Transformers

- Context Matters

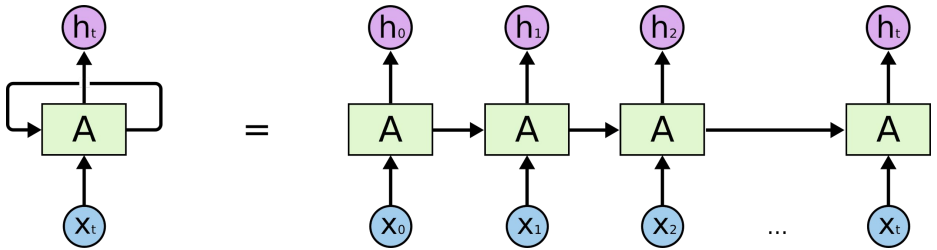
Luís Filipe Cunha
lfc@di.uminho.pt

José João Almeida
jj@di.uminho.pt

Before Transformers

Before Transformers

RNN sequential processing - slow
Vanishing gradient - information loss
CNN - limited context (fixed window)
LSTM and **Bi-LSTM** to the rescue!



- Complex;
- Difficult to train;
- Sequential ->The model can only reason about a word when all the previous words are already processed

Relevant Works

Computer Science > Computation and Language

[Submitted on 15 Feb 2018 (v1), last revised 22 Mar 2018 (this version, v2)]

Deep contextualized word representations

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer

We introduce a new type of deep contextualized word representation that models both (1) complex characteristics of word use (e.g., syntax and semantics), and (2) how these uses vary across linguistic contexts (i.e., to model polysemy). Our word vectors are learned functions of the internal states of a deep bidirectional language model (biLM), which is pre-trained on a large text corpus. We show that these representations can be easily added to existing models and significantly improve the state of the art across six challenging NLP problems, including question answering, textual entailment and sentiment analysis. We also present an analysis showing that exposing the deep internals of the pre-trained network is crucial, allowing downstream models to mix different types of semi-supervision signals.

Comments: NAACL 2018. Originally posted to openreview 27 Oct 2017. v2 updated for NAACL camera ready
Subjects: **Computation and Language (cs.CL)**
Cite as: arXiv:1802.05365 **[cs.CL]**
(or arXiv:1802.05365v2 **[cs.CL]** for this version)

Submission history

From: Matthew Peters [view email]
[v1] Thu, 15 Feb 2018 00:05:11 UTC (135 KB)
[v2] Thu, 22 Mar 2018 21:59:40 UTC (140 KB)

Computer Science > Computation and Language

[Submitted on 18 Jan 2018 (v1), last revised 23 May 2018 (this version, v5)]

Universal Language Model Fine-tuning for Text Classification

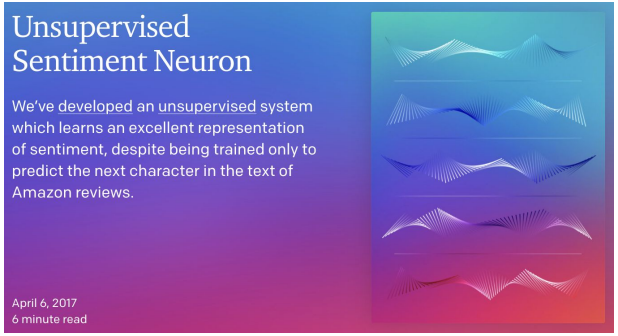
Jeremy Howard, Sebastian Ruder

Inductive transfer learning has greatly impacted computer vision, but existing approaches in NLP still require task-specific modifications and training from scratch. We propose Universal Language Model Fine-tuning (ULMFiT), an effective transfer learning method that can be applied to any task in NLP, and introduce techniques that are key for fine-tuning a language model. Our method significantly outperforms the state-of-the-art on six text classification tasks, reducing the error by 18-24% on the majority of datasets. Furthermore, with only 100 labeled examples, it matches the performance of training from scratch on 100x more data. We open-source our pretrained models and code.

Comments: ACL 2018, fixed denominator in Equation 3, line 3
Subjects: **Computation and Language (cs.CL)**; Machine Learning (cs.LG); Machine Learning (stat.ML)
Cite as: arXiv:1801.06146 **[cs.CL]**
(or arXiv:1801.06146v5 **[cs.CL]** for this version)

Submission history

From: Sebastian Ruder [view email]
[v1] Thu, 18 Jan 2018 17:54:52 UTC (30 KB)
[v2] Mon, 14 May 2018 13:57:04 UTC (851 KB)



NATURAL LANGUAGE PROCESSING

NLP's ImageNet moment has arrived

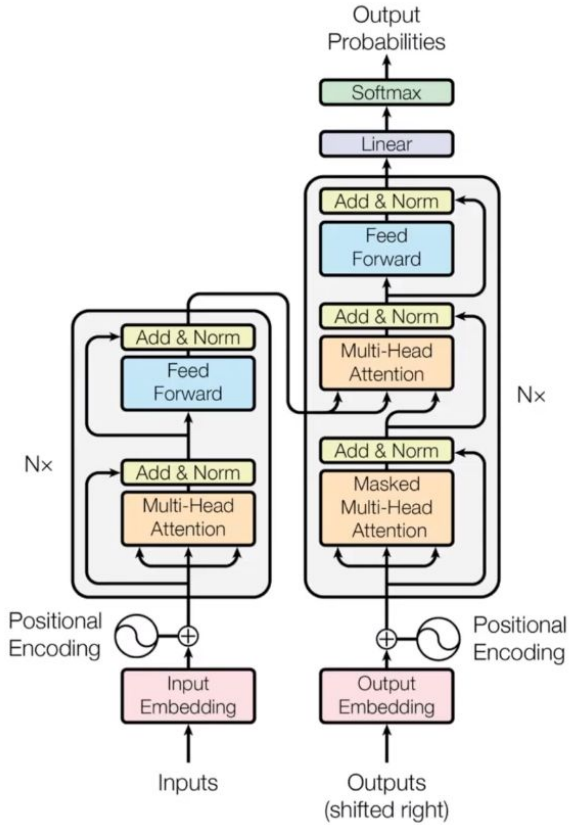
Big changes are underway in the world of NLP. The long reign of word vectors as NLP's core representation technique has seen an exciting new line of challengers emerge. These approaches demonstrated that pretrained language models can achieve state-of-the-art results and herald a watershed moment.

SEBASTIAN RUDER
12 JUL 2018 · 16 MIN READ

Transformers

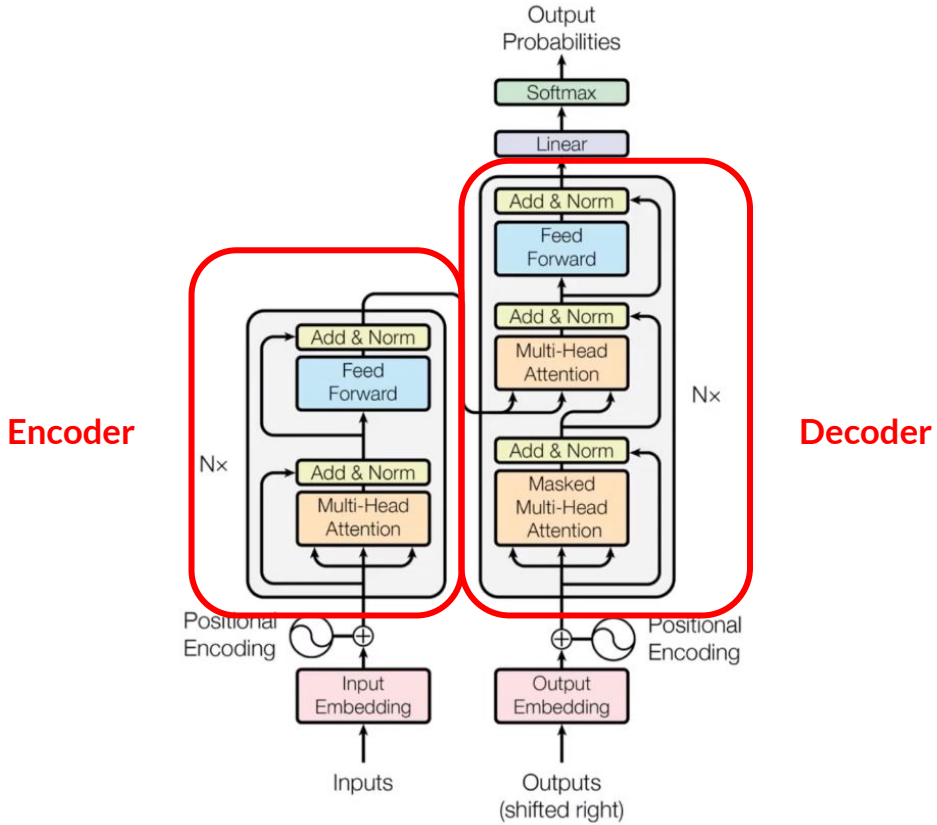
What is a Transformer?

- Introduced in the paper *"Attention is All You Need"* (Vaswani et al., 2017)
- Encoder - Decoder architecture
- **Attention Mechanism**
- **Positional Encodings**
- **Easier to parallelize (unlike LSTMs)**

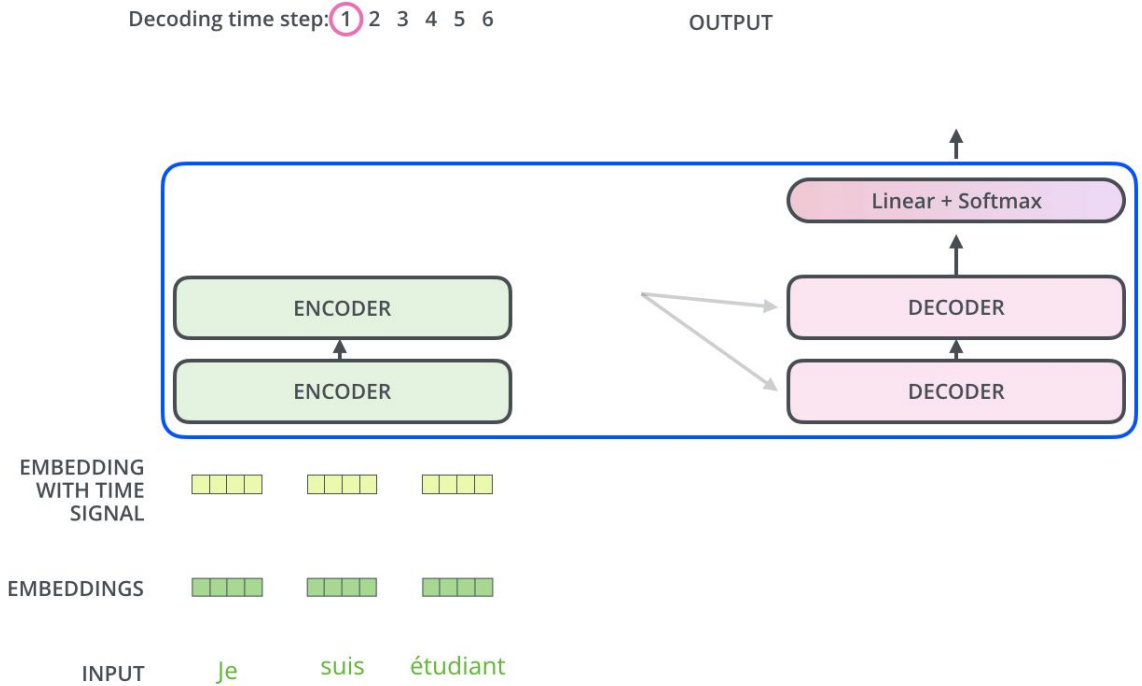


Encoder Decoder

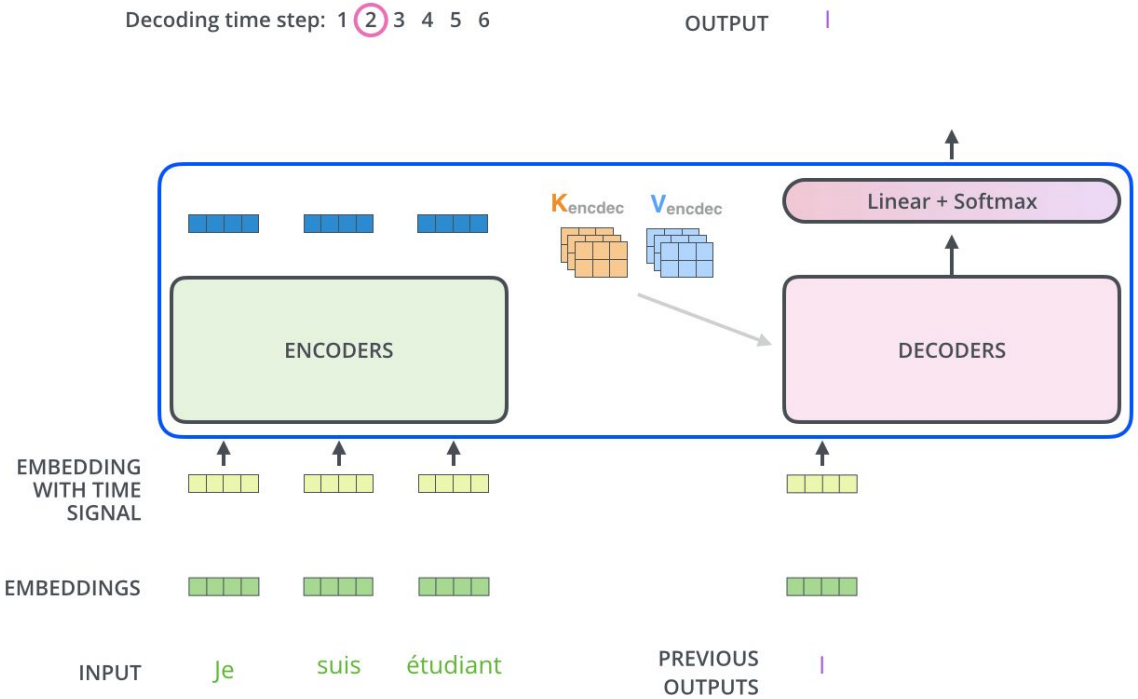
- Originally developed for seq-to-seq tasks (translation, summarization, …)
- **Encoder**: Processes the input data and outputs a sequence of vectors.
- **Decoder**: Takes the encoder's output and generates a sequence of predictions.



Encoder


Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant

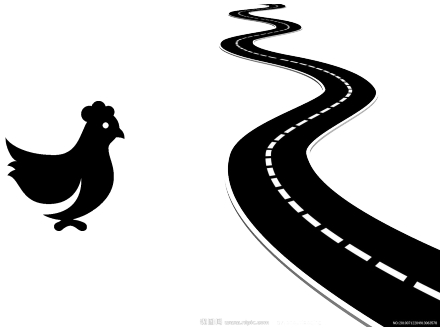
Decoder


Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Linear + Softmax
Kencdec
Vencdec
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

Self-Attention

Consider the following example:

- The chicken didn't cross the road because ***it*** was too tired.
- The chicken didn't cross the road because ***it*** was too wide.



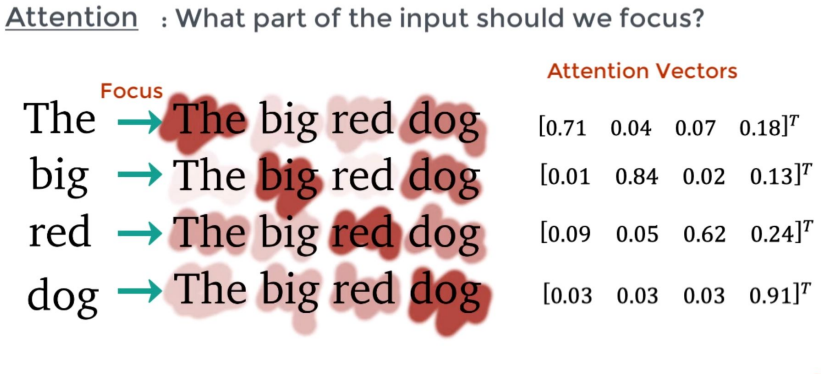
If we are to compute the meaning of this sentence, we'll need the meaning of ***it*** to be associated with the chicken in the first sentence and associated with the road in the second one

Contextual words that help us compute the meaning of words in context can be quite far away in the sentence or paragraph.

Self-Attention

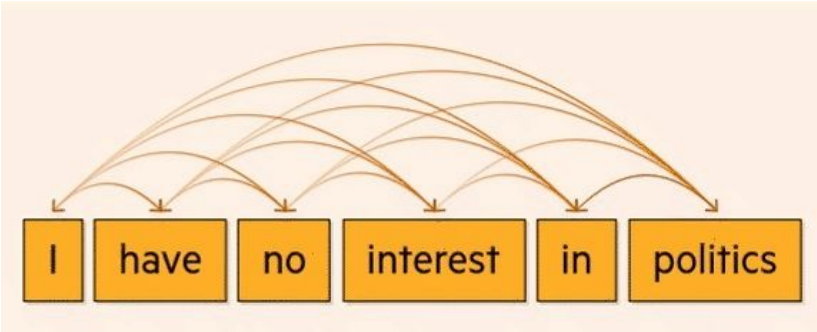
- **Self-Attention** allows each word in the input sequence to focus on different parts of the sequence.

I have no interest in politics



Self-attention looks at each token in a body of text and decides which others are most important to understand its meaning.

Self-Attention



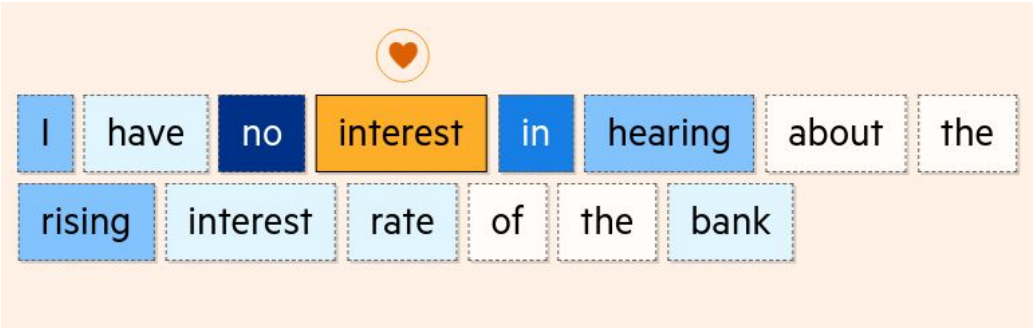
Assessing the whole sentence at once means the transformer is able to understand that interest is being used as a noun to explain an individual's take on politics.

If we tweak the sentence, the model understands interest is now being used in a financial sense.

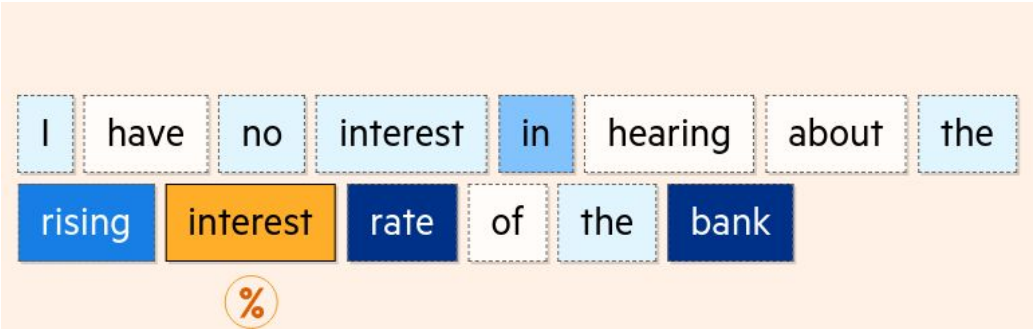
Self-Attention

When we combine the sentences, the model is still able to recognise the correct meaning of each word thanks to the attention it gives the accompanying text.

For the first use of interest, it is no and in that are most attended.

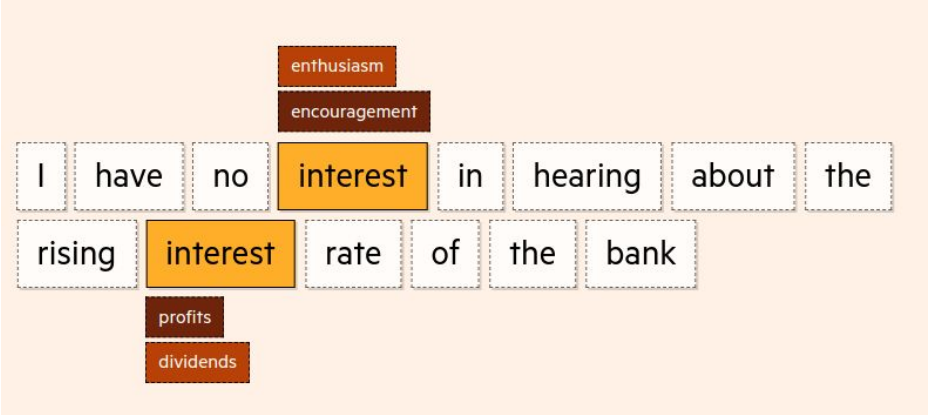


For the second, it is rate and bank.

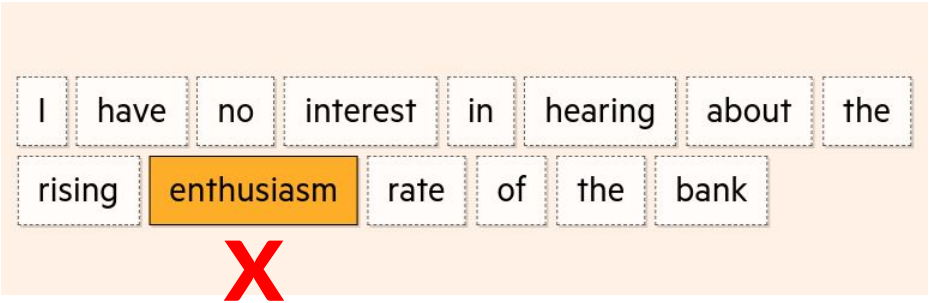


Self-Attention

This functionality is crucial for advanced text generation. Without it, words that can be interchangeable in some contexts but not others can be used incorrectly.



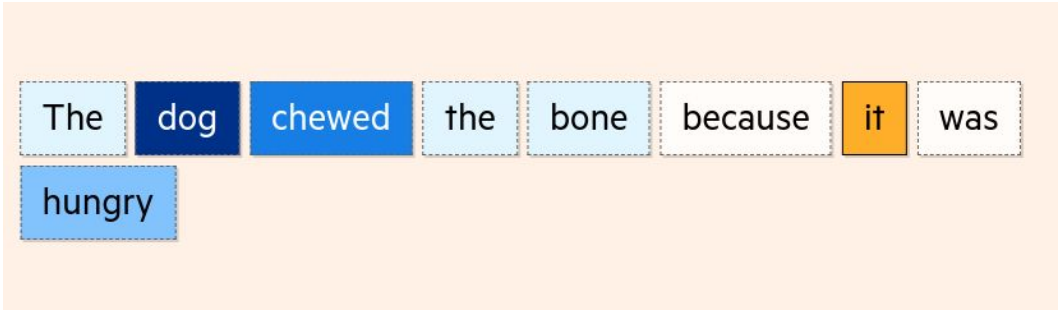
Effectively, self-attention means that if a summary of this sentence was produced, you wouldn't have enthusiasm used when you were writing about interest rates.



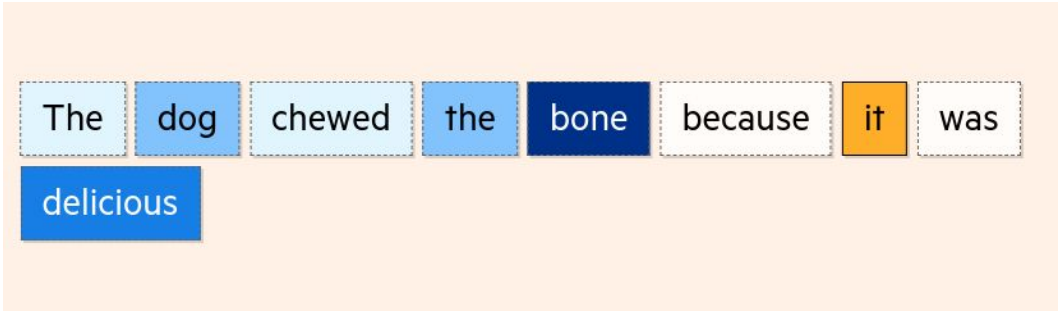
Self-Attention

This capability goes beyond words, like interest, that have multiple meanings.

Self-attention is able to calculate that it is most likely to be referring to dog.

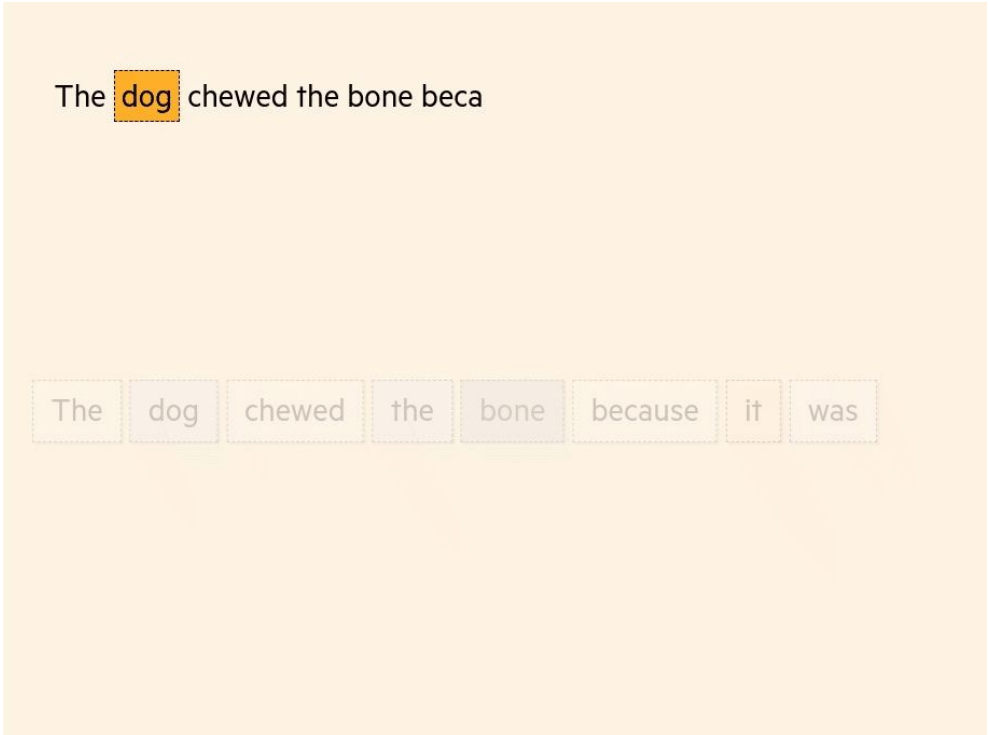


And if we alter the sentence, swapping hungry for delicious, the model is able to recalculate, with it now most likely to refer to bone.



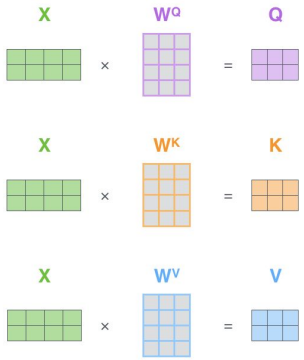
Self-Attention

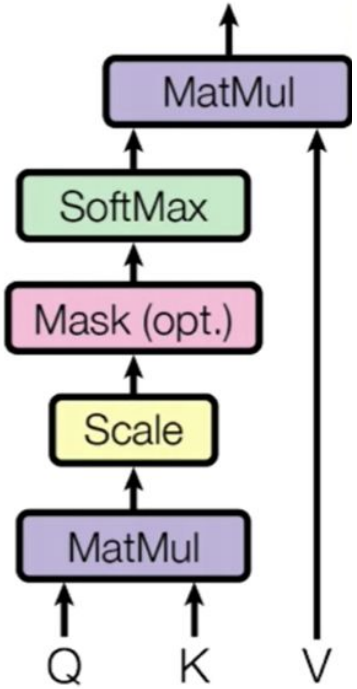
The benefits of self-attention for language processing increase the more you scale things up. It allows LLMs to take context from beyond sentence boundaries, giving the model a greater understanding of how and when a word is used.



Self Attention

- Each token in the input sequence attend to every other word in the sequence;
- Richer representation of the word by considering the entire context.
- For each word we generate three vectors:
  - **Query (Q)**: Represents the current word.
  - **Key (K)**: Represents the words to which the current word will attend.
  - **Value (V)**: Represents the information the current word will extract from the words it attends to.



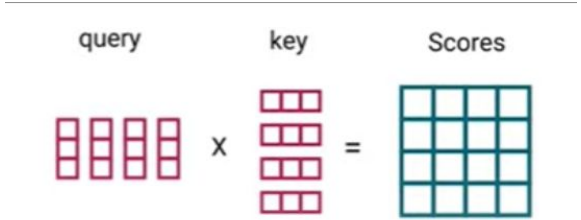


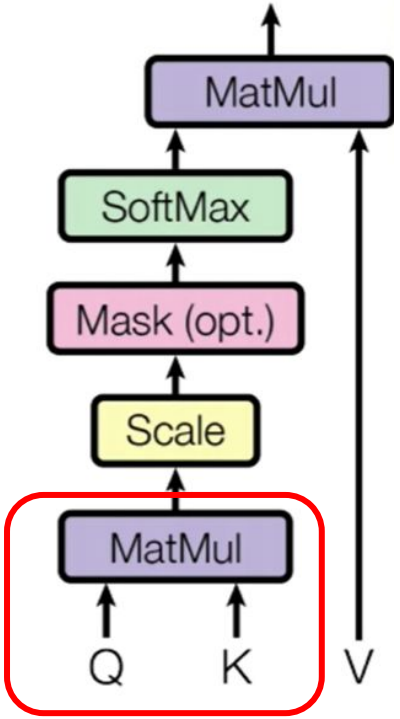
Attention scores

The **attention score** for a given word (query) attending to another word (key) is computed as the **dot product** of the query and key vectors.

This score determines how much focus (weight) the model should give to each word when computing the output.

$$\text{Attention Score} = Q \cdot K^T$$

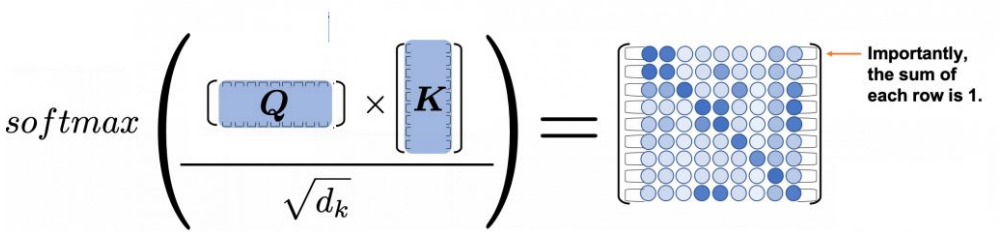


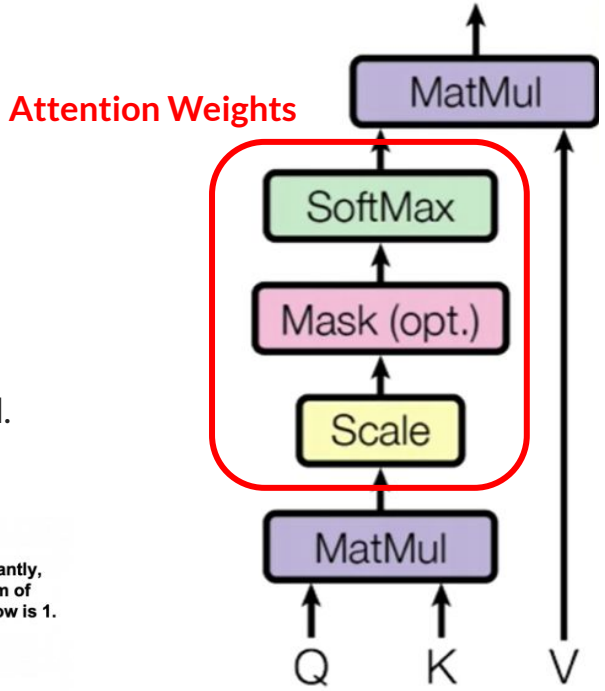


Attention Score

Attention Weights

- Scaled down the scores - more stable gradients to avoid exploding effects of multiplication.
- Softmax to get the **attention weights** (probability [0, 1])
- Higher scores get enhanced, and lower scores are depressed.
- Model learns which words to attend to.

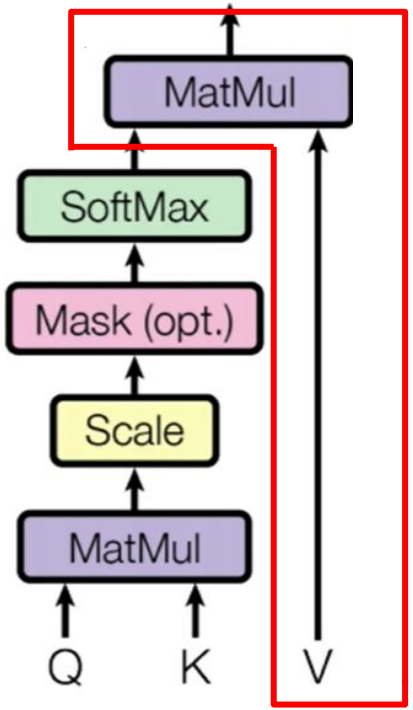




Self Attention

- **Attention weights** * **Value (V)**
- The higher softmax scores will keep the value of words the model learns is **more important**. The lower scores will drown out the **irrelevant** words.

$$\text{Attention}(Q, K, V) = \texttt{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V$$



Attention weights * Value (V)

ORIGINAL VIEW

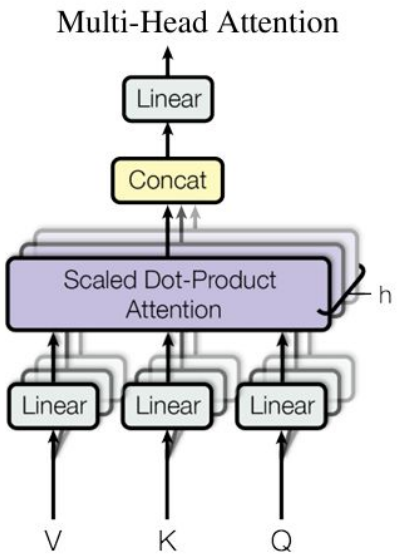
FOCUSED VIEW

Attention weights * Value (V)

Multi-Head Attention

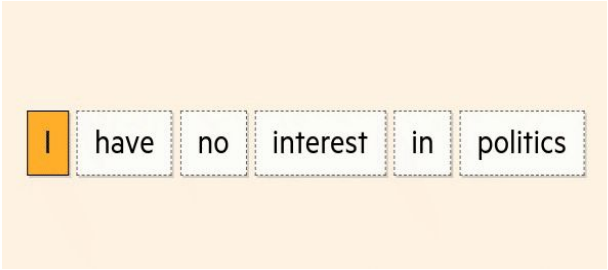
- Multiple sets of Query/Key/Value weight matrices (8 by default)
- Each set is randomly initialized (during training).
- Then, after training, each set is used to project the input embeddings into a different representation subspace.

Multiple heads allow the model to **look at different parts of the sentence** from **different perspectives**

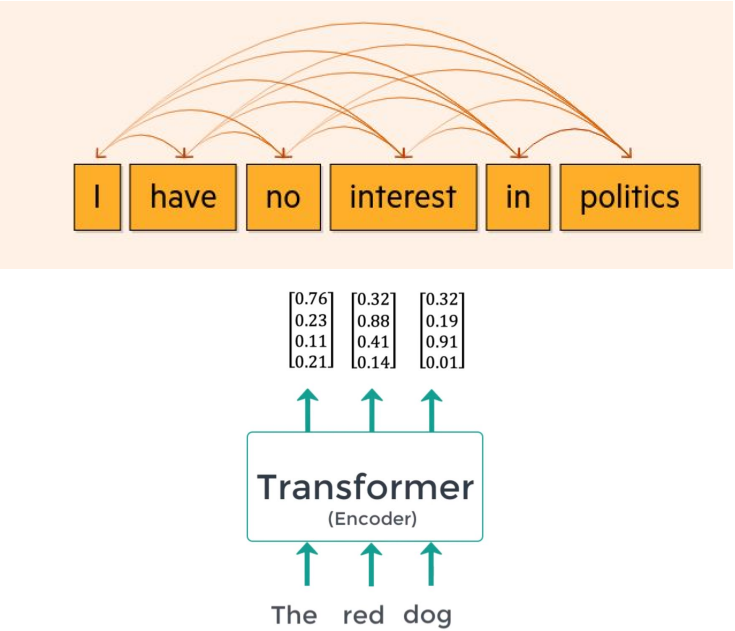


Parallelization

RNNs

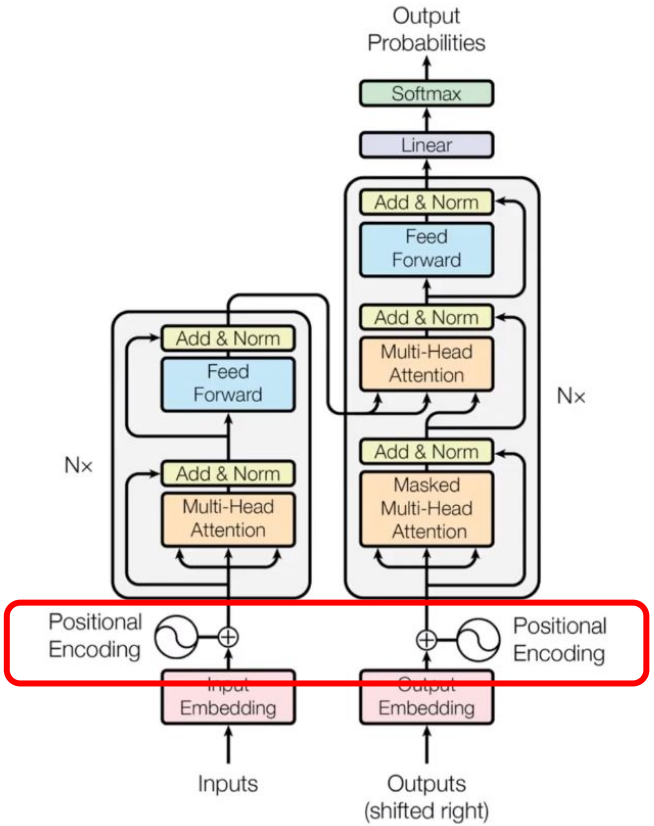


Transformer

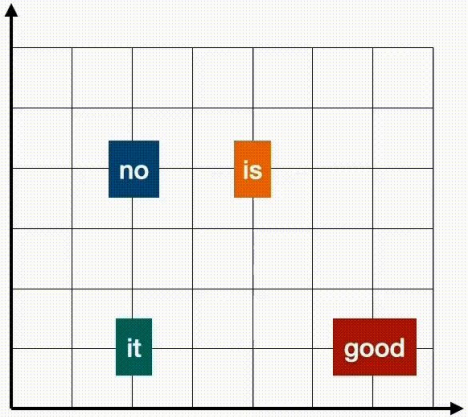


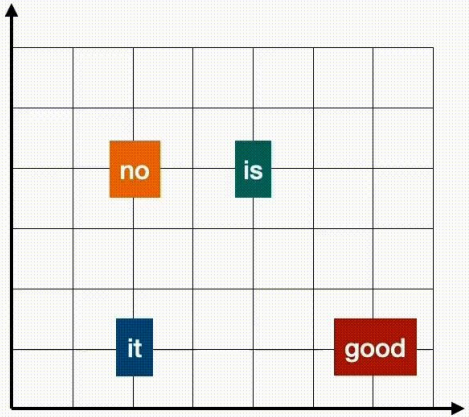
- Unlike **RNNs** or **LSTMs**, which process inputs sequentially, the **Transformer** processes all words in a sentence simultaneously.

Positional Encoding



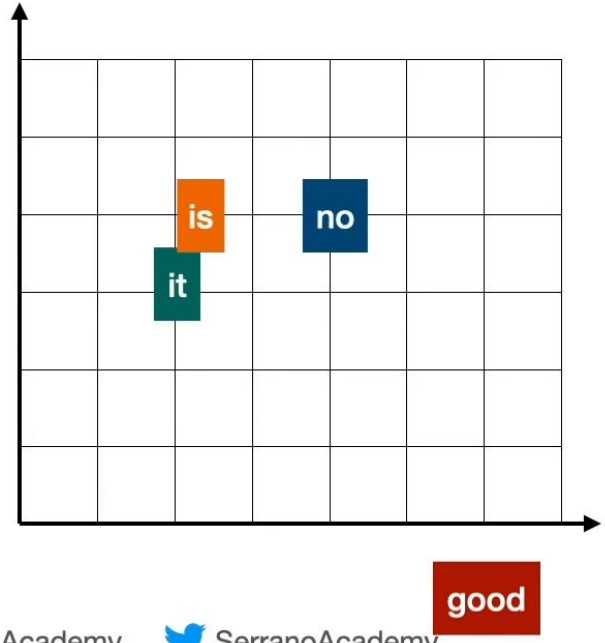
Positional encoding



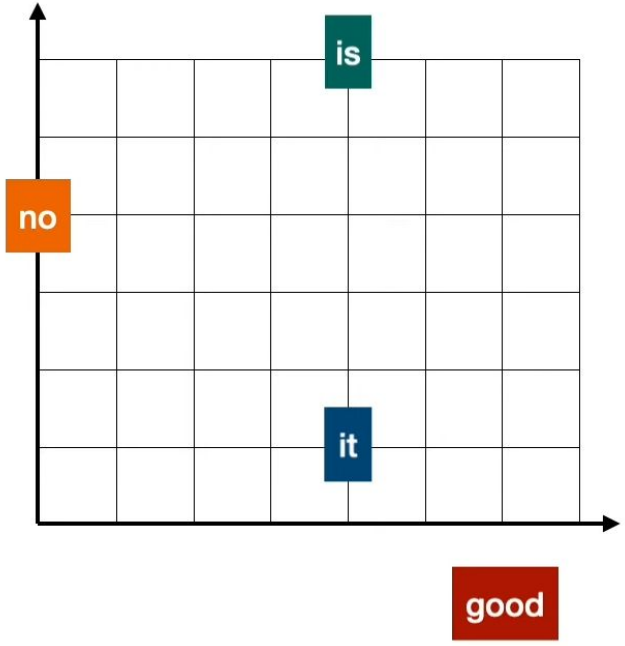


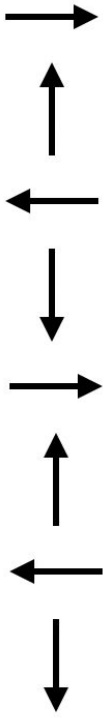
Positional encoding

Repeats after the 4th word



different!





Positional Encodings

“We only need to have a consistent way of changing the words”

Key Ideas:

- Each position gets a **unique**, consistent vector.
- Small shifts in original vectors ensure semantic stability.
- The encoding method must be simple and easy to learn

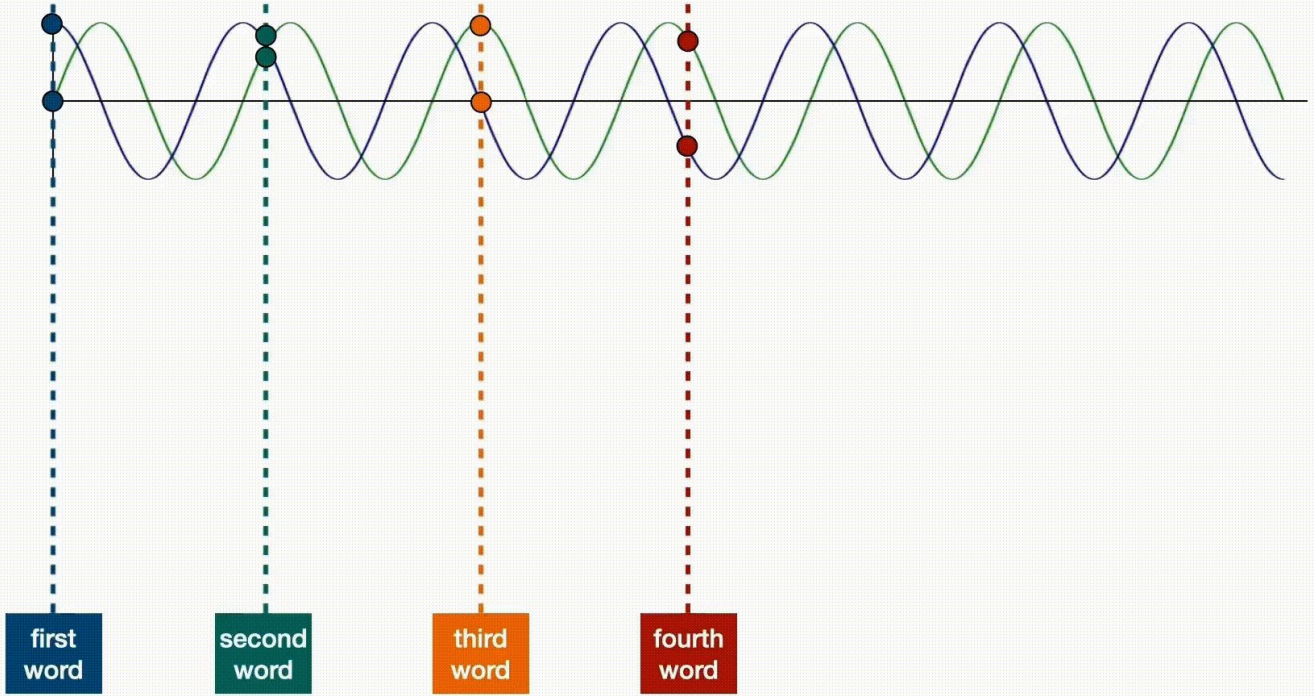
Note: Neural networks are **spurious correlation identifying beasts**.
The model sees the pattern so often during training that it learns to decode it.

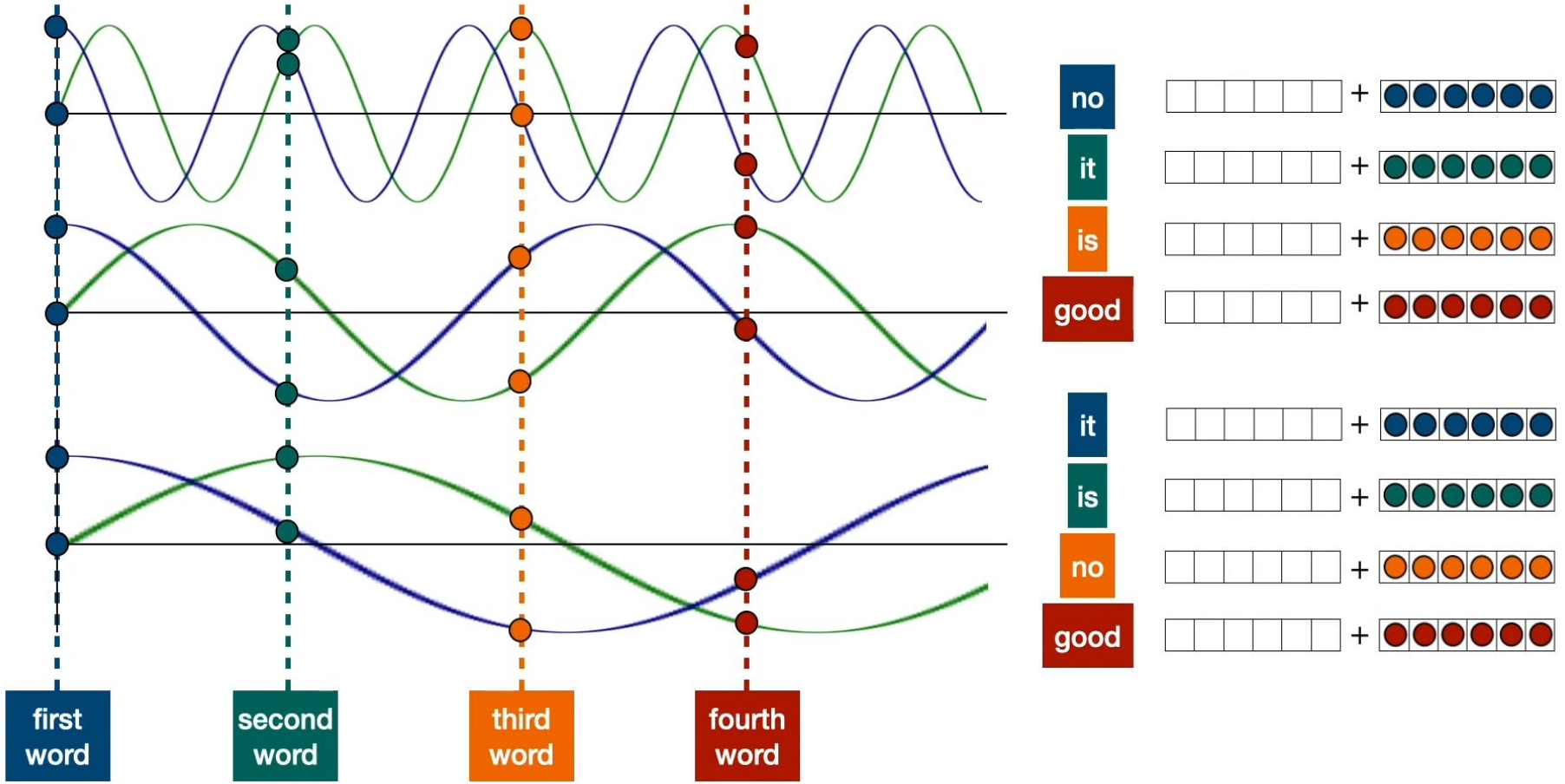
Positional Encoding

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- sine and cosine functions (sinusoidal functions) to generate the positional encodings
  - `pos`: token position (0, 1, 2, ...)
  - `i`: dimension index
  - d: embedding dimension

Larger embeddings?

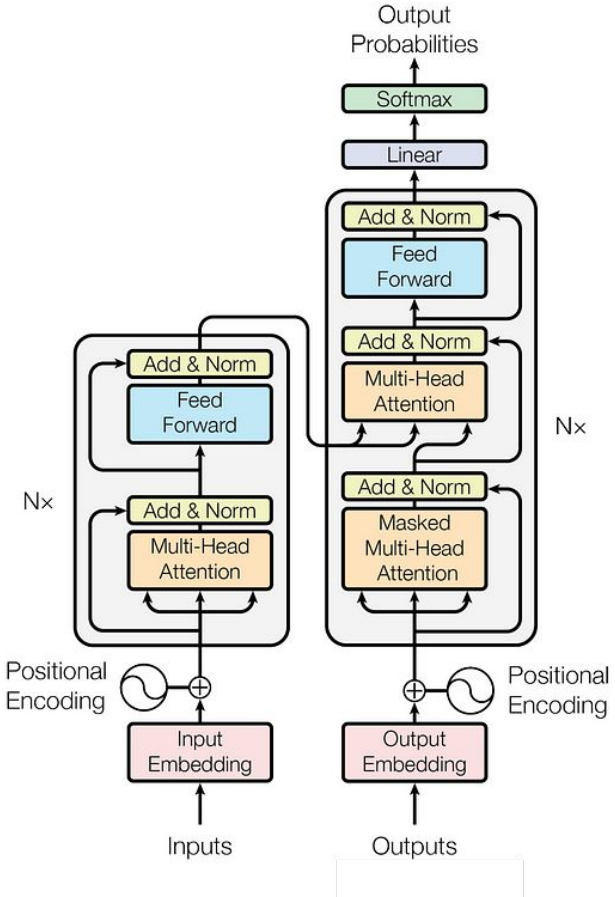



first word
second word
third word
fourth word
no
it
is
good
it
is
no
good

Transformer Models

BERT

Encoder



GPT

Decoder

Transformer Models

Computer Science > Computation and Language

[Submitted on 11 Oct 2018 (v1), last revised 24 May 2019 (this version, v2)]

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova

We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

Subjects: **Computation and Language (cs.CL)**
Cite as: arXiv:1810.04805 **[cs.CL]**
(or arXiv:1810.04805v2 **[cs.CL]** for this version)

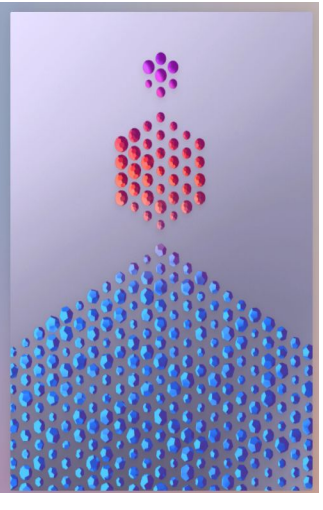
Submission history

From: Ming-Wei Chang [view email]
[v1] Thu, 11 Oct 2018 00:50:01 UTC (227 KB)
[v2] Fri, 24 May 2019 20:37:26 UTC (309 KB)

Improving Language Understanding with Unsupervised Learning

We've obtained state-of-the-art results on a suite of diverse language tasks with a scalable, task-agnostic system, which we're also releasing. Our approach is a combination of two existing ideas: transformers and unsupervised pre-training. These results provide a convincing example that pairing supervised learning methods with unsupervised pre-training works very well; this is an idea that many have explored in the past, and we hope our result motivates further research into applying this idea on larger and more diverse datasets.

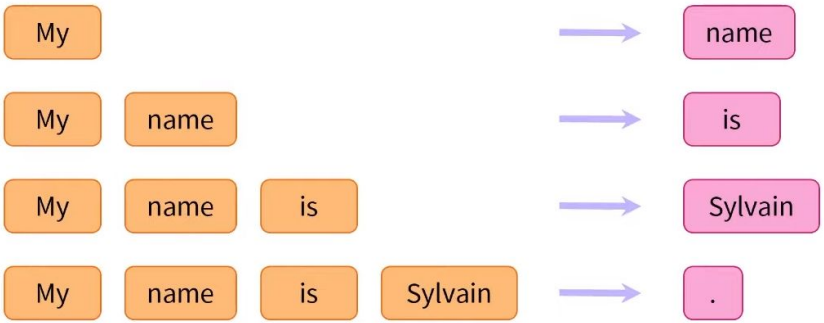
June 11, 2018
9 minute read



Pre-Training tasks

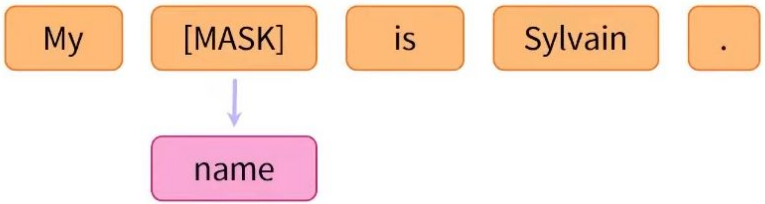
Auto-Regressive Language Modeling - GPT

- Predict the **next word** given all previous words.
- It only looks **to the left** — past tokens.



Masked Language Modeling (MLM) - BERT

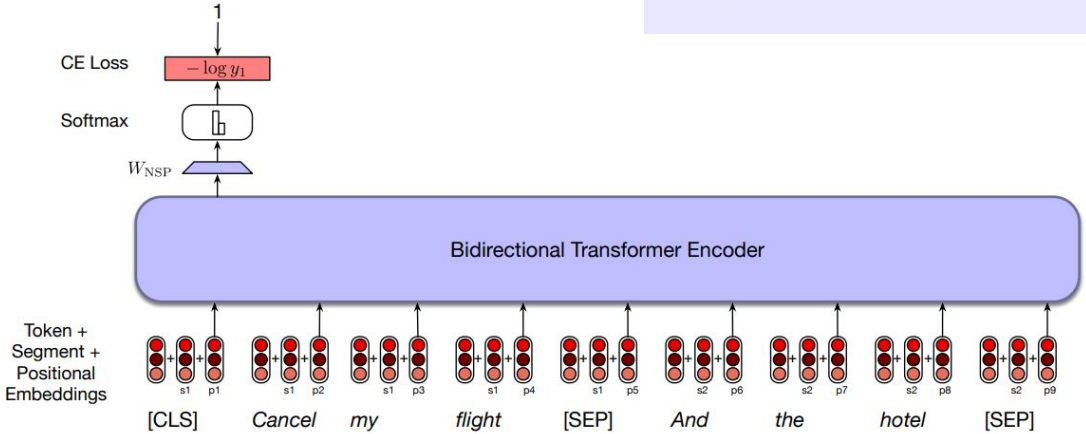
- Randomly mask out some tokens from the input.
- The model must **predict the masked tokens** based on the context.



Pre-Training tasks

Next Sentence Prediction (NSP) - BERT

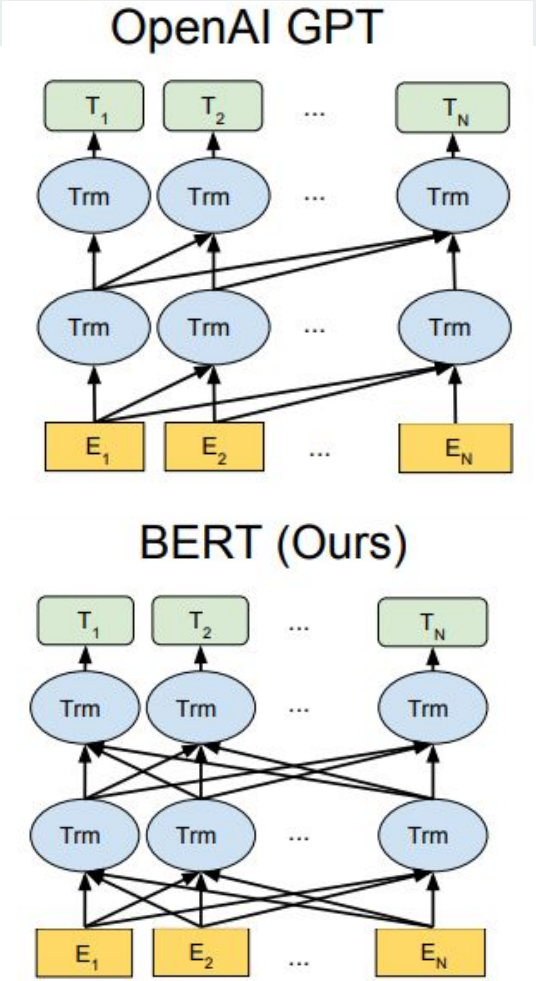
- Given two sentences, predict whether the second one actually follows the first one in the text.



During training, the output vector from the final layer associated with the [CLS] token represents the next sentence prediction. A learned set of classification weights is used to produce a two-class prediction from the raw [CLS] vector

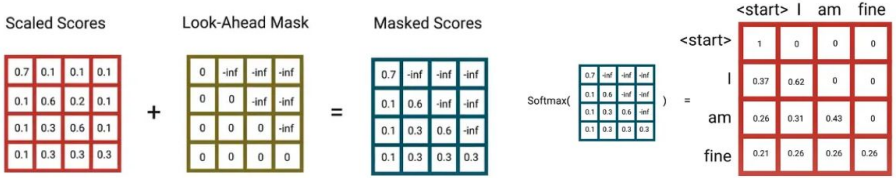
BERT vs GPT

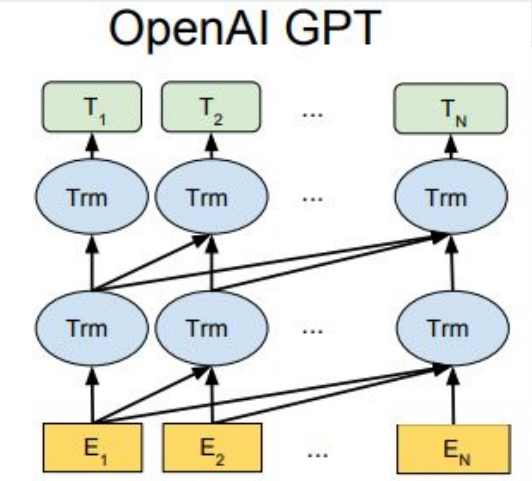
- GPT is a causal or left-to-right model
  - applied to problems such as summarization and machine translation
- Not useful in tasks such as named-entity tagging
  - require information from the right context as we process each element.
- BERT representations are jointly conditioned on both left and right context in all layers

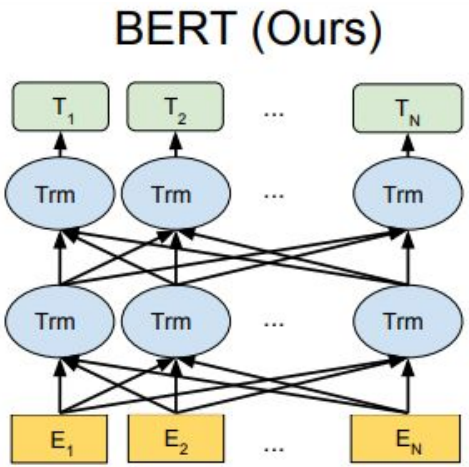


BERT vs GPT

- Mask Self-Attention
- GPT doesn't see the full input at once (unlike BERT). Instead, it only uses past tokens to predict the next.
- By seeing the “future” tokens, the model could cheat by seeing the actual token it's supposed to predict. For example, while predicting “world” in “Hello world”, it could just look ahead at “world”.



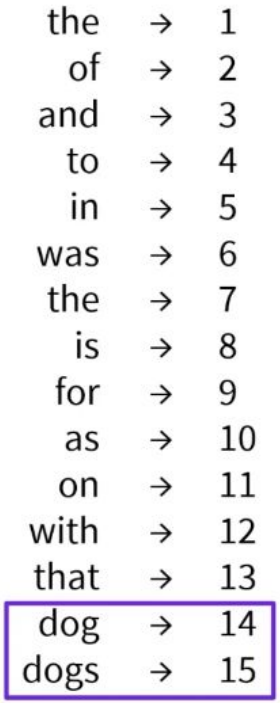




Tokenization

Word-based Tokenizer

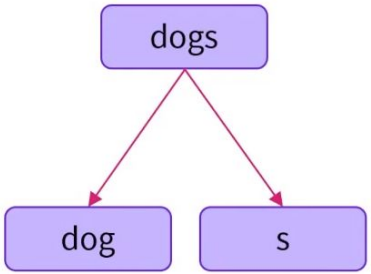
- Very similar words have entirely different meanings
- Large vocabulary - heavy
- Unknown words
- Lost of informations

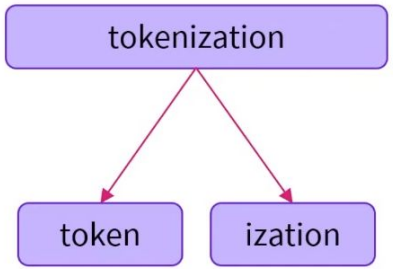


Subword-based Tokenizer

Frequently used words should not be split into smaller subwords

Rare words should be decomposed into meaningful subwords.





Token
Tokens
Tokenizing
Tokenization

Tokenization
Modernization
Immunization

Generating text with Decoder models

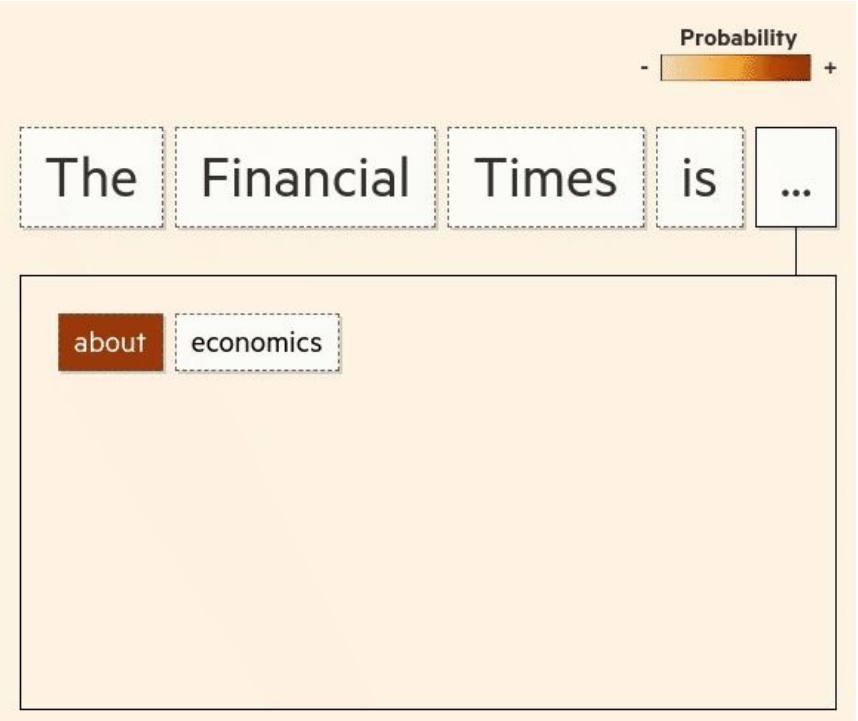
The model's aim is now to predict the next word in a sequence and do this repeatedly until the output is complete.

To do this, the model gives a probability score to each token, which represents the likelihood of it being the next word in the sequence.

And it continues to do this until it is happy with the text it has produced.

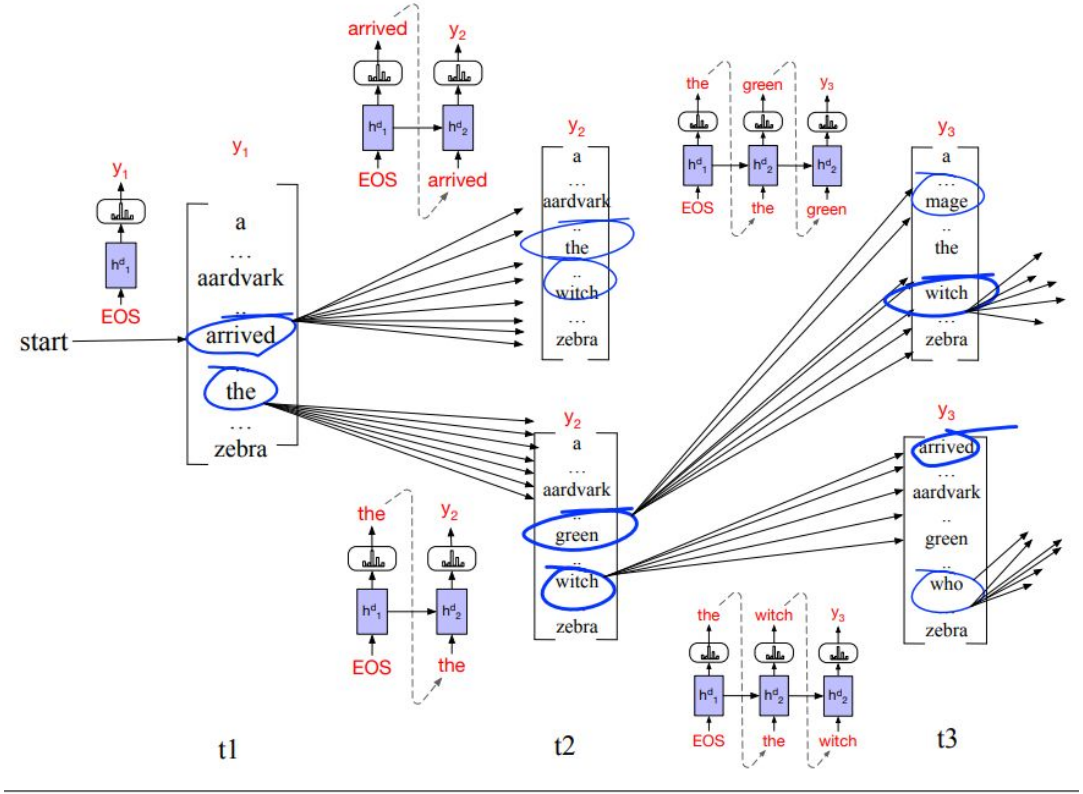
Problem: Predicting the following word in isolation (“greedy search”) can introduce problems. Sometimes, while each individual token might be the next best fit, the full phrase can be less relevant.

Solution: Rather than focusing only on the next word in a sequence, it looks at the probability of a larger set of tokens as a whole. (Beam Search)

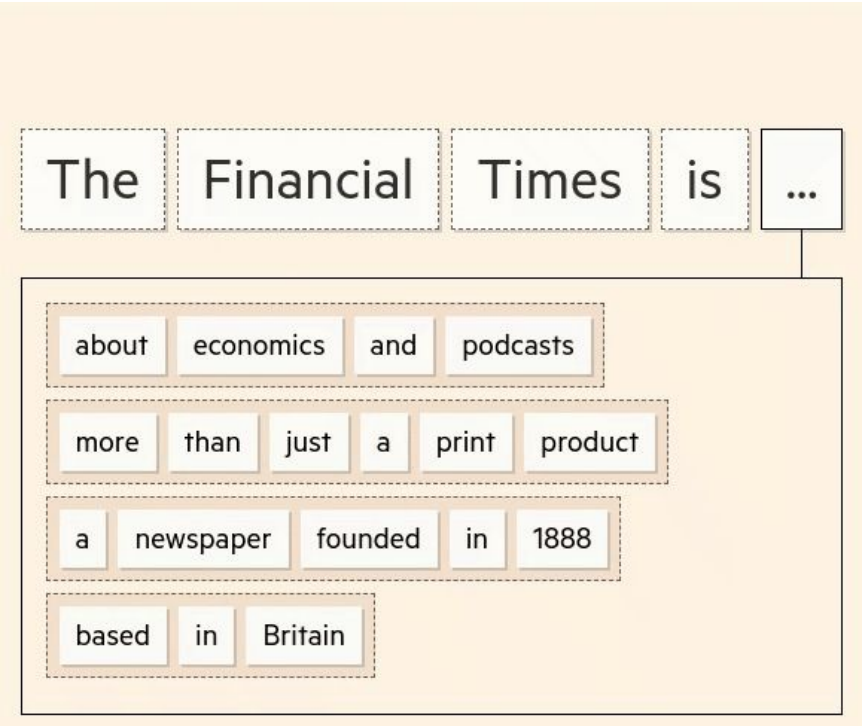


At each time step, we choose the k best hypotheses, compute the V possible extensions of each hypothesis, score the resulting k ∗V possible hypotheses and choose the best k to continue.

At time 1, the frontier is filled with the best 2 options: *arrived* and *the*. We then extend each of those, compute the probability of all the hypotheses so far (arrived the, arrived witch, the green, the witch) and compute the best 2 (in this case the green and the witch) to be the search frontier to extend on the next step.



With beam search, the model is able to consider multiple routes and find the best option, producing better and more coherent results



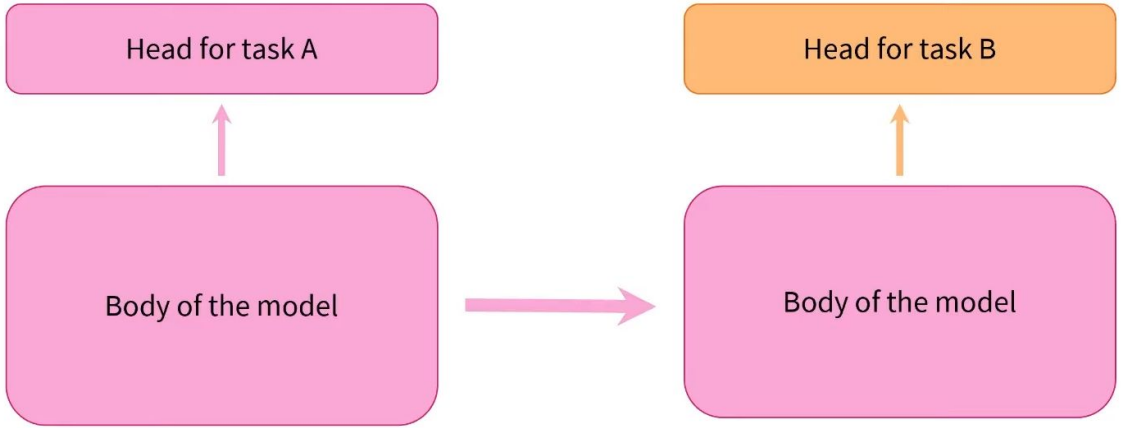
Fine-Tunning BERT

Fine-Tuning LLMS -Transfer Learning

Drop last layers that focus on the pretraining objective

Add a new head in order to fine tune the model to the new classification problem

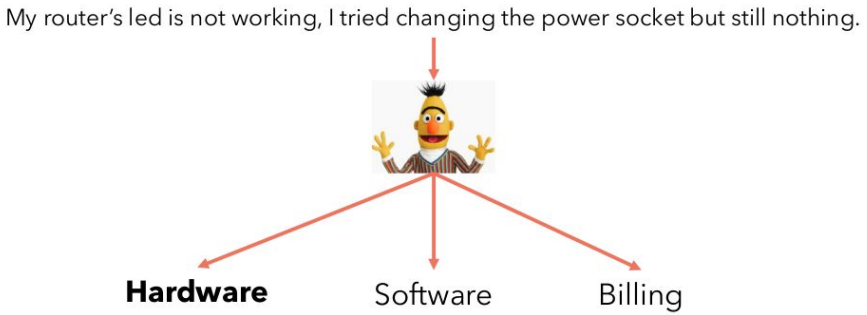
Train -> **huge dataset**
Fine-tune -> **small dataset**

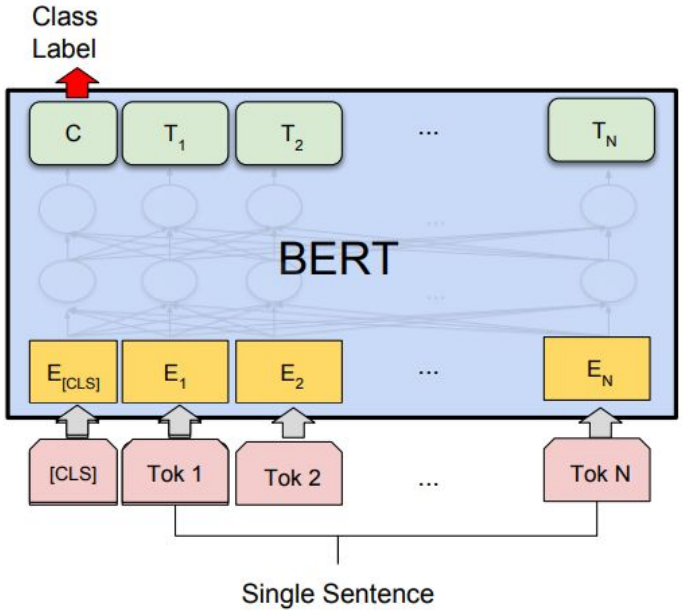


BERT - Pretrained on English wikipedia and 11k books

Text Classification

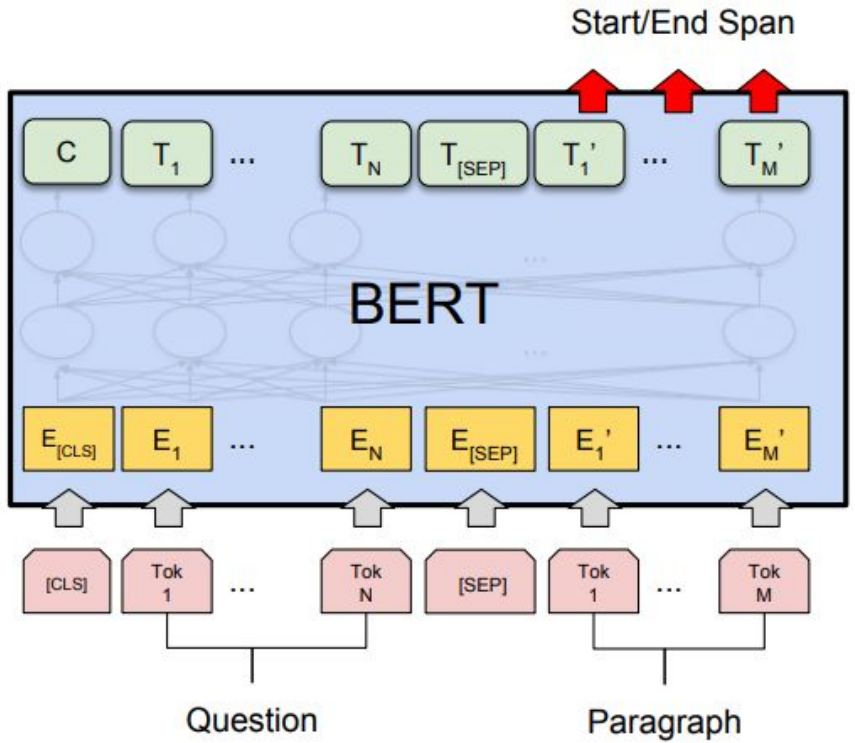
Classification tasks such as sentiment analysis are done similarly to Next Sentence classification, by adding a classification layer on top of the Transformer output for the [CLS] token.





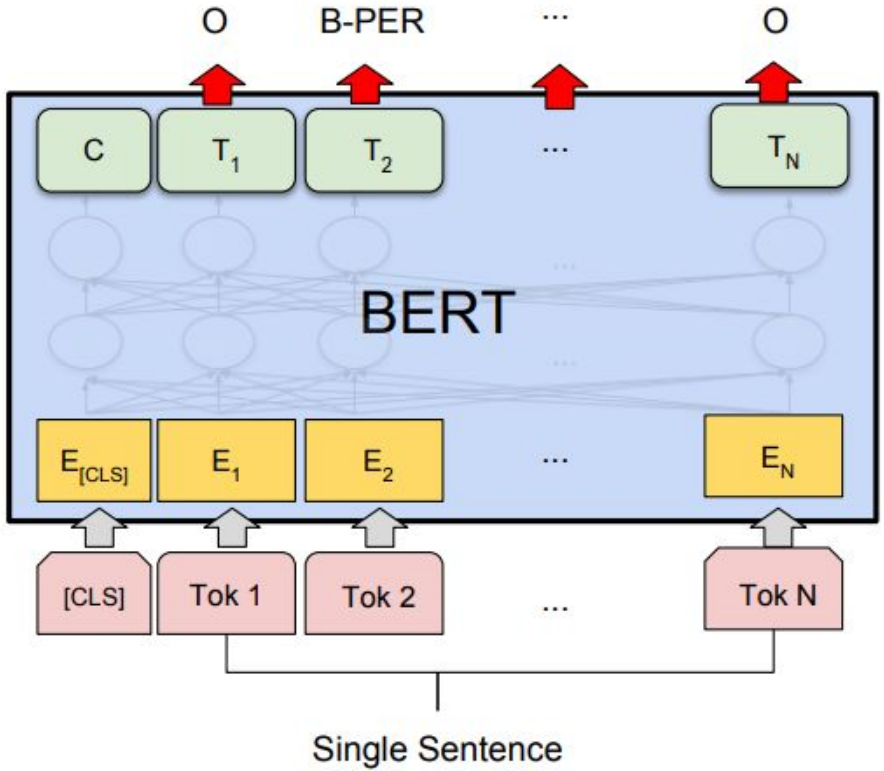
Question Answering

In **Question Answering tasks**, the software receives a question regarding a text sequence and is required to mark the answer in the sequence. Using BERT, a Q&A model can be trained by learning two extra vectors that mark the beginning and the end of the answer.

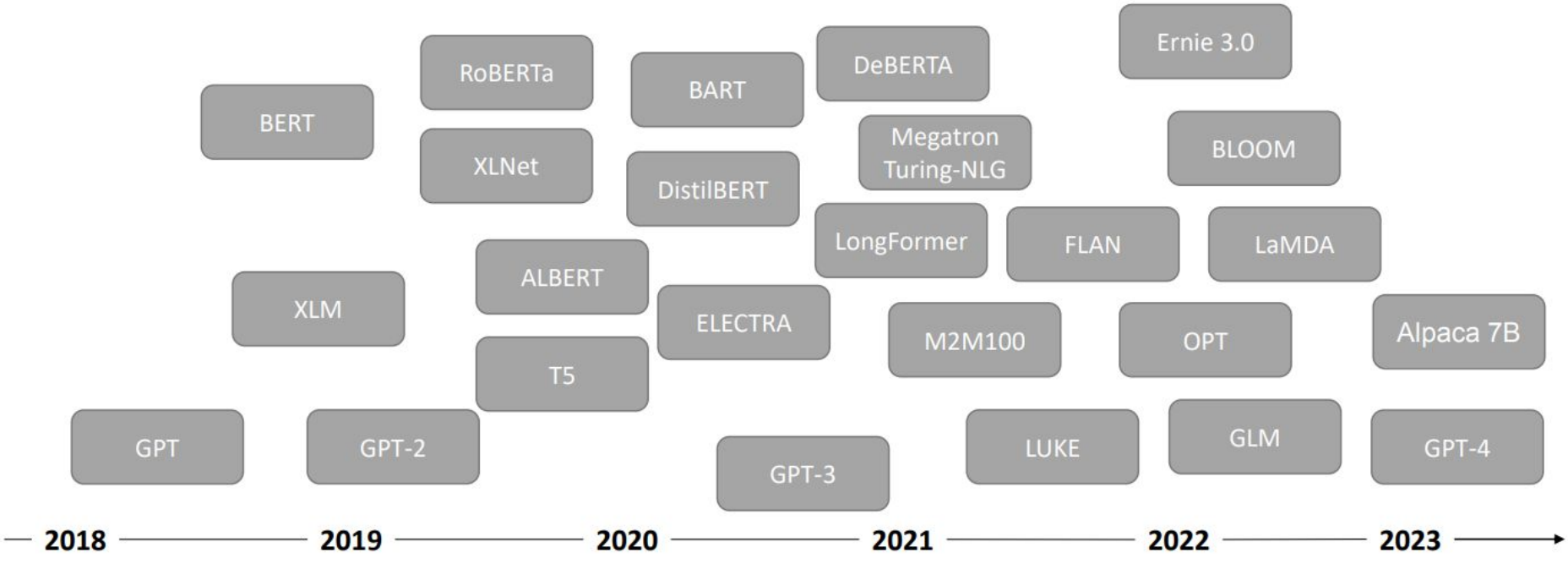


Fine-Tuning BERT - Named Entity Recognition

In **Named Entity Recognition (NER)**, the software receives a text sequence and is required to mark the various types of entities (Person, Organization, Date, etc) that appear in the text. Using BERT, a NER model can be trained by feeding the output vector of each token into a classification layer that predicts the NER label.

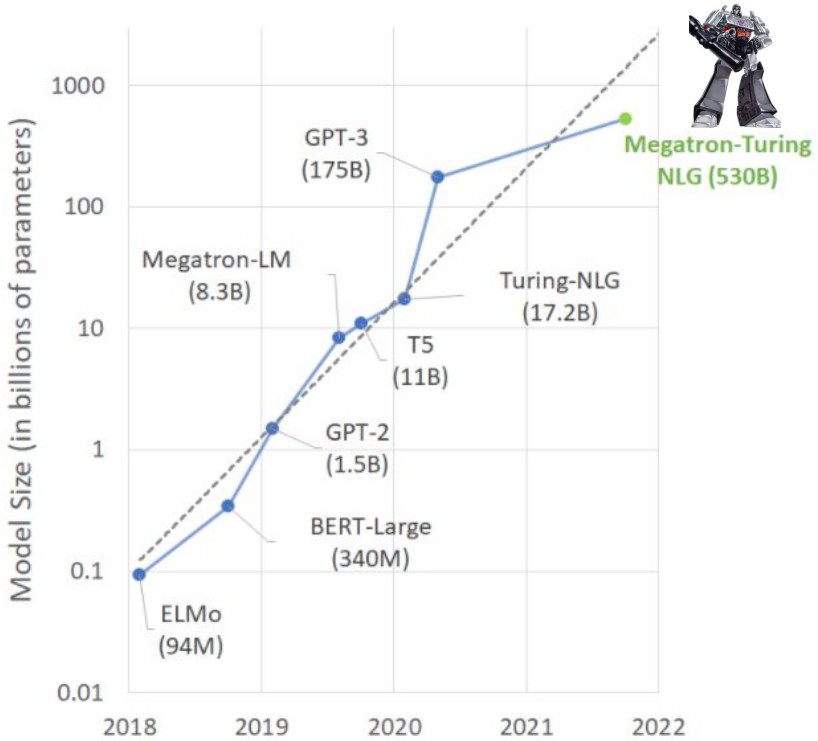


Large Language Models
Where are we going?


Ernie 3.0
RoBERTa
DeBERTA
BART
BERT
Megatron
Turing-NLG
BLOOM
XLNet
DistilBERT
LongFormer
FLAN
LaMDA
ALBERT
XLM
ELECTRA
M2M100
OPT
Alpaca 7B
T5
GLM
GPT
GPT-2
LUKE
GPT-4
GPT-3
2018
2019
2020
2021
2022
2023

Megatron-Turing Natural Language Generation

- Microsoft and Nvidia, October 2021
- Trained with 4480 A100 80GB GPUs
- 15 datasets consisting of a total of 339 billion tokens.



- Mistral - 7 billion parameters.
- LLaMA 3 - 70 billion parameters.
- PaLM - 540 billion parameters.
- GPT-4 is rumoured to have around 1.76 trillion parameters.
- Etc..

A larger number of parameters doesn't always guarantee better performance.

Fine-Tunning BERT

Named Entity Recognition

Transformers and Datasets

```
! pip install datasets transformers seqeval evaluate

import transformers

model_checkpoint = "neuralmind/bert-base-portuguese-cased"
batch_size = 16

from datasets import load_dataset
datasets = load_dataset("lfcc/portuguese_ner")
print(datasets)

DatasetDict({
    train: Dataset({
        features: ['tokens', 'ner_tags'],
        num_rows: 3716
    })
    test: Dataset({
        features: ['tokens', 'ner_tags'],
        num_rows: 930
    })
})
```

NER Labels

```
label_list = datasets["train"].features["ner_tags"].feature.names
print(label_list)
```

```
['B-Data', 'B-Local', 'B-Organizacao', 'B-Pessoa', 'B-Profissao', 'I-Data', 'I-Local',
'I-Organizacao', 'I-Pessoa', 'I-Profissao', 'O']
```

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
```

Tokenization

```
tokens = ["As", "aulas", "de", "NLP", "são", "interessantes", "!"]
tokenized_input = tokenizer(tokens, is_split_into_words=True)
print(tokenized_input)
```

```
{'input_ids': [101, 510, 6880, 125, 248, 18353, 453, 20764, 106, 102], 'token_type_ids': [0,
0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

```
new_tokens = tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
print(tokens)
['[CLS]', 'As', 'aulas', 'de', 'N', '##LP', 'são', 'interessantes', '!', '[SEP]']
```

```
len(tokens), len(new_tokens)
7, 10
```

```
tokenized_input.word_ids()
[None, 0, 1, 2, 3, 3, 4, 5, 6, None]
```

```
[0,0,0,4,0,0,0]
[
```

Token and Labels Alignment

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples[ "tokens"], truncation=True, max_length=512, is_split_into_words=True)

    labels = []
    for i, label in enumerate(examples["ner_tags"]):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx =  None
        label_ids = []
        for word_idx in word_ids:
            # Special tokens have a word id that is None. We set the label to -100 so they are automatically
            # ignored in the loss function.
            if word_idx is None:
                label_ids.append(- 100)
            # We set the label for the first token of each word.
            elif word_idx != previous_word_idx:
                label_ids.append(label[word_idx])
            # For the other tokens in a word, we set the label to either the current label or -100, depending on
            # the label_all_tokens flag.
            else:
                label_ids.append(- 100)
            previous_word_idx = word_idx

        labels.append(label_ids)

    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

Generating pre-processed dataset

```
tokenized_datasets = datasets.map(tokenize_and_align_labels, batched=True)
print(tokenized_datasets )

DatasetDict({
    train: Dataset({
        features: ['tokens', 'ner_tags', 'input_ids', 'token_type_ids', 'attention_mask',
'labels'],
        num_rows: 3716
    })
    test: Dataset({
        features: ['tokens', 'ner_tags', 'input_ids', 'token_type_ids', 'attention_mask',
'labels'],
        num_rows: 930
    })
})
```

```
from transformers import AutoModelForTokenClassification, TrainingArguments, Trainer

id2label = {i: label for i, label in enumerate(label_list)}
label2id = {label: i for i, label in enumerate(label_list)}

model = AutoModelForTokenClassification.from_pretrained(model_checkpoint,num_labels=len(label_list),
                                                        id2label=id2label, label2id=label2id)

output_model_name = "my_model_"
args = TrainingArguments(
    output_model_name,
    report_to="none",
    eval_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size, #16
    per_device_eval_batch_size=batch_size, #16
    num_train_epochs=3,
    weight_decay=0.01,
    save_strategy="epoch",
    load_best_model_at_end=True,      # Load the best model after training
    metric_for_best_model="f1",       # Or any metric you use for evaluation
    greater_is_better=True,           # True if higher F1/Accuracy is better
    #save_total_limit=1
    #push_to_hub=True,
)
```

Evaluation metrics

```
import evaluate
import numpy as np
metric = evaluate.load("seqeval")

def compute_metrics(p):
    print(p)
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    # Remove ignored index (special tokens)
    true_predictions = [
        [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]

    results = metric.compute(predictions=true_predictions, references=true_labels)
    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
```

Input with fixed size (padding)

```
from transformers import DataCollatorForTokenClassification

data_collator = DataCollatorForTokenClassification(tokenizer)
```

Model Trainer

```
trainer = Trainer(
    model,
    args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

trainer.train() # saves model according to args save strategy
trainer.evaluate()
trainer.save_model("model_name") # save manually
```

Evaluation

```
trainer.evaluate()
```

```
{'eval_loss': 0.06827212870121002,
 'eval_precision': 0.9443929564411492,
 'eval_recall': 0.9677113010446344,
 'eval_f1': 0.9559099437148217,
 'eval_accuracy': 0.9840616516332429,
 'eval_runtime': 6.8671,
 'eval_samples_per_second': 135.429,
 'eval_steps_per_second': 8.592,
 'epoch': 3.0}
```

Evaluation for each label

```
predictions, labels, _ = trainer.predict(tokenized_datasets["test"])
predictions = np.argmax(predictions, axis=2)

# Remove ignored index (special tokens)
true_predictions = [
    [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
    for prediction, label in zip(predictions, labels)
]
true_labels = [
    [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
    for prediction, label in zip(predictions, labels)
]

results = metric.compute(predictions=true_predictions, references=true_labels)
```

Inference

```
from transformers import pipeline

ner_pipeline = pipeline("token-classification", model=model, tokenizer= tokenizer,
aggregation_strategy="first")

# Run inference
text = """O João, médico, vive no Porto (Portugal) desde 2024.
A Soraia obeteve o seu mestrado na Universidade do Minho em 26 de Maio de 2021.
Depois de obter o grau, foi trabalhar para o o tribunal de Braga como juíza."
"""
results = ner_pipeline(text)

# Display results
for entity in results:
    print(f"{entity['word']} -> {entity['entity_group']} ({entity['score']:.3f})")
```

Transformers

- Context Matters

Luís Filipe Cunha
lfc@di.uminho.pt

José João Almeida
jj@di.uminho.pt