

Interested in the most state-of-the-art NLP techniques? Check out our **Super Study Guide: Transformers & LLMs** (<https://superstudy.guide/transformers-large-language-models/>)!

(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#cs-230---deep-learning>)CS 230 - Deep Learning  
(<https://stanford.edu/~shervine/teaching/cs-230>)

English

□

Convolutional Neural Networks

Recurrent Neural Networks

Tips and tricks

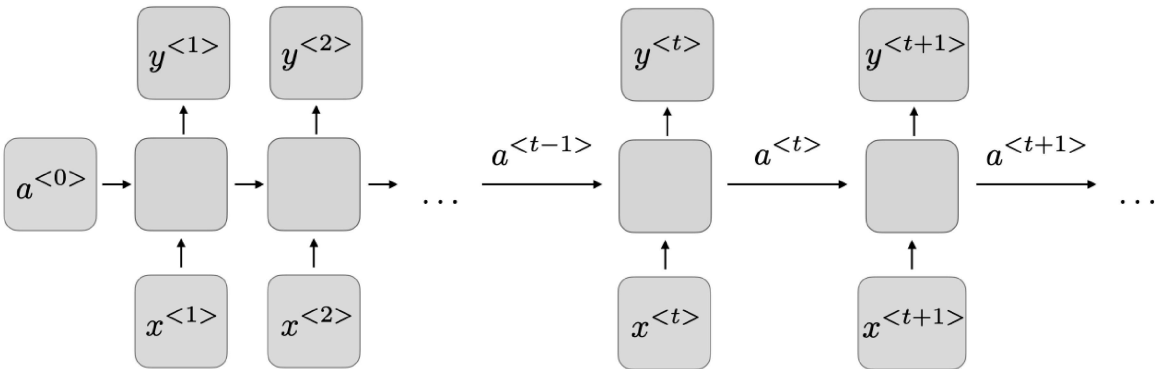
# (<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#cheatsheet>)Recurrent Neural Networks cheatsheet

☆ Star 6,368

By Afshine Amidi (<https://twitter.com/afshinea>) and Shervine Amidi (<https://twitter.com/shervine>)

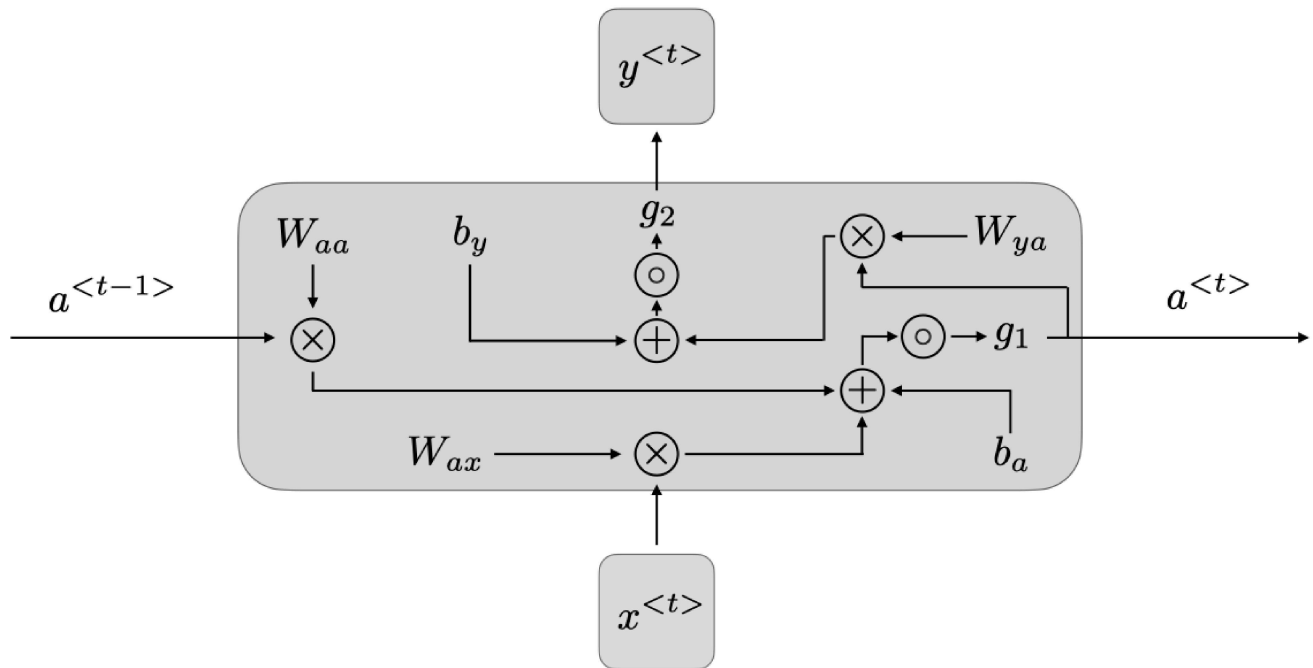
## (<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#overview>) Overview

❑ **Architecture of a traditional RNN** — Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



For each timestep  $tt$ , the activation  $a^{<t>}$   $a^{<t>}$  and the output  $y^{<t>}$   $y^{<t>}$  are expressed as follows:

$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$  and  $y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$   
 $(W_{aaa}a^{<t-1>} + W_{axx}x^{<t>} + b_a)$  and  $y^{<t>} = g_2(W_{yaa}a^{<t>} + b_y)$   
 where  $W_{ax}, W_{aa}, W_{ya}, b_a, b_y, W_{ax}, W_{aa}, W_{ya}, b_a, b_y$  are coefficients that are shared temporally and  $g_1, g_2$  activation functions.



The pros and cons of a typical RNN architecture are summed up in the table below:

Advantages	Drawbacks
<ul style="list-style-type: none"> <li>• Possibility of processing input of any length</li> <li>• Model size not increasing with size of input</li> <li>• Computation takes into account historical information</li> <li>• Weights are shared across time</li> </ul>	<ul style="list-style-type: none"> <li>• Computation being slow</li> <li>• Difficulty of accessing information from a long time ago</li> <li>• Cannot consider any future input for the current state</li> </ul>

**❑ Applications of RNNs** — RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$ $T_x = T_y = 1$		Traditional neural network

<p>One-to-many</p> $T_x = 1, T_y > 1$ $T_x = 1, T_y > 1$		Music generation
<p>Many-to-one</p> $T_x > 1, T_y = 1$ $T_x > 1, T_y = 1$		Sentiment classification
<p>Many-to-many</p> $T_x = T_y, T_x = T_y$		Name entity recognition
<p>Many-to-many</p> $T_x \neq T_y, T_x \neq T_y$ $= T_y$		Machine translation

❑ **Loss function** — In the case of a recurrent neural network, the loss function  $LL$  of all time steps is defined based on the loss at every time step as follows:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L(\hat{y}^{<t>}, y^{<t>})$$

$$L(y) = \sum_{t=1}^{T_y} L(y^{<t>})$$

$$L(y^{<t>}) = \sum_{t=1}^{T_y} L(y^{<t>})$$

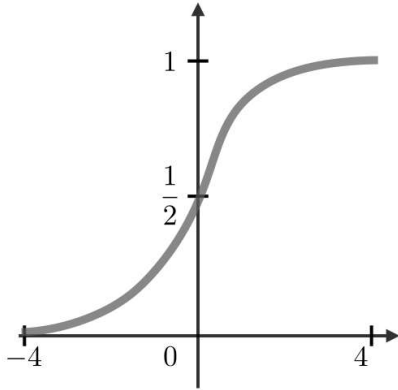
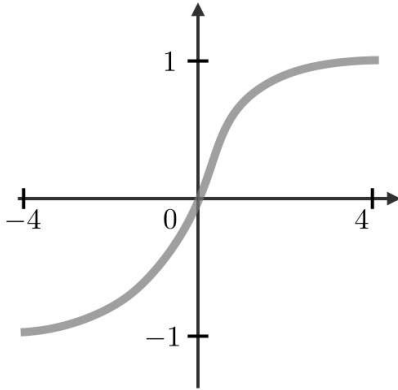
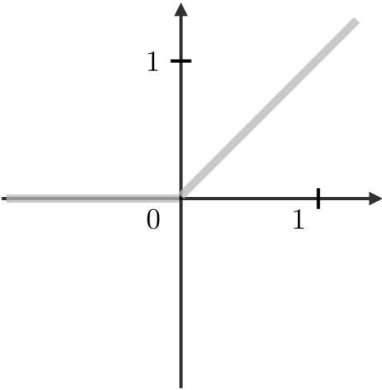
❑ **Backpropagation through time** — Backpropagation is done at each point in time. At timestep  $TT$ , the derivative of the loss  $LL$  with respect to weight matrix  $WW$  is expressed as follows:

$$\frac{\partial L^{(T)}}{\partial W} = \sum_{t=1}^T \frac{\partial L^{(T)}}{\partial W} \Big|_{(t)} \quad \partial W \partial L^{(T)} = \sum_{t=1}^T \partial W \partial L^{(T)} \Big|_{(t)}$$

(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#architecture>)

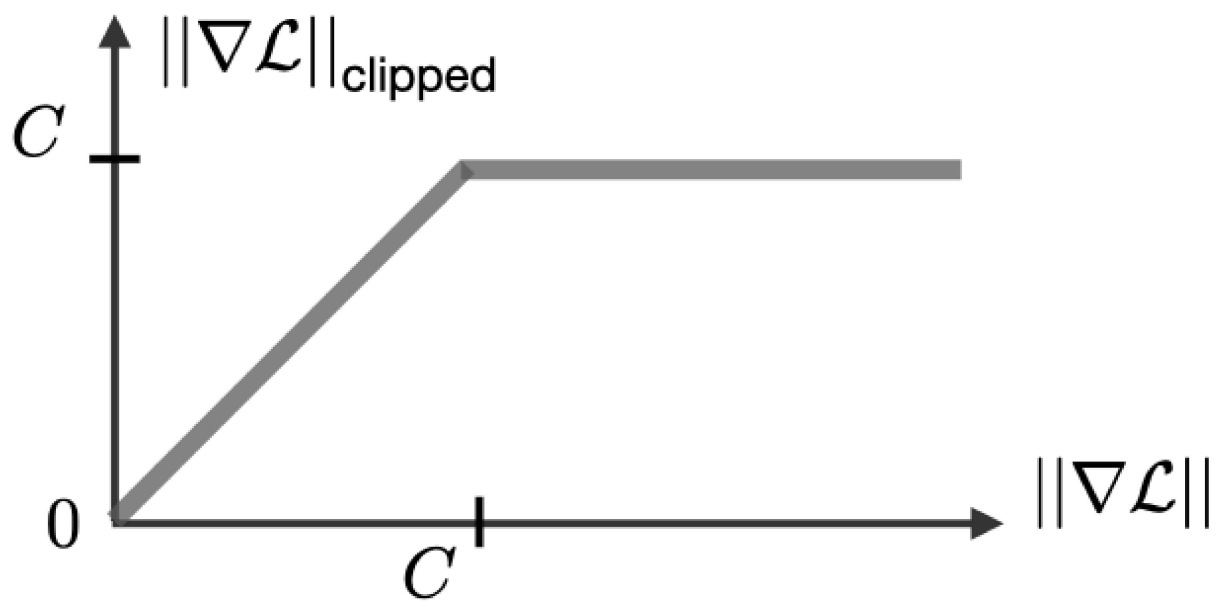
## Handling long term dependencies

□ **Commonly used activation functions** — The most common activation functions used in RNN modules are described below:

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1 + e^{-z}}$ $g(z) = 1 + e^{-z}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ $g(z) = e^z + e^{-z}e^z - e^{-z}$	$g(z) = \max(0, z)$ $g(z) = \max(0, z)$
		

□ **Vanishing/exploding gradient** — The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

□ **Gradient clipping** — It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.



□ **Types of gates** — In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted  $\Gamma$  and are equal to:

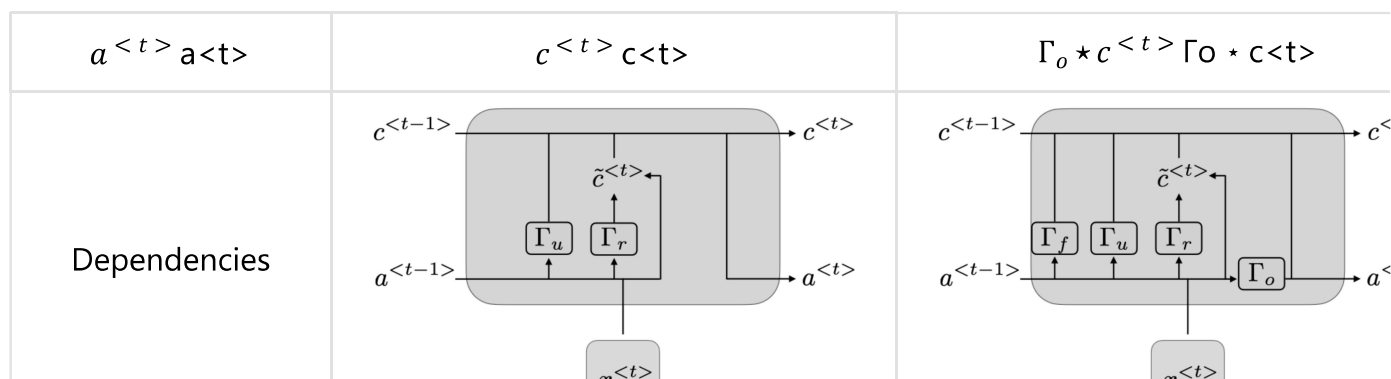
$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b) \quad \Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where  $W, U, b$  are coefficients specific to the gate and  $\sigma$  is the sigmoid function. The main ones are summed up in the table below:

Type of gate	Role	Used in
Update gate $\Gamma_u$	How much past should matter now?	GRU, LSTM
Relevance gate $\Gamma_r$	Drop previous information?	GRU, LSTM
Forget gate $\Gamma_f$	Erase a cell or not?	LSTM
Output gate $\Gamma_o$	How much to reveal of a cell?	LSTM

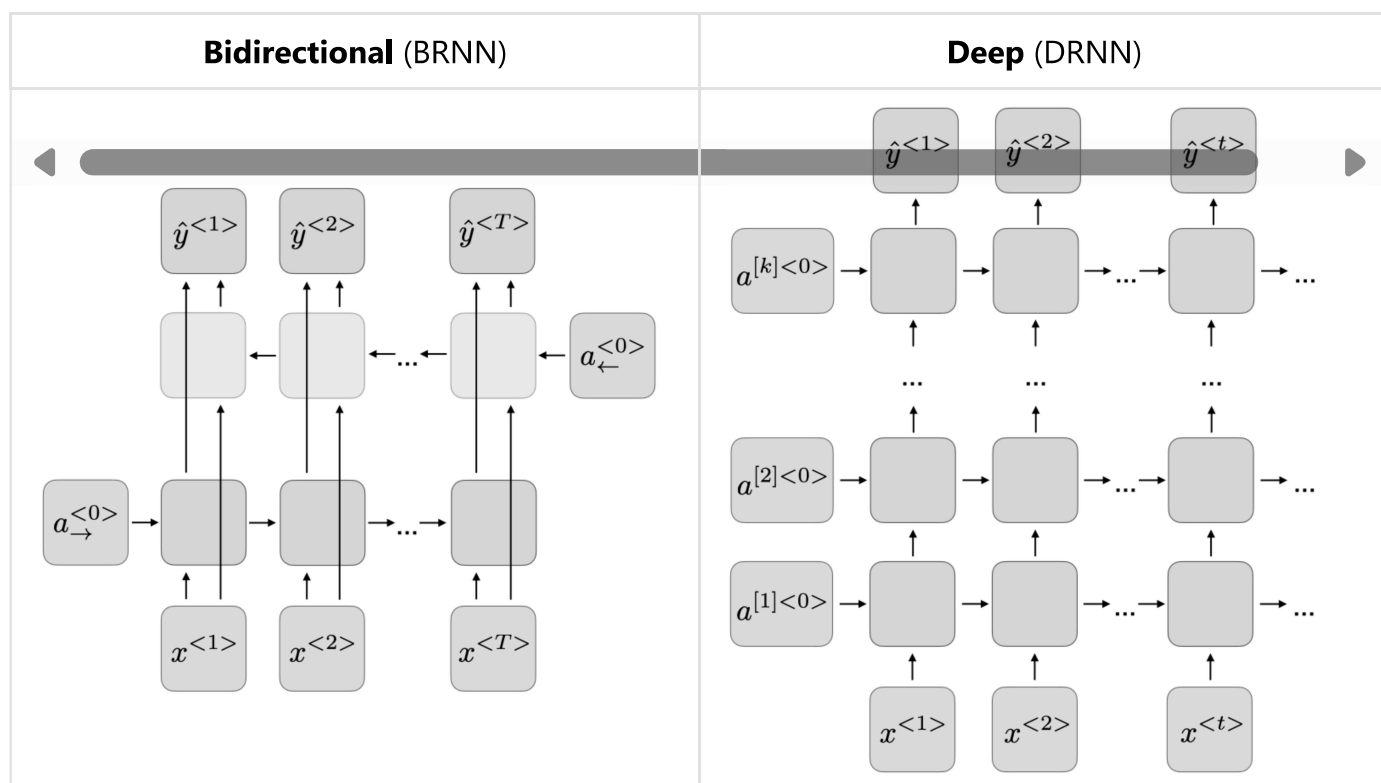
□ **GRU/LSTM** — Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

Characterization	Gated Recurrent Unit (GRU)	Long Short-Term Memory (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$	$\Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} + \Gamma_o * c^{<t>} + \Gamma_f * c^{<t-1>}$



Remark: the sign  $\star$  denotes the element-wise multiplication between two vectors.

□ **Variants of RNNs** — The table below sums up the other commonly used RNN architectures:



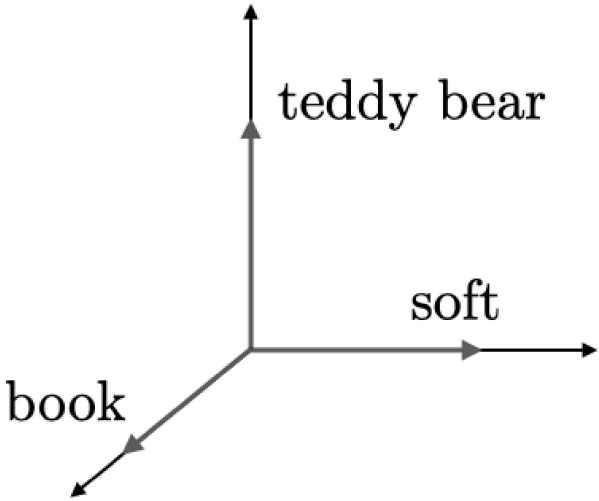
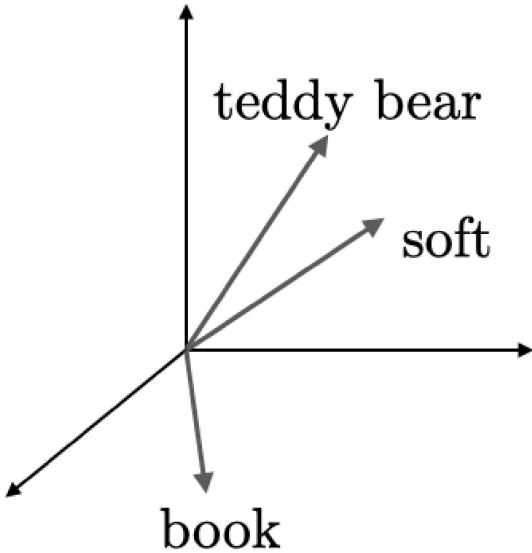
(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#word-representation>)

## Learning word representation

In this section, we note  $V$  the vocabulary and  $|V|$  its size.

### Motivation and notations

□ **Representation techniques** — The two main ways of representing words are summed up in the table below:

1-hot representation	Word embedding
	
<ul style="list-style-type: none"> <li>• Noted <math>o_{w w}</math></li> <li>• Naive approach, no similarity information</li> </ul>	<ul style="list-style-type: none"> <li>• Noted <math>e_{w w}</math></li> <li>• Takes into account words similarity</li> </ul>

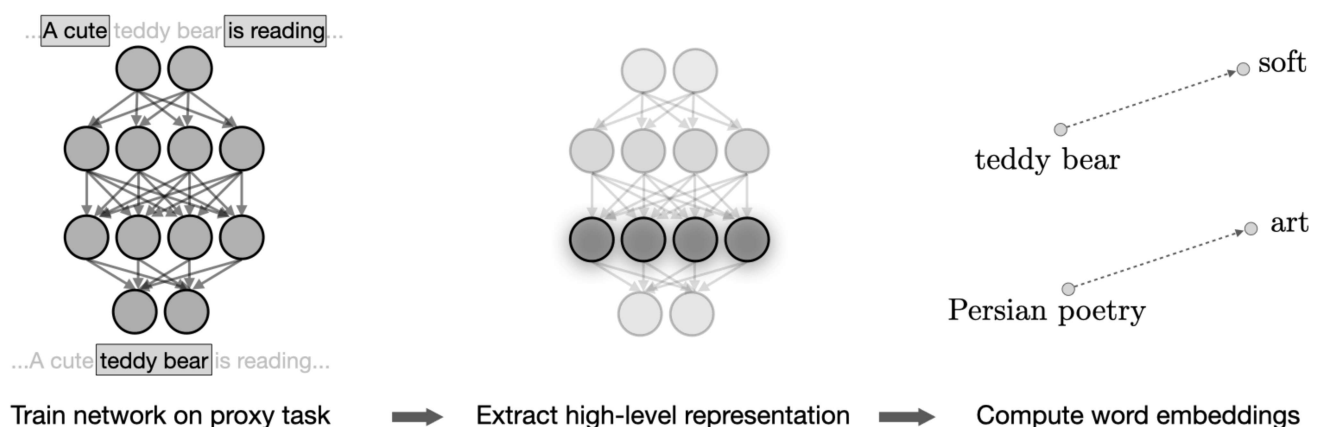
□ **Embedding matrix** — For a given word  $w$ , the embedding matrix  $E$  is a matrix that maps its 1-hot representation  $o_w$  to its embedding  $e_w$  as follows:

$$e_w = E o_w = E w$$

*Remark: learning the embedding matrix can be done using target/context likelihood models.*

## Word embeddings

□ **Word2vec** — Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW.



□ **Skip-gram** — The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word  $t$  happening with a context word  $c$ . By noting  $\theta_t$  a parameter associated with  $t$ , the probability  $P(t|c)$  is given by:

$$P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)} \quad P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)}$$

*Remark: summing over the whole vocabulary in the denominator of the softmax part makes this model computationally expensive. CBOW is another word2vec model using the surrounding words to predict a given word.*

□ **Negative sampling** — It is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of  $k$  negative examples and 1 positive example. Given a context word  $c$  and a target word  $t$ , the prediction is expressed by:

$$P(y = 1|c, t) = \sigma(\theta_t^T e_c) \quad P(y = 1|c, t) = \sigma(\theta_t^T e_c)$$

*Remark: this method is less computationally expensive than the skip-gram model.*

□ **GloVe** — The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurrence matrix  $X$  where each  $X_{i,j}$  denotes the number of times that a target  $i$  occurred with a context  $j$ . Its cost function  $J$  is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij}) (\theta_i^T e_j + b_i + b'_j - \log(X_{ij}))^2 \quad J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij}) (\theta_i^T e_j + b_i + b'_j - \log(X_{ij}))^2$$

where  $f$  is a weighting function such that  $X_{i,j} = 0 \Rightarrow f(X_{i,j}) = 0$  and  $X_{i,j} > 0 \Rightarrow f(X_{i,j}) > 0$ .

Given the symmetry that  $e$  and  $\theta$  play in this model, the final word embedding  $e_w^{(final)}$  is given by:

$$e_w^{(final)} = \frac{e_w + \theta_w}{2} \quad e_w^{(final)} = \frac{e_w + \theta_w}{2}$$

*Remark: the individual components of the learned word embeddings are not necessarily interpretable.*

(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#comparing-words>)

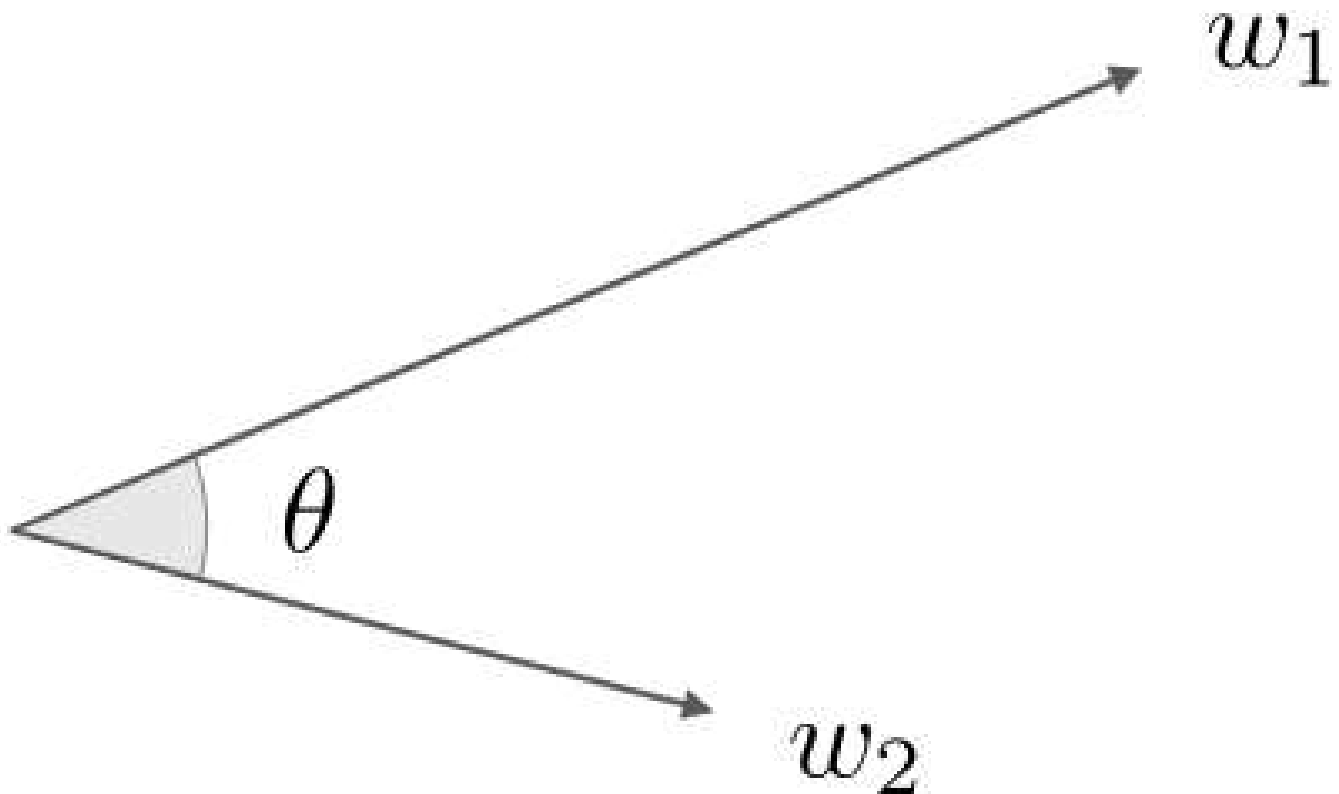


# Comparing words

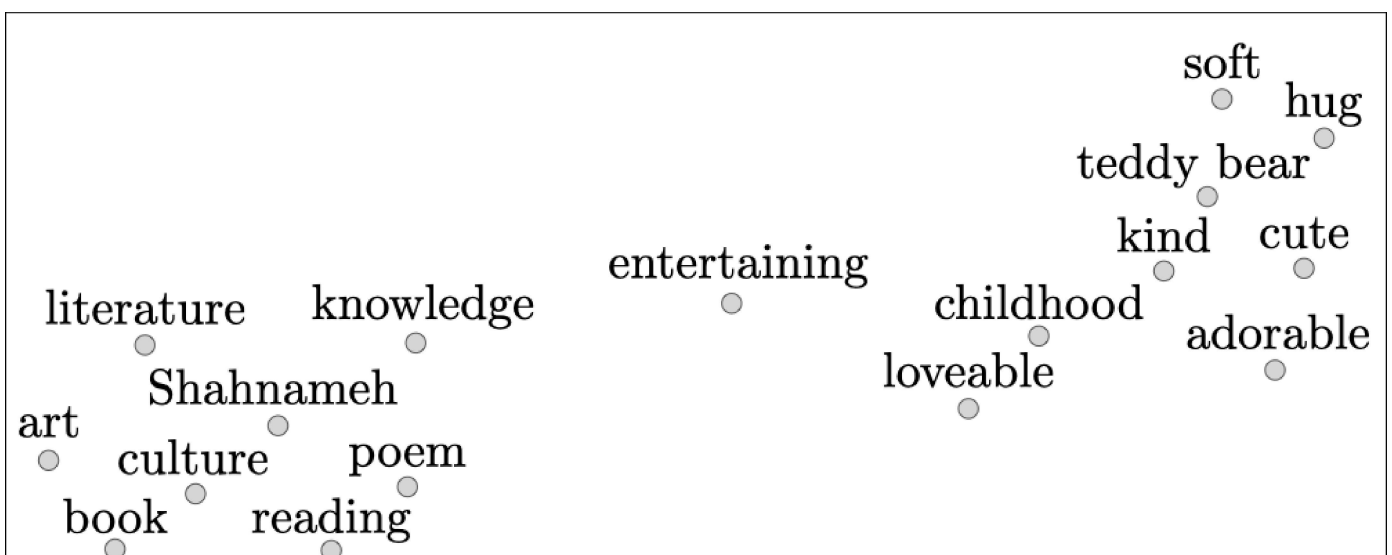
□ **Cosine similarity** — The cosine similarity between words  $w_1$  and  $w_2$  is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{\|w_1\| \|w_2\|} = \cos(\theta) \quad \text{similarity} = \frac{|w_1 \cdot w_2|}{\|w_1\| \|w_2\|} = \cos(\theta)$$

Remark:  $\theta$  is the angle between words  $w_1$  and  $w_2$ .



□ **tt-SNE** — tt-SNE (tt-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.



(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#language-model>)

## Language model

❑ **Overview** — A language model aims at estimating the probability of a sentence  $P(y)$ .

❑  **$n$ -gram model** — This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.

❑ **Perplexity** — Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words  $T$ . The perplexity is such that the lower, the better and is defined as follows:

$$PP = \prod_{t=1}^T \left( \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)^{\frac{1}{T}}$$

$j(t)T1|$

Remark: PP is commonly used in  $t$ -SNE.

(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#machine-translation>)

## Machine translation

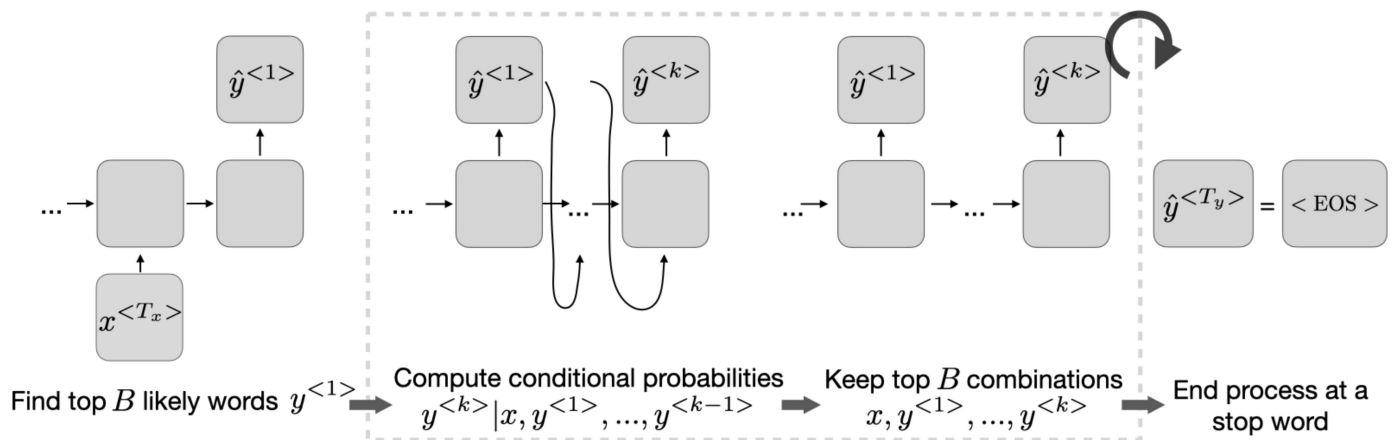
❑ **Overview** — A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model.

The goal is to find a sentence  $y$  such that:

$$y = \arg \max_{y < 1 >, \dots, y < T_y >} P(y < 1 >, \dots, y < T_y > | x) \\ y = y < 1 >, \dots, y < T_y > \arg \max P(y < 1 >, \dots, y < T_y > | x)$$

❑ **Beam search** — It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence  $y$  given an input  $x$ .

- Step 1: Find top  $BB$  likely words  $y^{<1>} y^{<1>}$
- Step 2: Compute conditional probabilities  $y^{<k>} | x, y^{<1>}, \dots, y^{<k-1>}$   
 $y^{<k>} | x, y^{<1>}, \dots, y^{<k-1>}$
- Step 3: Keep top  $BB$  combinations  $x, y^{<1>}, \dots, y^{<k>} x, y^{<1>}, \dots, y^{<k>}$



*Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.*

□ **Beam width** — The beam width  $BB$  is a parameter for beam search. Large values of  $BB$  yield to better result but with slower performance and increased memory. Small values of  $BB$  lead to worse results but is less computationally intensive. A standard value for  $BB$  is around 10.

□ **Length normalization** — In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\text{Objective} = \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log \left[ p(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}) \right] \quad \text{Objective} = T_y \alpha \frac{1}{T_y} \sum_{t=1}^{T_y} \log [p(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})]$$

$$\log [p(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})]$$

*Remark: the parameter  $\alpha$  can be seen as a softener, and its value is usually between 0.5 and 1.*

□ **Error analysis** — When obtaining a predicted translation  $\hat{y}$

that is bad, one can wonder why we did not get a good translation  $y^*$  by performing the following error analysis:

Case	$P(y^*   x) > P(\hat{y}   x) P(y^*   x) > P(y$	$P(y^*   x) \leq P(\hat{y}   x) P(y^*   x) \leq P(y$
	$  x)$	$  x)$

Root cause	Beam search faulty	RNN faulty
Remedies	Increase beam width	<ul style="list-style-type: none"> <li>• Try different architecture</li> <li>• Regularize</li> <li>• Get more data</li> </ul>

❑ **Bleu score** — The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on  $n$ -gram precision. It is defined as follows:

$$\text{bleu score} = \exp\left(\frac{1}{n} \sum_{k=1}^n p_k\right) \quad \text{bleu score} = \exp\left(\frac{1}{n} \sum_{k=1}^n p_k\right)$$

where  $p_n$  is the bleu score on  $n$ -gram only defined as follows:

$$p_n = \frac{\sum_{n\text{-gram} \in \hat{y}} \text{count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram} \in \hat{y}} \text{count}(n\text{-gram})} \quad p_n = \frac{\sum_{n\text{-gram} \in \hat{y}} \text{count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram} \in \hat{y}} \text{count}(n\text{-gram})}$$

$$\sum_{n\text{-gram} \in \hat{y}} \text{count}(n\text{-gram})$$

$$\sum_{n\text{-gram} \in \hat{y}} \text{count}_{\text{clip}}(n\text{-gram})$$

*Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.*

(<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#attention>)

## Attention

❑ **Attention model** — This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting  $\alpha^{<t,t'>}$  the amount of attention that the output  $y^{<t>}$  should pay to the activation  $a^{<t'>}$  and  $c^{<t>}$  the context at time  $t$ , we have:

$$c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1 \quad c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1$$

$$\sum_{t'} \alpha^{<t,t'>} = 1$$

*Remark: the attention scores are commonly used in image captioning and machine translation.*



*A cute teddy bear is reading Persian literature*



*A cute teddy bear is reading Persian literature*

□ **Attention weight** — The amount of attention that the output  $y^{<t>}$  should pay to the activation  $a^{<t'>}$  is given by  $\alpha^{<t,t'>}$  computed as follows:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t''=1}^{T_x} \exp(e^{<t,t''>})} \quad \alpha^{<t,t'>} = t'' = 1 \sum_{t''=1}^{T_x} \exp(e^{<t,t''>}) \exp(e^{<t,t'>})$$

Remark: computation complexity is quadratic with respect to  $T_x T_x$ .



(<https://twitter.com/shervinea>)



(<https://linkedin.com/in/shervineamidi>)



(<https://github.com/shervinea>)



(<https://scholar.google.com/citations?user=nMnMTm8AAAAJ>)



(<https://www.amazon.com/stores/author/B0B37XBSJL>)