions mentioned in the text
- Money: Monetary expressions such as currency amounts

What is named entity recognition?

Named entities are mentions of real-world entities such as proper nouns. Common named entities that are usually covered by NER systems include the following:

- Personal name (PER): Alan Turing, Lady Gaga, Elon Musk
- Organization (ORG): Google, United Nations, Giants
- Location (LOC): Mount Rainer, Bali Island, Nile
- Geopolitical entity (GPE): UK, San Francisco, Southeast Asia

However, different NER systems deal with different sets of named entities.

However, different NER systems deal with different sets of named entities. The concept of named entities is a bit overloaded in NLP to mean any mentions that are of interest to the application’s user.

Identifying named entities is in itself important, because named entities (who, what, where, when, and so on) are often what most people are interested in. But NER is also an important first step for many other NLP applications.

- One such task is relation extraction: extracting all relations between named entities from the given docu- ment. For example, given a press release document, you may want to extract an event described in the release, such as which company purchased which other company for what price. This often assumes that all the parties are already identified via NER.
- Another task that is closely related to NER is entity linking, where mentions of named entities are linked to some knowledge base, such as Wikipedia. When Wikipedia is used as a knowledge base, entity linking is also called Wikification.

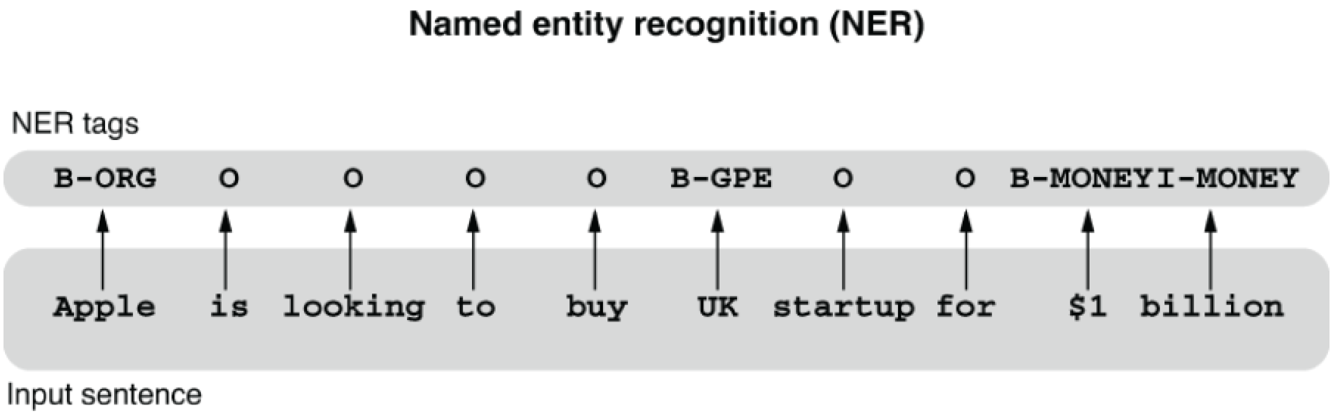
Tagging spans

Unlike POS tagging, where each word is assigned a POS tag, mentions to named entities can span over more than one word, for example, "the United States" and "World Trade Organization." A span in NLP is simply a range over one or more contiguous words. How can we use the same sequential tagging framework to model spans?

A common practice in NLP is to use some form of encoding to convert spans into per-word tags. The most common encoding scheme used in NER is called IOB2 tagging. It represents spans by a combination of the positional tag and the category tag. Three types of positional tags follow:

- B (Beginning): assigned to the first (or the only) token of a span
- I (Inside): assigned to all but the first token of a span
- O (Outside): assigned to all words outside of any spans

Now, let's take a look at the NER example which is shown in figure below.



The token "Apple" is the first (and the only) token of ORG (for "organization"), and it is assigned a B-ORG tag. Similarly, "UK", the first and the only token of GPE (for "geo- political entity"), is assigned B-GPE. For "$1" and "billion," the first and the second tokens of a monetary expression (MONEY), B-MONEY and I-MONEY are assigned, respectively. All the other tokens are given O. The rest of the pipeline for solving NER is very similar to that of part-of-speech tag- ging: both are concerned with assigning an appropriate tag for each word and can be solved by RNNs. In the next section, we are going to build a simple NER system using neural networks.

Applications:

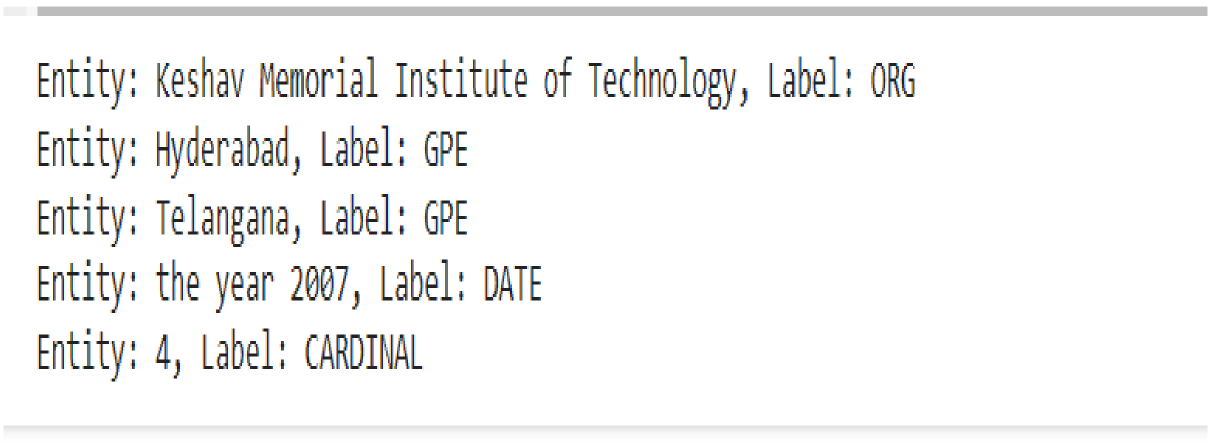
NER has numerous applications across various domains, including:

- Information Extraction: Identifying key entities from text for database population or knowledge graph construction.
- Question Answering: Identifying entities mentioned in questions and providing relevant answers.
- Sentiment Analysis: Analyzing sentiments expressed toward specific entities mentioned in text.
- Document Summarization: Extracting important named entities for summarizing documents or articles.

```python
import spacy


# Load the English language model

nlp = spacy.load("en_core_web_sm")


# Sample text

text = "Keshav Memorial Institute of Technology is a college of engineering in Hyderabad in the state of Telangana in south-central India Established in the year 2007 with 4 branches."


# Process the text using spaCy

doc = nlp(text)


# Extract named entities

entities = [(ent.text, ent.label_) for ent in doc.ents]


# Print the named entities and their labels

for entity, label in entities:

    print(f"Entity: {entity}, Label: {label}")
```

1. **Import spaCy**: We start by importing the spaCy library.
2. **Load Language Model**: We load the English language model provided by spaCy (**en_core_web_sm**). This model contains pre-trained word vectors and a named entity recognition component.
3. **Sample Text**: We define a sample text containing various named entities such as organization names, person names, and dates.

4. **Process Text**: The **nlp()** function is used to process the sample text, which tokenizes the text and applies various linguistic annotations, including named entity recognition.
5. **Extract Named Entities**: We iterate over the entities detected in the processed document (**doc.ents**) and extract the text of each entity along with its label.
6. **Print Named Entities**: We print the extracted named entities along with their corresponding labels.



```
Entity: Keshav Memorial Institute of Technology, Label: ORG
Entity: Hyderabad, Label: GPE
Entity: Telangana, Label: GPE
Entity: the year 2007, Label: DATE
Entity: 4, Label: CARDINAL
```

Modeling a Language

- Language modeling is a fundamental task in natural language processing (NLP) that involves predicting the probability distribution of the next word in a sequence of words.
- It's a crucial component in various NLP applications such as machine translation, speech recognition, text generation, and more.

What is a language model?

Imagine you are asked to predict what word comes next given a partial sentence:

“My trip to the beach was ruined by bad_____________ .”

What words could come next?

Many things could ruin a trip to a beach, but most likely it’s bad weather. Maybe it’s bad-mannered people at the beach, or maybe it’s bad food that the person had eaten before the trip, but most would agree that “weather” is a likely word that comes after this partial sentence. Few other nouns (people, food, dogs) and words of other parts of speech (be, the, run, green) are as appropriate as “weather” in this context.

What you just did is to assign some belief (or probability) to an English sentence. You just compared several alternatives and judged how likely they are as English sentences. Most people would agree that the probability of “My trip to the beach was ruined by bad weather” is a lot higher than “My trip to the beach was ruined by bad dogs.”

Formally, a language model is a statistical model that gives a probability to a piece of text. An English language model would assign higher probabilities to sentences that look like English. For example, an English language model would give a higher prob- ability to “My trip to the beach was ruined by bad weather” than it does to “My trip to the beach was ruined by bad dogs” or even “by weather was trip my bad beach the ruined to.” The more grammatical and the more “sense” the sentence makes, the higher the probability is.

Why are language models useful?

Although predicting the next word might come in handy when you are answering fill-in-the-blank questions for an exam, what particular roles do language models play in NLP?

It is essential for any systems that generate natural language.

For example, machine translation systems, which generate a sentence in a language given a sentence in another language, would benefit greatly from high-quality language models.

NOTE :In fact, neural machine translation models can be thought of as a variation of a language model that generates sentences in the target language conditioned on its input (sentences in the source language)

A similar situation arises in speech recognition, too, which is another task that generates text given spoken audio input.

For example, if somebody uttered “You’re right,” how would a speech recognition system know it’s actually “you’re right?” Because “you’re” and “your” can have the same pronunciation, and so can “right” and “write” and even “Wright” and “rite,” the system output could be any one of “You’re write,” “You’re Wright,” “You’re rite,” “Your right,” “Your write,” “Your Wright,” and so on.

Again, the simplest approach to resolving this ambiguity is to use a language model. An English language model would properly rerank these candidates and determine “you’re right” is the most likely transcription.

In fact, humans do this type of disambiguation all the time, though unconsciously. When you are having a conversation with somebody else at a large party, the actual audio signal you receive is often very noisy. Most people can still understand each other without any issues because people’s language models help them “correct” what you hear and interpolate any missing parts. You’ll notice this most if you try to con- verse in a less proficient, second language—you’d have a lot harder time understand- ing the other person in a noisy environment, because your language model is not as good as your first language’s. several alternatives and judged how likely they are as English sentences. Most people would agree that the probability of “My trip to the beach was ruined by bad weather” is a lot higher than “My trip to the beach was ruined by bad dogs.”

Training an RNN language model

Imagine you want to estimate the chance of tomorrow’s weather being rainy and the ground wet.

Let’s simplify this and assume there are only two types of weather, sunny and rainy.

There are only two outcomes for the ground: dry or wet.

This is equivalent to estimating the probably of a sequence: [rain, wet].

Further assume that there's a 50-50 chance of rain on a given day. After raining, the ground is wet with a 90% chance. Then, what is the probability of the rain and the ground being wet? It's simply 50% times 90%, which is 45%, or 0.45. If we know the probability of one event happening after another, you can simply multiply two probabilities to get the total probability for the sequence. This is called the chain rule in probability theory.

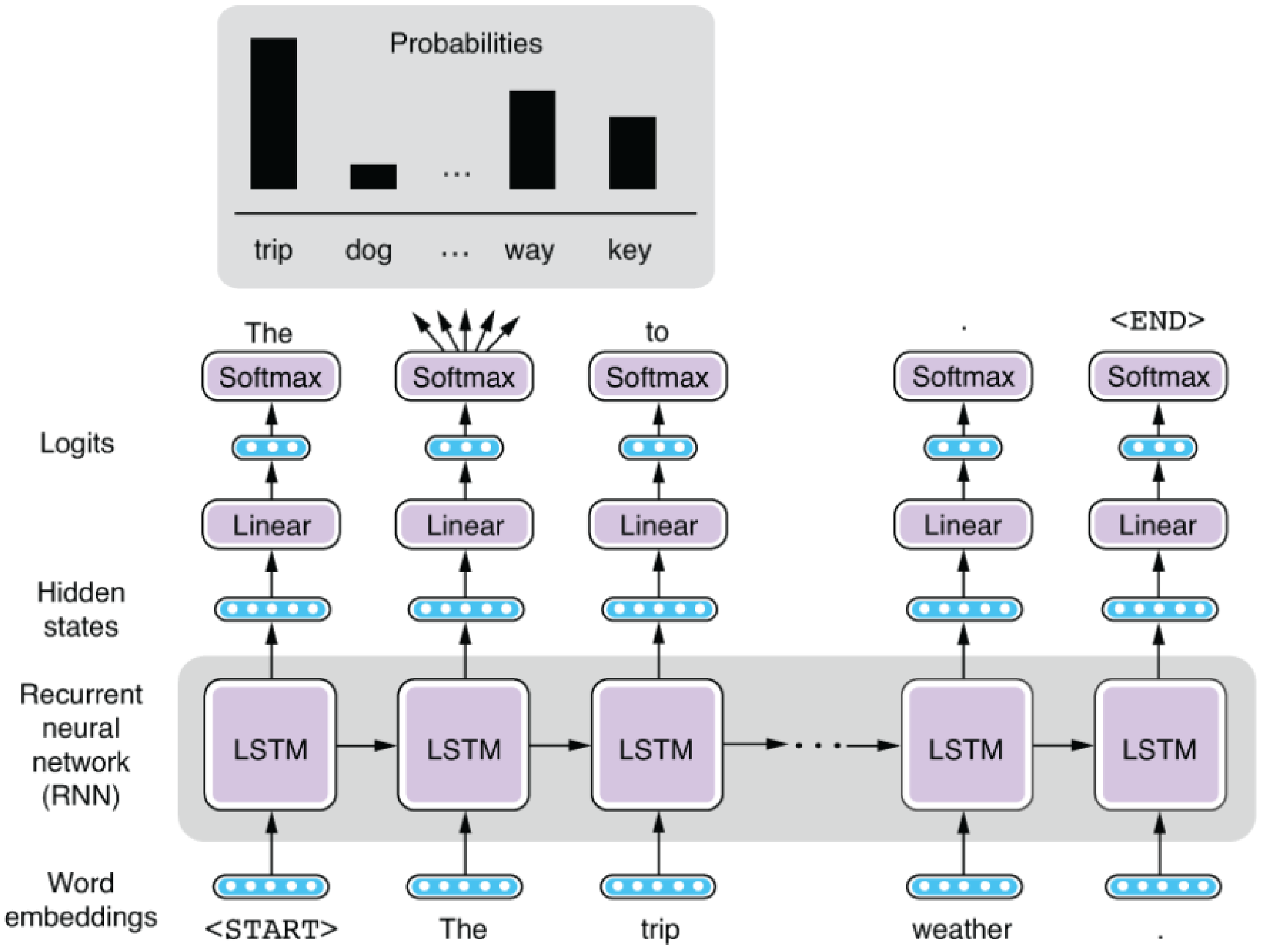
Similarly, if you can correctly estimate the probability of one word occurring after a partial sentence, you can simply multiply it with the probability of the partial sentence. Starting from the first word, you can keep doing this until you reach the end of the sentence. For example, if you'd like to compute the probability for "The trip to the beach was . . . ," you can multiply the following:

- The probability of "The" occurring at the beginning of a sentence
- The probability of "trip" occurring after "The"
- The probability of "to" occurring after "The trip"
- The probability of "the" occurring after "The trip to"
- And so on

This means that to build a language model, you need a model that predicts the probability (or, more precisely, the probability distribution) of the next word given the context. You may have noticed that this sounds a little familiar. Indeed, what's done here is very similar to the sequential-labeling models that we've been talking about in this chapter. For example, a part-of-speech (POS) tagging model predicts the probability distribution over the possible POS tags given the context. A named entity recognition (NER) model does it for the possible named entity tags. The difference is that a language model does it for the possible next words, given what the model has encountered so far.

Summary

To build a language model, you tweak an RNN-based sequence-labeling model a little bit so that it gives the estimates for the next word, instead of POS or NER tags. The Skip-gram model, which predicts the words in a context given the target word. Notice the similarity here—both models predict the probability over possible words. The input to the Skip-gram model is just a single word, whereas the input to the language model is the partial sequence. You can use a similar mechanism for converting one vector to another using a linear layer, then converting it to a probability distribution using softmax. The architecture is shown in figure below.



The way RNN-based language models are trained is similar to other sequential-labeling models. The loss function we use is the sequential cross-entropy loss, which measures how "off" the predicted words are from actual words. The cross-entropy loss is computed per word and averaged over all words in the sentence.

Text Generation using RNNs

language models give probabilities to natural language sentences, also we can generate natural language sentences from scratch using a language model!

Generating text using Recurrent Neural Networks (RNNs) is a fascinating application of deep learning in natural language processing (NLP).

RNNs, with their ability to capture sequential dependencies, are well-suited for generating coherent and contextually relevant text.

Feeding characters to an RNN

RNN language model we build here operates on characters, not on words or tokens.

All the RNN models we've seen so far operate on words, which means the input to the RNN was always sequences of words.

In theory, RNNs can operate on sequences of anything, be it tokens or characters or something completely different (e.g., waveform for speech recognition), as long as they are something that can be turned into vectors.

In building language models, we often feed characters, even including whitespace and punctuations as the input, treating them as words of length 1.

The rest of the model works exactly the same— individual characters are first embedded (converted to vectors) and then fed into the RNN, which is in turn trained so that it can best predict the distribution over the characters that are likely to come next.

Lets consider 2 cases on using RNNs with Character embeddings

Case1: Using characters will definitely make the RNN less efficient, as it would need more computation to "figure out" the same concept.

For example, a word-based RNN can receive the word "dog" at a timestep and update its internal states, whereas a character-based RNN would not able to do it until it receives three elements d, o, and g, and probably "_" (whitespace).

A character based RNN needs to "learn" that a sequence of these three characters means something special (the concept of "dog").

Case2: By feeding characters to RNNs, you can bypass many issues arising from dealing with tokens. One such issue is related to out-of-vocabulary (or OOV) words.

- When training a word-based RNN, you usually fix the entire set of vocabulary, often by enumerating all words that appeared in the train set. But whenever it encounters an OOV word in the test set, it doesn't know what to do with it.
- Oftentimes, it assigns a special token to all OOV words and treats them in the same way, which is not ideal.
- A character-based RNN, on the contrary, can still operate on individual characters, so it may be able to figure out what "doggy" means, for example, based on the rules it has learned by observing "dog" in the train set, even though it has never seen the exact word "doggy" before

Evaluating text using a language model

- The first step is to read a plain text dataset file and generate instances for training the model.
- NLTK (Natural Language Toolkit) does not have built-in support for character-level tokenization out of the box. However, you can easily implement character tokenization using basic Python string operations.

1. We import the NLTK library.
2. We define a sample text.
3. We tokenize the text into individual characters by converting it into a list of characters.

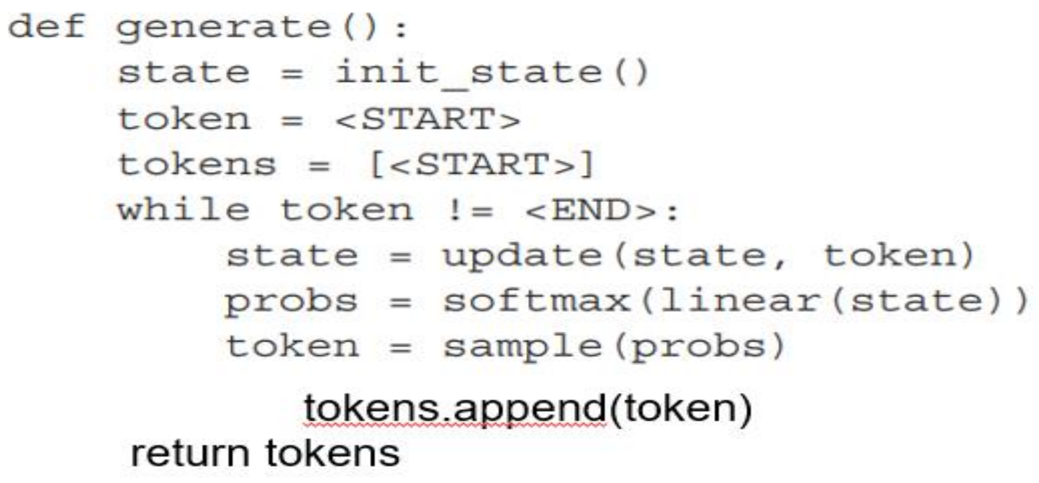
This simple approach splits the text into individual characters, creating a list where each character is a separate token.

Here's how you can tokenize text into characters using NLTK:

```
import nltk

# Sample text

text = "The quick brown fox jumps over the lazy dog."

# Tokenize text into characters

tokens = list(text)

# Print the tokens

print(tokens)
```

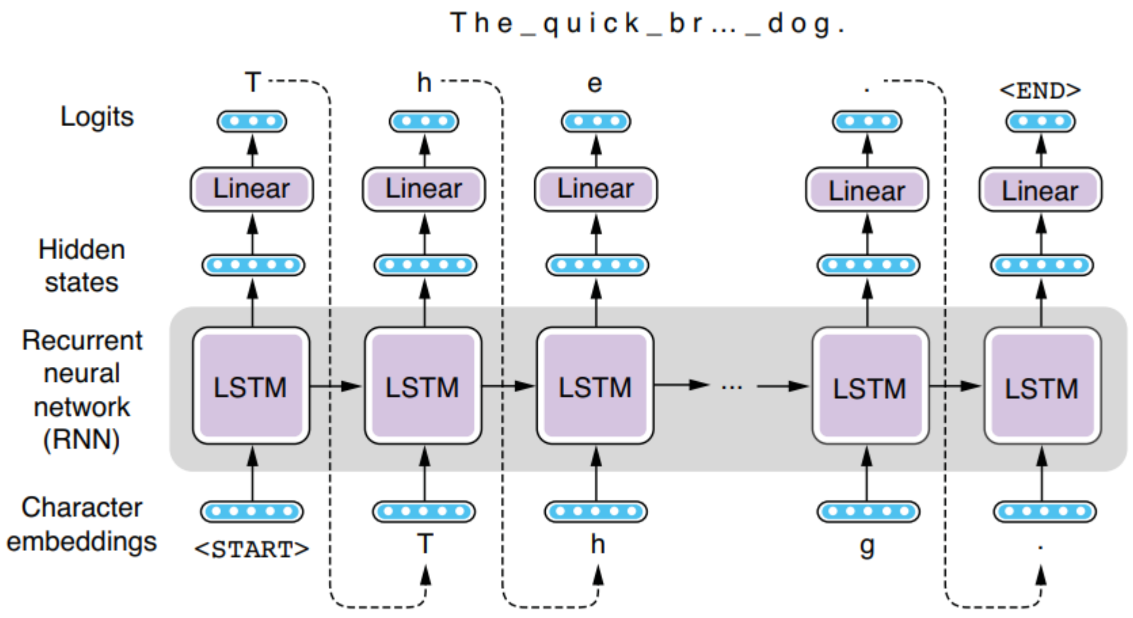
Generating text using a language model

- The most interesting aspect of (fully trained) language models is that they can predict possible characters that may appear next given some context.
- Specifically, they can give you a probability distribution over possible characters that may come next, from which you choose to determine the next character.
- For example, if the model has generated “t” and “h,” and the LM is trained on generic English text, it would probably assign a high probability on the letter “e,” generating common English words including the, they, them, and so on.
- If you start this process from the token and keep doing this until you reach the end of the sentence (i.e., by generating), you can generate an English sentence from scratch.



```
def generate():
    state = init_state()
    token = <START>
    tokens = [<START>]
    while token != <END>:
        state = update(state, token)
        probs = softmax(linear(state))
        token = sample(probs)
          tokens.append(token)
     return tokens
```

- This loop looks very similar to the one for updating RNNs with one key difference: here, we are not receiving any input but instead are generating characters and feeding them as the input.
- In other words, the RNN operates on the sequence of characters that the RNN itself generated so far.
- Such models that operate on past sequences they produced are called autoregressive models.



In the previous cases, init_state() and update() functions are the ones that initialize and update the hidden states of the RNN.

In generating text, we assume that the model and its parameters are already trained on a large amount of natural language text. softmax() is a function to run Softmax on the given vector, and linear() is the linear layer to expand/shrink the size of the vector.

The function sample() returns a character according to the given probability distribution.

For example, if the distribution is “a”: 0.6, “b”: 0.3, “c”: 0.1, it will choose “a” 60% of the time, “b” 30% of the time, and “c” 10% of the time.

This ensures that the generated string is different every time while every string is likely to look like a real English sentence.