Applied Transformers

Dott. Lorenzo Vaiani
Dipartimento di Automatica e Informatica
Politecnico di Torino

Politecnico
di Torino

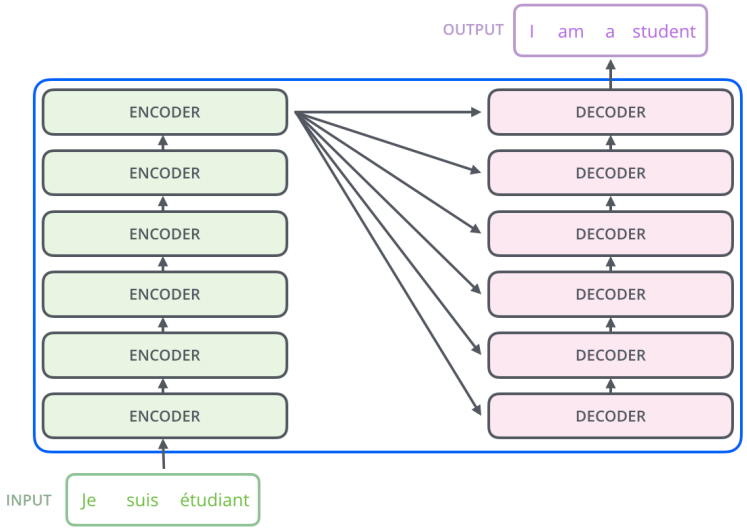
Lecture goals

- **Transformer architectures (recap)**
  - Self- and Cross-Attention
  - Encoder/Decoder/Sequence-to-Sequence
  - Architectures (BERT, GPT, BART)
- Exercises using Transformers
- Task-specific Transformers (exercises)
  - Sequence classification
  - Entity recognition
- From development to production
  - Serving model for AI applications

Transformer architecture

Proposed in 2017 introduces a major breakthrough in NLP research community.

- Self-attention mechanism
- Cross-attention mechanism
- Positional Encoding



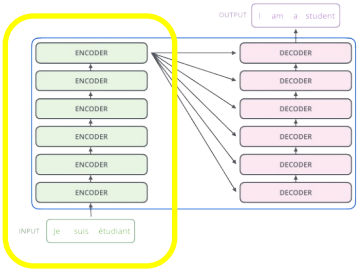
https://jalammar.github.io/illustrated-transformer/

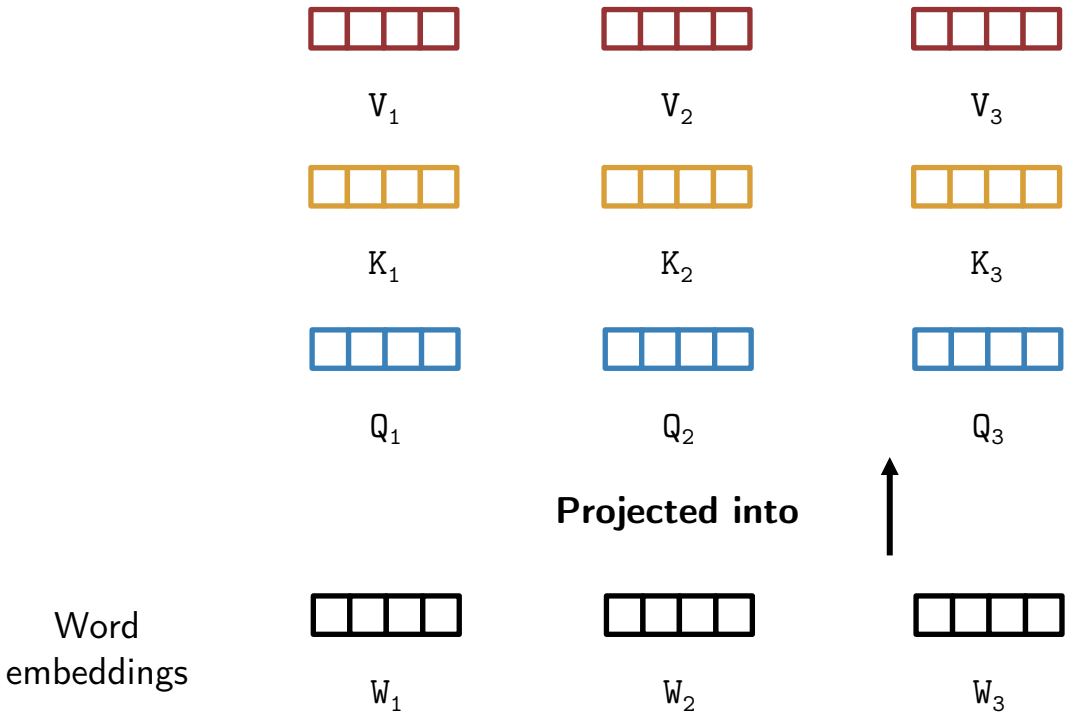
It has been originally proposed for Machine translation tasks.

Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
(November 2021, **30,000+ citations**)

Self-Attention mechanism

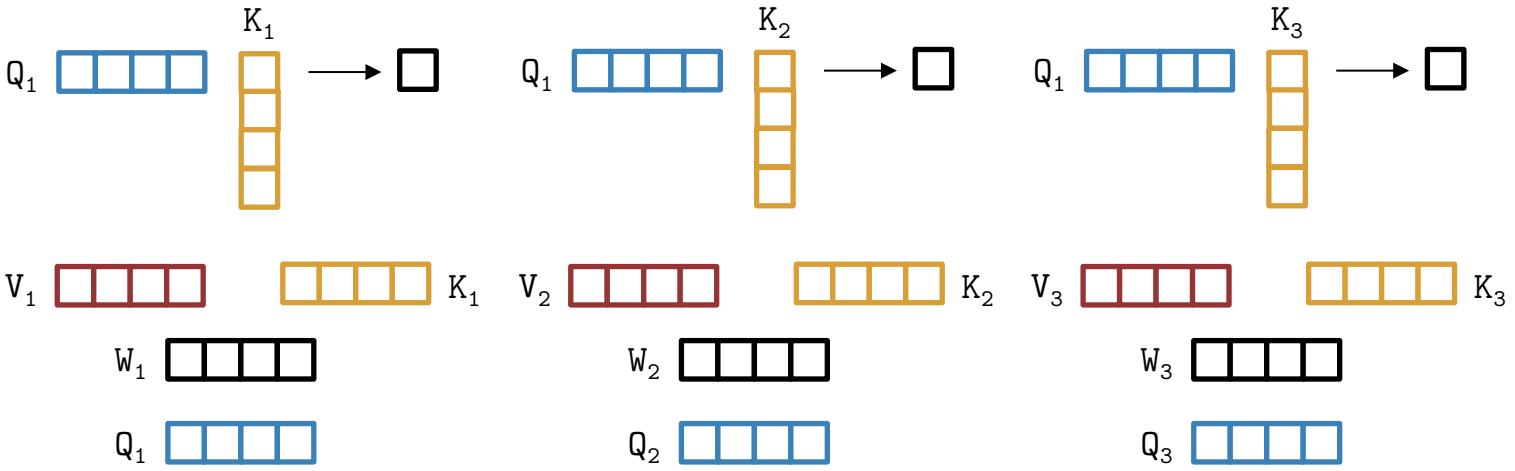
It is part of the encoder component.





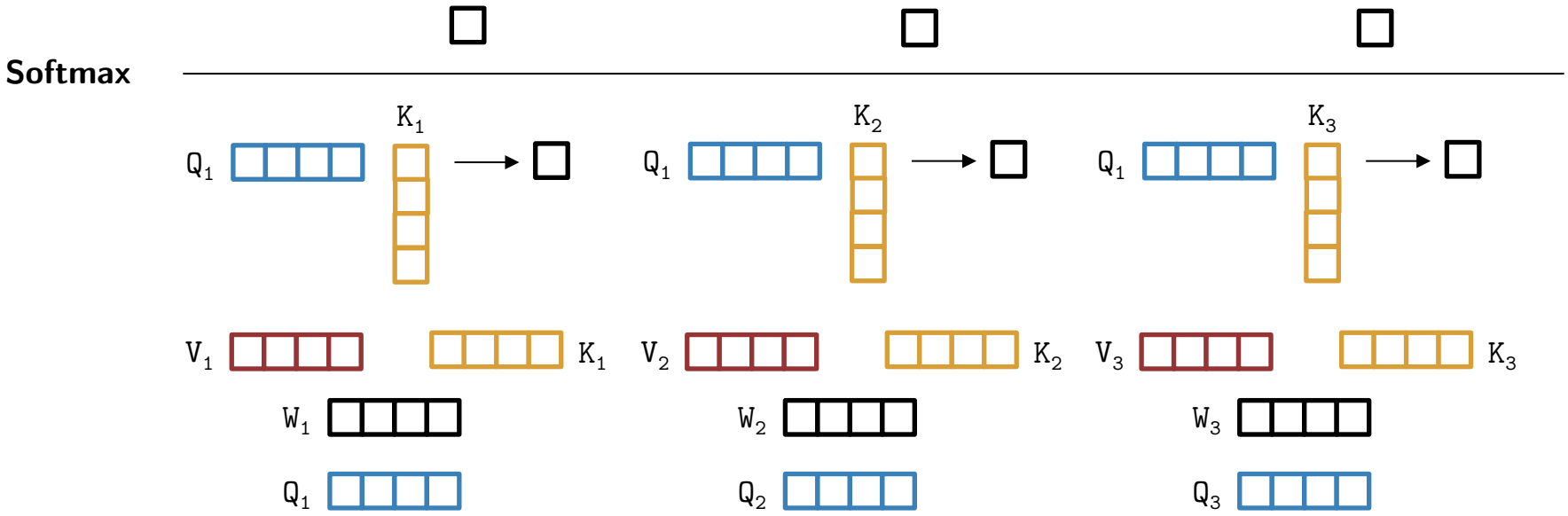
Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017. (November 2021, **30,000+ citations**)

Self-Attention mechanism



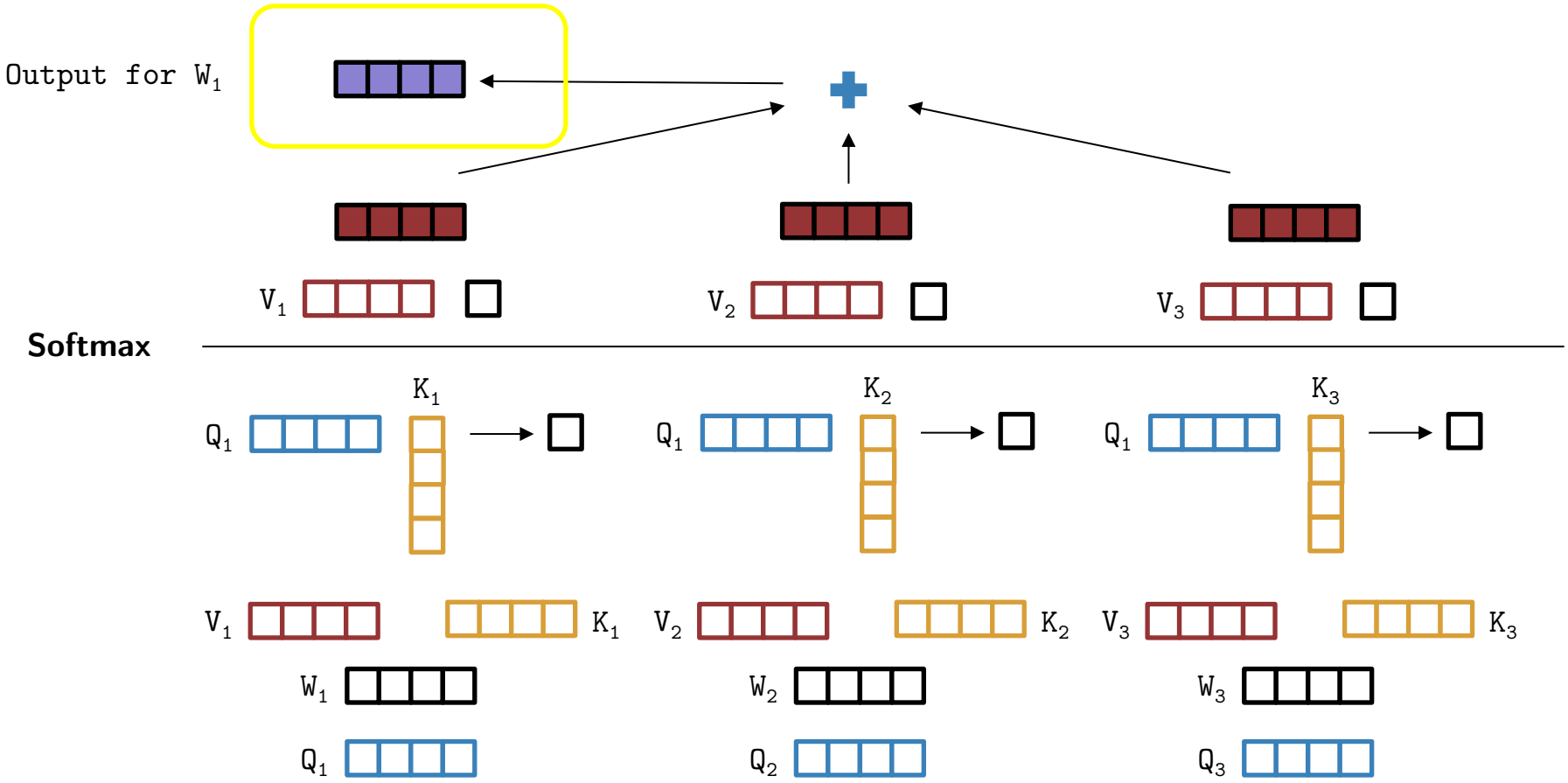
Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
(November 2021, **30,000+ citations**)

Self-Attention mechanism



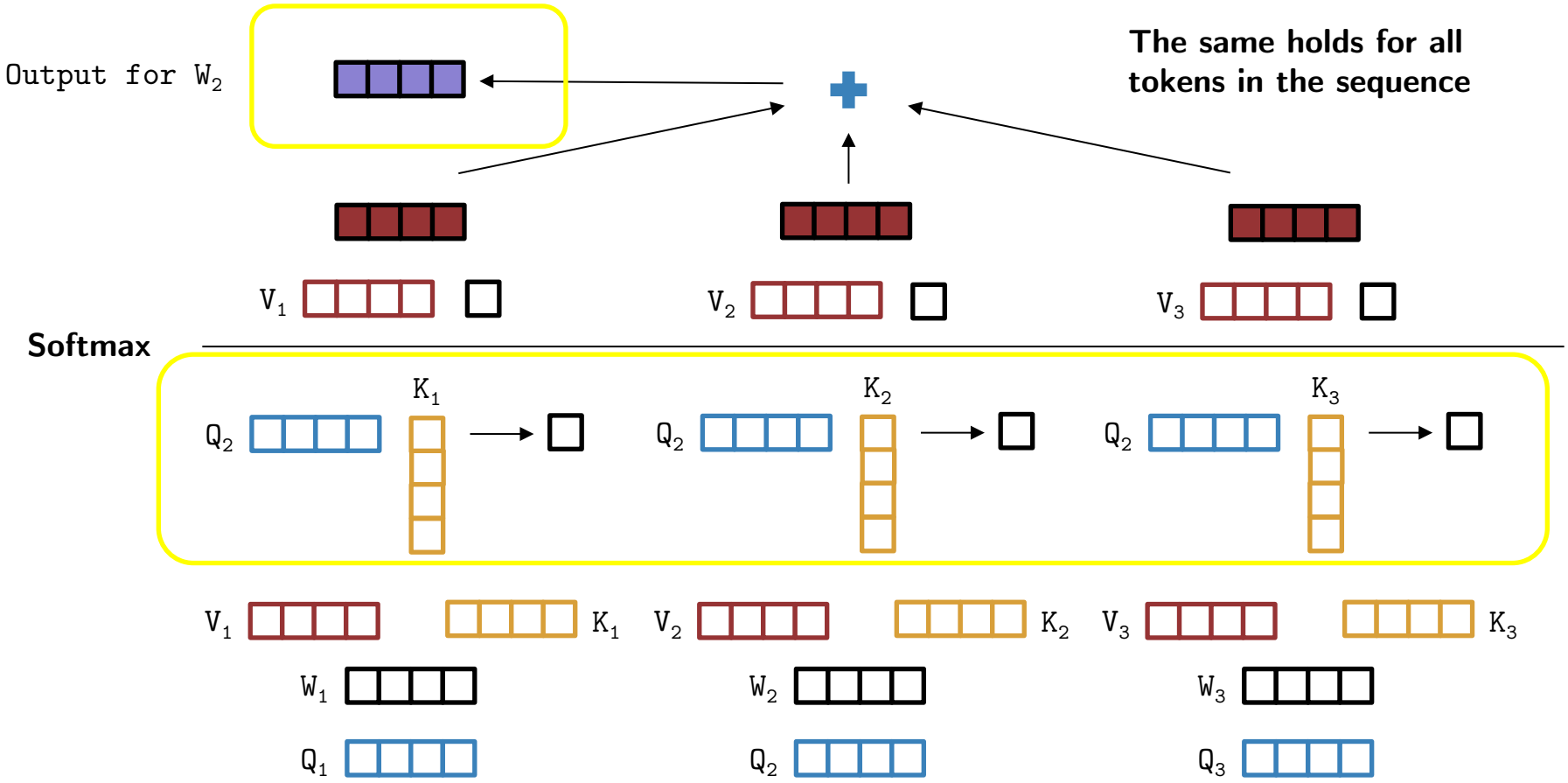
Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
(November 2021, **30,000+ citations**)

Self-Attention mechanism



Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
(November 2021, **30,000+ citations**)

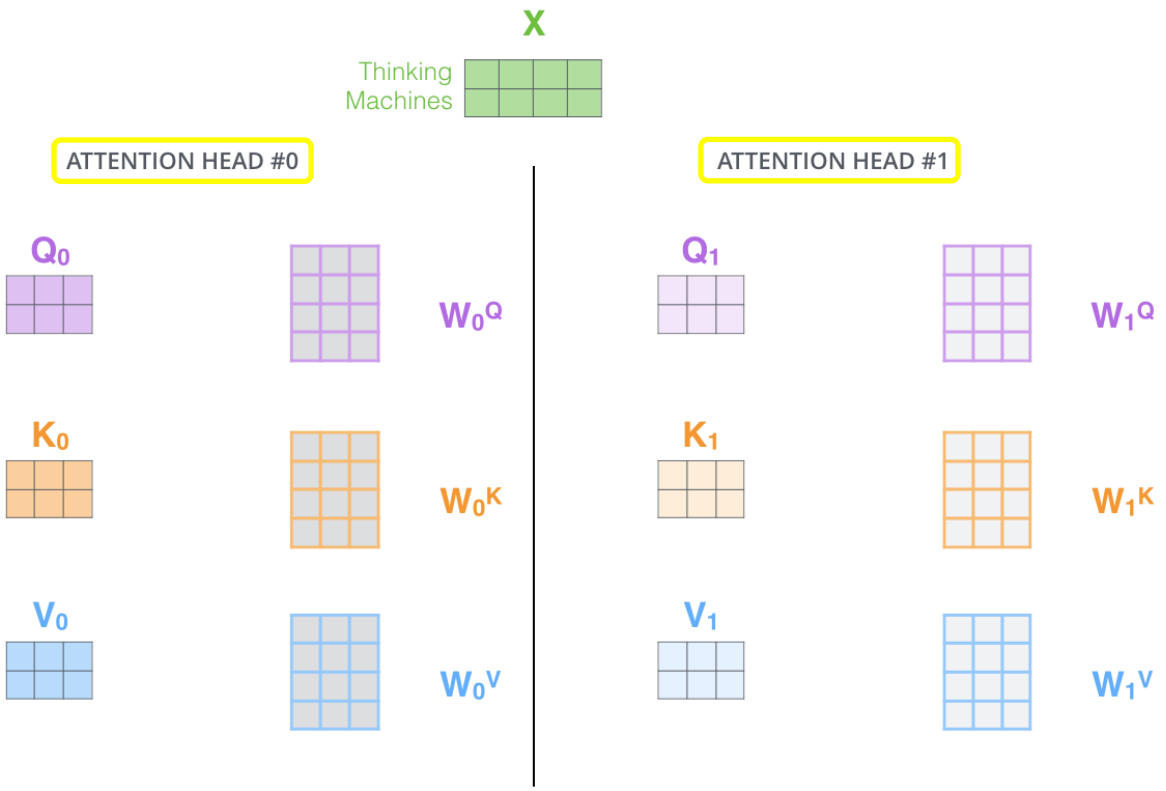
Self-Attention mechanism



Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
(November 2021: **30,000+ citations**; November 2023: **95,000+ citations**)

Self-Attention mechanism

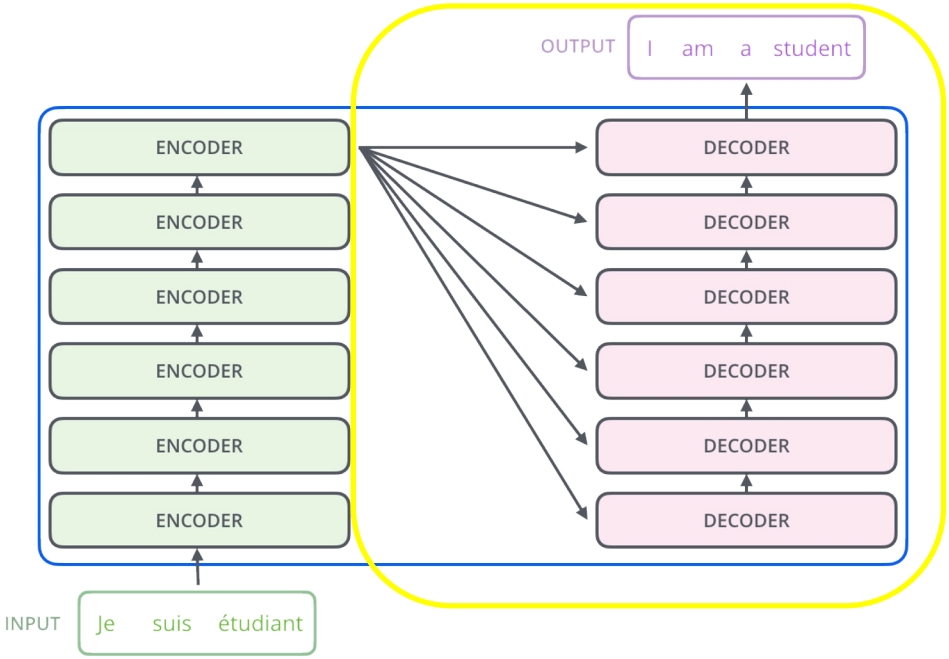
Transformer architecture stacks multiple encoder layers (multiple attention computation) and multiple heads within the same layer.



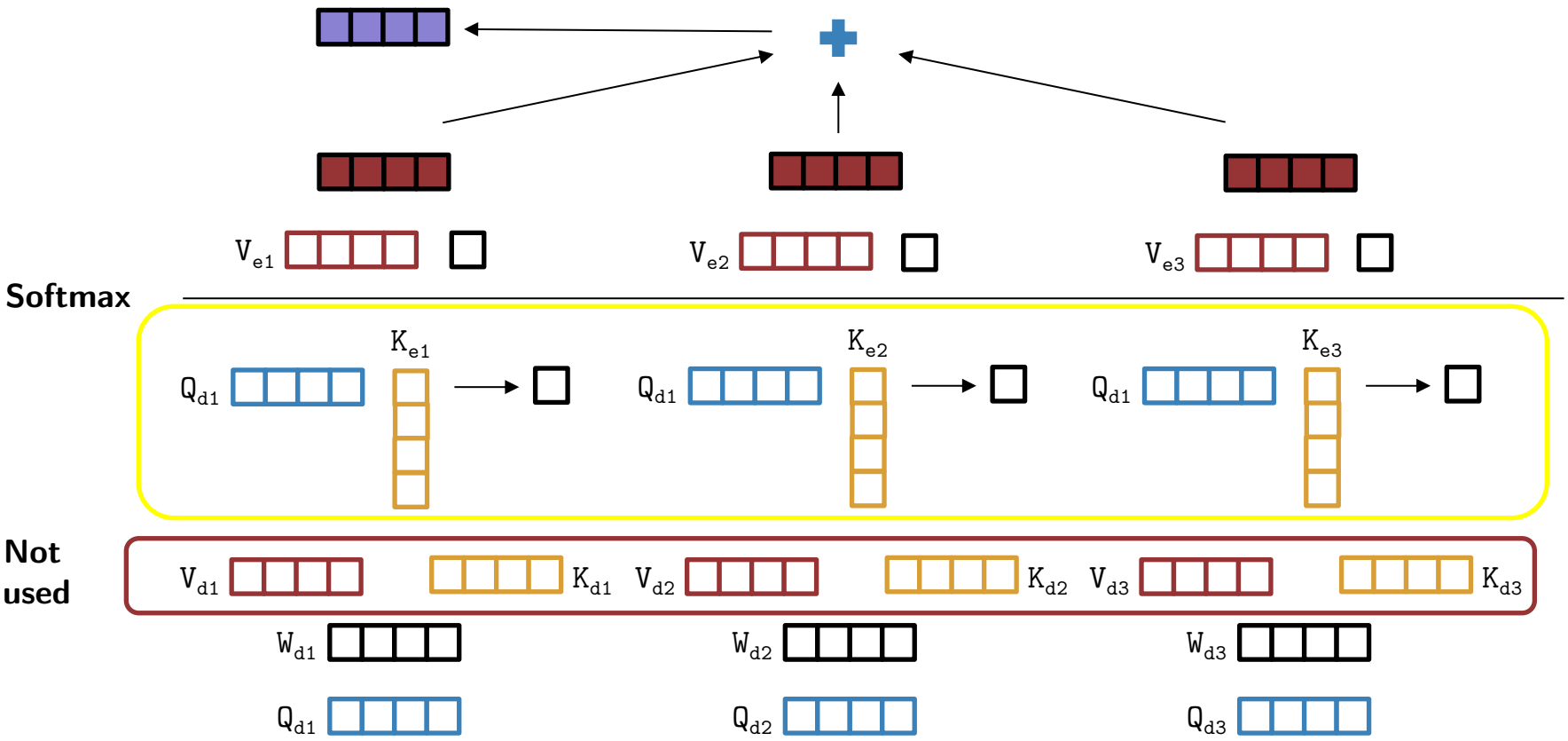
Cross-Attention

When the attention mechanism is performed on queries, keys and values generated from the same embedding is called self attention.

When attention is performed on queries generated from one embedding and keys and values generated **from another source** is called cross attention.

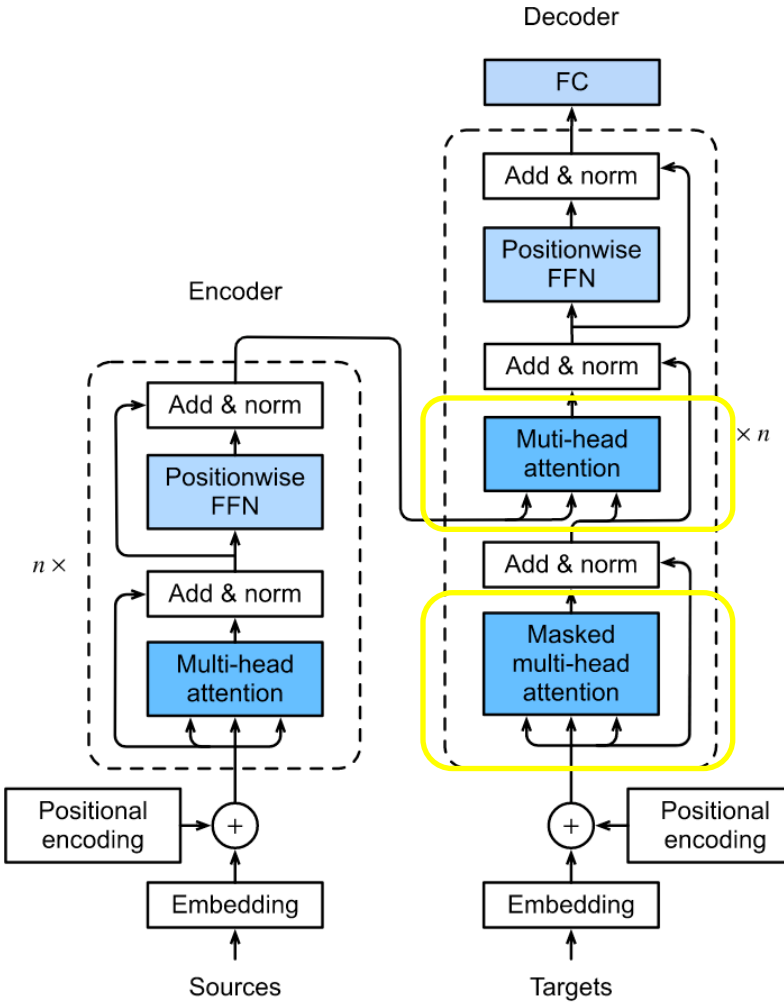


Cross-Attention



Query vectors are obtained strating from the nresult of the previous decoder steps

Cross-Attention



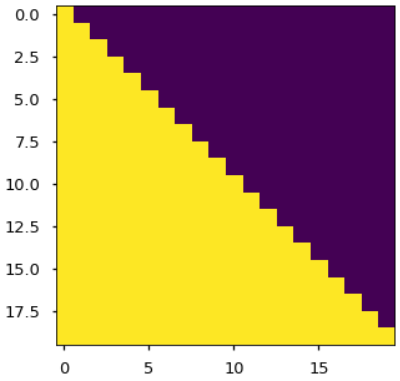
Cross attention

Masked 'self' attention

Masked self-attention

It is computed using the input
of the decoder itself (not cross)

Masked self-attention

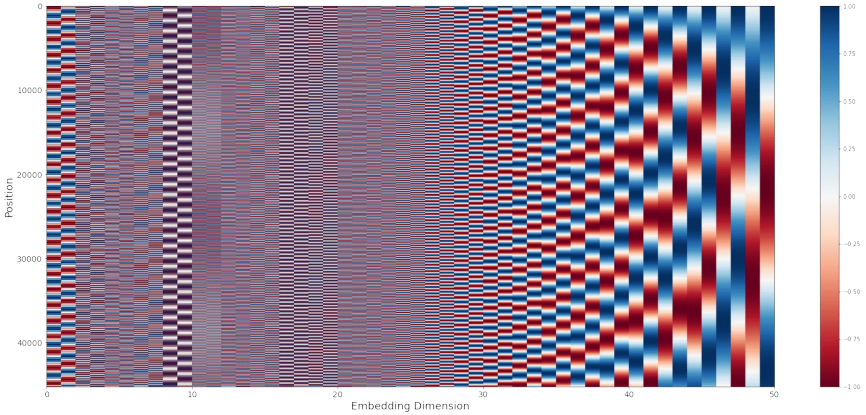


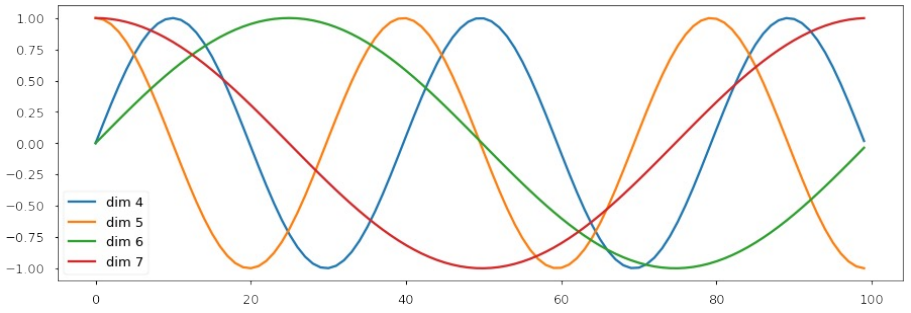
At each step, the decoder only access the
words that have already been generated!

Positional Encodings

The transformer architecture does not contain information about the position of current word. Positional encodings are used to inject some information about the relative or absolute position of the tokens in the sequence.

Different sine/cosine functions are used to encode the position of each token (1 vector for each token). Its use allows the encoding of **local** and **global** positions.





Positional Encodings

The use of several sine/cosine waves, with different frequencies, allows the generation of a vector that can be added to the embedding vector. This vector is called the positional encoding vector.

The great thing about **this vector is that it is a function of the position of the word in the sentence**. This means that the model can understand that the words in the sentence are in a specific order. This is important because the model needs to understand the context of the sentence.

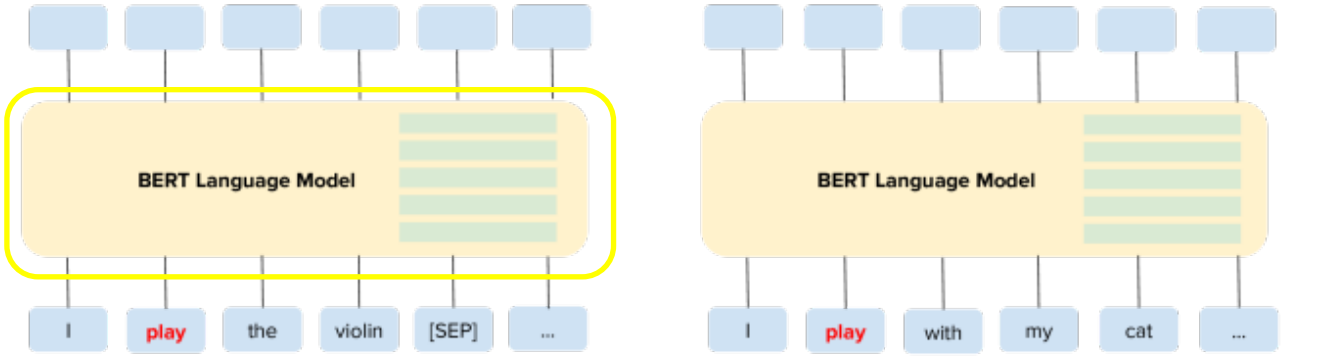
https://erdem.pl/2021/05/understanding-positional-encoding-in-transformers

Let's understand better

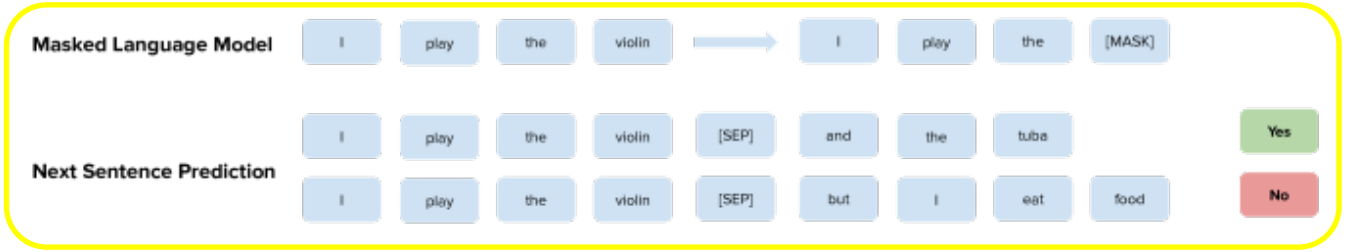
Encoder model - BERT

BERT is an encoder-only mode based on transformer architecture.

`12x transformer blocks (bert-base)`



`Self-supervised pretraining tasks`



Nozza, Debora, Federico Bianchi, and Dirk Hovy. "What the [mask]? making sense of language-specific BERT models." *arXiv preprint arXiv:2003.02912* (2020).

BERT – Masked Language Modeling

For each sentence in the training data, the model randomly masks 15% of the words, then runs the entire masked sentence through the model.

The model then **uses the unmasked words** to predict the masked words.

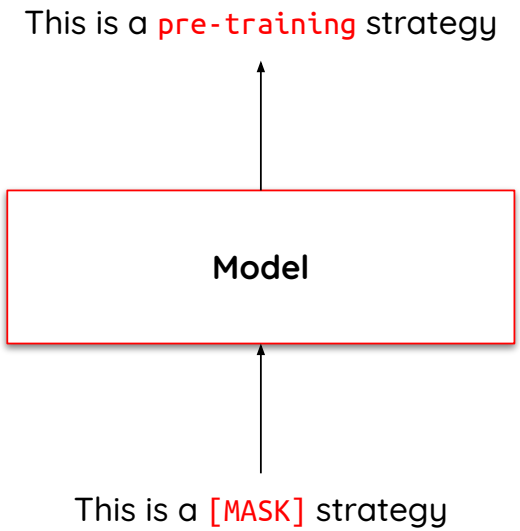


Image source: La Quatra, M. (2022). *Deep Learning for Natural Language Understanding and Summarization* (Doctoral dissertation, Politecnico di Torino).

BERT – Next sentence prediction

For each sentence in the training data, the model randomly selects another sentence from the training data as the second sentence.

The model then runs both sentences through the model and tries to predict **if the second sentence is the next sentence** in the original document.

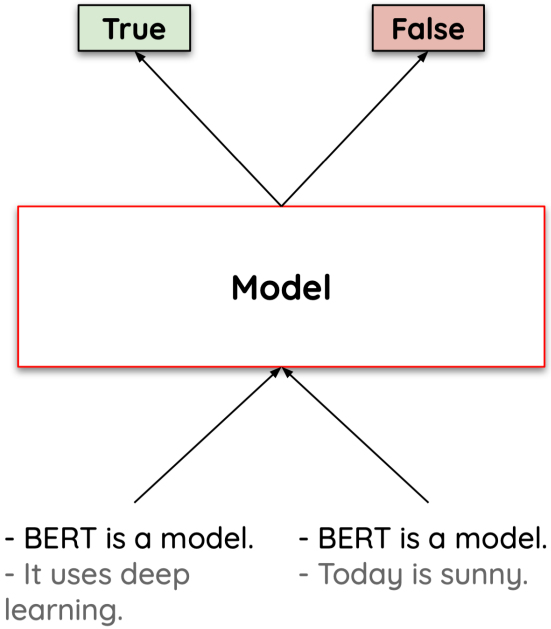


Image source: La Quatra, M. (2022). *Deep Learning for Natural Language Understanding and Summarization* (Doctoral dissertation, Politecnico di Torino).

Encoder model - RoBERTa

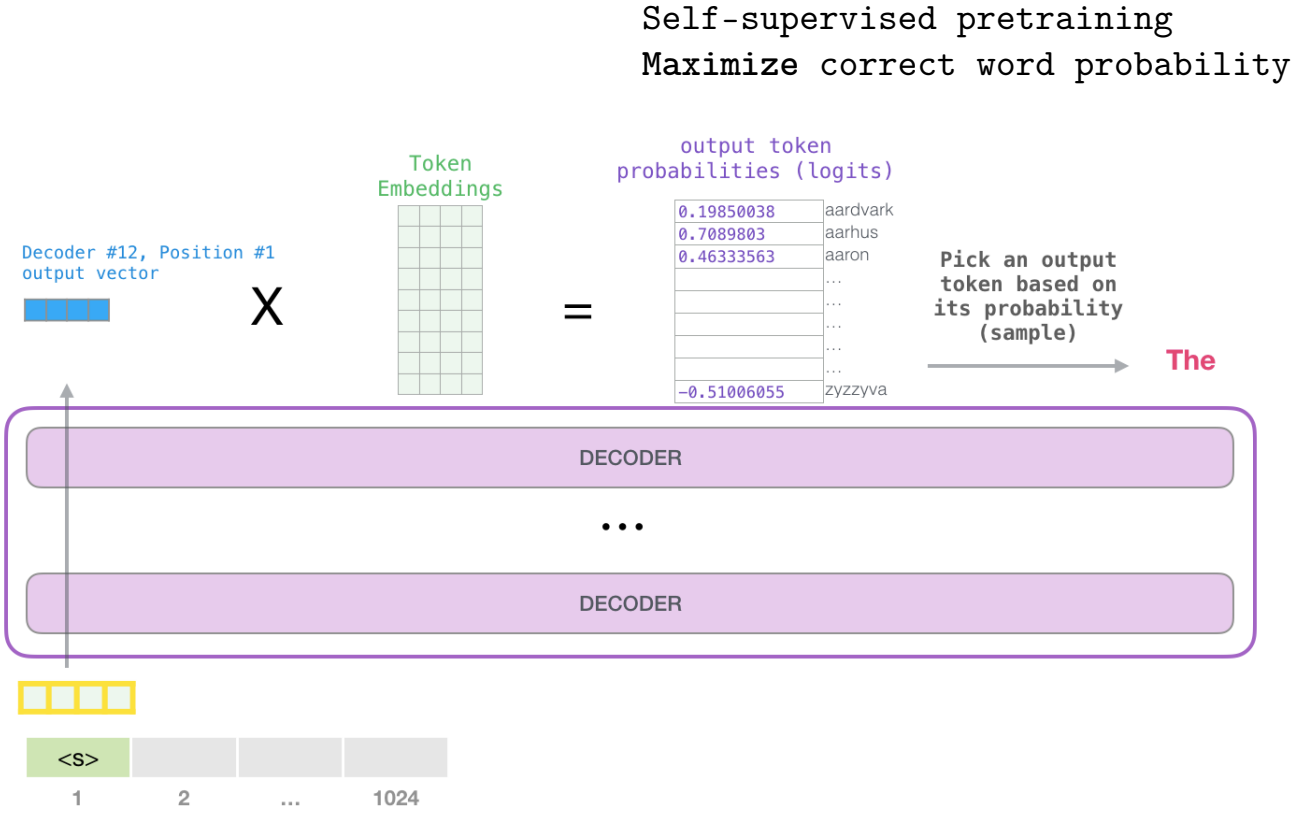
BERT relies on randomly masking and predicting tokens. The original BERT implementation performed masking once during data preprocessing, resulting in a single static mask.

RoBERTa (A Robustly Optimized BERT Pretraining Approach) authors propose **dynamic masking** where they generate the masking pattern every time we feed a sequence to the model. This becomes crucial when pretraining for more steps or with larger datasets.

Decoder model - GPT

OpenAI

GPT is a decoder-only mode based on transformer architecture.



https://jalammar.github.io/illustrated-gpt2/

GPT - Next word prediction

OpenAI

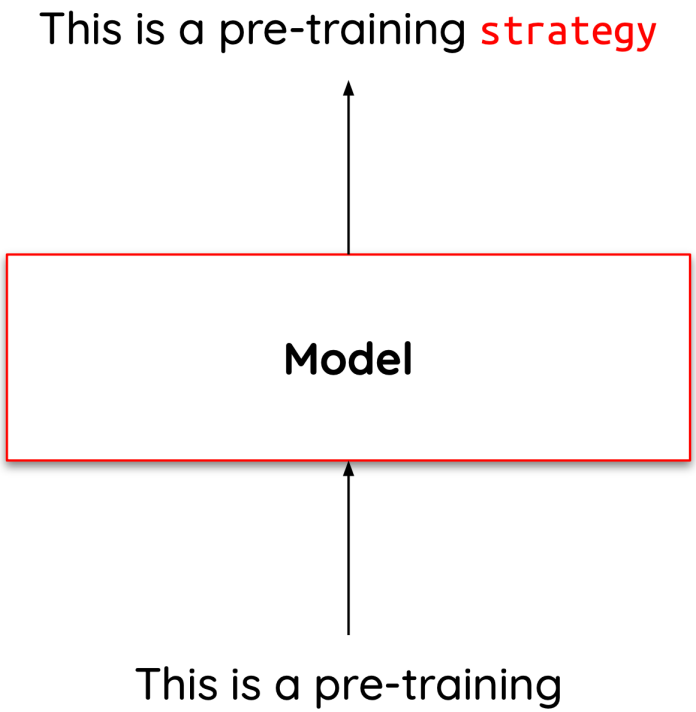


Image source: La Quatra, M. (2022). *Deep Learning for Natural Language Understanding and Summarization* (Doctoral dissertation, Politecnico di Torino).

Encoder-decoder model - BART

BART is an encoder-decoder architecture (seq2seq model).

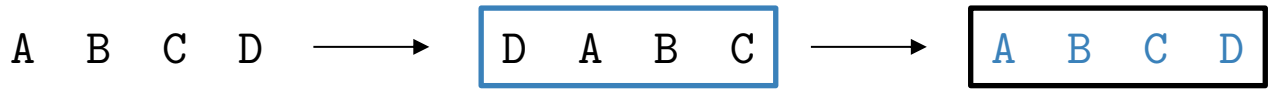
The self-supervised pretraining involves:

1. corrupting text with an arbitrary noising function
2. learning a model to reconstruct the original text

BART - corruptions

Examples of corruptions:

1. **Sentence permutation:** A document is divided into sentences based on full stops, and these sentences are shuffled in a random order.



2. **Token infilling:** a number of text spans are sampled. Each span is replaced with a single [MASK] token.

1 2 3 4 5 6 7 8 1 [MASK] 5 6 7 8 1 2 3 4 5 6 7 8

BART - corruptions

Examples of corruptions:

3. Token Masking: Random tokens are replaced with the [MASK] token

`1 2 3 4 5 6 7 8` `1 2 3 [MASK] 5 6 7 8` `1 2 3 4 5 6 7 8`

4. Token deletion: Random tokens are deleted from the input. In contrast to token masking, the model must decide which positions are missing inputs.

`1 2 3 4 5 6 7 8` `1 2 3 5 6 7 8` `1 2 3 4 5 6 7 8`

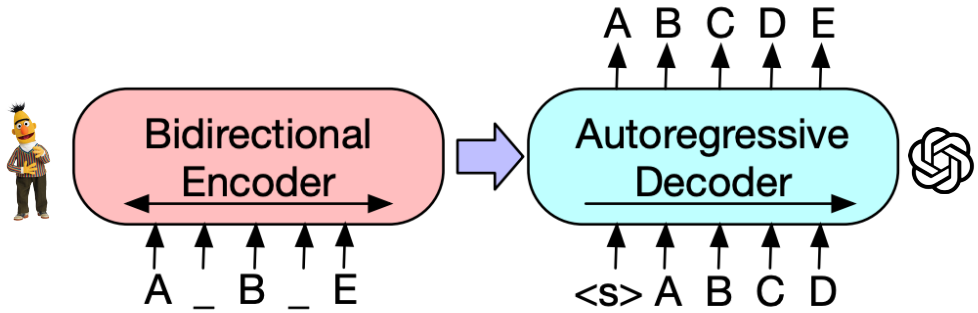
BART - corruptions

Examples of corruptions:

5. Document rotation: a token is chosen uniformly at random, and the document is rotated so that it begins with that token.

1 2 3 4 5 6 7 8 → 7 8 1 2 3 4 5 6 → 1 2 3 4 5 6 7 8

BART - corruptions



Tasks:

- Machine Translation
- Text Summarization
- ...

Lewis, Mike, et al. "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension." arXiv preprint arXiv:1910.13461 (2019).

Lecture goals

- Transformer architectures (recap)
  - Self- and Cross-Attention
  - Encoder/Decoder/Sequence-to-Sequence
  - Architectures (BERT, GPT, BART)
- **Exercises using Transformers**
- Task-specific Transformers (exercises)
  - Sequence classification
  - Entity recognition
  - Semantic Search
- From development to production
  - Serving model for AI applications

Exercise 1

Given a randomly initialized BERT model, illustrate an example of the self-supervised procedure to pretrain the model using **masked language modelling** task. Provide examples for the input and output of the model.

`Input: model, list_sentences`

Exercise 1 - procedure

`Input: model, list_sentences`

- Generate masked sentences starting from `list_sentences`
  - For each sentence mask **p%** of tokens
- Training example:
  - `Input : The [MASK] is on the [MASK]`
  - `Output: The` **`pen`** `is on the` **`table`**
- For each sentence in list:
  - Forward pass to probability for each masked token
  - Compute loss and backpropagate to train the model with correct token.

Exercise 1 – pseudo code

```
Input: model, list_sentences

for each sentence in list_sentences:
        tokenized = model.tokenizer.tokenize(sentence)
        # tokenized = [1, 5, 15, 78 ...]
        masked = get_masked_sentence(tokenized, p%=0.15)
        # sentence = 'The pen is on the table'
        # masked (text) = 'The [MASK] is on the [MASK]'
        # masked (token mapping) = [1 [MASK] 15 78 20 [MASK]]
        # masked (full) = [1 100 15 78 20 100] 100 could be the id of MASK
        pred = model.forward(masked)
        model.compute_loss(pred, sentence)
        model.backward()
```

Exercise 2

Given a randomly initialized BERT model, illustrate an example of the self-supervised procedure to pretrain the model using Next Sentence Prediction task. Provide examples for the input and output of the model.

```
Input: model, documents [list of list of sentences]
```


Exercise 2 - procedure

`Input: model, documents [list of list of sentences]`

- Generate training dataset
  - For each sentence create a positive pair (sentence itself and its followinf sentence)
  - For each sentence create a negative pair (sentence itself and a randomly selected sentence from another document)
- Training example:
  - `Input : S1: PoliTO is a university. S2: It is located in Turin.`
  - `Output: 1`
  - `Input : S1: PoliTO is a university. S3: Next season coming soon.`
  - `Output: 0`
- For each example in dataset:
  - Forward pass to classify the example
  - Compute the loss and backward for training (binary classification).

Exercise 2 – pseudo code

```
Input: model, documents [list of list of sentences]


for each doc in documents:
      for each sent in doc:
            positive_pair = (sent, sent+1, 1)
#('PoliTO is an university', 'It is located in Turin', 1)
            neg_example = random_sampling(documents, doc)
            negative_pair = (sent, neg_example, 0)
#('PoliTO is an university', 'Next season coming soon', 0)
            pred = model.forward(positive_pair)
            model.compute_loss(pred, 1) # positive_pair[2]
            model.backward()
...
```

Exercise 2b

Given a randomly initialized BERT model, let's see the steps to pre-train the model on a 'new language': Klingon.

The data collection is provided with a split into train/eval/test.

Code sample – kli-BERT - **tokenizer**

```
#! pip install tokenizers

from pathlib import Path
from tokenizers import ByteLevelBPETokenizer

# Get a //huge// training corpus
paths = [str(x) for x in Path("./klingon_data/").glob("**/*.txt")]

# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()
# Customize training
tokenizer.train(files=paths, vocab_size=52_000, min_frequency=2, special_tokens=[
    "<s>",
    "<pad>",
    "</s>",
    "<unk>",
    "<mask>",
])
# Save files to disk
tokenizer.save_model(".", "kli-bert")
```

Code sample – kli-BERT - **dataset**

```python
from torch.utils.data import Dataset

class KlingonDataset(Dataset):
    def __init__(self, evaluate: bool = False):
        tokenizer = ByteLevelBPETokenizer(
            "./models/kli-BERT/vocab.json",
            "./models/kli-BERT/merges.txt",
        )
        tokenizer._tokenizer.post_processor = BertProcessing(
            ("</s>", tokenizer.token_to_id("</s>")),
            ("<s>", tokenizer.token_to_id("<s>")),
        )
        tokenizer.enable_truncation(max_length=512)

        self.examples = []

        src_files = Path("./data/").glob("*-eval.txt") if evaluate else Path("./data/").glob("*-train.txt")
        for src_file in src_files:
            print("Processing: ", src_file)
            lines = src_file.read_text(encoding="utf-8").splitlines()
            self.examples += [x.ids for x in tokenizer.encode_batch(lines)]

    def __len__(self):
        return len(self.examples)

    def __getitem__(self, i):
        # We'll pad at the batch level.
        return torch.tensor(self.examples[i])
```

Code sample – kli-BERT – **training**

https://github.com/huggingface/transformers/blob/master/examples/legacy/run_language_modeling.py

```
python run_language_modeling.py --output_dir ./models/kli-BERT
    --model_type bert
    --mlm #for bert models
    --config_name ./models/kli-BERT
    --tokenizer_name ./models/kli-BERT
    --do_train
    --do_eval
    --learning_rate 1e-4
    --num_train_epochs 5
    --save_total_limit 2
    --save_steps 2000
    --per_gpu_train_batch_size 16
    --evaluate_during_training
    --seed 42
```

Exercise 3

Given a BERT model and a set of sentences, encode them to obtain fixed-size vector representations to build a semantic search engine.

```
Input: model, list_sentences
```


Exercise 3 - procedure

`Input: model, list_sentences`

- Define a pooling function (F) to aggregate token-level vectors
  - Average
  - Max
  - Min
- Alternatively, define a way to get fixed-size representation (e.g., `[CLS]` token) – **more on this in the next slides**
- For each sentence:
  - Forward pass to the model
  - Use model's output to obtain fixed size representations
- Index vectors using an arbitrary software (e.g., ElasticSearch, Apache SolR)

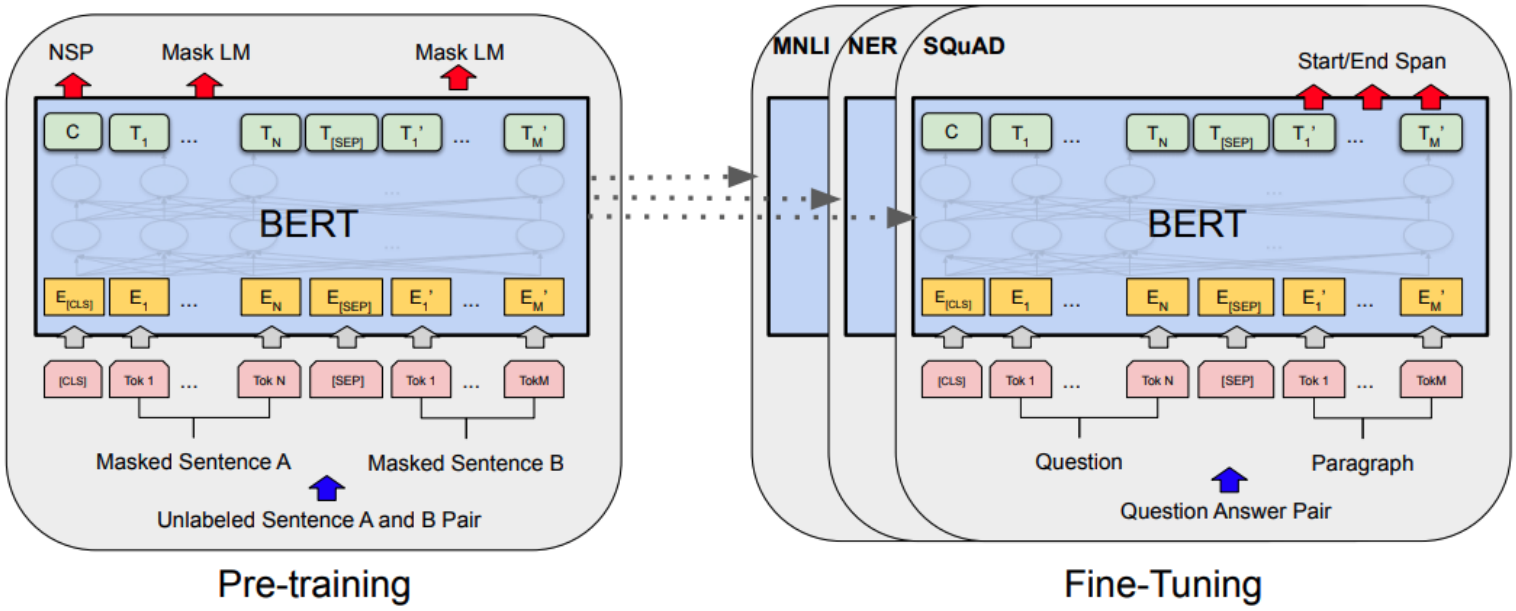
Exercise 3 – pseudo code

```
Input: model, sentences
def avg_pooling(model, s):
        out = model.forward(s)
        tokens_vectors = out.last_hidden_states
        sentence_vector = mean(token_vectors)

for each s in sentences:
         sentence_vector = avg_pooling(model, s)

for i, v in enumerate(sentence_vector):
        doc = {‘vector’: v, ‘text’: sentences[i]}
        elastic.index(‘my-index’, body=doc)
```

Lecture goals

- Transformer architectures (recap)
  - Self- and Cross-Attention
  - Encoder/Decoder/Sequence-to-Sequence
  - Architectures (BERT, GPT, BART)
- Exercises using Transformers
- **Task-specific Transformers (exercises)**
  - Sequence classification
  - Entity recognition
- From development to production
  - Serving model for AI applications

Train task-specific models



Large language models require high amount of training data.

- Self-Supervised Learning on standard pretraining tasks (e.g., MLM)
- Supervised fine-tuning with 'limited' data exaples

Sequence classification

Task: given a text sequence (sentence or paragraph) associate a class to the given sequence.

Sentiment Analysis

```
This was a good movie!
It was an average experience
I hated that, so boring!
```

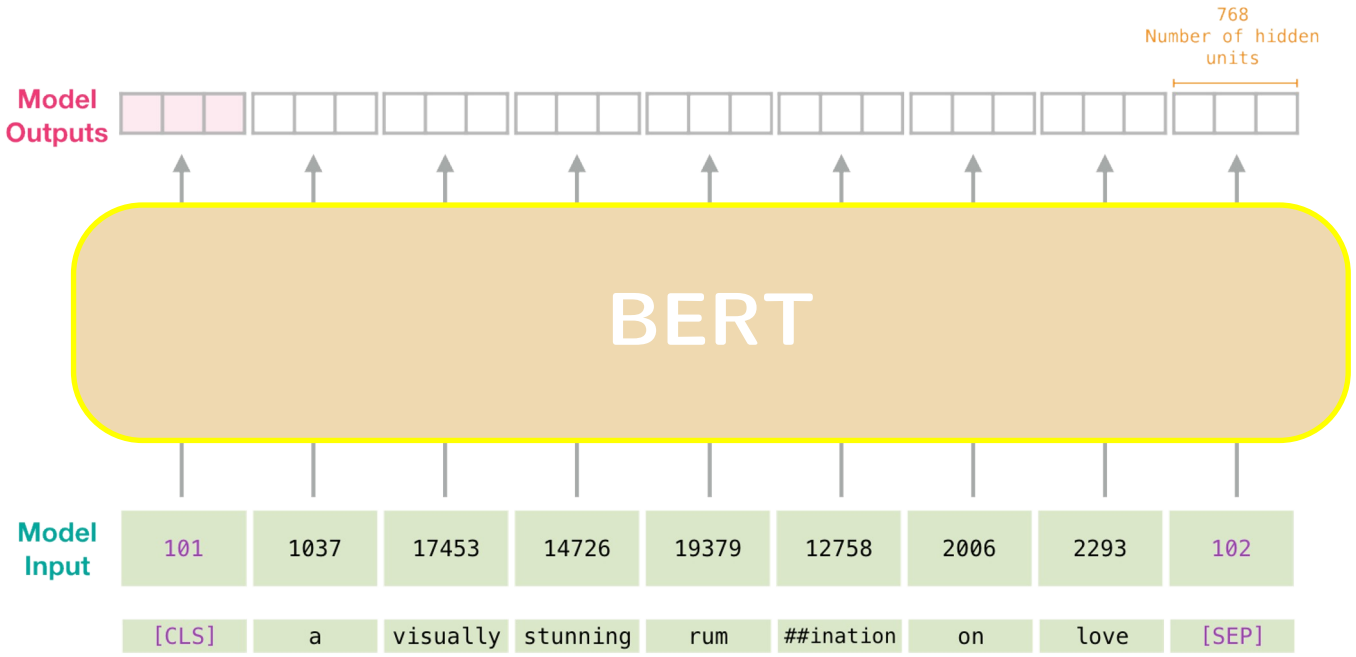
Text categorization

```
The team won the match -
Joe Biden was elected as US president -
Google stocks went up! -
```



Sequence classification

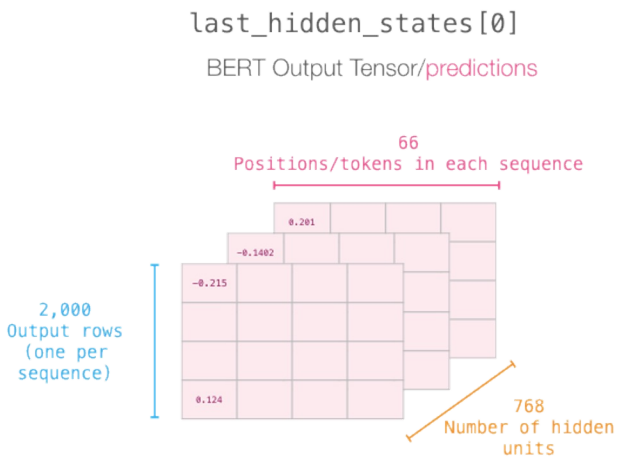
Standard BERT-based models do not have sequence classification capabilities. One vector per token!

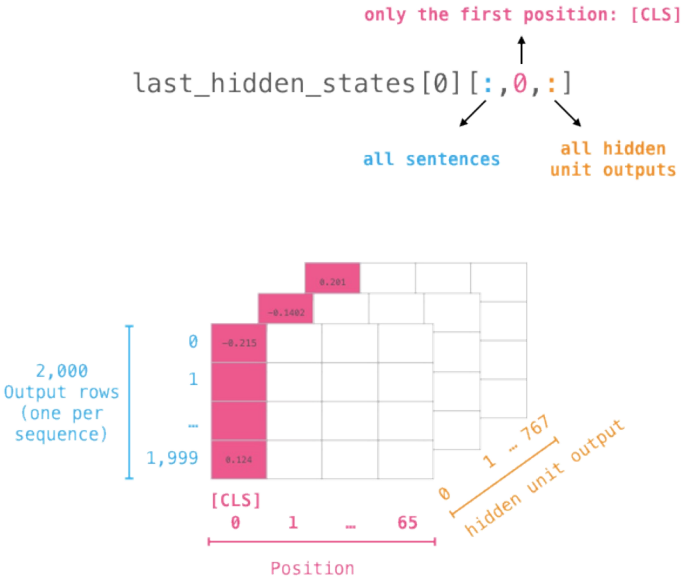


https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Sequence classification

A special token `[CLS]` is used as feature encoder for the whole sentence.

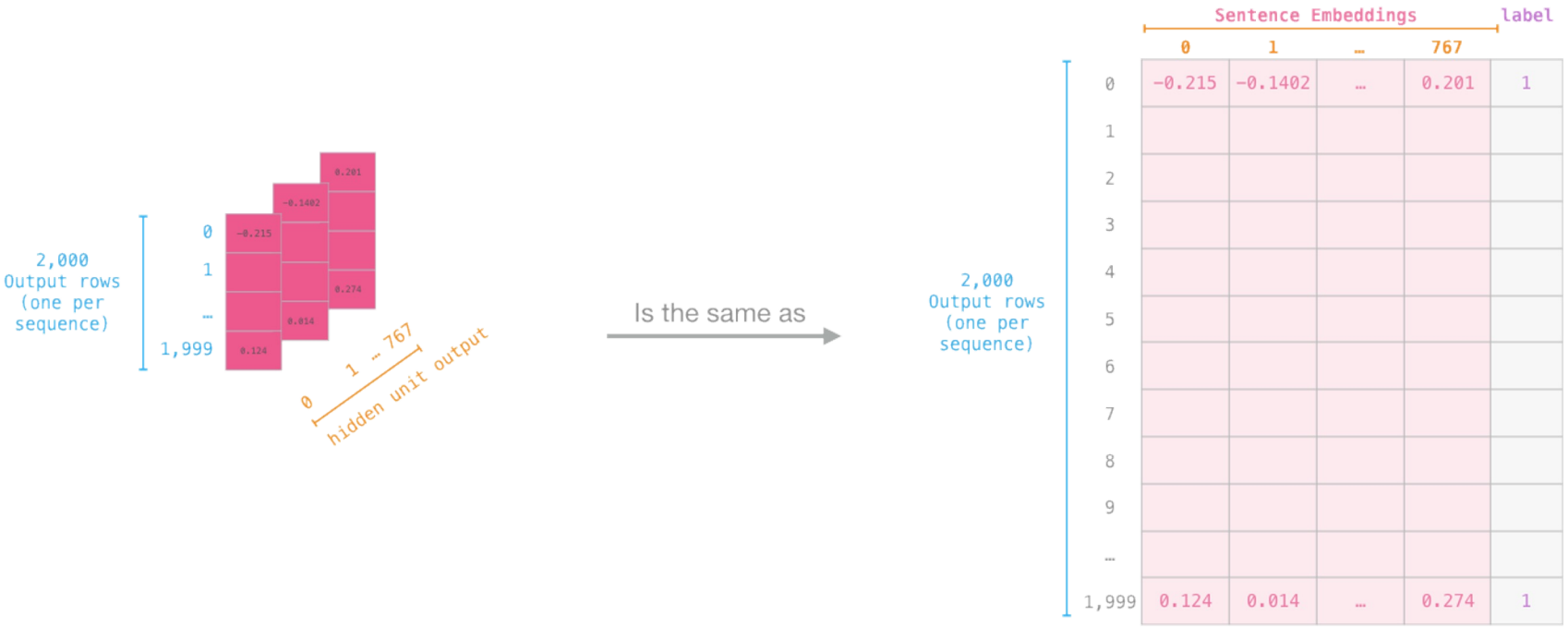




https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Sequence classification

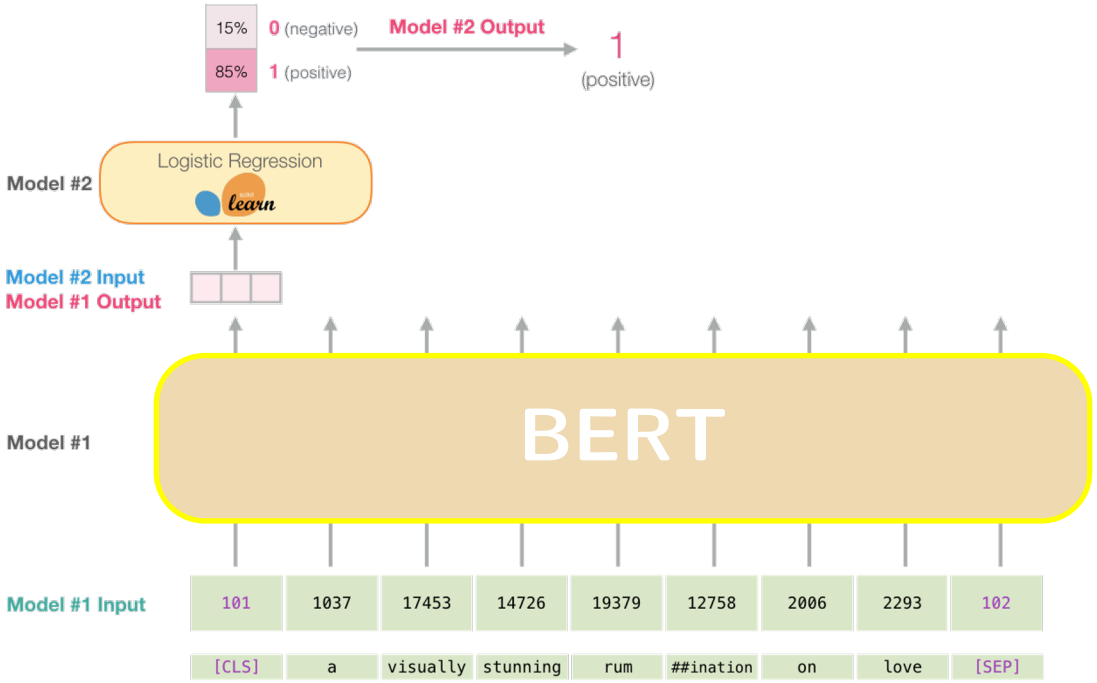
Special token `[CLS]` is used as feature encoder for the whole sentence.



https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Sequence classification

- **It is possible to train another model on sentence embeddings produced by BERT**
  - BERT is only used as an encoder (**BERT model is not finetuned**)

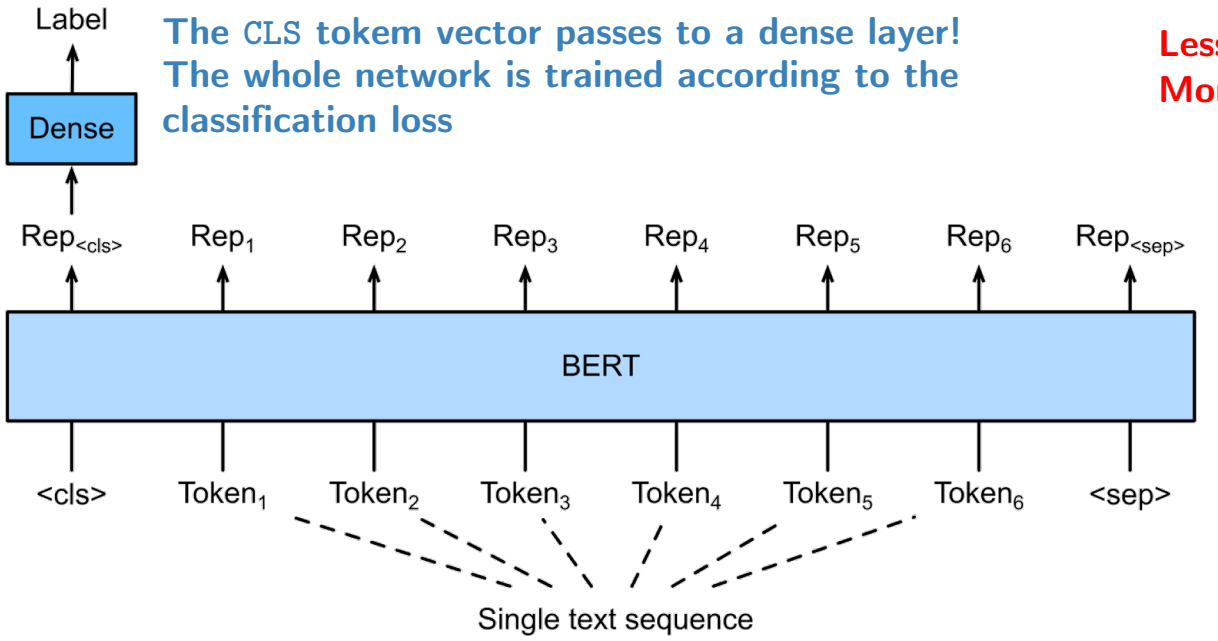


More efficient
Less effective

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Sequence classification

- **It is possible to include a 'classification head' on top of BERT to finetune the entire model**
  - A dense layer with RELU activation function (for example)



https://d2l.ai/chapter_natural-language-processing-applications/finetuning-bert.html

BERTForSequenceClassification - prepare

```python
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels = 2) #clf

# Tokenize

tokenized_train = tokenizer(train_sentences, padding="max_length", truncation=True, max_length=64)
tokenized_test = tokenizer(test_sentences, padding="max_length", truncation=True, max_length=64)
tokenized_val = tokenizer(val_sentences, padding="max_length", truncation=True, max_length=64)

# create dataset
# train eval test split

class DNLPDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_ds = DNLPDataset(tokenized_train, train_labels)
val_ds   = DNLPDataset(tokenized_val, val_labels)
test_ds  = DNLPDataset(tokenized_test, test_labels)
```

https://huggingface.co/transformers/model_doc/auto.html

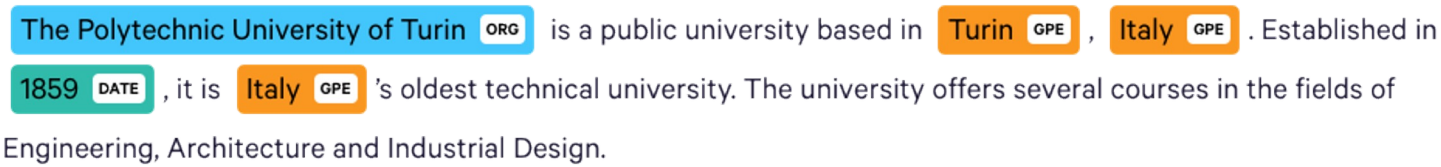
BERTForSequenceClassification - trainer

```python
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=4,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=32,   # batch size for evaluation
)

trainer = Trainer(
    model=model,                 # the instantiated 🤗 Transformers model
    args=training_args,          # training arguments, defined above
    train_dataset=train_ds, # training dataset
    eval_dataset=val_ds,     # evaluation dataset
)

trainer.train()
```

https://huggingface.co/transformers/main_classes/trainer.html

Entity Recognition (recap)

Entity recognition is the task of identifying (named) entities in text: people, locations, organizations...



Each token is associated with a class:

- No entity (NE)
- B-X (**beginning** of entity of type X)
- I-X (**inside** of entity of type X)

Entity Recognition (recap)

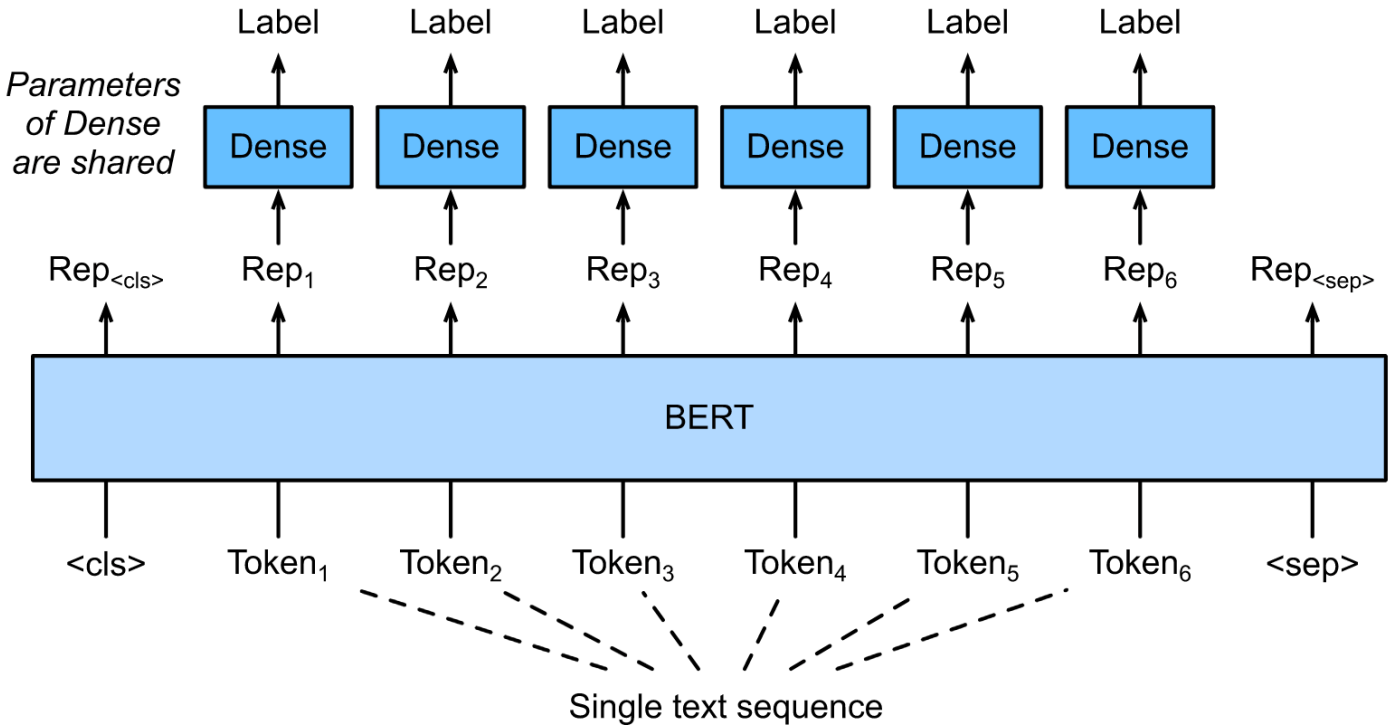
The	Polytechnic	University	of	Turin	is	a	public	university
B-ORG	I-ORG	I-ORG	I-ORG	I-ORG	NE	NE	NE	NE

It is similar to the classification task but:

- It is a token-level task
- There could be several classes
- There is a 'majority' class (NE)
- I-X class, always follows B-X

Token-level classification

Each token is classified to a given class.



https://d2l.ai/chapter_natural-language-processing-applications/finetuning-bert.html

BertForTokenClassification - **prepare**

```
print(texts[0][10:17], tags[0][10:17], sep='\n')
DNLP>> ['for', 'two', 'weeks', '.', 'Empire', 'State', 'Building']
DNLP>> ['O', 'O', 'O', 'O', 'B-location', 'I-location', 'I-location']

# Tags into numbers
unique_tags = set(tag for doc in tags for tag in doc)
tag2id = {tag: id for id, tag in enumerate(unique_tags)}
id2tag = {id: tag for tag, id in tag2id.items()}

from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
train_encodings = tokenizer(train_texts, is_split_into_words=True, return_offsets_mapping=True, padding=True,
truncation=True)
val_encodings = tokenizer(val_texts, is_split_into_words=True, return_offsets_mapping=True, padding=True,
truncation=True)
```

https://huggingface.co/transformers/custom_datasets.html

BertForTokenClassification - **prepare**

```python
import numpy as np

def encode_tags(tags, encodings):
    labels = [[tag2id[tag] for tag in doc] for doc in tags]
    encoded_labels = []
    for doc_labels, doc_offset in zip(labels, encodings.offset_mapping):
        # create an empty array of -100
        doc_enc_labels = np.ones(len(doc_offset),dtype=int) * -100
        arr_offset = np.array(doc_offset)

        # set labels whose first offset position is 0 and the second is not 0
        doc_enc_labels[(arr_offset[:,0] == 0) & (arr_offset[:,1] != 0)] = doc_labels
        encoded_labels.append(doc_enc_labels.tolist())

    return encoded_labels

train_labels = encode_tags(train_tags, train_encodings)
val_labels = encode_tags(val_tags, val_encodings)
```

Remember, sub-word tokenization!

https://huggingface.co/transformers/custom_datasets.html

BertForTokenClassification - **prepare**

```python
import torch

class NERDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_encodings.pop("offset_mapping") # we don't want to pass this to the model
val_encodings.pop("offset_mapping") # we don't want to pass this to the model
train_dataset = NERDataset(train_encodings, train_labels)
val_dataset = NERDataset(val_encodings, val_labels)
```

https://huggingface.co/transformers/custom_datasets.html

BertForTokenClassification - trainer

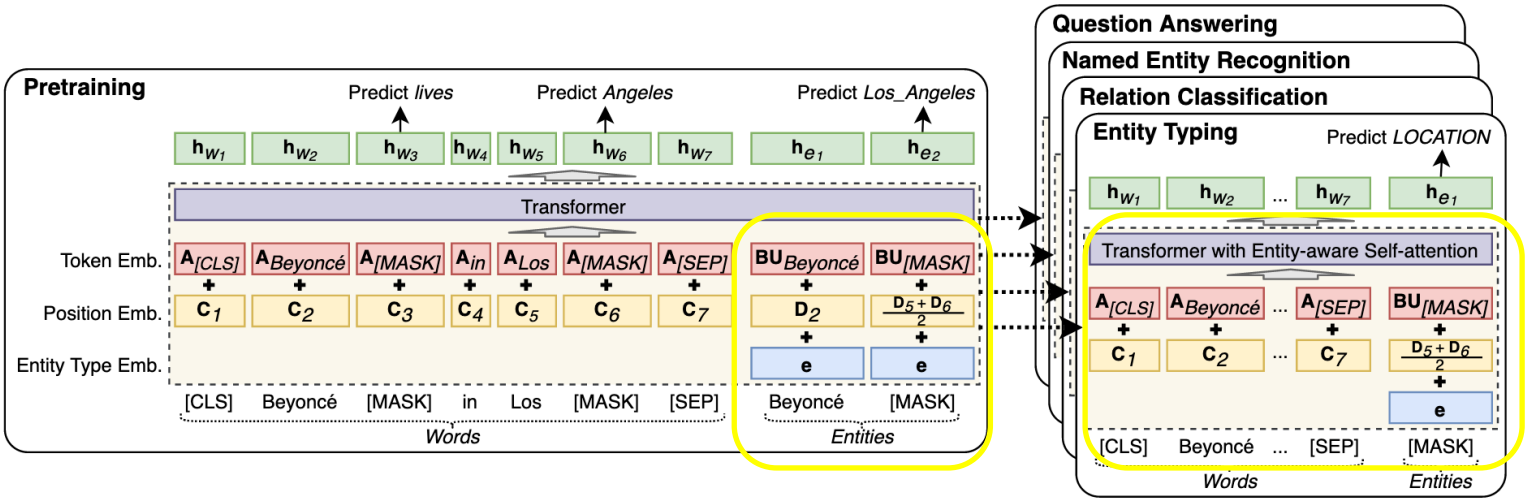
```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=4,              # total number of
    per_device_train_batch_size=16,  # batch si
    per_device_eval_batch_size=32,   # b
)

trainer = Trainer(
    model=model,                 # the instantiated 🤗 Transformers model
    args=training_args,          # training arguments, defined above
    train_dataset=train_ds,      # training dataset
    eval_dataset=val_ds,         # evaluation dataset
)

trainer.train()
```

The same trainer procedure of SequenceClassfication

https://huggingface.co/transformers/main_classes/trainer.html

Transformers for NER - LUKE

LUKE is a transformer-based model that has been specifically proposed for entity modeling.



Transformers for NER - LUKE

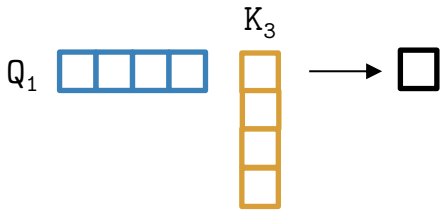
For each word pair, standard self-attention mechanism proceed as seen at the beginning of the lesson.

LUKE introduces 2 additional token types:

- Words
- Entities (that are **also** words) but are specifically modeled in LUKE

The model embed an entity-aware attention mechanism. The authors introduce 3 additional query matrices for entities:

standard	$\mathbf{K}\ \mathbf{x}_j^T\ \mathbf{Q}\ \mathbf{x}_i$
x_i word, x_j entity	$\mathbf{K}\ \mathbf{x}_j^T\ \mathbf{Q}_{w2e}\ \mathbf{x}_i$
x_j word, x_i entity	$\mathbf{K}\ \mathbf{x}_j^T\ \mathbf{Q}_{e2w}\ \mathbf{x}_i$
both x_j x_i entities	$\mathbf{K}\ \mathbf{x}_j^T\ \mathbf{Q}_{e2e}\ \mathbf{x}_i$



Taken from different matrix according to token type

Transformers for NER - LUKE

Pretraining Task: to learn entity representations the authors extend MLM. The authors randomly mask a certain percentage of the **entities** by replacing them with special `[MASK]` entities and then train the model to predict the masked entities.

The original entity corresponding to a `[MASK]` entity is predicted by applying the softmax function (classification) over all entities in the vocabulary

Backbone architecture: the standard LUKE model is based on **RoBERTa.**

Tasks: The pretrained model is finetuned in five entity-related tasks: entity typing, relation classification, **NER**, cloze-style QA, and extractive QA.

Evaluating NER model

Evaluating NER models is not straightforward:

- The classification of entities is at token-level
- It need to evaluate both entity boundaries and entity type

eval4ner is a python package implementing metrics proposed in MUC conference.

Evaluating NER model

- **Strict**: exact boundary surface string match and entity type;
- **Exact**: exact boundary match over the surface string, regardless of the type;

$$\text{Precision} = \frac{COR}{ACT} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{COR}{POS} = \frac{TP}{TP + FN}$$

Correct (COR) : both are the same (system output and golden annotation);
Incorrect (INC) : the output of a system and the golden annotation don't match;
Partial (PAR) : system and the golden annotation are somewhat "similar" but not the same;
Missing (MIS) : a golden annotation is not captured by a system;
Spurius (SPU) : system produces a response which doesn't exist in the golden annotation;
Possible (POS): COR + INC + PAR + MIS
Actual (ACT): COR + INC + PAR + SPU

https://www.davidsbatista.net/blog/2018/05/09/Named_Entity_Evaluation/

Evaluating NER model

- **Partial**: partial boundary match over the surface string, regardless of the type;
- **Type**: some overlap between the system tagged entity and the gold annotation is required

$$\text{Precision} = \frac{COR \ + \ 0.5 \ \times \ PAR}{ACT} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{COR \ + \ 0.5 \ \times \ PAR}{POS} = \frac{COR}{ACT} = \frac{TP}{TP + FP}$$

Correct (COR) : both are the same (system output and golden annotation);
Incorrect (INC) : the output of a system and the golden annotation don't match;
Partial (PAR) : system and the golden annotation are somewhat "similar" but not the same;
Missing (MIS) : a golden annotation is not captured by a system;
Spurius (SPU) : system produces a response which doesn't exist in the golden annotation;
Possible (POS): COR + INC + PAR + MIS
Actual (ACT): COR + INC + PAR + SPU

https://www.davidsbatista.net/blog/2018/05/09/Named_Entity_Evaluation/

Evaluating NER model

```
# ! pip install eval4ner
import eval4ner.muc as muc

# ground truth
grount_truths = [
    [('PER', 'John Jones'), ('PER', 'Peter Peters'), ('LOC', 'York')],
    [('PER', 'John Jones'), ('PER', 'Peter Peters'), ('LOC', 'York')],
    [('PER', 'John Jones'), ('PER', 'Peter Peters'), ('LOC', 'York')]
]

# NER model prediction
predictions = [
    [('PER', 'John Jones and Peter Peters came to York')],
    [('LOC', 'John Jones'), ('PER', 'Peters'), ('LOC', 'York')],
    [('PER', 'John Jones'), ('PER', 'Peter Peters'), ('LOC', 'York')]
]
# input texts
texts = [
    'John Jones and Peter Peters came to York',
    'John Jones and Peter Peters came to York',
    'John Jones and Peter Peters came to York'
]
muc.evaluate_all(predictions, grount_truths * 1, texts, verbose=True)

>>> NER evaluation scores:
>>>  strict mode, Precision=0.4444, Recall=0.4444, F1:0.4444
>>>   exact mode, Precision=0.5556, Recall=0.5556, F1:0.5556
>>> partial mode, Precision=0.7778, Recall=0.6667, F1:0.6944
>>>    type mode, Precision=0.8889, Recall=0.6667, F1:0.7222
```

https://github.com/cyk1337/eval4ner

Evaluating NER model

Sequeval: it is a tool that allows to evaluate the performance of a NER model. It compares the output of the model with the gold standard.

It relies of the token-level classification metrics (precision, recall, f1-score) and computes the average of the metrics for each entity type (and overall).

	Words	True Label	Predicted Label	
entity	Foreign	ORG	O	FN
entity	Ministry	ORG	O	
	spokesman	O	O	
entity	Shen	PER	PER	FP
entity	Guofang	PER	O	
	told	O	O	
entity	Reuters	ORG	ORG	TP

Source: https://towardsdatascience.com/entity-level-evaluation-for-ner-task-c21fb3a8edf

Evaluating NER model

Sequeval: it is a tool that allows to evaluate the performance of a NER model. It compares the output of the model with the gold standard.

```
# after pip install evaluate
# after pip install seqeval

import evaluate
seqeval = evaluate.load('seqeval')

predictions = [['O', 'O', 'B-MISC', 'I-MISC', 'I-MISC', 'I-MISC', 'O'], ['B-PER', 'I-PER', 'O']]
references = [['O', 'O', 'O', 'B-MISC', 'I-MISC', 'I-MISC', 'O'], ['B-PER', 'I-PER', 'O']]

results = seqeval.compute(predictions=predictions, references=references)

print (results)

'''
{
    'MISC':
        {'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1},
    'PER':
        {'precision': 1.0, 'recall': 1.0, 'f1': 1.0, 'number': 1},
    'overall_precision': 0.5,
    'overall_recall': 0.5,
    'overall_f1': 0.5,
    'overall_accuracy': 0.8
}
'''
```

Exercise 1

Your company (Deeply) produces IoT smart devices and wants to automatically analyze a collection of Twitter messages. Your input is a sequence of strings (i.e., the tweets) crawled directly from Twitter.

Describe an NLP pipeline that is able to extract relevant knowledge from data. Assume that no annotated data are available at this stage.

`Input: tweets`

Exercise 1

- Analyze tweets to filter only the one that refers to Deeply
  - Load pre-trained NER model
  - Use it on sentences to identify ORG entity type matching Deeply
- Use filtered sentences with a sentiment classifier:
  - Load pretrained sentiment classification model
  - Apply on filtered sentences
  - Divide positive and negative sentences
- Try to extract value:
  - Use the NER model to detect the product names
  - Rank products according to their number of positive/negative tweets

Exercise 1b

Your company (Deeply) produces IoT
smart devices and wants to automatically analyze a
collection of Twitter messages. Your input is a sequence of strings (i.e., the tweets) crawled directly from Twitter.

Describe an NLP pipeline that is able to extract relevant knowledge from data. Assume that now you have a budget for annotating 5K data samples for a specific task.

How would you spend the budget? Justify your answer.

`Input: tweets`

Exercise 1b

- Analyze tweets to filter only the one that refers to Deeply
  - Load pre-trained NER model
  - Use it on sentences to identify ORG entity type matching Deeply
  - No annotation at this stage because ORG is usually available on NER datasets.
- Use filtered sentences with a sentiment classifier:
  - Load pretrained sentiment classification model
  - Apply on filtered sentences
  - Divide positive and negative sentences
  - No annotation at this stage because the sentiment is independent from company/product.
- Try to extract value:
  - Load pre-trained NER model
  - Finetune it to recognize Deeply product names using the 5k annotations (it is unlikely to find real-world annotated data for this particular task).
  - Rank products according to their number of positive/negative tweets

Exercise 1b

- Analyze tweets to filter only the one that refers to Deeply
  - Load pre-trained NER model
  - Use it on sentences to identify ORG entity type matching Deeply
  - No annotation at this stage because ORG is usually available on NER datasets.
- Use filtered sentences with a sentiment classifier:
  - Load pretrained sentiment classification model
  - Apply on filtered sentences
  - Divide positive and negative sentences
  - No annotation at this stage because sentiment is independent from company/product.

No unique answer, the reasons must be suitable for the use-case!

- Try to extract value:
  - Load pre-trained NER model
  - Finetune it to recognize Deeply product names using the 5k annotations (it is unlikely to find real-world annotated data for this particular task)
  - Rank products according to their number of positive/negative tweets

I may require annotation only for products that are not found by a Regular Expression-based NER model

Exercise 2

Your company (Deeply) has a human-driven customer care. However, almost 95% of the requests received can be mapped to only 5 questions.

Build an unsupervised NLP pipeline that take as input the common questions (FAQ) and the query of the user. The purpose is to reduce the time spent by the customer care employee in answering very common questions.

`Input: QA answers (list), user_query`

Exercise 2

- Use BERT model to encode `QA answers` into semantically-aware fixed-size vectors
  - Load pre-trained BERT model
    - Alternatively use pre-trained Sentence-BERT model
  - Obtain vector representation for the questions
- Compute semantic similarity between query and questions
  - Compute the embeddings for the user query
  - Compute the cosine similarity between the query and each question
  - Answer to the user with the most similar question.

Exercise 2b

Your company (Deeply) has a human-driven customer care. However, almost 95% of the requests received can be mapped to only 5 questions.

Propose a draft of the pseudocode to build an unsupervised pipeline that take as input the common questions (FAQ) and the query of the user. The purpose is to reduce the time spent by the customer care employee in answering very common questions.

```
Input: QA answers (list), user_query
```

Home Exercise

Lecture goals

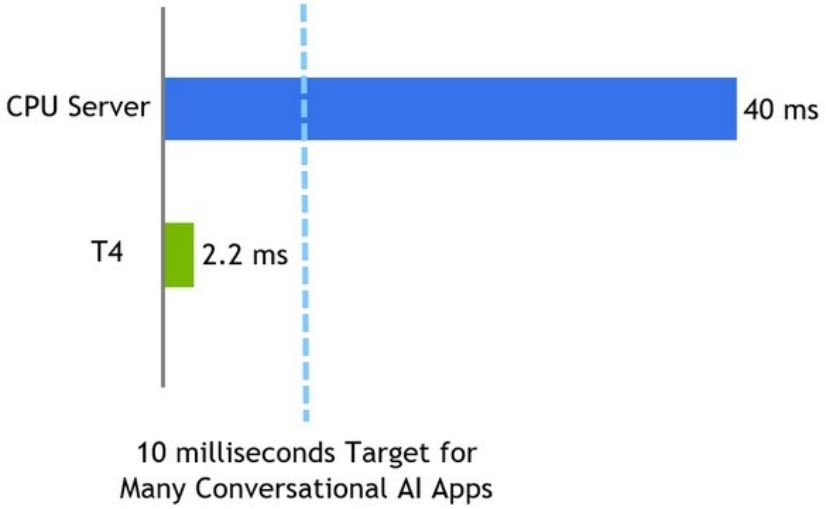
- Transformer architectures (recap)
  - Self- and Cross-Attention
  - Encoder/Decoder/Sequence-to-Sequence
  - Architectures (BERT, GPT, BART)
- Exercises using Transformers
- Task-specific Transformers (exercises)
  - Sequence classification
  - Entity recognition
- **From development to production**
  - Serving model for AI applications

Model deployment

- Deep learning models are able at solving several tasks
  - Text classification (e.g., sentiment analysis)
  - Text embeddings (e.g., for similarity search)
  - Text generation (e.g., as writing support tool)
  - ... and many others (as discussed in the remainder of this course)
- From a software engineering perspective
  - Model deployment is the process of integrating your model into an existing production environment.
  - It should be as fast and reliable as possible
  - It should be easy from a developer perspective (a lot of effort is already devoted to model training/testing...)

Model deployment

- Deep NLP models have strong computational requirements
  - Training the model is the most expensive part (it is done only once)
  - GPUs and TPUs greatly speed up the whole process



https://developer.nvidia.com/blog/nlu-with-tensorrt-bert/ @ 2019

What do you need?

Docker can be used to generate an environment where all dependencies are satisfied

Transformers library can be used to build DL-based NLP pipelines.

What do you need?

FastAPI can be used as a web framework to build API with python. It is sponsored as *production ready.*

Google Cloud Platform is one of several alternatives to host the generated web application on the cloud

References

1. [Post] The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) https://jalammar.github.io/illustrated-bert/

2. [Post] A Visual Guide to Using BERT for the First Time https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

3. [Post] The Illustrated Transformer https://jalammar.github.io/illustrated-transformer/

4. [Book] Mastering Transformers: Build state-of-the-art models from scratch with advanced natural language processing techniques https://www.packtpub.com/product/mastering-transformers/9781801077651

5. [Book] Tunstall, Lewis, Leandro von Werra, and Thomas Wolf. Natural language processing with transformers. " O'Reilly Media, Inc.", 2022.

6. Wolf, Thomas, et al. "Transformers: State-of-the-art natural language processing." Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. 2020.

Thank you!

TA contact: lorenzo.vaiani@polito.it

