Intro to AI Transformers

What are Transformers

- Specific type of NN
- NN: input layer ⇒ hidden layer (adjusts weights) ⇒ output layer
- more hidden layer, more complexity
- Models scale with Data
- transformers are data agnostic and work well with sequential data of any type!

https://www.youtube.com/watch?v=SMZQrJ_L1vo

GPT: Generative pretrained transformer

- Eg:
  - Text: Language models (BERT, BART, GPT)
  - Image: ViT (Vision Transformer)
  - Audio: Whisper
  - Video: ViViT (Video Vision Transformer)
  - Protein Sequencing: proteinBERT
- Transformers have the ability to do transfer learning
  - ⇒ Knowledge aquired by pre-trained models is used. Hence transfer learning
  - PreTraining: learn from mass data
  - FineTuning : learn from specific data

	Pretraining task example	***Finetuning task example***
Image data	Identifying automobiles	Identifying trucks
Text data	Acquiring the ability to generate English language sentences	Acquiring specific expertise in mimicking Shakespeare

LM vs LLM

Traditional Vs Neural Language models

Older	Newer
n gram and statistical methods	
Struggle to capture long range dependencies LRD	

Older:

> *"The concert was amazing. The atmosphere at the venue was out of this world and the crowd couldn't stop cheering"*

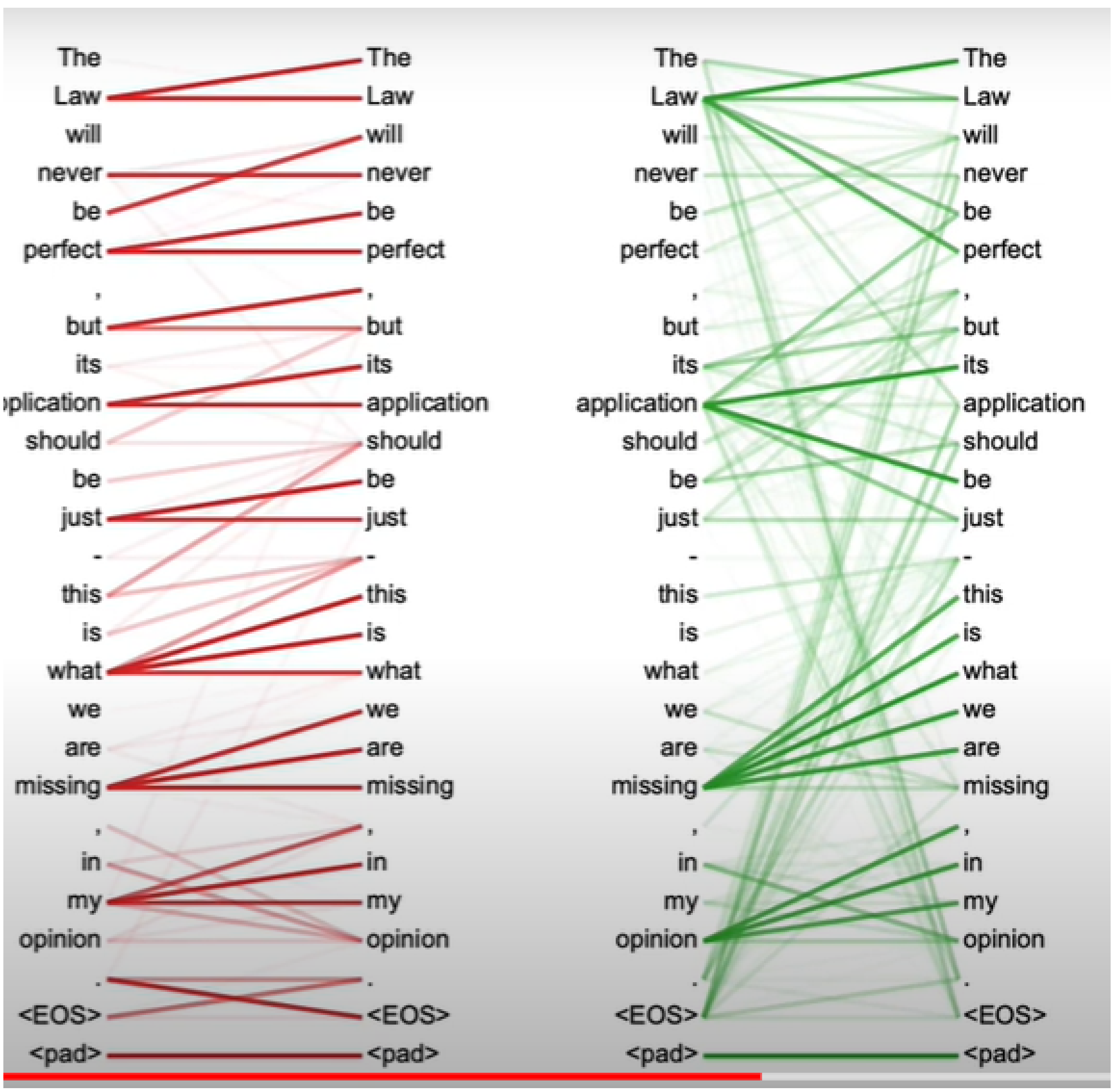
- understanding why crowed couldnt stop cheering requires the knowlege of the concert was amazing.
- In Count based models, farther these two sentences are, more difficult to understand

Newer:

Recurrent Neural Networks (RNNs) and LSTMs (Long Short term memory)

Transformers based:

- self_attention
    - the model assess all the words in the input while processing each individual word
    - each word is assigned weights based on relevance
    -



▼ History

Transformers and Language Models

Introduction of Transformers:

- **2017:** Transformer architecture introduced in the paper "Attention Is All You Need" by Vaswani et al.
- **2018 (June):** OpenAI released GPT-1, the first generative pre-trained transformer model, capable of generating coherent and contextually relevant text.
- **2018 (October):** Google released BERT (Bi-directional Encoder Representations from Transformers), designed for understanding text relationships and dependencies, useful for tasks like sentiment analysis, named entity recognition, and extractive question answering.

2019 Developments:

- **February:** OpenAI released GPT-2.
- **October:** Release of DistilBERT (a lighter, faster version of BERT), Facebook's BART, and Google's T5. BART and T5 were optimized for text generation tasks such as summarization and translation.

Growth of Model Sizes:

- **2019:** GPT-2 had 1.5 billion parameters.
- **2020 (May):** OpenAI released GPT-3 with 175 billion parameters, trained on 45 terabytes of text data.
- **Parameter Growth:** GPT-1 had 0.2 billion parameters; parameter sizes have been trending upwards due to performance scaling with the size of training data and compute.

2022-2023 Advancements:

- **2022:**
    - **Google's Lambda:** Specialized in conversation response generation.

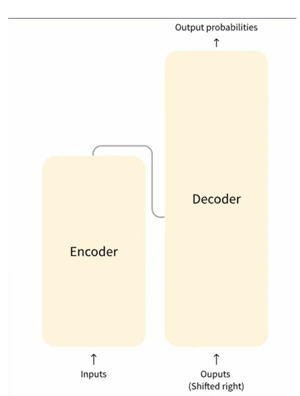
    - **November:** OpenAI released ChatGPT, a chatbot based on GPT-3.5, publicly available for a limited time.
- **2023:** Rumors of GPT-4 release with an estimated 1.76 trillion parameters.

Key Points

- **Transformer Architecture:** Introduced in 2017, foundational for modern language models.
- **GPT Series:** Progression from GPT-1 (2018) to GPT-3 (2020), with exponential growth in parameters.
- **BERT and Derivatives:** Focused on understanding text, BERT released in 2018, with lighter/faster versions and related models like DistilBERT, BART, and T5 in 2019.
- **Model Size and Performance:** Increased parameter sizes enhance performance, evident in the progression from GPT-1 to GPT-3 and beyond.
- **Recent Models:** Specialized models like Lambda for conversation generation and the chatbot ChatGPT.

How Do they Work?

- Transformer (type of NN) = Encoder (type of NN) + Decoder (type of NN)
- It can be Encoder only or Decoder only
-



what are Encoders and Decoders?

- both NN with many layers
- They work with **embeddings:** mathematical representations of the input data
  - Tiger—Lion (similar word embedding)
  - Strawberry—Apple (similar word embedding)
  - Strawberry —× lion (**far apart)**
- both have **attention layer:** learnt from training data through self-attention
  - attention mask: tensor which tells which token can be accessed
  - this helps to understand how each part of the input sequence relates to another
- Difference between them is how they approach self attention

What are tokens

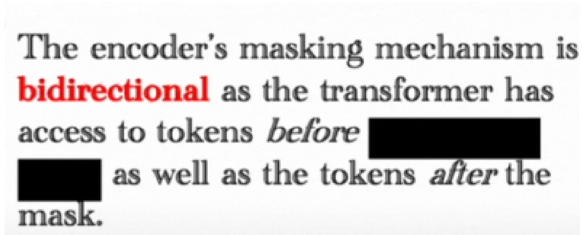
- subwords/ words that the text is broken into

How does Self-Attention work

- A part of the sentence that model needs to predict is Masked. Model needs to guess what could go there. This is called masking
- Self attention helps model to learn contextual information
- Input is given a **positional encoding** : tells where <something> is in input sequence
- Self-attention is an iterative process where the model learns rich contextual information about the role of each part within a sequence.

Encoders:

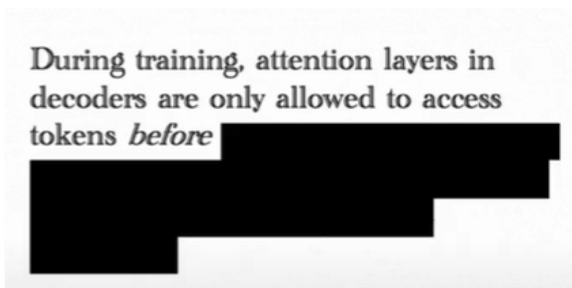
- masking: random
- Bidirectionality : **attention layers** can access inputs both sides of mask


The encoder's masking mechanism is **bidirectional** as the transformer has access to tokens *before* [redacted] as well as the tokens *after* the mask.

- BERT: Bidirectionally Encoded Representations from Transformers

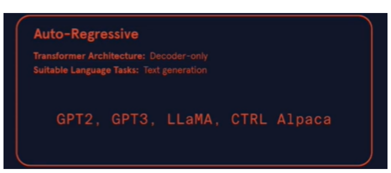
Decoders:

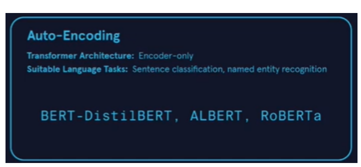
- masking: everything **after** the token to be predicted is masked
- Unidirectional: **attention layers** can access inputs only one side of mask(before the mask)
- Good at text generation

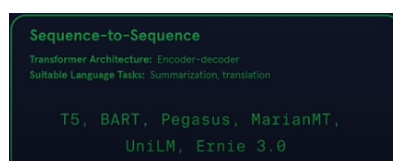

During training, attention layers in decoders are only allowed to access tokens *before* [redacted]

- GPT: Generative Pretrained transformer

Types of Transformer



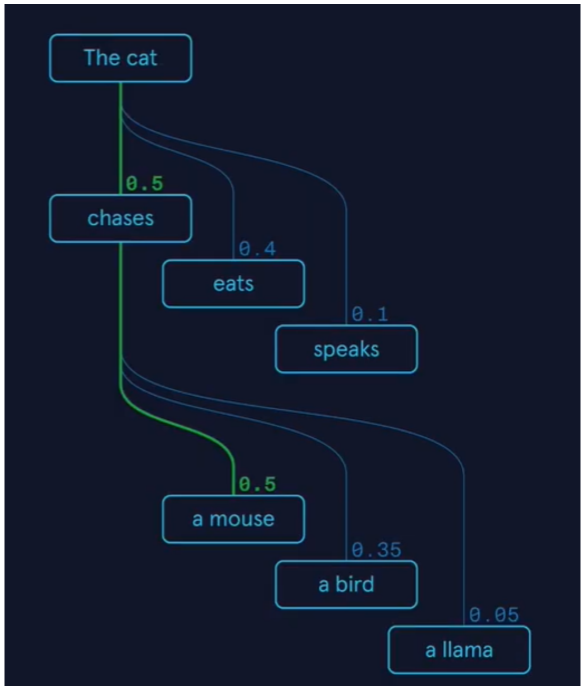




1. Auto regressive (GPT type)

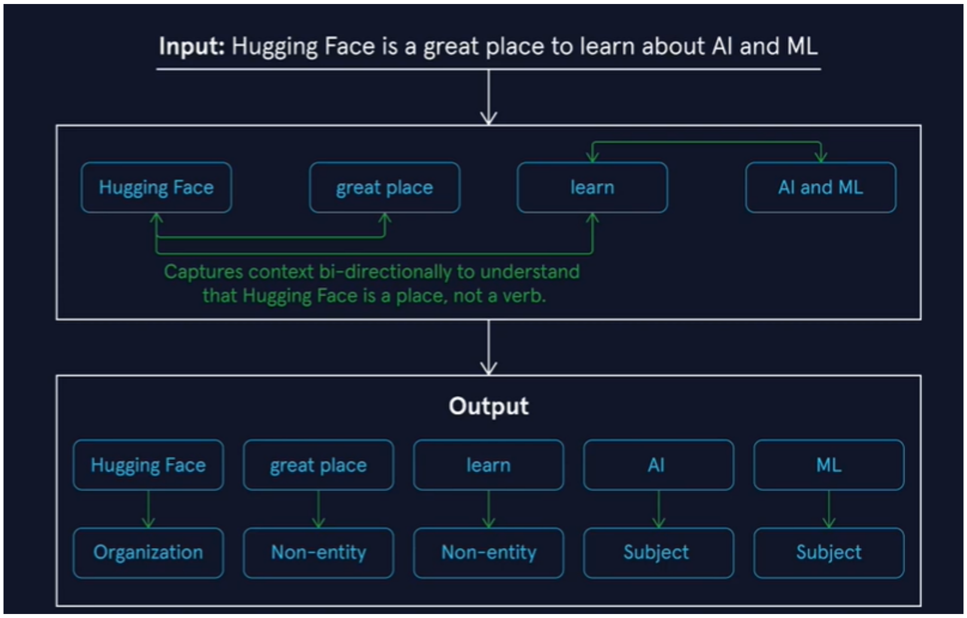
- Decoder only

- Generates token based on predicted probabilities
- Text generation



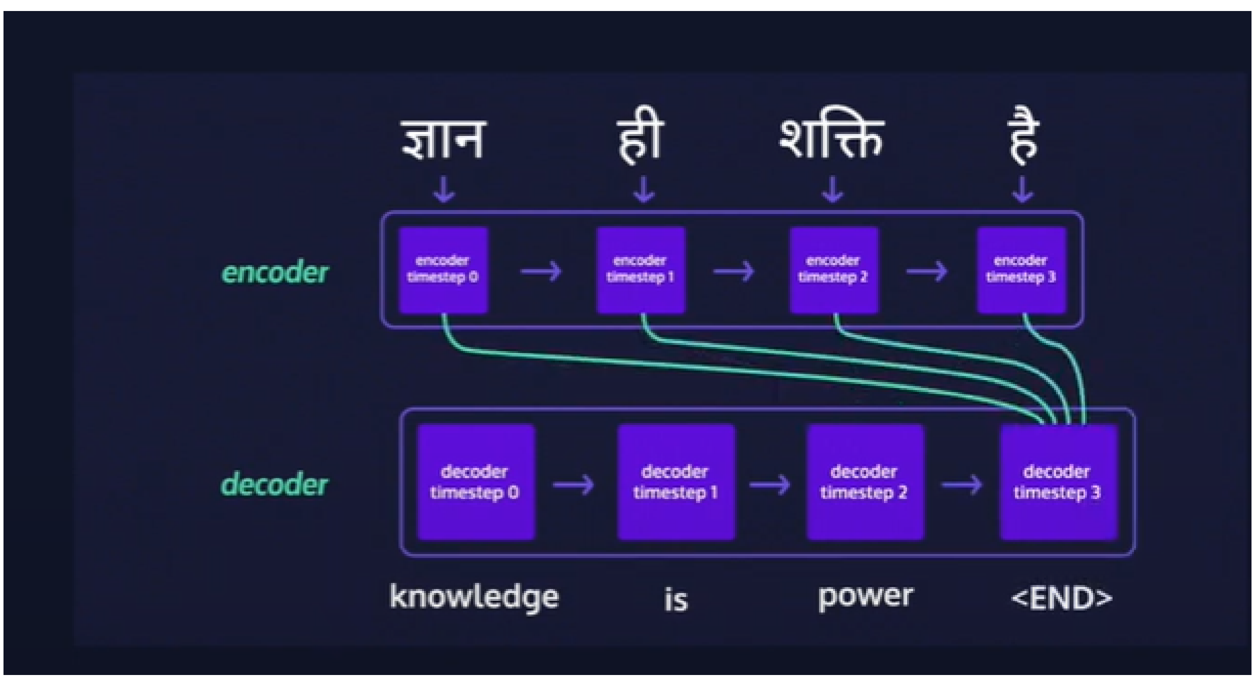
2. Auto Encoding (BERT like)

- Encoder only architecture
- Contextual Understanding: Named entity recognition, Sentiment analysis, Extractive question answering
-



3. Sequence to sequence(BART/T5 like)

- Encoder and Decoder architecture
- Transforming one sequence of data to another sequence of data
- Tasks: Translations, Summarization, generative question answering
-



Transformers + Hugging Face

```
from transformers import pipeline

classifier = pipeline(task = "sentiment-analysis",
                                            model = "distilbert-
```

```
text = ["I've been waiting to learn about transformers my whole
               "I hate this so much!"]

classifier(text)
```

Behind the Pipeline

- combines 3 steps

  a) pre-processing

  b) giving inputs to the model

  c) post processing

a) Pre-processing

- text ⇒ array of numbers. how?
  - Text ⇒ words, subwords, punctuations. TOKENS
  - Tokens are mapped to numbers and additional relevant inputs are added
- Pre-processing should be the same as the method used during model pretraining (get it from model hub)

```
from transformers import AutoTokenizer
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
```

```
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

b) Going through model

- `AutoModel.from_pretrained()` : General purpose transformer, You will have to use your own classification logic on top of this
- `AutoModelForSequenceclassification` : Model Heads helps us for specific tasks

```
from transformers import AutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(checkp
```

c) Postprocessing

- output of the transformer model will be just "logits" (raw unnormalized scores)
- Logits need to be passed through softmax function

Full example

```
from transformers import AutoTokenizer, AutoModelForSequenceClas
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
```

```
# Initializing the tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# Raw inputs
raw_inputs = ["I've been waiting to learn about transformers my
              "I hate this so much!"]

## YOUR SOLUTION HERE ##
inputs = tokenizer(raw_inputs, padding=True, truncation =True, r
inputs
```

```
# Initializing the model
model = AutoModelForSequenceClassification.from_pretrained(checl

## YOUR SOLUTION HERE ##
outputs = model(**inputs)
print(outputs.logits.shape)
print(outputs.logits)
```

```
import torch

# Converting the tensor output to a probability distribution
predictions = torch.nn.functional.softmax(outputs.logits, dim=-:

## YOUR SOLUTION HERE ##
print(predictions)
print(model.config.id2label)
```

Models

Architecture : Skeleton of the model. Definition of each layer

⇒ GPT, BERT, T5 are specific model architectures

Checkpoint: A set of weights generated through pre-training and finetuning a model architecture on specific data

- BERT ⇒ model architecture
- bert-base-uncased weights of bert
- distilbert-base-uncased-finetuned-sst-2-english : finetuned on sst-2 dataset
  - (sst-2: stanford sentiment treebank corpus)
  - Model Hub hosts all model checkpoints

- `AutoModel` : initiate a checkpoint
    - no model specified, it will assume a model based on the specified task
- If we know what model to use:
    - eg: Bert

    ```
    from transformers import BertConfig, BertModel
    config  = BertConfig()
    model = BertModel(config)
     # not trained, random weights assigned
    ```

  To train a model from scratch requires a lot of computation, hence use pre trained model

```
from transformers import BertModel
model = BertModel.from_pretrained("bert-base-cased")
#Model initiaalized with the weights of the checkpoint
```

Model Cards:

- parameter size
- training data
- task specific training
- in short:

    - technical details about the model
    - its intended uses & potential limitations, including biases and ethical considerations (as detailed in Mitchell, 2018)
    - the training parameters and experimental info

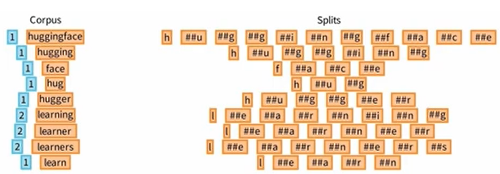
- details about the datasets used to train the model
- evaluation results about the model’s performance

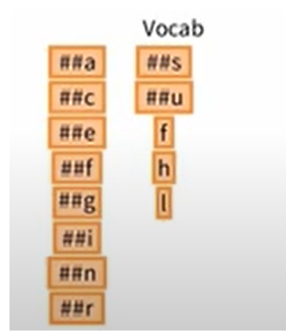
Tokenizers:

- Transformer models only process numbers, we need to convert text/audio/video ⇒ Numbers

1. Tokenization:

- smallest unit of text that is turned into a number ⇒ Token
- characters, words, subwords, groups of words/subwords, etc.
- Models have their own type of tokenization
    - Byte level BPE ⇒ GPT-2
    - WordPiece ⇒ BERT
        - staring letter then rest have ##
        - maintain another list with no duplicates

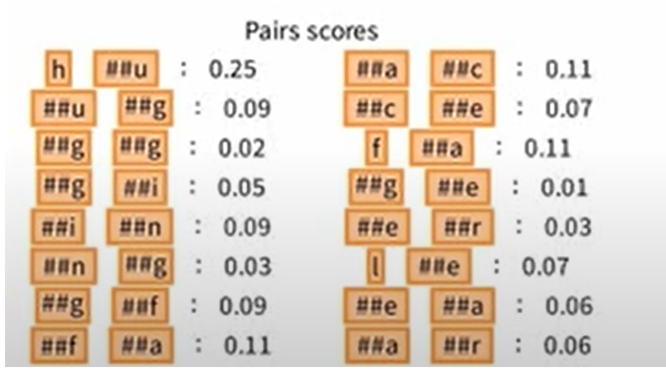




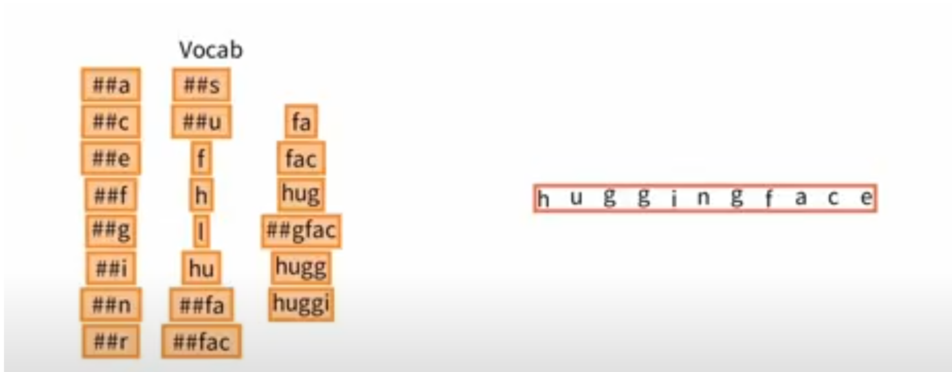
        - Calculate **score for pairs**
        -

$$score = \frac{freqofpair}{(freq1stelem) * (freq2ndelem)}$$

- score of `h` `##u` = 4/4*4 = 0.25
- highest pair score = hu (0.25) ⇒ Goes in Vocab



- Now combine `hu` in the split. Now find the next set of most common pair score.
- `##fa` ⇒ `##fac` ⇒ `f##a`
-



- find biggest sequence in vocab
- huggi n gfac e ⇒ this is how you tokenize

◦ SentencePiece or Unigram ⇒ multilingual models

```
from transformers import AutoTokenizer
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
```

```
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokens = tokenizer.tokenize(text) #different from tokenizer(text
print(tokens)
```

2. Encoding

- Text ⇒ numbers (input IDs) ⇒ tensor
  - does it by referencing the vocabulary on which the model is trained
  - Since each model has it's own vocab, it is necessary that we use the same tokenizing method as when it was pretrained

3. Decoding

- reverse of encoding
- numbers ⇒ text
-

```
decoded = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110,
print(decoded)
```

4. Batching Padding and Truncation

Batching:

- to send multiple sentences at once

but sentences are of difference sizes.. to make them equal we pad them

Padding

- to make sure all the sentences are of the same length

- adds a padding token
- `padding = 'longest'`, will pad the sequence up to the maximum sequence length in the given batch of sequences.
- `padding = 'max_length'`, restricts the maximum length allowed to that of the model. In case of BERT this is 512.

```
model_inputs_padded_3 = tokenizer(sequences,padding = "max_lengt
print(model_inputs_padded_3)
```

Truncation

if one sentence is tooooo long then we truncate it

Encoder- Decoder Models

- Encoder: randomly masks ⇒ context understanding, named entity recognition
- Decoder: attention layer only reads before the mask ⇒ Good for generation of texts
- Encoder+decoder:
    - Good at generating sequence (decoder) while understanding the context (encoder)
    - eg: T5, BART, Marian
- T5: Text to Text Transfer Transform
- 15% of words replaced with placeholder token ⇒ corrupt text
- This goes to encoder.. placeholder token is masked ⇒ decoder predicts them using uncorrupt text as target

`tokens_input = tokenizer.encode("summarize: "+text, max_length=512, truncation=True)`

Decoder Models

- ability to perform autoregressive tasks.

- output of one timestep can be input of subsequent time step: Make it coherent to the text
- The brown cat … ⇒ token gen: "jumps"
- next input ⇒ The brown cat jumps ….

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("distilgpt2")
model = GPT2LMHeadModel.from_pretrained("distilgpt2")
```

```
#encoder
text = "Let's turn this string into a PyTorch tensor of tokens.'

pt_tokens = tokenizer.encode(text, return_tensors = 'pt')
print(pt_tokens)
```

```
#decoder
list_tokens = [1532, 345, 821, 3555, 428, 11, 345, 875, 9043, 5(

decoded_tokens = tokenizer.decode(list_tokens)
print(decoded_tokens)
```

`If you're reading this, you decoded me!`

```
 prompt = "Hello, my name is"

 inputs = tokenizer.encode(prompt, return_tensors="pt")

 output = model.generate(inputs, max_length=75, num_beams = 1, (

 print(tokenizer.decode(output[0]))
```

- `pad_token_id` : to avoid warning ⇒ tells model to use the end of sequence

Temperature of model

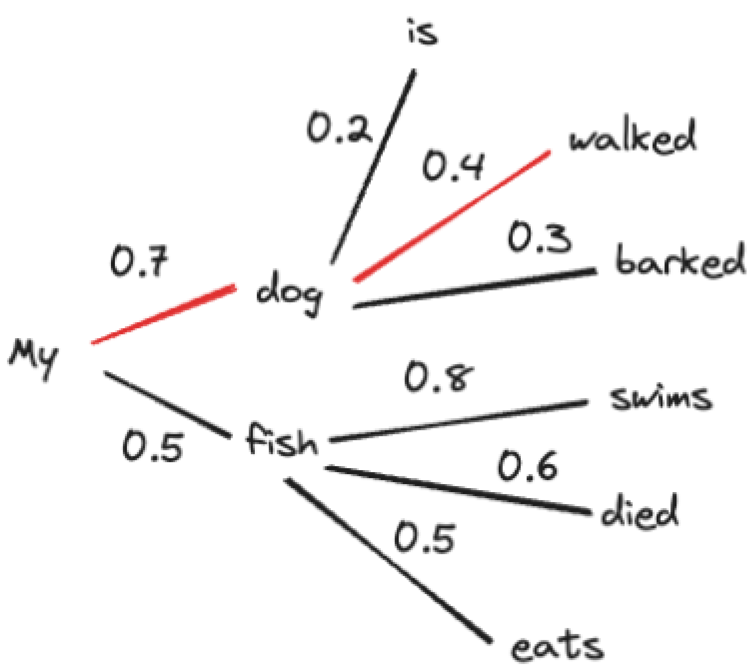
- warmer(higher): less predictable, more creative
- cooler(lower): more likely output

```
output = model.generate(input, max_length=75,num_return_sequence
```

Token Selection Strategy for GPT-2

Greedy Search :

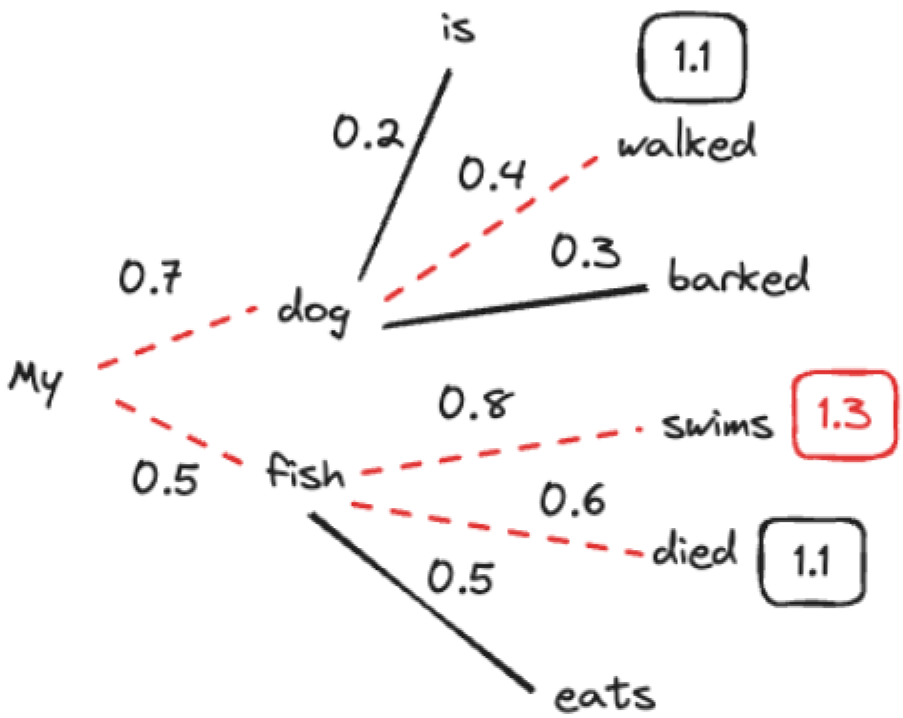
- Model selects the most likely next node



My⇒ Dog ⇒ Walked
- problem: swims has highest prob, but is ignored

Beam Search

- Calculates the probabilities of each path(beams) and the highest prob path is taken
- `num_beams` parameter tells the "red lines"



my fish swims

N-gram Penalty

- Just to prevent getting in the loop
    - Prevents by not allowing repetition of the n-sequence

Sampling

- sampling, which chooses the next token at random from among a collection of likely next tokens

Summary:

Parameters of `model.generate()`

```
sample_outputs = model.generate(
    inputs, #encoded input tokenizer.encode(prompt, return_tenso
    no_repeat_ngram_size=2, #n-gram penalty (avoids repetition)
    max_new_tokens=40,
    pad_token_id=tokenizer.eos_token_id, # to prevent error mess
    do_sample = True,
    temperature = 0.6,
    top_k = 50 #from 50 most likely

)

print(tokenizer.decode(sample_outputs[0]))
```

Carbon Emissions

- carbon emission calculator: https://mlco2.github.io/impact/#compute
- Code carbon: https://codecarbon.io/
    -

    ```
    from codecarbon import EmissionsTracker
    tracker = EmissionsTracker()
    tracker.start()
    # Code whose carbon you want to track here
    tracker.stop()
    ```

- HuggingFace
    -

```
from huggingface_hub import HfApi

api = HfApi()
models = api.list_models(emissions_thresholds=(None, 100), ca
len(models) # in emissions_thresholds, the first argument is
# >>> 191
```