

# Word Vectors

Word2Vec, BERT

---

CHIEN CHIN CHEN

# When he was an Intern ...

In 2012, Thomas Mikolov, an intern at Microsoft, found a way to encode the meaning of words in a modest number of vector dimensions.

In 2013 (at Google), Mikolov and his teammates released **Word2vec**.

**Tomáš Mikolov**  
  
Tomas Mikolov at NEURON 2018

<b>Born</b>	1982 Šumperk
<b>Residence</b>	Mountain View, California, United States
<b>Citizenship</b>	Czech
<b>Alma mater</b>	Brno University of Technology
<b>Scientific career</b>	
<b>Fields</b>	Computer Science
<b>Institutions</b>	Google & Facebook

Efficient estimation of word representations in vector space

T Mikolov, K Chen, G Corrado, J Dean  
arXiv preprint arXiv:1301.3781

11728

2013

# Word2Vec

---

He trained a **neural network** to predict word occurrences near a target word.

- **The weights of the network** help represent words!!

Basically, it is a **language modeling problem!!**

It is **unsupervised learning** that the prediction is based on a super large corpus of unlabeled text.

- Or ... supervised ... but the labeling has been done already!!

# Two Approaches

---

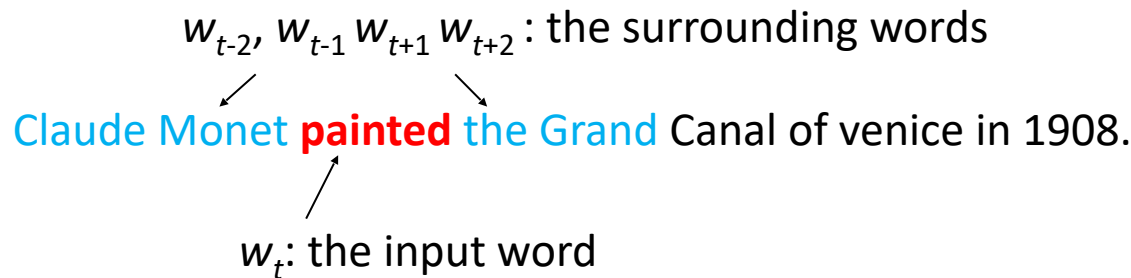
Two approaches to train the prediction model:

- **Skip-gram**: predicts the context of words from a word of interest.
- **Continuous bag-of-words (CBOW)**: predicts the target word from the nearby words.

# Skip-gram Approach (1/9)

---

In skip-gram, the model is trying to predict the surrounding window of words based on an input word.

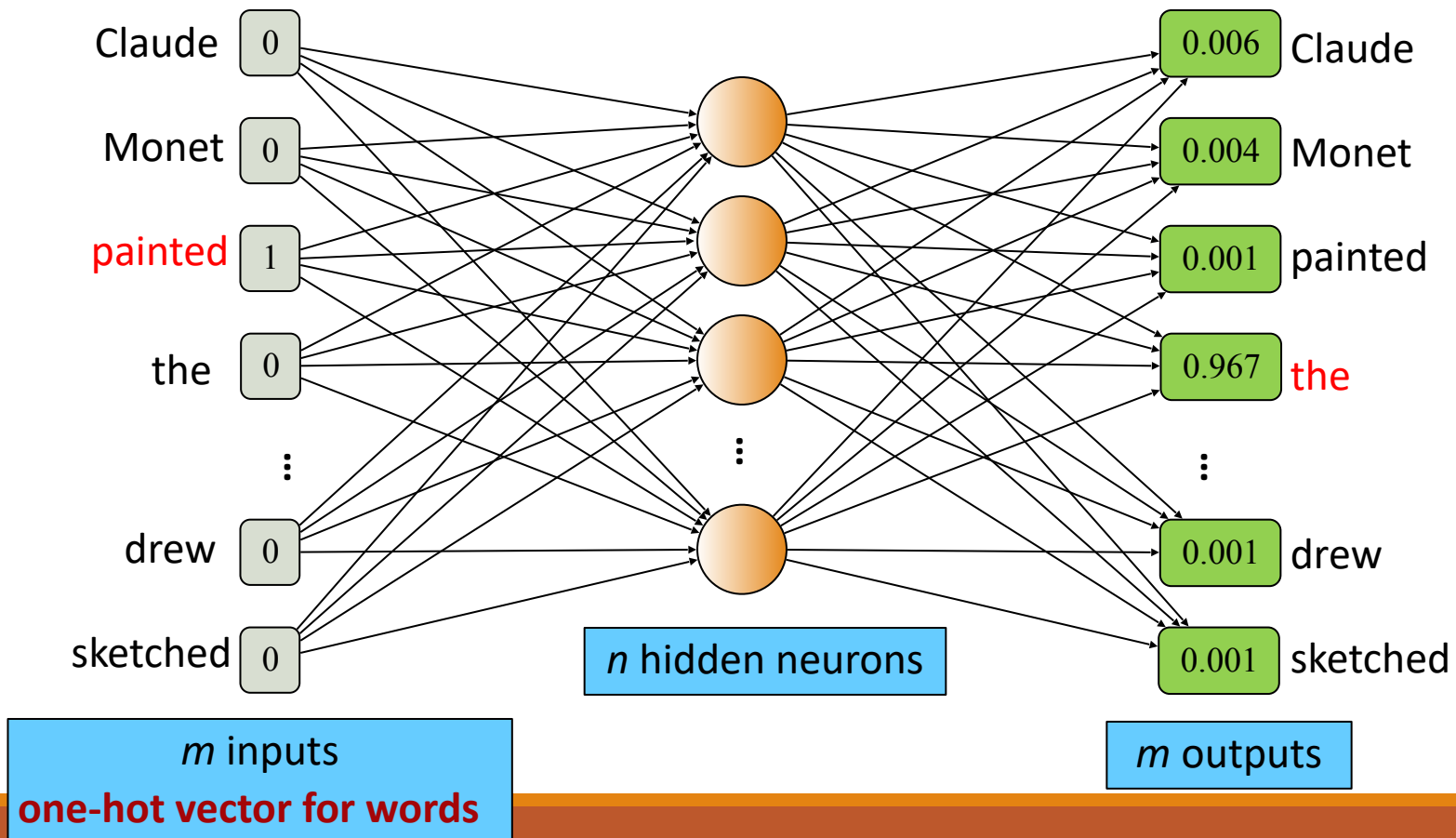


The neural network consists of two layers of weights.

- The **hidden layer** consists of  $n$  neurons.
  - $n$  is the number of vector dimension used to represent a word.
- The **input** and **output layers** contain  $m$  neurons.
  - $m$  is the number of words in the vocabulary.
  - Note that the output layer uses **softmax** as an activation function.

# Skip-gram Approach (2/9)

The prediction (*testing*) phase of the model:



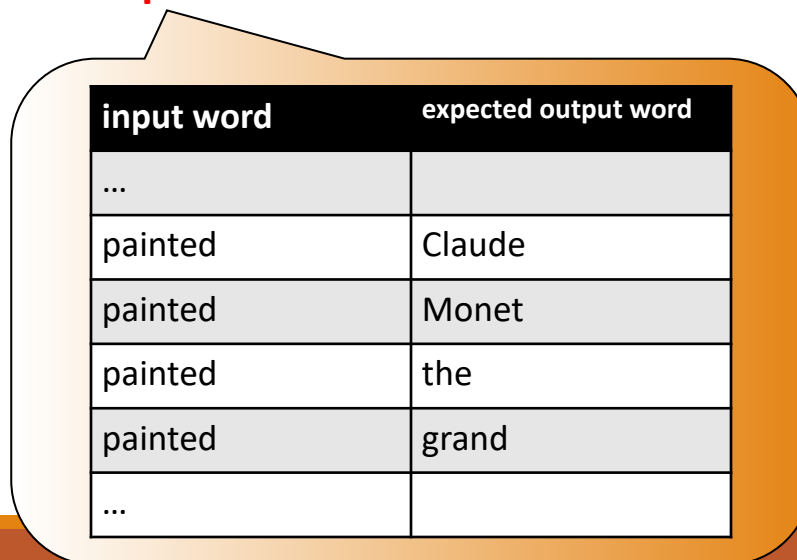
# Skip-gram Approach (3/9)

---

How to train the prediction model?

- You need a text corpus.
- Suppose the window size of the context is 2.
- Process every token in the corpus to produce a set of **training word pairs**:

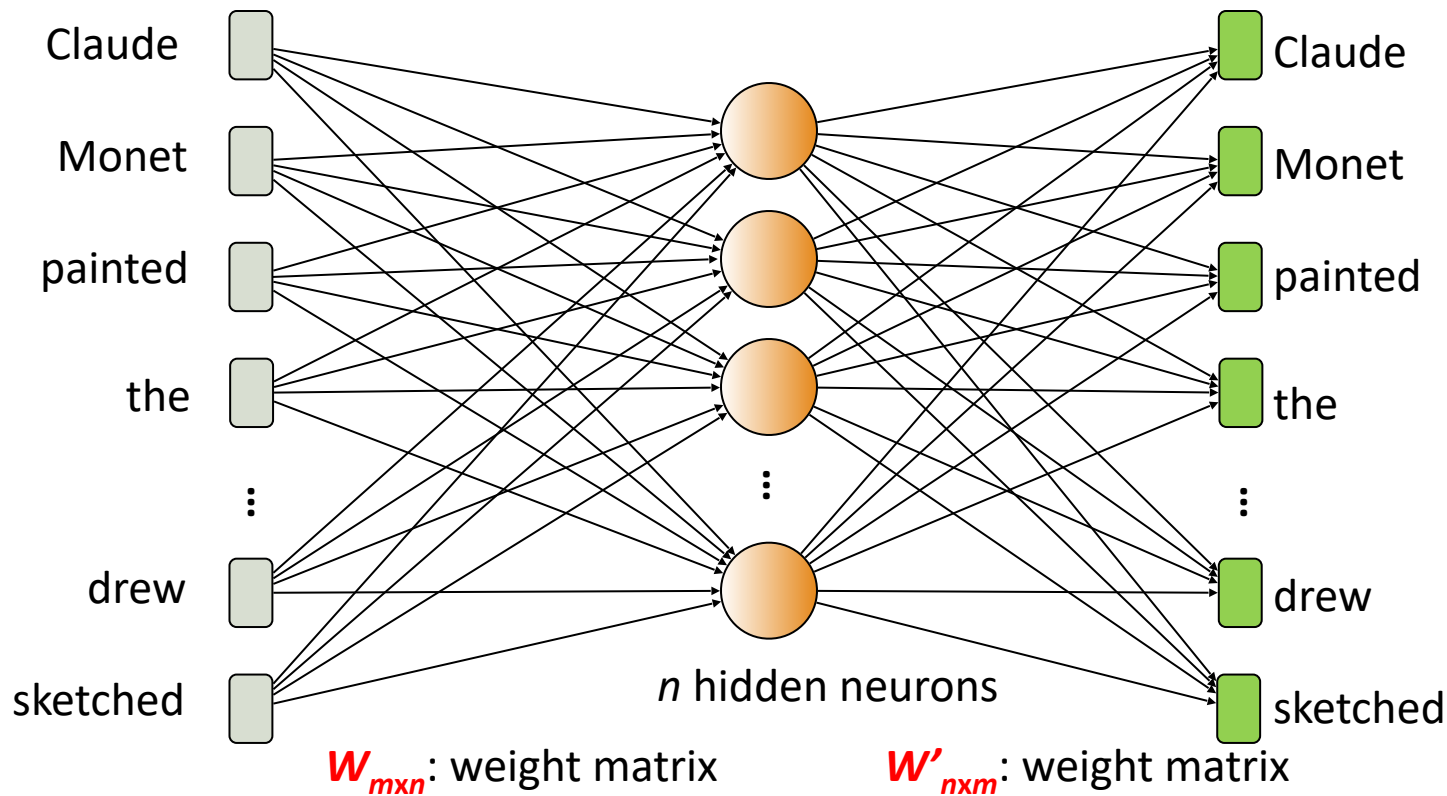
Claude Monet painted the Grand Canal of venice in 1908.



input word	expected output word
...	
painted	Claude
painted	Monet
painted	the
painted	grand
...	

# Skip-gram Approach (4/9)

- Randomly initialize the network (weights):



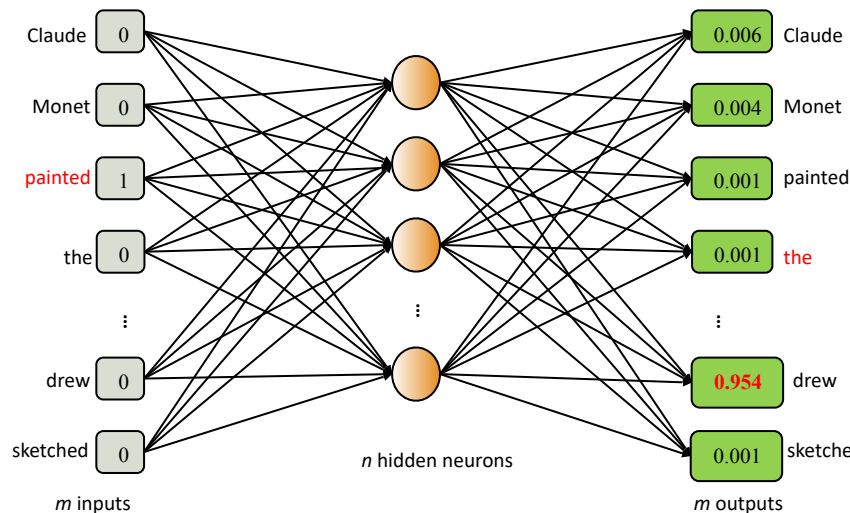
**Randomly initialize the weight matrices**



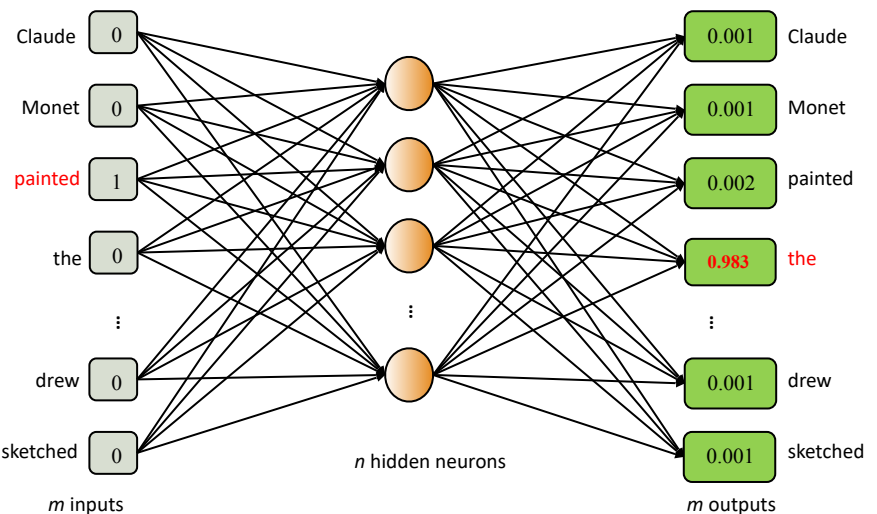
# Skip-gram Approach (5/9)

The objective of the model (the weights) is to make a correct prediction of each training word pair!!

- Training pair: {painted, the}



Bad weights...



Good weights...

# Skip-gram Approach (6/9)

---

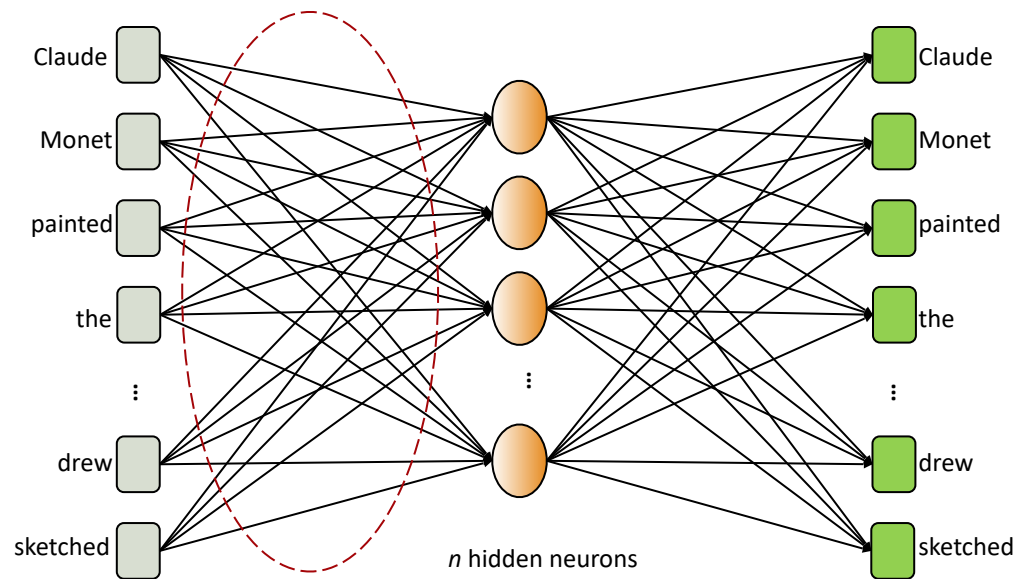
Formal definition of the objective:

- To maximize:  $\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$ 
  - $c$ : window size (e.g., 2)
  - $w_t$ : input word
  - $w_{t+j}$ : expected output word
- $p(w_{t+j} | w_t) = \frac{\exp(V_t^T V'_{t+j})}{\sum_{l=1}^m \exp(V_t^T V'_l)}$ 
  - $V$ : a row/column of the weight matrices

Methodologies like backpropagation can be employed to object good weights!!

# Skip-gram Approach (7/9)

After training, the weight matrix  $W$  consists of word embedding.



$W_{m \times n}$ : weight matrix

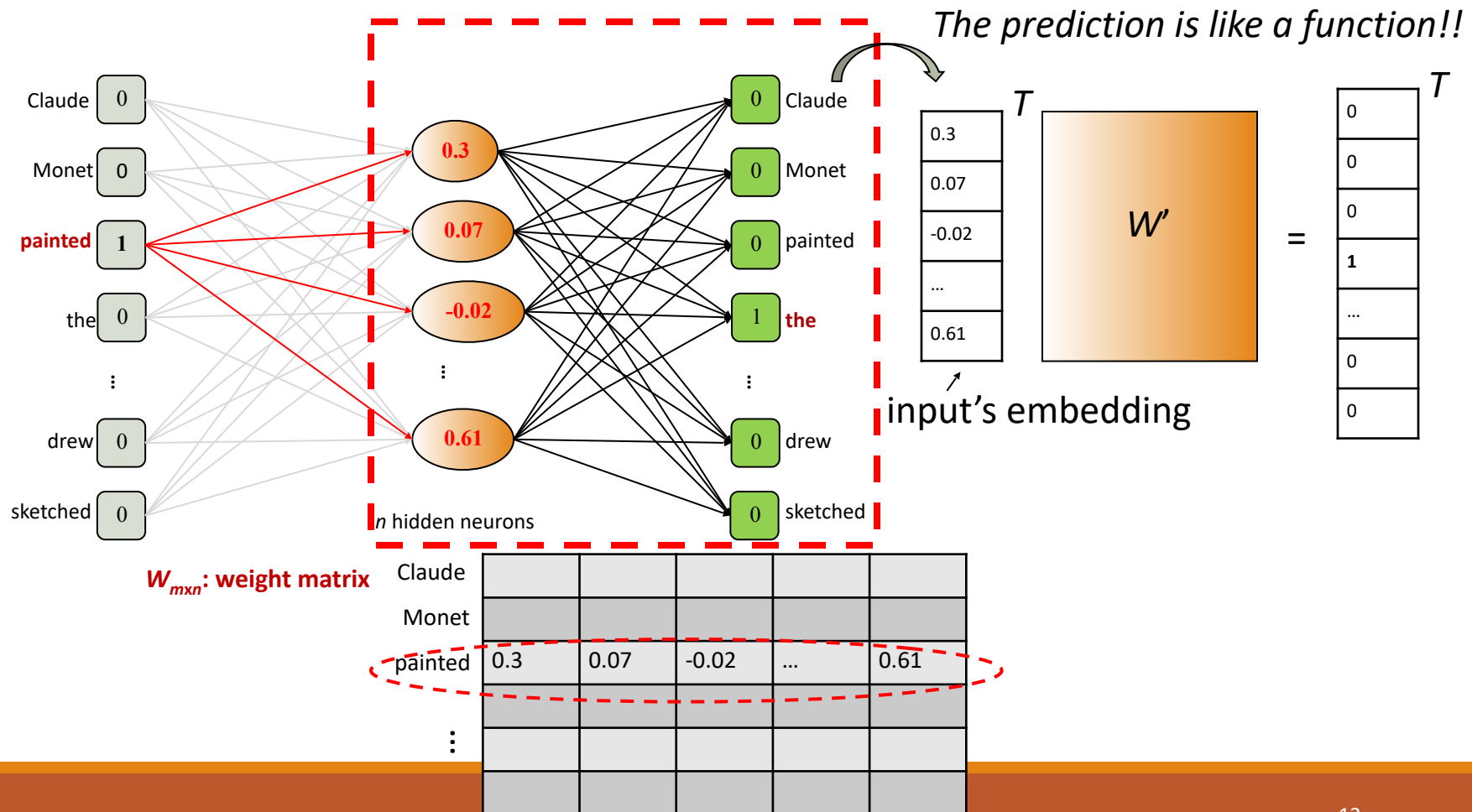


Claude					
Monet					
painted	0.3	0.07	-0.02	...	0.61
:					

WHY?

Every word now can be represented as a  $n$ -dim embedding vector!!

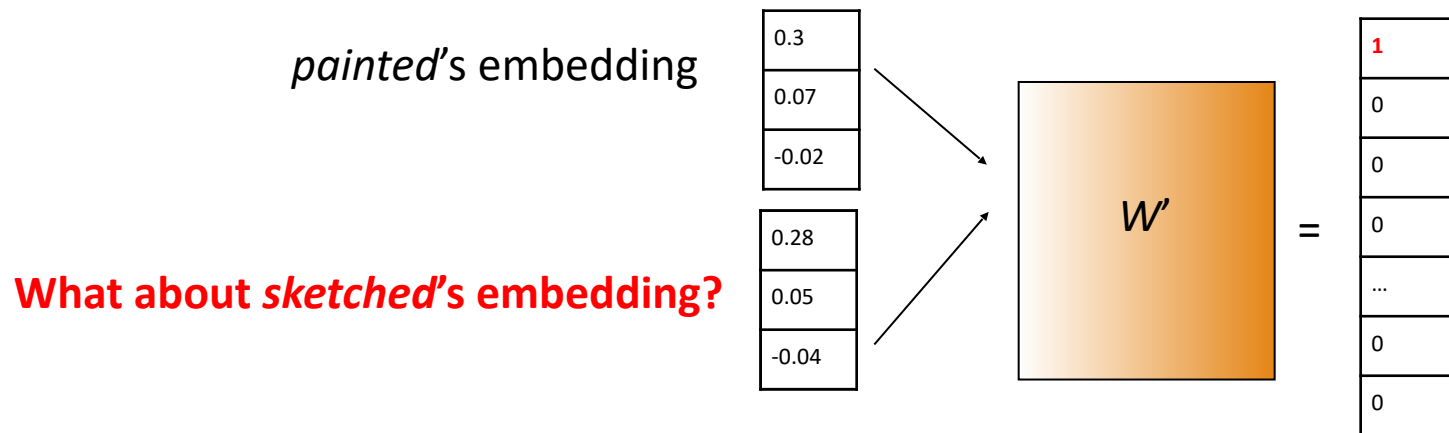
# Skip-gram Approach (8/9)



# Skip-gram Approach (9/9)

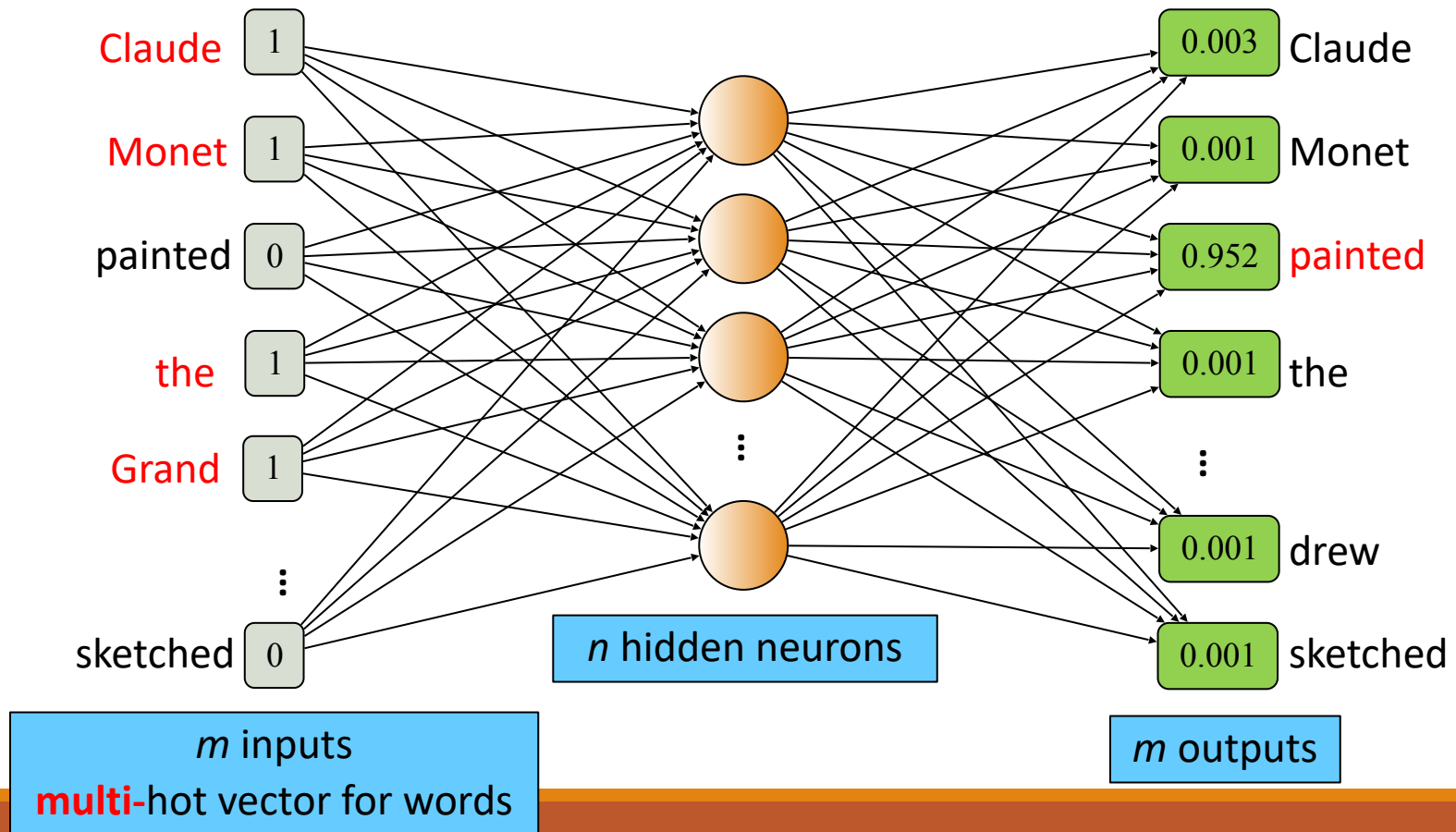
**Semantically similar words (synonyms) will produce similar embeddings!!**

- Because they were trained to predict similar surrounding words!!
- Or ... they form similar training pairs!!
- Training dataset: {**painted**, Claude} {**painted** Monet} ...  
                          {**sketched**, Claude} {**sketched**, Monet} ...



# CBOW

Predict the center word based on the surrounding words.



# Use of Embeddings

## 1. Find close words

- Cosine similarity of vectors.
- Words close to “France”

Word Cosine distance	
spain	0.678515
belgium	0.665923
netherlands	0.652428
italy	0.633130
switzerland	0.622323
luxembourg	0.610033
portugal	0.577154
russia	0.571507
germany	0.563291
catalonia	0.534176

Example provided by Google word2vec project

## 2. Reasoning the relationships between words

- $V_{San\_Francisco} - V_{california} + V_{colorado} = V_{Denver}$
- $V_{king} - V_{man} + V_{woman} = V_{queen}$

## 3. Inputs of sophisticate deep learning networks

# Which Approach is Better?

---

Mikolov highlighted:

- Skip-gram works well with small corpora and rare terms.
- CBOW shows higher accuracies for frequent words and is much faster to train.

*Some studies show skip-gram is better...*



# Pretrained Vectors

---

The computation of the word vectors can be resource intensive!!

Luckily, for most applications, you won't need to compute your own word vectors.

- Google provides a pretrained Word2vec vector set based on English Google News articles.
- <https://code.google.com/archive/p/word2vec/>

## Pre-trained word and phrase vectors

We are publishing pre-trained vectors trained on part of Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The phrases were obtained using a simple data-driven approach described in [2]. The archive is available here: [GoogleNews-vectors-negative300.bin.gz](https://code.google.com/archive/p/word2vec/).

Download it before  
you use it!!

- Facebook published pretrained **fastText** models for 294 languages!!

But if your application relies on **specialized vocabulary**, you will need to train word vectors on text from your application domain.

# Word2Vec Practice (1/2)

```
In [1]: import gensim
```

```
In [2]: w2v_model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz', binary=True)
```

```
In [3]: w2v_model.vector_size
```

```
Out[3]: 300
```

```
In [4]: w2v_model.most_similar('university')
```

```
Out[4]: [('universities', 0.7003918886184692),  
         ('faculty', 0.6780906915664673),  
         ('university', 0.6758289933204651),  
         ('undergraduate', 0.6587095260620117),  
         ('university', 0.6585438251495361),  
         ('campus', 0.6434986591339111),  
         ('college', 0.638526976108551),  
         ('academic', 0.6317198276519775),  
         ('professors', 0.6298649907112122),  
         ('undergraduates', 0.6149812936782837)]
```

```
In [5]: w2v_model.most_similar('obama')
```

```
Out[5]: [('mccain', 0.7319012880325317),  
         ('hillary', 0.7284600138664246),  
         ('obamas', 0.7229630947113037),  
         ('george_bush', 0.720567524433136),  
         ('barack_obama', 0.7045838832855225),  
         ('palin', 0.7043113708496094),  
         ('clinton', 0.69344448480606079),  
         ('clintons', 0.6816835403442383),  
         ('sarah_palin', 0.6815145015716553),  
         ('john_mccain', 0.6800708770751953)]
```

## Gensim is a FREE Python library



Scalable statistical semantics



Analyze plain-text documents for semantic structure



Retrieve semantically similar documents

# Word2Vec Practice (2/2)

```
In [5]: w2v_model.most_similar('obama')
```

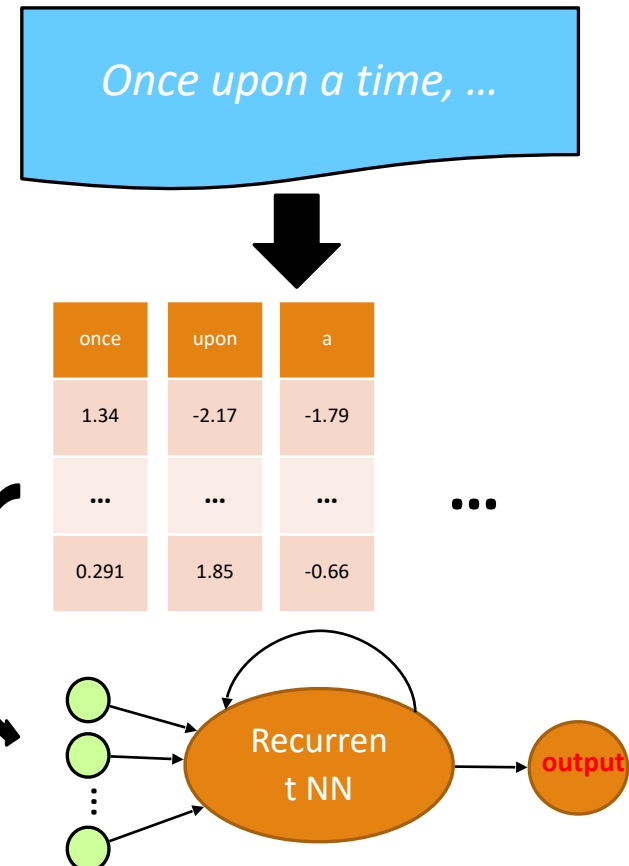
```
Out[5]: [('mccain', 0.7319012880325317),  
         ('hillary', 0.7284600138664246),  
         ('obamas', 0.7229630947113037),  
         ('george_bush', 0.720567524433136),  
         ('barack_obama', 0.7045838832855225),  
         ('palin', 0.7043113708496094),  
         ('clinton', 0.6934448480606079),  
         ('clintons', 0.6816835403442383),  
         ('sarah_palin', 0.6815145015716553),  
         ('john_mccain', 0.6800708770751953)]
```

```
In [6]: obama_vector = w2v_model['obama']
```

```
In [7]: obama_vector
```

```
Out[7]: array([-1.23535156e-01,  7.22656250e-02,  1.71875000e-01,  
              -1.25976562e-01, -3.02734375e-01, -4.49218750e-01,  
               1.71875000e-01, -4.34570312e-02, -1.32812500e-01,  
              -5.27343750e-01,  1.39648438e-01, -1.23535156e-01,  
              -1.46484375e-01,  4.12597656e-02,  2.99072266e-01,  
               4.44335938e-02, -9.37500000e-02,  4.25781250e-01,  
              -2.29492188e-01,  3.00781250e-01, -4.27734375e-01,  
               3.16406250e-01, -2.02148438e-01,  1.03515625e-01,  
              -3.18359375e-01, -9.08203125e-02, -5.81054688e-01,  
              -2.25585938e-01,  6.29882812e-02,  2.00195312e-01,  
               2.33154297e-02, -2.45117188e-01,  3.67187500e-01,  
               1.21093750e-01,  1.58203125e-01,  3.59375000e-01,  
              -2.59765625e-01, -3.36914062e-02, -1.35742188e-01,  
               1.04235000e-01,  0.82700000e-01,  2.51053125e-01])
```

Inputs of sophisticate deep learning networks



# BERT and Transformers

---

Reference:

- <https://towardsdatascience.com/transformers-state-of-the-art-natural-language-processing-1d84c4c7462b>
- <http://jalammar.github.io/illustrated-transformer/>

# BERT - Transformer

## BERT – Bidirectional **Encoder** Representations from **Transformers**

- A well-known word embedding method, broke several records of natural language processing contests.
- **The pre-trained model** was released in 2018
  - <https://github.com/google-research/bert>

### Bert: Pre-training of deep bidirectional transformers for language understanding

J Devlin, **MW Chang**, K Lee, K Toutanova - arXiv preprint arXiv ..., 2018 - arxiv.org

We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and ...

☆ 77 Cited by 10047 Related articles All 25 versions 🔗

#### 教育背景



**University of Illinois at Urbana-Champaign**

Doctor of Philosophy (Ph.D.) · Computer Science

2005 年 – 2011 年



**National Taiwan University**

Master of Science (M.S.) · Computer Science

2001 年 – 2003 年



**National Taiwan University**

Bachelor of Science (B.S.) · Computer Science

1997 年 – 2001 年

# Transformer (1/2)

---

Transformer is designed to handle **sequential data**, especially natural languages.

It is based on the **encoder-decoder** architecture enhanced with the **attention** mechanism.

## Attention is all you need

[A Vaswani, N Shazeer, N Parmar...](#) - Advances in neural ..., 2017 - papers.nips.cc

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder and decoder configuration. The best performing such models also connect the encoder and decoder through an attention mechanism ...



Cited by 12564

[Related articles](#)

[All 21 versions](#)

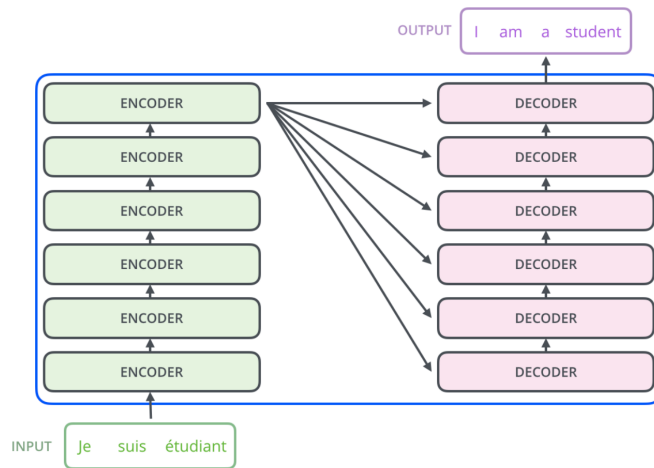


# Transformer (2/2)

**Encoder:** to extract important features from data – **code**

- To compress, remove noisy information of data
- Or ... to produce meaningful representation (vector) of the given data.

**Decoder:** to restore data (to generate correct data) from the code of the data



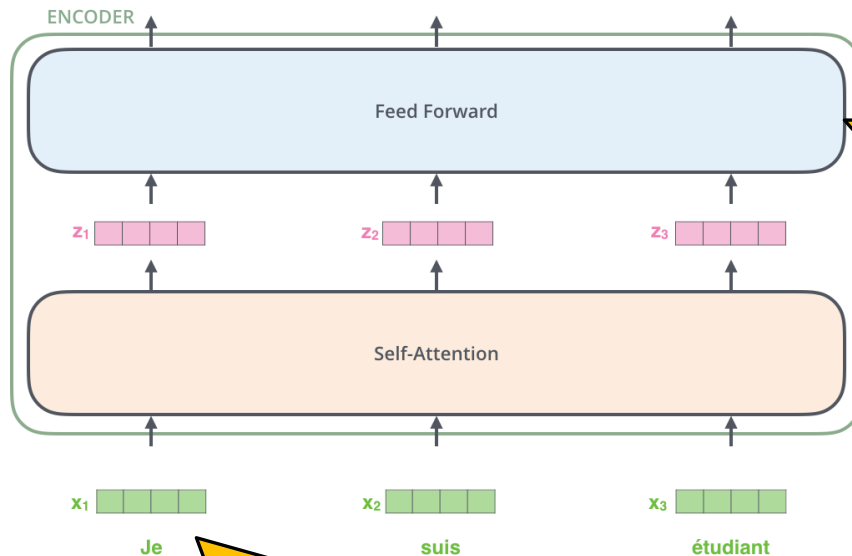
Not new, but **attention** takes it to another level!!

<http://jalammar.github.io/illustrated-transformer/>

# Encoder

The input **s** first flow through a *self-attention* layer

- A layer that helps the encoder **look at other words** in the input sentence as it encodes a specific word.



The feed-forward neural network generates an output for each  $z_i$  and the output from the feed-forward neural network is passed into the next encoder block's self-attention layer and so on.

Each input token is first represented as a embedding



# Self-Attention (1/8)

---

Think about this:

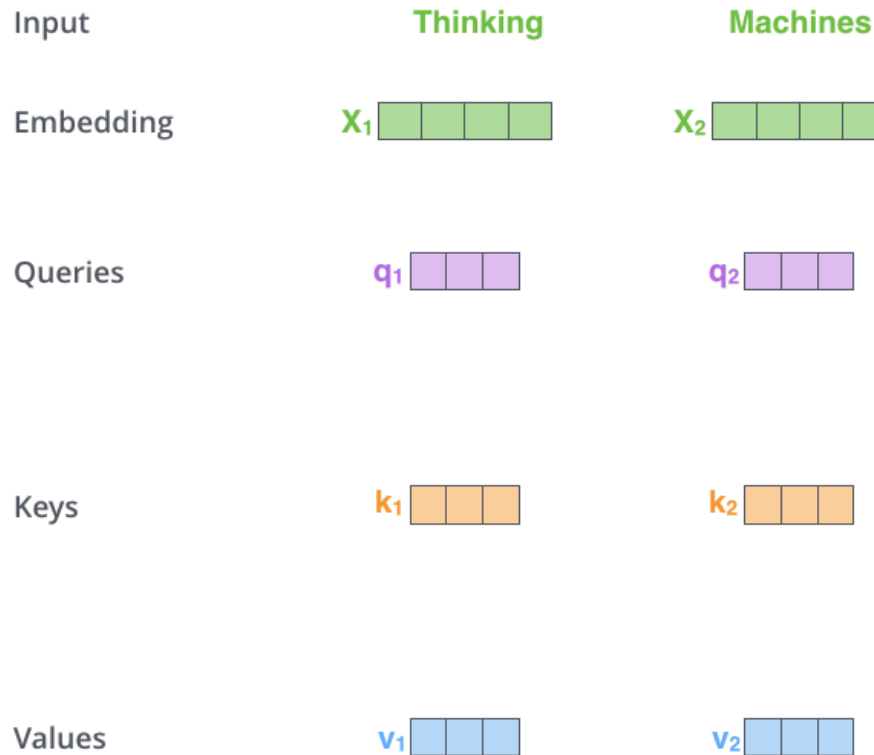
*“The animal didn’t cross the street because **it** was too tired”*

- *it* refers *animal*, so the code of *it* should include the information (code) of *animal*!!

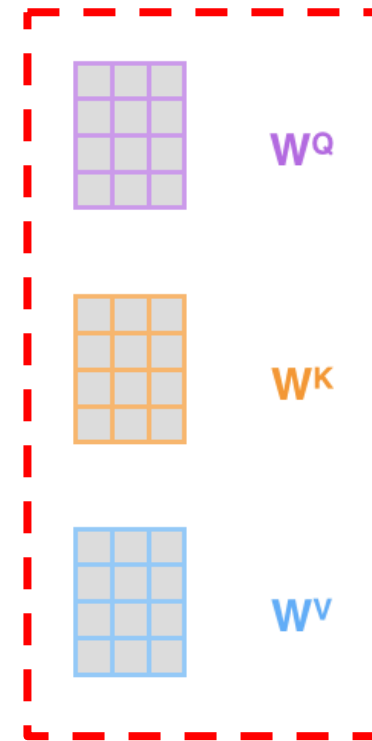
For an input embedding  $x$ , attention generates three meta-outputs:

- Query:  $q_x$
- Key:  $k_x$
- Value:  $v_x$

# Self-Attention (2/8)



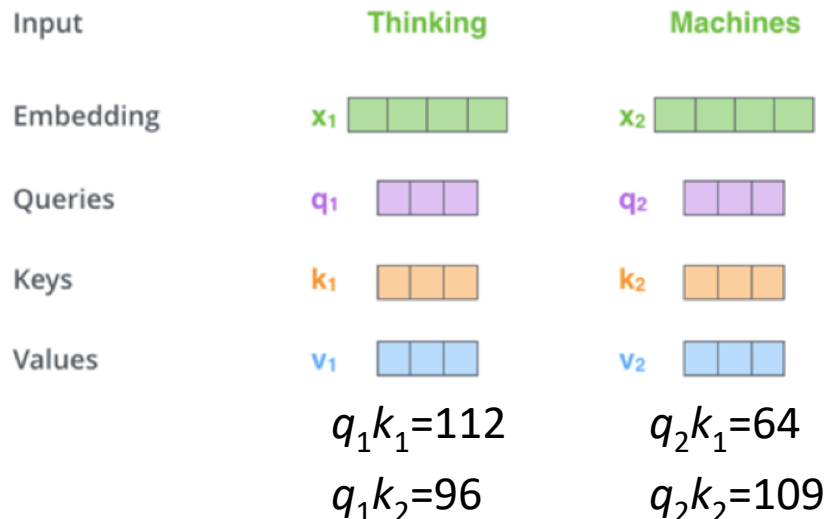
The matrices are learning parameters



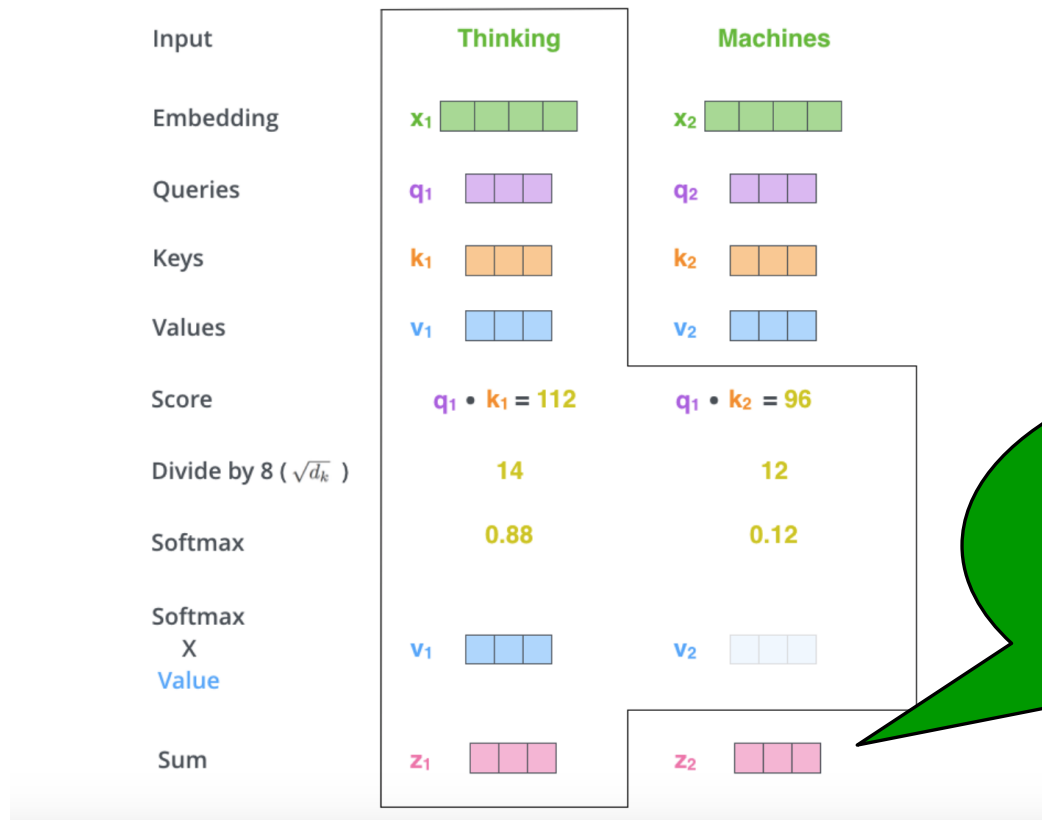
# Self-Attention (3/8)

After having the 3 meta-outputs  $Q$ ,  $K$ , and  $V$  of **ALL** input embeddings

- The self-attention layer calculates a **score vector** for each input embedding.
- Which is used to aggregated the input embedding's output



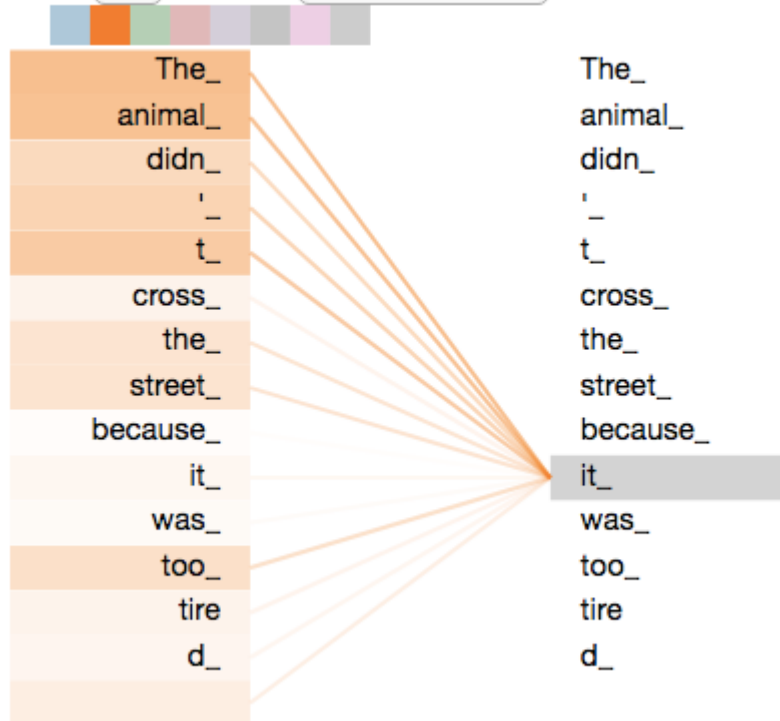
# Self-Attention (4/8)



This way, the  $z$  (meta-code) of each input is the weighted combination of all values (all information of the input tokens)

# Self-Attention (5/8)

Layer: 5 Attention: Input - Input



The scores show that *The* and *animal* contribute a lot to the output of *it*

<http://jalammar.github.io/illustrated-transformer/>

# Self-Attention (6/8)

---

## Multi-head:

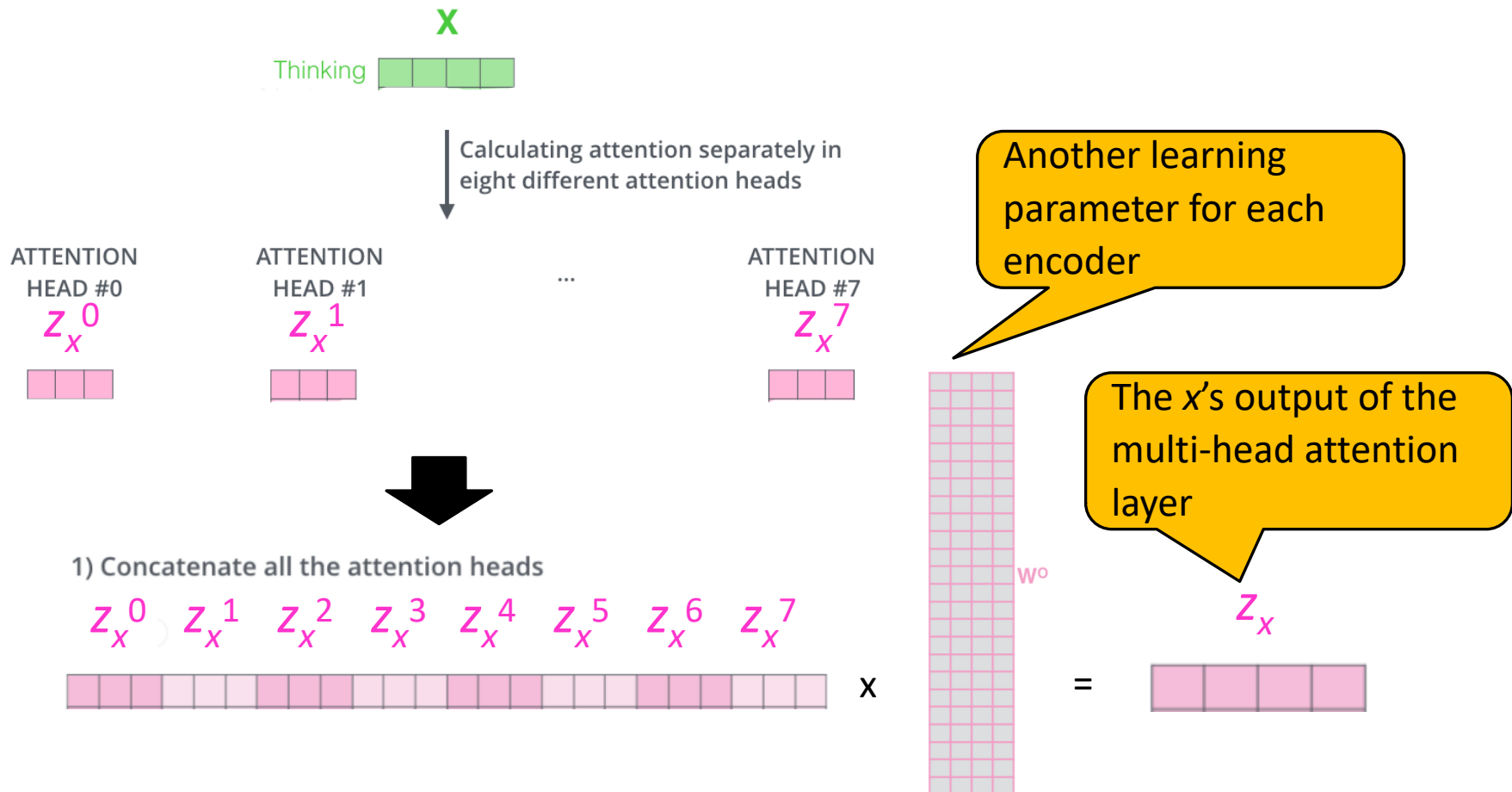
- A single attention-head could miss some of the words in input that contribute most to the spotlight word.
- Instead of using a single attention-head, **multiple** attention-heads are employed.
  - Each attention-head has its own learning parameters:  $W^Q$ ,  $W^K$ , and  $W^V$ .

In practice, attention models typically uses **8** attention heads!!

- So **EACH** encoder has **8 sets** of learning parameters.

Each attention-head produces a  $z_x$  that we need to aggregate them all to produce the final output!!

# Self-Attention (7/8)



# Self-Attention (8/8)

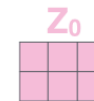
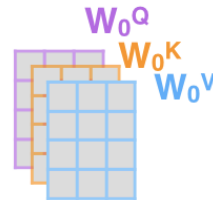
1) This is our input sentence\*

Thinking  
Machines

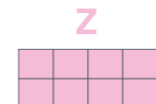
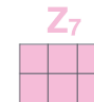
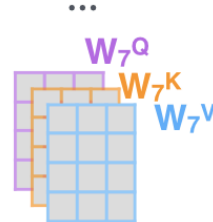
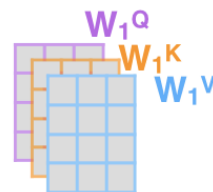
2) We embed each word\*



3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices



\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



<http://jalammar.github.io/illustrated-transformer/>



# The Whole Picture of Transformer

Other components:

- Positional encoding
- Add & normalization
- Decoder

We are not going to the detail  
of these components  
but back to BERT

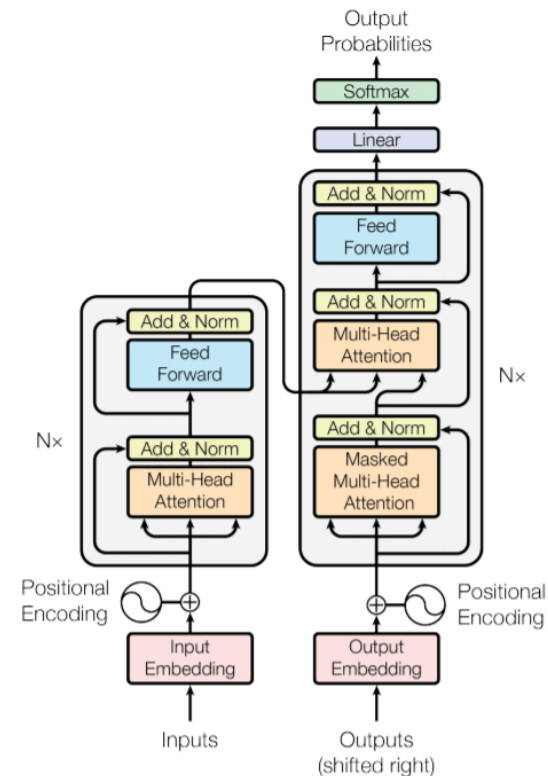
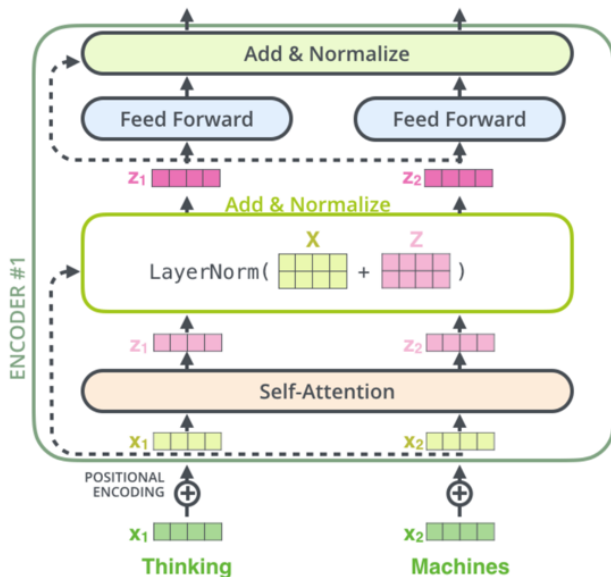


Figure 1: The Transformer - model architecture.

# BERT

In BERT, **only the encoder part** of the transformer is used.

- With multi-heads of self-attention, add & norm, feed forward network, and positional encoding.



<http://jalammar.github.io/illustrated-bert/>

<https://github.com/google-research/bert>

	H=128	H=256	H=512	H=768
L=2	2/128 (BERT-Tiny)	2/256	2/512	2/768
L=4	4/128	4/256 (BERT-Mini)	4/512 (BERT-Small)	4/768
L=6	6/128	6/256	6/512	6/768
L=8	8/128	8/256	8/512 (BERT-Medium)	8/768
L=10	10/128	10/256	10/512	10/768
L=12	12/128	12/256	12/512	12/768 (BERT-Base)

**BERT-Large:** 24-layer, 1024-hidden, 16-heads

**BERT-Base Multilingual:** 102 languages

**BERT-BASE Chinese:** Traditional and Simple

# BERT vs Word2Vec

---

BERT generates the embedding of a word by considering **ALL** the words of the input sentence simultaneously.

- By means of the attention mechanism.

The embeddings of Word2Vec are **context-independent!!**

- *“the game will lead to a **tie** if both the guys **tie** their final **tie** at the same time.”*
- Word2vec would give the three **tie** the same embedding while each **tie**’s embedding in BERT will be different.

Because BERT is context-dependent, we need the **pre-trained model** every time while generating the embeddings of words.

- Word2Vec: we simply copy and use the trained vectors.

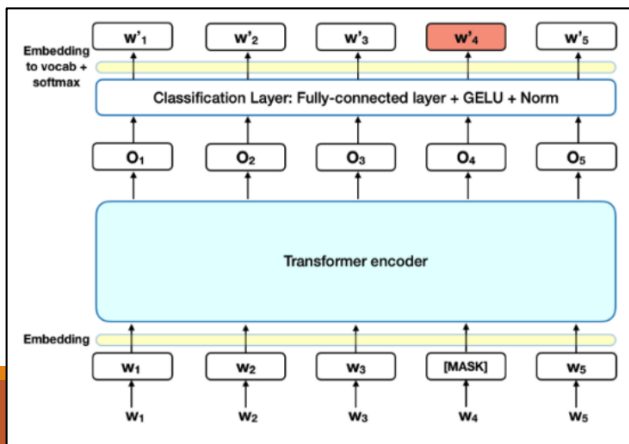
# How do **THEY** train the models

BERT was pre-trained on a large corpus of un-labelled text including the entire Wikipedia(that's 2,500 million words) and book corpus (800 million words).

Two training methods:

- **Unmasked language model**

- to predict the masked words correctly given the context of unmasked words.



- **Next sentence prediction**

- to predict the order of the two sentences.

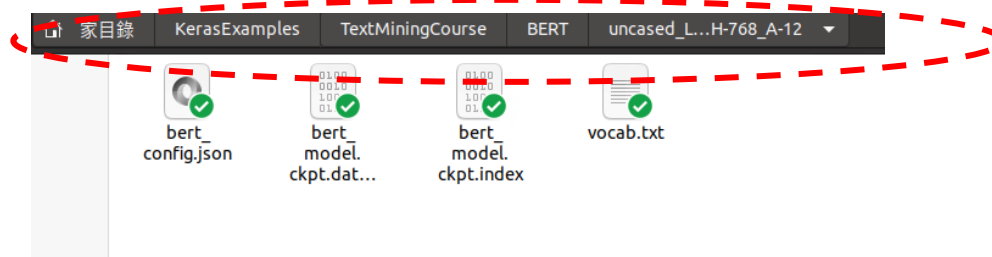
Return 1 if the first sentence comes after the second sentence and 0 otherwise

# How do **We** use the models (1/6)

First, download one of the pre-trained models

- <https://github.com/google-research/BERT>
- The zip file contains three item:
  - Checkpoint file: the pre-trained weights
  - Vocab file: dictionary and word id
  - Config file: hyper-parameters

Do not forget where you put the files



Then, choose a library

- Here, we take `keras-bert` as an example
- <https://github.com/CyberZHG/keras-bert>
- To install the library: `pip install keras-bert`

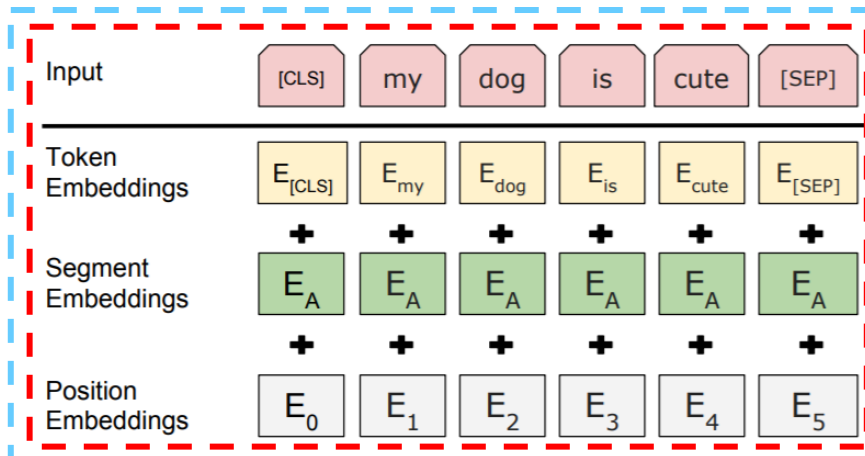
# How do We use the models (2/6)

Single text task or text pair task:

- **Single text task** – e.g., to classify a given text is a spam or not.
- **Text pair task** – e.g., to classify a given question-answer pair is relevant or not.

BERT takes 3 types of input: *token embeddings*, *segment embeddings*, and *positional embeddings*

*example of a single text*



*example of a text pair*

# How do We use the models (3/6)

---

```
In [1]: from keras_bert import extract_embeddings
        from keras_bert import load_vocabulary
        from keras_bert import Tokenizer
```

```
model_path = 'uncased_L-12_H-768_A-12'
dict_path = 'uncased_L-12_H-768_A-12/vocab.txt'
```

```
In [2]: bert_token_dict = load_vocabulary(dict_path)
        bert_tokenizer = Tokenizer(bert_token_dict)
```

```
In [3]: single_input_text = ['all work and no play.']

tokens = bert_tokenizer.tokenize(single_input_text[0])
indices, segments = bert_tokenizer.encode(single_input_text[0])
print(len(tokens))
print(indices)
print(segments)
print(tokens)
```

8

[101, 2035, 2147, 1998, 2053, 2377, 1012, 102]

[0, 0, 0, 0, 0, 0, 0, 0]

['[CLS]', 'all', 'work', 'and', 'no', 'play', '.', '[SEP]']

# How do We use the models (4/6)

```
In [8]: tokens = bert_tokenizer.tokenize(first='all work and no play.', second='this is an order.')
indices, segments = bert_tokenizer.encode(first='all work and no play.', second='this is an order.')
print(len(tokens))
print(indices)
print(segments)
print(tokens)

14
[101, 2035, 2147, 1998, 2053, 2377, 1012, 102, 2023, 2003, 2019, 2344, 1012, 102]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
['[CLS]', 'all', 'work', 'and', 'no', 'play', '.', '[SEP]', 'this', 'is', 'an', 'order', '.', '[SEP]']
```

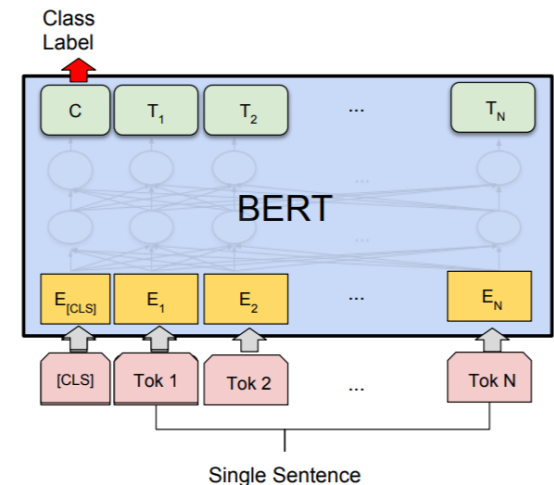
Once the inputs are ready, we pass them to **the pre-trained model** to output all (including [CLS] and [SEP]) their embeddings!!

Usually, we do not need all the embeddings.

- We only keep (use) the embedding of the first [CLS] token.
- The embedding aggregates the information (context) of the input text.

We then pass the embedding to the downstream task.

- E.g., a feed-forward network for spam classification.





# How do We use the models (5/6)

---

```
In [4]: embeddings = extract_embeddings ( model_path , single_input_text )
print ( len ( embeddings ))
print ( len ( embeddings [ 0 ]))
print ( len ( embeddings [ 0 ][ 0 ]))
print ( embeddings [ 0 ][ 0 ])
```

1

8

768

```
[-1.42068073e-01 3.50723922e-01 3.51595491e-01 -4.90611196e-02
 2.13707358e-01 -6.79893851e-01 3.72713089e-01 2.14657858e-01
 4.42356557e-01 -1.61857173e-01 -8.90293181e-01 2.69119680e-01
 2.63698488e-01 -2.33761013e-01 4.26663637e-01 3.66816781e-02
 3.67483169e-01 4.55938369e-01 -4.44805264e-01 -2.61960447e-01
 ... ..]
```

# How do We use the models (6/6)

```
In [6]: two_input_text = [('all work and no play.', 'this is an order.'])
```

```
In [7]: two_inputs_embeddings = extract_embeddings(model_path, two_input_text)
print(len(two_inputs_embeddings[0]))
print(two_inputs_embeddings[0][0])
```

14

```
[ -6.02293909e-01  1.64158642e-01  1.98453903e-01  6.07691646e-01
  2.02574238e-01 -6.00710660e-02  1.20144188e-01 -2.48182192e-01
  3.67996663e-01 -2.47432753e-01 -2.67240167e-01 -3.03636789e-01
  5.86615264e-01  1.18524395e-03  2.56111503e-01 -1.35984093e-01
  4.13614869e-01  1.69194758e-01 -2.54904568e-01  4.86191988e-01
  1.80502146e-01  1.25615388e-01  5.78263462e-01 -2.72199400e-02
  5.99037528e-01 -6.78412691e-02  5.49786329e-01  3.58132839e-01]
```

# Pre-training and Fine Tuning

Not just inputting the embedding of [CLS] to downstream task, you can connect BERT with downstream networks!!

- For example, connect BERT with a fully connected classification layer
- That is what we called “**end-to-end**” model!!

The training process will not only learn the weights of the classification layer, but also **tune the weights of BERT** (e.g.,  $W^Q$ ,  $W^K$ , and  $W^V$ ).

- You customize BERT to your tasks (rather than using a generalized pre-trained model).

Note that fine tuning such end-to-end model would be **time (memory) consuming**.

- BERT-Base, Multilingual Cased: 104 languages, 12-layer, 768-hidden, 12-heads, **110M parameters**
- BERT-Large, Uncased (Whole Word Masking): 24-layer, 1024-hidden, 16-heads, **340M parameters**

