Natural Language Understanding, Generation and Machine Translation - Week 4 - Introduction to Machine Translation

Antonio León Villares

February 2023

Contents

Based on:

- *Efficient Estimation of Word Representations in Vector Speace, by Mikolov et al.*
- *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*
- *BERT Neural Network - EXPLAINED! by CodeEmporium*

1 Static Embeddings: Word2Vec

1.1 Static Embeddings

- **What is a static embedding?**
  - a **vector representation** of a word, **independent** of its **context**
  - the sort of embeddings we've been using as **inputs** to our **network**
- **Are static word embeddings complicated to generate?**
  - no, they use simple **linear models**
  - as such, can be even trained from scratch
- **What is the distributional hypothesis?**
  - **similar words** occur in **similar contexts**
  - **Word2Vec** is a set of models, which use the **distributional hypothesis** to derive **static word embeddings**
- **Why don't static embeddings use non-linearities?**
  - **static embeddings** have been shown to be able to handle **simple linear** relationships:

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

  - adding **non-linearities** hurts this simple, albeit powerful property
  - we might learn more **complex** embeddings, but they won't make as much sense
  - involving a simple linear model is good for:
    - **Interpretability**: can more easily "understand" embeddings
    - **Training Speed**: since we don't model complex non-linear relationships, we only really need one layer, so training speeds up

1.2 CBOW

- **What is the CBOW model?**
  - **CBOW** stands for **Continuous Bag-Of-Words** model
  - it tries to predict a **word** given its **context** (for example, using the 2 **previous** and **following** words)
  - for example:

    "Lincoln won the `[MISSING]` in 1860" $\mapsto$ "election"

- **What is the structure of the CBOW model?**
  - it contains a single (linear) **hidden layer**
  - the **parameter matrix** W for this hidden layer is what actually contains the learnt embeddings, and is **shared**
  - the context vectors, once passed through the hidden layer, are passed through another matrix to get the **output** (the predicted **central word**, given the **context)**

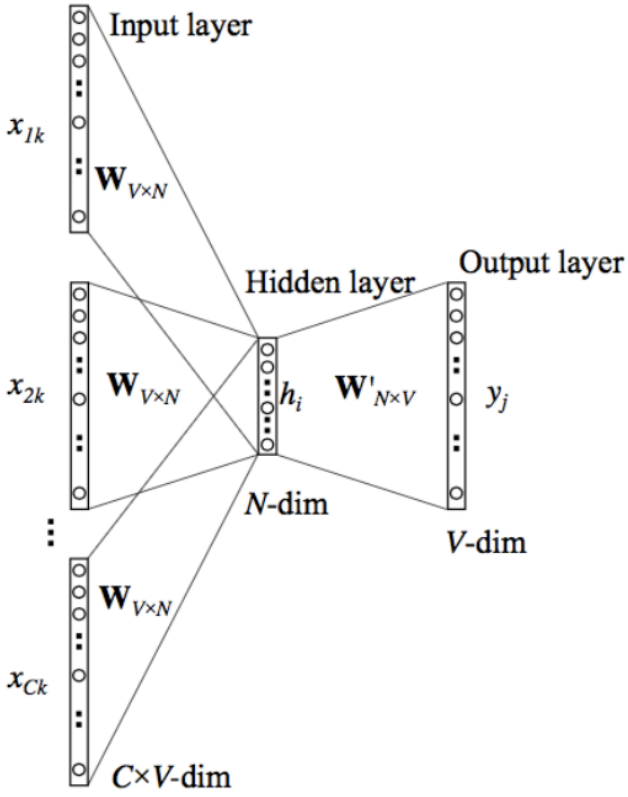
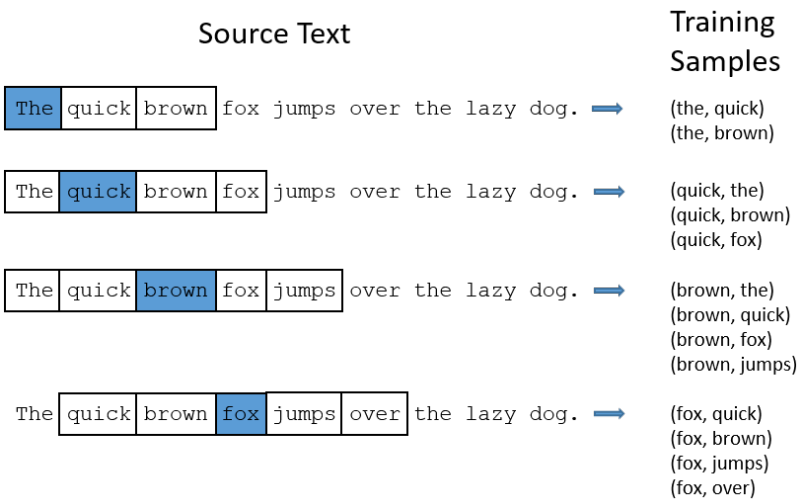


Figure 1: W is a shared matrix: each context word (encoded as a **one-hot vector**) gets multiplied by W. Thus, the embeddings are in the **columns** of W. In the Mikolov paper, the embeddings for the context vectors are **averaged** to give the **output**. In the slides (and the diagram above), the embeddings are **stacked**, and passed through a final parameter matrix W' to produce the output prediction.

1.3 Skipgram

- **What is the Skipgram model?**

- unlike with **CBOW**, **Skipgram** learns embeddings by predicting **context words** given a **central word**
- for example:



- **What is the structure of Skipgram?**
  - very similar to **CBOW**
  - the main difference is that the output of the network will be an individual context word, given as a **distribution** (using softmax)

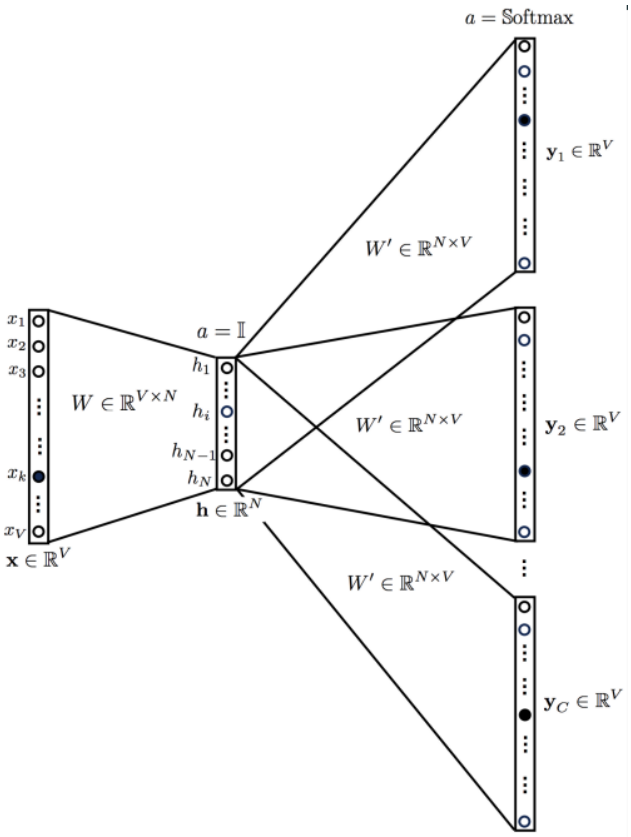


Figure 2

– I much prefer the presentation by Jurafsky and Martin, which can be found in Section 1 of my FNLP notes

We can look at a concrete example. Consider:

"The quick brown fox jumps"

In CBOW, we'd one-hot encode "brown", and then generate training samples of the form (The, brown), (quick, brown), (fox, brown) and (jumps, brown). Then, we generate the embeddings for "The", "quick", "fox", "jumps" by multiplying the one-hot encodings by the parameter matrix W. In Mikolov et al., the resulting embeddings are just averaged to produce an output for "brown". In the course, the embeddings are "stacked" and multiplied by another matrix W' to generate the output for "brown".

On the other hand, in Skipgram, our training samples will look like (brown, The), (brown, quick), (brown, fox) and (brown, jumps). When training the model we pass each of the training samples individually through the model: firstly through the parameter matrix W to get the embeddings of each word, then apply W', and then softmax to get a distribution for the context word.

- **What objective function does Skipgram seek to maximise?**
  - let W be the set of **central words**
  - let C be the set of **context words**
  - let $D \subseteq W \times C$ be a **corpus** of word-context pairs
  - for example, if we have a sentence:

    "The quick brown fox jumps."

    we'd have:

    `(brown, the), (brown, quick), (brown, fox), (brown, jumps)` $\in D$

  - we seek to find parameters θ such that we **maximise** the likelihood of predicting a context c given a central word w:

$$\underset{\theta}{argmax} \prod_{(w,c)\in D} P(c \mid w; \theta)$$

- **How is this probability computed?**
  - for each $w \in W$, we learn an embedding $\underline{v}_w$
  - for each $c \in C$, we learn an embedding $\underline{v}_c$
  - if similar words appear in similar contexts, we'd want to **maximise** the similarity of a **central word** with its **context word**
  - this in turn will maximise the **similarity** of words which appear in similar context
  - thus, **Skipgram** computes:

$$P(c \mid w; \theta) = \frac{\exp(\underline{v}_w \cdot \underline{v}_c)}{\sum_{c' \in C} \exp(\underline{v}_{c'} \cdot \underline{v}_w)}$$

- **How is negative sampling used to speed up the training of the model?**
  - the **denominator** of $P(c \mid w; \theta)$ is expensive to compute: it involves dot products and sums over **all** the context words
  - instead, with **negative sampling**, we randomly generate **negative sample** words N (i.e words which don't appear in the context of our central word)
  - we then do many small **logistic regression** classification tasks with each word in N. For some $c \in N$ and $w \in W$:
$$P(c \mid w; \theta) = \sigma(\underline{v}_c \cdot \underline{v}_w) = \frac{1}{1 + \exp(-\underline{v}_c \cdot \underline{v}_w)}$$
  - all these independent logistic classifiers can be used to approximate the **large** softmax
  - the idea is that a given word will appear on very **specific** contexts, so it won't co-occur with **most** context words
  - as such, it is inefficient to use **all** the context words, when a smaller sample will still be **representative**
- **How does negative sampling alter the Skipgram objective function?**
  - we still have D to be the set of **positive sample pairs** (i.e words and contexts which occur together)
  - let D' be the set of **negative sample pairs** (i.e a representative sample of words and contexts which **don't** occur together)
  - let:
$$P(D = 1 \mid w, c)$$
    denote the probability that $(w, c) \in D$ and:
$$P(D = 0 \mid w, c) = 1 - P(D = 1 \mid w, c)$$
    that $(w, c) \notin D$
  - then we seek:
$$\begin{aligned}
&\underset{\theta}{argmax} \prod_{(w,c) \in D} P(D = 1 \mid w, c; \theta) \prod_{(w,c) \in D'} P(D = 0 \mid w, c; \theta) \\
=&\underset{\theta}{argmax} \sum_{(w,c) \in D} \log\left(P(D = 1 \mid w, c; \theta)\right) + \sum_{(w,c) \in D'} \log\left(P(D = 0 \mid w, c; \theta)\right) \\
=&\underset{\theta}{argmax} \sum_{(w,c) \in D} \log\left(P(D = 1 \mid w, c; \theta)\right) + \sum_{(w,c) \in D'} \log\left(1 - P(D = 1 \mid w, c; \theta)\right) \\
=&\underset{\theta}{argmax} \sum_{(w,c) \in D} \log(\sigma(\underline{v}_c \cdot \underline{v}_w)) + \sum_{(w,c) \in D'} \log(\sigma(-\underline{v}_c \cdot \underline{v}_w))
\end{aligned}$$
  - for the last step, we have used the property of the sigmoid:
$$1 - \sigma(z) = \sigma(-z)$$
- **How are the samples for negative sampling selected?**
  - use **weighted unigram frequency**:
$$P_\alpha(c) = \frac{count(c)^\alpha}{\sum_{c' \in C} count(c')^\alpha}$$

- typically use $\alpha \approx 0.75$, so that uncommon words can be sampled

- **How is the Skipgram model trained?**
  - we optimise the **objective function** with **gradient descent** and **backpropagation**
  - we also want to throw out words **proportionally** to their **frequency**:
    * faster training
    * reduces importance of frequent words (i.e "the", "is", "a" aren't too useful for embeddings)

1.4 Evaluating Word2Vec

- **Wich method is better out of CBOW and Skipgram?**
  - it depends on the task at hand
  - **CBOW** is **faster** to train, and tends to capture **frequent** words better (since it is trained to predict a word given context, it will assign more probability mass to frequent words)
  - **Skipgram** has the benefit that it can generate good embeddings for **infrequent words** (since these depend on context), so it can be trained with smaller datasets
  - **CBOW** tends to better capture **syntactic** information, whilst **Skipgram** tends to better capture **semantic** information (according to Mikolov paper)

Model Architecture	Semantic-Syntactic Word Relationship test set	
	Semantic Accuracy [%]	Syntactic Accuracy [%]
RNNLM	9	36
NNLM	23	53
CBOW	24	64
Skip-gram	55	59

- **How can the quality of the embeddings be evaluated?**
  - we already saw that embeddings tend to capture semantic relations fairly well:

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

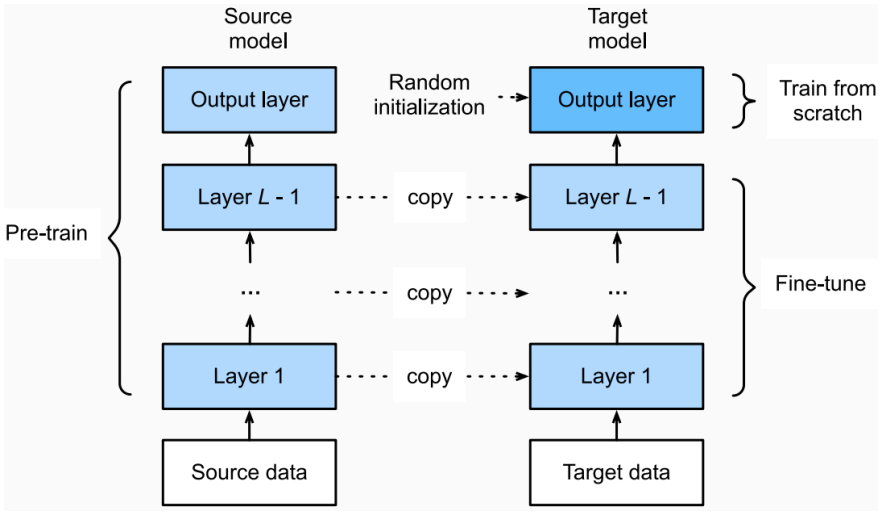
  - another way of verifying this is **question answering**: given a word ("question"), find the closest embedding ("answer"):

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

2 Contextualised/Dynamic Word Embeddings: BERT

2.1 Pre-Training and Fine-Tuning

- **What is pre-training?**
  - training a **generic source model** on some standard, large dataset
  - the training task can be pretty general
  - for example, **ResNET** is pre-trained on classification for **ImageNet**
- **What is fine-tuning?**
  - we take the **pre-trained source model**, replace its **output** layers and train it to suit some **specific task**
  - the new model is known as the **target model**
- **What is the purpose of pre-training and fine-tuning?**
  - with **pre-training**, we can learn good initialisation parameters (i.e parameters which are good at **feature extraction**)
  - with **fine-tuning** we leverage the learnt features for a more complex task
  - ultimately, this should reduce the **training time** of the **target model**

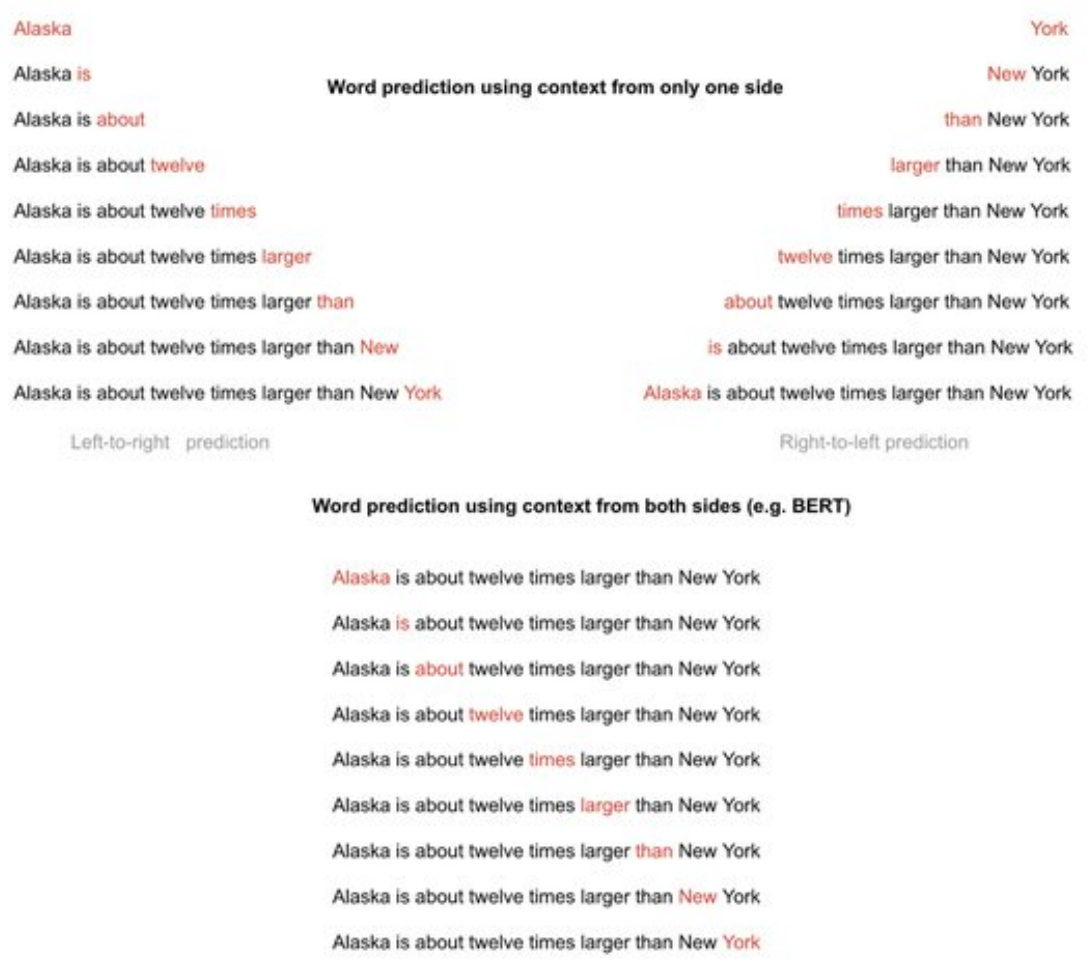


- **What architecture/training considerations should be made when fine-tuning?**
  1. **Truncate Output**: the **fine-tuning** objective will likely be different from the **pre-training objective** (i.e fewer output classes), so the final layers of the architecture might need to be modified slightly.
  2. **Weight Freezing**: to speed up **fine-tuning**, we can **fix** the weights at the start of the network, and only learn new parameters for the **final** layers.
  3. **Learning Rate**: assuming **pre-training** learnt good features, a **smaller learning rate** should be used.
- **How does pre-training and fine-tuning fit into NLP?**
  - we can **pre-train** large models on **language modelling** tasks
  - the final layers of the **source model** should then provide us with **powerful embeddings** encoding textual information
  - we can then use these **embeddings** for our **target model** (i.e **question answering**, **sentence completion**, etc ...)
- **What are contextualised/dynamic word embeddings?**
  - **embeddings** learnt from **pre-trained models**
  - the **same** word can have **different** embeddings, depending on the **context** in which it appears
  - **dynamic embeddings** can be **fine-tuned** to tackle a specific task
  - these are much more **expensive** to train (both in time and memory), which is why **fine-tuning** is useful

2.2 BERT's Architecture

- **What is the purpose of BERT?**
  - BERT stand for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers
  - designed to be **pre-trained** to learn **dynamic embeddings**
  - for many **fine-tuning** tasks, we just need to replace the **output** layer
  - achieved **stated of the art** results in 11 NLP tasks via **fine-tuning**

- **Why is `BERT` said to be bidirectional?**
  - `BERT` is **pre-trained** as a **language model**
  - it learns embeddings by leveraging both **left** and **right** contexts
  - for comparison, simple **RNN** language models model language in a **left-to-right** manner
  - we also considered learning **bidirectional** embeddings with a **left** and **right** RNN, and then concatenating them to produce a final embedding
  - however, `BERT` is truly bidirectional, in the sense that it learns an embedding by considering **all** contexts **at the same time**



- **How do `BERT`'s bidirectional embeddings compete with other models, like `GPT` and `ELMo`?**
  - intuitively, a **truly bidirectional** embedding should be able to more powerfully represent a word
  - in `GPT`, they use **transformers** to learn **left-to-right** context embeddings (each token can only self-attend tokens to its left)

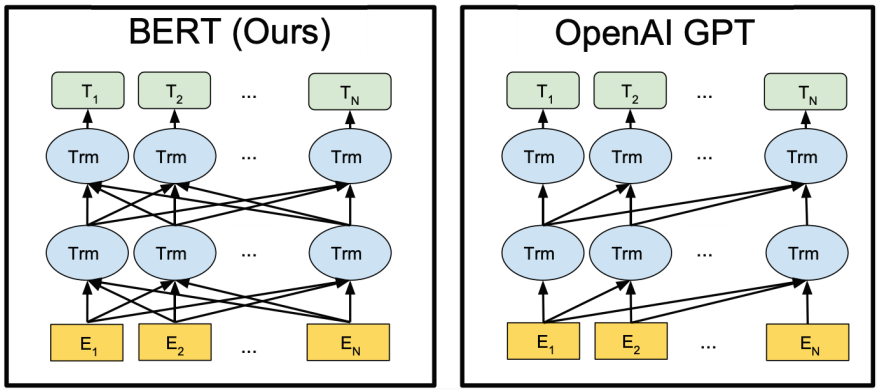


Figure 3: `GPT` and `BERT` both use **transformers**; however, `GPT` only uses **left-to-right** self-attention.

- in `ELMo`, they use independently trained **LSTMs** to learn **left-to-right** and **right-to-left** embeddings, which are then concatenated

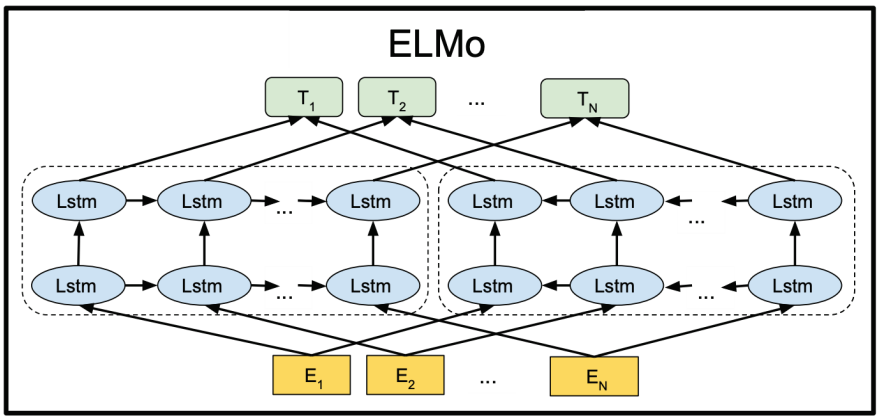


Figure 4: `ELMo` and `BERT` both use **bidirectional embeddings**; however, `ELMo`**'s** left and right embeddings are learnt independently and then concatenated.

- `BERT`**'s** approach makes it so that the learnt embeddings are better for **sentence-level tasks**, where a word will depend on both **left** and **right** contexts, which themselves should be **interdependent**

- **What is `BERT`'s model architecture?**
  - `BERT` is a **multilayer transformer**
  - the authors devised 2 different `BERT`**'s**
  - let:
    - L: number of transformer blocks
    - H: dimensionality of hidden layer
    - A: number of self-attention heads
  - then:

$$\texttt{BERT}_{BASE} = \texttt{BERT}(L = 12, H = 768, A = 12) \implies 110M \ parameters$$

$$\texttt{BERT}_{LARGE} = \texttt{BERT}(L = 24, H = 1024, A = 16) \implies 340M \ parameters$$

  - $\texttt{BERT}_{BASE}$ was chosen to have the same model size as `GPT` (for comparison purposes)

- **What types of input does `BERT` accept?**
  - `BERT` can handle both **sentence pairs** (i.e $\langle question, answer \rangle$) and **single sentences** in an **unambiguous** manner
  - each sentence is broken down into a sequence of **tokens** (out of 30,000 total tokens, represented as WordPiece embeddings)
  - there are 2 **special tokens**:
    * `[CLS]`: first token of any sequence. The hidden state for this token is used as a representation for **classification**.
    * `[SEP]`: used to separate 2 sentences. A **sentence pair** is passed as a single sentence, which is separated by this token.
- **How are sentences converted into an input for `BERT`?**

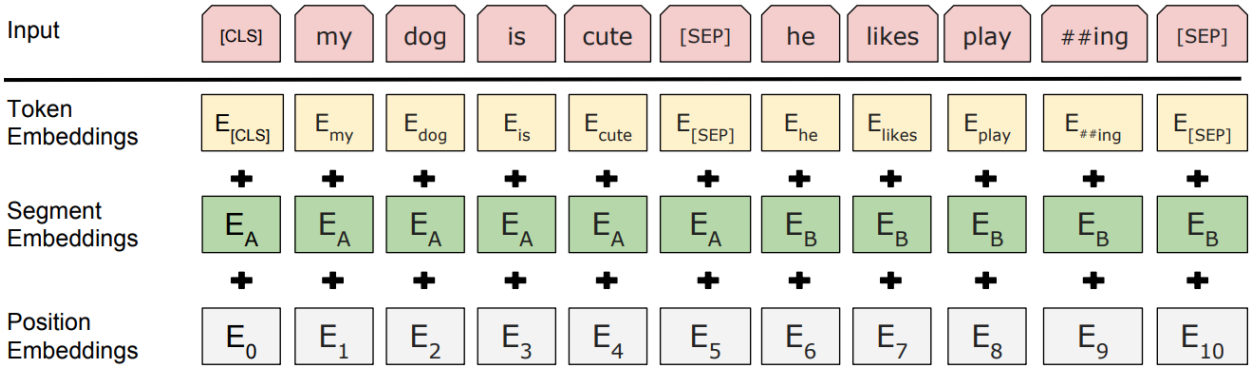
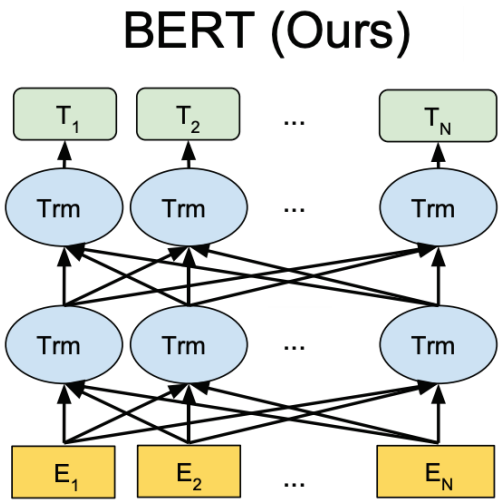


Figure 5: The **input representation** for a token is a sum over **token embeddings** (the static embedding for that token according to WordPiece), **segment embeddings** (an embedding, corresponding to whether the token belongs to the first sentence (sentence `A`) or the second sentence (sentence `B`)) and **position embeddings** (embedding corresponding to the position of the token in the whole input sentence).

- **What is the purpose of the segment embeddings?**
  - we need to know whether a token is part of the first or second sentence
  - this is useful for example in **question answering** or **sentence continuation**
- **What does `BERT` output (during pre-training)?**
  - two types of output with different functionality:
    1. $\underline{T_i} \in \mathbb{R}^H$: the final hidden vector for the ith input token (the **transformed** version of E_i, after passing through the transformer)



2. $\underline{C} \in \mathbb{R}^H$: the final hidden vector of the `[CLS]` token at the start of the sentence

2.3 Pre-Training `BERT`: Masking and Next-Sentence Prediction

- **How is `BERT` pre-trained?**
  - `BERT` is **pre-trained** using 2 **unsupervised tasks** (so no **supervised language modelling**)
  - these are:
    1. **MLM** (Masked Language Modelling): mask some words in the input (using a `[MASK]` token), and make `BERT` predict the word given the context
    2. **NSP** (Next Sentence Prediction): determine whether 2 sentences follow each other.
- **Why is MLM used for training `BERT`?**
  - if we just ask `BERT` to predict a word at a given index, the **self-attention** mechanism would allow it to just "copy"
  - with **masking**, we substitue word occurrences with the `[MASK]` token, which forces context-based prediction:

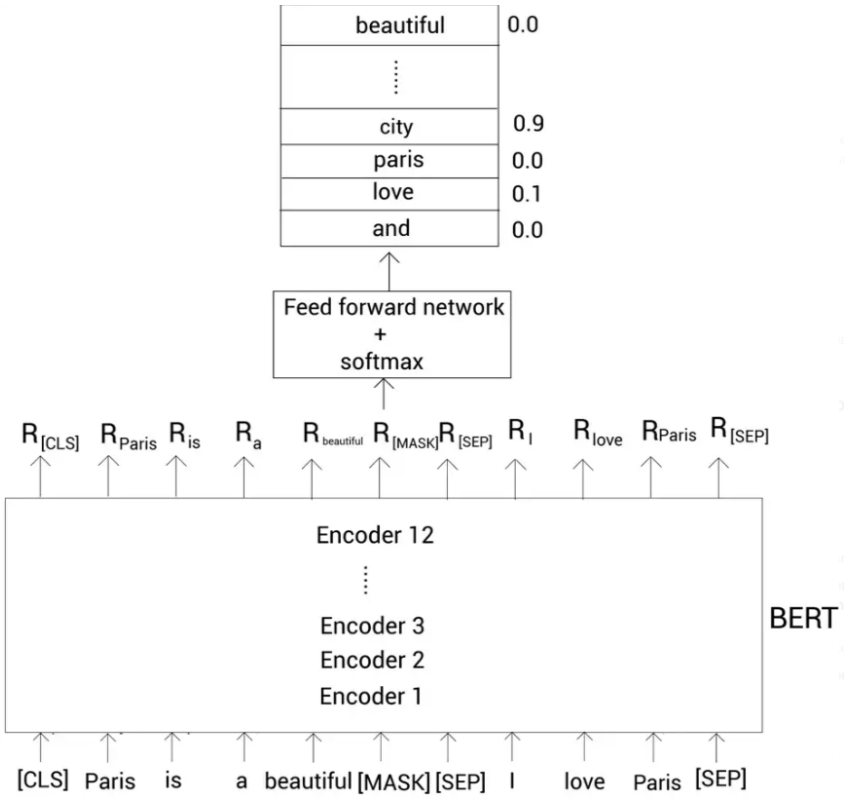
    I like playing football during the summer. $\implies$ I like `[MASK]` football `[MASK]` the summer.
- **How is masking applied when pre-training `BERT`?**
  - we randomly select 15% of the input tokens, and **mask** them
  - for each token to be masked:
    - use the `[MASK]` token (80% of the time)

      My dog is hairy. $\implies$ My dog is `[MASK]`.
    - replace with a random token (10% of the time)

      My dog is hairy. $\implies$ My dog is apple.
    - leave the token unchanged (10% of the time)

      My dog is hairy. $\implies$ My dog is hairy.

– with T_i we then have to predict what the masked token's value was before masking



- **Why can't we apply the `[MASK]` token for each of the masked tokens?**

  1. **Always `[MASK]`**
     - creates a mismatch between **pre-training** and **fine-tuning** (since `[MASK]` won't be seen during fine-tuning)
     - if our objective is just to predict the token before masking, we won't learn a good representation for other words (we only care about "unmasking")
  2. **`[MASK]` + Random Token**
     - `BERT` would learn that an observed word is never correct
     - thus, the transformed embeddings it produces won't correspond to the observed word
  3. **`[MASK]` + Same Token**
     - `BERT` would learn to trivially copy tokens

  - by combining the 3 masking strategies, `BERT` won't know which word it is asked to predict, or which words have been randomly replaced
  - this forces it to keep a **distributional contextual representation** of every possible token

- **What is the NSP task?**

  - we consider 2 sentences `A`,`B`: 50% of the time, `B` follows `A`, and 50% of the time it is randomly selected

– `BERT` then needs to predict which of the 2 cases occurs:

$$\text{Input} = \texttt{[CLS] the man went to [MASK] store [SEP]}$$
$$\texttt{he bought a gallon [MASK] milk [SEP]}$$
$$\text{Label} = \texttt{IsNext}$$

$$\text{Input} = \texttt{[CLS] the man [MASK] to the store [SEP]}$$
$$\texttt{penguin [MASK] are flight ##less birds [SEP]}$$
$$\text{Label} = \texttt{NotNext}$$

– understanding sentence continuity is important for downstream tasks, like **question answering** and **natural language inference**

- **On what data is `BERT` pre-trained?**
  - BooksCorpus: 800M words
  - English Wikipedia: 2,500M words

2.4 Fine-Tuning `BERT`

- **What sort of inputs and outputs can be used for fine-tuning?**
  - **Inputs**:
    * sentence pairs for **paraphrasing**
    * hypothesis-premise pairs in **entailment**
    * question-passage pairs in **question answering**
    * text-$\emptyset$ pair in **text classification** or **sequence tagging**
  - **Outputs**:
    * **answer span** in QA (i.e given a question and a piece of text, find the start and end tokens corresponding to the answer)
    * sequence of **labels** in **named entity recognition**
    * `[CLS]` representation can be fed to output layer for **classification** (i.e entailment, sentiment analysis)

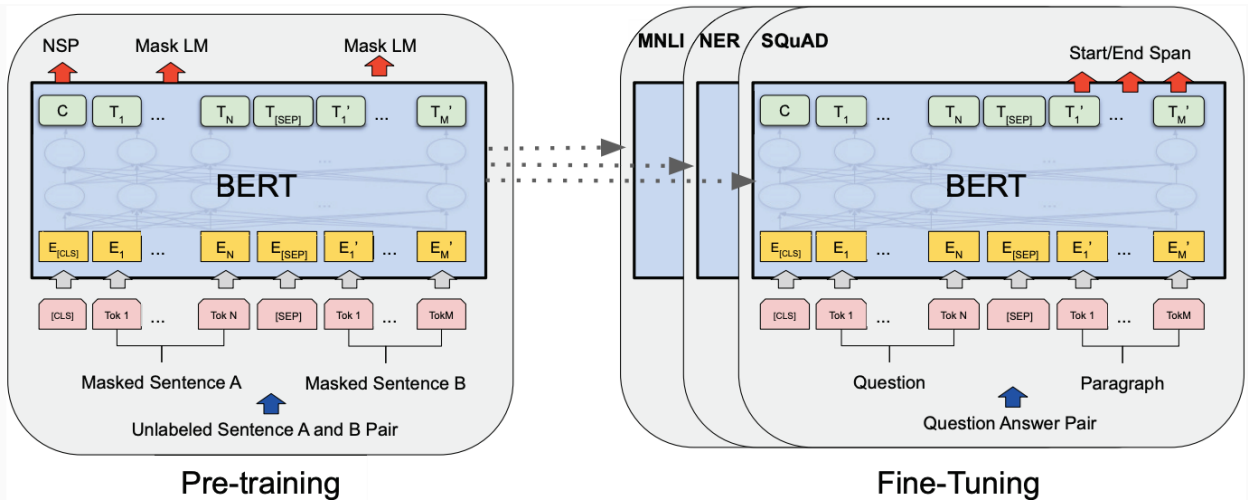


Figure 6

- **What is the GLUE benchmark?**
  - a set of **NLU** tasks
  - `BERT` obtained **state of the art** results in 11 of these tasks

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Table 1: GLUE Test results, scored by the evaluation server (https://gluebenchmark.com/leaderboard). The number below each task denotes the number of training examples. The "Average" column is slightly different than the official GLUE score, since we exclude the problematic WNLI set.[8] BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.

Figure 7: MNLI,QNLI,WNLI: natural language inference; QQP: question equivalence; SST-2: sentiment; CoLA: linguistic acceptability (whether sentence is grammatical); STS-B: semantic similarity; MRPC: paraphrasing; RTE: entailment.

  - we can see that $\texttt{BERT}_{LARGE}$ significantly outperforms $\texttt{BERT}_{BASE}$, especially when little data is available

- **How does feature extraction differ from fine-tuning?**
  - in **fine-tuning**, we replace the **output** layer, in order to obtain some representating **geared** towards the specific task at hand
  - however, this isn't always possible: not all tasks can be tackled with a **transformer encoder architecture**, and require task specific architecture
  - tackling such tasks can be done with **feature extraction**: use the **features** learnt by pre-trained `BERT` a feature representations for the inputs of the specific tasks

– the **features** need not only be the **output embeddings**: we can consider hidden representations in intermediate layers

System	Dev F1	Test F1
ELMo (Peters et al., 2018a)	95.7	92.2
CVT (Clark et al., 2018)	-	92.6
CSE (Akbik et al., 2018)	-	**93.1**
Fine-tuning approach		
$\text{BERT}_{\text{LARGE}}$	96.6	92.8
$\text{BERT}_{\text{BASE}}$	96.4	92.4
Feature-based approach ($\text{BERT}_{\text{BASE}}$)		
Embeddings	91.0	-
Second-to-Last Hidden	95.6	-
Last Hidden	94.9	-
Weighted Sum Last Four Hidden	95.9	-
Concat Last Four Hidden	96.1	-
Weighted Sum All 12 Layers	95.5	-

Figure 8: Performance of `BERT` using fine-tuning and feature extraction. Notice, with feature extraction, we can consider rich representations constructed from many intermediate layers. These results are for CoNLL-2003, a Named Entity Recognition task.

- **How has `BERT` impacted the NLP world?**
  - `BERT` caused a **paradigm shift** towards the pre-training + fine-tuning approach, which has lead to many state of the art results
  - however, this is challenging:
    - **pre-training** requires a lot of computational and memory resources
    - **fine-tuning**, whilst quicker, still requires large memory capabilities
  - as such, achieving state of the art results is a game that only a few can play, although it prompts us to come up with smarter ideas for architectures, objective functions, evaluation, etc...