

Learning text representation with Vectors

Pierre Colombo

pierre.colombo@centralesupelec.fr

MICS - CentraleSupélec

Introduction To (Deep) Natural Language Processing



Lectures Outline

This class will consist of both theory & **practice**:

1. Intro to Natural Language Processing
2. Learning text representation with Vectors
3. Training Large Language Models
4. Building NLP applications (RAG, VLLM)
5. (OPEN)

Today Lecture Outline

- **Representing Words** in Vectors
- **Representing Documents** in Vectors

Representation Techniques

- **Hand-Crafted Feature-Based** Representation
- **Count-Based** Representation
- **Prediction-Based** Representation
- **Transformer-Based** Representation

Framework

We assume:

- A **token** is the basic unit of discrete data, defined to be an item from a vocabulary indexed by 1, ..., V.
- A **document** is a sequence of N words denoted by $d = (w_1, w_2, \dots, w_N)$, where w_n is the N-th word in the sequence.
- A **corpus** is a collection of M documents denoted by $D = (d_1, d_2, \dots, d_M)$

Example: *Wikipedia, All the articles of the NYT in 2021 ...*

In this lecture, a token will be a word

What is a word?

There are many ways to define a word based on what aspect of language we consider (typography, syntax, semantics...)

Definition (Semantic):

*Words are the smallest linguistic expressions that are conventionally associated with a non-compositional meaning and can be articulated in isolation to convey semantic content.**

*Stanford Encyclopedia of Philosophy

Objective

Given a vocabulary w_1, \dots, w_V and a corpus D, our goal is to associate each word with a representation?

What do we want from this representation?

- identify a word (bijection)
- capture the similarities of words (based on morphology, syntax, semantics,...)
- Help us solve downstream tasks

NB: Vector-based representations of text are called *embedding*

1-Hot Encoding

Traditional way to represent words **as atomic symbols** with a unique integer associated with each word:

$$\{1=\text{movie}, 2=\text{hotel}, 3=\text{apple}, 4=\text{movies}, 5=\text{art}\}$$

Equivalent to represent words as 1-hot vectors:

$$\text{movie} = [1, 0, 0, 0, 0]$$
$$\text{hotel} = [0, 1, 0, 0, 0]$$

...

$$\text{art} = [0, 0, 0, 0, 1]$$

1-Hot Encoding

Most basic representation of any textual unit in NLP. Always start with it.

Implicit assumption: word vectors are an **orthonormal basis**

- orthogonal
- normalized

Problem 1: Not very informative

→ Weird to consider “movie” and “movies” as independent entities or to consider all words equidistant:

$$\| \text{house} - \text{home} \| = \| \text{house} - \text{car} \|$$

Problem 2: Polysemy

→ Should the Mouse of a computer get the same vector as the mouse animal?

Hand-Crafted Feature Representation

Example of potential features:

- Morphology : prefix, suffix, stem...
- Grammar : part of speech, gender, number,...
- Shape : capitalization, digit, hyphen

Those features can be defined based on relations to other words

- Synonyms of...
- Hyponyms of...
- Antonyms of...

Hand-Crafted Feature Representation

Example of potential features:

- Morphology : prefix, suffix, stem...
- Grammar : part of speech, gender, number,...
- Shape : capitalization, digit, hyphen

Those features can be defined based on relations to other words

- Synonyms of...
- Hyponyms of...
- Antonyms of...

We present one popular hand-crafted semantically based representation of words ⇒ the WordNet

Hand-Crafted Representations: Limits

- Requires **a lot of human annotations**
 - **Subjectivity** of the annotators
 - **Does not adapt** to new words (languages are not stationary!):
Mocktail, Guac, Fave, Biohacking were added to the Merriam-Webster Dictionary in 2018
- It **does not scale** easily to new languages, new concepts, new words...

How to Infer “Good” Representations with Data?

Distributional Hypothesis

You shall know a word by the company it keeps” Firth (1957)

Idea: Model the *context* of a word to build **its vectorial representation**

Example: What is the meaning of “ Bardiwac ” ?

- He handed her a glass of bardiwac .
- Beef dishes are made to complement the bardiwacs .
- Nigel staggered to his feet, face flushed from too much bardiwac .
- Malbec, one of the lesser-known bardiwac grapes, responds well to Australia’s sunshine.
- I dined off bread and cheese and this excellent bardiwac
- The drinks were delicious: blood-red bardiwac as well as light, sweet Rhenish.

→ bardiwac is a heavy red alcoholic beverage made from grapes

Distributional word representation in a nutshell

1. Define what is *the context* of a word
2. **Count** how many times each target word occurs in this context
3. Build vectors out of (a function of) these context occurrence counts

$$x_w = f(w, \text{Context}(w))$$

How to define “*the context*” of a word?

It can be defined as

- **The surrounding words** (left and right words)
- **All the other words** of the sentence/the paragraph
- All the words **after preprocessing and filtering-out some words**

How to Model the Context to get

$$x_w = f(w, \text{Context}(w))$$

Approach 1: Count-Based

1. Measure frequency of words in the context for each word in the vocabulary
2. Define vector representations based on those frequency

How to Model the Context to get

$$x_w = f(w, \text{Context}(w))$$

Approach 1: Count-Based

1. Measure frequency of words in the context for each word in the vocabulary
2. Define vector representations based on those frequency

Approach 2: Prediction-Based

Counting the Occurrences of the words in the context of dog

The **dog** barked in the **park**.
The owner of the **dog** put him
on the leash since he **barked**.

barked	++
park	+
owner	+
leash	+
co-occurrence # dog	

Co-Occurrence Matrix

	leash	walk	run	owner	pet	barked
dog	3	5	2	5	3	2
cat	0	3	3	2	3	0
lion	0	3	2	0	1	0
light	0	0	0	0	0	0
bark	1	0	0	2	1	0
car	0	0	1	3	0	0

Define vector representation based on the Co-Occurrence

	leash	walk	run	owner	pet	barked	the
dog	3	5	2	5	3	2	8
lion	0	3	2	0	1	0	6
light	0	0	0	0	0	0	5
bark	1	0	0	2	1	0	0
car	0	0	1	3	0	0	3

- **Naïve Approach:** Take the row of the co- occurrence matrix

Define vector representation based on the Co-Occurrence

	leash	walk	run	owner	pet	barked	the
dog	3	5	2	5	3	2	8
lion	0	3	2	0	1	0	6
light	0	0	0	0	0	0	5
bark	1	0	0	2	1	0	0
car	0	0	1	3	0	0	3

Limits:

- Representations depends **on the size of the corpus**
- Frequent words impacts a lot the representations
- Representations **very sensitive to change** in very infrequent words

Solution: Pointwise Mutual Information (PMI)

Idea: Instead of absolute co-occurrence statistics, use probability (relative) of co-occurrences

$$PMI(w_1; w_2) = \log \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

Solution: Pointwise Mutual Information (PMI)

Idea: Instead of absolute co-occurrence statistics, use probability (relative) of co-occurrences

$$PMI(w_1; w_2) = \log \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

Intuition

- The more dependent *dog* and *cat* the closer $P(\text{dog}, \text{cat})$ is from $P(\text{dog})P(\text{cat})$ the smaller the PMI

Solution: Pointwise Mutual Information (PMI)

Idea: Instead of absolute co-occurrence statistics, use probability (relative) of co-occurrences

$$PMI(w_1; w_2) = \log \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

Intuition

- **The more dependent *dog* and *cat* the closer $P(\text{dog}, \text{cat})$ is from $P(\text{dog})P(\text{cat})$ the smaller the PMI**

$$PMI(w_1; w_2) = \log \frac{\frac{1}{V^2} \# \{w_1, w_2\}}{\frac{1}{V} \# \{w_1\} \frac{1}{V} \# \{w_2\}}$$

Pointwise Mutual Information (PMI)

	leash	walk	run	owner	pet	barked	the
dog	2.75	2.24	3.16	2.24	2.75	3.16	1.77
lion	0	2.75	3.16	0	3.85	0	2.06
car	0	0	3.85	2.75	0	0	2.75

Word embedding vectors are the row of the PMI matrix

- We usually take the Positive PMI (assigned to 0 when negative) + Smooth unobserve pairs (Laplace smoothing: add 1)
- Does not depend on size of the corpus (the PMI is **normalized**)
- Much less sensitive to change in frequent words (**log**)

Pointwise Mutual Information (PMI)

Limits:

- **Very large** matrix $O(V^2)$! Very large word vectors
- Hard to use large vectors in practice (i.e. 1M word vocabulary)
- **Cannot compare word vectors** estimated on 2 different corpora unless they have exactly the same vocabulary!

Idea: Build vectors with predefined size based on the PMI matrix

→ **Dimensionality Reduction Technique**

Singular Value Decomposition (SVD)

We can decompose the PMI Matrix with SVD

1. We build a symmetric definite matrix based on the PMI
2. We decompose it with the SVD

$$P = U_p \Sigma V_p^T$$

3. U is of size (V, d) gives us the representation of each word in a latent/embedding space

Properties of SVD:

- U is a orthonormal matrix
- U aggregates the highest variance of the original word embeddings

Limits of Dimensionality Reduction Approach

- Need to store a matrix of size $O(V^2)$
 - SVD is $O(V*d^2)$
- It is inefficient to build a very large matrix for reducing:
Can we do both simultaneously?

Solution: Prediction-Based Word Embedding Approaches

Prediction-Based Model

Idea:

- Learn directly **dense word vectors**
- Using the *distributional hypothesis*
- **Implicitly** , by parameterizing words as dense vectors
- and **learning to predict context** using this parametrization

Many word embedding methods use these ideas successfully

We present the *word2vec skip-gram* model (one of the most popular)

Word2Vec Skip-Gram Model

For each Sentence

1. Sample **a target word**
2. Predict **context words** defined as words in a fixed window from the target word

my dog is barking and chasing its tail

Word2Vec Skip-Gram Model

For each Sentence

1. Sample **a target word**
2. Predict **context words** defined as words in a fixed window from the target word



Word2Vec Skip-Gram Model

Given $d \in \mathbb{N}$, let $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$ two word representations (or word *embedding*) matrices. For each sequence (w_1, \dots, w_T) :

- Pick a *focus* word w , associated to the vector $\mathbf{w} \in \mathbb{R}^d$ (\mathbf{w} is the row associated to w in \mathbf{W})
- Pick a *context* word c , associated to the vector $\mathbf{c} \in \mathbb{R}^d$ (\mathbf{c} is the row associated to c in \mathbf{C})
- Maximize $\max_{\mathbf{W} \in \mathbb{R}^{(V,d)}, \mathbf{C} \in \mathbb{R}^{(V,d)}} \log p(c|w)$ (*maximum likelihood estimator*)



my dog is barking and chasing its tail

Word2Vec Skip-Gram Model

1. How to define $\log(p(c \mid w))$

2. How to optimize $\log(p(c \mid w))$

Word2Vec Skip-Gram Model

1. How to define $\log(p(c \mid w))$

2. How to optimize $\log(p(c \mid w))$

Intuition

- This is a classification problem
- The labels we want to predict are **the context words**
- Classification with a **very large number of labels** ($V \sim 100K$)

Ideas:

→ **Softmax**

→ Simplify the softmax with **Negative Sampling** for Efficiency

Word2Vec Skip-Gram Model

**Softmax of dot-products
context vs. words vectors:**

$$p(c | w) = \frac{e^{w.c}}{\sum_v e^{w.v}}$$

We compute the log-likelihood, **our objective function** , as:

$$\log p(c | w) = w \cdot c - \log \sum_v e^{w.v}$$

Limits: $O(V)$ to compute the loss (at every iteration)

→ **Negative Sampling**

Word2Vec Skip-Gram Model: Negative Sampling

Idea: Instead of computing the probability objective over the entire vocabulary (all the $V-1$ negative context words)

→ We sample K words that are not in the context of w $v \in N_K$
 $(K << V)$

Word2Vec Skip-Gram Model: Negative Sampling

Idea: Instead of computing the probability objective over the entire vocabulary (all the $V-1$ negative context words)

→ We sample K words that are not in the context of w $v \in N_K$ ($K \ll V$)

New objective function:

$$\sigma(\mathbf{w}, \mathbf{c}) + \frac{1}{K} \sum_{v \in N_K} \log (\sigma(-w, v)) \text{ with } \sigma(x, y) = \frac{1}{1 + e^{-x.y}}$$

Word2Vec Skip-Gram Model: Negative Sampling

Idea: Instead of computing the probability objective over the entire vocabulary (all the $V-1$ negative context words)

→ We sample ***K words that are not in the context of w*** $v \in N_K$
 $(K << V)$

New objective function:

$$\sigma(\mathbf{w}, \mathbf{c}) + \frac{1}{K} \sum_{v \in N_K} \log (\sigma(-\mathbf{w}, v)) \text{ with } \sigma(x, y) = \frac{1}{1 + e^{-x.y}}$$

→ **O(K)** to compute with K independent of V

Word2Vec Model: Optimization

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(v,d)}$ and $\mathbf{C} \in \mathbb{R}^{(v,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i, c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$

 ### Step 3: Parameter update with SGD

$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$

$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$

end

Word2Vec Model: Optimization

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i, c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$$

 ### Step 3: Parameter update with SGD

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

$$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

end

Word2Vec Model: Optimization

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i, c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$$

 ### Step 3: Parameter update with SGD

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

$$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

end

Word2Vec Model: Optimization

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i, c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$$

 ### Step 3. Parameter update with SGD

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

$$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

end

Word2Vec Model: Optimization

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i, c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$$

 ### Step 3: Parameter update with SGD

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

$$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$$

end

Word2Vec Model: Optimization

Loop: over the dataset E times (number of epochs)

Complexity: $O(d*K*T)$

- No Memory bottleneck
- Scale to Billion-tokens datasets

Algorithm 1 Skip-Gram Word2vec Training

Given a corpus C , made of a set of unique tokens V . Hyperparameters: number of negative samples K , a window size l , dimension of word vectors d , learning rate (α_t)

Initialize Randomly: $\mathbf{W} \in \mathbb{R}^{(V,d)}$ and $\mathbf{C} \in \mathbb{R}^{(V,d)}$

for step t in $0..T$ **do**

 ### Step 1: Sampling

 Sample $s = (w_1, \dots, w_n) \in C$ # a sequence in your corpus (e.g. sentence)

 Sample a pair $(i, j) \in [1, \dots, n]$ with $|i - j| \leq l$

 we note $w = w_i$, $c = w_j$ represented by vectors \mathbf{w} in \mathbf{W} and \mathbf{c} in \mathbf{C}

 Sample $N_K = \{v_1, \dots, v_K\} \subset V$ represented by $\{\mathbf{v}_1, \dots, \mathbf{v}_K\}$ in \mathbf{C} # Negative samples

 ### Step 2: Compute loss

$l(\mathbf{W}, \mathbf{C}) = -\sigma(\mathbf{w}, \mathbf{c}) - \frac{1}{K} \sum_{v \in N_K} \log \sigma(-\mathbf{w}, \mathbf{v})$

 ### Step 3: Parameter update with SGD

$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$

$\mathbf{C}_t = \mathbf{C}_{t-1} - \alpha_t \cdot \nabla l(\mathbf{W}_{t-1}, \mathbf{C}_{t-1})$

end

Word2Vec Skip-Gram Model & the PMI

(Levy & Goldberg 2014) showed that

- Estimating the embedding matrix with Skip-Gram and Negative Sampling (SGNS)...
- ...is equivalent to computing the shifted-PMI matrix

Word2Vec

- Not popular in practice anymore :’(
- Worked very well with Deep Learning architecture (e.g. LSTM models) to model specific tasks (e.g. NER)
- Recently “beaten” by contextualized approaches (BERT)

Extensions

- Lots of variant of the Skip-Gram exists (CBOW, Glove...)
- Multilingual setting: build shared representations across languages (fasttext)

Limits

- Doesn’t model morphology
- **Fixed Vocabulary** : What if we add new tokens in the vocabulary?
- **Polysemy** : Each token has a unique representation (e.g. cherry)

Evaluation of Word Embeddings

How to evaluate the quality of word embeddings?

Extrinsic Evaluation

- Use them in a task-specific model and measure the performance on your task (cf. lecture 5 & 6)

Intrinsic Evaluation

→ Idea : “*similar*” words *should have similar vectors*

What do we mean by “similar” words?

- Morphologically similar: e.g. *computer, computers*
- Syntactically similar: e.g. determiners
- Semantically similar: e.g. *animal, cat*

Intrinsic Evaluation of Word Embeddings

How to evaluate the quality of word embeddings?

Qualitative Evaluation

- Visualize word embedding space
- Case by case: look at nearest neighbors of given words

Quantitative Evaluation

- Is Word embedding similarity related with human judgment ?

Intrinsic Evaluation of Word Embeddings

Visualization

Word Vectors are high dimensions (usually ~100)

- Project the word embedding vectors using PCA or T-SNE
- Visualize in 2D or 3D
- Analyse the clusters

Intrinsic Evaluation of Word Embeddings

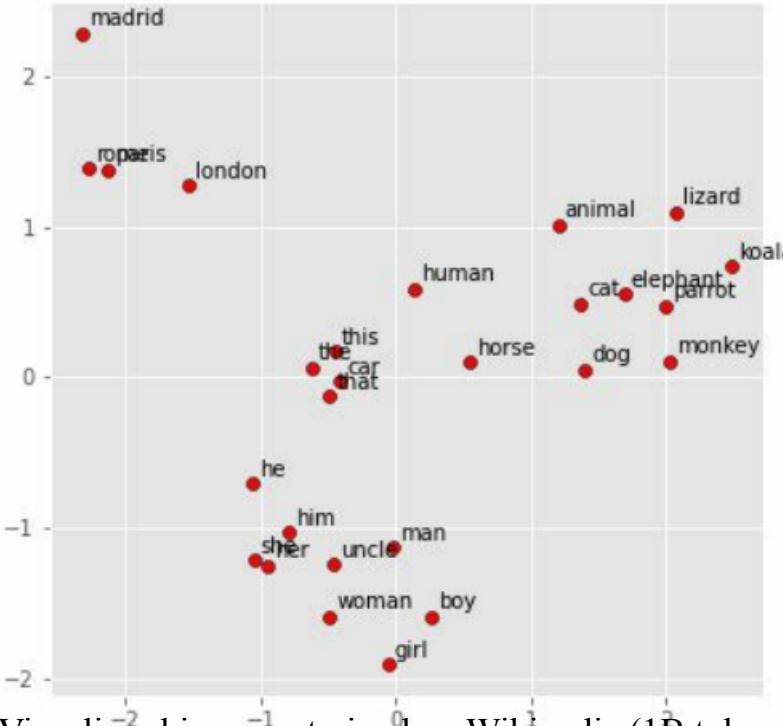


Figure: Visualize skip-gram trained on Wikipedia (1B tokens) (fasttext.cc) vectors with PCA

Intrinsic Evaluation of Word Embeddings

How to measure similarity in the word embedding space?

- Cosine Similarity

$$sim(w_i, w_j) = \cos(x_{w_i}, x_{w_j}) = \frac{x_{w_i}^T x_{w_j}}{\|x_{w_i}\| \|x_{w_j}\|}$$

- L2 Distance

$$sim(w_i, w_j) = L_2(x_{w_i}, x_{w_j}) = \|x_{w_i} - x_{w_j}\|$$

Intrinsic Evaluation of Word Embeddings

Nearest-Neighbor with the cosine similarity (skip-gram trained on Wikipedia (1B tokens))

moon	score	talking	score	blue	score
mars	0.615	discussing	0.663	red	0.704
moons	0.611	telling	0.657	yellow	0.677
lunar	0.602	joking	0.632	purple	0.676
sun	0.602	thinking	0.627	green	0.655
venus	0.583	talked	0.624	pink	0.612

Intrinsic Evaluation of Word Embeddings

We can **compare the similarity between words in the embedding space with human judgment**

1. **Collect Human Judgment** (or download dataset e.g. WordSim353) on a list of pairs of words
2. **Compute similarity** of the **word vectors** of those pairs
3. **Measure correlation** between both

Word 1	Word 2	Word2vec Cosine Similarity	Human Judgment
tiger	tiger	1.0	10
dollar	buck	0.3065	9.22
dollar	profit	0.3420	7.38
smart	stupid	0.4128	5.81

Application of Word Embeddings

- Downstream Tasks (Lecture 5 and 6)
- **Word Sense Induction**
- **Semantic analysis** (semantic shift in time, across communities...)

Representing Documents With Vectors

Representing Documents into Vectors

Similarly to what we saw for word-level representation we can **represent documents into vectors**

1. Using word vectors
2. Count-Based Representations
3. Generative Probabilistic Graphical Model (e.g. LDA seen in the *lab*)
4. Using language models

Representation of documents based on words

Based on word vectors representing sentence/document with vector can be done in a straightforward way:

- Given sequence of word represented by x_1, \dots, x_n , define $f: \rightarrow \mathbf{R}$

$$[x_1, \dots, x_n] \rightarrow f(x_1, \dots, x_n)$$

For instance:

$$[x_1, \dots, x_n] \rightarrow \frac{1}{n} \sum_i x_i$$

Count-Based Representation of Documents

Given a Corpus made of novels of Shakespeare (Macbeth, Hamlet...), each document is a novel here:

1. Get the vocabulary of the Corpus
2. Compute the **Count-Based Matrix at the document-level**

Count-Based Representation of Documents

Given a Corpus made of novels of Shakespeare (Macbeth, Hamlet...), each document is a novel here:

1. Get the vocabulary of the Corpus
2. Compute the **Count-Based Matrix at the document-level**

Build the *term-frequency* matrix

$$tf_{t,d} = |\{t \in d\}|$$

Count-Based Representation of Documents

Given a Corpus made of novels of Shakespeare (Macbeth, Hamlet...), each document is a novel here:

1. Get the vocabulary of the Corpus
2. Compute the **Count-Based Matrix at the document-level**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Count-Based Representation of Documents

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

→ We get a vector representation for each document of the corpus

NB: such a model is called a *bag-of-word model* because the ordering of the words in each document does not matter

Count-Based Representation of Documents

Limits: High sensitivity to frequent words OR to very infrequent words

How to improve?

- A word that is in all documents of the corpus (e.g. “the”) is not informative at all for the document representation, still it impacts the document vector
- A word that is in only 1 document is likely to be very informative of the document

Solution:

- Weight the count with
- Inverse Document Frequency

Count-Based Representation of Documents

Weighting the importance of each term with the *document frequency*

Definition: Given N the total number of documents , a term t (token),

$$idf_{t,C} = \log \frac{|C|}{|\{d \in C, st t \in d\}|}$$

NB: Compute the *log* to smooth the impact of words that are in only a few documents

TF-IDF Representation of Documents

Matrix becomes: $tf \times idf(t, d, C)$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

TF-IDF Representation of Documents

We can then apply dimension reduction technique to get dense vectors

- E.g. we can apply SVD: **Latent Semantic Analysis**

Recurrent Neural Network

Vanilla Recurrent Neural Network

We would like to model sequences (e.g. words) (X_1, \dots, X_T) in $\mathbb{R}^{d,T}$

We can introduce **a recurrence relation** into our MLP to model it:

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + b_i), \forall i \in [|1, L-1|]$$

with $h_{1,t} = X_t$ and $\hat{Y}_t = dnn(X_t) = h_{L,t} \forall t \in [|1, T-1|]$

Recurrent Neural Network

Illustration of a 1-layer Recurrent Neural Network

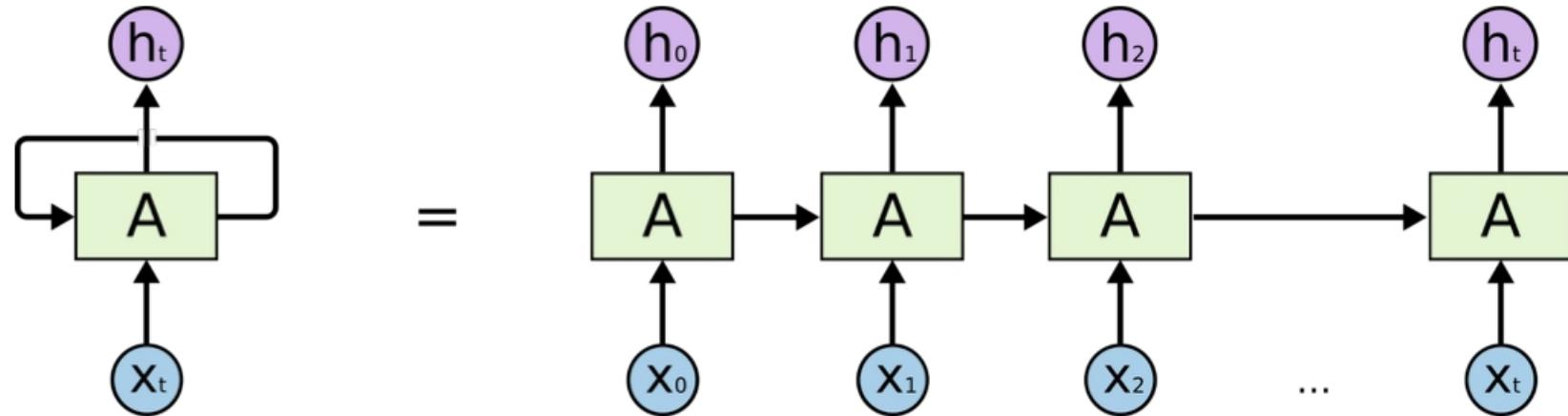


Figure from [colah](#)

Recurrent Neural Network

Illustration of a 1-layer Recurrent Neural Network

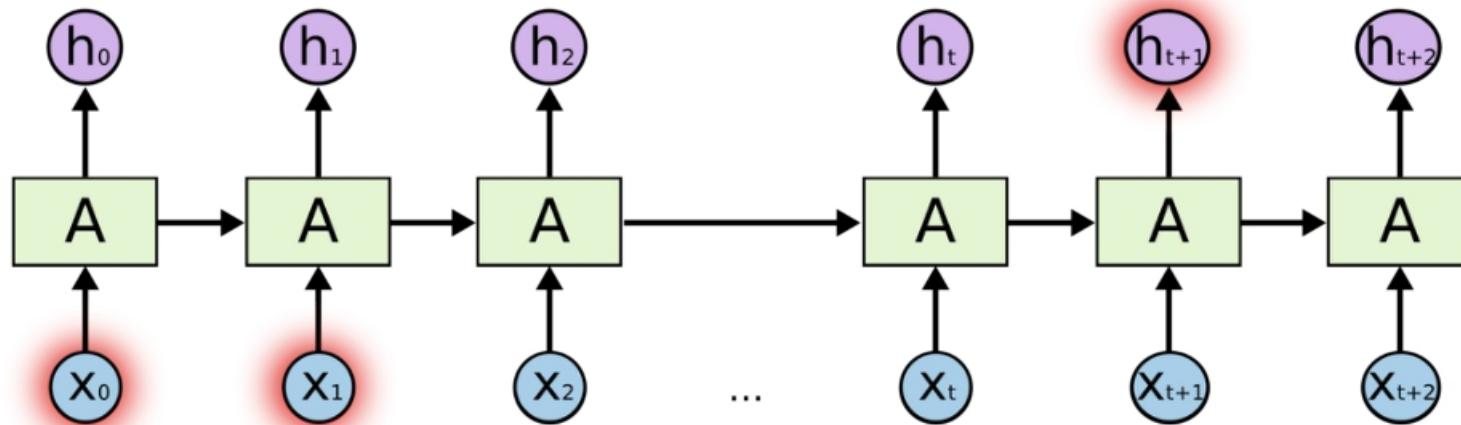


Figure from [colah](#)

Training Recurrent Neural Network

Recurrent Neural Network are trained with an extension of the Backpropagation algorithm

→ Backpropagation Through Time (BPTT)

BPTT follows exactly the same ideas as backpropagation

- SGD
- Chain Rule starting from the last layer and the last hidden state
- **With extra derivative dependencies between state t and $t+1$**

Limits of Recurrent Neural Networks

Vanilla Recurrent Neural Network have trouble to capture long-term dependencies

Idea:

- Encode **explicitly in a vector a “memory” in the recurrent architecture**
- Control what is memorized and forgotten
- Train all those parameters **end-to-end**

Attention Mechanism

Motivation for Attention Mechanisms

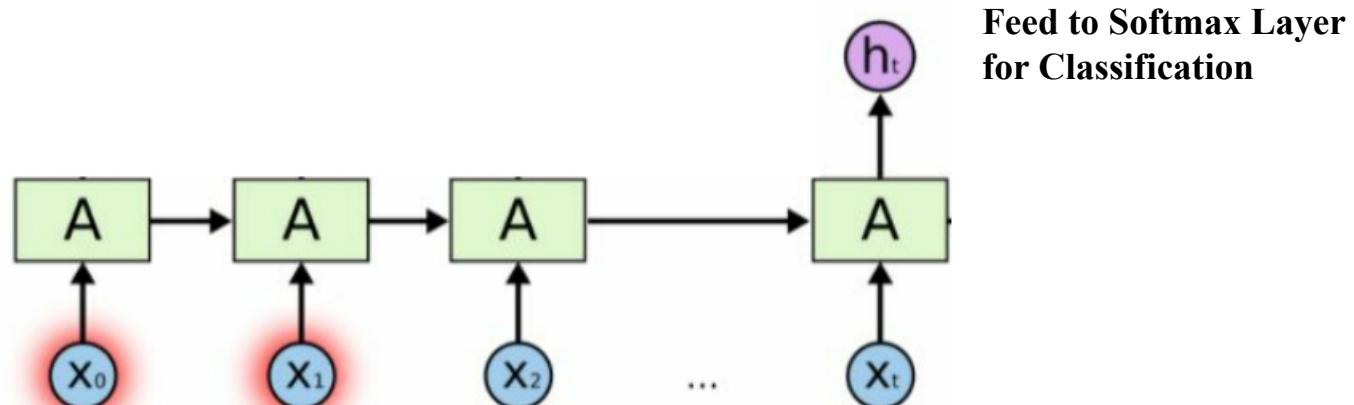
- The Deep Learning Architecture that we have seen so far are **hard to interpret (black-box)**
- Recurrent Network provide a fixed vector encoding of a sequence at each step

→ **Attention Mechanisms**

Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment analysis)

Solution 1: Use a LSTM model → Problem (not interpretable)

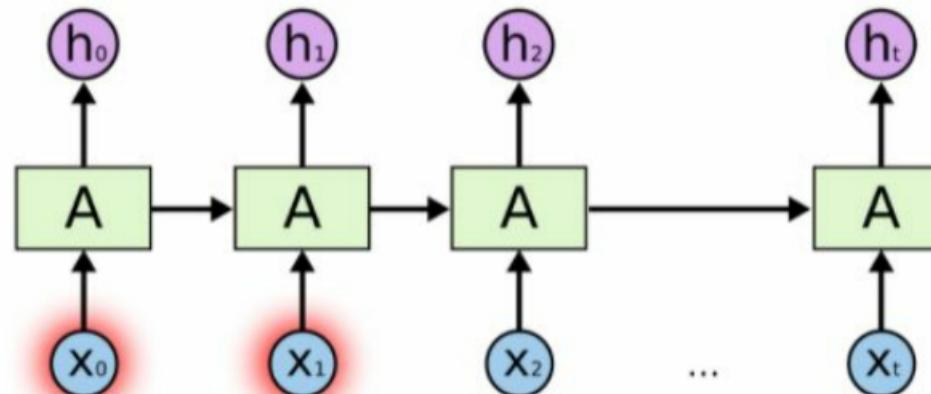


Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment)

Solution 2: Integrate an Attention Mechanism to interpret what input impacts the prediction

→ Learn a ponderation/weighting of the hidden states h_t



Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment)

How to learn this weighting?

1. Define a specific type of layer to learn the ponderation
2. Train this layer end-to-end with all the other parameters of the model

Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment)

How to learn this weighting?

Given (h_1, \dots, h_T) hidden representations of (x_1, \dots, x_T) (e.g. output of a LSTM Layer).

$$q_i = \tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}$$

$$s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [1, T]} s_t = 1$$

$$\tilde{h}_T = \sum_{t \in [1, T]} s_t \cdot h_t$$

Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment)

How to learn this weighting?

Given (h_1, \dots, h_T) hidden representations of (x_1, \dots, x_T) (e.g. output of a LSTM Layer).

$$q_i = \tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}$$

$$s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [1, T]} s_t = 1$$

$$\tilde{h}_T = \sum_{t \in [1, T]} s_t \cdot h_t$$

Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment)

How to learn this weighting?

Given (h_1, \dots, h_T) hidden representations of (x_1, \dots, x_T) (e.g. output of a LSTM Layer).

$$q_i = \tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}$$

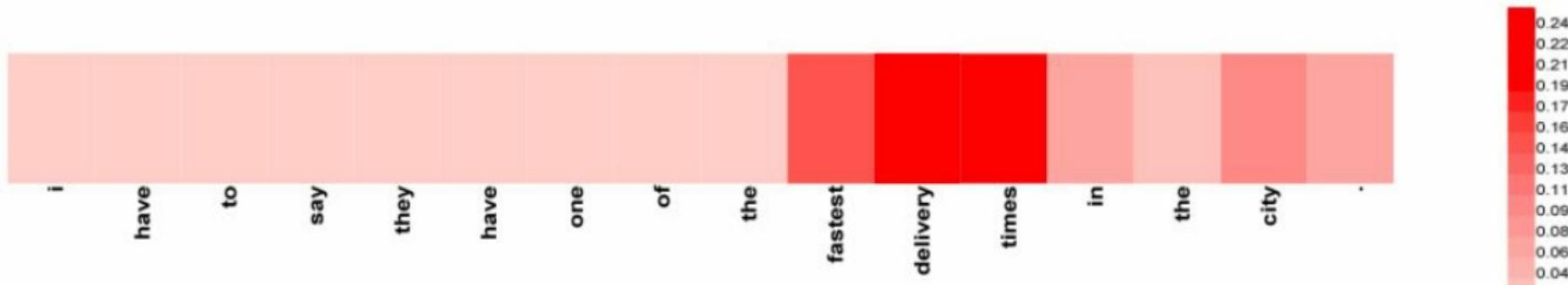
$$s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [1, T]} s_t = 1$$

$$\tilde{h}_T = \sum_{t \in [1, T]} s_t \cdot h_t$$

Attention Mechanism for Sequence Classification

We want to classify (X_0, X_t) sequences (e.g. sentiment classification)

After we trained the model, **Attention scores** can be used to interpret the model behavior and what input vector impacted the decision



(Wang et. al 2016)

Attention Mechanism for Sequence Classification

Many variant of Attention Mechanisms (in combination with LSTM layers) have been designed

Design Choices

- How to define the *query vectors* ?
- How to define the *scoring function* ?

Many variants exists but the principles are the same.

The Transformer Architecture

Attention might be all we need

Do we really need recurrent layers?

RNN models (such as vanilla RNN, LSTM...) were designed to model sequential data

Still, for most tasks, we **need both left and right context** (e.g. **sequence classification, sequence labelling..**)

Why not modelling sequences in a bi-directional way directly
→ **Using Self-Attention Mechanism**

Self-Attention Layers

Given a sequence of input vectors $(x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $(h_{0,1}, \dots, h_{0,T})$).

Objective:

- Build a representation of the input vectors based on the **surrounding vectors** (both right-and left-context)

Idea:

- **No need of recurrent cells**
- **Self-Attention**

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector \tilde{h}_t
- Using the best ponderation of the information encoded in (v_1, \dots, v_T)
- This ponderation being computed by finding the key vectors in (k_1, \dots, k_T) that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector \tilde{h}_t
- Using the best ponderation of the information encoded in (v_1, \dots, v_T)
- This ponderation being computed by finding the key vectors in (k_1, \dots, k_T) that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector \tilde{h}_t
- Using the best ponderation of the information encoded in (v_1, \dots, v_T)
- This ponderation being computed by finding the key vectors in (k_1, \dots, k_T) that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector \tilde{h}_t
- Using the best ponderation of the information encoded in (v_1, \dots, v_T)
- This ponderation being computed by finding the key vectors in (k_1, \dots, k_T) that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$q_t = W_Q h_t, \forall t \in [|1, T|] \text{ with } W_Q \in \mathbb{R}^{\delta_q \times \delta}$$

$$k_t = W_K h_t, \forall t \in [|1, T|] \text{ with } W_K \in \mathbb{R}^{\delta_k \times \delta}$$

$$v_t = W_V h_t, \forall t \in [|1, T|] \text{ with } W_V \in \mathbb{R}^{\delta_v \times \delta}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$q_t = W_Q h_t, \forall t \in [|1, T|] \text{ with } W_Q \in \mathbb{R}^{\delta_q \times \delta}$$

$$k_t = W_K h_t, \forall t \in [|1, T|] \text{ with } W_K \in \mathbb{R}^{\delta_k \times \delta}$$

$$v_t = W_V h_t, \forall t \in [|1, T|] \text{ with } W_V \in \mathbb{R}^{\delta_v \times \delta}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$q_t = W_Q h_t, \forall t \in [|1, T|] \text{ with } W_Q \in \mathbb{R}^{\delta_q \times \delta}$$

$$k_t = W_K h_t, \forall t \in [|1, T|] \text{ with } W_K \in \mathbb{R}^{\delta_k \times \delta}$$

$$v_t = W_V h_t, \forall t \in [|1, T|] \text{ with } W_V \in \mathbb{R}^{\delta_v \times \delta}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$\tilde{H} = \text{softmax}\left(\frac{Q K^T}{\sqrt{\delta_K}}\right)V$$

i.e. $\tilde{h}_t = \text{softmax}\left(\frac{q_t K^T}{\sqrt{\delta_K}}\right).V = \sum_{t'} s_{t'} v_{t'}$ with $s_{t'} = \frac{e^{q_t k_{t'}}}{\sum_t e^{q_t k_t}}$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$\tilde{H} = \text{softmax}\left(\frac{Q K^T}{\sqrt{\delta_K}}\right)V$$

i.e. $\tilde{h}_t = \text{softmax}\left(\frac{q_t K^T}{\sqrt{\delta_K}}\right).V = \sum_{t'} s_{t'} v_{t'} \text{ with } s_{t'} = \frac{e^{q_t' k_t}}{\sum_t e^{q_t' k_t}}$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, \dots, x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, \dots, h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, \dots, h_T)$.

The *query* $Q = (q_1, \dots, q_T)$, the *key* $K = (k_1, \dots, k_T)$ and the *value* $V = (v_1, \dots, v_T)$ vectors.

$$\tilde{H} = \text{softmax}\left(\frac{Q K^T}{\sqrt{\delta_K}}\right)V$$

i.e. $\tilde{h}_t = \text{softmax}\left(\frac{q_t K^T}{\sqrt{\delta_K}}\right).V = \sum_{t'} s_{t'} v_{t'}$ with $s_{t'} = \frac{e^{q_t k_{t'}}}{\sum_t e^{q_t k_t}}$

Self-Attention Layers

Attention is a building block.

Think of it as a "soft" **kv** dictionary lookup:

1. Attention weights $a_{1:N}$ are query-key similarities:

$$\hat{a}_i = \mathbf{q} \cdot \mathbf{k}_i$$

Normalized via softmax: $a_i = e^{\hat{a}_i} / \sum_j e^{\hat{a}_j}$

2. Output **z** is attention-weighted average of values $v_{1:N}$:

$$\mathbf{z} = \sum_i \hat{a}_i \mathbf{v}_i = \hat{\mathbf{a}} \cdot \mathbf{v}$$

3. Usually, **k** and **v** are derived from the same input **x**:

$$\mathbf{k} = \mathbf{W}_k \cdot \mathbf{x}$$

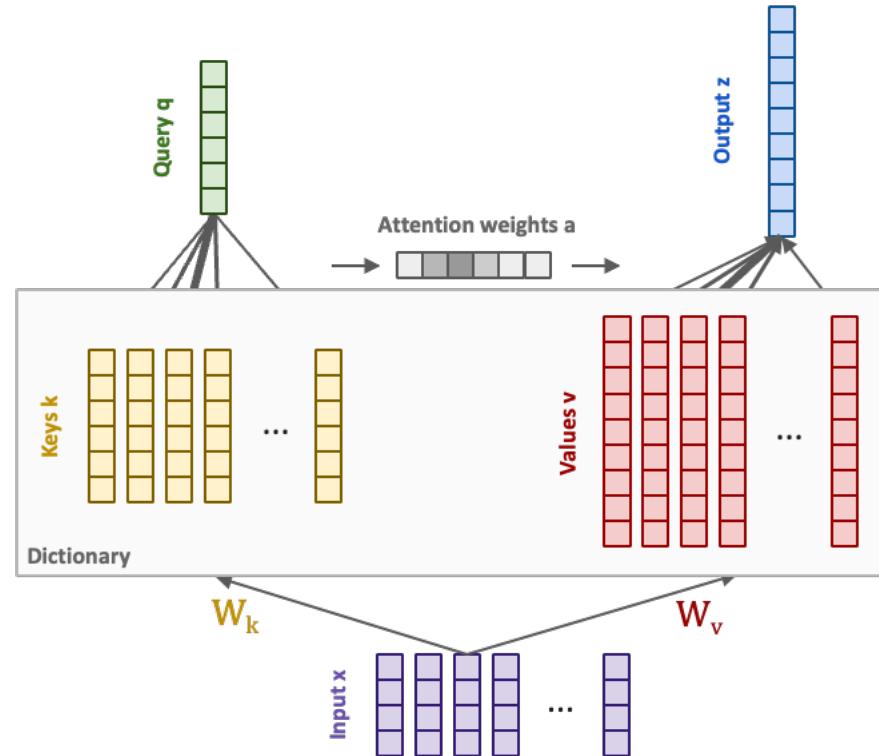
$$\mathbf{v} = \mathbf{W}_v \cdot \mathbf{x}$$

The query **q** can come from a separate input **y**:

$$\mathbf{q} = \mathbf{W}_q \cdot \mathbf{y}$$

Or from the same input **x**! Then we call it "self attention":

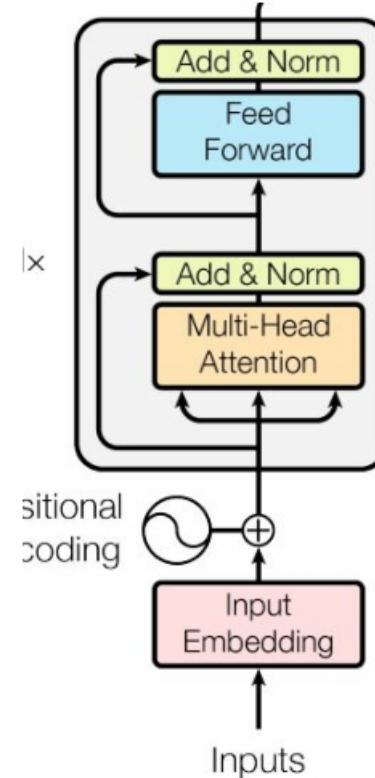
$$\mathbf{q} = \mathbf{W}_q \cdot \mathbf{x}$$



The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]
- With Skip-Layer and Normalization Layers in between
- Encoding the position with positional vector



Multi-head Self-Attention Layers

1. We usually use **many queries** $q_{1:M}$, not just one.

Stacking them leads to the Attention matrix $A_{1:N,1:M}$

and subsequently to many outputs:

$$z_{1:M} = \text{Attn}(q_{1:M}, x) = [\text{Attn}(q_1, x) | \text{Attn}(q_2, x) | \dots |$$

2. $\text{Attn}(q_M, x)$ We usually use "**multi-head**" attention. This means

the

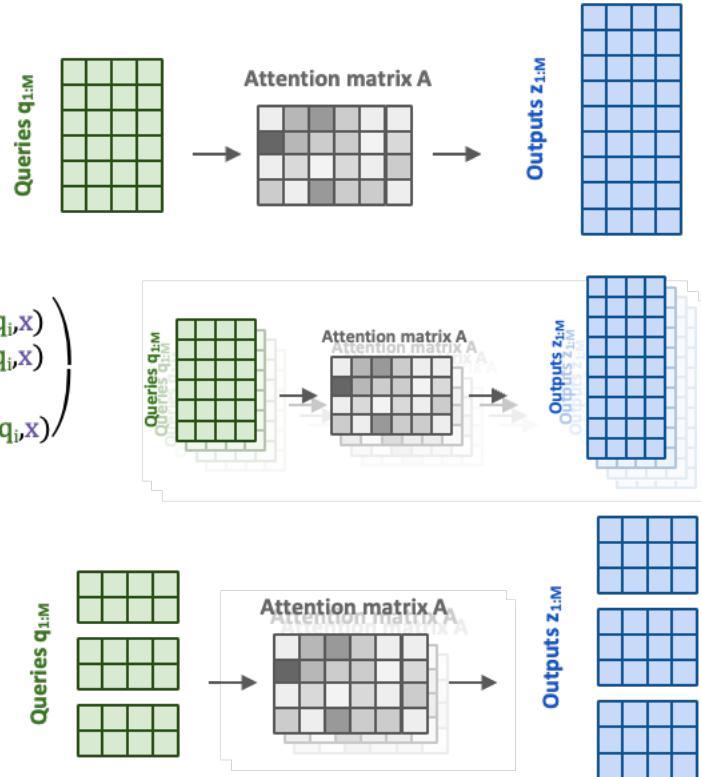
operation is repeated K times and the results are

concatenated along the feature dimension. Ws differ.

3. The most commonly seen formulation:

$$z = \text{softmax}(QK'/\sqrt{d_{\text{key}}})V$$

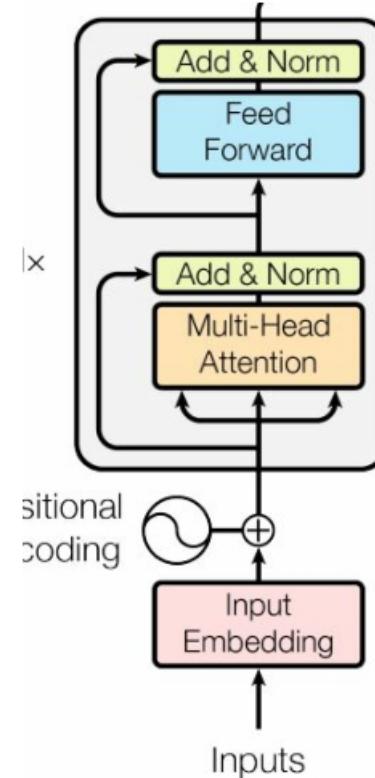
Note that the complexity is $O(N^2)$



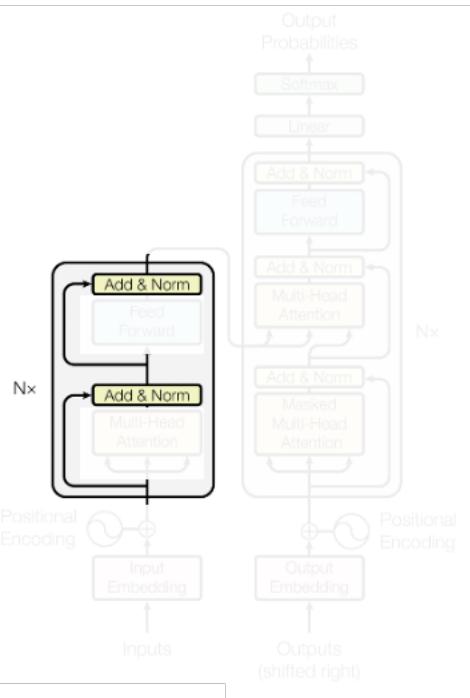
The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]
- With Skip-Layer and Normalization Layers in between
- Encoding the position with positional vector



Residual Connections



Residual/skip connections

Each module's output has the exact same shape as its input.

Following ResNets, the module computes a "residual" instead of a new value:

$$z_i = \text{Module}(x_i) + x_i$$

This was shown to dramatically improve trainability.

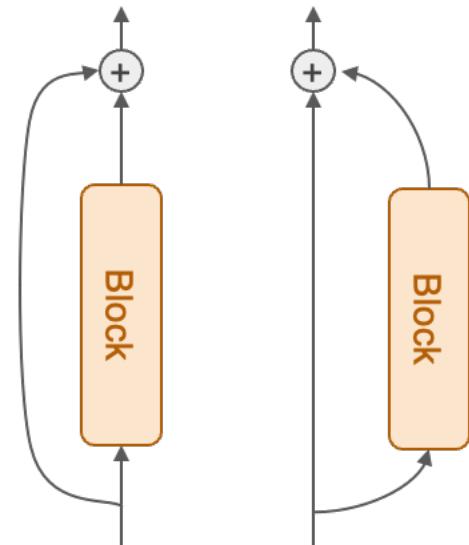
LayerNorm

Normalization also dramatically improves trainability.

There's **post-norm** (original) and **pre-norm** (modern)

$$\text{Module}(x_i) + x_i$$

"Skip connection" == "Residual block"

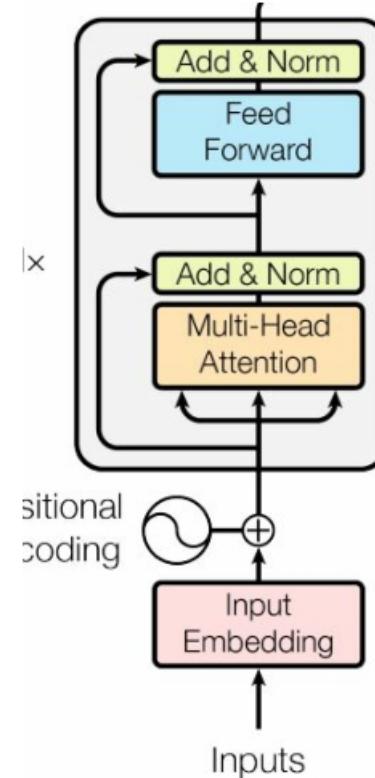


$$z_i = \text{Module}(\text{LN}(x_i)) + x_i$$

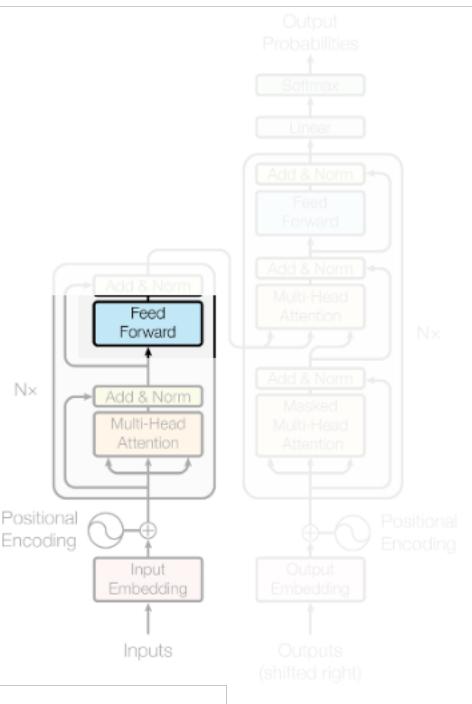
The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]
- With Skip-Layer and Normalization Layers in between
- Encoding the position with positional vector



MLP



Point-wise MLP

A simple MLP applied to each token individually:

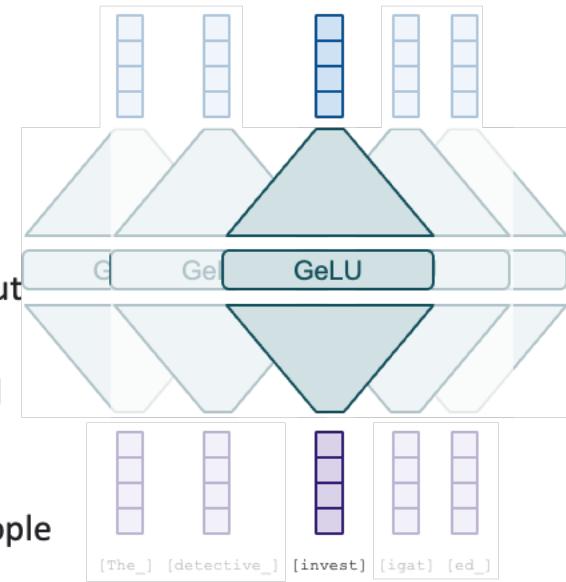
$$z_i = W_2 \text{GeLU}(W_1 x + b_1) + b_2$$

Think of it as each token pondering for itself about what it has observed previously.

There's some weak evidence this is where "world knowledge" is stored, too.

It contains the bulk of the parameters. When people make giant models and sparse/moe, this is what becomes giant.

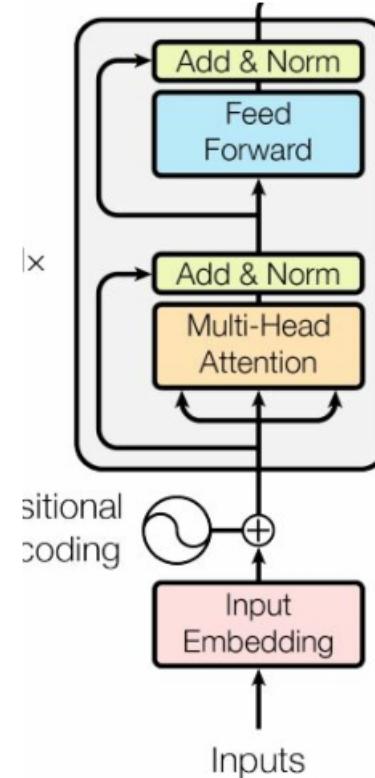
Some people like to call it 1×1 convolution.



The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]
- With Skip-Layer and Normalization Layers in between
- Encoding the position with positional vector



Positional Embedding Vector

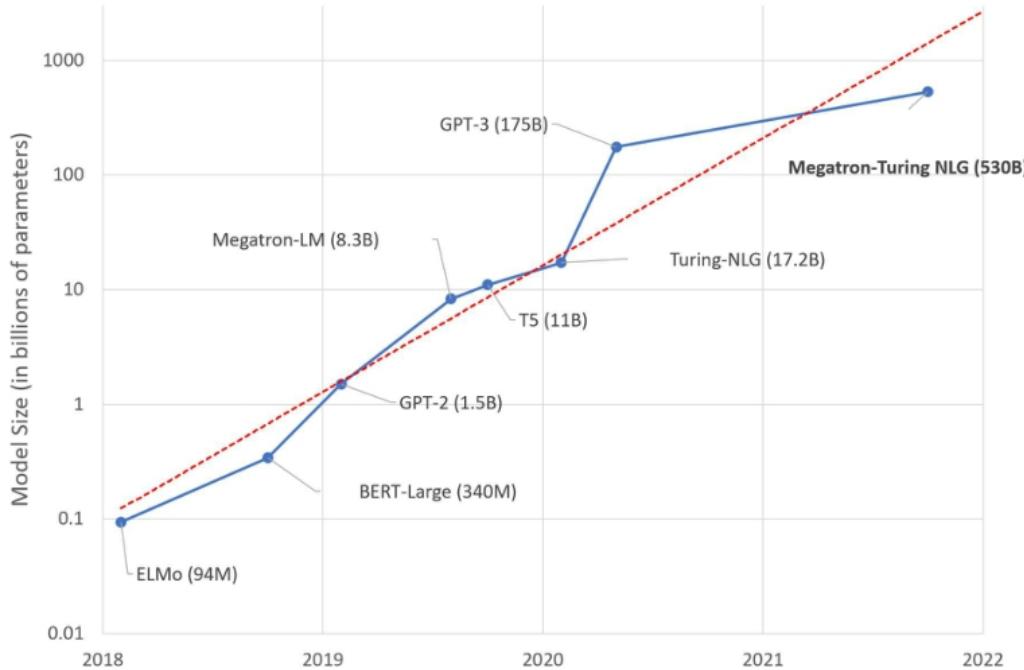
- **Limitation:** self attention does not take position into account!
- Indeed, shuffling the input gives the same results
- **Solution:** add position encodings.
- Replace the matrix \mathbf{W} by $\mathbf{W} + \mathbf{E}$, where $\mathbf{E} \in \mathbb{R}^{d \times T}$
- \mathbf{E} can be learned, or defined using sin and cos:

$$e_{2i,j} = \sin\left(\frac{j}{10000^{2i/d}}\right)$$
$$e_{2i+1,j} = \cos\left(\frac{j}{10000^{2i/d}}\right)$$

Scaling Laws Intuition

- The larger the dimension of the weight matrices
- The larger the number of parameters in the model
- The more “expressive” is the model
- The better it will generalize

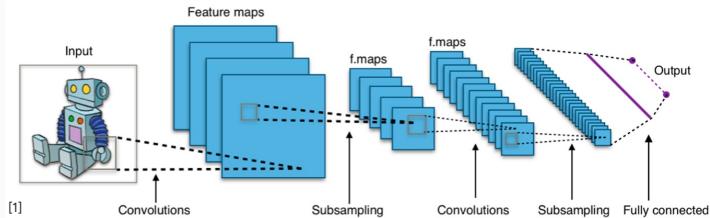
Typical Architecture Sizes



The classic landscape:
One architecture
per "community"

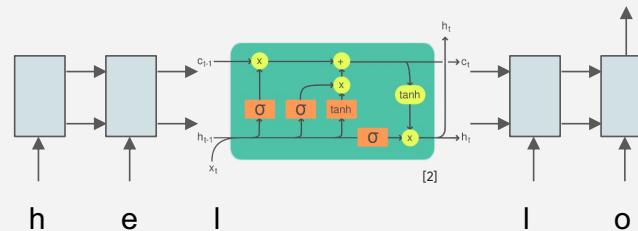
Computer Vision

Convolutional NNs (+ResNets)



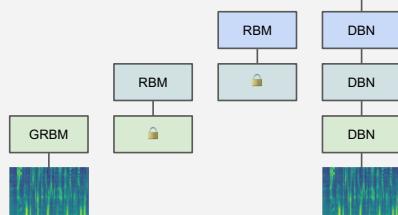
Natural Lang. Proc.

Recurrent NNs (+LSTMs)



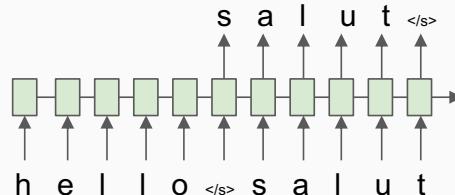
Speech

Deep Belief Nets (+non-DL)



Translation

Seq2Seq



[1] CNN image CC-BY-SA by Aphex34 for Wikipedia https://commons.wikimedia.org/wiki/File:Typical_cnn.png

[2] RNN image CC-BY-SA by GChe for Wikipedia https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg

RL

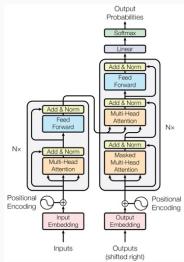
BC/GAIL

Algorithm 1 Generative adversarial imitation learning

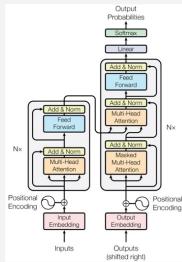
- 1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
- 2: **for** $i = 0, 1, 2, \dots$ **do**
- 3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
- 4: Update the discriminator parameters from w_i to w_{i+1} with the gradient
$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s, a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s, a))] \quad (17)$$
- 5: Take a policy step from θ_i to θ_{i+1} , using the TRPO rule with cost function $\log(D_{w_{i+1}}(s, a))$. Specifically, take a KL-constrained natural gradient step with
$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(a|s)Q(s, a)] - \lambda \nabla_\theta H(\pi_\theta), \quad (18)$$
where $Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s, a)) | s_0 = \bar{s}, a_0 = \bar{a}]$
- 6: **end for**

The Transformer's takeover: One community at a time

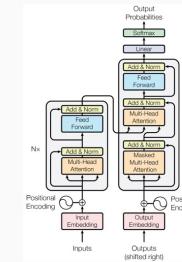
Computer Vision



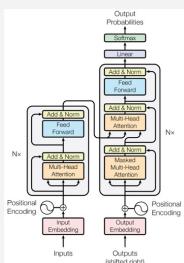
Natural Lang. Proc.



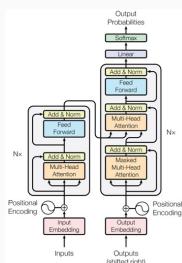
Reinf. Learning



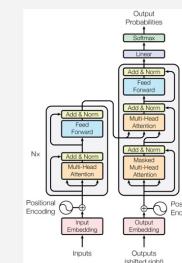
Speech



Translation



Graphs/Science



Transformer image source: "Attention Is All You Need" paper

Lecture Summary

Deep Learning is a powerful and general modelling approach

- **Designing Architectures** , i.e. composition of linear transformation and non-linear transformation (possibly including recurrences)
- All those transformations **should be differentiable**
- All the parameters of the model **are trained with backpropagation**
- **Toward a specific task** s.t. regression or classification
- All the hyperparameters are chosen based on **best-practices** or empirical research

Bibliography and Acknowledgment

- ❖ Distributional Semantics, Evert 2015
- ❖ Foundations of Distributional Semantics, Evert & Lenci 2009
- ❖ Stanford Information-Retrieval Book
- ❖ Miller et. al 1985: WordNet
- ❖ Mikolov et. al 2013: Word2vec
- ❖ Grave & Joulin, ENSAE 2018-2019

All these class have been taken from <https://nlp-ensae.github.io/materials/> and is taken from Benjamin Muller