KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

III YEAR II SEM

NATURAL LANGUAGE PROCESSING (CM621PE)

UNIT-3 NOTES

Course Objectives: The course will help to

1. Understand NLP concepts.
2. Understand RNN concepts for language modeling.
3. Understand sequence to sequence models.
4. Understand attentions and transformers.
5. Understand NLP applications using deep learning techniques.

Course Outcomes: After learning the concepts of this course, the student is able to

1. Describe NLP concepts and techniques.
2. Explain RNN for NLP
3. Develop sequence to sequence models along with CNN.
4. Demonstrate attentions and transformers for NLP.
5. Implement applications of NLP using deep learning techniques.

UNIT-III

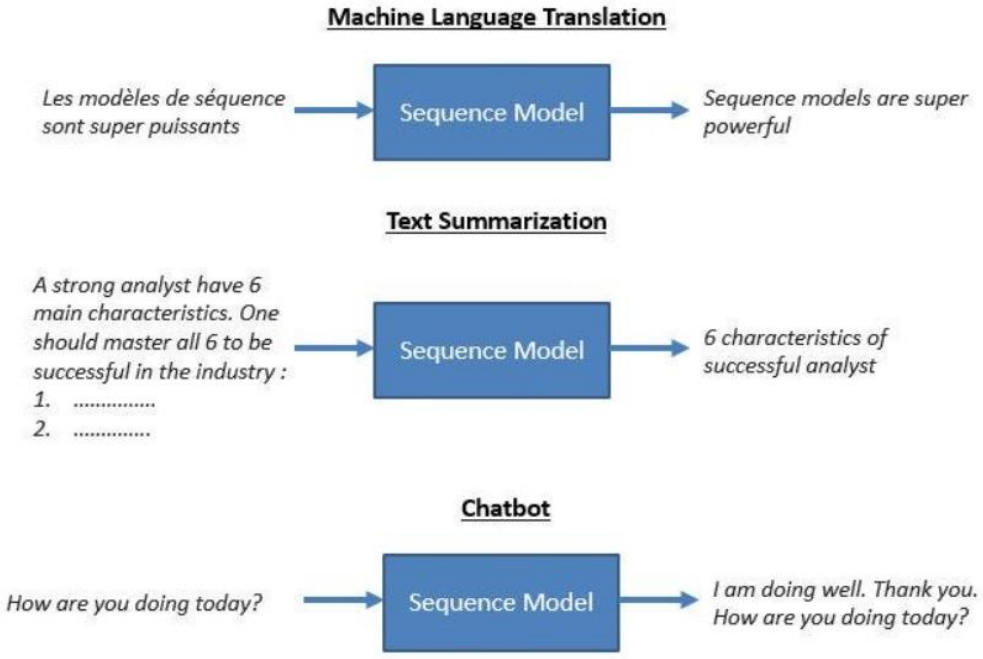
Sequence-to-sequence models: Introducing sequence-to-sequence models - Machine translation - Building your first translator - How Seq2Seq models work - Evaluating translation systems - Case study: Building a chatbot

Convolutional neural networks: Introducing convolutional neural networks (CNNs)- RNNs and their shortcomings - Case study: Text classification

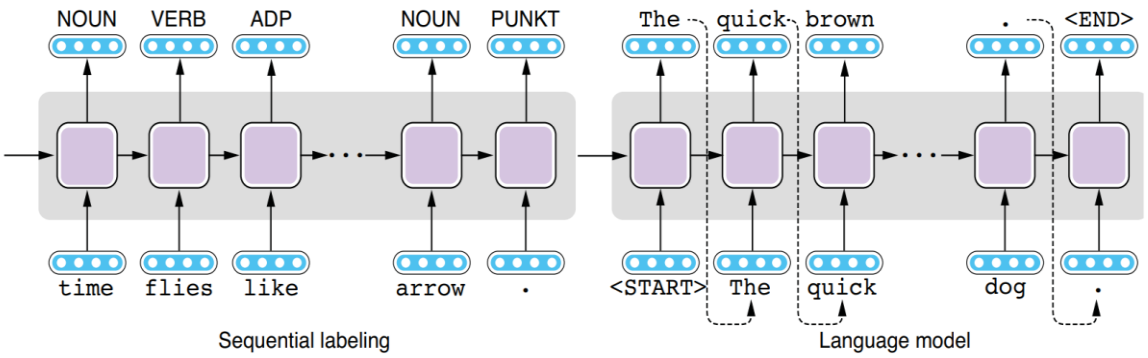
SEQUENCE TO SEQUENCE MODELS

Sequence-to-sequence (Seq2Seq) models are some of the most important complex NLP models and are used for a wide range of applications, including machine translation.

Seq2Seq models and their variations are used as the fundamental building blocks in many real-world applications, including Google Translate and speech recognition.



- sequence-labeling model takes a sequence of some units (e.g., words) and assigns a label (e.g., a part-of-speech (POS) tag) to each unit, whereas
- A language model takes a sequence of some units (e.g., words) and estimates how probable the given sequence is in the domain in which the model is trained.



Although these two models are useful for a number of NLP tasks, for some, we may want the best of both worlds—we may want our model to take some input (e.g., a sentence) and generate something else (e.g., another sentence) in response. For example, if we wish to translate some text written in one language into another, we need our model to take a sentence and produce another. Can we do this with sequential-labeling models?

No, because they can produce only the same number of output labels as there are tokens in the input sentence. This is obviously too limiting for translation—one expression in a language (say, “Enchanté” in French) can have an arbitrarily large or small number of words in another (say, “Nice to meet you” in English). Can we do this with language models? Again, not really.

Although we can generate realistic-looking text using language models, we have almost no control over the text they generate. In fact, language models do not take any input. The model on the left (the sequential-labeling model) takes a sentence as its input and produces some form of representations, whereas the model on the right produces a sentence with variable length that looks like natural language text. We already have the components needed to build what we want, that is, a model that takes a sentence and transforms it into another. The only missing part is a way to connect these two so that we can control what the language model generates. In fact, by the time the model on the left finishes processing the input sentence, the RNN has already produced its abstract representation, which is encoded in the RNN’s hidden states.

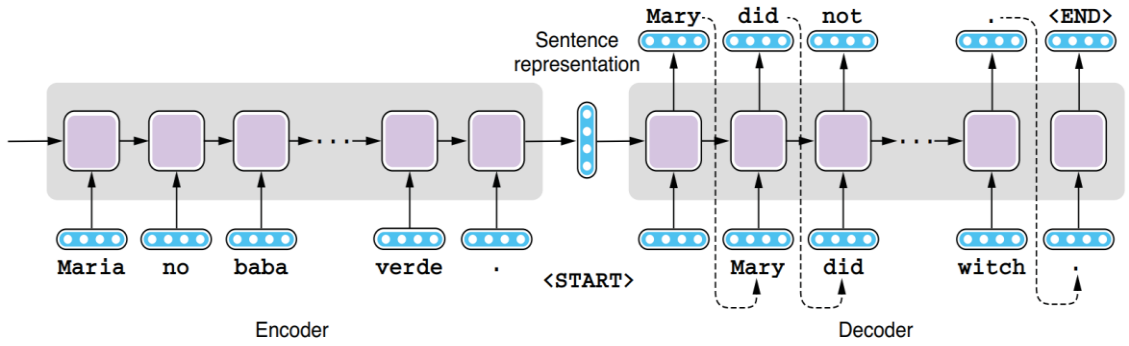
If we can simply connect these two so that the sentence representation is passed from left to right and the language model can generate another sentence based on the representation.

The basic idea behind Seq2Seq models is to use one neural network to encode the input sequence into a fixed-size vector representation, and then use another neural network to decode this representation into the output sequence.

A Seq2Seq model consists of two subcomponents—an encoder and a decoder.

An encoder takes a sequence of some units (e.g., a sentence) and converts it into some internal representation.

A decoder, on the other hand, generates a sequence of some units (e.g., a sentence) from the internal representation.



The many-to-many architecture of an RNN is what the sequence-to-sequence model (seq2seq) uses.

Its ability to transfer arbitrary-length input sequences to arbitrary-length output sequences has made it useful for various applications.

In most cases, the length of the input and output differ.

For example

Consider the sentence "I am doing good" to be mapped to "Je vais bien".

This algorithm handles these scenarios with varying input and output sequence lengths.

Encoder Block

The main purpose of the encoder block is to process the input sequence and capture information in a fixed-size context vector.

Architecture:

- The input sequence is put into the encoder.
- The encoder processes each element of the input sequence using neural networks (or transformer architecture).
- Throughout this process, the encoder keeps an internal state, and the ultimate hidden state functions as the context vector that encapsulates a compressed representation of the entire input sequence. This context vector captures the semantic meaning and important information of the input sequence.

The final hidden state of the encoder is then passed as the context vector to the decoder.

Decoder Block

The decoder block is similar to encoder block. The decoder processes the context vector from encoder to generate output sequence incrementally.

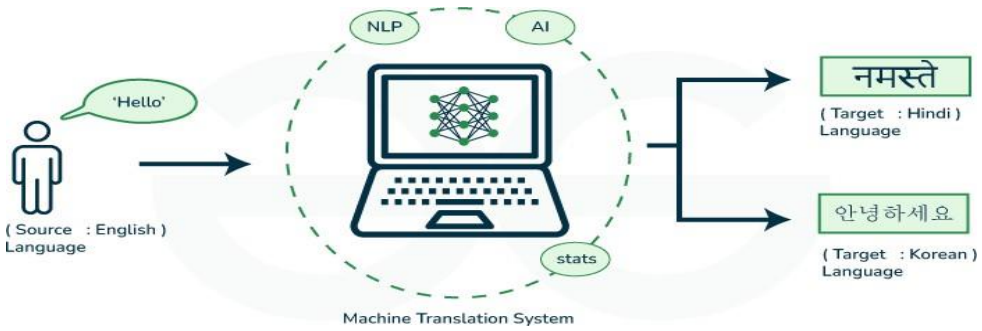
Architecture:

- In the training phase, the decoder receives both the context vector and the desired target output sequence (ground truth).
- During inference, the decoder relies on its own previously generated outputs as inputs for subsequent steps.

As a whole, a Seq2Seq model takes a sequence and generates another sequence. As with the language model, the generation stops when the decoder produces a special token, , which enables a Seq2Seq model to generate an output that can be longer or shorter than the input sequence.

Many variants of Seq2Seq models exist, depending on what architecture you use for the encoder, what architecture you use for the decoder, and how information flows between the two.

Machine translation is a sub-field of computational linguistics that focuses on developing systems capable of automatically translating text or speech from one language to another. In Natural Language Processing (NLP), the goal of machine translation is to produce translations that are not only grammatically correct but also convey the meaning of the original content accurately.



Need for Machine Translation

There is an explosion of data in the modern world with the information revolution and since language is the most effective medium of communication for humans, there is also an increased need for translating from one language to another using NLP tools and other techniques.

- The demand for translation is majorly due to the exponential rise increase in the exchange of information between various regions using different regional languages.
- **Examples** such as access to web documents in non-native languages, using products from across countries, real-time chat, legal literature, etc. are some use cases.

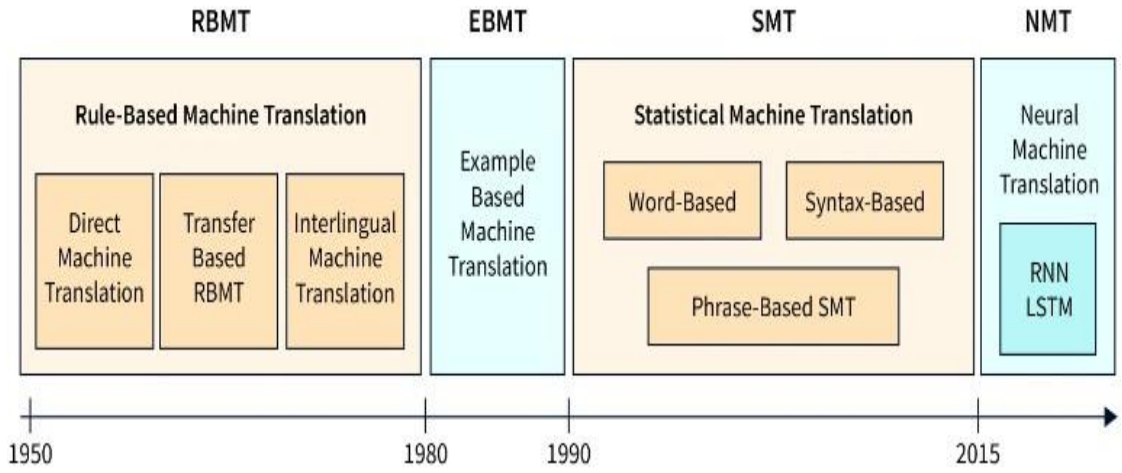
Sequence to Sequence Model

- With the advent of **recurrent network models** for a variety of complex tasks where the order of the task (like the sequence of words in a sentence) is important, Sequence to Sequence has become very popular for machine translation tasks in NLP.
- Sequence to Sequence models primarily consists of an encoder and decoder model architecture and utilize a combination of recurrent units in their layers.
- The encoder and the decoder are two distinct individual components working hand in hand together to produce state-of-the-art results while performing different kinds of computations.
- The input sequences of words are taken as the input by the encoder, and the decoder component typically has the relevant architecture to produce the associated target sequences.

- In the machine translation task, the source language is passed as input to the encoder as a sequence of vectors, and then it is passed through a series of recurrent units whose end output is stored as the last hidden state as the encoder state representation.
- This hidden state is used as input for the decoder and passed through a series of recurrent again as a for loop, usually each output sequentially so that the output produced has the highest probability.

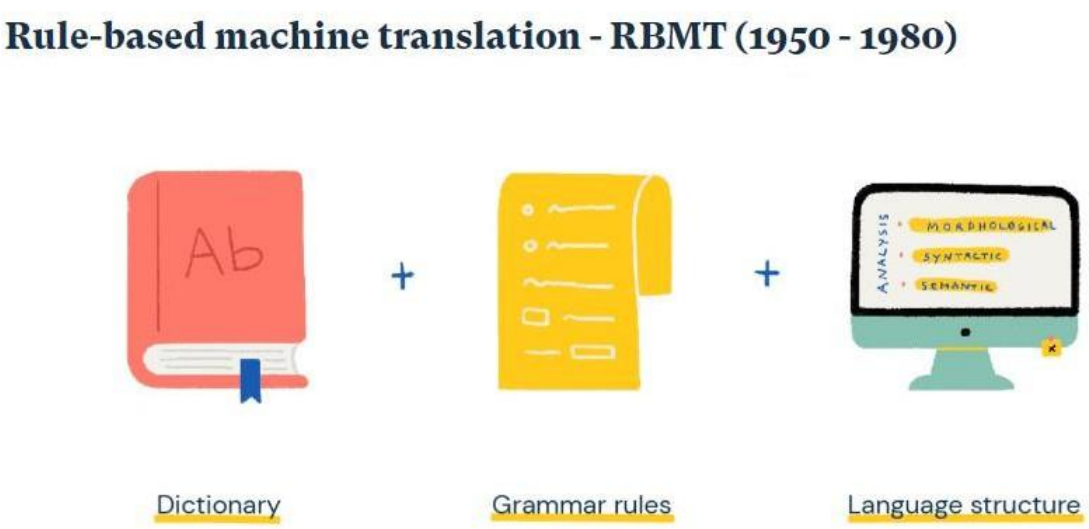
What are the key approaches in Machine Translation?

- In machine translation, the original text is decoded and then encoded into the target language through two step process that involves various approaches employed by language translation technology to facilitate the translation mechanism.
- Machine translation methods are mainly classified into rule-based and corpus-based methods during the developments from the 1950s till now. Recently neural network-based models are primarily employed for translation tasks using NLP.



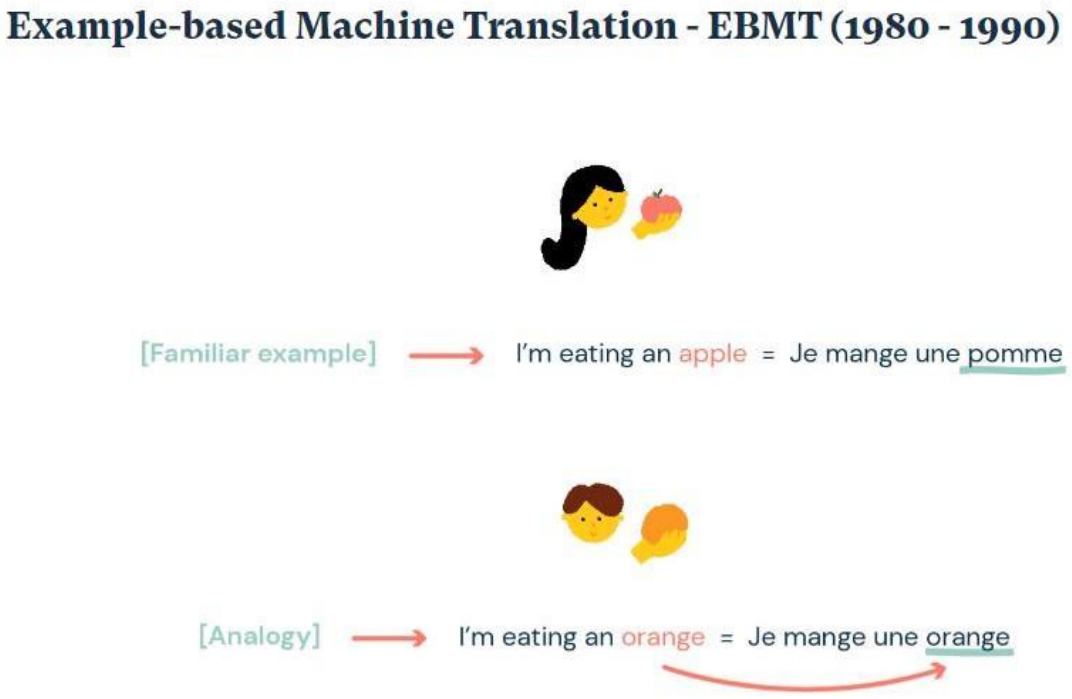
Rule-based machine translation

- Also called **knowledge-based machine translation**, these are the earliest set of classical methods used for machine translation.
- These translation systems are mainly based on **linguistic information** about the source and target languages that are derived from dictionaries and grammar covering the characteristic elements of each language separately.
- Once we have input sentences in some source languages, RBMT systems generally generate the translation to output language based on the morphological, syntactic, and semantic analysis of both the source and the target languages involved in the translation tasks.
-



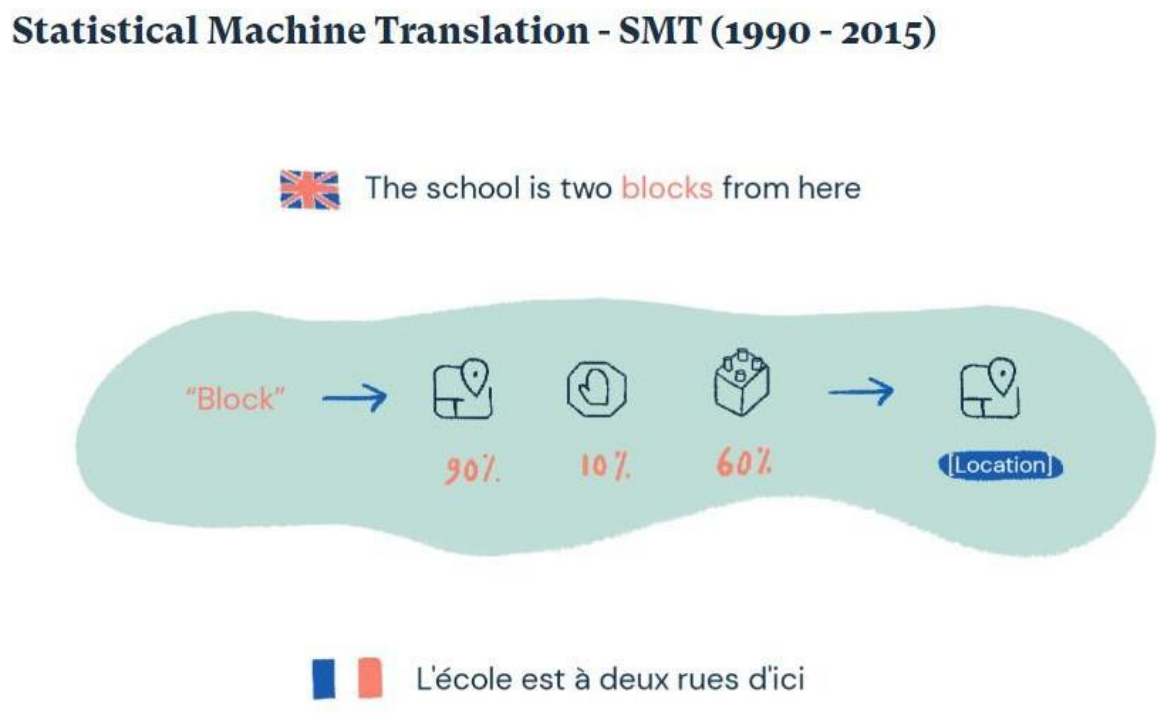
Example-based Machine Translation - EBMT (1980 - 1990)

- EBMT is essentially a translation by **analogy**. The system is provided with examples of correct translations that it uses to solve a new translation.
- As such, this approach requires a bilingual corpus **with parallel sentences**.
- During translation, the system **breaks down a source sentence into smaller phrases** and matches them against the examples given. When there is a match, the specific phrase can be substituted with the corresponding target phrases from the example.



Statistical Machine Translation or SMT

- It works by alluding to statistical models that depend on the investigation of huge volumes of bilingual content. It expects to decide the correspondence between a word from the source language and a word from the objective language. A genuine illustration of this is Google Translate.
- Presently, SMT is extraordinary for basic translation, however its most noteworthy disadvantage is that it doesn't factor in context, which implies translation can regularly be wrong or you can say, don't expect great quality translation.
- The principle theory of SMT-based models is that they rely on the Bayes Theorem to perform the analysis of parallel texts.



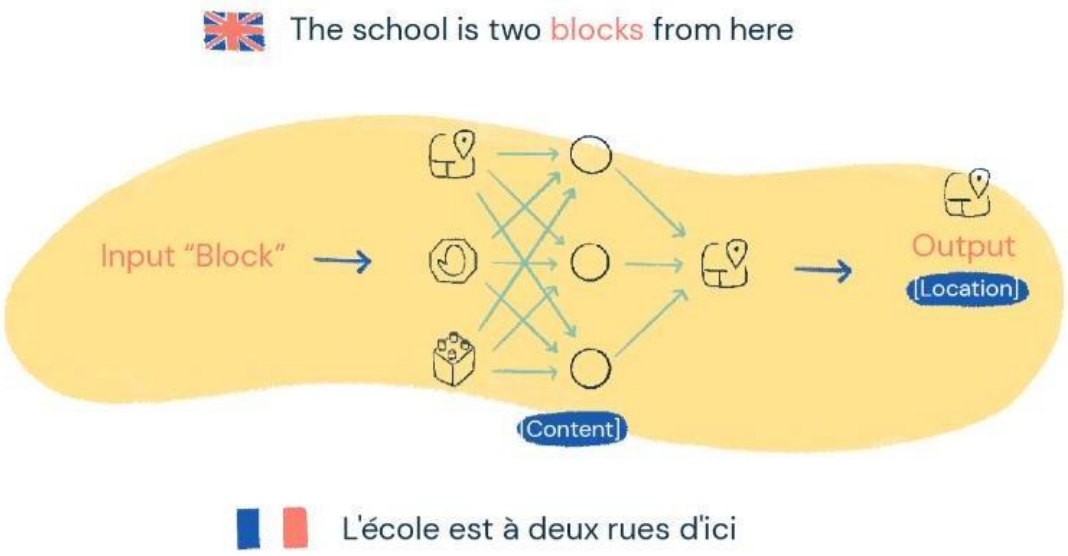
Hybrid Machine Translation or HMT

- Hybrid Machine Translation approaches take advantage of both statistical and rule-based translation methodologies which were proven to have better efficiency in the area of machine translation systems using NLP.

Neural Machine Translation or NMT

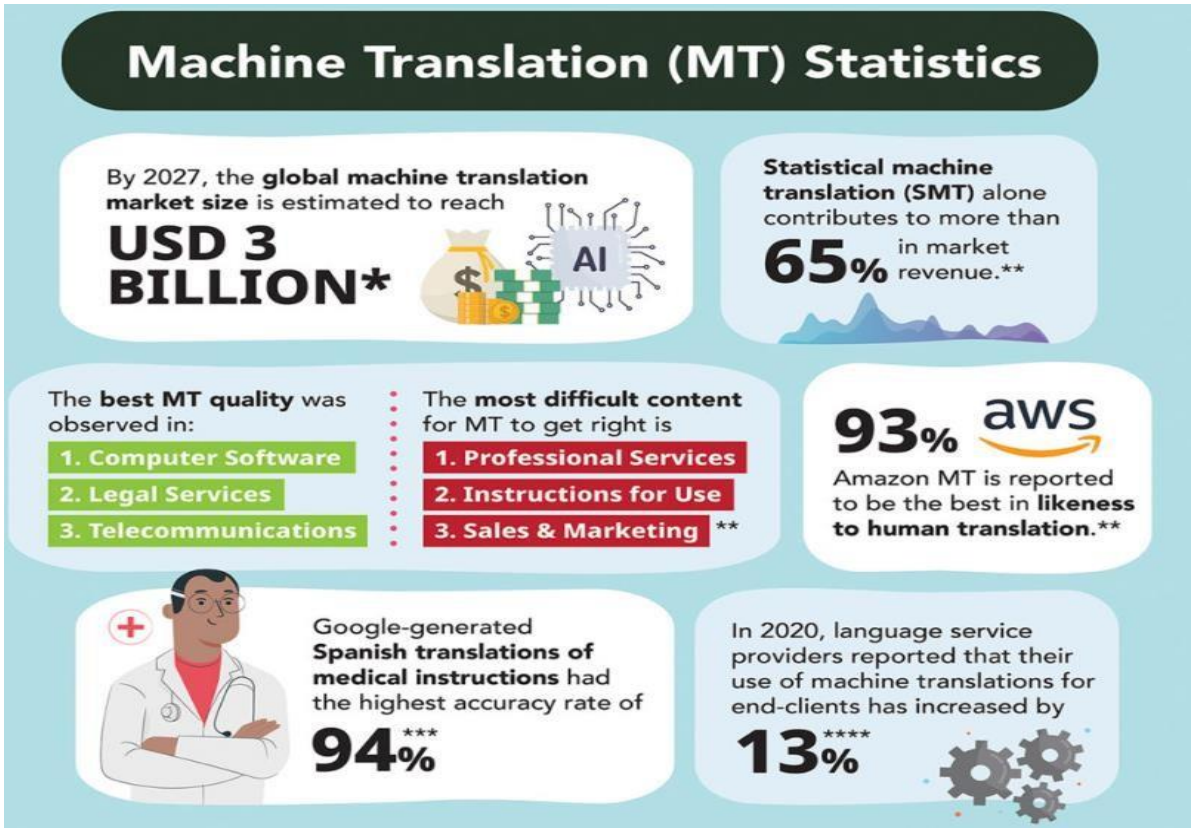
- Neural Machine Translation methods in NLP employ artificial intelligence techniques and, in particular, rely upon neural network models which were originally based on the human brain to build machine learning models with the end goal of translation and also further learning languages, and improving the performance of the machine translation systems constantly.
- Neural Machine Translation works by incorporating training data, which can be generic or custom depending on the user’s needs.

Neural Machine Translation - NMT (2015 - Present)



Why Machine Translation is difficult?

- Words and sentences can be ambiguous
- Translation sometimes requires a deeper analysis of discourse structures.
- lack of data for certain languages
- Named entities, as well as expressions of time, space, and quantity, can be difficult for the MT to handle
- A sentence can be translated in multiple ways and sometimes the right choice could be subjective.
- Multilingual sentences

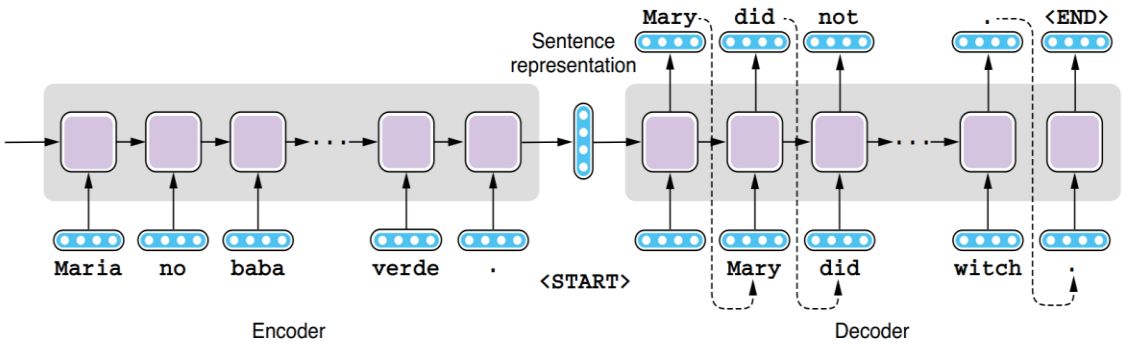


How Seq2seq Models work

A Seq2Seq model consists of two subcomponents—an encoder and a decoder.

An encoder takes a sequence of some units (e.g., a sentence) and converts it into some internal representation.

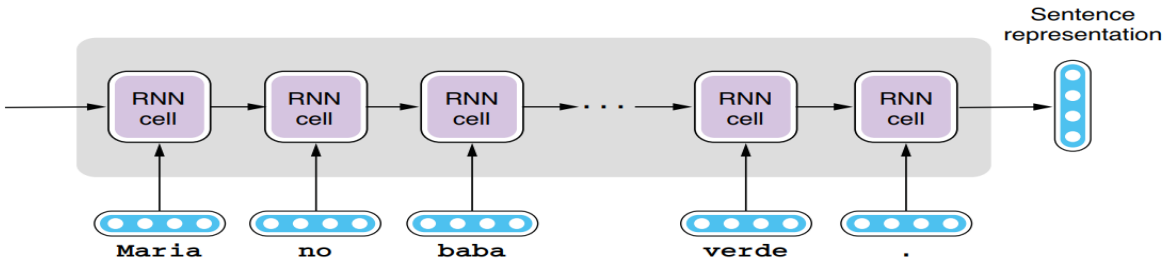
A decoder, on the other hand, generates a sequence of some units (e.g., a sentence) from the internal representation.



Encoder

- The encoder of a Seq2Seq model is not much different from the sequential-labeling models its main job is to take the input sequence (usually a sentence) and convert it into a vector representation of a fixed length.

- we can use an LSTM-RNN as shown in figure below.



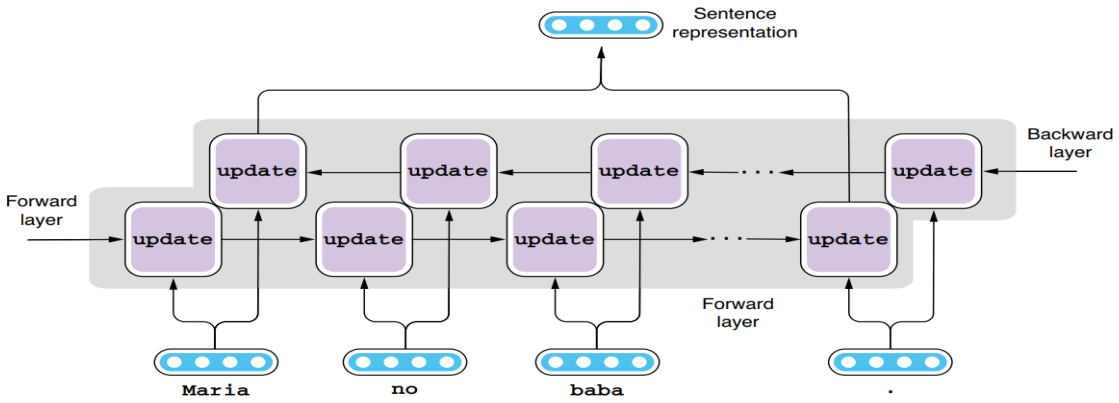
Unlike sequential-labeling models, we need only the final hidden state of an RNN, which is then passed to the decoder to generate the target sentence.

we can also use a multilayer RNN as an encoder, in which case the sentence representation is the concatenation of the output of each layer, as illustrated in figure below

Unlike sequential-labeling models, we need only the final hidden state of an RNN, which is then passed to the decoder to generate the target sentence.

we can also use a multilayer RNN as an encoder, in which case the sentence representation is the concatenation of the output of each layer, as illustrated in figure below

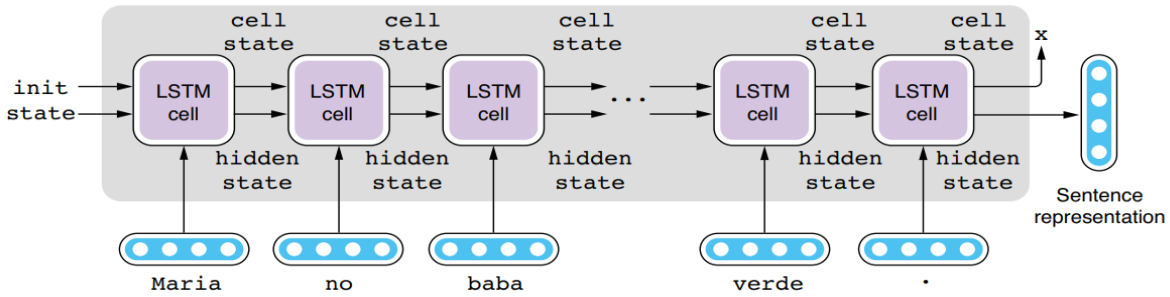
Similarly, we can use a bidirectional (or even a bidirectional multilayer) RNN as an encoder. The final sentence representation is a concatenation of the output of the forward and the backward layers, as shown in figure below



NOTE: Remember that an LSTM cell produces two types of output: the cell state and the hidden state.

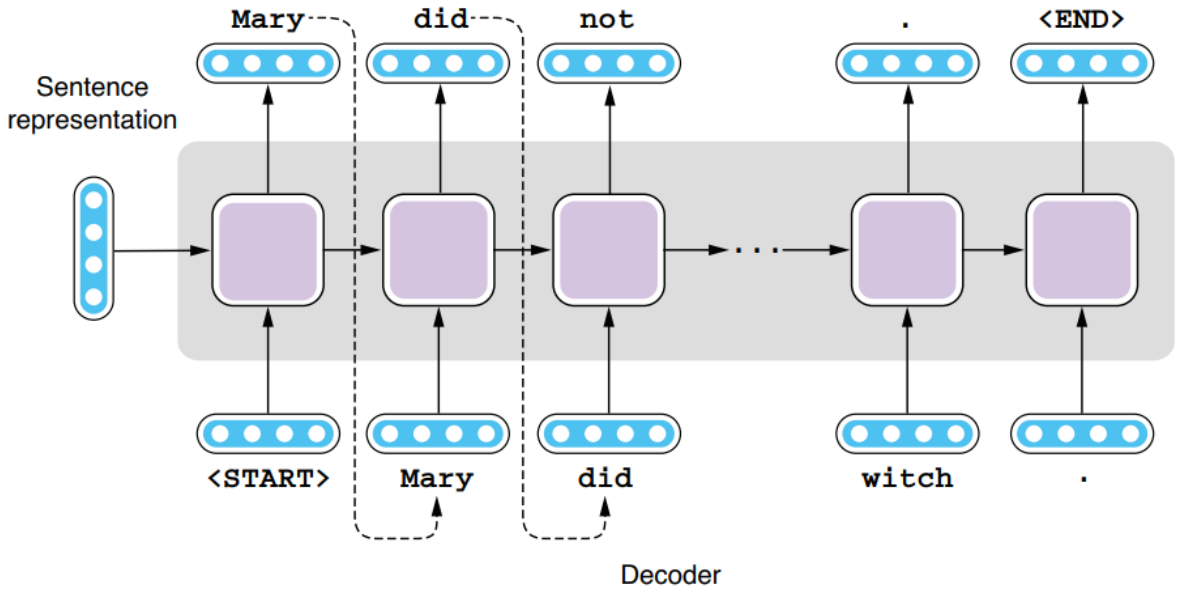
When using LSTM for encoding a sequence, we usually just use the final hidden state while discarding the cell state.

Think of the cell state as something like a temporary loop variable used for computing the final outcome (the hidden state). See figure below for an illustration



Decoder

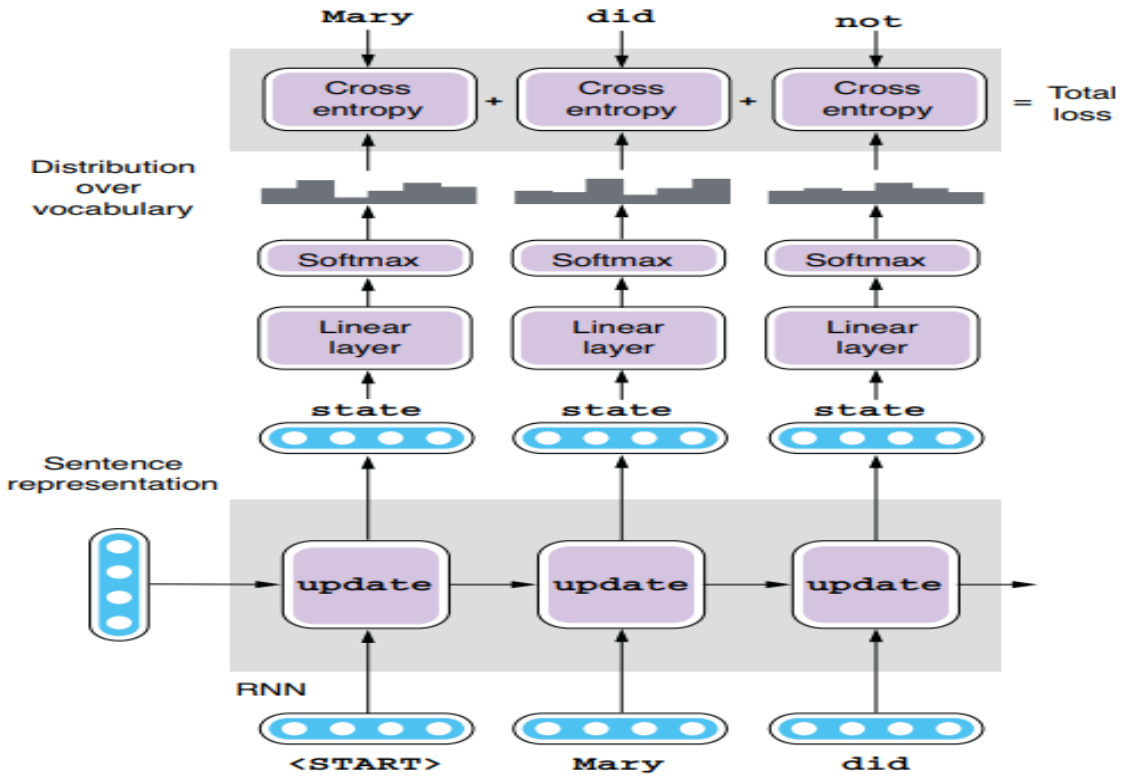
- The decoder of a Seq2Seq model is similar to the language model in fact, they are identical except for one crucial difference—a decoder takes an input from the encoder. The language models we saw are called unconditional language models because they generate language without any input or precondition.
- On the other hand, language models that generate language based on some input (condition) are called conditional language models. A Seq2Seq decoder is one type of conditional language model, where the condition is the sentence representation produced by the encoder.



- Just as with language models, Seq2Seq decoders generate text from left to right. Like the encoder, you can use an RNN to do this.
- A decoder can also be a multilayer RNN. However, a decoder cannot be bidirectional—you cannot generate a sentence from both sides.
- Models that operate on the past sequence they produced are called autoregressive models.

Non-autoregressive models

- If generating text from left to right is too limiting, you have a good point. Humans also do not always write language linearly—we often revise, add, and delete words and phrases afterward.
- Also, generating text in a linear fashion is not very efficient. The latter half of a sentence needs to wait until its first half is completed, which makes it very difficult to parallelize the generation process.
- Researchers are putting a lot of effort into developing non-autoregressive MT models that do not generate the target sentence in a linear. However, they haven't exceeded autoregressive models in terms of translation quality, and most research and production MT systems still adopt autoregressive models
- How the decoder behaves is a bit different between the training and the prediction stages.
- At the training stage, we know exactly how the source sentence should be translated into the target sentence. In other words, we know exactly what the decoder should produce, word by word. Because of this, decoders are trained in a similar way to how sequential-labeling models are trained.
- First, the decoder is fed the sentence representation produced by the encoder and a special token , which indicates the start of a sentence.
- The first RNN cell processes these two inputs and produces the first hidden state. The hidden state vector is fed to a linear layer that shrinks or expands this vector to match the size of the vocabulary.
- The resulting vector then goes through softmax, which converts it to a probability distribution.
- This distribution dictates how likely each word in the vocabulary is to come next



Then, this is where the training happens. If the input is “Maria no daba una bofetada a la bruja verde,” then we would like the decoder to produce its English equivalent: “Mary did not slap the green witch.” This means that we would like to maximize the probability that the first RNN cell generates “Mary” given the input sentence. You use the cross-entropy loss to measure how far apart the desired outcome is from the actual output of your network. If the probability for “Mary” is large, then good—the network incurs a small loss. On the other hand, if the probability for “Mary” is small, then the network incurs a large loss, which encourages the optimization algorithm to change the parameters (magic constants) by a large amount.

Then, we move on to the next cell. The next cell receives the hidden state computed by the first cell and the word “Mary,” regardless of what the first cell generated. Instead of feeding the token generated by the previous cell, as we did when generating text using a language model, we constrain the input to the decoder so that it won’t “go astray.” The second cell produces the hidden state based on these two inputs, which is then used to compute the probability distribution for the second word. We compute the cross-entropy loss by comparing the distribution against the desired output “did” and move on to the next cell. We keep doing this until we reach the final token, which is . The total loss for the sentence is the average of all the losses incurred for all the words in the sentence, as shown in figure . Finally, the loss computed this way is used to adjust the model parameters of the decoder, so that it can generate the desired output the next time around. Note that the parameters of the encoder are also adjusted in this process, because the loss propagates all the way back to the encoder through the sentence representation. If the sentence representation produced by the encoder is not good, the decoder won’t be able to produce high-quality target sentences no matter how hard it tries.

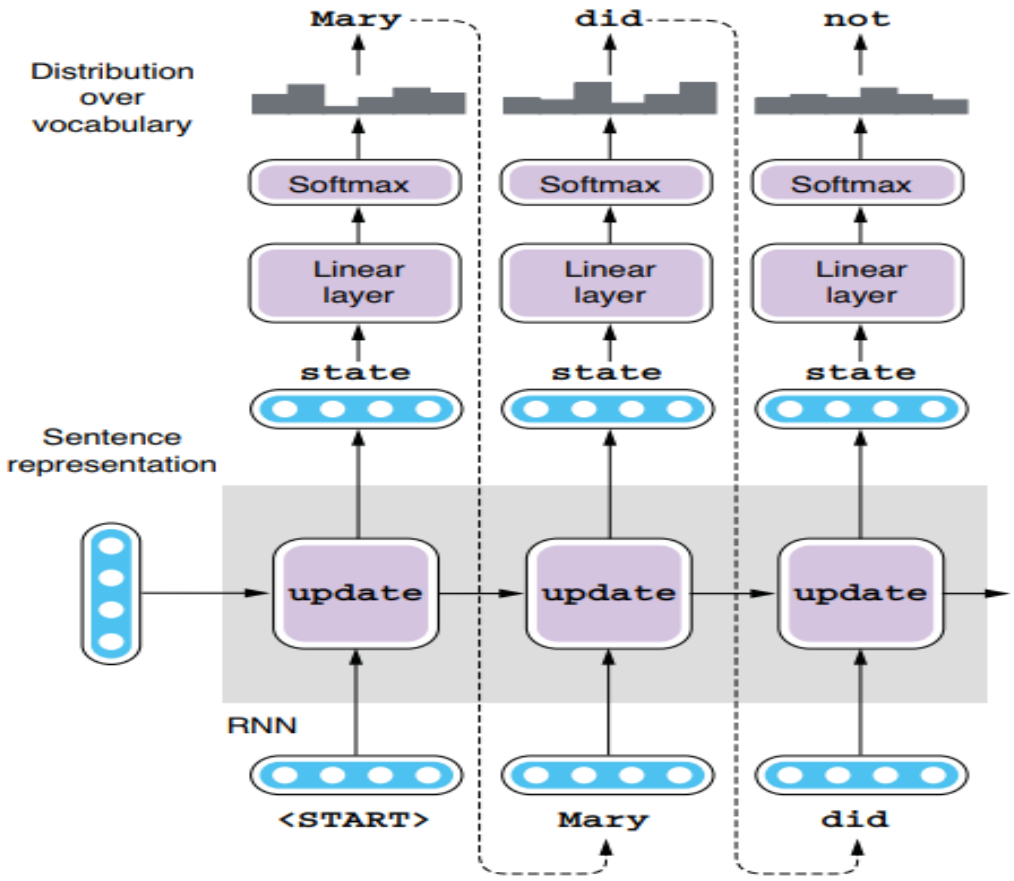
Greedy decoding

How the decoder behaves at the **prediction stage**?

where a source sentence is given to the network, but we don't know what the correct translation should be. At this stage, a decoder behaves a lot like the language models.

It is fed the sentence representation produced by the encoder, as well as a special token , which indicates the start of a sentence.

The first RNN cell processes these two inputs and produces the first hidden state, which is then fed to the linear layer and the softmax layer to produce the probability distribution over the target vocabulary.

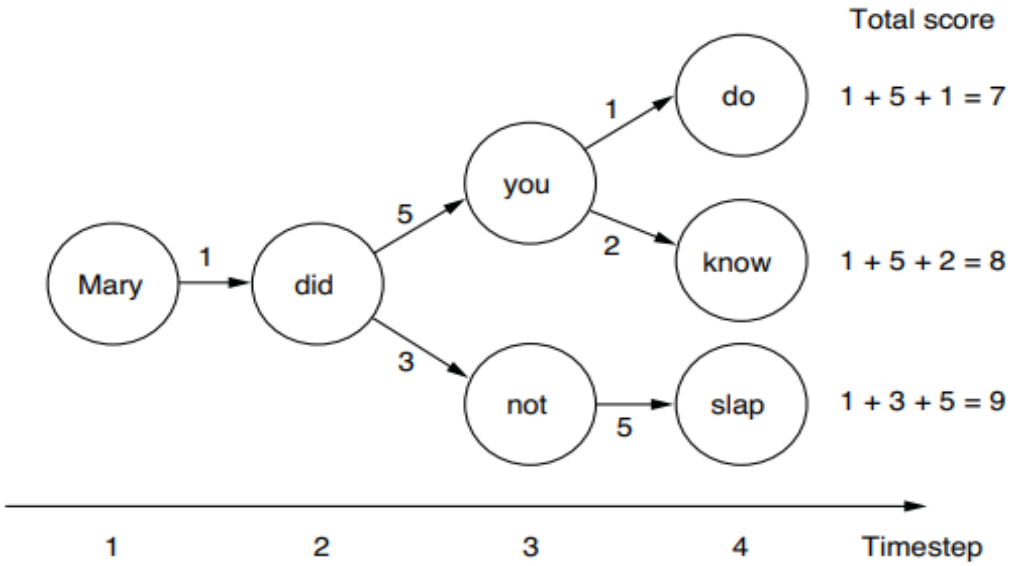


Here comes the key part—unlike the training phase, you don't know the correct word to come next, so you have multiple options.

You can choose any random word that has a reasonably high probability (say, "dog"), but probably the best you can do is pick the word whose probability is the highest (you are lucky if it's "Mary").

- The MT system produces the word that was just picked and then feeds it to the next RNN cell. This is repeated until the special token is encountered.

- First of all, the goal of MT decoding is to maximize the probability of the target sentence as a whole, not just individual words. This is exactly what you trained the network to do—to produce the largest probability for correct sentences.
- However, the way words are picked at each step described earlier is to maximize the probability of that word only. In other words, this decoding process guarantees only the locally maximum probability.
- This type of myopic, locally optimal algorithm is called greedy in computer science, and the decoding algorithm is called greedy decoding.
- However, just because you are maximizing the probability of individual words at each step doesn't mean you are maximizing the probability of the whole sentence.
- Greedy algorithms, in general, are not guaranteed to produce the globally optimal solution, and using greedy decoding can leave you stuck with suboptimal translations. This is not very intuitive to understand, so let me use a simple example to illustrate this. When you are picking words at each timestep, you have multiple words to pick from.
- You pick one of them and move on to the next RNN cell, which produces another set of possible words to pick from, depending on the word you picked previously. This can be represented using a tree structure like the one shown in figure
- The diagram shows how the word you pick at one timestep (e.g., "did") branches out to another set of possible words ("you" and "not") to pick from at the next timestep



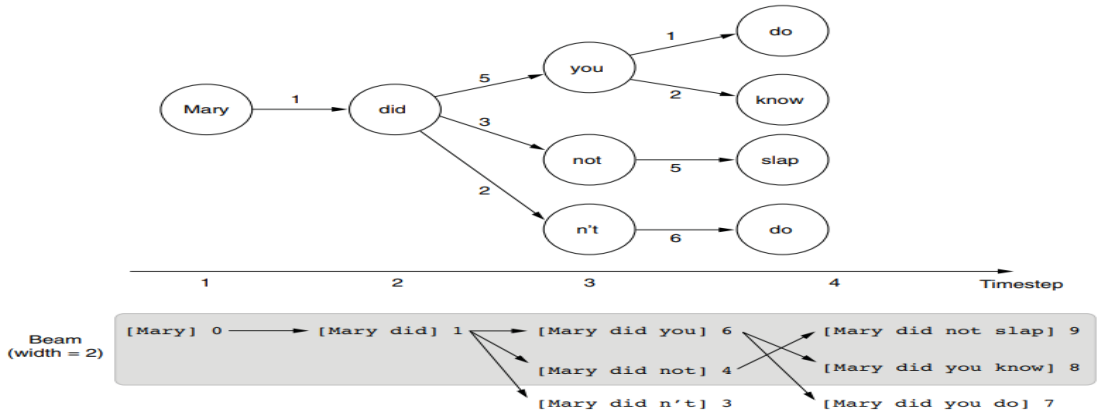
Each transition from word to word is labeled with a score, which corresponds to how large the probability of choosing that transition is. Your goal here is to maximize the total sum of the scores when you traverse one path from timestep 1 to 4. Mathematically, probabilities are real numbers between 0 to 1, and you should multiply (instead of add) each probability to get the total, but I'm simplifying things here. For example, if you go from "Mary" to "did," then on to "you" and "do," you just generated a sentence "Mary did you do" and the total score is 1 + 5 + 1 = 7.

The greedy decoder we saw earlier will face two choices after it generates "did" at timestep 2: either generate "you" with a score of 5 or "not" with a score of 3. Because all it does is pick the one with the highest score, it will pick "you" and move on. Then it will face another branch

after timestep 3—generating “do” with a score of 1 or generating “know” with a score of 2. Again, it will pick the largest score, and you will end up with the translation “Mary did you know” whose score is 1+ 5 + 1 = 8. This is not a bad result. At least, it is not as bad as the first path, which sums up to a score of 7. By picking the maximum score at each branch, you are making sure that your final result is at least decent. However, what if you picked “not” at timestep 3? At first glance, this doesn’t seem like a good idea, because the score you get is only 3, which is smaller than you’d get by taking the other path, 5. But at the next timestep, by generating “slap,” you get a score of 5. In retrospect, this was the right thing to do—in total, you get 1 + 3 + 5 = 9, which is larger than any scores you’d get by taking the other “you” path. By sacrificing short-term rewards, you are able to gain even larger rewards in the long run. But due to the myopic nature of the greedy decoder, it will never choose this path—it can’t backtrack and change its mind once it’s taken one branch over another

Beam search decoding

The main idea of beam search decoding is—instead of committing to one path, it purses multiple paths (called hypotheses) at the same time. In this way, we leave some room for “dark horses,” that is, hypotheses that had low scores at first but may prove promising later. Let’s see this in action using the example in figure below, a slightly modified version of figure of decoding decision tree as seen above.



The key idea of beam search decoding is to use a beam , which you can think of as some sort of buffer that can retain multiple hypotheses at the same time.

The size of the beam, that is, the number of hypotheses it can retain, is called the beam width. Let’s use a beam of size 2 and see what happens.

Initially, your first hypothesis consists of only one word, “Mary,” and a score of 0. When you move on to the next word, the word you chose is appended to the hypothesis, and the score is incremented by the score of the path you have just taken.

For example, when you move on to “did,” it will make a new hypothesis consisting of “Mary did” and a score of 1.

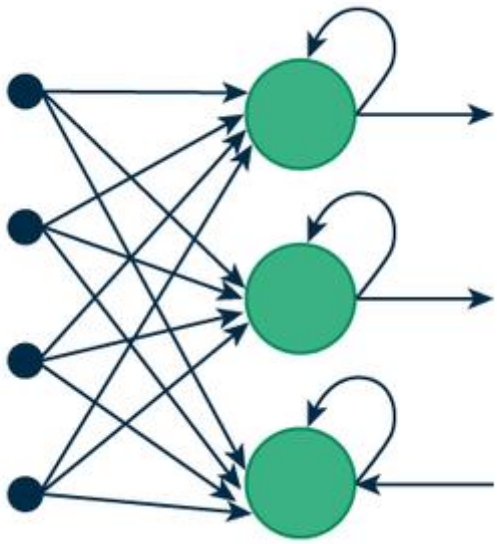
If you have multiple words to choose from at any particular timestep, a hypothesis can spawn multiple child hypotheses. At timestep 2, you have three different choices—“you,” “not,” and

“n’t”—which generate three new child hypotheses: [Mary did you] (6), [Mary did not] (4), and [Mary did n’t] (3). And here’s the key part of beam search decoding: because there’s only so much room in the beam, any hypotheses that are not good enough fall off of the beam after sorting them by their scores. Because the beam can hold up to only two hypotheses in this example, anything except the top two gets kicked out of the beam, which leaves [Mary did you] (6) and [Mary did not] (4).

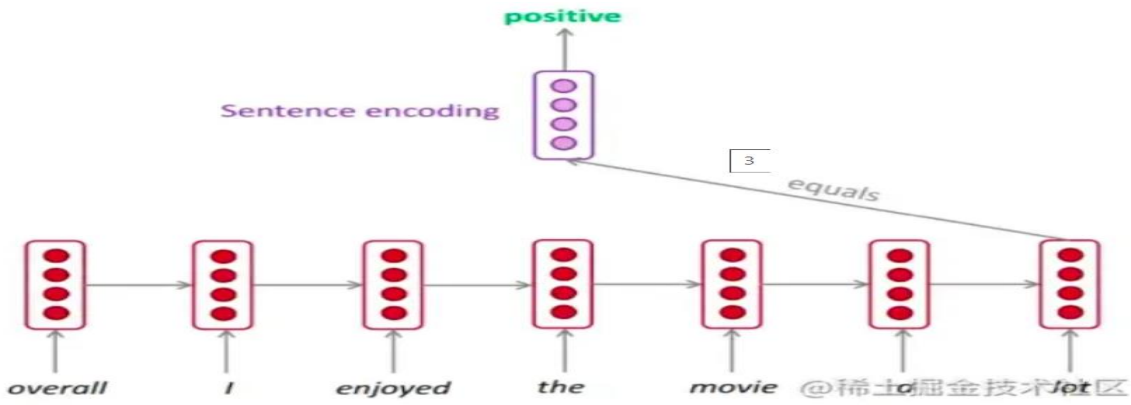
At timestep 3, each remaining hypothesis can spawn up to two child hypotheses. The first one ([Mary did you] (6)) will generate [Mary did you know] (8) and [Mary did you do] (7), whereas the second one ([Mary did not] (4)) turns into [Mary did not slap] (9). These three hypotheses are sorted by their scores, and the best two will be returned as the result of the beam search decoding.

RNNs and their Shortcomings-Introducing CNNs

- A recurrent neural network (RNN) is a deep learning model that is trained to process and convert a sequential data input into a specific sequential data output.
- Sequential data is data—such as words, sentences, or time-series data.

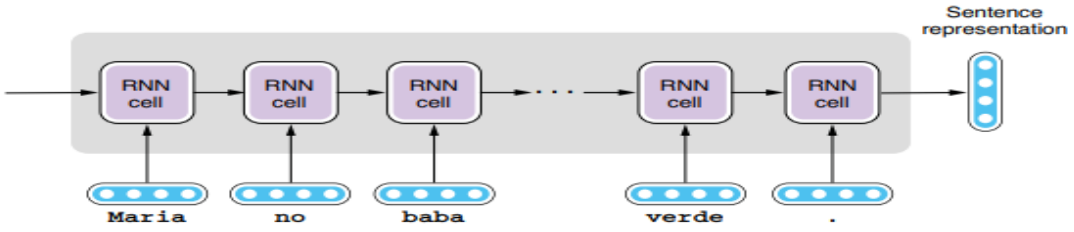


- RNN is a type of neural network that has a “loop” in it, which processes the input sequence one element at a time from the beginning until the end.
- The internal loop variable, which is updated at every step, is called the hidden state.
- When the RNN finishes processing the entire sequence, the hidden state at the final timestep represents the compressed content of the input sequence, which can be used for NLP tasks including sentence classification.



1. Select a sequence of words (sentences) as input
2. Find the word embedding to convert the word to a vector. Unlike neural networks, RNN does not connect all word vectors into one matrix since the purpose of RNN is to absorb information from each word vector separately to obtain sequential information.
3. For each step (each word vector), we use the current hidden state (multiplied by the weight matrix) and the information in the new word vector to calculate the current hidden state. When the model reaches the last word of the text, we will get a hidden state that contains information from the previous hidden state (and the previous word).
4. In terms of the problem we are solving, there are several ways to calculate the output. For example, if we are studying a language model (find the direct word after a sentence), we can place the last hidden state through another linear layer and activation function. As such, it makes sense to use the last hidden state since the next word may be the most relevant to the previous word. If we are doing sentiment analysis, we can use the last hidden state, but a more effective method is to calculate the output state for all the previous hidden states and obtain the average of these output states. The basic principle of using the output state of all previously hidden states is that the overall mood may be implied anywhere in the text (not just from the last word).

- Alternatively, you can take out the hidden state after every step and use it to assign labels (such as PoS and named entity tags) to individual words.
- The structure that is applied repeatedly in the loop is called a cell.
- An RNN with a simple multiplication and nonlinearity is called a vanilla RNN.
- On the other hand, LSTM and GRU-based RNNs use more complicated cells that employ memory and gating.



Advantages over Neural Networks

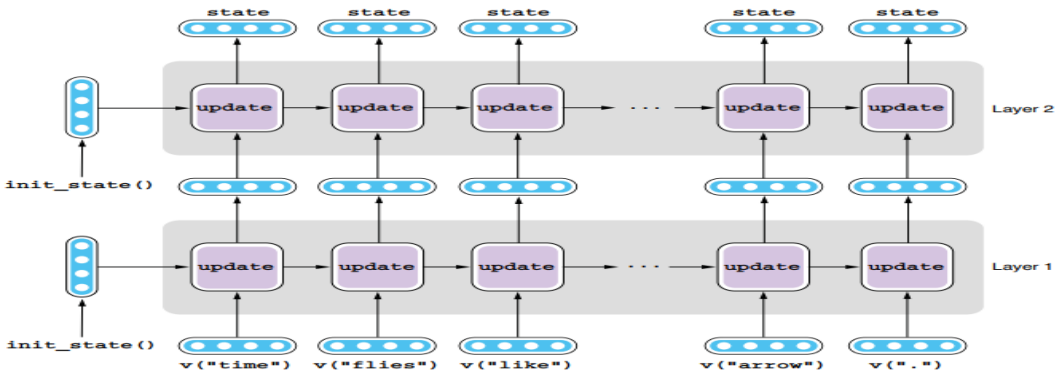
1. It can handle any length of the input, which solves the problem of the fixed window size of the neural network.
2. Applying the same weight matrix to each input is a problem faced by neural networks. Having the same weight matrix makes it more effective to use RNN to optimize the weight matrix.
3. The model size remains constant for any length of the input.
4. In theory, the RNN would still consider many words at the front of a sentence, which would provide the model with context for the entire text. The context will be important in our sentiment analysis exercises for film reviews.

RNNs are a powerful tool in modern NLP with a wide range of applications; however, they are not without shortcomings.

Disadvantages

1) RNNs are slow—they need to scan the input sequence element by element no matter what. Their computational complexity is proportional to the length of the input sequences.

2) Due to their sequential nature, RNNs are hard to parallelize. Think of a multilayer RNN where multiple RNN layers are stacked on top of each other. In a naive implementation, each layer needs to wait until all the layers below it finish processing the input



3) Third, the RNN structure is inefficient for some tasks.

For example, detecting grammatical English sentences .

If a sentence contains phrases such as “I am” and “you are,” it’s grammatical.

If it contains “I are” or “you am,” it’s not.

we built a simple LSTM-RNN with a nonlinearity to recognize the grammaticality of two word sentences with a vocabulary of four words. But what if you need to classify whether an arbitrary long sentence with a very large vocabulary is grammatical? Sudddenly, this process starts to sound very complex. Your LSTM needs to learn to pick up the signal (subject-verb agreement) from a large amount of noise (all other words

and phrases that have nothing to do with agreement), while learning to do all this using the update operation that gets repeated for every single element of the input.

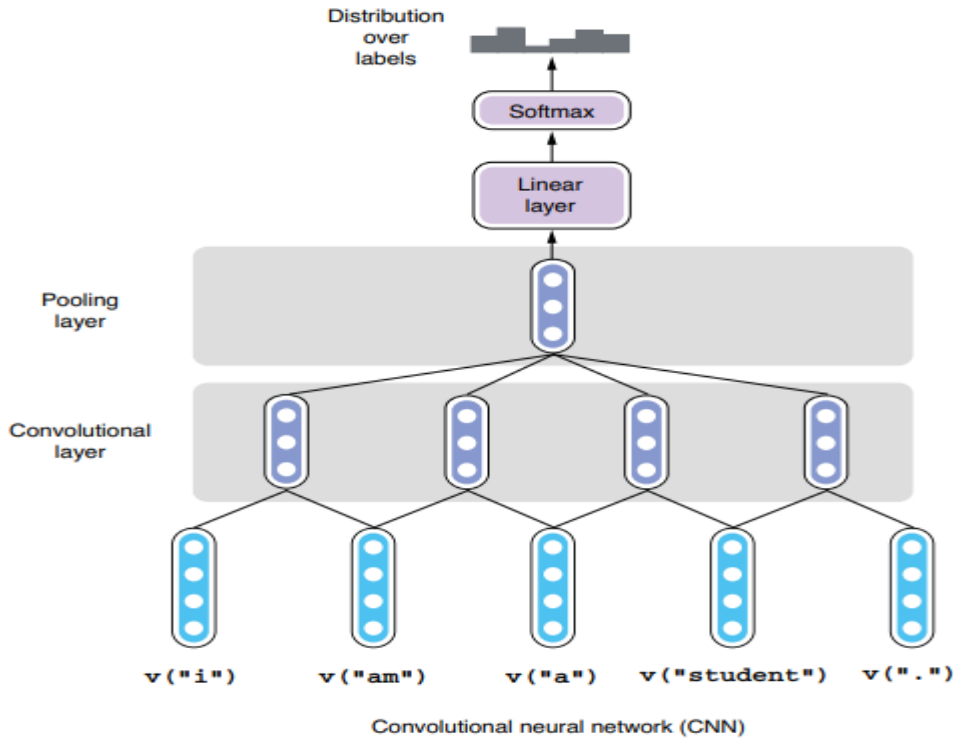
But if you think about it, no matter how long the sentence is or how large the vocabulary is, your network's job should still be quite simple—if the sentence contains valid collocations (such as "I am" and "you are"), it's grammatical. Otherwise, it's not.

Pattern matching for sentence classification

- Text classification in general, many tasks can be effectively solved by this "pattern matching."
- spam filtering, for example—if you want to detect spam emails, simply look for words and phrases such as "terrorism" and "business opportunity" without even reading the entire email; it doesn't matter where these patterns appear.
- If you want to detect sentiment from movie reviews, detecting positive and negative words such as "amazing" and "awful" would go a long way.
- Learning and detecting such local linguistic patterns, regardless of their location, is an effective and efficient strategy for text-classification tasks, and possibly for other NLP tasks as well.

Convolutional neural networks (CNNs)

- A CNN is a type of neural network that involves a mathematical operation called convolution, which, put simply, detects local patterns that are useful for the task at hand.
- A CNN usually consists of one or more convolutional layers, which do convolution, and pooling layers, which are responsible for aggregating the result of convolution.
-



- CNNs, which are inspired by the visual system in the human brain, have been widely used for computer vision tasks such as image classification and object detection.
- In recent years, the use of CNNs has been increasingly popular in NLP, especially for tasks such as text classification, sequential labeling, and machine translation

CNN Layers

convolutional layers are the essential part of the CNN architecture.

The term convolution is all about pattern matching

Pattern matching using filters

- Convolutional layers are the most important component in CNNs.
- Convolutional layers apply a mathematical operation called convolution to input vectors and produce output.

what is convolution?

- Understanding the strict definition of convolution requires knowing linear algebra.
- Lets consider the grammaticality-detection example and see how we'd apply a convolutional layer to the task.

- To recap, our neural network receives a two-word sentence as an input and needs to distinguish grammatical sequences from ungrammatical ones.
- There are only four words in the vocabulary—"**I," "you," "am," and "are,"** which are represented by word embeddings.
- Similarly, there are only four possibilities for the input sentence—
- **"I am," "I are," "you am," and "you are."**
- You want the network to produce 1s for the first and the last cases and 0s for others.

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

Word embeddings

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

Patterns and desired output

- Now, let's represent word embeddings as patterns.
- We'll draw a black circle for value –1 and a white one for 1. Then you can represent each word vector as a pair of two circles.
- Similarly, you can represent each two-word sentence as a small "patch" of two vectors, or four circles.
- Our task is beginning to look more like a pattern-recognition task, where the network needs to learn black-and-white patterns that correspond to grammatical sentence.

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

Word embeddings

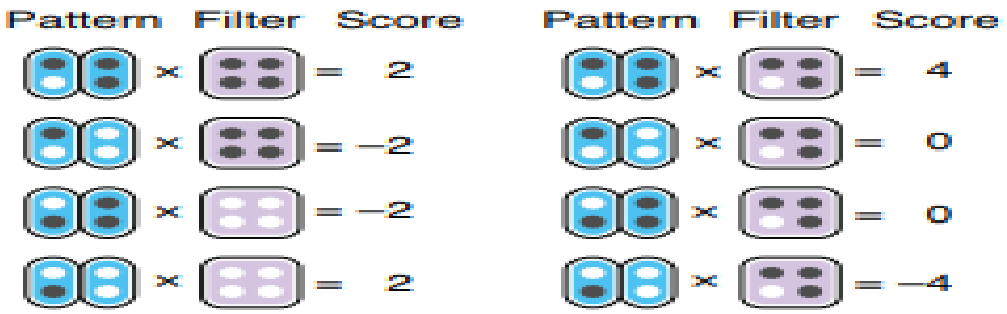
Word 1	Word 2	x1	x2	Pattern	Desired
I	am	[-1, 1]	[-1, -1]		1
I	are	[-1, 1]	[1, 1]		0
you	am	[1, -1]	[-1, -1]		0
you	are	[1, -1]	[1, 1]		1

Patterns and desired output

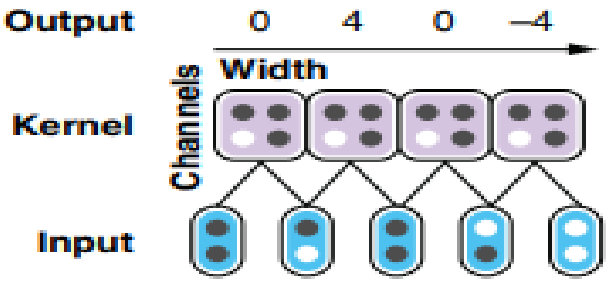
- Then, let's think of a "filter" of the same size (two circles × two circles).
- Each circle of this filter is also either black or white, corresponding to values –1 and 1.

- Try to look at a pattern through this filter and determine whether the pattern is the one you are looking for. You do this by putting the filter over a pattern and counting the number of color matches between the two.
- For each one of four positions, you get a score of +1 if the colors match (blackblack or white-white) and a score of –1 if they don't (black-white or white-black).

Our final score is the sum of four scores, which varies from –4 (no matches) to +4 (four matches).



- The score you get varies depending on the pattern and the filter, but as you can see in the figure, the score becomes larger when the filter looks similar to the pattern and becomes smaller when the two are not similar.
- we get the largest score (4) when the two match exactly and the smallest score (–4) when the two are exactly opposite.
- The filter acts as a pattern detector against the input. Although this is a very simplified example, it basically shows what a convolutional layer is doing.
- In convolutional neural networks, such filters are called **kernels**.
- In a more general setting, you have an input sentence of arbitrary length, and you slide a kernel over the sentence from left to right.



- The kernel is repeatedly applied to two consecutive words to produce a sequence of scores. Because the kernel we are using here covers two words, it is said to have a size of 2.

- Also, because there are two dimensions in the input embeddings (which are called channels), the number of the kernel's input channels is 2.

NOTE:

- The reason embedding dimensions are called channels is because CNNs are most commonly applied to computer vision tasks where the input is often a 2-D image of different channels that correspond to intensities of different colors (such as red, green, and blue).
- In computer vision, kernels are two dimensional and move over the input 2-D images, which is also called 2-D convolution. In NLP, however, kernels are usually one-dimensional (1-D convolution) and have only one size

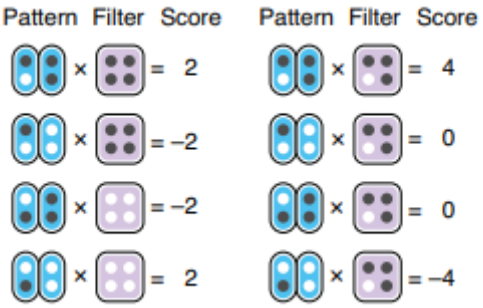
Rectified linear unit (ReLU)

- How we can get the desired output using kernels.

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

Patterns and desired output

- How do we use the filter shown in the second column



- The kernel, which we'll call kernel 1 from now on, matches the first pattern exactly and gives it a high score, while giving zero or negative scores to others.
- Figure below shows the score (called score 1) when kernel 1 is applied to each pattern.

Word 1	Word 2	Pattern	Kernel 1	Score 1	Desired
I	am			4	1
I	are			0	0
you	am			0	0
you	are			-4	1

- Let's forget the magnitude of the scores for now and focus on their signs (positive and negative).
- The signs for the first three patterns match between Score 1 and Desired, but not for the last pattern. To score it correctly—that is, to give it a positive score—you need to use another filter that matches the last pattern exactly.
- Let's call this kernel 2. Figure below shows the score (called score 2) when kernel 2 is applied to each pattern

Word 1	Word 2	Pattern	Kernel 2	Score 2	Desired
I	am			-4	1
I	are			0	0
you	am			0	0
you	are			4	1

- Kernel 2 can give correct scores that match the signs of the desired ones for the last three patterns, but not for the first one.
- But if you observe figures below carefully, it looks like you could get closer to the desired scores if there was a way to somehow disregard the output when a kernel gives negative scores and then combine the scores from multiple kernels.

Word 1	Word 2	Pattern	Kernel 1	Score 1	Desired
I	am			4	1
I	are			0	0
you	am			0	0
you	are			-4	1

Word 1	Word 2	Pattern	Kernel 2	Score 2	Desired
I	am			-4	1
I	are			0	0
you	am			0	0
you	are			4	1

Let's think of a function that clamps any negative input to zero while passing any positive values through unchanged. In Python, this function can be written as follows:

```
def f(x):
    if x >= 0:
        return x
    else:
        return 0
```

or even simpler

```
def f(x):
    return max(0, x)
```

we can disregard negative values by applying this function to score 1 and score 2, as shown in figures below

Word 1	Word 2	Pattern	Kernel 1	Score 1	f(Score 1)	Desired
I	am			4	4	1
I	are			0	0	0
you	am			0	0	0
you	are			-4	0	1

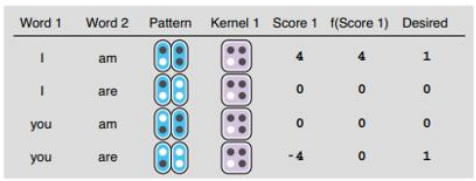
Applying ReLU to score 1

Word 1	Word 2	Pattern	Kernel 2	Score 2	f(Score 2)	Desired
I	am			-4	0	1
I	are			0	0	0
you	am			0	0	0
you	are			4	4	1

Applying ReLU to score 2

- This function, which is called a rectified linear unit, or ReLU, is one of the simplest yet most commonly used activation functions in deep learning.
- It is often used with a convolutional layer, and although it is very simple (all it does is just clamp negative values to zero), it is still an activation function that enables neural networks to learn complex nonlinear functions.
- It also has favorable mathematical properties that make it easier to optimize the network, although the theoretical details are beyond the scope of this book

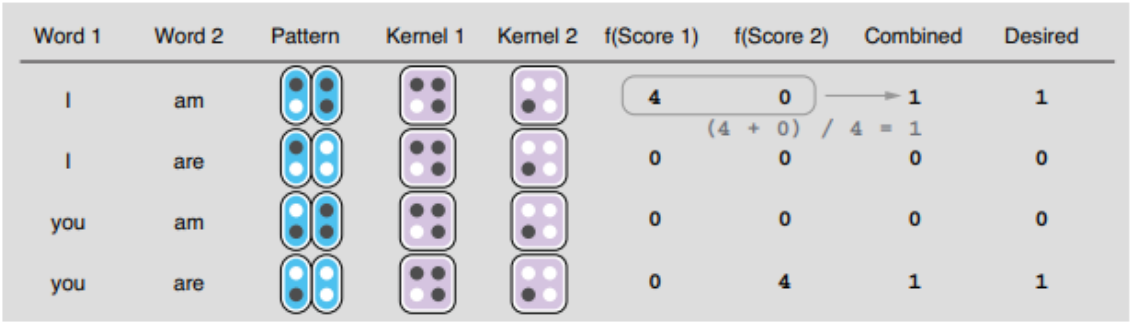
Combining scores If you look at both figures below, the “clamped” scores—shown in the f(Score 1) and f(Score 2) columns—capture the desired scores at least partially.



Word 1	Word 2	Pattern	Kernel 1	Score 1	f(Score 1)	Desired
I	am			4	4	1
I	are			0	0	0
you	am			0	0	0
you	are			-4	0	1

Word 1	Word 2	Pattern	Kernel 2	Score 2	f(Score 2)	Desired
I	am			-4	0	1
I	are			0	0	0
you	am			0	0	0
you	are			4	4	1

- All we need to do is combine them together (by summing) and adjust the range (by dividing by 4). Figure below shows the result of this



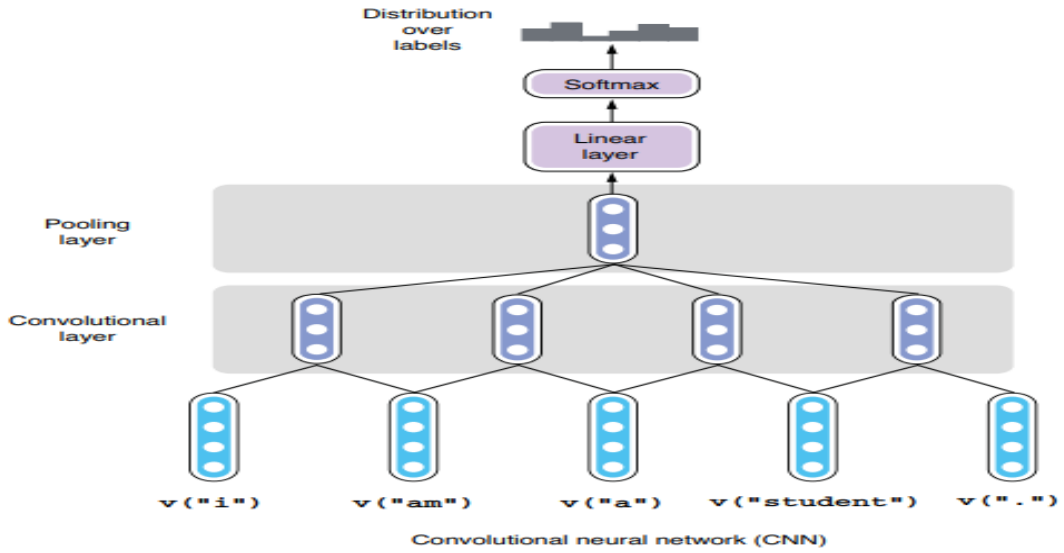
Word 1	Word 2	Pattern	Kernel 1	Kernel 2	f(Score 1)	f(Score 2)	Combined	Desired
I	am				4	0	1	1
I	are				0	0	0	0
you	am				0	0	0	0
you	are				0	4	1	1

(4 + 0) / 4 = 1

- After combining, the scores match the desired outcomes exactly.
- All we did so far was design kernels that match the patterns we want to detect and then simply combine the scores.
- Compare this to the RNN example where we needed to use some complicated numeric computation to derive the parameters.
- Hopefully this example is enough to show you how simple and powerful CNNs can be for text classification!

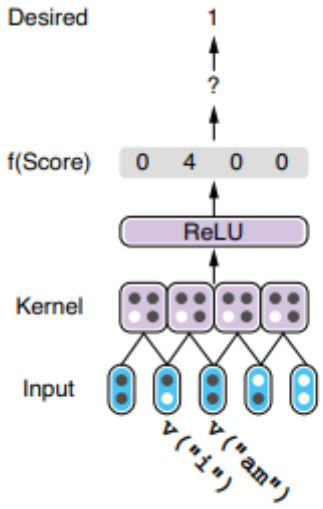
Pooling layers

- The input to a CNN can be of arbitrary length.
- CNN needs to not only detect patterns but also find them in a potentially large amount of noise in the input.
- As we saw in section 7.2, you slide a kernel over the sentence from left to right, and the kernel is repeatedly applied to two consecutive words to produce a sequence of scores.



what to do with these produced scores?

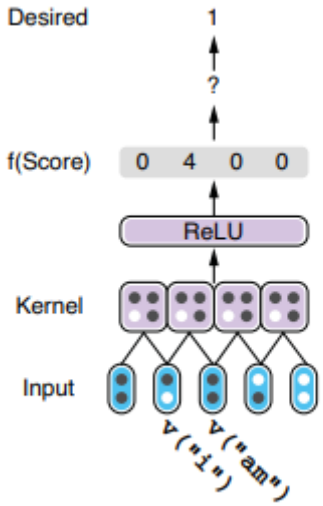
Specifically, what operation should we use in the "?" position to derive the desired score? in figure below.



This operation needs to have some properties—it must be something that can be applied to an arbitrarily large number of scores, because the sentence can be very long.

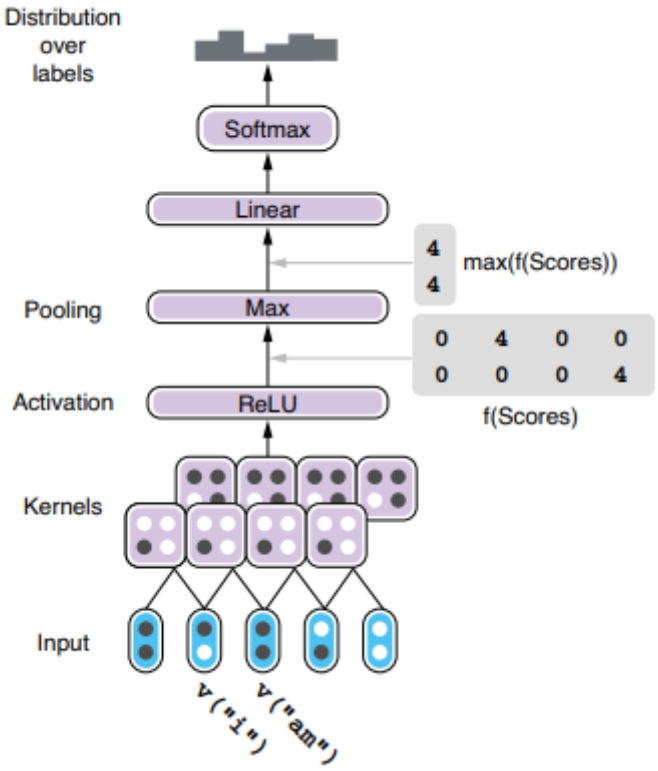
It also needs to aggregate the scores in a way that is agnostic to where the target pattern (word embeddings for "I am") is in the input sentence.

The simplest thing you can do to aggregate the scores is to take their maximum. Because the largest score in figure below is 4,



it will become the output of this layer. This aggregation operation is called pooling, and the neural network substructure that does pooling is called a pooling layer.

The pooled score will be fed to a linear layer, optionally combined with scores from other kernels, and used as a predicted score. This entire process is illustrated in figure below.



Now we have a fully functional CNN!

- The output from the linear layer is fed to softmax to produce a probability distribution over labels. These predicted values are then compared with the true labels to produce the loss and used for optimizing the network.
- Because the kernels in a CNN do not depend on each other (unlike RNNs, where one cell needs to wait until all the preceding cells finish processing the input), CNNs are computationally efficient.
- GPUs can process these kernels in parallel without waiting on other kernels' output. Due to this property, CNNs are usually faster than RNNs of similar size