Business Context:

In this project, we will witness and understand the powerful capabilities of a **State-of-The-Art Transformer** model called **BERT**:

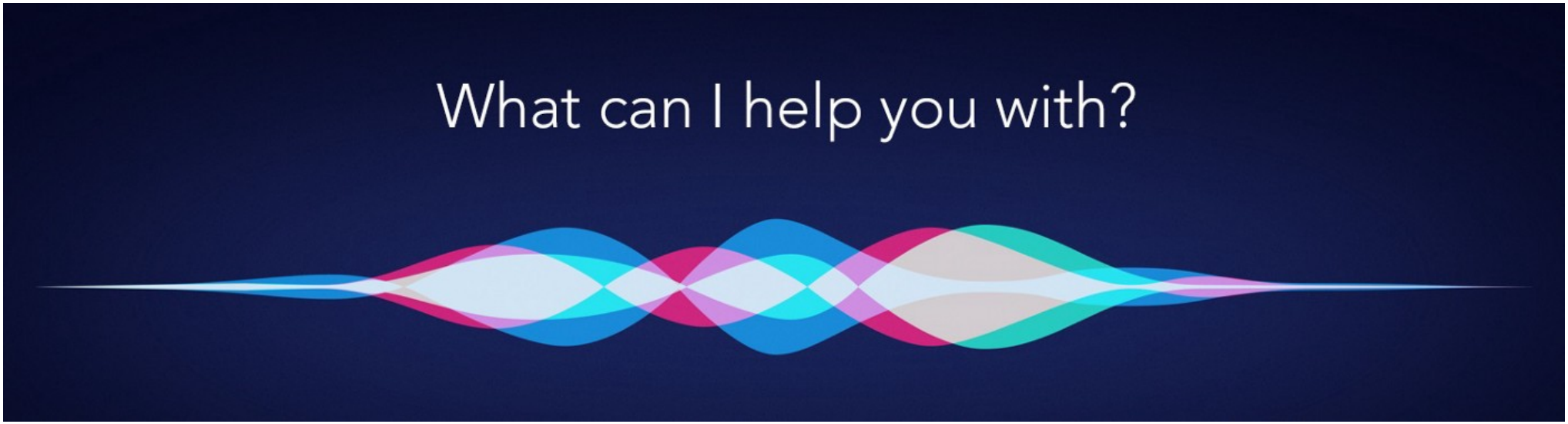
- We will first cover the requirements and history of NLP, Text classification using **Bag-of-Words(BOW)** & **TF-IDF**.
- Next, we move on to word embeddings where we discuss on popular architectures like **Word2Vec**, **Glove**.....
- Post that we move ahead with the shortcomings of these architectures in long sequences and bi-directional semantic understanding of language and discuss on **popular RNN models** as an improvement on the same.
- Further, we move ahead with the **Transformer architecture** and how the concept of **self attention with encoder & decoder** has helped surpass performance metrics set by previous architectures.
- Finally, we will understand the **architecture of BERT** in detail and use it for **Multi-Class text classification** on a very large text corpus.

References:

- Google Images
- Attention research paper -> https://arxiv.org/abs/1706.03762 (https://arxiv.org/abs/1706.03762)
- BERT research paper -> https://arxiv.org/abs/1810.04805 (https://arxiv.org/abs/1810.04805)
- Wikipedia, Google

Introduction:

- Natural Language Processing(NLP) is a field of artificial intelligence that studies the interaction between computers and human languages, especially how to process and analyze large amounts of natural language data through computer programming.

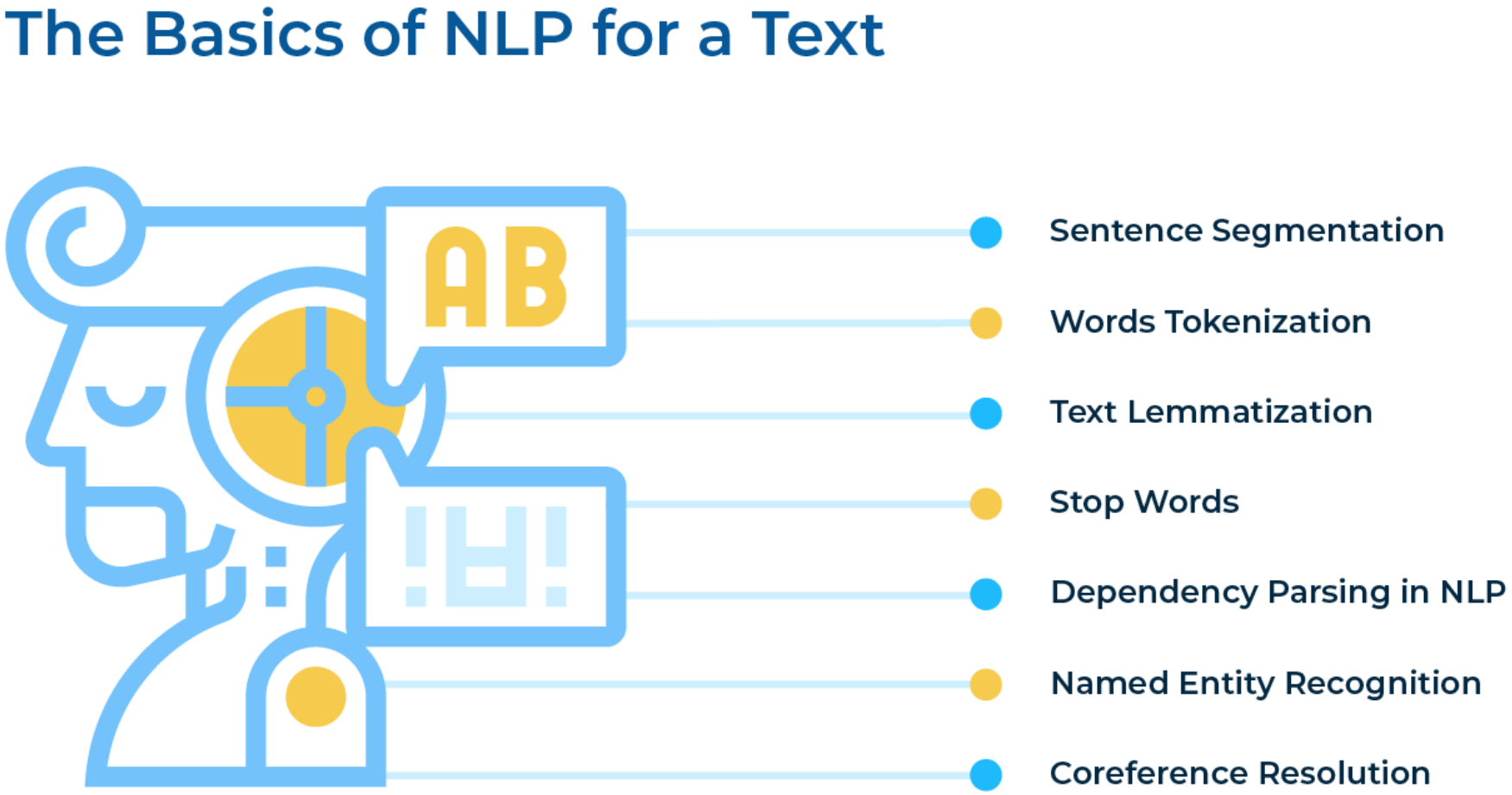


- NLP enables computers to understand natural language as humans do. Whether the language is spoken or written, natural language processing uses artificial intelligence to take real-world input, process it, and make sense of it in a way a computer can understand.

NLP Applications:

NLP is most commonly used for the below use cases:

- Classify text data based on topic/content
- Perform sentiment analysis on reviews
- Chatbots/Question Answering
- Machine Translation
- Named Entity Recognition and so on....



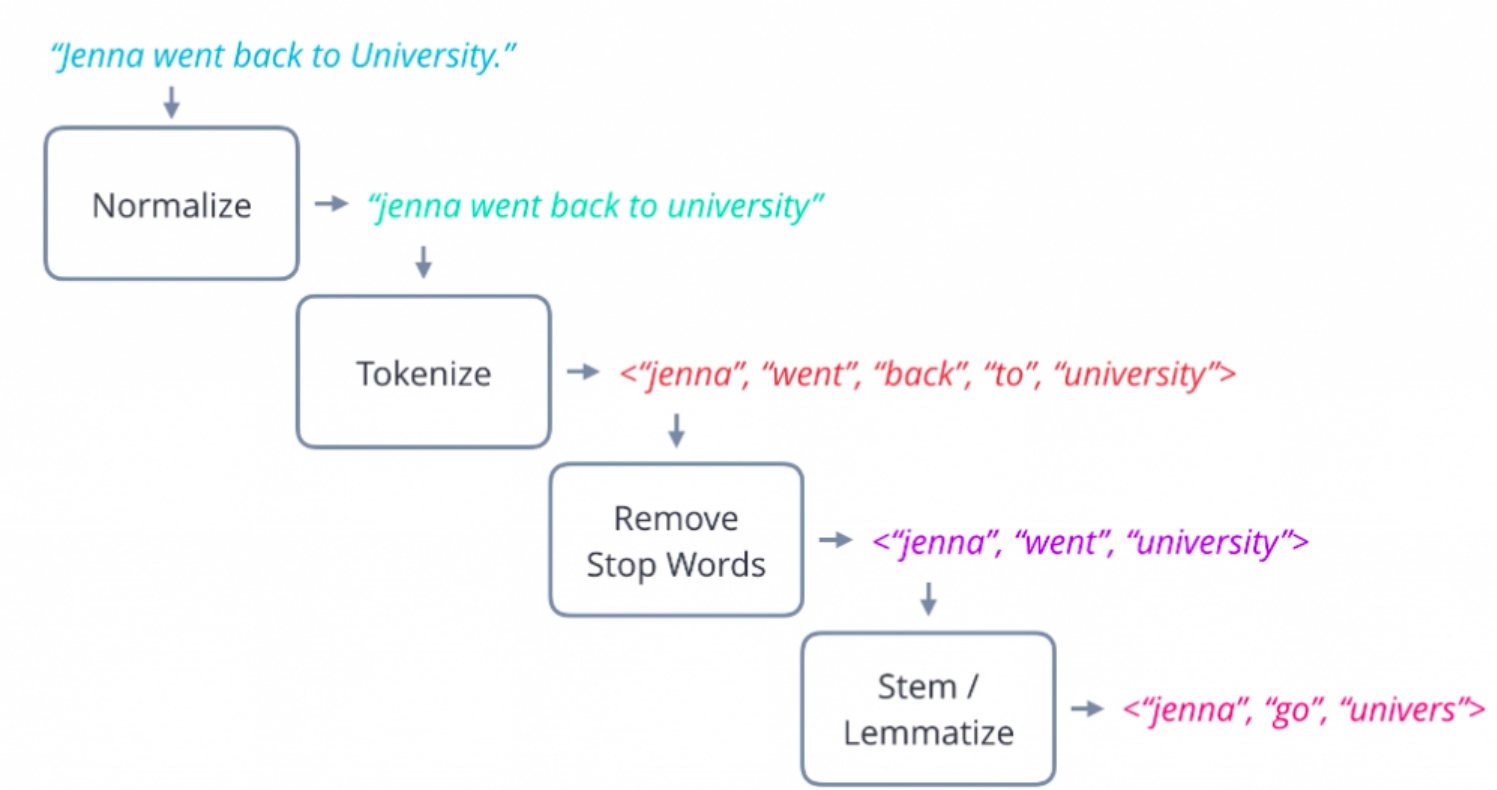
- There are a variety of techniques to extract information from raw text data and use it to train models. Let's discuss this in detail.

Text Preprocessing & Word Vectors:

- Computers are unable to understand the concepts of words. In order for the computer to understand and interpret natural language, a mechanism for representing text is required.

- The standard mechanism for text representation are word vectors where words or phrases from a given language vocabulary are mapped to vectors of real numbers. Or in other words, the text data is converted into a meaningful representation of numbers for the computers to process and interpret.
- Before any vectorized text data is fed to a machine learning/deep learning algorithm, there are certain preprocessing steps which are followed. Below are some general steps which are performed as per the requirement:

1. Punctuation & Stop Word removal
2. Stemming
3. Lemmatization
4. Parts of Speech(POS) tagging.



- Once the text preprocessing steps are completed, we then move on to Numerical/Vector representation of text data.

Bag of Words(BOW):

- The Bag of Words(BOW) model has one of the most simple concepts: build a vocabulary from the document corpus, and count the number of times a word appears in each document.
- Bag of Words or BoW vector representations are the most commonly used traditional vector representation. Each word or n-gram is linked to a vector index and marked as 0 or 1 depending on whether it occurs in a given document.
- BoW representations are often used in methods of document classification where the frequency of each word, bi-word or tri-word is a useful feature for training classifiers.

	the	red	dog	cat	eats	food
1. the red dog →	1	1	1	0	0	0
2. cat eats dog →	0	0	1	1	1	0
3. dog eats food →	0	0	1	0	1	1
4. red cat eats →	0	1	0	1	1	0

Advantages of BOW:

- BOW is widely used owing to its simplicity.
- Serves as a base model to get an idea of the performance metrics before proceeding to better word embeddings.
- Particularly helpful when we are working on few documents and they are very domain-specific/niche domains like sentiment analysis on Political news.

Limitations of BOW:

- Does not encode any information with regards to the meaning of a given word.
- eg- The sentences "I love playing carrom and hate hockey" and "I love playing hockey and hate carrom" will result in similar vectorized representations although both sentences carry different meanings.
- For large documents, the resultant vectors will be of high dimension. The more files, the larger the vocabulary, so the feature matrix will be a huge sparse matrix.

TF-IDF:

- Term frequency–inverse document frequency(TF-IDF) is a methodology used to reflect how important a word/n-gram is to a document in a collection or corpus.
- It helps to provide some weightage to a given word based on the context it occurs.

- The tf–idf value increases proportionally to the number of times a word appears in a document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

sentence 1	earth is the third planet from the sun
sentence 2	Jupiter is the largest planet

Word	TF (Sentence 1)	TF (Sentence 2)	IDF	TF*IDF (sentence 1)	TF*IDF (Sentence 2)
earth	1/8	0	log(2/1)=0	0.0375	0
is	1/8	1/5	log(2/2)=0	0	0
the	2/8	1/5	log(2/2)=0	0	0
third	1/8	0	log(2/1)=0.3	0.0375	0
planet	1/8	1/5	log(2/2)=0	0	0
from	0	0	log(2/1)=0.3	0	0
sun	1/8	0	log(2/1)=0.3	0.0375	0
largest	0	1/5	log(2/1)=0.3	0	0.06
Jupiter	0	1/5	log(2/1)=0.3	0	0.06

Advantages of TF-IDF:

- Easy way to extract the most descriptive keywords in a document.
- Helps measure content uniqueness and relevance.

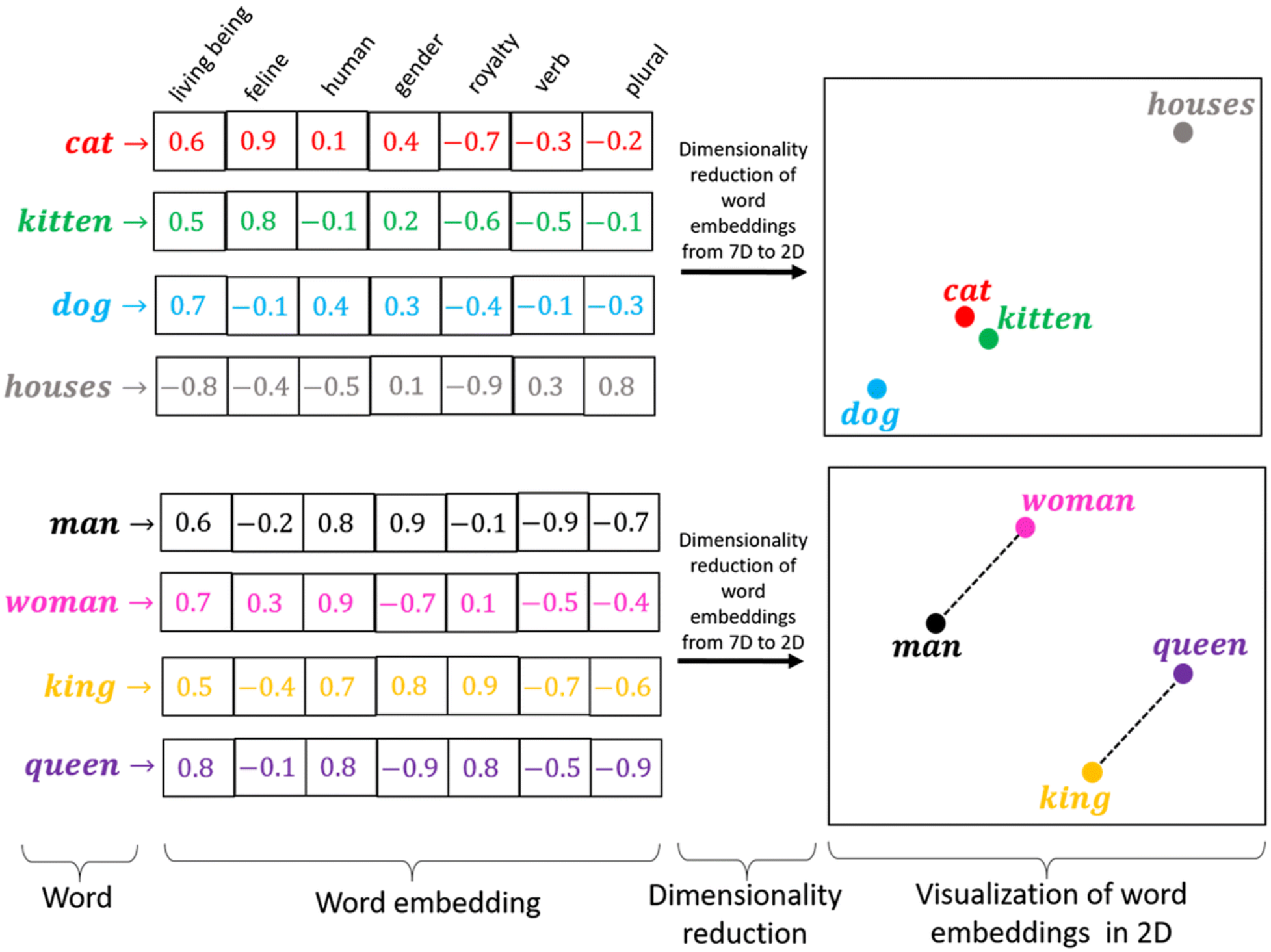
Disadvantages of TF-IDF:

- Unable to capture the word meaning
- Cannot capture text position, semantics or co-occurences

Hence **Word frequency/Tf-Idf** may not necessarily the best way to represent text. Let's try to understand now how **Word Embeddings** changed this scenario.

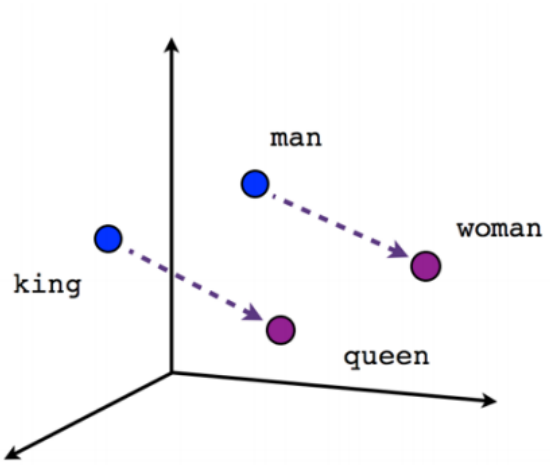
Word Embeddings:

- Word Embeddings are fixed length, dense and distributed vector representations for words.
- The task of representing words and documents as vectors serve many useful operations (e.g. addition, subtraction, distance measures, etc) and lend themselves well to be used in many Machine Learning (ML) algorithms and strategies.
- A very important part of natural language-based solutions is the study of language models, where the models focus mainly on predicting the next word given a number of previous words. eg- Speech Recognition Software.

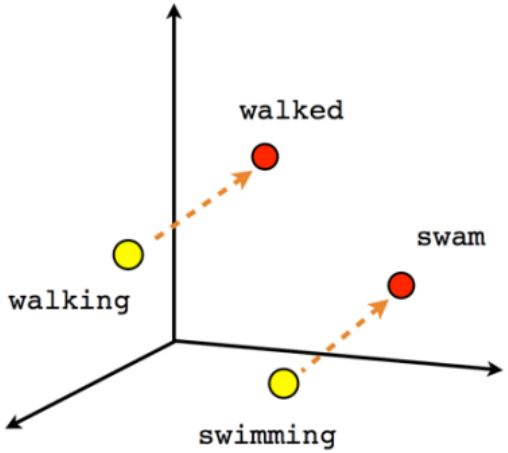

living being
feline
human
gender
royalty
verb
plural
cat → 0.6 0.9 0.1 0.4 −0.7 −0.3 −0.2
kitten → 0.5 0.8 −0.1 0.2 −0.6 −0.5 −0.1
dog → 0.7 −0.1 0.4 0.3 −0.4 −0.1 −0.3
houses → −0.8 −0.4 −0.5 0.1 −0.9 0.3 0.8
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
man → 0.6 −0.2 0.8 0.9 −0.1 −0.9 −0.7
woman → 0.7 0.3 0.9 −0.7 0.1 −0.5 −0.4
king → 0.5 −0.4 0.7 0.8 0.9 −0.7 −0.6
queen → 0.8 −0.1 0.8 −0.9 0.8 −0.5 −0.9
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Word-2-Vec:

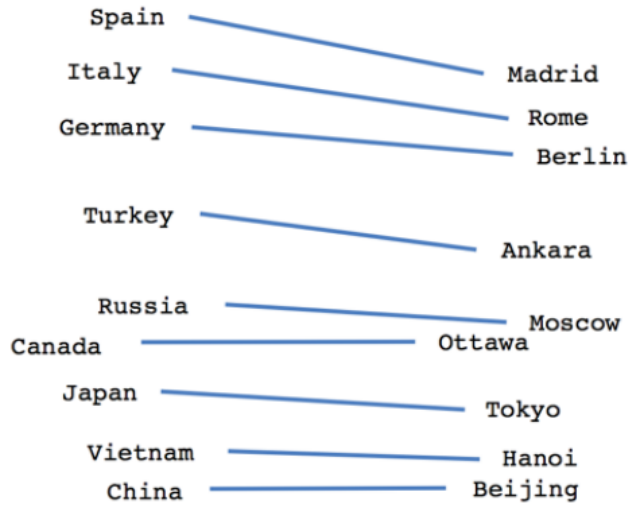
- Word2Vec is a shallow, two-layer neural networks which is trained to reconstruct contexts of words.
- It takes as its input a large corpus of words and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.
- Word vectors are positioned in such a way in the vector space, words sharing common contexts in the corpus are located in close proximity to one another in the space.



Male-Female



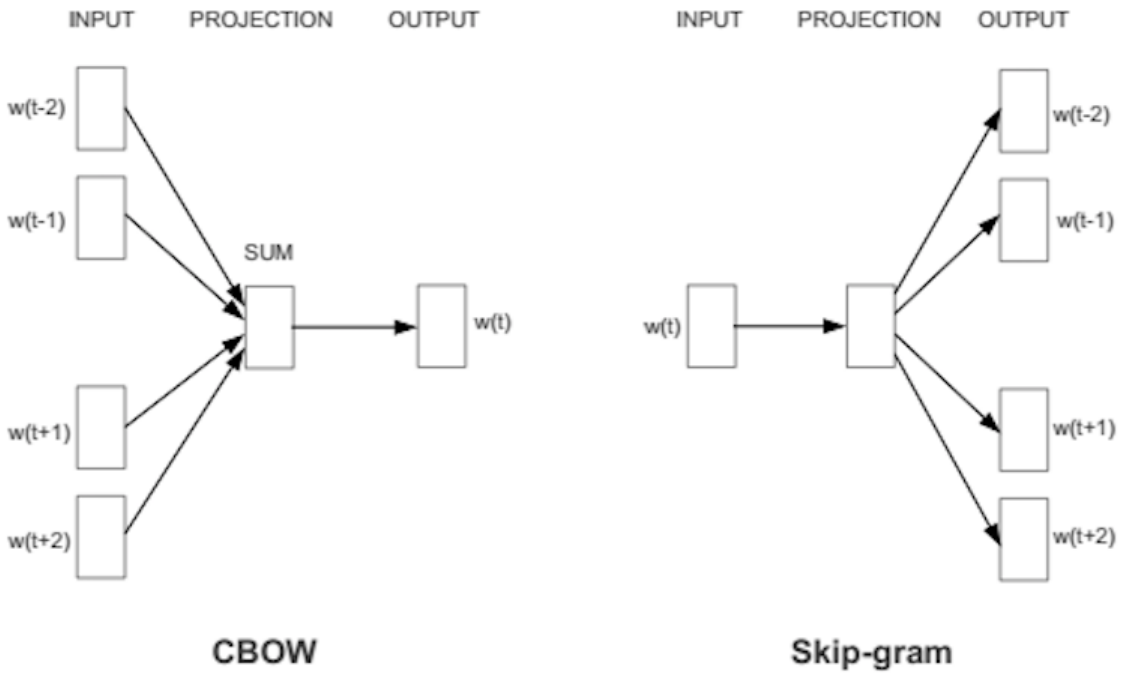
Verb tense



Country-Capital

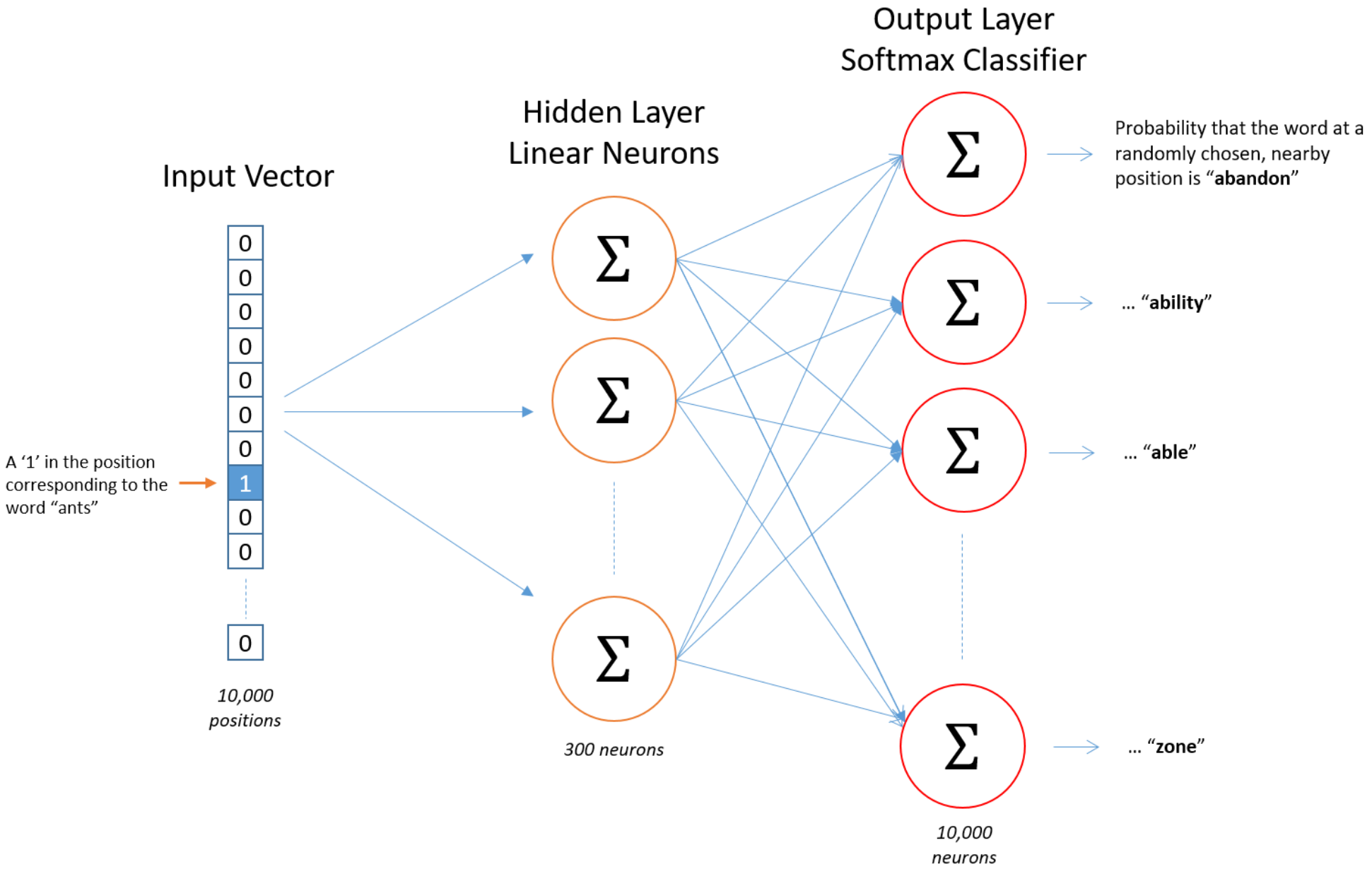
There are two main Word2Vec architectures used to produce a distributed representation of words:

- Continuous bag-of-words (CBOW) - Predicts target words from the surrounding context words.
- eg- Context ->['George is sleeping in his'], Target ->['room']
- Skip-gram - Predicts surrounding context words from the target words (inverse of CBOW).



Architecture:

- In Word2Vec architecture, you take a large input vector, compress it down to a smaller dense vector and then instead of decompressing it back to the original input vector, you output probabilities of target words.
- Words are fed as one-hot vectors, which is basically a vector of the same length as the vocabulary, filled with zeros except at the index that represents the word we want to represent, which is assigned “1”.
- The hidden layer is a standard fully-connected (Dense) layer whose weights are the word embeddings.
- The output layer outputs probabilities for the target words from the vocabulary.



- The network's output is a probability distribution of target words, which may not necessarily be a one-hot vector like the labels.
- Word2Vec generates a vector space containing each unique word in the corpus, usually several hundred dimensions, so that words in the corpus that have a common context will be close to each other in the vector space.
- The word vector dimension are generally set to 300.

Advantages of Word2Vec:

- Able to capture multiple different degrees of similarity between words.
- eg - patterns such as “Man is to King as Woman is to Queen”.
- Not very resource intensive compared to other embeddings like Glove..

Limitations of Word2Vec:

- Does not consider statistical information regarding word co-occurrences.
- Context is small.

Glove:

- Glove works similarly as Word2Vec.
- While you can see above that Word2Vec is a "predictive" model that predicts context given word, Glove learns by constructing a co-occurrence matrix that basically count how frequently a word appears in a context.
- It forms a gigantic matrix in turn, which is further factorized to achieve a lower-dimension representation.
- Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase linear substructures of the word vector space.

Let's try to understand this with an example:

1. I like deep learning.
2. I enjoy flying.
3. I like NLP.

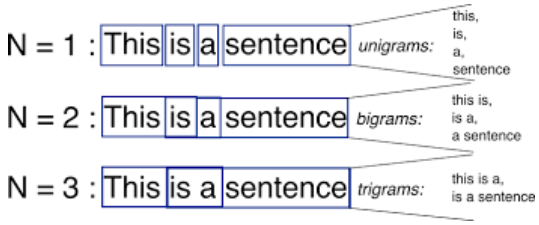
- We obtain the co-occurence matrix for the above sentences as shown below:

$$
X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array}
\begin{array}{c}
\begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\
\begin{bmatrix}
0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\end{array}
$$

- Once we have the co-occurrence matrix filled we can plot its results into a multi-dimensional space. Since 'deep' and 'NLP' share the same co-occurrence values, they would be placed in the same place.
- The relationship of word vectors can be examined by studying the ratio of their co-occurrence probabilities with various probe words further.

Fast-Text:

- FastText, builds on Word2Vec by learning vector representations for each word and the n-grams found within each word. The values of the representations are then averaged into one vector at each training step.
- This adds a lot of additional computation to training but also enables word embeddings to encode sub-word information. FastText vectors have been shown to be more accurate than Word2Vec vectors by a number of different measures.



- The logic behind FastText working lies in the fact that languages that rely heavily on morphology and compositional word-building (such as Turkish, Finnish and other highly inflexional languages) have some information encoded in the word parts themselves, which can be used to help generalize to unseen words.

Embedding Models Comparison:

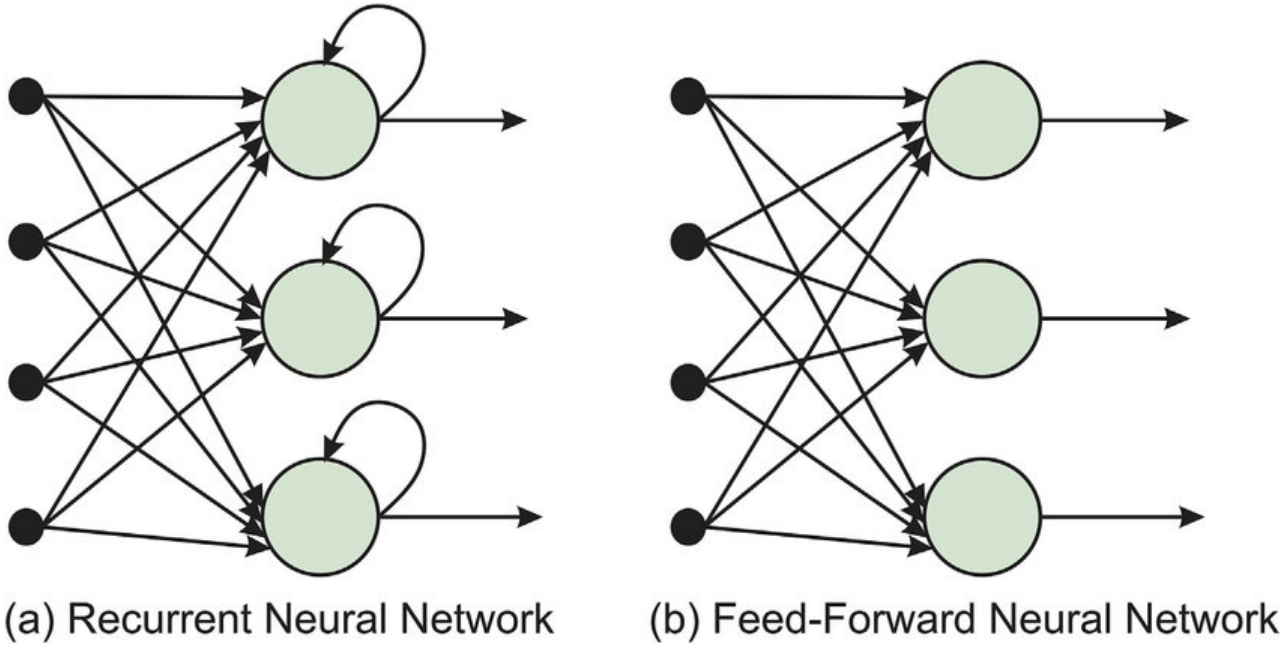
- Word2Vec takes texts as training data for a neural network. The resulting embedding captures whether words appear in similar contexts.
- GloVe focuses on words co-occurrences over the whole corpus. Its embeddings relate to the probabilities that two words appear together.
- FastText improves on Word2Vec by taking word parts into account, too. This trick enables training of embeddings on smaller datasets and generalization to unknown words.

Limitations of Word-Embeddings:

- Use of shallow Language Models. This limits the amount of information they could capture. Improvement can be done using deeper and more complex language models(LSTMs).
- Does not take the context of the word into account. Embeddings like Word2Vec will give the same vector for the same word regardless of the context.
- Problem of contextual Awareness remains unsolved.

Recurrent Neural Network:

- Recurrent neural networks (RNN) are a class of neural networks that are helpful in modeling sequence data.
- Otutput from previous step are fed as input into the current step.
- In traditional neural networks, all the inputs and outputs are independent of each other. Comparatively, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. RNNs solved this problem with a Hidden Layer having some Hidden State data which helps remember some information about a sequence.



Un-rolled in time, the Recurrent Neural Network looks like this:

Advantages:

- RNN helps remember information through time. It is useful in scenarios like time series prediction because of the feature to remember previous inputs as well.
- Great when it comes to short contexts

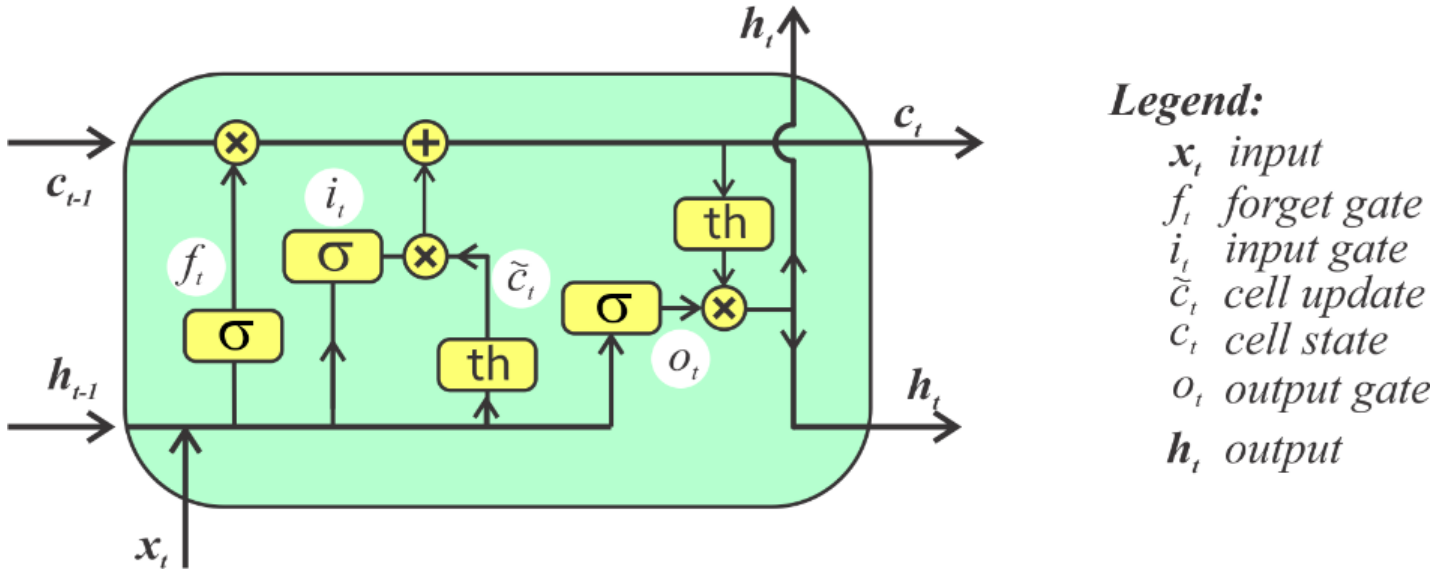
Disadvantages:

- Vanishing & Exploding gradient problems are very frequent.
- Cannot process very long sequences.

LSTM as improvement over RNNs:

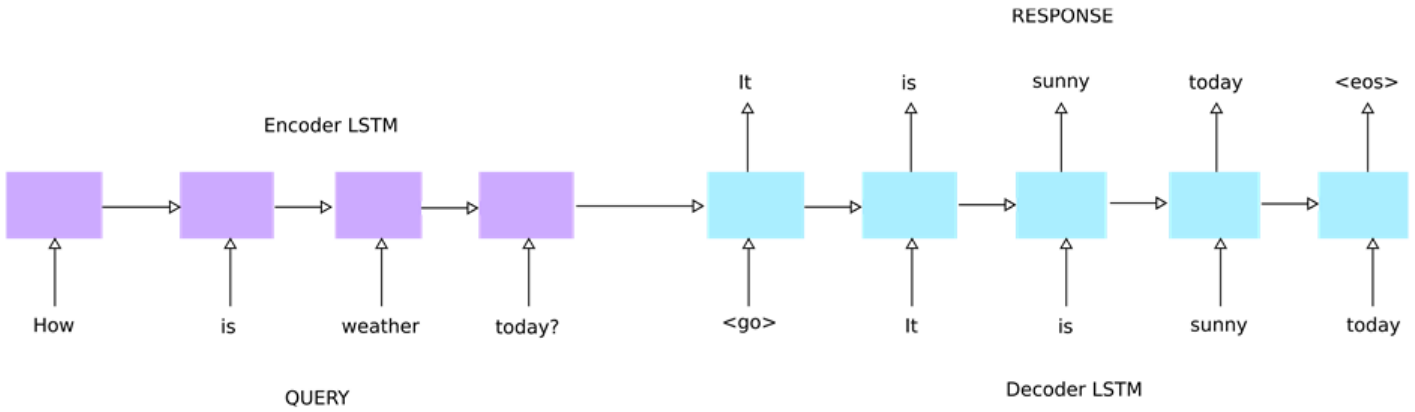
LSTMs improve over RNNs by keeping information arranged as follows:

- The previous cell state (i.e. the information that was present in the memory after the previous time step)
- The previous hidden state (i.e. this is the same as the output of the previous cell)
- The input at the current time step (i.e. the new information that is being fed in at that moment)
- The final output is provided by several operations involving the the 3 gates and ceell states as shown in the figure below:



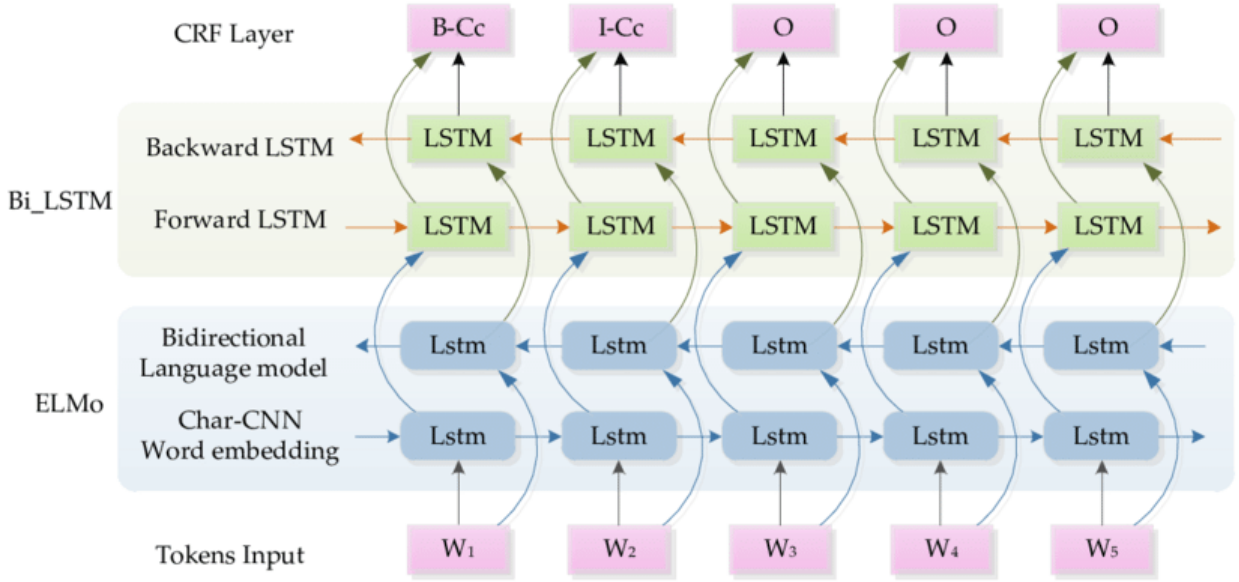
Sequence to Sequence learning with LSTMs:

- Sequence-to-sequence learning (Seq2Seq) is about training models to convert sequences from one domain (e.g. sentences in English) to sequences in another domain (e.g. the same sentences translated to French).
- This can be used for machine translation or for free-from question answering.
- Generally, input sequences and output sequences have different lengths (e.g. for machine translation task) and the entire input sequence is required to start predicting the target.
- A LSTM layer/stack acts as "encoder": it processes the input sequence and returns its own internal state. The state is preserved here which serves as "context", or "conditioning", of the decoder in the next step.
- A LSTM layer/stack acts as "decoder": it is trained to predict the next characters of the target sequence, given previous characters of the target sequence, offset by one timestep in the future.



Dawn of ELMO:

- ELMo has been presented as a novel way to represent words in vectors or embeddings. These word embeddings are helpful in achieving state-of-the-art (SOTA) results in several NLP tasks.
- ELMo was the solution to the problem of Polysemy – same words having different meanings based on their context.
- Instead of training shallow feed-forward networks (Word2vec), complex Bi-directional LSTM architectures were used to train word embeddings, resulting in same word having multiple embeddings based on the context.



- ELMo word vectors are computed on top of a two-layer bidirectional language model (BILM). 2 layers are stacked together in this model & each layer has 2 passes — forward and backward.
- Character-level convolutional neural network (CNN) is used to represent words of a text string into raw word vectors.
- These raw word vectors act as inputs to the first layer of BILM.
- The forward pass contains information about a certain word and the context (other words) before that word.
- The backward pass contains information about the word and the context after it.
- This pair of information, from both the passes helps to form the intermediate word vectors.
- These intermediate word vectors are then fed into the next layer of BILM.

- The final output from ELMO is the weighted sum of the raw word vectors and the 2 intermediate word vectors.

ELMO key difference with Word-2-Vec & Glove:

- Unlike traditional word embeddings such as Word-2-Vec & Glove, the vector assigned to a token or word in ELMO is actually a function of the entire sentence containing that word. Therefore, the same word can have different word vectors under different contexts.
- **Polysemy example**:

1. I have thoroughly read the newspaper yesterday.
2. Can you please go ahead and read the document now?

- In the above sentences, the word "read" has different meanings based on the context.

ULMFIT:

- This model could train language models that could be fine-tuned to provide excellent results even with fewer data on a variety of document classification tasks.
- With ELMO & ULMFIT, Transfer Learning in NLP was set as a benchmark rule to get better
- Transfer Learning in NLP now involved Pre-Training & Fine-Tuning as the 2 key steps.

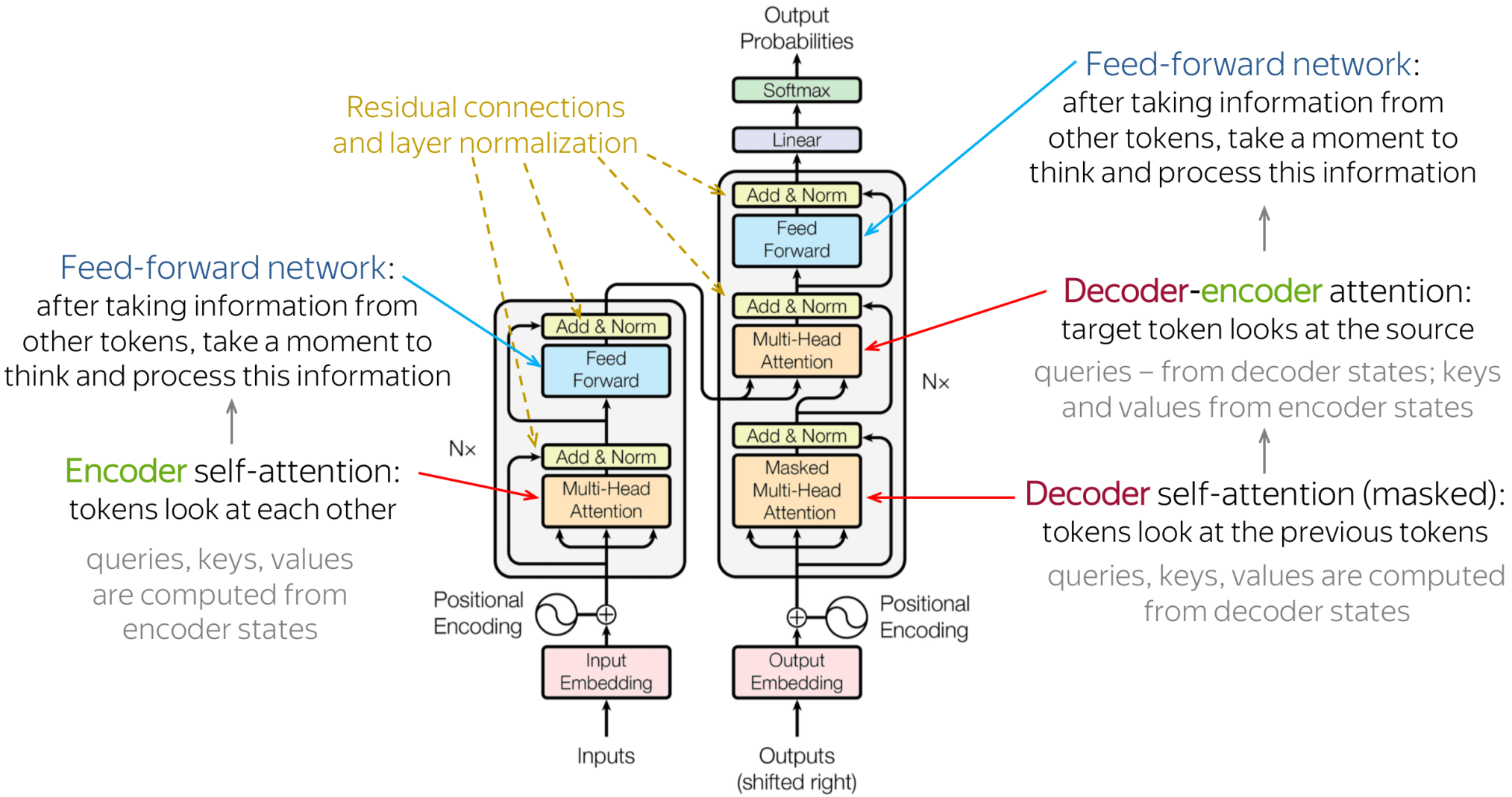
Open AI's GPT Model:

- OpenAI's GPT replaced the LSTM-based architecture for Language Modeling with a **Transformer-based architecture**.
- The GPT model could be fine-tuned to multiple NLP tasks beyond document classification, such as common sense reasoning, semantic similarity, and reading comprehension.
- GPT also emphasized the importance of the **Transformer** framework, which has a simpler architecture and can train faster than an LSTM-based model.
- GPT was able to learn complex patterns in the data by using **Attention mechanism**.

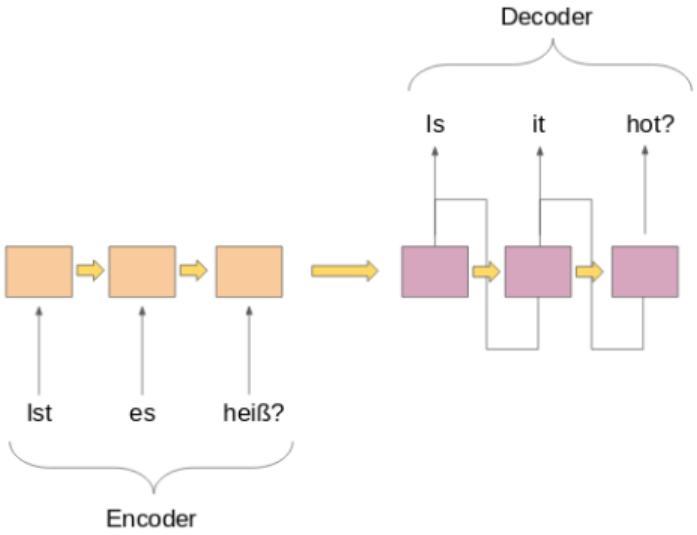
With several high level terms & concepts like **Transformer** & **Attention** coming into the picture, let's try to understand these concepts in detail further.

Transformers:

- The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.
- The Transformer was proposed in the paper Attention Is All You Need -> https://arxiv.org/abs/1706.03762 (https://arxiv.org/abs/1706.03762)
- This architecture relied on self-attention to compute representations of its input and output without using sequence-aligned RNNs/LSTMs.

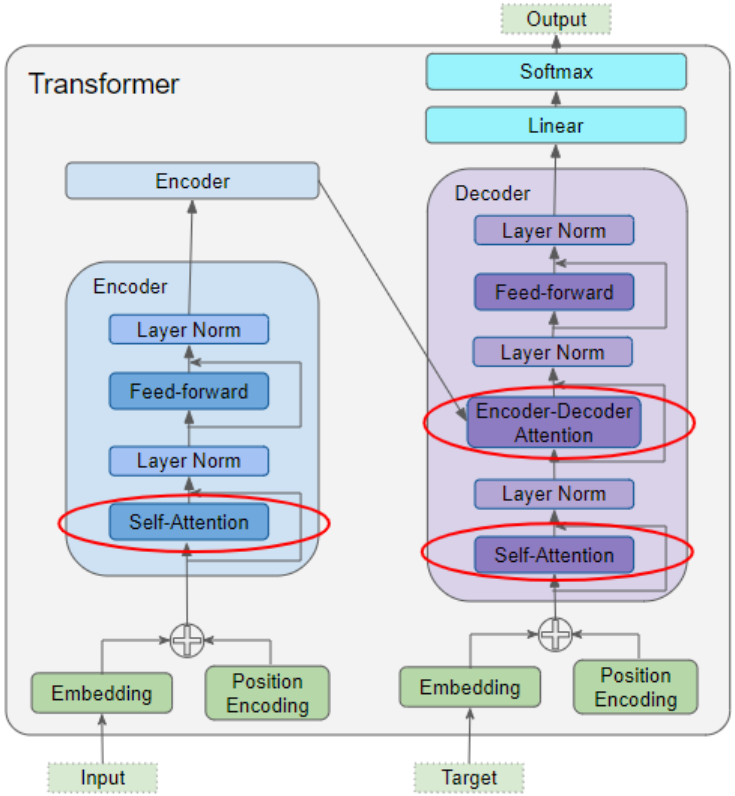


- The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other, both having same no of units.
- The role of an encoder layer is to encode a sentence of a particular language into a numerical form using the attention mechanism, while the decoder aims to use the encoded information from the encoder layers to give the translation in a different language.
- The number of encoder and decoder units is a hyperparameter. In the research paper mentioned above, 6 encoders and decoders have been used.



Encoder-Decoder Architecture:

- The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position wise fully connected feed-forward network. A residual connection is used around each of the two sub-layers, followed by layer normalization.
- The output of each sub-layer is LayerNorm(x + Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. To facilitate theresidual connections, all sub-layers in the model, as well as the embeddinglayers, produce outputs of dimension dmodel = 512.
- The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.
- Similar to the encoder, residual connections are used around each of the sub-layers, followed by layer normalization. Also, the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions.
- This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.
- The word embeddings of the input sequence are passed to the first encoder. These are then transformed and propagated to the next encoder
- The output from the last encoder in the encoder-stack is passed to all the decoders in the decoder-stack.
- In addition to the self-attention and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.



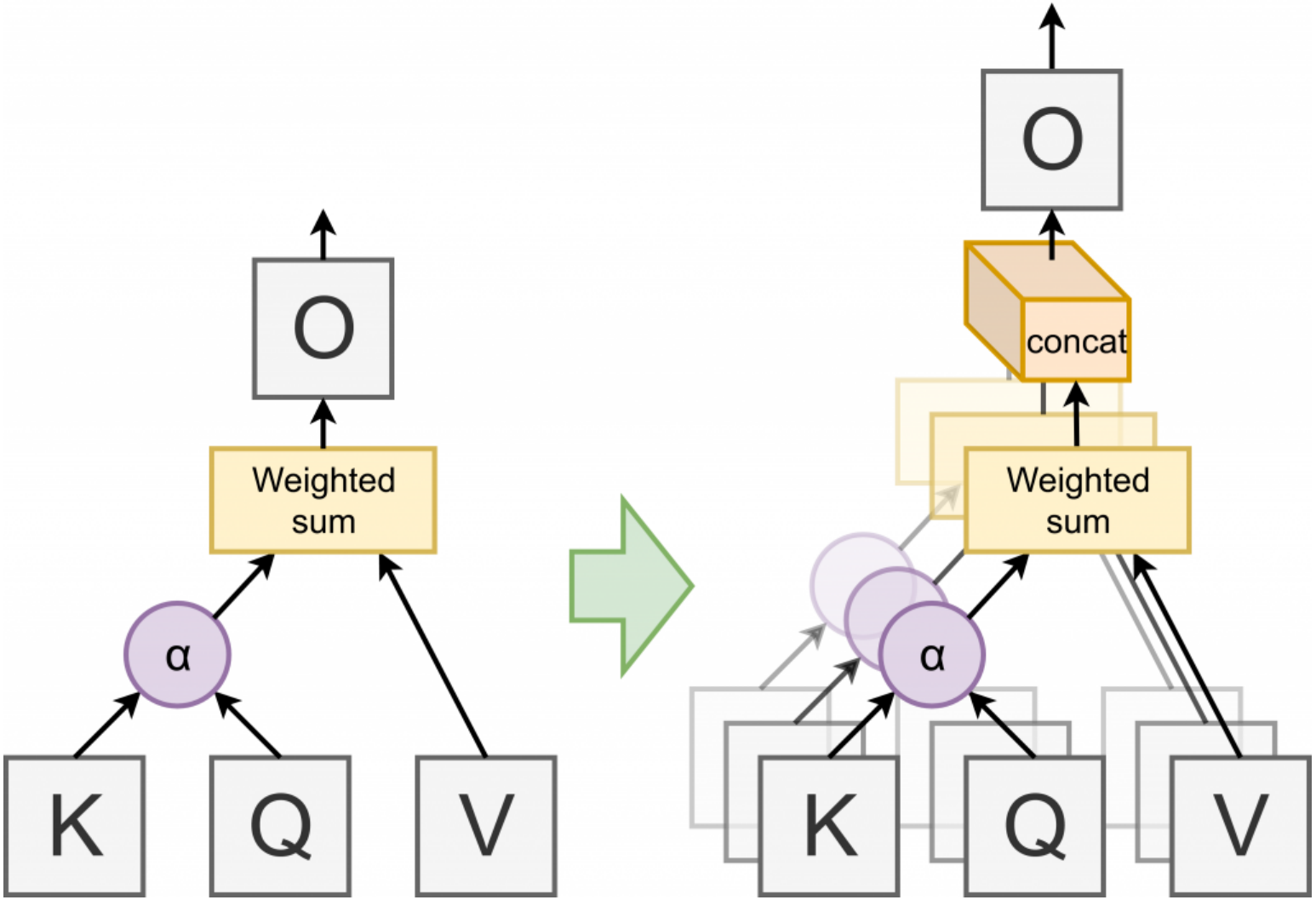
Self Attention:

- Self-attention is a new technique employed in this architecture.
- Instead of looking at prior hidden vectors when considering a word embedding, self-attention is a weighted combination of all other word embeddings (including those that appear later in the sentence as shown in the figure below:

- In the figure above, how does a model determine what is **it** referring to the street or to the animal? With the help of sel-attention, the model tries to associate the word “it” with “animal” in the same sentence.
- Self-attention allows the model to look at the other words in the input sequence to get a better understanding of a certain word in the sequence.

How self-attention is implemented:

- An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

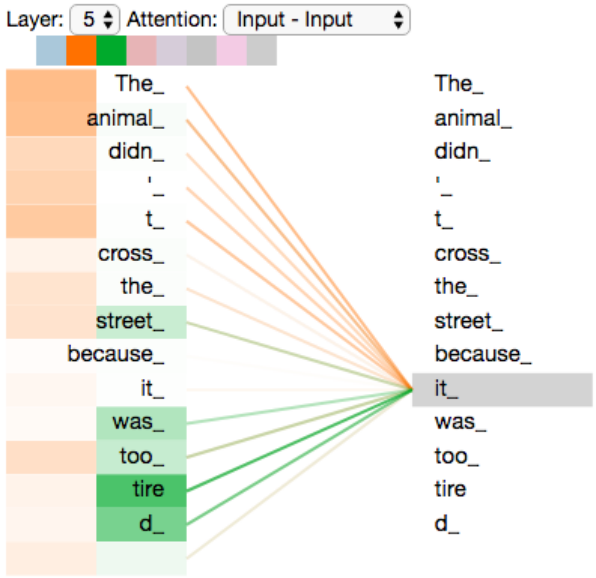


- The word embedding is transformed into three separate matrices -- queries, keys, and values -- via multiplication of the word embedding against three matrices with learned weights.

- For each word embedding, the following steps are performed:
- Calculation of dot product of the current word's query vector with that word's key vector. e.g. for w1 in a sentence with just two words in it we would calculate: (q1.k1, q1.k2).
- The query and key vectors are turned into a single value via the dot product.
- They are then adjusted via normalization and softmax.
- Next, each word embedding's value vector is multiplied by its softmax & the final result is obtained by summation.

Multi-Headed Attention:

- By repeating the self-attention mechanism multiple times the model can learn to seperate different kinds of useful semantic information onto different channels/heads:



Finally BERT makes its entry:

So far, we have learned and understood the **Transformer Architecture** and seen how different models have helped in achieving better results in multiple NLP tasks. Let us now explore **BERT** in detail:

- BERT is based on the **Transformer Architecture** as discussed above.
- It stands for **Bidirectional Encoder Representations from Transformers**. Unlike other language representation models, BERT is designed to pretrain **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 finetuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as classification, question answering and language inference, without substantial task specific architecture modifications.
- BERT is based on the transformer achitecture as described above
- BERT is pre-trained on the **BooksCorpus (800M words)** and **English Wikipedia (2,500M words)**.
- BERT Transformer uses bidirectional self-attention, while it's predecessor the GPT Transformer uses constrained self-attention where every token can only attend to context to its left.
- BERT is a “deeply bidirectional” model. Bidirectional means that BERT learns information from both the left and the right side of a token’s context during the training phase.

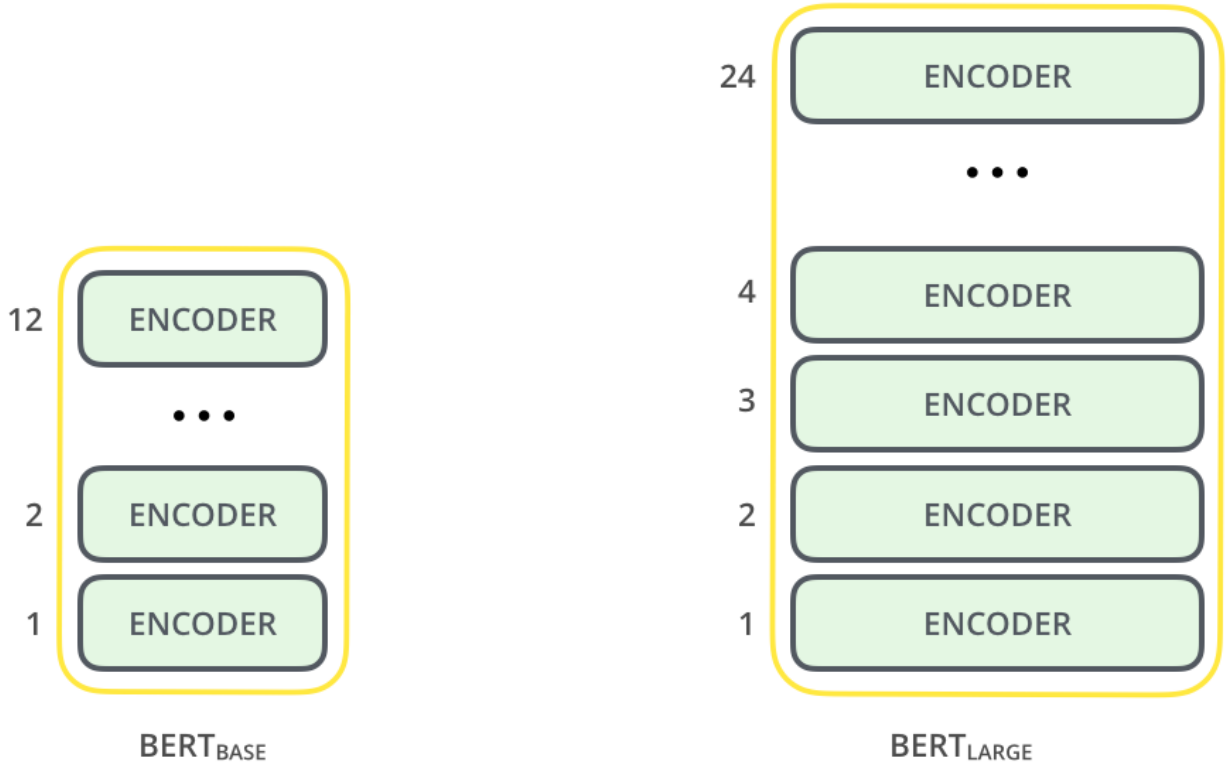
See the example below:

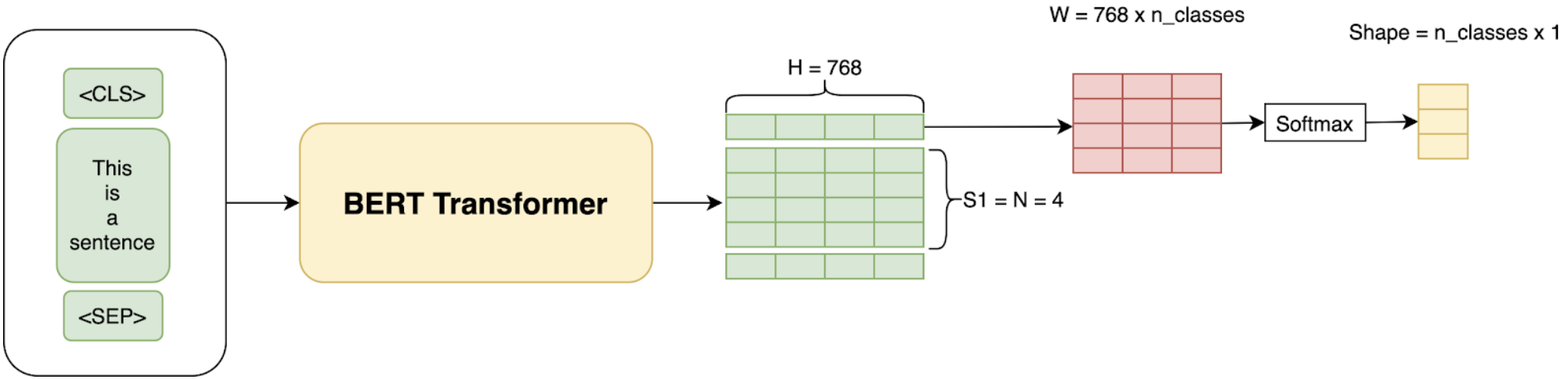
1. **He and his friends went to sit near the river bank and enjoy the sunset**.
2. **Tom needs to go to the bank by saturday to claim his lottery amount awarded as a cheque**.

In the above case, if we try to predict the nature of the word **“bank”** by only taking either the left or the right context, then we will be making an error in at least one of the two given examples. A way to deal with this is to consider both the left and the right context before making a prediction. This is exactly what is done by BERT.

BERT Framework and Architecture:

- BERT consists of stacks of encoder-decoder layered upon one another as shown in the diagram below.
- The number of layers in the Transformer Block is denoted as L, the hidden size as H, and the number of self-attention heads as A
- The 2 model with their respective configurations are:
- BERTBASE (L=12, H=768, A=12, Total Parameters=110M)
- BERTLARGE (L=24, H=1024, A=16, Total Parameters=340M).

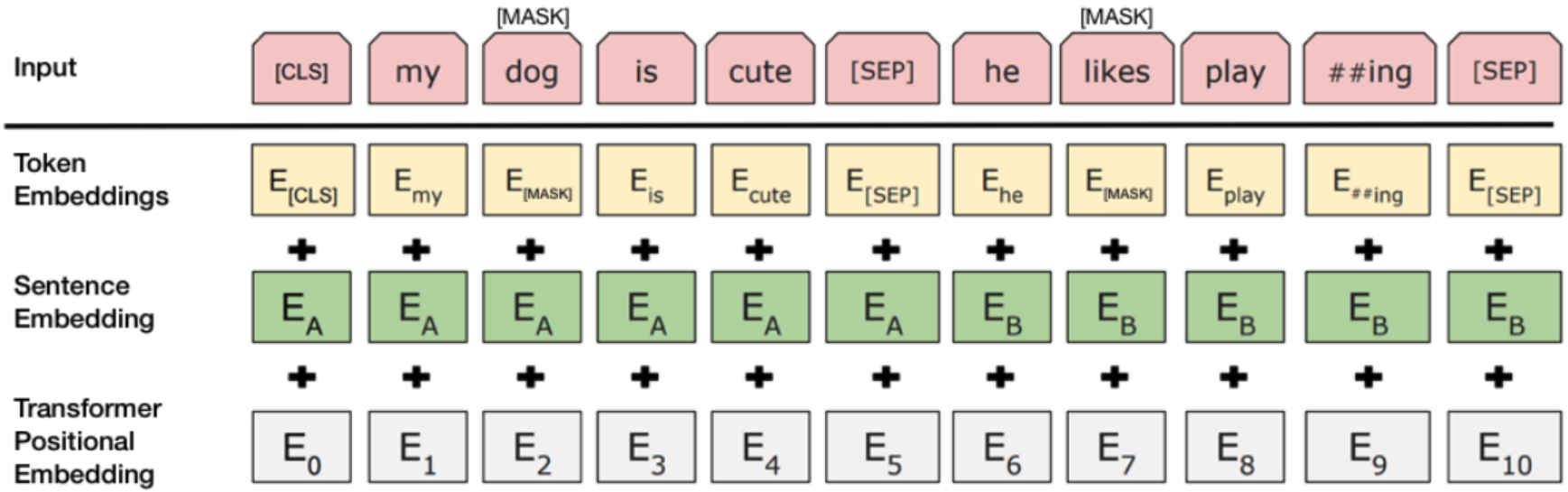




There are 2 steps in the BERT framework:

- Pre-Training and Fine-Tuning.
- During pre-training, the model is trained on unlabeled data over different pre-training tasks.
- For finetuning, the BERT model is first initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labeled data from the downstream tasks. Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters.

Word Embeddings & I/O representation in BERT:



Every word embedding in BERT is a combination of the below 3 things:

- Position Embeddings: BERT learns and uses positional embeddings to express the position of words in a sentence. These are added to overcome the limitation of Transformer which, unlike an RNN/LSTM, is not able to capture “sequence” or “order” information

- Segment Embeddings: BERT can also take sentence pairs as inputs for tasks (Question-Answering). That's why it learns a unique embedding for the first and the second sentences to help the model distinguish between them.
- Token Embeddings: These are the embeddings learned for the specific token from the WordPiece token vocabulary

BERT Special Tokens:

BERT is a pretrained model that expects input data in a specific format, the various tokens used are as follows:

- **[UNK]** – The unknown token. A token that is not in the vocabulary cannot be converted to an ID and is set to be this token instead.
- **[SEP]** – The separator token, which is used when building a sequence from multiple sequences, e.g. two sequences for sequence classification or for a text and a question for question answering. It is also used as the last token of a sequence built with special tokens.
- **[PAD]** – The token used for padding, for example when batching sequences of different lengths.
- **[CLS]** – The classifier token which is used when doing sequence classification (classification of the whole sequence instead of per-token classification). It is the first token of the sequence when built with special tokens.
- **[MASK]** – The token used for masking values. This is the token used when training this model with masked language modeling. This is the token which the model will try to predict.

BERT Pre-Training:

BERT is pre-trained on the following 2 NLP tasks:

1. Masked Language Modelling(MLM):

- BERT is designed as a deeply bidirectional model. The network effectively captures information from both the right and left context of a token from the first layer itself and all the way through to the last layer.
- Bidirectional conditioning would allow each word to indirectly "see itself", and the model could trivially predict the target word in a multi-layered context. In order to train a deep bidirectional representation, simply mask some percentage of the input tokens at random, and then predict those masked tokens.
- The final hidden vectors corresponding to the mask tokens are fed into an output softmax over the vocabulary.
- 15% of all WordPiece tokens are masked in each sequence at random and the only predict the masked words are predicted rather than reconstructing the entire input.
- A downside involved in this method of pre-training is that of a mismatch between pre-training and fine-tuning, since the [MASK] token does not appear during fine-tuning.
- To mitigate this, "masked" words are not always replaced with the actual [MASK] token. The training data generator chooses 15% of the token positions at random for prediction. If the i-th token is chosen, then replacement of the i-th token is done with (1) the [MASK] token 80% of the time (2) a random token 10% of the time (3) the unchanged i-th token 10% of the time. Then, it will be used to predict the original token with cross entropy loss.

2. Next Sentence Prediction(NSP):

- In order to train a model that understands sentence relationships, pre-training is done for a binarized next sentence prediction task that can be trivially generated from any monolingual corpus.
- Specifically, when choosing the sentences A and B for each pretraining example, 50% of the time B is the actual next sentence that follows A (labeled as IsNext), and 50% of the time it is a random sentence from the corpus (labeled as NotNext).
- This kind of pre-training is very beneficial to both QA and NLI.

MLM Demonstration:

- Assuming the unlabeled sentence is **my dog is hairy**, and during the random masking procedure we chose the 4-th token, the masking procedure can be further illustrated as below:
- 80% of the time: Replace the word with the [MASK] token ==> **my dog is hairy** ==> **my dog is [MASK]**
- 10% of the time: Replace the word with a random word ==> **my dog is hairy** ==> **my dog is apple**
- 10% of the time: Keep the word unchanged ==> **my dog is hairy** ==> **my dog is hairy**. The purpose of this is to bias the representation towards the actual observed word.

Advantages of MLM:

- The advantage of this procedure is that the Transformer encoder does not know which words it will be asked to predict or which have been replaced by random words, so it is forced to keep a distributional contextual representation of every input token.

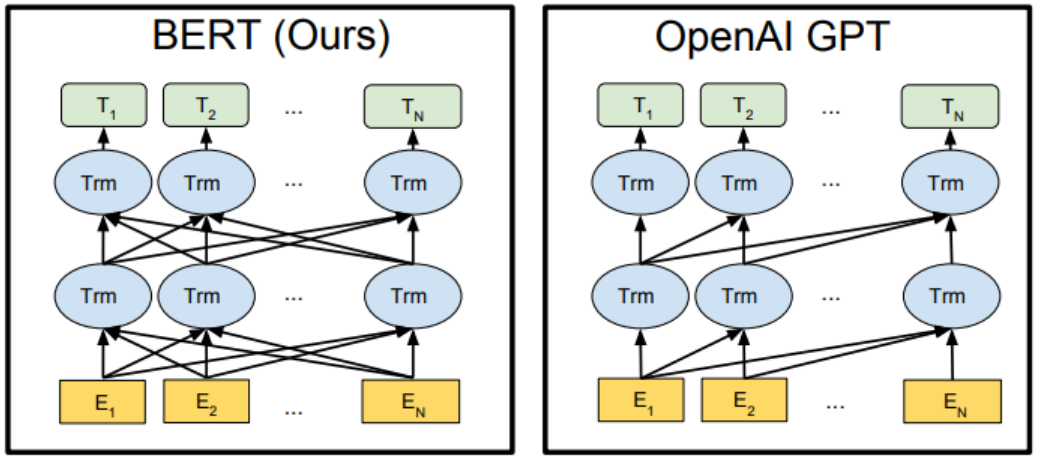
NSP Demonstration:

- Next Sentence Prediction task can be illustrated as follows:
- Input = [CLS] the man went to [MASK] store [SEP]
- he bought a gallon [MASK] milk [SEP]
- Label = IsNext
- Input = [CLS] the man [MASK] to the store [SEP]
- penguin [MASK] are flight ##less birds [SEP]
- Label = NotNext
- To generate each training input sequence, two spans of text are sampled from the corpus.
- The first sentence receives the A embedding and the second receives the B embedding.
- 50% of the time B is the actual next sentence that follows A and 50% of the time it is a random sentence, which is done for the “next sentence prediction” task. They are sampled such that the combined length is 512 tokens.

Fine Tuning Procedure:

- The optimal hyperparameter values are task-specific, however the following range of possible values are found to work well across all tasks:
- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4
- Large data sets (100k+ labeled training examples) were far less sensitive to hyperparameter choice than small data sets.

BERT comparison with predecessor GPT:



- GPT is trained on the BooksCorpus (800M words); BERT is trained on the BooksCorpus (800M words) and Wikipedia (2,500M words).
- GPT uses a sentence separator ([SEP]) and classifier token ([CLS]) which are only introduced at fine-tuning time; BERT learns [SEP], [CLS] and sentence A/B embeddings during pre-training.
- GPT was trained for 1M steps with a batch size of 32,000 words; BERT was trained for 1M steps with a batch size of 128,000 words.
- GPT used the same learning rate of 5e-5 for all fine-tuning experiments; BERT chooses a task-specific fine-tuning learning rate which performs the best on the development set.

BERT Tokenization Demo:

In [9]:
```
!pip install transformers
!pip install torch
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.7/dist-packages (4.9.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-packages (from transformers) (4.41.1)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-packages (from transformers) (1.19.5)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.7/dist-packages (from transformers) (5.4.1)
Requirement already satisfied: huggingface-hub==0.0.12 in /usr/local/lib/python3.7/dist-packages (from transformers) (0.0.12)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.7/dist-packages (from transformers) (2019.12.20)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from transformers) (2.23.0)
Requirement already satisfied: tokenizers<0.11,>=0.10.1 in /usr/local/lib/python3.7/dist-packages (from transformers) (0.10.3)
Requirement already satisfied: sacremoses in /usr/local/lib/python3.7/dist-packages (from transformers) (0.0.45)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (from transformers) (3.0.12)
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-packages (from transformers) (4.6.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages (from transformers) (21.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from huggingface-hub==0.0.12->transformers) (3.7.4.3)
Requirement already satisfied: pyparsing>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from packaging->transformers) (2.4.7)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata->transformers) (3.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->transformers) (2021.5.30)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->transformers) (2.10)
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from sacremoses->transformers) (7.1.2)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from sacremoses->transformers) (1.15.0)
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (from sacremoses->transformers) (1.0.1)
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.9.0+cu102)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from torch) (3.7.4.3)
```

Load Tokenizer:

In [6]:
```
import torch
from transformers import BertTokenizer

bert_base_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
HBox(children=(FloatProgress(value=0.0, description='Downloading', max=231508.0, style=ProgressStyle(descripti…


HBox(children=(FloatProgress(value=0.0, description='Downloading', max=28.0, style=ProgressStyle(description_w…


HBox(children=(FloatProgress(value=0.0, description='Downloading', max=466062.0, style=ProgressStyle(descripti…


HBox(children=(FloatProgress(value=0.0, description='Downloading', max=570.0, style=ProgressStyle(description_…
```

In [10]:
```
sample_text = "Let's observe how word embeddings and tokenization works in BERT"
tokenizer_output = bert_base_tokenizer.tokenize(sample_text)

tokenizer_output
```

Out[10]:
```
['let',
 "'",
 's',
 'observe',
 'how',
 'word',
 'em',
 '##bed',
 '##ding',
 '##s',
 'and',
 'token',
 '##ization',
 'works',
 'in',
 'bert']
```

In [12]:
```
bert_base_tokenizer.all_special_tokens
```

Out[12]:
```
['[UNK]', '[SEP]', '[PAD]', '[CLS]', '[MASK]']
```

In [14]:
```
bert_base_tokenizer.vocab.keys()
```

Out[14]:
```
odict_keys(['[PAD]', '[unused0]', '[unused1]', '[unused2]', '[unused3]', '[unused4]', '[unused5]', '[unused6]', '[unused7]', '[unused8]', '[unused9]', '[unused10]', '[unused
11]', '[unused12]', '[unused13]', '[unused14]', '[unused15]', '[unused16]', '[unused17]', '[unused18]', '[unused19]', '[unused20]', '[unused21]', '[unused22]', '[unused23]',
'[unused24]', '[unused25]', '[unused26]', '[unused27]', '[unused28]', '[unused29]', '[unused30]', '[unused31]', '[unused32]', '[unused33]', '[unused34]', '[unused35]', '[unu
sed36]', '[unused37]', '[unused38]', '[unused39]', '[unused40]', '[unused41]', '[unused42]', '[unused43]', '[unused44]', '[unused45]', '[unused46]', '[unused47]', '[unused4
8]', '[unused49]', '[unused50]', '[unused51]', '[unused52]', '[unused53]', '[unused54]', '[unused55]', '[unused56]', '[unused57]', '[unused58]', '[unused59]', '[unused60]',
'[unused61]', '[unused62]', '[unused63]', '[unused64]', '[unused65]', '[unused66]', '[unused67]', '[unused68]', '[unused69]', '[unused70]', '[unused71]', '[unused72]', '[unu
sed73]', '[unused74]', '[unused75]', '[unused76]', '[unused77]', '[unused78]', '[unused79]', '[unused80]', '[unused81]', '[unused82]', '[unused83]', '[unused84]', '[unused8
5]', '[unused86]', '[unused87]', '[unused88]', '[unused89]', '[unused90]', '[unused91]', '[unused92]', '[unused93]', '[unused94]', '[unused95]', '[unused96]', '[unused97]',
'[unused98]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '[unused99]', '[unused100]', '[unused101]', '[unused102]', '[unused103]', '[unused104]', '[unused105]', '[unused106]', '[u
nused107]', '[unused108]', '[unused109]', '[unused110]', '[unused111]', '[unused112]', '[unused113]', '[unused114]', '[unused115]', '[unused116]', '[unused117]', '[unused11
8]', '[unused119]', '[unused120]', '[unused121]', '[unused122]', '[unused123]', '[unused124]', '[unused125]', '[unused126]', '[unused127]', '[unused128]', '[unused129]', '[u
nused130]', '[unused131]', '[unused132]', '[unused133]', '[unused134]', '[unused135]', '[unused136]', '[unused137]', '[unused138]', '[unused139]', '[unused140]', '[unused14
1]', '[unused142]', '[unused143]', '[unused144]', '[unused145]', '[unused146]', '[unused147]', '[unused148]', '[unused149]', '[unused150]', '[unused151]', '[unused152]', '[u
nused153]', '[unused154]', '[unused155]', '[unused156]', '[unused157]', '[unused158]', '[unused159]', '[unused160]', '[unused161]', '[unused162]', '[unused163]', '[unused16
4]', '[unused165]', '[unused166]', '[unused167]', '[unused168]', '[unused169]', '[unused170]', '[unused171]', '[unused172]', '[unused173]', '[unused174]', '[unused175]', '[u
nused176]', '[unused177]', '[unused178]', '[unused179]', '[unused180]', '[unused181]', '[unused182]', '[unused183]', '[unused184]', '[unused185]', '[unused186]', '[unused18
7]', '[unused188]', '[unused189]', '[unused190]', '[unused191]', '[unused192]', '[unused193]', '[unused194]', '[unused195]', '[unused196]', '[unused197]', '[unused198]', '[u
nused199]', '[unused200]', '[unused201]', '[unused202]', '[unused203]', '[unused204]', '[unused205]', '[unused206]', '[unused207]', '[unused208]', '[unused209]', '[unused21
0]', '[unused211]', '[unused212]', '[unused213]', '[unused214]', '[unused215]', '[unused216]', '[unused217]', '[unused218]', '[unused219]', '[unused220]', '[unused221]', '[u
```

In [ ]: