

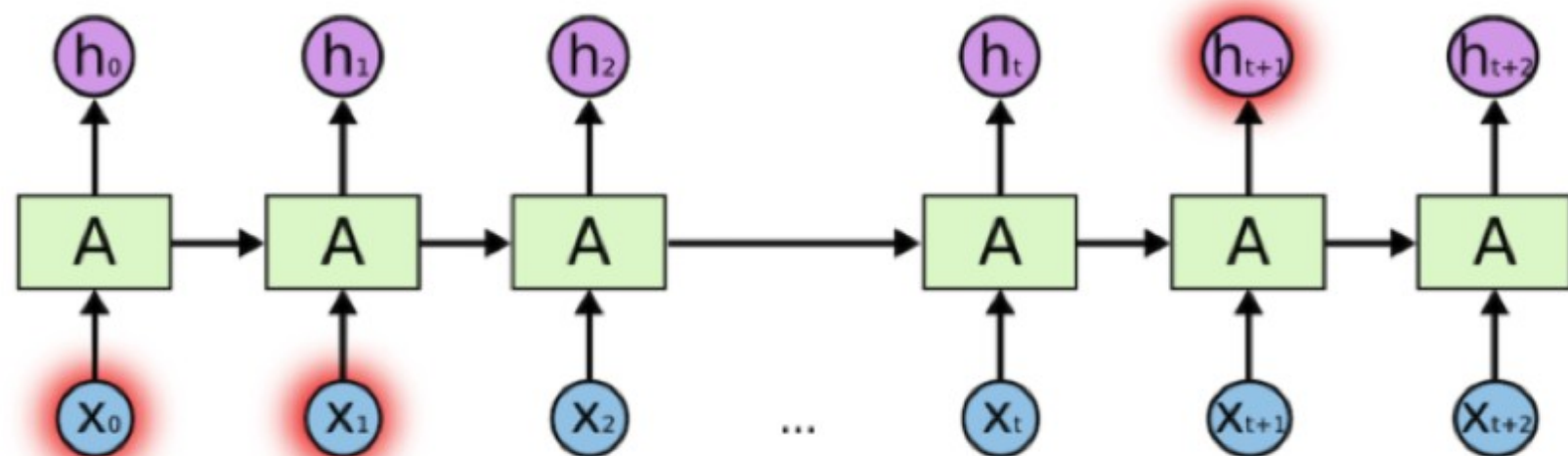
SCS-3546 Deep Learning

Session 7

Deep Models for Text

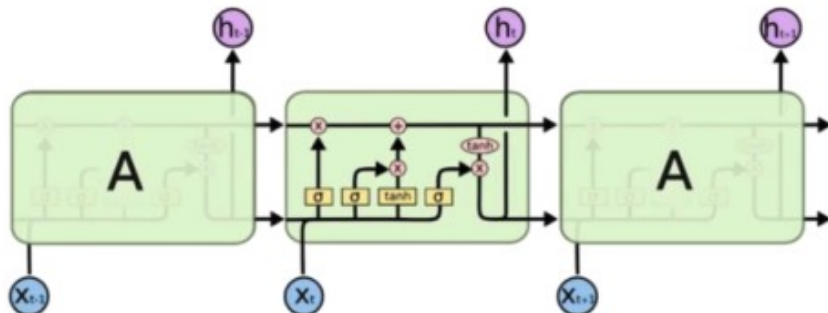
Recap

- RNNs remember things **learned from prior input(s)** while **generating output(s)**
- take two inputs, the current example they see, and a representation of the previous input
- The output at time step t depends on the current input as well as the input at time $t-1$



LSTM

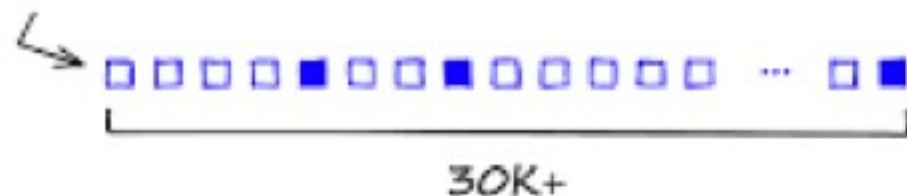
- Long short-term memory is a special kind of RNN
 - specially made for solving vanishing gradient problems
- LSTM neurons have a branch that allows passing information to skip the long processing of the current cell
- capable of learning long-term dependencies
- It will do fine until 100 words, but around 1,000 words, it starts to lose its grip
- very slow to train
- take input sequentially one by one, which doesn't use up GPUs very well



Different types of matrices - terminology

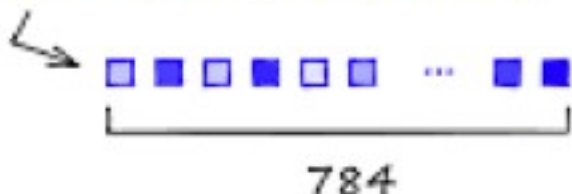
sparse

$[0, 0, 0, 1, 0, \dots 0]$



dense

$[0.2, 0.7, 0.1, 0.8, 0.1, \dots 0.9]$



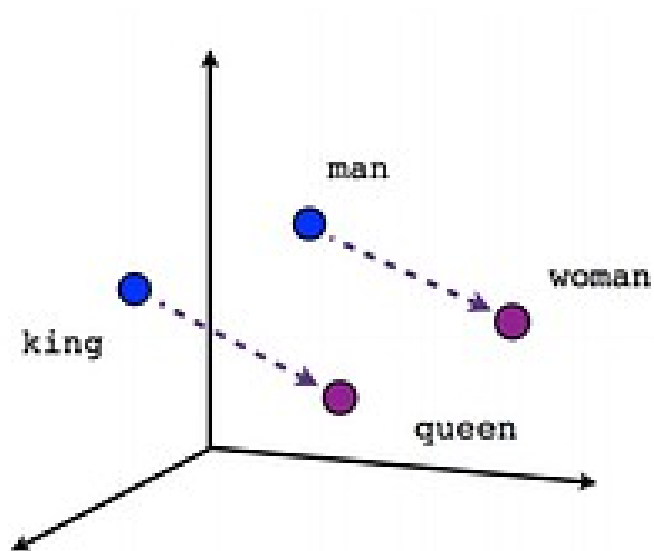
Sparse: everything zero except the target word

Dense: float values

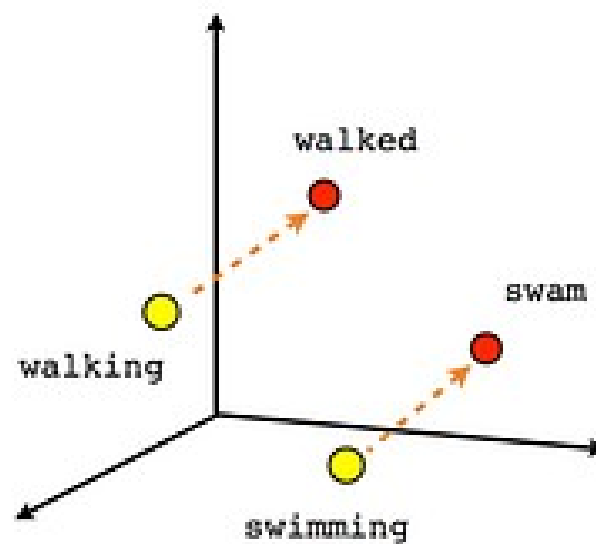
Applications of Deep Models

- Text Classification and Categorization
- Named Entity Recognition
- Part of Speech Tagging
- Semantic Parsing and Query Answering
- Summarization
- Paraphrasing Detection
- Synonym generation for search
- Machine Translation

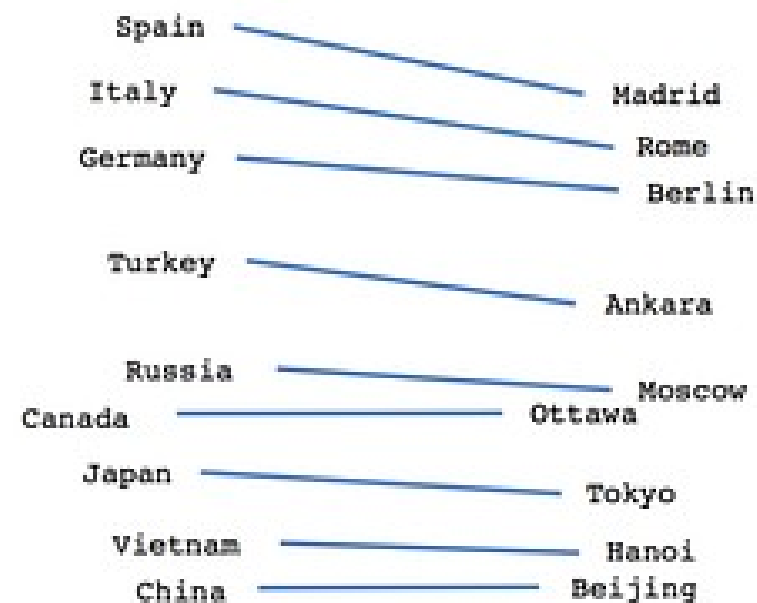
Word Representations



Male-Female



Verb tense



Country-Capital

- words and documents are represented in the form of real-valued vectors
- It is an **advancement in NLP** that has improved the ability of computers to understand text-based content in a better way

Given a vocabulary of 10,000 words, what's the simplest way to represent each word numerically?

traditional method of encoding words:

- assigning an integer index to each unique word in the vocabulary
- then create vectors of a length equal to the size of the vocabulary
- word encoding:
 - all entries zero except the one at the index of the word, which was set to 1

Encoding with Integers

Vocabulary

index:	word:
--------	-------

0	aardvark
---	----------

1	able
---	------

...	...
-----	-----

2409	black
------	-------

2410	bling
------	-------

...	...
-----	-----

3202	candid
------	--------

3203	cast
------	------

3204	cat
------	-----

...	...
-----	-----

5281	is
------	----

5282	island
------	--------

...	...
-----	-----

8676	the
------	-----

8677	thing
------	-------

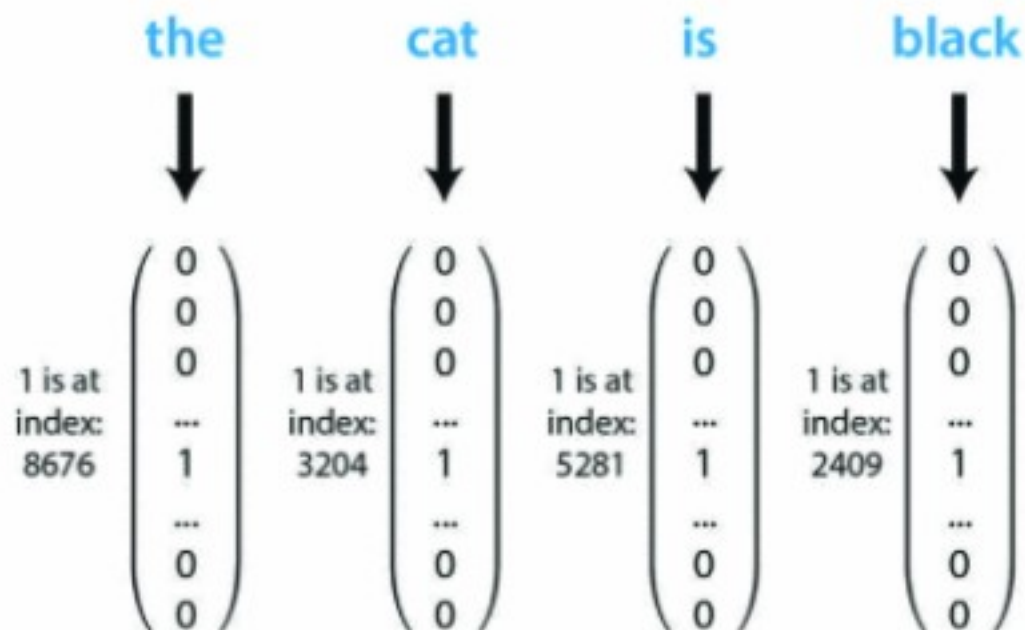
...	...
-----	-----

9999	zombie
------	--------

10,000
words
with
indices

Vocabulary

index:	Word:
0	aardvark
1	able
...	...
2409	black
2410	bling
...	...
3202	candid
3203	cast
3204	cat
...	...
5281	is
5282	island
...	...
8676	the
8677	thing
...	...
9999	zombie




Long Sparse (mostly zero) vectors

Rome = [1, 0, 0, 0, 0, 0, ..., 0]

Paris = [0, 1, 0, 0, 0, 0, ..., 0]

Italy = [0, 0, 1, 0, 0, 0, ..., 0]

France = [0, 0, 0, 1, 0, 0, ..., 0]



Q

1. similarity issue

- similar words like “cat” and “tiger” have somewhat similar features?

2. The vocabulary size and computation issues

Q

How about document encoding:

- The simplest methods either **concatenate or average** the sentence's constituent word embeddings (**or some mixture of both**)
- each entry one (or a count of the number of occurrences) if the word is present in the document or zero if not
- One hot encoding to see if the word is present in the document (e.g. topic modeling for a document to see the presence of specific words)

Term frequency-inverse document frequency (TF-IDF)

	ablaze	accident	car	caught	fire	jam	kind	sadly	set	swear	true	up	world
0	0.00	0.00	0.00	0.0	0.0	0.00	0.67	0.53	0.00	0.00	0.53	0.0	0.00
1	0.47	0.00	0.00	0.0	0.0	0.47	0.00	0.00	0.47	0.37	0.00	0.0	0.47
2	0.00	0.59	0.47	0.0	0.0	0.00	0.00	0.00	0.00	0.47	0.47	0.0	0.00
3	0.00	0.00	0.64	0.4	0.4	0.00	0.00	0.32	0.00	0.00	0.00	0.4	0.00



advantages:

- Every word is represented by an integer that can be used as a fast look up index to additional information about the word (such as its text string)
- Every word vector is a fixed-sized data structure which makes them easier to process
- Comparing two words for equality is very fast
- We can compare the similarity of documents using techniques such as TF-IDF and cosine similarity



Disadvantages:

- The vectors don't capture any word meaning
- The order of the words is arbitrary
- The vectors are large and therefore wasteful of computer memory and CPU
- Vectors with continuous (floating point) values are more amenable to use with neural nets than those with discrete values



preferred approach:

- Words that are close together in their encodings are similar in some sense (capture features of words)
- The vectors are short and dense rather than long and sparse
- The vectors capture some aspect of the meaning of the words
- The different uses of a particular token can be disambiguated (e.g. "bow")
 - "bow and arrow"
 - "bow of a ship"
- The vectors are more amenable for use with neural nets

Capture features

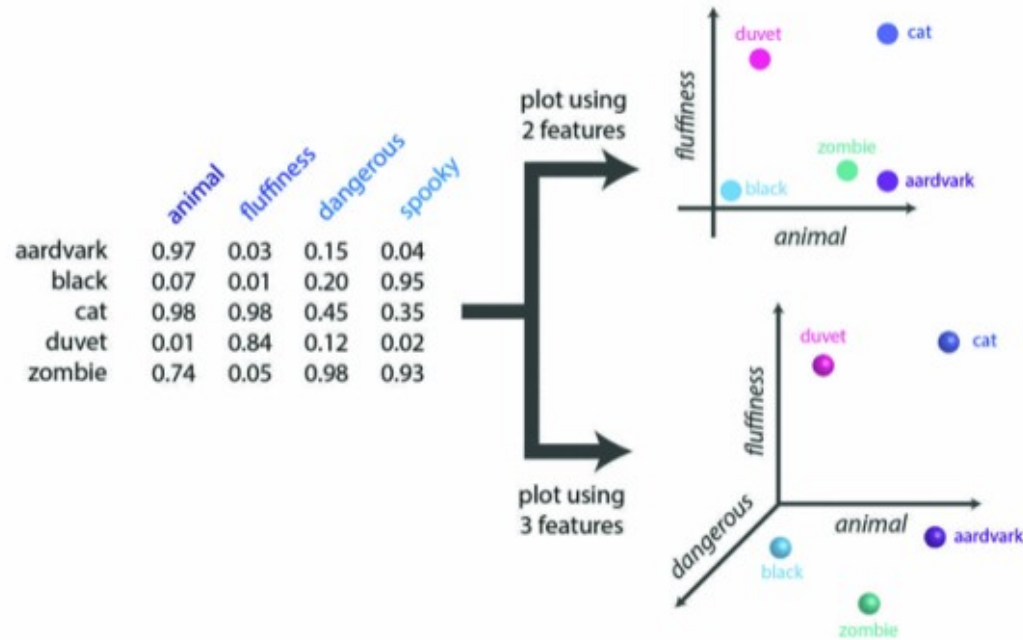
	<i>animal</i>	<i>fluffiness</i>	<i>dangerous</i>	<i>spooky</i>
aardvark	0.97	0.03	0.15	0.04
black	0.07	0.01	0.20	0.95
cat	0.98	0.98	0.45	0.35
duvet	0.01	0.84	0.12	0.02
zombie	0.74	0.05	0.98	0.93

4 semantic features

Given the word “aardvark”:



- We can give it a high value for the feature “animal”, and,
- Relatively low values for “fluffiness”, “dangerous”, and “spooky”



So, we could use the vectors for:

- Classifying the document as being about a topic without having to compare it to others already associated with a topic
- Measure the sentiment of a document without having to hand-engineer sentiment ratings for each word in the vocabulary
- Use the meaning of words and sentences to provide better automated translation

Word Embeddings

- Encoding words as numbers
- Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding
- Enable us to do arithmetic

Embeddings

to capture the semantic and syntactic relationships between words in a text corpus

- is a **dense vector** of floating point values
 - the length of the vector is a parameter we specify

```
embedding_dim = 128
num_ns=10
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                  loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
```

- Instead of specifying the values for the embedding manually, they are trainable parameters
- weights learned by the model during training, in the same way a model learns weights for a dense layer
- The dimension of the embedding vector refers to the number of elements or features in the vector

- A higher dimensional embedding:
 - can capture fine-grained relationships between words
 - but takes more data to learn
 - E.g. a word embedding with 50 values holds the capability of representing 50 unique features
- The choice of the dimension for the embedding **vector is typically based on empirical evaluation** and depends on the specific task at hand
- In general, higher-dimensional embedding vectors **can capture more fine-grained relationships** between words
 - but may also require more training data and computational resources
- lower-dimensional embedding vectors may be **faster** to compute
 - but may not capture all of the relevant relationships between words

(256, 512, 1024)

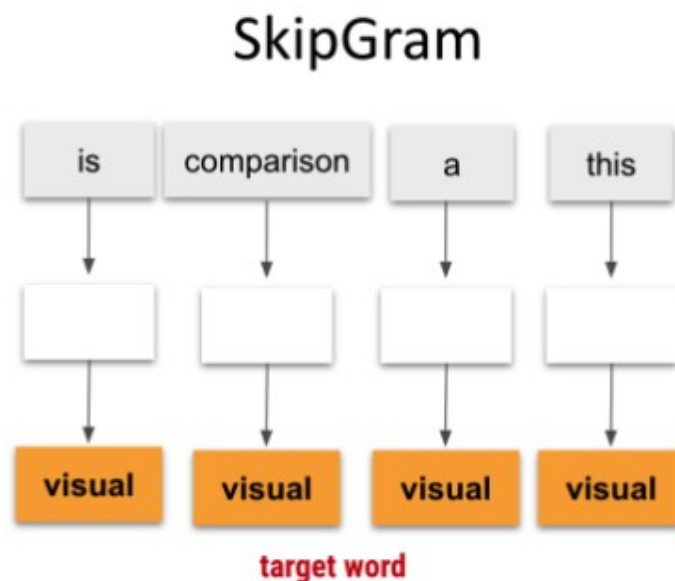
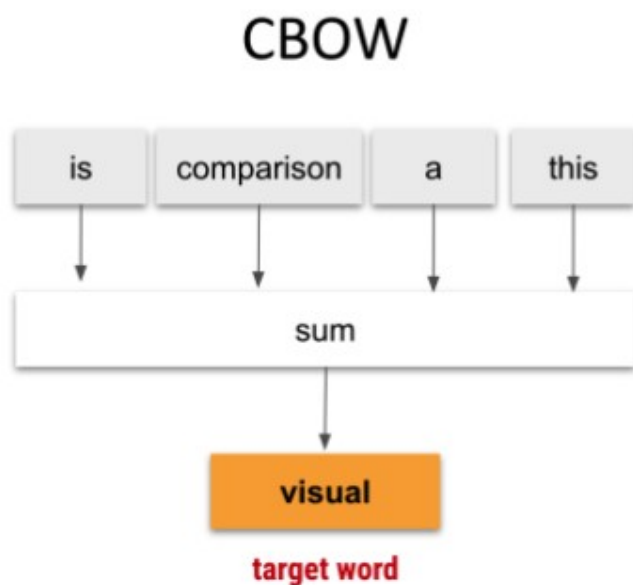
Popular embeddings:

- GloVe
 - Word2Vec
 - FastText
 - TagLM
-
- GloVe and Word2Vec look at words in the context of other words
 - Word2Vec can not distinguish between "bow and arrow" versus "bow of a ship"
 - Representation is a blend of all the words
 - FastText looks at what the next few characters would be (e.g. next 2-3 letters)
 - TagLM: uses tags with parts of speech - POS

Transformer based:

- Bert
- Albert
- RoBert
- GPT
- XLNet
- Performer

- a family of NN model architectures and optimizations that can be used to learn word embeddings from large text datasets
- successful on a variety of downstream natural language processing tasks
- two main methods for learning the embeddings:
 - **Bag-of-Words**
 - **Skip-gram**



Skip-gram Word2Vec

An architecture for computing word embeddings

The network takes in a target word and its context (i.e., the words that appear near the target word in the text) as input,

and tries to predict the probability of each word in the vocabulary appearing in the context



○ Continuous Bag-of-Words Model

- better numerical representation for frequently occurring words
- predicts the middle word based on surrounding context words
- The context consists of a few words before and after the current (middle) word
- This architecture is called a bag-of-words model as the order of words in the context is not important
- the distributed representations of context (or surrounding words) are combined to **predict the word in the middle**
- Each word is encoded using One Hot Encoding in the defined vocabulary and sent to the CBOW neural network
- The hidden layer is a standard fully-connected dense layer
- The output layer generates probabilities for the target word from the vocabulary.

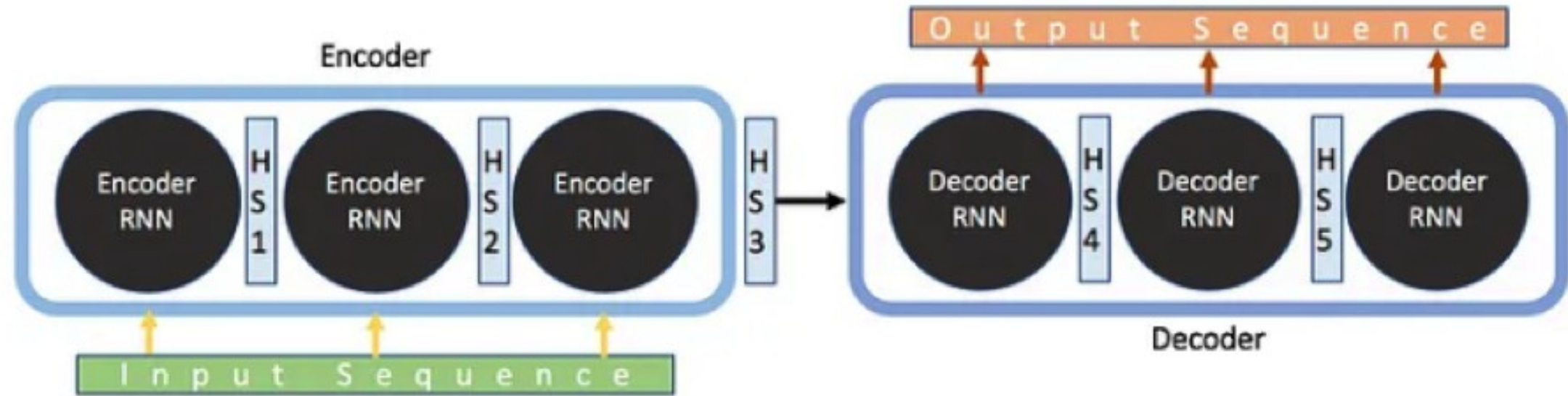
- **Continuous Skip-gram Model**

- predicts words within a certain range before and after the current word in the same sentence
- given a word, we'll try to predict its neighboring words
- Skip-gram tends to work better for smaller datasets and less frequent words

ELMo (Embeddings from Language Models)

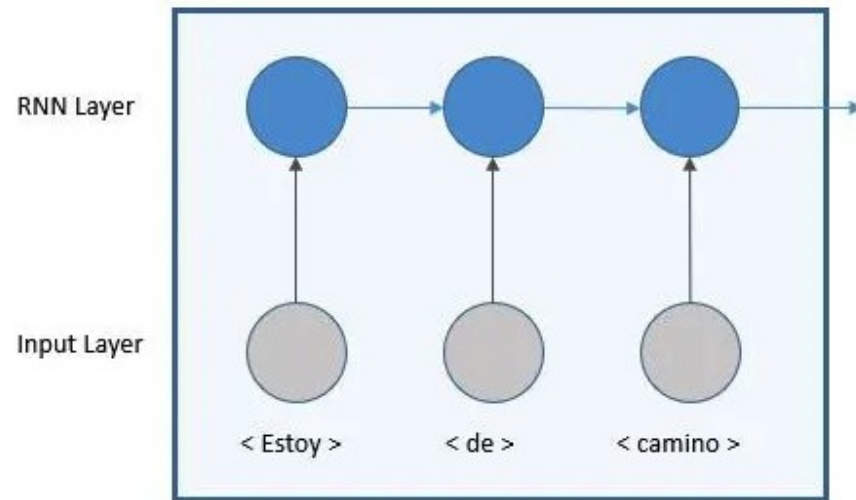
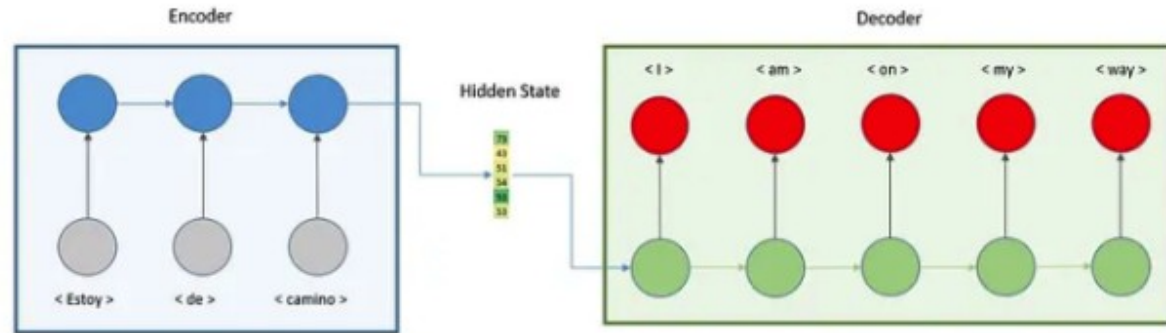
- A deep contextualized word representation model that was introduced in 2018 by researchers at Allen Institute for AI
- It is a type of language model that uses a **bi-directional LSTM** (Long Short-Term Memory) network to generate word embeddings that capture the context-dependent meanings of words
- generates word embeddings that are contextualized
- achieved by concatenating the internal states of a pre-trained bi-directional LSTM network
 - LSTM network has been trained to predict the next word in a sentence given the previous words

Sequence To Sequence Models

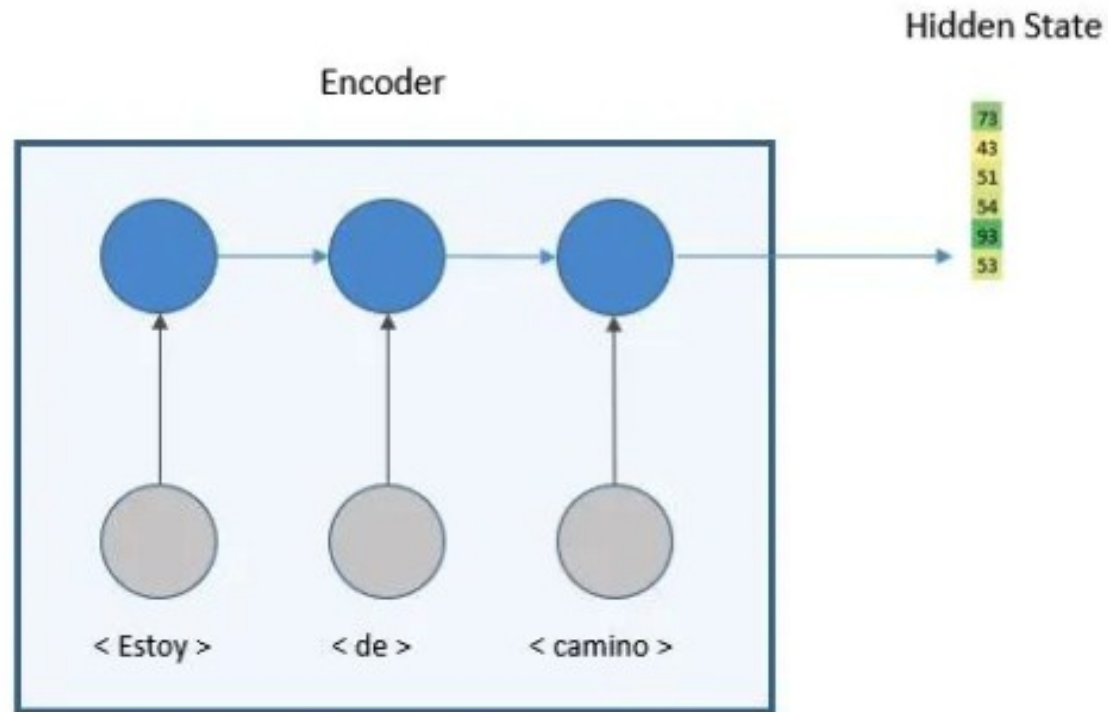


- convert one sequence to another
- read an entire sequence (e.g. a sentence), encode it into a more general representation, then decode it into a new sequence
- The encoders and decoders are often **RNN's**
- Standard seq2seq model
 - Neural Machine Translation
 - the input and output vectors are embeddings

Encoder

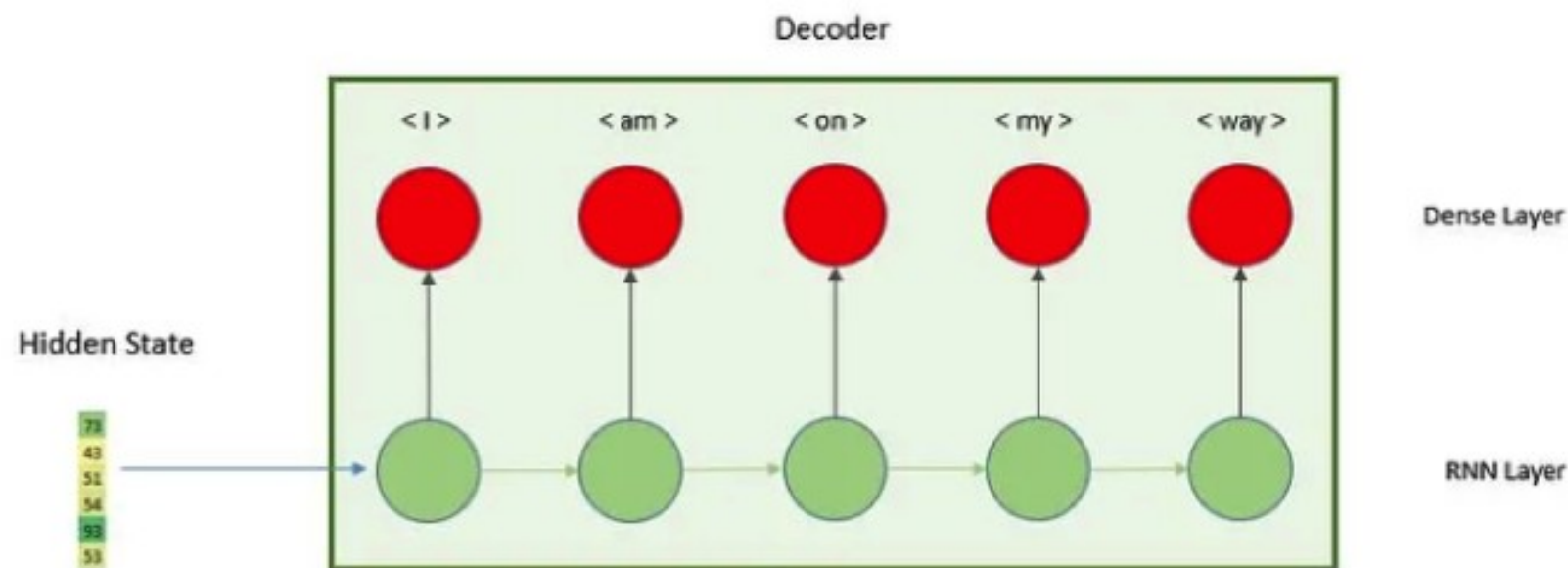


- we convert a sequence of words in Spanish into a two-dimensional vector
- the two-dimensional vector is also known as **hidden state**
- The output of the encoder, the hidden state, is the state of the last RNN timestep



- The output of the encoder
 - a two-dimensional vector that encapsulates the whole meaning of the input sequence
- The length of the vector depends on the number of cells in the RNN

Decoder



- To decode means to convert a coded message into intelligible language
- the role of the decoder will be to convert the two-dimensional vector into the output sequence, the English sentence
- It is also built with RNN layers and a dense layer to predict the English word

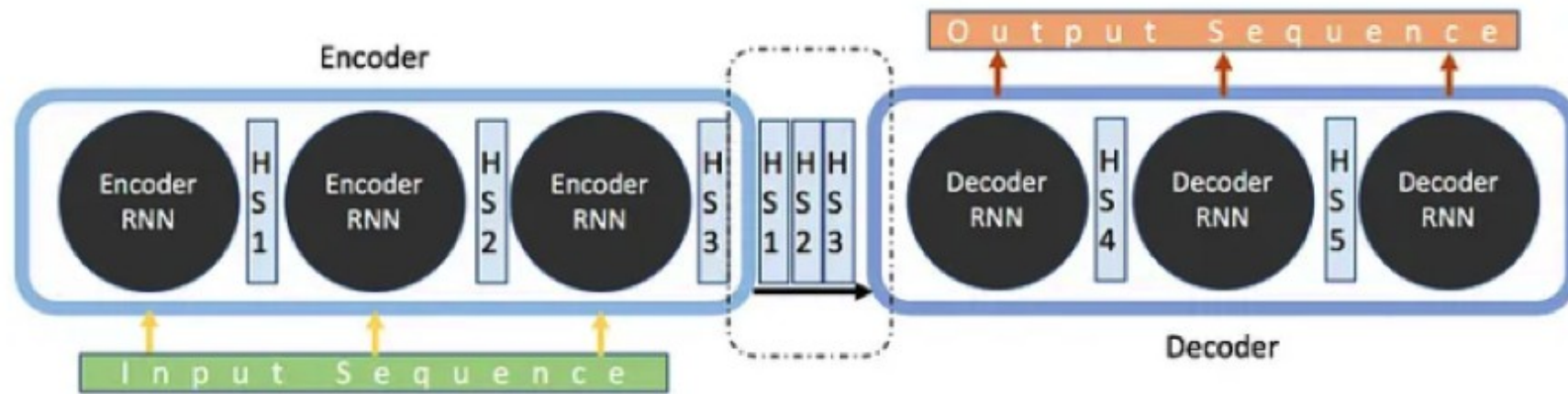
Advantages

- End-to-end training
- Distributed representations allow reuse of learned language features
- Ability to use much larger context than small-n-grams
- Better quality results

Issues so far

- Embeddings like word2vec assume that **collocation** is important but don't make any finer distinction
- Some words in the context are much more important than others
- The words can be related in many ways
 - Proximity of words may not be the best way with a small window
 - Relationships can be far away; they don't need to be in the same sentence
- Meaning unfurls itself slowly and retroactively
- - Loose information very closely
 - Suffer from information and processing bottlenecks
 - Are slow (unparallelizable)
 - have to work word by word
 - no parallelization because they have to work their way through the sequence
 - Are poor at using information that is not very close to the word being translated
 - Hard to interpret the meaning of the state
 - Usually good at producing grammatically correct output but occasionally throw in

Attention



The notion of *attention* is to have a model learn what parts of the context to pay the most attention to -

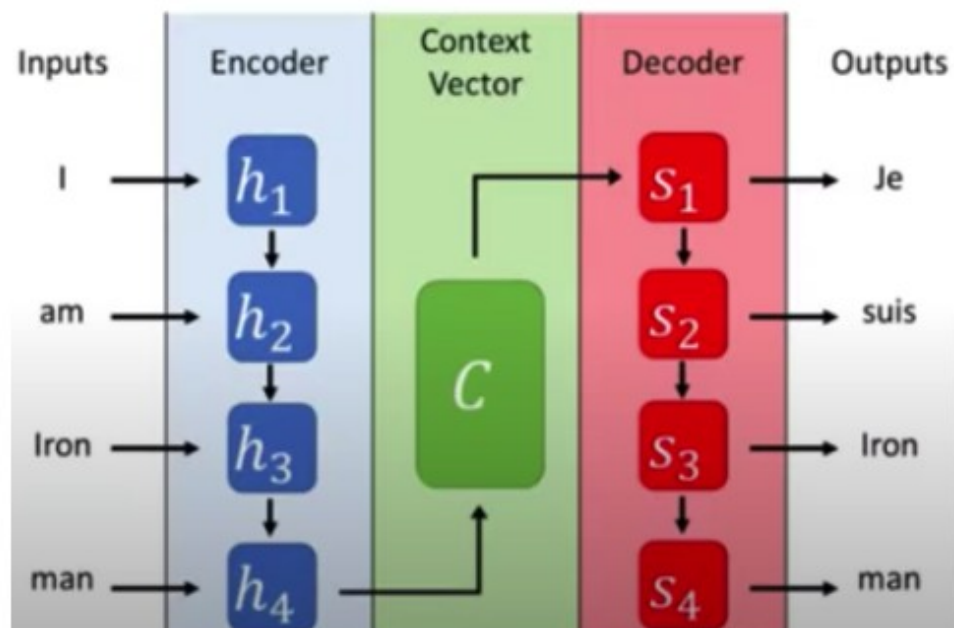
was introduced to improve the performance of the encoder-decoder model for machine translation

- Permit the decoder to utilize the most relevant parts of the input sequence in a flexible manner
 - by a weighted combination of all the encoded input vectors
 - with the most relevant vectors being attributed the highest weights

Why?

- not all the words are equally important
- Stop words have no valuable information content

- different from the classic seq-to-seq model in two ways
 - First, as compared to a simple seq-to-seq model, here, the encoder passes a lot more data to the decoder
 - With seq-to-seq, only the final, hidden state of the encoding part was sent to the decoder
 - but now the encoder passes all the hidden states, even the intermediate ones
- The decoder part:
 - It checks each hidden state that it received as every hidden state of the encoder is mostly associated with a particular word of the input sentence
 - It give each hidden state a score
 - Each score is multiplied by its respective softmax score
 - For amplifying hidden states with high scores and drowning out hidden states with low scores



Problem of long and complex text units (e.g. sentence) and forgetting the earlier parts

Idea:

- give context vector access to the entire input sequence instead of just the last hidden state
- So even if length of sentences increases, then context vector can still capture the content of the sentence
- So we give attention weights to the inputs so the decoder could focus on the relevant positions in the input sequence

Context bottleneck: by the time that information gets to the last hidden layer, earlier important information would get forgotten

- To overcome the context bottleneck we will pass much more information:
 - All of the hidden states from the input sequence, not just the last
- involves creating a **weighted sum of the input features or representations** based on their **relevance to the task**
- **Relevance** is calculated by comparing the input features to a learned query vector, which is specific to the task and learned during training
- The resulting attention weights can then be used to selectively amplify or suppress the contribution of each input feature to the final output

- Attention uses queriable memory similar to a cross between a Map (Python dict) and a search tree
 - We can query using a key and get a weighted average of matching values, weighted by how well the keys match the query
 - Keys, values and queries are all tensors
 - So, rather than trying to keep everything in a single context vector for the sentence we will use a tensor
- The longer the sentence, the bigger the tensor (usually one column per word in the sentence), which solves the capacity and gradient flow problem
- The embeddings of each word are typically a concatenation of the hidden states of a forward and reverse RNN at that position in the sentence

Several different types of attention mechanisms

Potent extensions of RNNs

Self-Attention:

- Rather than a RNN, let's use a feedforward net
- drop the recurrent connection
- keys and values stay the same, but query becomes the current input
- Translation would be word to word

Recurrent Attention:

- Take each word in the sentence and let say do a dot product and see how similar the word is to other words in the sentence
- query is the hidden state from the previous step; calculates attention at each step
- Similar to the Word2vec but amplify the meaning of each word based on the context

Learned Self-Attention:

- something the model learns during training
- If we allow keys and values to be learned we move into a new area:
 - Memory Networks (e.g. Neural Turing Machines)
- The attention result is a sum of the weights of the words
- The weights are a softmax of the similarity of each key and query
- Two common methods of similarity
 - Additive Similarity: Single layer neural network
 - Multiplicative Similarity: Essentially cosine similarity (faster)

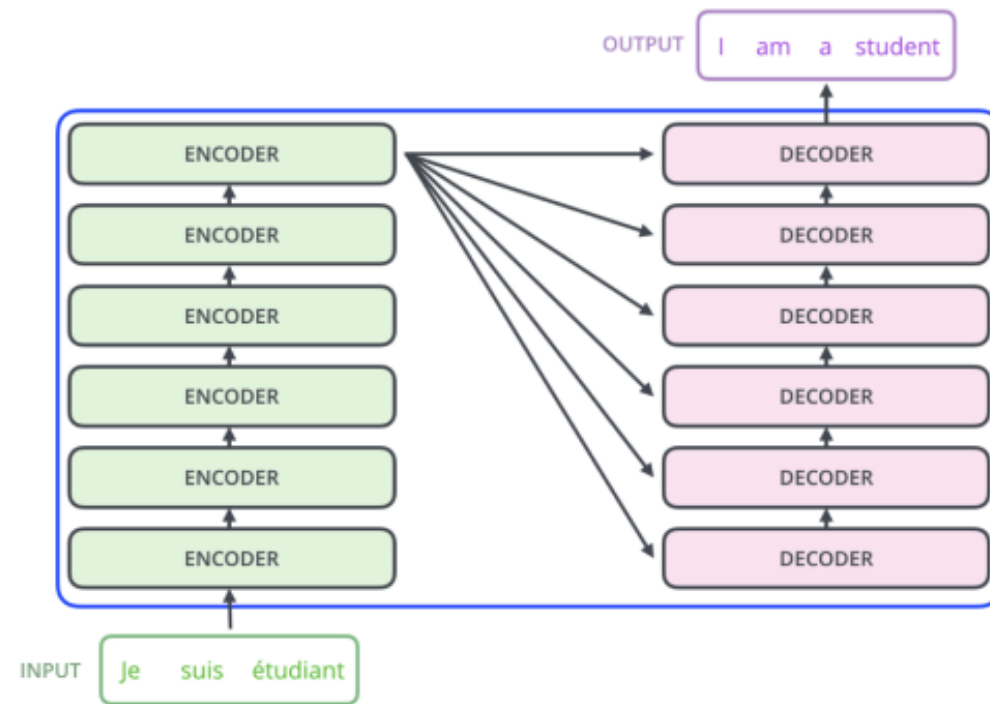
Multi-Head Attention:

- we can use several specialized attention mechanisms
 - Semantic
 - Grammatical
 - Tense
 - Context
- Putting syntax knowledge back may take us to a new level again
- Typically still use LSTM but The Transformer doesn't (next) doesn't use RNN's at all
- Attention Visualization
- You can find the example of the implementation here

Advantages of Attention

- . Better performance
- . Solves the bottleneck problem
- . Helps with the vanishing gradient problem
- . Provides some interpretability
- . Provides soft alignment

stack encoders and decoders on top of each other to build a transformer



- Whereas RNN's must proceed step-by-step (word by word), the Transformer uses only a small (configurable) number of steps
- At each step it applies a self-attention mechanism to weigh the relative strength of the relationships the other words in the sentence have to the word being processed
- A weighted average of all of the current words' representations are used to produce an updated representation for the target word

BERT

Bidirectional Encoder Representations from Transformers

- A transformer based method of pre-training language representations
- relies on an attention mechanism
- BERT-Base, which includes 110 million parameters
- BERT-Large, which has 340 million parameters

