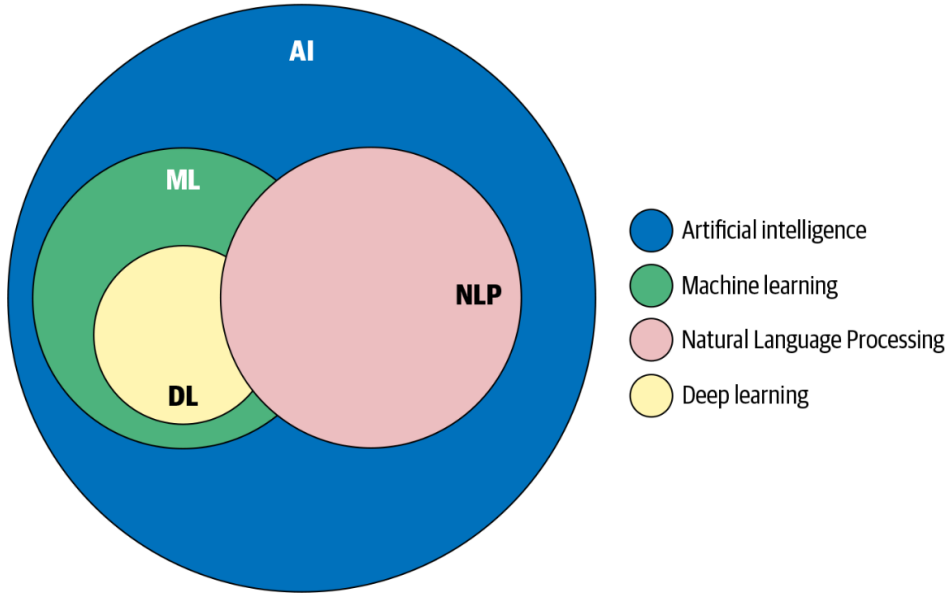
NLP – Transformers

Dr. Sarwan Singh

Scientist – D, NIELIT Chandigarh

Agenda

- Sequence-to-sequence (seq2seq) Models
- Recurrent Neural Network (RNN) Models
- Attention Mechanism
- Transformer
- BERT

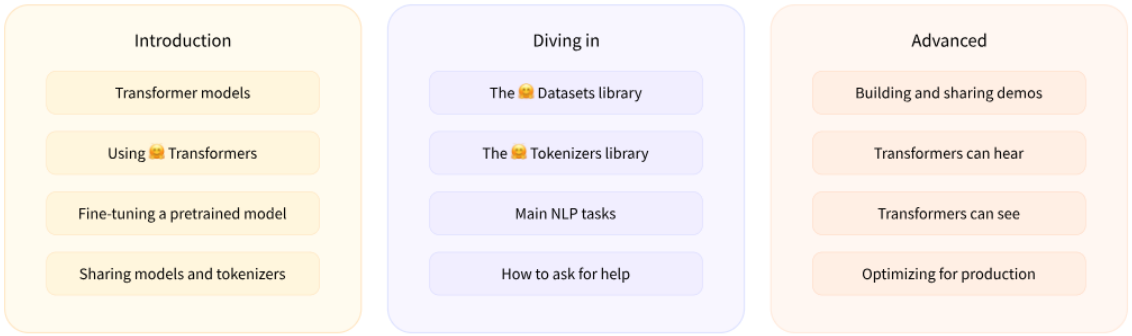


References

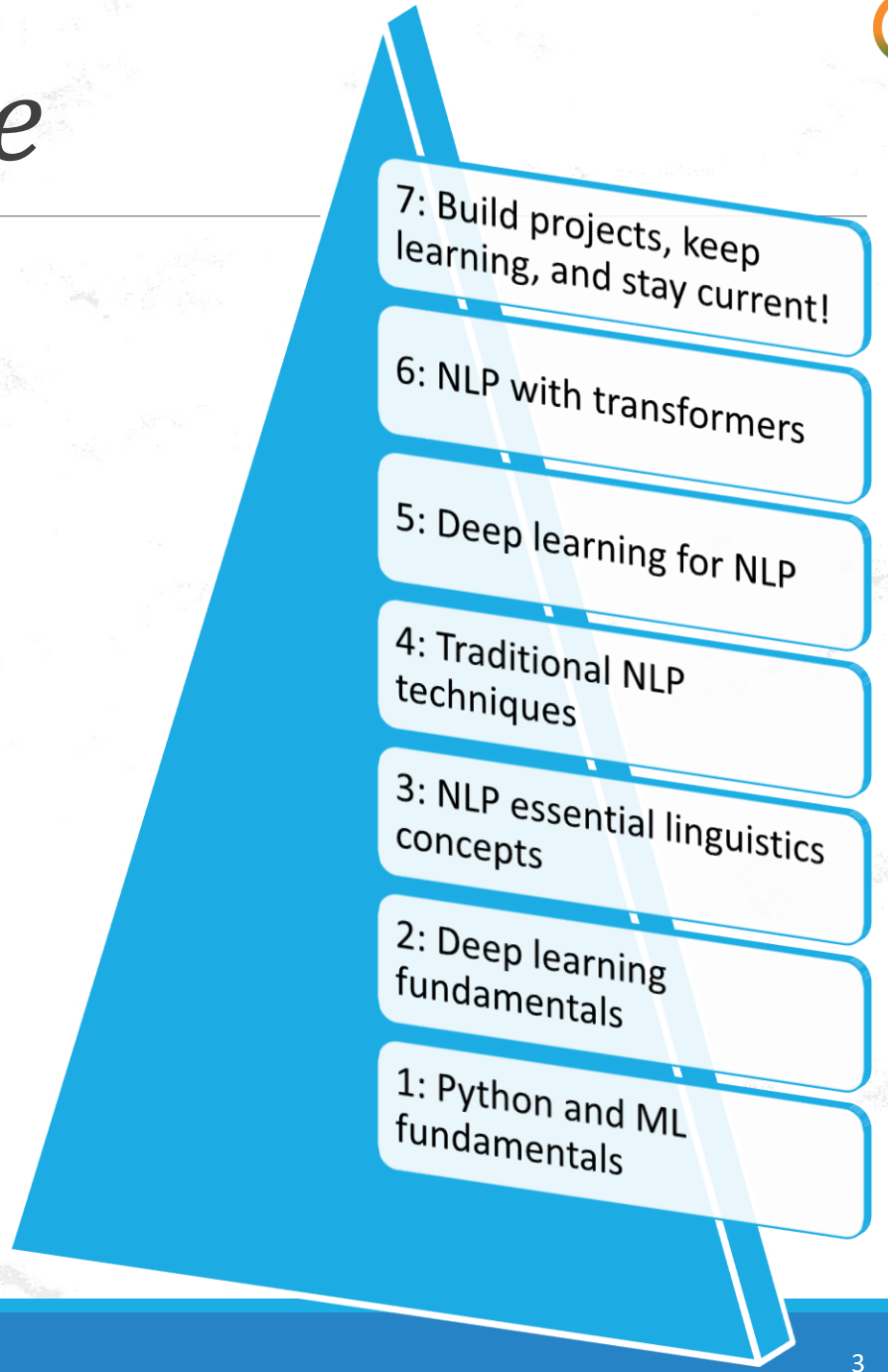
- towardsdatascience.com, huggingface.co, analyticsvidhya.com

NLP Ecosystem - *glance*

- 1: Python and ML fundamentals
- 2: Deep learning fundamentals
- 3: NLP essential linguistics concepts
- 4: Traditional NLP techniques
- 5: Deep learning for NLP
- 6: NLP with transformers
- 7: Build projects, keep learning, and stay current!



NLP - paradigm at Huggingface.co



Transformers are everywhere

- Transformer models are used to solve all kinds of NLP tasks.
- some of the companies and organizations using Hugging Face and Transformer models, who also contribute back to the community by sharing their models:

- The Transformers library provides the functionality to create and use those shared models. The Model Hub contains thousands of pretrained models that anyone can download and use. You can also upload your own models to the Hub!

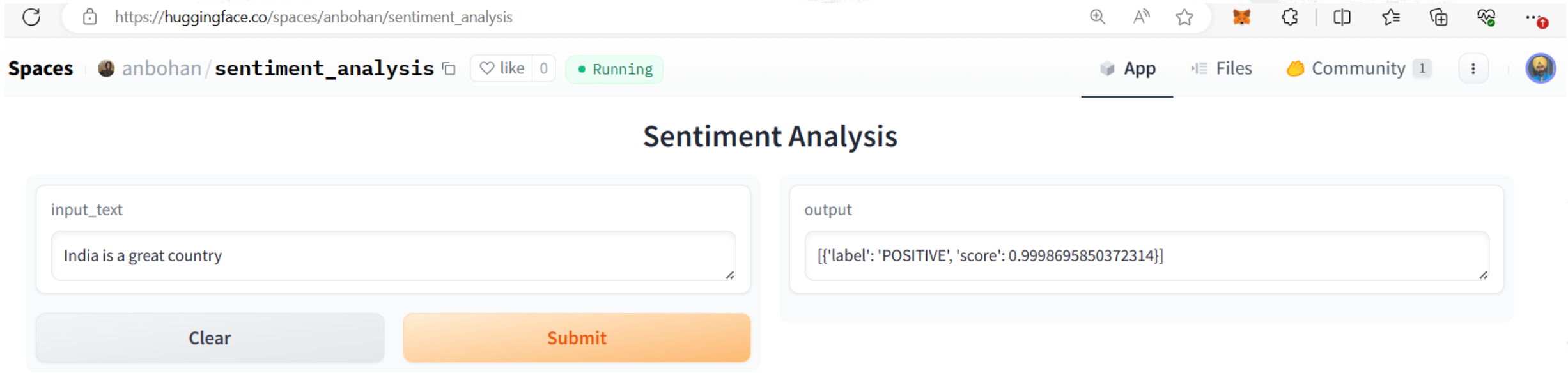
Transformers are everywhere

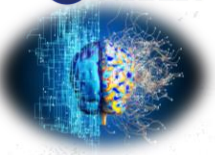
- The most basic object in the Transformers library is the pipeline() function. It connects a model with its necessary preprocessing and post-processing steps, allowing us to directly input any text and get an intelligible answer

  - from transformers import pipeline
  - classifier = pipeline("sentiment-analysis")
  - classifier("I've been waiting for a HuggingFace course my whole life.")

  - [{'label': 'POSITIVE', 'score': 0.9598047137260437}]

```
1. import gradio as gr
2. from transformers import pipeline
3. sentiment = pipeline("sentiment-analysis")
4. def get_sentiment(input_text):
5.     return sentiment(input_text)
6. iface7 = gr.Interface(get_sentiment, inputs= "text", outputs= "text", title="Sentiment Analysis")
7. iface7.launch(inline=False)
```

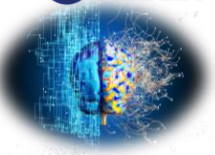




We can even pass several sentences!

1. classifier(
2. ["I've been waiting for a HuggingFace course my whole life.", "I hate this so much!"]
3.)

- output
  1. [{'label': 'POSITIVE', 'score': 0.9598047137260437},
  2. {'label': 'NEGATIVE', 'score': 0.9994558095932007}]



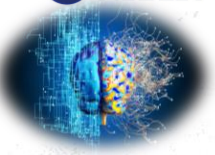
Transformers are everywhere

- There are three main steps involved when you pass some text to a pipeline:

1. The text is preprocessed into a format the model can understand.
2. The preprocessed inputs are passed to the model.
3. The predictions of the model are post-processed, so you can make sense of them.

Some of the available pipelines

- feature-extraction (get the vector representation of a text)
- fill-mask
- NER (named entity recognition)
- question-answering
- sentiment-analysis
- summarization
- text-generation
- translation
- zero-shot-classification



Zero-shot classification

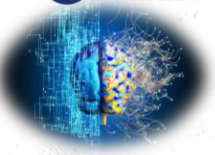
- annotating text is usually time-consuming and requires domain expertise.
- For this use case, the zero-shot-classification pipeline is very powerful: it allows you to specify which labels to use for the classification, so you don't have to rely on the labels of the pretrained model.
- We already seen how the model can classify a sentence as positive or negative using those two labels — but it can also classify the text using any other set of labels you like.

Zero-shot classification

- from transformers import pipeline
- classifier = pipeline("zero-shot-classification")
- classifier(
- "This is a course about the Transformers library",
- candidate_labels=["education", "politics", "business"],
-)

```
{'sequence': 'This is a course about the Transformers library',
 'labels': ['education', 'business', 'politics'],
 'scores': [0.8445963859558105, 0.111976258456707, 0.043427448719739914]}
```

This pipeline is called *zero-shot* because you don't need to fine-tune the model on your data to use it. It can directly return probability scores for any list of labels we want



Text Generation

- use a pipeline to generate some text
- The main idea here is that you provide a prompt and the model will auto-complete it by generating the remaining text.
- This is similar to the predictive text feature that is found on many phones.
- Text generation involves randomness, so it’s normal if you don’t get the same results

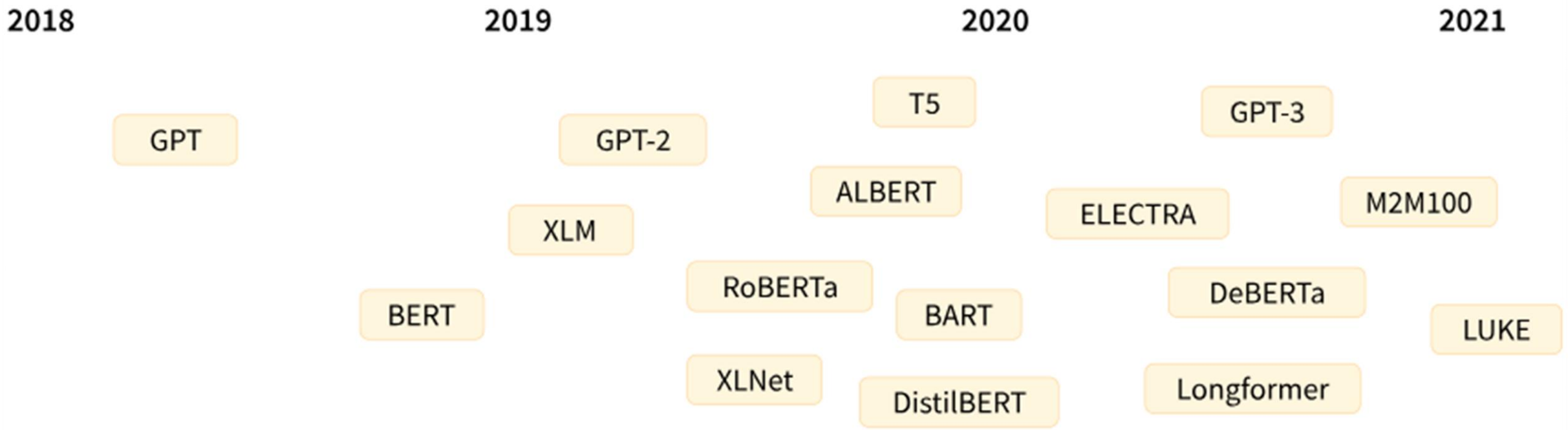
Text Generation

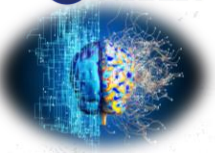
- from transformers import pipeline
- generator = pipeline("text-generation")
- generator("In this course, we will teach you how to")

```
[{'generated_text': 'In this course, we will teach you how to understand and use '
                    'data flow and data interchange when handling user data. We '
                    'will be working with one or more of the most commonly used '
                    'data flows — data flows of various types, as seen by the '
                    'HTTP'}]
```

- You can control how many different sequences are generated with the argument num_return_sequences and the total length of the output text with the argument max_length

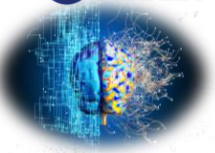
A bit of Transformer history





A bit of Transformer history

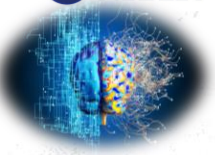
- The Transformer architecture was introduced in June 2017. The focus of the original research was on translation tasks. This was followed by the introduction of several influential models, including:
- June 2018: GPT, the first pretrained Transformer model, used for fine-tuning on various NLP tasks and obtained state-of-the-art results
- October 2018: BERT, another large pretrained model, this one designed to produce better summaries of sentences (more on this in the next chapter!)
- February 2019: GPT-2, an improved (and bigger) version of GPT that was not immediately publicly released due to ethical concerns
- October 2019: DistilBERT, a distilled version of BERT that is 60% faster, 40% lighter in memory, and still retains 97% of BERT's performance
- October 2019: BART and T5, two large pretrained models using the same architecture as the original Transformer model (the first to do so)
- May 2020, GPT-3, an even bigger version of GPT-2 that is able to perform well on a variety of tasks without the need for fine-tuning (called zero-shot learning)



A bit of Transformer history

- This list is far from comprehensive, and is just meant to highlight a few of the different kinds of Transformer models. Broadly, they can be grouped into three categories:

- GPT-like (also called auto-regressive Transformer models)
- BERT-like (also called auto-encoding Transformer models)
- BART/T5-like (also called sequence-to-sequence Transformer models)

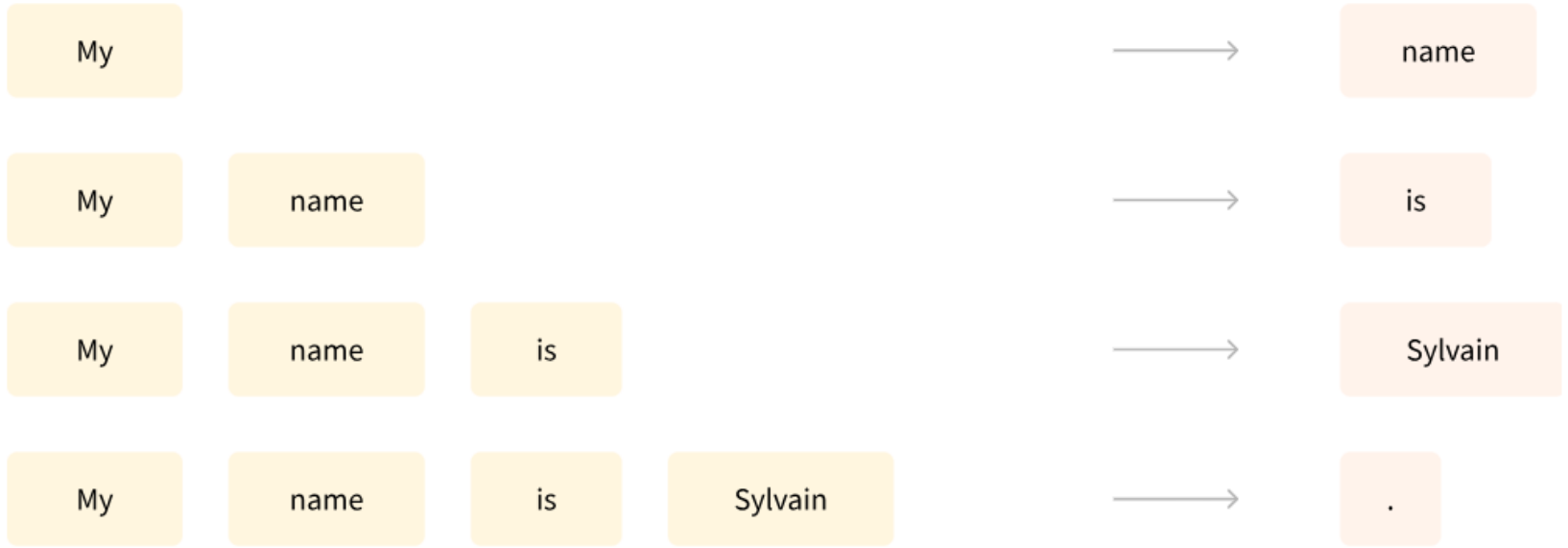


Transformers are language models

- All the Transformer models mentioned above (GPT, BERT, BART, T5, etc.) have been trained as *language models*. This means they have been trained on large amounts of raw text in a self-supervised fashion.
- Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model. That means that humans are not needed to label the data!
- This type of model develops a statistical understanding of the language it has been trained on, but it's not very useful for specific practical tasks.
- Because of this, the general pretrained model then goes through a process called *transfer learning*. During this process, the model is fine-tuned in a supervised way — that is, using human-annotated labels — on a given task.

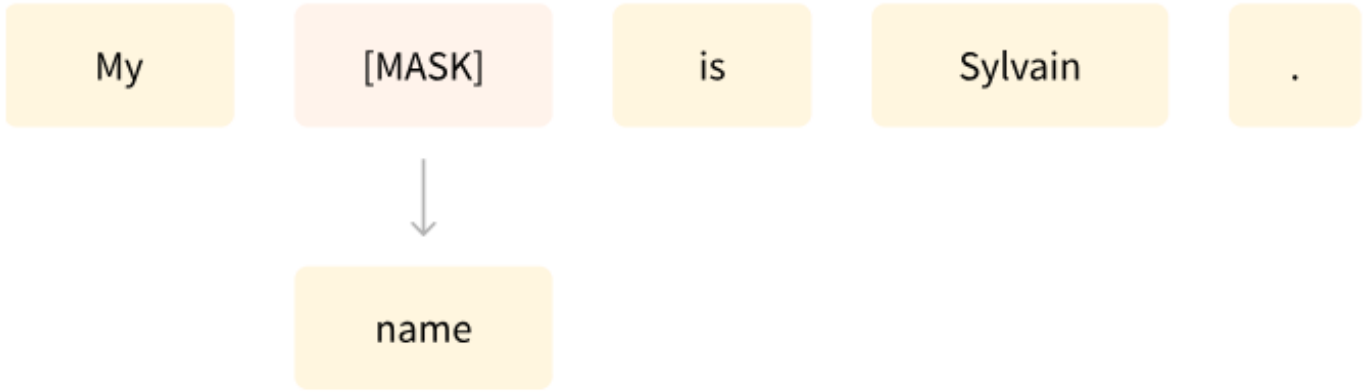
Transformers are language models

- An example of a task is predicting the next word in a sentence having read the *n* previous words.
- This is called *causal language modeling* because the output depends on the past and present inputs, but not the future ones.



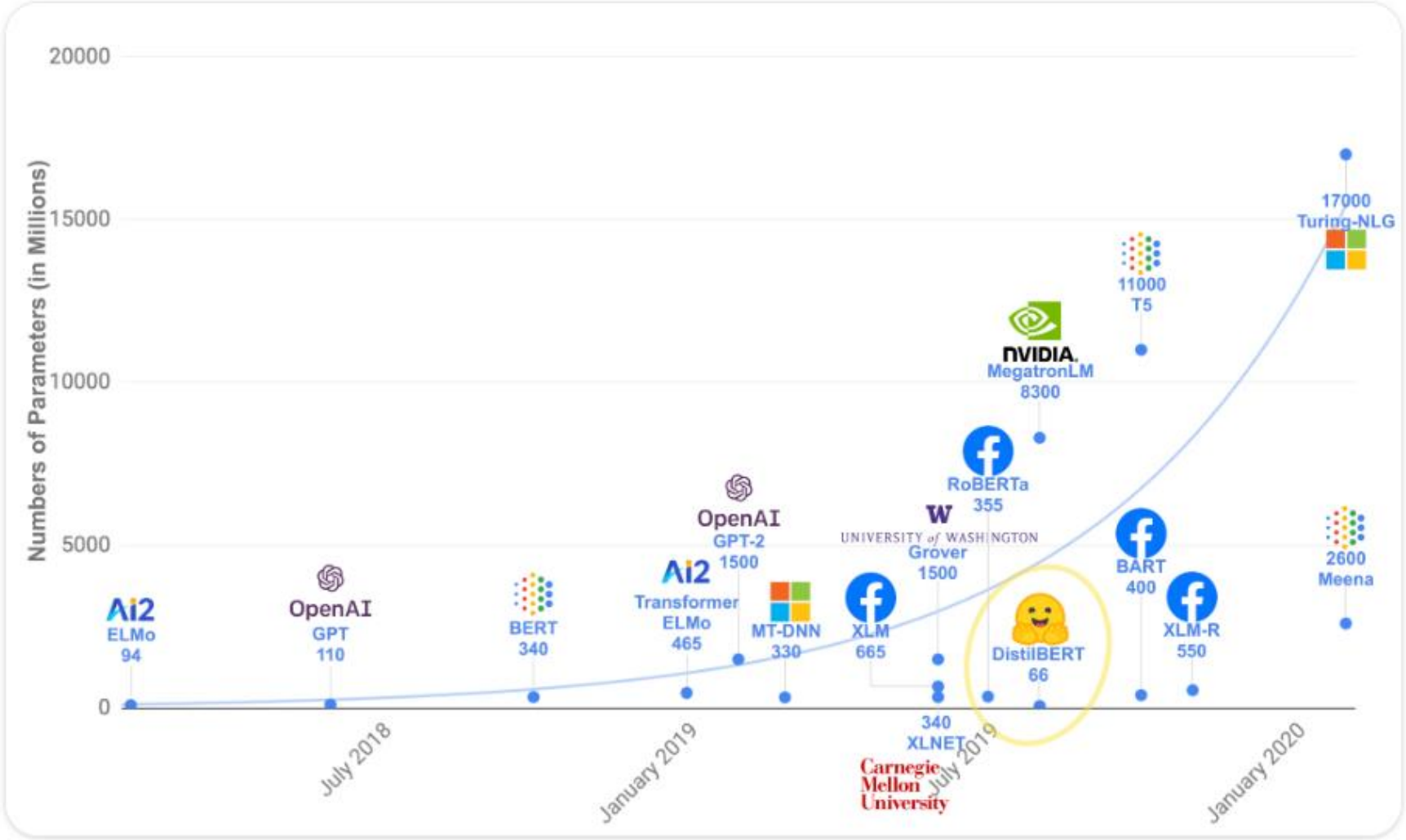
Transformers are language models

- Another example is *masked language modeling*, in which the model predicts a masked word in the sentence.



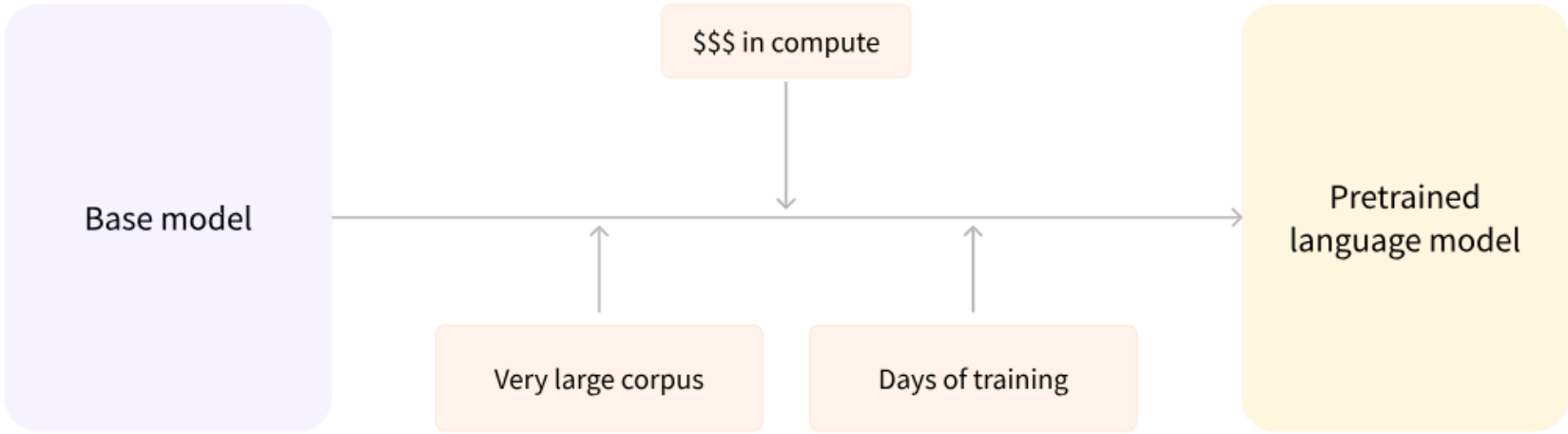
Transformers are big models

- Apart from a few outliers (like DistilBERT), the general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on.

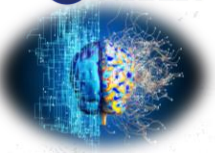


Transformers are big models

- *Pretraining* is the act of training a model from scratch: the weights are randomly initialized, and the training starts without any prior knowledge



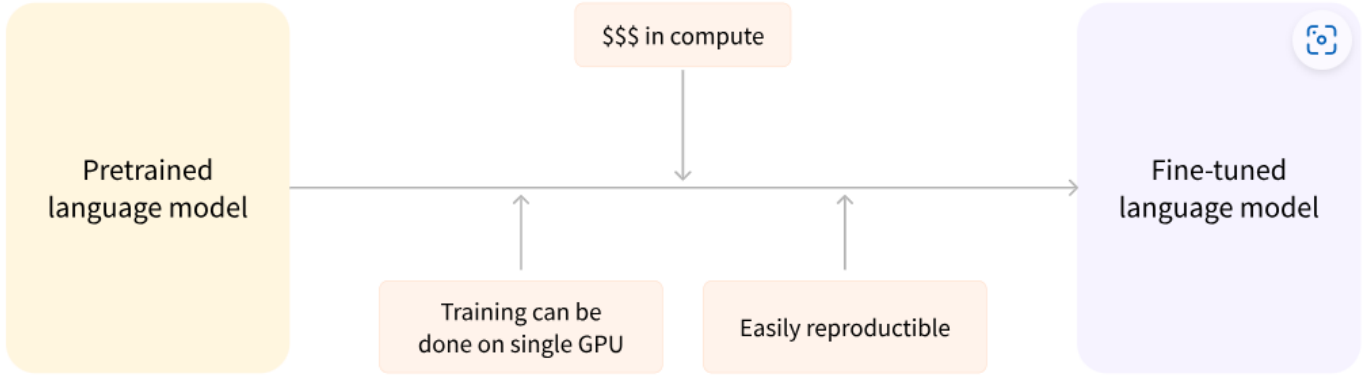
- This pretraining is usually done on very large amounts of data. Therefore, it requires a very large corpus of data, and training can take up to several weeks.



Transformers are big models

- *Fine-tuning*, on the other hand, is the training done **after** a model has been pretrained. To perform fine-tuning, you first acquire a pretrained language model, then perform additional training with a dataset specific to your task.
- Reasons why we cannot start from scratch:
  - The pretrained model was already trained on a dataset that has some similarities with the fine-tuning dataset. The fine-tuning process is thus able to take advantage of knowledge acquired by the initial model during pretraining (for instance, with NLP problems, the pretrained model will have some kind of statistical understanding of the language you are using for your task).
  - Since the pretrained model was already trained on lots of data, the fine-tuning requires way less data to get decent results.
  - For the same reason, the amount of time and resources needed to get good results are much lower.

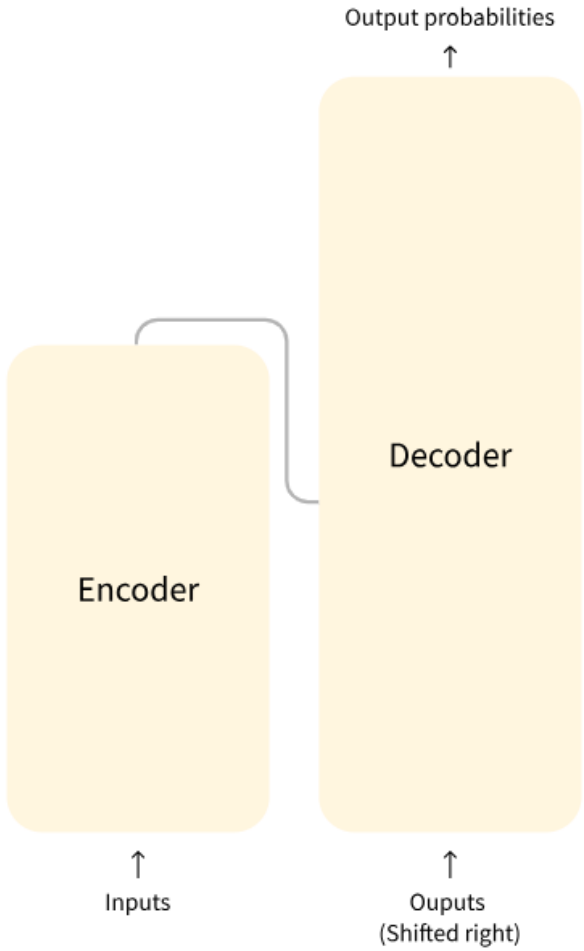
- For example, one could leverage a pretrained model trained on the English language and then fine-tune it on an arXiv corpus, resulting in a science/research-based model. The fine-tuning will only require a limited amount of data: the knowledge the pretrained model has acquired is "transferred," hence the term *transfer learning*.

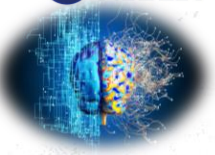


- Fine-tuning a model therefore has lower time, data, financial, and environmental costs. It is also quicker and easier to iterate over different fine-tuning schemes, as the training is less constraining than a full pretraining.

General Architecture of Transformer

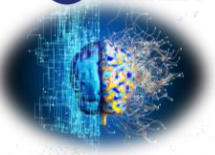
- **Encoder (left)**: The encoder receives an input and builds a representation of it (its features). This means that the model is optimized to acquire understanding from the input.
- **Decoder (right)**: The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence. This means that the model is optimized for generating outputs.





Transformers are big models

- Each of these parts can be used independently, depending on the task:
  - **Encoder-only models**: Good for tasks that require understanding of the input, such as sentence classification and named entity recognition.
  - **Decoder-only models**: Good for generative tasks such as text generation.
  - **Encoder-decoder models** or **sequence-to-sequence models**: Good for generative tasks that require an input, such as translation or summarization.

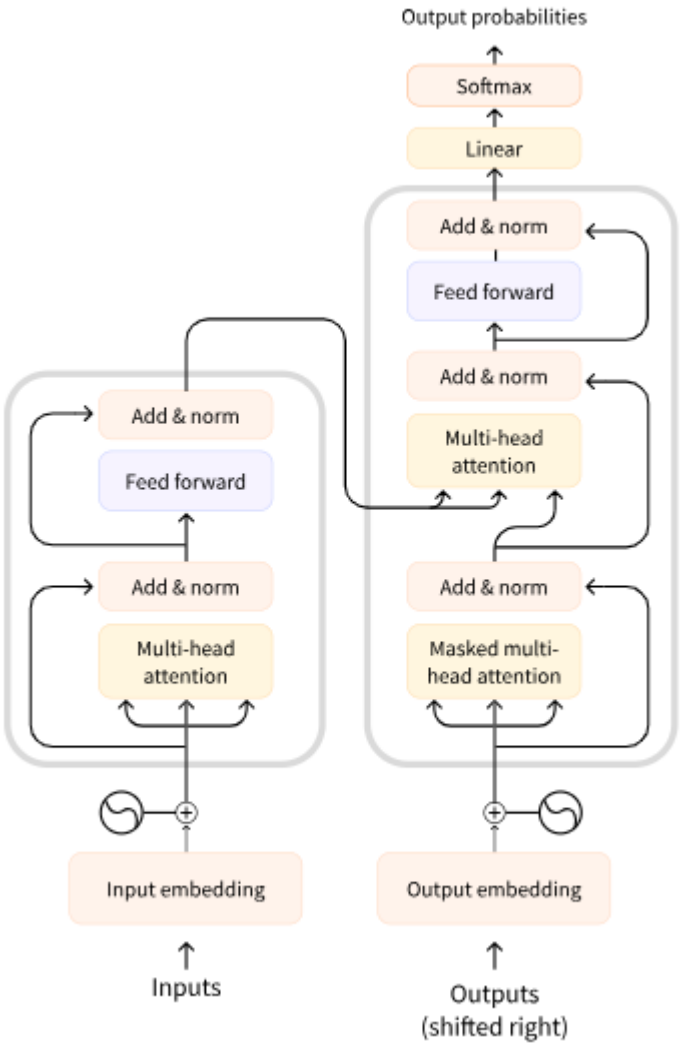


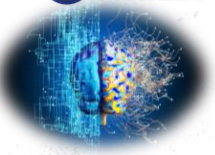
The original architecture

- The Transformer architecture was originally designed for translation.
- During training, the encoder receives inputs (sentences) in a certain language, while the decoder receives the same sentences in the desired target language.
- In the encoder, the attention layers can use all the words in a sentence (translation of a given word can be dependent on what is after as well as before it in the sentence).
- The decoder, however, works sequentially and can only pay attention to the words in the sentence that it has already translated (so, only the words before the word currently being generated).
- For example, when we have predicted the first three words of the translated target, we give them to the decoder which then uses all the inputs of the encoder to try to predict the fourth word.

The original architecture

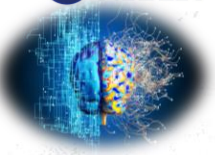
- To speed things up during training (when the model has access to target sentences), the decoder is fed the whole target, but it is not allowed to use future words (if it had access to the word at position 2 when trying to predict the word at position 2, the problem would not be very hard!). For instance, when trying to predict the fourth word, the attention layer will only have access to the words in positions 1 to 3.





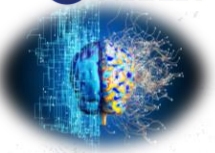
The original architecture

- The first attention layer in a decoder block pays attention to all (past) inputs to the decoder, but the second attention layer uses the output of the encoder. It can thus access the whole input sentence to best predict the current word.
- This is very useful as different languages can have grammatical rules that put the words in different orders, or some context provided later in the sentence may be helpful to determine the best translation of a given word.
- The *attention mask* can also be used in the encoder/decoder to prevent the model from paying attention to some special words — for instance, the special padding word used to make all the inputs the same length when batching together sentences.



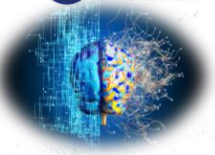
Architectures vs. checkpoints

- **Architecture**: This is the skeleton of the model — the definition of each layer and each operation that happens within the model.
- **Checkpoints**: These are the weights that will be loaded in a given architecture.
- **Model**: This is an umbrella term that isn't as precise as "architecture" or "checkpoint": it can mean both. This course will specify *architecture* or *checkpoint* when it matters to reduce ambiguity.
- For example, BERT is an architecture while bert-base-cased, a set of weights trained by the Google team for the first release of BERT, is a checkpoint. However, one can say "the BERT model" and "the bert-base-cased model."



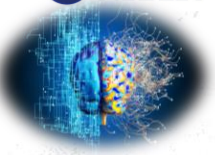
Encoder Model

- Encoder models use only the encoder of a Transformer model. At each stage, the attention layers can access all the words in the initial sentence. These models are often characterized as having "bi-directional" attention, and are often called *auto-encoding models*.
- The pretraining of these models usually revolves around somehow corrupting a given sentence (for instance, by masking random words in it) and tasking the model with finding or reconstructing the initial sentence.
- Encoder models are best suited for tasks requiring an understanding of the full sentence, such as sentence classification, named entity recognition (and more generally word classification), and extractive question answering.



Decoder models

- Decoder models use only the decoder of a Transformer model. At each stage, for a given word the attention layers can only access the words positioned before it in the sentence. These models are often called *auto-regressive models*.
- The pretraining of decoder models usually revolves around predicting the next word in the sentence.
- These models are best suited for tasks involving text generation



Major Representatives in family of models

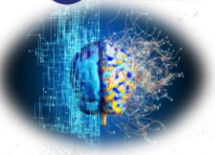
Encoder family of models :

- ALBERT
- BERT
- DistilBERT
- ELECTRA
- RoBERTa

- **Decoder family of models :**
- CTRL
- GPT
- GPT-2
- Transformer XL

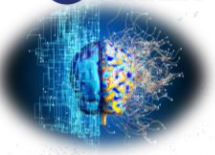
Sequence-to-sequence models

- Encoder-decoder models (also called *sequence-to-sequence models*) use both parts of the Transformer architecture. At each stage, the attention layers of the encoder can access all the words in the initial sentence, whereas the attention layers of the decoder can only access the words positioned before a given word in the input.
- The pretraining of these models can be done using the objectives of encoder or decoder models, but usually involves something a bit more complex. For instance, **T5** is pretrained by replacing random spans of text (that can contain several words) with a single mask special word, and the objective is then to predict the text that this mask word replaces.



Sequence-to-sequence models

- Sequence-to-sequence models are best suited for tasks revolving around generating new sentences depending on a given input, such as summarization, translation, or generative question answering.
- Representatives of this family of models include:
  - BART
  - mBART
  - Marian
  - T5



Bias and limitations

- If your intent is to use a pretrained model or a fine-tuned version in production, please be aware that, while these models are powerful tools, they come with limitations.
- The biggest of these is that, to enable pretraining on large amounts of data, researchers often scrape all the content they can find, taking the best as well as the worst of what is available on the internet.

```
from transformers import pipeline

unmasker = pipeline("fill-mask", model="bert-base-uncased")
result = unmasker("This man works as a [MASK].")
print([r["token_str"] for r in result])

result = unmasker("This woman works as a [MASK].")
print([r["token_str"] for r in result])
```

```
['lawyer', 'carpenter', 'doctor', 'waiter', 'mechanic']
['nurse', 'waitress', 'teacher', 'maid', 'prostitute']
```

Summary

Model	Examples	Tasks
Encoder	ALBERT, BERT, DistilBERT, ELECTRA, RoBERTa	Sentence classification, named entity recognition, extractive question answering
Decoder	CTRL, GPT, GPT-2, Transformer XL	Text generation
Encoder-decoder	BART, T5, Marian, mBART	Summarization, translation, generative question answering

Summary

Any Queries ?